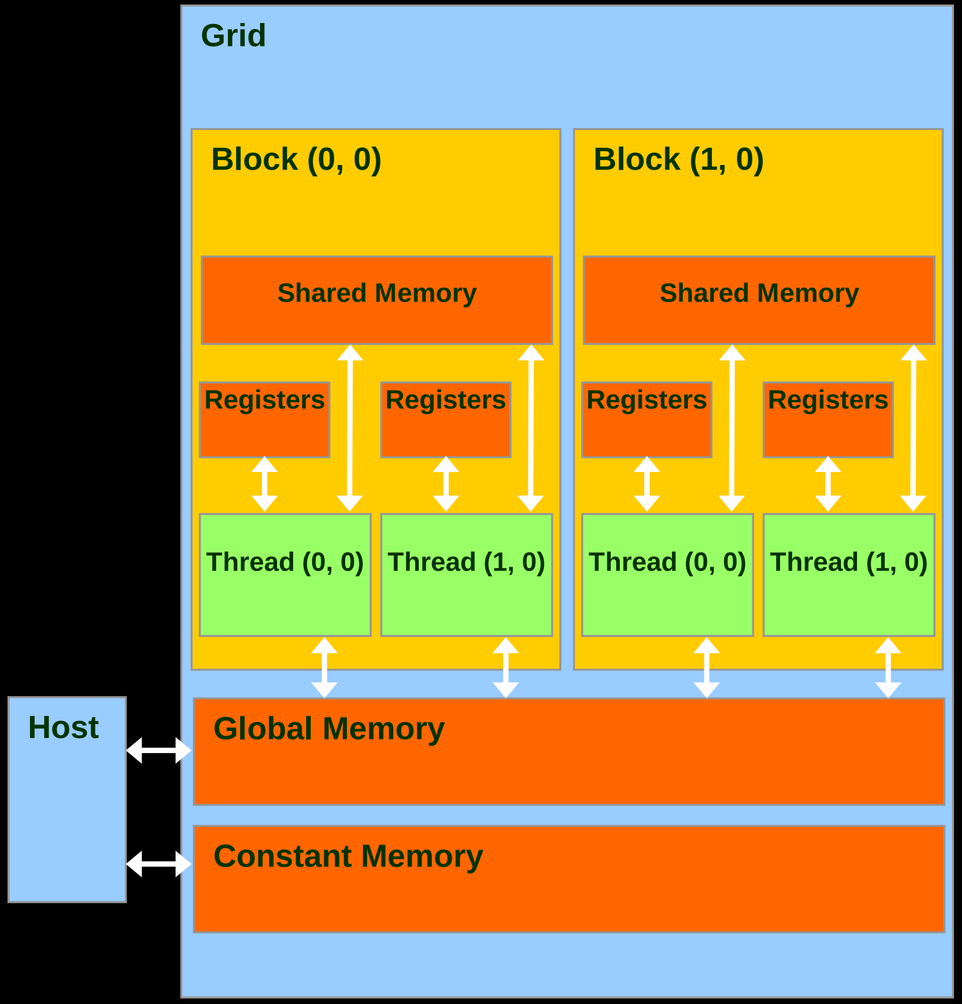
GPU Programming in Computer Vision

CUDA Memories

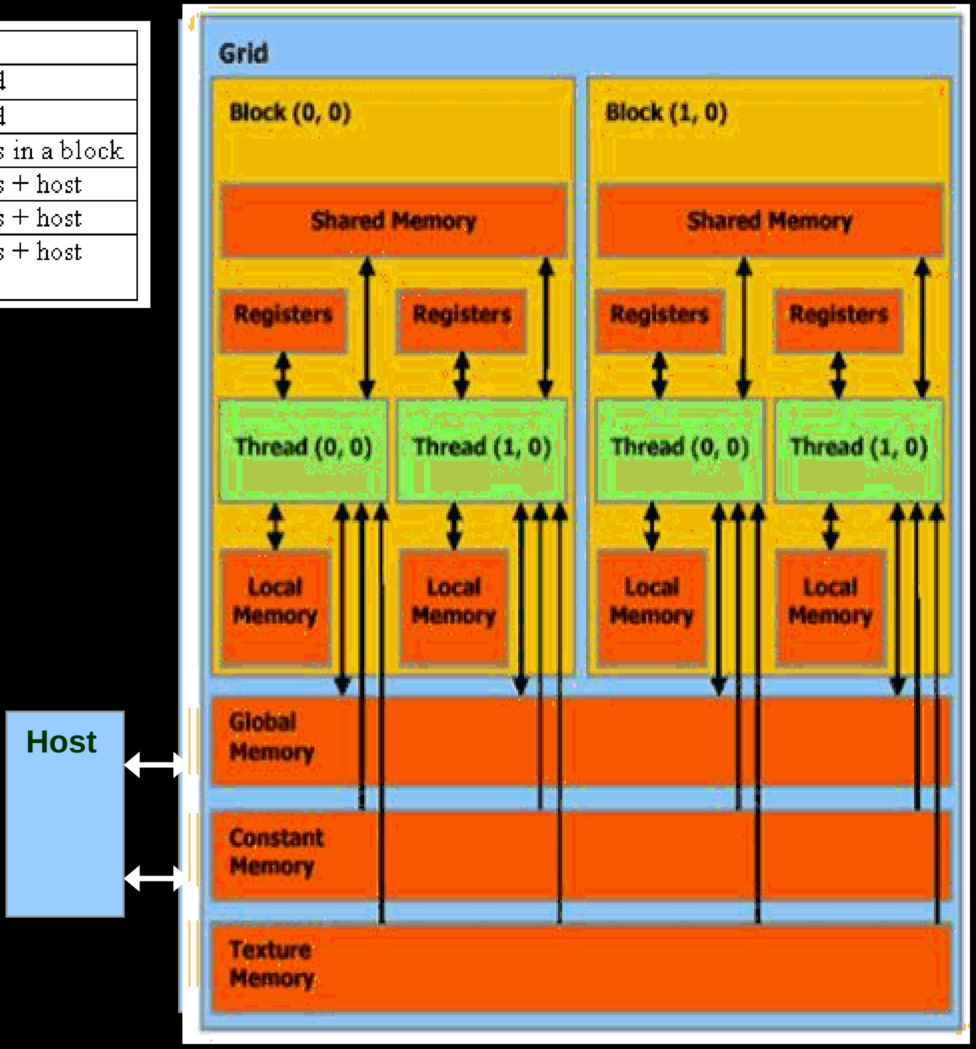
Hardware Implementation of CUDA Memories

