ACACES 2018 Summer School

GPU Architectures: Basic to Advanced Concepts

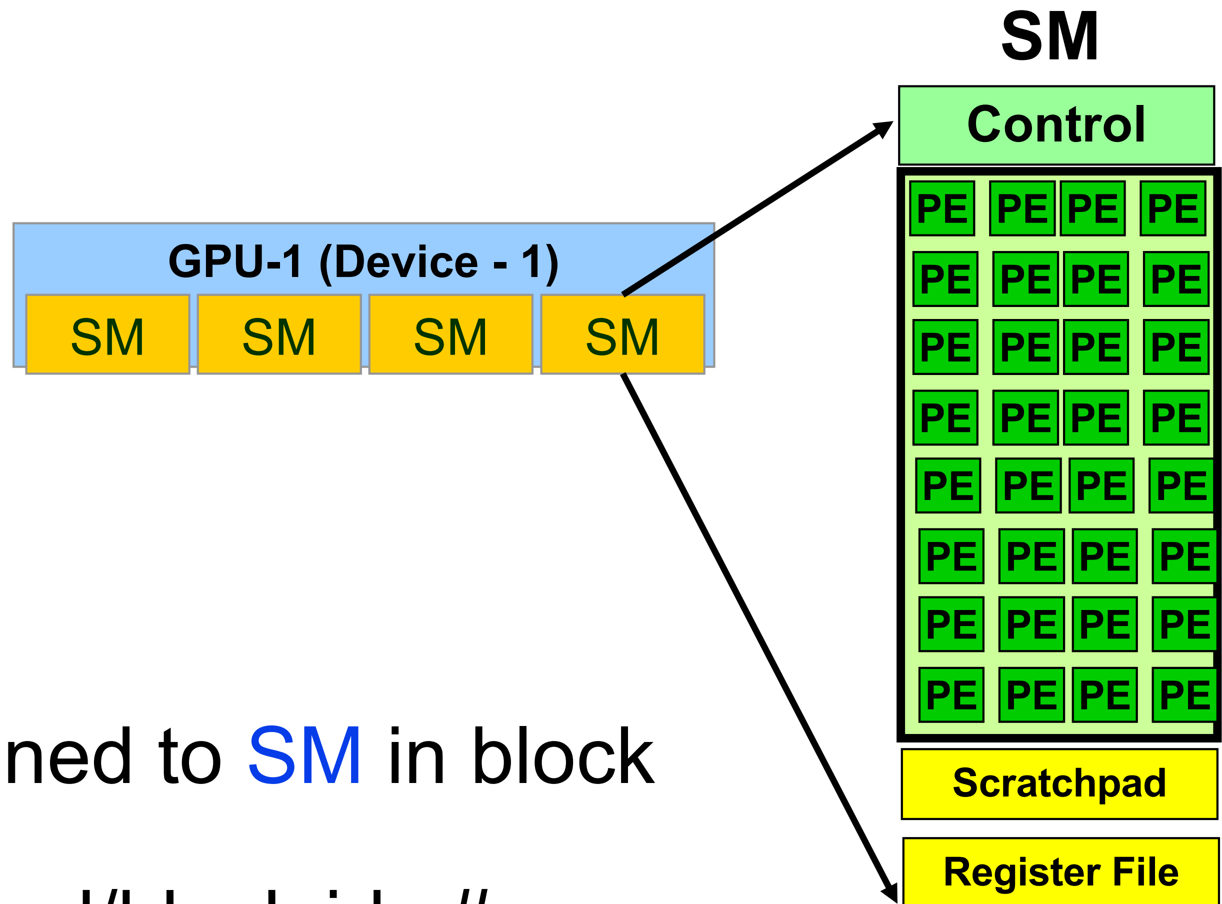
Adwait Jog, Assistant Professor

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

Streaming Multi-Processor (SM)



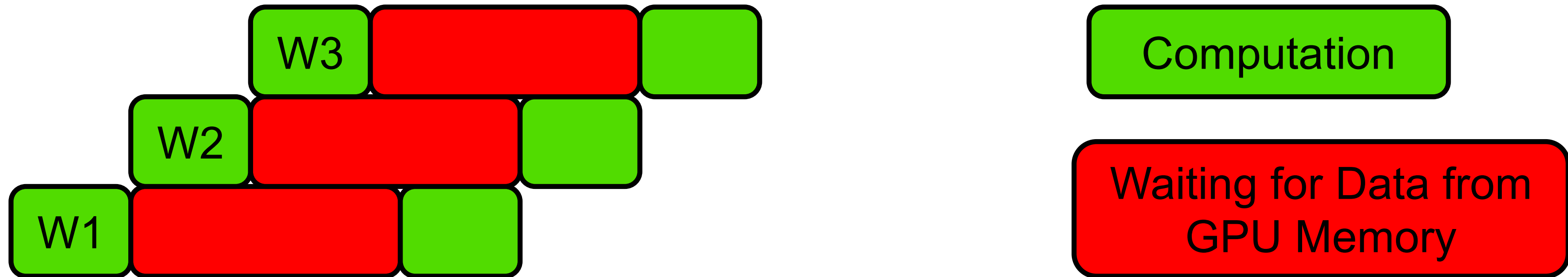
- Threads are assigned to SM in block granularity
- SM maintains thread/block idx #s
- SM manages/schedules thread execution
- Multiple blocks can be allocated to the SM
  - Based on the amount of resources (shared memory, register file etc.)

GPU Execution Model

- Blocks assigned to each SM are scheduled on the associated SIMD hardware (i.e., on the Processing Elements (PEs) or CUDA Cores).
- SM bundles threads (from various blocks) into warps (wavefronts) and runs them in lockstep on across PEs.
- An NVIDIA warp groups 32 consecutive threads together (AMD wave-fronts group 64 threads together)
- Warps are:
  - Scheduling units in SM
  - Scheduled in multiplexed and pipelined manner on the SM

Tolerating Long Latencies

- Execution in an SM



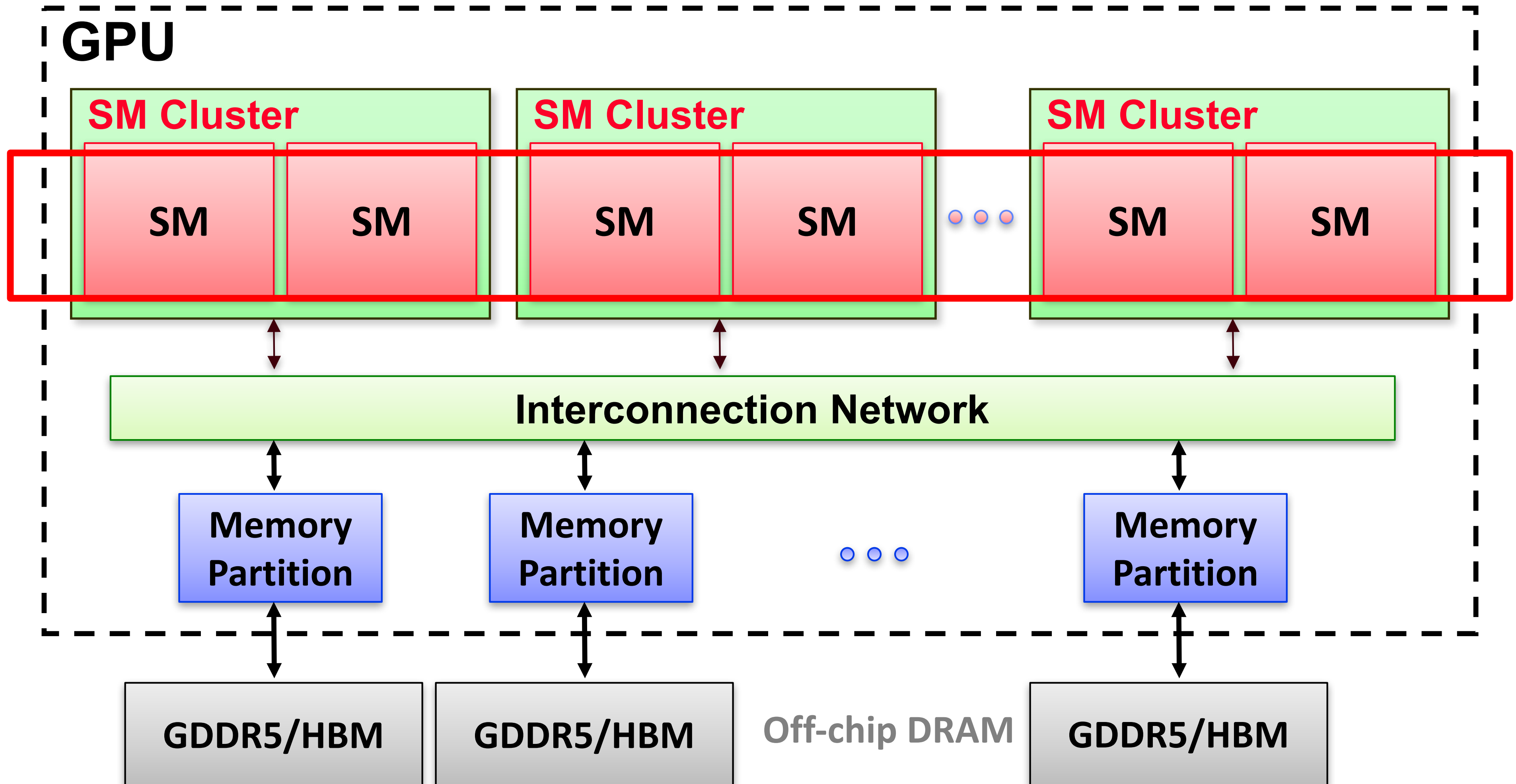
GPU attempts to hide long memory latency with computation from other warps

How SMs are able to context switch between warps so quickly?

Key Points so Far

- Programmer organize threads into “blocks” (up to 1024 threads per block)
- Motivation: Write parallel software once and run on future hardware
- Hardware spawns more threads/warps than GPU can run (some may wait)
- Warps associated with blocks can help in tolerating long latencies.
- GPUs support large register files (for fast context switching) and high bandwidth memories (for providing data to large number of concurrent threads)

GPU Architecture Overview

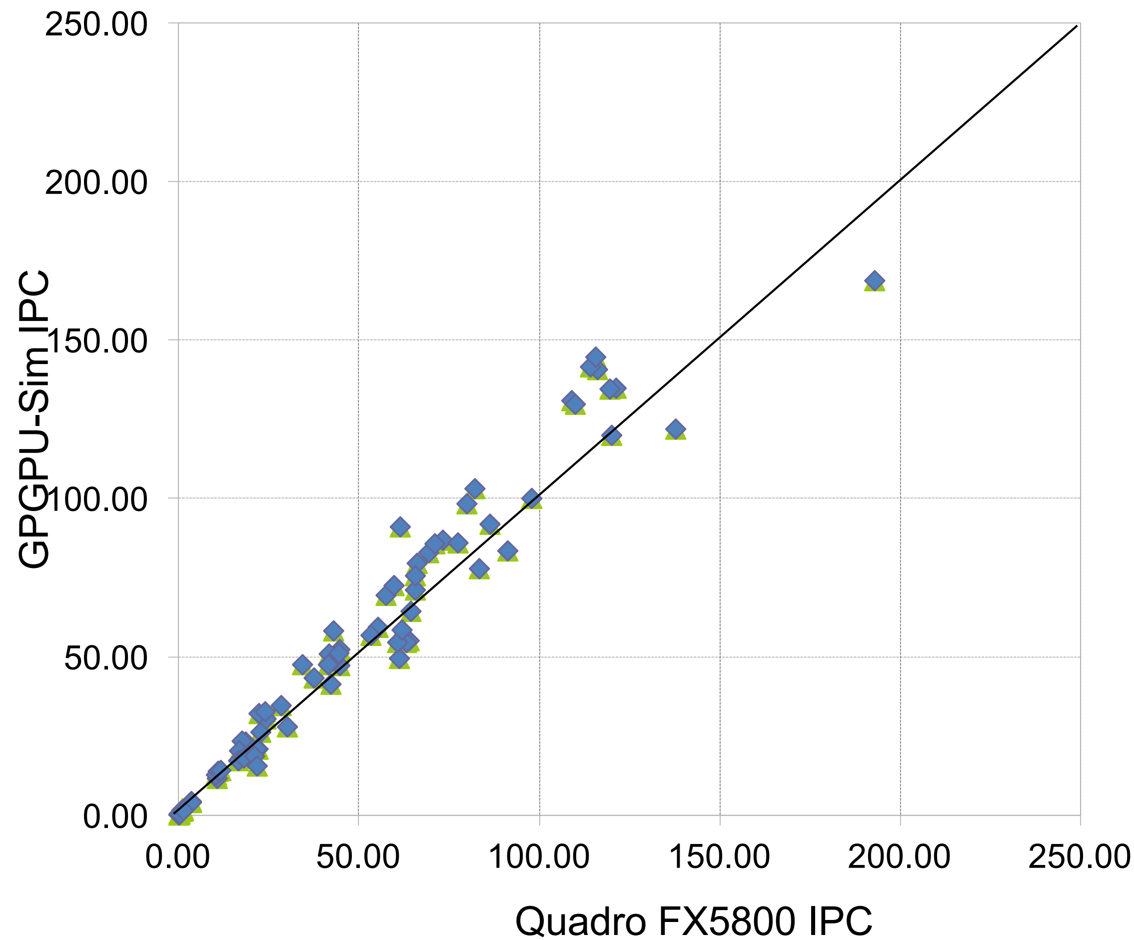


GPU Microarchitecture

- 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



Correlation
~0.976

GPU Instruction Set Architecture (ISA)

