Review Last Class

1, Case Study one, diff() using shared memory

2, # of Global mem access reduced

Today Class

1, Two ways to define shared memory

2, Common Cuda Programming Strategy

3, Memory Access Efficiency, CGMA

Two ways to Define Shared Mem

Using shared memory

- using _ _shared_ _ qualifier, declare memory as shared.
- Two ways of defining shared memory
  - **Static, a local array of fixed size,**

e.g.

```
_ _ global_ _ kernel3(…..)
{
    _ _shared_ _ float sdata[BLOCKSIZE]; //could be 2D, a[][]
    sdata[sindex] = gA[gindex];
    ……
}
```

Two ways to Define Shared Mem

- Two ways of defining shared memory

Dynamic (second kind)

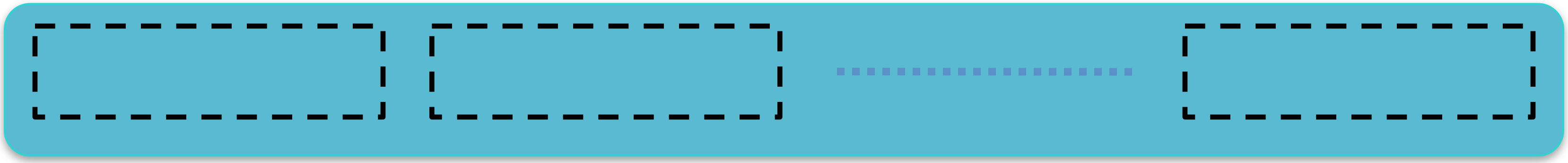
```
// when the size of the array isn't known at compile time...
__global__ void adj_diff(int *result, int *input)
{
  // use extern to indicate a __shared__ array will be
  // allocated dynamically at kernel launch time
  extern __shared__ int s_data[];
  ...
}
```

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

A Common Programming Strategy

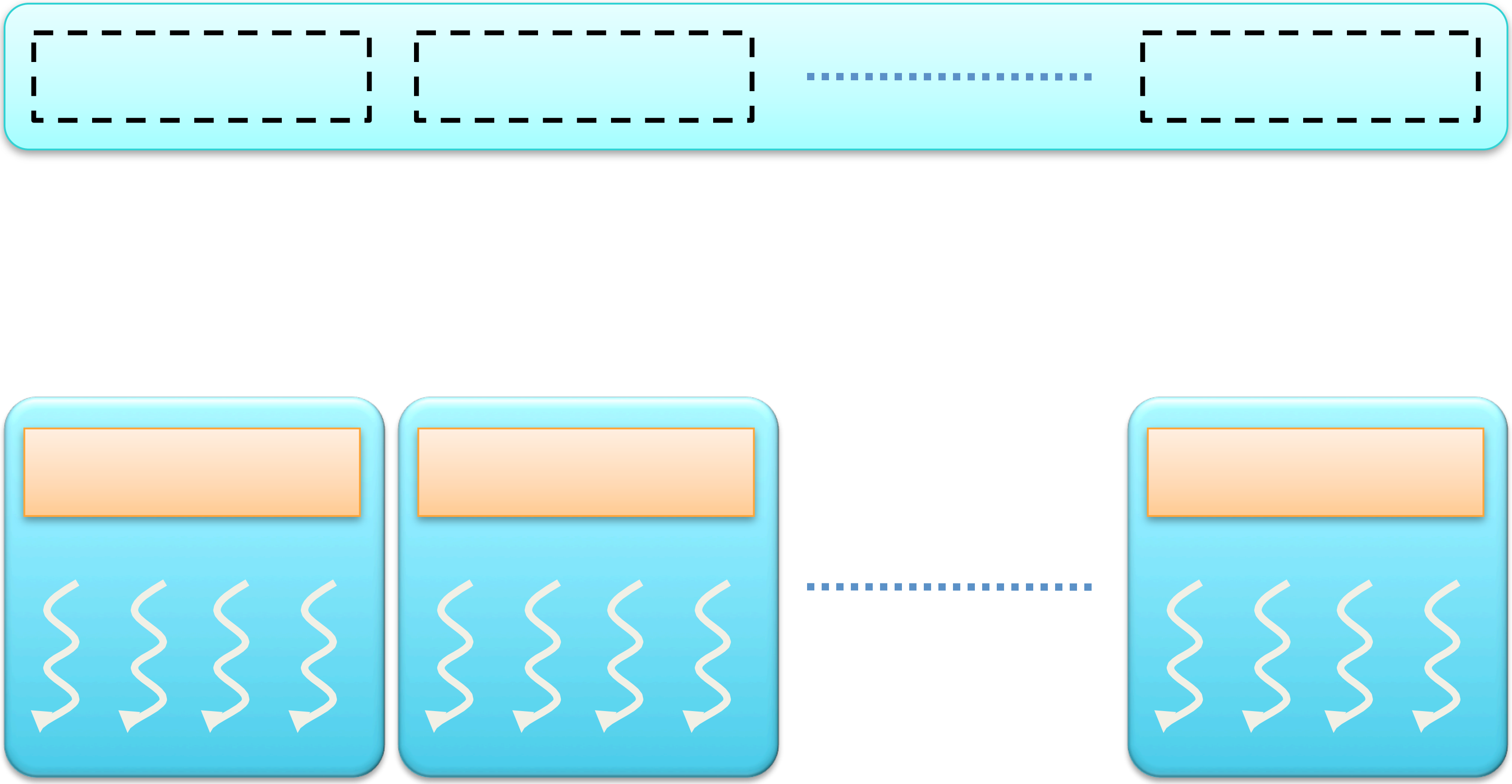
- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory
- Tile data to take advantage of fast shared memory:
  - Generalize from `adjacent_difference` example
  - Divide and conquer

