

Part 3: OpenCL

Andreas Klöckner

Computer Science
University of Illinois at Urbana-Champaign

Outline

- 1 OpenCL
- 2 PyOpenCL
- 3 Parallel patterns

What is OpenCL?

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- Comes with RTCG

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)



Who?

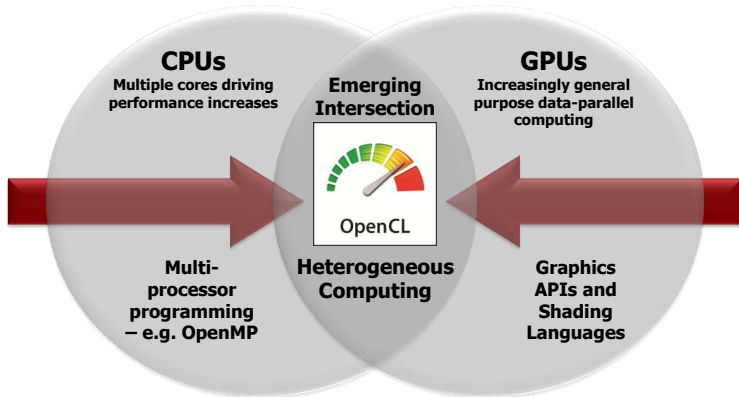
- **Diverse industry participation**
 - Processor vendors, system OEMs, middleware vendors, application developers
- **Many industry-leading experts involved in OpenCL's design**
 - A healthy diversity of industry perspectives
- **Apple made initial proposal and is very active in the working group**
 - Serving as specification editor



© Copyright Khronos Group, 2010 - Page 4

Credit: Khronos Group

Why?



OpenCL is a programming framework for heterogeneous compute resources

© Copyright Khronos Group, 2010 - Page 3

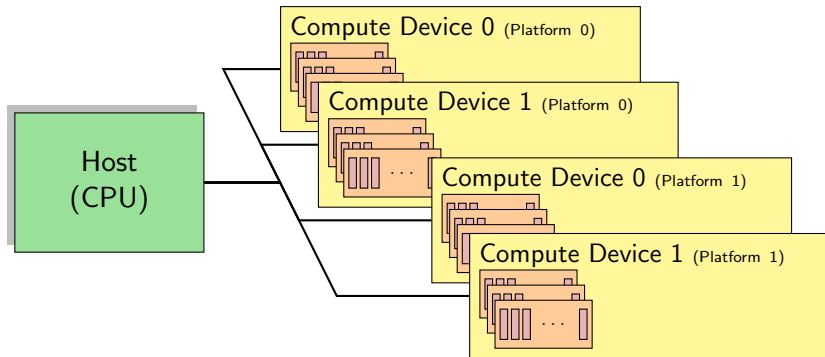
Credit: Khronos Group

OpenCL: Computing as a Service

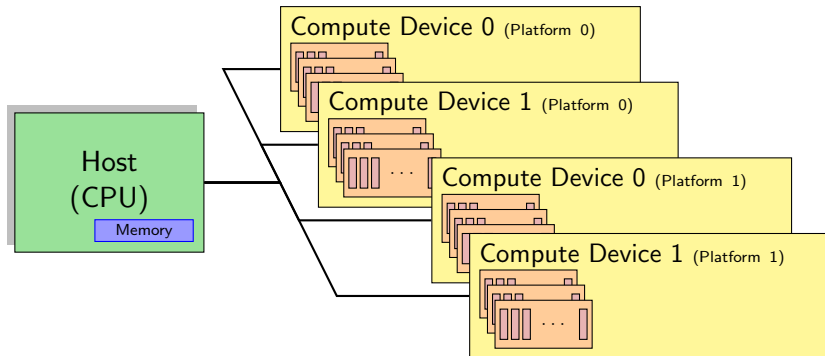


Host
(CPU)

