Module 3.2 - CUDA

Why are Python (and friends) ”slow”?

- Function calls
- Types
- Loops

Function Calls

- Function calls are not free
- Checks for args, special keywords andm lists
- Methods check for overrides and class inheritance

Types

Critical code

```
out[o] = in_storage[j] + 3
```

- Doesn't know type of `in_storage[j]`
- May need to coerce 3 to float or raise error
- May even call `__add__` or `__ladd__`!

How does it work?

Work

```
def my_code(x, y):
     for i in range(100):
         x[i] = y + 20
  ...
  my_code(x, y)
  fast_my_code = numba.njit()(my_code)
  fast_my_code(x, y)
  fast_my_code(x, y)
```

Notebook

Colab Notebook

Terminology : JIT Compiler

- Just-in-time
- Waits until you call a function to compile it
- Specializes code based on the argument types given.

Parallel Range

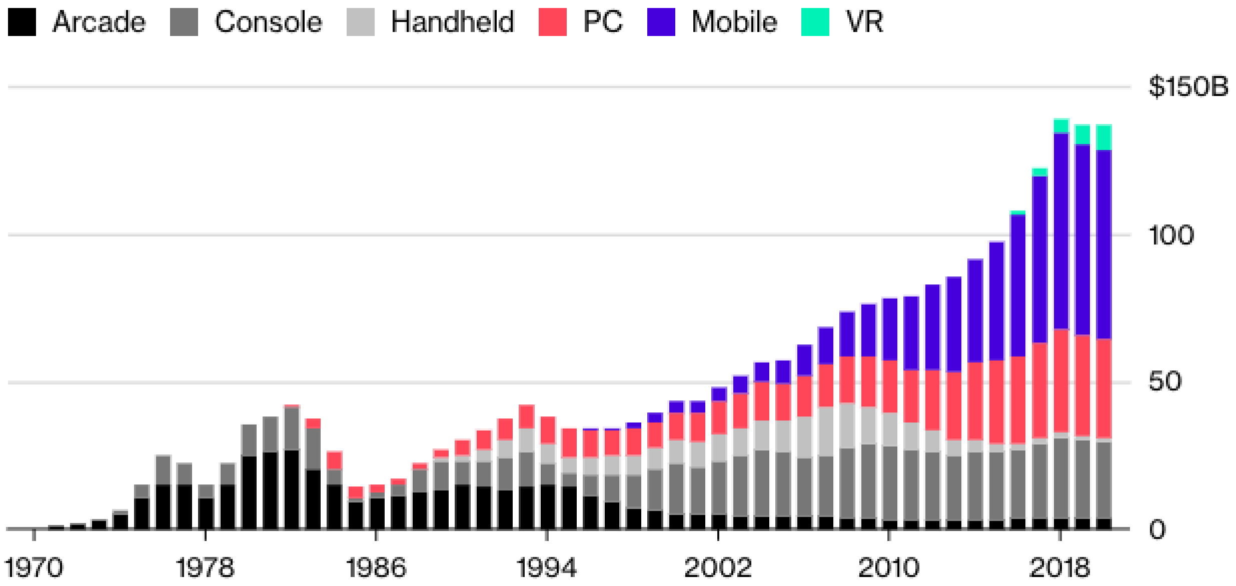
- Replace `for` loops with parallel version
- Tells compiler it can run in any order
- Be careful! Ideally these loops don't change anything

Quiz

CUDA

- NVidia's programming language for GPU
- Compute Unified Device Architecture
- Like standard programming but in parallel

NVidia Structure



Source: Pelham Smithers

NVidia Structure

Main Driver

- Custom shader languages
- Graphics targeted operations

General Purpose GPUs

- NVidia: can we make these programmable
- ~2008: CUDA langauge

Machine Learning

- Growth in ML parallels GPU development
- Major deep learning results require GPU
- All modern training is on GPU (or more)

Is this enough?

BERT-Large Training Times on GPUs

Time	System	Number of Nodes	Number of V100 GPUs
47 min	DGX SuperPOD	92 x DGX-2H	1,472
67 min	DGX SuperPOD	64 x DGX-2H	1,024
236 min	DGX SuperPOD	16 x DGX-2H	256

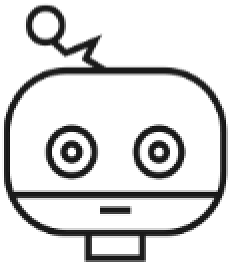
GPUs

Challeges

- Hard to code for directly.
- Particularly hard to code efficiently.
- Goal: hide complexity from users.