A Common Programming Strategy



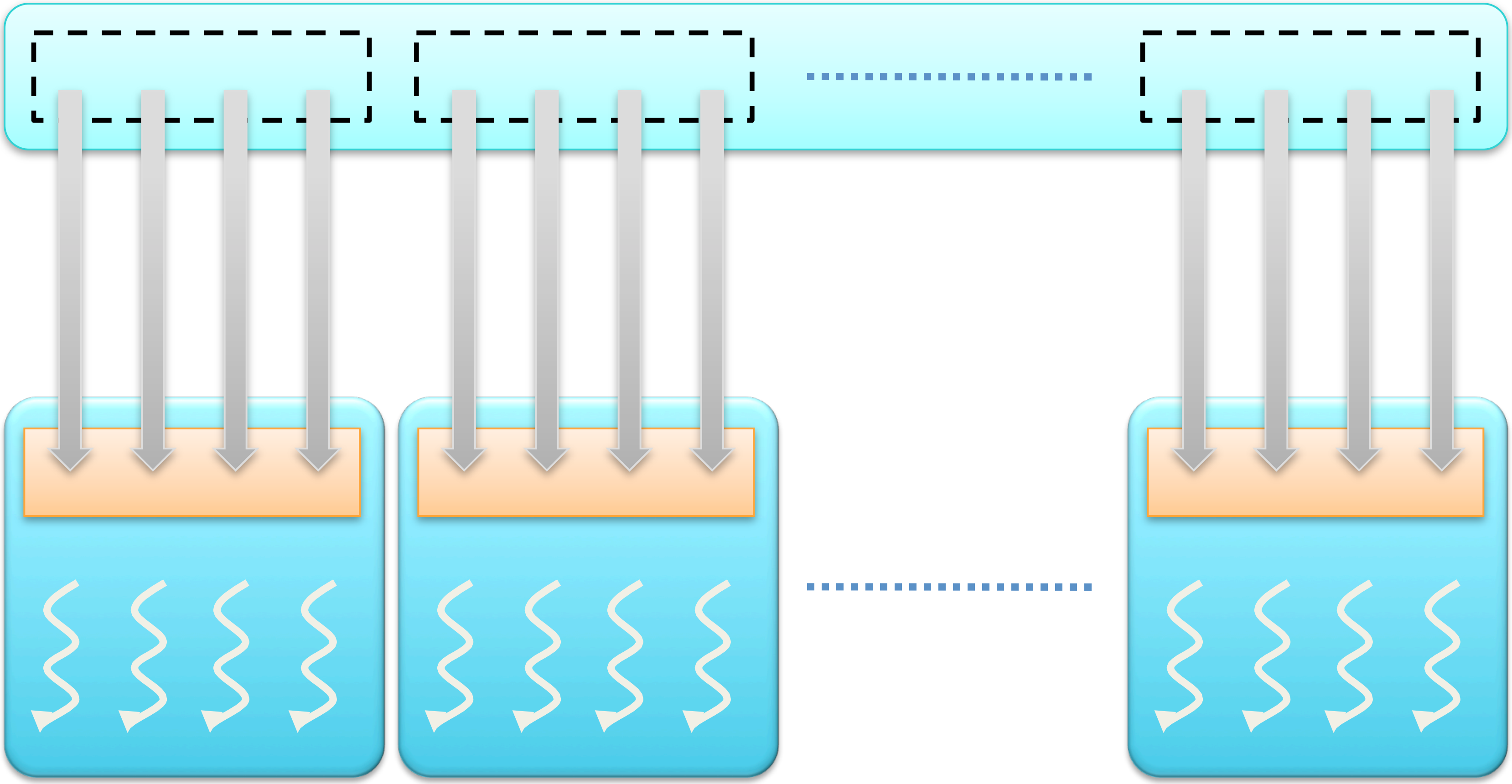
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



