ACACES 2018 Summer School

GPU Architectures: Basic to Advanced Concepts

Adwait Jog, Assistant Professor

College of William & Mary
(http://adwaitjog.github.io/)

Course Outline

- ❑ Lectures 1 and 2: Basics Concepts
  - Basics of GPU Programming
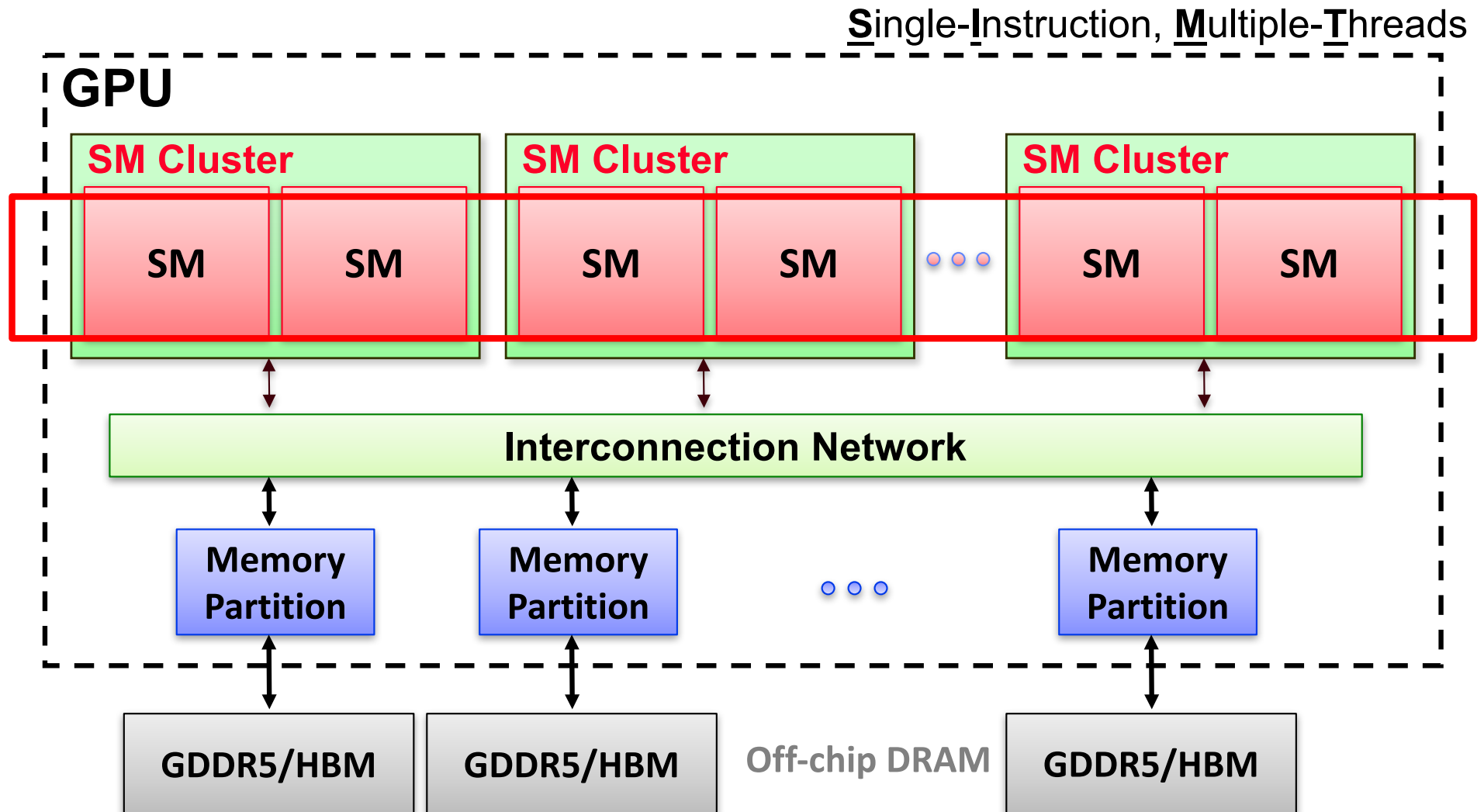 - **Basics of GPU Architecture**

- ❑ Lecture 3: GPU Performance Bottlenecks
  - Memory Bottlenecks
  - Compute Bottlenecks
  - Possible Software and Hardware Solutions

- ❑ Lecture 4: GPU Security Concerns
  - Timing channels
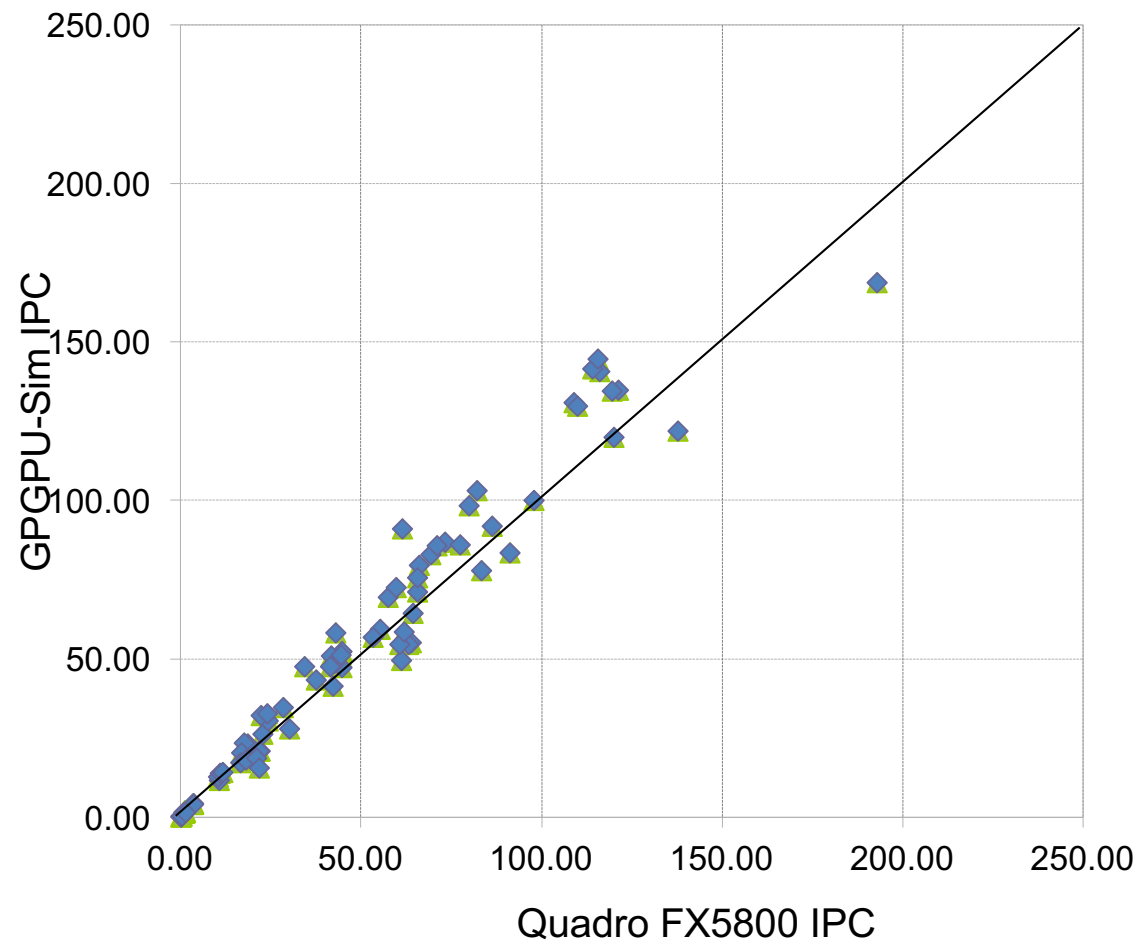 - Possible Software and Hardware Solutions

GPU Architecture Overview

GPU Microarchitecture

❑ Not many details are publicly available about GPU microarchitecture.

❑ Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.

GPGPU-Sim from UBC – A Cycle-level Simulator

HW - GPGPU-Sim Comparison

Correlation ~0.976

GPU Instruction Set Architecture (ISA)

❑ NVIDIA defines a <u>virtual ISA</u>, called "PTX" (Parallel Thread eXecution)

❑ More recently, Heterogeneous System Architecture (HSA) Foundation (AMD, ARM, Imagination, Mediatek, Samsung, Qualcomm, TI) defined the HSAIL virtual ISA.

❑ PTX is Reduced Instruction Set Architecture (e.g., load/store architecture)

❑ Virtual: infinite set of registers (much like a compiler intermediate representation)

❑ PTX translated to hardware ISA by backend compiler ("ptxas"). Either at compile time (nvcc) or at runtime (GPU driver).

Some Example PTX Syntax

- Registers declared with a type:

  ```
  .reg .pred  p, q, r;

  .reg .u16   r1, r2;

  .reg .f64   f1, f2;
  ```

- ALU operations

  ```
  add.u32 x, y, z;        // x = y + z

  mad.lo.s32 d, a, b, c; // d = a*b + c
  ```

- Memory operations:

  ```
  ld.global.f32 f, [a];

  ld.shared.u32 g, [b];

  st.local.f64  [c], h
  ```
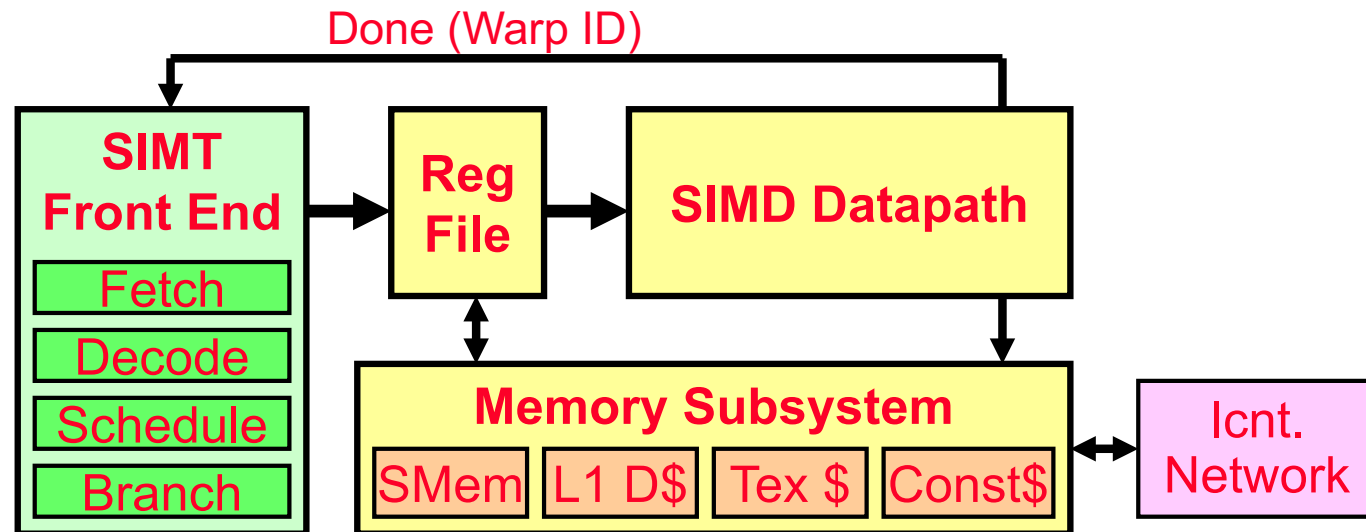
- Compare and branch operations:

  ```
  setp.eq.f32 p, y, 0;  // is y equal to zero?

  @p bra L1  // branch to L1 if y equal to zero
  ```
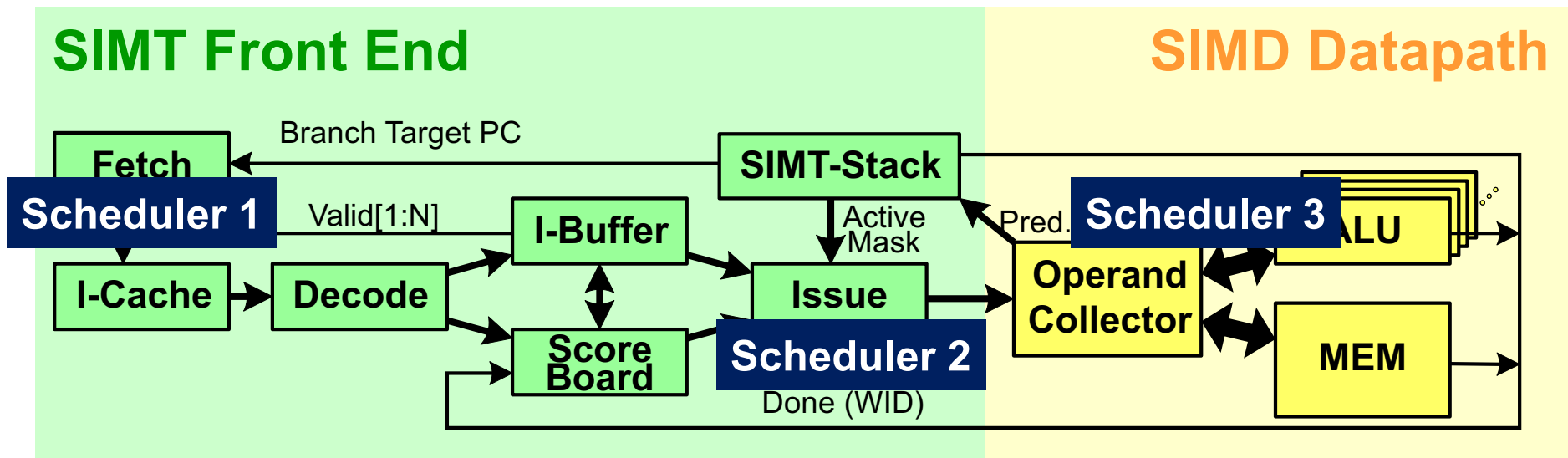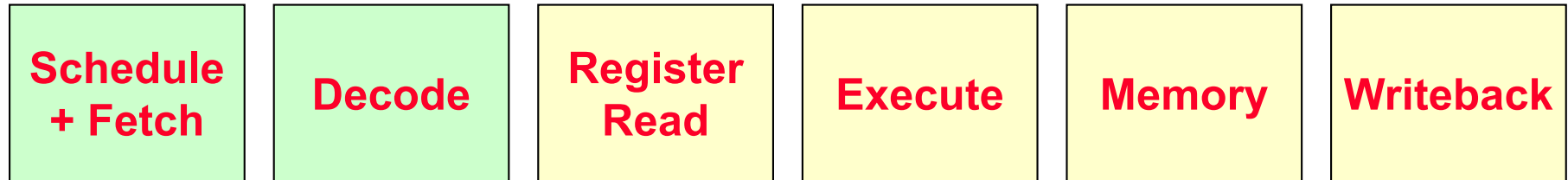
Inside an SM (1)

