CSC 2224: Parallel Computer Architecture and Programming GPU Architecture: Introduction

Prof. Gennady Pekhimenko

University of Toronto

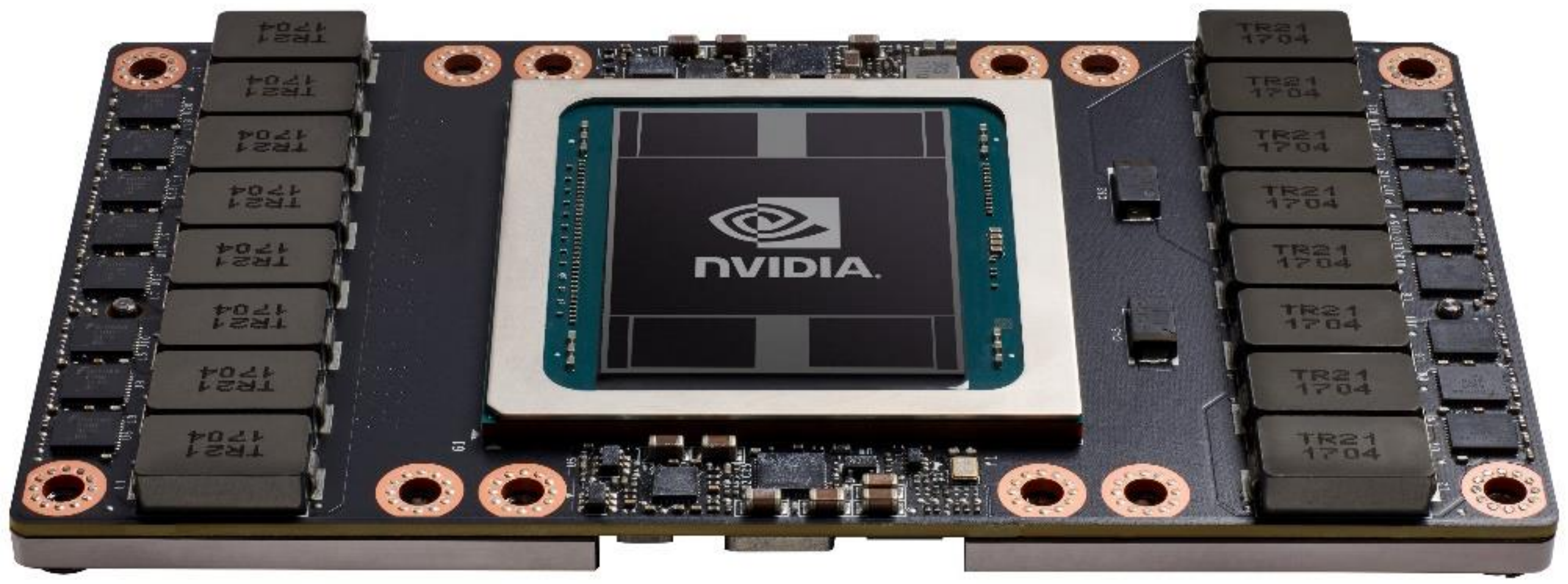
Fall 2020

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan

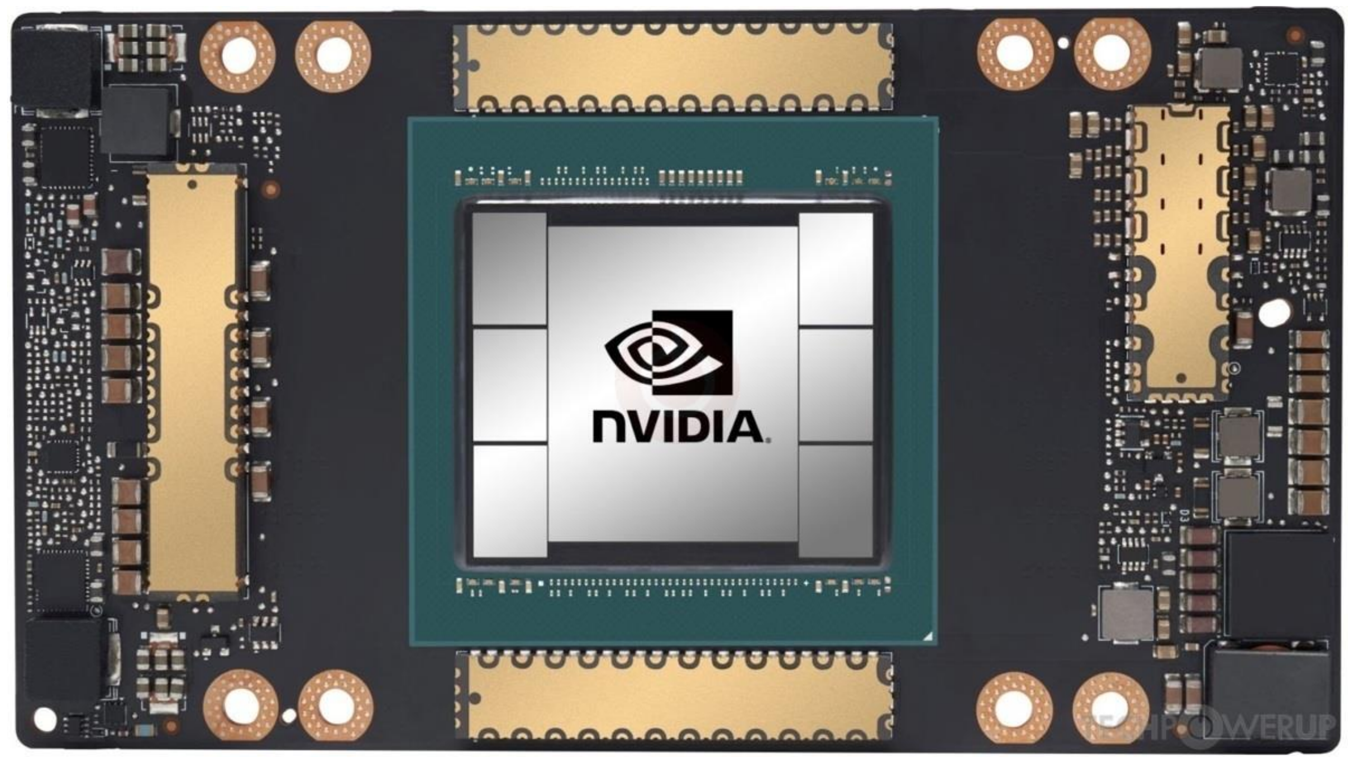
https://www.youtube.com/watch?v=-P28LKWTzrI

What is a GPU?

- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100’s of ALUs; 10’s of 1000s of concurrent threads

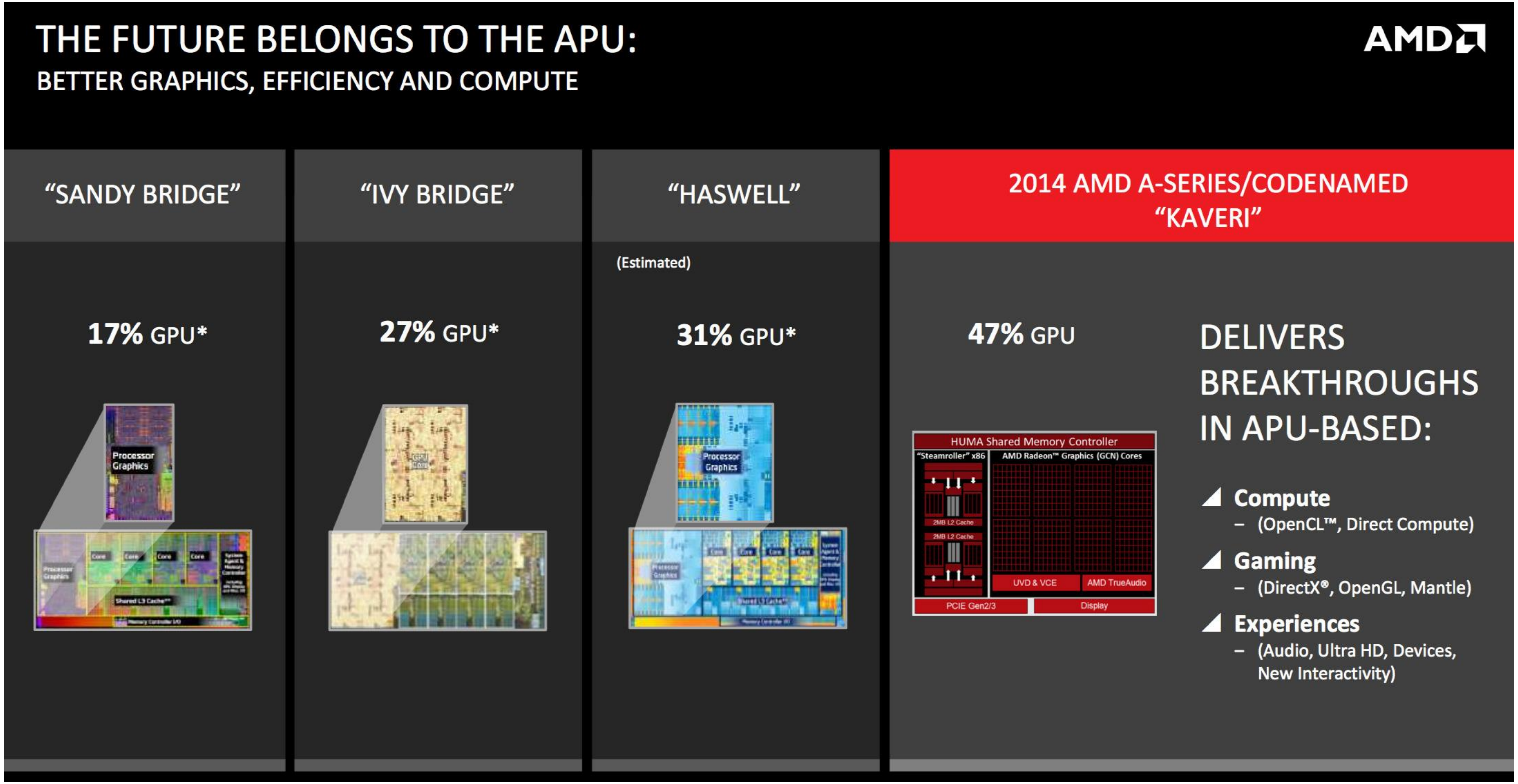


NVIDIA Volta: V100



NVIDIA Volta: A100

The GPU is Ubiquitous



THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

AMD

"SANDY BRIDGE"	"IVY BRIDGE"	"HASWELL"	2014 AMD A-SERIES/CODENAMED "KAVERI"
		(Estimated)	
17% GPU*	27% GPU*	31% GPU*	47% GPU

DELIVERS BREAKTHROUGHS IN APU-BASED:

- **Compute**
  - (OpenCL™, Direct Compute)
- **Gaming**
  - (DirectX®, OpenGL, Mantle)
- **Experiences**
  - (Audio, Ultra HD, Devices, New Interactivity)

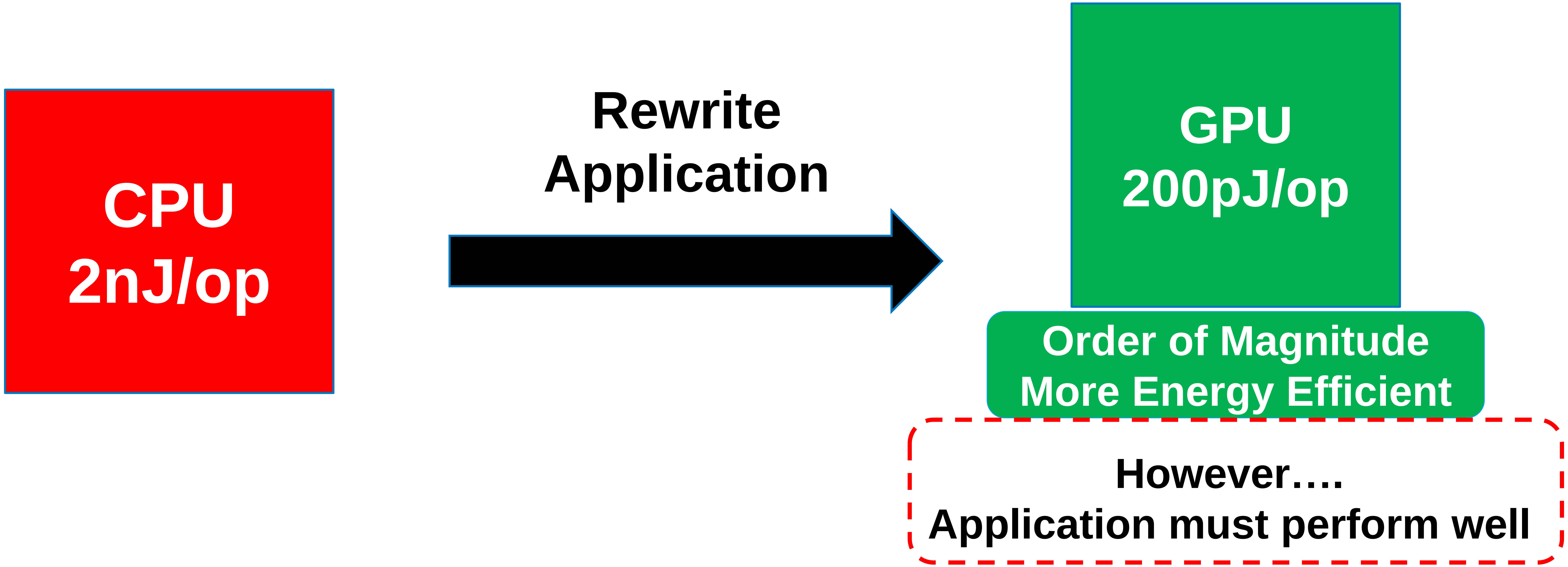
[APU13 keynote]

“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)

Why use a GPU for computing?

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



CPU
2nJ/op

Rewrite
Application

GPU
200pJ/op

Order of Magnitude
More Energy Efficient

However….
Application must perform well

Agenda

- Three key ideas that make GPUs run fast
- GPU memory hierarchy
- Closer look at a modern GPU architecture (Nvidia’s Volta)
    - Memory: higher bandwidth, larger capacity
    - Compute: application-specific hardware

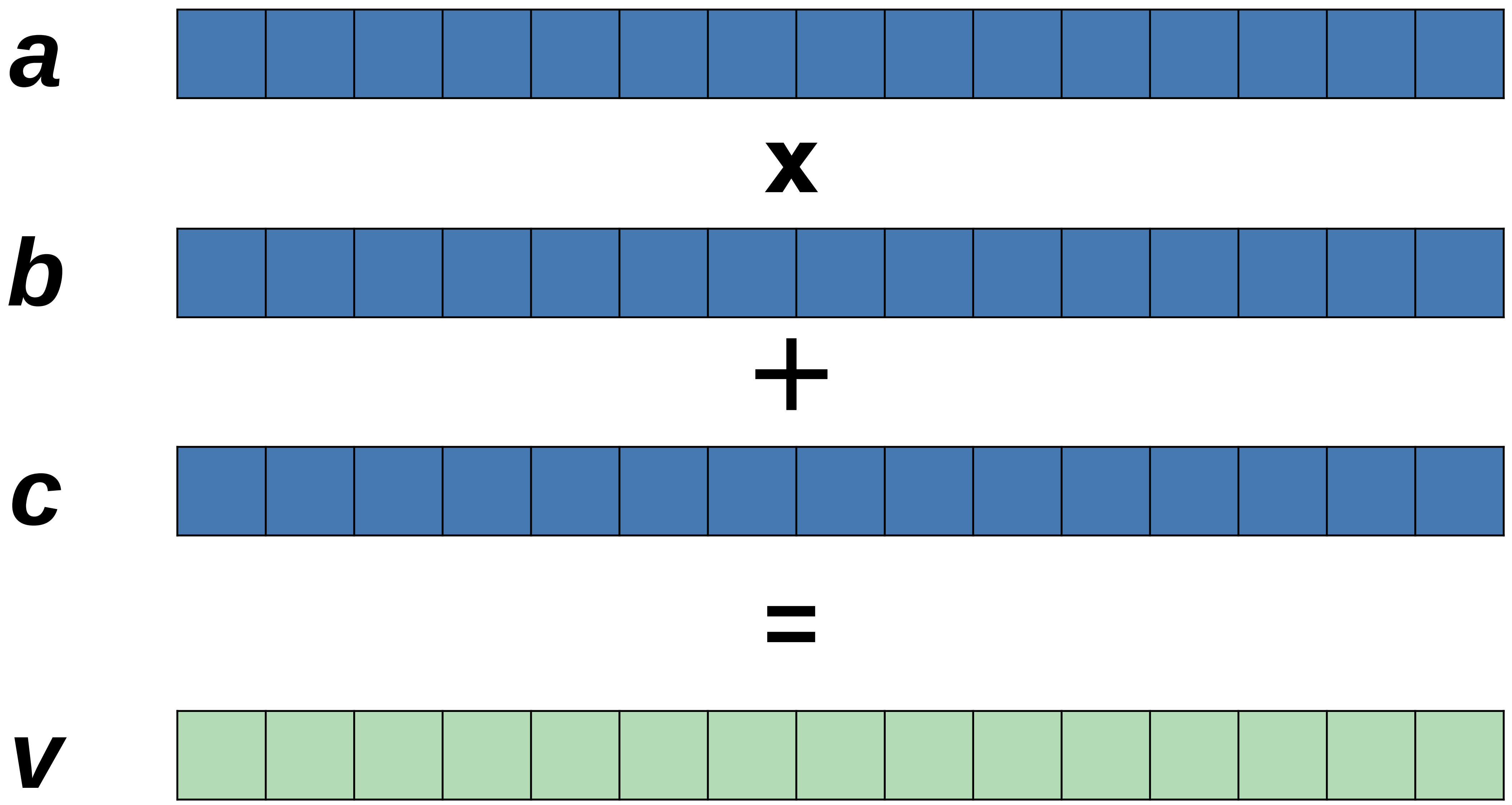
Why GPUs Run Fast?

- Three key ideas behind how modern GPU processing cores run code

- Knowing these concepts will help you:
    1. Understand GPU core designs
    2. Optimize performance of your parallel programs
    3. Gain intuition about what workloads might benefit from such a parallel architecture

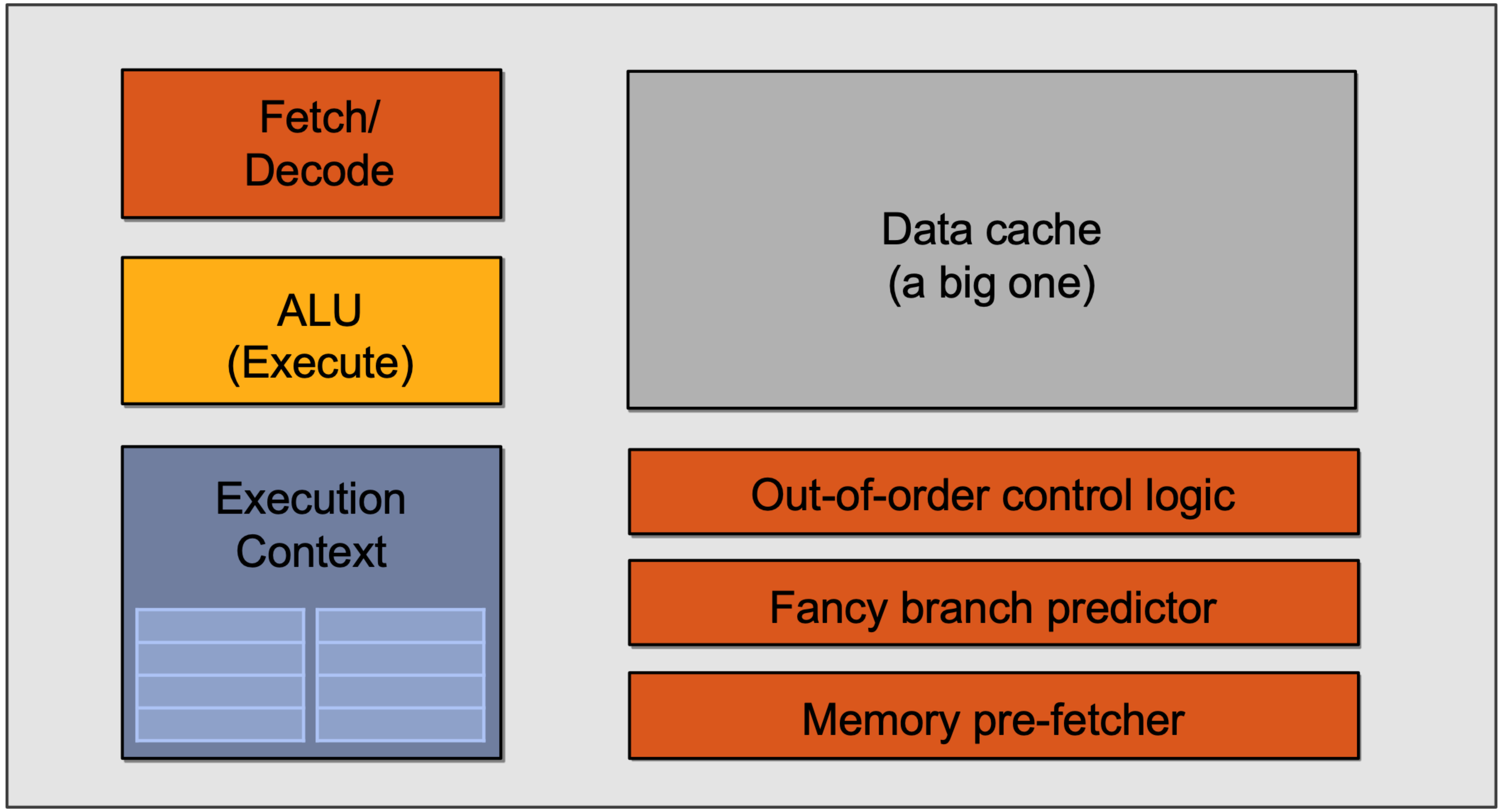
Example Program: Vector Multiply-Add

- Compute ***v*** = ***a*** **·** ***b*** + ***c*** *(**a**, **b**, **c** and **v** are vectors with a length of N)*



