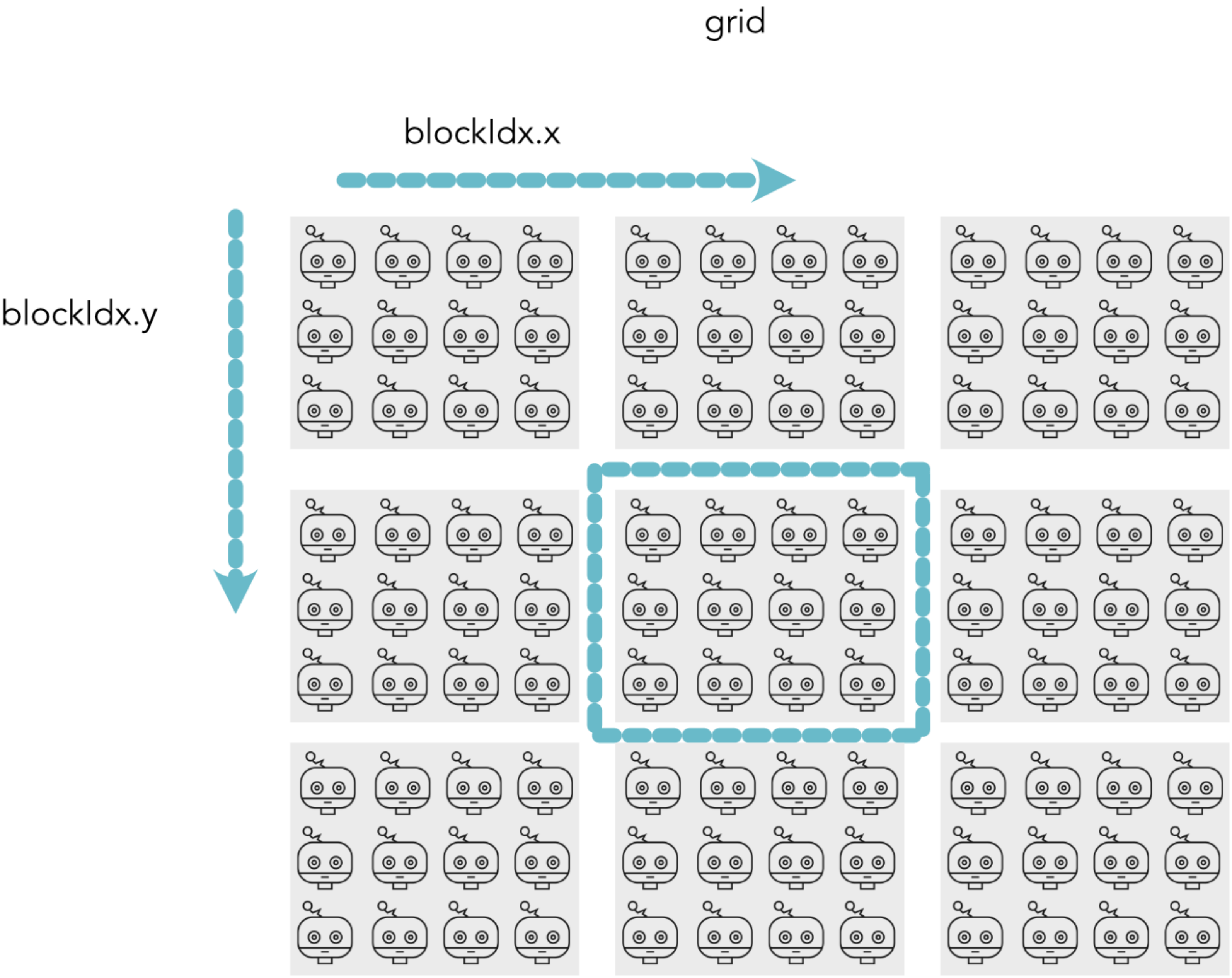
Module 3.3 - CUDA 2

thread


grid
blockIdx.x
blockIdx.y

Stack

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

Thread Names

Printing code

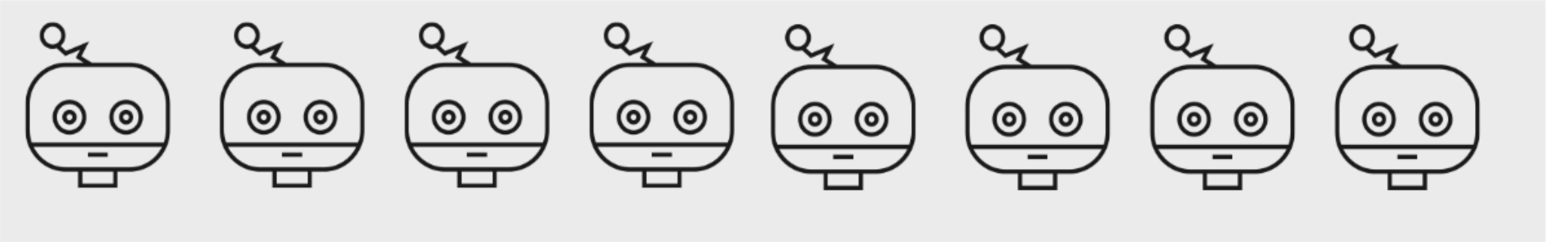
```python
def printer(a):
    print(cuda.threadIdx.x, cuda.threadIdx.y)
    a[:] = 10 + 50
# printer = cuda.jit()(printer)
# a = np.zeros(10)
# printer[1, (10, 10)](a)
```

Output

Output

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

