Matrix multiplication: A “tiled” or “blocked” approach with shared memory

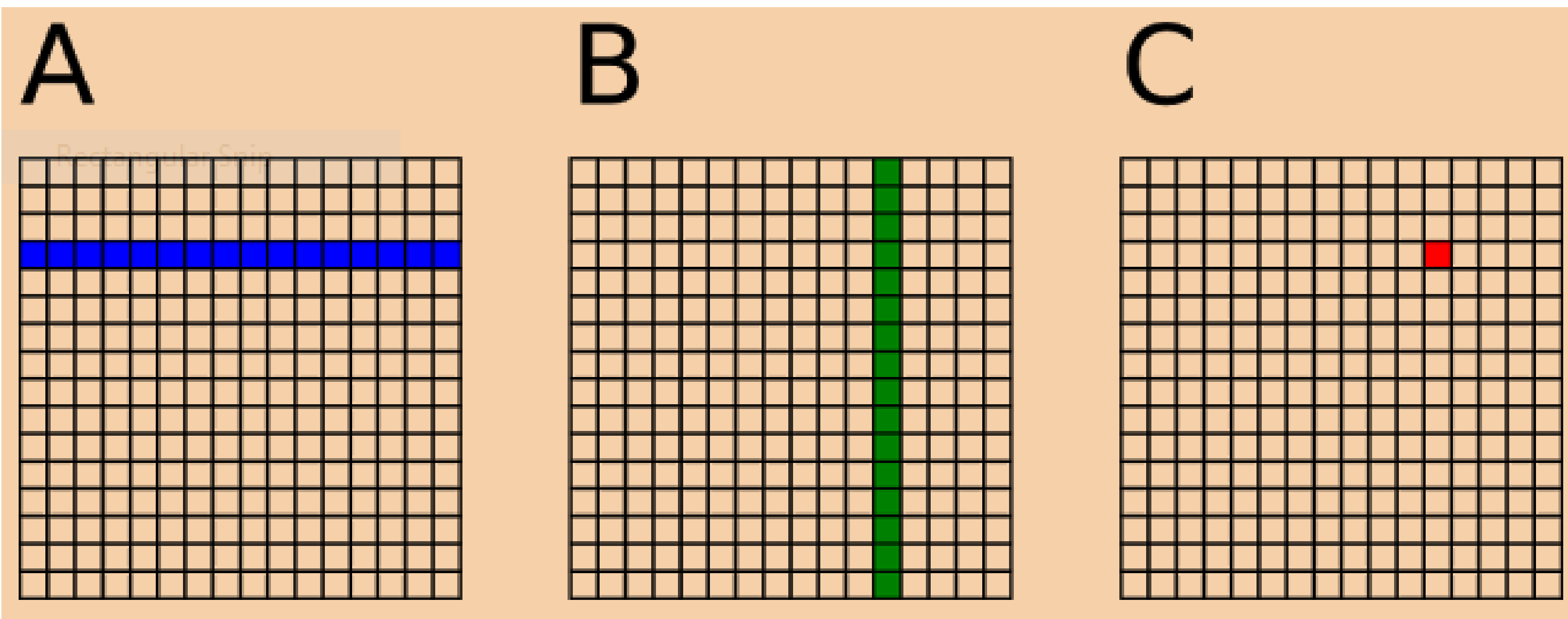
ME574 SPRING 2020

Multiply square (nxn) matrices:

$$\sum_{k=0}^{n-1} A[i,k]B[k,j] = C[i,j]$$

"Naive" approach: 2D comp. grid, each thread computes one element of C

- Each entry in C computed using a row of A and a column of B
- Computation of each element in row 3 accesses every element of row 3 of A
- How many reads of each element of each input array?
- Easiest to implement, but seriously inefficient due to data access redundancy

