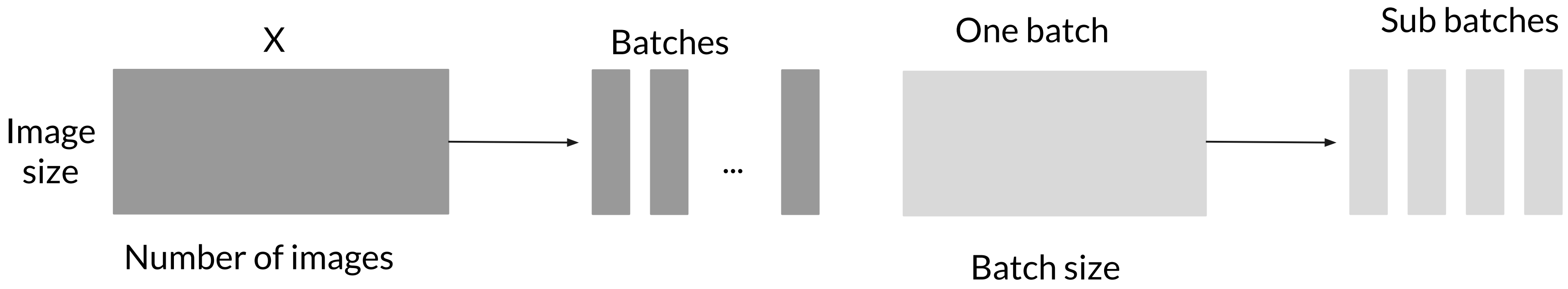
CME 213 Tutorial on Final Project

Mar 12, 2020

Today's Agenda

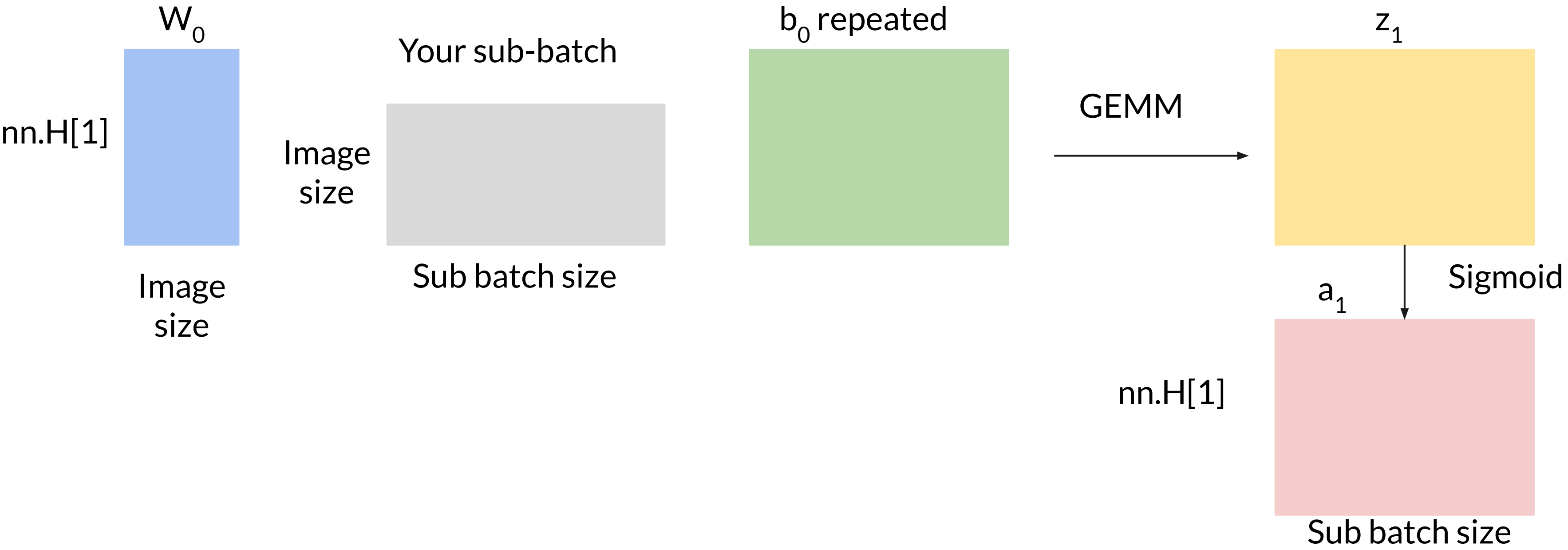
- Steps for parallel train
    - Distribute data
    - Feedforward
    - Backprop
    - Gradient descent
- Brief recap on shared memory
- An Efficient GEMM
- OH

