Module 3.5- Matrix Multiplication

Example 1: Sliding Average

Compute sliding average over a list

```
sub_size = 2
a = [4, 2, 5, 6, 2, 4]
out = [3, 3.5, 5.5, 4, 3]
```

Basic CUDA

Compute CUDA

```
def sliding(out, a):
    i = numba.cuda.blockIdx.x * TPB \
        + numba.cuda.threadIdx.x
    if i + sub_size < a.size:
        out[i] = 0
        for j in range(sub_size):
            out[i] += a[i + j]
        out[i] = out[i] / sub_size
```

Better CUDA

Two global reads per thread ::

```
def sliding(out, a):
    shared = numba.cuda.shared.array(TPB + sub_size)
    i = numba.cuda.blockIdx.x * TPB \
        + numba.cuda.threadIdx.x
    local_idx = numba.cuda.threadIdx.x
    if i + sub_size < a.size:
        shared[local_idx] = a[i]
        if local_idx < sub_size and i + TPB < a.size:
            shared[local_idx  + TPB] = a[i + TPB]
        numba.cuda.syncthreads()
        temp = 0
        for j in range(sub_size):
            temp += shared[local_idx + j]
        out[i] = temp / sub_size
```

Example 2: Reduction

Compute sum reduction over a list

```
a = [4, 2, 5, 6, 1, 2, 4, 1]
out = [26]
```

Algorithm

- Parallel Prefix Sum Computation
- Form a binary tree and sum elements

Associative Trick

Formula

$$a = 4 + 2 + 5 + 6 + 1 + 2 + 4 + 1$$

Same as

$$a = (((4 + 2) + (5 + 6)) + ((1 + 2) + (4 + 1)))$$

Thread Assignments

Round 1 (4 threads needed, 8 loads)

$$a = (((4 + 2) + (5 + 6)) + ((1 + 2) + (4 + 1)))$$

Round 2 (2 threads needed, 4 loads)

$$a = ((6 + 11) + (3 + 5))$$

Round 3 (1 thread needed, 2 loads)

