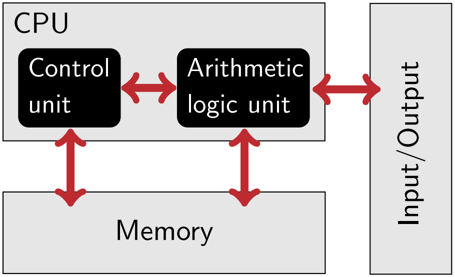
Accelerator-Based Programming

Background on computer architecture

Data parallelism with SIMD

Computer architecture model: stored-program concept

- Stored-program concept
- Conceived by A. Turing (1936)
- Instructions are bit streams stored as *data* in memory
- Instructions read and executed by **control unit**



- Separate arithmetic/logic unit does actual computations
- Outcome of computation can redirect instruction flow (e.g. “branch”, **if** in programming)
- Programming a computer: Modify instructions in memory, done by a **compiler** translating high-level code (C, Java, Fortran) into machine instructions
- Instructions and data must both be fetched from memory into control and arithmetic units $\rightarrow$ bottleneck

Visualization of program execution

Dense matrix-vector multiplication in C/C++

$$y = Ax$$

A in column-major format

```
for (int i=0; i<M; ++i) {
  y[i] = 0;
  for (int j=0; j<N; ++j)
    y[i] += A[j*M+i] * x[j];
}
```

Inside the inner `j` loop, processor has to do the following actions:

1	Read instruction	Compute offset `o=sizeof(double)*(j*M+i)` to array pointer `A`
2	Read instruction	Load `a_ij=(A+o)` from memory
3	Read instruction	Load `x_j=x+j*sizeof(double)` from memory
4	Read instruction	Multiply `tmp = a_ij * x_j`
5	Read instruction	Add `y_i = y_i + tmp`
6	Read instruction	Increment loop counter `j = j+1`
7	Read instruction	Compare `j < N`
8	jump to back to line 1 if comparison true	
	Instructions	Operations on data
	Control unit	Arithmetic logic unit

Organization of a processor: Instructions vs. data

Control unit (front end)

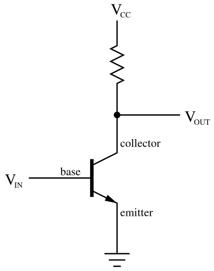
- Machine code is fetched from memory, decoded and execution resources are allocated
- Scheduling of **operations**
  - Some take longer than others
  - Some depend on previous
- Needs to be done (well) in advance of operations
- Branches (`if` statements, loop comparisons) and function calls change instruction flow
- Conditional branches: Jump value depends on data
- Jumps make it difficult to fetch the "next" instruction in advance

Arithmetic logic unit (back end)

- Integer operations (including pointers)
- Floating point operations ("actual work" in HPC)
- Memory operations (load/store)
- Some operations consist of up to 100 000 binary switches (e.g. floating point multiplication) $\rightarrow$ must be broken in several steps $\rightarrow$ FP multiplication takes 3–5 clock cycles on good processors

Electrical construction of a processor

- A CPU (central processing unit; microprocessor) consists of transistors (made of silicon) and wires (typically copper) between them
- A transistor is like a small switch
  - Electric circuit between "emitter" and "collector"
  - Current on the "base" entry of transistor (on/off) determines state at collector (on/off)



- In a processor, $10^9 - 10^{11}$ transistors work together
  - Perform logic operations
  - Volatile storage (SRAM)
- The state on "output" parts of a unit are gathered in regular intervals, at the **clock frequency**
- Typical clock frequencies today: 1.5 – 5 GHz
  - Up to 5 billion steps are possible per second

Two ways to make a processor fast

Increase the clock frequency

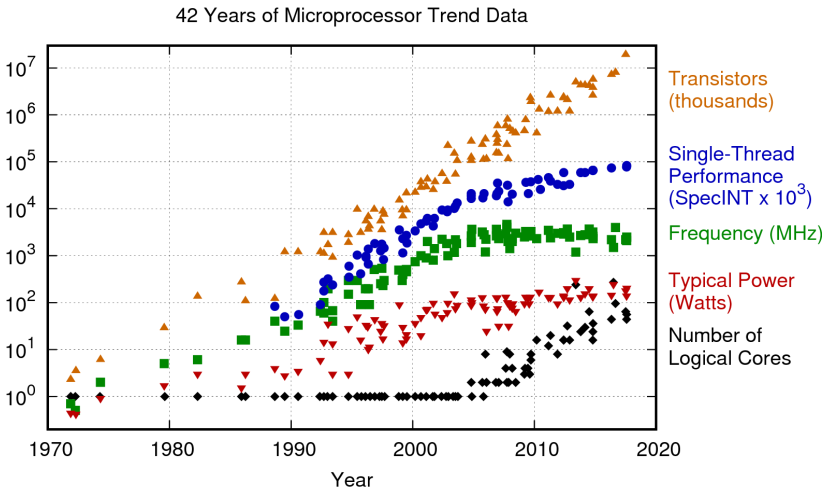
- Dennart scaling: smaller transistors have less delay
- Smaller transistors can be clocked higher
- Dennard scaling broke down around 2004
- Today's transistor have component lengths down to $\sim 5\,\mathrm{nm}$, 10 times smaller than in 2005
- Clock speed limited by the delay of all transistors working together in a clock cycle