Distribute Training Data

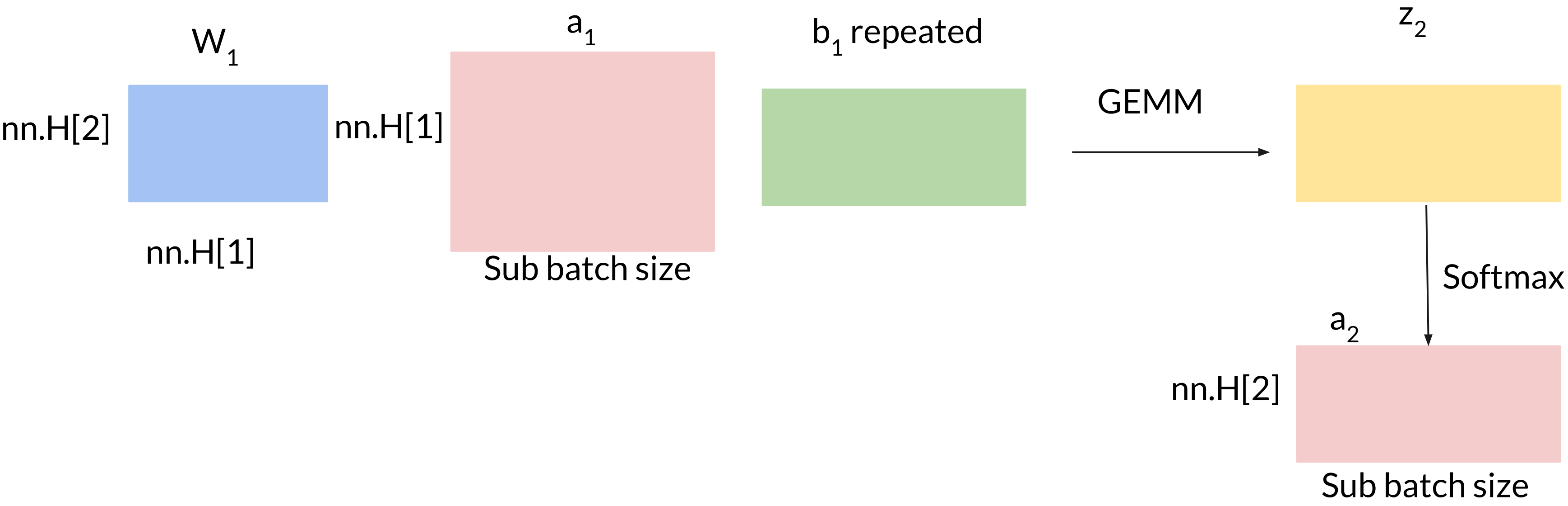


- Scatter each batch into sub-batches to each rank and its GPU before training
    - Do this once before training to avoid repeating overhead
    - Enough memory on GPU to hold all needed training data
- Same for y
- Copy initial weights and biases to GPU

