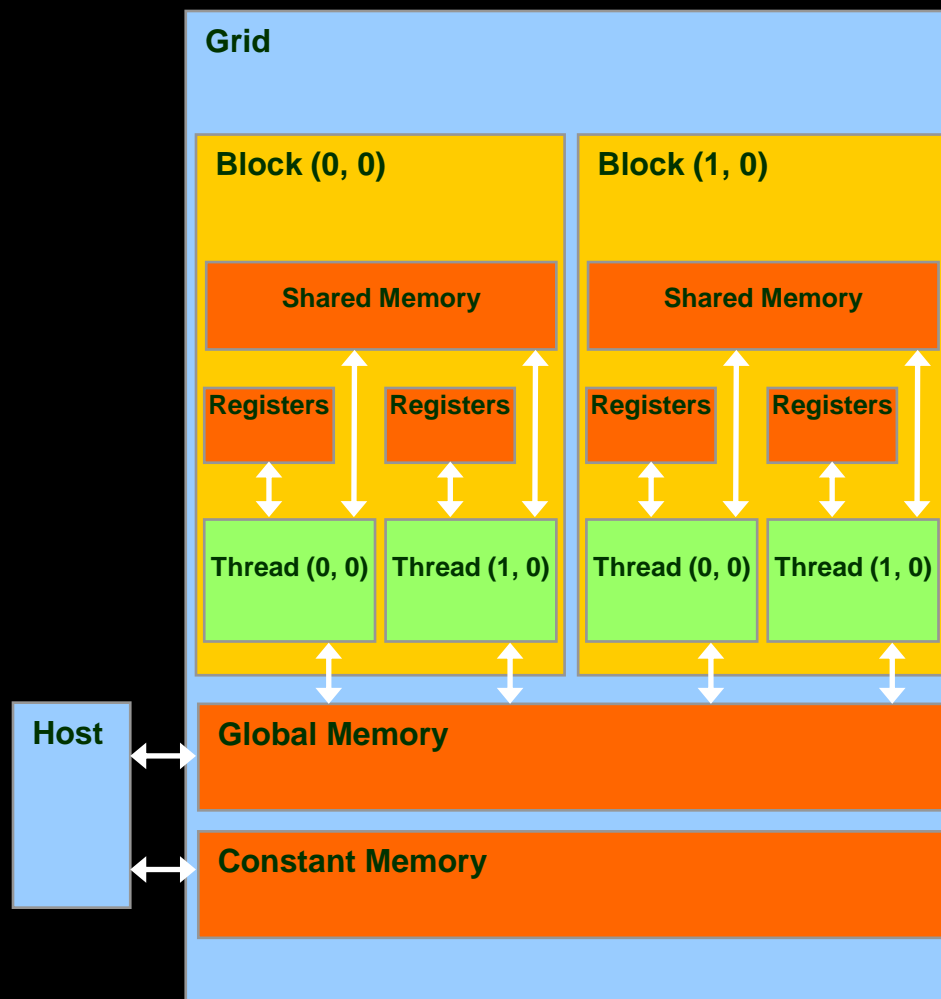# CS 193G

Lecture 4: 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**

Grid

Block (0, 0)

Shared Memory

Registers   Registers

Thread (0, 0)   Thread (1, 0)

Block (1, 0)

Shared Memory

Registers   Registers

Thread (0, 0)   Thread (1, 0)

Host

Global Memory

Constant 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?

```
Can host
access it?
```

Yes   No

```
Outside of
any function
```

```
In the
kernel
```

```
__constant__ int constant_var;    int var;

__device__   int global_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
  ...
}
```

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)

{

  // allocate a __shared__ array with 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 + threadIdx.x;
  s_data[threadIdx.x] = input[i];


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


  if(threadIdx.x > 0)
    result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
  else if(i > 0)
  {
    // handle thread block boundary
    result[i] = s_data[threadIdx.x] - 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**

About Pointers

- **Yes, you can use them!**
- **You can point at any memory space per se:**

```
__device__ int my_global_variable;
__constant__ int my_constant_variable = 13;


__global__ void foo(void)
{
    __shared__ int my_shared_variable;

    int *ptr_to_global = &my_global_variable;
    const int *ptr_to_constant = &my_constant_variable;
    int *ptr_to_shared = &my_shared_variable;
    ...
    *ptr_to_global = *ptr_to_shared;
}
```

About Pointers

- **Pointers aren't typed on memory space**
  - `__shared__ int *ptr;`
  - `ptr` is a `__shared__` pointer variable, not a pointer to a `__shared__` variable!

