Lec5: Memory Architecture and Data Locality

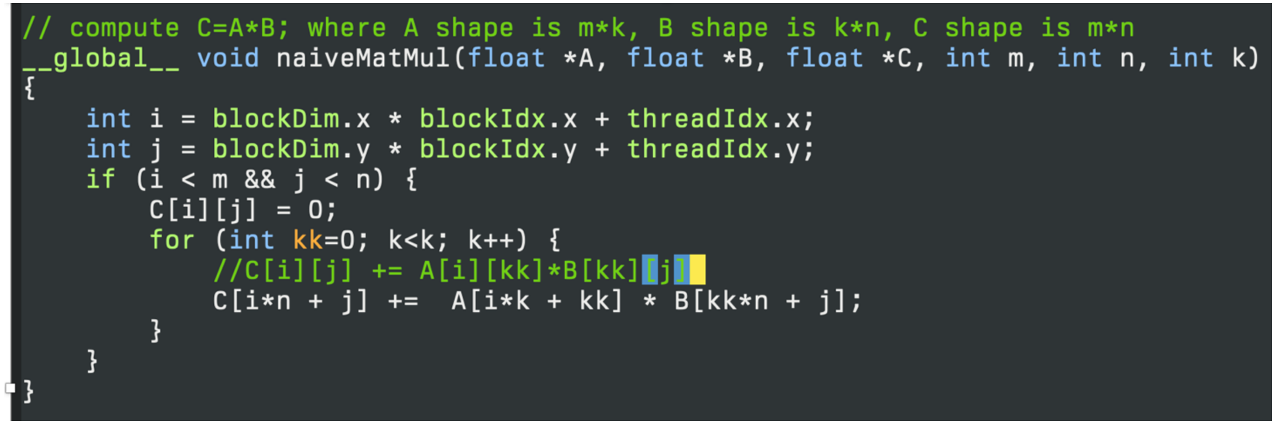
Agenda

- Memory organization
- Position of data for efficient access by massive threads
- Memory bound
- Cache/Tiling
- Data locality

Introduction

- So far, we have studied
  - Writing CUDA kernels
  - Configure and coordinate threads
  - Understand scheduling of blocks and warps
  - GPU architecture (except memory)
- But the kernels we wrote might perform at small fraction of hardware capability, because
  - Frequent access to global memory
  - Un-optimized access to global memory
  - Overall performance bounded by low throughput of global memory

Analysis of matrix multiplication



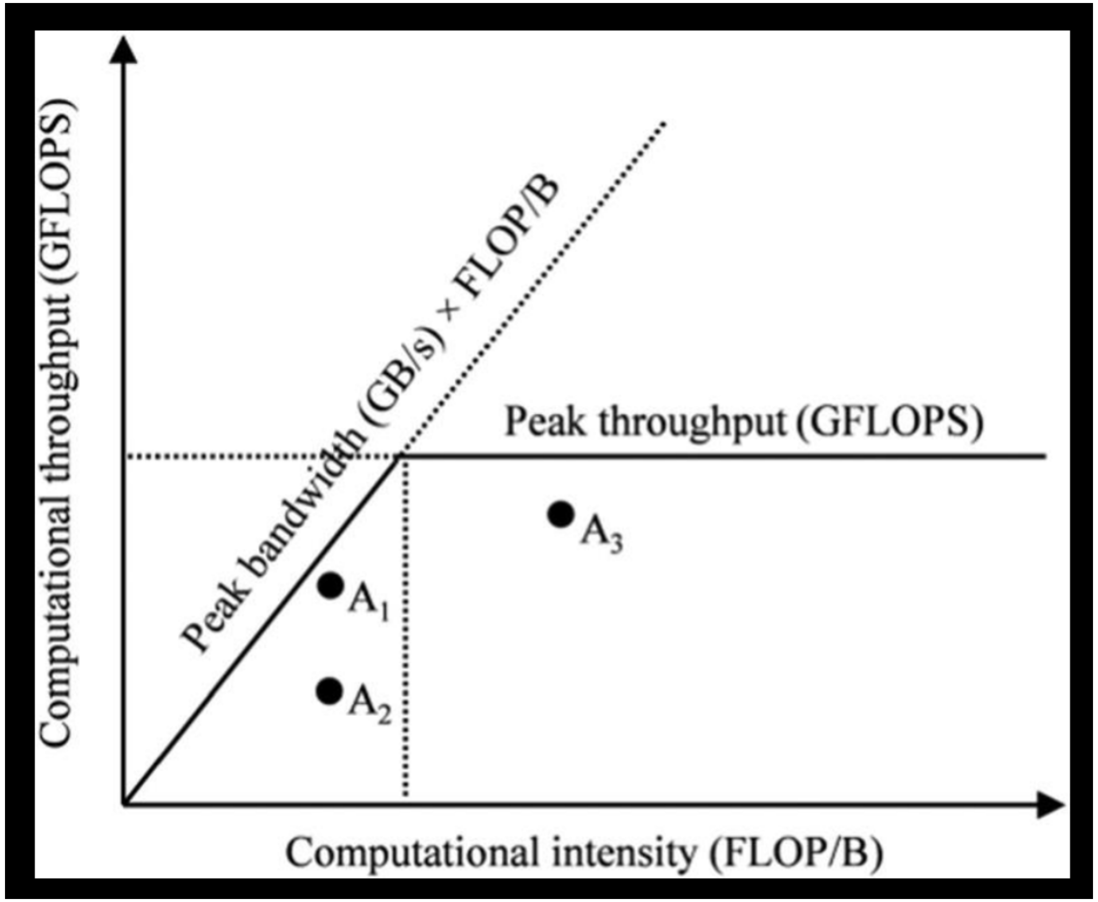
```
// compute C=A*B; where A shape is m*k, B shape is k*n, C shape is m*n
__global__ void naiveMatMul(float *A, float *B, float *C, int m, int n, int k)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    int j = blockDim.y * blockIdx.y + threadIdx.y;
    if (i < m && j < n) {
        C[i][j] = 0;
        for (int kk=0; k<k; k++) {
            //C[i][j] += A[i][kk]*B[kk][j]
            C[i*n + j] +=  A[i*k + kk] * B[kk*n + j];
        }
    }
}
```