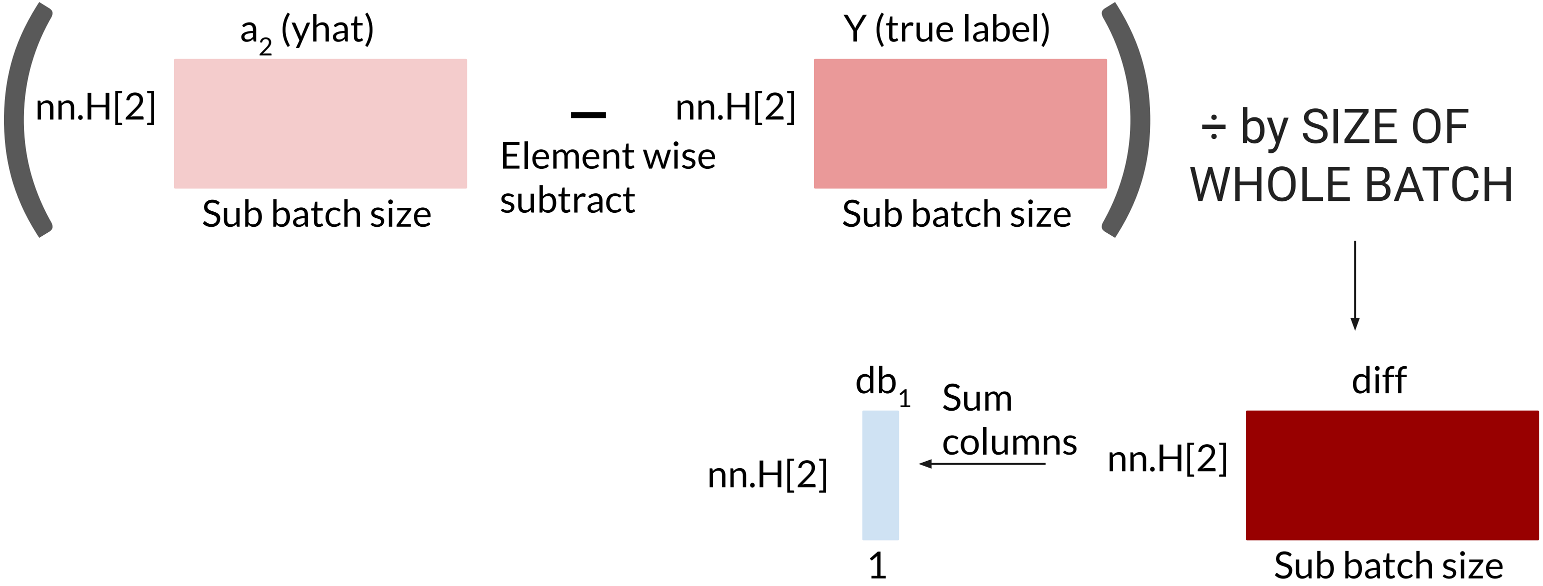
Feedforward



Feedforward

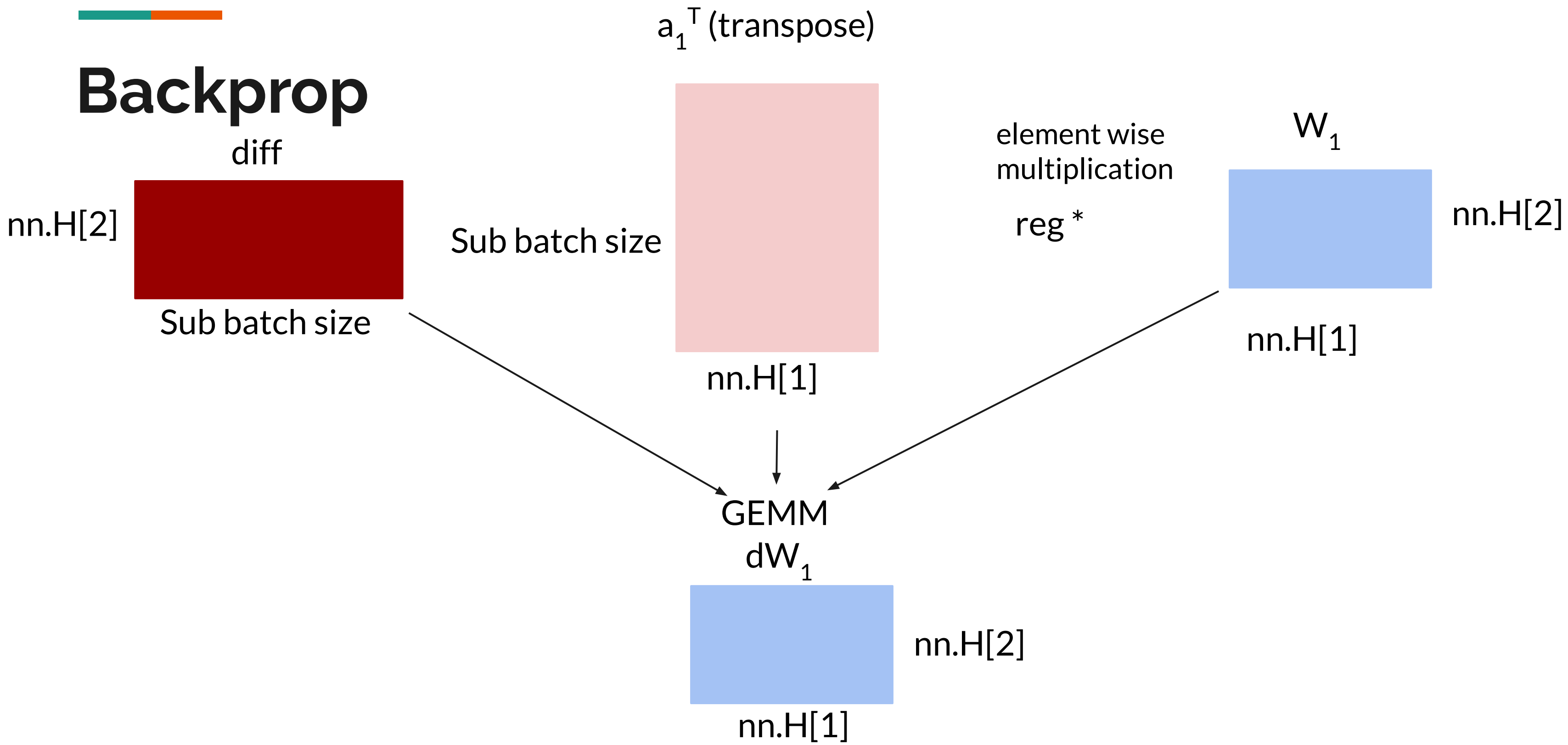


Backprop

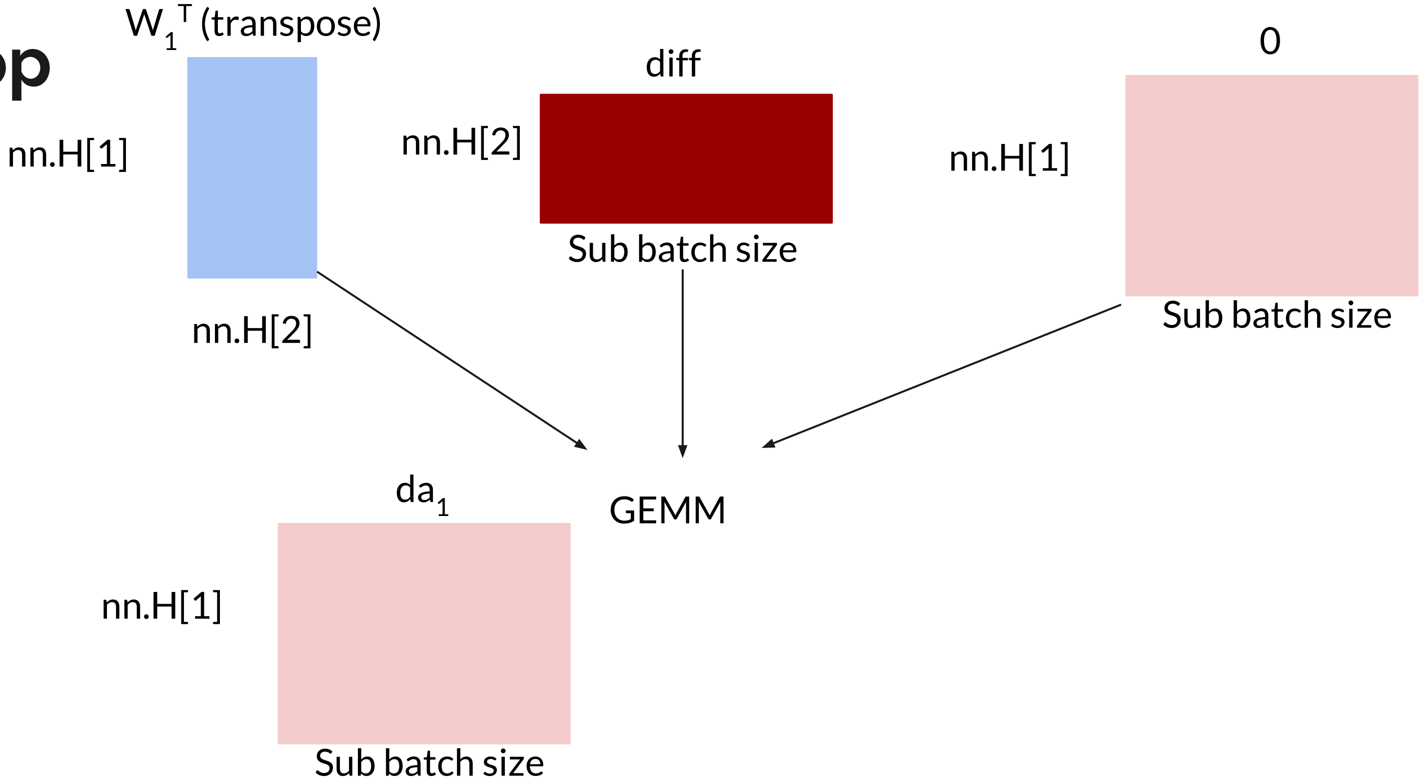


a_2 (yhat)
nn.H[2]
Sub batch size
–
Element wise subtract
Y (true label)
nn.H[2]
Sub batch size
÷ by SIZE OF WHOLE BATCH
diff
nn.H[2]
Sub batch size
Sum columns
db_1
nn.H[2]
1

Backprop

