

GPU Programming in Computer Vision

**Thomas Möllenhoff, Robert Maier,
Mohamed Souiai, Caner Hazırbaş**

CUDA Memories

**Technical University Munich, Computer Vision Group
Winter Semester 2014/2015, March 2 – April 3**

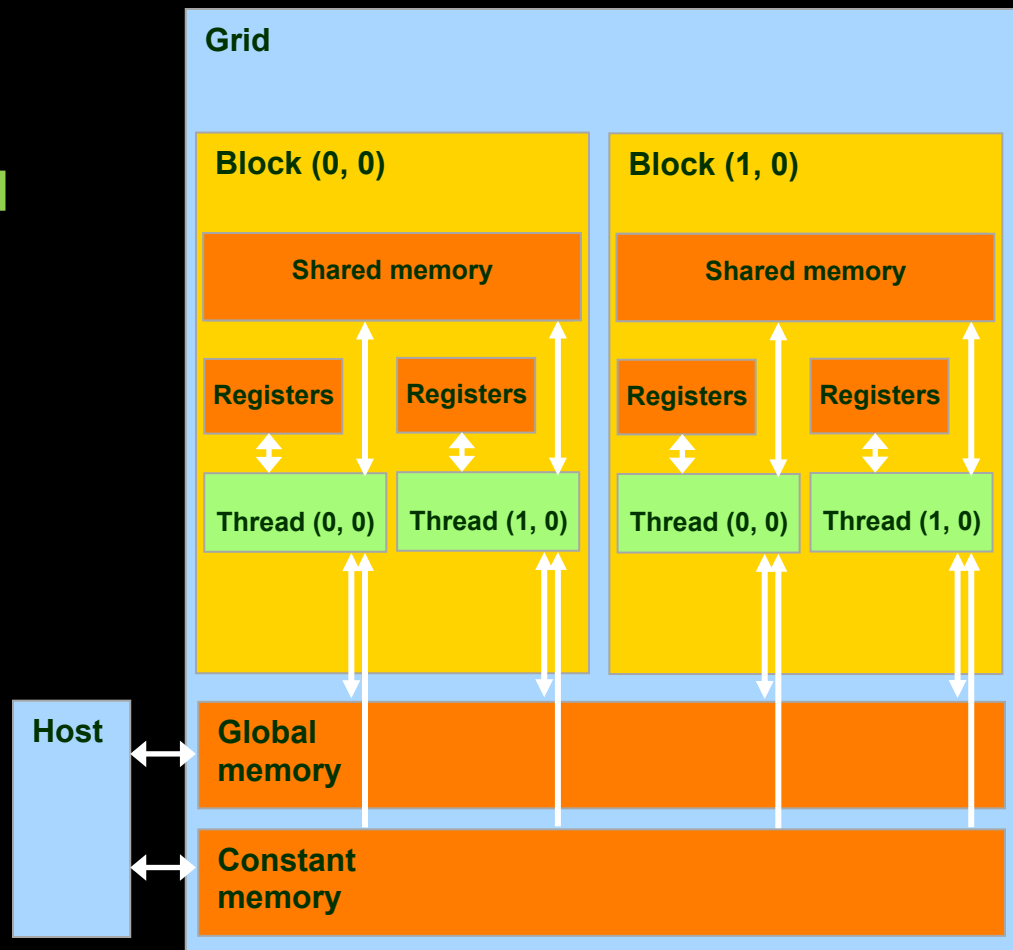
Outline

- Overview of Memory Spaces
- Shared Memory
- Texture Memory
- Constant Memory
- Common Strategy for Memory Accesses
- See the Programming Guide for more details

OVERVIEW OF MEMORY SPACES

CUDA Memories

- Each thread can:
 - read / write **per-thread registers**
 - read / write **per-block shared memory**
 - read / write **per-grid global memory**
 - read **per-grid constant memory**