- NVIDIA defines a virtual ISA, 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
```

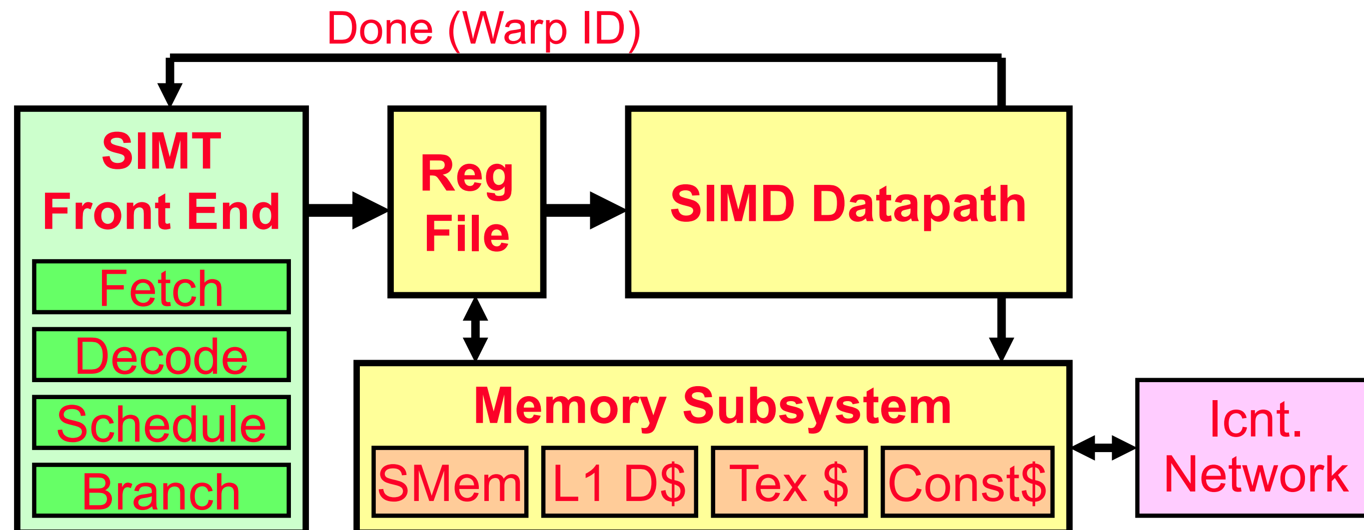
- Memory operations:

```
ld.global.f32 f, [a];
ld.shared.u32 g, [b];
st.local.f64  [c], h
```

- Compare and branch operations:

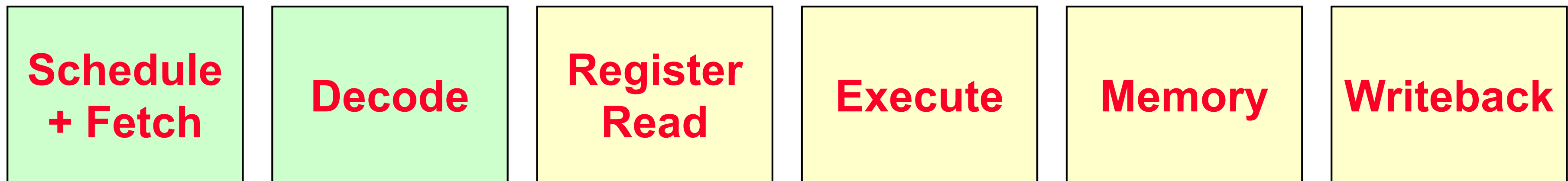
```
    setp.eq.f32 p, y, 0;  // is y equal to zero?
@p bra L1  // branch to L1 if y equal to zero
```

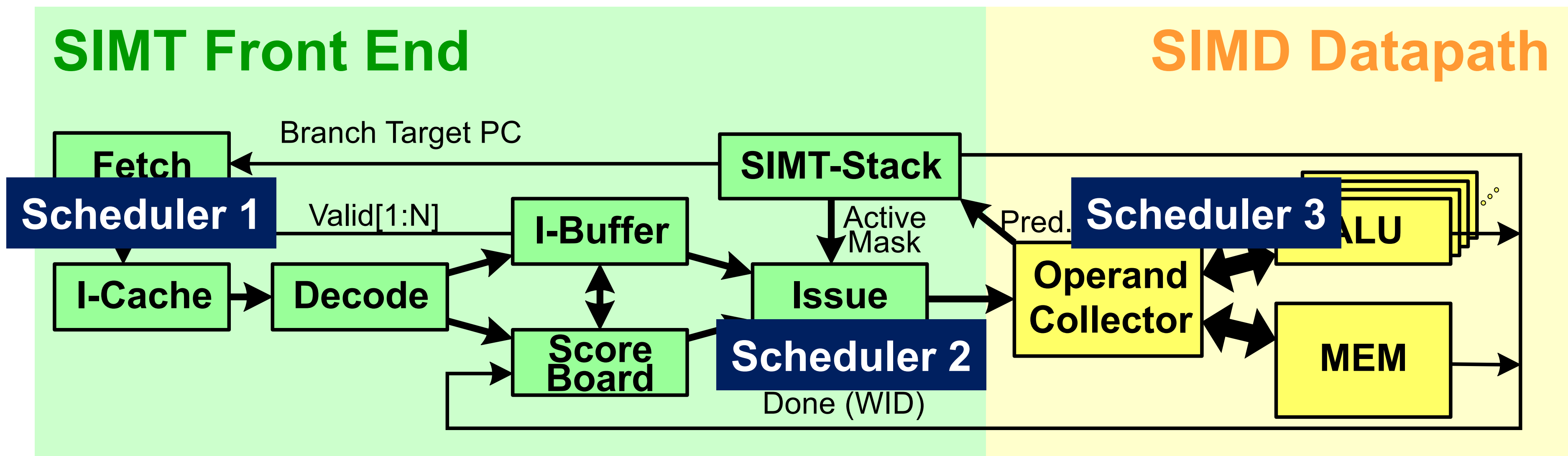
Inside an SM (1)



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

Inside an SM (2)

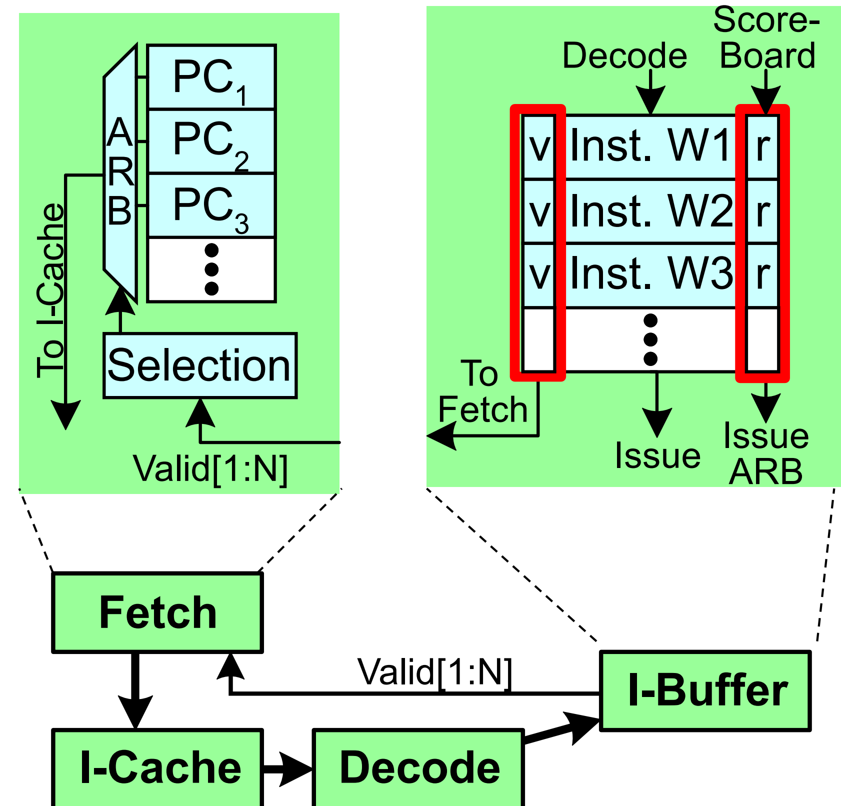




- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple SIMD functional units

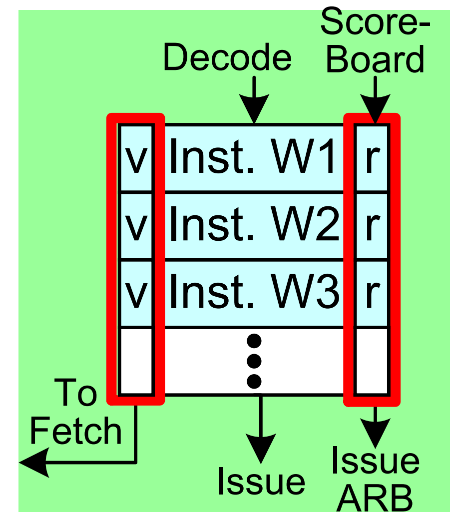
Fetch + Decode

- 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

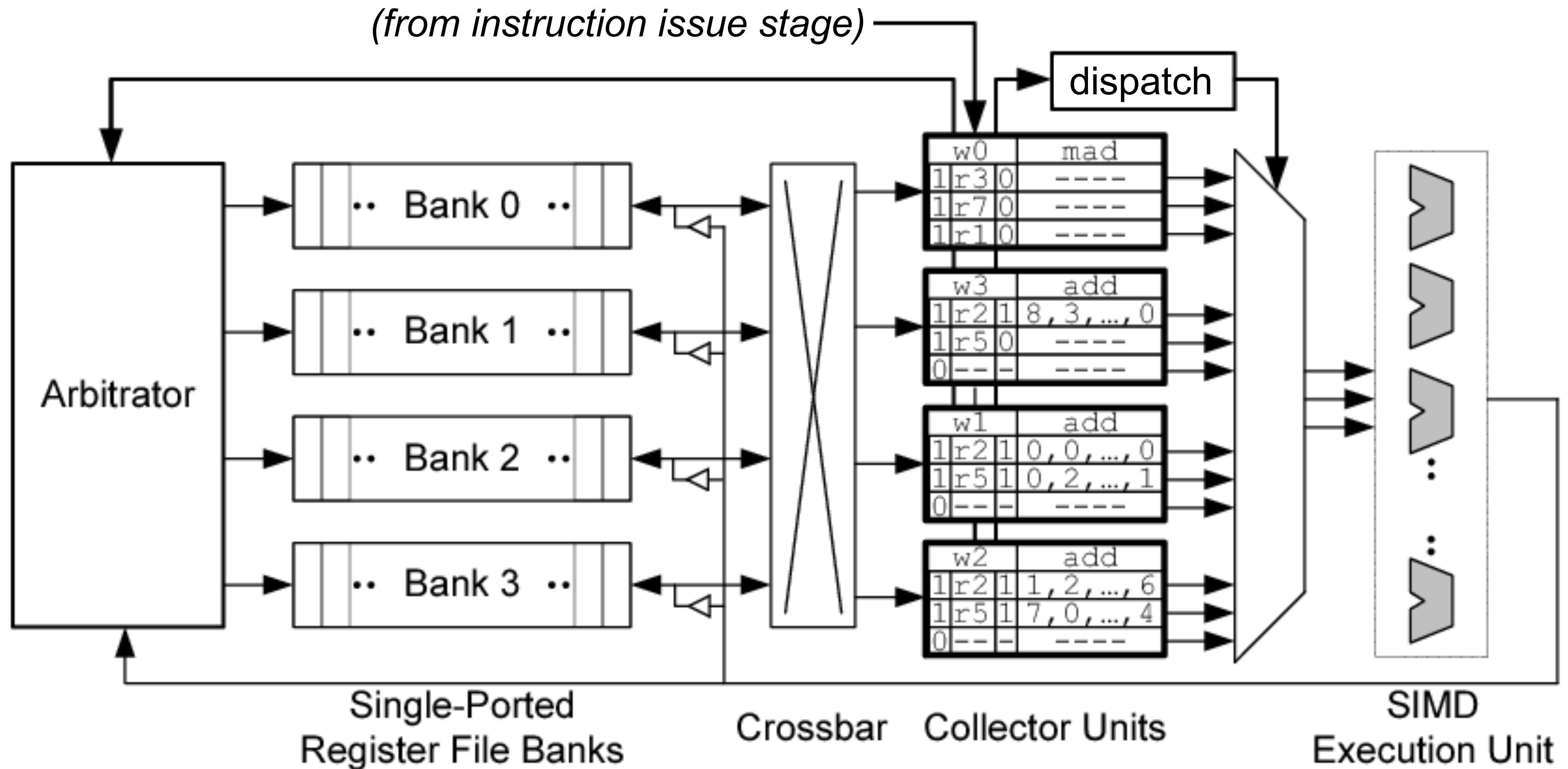
- Select a warp and issue an instruction from its I-Buffer for execution
  - Scheduling: Greedy-Then-Oldest (GTO)
  - GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  - Fermi: Odd/Even scheduler
  - To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)



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

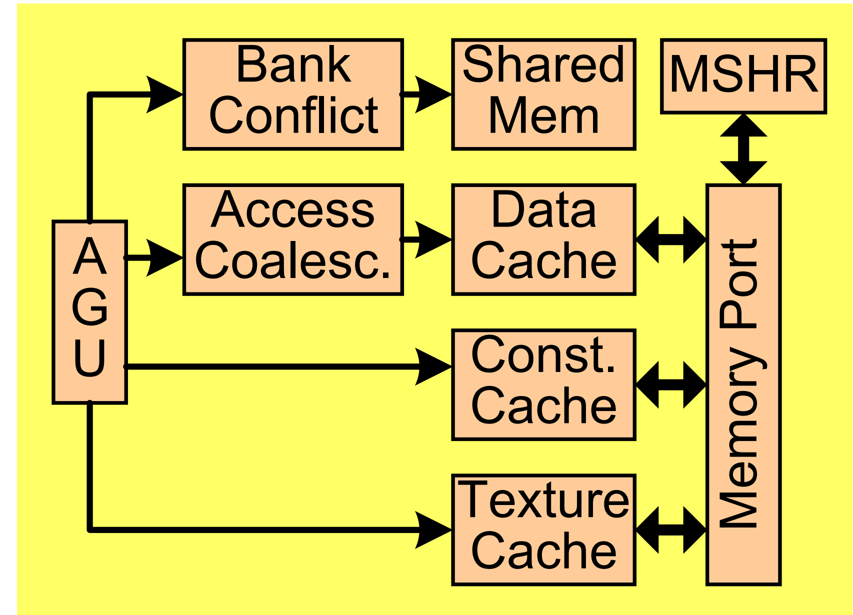


ALU Pipelines

- 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

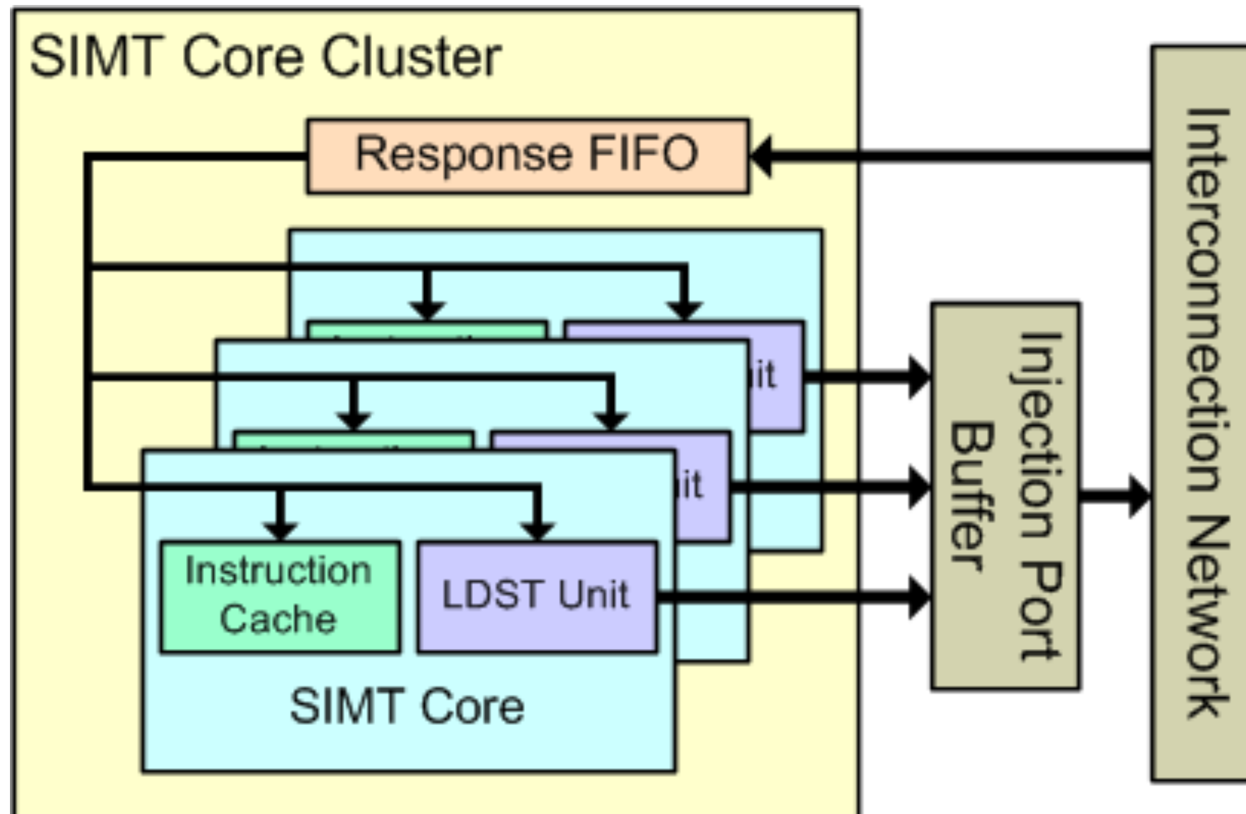


Writeback

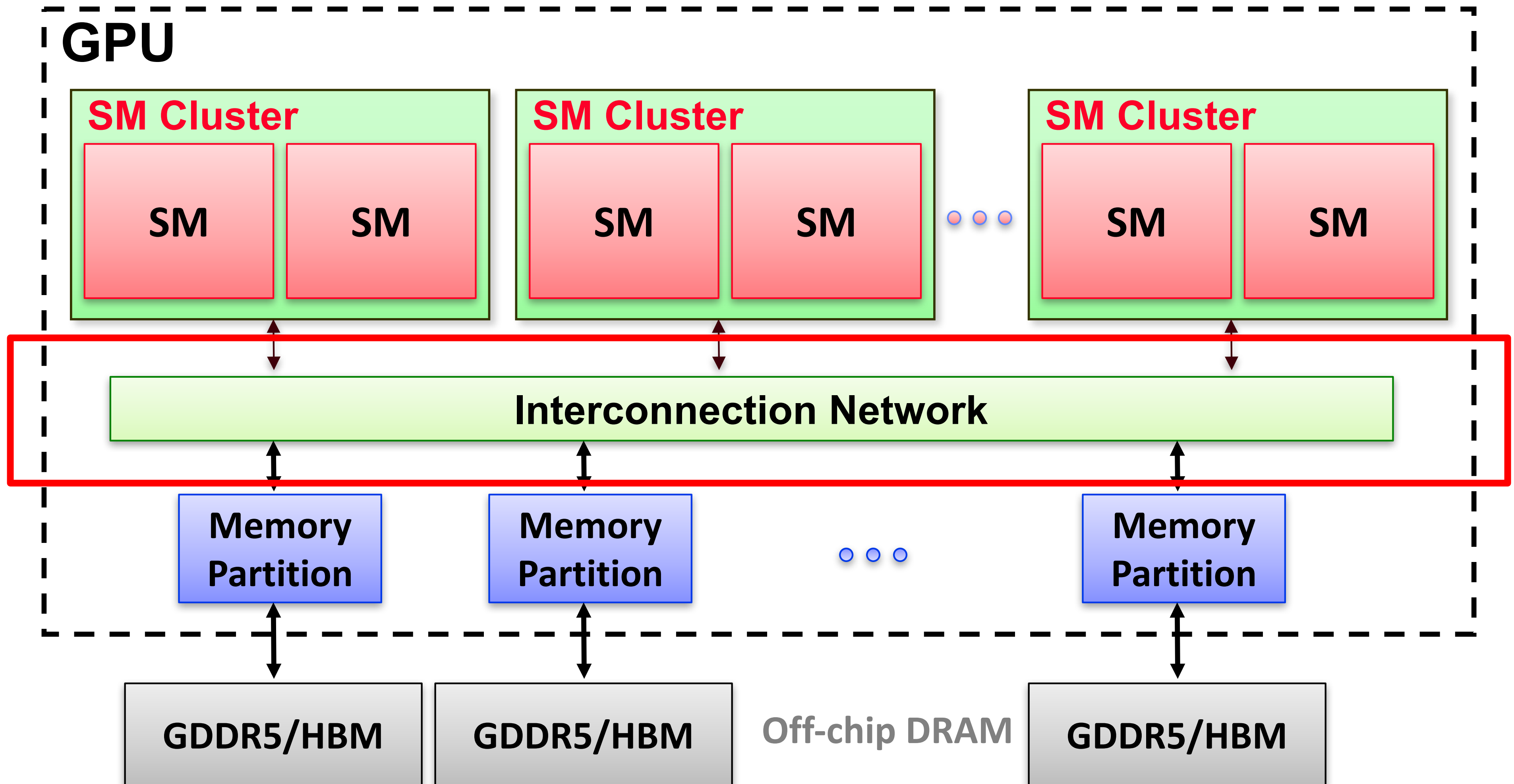
- Each pipeline has a result bus for writeback
- Exception:
  - SP and SFU pipe shares a result bus
