GPU Programming in Computer Vision

CUDA Memories

Outline

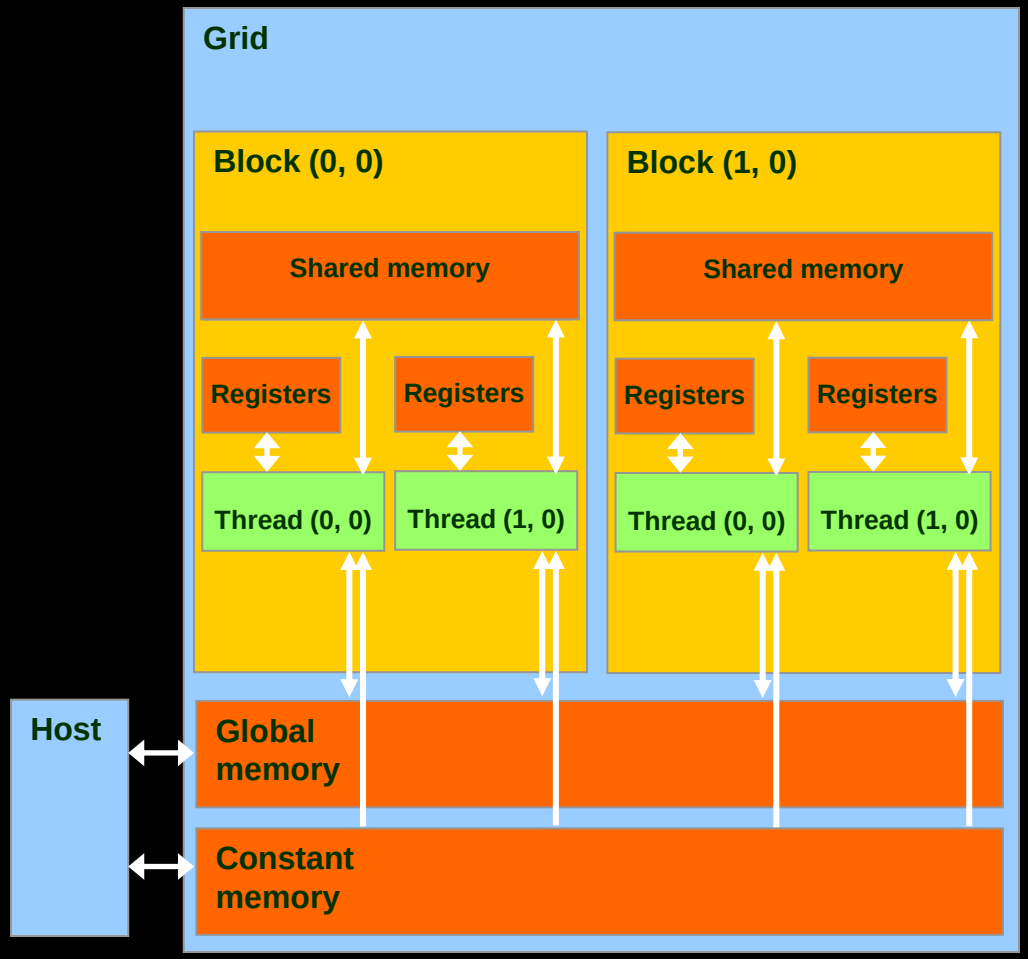
- Overview of Memory Spaces
- Shared Memory
- Texture Memory
- Common programming strategy for memory accesses

- 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/only per-grid constant memory