- Each thread can:
  - Read/write per-thread registers
  - Read/write per-thread local memory
  - Read/write per-block shared memory
  - Read/write per-grid global memory
  - Read/only per-grid constant memory



More about 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

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 will 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

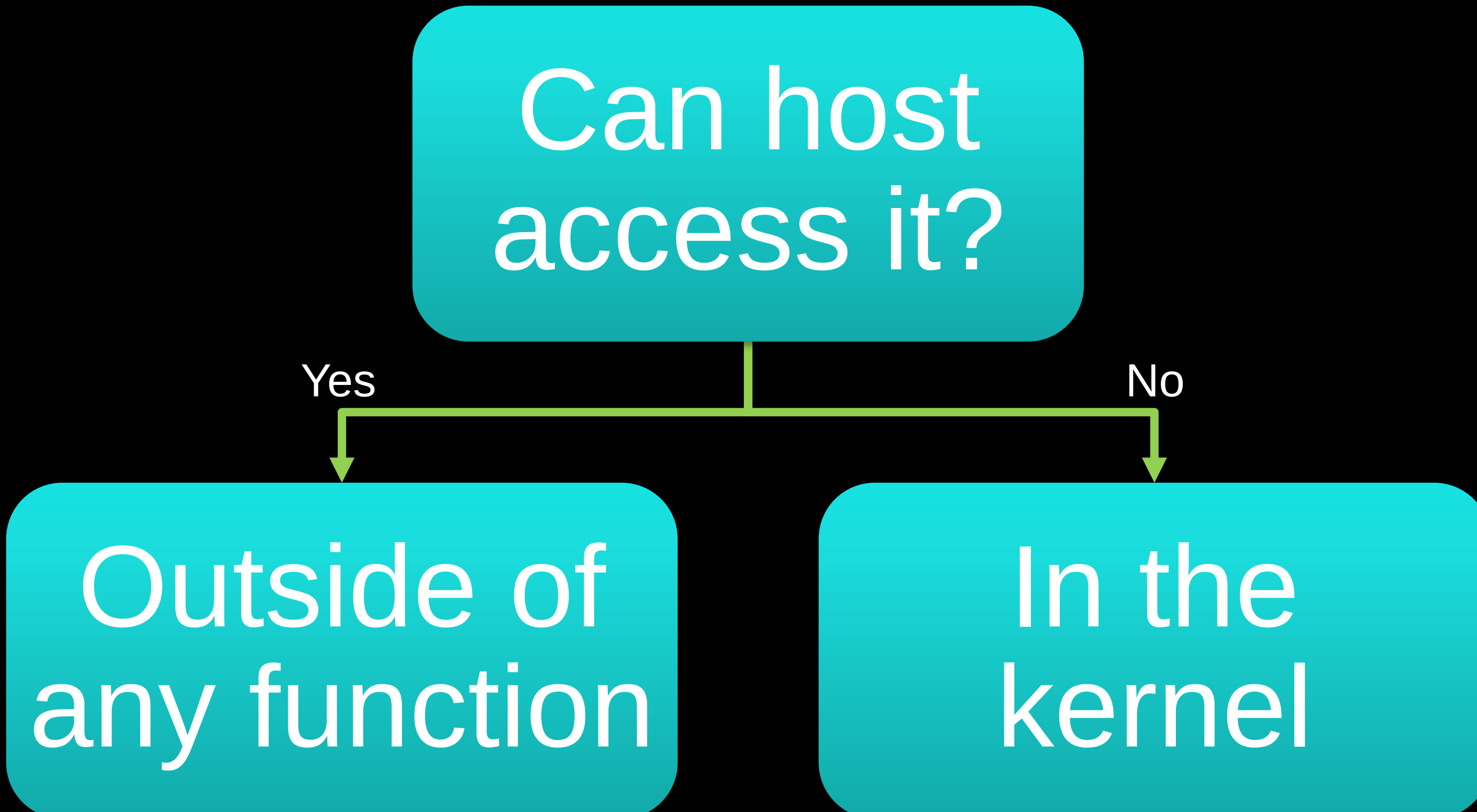
- **scalar variables reside in fast, on-chip registers**
- **shared variables reside in fast, on-chip memories**
- **thread-local arrays & global variables reside in uncached off-chip memory**
- **constant variables reside in cached off-chip memory**

