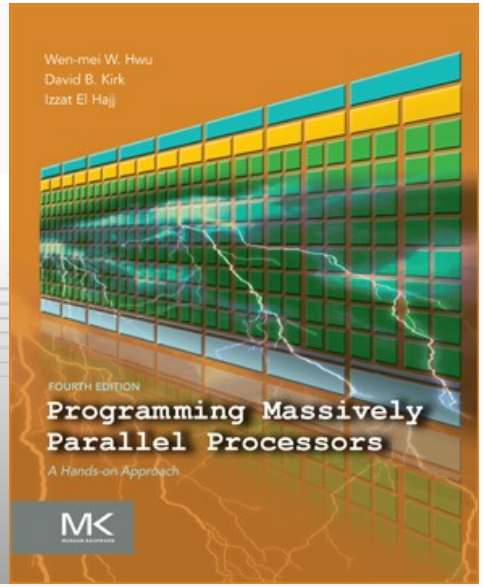
MORGAN KAUFMANN

Programming Massively Parallel Processors

A Hands-on Approach

CHAPTER 6 › Performance Considerations

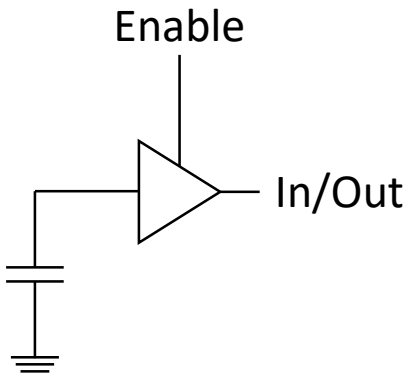


Performance Optimizations

- Performance optimizations covered so far:
  - Tuning resource usage to maximize occupancy
    - Threads per block, shared memory per block, registers per thread
  - Minimizing control divergence to increase SIMD efficiency
  - Shared memory tiling to increase data reuse

- More optimizations covered today:
  - Memory coalescing
  - Thread coarsening

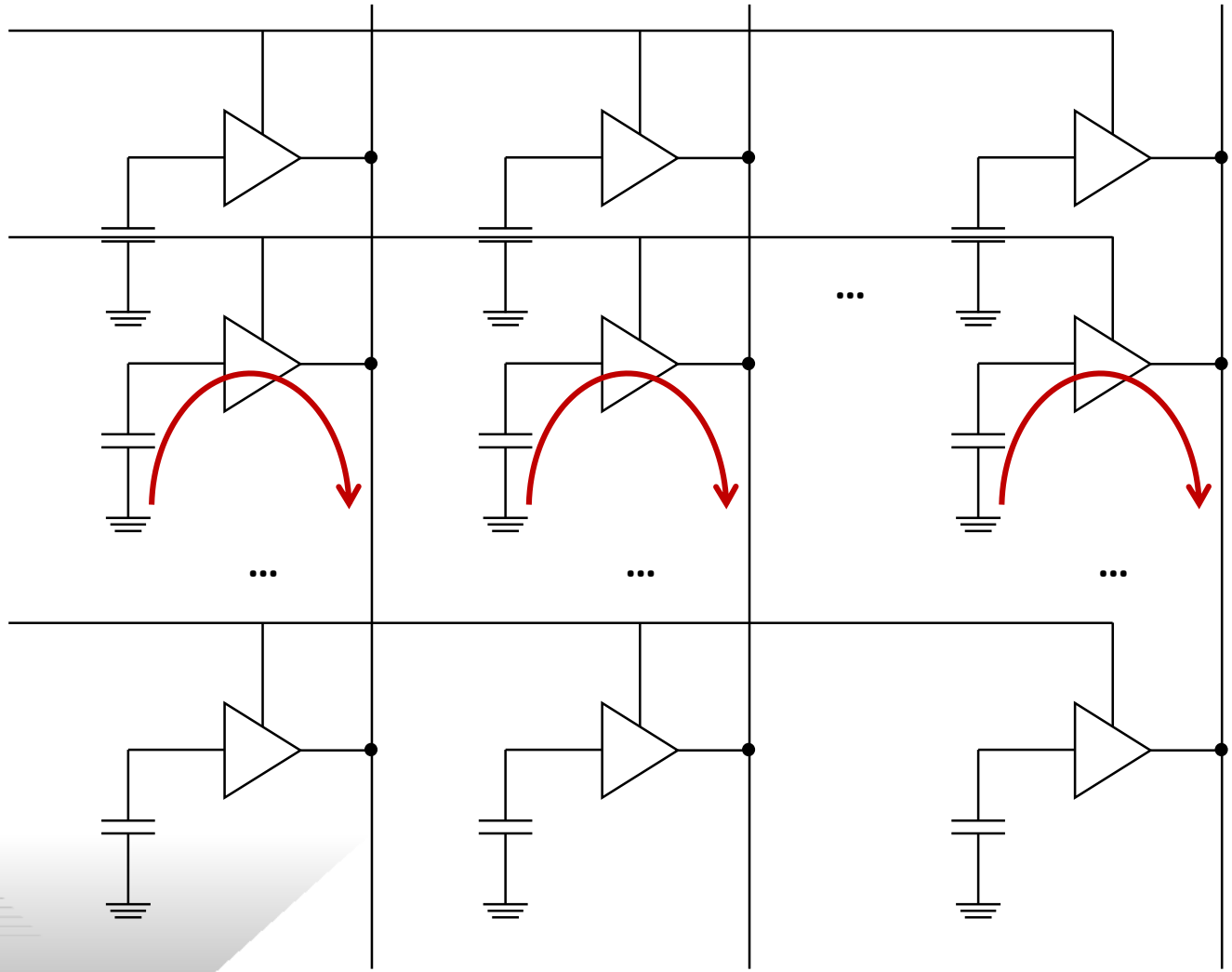
DRAM Cell



A DRAM cell consists of a capacitance that stores a charge and a three-state device (single transistor) that allows data to be read/written (capacitor discharge/charged)

