

CS110A

# GPU

Chundong Wang

1



信息科学与技术学院

School of Information Science and Technology

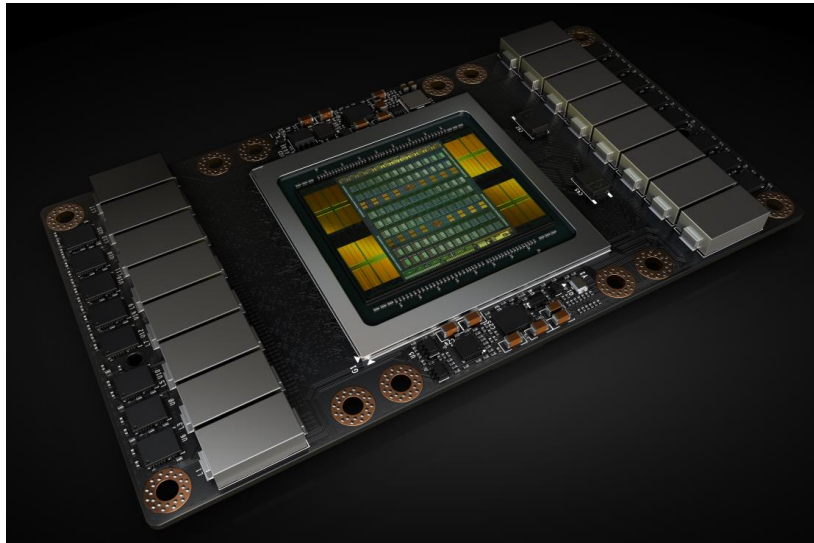
# How do CPU and GPU work?



# What is a GPU?

GPU = Graphics Processing Unit

- Accelerator for raster-based graphics (OpenGL, DirectX)
- Highly programmable (Turing complete)
- Commodity hardware
- Hundreds of ALUs, tens of thousands of concurrent threads



NVIDIA Tesla V100

# “Early” GPU History

- ❑ 1981: IBM PC Monochrome Display Adapter (2D)
- ❑ 1996: 3D graphics (e.g., 3dfx Voodoo)
- ❑ 1999: register combiner (NVIDIA GeForce 256)
- ❑ 2001: programmable shaders (NVIDIA GeForce 3)
- ❑ 2002: floating-point (ATI Radeon 9700)
- ❑ 2005: unified shaders (ATI R520 in Xbox 360)
- ❑ 2006: compute (NVIDIA GeForce 8800)
- ❑ What happened in 2007?

# NVIDIA Tesla architecture (2007)

First alternative, non-graphics-specific (“compute mode”) interface to GPU hardware

- Let’s say a user wants to run a non-graphics program on the GPU’s programmable cores...
- Application can allocate buffers in GPU memory and copy data to/from buffers
- Application (via graphics driver) provides GPU a single kernel program binary
- Application tells GPU to run the kernel in an SPMD fashion (“run N instances of this kernel”)
  - SPMD: single program, multiple data

# CUDA programming language

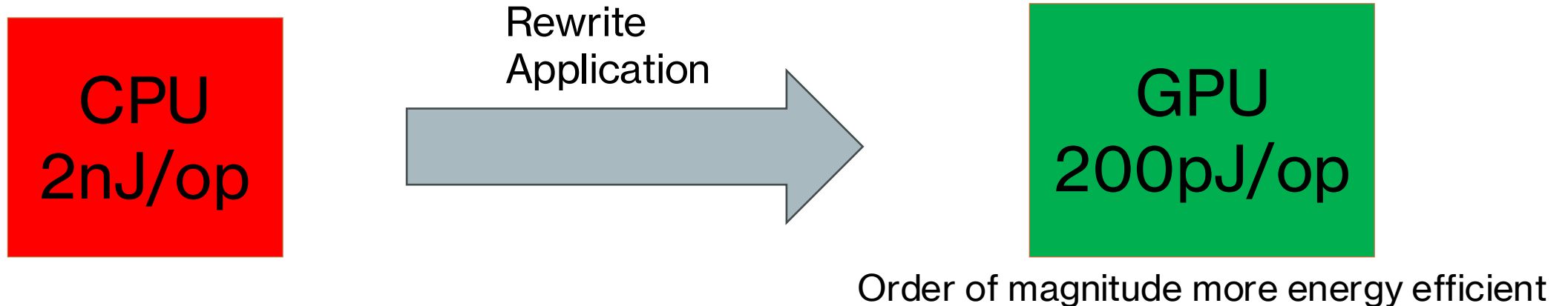
- Introduced in 2007 with NVIDIA Tesla architecture
- “C-like” language to express programs that run on GPUs using the compute-mode hardware interface
- Relatively low-level: CUDA’s abstractions closely match the capabilities/performance characteristics of modern GPUs  
(design goal: maintain low abstraction distance)

# NVIDIA GPU Generations

| Generation       | Year      | Key Advancement                                                |
|------------------|-----------|----------------------------------------------------------------|
| Tesla to Maxwell | 2006-2014 | Foundational era: introduced CUDA and the unified shader model |
| Pascal           | 2016      | High Bandwidth Memory (HBM2), NVLink                           |
| Volta            | 2017      | First-generation Tensor Cores                                  |
| Ampere           | 2020      | Third-gen Tensor Cores, Multi-Instance GPU (MIG)               |
| Hopper           | 2022      | Transformer Engine for LLM training                            |
| Blackwell        | 2024      | Multi-die chiplet design, FP4/FP6 precision                    |

# Why use a GPU for computing?

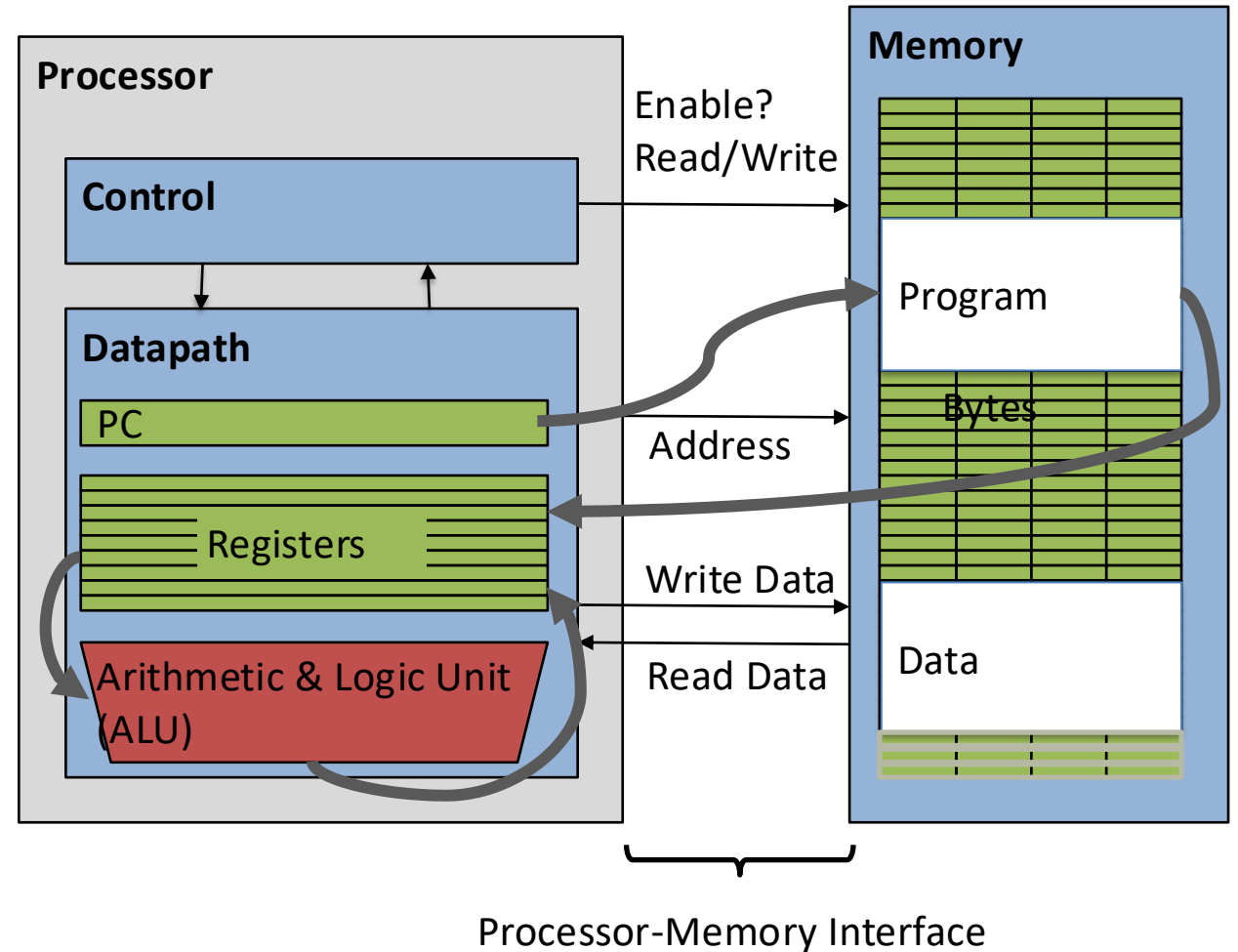
- GPU uses larger fraction of silicon for computation than CPU.
  - So GPU runs faster.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



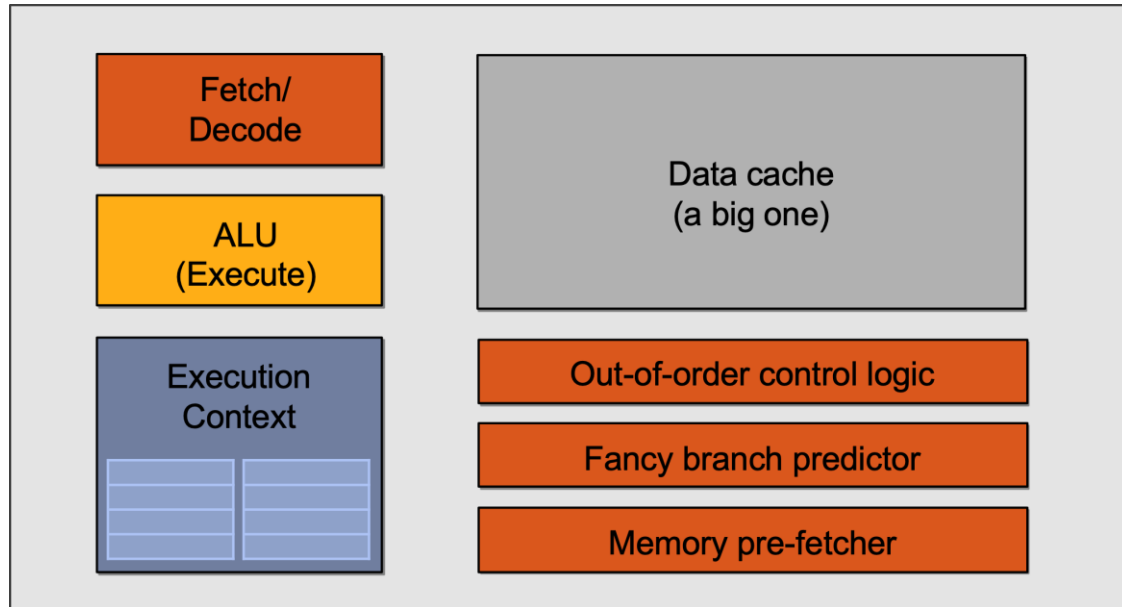
# Review: how to run code on a CPU

Lets say a user wants to run a program on a multi-core CPU...

