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

Andreas Klöckner

Courant Institute of Mathematical Sciences
New York University

PASI: The Challenge of Massive Parallelism
Lecture 1 · January 3, 2011

Course Outline

Session 1: Intro

- GPU arch. motivation
- Intro to OpenCL
- Intro to PyOpenCL
- First Steps

Session 2: Dive into CL

- CL runtime
- CL device programming language
- Notes on CL implementations

Session 3: Code Generation

- Example uses
- Methods of RTCG
- Tuning objectives
- Case study

Session 4: Advanced Topics

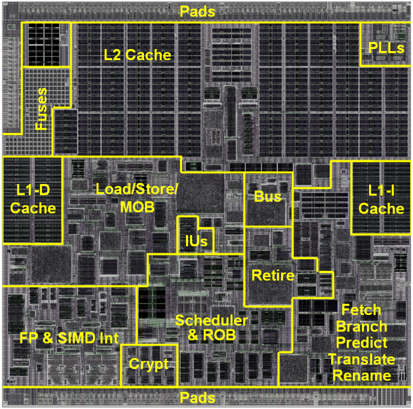
- Multi-GPU: CL+MPI, Virtual CL
- PyCUDA
- Discontinuous Galerkin Methods on GPUs

Outline

Outline

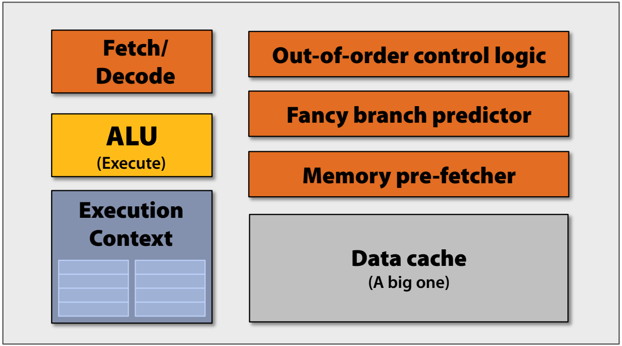
Outline

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)

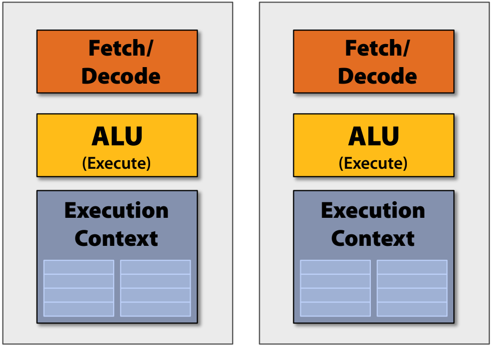
Slimming down

Idea #1:

Remove components that help a single instruction stream run fast

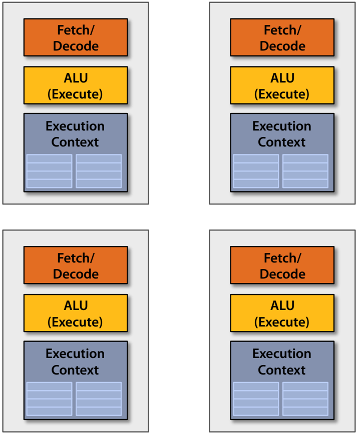
Credit: Kayvon Fatahalian (Stanford)

More Space: Double the Number of Cores



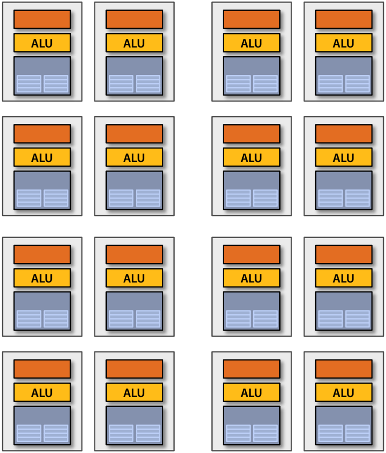
Credit: Kayvon Fatahalian (Stanford)

. . . again



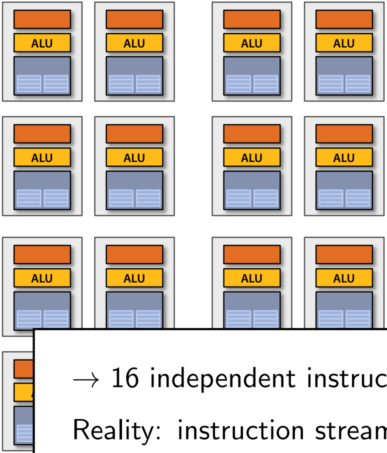
Credit: Kayvon Fatahalian (Stanford)