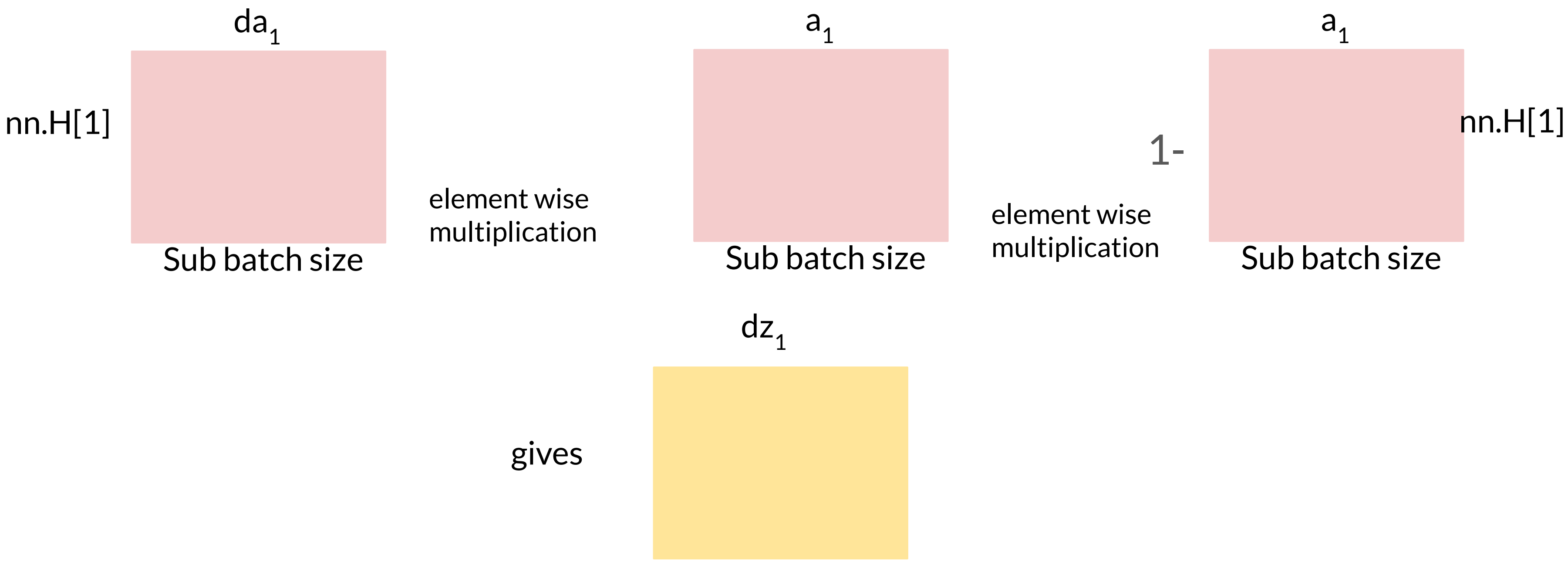

diff
nn.H[2]
Sub batch size
a_1^T (transpose)
Sub batch size
nn.H[1]
element wise
multiplication
reg *
W_1
nn.H[2]
nn.H[1]
GEMM
dW_1
nn.H[2]
nn.H[1]

Backprop



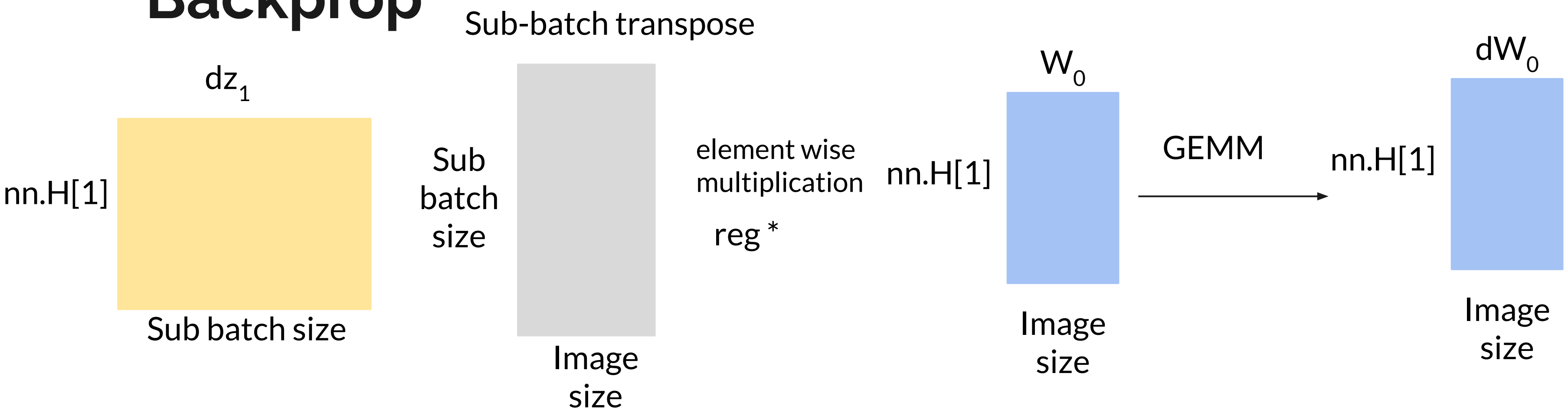
W_1^T (transpose)
nn.H[1]
nn.H[2]
diff
nn.H[2]
Sub batch size
0
nn.H[1]
Sub batch size
GEMM
da_1
nn.H[1]
Sub batch size