- ❑ OS loads program text into memory
- ❑ OS selects CPU execution context
- ❑ OS interrupts processor, prepares execution context (sets contents of registers, program counter, etc. to prepare execution context)
- ❑ Go!
- ❑ Processor begins executing instructions from within the environment maintained in the execution context.



# Single-core CPU Execution



**madd** stalled,  
jump to the next  
independent instruction

Can also be executed  
out-of-order  
through register renaming

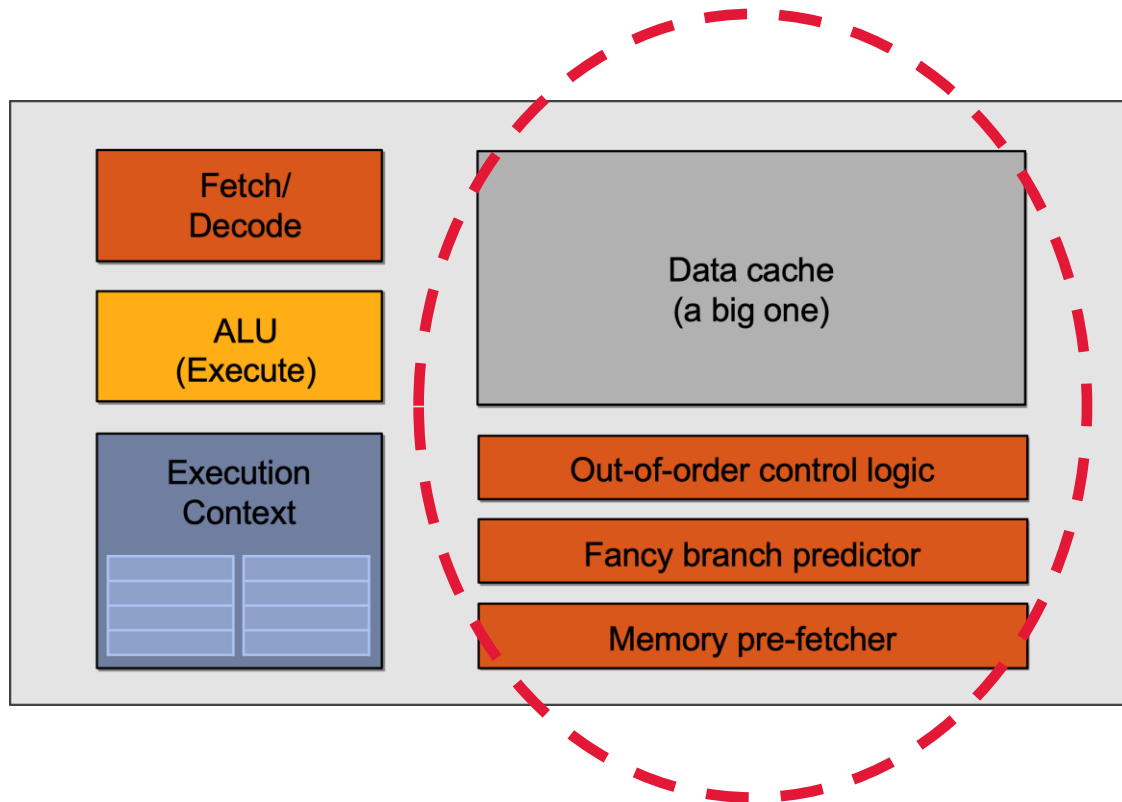
mov R1, 0  
START:  
ld R2, a[R1]  
ld R3, b[R1]  
ld R4, c[R1]  
madd R5, R2, R3, R4  
st R5, v[R1]  
add R1, R1, 1  
bra START if R1 < N

START:  
ld R2, a[R1]  
ld R3, b[R1]  
ld R4, c[R1]  
madd R5, R2, R3, R4  
st R5, v[R1]  
add R1, R1, 1  
bra START if R1 < N

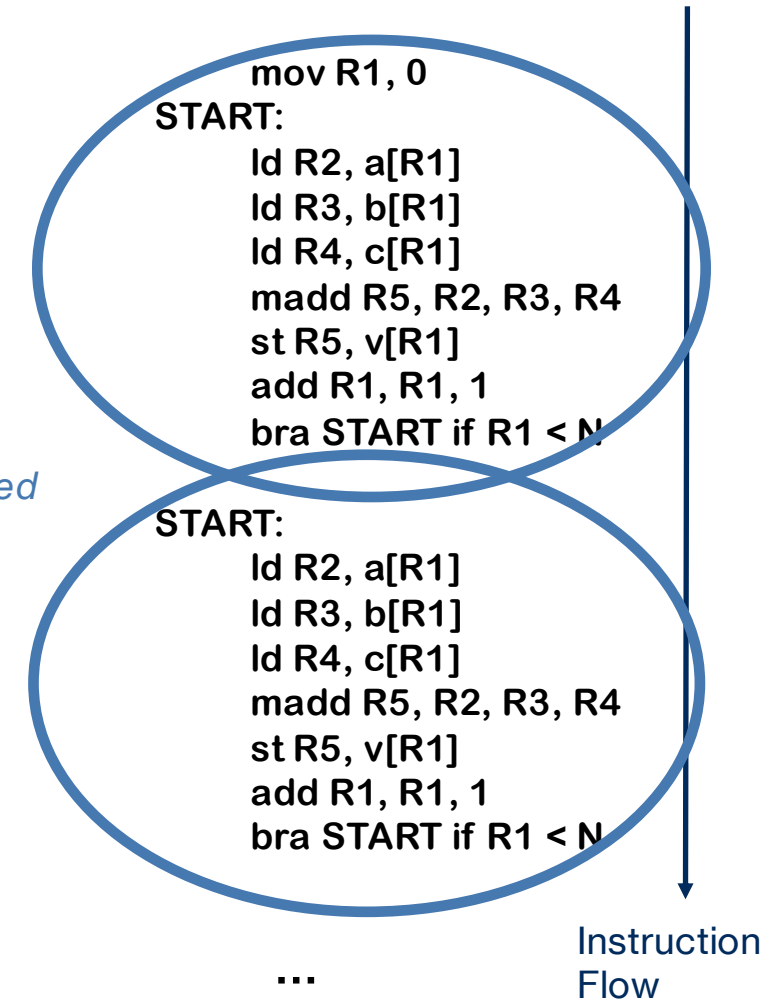
...

