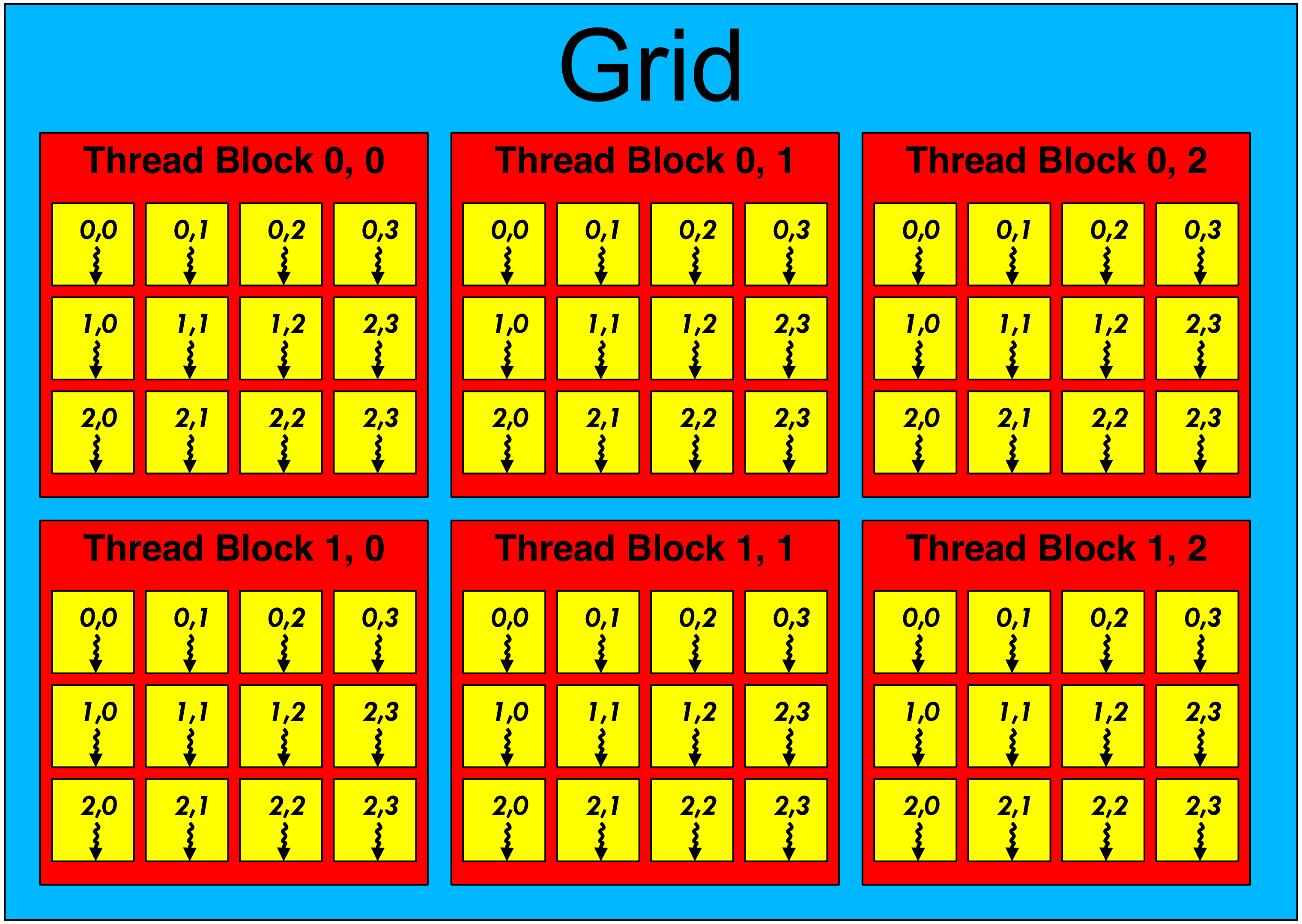
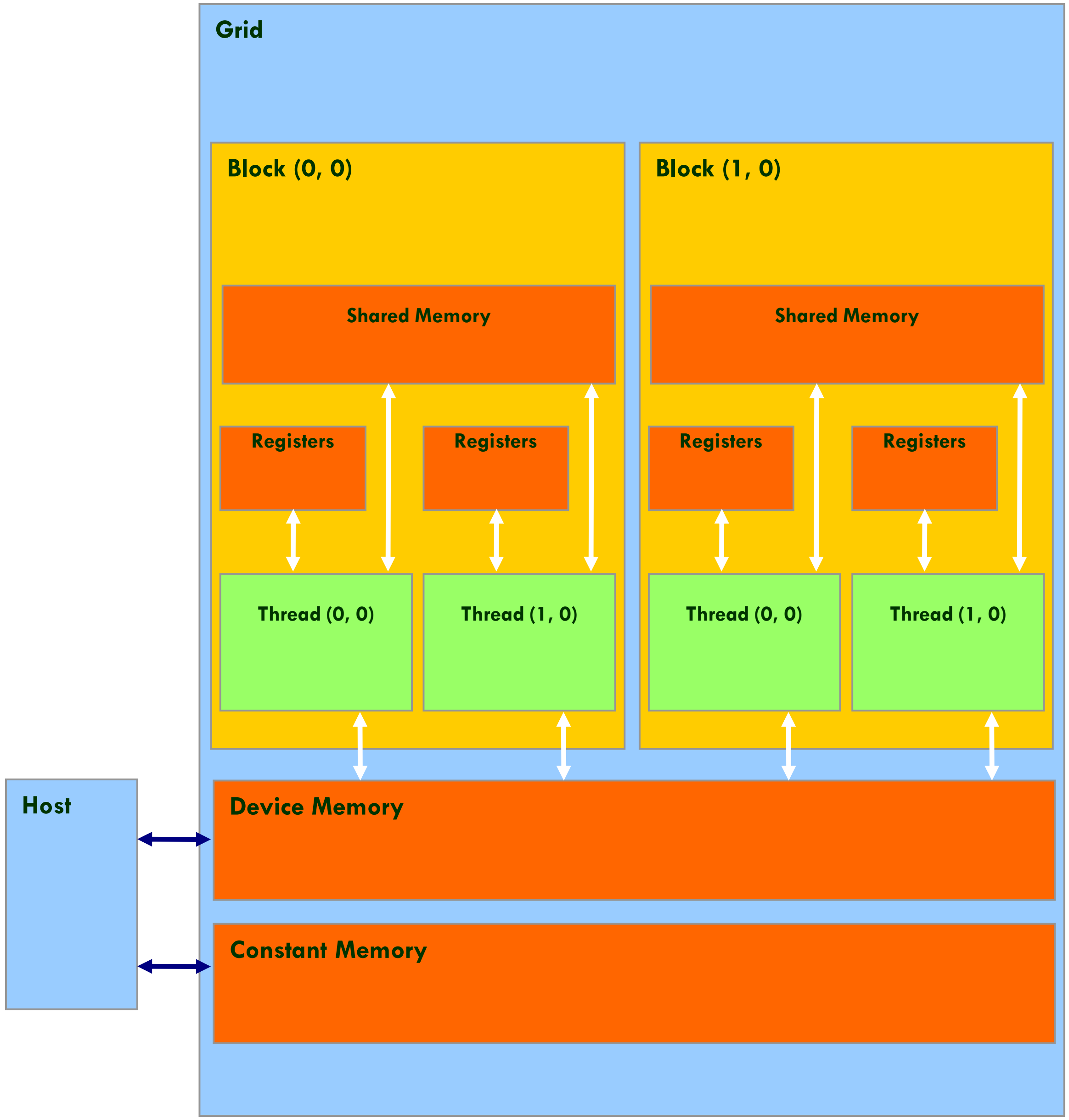
Grids, Thread Blocks and Threads



