PyOpenCL Tutorial

Andreas Klöckner

Computer Science
University of Illinois at Urbana-Champaign

May 8, 2014

Outline

CL vs CUDA side-by-side

CUDA source code:

```
__global__ void transpose(
    float *A_t, float *A,
    int a_width, int a_height)
{
  int base_idx_a   =
    blockIdx.x * BLK_SIZE +
    blockIdx.y * A_BLOCK_STRIDE;
  int base_idx_a_t =
    blockIdx.y * BLK_SIZE +
    blockIdx.x * A_T_BLOCK_STRIDE;

  int glob_idx_a =
    base_idx_a + threadIdx.x
    + a_width * threadIdx.y;
  int glob_idx_a_t =
    base_idx_a_t + threadIdx.x
    + a_height * threadIdx.y;

  __shared__ float A_shared[BLK_SIZE][BLK_SIZE+1];

  A_shared[threadIdx.y][threadIdx.x] =
    A[glob_idx_a];

  __syncthreads();

  A_t[glob_idx_a_t] =
    A_shared[threadIdx.x][threadIdx.y];
}
```

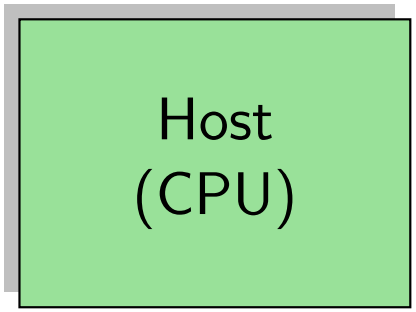
OpenCL source code:

```
void transpose(
  __global float *a_t, __global float *a,
  unsigned a_width, unsigned a_height)
{
  int base_idx_a   =
    get_group_id(0) * BLK_SIZE +
    get_group_id(1) * A_BLOCK_STRIDE;
  int base_idx_a_t =
    get_group_id(1) * BLK_SIZE +
    get_group_id(0) * A_T_BLOCK_STRIDE;

  int glob_idx_a =
    base_idx_a + get_local_id(0)
    + a_width * get_local_id(1);
  int glob_idx_a_t =
    base_idx_a_t + get_local_id(0)
    + a_height * get_local_id(1);

  __local float a_local[BLK_SIZE][BLK_SIZE+1];

  a_local[get_local_id(1)*BLK_SIZE+get_local_id(0)] =
    a[glob_idx_a];

  barrier(CLK_LOCAL_MEM_FENCE);

  a_t[glob_idx_a_t] =
    a_local[get_local_id(0)*BLK_SIZE+get_local_id(1)];
}
```

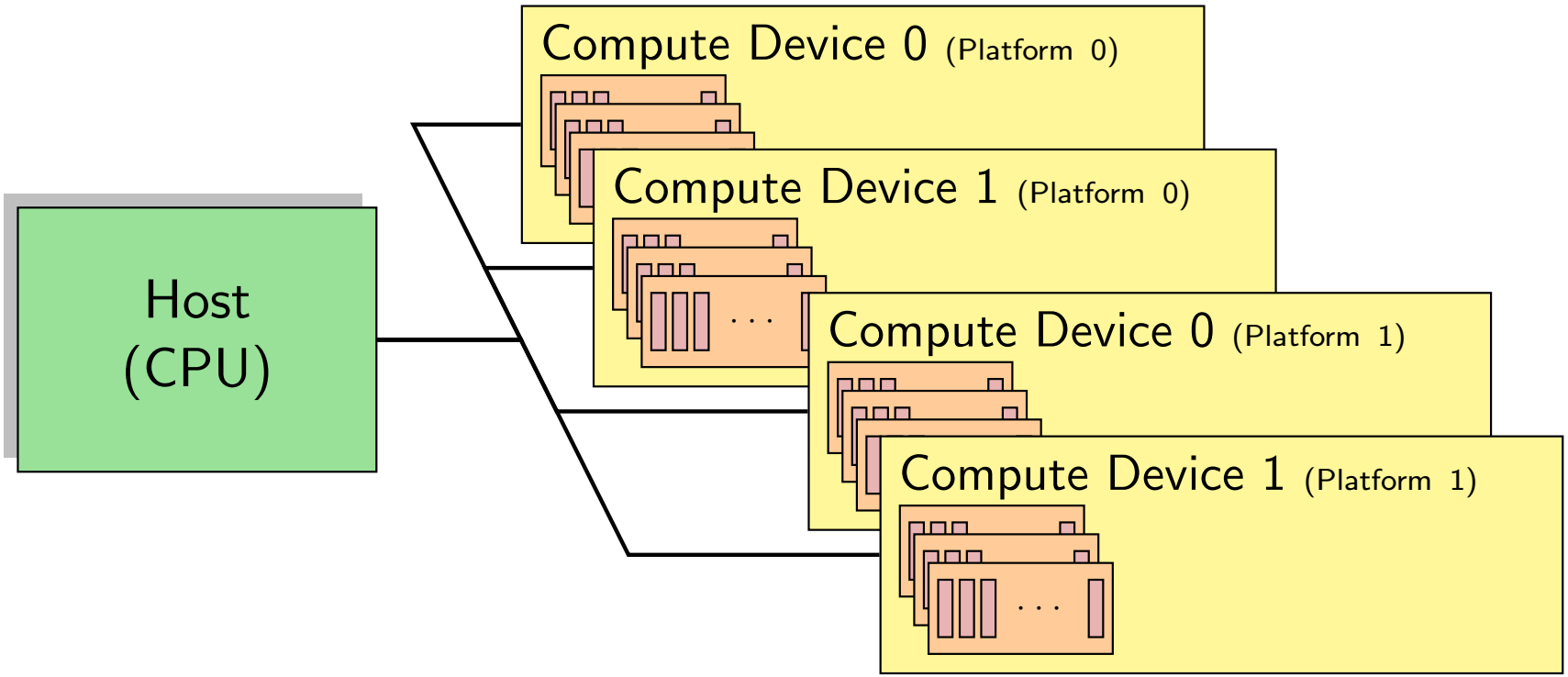
OpenCL ↔ CUDA: A dictionary

OpenCL	CUDA
Grid	Grid
Work Group	Block
Work Item	Thread
`__kernel`	`__global__`
`__global`	`__device__`
`__local`	`__shared__`
`__private`	`__local__`
`image`*n*`d_t`	`texture<type, `*n*`, ...>`
`barrier(LMF)`	`__syncthreads()`
`get_local_id(012)`	`threadIdx.xyz`
`get_group_id(012)`	`blockIdx.xyz`
`get_global_id(012)`	– (reimplement)