- Let’s analyze the performance of this kernel: what’s the maximum GFLOP/s possible?
- Look at the inner loop iteration: each iteration:
  - 2 FLOP
  - 2 global memory access—8B
  - Arithmetic intensity: 2FLOP/8B = **0.25 FLOP/B**

Naïve Matrix Multiplication Kernel

- (From last slide) The arithmetic intensity (or compute-to-global-memory-ratio) of the naïve matmul kernel is 0.25FLOP/B.
- As an example, the Ampere A100 GPU, the global memory bandwith is 1555 GB/s
- Which means the achieved FLOPS is 0.25*1555 = 389 GFLOP/s
- However, 389 GFLOP/s is only **2% of the peak** SP throughput which is 19.5 TFLOP/s
- If accounting for TensorCores, the peak SP throughput is 156 TFLOP/s, and 389 GFLOP/s is only **0.25% of the peak**
- This is not ideal--any way to improve substantially?

The Roofline Model

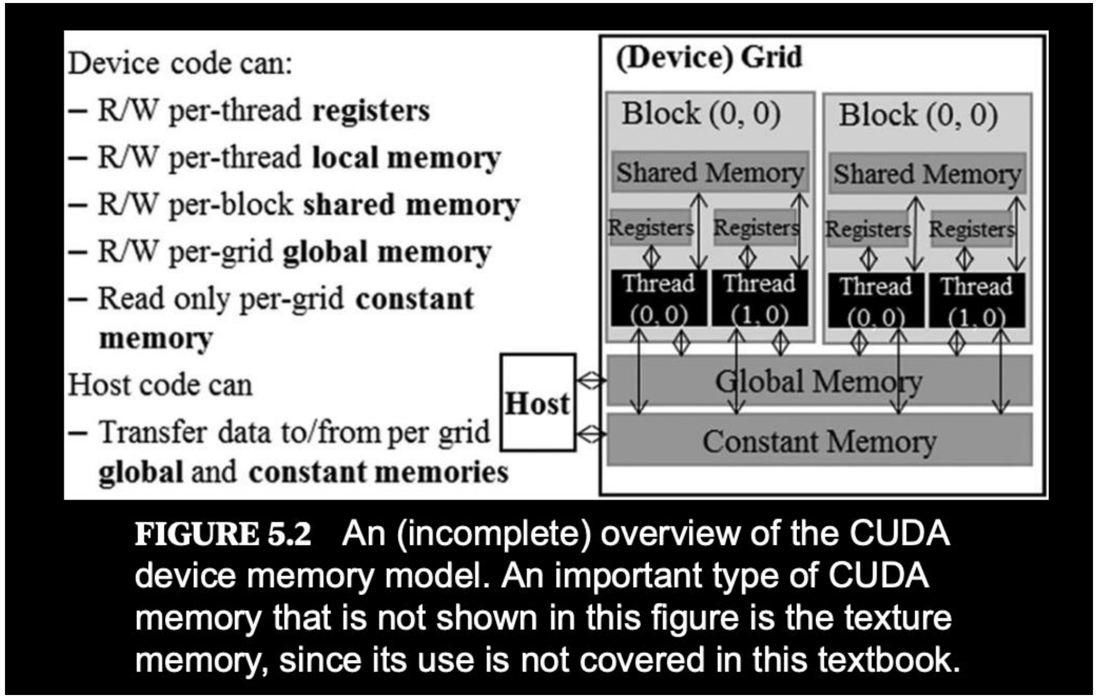


- Low computational intensity – memory bound
- High computational intensity – compute bound
- You want compute bound!
- What’s the FLOP/B needed to achieve peak compute throughput on A100? 19,500 GFLOP/second)/(1555 GB/second)=**12.5 FLOP/B**

Improving Naïve MatMul Kernel

- Need 12.5 FLOP/B intensity to be compute bound!
- I.e., 50 FLOP/element! (1 SP element is 4B)
- Can we improve computation intensity of MatMul?
- We want to reduce traffic to global memory, which means that
  - Need to increase data reuse on-chip (cache? Register?)
  - Need to change matmul algorithm and implementation
- Depending on how much reduction to global memory is achieved, performance of MatMul kernel can differ by **2-3 orders of magnitude!**

CUDA memory types



Device code can:
- R/W per-thread **registers**
- R/W per-thread **local memory**
- R/W per-block **shared memory**
- R/W per-grid **global memory**
- Read only per-grid **constant memory**

Host code can
- Transfer data to/from per grid **global** and **constant memories**

FIGURE 5.2 An (incomplete) overview of the CUDA device memory model. An important type of CUDA memory that is not shown in this figure is the texture memory, since its use is not covered in this textbook.