DRAM Array

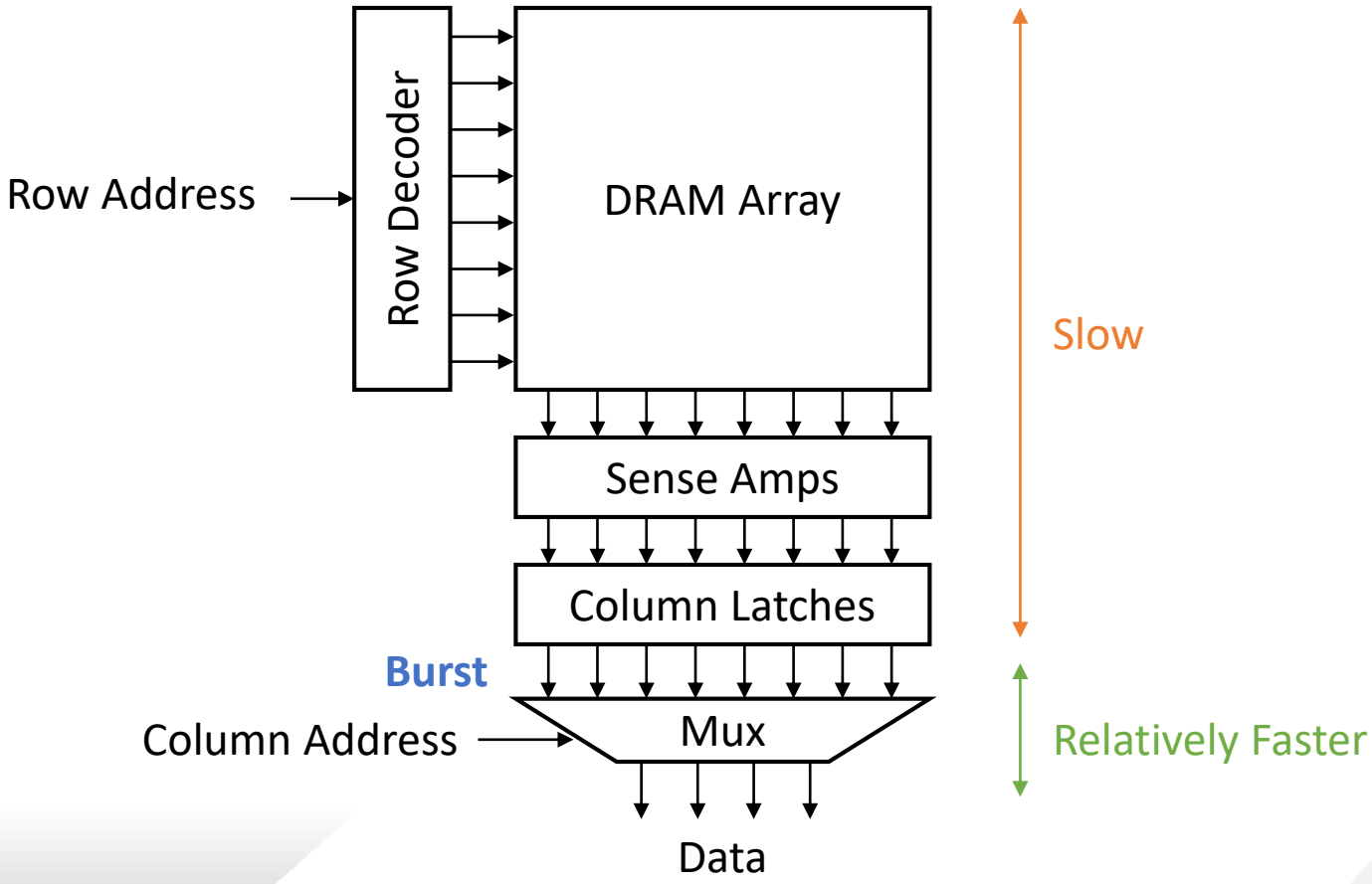
activate row



read one bit from each column

A DRAM bank consists of a 2D array of DRAM cells activated one row at a time, and read at the column

