

CS-206 Concurrency

Lecture 11

Data Parallel Computing

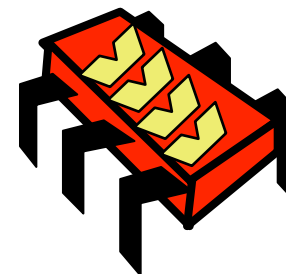
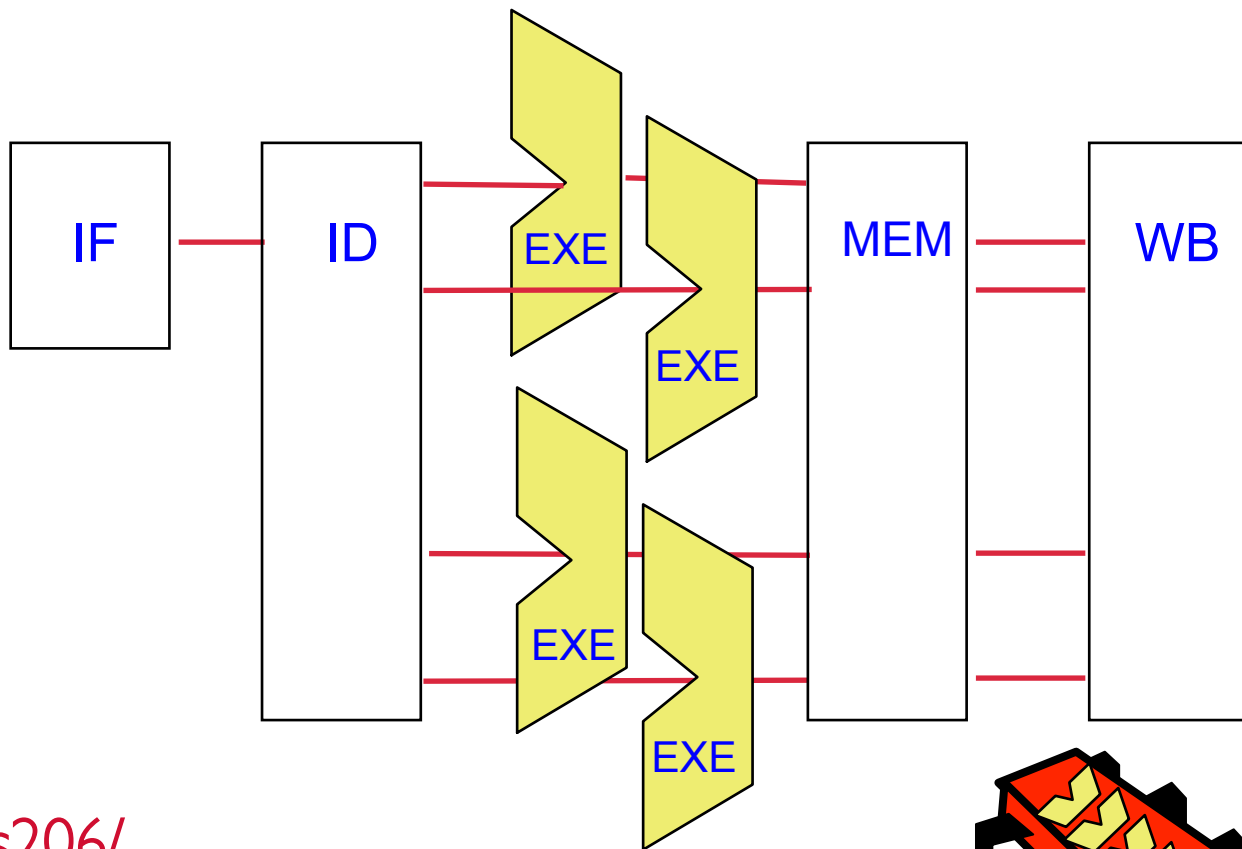
Spring 2015

Prof. Babak Falsafi

parsa.epfl.ch/courses/cs206/

Adapted from slides originally developed by Andreas Di Blas, Babak Falsafi, Simon Green, David Kirk, Andreas Moshovos, David Patterson and Waqar Saleem

EPFL Copyright 2015

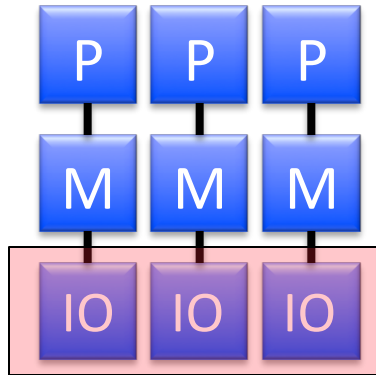


Where are We?

		Lecture & Lab			
M	T	W	T	F	
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb	
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb	
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar	
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar	
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar	
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar	
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr	
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr	
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr	
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr	
27-Apr	28-Apr	29-Apr	30-Apr	1-May	
4-May	5-May	6-May	7-May	8-May	
11-May	12-May	13-May	14-May	15-May	
18-May	19-May	20-May	21-May	22-May	
25-May	26-May	27-May	28-May	29-May	

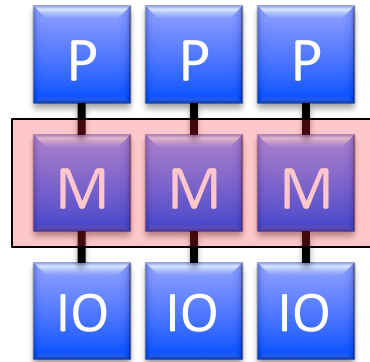
- ▶ Data Parallel Computing
 - ▷ Vector
 - ▷ GPU
- ▶ GPU architecture
- ▶ CUDA
- ▶ Next week
 - ▷ More CUDA

Recall: Historical View



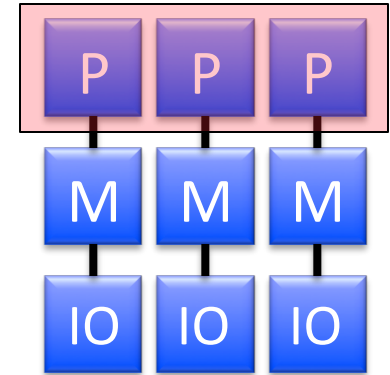
Join at: I/O (Network)

Program with: Message passing
Hadoop
SQL (databases)



Memory

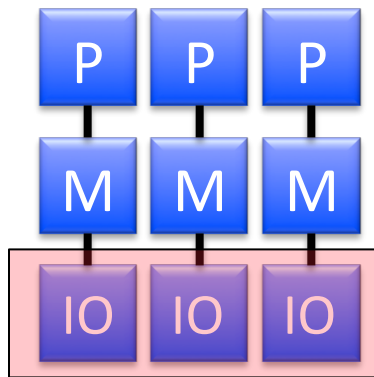
Shared Memory
Java threads
Posix threads



Processor

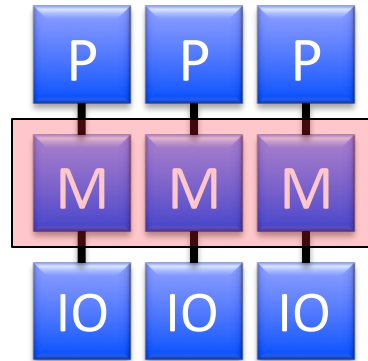
Data Parallel,
SIMD, Vector,
GPU, MapReduce

From now on: Data Parallel



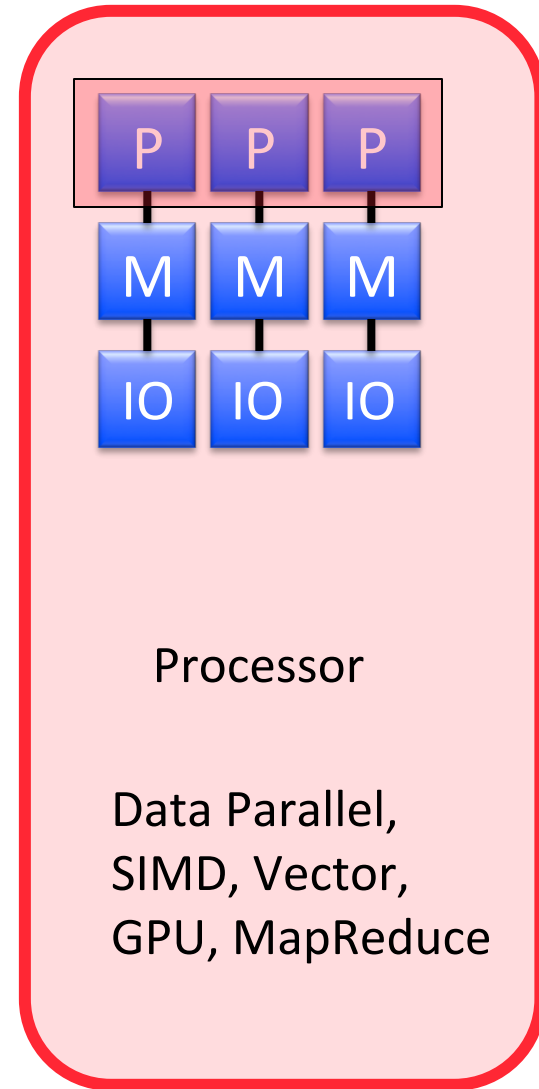
Join at: I/O (Network)

Program with: Message passing
Hadoop
SQL (databases)



Memory

Shared Memory
Java threads
Posix threads



Processor

Data Parallel,
SIMD, Vector,
GPU, MapReduce

Recall: Forms of Parallelism

▶ Throughput parallelism

- ▷ Perform many (identical) sequential tasks at the same time
- ▷ E.g., Google search, ATM (bank) transactions

▶ Task parallelism

- ▷ Perform tasks that are functionally different in parallel
- ▷ E.g., iPhoto (face recognition with slide show)

▶ Pipeline parallelism

- ▷ Perform tasks that are different in a particular order
- ▷ E.g., speech (signal, phonemes, words, conversation)

▶ Data parallelism

- ▷ Perform the same task on different data
- ▷ E.g., Graphics, data analytics



Reduce time for one job

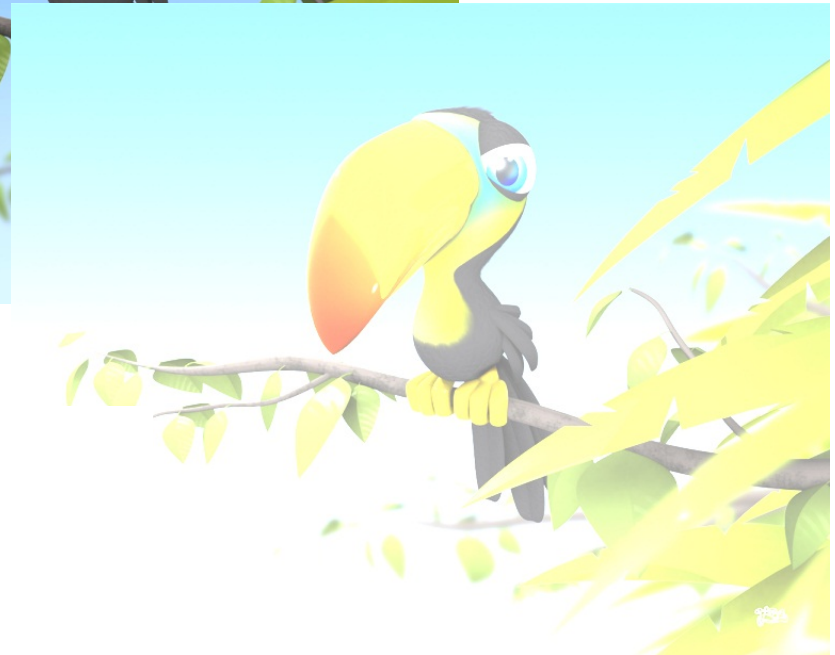
Recall: Forms of Parallelism

- ▶ Throughput parallelism
 - ▷ Perform many (identical) sequential tasks at the same time
 - ▷ E.g., Google search, ATM (bank) transactions
- ▶ Task parallelism
 - ▷ Perform tasks that are functionally different in parallel
 - ▷ E.g., iPhoto (face recognition with slide show)
- ▶ Pipeline parallelism
 - ▷ Perform tasks that are different in a particular order
 - ▷ E.g., speech (signal, phonemes, words, conversation)
- ▶ **Data parallelism**
 - ▷ Perform the same task on different data
 - ▷ E.g., Graphics, data analytics



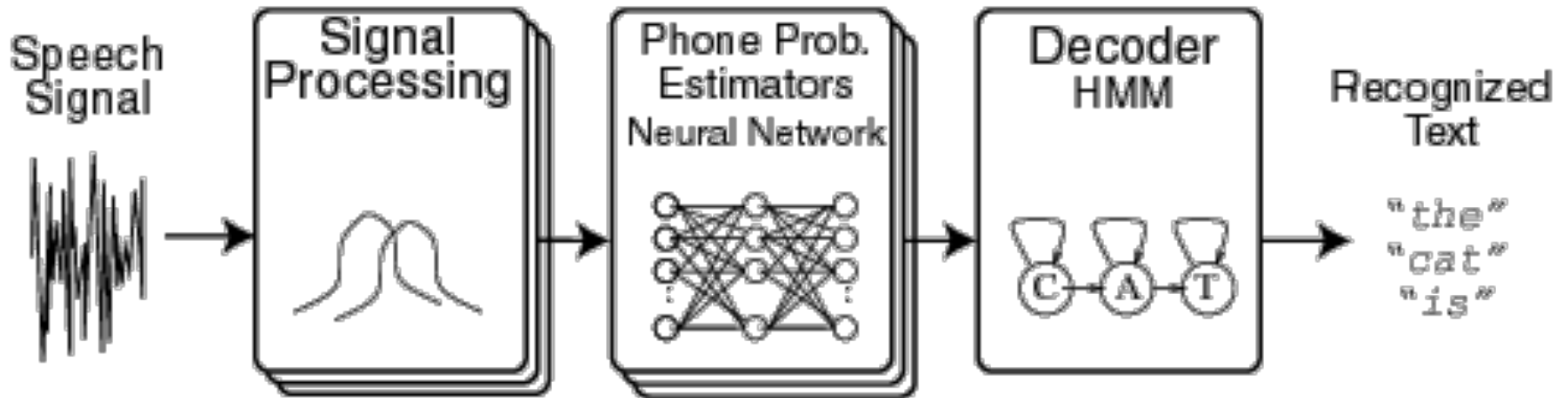
Reduce time for one job

Example: Image Processing/Graphics



```
int a[N]; // N is large
for (i =0; i < N; i++)
    a[i] = a[i] * fade;
```

Example: Speech Recognition (e.g., Siri)



- ▶ Signal processing: same algorithm run on a sample
- ▶ Neural network: propagate values across neurons



Signal Processing: Data Parallel Transforms

sampled signal (a vector)

transform (a matrix)

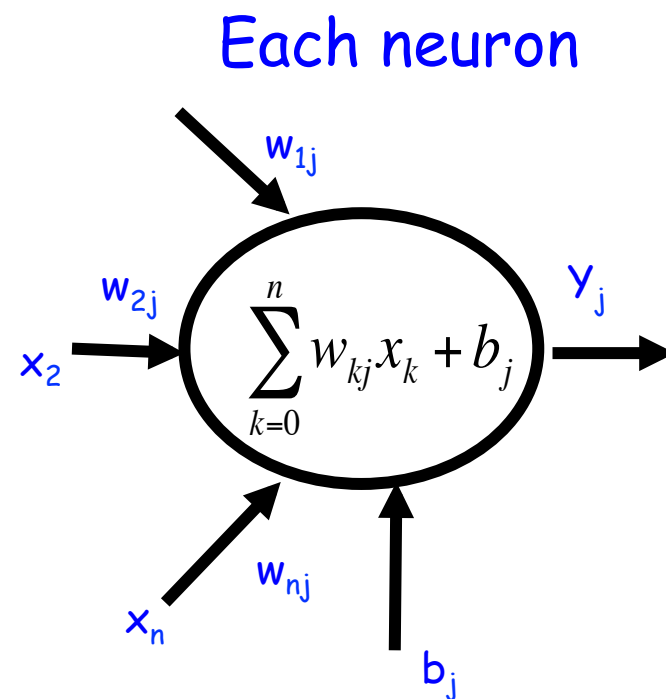
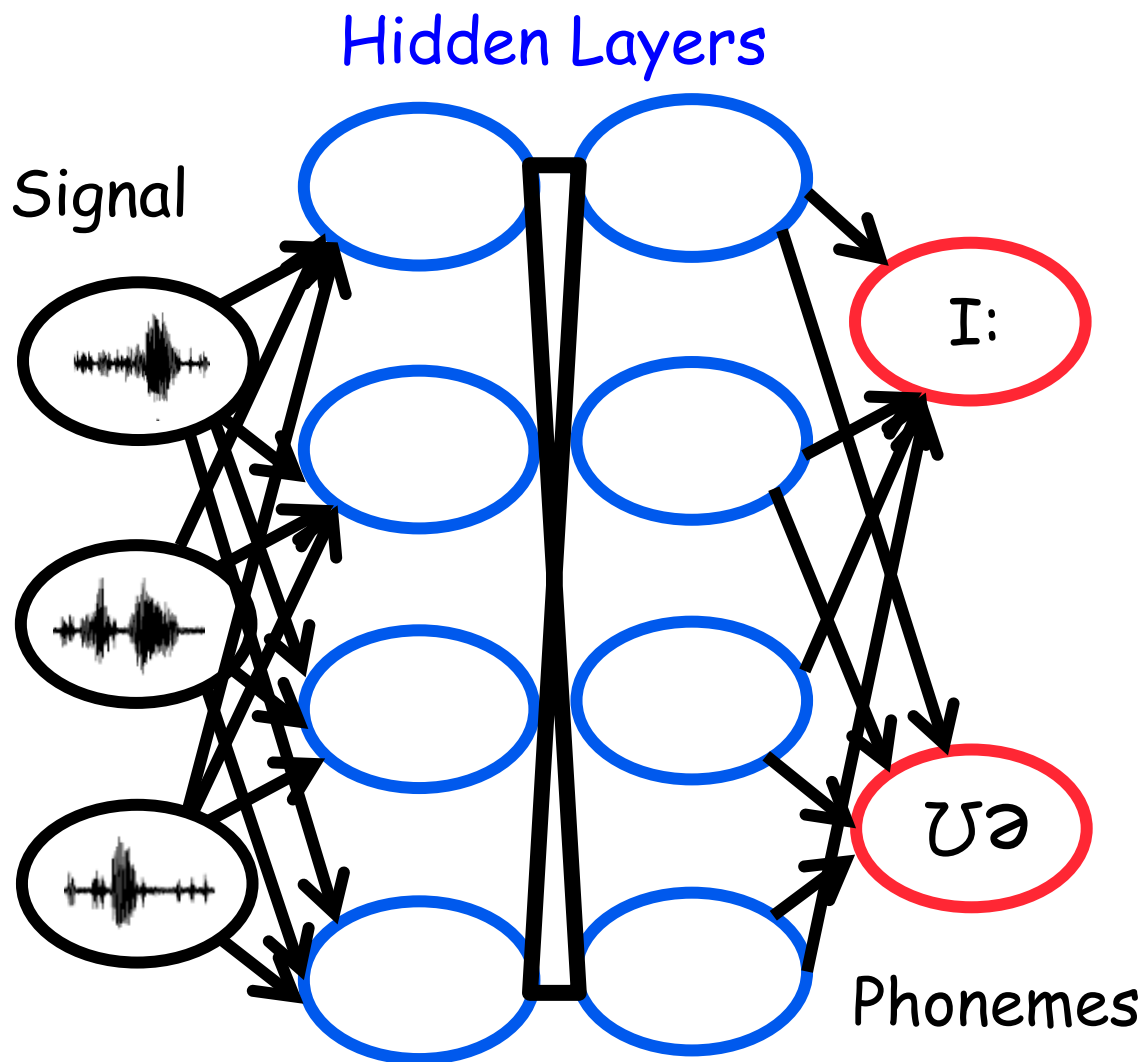
$$x \mapsto Mx$$

Example: Discrete Fourier Transform (DFT) size 4

$$DFT_4 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -1 & \\ & & & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & i \end{bmatrix} \begin{bmatrix} 1 & 1 & & \\ & 1 & -1 & \\ & & 1 & 1 \\ & & & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

Matrix operations are embarrassingly data parallel!

