SeSE course: GPU programming for Machine Learning and Data Processing

CPU and GPU hardware

Martin Kronbichler

Uppsala University

Welcome

- Second course part: **How to scale machine learning models** and other computations
- How do we scale from CPU to GPU?
  - How does it work?
  - What are the limitations?
  - How to implement our own algorithms?
- You can combine this knowledge on development for a single node with knowledge on *production workflows* in a cloud

Final course goal: Work on a project where you try these techniques

About me

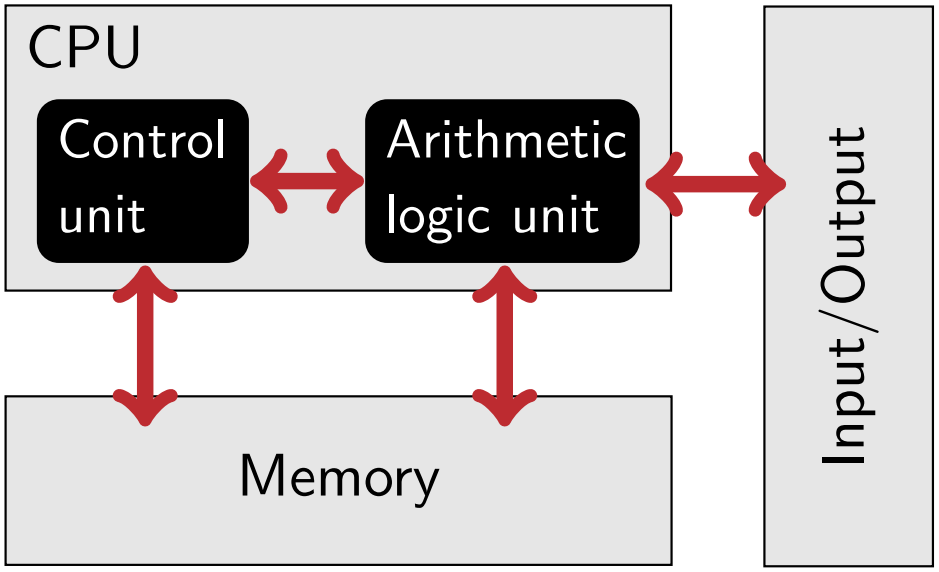
Martin Kronbichler

- PhD in Scientific Computing in 2012
  - Main research on high-performance computing (solving partial differential equations with finite element or finite difference methods)
  - Hardware-adapted methods, exascale algorithms on CPUs and GPUs
  - Run and targeted computations for some of the biggest computers in the world: SuperMUC-NG, Piz Daint, Summit
  - Recent interests include machine learning hardware (matrix multiplication) and specific models
- Worked in both Sweden (2007–2012; since 2021) and Germany (2012–2020)
- Principal developer of finite element library deal.II, github.com/dealii/dealii
- Currently associate professor at Department of Information Technology, Uppsala University
- I am interested in interdisciplinary work, combining mathematical algorithms, computer science and applications

Classical computer architecture – the design of CPUs (central processing units)

Computer architecture model: stored-program concept

- Von Neumann computer: 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)

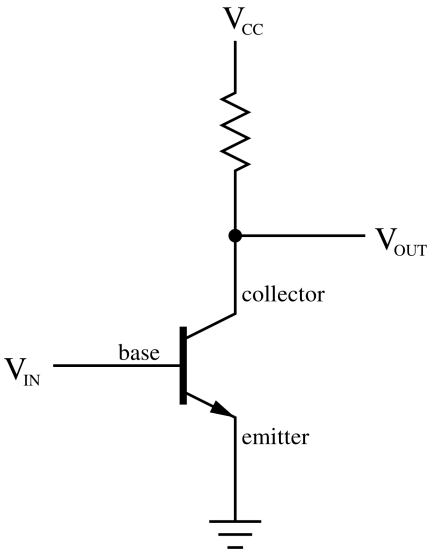
- Integer operations (including pointers)
- Floating point operations ("actual work" in HPC)
- Memory operations (load/store)
- Some operations consist of up to 100 000 binary switches (e.g. floating point multiplication) $\rightarrow$ must be broken in several steps $\rightarrow$ FP multiplication takes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

Model of a CPU

- Above model:
  - One instruction at a time
  - Instruction i has to complete before executing instruction $i + 1$
- Granularity for one instruction: Clock cycle, electrical signal traveling through CPU
- Modern CPUs do not actually work like this
- Do **much** more work within a single clock cycle!

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

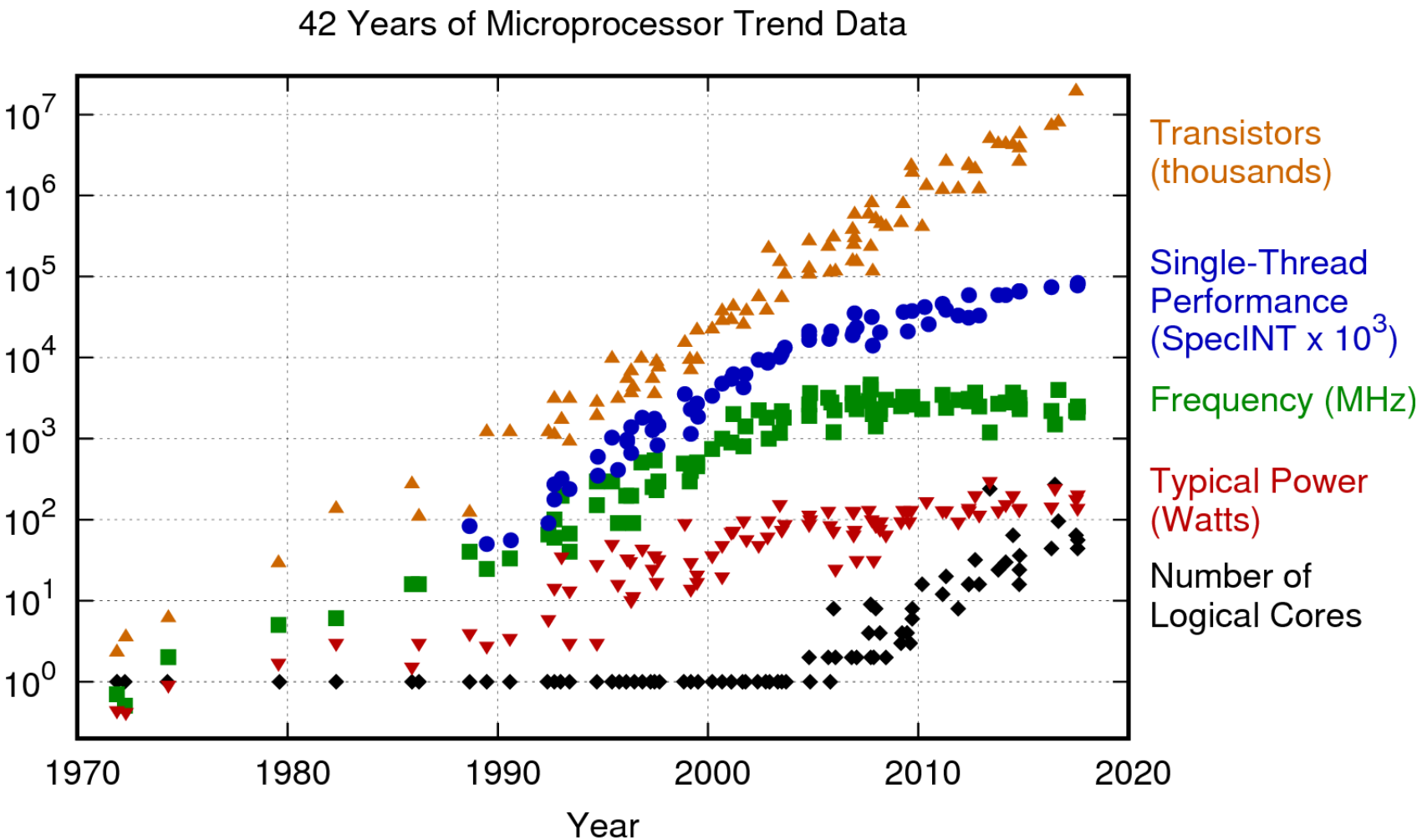
- Dennart scaling: smaller transistors have less delay
- Smaller transistors can be clocked higher
- Dennard scaling broke down around 2004
- Today's transistor have component lengths down to $\sim 5\,\text{nm}$, 10 times smaller than in 2005
- Clock speed limited by the delay of all transistors working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