Hardware Memory Spaces in CUDA



Vector addition GPU code

```
// compute vector sum c = a + b
// each thread performs one pair-wise addition
__global__ void vector_add(float* A, float* B, float* C) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    C[i] = A[i] + B[i];
}
```

GPU code

```
int main() {
  // initialization code here ...

  // launch N/256 blocks of 256 threads each
  vector_add<<< N/256, 256 >>>(deviceA, deviceB, deviceC);

  // cleanup code here ...
}
```

Host code

(can be in the same file)

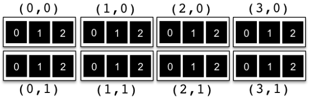
- The host always initiates work for the device
  - For Kepler GPUs, device can generate work for itself (Dynamic Parallelism)

- Examples for common extensions are
  - `__global__` defines a GPU kernel that is callable by the CPU
  - `__device__` defines a GPU function that is callable by a GPU kernel or by another device function, but not callable by a CPU (host)
  - `__host__` or no qualifier marks a CPU function

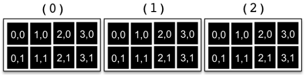
- A CUDA kernel is a grid of thread blocks
  - The grid can be 1D, 2D or 3D
- A thread block is, in turn, a 1D, 2D, or 3D array of threads
- A thread is executed by exactly one stream processor or CUDA core
- A thread block is executed by exactly one Streaming Multiprocessor (SM)
- However, CUDA cores and SM can interleave execution of multiple threads and thread blocks

- CUDA maintains special variables that store:
  - thread index within a thread block `threadIdx(.x, .y, .z)`
  - block index within a kernel grid `blockIdx(.x, .y, .z)`
  - dimension of a thread block `blockDim(.x, .y, .z)`
  - dimension of a kernel grid `gridDim(.x, .y, .z)`
- These variables are used without declaration
- Mainly used in assigning the work per block/thread
- The maximum values of the `(x, y, z)` in both `blockDim` and `gridDim` are GPU-dependent
- At least the `x` component needs to be declared. The default value for the `y` and `z` components is 1

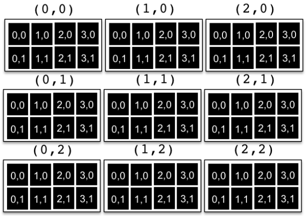
```
dim3 dimBlock(3)
dim3 dimGrid(4,2)
```



```
dim3 dimBlock(4, 2)
dim3 dimGrid(3)
```



```
dim3 dimBlock(4, 2)
dim3 dimGrid(3, 3)
```



Thread Scheduling

- Order in which thread blocks are scheduled is undefined!
  - any possible interleaving of blocks should be valid
  - presumed to run to completion without preemption
  - can run in any order
  - can run concurrently OR sequentially

- Order of threads within a block is also undefined!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

Global synchronization

- Q: How do we do global synchronization with these scheduling semantics?
- A1: Not possible!
- A2: Finish a grid, and start a new one!

```
step1<<<grid1,blk1>>>(...);
// CUDA ensures that all writes from step1 are complete.
step2<<<grid2,blk2>>>(...);
```

- We don't have to copy the data back and forth!

Atomics

- Guarantee that only a single thread has access to a piece of memory during an operation
- No dropped data, but ordering is still arbitrary
- Different types of atomic instructions
- Atomic Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor
- Can be done on device memory and shared memory
- Much more expensive than load + operation + store

Example: Histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
```

Example: Histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    buckets[c] += 1;
}
```

Why is this incorrect?

Example: Histogram

```
// Determine frequency of colors in a picture.
// Colors have already been converted into integers
// between 0 and 255.
// Each thread looks at one pixel,
// and increments a counter atomically

__global__ void histogram(int* colors, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

Example: Work queue

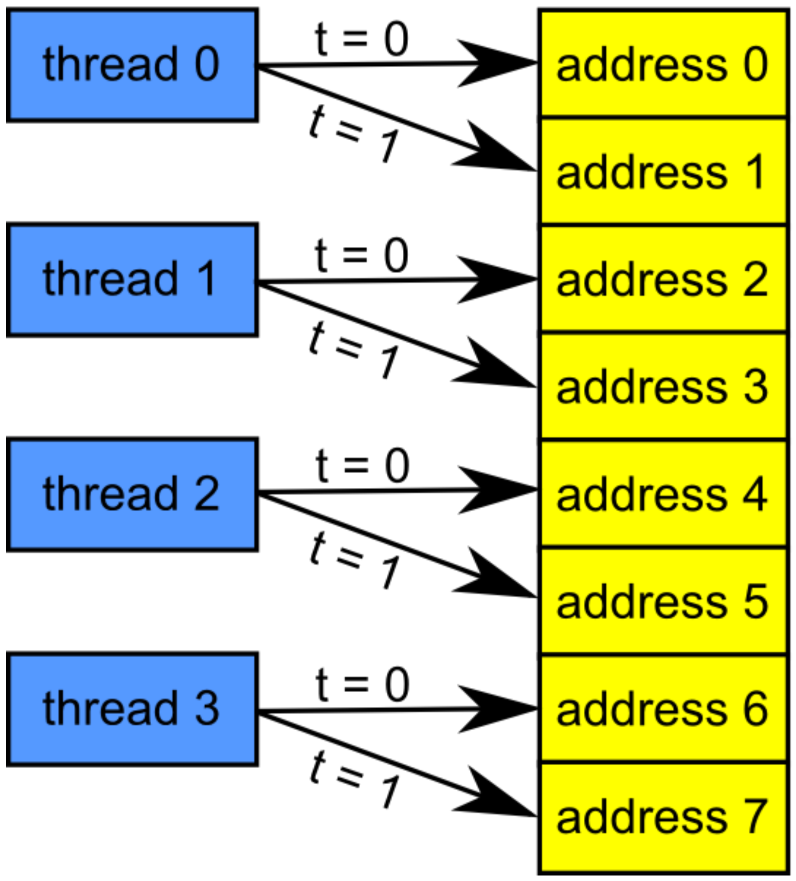
```
// For algorithms where the amount of work per item
// is highly non-uniform, it often makes sense to
// continuously grab work from a queue.

