Scripting GPUs with PyOpenCL

Andreas Klöckner

Division of Applied Mathematics
Brown University

October 8, 2009

Thanks

- Tim Warburton (Rice)
- Jan Hesthaven (Brown)
- David Garcia
- Nicolas Pinto (MIT)
- PyOpenCL, PyCUDA contributors
- Nvidia Corporation

Outline

BROWN

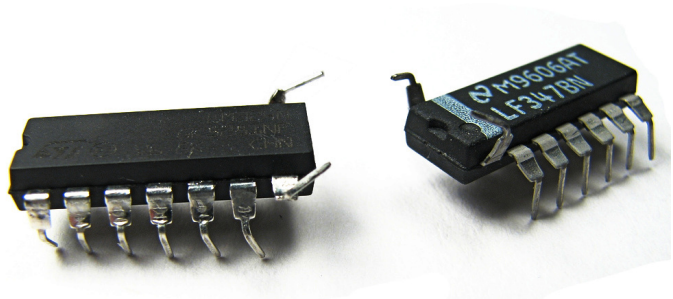
Outline

Outline

BROWN

GPU Computing?

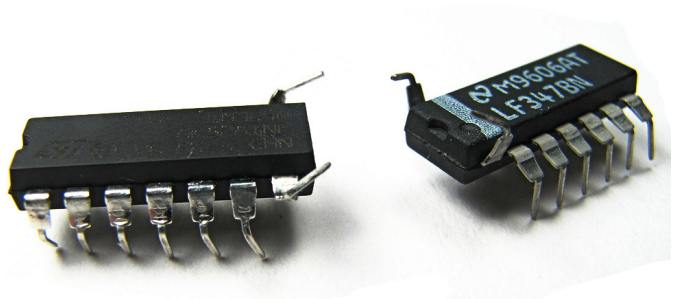
- Design target for CPUs:
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate



BROWN

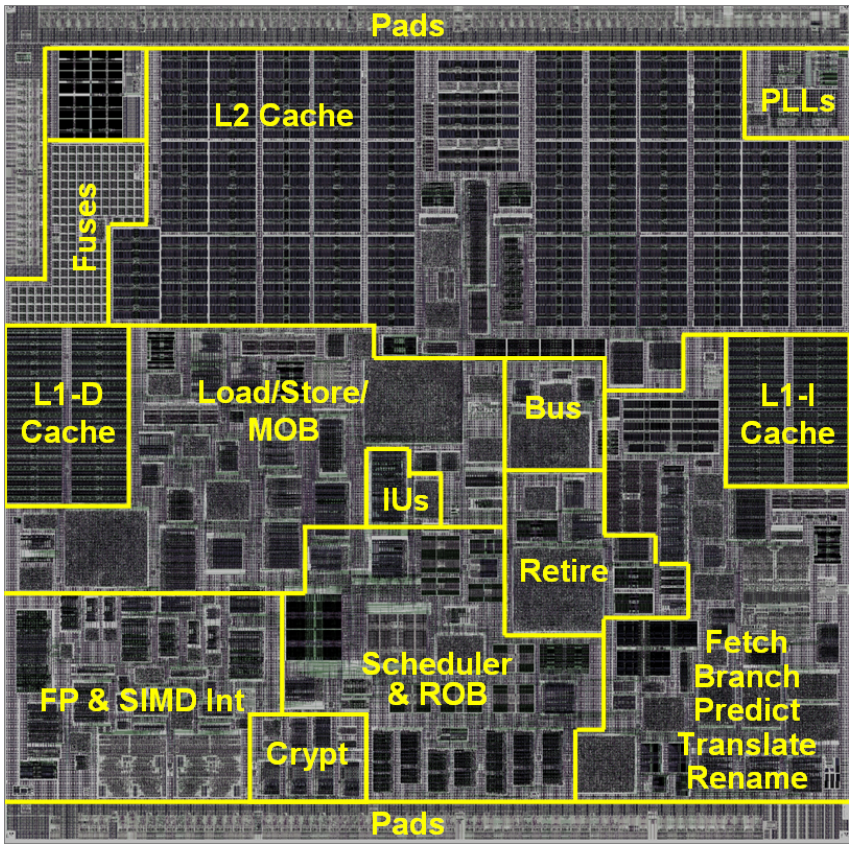
GPU Computing?

- Design target for CPUs:
  - Make a single thread very fast
  - Hide latency through large caches
  - Predict, speculate
- GPU Computing takes a different approach:
  - Throughput matters—single threads do not
  - Hide latency through parallelism
  - Let programmer deal with "raw" storage hierarchy



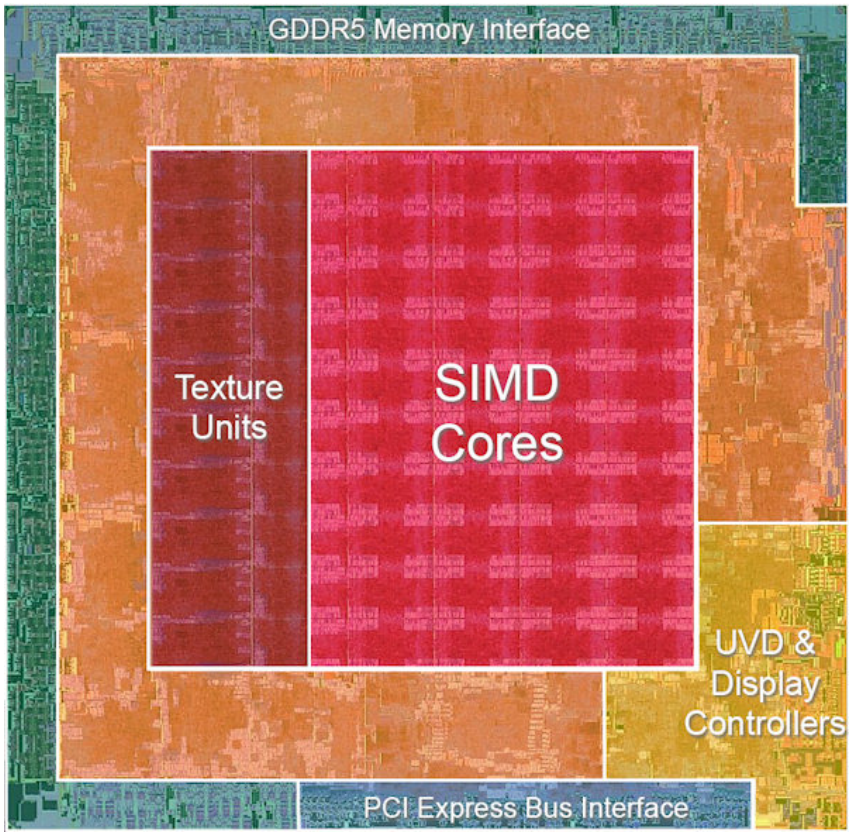
BROWN

CPU Chip Real Estate



Die floorplan: VIA Isaiah (2008).
65 nm, 4 SP ops at a time, 1 MiB L2.

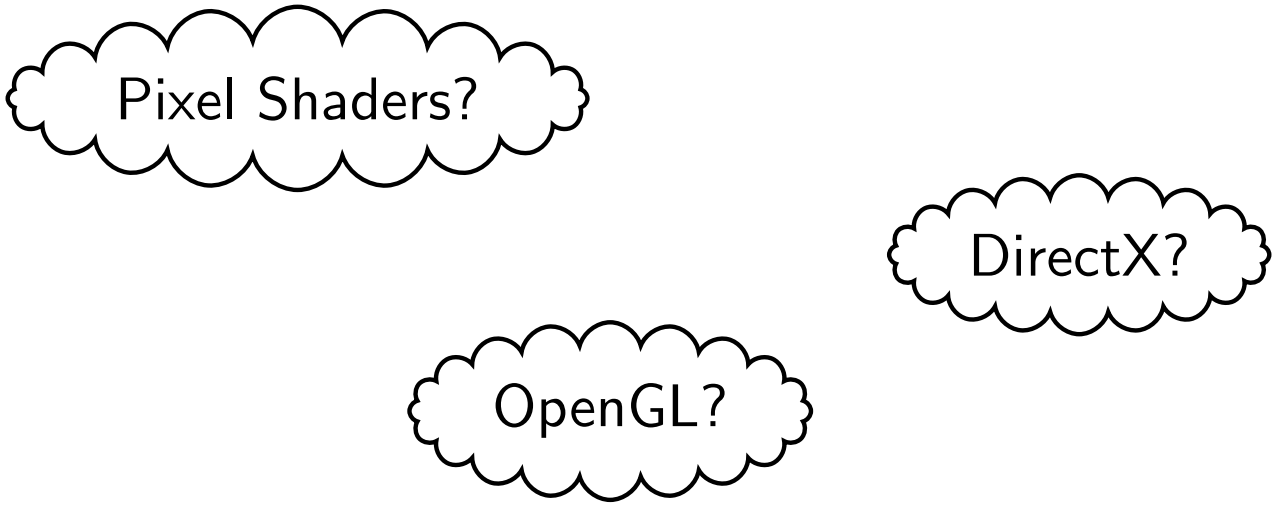
GPU Chip Real Estate



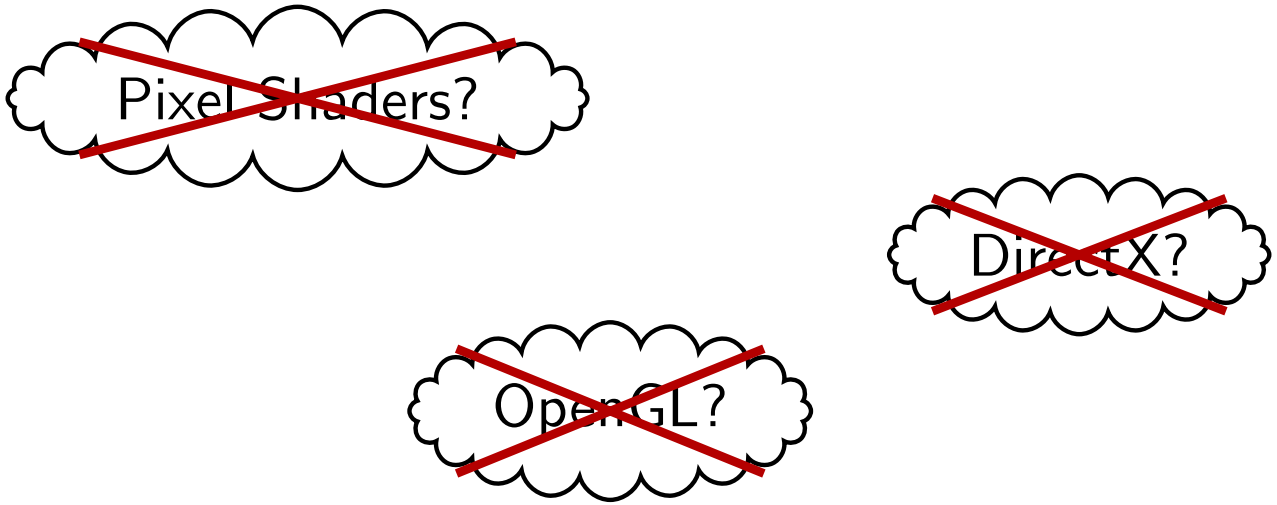
Die floorplan: AMD RV770 (2008).
55 nm, 800 SP ops at a time.

Outline

Programming Models

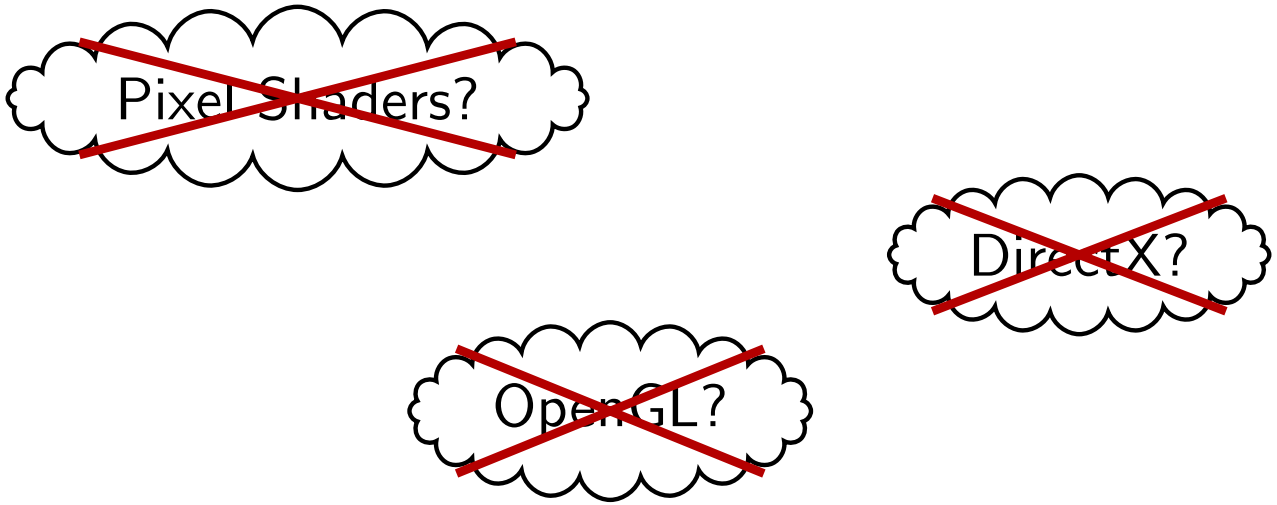


Programming Models



BROWN

Programming Models



- OpenCL: Dedicated Programming Interface.
- Not much "graphicsy" stuff visible

The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

BROWN

The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

Devices differ by

- Memory Types, Latencies, Bandwidths
- Vector Widths
- Units of Scheduling

BROWN

The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

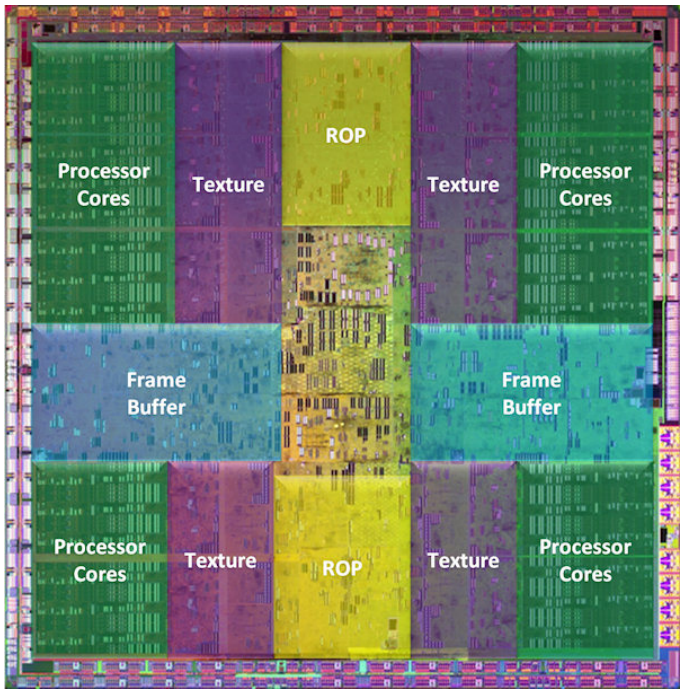
Devices differ by

- Memory Types, Latencies, Bandwidths
- Vector Widths
- Units of Scheduling

One code will not transparently run well on all of them!

BROWN

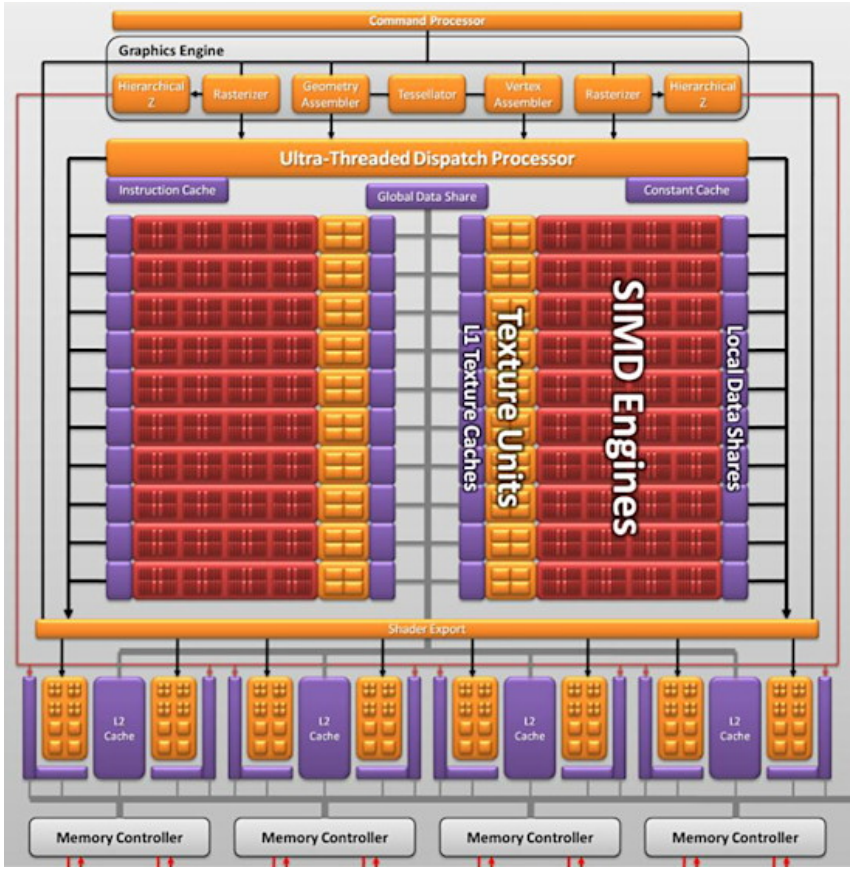
GPU Architecture (e.g. Nvidia GT200)



- 1 GPU = 30 SIMDs
- 1 SIMD: 32×32 PCs, HW Sched + 1 ID (1/4 clock) + 8 SP + 1 DP + 16 KiB Shared + 32 KiB Reg
- Device $\leftrightarrow$ RAM: **140 GB/s**
- Device $\leftrightarrow$ Host: **6 GB/s**
- User manages memory hierarchy

BROWN

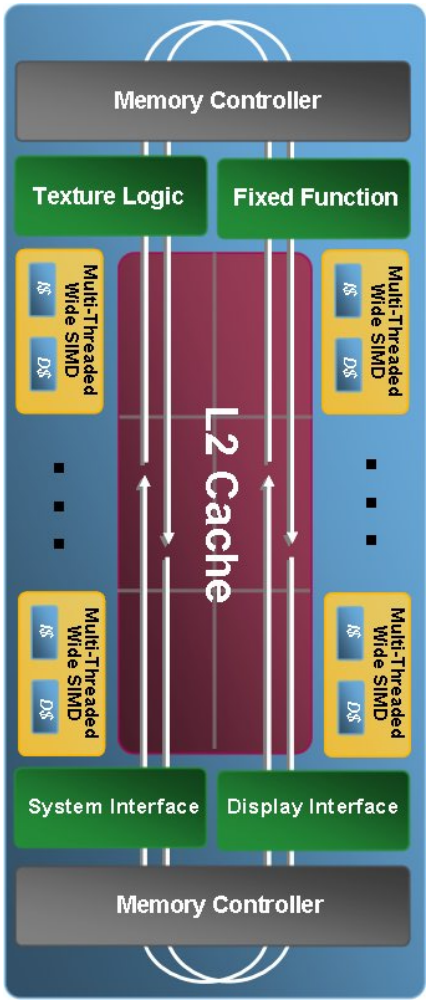
GPU Architecture (e.g. ATI RV870)



- 1 GPU = 20 SIMDs
  + 64 KiB Global Share
  + 4 × 128 KiB L2
- 1 SIMD = 1 ID + 16×5 SP
  + 16 DP + 32 KiB Share
  + HW Sched + 8 KiB L1
- GDDR5 RAM (150 GB/s)
- PCIe2 Host DMA (6 GB/s)

BROWN

Intel Larabee: Architecture



- Unreleased (2010?)
- x86-64 + SSE + "vector-complete" 512-bit ISA ("LRBni")
- 4x "Hyperthreading"
- 32 (?) cores per chip
- "Fiber/Strand" software threads
- Recursive Launches
- Coherent Caches (w/ explicit control)
- Performance?