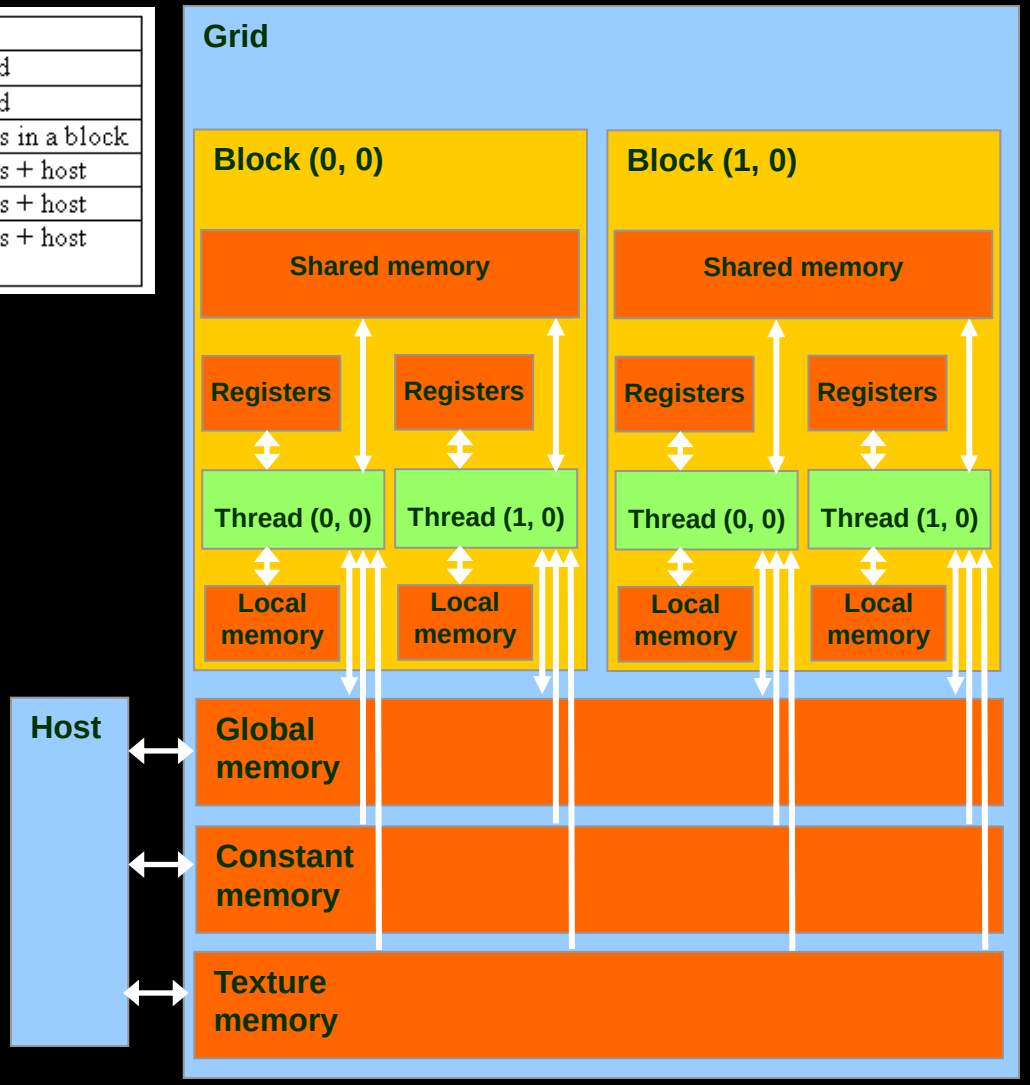


CUDA Memories

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
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	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
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

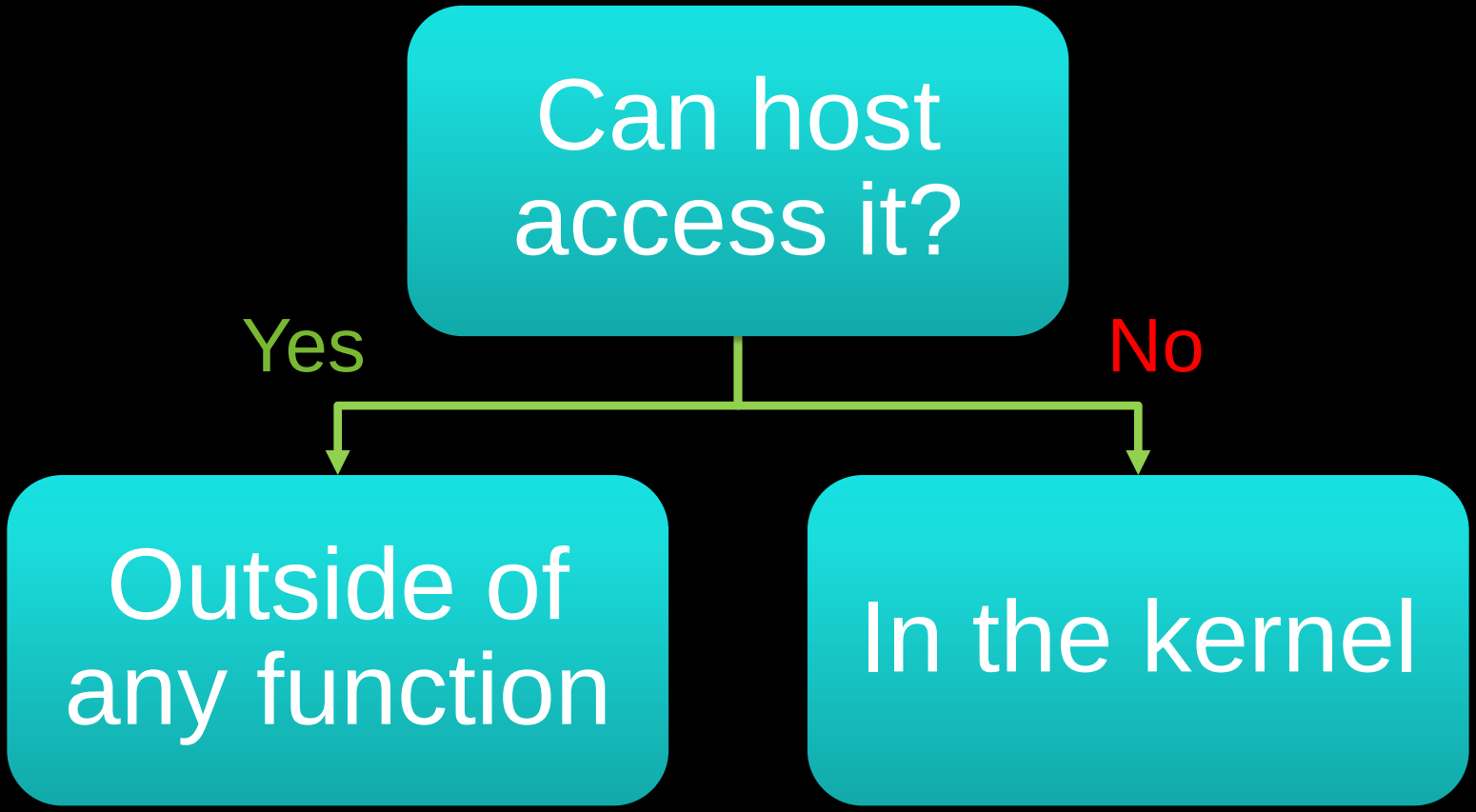
- **scalar variables** reside in fast, on-chip registers
- **shared variables** reside in fast, on-chip memories
- **thread local arrays** & **global variables** reside in off-chip memory
- **constant variables** reside in cached off-chip

CUDA Variable Type Scale

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

Where to declare variables?




```
__constant__ int constant_var;
__device__   int global_var;
```

```
int var;
int array_var[10];
__shared__ int shared_var;
```