Do more work per clock

- Use many transistors at the same time
- Naive thinking: use "complex" instructions that contain more work
- Reality: process several instructions in **parallel**
- Break down long dependency chains of an instruction that limit clock rate into several cycles
- Basic split: Do instruction decoding in the clock cycle before execution $\rightarrow$ **pipelining**

Increasing CPU complexity over time

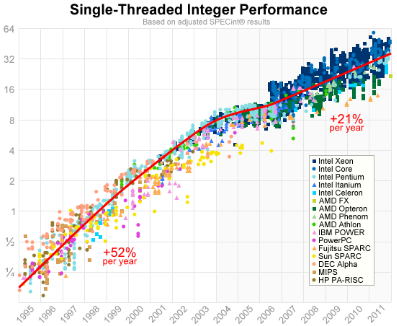
- Moore's "law": Chip complexity doubles every ∼ 18 months (since 1960s)
- Until 2004: Performance doubled every ∼ 18 months



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 2010-2017 by K. Rupp

A view on single-thread performance

Consider the industry-standard benchmark set SPEC CPU (SPECint for integer performance, SPECfp for floating point performance)



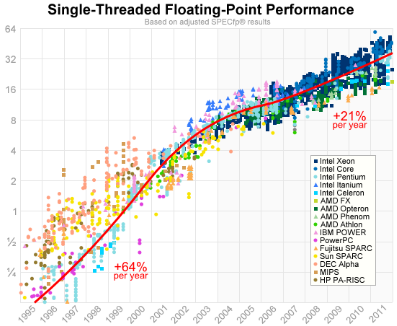


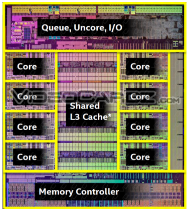
Image source: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

What has improved since 2004: chips are getting more and more cores (parallelism)

Why is frequency not increasing any more?

- **Chip frequency stagnates** around 2.0–5 GHz since 2004
- **Chip power dissipation** goes as $P \sim c_1 V + c_3 f V^2$ for voltage V and frequency f (very simplified)
- Voltage is (in some range) proportional to frequency: $P \sim \tilde{c}_1 f + \tilde{c}_3 f^3$ (more frequency $\rightarrow$ shorter signals must be stronger $\rightarrow$ more voltage)
- Constants c_1 and c_3 are such that the f^3 term dominates above $\sim$ 3 GHz
- More than 150–300 W per chip are hard to transport away
- Within one clock cycle at 3 GHz, signals travel $<$ 10 cm (speed of light)
- To get clear signals within 5 mm, frequency cannot (physically) exceed 10–30 GHz

Intel Haswell 8-core layout ($17.6 \times 20.2\text{mm}^2$ chip size, 22 nm feature size in transistors, 2014, similar to CPU on Snowy)



More parallelism!

Hard to further increase frequency → use more transistors to

(i) Do **more work at the same frequency** within a core

- Pipelined functional units
- Superscalar architecture

  } instruction-level parallelism
- **Single instruction multiple data** (SIMD, vectorization): data-level parallelism
- Out-of-order execution
- Larger caches
- Measured as: Instructions per cycle (IPC) or cycles per instructions (CPI)
- Long tradition in computer architecture

(ii) Increase the **number of cores**

- Multi-core and many-core systems, increasingly used (modern Intel CPUs have up to 40 cores, AMD CPUs up to 64 cores), Intel Xeon Phi and NVIDIA/AMD GPUs have 12–108 “cores”
- Accelerator programming: **Reduce sophistication, increase parallelism**
- Challenging part: how do the cores talk to each other

Doing things in parallel: pipelining

- Break down complicated/expensive operations into several parts
- Basic example illustrated on the right
  - Fetch an instruction
  - Decode the instruction
  - Execution
  - Write the result back
- Break down "execution" phase, e.g. floating point addition:
  - Make exponents equal
  - Add significants
  - Normalize and round

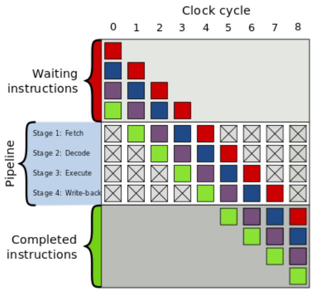


Image source: Wikipedia

- Each pipeline stage can take new data in every clock cycle, even though old result not finished yet
- Make pipeline stages similarly expensive to maximize frequency
- Typical pipeline lengths today: 12–20 stages

Pipelining: latency versus throughput

- A fully pipelined execution unit can deliver a result in every clock cycle, but it might take several clock cycle to produce that result
- Must distinguish latency and throughput
  - **Latency**: Time (in clock cycles) it takes from starting an operation until it is finished
  - **Throughput**: How many results produced per clock cycle
- To have the same unit (clock cycles) for both latency and throughput, one often speaks of the reciprocal throughput
- Example: fully pipelined floating point addition
  - Latency is 3 clock cycles in execution regime (disregarding time in control unit)
  - The reciprocal throughput is 1 clock cycle, meaning 1 result produced per clock cycle
- Example: floating point division, partly pipelined on modern CPUs
  - Latency 14 clock cycles in execution regime
  - Reciprocal throughput 4 clock cycles (i.e., 1 division every 4 clock cycles)