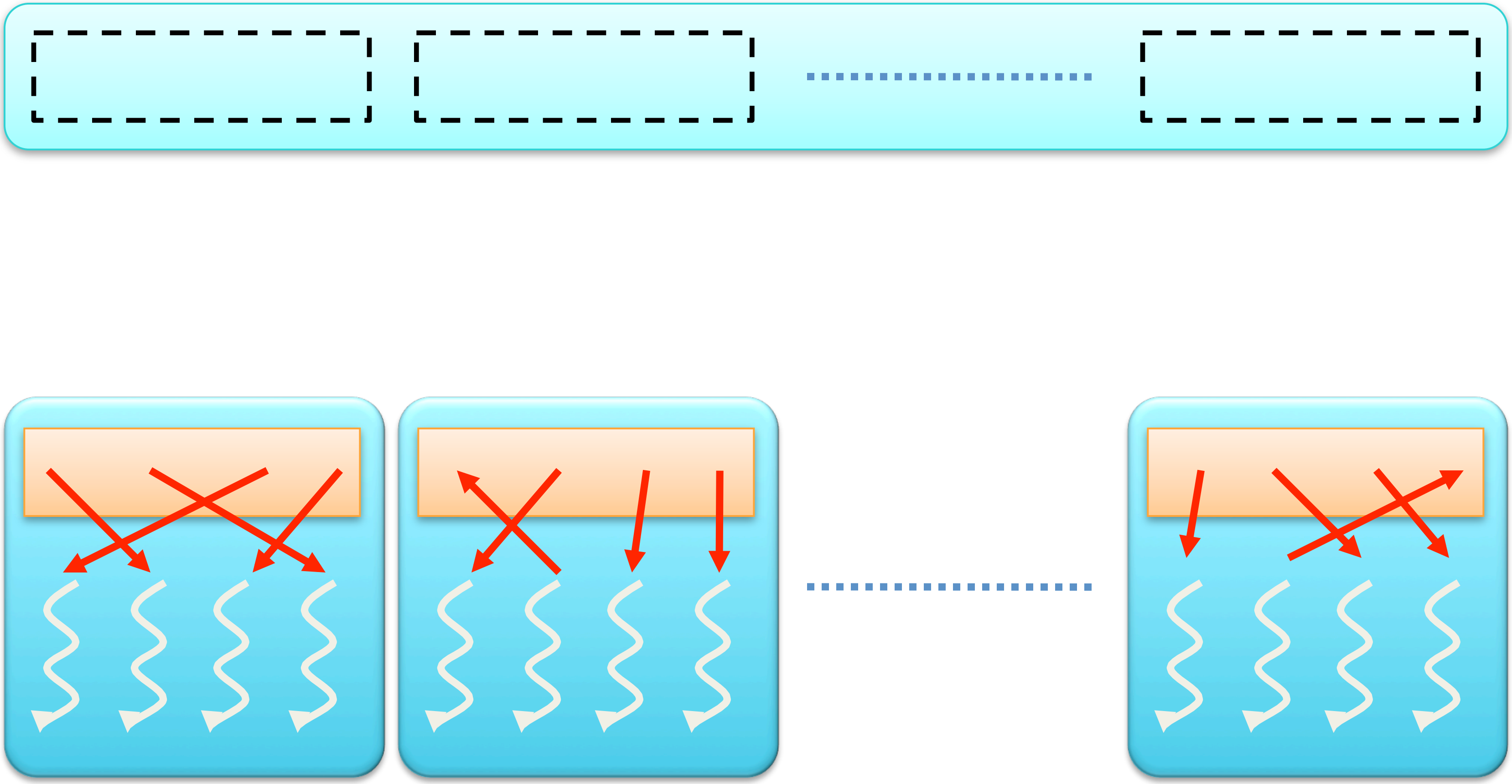
- Handle each data subset with one thread block

