Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

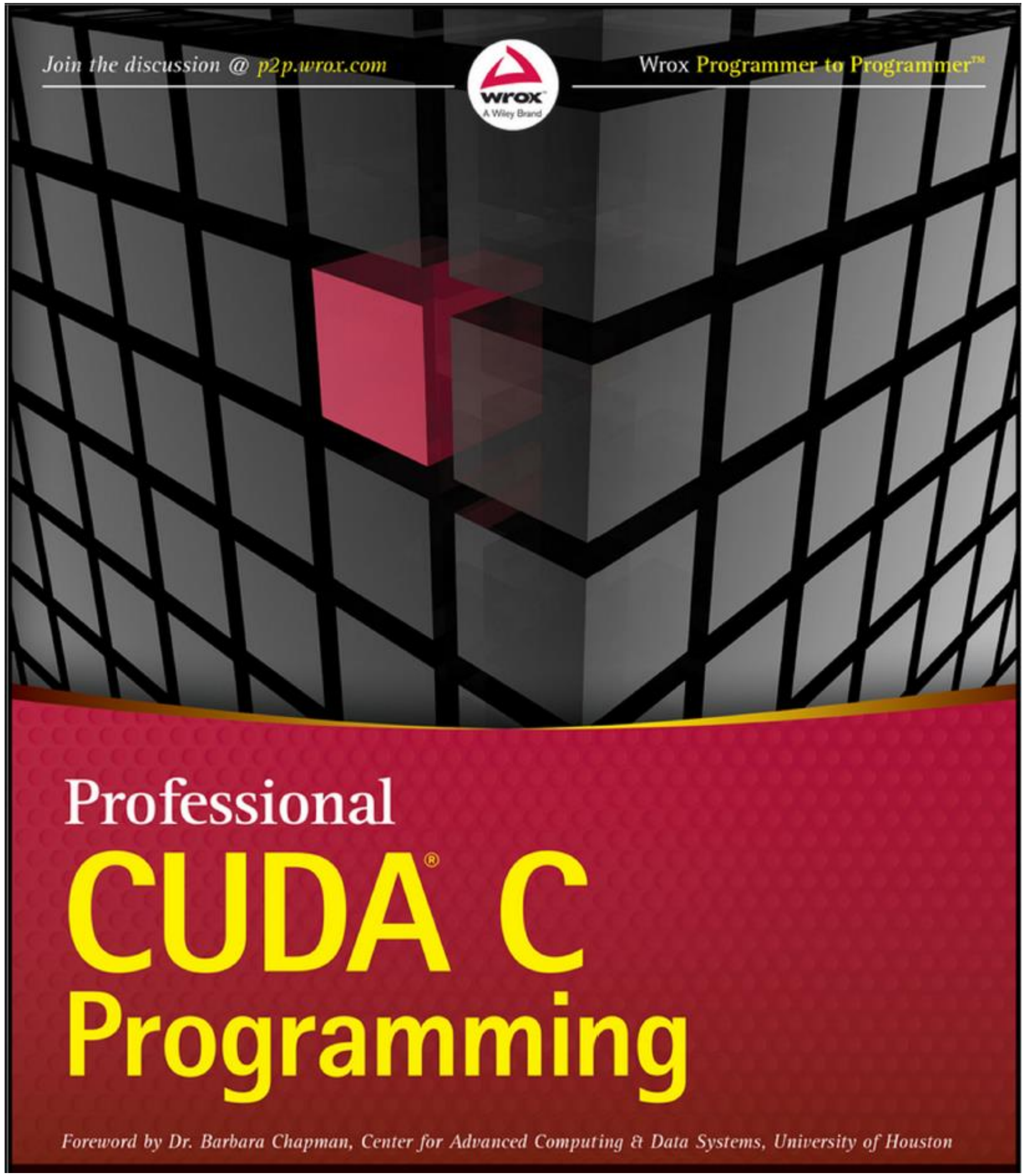
School of Electrical Engineering
and Computer Science

CEG 4136 Computer Architecture III

Fall 2024

Professor: Mohamed Ali Ibrahim, Ph.D., P. Eng.

Source:



Join the discussion @ p2p.wrox.com

Wrox Programmer to Programmer™

Professional CUDA® C Programming

Foreword by Dr. Barbara Chapman, Center for Advanced Computing & Data Systems, University of Houston

Chapter 4: Global Memory

Outline

- Learning the CUDA memory model
- Managing CUDA memory
- Programming with global memory
- Exploring global memory access patterns
- Probing global memory data layout
- Programming with unifi ed memory
- Maximizing global memory throughput

Introducing the CUDA Memory Model

- Key Point:
  - Memory access and management are crucial in programming, especially for high-performance computing with accelerators.

- Challenges:
  - Many workloads are limited by the speed of data load and storage.
  - High-performance memory is expensive and not always feasible.

- Solution:
  - CUDA memory model helps optimize latency and bandwidth by unifying host and device memory systems, providing full control over memory hierarchy.

Benefits of a Memory Hierarchy

- Principle of Locality:
  - Temporal Locality: Recently accessed data is likely to be accessed again soon.
  - Spatial Locality: Nearby memory locations are likely to be accessed when one is.

- Memory Hierarchy Structure:
  - Optimizes performance by storing data in lower-latency memory when actively used, and high-latency memory when not in use.
  - Balances cost, capacity, and latency across different memory levels.

Types of Memory in Hierarchy (Figure 4-1)

- Registers (smallest, fastest)
  - Most expensive and lowest capacity.
- Caches
  - Slightly larger, used for frequently accessed data.
- Main Memory
  - DRAM-based, larger capacity but higher latency.
- Disk Memory (biggest, slowest)
  - Used for long-term data storage with the largest capacity but slowest access speed.

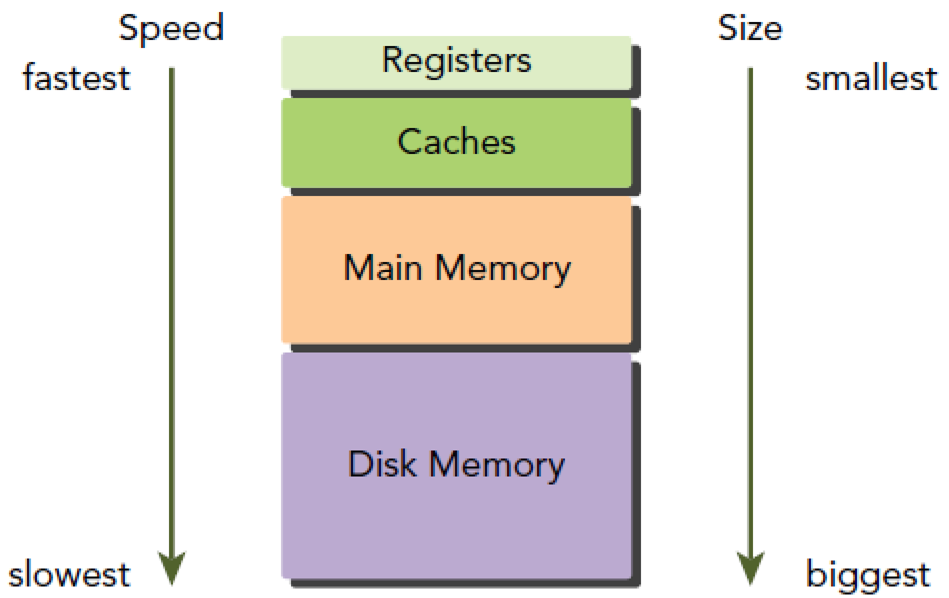


FIGURE 4-1

CUDA Memory Model Overview

- Two Classifications of Memory:
    - Programmable: Explicit control over data placement.
    - Non-programmable: Automatic data placement, no control by the programmer.

- CPU vs CUDA Memory:
    - CPU memory hierarchy includes non-programmable memory (e.g., L1 and L2 cache).
    - CUDA exposes many types of programmable memory, such as:
        - Registers
        - Shared memory
        - Local memory
        - Constant memory
        - Texture memory
        - Global memory

Memory Hierarchy in CUDA (Figure 4-2)

- Registers:
  - The fastest and smallest memory space, unique to each thread.
- Shared Memory:
  - Accessible to all threads within the same thread block.
- Local Memory:
  - Private memory space for individual threads.
- Global Memory, Constant Memory, Texture Memory:
  - Visible to all threads, optimized for different uses, and persistent across the lifetime of the application.

Memory Hierarchy in CUDA

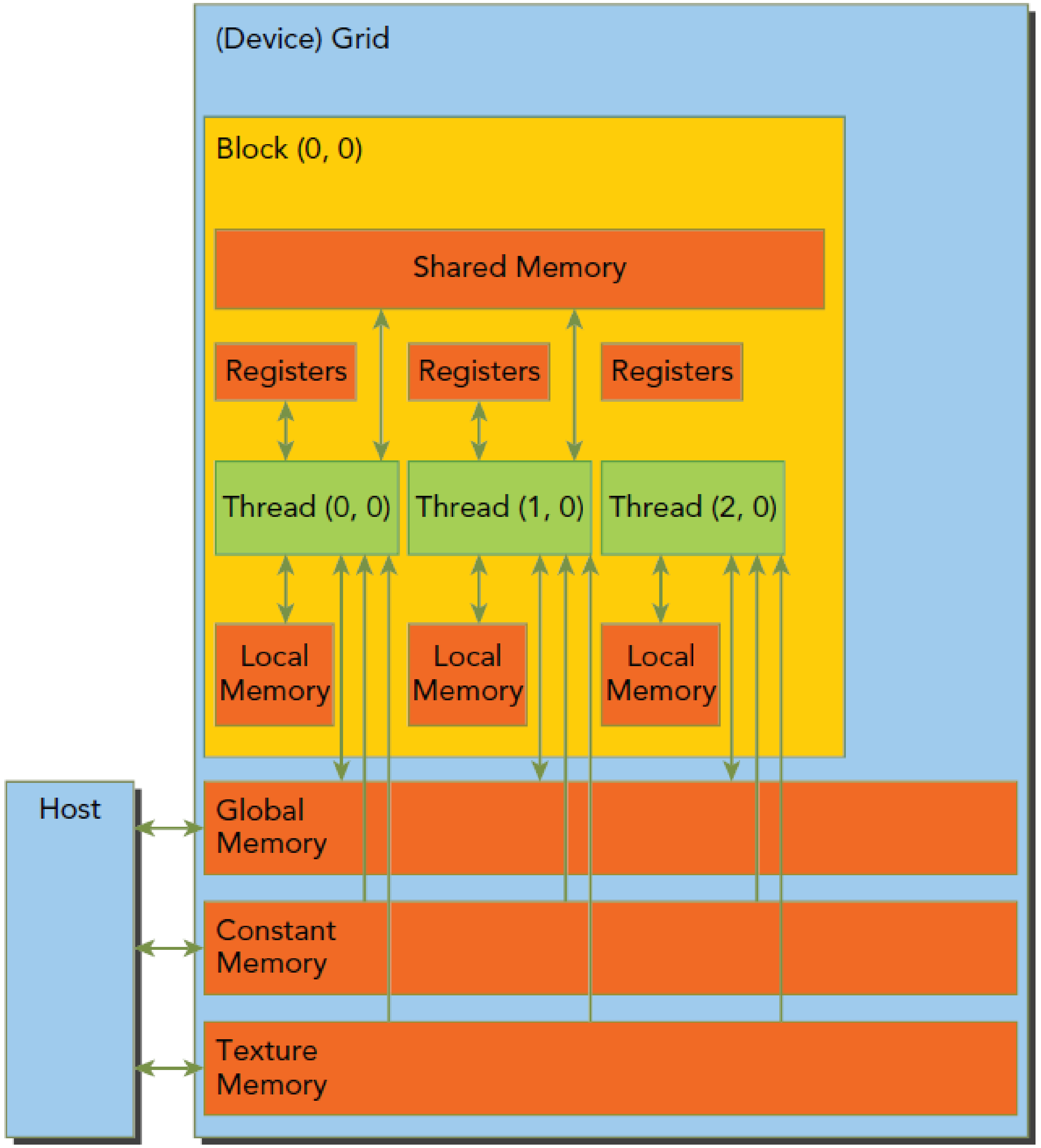


FIGURE 4-2

Registers in CUDA

- Fastest Memory Space:
  - Registers are the fastest memory on a GPU.
  - Used for storing automatic variables declared in kernels.

- Private to Threads:
  - Each thread has its own set of registers.
  - Registers share their lifetime with the kernel.

- Limitations and Resource Management:
  - Limited number of registers per thread.
  - Overuse can lead to register spilling (data is stored in slower local memory).
  - Optimizing register use can improve performance and thread block occupancy.

Controlling Register Usage in CUDA

- Register Spilling:
  - Occurs when more registers are needed than available.
  - Results in reduced performance as data spills into slower memory.

- Managing Register Usage:
  - Use the `-Xptxas` option to check register usage.
  - Control registers with launch bounds and compiler options like `-maxrregcount`.

- Example:

```
__global__ void kernel(...) {
  // kernel body
}
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor);
```

Constant Memory in CUDA

- Overview of Constant Memory:
  - Cached, read-only memory optimized for use by all threads in a warp.
  - Best used when all threads read from the same memory location.
- Declaration:
  - Constant variables are declared with the `__constant__` qualifier.
  - Can only be read and must be initialized by the host.
- Memory Access Optimization:
  - Works best when all threads in a warp read the same data to reduce memory transactions.

Texture Memory in CUDA

- What is Texture Memory?
  - Cached, read-only memory, optimized for 2D spatial locality.
  - Used for applications requiring interpolation, such as image processing.

- Performance Consideration:
  - Use texture memory for applications needing optimized access to 2D data.
  - Can be slower than global memory for certain applications.

Global Memory in CUDA

- Characteristics:
  - Largest and most commonly used memory on the GPU.
  - Highest latency, but accessible by all threads.

- Optimization Tips:
  - Ensure memory transactions are aligned (32, 64, 128 bytes).
  - Efficient global memory access depends on distributing memory addresses across threads and ensuring alignment per transaction.

- Best Practices:
  - Minimize the number of transactions per memory request.
  - Exploit data locality to improve throughput efficiency.

GPU Caches in CUDA

- Types of GPU Caches:
  - L1 and L2 caches
  - Read-only constant cache
  - Read-only texture cache

- Caching Mechanism:
  - L1/L2 caches are used for storing data in local/global memory.
  - Memory load operations can be cached, but store operations cannot.
  - Read-only constant and texture caches are optimized for respective memory spaces.

Events and metrics

In CUDA profiling:

- Events are countable activities tracked via hardware counters during kernel execution.
- Metrics are characteristics of a kernel, calculated based on one or more events.

