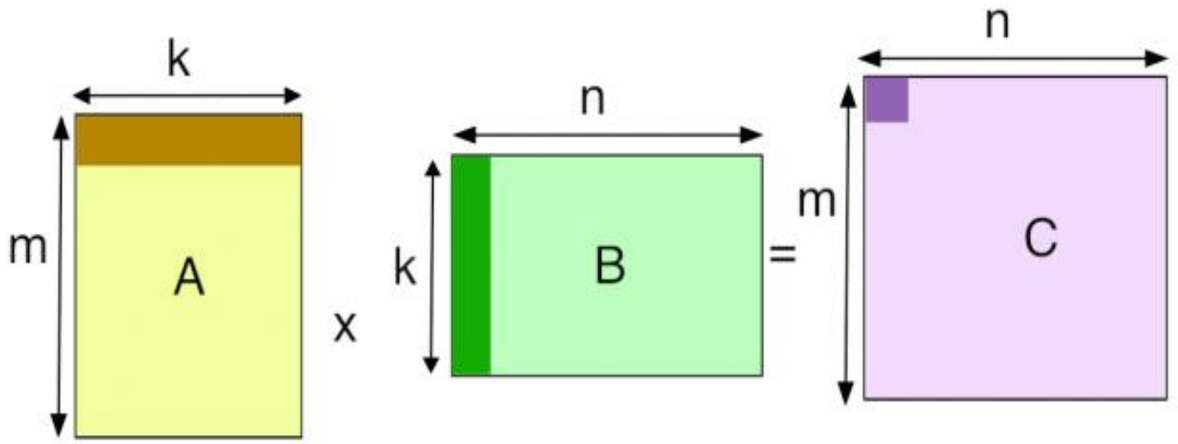
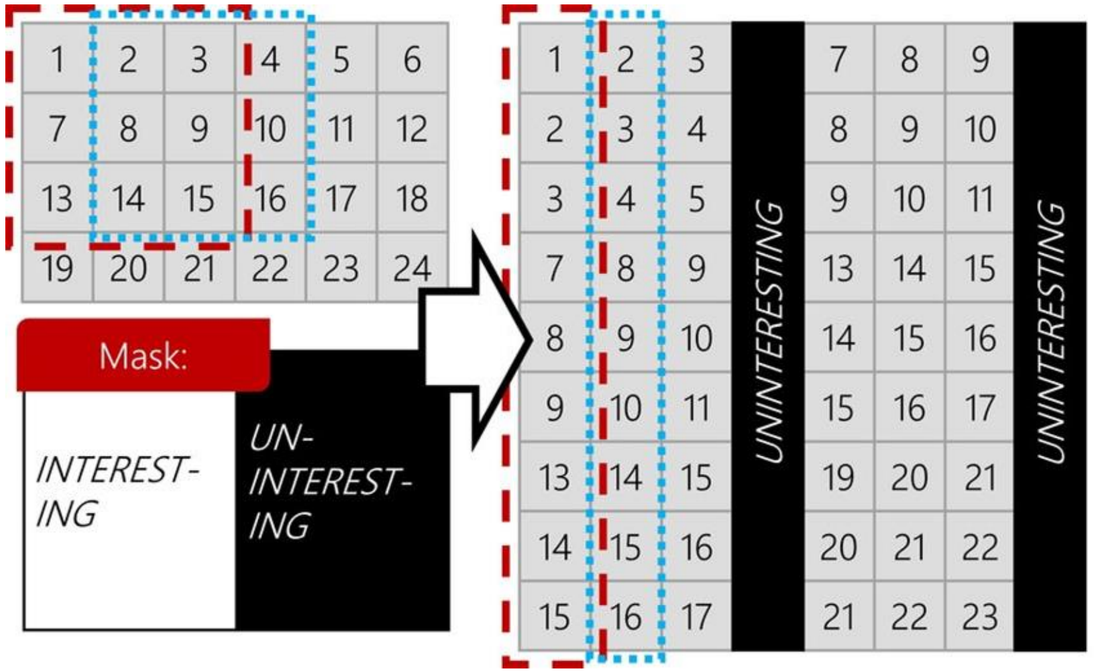
RICE UNIVERSITY

Week-2: GEMM Operation Optimization

Motivation: Why GEMM Matters

- GEMM dominates compute workloads in ML and scientific computing.
- Performance gains in GEMM directly impact model training and HPC efficiency.
- Optimized GEMM enables high GPU utilization and energy efficiency.





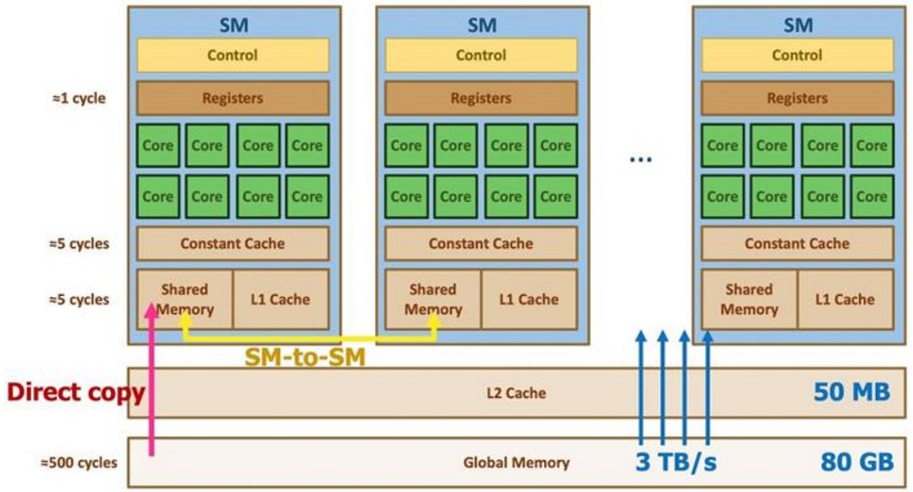
What is GEMM?

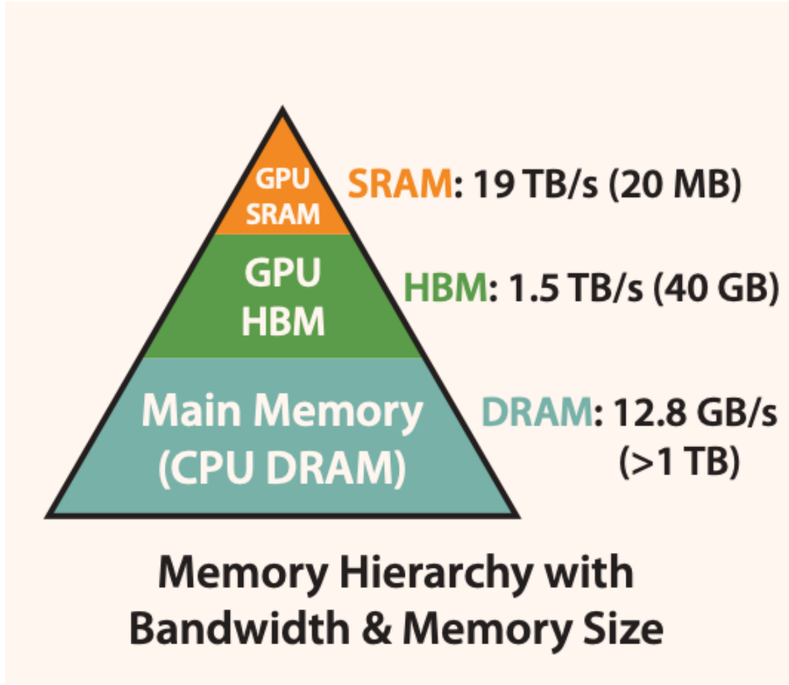
- Definition: C = αAB + βC (General Matrix-Matrix Multiplication).
- BLAS Level-3 routine, foundation for many numerical computations.
- Key operation in deep learning frameworks and HPC systems.

BLAS subprograms		Semantic	FP ops.	Mem. ops.	Ratio
Level 1	Vector Addition	$y_i = x_i + y_i$	n	$3n$	1:3
	Vector Scaling	$x_i = sx_i$	n	$2n$	1:2
	Dot Product	$s = \sum_{i=0}^{n-1} x_i y_i$	$2n$	$2n$	2:2
Level 2	Matrix-vector multiplication	$y_i = y_i + \sum_{j=0} a_{ij} x_j$	$2n^2+n$	n^2+3n	2:1
	Rank-one update	$a_{ij} = a_{ij} + x_i y_j$	$2n^2$	$2n^2+2n$	2:2
Level 3	Matrix-Matrix multiplication	$c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} b_{kj}$	$2n^3+n^2$	$4n^2$	n:2