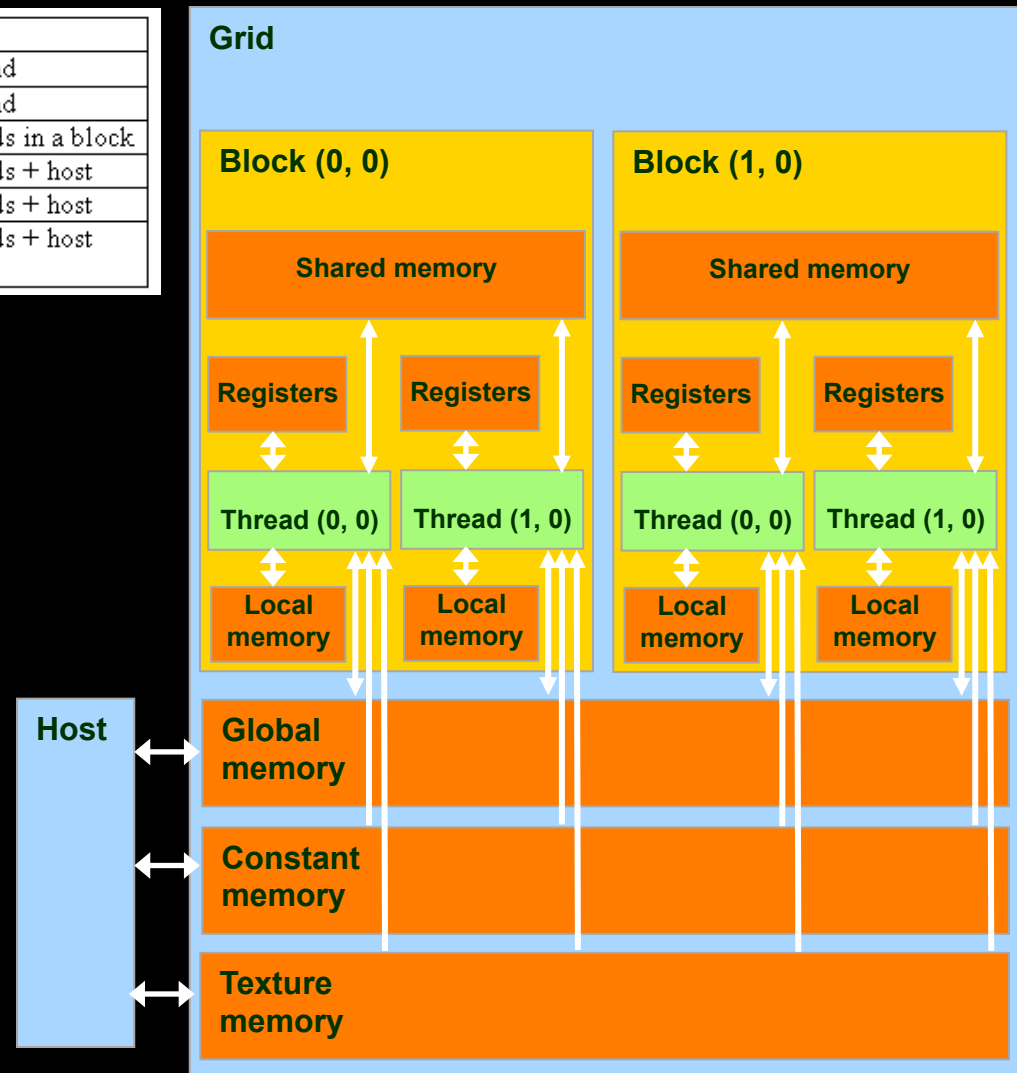


CUDA Memories

Memory	Location	Cached	Access	Scope
Register	On-chip	No	Read/write	One thread
Local	Off-chip	No	Read/write	One thread
Shared	On-chip	N/A	Read/write	All threads in a block
Global	Off-chip	No	Read/write	All threads + host
Constant	Off-chip	Yes	Read	All threads + host
Texture	Off-chip	Yes	Read (CUDA 2.1 and previous)	All threads + host

Other memories:

- **local Memory**
- **texture Memory**
 - both are part of global memory



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int var;</code>	register	thread	thread
<code>int array_var[10];</code>	local	thread	thread
<code>__shared__ int shared_var;</code>	shared	block	block
<code>__device__ int global_var;</code>	global	grid	application
<code>__constant__ int constant_var;</code>	constant	grid	application

- “automatic” scalar variables without qualifier reside in a register
 - compiler may spill to thread local memory
- “automatic” array variables without qualifier reside in thread local memory

CUDA Variable Type Performance

Variable declaration	Memory	Penalty
<code>int var;</code>	register	1x
<code>int array_var[10];</code>	local	100x
<code>__shared__ int shared_var;</code>	shared	1x
<code>__device__ int global_var;</code>	global	100x
<code>__constant__ int constant_var;</code>	constant	1x

- **scalar variables** reside in fast, on-chip registers
- **shared variables** reside in fast, on-chip memories
- **thread local arrays & global variables** reside in off-chip memory
- **constant variables** reside in cached off-chip

CUDA Variable Type Scale

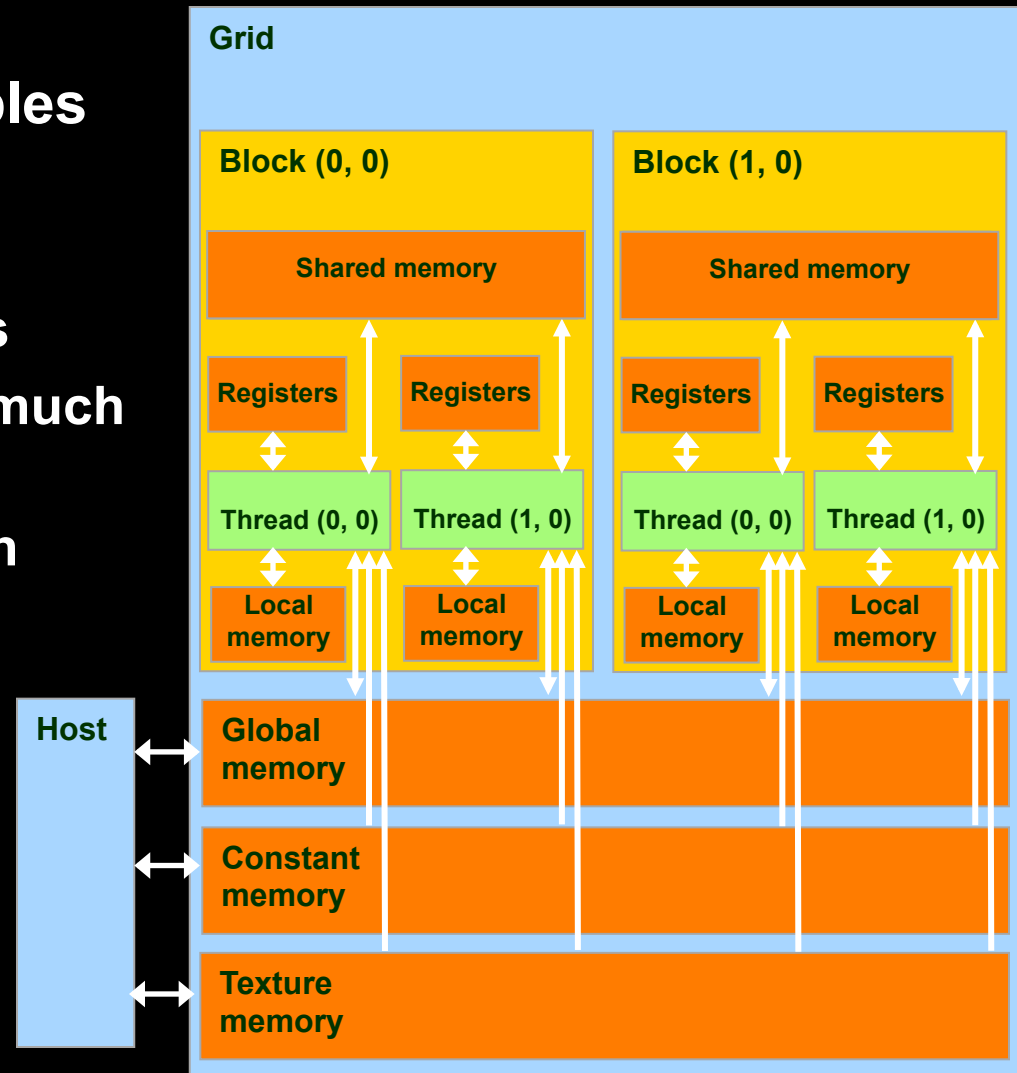
Variable declaration	Instances	Visibility
<code>int var;</code>	100,000s	1
<code>int array_var[10];</code>	100,000s	1
<code>__shared__ int shared_var;</code>	100s	100s
<code>__device__ int global_var;</code>	1	100,000s
<code>__constant__ int constant_var;</code>	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