❑ Fine-grained multithreading

- Interleave warp execution to hide latency
- Register values of all threads stays in core

Inside an SM (2)

Schedule + Fetch	Decode	Register Read	Execute	Memory	Writeback

SIMT Front End

SIMD Datapath

Branch Target PC

Fetch

Scheduler 1

Valid[1:N]

I-Cache → Decode

I-Buffer

SIMT-Stack

Active Mask

Pred.

Scheduler 3

Score Board

Issue

Scheduler 2

Operand Collector

ALU

MEM

Done (WID)

- ❑ Three decoupled warp schedulers

- ❑ Scoreboard

- ❑ Large register file

- ❑ Multiple SIMD functional units

Fetch + Decode

❑ Arbitrate the I-cache among warps

  ● Cache miss handled by fetching again later

❑ Fetched instruction is decoded and then stored in the I-Buffer

  ● 1 or more entries / warp

 ● Only warp with vacant entries are considered in fetch
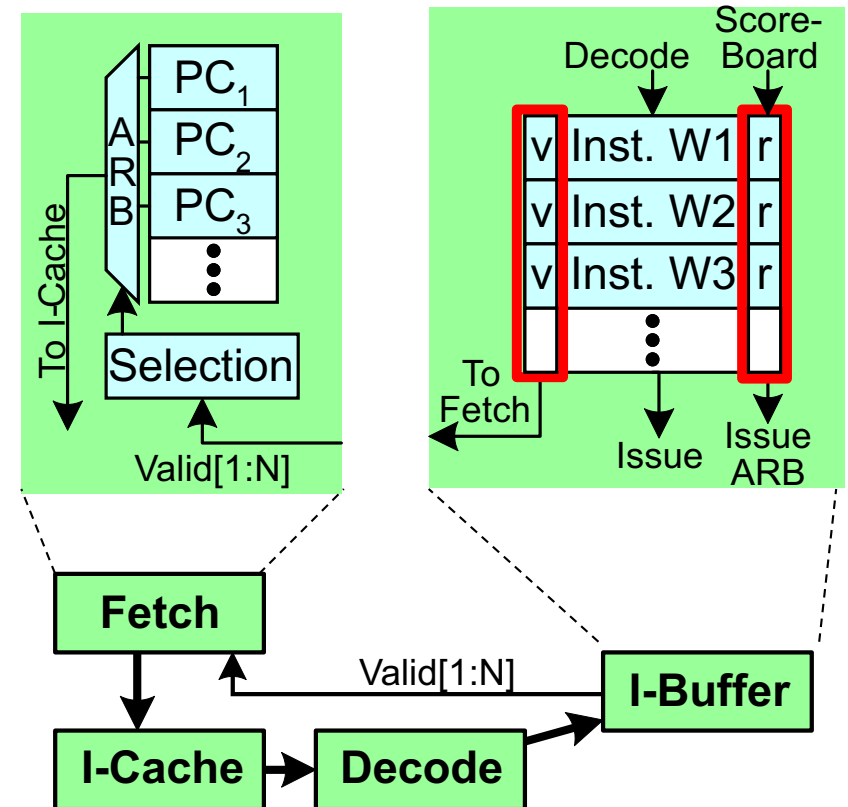
Instruction Issue

❑ Select a warp and issue an instruction from its I-Buffer for execution

- Scheduling: Greedy-Then-Oldest (GTO)
- GT200/later Fermi/Kepler: Allow dual issue (superscalar)
- Fermi: Odd/Even scheduler
- To avoid stalling pipeline might

  keep instruction in I-buffer until

  know it can complete (replay)
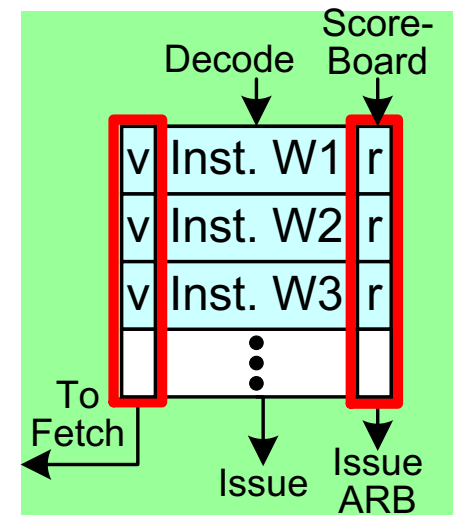
Scoreboard

- Checks for RAW and WAW dependency hazard
  - Flag instructions with hazards as *not ready* in I-Buffer (masking them out from the scheduler)

- Instructions reserves registers at issue
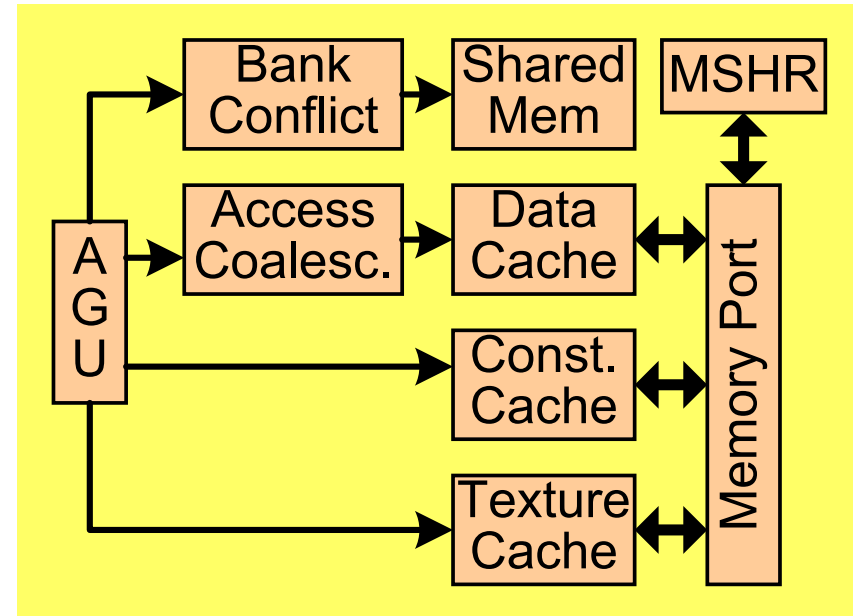
- Release them at writeback

Operand Collector
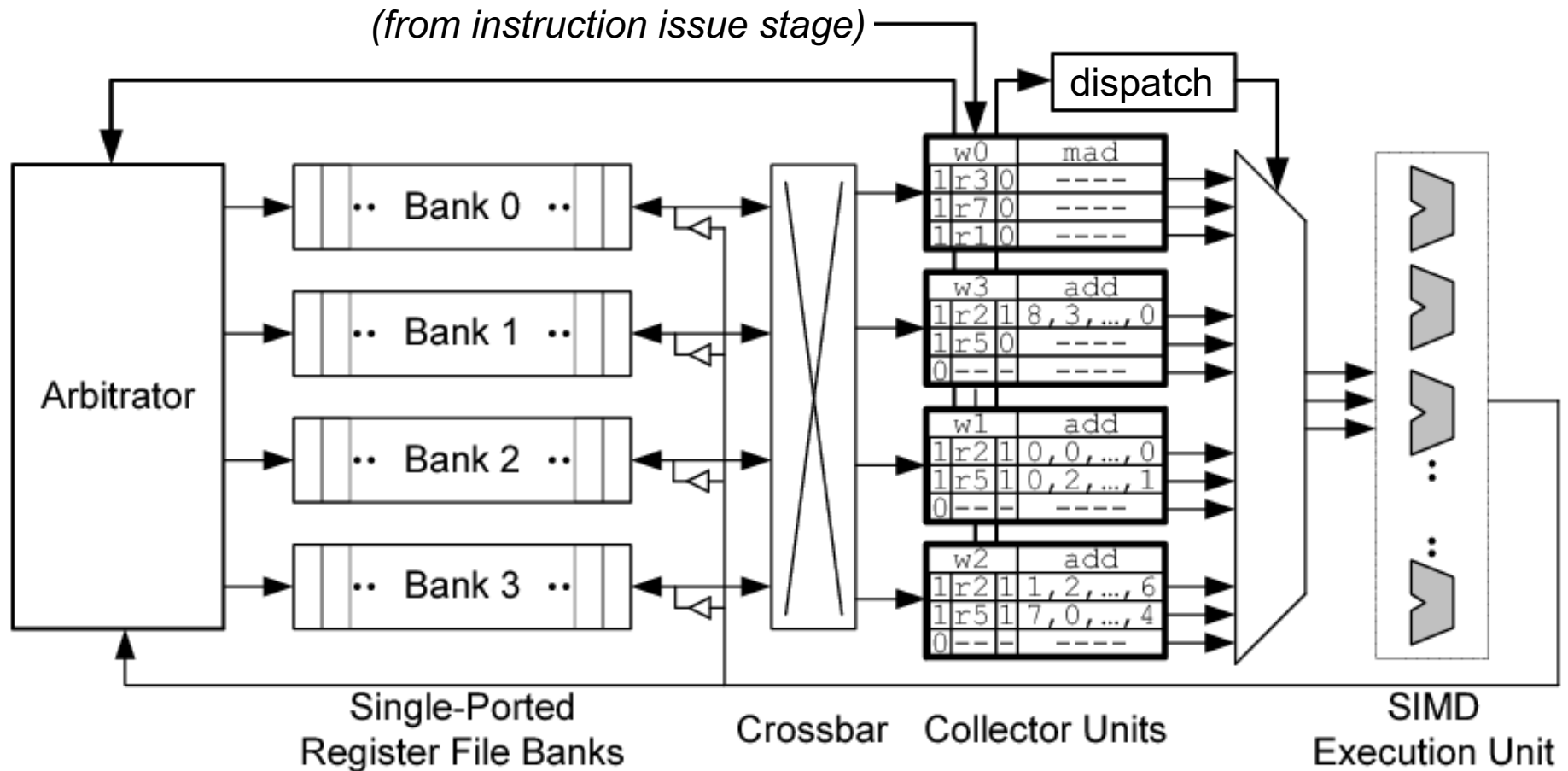
ALU Pipelines

- SIMD Execution Unit

- Fully Pipelined

- Each pipe may execute a subset of instructions

- Configurable bandwidth and latency (depending on the instruction)