__global__
void workq(int* work_q, int* q_counter,
           int queue_max, int* output)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int q_index = atomicInc(q_counter, queue_max);
    int result = do_work(work_q[q_index]);
    output[i] = result;
}
```

CUDA: optimizing your application

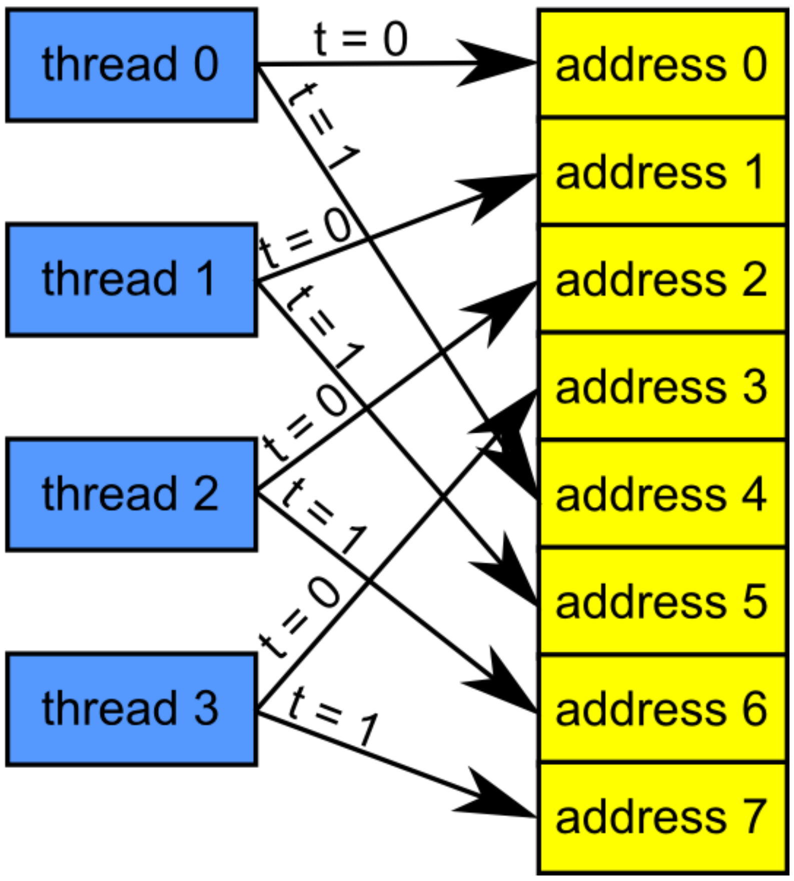
Coalescing

Coalescing

traditional multi-core
optimal memory access pattern



many-core GPU
optimal memory access pattern



Consider the stride of your accesses

```
__global__ void foo(int* input, float3* input2) {
   int i = blockDim.x * blockIdx.x + threadIdx.x;

   // Stride 1, OK!
   int a = input[i];

   // Stride 2, half the bandwidth is wasted
   int b = input[2*i];

   // Stride 3, 2/3 of the bandwidth wasted
   float c = input2[i].x;
}
```

Example: Array of Structures (AoS)

```
struct record {
    int key;
    int value;
    int flag;
};

record *d_records;
cudaMalloc((void**)&d_records, ...);
```

Example: Structure of Arrays (SoA)

```
Struct SoA {
    int* keys;
    int* values;
    int* flags;
};

SoA d_SoA_data;
cudaMalloc((void**)&d_SoA_data.keys, ...);
cudaMalloc((void**)&d_SoA_data.values, ...);
cudaMalloc((void**)&d_SoA_data.flags, ...);
```

Example: SoA vs AoS

```
__global__ void bar(record* AoS_data,
                    SoA  SoA_data) {
  int i = blockDim.x * blockIdx.x + threadIdx.x;

  // AoS wastes bandwidth
  int key1 = AoS_data[i].key;

  // SoA efficient use of bandwidth
  int key2 = SoA_data.keys[i];
}
```

Memory Coalescing

- Structure of arrays is often better than array of structures
- Very clear win on regular, stride 1 access patterns
- Unpredictable or irregular access patterns are case-by-case
- Can lose a factor of 10 – 30!

CUDA: optimizing your application

Shared Memory

Using shared memory

```
// 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 device memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

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

Using shared memory

```
// 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 device memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

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

How do we use device memory bandwidth?

Using shared memory

```
// 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 device memory
    int x_i = input[i];
    int x_i_minus_one = input[i-1];

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

How do we use device memory bandwidth?