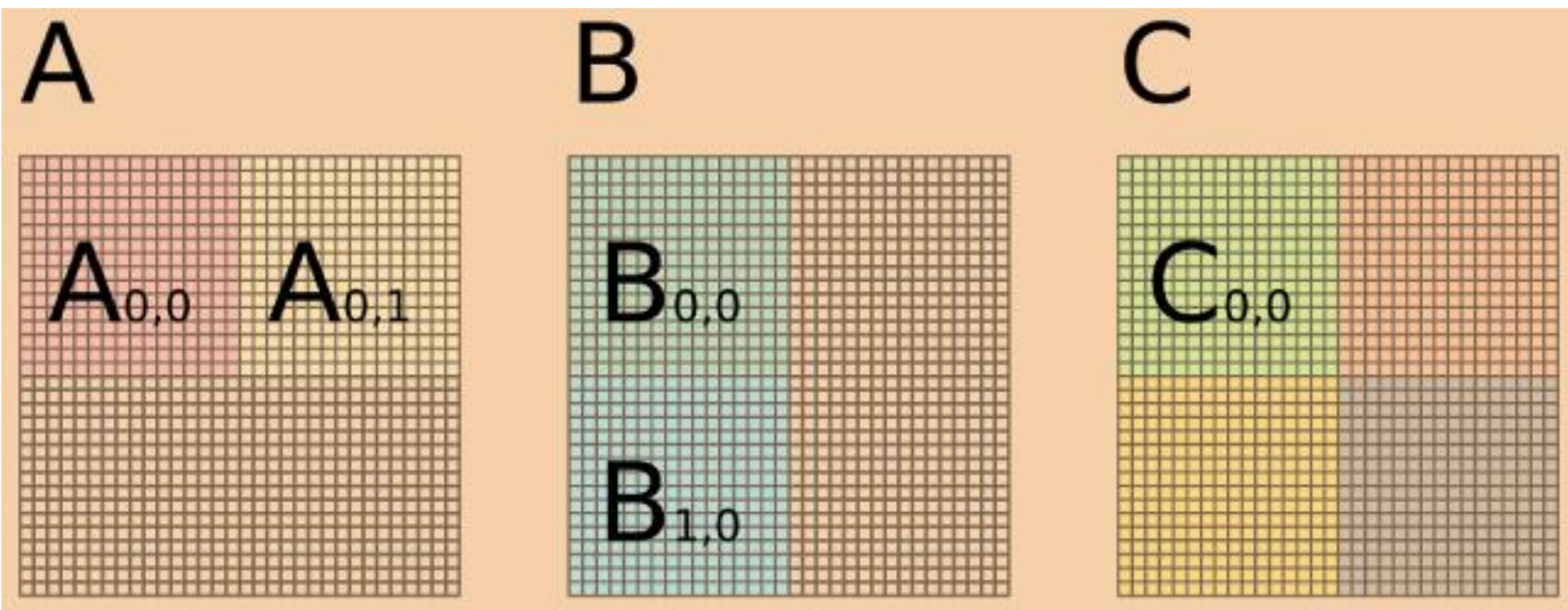


Tiled approach to matrix multiplication: $\sum_{k=0}^{BPG-1} A[i,k]B[k,j] = C[i,j]$

Maintain thread to element correspondence

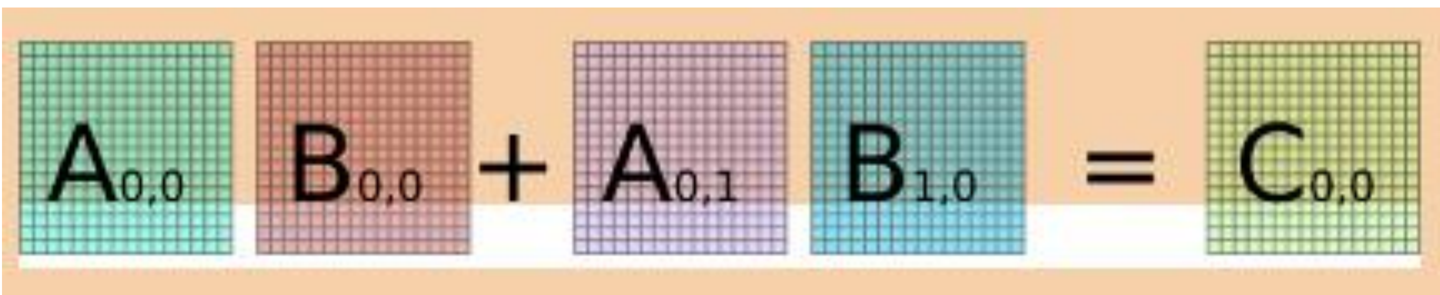
Consider inputs as block matrices ($A[i,k], B[k,j]$ are TPB x TPB sub-matrices)

- Each BLOCK in C computed using a row of A BLOCKS and a column of B BLOCKS
- Computation of each BLOCK involves sequence of BLOCK products
- How many reads of each element of each input array?