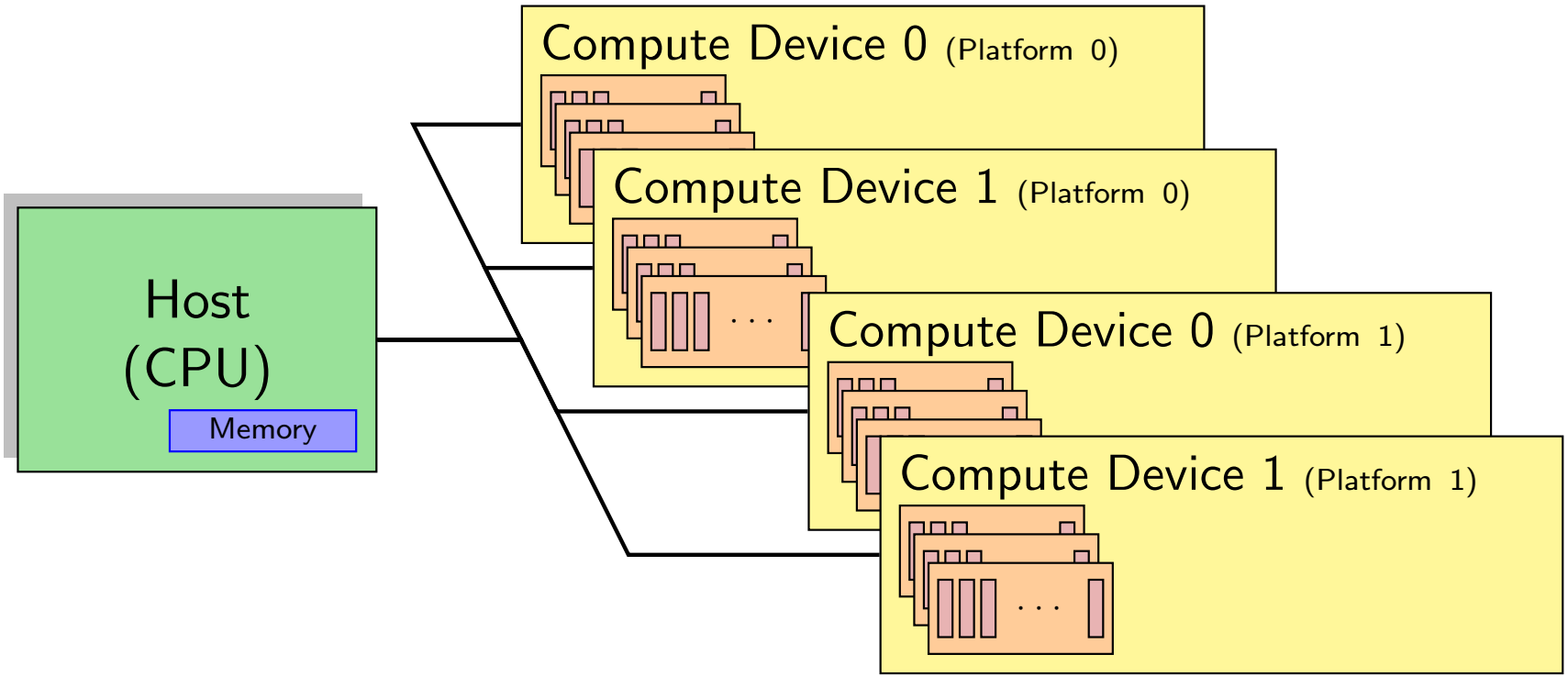
OpenCL: Computing as a Service



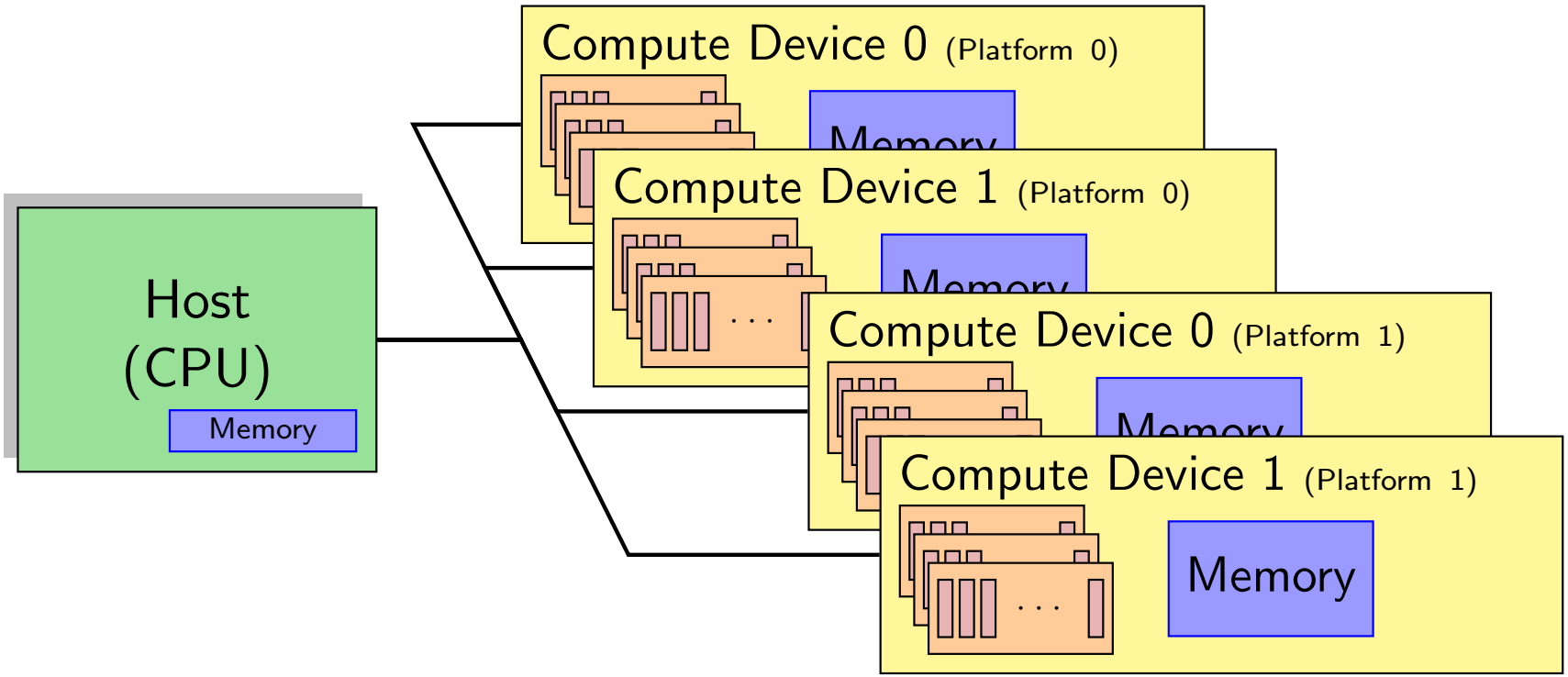
OpenCL: Computing as a Service



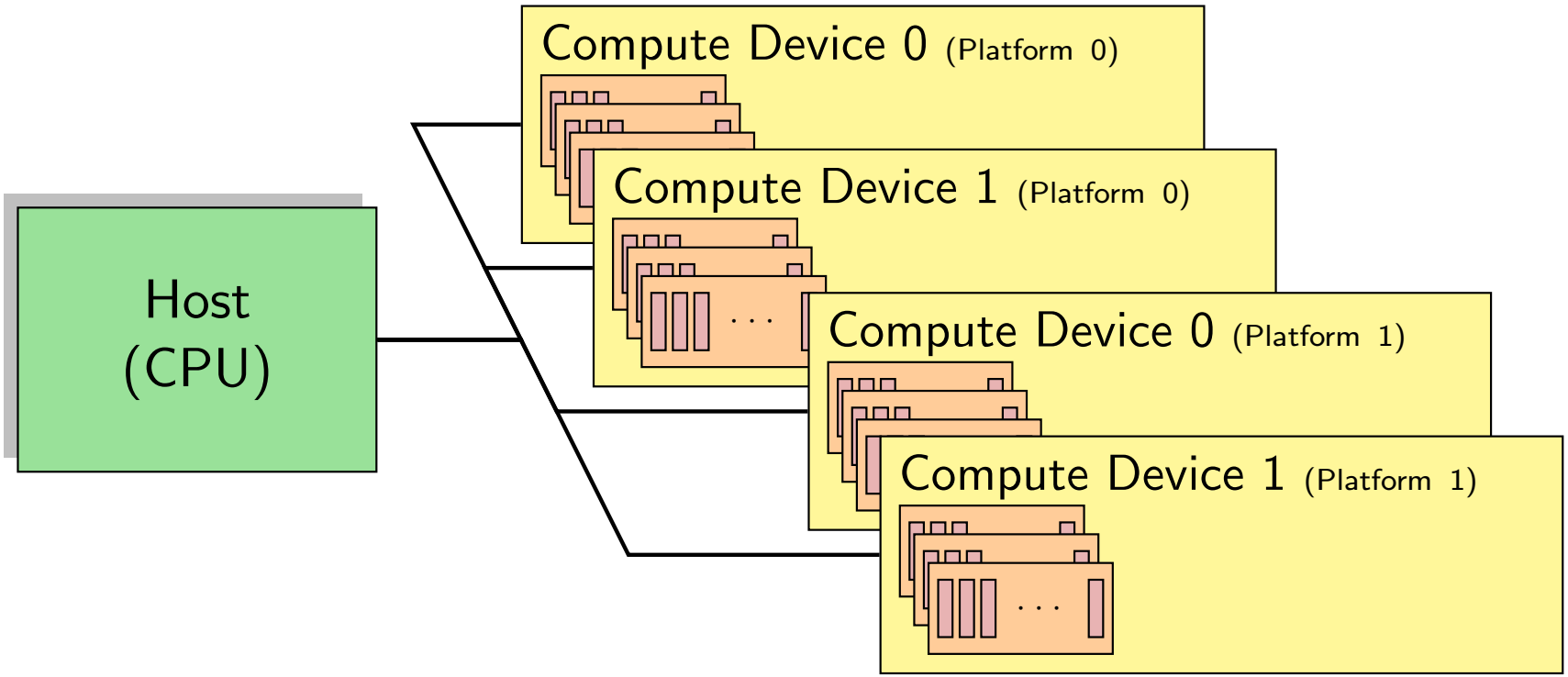
OpenCL: Computing as a Service



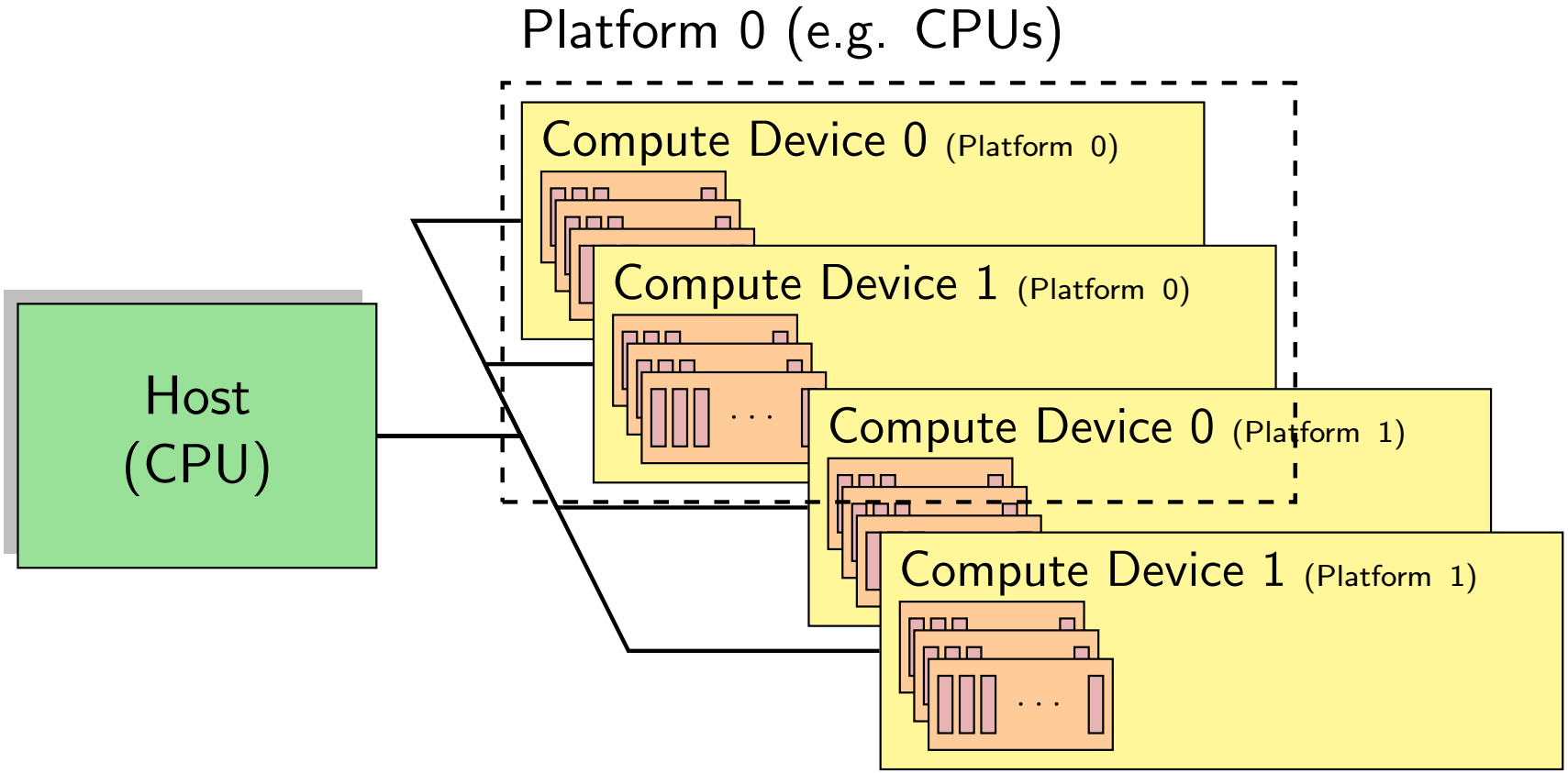
OpenCL: Computing as a Service



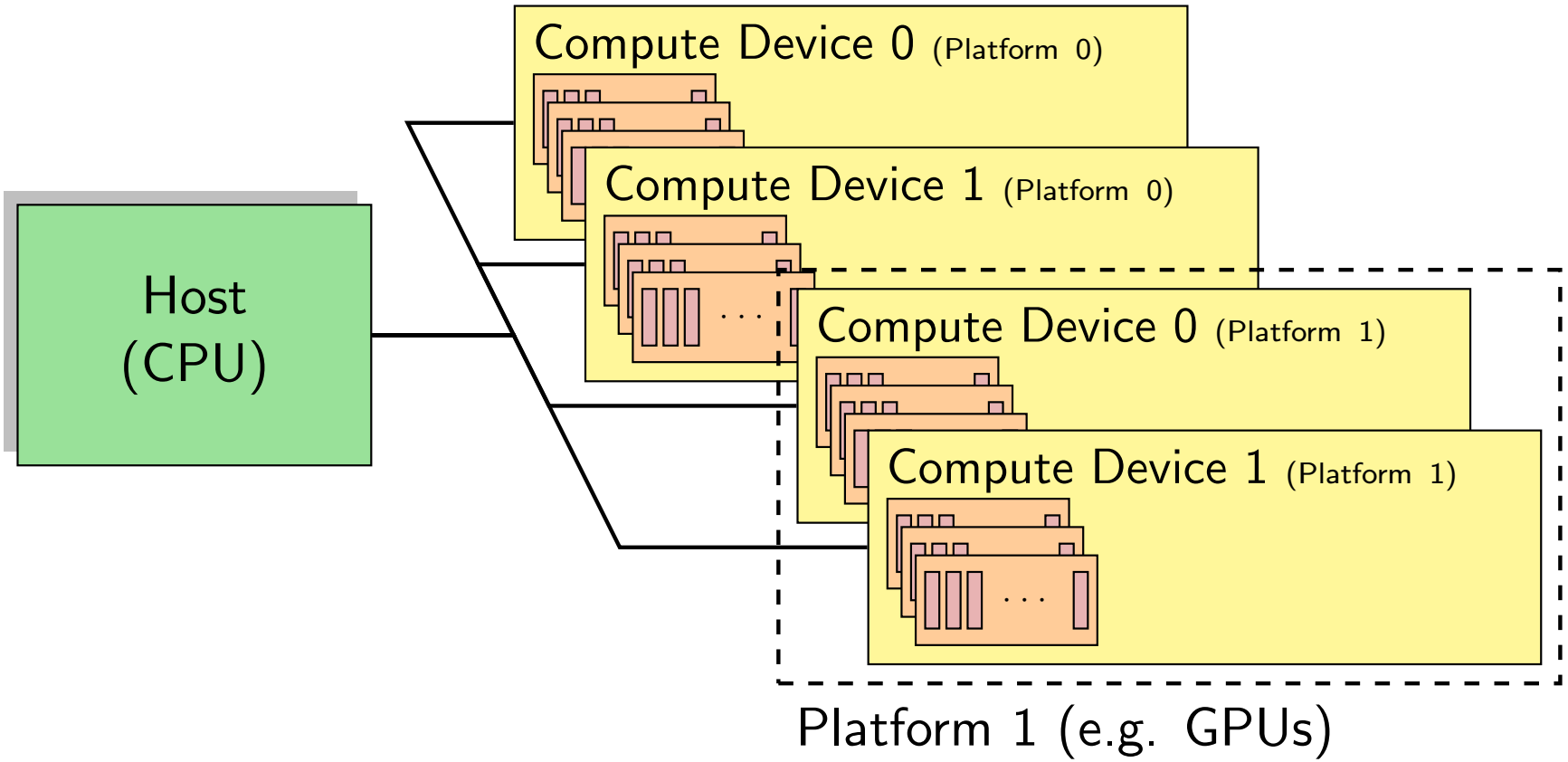
OpenCL: Computing as a Service



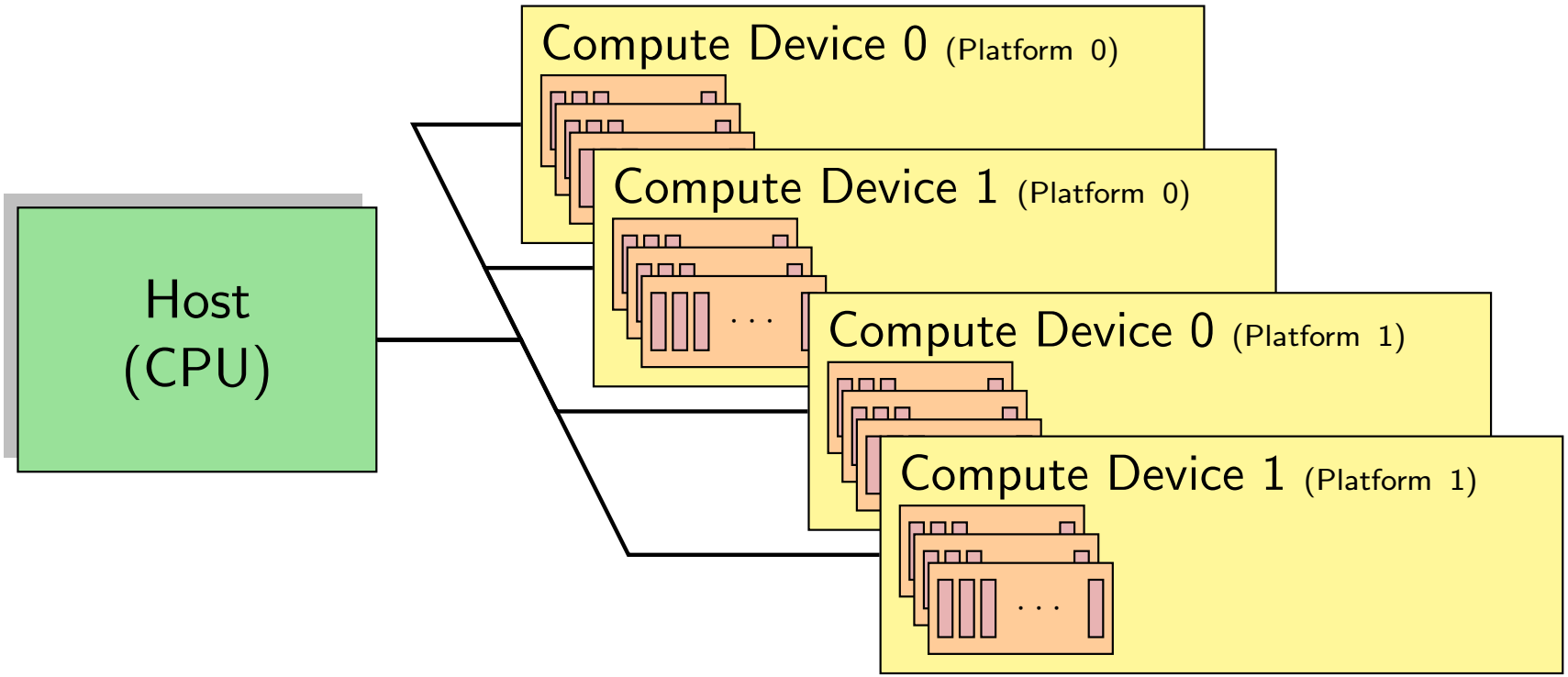
OpenCL: Computing as a Service



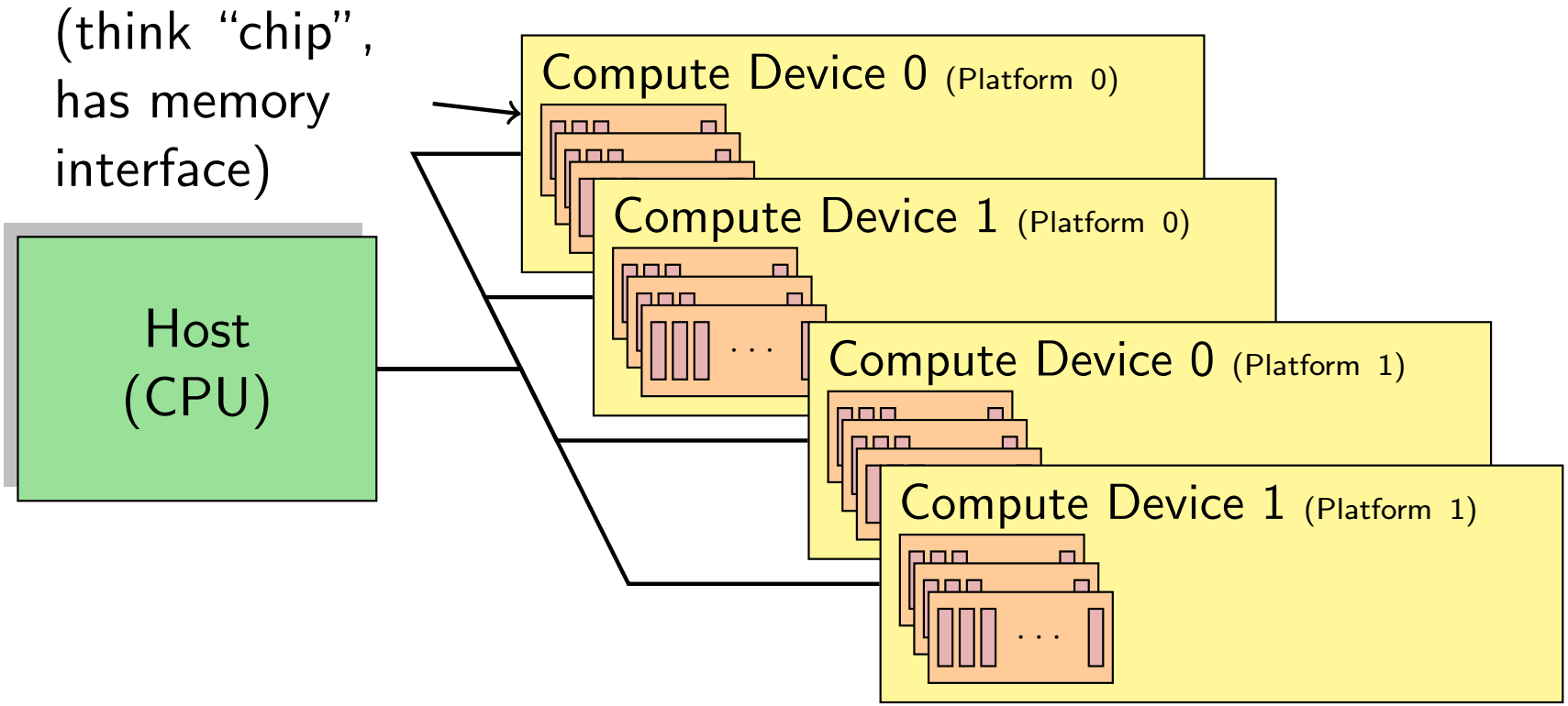
OpenCL: Computing as a Service



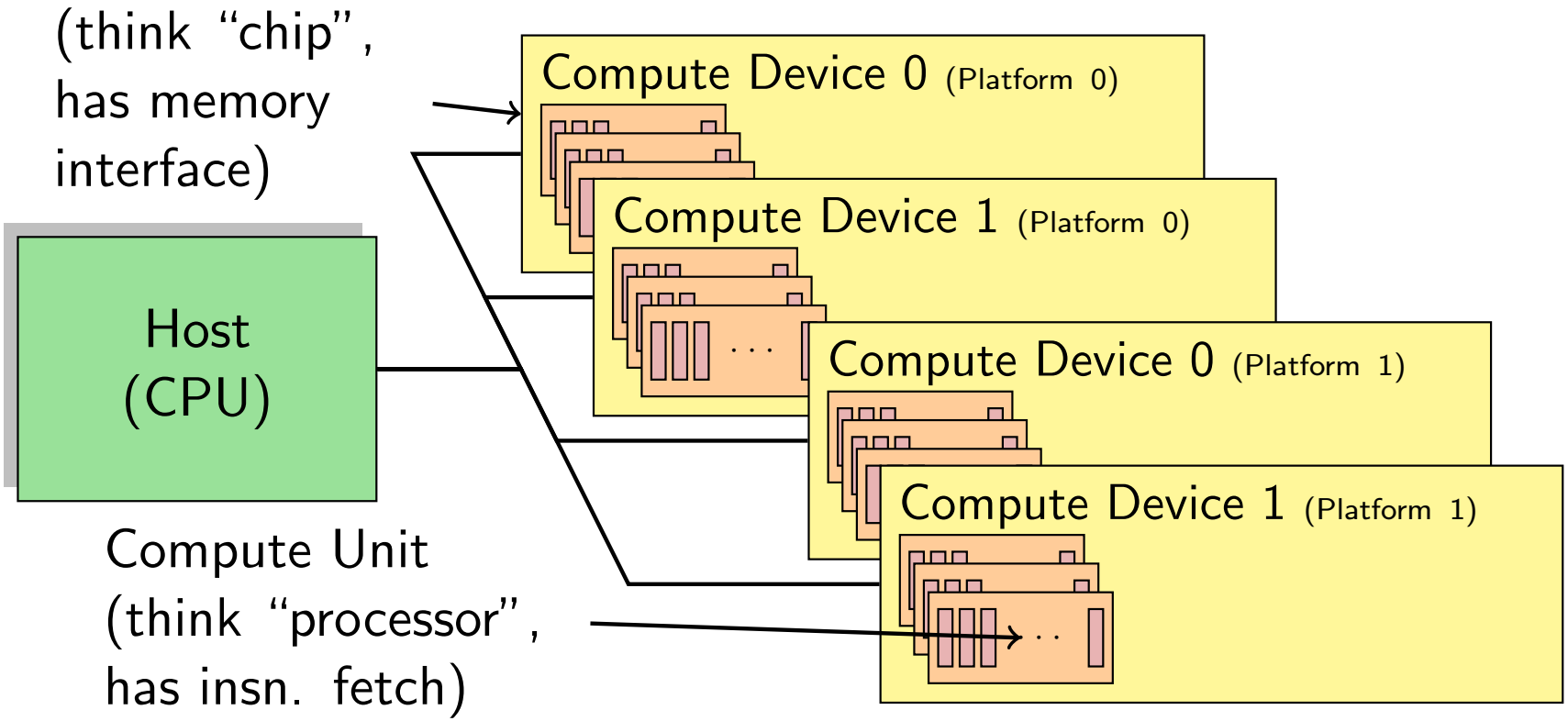
OpenCL: Computing as a Service



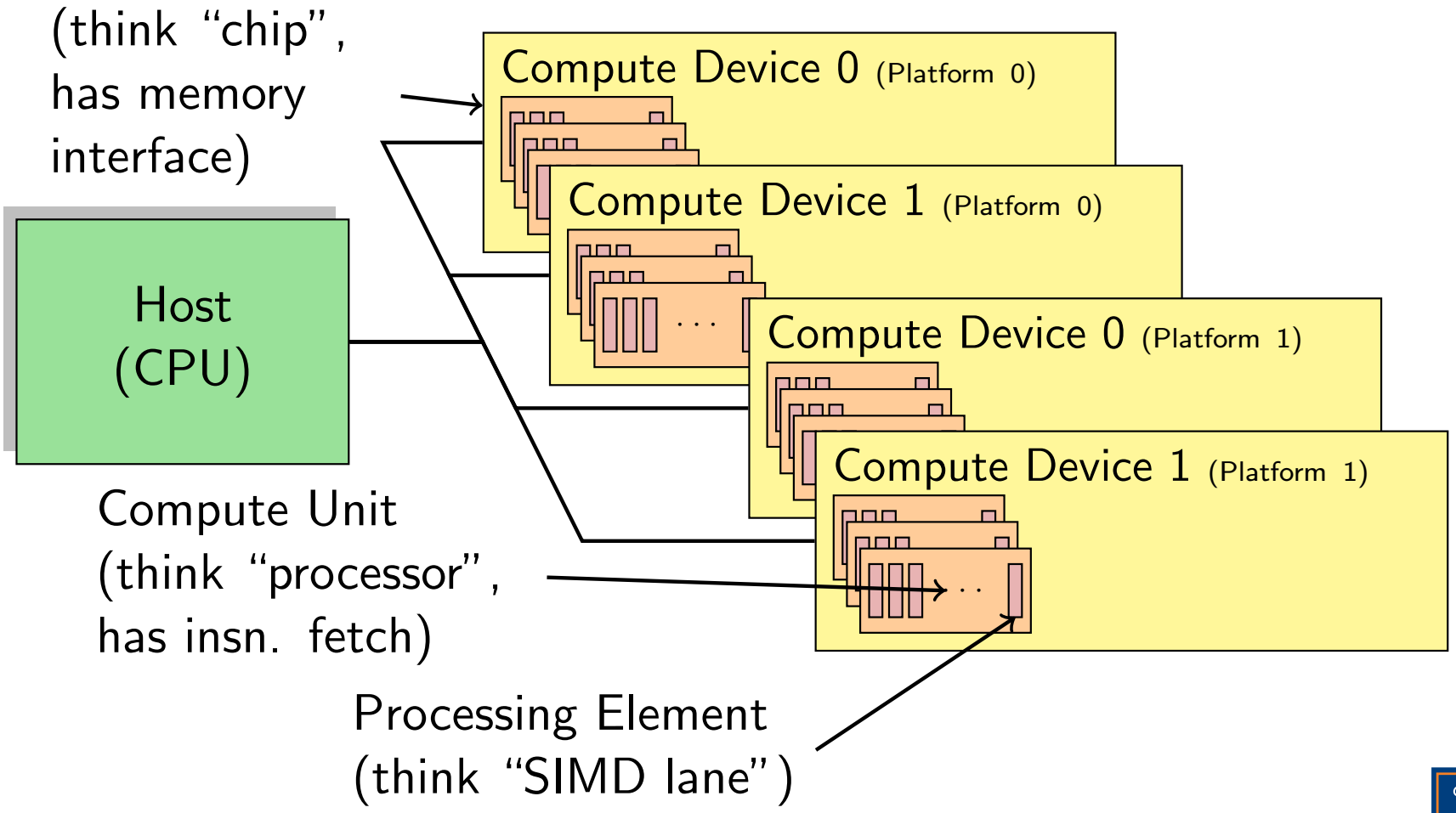
OpenCL: Computing as a Service



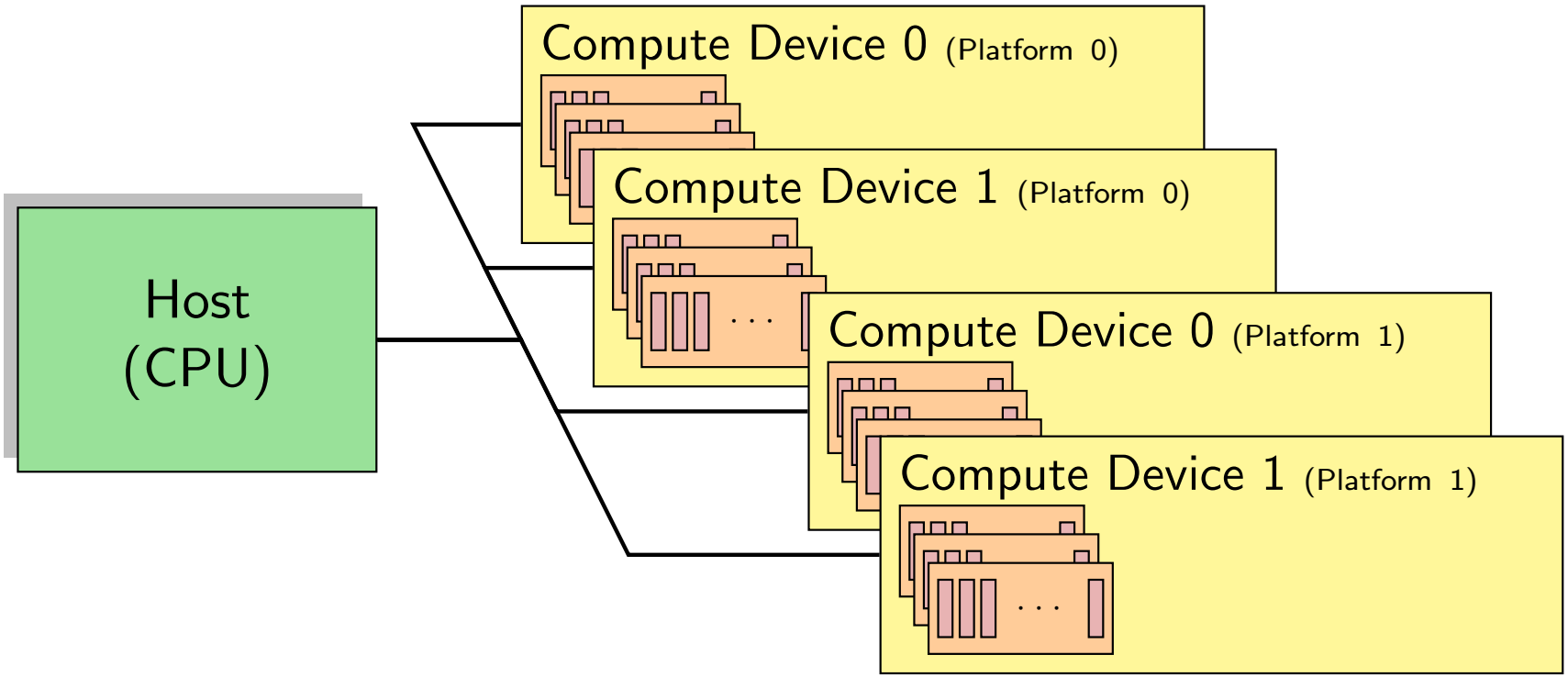
OpenCL: Computing as a Service



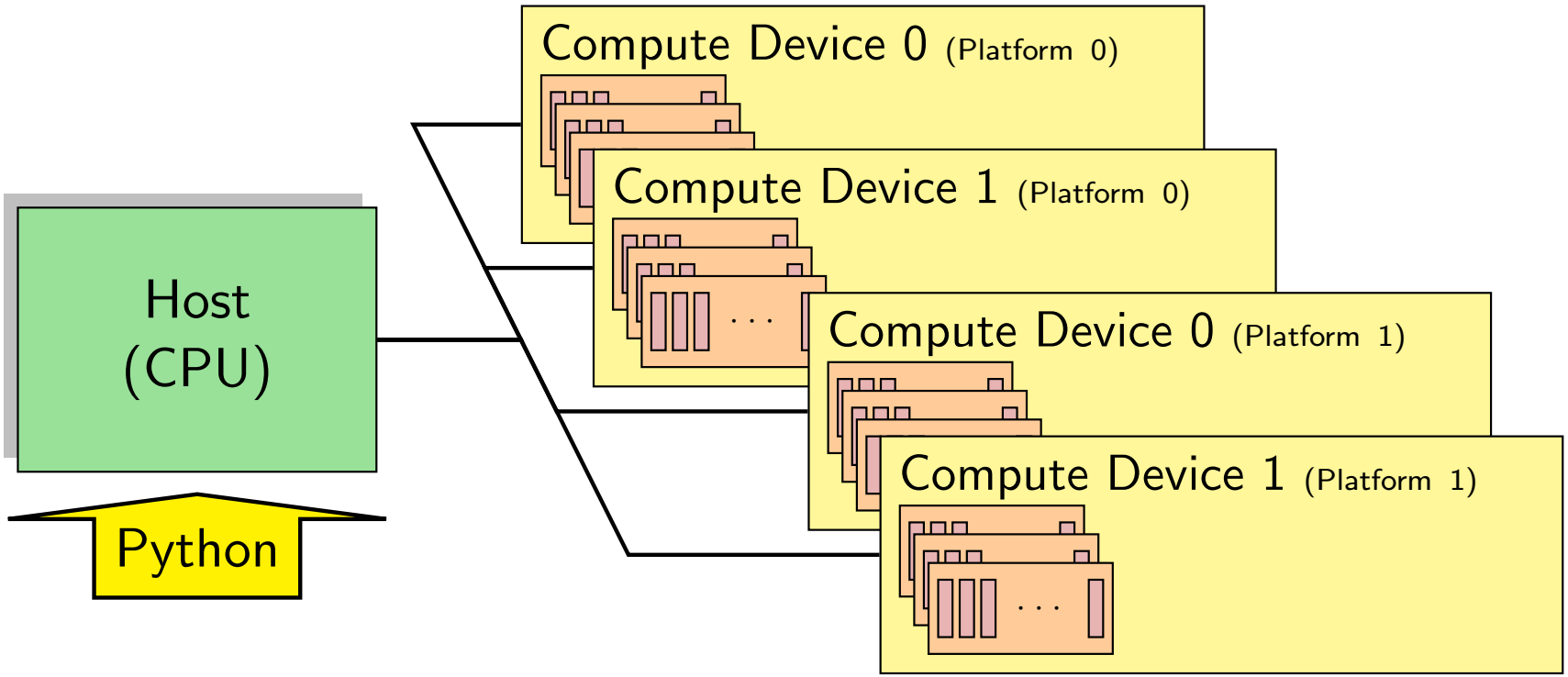
OpenCL: Computing as a Service



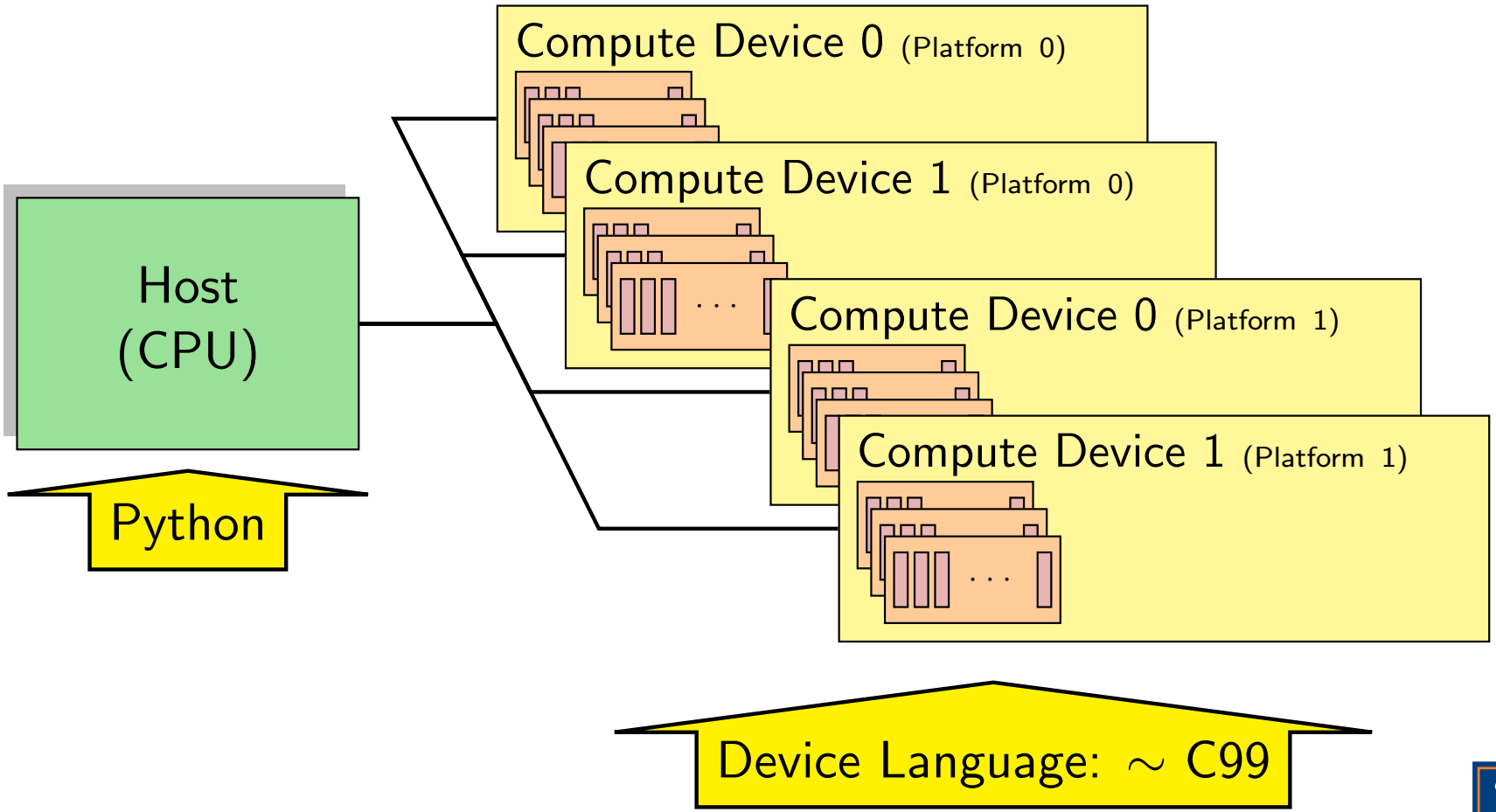
OpenCL: Computing as a Service



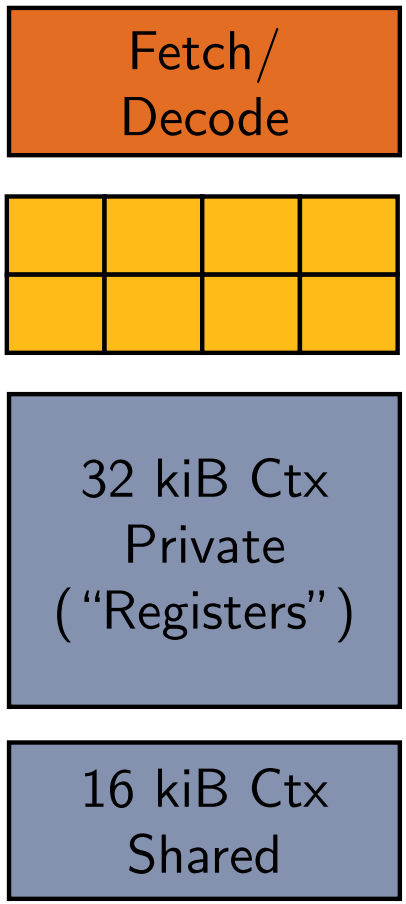
OpenCL: Computing as a Service



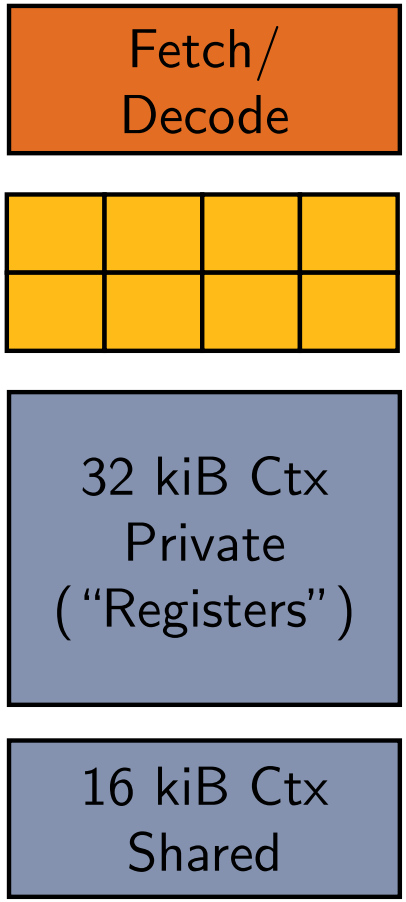
OpenCL: Computing as a Service

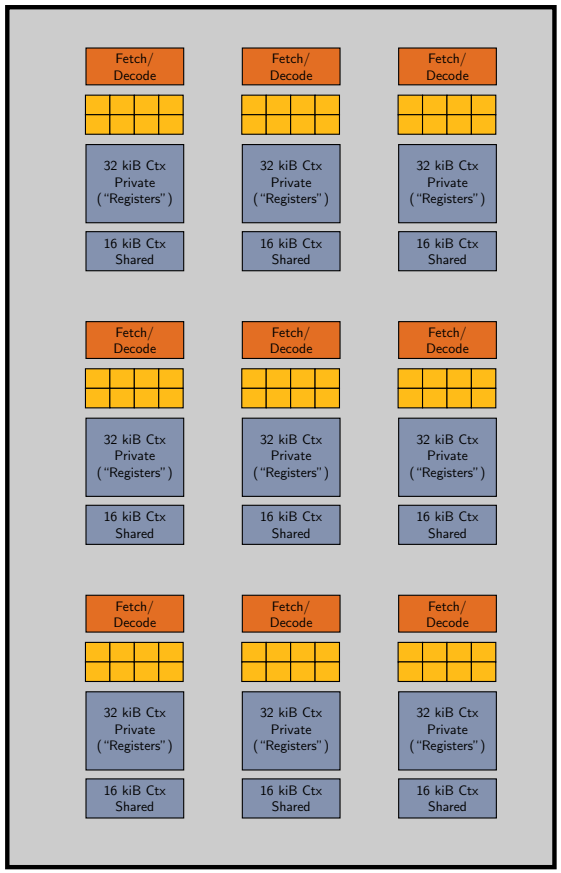


Connection: Hardware $\leftrightarrow$ Programming Model

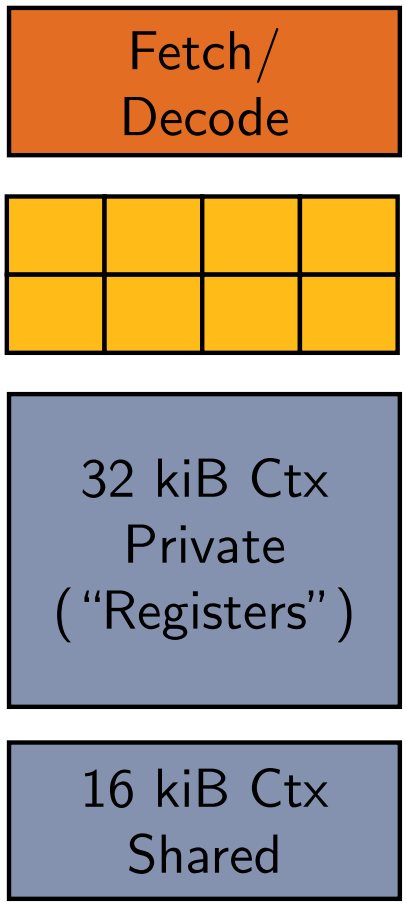


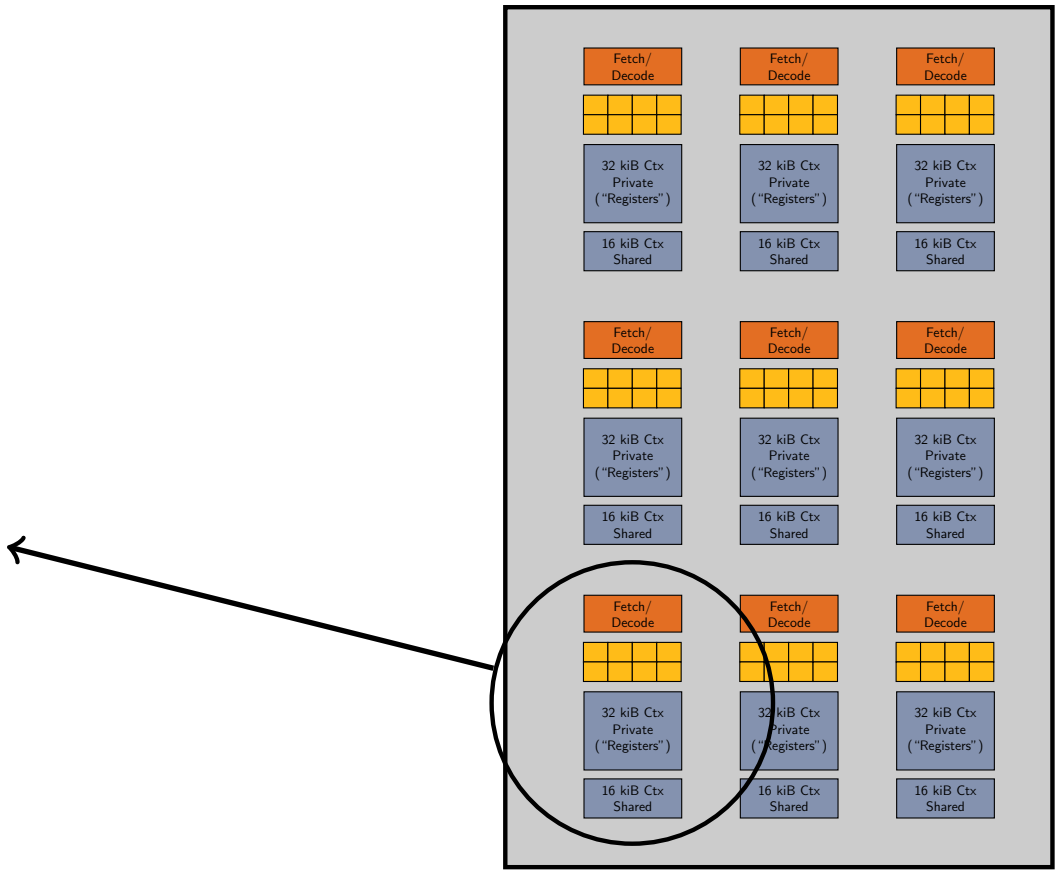
Connection: Hardware $\leftrightarrow$ Programming Model