Implication of pipelining: Need independent instructions

- To keep the pipeline fed, must ensure that new operands flow into the execution units in every clock cycle
- Problem arises in case some operand (input data) has not yet arrived from memory
  - Pipeline will have to wait until all data available
  - Pipeline stall
- Similarly when starting a new block of operations → no result produced during *wind-up* phase of pipeline
- Pipeline stalls also when input operands depend on outcome of pipeline

```
double sum = 0;
for (int i=0; i<N; ++i)
  sum += x[i];
```

Floating point addition has latency of 4 cycles due to pipeline → can finish one iteration at most every four cycles

Doing things in parallel: superscalarity

Superscalar = produce more than one "result" per clock cycle

- Multiple instructions fetched and decoded concurrently (typical today: 4–8)
- Multiple integer execution units (address calculations, loop counter increment; add, mult, shift, mask)
  - Example: Many modern CPUs can do 4 integer additions per clock cycle (e.g. latency 1 cycle, reciprocal throughput 0.25 cycles)
- Multiple floating point units
  - Example (2010): one addition and one multiplication unit
  - Example (today): two fused multiply-add units per core, i.e., two operations of form $c = c \pm a * b$ done per clock $\rightarrow$ 4 floating point operations per cycle (2 additions, 2 multiplications)
- Memory loads and stores execute in parallel to arithmetic operations
- Multiple load or store execution units
- Granularity of superscalarity: **execution units**
- Typical value today: 6–10 execution units

Keeping the pipeline fed: out-of-order execution

Code example:

```
double sum1 = 0.;
sum1 += a[0] * b[0];
sum1 += a[1] * b[1];
sum1 += a[2] * b[2];
double sum2 = 0.;
sum2 += c[0] * d[0];
sum2 += c[1] * d[1];
sum2 += c[2] * d[2];
if (sum2 > sum1)
  return sum2;
else
  return sum1;
```

Compute two 3D dot products and return larger result

Assume machine code exactly follows the above statements

- Assume elements from `a`, `b` and `c`, `d` loaded from memory in parallel with 4 cycle latency
- Assume multiply and add have 4 cycle latency each
- Naive execution takes $6 \times (4[\text{load}] + 4[\text{multiply}] + 4[\text{add}])$ cycles up to `if`
- Solution: **execute operations out-of-order** to do things in parallel
  - Start loads `a[1]` and `b[1]` before multiply `a[0] * b[0]`
  - All multiplications
  - Additions `sum1`, `sum2`
  - Real latency $4[\text{load}] + 4[\text{multiply}] + 3 \times 4[\text{add}]$

Keeping the pipeline fed: out-of-order execution

- Instruction reordering could be done by compiler, but much more effective when done by **out-of-order execution** (OOOE) in CPU
  - Compiler limited by being able to prove validity of certain rearrangements depending on conditions, memory addresses, etc.
  - Compiler rarely knows which data comes from cache (wait little) or from main memory (wait longer)
  - When executing, many more facts about execution are known
- OOOE window on modern CPUs: up to ∼500 instructions
  - instruction decode is often tens of instructions ahead of execution
- Limit to OOOE: Conditional executions (`if` statements, increment + check in `for` loops), called **conditional branches**

Keeping the pipeline fed: speculative execution

- Out-of-order execution limited by conditional branches
  - branch ratio 1:5 – 1:20 usual in code, i.e., one instruction out of five is a branch instruction
  - delay from memory can be more than 100 cycles
  - how to break the barrier and find useful instructions?
- CPU *guesses* on the outcome of the branch and **speculatively** executes the predicted case
  - for example, execute `true` part of `if` statement
- Verify the prediction few clock cycles later
  - correct guess: the work was useful
  - wrong guess: must play back the state and replace by work on the right branch
- In case of a **branch mispredict**, all content in pipeline is lost and it takes often several cycles to fill things up again
- CPU uses sophisticated algorithms to maximize the correct guesses (branch prediction unit)
  - Typical branch prediction accuracy above 95%, often 99%
  - But wrong decision on branchy code leads to $> 10\times$ slowdown

Summary of sophisticated CPU technologies

- Pipelining used to simplify hardware $\rightarrow$ higher frequencies
- Superscalar execution allows to execute several operations in parallel
    - reflected in fetch & decode (control unit), execute, and retire (control unit)
    - also for arithmetic logic unit several operations at once
- Each execution unit is responsible for several operations
- Out-of-order execution runs ahead of the slowest operation
- Branch prediction aims to fetch and decode the right instructions in case of branches
- Waits for operands from memory, mispredicted branches, instruction streams that do not utilize all execution units all lead to slowdown

