# GPU

Jakub Žádník

**Customized Parallel Computing group**
**http://tuni.fi/cpc**

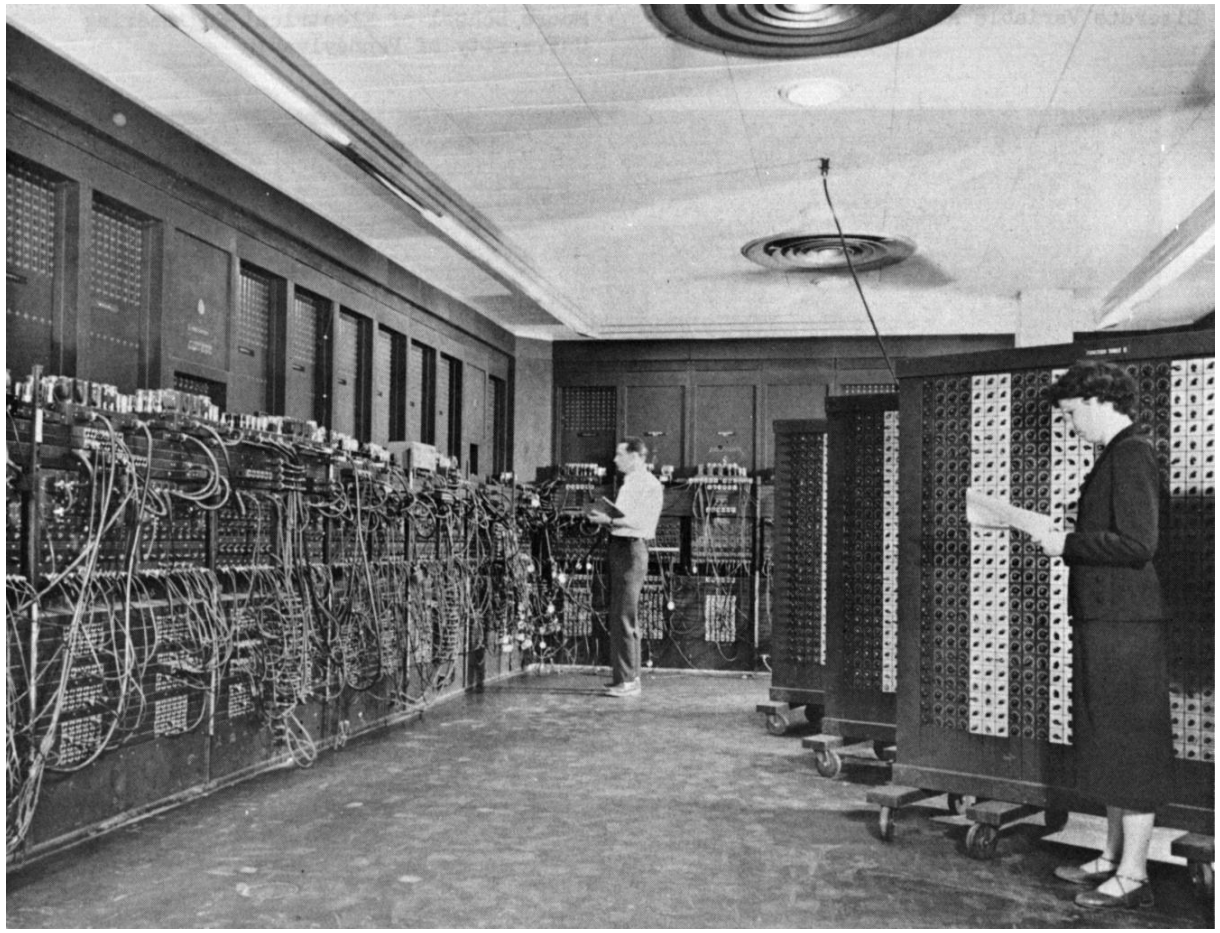
# Central Processing Unit (CPU)

ENIAC (1945, USA): The first

- Programmable
- Electronic
- **General-purpose**

digital computer

Equivalent of ~500 FLOPS (floating point operations per second)



https://commons.wikimedia.org/wiki/File:Glen_Beck_and_Betty_Snyder_program_the_ENIAC_in_building_328_at_the_Ballistic_Research_Laboratory.jpg

# Hardware Acceleration

**General-purpose CPU** traditionally optimized for

- calculating single values (scalar)
- in a single stream of instructions (single-threaded)

It can compute all workloads, but not necessarily efficiently

⇒ **Hardware acceleration:** Perform a small set of operations very efficiently (speed, energy)

- Digital Signal Processing (DSP)
- Cryptography
- Networking
- Regular expressions
- Compression
- **Computer graphics**
- **AI**

# Graphics Processing Unit (GPU)

**Pixels**

- same operation performed independently on many small pieces of data => high degree of **parallelism**
- 4K image size: 3840x2160 = 8.5 Mpixels

Early GPUs: Hardware (fixed-function) accelerators for displaying pixels on screen

Over time  increased programmability => GPGPU (general-purpose GPU)

- NVIDIA CUDA 2007: Computing platform and API for **general-purpose parallel programming**

# Hardware acceleration

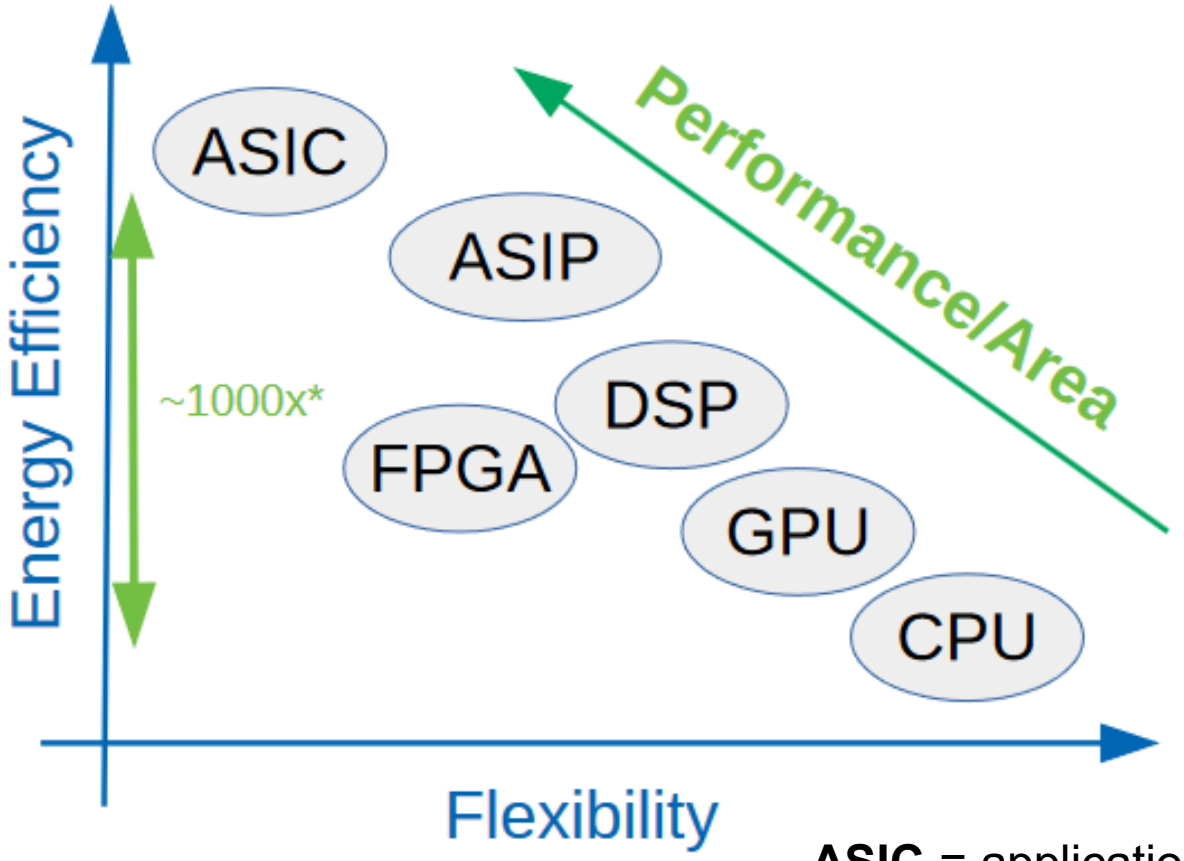
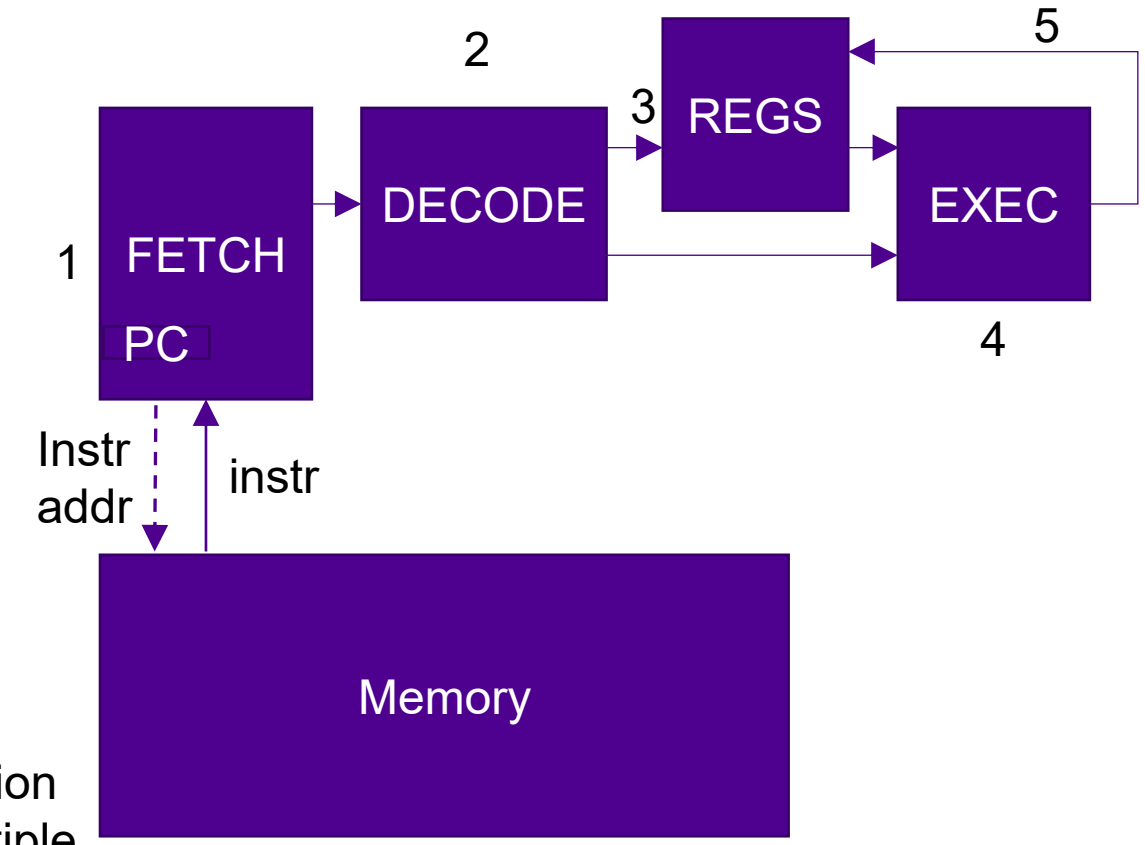



Image: Kanishkan Vadivel

**ASIC** = application-specific integrated circuit
**ASIP** = application-specific instruction-set processor
**DSP** = domain-specific processor
**FPGA** = field-programmable gate array

# Instruction Execution

1. Fetch instruction from memory
   - increment program counter (PC)
2. Decode instruction
3. Read operands from registers
4. Execute operation
   - modify PC if executing branch instruction
5. Write result to register




- Scalar CPU: The whole cycle is performed for one operation
- Fixed function accelerator: Can reuse some parts for multiple operations, or even skip them entirely.