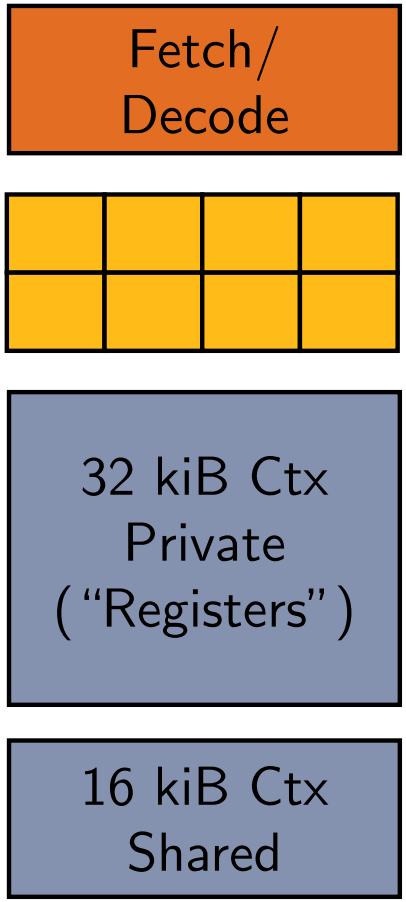


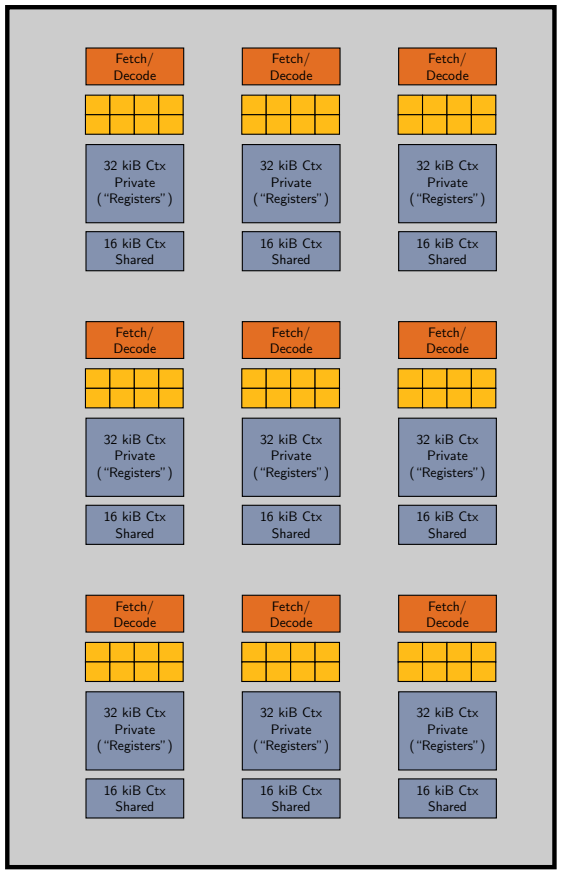
Connection: Hardware ↔ Programming Model



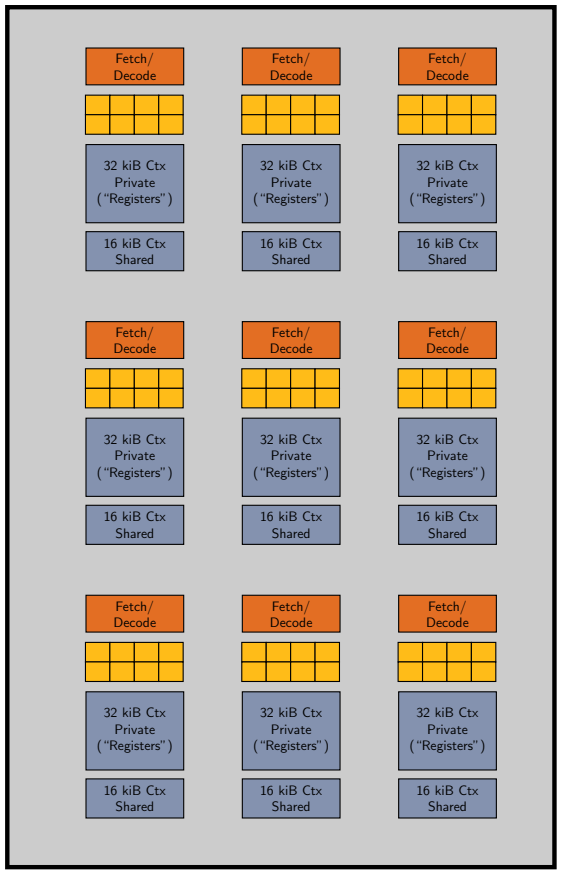


Connection: Hardware $\leftrightarrow$ Programming Model





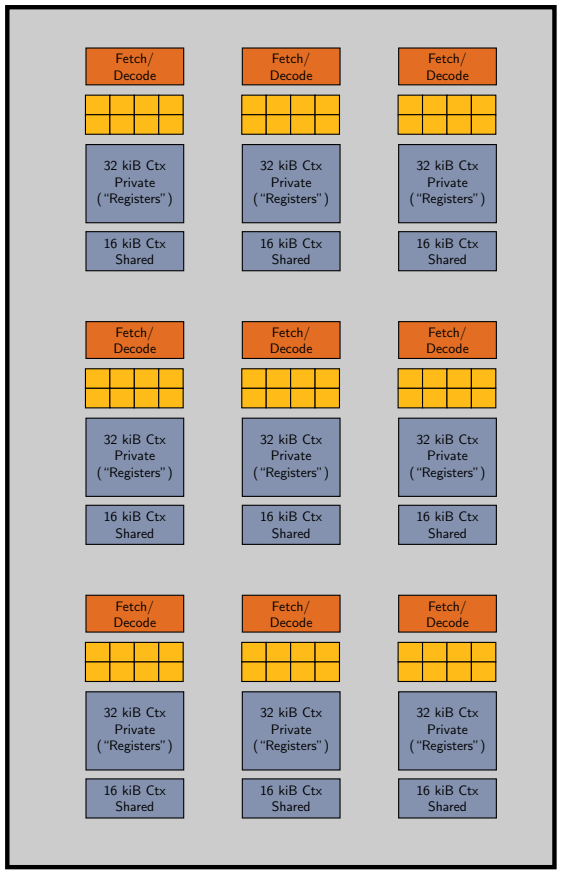
Connection: Hardware $\leftrightarrow$ Programming Model



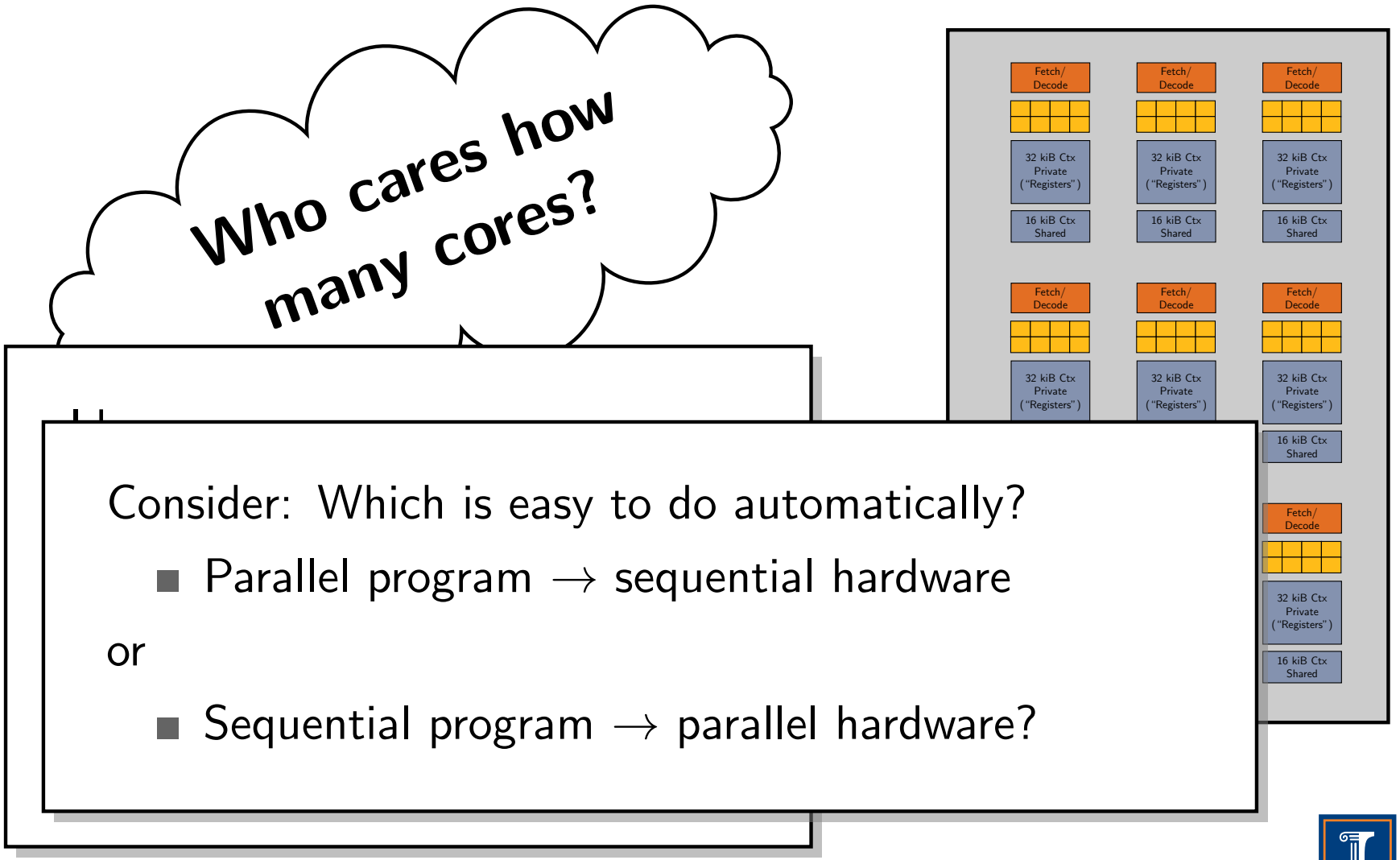
Connection: Hardware ↔ Programming Model

Idea:

- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core



Connection: Hardware $\leftrightarrow$ Programming Model



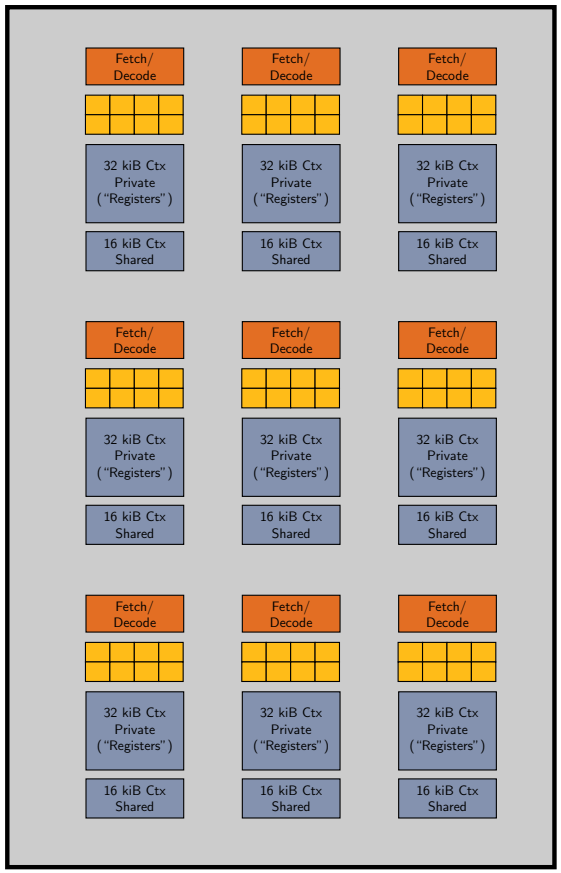
Consider: Which is easy to do automatically?

- Parallel program $\rightarrow$ sequential hardware

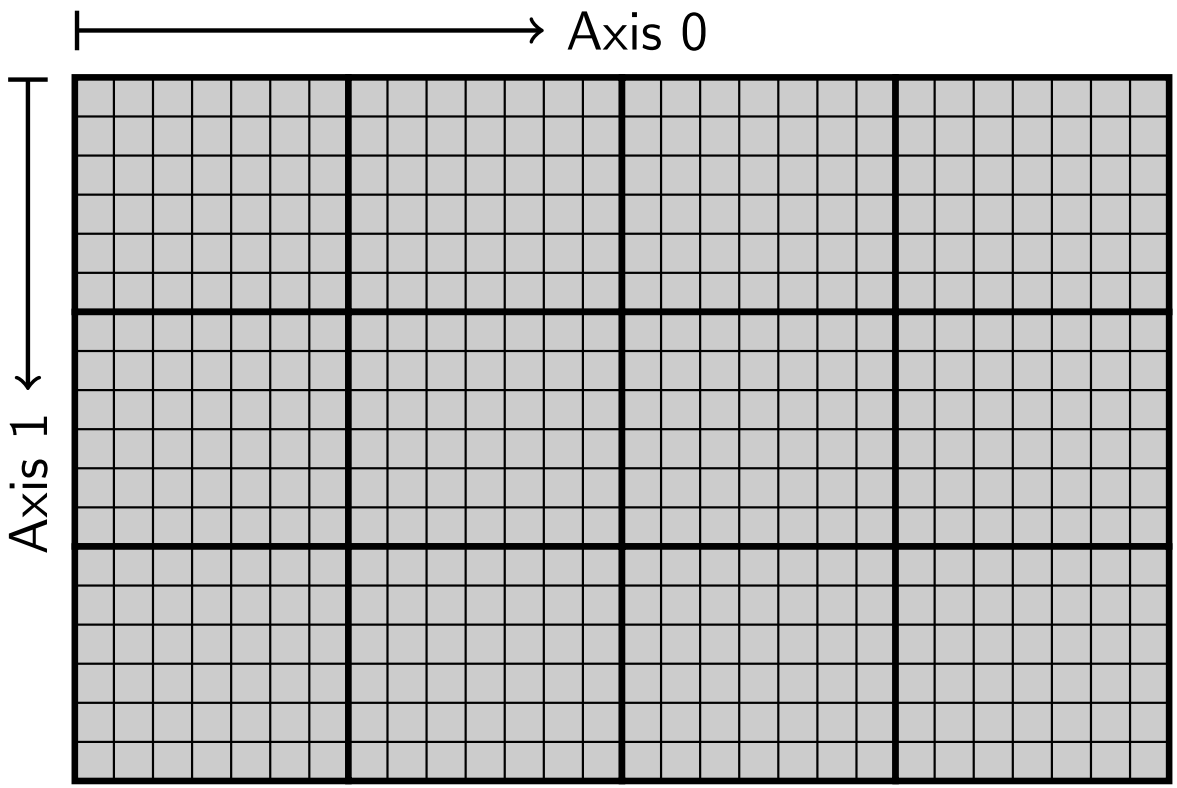
or

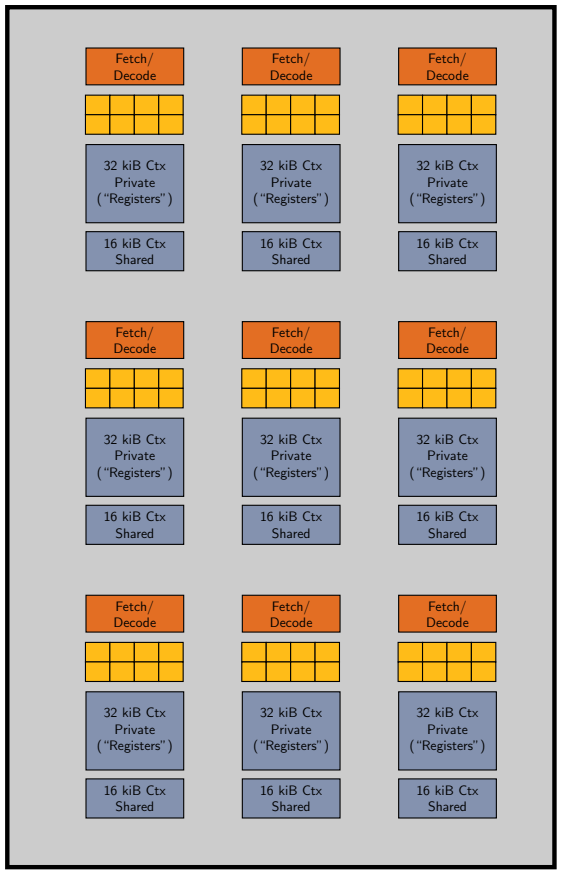
- Sequential program $\rightarrow$ parallel hardware?

Connection: Hardware $\leftrightarrow$ Programming Model

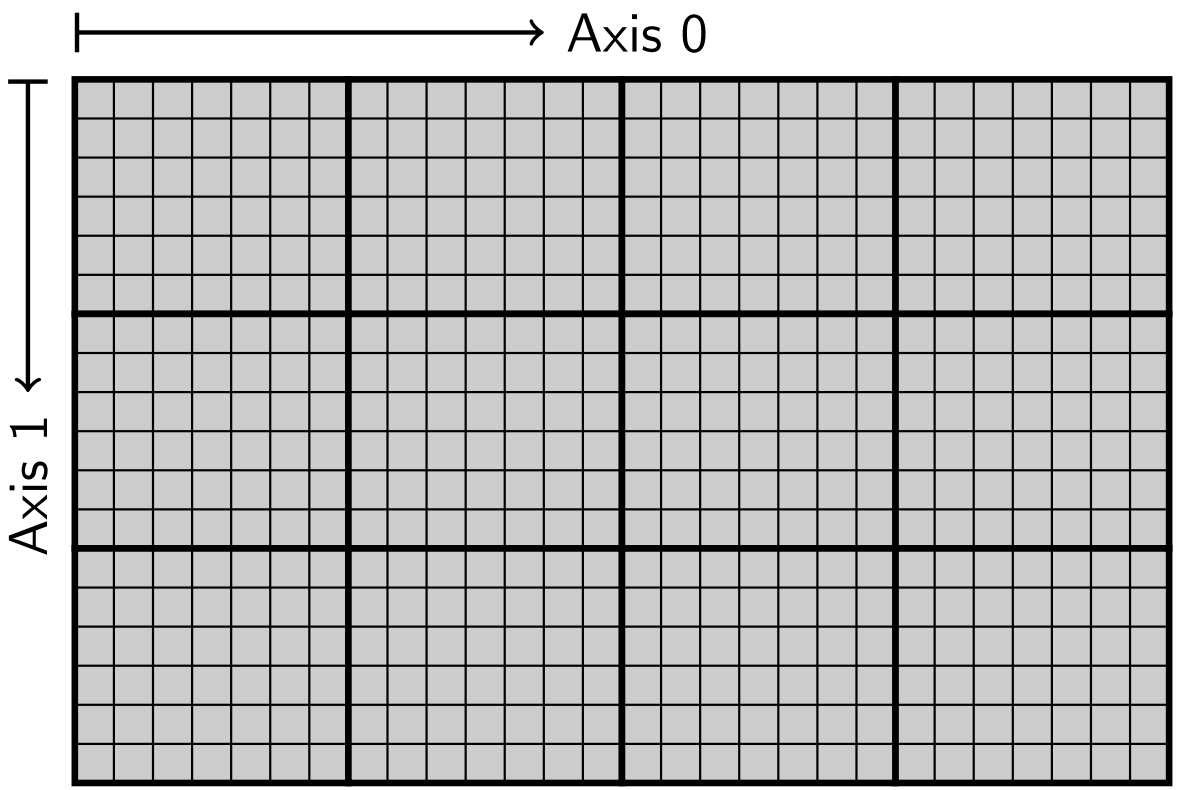


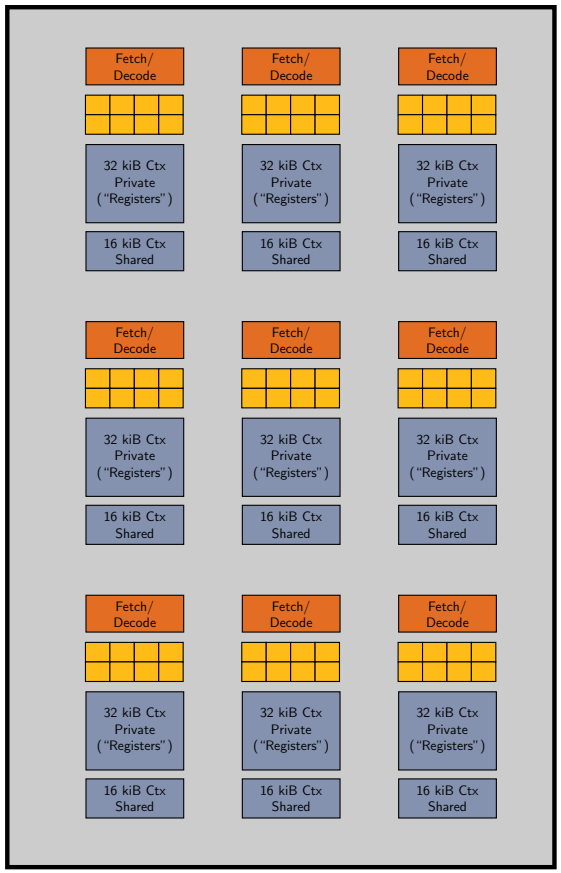
Connection: Hardware $\leftrightarrow$ Programming Model





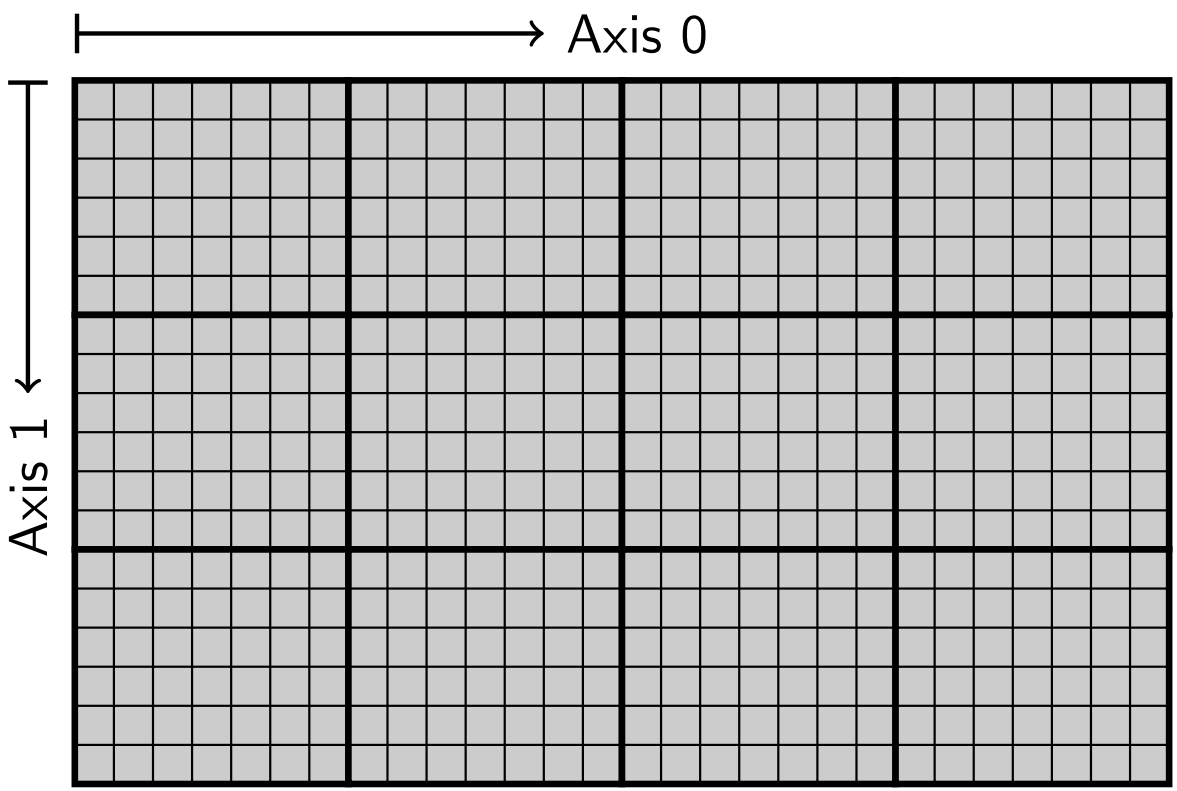
Connection: Hardware ↔ Programming Model