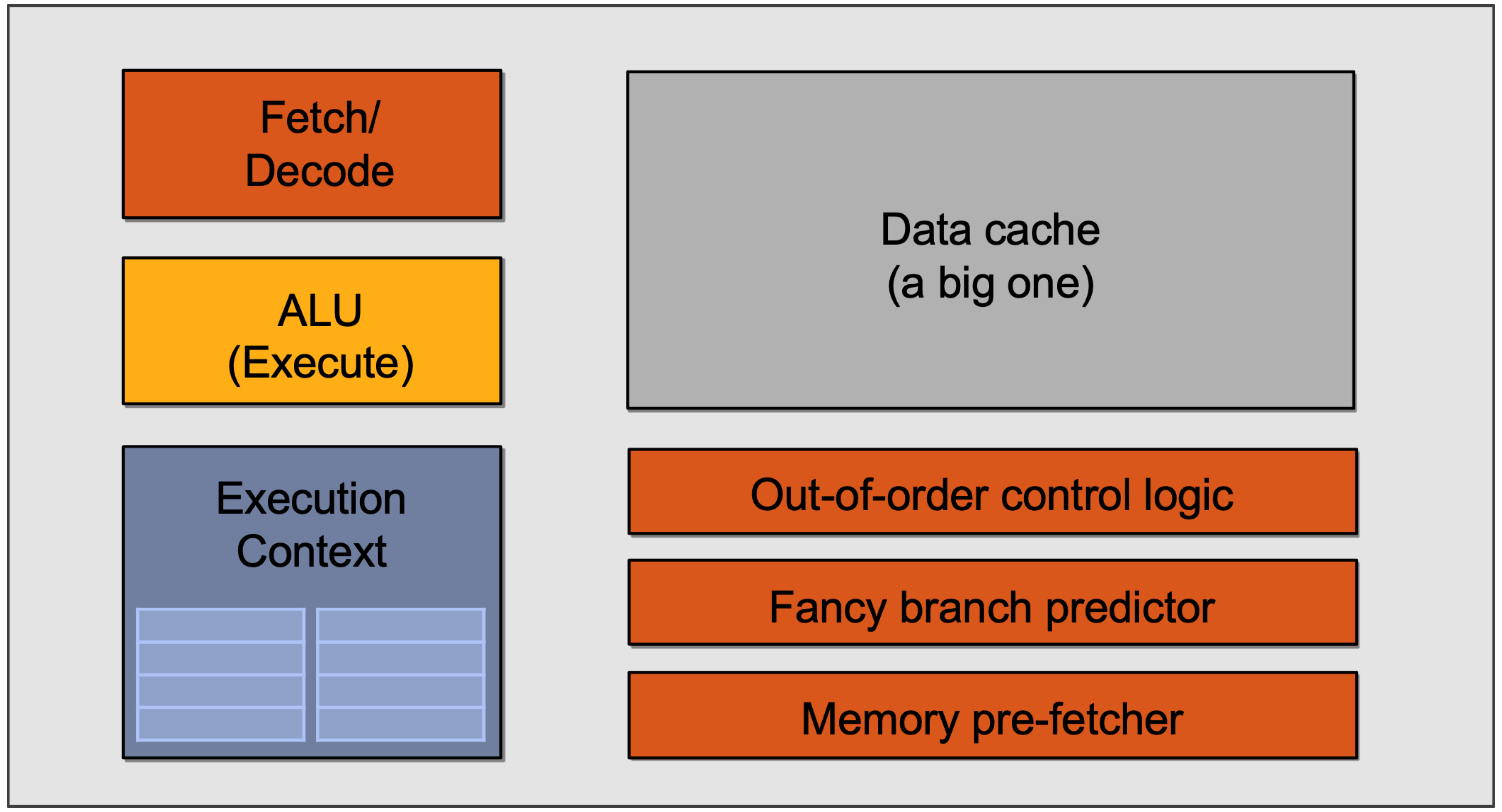
```
void mul_add (int N, float* a, float* b, float* c, float* v) {
      for (int i = 0; i < N; i++) {
            v[i] = a[i] * b[i] + c[i]
      }
}
```

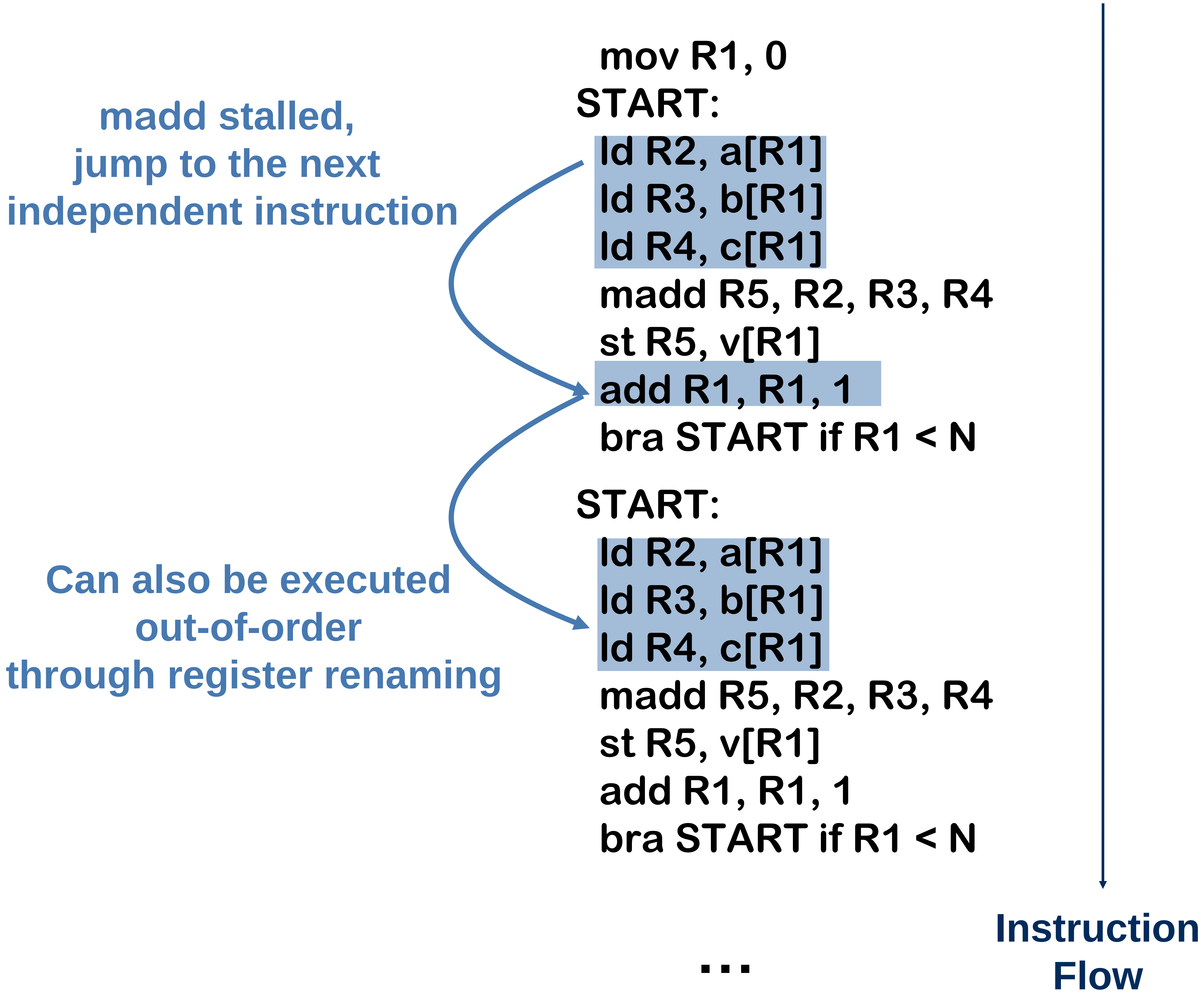
Single-core CPU Execution



```
  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
```

Single-core CPU Execution





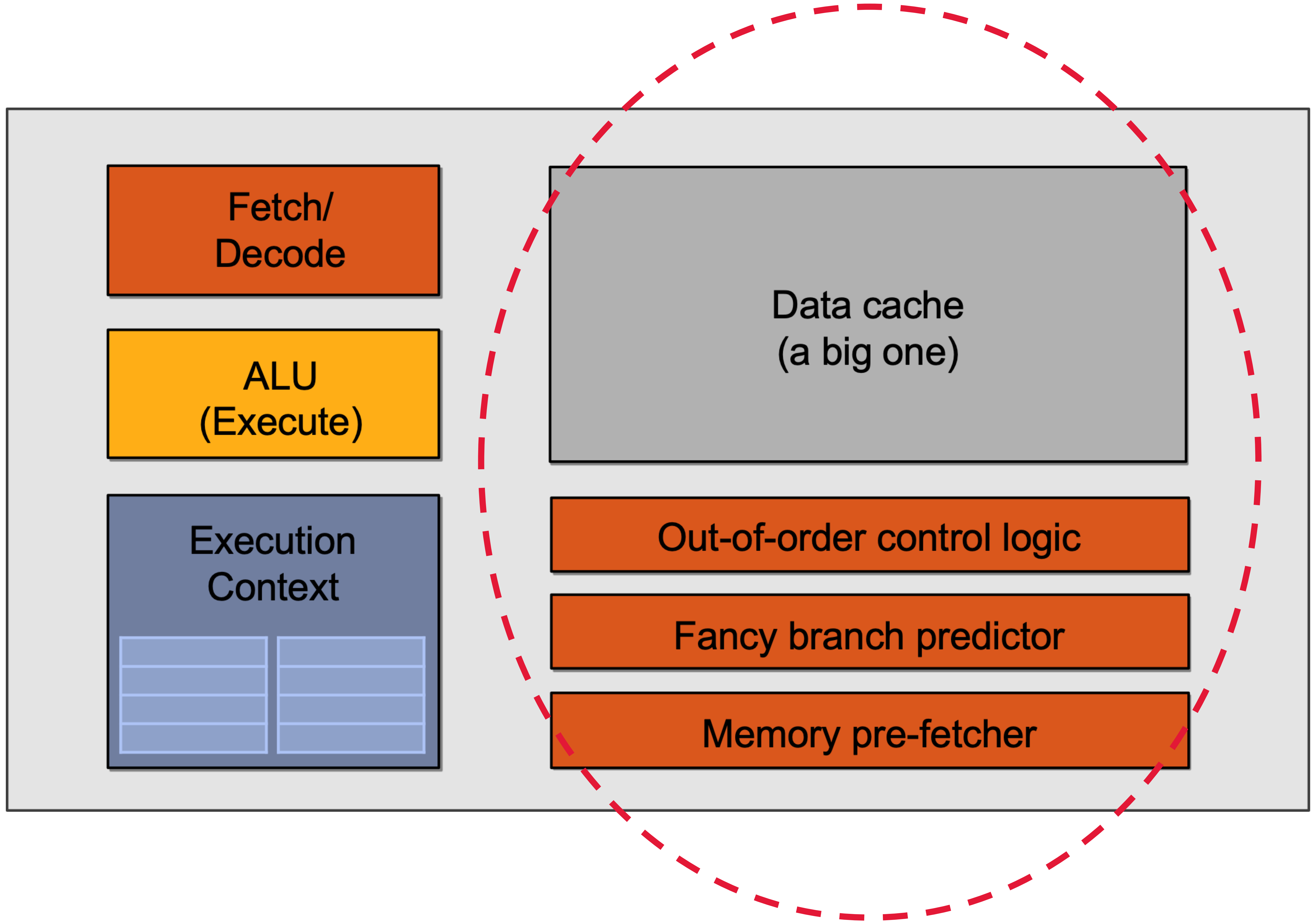
madd stalled, jump to the next independent instruction

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

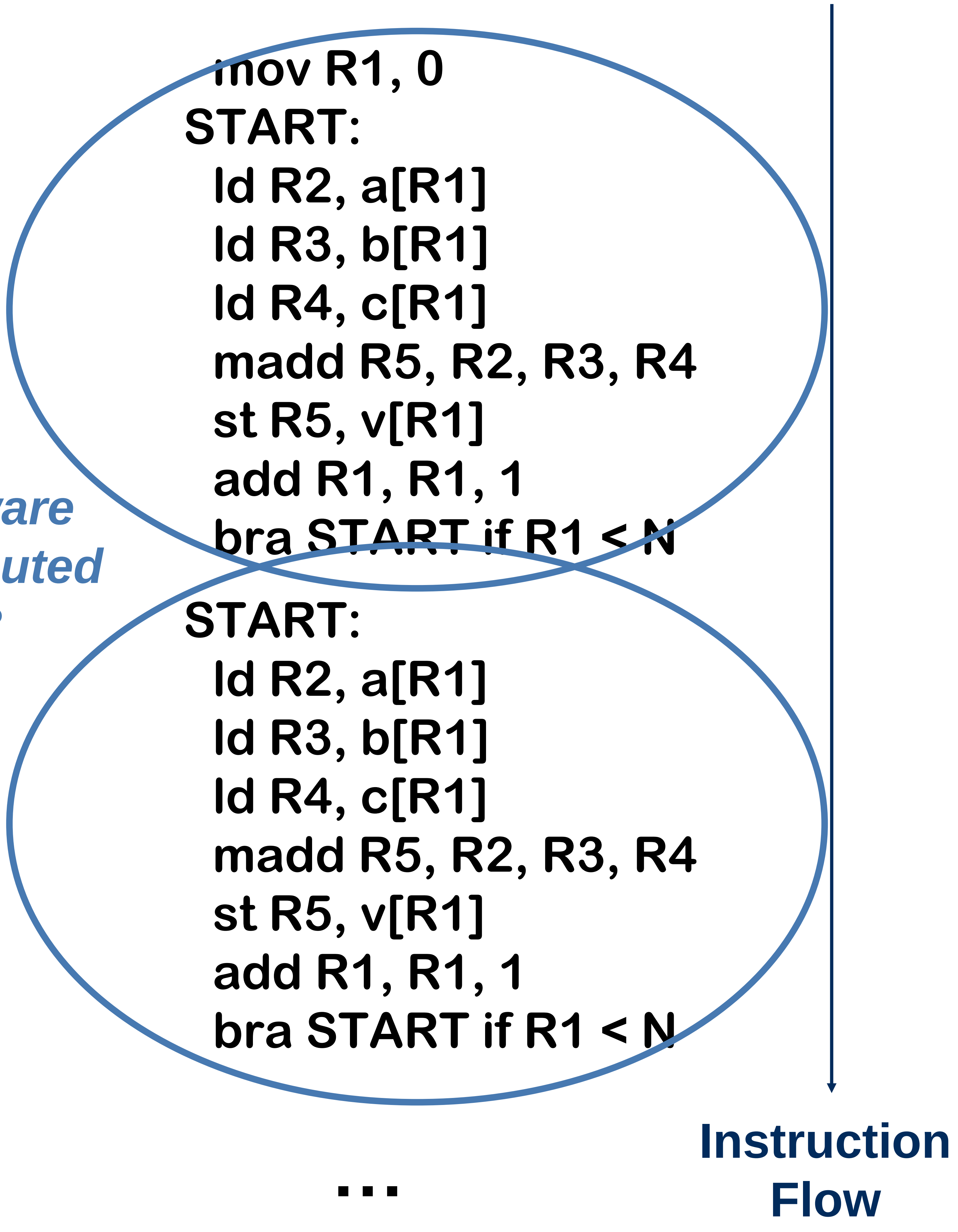
```
  mov R1, 0
START:
  ld R2, a[R1]
  ld R3, b[R1]
  ld R4, c[R1]
  madd R5, R2, R3, R4
  st R5, v[R1]
  add R1, R1, 1
  bra START if R1 < N

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

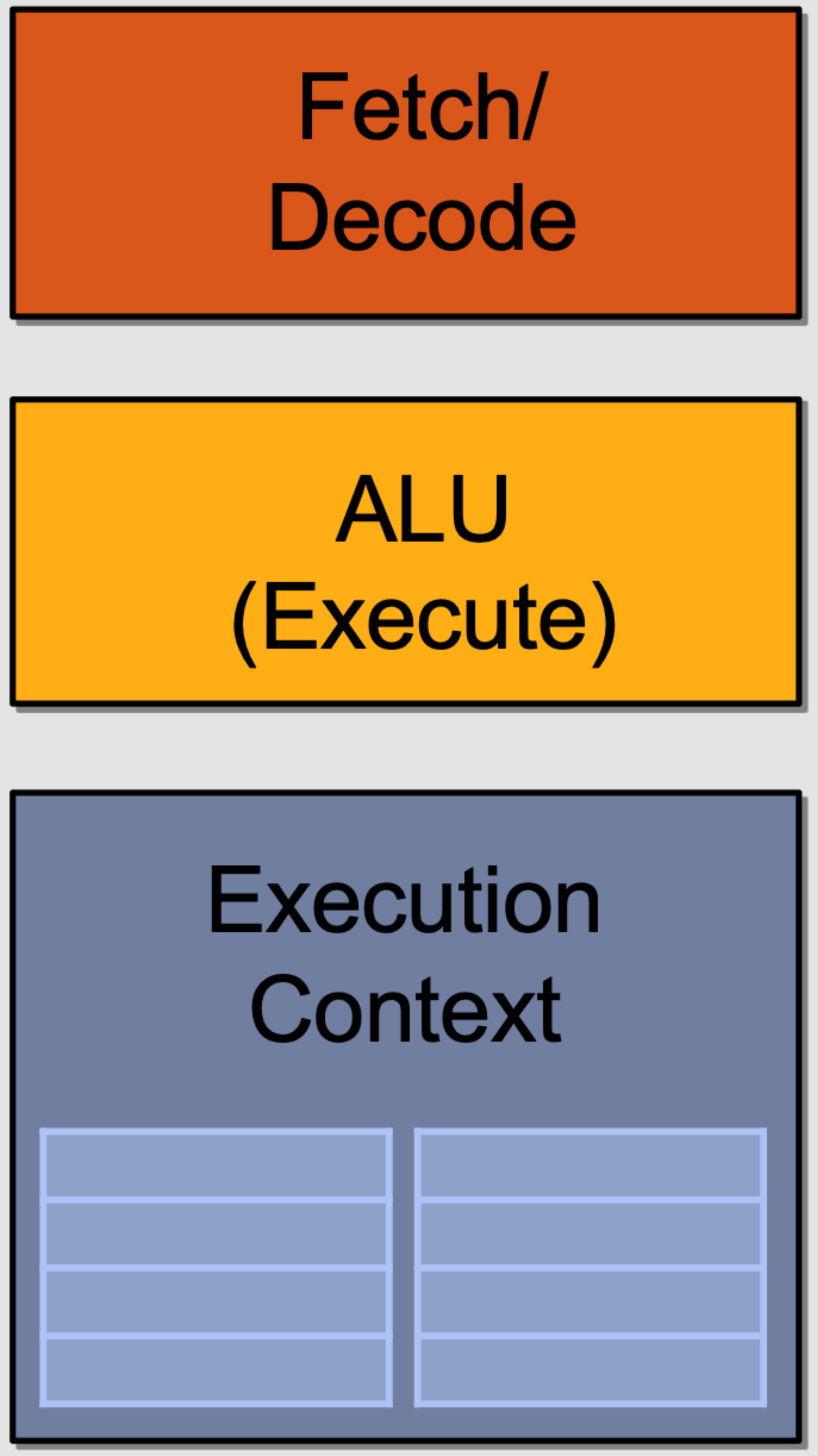


But what if we tell the hardware these two blocks can be executed in parallel to begin with?



