# The argument for Decoupled Triton

Quinn Pham

# Fused Multiply-Add Example

# Fused Multiply-Add Example

## Decoupled Triton

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
Func f;
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

# Fused Multiply-Add Example

**Decoupled Triton**

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

**Triton Python**

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
  for x_iter in range(0, x, 1):
    for y_iter in range(0, y, 1):
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
  for x_iter in range(0, x, 1):
    for y_iter in range(0, y, 1):
      a = tl.load(a_ptr + x_iter * y + y_iter)
      b = tl.load(b_ptr + x_iter * y + y_iter)
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
  for x_iter in range(0, x, 1):
    for y_iter in range(0, y, 1):
      a = tl.load(a_ptr + x_iter * y + y_iter)
      b = tl.load(b_ptr + x_iter * y + y_iter)
      f = a * s + b
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
  for x_iter in range(0, x, 1):
    for y_iter in range(0, y, 1):
      a = tl.load(a_ptr + x_iter * y + y_iter)
      b = tl.load(b_ptr + x_iter * y + y_iter)
      f = a * s + b
      tl.store(f_ptr + x_iter * y + y_iter, f)
```

# Fused Multiply-Add Example

## Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.compile_to_kernel();
```

## Triton Python

```
import triton
import triton.language as tl

@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y):
  for x_iter in range(0, x, 1):
    for y_iter in range(0, y, 1):
      a = tl.load(a_ptr + x_iter * y + y_iter)
      b = tl.load(b_ptr + x_iter * y + y_iter)
      f = a * s + b
      tl.store(f_ptr + x_iter * y + y_iter, f)

...
f_grid = (1,)
f_kernel[f_grid](a, b, s, f, x, y)
```

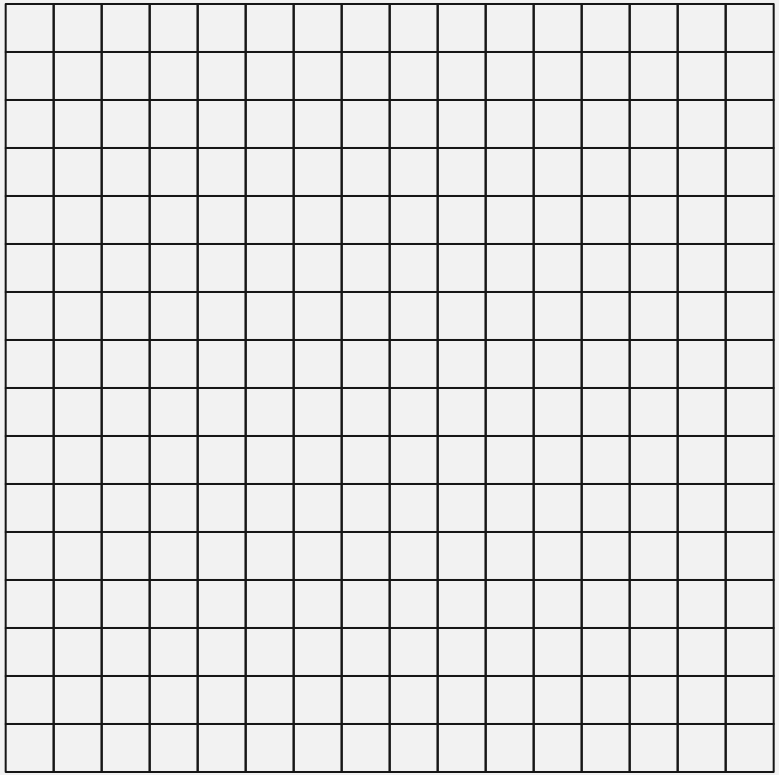
# Programmability

What advantages does defining kernels in Decoupled Triton have over writing Triton Kernels by hand?

Manually modifying an existing Triton kernel according to a schedule is non-trivial due to index remappings, offset calculations, and other low-level details

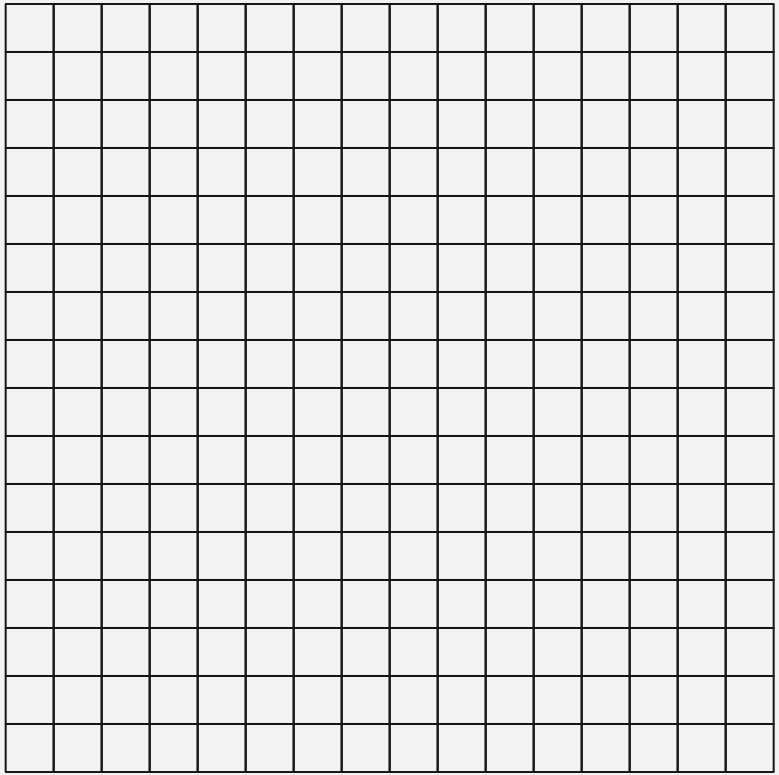
# Blocking

# Blocking



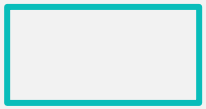
# Blocking

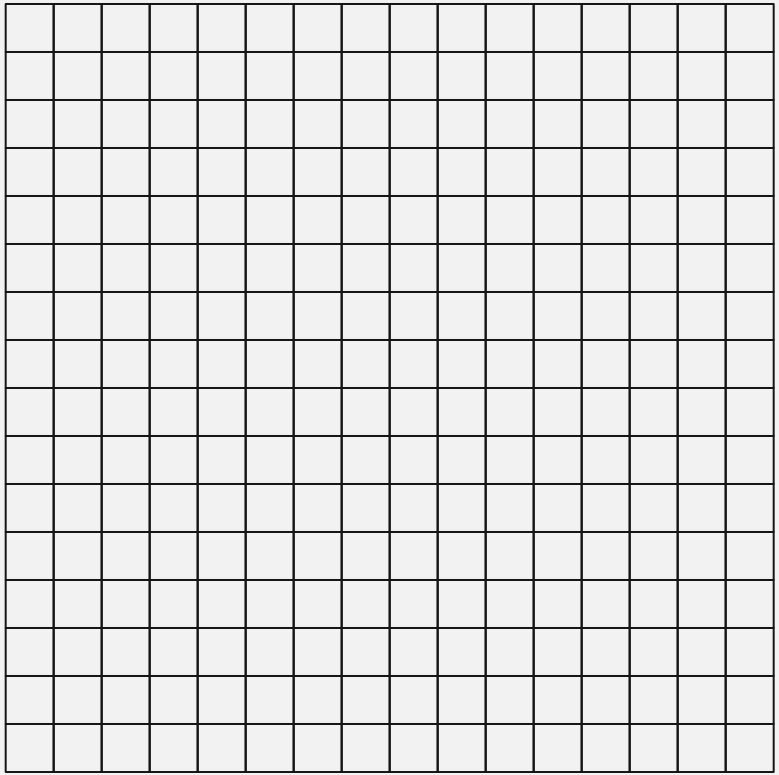
```
f.block(x:2, y:4)
```



# Blocking

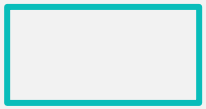
```
f.block(x:2, y:4)
```

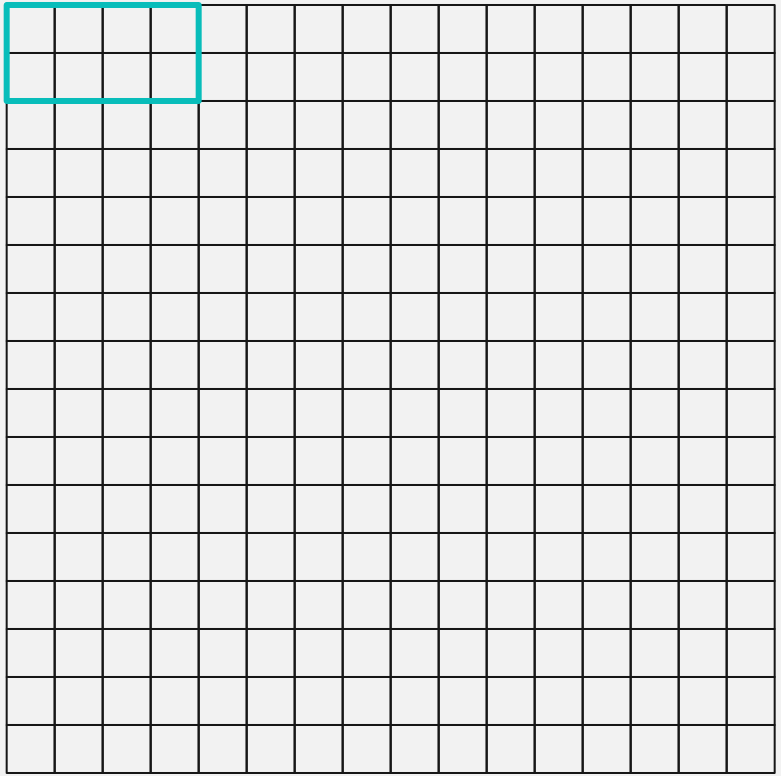




# Blocking

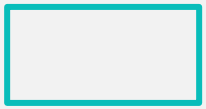
```
f.block(x:2, y:4)
```

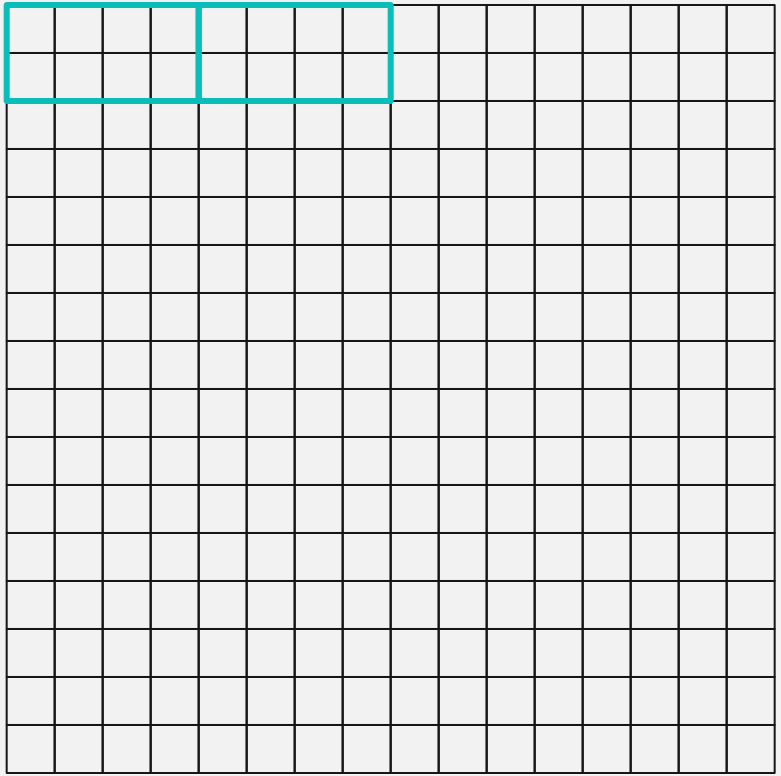




# Blocking

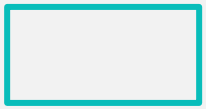
```
f.block(x:2, y:4)
```

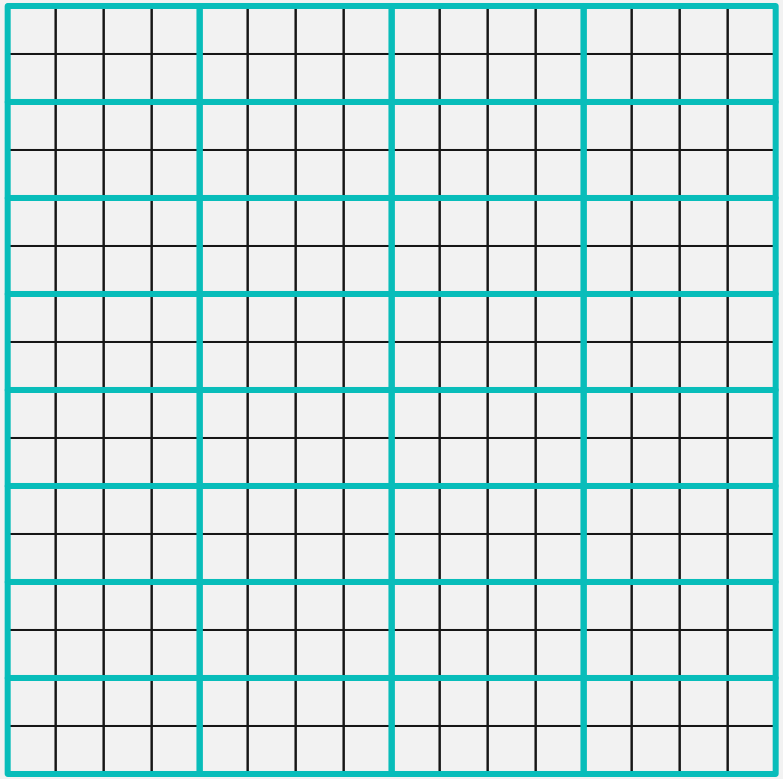




# Blocking

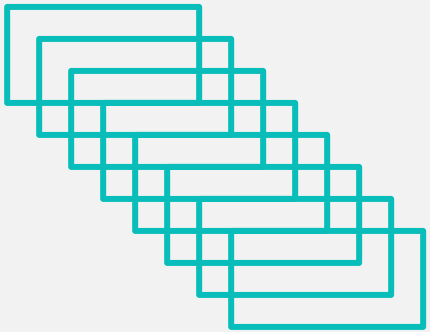
```
f.block(x:2, y:4)
```

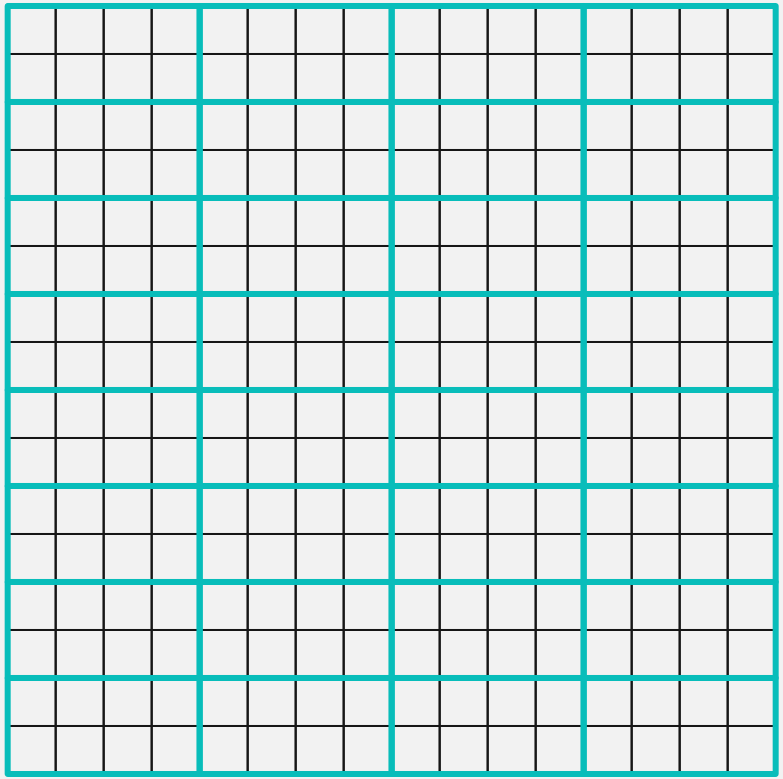




# Blocking

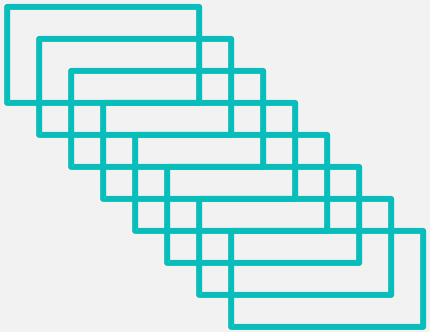
```
f.block(x:2, y:4)
```

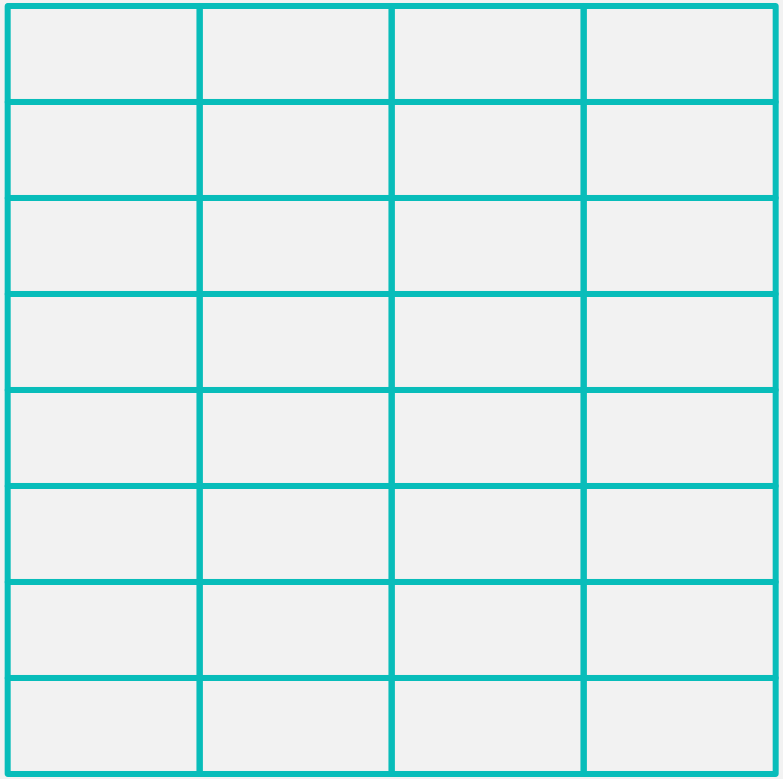




# Blocking

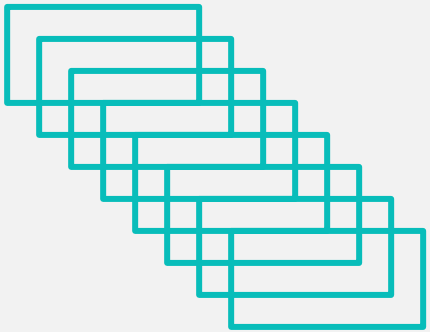
```
f.block(x:2, y:4)
```





# Blocking

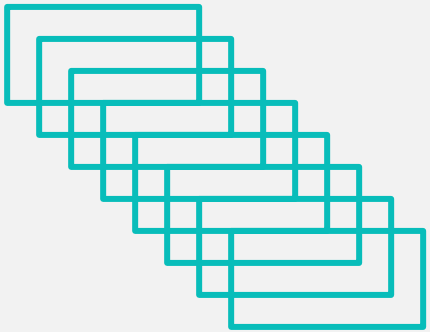
```
f.block(x:2, y:4)
```



| | | | |
|---|---|---|---|
| 0, 0 | 0, 1 | 0, 2 | 0, 3 |
| 1, 0 | 1, 1 | 1, 2 | 1, 3 |
| 2, 0 | 2, 1 | 2, 2 | 2, 3 |
| 3, 0 | 3, 1 | 3, 2 | 3, 3 |
| 4, 0 | 4, 1 | 4, 2 | 4, 3 |
| 5, 0 | 5, 1 | 5, 2 | 5, 3 |
| 6, 0 | 6, 1 | 6, 2 | 6, 3 |
| 7, 0 | 7, 1 | 7, 2 | 7, 3 |

# Blocking

```
f.block(x:2, y:4)
```



| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Blocking in Triton

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for x_iter in range(0, x_BLOCK_SIZE, 1):
    for y_iter in range(0, y_BLOCK_SIZE, 1):
      a = tl.load(a_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      b = tl.load(b_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter)
      f = a * s + b
      tl.store(f_ptr + (x_block_start + x_iter) * y + y_block_start + y_iter, f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4)
```

# Blocking in Decoupled Triton

# Blocking in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.compile_to_kernel();
```

# Blocking in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.compile_to_kernel();
```

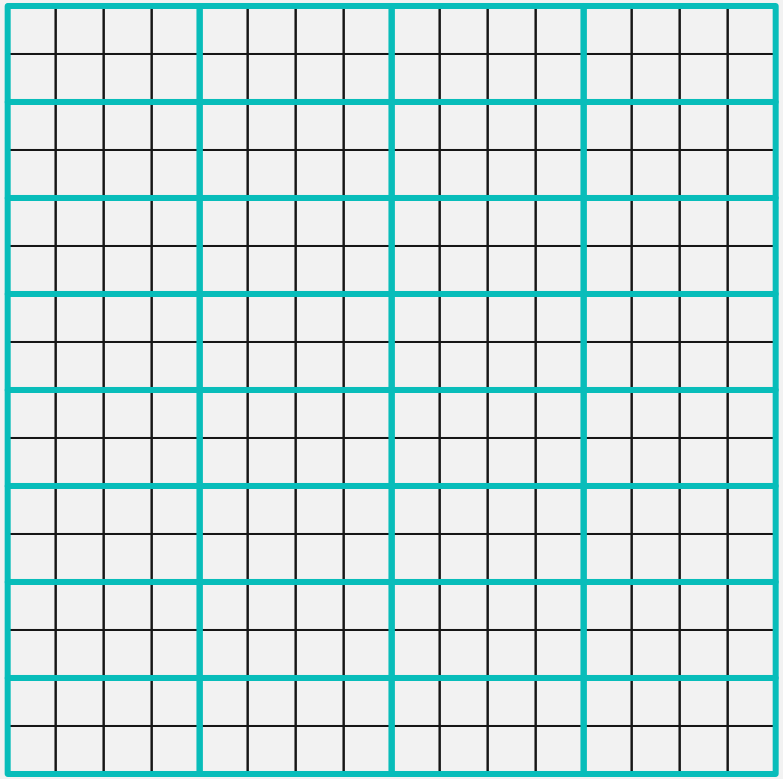
# Blocking in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.compile_to_kernel();
```