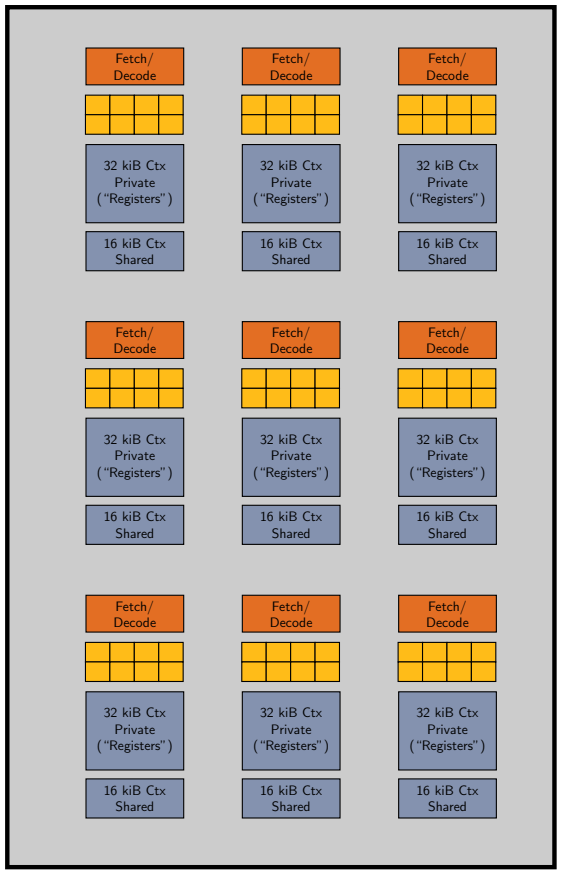


Hardware

Connection: Hardware $\leftrightarrow$ Programming Model

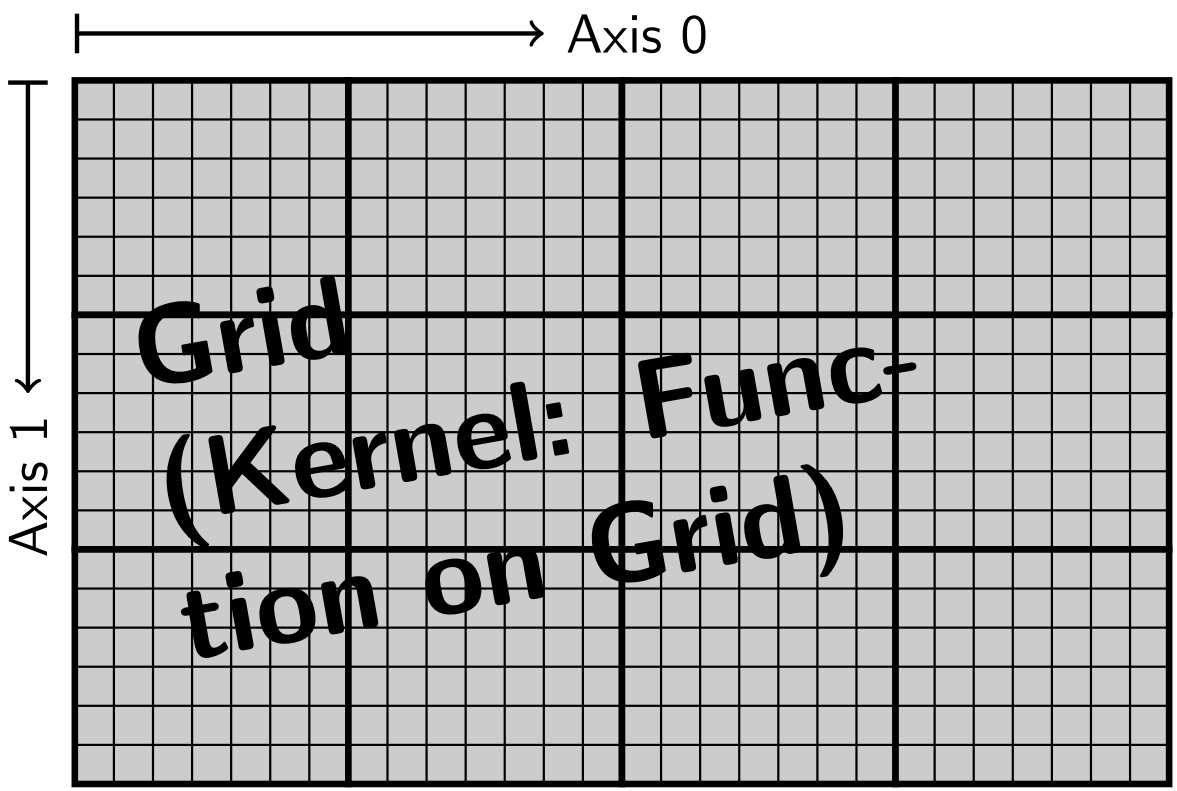


Software representation

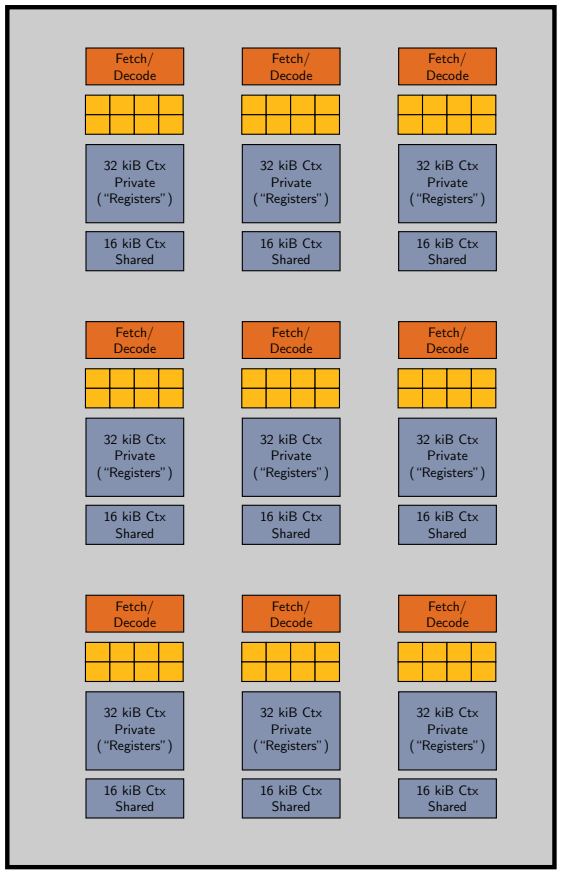


Hardware

Connection: Hardware ↔ Programming Model

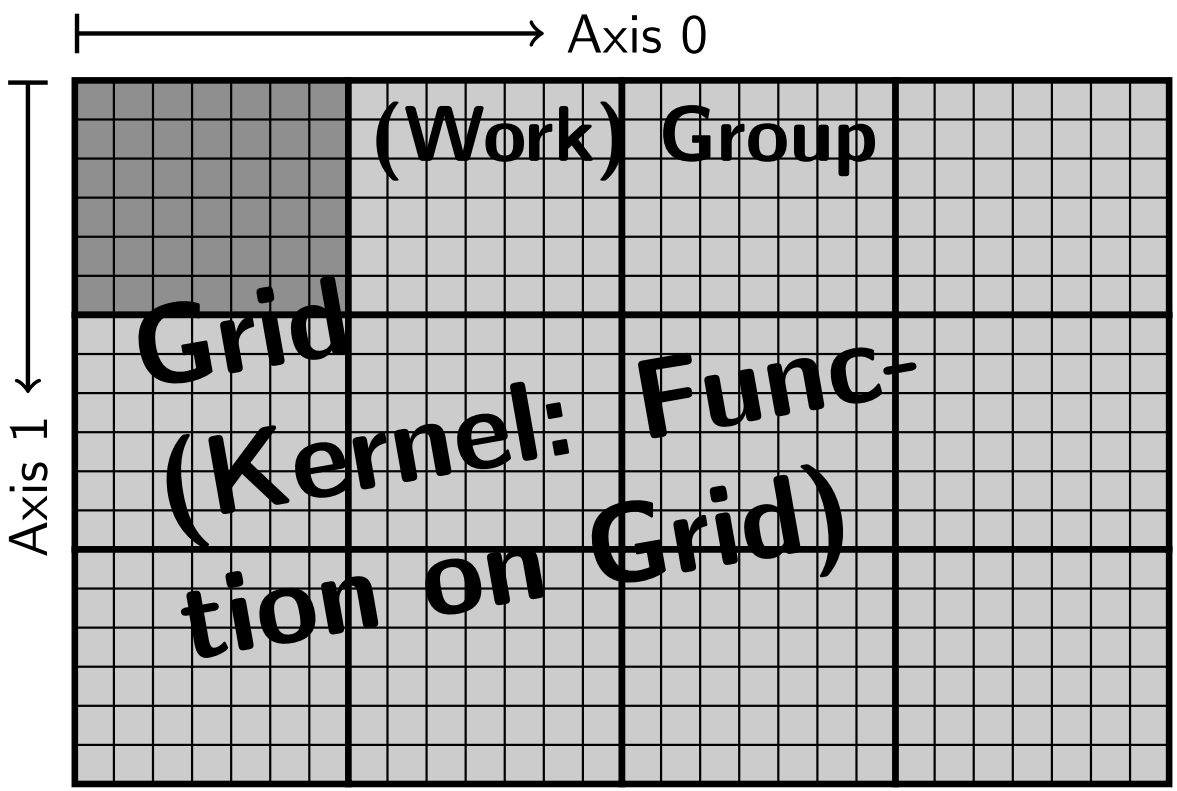


Software representation

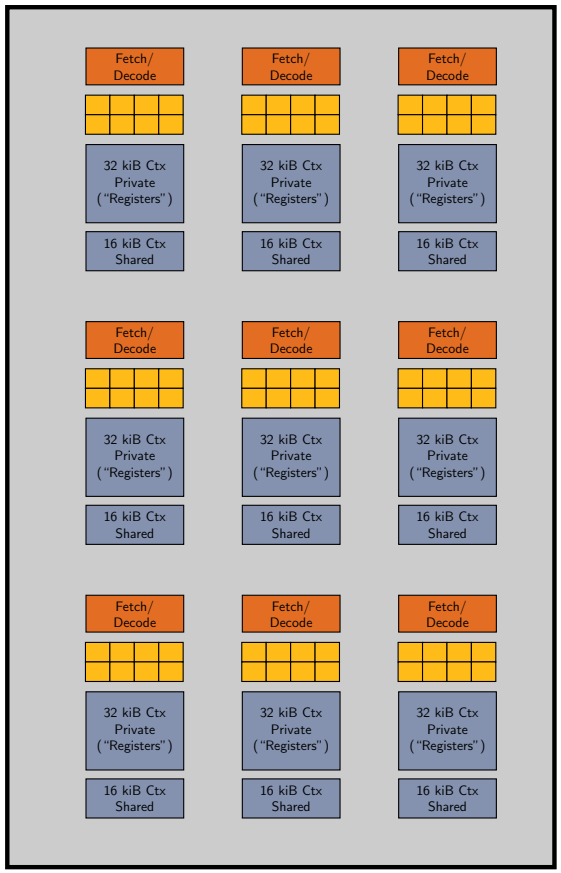


Hardware

Connection: Hardware ↔ Programming Model

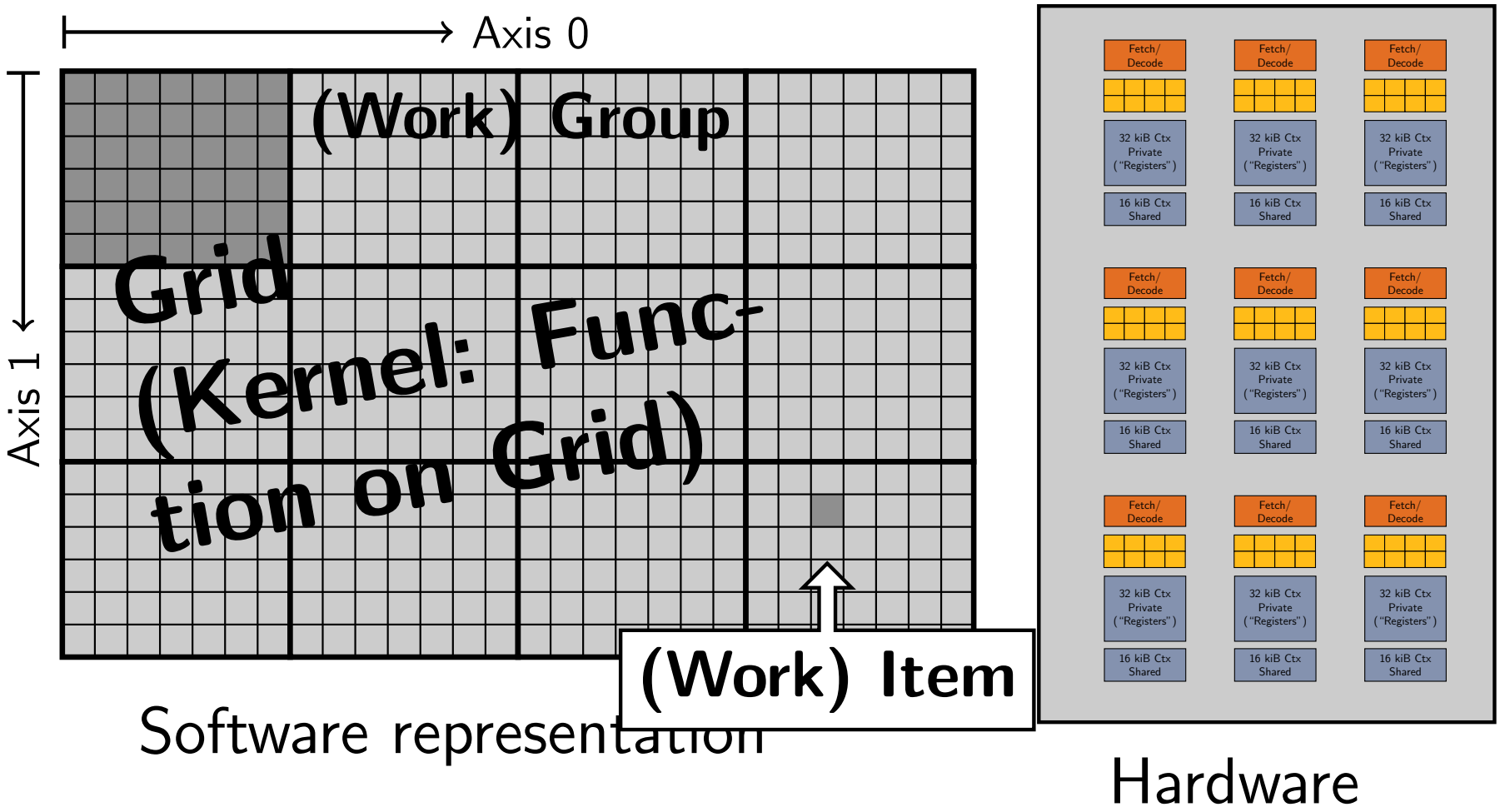


Software representation

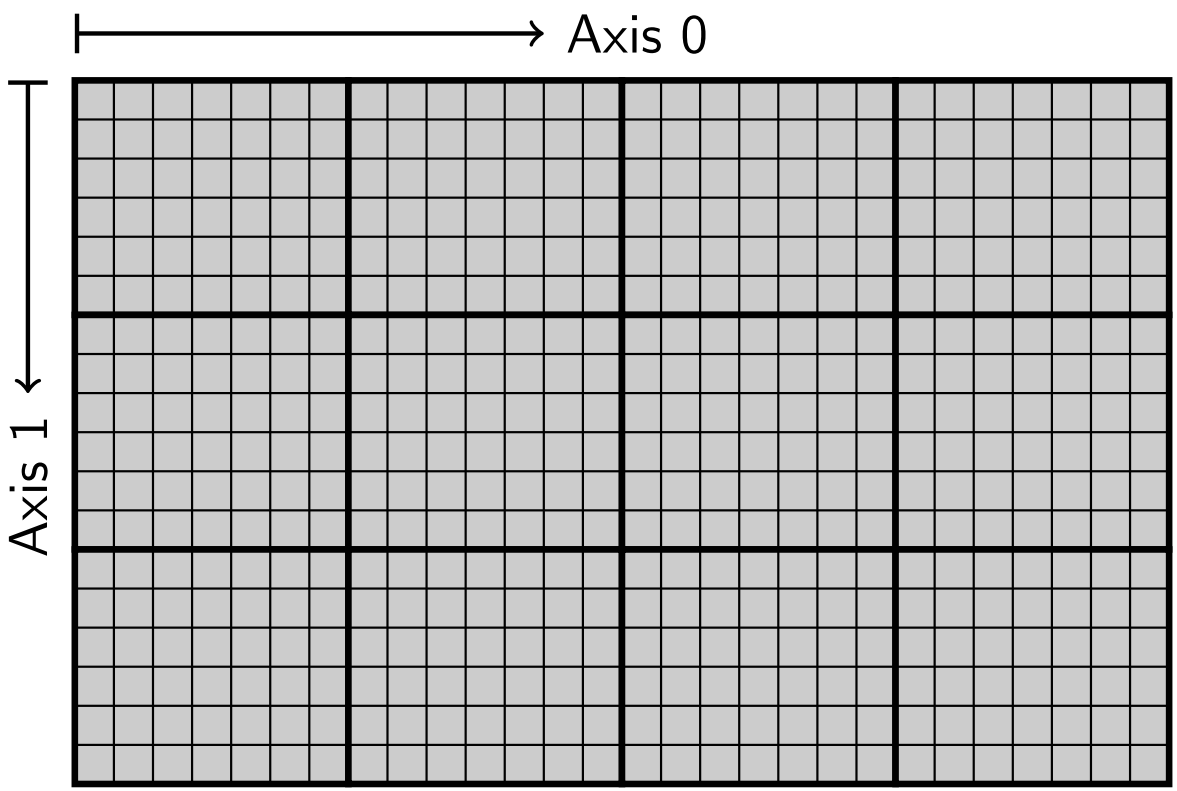


Hardware

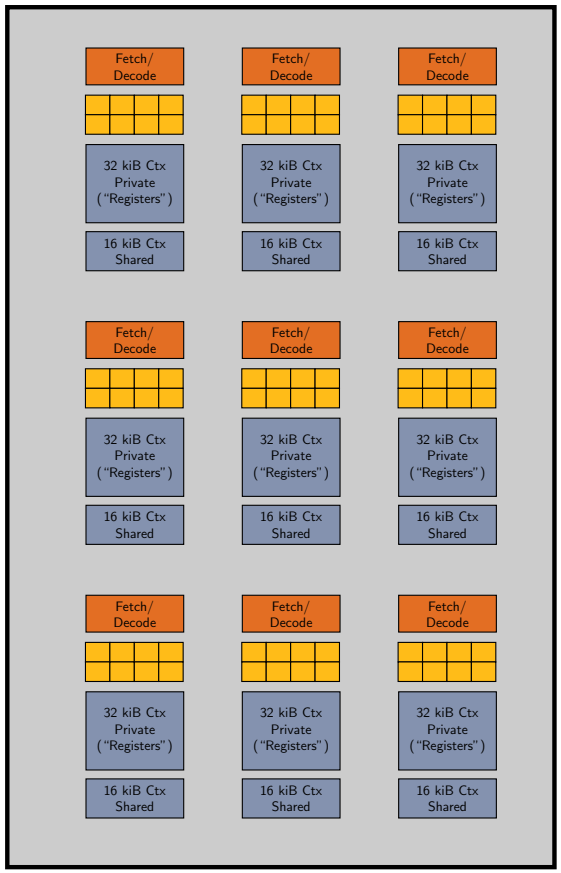
Connection: Hardware ↔ Programming Model



Connection: Hardware $\leftrightarrow$ Programming Model

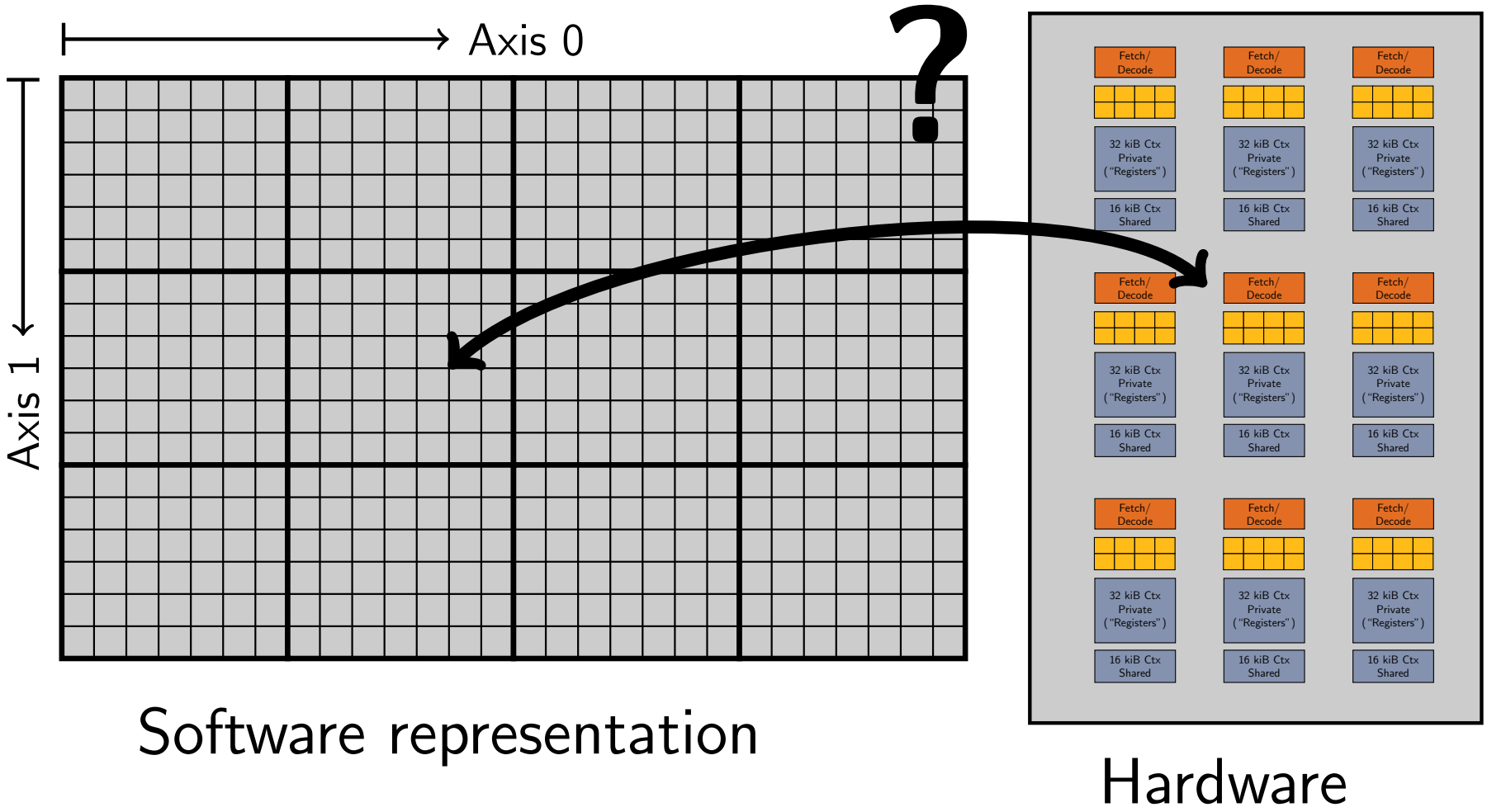


Software representation

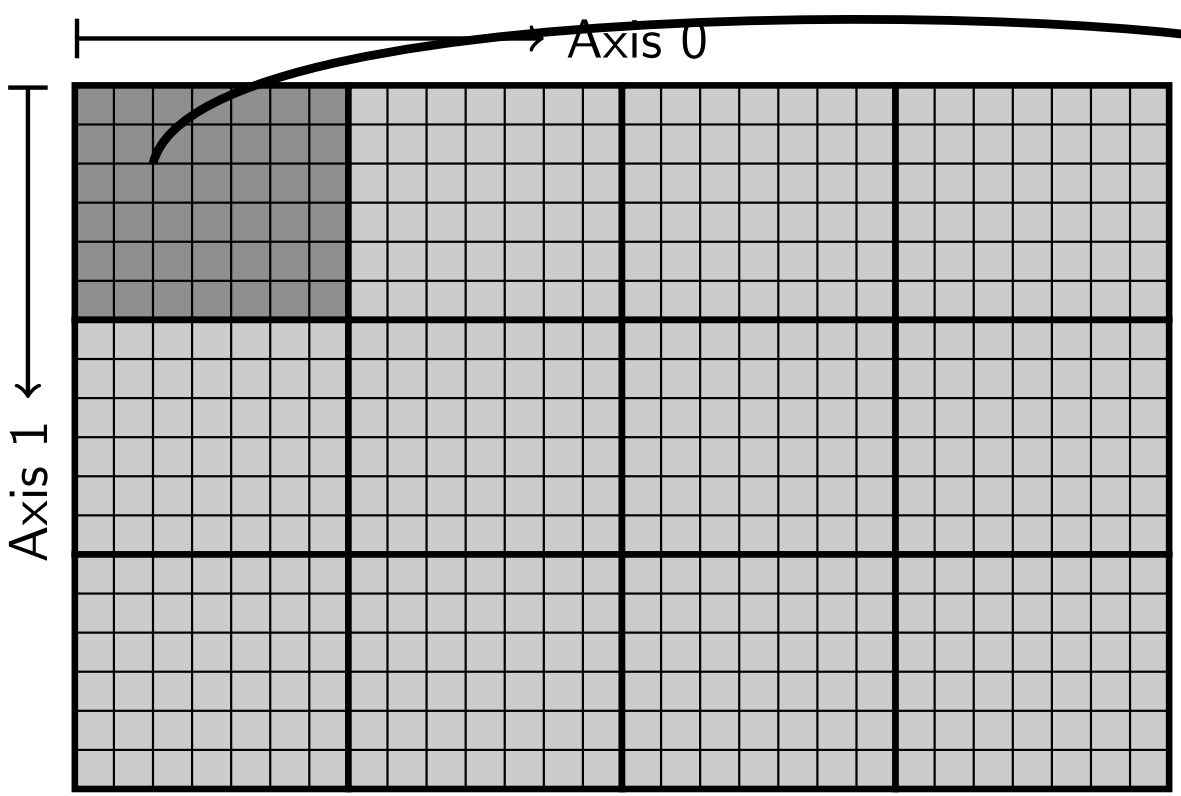


Hardware

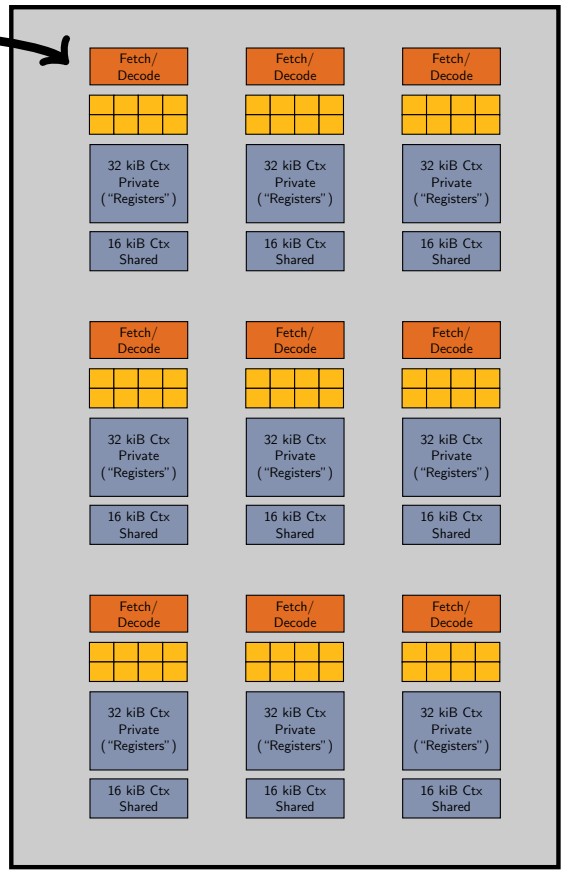
Connection: Hardware ↔ Programming Model