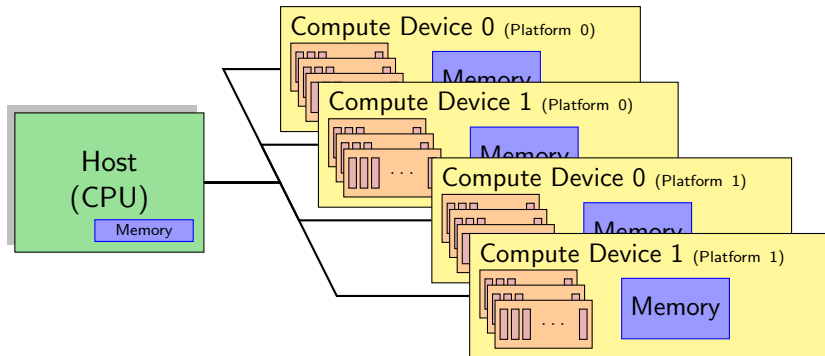
OpenCL: Computing as a Service



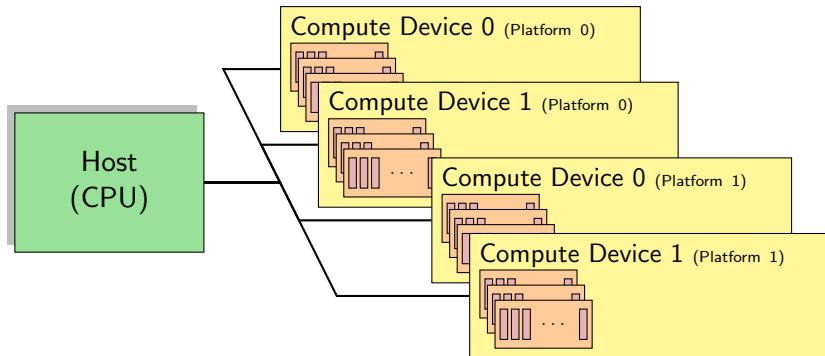
OpenCL: Computing as a Service



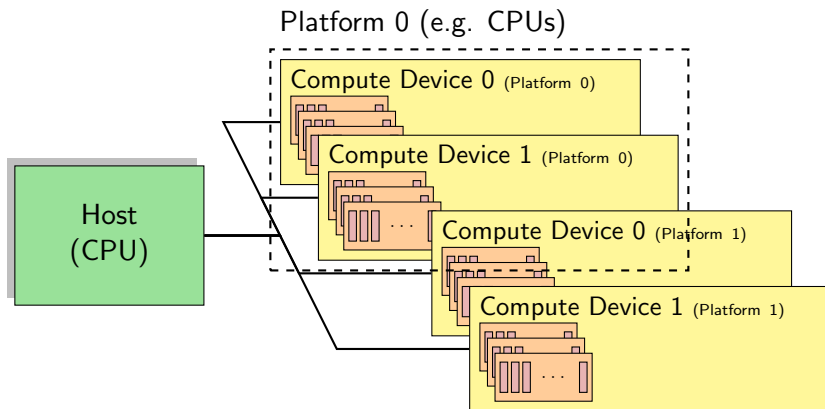
OpenCL: Computing as a Service



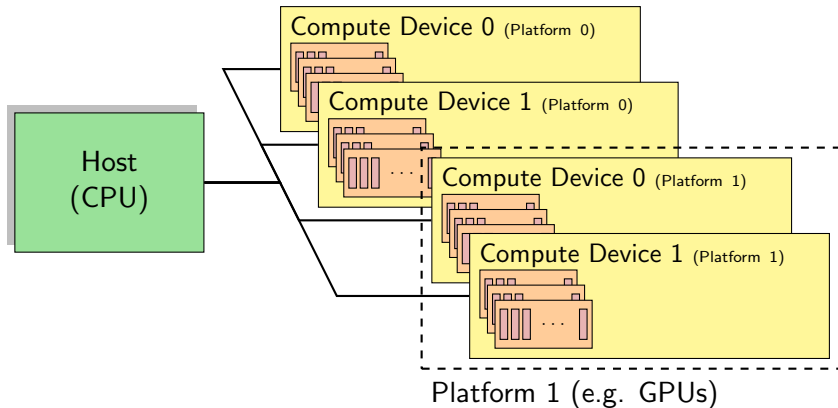
OpenCL: Computing as a Service



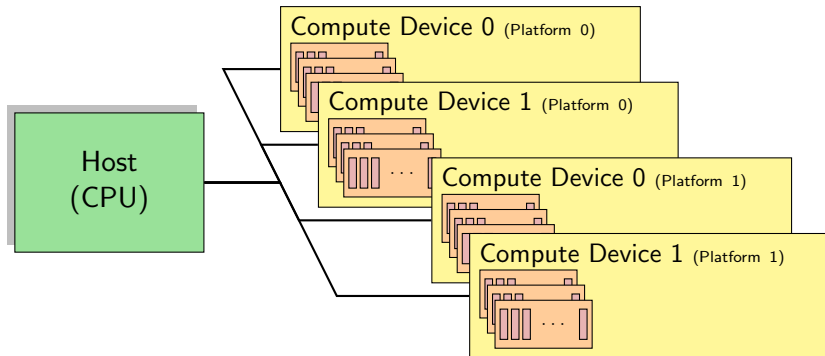
OpenCL: Computing as a Service



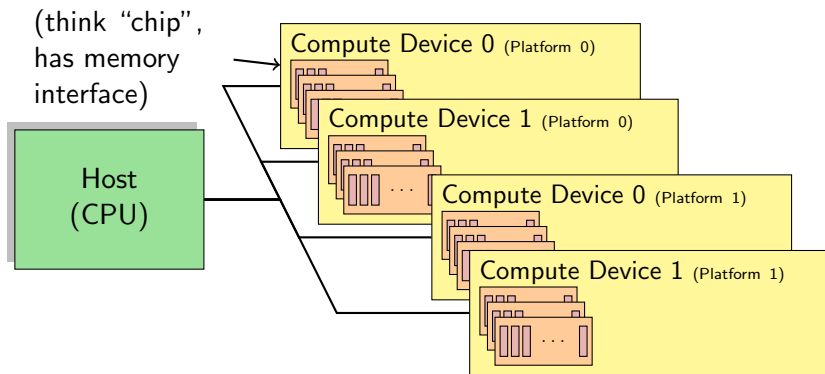
OpenCL: Computing as a Service



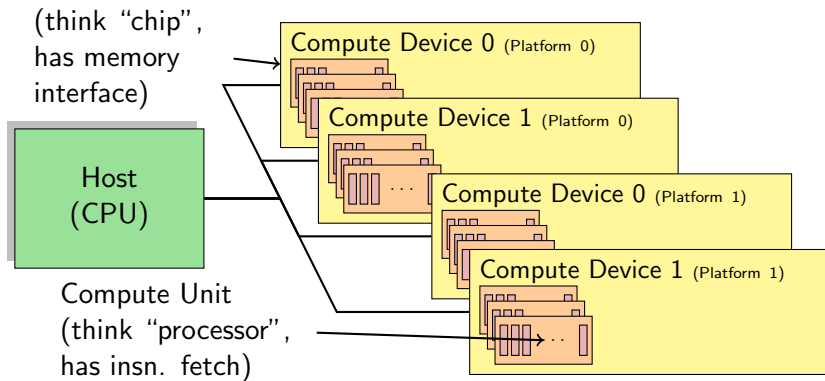
OpenCL: Computing as a Service



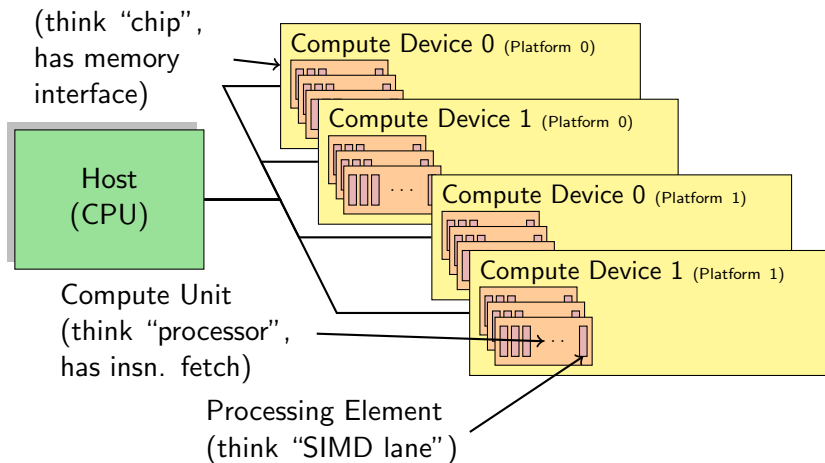
OpenCL: Computing as a Service



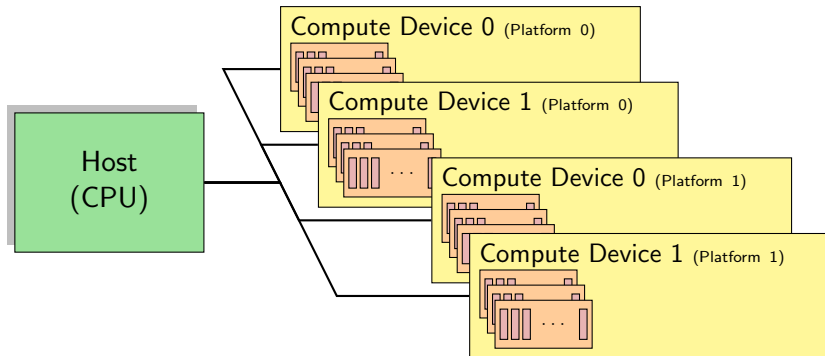
OpenCL: Computing as a Service



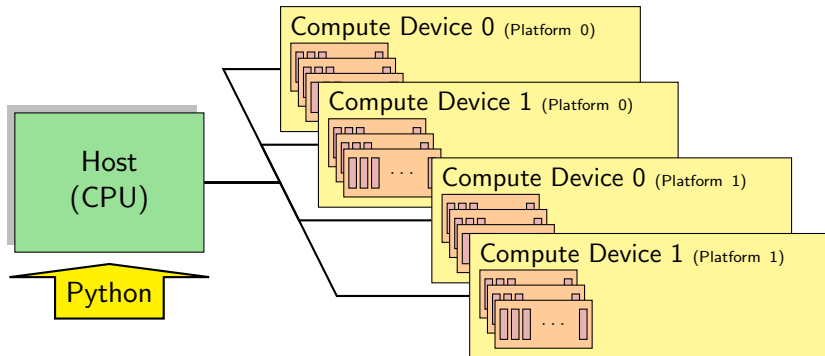
OpenCL: Computing as a Service



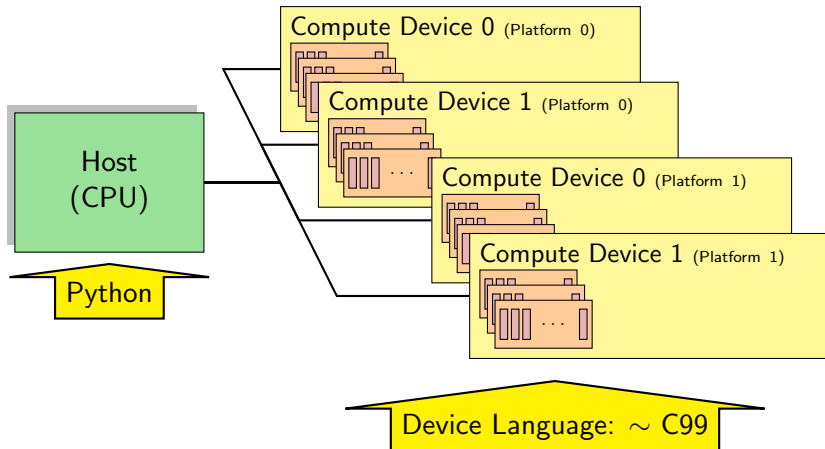
OpenCL: Computing as a Service



OpenCL: Computing as a Service



OpenCL: Computing as a Service



Connection: Hardware \leftrightarrow Programming Model

Fetch/
Decode

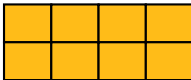


32 kiB Ctx
Private
("Registers")