```
 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

       ...
```

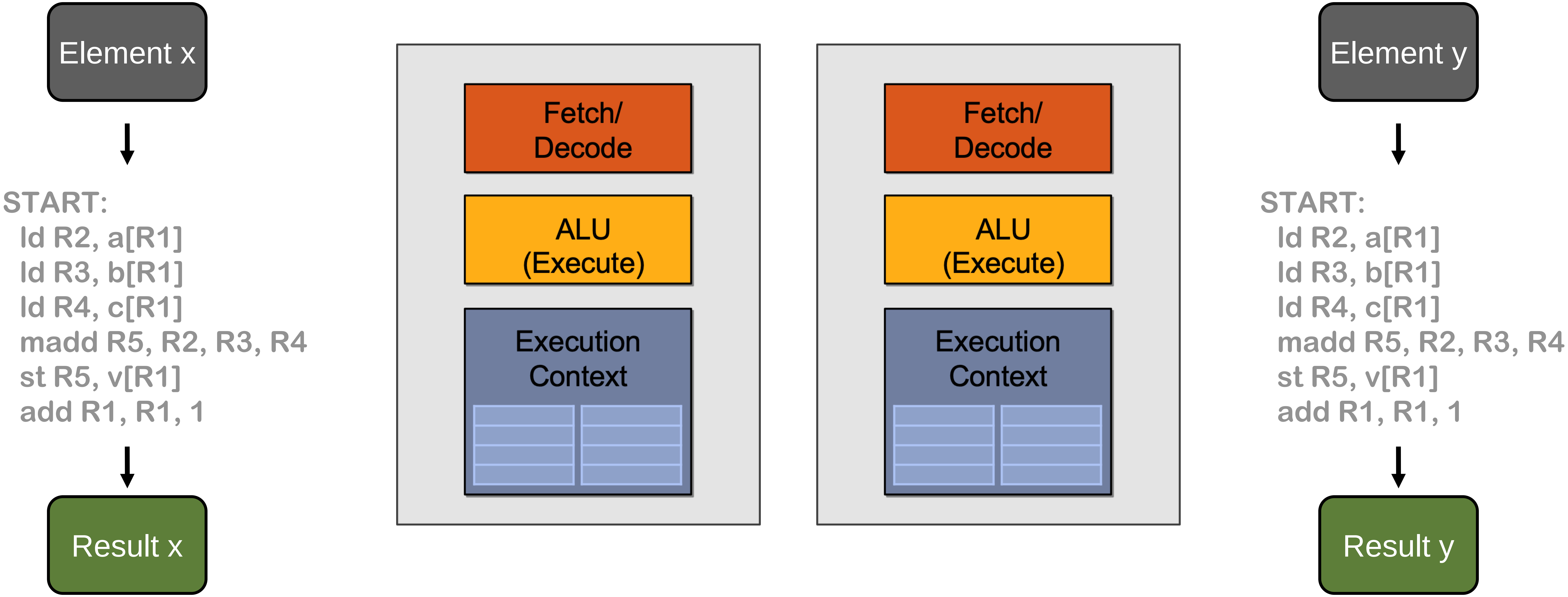
Slimming Down



Idea #1:
Use increasing transistor count to add more cores to the processor

… rather than use transistors to increase sophistication of processor logic that accelerates a single instruction stream (e.g., out-of-order and speculative operations)

Two cores (Two Elements in Parallel)



Element x

```
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
```

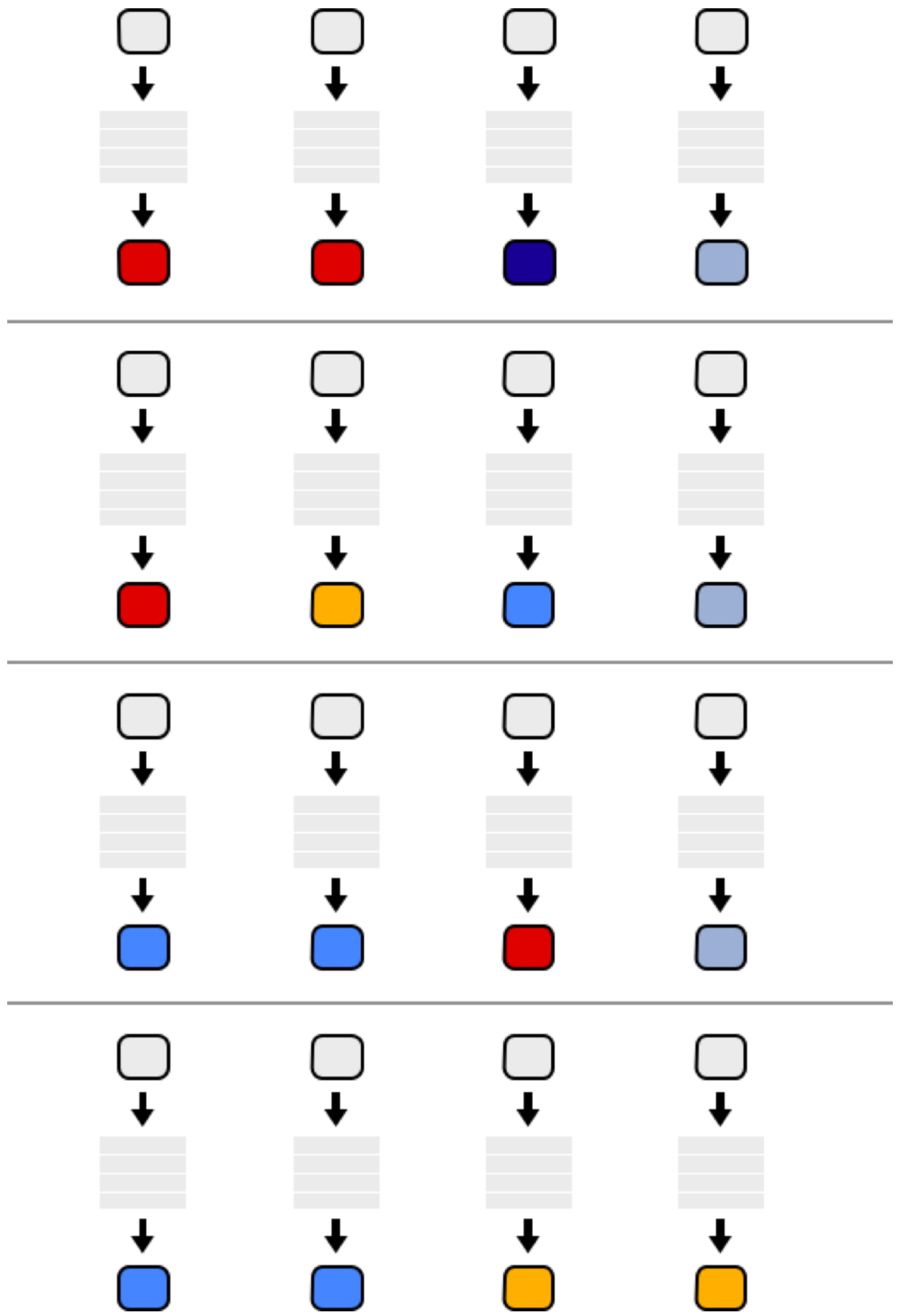
Result x

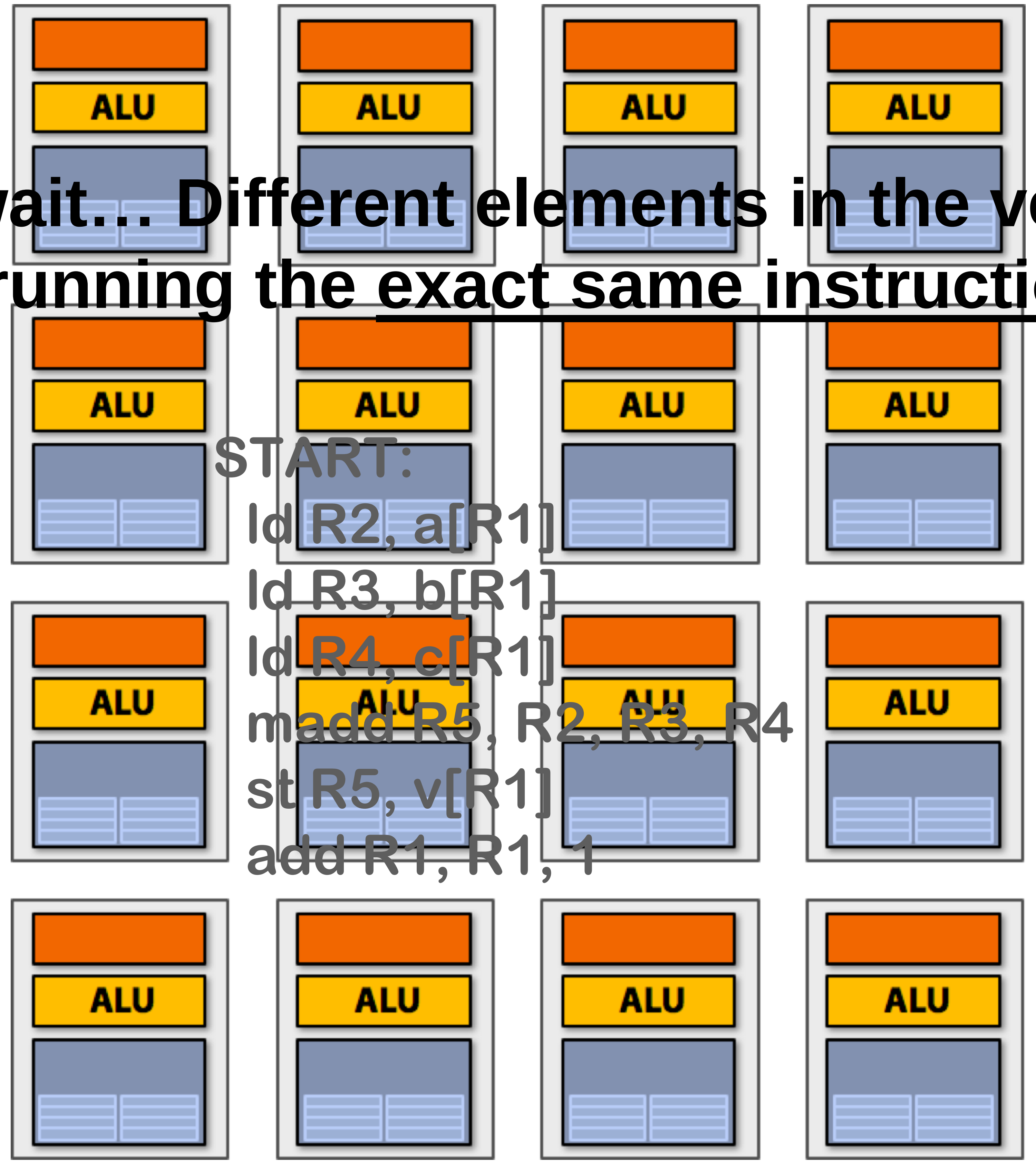
Element y

```
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
```

Result y

Sixteen Cores



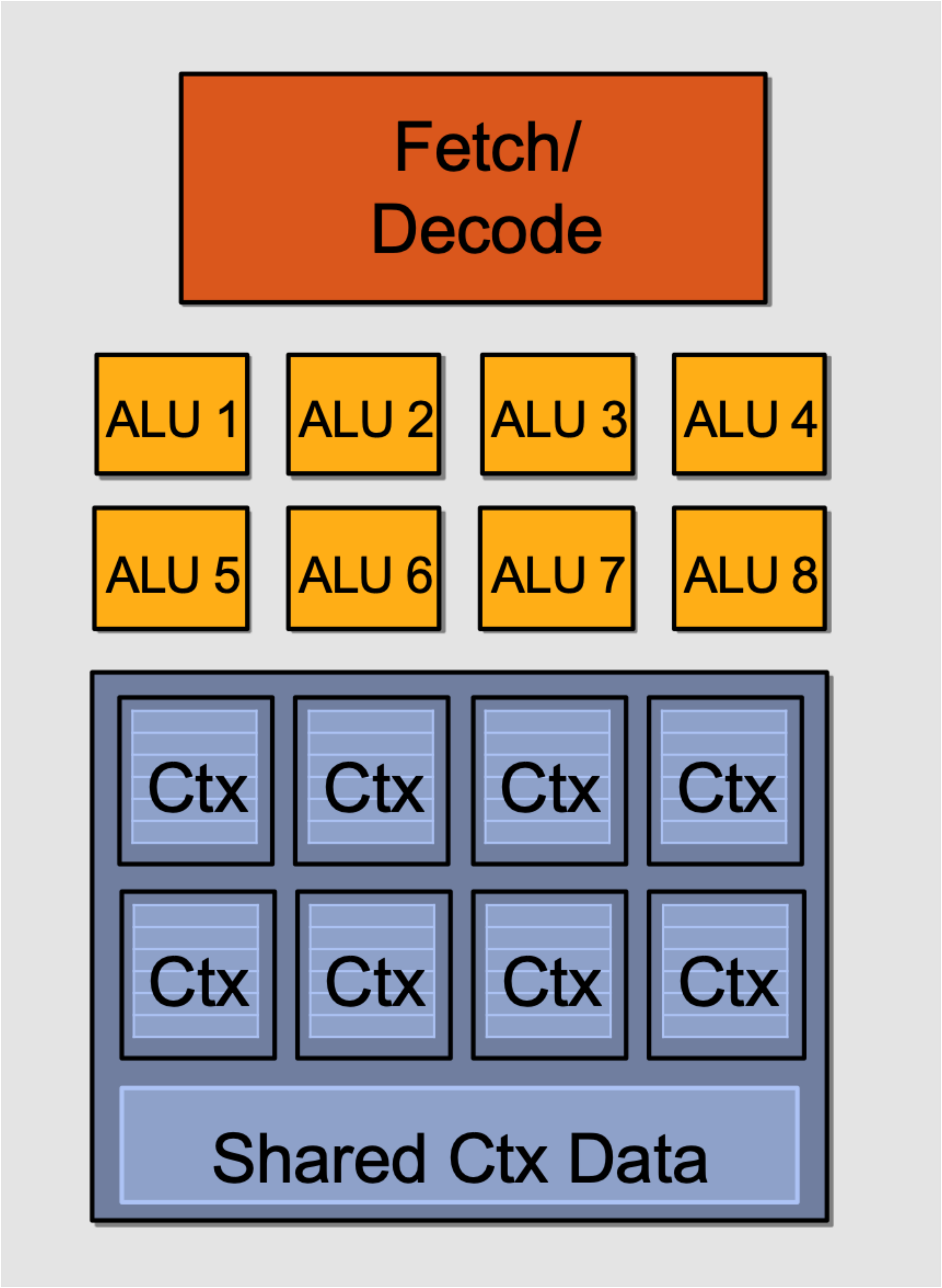


But wait… Different elements in the vector are running the exact same instructions!

```
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
```

16 cores = 16 simultaneous instruction streams

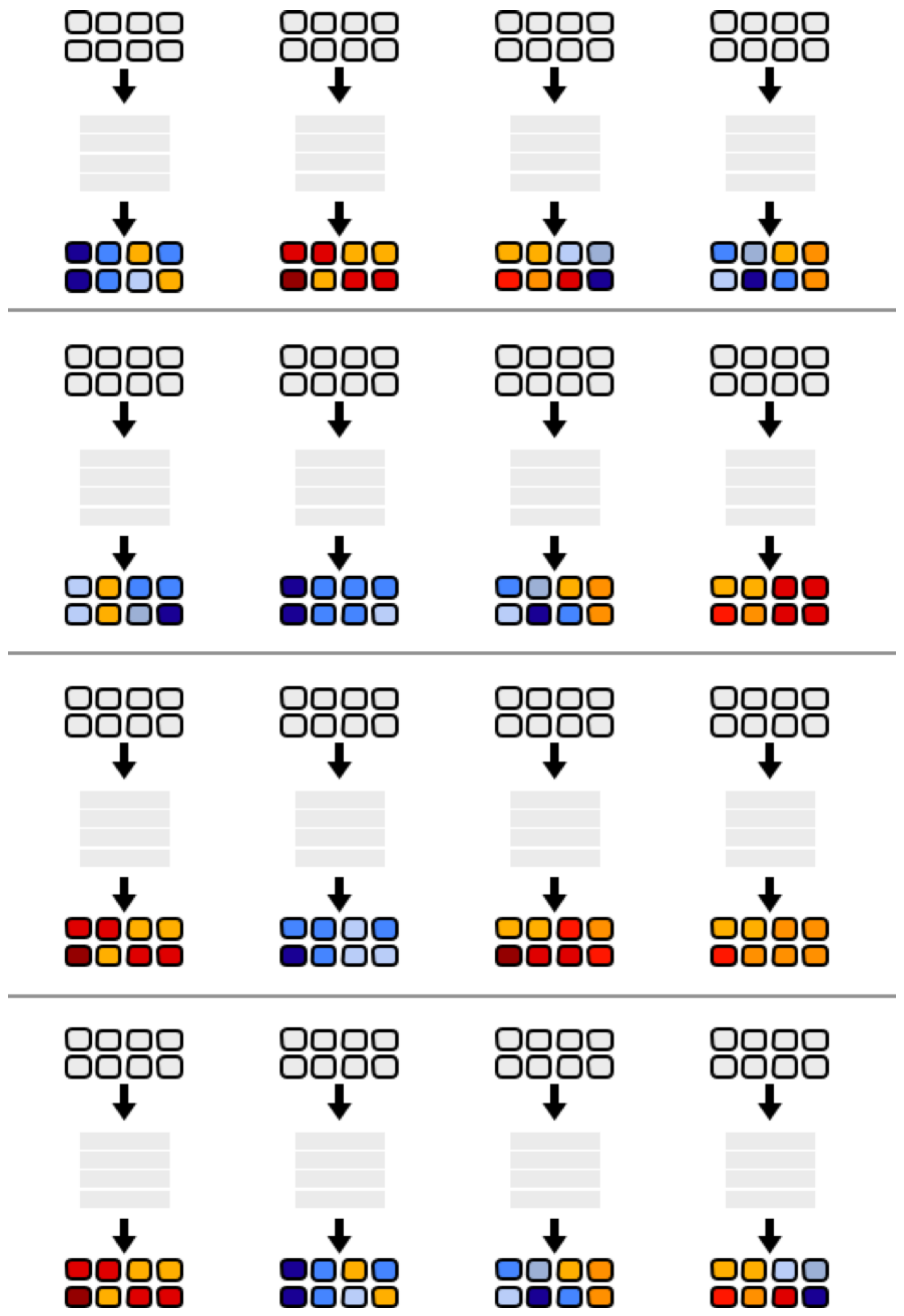
Instruction Stream Sharing



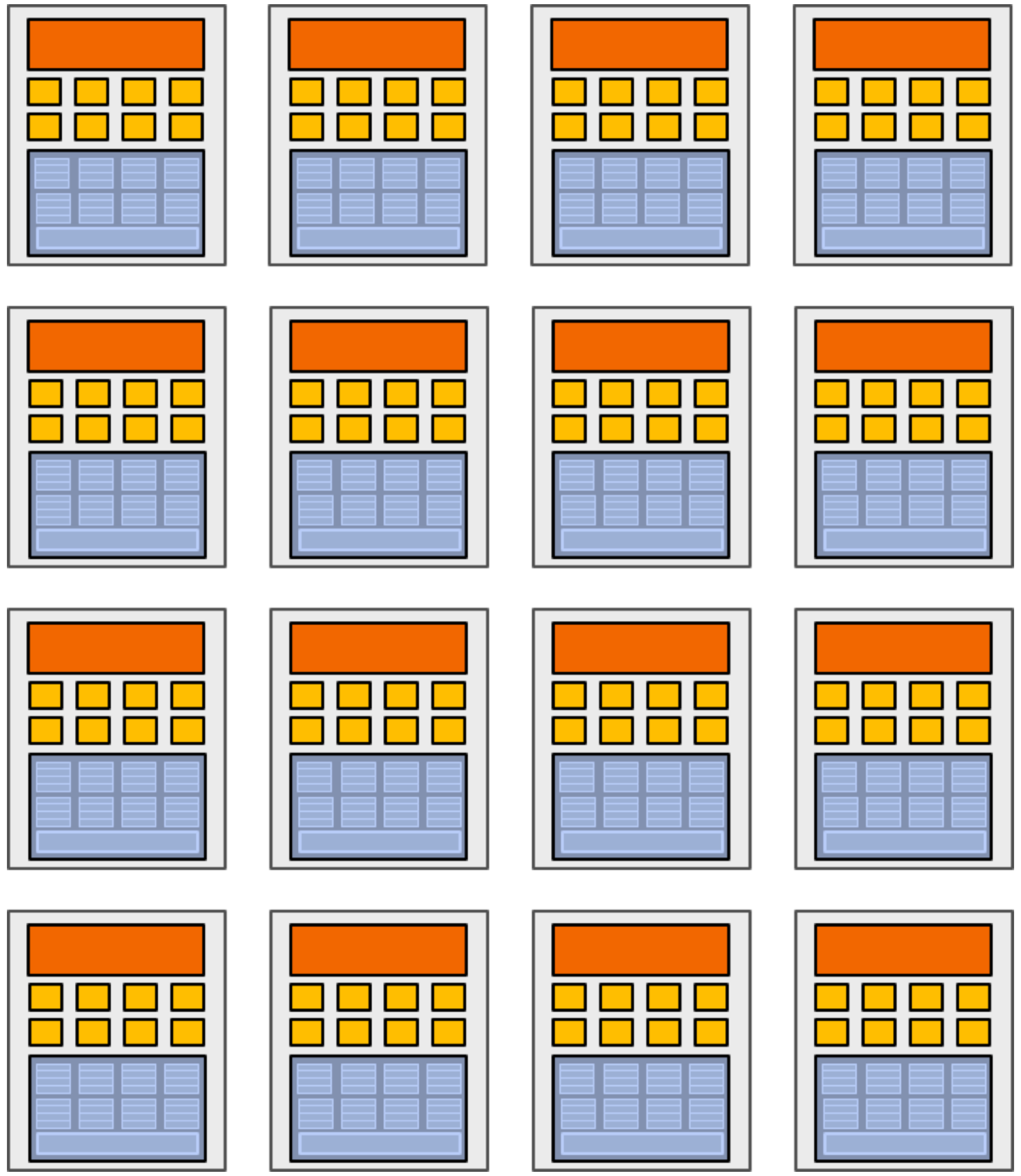
Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

SIMD processing!

128 Elements in Parallel

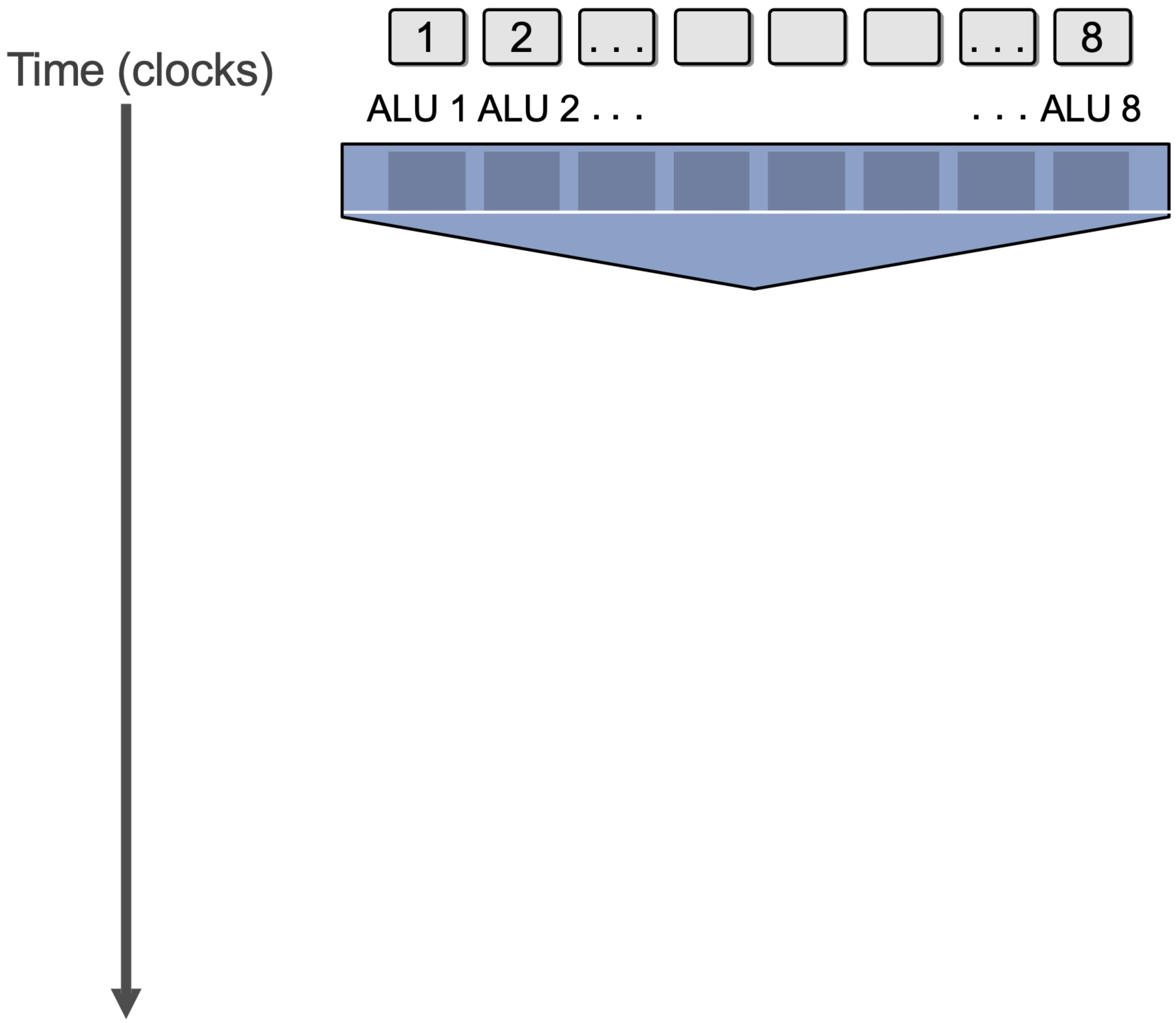


16 cores x 8 ALUs/core = 128 ALUs



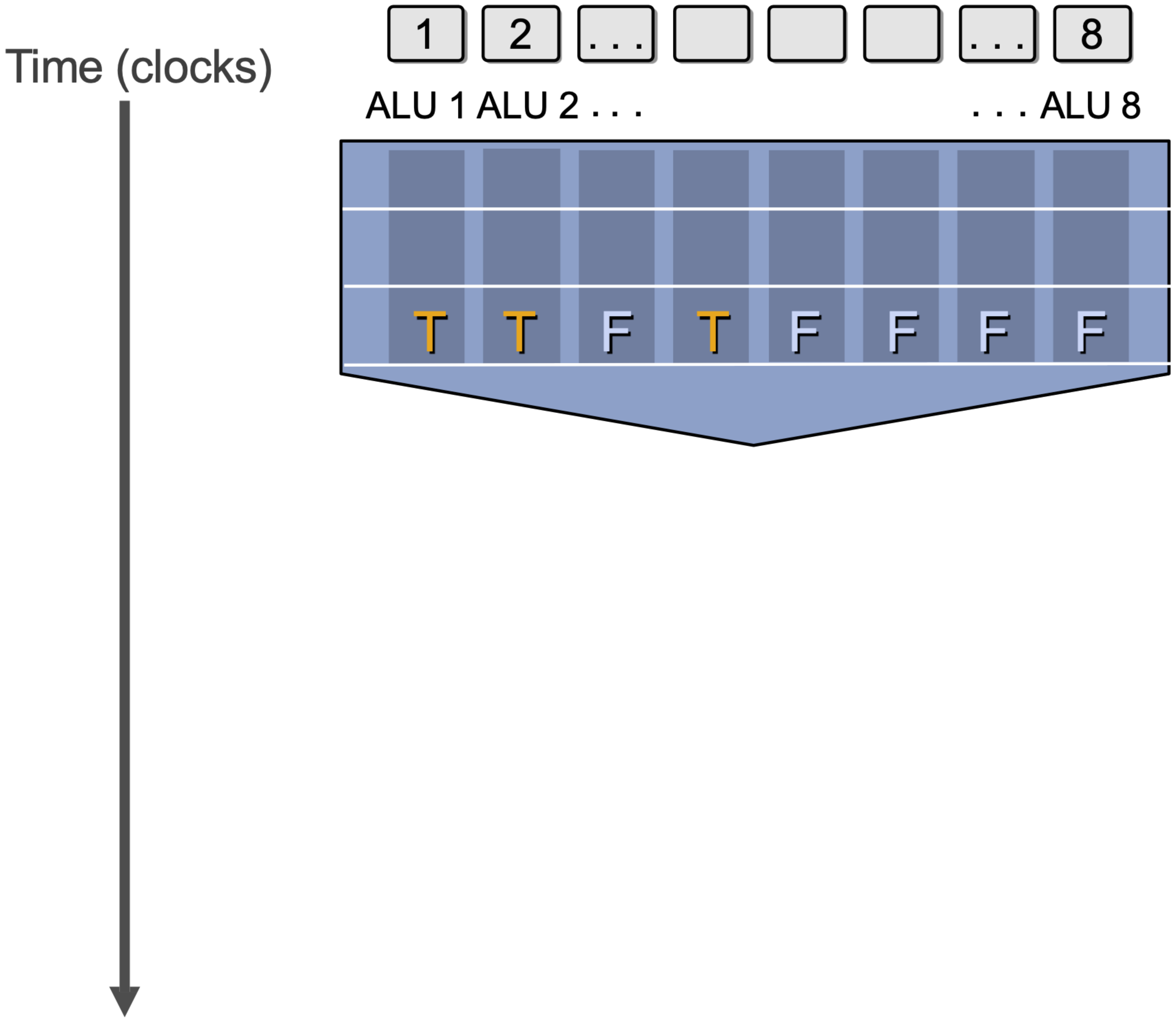
16 cores = 16 simultaneous instruction streams

What about Branches?



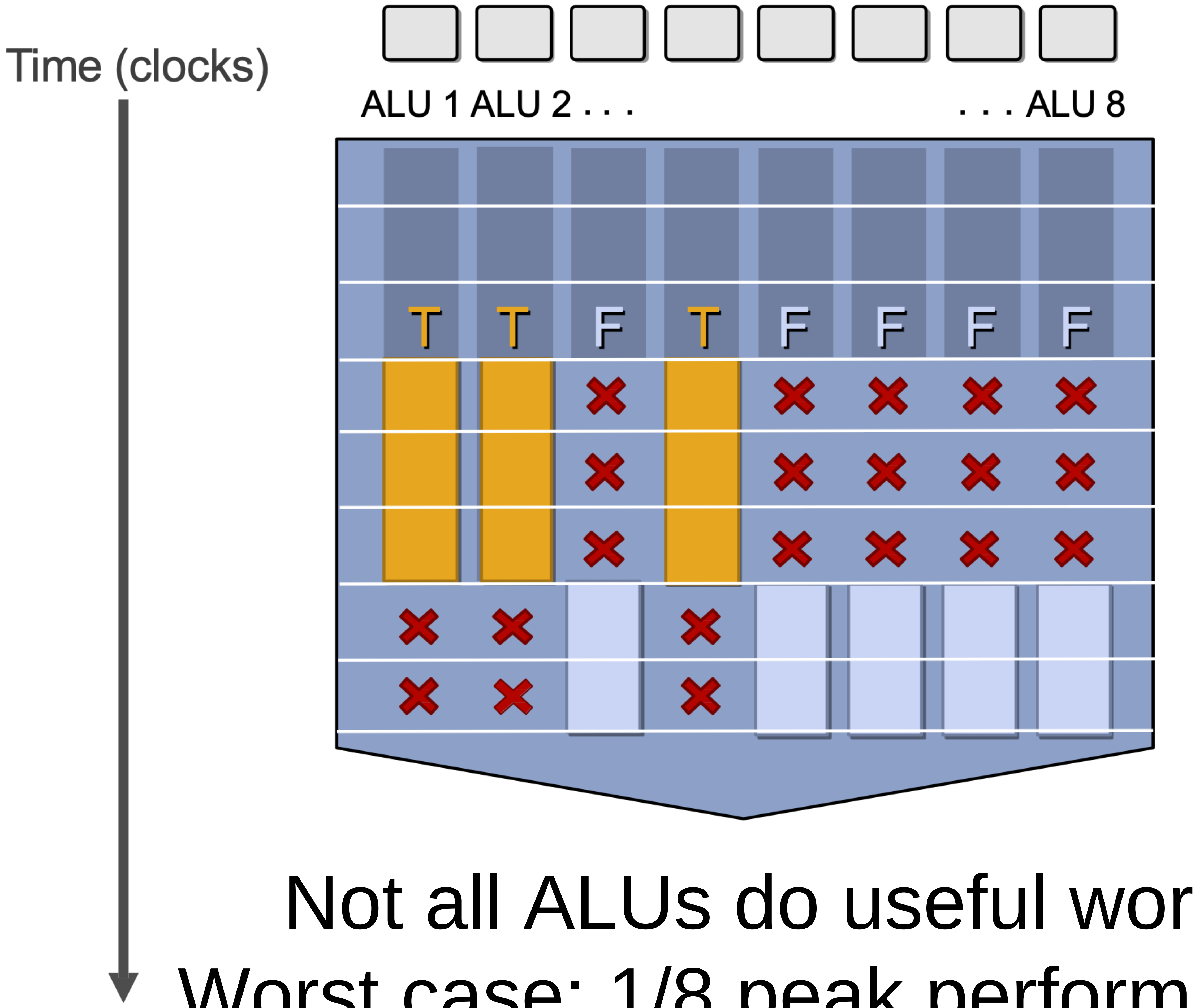
```
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```