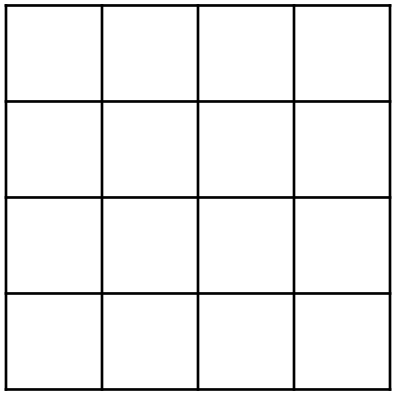
DRAM Bank



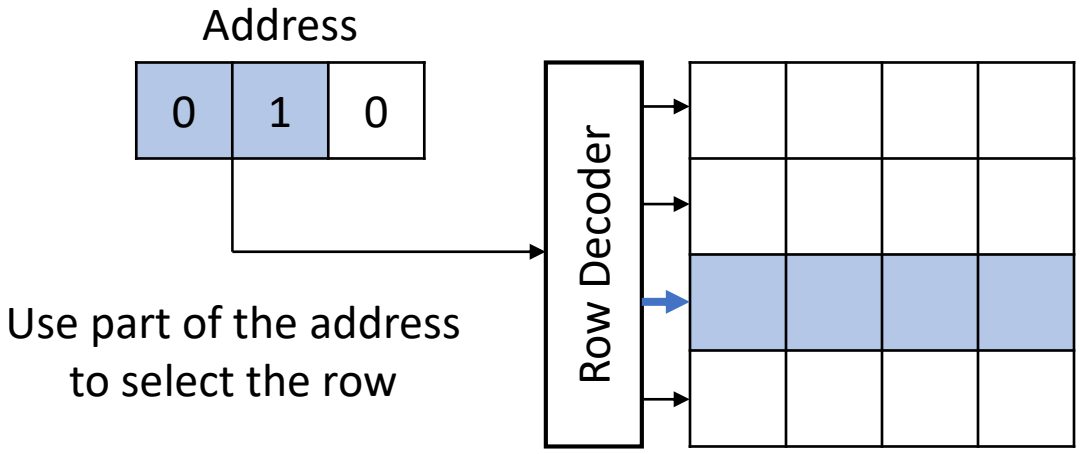
DRAM Bank Example

Address

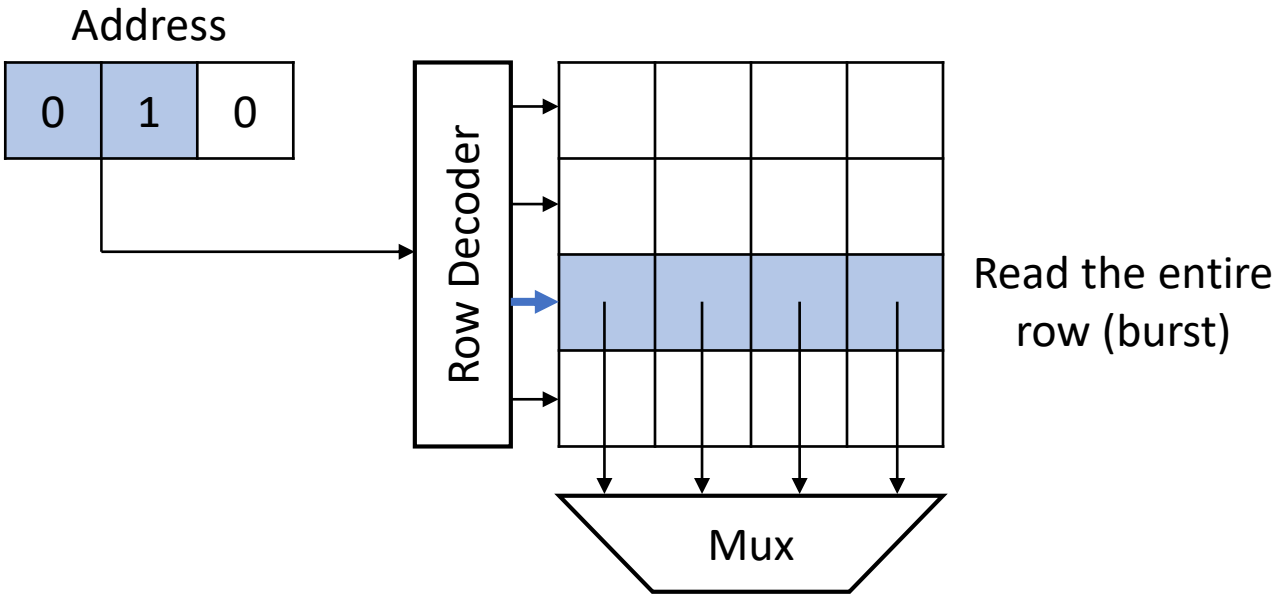
0	1	0



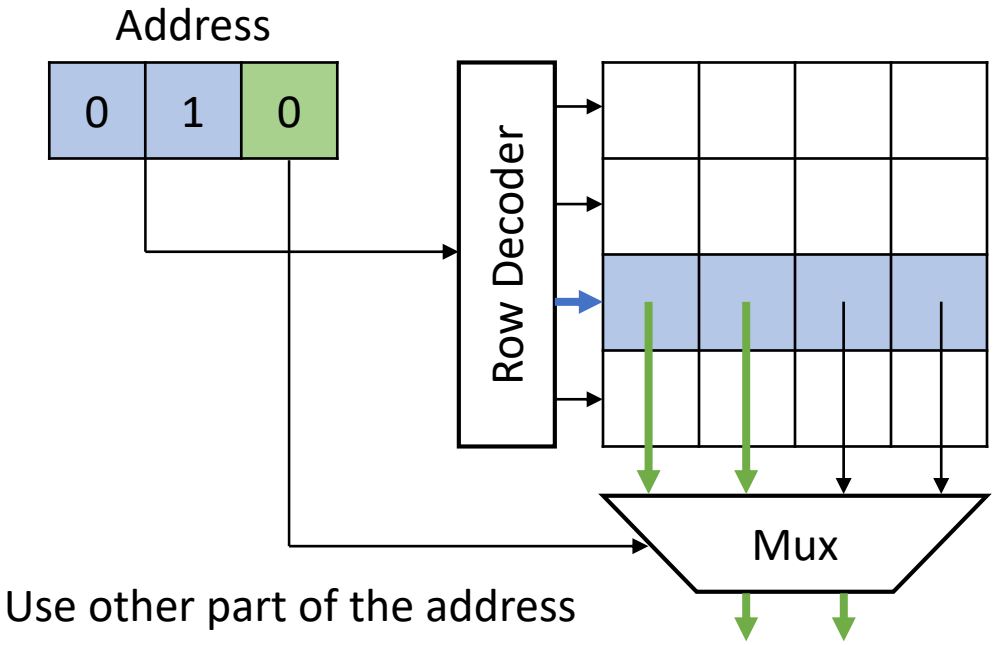
DRAM Bank Example



DRAM Bank Example

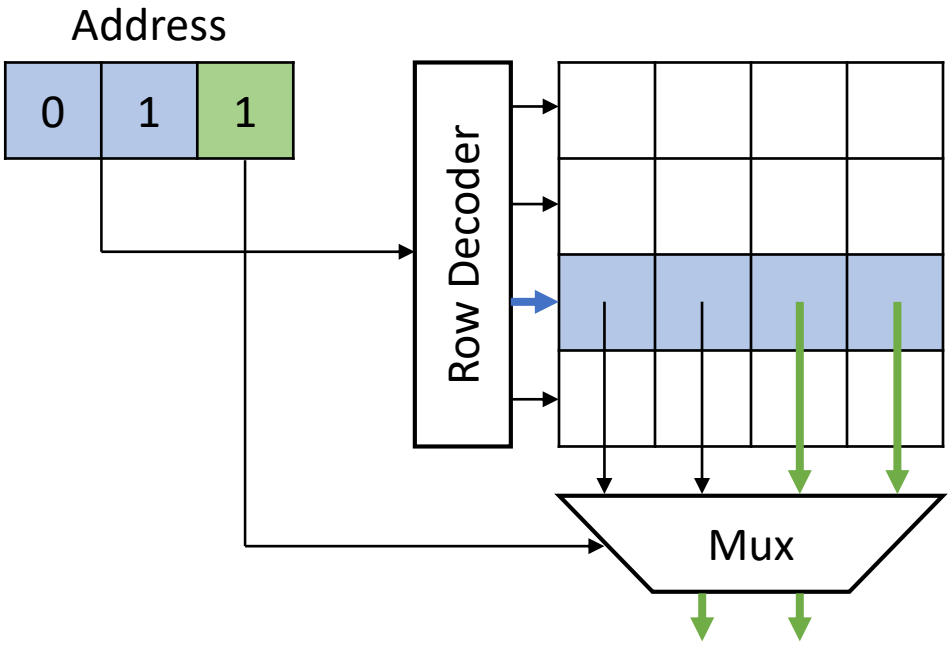


DRAM Bank Example



Use other part of the address to select columns within row

DRAM Bank Example



If other access from the same burst, no need to read the row again

DRAM Burst

- Accessing data in different bursts
  - Need to access the array again

Timeline: 

