CUDA Memories

CME343 / ME339 | 20 May 2011

David Tarjan [dtarjan@nvidia.com]

NVIDIA Research

Recap

- CUDA : Heterogeneous Parallel Computing
- Utilize the power of the CPU and GPU
- Split a problem into serial sections and parallel sections
- Serial sections get executed on the CPU as host code
- Parallel sections get executed on the GPU by launching a kernel

Stencil Kernel Prologue

```
// this function will be our running example today
void stencil(int n, int radius, float* w, float* x, float* y)
{
  for(int i = 0; i < n; i++)
  {
        float sum = 0.f;
        if (radius < i && i < n - radius)
        {
                for(int j = -radius; j < radius; j++)
                        sum += w[j+radius]*x[i+j];
        }
        y[i] = sum;
   }
}
```

Recap: Stencil Kernel [1/3]

```
__global__ void stencil(int n, int radius, float* w, float* x,
  float* y) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  float sum = 0.f;
  if (radius < i && i < n - radius) {
    for(int j = -radius; j < radius; j++)
        sum += w[j+radius]*x[i+j];
  }
  y[i] = sum;
}
int main() {
  ...
  int nblocks = (n + 255)/256;
  stencil<<<nblocks, 256>>>(n, radius, d_w, d_x, d_y);
  ...
}
```

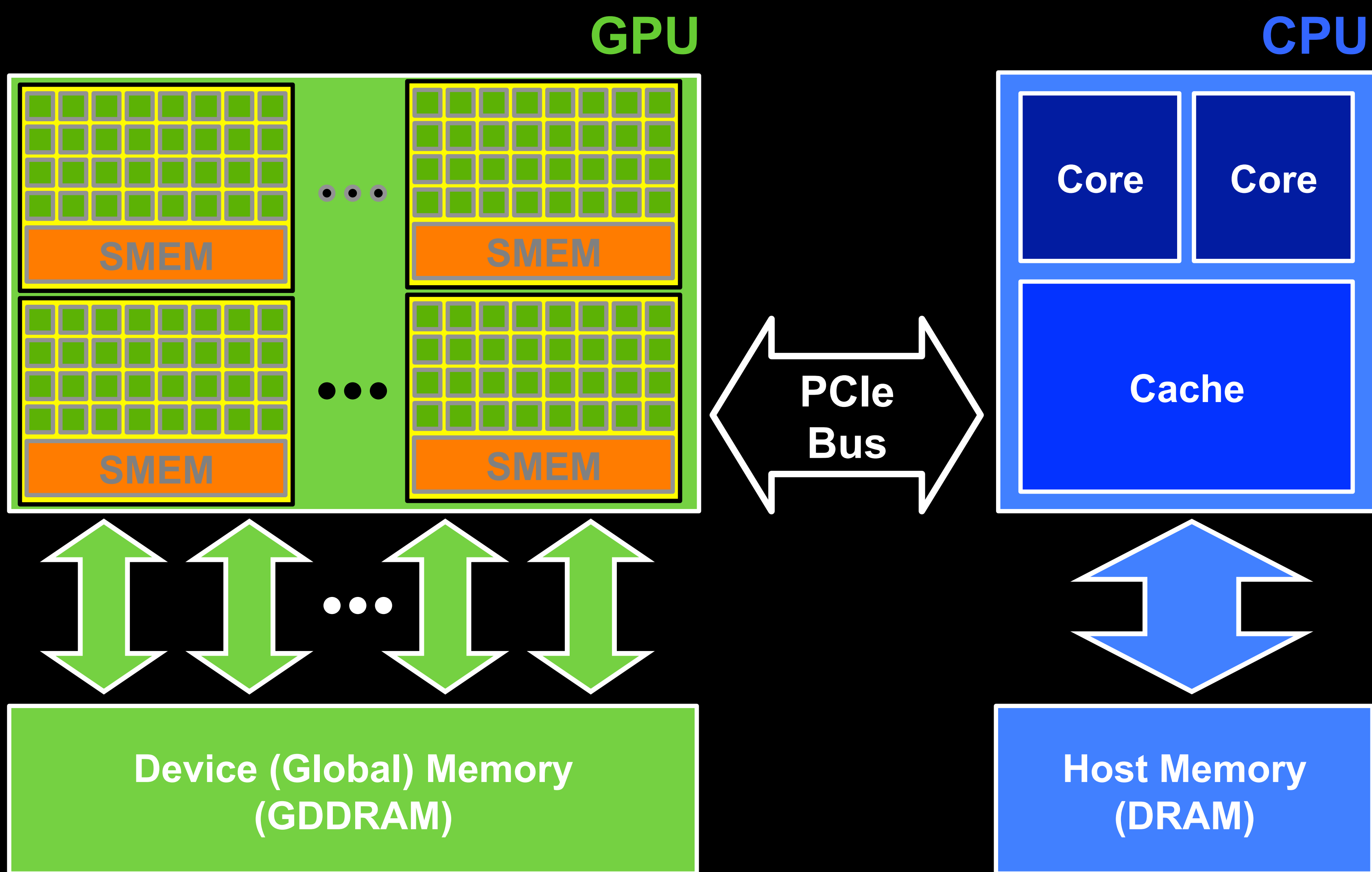
Recap: Stencil Kernel [2/3]

```
int main()
{
   // allocate and initialize host (CPU) memory
   …
   // allocate device (GPU) memory
   float *d_w, *d_x, *d_y;
   cudaMalloc((void**) &d_x, n * sizeof(float));
   cudaMalloc((void**) &d_y, n * sizeof(float));
   cudaMalloc((void**) &d_w, 2*radius * sizeof(float));

   // copy x and w from host memory to device memory
   cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
   cudaMemcpy(d_w, w, 2*radius*sizeof(float),
   cudaMemcpyHostToDevice);

  // invoke parallel stencil kernel with 256 threads / block
   int nblocks = (n + 255)/256;
   stencil<<<nblocks, 256>>>(n, radius, d_w, d_x, d_y);
```