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

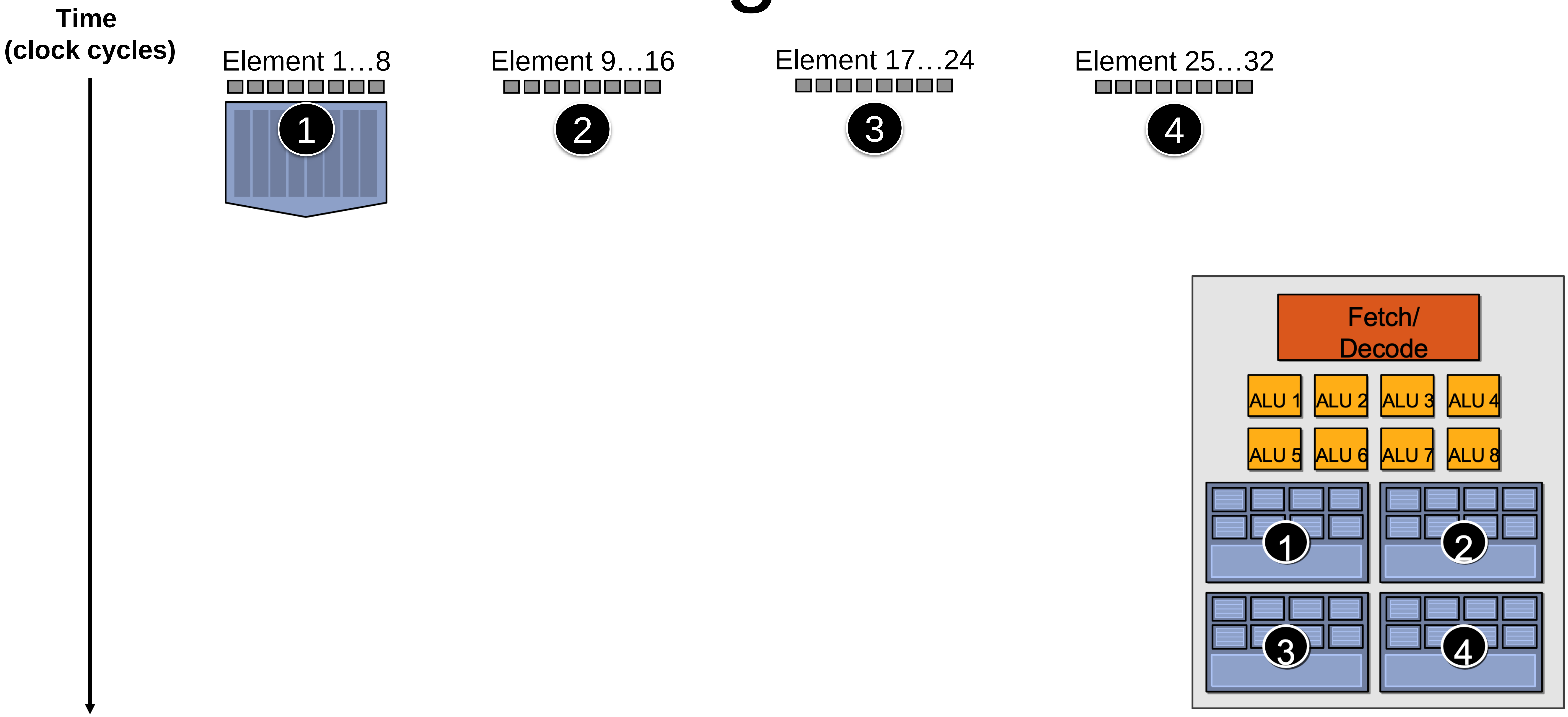
- “Implicit SIMD”
  - Compiler generates a scalar binary (scalar as opposed to vector instructions)
  - But N instances of the program are *always running* together on the processor
    i.e., execute(my_function, N) // execute my_function N times
  - Hardware (not compiler) is responsible for simultaneously 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!)

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)