16 kiB Ctx
Shared

Connection: Hardware \leftrightarrow Programming Model

Fetch/
Decode

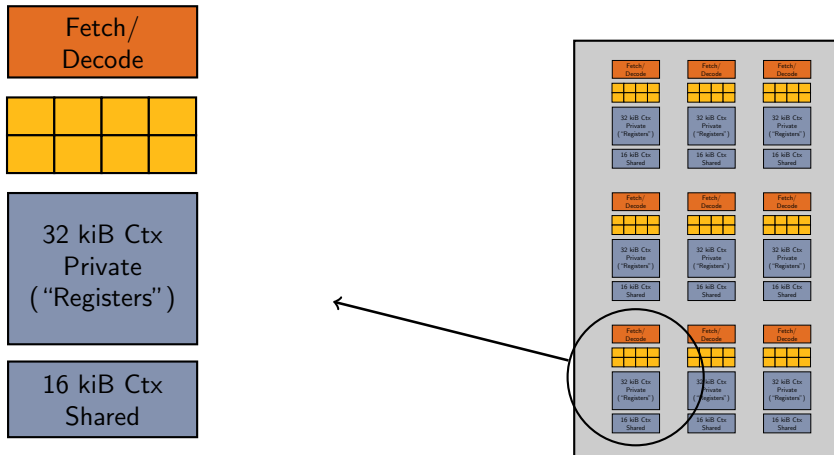


32 kiB Ctx
Private
("Registers")

16 kiB Ctx
Shared

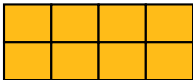


Connection: Hardware \leftrightarrow Programming Model



Connection: Hardware \leftrightarrow Programming Model

Fetch/
Decode

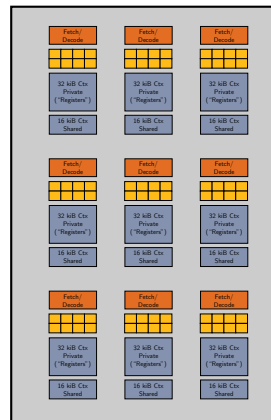


32 kiB Ctx
Private
("Registers")

16 kiB Ctx
Shared



Connection: Hardware \leftrightarrow Programming Model

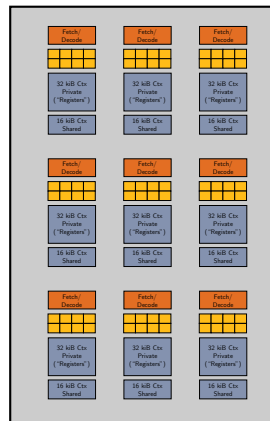


Connection: Hardware \leftrightarrow Programming Model

Who cares how many cores?

Idea:

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



Connection: Hardware \leftrightarrow Programming Model

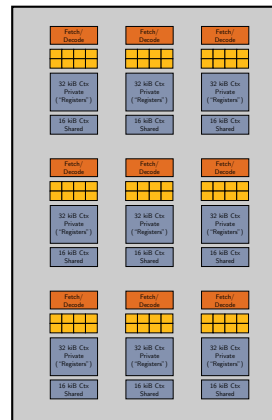
Who cares how many cores?

Consider: Which is easy to do automatically?

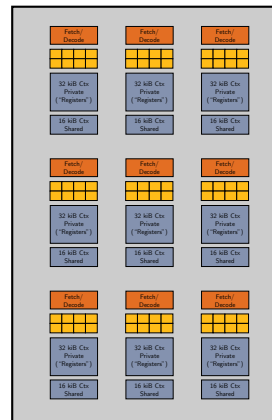
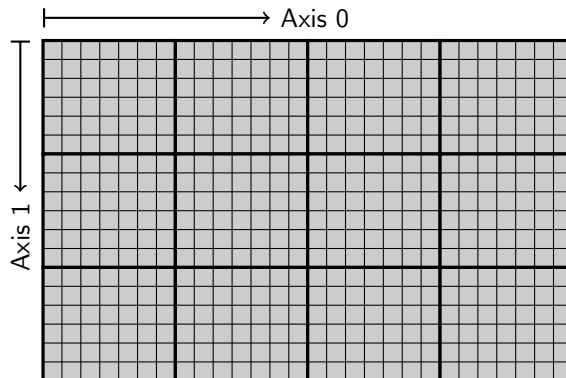
- Parallel program \rightarrow sequential hardware or
- Sequential program \rightarrow parallel hardware?



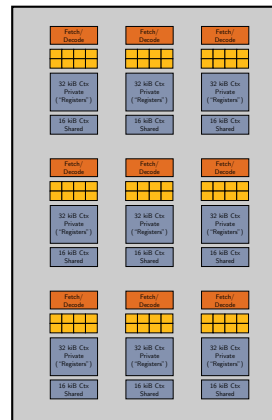
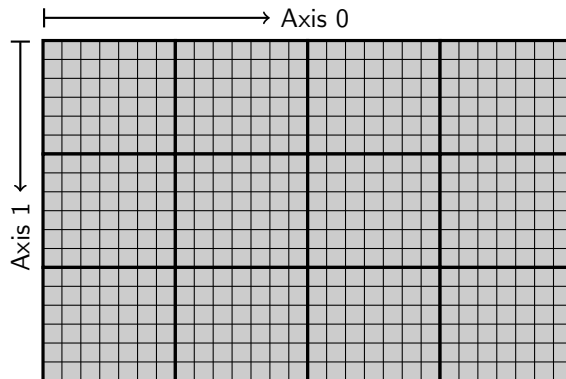
Connection: Hardware \leftrightarrow Programming Model