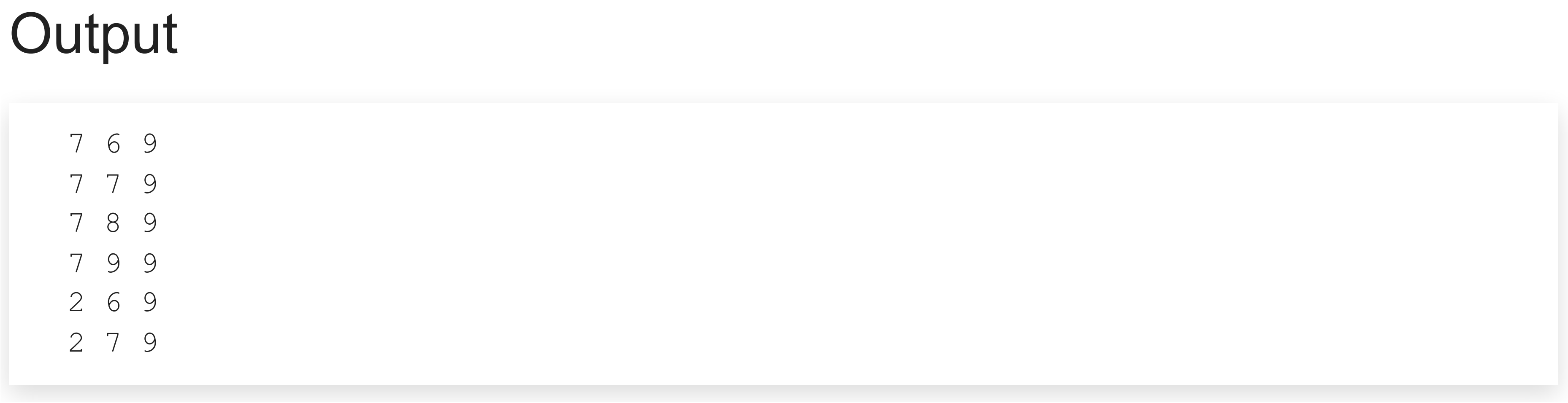
block



Thread Names

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

Output



Output

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

What’s my name?

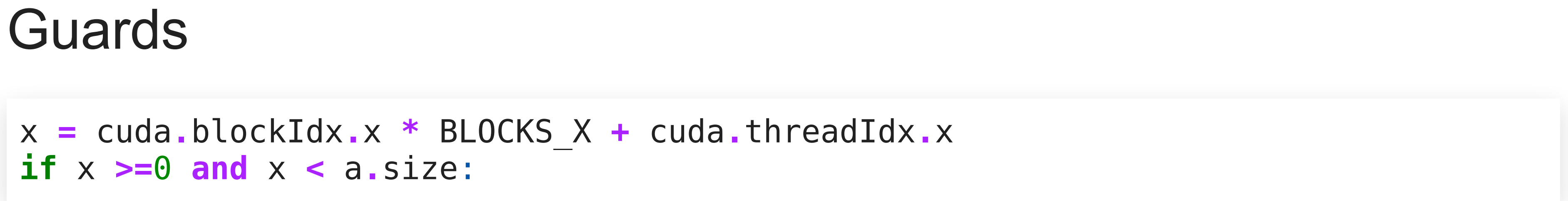
```
BLOCKS_X = 32
BLOCKS_Y = 32
THREADS_X = 10
THREADS_Y = 10
def fn(a):
    x = cuda.blockIdx.x * THREADS_X + cuda.threadIdx.x
    y = cuda.blockIdx.y * THREADS_Y + cuda.threadIdx.y

# fn = cuda.jit()(fn)
# fn[(BLOCKS_X, BLOCKS_Y), (THREADS_X, THREADS_Y)](a)
```