Backprop

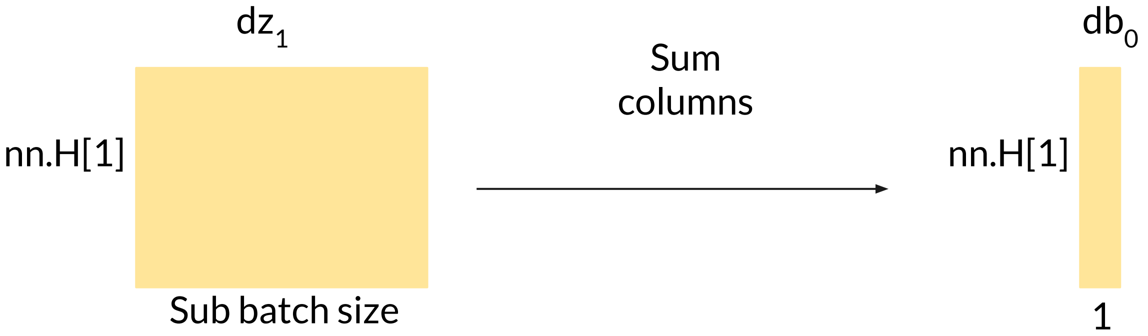


da_1
nn.H[1]
Sub batch size
element wise multiplication
a_1
Sub batch size
element wise multiplication
1-
a_1
nn.H[1]
Sub batch size
gives
dz_1

Backprop


Sub-batch transpose
dz1
nn.H[1]
Sub batch size
Sub
batch
size
Image
size
element wise
multiplication
reg *
W0
nn.H[1]
Image
size
GEMM
dW0
nn.H[1]
Image
size

Backprop



Gradient Descent

- Now we have all local gradients wrt sub-batch
- Transfer them back to CPU
- Perform Allreduce on all gradients so every process has complete gradients
- Copy gradients back to GPU
- Subtract gradients (* learning rate) from weights and biases
- And that’s one batch in one epoch!

Tips

- Gradients are the same size as the corresponding weights/biases
- Mostly, number of columns of matrices is the sub-batch size
- You can compare your results step by step with CPU result when developing/debugging
- Nail down the one operation/kernel that's incorrect
- If you are confident your kernel is correct, check input data

Shared Memory Recap

- Store data in shared memory to avoid going to global memory repeatedly
- Annotated by the __shared__ keyword
- Scope of access is within the thread block
- Need to map thread id to index for shared memory
- Need synchronization!
- Think bank conflict if performance is bad

Shared Memory Recap - Code Snippet

```
__shared__ double sm[SM_SIZE][SM_SIZE+1];

// thread index calculation

for (int i = 0; i < num_iter; i++){

    // write to shared memory

    __syncthreads();

    // do compution with data in sm

    __syncthreads();
}
```

Efficient GEMM - Hierarchical Tiling

- Many choices of efficient GEMM, you don't have to implement this particular one
- See part 1 of project writeup for more references
- Work on this after you have functioning, correct implementation
- This is high level, you need to figure out the implementation details!
- Images from reference for this GEMM algorithm

GEMM - Simplest Implementation

Matrix dimensions: A: (M, K), B: (K, N), C: (M, N)

```
for (int i = 0; i < M; ++i)
    for (int j = 0; j < N; ++j)
        for (int k = 0; k < K; ++k)
            C[i][j] += A[i][k] * B[k][j];
```

Read - inefficient, each row of A and column of B fetched multiple times and hard to reuse

GEMM - Accumulate Outer Product

```
for (int k = 0; k < K; ++k)       // K dimension now outer-most loop
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j)
            C[i][j] += A[i][k] * B[k][j];
```