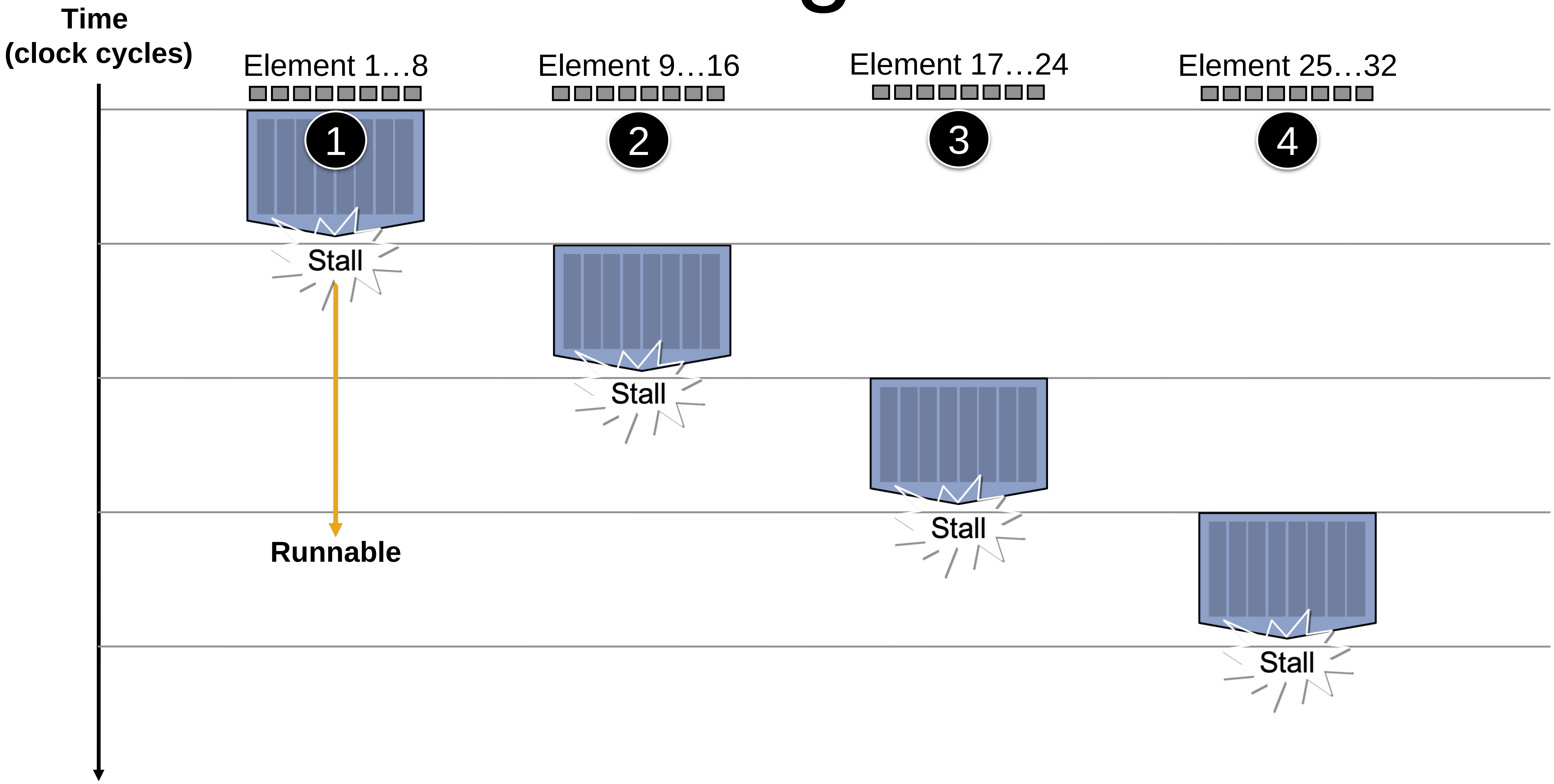
Element 1…8

Element 9…16

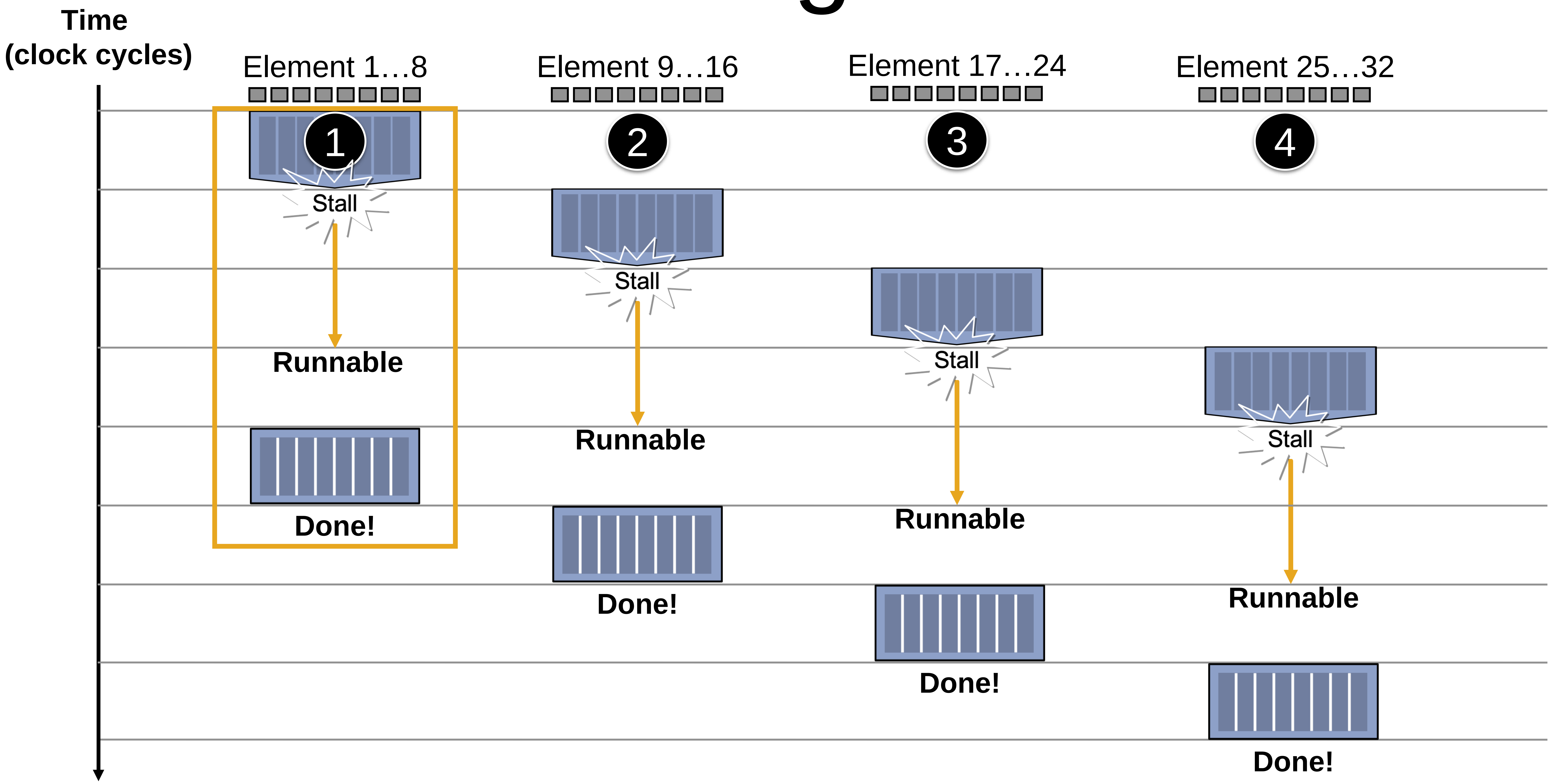
Element 17…24

Element 25…32

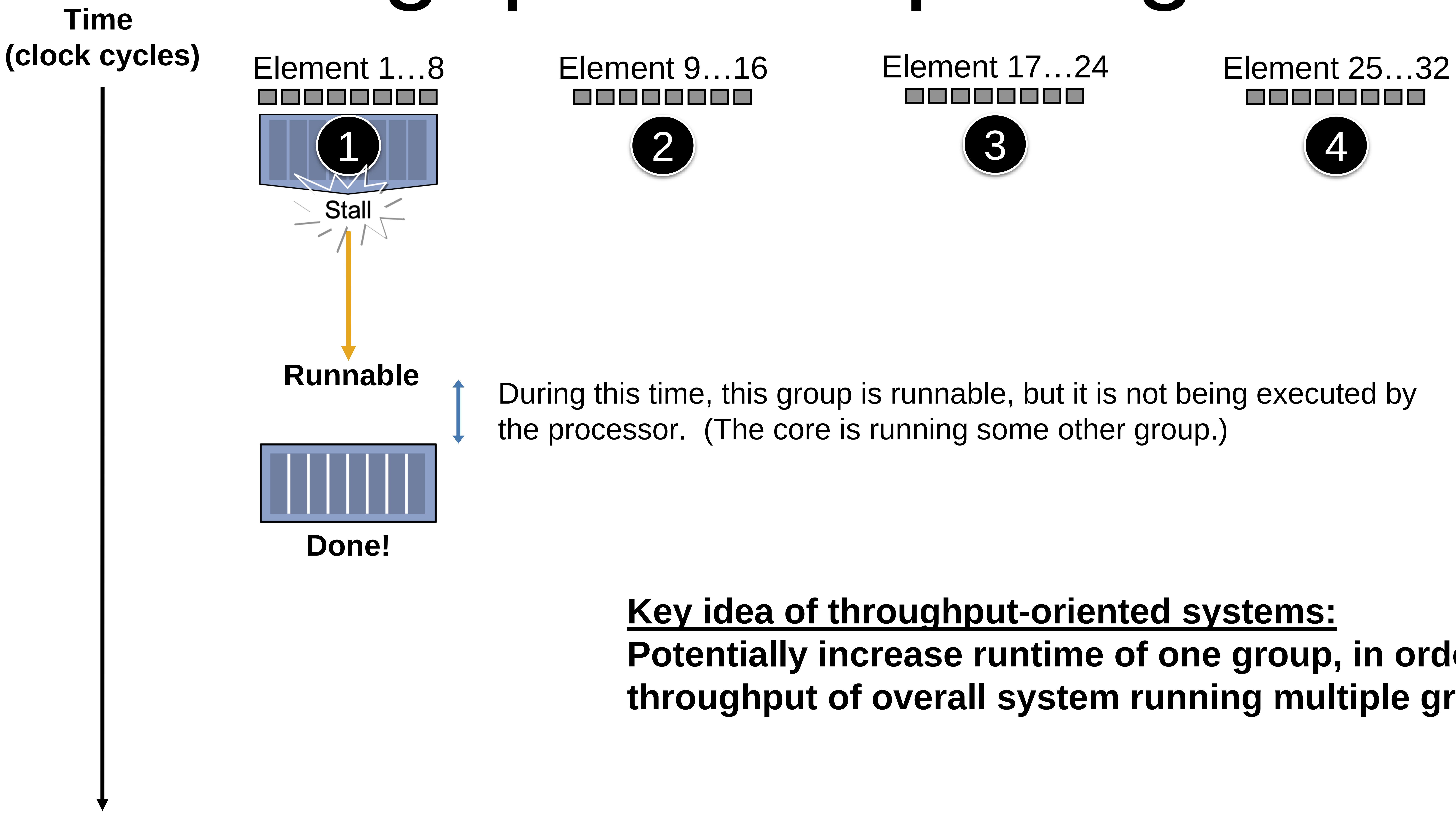
Hiding Stalls



Hiding Stalls



Throughput Computing Trade-off



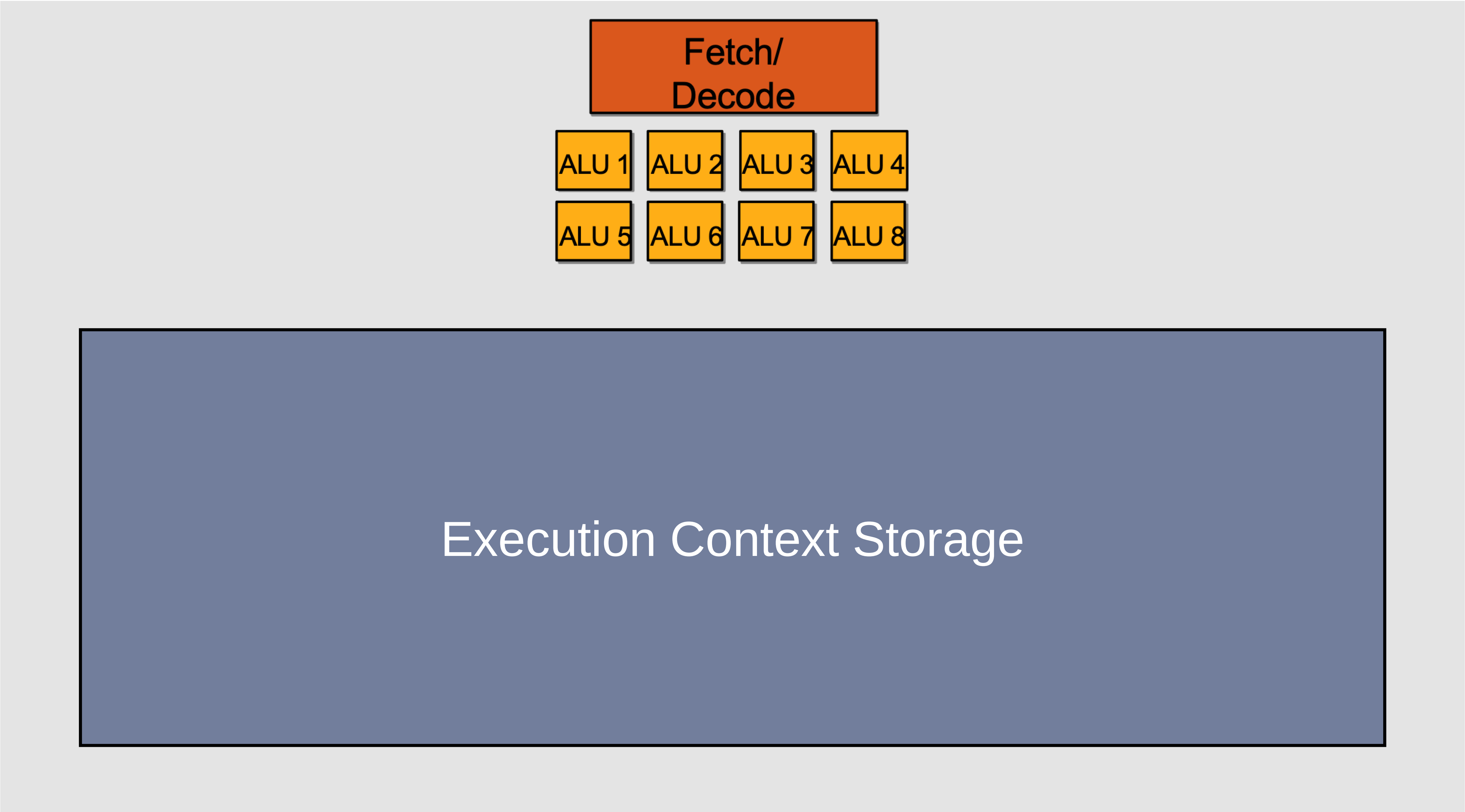
During this time, this group is runnable, but it is not being executed by the processor. (The core is running some other group.)

<u>Key idea of throughput-oriented systems:</u>

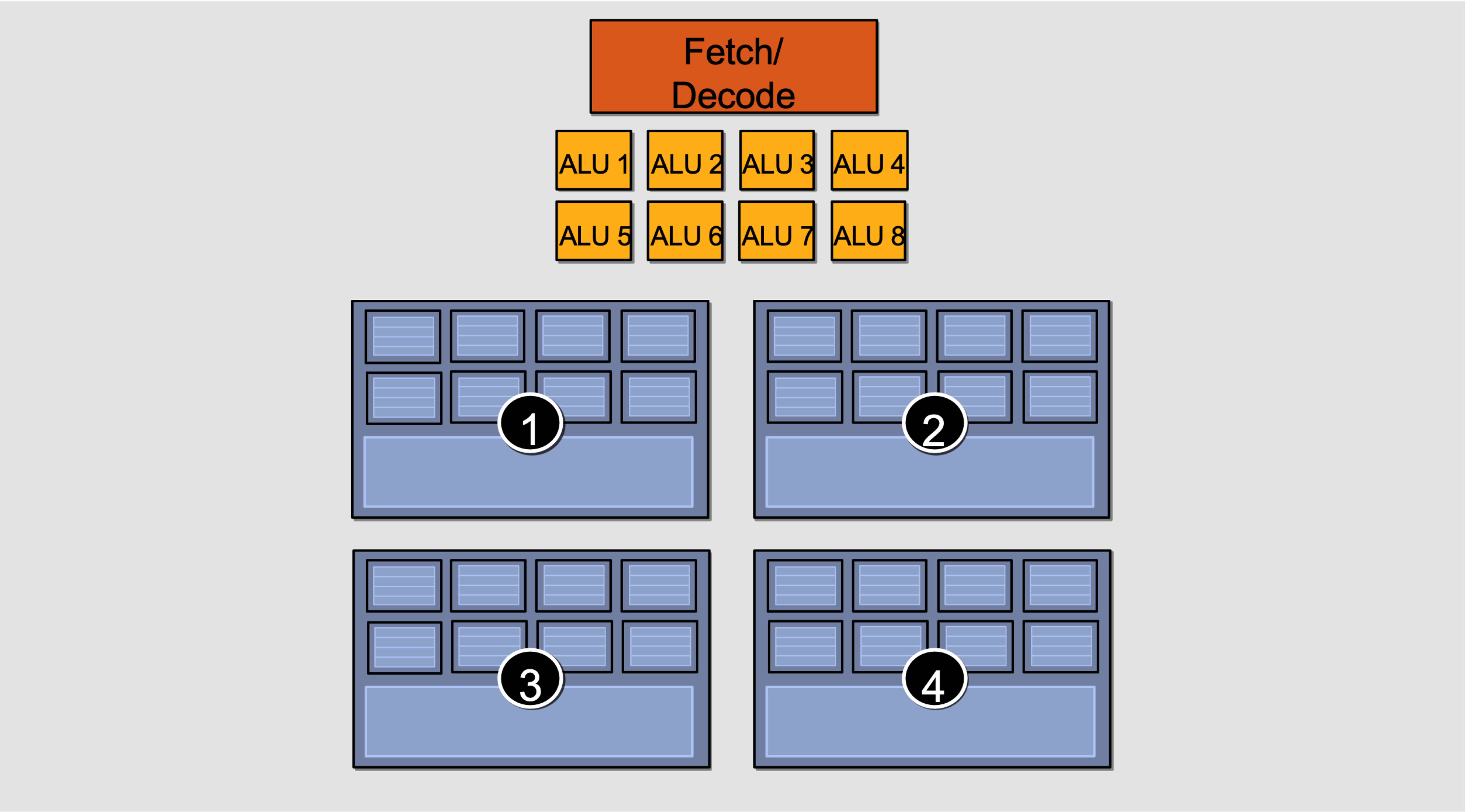
Potentially increase runtime of one group, in order to increase throughput of overall system running multiple groups.

Storing Execution Contexts

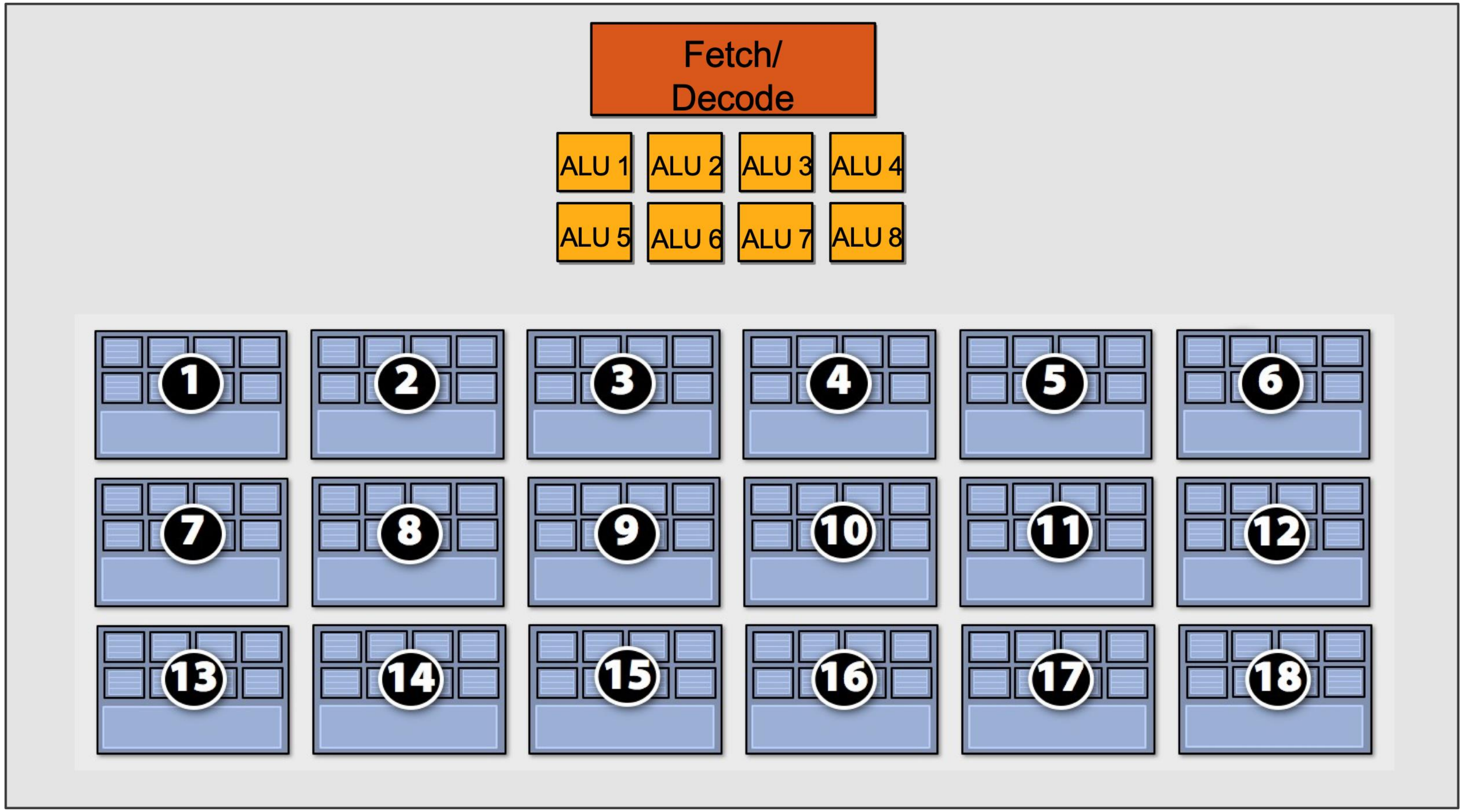
- Consider on-chip storage of execution contexts a finite resource
- Resource consumption of each thread group is program-dependent



Four Large Contexts (*Low* Latency Hiding)



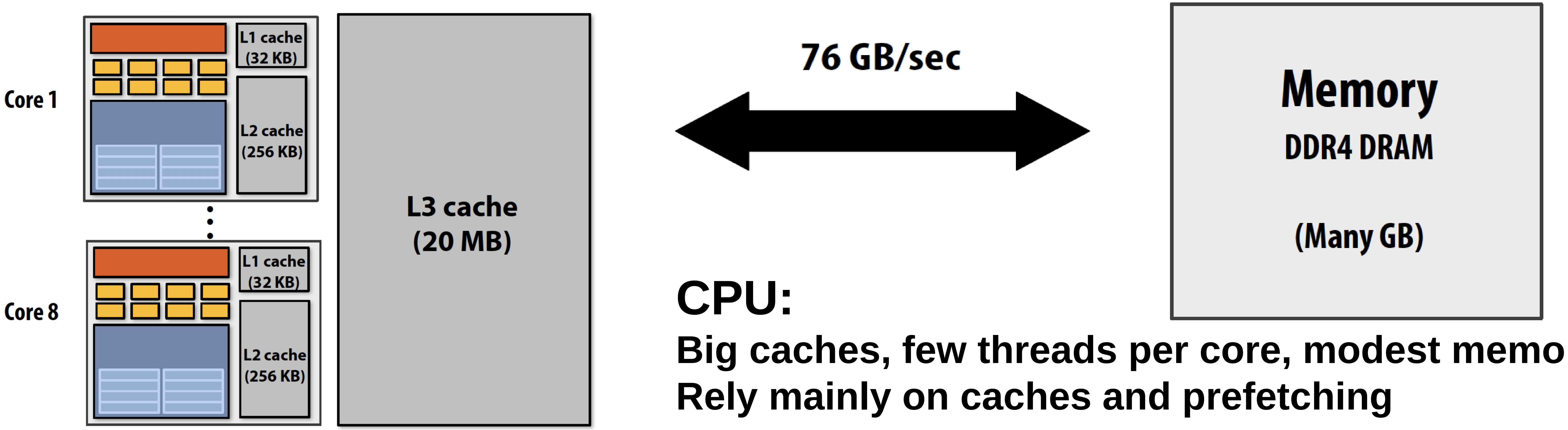
Eighteen Small Contexts (*High* Latency Hiding)



Summary: Three Key Ideas

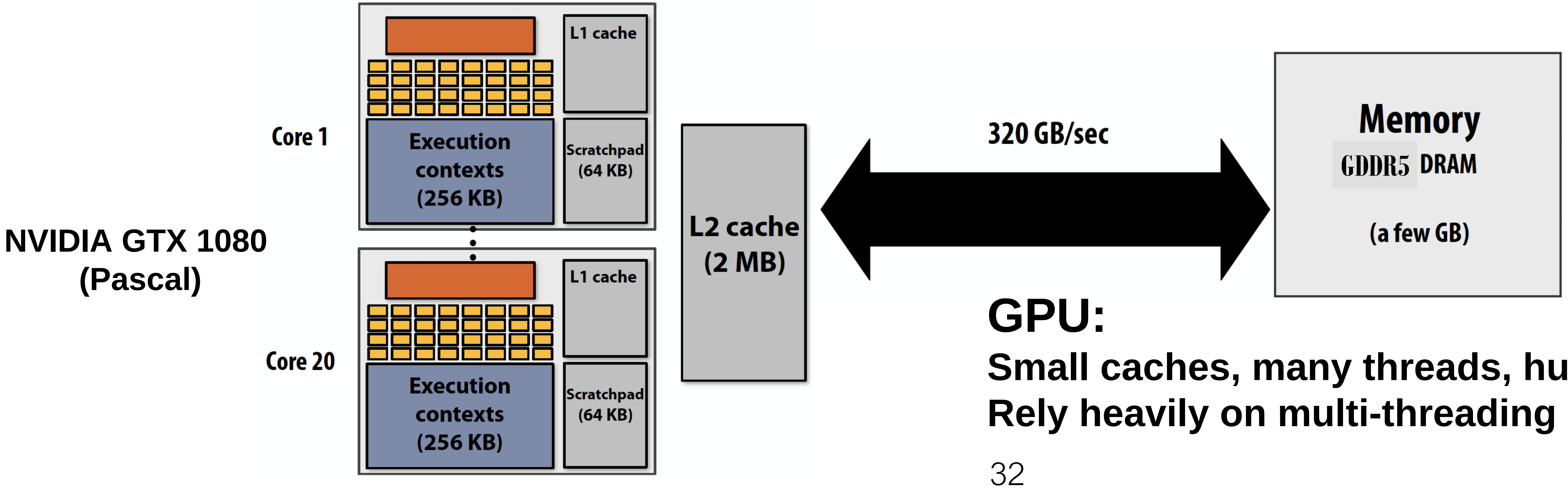
1. Use many “slimmed down cores” to run in parallel
2. Pack cores full of ALUs (by sharing instruction stream on multiple data)
3. Avoid latency stalls by interleaving execution of many groups of threads
   - When one group stalls, work on another group

CPU v.s. GPU Memory Hierarchies



CPU:

Big caches, few threads per core, modest memory BW
Rely mainly on caches and prefetching

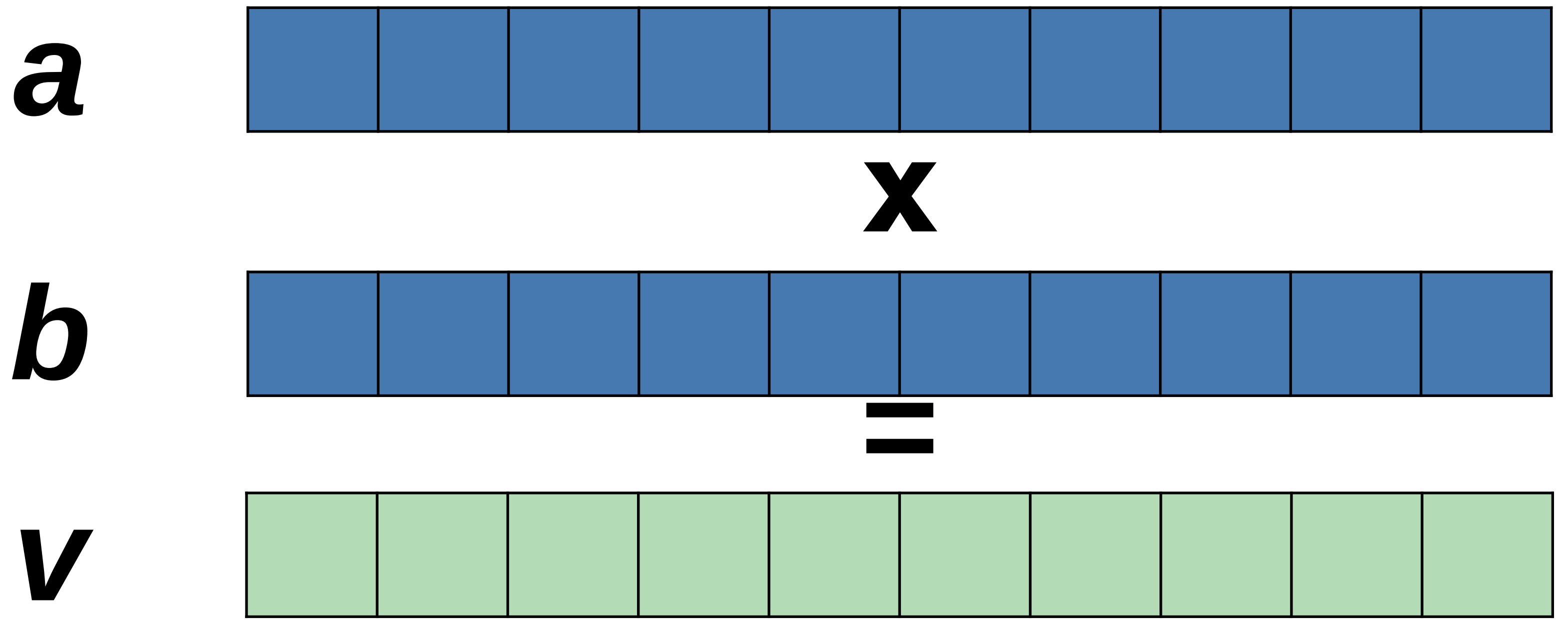


GPU:

Small caches, many threads, huge memory BW
Rely heavily on multi-threading for performance