Example: Thread local variables

```
// motivate per-thread variables with
// Ten Nearest Neighbors application
__global__ void ten_nn(float2 *result, float2 *ps, float2 *qs,
                       size_t num_qs)
{
  // p goes in a register
  float2 p = ps[threadIdx.x];

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

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

Register

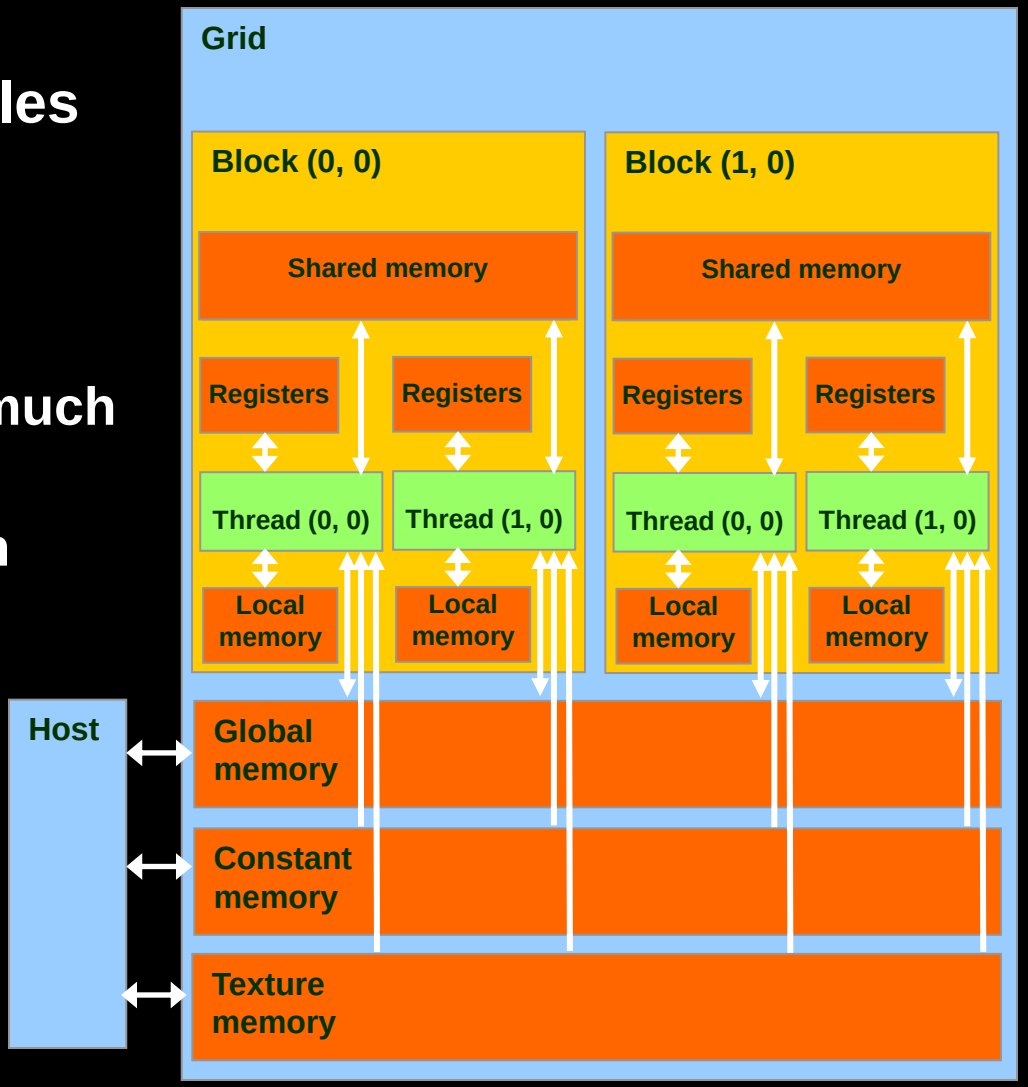
Local memory

Register

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



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_naive(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_naive(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_naive(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;
}
```

Idea:
eliminate redundancy
by sharing data

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_shm(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
  sh_data[iblock] = input[i];

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

  ...
}
```

Shared Memory: Example

```
// forward differences discretization of derivative
__global__ void diff_shm(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[i];
    res = xplus1 - x0;
  }
  result[i] = res;
}
```

Shared Memory: Example

```
// forward differences discretization of derivative
__global__
void diff_naive(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;
}
```

```
// forward differences discretization of derivative
__global__
void diff_shm(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, of constant size
     BLOCK_SIZE
  __shared__ float sh_data[BLOCK_SIZE];

  // each thread reads one element to sh_data
  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[i];
    res = xplus1 - x0;
  }
  result[i] = res;
}
```

Optimization Analysis

Implementation	Original	Improved
Global Loads	2N	N + N/BLOCK_SIZE
Global Stores	N	N
Throughput	36.8 GB/s	57.5 GB/s
SLOCs (# lines of code)	18	35
Relative Improvement	1x	1.57x
Improvement/SLOC	1x	0.81x

- **Experiment performed on a GT200 chip**
  - **Improvement likely better on an older architecture**
  - **Improvement likely worse on a newer architecture**
- **Optimizations tend to come with a development cost**

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)
{
  ...
  int smBytes = block.x*block.y
          *block.z*sizeof(float);
  kernel <<<grid,block,smBytes>>> (...);
  ...
}
```

Synchronizing Threads within a Block

- `__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

TEXTURE MEMORY

Texture Memory

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

Textures: Utilize Texture Memory

- **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 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

Textures: Usage (General)

- Host (CPU) code:
  - allocate global memory
  - create a texture reference object
  - bind the texture reference to the allocated memory
  - when done: unbind texture reference
- Device (GPU) code:
  - Fetch (reads) using texture reference
  - tex1D(texRef,x), tex2D(texRef,x,y), tex3D(texRef,x,y,z)

Textures: 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)