The next thread also reads input[i]

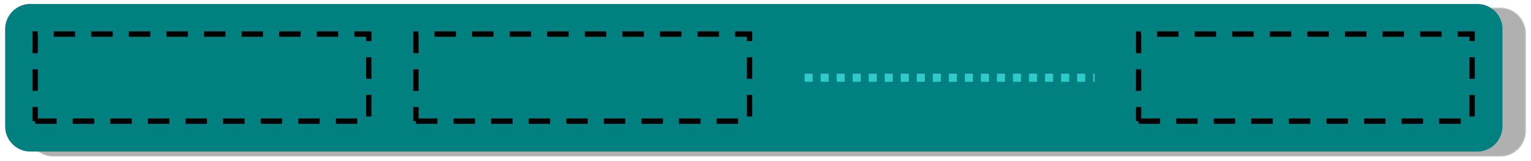
Using shared memory

```
__global__ void adj_diff(int *result, int *input) {
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
  // each thread reads 1 device memory elt, stores it in s_data
  s_data[threadIdx.x] = input[i];

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

  if(threadIdx.x > 0) {
    result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
  } else if(i > 0) {
    // I am thread 0 in this block: handle thread block boundary
    result[i] = s_data[threadIdx.x] - input[i-1];
  }
}
```

Using shared memory: coalescing

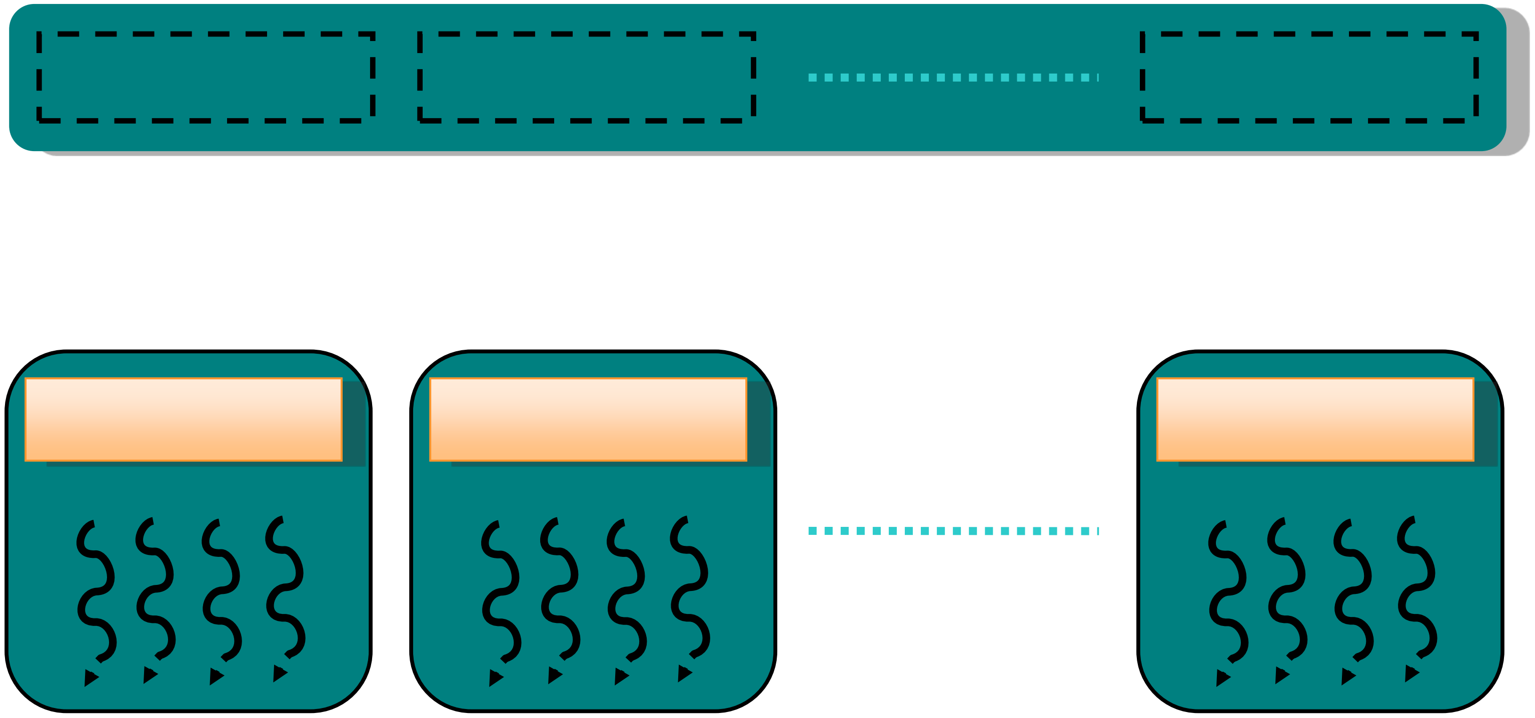
```
__global__ void adj_diff(int *result, int *input) {
  unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

  __shared__ int s_data[BLOCK_SIZE]; // shared, 1 elt / thread
  // each thread reads 1 device memory elt, stores it in s_data
  s_data[threadIdx.x] = input[i];    // COALESCED ACCESS!

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

  if(threadIdx.x > 0) {
    result[i] = s_data[threadIdx.x] - s_data[threadIdx.x-1];
  } else if(i > 0) {
    // I am thread 0 in this block: handle thread block boundary
    result[i] = s_data[threadIdx.x] - input[i-1];
  }
}
```