# Tensorization

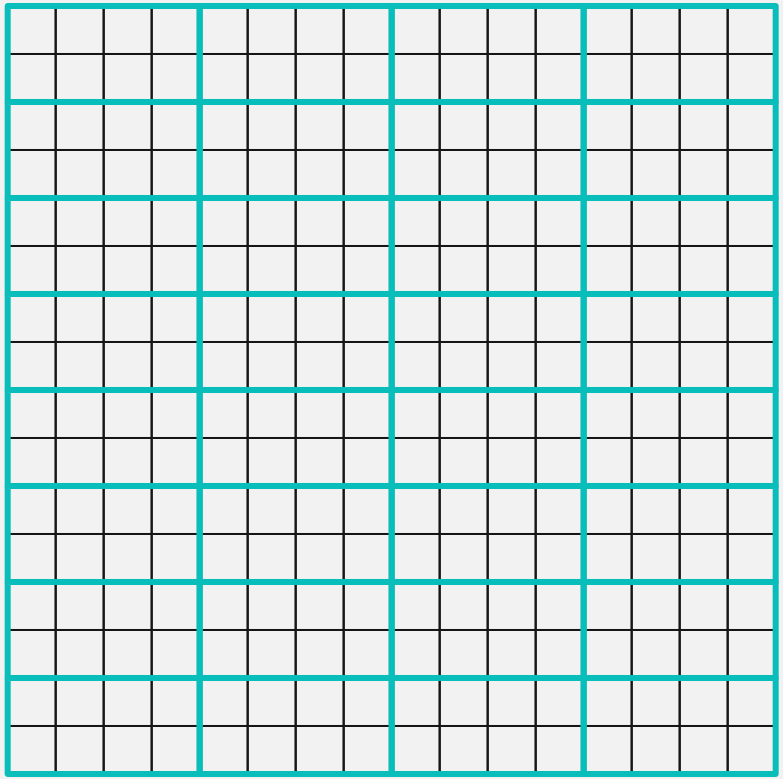
# Tensorization

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)
```



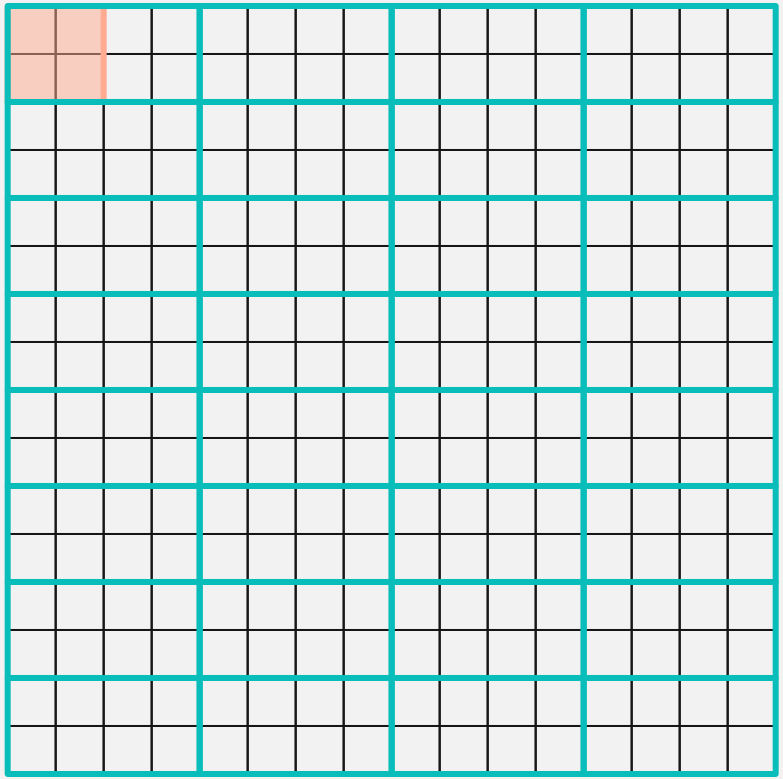
# Tensorization

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)
```



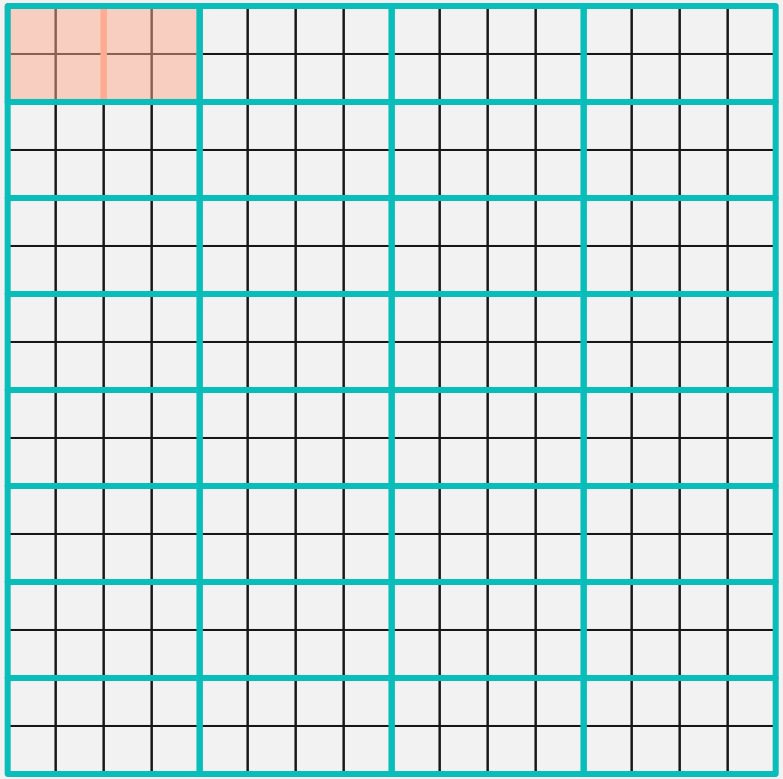
# Tensorization

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)
```



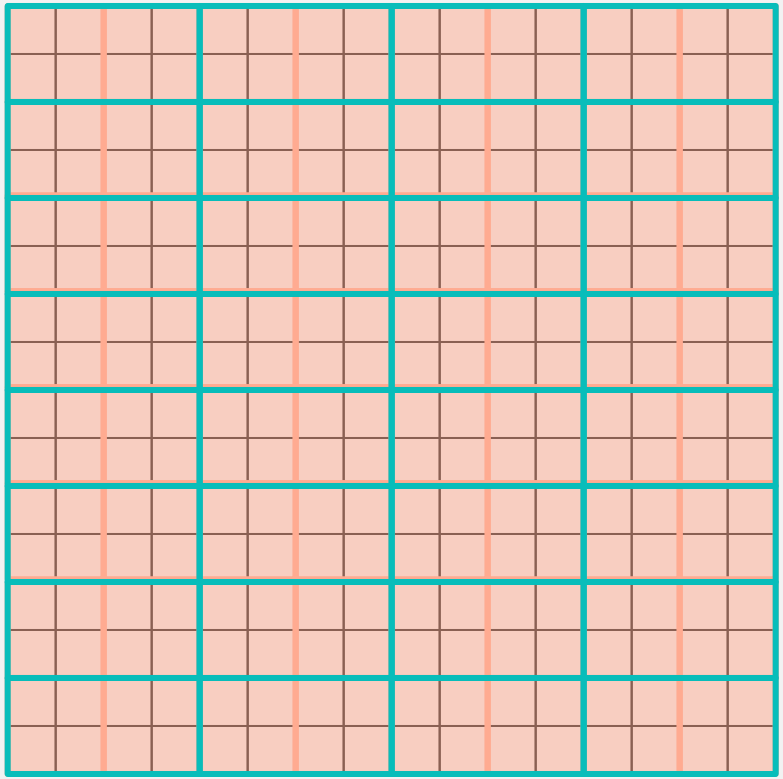
# Tensorization

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)
```



# Tensorization

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)
```



# Tensorization in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2)
```

# Tensorization in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2)
```

# Tensorization in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2)
```

# Tensorization in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2)
```

# Tensorization in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)
  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2)
```

# Tensorization in Decoupled Triton

# Tensorization in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.compile_to_kernel();
```

# Tensorization in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.compile_to_kernel();
```

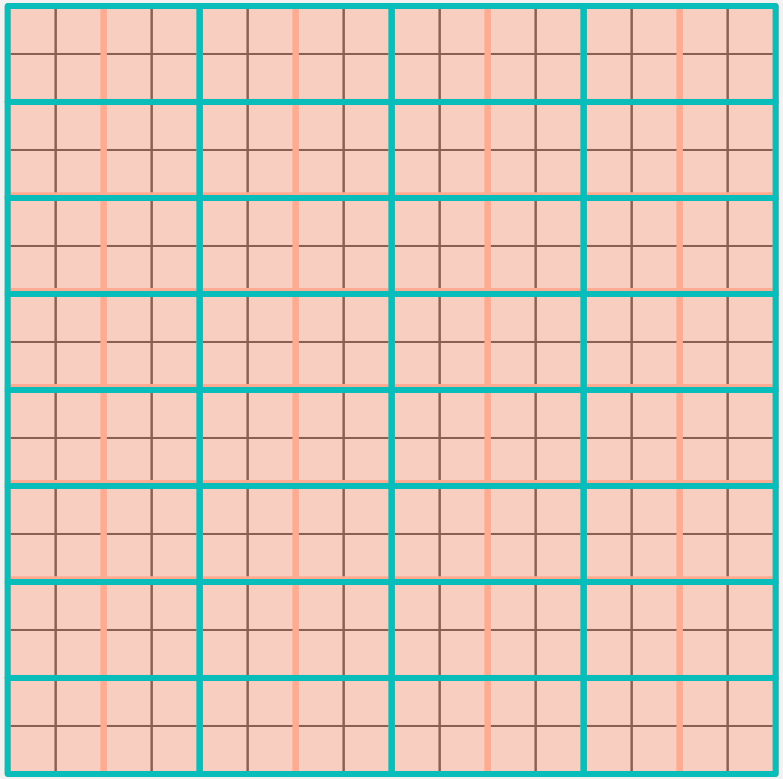
# Tensorization in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.compile_to_kernel();
```

# Grouping

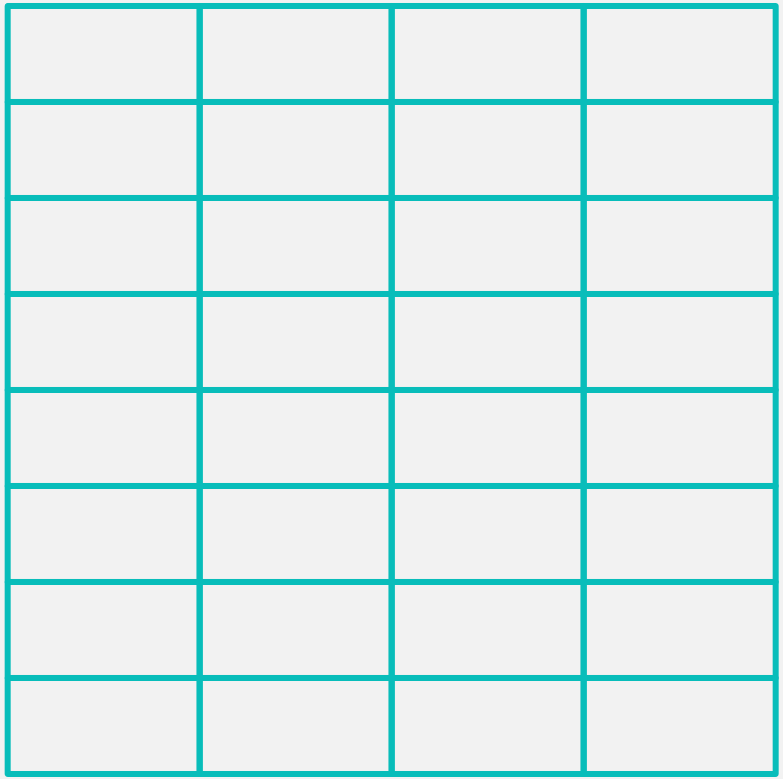
# Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```



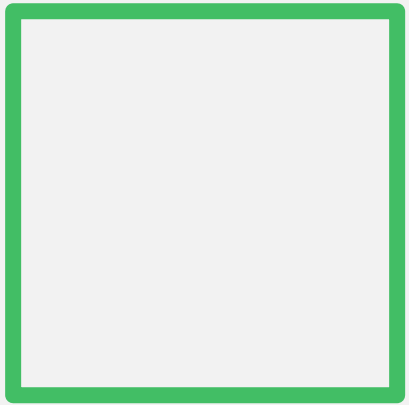
# Grouping

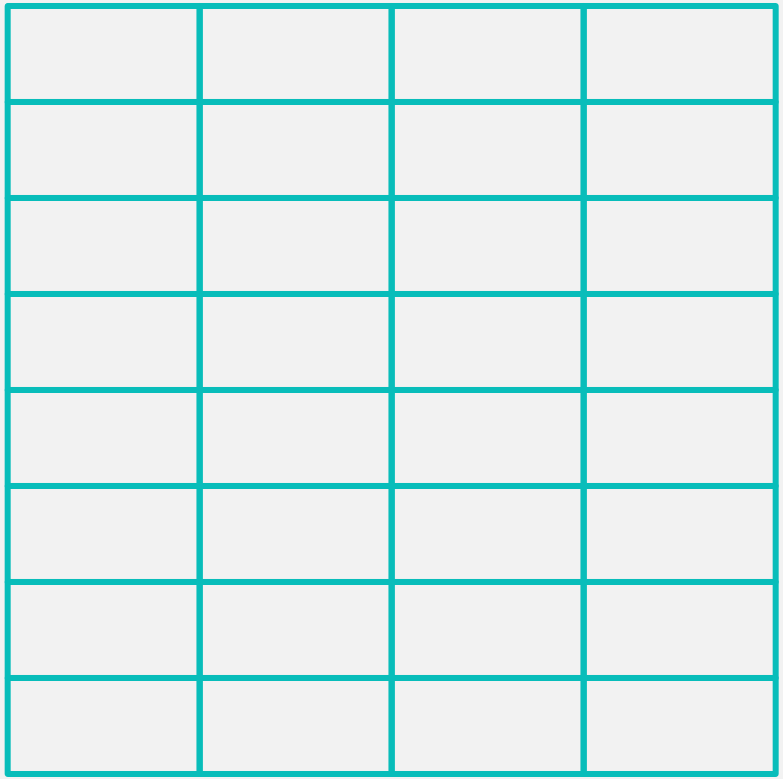
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```



# Grouping

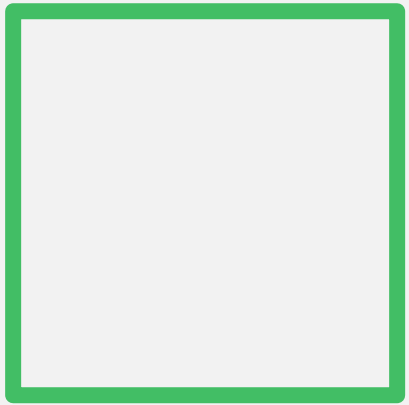
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```





# Grouping

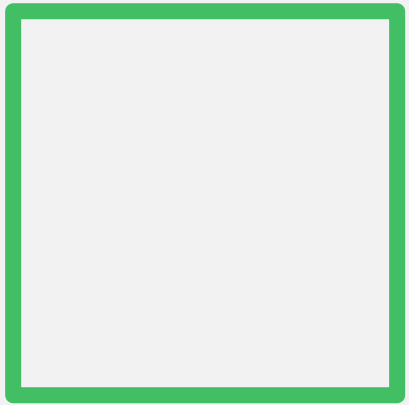
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```

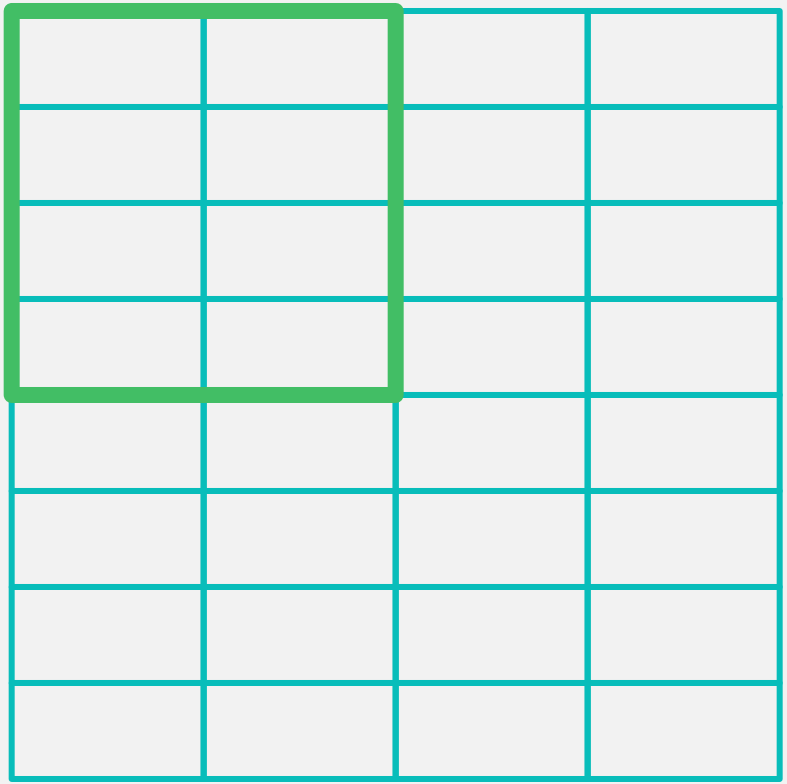


| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Grouping

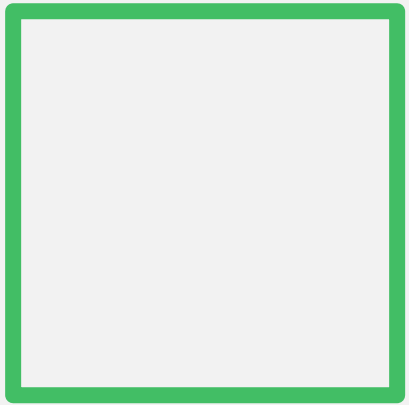
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```

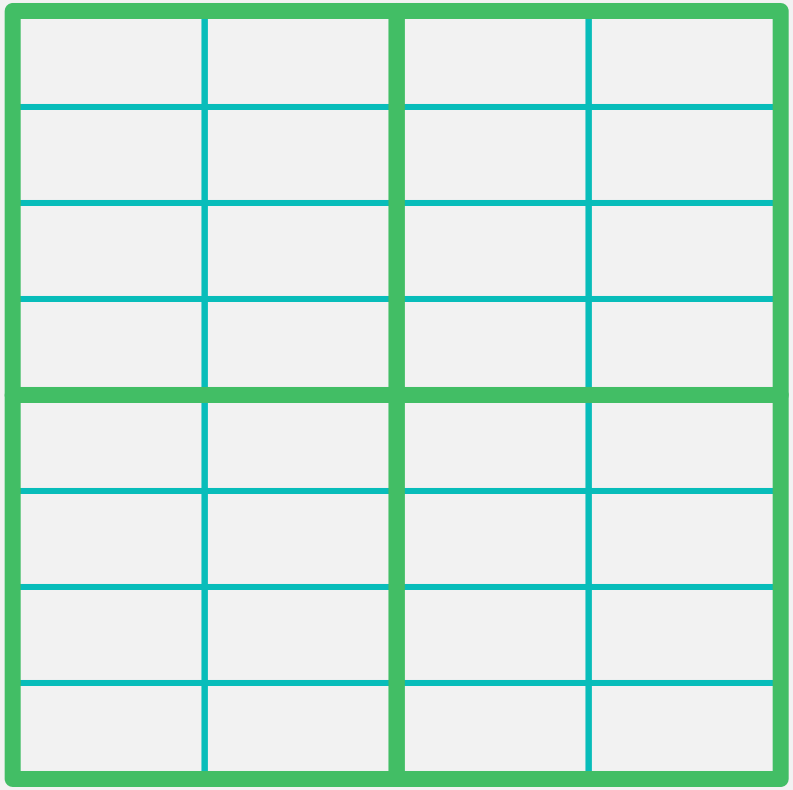




# Grouping

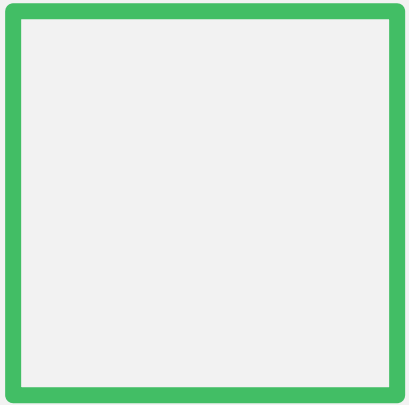
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```

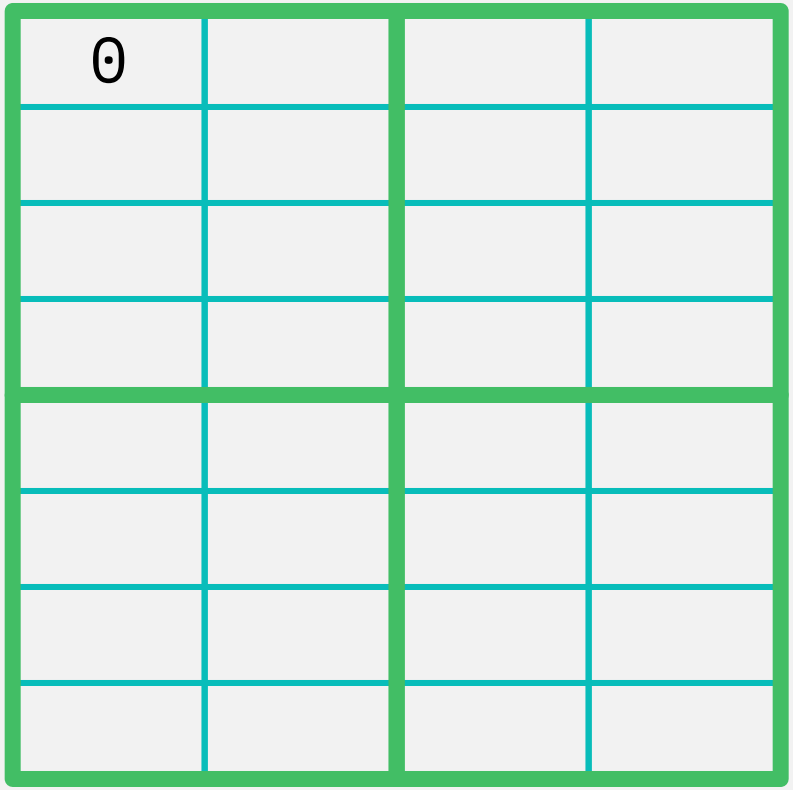




# Grouping

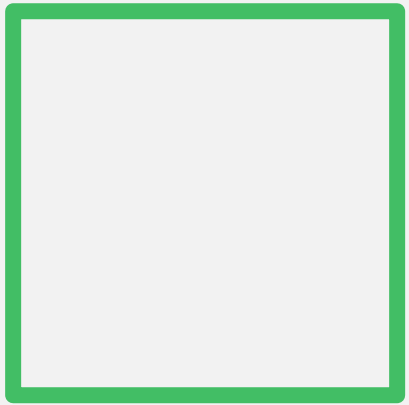
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
```