Local Memory

Compiler might place variables in **local memory**:

- too many register variables
- a structure consumes too much register space
- an array is not indexed with constant quantities, i.e. when the addressing of the array is not known at compile time



Example: Thread Local Variables

```
__global__ void kernel(float2 *result, float2 *a, float2 *b)
{
```

```
// p goes in a register
float2 p = a[threadIdx.x];
```

Register

```
// big array, or indices are data dependent
float2 heap[10];
```

Local
memory

```
// small array, and indices known at compile time
float2 bvals[2];
bvals[0] = b[threadIdx.x];
bvals[1] = b[threadIdx.x + blockDim.x];
```

Register

```
...
```

```
}
```

SHARED MEMORY

Global and Shared Memory

- **Global memory is located off-chip**
 - high latency (often the bottleneck of computation)
 - important to minimize accesses
 - not cached for CC 1.x GPUs
 - main difficulty: try to coalesce accesses (more later)
- **Shared memory is on-chip**
 - low latency
 - like a user-managed per-multiprocessor cache
 - minor difficulty: try to minimize or avoid bank conflicts (more later)

Take Advantage of Shared Memory

- **Hundreds of times faster than global memory**
- **Threads can cooperate via shared memory**
- **Avoid multiple loads of same data by different threads of the block**
- **Use one/a few threads to load/compute data shared by all threads in the block**

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_global(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    float res = 0;
    if (i+1 < n)
    {
        // each thread loads two elements from global memory
        float xplus1 = input[i+1];
        float x0      = input[i];
        res = xplus1 - x0;
    }
    result[i] = res;
}
```

two loads

What are the bandwidth requirements of this kernel?

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_global(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    float res = 0;
    if (i+1 < n)
    {
        // each thread loads two elements from global memory
        float xplus1 = input[i+1];
        float x0      = input[i];
        res = xplus1 - x0;
    }
    result[i] = res;
}
```

again by thread $i-1$
once by thread i

**How many times does this
kernel load `input[i]`?**

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_global(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    float res = 0;
    if (i+1 < n)
    {
        // each thread loads two elements from global memory
        float xplus1 = input[i+1];
        float x0      = input[i];
        res = xplus1 - x0;
    }
    if (i<n) result[i] = res;
}
```

Idea:
eliminate redundancy
by **sharing data**

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_shared(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    int iblock = threadIdx.x; // local "block" version of i

    // allocate shared array, of constant size BLOCK_SIZE
    __shared__ float sh_data[BLOCK_SIZE];

    // each thread reads one element and writes into sh_data
    if (i<n) sh_data[iblock] = input[i];

    // ensure all threads finish writing before continuing
    __syncthreads();

    ...
}
```

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_shared(float *result, float *input, int n)
{
    ...
    float res = 0;
    if (i+1 < n)
    {
        // handle thread block boundary
        int xplus1 = (iblock+1 < blockDim.x ? sh_data[iblock+1] :
                                                             input[i+1]);
        int x0      = sh_data[iblock];
        res = xplus1 - x0;
    }
    if (i < n) result[i] = res;
}
```

Shared Memory: Example

```
// forward differences discretization of derivative
__global__
void diff_global(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;

    float res = 0;
    if (i+1 < n)
    {
        // each thread loads two elements
        float xplus1 = input[i+1];

        float x0      = input[i];
        res = xplus1 - x0;
    }
    if (i<n) result[i] = res;
}
```

```
// forward differences discretization of derivative
__global__
void diff_shared(float *result, float *input, int n)
{
    int i = threadIdx.x + blockDim.x*blockIdx.x;
    int iblock = threadIdx.x; // local version of i

    // allocate shared array
    __shared__ float sh_data[BLOCK_SIZE];

    // each thread reads one element to sh_data
    if (i<n) sh_data[iblock] = input[i];

    // ensure all loads complete before continuing
    __syncthreads();

    float res = 0;
    if (i+1 < n)
    {
        // handle thread block boundary
        float xplus1 = (iblock+1<blockDim.x?
                        sh_data[iblock+1] :
                        input[i+1]);
        float x0      = sh_data[iblock];
        res = xplus1 - x0;
    }
    if (i<n) result[i] = res;
}
```

Shared Memory: Dynamic Allocation

- **Size known at compile time**

```
__global__ void kernel (...)  
{  
    ...  
    __shared__ float s_data[BLOCK_SIZE];  
    ...  
}
```

```
int main(void)  
{  
    ...  
  
    kernel <<<grid,block>>> (...);  
    ...  
}
```

- **Size known at kernel launch**

```
__global__ void kernel (...)  
{  
    ...  
    extern __shared__ float s_data[];  
    ...  
}
```

```
int main(void)  
{  
    ...  
    // allocate enough shared memory  
    size_t smBytes = block.x * block.y * block.z  
                    * sizeof(float);  
    kernel <<<grid,block,smBytes>>> (...);  
    ...  
}
```

- **Always use dynamic allocation**

- flexibility w.r.t. maximal block size: can specify at run time
- no waste of resources: more blocks can run in parallel

Shared Memory: Synchronization

- `__syncthreads () ;`
- Synchronizes all threads **in a block**
 - generates a barrier synchronization instruction
 - no thread can pass this barrier **until all threads in the block reach it**
 - used to avoid Read-After-Write / Write-After-Read / Write-After-Write hazards for shared memory accesses
- Allowed in conditional code („if“, „while“, etc.)
only if the conditional is uniform across the block
 - e.g. **every** thread follows the same „if“- or „else“-path

Shared Memory: Synchronization

- Always use `__syncthreads()` after writing to shared memory to ensure that data is ready for accessing