Connection: Hardware \leftrightarrow Programming Model

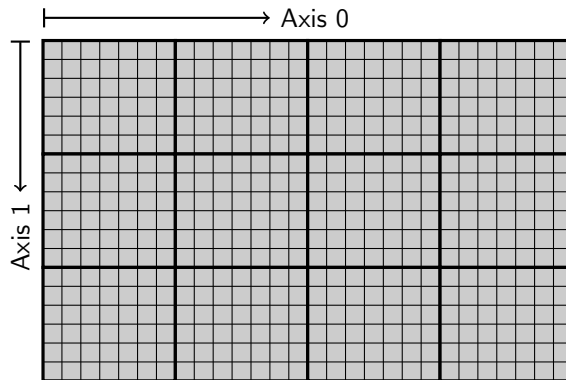


Connection: Hardware \leftrightarrow Programming Model

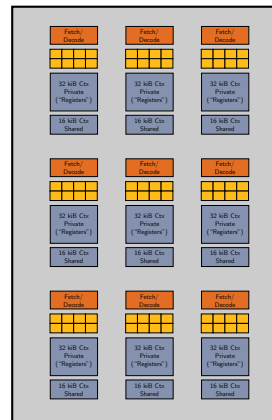


Hardware

Connection: Hardware \leftrightarrow Programming Model

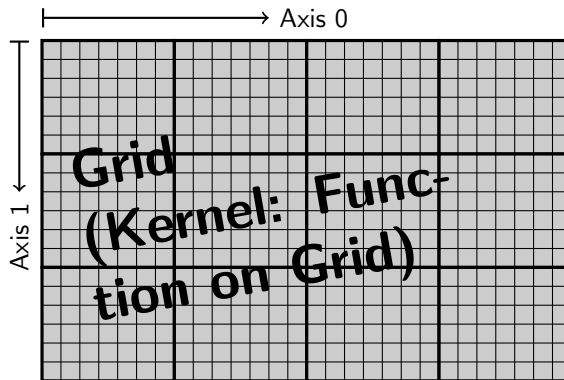


Software representation

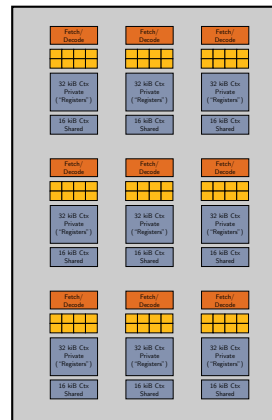


Hardware

Connection: Hardware \leftrightarrow Programming Model

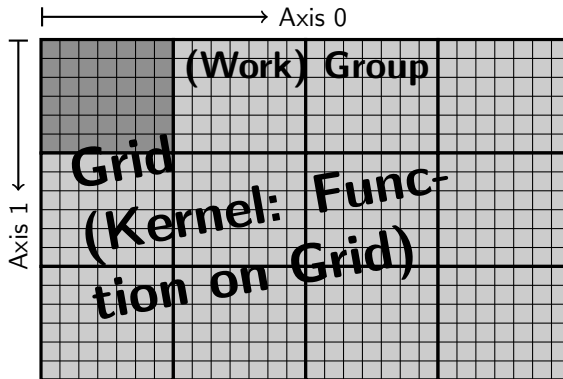


Software representation

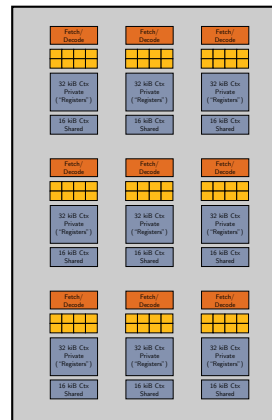


Hardware

Connection: Hardware \leftrightarrow Programming Model

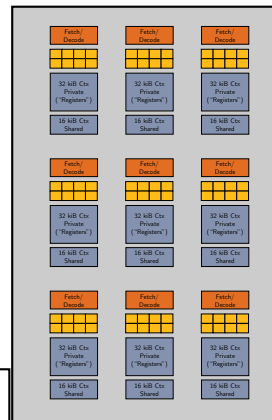
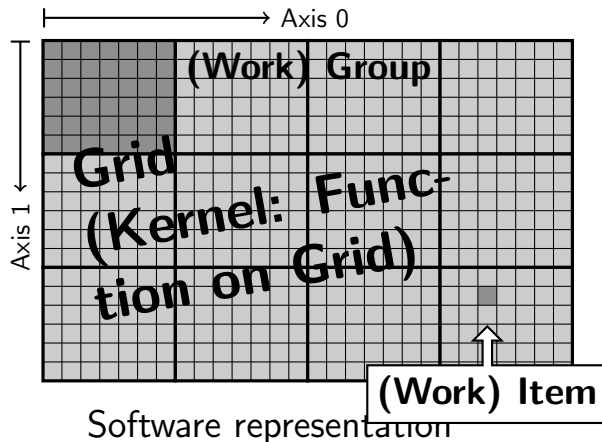


Software representation

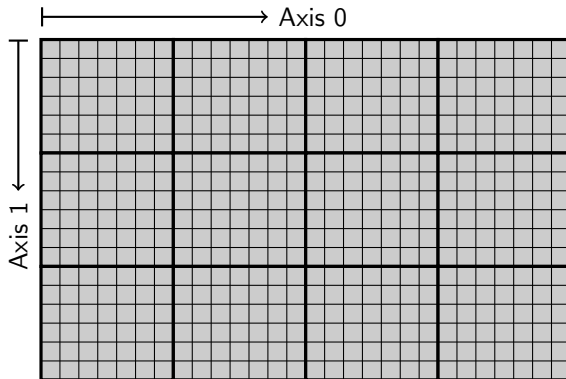


Hardware

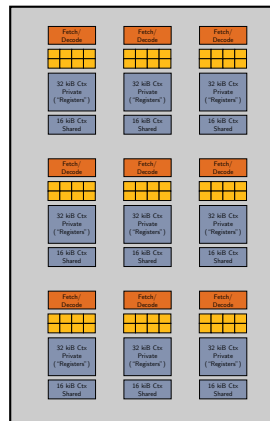
Connection: Hardware \leftrightarrow Programming Model



Connection: Hardware \leftrightarrow Programming Model

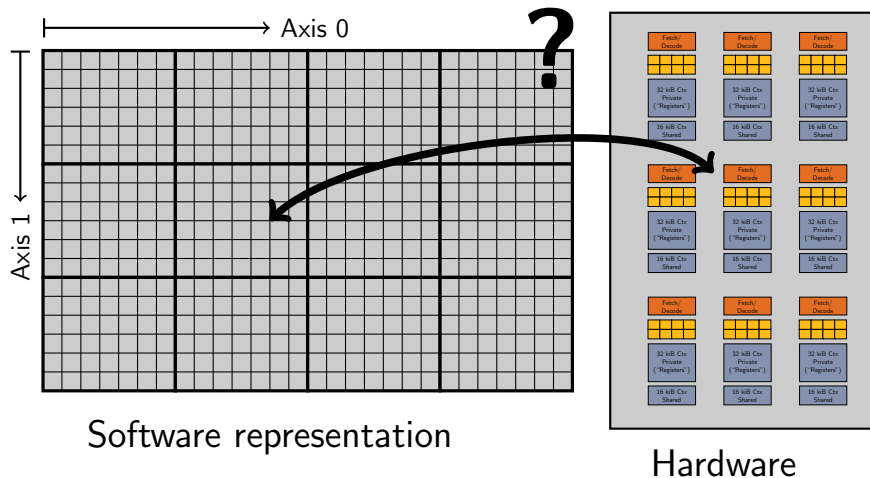


Software representation

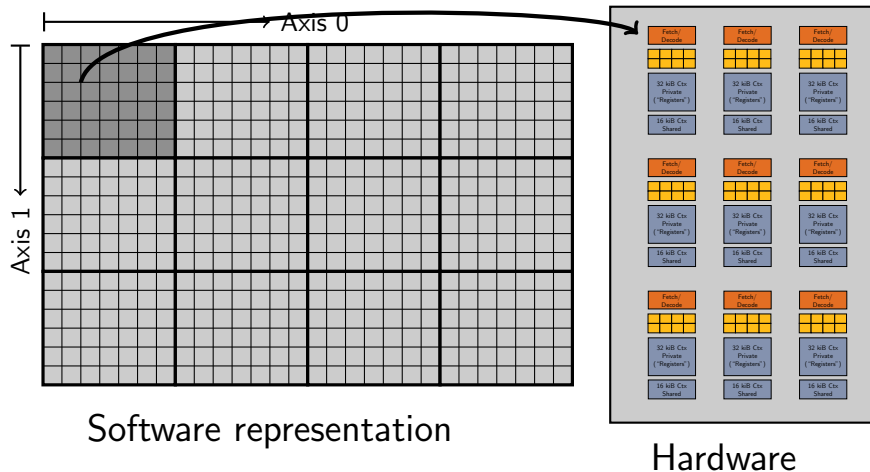


Hardware

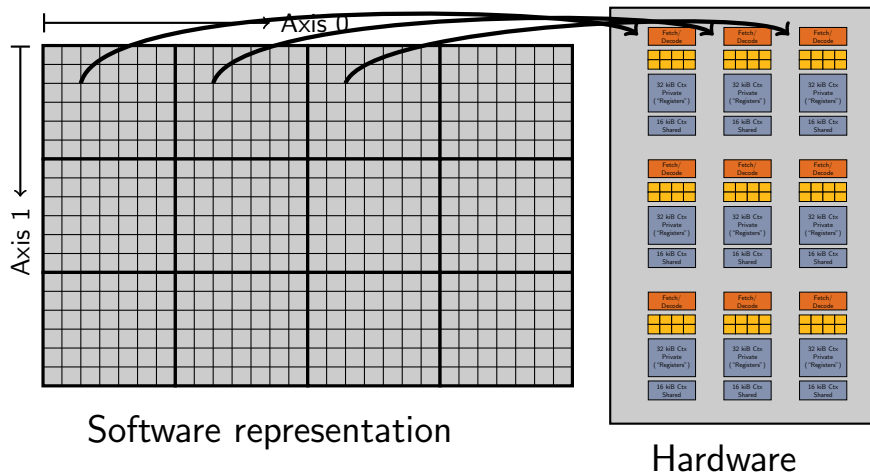
Connection: Hardware \leftrightarrow Programming Model