- Scope
  - Per-grid
  - Per-block
  - Per-thread
- The programmer dictates where the variables stays

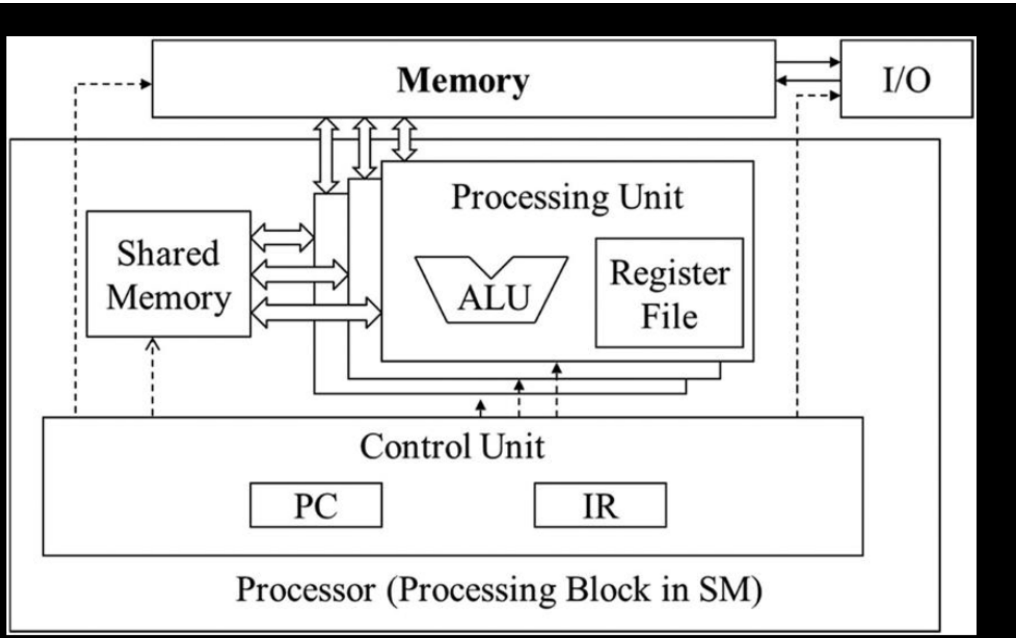
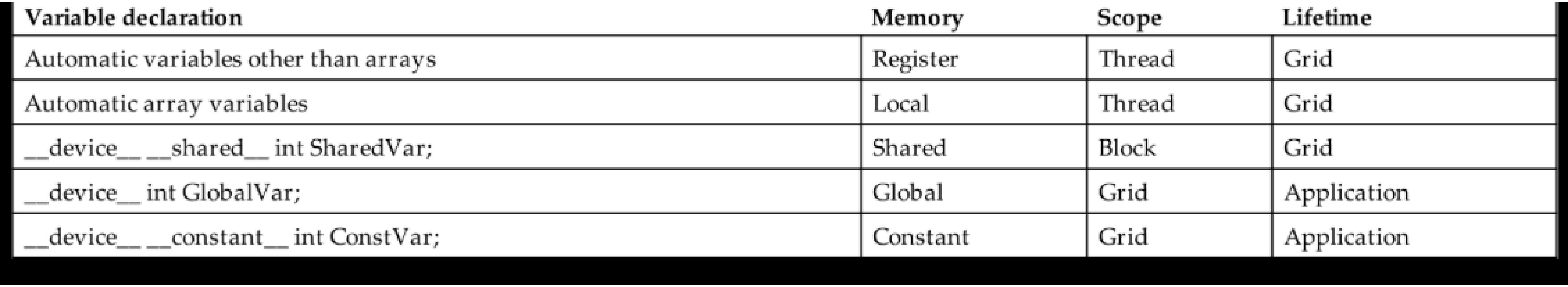


FIGURE 5.4 Shared memory versus registers in a CUDA device SM.

- Aggregate register bandwidth is usually >=10x shared memory >=100x global memory

- Also data in register takes less instructions to access

Variable <-> Memory type



Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Grid
Automatic array variables	Local	Thread	Grid
__device__ __shared__ int SharedVar;	Shared	Block	Grid
__device__ int GlobalVar;	Global	Grid	Application
__device__ __constant__ int ConstVar;	Constant	Grid	Application

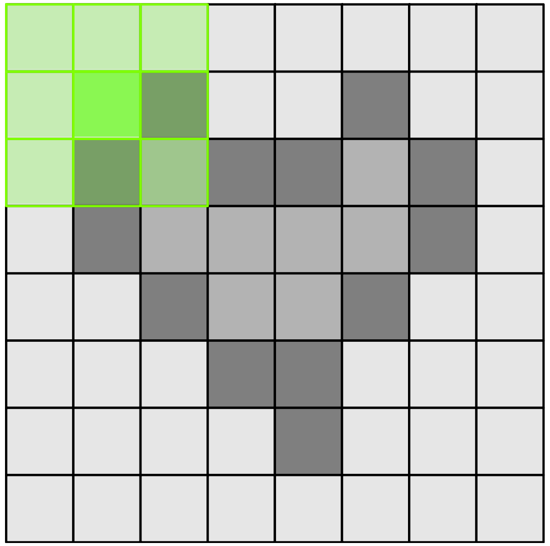
MatMul: Tiling to reduce memory traffic

- To reduce global memory traffic, let’s use increase data reuse in **shared memory**
- we could try to increase data reuse in other places as well like registers, but for now let’s focus on shared memory
- The idea is to load parts of A and B into the shared memory and make the most use of them before kicking them out and bring in other parts.

Shared Memory, Tiling

Caching Data in Shared Memory

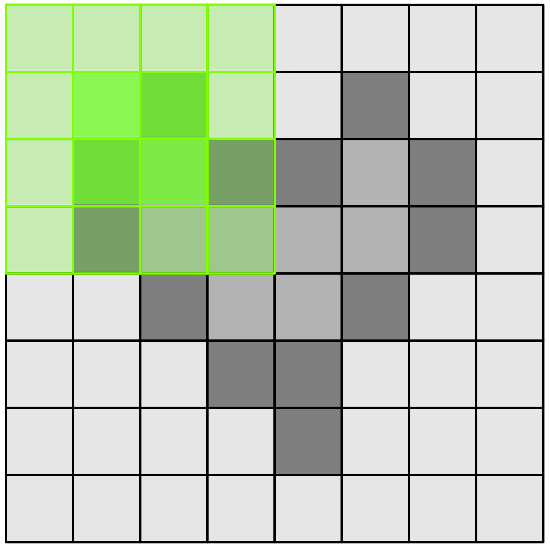
- Same memory locations accessed by neighboring threads



```
for (int i = 0; i < 3; i++){
    for (int j = 0; j < 3; j++){
        sum += gauss[i][j] * Image[(i+row-1)*width + (j+col-1)];
    }
}
```