A network of neurons



Data Parallel Computation on Neurons

```
float nron [N]; // for large N for (each neu[i])
```

```
for (i=0; i < N; i++)
```

```
    for (j=0; j < nron[i].outputs; j++)
```

```
        nron[i].y[j] =
```

```
sigmoid(  $\sum_{k=0}^{nron[i].inputs} nron[i].w_{kj} nron[i].x_k + nron[i].b_j$  )
```

Example: Data Analytics

- ▶ Google processes 20 PB a day
- ▶ Wayback Machine has 3 PB + 100 TB/month
- ▶ Facebook has 2.5 PB of user data + 15 TB/day
- ▶ eBay has 6.5 PB of user data + 50 TB/day
- ▶ CERN's Large Hydron Collider generates 15 PB a year

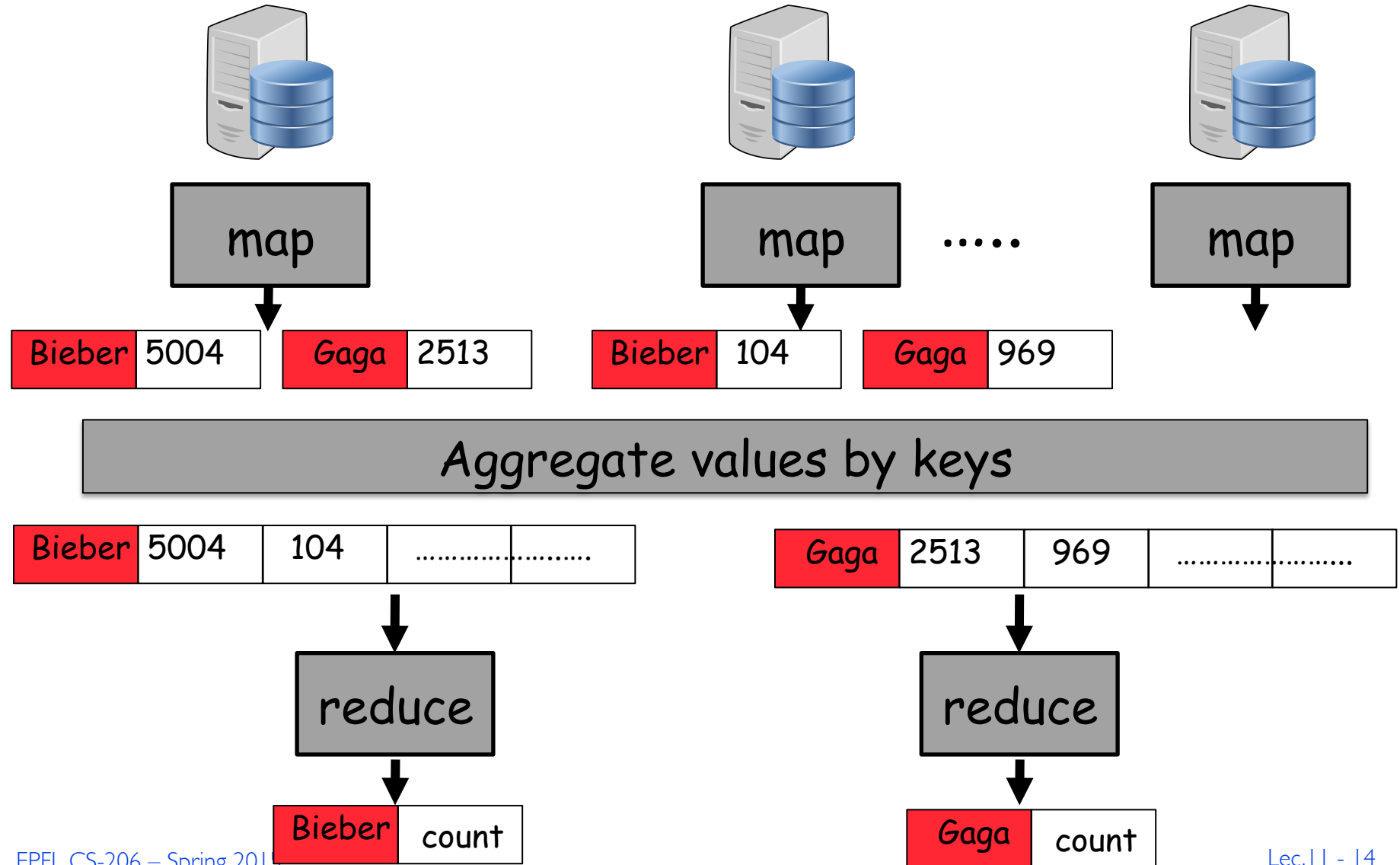
How do we aggregate this data?

MapReduce in Data Analytics

- ▶ It's about aggregating statistics over data
- ▶ Divide up the data among servers
- ▶ Compute the stats (independently)
- ▶ Then aggregate/reduce

- ▶ Example: CloudSuite classification benchmark
 - ▷ 10's of GB of web pages
 - ▷ Rank pages based on the word occurrence (popularity)
 - ▷ Look for celebrities
 - ▷ It's an embarrassingly (data) parallel problem!

MapReduce from Google: Data Parallel Computing on Volume Servers



This Course:

Data Parallel Processor Architecture

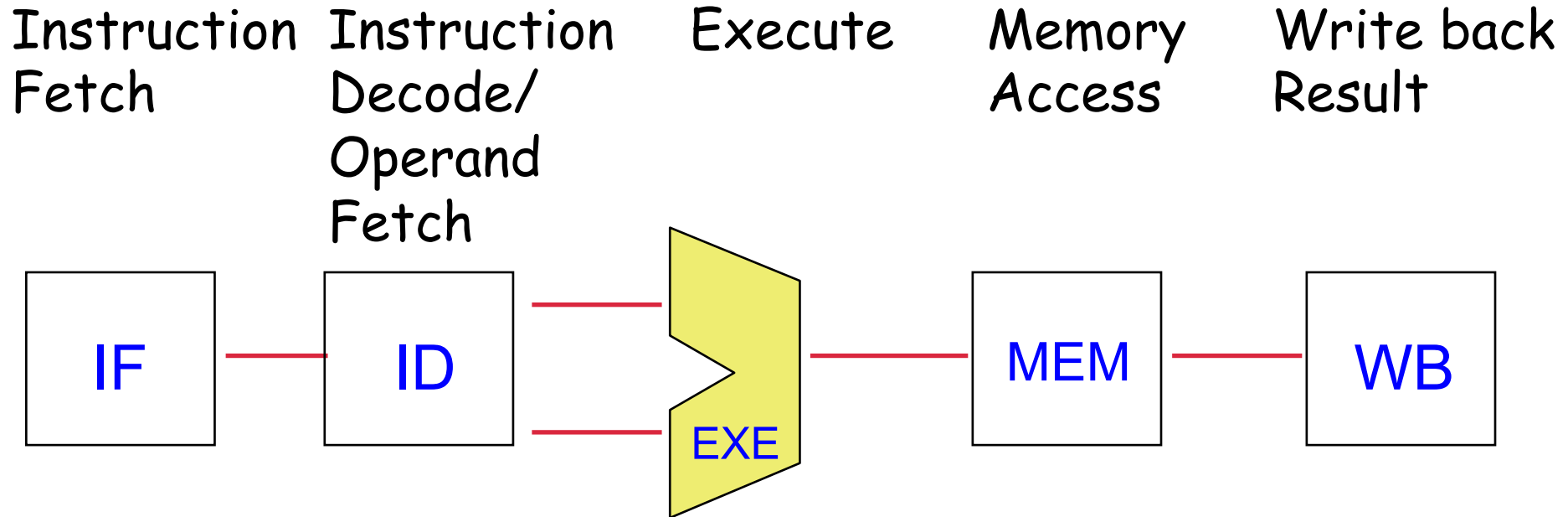
1. Vector Processors

- ▷ Pipelined execution
- ▷ SIMD: Single instruction, multiple data
- ▷ Example: modern ISA extensions

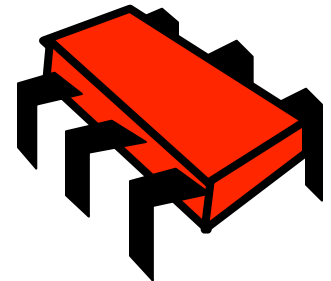
2. Graphics Processing Units (GPUs)

- ▷ Dense grid of ALUs
- ▷ SIMT: Single instruction, multiple threads
- ▷ Integrated vs. discrete

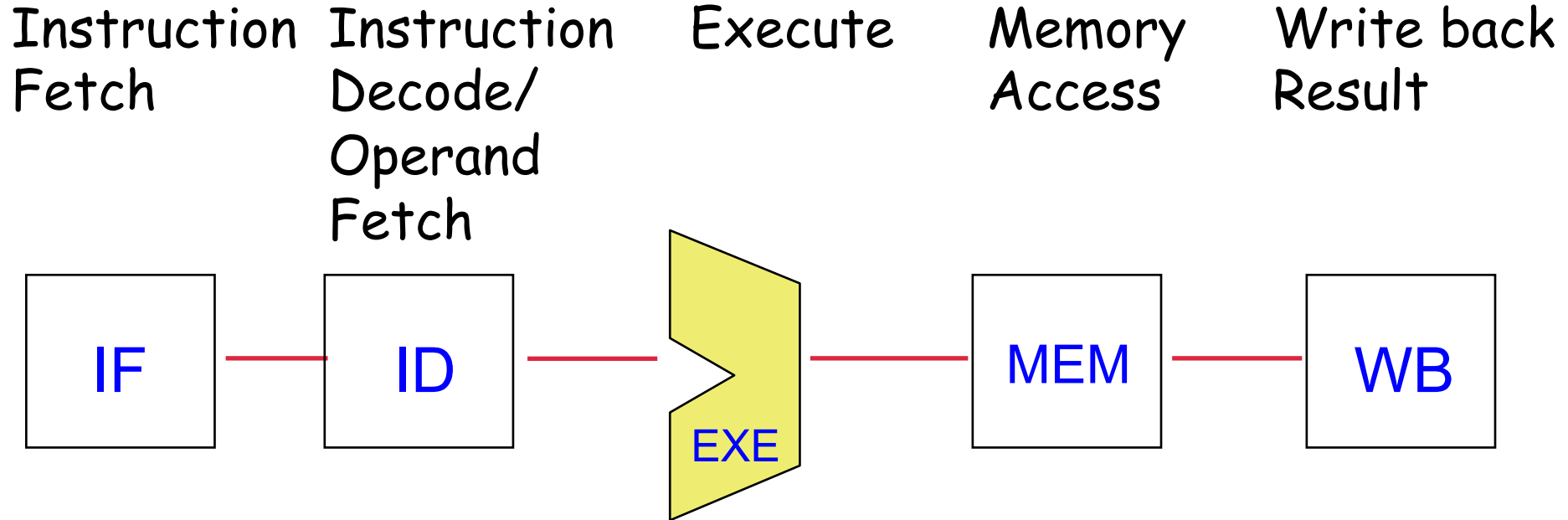
Recall: MIPS Processor (Instruction Cycle)



- ▶ Instructions are fetched from instruction cache and decoded
- ▶ Operands are fetched from register file
- ▶ Execute is the ALU (arithmetic logic unit)
- ▶ Memory access to data cache
- ▶ Write results back to register file



Recall: MIPS Pipeline (Instruction Cycle)



```
int a[N]; // N is large
for (i =0; i < N; i++)
    a[i] = a[i] * fade;
```



Fader loop in assembly

```
for (i =0; i < N; i++)  
    a[i] = a[i] * fade;
```

- ▶ The loop iterates N times (once for each array element)
- ▶ Same exact operation for each element
- ▶ Assume 32-bit “mul”

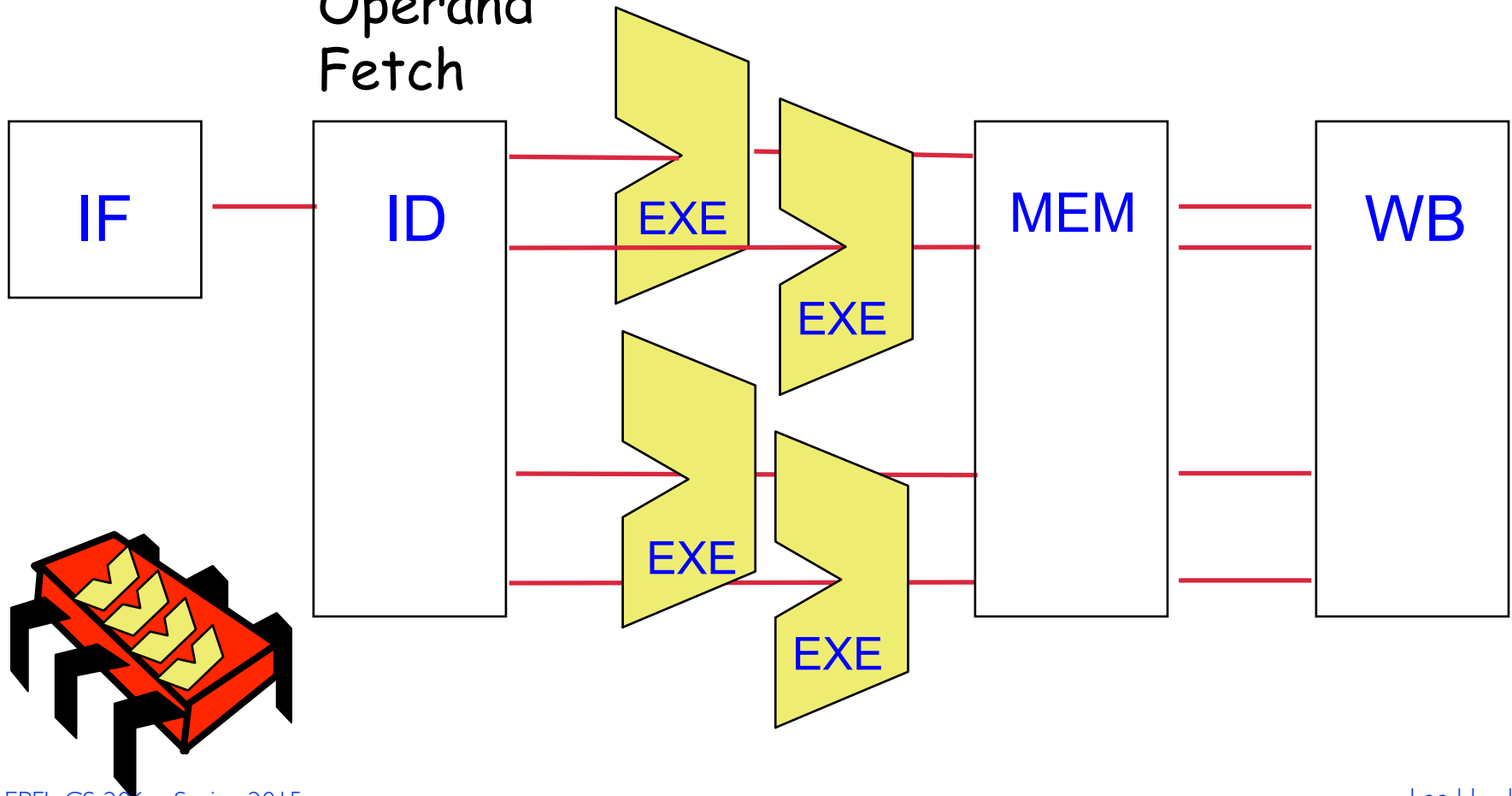
```
; a[] -> $2,  
; fade -> $3,  
; &a[N] -> $4,  
; $5 is a temp
```

loop:

```
lw    $5, 0($2)  
mul   $5, $3, $5  
sw    $5, 0($2)  
addi  $2, $2, 4  
bne   $2, $4, loop
```

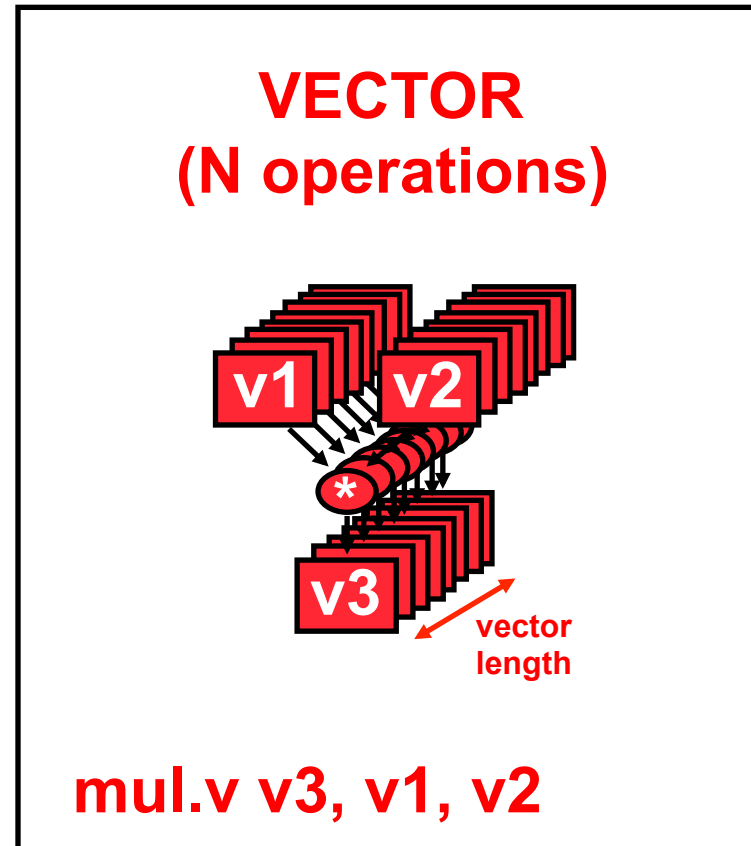
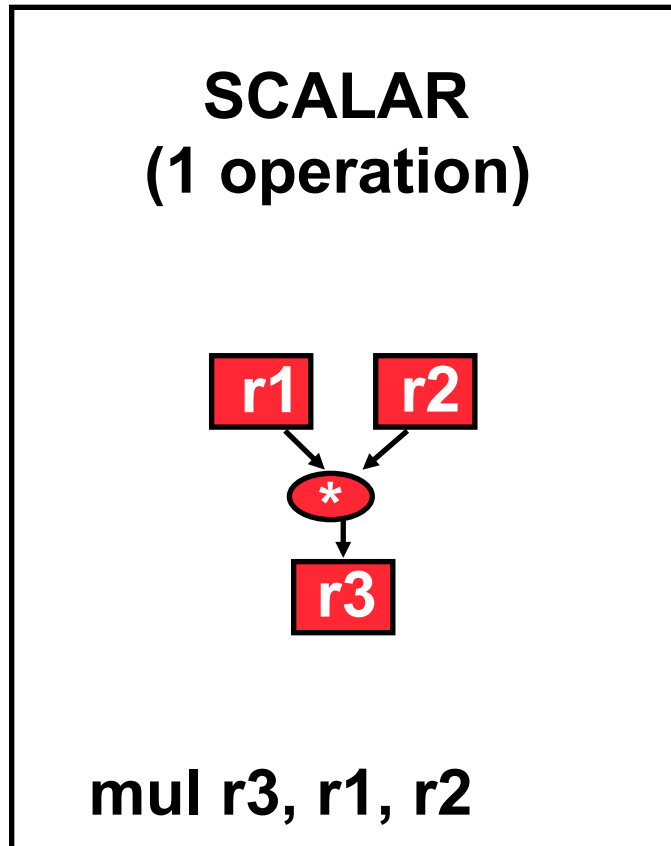
Vector Processor: One instruction, multiple data

Instruction Fetch Instruction Decode/
Operand Fetch Execute Memory Access Write back Result



Vector Processing

- ▶ Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



Example vector instructions

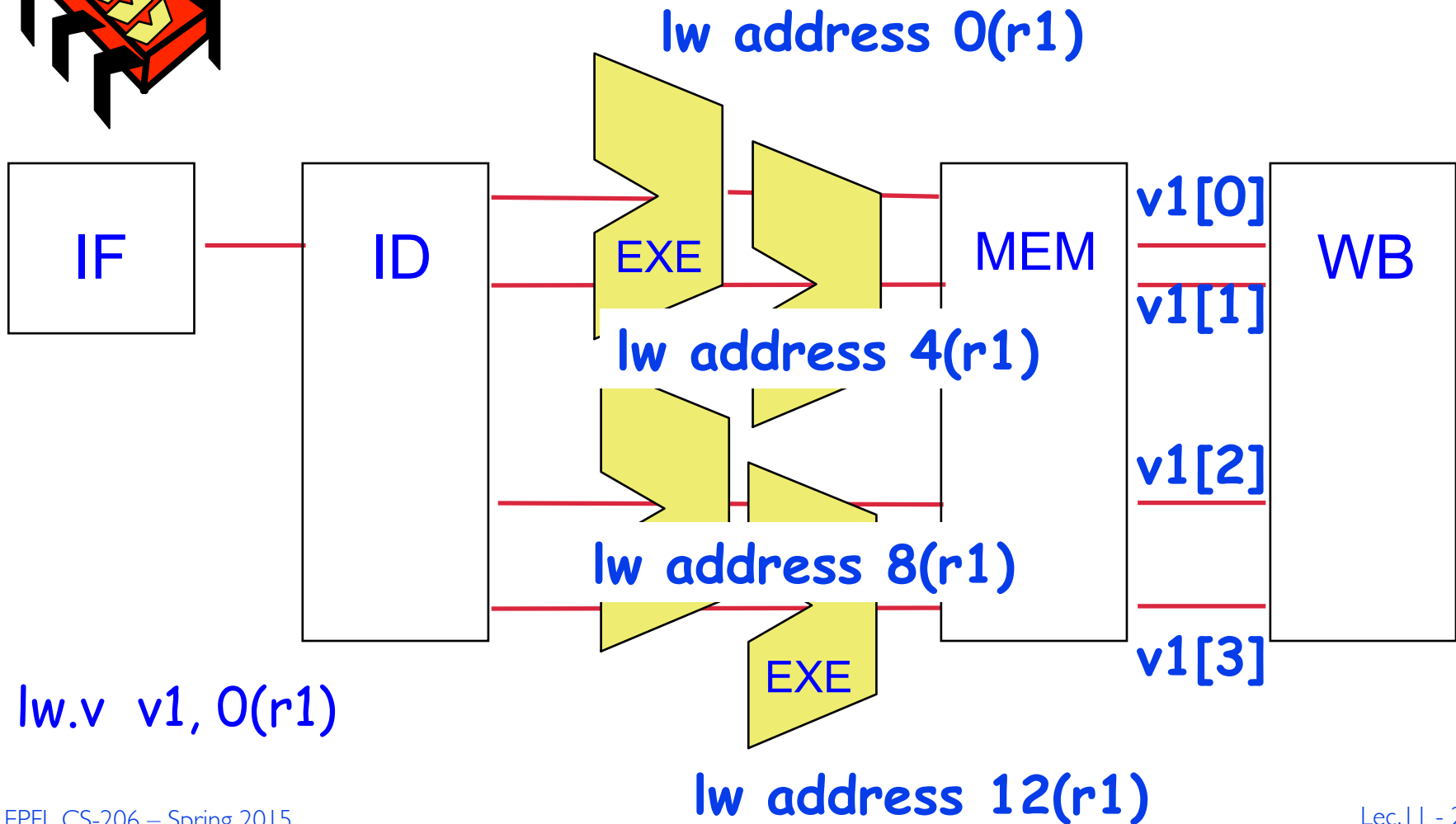
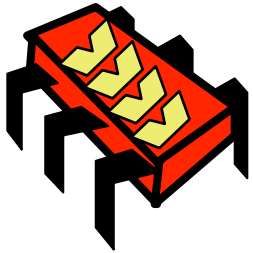
Each vector register is multiple scalar registers