Each column of A and row of B loaded exactly once

But writing to all c elements as we loop -> write inefficient

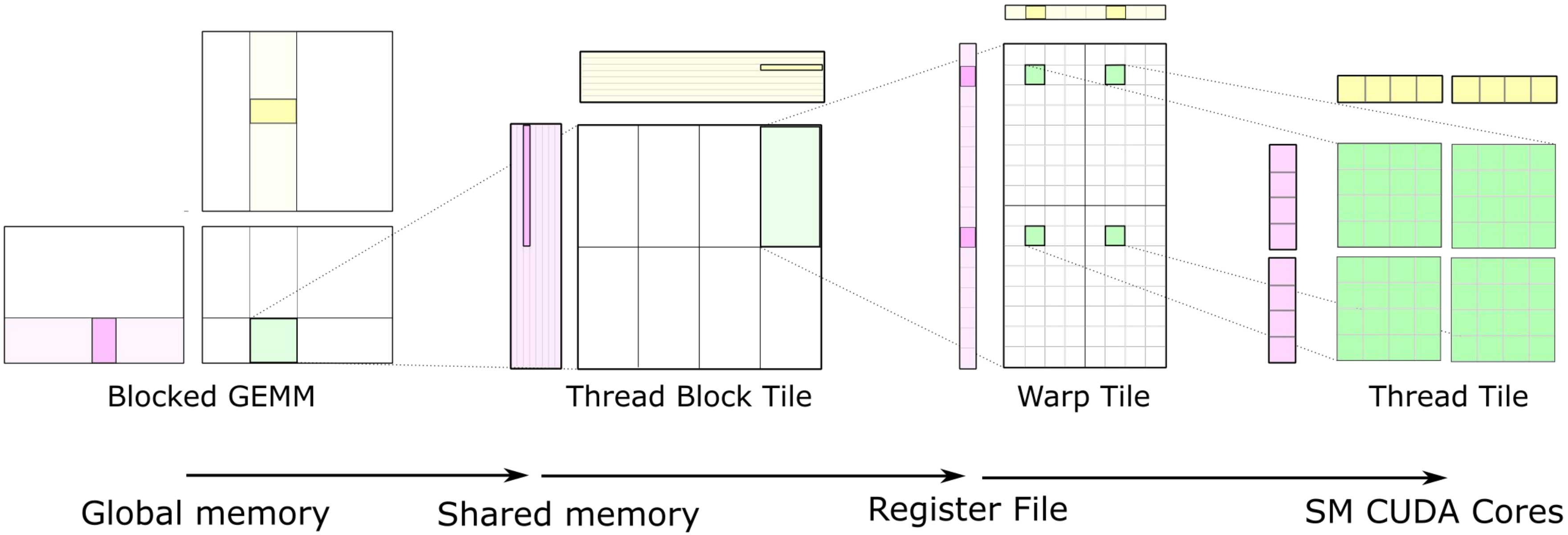
GEMM - Partition C

```
for (int m = 0; m < M; m += Mtile)                // iterate over M dimension
    for (int n = 0; n < N; n += Ntile)            // iterate over N dimension
        for (int k = 0; k < K; ++k)
            for (int i = 0; i < Mtile; ++i)       // compute one tile
                for (int j = 0; j < Ntile; ++j) {
                    int row = m + i;
                    int col = n + j;
                    C[row][col] += A[row][k] * B[k][col];
                }
```