- **Don't confuse the compiler!**

```
__device__ int my_global_variable;
__global__ void foo(int *input)
{
  __shared__ int my_shared_variable;


  int *ptr = 0;
  if(input[threadIdx.x] % 2)
    ptr = &my_global_variable;
  else
    ptr = &my_shared_variable;
  // to which memory space does ptr point?
}
```

Advice

- **Prefer dereferencing pointers in simple, regular access patterns**
- **Avoid propagating pointers**
- **Avoid pointers to pointers**
    - **The GPU would rather not pointer chase**
    - **Linked lists will not perform well**
- **Pay attention to compiler warning messages**
    - `Warning: Cannot tell what pointer points to, assuming global memory space`
   - **Crash waiting to happen**

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**
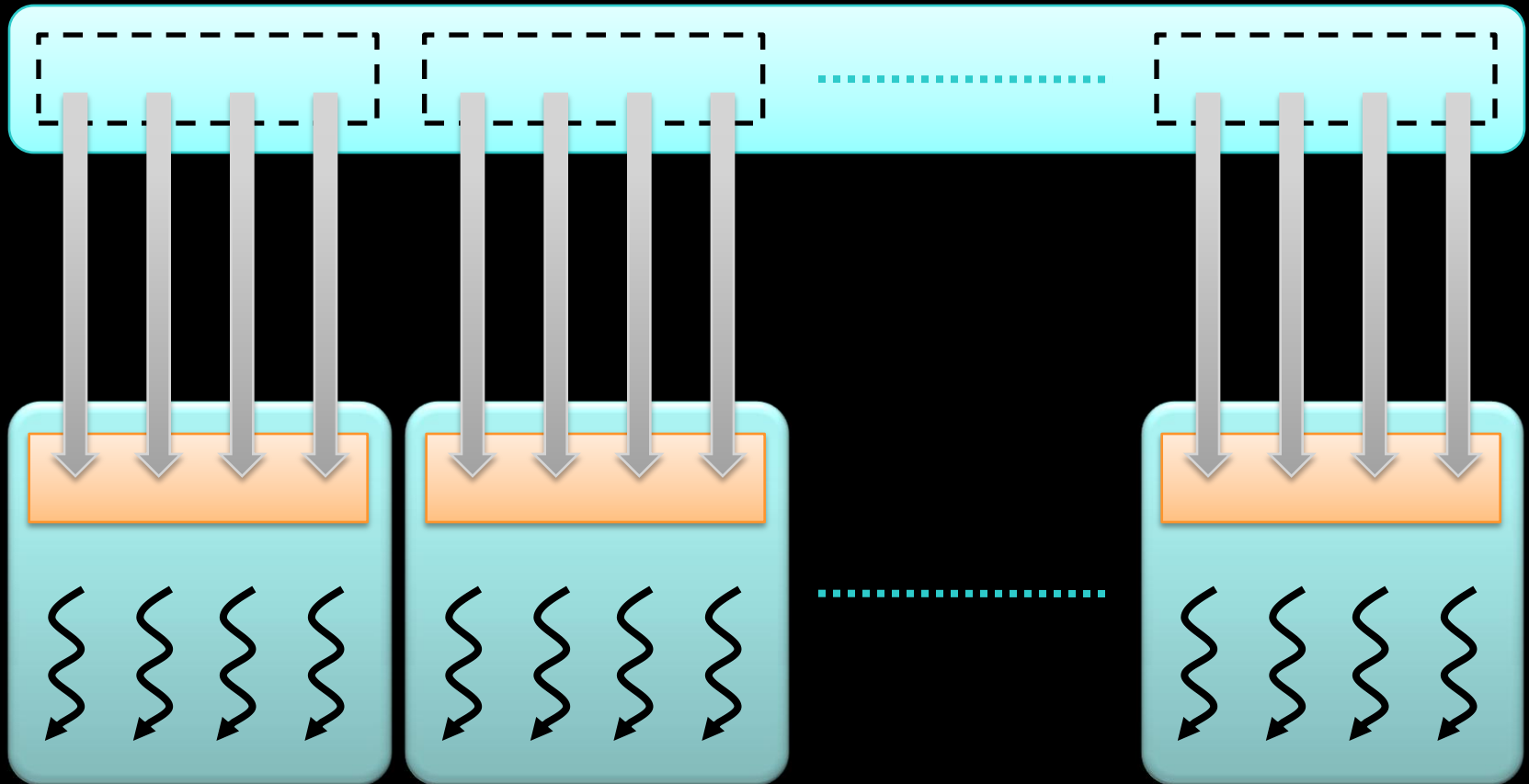 - **Divide and conquer**

