Easy, Effective, Efficient: GPU Programming in Python with PyOpenCL and PyCUDA

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

DTU GPU-Lab Workshop
Lecture 1 · August 17, 2011

Course Outline

Morning Session: Intro

- Python, numpy, GPUs
- OpenCL
- Basic PyOpenCL
- Tour of PyOpenCL Runtime
- Advanced PyOpenCL usage
- OpenCL device language
- PyOpenCL: Built-in tools
- CL Implementation Notes

Lunch Lab

- Python, numpy
- Basic PyOpenCL

Afternoon Session: Advanced

- Behind the scenes
- RTCG: How and Why, Templating
- Automated Tuning
- mpi4py and PyOpenCL
- Interfacing Python with Fortran and C/C++
- A brief look at PyCUDA

Afternoon Lab

- Continue on Lab 1
- Advanced PyOpenCL

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
4. OpenCL implementations

Outline

1 Intro: Python, Numpy, GPUs, OpenCL
- Python, Numpy
- GPUs
- OpenCL

2 GPU Programming with PyOpenCL

3 OpenCL viewed from Python

4 OpenCL implementations

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
   - Python, Numpy
   - GPUs
   - OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
4. OpenCL implementations

Python in 4 Minutes

Literals `1234, 1234., 0xabc`

```
"a string" """a multi-line
string""" ["a", "list"]
("a", "tuple", 17)
{"a": 17, "dictionary": 19}
```

Flow Control

```
if True and a == 10:
   print "?" # a comment
while 0 <= x < 17 :
   pass # break, continue
for i in [0, 1, 2]:
   raise Exception("!")
```

Functions, Classes

```
def my_function(x):
   return 17*x
class MyClass:
   def __init__(self, x):
      self.x = x
```

Program Semantics

```
a = [1,2,4]
b = a
b.append(17)
print a
# [1, 2, 4, 17]
```

Python in 4 Minutes

Literals `1234, 1234., 0xabc`

```
"a string" """a multi-line
string""" ["a", "l
("a", "tuple", 17)
{"a": 17, "diction
```

Flow Control

```
if True and a == 1
    print "?" # a c
while 0 <= x < 17
    pass # break, c
for i in [0, 1, 2]
    raise Excepti
```

Functions, Classes

```
def my_function(x):
    return 17*x
```

`http://docs.python.org`

More stuff:

- Python 2 vs Python 3
- 'Batteries included'
- The package index
- Cython, Jython, IronPython, PyPy
- Interactive console, IPython, PuDB, Virtualenv, Pip, Spyder, PEP 8

Numpy in 4 Minutes

Creating/Modifying Arrays

```
import numpy as np
x = np.array([[1,2],[4,5]])
print x.shape # (2,2)

y = np.zeros((20000, 3),
  dtype=np.float64)
z = np.empty((20000, 3))
u = np.ones((30, 40))
v = np.linspace(1, 5,20,
  endpoint=False)

# also: mgrid, eye, arange
+, -, *, +=, np.dot
```

Indexing Arrays

```
a = x[:, 1] # a 'view'
a[:, :] = 17
y = 17
x[3:-3:-1, :] = 17
x[x == 19] = 17
```

Broadcasting

```
y[:, :] = 17
y[:, :] = [0, 1, 2]
w = np.array([0, 1, 2]) \
  [:, np.newaxis] * [0, 1, 2]
```

NYU

Numpy in 4 Minutes

Creating/Modifying Arrays

```
import numpy as np
x = np.array([[1,2
print x.shape # (2

y = np.zeros((2000
  dtype=np.float64
z = np.empty((2000
u = np.ones((30, 4(
v = np.linspace(1,
  endpoint=False)

# also: mgrid, eye
+, -, *, +=, np.dot
```

Indexing Arrays

```
a = x[:, 1] # a 'view'
```

2]

http://docs.scipy.org

More stuff:

- 'ufuncs' `sin,exp,...`
- Linear Algebra, FFT, ..., SciPy
- Structured/masked arrays
- 'Fancy' Indexing
- Matplotlib, MayaVi2
- C API
- Google 'Numpy Medkit'

Questions?

?

Outline

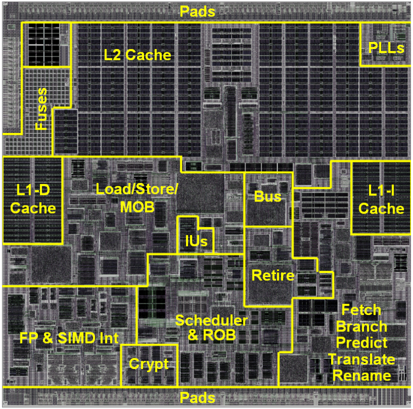
1 Intro: Python, Numpy, GPUs, OpenCL
- Python, Numpy
- GPUs
- OpenCL

2 GPU Programming with PyOpenCL

3 OpenCL viewed from Python

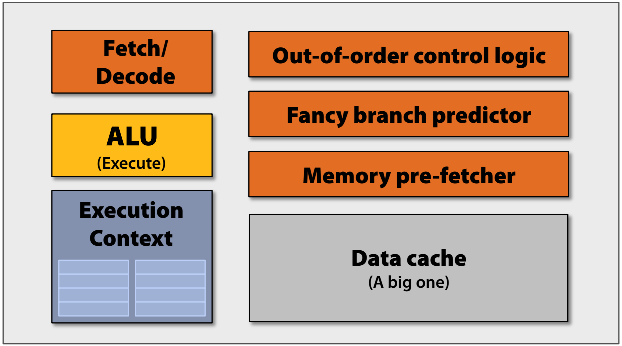
4 OpenCL implementations

CPU Chip Real Estate



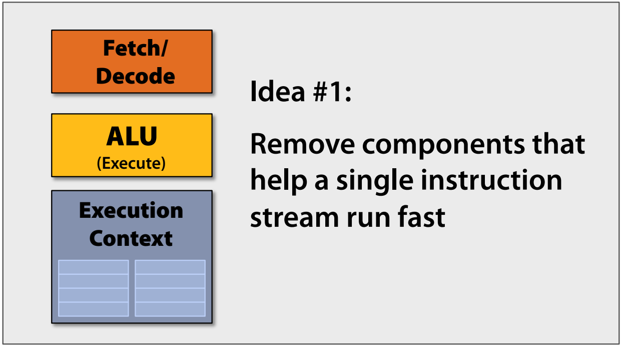
Die floorplan: VIA Isaiah (2008).
65 nm, 4 SP ops at a time, 1 MiB L2.

"CPU-style" Cores



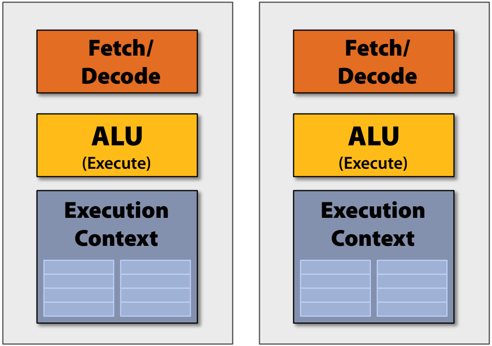
Credit: Kayvon Fatahalian (Stanford)

Slimming down



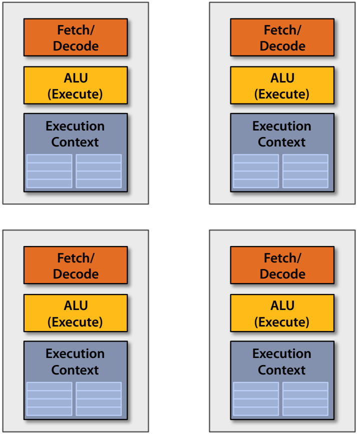
Credit: Kayvon Fatahalian (Stanford)

More Space: Double the Number of Cores



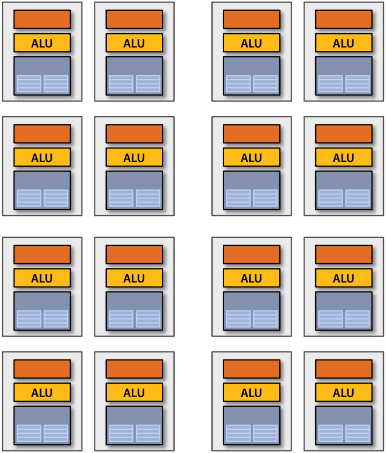
Credit: Kayvon Fatahalian (Stanford)

. . . again



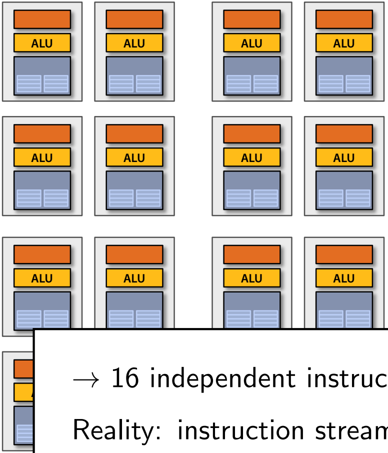
Credit: Kayvon Fatahalian (Stanford)

. . . and again



Credit: Kayvon Fatahalian (Stanford)

. . . and again

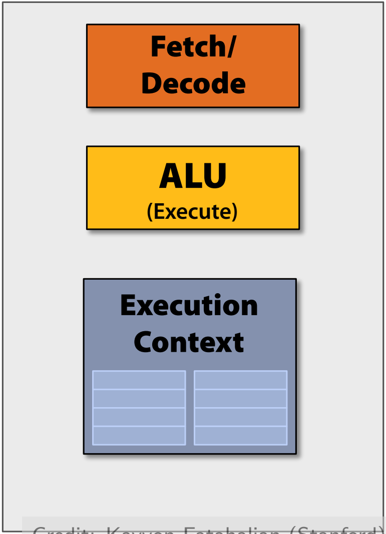


→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

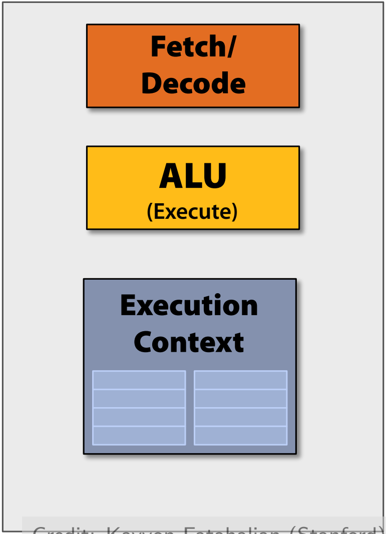
Credit: Kayvon Fatahalian (

Saving Yet More Space



Credit: Kayvon Fatahalian (Stanford)

Saving Yet More Space



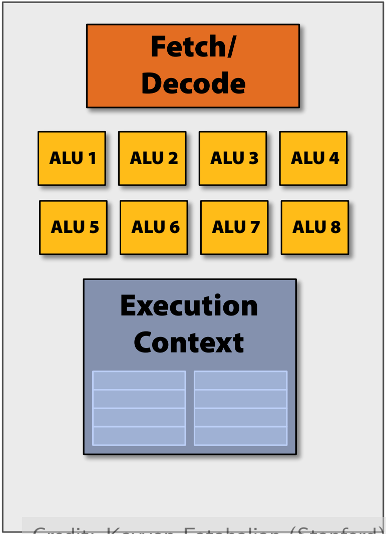
Credit: Kayvon Fatahalian (Stanford)

Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ **SIMD**

Saving Yet More Space



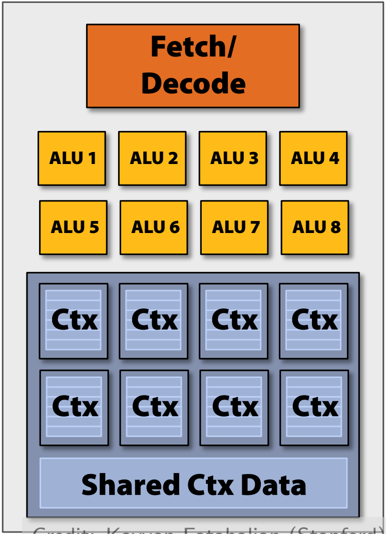
Credit: Kayvon Fatahalian (Stanford)

Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ **SIMD**

Saving Yet More Space



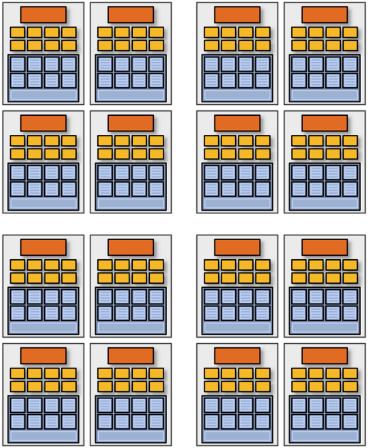
Credit: Kayvon Fatahalian (Stanford)

Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

$\rightarrow$ **SIMD**

Gratuitous Amounts of Parallelism!

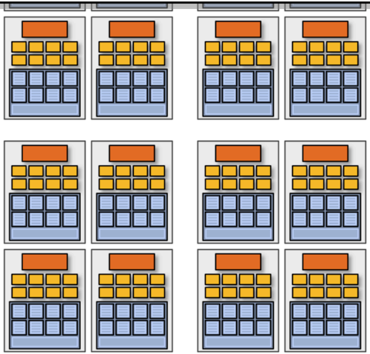


Credit: Kayvon Fatahalian (Stanford)

Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams

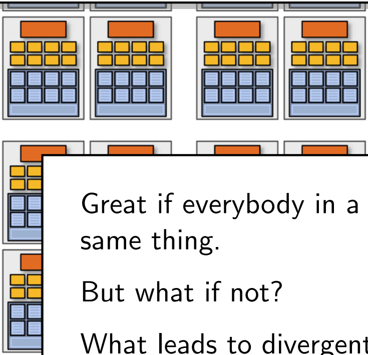


Credit: Kayvon Fatahalian (Stanford)

Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams



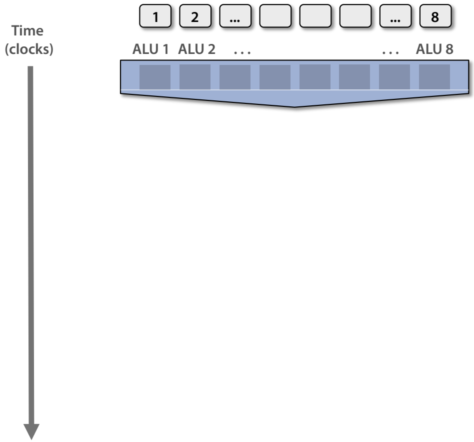
Great if everybody in a group does the same thing.

But what if not?

What leads to divergent instruction streams?