Textures: 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

Textures: Usage (Bind/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**
  - **desc: number of bits for each texture channel**
    - cudaCreateChannelDesc<float>() // or float2, float4, int, …

- **Unbind texture**

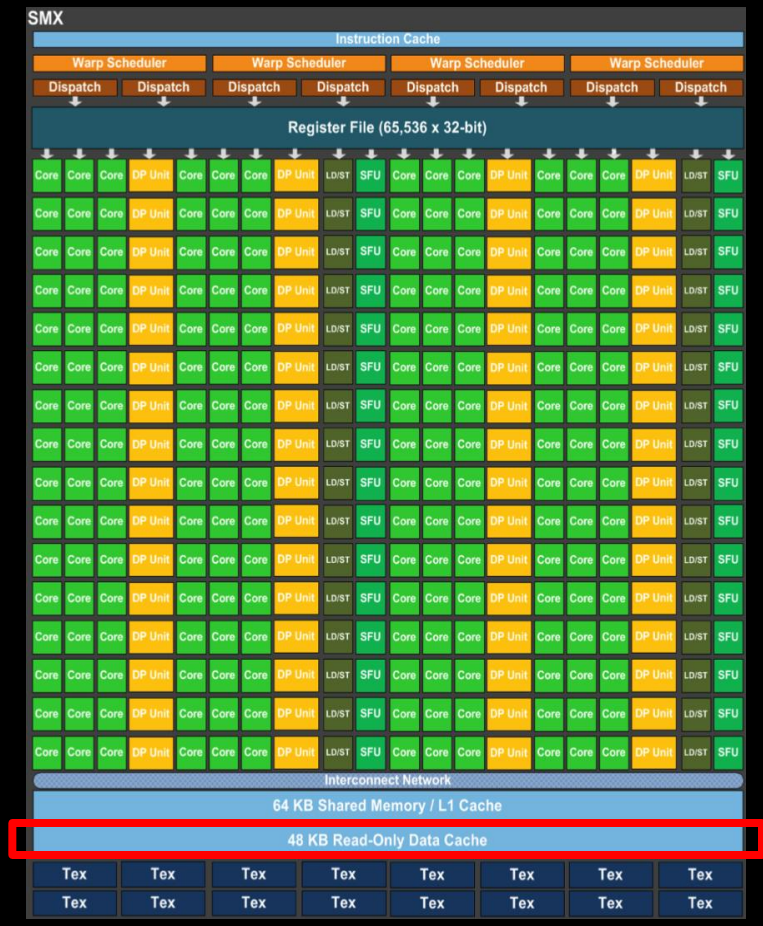
  cudaUnbindTexture(texRef)

Textures: Example

```
texture<float,2,cudaReadModeElementType> tex1;   // 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(tex1, x+0.5f, y+0.5f);   // add 0.5f to get center of pixel
    ...
}

int main(void)
{
    ...
    cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
    tex1.addressMode[0] = cudaAddressModeClamp;   // clamp x to border
    tex1.addressMode[1] = cudaAddressModeClamp;   // clamp y to border
    tex1.filterMode = cudaFilterModeLinear;       // linear interpolation
    tex1.normalized = false;   // access as (x+0.5f,y+0.5f), not as ((x+0.5f)/w,(y+0.5f)/h)
    cudaBindTexture2D(NULL, &tex1, d_ptr, &desc, w, h, pitchBytes);
    kernel <<<grid,block>>> (...);
    cudaUnbindTexture(tex1);
    ...
}
```

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 as file scope
- Qualifier: __constant__

```
__constant__ float myparam;
__constant__ float constKernel[KERNEL_SIZE];
```

- Read from device
  - float val = myparam; val = constKernel[0];
- Write from host
  - cudaMemcpyToSymbol (constKernel, h_ptr, sizeBytes);

COMMON PROGRAMMING STRATEGY FOR MEMORY ACCESSES

Global Memory: Coalescing

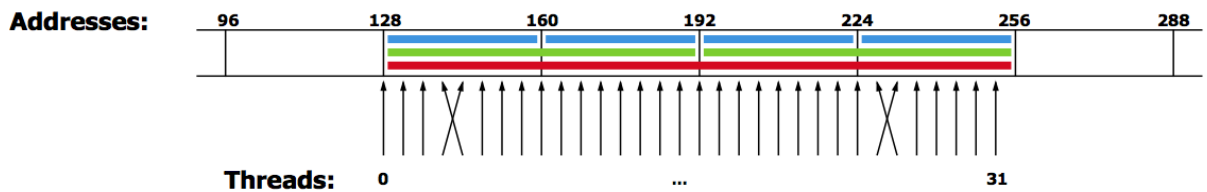
- Global memory access is slow
  - 400-800 clock cycles
- Hardware coalesces (combines) memory accesses
  - chunks of size 32 B, 64 B, 128 B
  - aligned to multiples of 32 B, 64 B, 128 B, respectively
- Coalescing is per warp (CC 1.x: per halfwarp)
  - each thread reads a char: 1B*32 = 32 B chunk
  -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

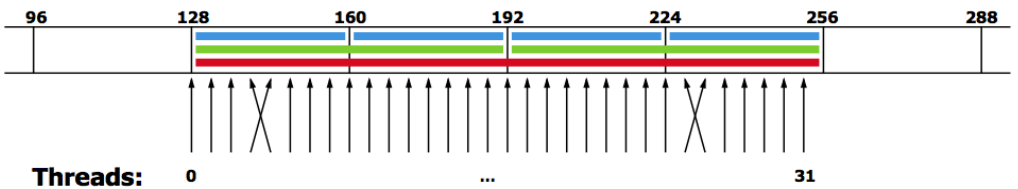
Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