GEMM and GPU Memory Hierarchy

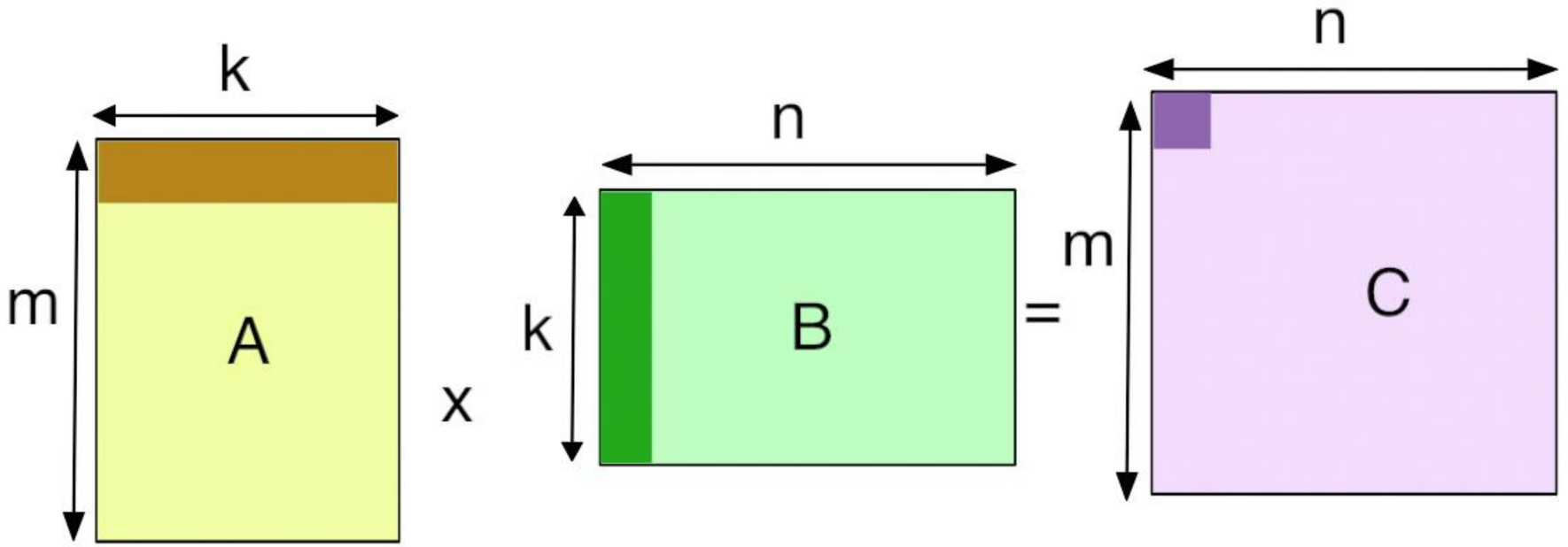
- GPU architecture includes multiple memory levels: global, shared, registers.
- Bottlenecks often stem from memory access rather than arithmetic.
- Optimization focuses on reuse and minimizing bandwidth waste.





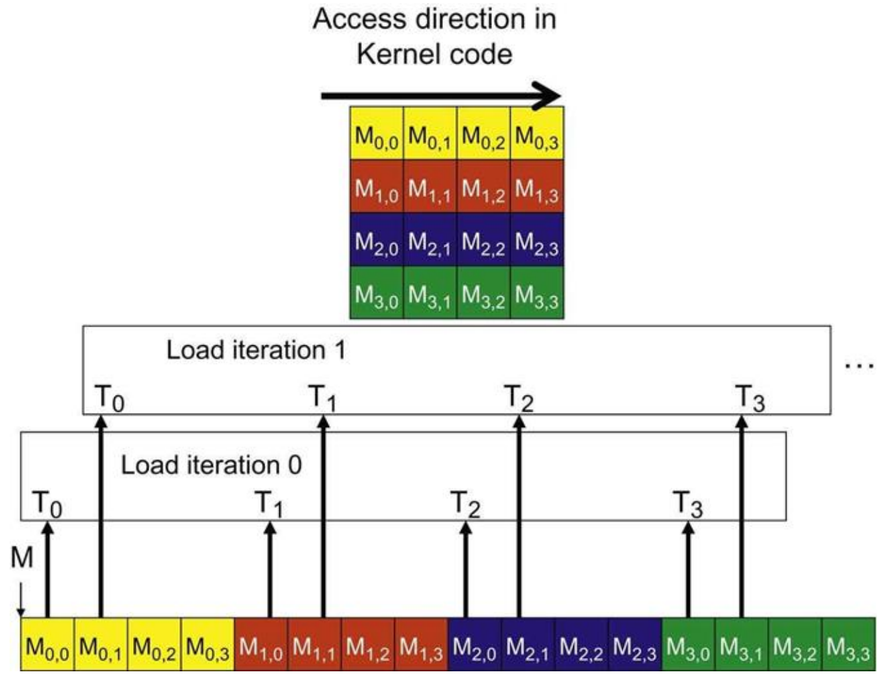
Naïve GEMM Kernel

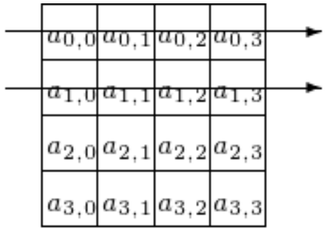
- Each thread computes one output element.
- Triple nested loop, redundant memory access from global memory.
- Performs only ~1–2% of cuBLAS throughput.

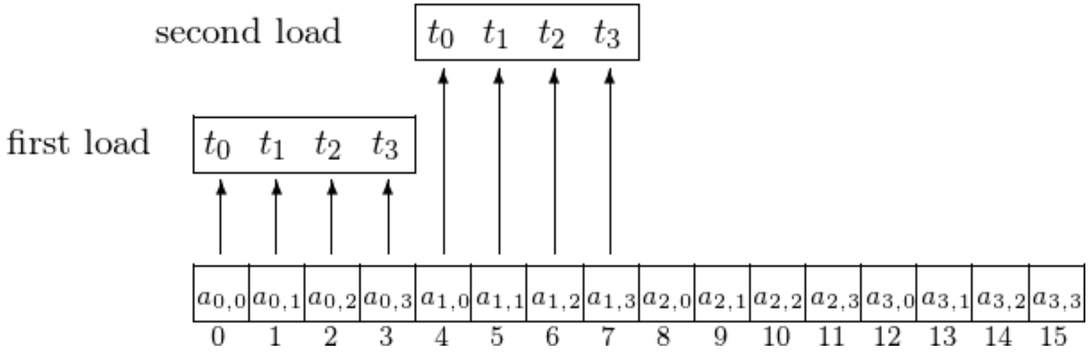


Memory Coalescing

- Threads should access contiguous memory addresses.
- Non-coalesced accesses reduce effective memory bandwidth.
- Align and batch memory loads (vectorized access).