- Default: SP + SFU pipes

Memory Unit

❑ **Model timing for memory instructions**

❑ **Support half-warp (16 threads)**

  ● Double clock the unit

  ● Each cycle service half the warp
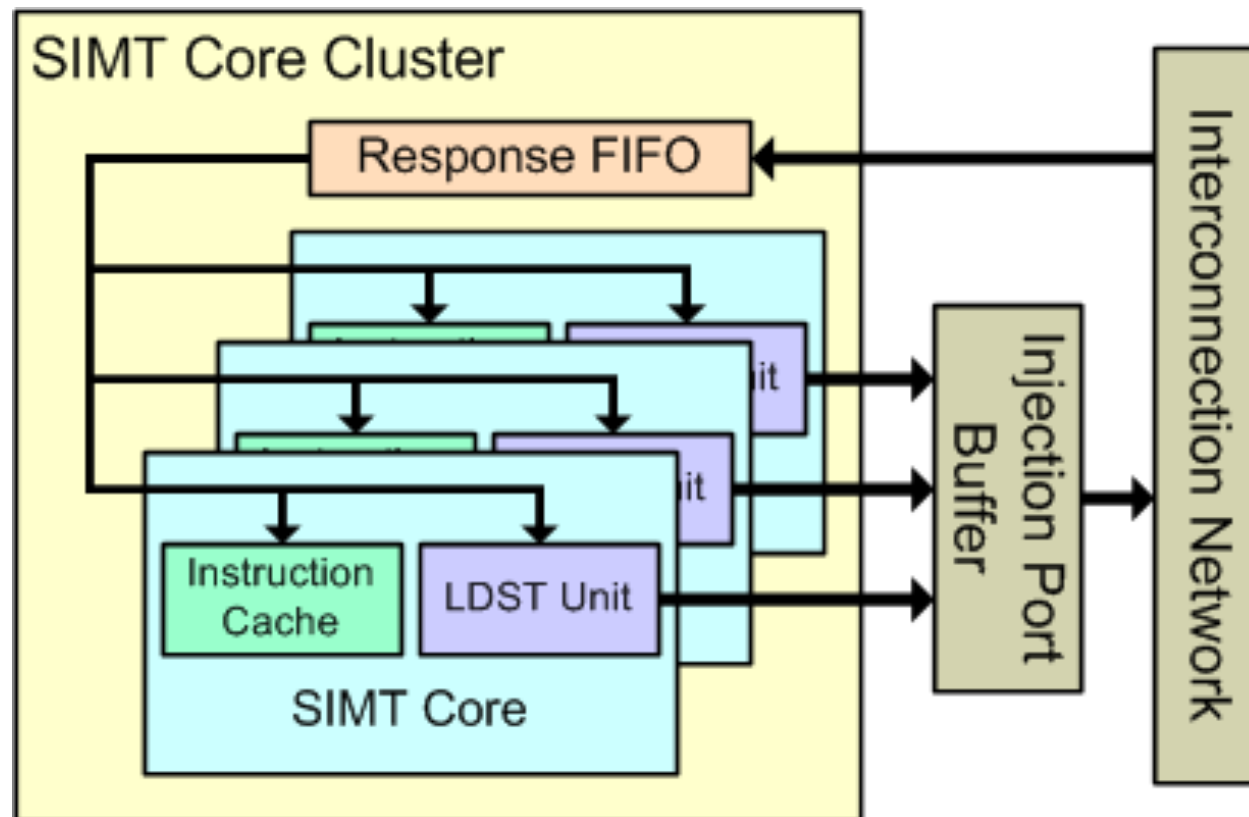
❑ **Has a private writeback path**

	Bank Conflict	→	Shared Mem	MSHR
AGU	Access Coalesc.	→	Data Cache	Memory Port
			Const. Cache	
			Texture Cache	

Writeback

❑ Each pipeline has a result bus for writeback

❑ Exception:

- SP and SFU pipe shares a result bus
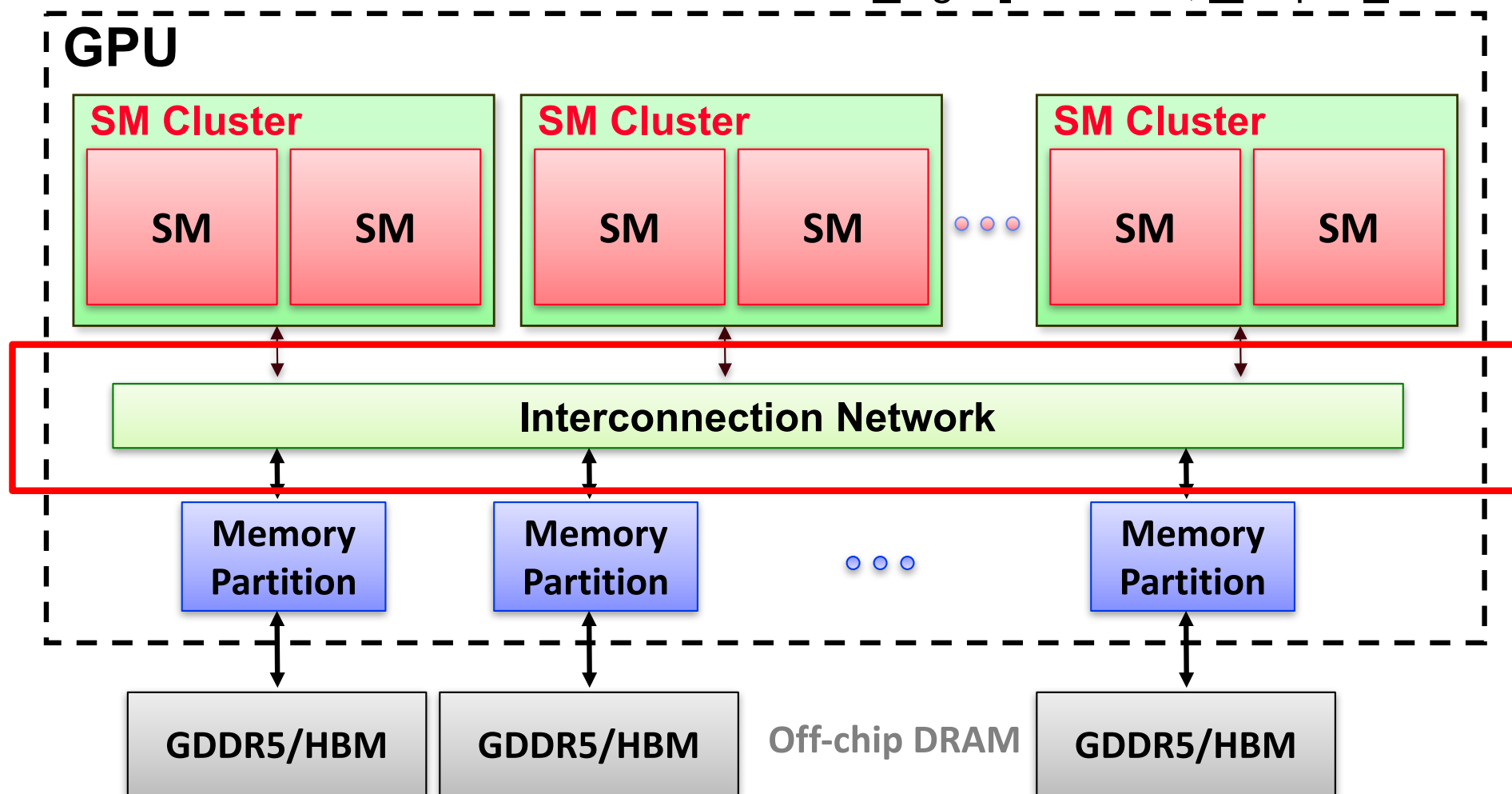- Time slots on the shared bus is pre-allocated

SM Cluster

❑ Collection of SIMT cores

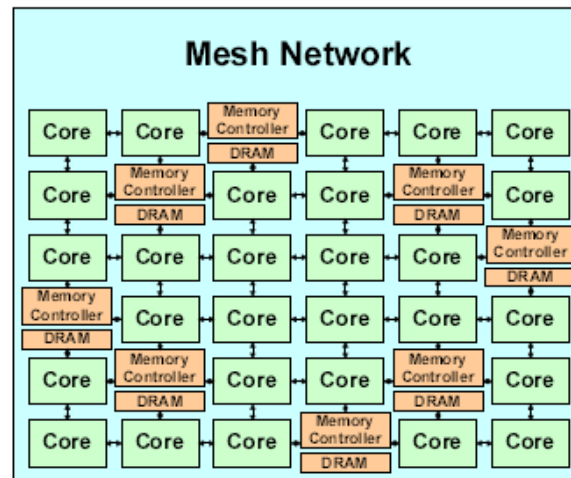# GPU Architecture Overview

Interconnection Network Model

❑ **Intersim (Booksim) a flit level simulator**

- Topologies (Mesh, Torus, Butterfly, …)
- Routing (Dimension Order, Adaptive, etc.)
- Flow Control (Virtual Channels, Credits)

❑ **Two separate networks**

- From SIMT cores to memory partitions
  - Read Requests, Write Requests
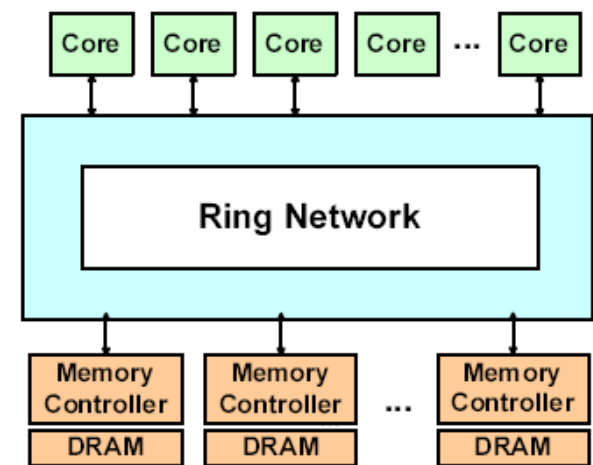- From memory partitions to SIMT cores
  - Read Replies, Write Acks
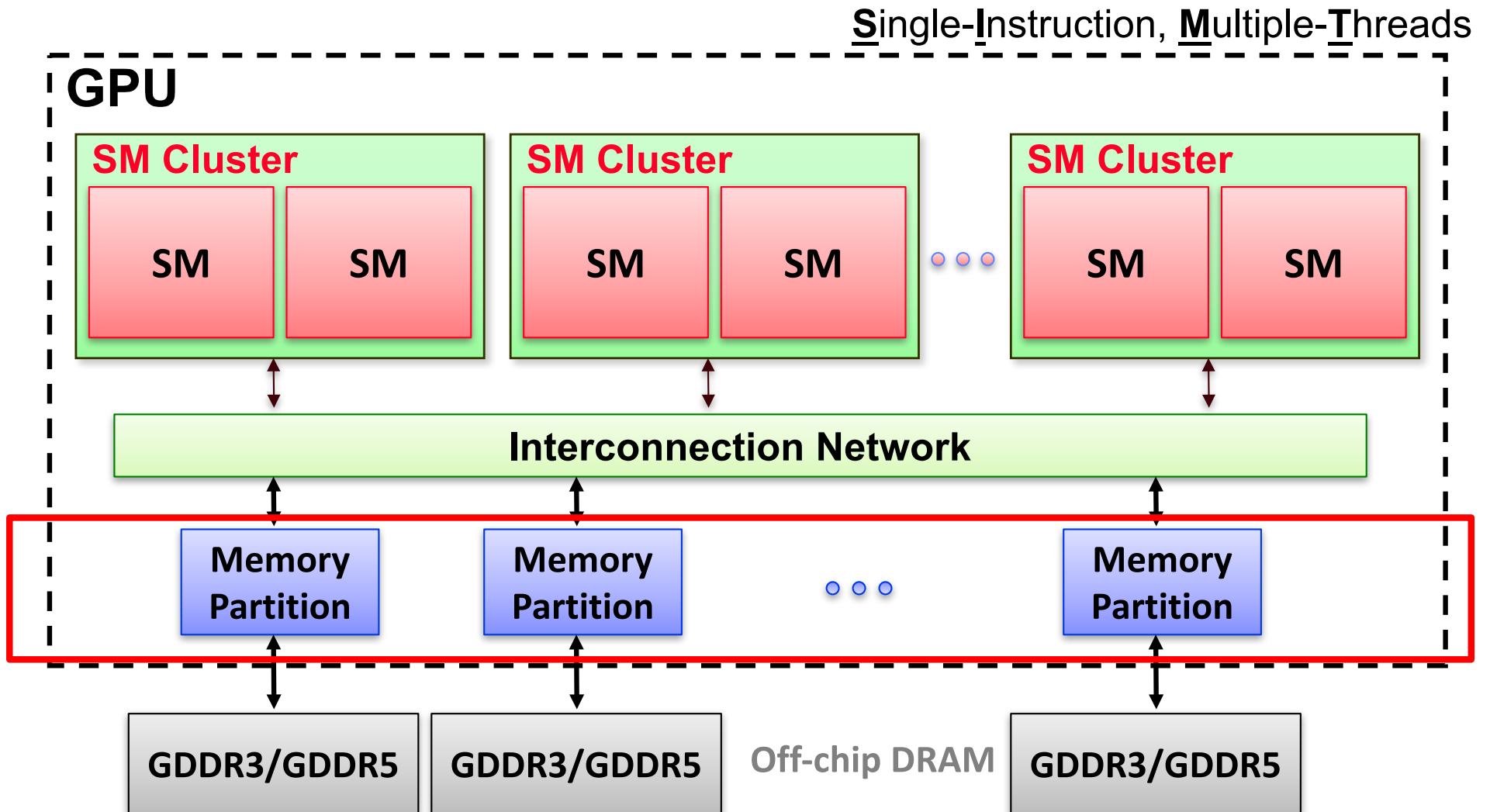
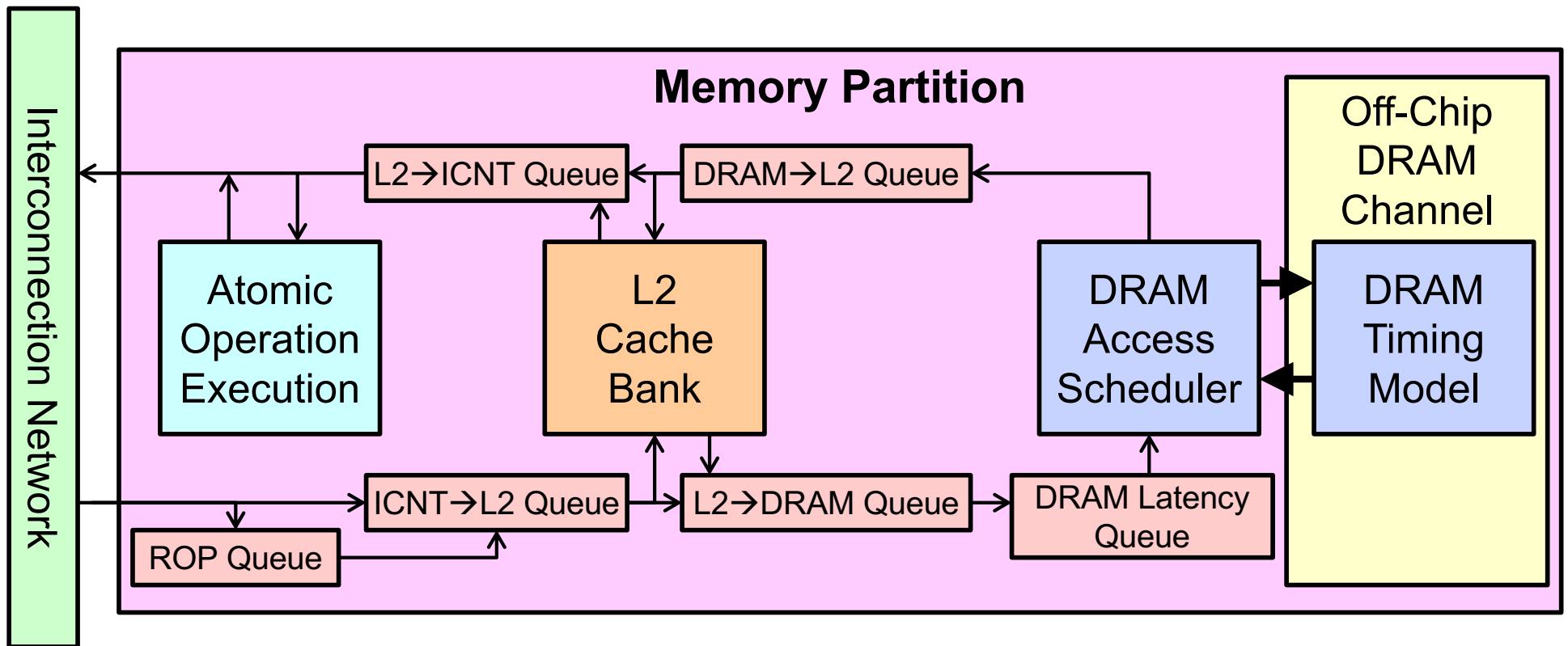# Topology Examples
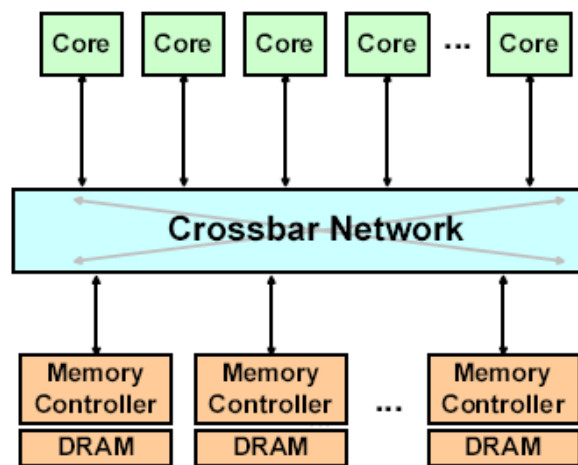
(a) Crossbar