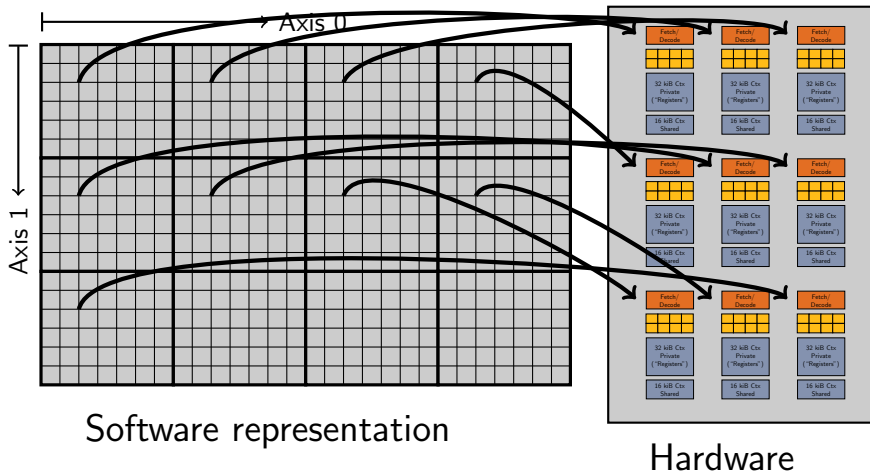
Connection: Hardware \leftrightarrow Programming Model



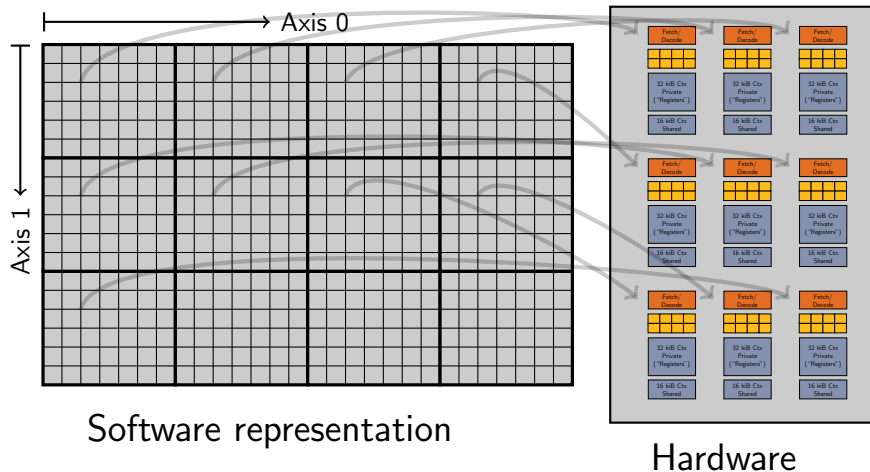
Connection: Hardware \leftrightarrow Programming Model



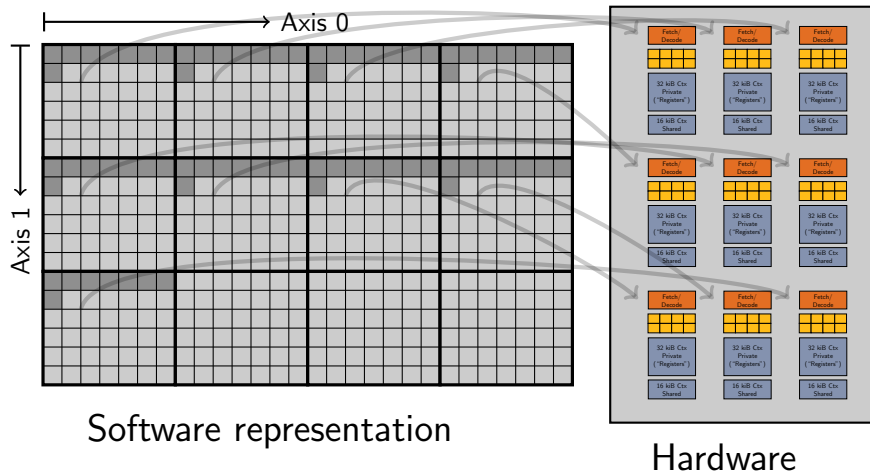
Connection: Hardware \leftrightarrow Programming Model



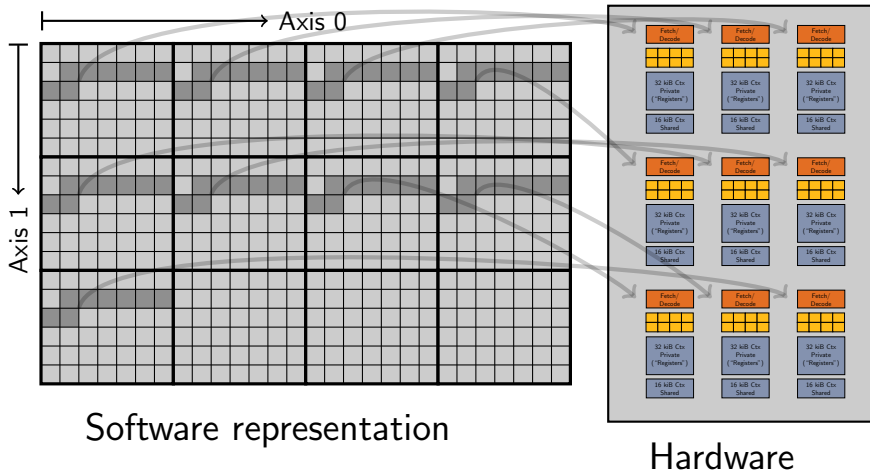
Connection: Hardware \leftrightarrow Programming Model



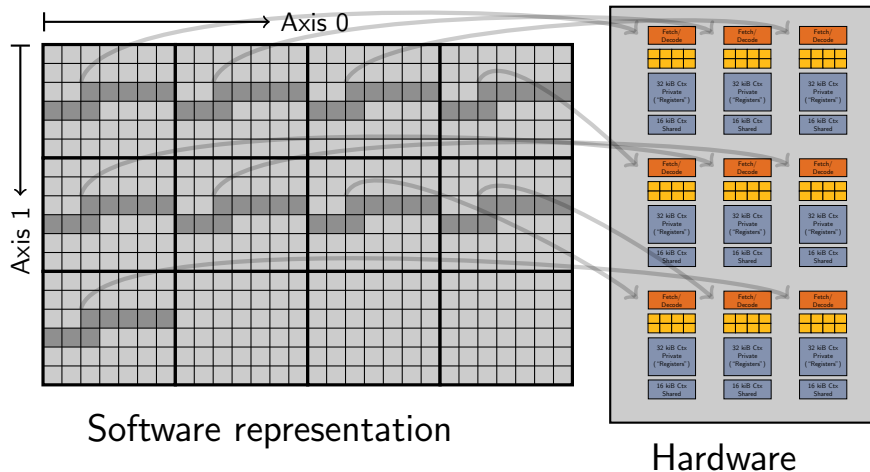
Connection: Hardware \leftrightarrow Programming Model



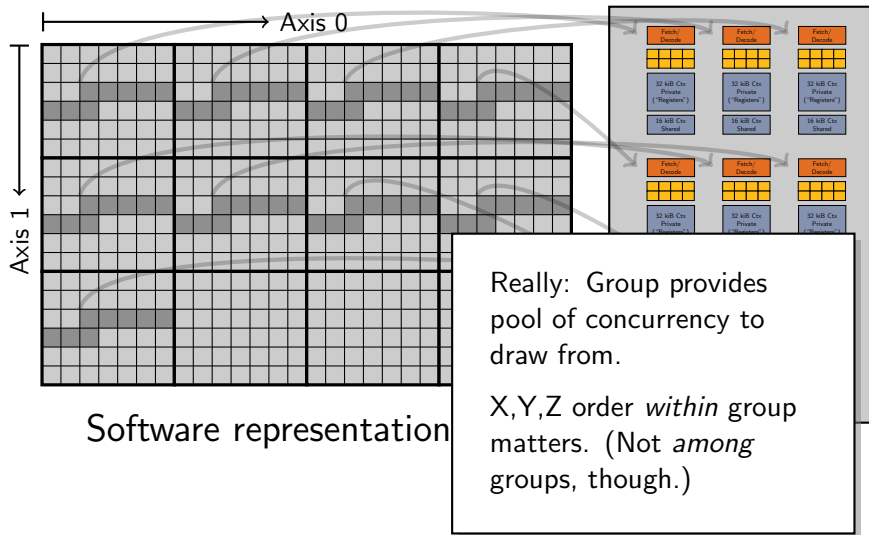
Connection: Hardware \leftrightarrow Programming Model