# ISA vs. microarchitecture

- Instruction set architecture (ISA):
  - API of a processor
  - defines, e.g., which instructions and registers are available, what are the supported data types, etc.
  - determines the **result** of a program

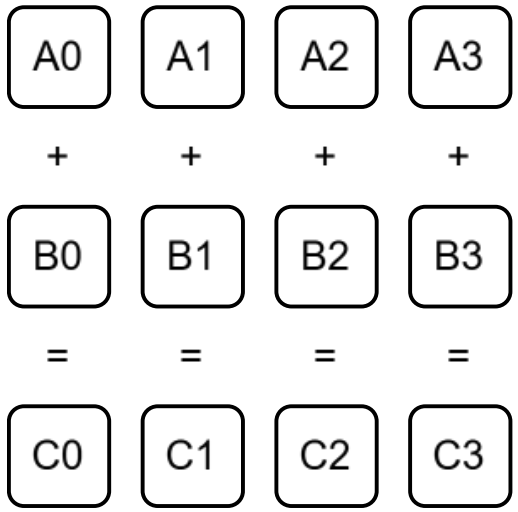
- Microarchitecture
  - implementation of ISA
  - e.g., AMD Zen or Intel Lion Cove are microarchitectures of an x86_64 ISA
  - changing microarchitecture changes **performance** but not the result

# Outline

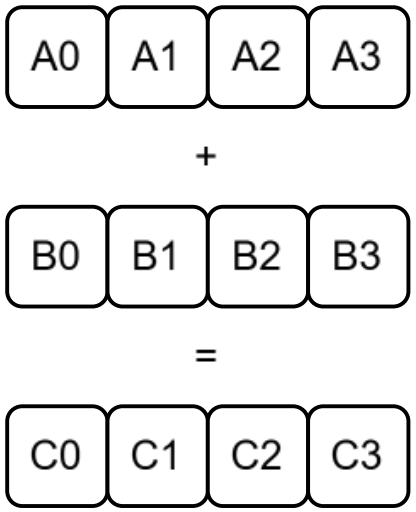
1. Programming model
2. Memory Hierarchy
3. System Integration

# Parallelism

serial



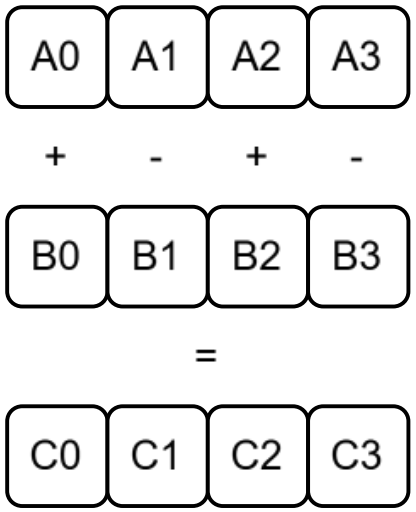

**Data-level parallelism (DLP)**




instr. fetch + decode shared among multiple data items

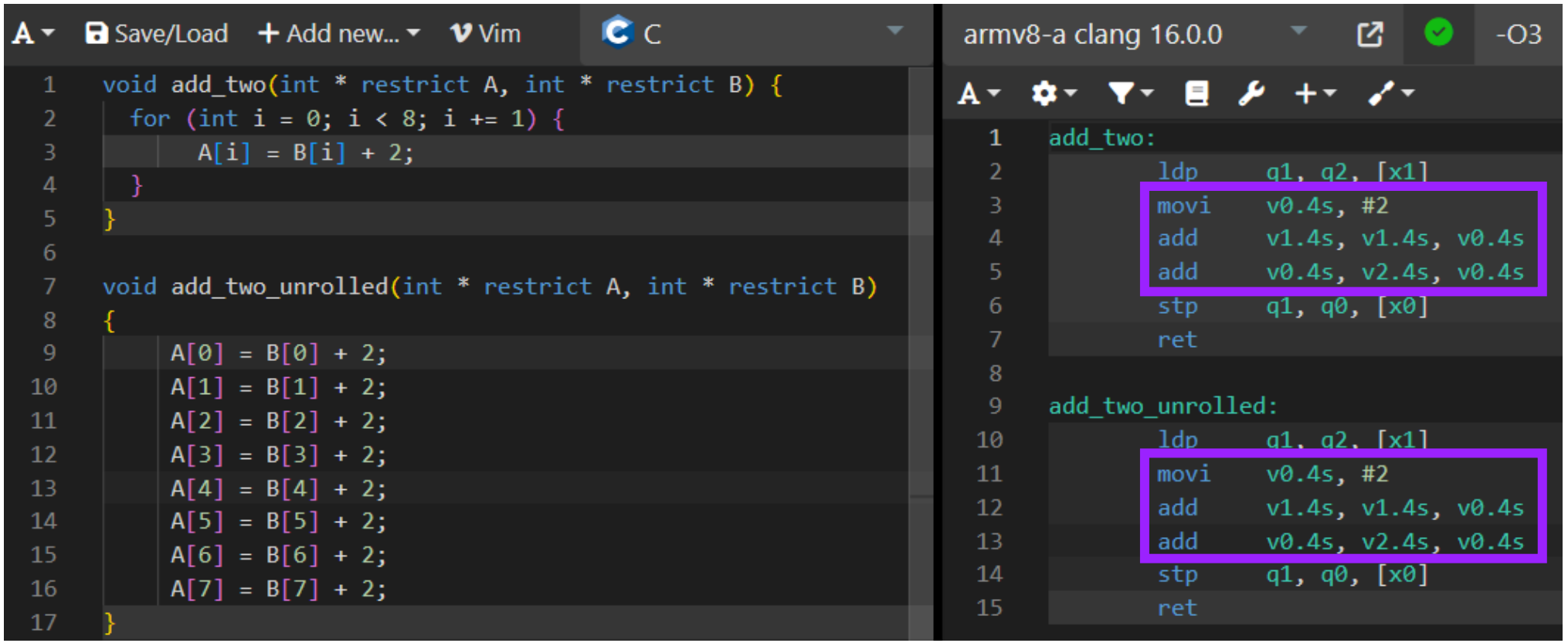
- modern GPUs

Instruction-level parallelism (ILP)



# Programming Model - CPU

Modern CPUs exploit DLP in the form of **single instruction multiple data (SIMD)** ISA extensions:



# Programming Model - GPU

GPUs are typically programmed using **single program multiple data (SPMD)** programming model

- **kernel**: a function that defines what happens for one "**work item**" (e.g., a pixel)
- Example OpenCL kernel:

```
kernel void add_two(int *A, int *B) {
  int i = get_global_id(0); // get id of the data item
  A[i] = B[i] + 2;
}
```

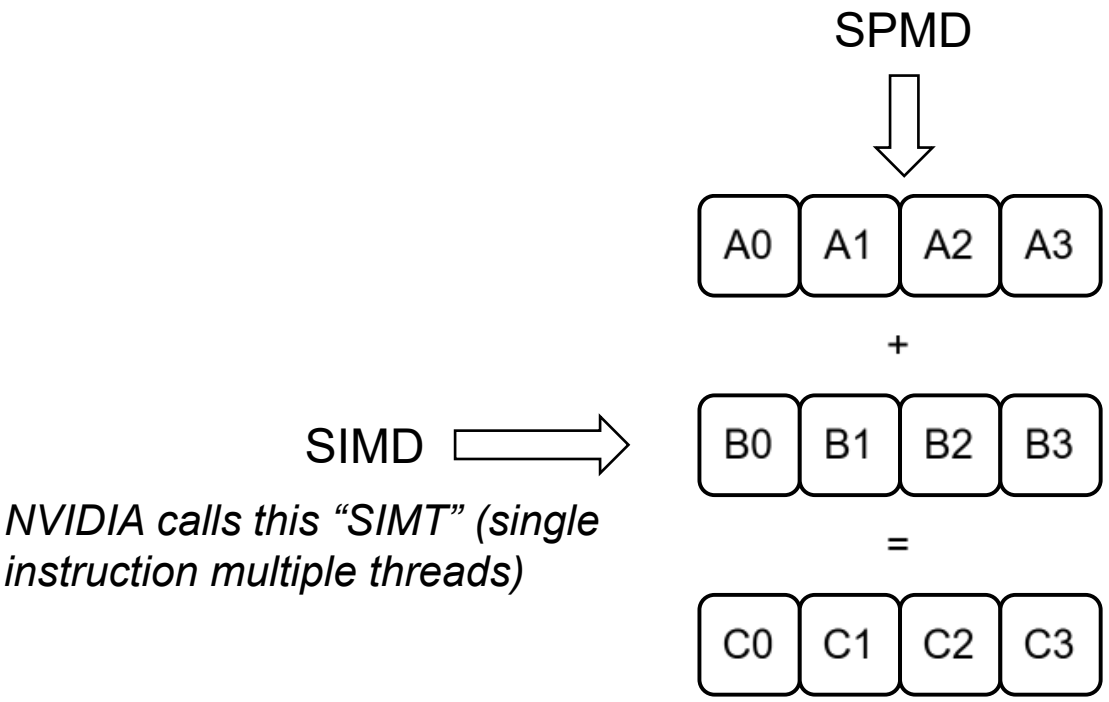
kernel execution replicated over all work items

SIMD: Defines **how** is DLP is achieved (i.e., via SIMD instructions)
SPMD: Defines **what** happens, the "how" is implementation-dependent; (can be SIMD, ILP, or even serial execution if no parallel resources)

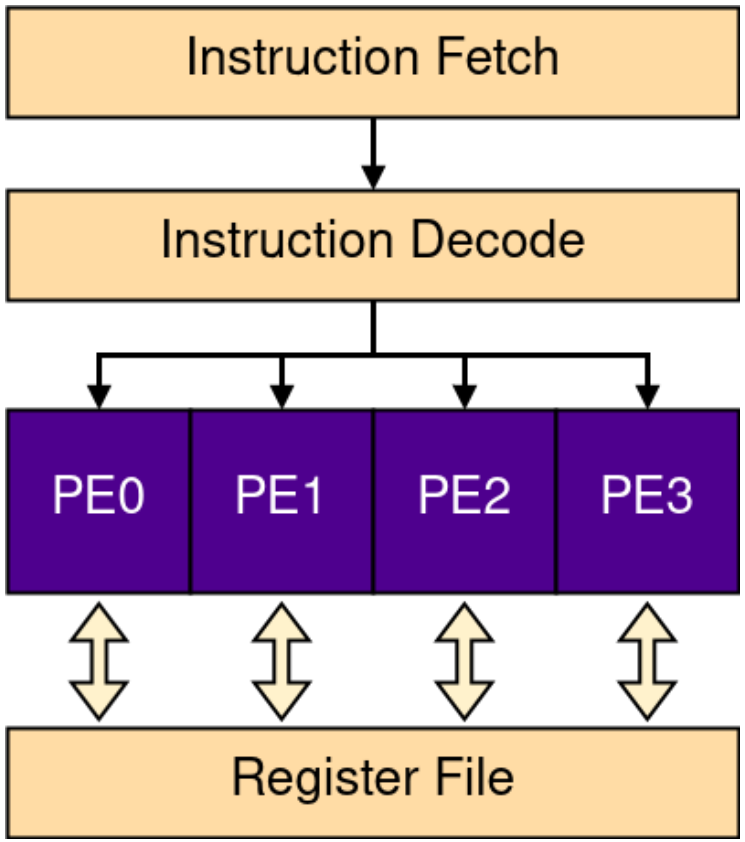
# Programming Model - GPU

On a GPU: kernel execution happens in a **lockstep**:

- Same instruction executed over work items (WI) in one “warp”/”wave”/”**subgroup**” (typically 32 or 64 on desktop, up to 128 on mobile GPUs)
- WIs within a subgroup share a program counter (PC)