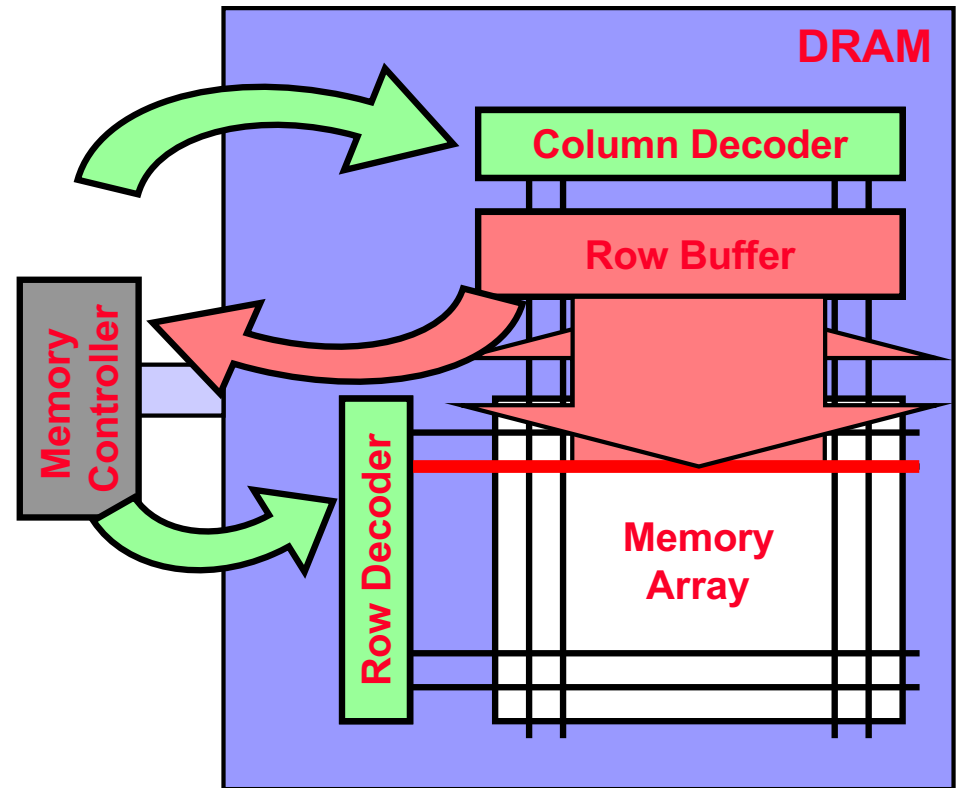(b) Mesh

(c) Ring

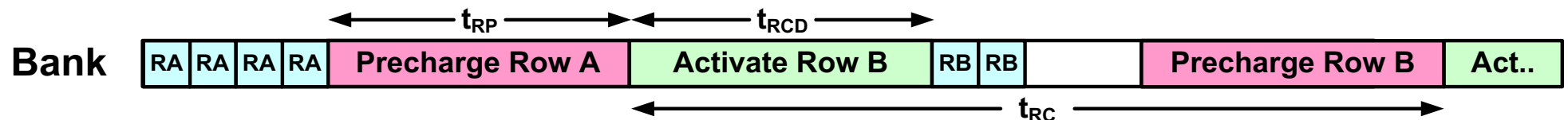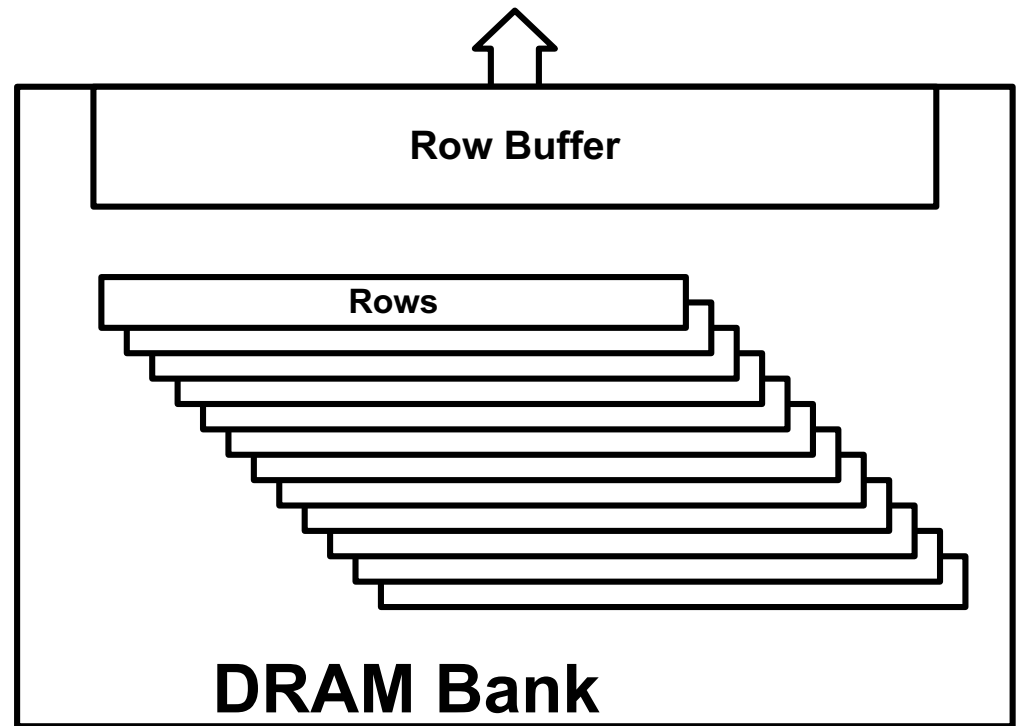# GPU Architecture Overview

Memory Partition

DRAM Access

- Row access
  - Activate a row or page of a DRAM bank
  - Load it to row buffer

- Column access
  - Select and return a block of data in row buffer

- Precharge
  - Write back the opened row into DRAM
  - Otherwise it will be lost!

DRAM

Column Decoder

Row Buffer

Memory Controller

Row Decoder

Memory Array

DRAM Row Access Locality

t_{RC} = row cycle time

t_{RP} = row precharge time

t_{RCD} = row activate time

Row Buffer

Rows

DRAM Bank

| Bank | RA | RA | RA | RA | Precharge Row A | Activate Row B | RB | RB | | Precharge Row B | Act.. |

DRAM Bank-level Parallelism

- To increase DRAM performance and utilization
  - Multiple banks per DRAM chip
- To increase bus width
  - Multiple chips per Memory Controller
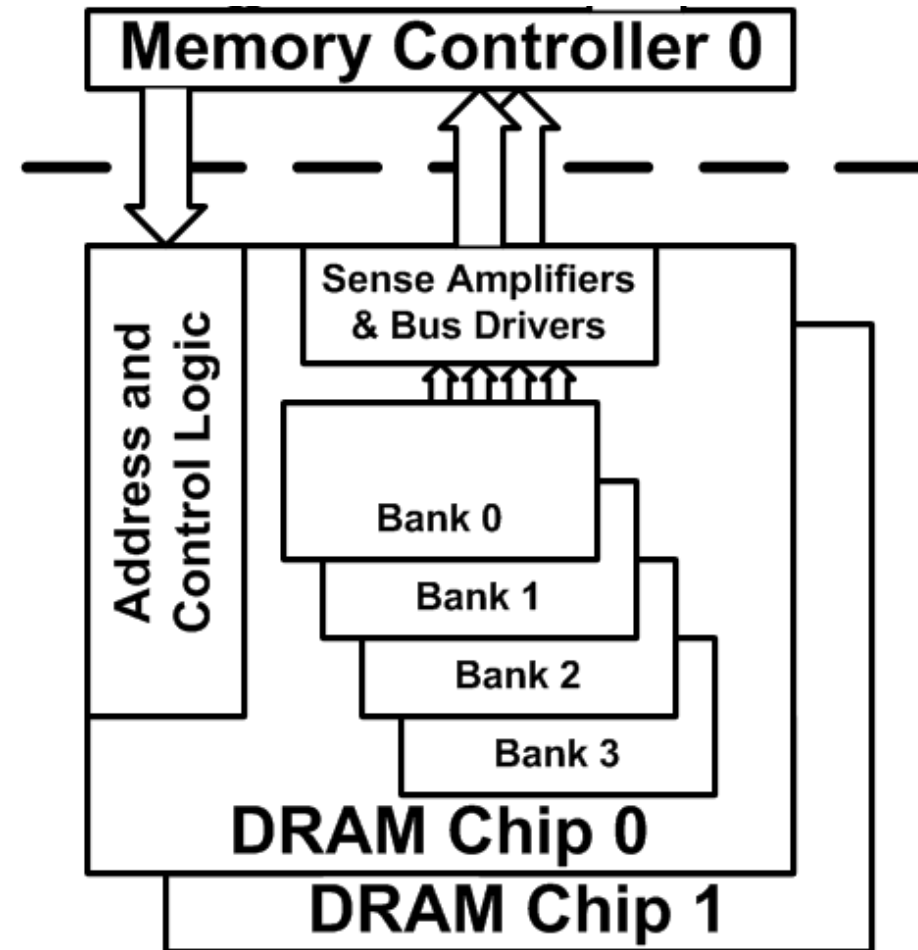
Scheduling DRAM Requests