Credit: Kayvon Fatahalian (

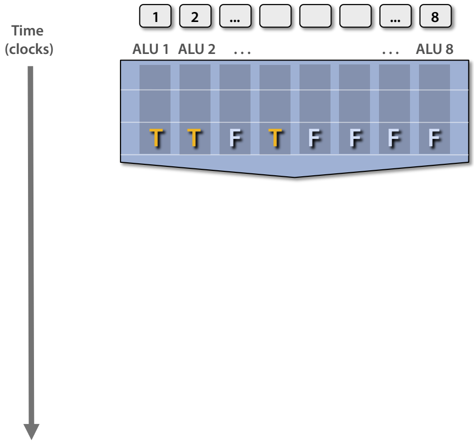
Branches




```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

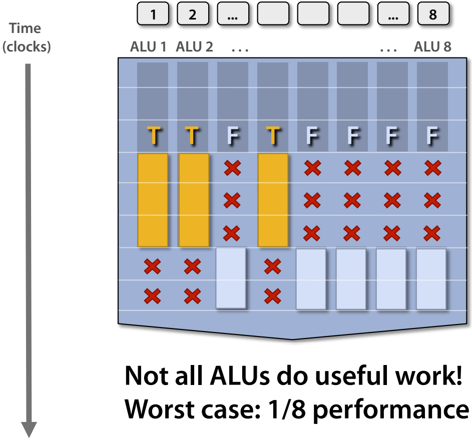
Branches




```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

Branches

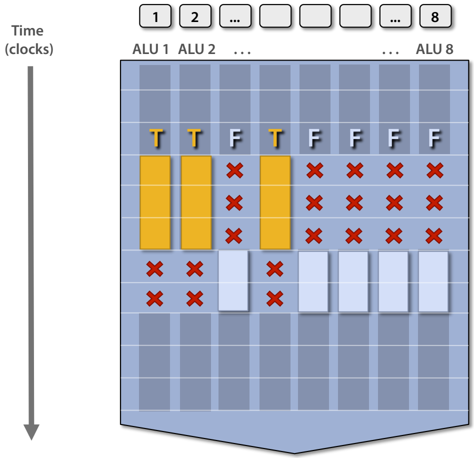


Not all ALUs do useful work!
Worst case: 1/8 performance

```
<unconditional
shader code>

if (x > 0) {
        y = pow(x, exp);
        y *= Ks;
        refl = y + Ka;
} else {
        x = 0;
        refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

Branches