Feeding the operations on a processor: Registers

- Arithmetic operations have 1–3 input arguments and 1 output argument
  - Example: Floating point addition, $x_3 = x_1 + x_2$
  - Assembly instruction on x86-64 (Intel, AMD) with VEX encoding: `vaddsd %xmm1, %xmm2, %xmm3`
  - Assembly instruction with ARM64: `fadd d3, d1, d2`
- Arguments to all operations held in **registers**
- Registers accessed without delay ("instruction latency")
  - Performance of register file access = capabilities of the execution resources
- Intel/AMD CPUs have 16 floating point registers and 16 integer registers (32 floating point registers with AVX-512)
- ARM64 instruction set has 32 floating point registers and 32 integer registers
- Registers get assigned when compiler translates user code (e.g. C++) into assembly code (machine code)
- Online assembly code generator `https://godbolt.org`

Simplified view of memory on a CPU

- Registers transparent from a programmer's perspective, their content only gets visible when stored in memory
- To perform arithmetic work, a program needs to first load data from memory into registers
- Once done, register content gets written back to memory
- All data of a program held in **main memory (RAM)** (random-access memory)
  - RAM is large (e.g. 64–512+ GB on typical HPC machines)
  - RAM physically away from CPU cores (memory modules)
  - RAM access incurs significant latency due to distance: $40 - 100$ns, i.e., up to 400 clock cycles
  - Memory modules accessed via connections (pin)
  - Several memory modules in parallel
  - Memory **bandwidth** limited by the number of **memory channels** and the bus width ($\approx$ # pins)
  - Physics: 1 capacitor + 1 transistor, DRAM (dynamic RAM)
  - RAM memory is **volatile** (today), i.e., content lost when off
- Programs interact explicitly (read/write) with non-volatile **storage**

Performance register access versus memory access

- Execution units need to be fed with up to 3 input arguments (x_1, x_2, x_3), result of operation needs to be transferred back
- Example: Arithmetic units of 48 cores of Intel Skylake (2017) process 66 TB/s of data (3 input, 1 output), but main memory only provides 256 GB/s in theory
- **Memory wall**: transfer from memory becomes slower over time
- Affects both GPUs and CPUs

Arithmetic throughput versus main (RAM) memory bandwidth since 2006

$$\frac{\text{flop}}{\text{byte}} = \frac{\text{arithmetic tp [GFlop / s]}}{\text{memory bw [GB / s]}}$$

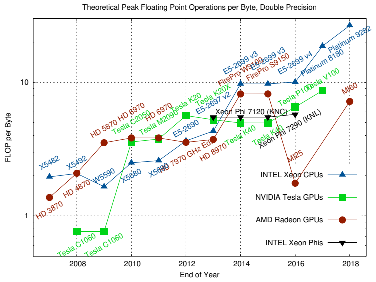


Image source: https://github.com/karlrupp/cpu-gpu-mic-comparison

Solution to memory bandwidth & latency problem: caches

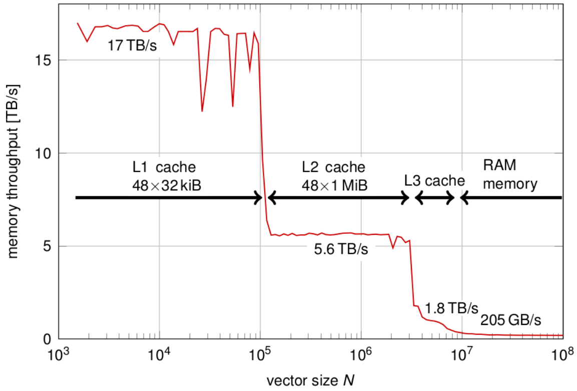
- Sit between registers and main memory
- Smaller than RAM, but (much) faster
- **Hierarchy of caches** to balance size versus speed
- Typical layout today: 3 levels of data caches, for example on Intel server processors
  - **Level 1** (L1) data cache: 32 kiB / core
    - Best throughput 192 byte / cycle (2×64 byte read, 1×64 byte write) via load/store execution units
    - Latency: 4–5 clock cycles
  - **Level 2** (L2) cache: 1 MiB / core
    - Theoretical throughput 64 byte / cycle
    - Result from L2 goes into L1 cache
    - Latency: 14 clock cycles
    - L2 cache includes data of L1 cache
  - **Level 3** (L3) cache: 1.375 MiB / core
    - Shared across all cores
    - Theoretical throughput up to 30 byte / cycle
    - Latency: 50–70 clock cycles
    - L3 cache is "non-inclusive victim cache", i.e., does not necessarily replicate data from L2 cache; victim = L3 is only filled by what falls out from L2 cache

Visualization of Memory Bandwidth

Benchmark:

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

- daxpy code
- 48 CPU cores
- Vary vector size between 1536 and 10^8
- Repeat sufficiently often to get good timings