$$a = (17 + 8)$$

Round 4

Quiz

Quiz

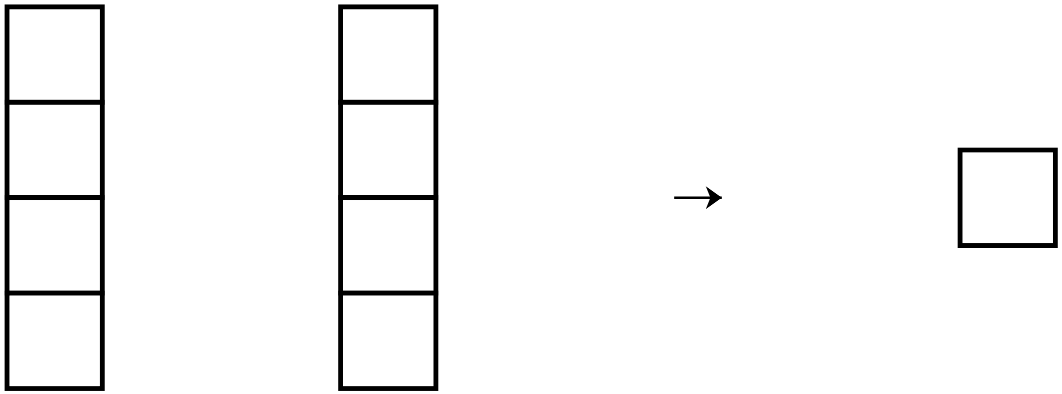
Motivation: Computing Splits

Linear Split

$$\mathrm{lin}(x; w, b) = x_1 \times w_1 + x_2 \times w_2 + b$$

Dot Product

$$x \cdot w = x_1 \times w_1 + x_2 \times w_2 + \ldots + x_n \times w_n$$

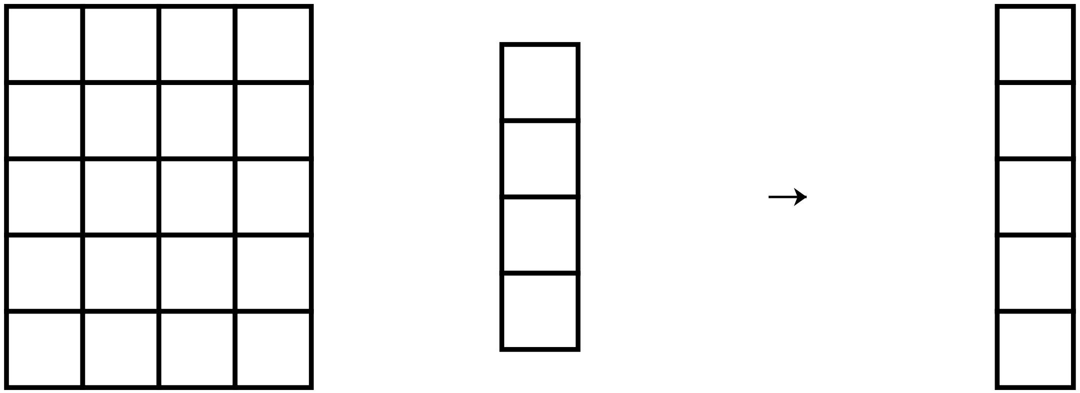


Dot Product in NN

- Computes 1 split for 1 data point

Batch Dot Product

Compute dot product for a *batch* of examples $x^1, \ldots, x^J$



Batch Dot Product in NN

- Computes 1 split for 5 data points

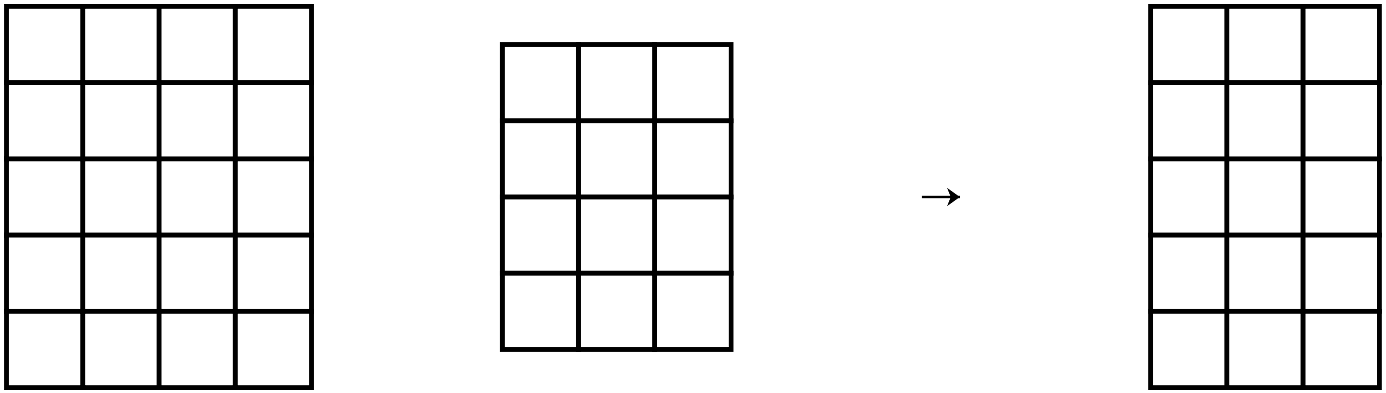
Math View

$$
\begin{aligned}
\mathrm{lin}(x; w, b) &= x_1 \times w_1 + x_2 \times w_2 + b \\
h_1 &= \mathrm{ReLU}(\mathrm{lin}(x; w^0, b^0)) \\
h_2 &= \mathrm{ReLU}(\mathrm{lin}(x; w^1, b^1)) \\
m(x_1, x_2) &= \mathrm{lin}(h; w, b)
\end{aligned}
$$

Parameters: $w_1, w_2, w_1^0, w_2^0, w_1^1, w_2^1, b, b^0, b^1$

Batch Dot Product for each split

- Computes 3 splits for 5 data points (15 dot products)

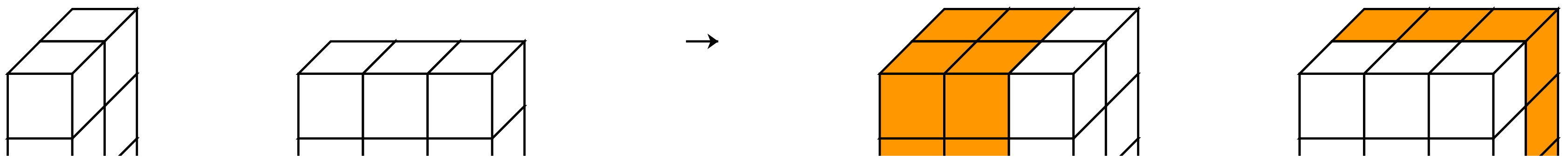


Matrix Multiply

- Key algorithm for deep learning
- Has properties of both zip and reduce

Matmul

- Computed this in Module 2 already



Operator Fusion

User API

- Basic mathematical operations
- Chained together as boxes with broadcasting
- Optimize within each individually