- Accessing data in the same burst
  - No need to access the array again, just the multiplexer

Timeline: 

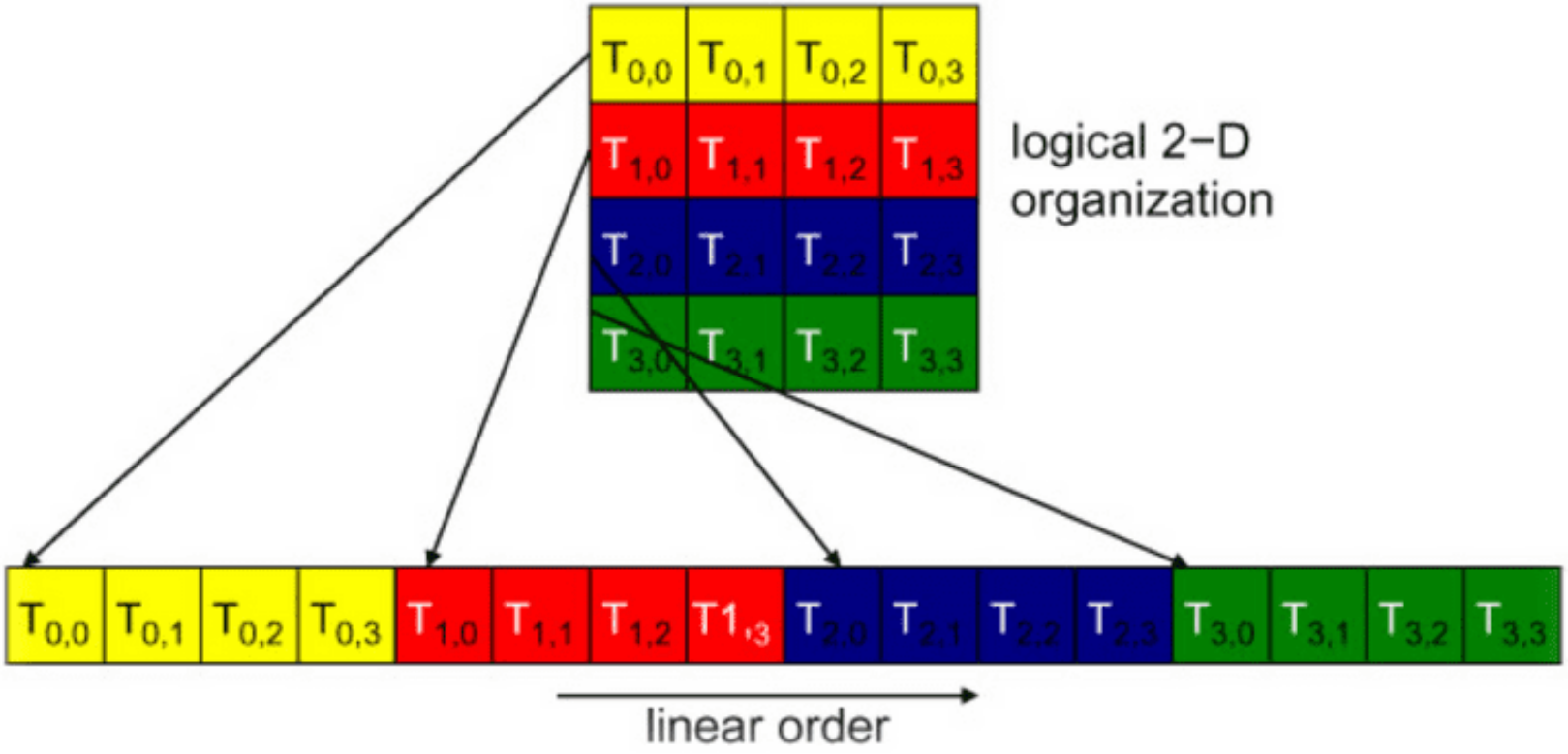
- Accessing data in the same burst is faster than accessing data in different bursts

Memory Coalescing

- When threads in the same warp access consecutive memory locations in the same burst, the accesses can be combined and served by one burst
  - One DRAM transaction is needed
  - Known as **memory coalescing**

- If threads in the same warp access locations not in the same burst, accesses cannot be combined
  - Multiple transactions are needed
  - Takes longer to service data to the warp
  - Sometimes called **memory divergence**

Memory Coalescing



Memory Coalescing Examples

- Vector addition:

```
int i = blockDim.x*blockIdx.x + threadIdx.x;
z[i] = x[i] + y[i];
```

  - Accesses to `x`, `y`, and `z` are coalesced
    - e.g., threads 0 to 31 access elements 0 to 31, resulting in one memory transaction to service warp 0

- Matrix-matrix multiplication:

```
int row = blockDim.y*blockIdx.y + threadIdx.y;
int col = blockDim.x*blockIdx.x + threadIdx.x;
for(unsigned int i = 0; i < N; ++i) {
    sum += A[row*N + i]*B[i*N + col];
}
```

Note: a warp contains consecutive threads in the x dimension followed by the y dimension

  - Accesses to `A` and `B` are coalesced
    - e.g., threads 0 to 31 access element 0 of `A` on the first iteration, resulting in one memory transaction to service warp 0
    - e.g., threads 0 to 31 access elements 0 to 31 of `B` on the first iteration, resulting in one memory transaction to service warp 0