Instruction  
Flow

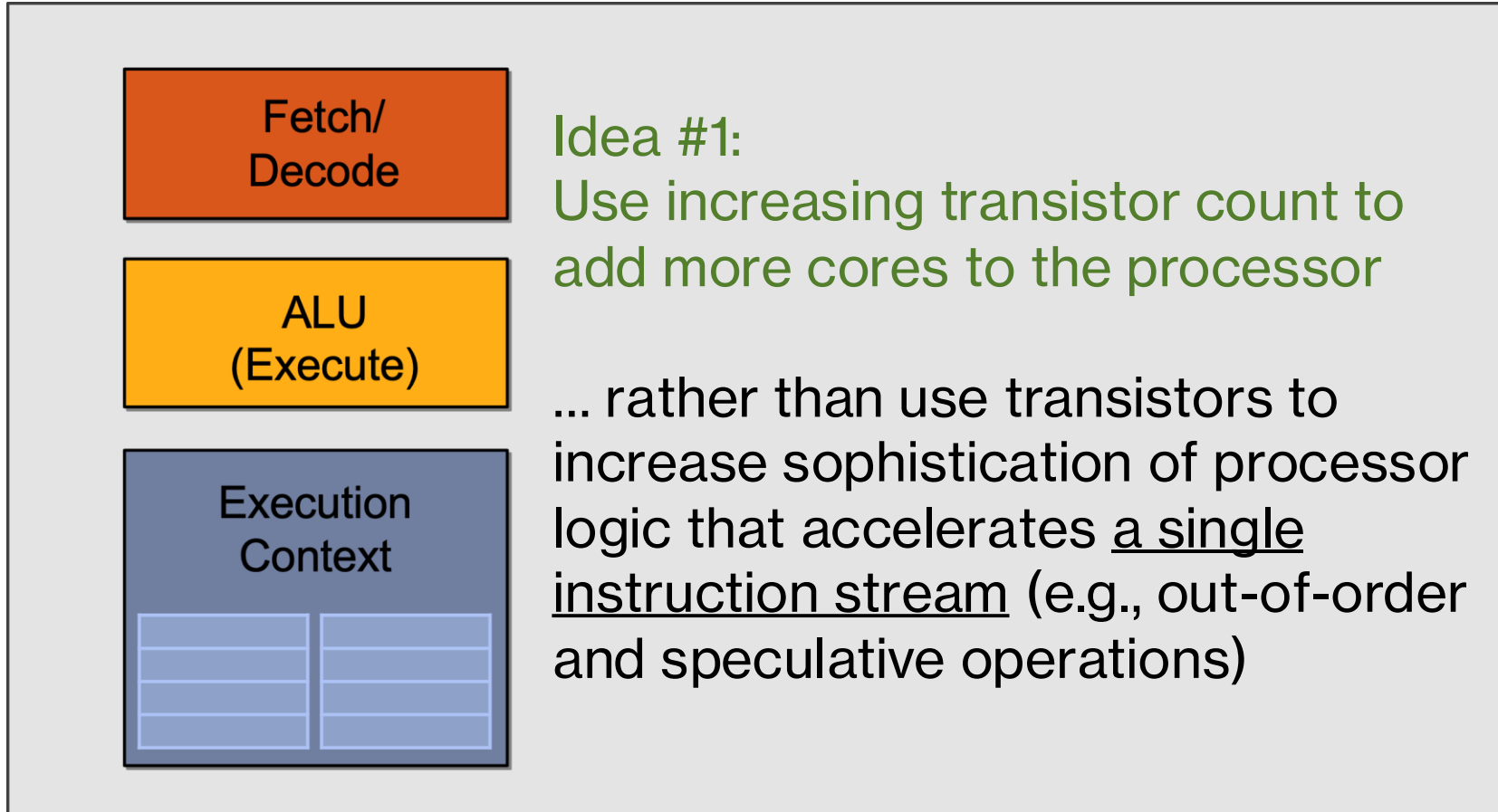
# Single-core CPU Execution



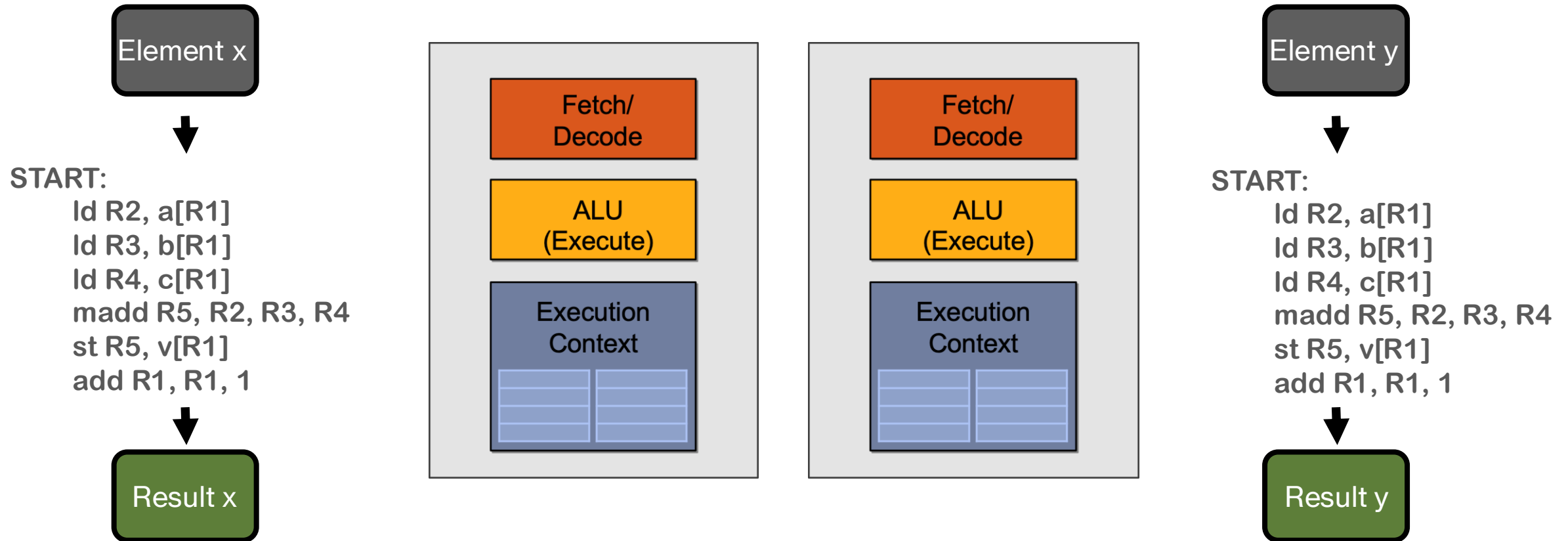
*But what if we tell the hardware  
these two blocks can be executed  
in parallel to begin with?*



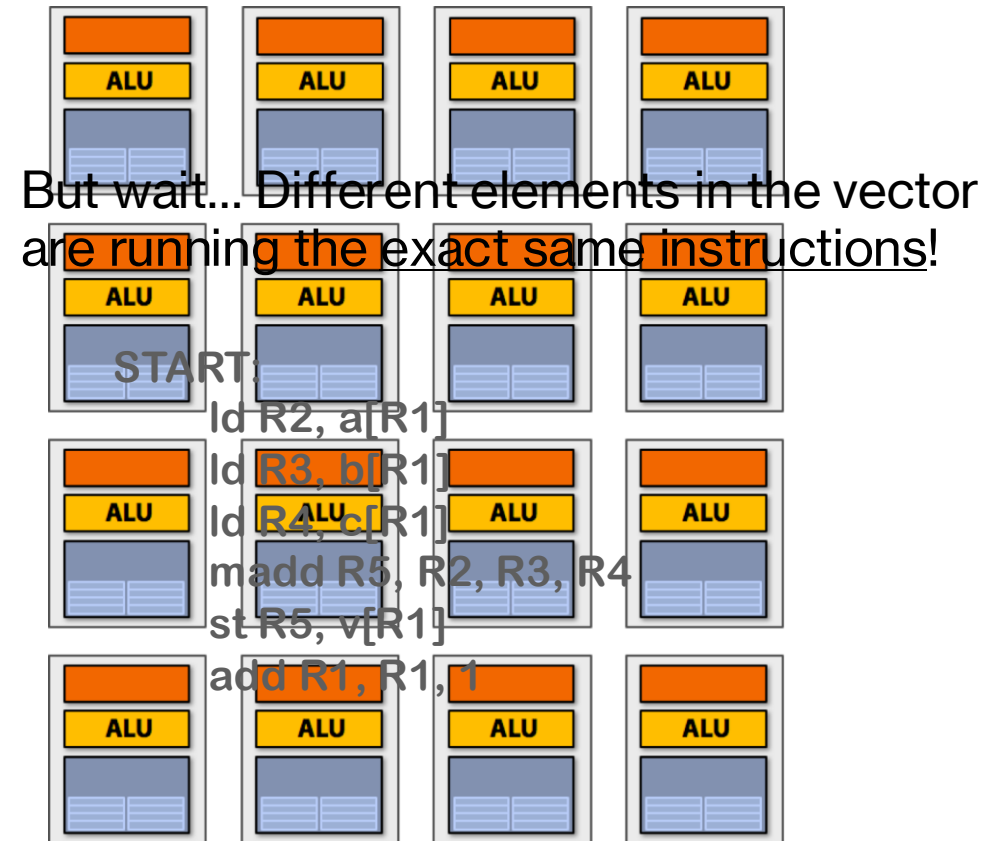
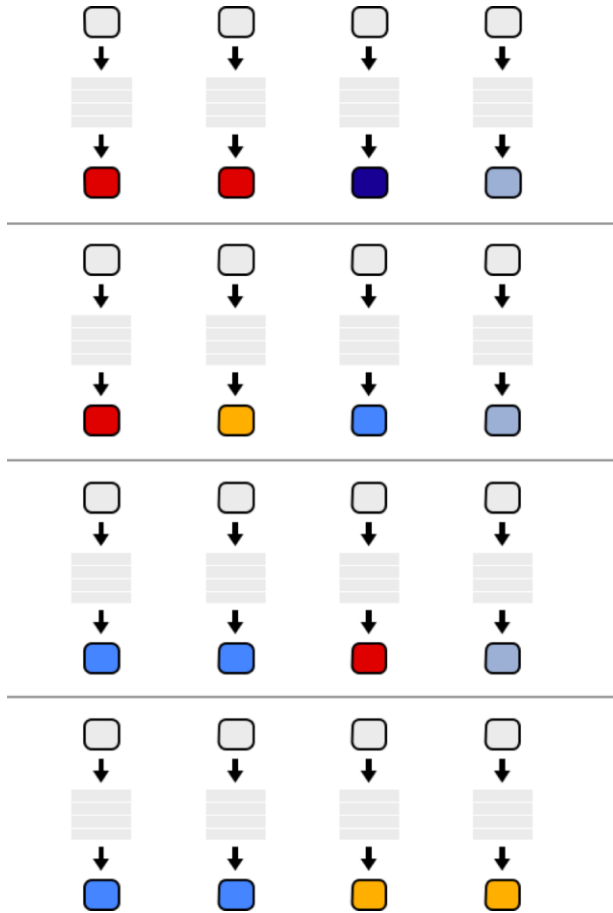
# Slimming Down



# Two cores (Two Elements in Parallel)

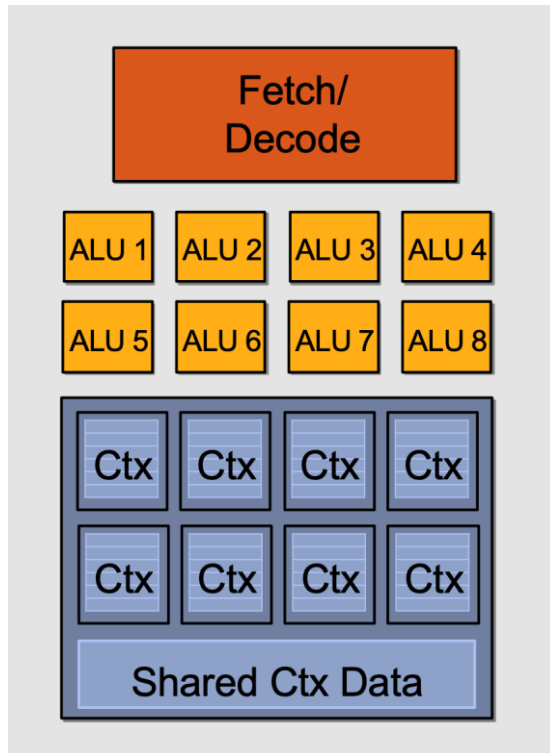


# Sixteen Cores



16 cores = 16 simultaneous instruction streams

# Instruction Stream Sharing

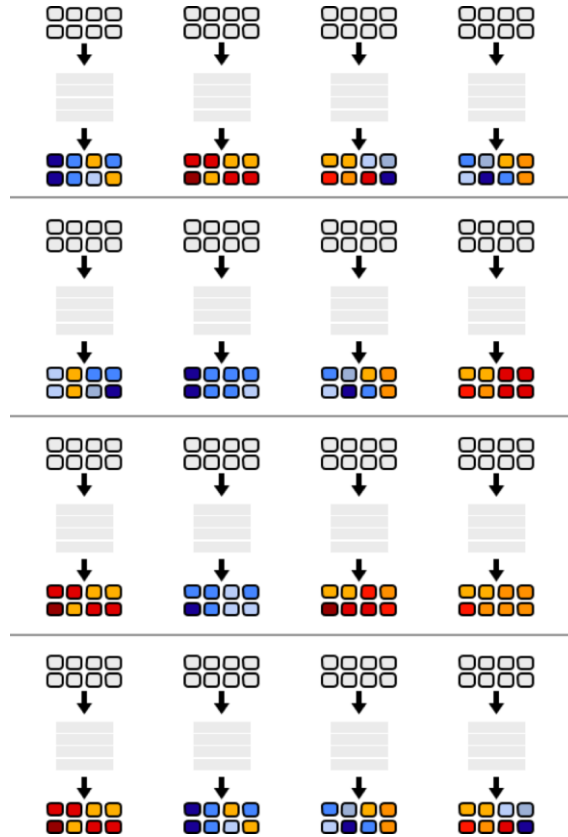


Idea #2:

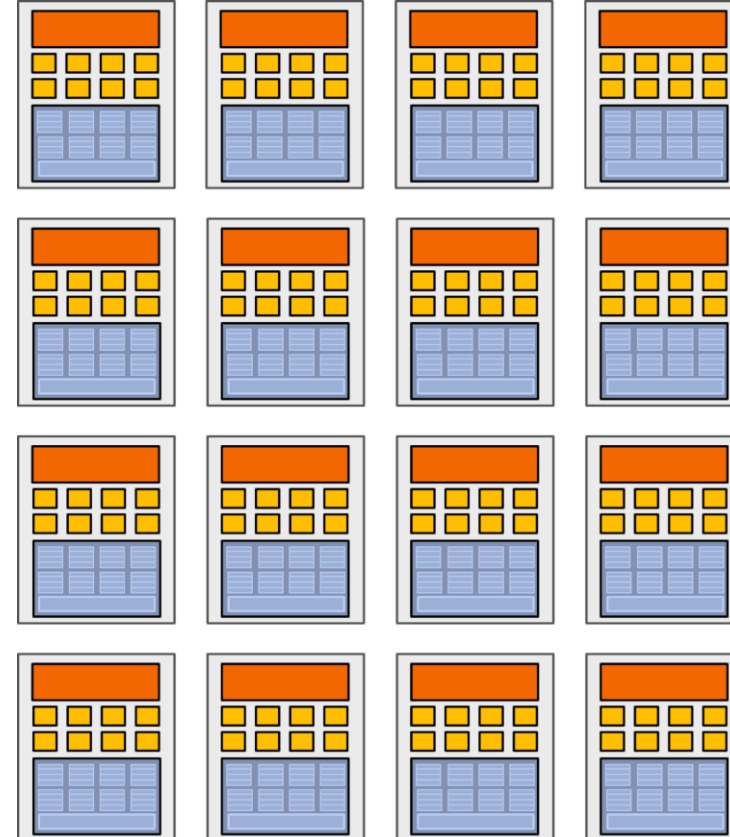
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing!

# 128 Elements in Parallel

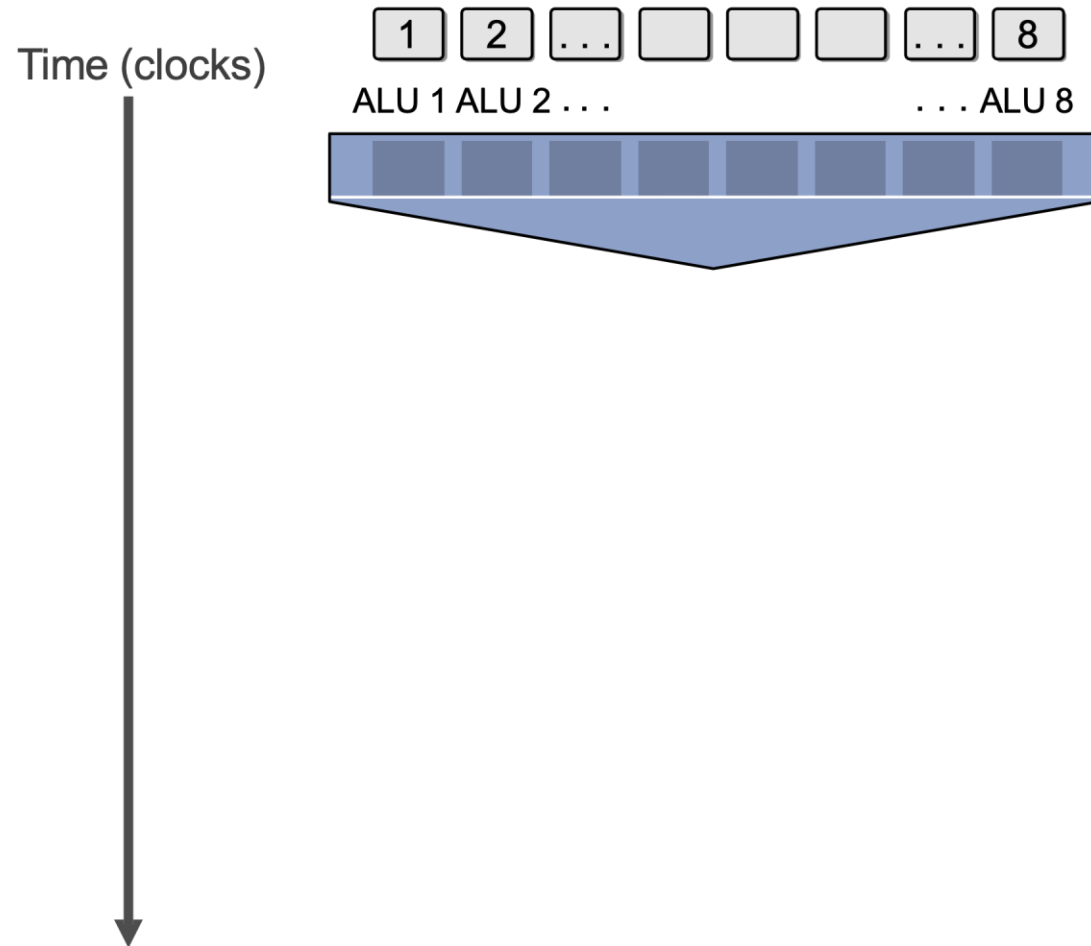


16 cores x 8 ALUs/core = 128 ALUs



16 cores = 16 simultaneous instruction streams

# What about Branches?

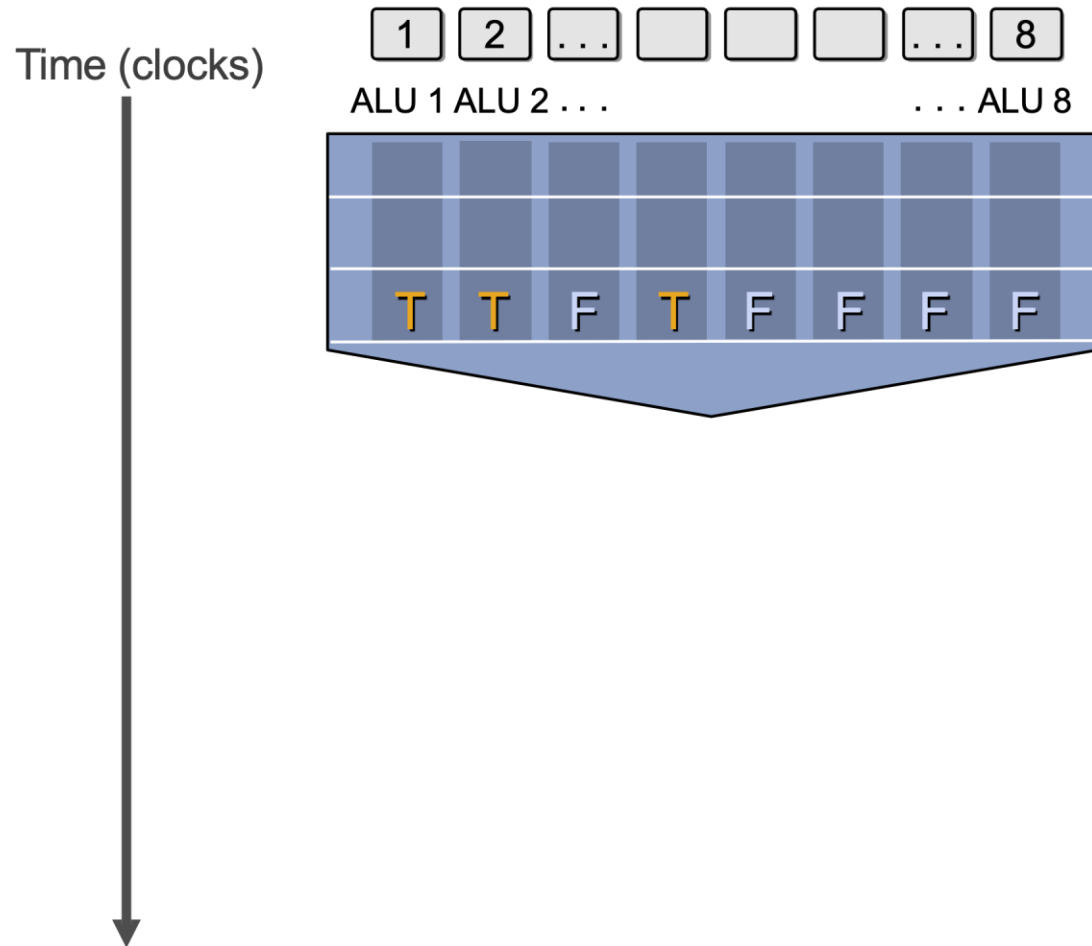


<unconditional shader code>

```
if (x > 0) {  
    y = pow(x, exp);  
    y *= Ks;  
    refl = y + Ka;  
} else {  
    x = 0;  
    refl = Ka;  
}
```