Threads

thread

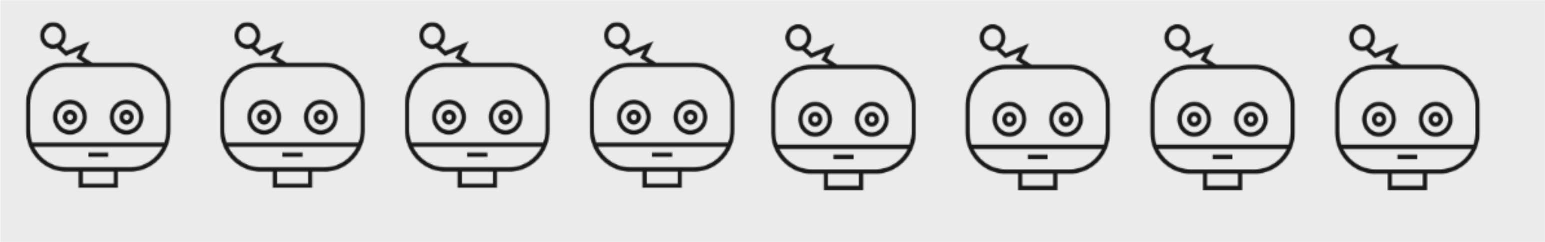


Thread code

```
def add(a, b):
    b = a + 10
cuda_add = numba.cuda.jit()(add)

cuda_add[1, 1](a, b)
```

