GPU Programming

Farshad Khunjush

Department of Computer Science and Engineering
Shiraz University

Fall 2025

Optimization Techniques

Farshad Khunjush

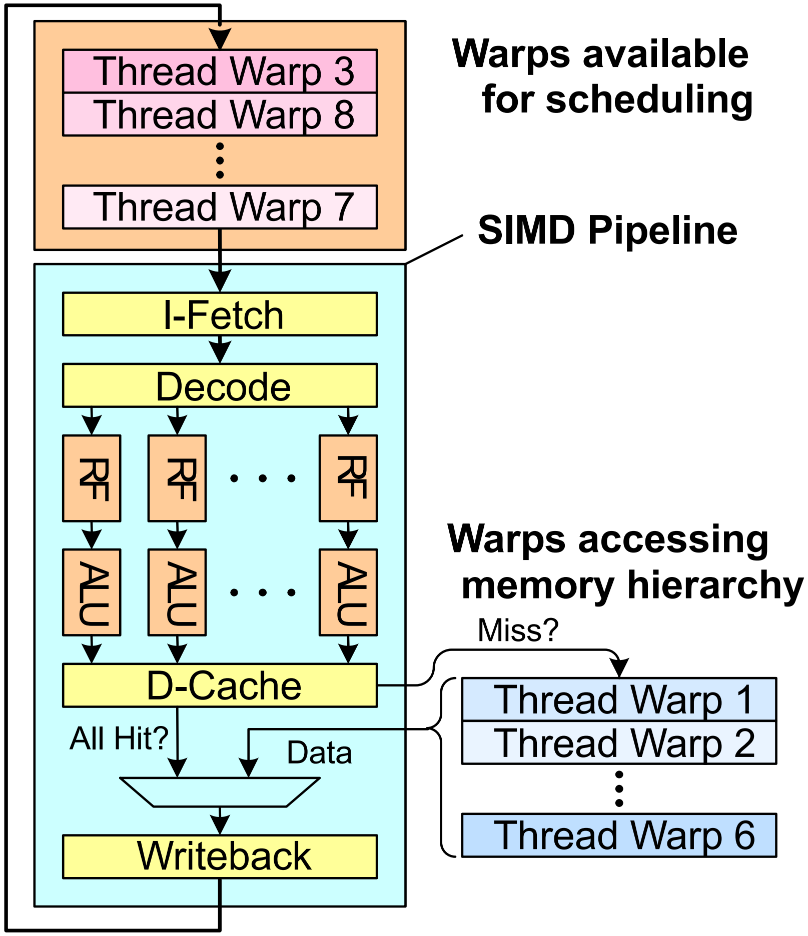
Performance Considerations

- Main bottlenecks
  - Global memory access
  - CPU-GPU data transfers
- Memory access
  - Latency hiding
    - Occupancy
  - Memory coalescing
  - Data reuse
    - Shared memory usage
- SIMD (Warp) Utilization: Divergence
- Other considerations
  - Atomic operations: Serialization
  - Data transfers between CPU and GPU
    - Overlap of communication and computation

Memory Access

Latency Hiding via Warp-Level FGMT

- Warp: A set of threads that execute the same instruction (on different data elements)
- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No interlocking)
  - Interleave warp execution to hide latencies
- Register values of all threads stay in register file
- FGMT enables long latency tolerance
  - Millions of pixels