Key points:

- Most counters are reported per streaming multiprocessor (not the entire GPU).
- A single profiling run can collect only a few counters, and some counters are mutually exclusive, requiring multiple runs to gather all necessary data.
- Counter values may vary across runs due to GPU execution differences, such as thread block and warp scheduling.

CUDA Variable Declarations

- CUDA Variable Types:
  - Register: Stored per thread.
  - Local: Stored per thread, may spill into local memory.
  - Shared: Accessible by all threads in a block.
  - Global: Visible to all threads across the device.
  - Constant: Read-only, cached memory for all threads.

- Memory Characteristics (Table 4-2):
  - Registers: On-chip, thread scope, fast access.
  - Local: Off-chip, thread scope, slower access.
  - Shared: On-chip, block scope, used for inter-thread communication.
  - Global/Constant/Texture: Global scope, cached for efficiency.

The principal traits of the various memory types are summarized in Table 4-2.

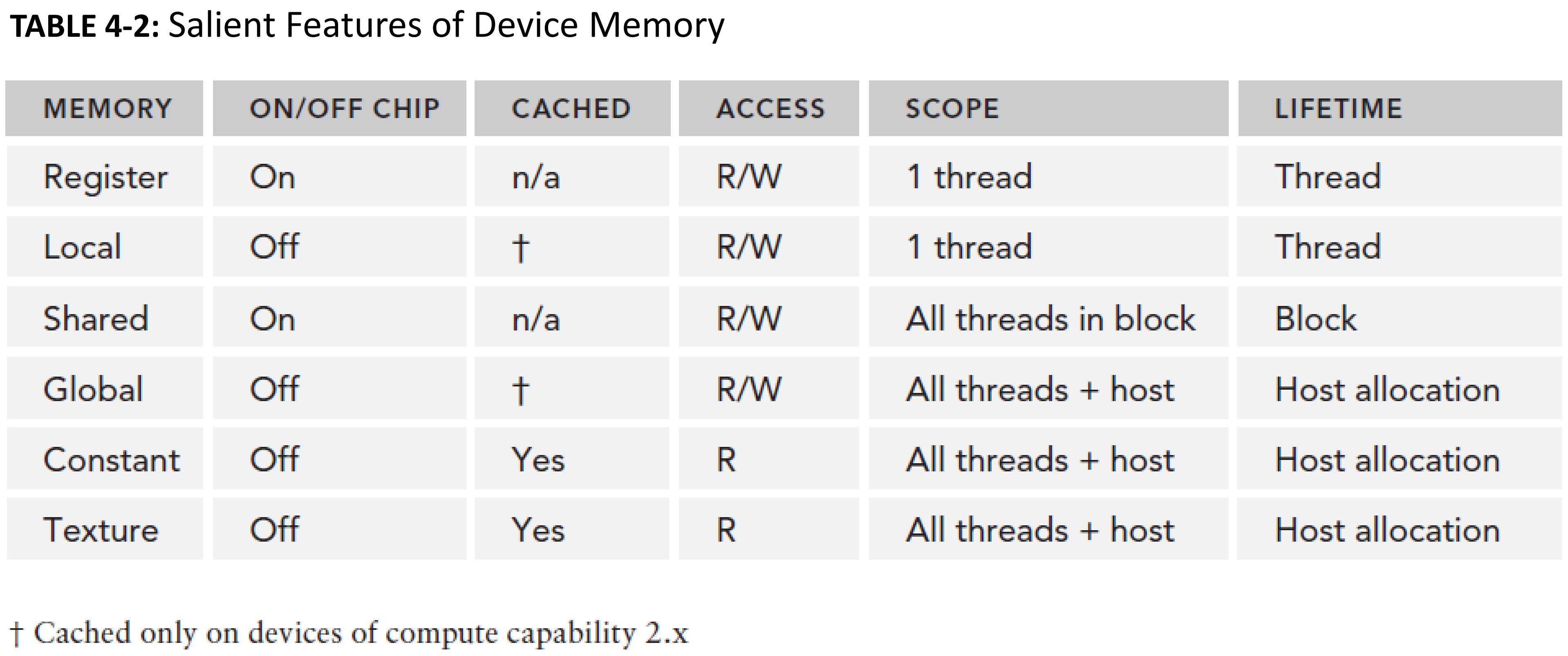


TABLE 4-2: Salient Features of Device Memory

MEMORY	ON/OFF CHIP	CACHED	ACCESS	SCOPE	LIFETIME
Register	On	n/a	R/W	1 thread	Thread
Local	Off	†	R/W	1 thread	Thread
Shared	On	n/a	R/W	All threads in block	Block
Global	Off	†	R/W	All threads + host	Host allocation
Constant	Off	Yes	R	All threads + host	Host allocation
Texture	Off	Yes	R	All threads + host	Host allocation

† Cached only on devices of compute capability 2.x

Static Global Memory Declaration (Example)

- Static Global Variable Example:

```
__device__ float devData;

__global__ void checkGlobalVariable() {
    printf("Value: %f", devData);
    devData += 2.0f;
}
```

Example_1_globaleVariable

- Initialization:
  - Use `cudaMemcpyToSymbol` to initialize global variables from the host side.

- Host and Device Code Interaction:
  - Host code and device code cannot directly access each other's variables.
  - Global variables in device memory can be accessed using API calls.

Accessing Device Memory from Host

- Accessing Device Global Variables:
  - Use `cudaMemcpyToSymbol` or `cudaGetSymbolAddress` to access global variables from the host.

- Pointer Usage:
  - The `cudaMemcpy` function can transfer data to the global memory using pointers.

- Key Notes:
  - Host cannot directly dereference device pointers.
  - Pinned memory is required for direct host-device memory access without CPU intervention.

Memory Transfer in CUDA

- cudaMemcpy Function
  - Purpose: Transfers data between host and device.
  - Function Signature:

```
cudaError_t cudaMemcpy(void *dst, const void
*src, size_t count, enum cudaMemcpyKind
kind);
```

- Parameters:
  - dst: Destination pointer.
  - src: Source pointer.
  - count: Number of bytes to copy.
  - kind: Specifies the direction of the transfer, can be:
    - `cudaMemcpyHostToHost`
    - `cudaMemcpyHostToDevice`
    - `cudaMemcpyDeviceToHost`
    - `cudaMemcpyDeviceToDevice`

- Synchronization: The function is synchronous, meaning data is transferred immediately, blocking execution until the transfer is complete.

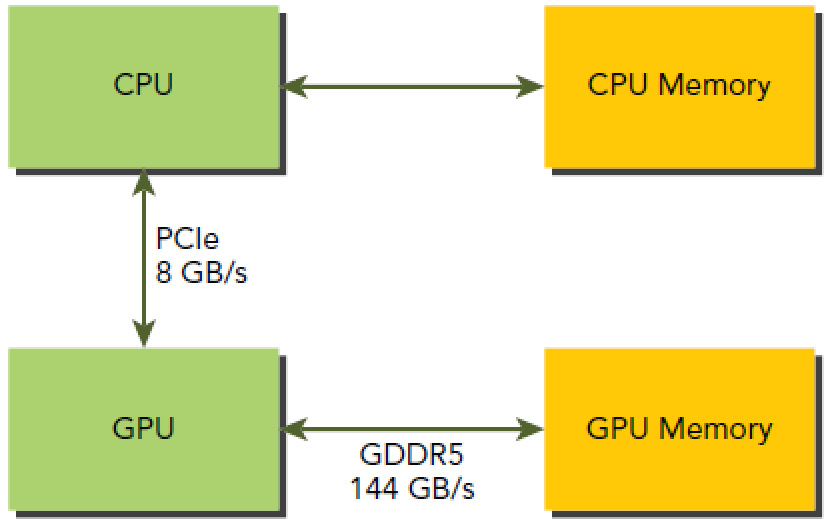


Figure 4-3 highlights the connectivity of CPU and GPU memory

Example: Simple Memory Transfer (memTransfer.cu)

- Purpose: Demonstrates memory transfer from host to device and back.
- Steps:
  1. Set up the device using cudaSetDevice(dev).
  2. Allocate memory on the host and device:
     - Host: `float *h_a = (float *)malloc(nbytes);`
     - Device: `cudaMalloc((float **)&d_a, nbytes);`
  3. Initialize host memory with values.
  4. Transfer data from host to device:

```
cudaMemcpy(d_a, h_a, nbytes, cudaMemcpyHostToDevice);
```

  5. Transfer data back from device to host:

```
cudaMemcpy(h_a, d_a, nbytes, cudaMemcpyDeviceToHost);
```

  1. Free allocated memory.

Profiling Memory Transfer with nvprof

- Profiling Example with nvprof:
  - Profiling result shows data transfer timings for `HostToDevice (DtoH)` and `DeviceToHost (HtoD)`

Example_2_memTransfer

Memory Bandwidth in GPU and CPU

- Figure 4-3: Connectivity and Bandwidth
  - This figure illustrates the bandwidth between the CPU and GPU through the PCIe bus.

- Key Details:
  - GPU GDDR5 memory has a high theoretical bandwidth of 144 GB/s.
  - The PCI Express (PCIe) Gen2 bus provides a much lower bandwidth of 8 GB/s between the CPU and GPU.

- Takeaway: There is a significant disparity in memory bandwidth between the PCIe bus and the GPU memory. Minimizing data transfers between the host and device is essential for maximizing performance.

Figure 4-3 highlights the connectivity of CPU and GPU memory

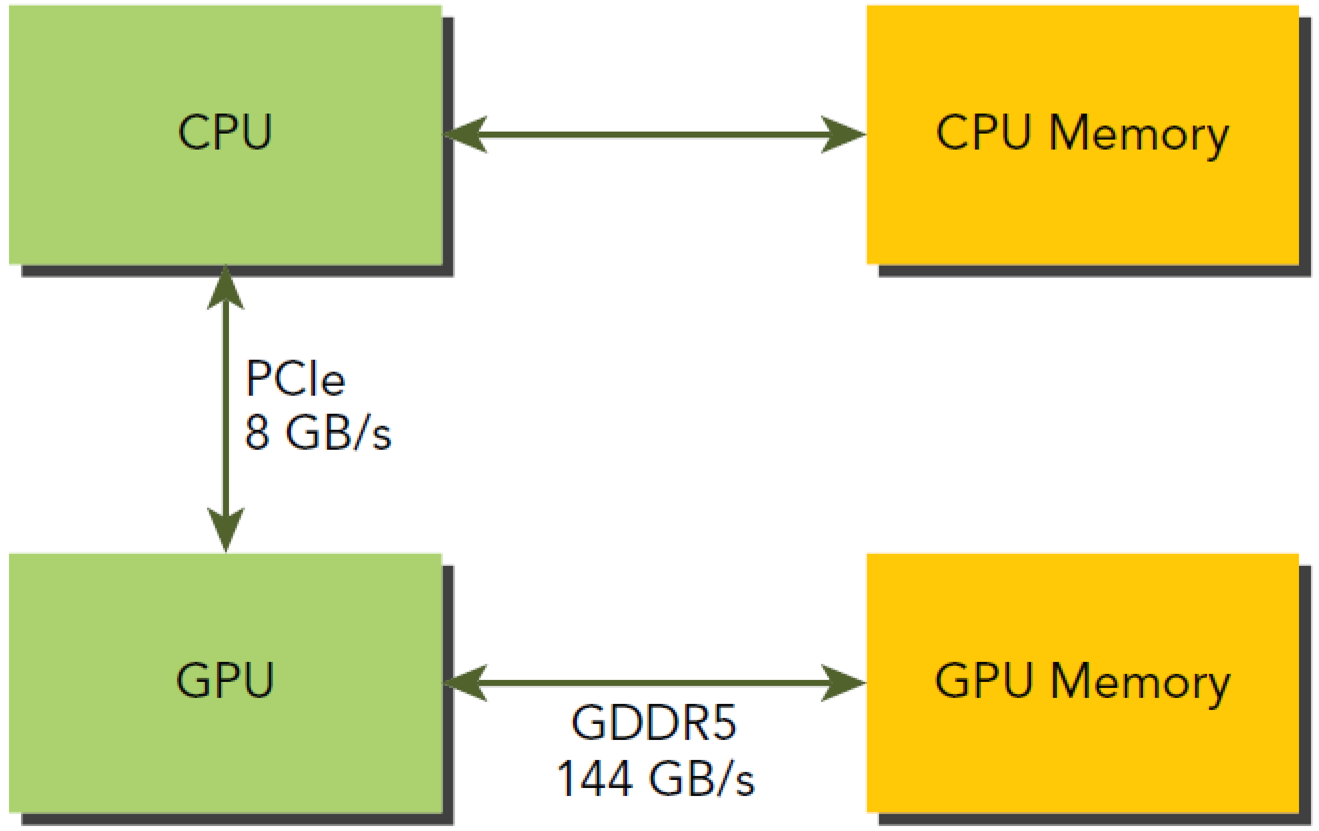


FIGURE 4-3

Pinned Memory

- **Problem with Pageable Memory:** Host memory is pageable, meaning it can be moved by the operating system, which can cause slowdowns due to page faults.

- **Solution:** Use **pinned memory** (also known as **page-locked memory**) to avoid page faults and allow faster memory transfers.

Figure 4-4: Pageable vs Pinned Memory Transfers

- This figure compares **pageable memory** and **pinned memory**:
    - **Pageable Memory Transfer:** Slower due to potential page faults and virtual memory management.
    - **Pinned Memory Transfer:** Faster because the memory is locked in physical memory, allowing direct access for data transfer between the host and GPU.