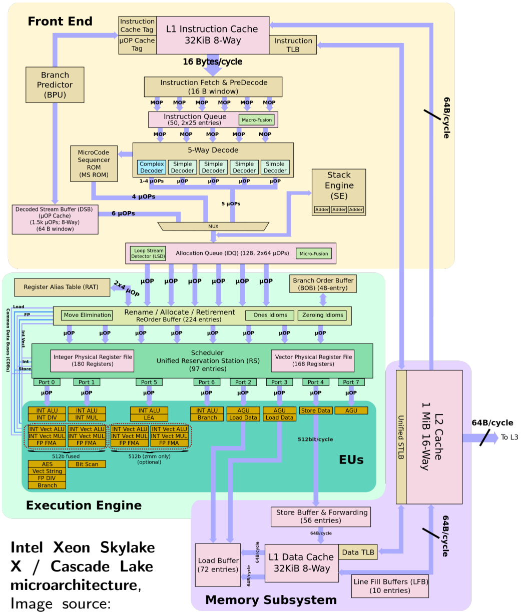
Experiment: Record time t and compute throughput as $N \times 24[\text{Byte}]/t[\text{s}]$

Recall: Bandwidth between execution units and registers 66 TB/s!

Summary CPU capabilities: typical values

- Typical instruction throughput in real code: IPC 0.5–3
- Best IPC of current processors
  - Apple A14 “Firestorm” processor (iPhone 12 from 2020)
  - A14 IPC approx. 30–50% ahead of best Intel and AMD desktop/laptop CPU
  - A14 IPC approx. 35–60% ahead of Intel and AMD server processors
- Intel and AMD processors run at higher frequencies
  - Desktop CPUs (Intel, AMD) run at 4.2–5.3 GHz
  - Server CPUs (Intel, AMD) run at 2.0–4.5 GHz (more cores $\rightarrow$ less frequency)
  - Mobile phone chips run at 2.0–3.1 GHz (Apple A14 “Firestorm”: 3.0 GHz)
  - GPUs run at 1.3–2.0 GHz
- Final performance (frequency $\times$ IPC) of best desktop (AMD Zen 3) for serial code very similar to Apple A14, depending on benchmark

CPU organization: Block diagram of Intel Xeon Skylake core


Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (50, 2x25 entries)
Macro-Fusion
5-Way Decode
Complex Decoder
Simple Decoder
MicroCode Sequencer ROM (MS ROM)
1-4 µOPs
µOP
4 µOPs
5 µOPs
6 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (64 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (128, 2x64 µOPs)
Micro-Fusion
64B/cycle
Register Alias Table (RAT)
2x4 µOP
Branch Order Buffer (BOB) (48-entry)
Move Elimination
Rename / Allocate / Retirement ReOrder Buffer (224 entries)
Ones Idioms
Zeroing Idioms
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Integer Physical Register File (180 Registers)
Scheduler Unified Reservation Station (RS) (97 entries)
Vector Physical Register File (168 Registers)
Port 0
Port 1
Port 5
Port 6
Port 2
Port 3
Port 4
Port 7
INT ALU
INT DIV
INT MUL
LEA
Branch
AGU
Load Data
Store Data
INT Vect ALU
INT Vect MUL
FP FMA
512b fused
512b (2mm only) (optional)
512bit/cycle
AES
Vect String
FP DIV
Bit Scan
EUs
Execution Engine
Unified STLB
L2 Cache 1 MiB 16-Way
64B/cycle
To L3
Store Buffer & Forwarding (56 entries)
64B/cycle
Load Buffer (72 entries)
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

Intel Xeon Skylake X / Cascade Lake microarchitecture, Image source: https://en.wikichip.org

- Modern CPUs very complex
- Actual work done in orange blocks, mostly in “INT ALU” and “FP FMA”
- Huge logic is necessary to feed execution units with operands
- CPU **registers** hold operands to be accessed by instructions
- Data must come from memory into registers
- Typical CPUs have 16–64 registers for each integers and floating point

Organization of Multi-Core CPUs

- Modern laptop/desktop CPUs have 2–16 cores
- Server processors share most of architecture + some specific features and more cores (≤ 96)
- Core = able to execute independent programs autonomously
- High-speed connection between cores on the same silicon die
- Similar to wires within core
- Core = frontend + backend + L1/L2 cache + slice of L3 cache ("LLC")
- Memory controller and I/O outside of cores on the same die
- Intel/AMD systems use 2 CPU sockets per node, the Fujitsu A64FX in Fugaku 1 CPU per node

Example: The Intel Xeon Skylake manufactured on a silicon die of 28 cores

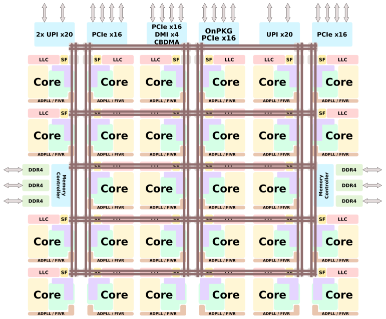


Image source: https://en.wikichip.org

First accelerator-type solution: SIMD

Data-level parallelism by SIMD

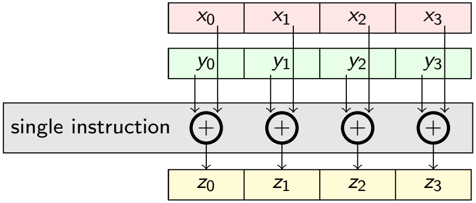
Machinery to schedule and control instructions is expensive compared to actual operations