Tile size of C small enough -> local accumulations fit on register

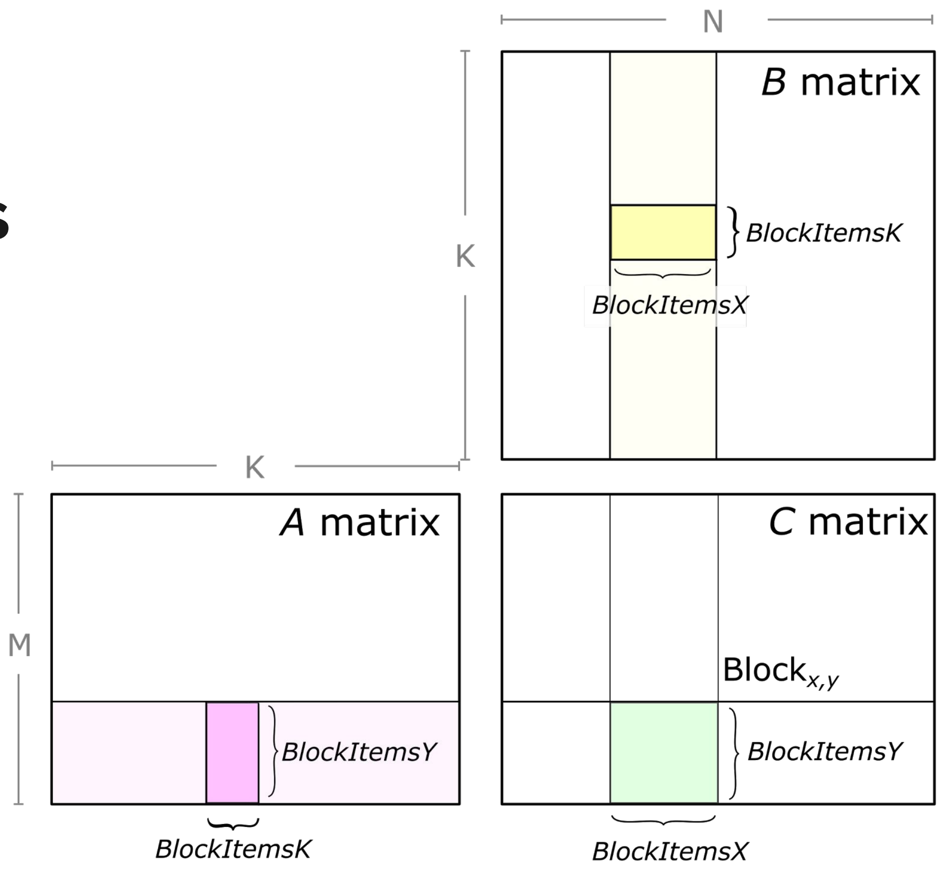
Outermost 2 loops trivially parallel

Hierarchical GEMM Structure - Overview



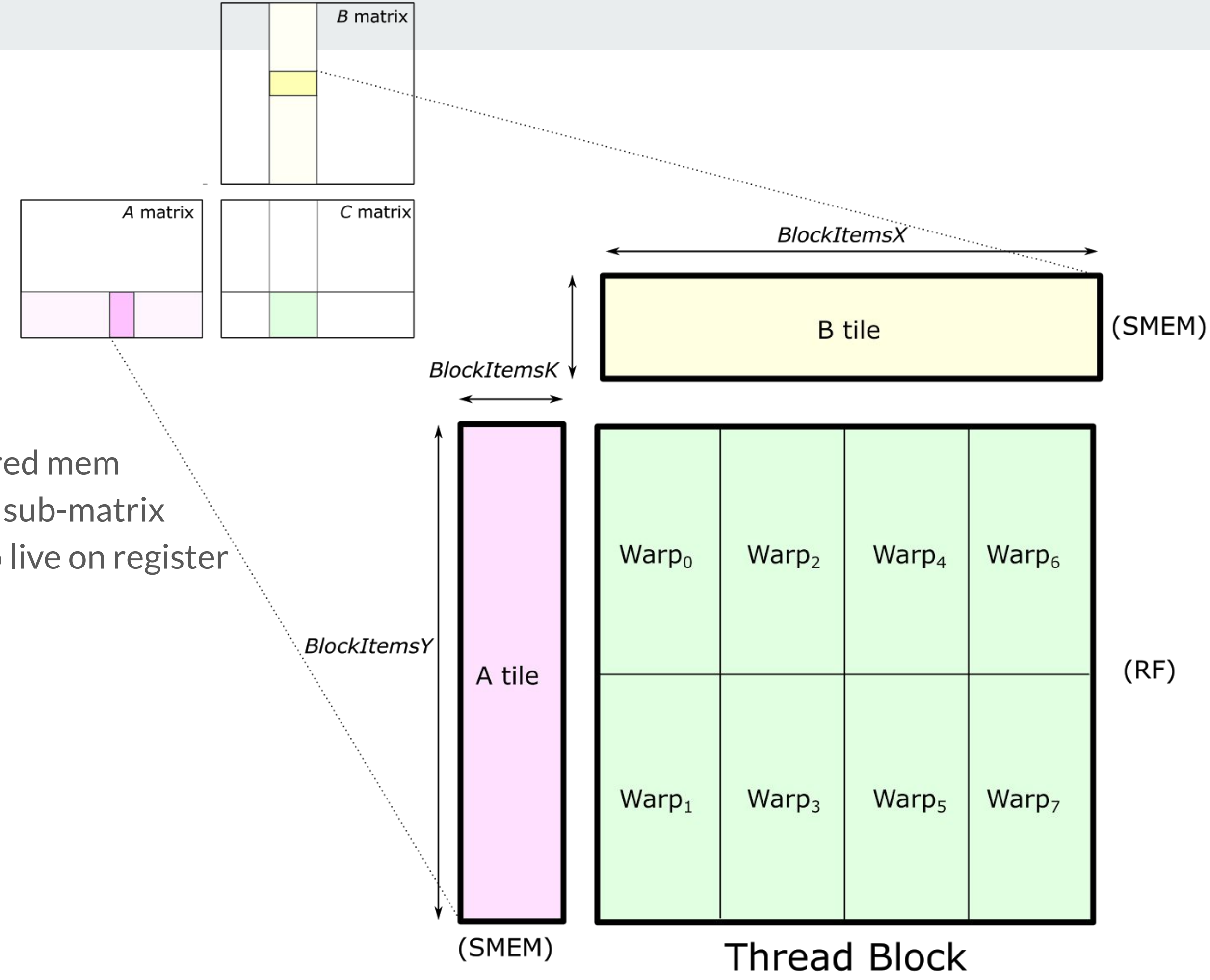
Tiling of Thread Blocks

- The block computes the green sub-matrix
- Need red region of A and yellow region of B
- Loop through K dimension to accumulate



Tiling of Warps

- One iteration in K
- Load A tile and B tile to shared mem
- Each warp compute a tile in sub-matrix
- Want accumulated result to live on register
- Need to tune block size