A Common Programming Strategy



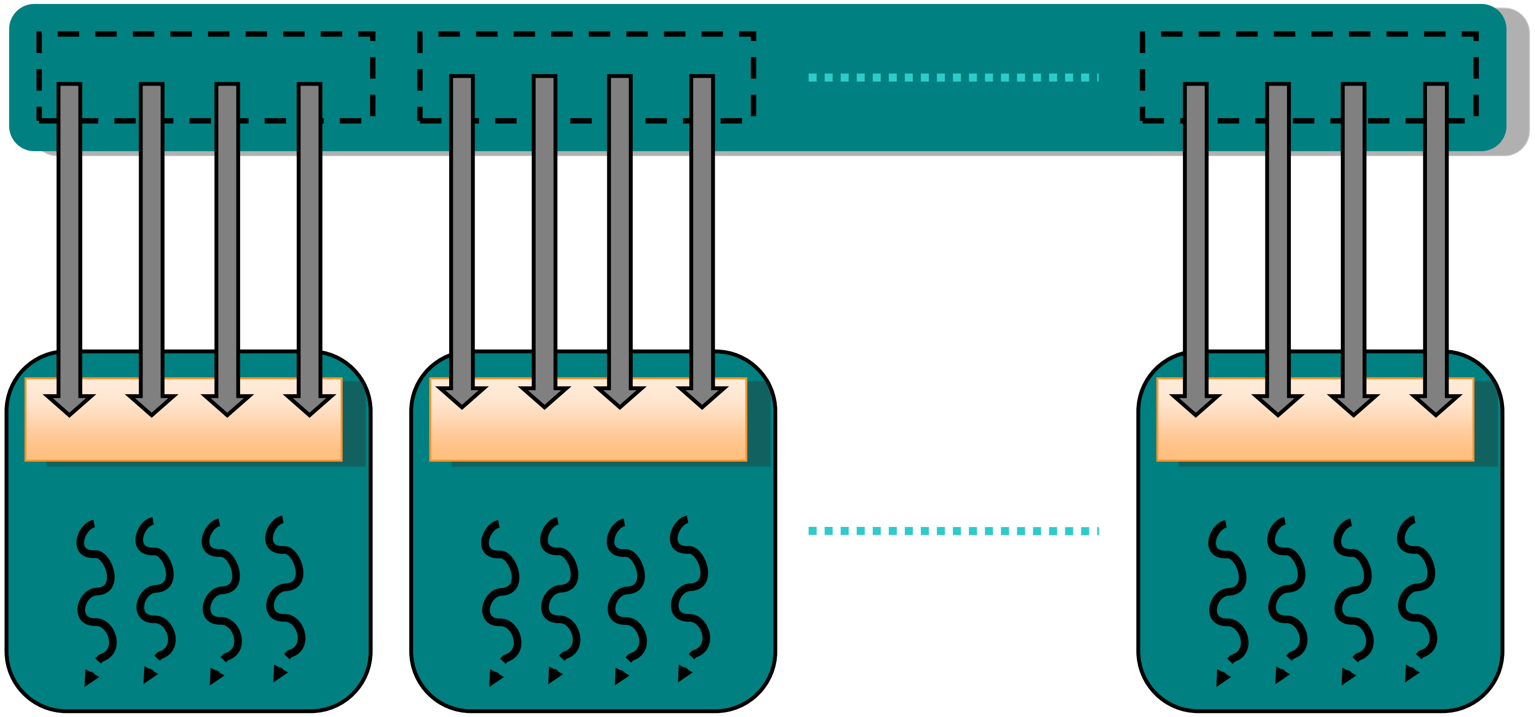
- Partition data into subsets that fit into shared memory

A Common Programming Strategy



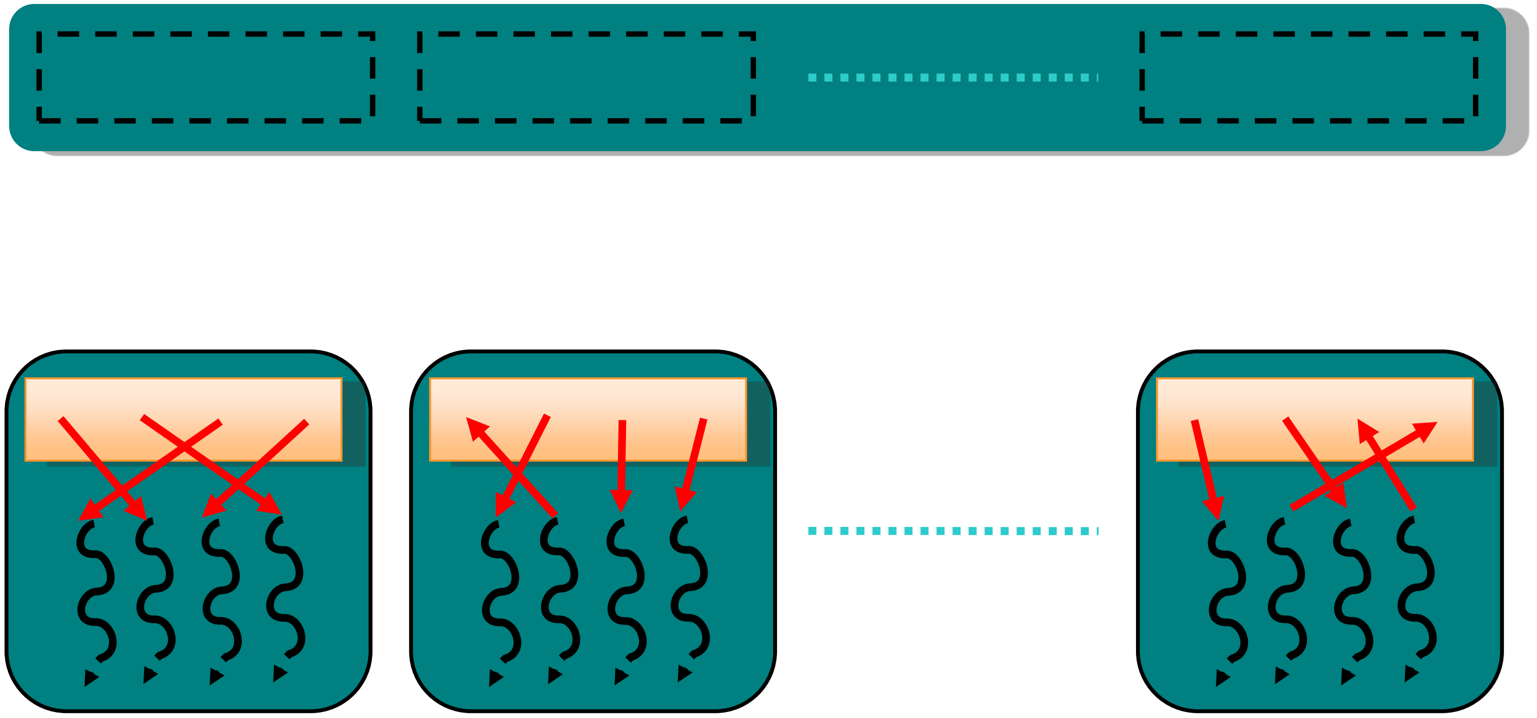
- Handle each data subset with one thread block