```
__global__ void share_data(int *input)
{
    extern __shared__ int data[];
    data[threadIdx.x] = input[threadIdx.x];
    __syncthreads();
    // the state of the entire data array
    // is now well-defined for all threads in the block
}
```

- Don't synchronize or serialize unnecessarily

TEXTURE MEMORY

Texture Memory

- Actually part of global memory
- Read-only, **cached**
- Global memory reads are performed through **extra hardware** for texture manipulation



Textures

- **Texture** is a CUDA abstraction for **reading** data
- **Benefits:**
 - data is cached
 - optimized for 2D spatial locality
 - 32 B cache line (smaller than global mem cache line 128 B)
 - filtering (interpolation) with no additional costs
 - linear / bilinear / trilinear
 - wrap modes with no additional costs
 - for „out-of-bounds“ addresses
 - addressable in 1D, 2D, or 3D
 - using integer or normalized $[0,1)$ coordinates

Texture Usage: Overview

- Host (CPU) code:
 - allocate global memory
 - create a **texture reference** object
 - **bind** the texture reference to the allocated memory
 - use texture reference in kernels
 - when done: **unbind** texture reference
- Device (GPU) code:
 - fetch (read) using texture reference
 - **`tex1D(texRef, x)`** , **`tex2D(texRef, x, y)`** ,
`tex3D(texRef, x, y, z)`

Texture Usage: Texture Reference

- Define a **texture reference** at file scope:

```
texture <Type, Dim, ReadMode> texRef;
```

- **Type:** int, float, float2, float4, ...
- **Dim:** 1, 2, or 3, data dimension
- **ReadMode:**
 - **cudaReadModeElementType**
 - for integer-valued textures: return value as is
 - **cudaReadModeNormalizedFloat**
 - for integer-valued textures: normalize value to [0,1)

Texture Usage: Set Parameters

- Set boundary conditions for x and y
 - `texRef.addressMode[0] = cudaAddressModeClamp`
 - `texRef.addressMode[1] = cudaAddressModeClamp`
 - `cudaAddressModeClamp`, `cudaAddressModeWrap`
- Enable/disable filtering
 - `texRef.filterMode = cudaFilterModePoint`
 - `cudaFilterModePoint`, `cudaFilterModeLinear`
- Set whether coordinates are normalized to [0,1)
 - `texRef.normalized = false`

Texture Usage: Bind and Unbind

- **Bind texture to array**

cudaBindTexture2D

(NULL, &texRef, ptr, &desc, width, height, pitch)

- **ptr**: pointer to allocated array memory
- **width**: width of array
- **height**: height of array
- **pitch**: pitch of array in bytes
 - if ptr was allocated using cudaMalloc(), this is **width*sizeof(ptr[0])**
- **desc**: number of bits for each texture channel
 - **cudaCreateChannelDesc<float>()** // or float2, float4, int, ...

- **Unbind texture**

cudaUnbindTexture(texRef)

Textures: Example

```
texture<float,2,cudaReadModeElementType> texRef;    // at file scope

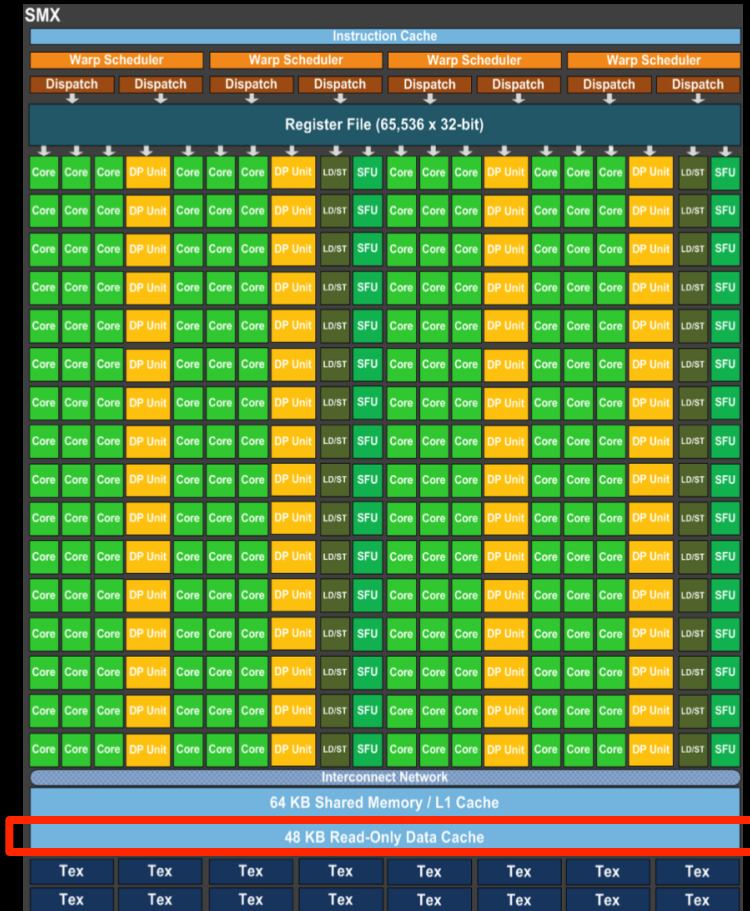
__global__ void kernel (...)
{
    int x = threadIdx.x + blockDim.x*blockIdx.x;
    int y = threadIdx.y + blockDim.y*blockIdx.y;
    float val = tex2D(texRef, x+0.5f, y+0.5f);    // add 0.5f to get center of pixel
    ...
}