Data Reuse

- Shared memory tiling



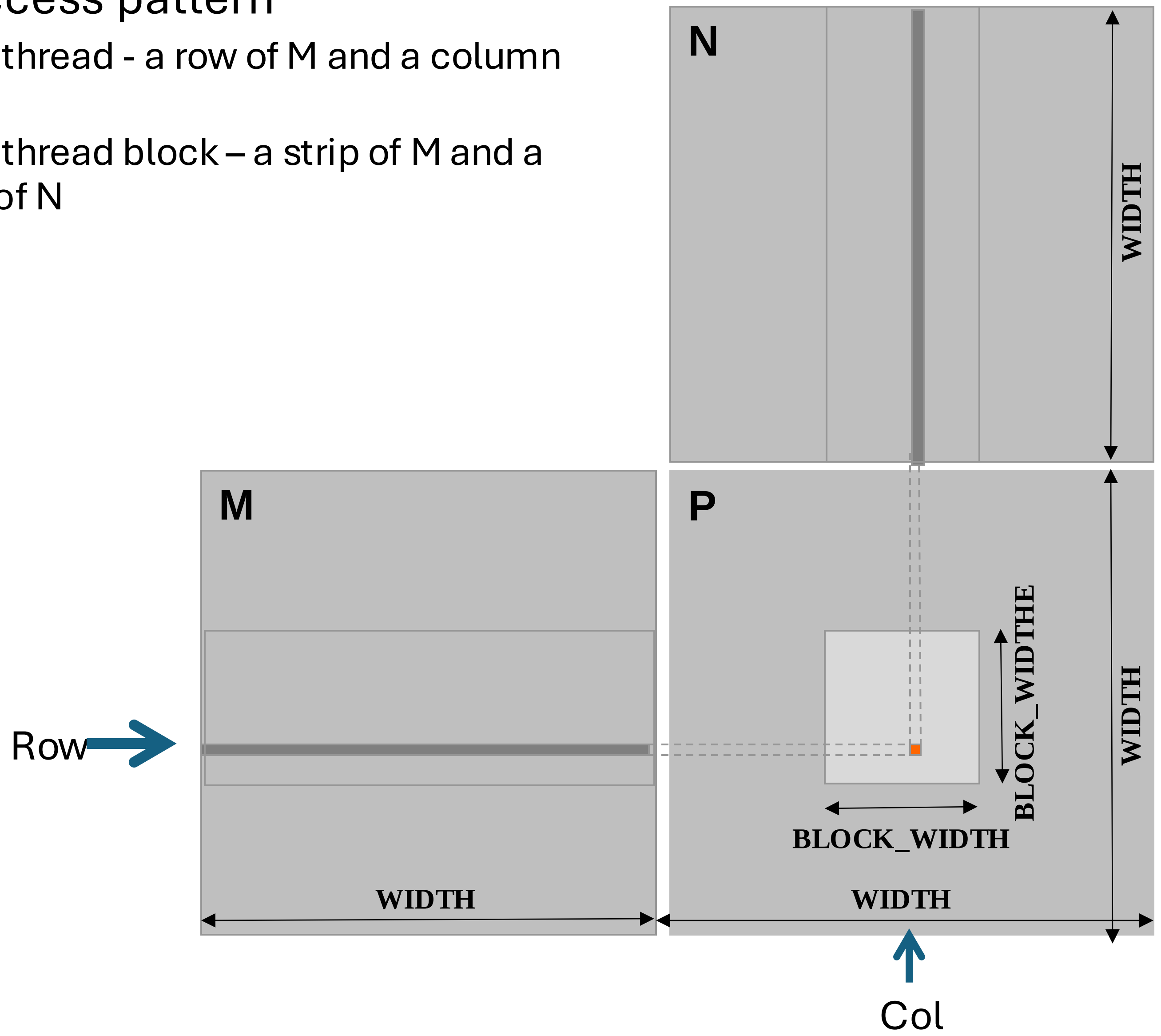
How much reduction to global memory achieved?

```
__shared__ int l_data[(L_SIZE+2)*(L_SIZE+2)];
…
Load tile into shared memory
__syncthreads();
for (int i = 0; i < 3; i++){
  for (int j = 0; j < 3; j++){
    sum += gauss[i][j] * l_data[(i+l_row-1)*(L_SIZE+2)+j+l_col-1];
  }
}
```