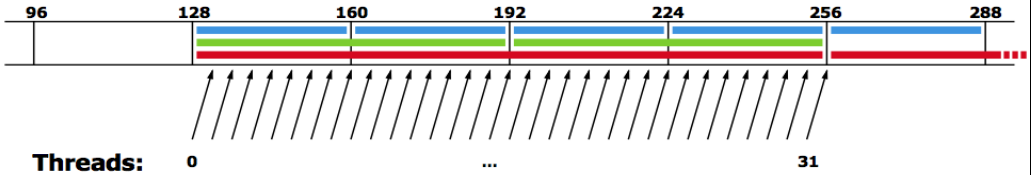
Global Memory: Coalescing

Aligned and non-sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis-aligned and sequential



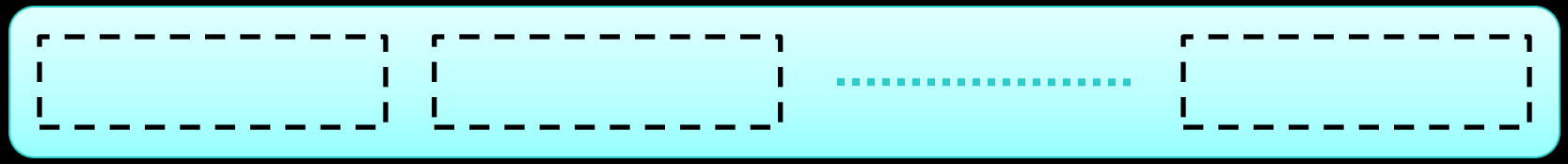
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

A Common Programming Strategy

- Global memory access is slow (400-800 clock cycles)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory
  - process each subset in an own thread block

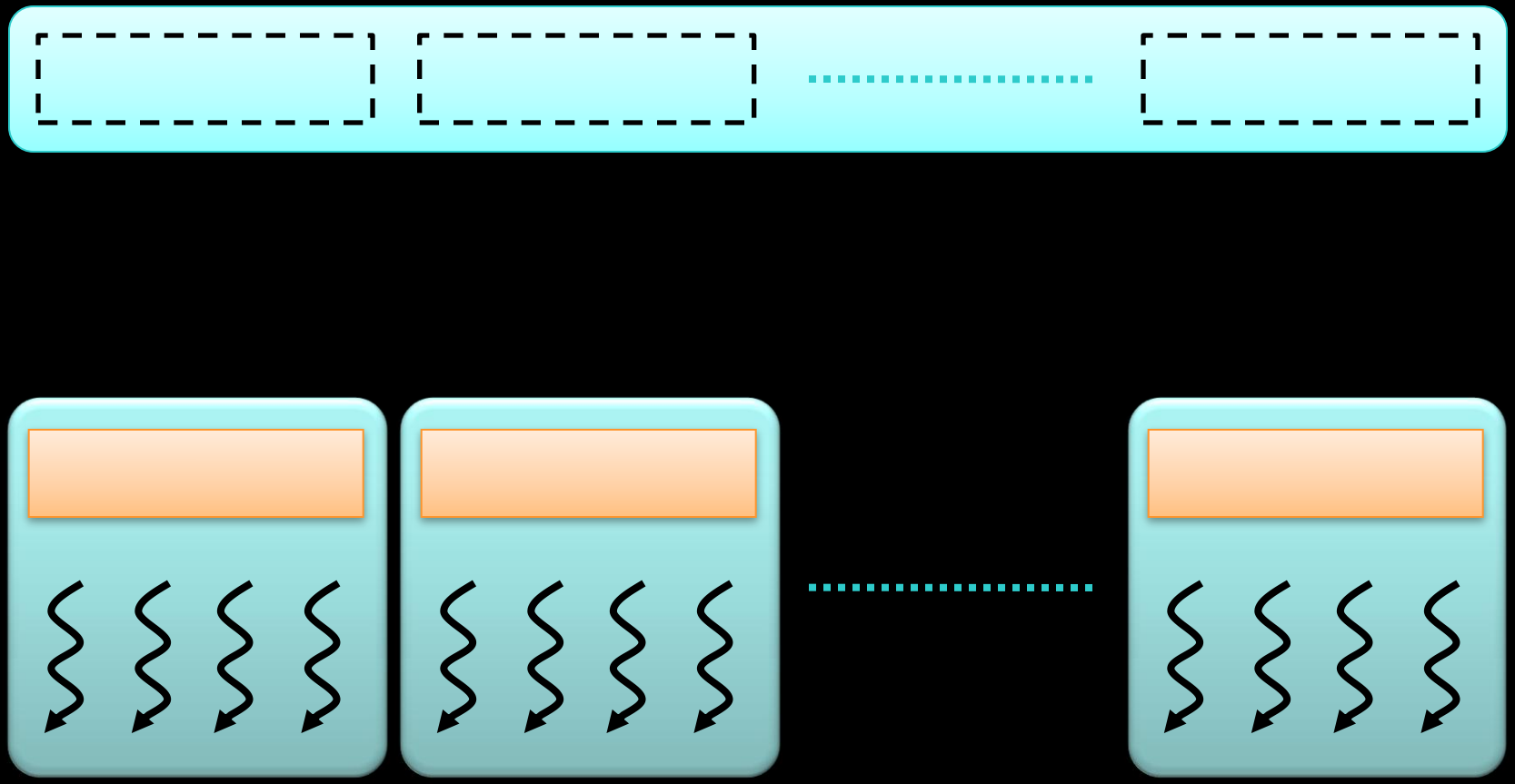
- Load data from global memory to shared memory
  - using as coalesced accesses as possible
- Process data in shared memory
- Store data back to global memory
  - using as coalesced accesses as possible