- ▶ In our example, a vector register V has 4 scalars

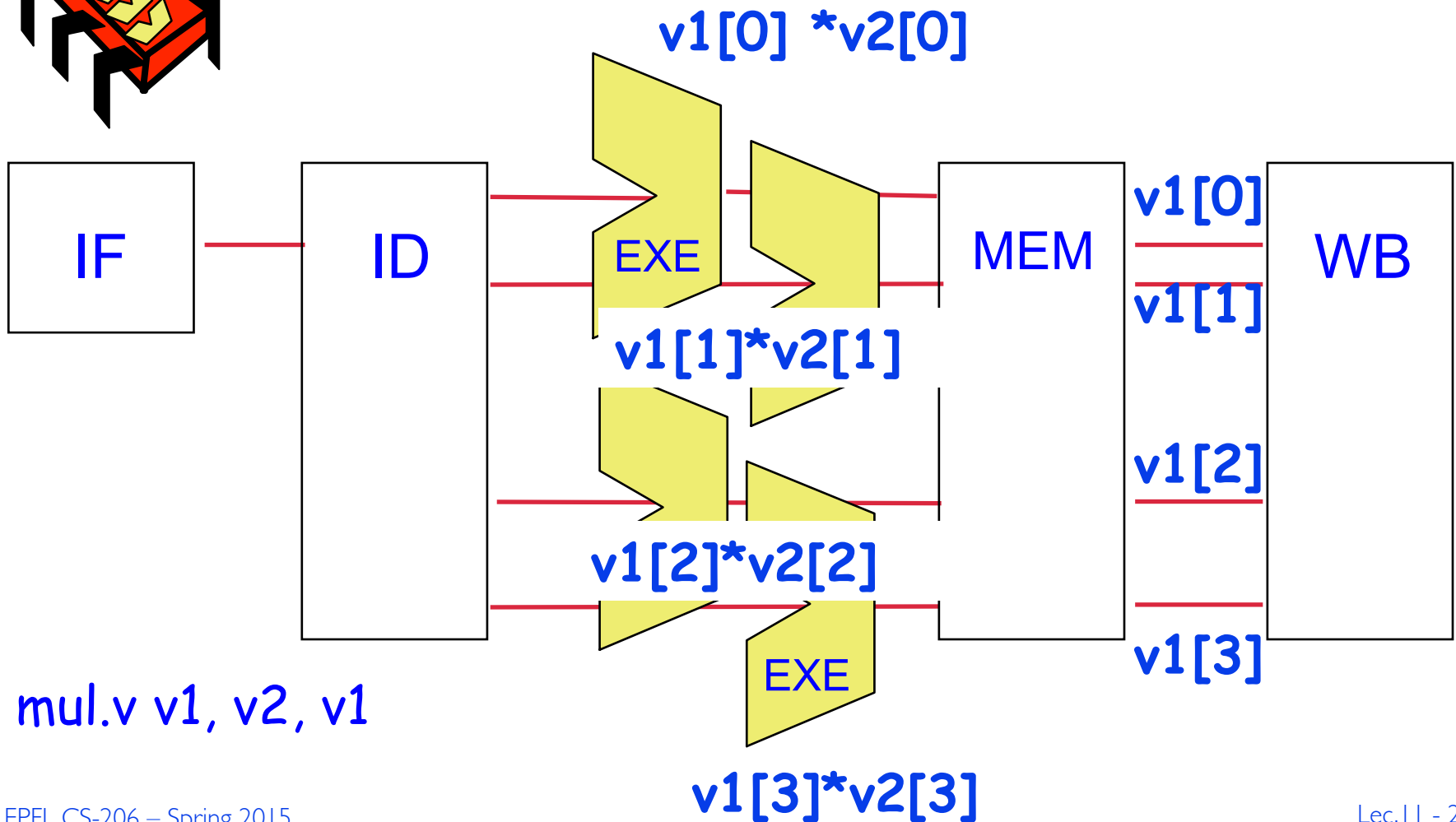
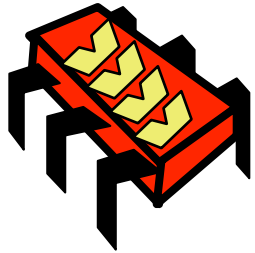
So,

- ▶ `mul.v` `v1, v2, v1` vector dot product $v1 * v2$
- ▶ `mul.sv` `v1, r1, v1` multiplies scalar $r1$ to all elements of $v1$
- ▶ `lw.v` `v1, 0(r1)` loads vector $v1$ from address $r1$
- ▶ `sw.v` `v1, 0(r1)` stores vector $v1$ at address $r1$

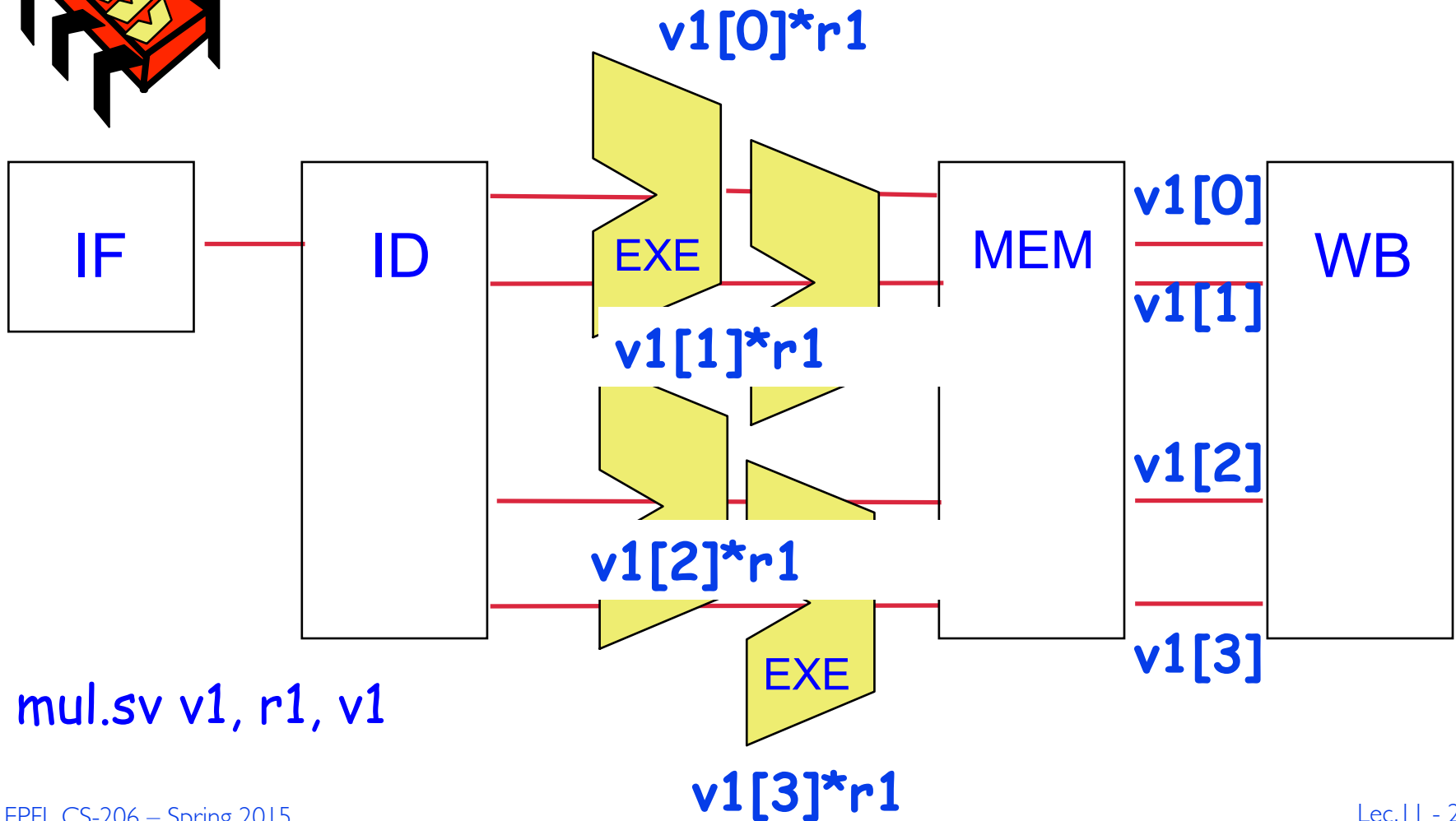
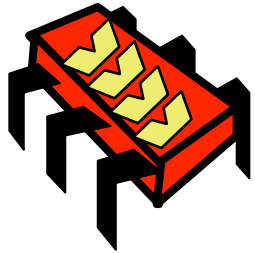
lw.v loads four integers like 4 parallel lw



mul.v vector dot product (4 parallel multiplies)



add.v adds two vectors (4 parallel adds)



Fader loop in Vector MIPS assembly

```
for (i =0; i < N; i++)  
    a[i] = a[i] * fade;  
; a[] -> $2,  
; fade -> $3,  
; &a[N] -> $4  
; $v1 is temp
```

► Should do it four
elements at a time

loop:

Fader loop in Vector MIPS assembly

```
for (i =0; i < N; i++)  
    a[i] = a[i] * fade;  
; a[] -> $2,  
; fade -> $3,  
; &a[N] -> $4  
; $v1 is temp
```

► Should do it four elements at a time

► How many fewer instructions?

```
loop:  
lw.v    $v1, 0($2)  
mul.sv  $v1, $3, $v1  
sw.v    $v1, 0($2)  
addi    $2, $2, 16  
bne     $2, $4, loop
```

Operation & Instruction Count

(from F. Quintana, U. Barcelona.)

Spec92fp Program	Operations (Millions)			Instructions (M)		
	Scalar	Vector	S/V	Scalar	Vector	S/V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Automatic Code Vectorization

```
for (i =0; i < N; i++)  
    a[i] = a[i] * fade;
```

Compiler can detect vector operations

- ▶ Inspect the code
- ▶ Vectorize automatically

But, what about

```
for (i =0; i < N; i++)  
    a[i] = a[b[i]] * fade;
```

Automatic Code Vectorization

```
for (i =0; i < N; i++)  
    a[i] = a[i] * fade;
```

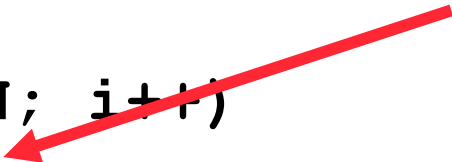
Compiler can detect vector operations

- ▶ Inspect the code
- ▶ Vectorize automatically

But, what about

```
for (i =0; i < N; i++)  
    a[i] = a[b[i]] * fade;
```

**b[i] unknown
at compile
time!**



x86 architecture SIMD support

- ▶ Both current AMD and Intel's x86 processors have ISA and microarchitecture support SIMD operations.
- ▶ ISA SIMD support
 - ▷ MMX, 3DNow!, SSE, SSE2, SSE3, SSE4, AVX
 - ▷ See the flag field in `/proc/cpuinfo`
 - ▷ SSE (Streaming SIMD extensions): ISA extensions to x86
 - ▷ SIMD/vector operations
- ▶ Micro architecture support
 - ▷ Many functional units
 - ▷ 8 | 28-bit **vector registers**, XMM0, XMM1, ..., XMM7

SSE programming

- ▶ Vector registers support three data types:
 - ▷ Integer (16 bytes, 8 shorts, 4 int, 2 long long int, 1 dqword)
 - ▷ single precision floating point (4 floats)
 - ▷ double precision float point (2 doubles).

4x floats



2x doubles



16x bytes



8x words



4x dwords



2x qwords



1x dqword



SSE instructions

▶ Arithmetic instructions

- ▷ ADD, SUB, MUL, DIV, SQRT, MAX, MIN, RCP, etc
- ▷ PD: two doubles, PS: 4 floats, SS: scalar
 - ▷ ADDPS – add four floats, ADDSS: scalar add

▶ Logical instructions

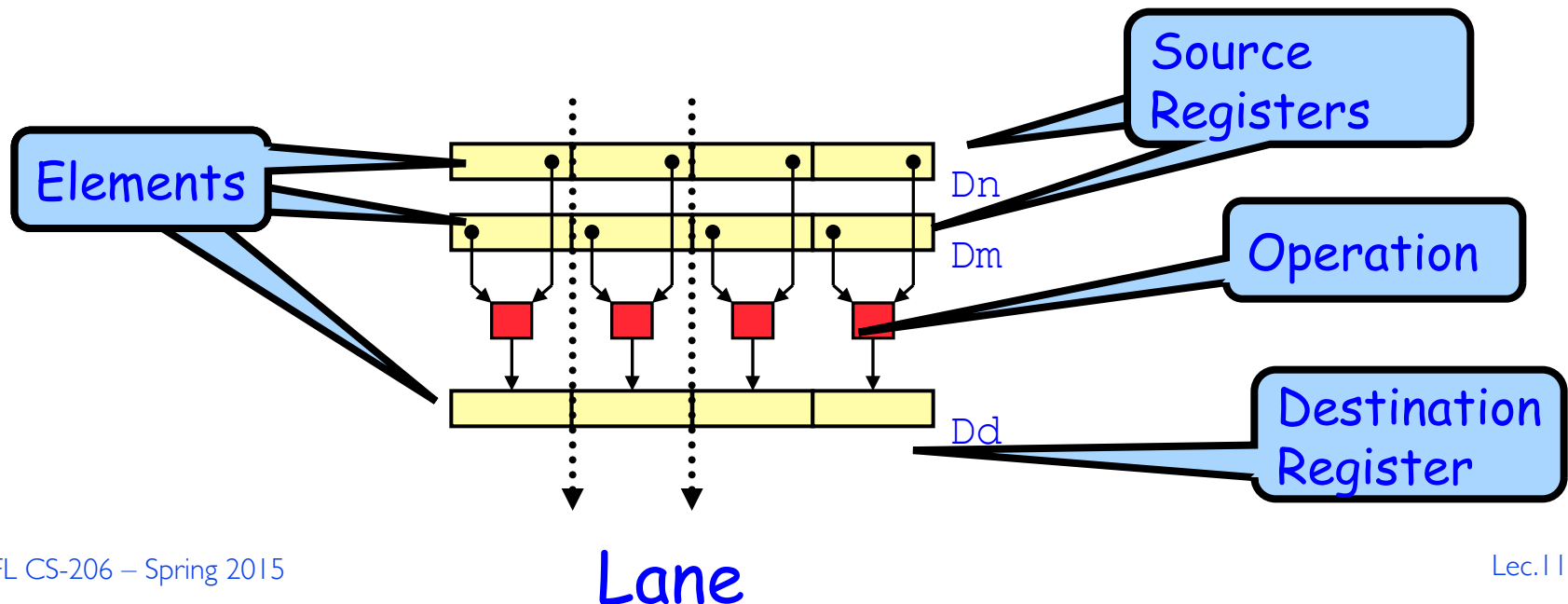
- ▷ AND, OR, XOR, ANDN, etc
 - ▷ ANDPS – bitwise AND of operands
 - ▷ ANDNPS – bitwise AND NOT of operands

▶ Comparison instruction:

- ▷ CMPPS, CMPSS – compare operands and return all 1's or 0's

SIMD extensions in ARM: NEON

- ▶ 32 x 64-bit registers (also used as 16 x 128-bit registers)
- ▶ Registers considered as vectors of same data type
- ▶ Data types: signed/uns. 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
- ▶ Instructions perform the same operation in all lanes



This Course:

Data Parallel Processor Architecture

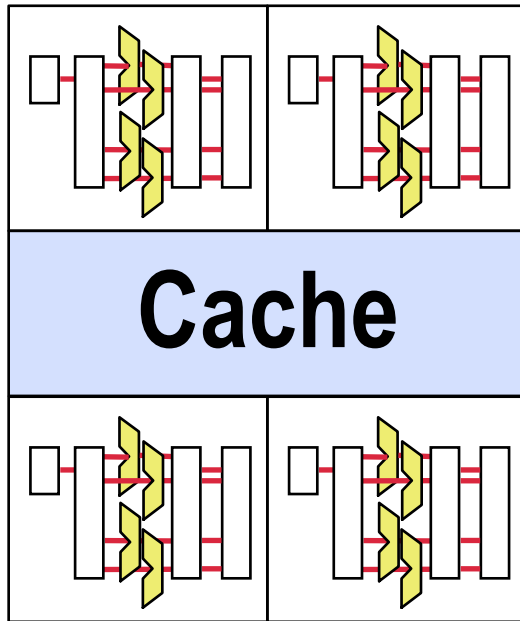
1. Vector Processors

- ▷ Pipelined execution
- ▷ SIMD: Single instruction, multiple data
- ▷ Example: modern ISA extensions

2. Graphics Processing Units (GPUs)

- ▷ Dense grid of ALUs
- ▷ SIMT: Single instruction, multiple threads
- ▷ Integrated vs. discrete

CPU vs. GPU

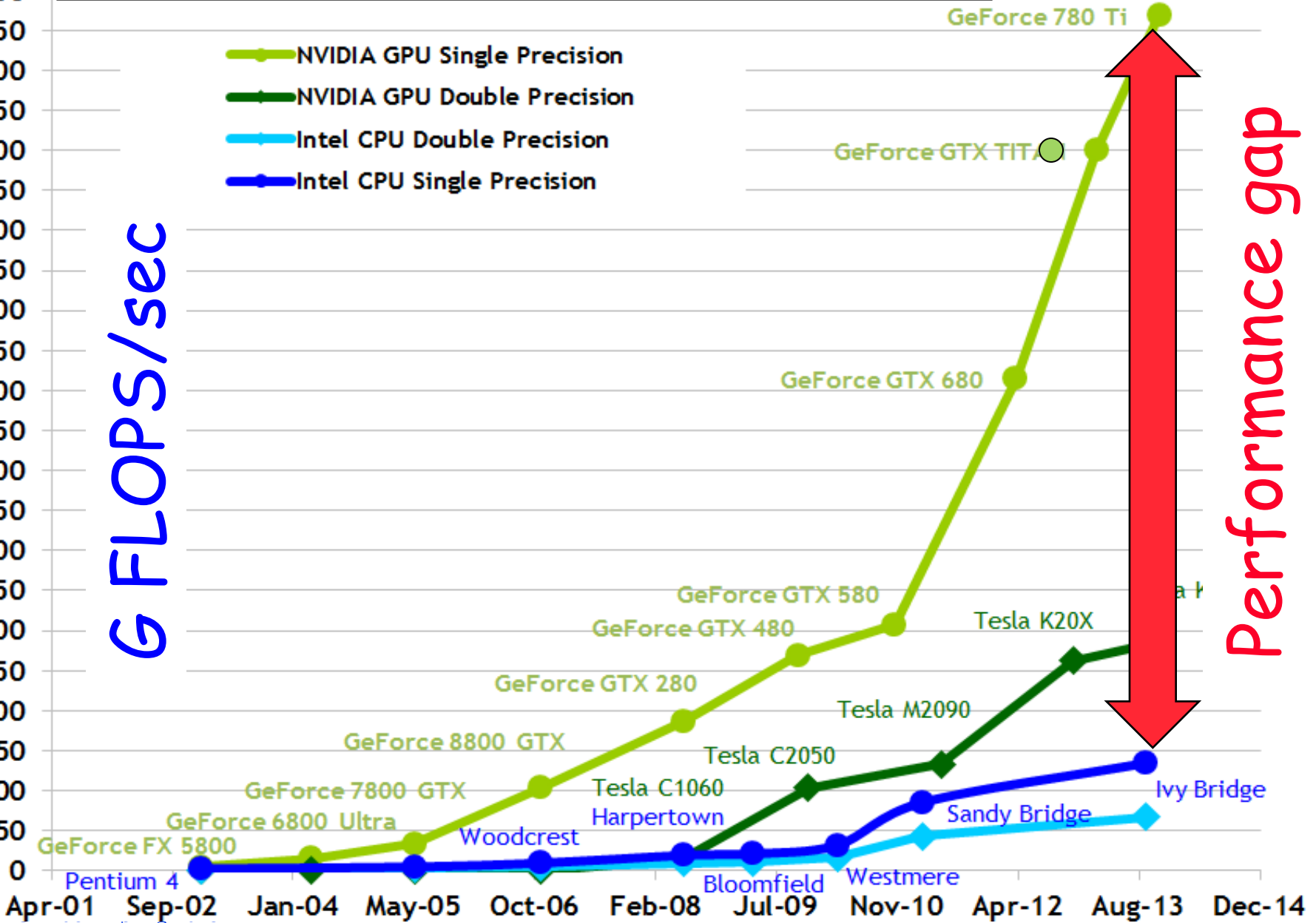


- ▶ Tens of cores
- ▶ Mostly control logic
- ▶ Large caches
- ▶ Regular threads (e.g., Java)
- ▶ Thousands of tiny cores
- ▶ Mostly ALU
- ▶ Little cache
- ▶ Special threads (e.g., CUDA)

GPUs are highly concurrent!