Pageable vs Pinned Memory Transfers

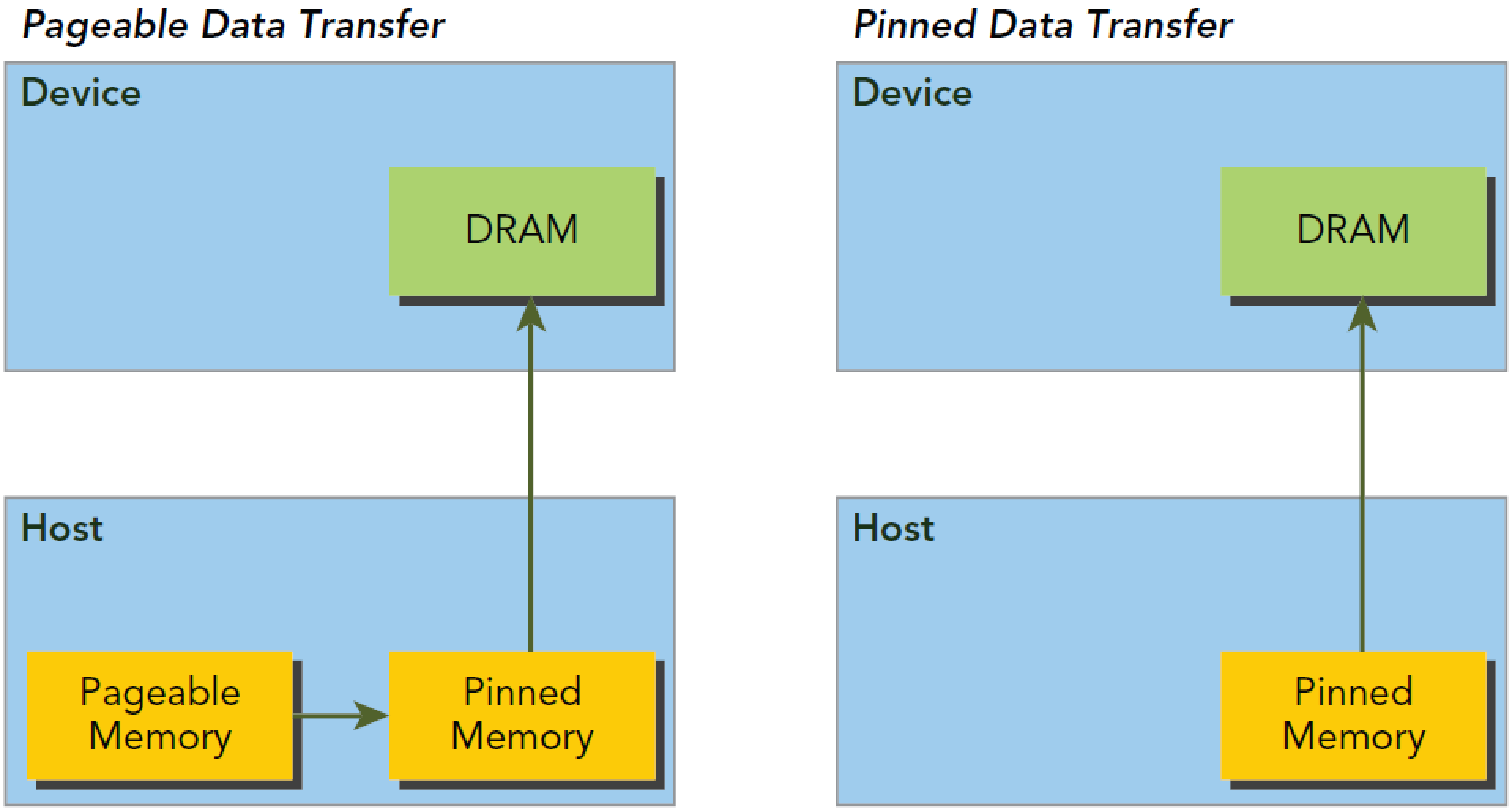


FIGURE 4-4

Using `cudaMallocHost` for Pinned Memory

- **Function**:
```
cudaError_t cudaMallocHost(void **devPtr, size_t count);
```
- **Behavior**: Allocates page-locked memory on the host, allowing for direct and faster transfers to and from the device compared to pageable memory.

Pinned Memory Example

- Allocate pinned memory:
  ```
  cudaMallocHost((void**)&h_aPinned, nbytes);
  ```
- Transfer data between pinned host memory and the device.
- Free pinned memory:
  ```
  cudaFreeHost(h_aPinned);
  ```

Memory Transfer Between Host and Device

- Pinned Memory:
  - Cost: More expensive to allocate and deallocate compared to pageable memory.
  - Benefit: Provides higher throughput for large data transfers.

Performance with Pinned Memory

- Speedup:
  - Performance depends on device compute capability.
  - On Fermi devices, pinned memory is beneficial when transferring more than 10 MB of data.

- Optimization
  - Batch small transfers into larger ones to reduce overhead

Overlapping Transfers with Kernel Execution

- Concurrency:
  - Data transfers can be overlapped with kernel execution to improve performance.
  - Recommendation: Either minimize or overlap data transfers whenever possible.

Zero-Copy Memory Overview

- Two Categories of Heterogeneous Architectures:
  - Integrated: CPUs and GPUs share the same memory.
  - Discrete: Devices are connected to the host via PCIe bus.

Benefits of Zero-Copy Memory

- Integrated Architectures:
  - Zero-copy memory benefits both performance and programmability.
  - No need for data copies over the PCIe bus.

Example_3_sumArrayZerocopy

Usage in Discrete Systems

- Discrete Systems:
  - Zero-copy memory is advantageous only in special cases.
  - Synchronization required to avoid data hazards from multiple threads accessing the same memory location..

Drawbacks of Zero-Copy Memory

- High Latency:
  - Device kernels that read from zero-copy memory can experience high latency.
  - Caution: Avoid overusing zero-copy memory.

Unified Virtual Addressing (UVA) Overview

- Supported from CUDA 4.0:
  - Introduced for devices with Compute Capability 2.0+.
  - Supported on 64-bit Linux systems.
- Key Feature: UVA allows the CPU, host memory, and device memory to share a single virtual address space.

Understanding Figure 4-5:

- Without UVA:
  - Multiple separate memory spaces for CPU and GPUs (GPU0, GPU1).
  - Each memory space must be managed individually (manual pointer management).

With UVA:
  - Single unified memory space for CPU and GPUs.
  - Memory management is simplified, and pointers are transparent.

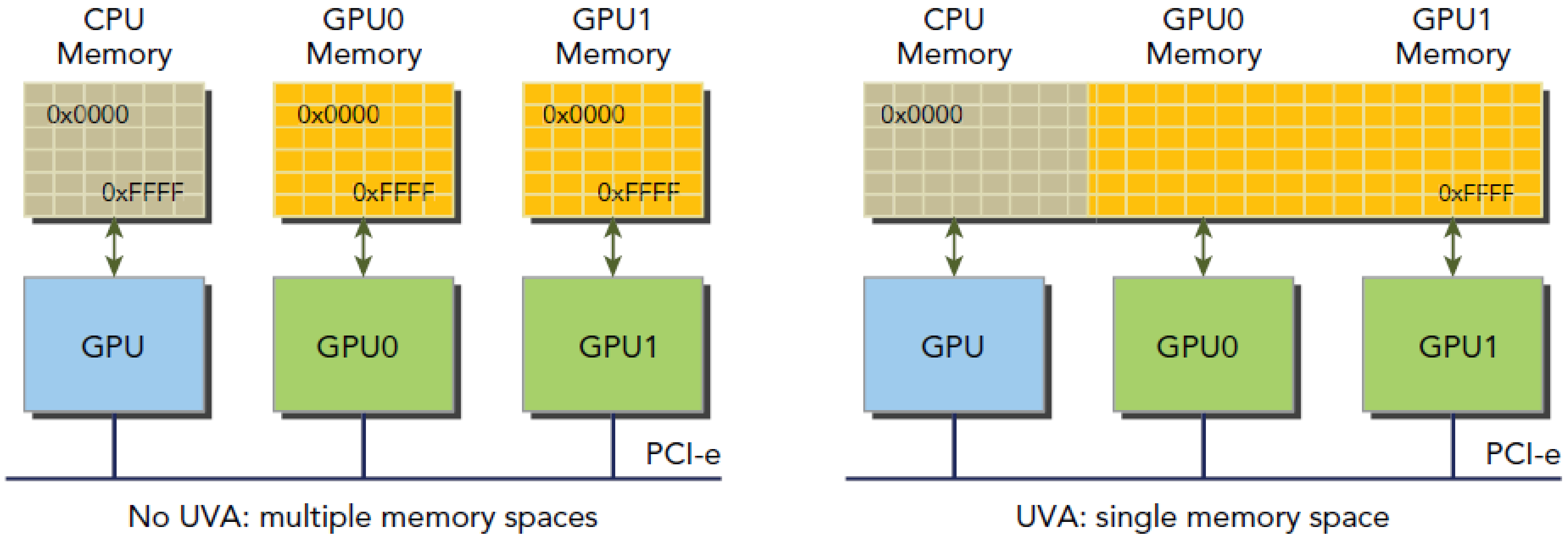


FIGURE 4-5

Benefits of UVA

- Before UVA: Developers had to manage separate pointers for host and device memory.
- With UVA:
  - Pointers are transparent: No need to distinguish between host and device memory pointers.
  - Simplified Code: Reduces complexity in memory management.

Zero-Copy Memory with UVA

- Pinned Host Memory:
  - Allocated using cudaHostAlloc with cudaHostAllocMapped.
  - Host and Device pointers are identical.
  - No need to pass separate device pointers to kernels.

Code Example and Compilation

- Code Example:

```
cudaHostAlloc((void **)&h_A, nBytes, cudaHostAllocMapped);
cudaHostAlloc((void **)&h_B, nBytes, cudaHostAllocMapped);

sumArraysZeroCopy<<<grid, block>>>(h_A, h_B, d_C, nElem);
```

- Compilation:

```
$ nvcc -O3 -arch=sm_61 sumArrayZerocopyUVA.cu -o sumArrayZerocopyUVA
```

Example_4_sumArrayZerocpyUVA

Simplified Code and Performance

- Results:
  - Same performance as pre-UVA implementations.
  - Advantages: Improves code readability and maintainability without performance loss.

Introduction to Unified Memory

- Introduced with CUDA 6.0 to simplify memory management.
- Unified memory creates a pool of managed memory.
- Accessible on both CPU and GPU with a single pointer.
- System automatically migrates data between host and device, transparent to the application.

Unified Virtual Addressing (UVA) vs Unified Memory

- Unified Memory relies on UVA support, but they are different:
  - UVA provides a unified virtual memory address space.
  - UVA does not migrate data, but Unified Memory does.

Advantages of Unified Memory

- Offers a "single-pointer-to-data" model.
- Decouples memory and execution spaces.
- Data is transparently migrated between host and device.
- Improves locality and performance compared to zero-copy memory.

Managed Memory

- Refers to Unified Memory allocations managed by the system.
- Interoperable with device-specific allocations like `cudaMalloc`.
- Allows both managed and un-managed memory types in a kernel.
- Unified Memory operations valid for both device and host.

Managed Memory

- Refers to Unified Memory allocations managed by the system.
- Interoperable with device-specific allocations like `cudaMalloc`.
- Allows both managed and un-managed memory types in a kernel.
- Unified Memory operations valid for both device and host.

Declaring Managed Memory

- Managed memory can be statically declared with the `__managed__` keyword:

```
__device__ __managed__ int y;
```

- Managed memory can be dynamically allocated:

```
  cudaMallocManaged(&devPtr, size, flags);
```

Key Characteristics of Managed Memory

- Automatic data migration between host and device.
- No duplicate pointers required.
- In CUDA 6.0, device code cannot call `cudaMallocManaged`
  - it must be allocated from the host.

Example

- Hands-on example in matrix addition with Unified Memory follows

Memory Access Patterns

- Global Memory Usage:
  - Most device data access begins in global memory.
  - Memory bandwidth is a common limiting factor in GPU applications.
  - Proper tuning of global memory is essential for kernel performance.
  - Poor memory usage tuning will result in negligible performance gains from other optimizations.

Memory Access Patterns

- Optimal Memory Access:
  - Warp Execution: Instructions and memory operations are issued per warp.
  - Threads within a warp provide memory addresses to load/store.
  - The 32 threads in a warp present a single memory access request that is serviced by one or more memory transactions.
  - Memory Patterns:
    - Access patterns are determined by the distribution of addresses across threads in a warp.
    - Different patterns affect the number of memory transactions.
    - Goal: achieve optimal global memory access through efficient memory patterns.

Aligned and Coalesced Access

- Global Memory Caching:
  - Global memory accesses are staged through caches (L1 and L2).
  - Data resides in DRAM, and memory transactions are serviced through 2-byte (L2 cache) or 128-byte (L1 cache) segments.
  - L1 Cache can be enabled/disabled at compile time for better memory caching.
- L1 Cache Details:
  - L1 cache line size: 128 bytes.
  - Threads in a warp request 4-byte values, leading to 128-byte requests per warp.

Aligned and Coalesced Access

- Key Memory Access Characteristics:
  - Aligned Memory Access: Data addresses are aligned with cache granularity (32 or 128 bytes).
  - Coalesced Memory Access: All 32 threads in a warp access contiguous chunks of memory.
  - Goal: Achieve aligned and coalesced memory access for maximum efficiency.

Figure 4-6

- Shows memory caching stages, where SM (Streaming Multiprocessor) uses L1, L2 caches, and DRAM for memory transactions.

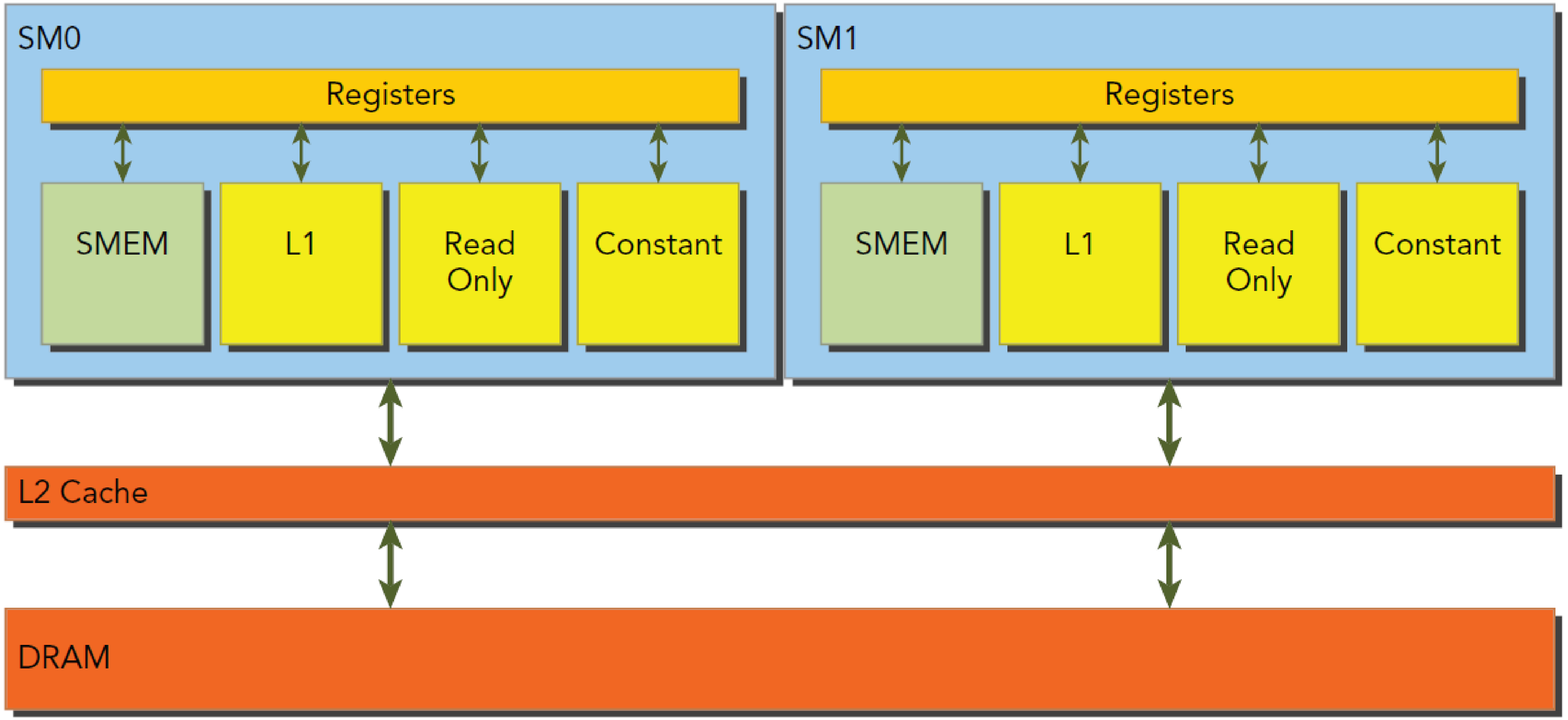


FIGURE 4-6

Aligned Memory Access

- Aligned Access Optimization:
  - Aligned memory addresses prevent wasted bandwidth.
  - Misaligned access causes additional transactions, lowering throughput.