- Scalar code: 50–80% of energy consumption due to instruction control, only 10–30% for arithmetic work
- Balance this by **vectorization**: Perform operations for several data items at once
- Data-level parallelism, concept called single-instruction/multiple data (SIMD)
- Multiple entries in SIMD array called **lanes**

```
for (int i=0; i<20; ++i)
  z[i] = x[i] + y[i];
```

$\downarrow$

```
for (int i=0; i<20; i+=4)
  z[i:i+3] = x[i:i+3] + y[i:i+3];
```



Workflow for SIMD operations

General workflow when working with SIMD (vectors) inside a CPU

- CPU treats **consecutive** elements of an array like a single entity
- A **single instruction** to process **all elements** at once, in example of last slide:
  - Instruction 1: Load 4 elements from array `x`, starting at index `i`
  - Instruction 2: Load 4 elements from array `y`, starting at index `i`
  - Instruction 3: Perform addition of 4 elements
  - Instruction 4: Store result
- Operations correspond to a **vector instruction**
- We use a more powerful instruction, **exactly** like a data type with more bits
- Inherently parallel operation: Data parallelism

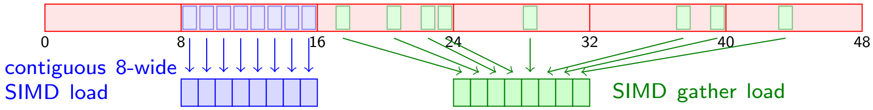
Vectorization available today

- Today's standards: 4–8 data items in double precision (256–512 bit)
  - Intel Sandy Bridge, Ivy Bridge, Haswell, Broadwell, Skylake client: 256 bit / 4 doubles with AVX
  - Intel Skylake, Cascade Lake, Ice Lake Server: 512 bit / 8 doubles with AVX-512
  - AMD Zen/Ryzen 1–3: 256 bit / 4 doubles (AVX)
  - Other architectures also have SIMD, e.g. ARM Neon for small-scale chips (128 bit for floats) or the Fujitsu A64FX chip based on the ARM architecture provides 512-bit SIMD through the so-called SVE (scalable vector extensions)
- Only effective for an array of data processed by same instruction
- Note: SIMD-heavy code will cause the processor to **clock lower** due to increased energy consumption!
  - Example: Xeon Gold 6230 with all 20 cores are loaded: 2.8 GHz no/light vectorization, 2.4 GHz AVX-2 heavy/AVX-512 light vectorization, 2.0 GHz for AVX-512 heavy vectorization
  - Check frequency table at `https://en.wikichip.org/wiki/intel/xeon_gold/6230`

What gets vectorized? And how?

- Vectorization available for basic operations such as `add`, `sub`, `mul`, `div`, `sqrt`, `fmadd`, `mov` (load/store)
- Compiler at high optimization levels will look for opportunities (in loop optimization passes and for straight-line code)
- Vectorization can also perform operations only on some lanes (so-called **masked operations**)
- My experience: Most interesting loops typically not vectorized
  - If there is the slightest possibility of data overlap (aliasing), mis-alignment or re-ordering of floating point operations, the compiler will **not vectorize**
  - Happens often in C++ (pointers can cause aliasing)
- Can also write explicitly vectorized code (intrinsics)
- Try to use vectorized libraries (BLAS + LAPACK, etc.)

Data access with SIMD



- Two main possibilities for data access
  - Contiguous (coalesced) access: Array elements loaded/stored from adjacent places in memory
  - Unstructured access based on index offsets: SIMD gather (load) or SIMD scatter (store)
- Contiguous access is fast (almost like scalar load), gather/scatter access (much) slower, e.g. around $4\times$ for 8-wide SIMD

Example of data access patterns

Example:

```
for (int i=0; i<32; ++i)
  z[i] = x[2*i] + y[20-i];
```

- How to load `x` vector?
  - Many SIMD instruction sets do not have strided loads
  - Typically best: perform two loads and reorder elements within lanes
  - Called permute/shuffle instructions
- How to load `y` vector?
  - SIMD load and permute
- Vectorization includes permute operations within a lane, called shuffle or swizzle operations
- In general, exchange between lanes (permute, shuffle) has associated cost of latency and throughput

Recognize a vectorized loop

Benchmark case: STREAM triad in single precision

```
for (unsigned int i=0; i<N; ++i)
  z[i] = a * x[i] + y[i];
```

Non-vectorized code
`g++ -O2 -march=haswell stream_triad.cc`

```
// short prologue
.L3:
vmovss   (%rsi,%rax,4), %xmm1
vmovss   (%rdx,%rax,4), %xmm2
vfmadd213ss %xmm2, %xmm0, %xmm1
vmovss   %xmm1, (%rcx,%rax,4)
incq     %rax
cmpq     %rax, %rdi
jne      .L3
// short epilogue
```

Vectorized code
`g++ -O3 -march=haswell stream_triad.cc`

```
// long prologue
.L5:
vmovups (%rsi,%rax), %ymm1
vmovups (%rdi,%rax), %ymm3
vfmadd213ps %ymm3, %ymm2, %ymm1
vmovups %ymm1, (%rdx,%rax)
addq    $32, %rax
cmpq    %rcx, %rax
jne     .L5
// long epilogue
```