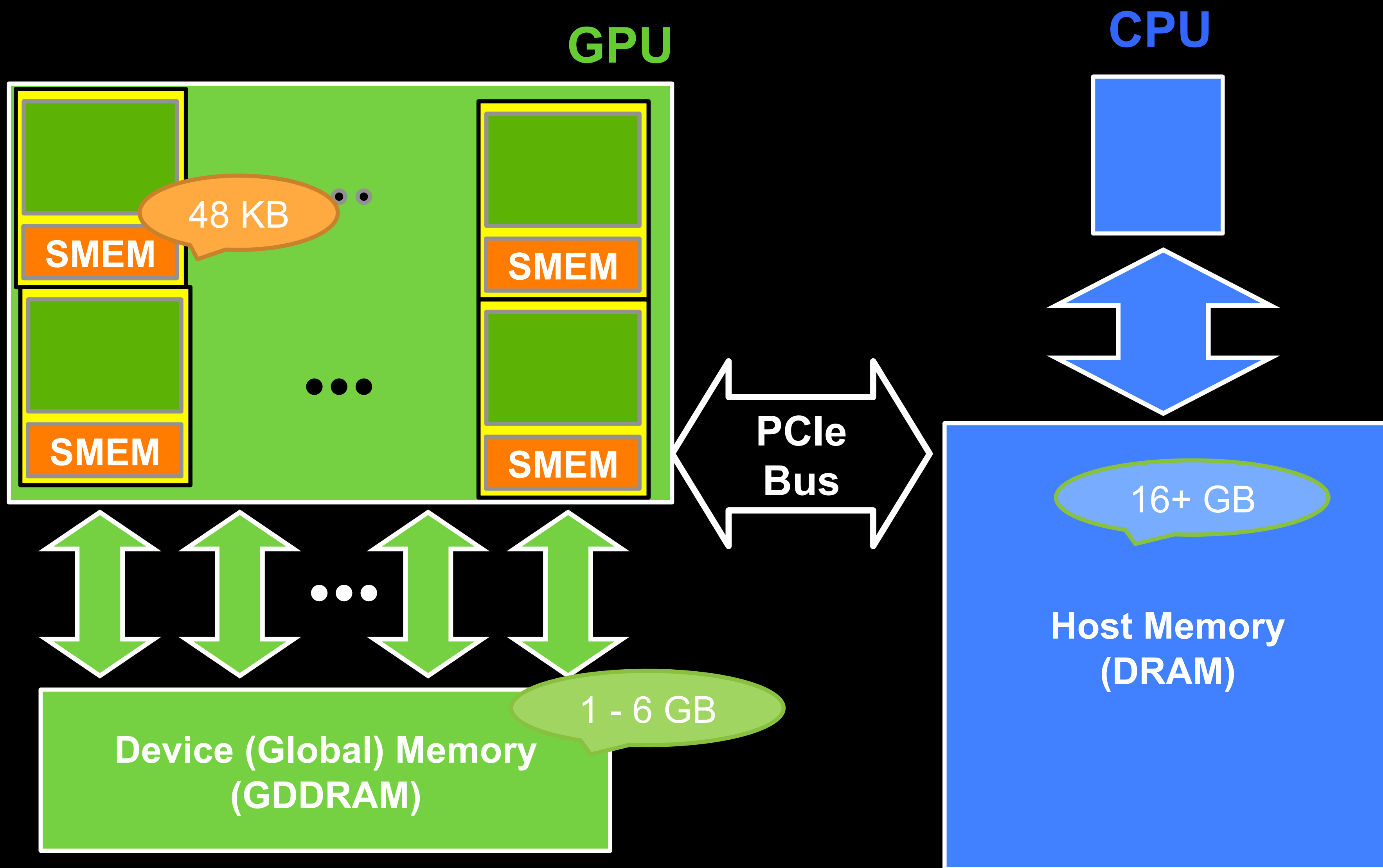
Recap: Stencil Kernel [3/3]

```
  // invoke parallel stencil kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  stencil<<<nblocks, 256>>>(n, radius, d_w, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(y, d_y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result…


  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_w);

 return 0;
}
```

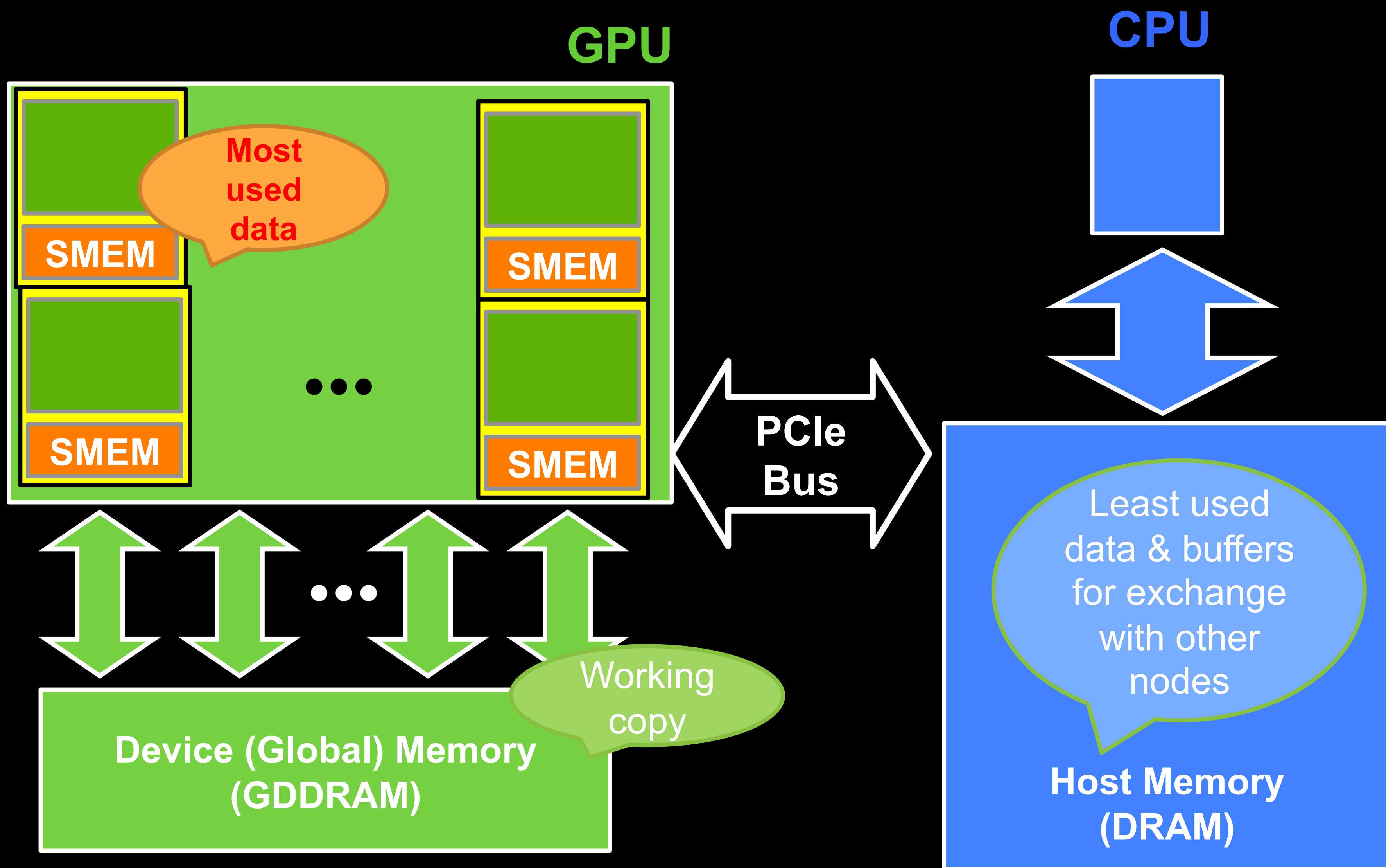
System Organisation



Memory Capacities



Where to keep what data?



