Parallel and Distributed Computing (B4B36PDV)

Matěj Kafka, Michal Jakob

kafkamat@fel.cvut.cz

https://pdv.pages.fel.cvut.cz

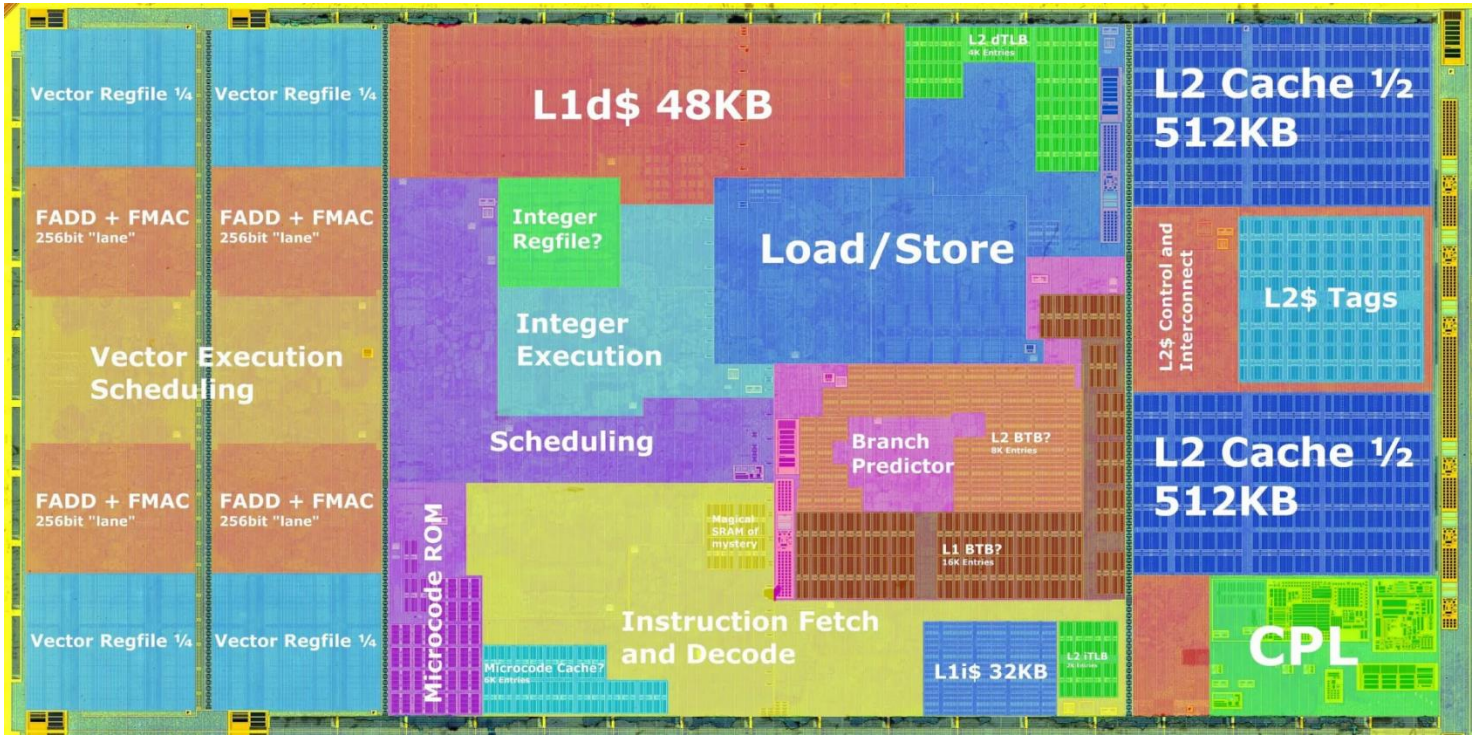
Small-scale parallelism

SIMD (VECTORIZATION)

SIMD

Lighter-weight parallelism

- The point of a CPU is to compute things.
- But most of the CPU is loading/storing data, scheduling instructions, predicting and handling branches,…

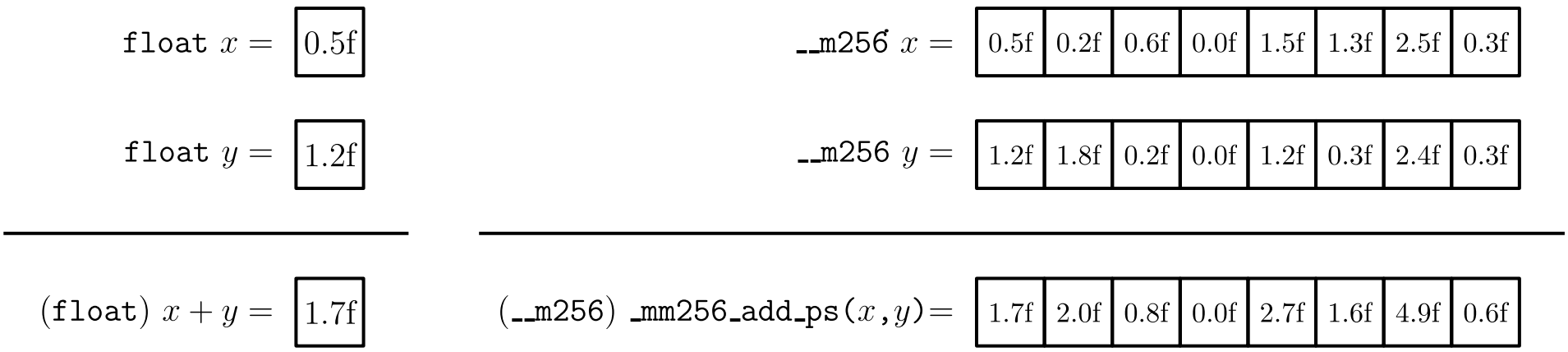


Source: https://nemez.net/die/

SIMD

Lighter-weight parallelism

- Could we somehow utilize more of the CPU for useful computation?
- Idea: Instructions that apply the same arithmetic operation to multiple values -> **vectorization**.
  - SIMD = single instruction, multiple data
  - We amortize the overhead of control flow, memory access, scheduling,… by only executing those steps once, but running multiple arithmetic operations.



SIMD

Lighter-weight parallelism

- Implemented by most architectures as a set of CPU instructions that operate on a separate set of registers.
  - Typically available for fixed vector widths: 128, 256, 512 bits

```
; load next value
mov eax, [rdi]
mov ebx, [rsi]
; multiply values
mul ecx, eax, ebx
; add to accumulator
add r8d, r8d, ecx
; increment input iterators
; 32bit integer
add rdi, 4
add rsi, 4
; decrease remaining count
sub rdx, 1
```

```
; load next block of input
vmovdqu ymm1, [rdi]
vmovdqu ymm2, [rsi]
; multiply vectors
vpmulld ymm3, ymm1, ymm2
; add to accumulator
vpaddd ymm0, ymm0, ymm3
; increment input iterators
; (8 integers, each 4 bytes)
add rdi, 32
add rsi, 32
; decrease remaining count
sub rdx, 8
```

Available SIMD operations

What can we do with CPU vector instructions?