BROWN

Gains and Losses

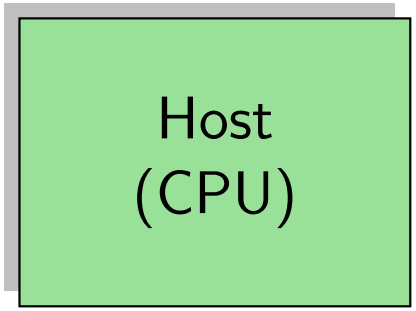
Gains	Losses
⊕ Memory Bandwidth (140 GB/s vs. 12 GB/s) ⊕ Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	

Gains and Losses

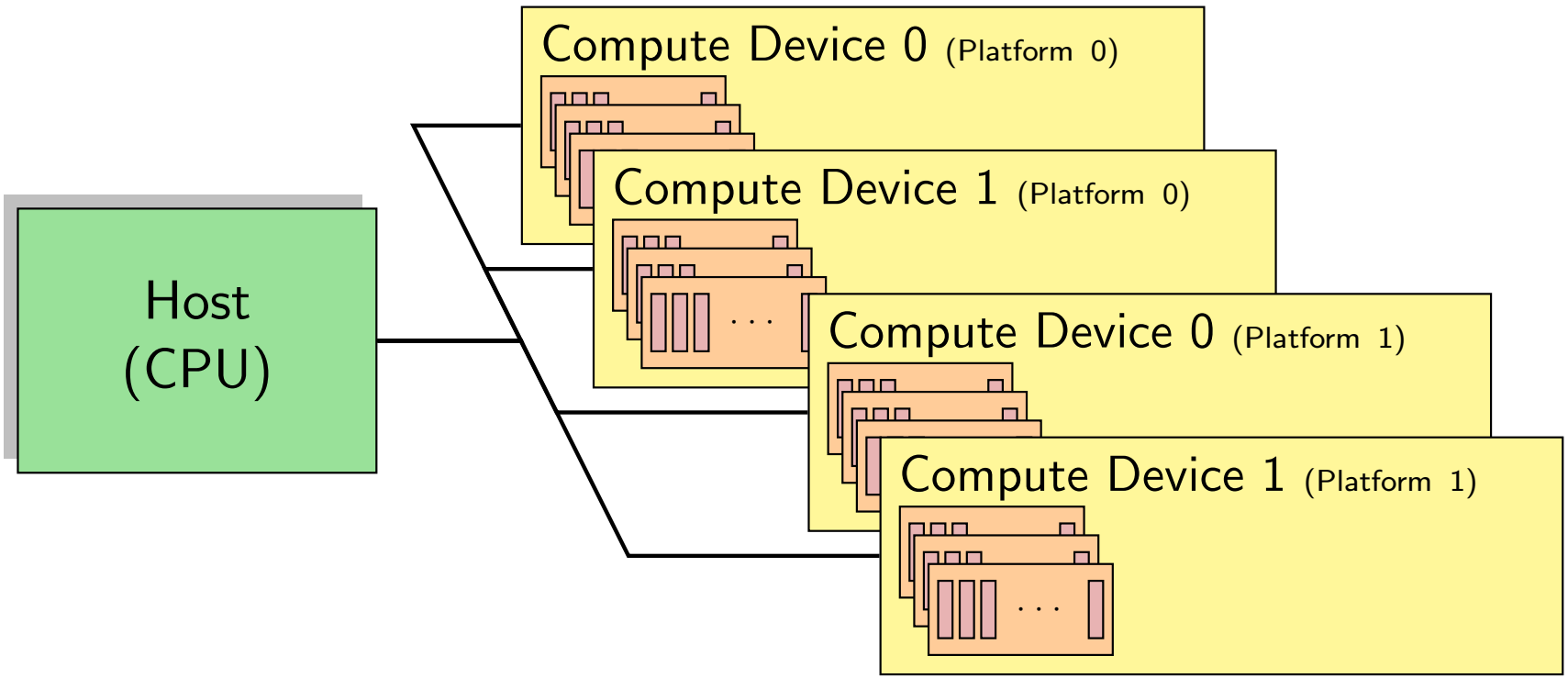
Gains	Losses
⊕ Memory Bandwidth (140 GB/s vs. 12 GB/s)	⊖ Cheap branches (i.e. `ifs`)
⊕ Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)	⊖ "Free" double precision
	⊖ Exceptions
	⊖ `new`/`delete`
	⊖ Recursion
	⊖ Function pointers
	⊖ IEEE 754 FP compliance