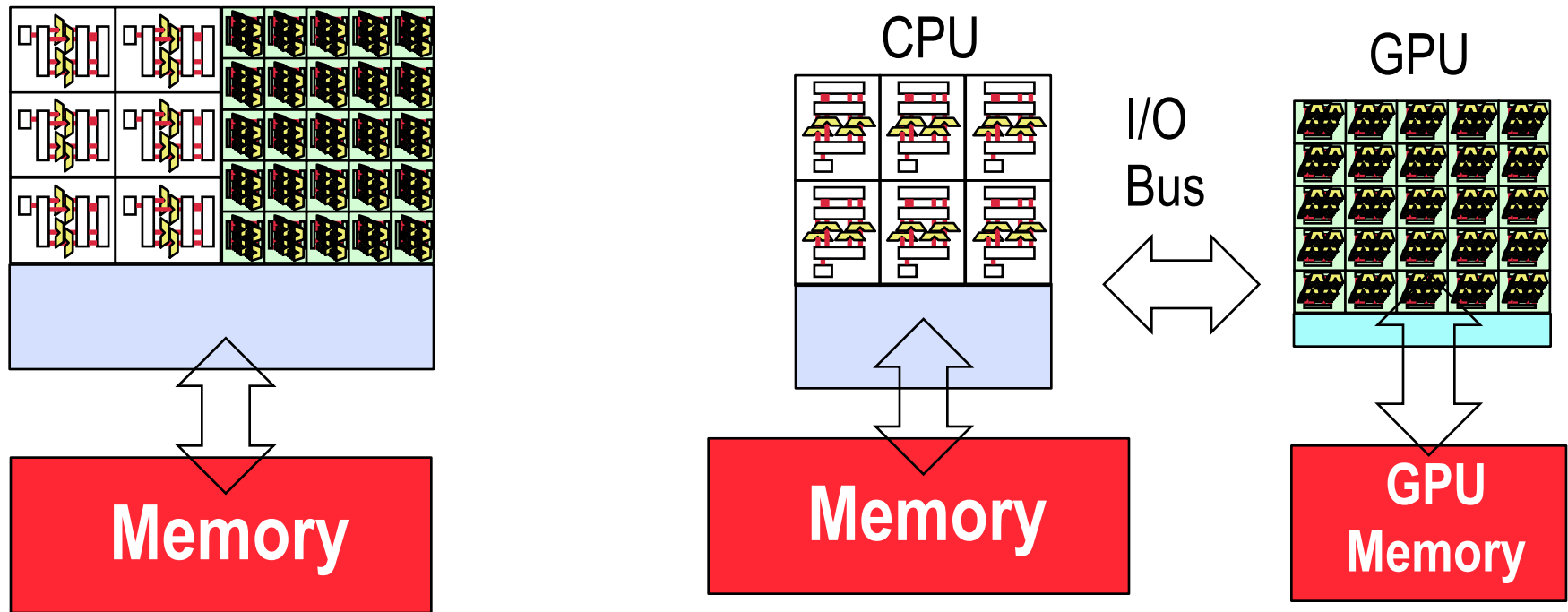
G FLOPS/sec

- NVIDIA GPU Single Precision
- NVIDIA GPU Double Precision
- Intel CPU Double Precision
- Intel CPU Single Precision



Performance gap

Integrated vs. Discrete GPU



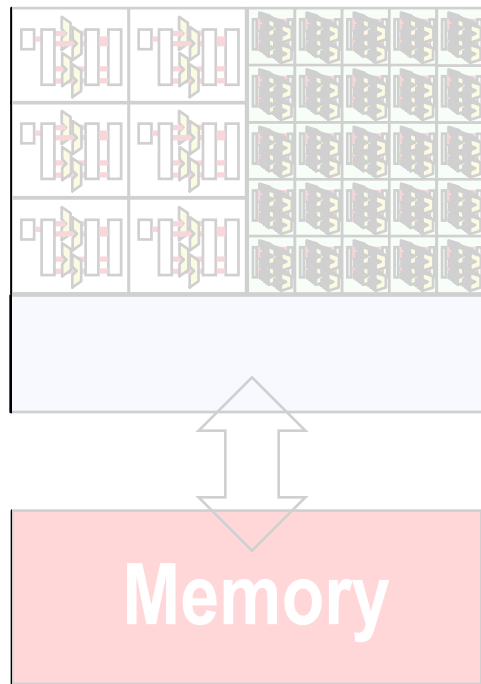
Integrated (e.g., AMD)

- ▶ Shared cache hierarchy
- ▶ One memory

Discrete (e.g., nVidia)

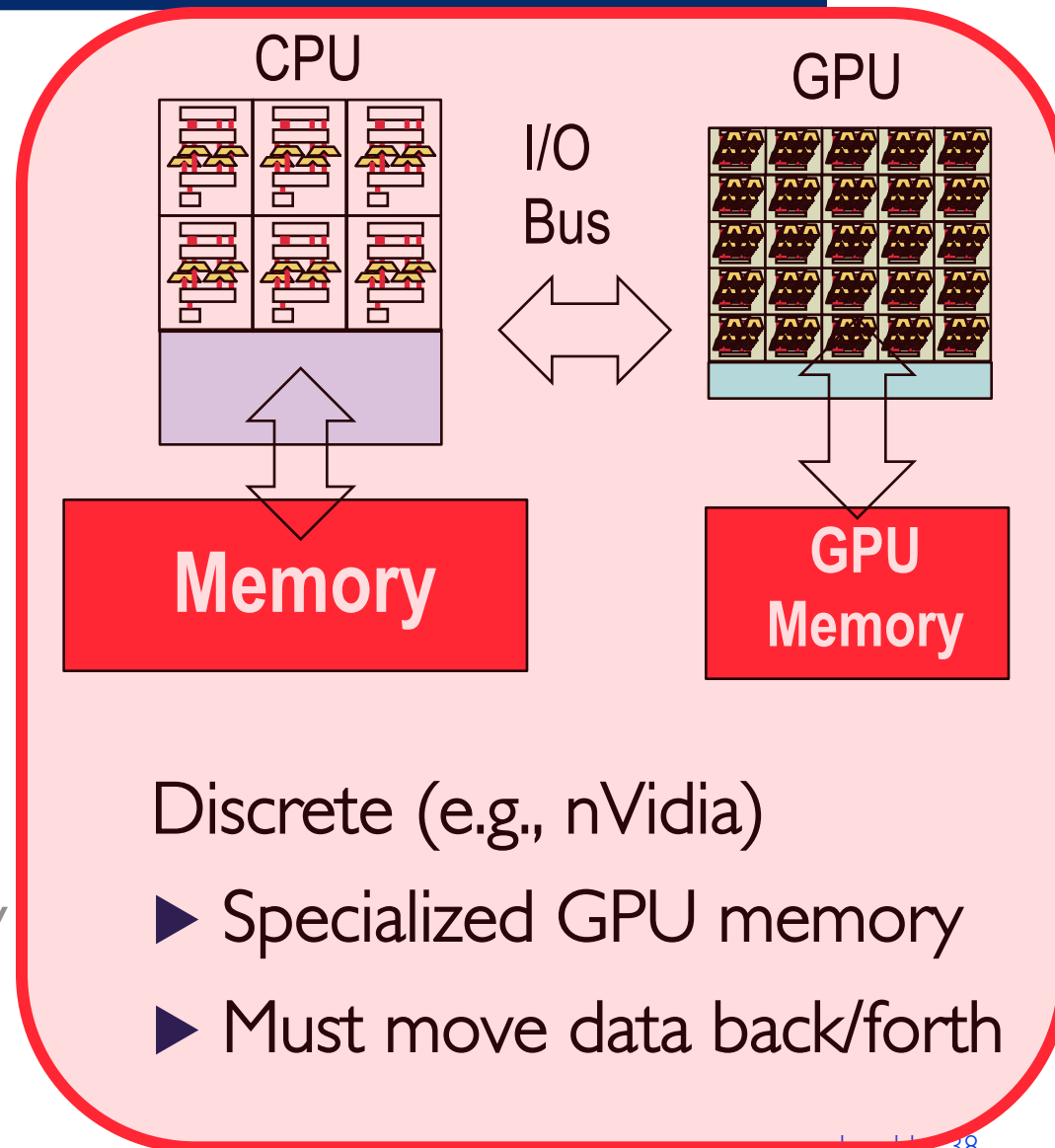
- ▶ Specialized GPU memory
- ▶ Must move data back/forth

This course: Discrete GPU



Integrated (e.g., AMD)

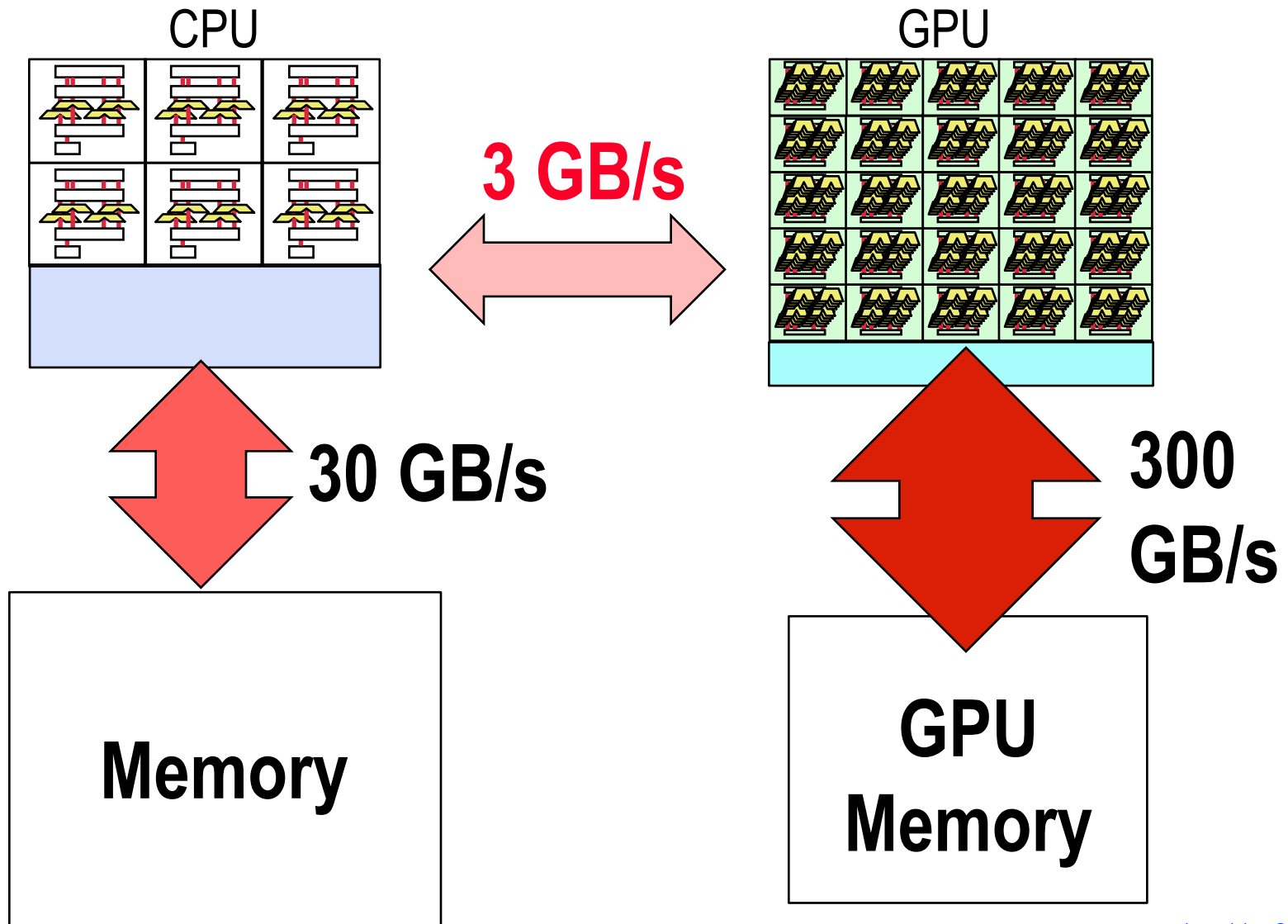
- ▶ Shared cache hierarchy
- ▶ One memory



Discrete (e.g., nVidia)

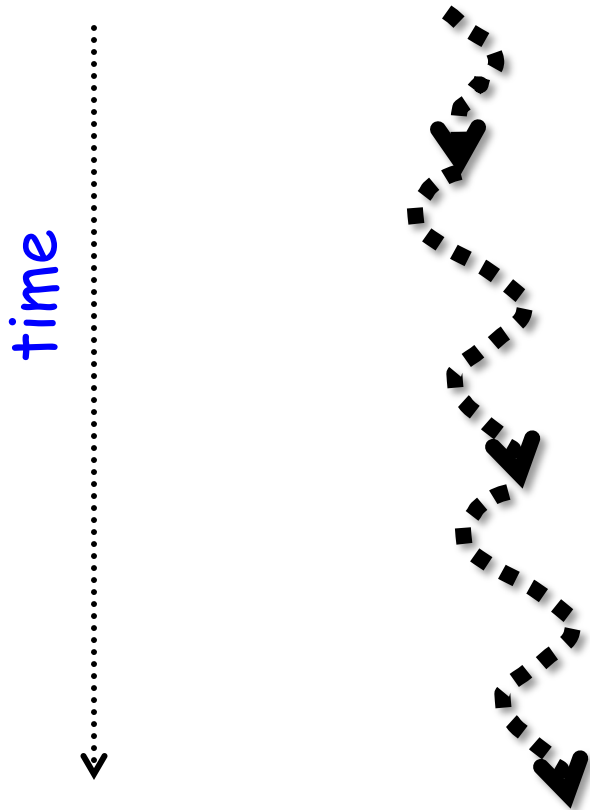
- ▶ Specialized GPU memory
- ▶ Must move data back/forth

Warning! CPU/GPU connection is a bottleneck



Sequential Execution Model / SISD

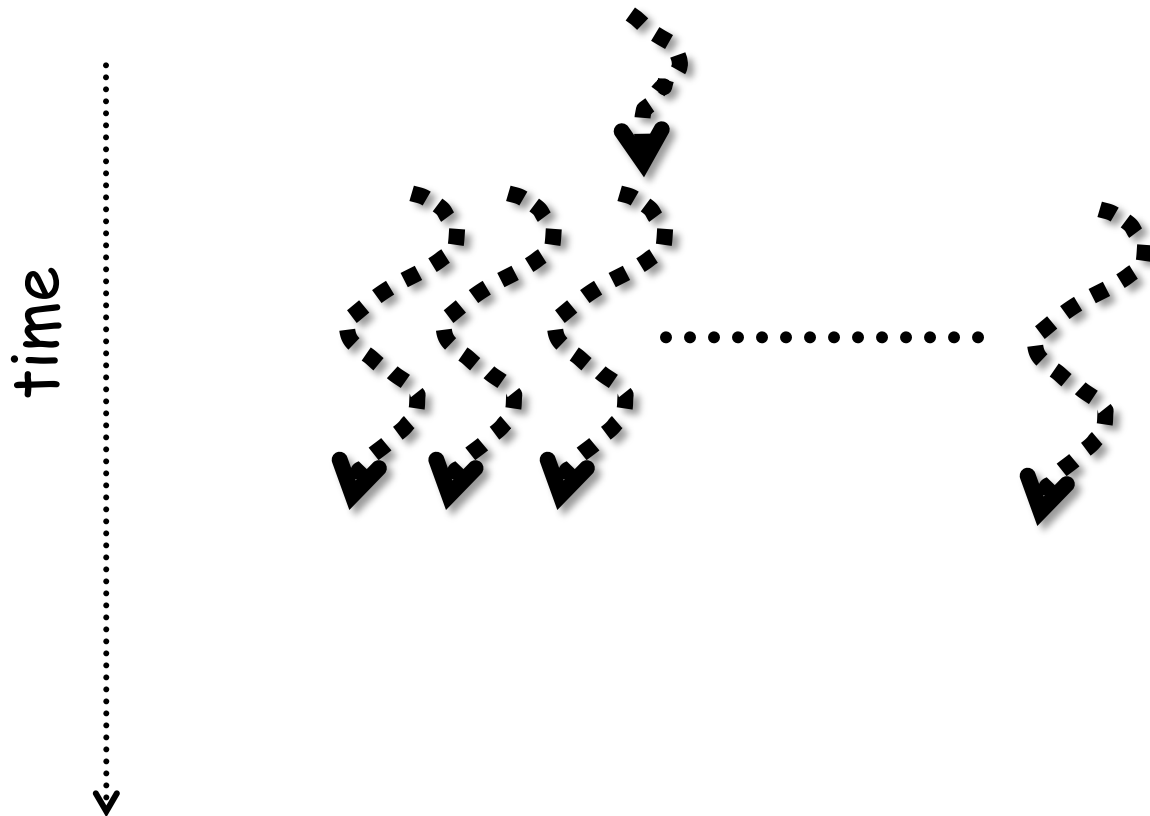
```
int a[N]; // N is large
for (i =0; i < N; i++)
    a[i] = a[i] * fade;
```



Flow of control / Thread
One instruction at the time
Optimizations possible at
the machine level

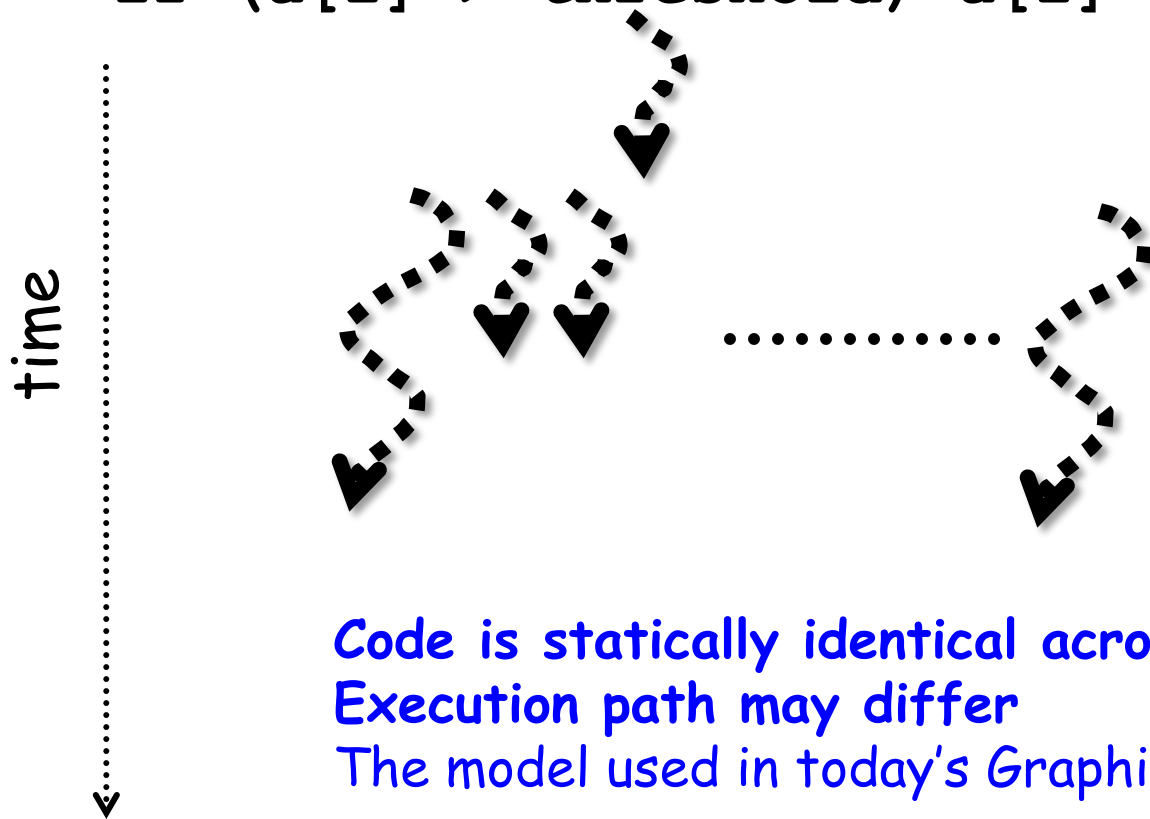
Data Parallel Execution Model / SIMD

```
int a[N]; // N is large
for all elements do in parallel
    a[i] = a[i] * fade;
```



Single Program Multiple Data / SPMD

```
int a[N]; // N is large
for all elements do in parallel
  if (a[i] > threshold) a[i]*= fade;
```



Code is statically identical across all threads
Execution path may differ
The model used in today's Graphics Processors

Killer app? 3D Graphics

Example apps:

- ▷ Games
- ▷ Engineering/CAD

Computation:

- ▷ Start with triangles (points in 3D space)
- ▷ Transform (move, rotate, scale)
- ▷ Paint / Texture mapping
- ▷ Rasterize → convert into pixels
- ▷ Light & Hidden “surface” elimination

Bottom line:

- ▷ Tons of independent calculations
- ▷ Lots of identical calculations

Target Applications

```
int a[N]; // N is large
```

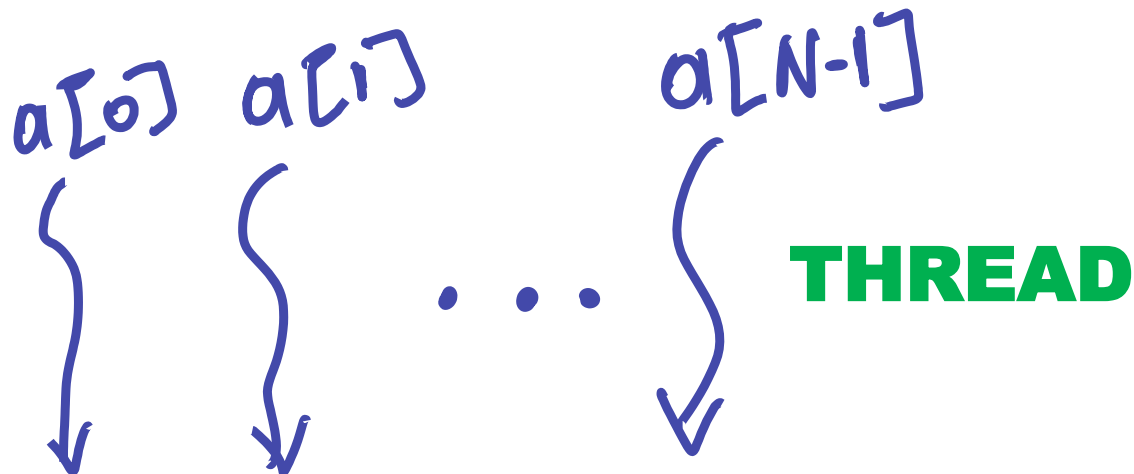
```
for all elements of an array
```

```
  a[i] = a[i] * fade
```