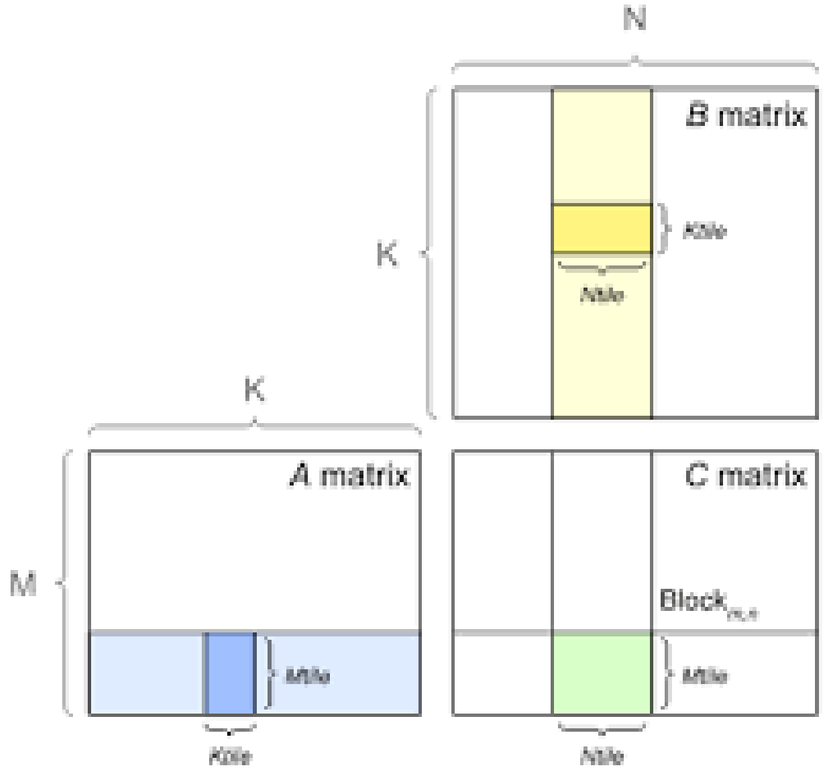


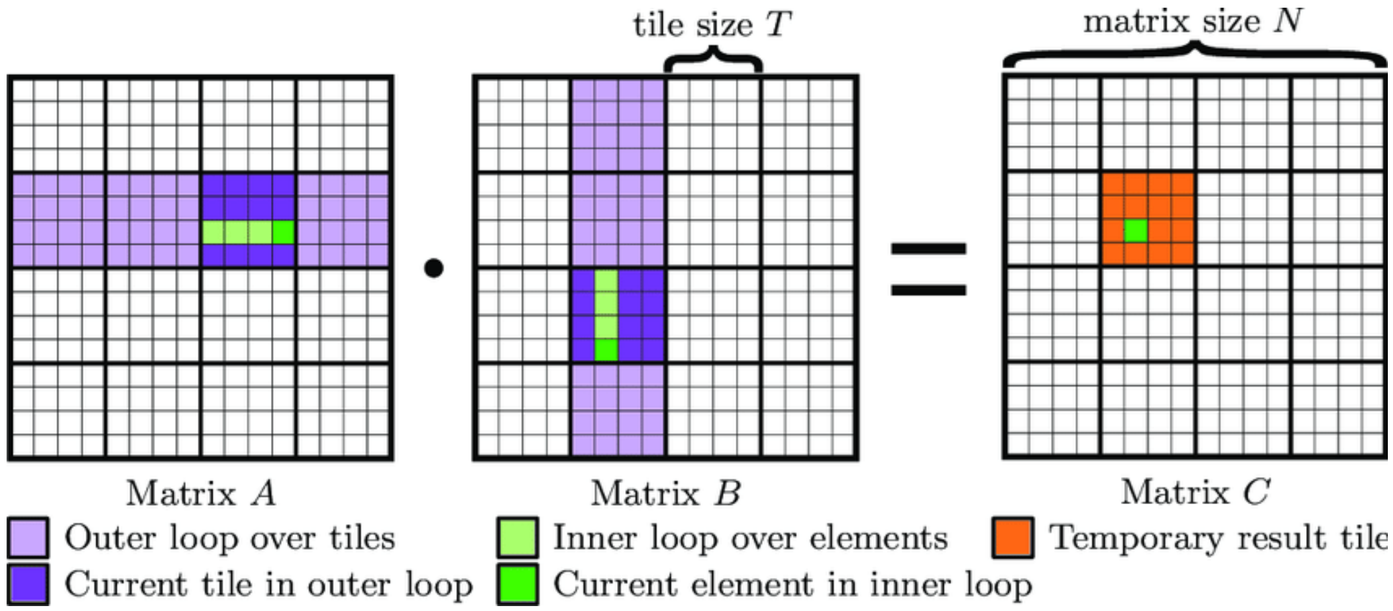




Tiling and Blocking

- Load sub-matrices (tiles) into shared memory for reuse.
- Each block computes a tile of the output matrix.
- Improves data locality and arithmetic intensity.

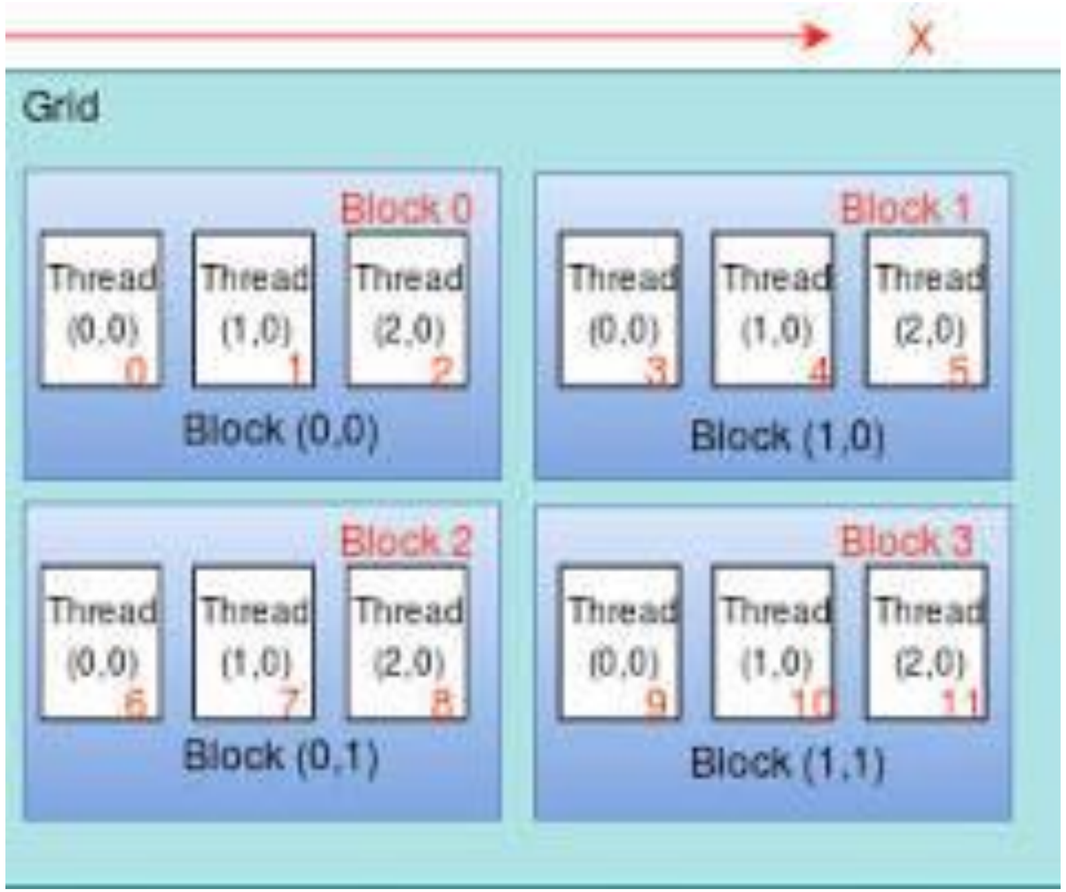




Indexing Calculation

GPU 2D thread indexing in CUDA maps a 2D matrix or image grid to parallel threads, typically calculating unique row (`y`) and column (`x`) indices using `blockIdx` , `blockDim` , and `threadIdx` . The standard formula to calculate global column (x) and row (y) indices for a thread is:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```



Key Components and Formulas:

- **Unique 2D ID:** `(x, y)` coordinates are generally computed as:
  - `x = blockIdx.x * blockDim.x + threadIdx.x`
  - `y = blockIdx.y * blockDim.y + threadIdx.y`
- **Linear Memory Index:** To access 1D linear memory (like a C array) from 2D threads, use the formula: `index = y * width + x` .

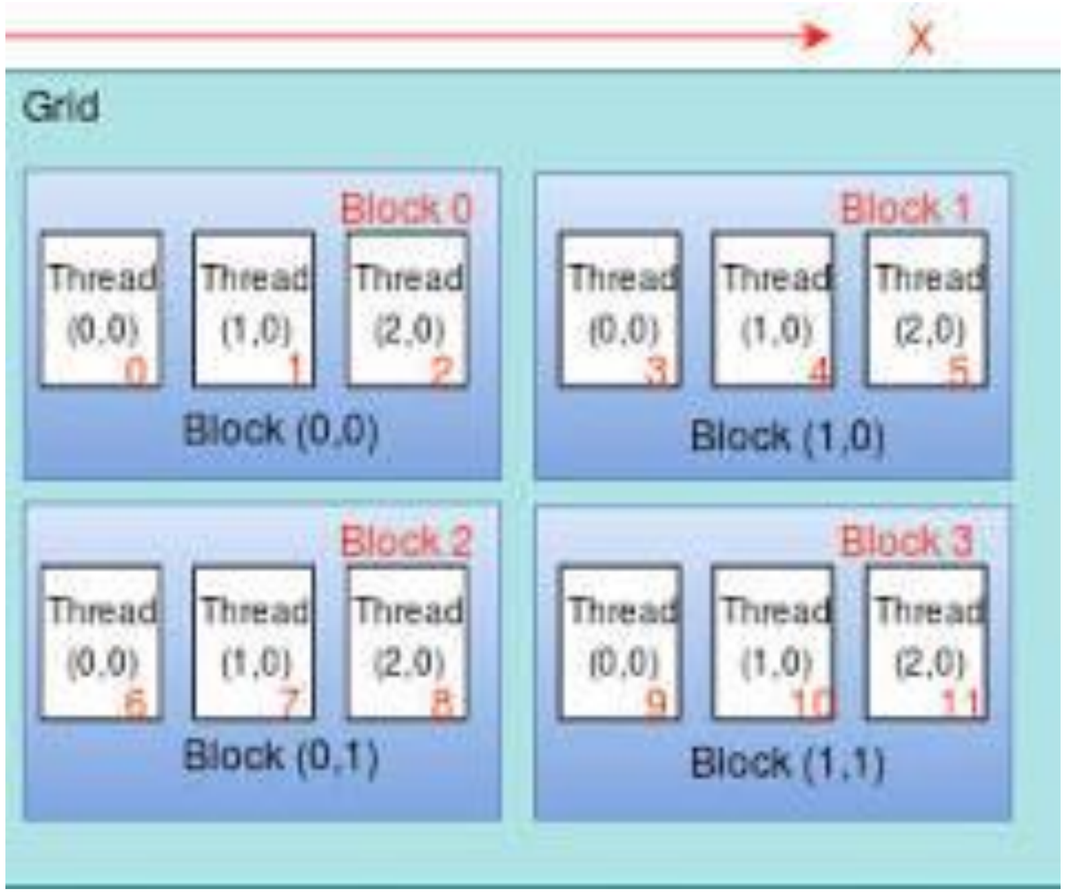
Indexing Calculation

- **Kernel Launch Configuration:** A 2D grid of 2D blocks is defined using `dim3`:

```cpp
dim3 blockDim(16, 16); // 256 threads per block
dim3 gridDim((width + blockDim.x - 1) / blockDim.x, (height + blockDim.
Kernel<<<gridDim, blockDim>>>(...);
``` [6]
```

**Important Considerations:**

- **Boundary Check:** Always ensure `if (x < width && y < height)` to avoid accessing out-of-bounds memory, especially if the data size is not a multiple of the block dimensions.
- **Efficiency:** 2D indexing is highly efficient for matrix operations, as it allows threads to map directly to matrix elements and enhances memory coalescing.
- **Thread ID:** The formula `threadIdx.x + threadIdx.y * blockDim.x` provides a unique 1D ID within a 2D block.