Tiled MatMul

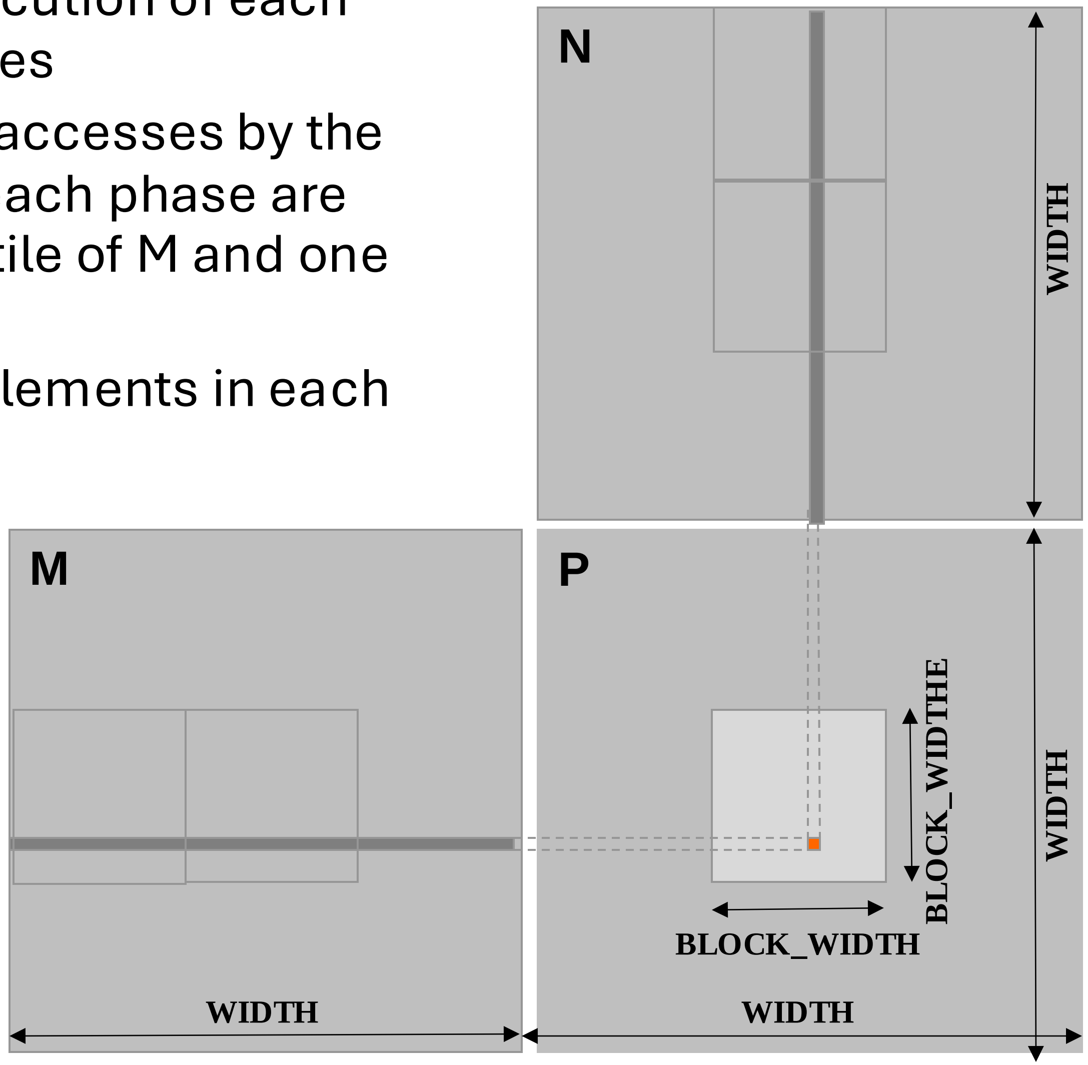
Matrix Multiplication

- Data access pattern
  - Each thread - a row of M and a column of N
  - Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

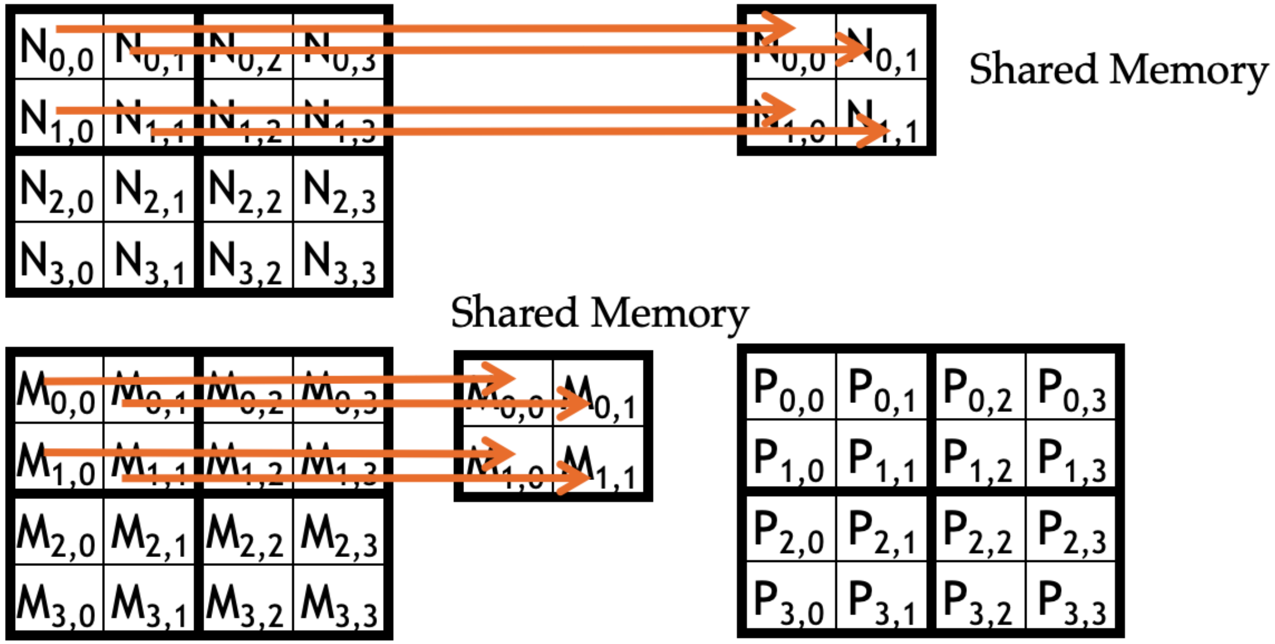
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of TS elements in each dimension