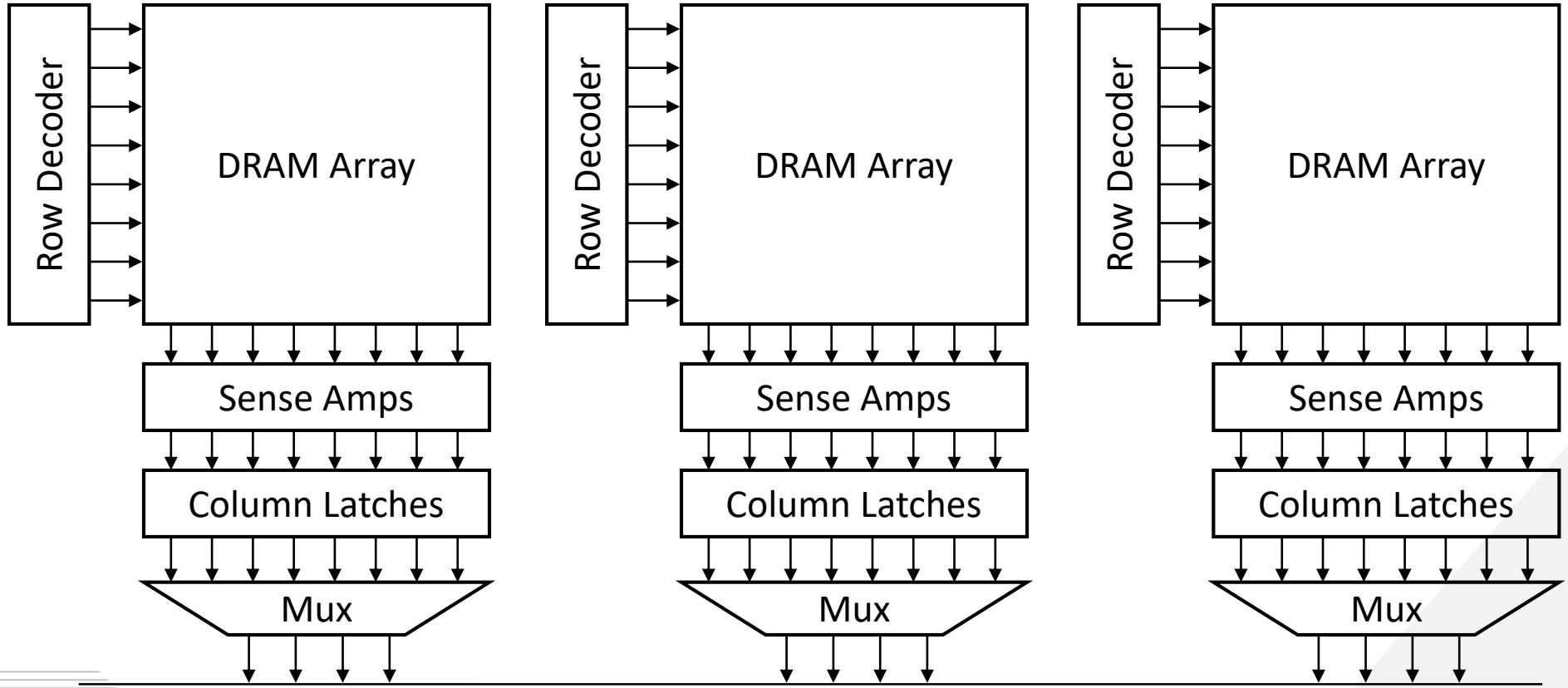
Memory Coalescing Examples

- Tiled matrix-matrix multiplication:

```
unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
unsigned int col = blockIdx.x*blockDim.x + threadIdx.x;
for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {
    A_s[threadIdx.y][threadIdx.x] =
                A[row*N + tile*TILE_DIM + threadIdx.x];
    B_s[threadIdx.y][threadIdx.x] =
                B[(tile*TILE_DIM + threadIdx.y)*N + col];
    ...
}
```

  - Accesses to `A` and `B` are coalesced
    - e.g., threads 0 to 31 access elements 0 to 31 of `A` on the first iteration, resulting in one memory transaction to service warp 0
      - Even better than without tiling where all threads accessed one element, underutilizing the burst
    - e.g., threads 0 to 31 access elements 0 to 31 of `B` on the first iteration, resulting in one memory transaction to service warp 0

Multiple DRAM Banks



Latency Hiding with Multiple Banks

- With one bank, time still wasted in between bursts

- Latency can be hidden by having multiple banks

- Need many threads to simultaneously access memory to keep all banks busy
  - Achieved with having high occupancy in SMs
    - Similar idea to hiding pipeline latency in the core