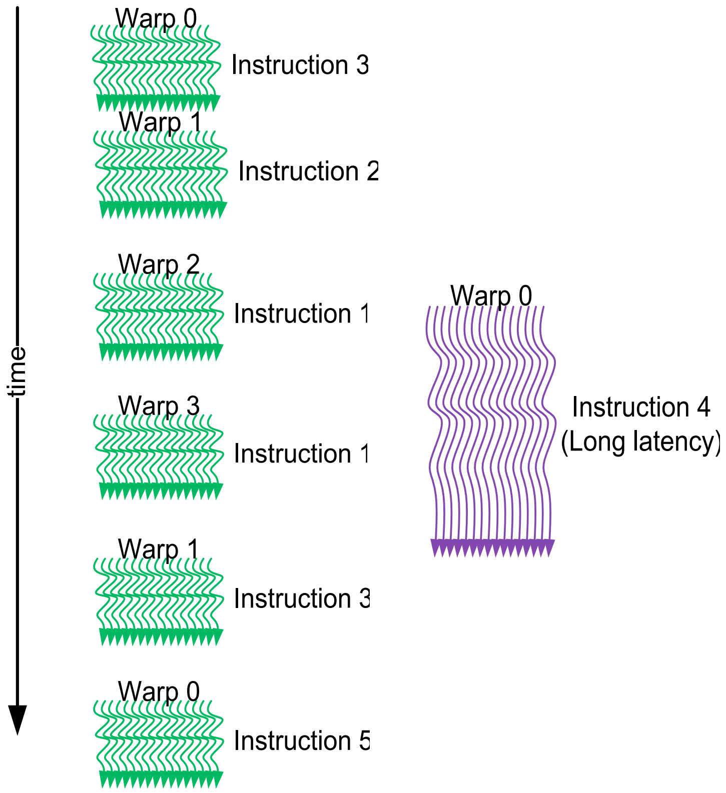


Slide credit: Tor Aamodt

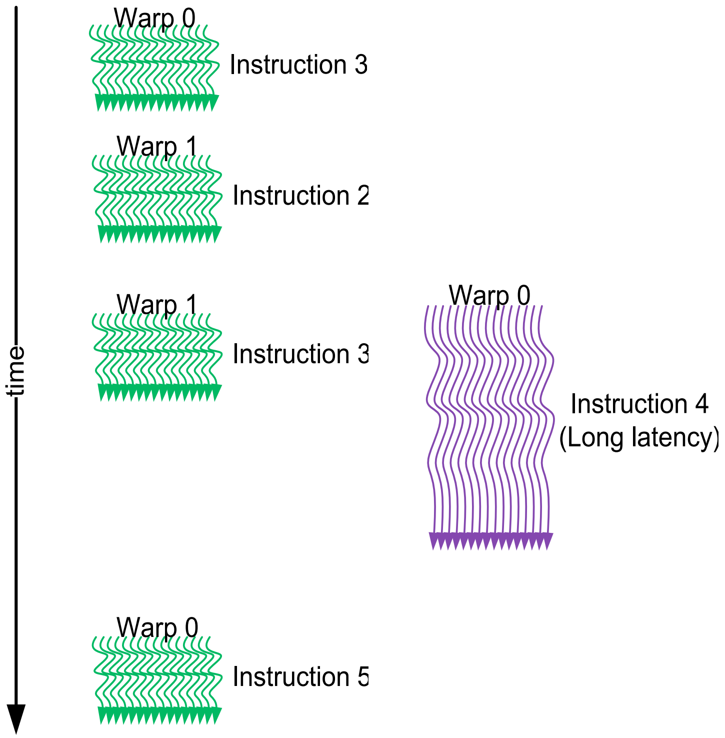
Latency Hiding and Occupancy

- FGMT can hide long latency operations (e.g., memory accesses)
- Occupancy: ratio of active warps to the maximum number of warps per GPU core