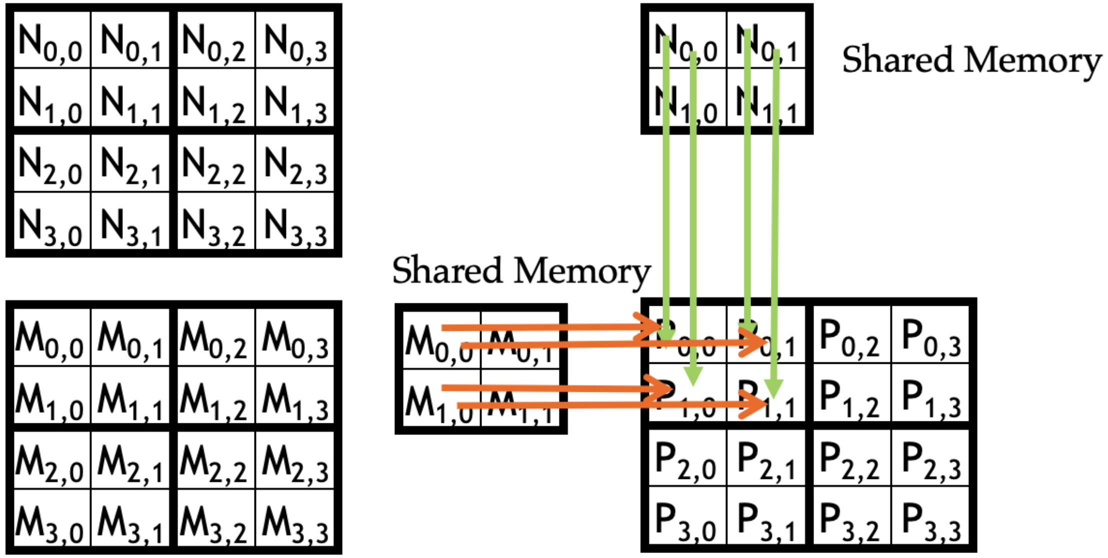
Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in tiled code

Phase 0 Load for Block (0,0)



Phase 0 Use for Block (0,0) (iteration 0)



$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Shared Memory

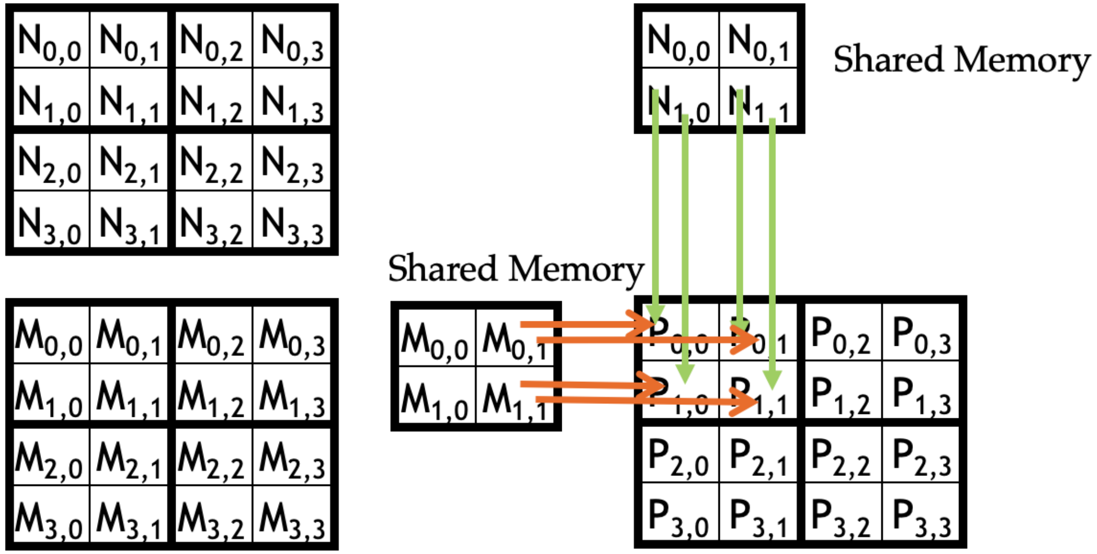
$N_{0,0}$	$N_{0,1}$
$N_{1,0}$	$N_{1,1}$

Shared Memory

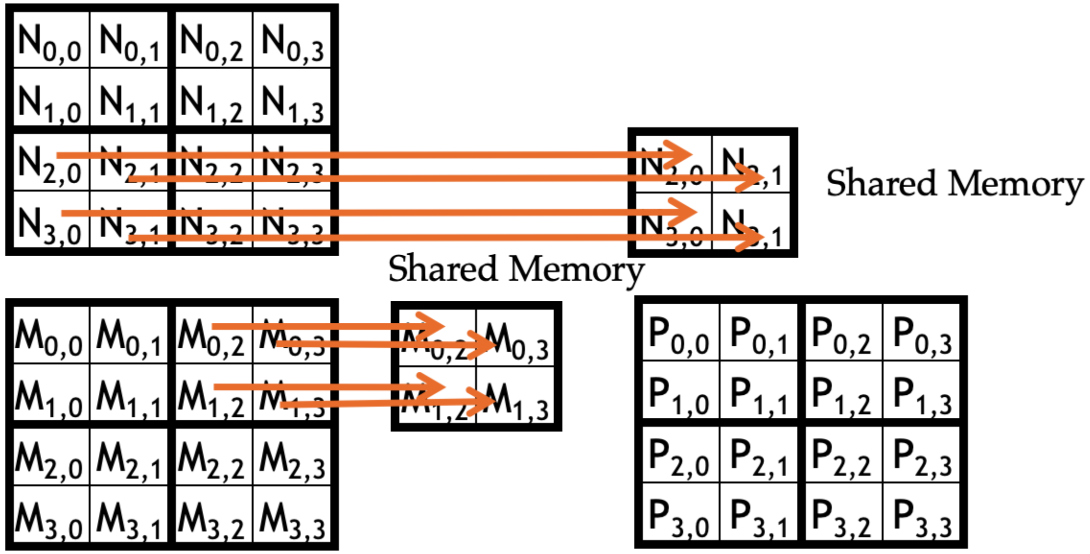
$M_{0,0}$	$M_{0,1}$
$M_{1,0}$	$M_{1,1}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Phase 0 Use for Block (0,0) (iteration 1)