  - Time slots on the shared bus is pre-allocated

SM Cluster

- Collection of SIMT cores



GPU Architecture Overview

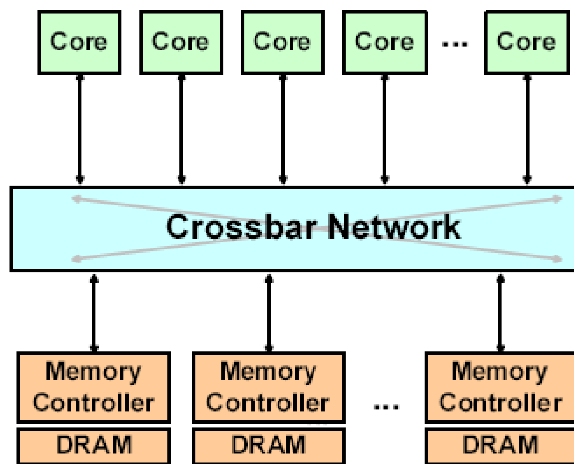


Interconnection Network Model

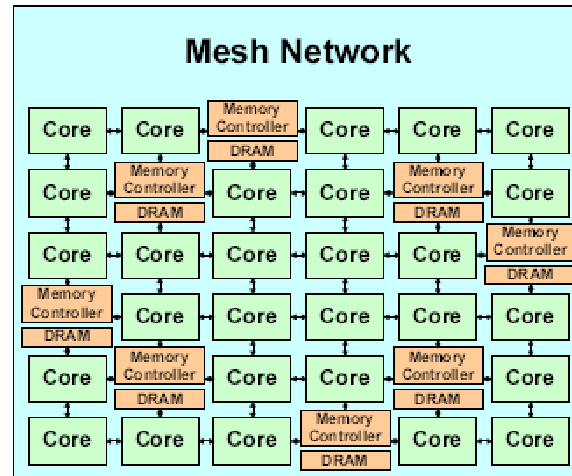
- Intersim (Booksim) a flit level simulator
  - Topologies (Mesh, Torus, Butterfly, …)
  - Routing (Dimension Order, Adaptive, etc.)
  - Flow Control (Virtual Channels, Credits)

- Two separate networks
  - From SIMT cores to memory partitions
    - Read Requests, Write Requests
  - From memory partitions to SIMT cores
    - Read Replies, Write Acks

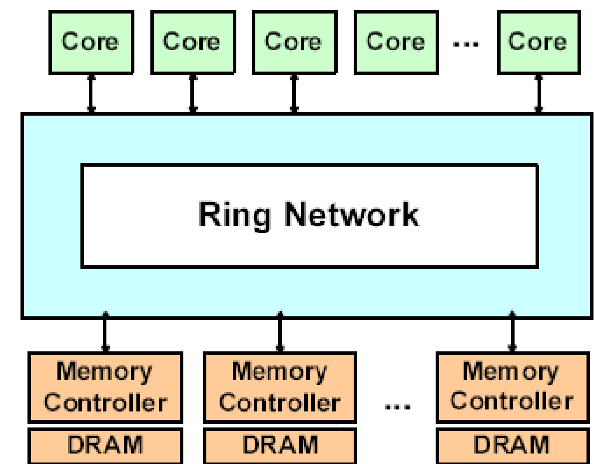
Topology Examples



(a) Crossbar

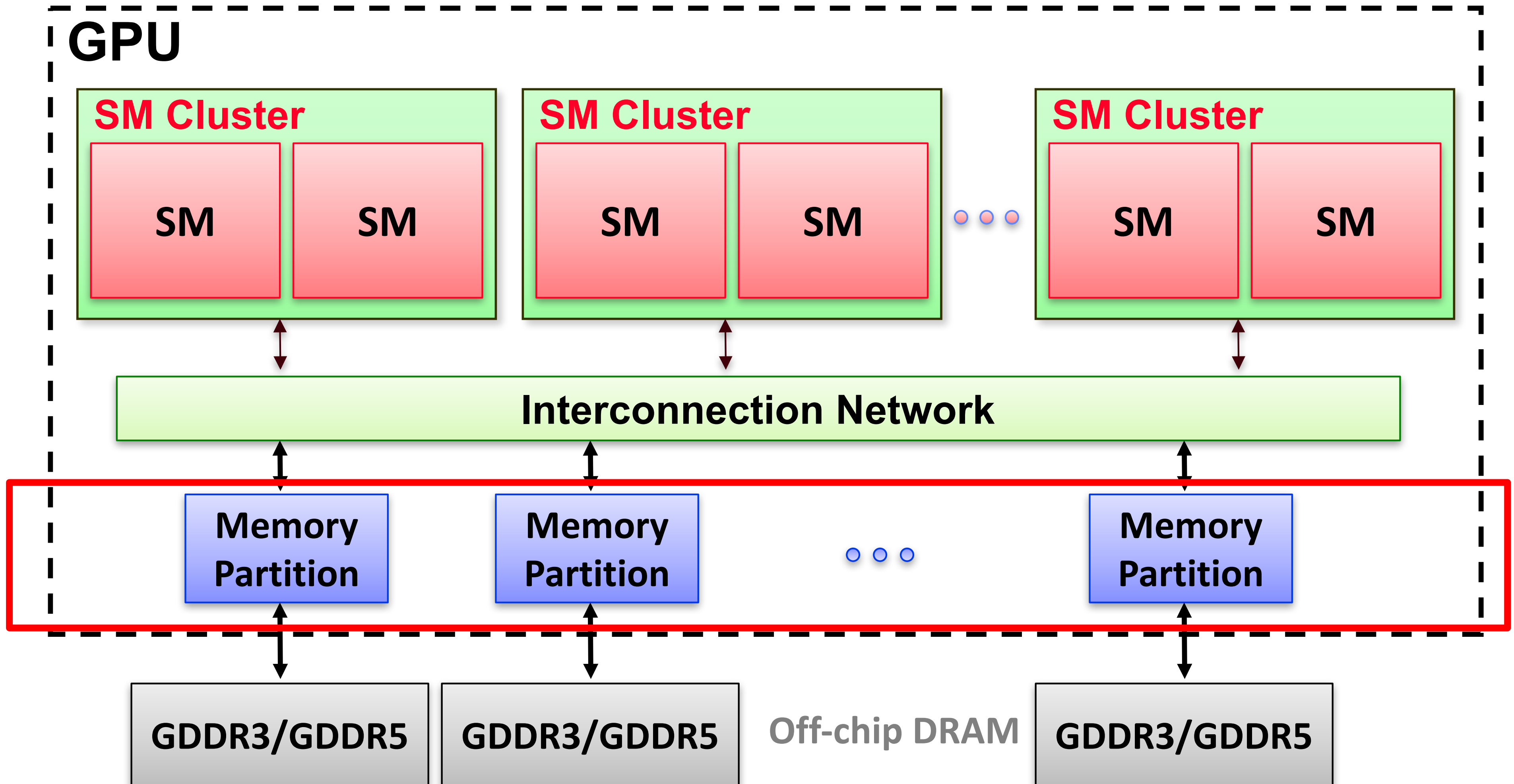


(b) Mesh

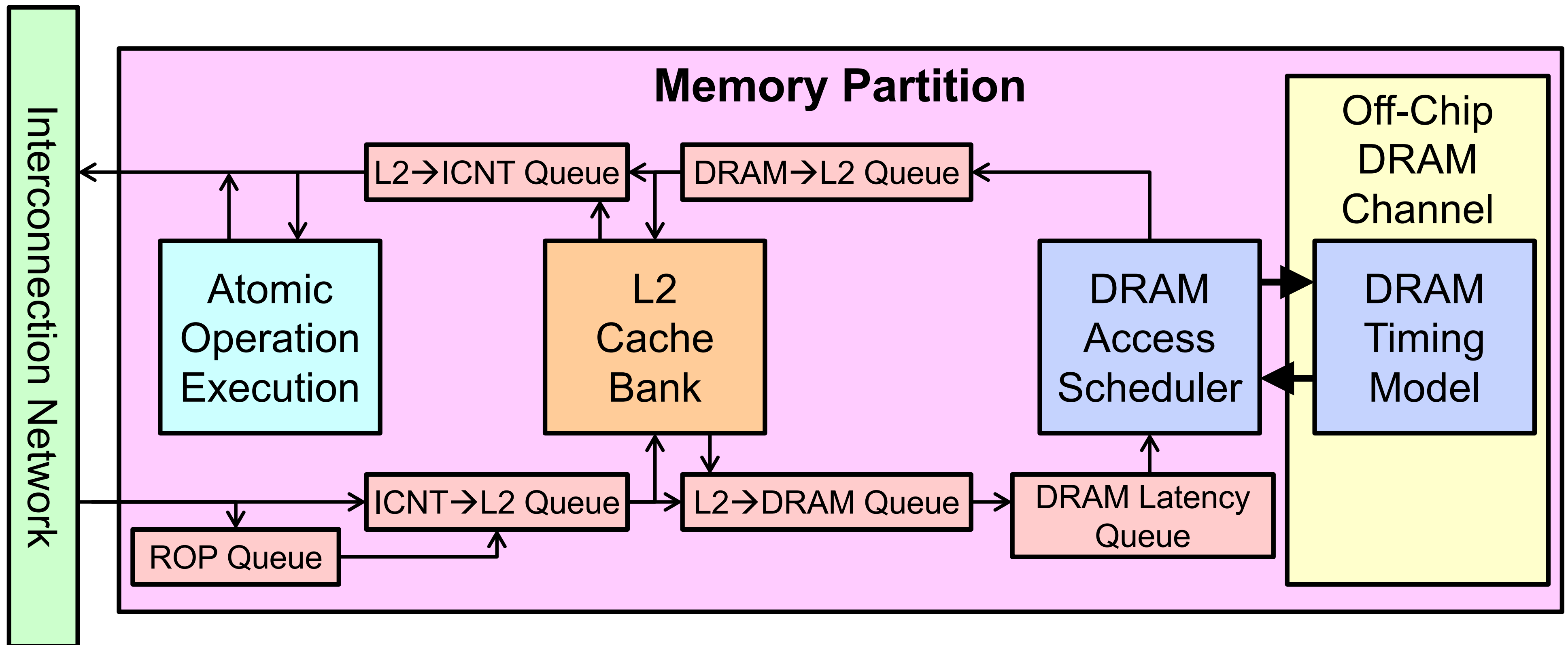


(c) Ring

GPU Architecture Overview

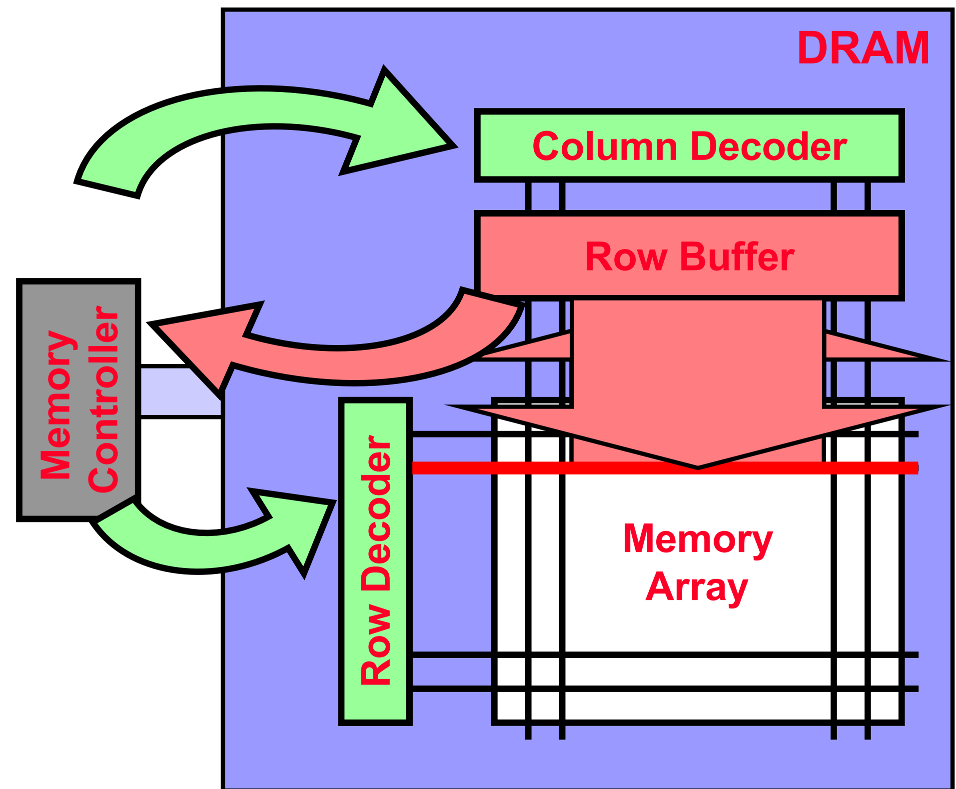


Memory Partition



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!

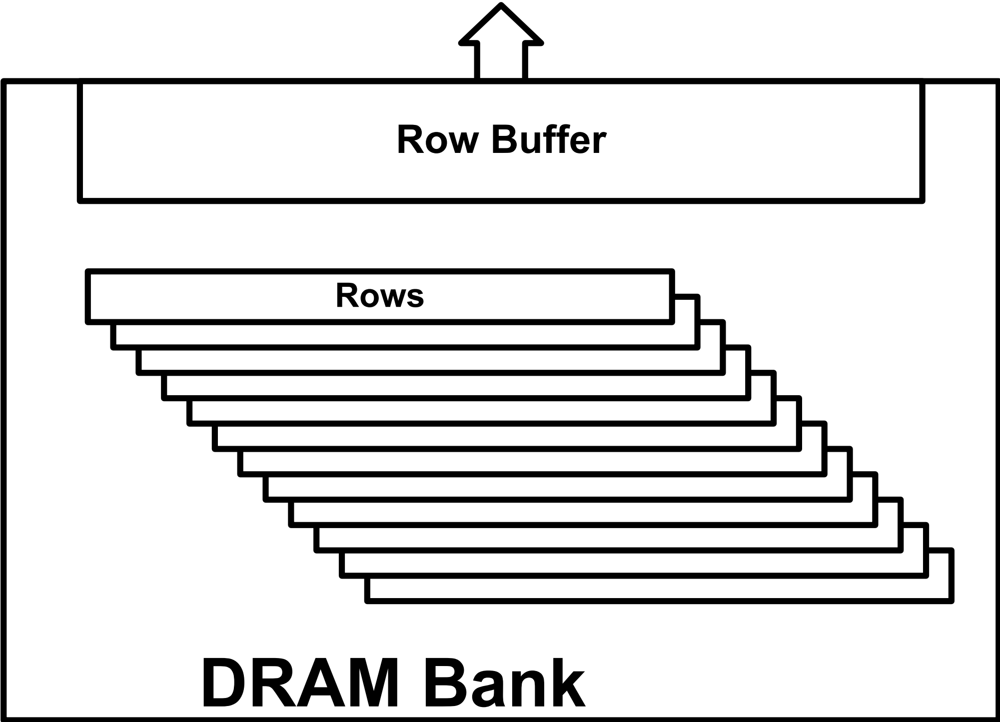


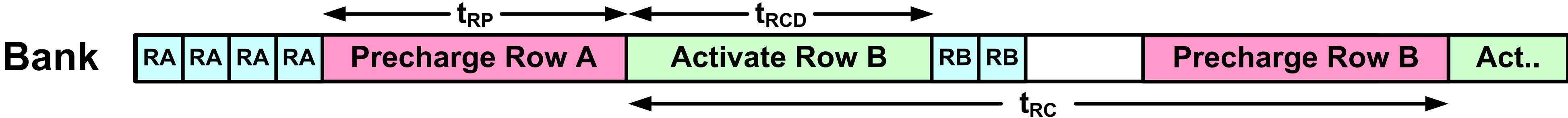
DRAM Row Access Locality

t_{RC} = row cycle time

t_{RP} = row precharge time

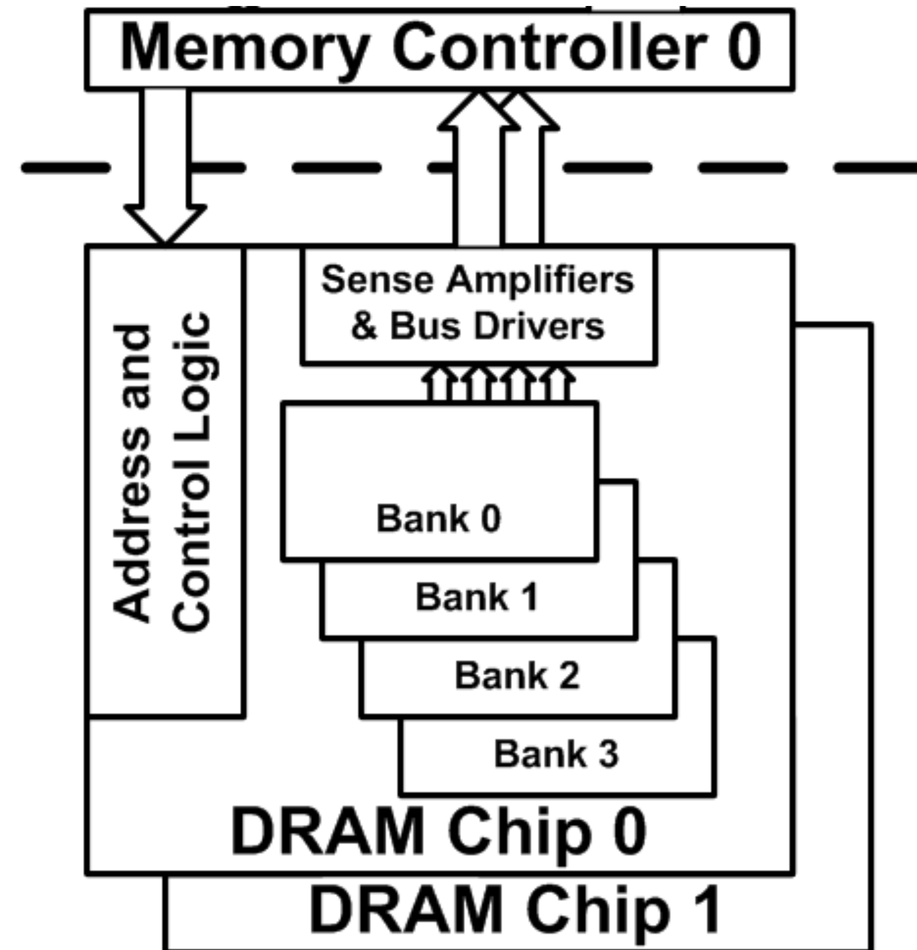
t_{RCD} = row activate time





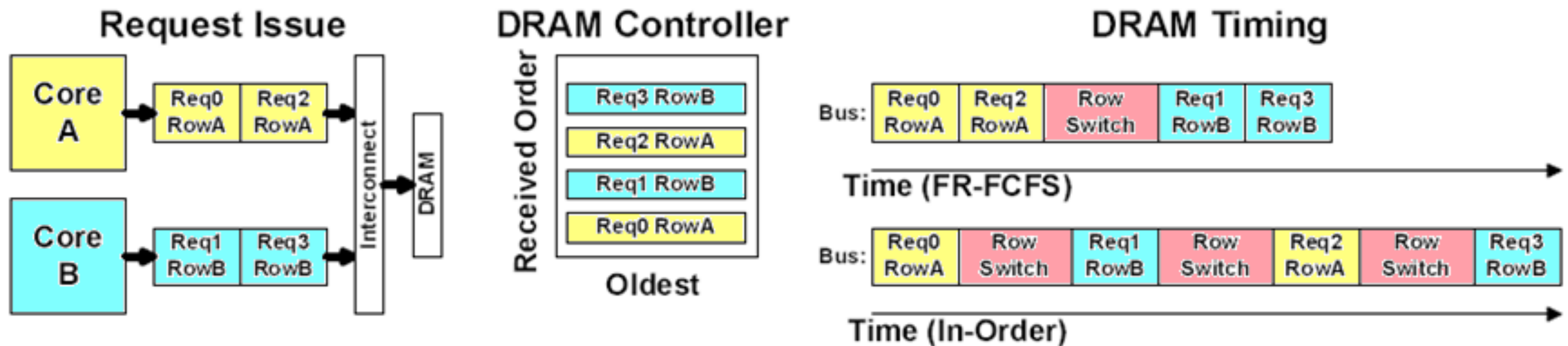
DRAM Bank-level Parallelism

- 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

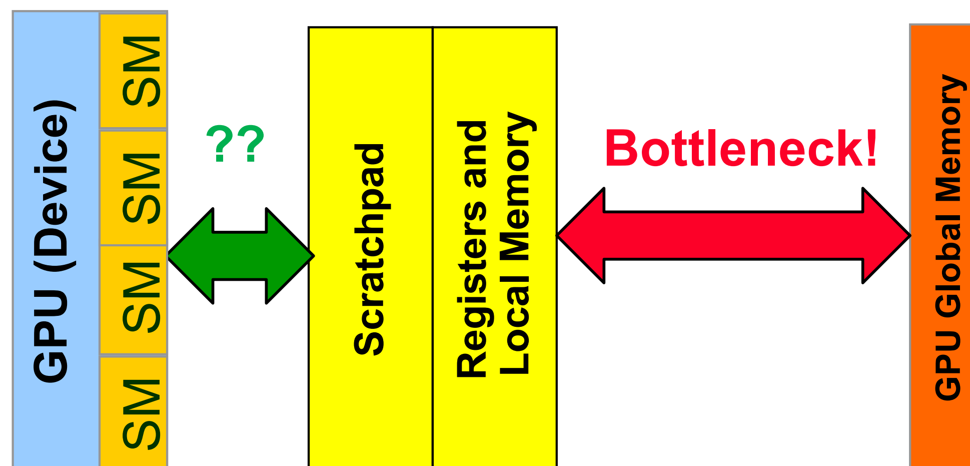