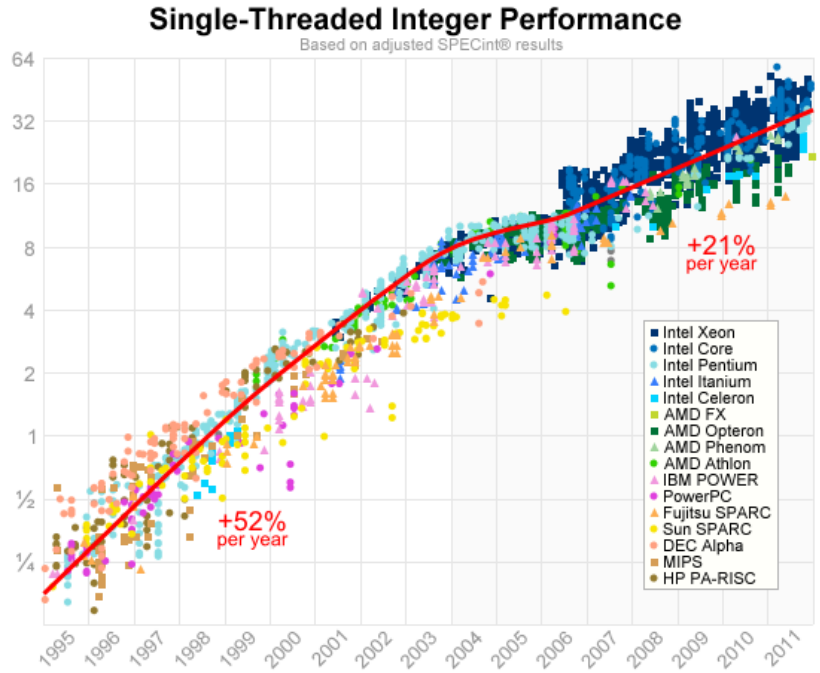
- Moore's "law": Chip complexity doubles every $\sim$ 18 months (since 1960s)
- Until 2004: Performance doubled every $\sim$ 18 months



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)



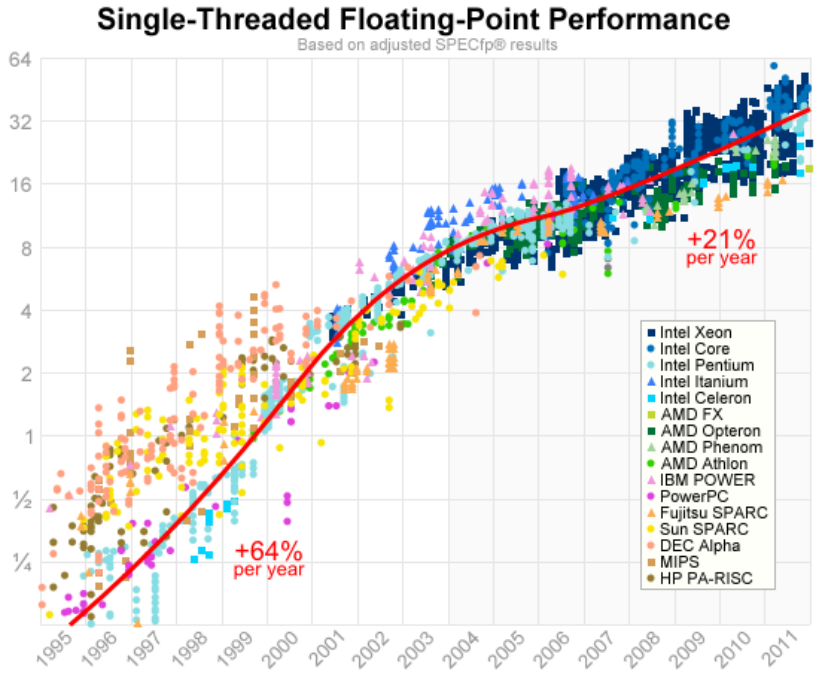


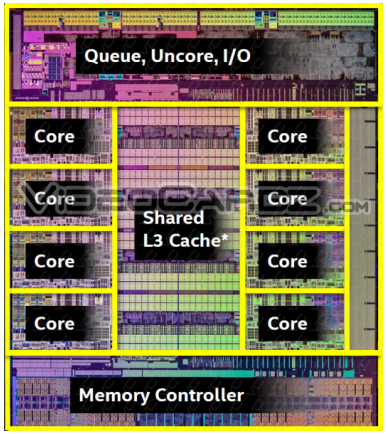
Image source: http://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/

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 ~ 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"
- Accelerators (mainly GPUs): **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

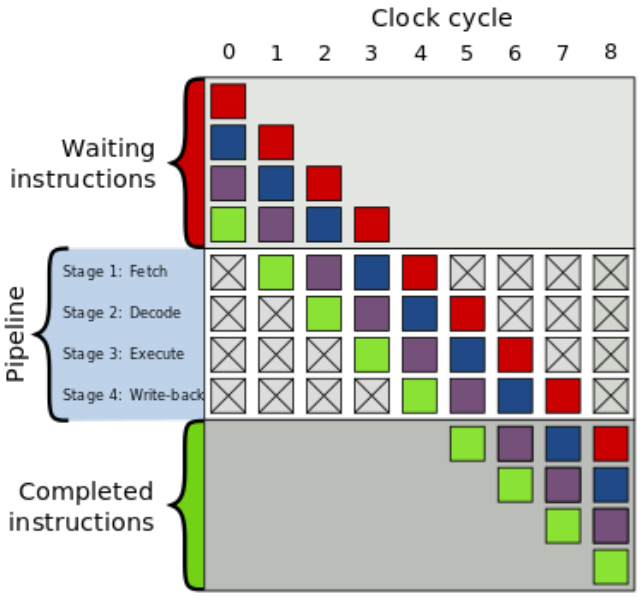


Image source: Wikipedia

- 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

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 $\rightarrow$ 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 $\rightarrow$ 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
- 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, $x_3 = x_1 + x_2$
- 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)