Kernel

► Lots of independent computations

▷ CUDA threads need not be completely independent



Programmer's View of the GPU

- ▶ GPU: a compute **device** that:
 - ▷ Is a coprocessor to the CPU or **host**
 - ▷ Has its own DRAM (**device memory**)
 - ▷ Runs many **threads in parallel**

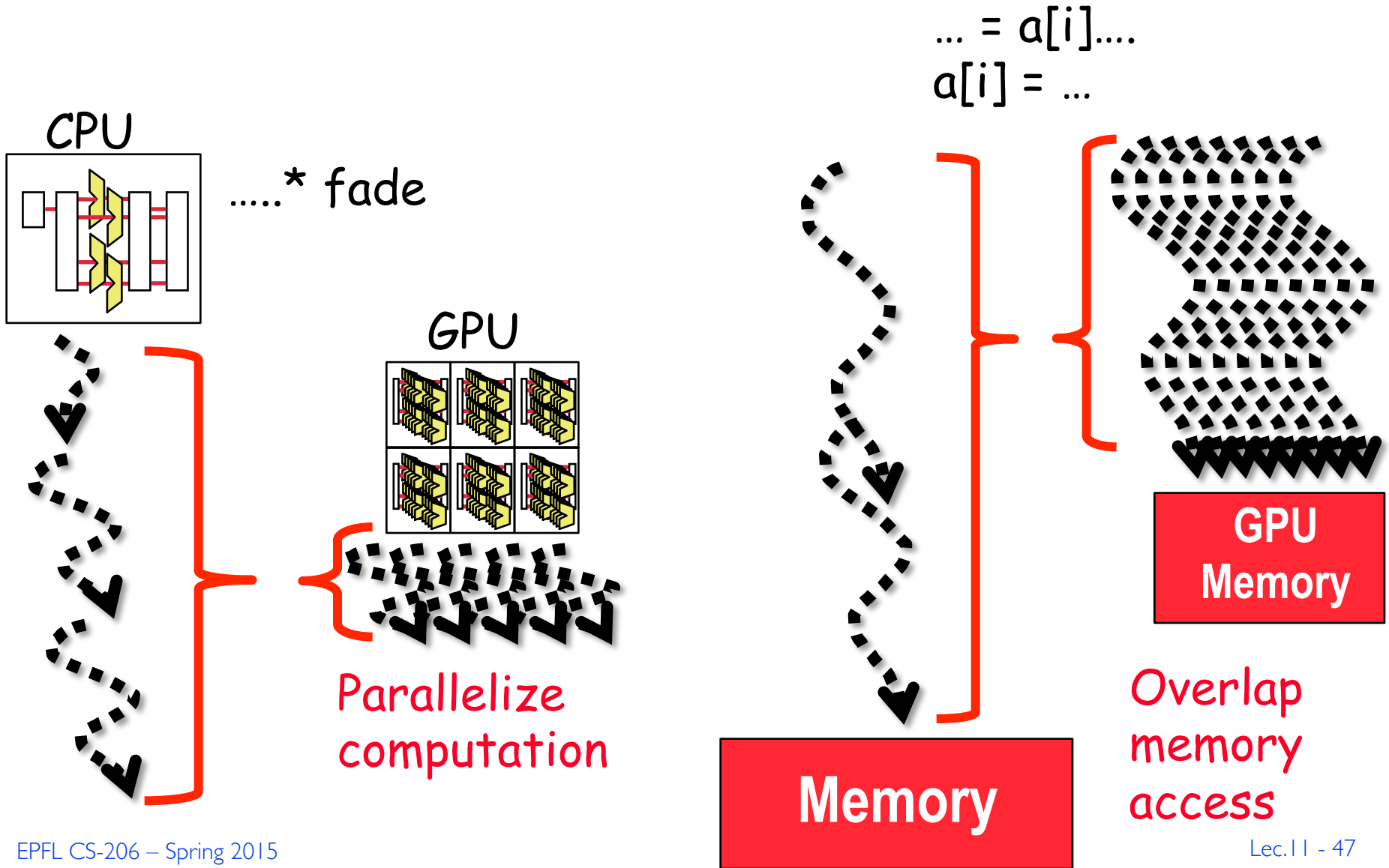
- ▶ Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads

GPU vs. CPU Threads

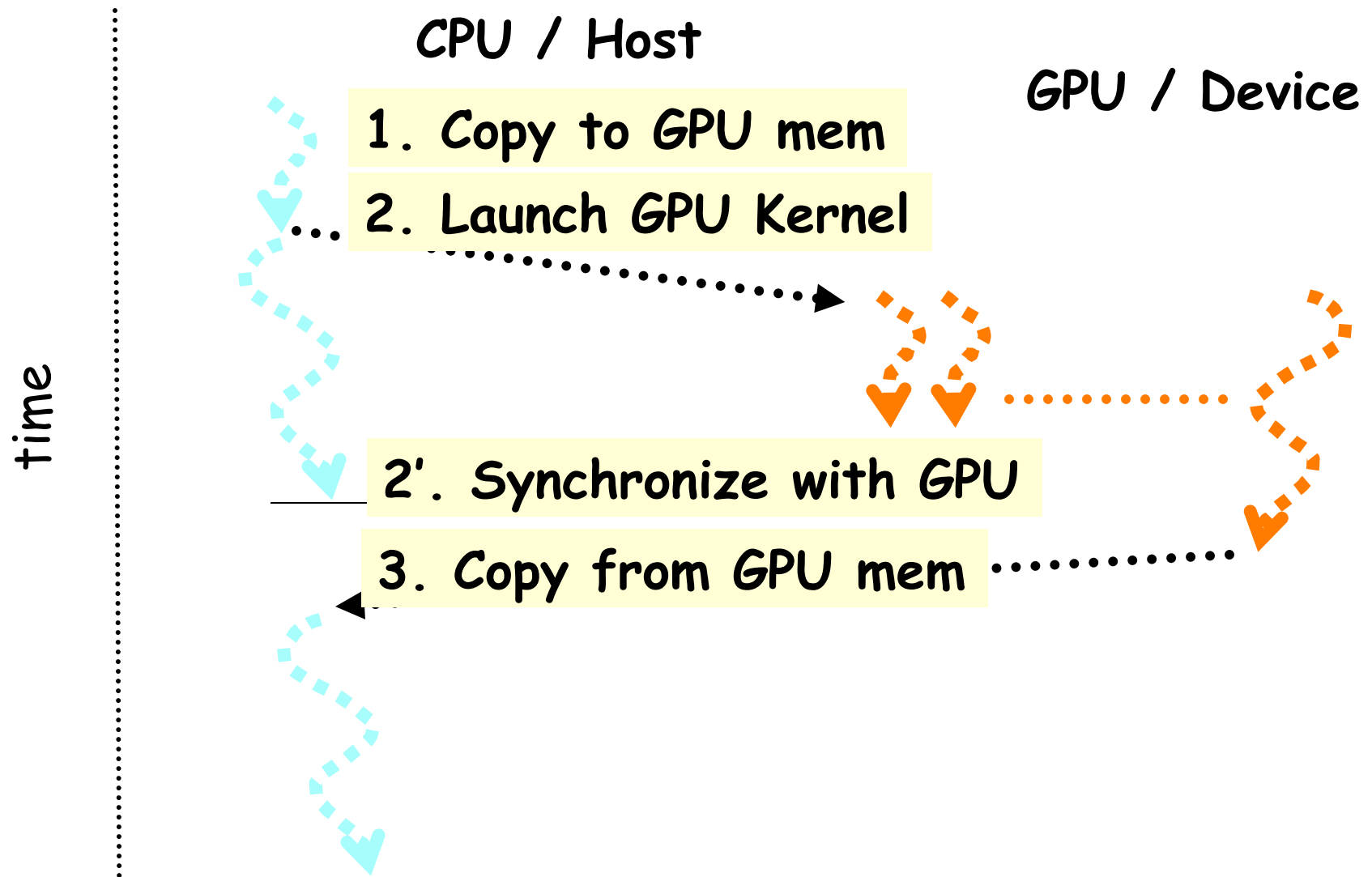
- ▶ GPU threads are extremely lightweight
 - ▷ Little creation overhead (unlike Java)
 - ▷ e.g., ~microseconds
 - ▷ All done in hardware

- ▶ GPU needs 1000s of threads for full efficiency
 - ▷ Multi-core CPU needs only a few

GPU threads help in two ways!