*NVIDIA calls this “SIMT” (single instruction multiple threads)*

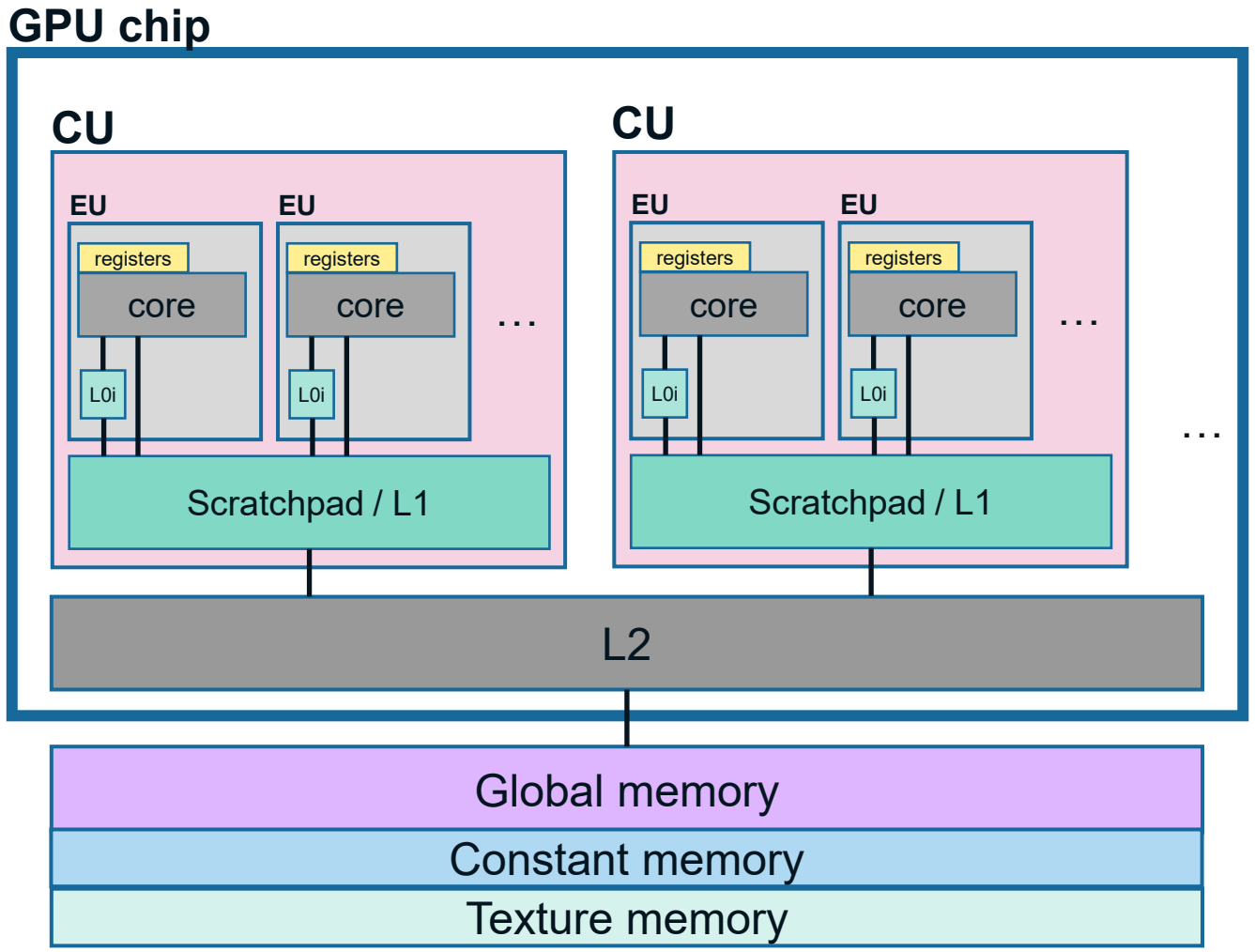


# Graphics Processing Unit

Warning: Confusing terminology ahead!

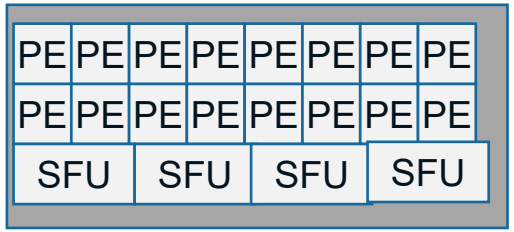
- Difference between **hardware abstraction** (what the part of GPU is called) and the **programming model** (API abstraction)
- I try to stick to vendor-independent terms used by Khronos and OpenCL
- Vendor-specific names (e.g., NVIDIA's) are included in a later slide

# Graphics Processing Unit




- **PE** (processing element): executes one **work item (WI)**
- **EU** (execution unit) runs one **subgroup** and contains:
  - **PEs**
  - special function units (SFUs), for example:
    - tensor cores
    - expensive math (f64, sqrt)
    - ray tracing cores
    - texture units
- **CU** (compute unit): multiple **EUs** sharing scratchpad/L1 cache
  - a **work group** is scheduled onto one CU
- **NDRange**: total number of **work items**
- *new: Thread Block Cluster (NVIDIA) – a level of hierarchy between grid and CU*

core (e.g., 16 PEs)



Tampere University

# SPMD – Example GPU Mapping

Example: Turing GPU microarchitecture (NVIDIA, 2018)


Full GPU – a hierarchy of processing elements


**Compute Unit**

- called “streaming multiprocessor”, SM (NVIDIA terminology)


**Execution Unit**

- called “processing block”, PB

local memory / L1

- called “shared memory”

images: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth

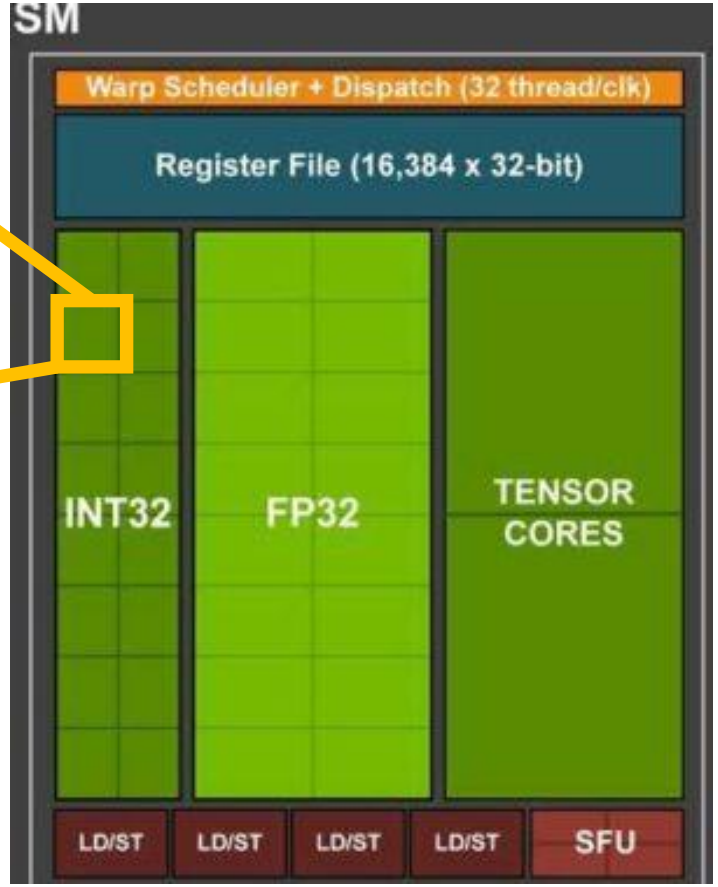
Tampere University

# SPMD – Example GPU Mapping

Example: Turing GPU microarchitecture (NVIDIA, 2018)

```
kernel add_two(int *A, int *B) {
  int i = get_global_id(0);
  A[i] = B[i] + 2;
}
```

x 16

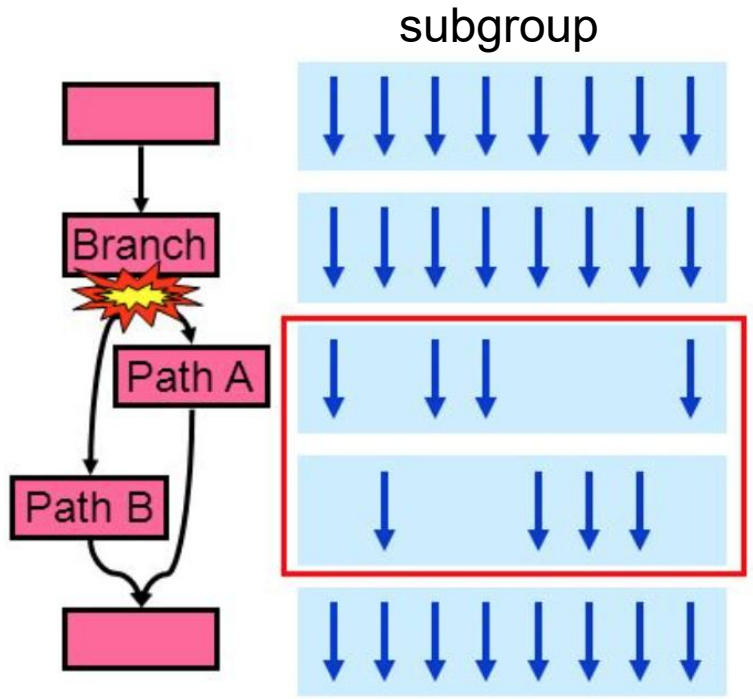



- On NVIDIA, subgroup (“warp”) size is 32 => One instruction is executed over **2 cycles**
- INT32 and FP32 operations can run concurrently => **instruction-level parallelism (ILP)** is possible by simultaneous execution of INT32 and FP32 instructions

images: https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth

# Problem - control flow

```
kernel void
diverging(int * restrict A,
  int * restrict B, int * restrict C)
{
    int i = get_global_id(0);
    int b = B[i];
    int c = C[i];
    if (b > c) {
        A[i] = b + 1;
    } else {
        A[i] = c - 1;
    }
}
```




- Each work item may take a different if-else branch
  - (b > c) may be true for i=0 but false for i=1

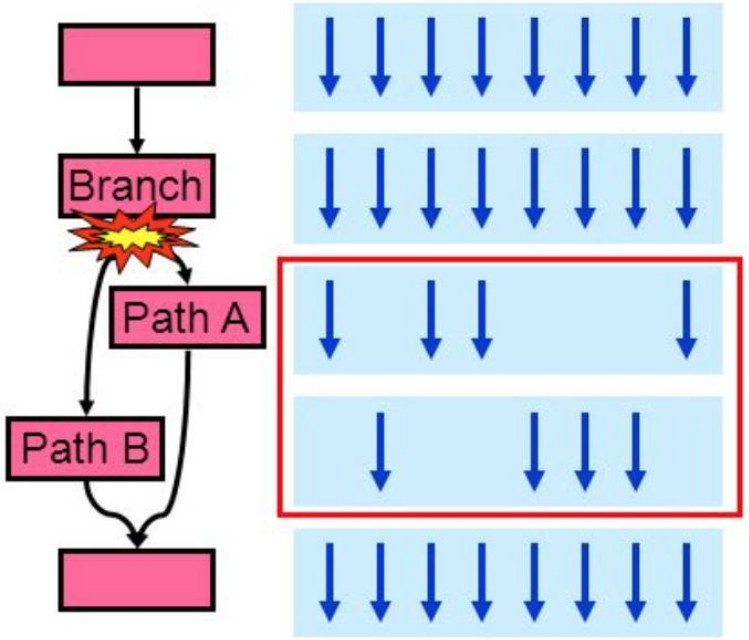
But all work items in a subgroup must execute the same instruction!

Tampere University

# Predication

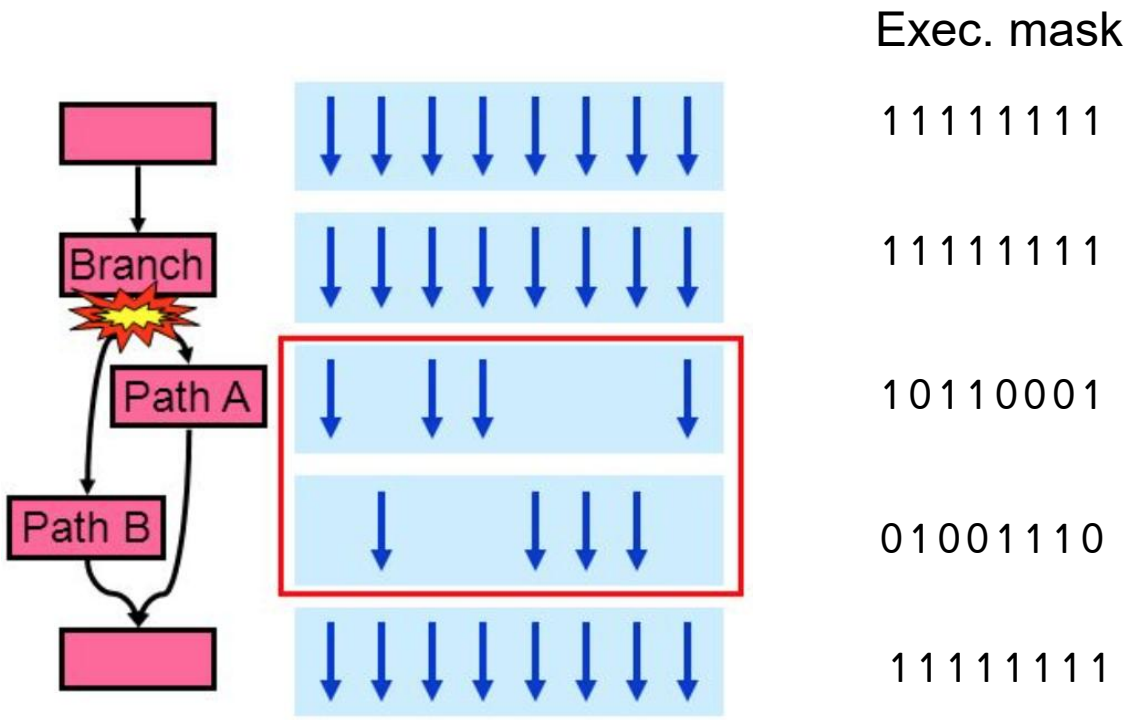
- If-conversion: replacing diverging “if-else” statement with a linear code
  - E.g., using max() built-in instead of if/else (not always applicable)

- **Predication**
  - Execute both paths in all WIs and ignore unused WIs
  - Number of clock cycles to execute the branch statement is roughly PathA + PathB