4 active warps



2 active warps

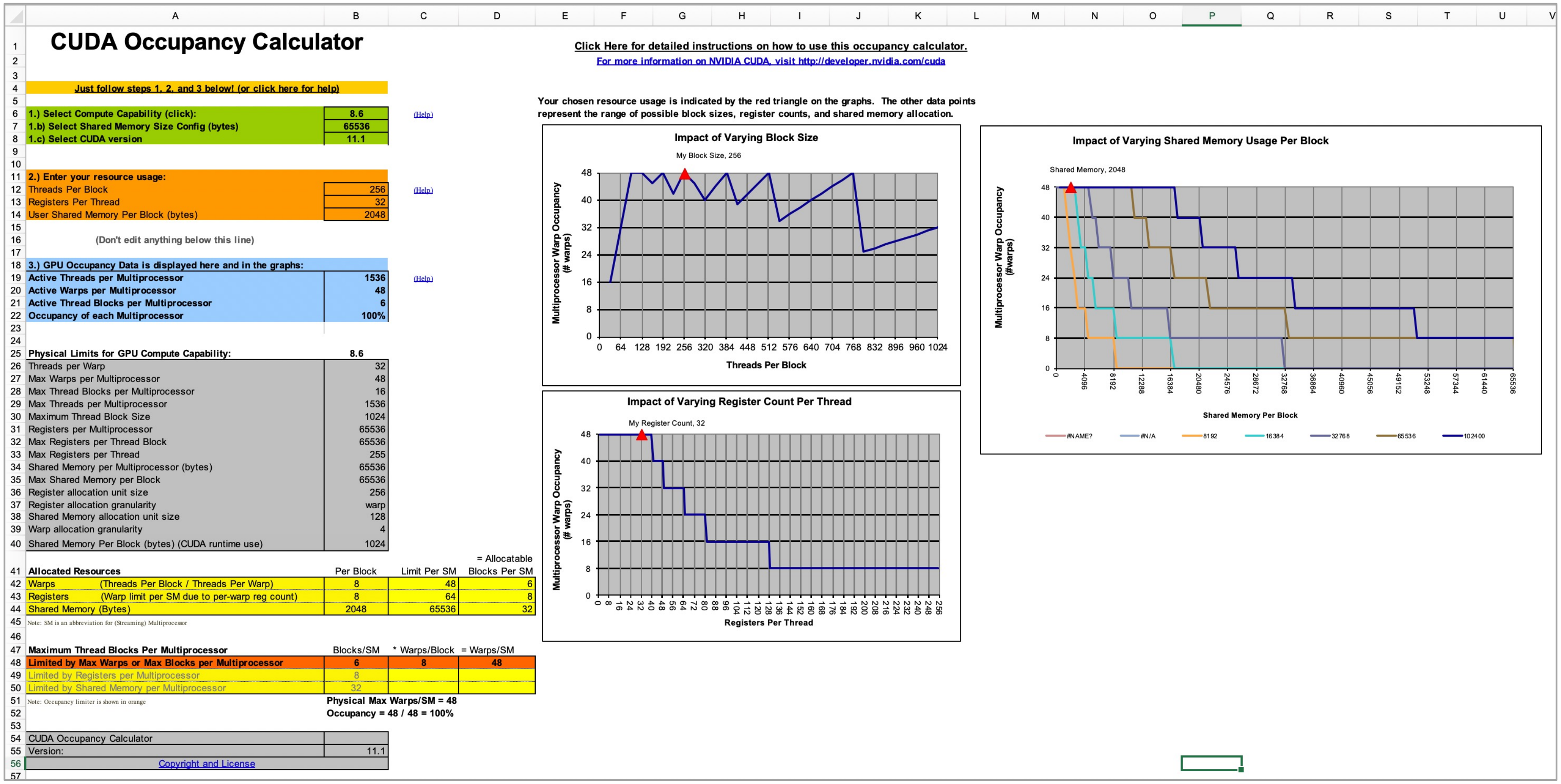


Occupancy

- GPU core, a.k.a. SM, resources (typical values)
  - Maximum number of warps per SM (64)
  - Maximum number of blocks per SM (32)
  - Register usage (256KB)
  - Shared memory usage (64KB)

- Occupancy calculation
  - Number of threads per block (defined by the programmer)
  - Registers per thread (known at compile time)
  - Shared memory per block (defined by the programmer)

CUDA Occupancy Calculator (I)



CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11.1	

2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1536	(Help)
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	8.6
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

				= Allocatable
Allocated Resources		Per Block	Limit Per SM	Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	8	48	6
Registers	(Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)		2048	65536	32

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48
Limited by Registers per Multiprocessor	8		
Limited by Shared Memory per Multiprocessor	32		

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 48
Occupancy = 48 / 48 = 100%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