- load/store from memory
- fill vector with a scalar value (broadcast)
- basic arithmetic operations, including bitwise ops
- min/max
- shuffling (permuting values in a vector)
  - only predefined patterns
- masked operations (e.g., blending, compression)
- comparisons

- …many other, often domain-specific instructions to speed up specific computations…

Use cases for SIMD

Where do you think SIMD could be useful?

Use cases for SIMD

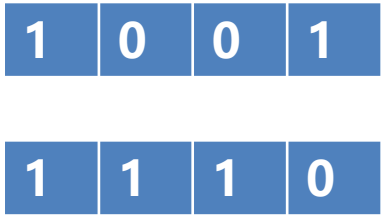
Everywhere…

- almost any string manipulation, memcpy,…
  - Heavily used in most `std::string` implementations.
- operations on data structures with continuous storage
  - Even trees, if you're clever.
- matrix and vector operations
  - As some of you will see next semester, a lot of things can be expressed using matrices.
- image processing (and most other GPU workloads)
- DSP (digital signal processing)
- sorting numeric values (e.g., bitonic sort)
- game simulations (e.g., Reversi from RPH)

Example: bitset (bit array)

SIMD example without SIMD ("SWAR")

- Surprisingly often, we work with sets of small numbers.
- How to represent them more compactly? Bitsets.
  - each bit in a number corresponds to a value
  - 1 = value is present
  - 0 = value is not present
- Common operations (union, intersection, membership) can be done in parallel using bitwise operations.
- How to count elements? Popcount.

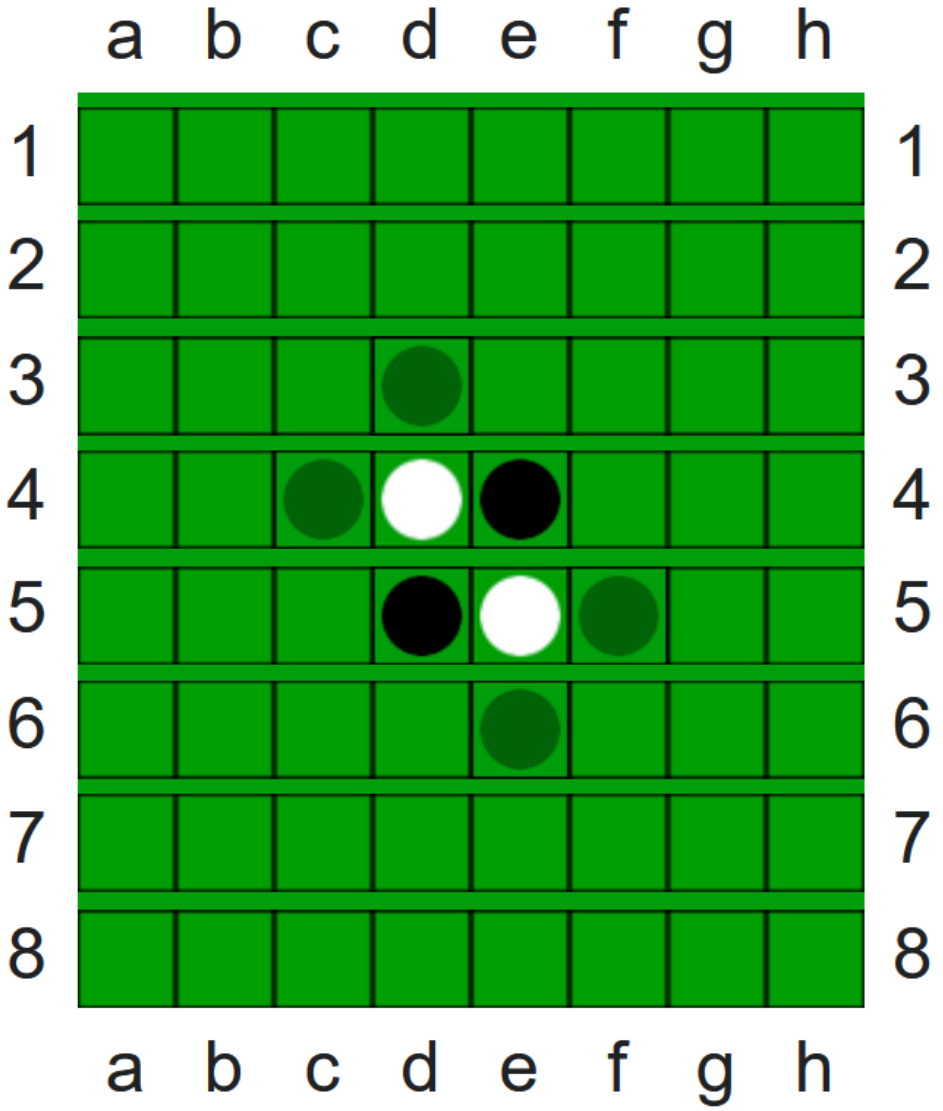


intersection = AND
union = OR
difference = XOR

Example: Reversi (bitboard)

Another SIMD example without SIMD (and with SIMD)

- We can represent the 8x8 board as a 64-bit bitset.
- One bitset for our stones, one for the opponent's.
- How to find valid moves?
  - More bitwise operations!
- How to evaluate multiple moves in parallel?
  - Most of move search is branchless -> SIMD…
  - …and multithreading.



Example: Reversi (bitboard)

Finding moves (and winning at RPH)

```
blankFields = ~(player | opponent)

# masks the 1st and last column, prevents overflow to the next line
maskedOpp = opponent & 0x7e7e7e7e7e7e7e7e

# get left moves
# OP
t = maskedOpp & (player << 1)
# OP or OOP
t |= maskedOpp & (t << 1)
# OO
o2 = maskedOpp & (maskedOpp << 1)
# OOOOP or OOOP or OOP or OP
t |= o2 & (t << 2)
# OOOOOOP or OOOOOP or OOOOP or OOOP or OOP or OP
t |= o2 & (t << 2)
# blank fields with any sequence of opponents & player to the right
leftMoves = blankFields & (t << 1)
```

Using SIMD from C/C++

Assembly is somewhat old-school…

- We could use inline assembly, but compiler does not understand the operations and cannot optimize around them, which is a major issue.
  - Also quite cumbersome to write.
- Compilers provide so-called "intrinsics" – arch-specific functions that correspond to specific CPU instructions.
  - Compilers also provide types to represent vectors of various width and types.

```
__m256 exp_vec(__m256 x) {
    __m256 three = _mm256_set1_ps(3.0f);
    __m256 addthree = _mm256_add_ps(x, three);
    __m256 subthree = _mm256_sub_ps(x, three);
    return _mm256_div_ps(
        _mm256_add_ps(_mm256_mul_ps(addthree, addthree), three),
        _mm256_add_ps(_mm256_mul_ps(subthree, subthree), three)
    );
}
```

https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

Example: String operations

String operations tend to be very easy to vectorize.