Figure 4-7:

- Illustrates aligned memory access where all 32 threads access contiguous, aligned memory addresses (128 bytes of data is fetched efficiently)

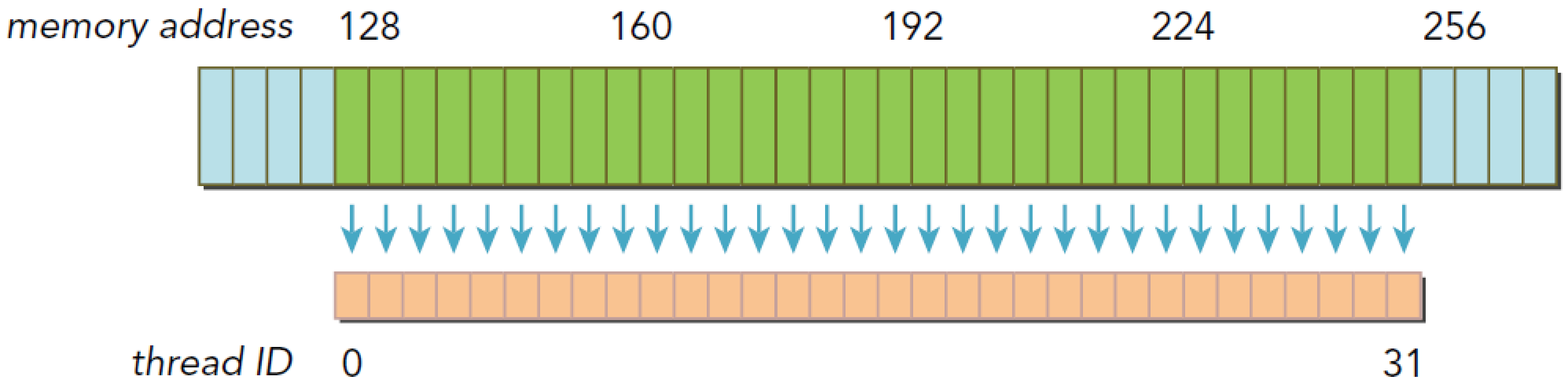


FIGURE 4-7

Misaligned and Uncoalesced Access

- Misaligned Memory Access:
  - Threads access non-aligned memory, requiring multiple memory transactions.
  - Results in wasted bandwidth and inefficient global memory utilization.

Figure 4-8:

- Demonstrates misaligned and uncoalesced access, where multiple transactions are needed to service the memory requests, leading to wasted resources.

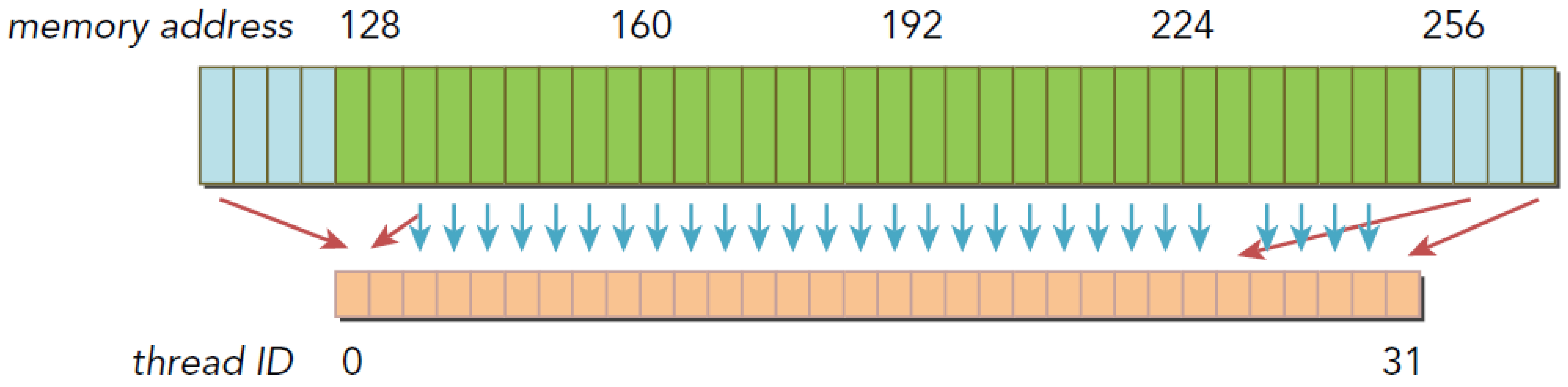


FIGURE 4-8

Global Memory Reads Overview

- **Title**: Global Memory Reads Overview
- **Key Points**:
    - In an SM, data is pipelined through three potential cache/buffer paths:
    - L1/L2 Cache
    - Constant Cache
    - Read-only Cache
- L1/L2 Cache is the default path.
- To pass data through other paths, explicit management is required.
- Whether global memory load operations pass through the L1 cache depends on:
    - **Device Compute Capability**
    - **Compiler Options**

Cache Behavior and Compiler Options

- **Title**: Cache Behavior and Compiler Options
- **Key Points**:
  - For Fermi GPUs (capability 2.x) and Kepler K40 GPUs (capability 3.5+):
  - L1 caching of global memory loads can be enabled/disabled via compiler flags.
  - **Default**:
    - L1 cache is enabled for global memory loads on Fermi.
    - Disabled on Kepler K40 and later.
- **Disabling L1 Cache**:
  - Compiler flag: `-Xptxas -dlcm=cg`
- **Enabling L1 Cache**:
  - Compiler flag: `-Xptxas -dlcm=ca`

L1 and L2 Cache Behavior

- **Title**: L1 and L2 Cache Behavior
- **Key Points**:
  - With L1 cache disabled:
    - All requests go directly to L2.
    - If L2 misses, requests are serviced by DRAM.
  - With L1 cache enabled:
    - Load requests first attempt L1.
    - On miss, requests go to L2, then DRAM.

L1 Cache Use on Kepler GPUs

- **Title**: L1 Cache Use on Kepler GPUs
- **Key Points**:
  - On Kepler K10, K20, K20x GPUs:
    - L1 is **not** used to cache global memory loads.
    - L1 is exclusively used for caching register spills to local memory.

Memory Load Access Patterns

- **Title**: Memory Load Access Patterns
- **Key Points**:
    - Two types of memory loads:
        - Cached (L1 cache enabled)
        - Uncached (L1 cache disabled)
    - Access patterns characterized by:
        - **Cached vs Uncached**
        - **Aligned vs Misaligned**: Load is aligned if the first memory address is a multiple of 32 bytes.
        - **Coalesced vs Uncoalesced**: Coalesced if warp accesses a contiguous chunk of data.

Performance Impact

- **Title**: Performance Impact of Memory Access Patterns
- **Key Points**:
  - Impact of memory access patterns on kernel performance will be examined in subsequent sections.

Cached Loads

- **Key Points**:
  - Cached load operations pass through L1 cache.
  - Memory transactions serviced at 128-byte granularity.
  - Cached loads can be:
    - Aligned/Misaligned
    - Coalesced/Uncoalesced
  - **FIGURE 4-9**: Illustrates an ideal case where memory accesses are aligned and coalesced. All threads in the warp fall within one cache line, maximizing bus utilization.

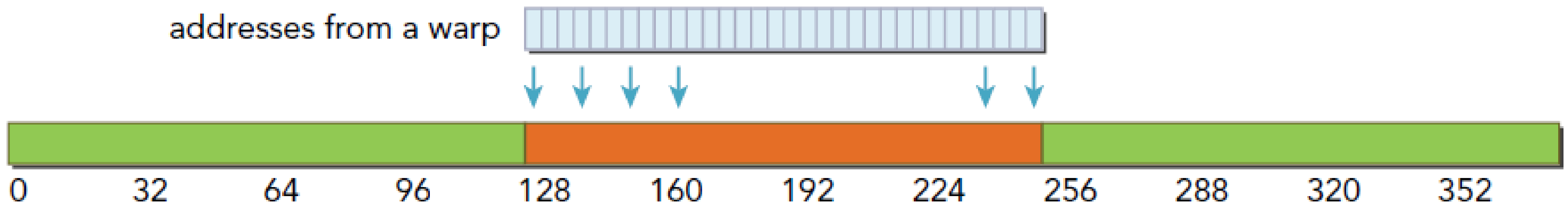


FIGURE 4-9

Aligned Access with Randomization

- **Title**: Aligned Access with Randomized Thread IDs
- **Key Points**:
  - Addresses are randomized but still fall within one cache line.
  - Only one 128-byte transaction required.
  - Bus utilization remains 100%.
  - **FIGURE 4-10**: Visualizes this aligned yet randomized case.

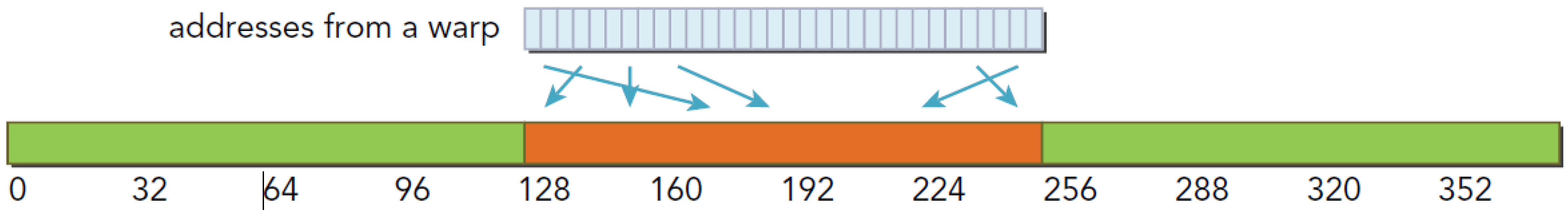


FIGURE 4-10

Misaligned Access

- **Title**: Misaligned Access (50% Utilization)
- **Key Points**:
    - Threads request 32 consecutive data elements that are misaligned.
    - Two 128-byte transactions required due to misalignment.
    - Bus utilization drops to 50%.
    - **FIGURE 4-11**: Shows how misalignment impacts memory load operations.

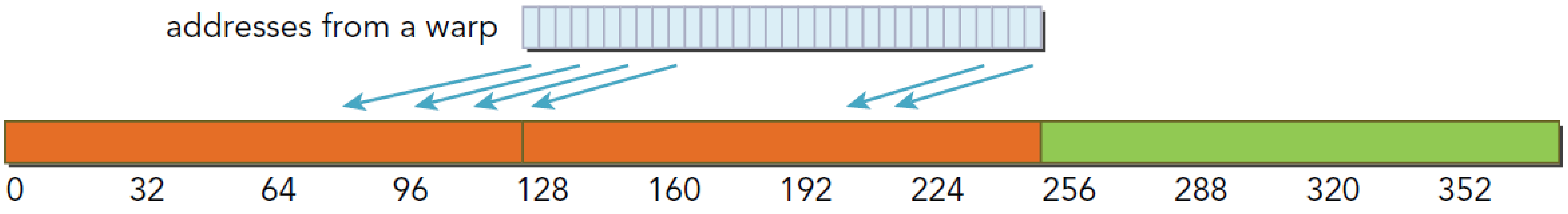


FIGURE 4-11

All Threads Request Same Address

- **Title**: Low Utilization with Same Address Request
- **Key Points**:
  - Threads request the same address.
  - Bus utilization is very low—only 3.125%.
  - **FIGURE 4-12**: Demonstrates how bus utilization plummets when all threads request the same address.

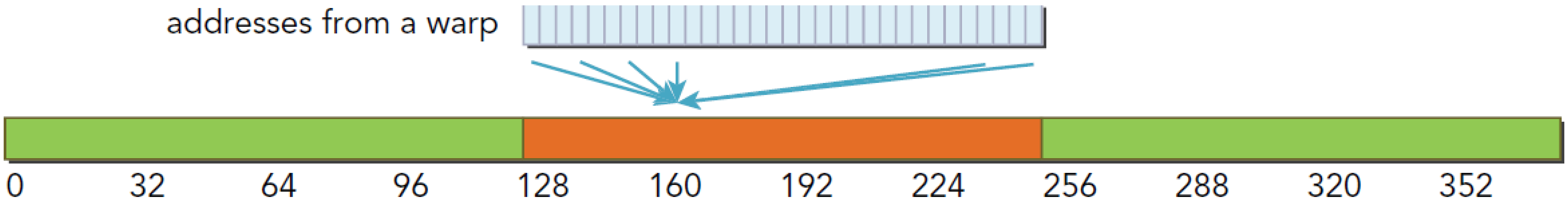


FIGURE 4-12

Worst-Case Scenario (32 Threads, 32 Cache Lines)

- **Key Points**:
- Threads request 32 four-byte addresses scattered across multiple cache lines.
  - Multiple memory transactions required, reducing efficiency.
  - **FIGURE 4-13**: Depicts this worst-case scenario, with addresses scattered across multiple cache lines.