```

Thread Indexing – 1D

1D grid of 1D blocks

```
__device__
int getGlobalIdx_1D_1D(){
      return blockIdx.x *blockDim.x + threadIdx.x;
}
```

1D grid of 2D blocks

```
__device__
int getGlobalIdx_1D_2D(){
      return blockIdx.x * blockDim.x * blockDim.y
             + threadIdx.y * blockDim.x + threadIdx.x;
}
```

1D grid of 3D blocks

```
__device__
int getGlobalIdx_1D_3D(){
      return blockIdx.x * blockDim.x * blockDim.y * blockDim.z
            + threadIdx.z * blockDim.y * blockDim.x
            + threadIdx.y * blockDim.x + threadIdx.x;
}
```

Thread Indexing – 2D

2D grid of 1D blocks

```
__device__ int getGlobalIdx_2D_1D(){
      int blockId   = blockIdx.y * gridDim.x + blockIdx.x;
      int threadId = blockId * blockDim.x + threadIdx.x;
      return threadId;
}
```

2D grid of 2D blocks

```
 __device__
int getGlobalIdx_2D_2D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x;
      int threadId = blockId * (blockDim.x * blockDim.y)
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

2D grid of 3D blocks

```
__device__
int getGlobalIdx_2D_3D(){
       int blockId = blockIdx.x + blockIdx.y * gridDim.x;
       int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                      + (threadIdx.z * (blockDim.x * blockDim.y))
                      + (threadIdx.y * blockDim.x) + threadIdx.x;
       return threadId;
}
```

Thread Indexing – 3D

3D grid of 1D blocks

```
__device__
int getGlobalIdx_3D_1D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                    + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * blockDim.x + threadIdx.x;
      return threadId;
}
```

3D grid of 2D blocks

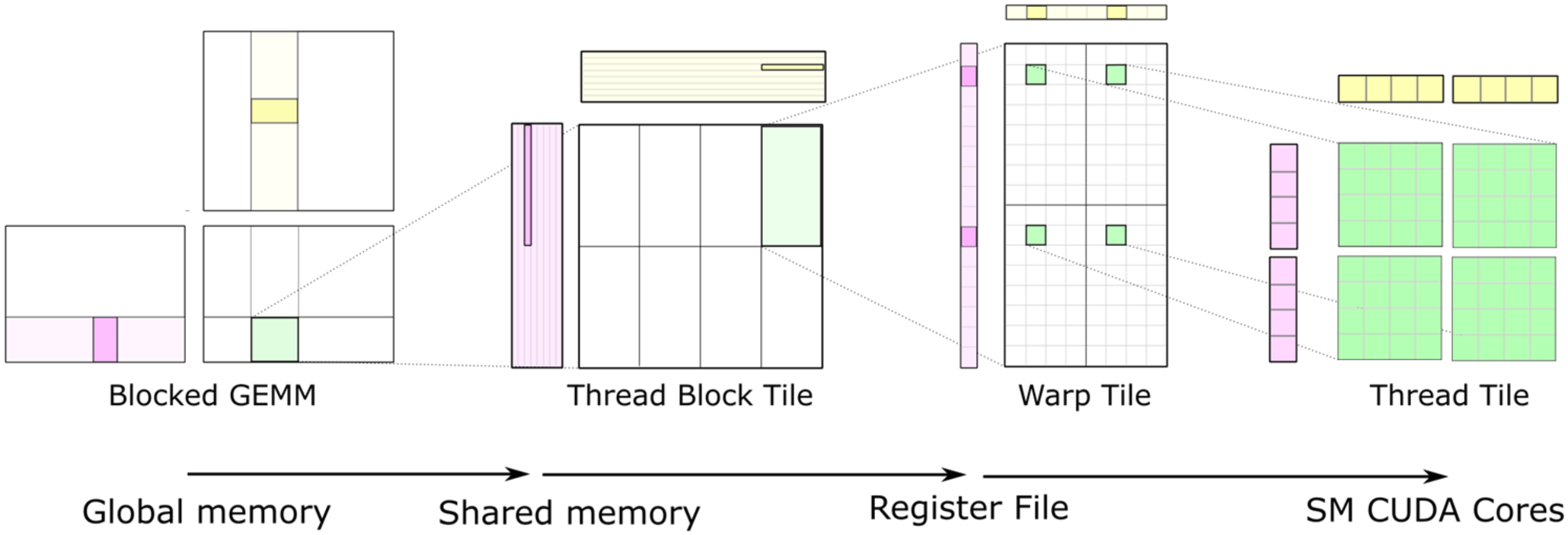
```
__device__
int getGlobalIdx_3D_2D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                    + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y)
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