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;
```

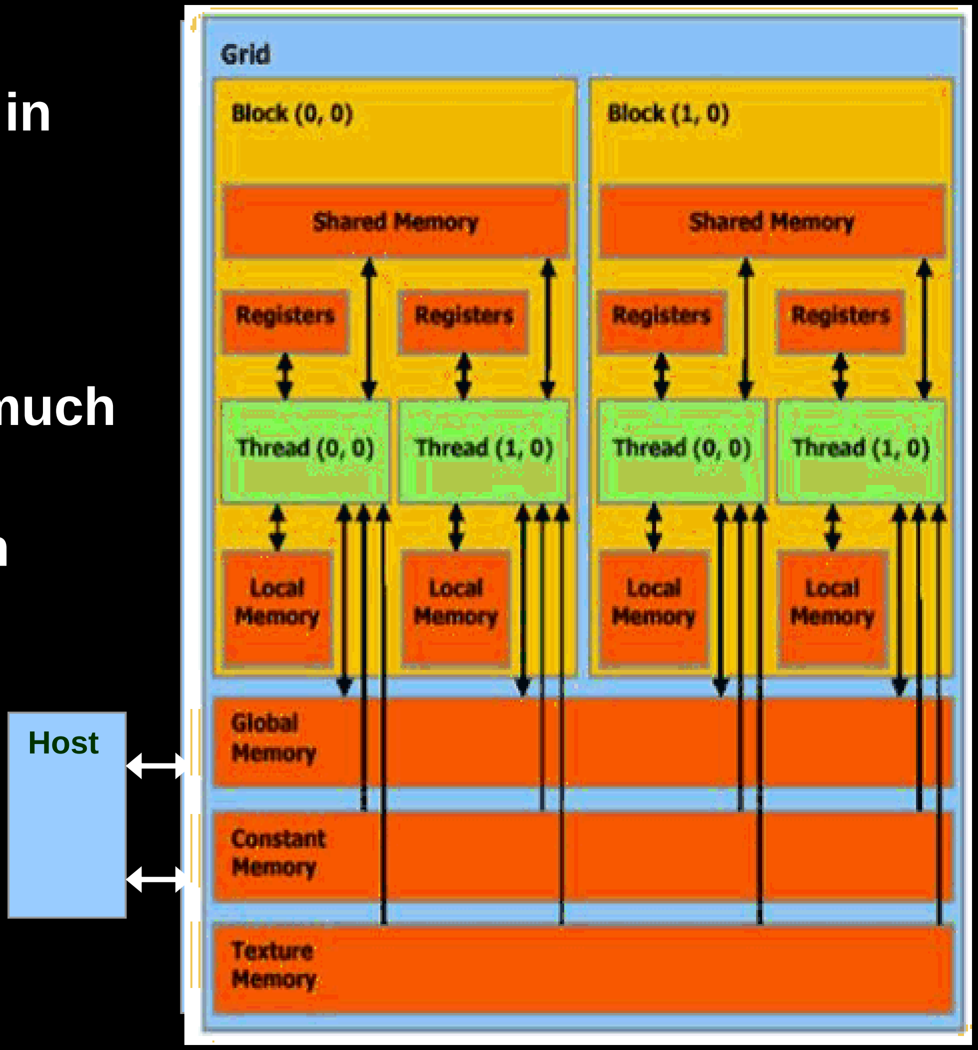
Example – thread-local variables

```
// 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];

  // per-thread heap goes in off-chip memory
  float2 heap[10];

  // read through num_qs points, maintaining
  // the nearest 10 qs to p in the heap
  ...
  // write out the contents of heap to result
  ...
}
```

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 – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // each thread loads two elements from global memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // what are the bandwidth requirements of this kernel?
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i - x_i_minus_one;
  }
}
```

Two loads

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread's global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // How many times does this kernel load input[i]?
    int x_i = input[i]; // once by thread i
    int x_i_minus_one = input[i-1]; // again by thread i+1

    result[i] = x_i – x_i_minus_one;
  }
}
```

Example – shared variables

```
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] – input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
  // compute this thread’s global index
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  if(i > 0)
  {
    // Idea: eliminate redundancy by sharing data
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

    result[i] = x_i - x_i_minus_one;
  }
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  // shorthand for threadIdx.x
  int tx = threadIdx.x;
  // allocate a __shared__ array, one element per thread
  __shared__ int s_data[BLOCK_SIZE];
  // each thread reads one element to s_data
  unsigned int i = blockDim.x * blockIdx.x + tx;
  s_data[tx] = input[i];