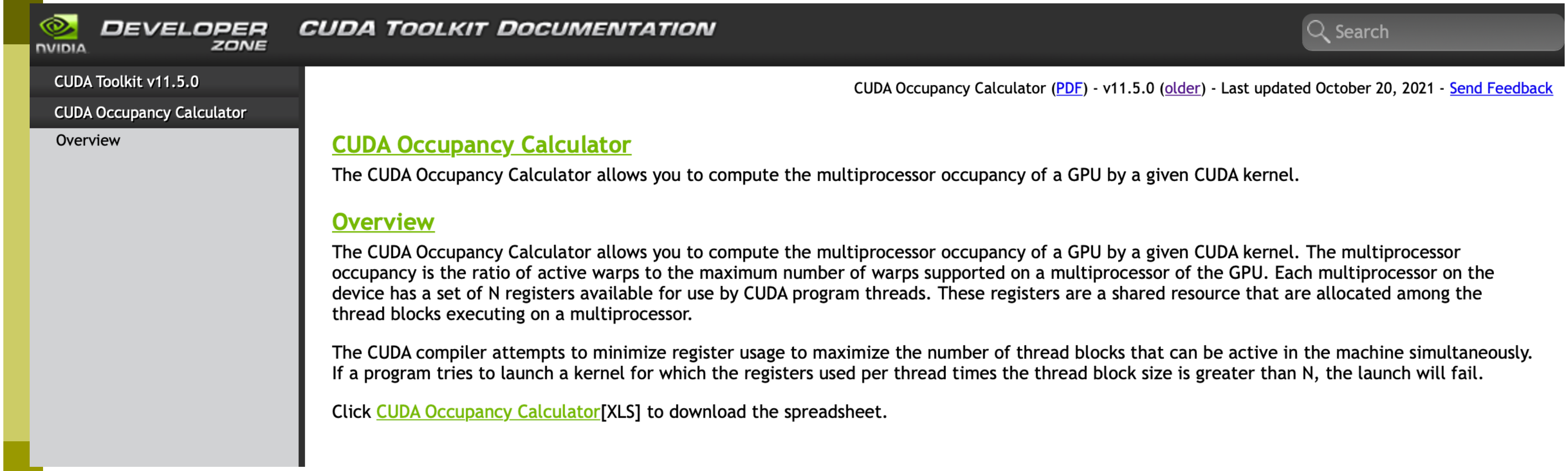
Click Here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls

CUDA Occupancy Calculator (II)



CUDA Occupancy Calculator (PDF) - v11.5.0 (older) - Last updated October 20, 2021 - Send Feedback

CUDA Occupancy Calculator

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel.

Overview

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. Each multiprocessor on the device has a set of N registers available for use by CUDA program threads. These registers are a shared resource that are allocated among the thread blocks executing on a multiprocessor.

The CUDA compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. If a program tries to launch a kernel for which the registers used per thread times the thread block size is greater than N, the launch will fail.

Click CUDA Occupancy Calculator[XLS] to download the spreadsheet.

https://docs.nvidia.com/cuda/cuda-occupancy-calculator/CUDA_Occupancy_Calculator.xls

Optimization: Algorithm

- Maximize parallelism
  - Avoid thread sync. if possible
  - Minimize control-flow divergence

- Maximize use of available memory bandwidth
  - GPUs have four types of memory (more later)

- Maximize arithmetic intensity (compute/access)
  - GPU spends its transistors on ALUs, not memory
    - Re-compute rather than cache
  - Avoid CPU-GPU data transfer:
    - Do more computation on GPU
    - Or, compute on CPU if low parallelism

GPU Memory System

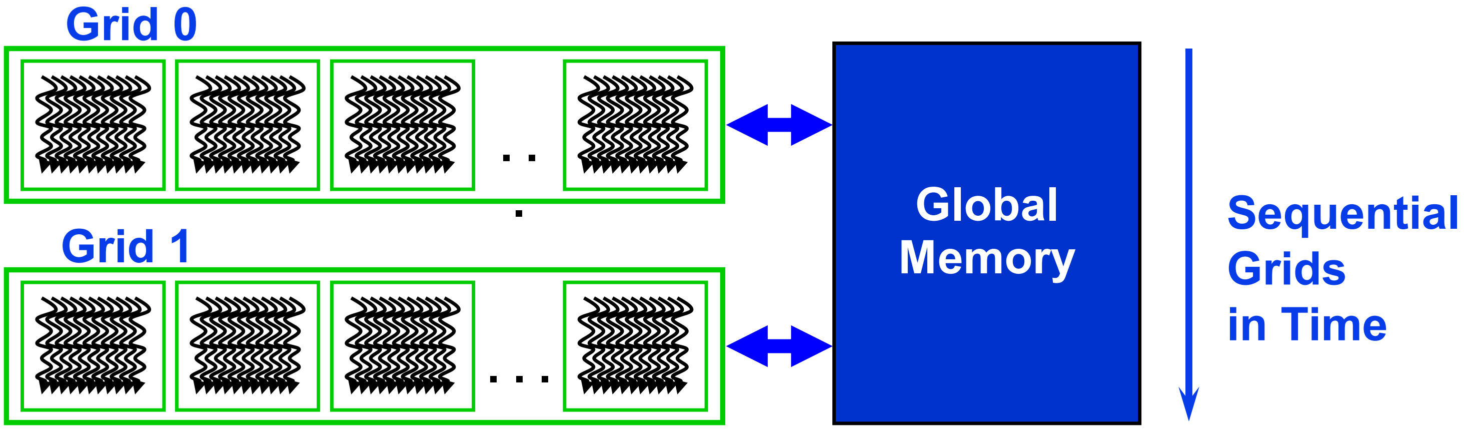
- Global and local memories
- Shared memory and L1 cache
- Registers
- Constant & Texture memories ◆ L2 cache