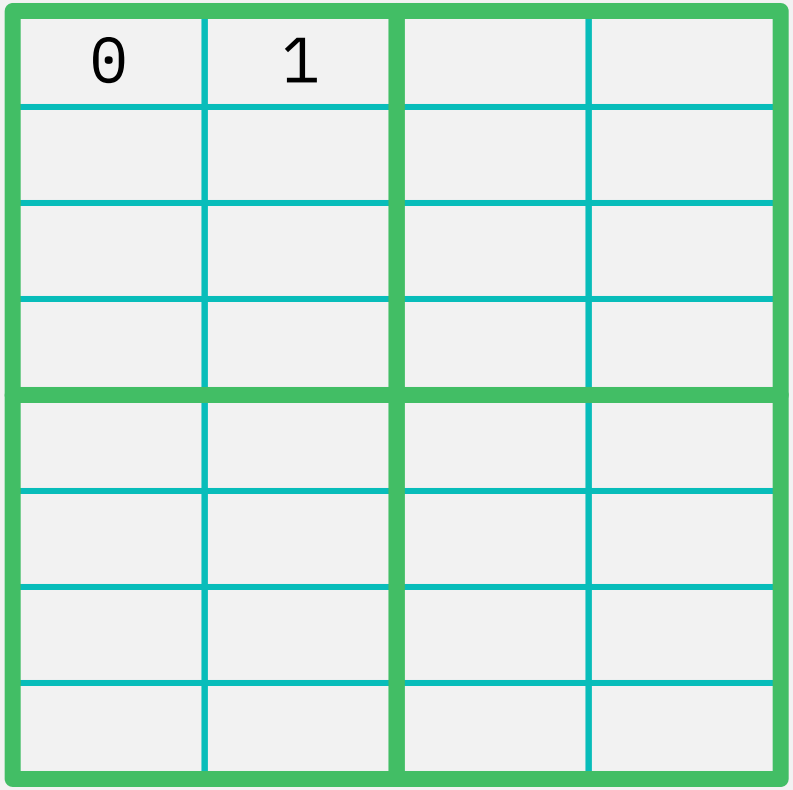




# Grouping

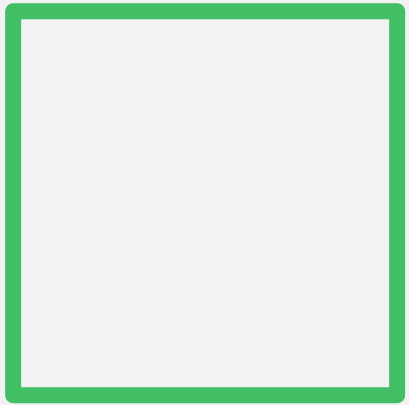
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```

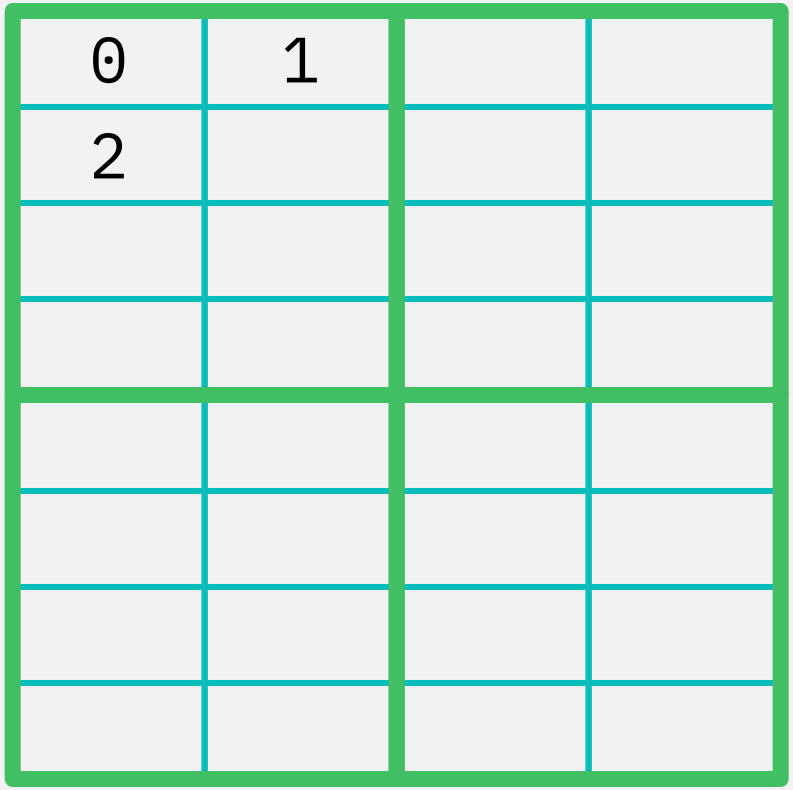




# Grouping

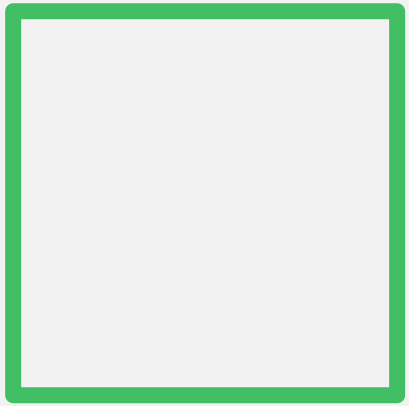
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```

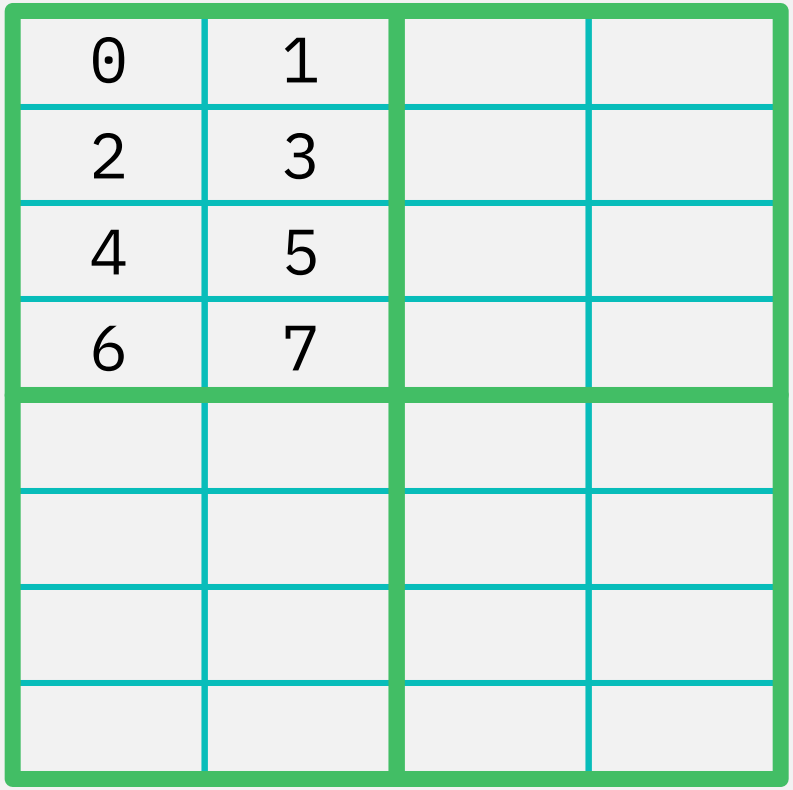




# Grouping

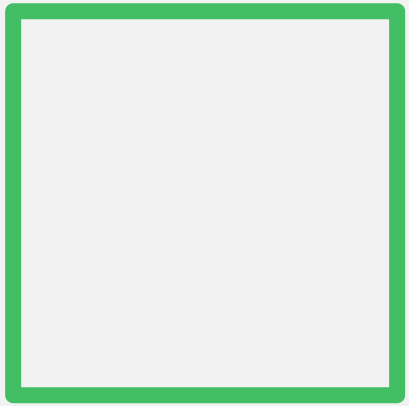
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```





# Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)
```



| | | | |
|---|---|---|---|
| 0 | 1 | 8 | 9 |
| 2 | 3 | 10 | 11 |
| 4 | 5 | 12 | 13 |
| 6 | 7 | 14 | 15 |
| 16 | 17 | 24 | 25 |
| 18 | 19 | 26 | 27 |
| 20 | 21 | 28 | 29 |
| 22 | 23 | 30 | 31 |

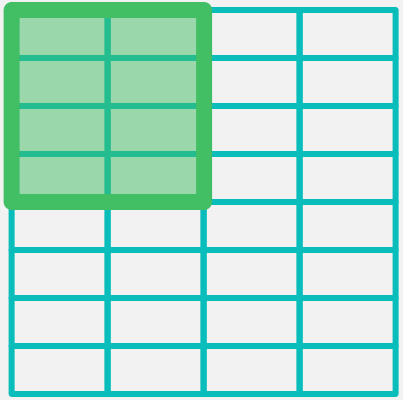
# Why group?

# Why group?

An operation for which a **group** of the result has better data locality than the default row-major ordering?

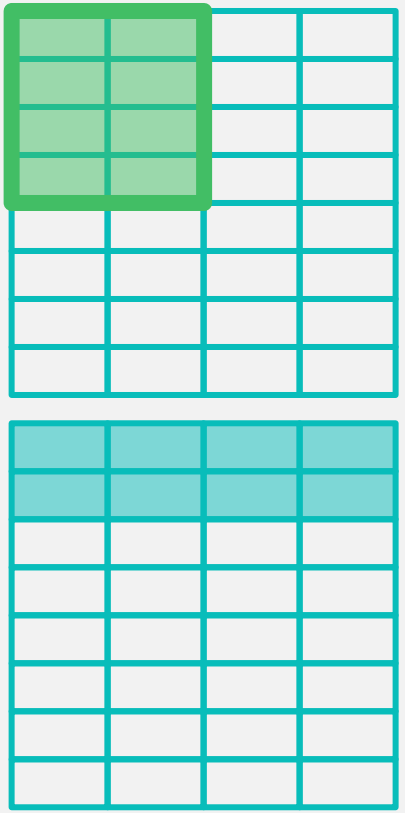
# Why group?

An operation for which a **group** of the result has better data locality than the default row-major ordering?

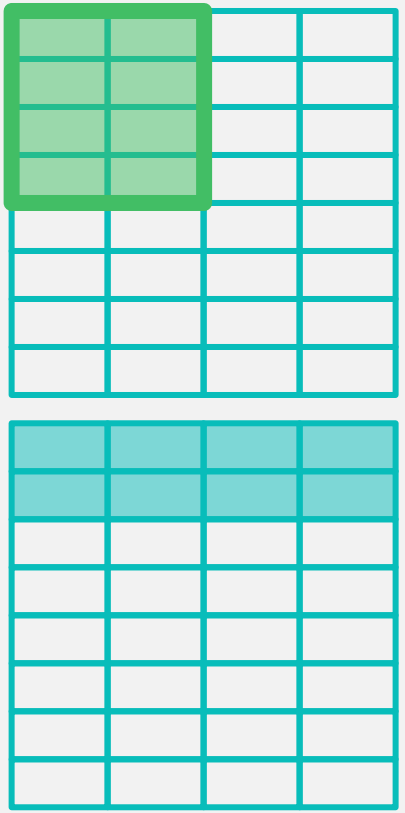


# Why group?

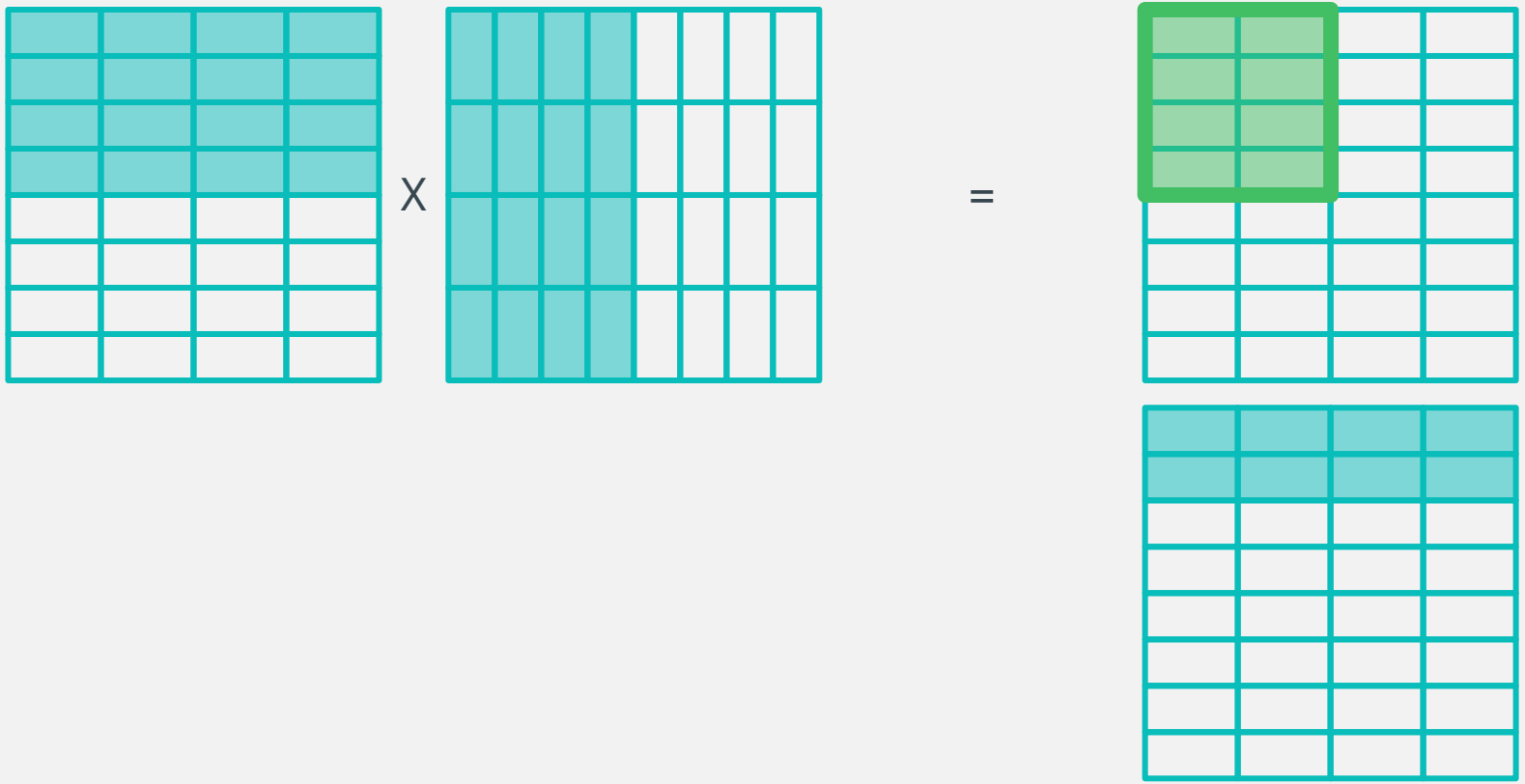
An operation for which a **group** of the result has better data locality than the default row-major ordering?



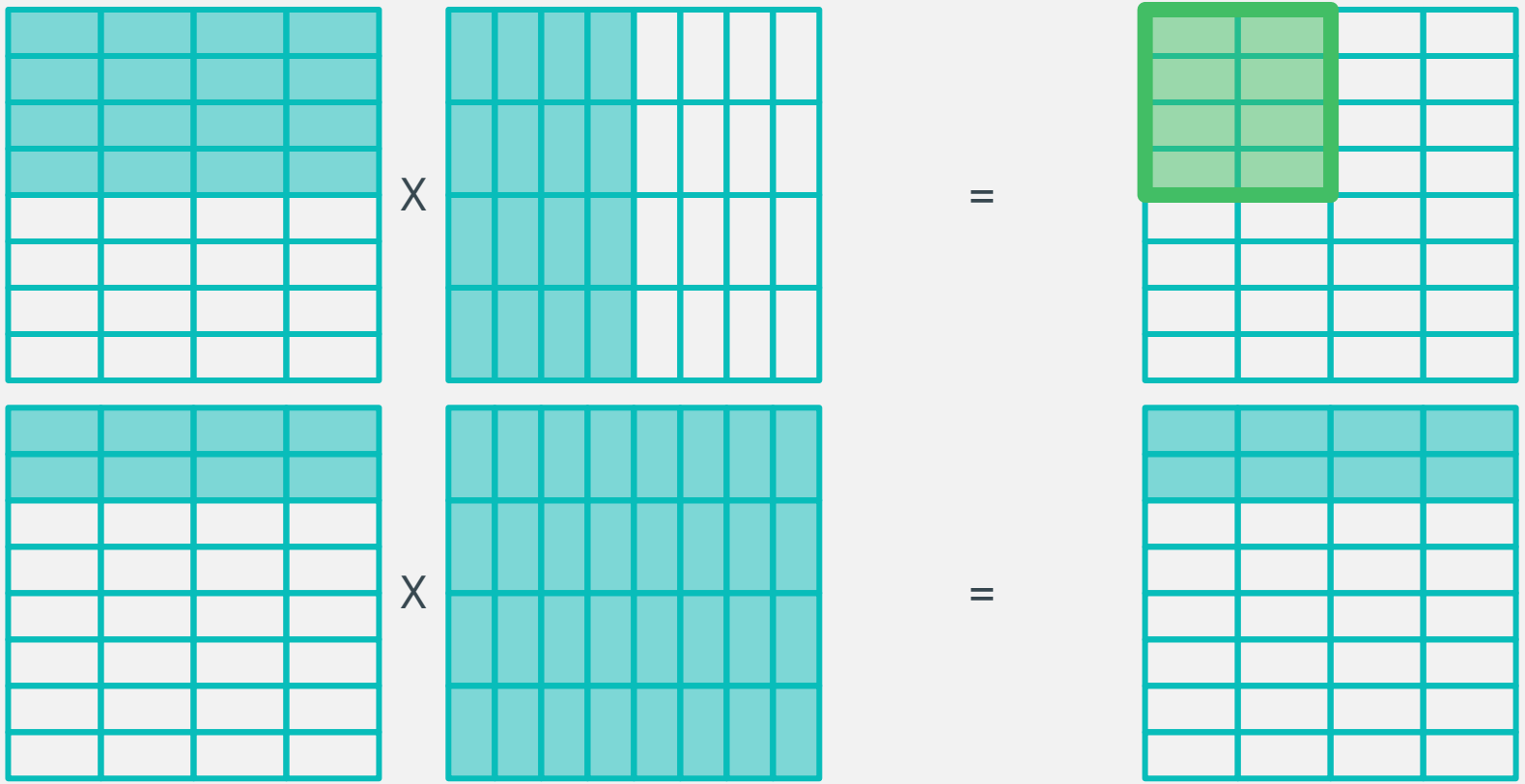
# Matrix Multiplication



# Matrix Multiplication



# Matrix Multiplication



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

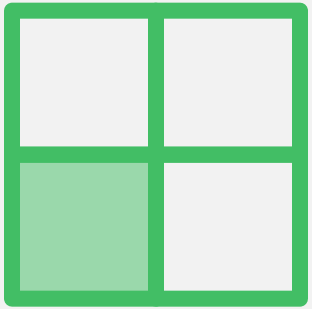
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

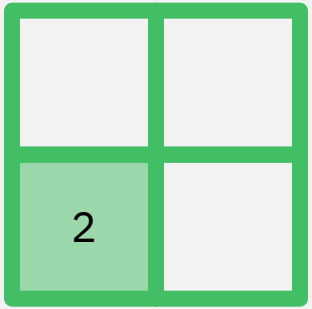
```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

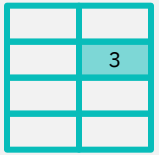
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

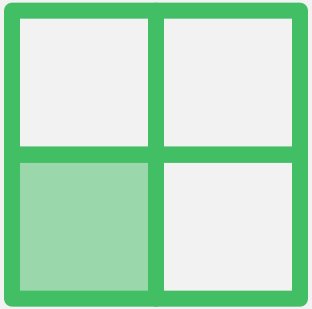
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

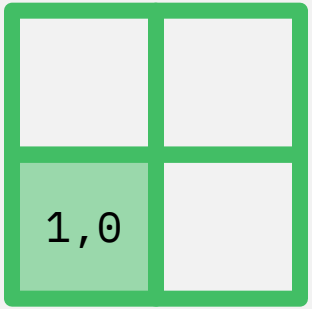
```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

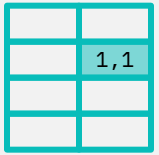
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

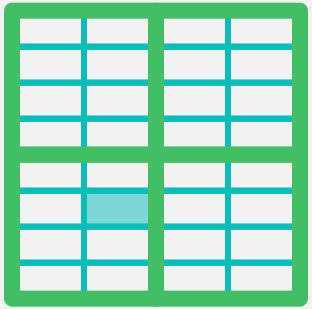
```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