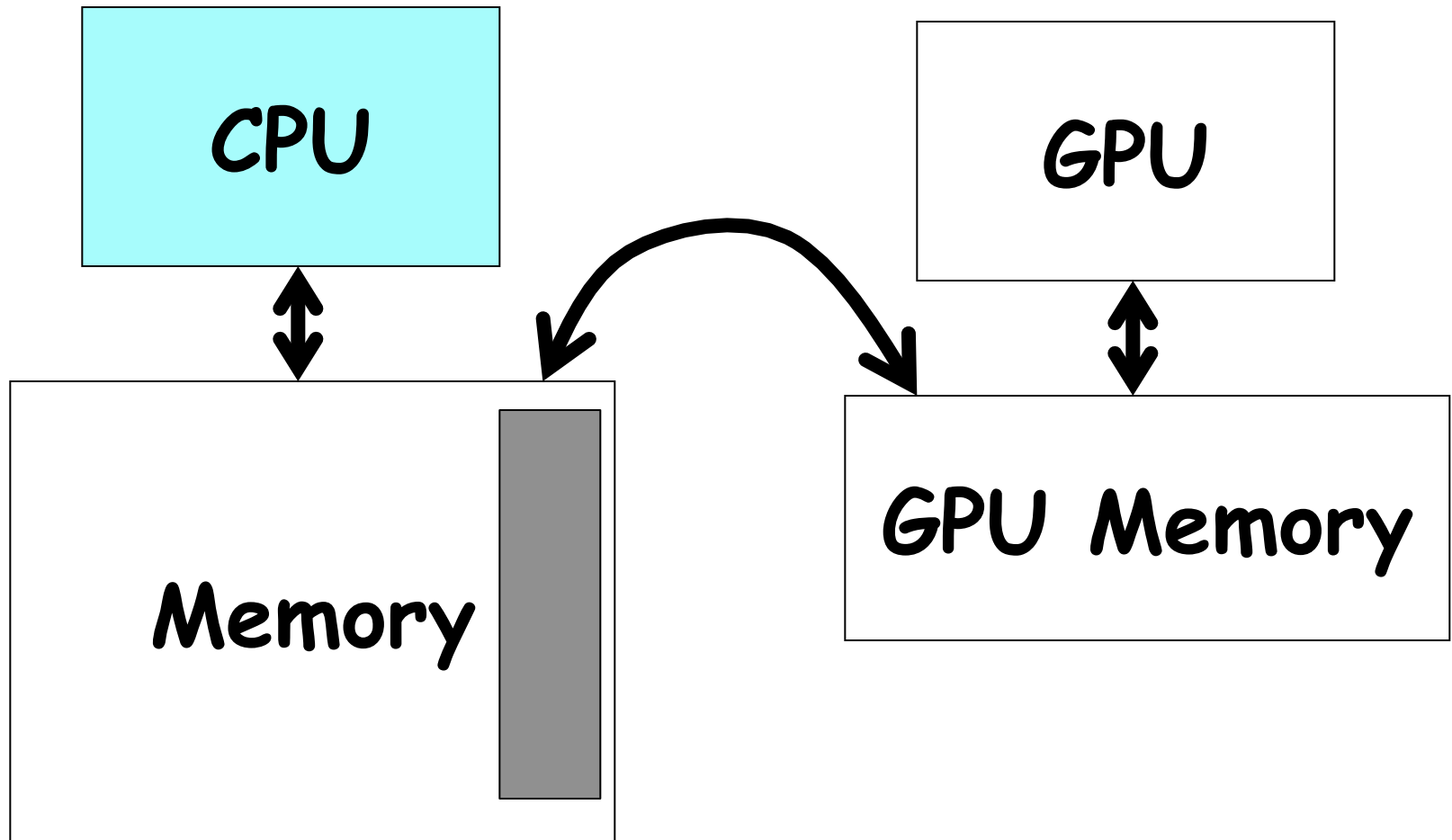


Execution Timeline



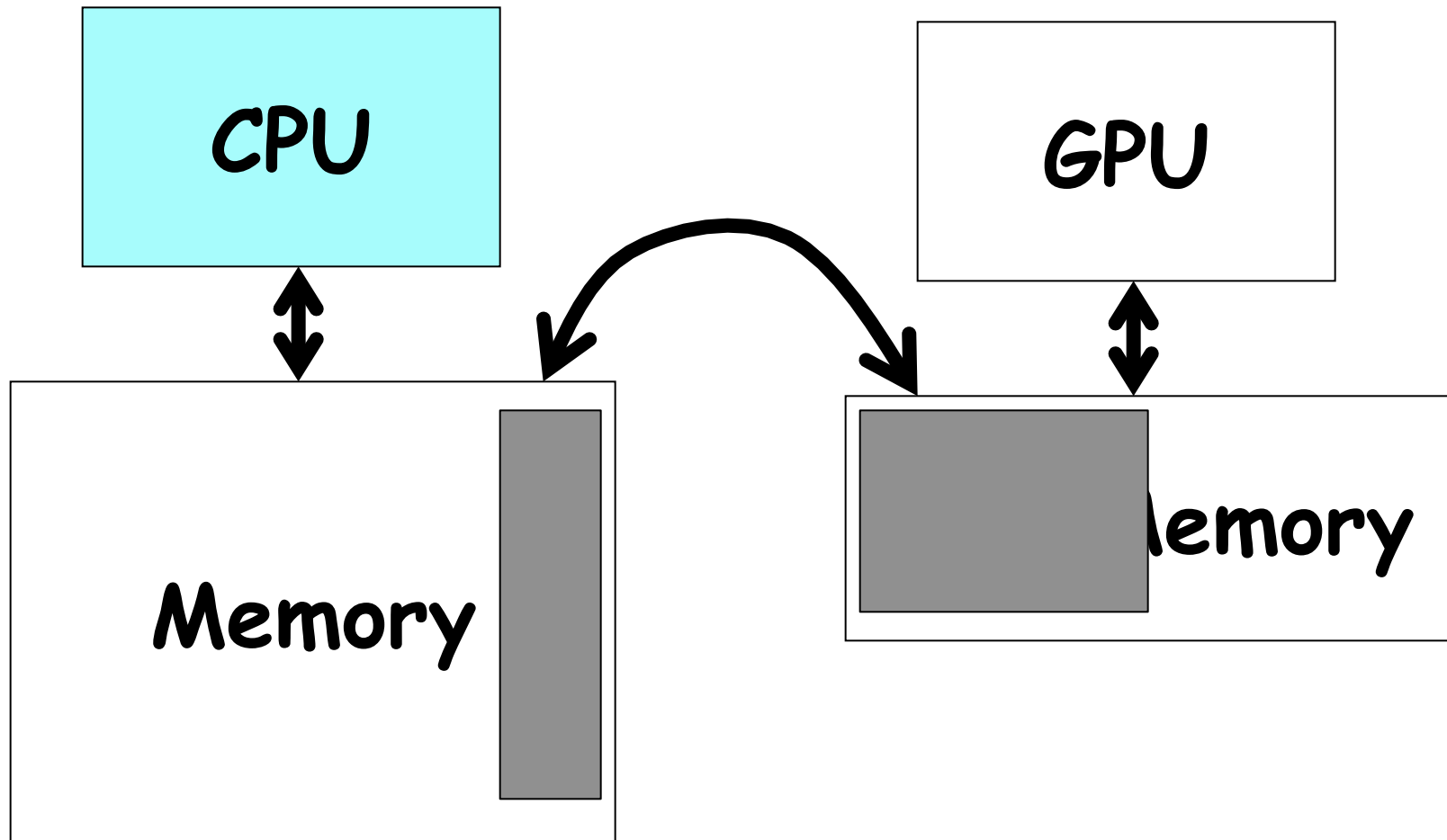
Programmer's view

- ▶ First create data in CPU memory



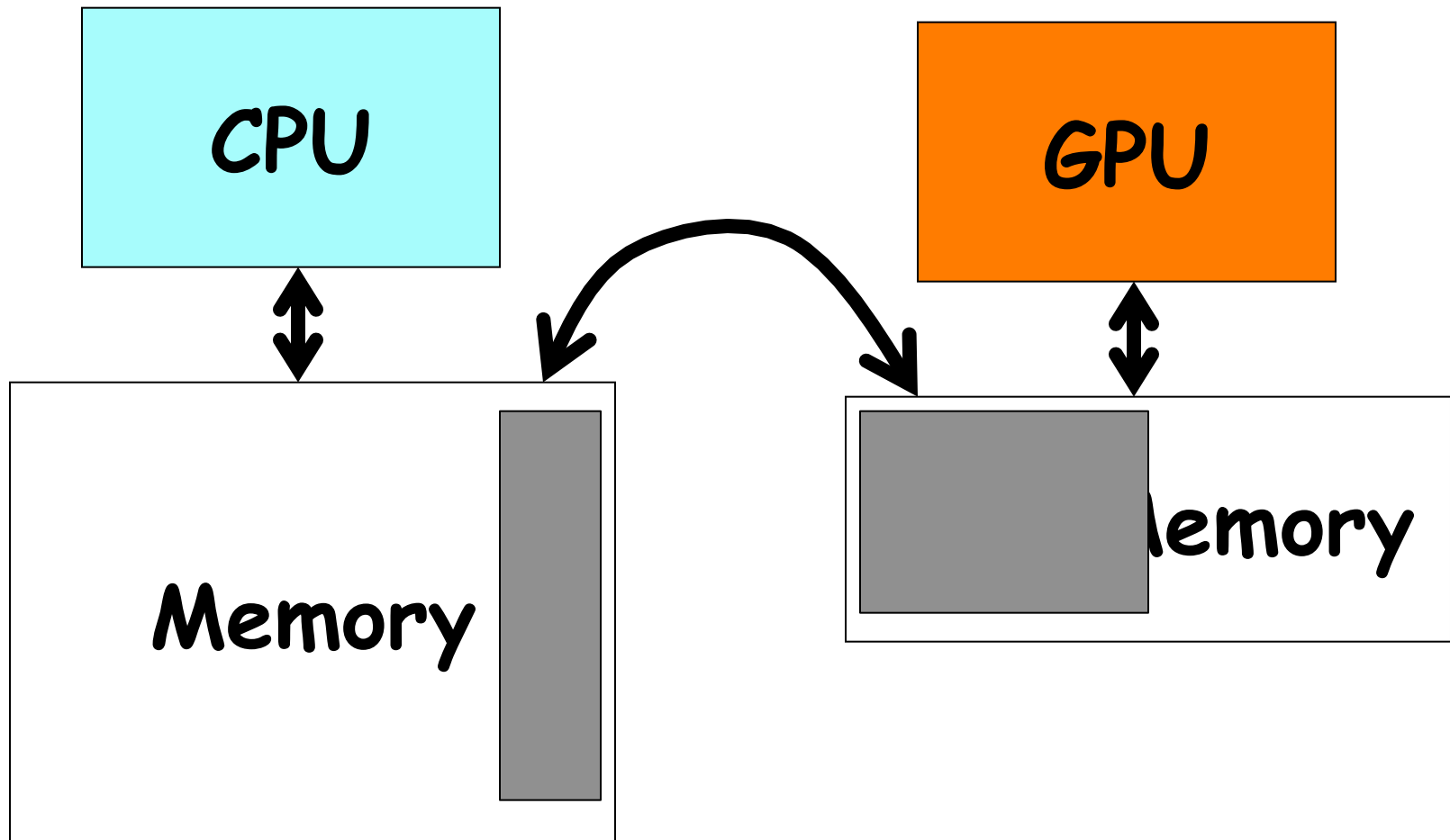
Programmer's view

- ▶ Then Copy to GPU



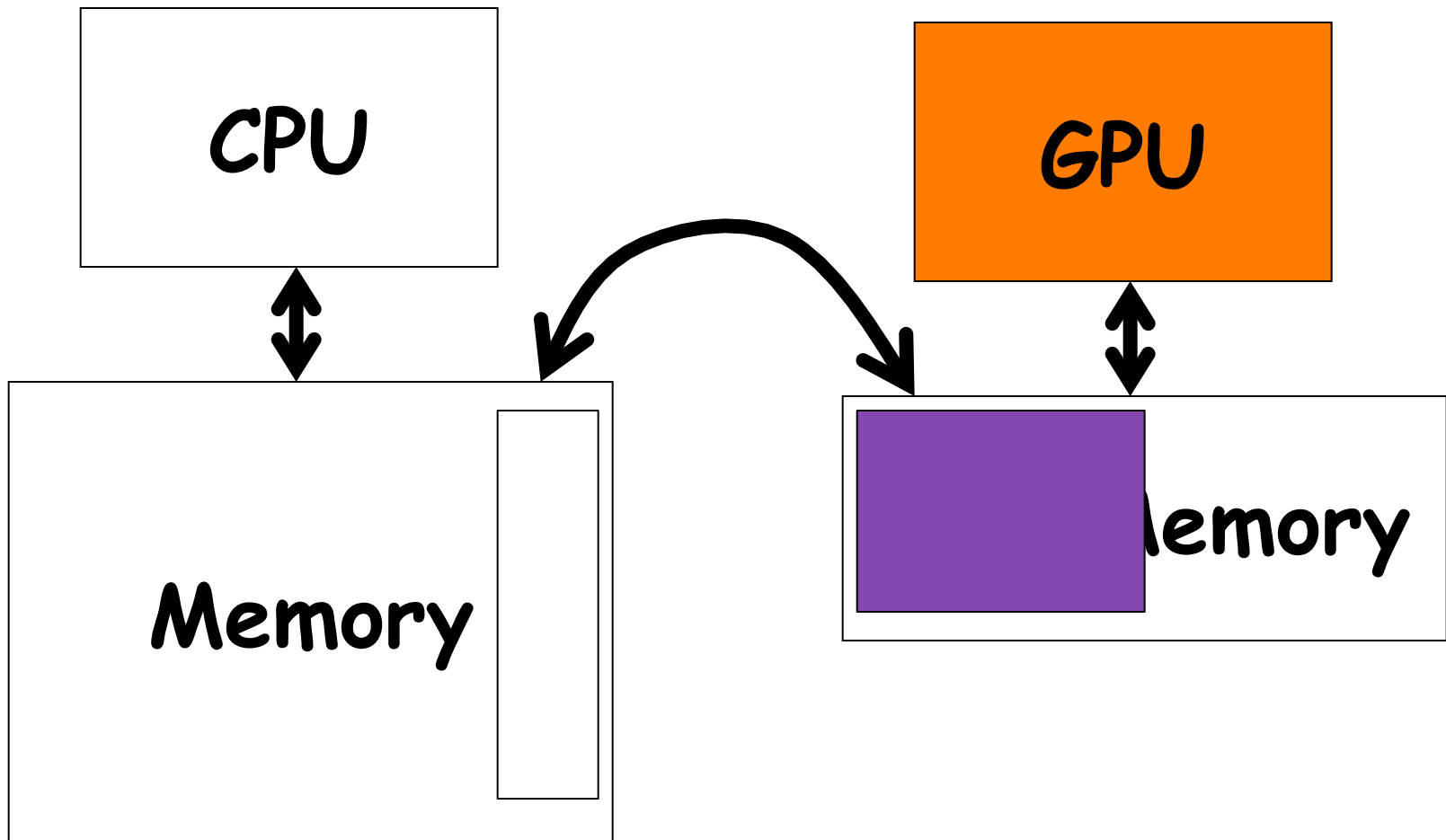
Programmer's view

- ▶ GPU starts computation → runs a **kernel**
- ▶ CPU can also continue



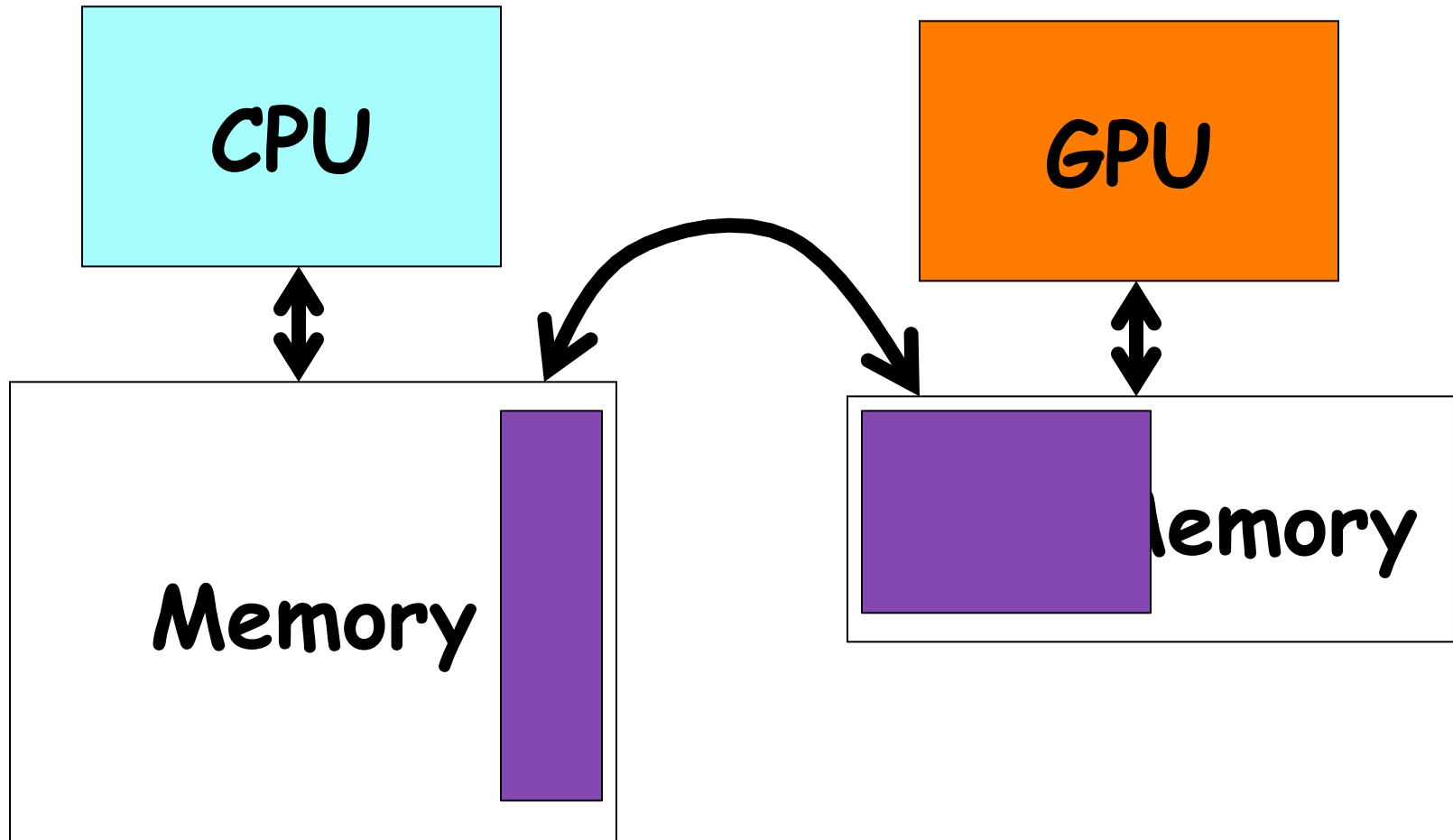
Programmer's view

- ▶ CPU and GPU Synchronize



Programmer's view

- ▶ Copy results back to CPU



Programming Languages

▶ CUDA

- ▷ nVidia
- ▷ Has market lead

▶ OpenCL

- ▷ Many including nVidia
- ▷ CUDA superset
- ▷ Targets many different devices, e.g., CPUs + programmable accelerators
- ▷ Fairly new

▶ Both are evolving

Computation partitioning

- ▶ Think of computation as a series of loops
- ▶ Think of data as an array

```
for (i = 0; i < big_number; i++)
```

```
    a[i] = some function
```

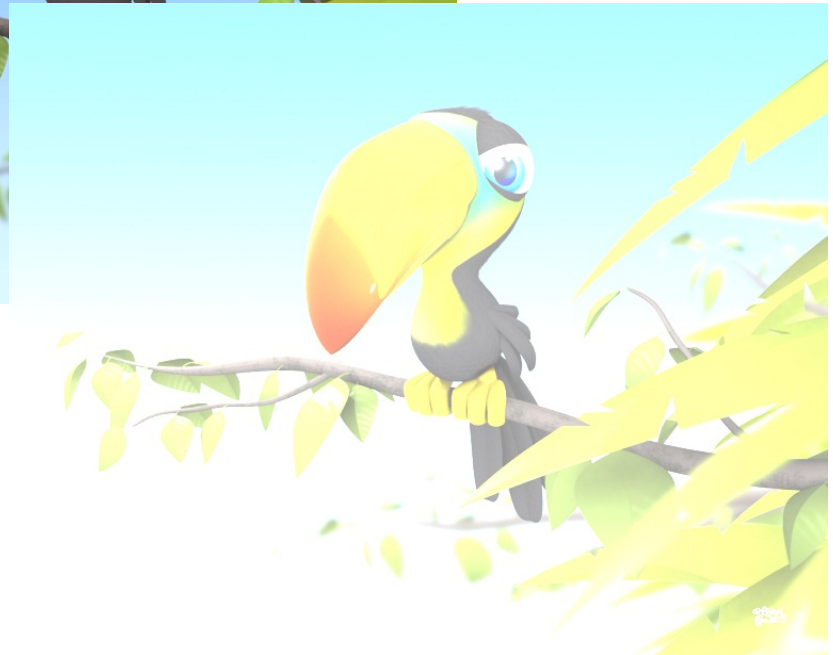
```
for (i = 0; i < big_number; i++)
```

```
    a[i] = some other function
```

```
for (i = 0; i < big_number; i++)
```

```
    a[i] = some other function
```

Kernels



What is the kernel here?

My first CUDA Program

```
__global__ void fadepic(int *a, int fade, int N)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) a[i] = a[i] * fade;
}
```

GPU

```
int main()
{
    int h[N];
    int *d;
    cudaMalloc ((void **) &d, SIZE);
    ....

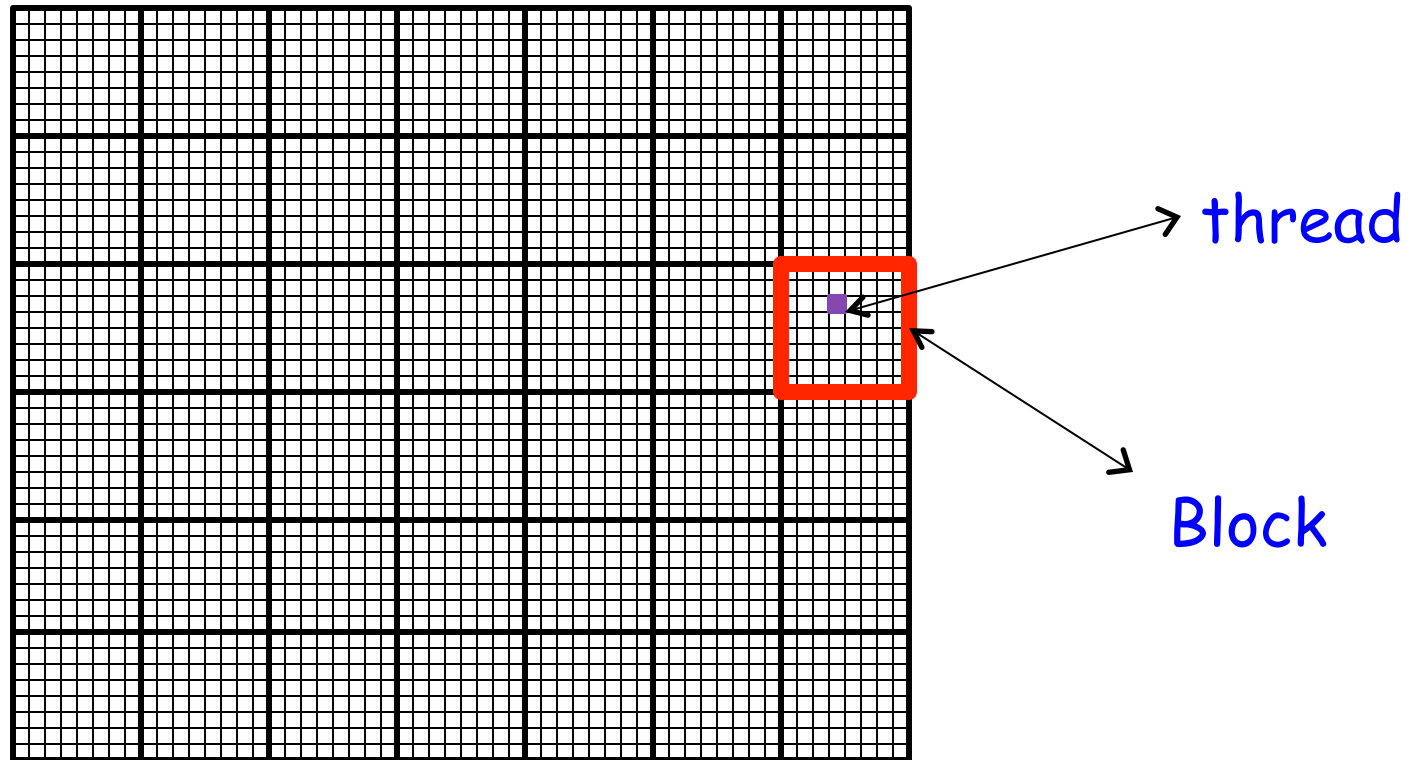
    cudaThreadSynchronize ();
    cudaMemcpy (d, h, SIZE, cudaMemcpyHostToDevice));

    fadepic<<< n_blocks, block_size >>> (d, 10.0, N);

    cudaDeviceSynchronize ();
    cudaMemcpy (h, d, SIZE, cudaMemcpyDeviceToHost));
    CUDA_SAFE_CALL (cudaFree (d));
}
```

CPU

Per Kernel Computation Partitioning



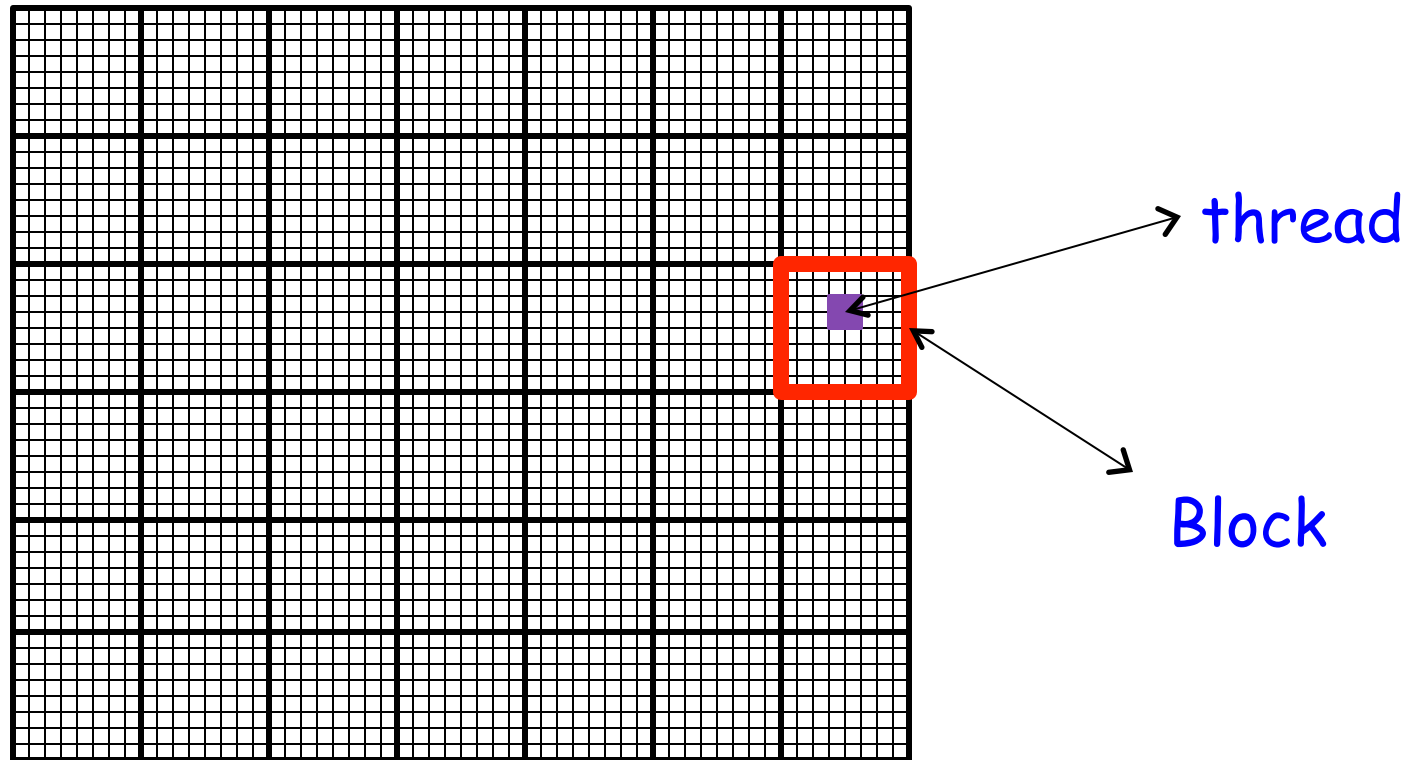
Threads within a block can communicate/synchronize

- ▷ Run on the same core

Threads across blocks can't communicate

- ▷ Shouldn't touch each others data (undefined behavior)

Per Kernel Computation Partitioning



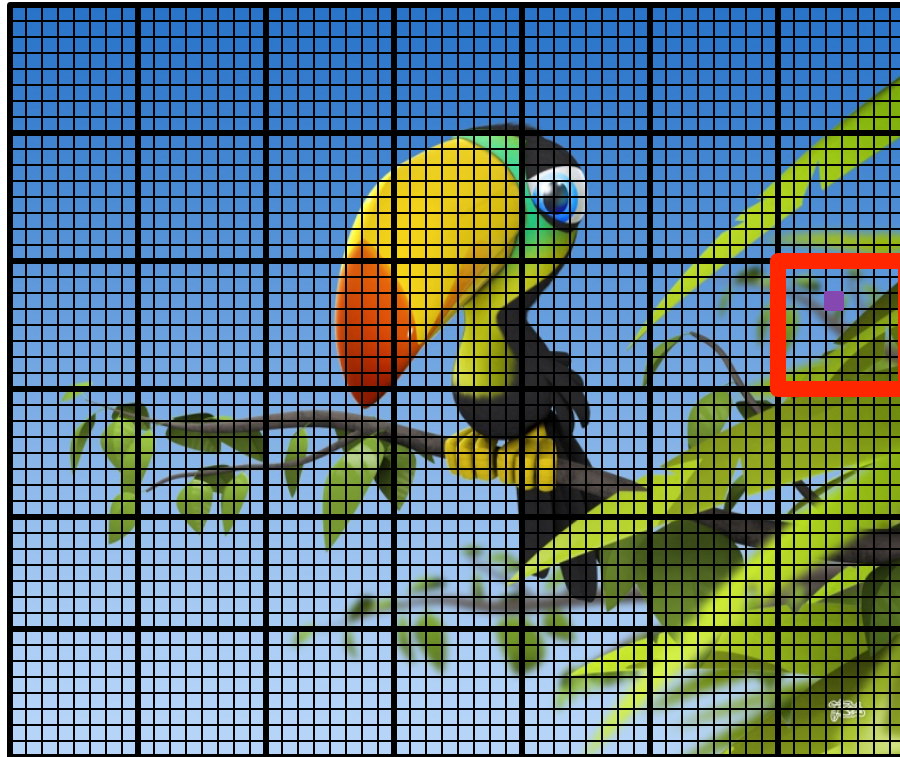
- ▶ One thread can process multiple data elements
- ▶ Other mappings are possible and often desirable
 - ▷ We will talk about this later

Fade example

- ▶ Each thread will process one pixel

for all elements do in parallel

```
a[i] = a[i] * fade;
```



Code Skeleton

▶ CPU:

- ▷ Initialize image from file
- ▷ Allocate buffer on GPU
- ▷ Copy image to buffer
- ▷ Launch GPU kernel
 - ▷ Reads and writes into buffer
- ▷ Copy buffer back to CPU
- ▷ Write image to a file

▶ GPU:

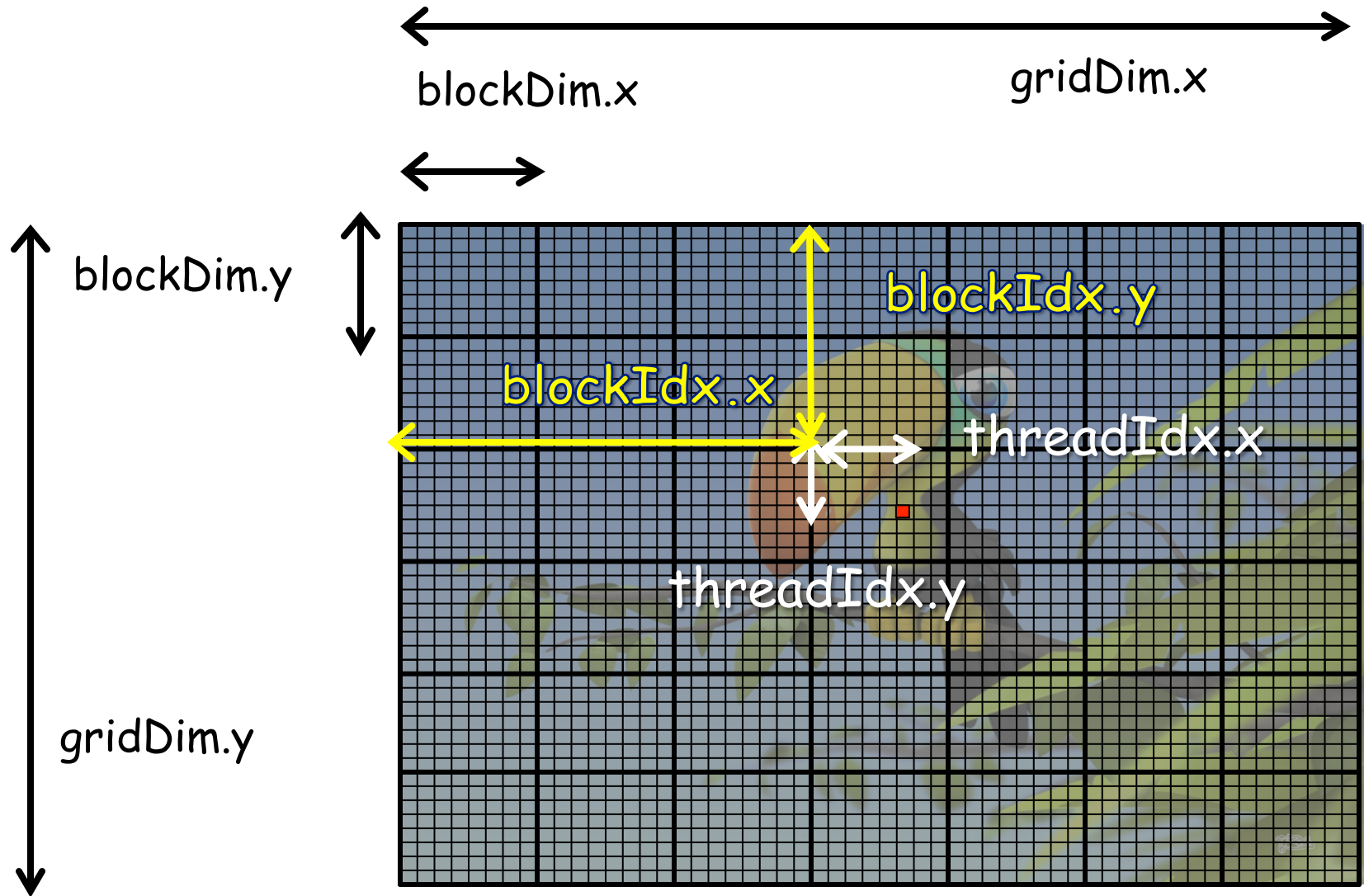
- ▷ Launch a thread per pixel

GPU Kernel pseudo-code

```
__global__ void fadepic (int *a,  
                        int fade,  
                        int N)  
{  
    int v = a[x][y];  
    v = v * fade;  
    a[x][y] = v;  
}
```

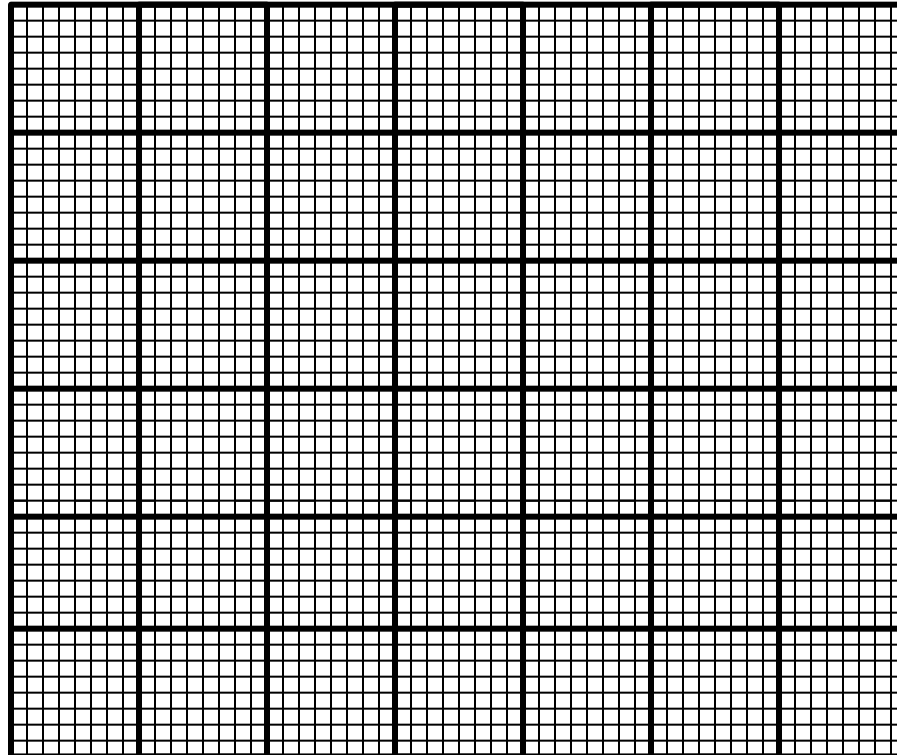
- ▶ This is the program for one thread
- ▶ It processes one pixel

Which thread computes which pixel?



gridDim

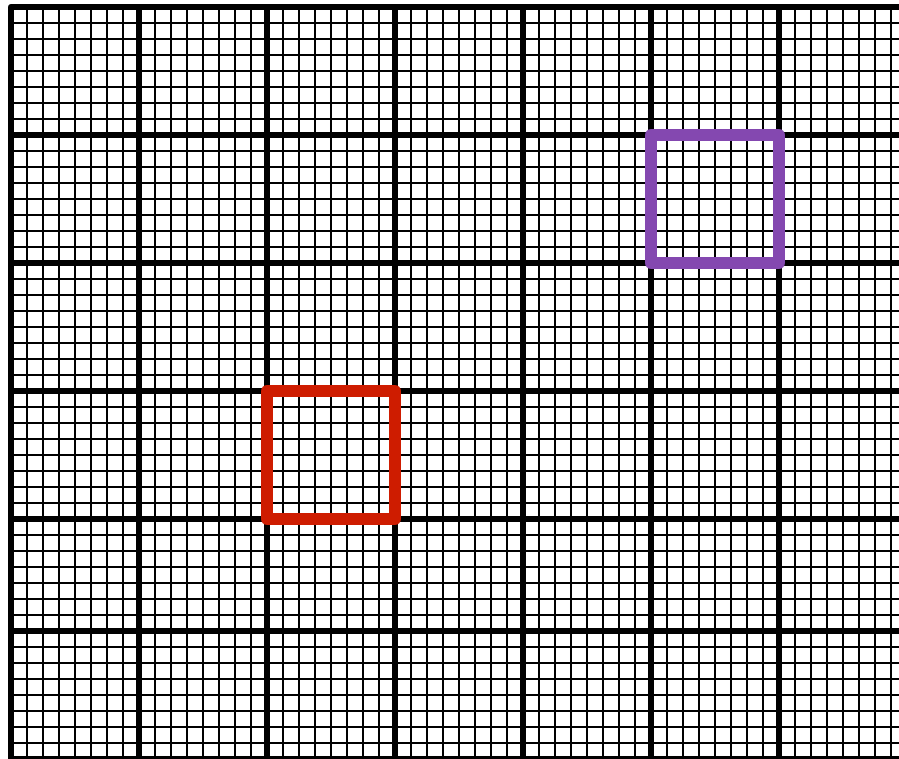
- ▶ $\text{gridDim.x} = 7$, $\text{gridDim.y} = 6$
- ▶ How many blocks per dimension?



blockIdx

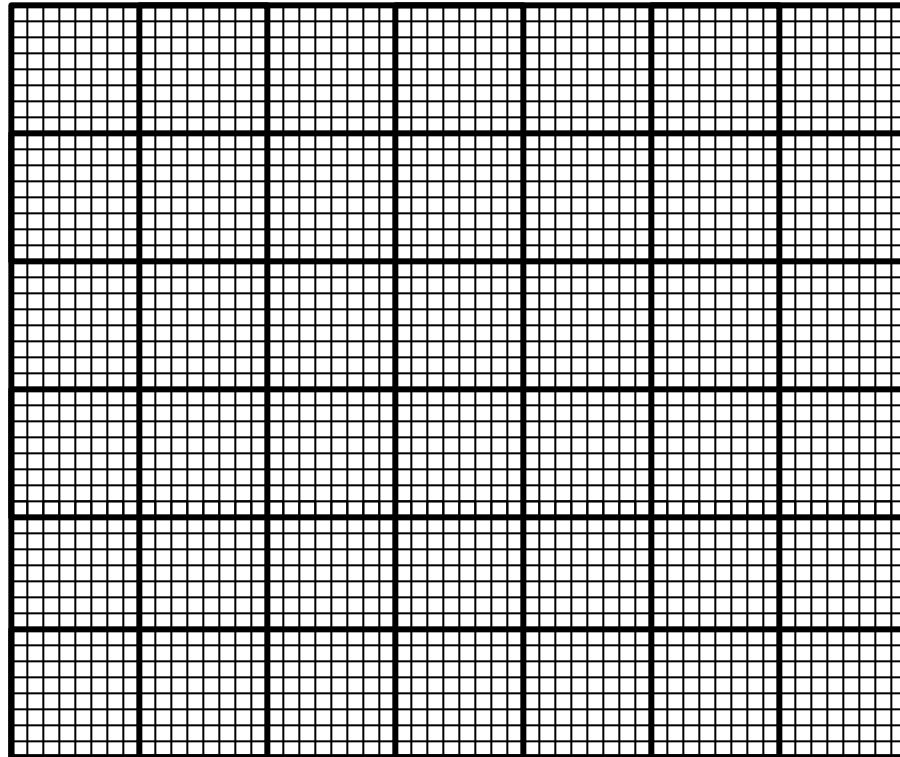
- ▶ blockIdx = coordinates of block in the grid
- ▶ $\text{blockIdx.x} = 2, \text{blockIdx.y} = 3$
- ▶ $\text{blockIdx.x} = 5, \text{blockIdx.y} = 1$

(0,0)



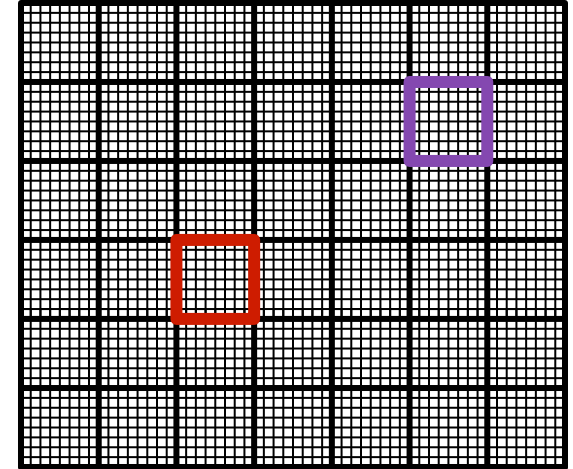
blockDim

- ▶ blockDim.x = 7, blockDim.y = 7
- ▶ How many threads in a block per dimension?

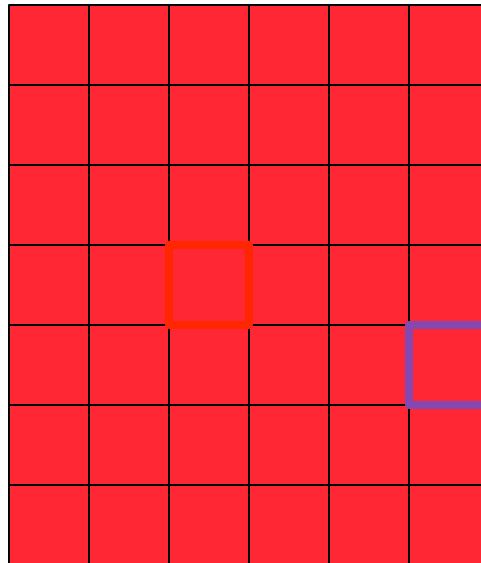


threadIdx

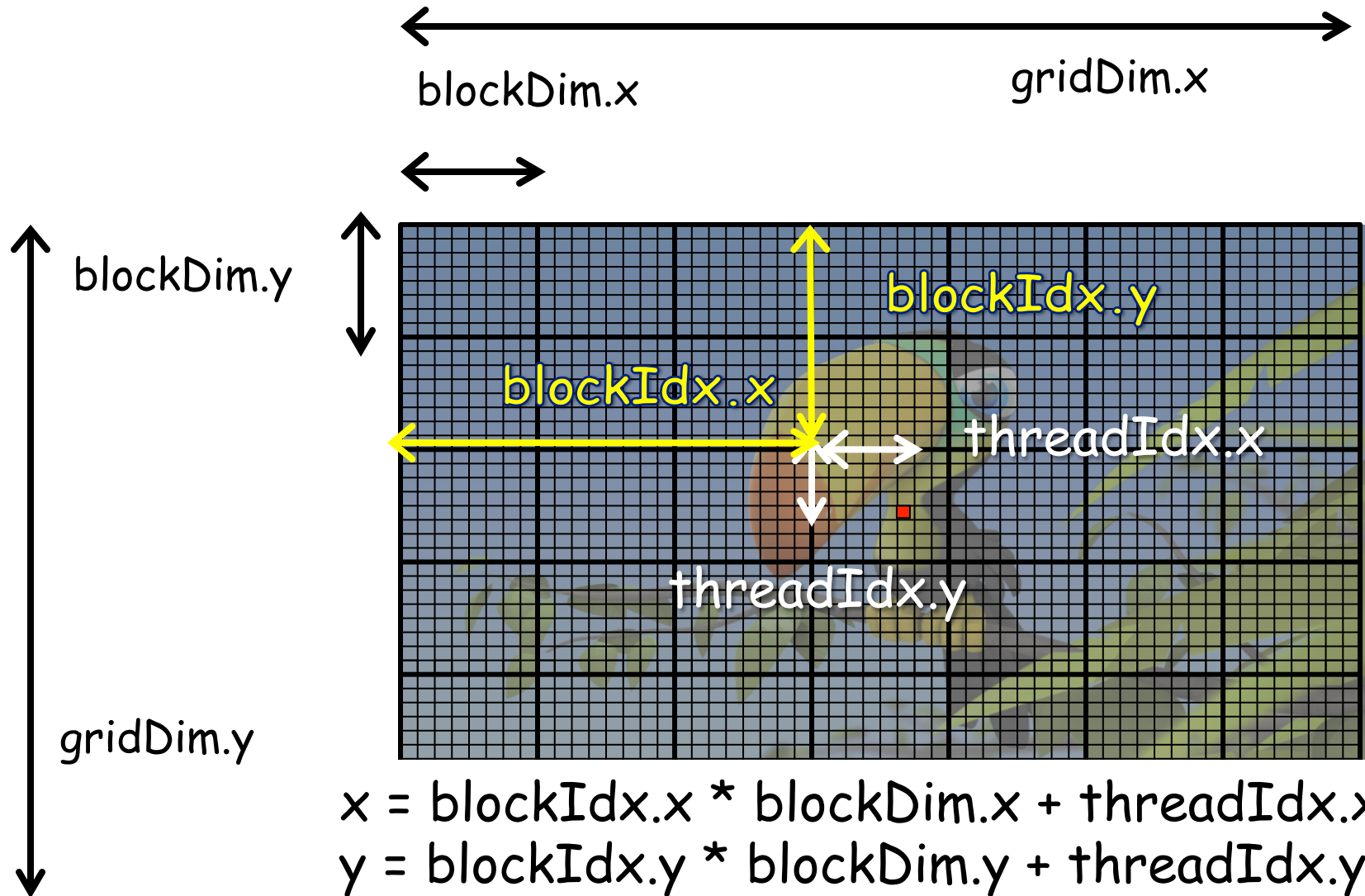
- ▶ threadIdx = coordinates of thread in the block
- ▶ threadIdx.x = 2, threadIdx.y = 3
- ▶ threadIdx.x = 5, threadIdx.y = 4



(0,0)



Which thread computes which pixel?



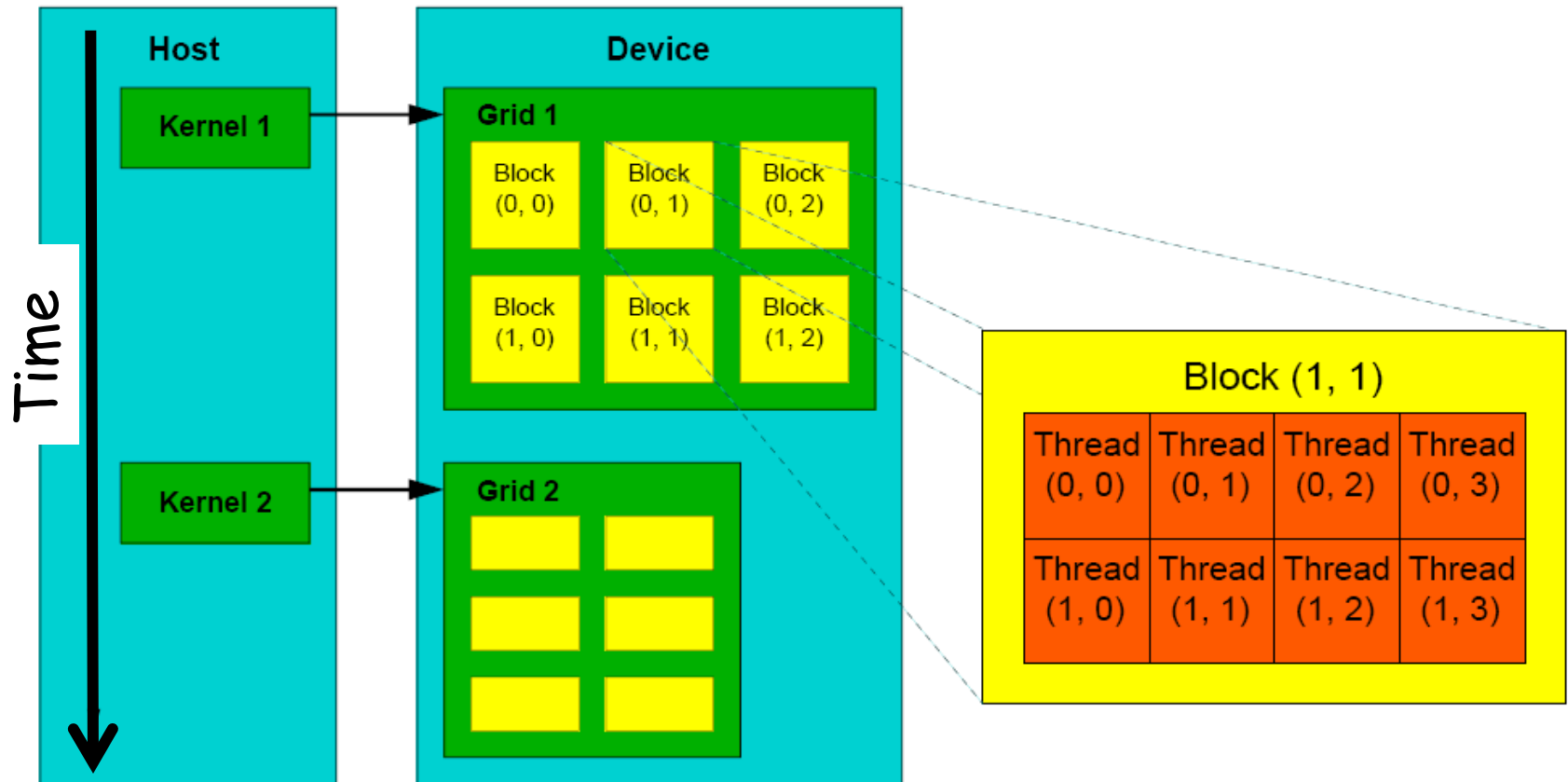
GPU Kernel pseudo-code

```
__global__ void fade (int *a,  
                    int fade,  
                    int N)  
{  
    int x = blockDim.x * blockIdx.x + threadIdx.x;  
    int y = blockDim.y * blockIdx.y + threadIdx.y  
    int offset = y * (blockDim.x * gridDim.x) + x;  
                // offset within unidimensional array  
  
    int v = a[offset];  
    v = v * fade;  
    a[offset] = v;  
}
```

GPU Kernel pseudo-code w/ limits

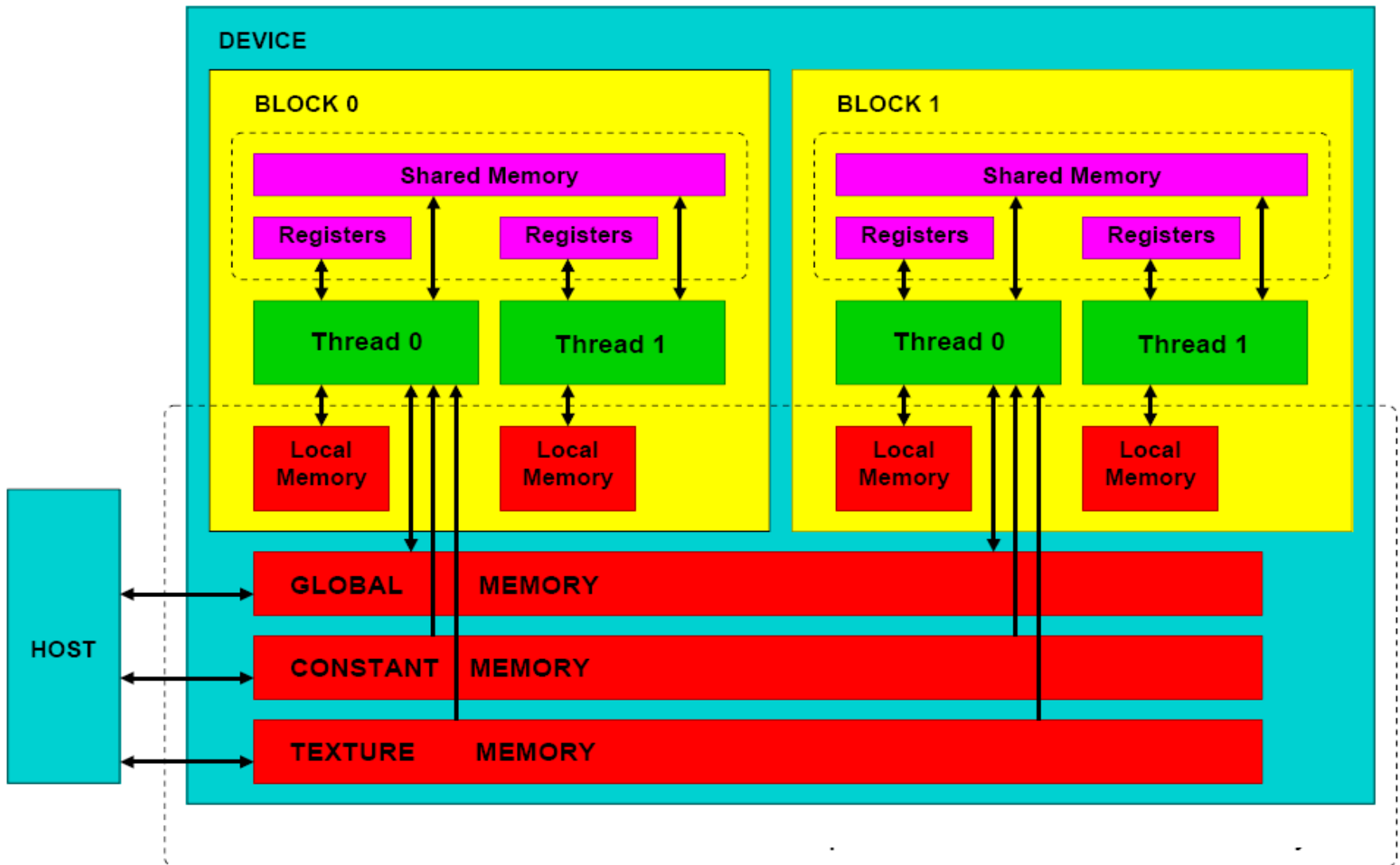
```
__global__ void fade (int *a,  
                    int fade,  
                    int N)  
{  
    int x = blockDim.x * blockIdx.x + threadIdx.x;  
    int y = blockDim.y * blockIdx.y + threadIdx.y  
    int offset = y * (blockDim.x * gridDim.x) + x;  
    if (offset > N) return;  
    int v = a[offset];  
    v = v * fade;  
    a[offset] = v;  
}
```