3D grid of 3D blocks

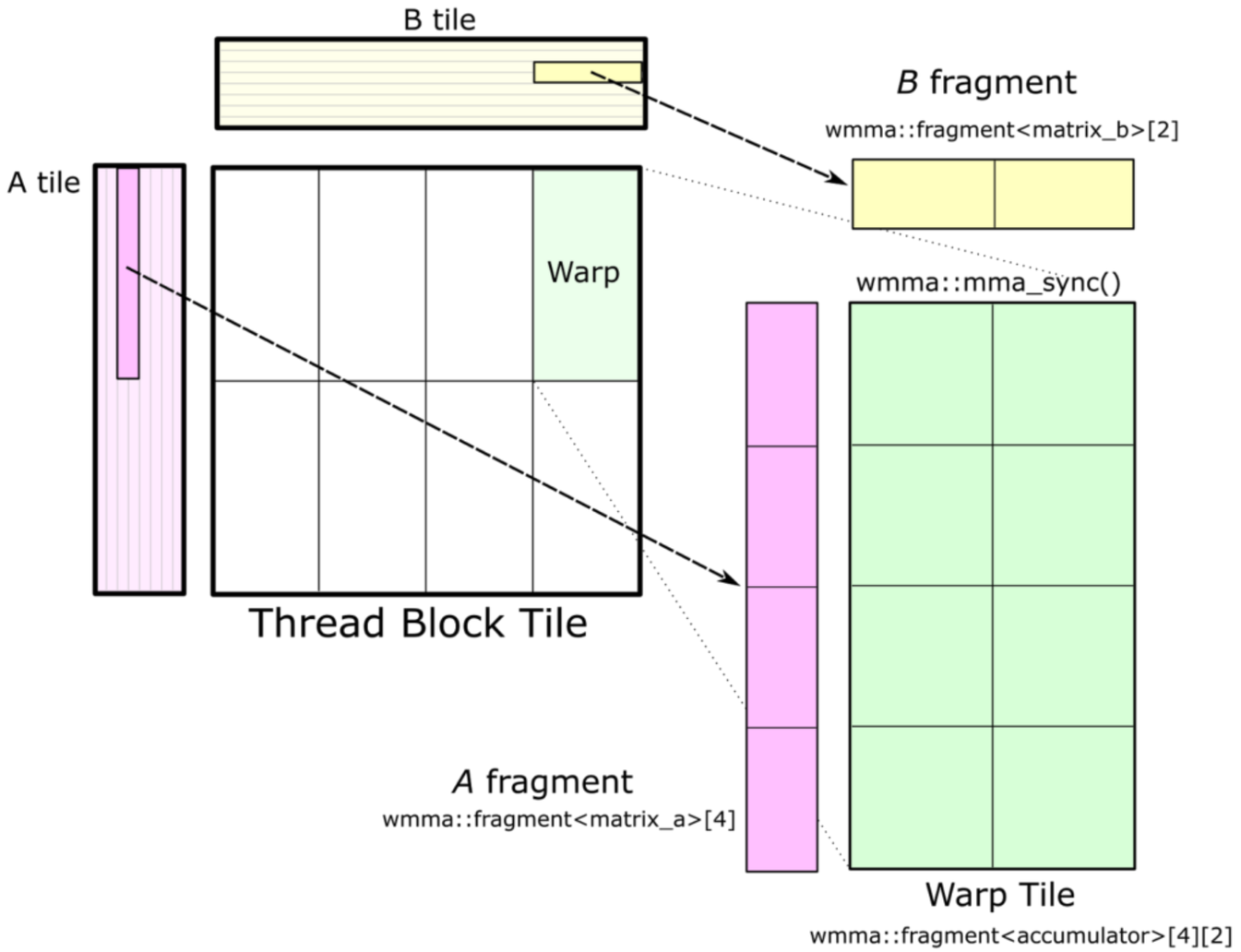
```
__device__
int getGlobalIdx_3D_3D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                     + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                     + (threadIdx.z * (blockDim.x * blockDim.y))
                     + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```

Shared Memory and Register Usage

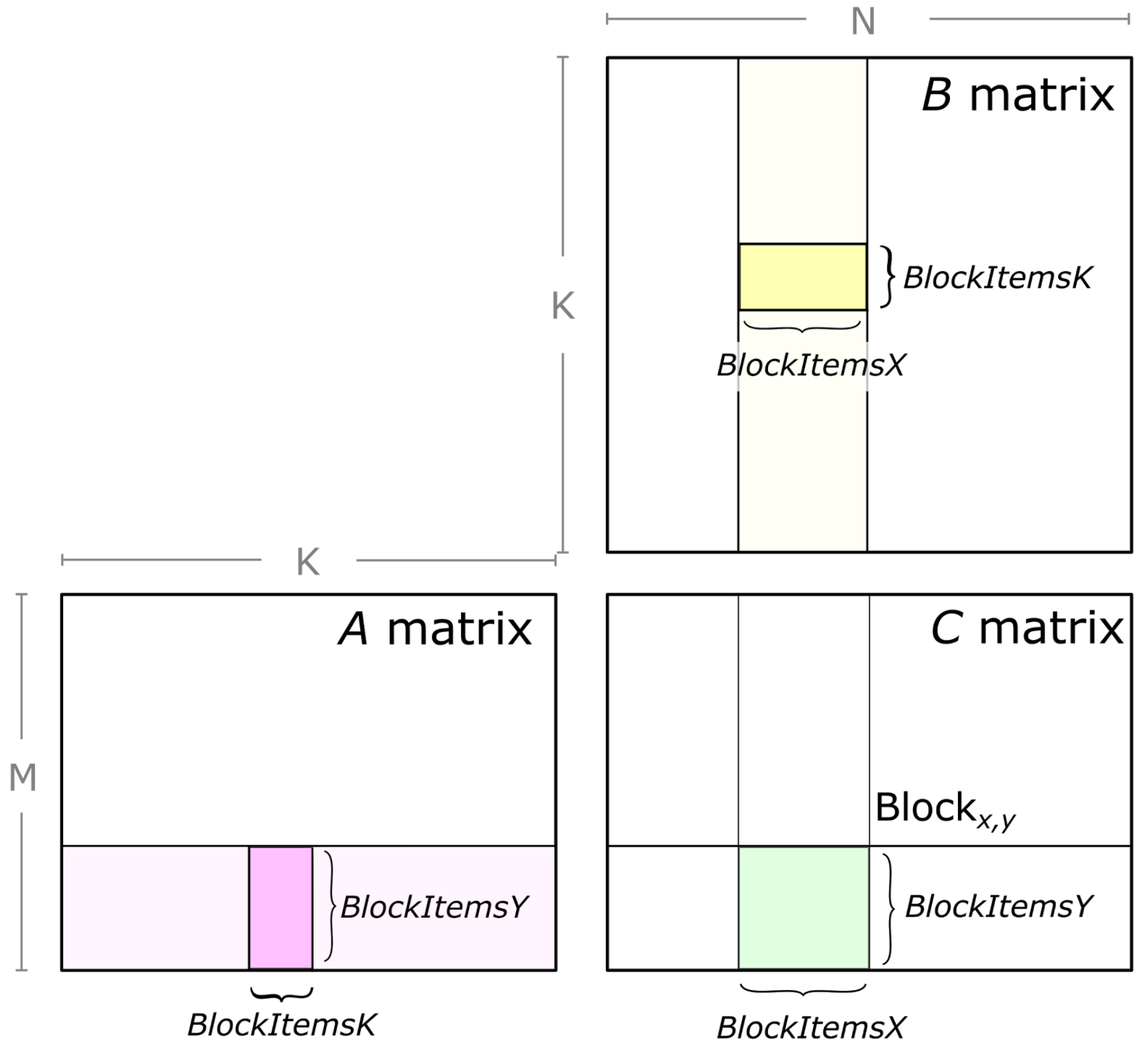
- Shared memory is faster but limited in size.
- Registers store partial results for each thread.
- Efficient usage minimizes global memory transactions.



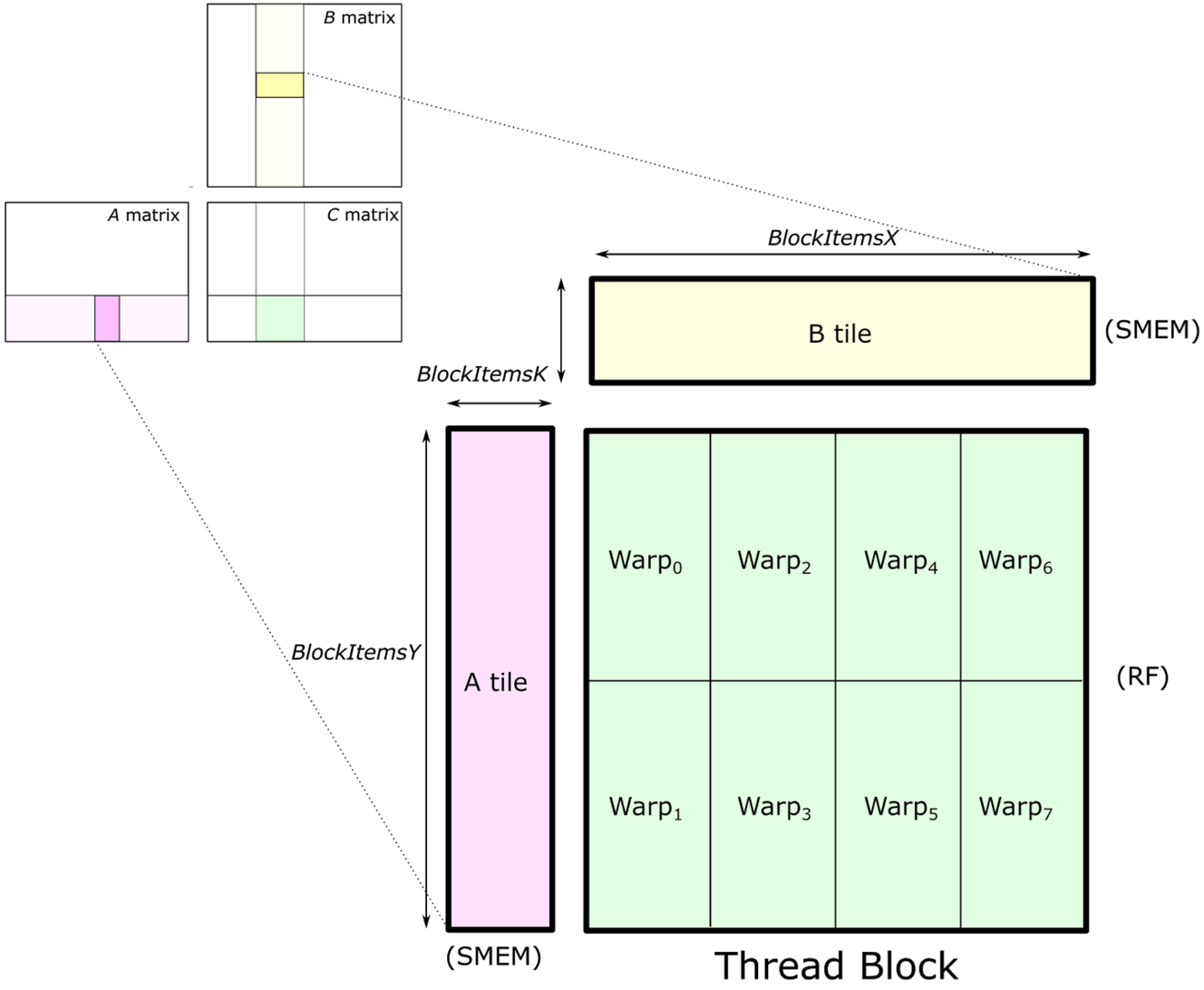