Outline

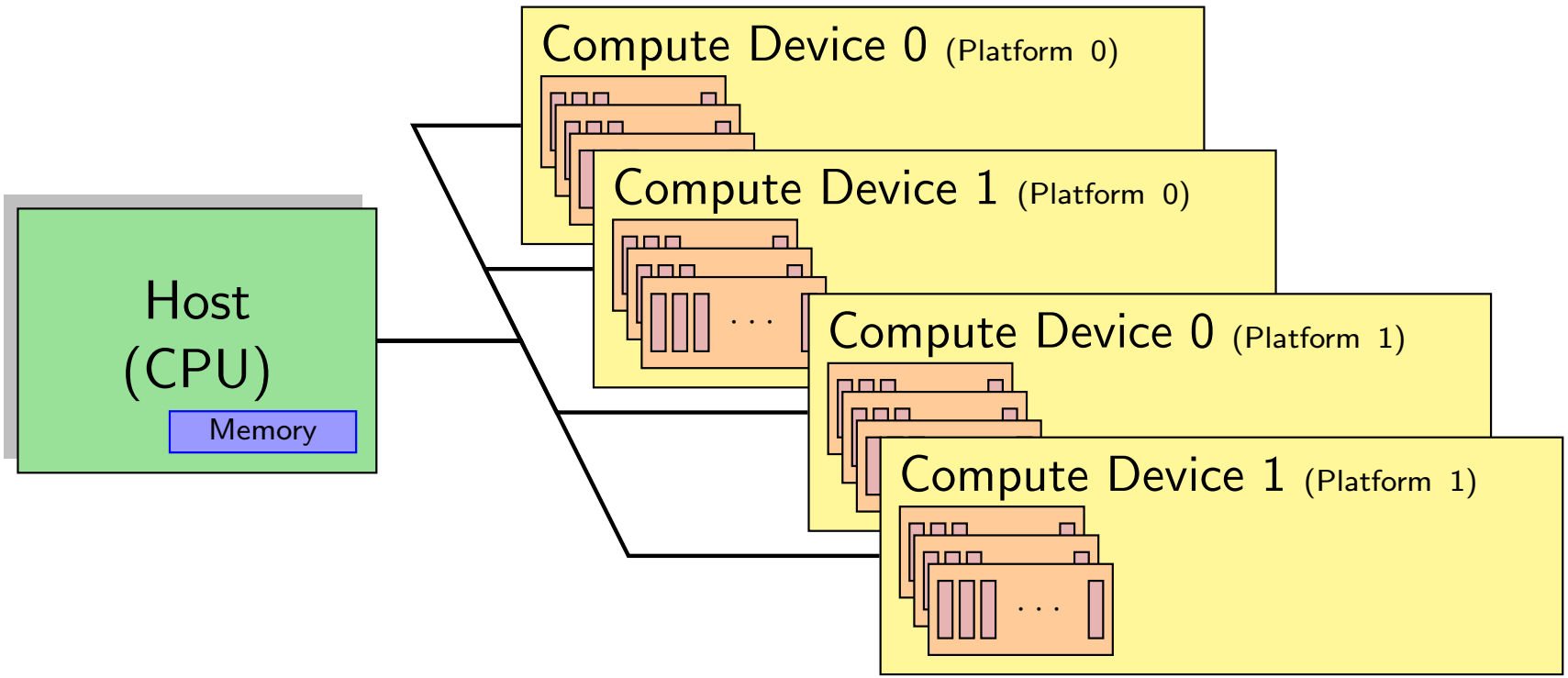
OpenCL: Computing as a Service



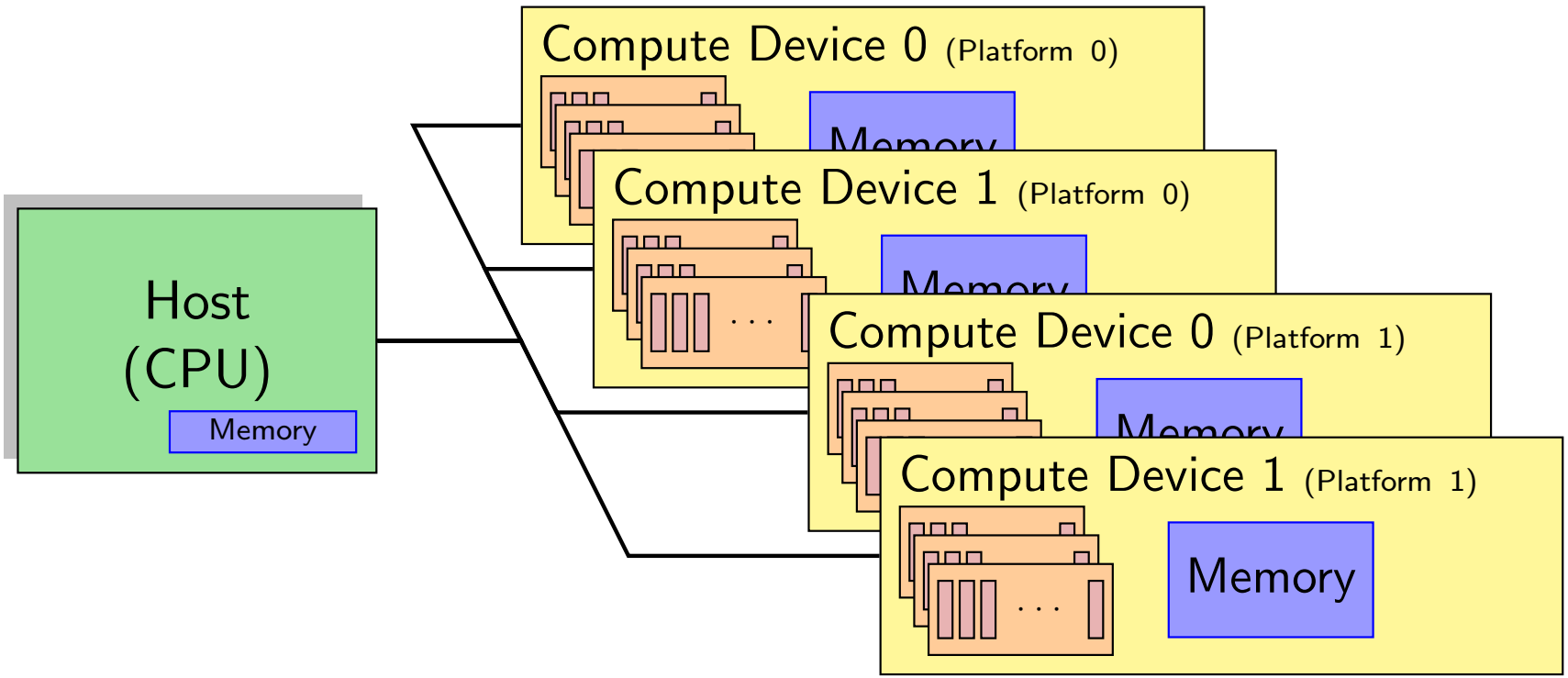
OpenCL: Computing as a Service



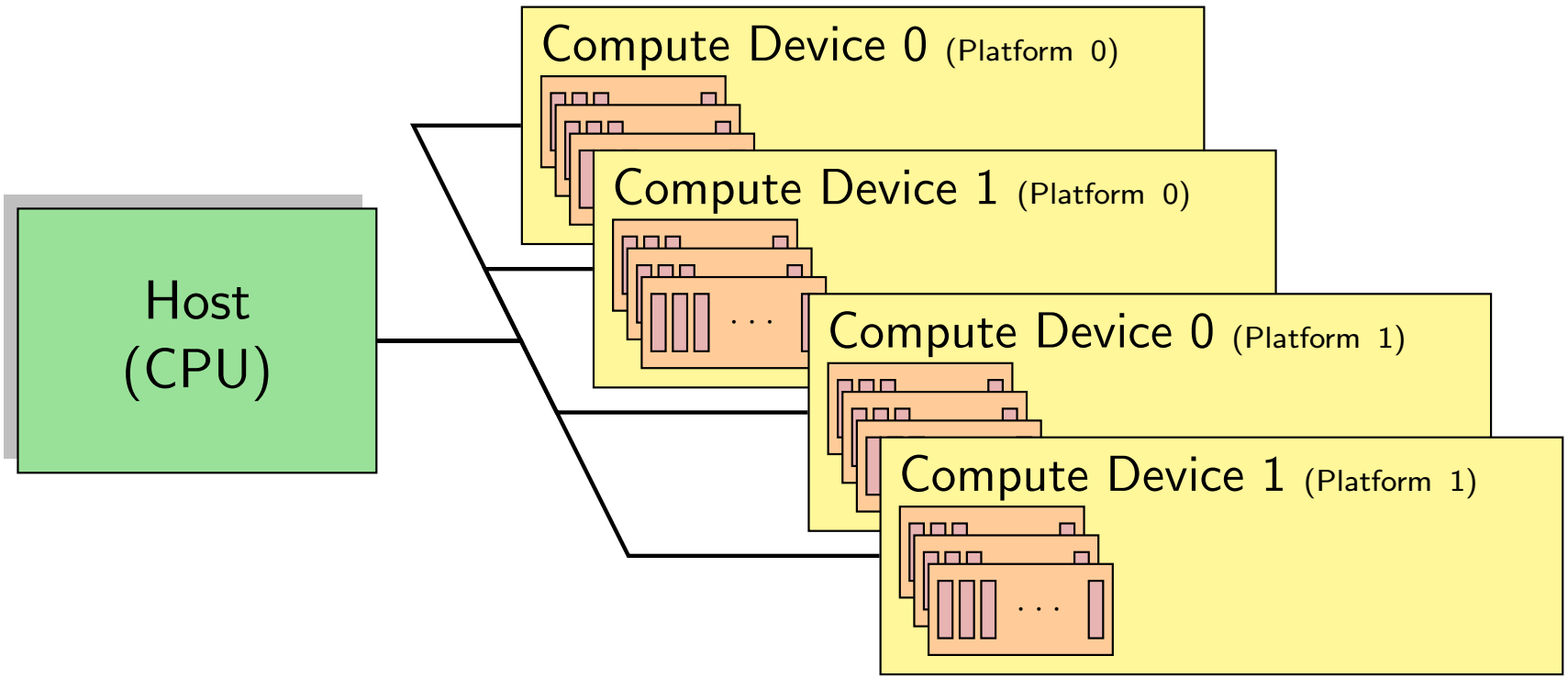
OpenCL: Computing as a Service



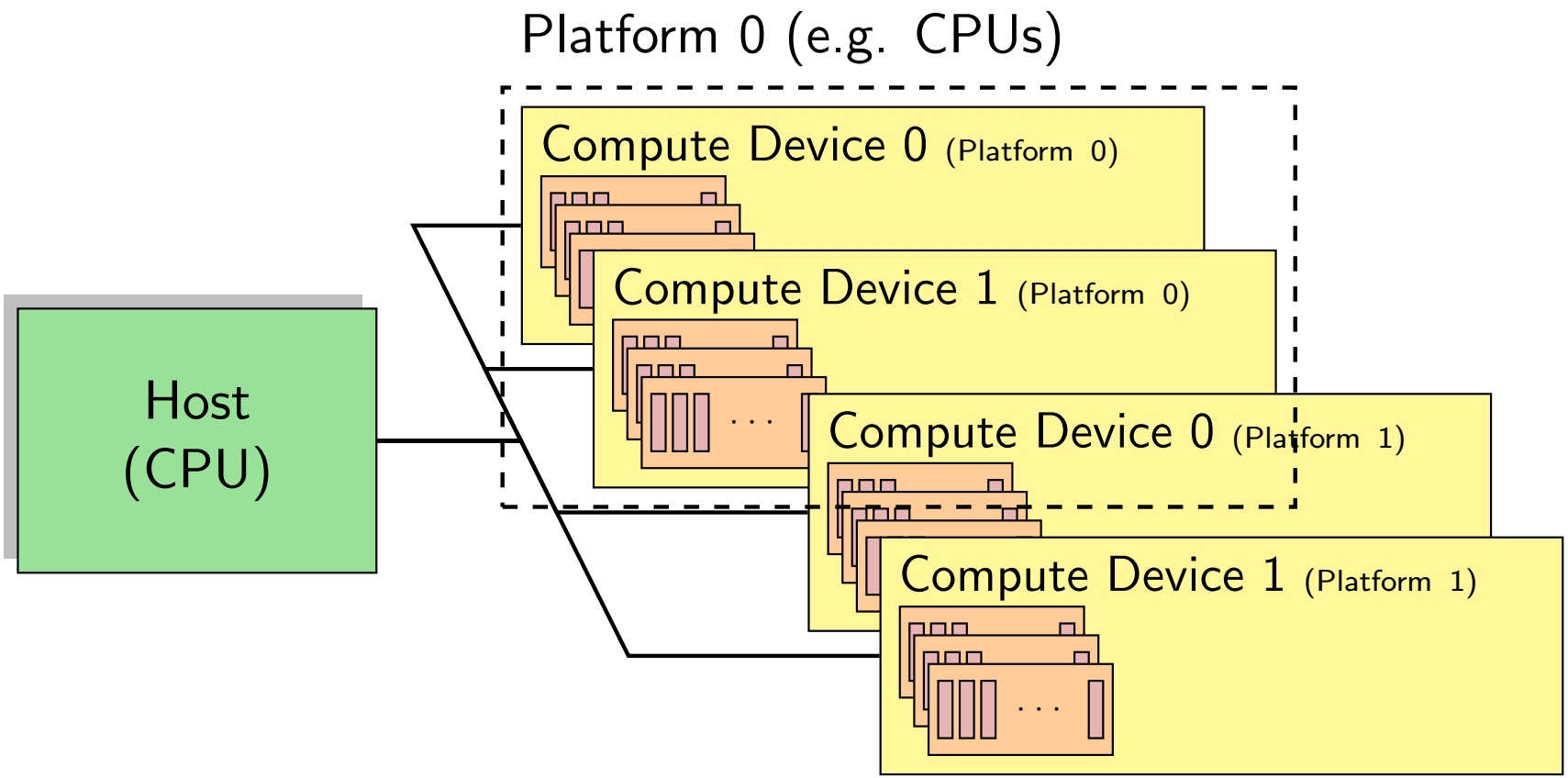
OpenCL: Computing as a Service



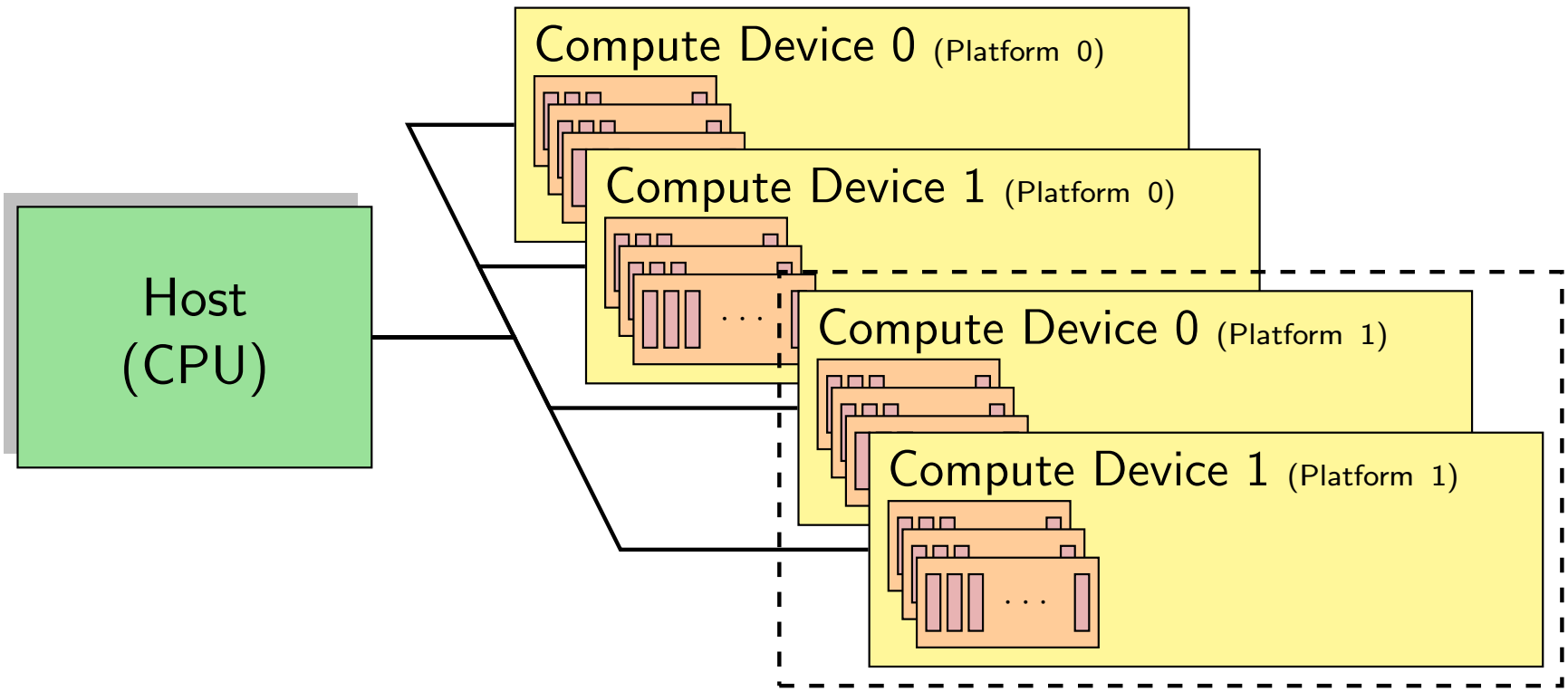
OpenCL: Computing as a Service



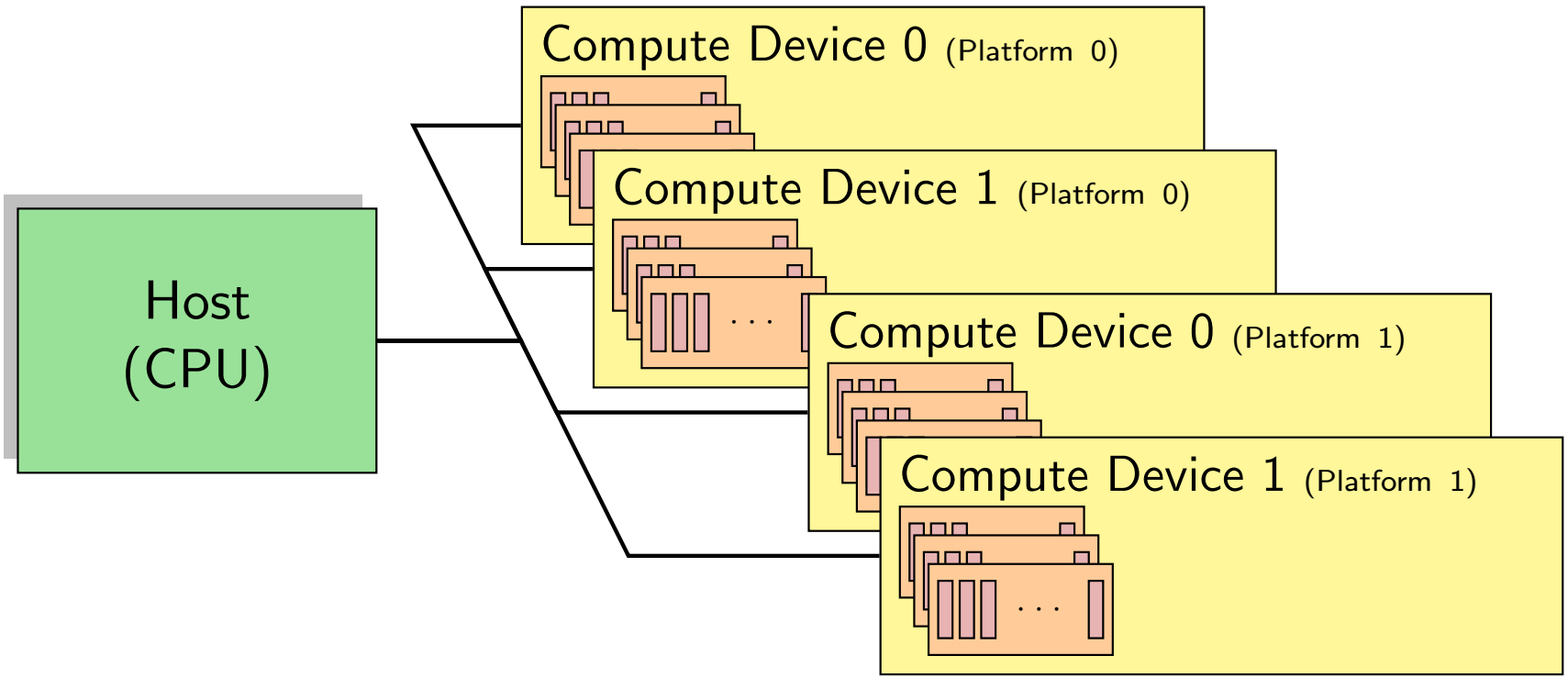
OpenCL: Computing as a Service



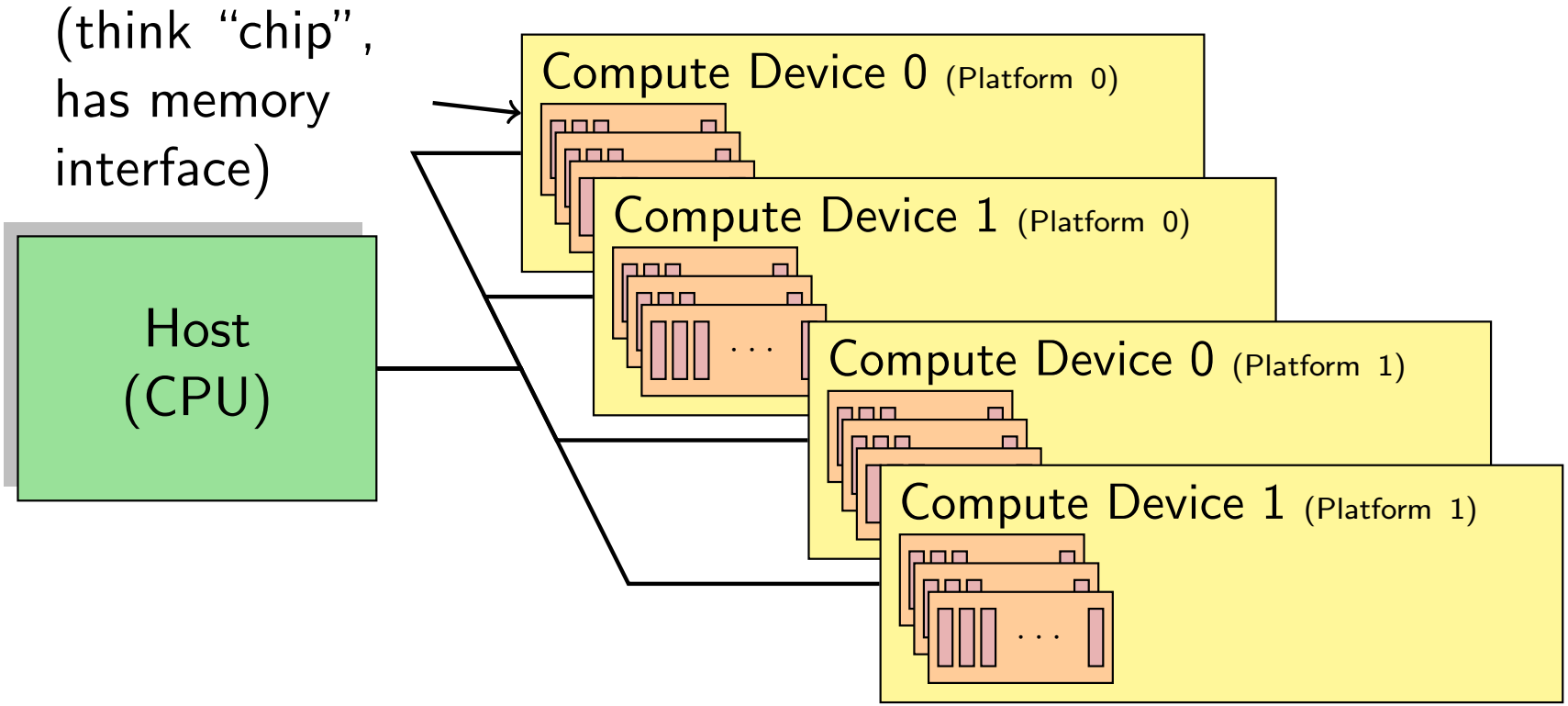
OpenCL: Computing as a Service



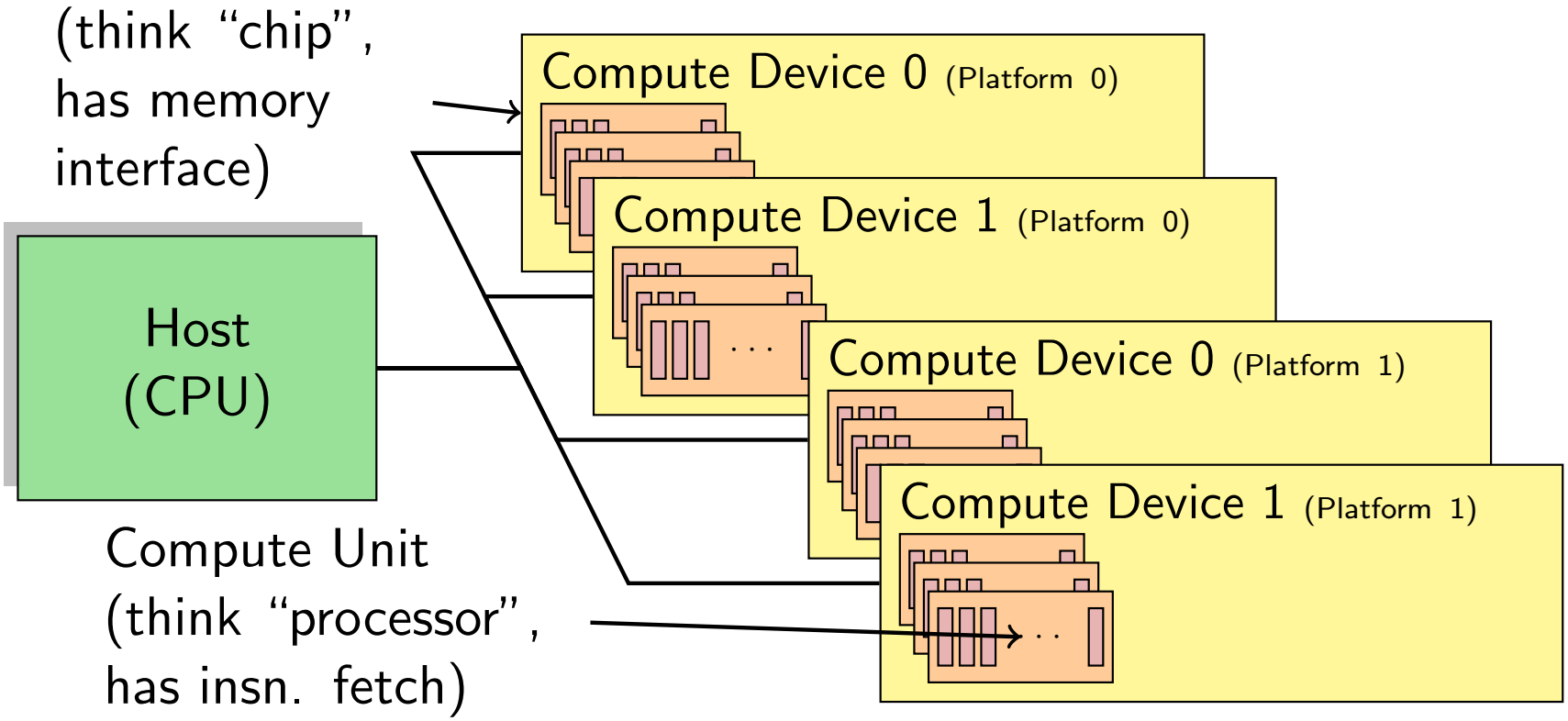
OpenCL: Computing as a Service



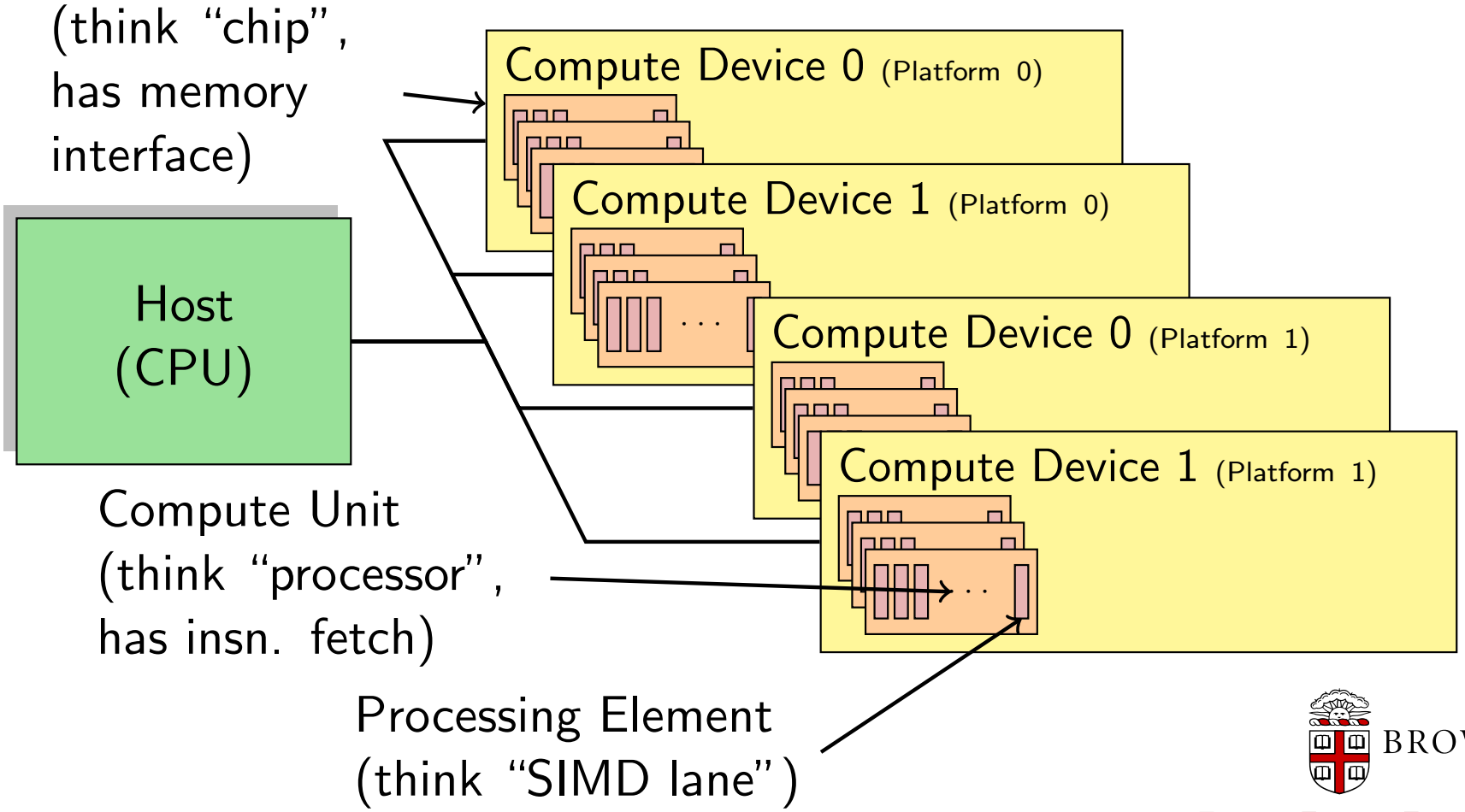
OpenCL: Computing as a Service



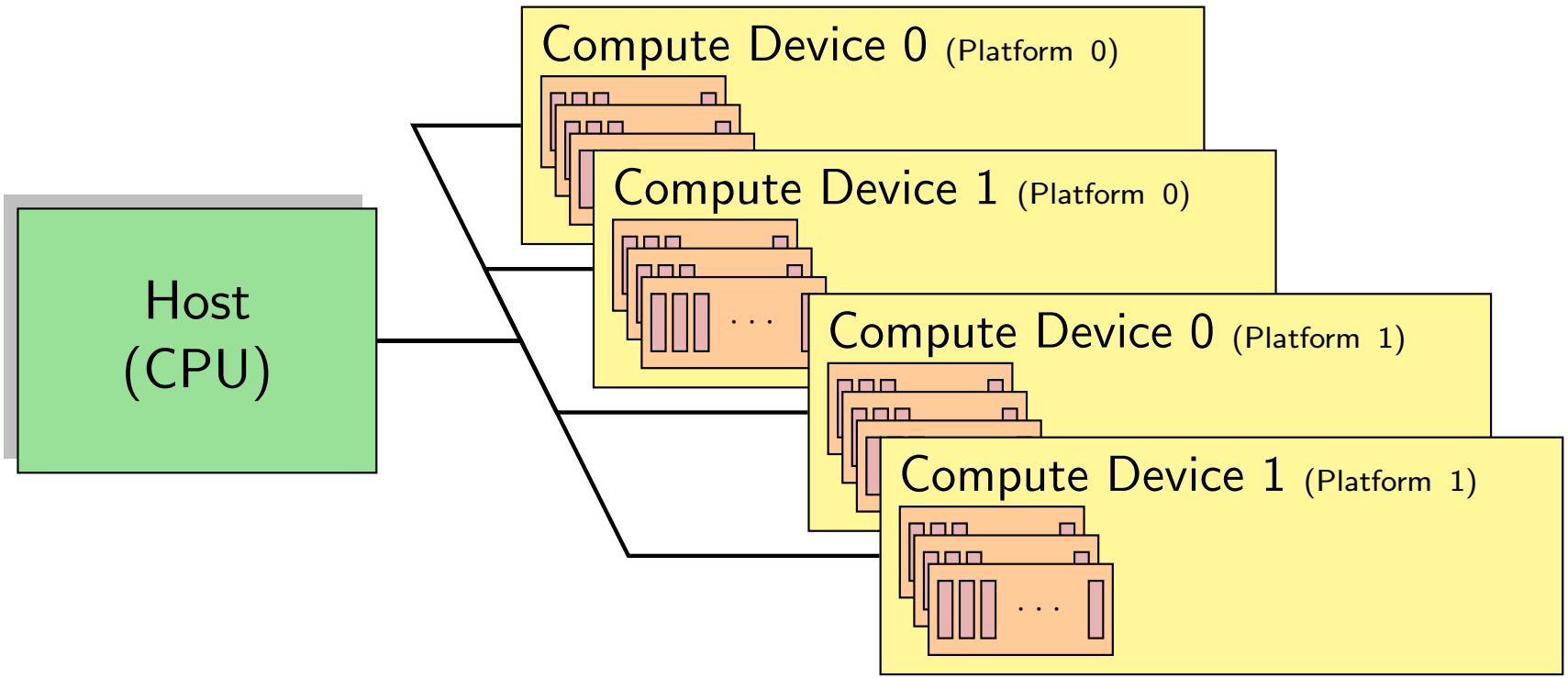
OpenCL: Computing as a Service



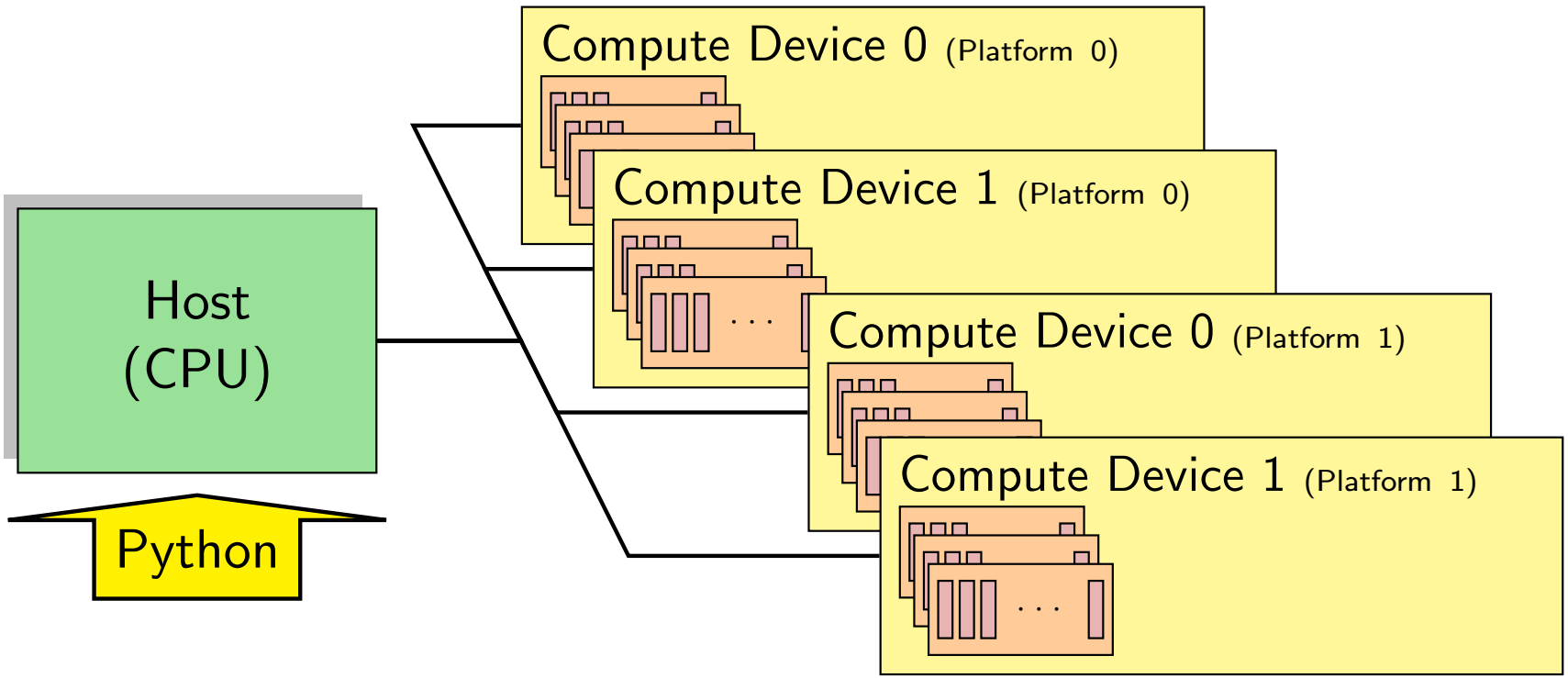
OpenCL: Computing as a Service



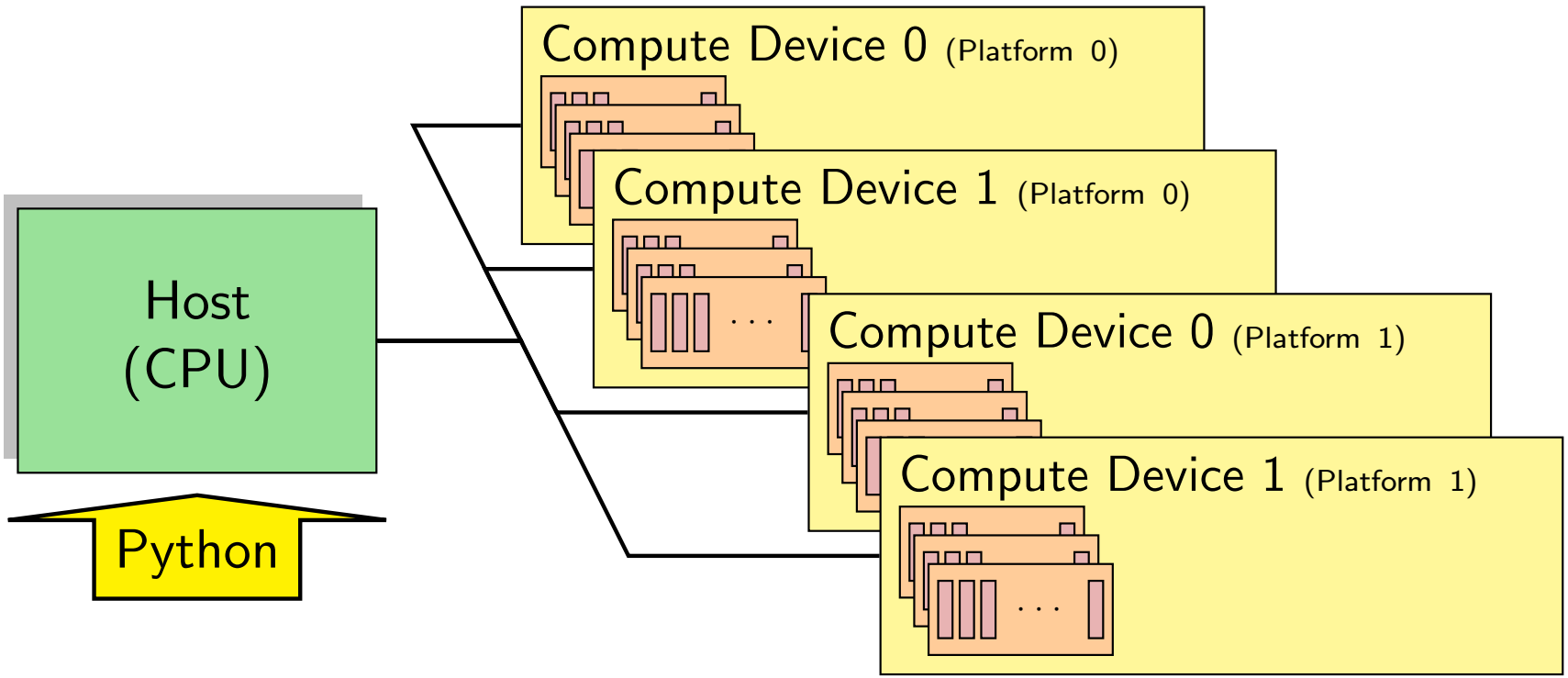
OpenCL: Computing as a Service



OpenCL: Computing as a Service

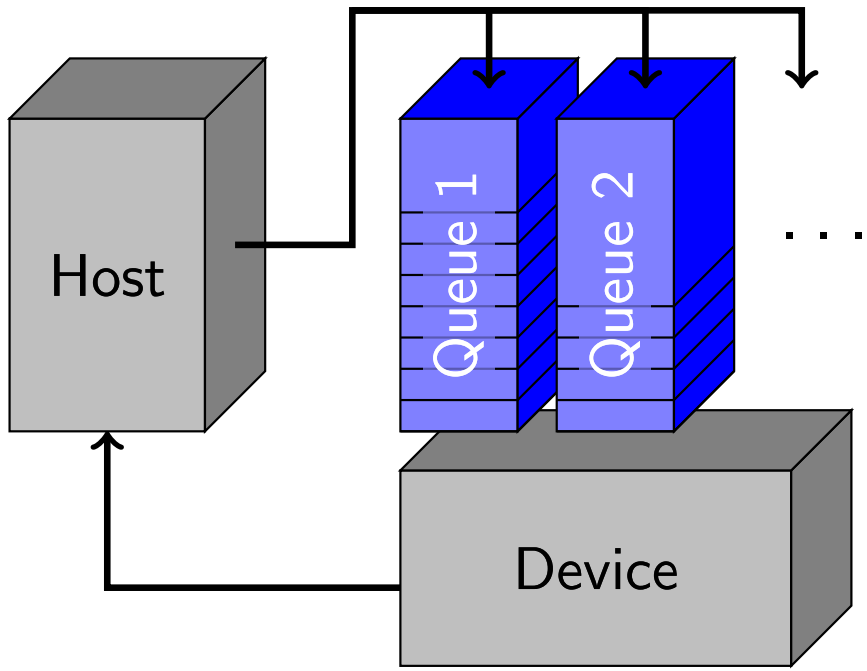


OpenCL: Computing as a Service



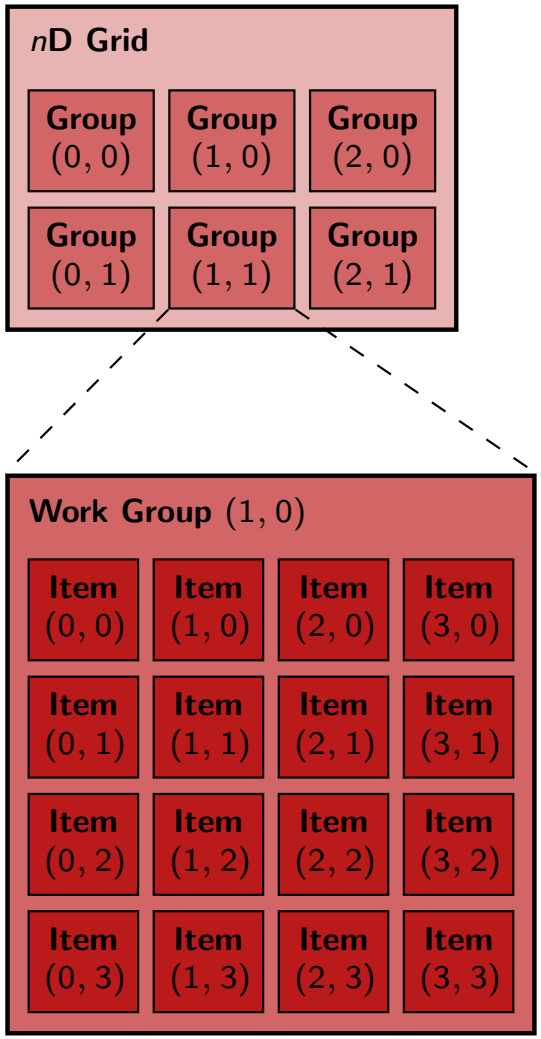
OpenCL: Command Queues

- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for drained queue
- Multiple Queues: Can overlap Compute + Transfer



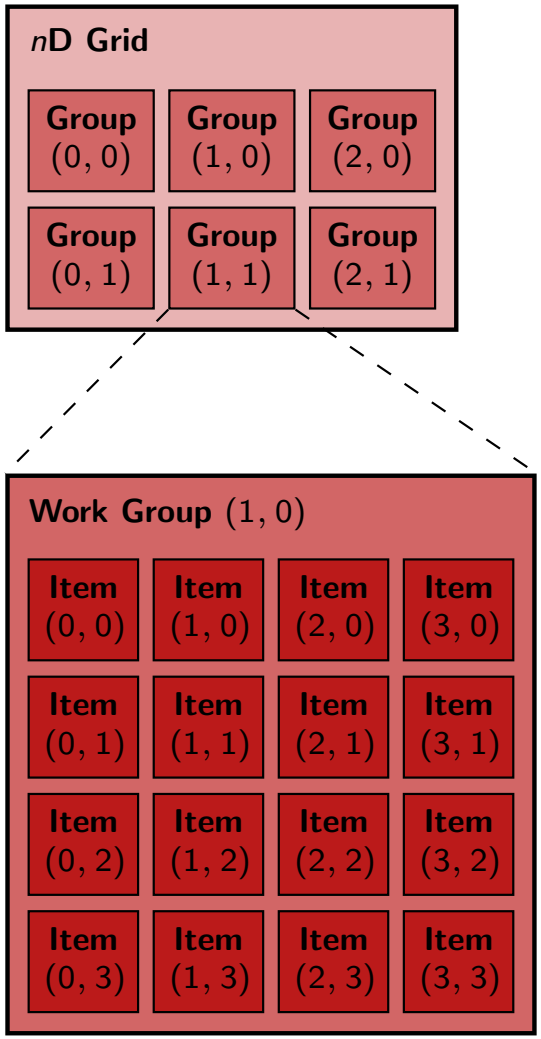
BROWN

OpenCL: Execution Model



- Two-tiered Parallelism
  - Grid $= N_x \times N_y \times N_z$ work groups
  - Work group $= S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items

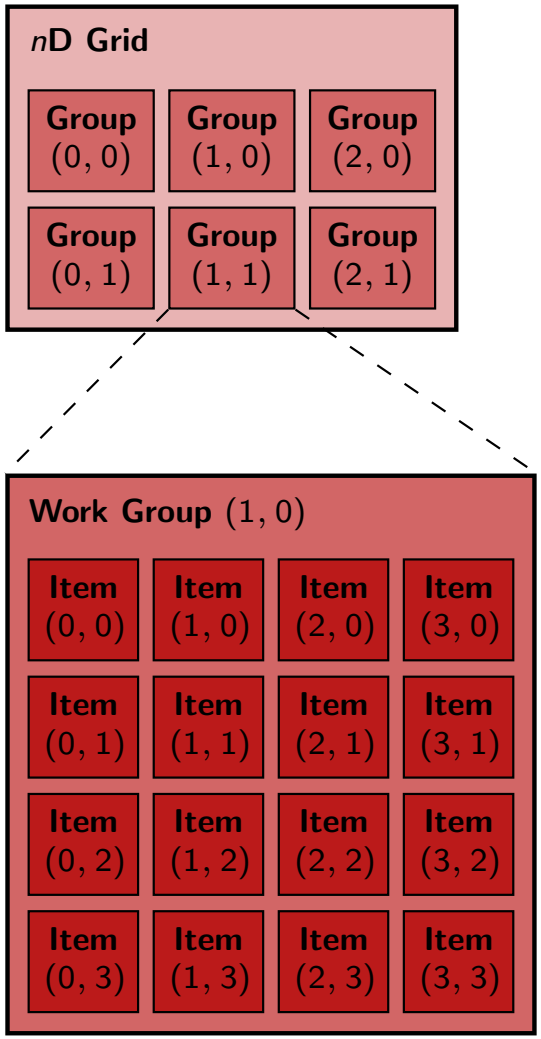
OpenCL: Execution Model



- Two-tiered Parallelism
  - Grid $= N_x \times N_y \times N_z$ work groups
  - Work group $= S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items
- Comm/Sync only within work group
  - Work group maps to compute unit

BROWN

OpenCL: Execution Model



- Two-tiered Parallelism
  - Grid $= N_x \times N_y \times N_z$ work groups
  - Work group $= S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items
- Comm/Sync only within work group
  - Work group maps to compute unit
- Grid/Group $\approx$ outer loops in an algorithm
- Device Language:
  `get_{global,group,local}_{id,size}(axis)`

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
$\rightarrow$ "Work Item" $\neq$ "Thread"!

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
→ "Work Item" ≠ "Thread"!

```
if (get_global_id(0) % 2 == 0)
  do_something();
else
  do_another_thing();
do_the_rest();
```

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
→ "Work Item" ≠ "Thread"!

```
if (get_global_id(0) % 2 == 0)
  do_something();
else
  do_another_thing();
do_the_rest();
```

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
→ "Work Item" ≠ "Thread"!

```
if (get_global_id(0) % 2 == 0)
  do_something();
else
  do_another_thing();
do_the_rest();
```

Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
→ "Work Item" ≠ "Thread"!

```
if (get_global_id(0) % 2 == 0)
  do_something();
else
  do_another_thing();
do_the_rest();
```

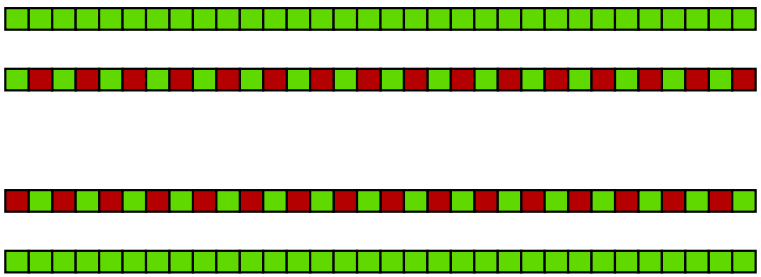
Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

- Explicit: Use (e.g.) `float2`, ..., `float16`.
- Implicit: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software)

Implicit SIMD: Groups of work items are scheduled together.
→ "Work Item" ≠ "Thread"!



```