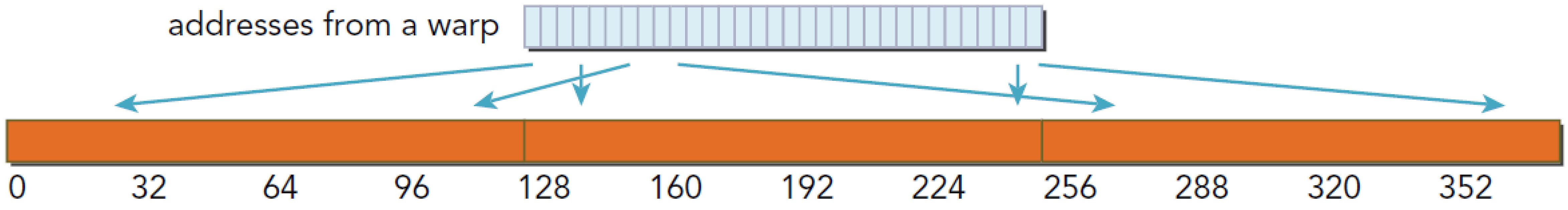


FIGURE 4-13

Uncached Loads

- **Uncached loads** do not pass through the L1 cache and operate on finer granularity: memory segments of 32 bytes, not cache lines of 128 bytes. This can lead to better bus utilization for misaligned or uncoalesced memory accesses.

- **FIGURE 4-14**
  - **Aligned and coalesced memory access**:
    - In this ideal case, addresses fall into four segments (32 bytes each).
    - **100% bus utilization** is achieved because the memory addresses are aligned and accessed in a coalesced manner.

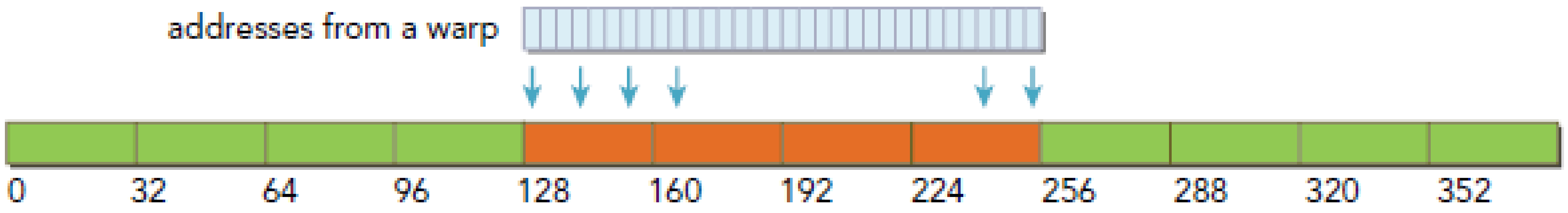


FIGURE 4-14

FIGURE 4-15

- **Randomized but aligned accesses**:
    - accesses are randomized within a 128-byte range.
    - As long as every thread accesses unique addresses, the data will still be fetched efficiently in four memory segments.
    - Performance remains high with good kernel execution.

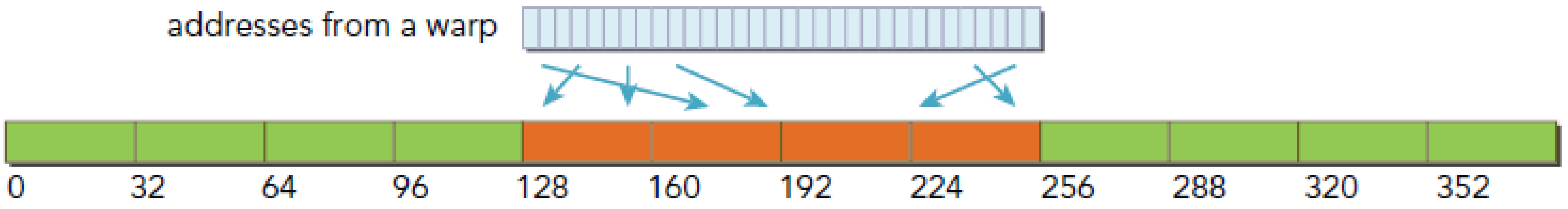


FIGURE 4-15

Misaligned and Scattered Loads

- **Impact of non-sequential access patterns** on memory performance.
- **FIGURE 4-16**
  - **Consecutive 4-byte elements with a misaligned load**:
    - A warp requests 32 consecutive 4-byte elements, but the memory load is not aligned to a 128-byte boundary.
    - The 128-byte transaction occurs over five memory segments, achieving at least **80% bus utilization**.

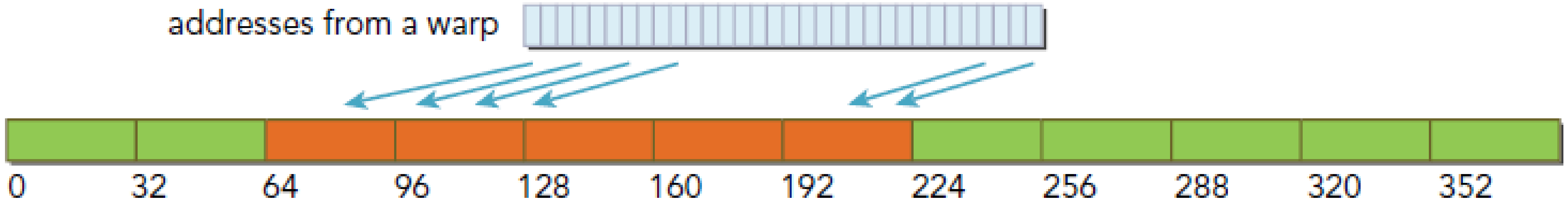


FIGURE 4-16

Difference Between CPU L1 Cache and GPU L1 Cache

- **Title**: CPU vs. GPU L1 Cache
- **Key Points**:
  - CPU L1 Cache: Optimized for both spatial and temporal locality.
  - GPU L1 Cache: Optimized only for spatial locality.
  - Frequent access to a cached memory location does **not** guarantee the data will remain in the GPU L1 cache.

FIGURE 4-18

- **Worst-case scenario**:
  - The warp requests 32 4-byte words scattered across global memory.
  - In this case, the data is distributed across several 128-byte cache lines, resulting in low efficiency. However, this worst case is still improved compared to cached loads.

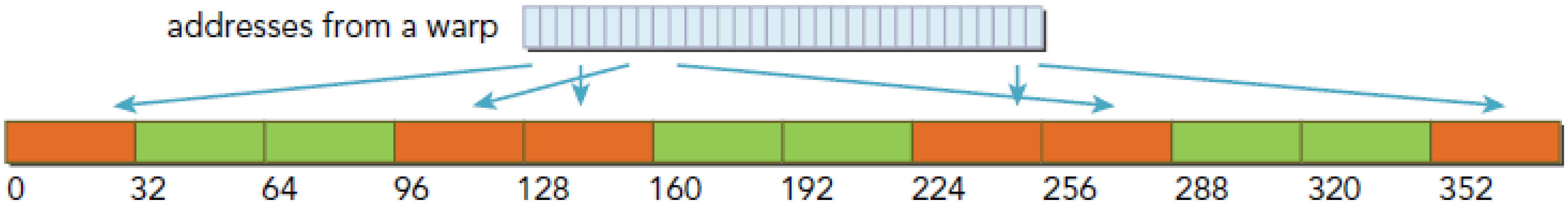


FIGURE 4-18

Example of Misaligned Reads and Their Impact on Performance

- **Misaligned Memory Access**: Occurs when memory loads from arrays are shifted, causing inefficiency in accessing memory.
- **Alignment Strategies**: Proper alignment of memory accesses helps to improve performance, particularly in parallel computing like CUDA.

Illustration of Misalignment Impact

- **Kernel Modification**: A CUDA kernel is adjusted to demonstrate the effect of misaligned loads, specifically by introducing an offset to misalign accesses for arrays `A` and `B`.
    - *Aligned Case (offset = 0):* Memory accesses are coalesced and efficient.
    - *Misaligned Case (offset = 11):* Memory accesses are not aligned, leading to inefficiency.

Performance Impact of Misaligned Access

- **Metric: Global Load Efficiency (gld_efficiency):**
  - Formula

$$\text{gld\_efficiency} = \frac{\text{Requested Global Memory Load Throughput}}{\text{Required Global Memory Load Throughput}}$$

  - Aligned Case (offset = 0 and 128): Efficiency is 100%.
  - Misaligned Case (offset = 11): Efficiency drops to ~49.81%, indicating that memory transactions double.

NVPROF Profiling Results

- Global Load Transactions (gld_transactions): Number of memory transactions.
    - Offset = 0: 65,184 transactions.
    - Offset = 11: 131,039 transactions (doubled).
    - Offset = 128: 65,744 transactions.

Effect of Disabling L1 Cache

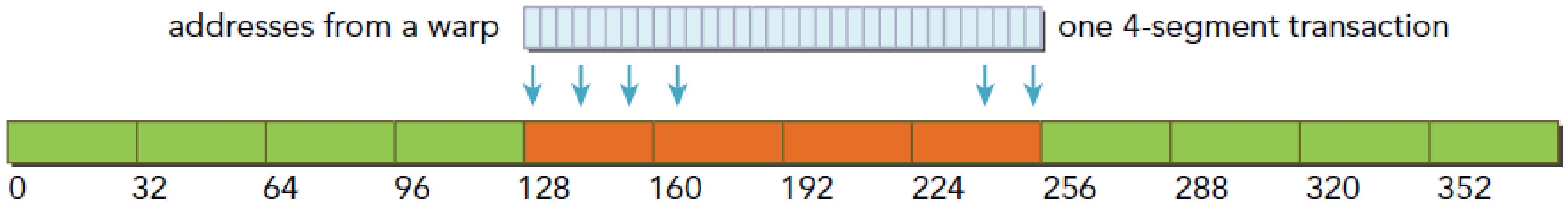
- Command: -Xptxas -dlcm=cg disables the L1 cache for global memory loads.
- Impact:
  - Performance slightly slower without L1 cache.
  - gld_efficiency:
    - Offset = 0 and 128: Remains at 100%.
    - Offset = 11: Improves to 80% (better than 49.81%).

Example_5_readSegment

Global Memory Writes

FIGURE 4-19

- Ideal case: All threads in the warp access a consecutive 128-byte range.
    - This is handled by one 4-segment transaction, ensuring 100% bus utilization



memory address

FIGURE 4-19

Global Memory Writes

FIGURE 4-20

- Aligned but scattered access: Threads in the warp access a 192-byte range, which spans across multiple segments.
  - This results in three 1-segment transactions, lowering the efficiency.

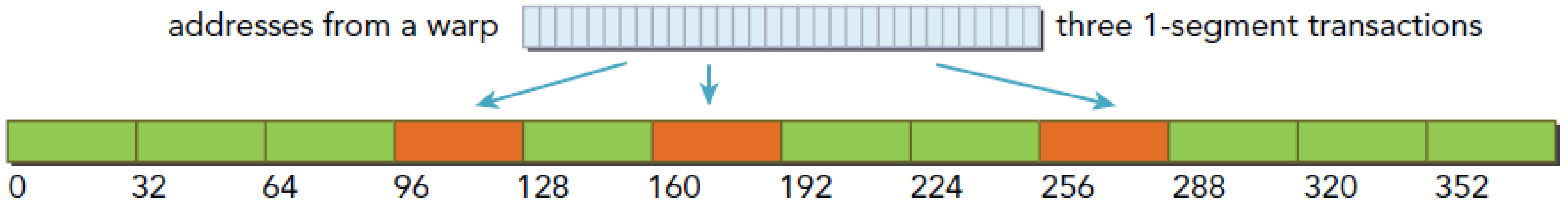


FIGURE 4-20

Global Memory Writes

FIGURE 4-21

- Two-segment transaction: When memory accesses are aligned but span a 64-byte range, the request is serviced by one two-segment transaction.
  - This results in higher efficiency compared to misaligned or scattered writes.

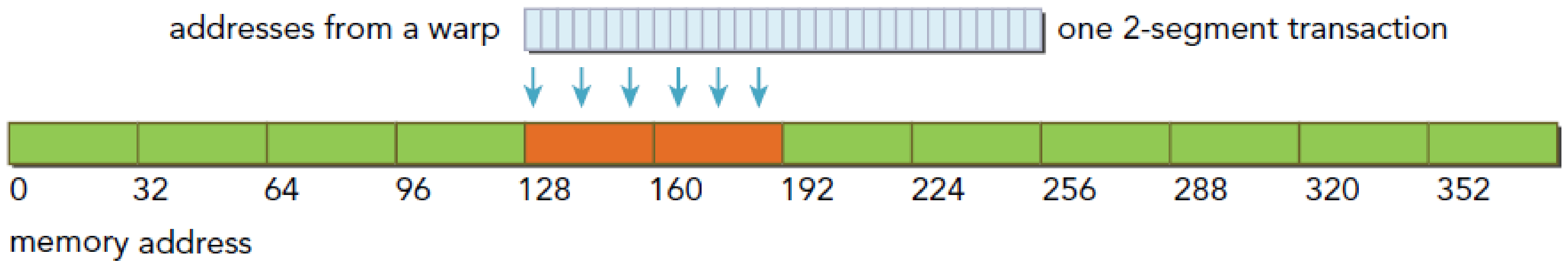


FIGURE 4-21

Misaligned Writes in CUDA

- Misaligned writes affect memory store efficiency in CUDA.
- This example modifies a vector addition kernel to demonstrate this effect.

Kernel Modification for Misalignment

- Modified kernel function `writeOffset` uses two indices:
  - i: Aligned index for reading arrays A and B.
  - k = i + offset: Misaligned index for writing to array C.
- Introduces potential misalignment depending on offset value.

Host Code Adjustment

- Host function `sumArraysOnHost` revised to incorporate offset:
  - Loops over array indices using i + offset for C array.
- Matches kernel changes for consistent results.

Compilation and Execution

- Compile command: `nvcc -O3 -arch=sm_61 writeSegment.cu -o writeSegment.`
- Execute with various offsets (0, 11, 128) to observe performance differences.

Results of Misaligned Writes

- Sample outputs for different offsets:
  - Offset 0: Aligned, fast execution (100% efficiency).
  - Offset 11: Misaligned, slower due to reduced efficiency (80%).
  - Offset 128: Aligned, high efficiency similar to offset 0.

Efficiency Metrics with nvprof

- Efficiency analysis using nvprof for global load/store metrics.
- Commands:

```
nvprof --metrics gld_efficiency --metrics gst_efficiency ./writeSegment $OFFSET
```

- Findings:
  - Offsets 0 and 128 have 100% load/store efficiency.
  - Offset 11 shows only 80% store efficiency due to misalignment.