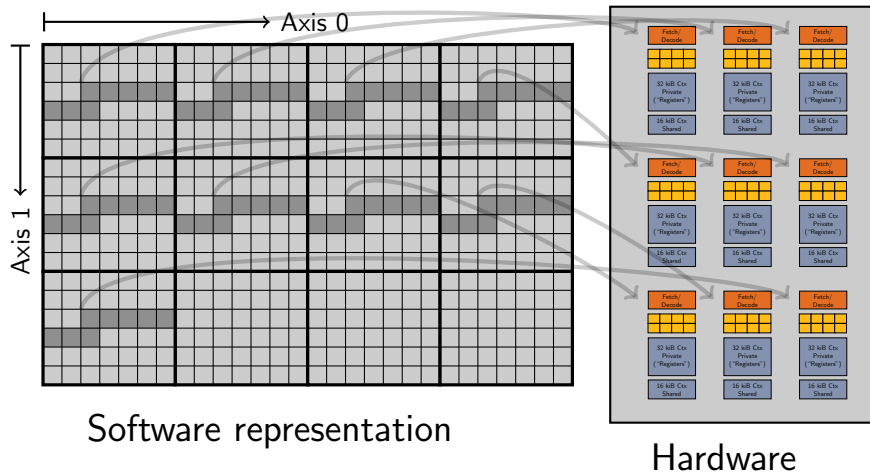
Connection: Hardware \leftrightarrow Programming Model



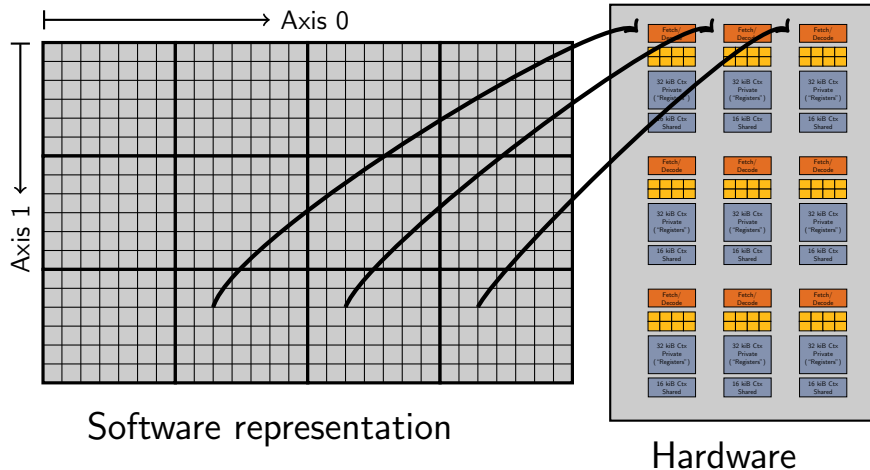
Connection: Hardware \leftrightarrow Programming Model



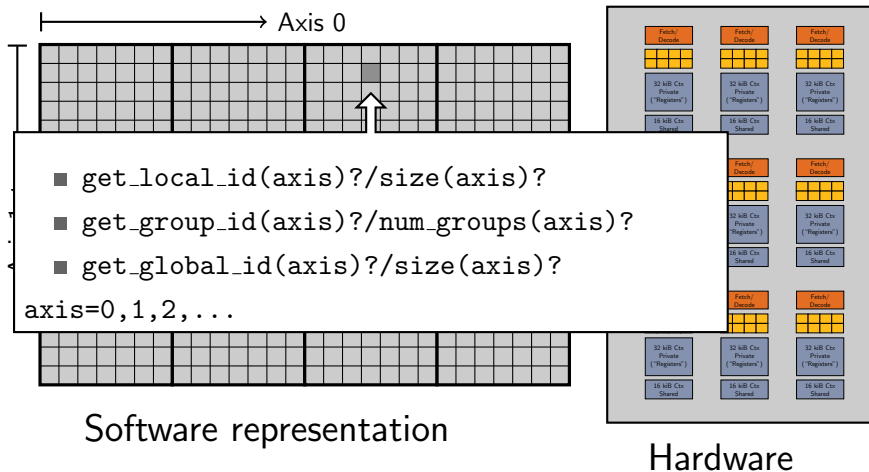
Connection: Hardware \leftrightarrow Programming Model



Connection: Hardware \leftrightarrow Programming Model

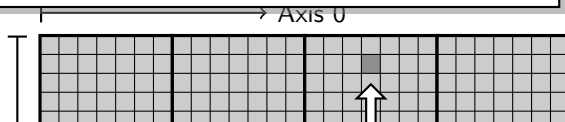


Connection: Hardware \leftrightarrow Programming Model



Computing 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

1 OpenCL

2 PyOpenCL

3 Parallel patterns

DEMO TIME

Outline

1 OpenCL

2 PyOpenCL

3 Parallel patterns

- Map
- Reduce
- Scan

Outline

1 OpenCL

2 PyOpenCL

3 Parallel patterns

- Map

- Reduce

- Scan

Map

$$y_i = f_i(x_i)$$

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

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 $i \in \{1, \dots, n\}$.

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 $i \in \{1, \dots, N\}$.

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 $i \in \{1, \dots, N\}$.

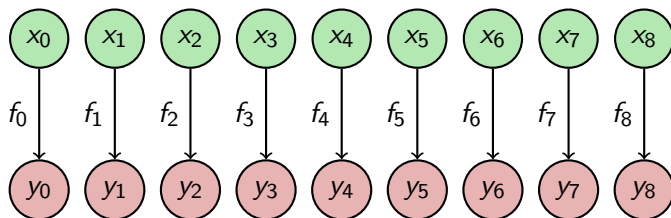
Notation: (also for rest of this lecture)

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

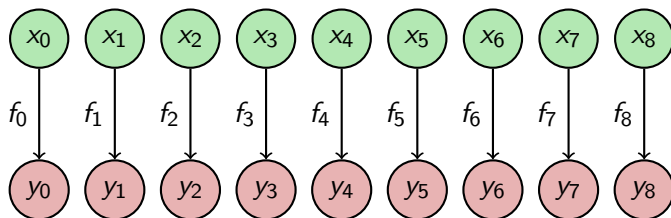
Often: $f_1 = \dots = f_N$. Then

- Python function `map`

Map: Graph Representation



Map: Graph Representation

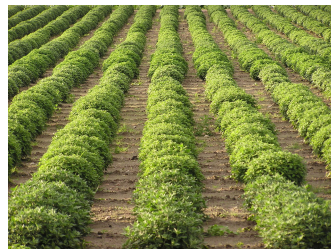


Trivial? Often: no.

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

1 OpenCL

2 PyOpenCL

3 Parallel patterns

- Map

- **Reduce**

- Scan

Reduction

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

where N is the input size.

Reduction

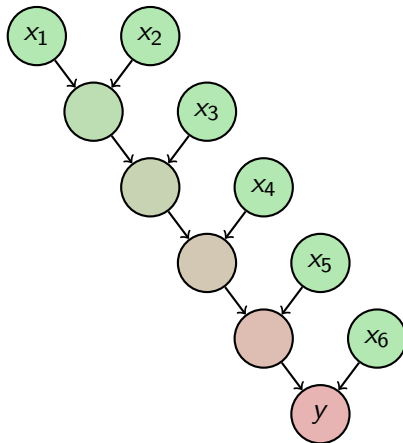
$$y = f(\dots f(f(x_1, x_2), x_3), \dots, x_N)$$

where N is the input size.

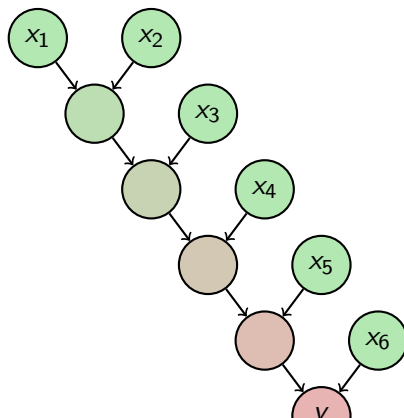
Also known as...