A Common Programming Strategy



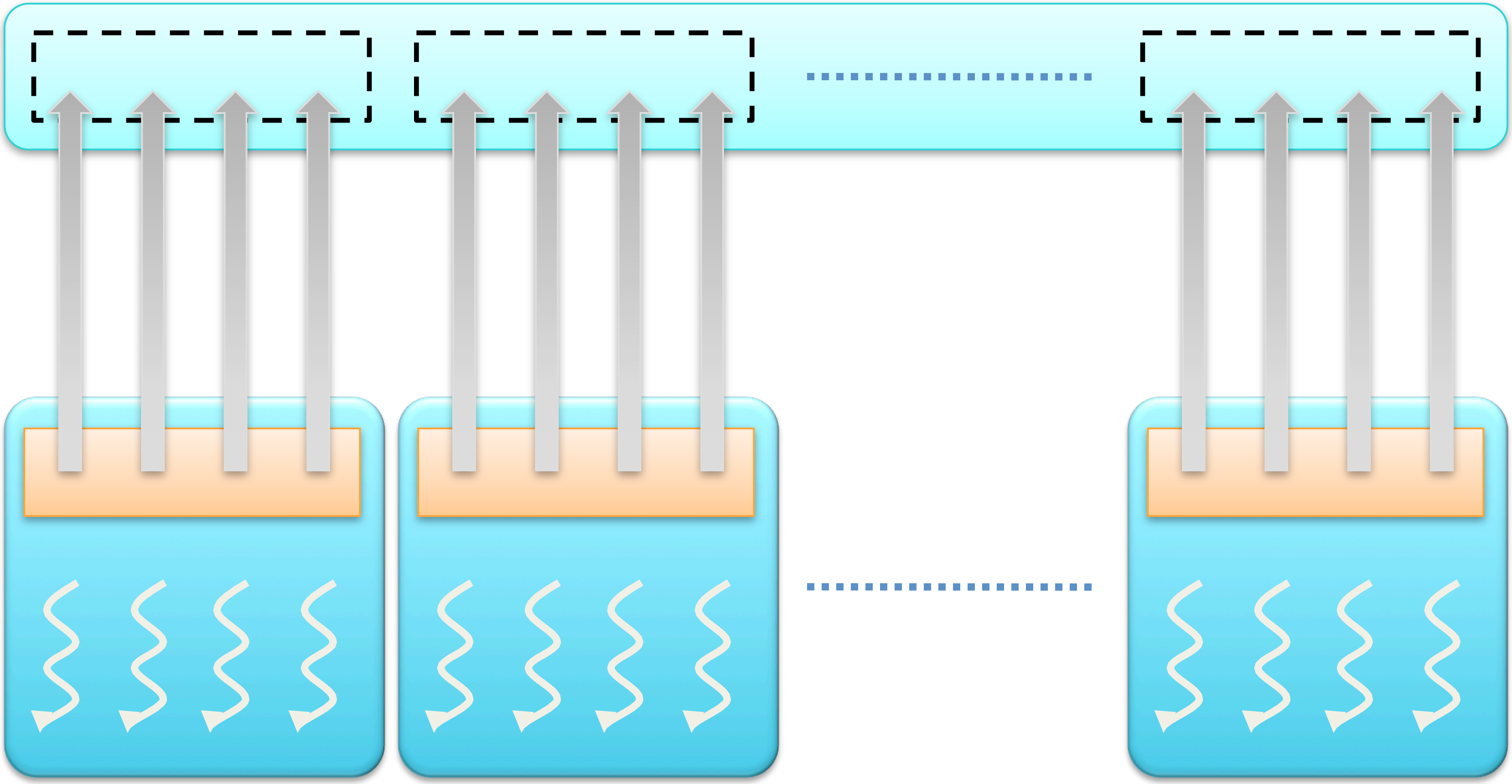
- Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism

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 causes 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

Mem Access Efficiency

- Without Using Shared Memory
  - Traffic Congestion in global memory access
    - Like the simple(naïve) kernel of matrix multiplication.
  - Few threads proceed, some SMs are idles.
- Shared Mem used to reduce # of global memory access.

Mem Access Efficiency

- Without Using Shared Memory
  - Traffic Congestion in global memory access
    - Like the simple(naïve) kernel of matrix multiplication.
  - Few threads proceed, some SMs are idles.
- Shared Mem used to reduce # of global memory access.

Mem Access Efficiency

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int K=0 ; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 4.7

A simple matrix–matrix multiplication kernel using one thread to compute each d_P element.

Mem Access Efficiency

- CGMA
  - Ratio of floating-point calculation to global memory access
  - Major kernel indicator
  - The kernel of previous slide, has CGMA = 1.0.
- Assume a GPU has global memory bandwidth of 200GB/s, and float type takes 4 bytes,
  - Throughput is no more than 50 Giga float operands per second. (200 / 4)

Mem Access Efficiency

- Throughput is no more than 50 Giga float operands per second. (200 / 4)
- With a CGMA= 1.0, the naïve matrix multiplication kernel execute no more than 50 Giga float operations per second
  - That is 50 GFLOPS.
  - Much less than 1500 GFLOPS of the peak performance.
  - We have to increase the CGMA.

Mem Access Efficiency

- To achieve the peak performance of 1500 GFLOPS,
- (If global bandwidth is 200GB/s), what the desirable value of CGMA?
- 30
- Shared memory is one of techniques to boost CGMA, thus boost performance.

Wrap up

1, Two ways to define shared memory

2, Common Cuda Programming Strategy

3, Memory Access Efficiency, CGMA

Next Class

1, Tiled (improved) matrix multiplication

2, Memory as a limiting factor to parallelism