A Common Programming Strategy
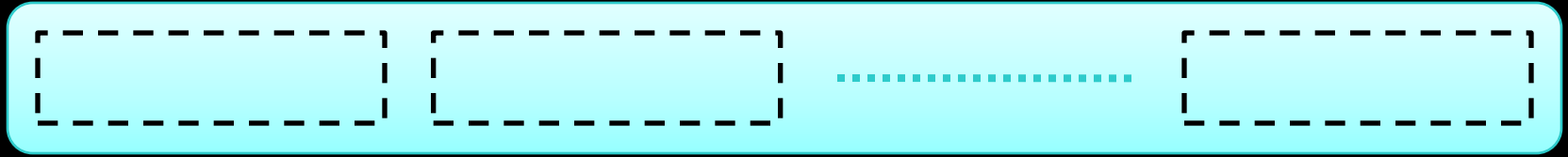
Partition data into **subsets** that fit into **shared memory**
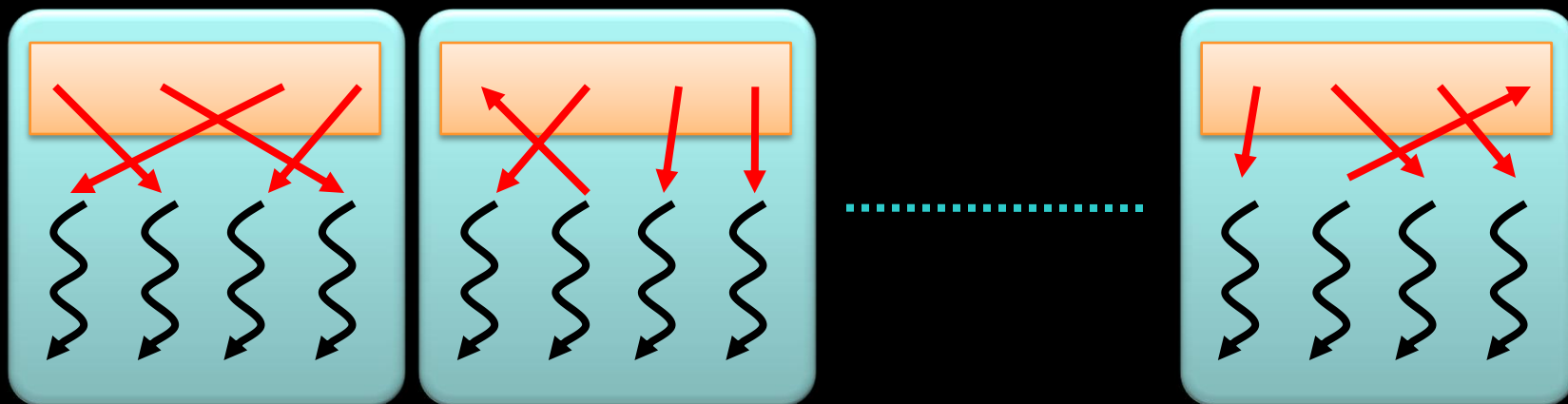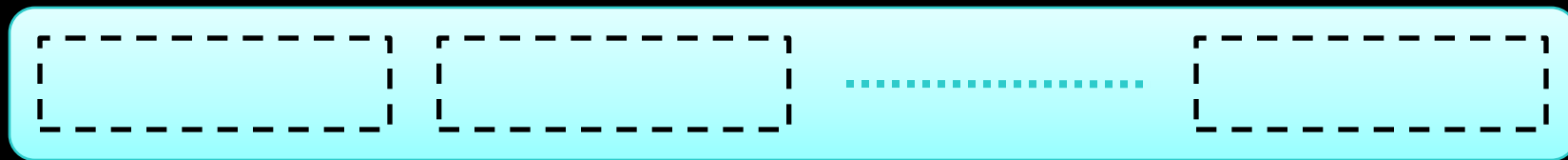