```
<unconditional
shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional
shader code>
```

Credit: Kayvon Fatahalian (Stanford)

Remaining Problem: Slow Memory

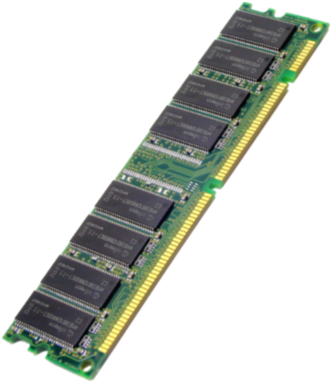
Problem

Memory still has very high latency. . .
. . . but we've removed most of the hardware that helps us deal with that.

We've removed

- caches
- branch prediction
- out-of-order execution

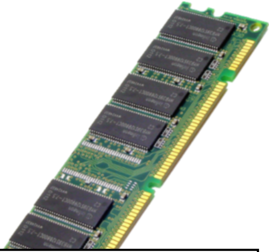
So what now?



Remaining Problem: Slow Memory

Problem

Memory still has very high latency. . .
. . . but we've removed most of the hardware that helps us deal with that.



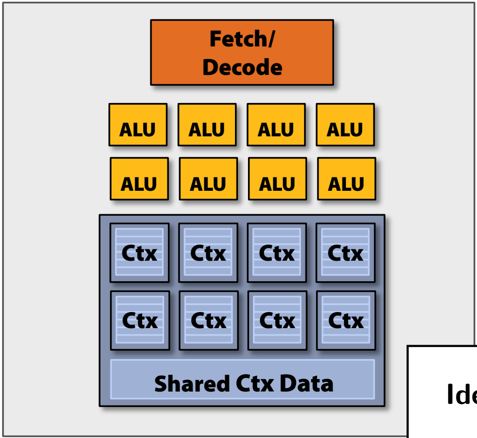
We've removed

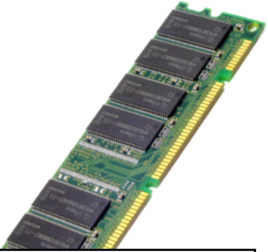
- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3

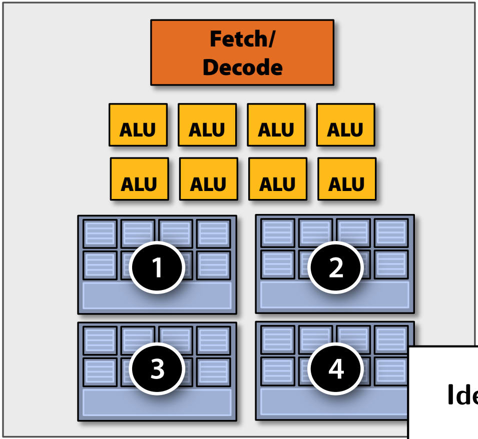
	Even more parallelism
+	Some extra memory
=	A solution!

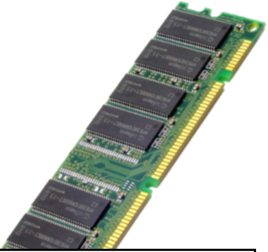




Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!





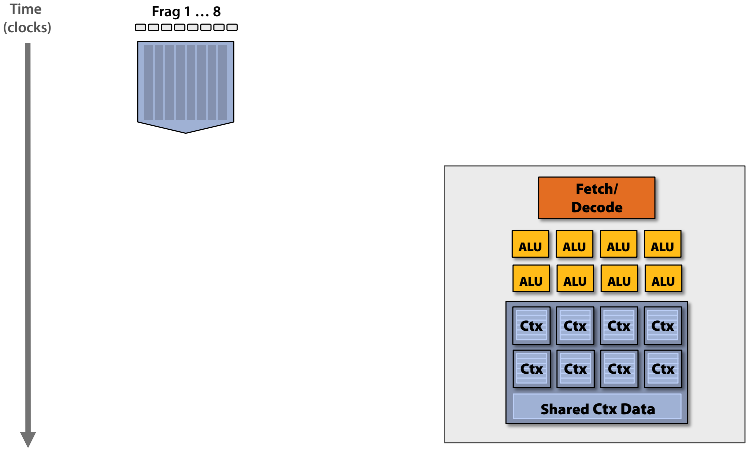
Idea #3

Even more parallelism

\+ Some extra memory

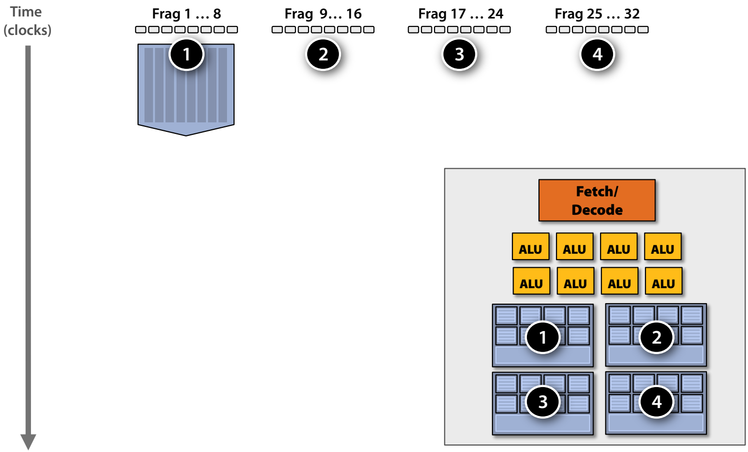
= A solution!

Hiding Memory Latency



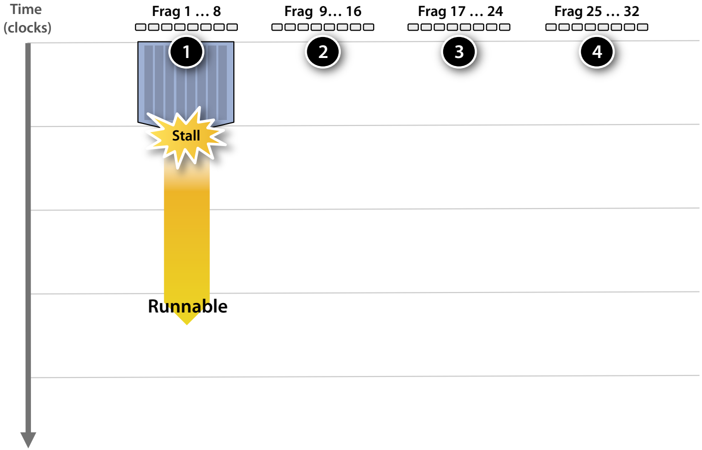
Credit: Kayvon Fatahalian (Stanford)

Hiding Memory Latency



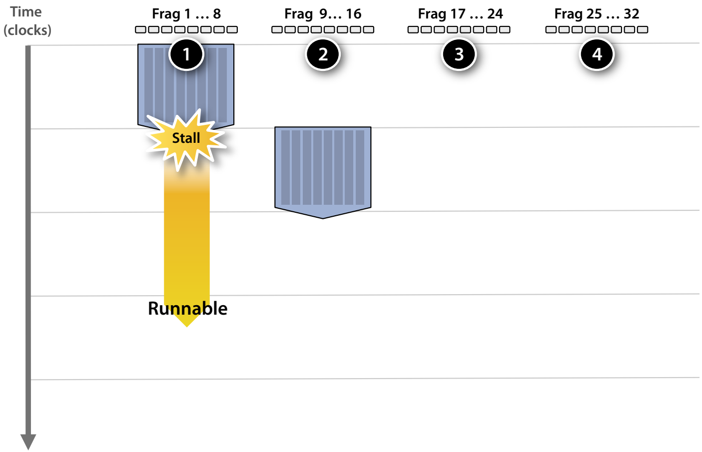
Credit: Kayvon Fatahalian (Stanford)

Hiding Memory Latency



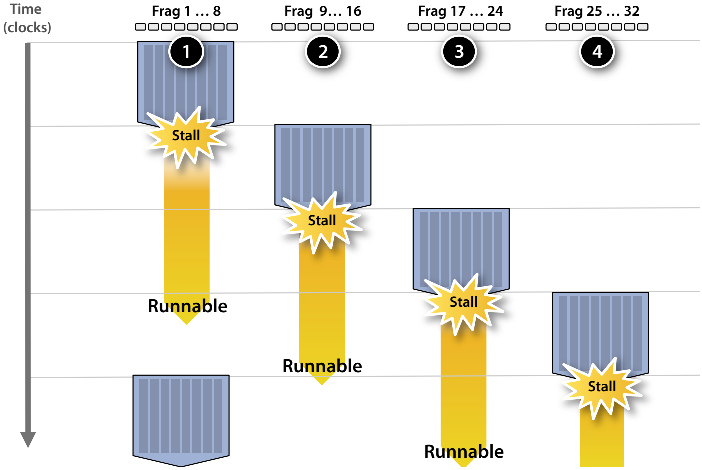
Credit: Kayvon Fatahalian (Stanford)

Hiding Memory Latency



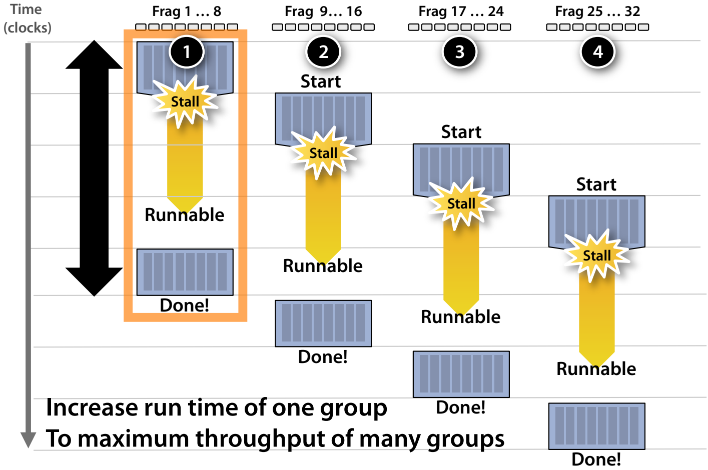
Credit: Kayvon Fatahalian (Stanford)

Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

Hiding Memory Latency



Credit: Kayvon Fatahalian (Stanford)

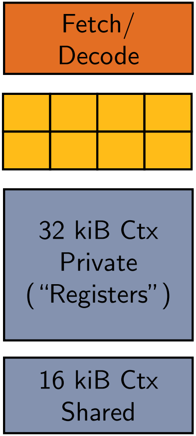
GPU Architecture Summary

Core Ideas:

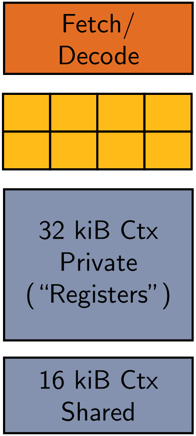
1. Many slimmed down cores
   $\rightarrow$ lots of parallelism
2. More ALUs, Fewer Control Units
3. Avoid memory stalls by interleaving execution of SIMD groups ("warps")

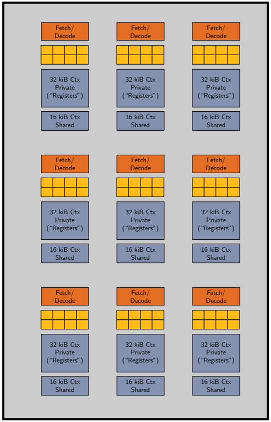
Credit: Kayvon Fatahalian (Stanford)

Connection: Hardware ↔ Programming Model

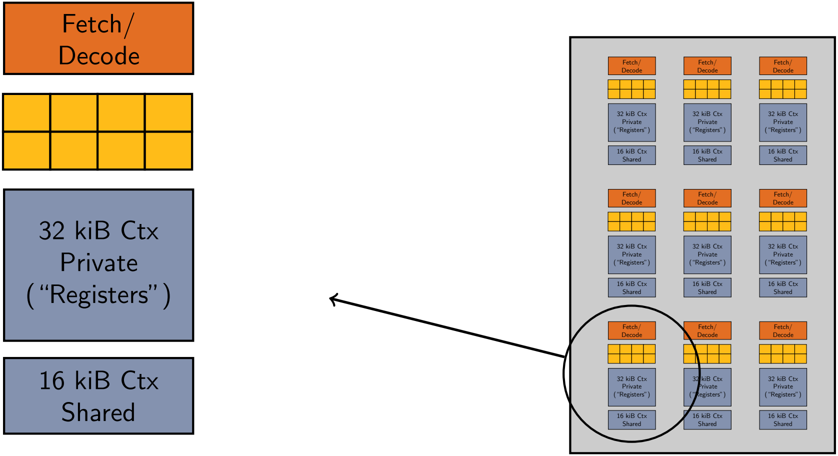


Connection: Hardware ↔ Programming Model

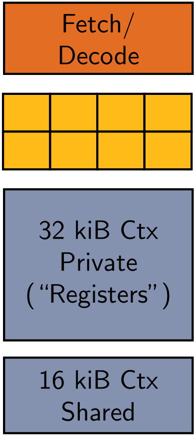


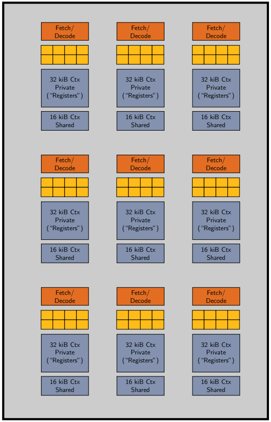


Connection: Hardware ↔ Programming Model

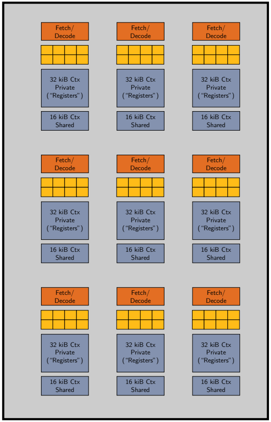


Connection: Hardware ↔ Programming Model





Connection: Hardware ↔ Programming Model

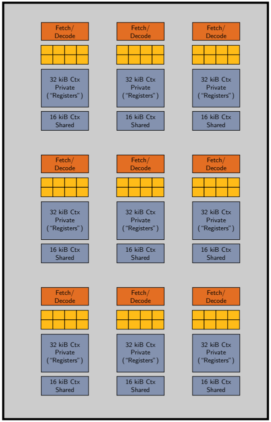


Connection: Hardware ↔ Programming Model

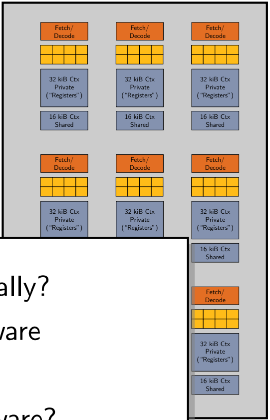
Who cares how many cores?

Idea:

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



Connection: Hardware ↔ Programming Model



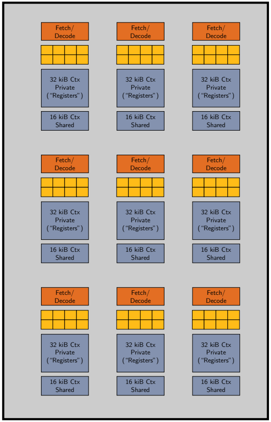
Consider: Which is easy to do automatically?

- Parallel program → sequential hardware

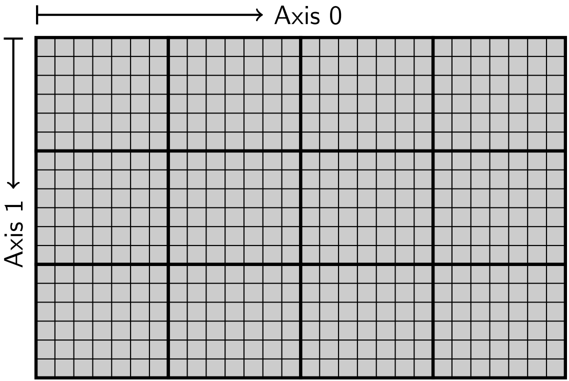
or

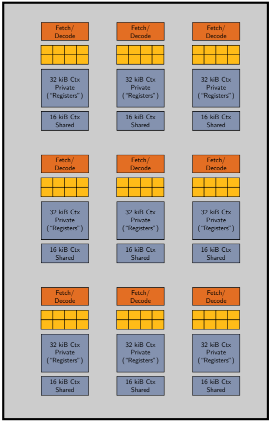
- Sequential program → parallel hardware?

Connection: Hardware ↔ Programming Model

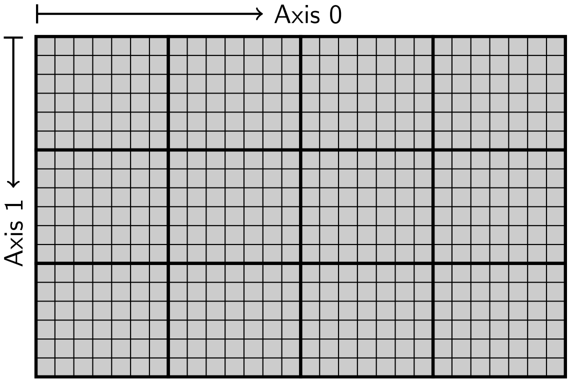


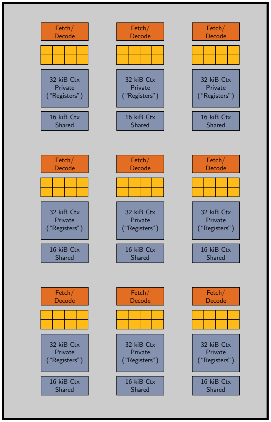
Connection: Hardware ↔ Programming Model





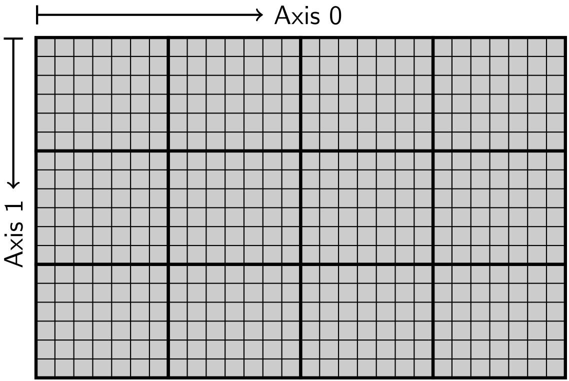
Connection: Hardware ↔ Programming Model



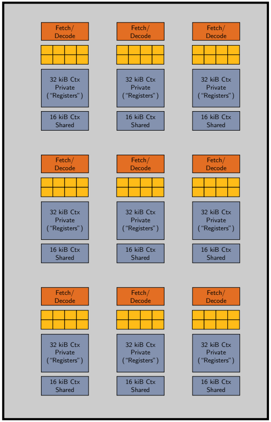


Hardware

Connection: Hardware ↔ Programming Model

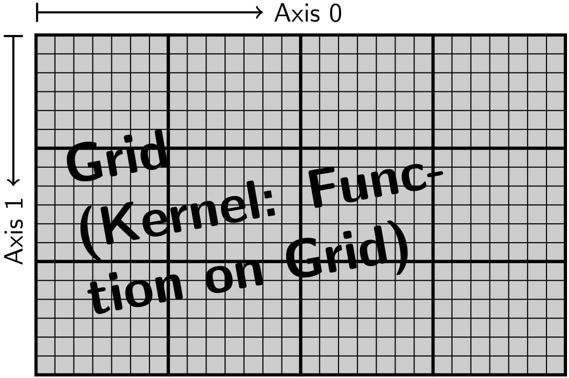


Software representation

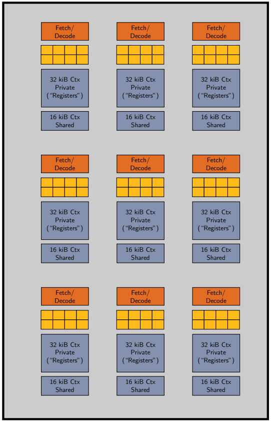


Hardware