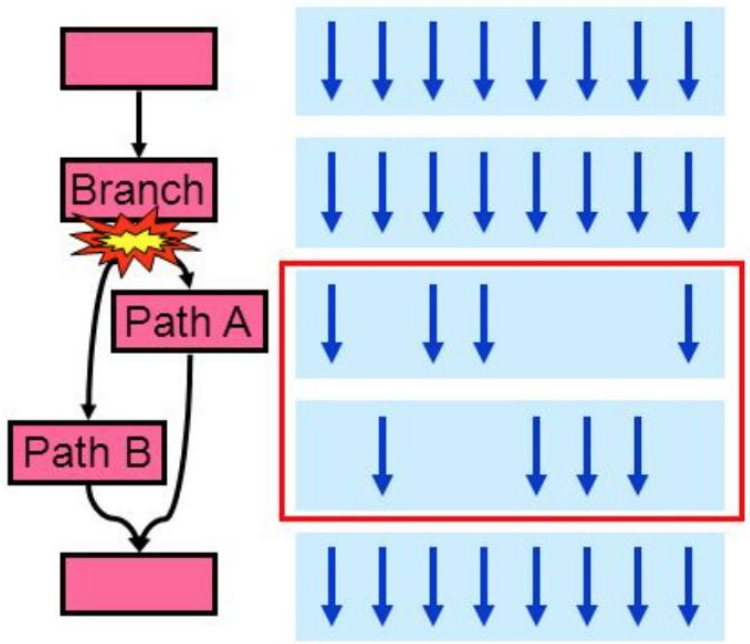
# Execution mask

- A register that holds N bits (N = subgroup size) telling which WIs are disabled/enabled
  - Operations are performed on all WIs
  - Only “enabled” WIs participate in the result (write to register, write to memory, …)



# Execution mask

- Stack
  - Push exec. masks on stack to allow nested control flow
  - Maximum stack depth: needs to be checked by driver/compiler




Exec. mask

11111111

11111111

10110001

01001110

11111111

# Throughput vs. Latency

- CPU: Latency machine*
  - Focus on fast execution of single-threaded code
  - Low latency, low throughput
- GPU: Throughput machine*
  - Focus on raw processing speed (FLOPS)
  - High latency, high throughput
  - Instructions are often batched together to maximize throughput
- The latency/throughput tradeoff is reflected in the **memory hierarchy** design of CPUs and GPUs

*does not apply 100%, modern CPUs and GPUs have features that make them converge to each other

# Scoreboard and Latency Hiding

- **Scoreboard** = “wait list” for operands/results of operations
- When a long operation is performed (e.g., memory access, slow math operation), it is marked in scoreboard as being “in progress”
- Subgroup scheduler then uses it to schedule other instructions to run in the meantime while the original subgroup is waiting
  - this is known as **latency hiding**
  - To be efficient, multiple subgroups should be running on one EU, to allow switching context and hide the latencies

# Workload-specific blocks

More or less fixed function accelerators of a GPU*

- Graphics
  - texture fetch, ray tracing units
- AI
  - “tensor cores” = multiply-accumulate accelerators
  - Low precision (down to 4-bit integer / floating point)
- other
  - FP64, special operations (trigonometry, sqrt, …)

Typically fewer of them than regular PEs

- They use the “wait list” mechanism of the **scoreboard**

**If GPU is an accelerator for a CPU, then they are an accelerator of an accelerator*

# Terminology cheatsheet

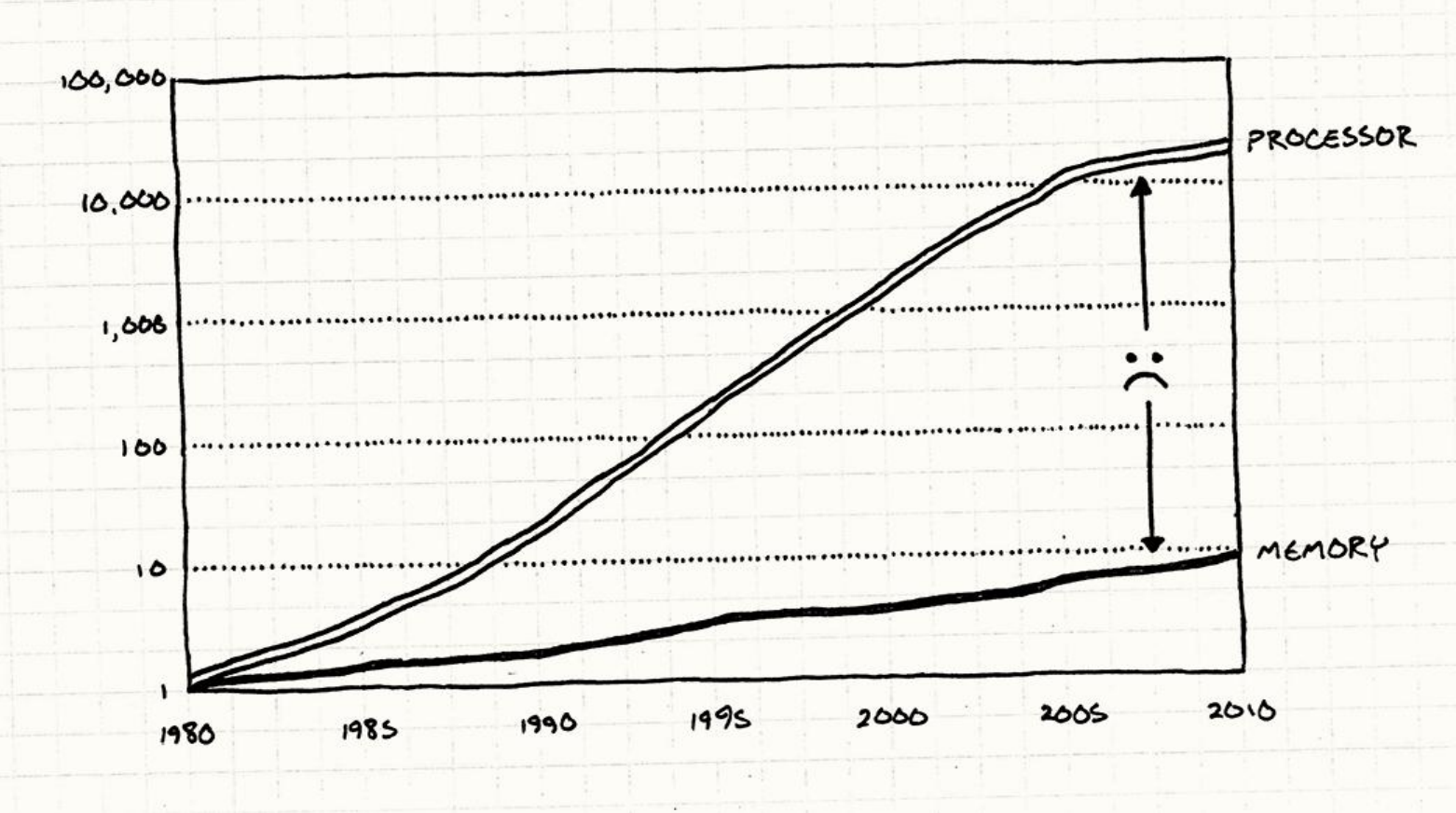
| Vendor-independent | Vendor-specific |
|---|---|
| Work Item (WI)* | Thread (NVIDIA) |
| Private memory | Local memory (NVIDIA) |
| Local memory, scratchpad | Shared memory (NVIDIA, graphics APIs) |
| | Local Data Share (LDS, AMD) |
| | Shared Local Memory (SLM, Intel) |
| Work Group (WG)* | Thread block (NVIDIA) |
| Subgroup*** | Warp (NVIDIA), wave/wavefront (AMD) |
| Execution Unit (EU) | Processing Block (PB, NVIDIA) |
| Compute Unit (CU)** | Streaming Multiprocessor (SM, NVIDIA) |
| | Xe-Core (Intel) |
| NDRange | Grid (NVIDIA) |

* used also by Intel, AMD, ** used also by AMD, *** used also by Intel

# Outline

1. Programming model
2. Memory Hierarchy
3. System Integration

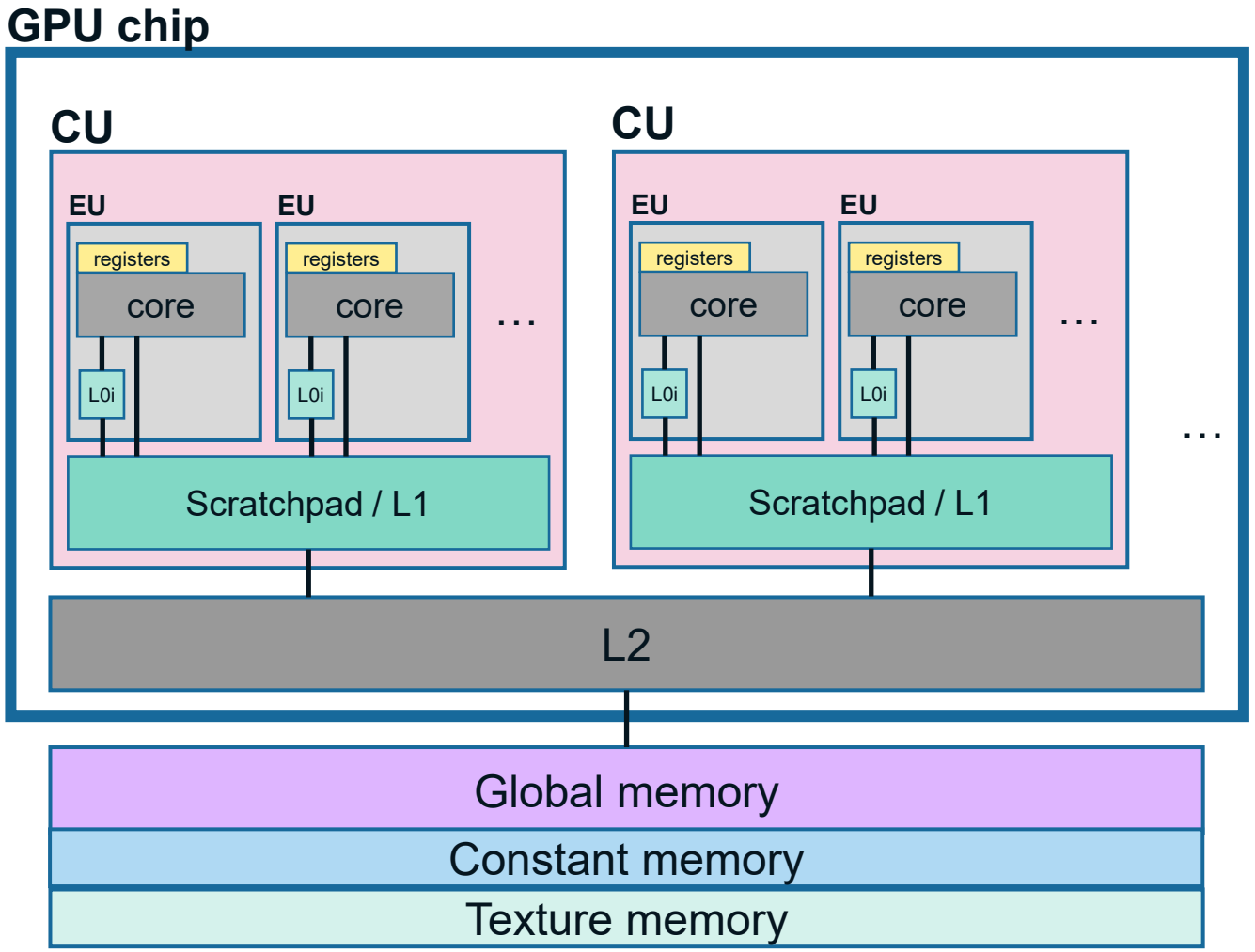
# The “Memory Wall”




Processor and RAM speed relative to their respective speeds in 1980, http://gameprogrammingpatterns.com/data-locality.html