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

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

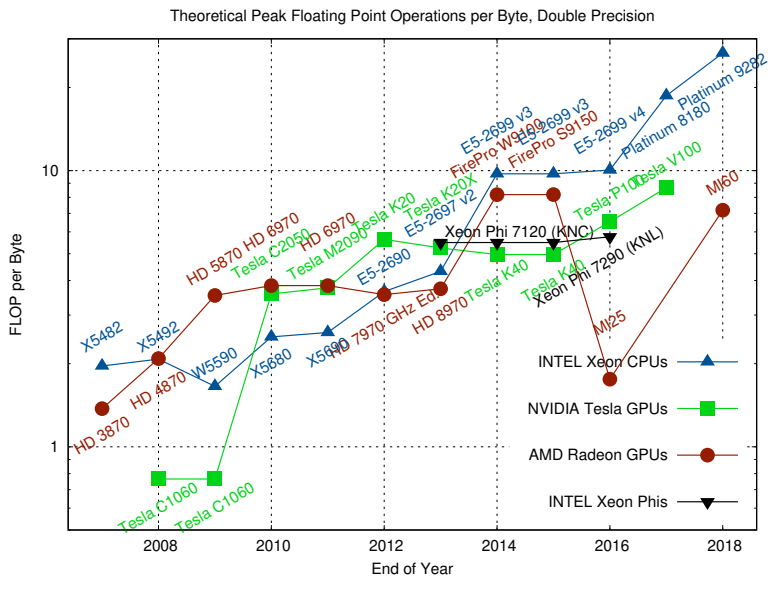


Image source: https://github.com/karlrupp/cpu-gpu-mic-comparison

Solution to memory bandwidth & latency problem: caches

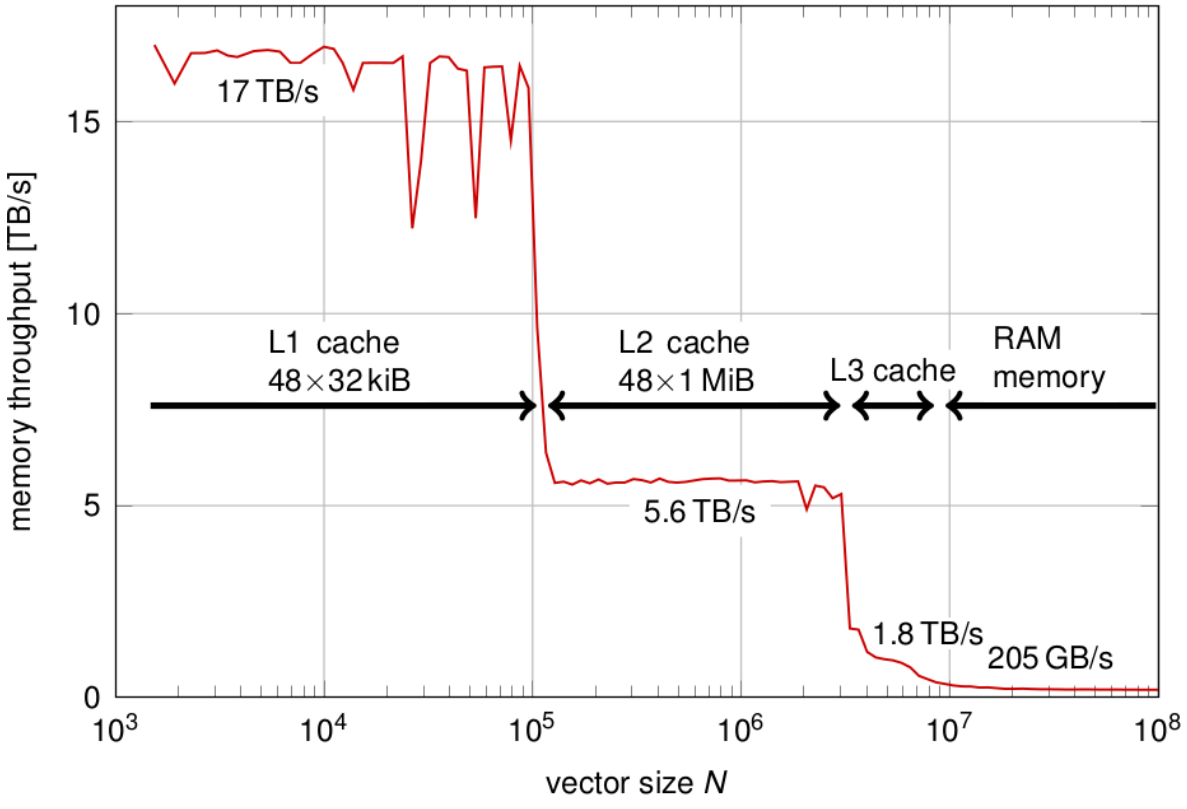
- Sit between registers and main memory
- Smaller than RAM, but (much) faster
- **Hierarchy of caches** to balance size versus speed
- Typical layout today: 3 levels of data caches, for example on Intel server processors
  - **Level 1** (L1) data cache: 32 kiB / core
    - Best throughput 192 byte / 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!

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

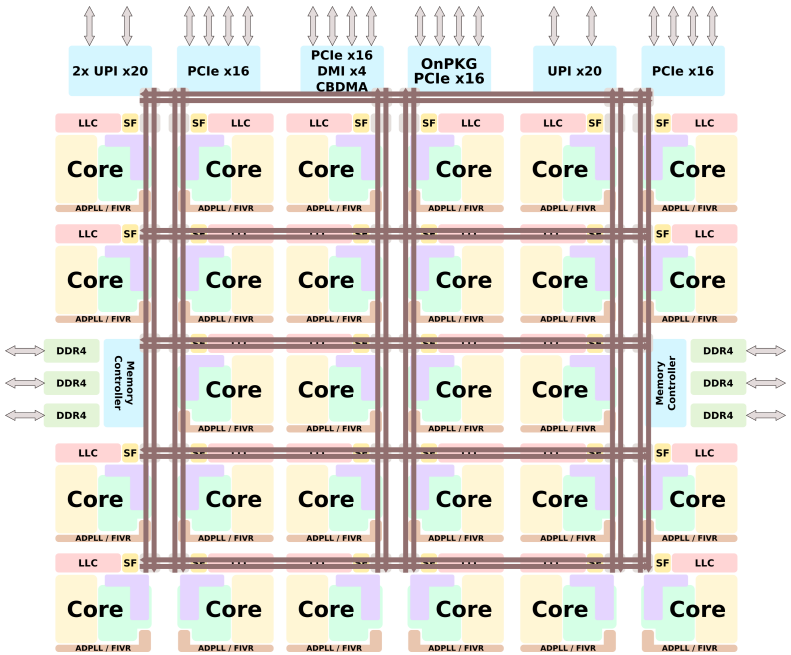


Image source: https://en.wikichip.org

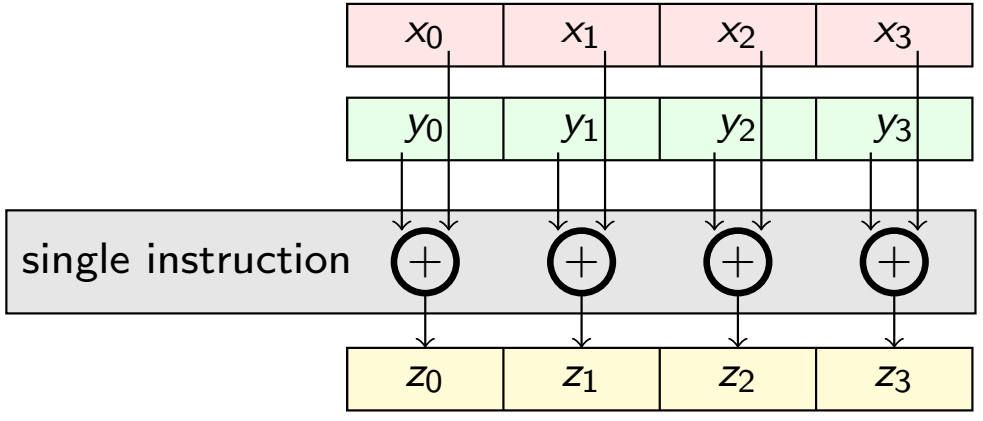
Data-level parallelism on CPUs: SIMD

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];
```

```
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. 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: Intel 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`