int main()
{
    ...
    texRef.addressMode[0] = cudaAddressModeClamp;    // clamp x to border
    texRef.addressMode[1] = cudaAddressModeClamp;    // clamp y to border
    texRef.filterMode = cudaFilterModeLinear;        // linear interpolation
    texRef.normalized = false;    // access as (x+0.5f,y+0.5f), not as ((x+0.5f)/w,(y+0.5f)/h)
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
    cudaBindTexture2D(NULL, &texRef, d_ptr, &desc, w, h, w*sizeof(d_ptr[0]));
    kernel <<<grid,block>>> (...);
    cudaUnbindTexture(texRef);
    ...
}
```

CONSTANT MEMORY

Constant Memory

- Part of global memory
- Read-only, **cached**
 - cache is **dedicated**
 - same as for textures
 - will not be overwritten by other global memory reads
- **fast**
- **limited size (48 KB)**
 - few small crucial parameters



Constant Memory

- Defined at file scope
- Qualifier: **__constant__**
 - __constant__ float myparam;**
 - __constant__ float constKernel[KERNEL_SIZE];**
 - array size must be known, no dynamic allocation possible
- Reading only on device
 - **float val = myparam; val = constKernel[0];**
- Writing only on host
 - **cudaMemcpyToSymbol (constKernel, h_ptr, sizeBytes);**