if (get_global_id(0) % 2 == 0)
  do_something();
else
  do_another_thing();
do_the_rest();
```

Questions?

?

Outline

BROWN

Outline

Why do Scripting for OpenCL?

- Compute Devices are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

$\longrightarrow$ complement each other

BROWN

Why do Scripting for OpenCL?

- Compute Devices are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\longrightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough

BROWN

Why do Scripting for OpenCL?

- Compute Devices are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput

  $\longrightarrow$ complement each other
- CPU: largely restricted to control tasks ($\sim$1000/sec)
  - Scripting fast enough
- Python + OpenCL = **PyOpenCL**

BROWN

Whetting your Appetite

```
2  import pyopencl as cl
3  import numpy, numpy.linalg as la
4
5  a = numpy.random.rand(128*128).astype(numpy.float32)
6
7  ctx = cl.Context()
8  queue = cl.CommandQueue(ctx)
9
10 mf = cl.mem_flags
11 a_buf = cl.Buffer(ctx,
12         mf.READ_WRITE | mf.COPY_HOST_PTR,
13         hostbuf=a)
```

Whetting your Appetite

```
17  prg = cl.Program(ctx, """
18      __kernel void twice(__global float *a)
19      {
20        a[get_global_id(0)] *= 2;
21      }
22      """).build()
23
24  prg.twice(queue, a.shape, a_buf, local_size=(128,))
25
26  a_twice = numpy.empty_like(a)
27  cl.enqueue_read_buffer(queue, a_buf, a_twice).wait()
28
29  assert la.norm(a_twice-2*a) == 0
```

Whetting your Appetite

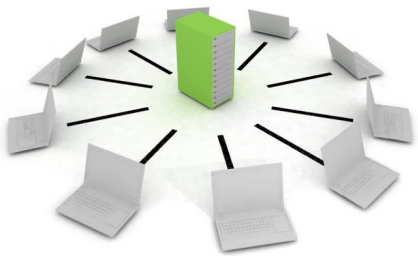
```
17  prg = cl.Program(ctx, """
18      __kernel void twice(__global float *a)
19      {
20        a[get_global_id(0)] *= 2;
21      }
22      """).build()
23
24  prg.twice(queue, a.shape, a_buf, local_size=(128,))
25
26  a_twice = numpy.empty_like(a)
27  cl.enqueue_read_buffer(queue, a_buf, a_twice).wait()
28
29  assert la.norm(a_twice-2*a) == 0
```

Compute kernel

Outline

Contexts

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

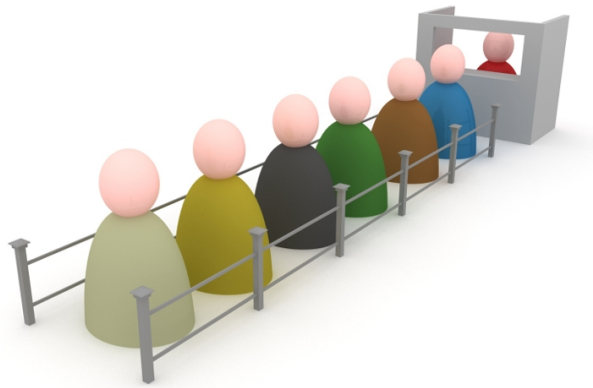


- Spans one or more Devices
- Create from device type or list of devices
  - See docs for `cl.Platform`, `cl.Device`
- `dev_type`: *`DEFAULT`*, `ALL`, `CPU`, `GPU`
- Needed to...
  - ...allocate Memory Objects
  - ...create and build Programs
  - ...host Command Queues
  - ...execute Grids

Command Queues and Events

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

- Attached to single device
- `event = enqueue_XXX(queue, ..., wait_for=[evt1, evt2])`
- `event.wait()`
- Command in queue implicitly waits for previous command's completion

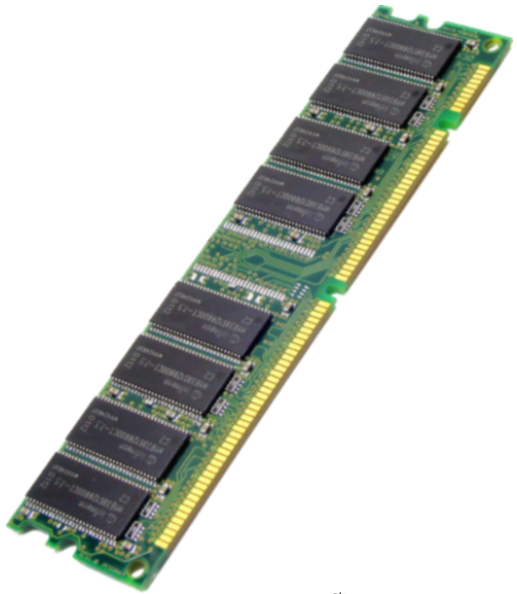


BROWN

Memory Objects: Buffers

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

- Chunk of device memory
- No type information: "Bag of bytes"
- Specify `hostbuf` or `size` (or both)
- `hostbuf`: Needs Python Buffer Interface
  e.g. `numpy.ndarray`, `str`.
- `flags`:
  - `READ_ONLY/WRITE_ONLY/READ_WRITE`
  - `{ALLOC,COPY,USE}_HOST_PTR`

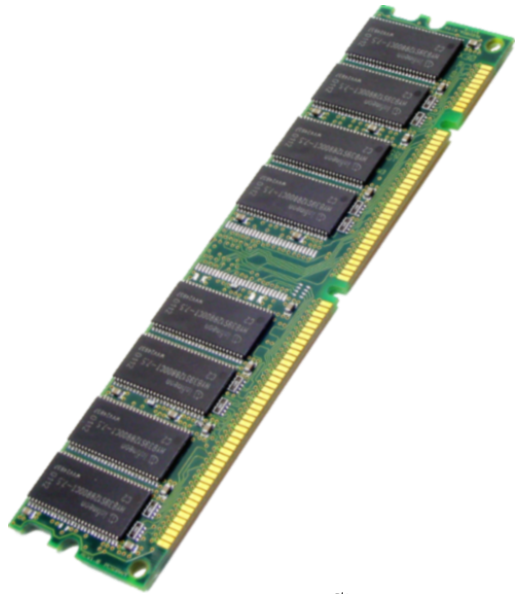


BROWN

Memory Objects: Buffers

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

- Passed to device code as pointers
  (e.g. `float *`, `int *`)
- `enqueue_{read,write}_buffer(`
  `queue, buf, hostbuf)`
- Can be mapped into host address space:
  `cl.MemoryMap`.