Shared memory in Matmul



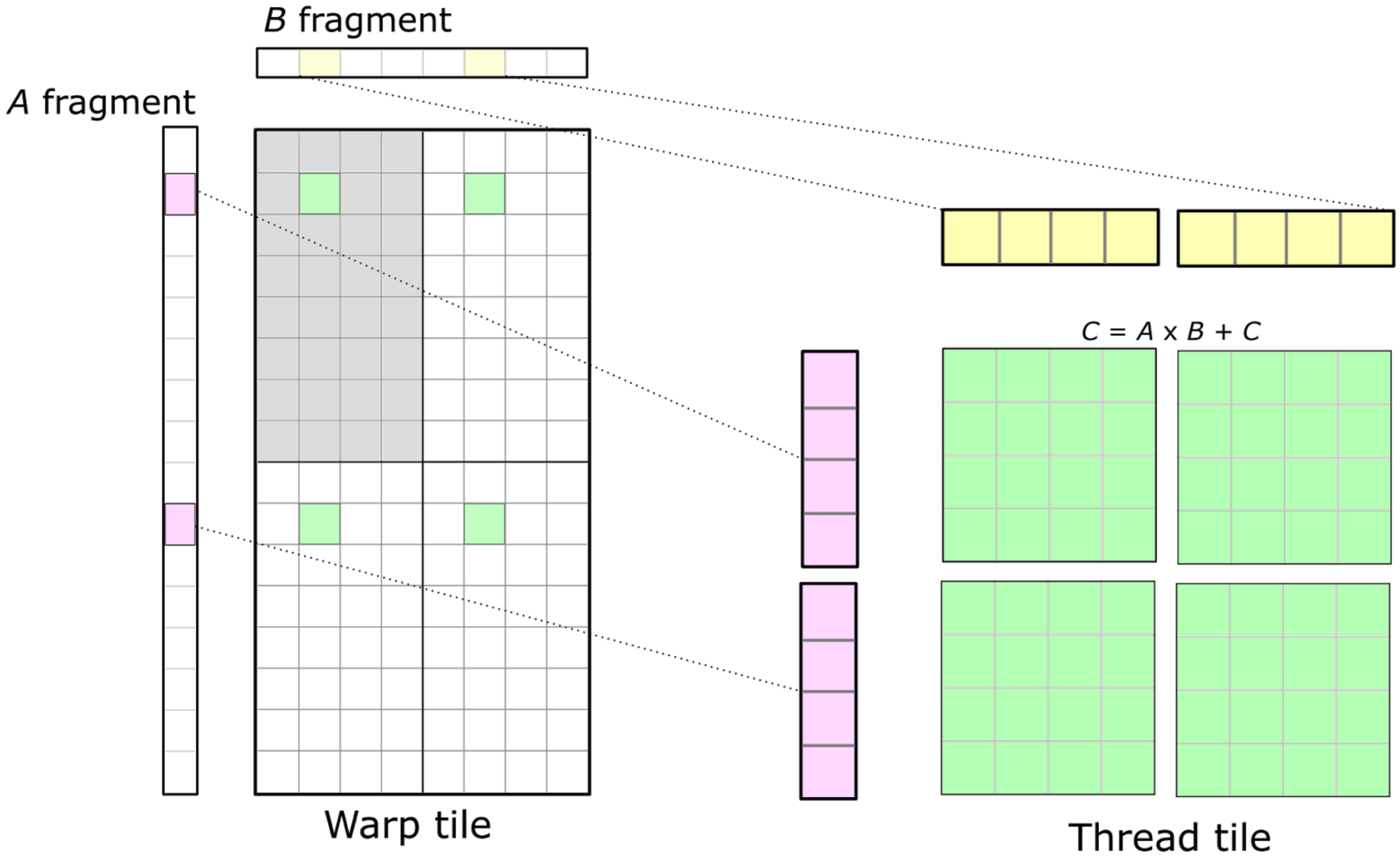
Shared memory in Matmul



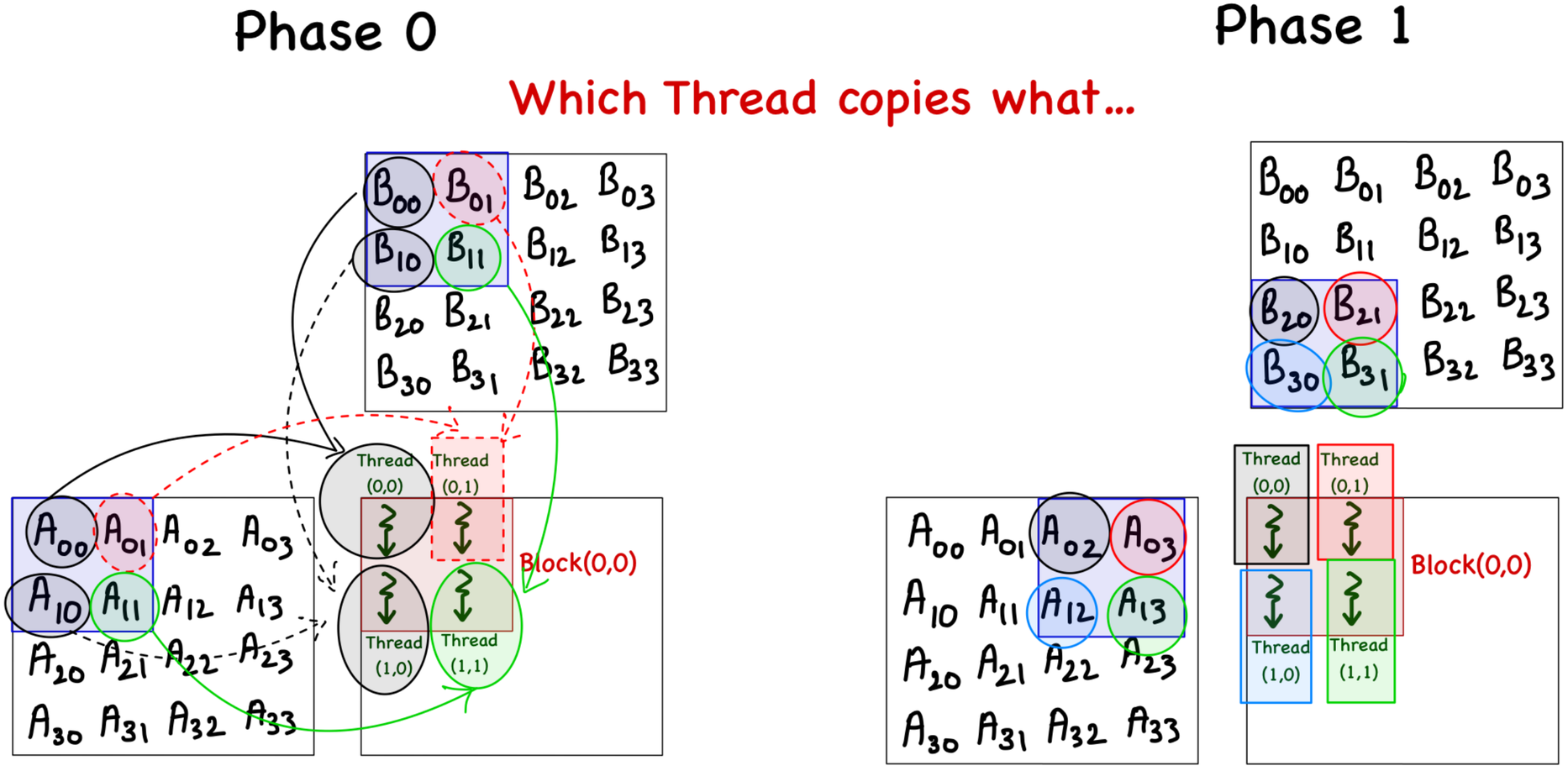
Shared memory in Matmul



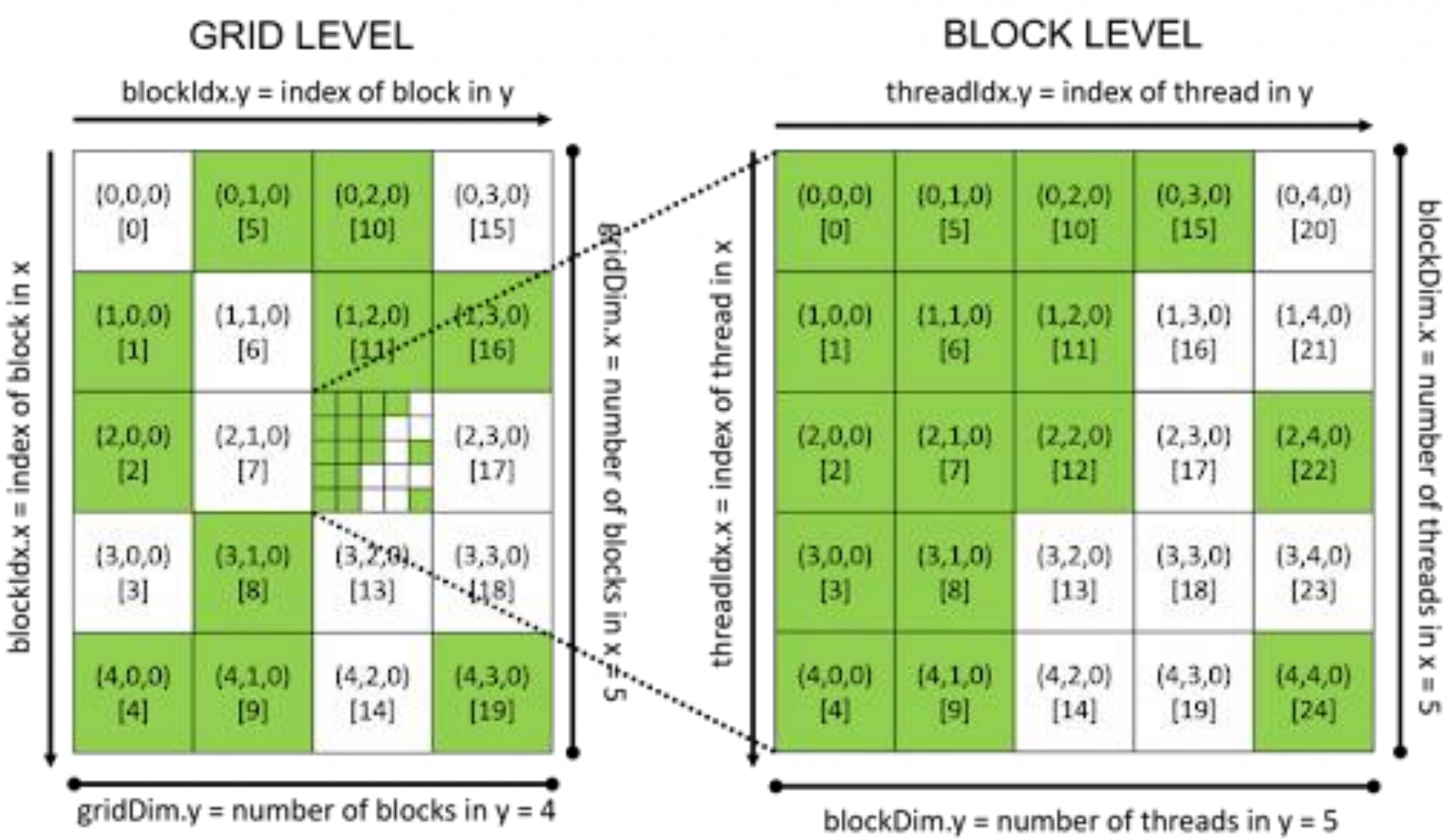
Shared memory in Matmul



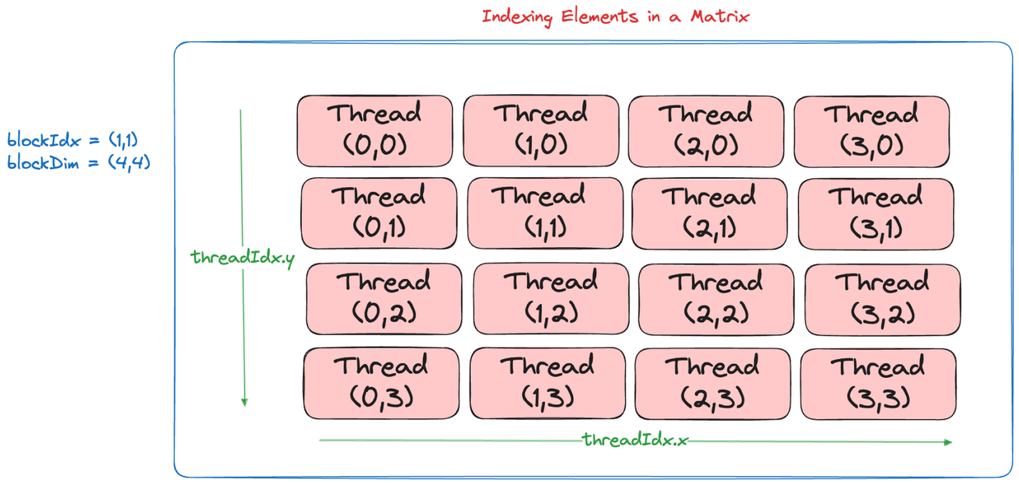
Example Tiled MatMul with Block-size = 2

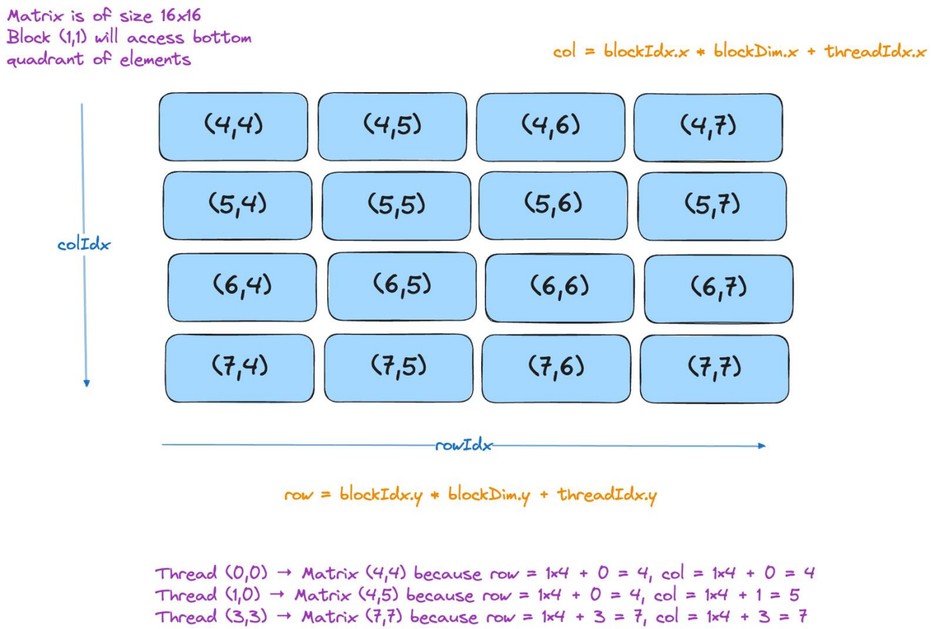


CUDA Thread Indexing



CUDA Thread Indexing Mapping to Matrix

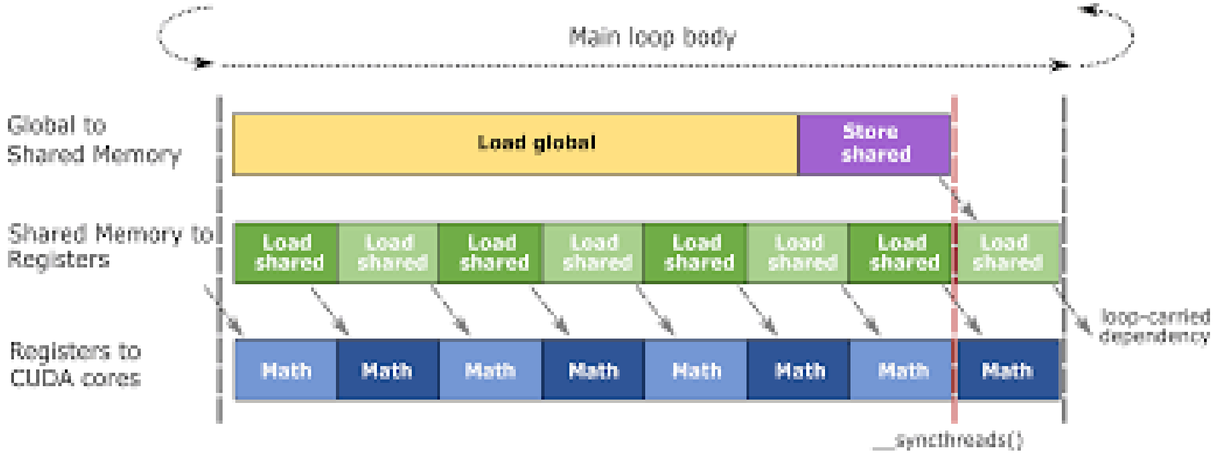


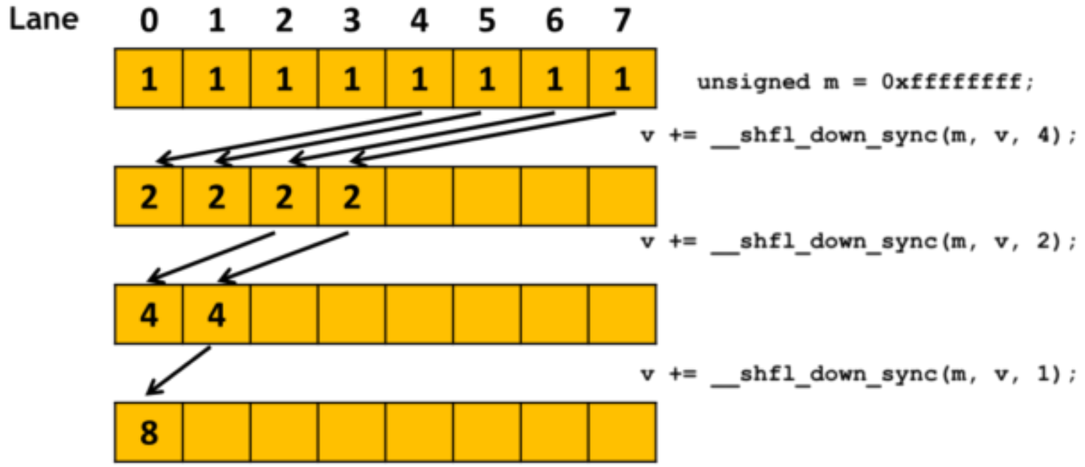


Thread (0,0) → Matrix (4,4) because row = 1x4 + 0 = 4, col = 1x4 + 0 = 4
Thread (1,0) → Matrix (4,5) because row = 1x4 + 0 = 4, col = 1x4 + 1 = 5