A Common Programming Strategy



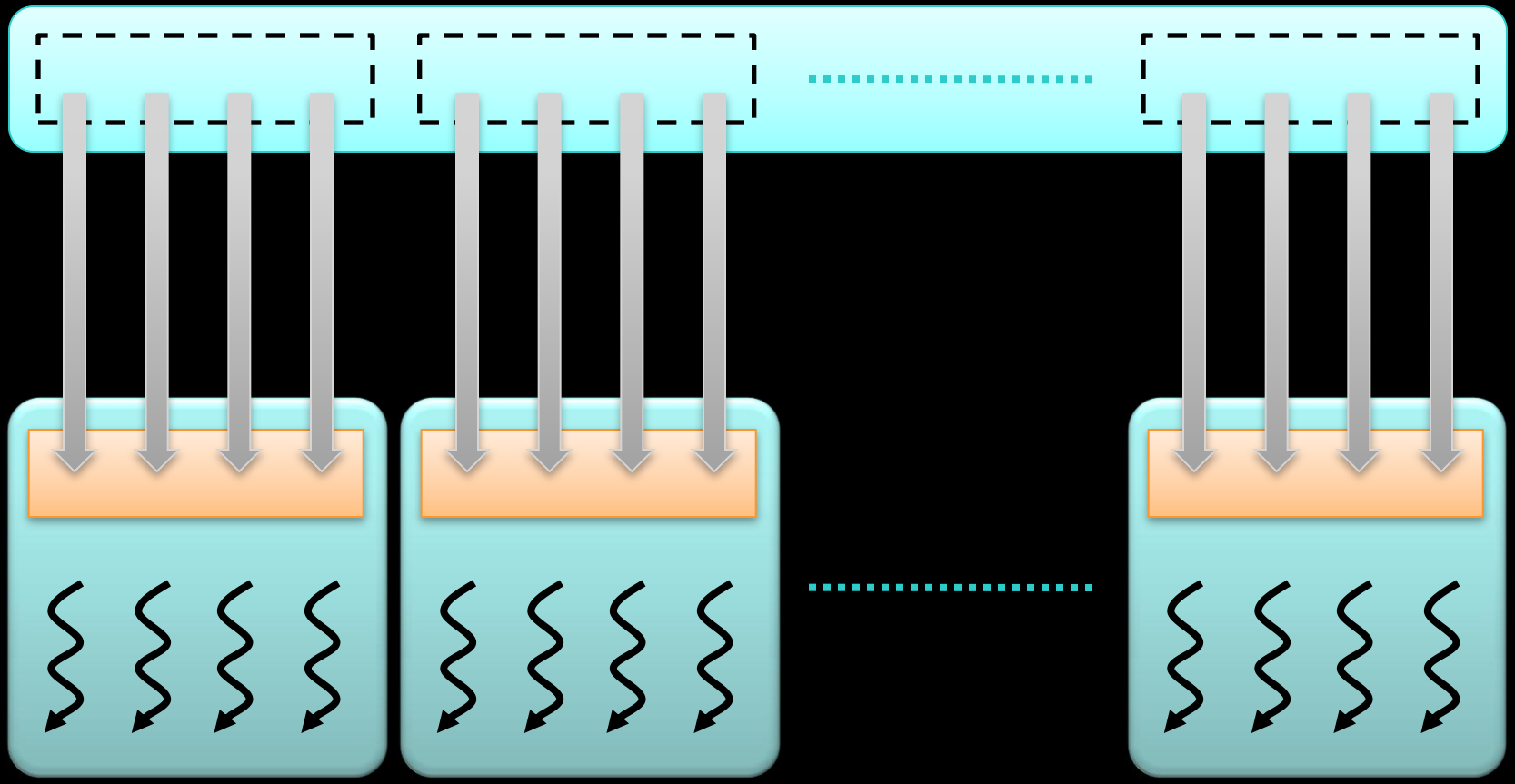
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



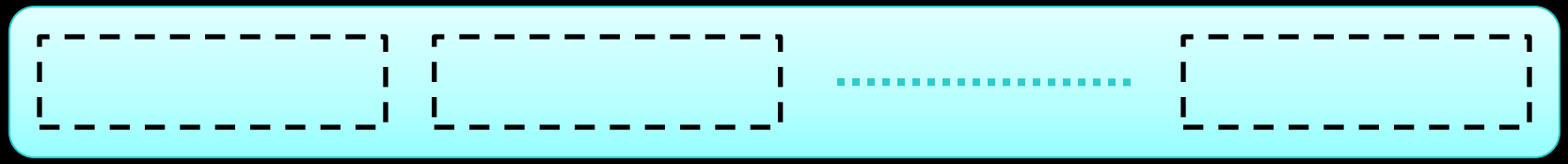
- Handle each data subset with one thread block