  // avoid race condition: ensure all loads
  // complete before continuing
  __syncthreads();
  ...
}
```

Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
  ...
  if(tx > 0)
    result[i] = s_data[tx] - s_data[tx-1];
  else if(i > 0)
  {
    // handle thread block boundary
    result[i] = s_data[tx] - input[i-1];
  }
}
```

Example – shared variables

```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
  extern __shared__ int s_data[];
  ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block_size * sizeof(int)>>>(r,i);
```

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	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

Texture Memory

- actually a part of global memory
- read-only, cached
- connected with extra hardware (separate from thread processors) for texture manipulation, e.g. interpolation, filtering
- no coalescing requirements for fast access
- texture units can bind parts of the global memory – global declaration:

  texture <Type, Dim, ReadMode> texRef;

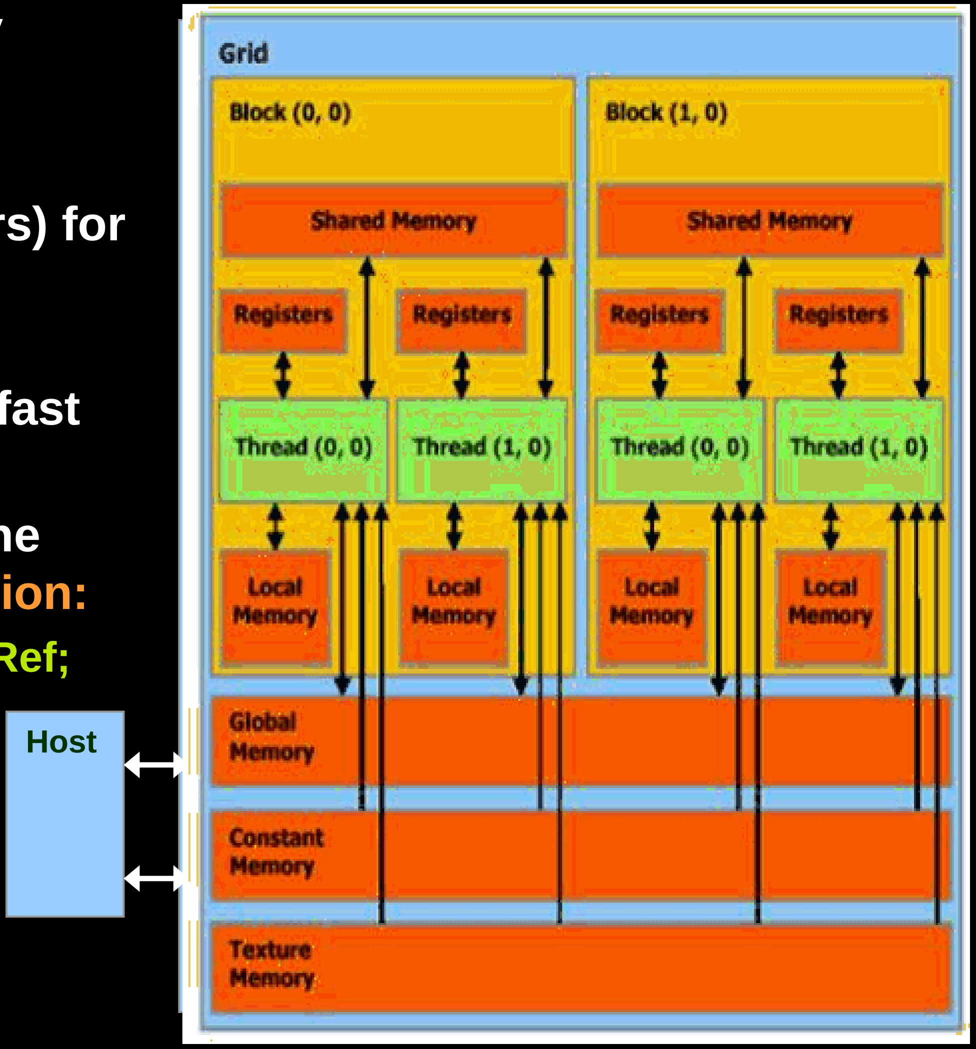
  Type: int, float, vector type

  Dim: 1/2/3D [optional] (default: 1)

  ReadMode:

  - cudaReadModeElementType

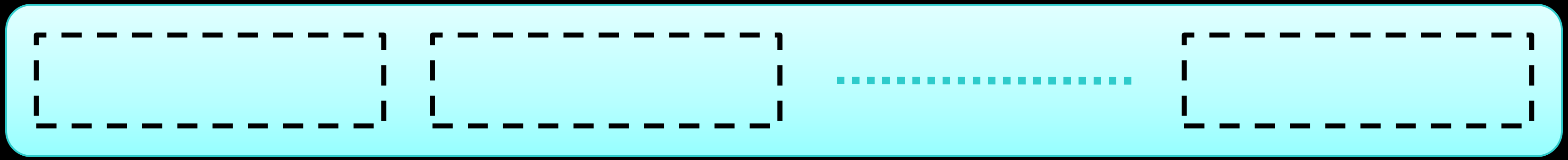
  - cudaReadModeNormalizedFloat



A Common Programming Strategy

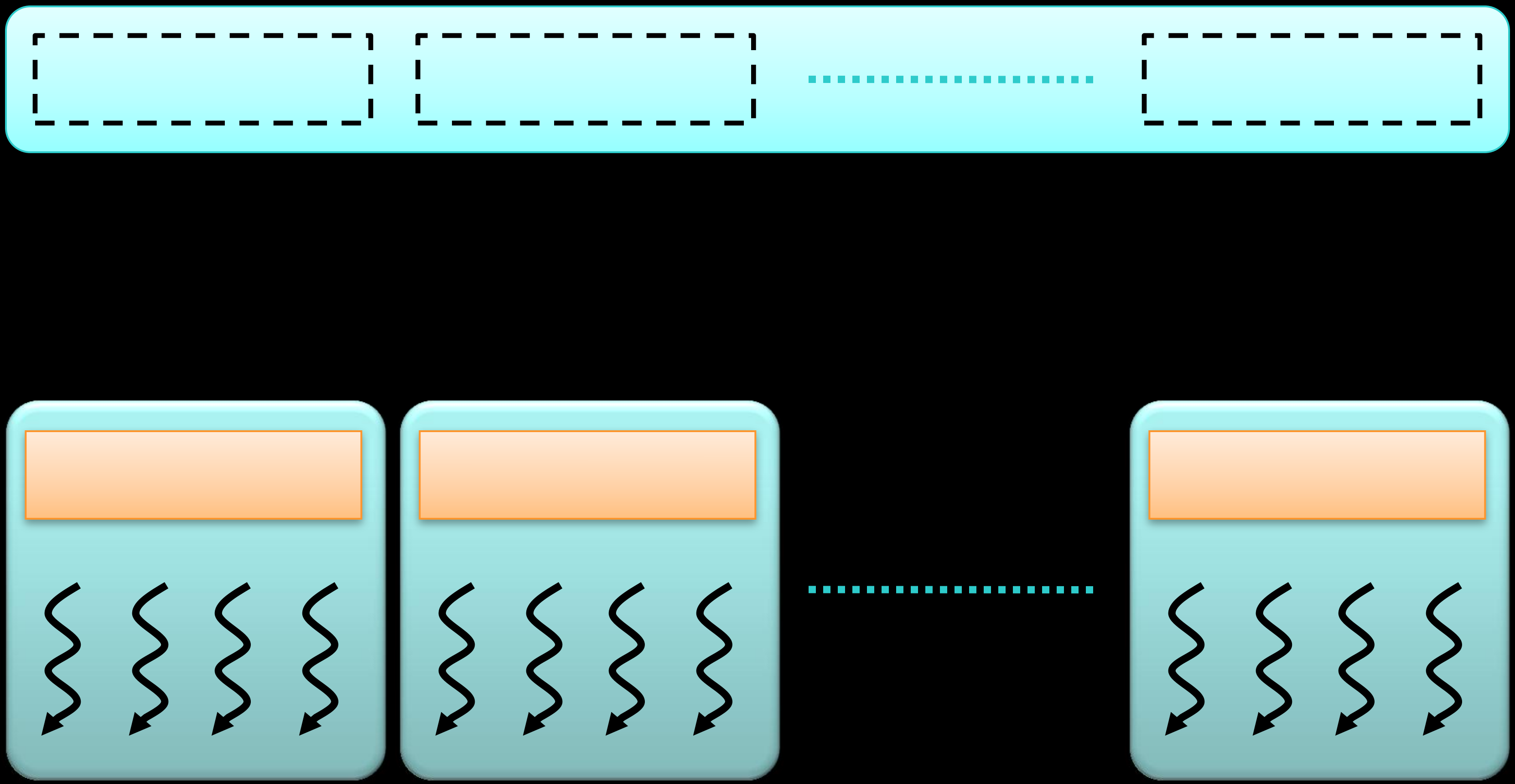
- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from `adjacent_difference` example
  - Divide and conquer