Graphics Processing Units – GPUs

GPU computing history

Early 2000's:

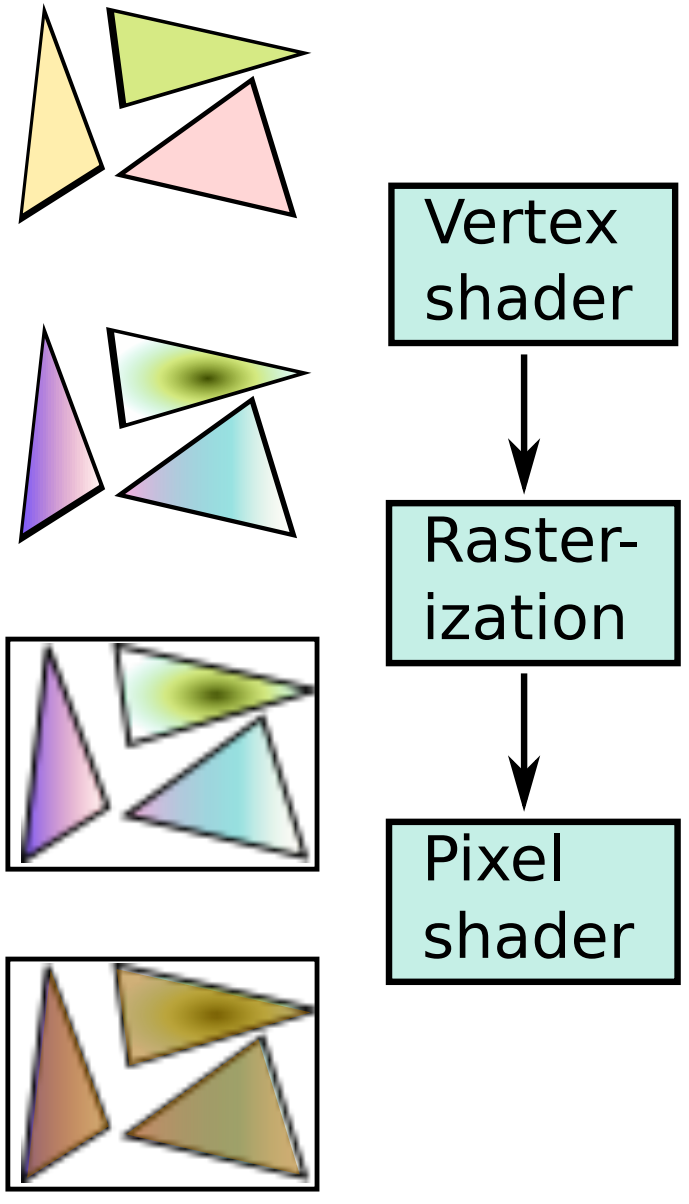
- GPUs (graphics processing units) for 2D rendering of 3D scene (transform, clipping, and lightning)
- Question: Use extended computing capabilities be something else?

Shader programming:

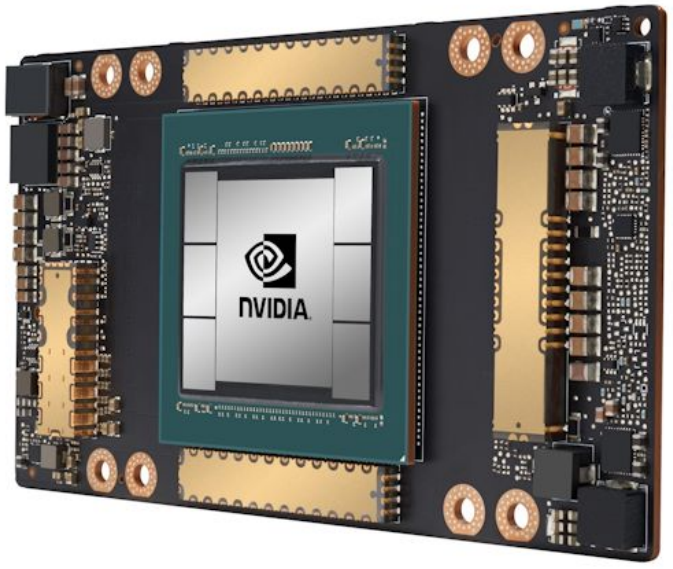
- Exploit rendering pipeline
- Vertex and pixel shaders
- Vectors (colors, positions) and matrices natively
- Really messy!

2006:

- Nvidia CUDA: make GPU programmable



GPUs in 2020



- GPUs concentrate on compute aspects
- Massive parallelism
- Performance higher than most other architectures
- Power efficiency: 2–5 times better performance / Watt than a CPU
- GPUs relatively cheap, since “already paid for” by gaming industry