Decipher vectorization on x86-64

- Vectorized operations indicated by “packed” operation (second to last letter `p` in SIMD code) and the register name
  - `xmm0`–`xmm15`: 128 bit / 4 floats (single-precision)
  - `ymm0`–`ymm15`: 256 bit / 8 floats
  - `zmm0`–`zmm31`: 512 bit / 16 floats
- AVX and newer include leading `v` in command (VEX/EVEX encoding)

Example: Floating point multiplication, `mulss`, `mulps`, `vmulps`

	`mul`	`s`	`s`	`%xmm0, %xmm1`
basic x86-64	multiply	scalar	single	64 bit
	`mul`	`p`	`s`	`%xmm0, %xmm1`
basic x86-64	multiply	packed	single	128 bit
`v`	`mul`	`p`	`d`	`%xmm0, %xmm1, %xmm1`
VEX-encoded	multiply	packed	single	128 bit
`v`	`mul`	`p`	`d`	`%ymm0, %ymm1, %ymm1`
VEX-encoded	multiply	packed	single	256 bit
`v`	`mul`	`p`	`d`	`%zmm0, %zmm1, %zmm1`
VEX-encoded	multiply	packed	single	512 bit

Alternative vectorization strategy: intrinsics

- Vectorization possibilities for compiler often limited
  - Floating point associativity rules prohibit vectorization of sum in the dot product without `-ffast-math` flag in compiler
  - Unclear data dependencies or aliasing for compiler
- Alternative is to explicitly write vectorized instructions:

```
for (int i=0; i<20; i+=4)
  z[i:i+3] = a * x[i:i+3] + y[i:i+3];
```

in terms of the vector registers:

```
reg3 = broadcast(a); // duplicate a in all register lanes
for (int i=0; i<20; i+=4)
  {
    reg1 = load_range(x+i, x+i+4);
    reg2 = load_range(y+i, y+i+4);
    reg4 = add(multiply(reg3, reg1), reg2);
    store_range(reg4, z+i, z+i+4);
  }
```

- Coding in this style supported by compilers and special `vector` data types $\rightarrow$ intrinsics

Structure of intrinsics

- SIMD data types in x86-64 are called (for all compilers)
  - `__m128d`: double precision, 128 bit, i.e., 2 doubles
  - `__m256d`: double precision, 256 bit, i.e., 4 doubles
  - `__m512d`: double precision, 512 bit, i.e., 8 doubles
  - `__m512`: single precision, 512 bit, i.e., 16 floats
  - `__m512i`: long long integer, 512 bit, i.e., 8 integers
  - ...
- Operations between intrinsics have particular name, e.g.:

  ```
  inline __m512 _mm512_mul_ps(__m512 a, __m512 b);
  ```
- Different compilers implement vector types differently
  - Example of gcc:

    ```
    typedef float __m512 __attribute__((__vector_size__(64)));
    ```
- Codes with intrinsics not portable – must hardcode the length of the vector, which depends on hardware support
- Solution: Wrapper class in C++
- In `vectorization.h` we call it `VectorizedArray`
  - Centralize decision about available code in a single place (wrapper class), use `VectorizedArray` in many places
  - Overload arithmetic operators `+,-,*,/` for simpler use

The VectorizedArray class from vectorization.h

```
// Generic class implementation, just wraps around generic 'Number' type
template <typename Number>
struct VectorizedArray {
  static const unsigned int n_array_elements = 1; // no vectorization
  Number data;
  ...
};

#if defined(__AVX512F__)
// Specialization of VectorizedArray class for float and AVX-512.
template <> struct VectorizedArray<float> {
public:
  static const unsigned int n_array_elements = 16;
  __m512 data;
  ...
};

#elif defined(__AVX__)
// Specialization of VectorizedArray class for float and AVX.
template <> struct VectorizedArray<float> {
public:
  static const unsigned int n_array_elements = 8;
  __m256 data;
  ...
};

#elif defined(__SSE2__)
// Specialization of VectorizedArray class for float and SSE2.
template <> struct VectorizedArray<float> {
public:
  static const unsigned int n_array_elements = 4;
  __m128 data;
  ...
};
#endif
```

Operator overloading for `VectorizedArray`

```
template <> struct VectorizedArray<float> {
  ...
  VectorizedArray & operator += (const VectorizedArray &a) {
    data = _mm512_add_ps(data, a.data); }
  VectorizedArray & operator -= (const VectorizedArray &a) {
    data = _mm512_sub_ps(data, a.data); }
  VectorizedArray & operator *= (const VectorizedArray &a) {
    data = _mm512_mul_ps(data, a.data); }
  ...
};

template <typename Number>
VectorizedArray<Number>
operator + (const VectorizedArray<Number> &u, const VectorizedArray<Number> &v)
{
  VectorizedArray<Number> tmp = u;
  return tmp+=v;
}

template <typename Number>
VectorizedArray<Number>
operator - (const VectorizedArray<Number> &u, const VectorizedArray<Number> &v)
{
  VectorizedArray<Number> tmp = u;
  return tmp-=v;
}

template <typename Number>
VectorizedArray<Number>
operator * (const VectorizedArray<Number> &u, const VectorizedArray<Number> &v)
{
  VectorizedArray<Number> tmp = u;
  return tmp*=v;
}
...
```