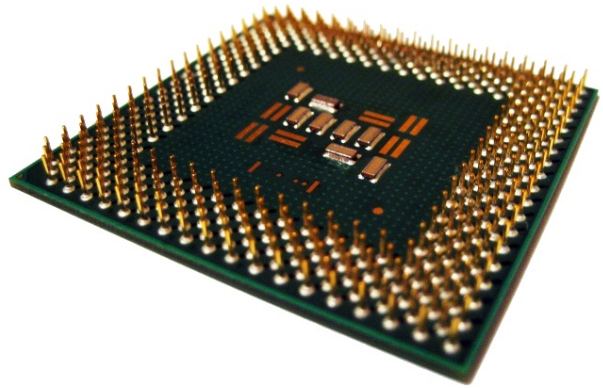


BROWN

Program Objects

```
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, (`G_x, G_y, G_z`), arg, ..., local_size=(`S_x, S_y, S_z`), wait_for=None)`



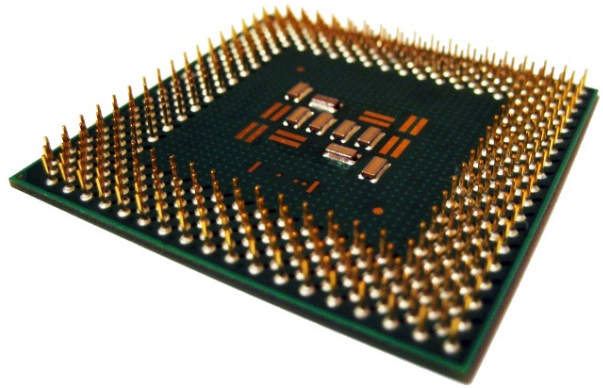
BROWN

Program Objects

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

`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`



BROWN

Another Look at the Example

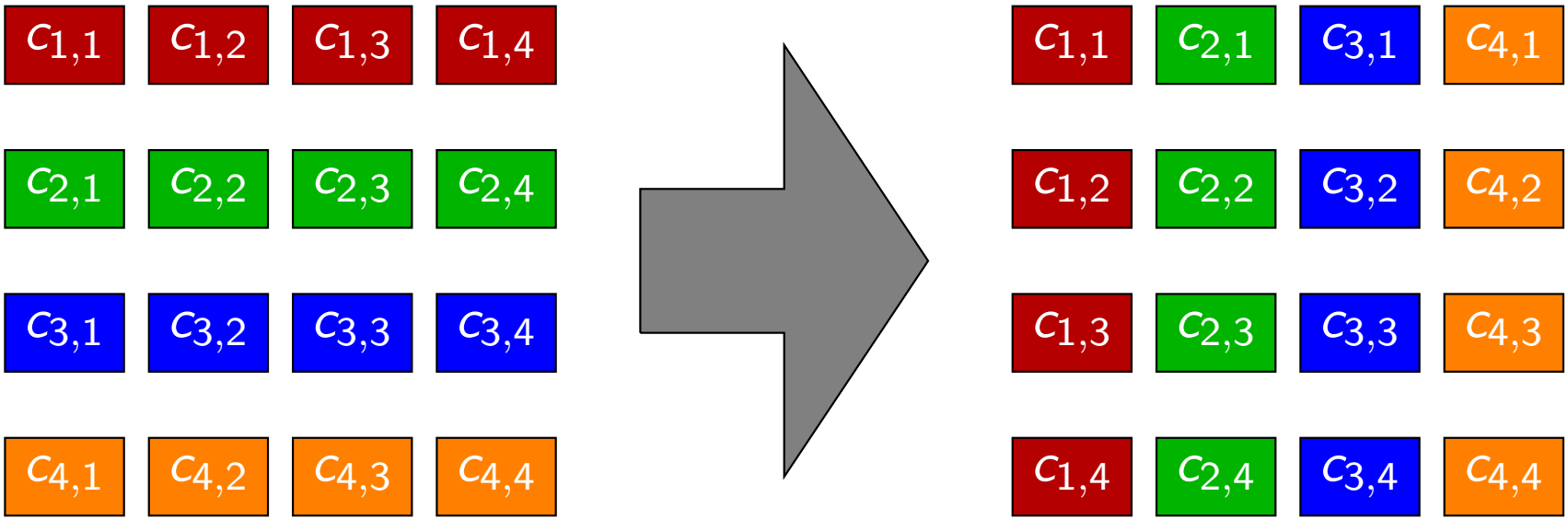
```
2  import pyopencl as cl
3  import numpy, numpy.linalg as la
4
5  a = numpy.random.rand(128*128).astype(numpy.float32)
6
7  ctx = cl.Context()
8  queue = cl.CommandQueue(ctx)
9
10 mf = cl.mem_flags
11 a_buf = cl.Buffer(ctx,
12         mf.READ_WRITE | mf.COPY_HOST_PTR,
13         hostbuf=a)
```

Another Look at the Example

```
17  prg = cl.Program(ctx, """
18      __kernel void twice(__global float *a)
19      {
20        a[get_global_id(0)] *= 2;
21      }
22      """).build()
23
24  prg.twice(queue, a.shape, a_buf, local_size=(128,))
25
26  a_twice = numpy.empty_like(a)
27  cl.enqueue_read_buffer(queue, a_buf, a_twice).wait()
28
29  assert la.norm(a_twice-2*a) == 0
```

Outline

Example: Matrix Transpose



BROWN

Transpose? Simple Enough!

```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

```
w, h = shape
return self.kernel(queue, (w, h),
        tgt, src, numpy.uint32(w), numpy.uint32(h))
```

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Evaluate

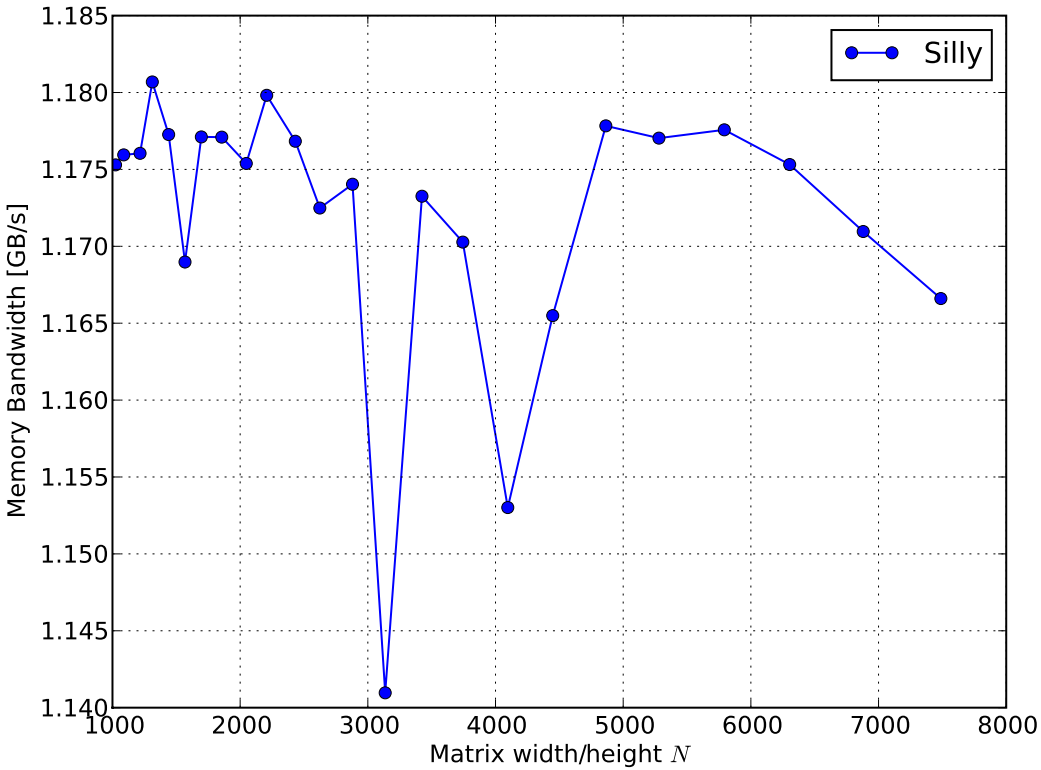
- Know your peak throughputs (roughly)
- Are you getting close?
- Are you tracking the right limiting factor?

Performance: Matrix transpose

Very likely: Bound by memory bandwidth.

Performance: Matrix transpose

Very likely: Bound by memory bandwidth.



Fantastic! Far slower than CPU. Why?

Intra-device Work Distribution

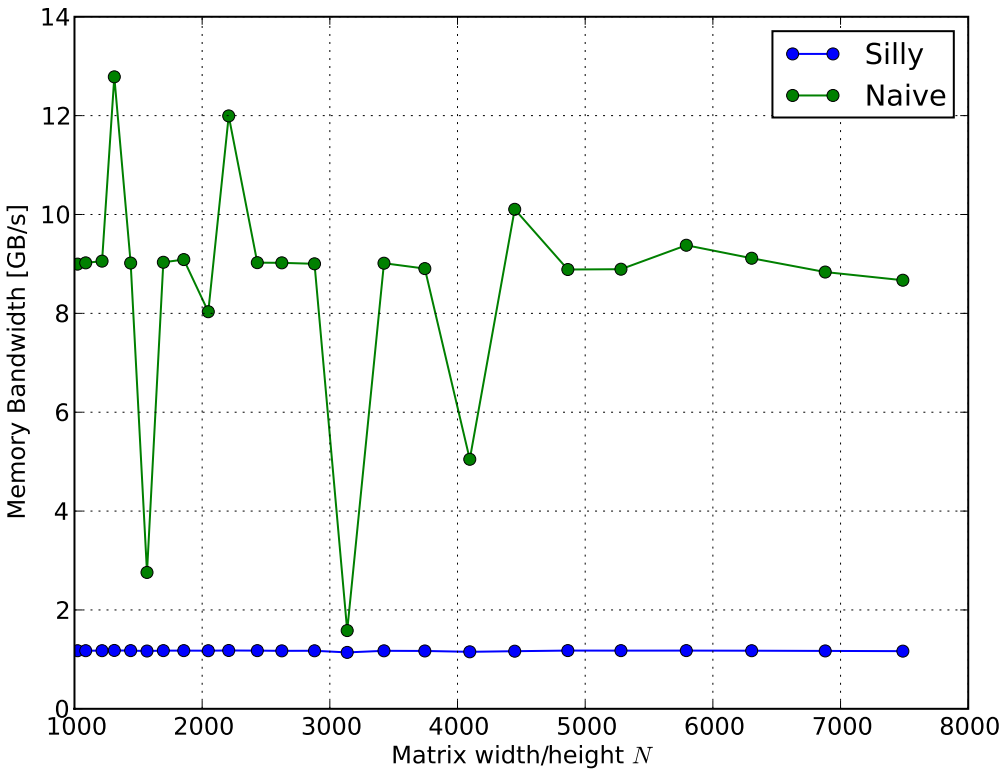
```
        w, h = shape
        return self.kernel(queue, (w, h),
            tgt, src, numpy.uint32(w), numpy.uint32(h))
```

```
        w, h = shape
        return self.kernel(queue, (w, h),
            tgt, src, numpy.uint32(w), numpy.uint32(h),
            local_size =(16, 16))
```

Work Groups

- `local_size` matters.
- Determines work distribution among processors
- Optimal size? Up to experimentation

Performance: Matrix transpose



Better. 1.5× faster than CPU–not great. Why?

Parallel Memories: Different Approaches

Problem

Digital memories have only one data bus.

Parallel Memories: Different Approaches

Problem

Digital memories have only one data bus.

So how can multiple threads read multiple data items from memory simultaneously?

Parallel Memories: Different Approaches

Problem

Digital memories have only one data bus.

So how can multiple threads read multiple data items from memory simultaneously?

Solutions: Parallel Access to Memory

- Split a really wide data bus, but have only one address bus

Parallel Memories: Different Approaches

Problem

Digital memories have only one data bus.

So how can multiple threads read multiple data items from memory simultaneously?

Solutions: Parallel Access to Memory

- Split a really wide data bus, but have only one address bus
- Have many "small memories" ("*banks*") with separate address busses. Pick bank by LSB of address.

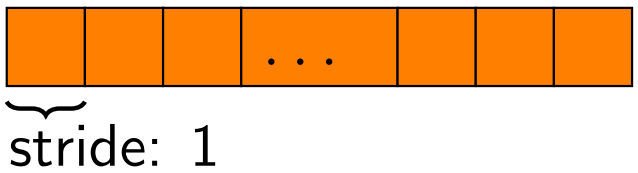
Naive: Using Global Memory

```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

Naive: Using Global Memory

```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

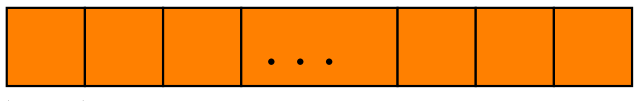
Reading from global mem:



Naive: Using Global Memory

```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

Reading from global mem:

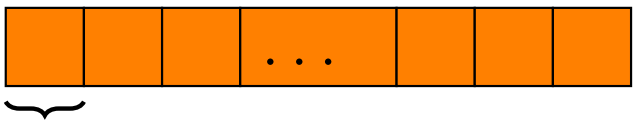


stride: 1 → one mem.trans.

Naive: Using Global Memory

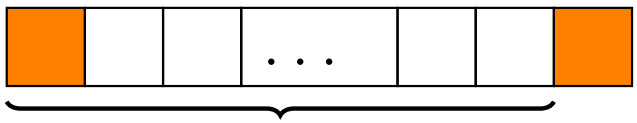
```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

Reading from global mem:



stride: 1 → one mem.trans.

Writing to global mem:

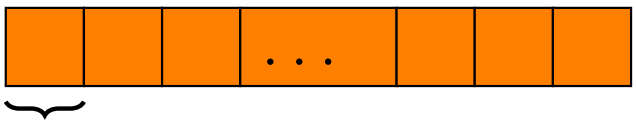


stride: 16

Naive: Using Global Memory

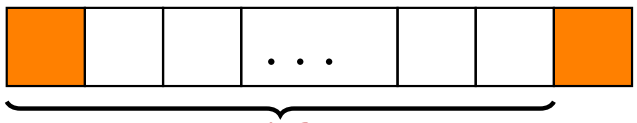
```
self.kernel = cl.Program(ctx, """
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
""").build().transpose
```

Reading from global mem:



stride: 1 → one mem.trans.

Writing to global mem:



stride: 16 → **16 mem.trans.!**

Transpose: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn't mind.

Transpose: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn't mind.

Idea

- Don't transpose element-by-element.
- Transpose block-by-block instead.

Transpose: Idea

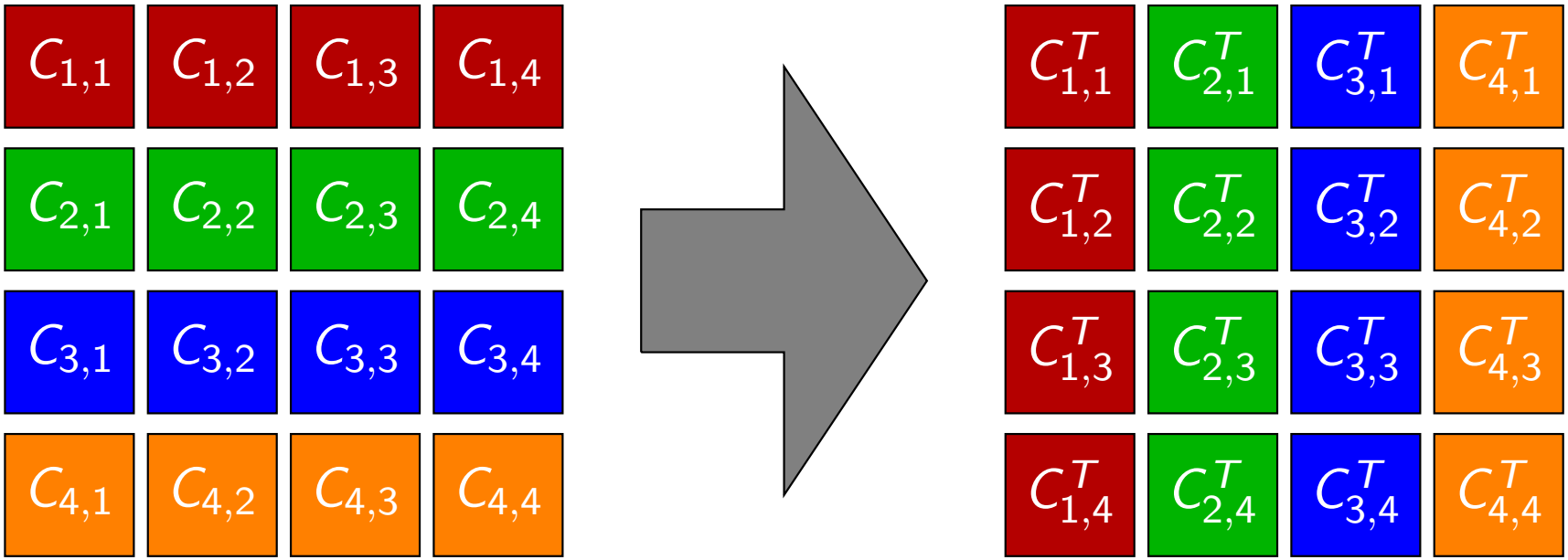
- Global memory dislikes non-unit strides.
- Local memory doesn't mind.

Idea

- Don't transpose element-by-element.
- Transpose block-by-block instead.

1. Read untransposed block from global and write to local
2. Synchronization?
3. Read block transposed from local and write to global

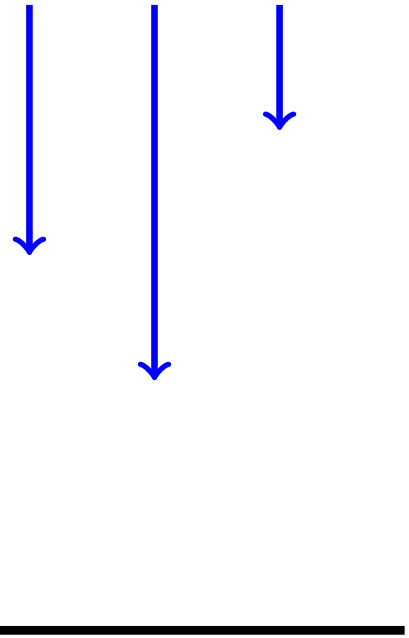
Illustration: Blockwise Transpose



BROWN

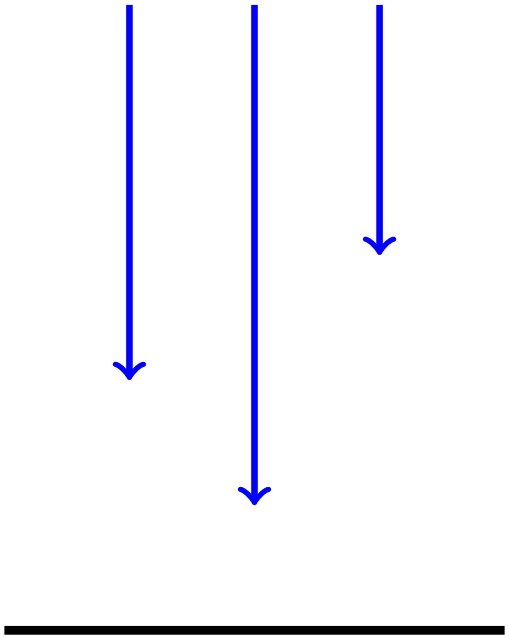
Synchronization

What is a Barrier?