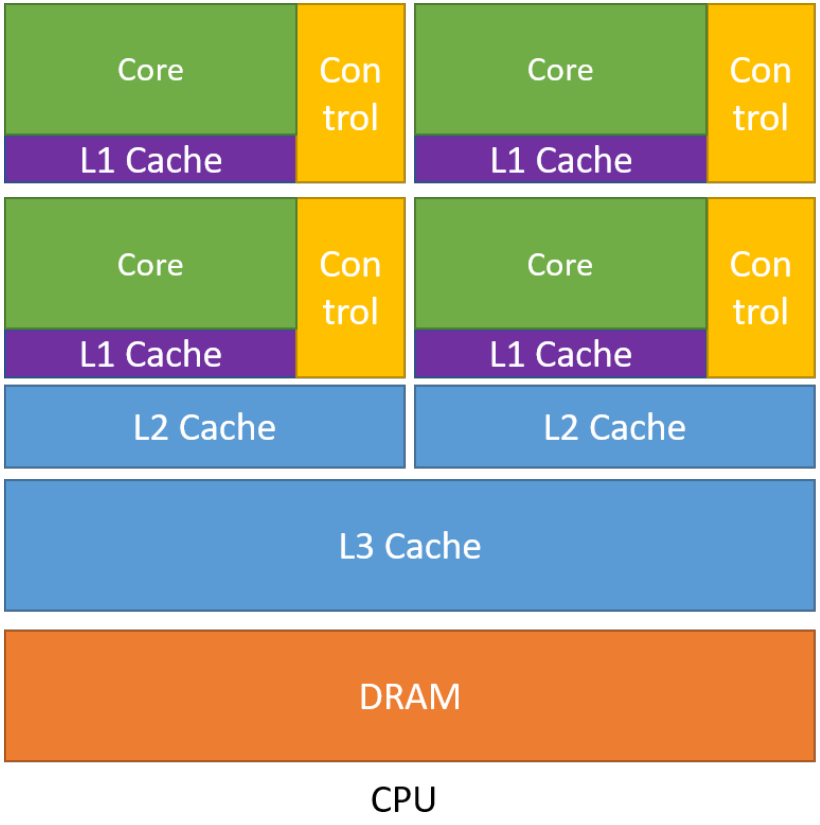
Performance metrics

- **Compute performance**: # floating point operations (add, subtract, mult, fused multiply-add) done per second (Flop/s)
  - Divisions & square roots run slower
  - Trigonometric functions sometimes in software by many (20-100) floating point operations, sometimes in hardware
- **Memory bandwidth**: memory operations per second
  - Can differ for read and write operations
  - Differs between fast cache memory and slow main memory
- GPU come with medium-big **on-board memory** (GDDR, HBM): 8–40GB (or 80 GB on highest end as of 2021)
- Comparison
  - 2020: NVIDIA Ampere A100, 9.7 TFlop/s and 1.6 TB/s with 400 W (HPC-oriented GPU)
  - 2021: Intel Xeon Platinum (Ice Lake), 40 cores, 2.3 GHz, 2.9 TFlop/s and 205 GB/s with 270 W (generic CPU)
  - 2021: AMD Epyc 7713, 64 cores, 2.0 GHz, 2.0 TFlop/s and 205 GB/s at 225 W (generic CPU)
  - 2019: Fujitsu A64FX, 48 cores, 2.2 GHz, 3.1 TFlop/s and 900 GB/s at 130 W (special-purpose HPC)

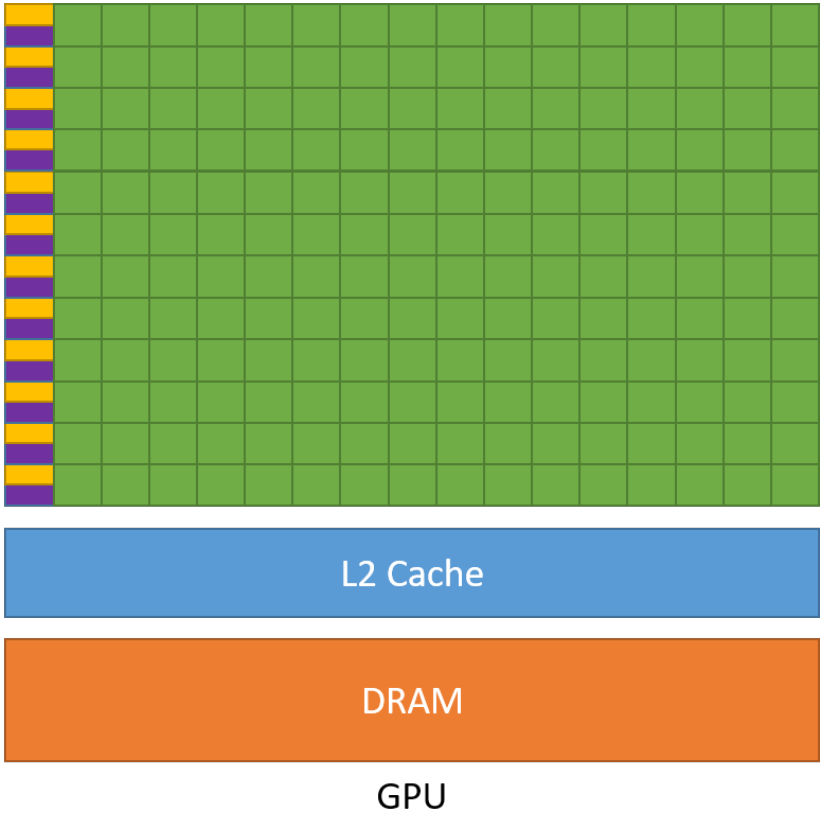
Efficiency of CPU vs GPU

CPU: Few complex cores, big control unit with branch prediction & out-of-order execution, favor latency over bandwidth via big caches & prefetching

GPU: Many simple cores, concentrate on compute, little memory, control unit responsible for many cores, schedule from several parallel streams in parallel to hide latency



CPU



GPU

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CPUs and GPUs are different

- Different goals produce different designs!
  - CPU must be good at everything
  - GPU focuses on massively parallel computations with independent work
    - Less flexible and more specialized
- CPU minimizes **latency experienced by 1 thread**
  - A lot of area devoted to extract "parallelism" on the fly (pipelining, out-of-order, speculative execution)
  - High clock frequencies
  - Execution units with low latency but more power draw ("bigger" transistors)
- GPU maximizes **throughput of all threads**
  - Provide lots of resources to enable many threads in flight, e.g. registers to hold operands
  - Multithreading can hide latency $\rightarrow$ limited-sized caches
  - Share control logic across many threads

GPU vs CPU+SIMD

- SIMD on CPUs has similar goals as GPU: increase compute
- On a given transistor manufacturing technology (lithography), SIMD on CPUs typically inferior against GPU
  1. SIMD width not wide enough to hide control logic enough
  2. SIMD execution units work with caches primarily designed by latency tradeoffs
  3. Less parallelism exposed by traditional CPU programs, out-of-order mechanism does not reach far enough
  4. Memory access latency covered by prefetching: More power-hungry than real access
  5. SIMD execution units "extend" latency hardware $\rightarrow$ have lower latency and thus bigger, more power-hungry transistors
  6. Memory outside caches (RAM) uses less parallel but lower latency DDR RAM architecture rather than GDDR or HBM (high-bandwidth memory)
  7. Higher clock frequencies (NVIDIA A100: $\sim$ 1.4 GHz, CPUs $>$ 2 GHz all-core)
- AMD CPUs suffer from 1–7, Intel CPUs from 2–7
- Interesting CPU architecture A64FX: Hardware design with 1, 2, 5, 6, 7 closer to GPU

Hardware model for a GPU: accelerator

- GPUs are chips with higher computing performance for **some tasks**, mostly those with > 1000 independent work streams
- GPUs would be very bad for other parts, e.g. setup work in a computation that is inherently **serial**
- Remember Amdahl's law about speedup for n cores with parallel proportion p,