A COMMON STRATEGY FOR MEMORY ACCESSES

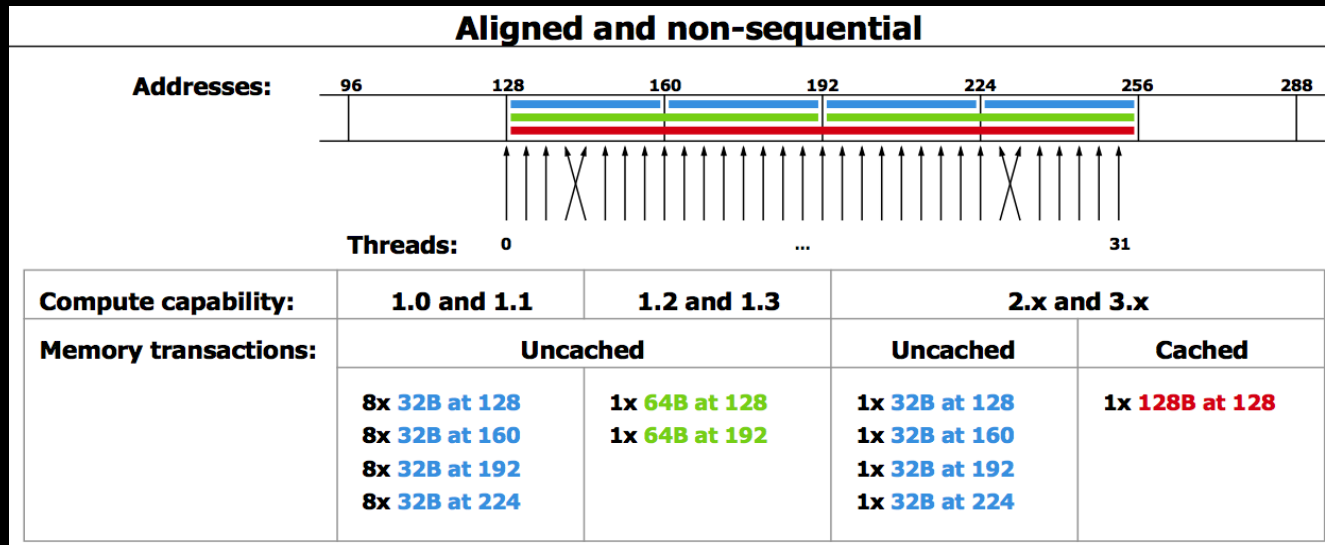
Global Memory: Coalescing

- Global memory access is **slow**
 - 400-800 clock cycles
- Hardware **coalesces** (combines) memory accesses
 - **chunks of size 32 B, 64 B, 128 B**
 - **aligned to multiples of 32 B, 64 B, 128 B, respectively**
- Coalescing is **per warp** (CC 1.x: per halfwarp)
 - each thread reads a **char**: $1\text{B} \times 32 = 32\text{ B chunk}$
 - each thread reads a **float**: $4\text{B} \times 32 = 128\text{ B chunk}$
 - each thread reads a **int2**: $8\text{B} \times 32 = 2 \times 128\text{ B chunks}$

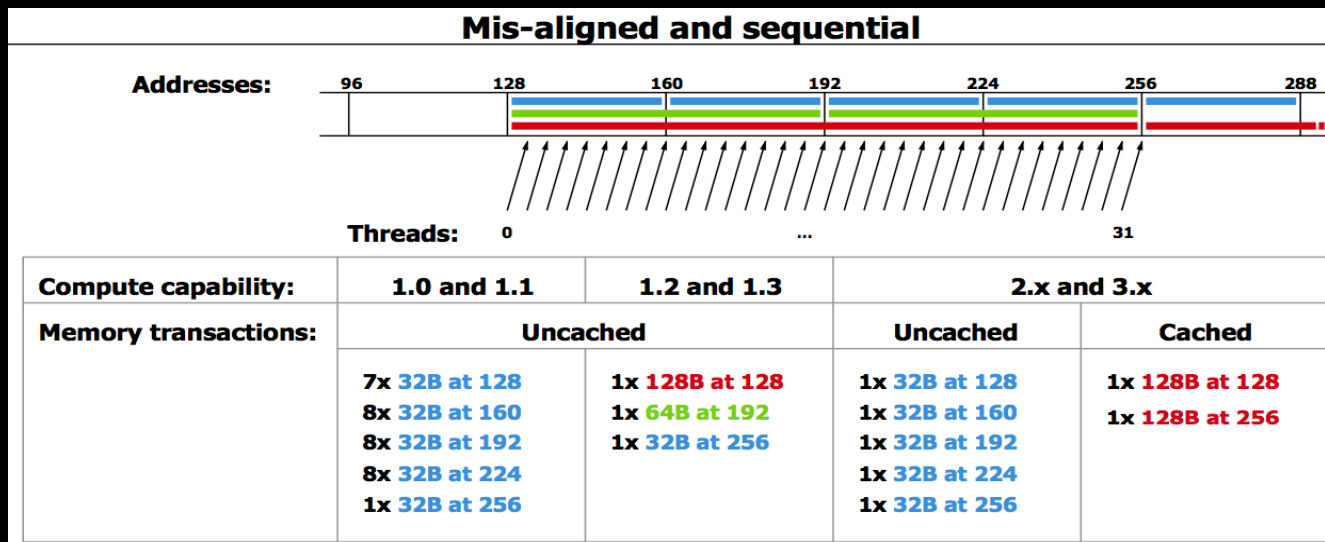
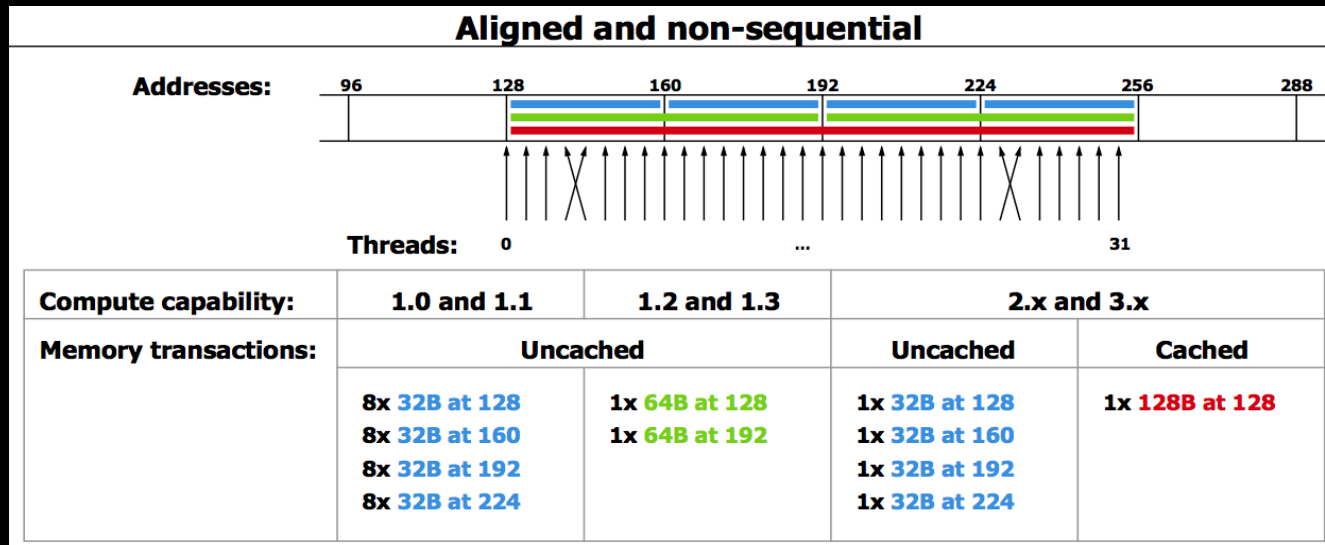
Global Memory: Coalescing

- Global memory access is **slow**
 - 400-800 clock cycles
- Make sure threads **within a warp** access
 - a **contiguous memory region**
 - as few **128 B segments as possible (CC>=2.0)**
 - CC >= 2.0: Cached accesses, cache line is always 128 B
 - CC 1.x: more restrictive as to when coalescing occurs
- **Huge performance hit** for non-coalesced accesses
 - memory accesses per warp will be **serialized**