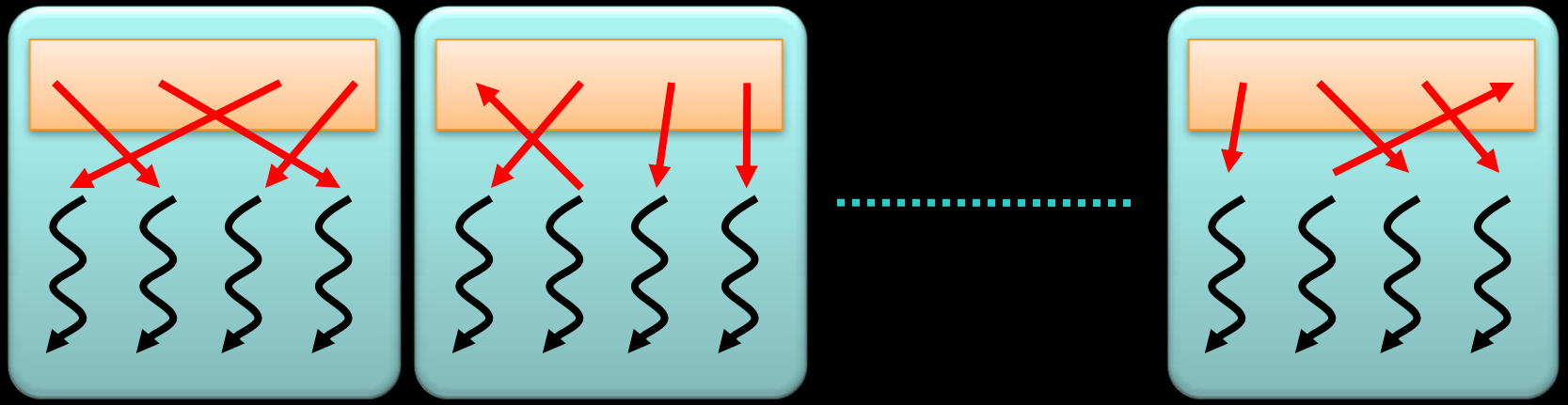
A Common Programming Strategy



- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

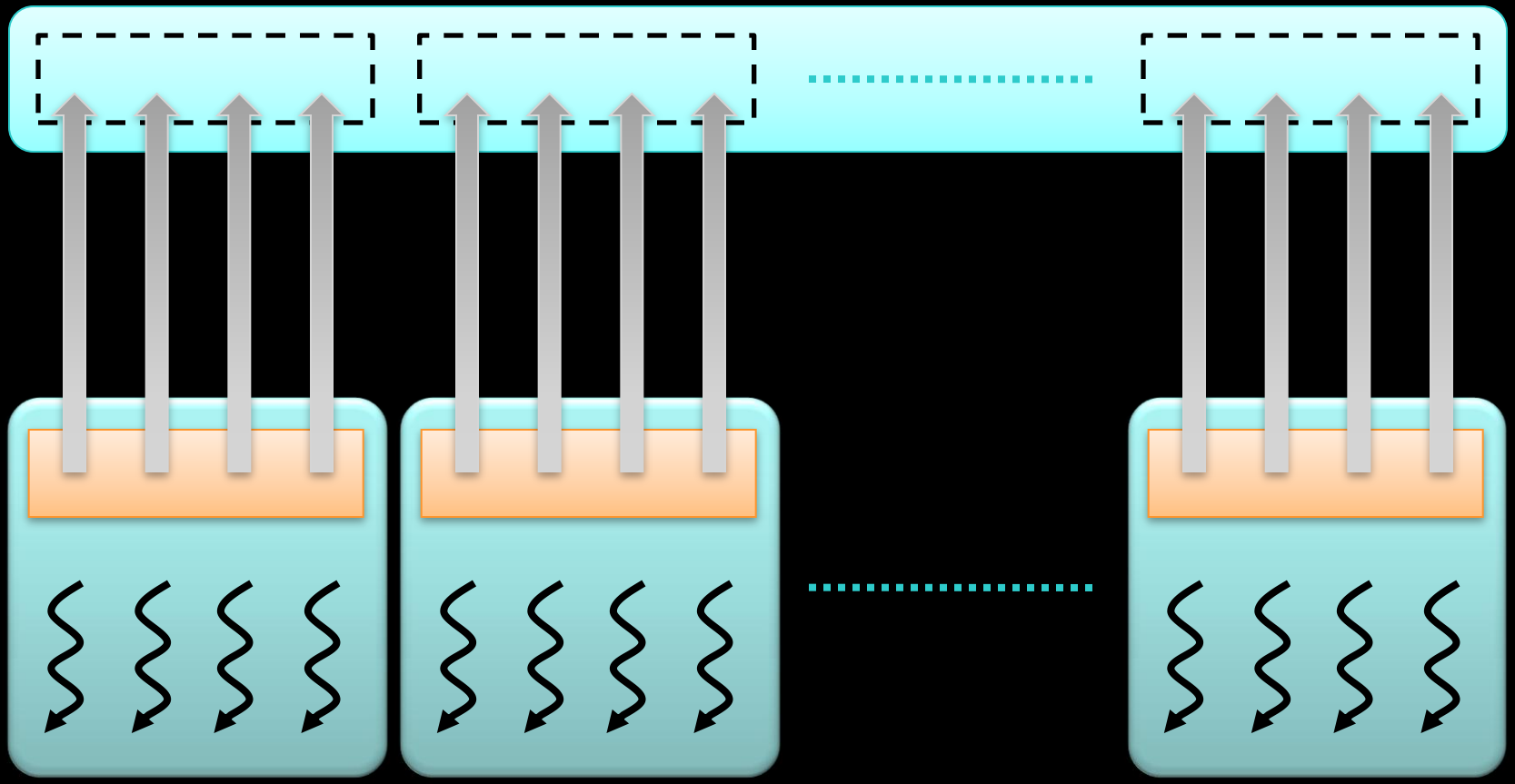
A Common Programming Strategy





- Perform the computation on the subset from shared memory

A Common Programming Strategy



- Copy the result from shared memory back to global memory

A Common Programming Strategy

- Carefully partition data according to access patterns
- Read-only ➔ `__constant__` memory (fast)
- R/W & shared within block ➔ `__shared__` memory (fast)
- R/W within each thread ➔ registers (fast)
- Indexed R/W within each thread ➔ local memory (slow)
- R/W inputs/results ➔ `cudaMalloc`‘ed global memory (slow)

Communication Through Memory

- Question:

```
__global__ void race(void)
{
  __shared__ int my_shared_variable;
  my_shared_variable = threadIdx.x;

  // what is the value of
  // my_shared_variable?
}
```

Communication Through Memory

- This is a **race condition**
- The result is **undefined**
- The order in which threads access the variable is undefined without explicit coordination
- Use barriers (`e.g., __syncthreads`) or atomic operations (e.g., `atomicAdd`) to enforce **well-defined** semantics