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

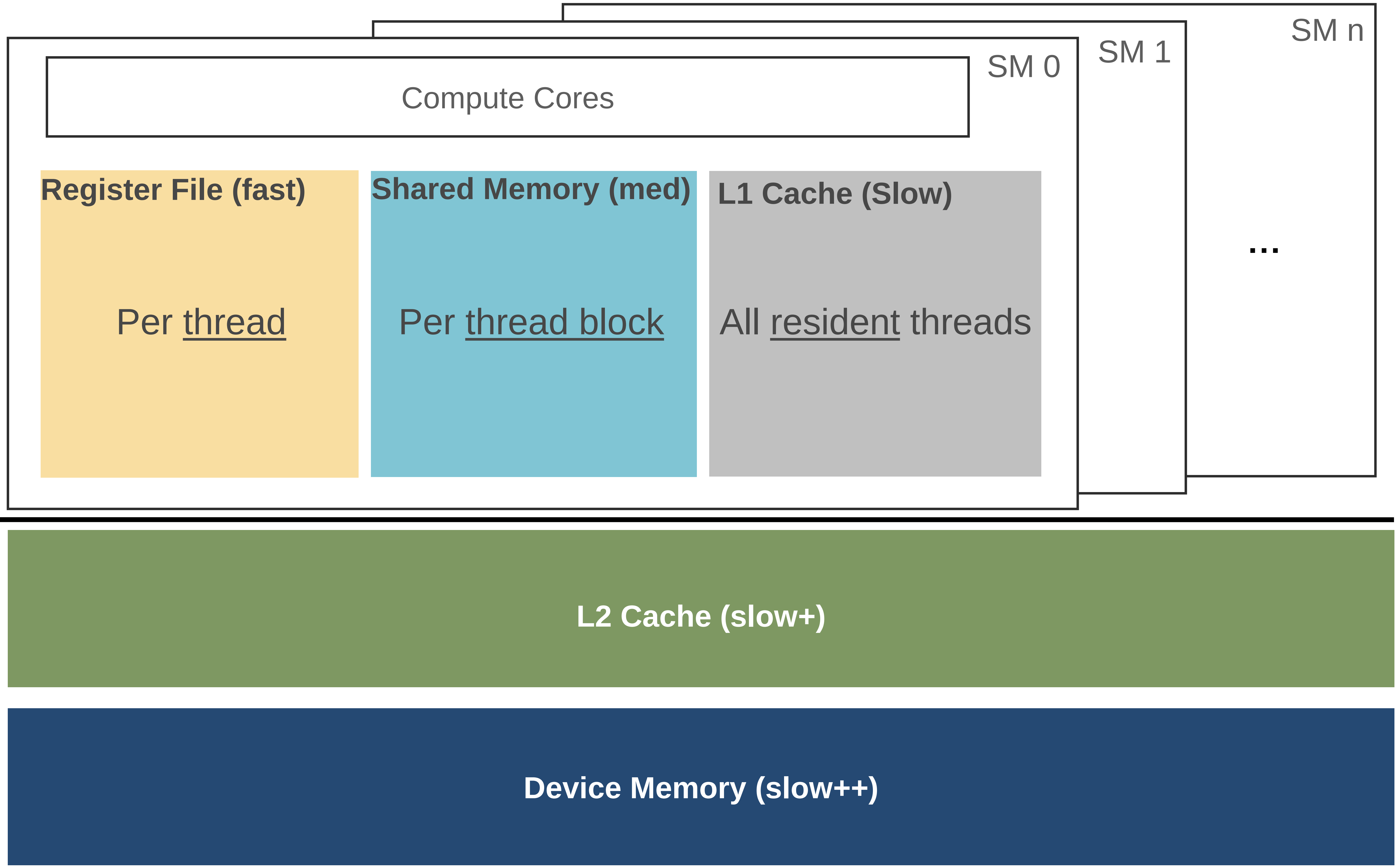
Performant parallel programs will:

- 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:

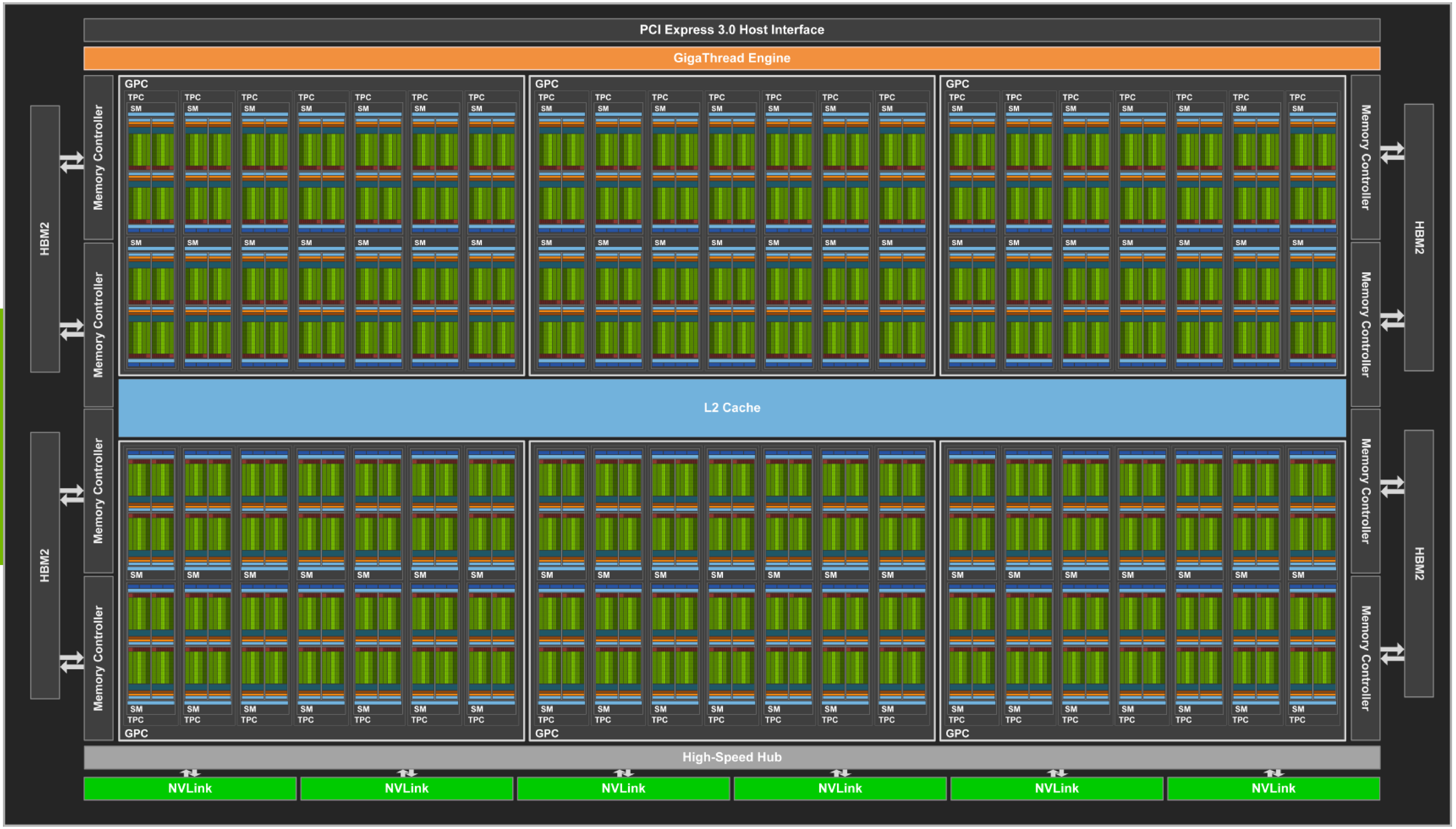
- L2 cache
    - Bandwidth filter for DRAM rather than reducing latency as in CPUs
- Device memory (DRAM)
    - Several spaces: global memory, texture memory, local memory
    - Different spaces have different caching policies

Modern GPU Architecture (Volta 2017)

21B transistors
815 mm²

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

Review #6

GPUs and the Future of Parallel Computing

Steve Keckler et al., *IEEE Micro 2011*

Due Oct. 22nd

CSC 2224: Parallel Computer Architecture and Programming GPU Architecture: Introduction

Prof. Gennady Pekhimenko

University of Toronto

Fall 2020

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan

CSC 2224: Parallel Computer Architecture and Programming
GPU Programming

Prof. Gennady Pekhimenko

University of Toronto

Fall 2020

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan

GPU Teaching Kit

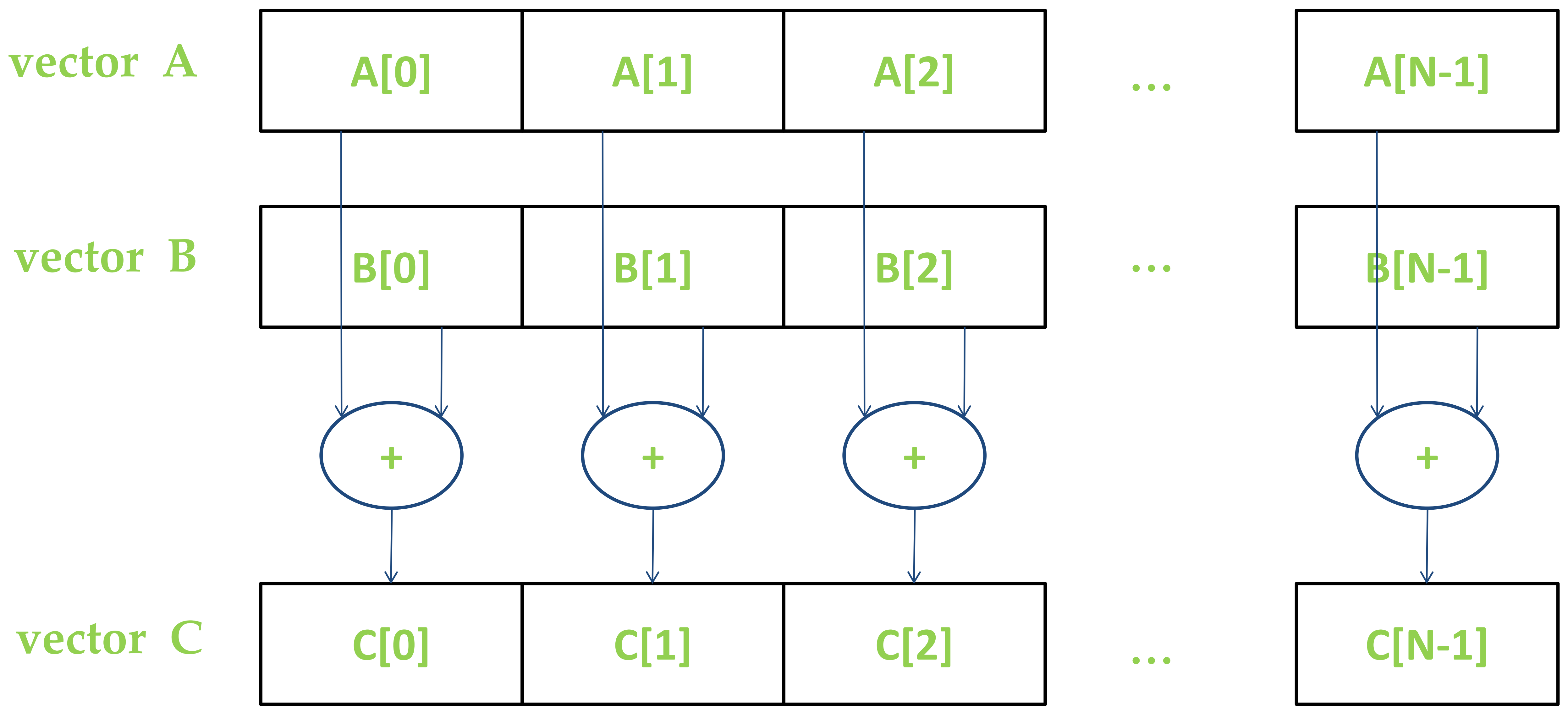
Accelerated Computing

Memory Allocation and Data Movement API Functions

Objective

- To learn the basic API functions in CUDA host code
  - Device Memory Allocation
  - Host-Device Data Transfer

Data Parallelism - Vector Addition Example



vector A A[0] A[1] A[2] … A[N-1]

vector B B[0] B[1] B[2] … B[N-1]

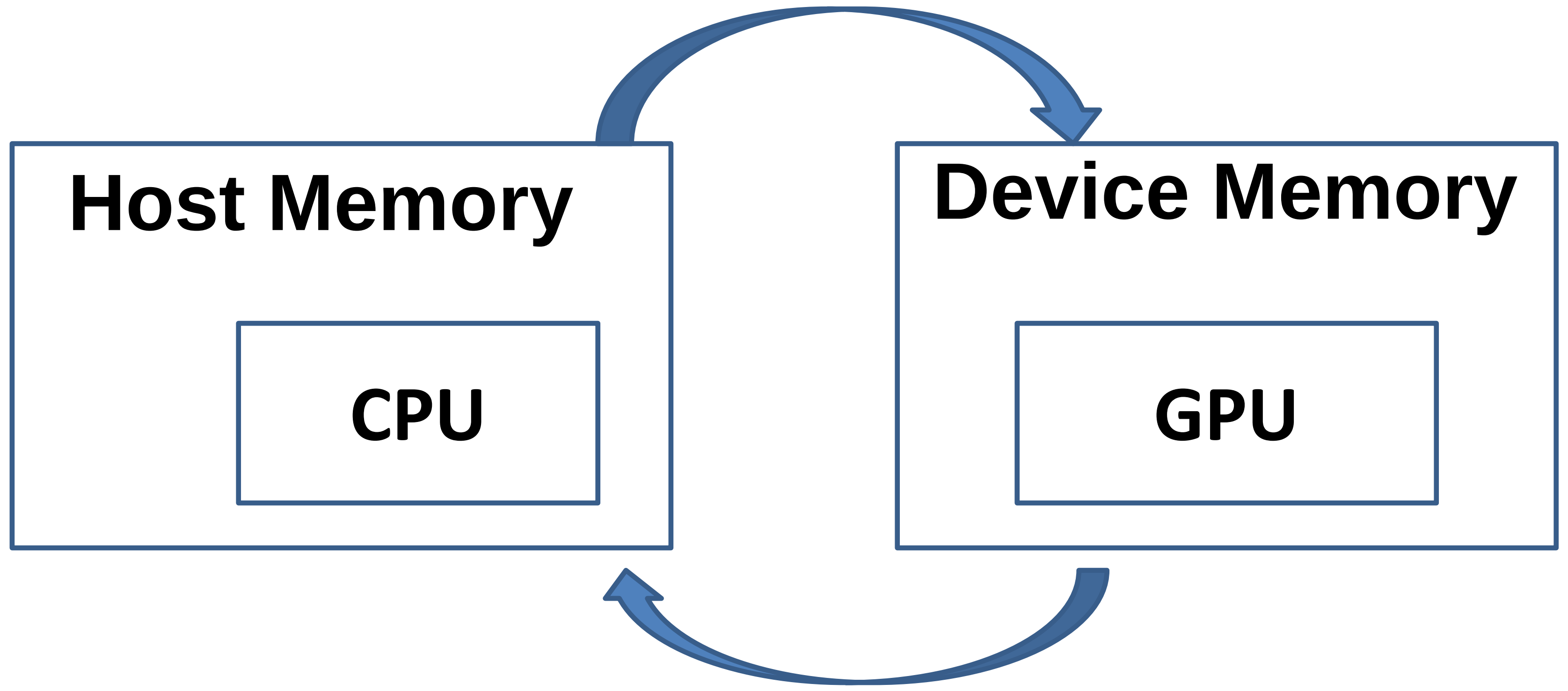
+ \+ \+ \+

vector C C[0] C[1] C[2] … C[N-1]

Vector Addition – Traditional C Code

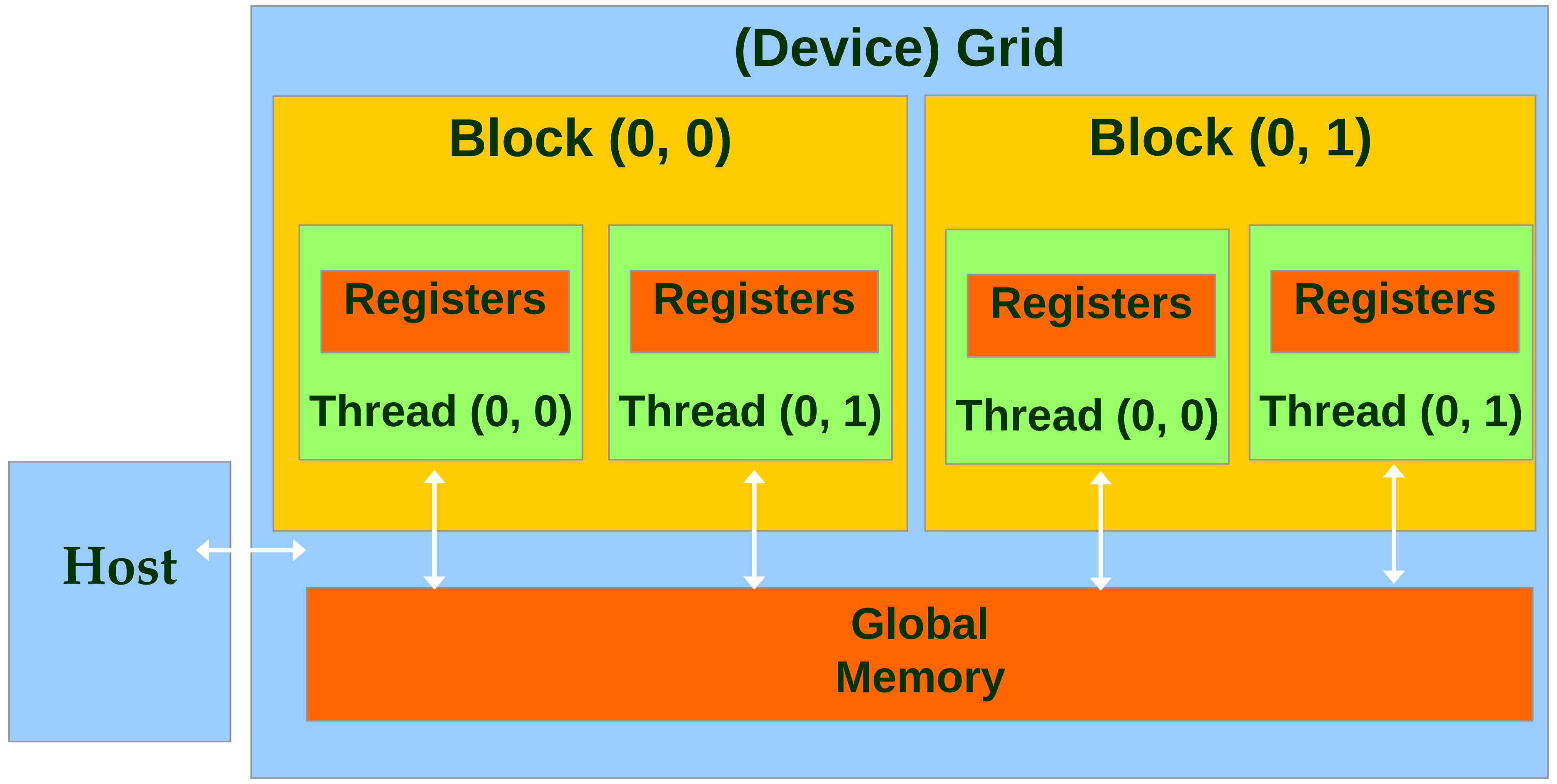
```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    …
          (h_A, h_B, h_C, N);
}
```

Heterogeneous Computing vecAdd CUDA Host Code



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n* sizeof(float);
  float *d_A, *d_B, *d_C;
  // Part 1
  // Allocate device memory for A, B, and C
  // copy A and B to device memory

  // Part 2
  // Kernel launch code – the device performs the actual vector addition

  // Part 3
  // copy C from the device memory

}
```

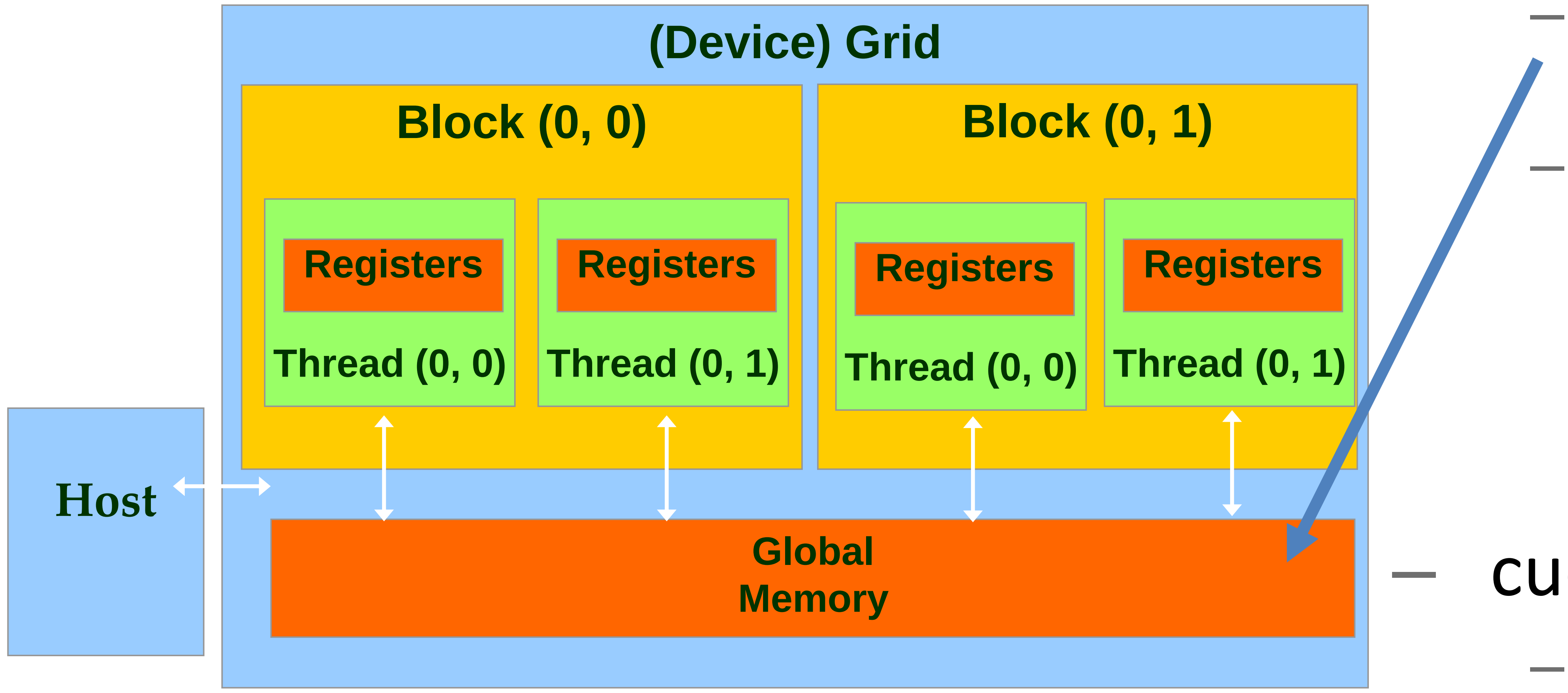
Partial Overview of CUDA Memories



- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

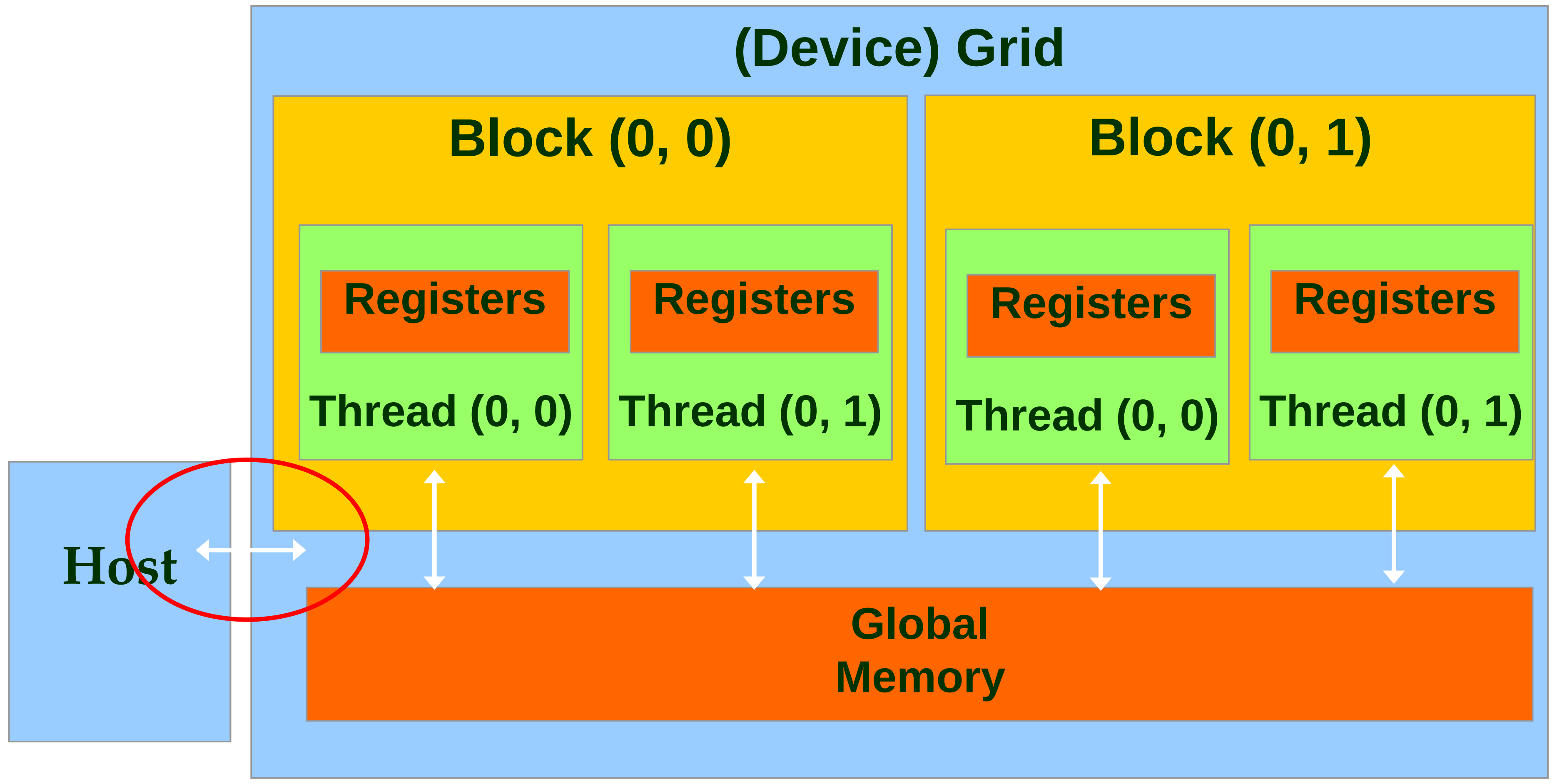
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions



- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type/Direction of transfer
  - Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
   int size = n * sizeof(float); float *d_A, *d_B, *d_C;

   cudaMalloc((void **) &d_A, size);
   cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

In Practice, Check for API Errors in Host Code