Computation of a block:

$$\sum_{k=0}^{BPG-1} A[i,k]B[k,j] = C[i,j]$$

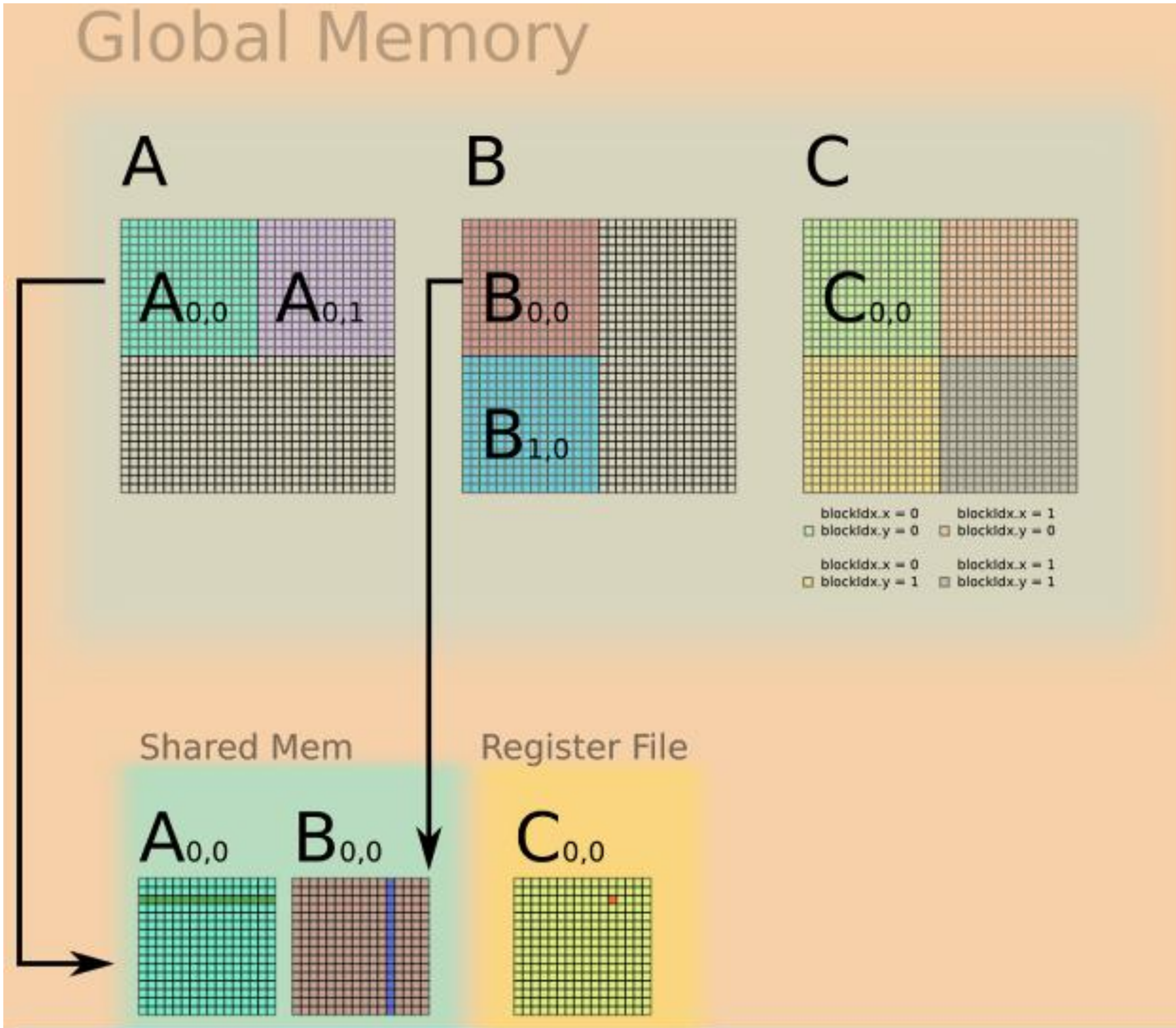


If we can implement this way, we need to do BPG BLOCK products, so each element only read BPG times, instead of accessing BPG*TPB when we do BPG*TPB element products