- Python function `reduce`

Reduction: Graph



Reduction: Graph

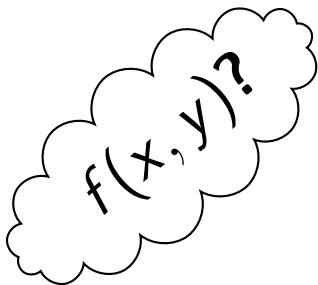


Painful! Not parallelizable.

Approach to Reduction

Can we do better?

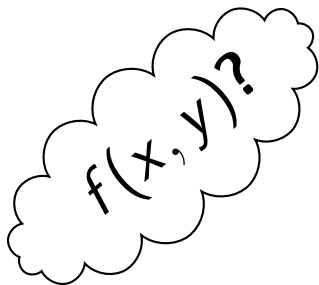
“Tree” very imbalanced. What property of f would allow ‘rebalancing’?



Approach to Reduction

Can we do better?

“Tree” very imbalanced. What property of f would allow ‘rebalancing’?



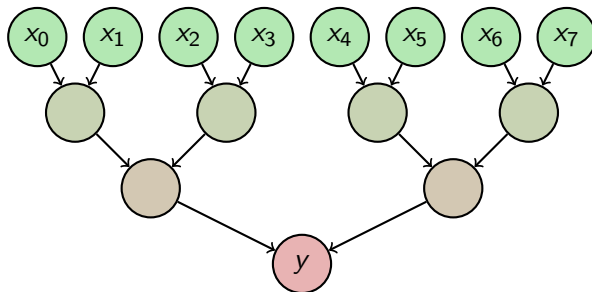
$$f(f(x, y), z) = f(x, f(y, z))$$

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

1 OpenCL

2 PyOpenCL

3 Parallel patterns

- Map

- Reduce

- Scan

Scan

$$y_1 = x_1$$

$$y_2 = f(y_1, x_2)$$

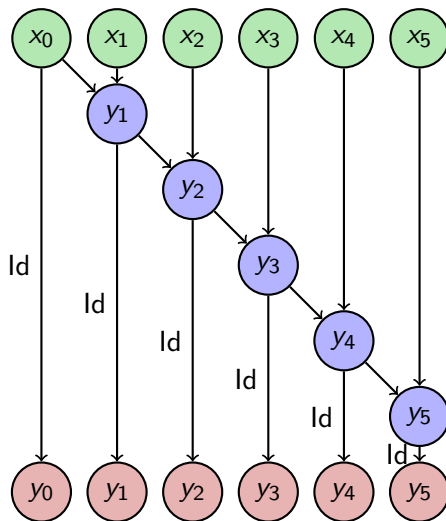
$$\vdots = \vdots$$

$$y_N = f(y_{N-1}, x_N)$$

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

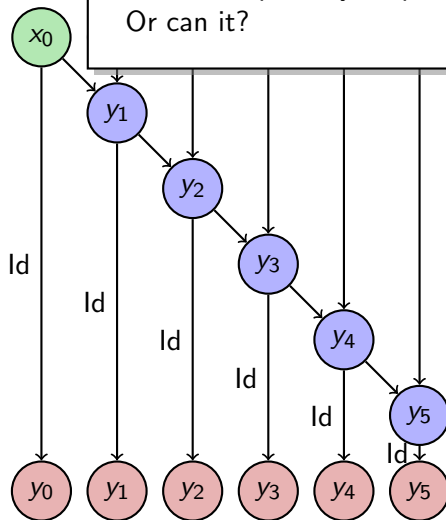
- Prefix Sum/Cumulative Sum
- Abstract view of: loop-carried dependence
- Also possible: Segmented Scan

Scan: Graph

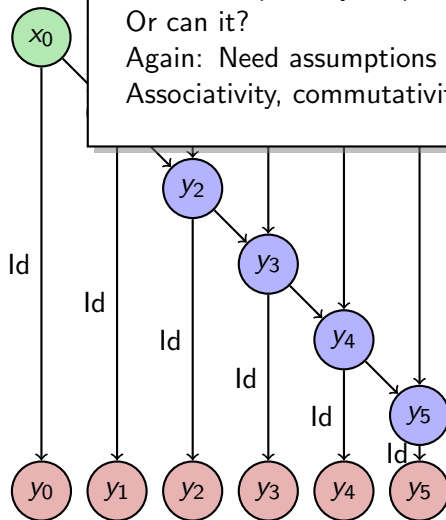


Scan: Graph

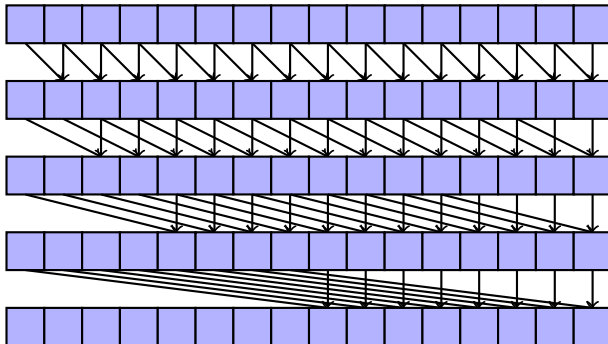
This can't possibly be parallelized.
Or can it?



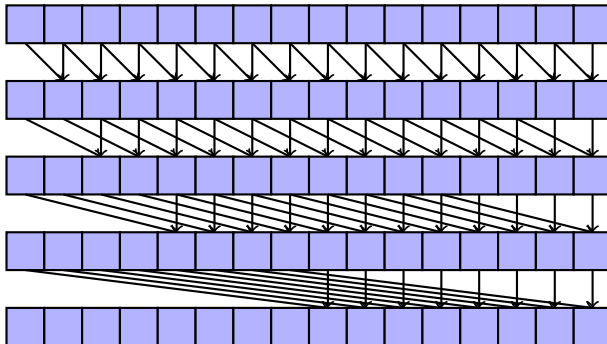
Scan: Graph



Scan: Implementation



Scan: Implementation



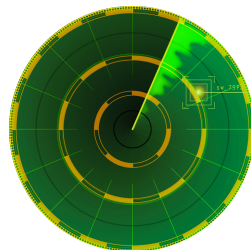
Work-efficient?

Scan: Examples

Low-level building block for many higher-level 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

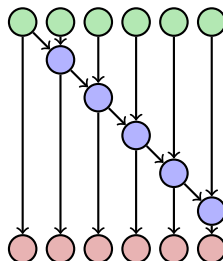


- Subtlety: Inclusive/Exclusive Scan
- Pattern sometimes hard to recognize
 - But shows up surprisingly often
 - Need to prove associativity/commutativity

DEMO TIME

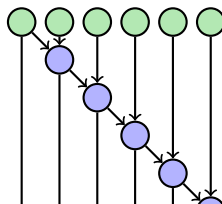
Scan: Features

- “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`.