A Common Programming Strategy



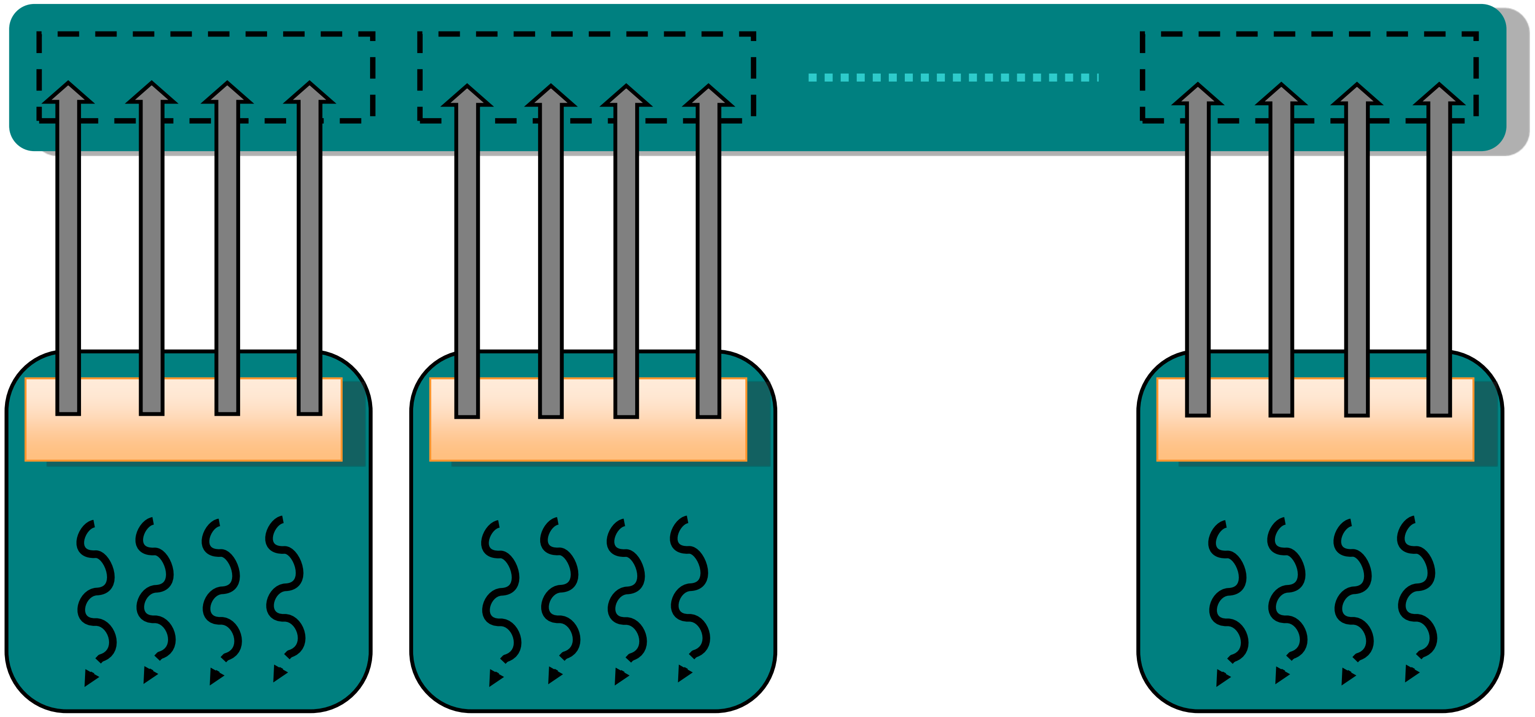
- Load the subset from device memory to shared memory, using multiple threads to exploit memory-level parallelism

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 device memory

CUDA: optimizing your application

Optimizing Occupancy

Thread Scheduling

- SM implements zero-overhead warp scheduling
  - A warp is a group of 32 threads that runs concurrently on a SM
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its inputs ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected

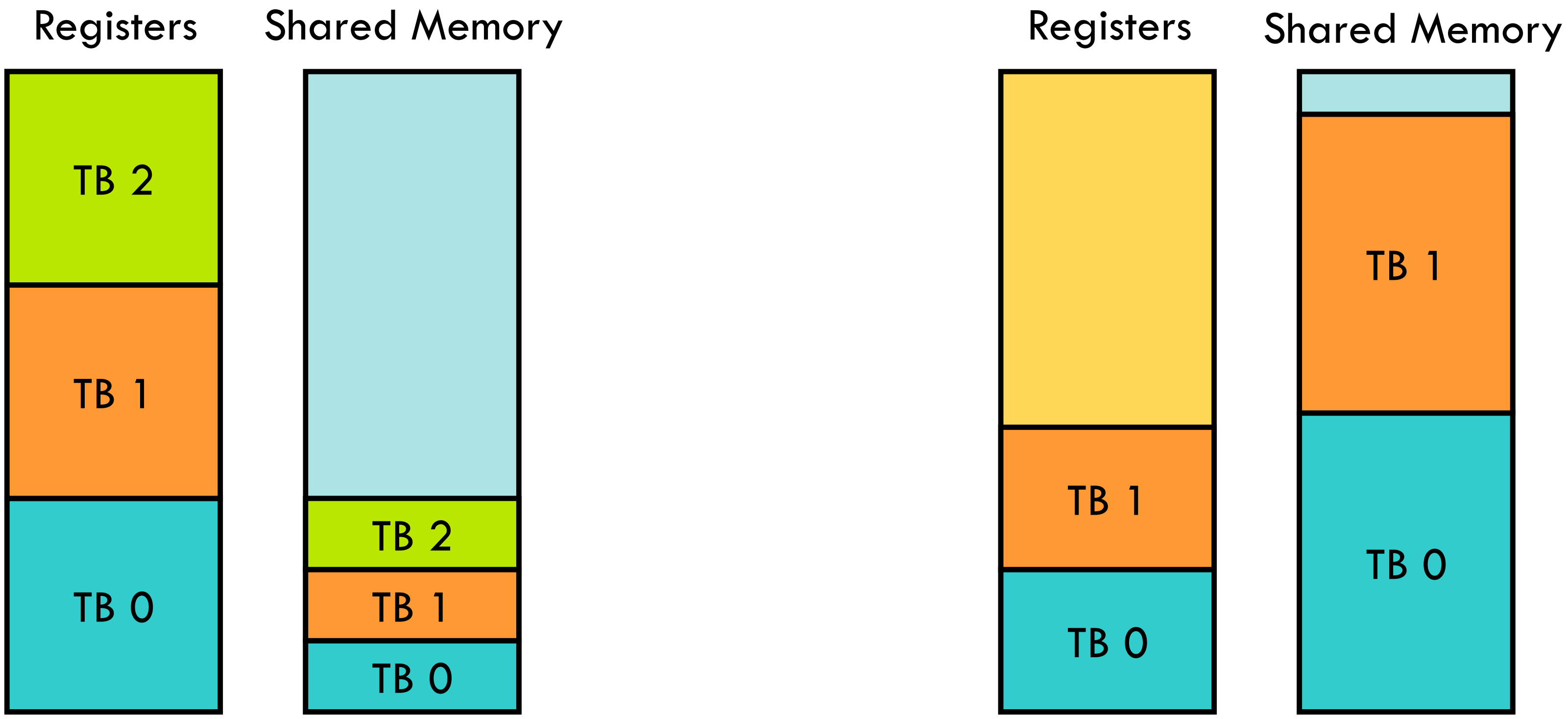
Stalling warps

- What happens if all warps are stalled?
  - No instruction issued → performance lost
- Most common reason for stalling?
  - Waiting on global memory
- If your code reads global memory every couple of instructions
  - You should try to maximize occupancy