Thread (3,3) → Matrix (7,7) because row = 1x4 + 3 = 7, col = 1x4 + 3 = 7

Warp- and Thread-Level Optimization

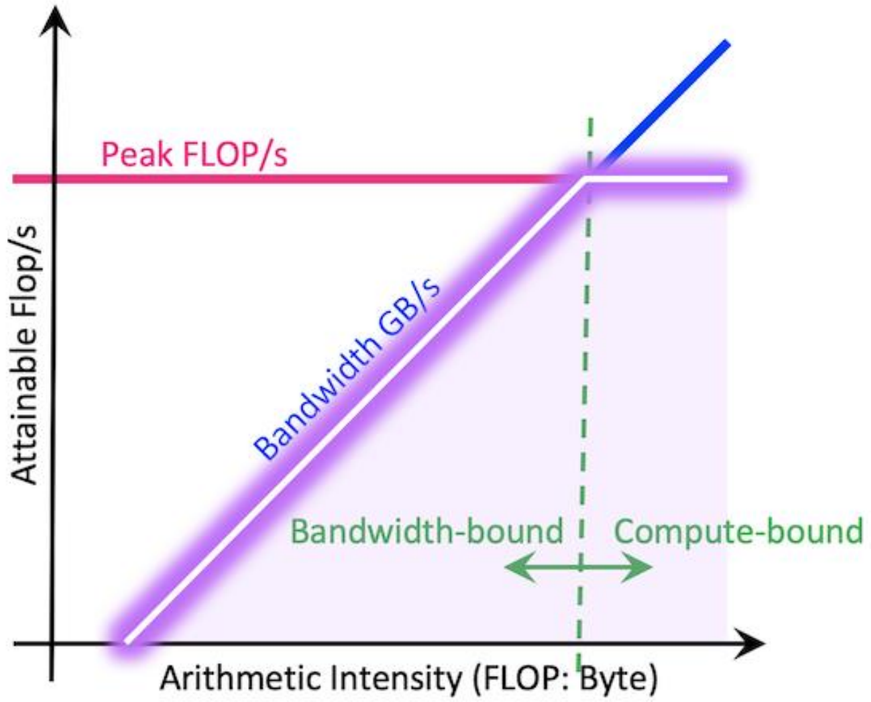
- Warp shuffle for intra-warp communication.
- Use vectorized operations (float4) for better throughput.
- Reduce instruction overhead per output element.





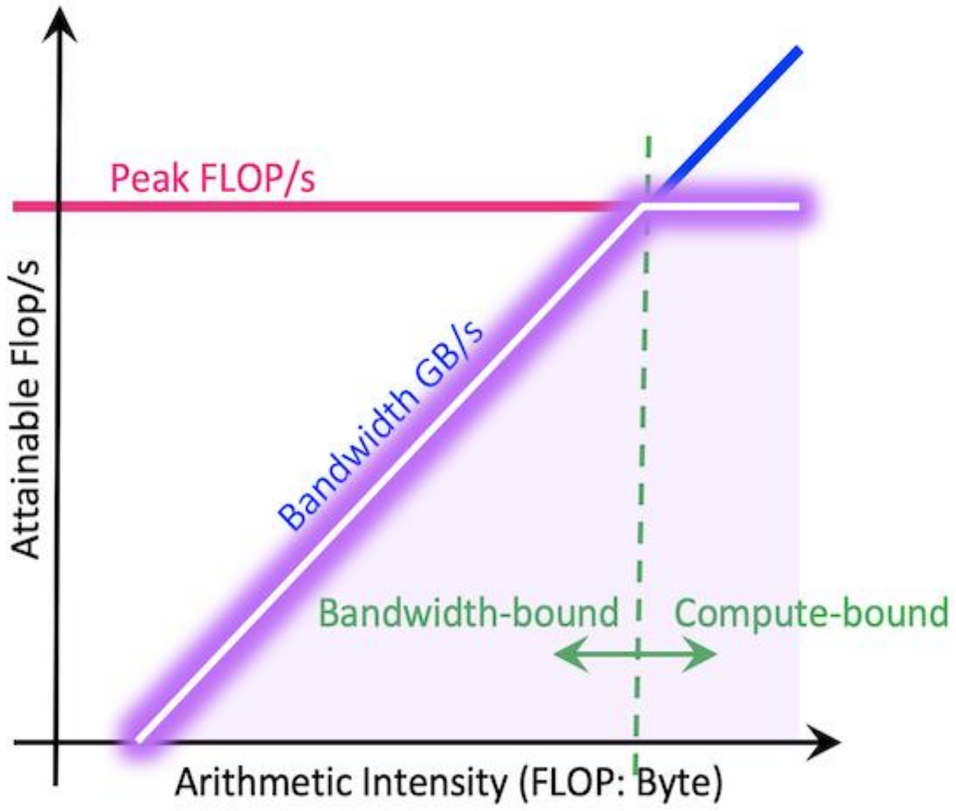
Performance Metrics

- FLOPs per second (GFLOPS).
- Arithmetic intensity: FLOPs / memory bytes loaded.
- Occupancy, memory coalescing, shared memory utilization.



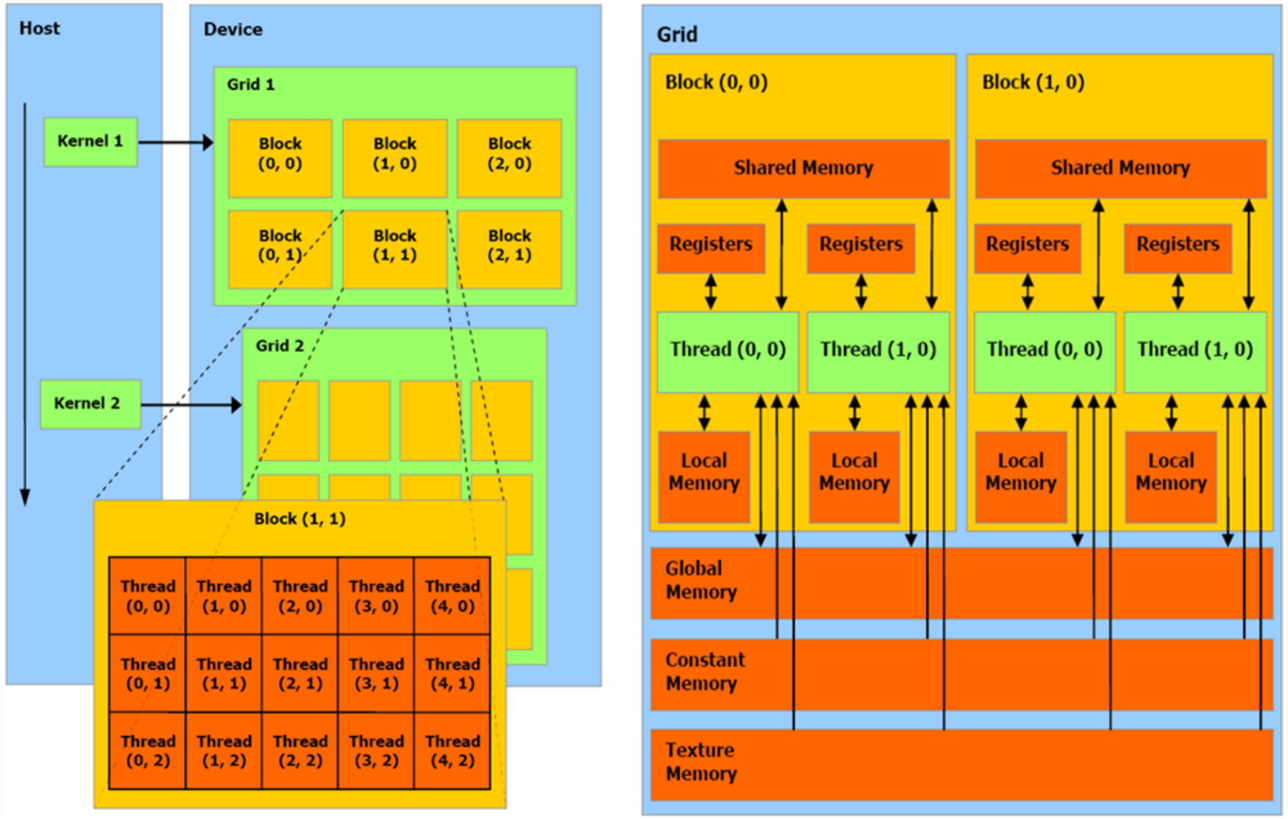
Roofline Model

- Shows performance limit by compute and memory bandwidth.