Example_6_writeSegment

Cause of Efficiency Drop at Offset 11

- Misaligned writes (offset 11) result in fragmented memory transactions:
    - 128-byte write request splits into one 4-segment and one 1-segment transaction.
    - Results in 160 bytes accessed instead of 128, causing only 80% efficiency.

Conclusion

- Misalignment significantly impacts CUDA performance.
- Aligning data access is crucial for optimized memory operations.

Array of Structures (AoS) vs. Structure of Arrays (SoA)

- Data Layout Options
  - Array of Structures (AoS):
    - Stores related fields (e.g., `x` and `y`) together in one structure.
    - Suitable for CPU cache locality but less efficient for GPUs.
  - Structure of Arrays (SoA):
    - Separates each field into independent arrays, storing all `x` values in one array and all `y` values in another.
    - Optimized for GPU memory access due to coalesced memory access patterns.

Figure 4-22 illustrates the memory layout of both AoS and SoA approaches

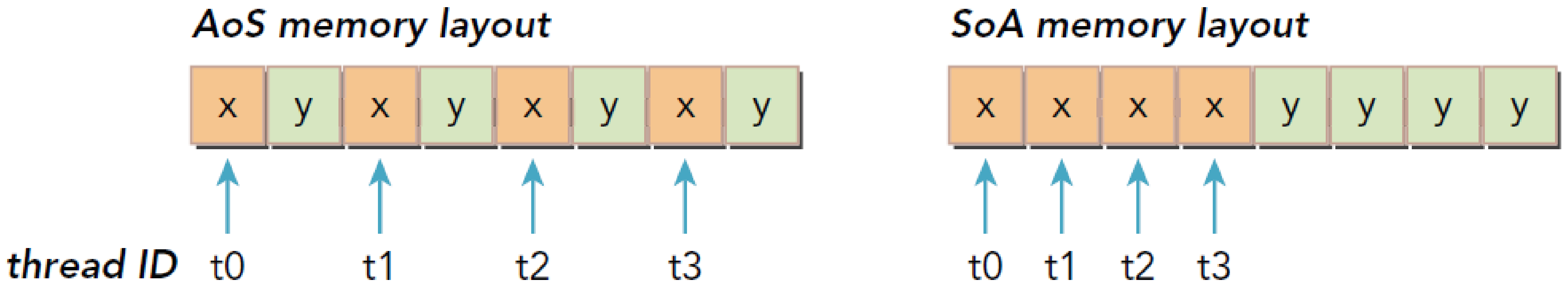


FIGURE 4-22

Example Definitions

- Defining AoS and SoA Structures
  - AoS Layout Definition:
    - Define `innerStruct` with `float x` and `float y`.
    - Array of structs: `innerStruct myAoS[N];`
  - SoA Layout Definition:
    - Separate arrays for `x` and `y`: `float x[N]; float y[N];`
    - Variable using this structure: `struct innerArray moa;`

Memory Layout Differences

- Visualizing AoS vs. SoA Memory Layouts
  - AoS Layout:
    - Threads load both x and y even if only x is needed.
    - Leads to inefficient bandwidth usage on GPU.
  - SoA Layout:
    - Fields stored independently allow coalesced memory access.
    - Maximizes memory bandwidth efficiency on GPU.

Implications for Parallel Programming

- Preference in SIMD/CUDA Programming
  - Coalesced Memory Access with SoA:
    - SoA layout enables more efficient memory access.
  - AoS Impact:
    - AoS layout is beneficial for CPU cache but less effective in GPU environments due to scattered memory access.

Implementation of AoS Layout in CUDA

- Kernel Implementation
  - Kernel Function (`testInnerStruct`):
    - Processes data using AoS layout.
    - Reads `x` and `y` fields, performs operations, stores results.

CUDA Memory Allocation and Setup

- CUDA Memory and Initialization
  - Global Memory Allocation:
    - Allocate memory for AoS structures in CUDA.
  - Host Array Initialization:
    - Initialize arrays on the host with `initialInnerStruct`.

Execution Configuration and Warm-up

- Execution Setup
  - Configure Execution:
    - Set grid and block sizes for kernel execution.
  - Warm-up Kernel:
    - Run a warm-up kernel to mitigate CUDA startup overhead.

Running the Kernel and Measuring Performance

- Kernel Execution and Performance
  - Kernel Execution:
  - Run `testInnerStruct` kernel and measure elapsed time.
  - Sample Output:
    - Example result: `elapsed 0.000286 sec` on Fermi M2070.

Example_7_simpleMathAoS

Example_8_simpleMathSoA

Conclusion

- Comparing AoS and SoA
  - AoS Summary:
    - Beneficial for CPU but less efficient for GPUs.
  - SoA Summary:
    - Maximizes GPU efficiency due to coalesced access patterns, making it preferable in CUDA applications.

Performance Tuning Goals

- **Objective**: Optimize device memory bandwidth utilization.
- **Key Strategies**:
  - **Aligned and Coalesced Access**: Reduces bandwidth waste by ensuring efficient data loading.
  - **Concurrent Memory Operations**: Maximizes memory throughput by overlapping operations to hide latency.
- **Focus Areas**:
  - Align and coalesce memory access.
  - Increase independent memory operations per thread for higher parallelism.

Unrolling Techniques

- **Purpose of Loop Unrolling**: Adds independent memory operations, increasing concurrent access.
- **Example**: Modified `readOffsetUnroll4` kernel with 4 independent memory loads per thread.
- **Result**: More concurrent accesses possible, reducing idle time and improving bandwidth usage.

Impact of Unrolling on Performance

- **Performance Gains**: Unrolling can provide up to 3x speedup over non-unrolled kernels.
- **Efficiency Metrics (Using `nvprof`)**:
  - **Aligned case**: High memory efficiency.
  - **Misaligned case**: Unrolled kernel shows improved load efficiency even with misalignment.

Example_9_readSegmentUnroll

Exposing More Parallelism

- **Optimal Block Size**: Experimented with 128, 256, and 512 threads per block.
- **Results**: 256 threads per block performed best in unrolled kernel, leveraging more blocks per SM.
- **Hardware Limitations**: On Fermi GPUs, max concurrency is limited by warp limits and SM resources.

Conclusion - Maximizing Bandwidth Utilization

- **Key Factors**:
  - **Memory Alignment & Coalescing**: Essential for reducing bandwidth waste.
  - **Increasing Independent Operations**: Loop unrolling and optimized execution configuration expose more parallelism.
- **Takeaway**: Align and unroll memory access patterns for optimal GPU performance.

Bandwidth in Kernel Performance

- **Kernel Performance Metrics**:
  - **Memory Latency**: Time to satisfy an individual memory request.
  - **Memory Bandwidth**: Rate of device memory access per SM, in bytes per time unit.
- **Methods to Improve Kernel Performance**:
  - **Hiding Memory Latency**: Maximize executing warps to increase bus saturation and in-flight memory accesses.
  - **Maximizing Memory Bandwidth Efficiency**: Proper alignment and coalescing of memory accesses.
- **Challenges in Kernel Design**:
  - Addressing bad access patterns for suboptimal performance.
  - Importance of tuning for achievable performance in real-world conditions.

Understanding Memory Bandwidth

- **Types of Memory Bandwidth**:
  - **Theoretical Bandwidth**: Max possible bandwidth with current hardware.
  - **Effective Bandwidth**: Actual measured bandwidth achieved by a kernel.

- **Effective Bandwidth Formula**:

$$\text{Effective Bandwidth (GB/s)} = \frac{(\text{bytes read} + \text{bytes written}) \times 10^{-9}}{\text{time elapsed}}$$

- **Example Calculation** (for a 2048x2048 matrix of 4-byte integers):

$$\text{Effective Bandwidth (GB/s)} = \frac{2048 \times 2048 \times 4 \times 2 \times 10^{-9}}{\text{time elapsed}}$$

- **Objective**: Measure and tune the effective bandwidth for a matrix transpose kernel.

Matrix Transpose Problem

- **What is Matrix Transpose**?
  - Common operation in linear algebra.
  - Involves swapping rows and columns of a matrix.

- **Example**:
  - Matrix (left) and Transposed Matrix (right) as shown in **Figure 4-23**.
- **Applications**:
  - Widely used in scientific and engineering computations.

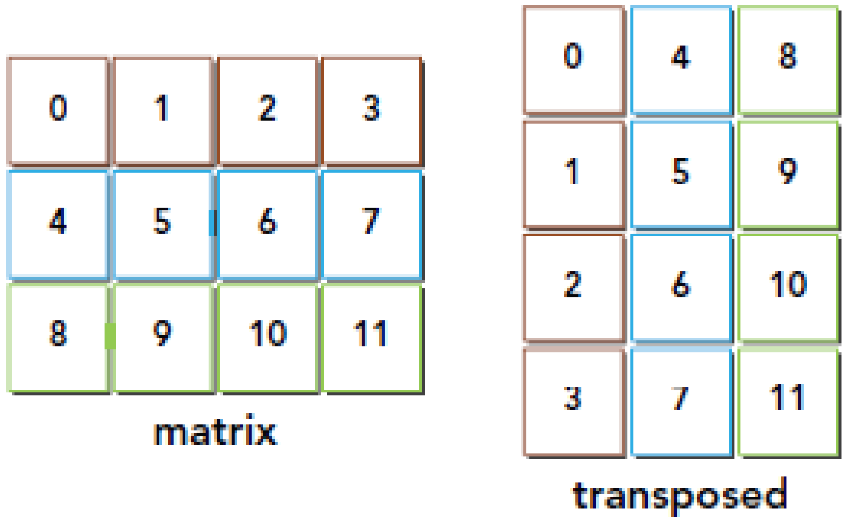


FIGURE 4-23

Host-Based Transpose Implementation

- **Code Example**:
    - Host function `transposeHost()` performs out-of-place transpose using single-precision floats.
    - Takes matrix stored in 1D array format.

- **Matrix Layout**:
    - The input matrix `in` and output matrix `out` are stored as 1D arrays.
    - Illustrated in **Figure 4-24**:
        - **Data layout of original matrix.**
        - **Data layout of transposed matrix.**

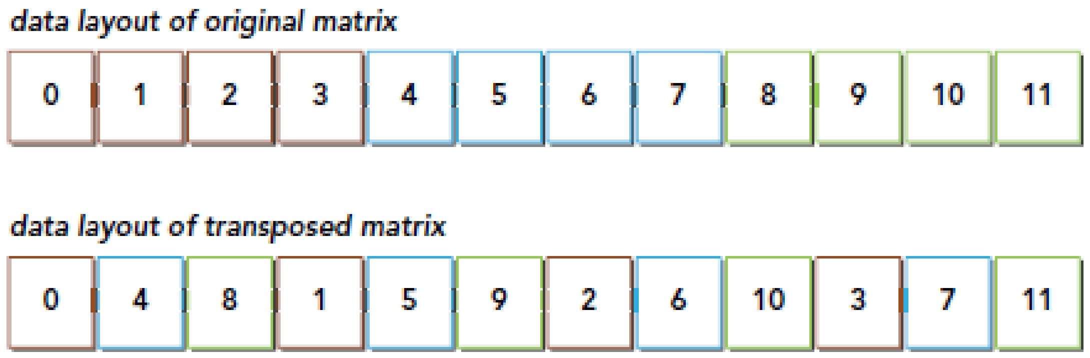

data layout of original matrix
0 1 2 3 4 5 6 7 8 9 10 11
data layout of transposed matrix
0 4 8 1 5 9 2 6 10 3 7 11

FIGURE 4-24

Observations on Access Patterns

- **Access Patterns**:
  - Reads: Accessed by rows in original matrix for coalesced access.
  - Writes: Accessed by columns in transposed matrix, resulting in strided access.

- **Strided Access**:
  - Reduces performance on GPUs due to uncoalesced memory access.
  - Aim to optimize read/write patterns in kernel design.

Improving Bandwidth Utilization

- **Two Approaches for Transpose Kernel**:
  - **Approach 1**: Reads by rows and stores by columns (Figure 4-25 left).
  - **Approach 2**: Reads by columns and stores by rows (Figure 4-26 right).

- **Performance Implications**:
  - With L1 cache enabled for loads:
    - Approach 2 can improve performance by caching reads.
  - Without L1 cache for global memory:
    - Performance difference is minimal on Kepler K10, K20, K20x devices.