<resume unconditional shader code>

# What about Branches?



<unconditional shader code>

```
if (x > 0) {
```

```
    y = pow(x, exp);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

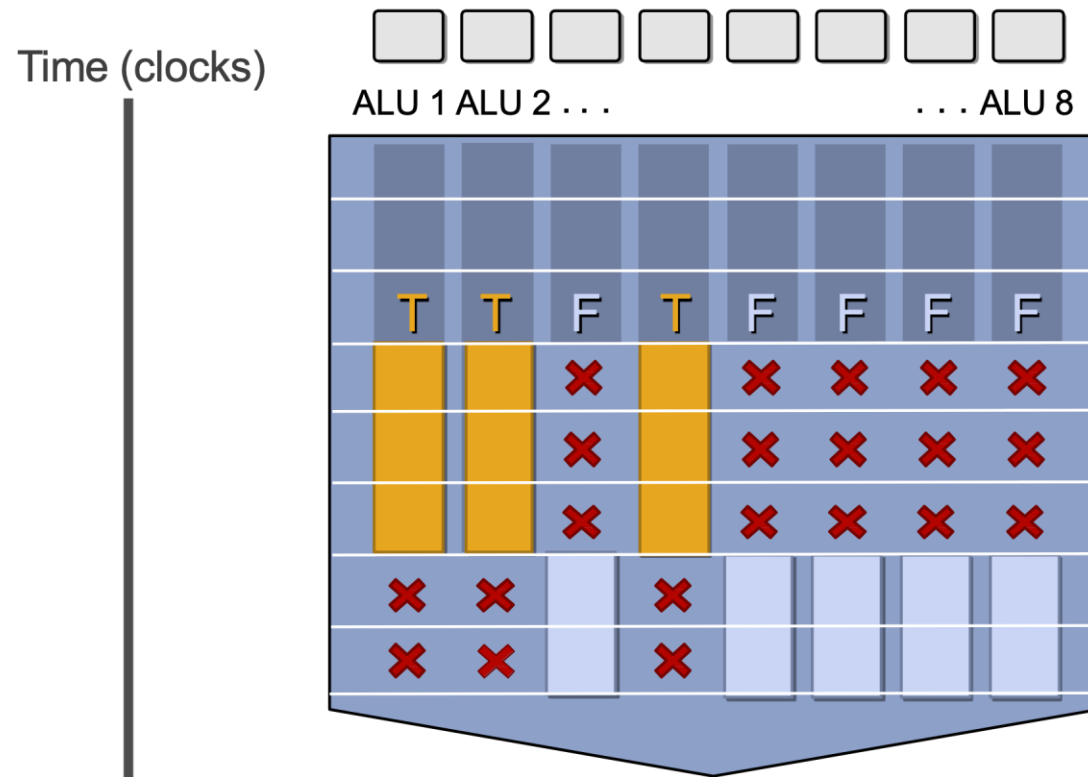
```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional shader code>

# What about Branches?



<unconditional shader code>

```
if (x > 0) {
```

```
    y = pow(x, exp);
```

```
    y *= Ks;
```

```
    refl = y + Ka;
```

```
} else {
```

```
    x = 0;
```

```
    refl = Ka;
```

```
}
```

<resume unconditional shader code>

Not all ALUs do useful work!

Worst case: 1/8 peak performance

# SIMD Execution on Modern GPUs

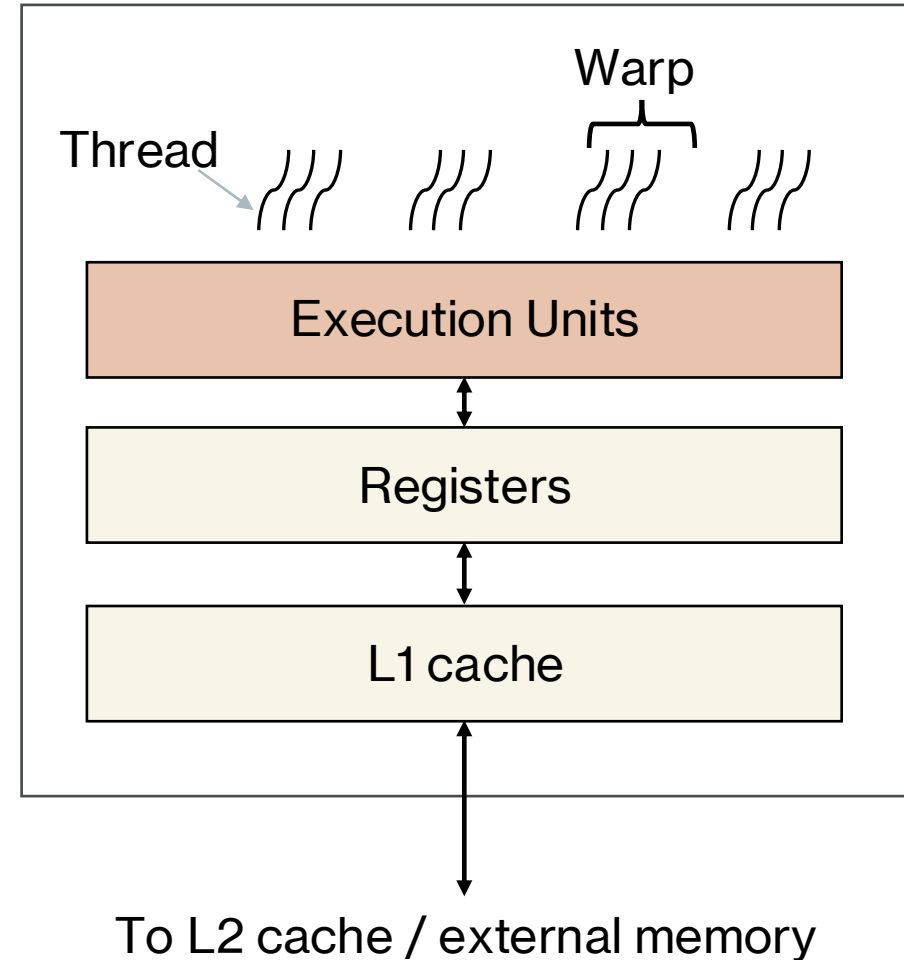
- “Implicit SIMD”
  - Compiler generates a scalar binary (scalar as opposed to vector instructions)
  - But N instances of the program are **always running** together on the processor  
i.e., `execute(my_function, N)` // execute my\_function N times
  - Hardware (not compiler) is responsible for simultaneously executing the same instruction on different data in SIMD ALUs
- SIMD width in practice
  - 32 on NVIDIA GPUs (a warp of threads) and 64 on AMD GPUs (wavefront)
  - Divergence can be a big issue (poorly written code might execute at 1/32 the peak capability of the machine!)

# Warp and SM

SM: Streaming multiprocessor

- ❑ Also known as Computing Unit in AMD's documents
- ❑ Each SM is a highly-multi-threaded processor
- ❑ Today, 24 to 28 warps of 32 threads each
- ❑ ~1K threads on each SM, ~20K threads on a GPU

## A sketch of SM



# Dealing with Stalls on In-order Cores

- Stalls occur when a core cannot run the next instruction because of a dependency on a previous long-latency operation
- We've removed fancy logic that helps avoid stalls
  - No more out-of-order execution to exploit instruction-level parallelism (ILP)
  - Traditional cache doesn't always help since a lot of workloads are streaming data
- But, we have a LOT of parallel work...

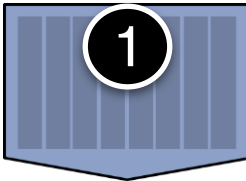
**Idea #3: Interleave processing of many warps on a single core to avoid stalls caused by high-latency operations**

# Hiding Stalls

Time  
(clock cycles)