 - worst case: reading **chars** from random locations

Global Memory: Coalescing



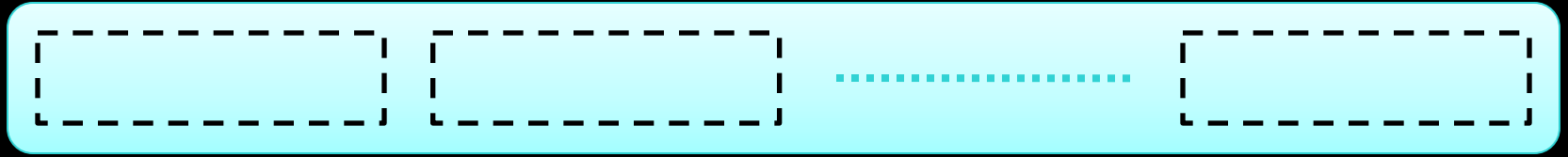
Global Memory: Coalescing



A Common Programming Strategy

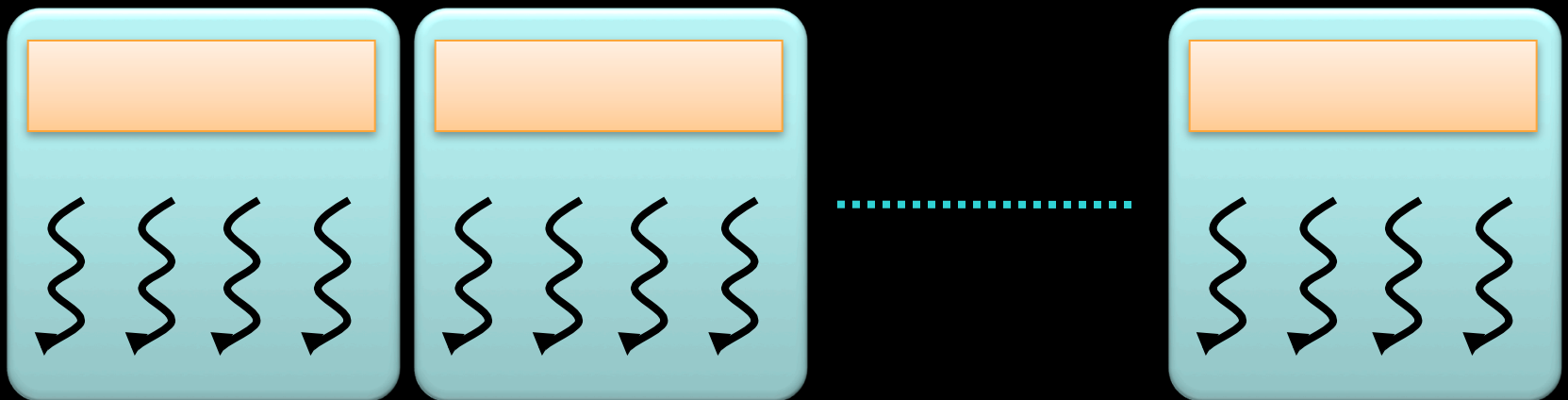
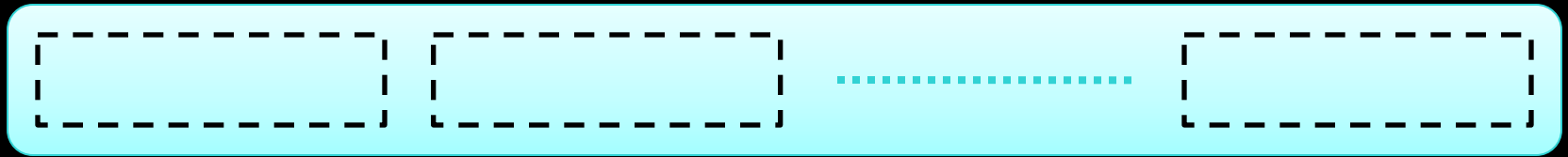
- 1. **Process data in chunks** to take advantage of fast shared memory
 - process each chunk in its own block
- 2. Load data **from global to shared** memory
 - **using as coalesced accesses as possible**
- 3. Process data **in shared** memory
 - freedom w.r.t. accesses: no coalescence requirements
- 4. Write data back **from shared to global** memory
 - **using as coalesced accesses as possible**

A Common Programming Strategy



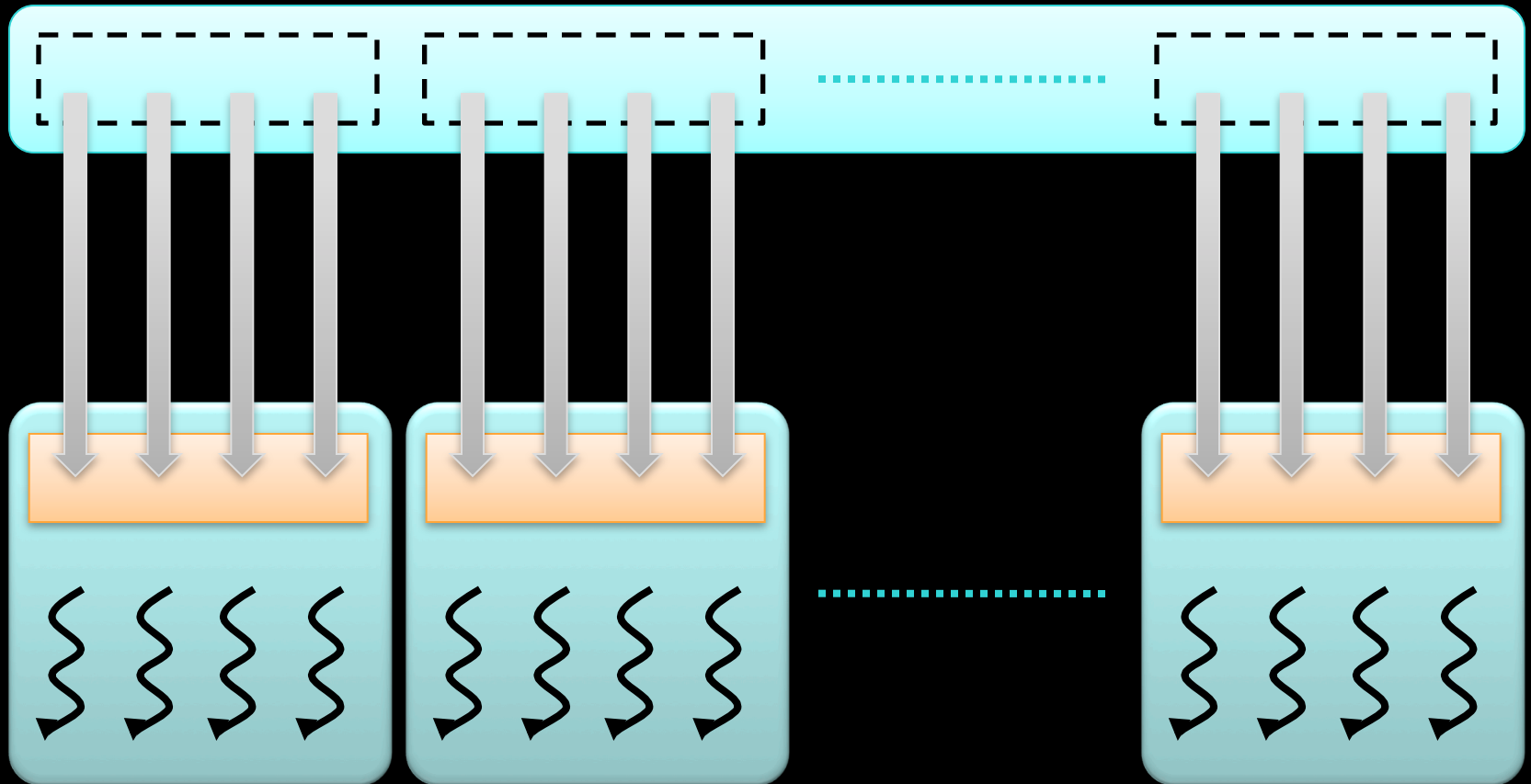
- Partition data into **several chunks**

A Common Programming Strategy



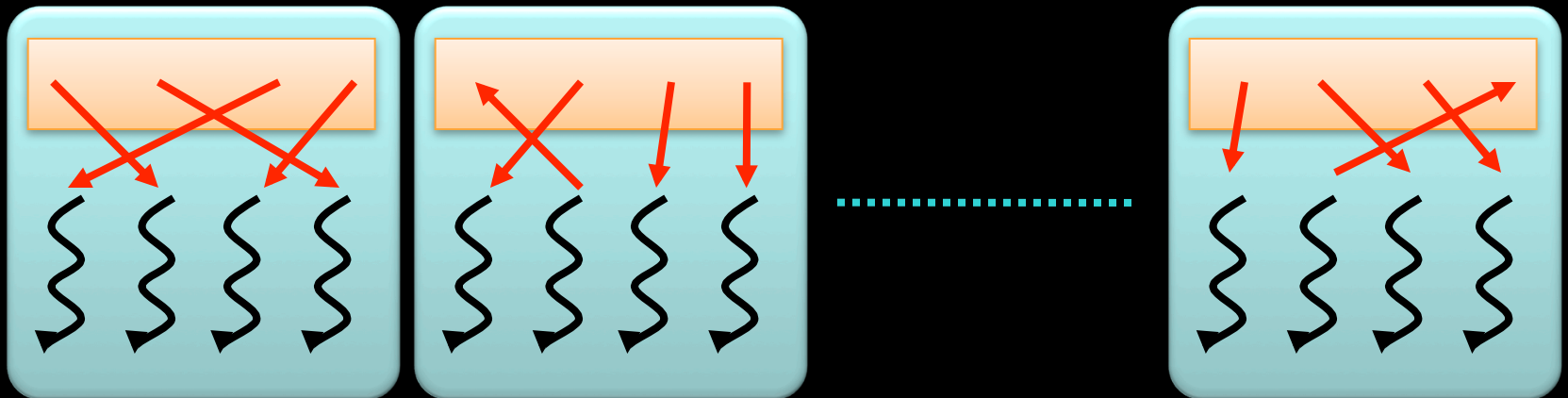
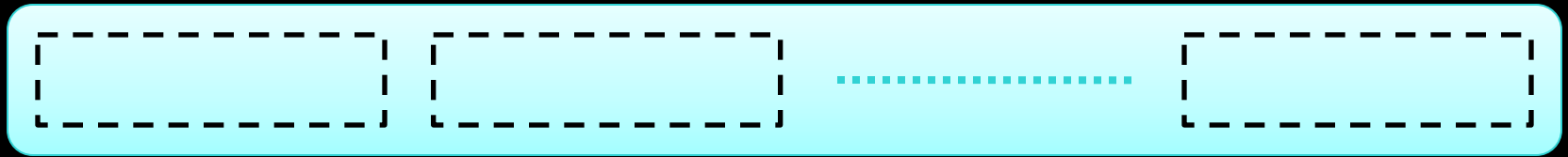
- **Handle each data chunk with one thread block**
 - each chunk **must fit** into shared memory for the block
 - this determines the maximal size of the chunks

A Common Programming Strategy



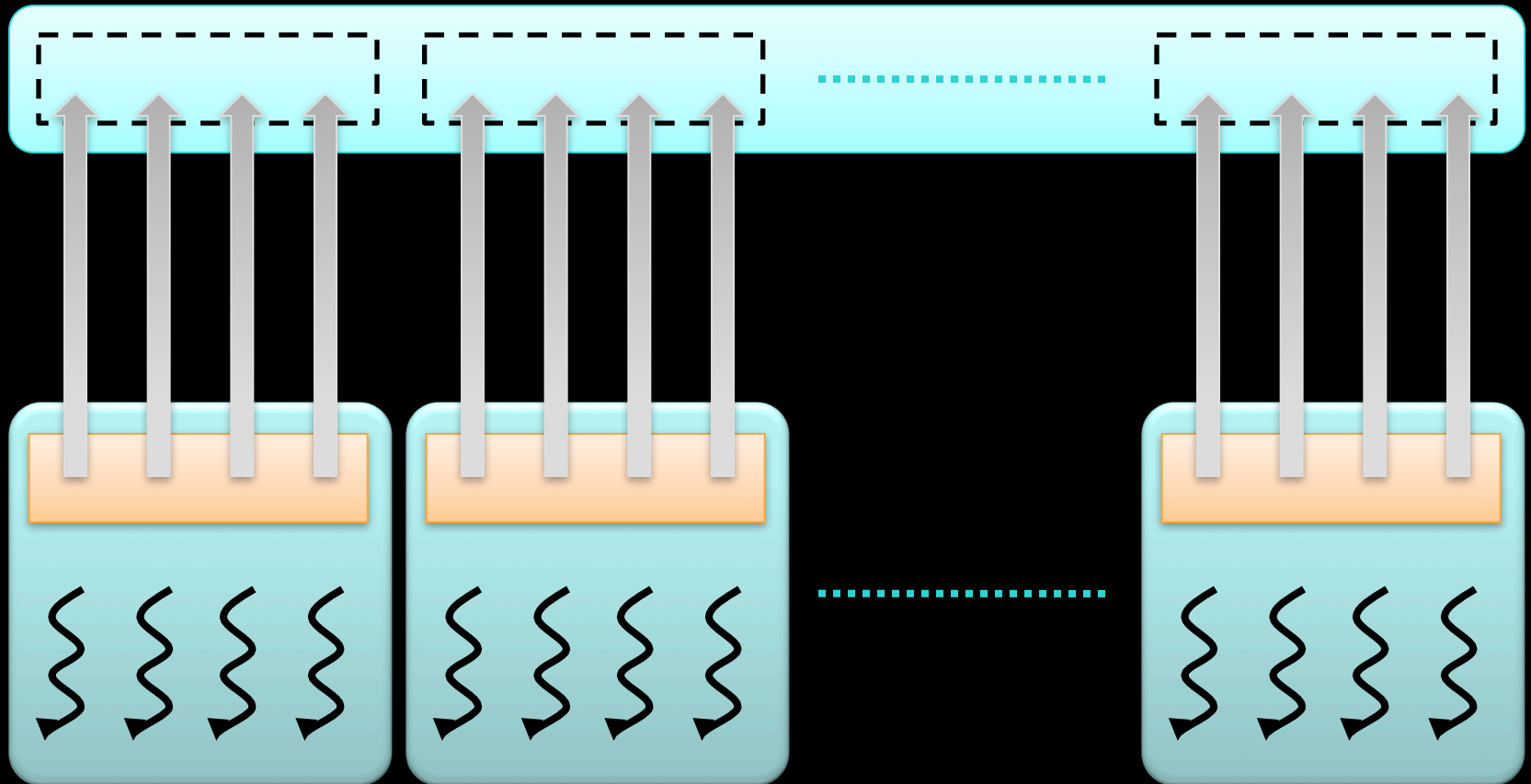
- Load data from global to shared memory
 - using as coalesced accesses as possible
 - distribute data loading across multiple threads

A Common Programming Strategy



- **Process data in shared memory**
 - much more freedom w.r.t. memory accesses
 - even random accesses may still be fast

A Common Programming Strategy



- Write data back from shared to global memory
 - using as coalesced accesses as possible
 - distribute data writing across multiple threads

The Most Important CUDA Optimization

- **Minimize the number of global memory accesses**
 - they are the slowest operations
 - essentially the only reason for slow kernel run time
- If you access global memory, **do it coalesced**
- **Rules of thumb:**
 - neighboring threads must access neighboring elements
 - `array[threadId.x + blockDim.x * blockIdx.x]`
 - two `float` arrays are better than one `float2` array
 - therefore: use layered memory layout for multi-channel images
 - if one value is used a lot in same thread: load in local variable
 - even if used just more than once
 - if one value is used by lots of threads: shared memory
 - but if used only by 2 or so threads, don't bother, global mem is still OK