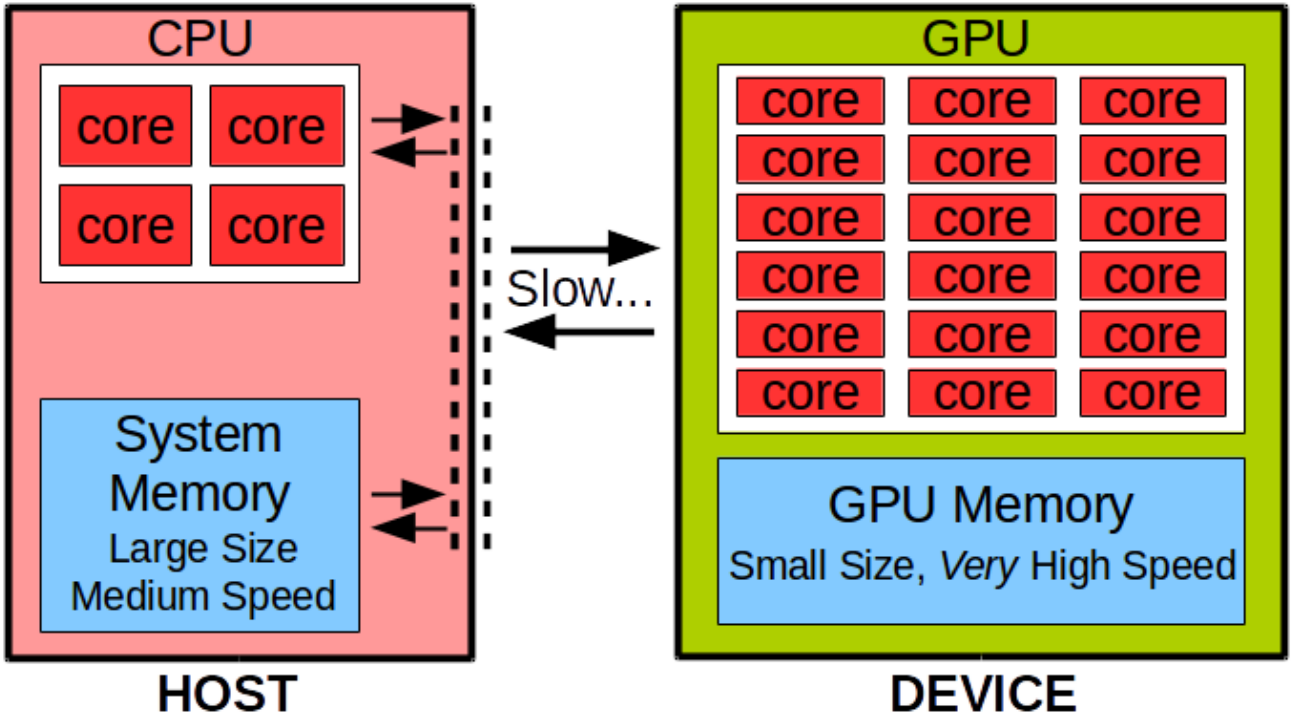
$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

  With 95% parallel work the speedup is limited by $20\times$
- Classical **CPU combined with GPU**
  - GPU is **accelerator** for suitable tasks (parallel, compute)
  - CPU runs the remaining tasks (bookkeeping, setup, IO, ...)
  - CPU and GPU might also share parallel tasks
  - Typical hardware is a **hybrid of accelerator and host**
  - Case for UPPMAX GPU nodes: Intel host CPU and NVIDIA T4 accelerator

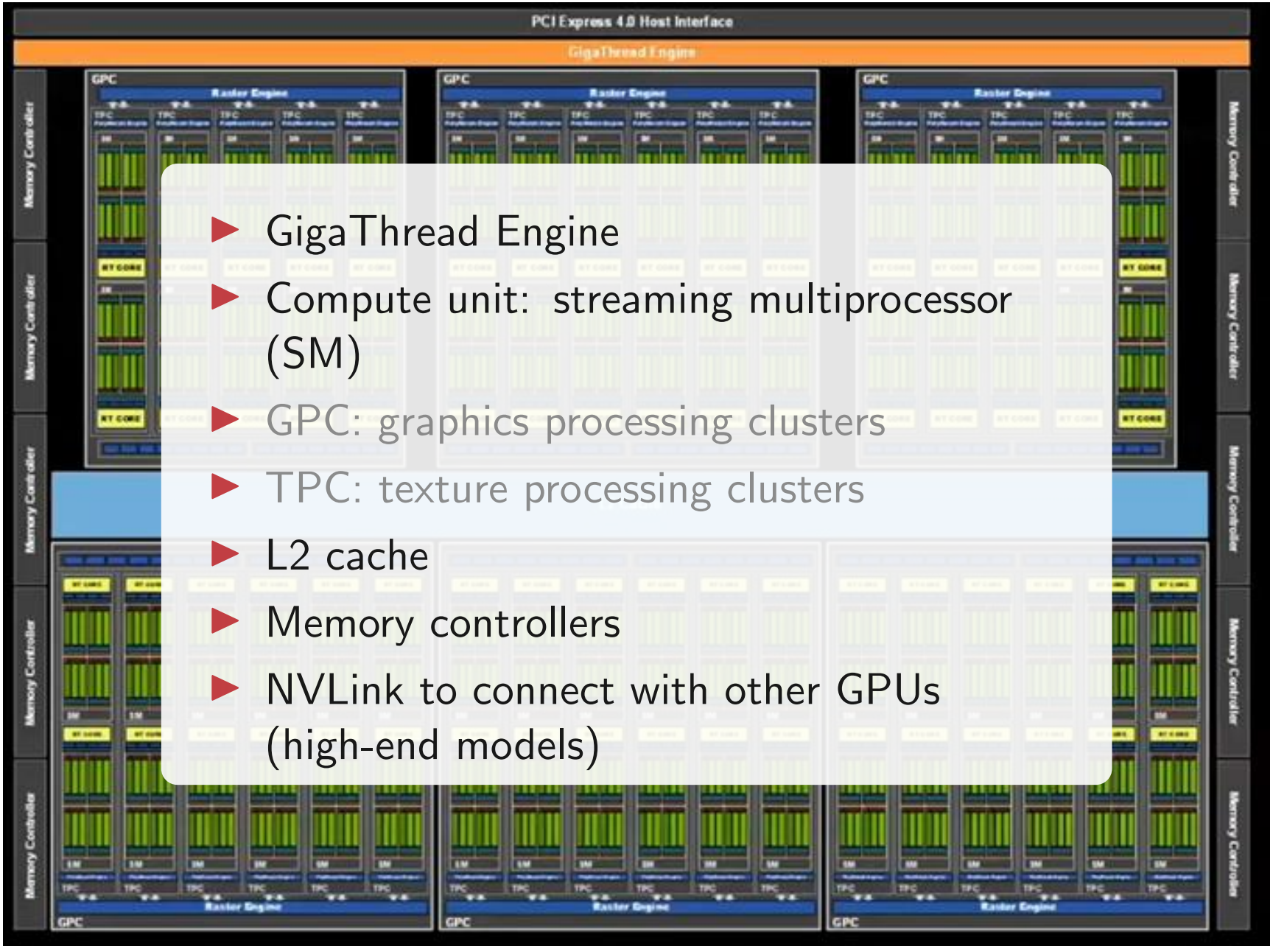
Visualization of hardware model

System:

- The Device connected to a *Host* (CPU)



Inside an NVIDIA GPU architecture



- GigaThread Engine
- Compute unit: streaming multiprocessor (SM)
- GPC: graphics processing clusters
- TPC: texture processing clusters
- L2 cache
- Memory controllers
- NVLink to connect with other GPUs (high-end models)

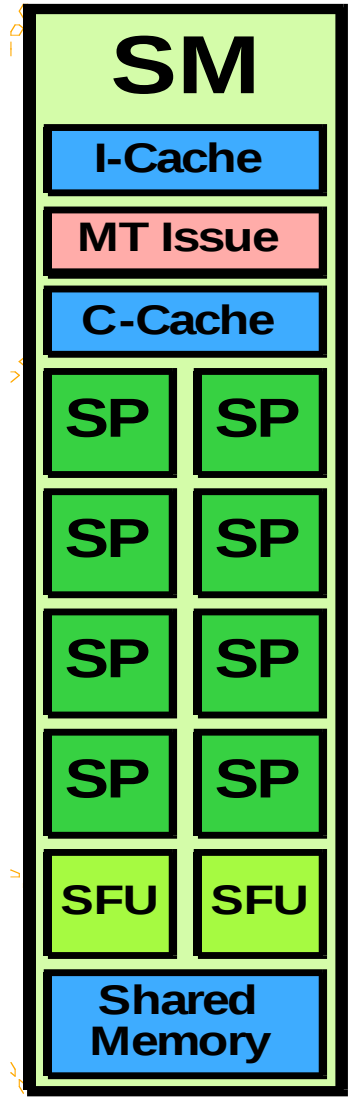
Hardware model of a GPU

Fundamental entity:

- *CUDA core* or *Streaming Processor* (*SP*)

Streaming Multiprocessor (SM):

- A collection of CUDA cores (8 / 16 / 32 / 192)
- All cores in one SM run the same instructions
- Has some fast, shared cache memory
- Can synchronize



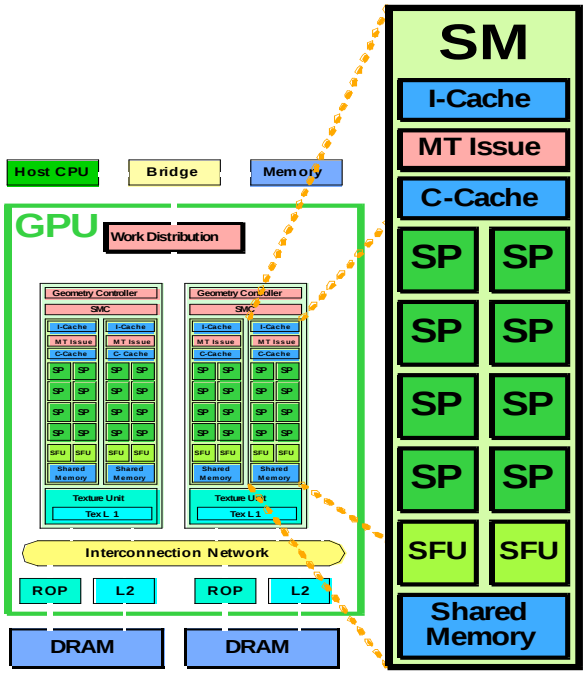
Hardware model of a GPU

The GPU

- A collection of SMs + memory

Scalable hardware design for consumer GPUs in graphics

- 2x2 8-core SMs, 32 cores



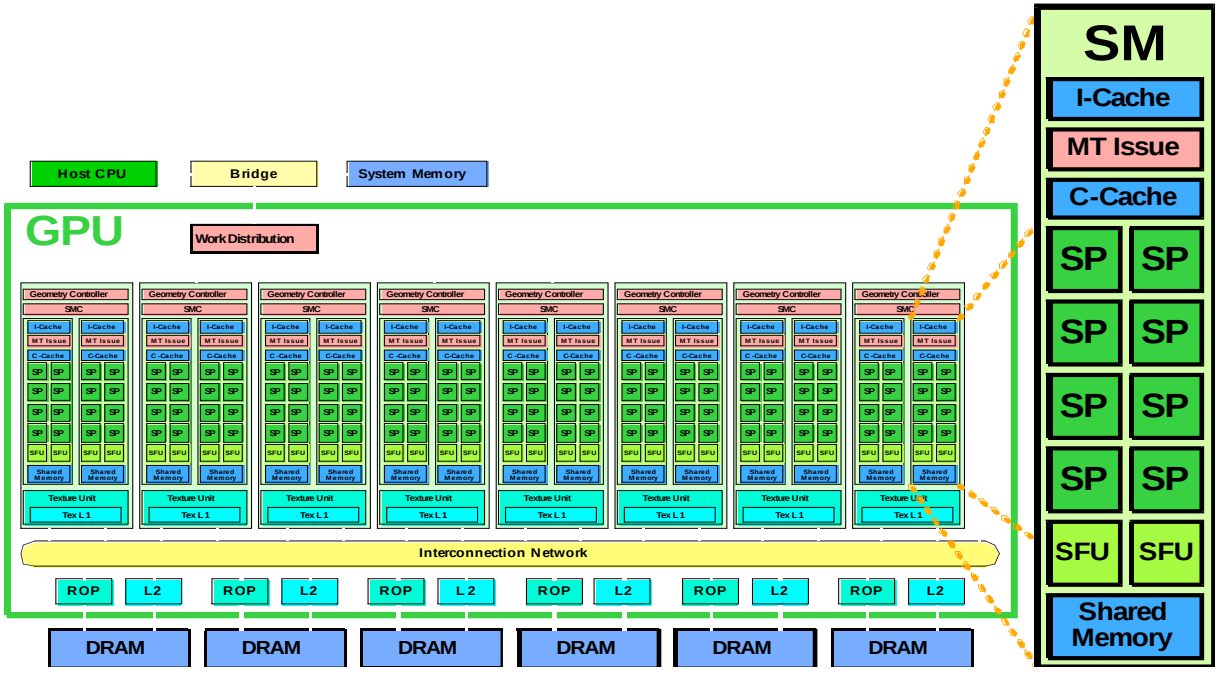
Hardware model of a GPU

The GPU

- A collection of SMs + memory

Scalable hardware design for consumer GPUs in graphics

- 8x2 8-core SMs, 128 cores



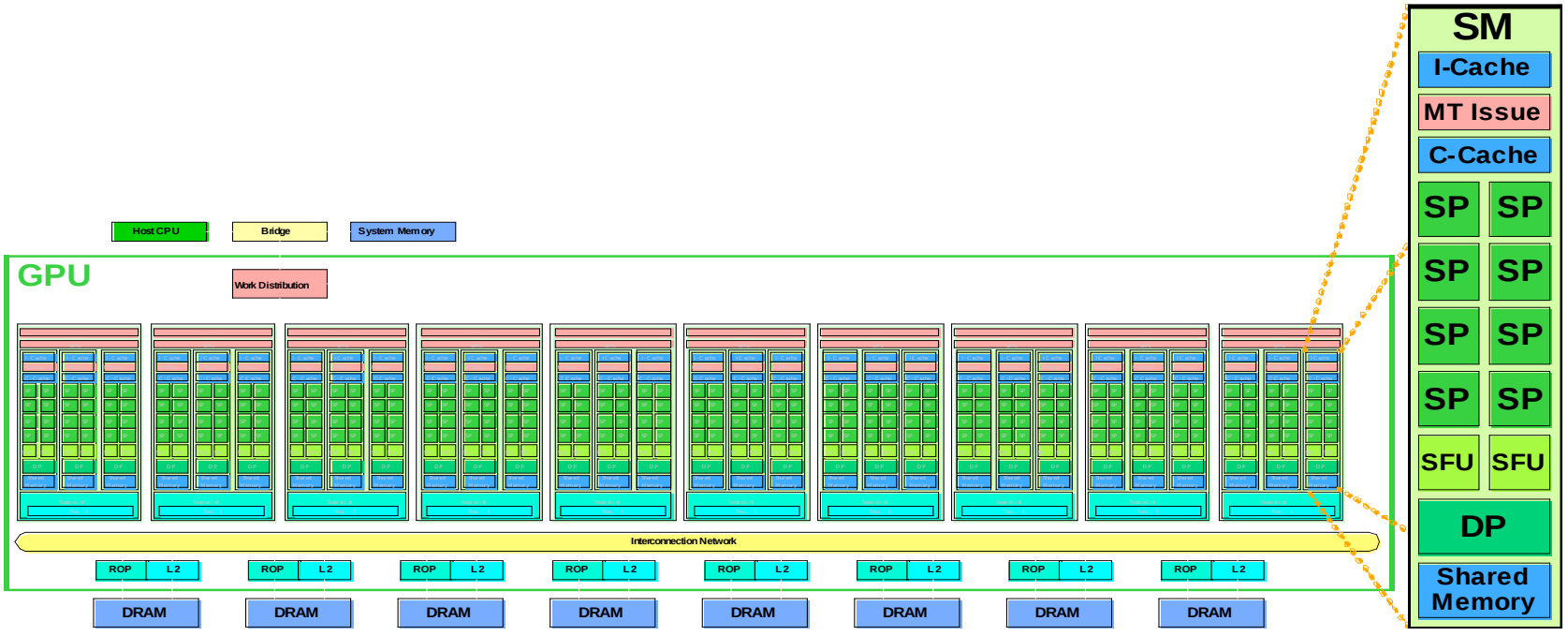
Hardware model of a GPU

The GPU

- A collection of SMs + memory

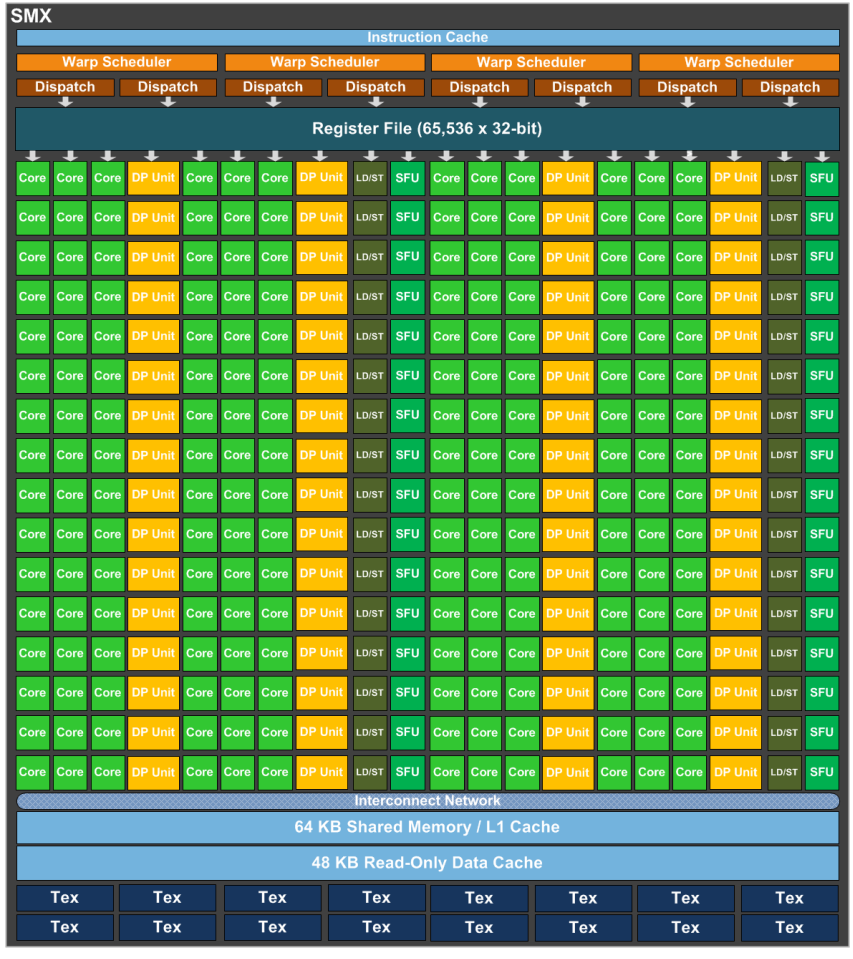
Scalable hardware design for consumer GPUs in graphics (data center/HPC: typically largest configuration)

- 10x3 8-core SMs, 240 cores



Architecture from 2012: NVIDIA Kepler

- SMX: 192 simpler cores
- GPU GK110 (K20): 15xSMX
- **2880 cores!**



Architecture from 2021: NVIDIA Ampere

- Different types of cores
  - CUDA cores (INT/FP32/FP64)
  - LD/ST
  - Special function units
  - Tensor core
- Register file
- Warp scheduler
- Data caches
- Instruction buffers/caches
- Texture units

Reading more about NVIDIA Ampere

NVIDIA Ampere core architecture

`https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`

See separate recording for a short insight into this document

Additional reading: `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`

Vendors for GPUs

- NVIDIA: market leader
- AMD: new player on the HPC market with good performance
  - US supercomputer Frontier (first to reach 1 Exaflop/s in double precision by late 2021, i.e., next week?) uses AMD GPUs
    See: `https://en.wikipedia.org/wiki/Frontier_(supercomputer)`
- arm (formely ARM): low-power devices (mobile platforms mostly)
- Intel: new effort on GPUs by their Xe architecture
  `https://en.wikipedia.org/wiki/Intel_Xe`

Current and upcoming supercomputers

	year	Petaflop/s	architecture	language
Fugaku	2020	500	Fujitsu ARM	C++/Fortran
Summit	2019	200	IBM CPUs + Nvidia GPUs	CUDA
Perlmutter	2021	100	AMD CPUs + Nvidia GPUs	CUDA
Frontier	2021	1500	AMD CPUs + AMD GPUs	HIP
Aurora	2022	1000	Intel CPUs + Intel GPUs	SYCL
El Capitan	2023	2000	AMD CPUs + AMD GPUs	HIP