- `strlen` (find a null byte)
  - Load 32 bytes into a SIMD register.
  - Compare with vector of zeros.
  - Compress mask into a scalar register.
  - Check if the value is zero, otherwise repeat.
- `str1 == str2`
  - Like strlen, just compare with the other string instead of zeros.
- `str1.find(str2)`
  - For a single char similar to strlen.
  - For multiple chars, look for e.g., the first and last chars at a certain offset from each other, then check if the string between matches.

http://0x80.pl/notesen/2016-11-28-simd-strfind.html

Using SIMD from C/C++

Autovectorization

- The compiler can also sometimes automatically optimize scalar code to use SIMD instructions = "autovectorization".
- Clang is somewhat good at it, GCC and MSVC not so much.
- Not a panacea, often we need to help the compiler by rearranging the computation, changing data structures,…
- We need to specify the microarchitecture to compile for, defaults are very conservative (`-march=…`).

```
void multiply_vec(uint32_t multiplier,
        std::vector<uint32_t>& vec) {
    for (auto& n : vec) {
        n = n * multiplier;
    }
}
```

```
.L4:
vpmulld ymm0, ymm1, YMMWORD PTR [rax]
add     rax, 32
vmovdqu YMMWORD PTR [rax-32], ymm0
cmp     rdx, rax
jne     .L4
```

https://godbolt.org/z/eWf5dEjeP

Using SIMD from modern C++

Intrinsics are not exactly readable.

- There is an experimental C++ standard library extension that provides **cross-platform** SIMD vectors.
- `std::experimental::simd`
- Seems on track to be accepted into C++26. For the experimental version, you must use reasonably modern GCC or Clang (**please update, you'll need it for the labs**).

```
using vec_f32 = std::experimental::native_simd<float>;

static vec_f32 exp_vec_cpp(vec_f32 x) {
    vec_f32 three{3.0f};
    vec_f32 x_plus_3 = x + three;
    vec_f32 x_minus_3 = x - three;
    return (x_plus_3 * x_plus_3 + three)
         / (x_minus_3 * x_minus_3 + three);
}
```

Using SIMD from modern C++

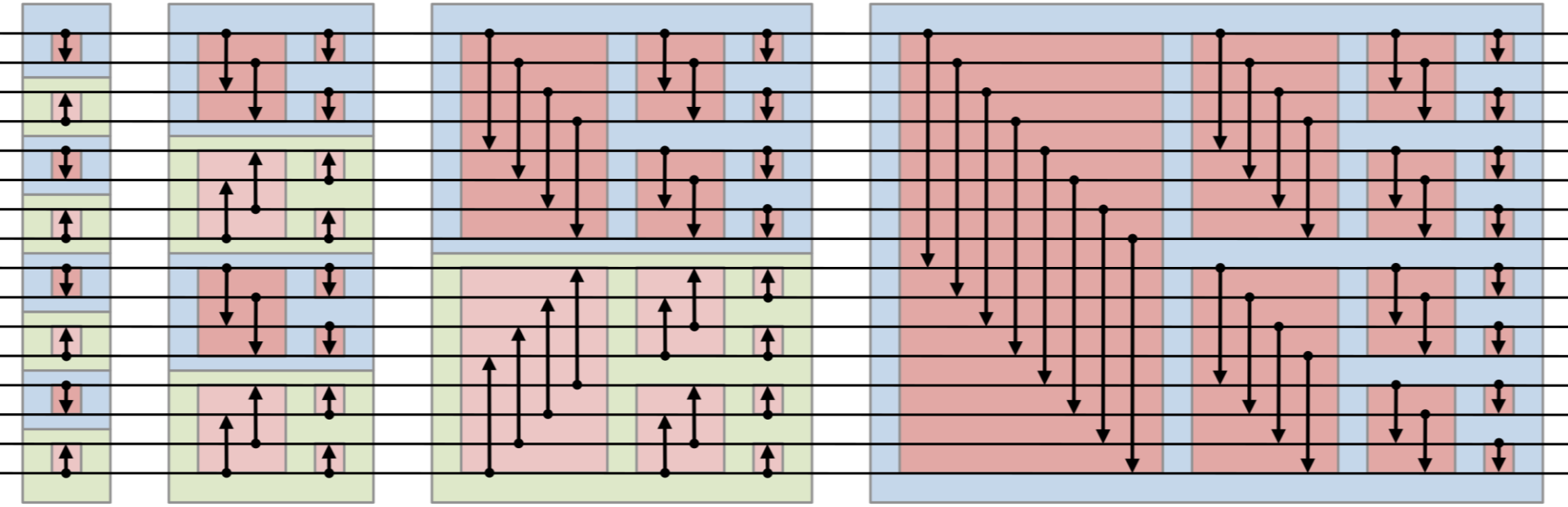
`std::experimental::simd`

- Due to being cross-platform, the exposed operations are somewhat limited compared to what, e.g., modern x64 provides, but good enough for our purposes.
- Note that available SIMD instructions differ a lot between different architectures, and even a single architecture (e.g., x64, ARM64) provides different SIMD instructions in different generations of CPUs (SSE, AVX, AVX2, AVX-512,...).
- There are similar libraries for a specific architecture (e.g., vectorclass).
- For many performance-sensitive workloads, the SIMD code is handwritten for a specific microarchitecture that it will run on, with manually optimized instructions to maximize throughput and saturate all available ALUs.
  - There are many tricks to do common operations quickly, mostly out of scope for PDV.