Minimize CPU<->GPU Data Transfers

- ~6GB/sec between CPU and GPU
- ~160GB/sec to GPU memory
- Transfers back and forth between CPU and GPU quickly become prohibitively expensive!
- Do as much as possible on the GPU

Example of Keeping Data on GPU

- **Iterative solver with boundary conditions**

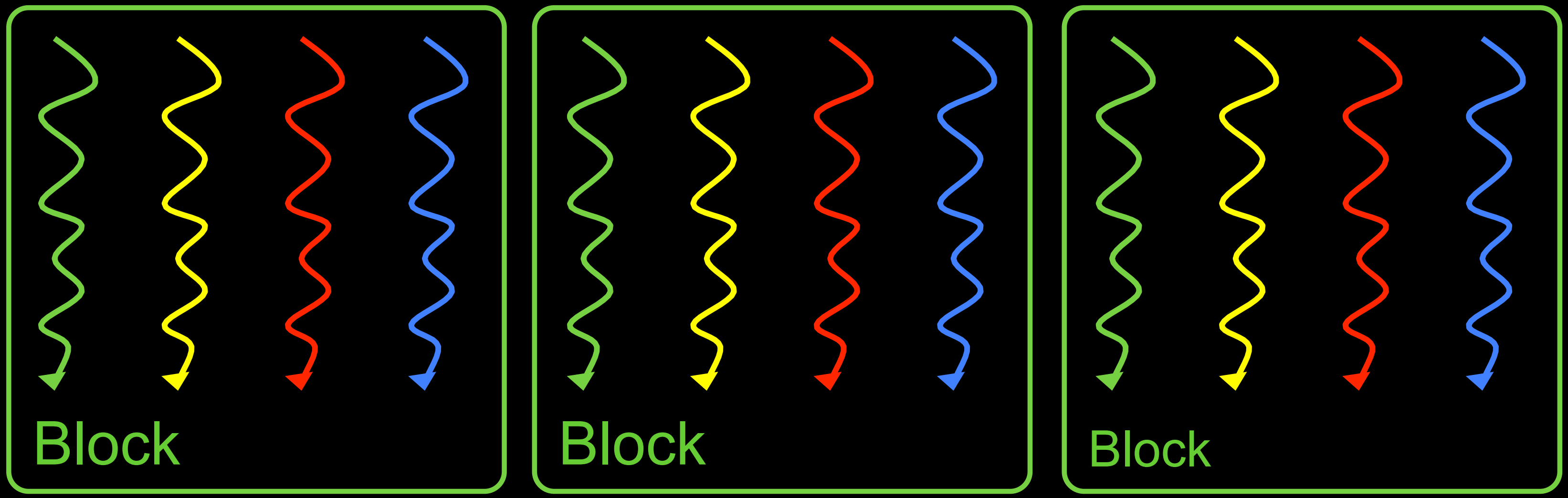
```
int main()
{
   ...
  // invoke relaxation step
   relax_step<<<nb_relax, bs_relax>>>(domain_size, d_weights,
   d_copy_A, d_copy_B);
  // implicit synchronization between all threads happens at the
  end of each kernel
  // now enforce boundary conditions
   enforce_boundary<<<nb_bound, bs_bound>>>(domain_size,
   d_copy_B);
  // invoke relaxation step again, but now input and output have
  been switched
   relax_step<<<nb_relax, bs_relax>>>(domain_size, d_weights,
   d_copy_B, d_copy_A);
```

Example of Keeping Data on GPU (2)

- **Input is too big for GPU memory**

```
int main()
{
   // input is too big for GPU memory or streaming
   float* big_x = ...;
   // allocate device (GPU) memory
   float *d_x, *d_partial_results;
   cudaMalloc((void**) &d_x, small_num * sizeof(float));
   cudaMalloc((void**) &d_partial_results, small_num * sizeof
   (float));
   for(int i=0; i < big_num; i+= small_num)
   {
     // copy x and y from host memory to device memory
     cudaMemcpy(d_x, &big_x[i], small_num*sizeof(float),
   cudaMemcpyHostToDevice);
     // invoke kernel that accumulates results
     some_kernel<<<nb, bs>>>(n, d_x, d_partial_results);
   }
```

Recall CUDA Thread Hierarchy



- Threads are grouped into **blocks**
- **Blocks** have fast communication through shared memory (variables prefixed with __shared__)
- Blocks have very fast synchronization with **__syncthreads()**

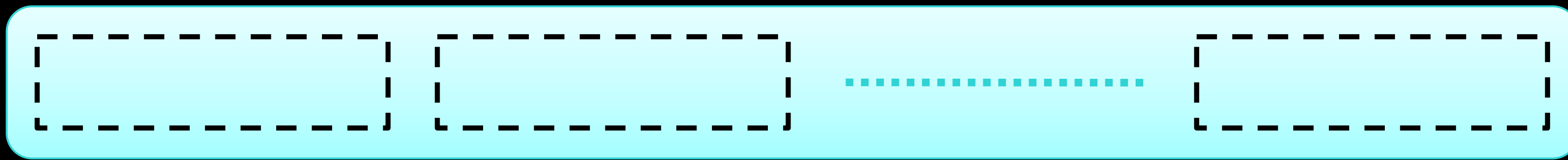
Block Synchronization

- A call to __syncthreads() ensures:
  - Every thread in the threadblock has arrived at this point in the program
  - All loads have completed
  - All stores have completed
- Can hang your program if used within an if,switch or loop statement
- Unless you can guarantee that all threads in the threadblock will reach this point in the program