Key GPU Performance Concerns

Key GPU Performance Concerns

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

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

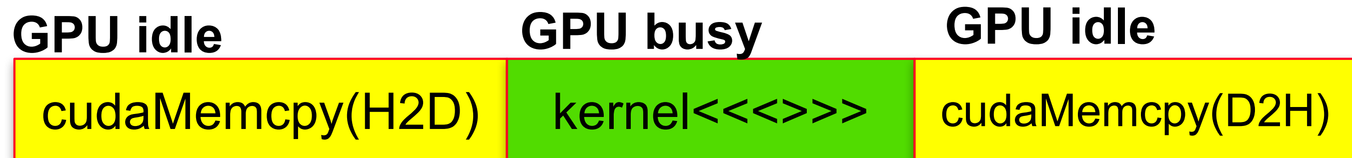


CUDA Streams

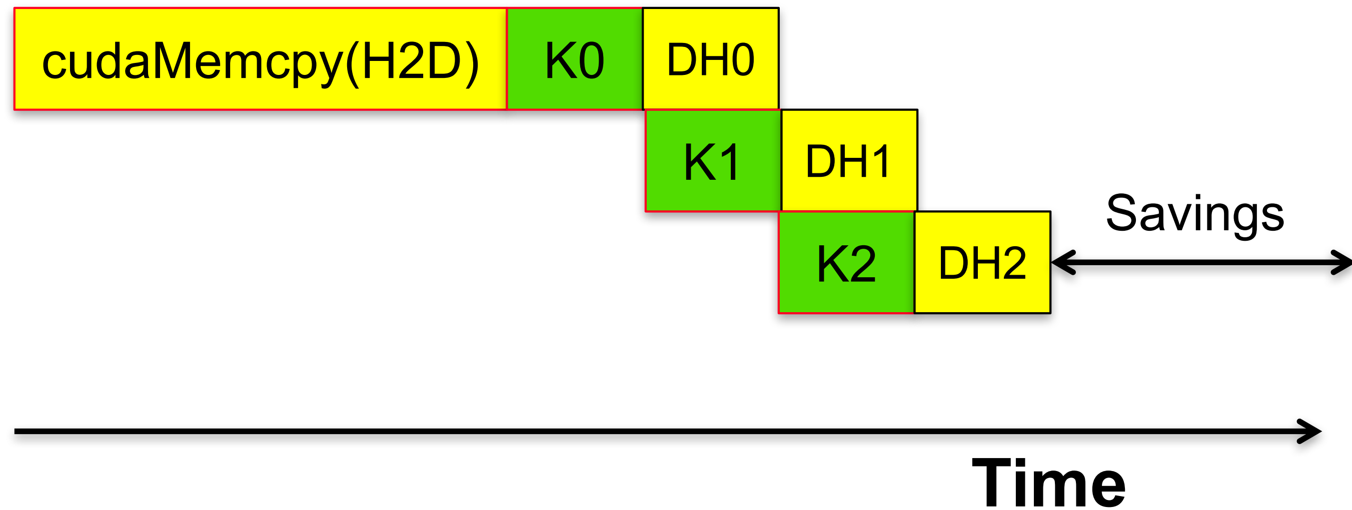
- CUDA (and OpenCL) provide the capability to overlap computation on GPU with memory transfers using "Streams" (Command Queues)
- A Stream orders a sequence of kernels and memory copy "operations".
- Operations in one stream can overlap with operations in a different stream.

How Can Streams Help?

Serial:



Streams:



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

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

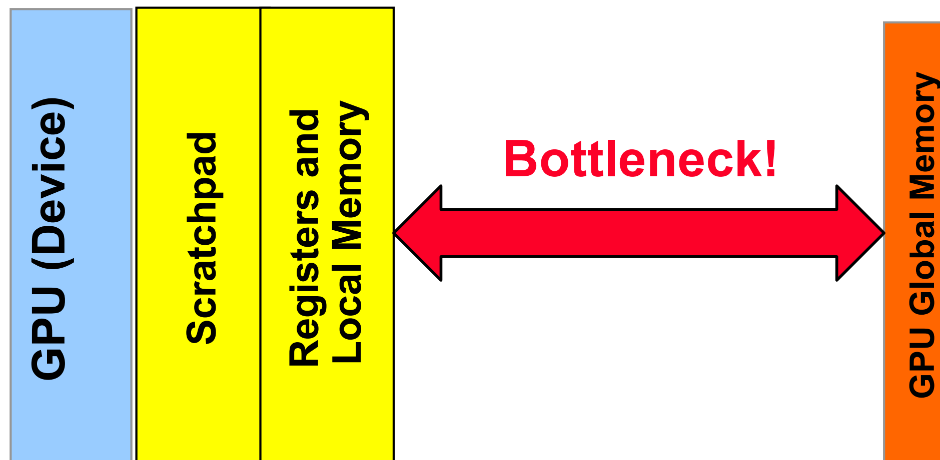
http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

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.

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

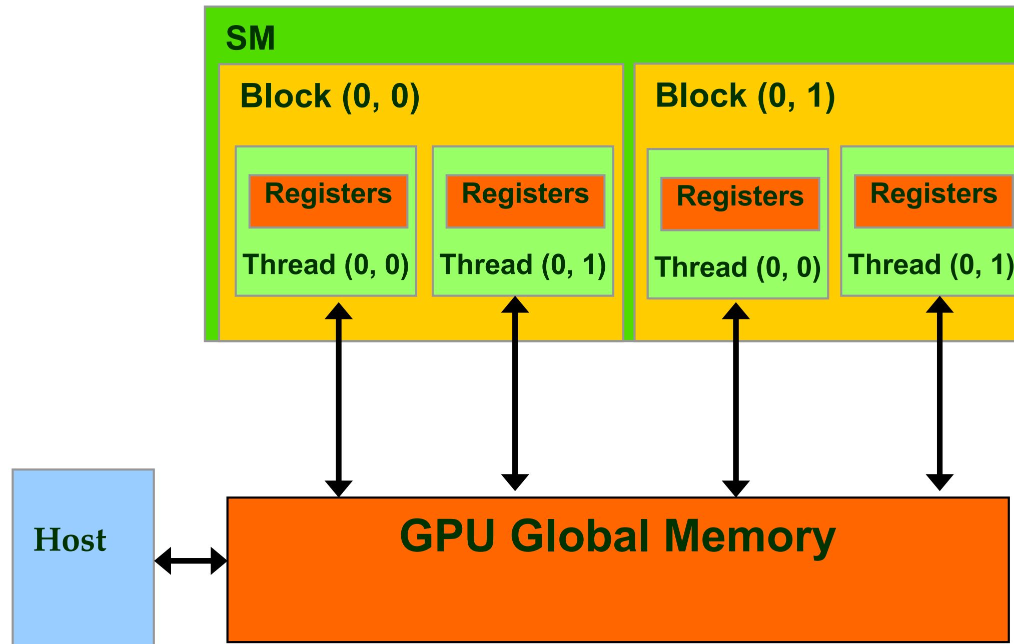


Let’s consider some software approaches first..
before moving on to hardware approaches

Background: GPU Memory Address Spaces

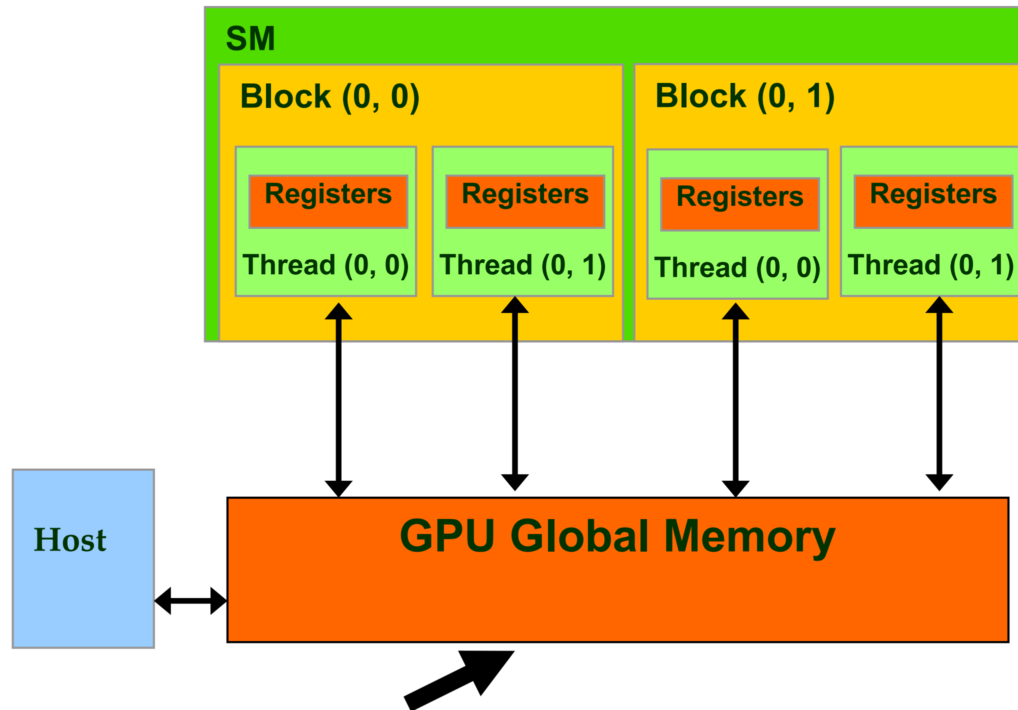
- GPU has three *address spaces* to support increasing visibility of data between threads: local, shared, global
- In addition two more (read-only) address spaces: Constant and texture.