Occupancy

- What determines occupancy?
- Limited resources!
  - Register usage per thread
  - Shared memory per thread block

Resource Limits (1)

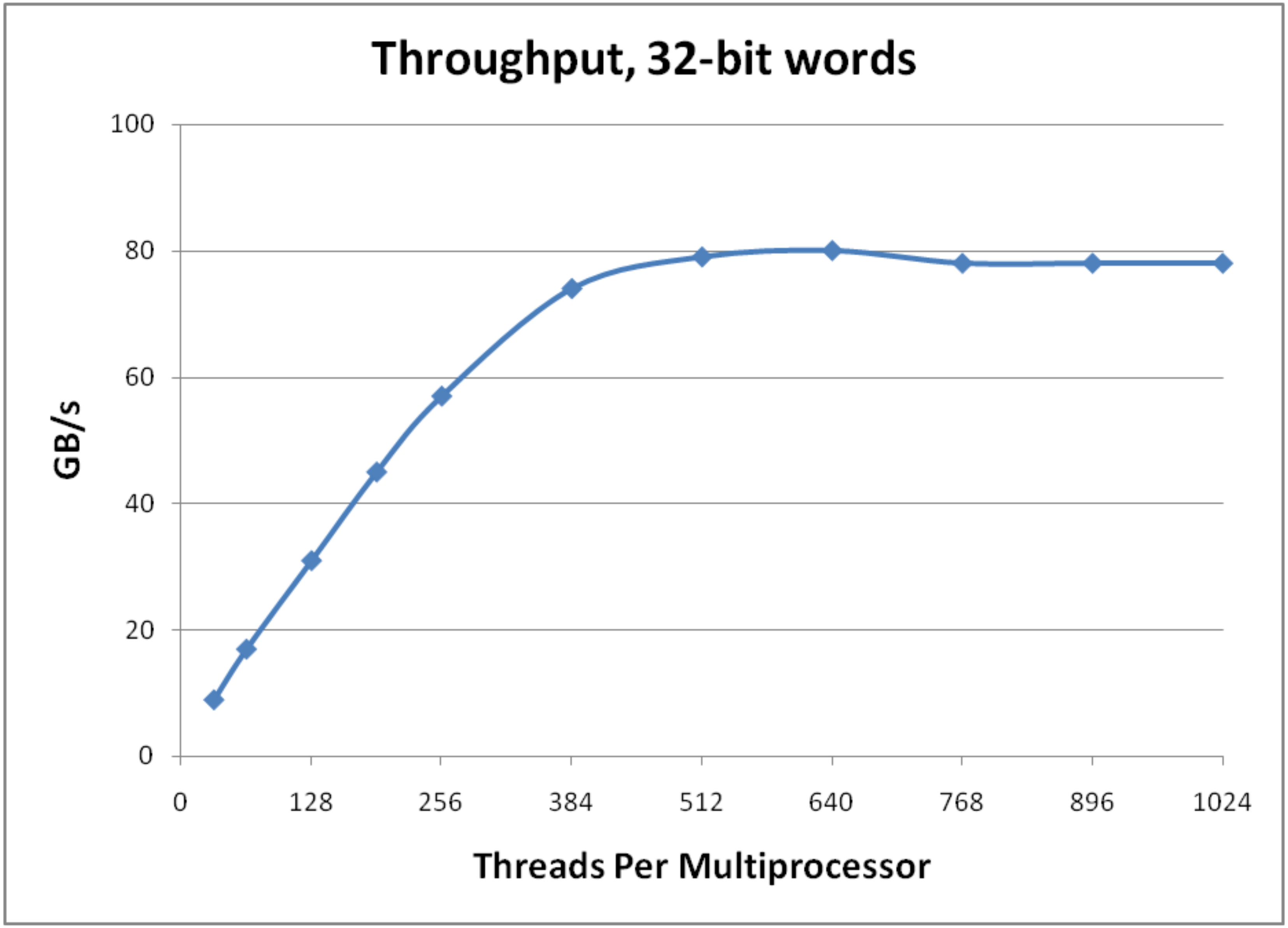


- Pool of registers and shared memory per SM
  - Each thread block grabs registers & shared memory
  - If one or the other is fully utilized ➡ no more thread blocks

Resource Limits (2)

- Can only have a limited number of blocks per SM
  - If they're too small, can't fill up the SM
- Higher occupancy has diminishing returns for hiding latency

Hiding Latency with more threads



How do you know what you're using?

- Use " `--ptxas-options="-v"`" to get register and shared memory usage

- You can plug those numbers into CUDA Occupancy Calculator

cmem[0]:kernel arguments
cmem[3] :user defined constant objects
cmem[16]:compiler generated constants (some of which may correspond to literal constants in the source code)