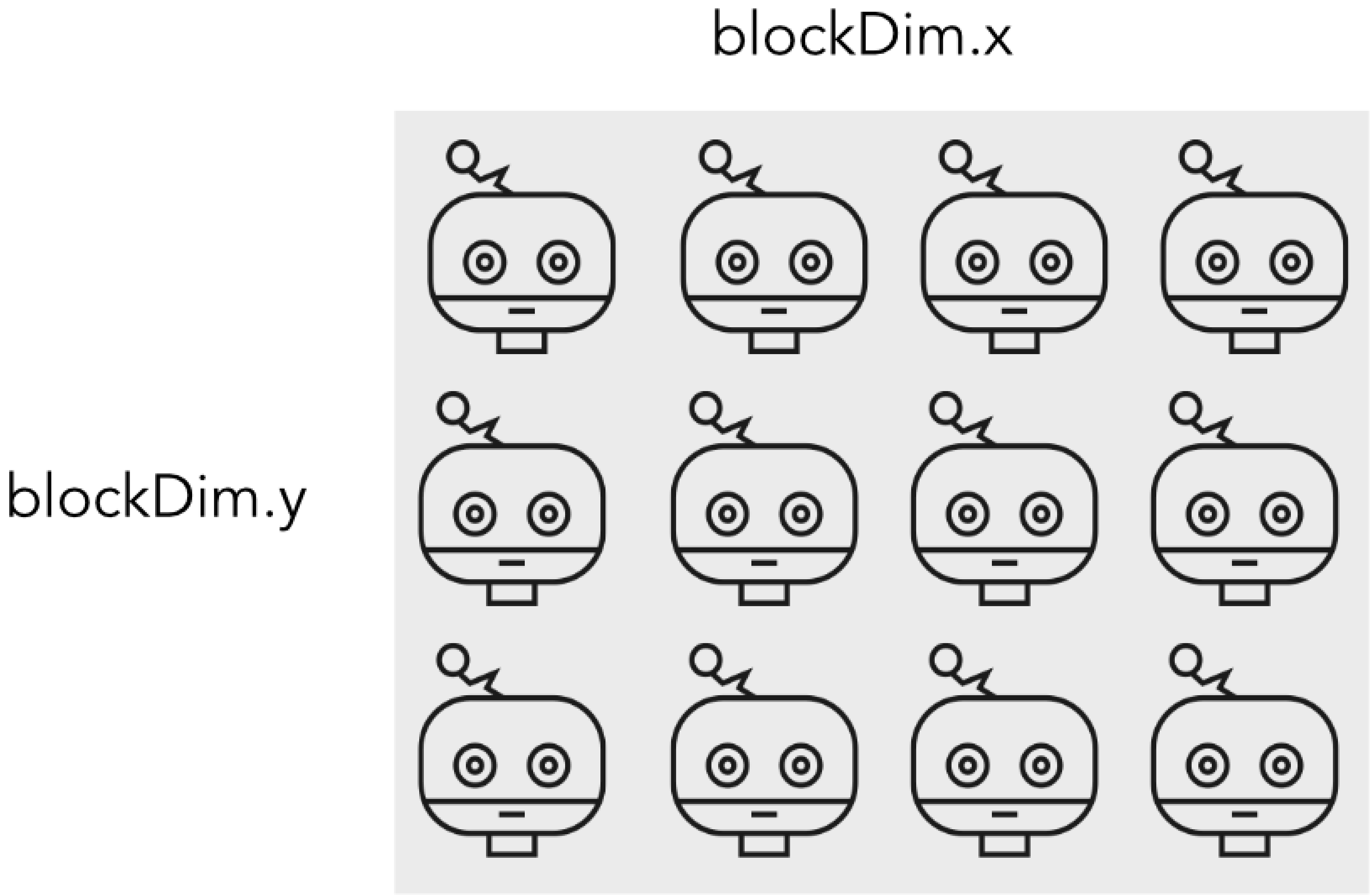
block



Threads code

def add(a, b): b = a + 10 cuda_add = numba.cuda.jit()(add)