Implementing STREAM triad with `VectorizedArray`

- Original code:

```
for (unsigned int i=0; i<N; ++i)
  z[i] = a * x[i] + y[i];
```

- We must split loop into regular part (divisible by SIMD length) and a remainder; for the regular part we do:

```
const unsigned int N_regular = N / simd_width * simd_width;
for (unsigned int i=0; i<N_regular; i += simd_width) {
    VectorizedArray<float> x_vec, y_vec;
    x_vec.load(x+i);
    y_vec.load(y+i);
    const VectorizedArray<float> z_vec = a * x_vec + y_vec;
    z_vec.store(z+i);
  }
```

- Finally, we need to run a remainder loop for the rest

```
// remainder
for (unsigned int i=N_regular; i<N; ++i)
  z[i] = a*x[i] + y[i];
```

- Code in `stream_triad_simd.cc`

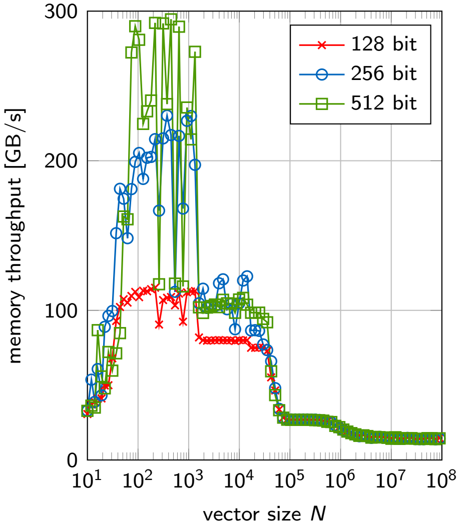
Compilation on Intel Xeon Gold for different instruction sets

Compile code with various compiler settings:

- Standard SSE2 set (128 bit) `g++ -O2 -funroll-loops stream_triad_simd.cc`
- AVX-2 instruction set (256 bit) `g++ -O2 -funroll-loops -march=haswell stream_triad_simd.cc`
- AVX-512 instruction set (512 bit) `g++ -O2 -funroll-loops -march=skylake-avx512 stream_triad_simd.cc`

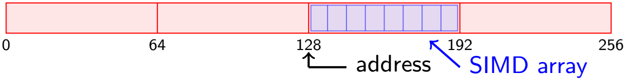
Result: wider vectorization is faster for **small sizes** (→ memory hierarchy)

Result when run on 1 core (3.7 GHz)

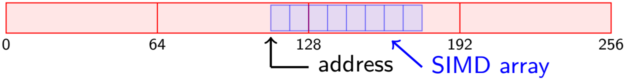


Aligned versus unaligned memory access

- SIMD load and store instruction are influenced by the memory address `VectorizedArray::load(double *address)`
  - Aligned memory access: pointer address in bytes divisible by the size of SIMD vectors in bytes, e.g. 64 for AVX-512



  - Unaligned memory access: pointer address divided by SIMD vector length has remainder



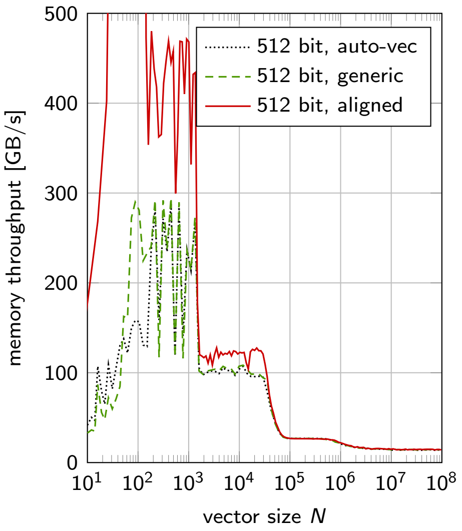
- Aligned access is faster due to the organization of memory
- Unaligned access: CPU must fetch two separate parts and concatenate them $\rightarrow$ performance loss
- Memory addresses from `std::vector` are guaranteed to be aligned by 16 bytes, but not more $\rightarrow$ performance penalty
- Alternatives:
  - Manual adjustment of memory addresses like in our code
  - Use vector classes/allocators which guarantee alignment

Aligned vs unaligned: performance

- Auto-vectorized code from `stream_triad.cc`
- With intrinsics `stream_triad_simd.cc`
- Generate aligned version by allocating slightly more memory and shifting the start address of the vector

Result: The aligned version runs considerably faster up to $N = 1300$ (three vectors: 31.2 kB) and slightly faster up to around $N = 4 \cdot 10^4$ (three vectors: 960 kB)

Result when run on 1 core (3.7 GHz)


512 bit, auto-vec
512 bit, generic
512 bit, aligned
memory throughput [GB/s]
vector size N
10^1
10^2
10^3
10^4
10^5
10^6
10^7
10^8