A Common Programming Strategy
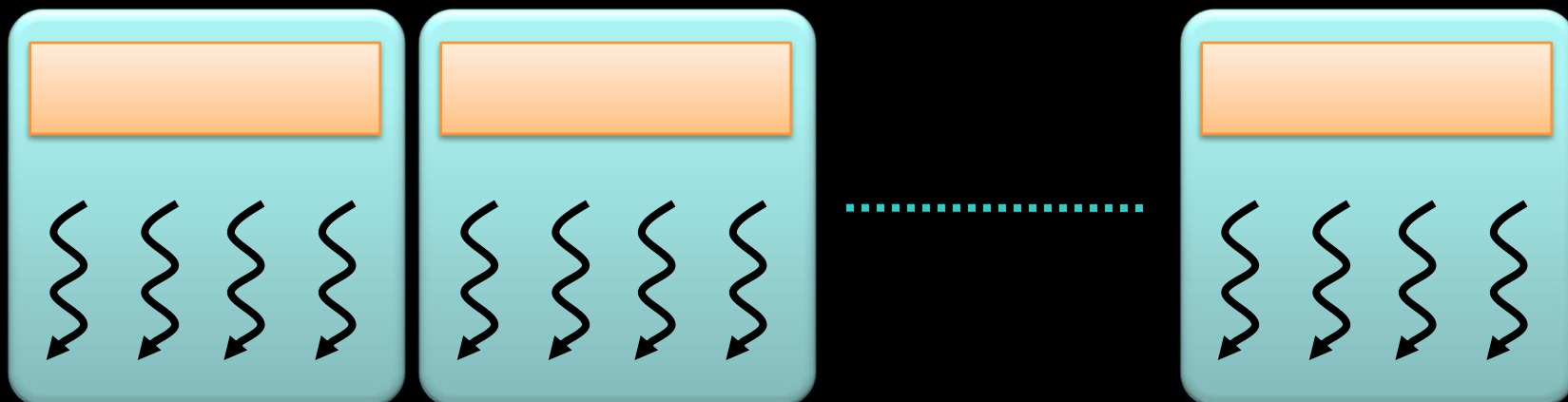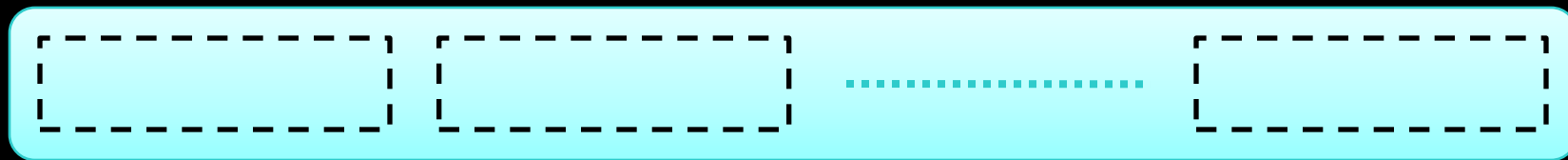
Handle each data subset with one thread block
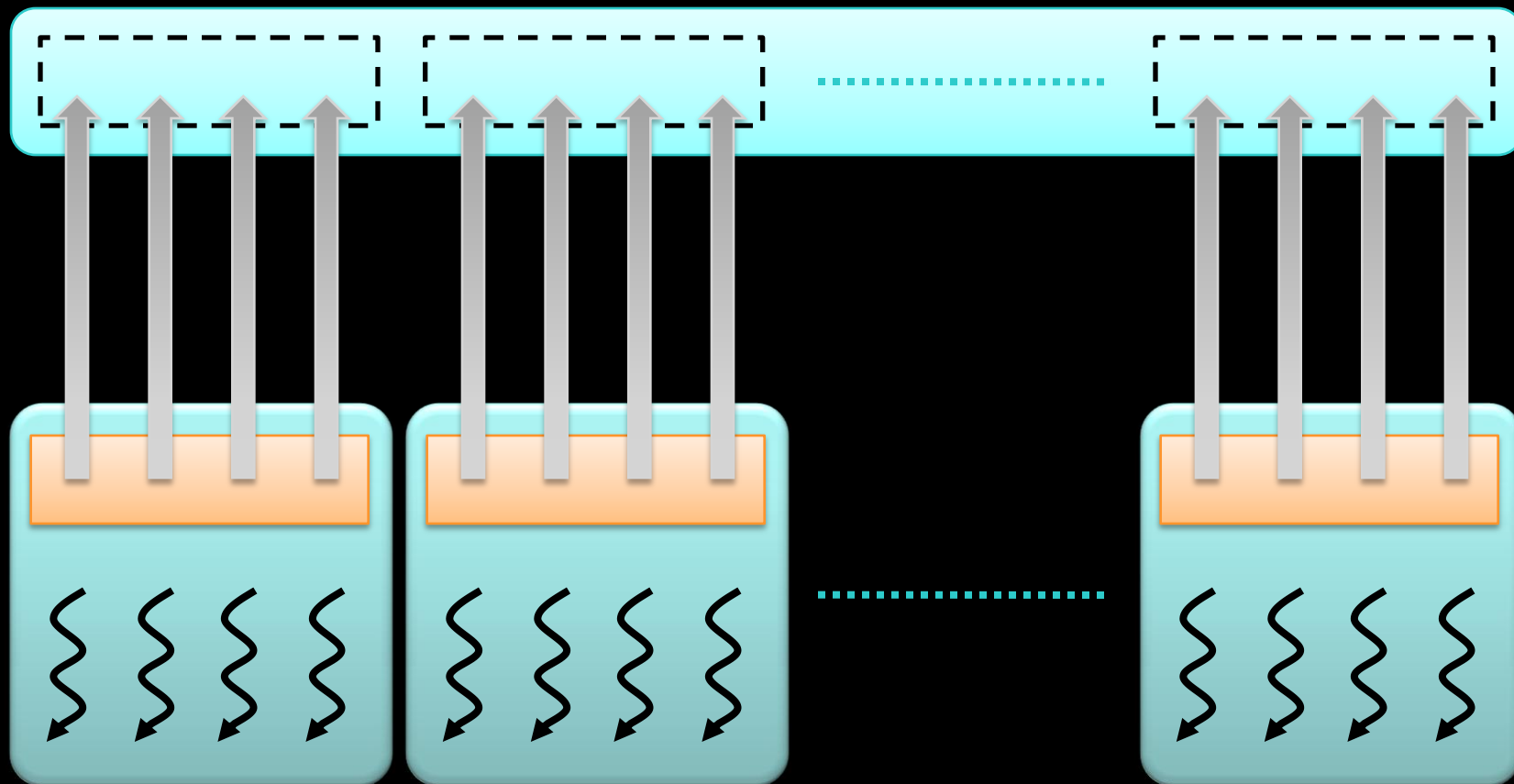
A Common Programming Strategy