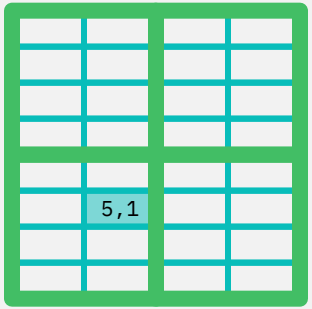
```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```



# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Group Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Group Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_blocks_in_group = x_GROUP_SIZE * y_GROUP_SIZE
group_id = pid // num_blocks_in_group
group_offset = pid % num_blocks_in_group
y_num_group_ids = y // y_BLOCK_SIZE // y_GROUP_SIZE
x_group_id = group_id // y_num_group_ids
y_group_id = group_id % y_num_group_ids
x_group_offset = group_offset // y_GROUP_SIZE
y_group_offset = group_offset % y_GROUP_SIZE
x_pid = x_group_id * x_GROUP_SIZE + x_group_offset
y_pid = y_group_id * y_GROUP_SIZE + y_group_offset
```

# Grouping in Decoupled Triton

# Grouping in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.compile_to_kernel();
```

# Grouping in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.compile_to_kernel();
```

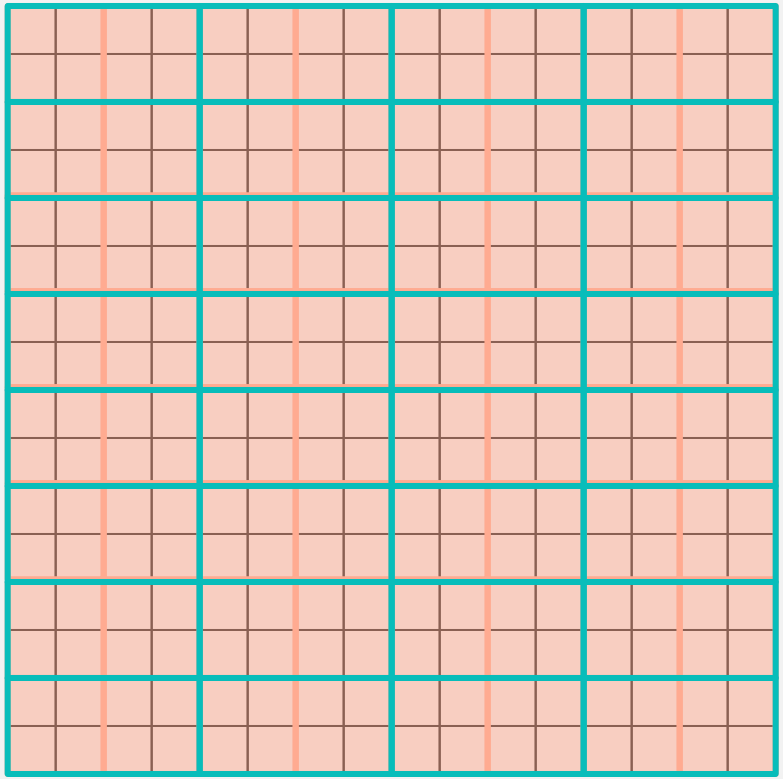
# Grouping in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.compile_to_kernel();
```

# Dilation

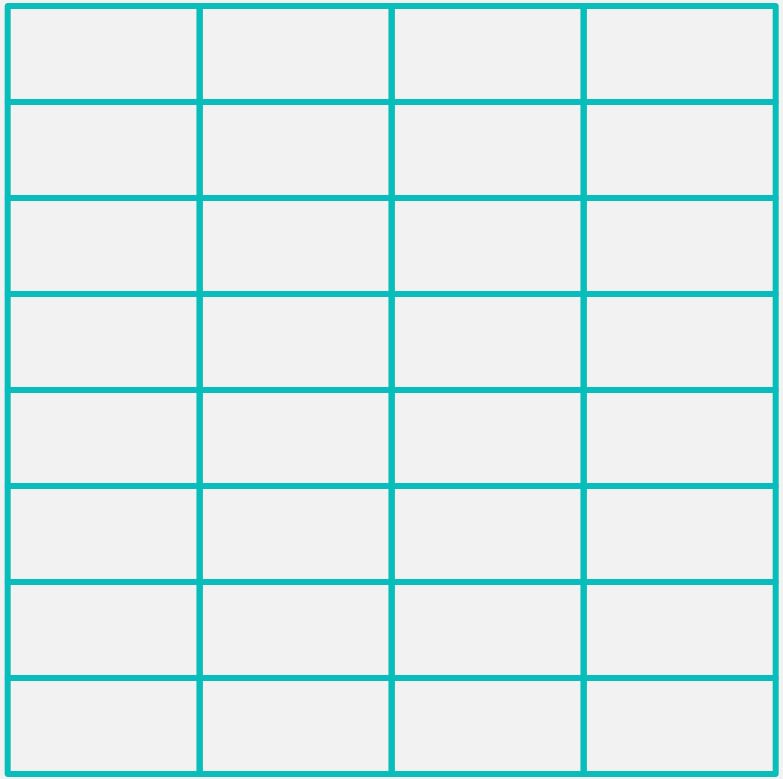
# Dilation

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```



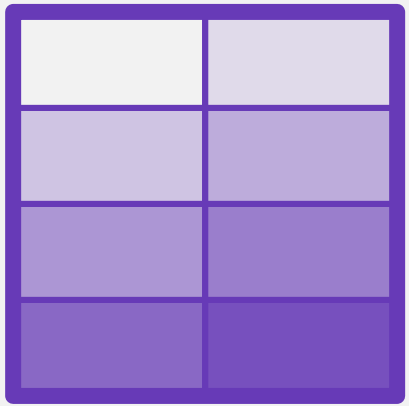
# Dilation

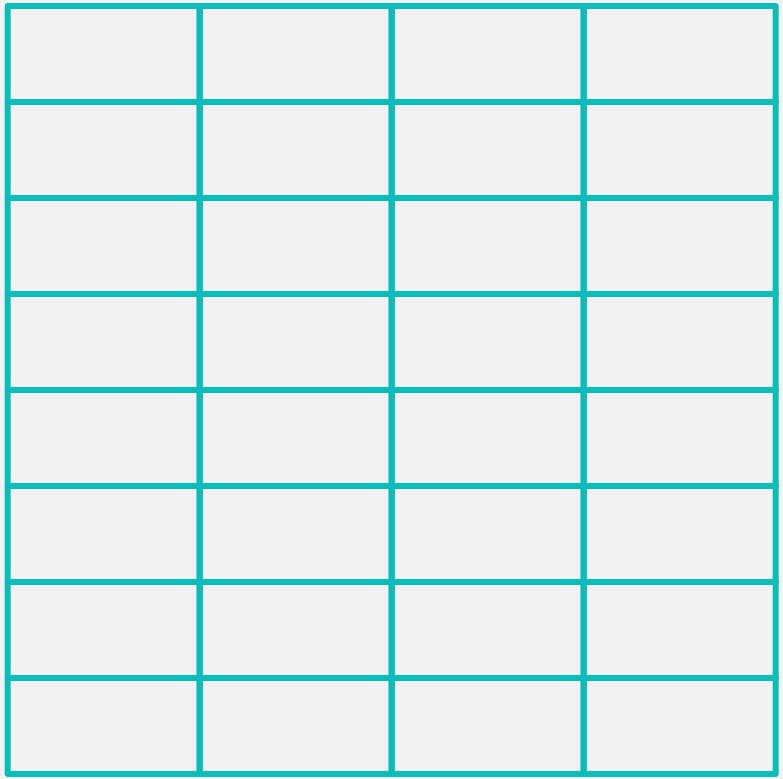
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```



# Dilation

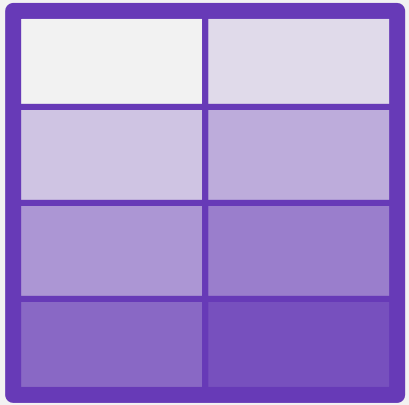
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```





# Dilation

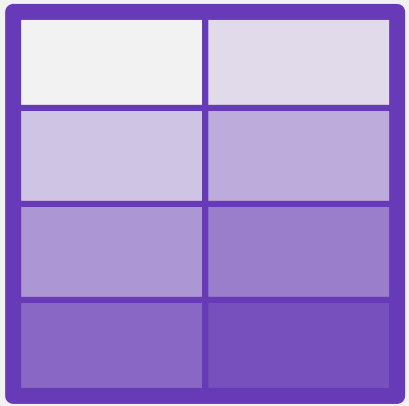
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

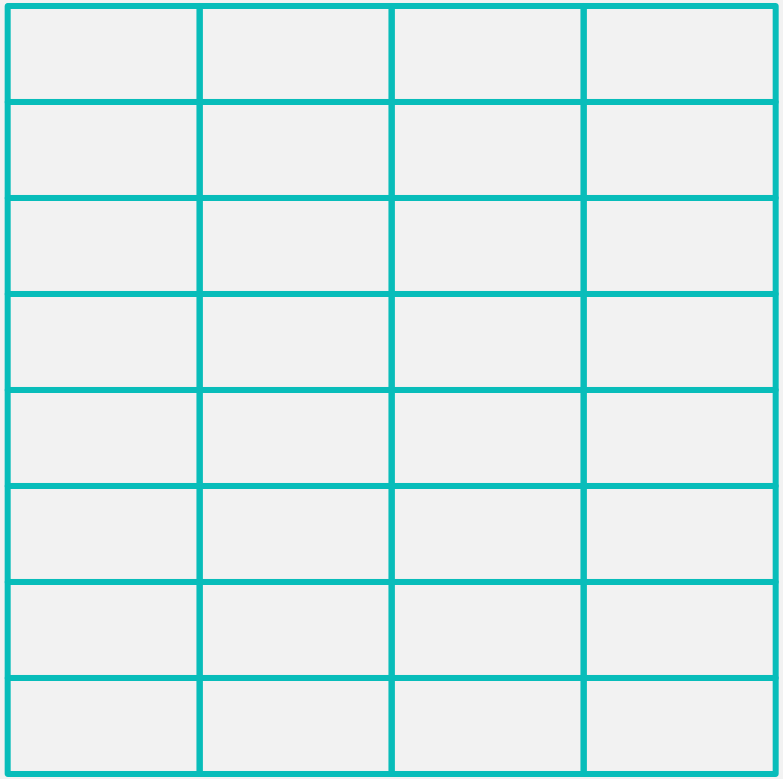


| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Dilation

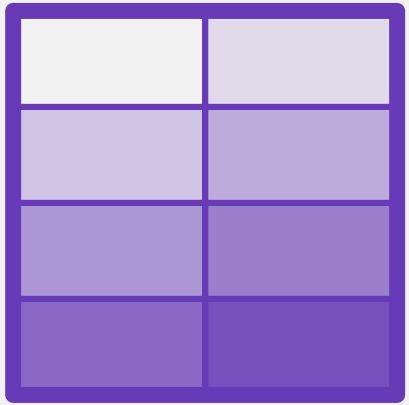
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

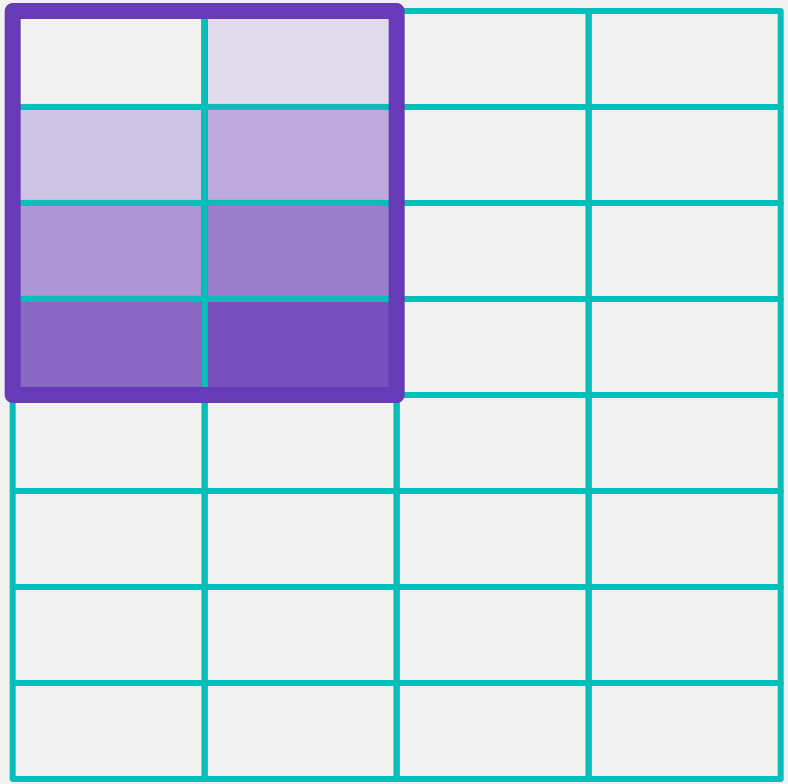




# Dilation

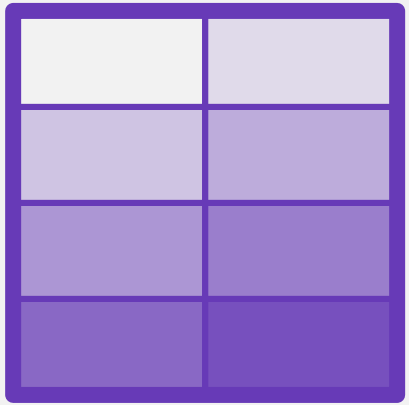
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

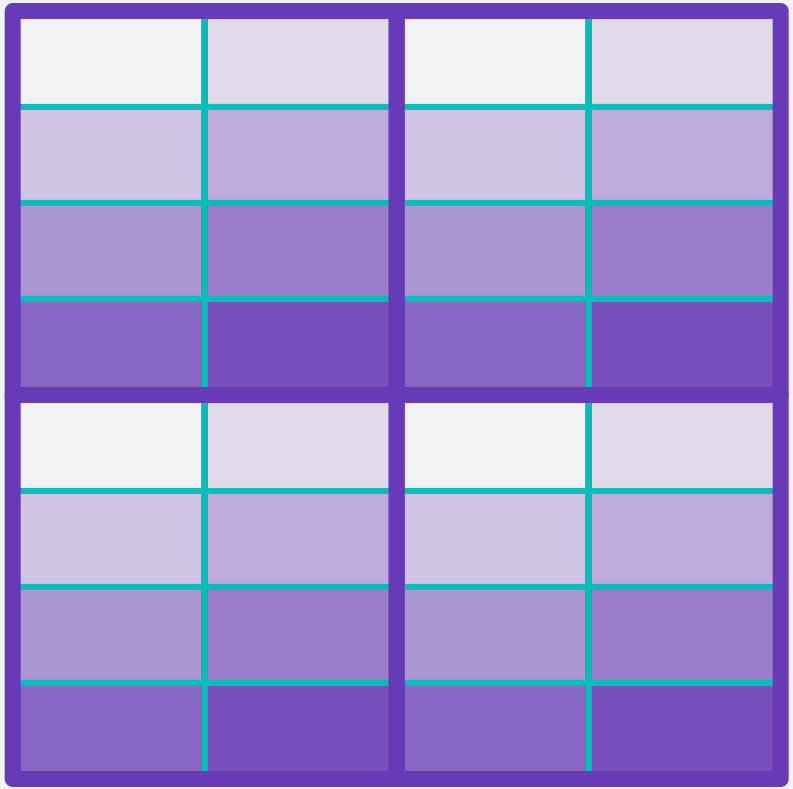




# Dilation

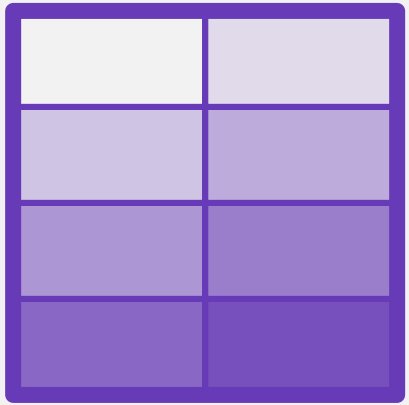
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

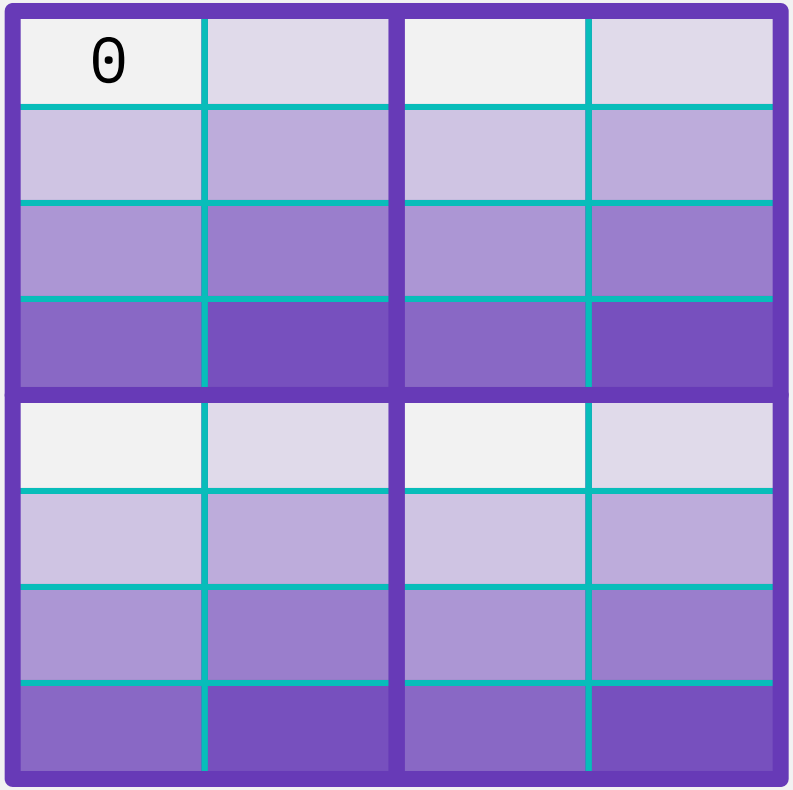




# Dilation

```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
```



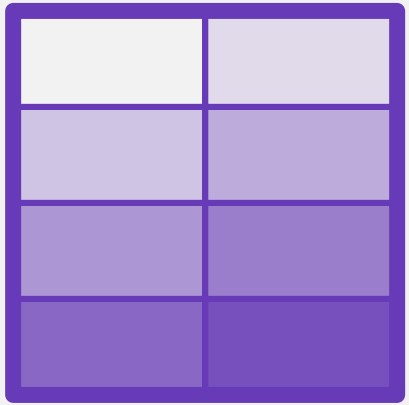


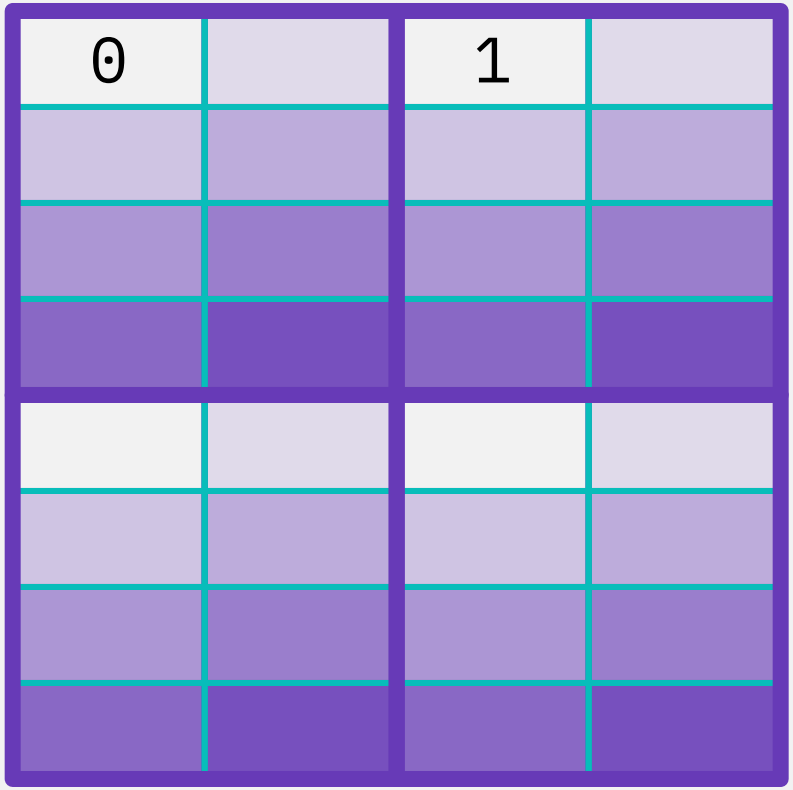
# Dilation