Simple Map

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

Guards



Guards

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

Communication

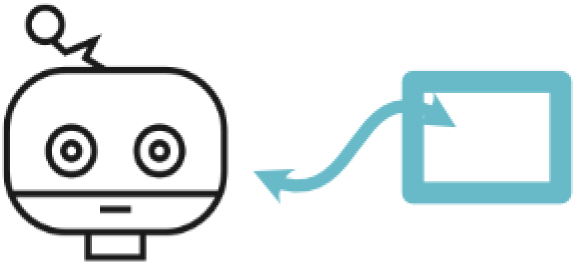
Names

- Why do the names matter?
- Determine communication
- Locality is key for speed.

Memory

- CUDA memory hierarchy
- Local > Shared > Global
- Goal: minimize global reads and writes

thread local memory



Example

```python
def local_fn(out, a):
    i = cuda.threadIdx.x
    local = cuda.local.array(10, numba.int32)
    local[0] = 10
    local[5] = local[0] + 10
    out[i] = local[5]

# local_fn = cuda.jit()(local_fn)
# local_fn[BLOCKS, THREADS](out, a)
# out
```

Constraints

- Memory must be typed
- Memory must be *constant* size
- Memory must be relatively small

BAD Example

```
def local_fn(out, a):
    local = cuda.local.array(a.size, numba.int32)
    local[0] = 10
    local[5] = 20

local_fn = cuda.jit()(local_fn)
local_fn[BLOCKS, THREADS](out, a)
```

Example

```
def block_fn(out, a):
    shared = cuda.shared.array(10, numba.int32)
    shared[0] = 10
    shared[5] = 20

# block_fn = cuda.jit()(block_fn)
# block_fn[BLOCKS, THREADS](out, a)
```

Communication

- Threads can read from shared memory
- Need to `sync` to ensure it is written before you read

Real Example

```python
def block_fn(out, a):
    shared = cuda.shared.array(THREADS, numba.int32)
    i = cuda.threadIdx.x
    shared[i] = a[i]
    cuda.syncthreads()
    out[i+1 % THREADS] = shared[i]

# block_fn = cuda.jit()(block_fn)
# block_fn[1, THREADS](out, a)
# out
```

Constraints

- Memory must be typed

- Memory must be *constant* size

- Memory must be relatively small

Algorithms

Thinking about Speed

- Algorithms: Reduce computation complexity
- Typical: Remove loops, code operations

Sliding Average

Compute sliding average over a list

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

Local Sum

Compute sliding average over a list

```
def slide_py(out, a):
    for i in range(out.size):
        out[i] = 0
        for j in range(sub_size):
            out[i] += a[i + j]
        out[i] = out[i] / sub_size
```

Planning for CUDA

- Count up the memory accesses
- How many global / shared / local reads?
- Can we make move things to be more local?

Basic CUDA

```python
# @cuda.jit
def slide_cuda(out, a):
   i = cuda.threadIdx.x
   if i + sub_size < a.size:
       out[i] = 0
       for j in range(sub_size):
            out[i] += a[i + j]
       out[i] = out[i] / sub_size
```

Planning for CUDA

- `sub_size` global reads per thread
- `sub_size` global writes per thread
- Each is being read too many times.

Strategy

- Use blocks to move from global to shared
- Use thread to move from shared to local

Better CUDA

One global write per thread

```python
# @cuda.jit
def slide_cuda(out, a):
    i = cuda.threadIdx.x
    if i + sub_size < a.size:
        temp = 0
        for j in range(sub_size):
            temp += a[i + j]
        out[i] = temp / sub_size
```

Pattern

Copy from global to shared

```
local_idx = cuda.threadIdx.x
shared[local_idx] = a[i]
cuda.syncthreads()
```

Better CUDA

```
# @cuda.jit
def slide_cuda(out, a):
    shared = cuda.shared.array(THREADS + sub_size)
    i = cuda.threadIdx.x
    if i + sub_size < a.size:
        shared[i] = a[i]
        if i < sub_size and i + THREADS < a.size:
            shared[i  + THREADS] = a[i + THREADS]
        cuda.syncthreads()
        temp = 0
        for j in range(sub_size):
            temp += shared[i + j]
        out[i] = temp / sub_size
```

Counts

- Significantly reduced global reads and writes
- Needed block shared memory to do this