Connection: Hardware $\leftrightarrow$ Programming Model

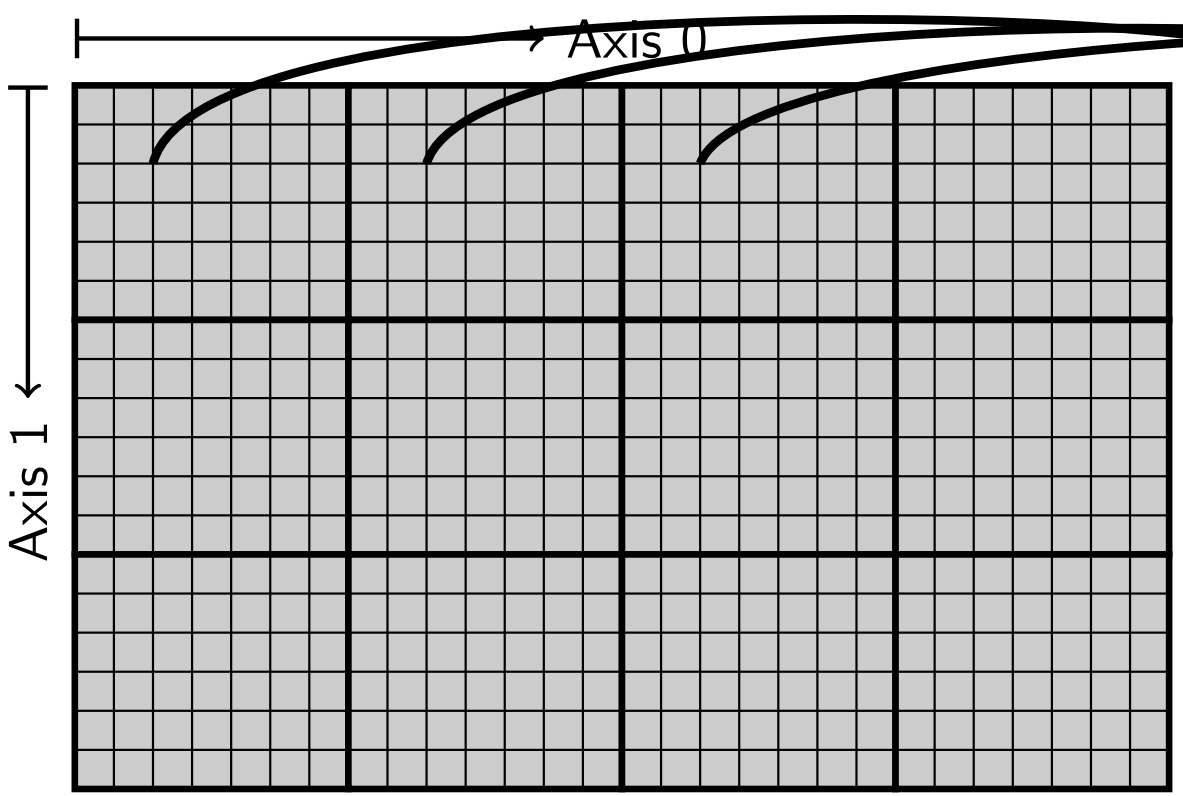


Software representation

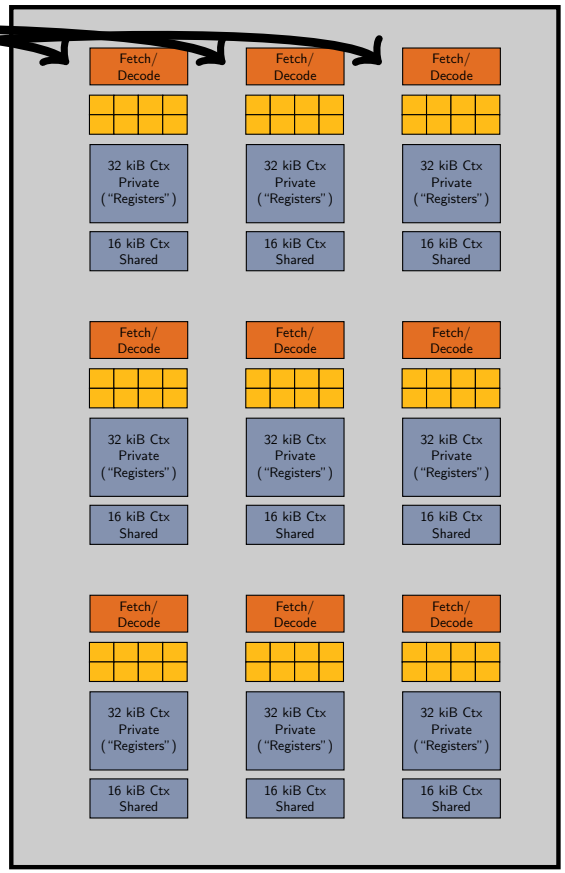


Hardware

Connection: Hardware ↔ Programming Model

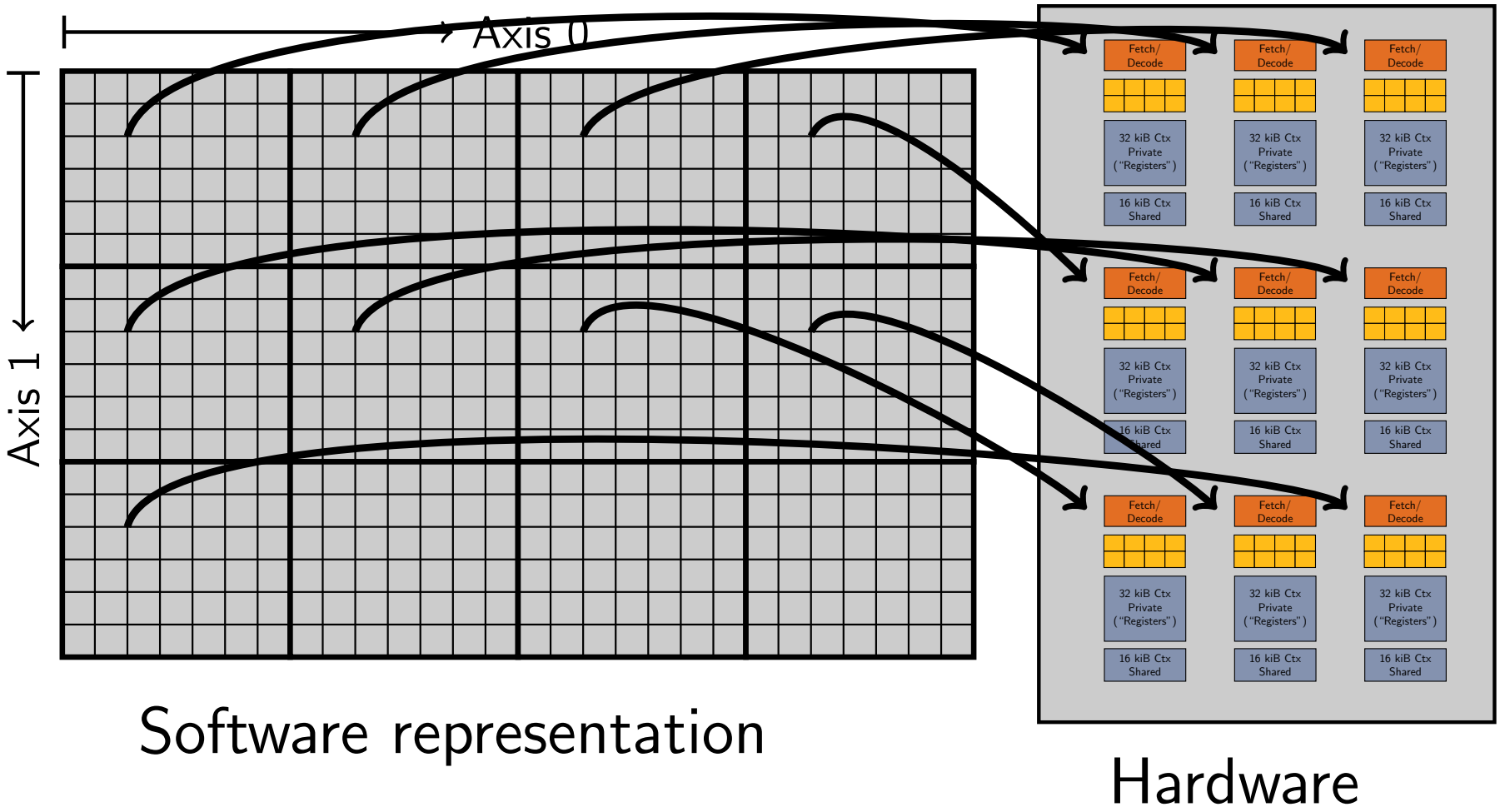


Software representation



Hardware

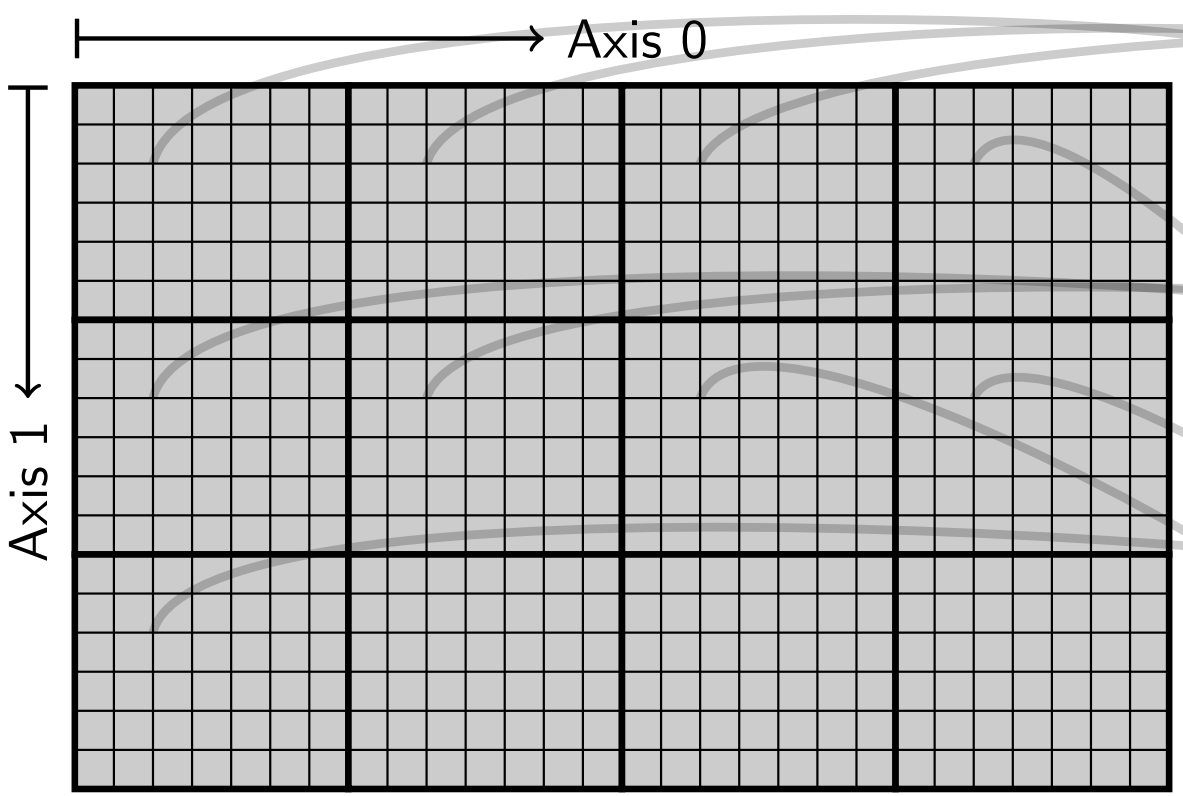
Connection: Hardware ↔ Programming Model



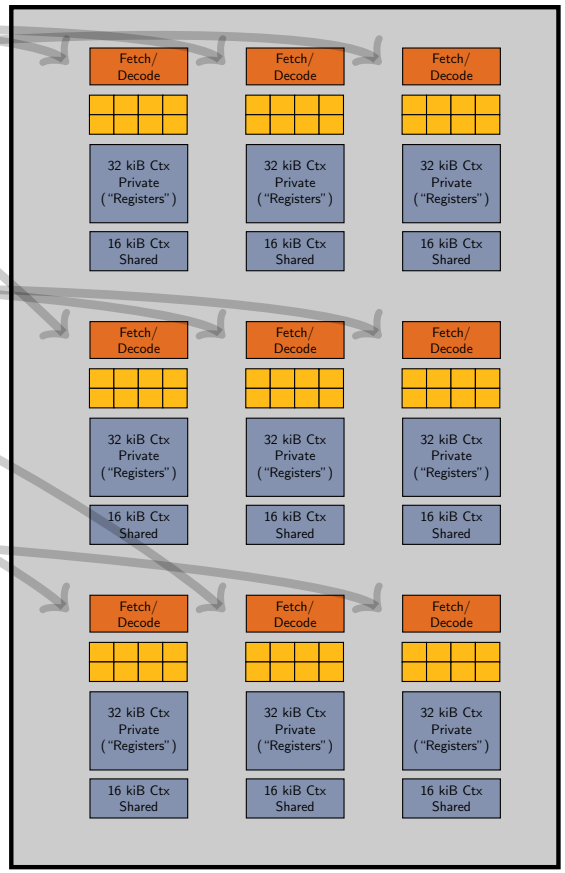
Software representation

Hardware

Connection: Hardware $\leftrightarrow$ Programming Model

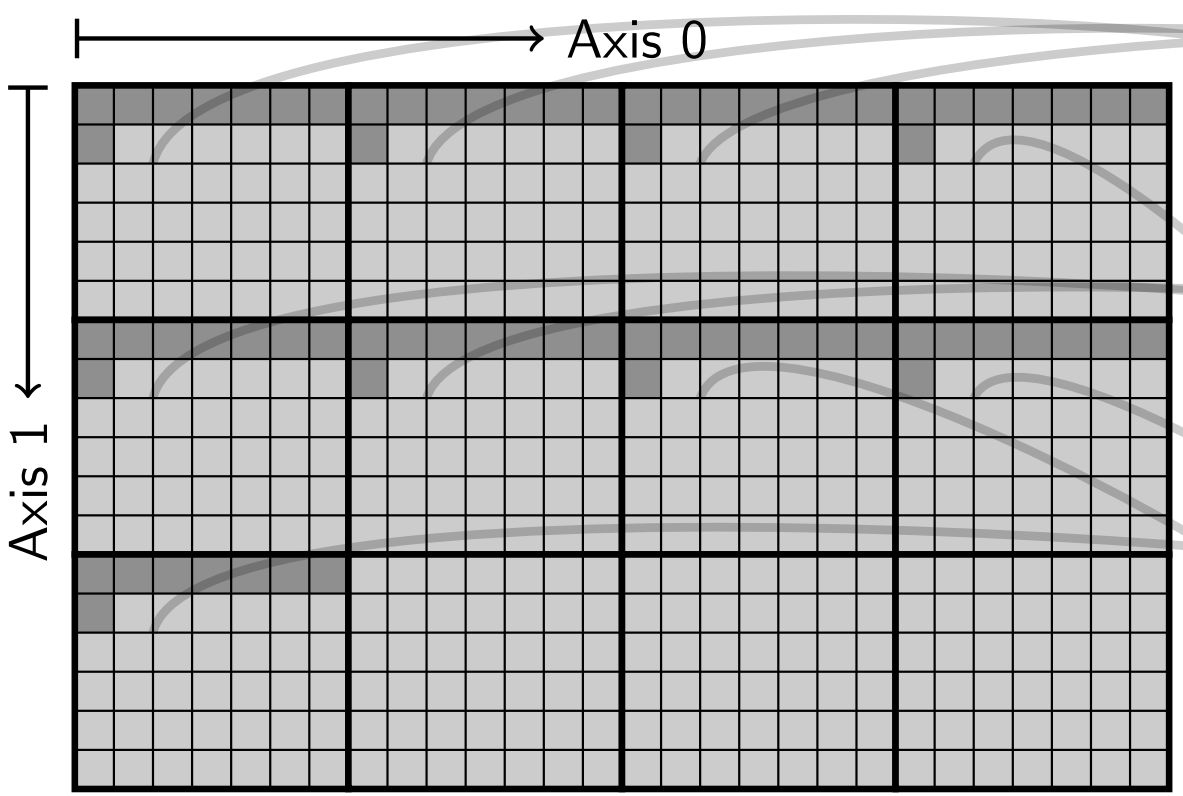


Software representation

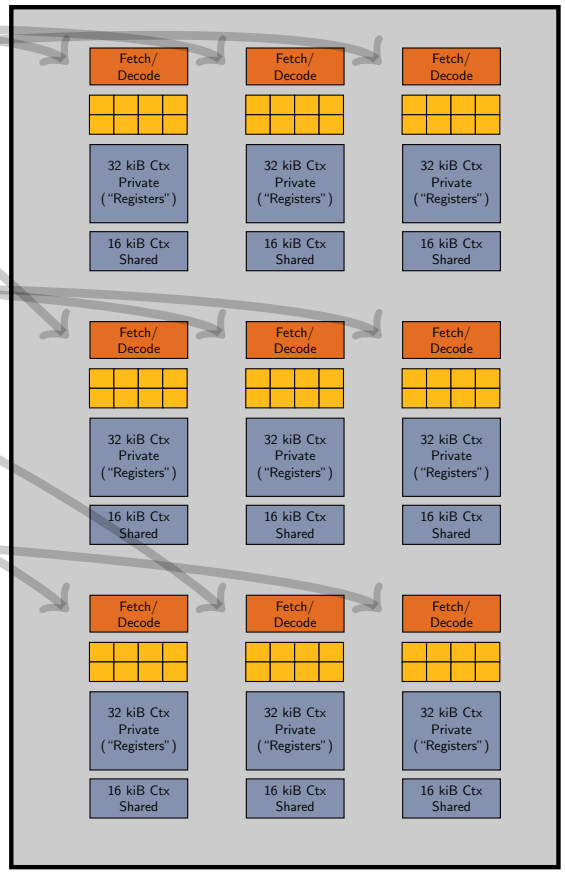


Hardware

Connection: Hardware $\leftrightarrow$ Programming Model

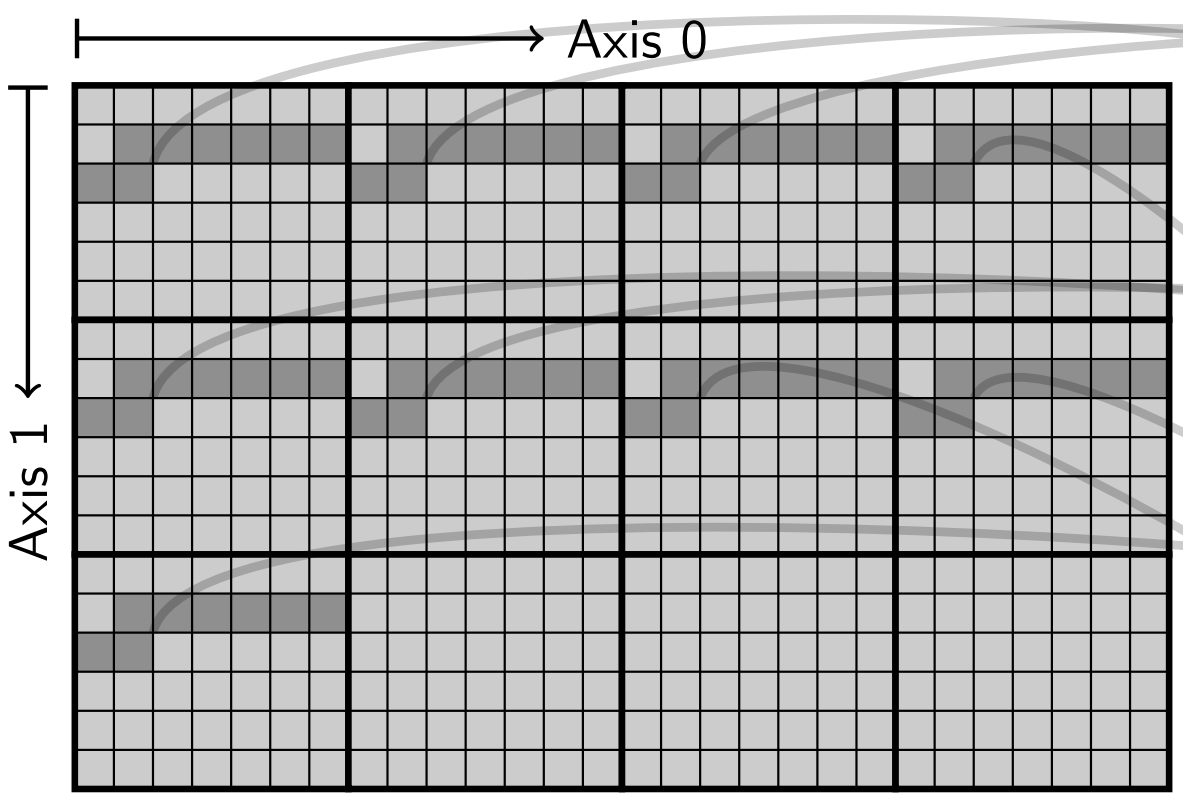


Software representation

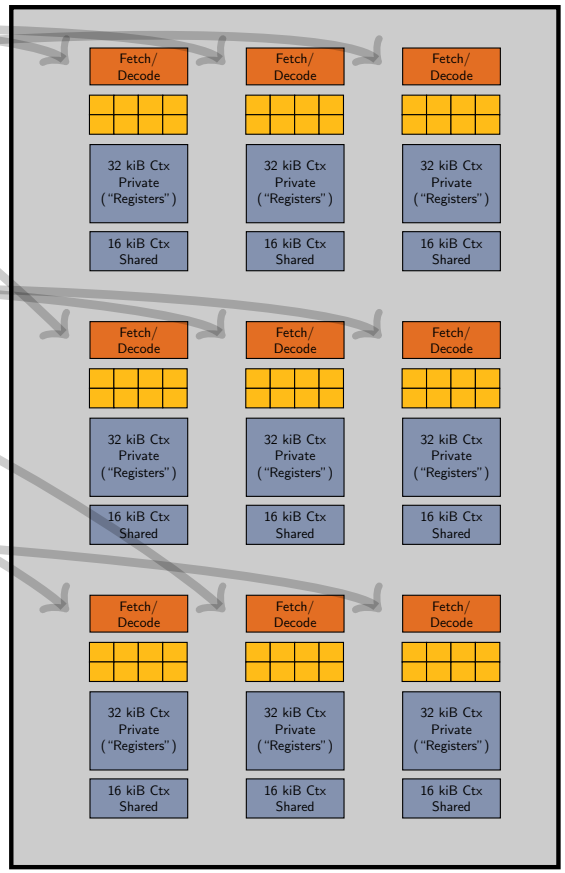


Hardware

Connection: Hardware $\leftrightarrow$ Programming Model

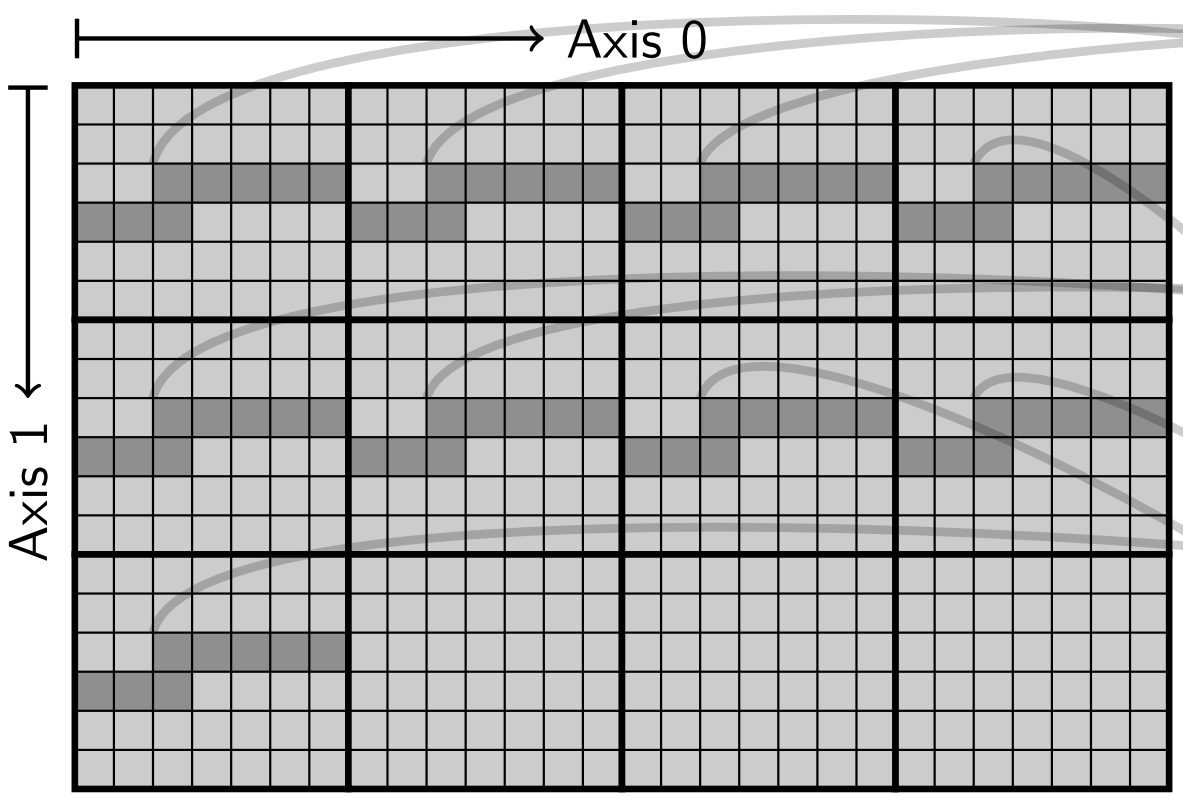


Software representation

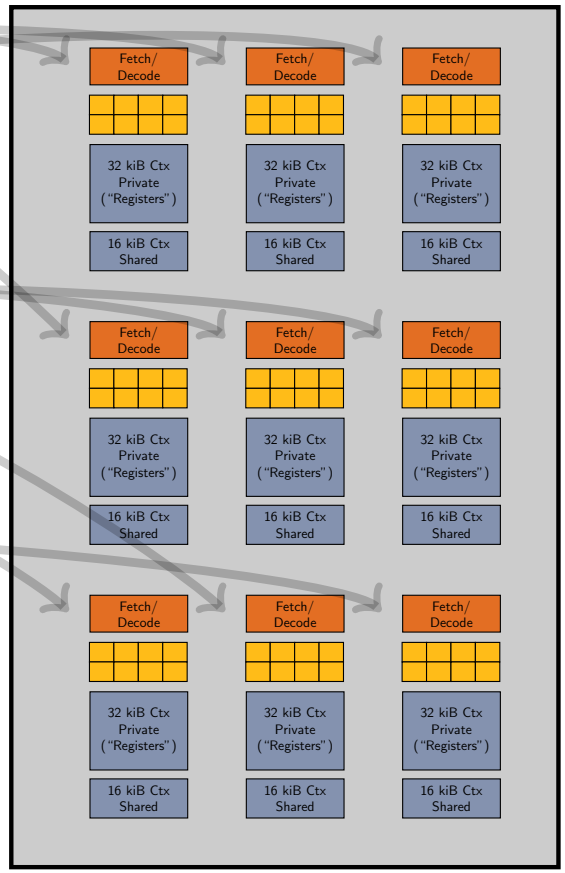


Hardware

Connection: Hardware ↔ Programming Model

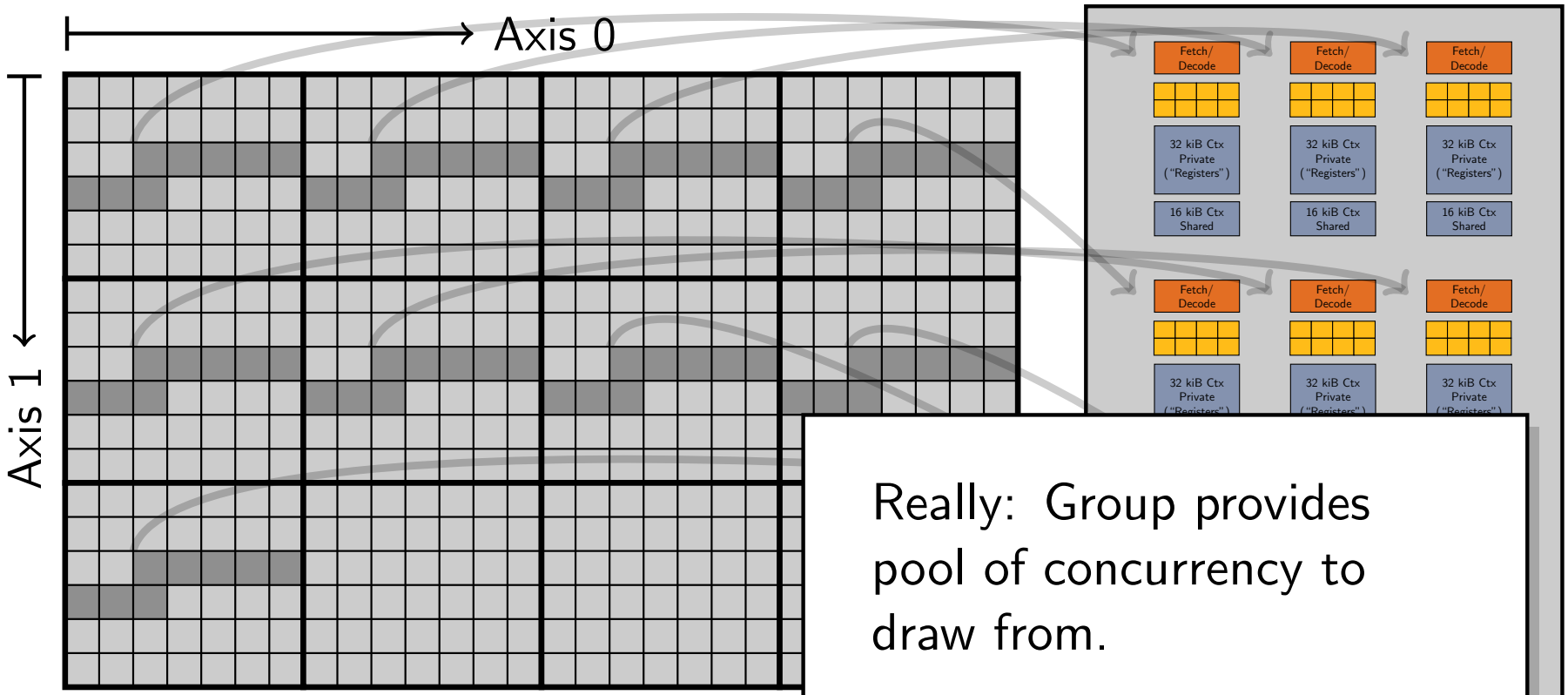


Software representation



Hardware

Connection: Hardware ↔ Programming Model

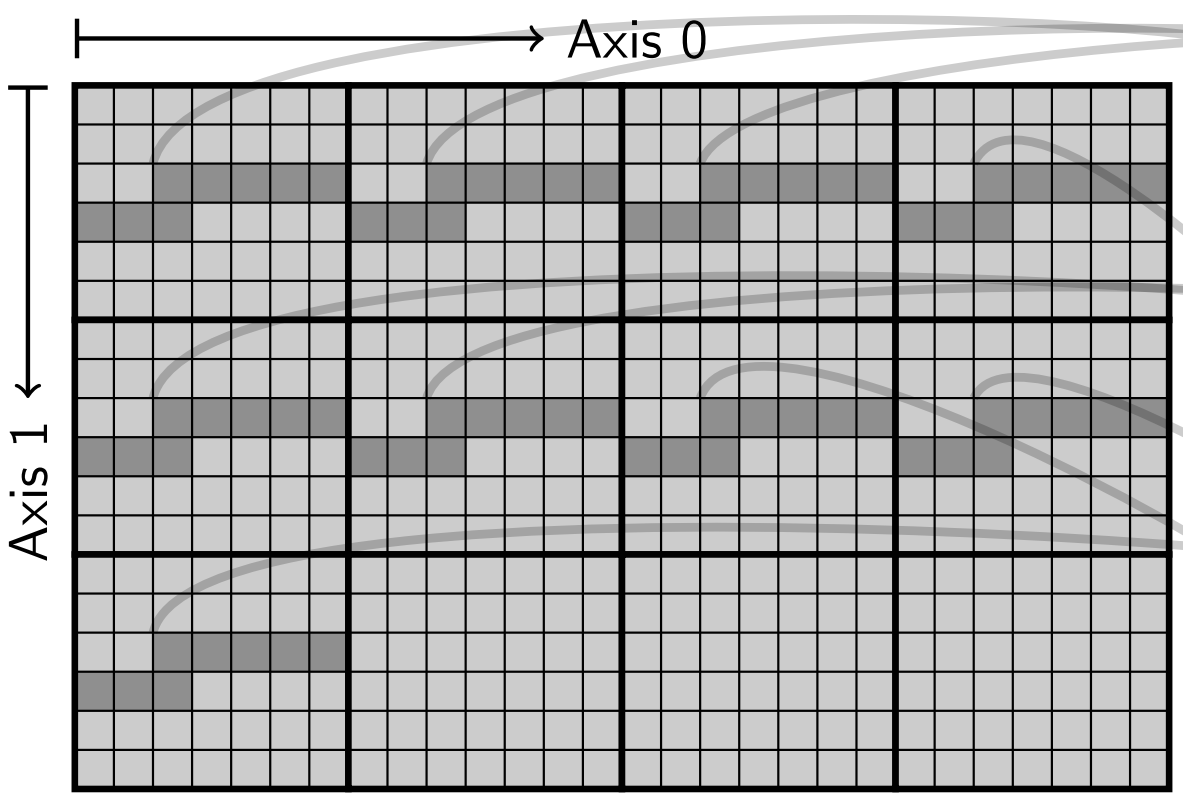


Software representation

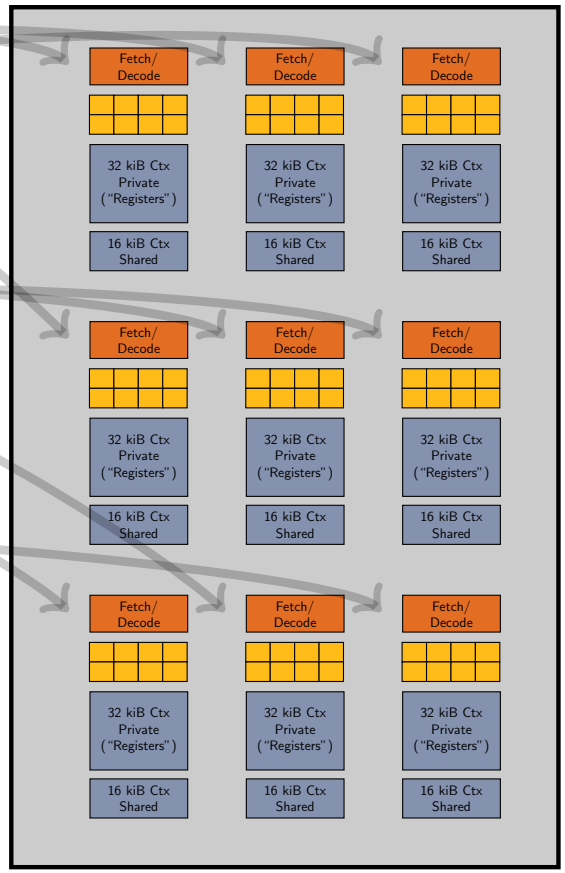
Really: Group provides pool of concurrency to draw from.