```
f.block(x:2, y:4)
```

```
f.tensorize(x:2, y:2)
```

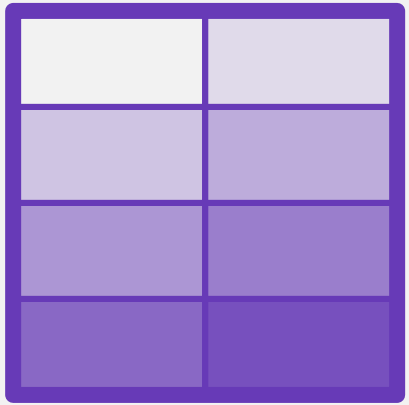
```
f.dilate(x:4, y:2)
```

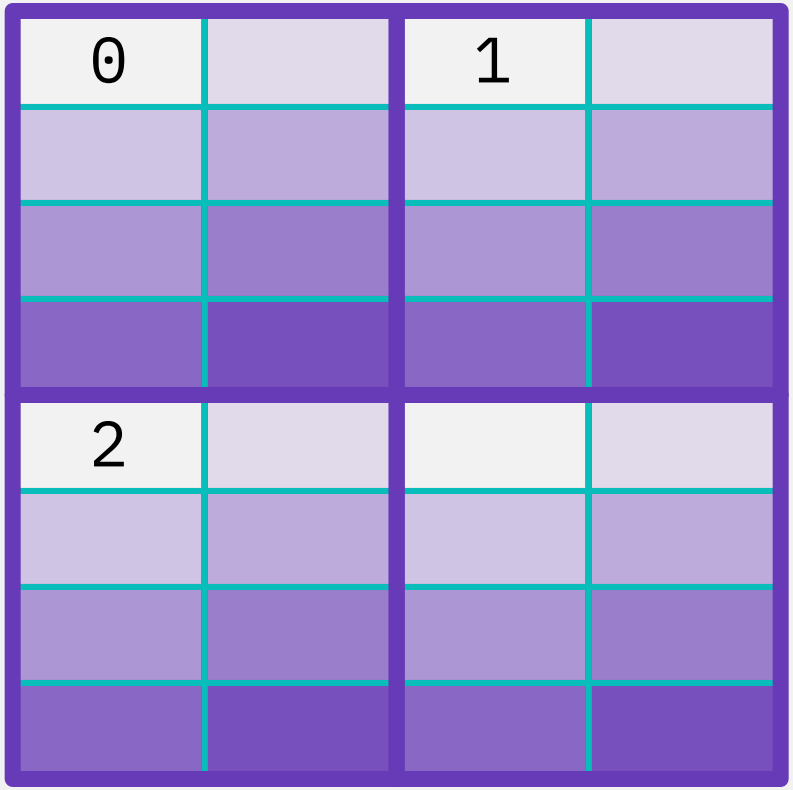




# Dilation

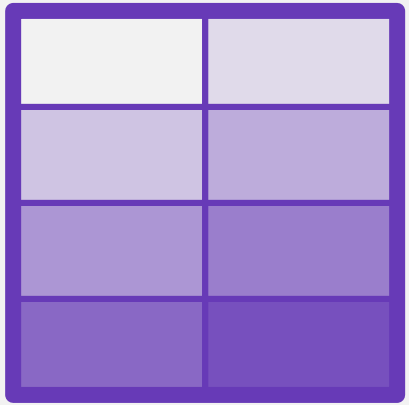
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

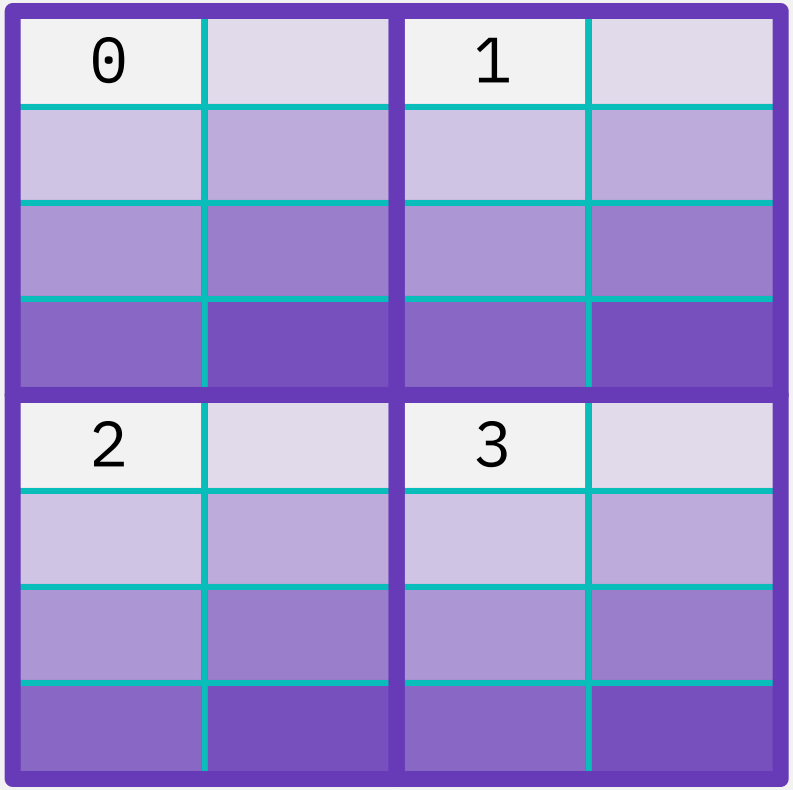




# Dilation

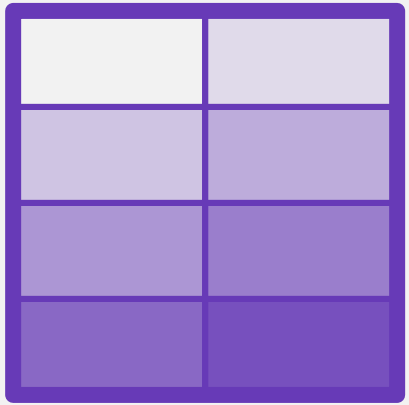
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

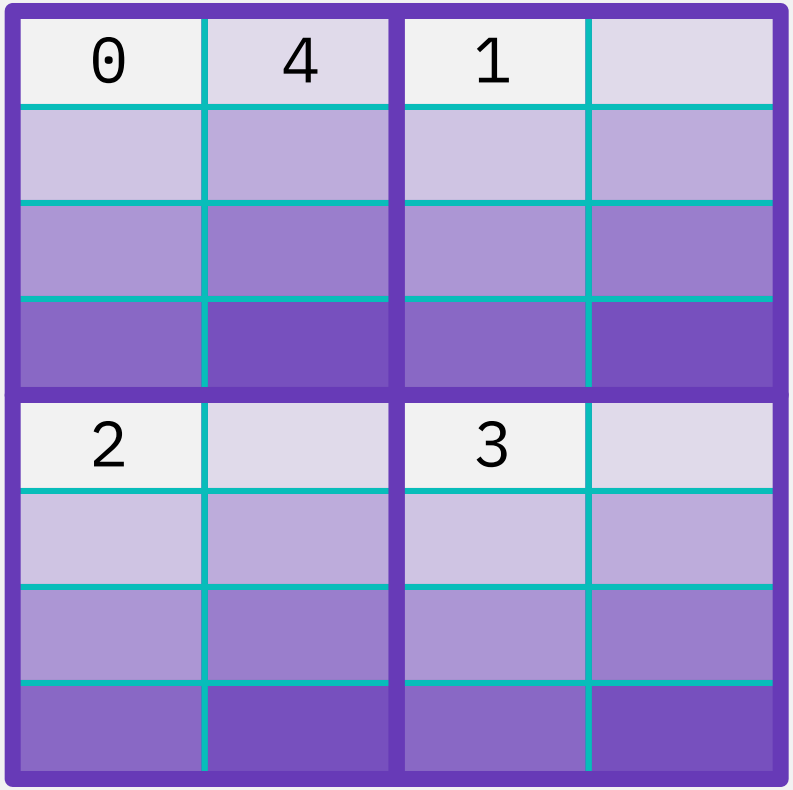




# Dilation

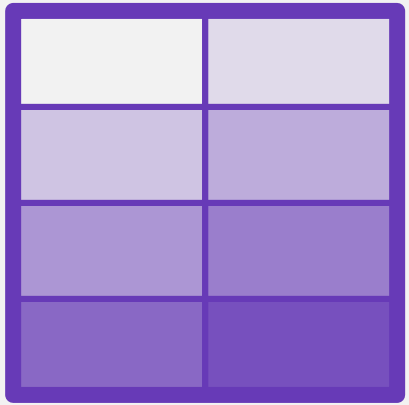
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

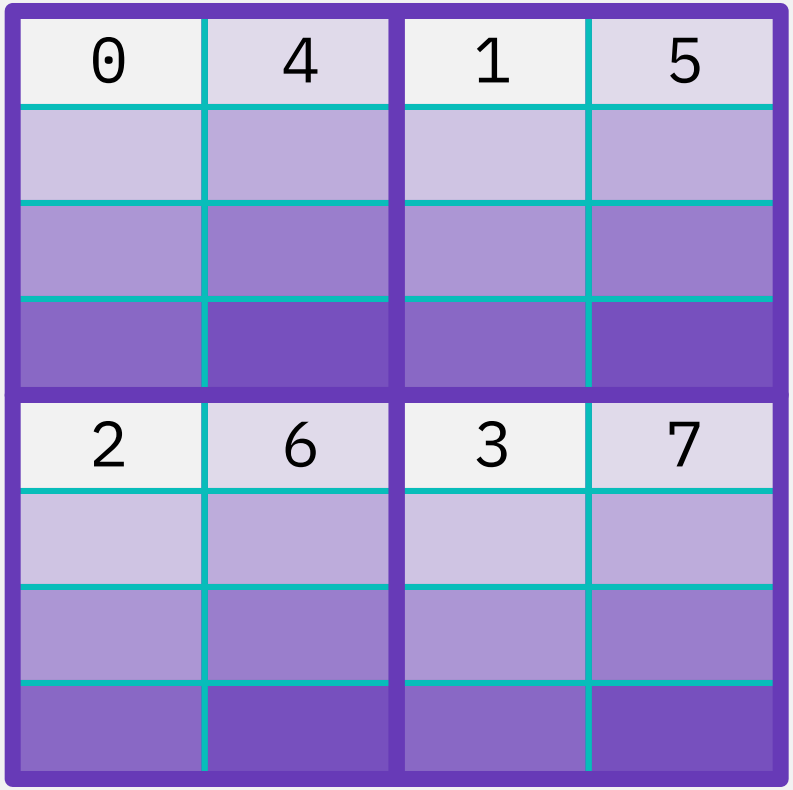




# Dilation

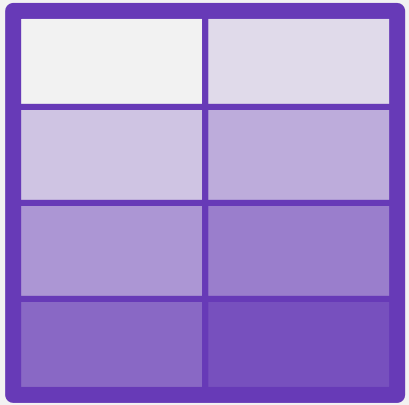
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```

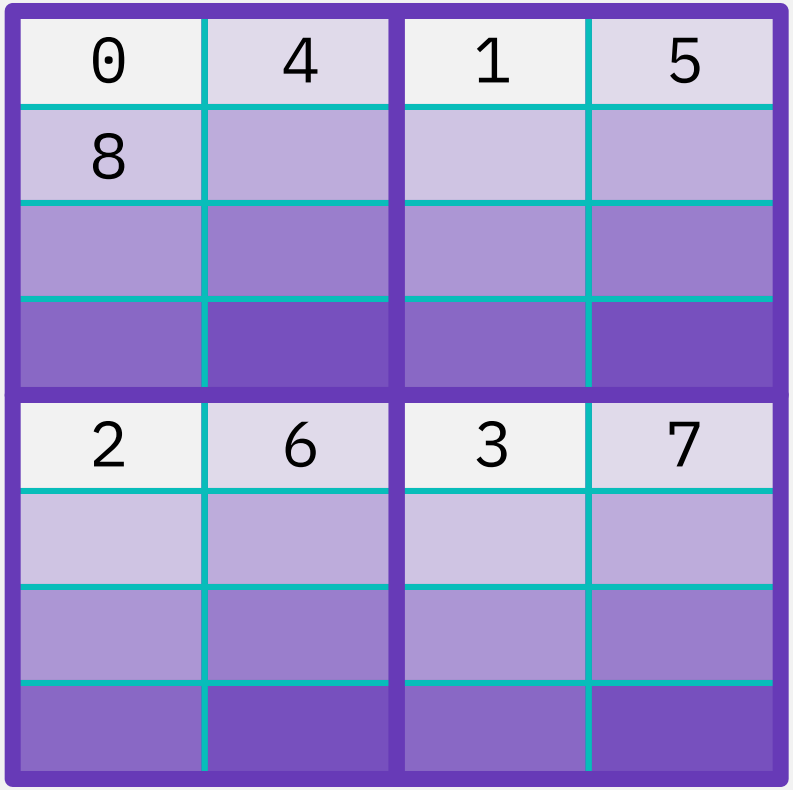




# Dilation

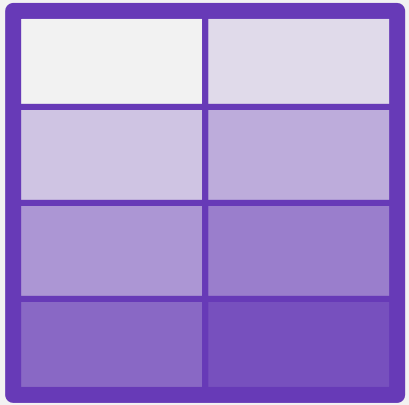
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```





# Dilation

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.dilate(x:4, y:2)
```



| | | | |
|---|---|---|---|
| 0 | 4 | 1 | 5 |
| 8 | 12 | 9 | 13 |
| 16 | 20 | 17 | 21 |
| 24 | 28 | 25 | 29 |
| 2 | 6 | 3 | 7 |
| 10 | 14 | 11 | 15 |
| 18 | 22 | 19 | 23 |
| 26 | 30 | 27 | 31 |

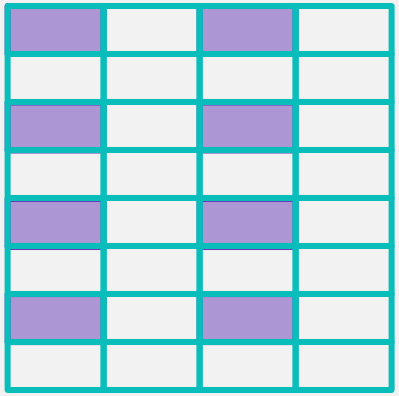
# Why dilate?

# Why dilate?

An operation for which a **dilation** of the result has better data locality than the default row-major ordering?

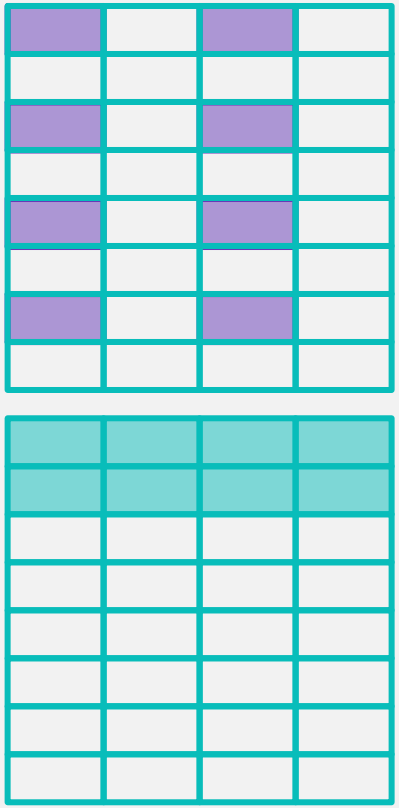
# Why dilate?

An operation for which a **dilation** of the result has better data locality than the default row-major ordering?



# Why dilate?

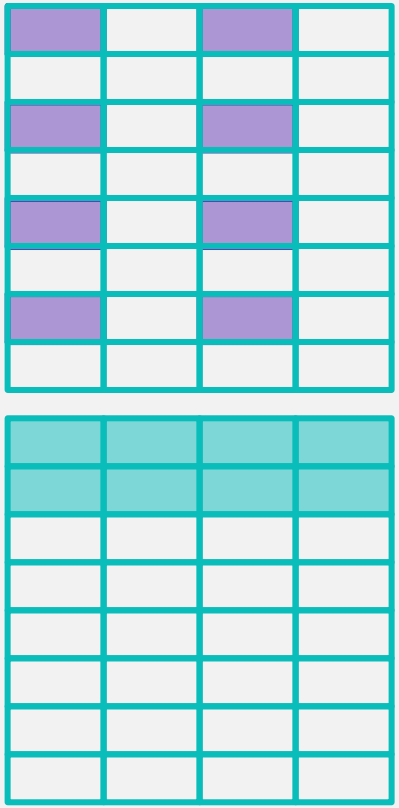
An operation for which a **dilation** of the result has better data locality than the default row-major ordering?



# Why dilate?

An operation for which a **dilation** of the result has better data locality than the default row-major ordering?

**Does one exist?**



# Dilation in Triton

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_SIZE):
    a = tl.load(a_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    b = tl.load(b_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
                y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :])
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

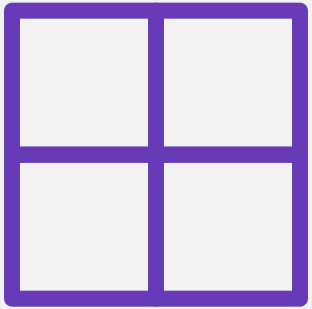
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

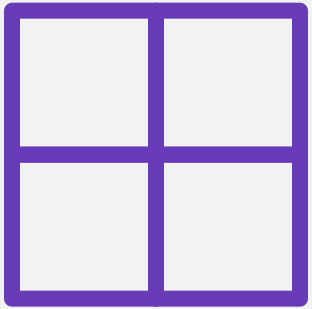
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

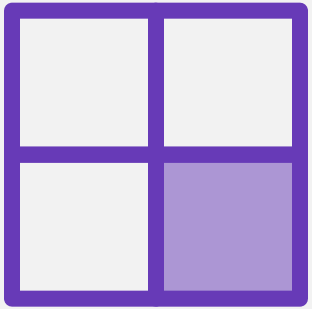
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

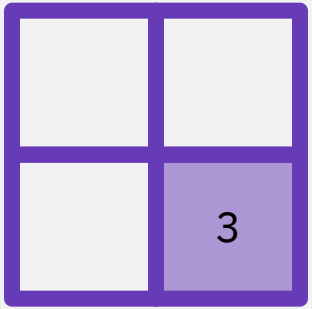
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

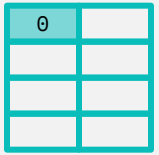
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

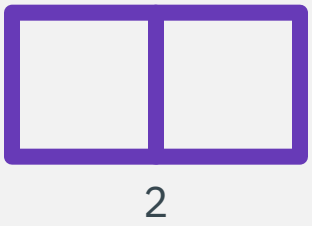
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

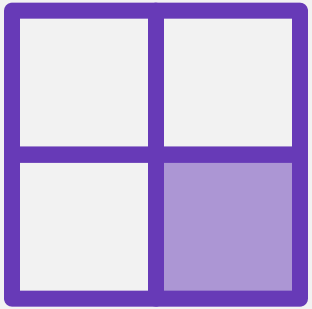
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

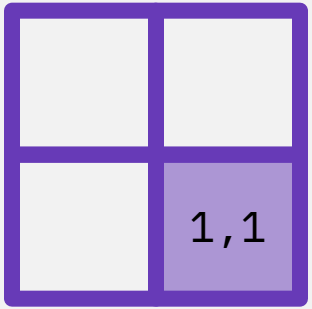
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

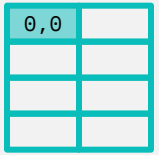
```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

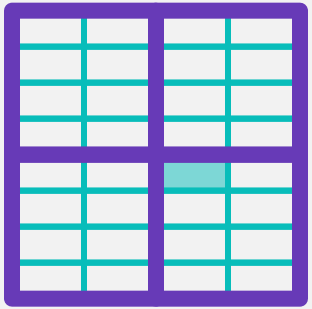
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

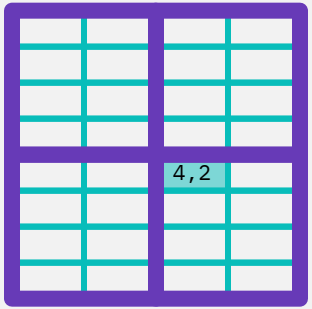
```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```



# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping

pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2)
```

```
...  # Dilation Remapping


pid = x_pid * (y // y_BLOCK_SIZE) + y_pid
num_dilate_ids = (x // x_BLOCK_SIZE) // x_DILATE_SIZE *
                 (y // y_BLOCK_SIZE) // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = (y // y_BLOCK_SIZE) // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_dilate_id * x_DILATE_SIZE + x_dilate_offset
y_pid = y_dilate_id * y_DILATE_SIZE + y_dilate_offset
```

# Dilation in Decoupled Triton

# Dilation in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.dilate(x:4, y:2);
f.compile_to_kernel();
```

# Dilation in Decoupled Triton

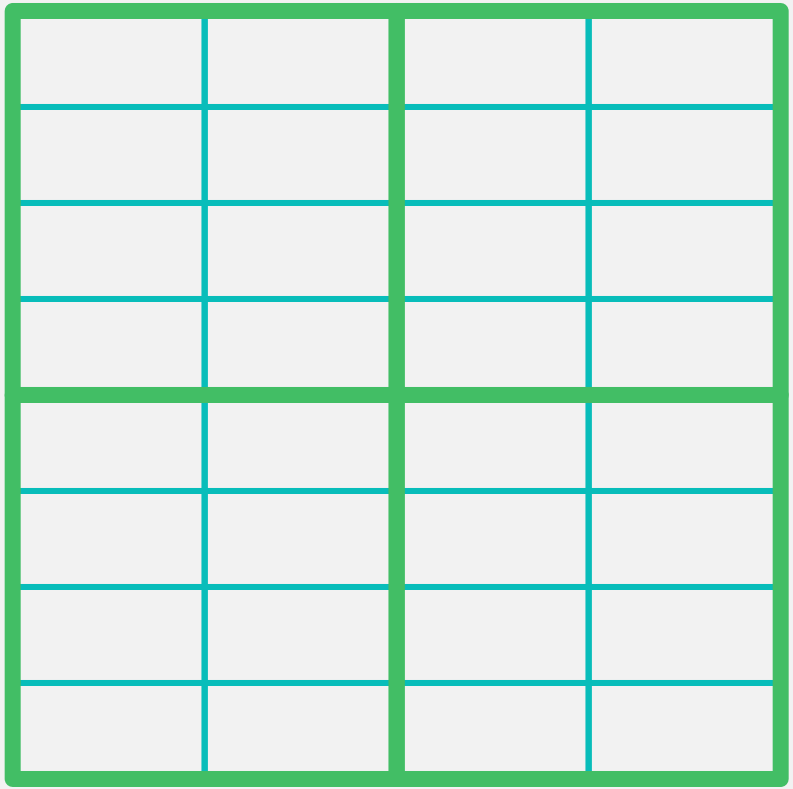
```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.dilate(x:4, y:2);
f.compile_to_kernel();
```