A Common Programming Strategy



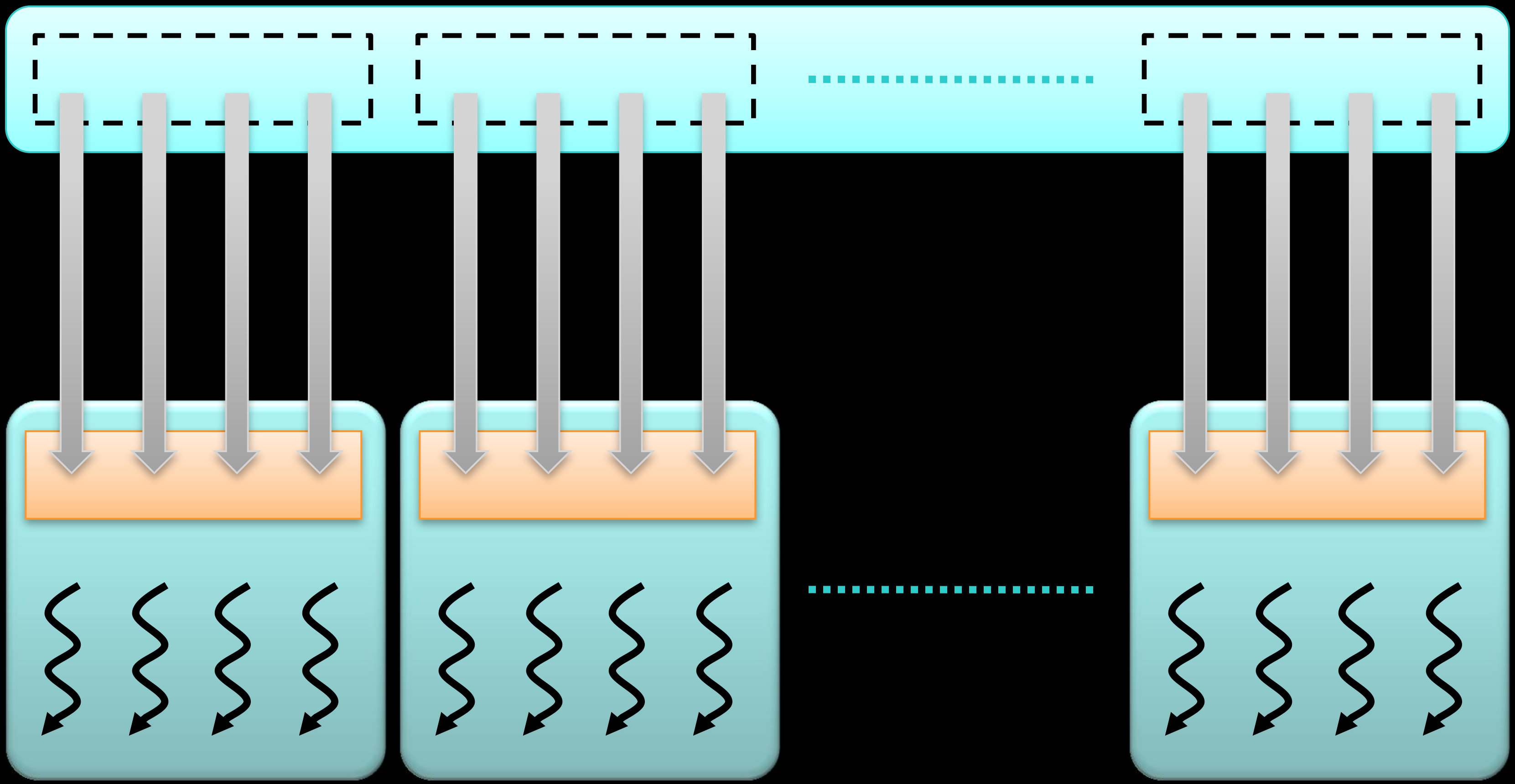
- **Partition data into subsets that fit into shared memory**

A Common Programming Strategy



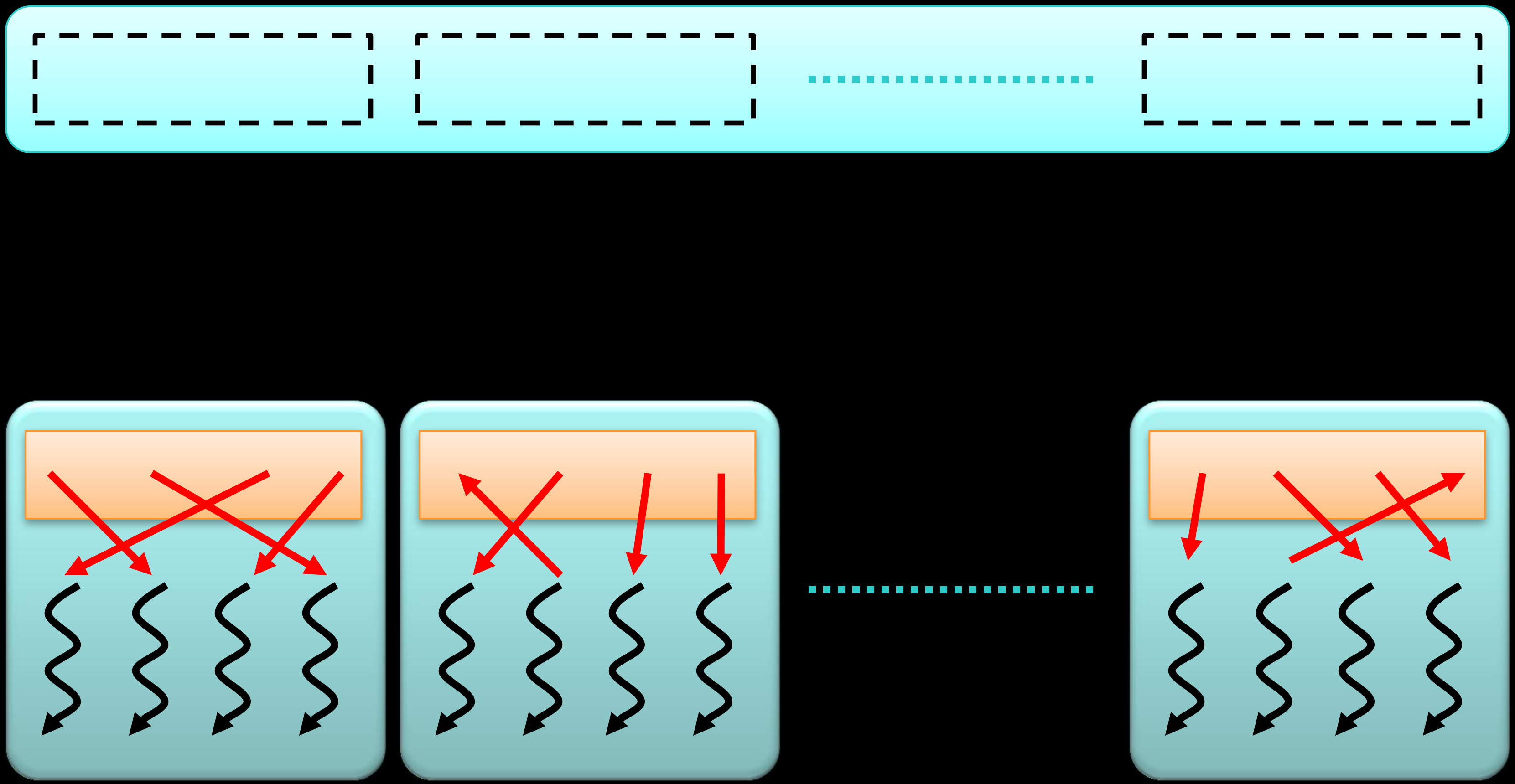
- Handle each data subset with one **thread block**