Fusion

- Optimization across operator boundary
- Save speed or memory in by avoiding extra forward/backward
- Can open even great optimization gains

Automatic Fusion

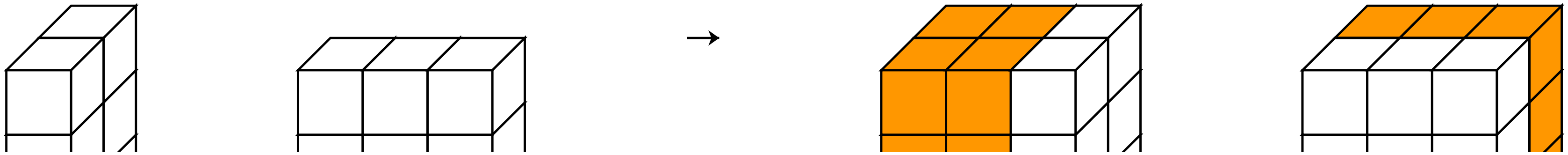
- Compiled language can automatically fuse operators
- Major area of research
- Example: TVM, XLA, ONXX

Automatic Fusion

Manual Fusion

- Utilize a pre-fused operator when needed
- Standard libraries for implementations

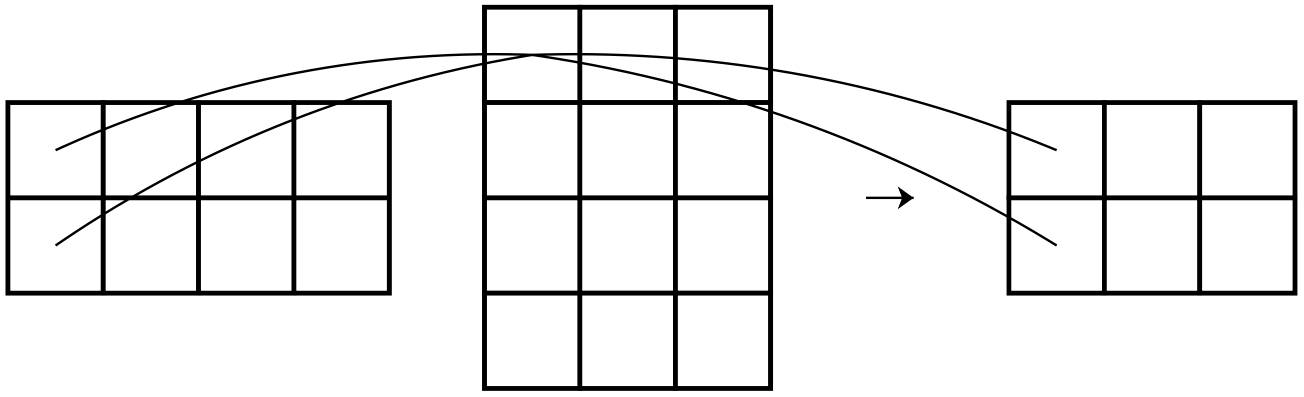
Matmul Simple



Advantages

- No three dimensional intermediate
- No `save_for_backwards`
- Can use core matmul libraries (in the future)