Approach 1: Reads by rows and stores by columns (Figure 4-25 left)

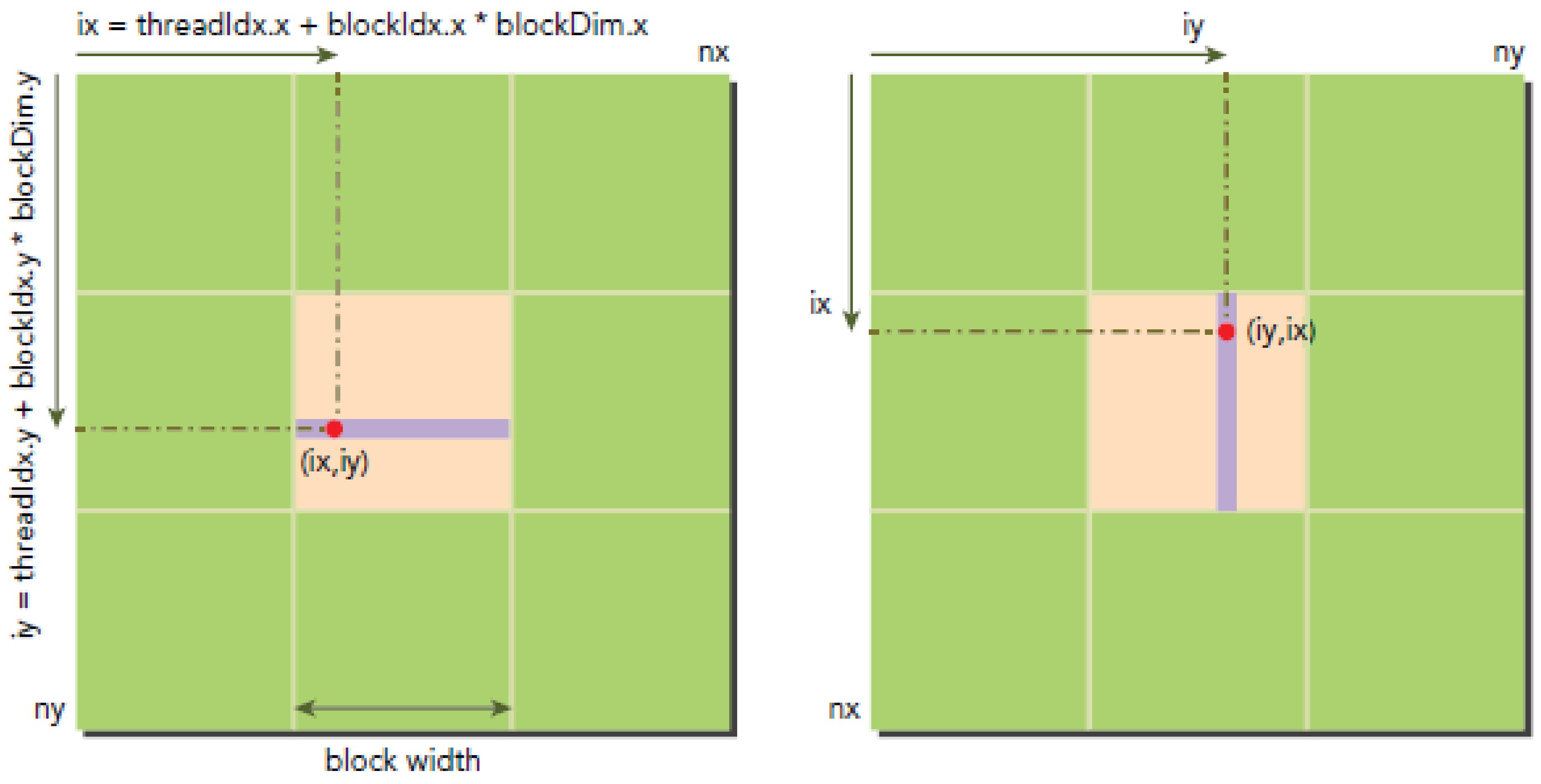
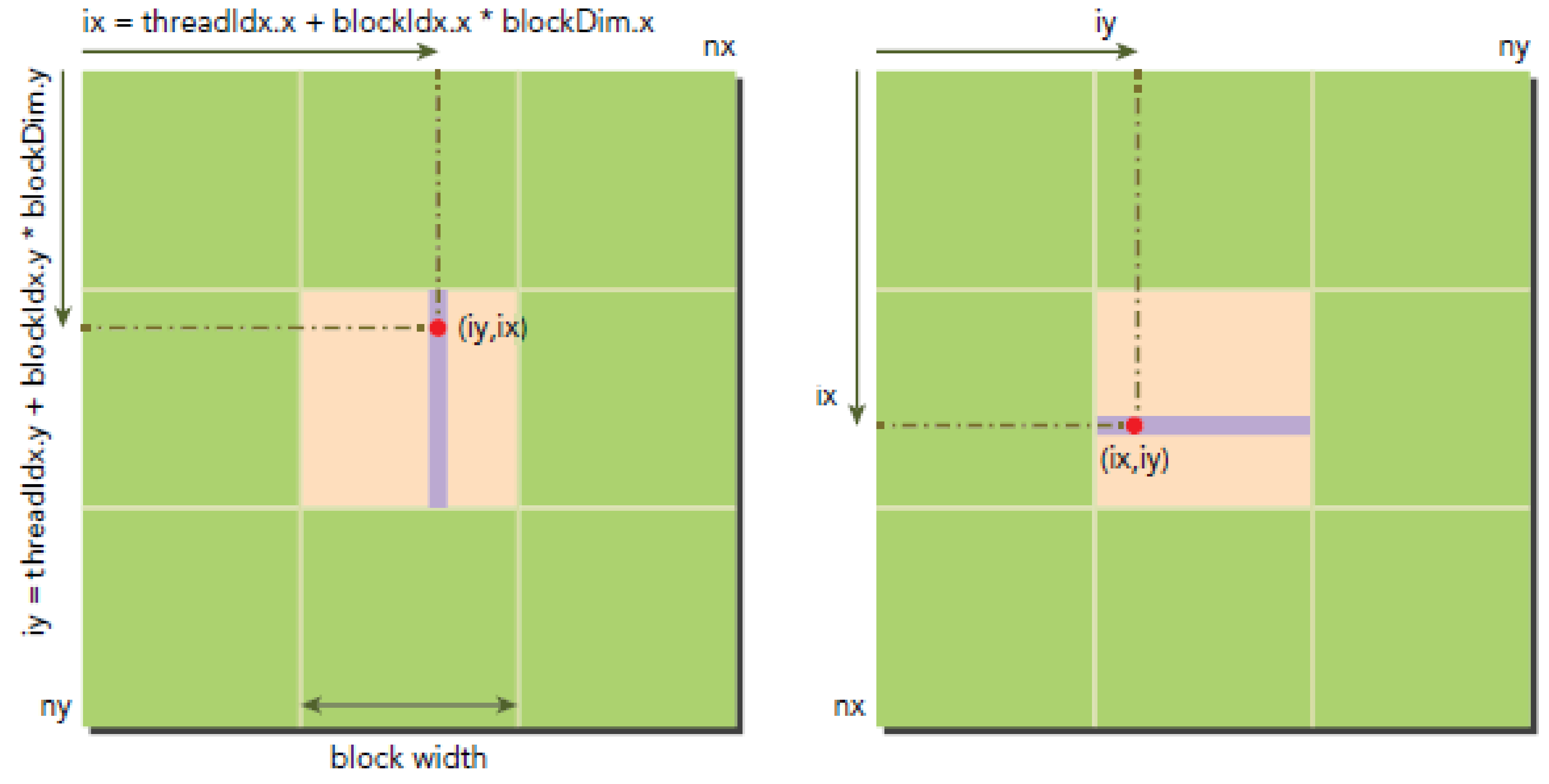

ix = threadIdx.x + blockIdx.x * blockDim.x
nx
iy = threadIdx.y + blockIdx.y * blockDim.y
(ix,iy)
ny
block width
iy
ny
ix
(iy,ix)
nx

FIGURE 4-25

Approach 2: Reads by columns and stores by rows (Figure 4-26 right)



ix = threadIdx.x + blockIdx.x * blockDim.x
nx
iy = threadIdx.y + blockIdx.y * blockDim.y
ny
(iy,ix)
block width
iy
ny
ix
(ix,iy)
nx

FIGURE 4-26

CUDA Matrix Transpose Overview

- **Objective**: Transpose a matrix efficiently on GPU using CUDA.
- **Code Setup**:
  - Defines kernel execution with `switch` statement for kernel selection.
  - Uses CUDA API for device setup, memory allocation, and data transfer.
  - Measures performance of kernels.

Kernel Variants for Transpose Operation

- **Kernels Implemented**:
  - `CopyRow`: Loads/stores by rows.
  - `CopyCol`: Loads/stores by columns.
  - `NaiveRow`: Reads rows, stores columns.
  - `NaiveCol`: Reads columns, stores rows.
- **Execution Commands**:
  - `./transpose 0` for CopyRow
  - `./transpose 1` for CopyCol
  - `./transpose 2` for NaiveRow
  - `./transpose 3` for NaiveCol

Performance Analysis - L1 Cache Enabled (Table 4-4 & 4-5)

- **Bandwidth Comparison**:
  - **`CopyRow`**: 125.67 GB/s (70.76% of peak)
  - **`CopyCol`**: 58.76 GB/s (33.09% of peak)
  - **`NaiveRow`**: 64.16 GB/s (36.13% of peak)
  - **`NaiveCol`**: 81.64 GB/s (45.97% of peak)
  - **Observation:** `NaiveCol` outperforms `NaiveRow` due to better cache utilization.

Performance Analysis - L1 Cache Disabled (Table 4-6)

- Bandwidth with L1 Disabled:
  - **`CopyRow`**: 128.07 GB/s (upper bound)
  - **`CopyCol`**: 40.42 GB/s (lower bound)
  - **`NaiveRow`**: 63.79 GB/s
  - **`NaiveCol`**: 47.13 GB/s
- **Conclusion**: Disabling L1 cache reduces performance significantly, especially for strided access patterns.

Load/Store Efficiency (Table 4-7 & 4-8)

- **Throughput and Efficiency**:
  - **`CopyRow`** achieves high load/store efficiency with row-based loading.
  - **`NaiveCol`** suffers from inefficiency in strided reads but benefits from L1 cache.
  - **Key Insight**: Efficient cache utilization is crucial for improving load/store efficiency in GPU memory-bound operations.

Diagonal Transpose in CUDA: Reading Rows vs. Columns

- **Concept Overview**:
  - In CUDA programming, grids of thread blocks are distributed among Streaming Multiprocessors (SMs).
  - Each thread block has a unique identifier, `bid`, based on a row-major order.
- **Figure 4-27**: Shows Cartesian coordinates and corresponding block IDs for a 4x4 grid.

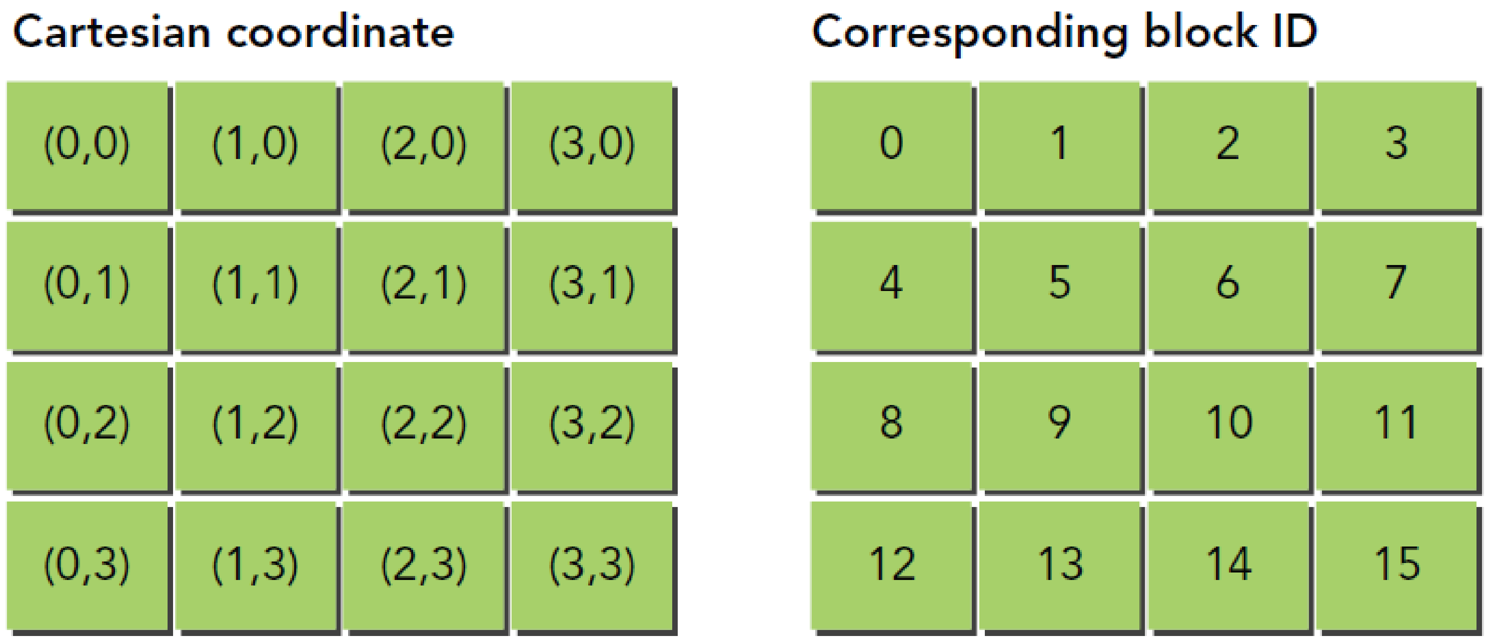
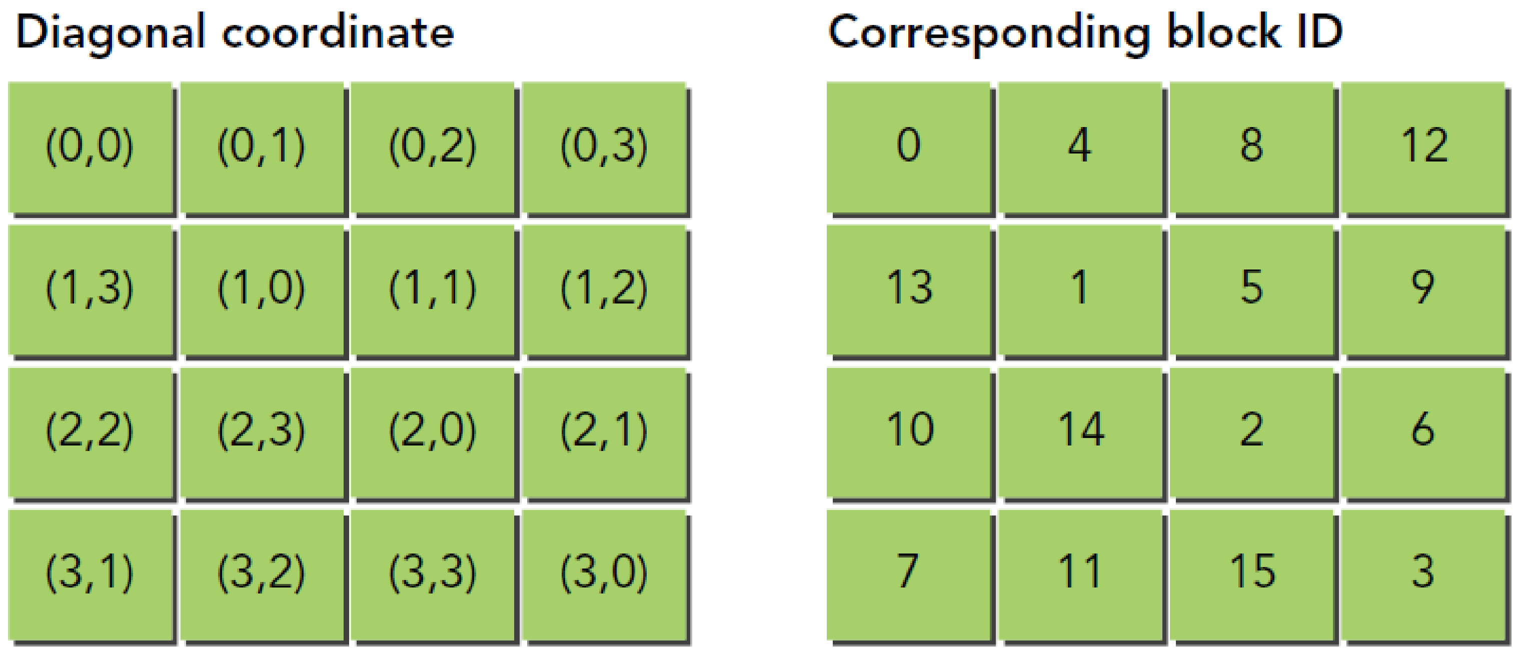


FIGURE 4-27

Cartesian vs. Diagonal Coordinates

- **Difference in Coordinate Systems**:
  - Cartesian coordinates are traditional; diagonal coordinates offer an alternative mapping.
- **Figure 4-28**: Demonstrates diagonal coordinates with a re-mapped block ID arrangement.