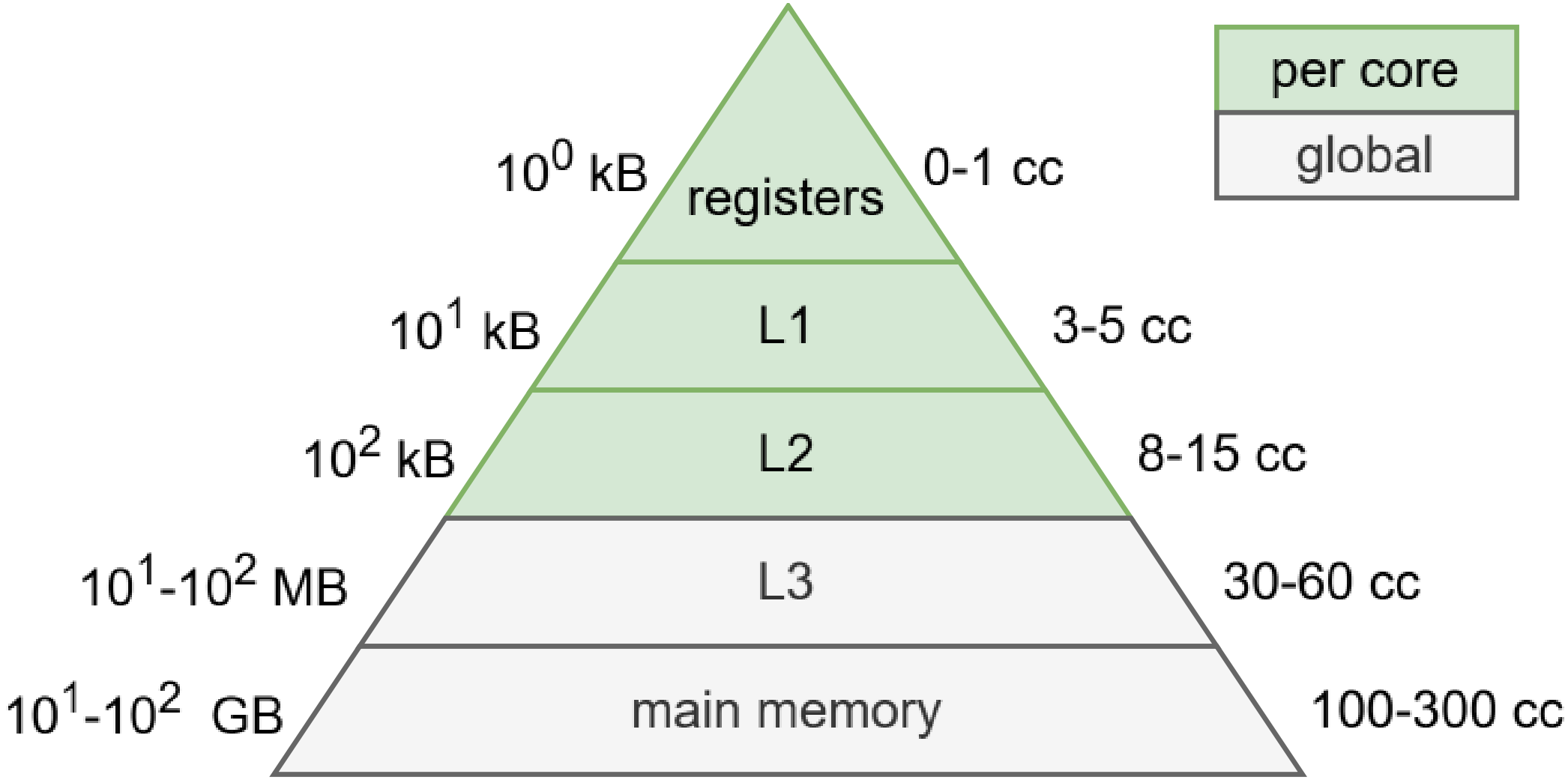
Applies to both CPUs and GPUs

# GPU Memory Hierarchy

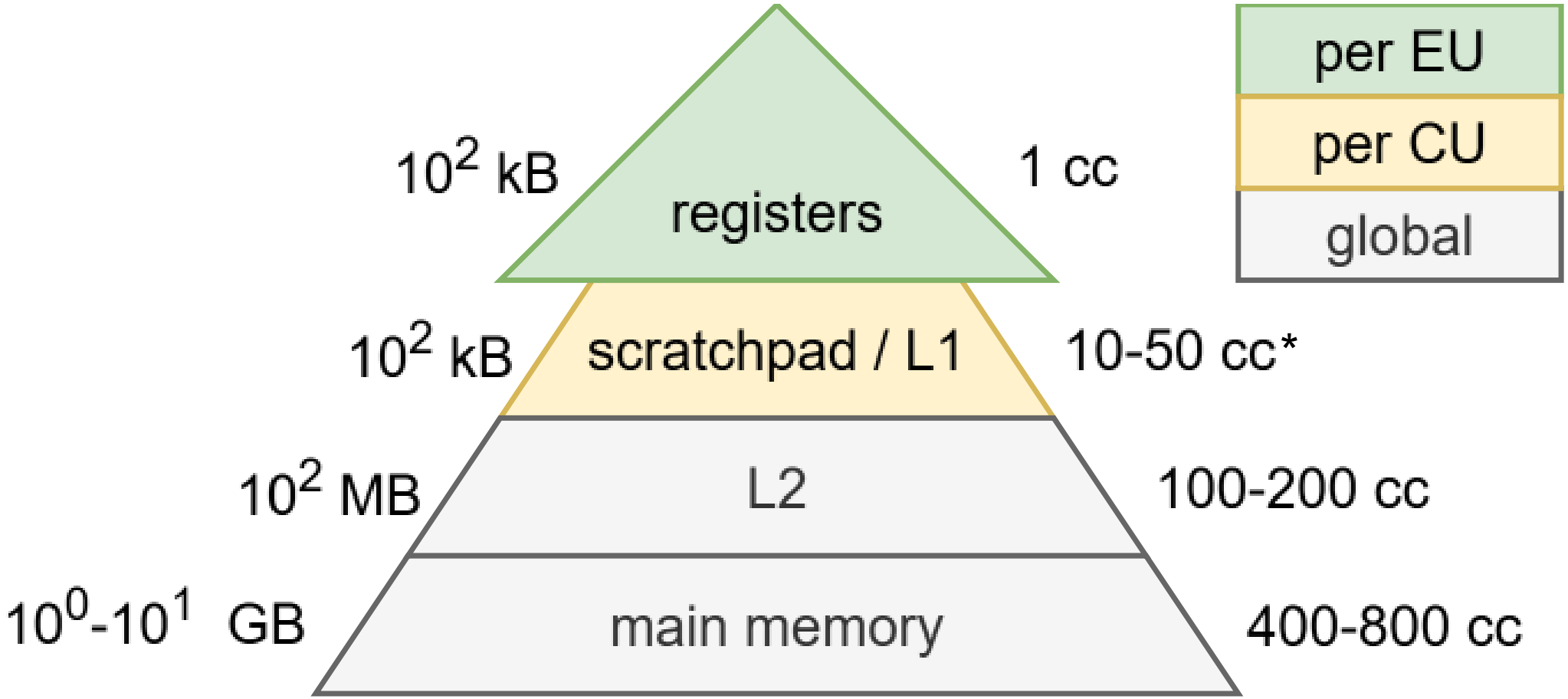



- L0i – private instruction cache
- Tradeoff between software-managed RAM (scratchpad) and hardware-managed cache (L1)
  - sometimes, they occupy the same SRAM and sometimes the ratio can be chosen
  - sometimes, they are physically separated
  - textures and register spills can be cached in L1/L2
  - for constants, a separate read-only cache is used
- Some GPUs can have L3 cache (e.g., AMD)
- GPUs have a separate read-only constant & texture memory

# CPU Memory Hierarchy



# GPU Memory Hierarchy




*(scratchpad may have lower latency than L1 cache)

# GPU Memory Hierarchy

| | CPU | GPU |
|---|---|---|
| Cache Line | 64B | Can be different* |
| Programmer Transparency | HW-managed | Both HW- and SW-managed |
| Latency Hiding | Prefetching, OoO** | Many subgroups in flight |
| Latency | Low | High |
| Bandwidth | Low | High |

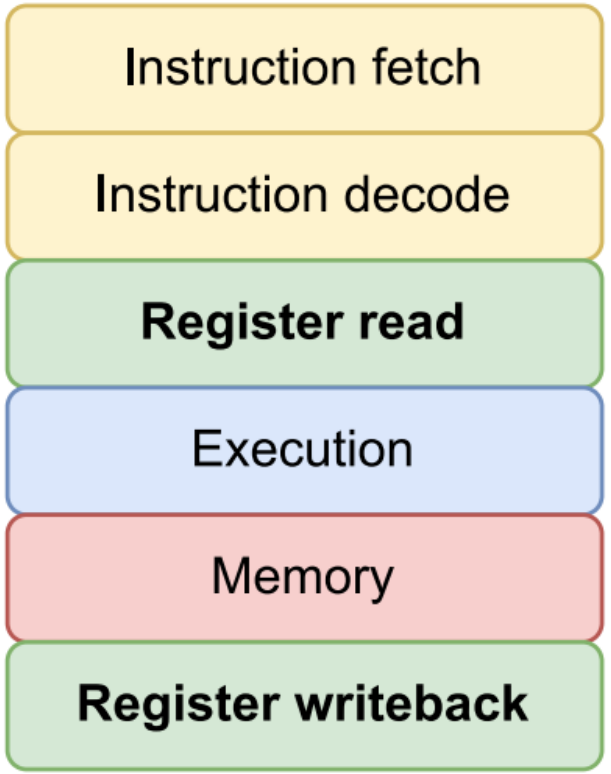
* For example, on some NVIDIA architectures, 128B is equal to one subgroup of floats (32x4B)
** Out-of-Order execution = executing instruction in a different order than what is in the instruction memory

# Registers

- Fastest storage, limit how many subgroups can stay active
  - More active subgroups => more latency hiding
- Large register files needed for fast context switching between subgroups
  - That's why total register capacity in GPUs can exceed total capacity of scratchpad/L1 or even L2

If subgroup requires too many registers:

1. Lower the number of concurrent subgroups (= lower occupancy)
2. **Spill** values to memory hierarchy (global memory, but can be cached in L1/L2)



# Register Spilling

Data private to a WI is *mostly* stored in registers.

**Spilling** (= moving values from registers to memory hierarchy) occurs when:

- WI uses too many registers
- Data structures like arrays are dynamically indexed because registers cannot be indexed:

```
kernel void private_array_spill(global float *out, const uint seed)
{
    size_t gid = get_global_id(0);
    float tmp[16];
    for (int i = 0; i < 16; ++i) {
        tmp[i] = (float)(gid + i);
    }

    uint idx = (uint)((gid ^ seed) & 0xff);
    float x = tmp[idx];

    out[gid] = x;
}
```

# GPU Performance

Performance is limited either by

- **compute throughput** (often expressed as FLOPS = floating point operations per second)
  - *(do not confuse with FLOPs = floating point operations!)*

or

- **memory bandwidth** (e.g. bytes per second)
  - due to the memory wall, this case is typically more common

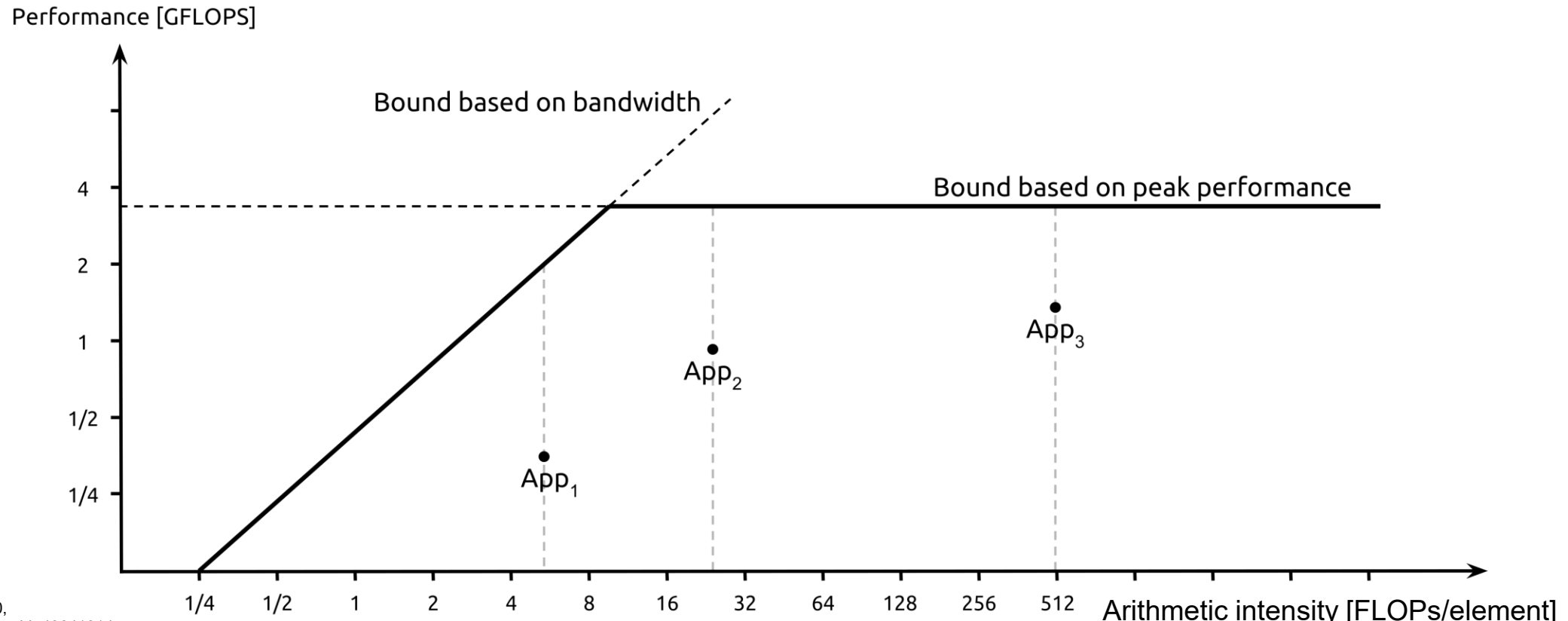
Tampere University

# Arithmetic Intensity

Ratio of the amount of computation (**W** [FLOPs]) to the number of memory accesses (**Q** [bytes or elements])