- Scheduling policies supported
  - First in first out (FIFO)
    - In-order scheduling
  - First Ready First Come First Serve (FR-FCFS)
    - Out of order scheduling
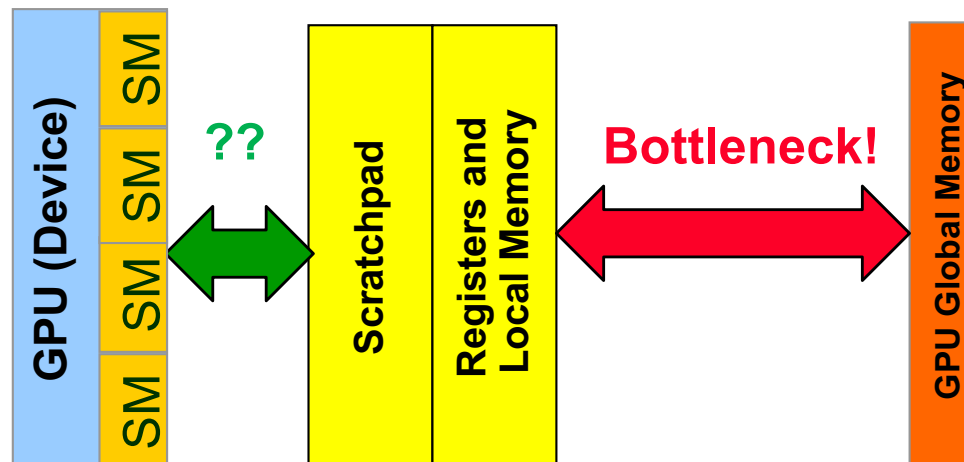   - Requires associative search
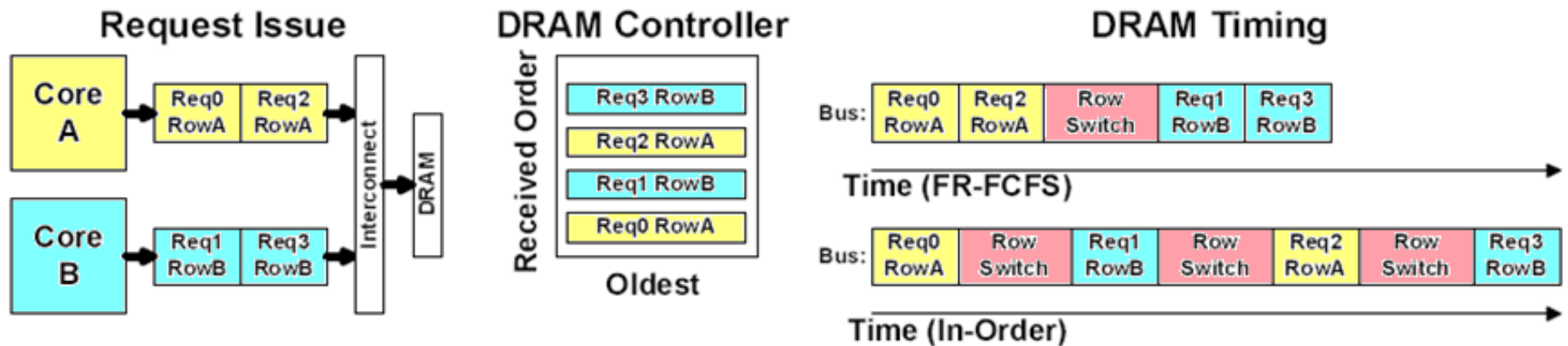
Key GPU Performance Concerns

Key GPU Performance Concerns

- I) Data transfers between CPU and GPU are one of the major performance bottlenecks.

Control	ALU	ALU
	ALU	ALU

Cache

CPU Memory

Bottleneck!

GPU Memory

- II) Data transfers between SMs and global memory is costly. Can on-chip memory help?

GPU (Device) — SM SM SM SM SM SM SM

??

Scratchpad | Registers and Local Memory
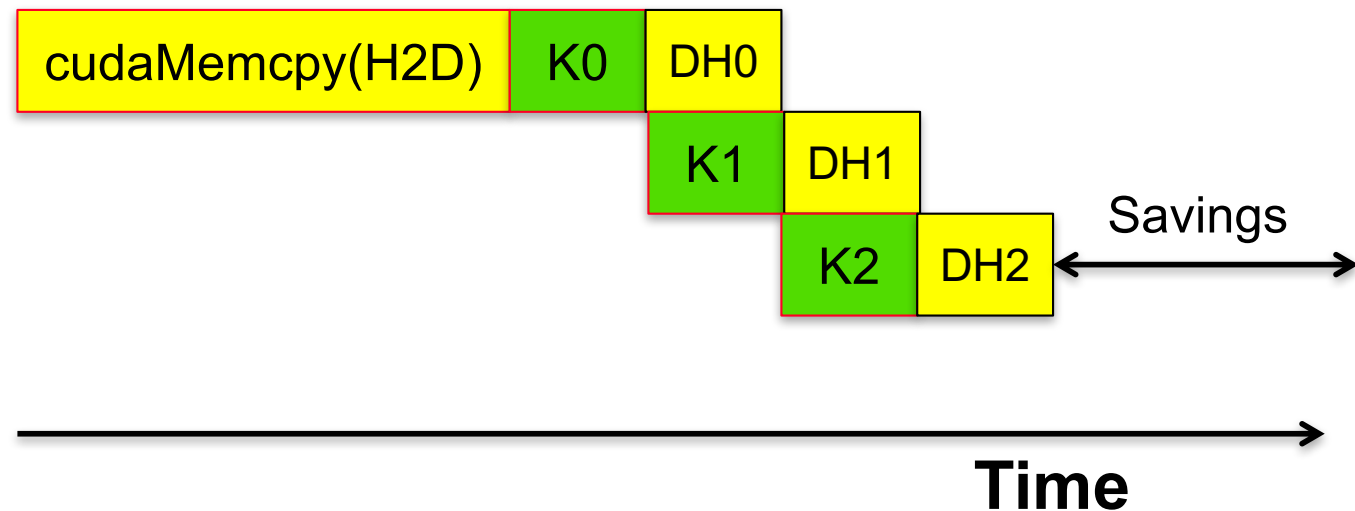
Bottleneck!

GPU Global Memory

CUDA Streams

❑ CUDA (and OpenCL) provide the capability to overlap computation on GPU with memory transfers using "Streams" (Command Queues)

❑ A Stream orders a sequence of kernels and memory copy "operations".

❑ Operations in one stream can overlap with operations in a different stream.

How Can Streams Help?

Serial:

GPU idle | GPU busy | GPU idle

| cudaMemcpy(H2D) | kernel<<<>>> | cudaMemcpy(D2H) |

Streams:

| cudaMemcpy(H2D) | K0 | DH0 |

K1 DH1

K2 DH2

Savings

Time

CUDA Streams

```
cudaStream_t streams[3];

for(i=0; i<3; i++)

    cudaStreamCreate(&streams[i]);  // initialize streams


for(i=0; i<3; i++) {

    cudaMemcpyAsync(pD+i*size,pH+i*size,size,

    cudaMemcpyHostToDevice,stream[i]); // H2D


    MyKernel<<<grid,block,0,stream[i]>>>(pD+i,size); // compute


    cudaMemcpyAsync(pD+i*size,pH+i*size,size,

    cudaMemcpyDeviceToHost,stream[i]);       // D2H
}
```

Manual CPU ⇔ GPU Data Movement

❑ **Problem #1:** Programmer needs to identify data needed in a kernel and insert calls to move it to GPU

❑ **Problem #2:** Pointer on CPU does not work on GPU since different address spaces

❑ **Problem #3:** Bandwidth connecting CPU and GPU is order of magnitude smaller than GPU off-chip

❑ **Problem #4:** Latency to transfer data from CPU to GPU is order of magnitude higher than GPU off-chip

❑ **Problem #5:** Size of GPU DRAM memory much smaller than size of CPU main memory

Additional Features in CUDA

❑ Dynamic Parallelism (CUDA 5 onwards): Launch kernels from within a kernel.   Reduce work for e.g., adaptive mesh refinement.

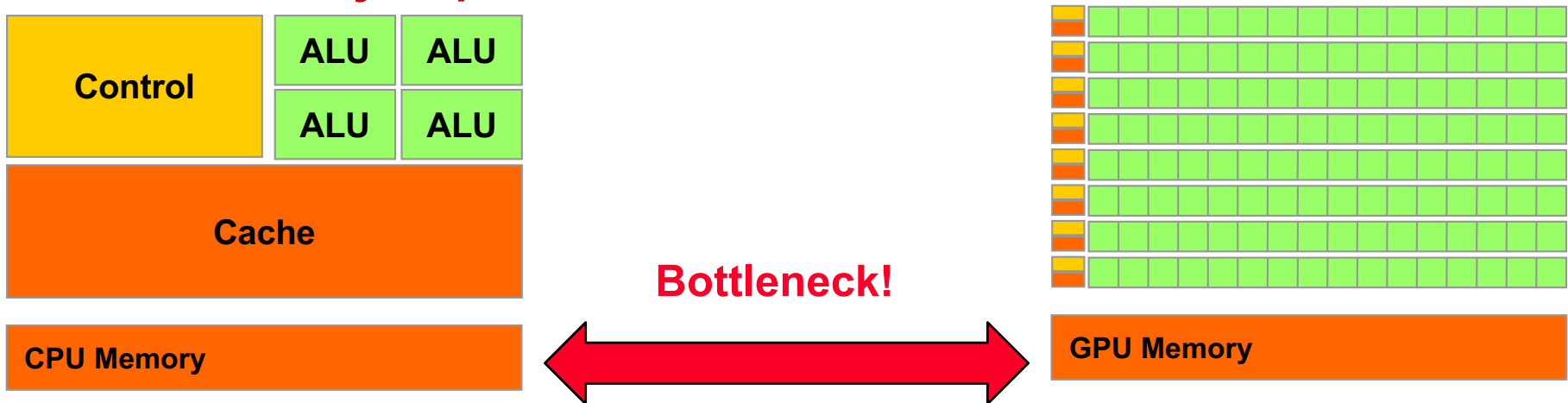❑ Unified Memory (CUDA 6 onwards): Avoid need for explicit memory copies between CPU and GPU

http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
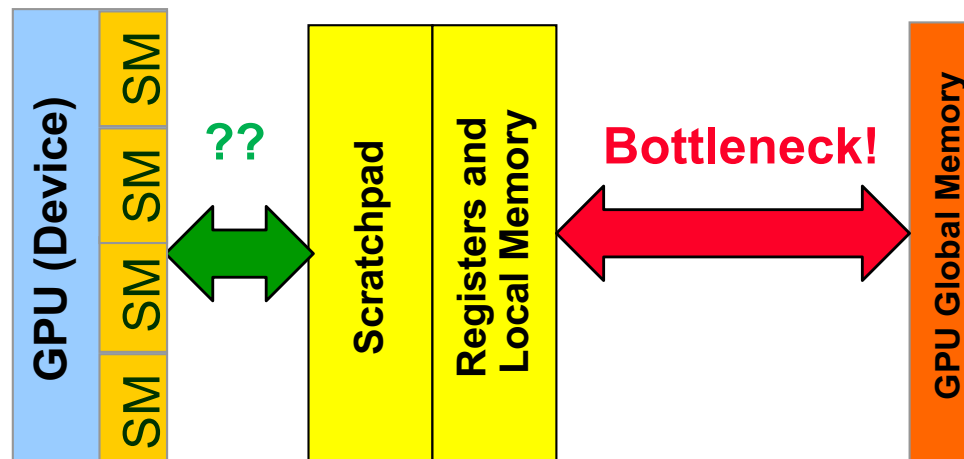
See also, Gelado, et al. ASPLOS 2010.

Key GPU Performance Concerns

– I) Data transfers between CPU and GPU are one of the major performance bottlenecks.

| Control | ALU | ALU |
| ALU | ALU |

Cache

CPU Memory

Bottleneck!

GPU Memory

– II) Data transfers between SMs and global memory are costly. Can on-chip memory help?

GPU (Device)

SM SM SM SM SM SM SM SM

??

Scratchpad

Registers and Local Memory

Bottleneck!

GPU Global Memory

Let's consider some software approaches first.. before moving on to hardware approaches

Background: GPU Memory Address Spaces

❑ GPU has three _address spaces_ to support increasing visibility of data between threads: local, shared, global

❑ In addition two more (read-only) address spaces: Constant and texture.