Connection: Hardware ↔ Programming Model

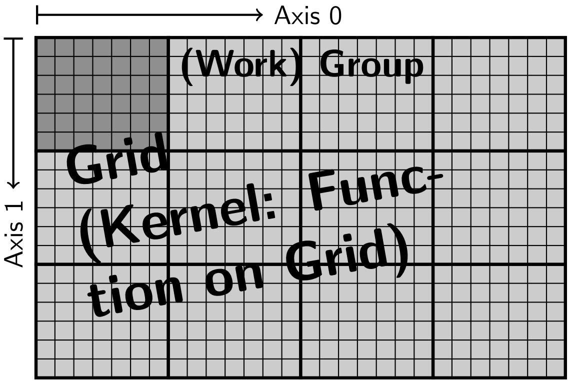


Software representation

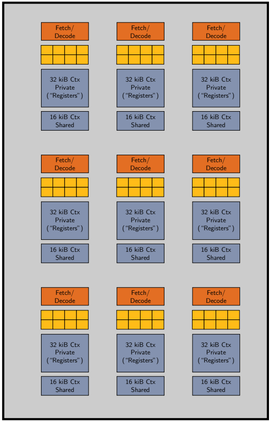


Hardware

Connection: Hardware ↔ Programming Model

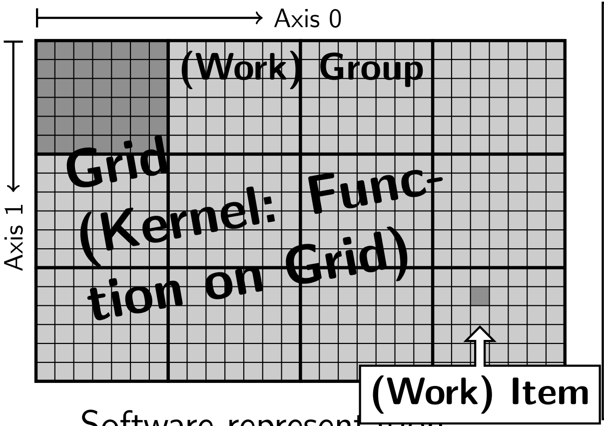


Software representation

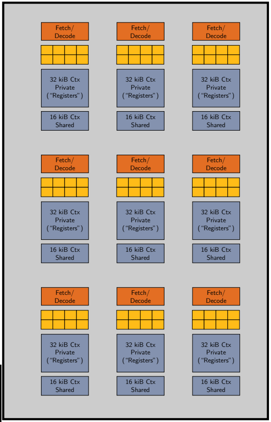


Hardware

Connection: Hardware ↔ Programming Model

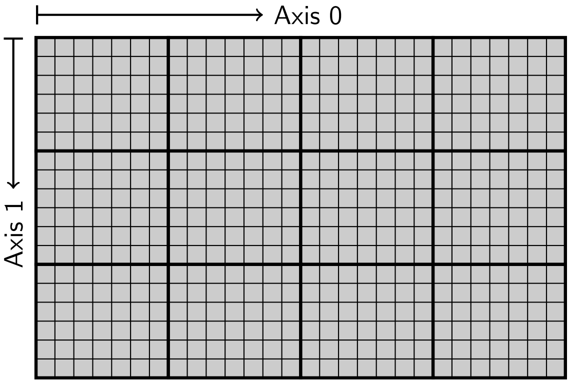


Software representation

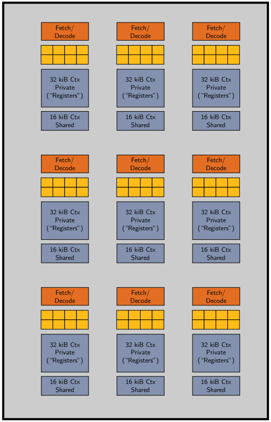


Hardware

Connection: Hardware ↔ Programming Model

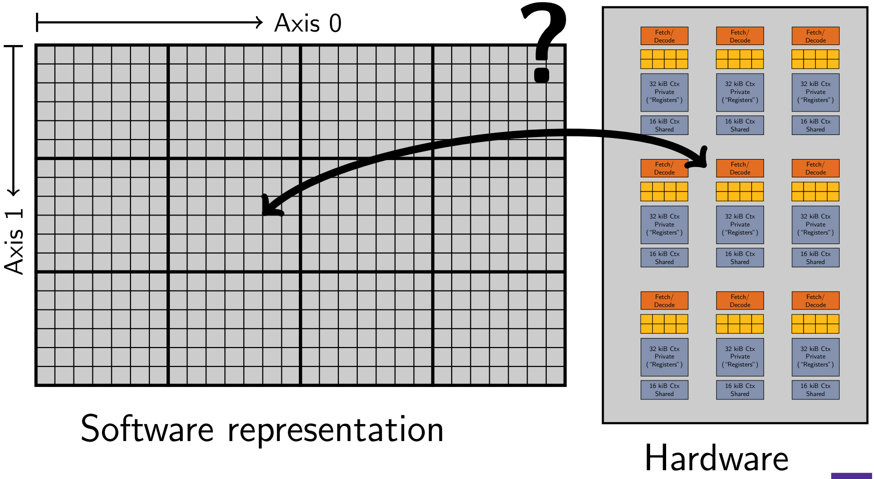


Software representation

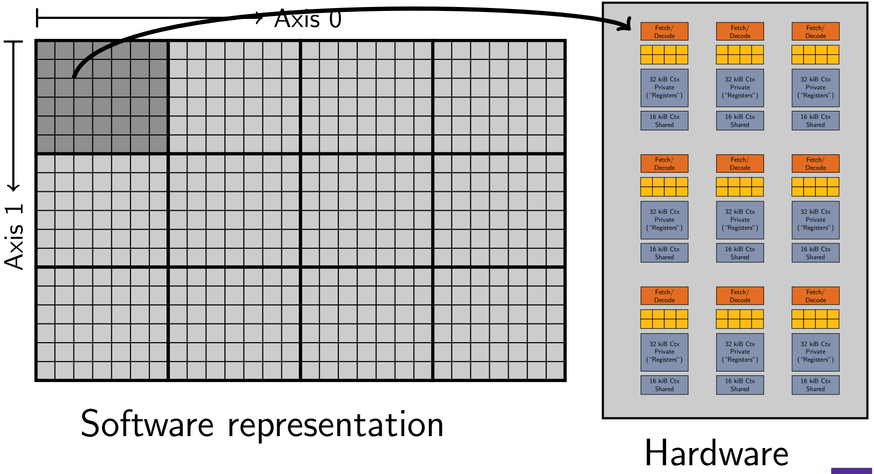


Hardware

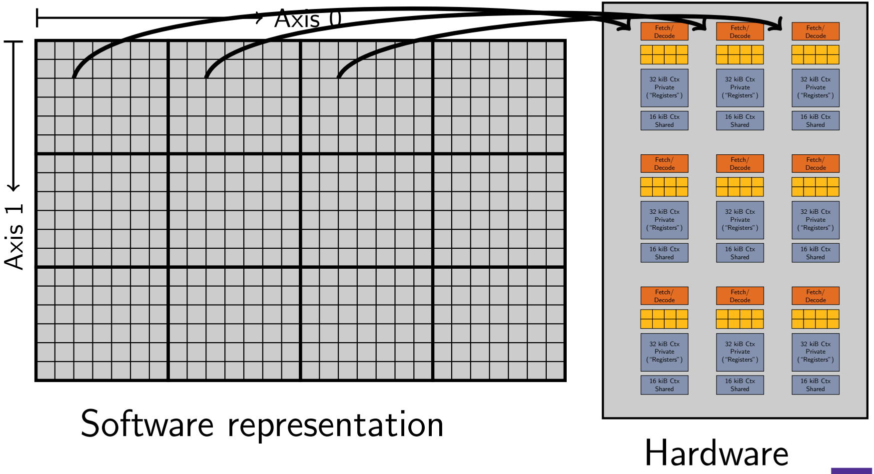
Connection: Hardware ↔ Programming Model



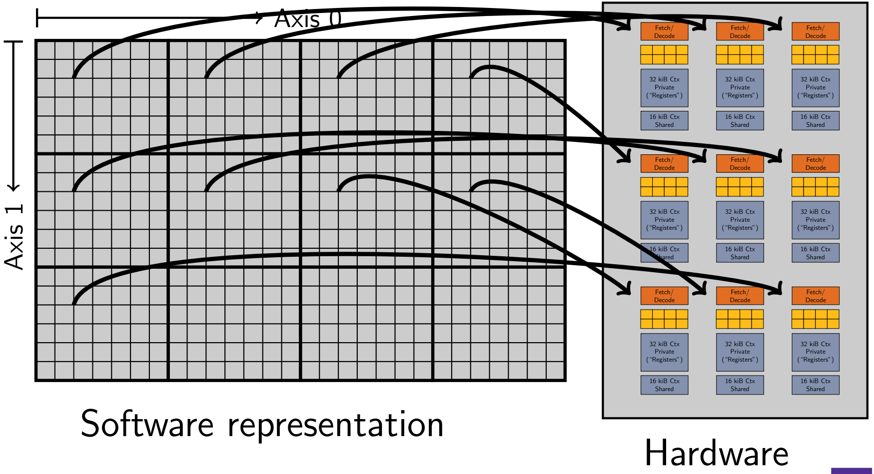
Connection: Hardware ↔ Programming Model



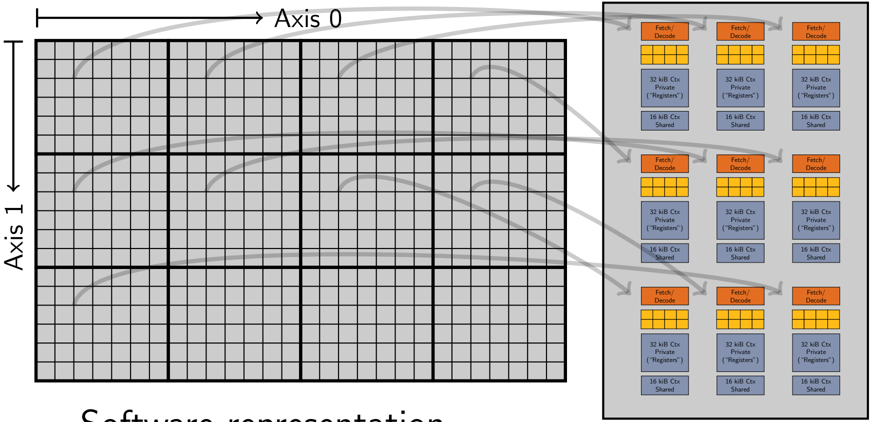
Connection: Hardware ↔ Programming Model



Connection: Hardware ↔ Programming Model



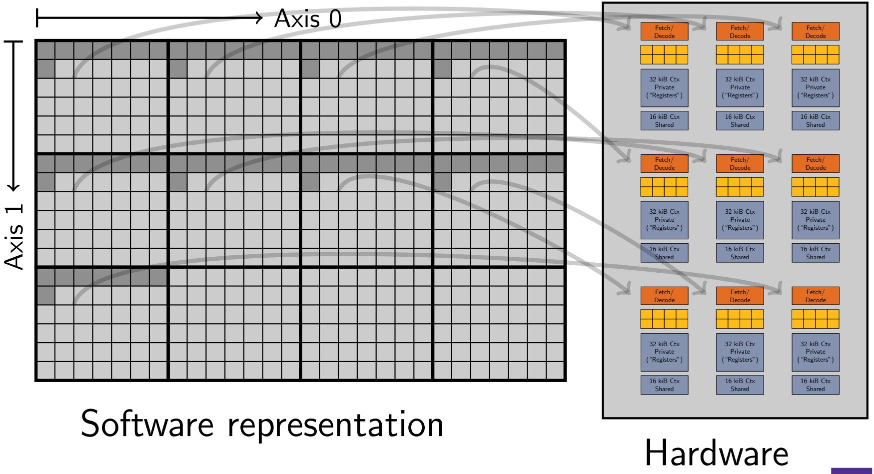
Connection: Hardware ↔ Programming Model



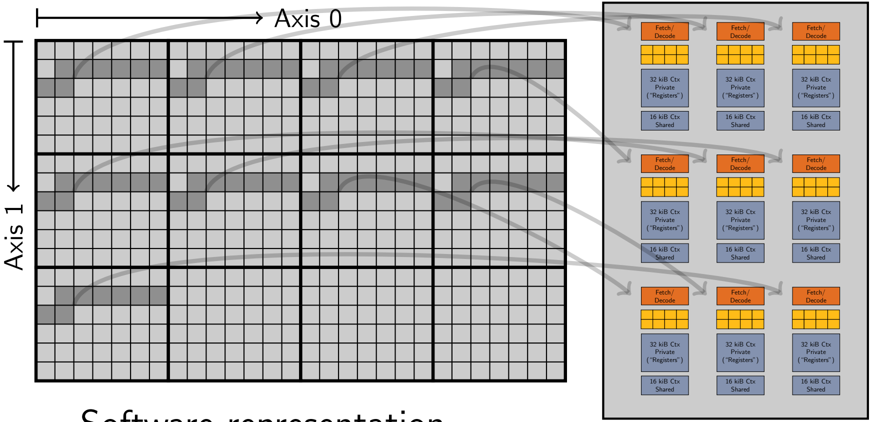
Software representation

Hardware

Connection: Hardware $\leftrightarrow$ Programming Model



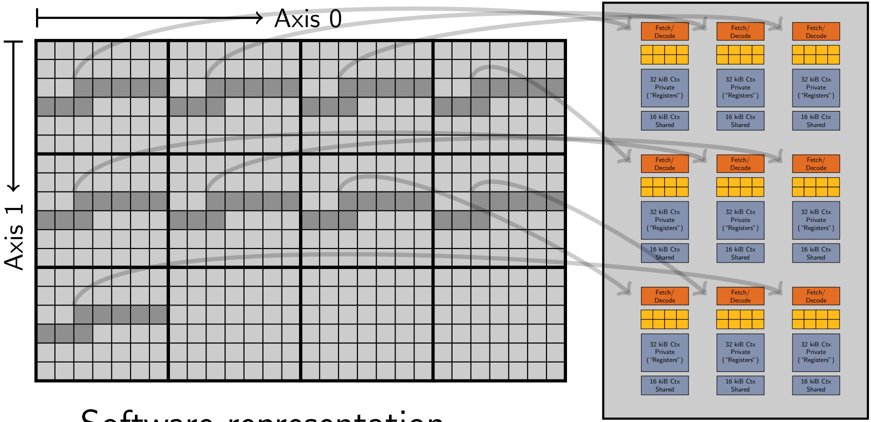
Connection: Hardware ↔ Programming Model



Software representation

Hardware

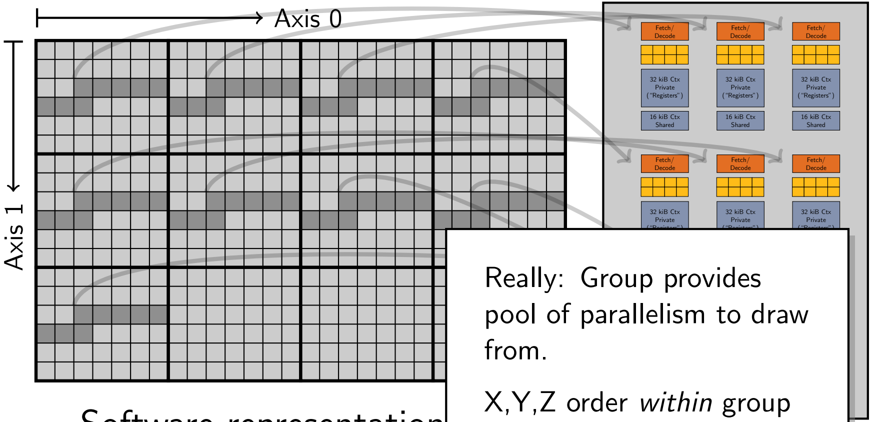
Connection: Hardware ↔ Programming Model



Software representation

Hardware

Connection: Hardware ↔ Programming Model

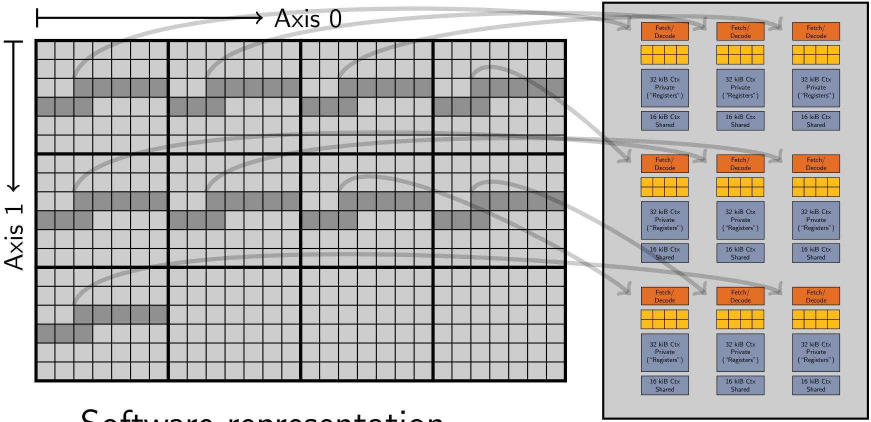


Software representation

Really: Group provides pool of parallelism to draw from.

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

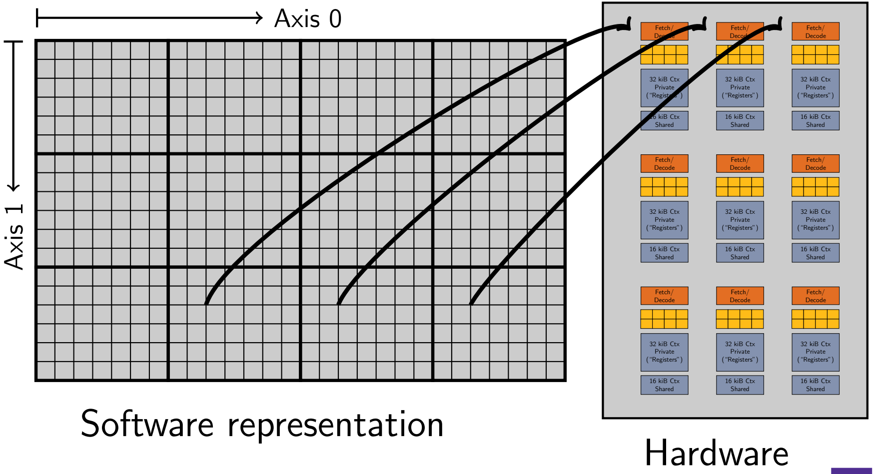
Connection: Hardware ↔ Programming Model



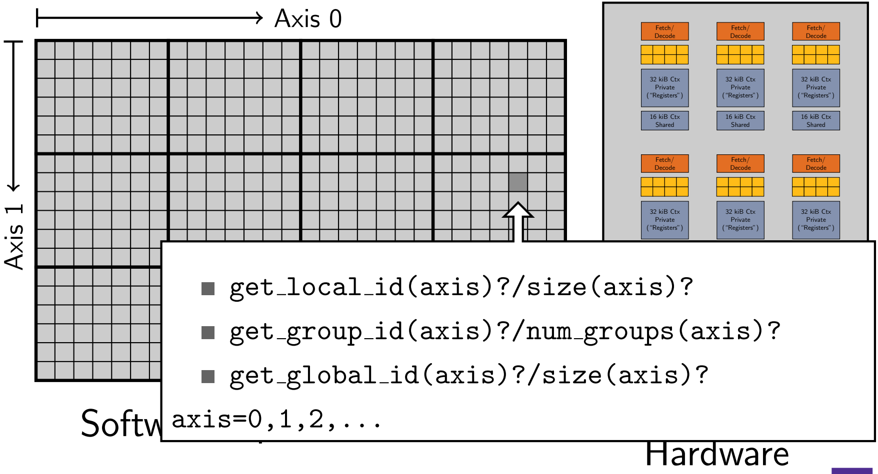
Software representation

Hardware

Connection: Hardware ↔ Programming Model



Connection: Hardware ↔ Programming Model

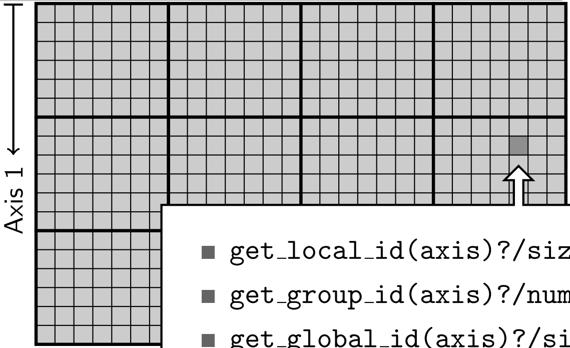


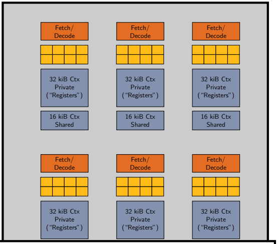
- get_local_id(axis)?/size(axis)?
- get_group_id(axis)?/num_groups(axis)?
- get_global_id(axis)?/size(axis)?

axis=0,1,2,...

Connection: Hardware ↔ Programming 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,...`

Softw

Hardware

GPU architecture: Overview

Now know about basic execution model.

Observe: Same model also applies to multi-core CPUs!

$\rightarrow$ the "OpenCL" execution model

Will learn more about GPUs later. In particular:

- Memory access
- Device Management
- Synchronization

Note: CPUs have a very different memory system.

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
   - Python, Numpy
   - GPUs
   - OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
4. OpenCL implementations

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 (!)

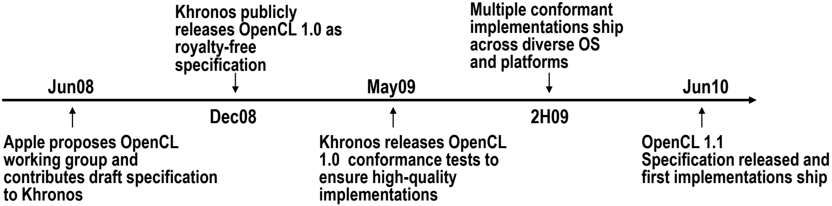
Who?

- **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

Credit: Khronos Group

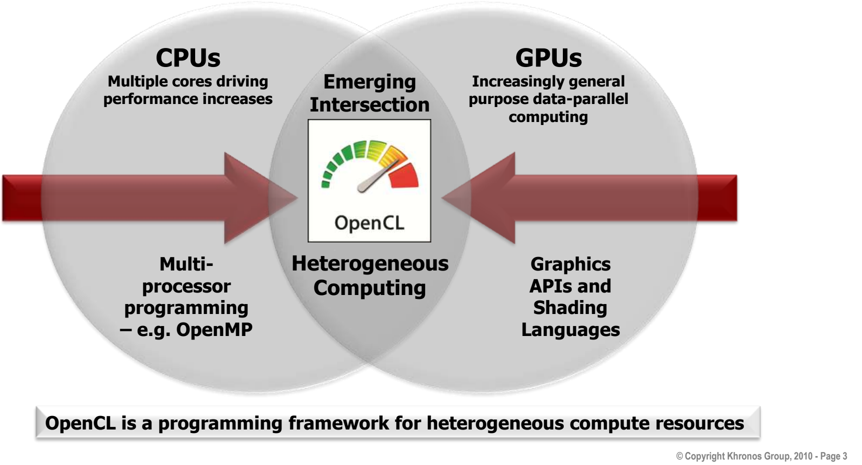
When?

- **Six months from proposal to released OpenCL 1.0 specification**
  - Due to a strong initial proposal and a shared commercial incentive
- **Multiple conformant implementations shipping**
  - Apple's Mac OS X Snow Leopard now ships with OpenCL
- **18 month cadence between OpenCL 1.0 and OpenCL 1.1**
  - Backwards compatibility protect software investment



Credit: Khronos Group

Why?



Credit: Khronos Group

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];
}
```