**I = W / Q** [FLOPs/element]

**Roofline model**



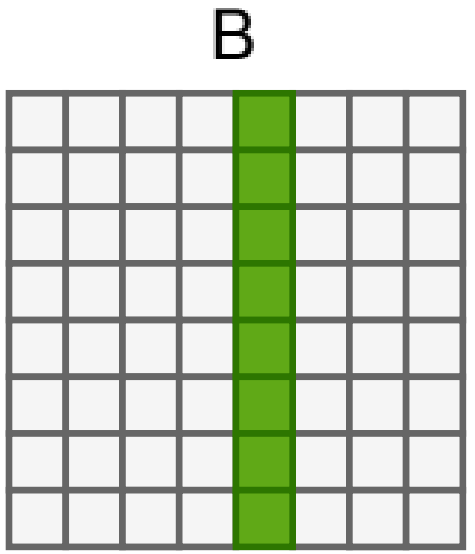
# GEMM (general matrix-matrix multiplication)

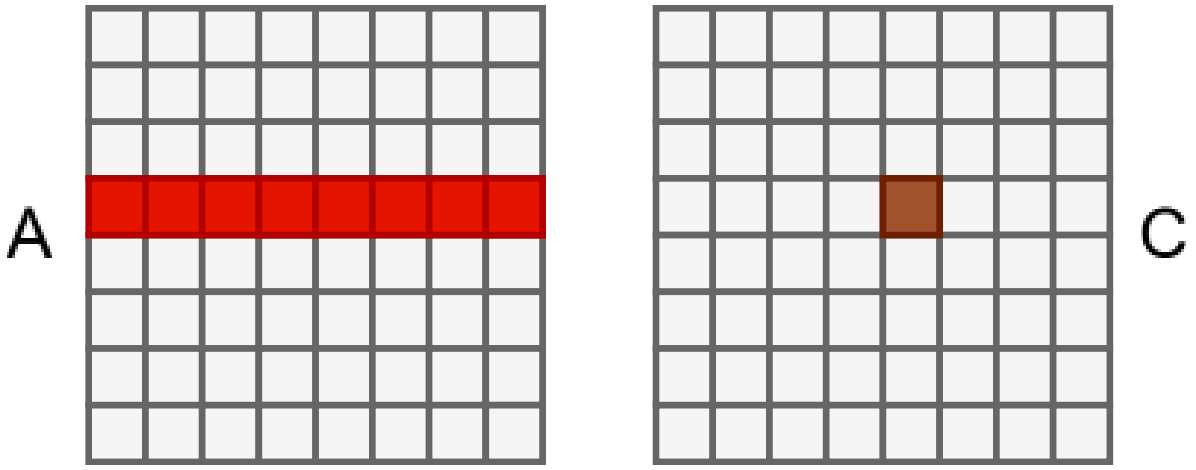
Memory-bound kernel:
Dot product (= computing 1 element of matrix-matrix multiplication, size NxN)

- 1 WI per element of C

per WI:
W = **2N-1** FLOPs (N-1 additions, N multiplications)
Q = **2N+1** elements (load row of A and column of B, store result)

I = (2N-1) / (2N+1) =~ **1** [FLOPs/element]



# GEMM

Compute bound kernel: Full matrix-matrix multiplication

- Entire matrix in 1 WI

$$C = A \times B$$

W = N^2 * (2N-1) =~ **2N^3** FLOPs (N-1 additions, N multiplications per C[i,j])
Q = **3*N^2** elements (load 2 matrices, store 1 matrix)

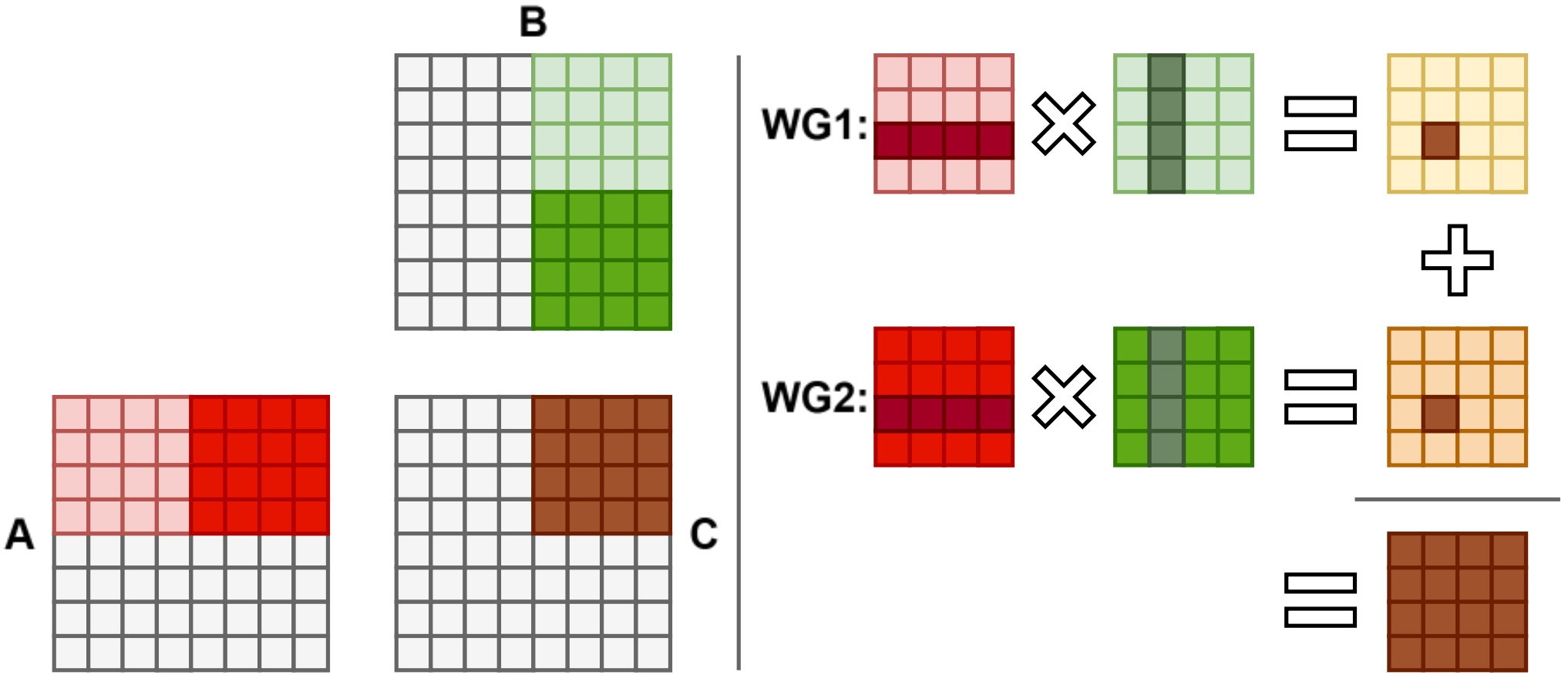
I =~ 2N^3 / 3N^2 = **2N / 3** [FLOPs/element]

If N is sufficiently large, kernel is compute bound, however, we are calculating everything as a scalar workload in a single PE => **underutilization of compute resources!** (unless we have enough matrices)

# Tiled GEMM

Tiled matrix-matrix multiplication (matrix size NxN, tile size TxT)

- 1 WI per element of C
- Each work group (WG) cooperatively loads tiles of A and B into local memory and computes a tile of C



Tampere University

# Tiled GEMM

Only **two** global memory reads per tile per work item!

Loading data from local (=fast) memory

```
// tile size
#define T 32

__kernel void tiled_gemm(const int N,
                         const __global float* A,
                         const __global float* B,
                         __global float* C) {

    // Thread identifiers
    const int row = get_local_id(0); // Local row ID (max: T )
    const int col = get_local_id(1); // Local col ID (max: T )
    const int globalRow = T*get_group_id(0) + row; // Row ID of C (0..N)
    const int globalCol = T*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[T][T];
    __local float Bsub[T][T];

    // Initialise the accumulation register
    float acc = 0.0f;
```

```
    // Loop over all tile pairs used for calculating this element
    const int numTiles = N/TS;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        const int tiledRow = T*t + row;
        const int tiledCol = T*t + col;
        Asub[col][row] = A[tiledCol*N + globalRow];
        Bsub[col][row] = B[globalCol*N + tiledRow];

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform the computation for a single tile
        for (int k=0; k<T; k++) {
            acc += Asub[k][row] * Bsub[col][k];
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final result in C
    C[globalCol*N + globalRow] = acc;
}
```

source: adapted from https://cnugteren.github.io/tutorial/pages/page4.html

# Tiled GEMM

Synchronization: **barrier**!

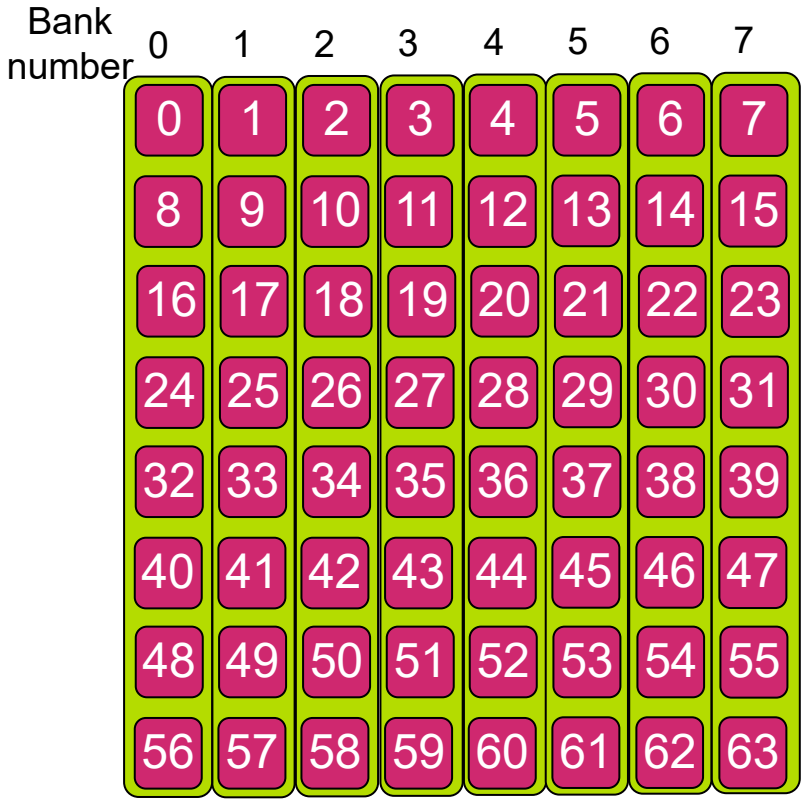
- Ensure all WIs within this work group reach this point before proceeding

```
// tile size
#define T 32

__kernel void tiled_gemm(const int N,
                         const __global float* A,
                         const __global float* B,
                         __global float* C) {

    // Thread identifiers
    const int row = get_local_id(0); // Local row ID (max: T )
    const int col = get_local_id(1); // Local col ID (max: T )
    const int globalRow = T*get_group_id(0) + row; // Row ID of C (0..N)
    const int globalCol = T*get_group_id(1) + col; // Col ID of C (0..N)

    // Local memory to fit a tile of TS*TS elements of A and B
    __local float Asub[T][T];
    __local float Bsub[T][T];

    // Initialise the accumulation register
    float acc = 0.0f;
```

```
    // Loop over all tile pairs used for calculating this element
    const int numTiles = N/TS;
    for (int t=0; t<numTiles; t++) {

        // Load one tile of A and B into local memory
        const int tiledRow = T*t + row;
        const int tiledCol = T*t + col;
        Asub[col][row] = A[tiledCol*N + globalRow];
        Bsub[col][row] = B[globalCol*N + tiledRow];

        // Synchronise to make sure the tile is loaded
        barrier(CLK_LOCAL_MEM_FENCE);

        // Perform the computation for a single tile
        for (int k=0; k<T; k++) {
            acc += Asub[k][row] * Bsub[col][k];
        }

        // Synchronise before loading the next tile
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    // Store the final result in C
    C[globalCol*N + globalRow] = acc;
}
```