- **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
}
```
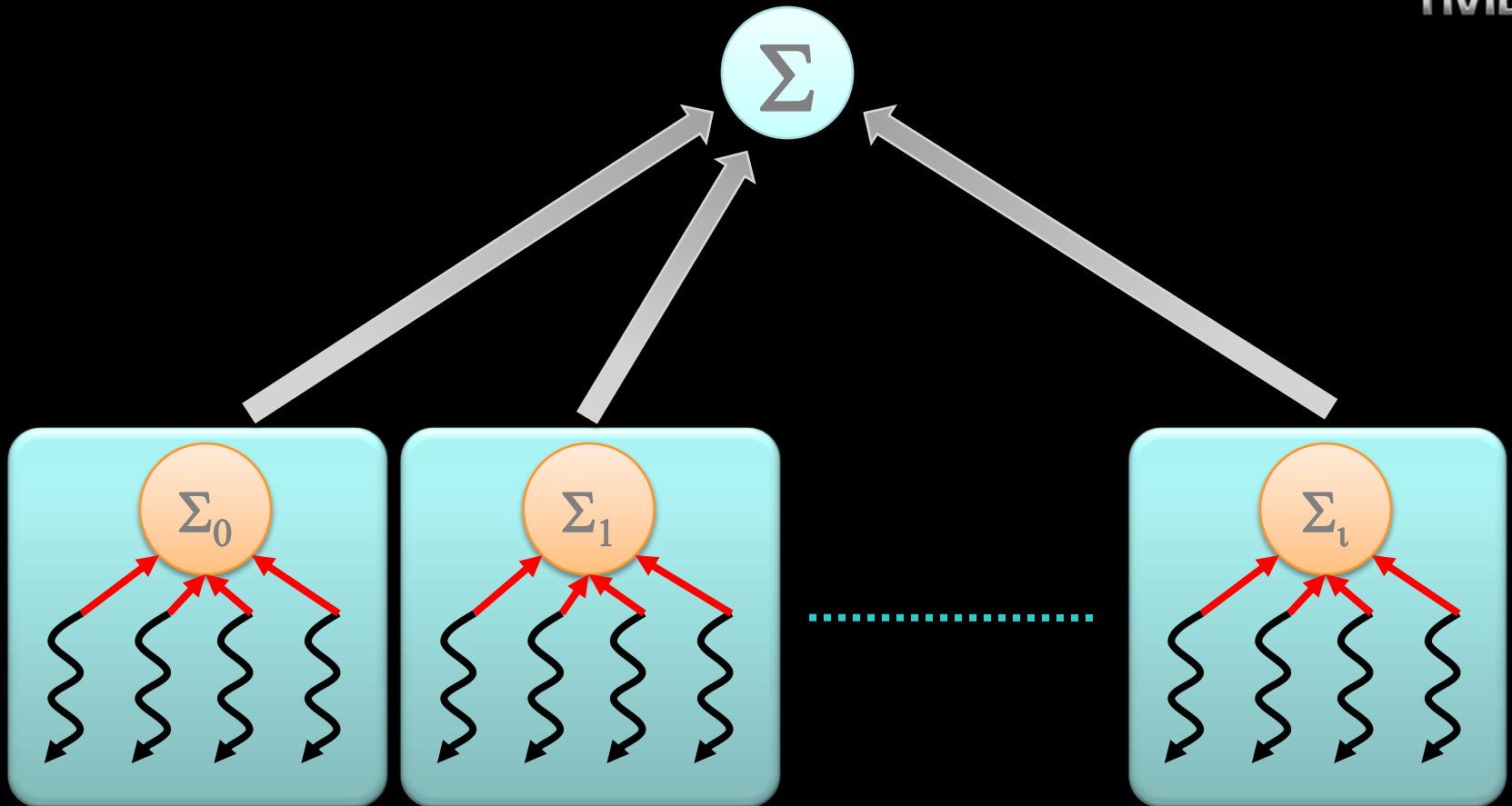
Resource Contention

- **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

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

    if(threadIdx.x == 0)
        partial_sum = 0;
    __syncthreads();

    // each thread updates the per-block partial sum
    atomicAdd(&partial_sum, input[threadIdx.x]);
    __syncthreads();

    // thread 0 of each block 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**

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
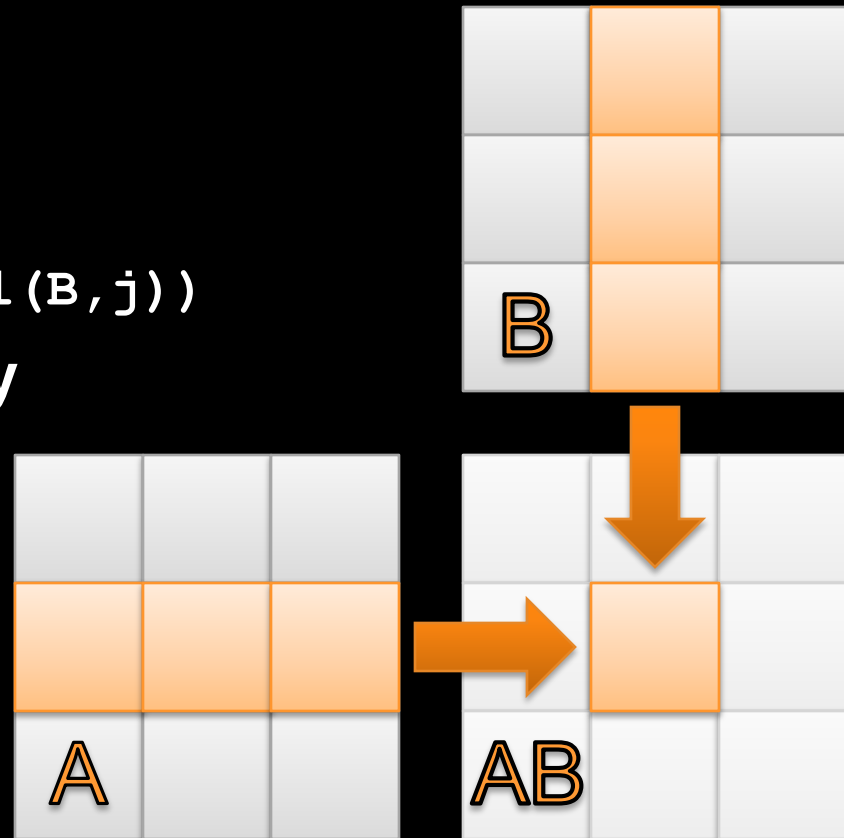
First Implementation