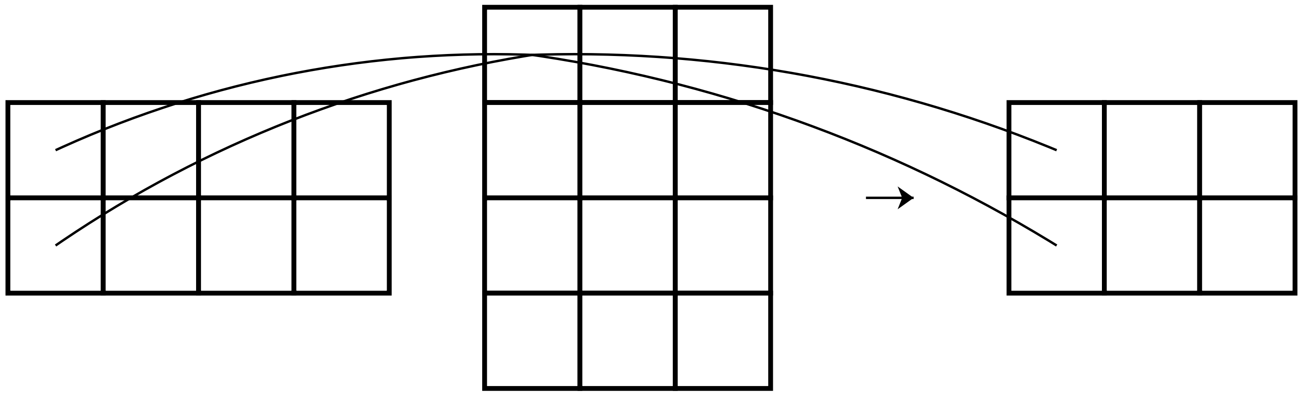
Computations



Starter Code

- Walk through output.
- Find row and column of input
- Simultaneous zip / reduce.

Example: Matmul



Simple Matmul

```
A.shape == (I, J)
B.shape == (J, K)
out.shape == (I, K)
```

Simple Matmul Pseudocode

```
for outer_index in out.indices():
  for inner_val in range(J):
      out[outer_index] += A[outer_index[0], inner_val] * \
                          B[inner_val, outer_index[1]]
```

Compare to zip / reduce

Code

```
ZIP STEP
  C = zeros(broadcast_shape(A.view(I, J, 1), B.view(1, J, K)))
  for C_outer in C.indices():
      C[C_out] = A[outer_index[0], inner_val] * \
                 B[inner_val, outer_index[1]]
  REDUCE STEP
  for outer_index in out.indices():
     for inner_val in range(J):
        out[outer_index] = C[outer_index[0], inner_val,
                             outer_index[1]]
```

Complexities

- Indices to strides
- Minimizing index operations
- Broadcasting

Matmul Speedups

What can be parallelized?

```
for outer_index in out.indices():
    for inner_val in range(J):
        out[outer_index] += A[outer_index[0], inner_val] * \
                            B[inner_val, outer_index[1]]
```

CUDA Matrix Mul

CUDA Matrix Mul

Basic CUDA ::

```
def mm_simple(out, a, b, K):
    i = numba.cuda.blockIdx.x * TPB \
        + numba.cuda.threadIdx.x
    j = numba.cuda.blockIdx.y * TPB \
        + numba.cuda.threadIdx.y
    for k in range(K):
        out[i, j] += a[i, k] * b[k, j]
```

Data Dependencies

- Which elements does out[i, j] depend on?
- How many are there?

Dependencies

Square Matrix

- Assume a, b, out are all 2x2 matrices
- Idea -> copy all needed values to shared?

Basic CUDA - Square Small

Basic CUDA ::

```
def mm_shared1(out, a, b, K):
    ...
    sharedA[local_i, local_j] = a[i, j]
    sharedB[local_i, local_j] = b[i, j]
    ...
    for k in range(K):
        t += sharedA[local_i, k] * sharedB[k, local_j]
    out[i, j] = t
```

Data Dependencies

- If the matrix is big, out[i, j] may depend on 1000s of elements.
- Grows larger than block size.
- Idea: Move the shared memory.

Diagram

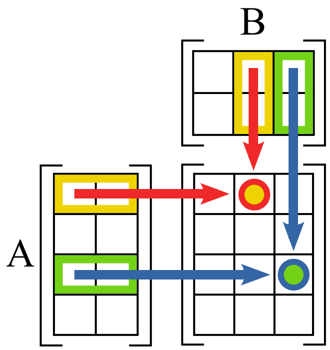
Large Square

Basic CUDA - Square Large

Basic CUDA ::

```
def mm_shared1(out, a, b, K):
    ...
    for s in range(0, K, TPB):
        sharedA[local_i, local_j] = a[i, s + local_j]
        sharedB[local_i, local_j] = b[s + local_i, j]
        ...
        for k in range(TPB):
            t += sharedA[local_i, k] * sharedB[k, local_j]
    out[i, j] = t
```

Non-Square - Dependencies



Challenges

- How do you handle the different size of the matrix?
- How does this interact with the block size?