$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
$N_{0,0}$ $N_{0,1}$
$N_{1,0}$ $N_{1,1}$
Shared Memory
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Shared Memory
$M_{0,0}$ $M_{0,1}$
$M_{1,0}$ $M_{1,1}$
$P_{0,0}$ $P_{0,1}$ $P_{0,2}$ $P_{0,3}$
$P_{1,0}$ $P_{1,1}$ $P_{1,2}$ $P_{1,3}$
$P_{2,0}$ $P_{2,1}$ $P_{2,2}$ $P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$

Phase 1 Load for Block (0,0)



$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$N_{2,0}$	$N_{2,1}$
$N_{3,0}$	$N_{3,1}$

Shared Memory

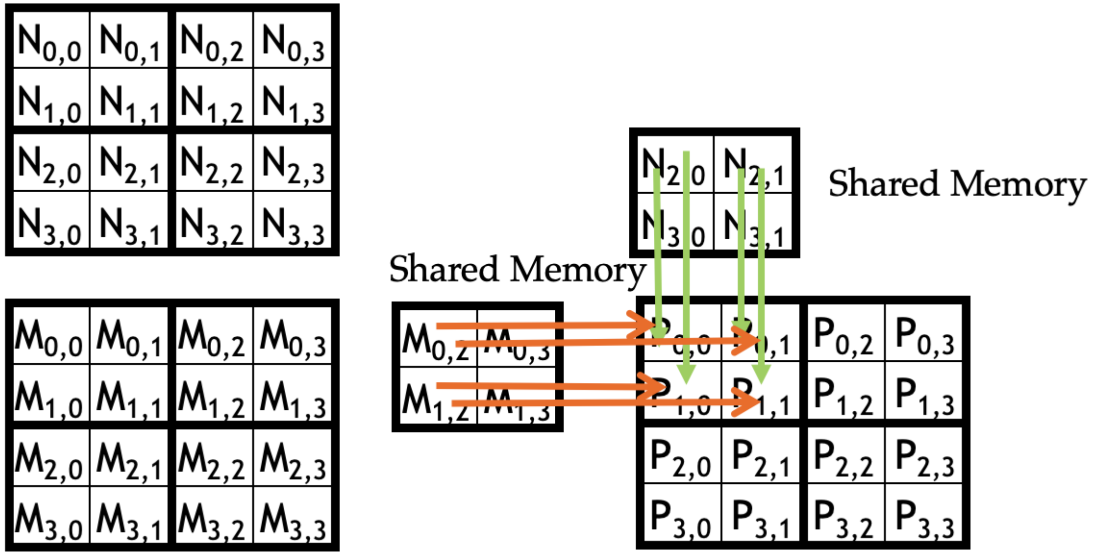
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Shared Memory

$M_{0,2}$	$M_{0,3}$
$M_{1,2}$	$M_{1,3}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Phase 1 Use for Block (0,0) (iteration 0)



$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Shared Memory

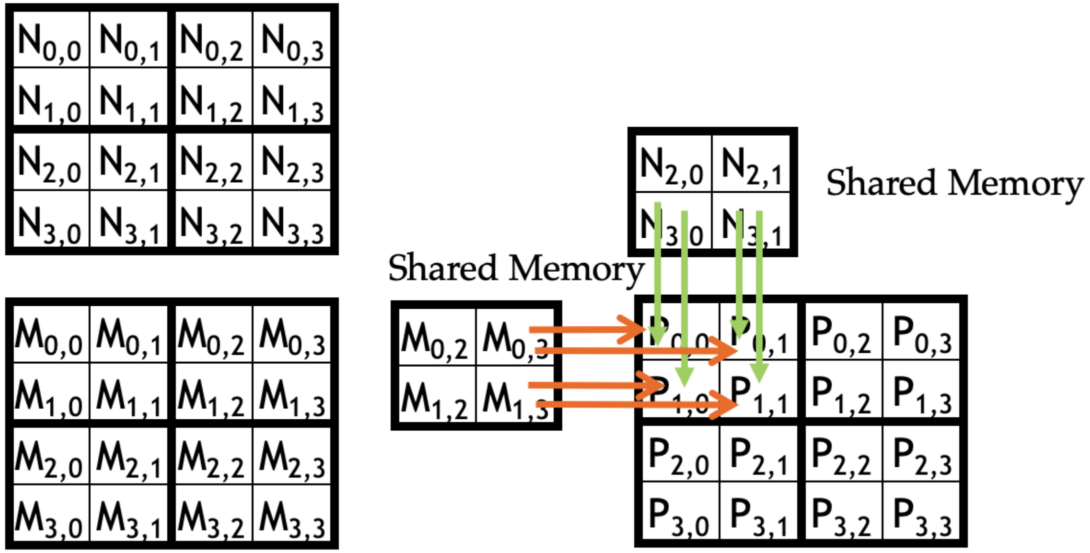
$N_{2,0}$	$N_{2,1}$
$N_{3,0}$	$N_{3,1}$

Shared Memory

$M_{0,2}$	$M_{0,3}$
$M_{1,2}$	$M_{1,3}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

Phase 1 Use for Block (0,0) (iteration 1)