Partial Overview of CUDA Memories



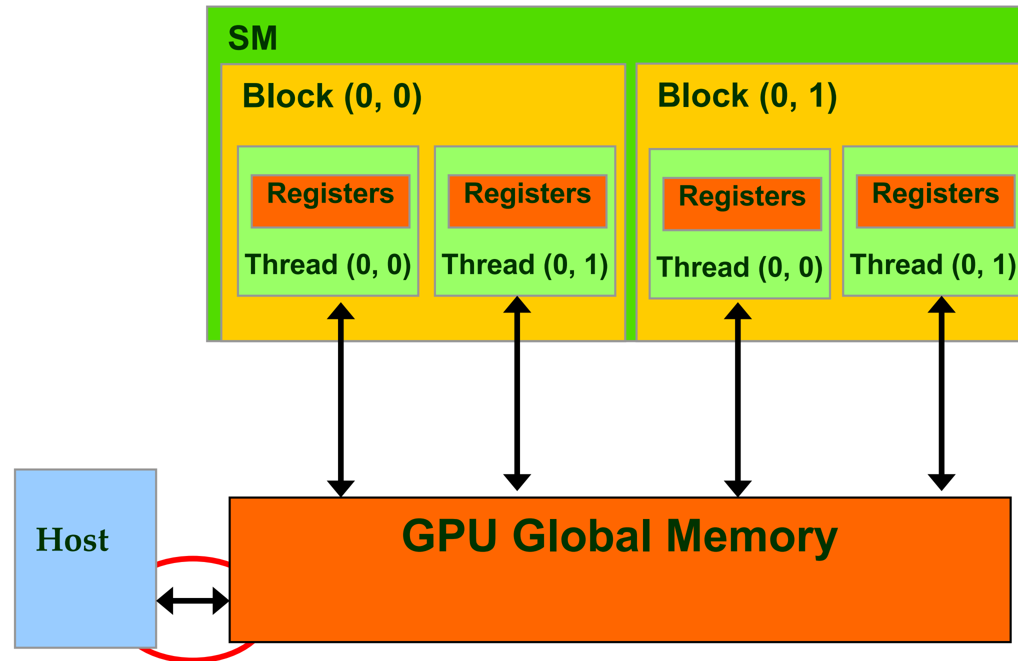
- Device code can:
  - R/W per-thread registers
  - R/W all-shared global memory
- Host code can
  - Transfer data to/from per grid global memory

CUDA Device Memory Management API functions



- cudaMalloc()
  - Allocates an object in the device global memory
  - Two parameters
    - **Address of a pointer** 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:

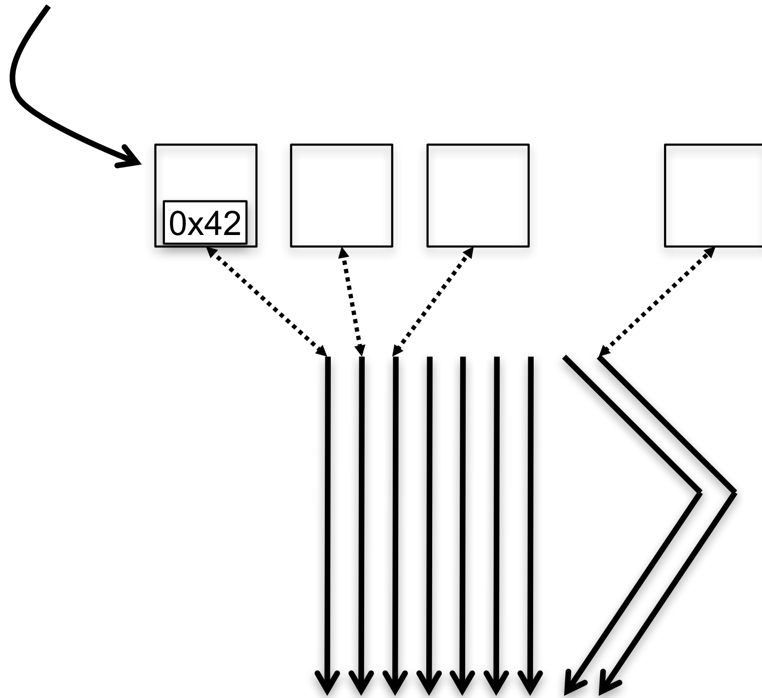
Transfer to device can be asynchronous

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

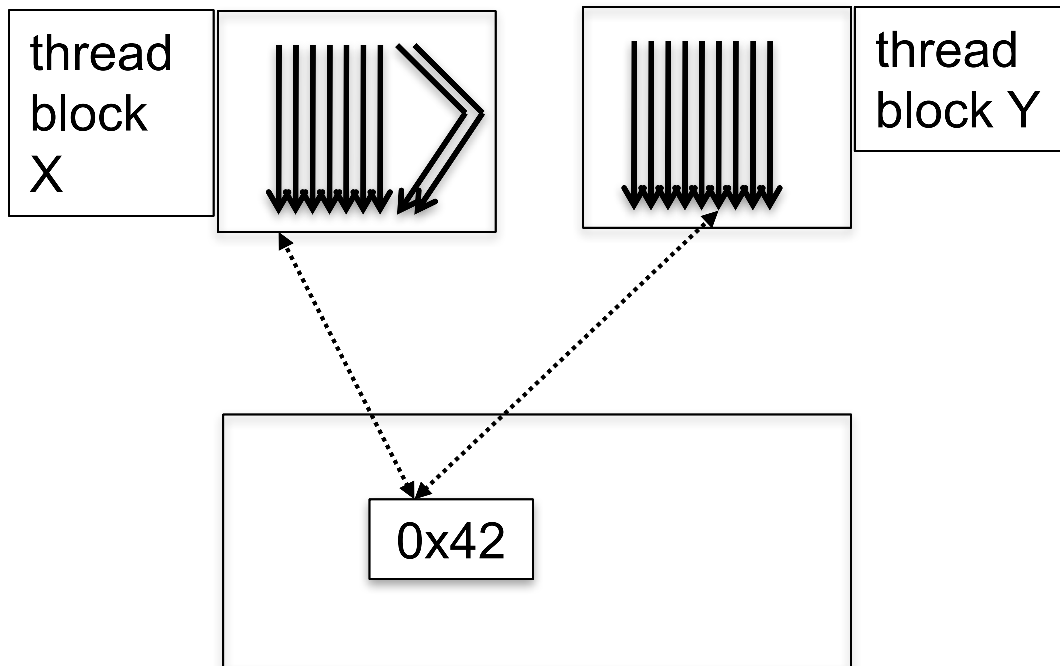
Each thread has own "local memory"



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



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

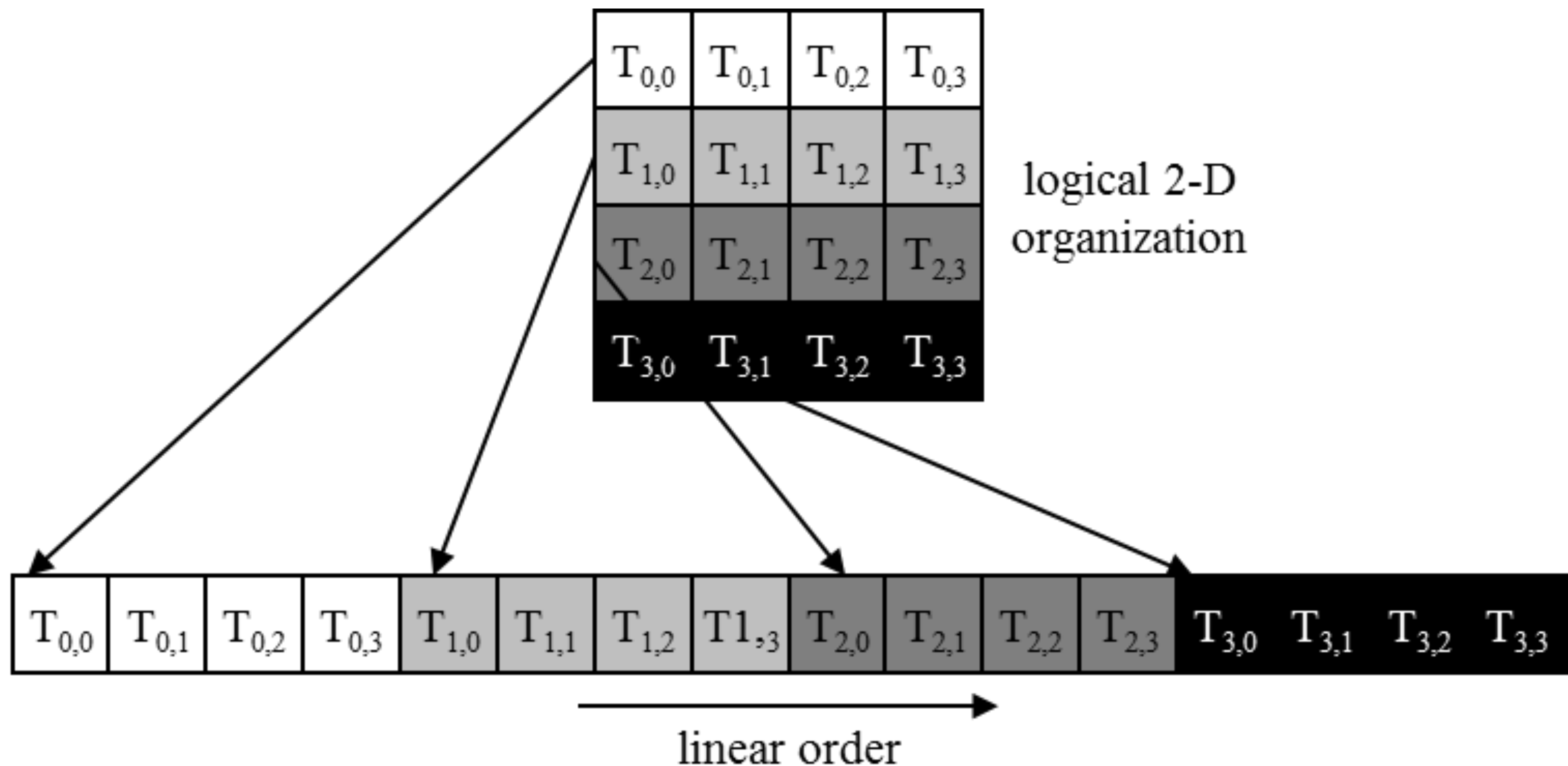
Commonly in GPGPU workloads threads write their own portion of global memory. Avoids need for synchronization—slow; also unpredictable thread block scheduling.

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.

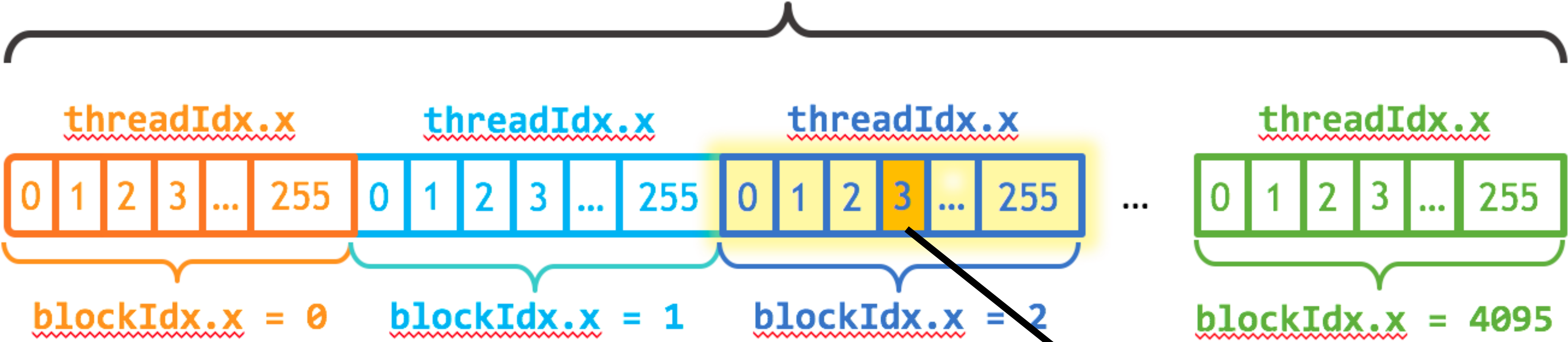
Warps in Multi-dimensional Thread Blocks

– 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

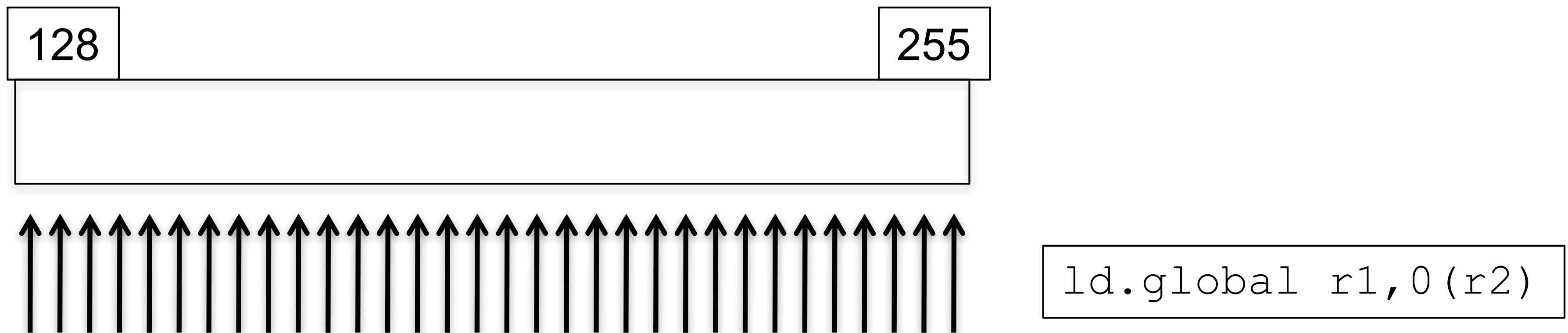
gridDim.x = 4096



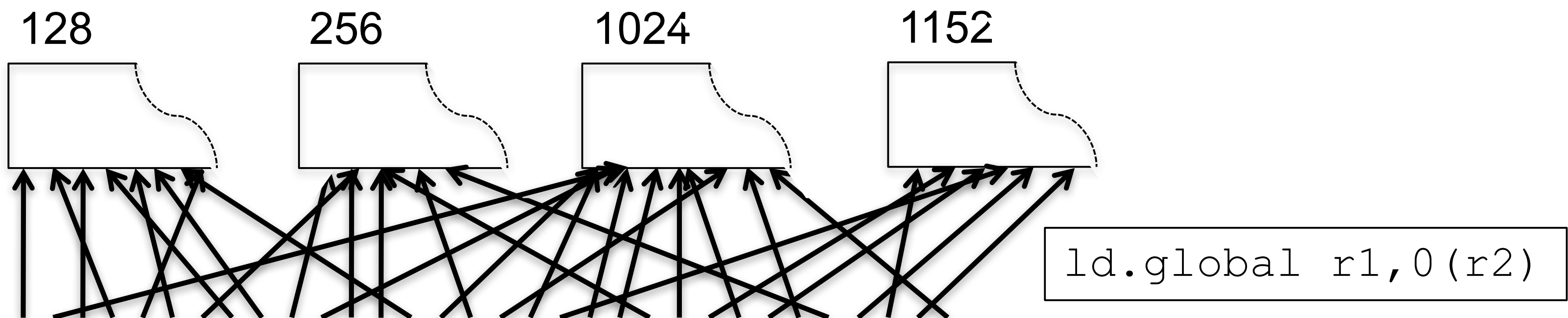