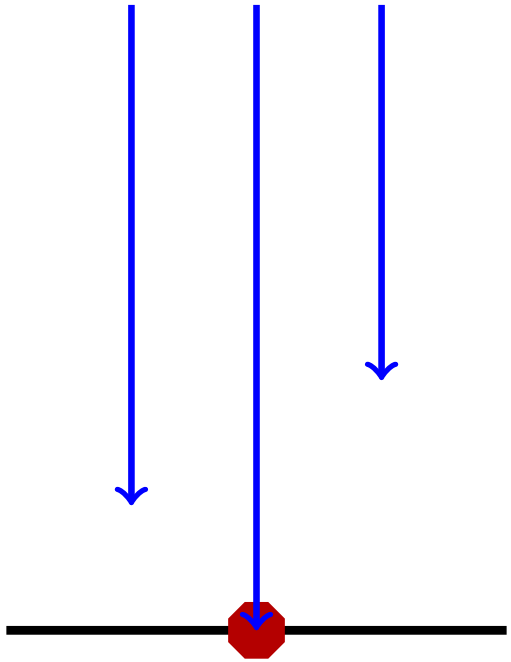
Synchronization

What is a Barrier?



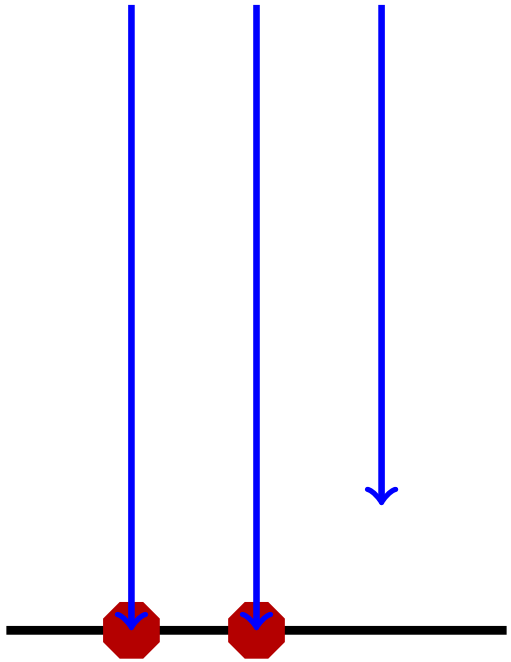
Synchronization

What is a Barrier?



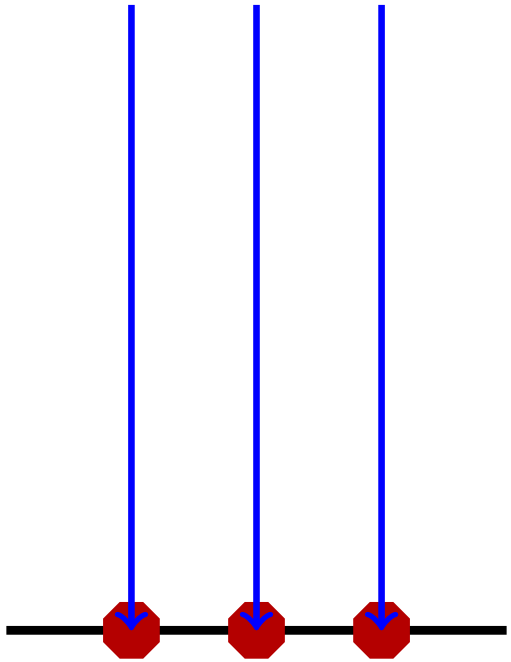
Synchronization

What is a Barrier?



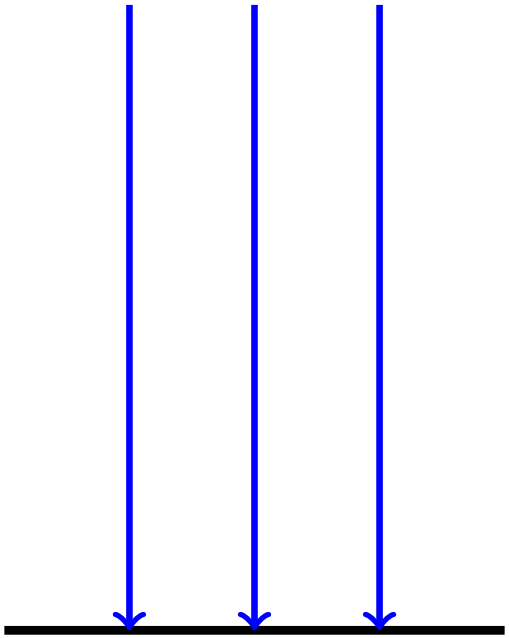
Synchronization

What is a Barrier?



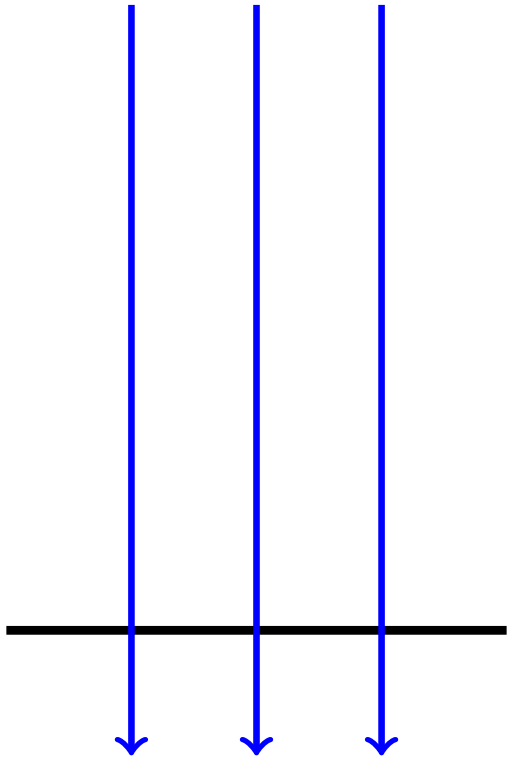
Synchronization

What is a Barrier?



Synchronization

What is a Barrier?



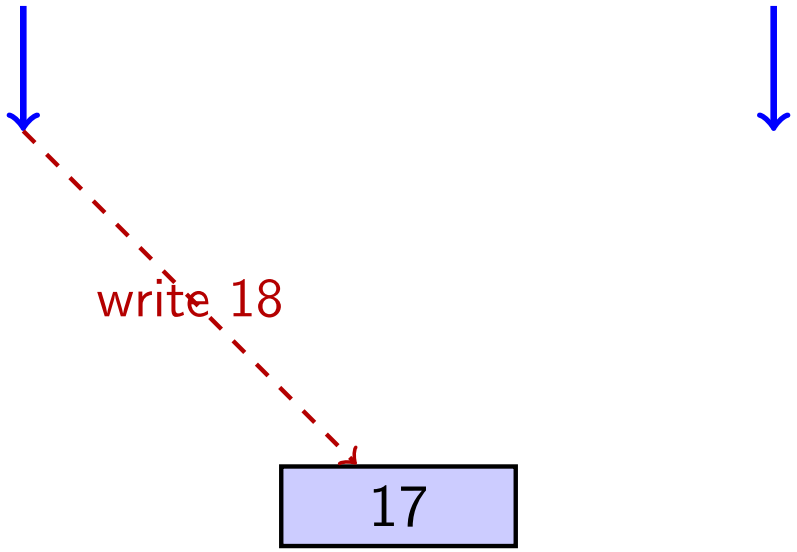
Synchronization

What is a Memory Fence?

17

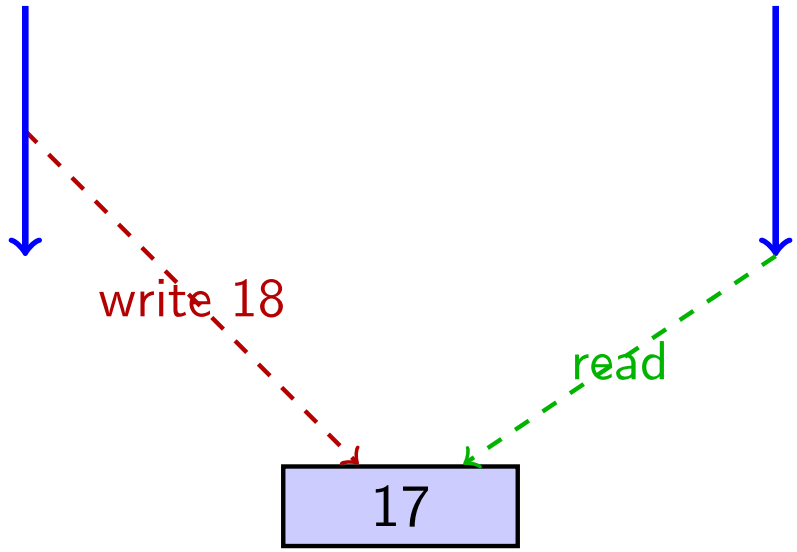
Synchronization

What is a Memory Fence?



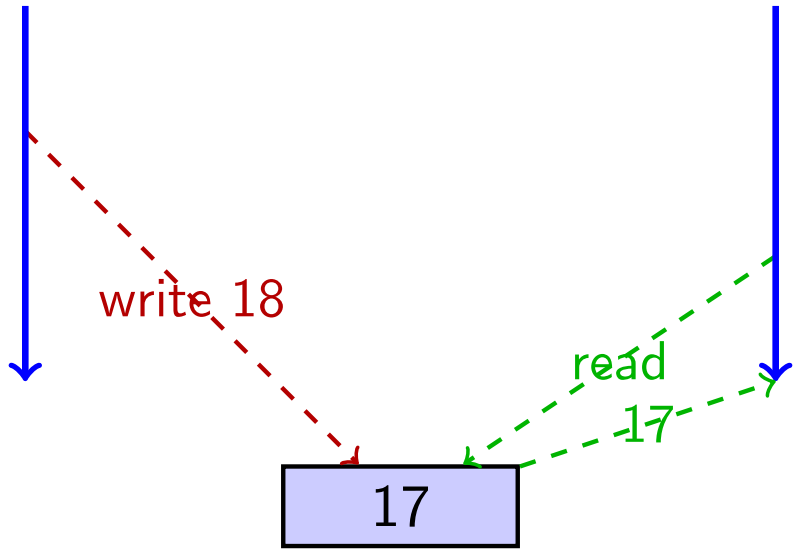
Synchronization

What is a Memory Fence?



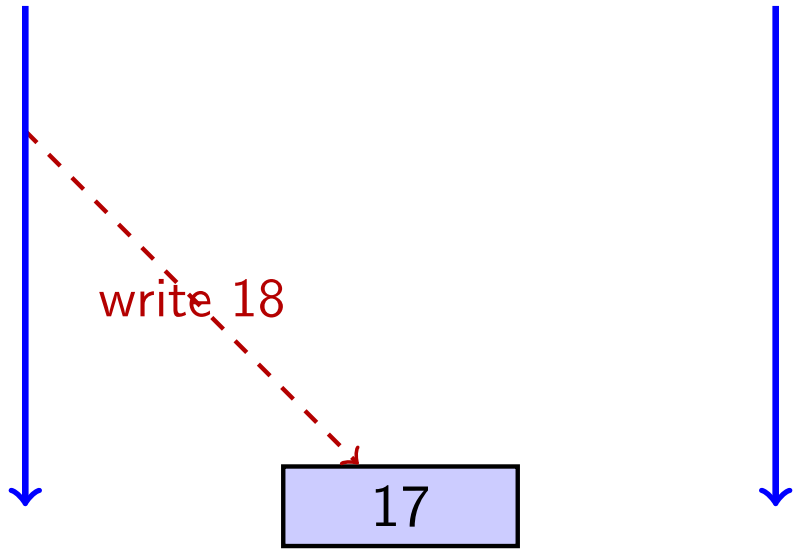
Synchronization

What is a Memory Fence?



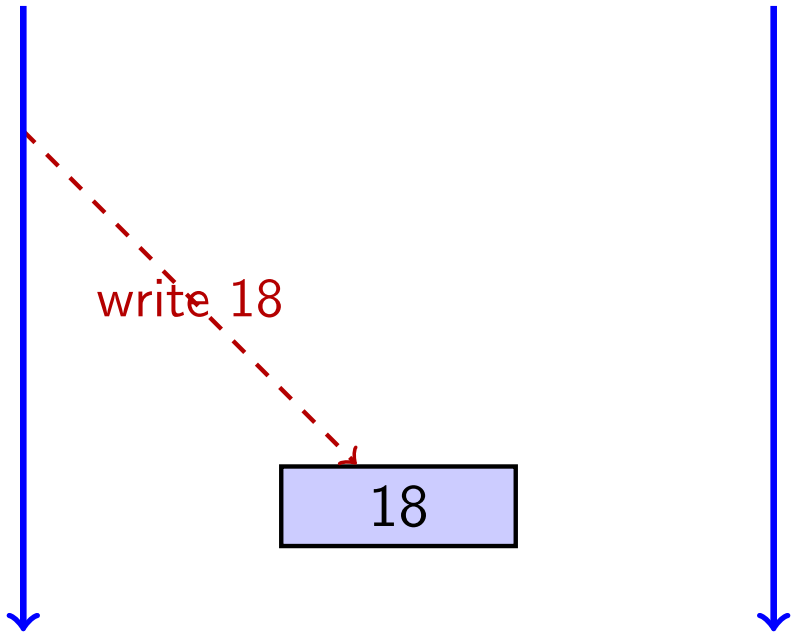
Synchronization

What is a Memory Fence?



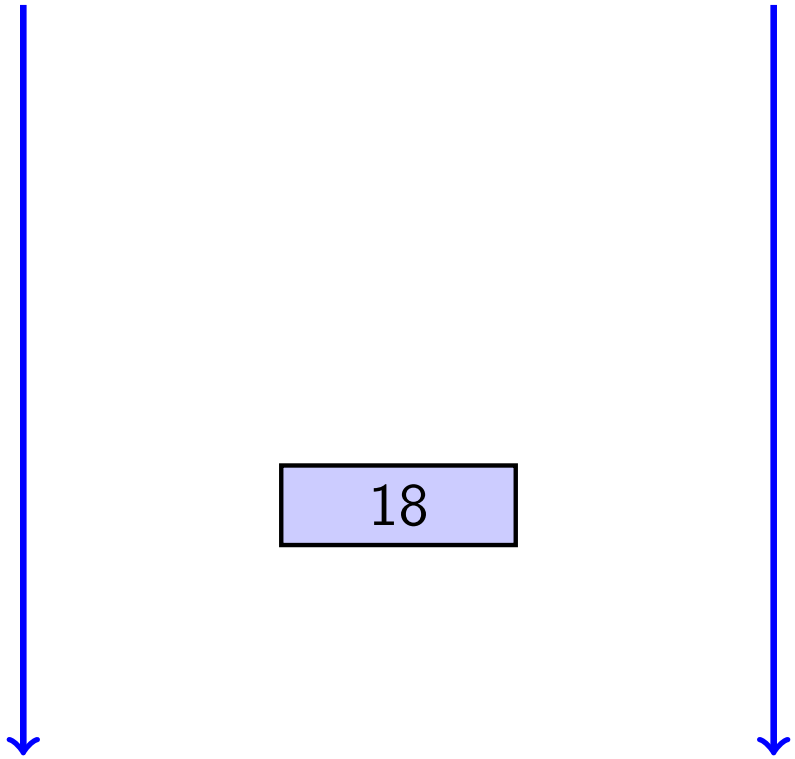
Synchronization

What is a Memory Fence?



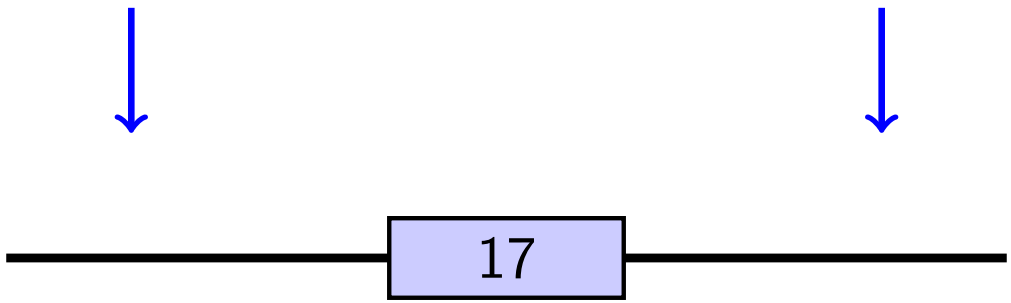
Synchronization

What is a Memory Fence?



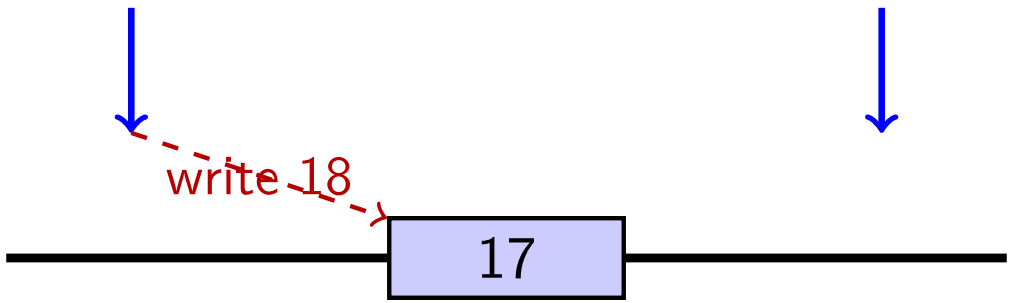
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



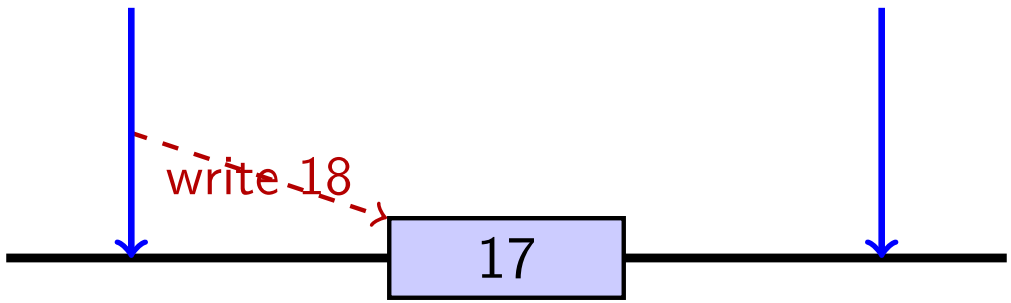
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



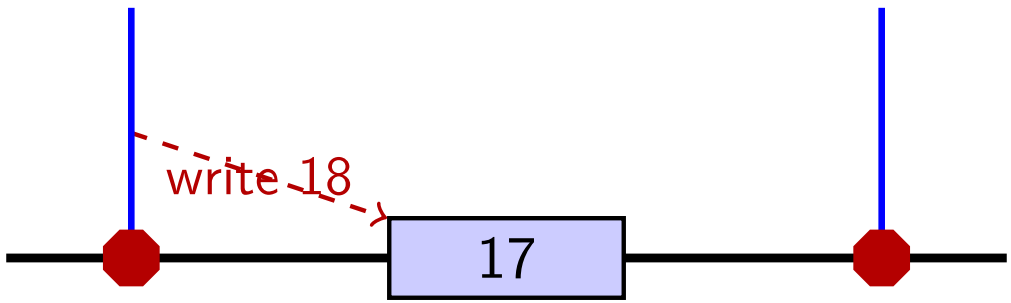
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



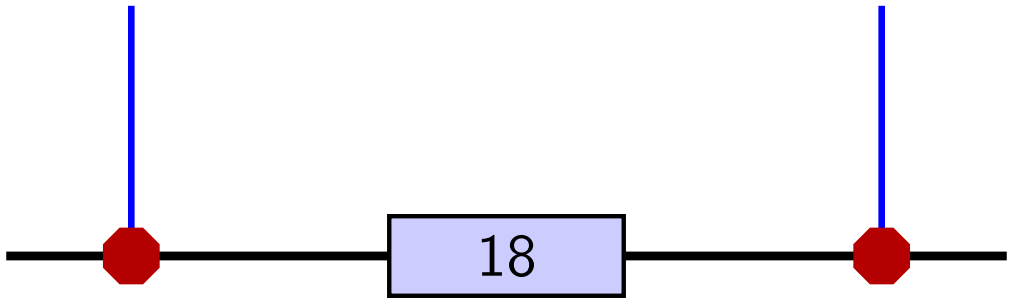
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