Element 1...8



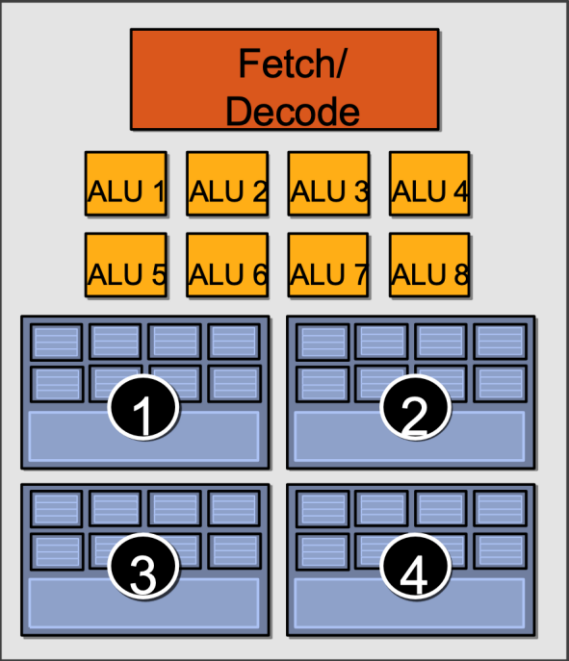
Element 9...16



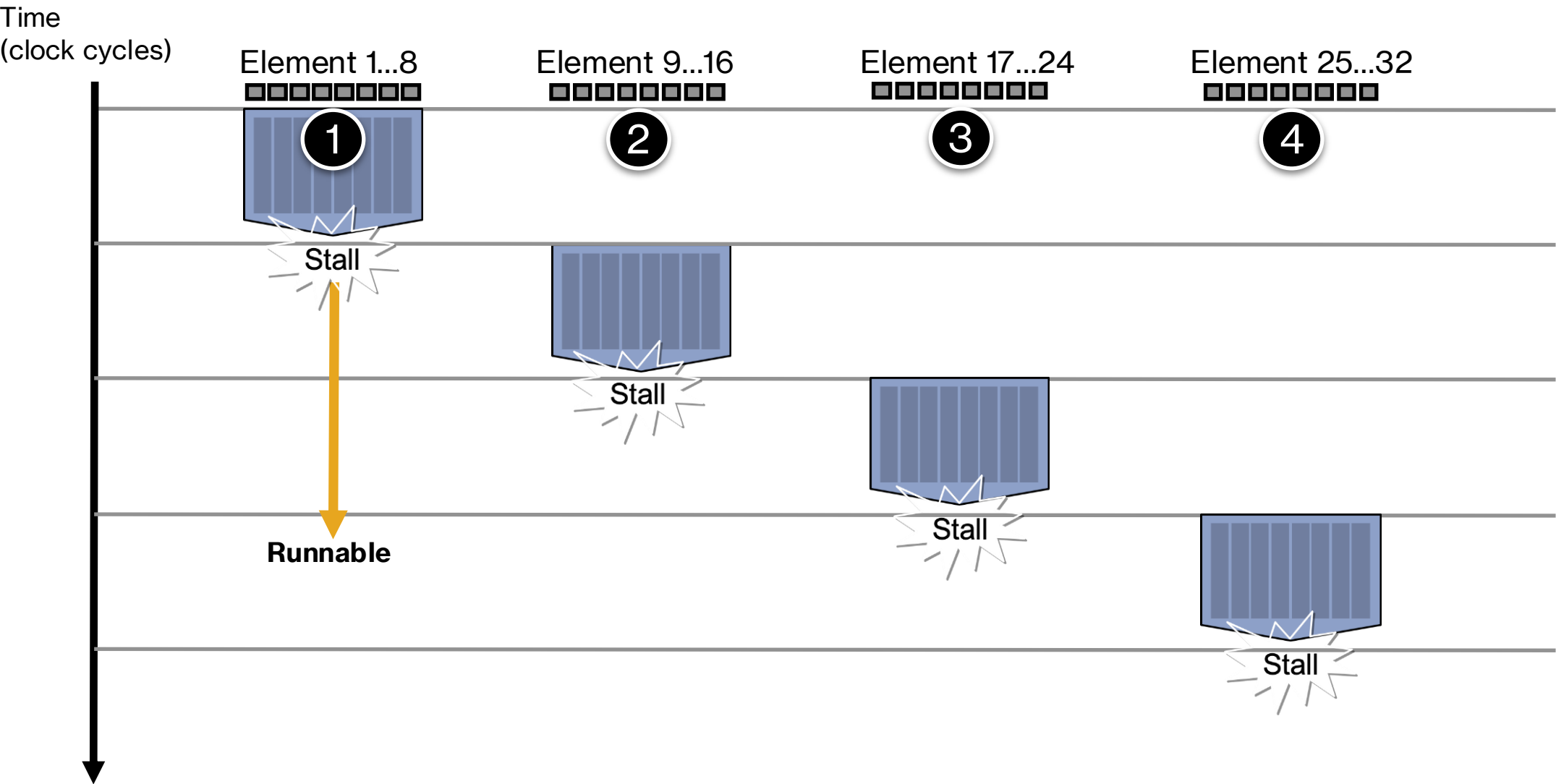
Element 17...24



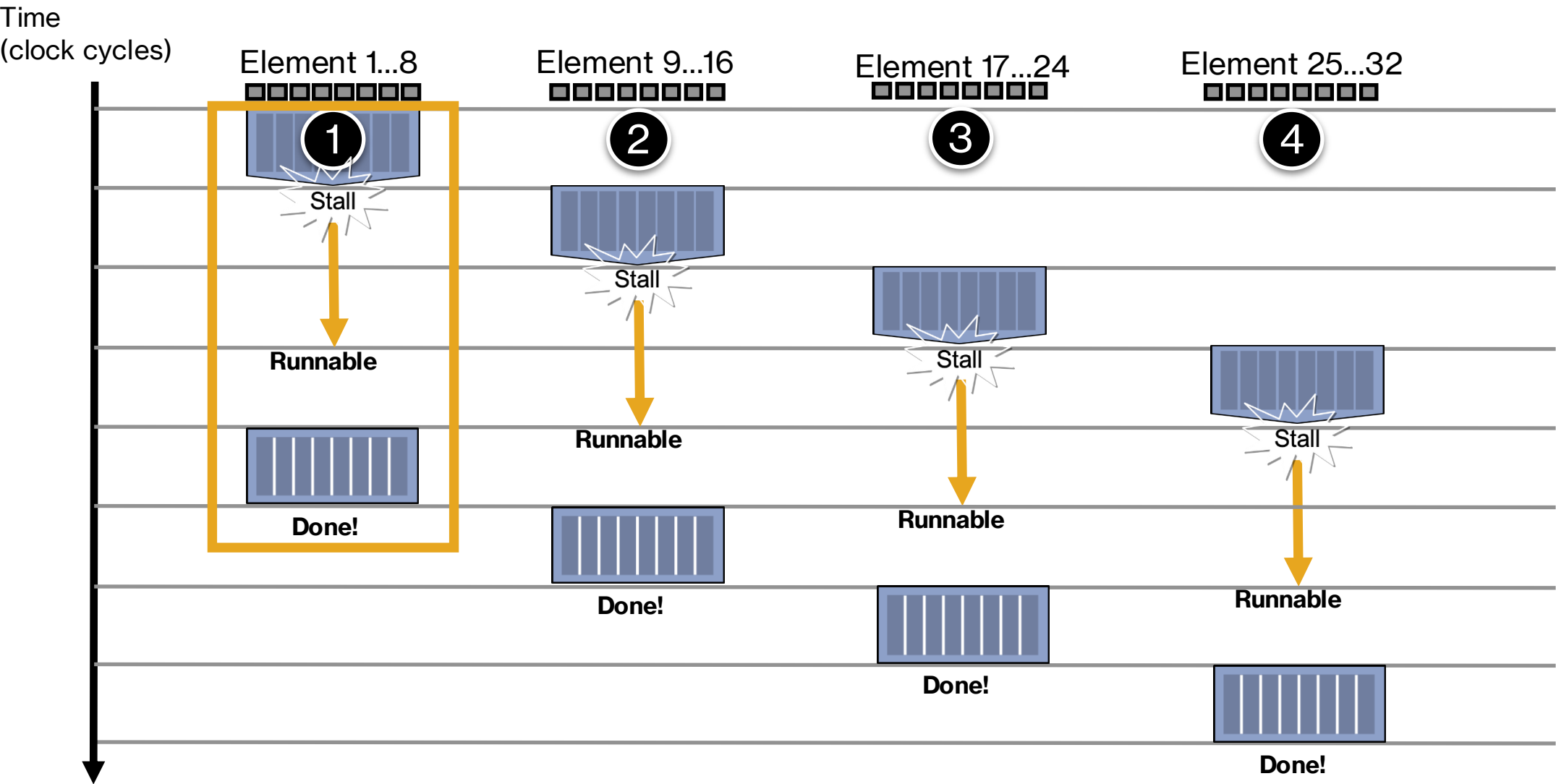
Element 25...32



# Hiding Stalls



# Hiding Stalls



# Throughput Computing Trade-off

Time  
(clock cycles)

Element 1...8



Runnable



Done!

Element 9...16



Element 17...24



Element 25...32



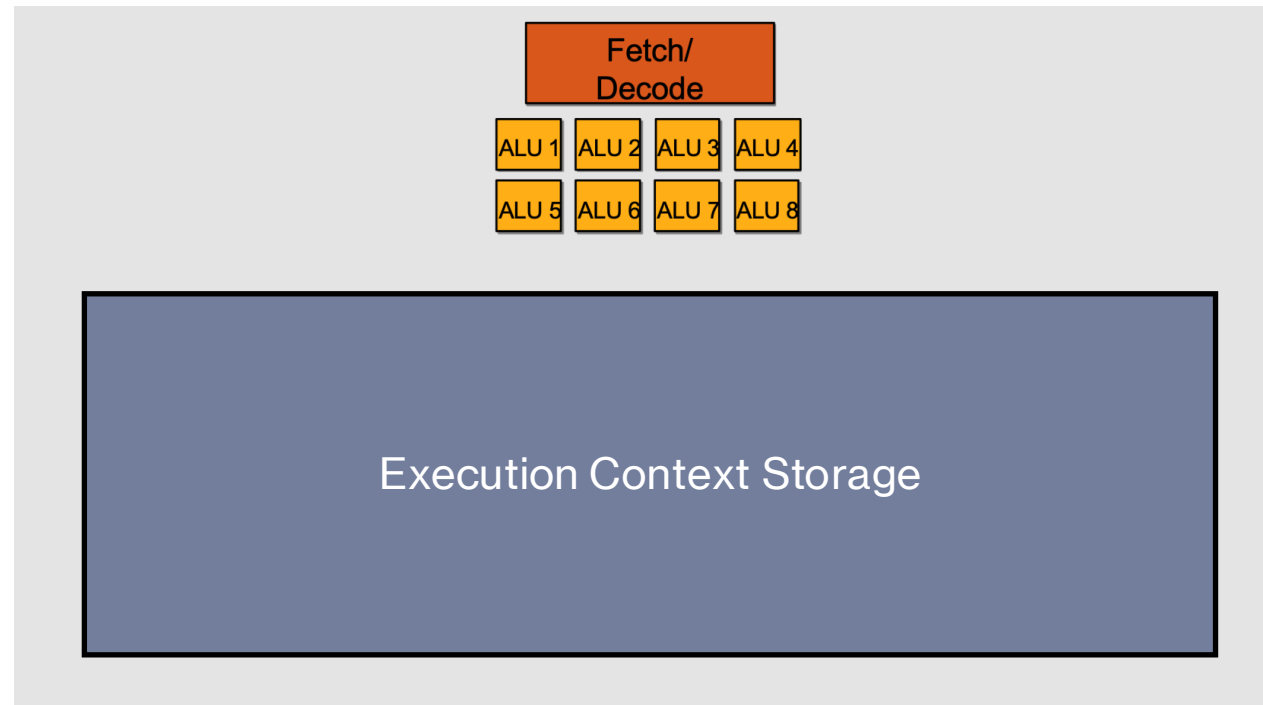
During this time, this group is runnable, but it is not being executed by the processor. (The core is running some other group.)