GPU’s Heterogeneous Memory Structures

- Designed originally for graphics computing
  - Specialized storage for various graphics data
  - Global, Constant, Texture
- Caches
  - For bandwidth improvement
  - Not latency (unlike CPU’s)
  - Not always coherent
- Memory hierarchy
  - Localized connectivity (improve bandwidth)

GPU Memory Hierarchy: Global Memory

- Global memory (per application):
  - Shared by all threads
  - GPU’s main memory (separate HW from GPU core)
  - ~10GB, ~300 GB/s of BW and latency of ~400 cycles
  - Inter-grid communication



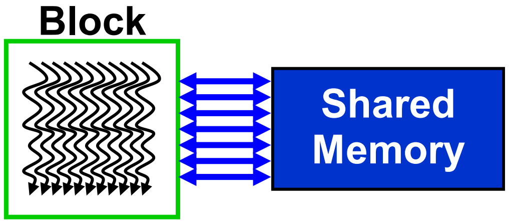
GPU Memory Hierarchy: Local Memory

- Local memory (per thread):
  - Private per thread
  - Everything on the stack that can’t fit in registers
    - Register spilling
  - Stored in global memory
    - Same latency as global memory
    - Much slower than registers!

Thread
Local Memory

GPU Memory Hierarchy: Shared Memory

- Shared memory (per block):
  - Shared within a thread block
  - **Managed by the programmer**
  - Very fast, located in the SM
    - Latency: ~5ns
    - Bandwidth: ~1 TB/s
  - Same HW as L1 cache (64KB)
    - 16/32/48KB of L1 cache
  - Inter-thread communication



GPU Memory Hierarchy: Caches

- L1 Cache:
  - Each SM has its own L1 cache
  - Older gen.: caches local & global memory
  - Newer gen.: only cache local memory
  - Same HW as shared memory
    - Configurable size: 16/32/48KB
- L2 Cache:
  - Shared by all SM’s
  - Caches all global & local memory accesses
  - ~1MB size, ~500 GB/s of BW

GPU Memory Hierarchy: Register File

- Registers:
  - Stack variables declared in kernels
  - Fastest access to data
    - ~10X faster than shared memory
  - Bandwidth: a few 10s of TB/s
  - Fundamental challenge in GPU microarchitecture
  - Example (Fermi architecture):
    - 32K 32-bit registers per SM
    - 48 warps per SM
      - 1536 threads per SM → 21 registers/thread
    - 2MB register file
      - 16 SMs→128KB per SM

GPU Memory Hierarchy: Constant & Texture

Historical leftover from graphics:

- Two more types of memory
  - But are not used that often
  - They are beneficial only for very specific types of apps
- Constant & Texture Memories:
  - Global with a special cache
  - Must be set from host before running kernel
    - Read-only over the course of a kernel execution
  - Can be used to reduce pressure on global memory

Optimization: Exploit Shared Memory

- Hundreds of times faster than global memory

- Use shared memory as a managed scratchpad
  - Bring data in from global memory
  - Operate in there (reuse)
  - Write results back to global memory