. . . and again



Credit: Kayvon Fatahalian (Stanford)

. . . and again

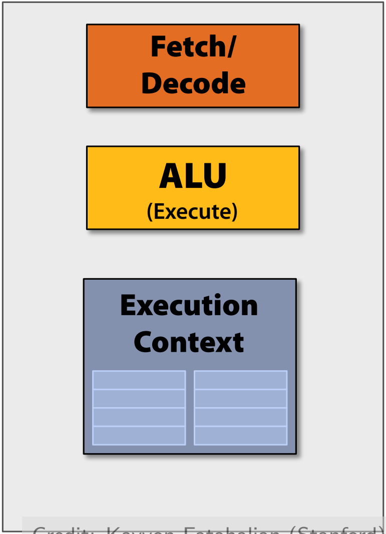


→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

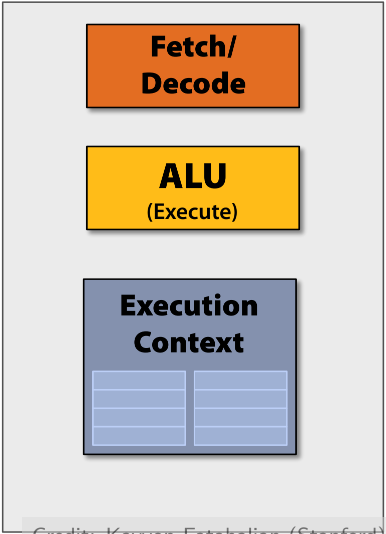
Credit: Kayvon Fatahalian (

Saving Yet More Space



Credit: Kayvon Fatahalian (Stanford)

Saving Yet More Space



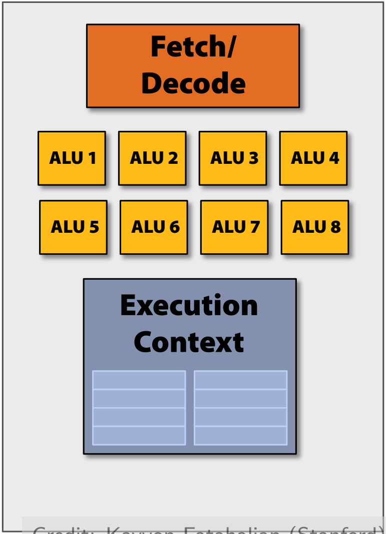
Credit: Kayvon Fatahalian (Stanford)

Idea #2

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

→ **SIMD**

Saving Yet More Space



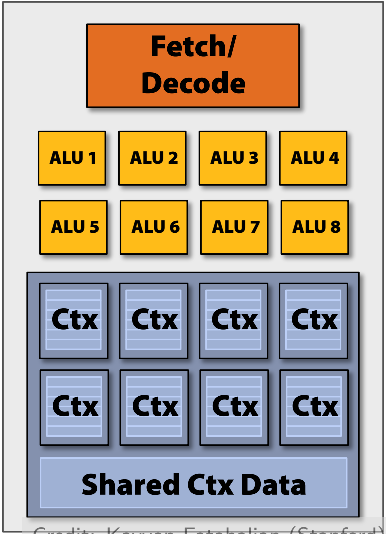
Credit: Kayvon Fatahalian (Stanford)

Idea #2

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

→ **SIMD**

Saving Yet More Space



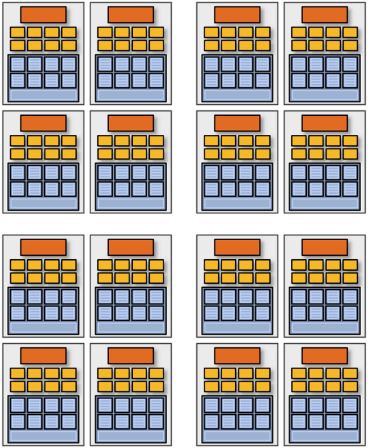
Credit: Kayvon Fatahalian (Stanford)

Idea #2

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

$\rightarrow$ **SIMD**

Gratuitous Amounts of Parallelism!