Key idea of throughput-oriented systems:

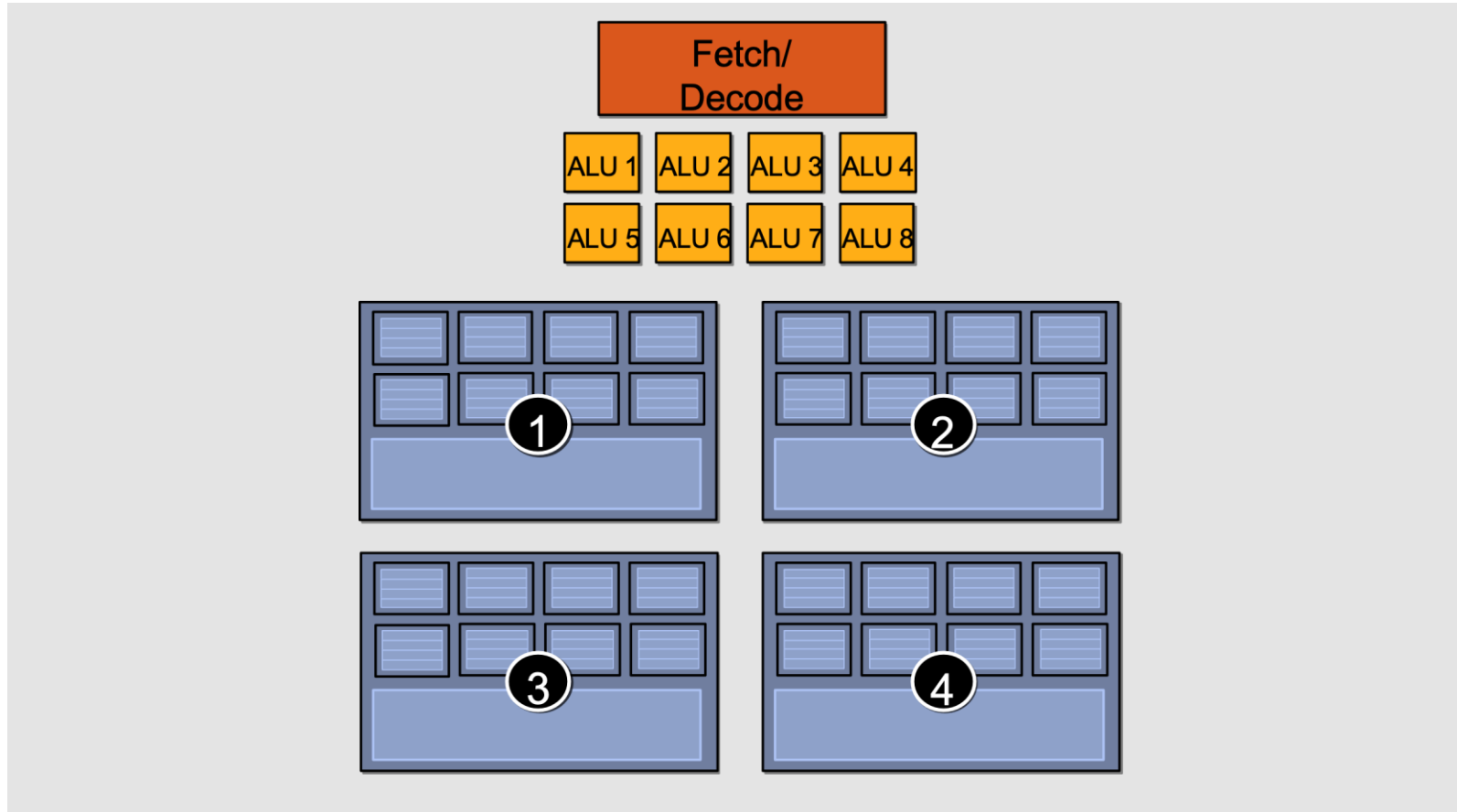
Potentially increase runtime of one group, in order to increase throughput of overall system running multiple groups.

# Storing Execution Contexts

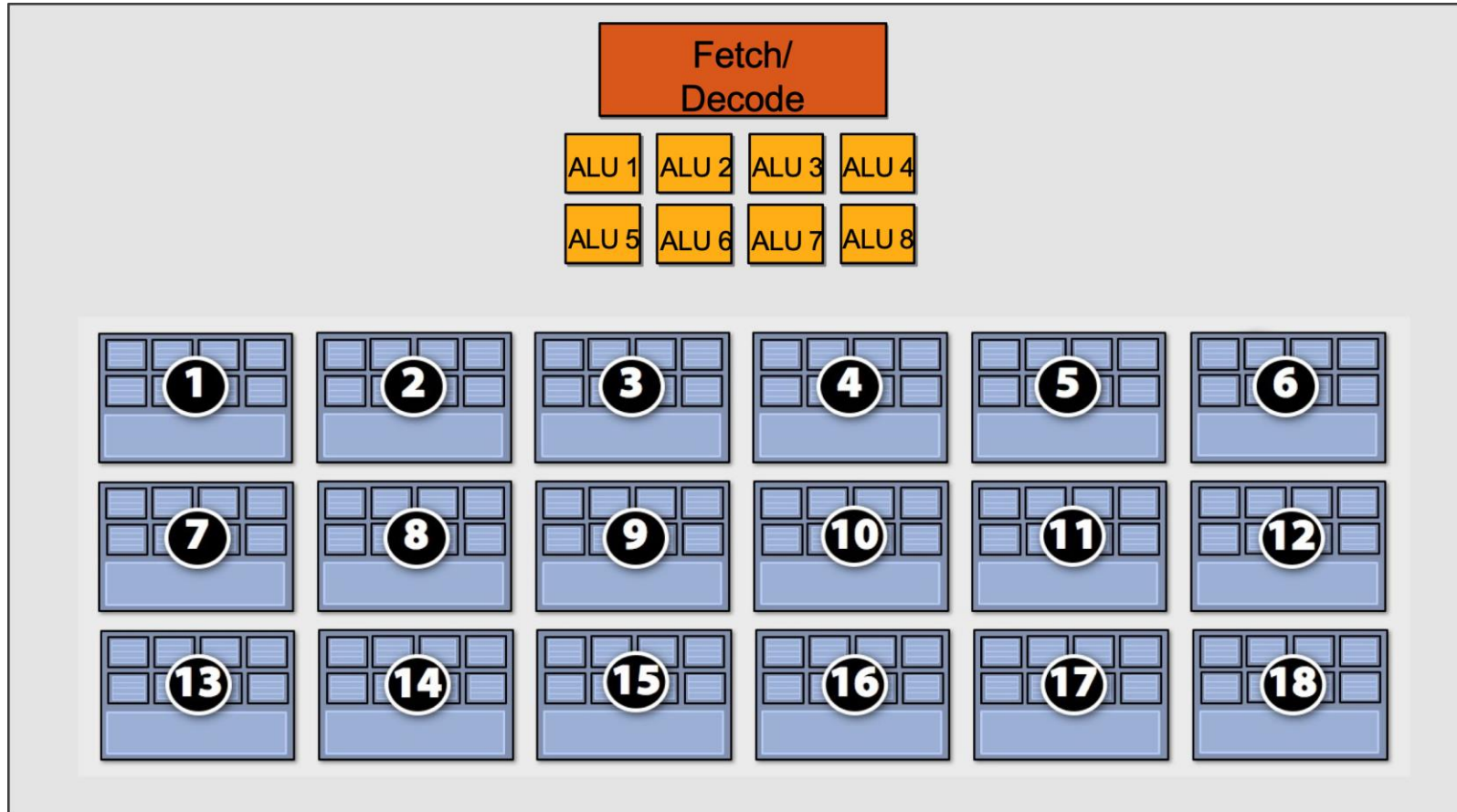
- Consider on-chip storage of execution contexts a finite resource
- Resource consumption of each thread group is program-dependent



# Four Large Contexts (*Low Latency Hiding*)



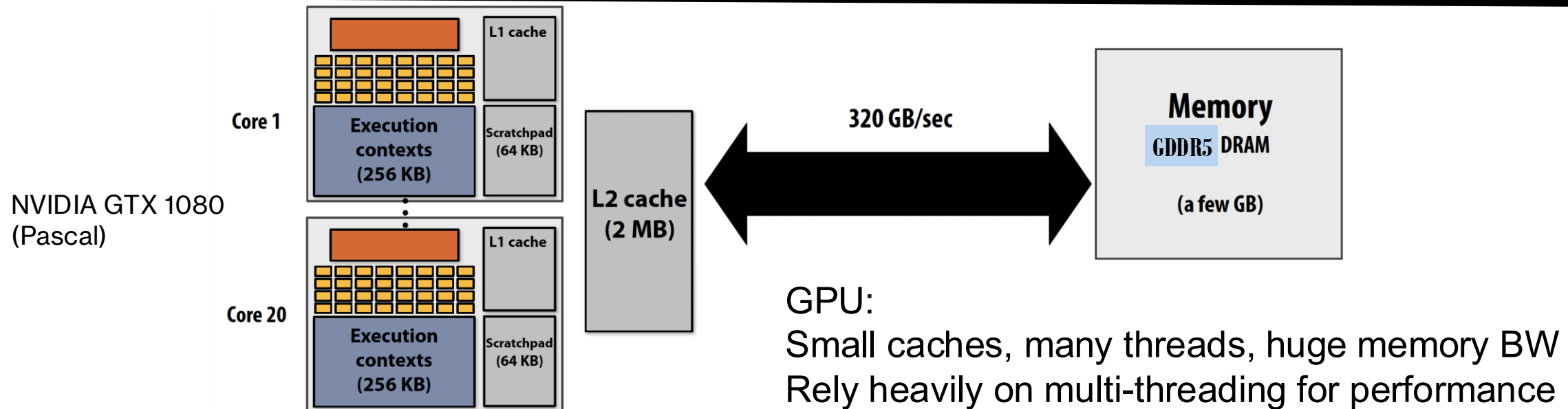
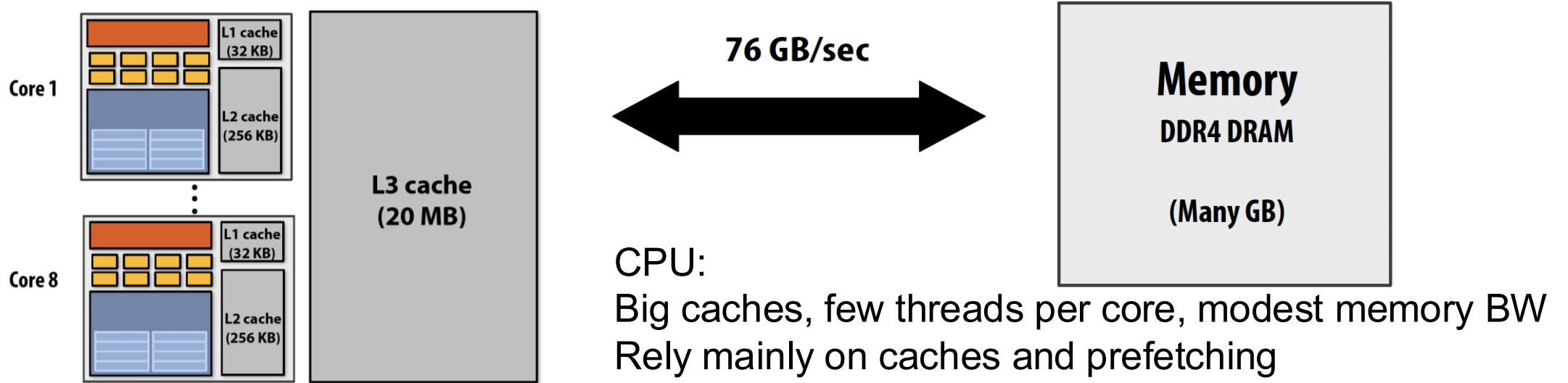
# Eighteen Small Contexts (*High Latency Hiding*)