Example: Vector min/max

Using std::experimental::simd.

```
size_t half = N / 2;
for (size_t i = 0; i < half; i += vec_f32::size()) {
    vec_f32 low = vec_f32{&data[i], element_aligned};
    vec_f32 high = vec_f32{&data[half + i], element_aligned};
    min(low, high).copy_to(&data[i], element_aligned);
    max(low, high).copy_to(&data[half + i], element_aligned);
}
```

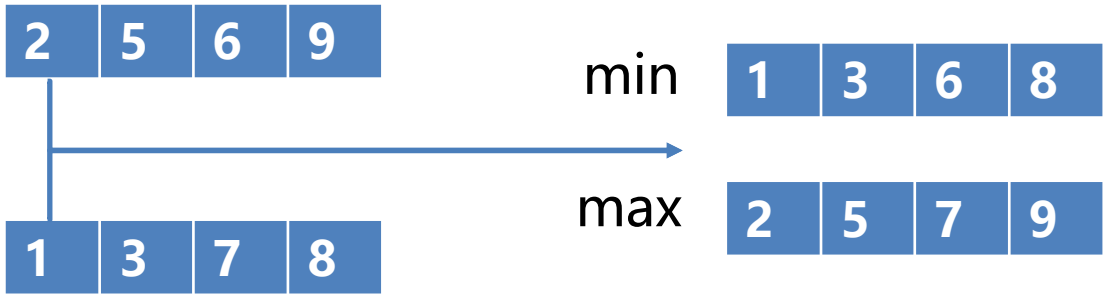


Source: https://commons.wikimedia.org/wiki/File:BitonicSort1.svg

Example: Vector min/max

Example use case (used in sorting networks)

Vector min/max with large blocks:



What about smaller blocks? (e.g., pairs)

x3	x2	x1	x0
2	5	6	4

Example: Vector min/max

Example use case (used in sorting networks)

What about smaller blocks? (e.g., pairs)

zero-extend

				x3	x2	x1	x0
0	0	0	0	2	5	6	4

shift right by 1

					x3	x2	x1
0	0	0	0	0	2	5	6

trim

	x3	x2	x1
0	2	5	6

compare with the original vector

x3	x2	x1	x0
2	5	6	4

we need min…

x3	x2	x1	x0
0	2	5	4

Example: Vector min/max

Example use case (used in sorting networks)

What about smaller blocks? (e.g., pairs)

We need the minimum…

x3	x2	x1	x0
0	2	5	4

…but only at even positions.

- min(x0, x1) at position 0
- min(x2, x3) at position 2
- …

Use a mask to zero out useless values.

x3	x2	x1	x0
0	2	0	4

Result is the OR of these two vectors.

Similarly, find maximum and store it at odd positions.

x3	x2	x1	x0
5	0	6	0

Mixing SIMD and OOP

Changing our data structures.

- Programs are commonly written in an object-oriented style – we use encapsulated objects with fields.
- For SIMD, we need to have an array of values.
- What to do about it?

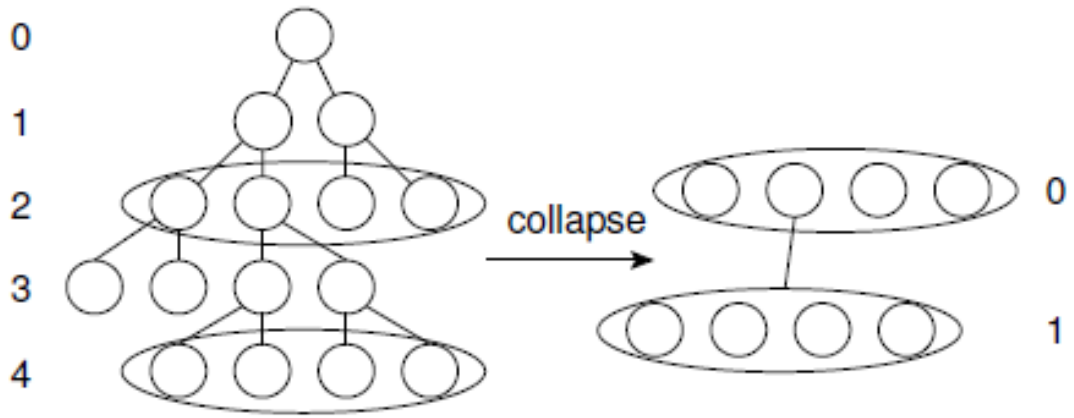
- "arrays-of-structs" (AoS) vs "struct-of-arrays" (SoA)
- E.g., instead of `List<(x, y, z)>`, we'll use `(List<x>, List<y>, List<z>)`.
- Very common in game engines.
- We must do this manually; compiler cannot change our data structures.

Tree data structures

Change the data structure again.

- Trees tend to use small nodes with pointers.
- Does not seem amenable to SIMD parallelization.

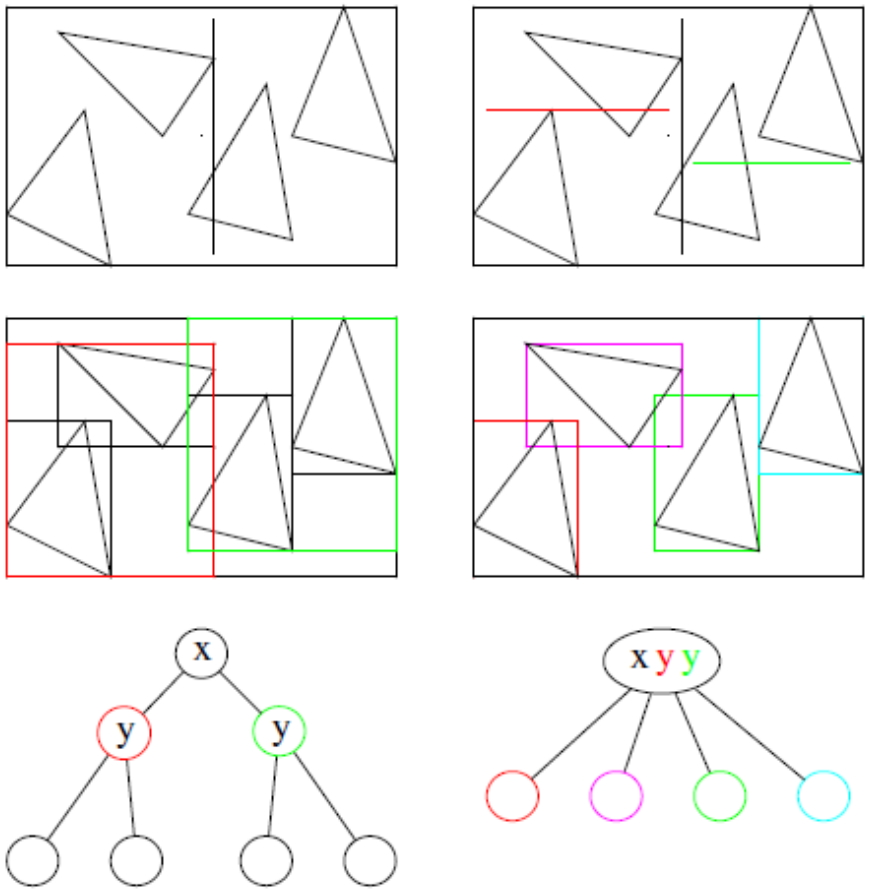
- Idea: Collapse a tree into a tree of arrays.
  - E.g., take every 3 levels of a tree and a build a single node with 8 children instead.
  - This also typically improves memory locality.