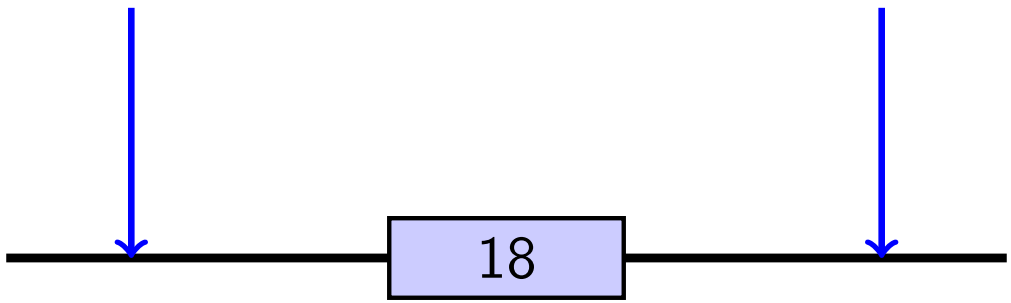
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



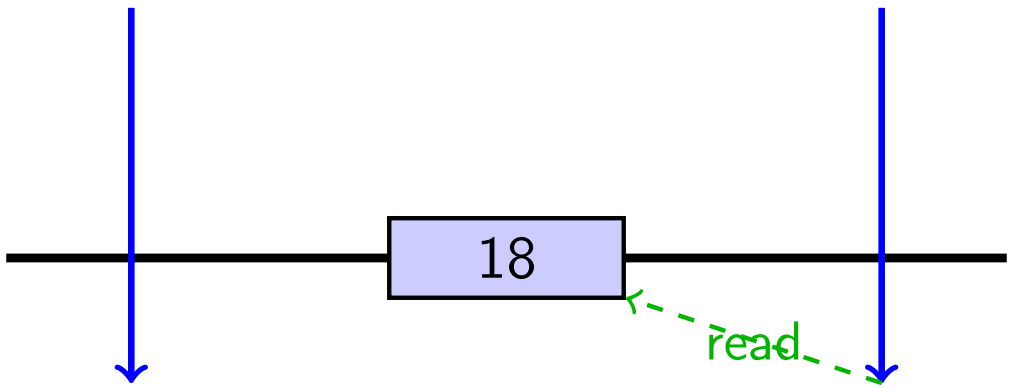
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



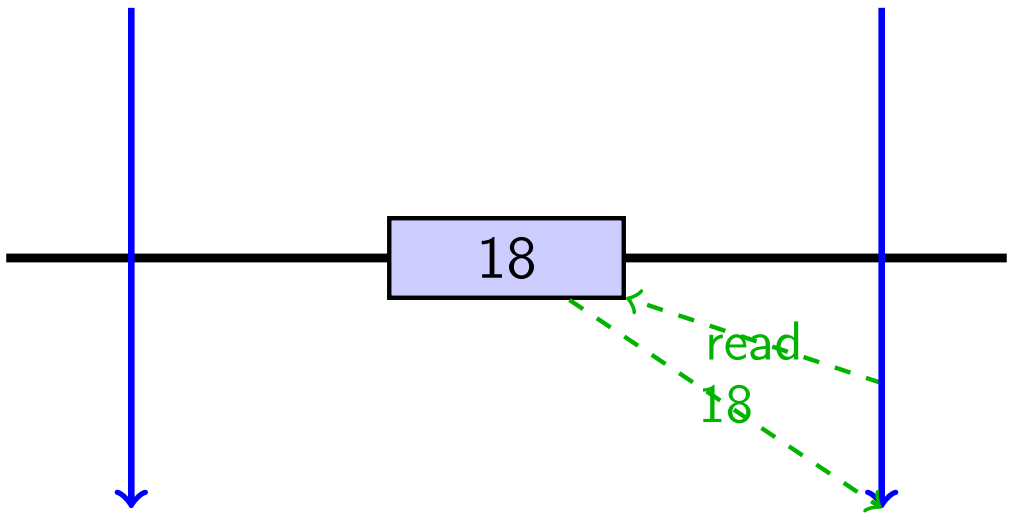
Synchronization

What is a Memory Fence? An ordering restriction for memory access.



Synchronization

What is a Memory Fence? An ordering restriction for memory access.



Improved: With Local Memory

Part 1/3:

```
#define BLOCK_SIZE 16
#define A_BLOCK_STRIDE (BLOCK_SIZE * a_width)
#define A_T_BLOCK_STRIDE (BLOCK_SIZE * a_height)

__kernel __attribute__
  ((reqd_work_group_size(BLOCK_SIZE, BLOCK_SIZE, 1)))
void transpose(
  __global float *a_t, __global float *a,
  unsigned a_width, unsigned a_height,
  __local float a_local[BLOCK_SIZE][BLOCK_SIZE])
```

Improved: With Local Memory

Part 2/3:

```
{
  int base_idx_a =
    get_group_id(0) * BLOCK_SIZE +
    get_group_id(1) * A_BLOCK_STRIDE;
  int base_idx_a_t =
    get_group_id(1) * BLOCK_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);
```

Improved: With Local Memory

Part 3/3:

```
        a_local [ get_local_id (1)][ 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)][ get_local_id (1)];
    }
```

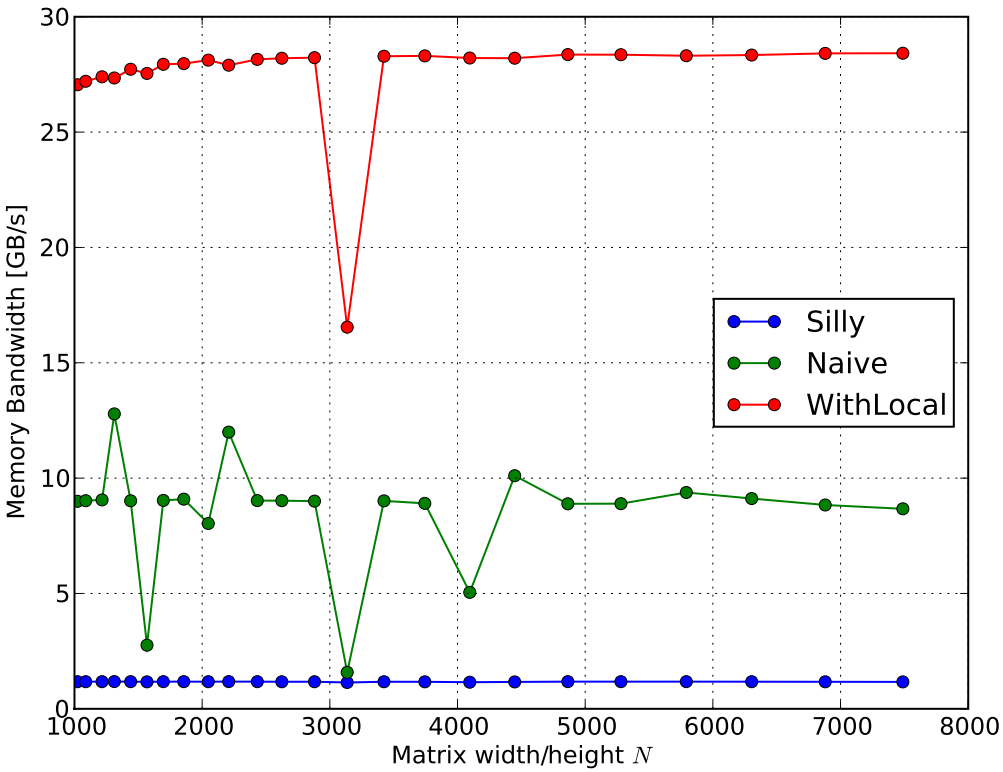
Improved: With Local Memory

Launch Code:

```
        w, h = shape

        return self.kernel(queue, (w, h),
            tgt, src, numpy.uint32(w), numpy.uint32(h),
            cl.LocalMemory(4*16*16),
            local_size=(16, 16))
```

Transpose example is `examples/transpose.py` in PyOpenCL source tree.

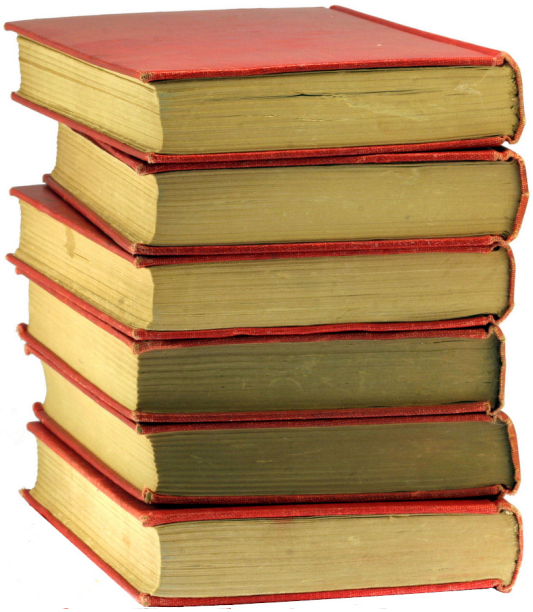
Performance: Matrix transpose



Much better. Not peak, but good enough.

Outline

PyOpenCL Philosophy



- Provide complete access
- Automatically manage resources
- Provide abstractions
- Allow interactive use
- Check for and report errors automatically
- Integrate tightly with `numpy`

BROWN

PyOpenCL: Completeness

PyOpenCL exposes *all* of OpenCL.

For example:

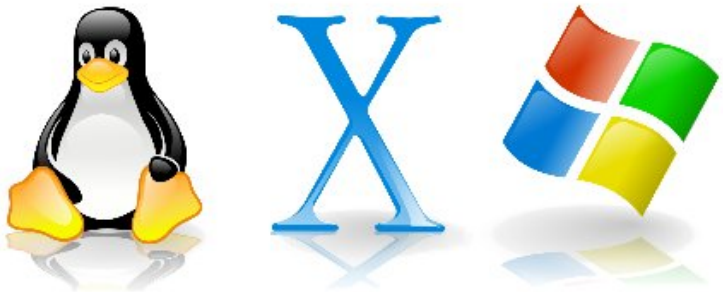
- Every `GetInfo()` query
- Images and Samplers
- Memory Maps
- Profiling and Synchronization
- GL Interop

BROWN

PyOpenCL: Completeness

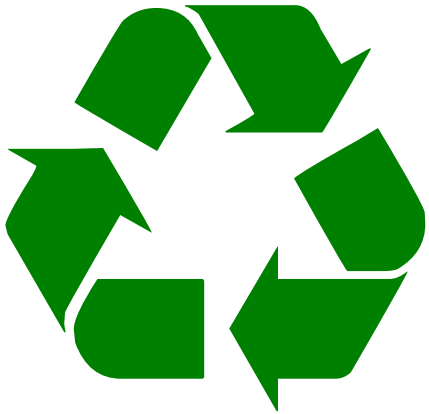
PyOpenCL supports (nearly) every OS that has an OpenCL implementation.

- Linux
- OS X
- Windows (getting there)



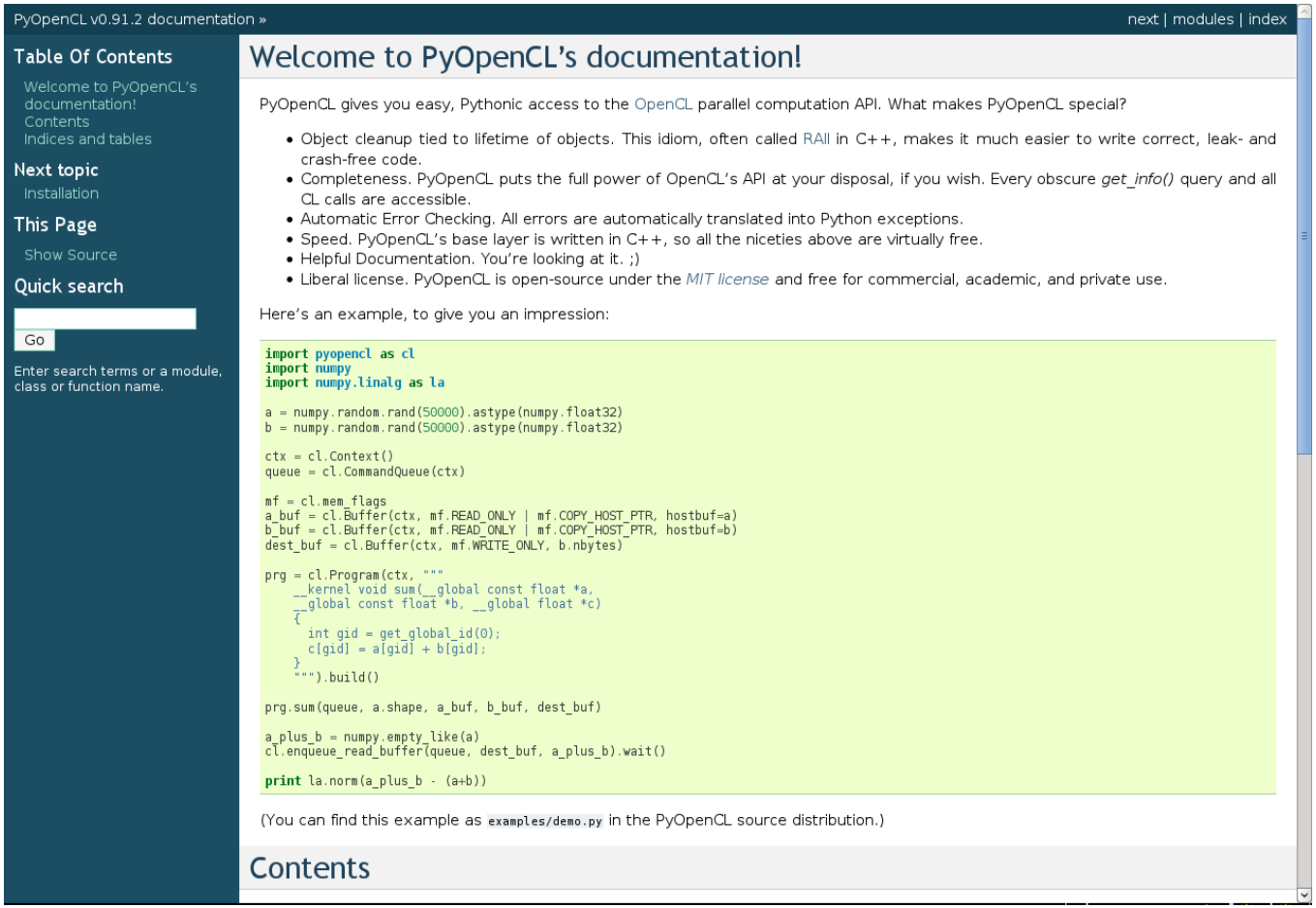
Automatic Cleanup

- Reachable objects (memory, streams, ...) are never destroyed.
- Once unreachable, released at an unspecified future time.
- Scarce resources (memory) can be explicitly freed. (`obj.release()`)
- Correctly deals with multiple contexts and dependencies.



BROWN

PyOpenCL: Documentation



PyOpenCL v0.91.2 documentation » next | modules | index

Table Of Contents

Welcome to PyOpenCL's documentation!
Contents
Indices and tables

Next topic

Installation

This Page

Show Source

Quick search

Go

Enter search terms or a module, class or function name.

Welcome to PyOpenCL's documentation!

PyOpenCL gives you easy, Pythonic access to the OpenCL parallel computation API. What makes PyOpenCL special?

- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code.
- Completeness. PyOpenCL puts the full power of OpenCL's API at your disposal, if you wish. Every obscure *get_info()* query and all CL calls are accessible.
- Automatic Error Checking. All errors are automatically translated into Python exceptions.
- Speed. PyOpenCL's base layer is written in C++, so all the niceties above are virtually free.
- Helpful Documentation. You're looking at it. ;)
- Liberal license. PyOpenCL is open-source under the *MIT license* and free for commercial, academic, and private use.

Here's an example, to give you an impression:

```
import pyopencl as cl
import numpy
import numpy.linalg as la

a = numpy.random.rand(50000).astype(numpy.float32)
b = numpy.random.rand(50000).astype(numpy.float32)

ctx = cl.Context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.nbytes)

prg = cl.Program(ctx, """
    __kernel void sum(__global const float *a,
    __global const float *b, __global float *c)
    {
      int gid = get_global_id(0);
      c[gid] = a[gid] + b[gid];
    }
    """).build()

prg.sum(queue, a.shape, a_buf, b_buf, dest_buf)

a_plus_b = numpy.empty_like(a)
cl.enqueue_read_buffer(queue, dest_buf, a_plus_b).wait()

print la.norm(a_plus_b - (a+b))
```

(You can find this example as `examples/demo.py` in the PyOpenCL source distribution.)

Contents

PyOpenCL: Vital Information

- `http://mathema.tician.de/software/pyopencl`
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Boost C++, Python 2.4+.
- Support via mailing list.

BROWN

Questions?

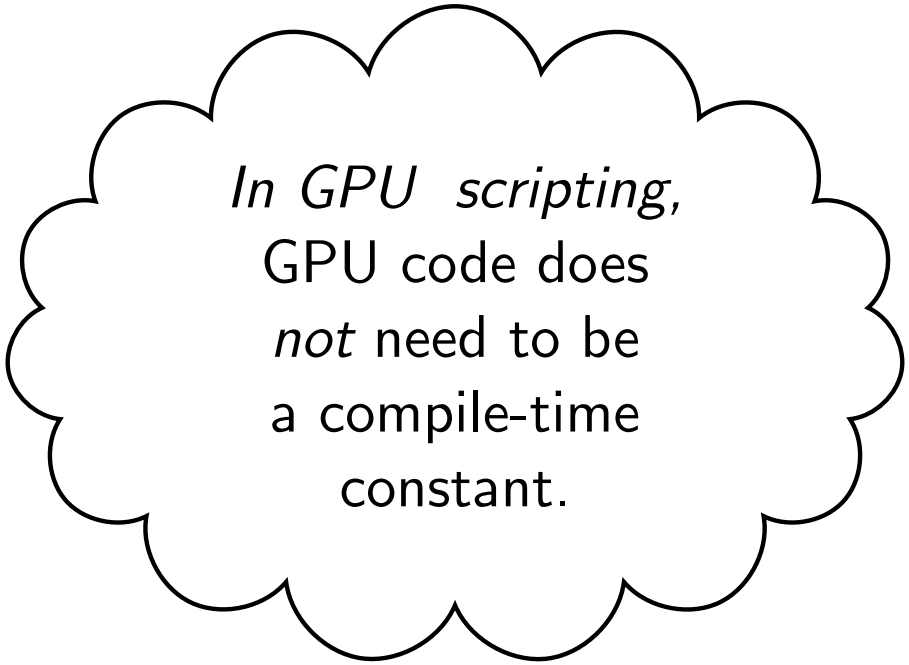
?