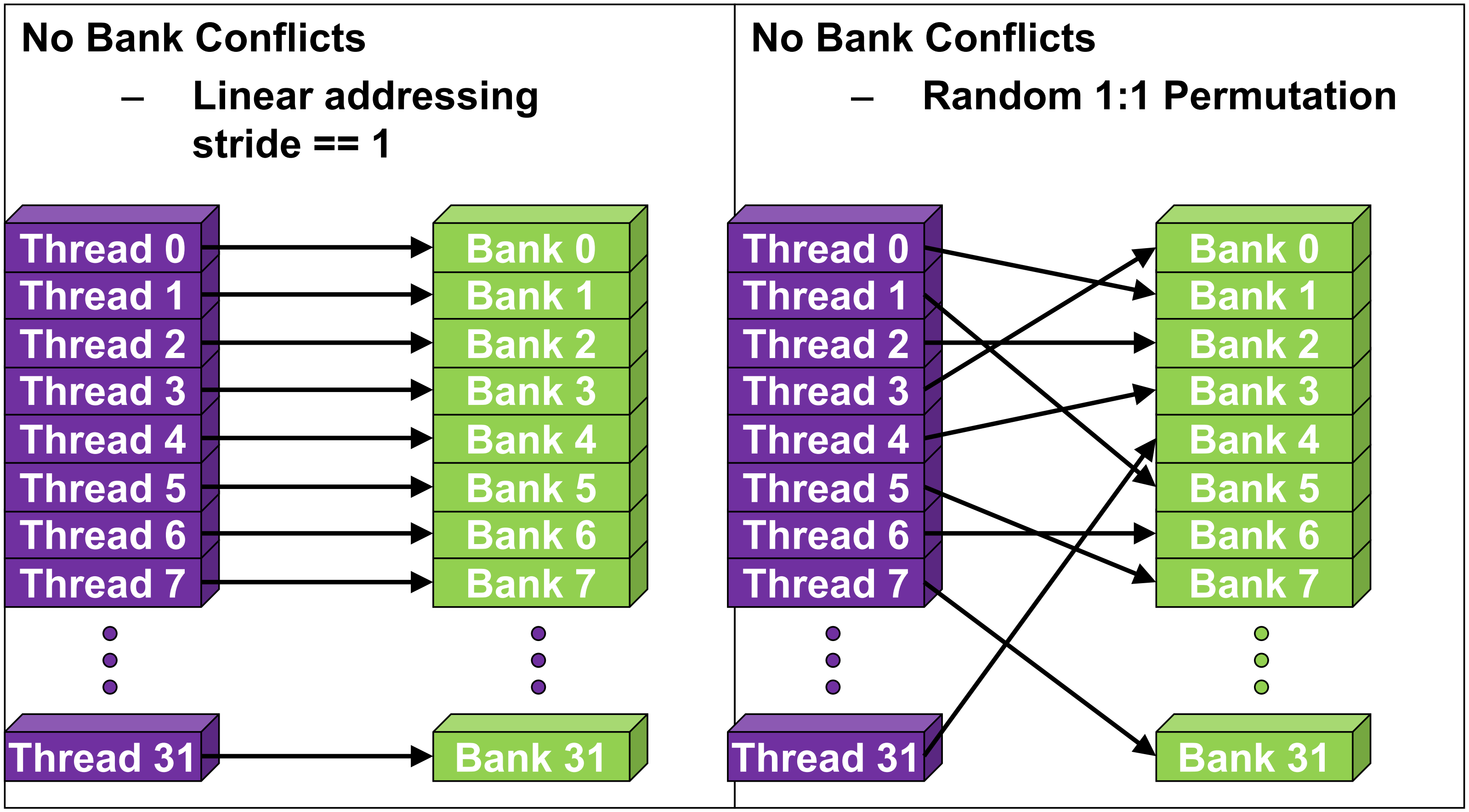
- Shared memory is fast as long as there are no bank conflicts

Shared Memory Bank Conflicts

- Shared memory is organized in 32 banks
  - Each bank can service one address at a time
  - At most 32 simultaneous accesses

- Multiple simultaneous accesses to the same bank
  - Different 4-byte words:
    - Bank conflict! - Conflicting accesses are serialized
  - The same 4-byte word:
    - Multicast – 1 fetch (could be different bytes within the word)