Partial Overview of CUDA Memories
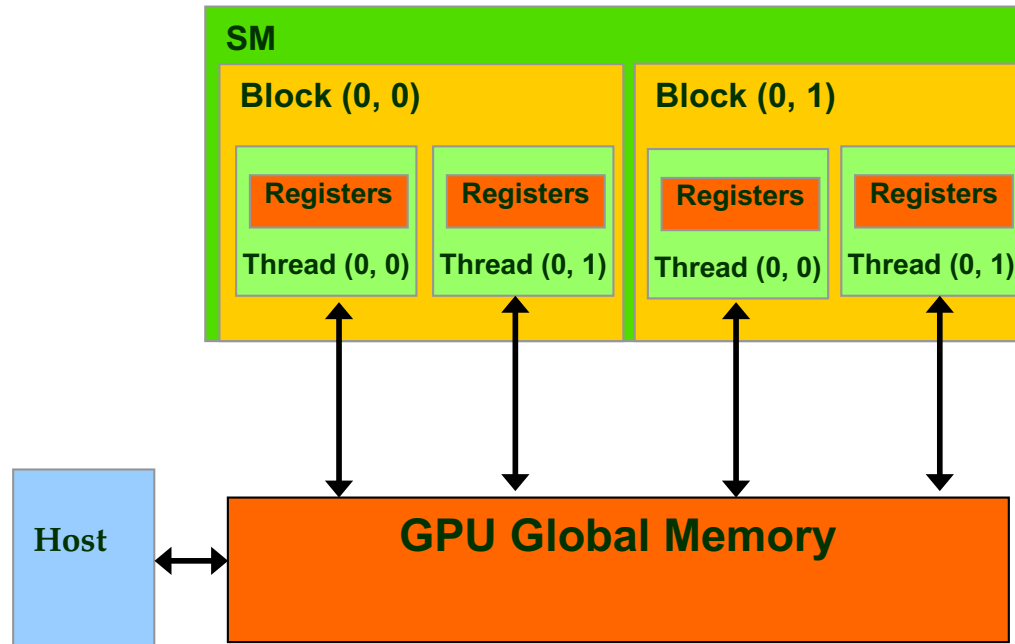
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory

- Host code can
  - Transfer data to/from per grid global memory

CUDA Device Memory Management API functions
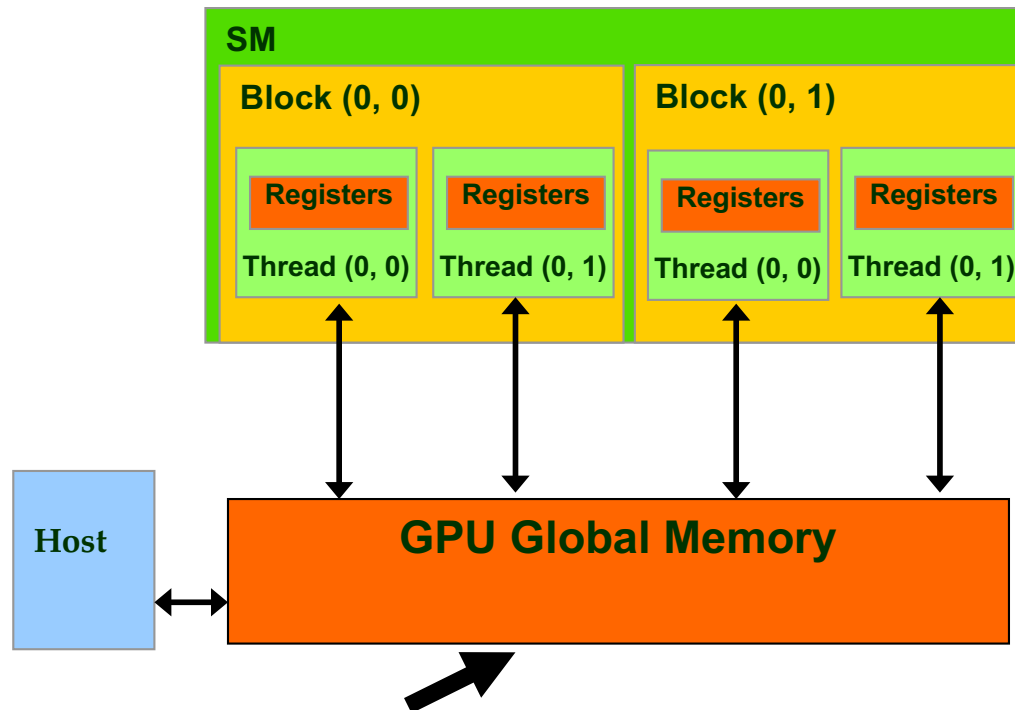
- cudaMalloc()
  - Allocates an object in the device <u>global memory</u>
  - Two parameters
    - **Address of a pointe**r to the allocated object
    - **Size of** allocated object in terms of bytes
- cudaFree()
  - Frees object from device global memory
  - One parameter
    - **Pointer** to freed object

Host-Device Data Transfer API functions
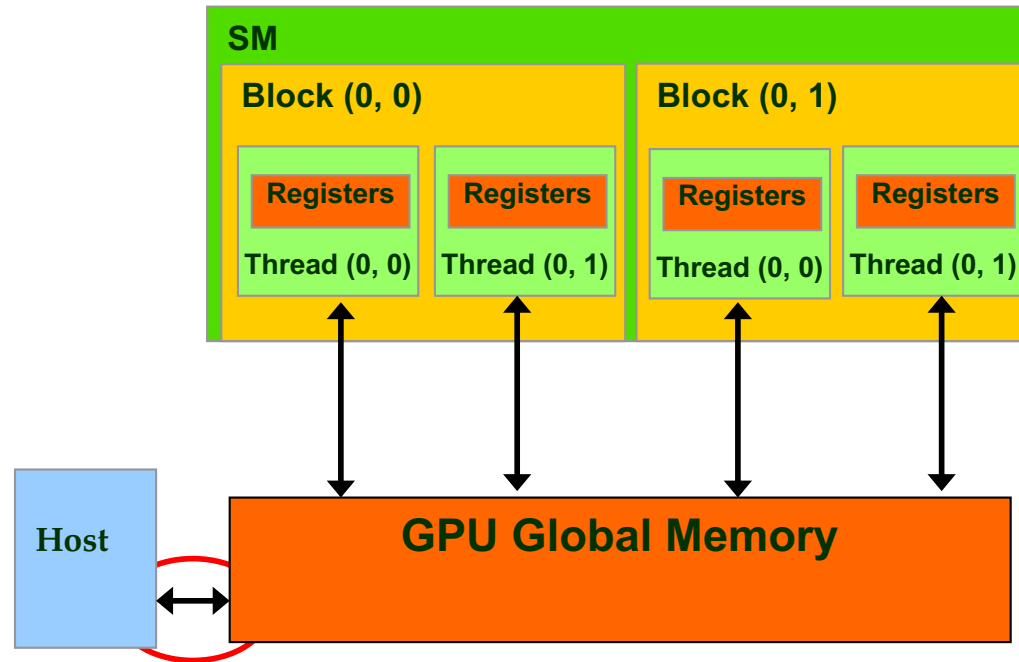
– cudaMemcpy()
  - – memory data transfer
  - – Requires four parameters
    - – Pointer to destination
    - – Pointer to source
    - – Number of bytes copied
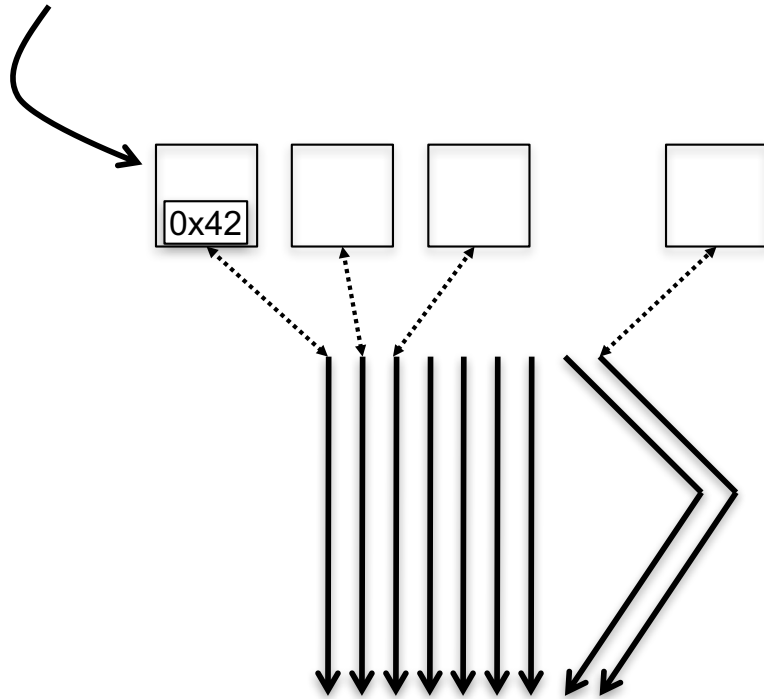   - – Type/Direction of transfer

Relatively new Features:

Transfer to device can be asynchronous

Explicit mention of memcpy by the users can be avoided by new CUDA Unified Memory

https://devblogs.nvidia.com/unified-memory-cuda-beginners/
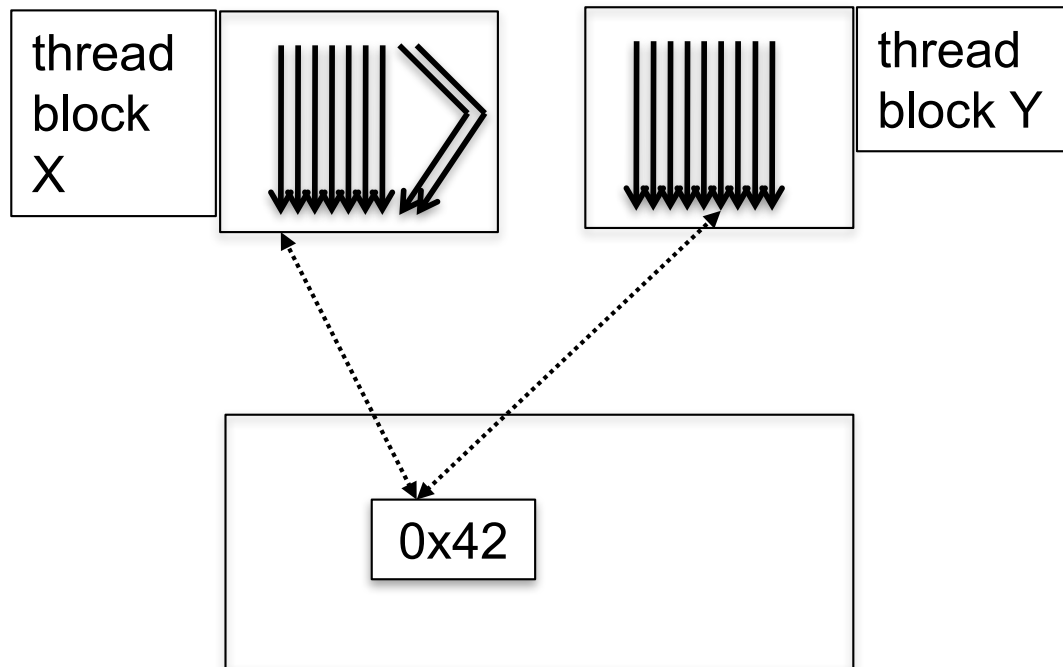
Local address Space

Each thread has own "local memory"

Example: Location at address 100 for thread 0 is different from location at address 100 for thread 1.

Contains local variables private to a thread.

Global Address Spaces

| thread block X | | | thread block Y |

0x42

Each thread in the different thread blocks (even from different kernels) can access a region called "global memory".

Commonly in GPGPU workloads threads write their own portion of global memory. Avoids need for synchronization—slow; also unpredictable thread block scheduling.

Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0

- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation

- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __*syncthreads()* to get correct results.