Tiled approach with shared memory: Stage 0

Load first block from “block row i” of A and “block column j” of B

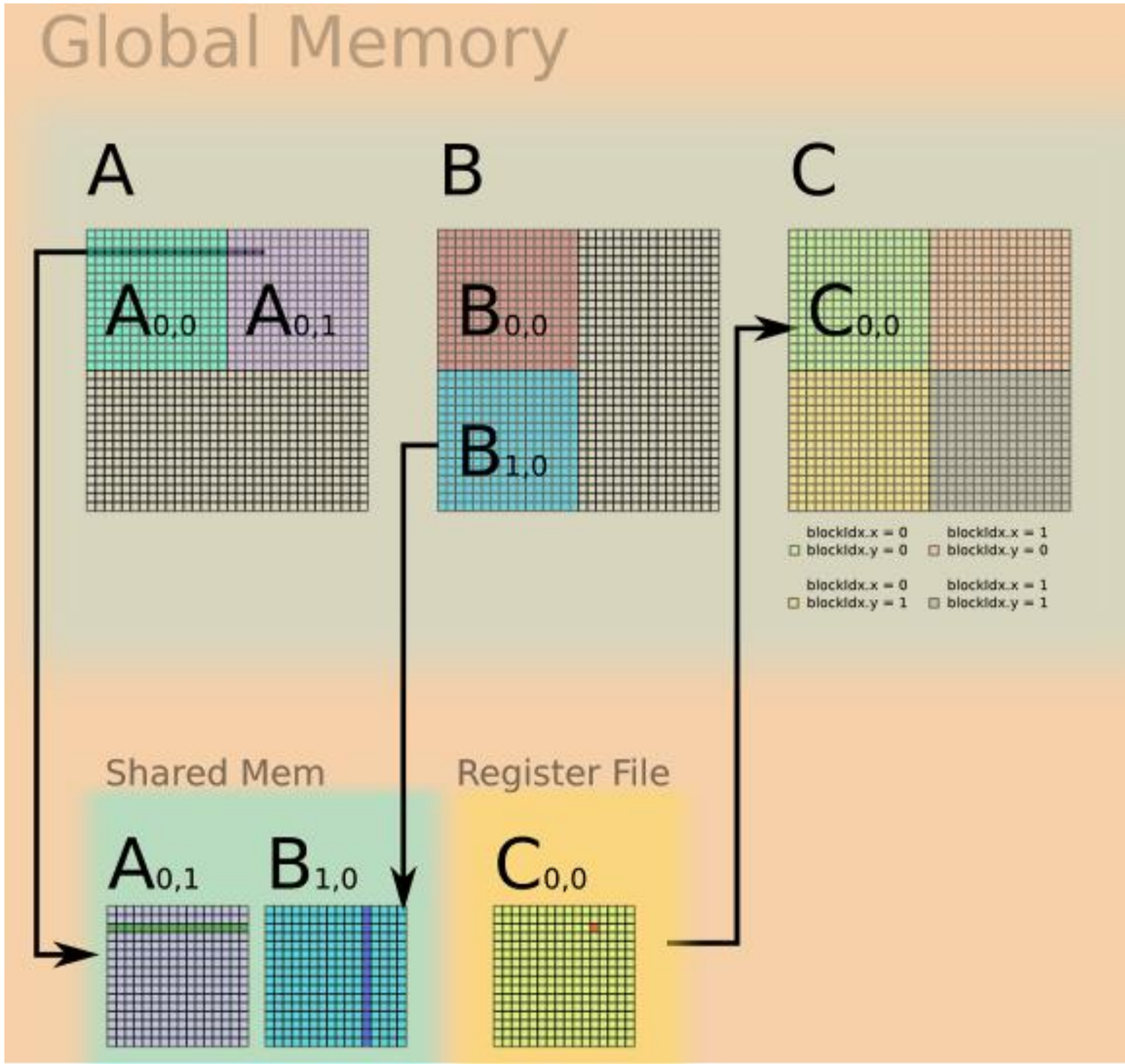
Each thread computes and stores (in reg. mem.) an element in the block product



Tiled approach matrix multiplication: Stage 1

Load next block from "block row i" of A and "block column j" of B

Each thread computes an element in the block product & increments reg. mem.



```
#File: mat_mult.py
import numpy as np
From numba import cuda
@cuda.jit
def matmul(A, B, C):
    ""“Square matrix mult. C = A * B
    """
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

```
File: mat_mult_shared.py
from numba import cuda, float32
TPB = 16 # Compute blocks of TPBxTPB elements.

@cuda.jit
def fast_matmul(A, B, C):
    # Define shared input arrays (spec. size, type)
    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)

    x, y = cuda.grid(2)
    tx, ty, bpg = cuda.threadIdx.x, cuda.threadIdx.y, cuda.gridDim.x
    if x >= C.shape[0] and y >= C.shape[1]:
        return # Bounds test
    # Each thread computes one element
    # Chunk dot product into dot products of TPB-long vectors.
    tmp = 0.
    for i in range(bpg):
        # Load ith row/col block of A/B of data into shared memory
        sA[tx, ty] = A[x, ty + i * TPB]
        sB[tx, ty] = B[tx + i * TPB, y]
        cuda.syncthreads() # Wait until all threads finish preloading

        # Computes partial product on the shared memory
        for j in range(TPB):
            tmp += sA[tx, j] * sB[j, ty]
        cuda.syncthreads() # Wait until all threads finish computing
    C[x, y] = tmp
```

Blocked/tiled matrix product

Shared memory (without distracting “halo” values)

2D shared array

Synchronize as necessary: here both after load and after computation

Reduce data access redundancy by factor of TPB

Measure performance difference

Figures from:

http://www.es.ele.tue.nl/~mwijtvliet/5KK73/?page=mmcuda