Shared Memory Bank Conflicts



No Bank Conflicts

– Linear addressing stride == 1

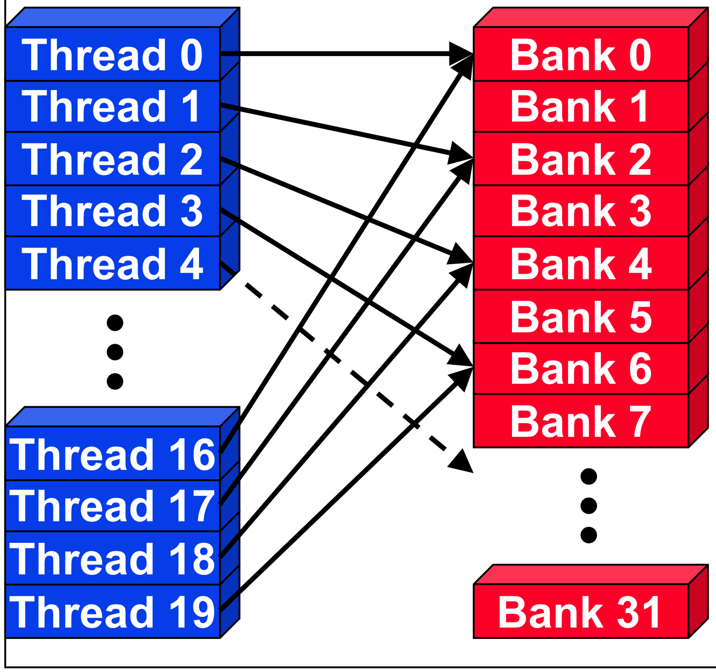
No Bank Conflicts

– Random 1:1 Permutation

Shared Memory Bank Conflicts

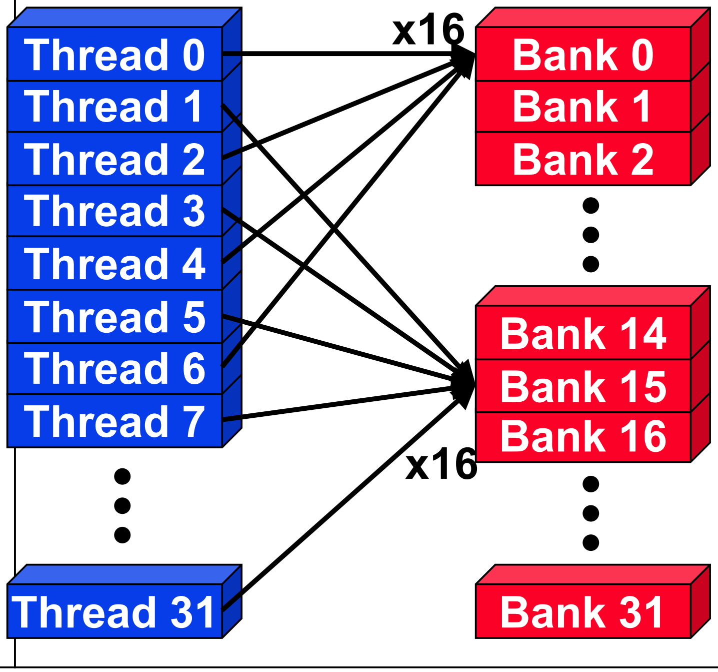
2-way Bank Conflicts

- **Linear addressing stride == 2**



16-way Bank Conflicts

- **Linear addressing stride == 16**

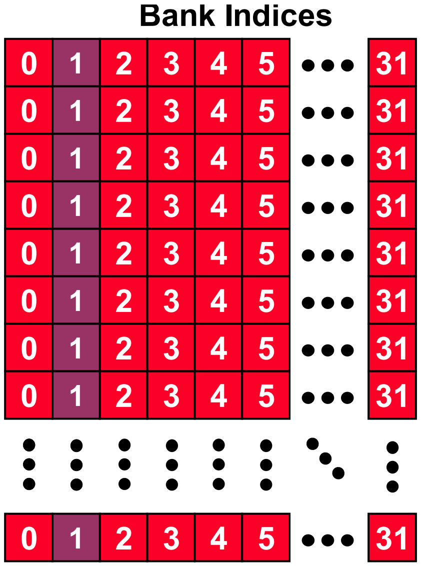


How Addresses are Mapped to Banks

- Each bank has a BW of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
  - This is called memory interleaving
- Modern GPUs have 32 banks
  - So bank = address % 32

Shared Memory Bank Conflicts - Example

- Operating on 2D array in shared memory
  - E.g., image processing
- Example: 32x32 block
  - Each thread processes a row
  - Threads in a block access the elements of a column simultaneously
    - E.g., Column 1 in purple
  - All 32 elements map to same bank
    - Elements are 32-bits apart in memory
    - 32-way bank conflict
    - 32 serialized accesses→SLOW!