```
__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**

width

Tiled Multiply

TILE_WIDTH
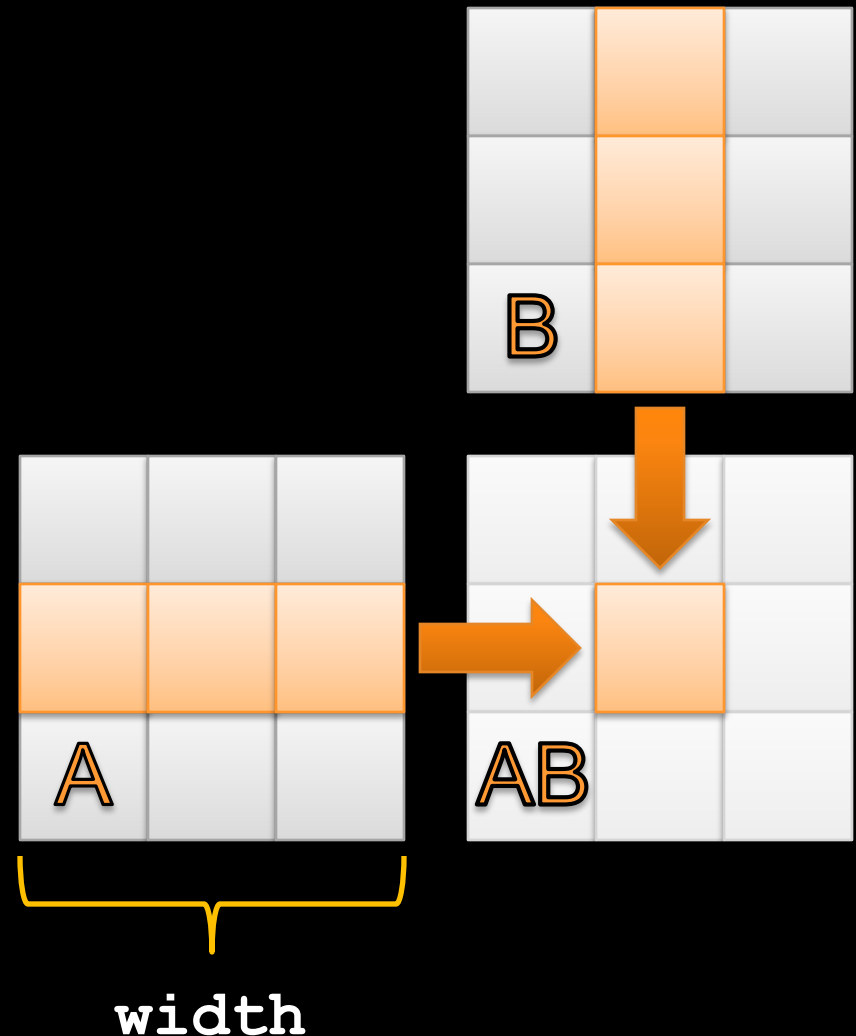
- **Partition kernel loop into phases**