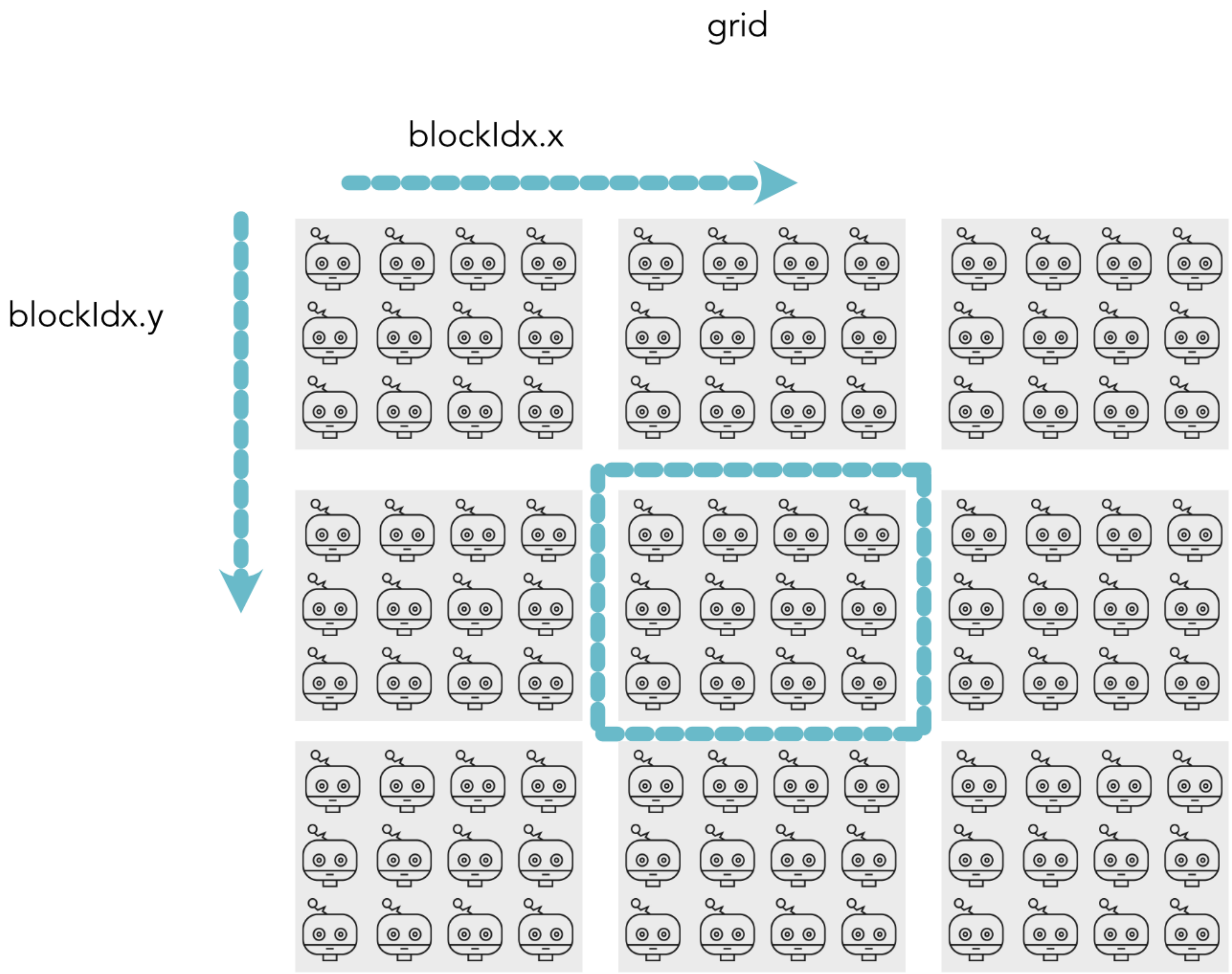
cuda_add1, 10


blockDim.x
blockDim.y

Threads code

```
def add(a, b):
    b = a + 10
cuda_add = numba.cuda.jit()(add)

cuda_add[1, (10, 10)](a, b)
```


grid
blockIdx.x
blockIdx.y

Block code

```
def add(a, b):
    b = a + 10
  cuda_add = numba.cuda.jit()(add)

  cuda_add[(10, 10), (10, 10)](a, b)
```

Check

```python
def printer(a):
    print("hello!")
    a[:] = 10 + 50
  printer = numba.cuda.jit()(printer)
  a = numpy.zeros(10)
  printer[10, 10](a)
```

Output

Output

```
hello!
hello!
hello!
hello!
hello!
...
```

Stack

- Threads: Run the code
- Block: Groups "close" threads
- Grid: All the thread blocks
- Total Threads: `threads_per_block` x `total_blocks`

Thread Names

Printing code

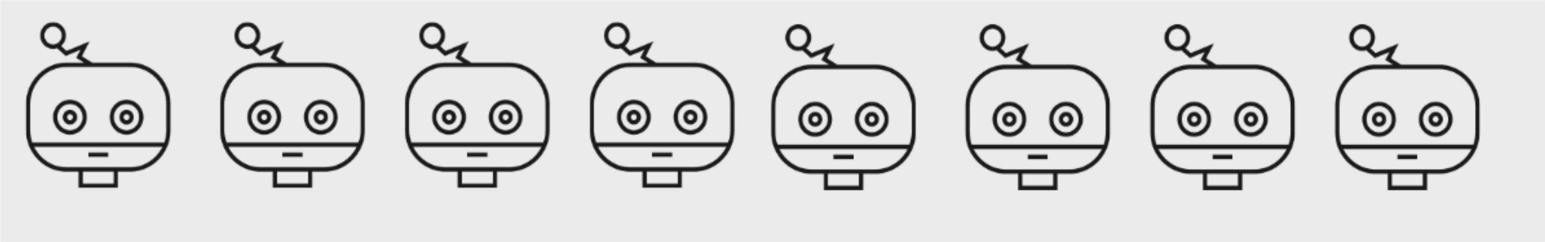
```
def printer(a):
    print(numba.cuda.threadIdx.x, numba.cuda.threadIdx.y)
    a[:] = 10 + 50
printer = numba.cuda.jit()(printer)
a = numpy.zeros(10)
printer[1, (10, 10)](a)
```

Output

Output

```
6 3
7 3
8 3
9 3
0 4
1 4
2 4
3 4
4 4
```

block



Thread Names

```python
def printer(a):
    print(numba.cuda.blockIdx.x,
          numba.cuda.threadIdx.x, numba.cuda.threadIdx.y)
    a[:] = 10 + 50
  printer = numba.cuda.jit()(printer)
  a = numpy.zeros(10)
  printer[10, (10, 10)](a)
```

Output

Output ::

```
7 6 9
7 7 9
7 8 9
7 9 9
2 6 9
2 7 9
```

What's my name?