Targeting a GPU architecture

- GPUs are relatively hard to program
- Often vendor-specific implementations
- Need to **write a separate code** for the GPU
- Limited compiler support
- GPU programs must consider hardware model of a host (CPU) and device (GPU)
- Algorithms need to be parallelized and mapped to the hardware: Must deal with separate memory spaces and limited bandwidth between host and device memory
- Requires software to be rewritten in specialized programming languages
- Optimization for compute performance requires knowledge about hardware

Memory model for the GPU

- GPU memory typically smaller than host memory
- Multiple GPUs each have their own device memory
- Data copied to the GPU may become stale on the host
- Transferring data to the GPU is expensive (low bandwidth of PCIe, better with newer NVLink)
- Best to keep working with transferred data as long as possible
- Possible to overlap data transfer with GPU computations
- Old model: Manage both memory types explicitly
- Newer model: Unified memory addresses
  - Can use pointers on both host and device transparently
  - But how do we make sure the data is where we want it to be?

High-level view of programming of a host/device system

- GPU memory allocation
- Transfer data from CPU to GPU
- CPU calls GPU kernel
- GPU kernel executes
- Once GPU kernel finished, transfer data back from GPU to CPU
- GPU memory release

- Host manages
  - both host and device memory;
  - data transfer between host and device;
  - starting device kernels.
- Device executes kernels that
  - are comprised by huge amounts of parallel threads at the same time
  - divide the data-parallel workload among these threads
  - switches execution between groups of threads to hide memory latency

Comparison on UPPMAX: CPU vs GPU

Rackham CPU node

- 20 cores
  - At most 40 simultaneous threads
  - Counting 8-wide SIMD (AVX-2), 320 "SIMD lanes"
- 128 GB of system memory
- 60 GB/s memory bandwidth
- 5120 kB L2 cache + 51200 kB L3 cache
- 250 W for just CPUs

Tesla T4 GPU

- Lightweight GPU launched in 2018
- 2560 CUDA "cores" (organized into 40 streaming multiprocessors)
  - At most 40,960 threads ("SIMT lanes")
- 16 GB video memory
- 300 GB/s memory bandwidth
- 3840 kB SM-specific memory/cache in total
  - Shared by 64 CUDA "cores"
- 70 W for GPU