▸ end

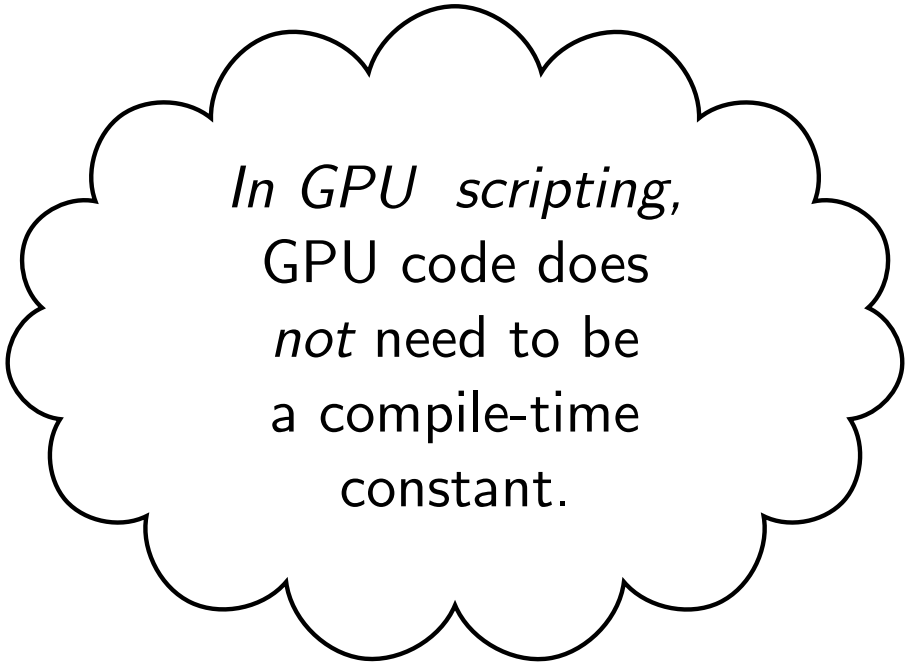
Outline

Outline

Metaprogramming



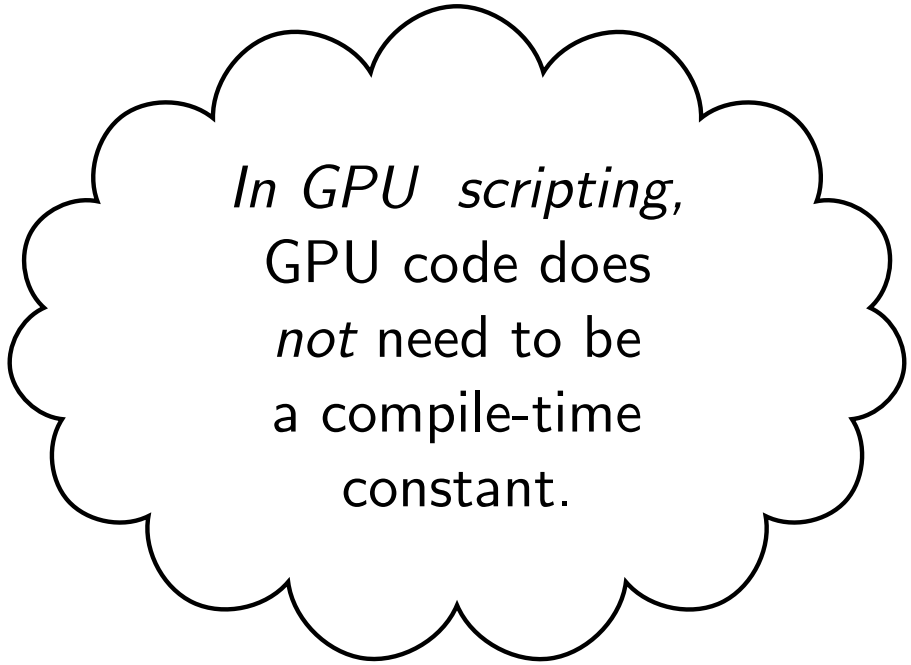
Metaprogramming



(Key: Code is data–it *wants* to be reasoned about at run time)

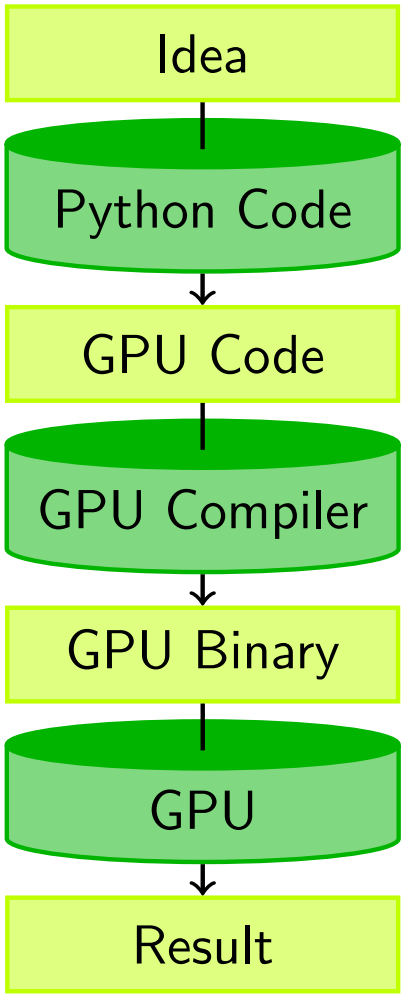
Metaprogramming

Idea



(Key: Code is data–it *wants* to be reasoned about at run time)

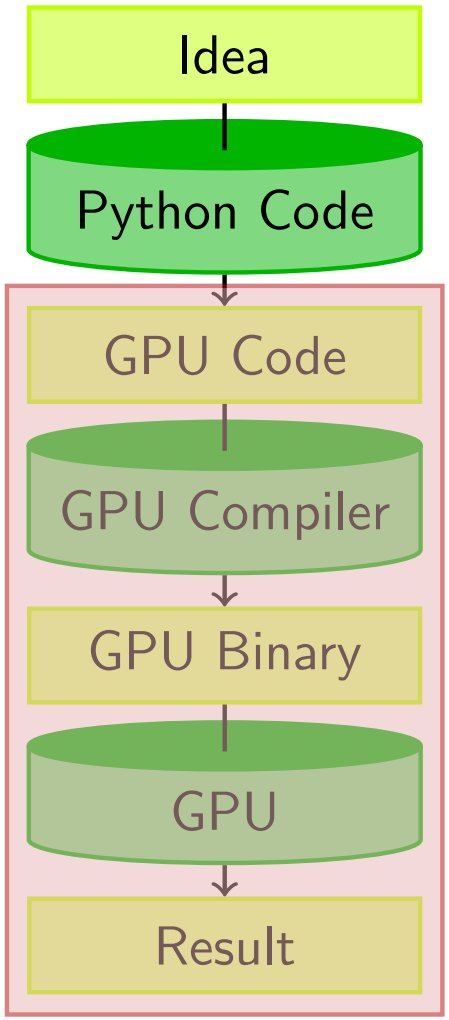
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

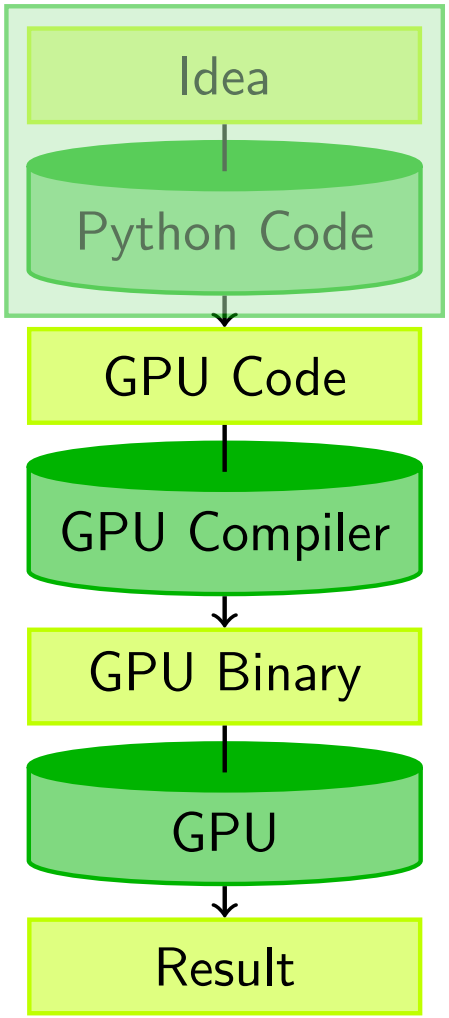
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

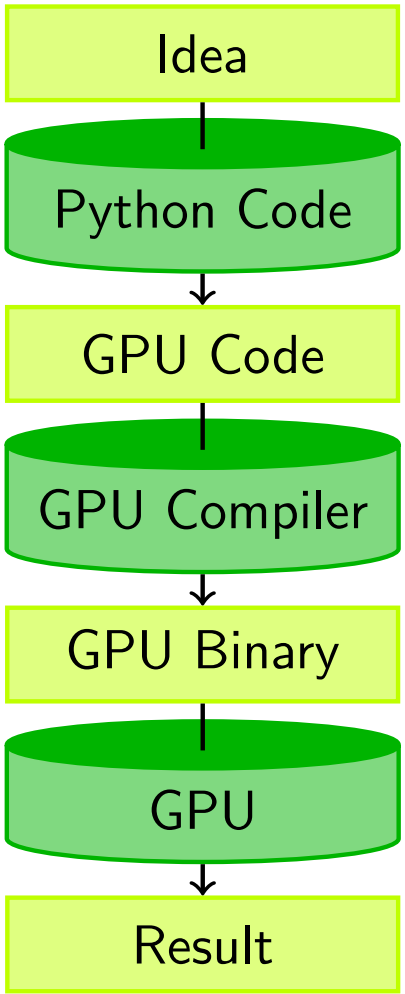
Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

Metaprogramming

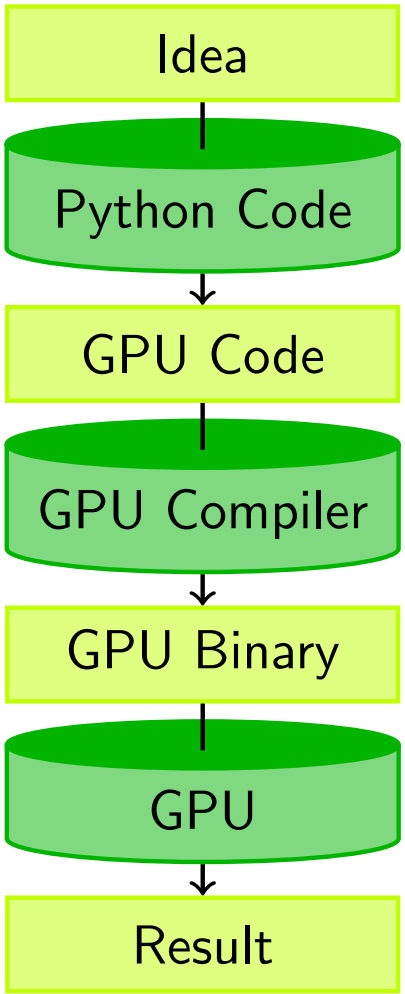


Good for code generation

In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

Metaprogramming



Good for code generation

In PyOpenCL GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

Machine-generated Code

Why machine-generate code?

- Automated Tuning
  (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables
  ($\rightarrow$ register pressure)
- Loop Unrolling

BROWN

PyOpenCL: Support for Metaprogramming

Three (main) ways of generating code:

- Simple %-operator substitution
- Use a templating engine (Jinja 2 works very well)
- `codepy`:
  - Build C syntax trees from Python
  - Generates readable, indented C

Many ways of evaluating code–most important one:

- Exact device timing via events

RTCG via Templates

```
from jinja2 import Template

tpl = Template("""
    __kernel void twice( __global {{ type_name }} *tgt)
    {
      int idx = get_local_id (0)
        + {{ local_size }} * {{ thread_strides }}
        * get_group_id (0);

      {% for i in range( thread_strides ) %}
          {% set offset = i* local_size %}
        tgt[idx + {{ offset }}] *= 2;
      {% endfor %}
    }""")

rendered_tpl = tpl.render(type_name="float",
    local_size = local_size, thread_strides = thread_strides)

knl = cl.Program(ctx, str( rendered_tpl )). build (). twice
```

RTCG via AST Generation

```
from codepy.cgen import *
from codepy.cgen.opencl import \
        CLKernel, CLGlobal, CLRequiredWorkGroupSize

mod = Module([
    FunctionBody(
        CLKernel(CLRequiredWorkGroupSize((local_size,),
            FunctionDeclaration(Value("void", "twice"),
            arg_decls=[CLGlobal(Pointer(Const(POD(dtype, "tgt"))))])),
        Block([
            Initializer (POD(numpy.int32, "idx"),
                " get_local_id (0) + %d * get_group_id(0)"
                % ( local_size * thread_strides ))
            ]+[
            Statement("tgt[idx+%d] *= 2" % (o*local_size))
            for o in range( thread_strides )]
            ))])

knl = cl.Program(ctx, str(mod)).build().twice
```

Questions?

?

Outline

Outline

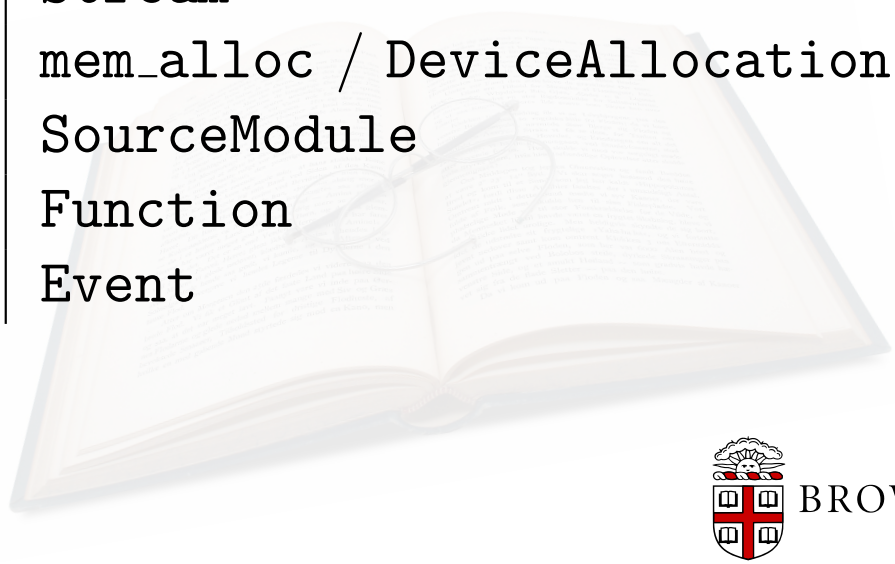
BROWN

OpenCL ↔ CUDA: A dictionary

OpenCL	CUDA
Grid	Grid
Work Group	Block
Work Item	Thread
`__kernel`	`__global__`
`__global`	`__device__`
`__local`	`__shared__`
`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)

PyOpenCL ↔ PyCUDA: A (rough) dictionary

PyOpenCL	PyCUDA
`Context`	`Context`
`CommandQueue`	`Stream`
`Buffer`	`mem_alloc` / `DeviceAllocation`
`Program`	`SourceModule`
`Kernel`	`Function`
`Event` (eg. `enqueue_marker`)	`Event`



BROWN

Whetting your appetite

```
1  import pycuda.driver as cuda
2  import pycuda.autoinit
3  import numpy
4
5  a = numpy.random.randn(4,4).astype(numpy.float32)
6  a_gpu = cuda.mem_alloc(a.nbytes)
7  cuda.memcpy_htod(a_gpu, a)
```

[This is `examples/demo.py` in the PyCUDA distribution.]

Whetting your appetite

```
 9  mod = cuda.SourceModule("""
10       __global__ void twice( float *a)
11       {
12         int idx = threadIdx.x + threadIdx.y*4;
13         a[idx] *= 2;
14       }
15       """)
16
17  func = mod.get_function("twice")
18  func(a_gpu, block=(4,4,1))
19
20  a_doubled = numpy.empty_like(a)
21  cuda.memcpy_dtoh(a_doubled, a_gpu)
22  print a_doubled
23  print a
```

Whetting your appetite

```
 9  mod = cuda.SourceModule("""
10       __global__ void twice( float *a)
11      {
12        int idx = threadIdx.x + threadIdx.y*4;
13        a[idx] *= 2;
14      }
15      """)
16
17  func = mod.get_function("twice")
18  func(a_gpu, block=(4,4,1))
19
20  a_doubled = numpy.empty_like(a)
21  cuda.memcpy_dtoh(a_doubled, a_gpu)
22  print a_doubled
23  print a
```

Compute kernel

Whetting your appetite, Part II

Did somebody say "Abstraction is good"?

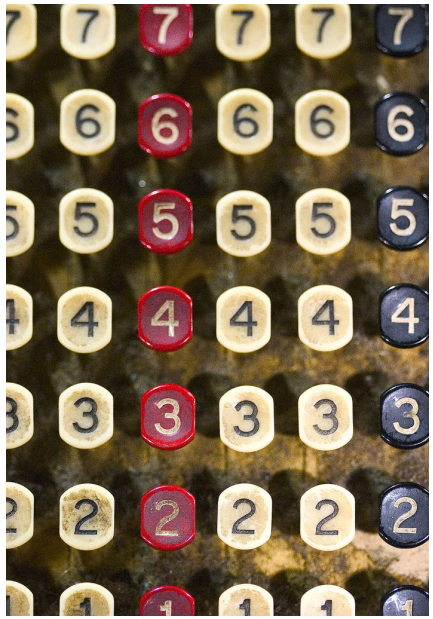
Whetting your appetite, Part II

```
1  import numpy
2  import pycuda.autoinit
3  import pycuda.gpuarray as gpuarray
4
5  a_gpu = gpuarray.to_gpu(
6      numpy.random.randn(4,4).astype(numpy.float32))
7  a_doubled = (2*a_gpu).get()
8  print a_doubled
9  print a_gpu
```

gpuarray: Simple Linear Algebra

`pycuda.gpuarray`:

- Meant to look and feel just like `numpy`.
  - `gpuarray.to_gpu(numpy_array)`
  - `numpy_array = gpuarray.get()`
- +, -, ∗, /, fill, sin, exp, rand, basic indexing, norm, inner product, ...
- Mixed types (int32 + float32 = float64)
- `print gpuarray` for debugging.
- Allows access to raw bits
  - Use as kernel arguments, textures, etc.



BROWN

Outline

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile

Automating GPU Programming

GPU programming can be time-consuming, unintuitive and error-prone.

- Obvious idea: Let the computer do it.
- One way: Smart compilers
  - GPU programming requires complex tradeoffs
  - Tradeoffs require heuristics
  - Heuristics are fragile
- Another way: Dumb enumeration
  - Enumerate loop slicings
  - Enumerate prefetch options
  - Choose by running resulting code on actual hardware

Loo.py Example

Empirical GPU loop optimization:

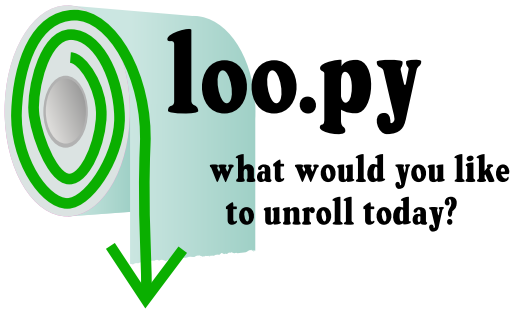
```
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
    ], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
    ])

gen_kwargs = {
        "min_threads": 128,
        "min_blocks": 32,
        }
```

→ Ideal case: Finds 160 GF/s kernel without human intervention.

BROWN

Loo.py Status

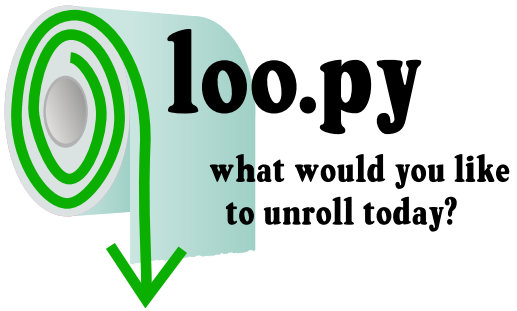
- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...



BROWN

Loo.py Status

- Limited scope:
  - Require input/output separation
  - Kernels must be expressible using "loopy" model
    (i.e. indices decompose into "output" and "reduction")
  - Enough for DG, LA, FD, ...
- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels



BROWN

Questions?

?

Thank you for your attention!

Want to get your hands dirty?
→ Hands-On Session this Afternoon

`http://mathema.tician.de/software/pyopencl`

BROWN

▸ image credits

Image Credits

- Fighting chips: flickr.com/oskay (CC)
- Isaiah die shot: VIA Technologies
- RV770 die shot: AMD Corp.
- Apples and Oranges: Mike Johnson - TheBusyBrain.com (CC)
- Nvidia Tesla Architecture: Nvidia Corp.
- RV870 Architecture: AMD Corp.
- Larabee Architecture: Intel Corp.
- C870 GPU: Nvidia Corp.
- Context: sxc.hu/svilen001
- Queue: sxc.hu/cobrasoft
- RAM stick: sxc.hu/gobran11
- CPU: sxc.hu/dimshik
- Old Books: flickr.com/ppdigital (CC)
- OpenCL Logo: Apple Corp./Ars Technica
- OS Platforms: flickr.com/aOliN.Tk
- Floppy disk: flickr.com/ethanhein (CC)
- Machine: flickr.com/13521837@N00 (CC)
- Dictionary: sxc.hu/topfer
- Adding Machine: flickr.com/thomashawk (CC)

BROWN