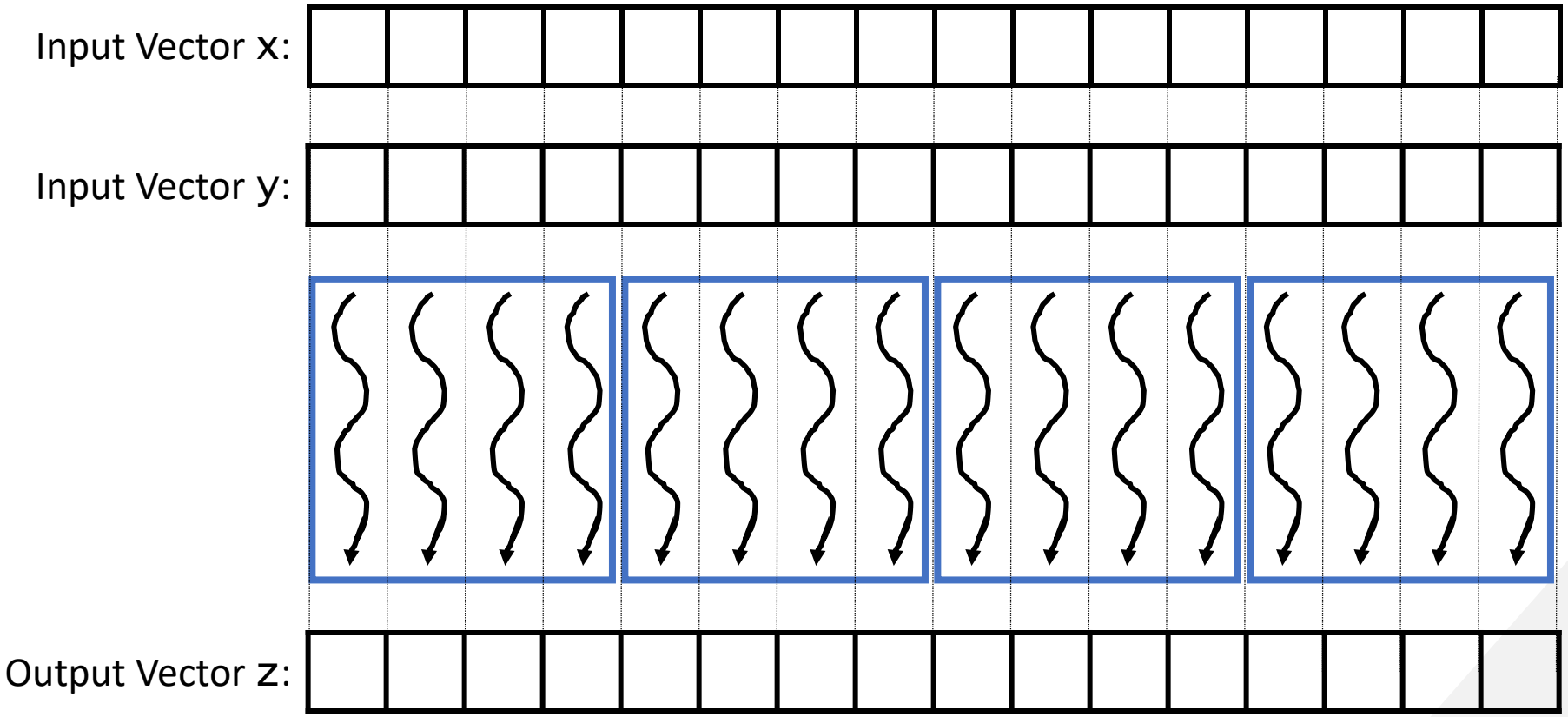
Fine-Grain Thread Granularity

- So far, parallelization approaches made threads as *fine-grain* as possible
  - Assign smallest possible unit of parallelism per thread
    - e.g., one vector element per thread in vector addition
    - e.g., one output pixel per thread in RGB to gray and in blur
    - e.g., one output matrix element per thread in matrix-matrix multiplication

- Advantage: provide hardware with as many threads as possible to fully utilize resources
  - If more threads provided than GPU can support, hardware can serialize the work with low overhead
  - If future GPUs come out with more resources, they can extract more parallelism without code being rewritten
    - Recall: *transparent scalability*

- Disadvantage: if there is a "price" for parallelizing work across more threads, this price is maximized
  - Okay if threads actually run in parallel
  - Suboptimal if threads are getting serialized by hardware

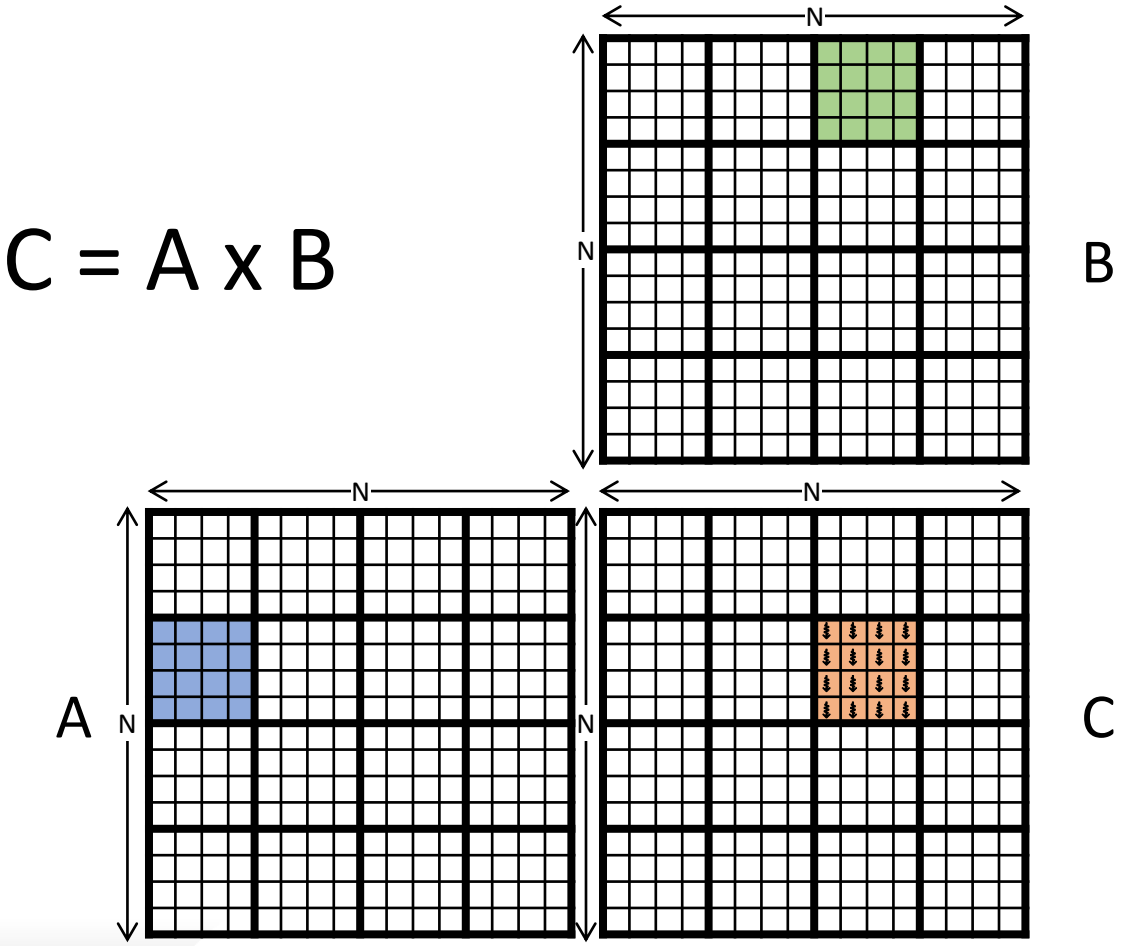
Recall: Vector Addition



What price was paid to parallelize work across more threads?