source: adapted from https://cnugteren.github.io/tutorial/pages/page4.html

# Memory banks



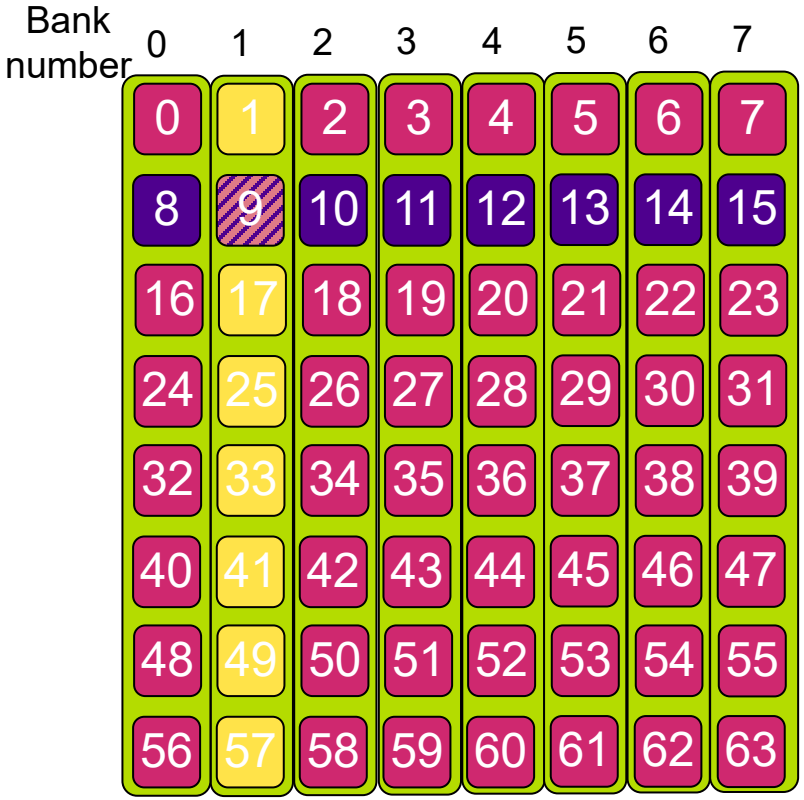

White number = integer address

- Memories are usually divided into banks
- Bank = internal structure of smaller memories capable of working in parallel
  - The number of banks = the number of possible parallel accesses
- Important to avoid **bank conflicts**
  - = multiple memory accesses accessing the same bank, common when using local memory

- for example, L1 of CPU can have 8 banks, scratchpad of GPU can have 32 banks

Tampere University

# Memory bank conflicts




White number = integer address

= column access  = row access

Imagine this running on multiple WIs of a subgroup:

## Recall tiled GEMM

- naïve implementation would iterate one tile column-wise => risk of bank conflict!

```
const int row = get_local_id(0); // Local row ID (max: T )
const int col = get_local_id(1); // Local col ID (max: T )
```

```
// Loop over all tile pairs used for calculating this element
const int numTiles = N/TS;
for (int t=0; t<numTiles; t++) {

    // Load one tile of A and B into local memory
    const int tiledRow = T*t + row;
    const int tiledCol = T*t + col;
    Asub[col][row] = A[tiledCol*N + globalRow];
    Bsub[col][row] = B[globalCol*N + tiledRow];

    // Synchronise to make sure the tile is loaded
    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the computation for a single tile
    for (int k=0; k<T; k++) {
        acc += Asub[k][row] * Bsub[col][k];
    }
```

To avoid bank conflict, Bsub is stored **transposed**

Asub[k][row…row+31] *contiguous* access

Bsub[col][k] *broadcast*

# Memory Access Patterns

Global memory optimization:

- Coalescing: *WIs request memory in a **pattern** that the hardware (coalescer) can merge into a few big accesses.*
  - For example, instead of 8 individual 4B memory transactions, one 32B transaction is issued

Local memory optimization:

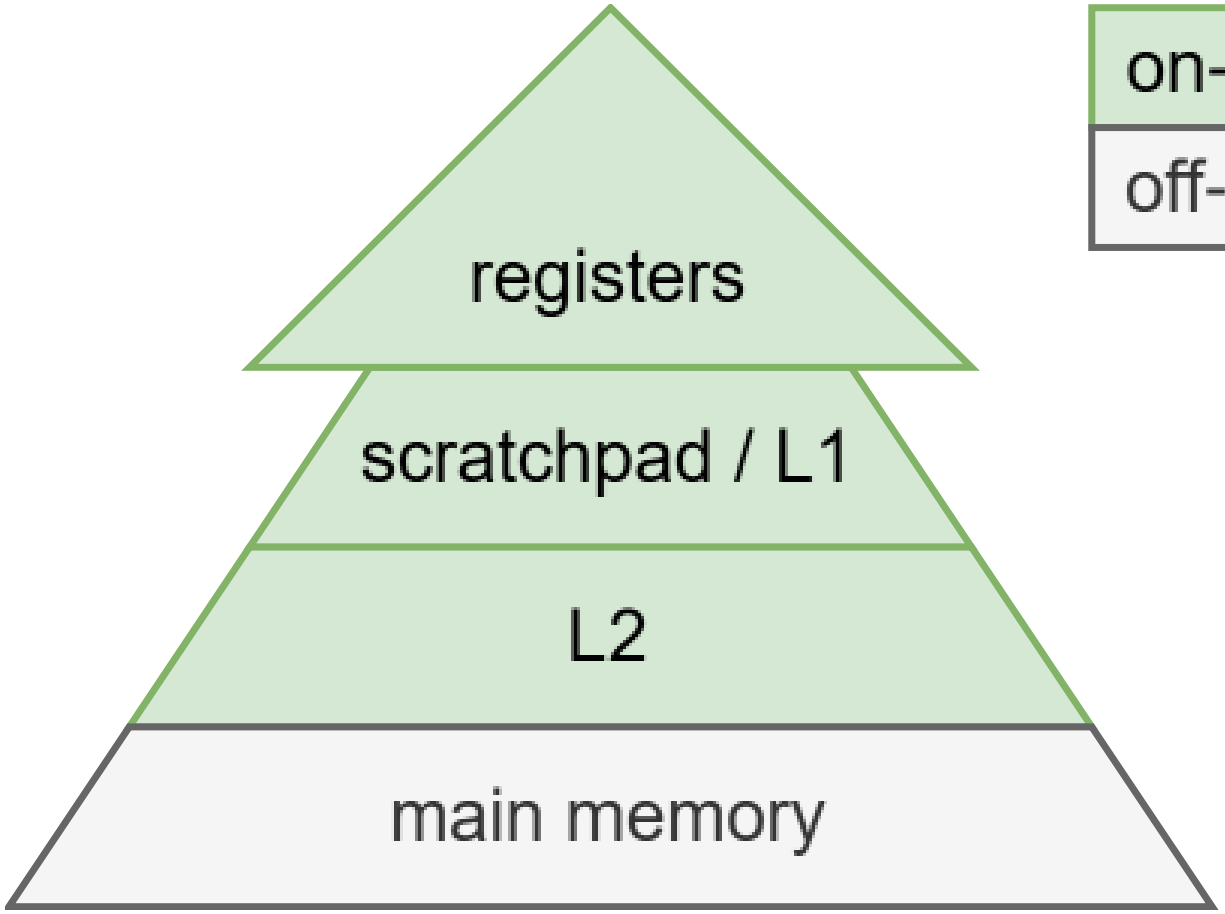
- Avoiding memory bank conflicts: *WIs request memory in a **pattern** that accesses different banks.*

The most optimal **memory access pattern** is typically contiguous (= consecutive addresses).

# Outline

1. Programming model
2. Memory Hierarchy
3. System Integration

# GPU Memory Hierarchy




on-chip SRAM

off-chip DRAM

SRAM: Static Random Access Memory
- optimized for speed

DRAM: Dynamic Random Access Memory
- optimized for density (byte per area)

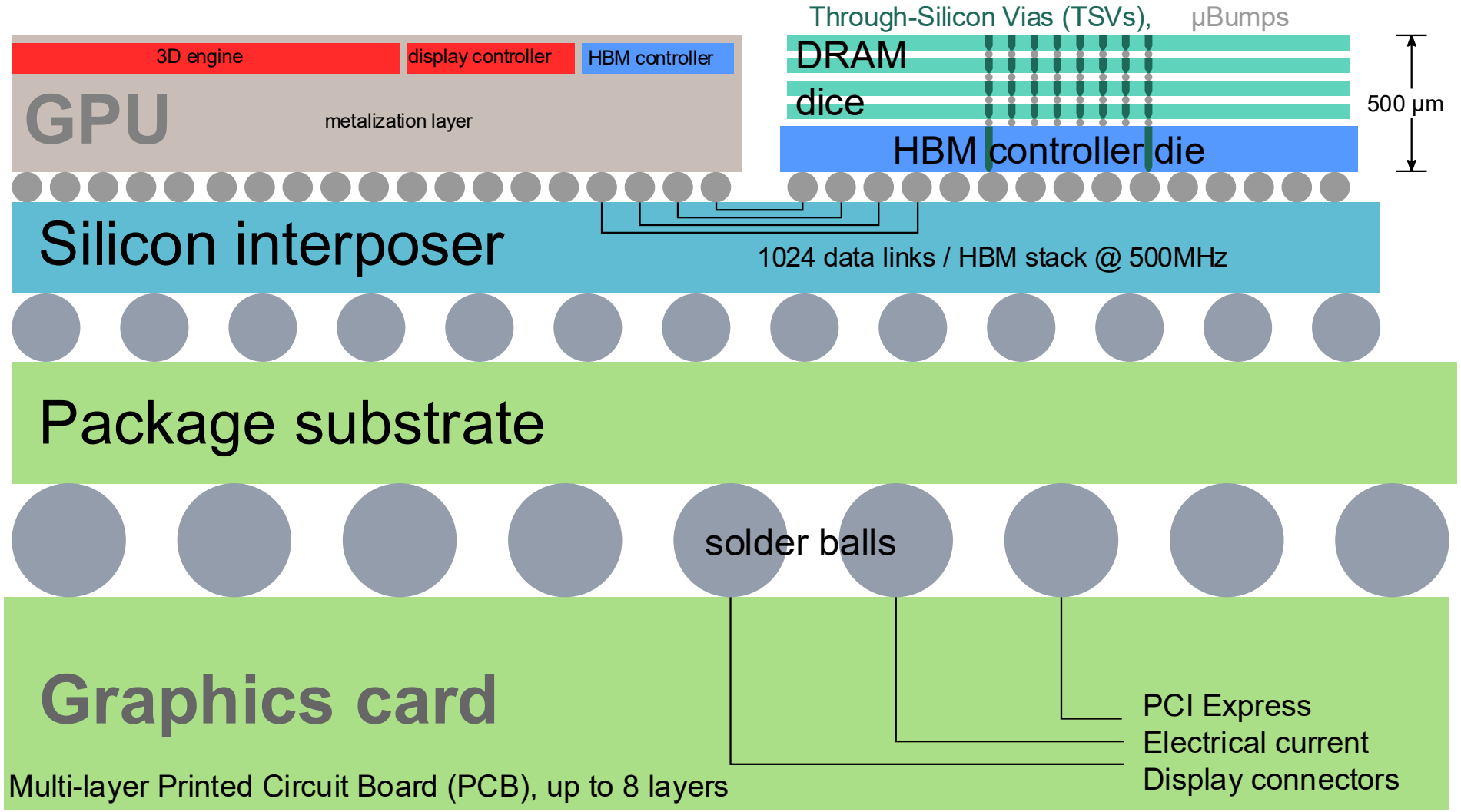
# DRAM Memory Technologies

- integrated GPU (iGPU): **DDR** (double data rate) or **LPDDR** (low-power DDR)
  - LPDDR5X: **8.5 Gb/s/pin** (32-128 pins => **34-136 GB/s**)
  - DDR5: **6.4 Gb/s/pin** (64-128 pins => **51-102 GB/s**)

- desktop GPU: **GDDR** (graphics DDR)
  - GDDR7**: 32 Gb/s/pin** (128-384 pins => **0.5-1.5 TB/s**)

- datacenter GPUs: **HBM** (high-bandwidth memory)
  - HBM4**: 8 Gb/s/pin** (2048 pins per stack => **2 TB/s** per stack; up to 8 stacks)

# High Bandwidth Memory




By Shmuel Csaba Otto Traian, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=46385502