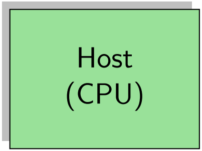
OpenCL source code:

```
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)];
}
```

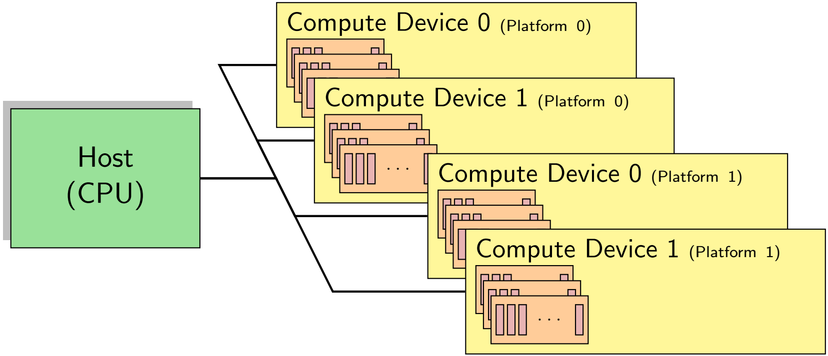
OpenCL ↔ CUDA: A dictionary

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)

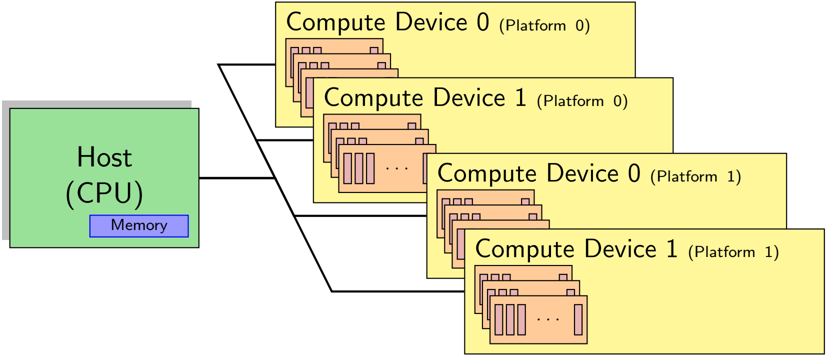
OpenCL: Computing as a Service



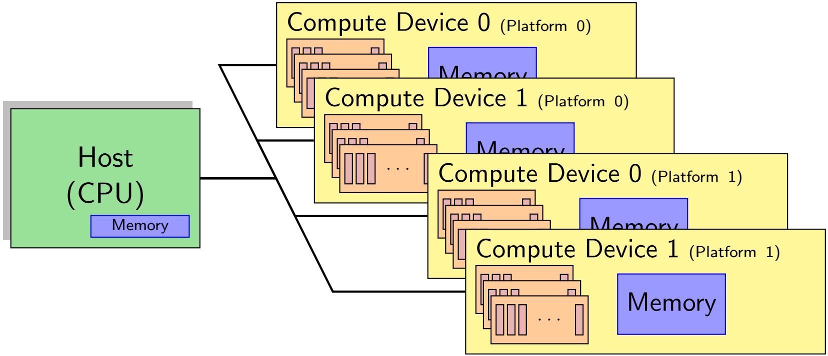
OpenCL: Computing as a Service



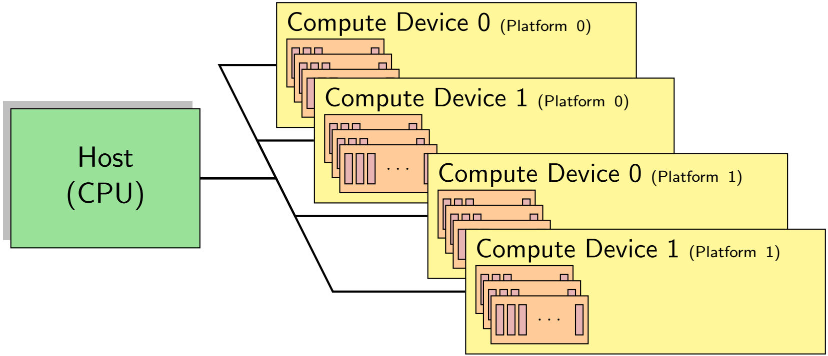
OpenCL: Computing as a Service



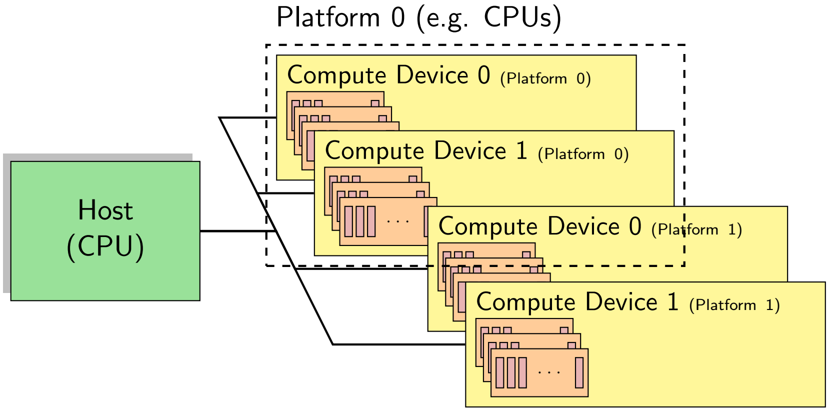
OpenCL: Computing as a Service



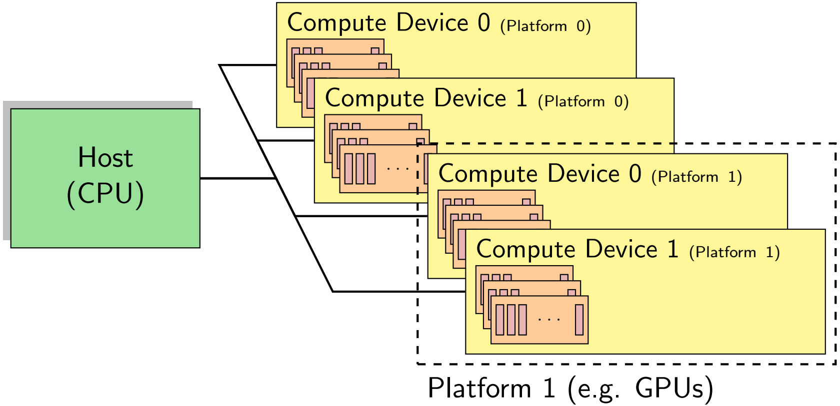
OpenCL: Computing as a Service



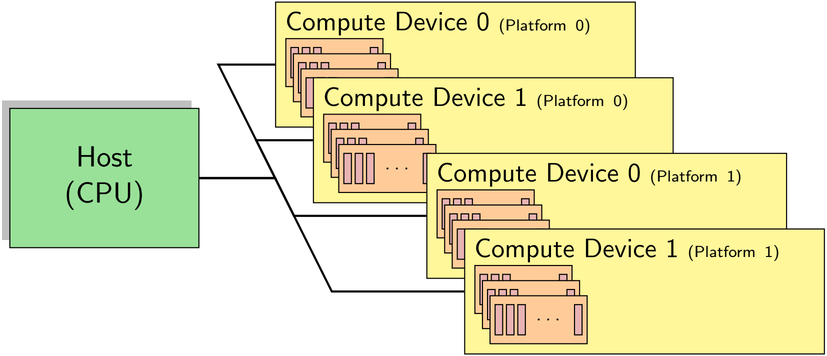
OpenCL: Computing as a Service



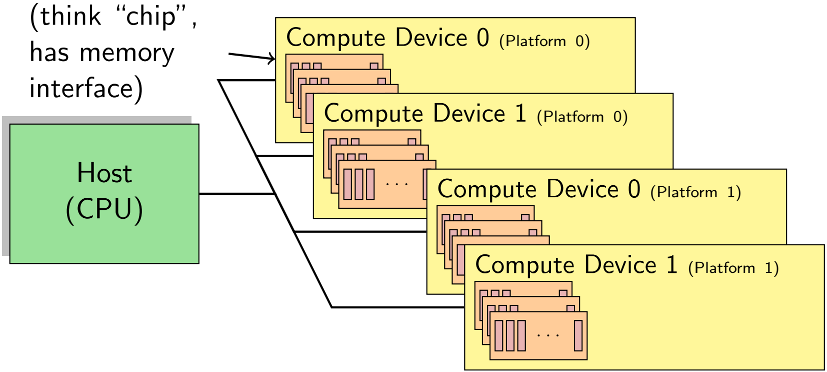
OpenCL: Computing as a Service



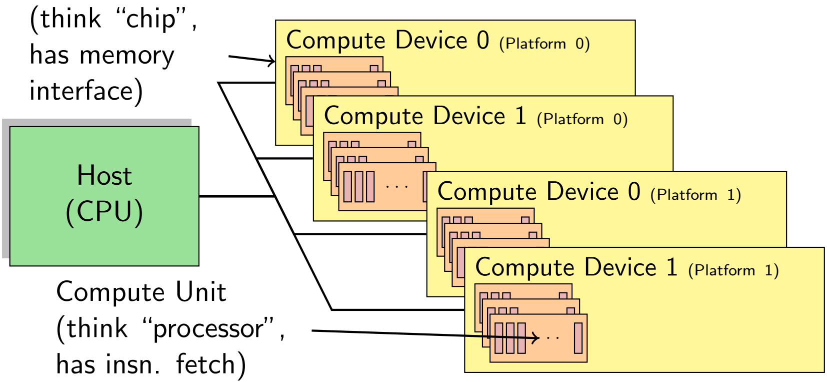
OpenCL: Computing as a Service



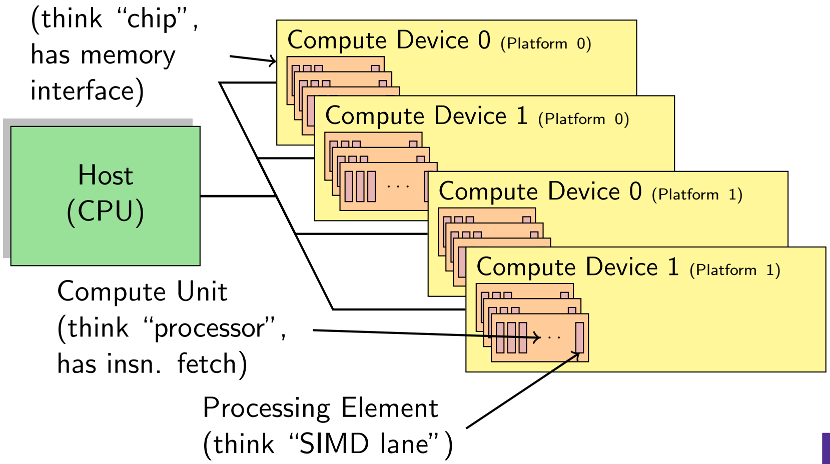
OpenCL: Computing as a Service



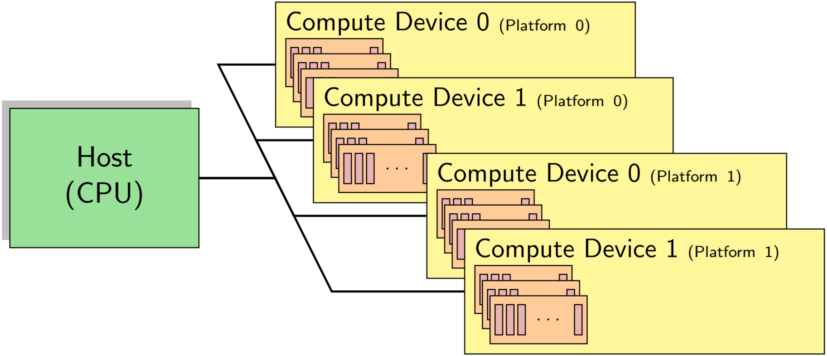
OpenCL: Computing as a Service



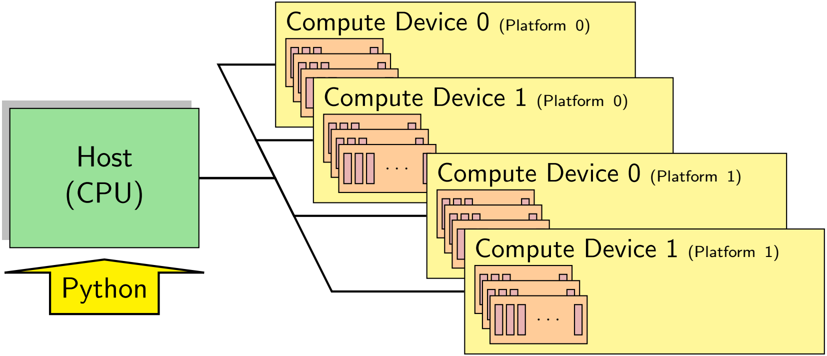
OpenCL: Computing as a Service



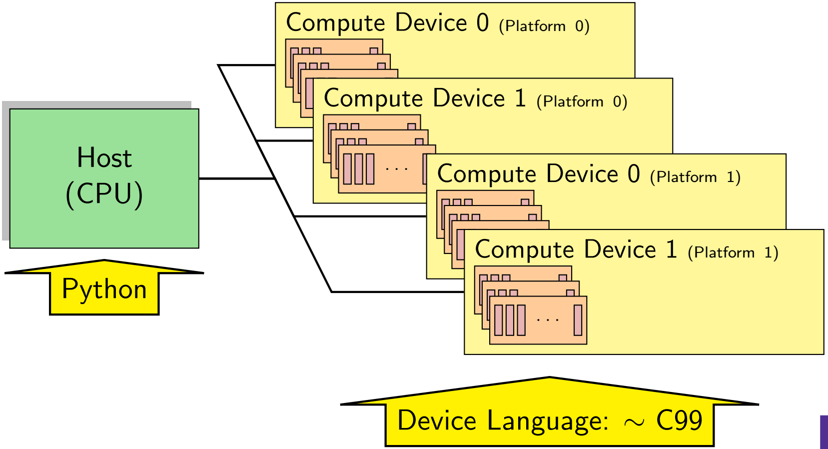
OpenCL: Computing as a Service



OpenCL: Computing as a Service



OpenCL: Computing as a Service



Why do Scripting for GPUs?

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

$\rightarrow$ complement each other

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

Outline

1 Intro: Python, Numpy, GPUs, OpenCL

2 GPU Programming with PyOpenCL
- First Contact
- About PyOpenCL

3 OpenCL viewed from Python

4 OpenCL implementations

Outline

1 Intro: Python, Numpy, GPUs, OpenCL

2 GPU Programming with PyOpenCL
- First Contact
- About PyOpenCL

3 OpenCL viewed from Python

4 OpenCL implementations

Dive into PyOpenCL

```python
import pyopencl as cl, numpy

a = numpy.random.rand(256**3).astype(numpy.float32)

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

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a)

prg = cl.Program(ctx, """
    __kernel void twice(__global float *a)
    { a[get_global_id(0)] *= 2; }
    """).build()

prg.twice(queue, a.shape, (1,), a_dev)
```

Dive into PyOpenCL

```
import pyopencl as cl , numpy

a = numpy.random.rand(256**3).astype(numpy.float32)

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

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a)

prg = cl.Program(ctx, """
    __kernel void twice(__global float *a)
    { a[get_global_id(0)] *= 2; }
    """).build()

prg.twice(queue, a.shape, (1,), a_dev)
```

Compute kernel

Dive into PyOpenCL: Getting Results

```
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a)

prg = cl.Program(ctx, """
    __kernel void twice(__global float *a)
    { a[get_global_id(0)] *= 2; }
    """).build()

prg.twice(queue, a.shape, (1,), a_dev)

result = numpy.empty_like(a)
cl.enqueue_copy(queue, result, a_dev)
import numpy.linalg as la
assert la.norm(result - 2*a) == 0
```

Dive into PyOpenCL: Grouping

```
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a)

prg = cl.Program(ctx, """
    __kernel void twice( __global float *a)
    { a[ get_local_id (0)+ get_local_size (0)*get_group_id (0)] *= 2; }
    """).build()

prg.twice(queue, a.shape, (256,), a_dev)

result = numpy.empty_like(a)
cl.enqueue_copy(queue, result, a_dev)
import numpy.linalg as la
assert la.norm(result - 2*a) == 0
```

Dive into PyOpenCL: Thinking on your feet

Thinking about GPU programming

How would we modify the program to. . .

Dive into PyOpenCL: Thinking on your feet

Thinking about GPU programming

How would we modify the program to...

1. ...compute $c_i = a_i b_i$?

Dive into PyOpenCL: Thinking on your feet

Thinking about GPU programming

How would we modify the program to...

1. ...compute $c_i = a_i b_i$?
2. ...use groups of 16×16 work items?

Dive into PyOpenCL: Thinking on your feet

Thinking about GPU programming

How would we modify the program to…

1. …compute $c_i = a_i b_i$?
2. …use groups of 16×16 work items?
3. …benchmark 1 work item per group against 256 work items per group? (Use `time.time()` and `.wait()`.)

Outline

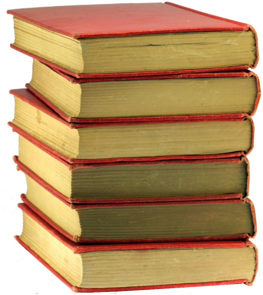
1. Intro: Python, Numpy, GPUs, OpenCL

2. GPU Programming with PyOpenCL
   - First Contact
   - About PyOpenCL

3. OpenCL viewed from Python

4. OpenCL implementations

PyOpenCL Philosophy



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

NYU

PyOpenCL: Completeness

PyOpenCL exposes *all* of OpenCL.

For example:

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

PyOpenCL: Completeness

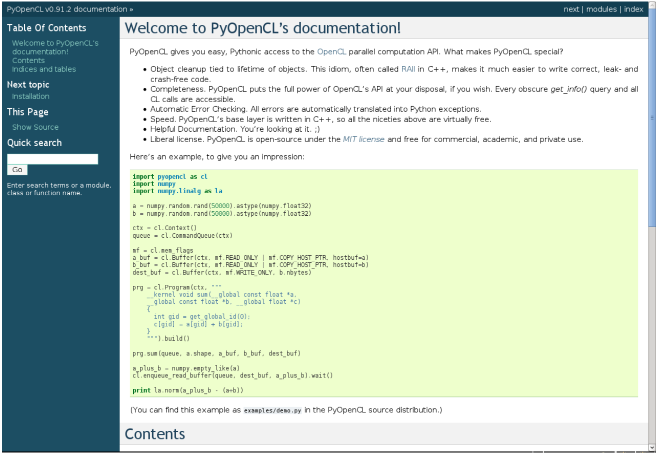
PyOpenCL supports (nearly) every OS that has an OpenCL implementation.