Example: Bounding volume hierarchy

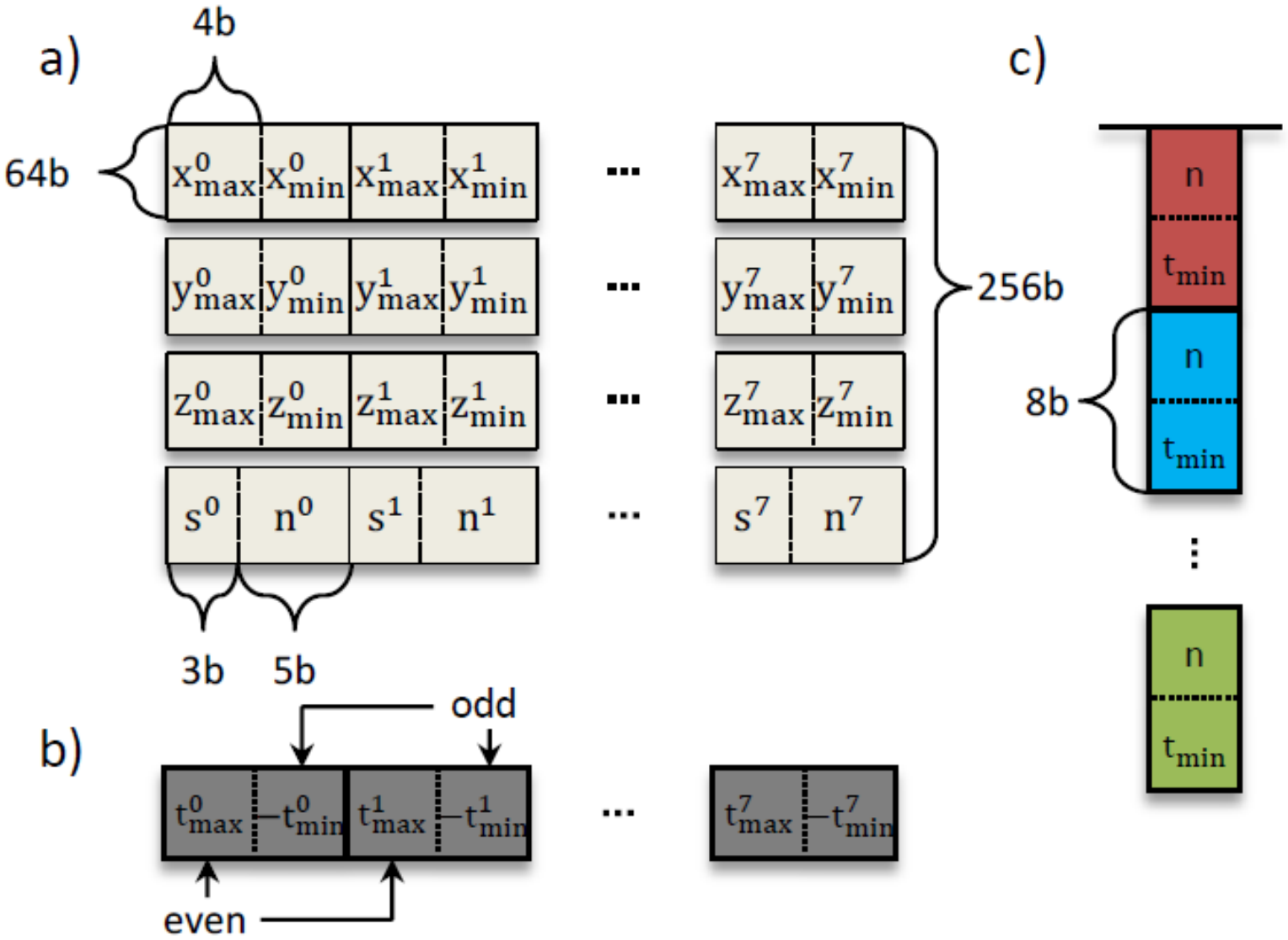
Faster raytracing using SIMD tree traversal.

- When raytracing, we're looking for the first object hit by a ray.
- Naive approach: intersect ray with each object, find min.
- Better: Build a tree that splits the objects into groups.
  - Store a bounding box for each group.
- When traversing the tree, only recurse when the ray intersects the bounding box.
- Ray-box intersection can be efficiently vectorized.



Example: Bounding volume hierarchy

Faster raytracing using SIMD tree traversal.