A Common Programming Strategy



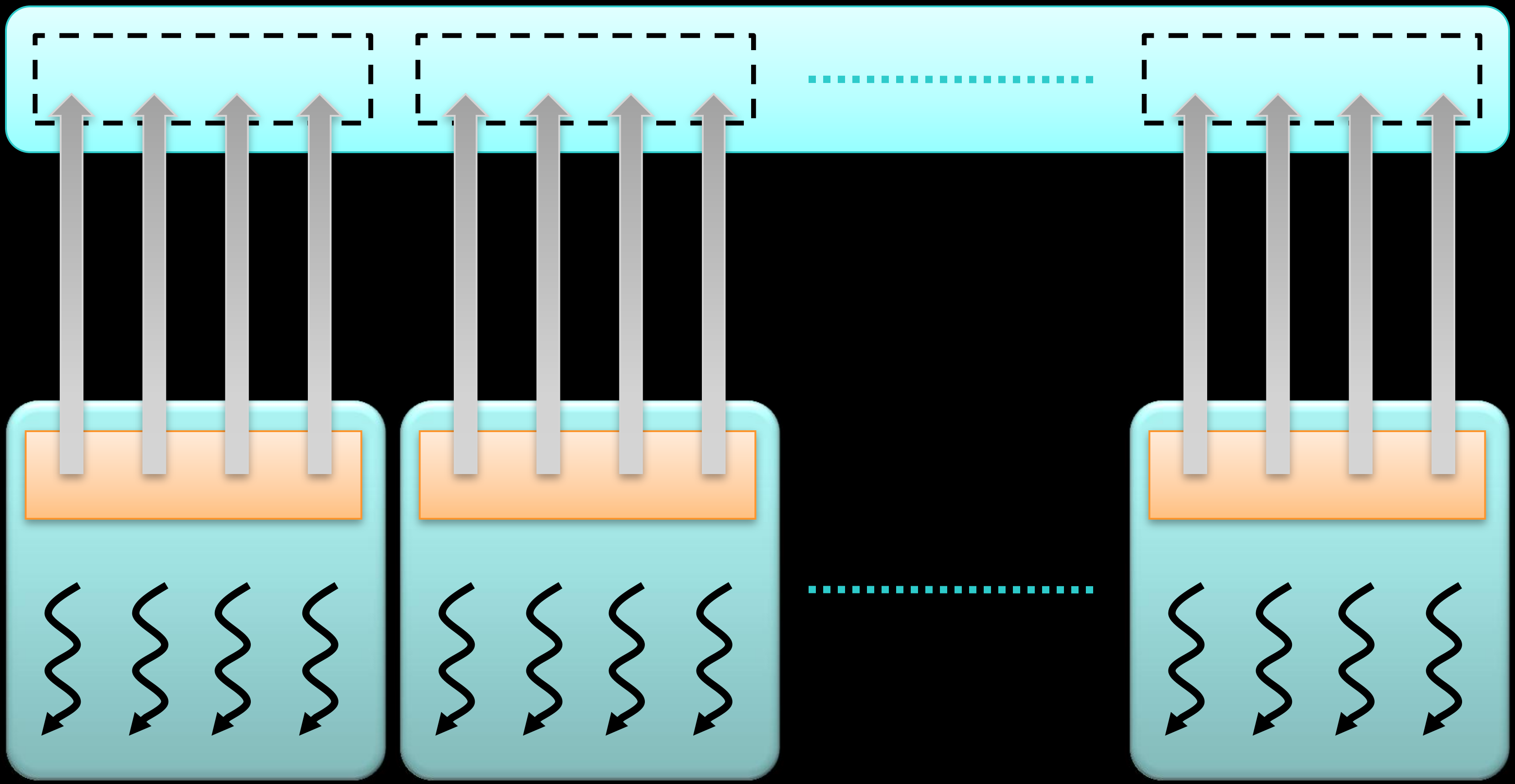
- **Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism**