A Common Programming Strategy

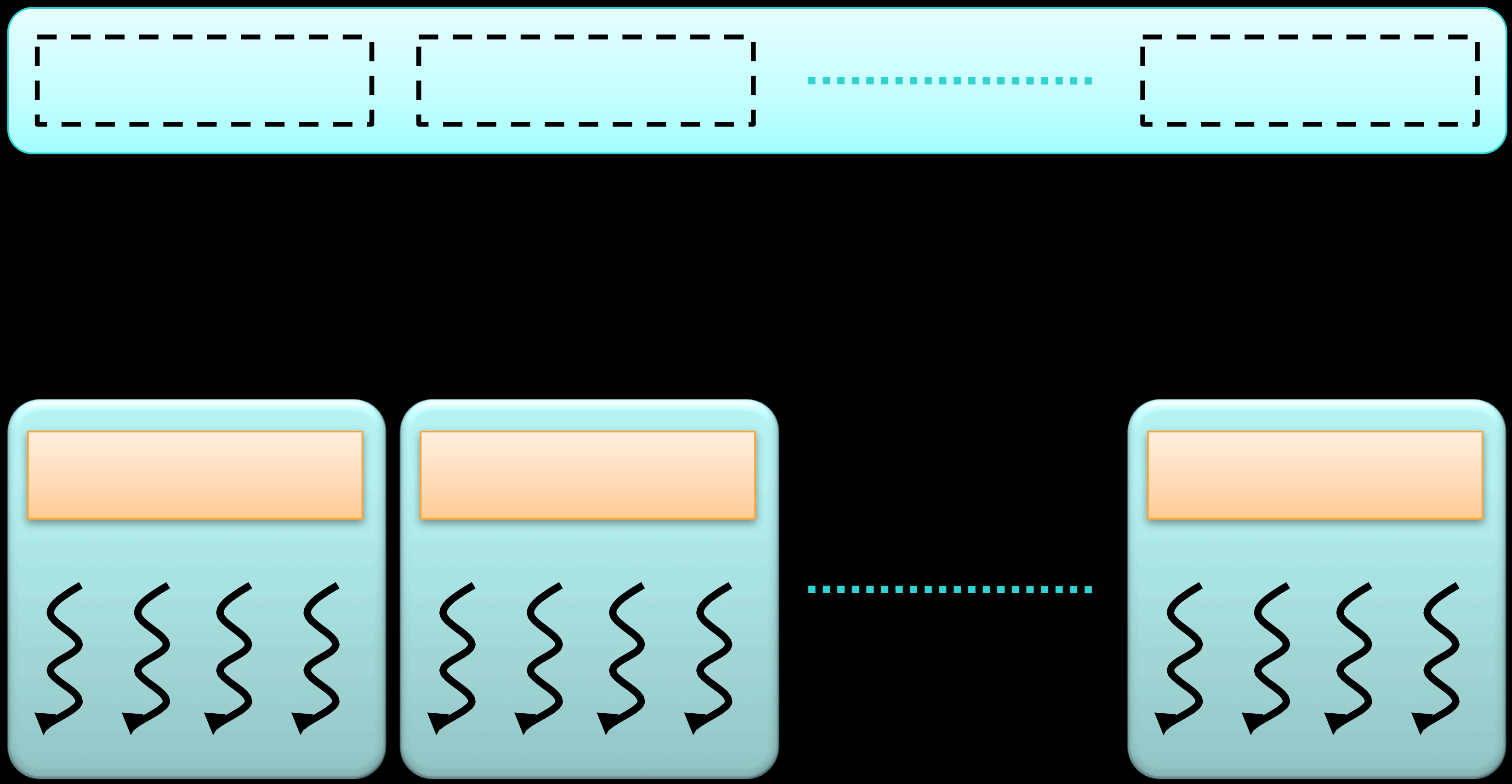
- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from `stencil` example
  - Divide and conquer

A Common Programming Strategy



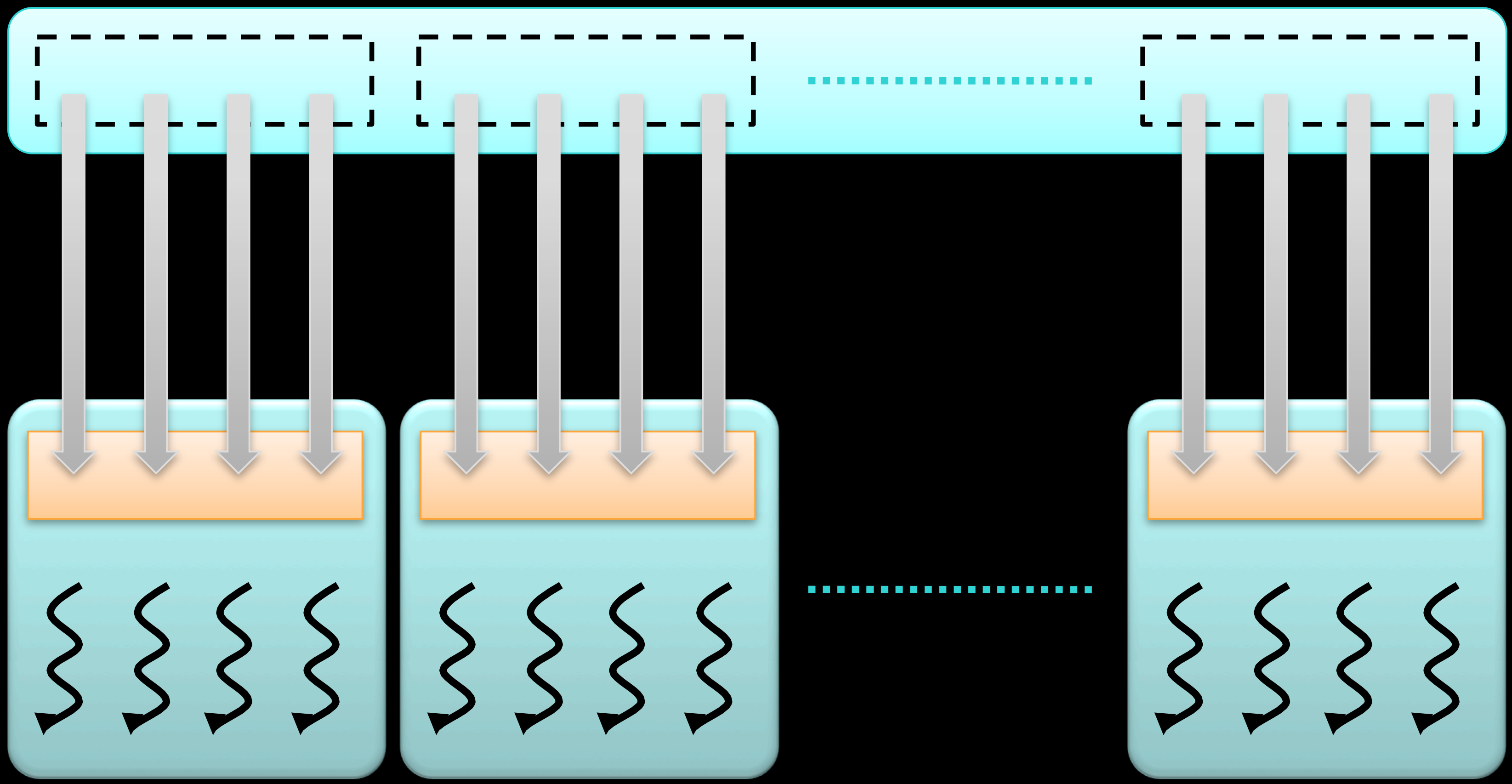
- **Partition data into subsets that fit into shared memory**