Communication Through Memory

- Use `__syncthreads` to ensure data is ready for access

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

Communication Through Memory

- Use atomic operations to ensure exclusive access to a variable

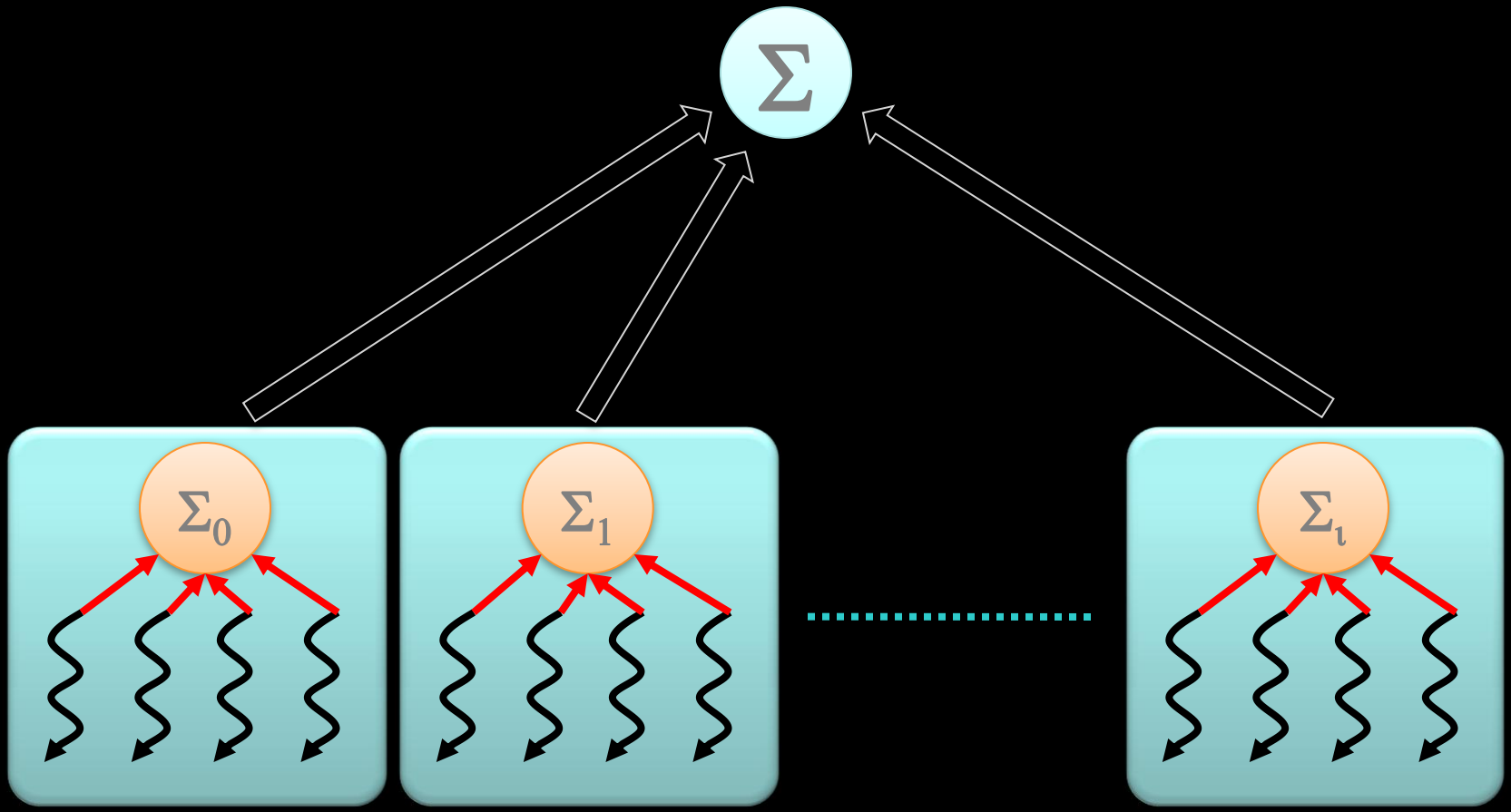
```
// assume *result is initialized to 0
__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);

  // after this kernel exits, the value of
  // *result will be the sum of the input
}
```

Resource Contention

- Atomic operations aren't cheap!
- They imply serialized access to a variable

```
__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);
}
...
// how many threads will contend
// for exclusive access to result?
sum<<<B,N/B>>>(input,result);
```

Hierarchical Atomics



- Divide & Conquer
  - Per-thread `atomicAdd` to a `__shared__` partial sum
  - Per-block `atomicAdd` to the total sum

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
  __shared__ int partial_sum;

  // thread 0 is responsible for
  // initializing partial_sum
  if(threadIdx.x == 0)
    partial_sum = 0;
  __syncthreads();

  ...
}
```

Hierarchical Atomics

```
__global__ void sum(int *input, int *result)
{
  ...
  // each thread updates the partial sum
  atomicAdd(&partial_sum,
            input[threadIdx.x]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0)
    atomicAdd(&result, partial_sum);
}
```

Advice

- Use barriers such as `__syncthreads` to wait until `__shared__` data is ready
- Prefer barriers to atomics when data access patterns are regular or predictable
- Prefer atomics to barriers when data access patterns are sparse or unpredictable
- Atomics to `__shared__` variables are much faster than atomics to global variables
- Don’t synchronize or serialize unnecessarily

Final Thoughts

- Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase throughput
- Use `__shared__` memory to eliminate redundant loads from global memory
  - Use `__syncthreads` barriers to protect `__shared__` data
  - Use atomics if access patterns are sparse or unpredictable
- Optimization comes with a development cost
- Memory resources ultimately limit parallelism