Somewhat dumb, but very parallel

GENERAL PURPOSE GPUS

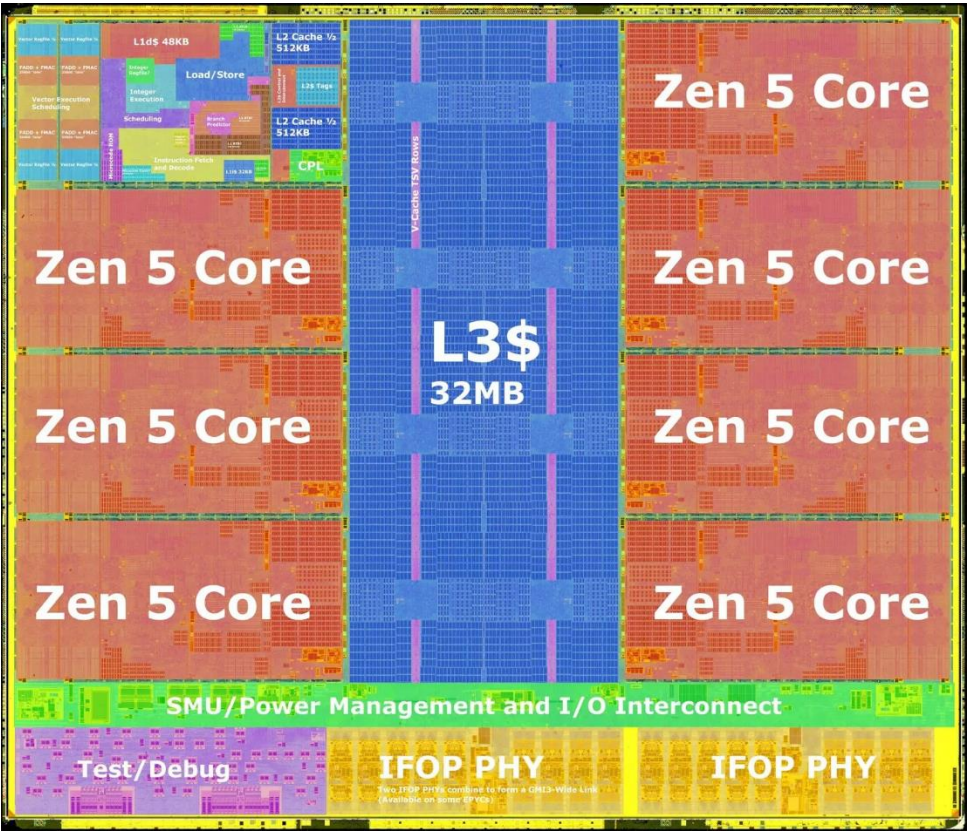
Short history of GPUs

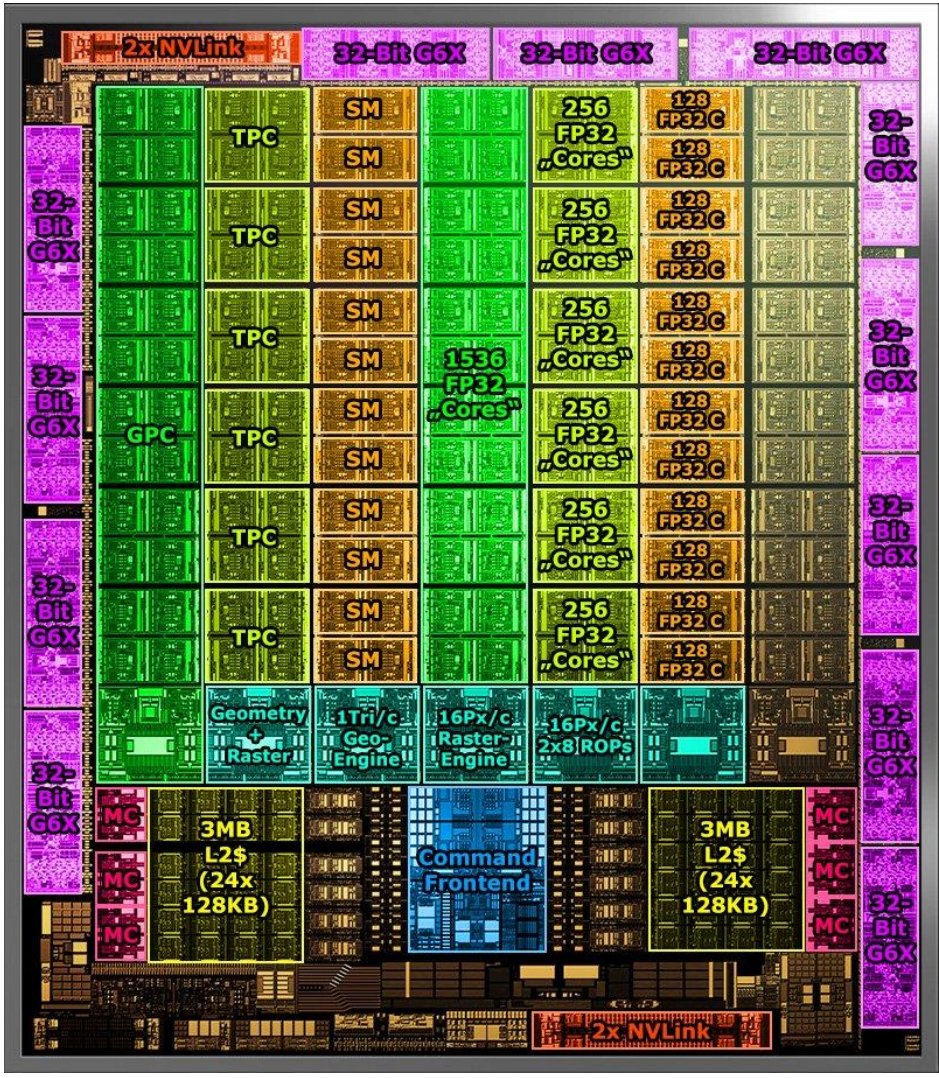
From fixed-function hardware to general computation.

- 3D rendering involves a lot of repetitive operations.
- Early GPUs provided hardware-implemented functions for common operations – linear transformation, texturing, interpolation,… –> **fixed-function pipeline**.
- Needed more flexibility -> shaders.
  - Small programs that process vertices or pixels in parallel.
- Primitive but highly parallel compute units turned out to be useful for scientific simulations, AI, raytracing,…
- Modern GPUs can be used as general-purpose computers for parallel workloads -> **GP GPU**s.

Less logic, more computation

Comparison of "useful" die area between a CPU and a GPU.





Source: https://x.com/Locuza_/status/1338265403022323713

Less logic, more computation

Comparison of thread counts and float operation throughput.

Threadripper PRO 7995WX

- 96 cores (192 hyperthreads)
- ~12 TFLOPS

NVIDIA RTX 4090

- 16384 shader units
- ~178 TFLOPS

Large scale parallelism

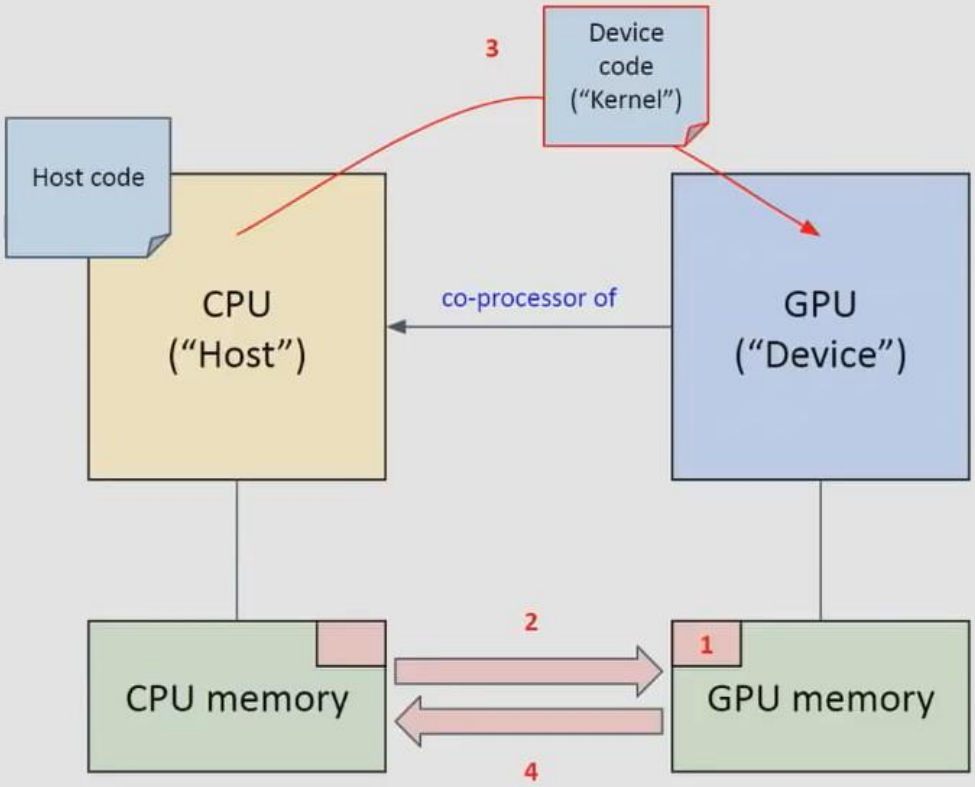
Comparing CPUs, SIMD and GPUs.