- **Load a tile of both matrices into __shared__ each phase**

- **Each phase, each thread computes a partial result**

B

A

AB
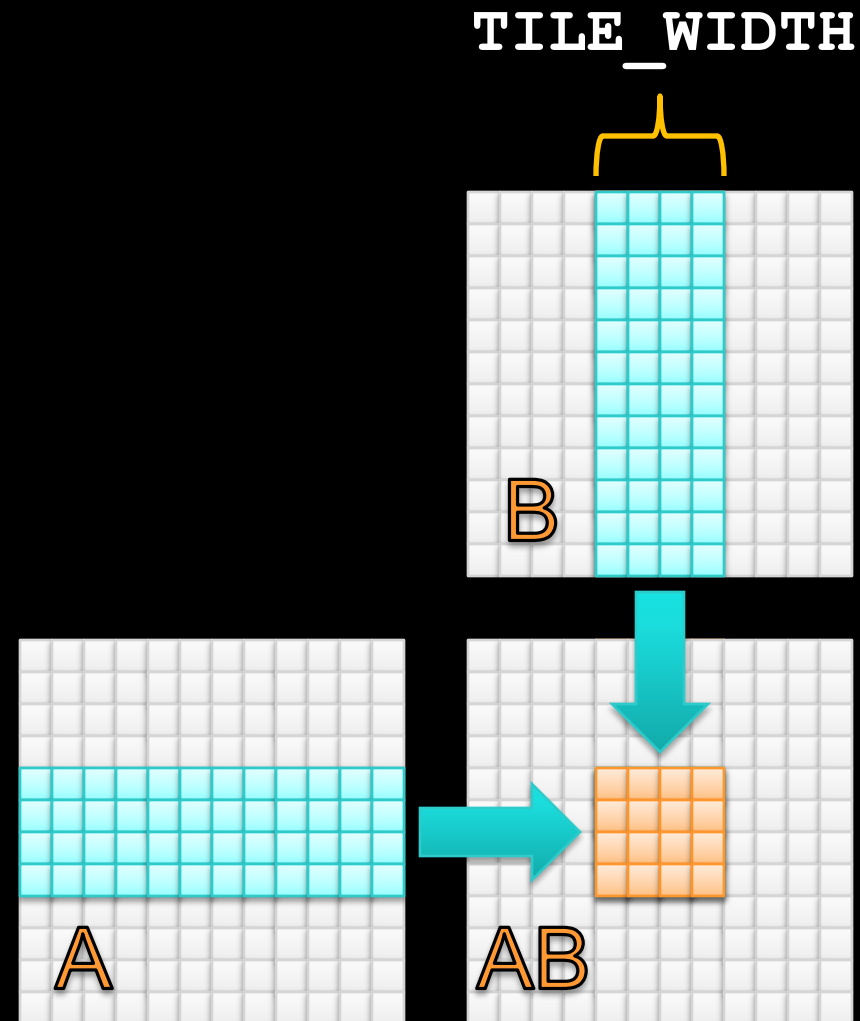
Better Implementation

```
__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
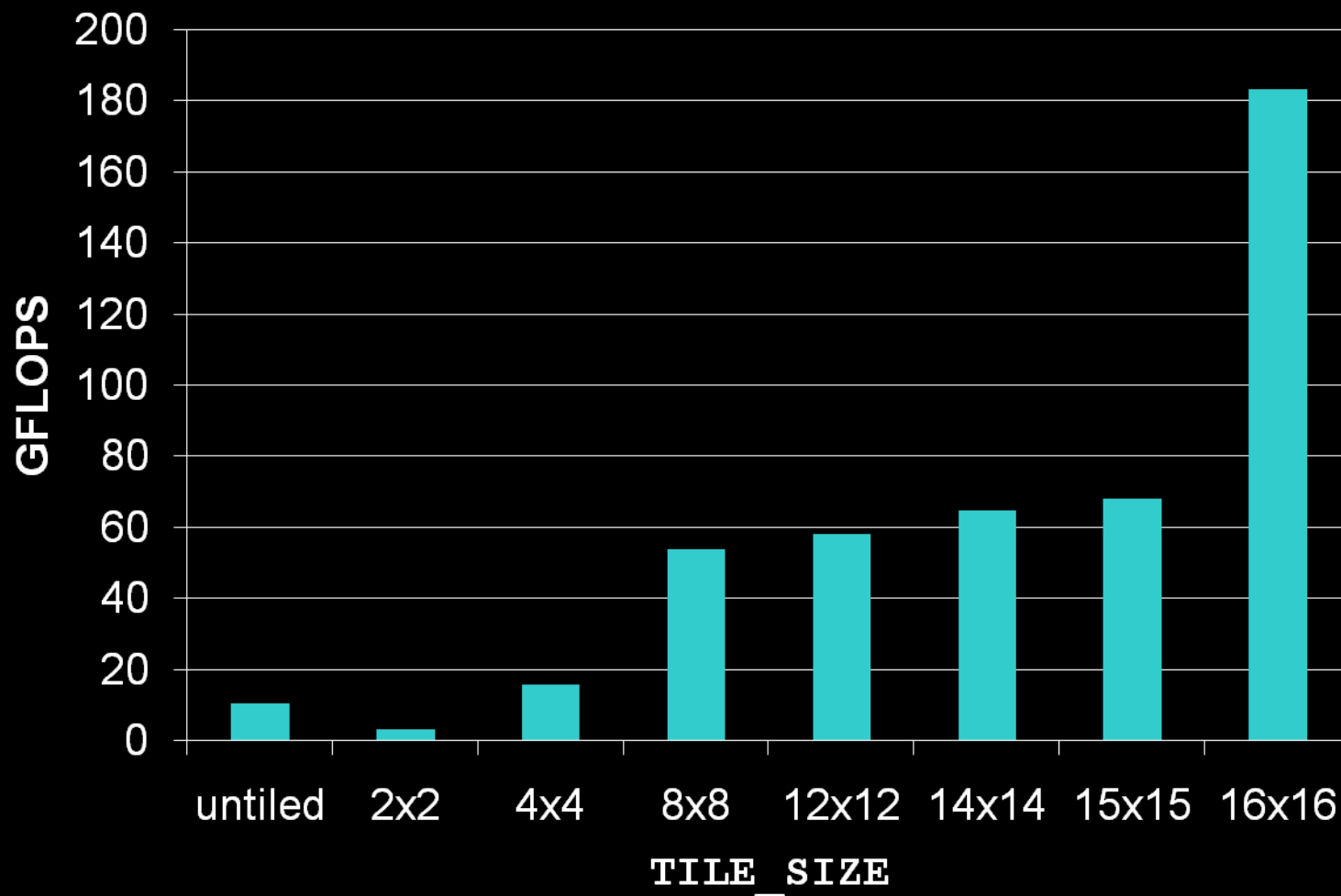
Memory Resources as Limit to Parallelism

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	<= 16384 / 768 threads = 21 per thread
__shared__ Memory	16KB	<= 16KB / 8 blocks = 2KB per block

- **Effective use of different memory resources reduces the number of accesses to global memory**
- **These resources are finite!**
- **The more memory locations each thread requires → the fewer threads an SM can accommodate**

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`