# Dilation in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.dilate(x:4, y:2);
f.compile_to_kernel();
```

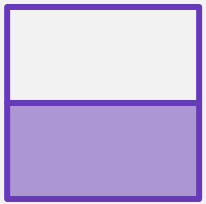
# Dilation with Grouping

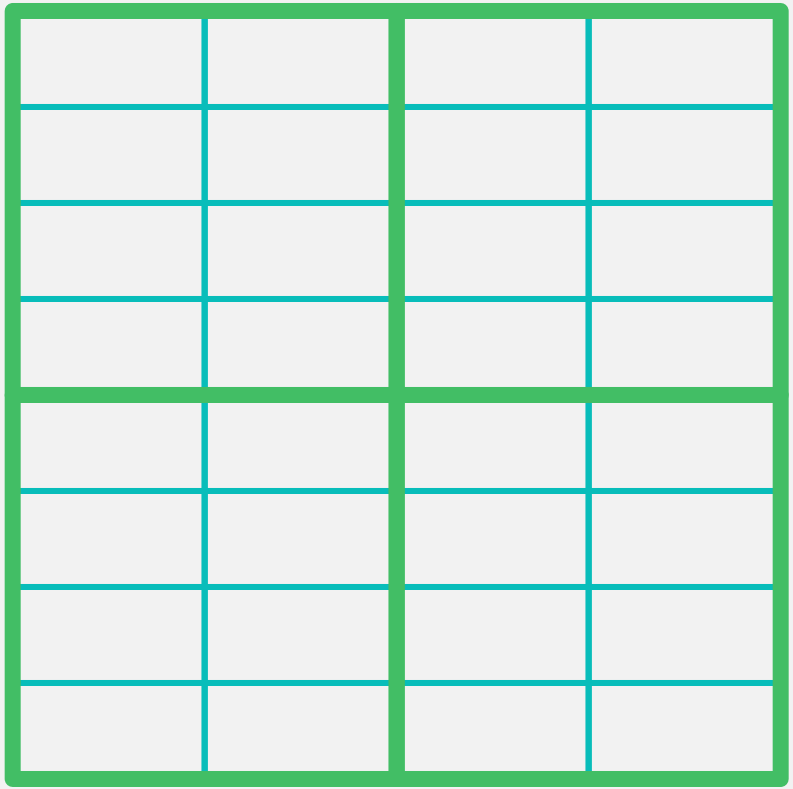
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.dilate(x:2, y:1)
```



# Dilation with Grouping

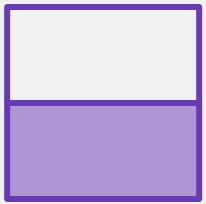
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

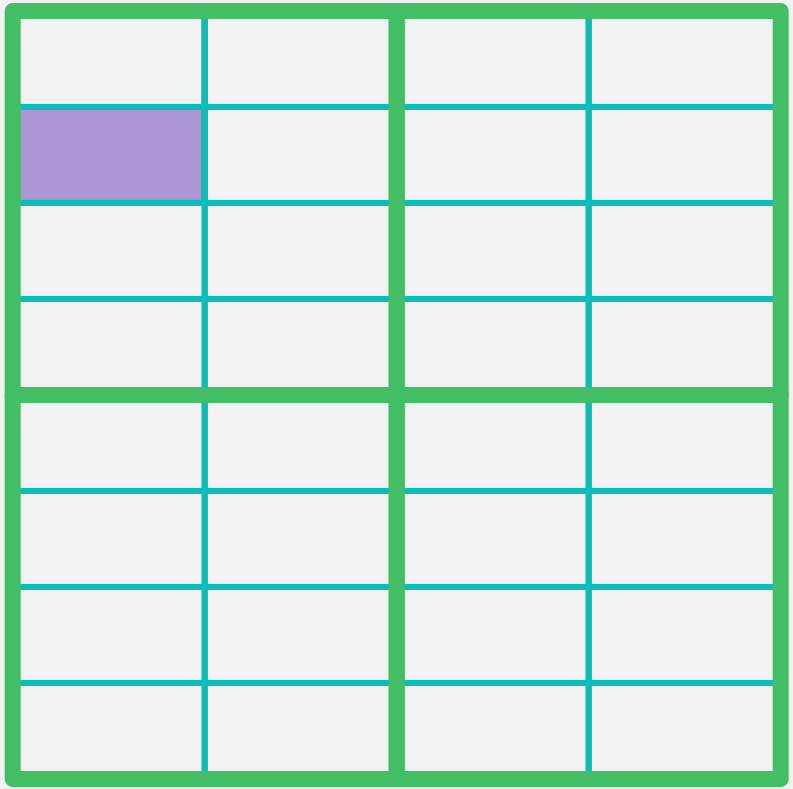




# Dilation with Grouping

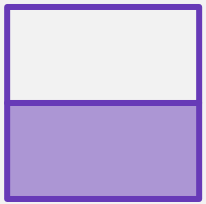
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

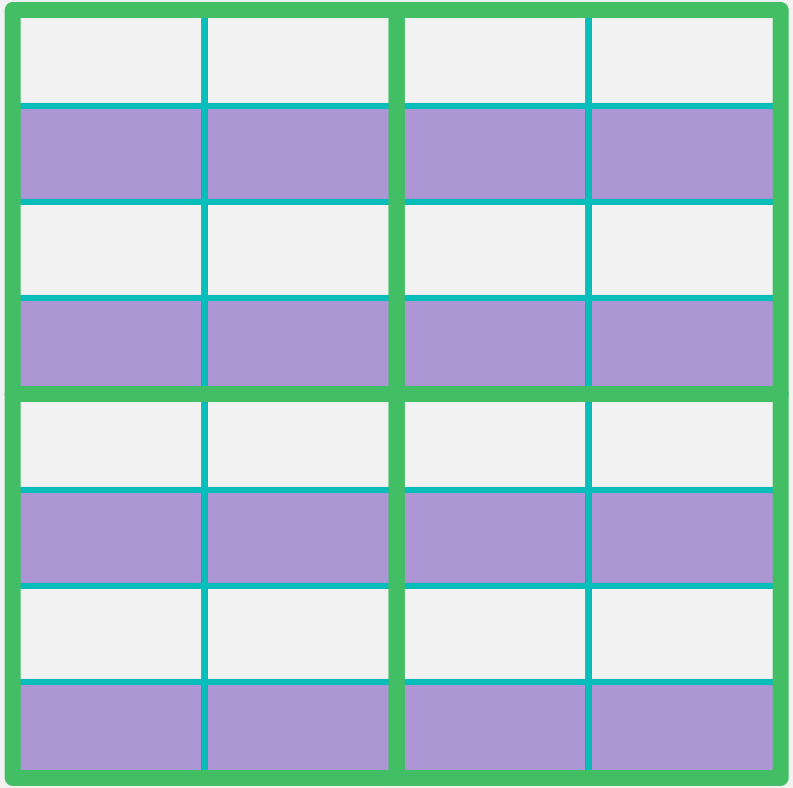




# Dilation with Grouping

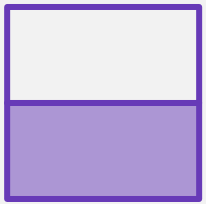
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

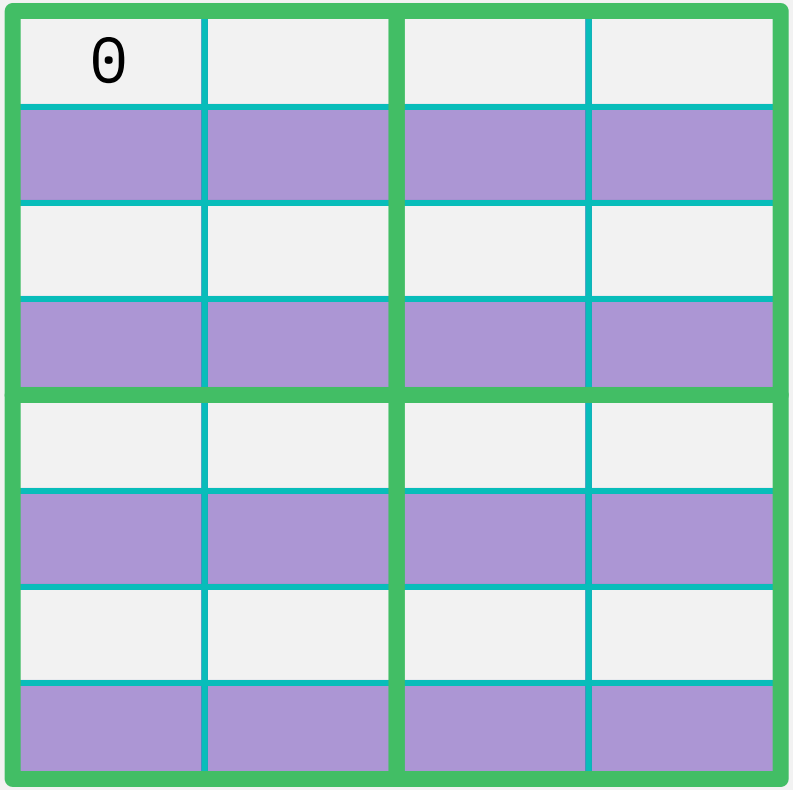




# Dilation with Grouping

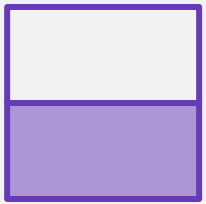
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

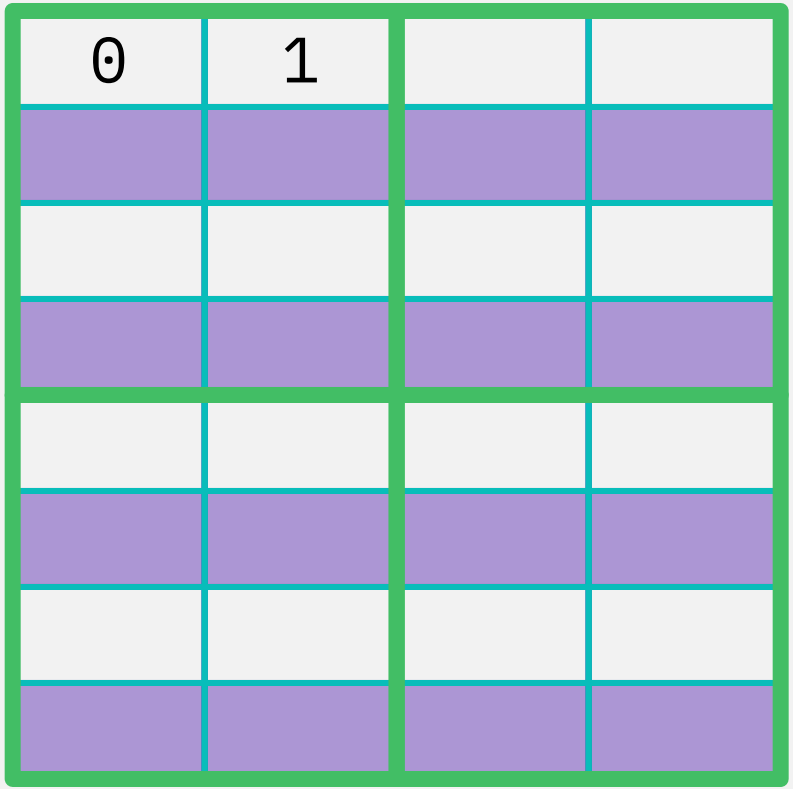




# Dilation with Grouping

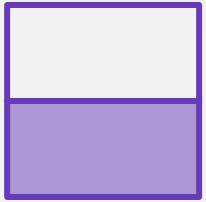
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

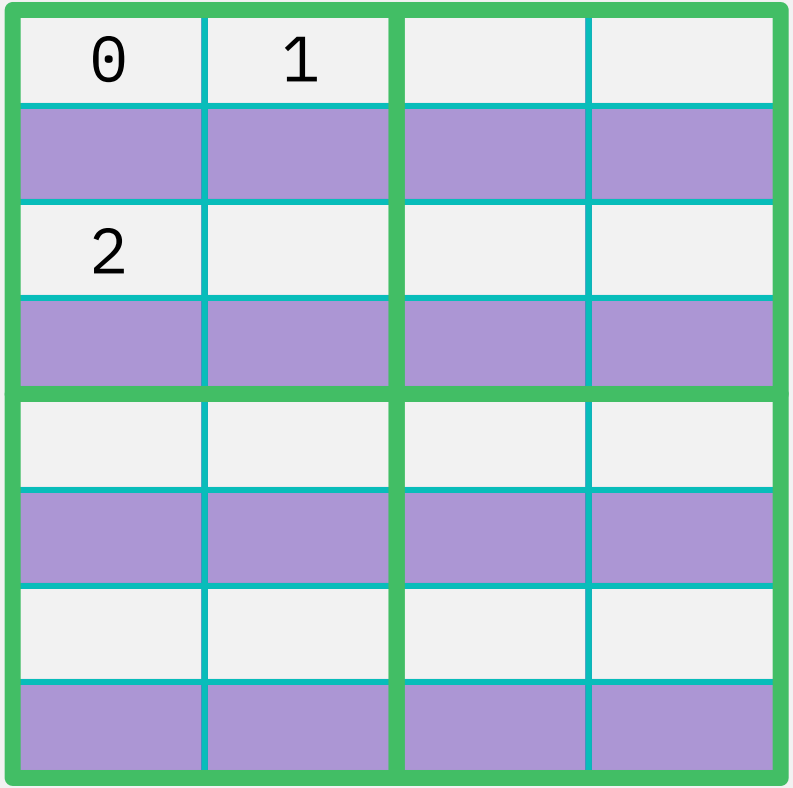




# Dilation with Grouping

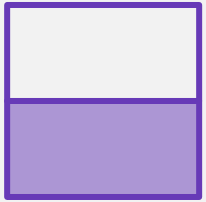
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

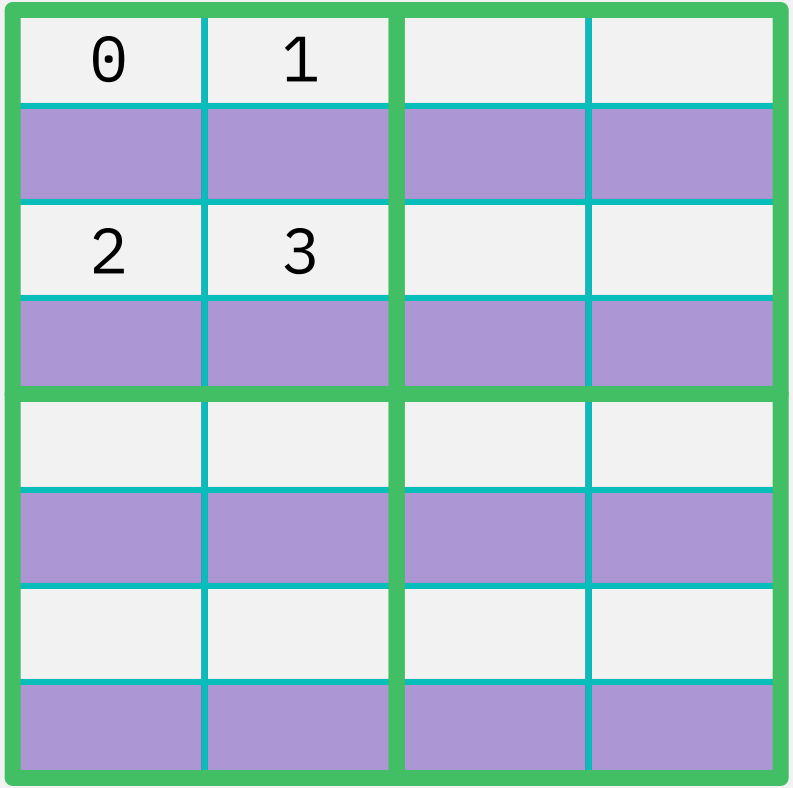




# Dilation with Grouping

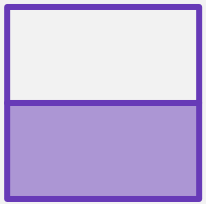
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

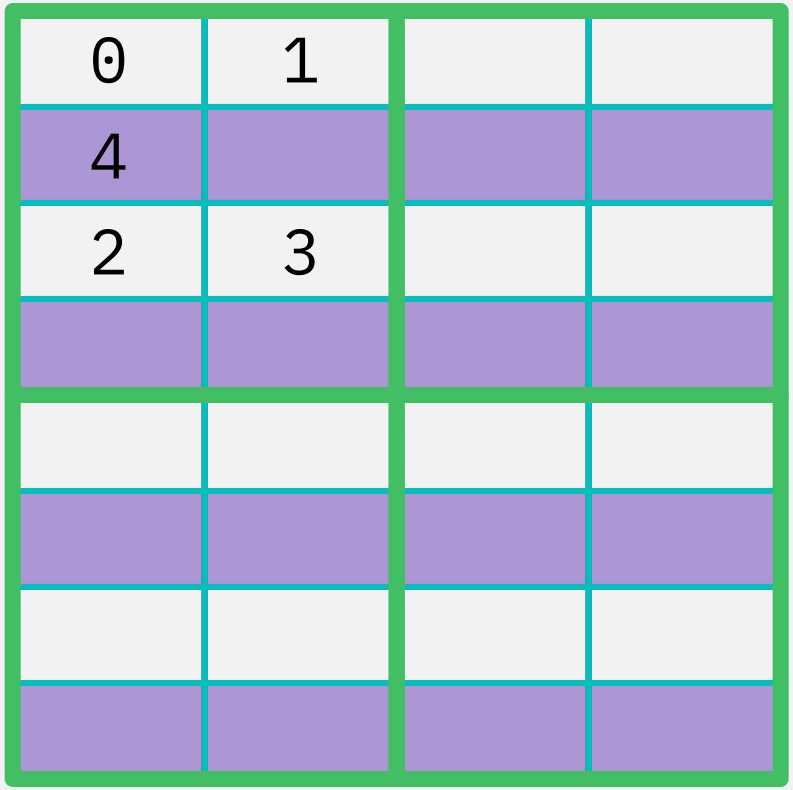




# Dilation with Grouping

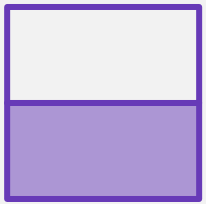
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```

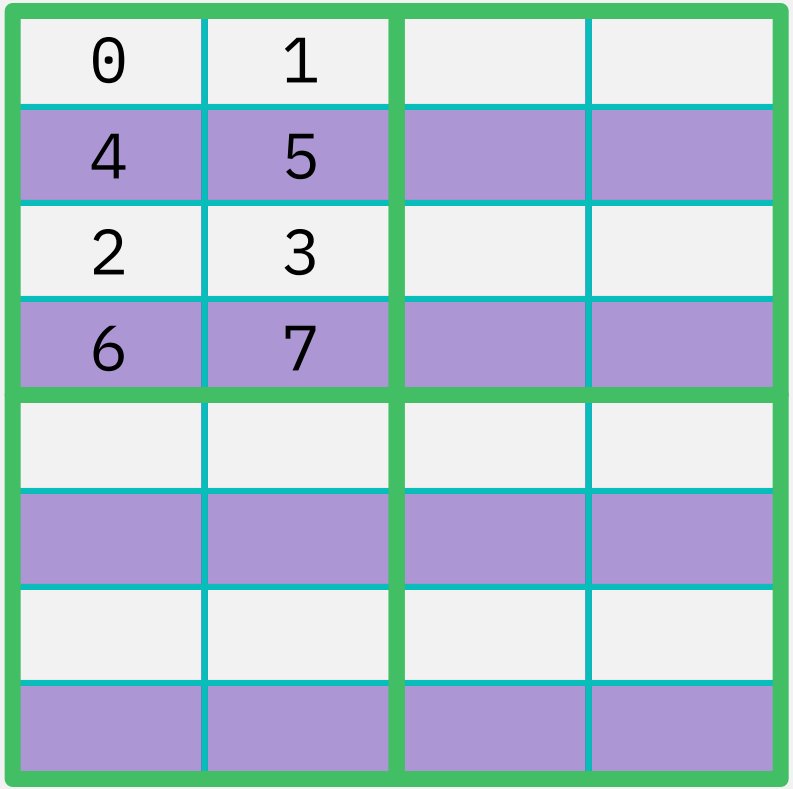




# Dilation with Grouping

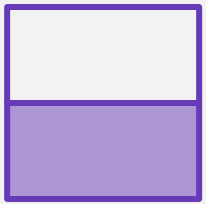
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```





# Dilation with Grouping

```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.group(x:4, y:2)
f.dilate(x:2, y:1)
```



| | | | |
|---|---|---|---|
| 0 | 1 | 8 | 9 |
| 4 | 5 | 12 | 13 |
| 2 | 3 | 10 | 11 |
| 6 | 7 | 14 | 15 |
| 16 | 17 | 24 | 25 |
| 20 | 21 | 28 | 29 |
| 18 | 19 | 26 | 27 |
| 22 | 23 | 30 | 31 |

# Dilation with Grouping in Triton

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation with Grouping in Triton

```
...
@triton.jit
def f_kernel(a_ptr, b_ptr, s, f_ptr, x, y, x_BLOCK_SIZE: tl.constexpr, y_BLOCK_SIZE: tl.constexpr, y_TENSOR_SIZE: tl.constexpr
             x_GROUP_SIZE: tl.constexpr, y_GROUP_SIZE: tl.constexpr, x_DILATE_SIZE: tl.constexpr, y_DILATE_SIZE: tl.constexpr):
  x_pid = tl.program_id(0)
  y_pid = tl.program_id(1)

  ...  # Grouping Remapping
  ...  # Dilation Remapping

  x_block_start = x_pid * x_BLOCK_SIZE
  y_block_start = y_pid * y_BLOCK_SIZE
  for y_iter in range(0, y_BLOCK_SIZE, y_TENSOR_S
    a = tl.load(a_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    b = tl.load(b_ptr + (x_block_start + tl.arang
                y_block_start + y_iter + tl.arang
    f = a * s + b
    tl.store(f_ptr + (x_block_start + tl.arange(0, x_BLOCK_SIZE)[:, None]) * y +
             y_block_start + y_iter + tl.arange(0, y_TENSOR_SIZE)[None, :], f)

...
f_grid = lambda meta: (triton.cdiv(x, meta['x_BLOCK_SIZE']), triton.cdiv(y, meta['y_BLOCK_SIZE']),)
f_kernel[f_grid](a, b, s, f, x, y, 2, 4, 2, 4, 2, 2, 1)
```

```
...  # Dilation Remapping

x_pid = x_pid % x_GROUP_SIZE
y_pid = y_pid % y_GROUP_SIZE
pid = x_pid * y_GROUP_SIZE + y_pid
num_dilate_ids = x_GROUP_SIZE // x_DILATE_SIZE * y_GROUP_SIZE // y_DILATE_SIZE
dilate_id = pid % num_dilate_ids
dilate_offset = pid // num_dilate_ids
y_num_dilate_ids = y_GROUP_SIZE // y_DILATE_SIZE
x_dilate_id = dilate_id // y_num_dilate_ids
y_dilate_id = dilate_id % y_num_dilate_ids
x_dilate_offset = dilate_offset // y_DILATE_SIZE
y_dilate_offset = dilate_offset % y_DILATE_SIZE
x_pid = x_group_id * x_GROUP_SIZE + (x_dilate_id * x_DILATE_SIZE + x_dilate_offset)
y_pid = y_group_id * y_GROUP_SIZE + (y_dilate_id * y_DILATE_SIZE + y_dilate_offset)
```