$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

Shared Memory

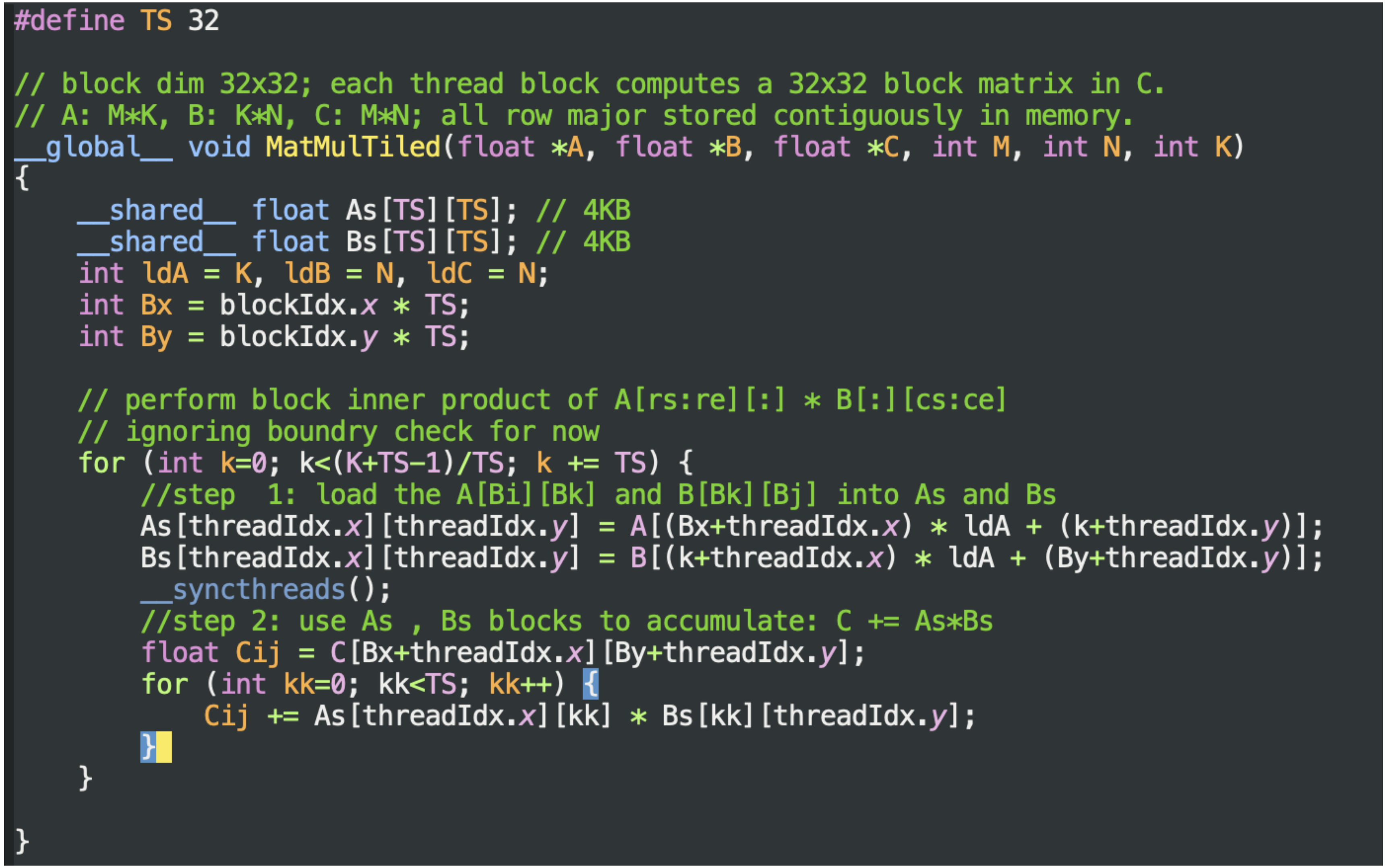
$N_{2,0}$	$N_{2,1}$
$N_{3,0}$	$N_{3,1}$

Shared Memory

$M_{0,2}$	$M_{0,3}$
$M_{1,2}$	$M_{1,3}$

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$

2D Tiled MatMul Kernel



```
#define TS 32

// block dim 32x32; each thread block computes a 32x32 block matrix in C.
// A: M*K, B: K*N, C: M*N; all row major stored contiguously in memory.
__global__ void MatMulTiled(float *A, float *B, float *C, int M, int N, int K)
{
    __shared__ float As[TS][TS]; // 4KB
    __shared__ float Bs[TS][TS]; // 4KB
    int ldA = K, ldB = N, ldC = N;
    int Bx = blockIdx.x * TS;
    int By = blockIdx.y * TS;

    // perform block inner product of A[rs:re][:] * B[:][cs:ce]
    // ignoring boundry check for now
    for (int k=0; k<(K+TS-1)/TS; k += TS) {
        //step  1: load the A[Bi][Bk] and B[Bk][Bj] into As and Bs
        As[threadIdx.x][threadIdx.y] = A[(Bx+threadIdx.x) * ldA + (k+threadIdx.y)];
        Bs[threadIdx.x][threadIdx.y] = B[(k+threadIdx.x) * ldA + (By+threadIdx.y)];
        __syncthreads();
        //step 2: use As , Bs blocks to accumulate: C += As*Bs
        float Cij = C[Bx+threadIdx.x][By+threadIdx.y];
        for (int kk=0; kk<TS; kk++) {
            Cij += As[threadIdx.x][kk] * Bs[kk][threadIdx.y];
        }
    }

}
```

Analysis: Improvement of Tiling

- Assuming TS=32.
- What’s the computational intensity FLOP/B now? (bytes refer to global memory traffic)?
  - Now the unit if a tile TS*TS
  - Load once: 2*TS*TS*4 bytes,
  - Compute: 2*TS*TS*TS
  - Ratio: TS/4 FLOP/B = 8 FLOP/B, a 32x over naïve MatMul.
- What’s the Peak Compute rate now?
  - 8 FLOP/B * 1500 GB/s = 12 TFLOP/s (~60% of hardware peak, not bad!)

Boundary Checks of Tiled MatMul