```
cudaError_t  err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess)  {
   printf(“%s in %s at line %d\n”,   cudaGetErrorString(err), __FILE__,
   __LINE__);
   exit(EXIT_FAILURE);
}
```

NVIDIA

GPU Teaching Kit

Accelerated Computing

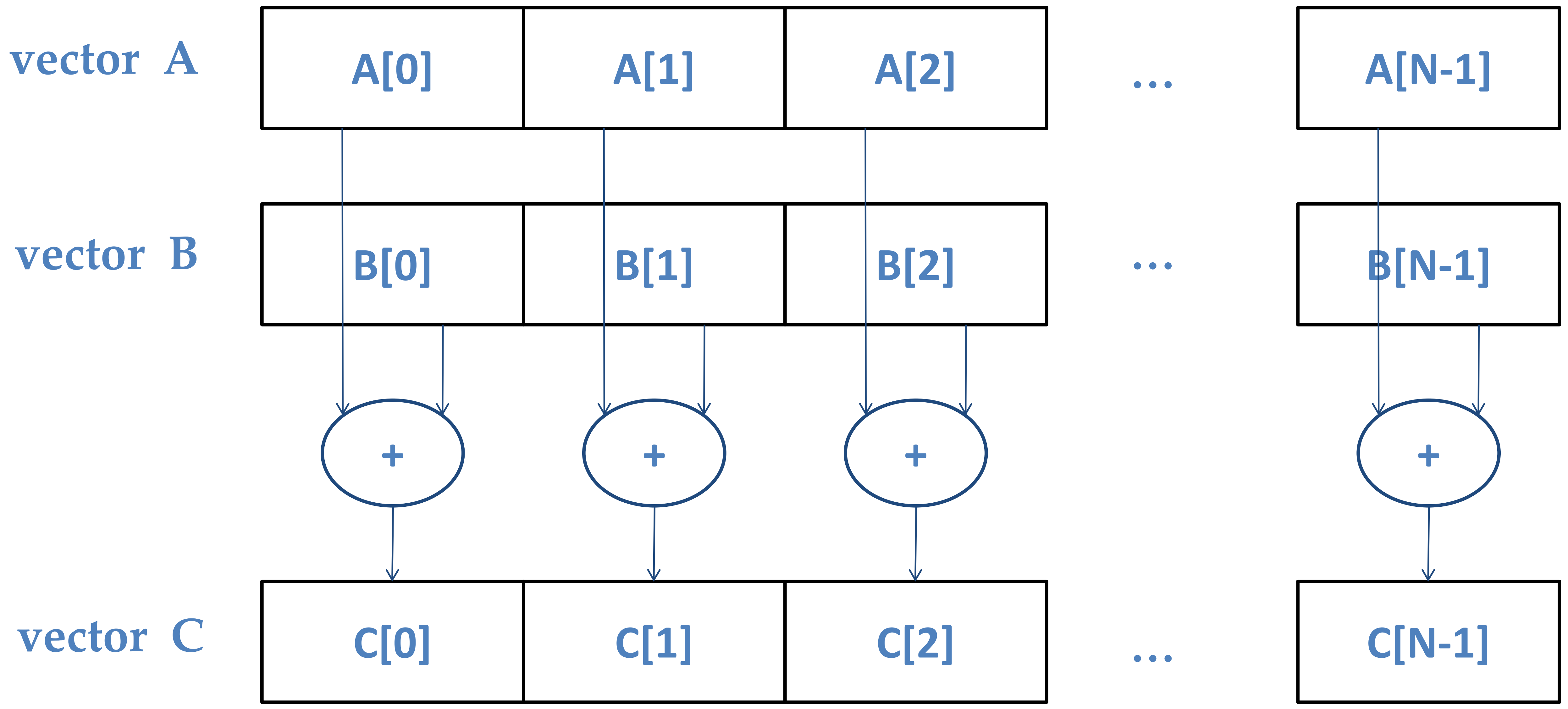
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Threads and Kernel Functions

Objective

- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
  - Hierarchical thread organization
  - Launching parallel execution
  - Thread index to data index mapping

Data Parallelism - Vector Addition Example



vector A: A[0] A[1] A[2] … A[N-1]

vector B: B[0] B[1] B[2] … B[N-1]

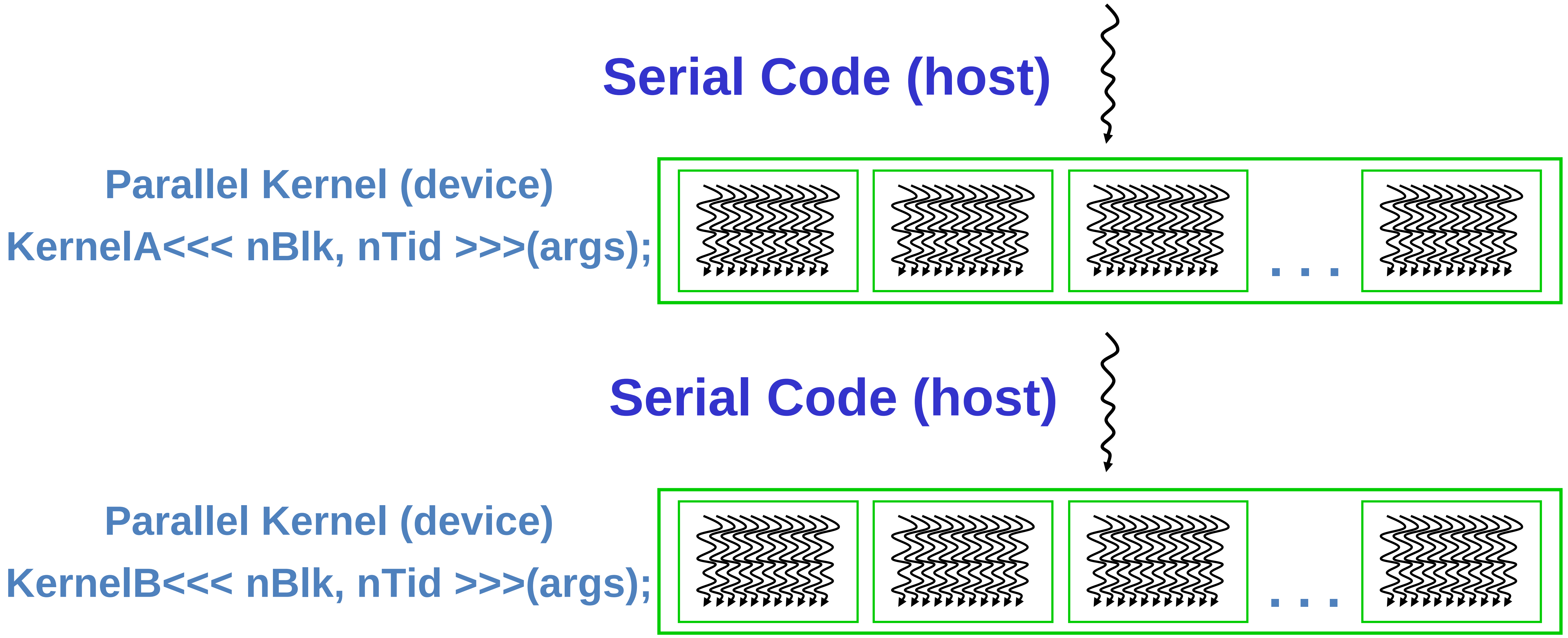
+ + + +

vector C: C[0] C[1] C[2] … C[N-1]

CUDA Execution Model

Heterogeneous host (CPU) + device (GPU) application C program

- Serial parts in **host** C code
- Parallel parts in **device** SPMD kernel code



A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted” Von-Neumann Processor

Memory

I/O

Processing Unit

ALU

Reg File

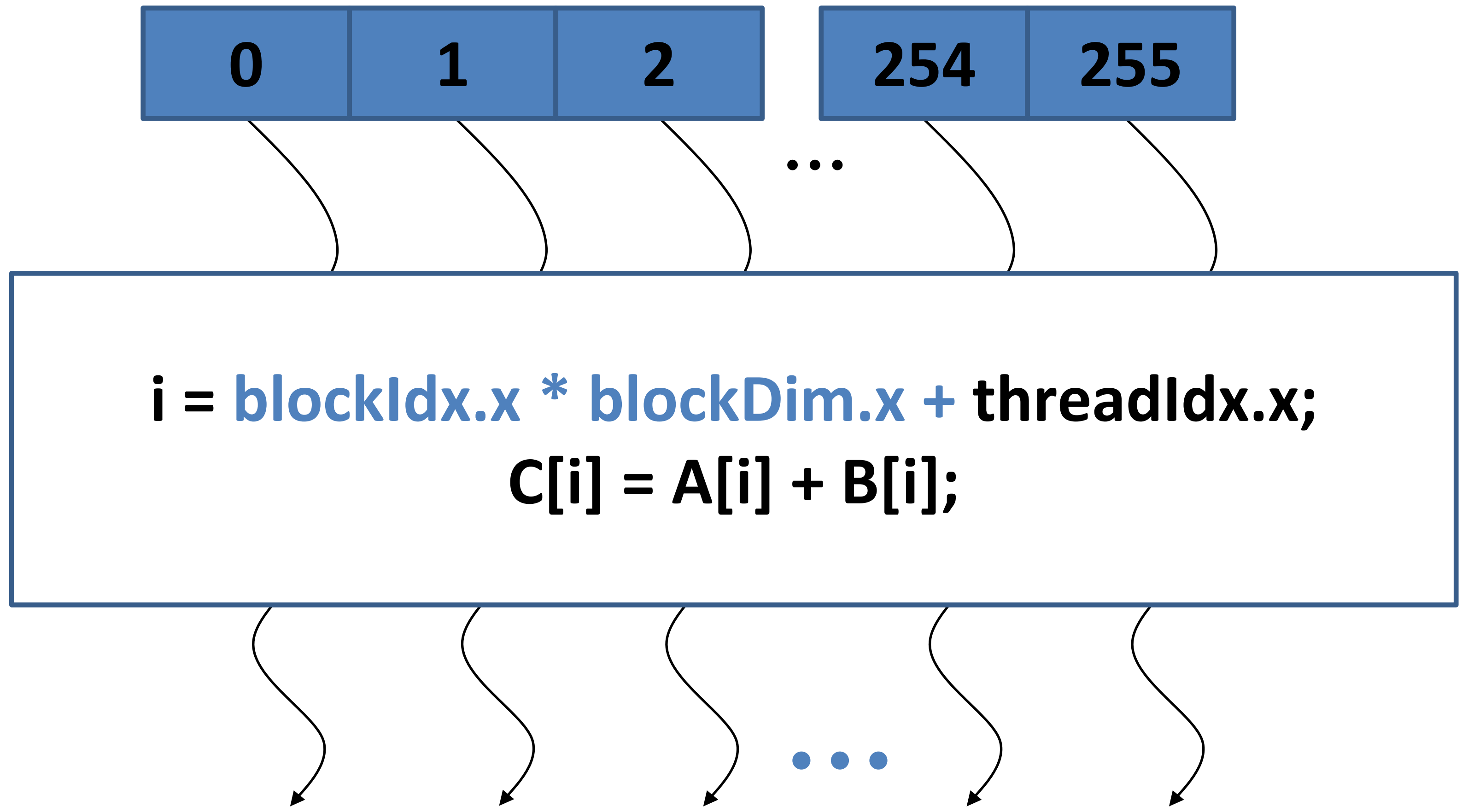
Control Unit

PC

IR

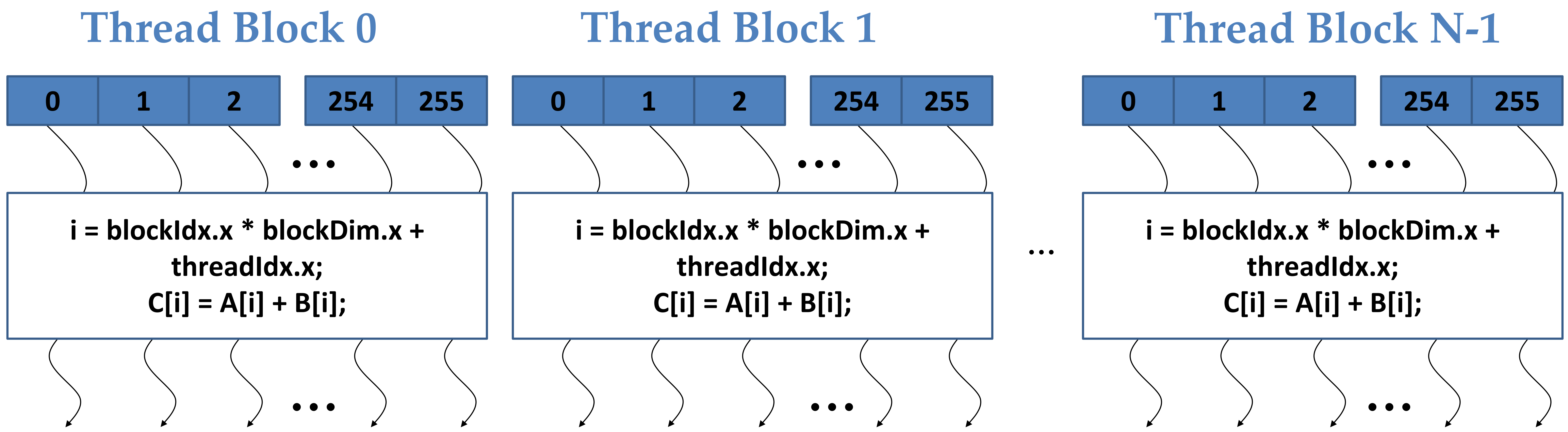
Arrays of Parallel Threads

- A CUDA kernel is executed by a grid (array) of threads
  - All threads in a grid run the same kernel code (Single Program Multiple Data)
  - Each thread has indexes that it uses to compute memory addresses and make control decisions



```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Blocks: Scalable Cooperation



Thread Block 0

0 1 2 … 254 255

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

Thread Block 1

0 1 2 … 254 255

```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

…

Thread Block N-1

0 1 2 … 254 255

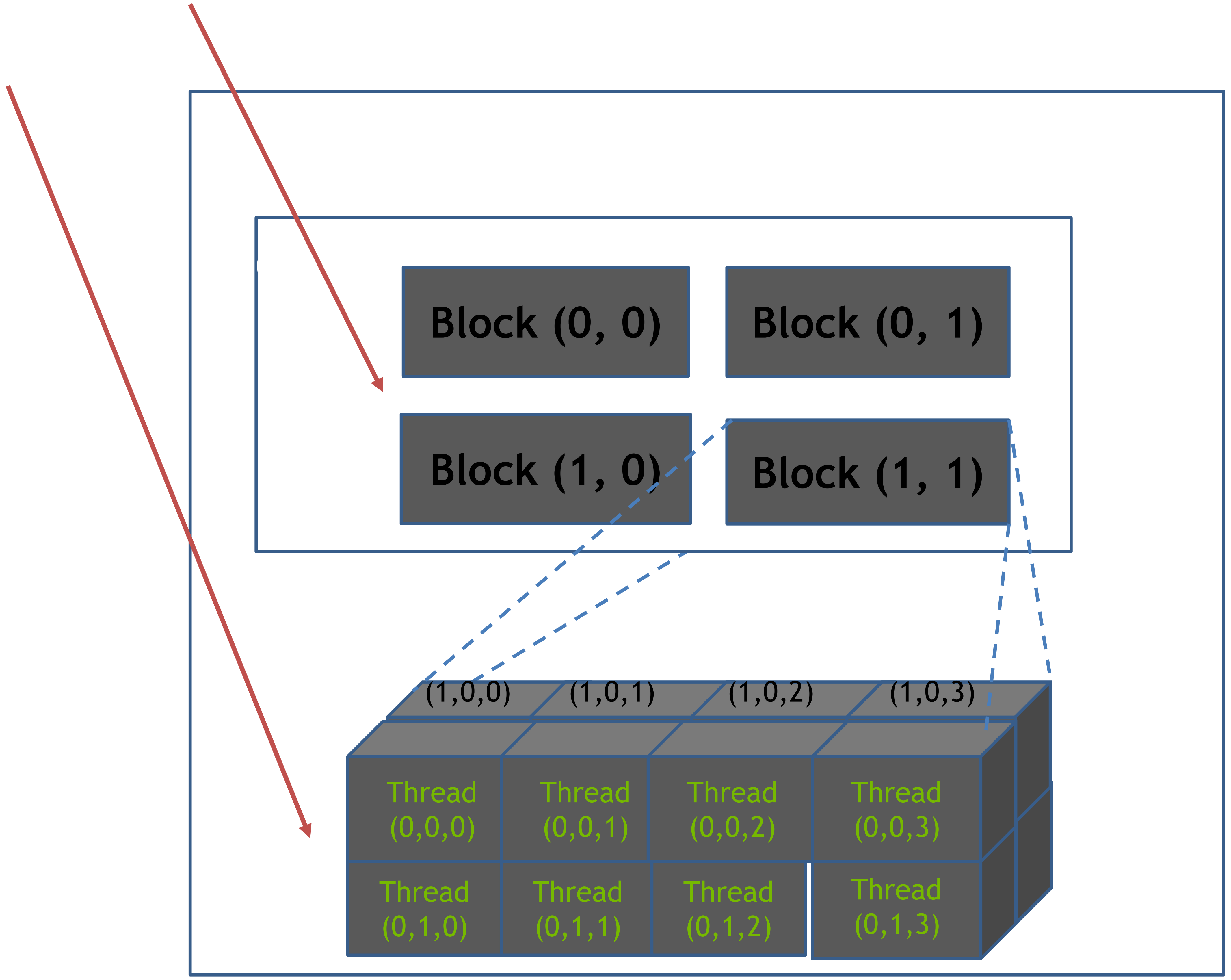
```
i = blockIdx.x * blockDim.x +
threadIdx.x;
C[i] = A[i] + B[i];
```

- Divide thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations on shared/global memory addresses** and **barrier synchronization**
  - Threads in different blocks do not interact **(except for atomic operations on global memory addresses)**

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
  - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
  - threadIdx: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



GPU Teaching Kit

Accelerated Computing

CUDA Parallelism Model

Kernel-Based SPMD Parallel Programming

Objective

- To learn the basic concepts involved in a simple CUDA kernel function
  - Declaration
  - Built-in variables
  - Thread index to data index mapping

Example: Vector Addition Kernel

Device Code