# Dilation in Decoupled Triton

# Dilation in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.dilate(x:2, y:1);
f.compile_to_kernel();
```

# Dilation in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.dilate(x:2, y:1);
f.compile_to_kernel();
```

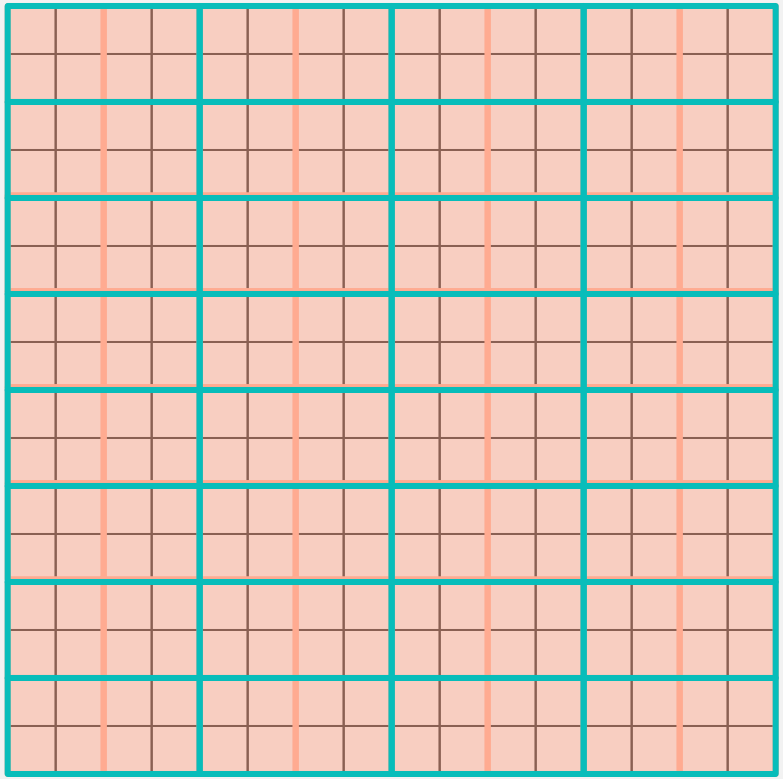
# Dilation in Decoupled Triton

```
In a;
In b;
SIn s;
Func f;
Var x;
Var y;

f[x, y] = a[x, y] * s + b[x, y];

f.block(x:2, y:4);
f.tensorize(x: 2, y:2);
f.group(x:4, y:2);
f.dilate(x:2, y:1);
f.compile_to_kernel();
```

# Aggregate and Sequentialize

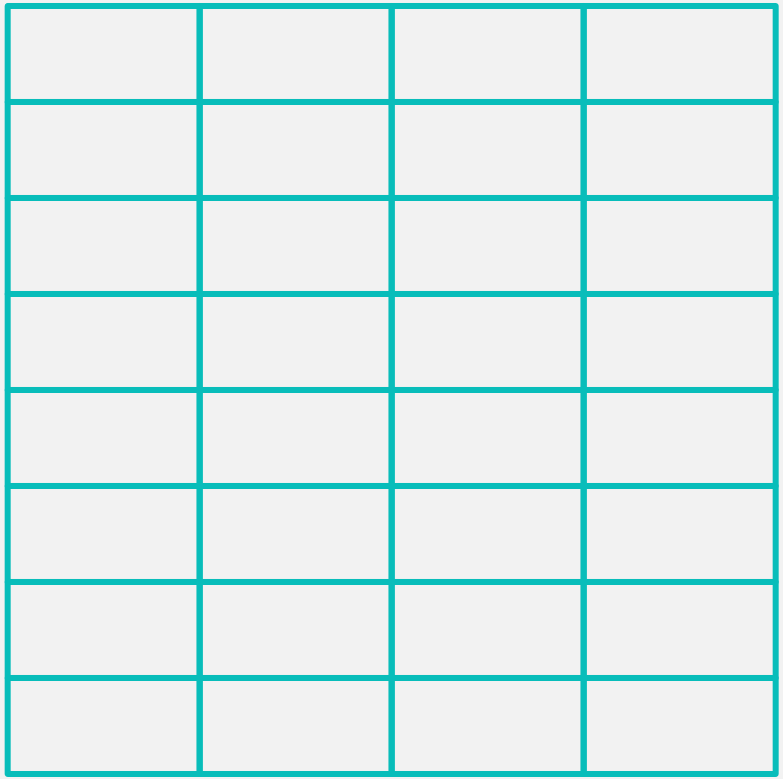
# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



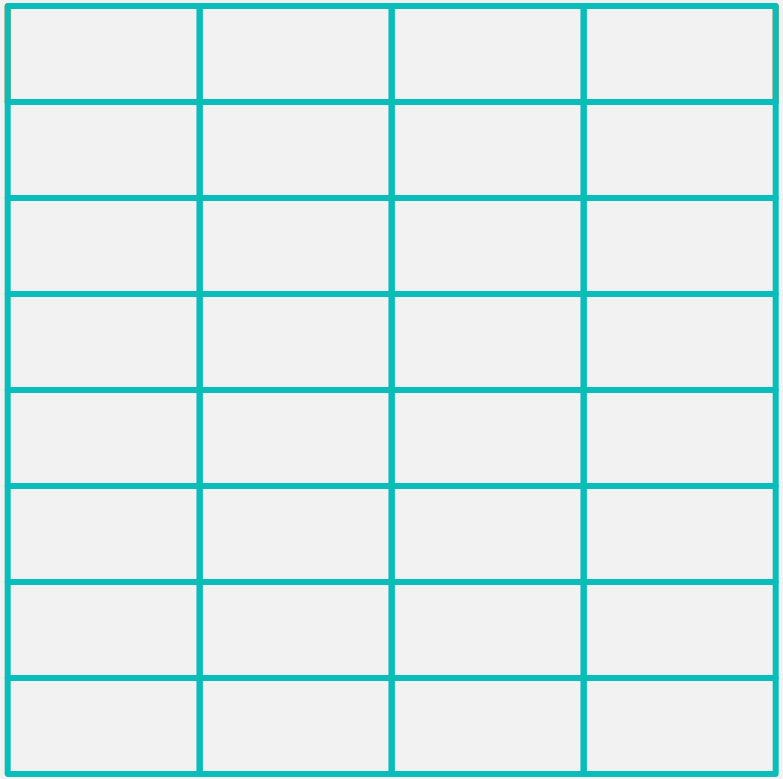
# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 |
| 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |

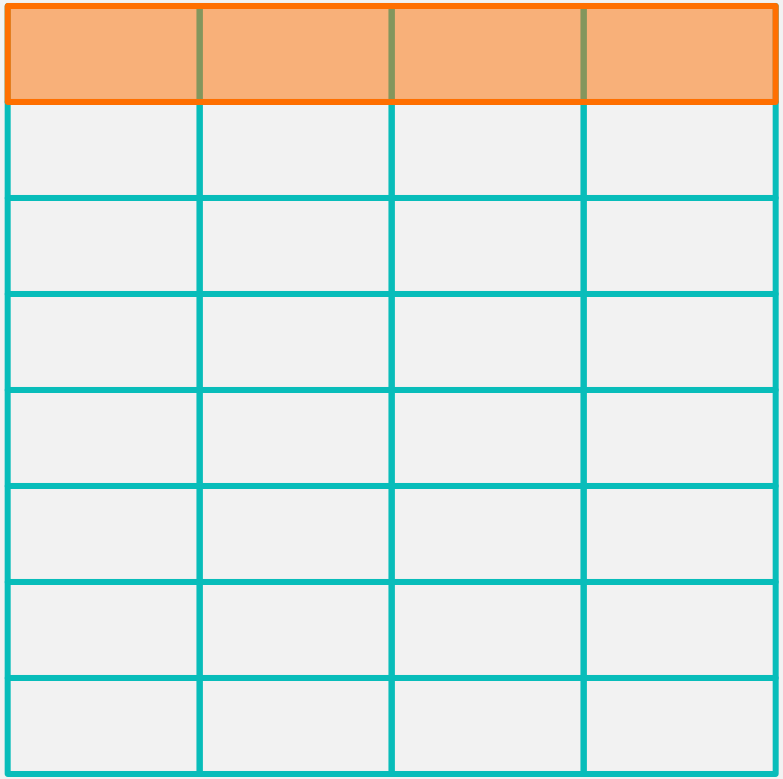
# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



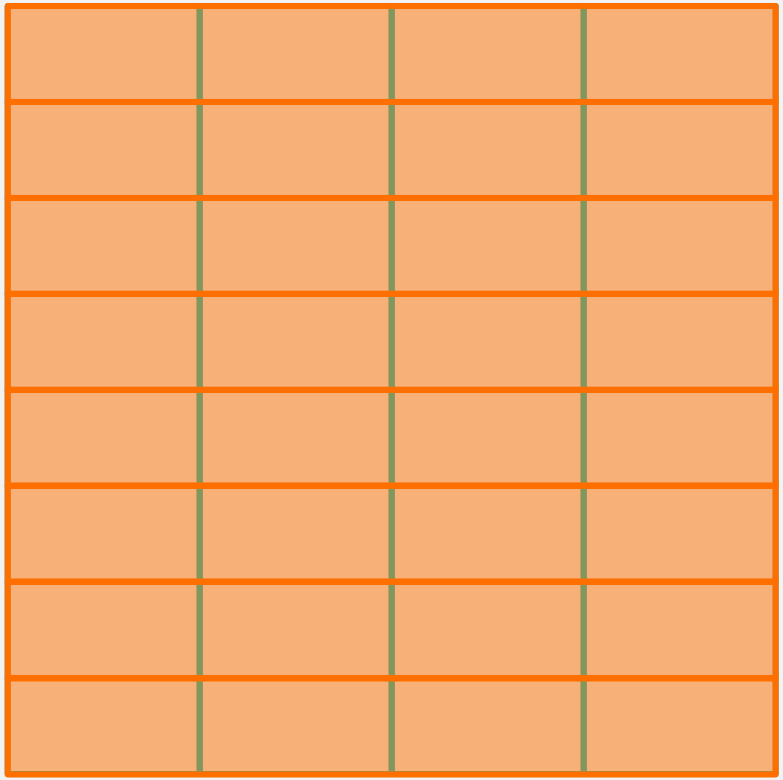
# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



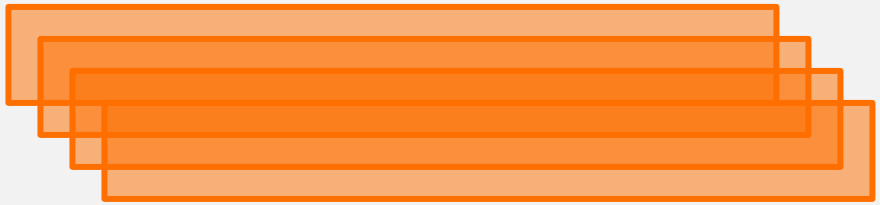
# Aggregate and Sequentialize

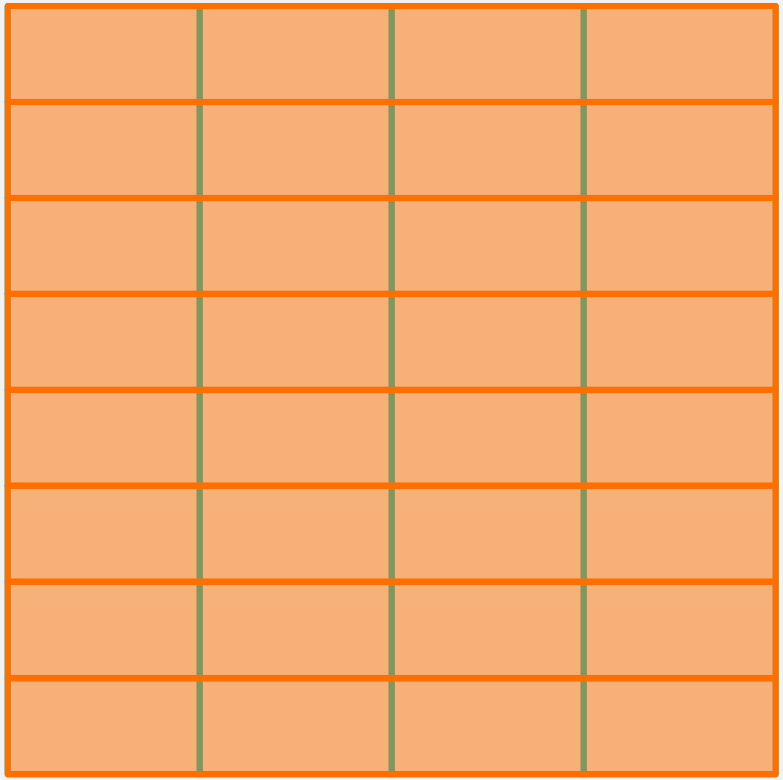
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



# Aggregate and Sequentialize

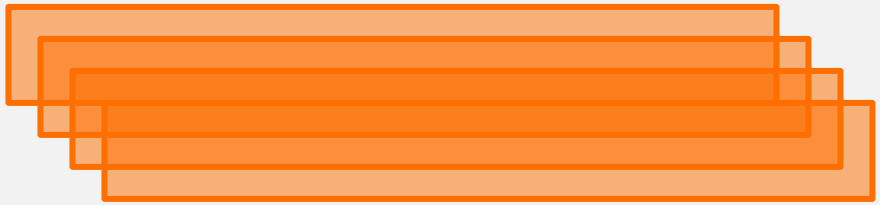
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```

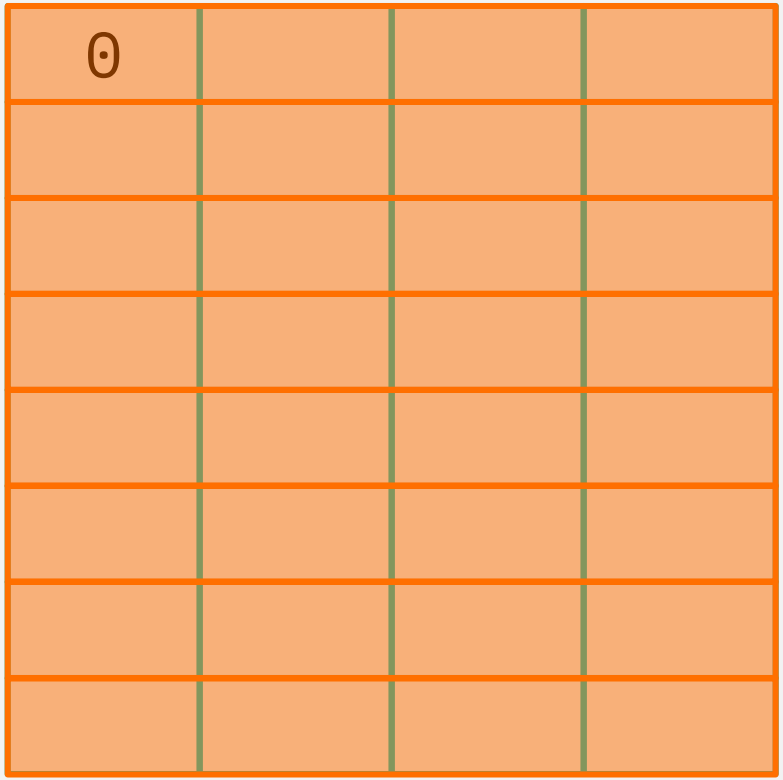




# Aggregate and Sequentialize

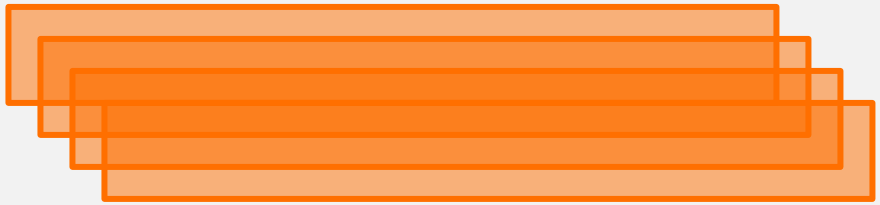
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```

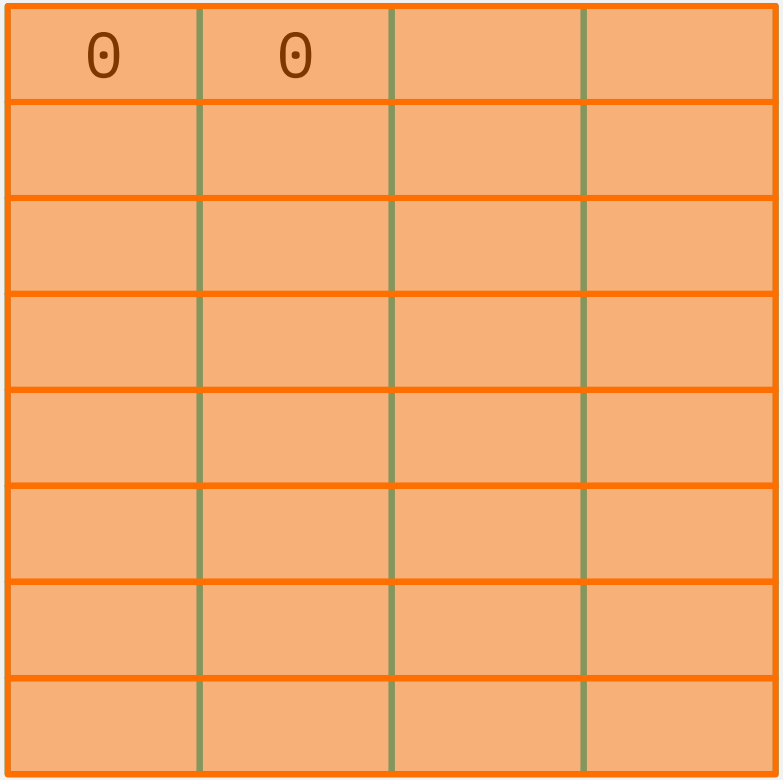




# Aggregate and Sequentialize

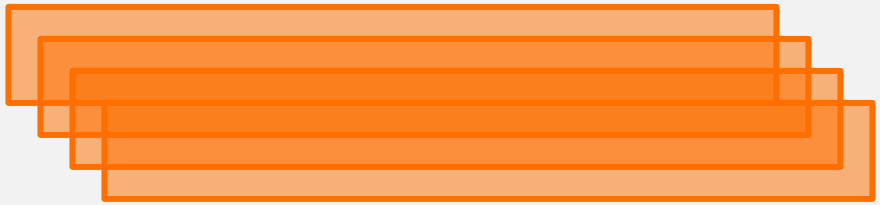
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```

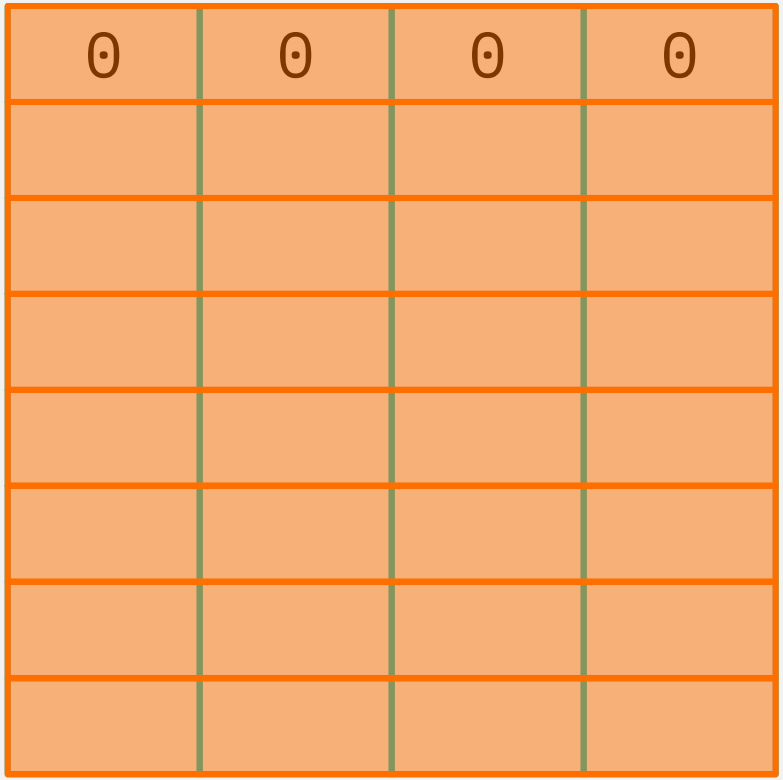




# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```





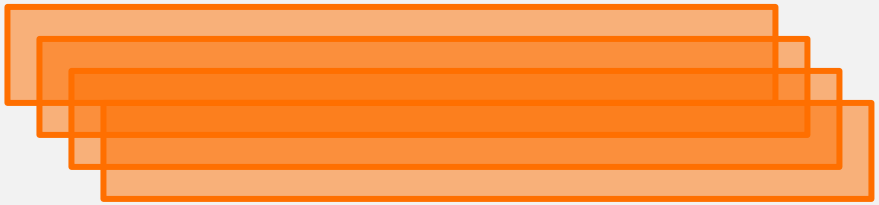
# Aggregate and Sequentialize

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```

| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

# Aggregate and Sequentialize

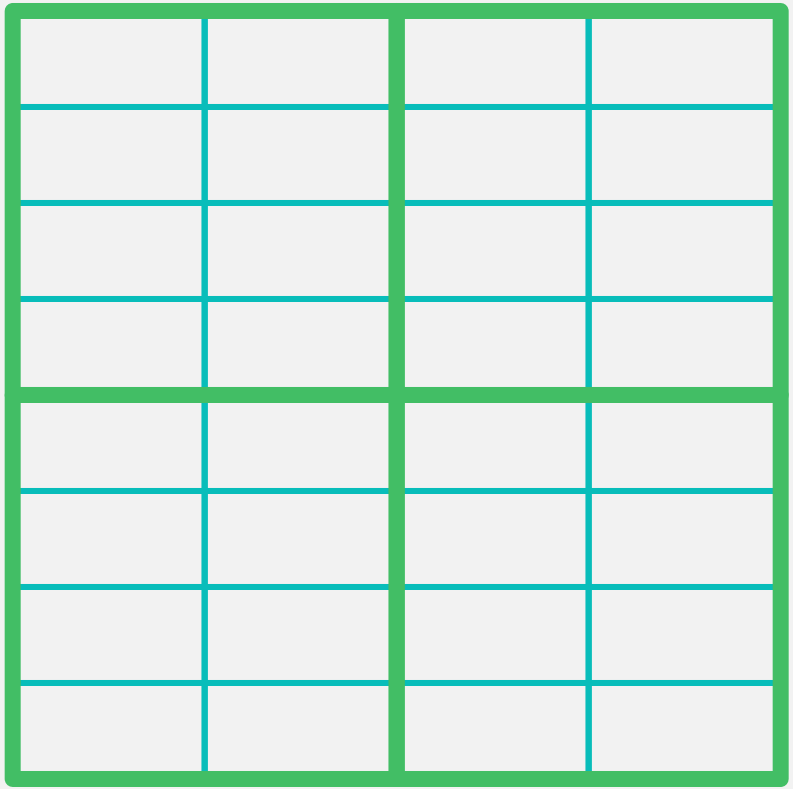
```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.aggregate_and_sequentialize(4)
```



| | | | |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 2 |
| 3 | 3 | 3 | 3 |
| 4 | 4 | 4 | 4 |
| 5 | 5 | 5 | 5 |
| 6 | 6 | 6 | 6 |
| 7 | 7 | 7 | 7 |