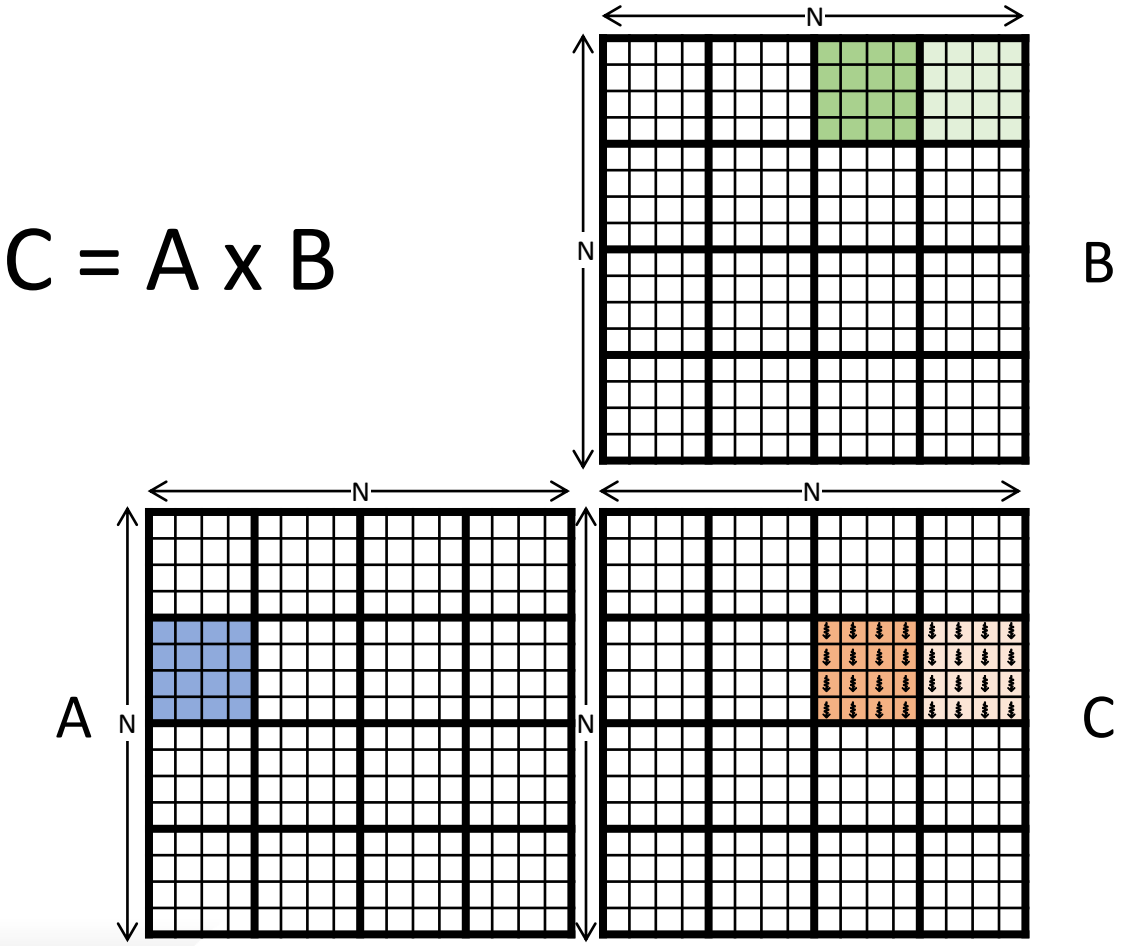
None. Making threads as fine-grain as possible is a good approach.

Recall: Tiled Matrix-Matrix Multiplication



What price was paid to parallelize work across more threads?

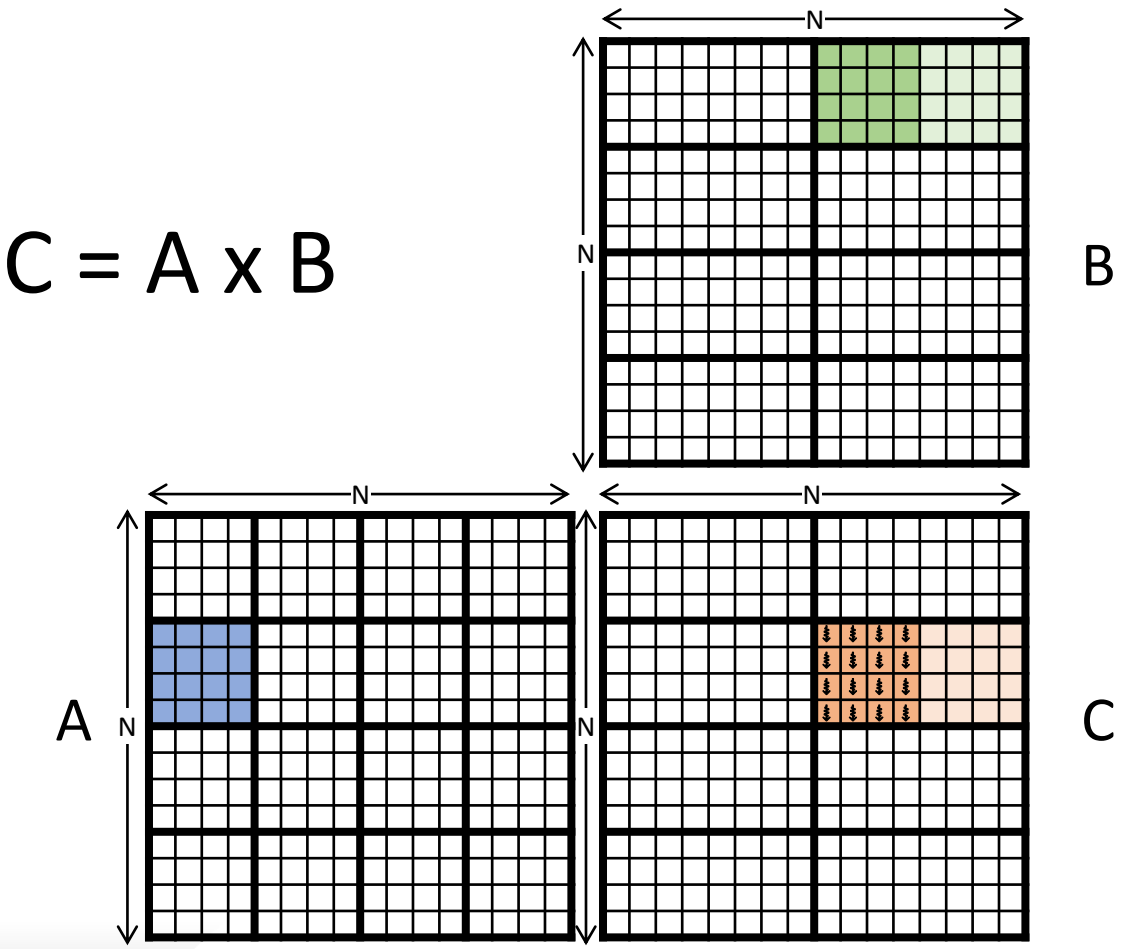
Recall: Tiled Matrix-Matrix Multiplication