# Summary: Three Key Ideas

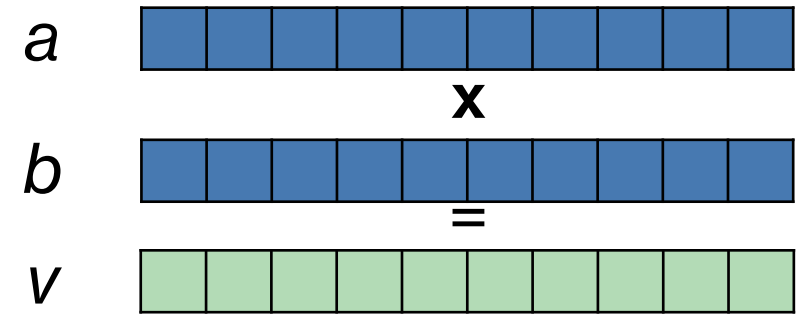
1. Use many “slimmed down cores” to run in parallel
2. Pack cores full of ALUs (by sharing instruction stream on multiple data)
3. Avoid latency stalls by interleaving execution of many groups of threads
  - When one group stalls, work on another group

# CPU v.s. GPU Memory Hierarchies



# Thought Experiment

- Consider element-wise multiplication of two vectors ***a*** and ***b***
- Assume vectors contain millions of elements
  - Load input ***a***[*i*]
  - Load input ***b***[*i*]
  - Compute ***a***[*i*] **x** ***b***[*i*]
  - Store result into ***v***[*i*]
- Three memory operations (12 bytes) for every MUL
- NVIDIA GTX 1080 GPU can do 2560 MULs per clock (@ 1.6 GHz)
- Need ~45 TB/sec of bandwidth to keep functional units busy (only have 320 GB/sec)



<1% GPU efficiency... but 4.2x faster than eight-core CPU in lab!

(3.2 GHz Xeon E5v4 eight-core CPU connected to 76 GB/sec memory bus will exhibit ~3% efficiency on this computation)

# Bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.  
**No amount of latency hiding helps this.**

Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

# Bandwidth is a *Critical* Resource

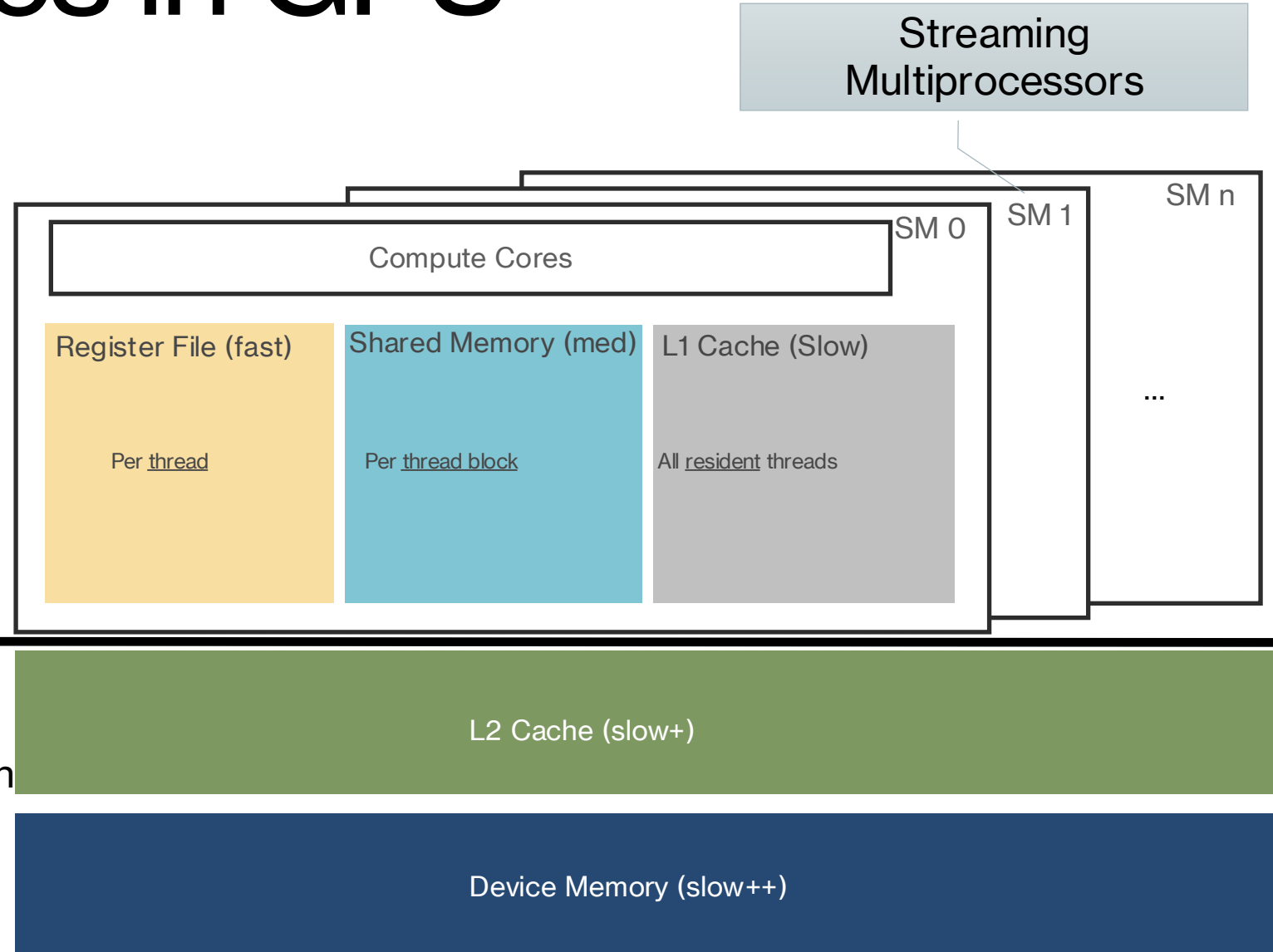
Performant parallel programs will:

- Organize computation to fetch data from memory less often
  - Reuse data previously loaded by the same thread
  - Share data across threads through scratchpad (inter-thread cooperation)
  - Access contiguous memory within the same warp (hardware managed memory coalescing)
- Request data less often (instead, do more arithmetic: it's “free”)
  - Useful term: “arithmetic intensity” — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

# Memory Spaces in GPU

## On-chip:

- Register file
  - Usage determined by compiler
  - Spills go to local memory
- Shared memory, i.e. scratchpad
  - Programmer managed
  - Bank conflicts
- L1 cache



## Off-chip:

- L2 cache
  - Bandwidth filter for DRAM rather than reducing latency as in CPUs
- Device memory (DRAM)
  - Several spaces: global memory, texture memory, local memory
  - Different spaces have different caching policies

# Modern GPU Architecture (Volta 2017)

21B transistors  
815 mm<sup>2</sup>

80 SM  
5120 CUDA Cores  
640 Tensor Cores

16/32 GB HBM2  
900 GB/s HBM2  
300 GB/s NVLink



\*full GV100 chip contains 84 SMs

# iGPU: Integrated GPU

## Key functions of iGPU

- **Rendering Graphics:** Handles all basic rendering tasks, from displaying desktop interfaces to running less demanding games.
- **Power Efficiency:** Typically consumes less power compared to dedicated GPUs, making it ideal for laptops and portable devices.
- **Cost-Effective:** Reduces the need for additional hardware, providing a budget-friendly option for users.

## A comparison between iGPU and Dedicated GPU

- Performance: dedicated GPU is superior while iGPU is fit for everyday tasks
- Power efficiency: iGPU is more power efficient
- Cost: the price of iGPU is generally lower

# Scalable Parallel Programming with CUDA

**Compute Unified Device Architecture (CUDA)** is a scalable parallel programming model and software platform for the GPU and other parallel processors that allows the programmer to bypass the graphics API and graphics interfaces of the GPU and simply program in C or C++.

Key abstractions

- A hierarchy of thread groups
- Barrier synchronization
- Shared memories (to be covered in memory lectures)

As for threads

- Kernel, a program or function for one thread, to be executed by many threads
- Thread block, a set of concurrent threads that execute the same thread program
- Grid, a set of thread blocks that execute the same kernel program

As for Synchronization

- ❑ Threads wait at a synchronization barrier until all threads in the thread block arrive at the barrier
- ❑ No thread in the block can proceed until all threads in the block have reached the barrier
- ❑ Atomic memory operations (AMO, to be covered later)

# The market of GPUs

|                               | NVIDIA                                                        | AMD                                                 | Ascend (Huawei)                                                 | MetaX (MuXi)                                             |
|-------------------------------|---------------------------------------------------------------|-----------------------------------------------------|-----------------------------------------------------------------|----------------------------------------------------------|
| Architecture                  | Ada (consumer), Hopper / Blackwell (datacenter)               | RDNA (gaming), CDNA (datacenter)                    | Da Vinci Architecture                                           | Self-designed domestic GPGPU                             |
| Core Components               | Streaming Multiprocessor + Tensor Core + RT Core              | Compute Unit + Matrix Core + Ray Accelerator        | Dedicated NPU AI core, vector unit, <b>no graphics hardware</b> | Unified GPGPU core (graphics + HPC + AI)                 |
| Software Stack & Interconnect | CUDA, cuDNN, TensorRT, NVLink                                 | ROCm, HIP, Infinity Fabric                          | CANN, MindSpore, Lingqu Interconnect                            | MXMACA, MetaXLink                                        |
| Key Features                  | Most mature ecosystem; highly optimized for LLM & ray tracing | Large HBM memory, open stack, good cost-performance | Fully domestic chip, powerful low-precision matrix calculation  | Support FP64 for scientific simulation, all-in-one GPGPU |
| Typical Chips                 | RTX4090, A100, H100, B200                                     | RX7900XTX, MI300X                                   | Ascend 910, Ascend 950                                          | N-series, C-series, X-series                             |
| Main Scenarios                | AI research, LLM training, gaming, visualization              | Gaming, HPC, cost-sensitive datacenter              | Large model training & inference (no gaming)                    | AI for Science, LLM, domestic visualization & HPC        |

# Summary

- ❑ GPU is promising
- ❑ CPU-GPU heterogenous computing
  - Either has its strengths. Why not jointly utilize them?

# Acknowledgement

Gennady Pekhimenko, University of Toronto

Kayvon Fatahalian and Kunle Olukotun, Stanford University

Daniel Sanchez, MIT