Name ::

```
BLOCKS_X = 32
  BLOCKS_Y = 32
  THREADS_X = 10
  THREADS_Y = 10
  def fn(a):
    x = numba.cuda.blockIdx.x * THREADS_X + numba.cuda.threadIdx.x
    y = numba.cuda.blockIdx.y * THREADS_Y + numba.cuda.threadIdx.y
    ...
  fn = numba.cuda.jit()(fn)
  fn[(BLOCKS_X, BLOCKS_Y), (THREADS_X, THREAD_Y)](a)
```

Simple Map

```
BLOCKS_X = 32
  THREADS_X = 32
  def fn(out, a):
    x = numba.cuda.blockIdx.x * THREADS_X + numba.cuda.threadIdx.x
    if x >=0 and x < a.size:
      out[x] = a[x] + 10
  fn = numba.cuda.jit()(fn)
  fn[BLOCKS_X, THREADS_X](a)
```

Guards

Guards

```
x = numba.cuda.blockIdx.x * BLOCKS_X + numba.cuda.threadIdx.x
if x >=0 and x < a.size:
```

Colab

- https://colab.research.google.com/drive/1nzH-BHZi-LYK9Ee4t3xvfSr73-qaASwQ#scrollTo=mVmikf3wrekV

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

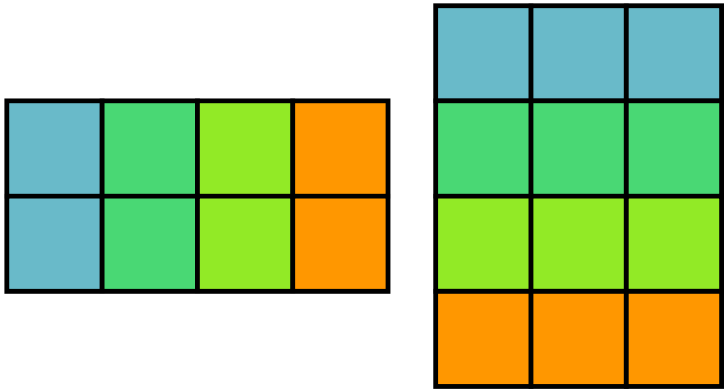
- Compiled language can automatically fuse operators
- Major area of research
- Example: TVM, XLA, ONXX