- Linux
- OS X
- Windows

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. (based on OpenCL's reference counting)

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:

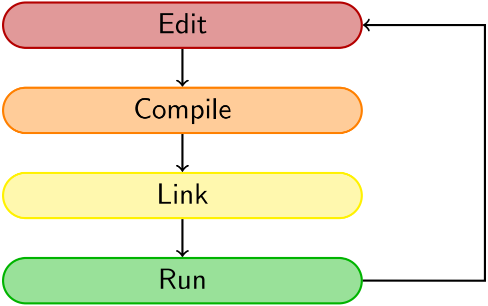
```python
import pyopencl as cl
import numpy
import numpy.linalg as la

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

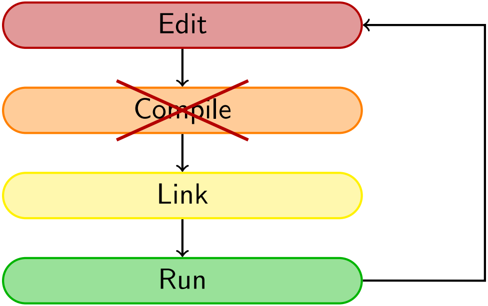
Scripting: Interpreted, not Compiled

Program creation workflow:



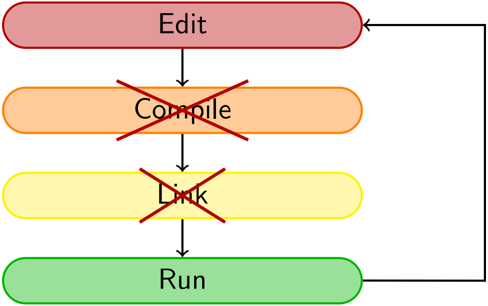
Scripting: Interpreted, not Compiled

Program creation workflow:

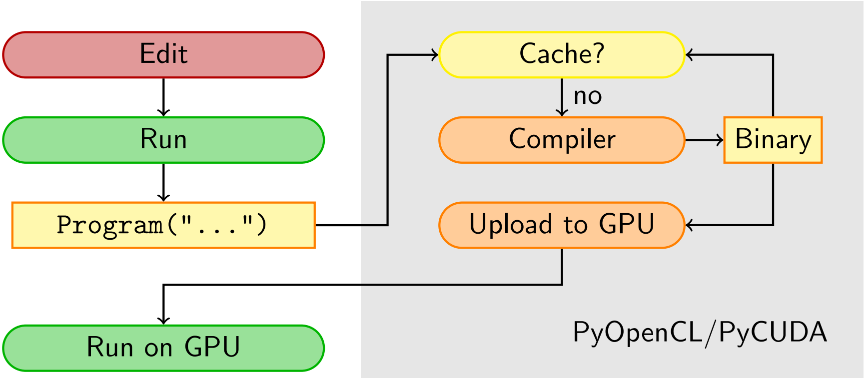


Scripting: Interpreted, not Compiled

Program creation workflow:



PyOpenCL, PyCUDA: Workflow



PyOpenCL: Vital Information

- `http://mathema.tician.de/software/pyopencl`
  - Downloaded 30k+ times
- Complete documentation
- MIT License
  (no warranty, free for all use)
- Requires: numpy, Python 2.4+.
- Community: mailing list, wiki
- Add-on packages (e.g. PyFFT, Sailfish, PyWENO)

An Appetizer

Remember your first PyOpenCL program?

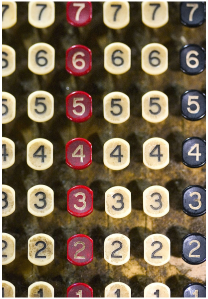
Abstraction is good:

```
import numpy
import pyopencl as cl
import pyopencl.array as cl_array

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

a_gpu = cl_array.to_device(
        ctx, queue, numpy.random.randn(4,4).astype(numpy.float32))
a_doubled = (2*a_gpu).get()
print a_doubled
print a_gpu
```

pyopencl.array: Simple Linear Algebra

pyopencl.array.Array:

- Meant to look and feel just like numpy.
  - p.a.to_device(ctx, queue, numpy_array)
  - numpy_array = ary.get()
- +, -, ∗, /, fill, sin, arange, exp, rand, . . .
- Mixed types (int32 + float32 = float64)
- print cl_array for debugging.
- Allows access to raw bits
  - Use as kernel arguments, memory maps



pyopencl.elementwise: Elementwise expressions

Avoiding extra store-fetch cycles for elementwise math:

```python
n = 10000
a_gpu = cl_array.to_device(
        ctx, queue, numpy.random.randn(n).astype(numpy.float32))
b_gpu = cl_array.to_device(
        ctx, queue, numpy.random.randn(n).astype(numpy.float32))

from pyopencl.elementwise import ElementwiseKernel
lin_comb = ElementwiseKernel(ctx,
        "float a, float *x, float b, float *y, float *z",
        "z[i] = a*x[i] + b*y[i]")

c_gpu = cl_array.empty_like(a_gpu)
lin_comb(5, a_gpu, 6, b_gpu, c_gpu)

import numpy.linalg as la
assert la.norm((c_gpu - (5*a_gpu+6*b_gpu)).get()) < 1e-5
```

pyopencl.reduction: Reduction made easy

Example: A dot product calculation

```
from pyopencl.reduction import ReductionKernel
dot = ReductionKernel(ctx, dtype_out=numpy.float32, neutral="0",
    reduce_expr="a+b", map_expr="x[i]*y[i]",
    arguments="__global const float *x, __global const float *y")

import pyopencl.clrandom as cl_rand
x = cl_rand.rand(ctx, queue, (1000*1000), dtype=numpy.float32)
y = cl_rand.rand(ctx, queue, (1000*1000), dtype=numpy.float32)

x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())
```

pyopencl.scan: Scan made easy

Example: A cumulative sum computation

```
from pyopencl.scan import InclusiveScanKernel
knl = InclusiveScanKernel(ctx, np.int32, "a+b")

n = 2**20-2**18+5
host_data = np.random.randint(0, 10, n).astype(np.int32)
dev_data = cl_array.to_device(queue, host_data)

knl(dev_data)
assert (dev_data.get() == np.cumsum(host_data, axis=0)).all()
```

Questions?

?

Outline

1 Intro: Python, Numpy, GPUs, OpenCL

2 GPU Programming with PyOpenCL

3 OpenCL viewed from Python
- Device Language
- The OpenCL runtime
- Synchronization
- Extensions

4 OpenCL implementations

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?

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
   - Device Language
   - The OpenCL runtime
   - Synchronization
   - Extensions
4. OpenCL implementations

OpenCL Device Language

OpenCL device language is C99, with these differences:

- ➕ Index getters
- ➕ Memory space qualifiers
- ➕ Vector data types
- ➕ Many generic ('overloaded') math functions including fast `native_...` varieties.
- ➕ Synchronization
- ➖ Recursion
- ➖ `malloc()`

Address Space Qualifiers

Type	Per	Access	Latency	
private	work item	R/W	1 or 1000	
local	group	R/W	2	
global	grid	R/W	1000	Cached?
`constant`	grid	R/O	1-1000	Cached
image*n*d_t	grid	R(/W)	1000	Spatially cached

Address Space Qualifiers

Type	Per	Access	Latency	
private	work item	R/W	1 or 1000	
local	group	R/W	2	
global	grid	R/W	1000	Cached?
`constant`	grid	R/O	1-1000	Cached
image*n*d_t	grid	R(/W)	1000	Spatially cached

Address Space Qualifiers

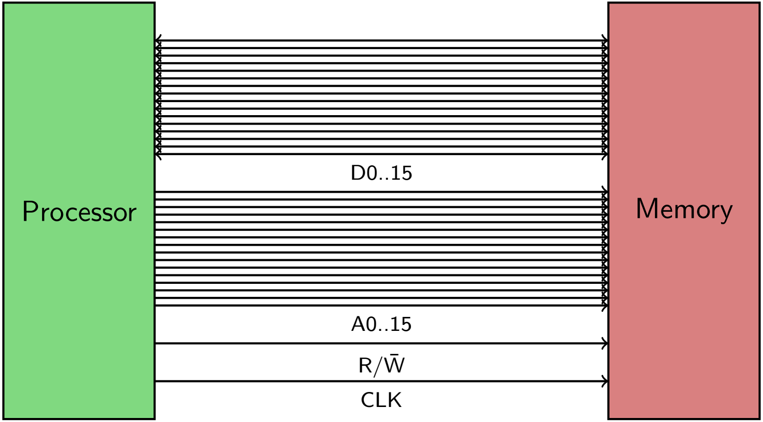
Type	Per	Access	Latency	
private	work item	R/W	1 or 1000	
local	group	R/W	2	
global	grid	R/W	1000	Cached?
`constant`	grid	R/O	1-1000	Cached
image*n*d_t	grid	R(/W)	1000	Spatially cached

Important

Different types of memory are good at different types of access. Successful algorithms combine many types' strengths.

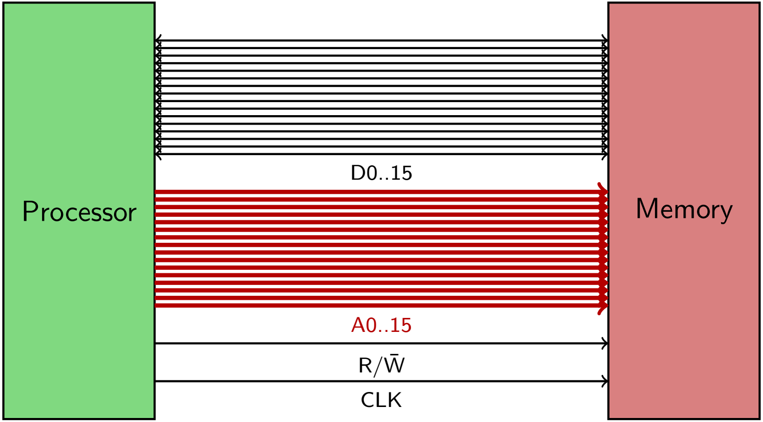
How does computer memory work?

One (reading) memory transaction (simplified):



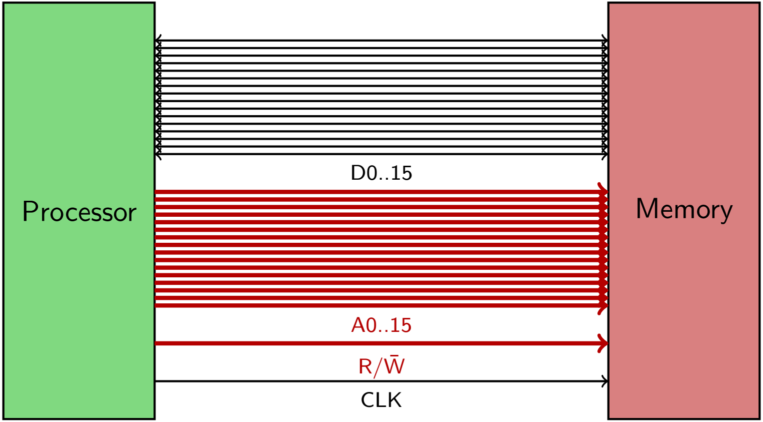
How does computer memory work?

One (reading) memory transaction (simplified):



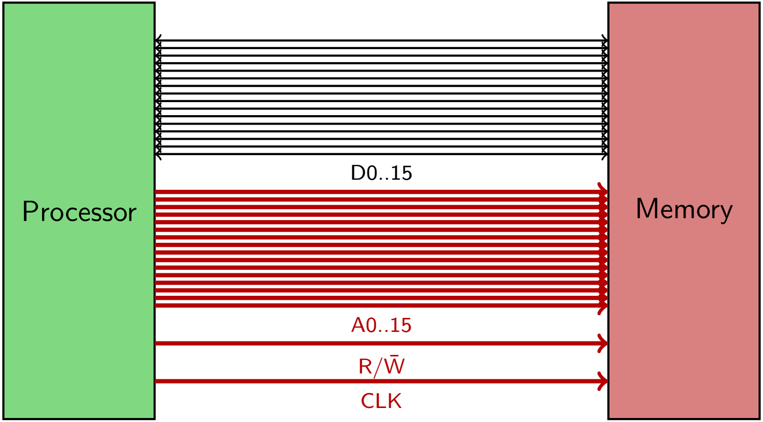
How does computer memory work?

One (reading) memory transaction (simplified):



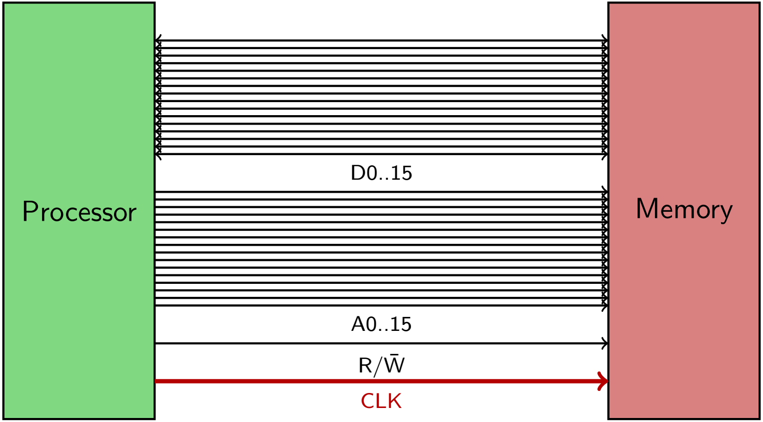
How does computer memory work?

One (reading) memory transaction (simplified):



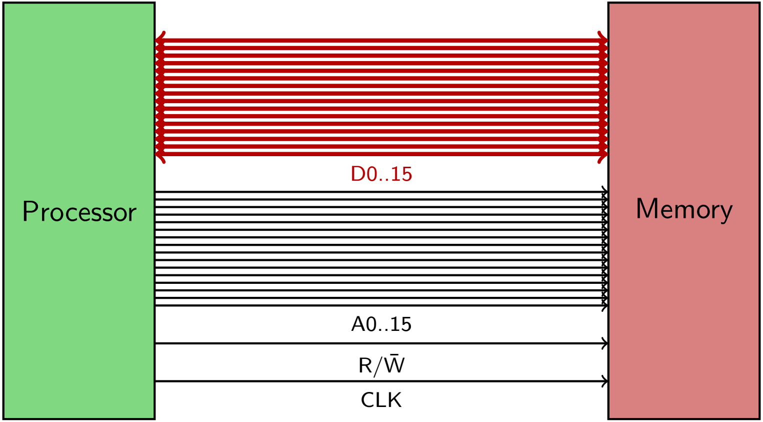
How does computer memory work?

One (reading) memory transaction (simplified):



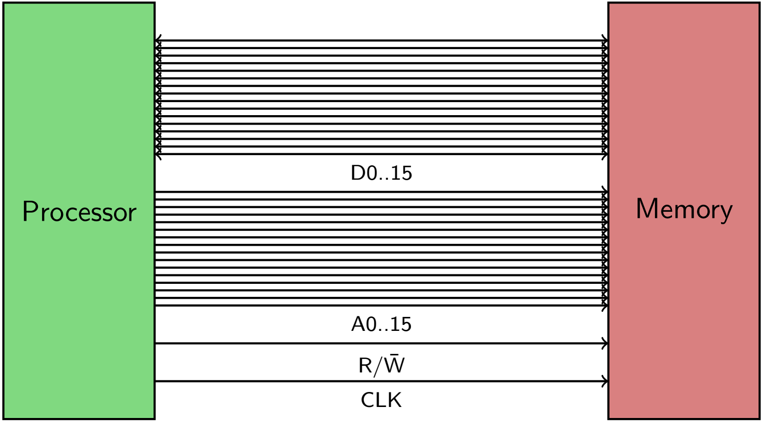
How does computer memory work?

One (reading) memory transaction (simplified):



How does computer memory work?

One (reading) memory transaction (simplified):



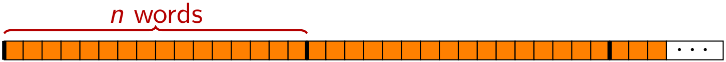
Observation: Access (and addressing) happens in bus-width-size "chunks".

Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.

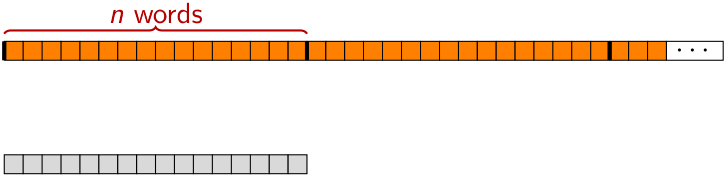


Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.

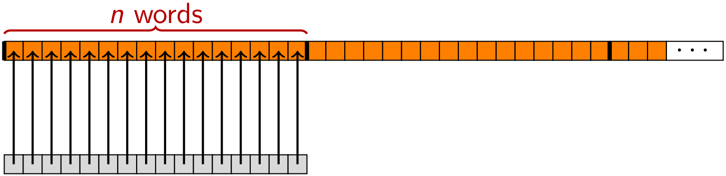


Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.



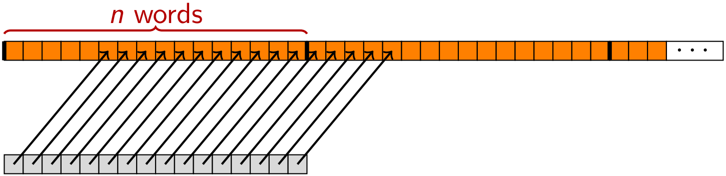
OK: `global_variable[get_global_id(0)]`
(Single transaction)

Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.



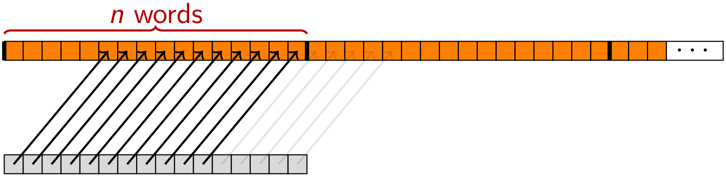
Bad: `global_variable[5+get_global_id(0)]`
(Two transactions)

Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.

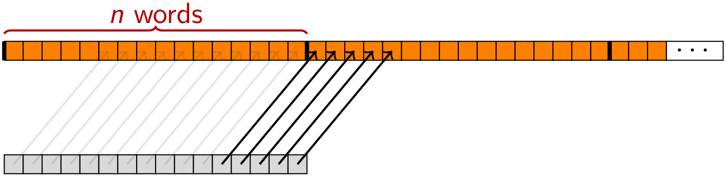


Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.

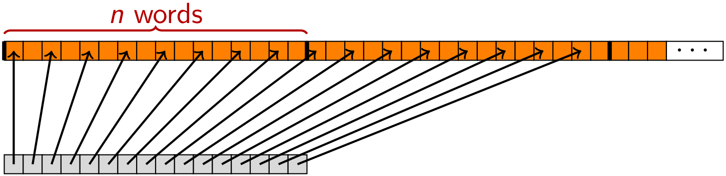


Global Memory

Rule of thumb

$$n = \min\left(\frac{\text{Bus width in bits}}{\text{Word size in bits}}, \text{SIMD group size}\right)$$

work items access global memory simultaneously. Full utilization only if all bits in bus transaction are useful.



Bad: `global_variable[2*get_global_id(0)]`
(Two transactions)

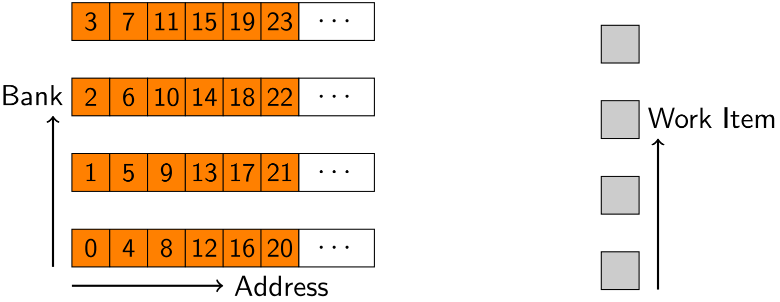
Making sense of Global Memory

Consider the following examples:

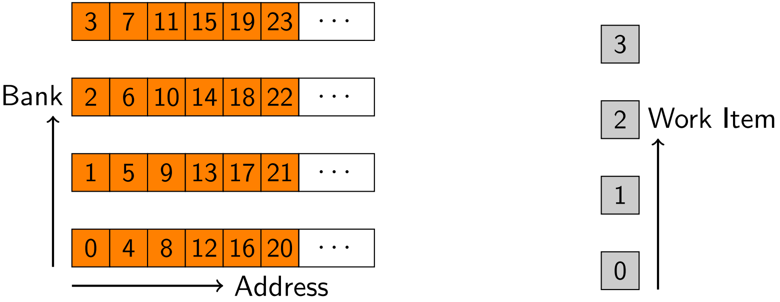
- List of XYZ vectors:
  - XXXX...YYYY...ZZZZ...(“SoA”)
  - XYZXYZXYZ...(“AoS”)
- Accessing a row-major (C order) matrix
  - by rows
  - by columns

Local Memory: Banking

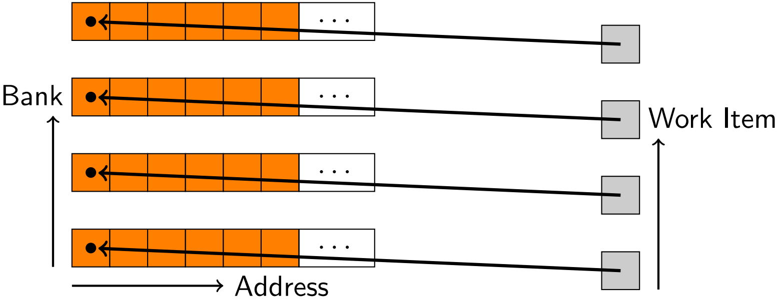
Local Memory: Banking



Local Memory: Banking

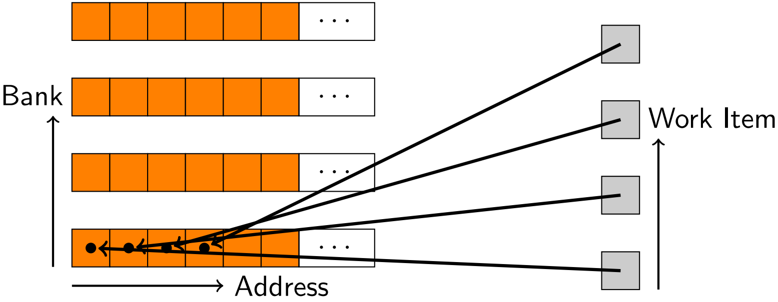


Local Memory: Banking



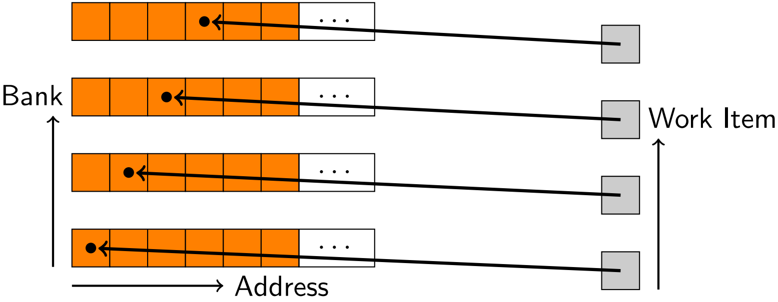
OK: `local_variable[get_local_id(0)]`,
(Single cycle)

Local Memory: Banking



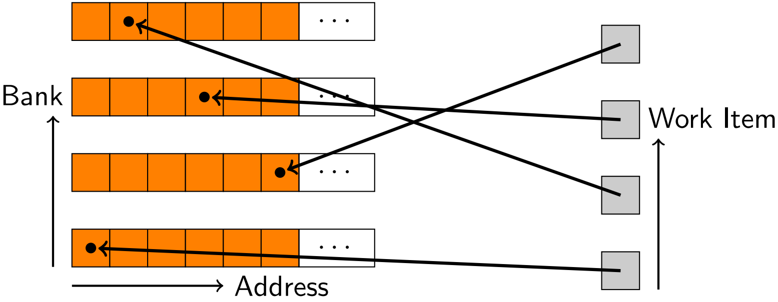
Bad: `local_variable[BANK_COUNT*get_local_id(0)]`
(`BANK_COUNT` cycles)

Local Memory: Banking



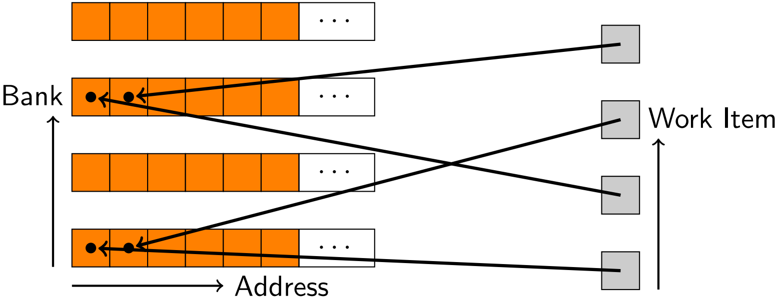
OK: `local_variable[(BANK_COUNT+1)*get_local_id(0)]`
(Single cycle)

Local Memory: Banking



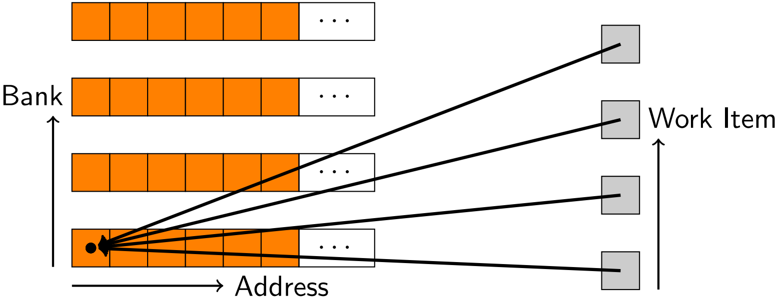
OK: `local_variable[ODD_NUMBER*get_local_id(0)]`
(Single cycle)

Local Memory: Banking



Bad: `local_variable[2*get_local_id(0)]`
(`BANK_COUNT`/2 cycles)

Local Memory: Banking



OK: `local_variable[f(get_group_id(0))]`
(Broadcast–single cycle)

Local Memory: Banking

Example: Nvidia GT200 has 16 banks.
Work items access local memory in groups of 16.

CL vector data types

`float`*n* `vec` (n=1,2,3,4,8,16) (also for `double` and integer types) Components:

- `vec.s012...abcdef` (or `xyzw`)
- `vec.s3120` (Swizzling)
- `vec.s024 = (float3)(1,2,3);` (Lvalue, Literals)

Usage:

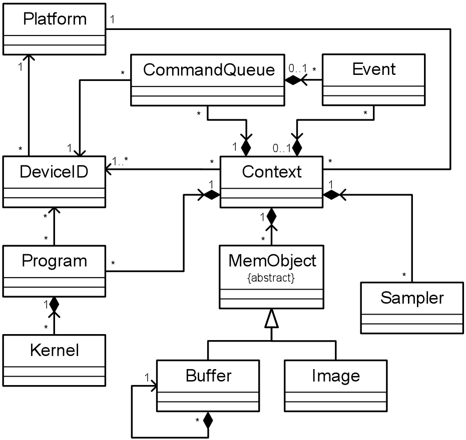
- Elementwise operations (+,-,`sin` (generic!),...)
- `float`*n* `vload`*n*`/vstore`*n*`(offset, float *)` (aligned!)
- `dot/distance`

Using CPU implementation: One of the sanest ways of using SSE/vector intrinsics!

Outline

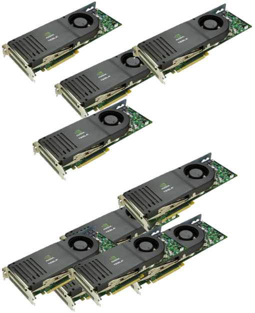
1. Intro: Python, Numpy, GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
   - Device Language
   - The OpenCL runtime
   - Synchronization
   - Extensions
4. OpenCL implementations

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.

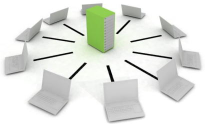
CL "Compute Device"

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)
```