A Common Programming Strategy



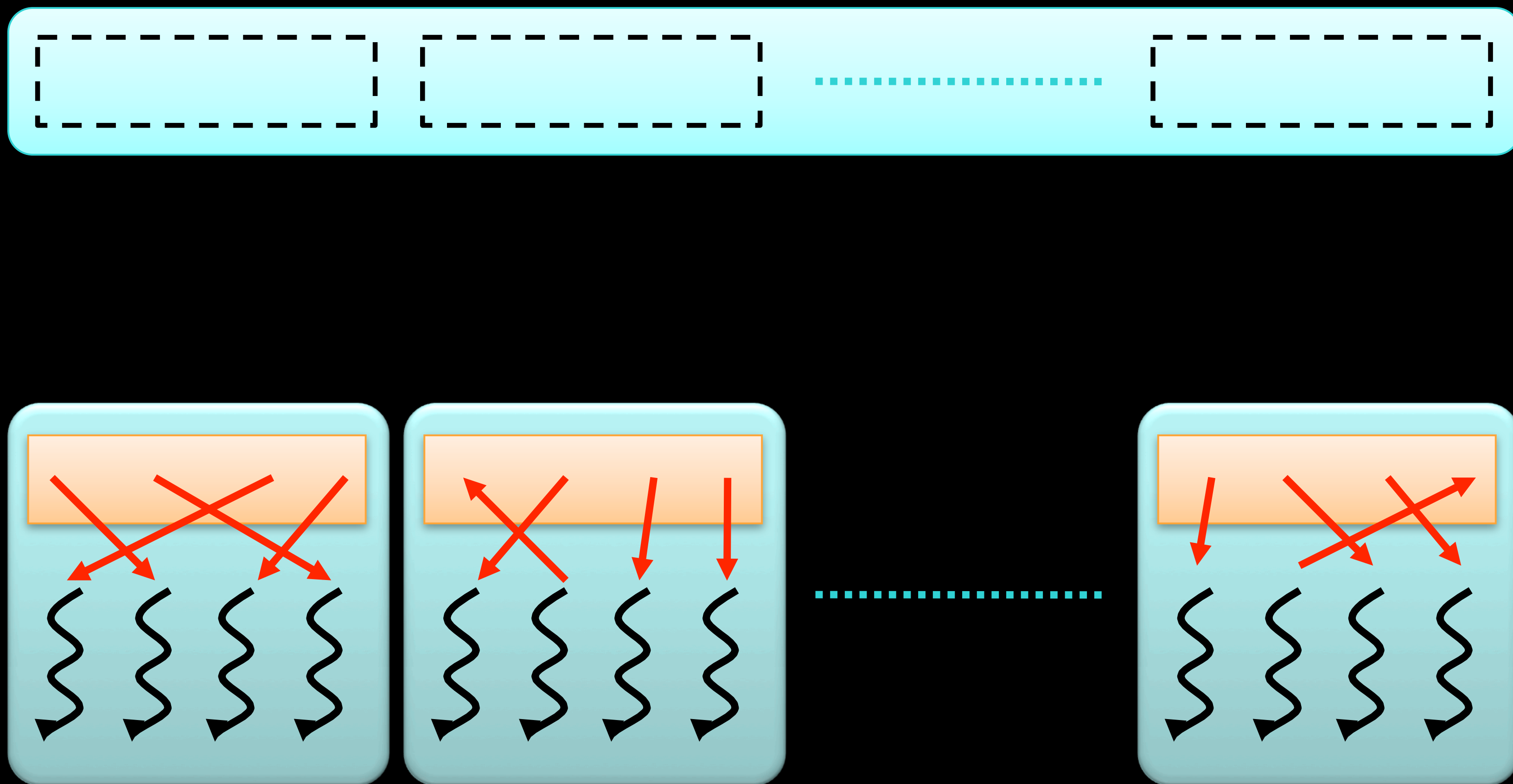
- Handle each data subset with one **thread block**

A Common Programming Strategy



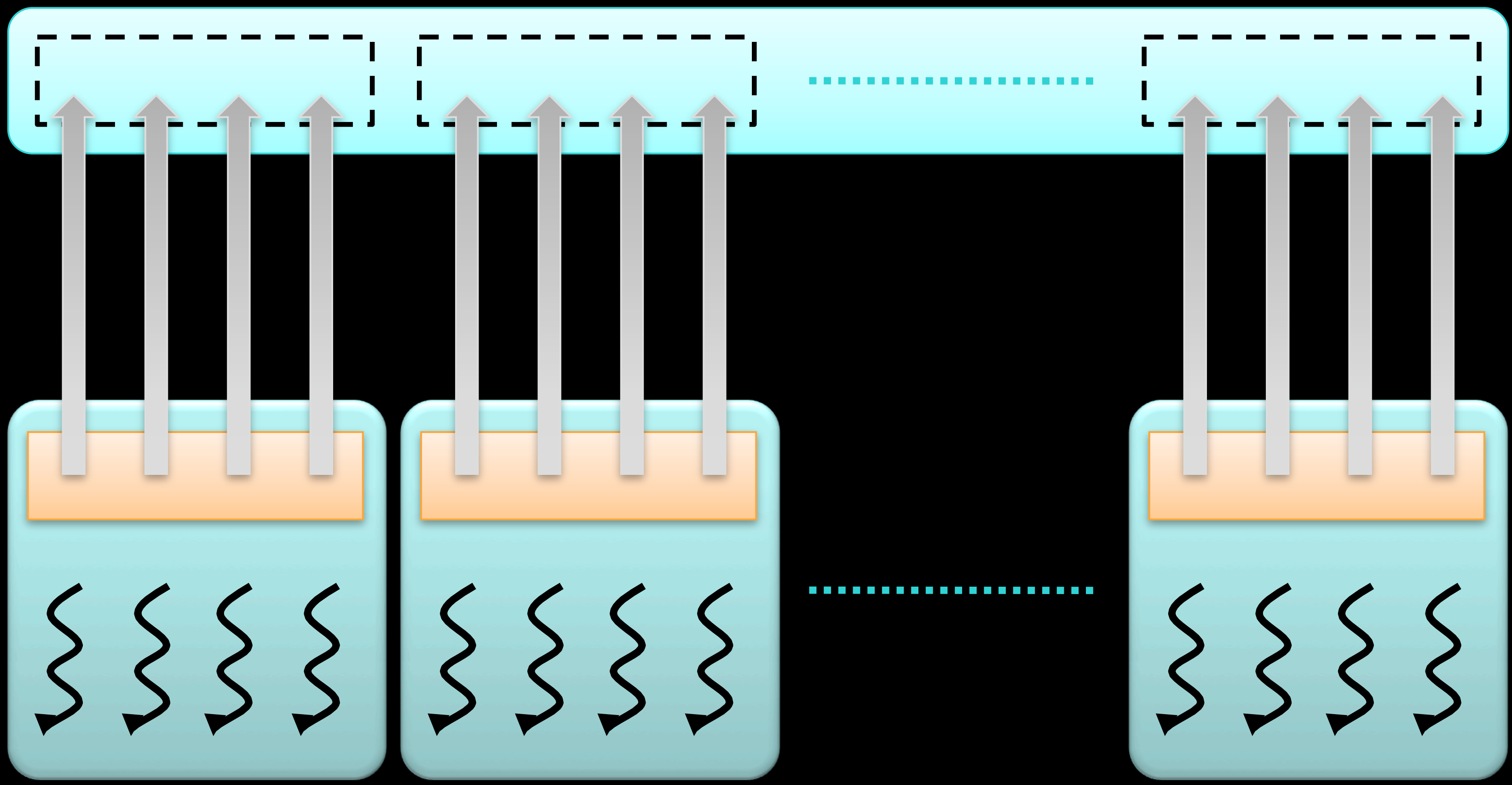
- **Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism**