```
index = blockIdx.x   * blockDim.x + threadIdx.x

index =     (2)      *   (256)    +     (3)       = 515
```

"Coalescing" global accesses

- Aligned accesses request single 128B cache blk

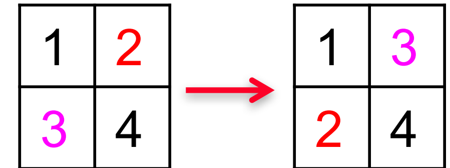


- Memory Divergence:



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];
  }
}
```

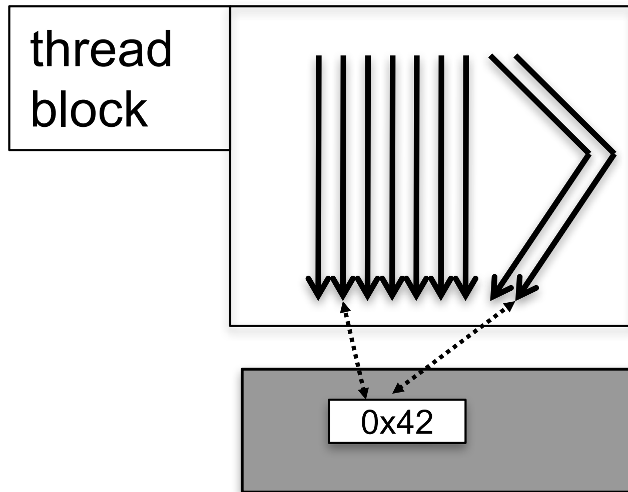


NOTE: "xIndex", "yIndex", "index_in", "index_out", and "i" are in local memory
(local variables are register allocated but stack lives in local memory)

"odata" and "idata" are pointers to global memory
(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)

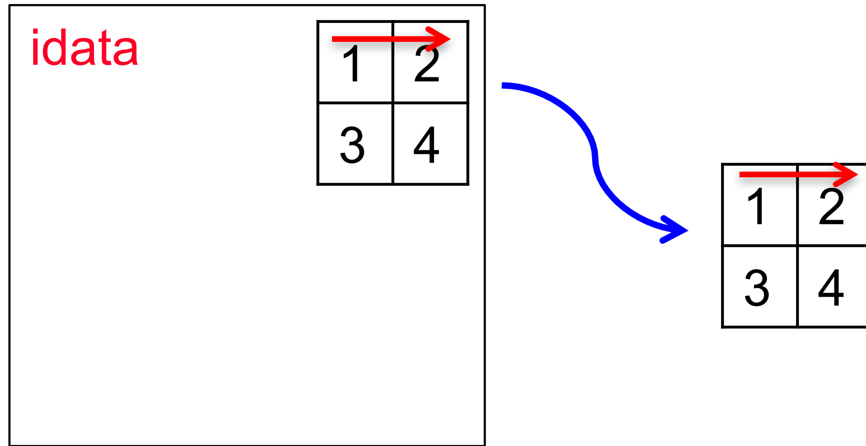
Shared memory address space is limited in size (16 to 48 KB).

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

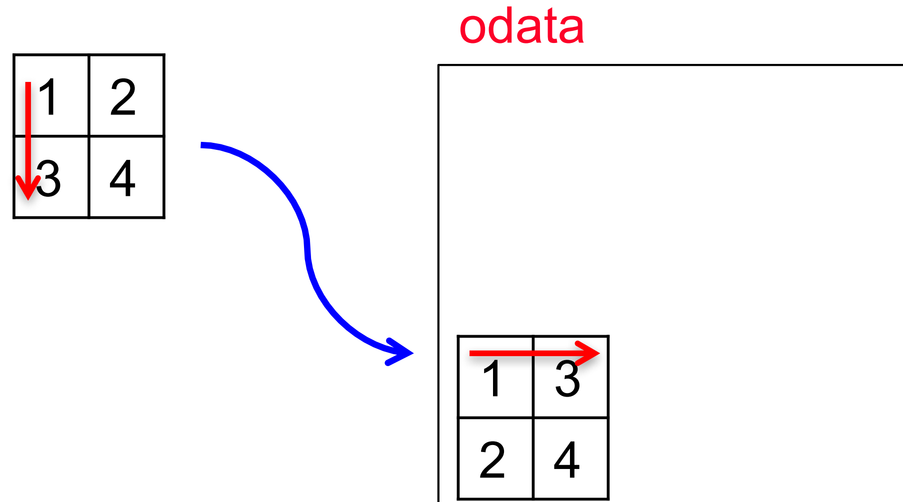
Synchronize threads in a thread block using __syncthreads();