- Goal: shift from memory-bound to compute-bound region.
- Increase arithmetic intensity via reuse and blocking.



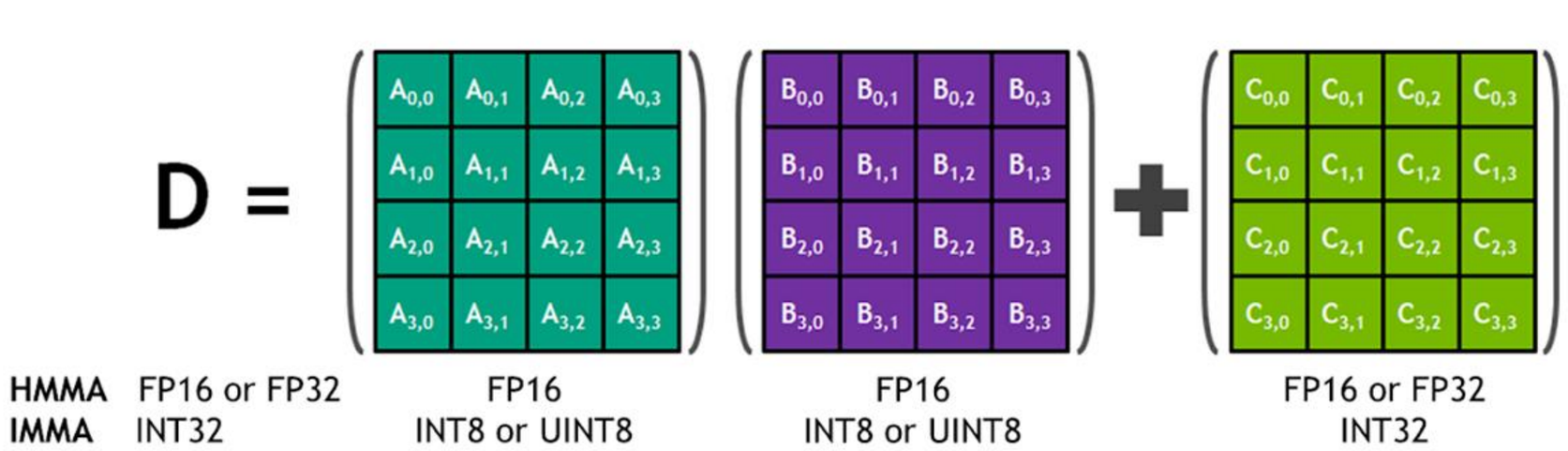
Launch Configuration

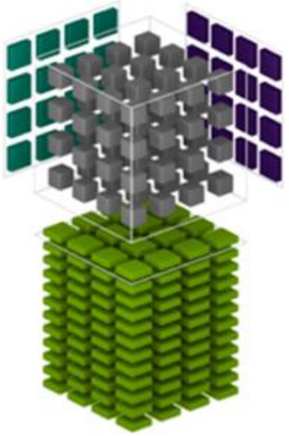
- Block and grid sizes affect occupancy and performance.
- Larger tiles: more reuse but may reduce occupancy.
- Tune parameters for target GPU architecture.



Tensor Cores and Mixed Precision

- Modern GPUs feature Tensor Cores for FP16/TF32 matrix ops.
- Provide massive speedup for GEMM in ML workloads.
- Trade-off between precision and performance.





RICE UNIVERSITY

yuke.wang@rice.edu