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

Example – shared variables

```
// 1D stencil example
// compute result[i] = sum(input[i+j]*weight[j] for j in …)
__global__ void stencil(int n, int radius, float* w, float* x,
   float* y)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   float sum = 0.f;
   if (radius < i && i < n - radius)
  {
    for(int j = -radius; j < radius; j++)
        sum += w[j+radius]*x[i+j];
  }
  y[i] = sum;
}
```

Example – shared variables

```
// 1D stencil example
// compute result[i] = sum(input[i+j]*weight[j] for j in …)
__global__ void stencil(int n, int radius, float* w, float* x,
   float* y)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   float sum = 0.f;
   if (radius < i && i < n - radius)
  {
    // what are the bandwidth requirements for this loop?
    for(int j = -radius; j < radius; j++)
       sum += w[j+radius]*x[i+j];
  }
  y[i] = sum;
}
```

4*radius loads

Example – shared variables

```
// 1D stencil example
// compute result[i] = sum(input[i+j]*weight[j] for j in …)
__global__ void stencil(int n, int radius, float* w, float* x,
   float* y)
{
   int i = threadIdx.x + blockDim.x * blockIdx.x;
   float sum = 0.f;
   if (radius < i && i < n - radius)
  {
    // Idea: Eliminate redundant loads by sharing data
    for(int j = -radius; j < radius; j++)
        sum += w[j+radius]*x[i+j];
  }
  y[i] = sum;
}
```

Example – shared variables

```
__global__ void stencil(int n, int radius, float* w, float* x,
   float* y)
{
  __shared__ float s_x[BLOCK_DIM + 2*MAX_RADIUS];
  __shared__ float s_w[2*MAX_RADIUS];
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  // copy data cooperatively into shared memory
  if(threadIdx.x < 2*radius)
     s_w[threadIdx.x] = w[threadIdx.x];
  if(radius < i)
     s_x[threadIdx.x] = x[i - radius];
  if(i < n – radius && threadIdx.x + 2*radius > blockDim.x-1)
     s_x[threadIdx.x + 2*radius] = x[i + radius];
  // avoid race condition: ensure all loads
  // complete before continuing
  __syncthreads();
```

Example – shared variables

```
  float sum = 0.f;
  if (radius < i && i < n - radius)
  {
    // all accesses now go to shared memory
    // note change in indexing
    for(int j = 0; j < 2*radius; j++)
        sum += s_w[j]*s_x[threadIdx.x+j];
  }
  // always write back to global memory
  // shared memory only live for the lifetime of a threadblock
  y[i] = sum;
}
```