# DRAM Memory Technologies

- integrated GPU (iGPU): **DDR** (double data rate) or **LPDDR** (low-power DDR)
  - LPDDR5X: **8.5 Gb/s/pin** (32-128 pins => **34-136 GB/s**)
  - DDR5: **6.4 Gb/s/pin** (64-128 pins => **51-102 GB/s**)

- desktop GPU: **GDDR** (graphics DDR)
  - GDDR7**: 32 Gb/s/pin** (128-384 pins => **0.5-1.5 TB/s**)
  - (CPU<->GPU) PCIe 6.0 x16: **256 GB/s** bidirectional

- datacenter GPUs: **HBM** (high-bandwidth memory)
  - HBM4**: 8 Gb/s/pin** (2048 pins per stack => **2 TB/s** per stack; up to 8 stacks)
  - (CPU<->GPU) PCIe 6.0 x16: **256 GB/s** bidirectional
  - (GPU<->GPU) AMD Infinity Fabric x16: **896 GB/s** (aggregate over 7 links)
  - (GPU<->GPU) NVIDIA NVLink: **1.8 TB/s** (aggregate over 18 links)

# Interconnect Data Movement

Direct Memory Access (DMA):

- HW units moving data directly between host* (CPU) RAM and GPU RAM without using CPU

Pinned memory:

- GPU keeps a buffer resident in host memory, prevents the OS from paging it out, can be directly accessed by DMA

** "host" typically means CPU, but on Raspberry Pi, bootloader resides on a GPU and is copied to CPU on startup*

# Chiplets

- Huge monolithic chips hard to manufacture with good yield

⇒ Partitioning a chip into separate sub-chips (**chiplets**) connected with fast interconnect

- Used both for CPU and GPU manufacturing

# AMD MI300X

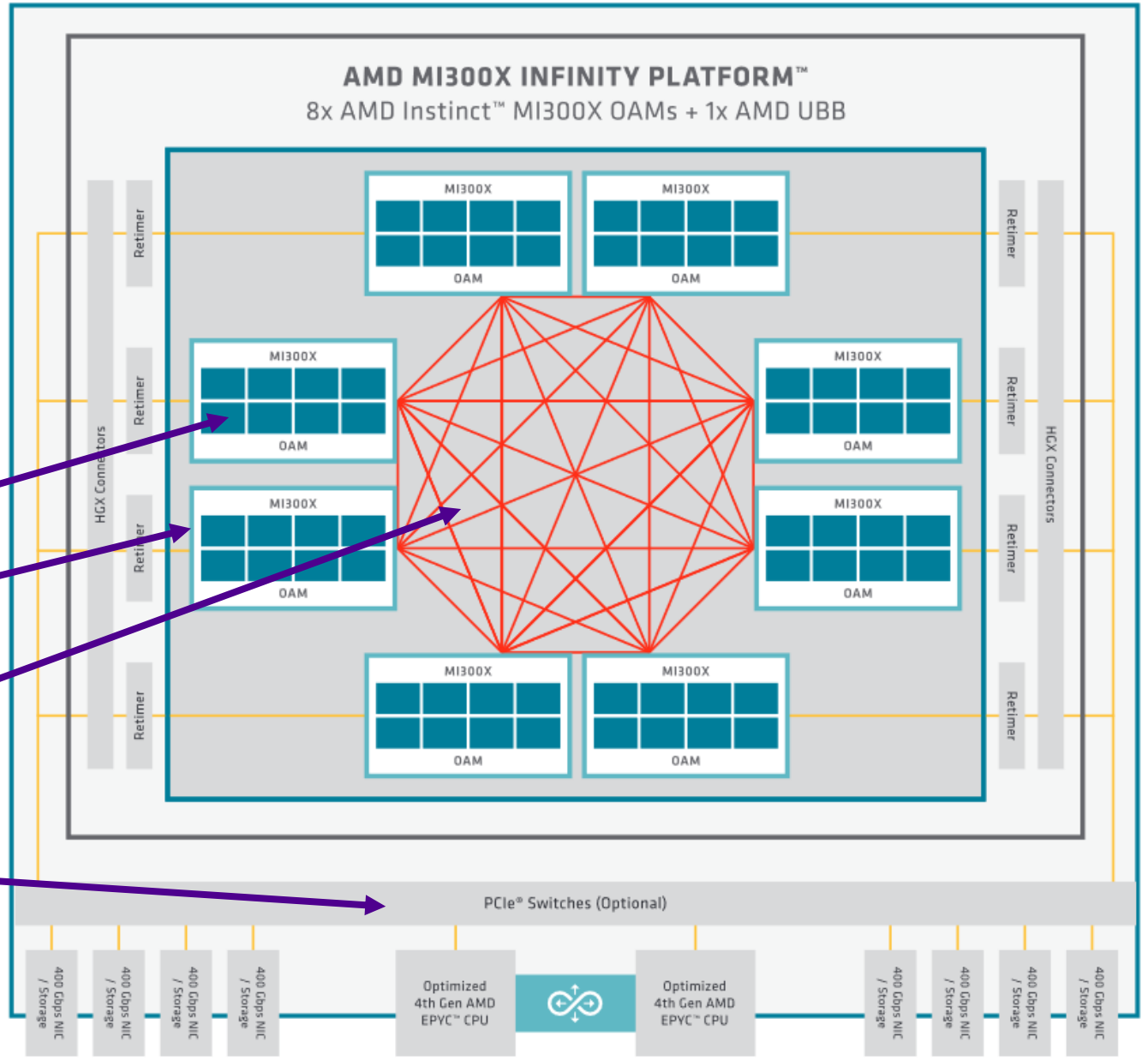
MI300X Infinity Platform (8x MI300X GPUs in a single package for AI workloads)

chiplets

separate GPUs

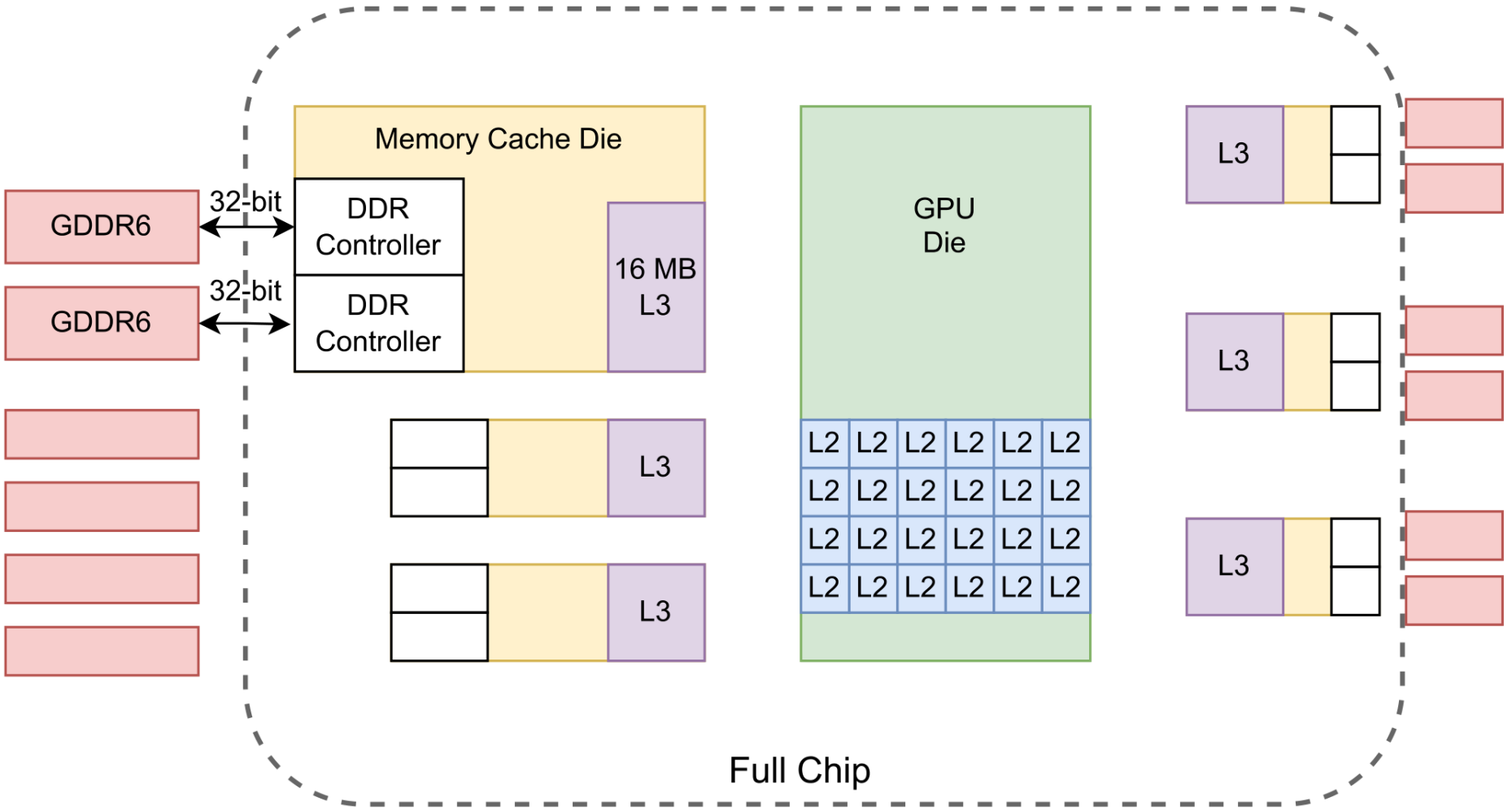
AMD Infinity Fabric

PCIe 5.0




https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/white-papers/amd-cdna-3-white-paper.pdf

# Chiplet Caches




AMD RDNA3 GPU chip contains six L3 cache chiplets

image: Topi Leppänen

# Hardware Lottery

*“a research idea wins because it is compatible with available software and hardware and not because the idea is superior to alternative research directions”*

Hooker, Sara. "The hardware lottery." Communications of the ACM 64.12 (2021): 58-65.

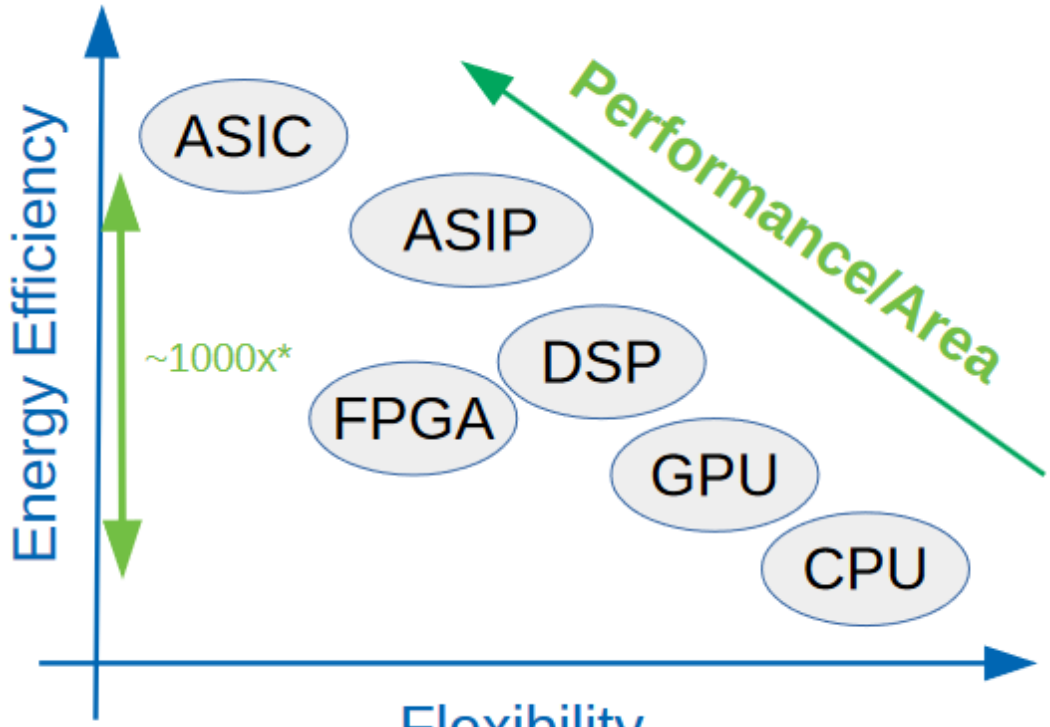



Image: Kanishkan Vadivel

**Q: Which ideas or problems are made easier / harder by GPUs?**

# Summary

- GPU = “throughput machine”
- Programming model: SPMD (what happens for one work item (WI))
- WIs within subgroup executed in lockstep (effectively achieving SIMD)
- Major bottleneck: Memory traffic
  - watch the arithmetic intensity
  - => reduce memory traffic by using scratchpad memory and caches
- Further bottleneck: CPU <-> GPU memory traffic
  - => reduce by DMA and page pinning