Warps in Multi-dimensional Thread Blocks

– The thread blocks are first linearized into 1D in row major order

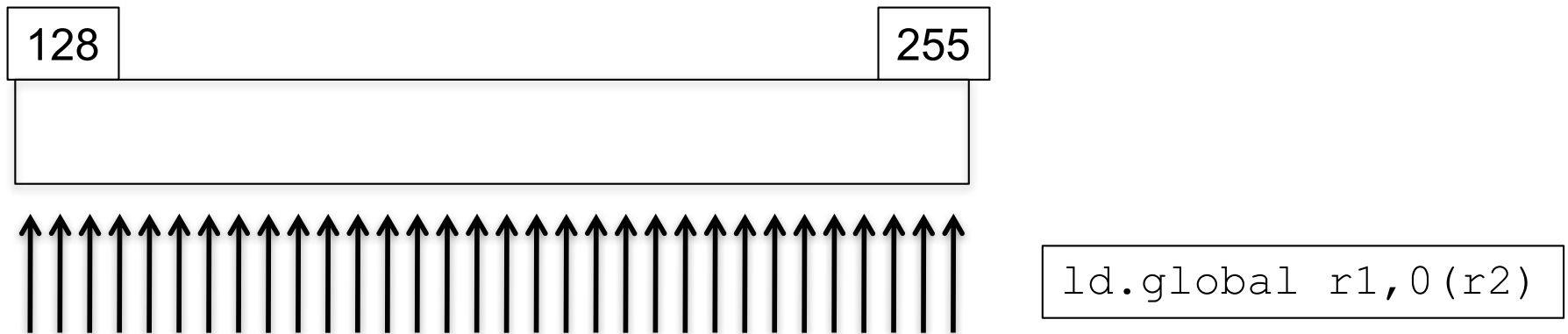– In x-dimension first, y-dimension next, and z-dimension last
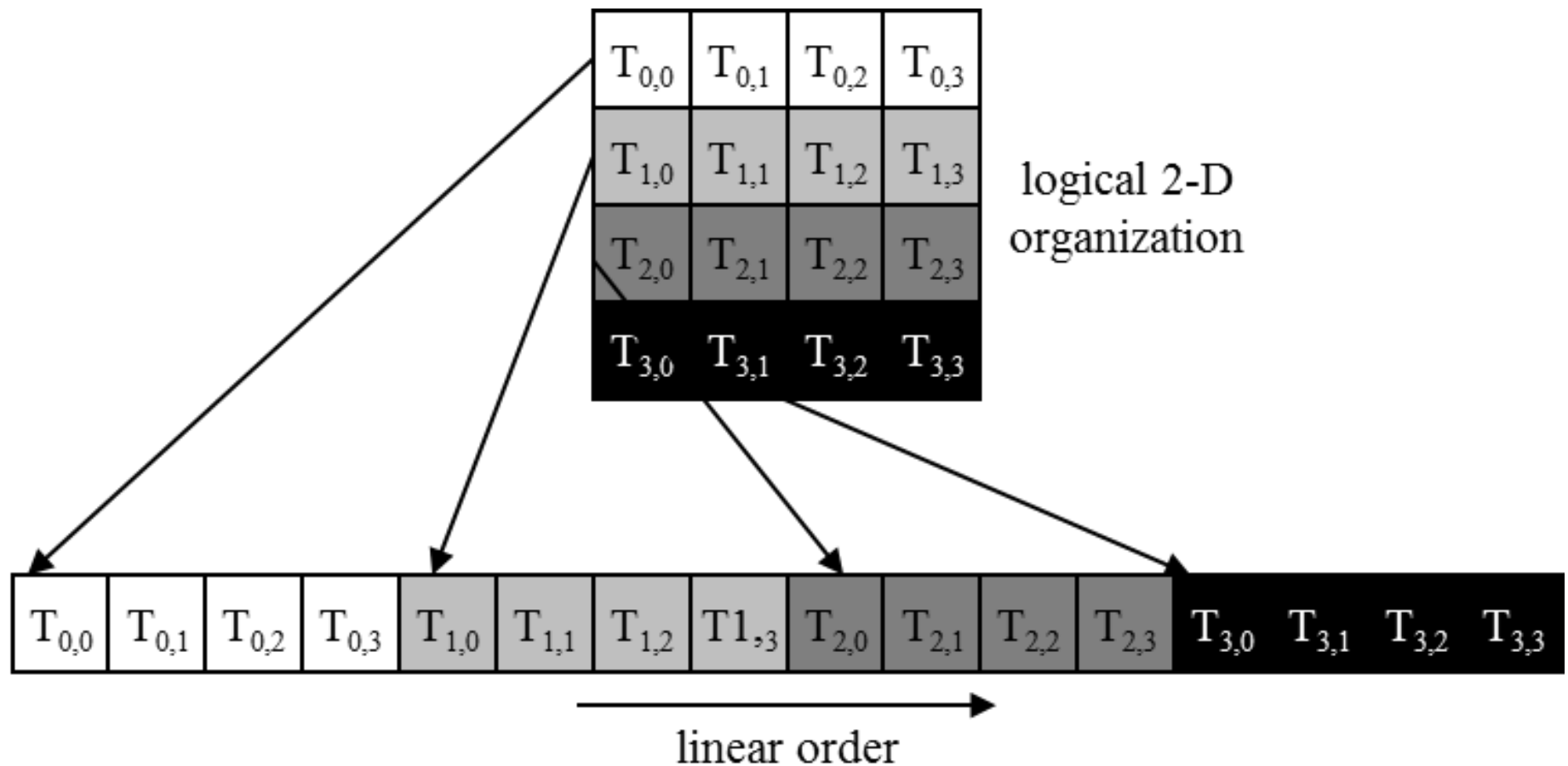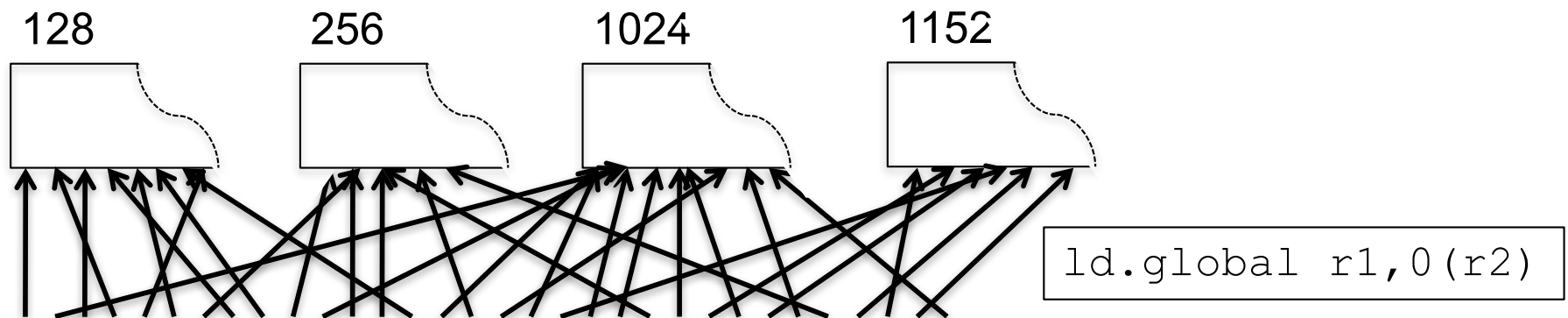
Reminder: Kernel, Blocks, Threads
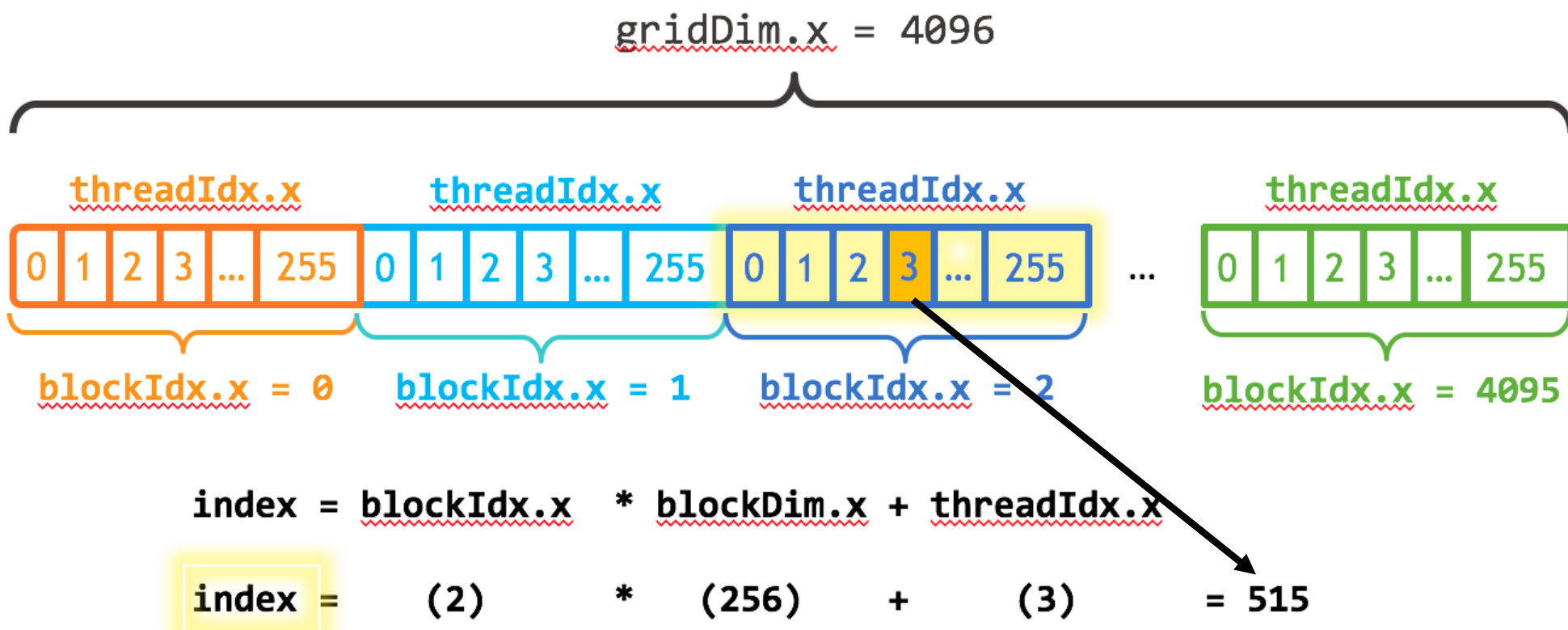
"Coalescing" global accesses

❑ Aligned accesses request single 128B cache blk

128		255

```
ld.global r1,0(r2)
```

❑ Memory Divergence:

128   256   1024   1152

```
ld.global r1,0(r2)
```

Example: Transpose (CUDA SDK)

```
__global__ void transposeNaive(float *odata, float* idata, int width)
{
    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;  // TILE_DIM = 16

    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;


    int index_in  = xIndex + width * yIndex;

    int index_out = yIndex + width * xIndex;

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) { // BLOCK_ROWS = 16
        odata[index_out+i] = idata[index_in+i*width];
    }
}
```
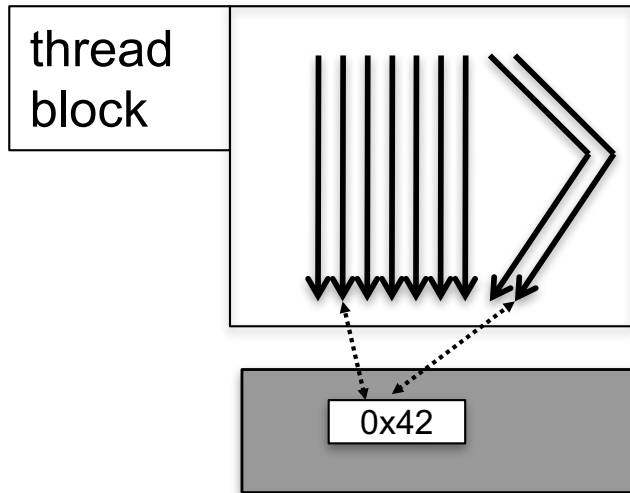
1	2
3	4

→

1	3
2	4
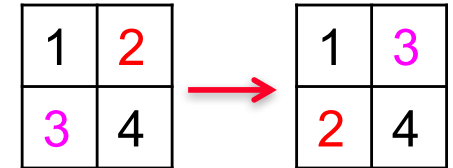
NOTE: "xIndex", "yIndex", "index_in", "index_out", and "i" are in <u>local memory</u>
  (local variables are register allocated but stack lives in local memory)

  "odata" and "idata" are pointers to <u>global memory</u>
  (both allocated using calls to cudaMalloc -- not shown above)

Write to global memory highlighted above is not "coalesced".

Scratchpad Memory

Each thread in the same thread block (work group) can access a memory region called scratchpad (or shared memory)
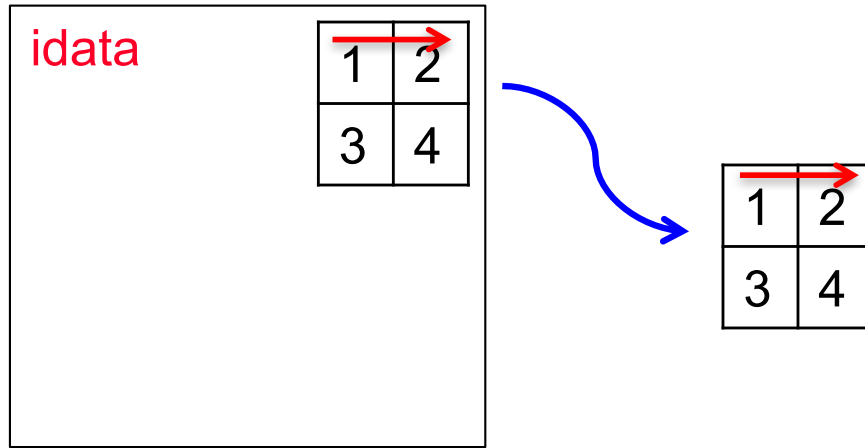
Shared memory address space is limited in size (16 to 48 KB).

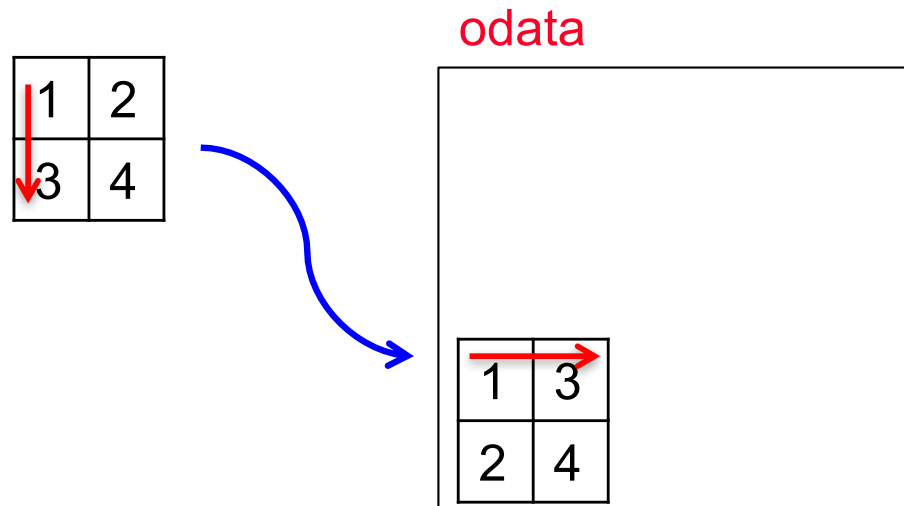Used as a software managed "cache" to avoid off-chip memory accesses.