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();
  ...
}
```

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 (explained next time) to enforce well-defined semantics

Communication Through Memory

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

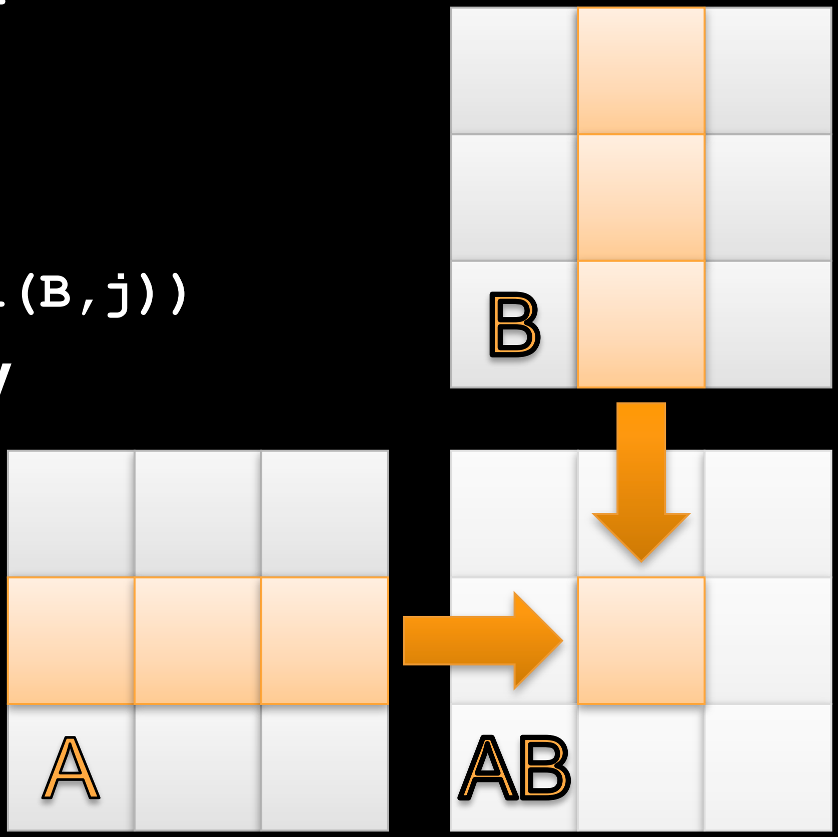
```
__global__ void share_data(int *input)
{
  __shared__ int data[BLOCK_SIZE];
  data[threadIdx.x] = input[threadIdx.x];
  __syncthreads();
  // the state of the entire data array
  // is now well-defined for all threads
  // in this block
}
```

Advice

- Use barriers such as `__syncthreads` to wait until `__shared__` data is ready
- Don't synchronize or serialize unnecessarily

Matrix Multiplication Example

- Generalize `adjacent_difference` example
- AB = A * B
  - Each element AB_{ij}
  - `= dot(row(A,i),col(B,j))`
- Parallelization strategy
  - Thread → AB_{ij}
  - 2D kernel



First Implementation

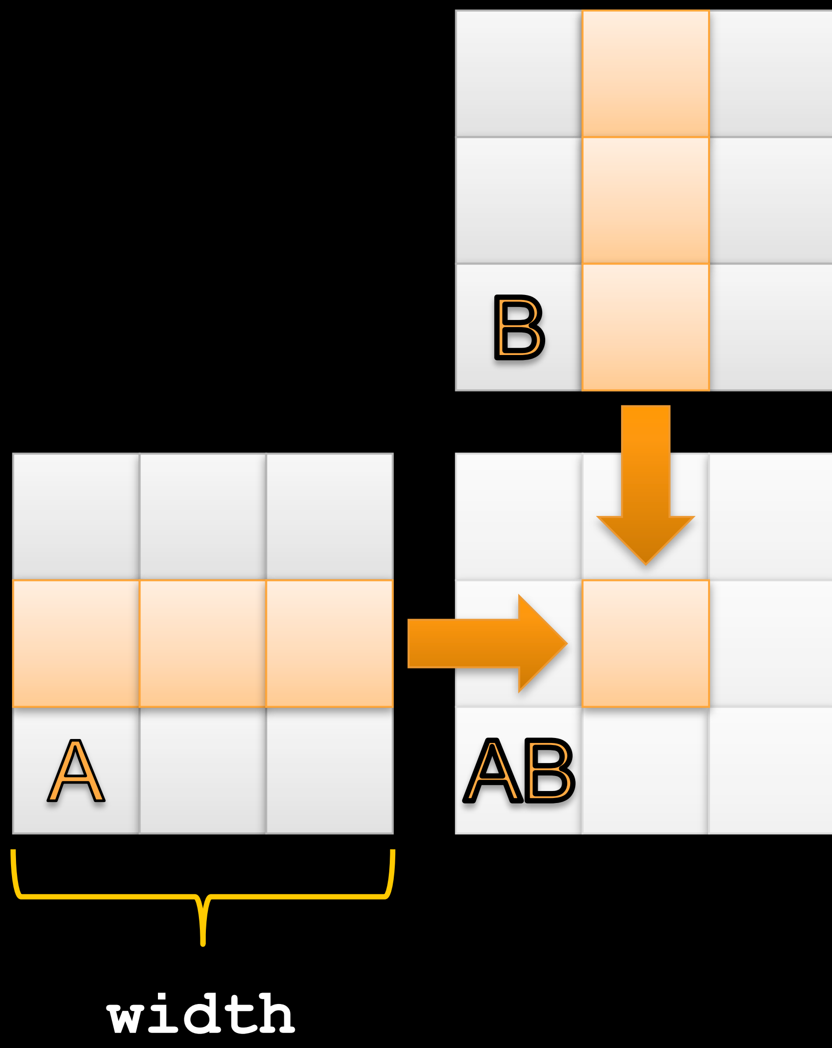
```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
  // calculate the row & col index of the element
  int row = blockIdx.y*blockDim.y + threadIdx.y;
  int col = blockIdx.x*blockDim.x + threadIdx.x;

  float result = 0;

  // do dot product between row of a and col of b
  for(int k = 0; k < width; ++k)
    result += a[row*width+k] * b[k*width+col];

  ab[row*width+col] = result;
}
```

How will this perform?

How many loads per term of dot product?	2 (a & b) = 8 Bytes
How many floating point operations?	2 (multiply & addition)
Global memory access to flop ratio (GMAC)	8 Bytes / 2 ops = 4 B/op
What is the peak fp performance of GeForce GTX 260?	805 GFLOPS
Lower bound on bandwidth required to reach peak fp performance	GMAC * Peak FLOPS = 4 * 805 = 3.2 TB/s
What is the actual memory bandwidth of GeForce GTX 260?	112 GB/s
Then what is an upper bound on performance of our implementation?	Actual BW / GMAC = 112 / 4 = 28 GFLOPS

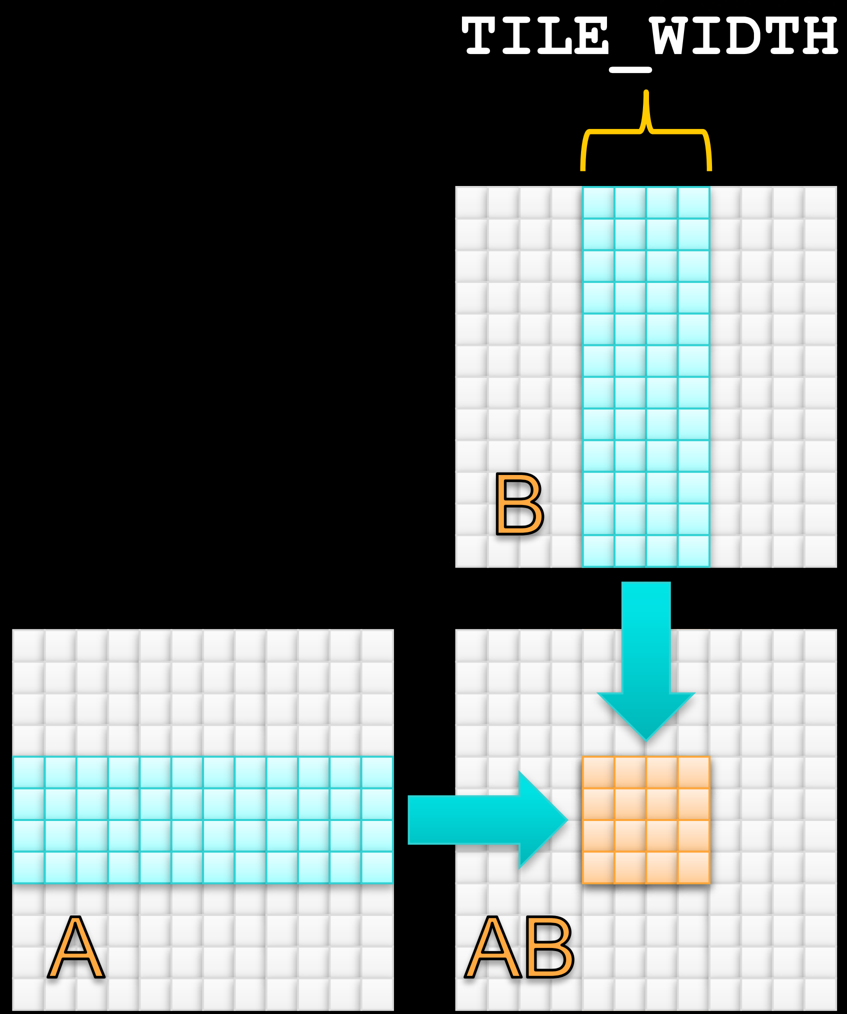
Idea: Use `__shared__` memory to reuse global data

- Each input element is read by `width` threads
- Load each element into `__shared__` memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

- Partition kernel loop into phases
- Load a tile of both matrices into `__shared__` each phase
- Each phase, each thread computes a partial result



Better Implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width)
{
  // shorthand
  int tx = threadIdx.x, ty = threadIdx.y;
  int bx = blockIdx.x,  by = blockIdx.y;
  // allocate tiles in __shared__ memory
  __shared__ float s_a[TILE_WIDTH][TILE_WIDTH];
  __shared__ float s_b[TILE_WIDTH][TILE_WIDTH];
  // calculate the row & col index
  int row = by*blockDim.y + ty;
  int col = bx*blockDim.x + tx;

  float result = 0;
```