Grids of Blocks of Threads



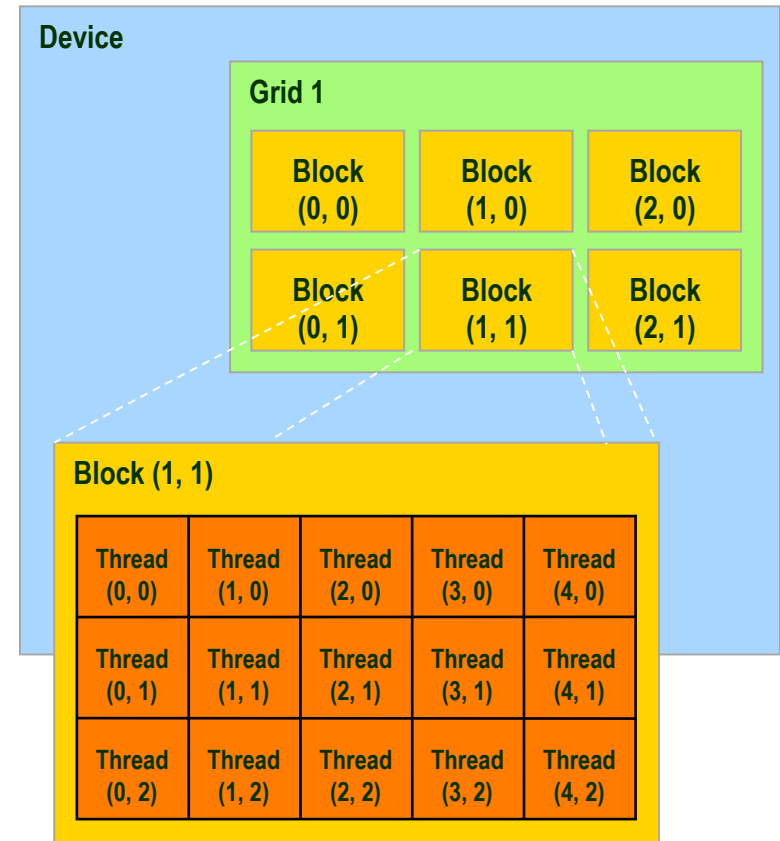
Cores and caches are clustered on chip for fast connectivity
Hardware partitioned naturally into grids

Programmer's view: Memory Model



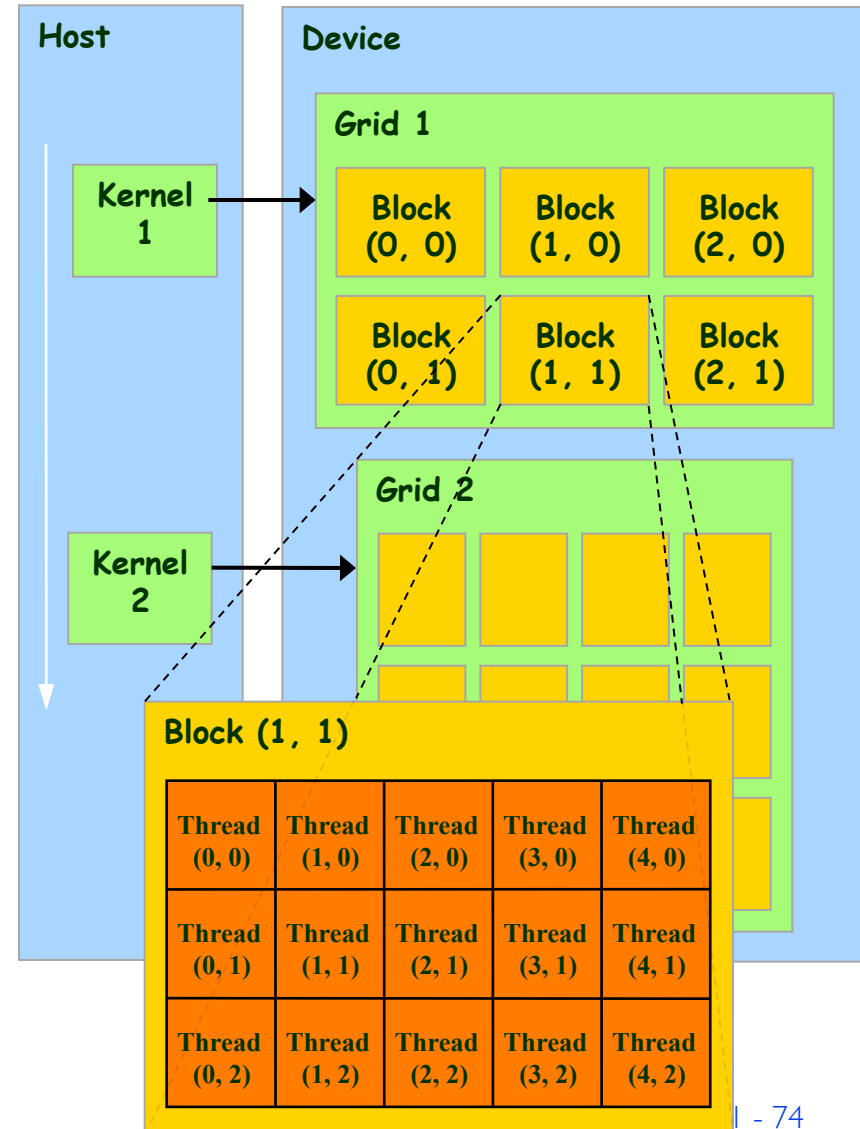
Grids of Thread Blocks: Dimension Limits

- ▶ Grid of Blocks **1D, 2D, or 3D**
 - ▷ Max x, y and z: $2^{32}-1$
 - ▷ Machine dependent
- ▶ Block of Threads: **1D, 2D, or 3D**
 - ▷ Max number of threads: 1024
 - ▷ Max x: 1024
 - ▷ Max y: 1024
 - ▷ Max z: 64



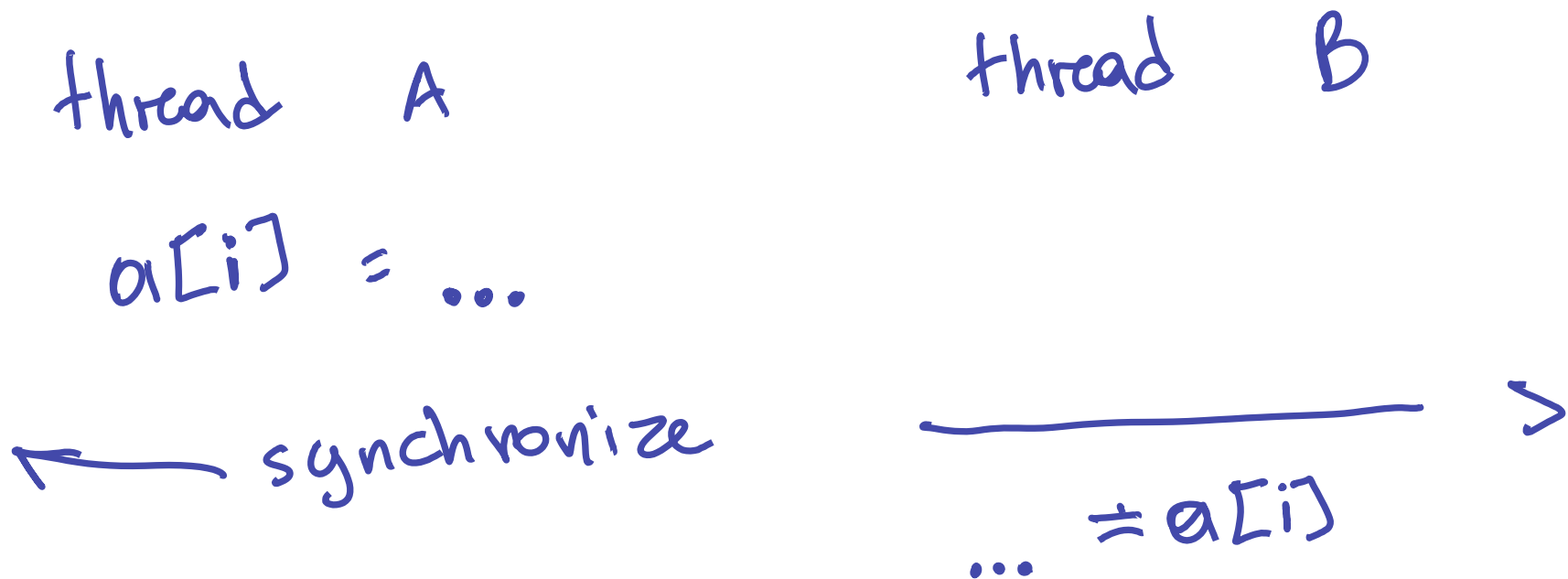
Thread Batching

- ▶ Kernel executed as a grid of thread blocks
- ▶ Threads in block cooperate
 - ▷ Synchronize their execution
 - ▷ Efficiently share data in block-local memory
- ▶ Threads across blocks cannot cooperate

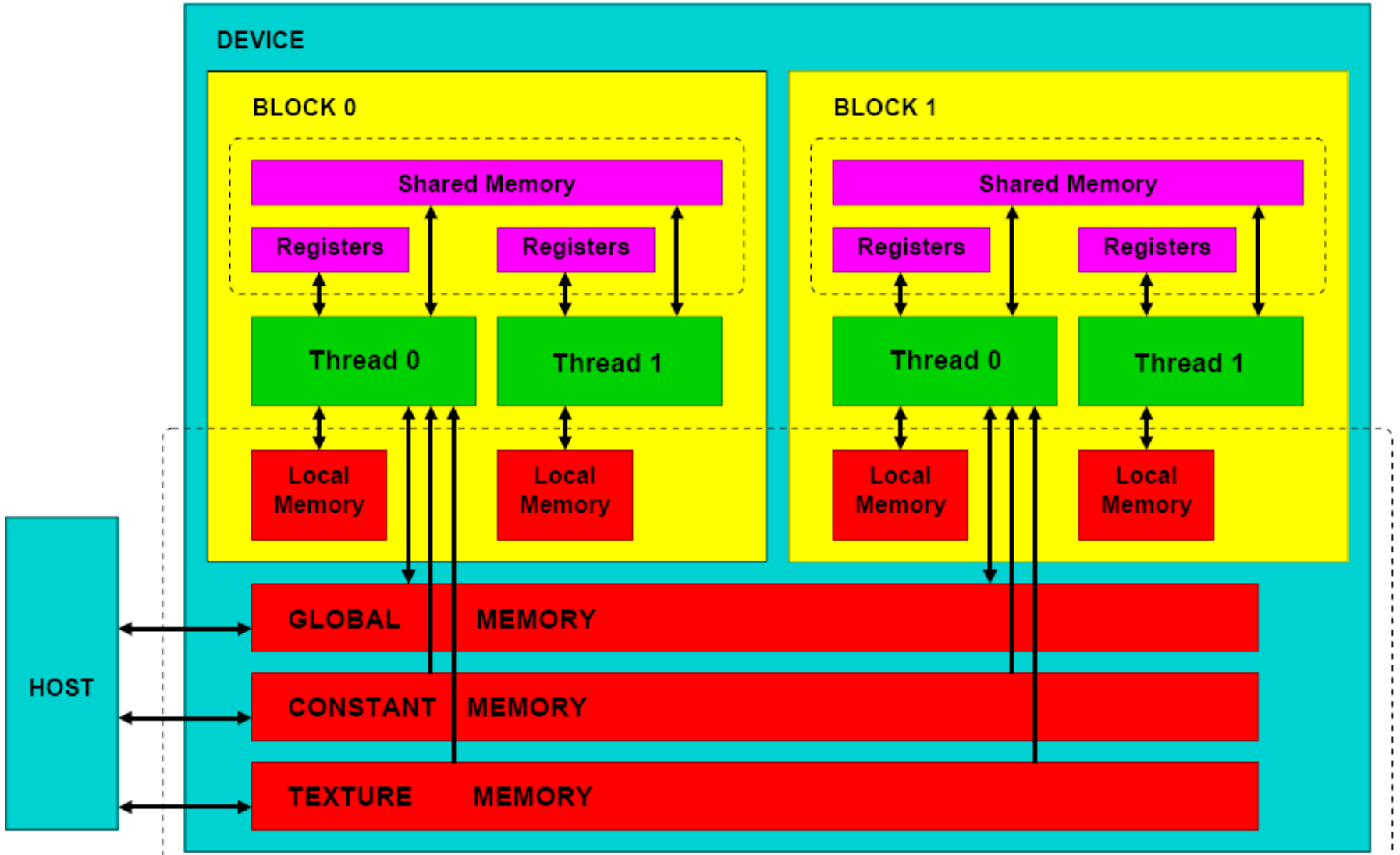


Thread Coordination Overview

- ▶ Race-free access to data



Only across threads within the same block
No communication across blocks



Arrows show whether read and/or write is possible

Memory Model Summary

Memory	Location	Access	Scope
Local	off-chip	R/W	thread
Shared	on-chip	R/W	all threads in a block
Global	off-chip	R/W	all threads + host
Constant	off-chip	RO	all threads + host
Texture	off-chip	RO	all threads + host
Surface	off-chip	R/W	all threads + host

Memory Model:

Global, Constant, and Texture Memories

► Global memory

- Communicating R/W data between host and device
- Contents visible to all threads
- May be cached (machine dependent)

► Texture and Constant Memories

- Constants initialized by host
- Contents visible to all threads
- May be cached (machine dependent)

Execution Model: Ordering

- ▶ Execution order is **undefined**
- ▶ Do not assume and use:
 - ▷ block 0 executes before block 1
 - ▷ thread 10 executes before thread 20
 - ▷ and **any** other ordering **even if you can observe it**
- ▶ Future implementations may break this ordering
- ▶ It's not part of the CUDA definition
- ▶ Why? More flexible hardware options

Reasoning about CUDA call ordering

- ▶ Access GPU via `cuda...()` calls and kernel invocations
 - ▷ `cudaMalloc`, `cudaMemcpy`

- ▶ Asynchronous from the CPU's perspective
 - ▷ CPU places a request in a “CUDA” queue
 - ▷ requests are handled in-order

Execution Model Summary (for your reference)

- ▶ Grid of blocks of threads
 - ▷ 1D/2D/3D grid of blocks of 1D/2D/3D threads
 - ▷ Threads and blocks have IDs
- ▶ Block execution order is undefined
- ▶ Same block threads can shared data fast
- ▶ Across blocks, threads:
 - ▷ Cannot cooperate
 - ▷ Communicate (slowly) through global memory
- ▶ Blocks do not migrate: execute on the same processor
- ▶ Several blocks may run over the same core

CUDA API: Example

```
int a[N];  
for (i =0; i < N; i++)  
    a[i] = a[i] + x;
```

1. Allocate CPU Data Structure
2. Initialize Data on CPU
3. Allocate GPU Data Structure
4. Copy Data from CPU to GPU
5. Define *Execution Configuration*
6. Run Kernel
7. CPU synchronizes with GPU
8. Copy Data from GPU to CPU
9. De-allocate GPU and CPU memory

I. Allocate CPU data structure

```
float *ha;
main (int argc, char *argv[])
{
    int N = atoi (argv[1]);
    ha = (float *) malloc (sizeof (float) * N);
    ...
}
```

2. Initialize CPU data (dummy)

```
float *ha;
```

```
int i;
```

```
for (i = 0; i < N; i++)
```

```
    ha[i] = i;
```

3. Allocate GPU data structure

```
float *da;
```

```
cudaMalloc ((void **) &da, sizeof (float) * N);
```

- ▶ Notice: no assignment side
 - ▷ NOT: `da = cudaMalloc (...)`
- ▶ Assignment is done internally:
 - ▷ That's why we pass `&da`
- ▶ Space is allocated in Global Memory on the GPU

GPU Memory Allocation

- ▶ The host manages GPU memory allocation:

- ▷ **cudaMalloc (void **ptr, size_t nbytes)**

- ▷ Must explicitly cast to (void **)

- ▷ `cudaMalloc ((void **) &da, sizeof (float) * N);`

- ▷ **cudaFree (void *ptr);**

- ▷ `cudaFree (da);`

- ▷ **cudaMemset (void *ptr, int value, size_t nbytes);**

- ▷ `cudaMemset (da, 0, N * sizeof (int));`

- ▶ Check the CUDA Reference Manual

4. Copy Initialized CPU data to GPU

```
float *da;
```

```
float *ha;
```

```
cudaMemCpy ((void *) da,           // DESTINATION  
            (void *) ha,           // SOURCE  
            sizeof (float) * N,    // #bytes  
            cudaMemcpyHostToDevice);  
                                                // DIRECTION
```

Host/Device Data Transfers

The host initiates all transfers:

- ▶ `cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction)`
- ▶ Asynchronous from the CPU's perspective
 - ▷ CPU thread continues
- ▶ In-order processing with other CUDA requests
- ▶ `enum cudaMemcpyKind`
 - ▷ `cudaMemcpyHostToDevice`
 - ▷ `cudaMemcpyDeviceToHost`
 - ▷ `cudaMemcpyDeviceToDevice`

5. Define Execution Configuration

- ▶ How many blocks and threads/block

```
int threads_block = 64;  
int blocks = N / threads_block;  
if (blocks % N != 0) blocks += 1;
```

- ▶ Alternatively:

```
blocks = (N + threads_block - 1) /  
         threads_block;
```

6. Launch Kernel & 7. CPU/GPU Synchronization

- ▶ GPU launch **blocks** **x** **threads_block** threads:

```
arradd <<<blocks, threads_block>>  
      (da, 10f, N);  
cudaDeviceSynchronize (); // forces CPU to wait
```

- ▶ arradd: kernel name
- ▶ <<<...>>> execution configuration
- ▶ (da, x, N): arguments
 - ▷ 256 byte limit / No variable arguments
 - ▷ Not sure this is still true

CPU/GPU Synchronization

- ▶ CPU does not block on `cuda...()` calls
 - ▷ Kernel/requests are queued and processed in-order
 - ▷ Control returns to CPU immediately
- ▶ Good if there is other work to be done
 - ▷ e.g., preparing for the next kernel invocation
- ▶ Eventually, CPU must know when GPU is done
- ▶ Then it can safely copy the GPU results
- ▶ **`cudaDeviceSynchronize ()`**
 - ▷ Block CPU until all preceding `cuda...()` and kernel requests have completed
 - ▷ Used to be `cudaThreadSynchronize ()`

8. Copy data from GPU to CPU & 9. Deallocate Memory

```
float *da;  
float *ha;  
  
cudaMemCpy ((void *) ha,          // DESTINATION  
            (void *) da,          // SOURCE  
            sizeof (float) * N,   // #bytes  
            cudaMemcpyDeviceToHost);  
                                            // DIRECTION  
  
cudaFree (da);  
// display or process results here  
free (ha);
```

The GPU Kernel

```
__global__ darradd (float *da, float x, int
N)
{
    int i = blockIdx.x * blockDim.x +
threadIdx.x;

    if (i < N) da[i] = da[i] + x;
}
```

CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- ▶ `__global__` defines a kernel function
 - ▷ Must return void
 - ▷ Can only call `__device__` functions
- ▶ `__device__` and `__host__` can be used together
 - ▷ Two difference versions generated

Can you do this one now?

$$(\mathbf{C}) = (\mathbf{A}) \cdot (\mathbf{B})$$

Summary

▶ Data Parallel Computing

- ▷ Much of media processing is data parallel
- ▷ All of data analytics on datacenters & beyond

▶ Platforms for data parallel computing

- ▷ Within CPU: SIMD/Vector
- ▷ Across CPU: GPU
- ▷ Across a single computer: cluster of servers

▶ GPUs: orders of magnitude more concurrent than CPU

▶ GPU programming

- ▷ It's complicated
- ▷ Take your time