- CPU = large, complex cores, prioritizes sequential execution speed over throughput
- SIMD on CPUs = parallel ALUs, rest of pipeline shared with scalar operations, low latency for sharing data
- General-purpose GPU
  - highly parallel, primitive cores
  - prioritizes throughput over latency
  - high communication cost from the CPU

- accelerator = any specialized peripheral that improves performance of a specific computation
  - GPUs, ASICs, "AI accelerators", video encoders,...

Controlling a GPU

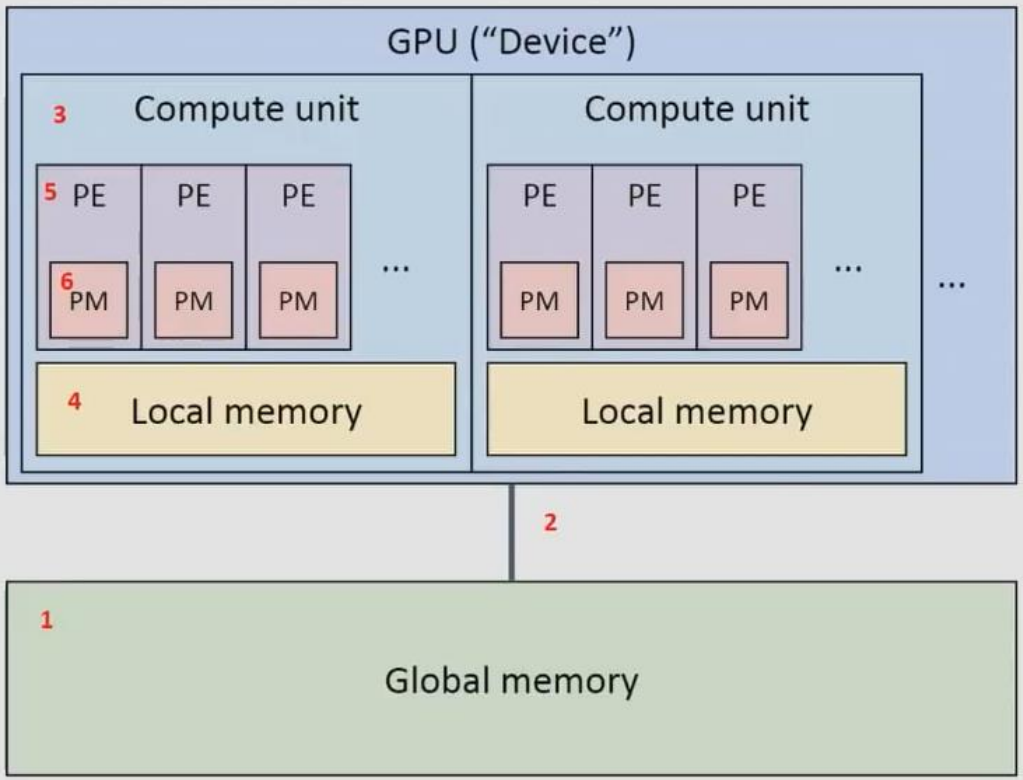
We'll discuss specific APIs later.



1. The CPU allocates memory on the GPU
2. The CPU copies data from CPU to GPU
3. The CPU launches kernel(s) on the GPU
4. The CPU copies data to CPU from GPU

Source: https://www.youtube.com/watch?v=miqZS6aS9K0

GPU structure & memory hierarchy

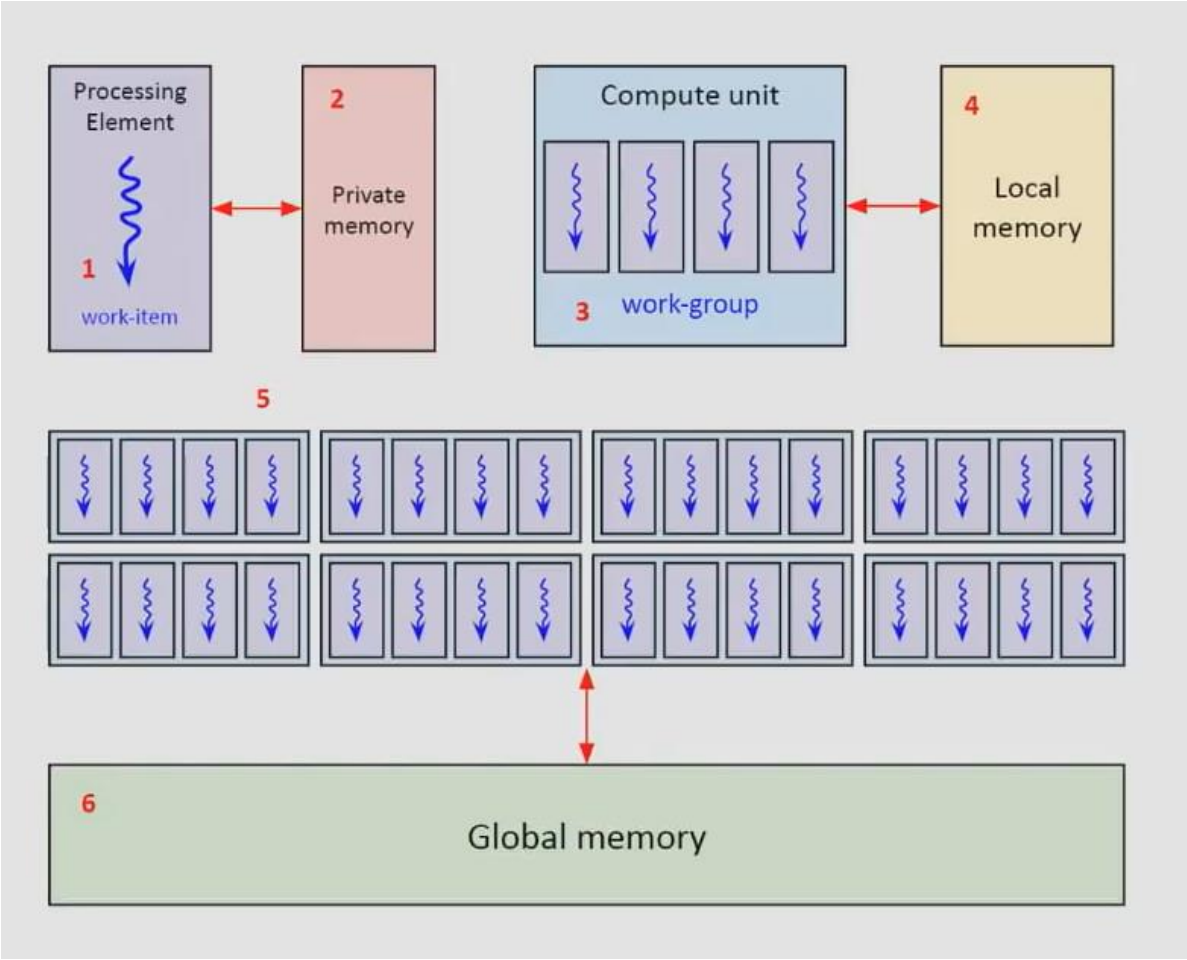


1. A GPU has a region of dedicated global memory
2. Global memory is connected via a bus
3. A GPU is divided into a number of compute units
4. Each compute unit has dedicated local memory
5. Each compute unit has a number of processing elements
6. Each processing element has dedicated private memory

Source: https://www.youtube.com/watch?v=miqZS6aS9K0

GPU threads

Lightweight threads, grouped into a 2-level hierarchy.