Synchronize threads in a thread block using __syncthreads();

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory

Step 2: Copy from shared memory into global memory using coalesce write

Use of Scratchpad

```
__global__ void transposescratchpad (float *odata, float *idata, int width)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];


  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;


  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*width;


  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }
  __syncthreads();


  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*width] = tile[threadIdx.x][threadIdx.y+i];
  }
}
```
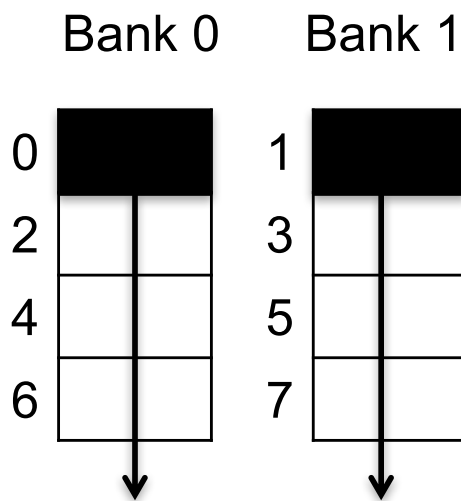
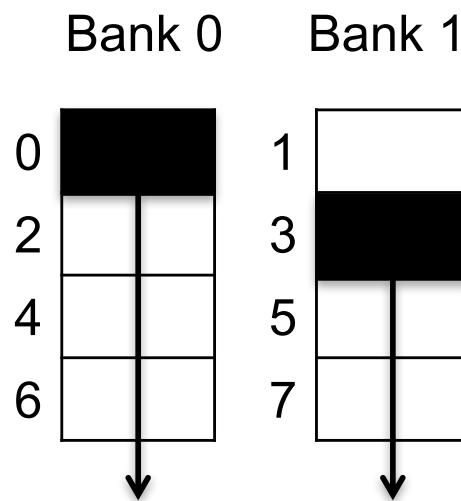GOOD: Coalesced write   BAD: Shared memory bank conflicts

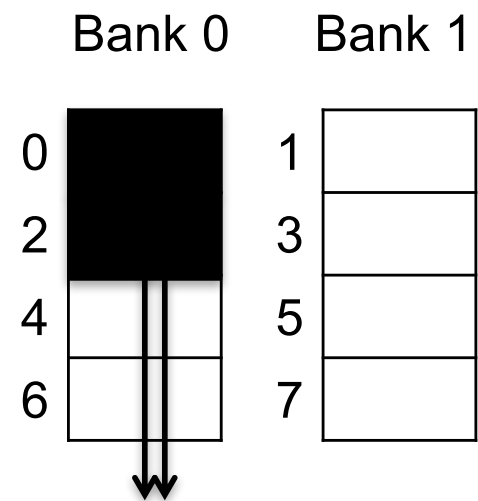https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

Bank Conflicts

❑ To increase bandwidth common to organize memory into multiple banks.

❑ Independent accesses to different banks can proceed in parallel

Example 1: Read 0, Read 1
(can proceed in parallel)
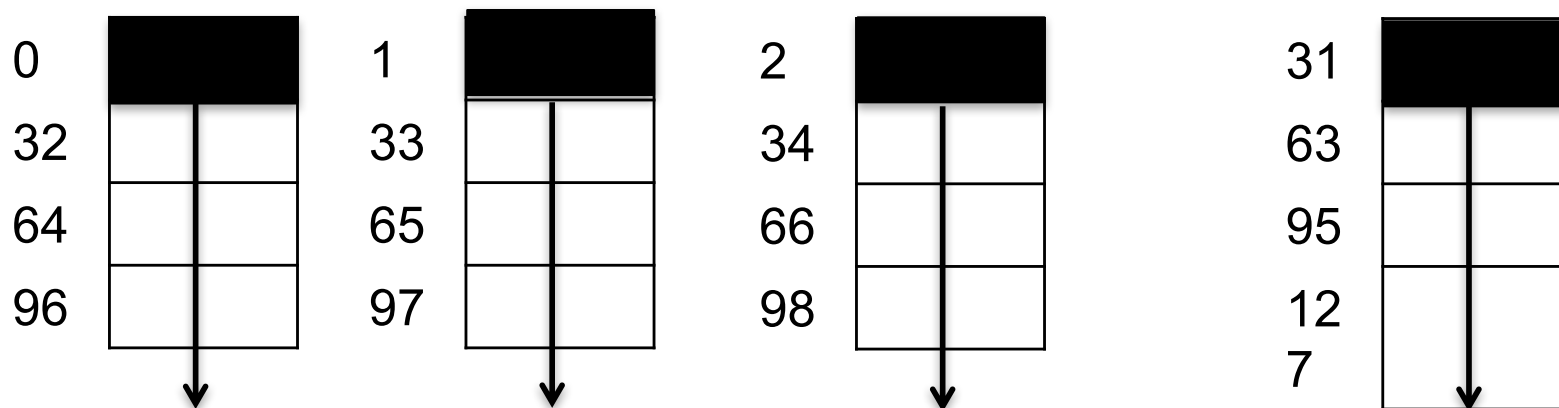
Example 2: Read 0, Read 3
(can proceed in parallel)

Example 3: Read 0, Read 2
(bank conflict)

Shared Memory Bank Conflicts

`__shared__ int A[BSIZE];`

`…`

`A[threadIdx.x] = … // no conflicts`

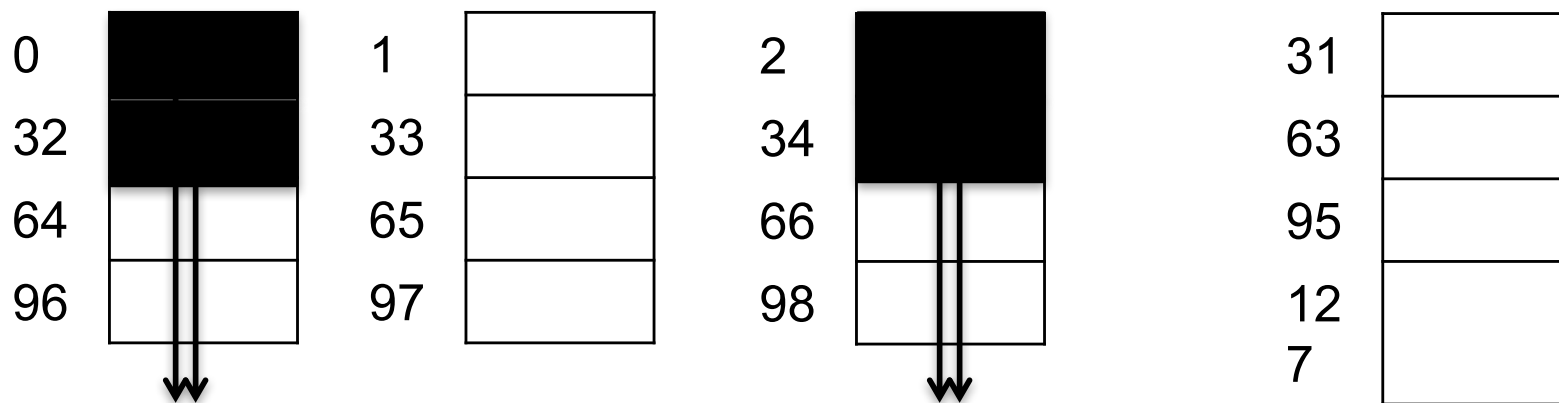# Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];

…

A[2*threadIdx.x] = // 2-way conflict
```
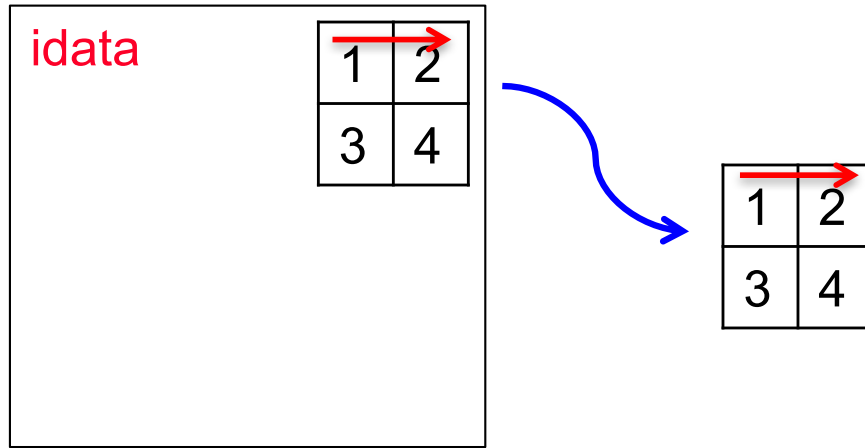
Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory

idata

| 1 | 2 |
| 3 | 4 |

| 1 | 2 |
| 3 | 4 |

Step 2: Copy from shared memory into global memory using coalesce write

odata

| 1 | 2 |
| 3 | 4 |

Problem: Access two locations in same shared memory bank.

| 1 | 3 |
| 2 | 4 |

Eliminate Bank Conflicts

```
__global__ void transposeNoBankConflicts (float *odata, float *idata, int width)
{
    __shared__ float tile[TILE_DIM][TILE_DIM+1];


    int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;

    int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

    int index_in = xIndex + (yIndex)*width;


    xIndex = blockIdx.y * TILE_DIM + threadIdx.x;

    yIndex = blockIdx.x * TILE_DIM + threadIdx.y;

    int index_out = xIndex + (yIndex)* width;


    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
    }
    __syncthreads();
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
        odata[index_out+i*width] = tile[threadIdx.x][threadIdx.y+i];
    }
}
```
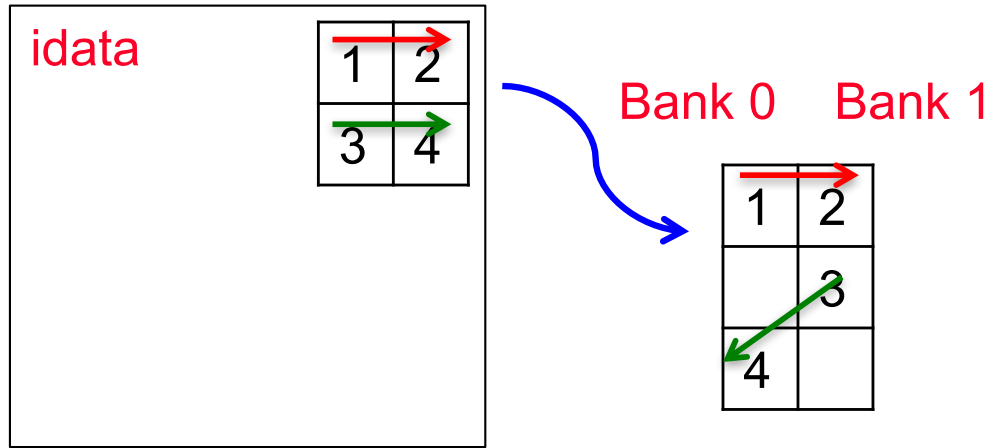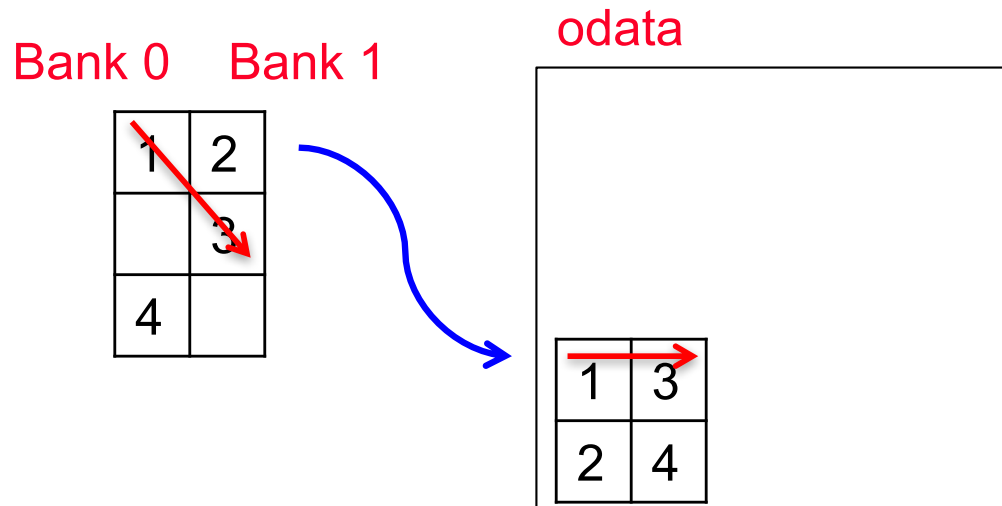
https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory

idata

| 1 | 2 |
| 3 | 4 |

Bank 0   Bank 1

1	2
	3
4	

Step 2: Copy from shared memory into global memory using coalesce write

Bank 0   Bank 1

1	2
	3
4	

odata

| 1 | 3 |
| 2 | 4 |

Reading Material

❏ NVIDIA Blogs:

- *https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/*

- https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

❏ GPGPU-sim Manual and Tutorial Slides

- http://www.gpgpu-sim.org/manual

- http://www.gpgpu-sim.org/micro2012-tutorial/

❏ More background material: Jog et al., OWL: Cooperative Thread Array Aware Scheduling Techniques for Improving GPGPU performance, ASPLOS'13