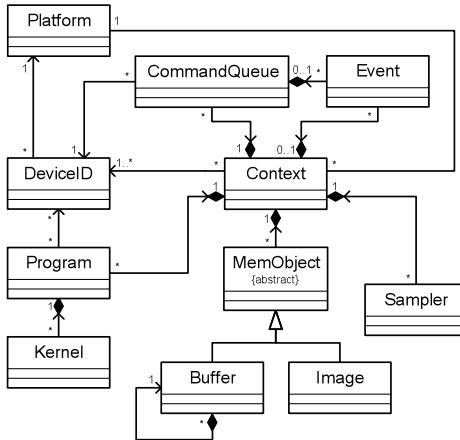
Outline

- 4 OpenCL runtime
 - A Kingdom of Nouns
 - Synchronization
- 5 OpenCL implementations

Outline

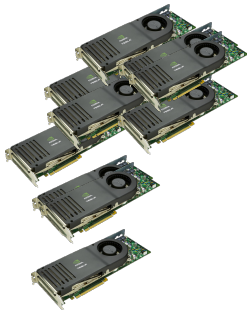
- 4 OpenCL runtime
 - A Kingdom of Nouns
 - Synchronization
- 5 OpenCL implementations

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 → *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

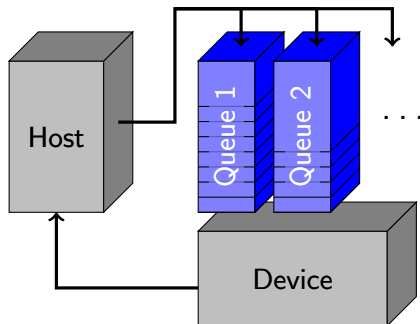
```
context = cl.Context(devices=None | [dev1, dev2], dev_type=None)
context = cl.create_some_context( interactive = True)
```



- 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

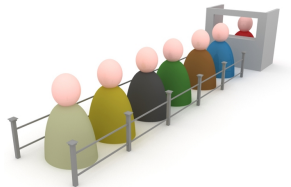
- Host and Device run asynchronously
- Host submits to queue:
 - Computations
 - Memory Transfers
 - Sync primitives
 - ...
- Host can wait for drained queue
- Profiling



Command Queues and Events

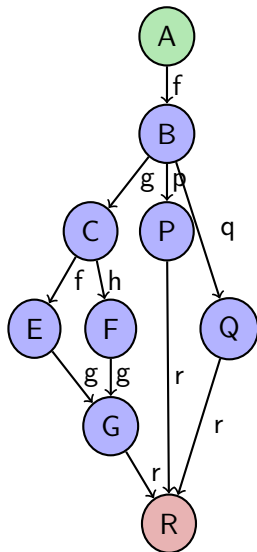
```
queue = cl.CommandQueue(context, device=None,  
    properties=None | [(prop, value ),...])
```

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

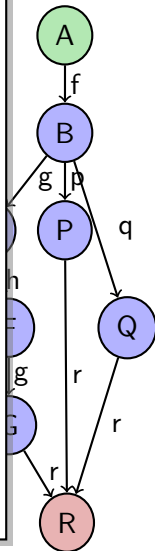


Capturing Dependencies

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



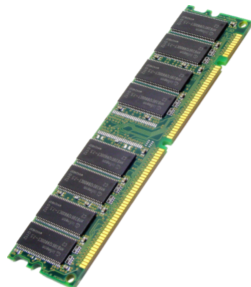
- Switch queue to out-of-order mode!
- Specify as list of events using `wait_for=` optional keyword to `enqueue_XXX`.
- Can also enqueue barrier.
- Common use case:
Transmit/receive from other MPI ranks.
- Possible in hardware on Nv Fermi, AMD Cayman: Submit parallel work to increase machine use.
 - 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:

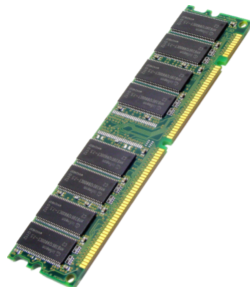
- Use `hostbuf` as initial content of buffer

USE_HOST_PTR:

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

ALLOC_HOST_PTR:

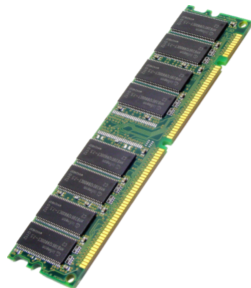
- *New* host memory (unrelated to `hostbuf`) is visible from device *and* host.



Memory Objects: Buffers

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

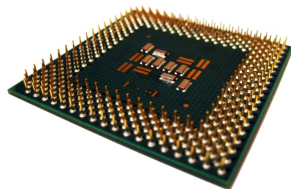
- Specify `hostbuf` or `size` (or both)
- `hostbuf`: Needs Python Buffer Interface
e.g. `numpy.ndarray`, `str`.
 - Important: Memory layout matters
- Passed to device code as pointers
(e.g. `float *`, `int *`)
- `enqueue_copy(queue, dest, src)`
- Can be mapped into host address space:
`cl.MemoryMap`.



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)`

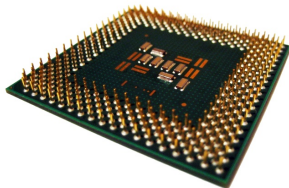


Program Objects

```
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`

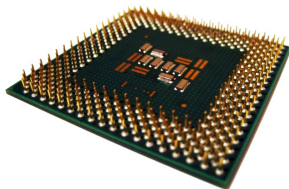


Program Objects

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

Explicitly sized scalars:

✗ Annoying, error-prone.

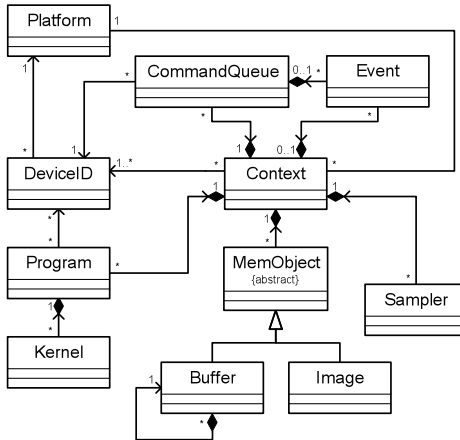


Better:

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

Use None for non-scalars.

OpenCL Object Diagram



Credit: Khronos Group

Outline

- 4 OpenCL runtime
 - A Kingdom of Nouns
 - Synchronization
- 5 OpenCL implementations

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:

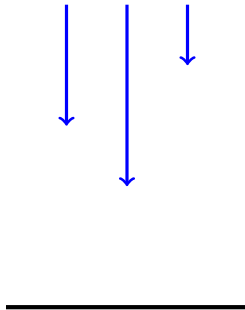
Results of the algorithm must be independent of the order in which work groups are executed.

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

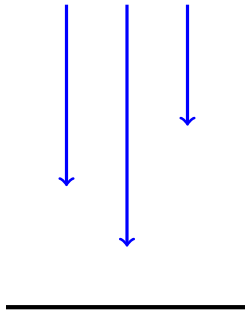
Synchronization

What is a Barrier?



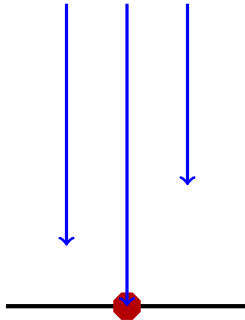
Synchronization

What is a Barrier?



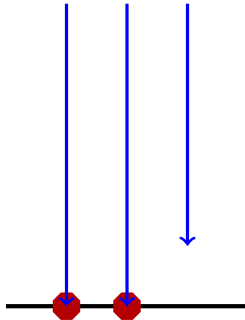
Synchronization

What is a Barrier?



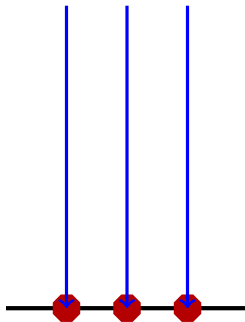
Synchronization

What is a Barrier?



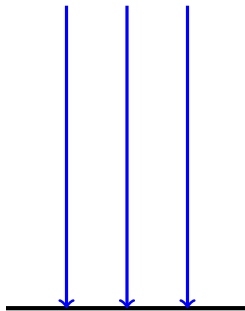
Synchronization

What is a Barrier?



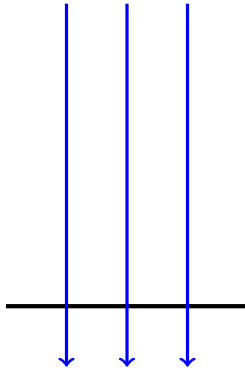
Synchronization

What is a Barrier?



Synchronization

What is a Barrier?

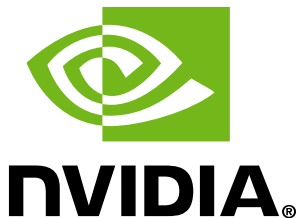


Outline

4 OpenCL runtime

5 OpenCL implementations

The Nvidia CL implementation



Targets only GPUs

Notes:

- Nearly identical to CUDA
 - No native C-level JIT in CUDA (→ 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
 - → 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: $SIMD_m VLIW_n$
- Lots of fixed-function hardware
- Last-level Cache (LLC) integrated between CPU and GPU