# Aggregate and Sequentialize with Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.aggregate_and_sequentialize(4)
```



# Aggregate and Sequentialize with Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.aggregate_and_sequentialize(4)
```

| | | | |
|---|---|---|---|
| 0 | 1 | 8 | 9 |
| 2 | 3 | 10 | 11 |
| 4 | 5 | 12 | 13 |
| 6 | 7 | 14 | 15 |
| 16 | 17 | 24 | 25 |
| 18 | 19 | 26 | 27 |
| 20 | 21 | 28 | 29 |
| 22 | 23 | 30 | 31 |

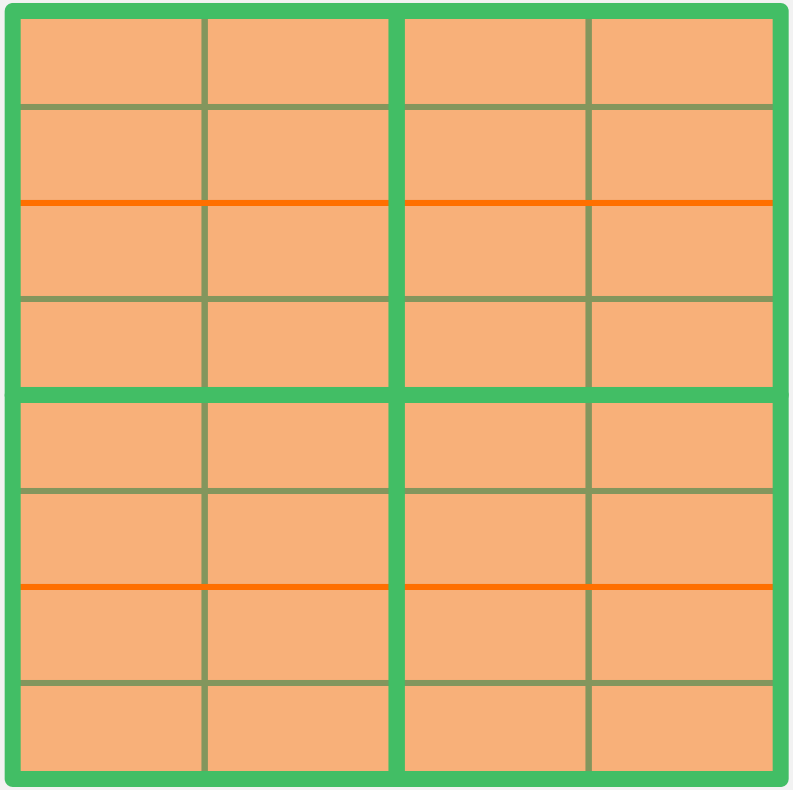
# Aggregate and Sequentialize with Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.aggregate_and_sequentialize(4)
```

| | | | |
|---|---|---|---|
| 0 | 1 | 8 | 9 |
| 2 | 3 | 10 | 11 |
| 4 | 5 | 12 | 13 |
| 6 | 7 | 14 | 15 |
| 16 | 17 | 24 | 25 |
| 18 | 19 | 26 | 27 |
| 20 | 21 | 28 | 29 |
| 22 | 23 | 30 | 31 |

# Aggregate and Sequentialize with Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.aggregate_and_sequentialize(4)
```



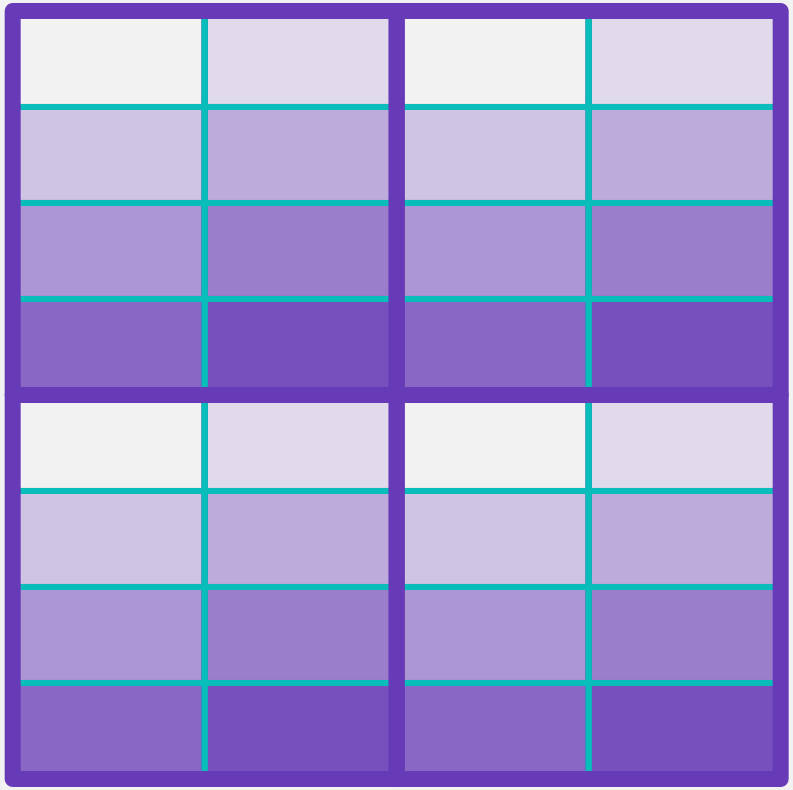
# Aggregate and Sequentialize with Grouping

```
f.block(x:2, y:4)

f.tensorize(x:2, y:2)

f.group(x:4, y:2)

f.aggregate_and_sequentialize(4)
```

| | | | |
|---|---|---|---|
| 0 | 0 | 2 | 2 |
| 0 | 0 | 2 | 2 |
| 1 | 1 | 3 | 3 |
| 1 | 1 | 3 | 3 |
| 4 | 4 | 6 | 6 |
| 4 | 4 | 6 | 6 |
| 5 | 5 | 7 | 7 |
| 5 | 5 | 7 | 7 |

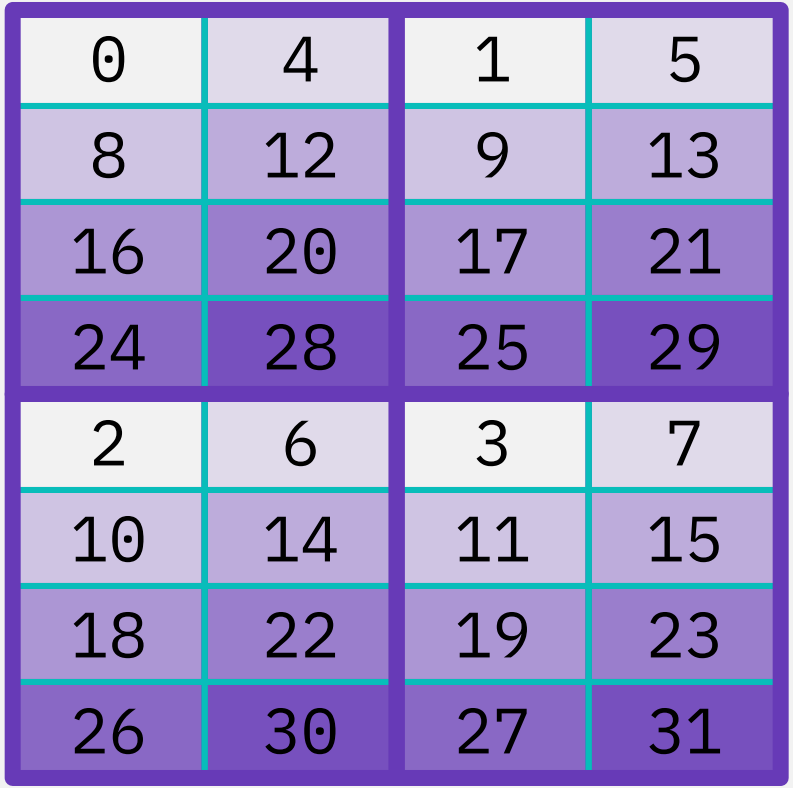
# Aggregate and Sequentialize with Dilation

```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
f.aggregate_and_sequentialize(4)
```



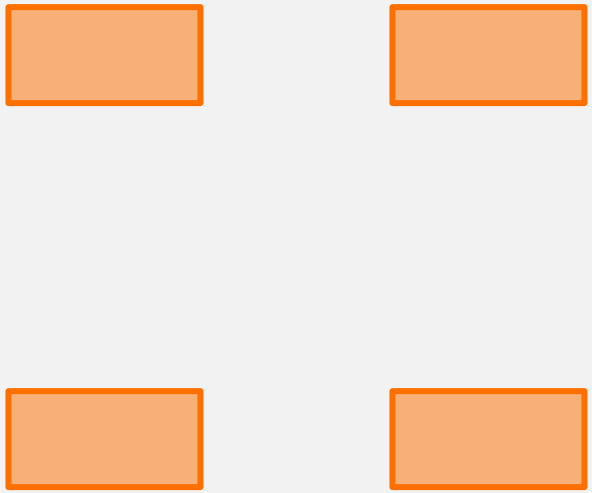
# Aggregate and Sequentialize with Dilation

```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
f.aggregate_and_sequentialize(4)
```



# Aggregate and Sequentialize with Dilation

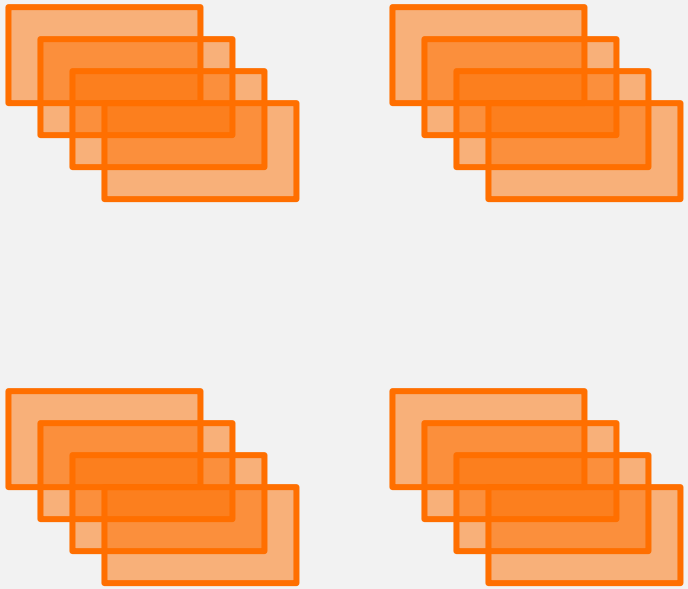
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
f.aggregate_and_sequentialize(4)
```

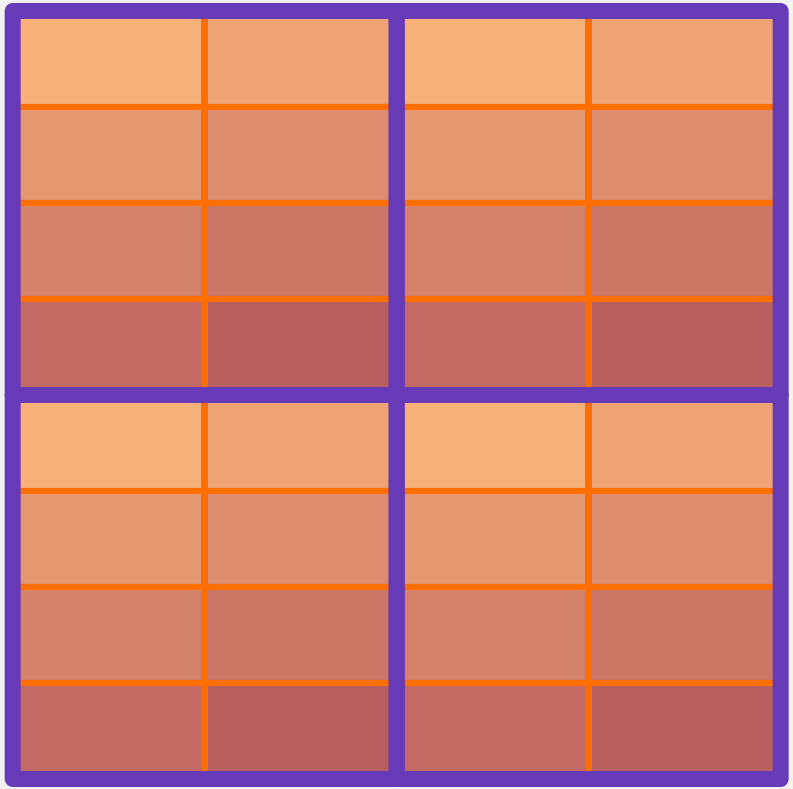


| | | | |
|---|---|---|---|
| 0 | 4 | 1 | 5 |
| 8 | 12 | 9 | 13 |
| 16 | 20 | 17 | 21 |
| 24 | 28 | 25 | 29 |
| 2 | 6 | 3 | 7 |
| 10 | 14 | 11 | 15 |
| 18 | 22 | 19 | 23 |
| 26 | 30 | 27 | 31 |

# Aggregate and Sequentialize with Dilation

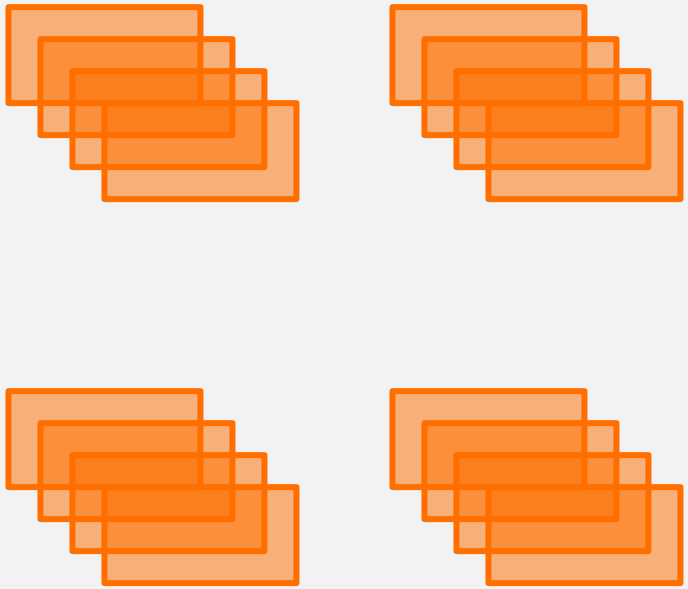
```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
f.aggregate_and_sequentialize(4)
```





# Aggregate and Sequentialize with Dilation

```
f.block(x:2, y:4)
f.tensorize(x:2, y:2)
f.dilate(x:4, y:2)
f.aggregate_and_sequentialize(4)
```



| | | | |
|---|---|---|---|
| 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 |
| 4 | 5 | 4 | 5 |
| 6 | 7 | 6 | 7 |
| 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 |
| 4 | 5 | 4 | 5 |
| 6 | 7 | 6 | 7 |

Fuse at

# Fuse at

```
In a;
In b;
SIn s;
Func f;
Func g;
Var x;
Var y;

g[x, y] = s;
f[x, y] = a[x, y] * g[x,y] + b[x, y];

f.fuse_at(g, x);
f.compile_to_kernel();
```

Fuse the computation of a func into the computation of another func at a particular loop level.

# Schedule Directive Recap

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |
| `tensorize()` | Enables SIMD operations within a program |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |
| `tensorize()` | Enables SIMD operations within a program |
| `group()` | Remaps program IDs to compute the result in a grouped order |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |
| `tensorize()` | Enables SIMD operations within a program |
| `group()` | Remaps program IDs to compute the result in a grouped order |
| `dilate()` | Remaps program IDs to compute the result in a dilated order |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |
| `tensorize()` | Enables SIMD operations within a program |
| `group()` | Remaps program IDs to compute the result in a grouped order |
| `dilate()` | Remaps program IDs to compute the result in a dilated order |
| `aggregate_and_sequentialize()` | Sequentializes block computation of nearby programs |

# Schedule Directive Recap

| | |
|---|---|
| `compile_to_kernel()` | Compiles a Triton kernel |
| `block()` | Enables program blocking |
| `tensorize()` | Enables SIMD operations within a program |
| `group()` | Remaps program IDs to compute the result in a grouped order |
| `dilate()` | Remaps program IDs to compute the result in a dilated order |
| `aggregate_and_sequentialize()` | Sequentializes block computation of nearby programs |
| `fuse_at()` | Fuses a function into another at a certain loop level |

# Performance

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

Liger-Kernel / src / liger_kernel / ops / **geglu.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

Liger-Kernel / src / liger_kernel / ops / **geglu.py**

triton / python / tutorials / **01-vector-add.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

Liger-Kernel / src / liger_kernel / ops / **geglu.py**

triton / python / tutorials / **01-vector-add.py**

triton / python / tutorials / **03-matrix-multiplication.py**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

Liger-Kernel / src / liger_kernel / ops / **geglu.py**

triton / python / tutorials / **01-vector-add.py**

triton / python / tutorials / **03-matrix-multiplication.py**

**torch.addmm**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

Liger-Kernel / src / liger_kernel / ops / **rms_norm.py**

Liger-Kernel / src / liger_kernel / ops / **layer_norm.py**

Liger-Kernel / src / liger_kernel / ops / **jsd.py**

Liger-Kernel / src / liger_kernel / ops / **swiglu.py**

Liger-Kernel / src / liger_kernel / ops / **geglu.py**

triton / python / tutorials / **01-vector-add.py**

triton / python / tutorials / **03-matrix-multiplication.py**

**torch.addmm**

**torch.mm**

# Performance

Can Decoupled Triton generate Triton kernels with performance that is **at least** equivalent to that of manually written Triton kernels?

Yes, with the existing scheduling directives Decoupled Triton can generate competitive Triton kernels for many common ML ops.

More research directions from here ...

# Autotuner

# Autotuner

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...


f.compile_to_kernel();
```

# Autotuner

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.autotune(...);
f.compile_to_kernel();
```

# Autotuner

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.autotune(...);
f.compile_to_kernel();
```

Autotuner to search the space of schedules automatically to determine the best schedule for a given hardware.

# Backward Pass Kernels

# Backward Pass Kernels

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.compile_fwd();
```

**Triton Python**

```
import triton
import triton.language as tl

@triton.jit
def f_fwd_kernel(...):
  ...
```

# Backward Pass Kernels

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.compile_fwd();
f.compile_bwd();
```

**Triton Python**

```
import triton
import triton.language as tl

@triton.jit
def f_fwd_kernel(...):
  ...

@triton.jit
def f_bwd_kernel(...):
  ...
```

# Backward Pass Kernels

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.compile_fwd();
f.compile_bwd();
```

Automatically generate the corresponding Triton kernel for the backward pass.

**Triton Python**

```
import triton
import triton.language as tl

@triton.jit
def f_fwd_kernel(...):
  ...

@triton.jit
def f_bwd_kernel(...):
  ...
```

# Backward Pass Kernels

**Decoupled Triton**

```
Func f;
Var x;
Var y;

f[x, y] = ...

f.compile_fwd();
f.compile_bwd();
```

Automatically generate the corresponding Triton kernel for the backward pass.

It will need different scheduling.

**Triton Python**

```
import triton
import triton.language as tl

@triton.jit
def f_fwd_kernel(...):
  ...

@triton.jit
def f_bwd_kernel(...):
  ...
```

# User Defined Program ID Remapping

# User Defined Program ID Remapping

group(), dilate(), and aggregate_and_sequentialize() are patterns for remapping program ID to result block.

# User Defined Program ID Remapping

group(), dilate(), and aggregate_and_sequentialize() are patterns for remapping program ID to result block.

Can we create a system to allow the user to define a mapping function that maps program ID to result block?

# User Defined Program ID Remapping

group(), dilate(), and aggregate_and_sequentialize() are patterns for remapping program ID to result block.

Can we create a system to allow the user to define a mapping function that maps program ID to result block?

Would such a system be useful?

# Program Synchronization

# Program Synchronization

Program synchronization to allow reductions of values across different programs.