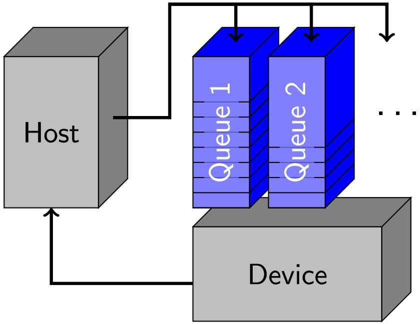


- Spans one or more Devices
- Create from device type or list of devices
  - See docs for `cl.Platform`, `cl.Device`
- `dev_type`: *`DEFAULT`*, `ALL`, `CPU`, `GPU`
- Needed to...
  - ...allocate Memory Objects
  - ...create and build Programs
  - ...host Command Queues
  - ...execute Grids

NYU

OpenCL: Command Queues

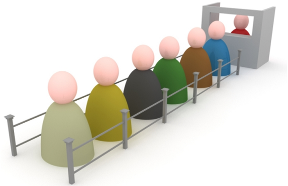
- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for drained queue
- Profiling



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



Building Blocks in Action

```python
import pyopencl as cl

platforms = cl.get_platforms()
my_platform = platforms[0]
print my_platform.vendor

devices = my_platform.get_devices()
my_device = devices[0]
print my_device.name

ctx = cl.Context([my_device])

cpq = cl.command_queue_properties
queue = cl.CommandQueue(ctx, my_device, cpq.PROFILING_ENABLE)
```

Simple version:

```python
ctx2 = cl.create_some_context()
queue2 = cl.CommandQueue(ctx)
```

Command Queues and Events

```
event = cl.enqueue_XXX(queue, ..., wait_for=[evt1, evt2])
```

Every enqueue operation returns an *Event*.

Also possible: Operation-less events ("Markers")

- Wait (`evt.wait()`), polling
- Specify dependencies

  Every enqueue operation takes a list arg `wait_for` of dependencies.
- Profile

  `event.profile....`
  - `QUEUED`, `SUBMIT`
  - `START`, `END`

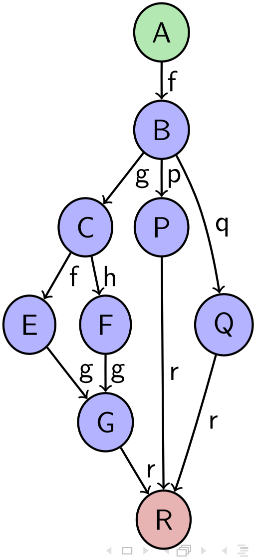
  (time stamp in ns)

Profiling example

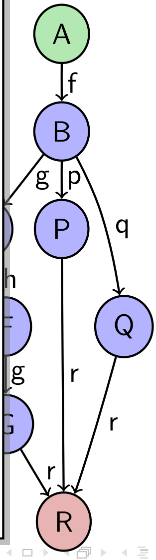
```
start_event = cl.enqueue_marker(queue)

# enqueue some commands

stop_event = cl.enqueue_marker(queue)
stop_event.wait()

elapsed_seconds = 1e-9*(
        start_event . profile .END - start_event. profile .END)

# --- OR ---

op_event = knl(queue, global_size , grp_size , args ...)
op_event.wait()
elapsed_seconds = 1e-9*(
        op_event. profile .END - op_event.profile.START)
```

Capturing Dependencies

$$
\begin{aligned}
B &= f(A)\\
C &= g(B)\\
E &= f(C)\\
F &= h(C)\\
G &= g(E,F)\\
P &= p(B)\\
Q &= q(B)\\
R &= r(G,P,Q)
\end{aligned}
$$



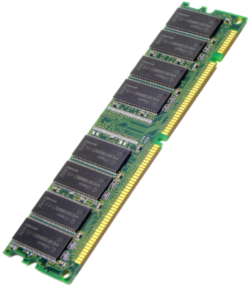
- Switch queue to out-of-order mode!
- Specify as list of events using `wait_for=` optional keyword to `enqueue_XXX`.
- Can also enqueue barrier.
- Common use case: Transmit/receive from other MPI ranks.
- Possible in hardware on Nv Fermi, AMD Cayman: Submit parallel work to increase machine use.
  -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

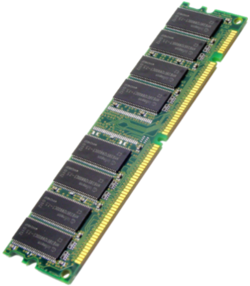
- Use `hostbuf` as initial content of buffer

`USE_HOST_PTR`:

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

`ALLOC_HOST_PTR`:

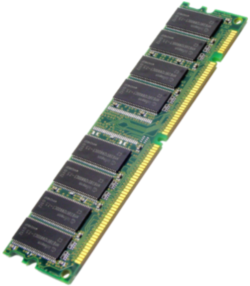
- *New* host memory (unrelated to `hostbuf`) is visible from device *and* host.



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`.