Warp Computation

Iterate over K dimension

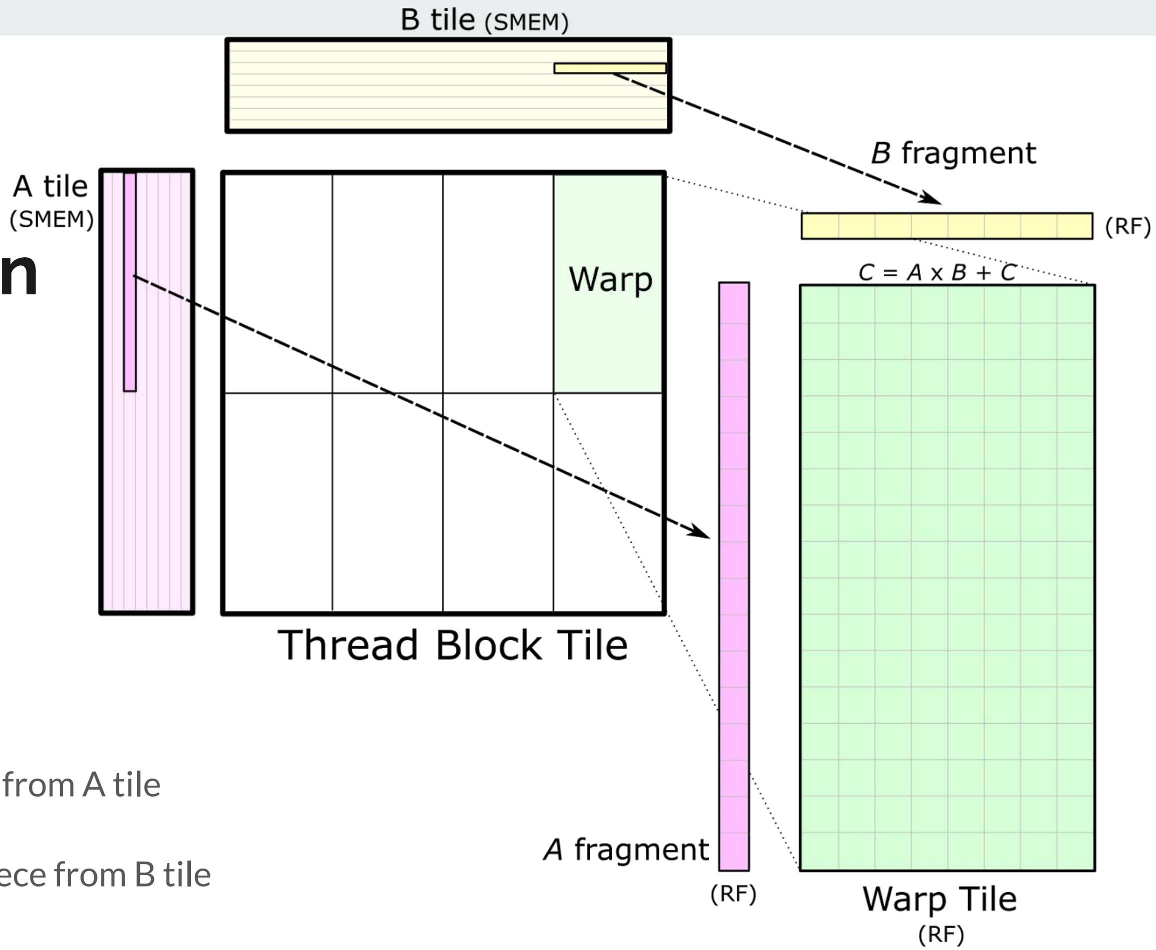
load pieces from A tile and B tile

Accumulate outer product

Data sharing:

wraps in the same row -> load same piece from A tile

wraps in the same column -> load same piece from B tile



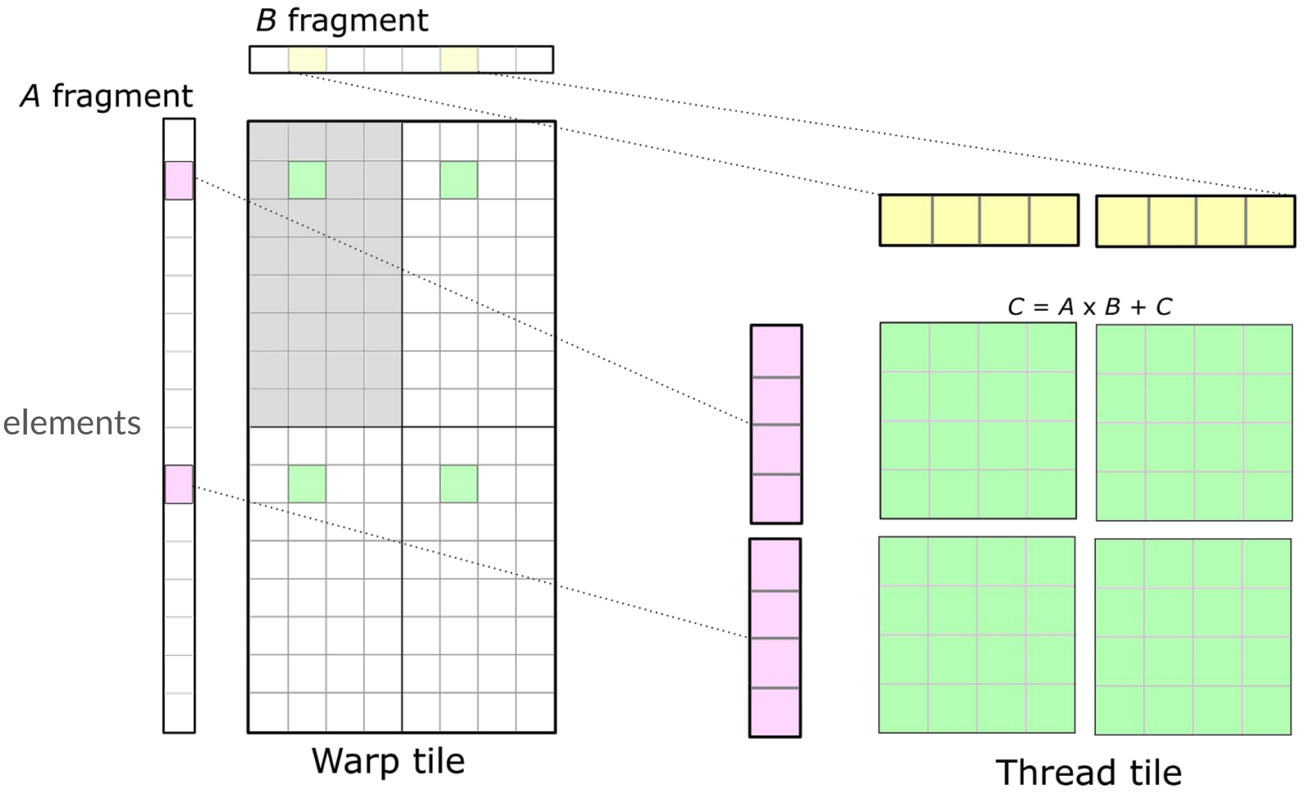
Thread Tiling

Each green square on left is 1 thread

Each accumulate a (small) number of elements

Threads in same row/cow ->

Fetch same A/B fragment



Summary

- Block tiling breaks up / parallelizes m and n loops
- Warp and thread tiling breaks up i and j loop
- Iterate over K dimension
- Use shared memory to improve read
- Keep accumulation result on register to improve write
- Configure warps and threads to promote data sharing
- Questions?