Automatic Fusion

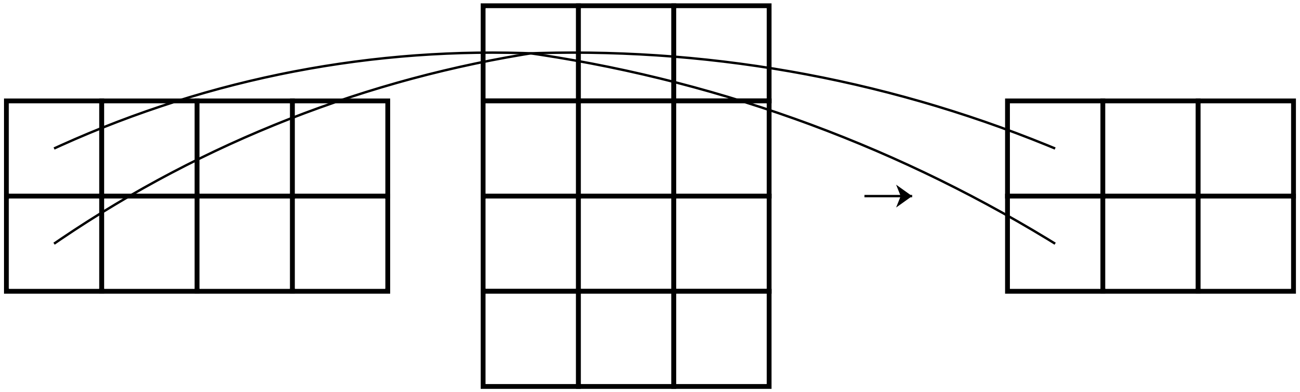
Manual Fusion

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

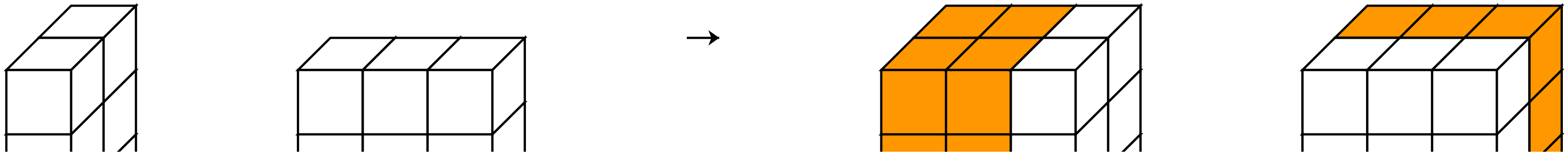
Example: Matmul



Example: Matmul



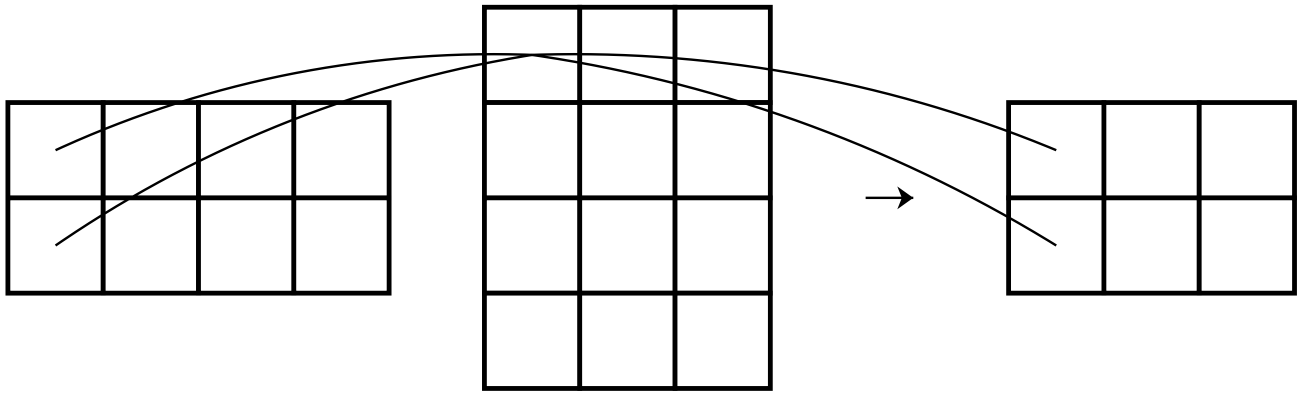
Matmul Simple



Advantages

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

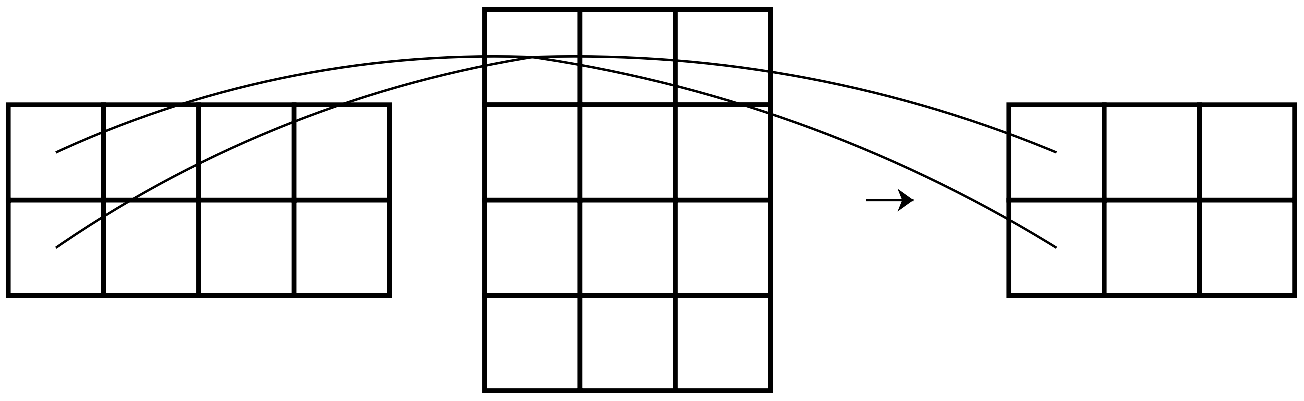
Computations



Starter Code

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

Example: Matmul



Matrix Multiply

$$
\begin{aligned}
f(M, N) &= MN \\
g'_M(f(M, N)) &= dN^T \\
g'_N(f(M, N)) &= M^T d
\end{aligned}
$$

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]]
```

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]]
```

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]]
```

Matrix Multiply

$$
\begin{aligned}
f(M, N) &= MN \\
g'_M(f(M, N)) &= \mathrm{grad}_{\mathrm{out}} N^\top \\
g'_N(f(M, N)) &= M^\top \mathrm{grad}_{\mathrm{out}}
\end{aligned}
$$

Optimizations

- Avoiding indexing
- Where to put parallelism?