CUDA: optimizing your application

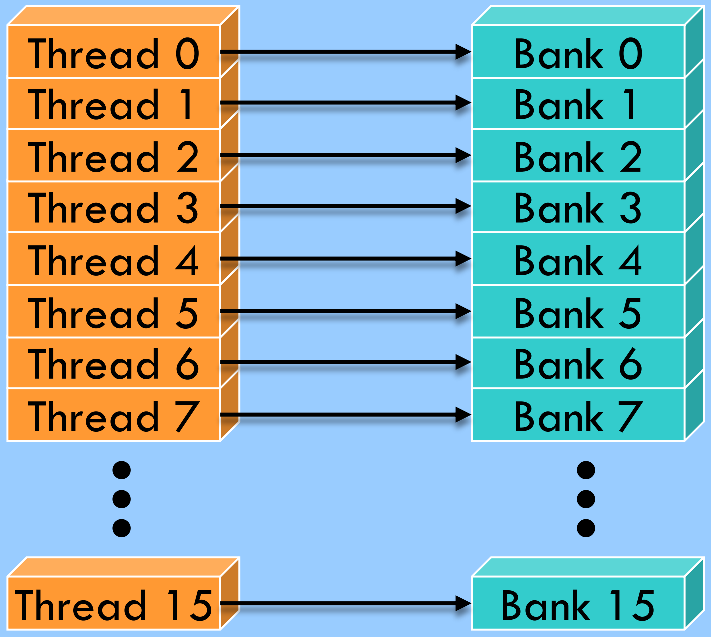
Shared memory bank conflicts

Shared Memory Banks

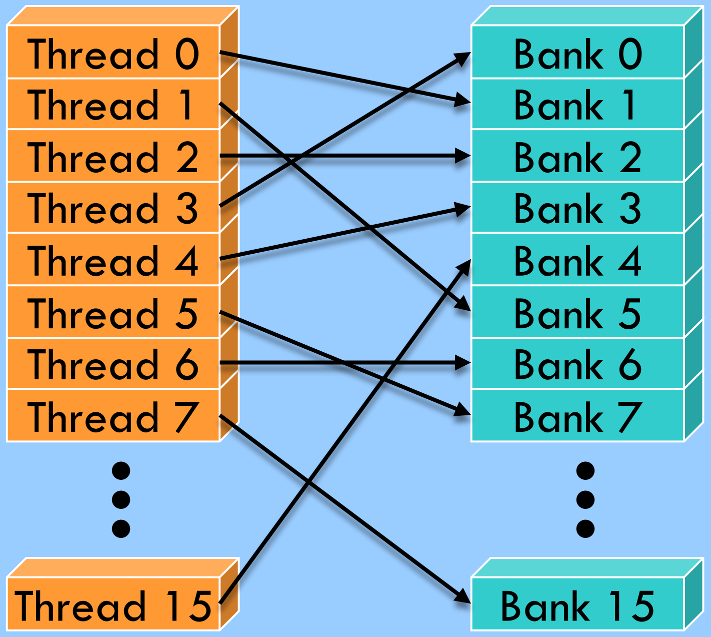
- Shared memory is banked
  - Only matters for threads within a warp
  - Full performance with some restrictions
    - Threads can each access different banks
    - Or can all access the same value
- Consecutive words are in different banks
- If two or more threads access the same bank but different value, we get bank conflicts

Bank Addressing Examples: OK

- **No Bank Conflicts**

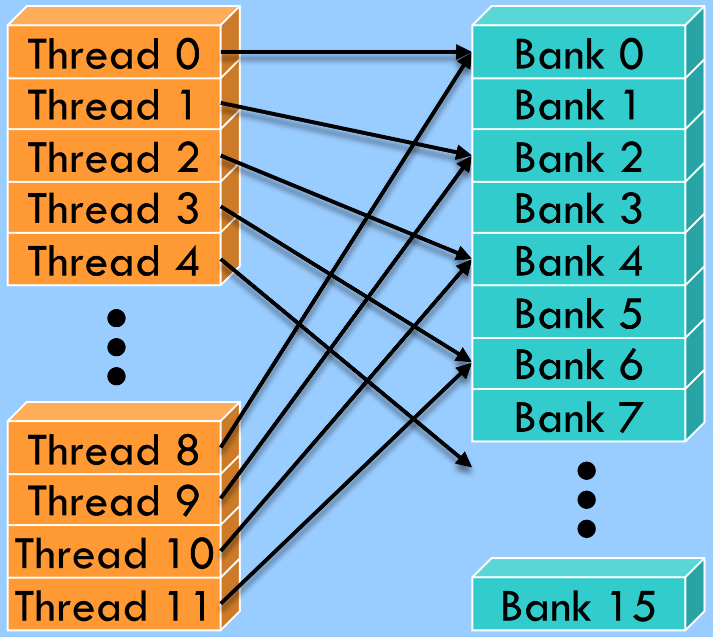


- **No Bank Conflicts**

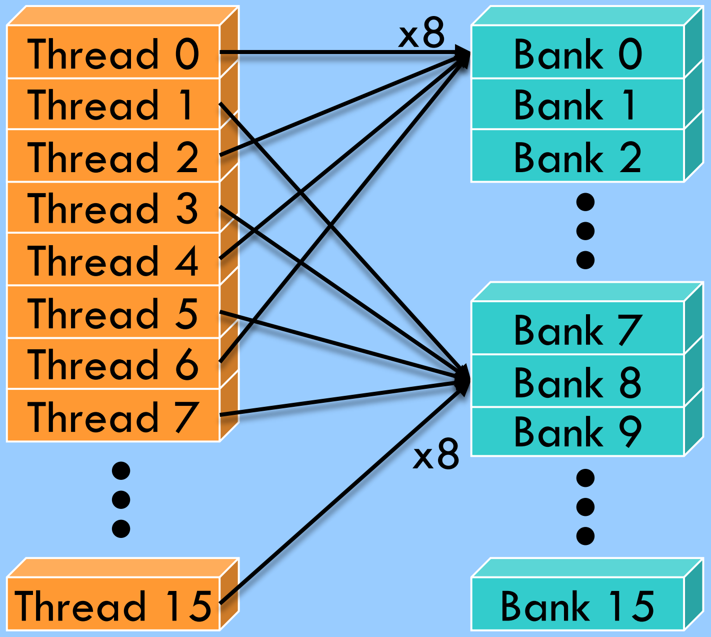


Bank Addressing Examples: BAD

- **2-way Bank Conflicts**



- **8-way Bank Conflicts**



Trick to Assess Performance Impact

- Change all shared memory reads to the same value
- All broadcasts = no conflicts
- Will show how much performance could be improved by eliminating bank conflicts

- The same doesn't work for shared memory writes
  - So, replace shared memory array indices with **`threadIdx.x`**
  - (Could also be done for the reads)

Other useful tools

cuda-memcheck
nvprof

Summary and conclusions

- Higher performance cannot be reached by increasing clock frequencies anymore
- Solution: introduction of large-scale parallelism
- Multiple cores on a chip
  - Today:
  - Up to 100 CPU cores in a node
  - Up to 5000 cores on a single GPU
  - Host system can contain multiple GPUs: 10,000+ cores
  - We can build clusters of these nodes!
- Future: 100,000s – millions of cores?

Summary and conclusions

- Many different types of many-core hardware
- Very different properties
  - Performance
  - Programmability
  - Portability
- It's all about the memory
- Choose the right platform for your application
  - Arithmetic intensity / Operational intensity
  - Roofline model