What price was paid to parallelize work across more threads?

Thread blocks processing horizontally adjacent output tiles of C redundantly load the same input tile of A

Optimizing Tiled Matrix-Matrix Multiplication



Optimization: Have one thread block process multiple output tiles sequentially and reuse the input tile that it loaded

Example: Tiled Matrix-Matrix Multiplication

```
__global__ void mm_tiled_coarse_kernel(float* A, float* B, float* C, unsigned int M,
                                       unsigned int N, unsigned int K) {
    __shared__ float A_s[TILE_DIM][TILE_DIM];
    __shared__ float B_s[TILE_DIM][TILE_DIM];
    unsigned int row = blockIdx.y*blockDim.y + threadIdx.y;
    unsigned int colStart = blockIdx.x*blockDim.x*COARSE_FACTOR + threadIdx.x;
    float sum[COARSE_FACTOR];
    for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
        sum[c] = 0.0f;
    }
    for(unsigned int tile = 0; tile < N/TILE_DIM; ++tile) {
        // Load A tile
        A_s[threadIdx.y][threadIdx.x] = A[row*N + tile*TILE_DIM + threadIdx.x];
        for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
            unsigned int col = colStart + c*TILE_DIM;
            // Load B tile
            B_s[threadIdx.y][threadIdx.x] = B[(tile*TILE_DIM + threadIdx.y)*N + col];
            __syncthreads();
            // Compute with tile
            for(unsigned int i = 0; i < TILE_DIM; ++i) {
                sum[c] += A_s[threadIdx.y][i]*B_s[i][threadIdx.x];
            }
            __syncthreads();
        }
    }
    for(unsigned int c = 0; c < COARSE_FACTOR; ++c) {
        unsigned int col = colStart + c*TILE_DIM;
        C[row*N + col] = sum[c];
    }
}
```

Thread responsible for multiple output elements

Processing for each output element is serialized with a *coarsening loop*

```
dim3 numBlocks((N + TILE_DIM - 1)/TILE_DIM/COARSE_FACTOR, (N + TILE_DIM - 1)/TILE_DIM);
```

Thread Coarsening

- **Thread coarsening** is an optimization were a thread is assigned multiple units of parallelism to process
  - i.e., a thread is made more *coarse-grain*
- Advantages:
  - Reduces the "price" paid for parallelization
    - Redundant memory loads in the tiled matrix multiplication
    - Could be redundant computations in other examples
    - Could be synchronization overhead or divergence (covered later)
- Disadvantages:
  - Underutilizes resources if *coarsening factor* is too high
    - Need to retune coarsening factor for each device
  - More resources per thread which may limit occupancy

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (minimizing idle cores during SIMD execution)	-	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (minimizing idle cores during SIMD execution)	-	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (minimizing idle cores during SIMD execution)	-	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data then updating the universal copy when done

Checklist of Common Optimizations

Optimization	Benefit to Compute Cores	Benefit to Memory	Strategies
Maximizing occupancy	More work to hide pipeline latency	More parallel memory accesses to hide DRAM latency	Tuning usage of SM resources such as threads per block, shared memory per block, and registers per thread
Enabling coalesced global memory accesses	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic and better utilization of bursts/cache-lines	Transfer between global memory and shared memory in a coalesced manner and performing un-coalesced accesses in shared memory (e.g., corner turning) Rearranging the mapping of threads to data Rearranging the layout of the data
Minimizing control divergence	High SIMD efficiency (minimizing idle cores during SIMD execution)	-	Rearranging the mapping of threads to work and/or data Rearranging the layout of the data
Tiling of reused data	Fewer pipeline stalls waiting for global memory accesses	Less global memory traffic	Placing data that is reused within a block in shared memory or registers so that it is transferred between global memory and the SM only once
Privatization (covered later)	Fewer pipeline stalls waiting for atomic updates	Less contention and serialization of atomic updates	Applying partial updates to a private copy of the data then updating the universal copy when done
Thread coarsening	Less redundant work, divergence, or synchronization	Less redundant global memory traffic	Assigning multiple units of parallelism to each thread in order to reduce the “price of parallelism” when it is incurred unnecessarily

Tensions Between Optimizations

- Maximizing occupancy
  - Maximizing occupancy hides pipeline latency, but too many threads may compete for the cache, evicting each others' data (*thrashing* the cache)
- Shared memory tiling
  - Using more shared memory enables more data reuse, but may limit occupancy
- Thread coarsening
  - Coarsening reduces parallelization overhead, but requires more resources per thread which may limit occupancy

Need to find the sweet spot that achieves the best compromise

Know Your Application's Bottleneck!

- The constraint that limits the performance of an application on a device is called a **bottleneck**
- The bottleneck depends on the application as well as the device itself
- Optimizations trade one resource for another to alleviate the bottleneck
- Make sure to properly diagnose your application's bottleneck before applying optimizations
  - Otherwise, you may be optimizing for the wrong resource

References

- Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2022.