Diagonal coordinate

(0,0)	(0,1)	(0,2)	(0,3)
(1,3)	(1,0)	(1,1)	(1,2)
(2,2)	(2,3)	(2,0)	(2,1)
(3,1)	(3,2)	(3,3)	(3,0)

Corresponding block ID

0	4	8	12
13	1	5	9
10	14	2	6
7	11	15	3

FIGURE 4-28

Implementing Diagonal Coordinates in CUDA

- **Mapping Explanation**:
  - Block IDs are mapped from diagonal to Cartesian coordinates for accurate data access.
  - **Code Sample**: Shows diagonal coordinate calculation and the transpose kernel for row-based access.
- **Mapping Formula**:
  - `block_x = (blockIdx.x + blockIdx.y) % gridDim.x;`
  - `block_y = blockIdx.x;`

Column-Based Diagonal Transpose Kernel

- **Alternative Kernel**:
  - Column-based transpose kernel using diagonal coordinates.
- **Code Sample**: Transpose kernel with column-based access.
- **Execution Command**: Run `./transpose 6` for row-based and `./transpose 7` for column-based transpose.

Performance Comparison (Table 4-10)

- **Bandwidth Efficiency**:
  - Row-based: 73.42 GB/s (41.32% of peak)
  - Column-based: 75.92 GB/s (42.72% of peak)
- **Conclusion**:
  - Column-based kernel performs better due to more efficient memory partition usage.

Partition Camping and Memory Access Patterns

- **Partition Camping Issue**:
  - Cartesian coordinates cause uneven memory partition usage, leading to partition camping.
  - Diagonal mapping distributes accesses more evenly across memory partitions.
- **Figures 4-29 and 4-30**: Show partition access patterns for Cartesian vs. diagonal mappings.

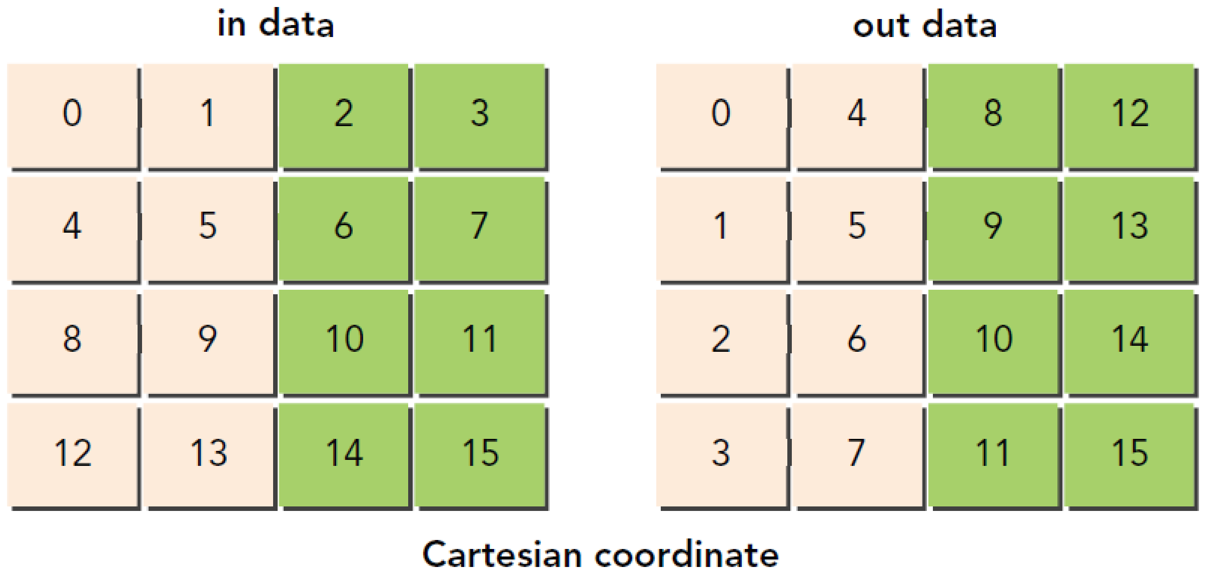


FIGURE 4-29

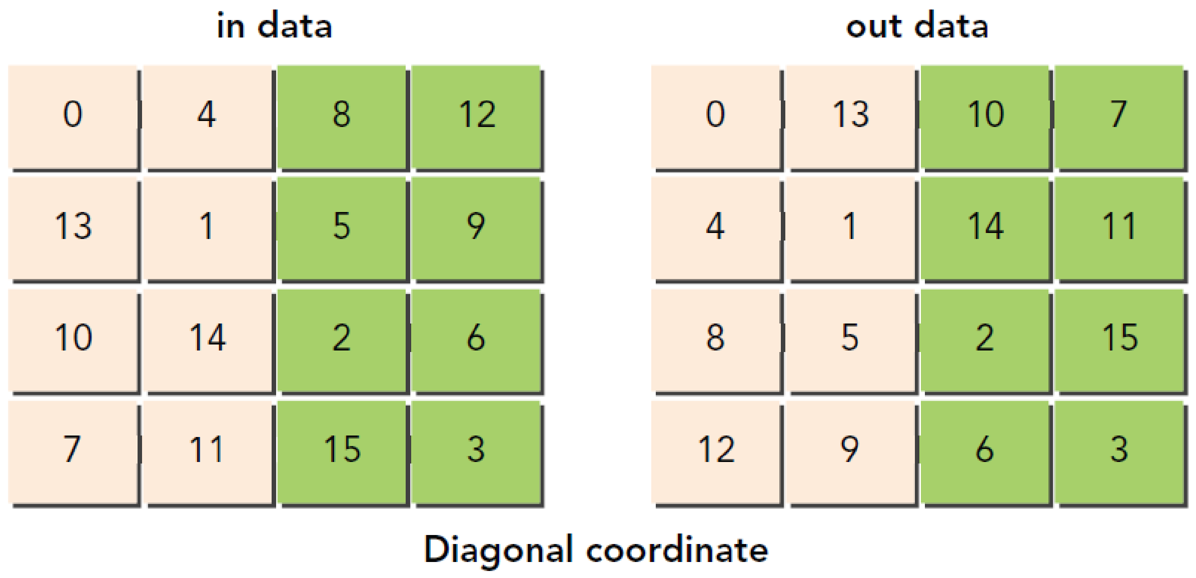


FIGURE 4-30

Exposing More Parallelism with Thin Blocks

- **Overview**:
  - Adjusting block size is an effective strategy to increase parallelism.
  - Experimenting with the `NaiveCol` kernel (column-based) with various block sizes shows performance differences.
- **Table 4-11**: Effective Bandwidth of Kernels with different block sizes (L1 Cache Disabled).

KERNEL	BLOCK SIZE	BANDWIDTH
`NaiveCol`	(32, 32)	38.13 GB/s
`NaiveCol`	(32, 16)	51.46 GB/s
`NaiveCol`	(32, 8)	54.82 GB/s
`NaiveCol`	(16, 32)	73.42 GB/s
`NaiveCol`	(16, 16)	80.27 GB/s
`NaiveCol`	(16, 8)	70.34 GB/s
`NaiveCol`	(8, 32)	102.76 GB/s
`NaiveCol`	(8, 16)	82.64 GB/s
`NaiveCol`	(8, 8)	59.59 GB/s

Optimal Block Size for Maximum Bandwidth

- **Best Block Size**:
  - The block size `(8, 32)` provides the highest bandwidth (102.76 GB/s), despite having the same parallelism as `(16, 16)`.
  - - This improvement is due to the "thin" block effect, increasing efficiency in store operations.
- **Figure 4-31**: Visual representation of thin blocks and block width.

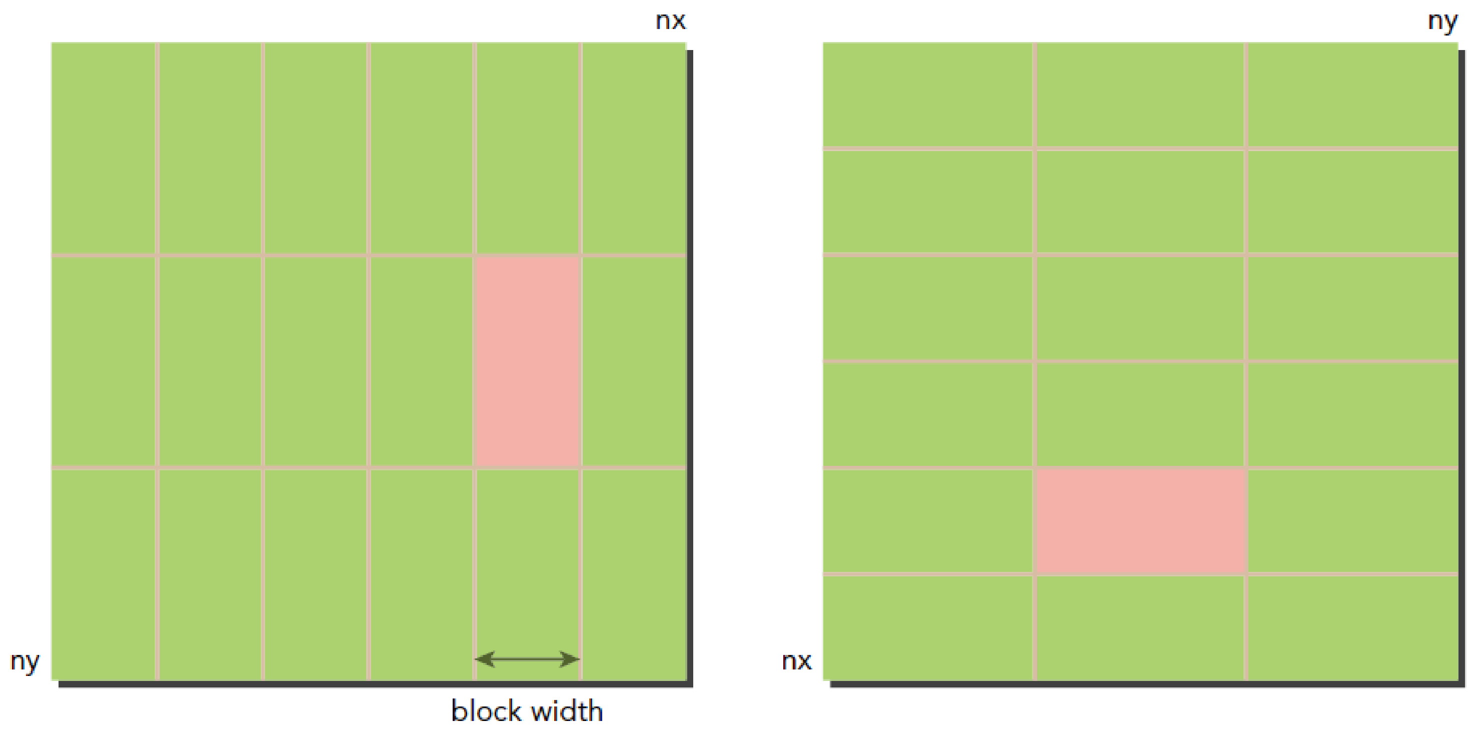
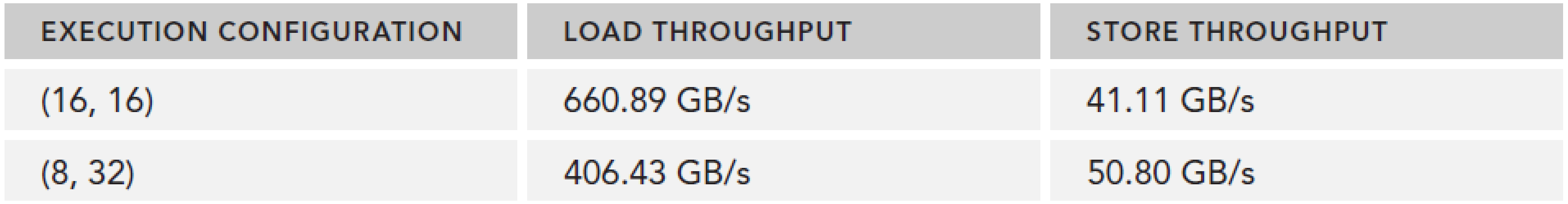


FIGURE 4-31

Load and Store Throughput Comparison

- **Using Metrics for Throughput**:
  - - Measured using `nvprof` to evaluate load and store throughput with different block sizes.
- **Table 4-12**: Comparison of load and store throughput between `(16, 16)` and `(8, 32)` block sizes.



EXECUTION CONFIGURATION	LOAD THROUGHPUT	STORE THROUGHPUT
(16, 16)	660.89 GB/s	41.11 GB/s
(8, 32)	406.43 GB/s	50.80 GB/s

Performance Comparison of Kernel Implementations

- **Testing Various Kernels**:
    - Commands for running different kernel implementations with block size `(8, 32)`.
- **Performance Insight**:
    - `Unroll4Col` kernel achieves the best effective bandwidth, surpassing the copy kernel.
- **Table 4-13**: Effective Bandwidth of Various Kernels with block size `(8, 32)` (L1 Cache Disabled).

KERNEL	BANDWIDTH	RATIO TO PEAK BANDWIDTH
Theoretical peak bandwidth	177.6	
`CopyRow` : Load/store using rows	102.30	57.57%
`NaiveRow` : Load rows/store columns	95.33	53.65%
`NaiveCol` : Load columns/store rows	101.99	57.39%
`Unroll4Row` : Load rows/store columns	82.04	46.17%
`Unroll4Col` : Load columns/store rows	113.36	63.83%

Conclusion and Key Takeaways

- **Thin Blocks and Bandwidth Efficiency**:
  - Thin block configurations (like `(8, 32)`) improve data access and maximize bandwidth.
  - Kernel optimizations (like `Unroll4Col`) can achieve high bandwidth, approaching 60-80% of peak theoretical bandwidth.
- **Practical Implications**:
  - Optimal block sizes and unrolling techniques are crucial for maximizing CUDA kernel performance.

Coming soon