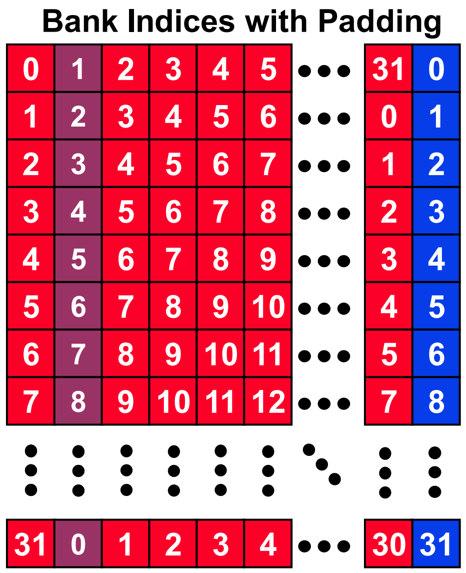


Bank Indices

0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
0	1	2	3	4	5	•••	31
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮
0	1	2	3	4	5	•••	31

Shared Memory Bank Conflicts - Example

- Solution 1: pad the rows
  - Add one element to the end of each row
  - Now each thread goes to a different bank
- Solution 2: transpose first
  - Suffer bank conflicts during transpose
  - But possibly save later conflicts
    - When passing multiple times over the same data
- Solution 3: operate on columns instead of rows
  - No bank conflict: one thread per bank Bank Indices with Padding

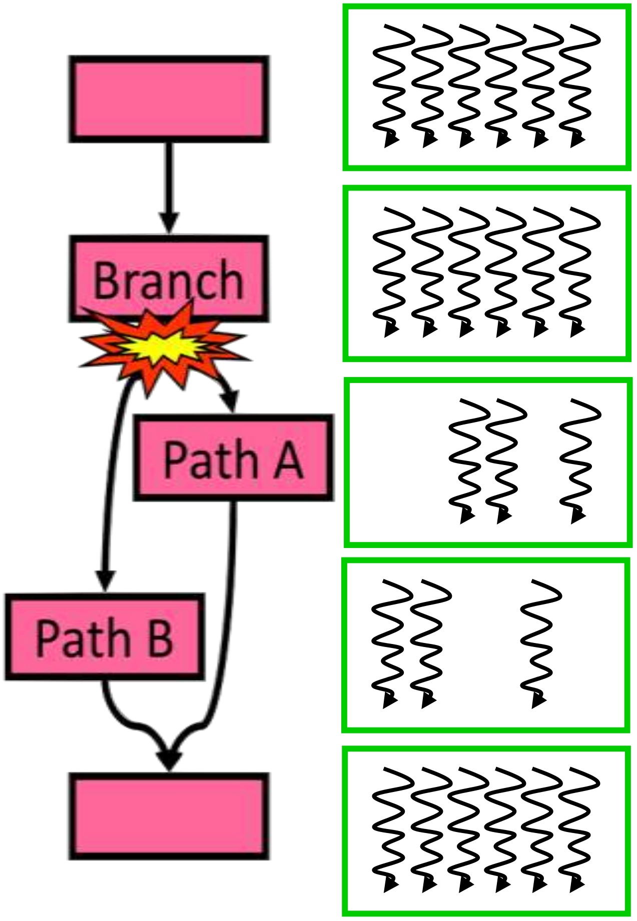


Bank Indices with Padding

0	1	2	3	4	5	•••	31	0
1	2	3	4	5	6	•••	0	1
2	3	4	5	6	7	•••	1	2
3	4	5	6	7	8	•••	2	3
4	5	6	7	8	9	•••	3	4
5	6	7	8	9	10	•••	4	5
6	7	8	9	10	11	•••	5	6
7	8	9	10	11	12	•••	7	8
⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮
31	0	1	2	3	4	•••	30	31

Control-Flow Divergence

- Threads in a warp might execute different paths
  - Branch instructions in the code
- Execute one path at a time
  - Diverging threads will be disabled
  - Limits parallelism and lowers performance



50% Performance Loss

Control-Flow Divergence

- A common case:
  - Avoid divergence when branch condition is a function of threadId (or tid)

- Example:

```
if (threadIdx.x > 2) {. . .}
```

  - Two different control paths for threads in a warp

Control-Flow Divergence

- Example that may look divergent but it’s not:

```
if (threadIdx.x / WARP_SIZE > 2) {. . .}
```

  - Two different control paths, different warps, same TB
    - All threads in a warp follow the same path
    - Divergent across warps is OK!