```
//  Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x+blockDim.x*blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
 // d_A, d_B, d_C allocations and copies omitted
 // Run ceil(n/256.0) blocks of 256 threads each
  vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C, n);
}
```

The ceiling function makes sure that there are enough threads to cover all elements.

More on Kernel Launch (Host Code)

Host Code

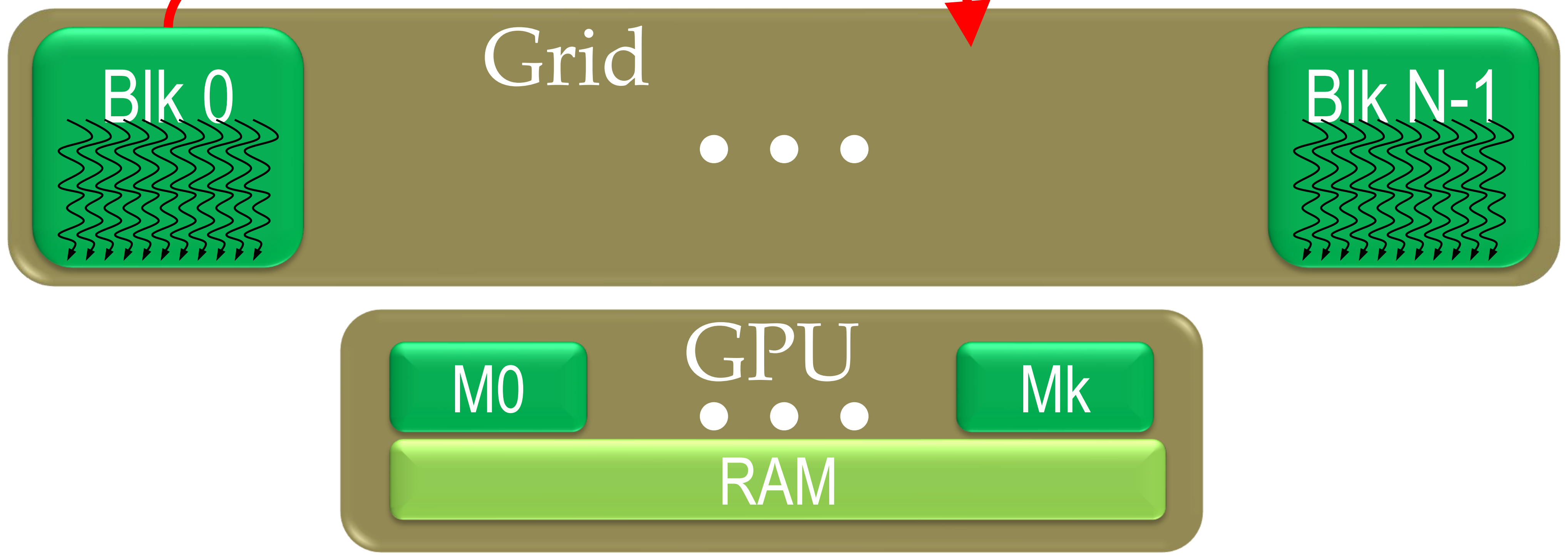
```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
  dim3 DimGrid((n-1)/256 + 1, 1, 1);
  dim3 DimBlock(256, 1, 1);
  vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C, n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell

```
__host__
void vecAdd(…)
{
  dim3 DimGrid(ceil(n/256.0),1,1);
  dim3 DimBlock(256,1,1);
vecAddKernel<<<DimGrid,DimBlock>>>(d_A,d_B,
d_C,n);
}
```

```
__global__
void vecAddKernel(float *A,
      float *B, float *C, int n)
{
   int i = blockIdx.x * blockDim.x
         + threadIdx.x;

   if( i<n ) C[i] = A[i]+B[i];
}
```

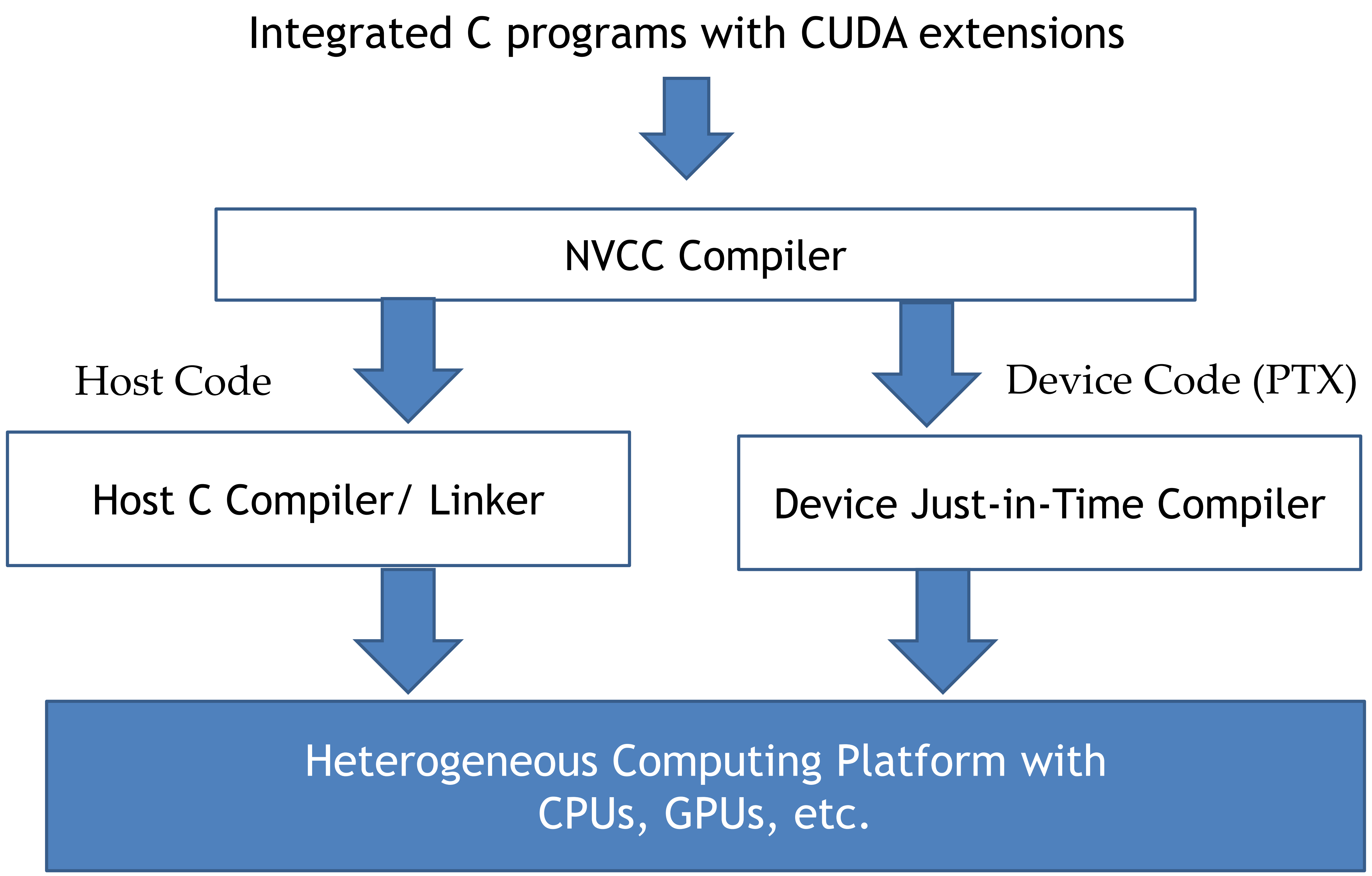


More on CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

Compiling A CUDA Program



Integrated C programs with CUDA extensions

NVCC Compiler

Host Code

Device Code (PTX)

Host C Compiler/ Linker

Device Just-in-Time Compiler

Heterogeneous Computing Platform with
CPUs, GPUs, etc.

NVIDIA

GPU Teaching Kit

Accelerated Computing

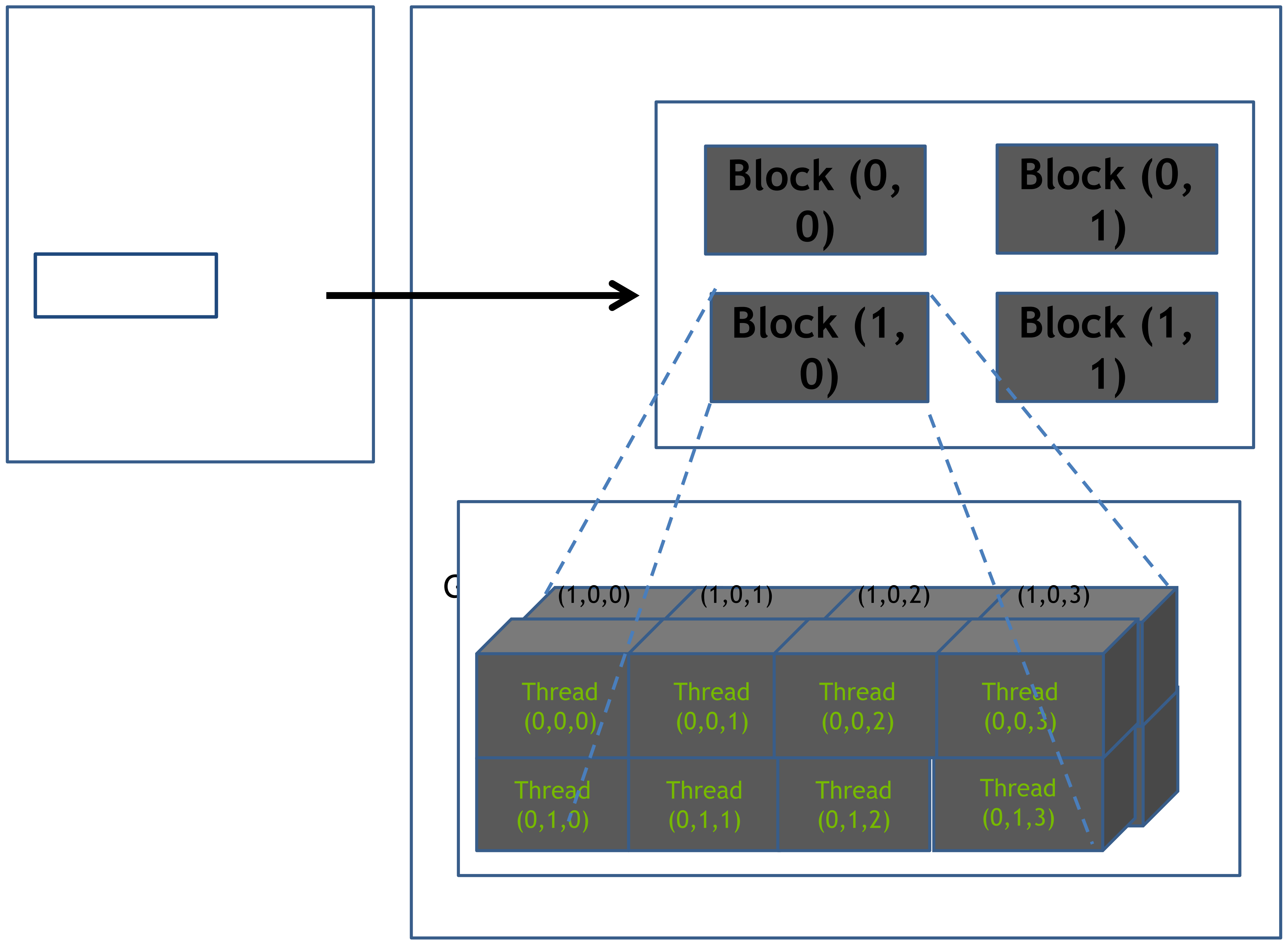
ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Multidimensional Kernel Configuration

Objective

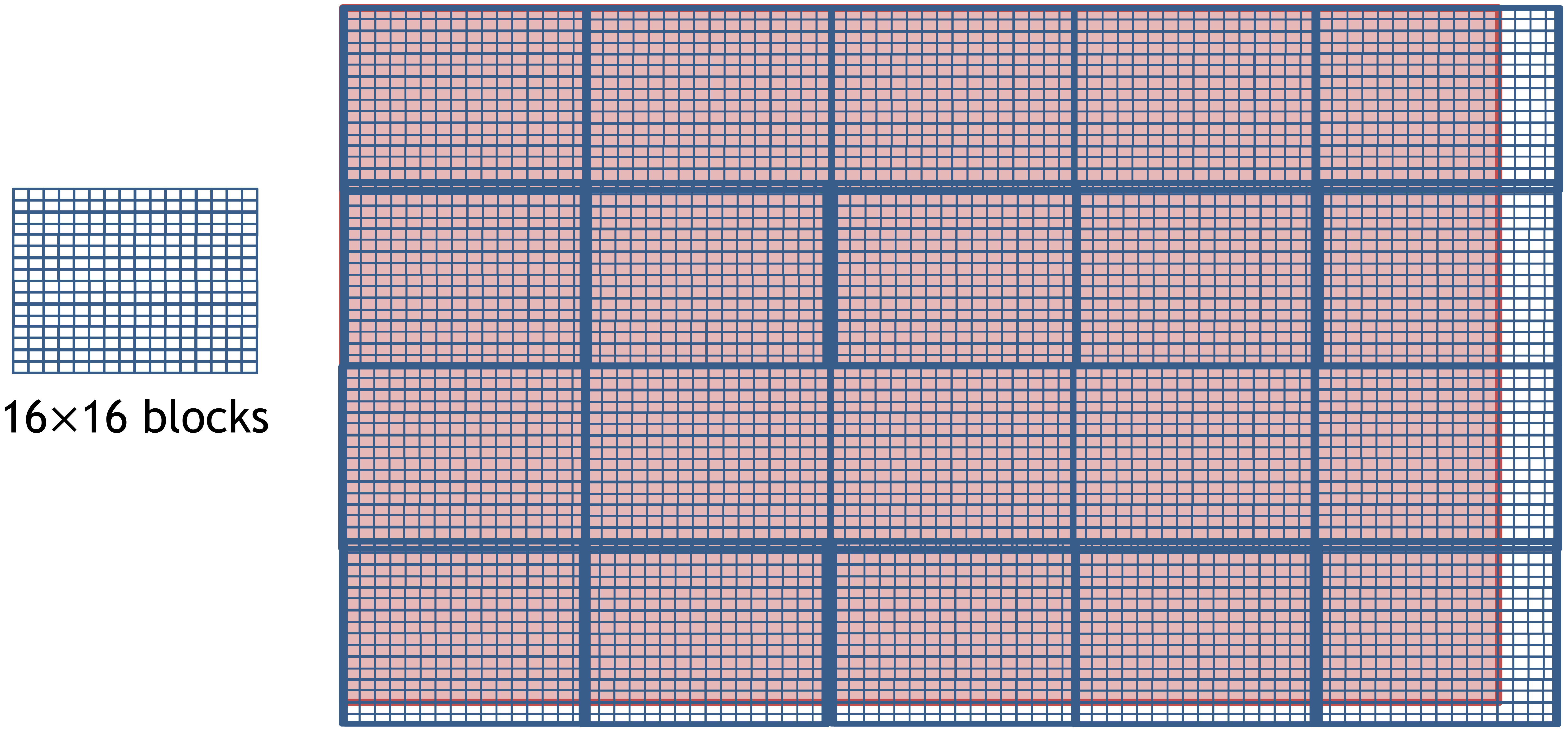
- To understand multidimensional Grids
  - Multi-dimensional block and thread indices
  - Mapping block/thread indices to data indices

A Multi-Dimensional Grid Example



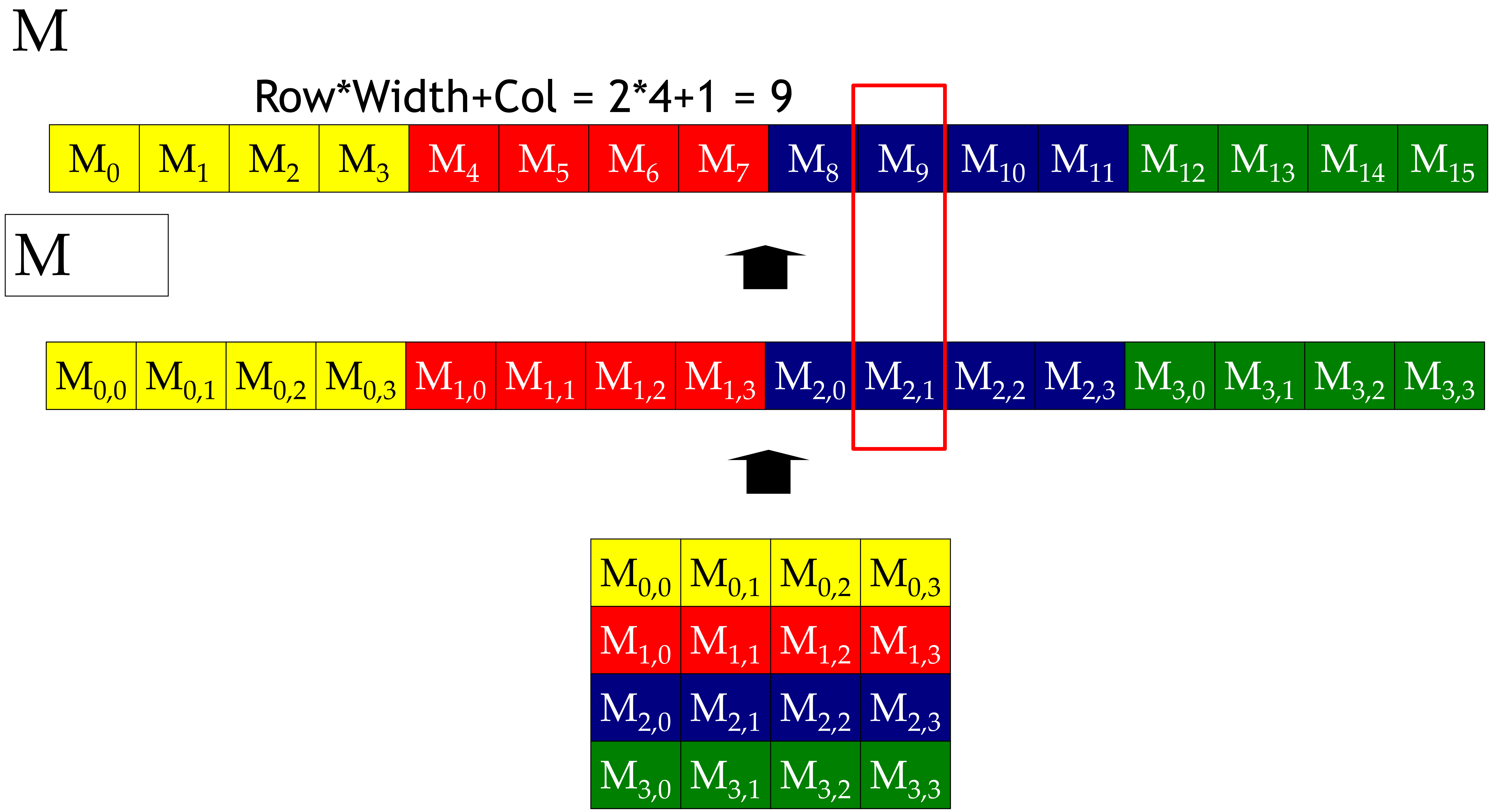
Note: Block index: (y, x), Thread index: (z, y, x)

Processing a Picture with a 2D Grid



62×76 picture

Row-Major Layout in C/C++



M

Row*Width+Col = 2*4+1 = 9

M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}

M

$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Source Code of a PictureKernel

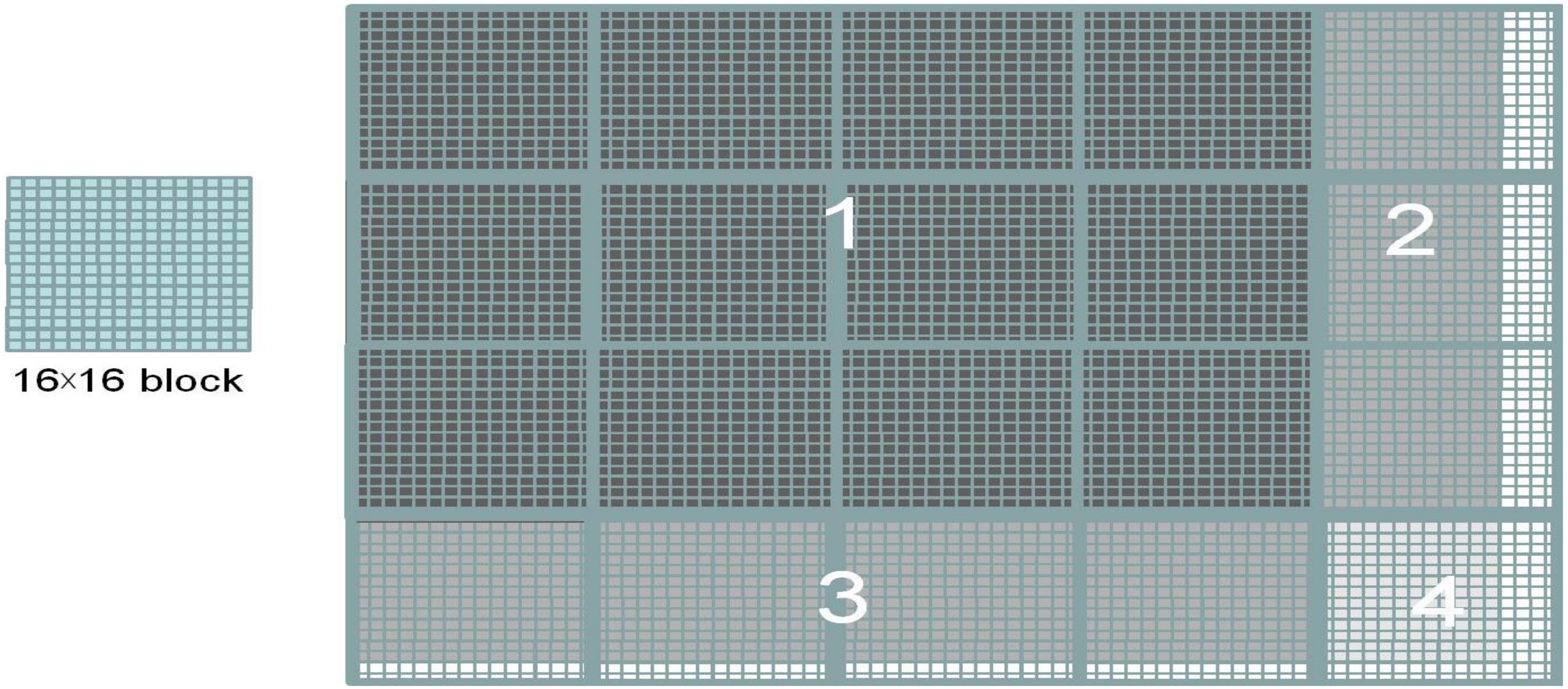
```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                                  int height, int width)
{

  // Calculate the row # of the d_Pin and d_Pout element
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < height) && (Col < width)) {
    d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
  }
}
```

Scale every pixel value by 2.0

Host Code for Launching PictureKernel

```
// assume that the picture is m      n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device
// output d_Pout has been allocated on device
…
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
…
```

Covering a 62×76 Picture with 16×16 Blocks



Not all threads in a Block will follow the same control flow path.

CSC 2224: Parallel Computer Architecture and Programming GPU Architecture: Introduction

Prof. Gennady Pekhimenko

University of Toronto

Fall 2020

The content of this lecture is adapted from the slides of Kayvon Fatahalian (Stanford), Olivier Giroux and Luke Durant (Nvidia), Tor Aamodt (UBC) and Edited by: Serina Tan