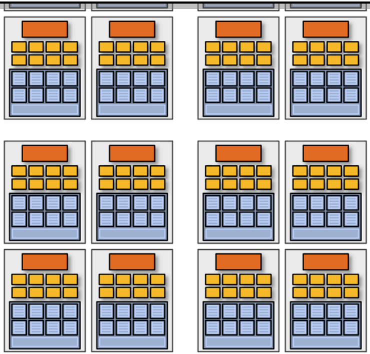


Credit: Kayvon Fatahalian (Stanford)

Gratuitous Amounts of Parallelism!

Example:

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

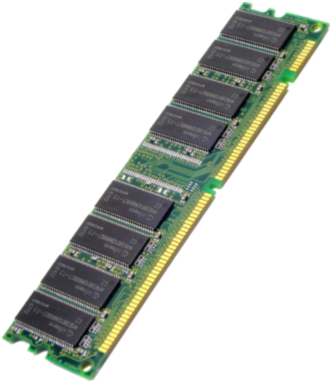


Credit: Kayvon Fatahalian (Stanford)

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

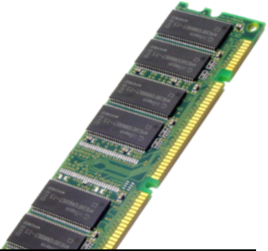
Problem

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

We've removed

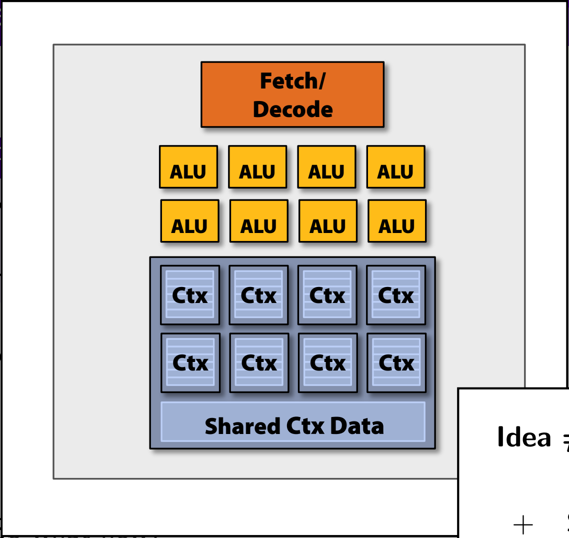
- caches
- branch prediction
- out-of-order execution

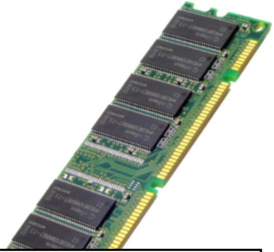
So what now?



Idea #3

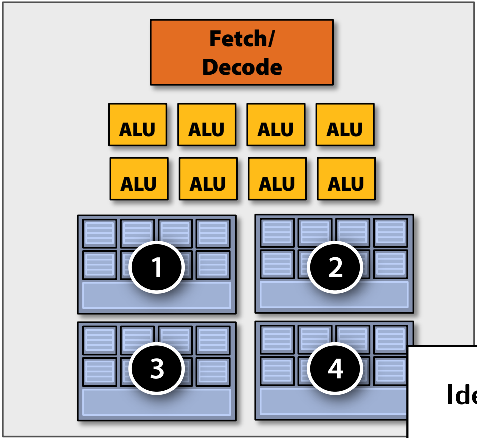
	Even more parallelism
+	Some extra memory
=	A solution!

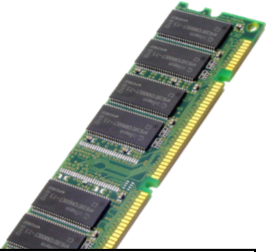




Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!





Idea #3

	Even more parallelism
+	Some extra memory
=	A solution!

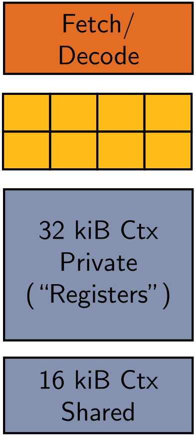
GPU Architecture Summary

Core Ideas:

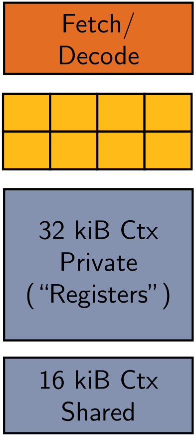
1. Many slimmed down cores
   $\rightarrow$ lots of parallelism
2. More ALUs, Fewer Control Units
3. Avoid memory stalls by interleaving execution of SIMD groups ("warps")