Command Queues and Buffers: A Crashy Puzzle

✔ OK

```
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a,
        is_blocking=False)
```

Command Queues and Buffers: A Crashy Puzzle

✔ OK

```
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a,
        is_blocking=False)
```

✖ Crash

```
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_copy(queue, a_dev,
        numpy.random.rand(256**3).astype(numpy.float32),
        is_blocking=False)
```

Command Queues and Buffers: A Crashy Puzzle

✔ OK

```
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a,
        is_blocking=False)
```

✖ Crash

```
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_copy(queue, a_dev,
        numpy.random.rand(256**3).astype(numpy.float32),
        is_blocking=False)
```

✔ OK

```
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_copy(queue, a_dev,
        numpy.random.rand(256**3).astype(numpy.float32),
        is_blocking=True)
```

Command Queues and Buffers: A Crashy Puzzle

✔ OK (usually!)

```python
a = numpy.random.rand(256**3).astype(numpy.float32)
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_copy(queue, a_dev, a,
        is_blocking=False)
```

✖ Crash

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_copy(queue, a_dev,
        numpy.random.rand(256**3).astype(numpy.float32),
        is_blocking=False)
```

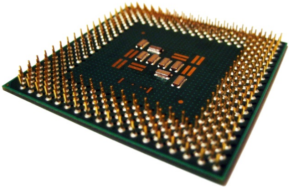
✔ OK

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_copy(queue, a_dev,
        numpy.random.rand(256**3).astype(numpy.float32),
        is_blocking=True)
```

Programs and Kernels

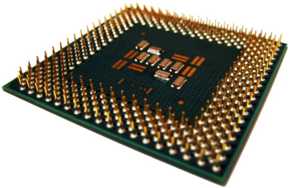
```
prg = cl.Program(context, src)
```

- `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,` (G_x, G_y, G_z), (L_x, L_y, L_z), `arg, ..., wait_for=None)`



Program Objects

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

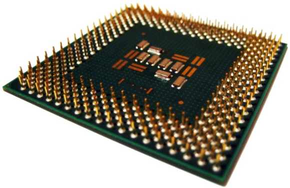


arg may be:

- `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:

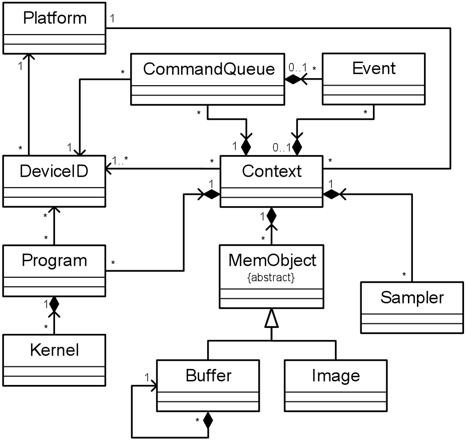
✖ Annoying, error-prone.

Better:

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

Use `None` for non-scalars.

OpenCL Object Diagram



Credit: Khronos Group

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
   - Device Language
   - The OpenCL runtime
   - Synchronization
   - Extensions
4. OpenCL implementations

NYU

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(...)`,
  `mem_fence(...)`
  `... =`
  `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:

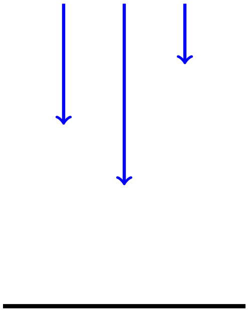
Results of the algorithm must be independent of the order in which work groups are executed.

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

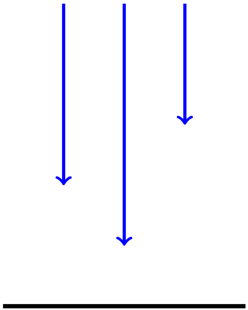
Synchronization

What is a Barrier?



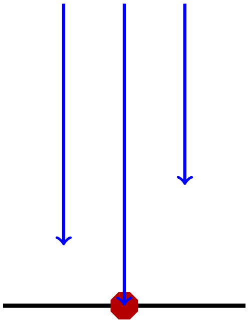
Synchronization

What is a Barrier?



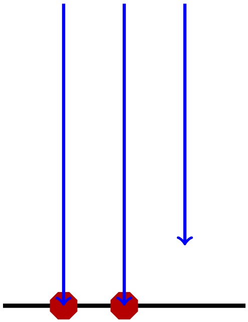
Synchronization

What is a Barrier?



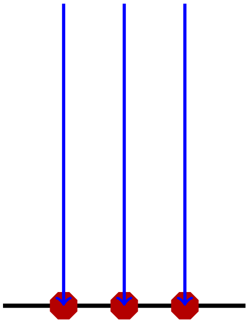
Synchronization

What is a Barrier?



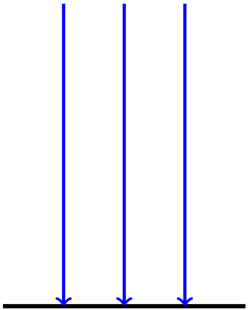
Synchronization

What is a Barrier?



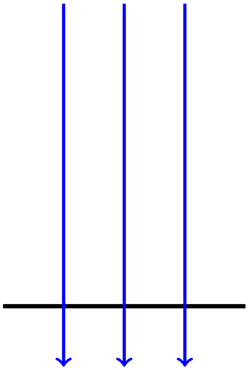
Synchronization

What is a Barrier?



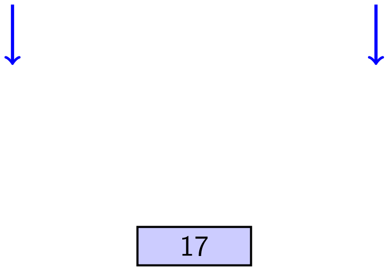
Synchronization

What is a Barrier?



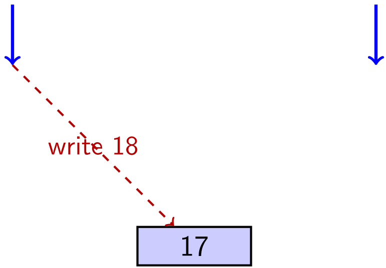
Synchronization

What is a Memory Fence?



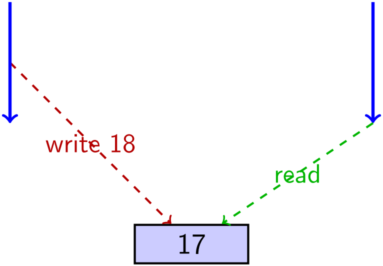
Synchronization

What is a Memory Fence?



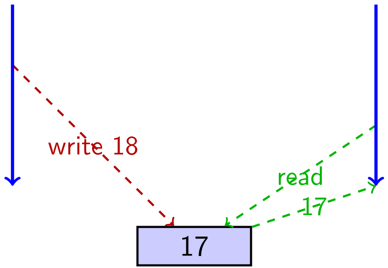
Synchronization

What is a Memory Fence?



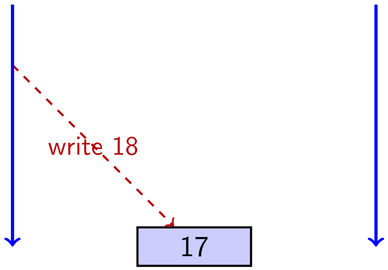
Synchronization

What is a Memory Fence?



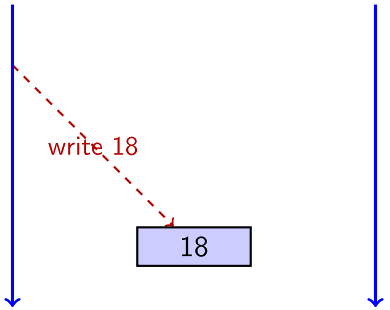
Synchronization

What is a Memory Fence?



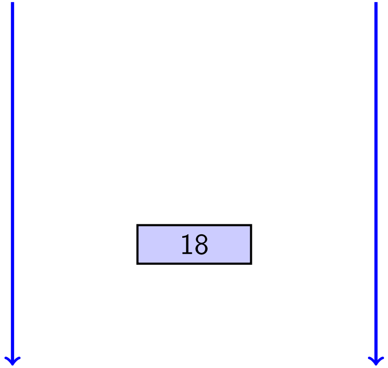
Synchronization

What is a Memory Fence?



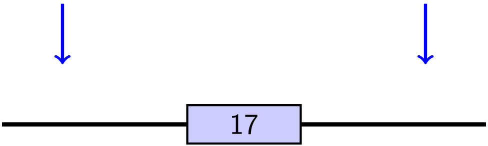
Synchronization

What is a Memory Fence?



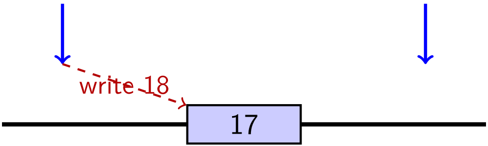
Synchronization

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



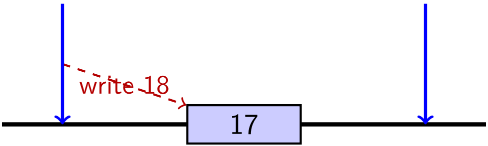
Synchronization

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



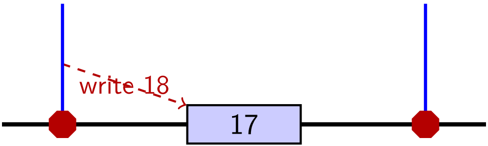
Synchronization

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



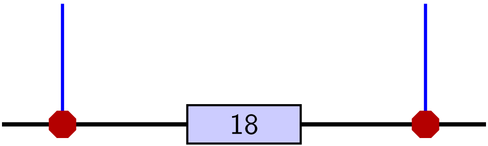
Synchronization

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



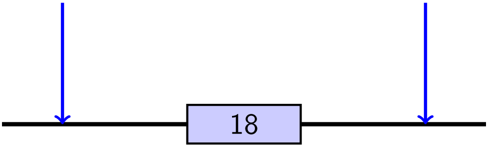
Synchronization

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



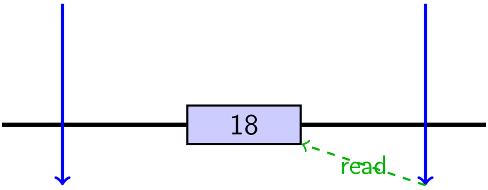
Synchronization

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



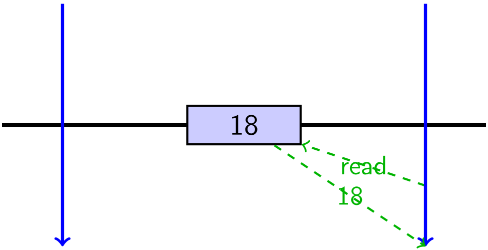
Synchronization

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



Synchronization

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

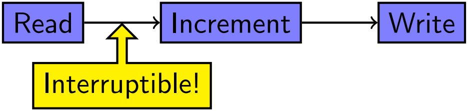


Atomic Operations

Collaborative (inter-block) Global Memory Update:

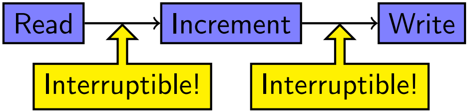
Atomic Operations

Collaborative (inter-block) Global Memory Update:



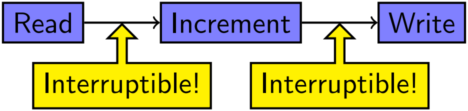
Atomic Operations

Collaborative (inter-block) Global Memory Update:

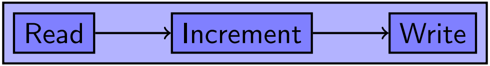


Atomic Operations

Collaborative (inter-block) Global Memory Update:

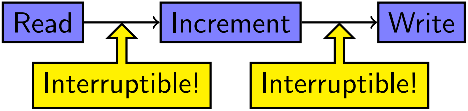


Atomic Global Memory Update:

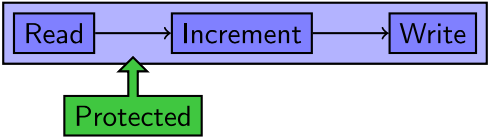


Atomic Operations

Collaborative (inter-block) Global Memory Update:

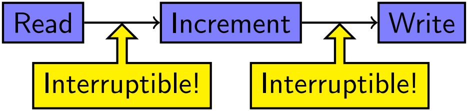


Atomic Global Memory Update:

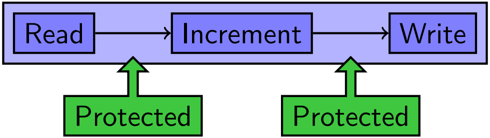


Atomic Operations

Collaborative (inter-block) Global Memory Update:

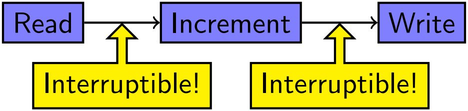


Atomic Global Memory Update:

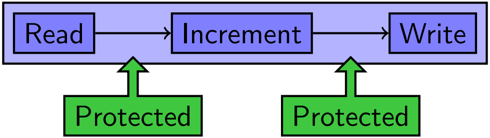


Atomic Operations

Collaborative (inter-block) Global Memory Update:



Atomic Global Memory Update:



How?

atomic_{add,inc,cmpxchg,...}(int *global, int value);

Outline

1 Intro: Python, Numpy, GPUs, OpenCL

2 GPU Programming with PyOpenCL

3 OpenCL viewed from Python
- Device Language
- The OpenCL runtime
- Synchronization
- Extensions

4 OpenCL implementations

Extensions: Future-proof CL

Similar extensions mechanism to OpenGL.

Two mechanisms:

- Runtime:
  - `cl_ext.h` header
  - availability checkable via `#ifdef`
  - `device.extensions`
- Device language:
  `#pragma OPENCL EXTENSION name : enable`

Important extension:

- cl_khr_fp64

Vendor and 'official' extensions.

Extension Example: `cl_ext_migrate_memobject`

- CL Memory Objects (Buffers, Images) tied to *context*, not *device*
- CL Standard: Implicit migration of data to location of use
- Compliant implementations are allowed to store all data on host, transfer out just for kernel
- With migration extension:
  - Migration becomes schedulable, takes part in command queue
  - More control over data locality
- ✔ Supported by PyOpenCL

Extension Example: `cl_ext_device_fission`

- Can partition a compute device
  - Equally
  - By name, counts
  - By affinity domain (L*n* cache, NUMA
- Help avoid starvation of processes that need a certain minimum throughput.
- Makes two-kernel producer-consumer model feasible.
  - Otherwise: No guarantee of progress!
- Available on Intel, AMD (CPU+GPU!)
- ✔ Supported by PyOpenCL

Outline

1. Intro: Python, Numpy, GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. OpenCL viewed from Python
4. OpenCL implementations

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)
- No linear memory texturing
- CUDA device emulation mode deprecated $\rightarrow$ Use AMD CPU CL (faster, too!)

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)
- VLIW4 (previously VLIW5)
  - *very* flop-heavy machine
  - $\rightarrow$ ILP and explicit SIMD
  - Non-vector memory coalescing only on Cayman+
- GCN: Vector *and* scalar unit
  - Move towards Nv-like programming model

CPU notes:

- Many work items per processor (emulated)
- `cl_amd_printf`
- "APU": CPU/GPU integration not very tight yet

The Intel CL implementation

CPUs now, GPUs with Ivy Bridge+

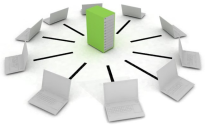
CPU notes:

- 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

The MOSIX Virtual CL implementation



- Aggregates all CL devices on a cluster into a single platform
- Looks like a “regular” CL implementation to the user
- Obvious scaling limits, but useful if the application is right
- Just heard from author: PyOpenCL supported as of version 1.10
- Aggregates communication to avoid network round-trips

Questions?

?

Image Credits

- Isaiah die shot: VIA Technologies
- Dictionary: sxc.hu/topfer
- C870 GPU: Nvidia Corp.
- Old Books: flickr.com/ppdigital (cc)
- OpenCL Logo: Apple Corp./Ars Technica
- OS Platforms: flickr.com/aOliN.Tk
- Floppy disk: flickr.com/ethanhein (cc)
- Adding Machine: flickr.com/thomashawk (cc)
- Dominoes: sxc.hu/rolve
- Context: sxc.hu/svilen001
- Queue: sxc.hu/cobrasoft
- Check mark: sxc.hu/bredmaker
- RAM stick: sxc.hu/gobran11
- CPU: sxc.hu/dimshik
- Onions: flickr.com/darwinbell (cc)
- Bricks: sxc.hu/guitargoa
- Yellow-Billed Kite: sxc.hu/doc_
- Pie Chart: sxc.hu/miamiamia
- Nvidia logo: Nvidia Corporation
- Apple logo: Apple Corporation
- AMD logo: AMD Corporation
- Intel logo: Intel Corporation
- Cluster: sxc.hu/svilen001