Better Implementation

```
  // loop over the tiles of the input in phases
  for(int p = 0; p < width/TILE_WIDTH; ++p)
  {
    // collaboratively load tiles into __shared__
    s_a[ty][tx] = a[row*width + (p*TILE_WIDTH + tx)];
    s_b[ty][tx] = b[(m*TILE_WIDTH + ty)*width + col];
    __syncthreads();

    // dot product between row of s_a and col of s_b
    for(int k = 0; k < TILE_WIDTH; ++k)
      result += s_a[ty][k] * s_b[k][tx];
    __syncthreads();
  }

  ab[row*width+col] = result;
}
```

Use of Barriers in `mat_mul`

- Two barriers per phase:
  - `__syncthreads` after all data is loaded into `__shared__` memory
  - `__syncthreads` after all data is read from `__shared__` memory
  - Note that second `__syncthreads` in phase `p` guards the load in phase `p+1`
- Use barriers to **guard** data
  - Guard against using uninitialized data
  - Guard against bashing live data

First Order Size Considerations

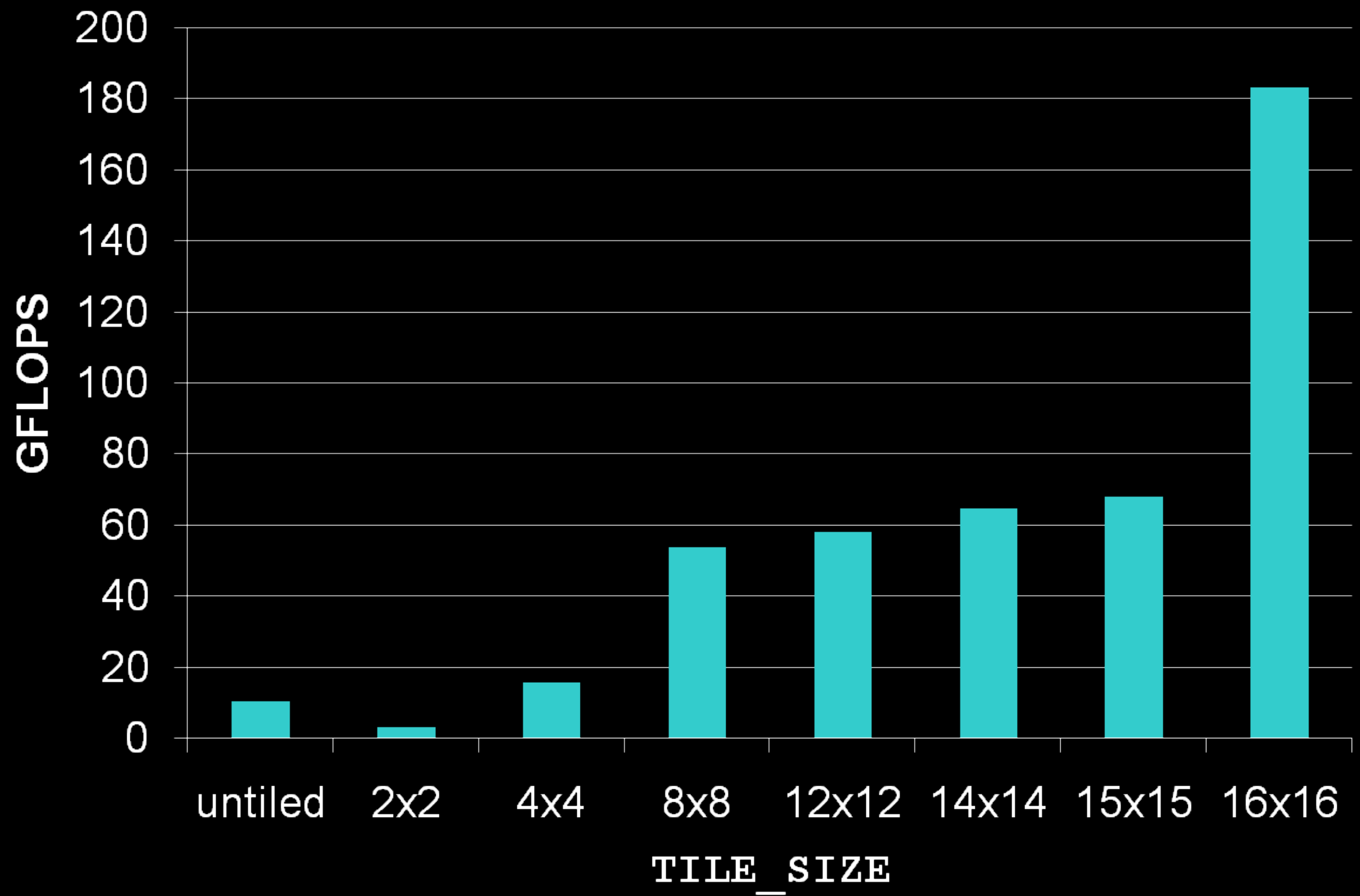
- Each **thread block** should have many threads
  - `TILE_WIDTH` = 16 → 16*16 = 256 threads
- There should be many thread blocks
  - 1024*1024 matrices → 64*64 = 4096 thread blocks
  - `TILE_WIDTH` = 16 → gives each SM 3 blocks, 768 threads
  - Full **occupancy**
- Each thread block performs 2 * 256 = 512 32b loads for 256 * (2 * 16) = 8,192 fp ops
  - Memory bandwidth no longer limiting factor

Optimization Analysis

Implementation	Original	Improved
Global Loads	$2N^3$	$2N^2$ *(N/`TILE_WIDTH`)
Throughput	10.7 GFLOPS	183.9 GFLOPS
SLOCs	20	44
Relative Improvement	1x	17.2x
Improvement/SLOC	1x	7.8x

- **Experiment performed on a GT200**
- **This optimization was clearly worth the effort**
- **Better performance still possible in theory**

TILE_SIZE Effects



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`

Questions?