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

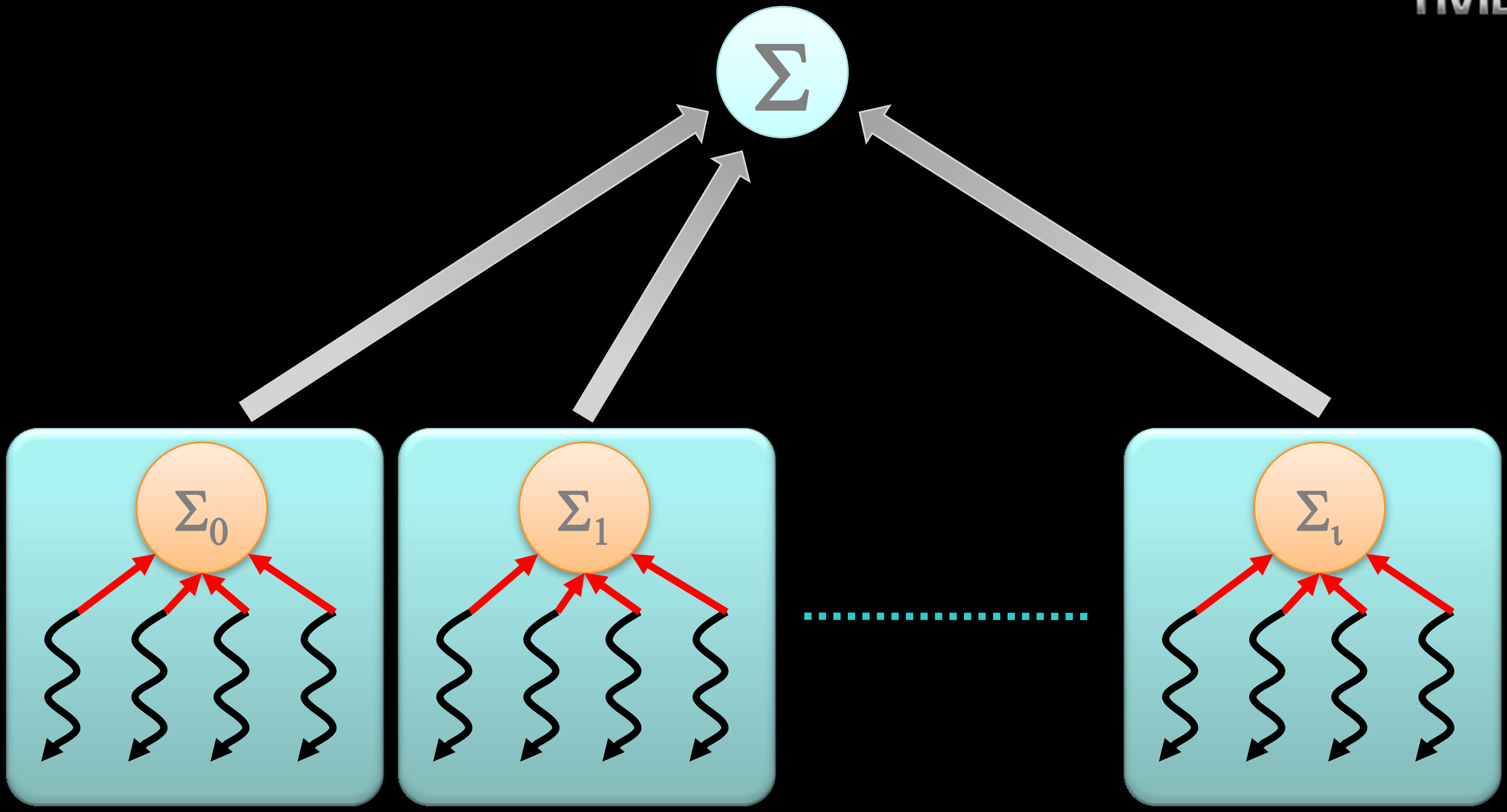
```
// assume *result is initialized to 0
__global__ void sum(int *input, int *result)
{
  atomicAdd(result, input[threadIdx.x]);

  // 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
- Tutorials
  - `thread_local_variables.cu`
  - `shared_variables.cu`
  - `matrix_multiplication.cu`

Cuda built-in vector types

- char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4,

  short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4,

  float1, float2, float3, float4, double2

- dim3 (based on uint3)

- default constructors

```
float a,b,c,d;
float4 f4 = make_float4 (a,b,c,d);
// f4.x=a; f4.y=b; f4.z=c; f4.w=d;
```