X,Y,Z order *within* group matters. (Not *among* groups, though.)

Connection: Hardware ↔ Programming Model

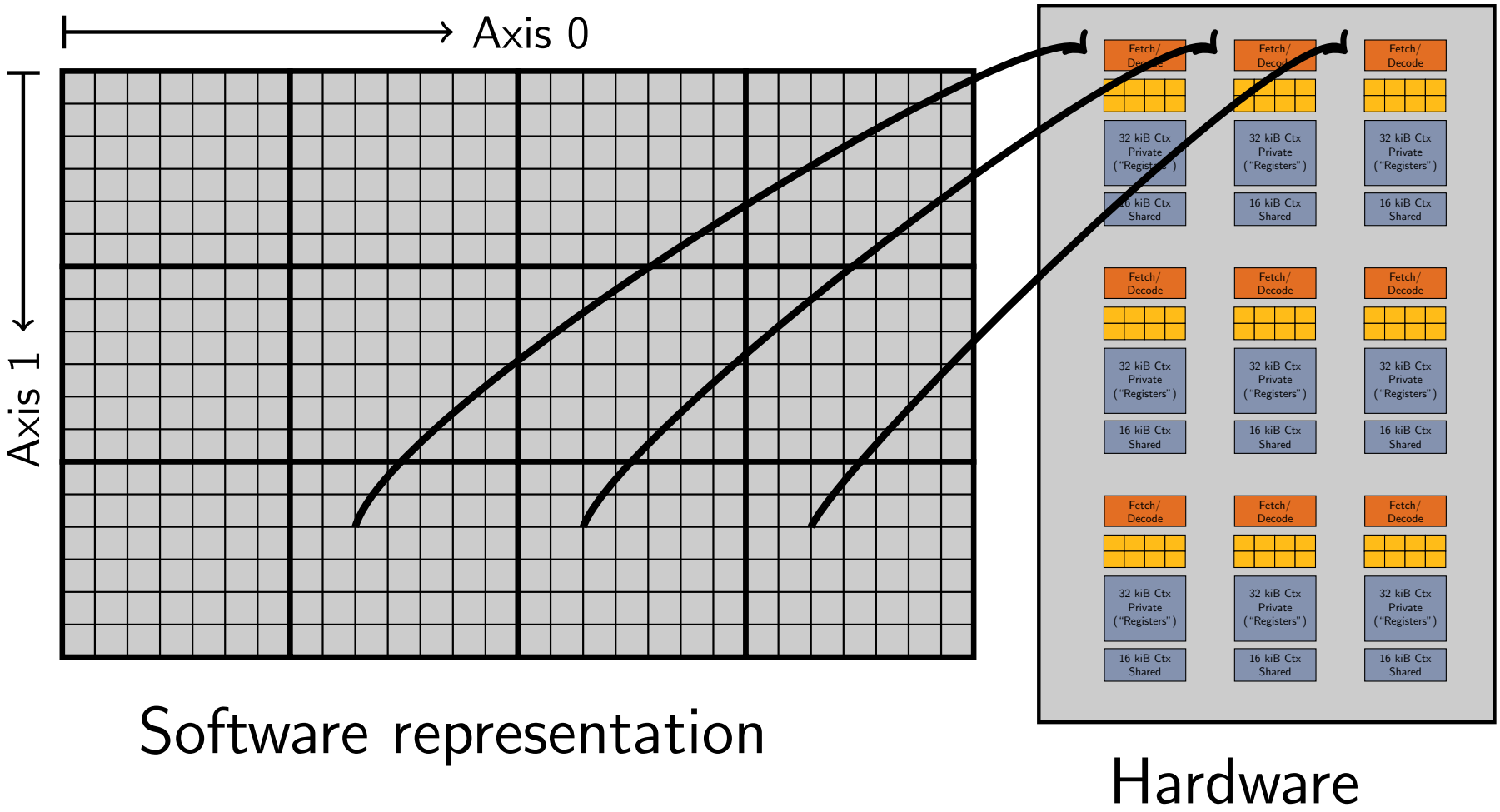


Software representation

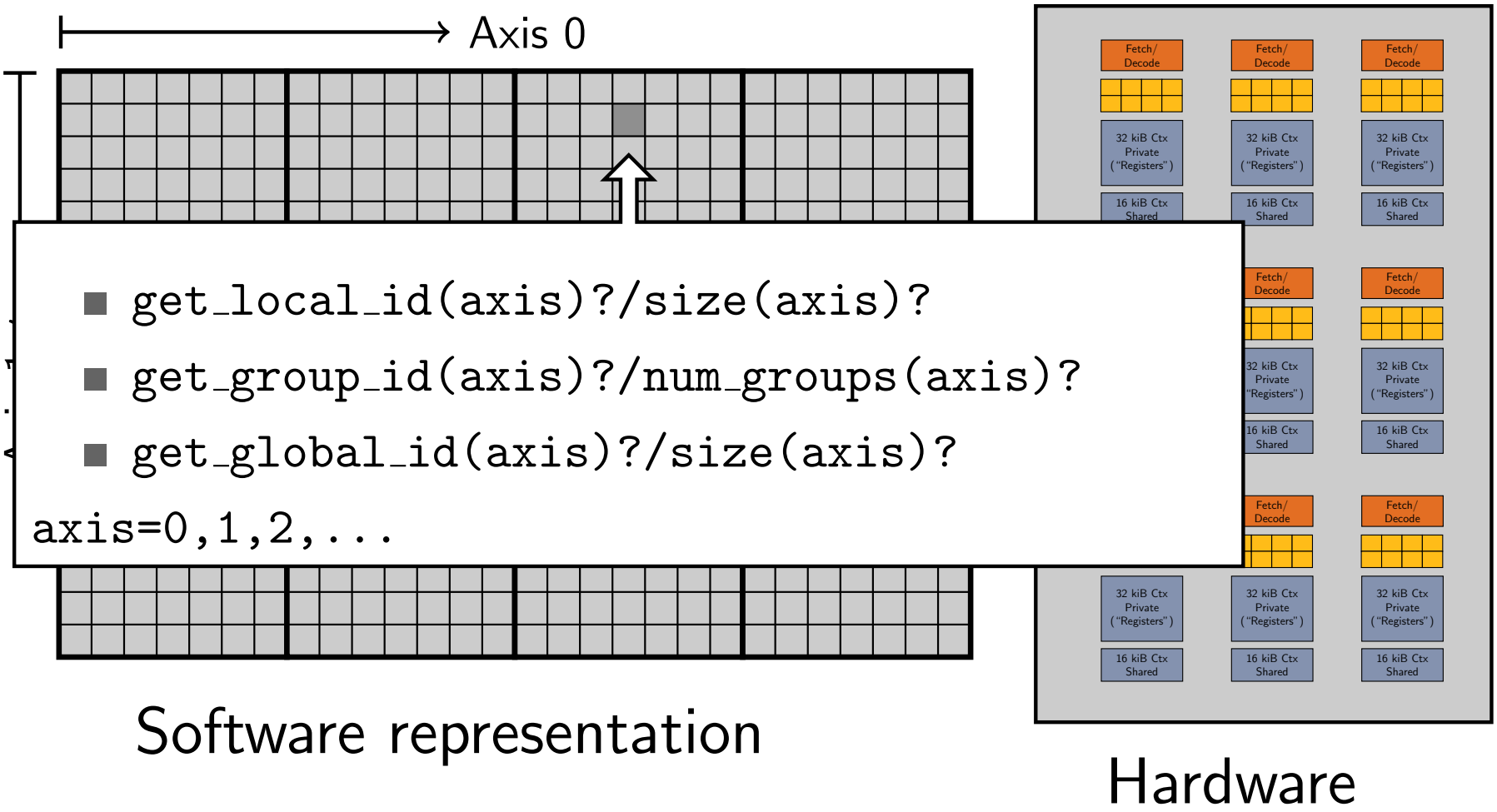


Hardware

Connection: Hardware $\leftrightarrow$ Programming Model

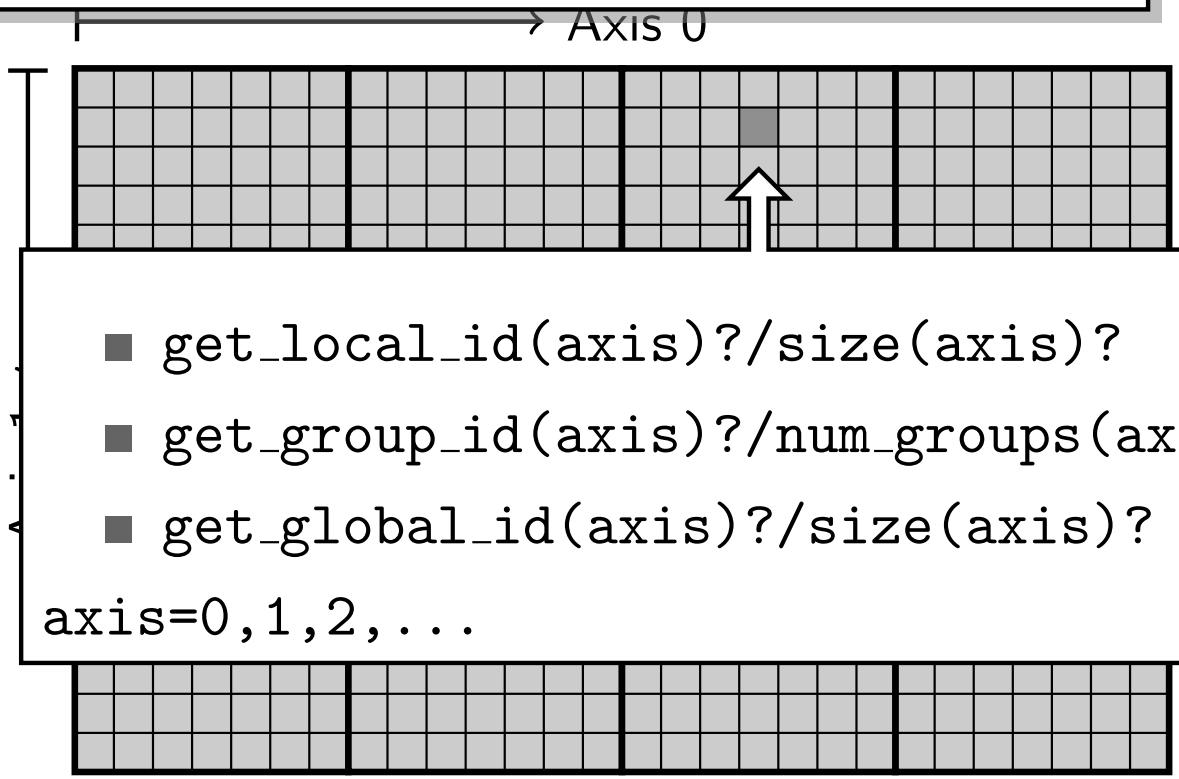


Connection: Hardware ↔ Programming Model



Co g Model

Grids can be 1,2,3-dimensional.



- `get_local_id(axis)?/size(axis)?`
- `get_group_id(axis)?/num_groups(axis)?`
- `get_global_id(axis)?/size(axis)?`

`axis=0,1,2,...`

Software representation

Hardware

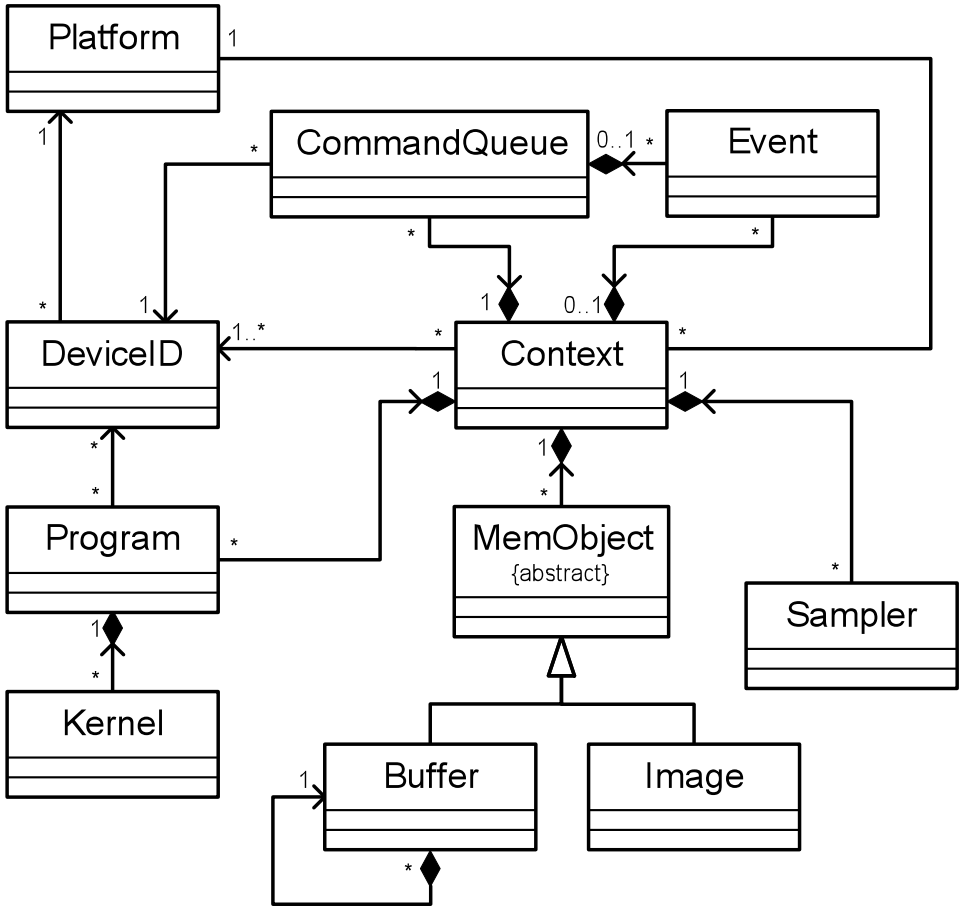
Outline

DEMO TIME

Outline

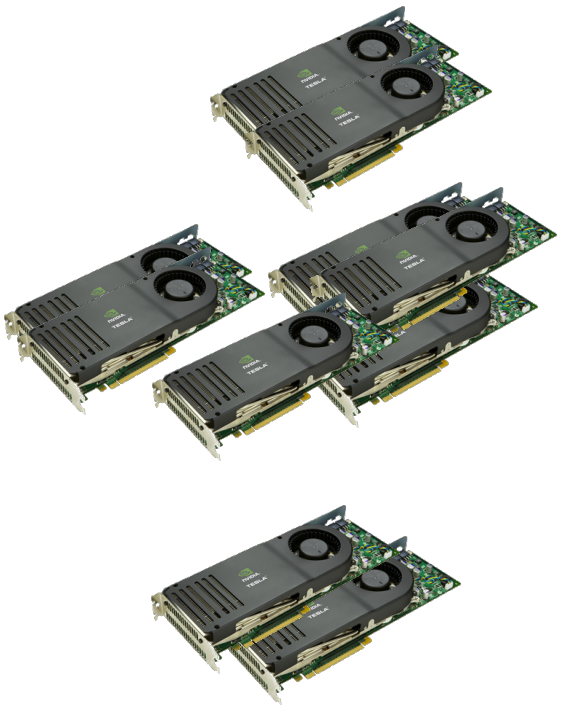
Outline

OpenCL Object Diagram



Credit: Khronos Group

CL "Platform"



- "Platform": a collection of devices, all from the same *vendor*.
- All devices in a platform use same CL driver/implementation.
- Multiple platforms can be used from one program $\rightarrow$ *ICD*.

`libOpenCL.so`: ICD loader

`/etc/OpenCL/vendors/`*`somename`*`.icd`: Plain text file with name of `.so` containing CL implementation.

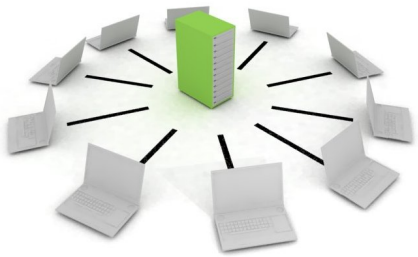
CL "Compute Device"

CL Compute Devices:

- CPUs, GPUs, accelerators, ...
  - Anything that fits the programming model.
- A processor die with an interface to off-chip memory
- Can get list of devices from platform.

Contexts

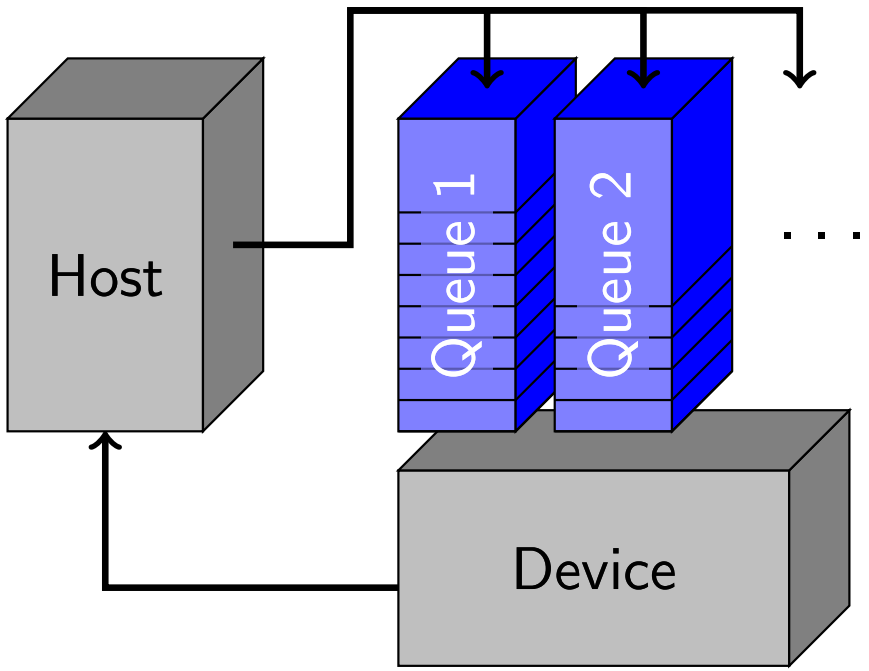
```
context = cl.Context(devices=None | [dev1, dev2], dev_type=None)
context = cl.create_some_context( interactive =True)
```



- Spans one or more Devices
- Create from device type or list of devices
  - See docs for `cl.Platform`, `cl.Device`
- `dev_type`: *`DEFAULT`*, `ALL`, `CPU`, `GPU`
- Needed to...
  - ...allocate Memory Objects
  - ...create and build Programs
  - ...host Command Queues
  - ...execute Grids

OpenCL: Command Queues

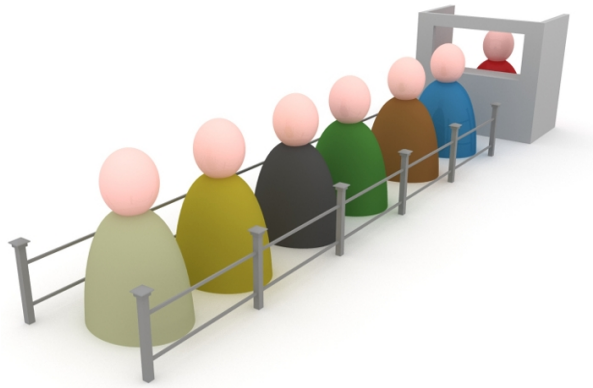
- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for drained queue
- Profiling



Command Queues and Events

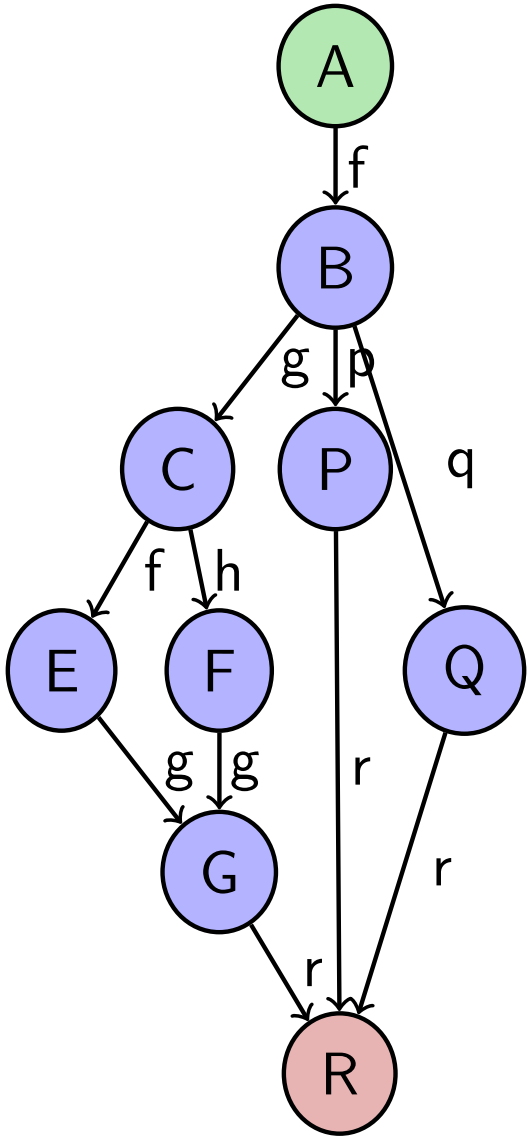
```
queue = cl.CommandQueue(context, device=None,
    properties=None | [(prop, value),...])
```

- Attached to single device
- cl.command_queue_properties...
  - `OUT_OF_ORDER_EXEC_MODE_ENABLE`:
    Do not force sequential execution
  - `PROFILING_ENABLE`:
    Gather timing info



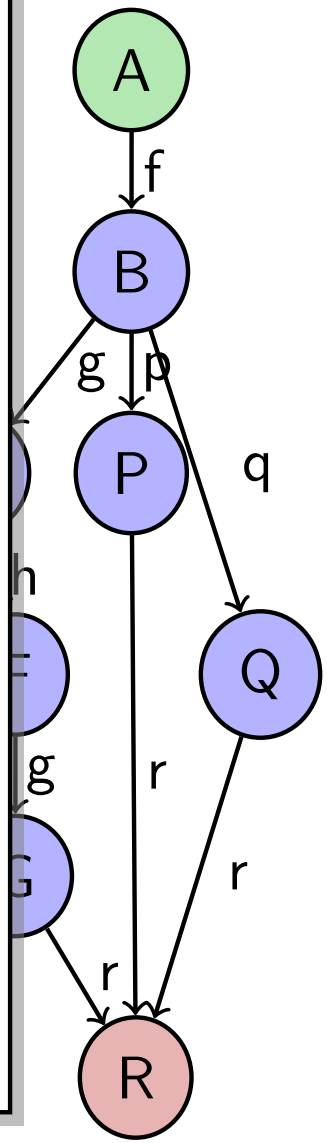
Capturing Dependencies

$B = f(A)$
$C = g(B)$
$E = f(C)$
$F = h(C)$
$G = g(E,F)$
$P = p(B)$
$Q = q(B)$
$R = r(G,P,Q)$



Ca

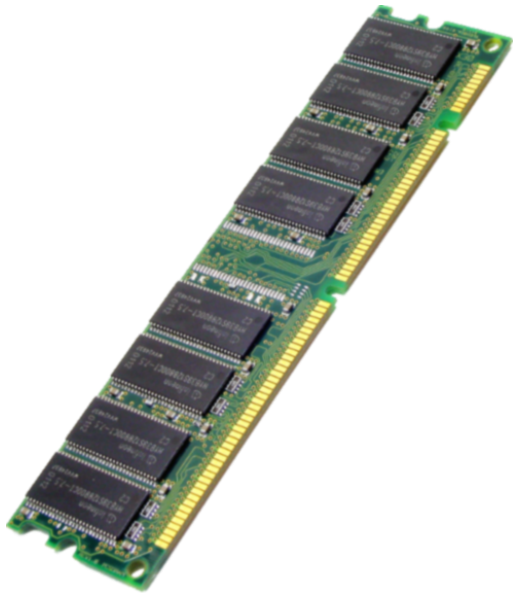
- Switch queue to out-of-order mode!
- Specify as list of events using `wait_for=` optional keyword to `enqueue_XXX`.
- Can also enqueue barrier.
- Common use case: Transmit/receive from other MPI ranks.
- Possible in hardware on Nv Fermi, AMD Cayman: Submit parallel work to increase machine use.
  - Not yet ubiquitously implemented



Memory Objects: Buffers

```
buf = cl.Buffer(context, flags, size=0, hostbuf=None)
```

- Chunk of device memory
- No type information: "Bag of bytes"
- Observe: *Not* tied to device.
  - → no fixed memory address
  - → pointers do *not* survive kernel launches
  - → movable between devices
  - → not even allocated before first use!
- `flags`:
  - `READ_ONLY/WRITE_ONLY/READ_WRITE`
  - `{ALLOC,COPY,USE}_HOST_PTR`



Memory Objects: Buffers

```
buf = cl.Buffer(context, flags, size=0, hostbuf=None)
```

`COPY_HOST_PTR`:

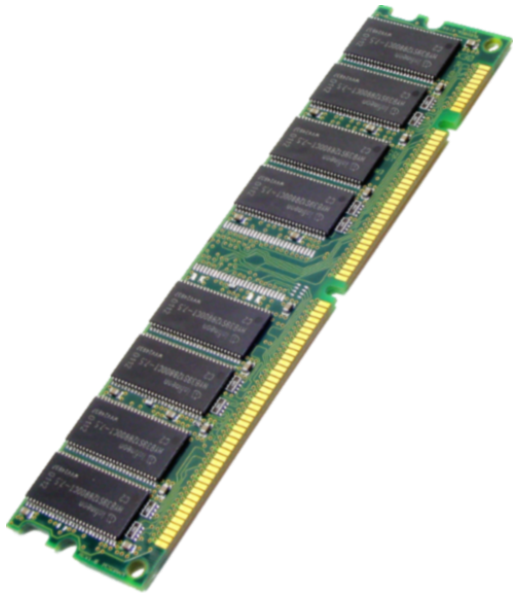
- Use `hostbuf` as initial content of buffer

`USE_HOST_PTR`:

- `hostbuf` *is* the buffer.
- Caching in device memory is allowed.