Source: https://www.youtube.com/watch?v=miqZS6aS9K0

Thread structure (Nvidia A100)

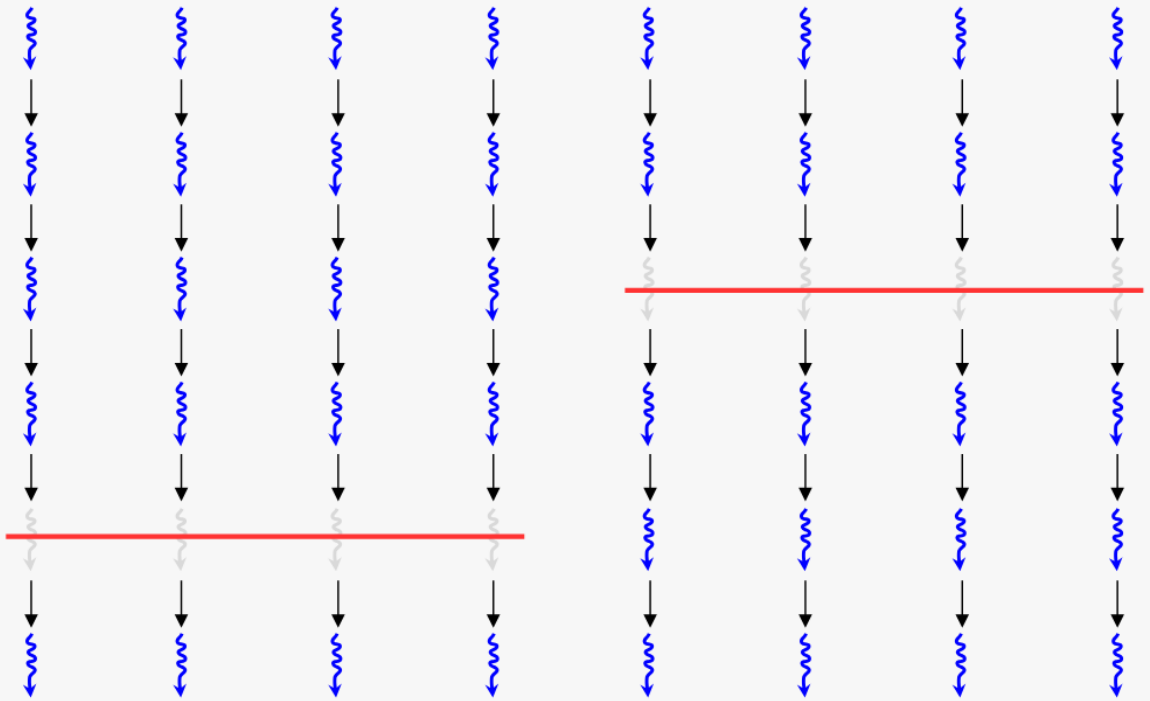
Highly parallel system with plenty of restrictions.

- General architecture is similar for most GPUs.
- Clusters -> streaming multiprocessors -> warps -> cores.
  - 32 cores per warp, 64 warps per SM, 12 SMs per cluster, 7 clusters per GPU.
- Threads are dynamically scheduled by the GPU onto cores, registers are dynamically allocated.
- All cores in a warp execute in lock-step (SPMD).
- Each warp has private registers (low latency) and instruction cache.
- Each SM has shared memory, reasonable latency.
- Whole GPU has global memory with L2 cache, very high latency (200+ cycles), but high throughput.

Control flow masking

Control flow is possible, but very slow.

```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



Source: https://www.youtube.com/watch?v=miqZS6aS9K0

How to program for GPUs?

- Write "kernels" that run on the GPU.
  - Commonly JIT compiled at runtime by the GPU driver.
- Both major GPU manufacturers have their own "native" API:
  - CUDA – Nvidia (by far the most popular API)
  - ROCm – AMD (can also target CUDA).
- Cross-platform APIs (not just GPUs):
  - OpenCL (C-based DSL, or a C/C++ API)
  - SYCL (specification for a native C++ API)
    - multiple implementations for different targets
  - OpenMP

```
// create a buffer of 4 ints to be used inside the kernel code
auto buffer = sycl::buffer<sycl::cl_int, 1>(4);

// describe how we want to partition the work
auto range = sycl::range<1>(buffer.size());

// submit work to the GPU
sycl::queue queue;
queue.submit([&](sycl::handler& cgh) {
    // get write-only access to the buffer on a device
    auto accessor = buffer.get_access<sycl::access::mode::write>(cgh);
    // process `range` in parallel
    cgh.parallel_for<class Iota>(range, [=](sycl::id<1> i) {
        // fill buffer with indexes
        accessor[i] = (sycl::cl_int) i.get(0);
    });
});

// access the buffer from the host (waits for the previous write op)
const auto host_accessor = buffer.get_access<sycl::access::mode::read>();

for (size_t i = 0; i < buffer.size(); i++) {
    if (host_accessor[i] != i) {
        std::cerr << "Incorrect result at index '" << i << "'\n";
        return 1;
    }
}
```

Summary of GPU programming

- different programming model
  - computation kernels, hardware-managed threads, lock-step
- different trade-offs
  - parallelism, throughput vs latency

- GPUs are a moving target (much more so than CPUs).

- Where to learn more?
  - General-Purpose Computing on GPU (B4M39GPU)
  - Parallel Programming (B4M35PAG)

Summary of parallel programming

What have you learned? (hopefully)

- Future of computing seems to lie in parallelism and specialized hardware.
- We saw multiple types of available parallelism.
  - ILP, multithreading, SIMD, GPUs
- Understanding how to think about decomposition, synchronization and knowing what the hardware wants is more important than specific techniques.
- What next?
  - Explore how parallelism is used in the wild as part of larger projects, not just toy examples.
  - Think about how to write parallel code that is safe and readable for others.
  - Learn how real-world CPUs and GPUs work.