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];
  }
}
```

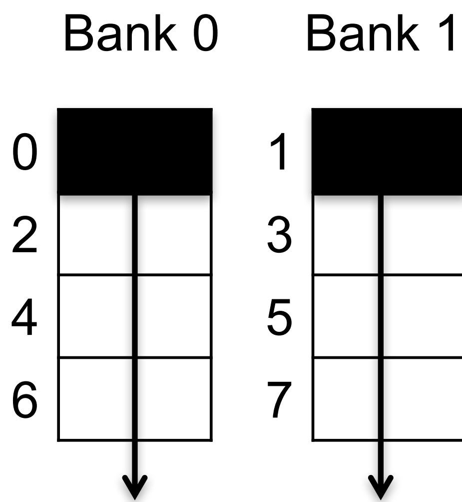
GOOD: Coalesced write

BAD: Shared memory bank conflicts

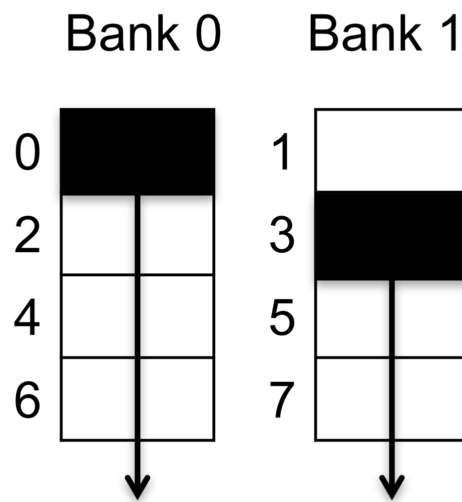
https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

Bank Conflicts

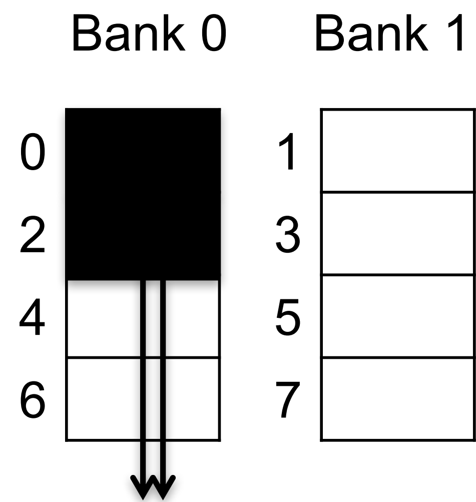
- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel



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



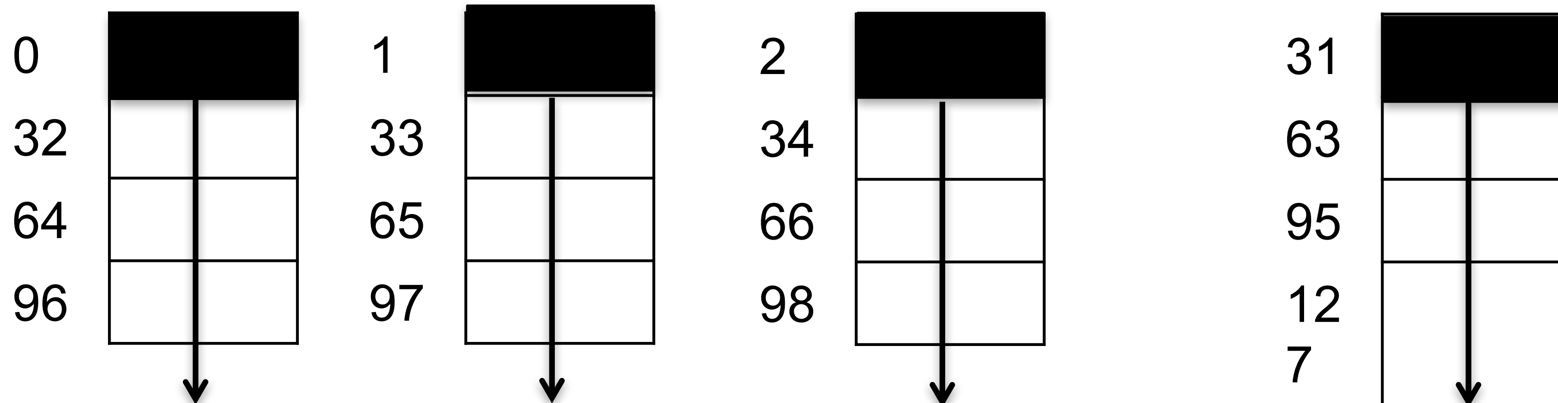
Example 2: Read 0, Read 3 (can proceed in parallel)



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

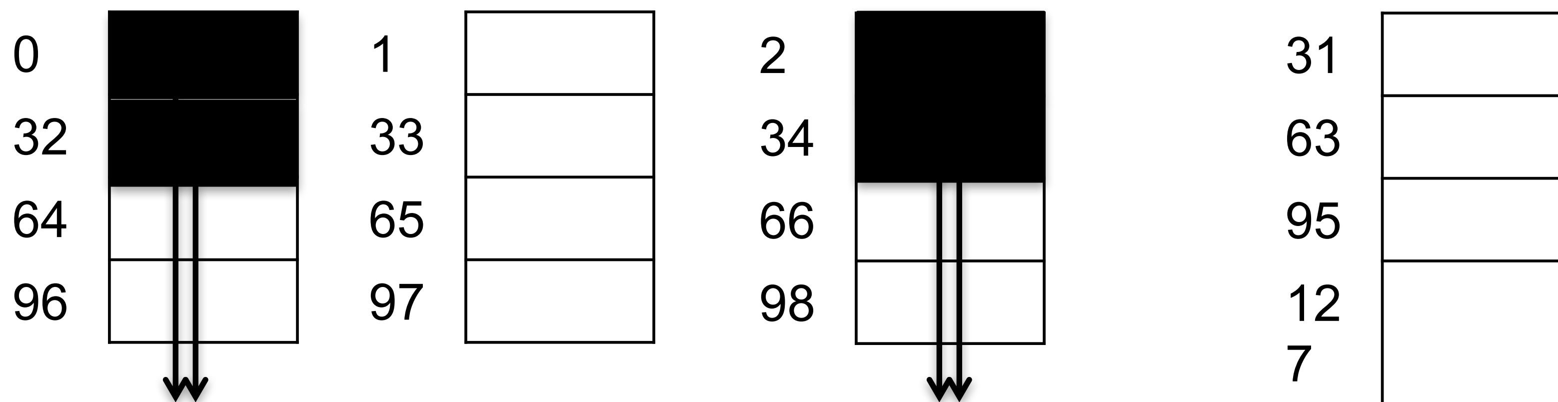
Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];

…

A[threadIdx.x] = … // no conflicts
```



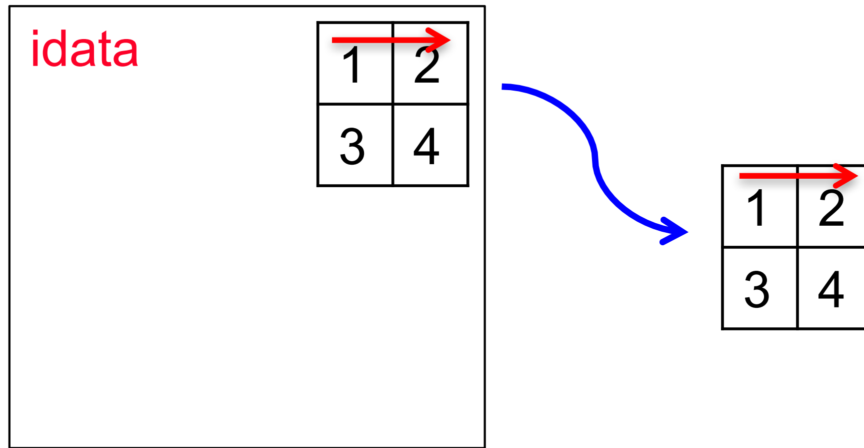
Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];

…

A[2*threadIdx.x] = // 2-way conflict
```

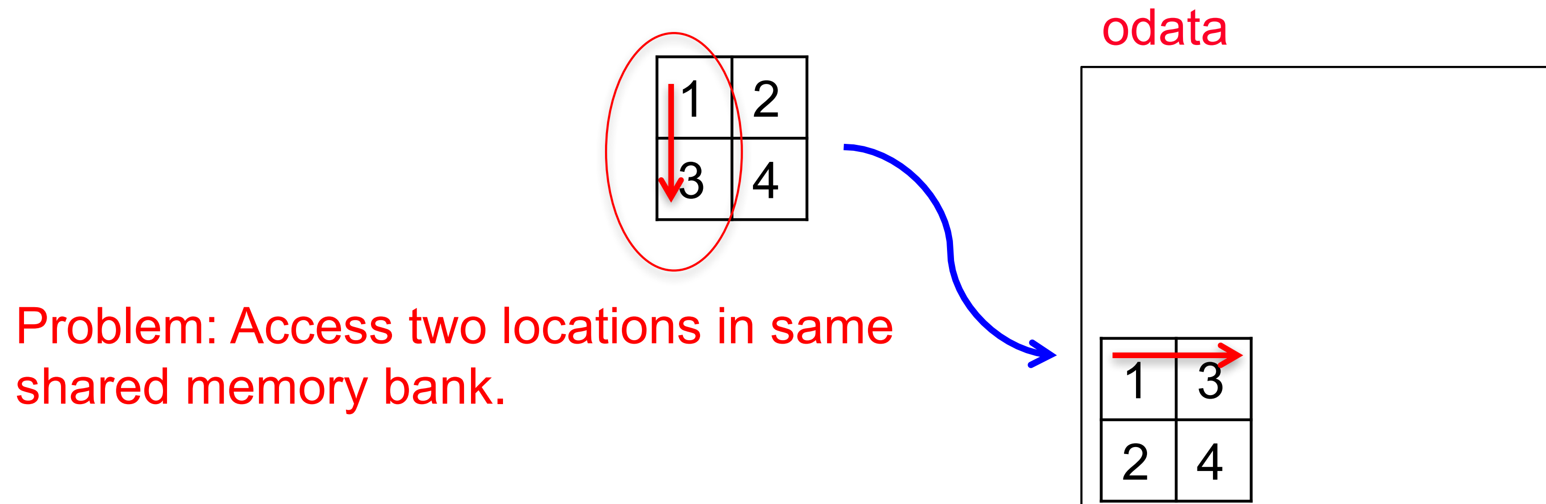


Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



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



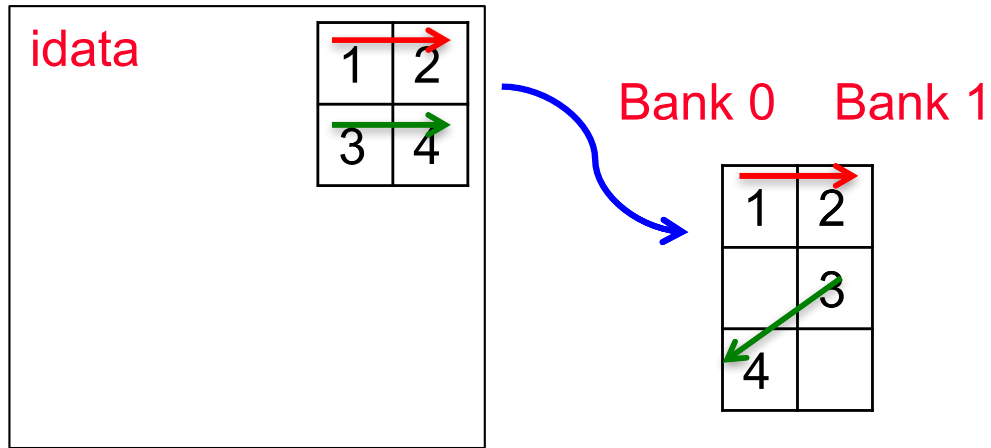
Eliminate Bank Conflicts

```
__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];
  }
}
```

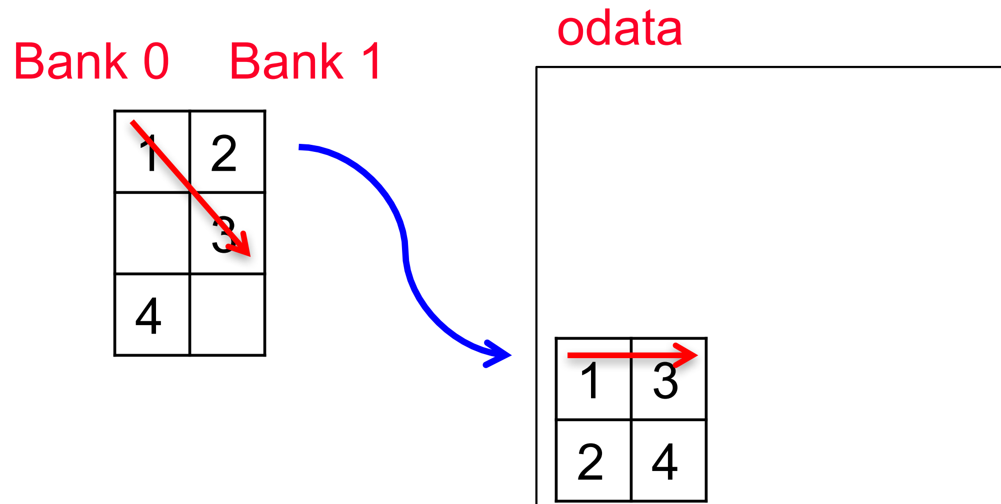
https://devblogs.nvidia.com/efficient-matrix-transpose-cuda-cc/

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



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



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