`ALLOC_HOST_PTR`:

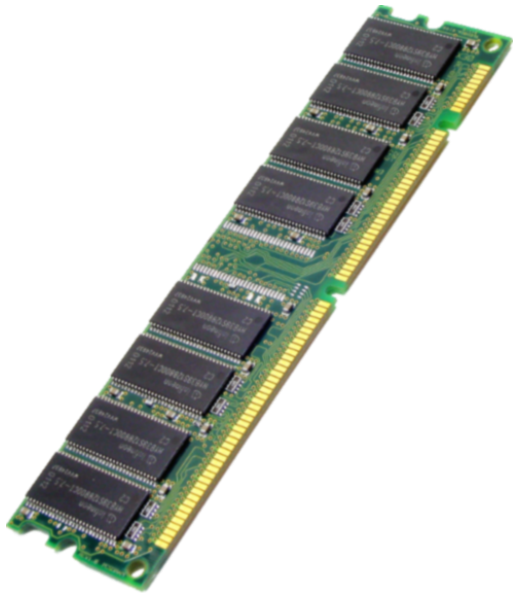
- *New* host memory (unrelated to `hostbuf`) is visible from device *and* host.



Memory Objects: Buffers

```
buf = cl.Buffer(context, flags, size=0, hostbuf=None)
```

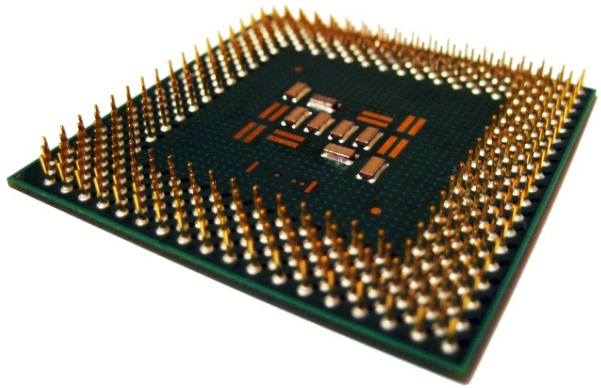
- Specify `hostbuf` or `size` (or both)
- `hostbuf`: Needs Python Buffer Interface
  e.g. `numpy.ndarray`, `str`.
  - Important: Memory layout matters
- Passed to device code as pointers
  (e.g. `float *`, `int *`)
- `enqueue_copy(queue, dest, src)`
- Can be mapped into host address space:
  `cl.MemoryMap`.



Programs and Kernels

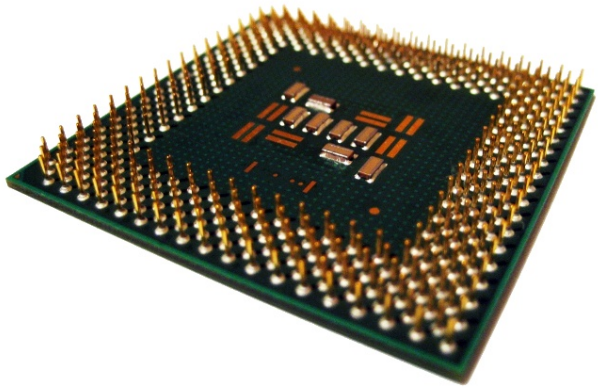
```
prg = cl.Program(context, src)
```

- `src`: OpenCL device code
  - Derivative of C99
  - Functions with `__kernel` attribute can be invoked from host
- `prg.build(options="", devices=None)`
- `kernel = prg.kernel_name`
- `kernel(queue, (Gx, Gy, Gz), (Lx, Ly, Lz), arg, ..., wait_for=None)`



Program Objects

```
kernel(queue, (Gx,Gy,Gz), (Sx,Sy,Sz), arg, ..., wait_for=None)
```

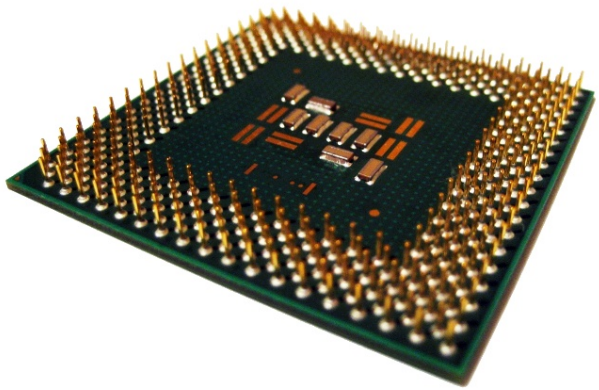


`arg` may be:

- `None` (a `NULL` pointer)
- `numpy` sized scalars: `numpy.int64`, `numpy.float32`, ...
- Anything with buffer interface: `numpy.ndarray`, `str`
- Buffer Objects
- Also: `cl.Image`, `cl.Sampler`, `cl.LocalMemory`

Program Objects

```
kernel(queue, (Gx,Gy,Gz), (Sx,Sy,Sz), arg, ..., wait_for=None)
```



Explicitly sized scalars:

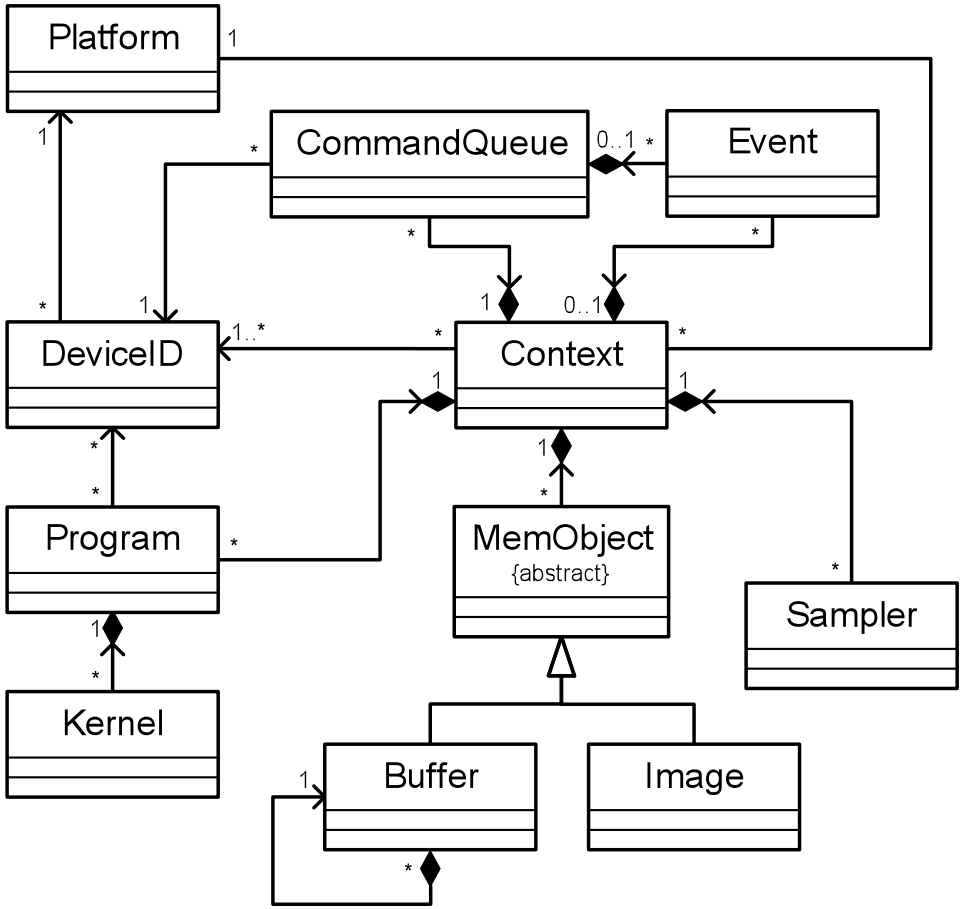
✖ Annoying, error-prone.

Better:

```
kernel.set_scalar_arg_dtypes([
    numpy.int32, None,
    numpy.float32])
```

Use `None` for non-scalars.

OpenCL Object Diagram



Credit: Khronos Group

Outline

Recap: Concurrency and Synchronization

GPUs have layers of concurrency.

Each layer has its synchronization primitives.

Recap: Concurrency and Synchronization

GPUs have layers of concurrency.
Each layer has its synchronization primitives.

- Intra-group:
  `barrier(...)`,
  `...` =
  `CLK_{LOCAL,GLOBAL}_MEM_FENCE`
- Inter-group:
  Kernel launch
- CPU-GPU:
  Command queues, Events

Synchronization between Groups

Golden Rule:

Results of the algorithm must be independent of the order in which work groups are executed.

Synchronization between Groups

Golden Rule:

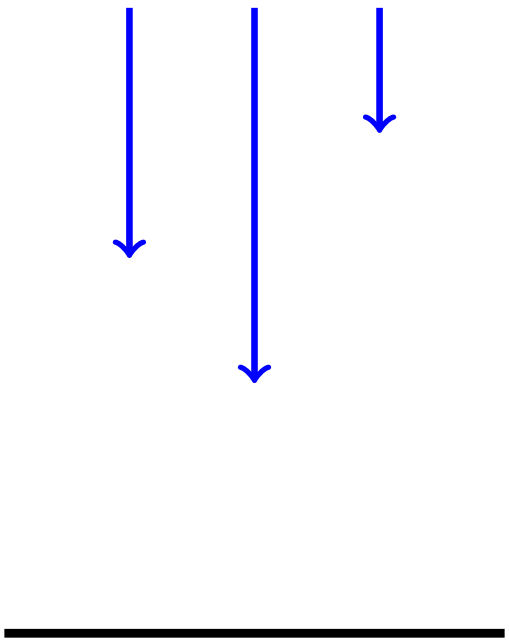
Results of the algorithm must be independent of the order in which work groups are executed.

Consequences:

- Work groups may read the same information from global memory.
- But: Two work groups may not validly write different things to the same global memory.
- Kernel launch serves as
  - Global barrier
  - Global memory fence

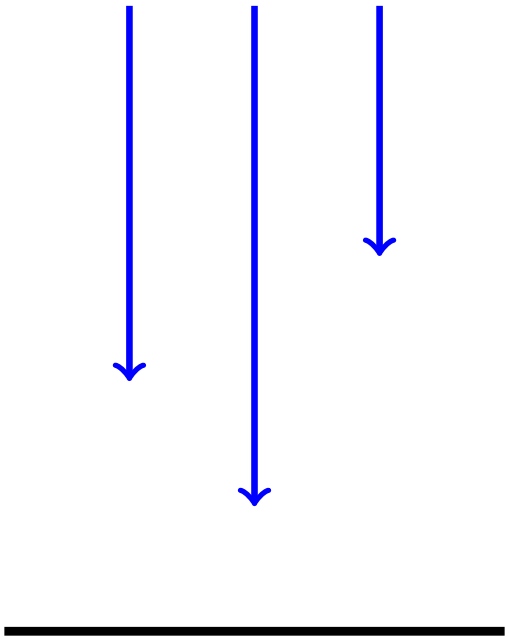
Synchronization

What is a Barrier?



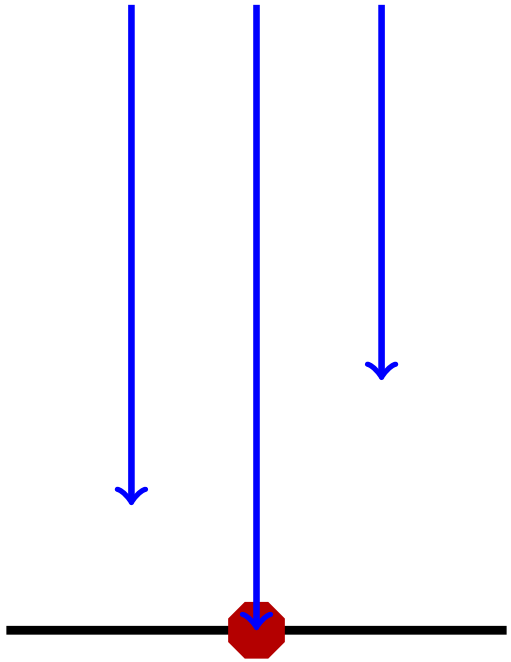
Synchronization

What is a Barrier?



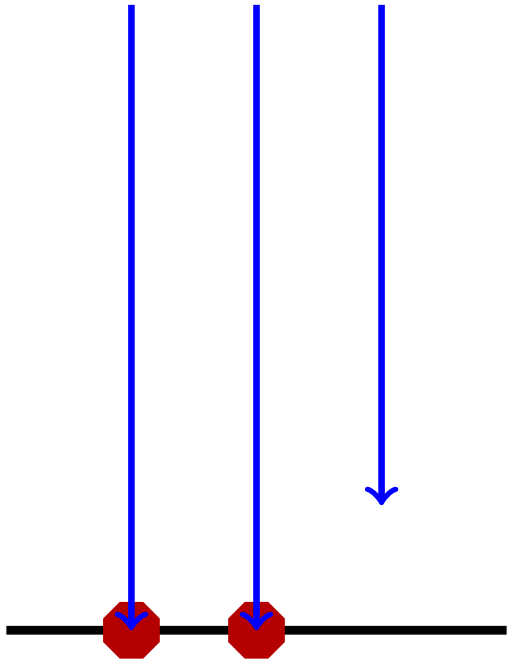
Synchronization

What is a Barrier?



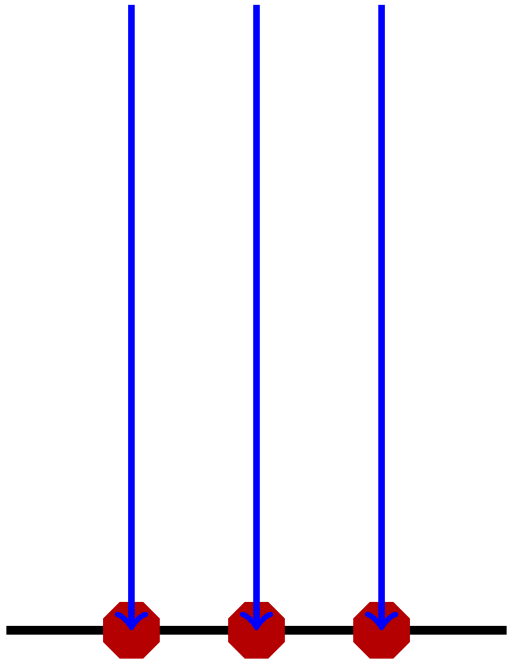
Synchronization

What is a Barrier?



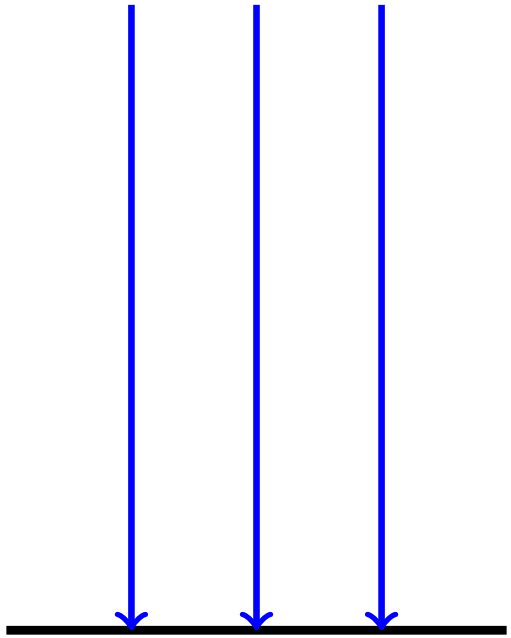
Synchronization

What is a Barrier?



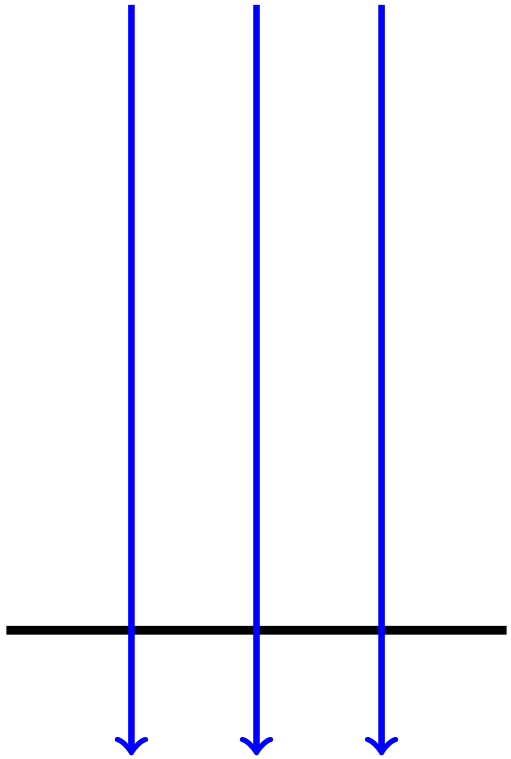
Synchronization

What is a Barrier?



Synchronization

What is a Barrier?



Outline

Outline

Map

$$y_i = f_i(x_i)$$

where $i \in \{1, \ldots, N\}$.

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Map

When does a function have a "side effect"?

In addition to producing a value, it

- modifies non-local state, or
- has an observable interaction with the outside world.

where $i \in \{1, \ldots, N\}$.

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Map

$$y_i = f_i(x_i)$$

where $i \in \{1, \ldots, N\}$.

Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

Map

$$y_i = f_i(x_i)$$

where $i \in \{1, \ldots, N\}$.

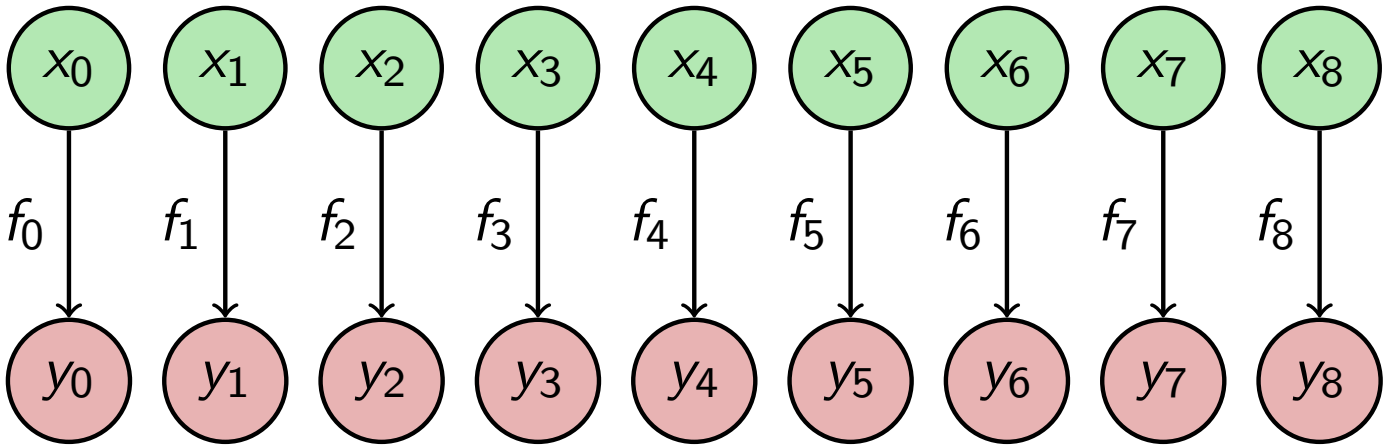
Notation: (also for rest of this lecture)

- x_i: inputs
- y_i: outputs
- f_i: (pure) functions (i.e. *no side effects*)

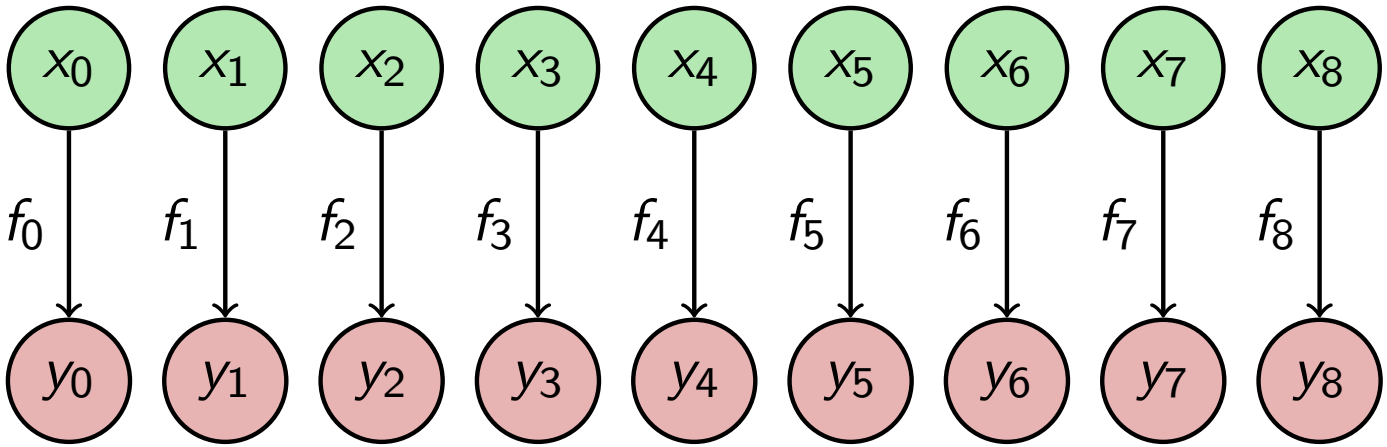
Often: $f_1 = \cdots = f_N$. Then

- Python function `map`

Map: Graph Representation



Map: Graph Representation



Trivial? Often: no.

Embarrassingly Parallel: Examples

Surprisingly useful:

- Element-wise linear algebra: Addition, scalar multiplication (*not* inner product)
- Image Processing: Shift, rotate, clip, scale, ...
- Monte Carlo simulation
- (Brute-force) Optimization
- Random Number Generation
- Encryption, Compression (after blocking)

DEMO TIME

Outline

Reduction

$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

where N is the input size.

Reduction

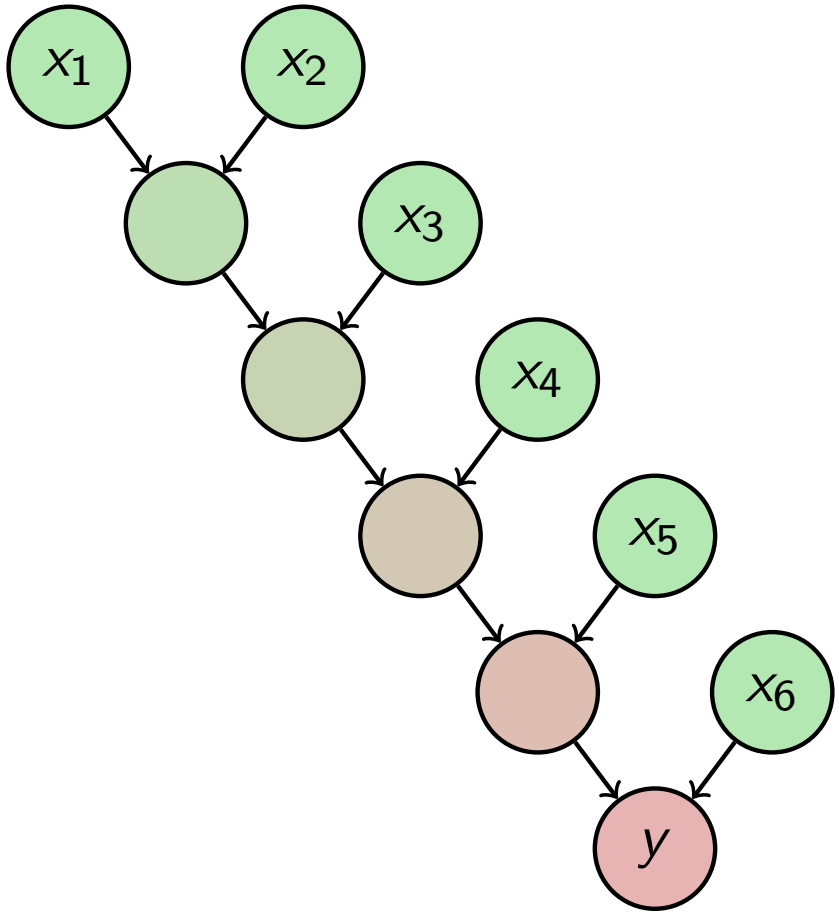
$$y = f(\cdots f(f(x_1, x_2), x_3), \ldots, x_N)$$

where N is the input size.

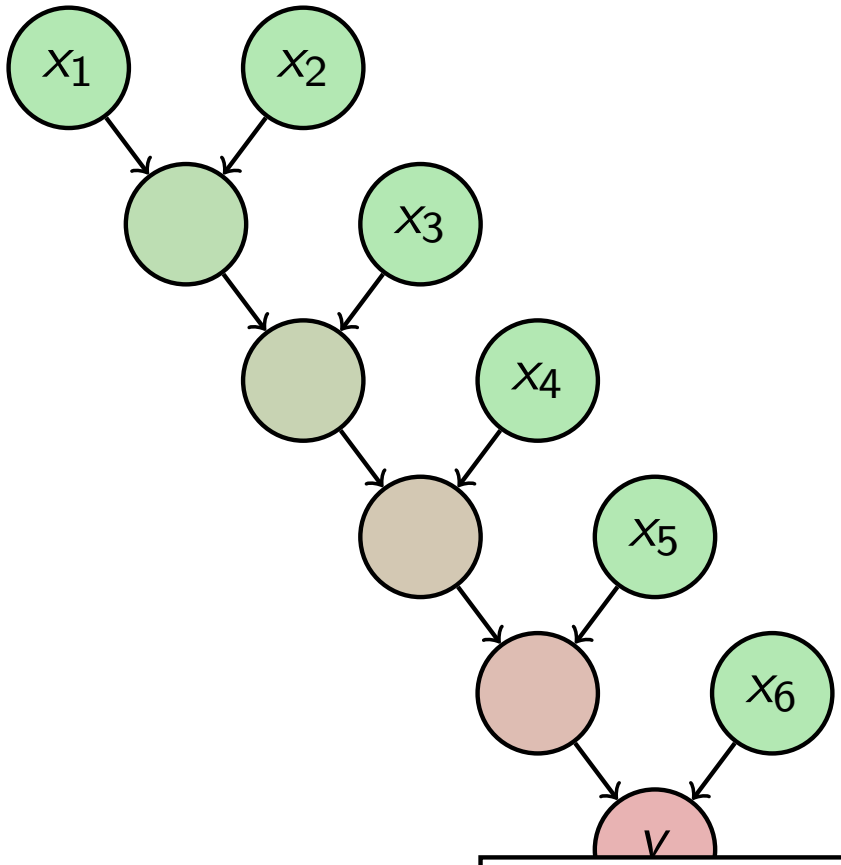
Also known as...

- Python function `reduce`

Reduction: Graph

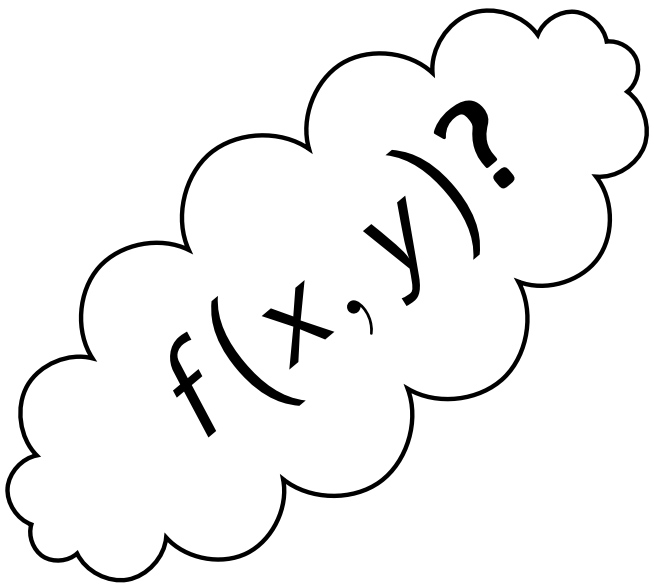


Reduction: Graph



Painful! Not parallelizable.

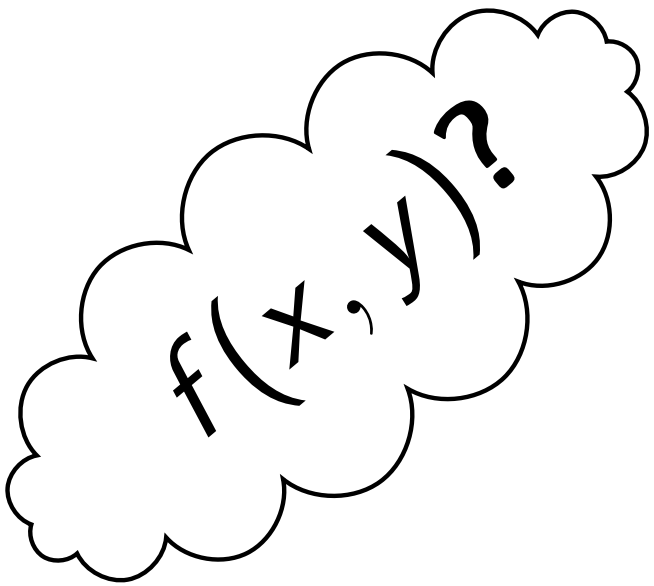
Approach to Reduction



Can we do better?

"Tree" very imbalanced. What property of f would allow 'rebalancing'?

Approach to Reduction



Can we do better?

"Tree" very imbalanced. What property of f would allow 'rebalancing'?

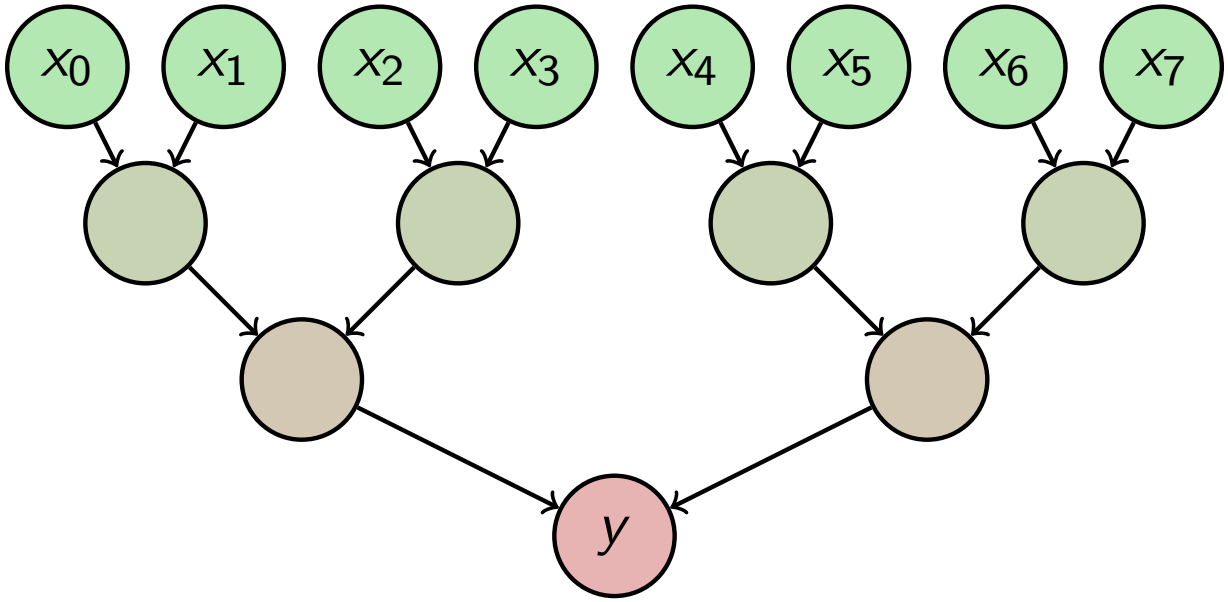
$$f(f(x,y),z) = f(x,f(y,z))$$

Looks less improbable if we let $x \circ y = f(x,y)$:

$$x \circ (y \circ z)) = (x \circ y) \circ z$$

Has a very familiar name: *Associativity*

Reduction: A Better Graph



Reduction: Examples

- Sum, Inner Product, Norm
  - Occurs in iterative methods
- Minimum, Maximum
- Data Analysis
  - Evaluation of Monte Carlo Simulations
- List Concatenation, Set Union
- Matrix-Vector product (but...)

DEMO TIME

Outline

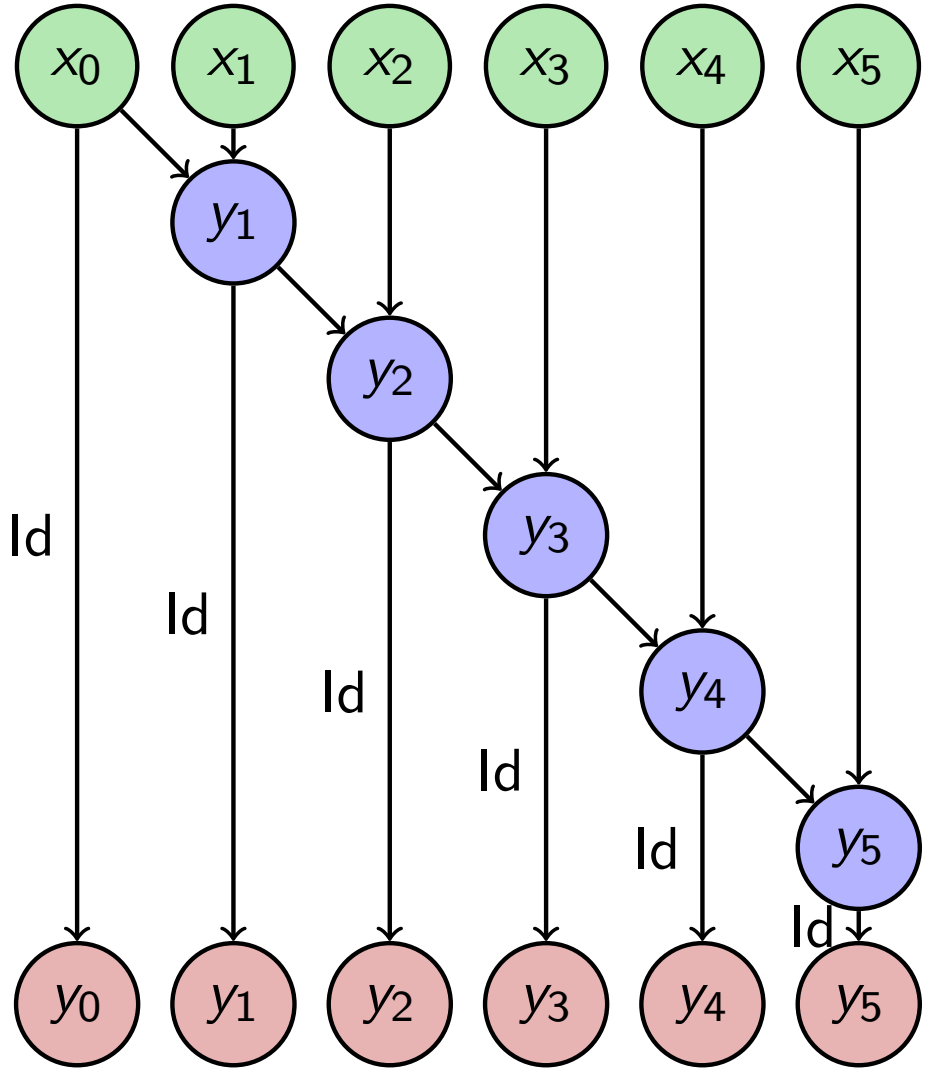
Scan

$$
\begin{aligned}
y_1 &= x_1 \\
y_2 &= f(y_1, x_2) \\
&\vdots = \vdots \\
y_N &= f(y_{N-1}, x_N)
\end{aligned}
$$

where N is the input size. (Think: N large, $f(x, y) = x + y$)

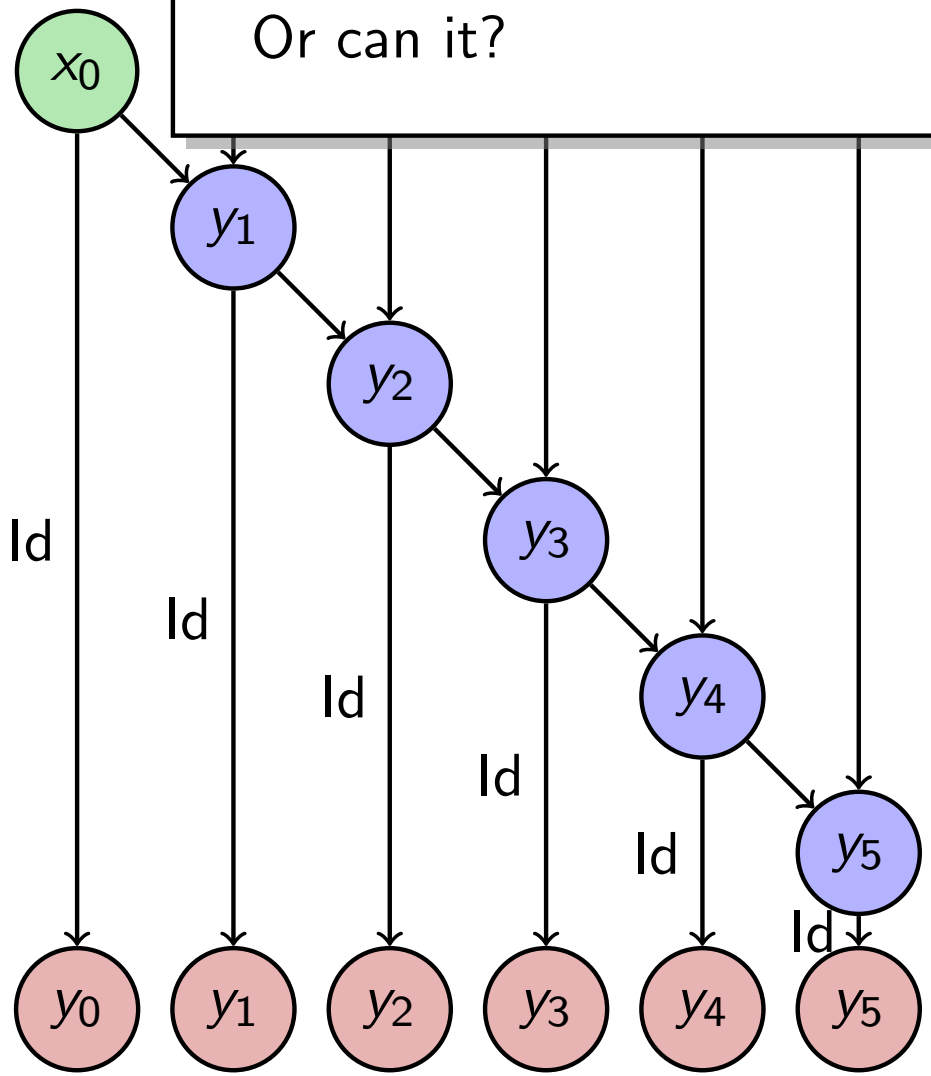
- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

Scan: Graph



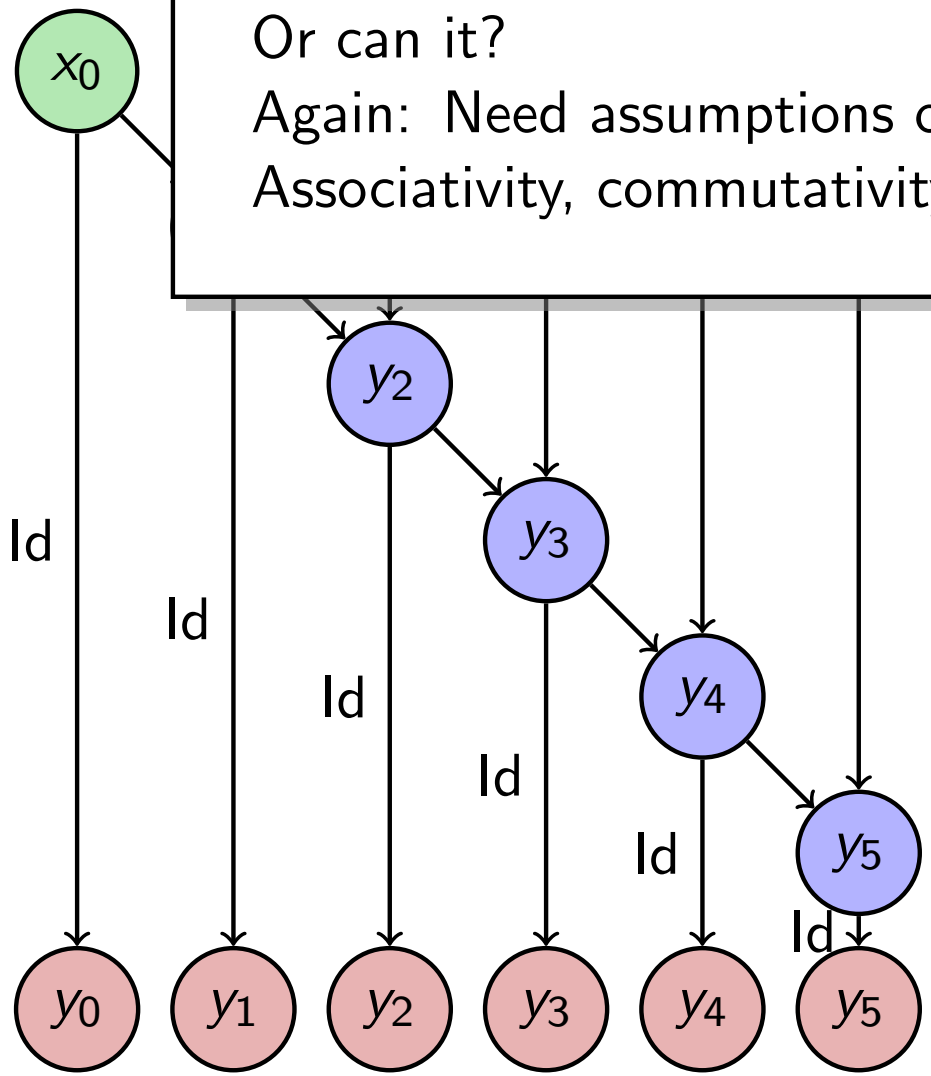
Scan: Graph

This can't possibly be parallelized.
Or can it?

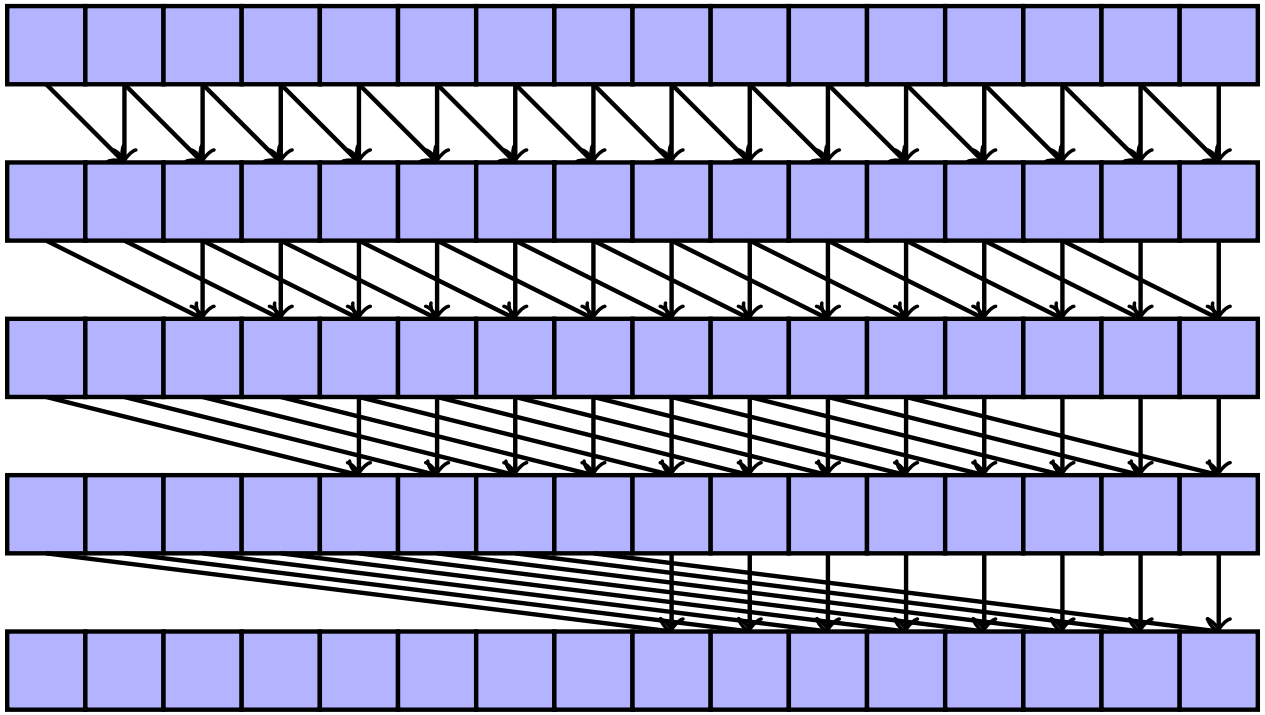


Scan: Graph

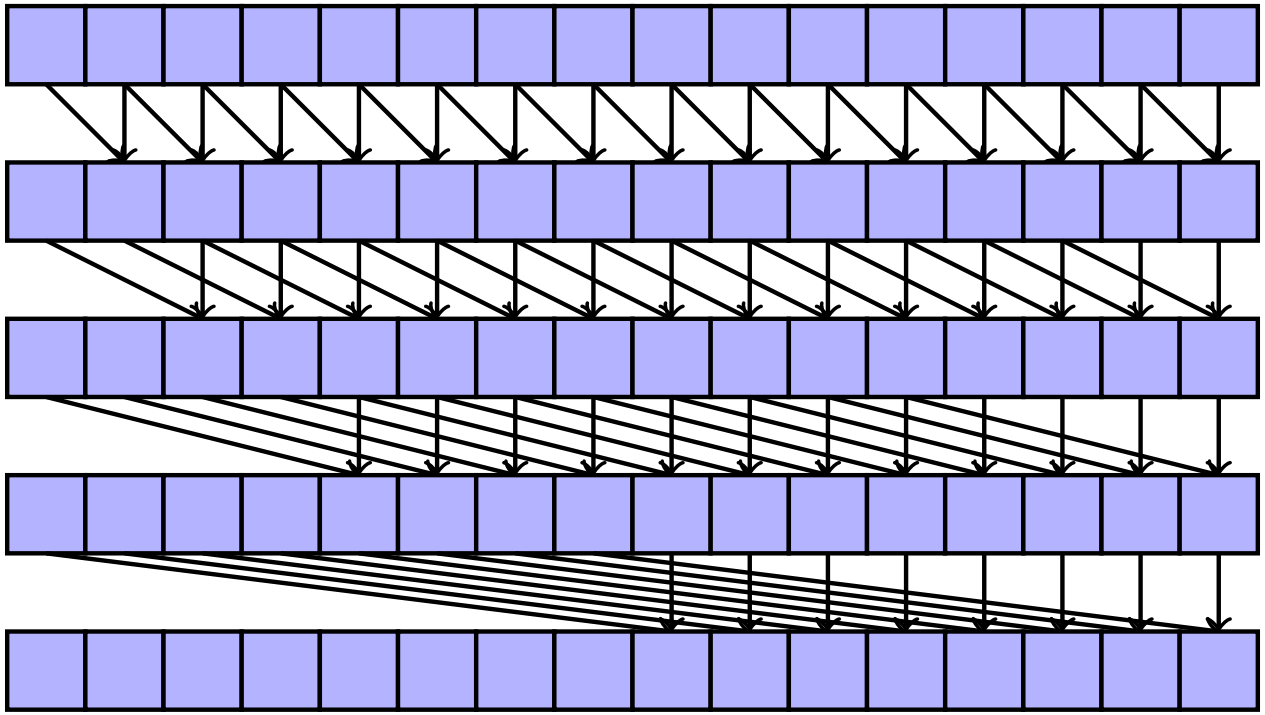
This can't possibly be parallelized.
Or can it?
Again: Need assumptions on f.
Associativity, commutativity.



Scan: Implementation



Scan: Implementation

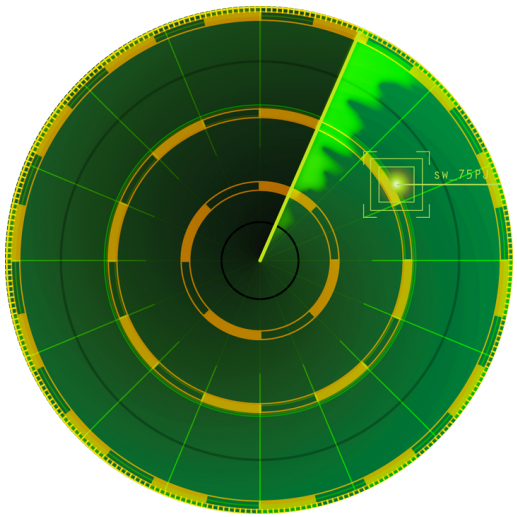


Work-efficient?

Scan: Examples

Low-level building block for many higher-level algorithms algorithms

- Index computations (!)
  - E.g. sorting
- Anything with a loop-carried dependence
- One row of triangular solve
- Segment numbering if boundaries are known
- FIR/IIR Filtering
- G.E. Blelloch: Prefix Sums and their Applications



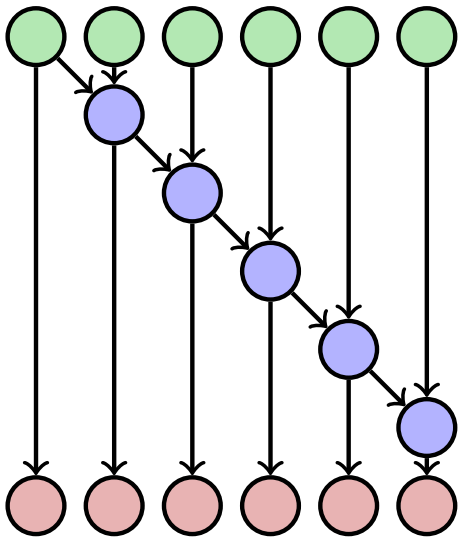
Scan: Issues

- Subtlety: Inclusive/Exclusive Scan
- Pattern sometimes hard to recognize
  - But shows up surprisingly often
  - Need to prove associativity/commutativity

DEMO TIME

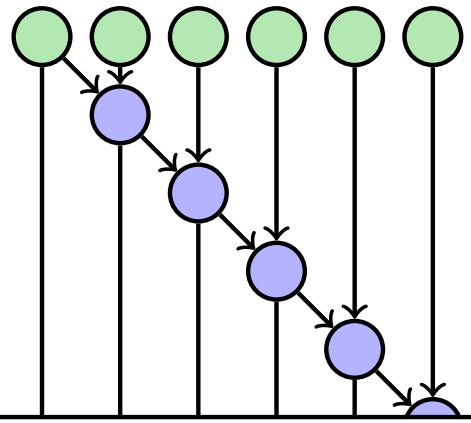
Scan: Features

- "Map" processing on input: $f(x_i)$
  - Also: stencils $f(x_{i-1}, x_i)$
- "Map" processing on output
  - Output stencils
  - Inclusive/Exclusive scan
- Segmented scan
- Works on compound types
- Efficient!



Scan: Features

- "Map" processing on input: $f(x_i)$
  - Also: stencils $f(x_{i-1}, x_i)$
- "Map" processing on output
  - Output stencils
  - Inclusive/Exclusive scan
- Segmented scan
- Works on compound
- Efficient!



Scan: a **fundamental** parallel primitive.

Anything involving index changes/renumbering!
(e.g. sort, filter, ...)

Scan: More Algorithms

- `copy_if`
- `remove_if`
- `partition`
- `unique`
- `sort` (plain and key-value)
- `build_list_of_lists`
- `bin_sort`

All in `pyopencl`, all built on scan.

Outline

Code generation

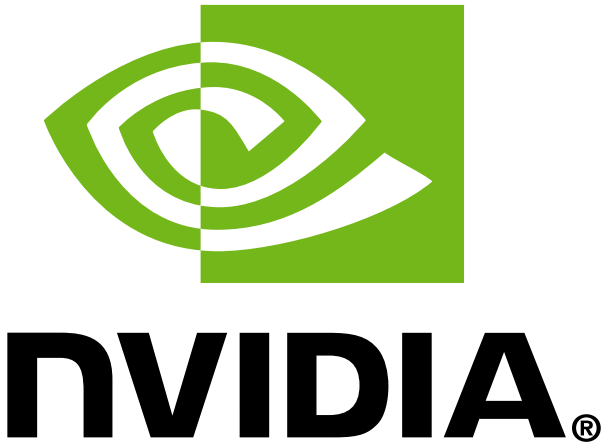
Three main ways to generate code:

- Textual substitution
- Templating
- Abstract syntax trees

DEMO TIME

Outline

The Nvidia CL implementation



Targets only GPUs

Notes:

- Nearly identical to CUDA
  - No native C-level JIT in CUDA ($\rightarrow$ PyCUDA)
- Page-locked memory:
  Use `CL_MEM_ALLOC_HOST_PTR`.
  (Careful: double meaning)

The Apple CL implementation

Targets CPUs and GPUs

General notes:

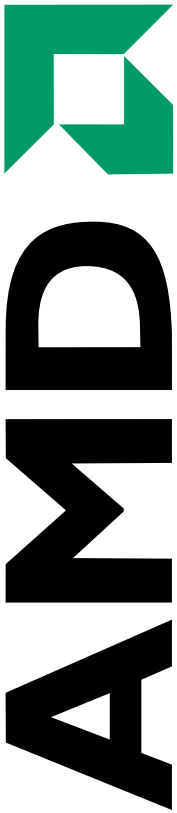
- Different header name
  `OpenCL/cl.h` instead of `CL/cl.h`
  Use `-framework OpenCL` for C access.
- Beware of imperfect compiler cache implementation
  (ignores include files)

CPU notes:

- One work item per processor

GPU similar to hardware vendor implementation.
(New: Intel w/ Sandy Bridge)

The AMD CL implementation



Targets CPUs and GPUs (from both AMD and Nvidia)

GPU notes:

- Wide SIMD groups (64)
- GCN: Vector *and* scalar unit (previously VLIW4/5)
  - *very* flop-heavy machine
  - $\rightarrow$ ILP and explicit SIMD

CPU notes:

- Many work items per processor (emulated)
- "APU": Growing CPU/GPU integration

The Intel CL implementation

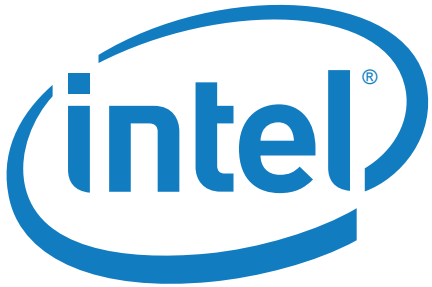
CPUs, GPUs with Ivy Bridge+

CPU notes:

- Good vectorizing compiler
- Only implementation of out-of-order queues for now
- Based on Intel TBB

GPU notes:

- Flexible design: SIMDm VLIWn
- Lots of fixed-function hardware
- Last-level Cache (LLC) integrated between CPU and GPU



Questions?

?