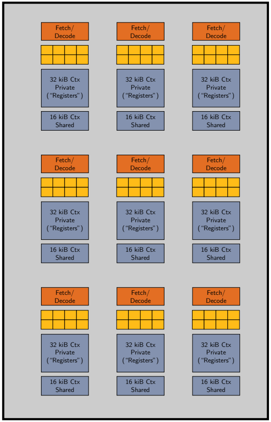
Credit: Kayvon Fatahalian (Stanford)

Connection: Hardware ↔ Programming Model

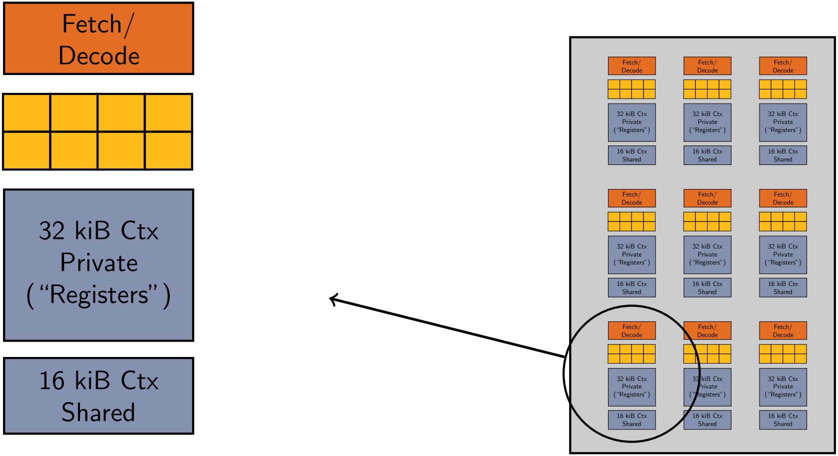


Connection: Hardware ↔ Programming Model

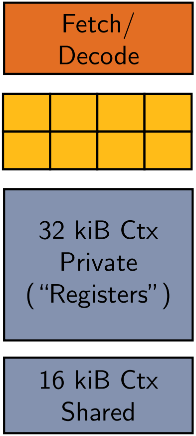


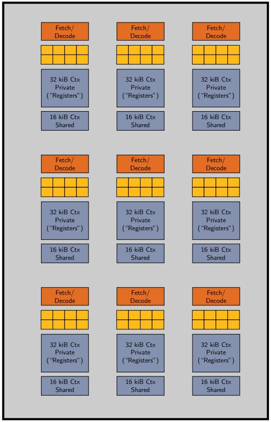


Connection: Hardware ↔ Programming Model

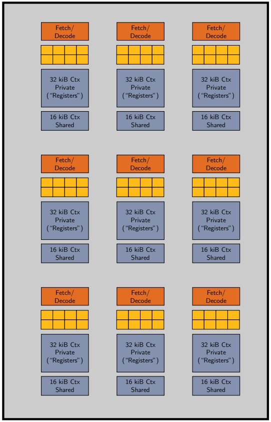


Connection: Hardware ↔ Programming Model





Connection: Hardware ↔ Programming Model

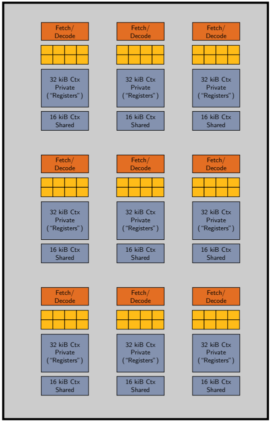


Connection: Hardware ↔ Programming Model

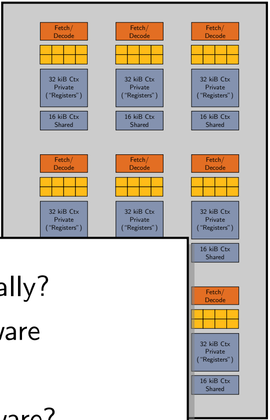
Who cares how many cores?

Idea:

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



Connection: Hardware ↔ Programming Model



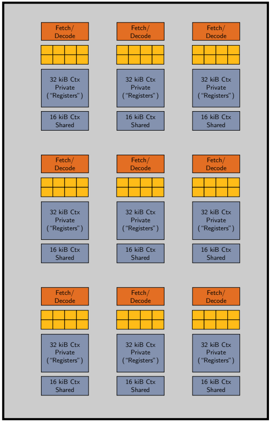
Consider: Which is easy to do automatically?

- Parallel program → sequential hardware

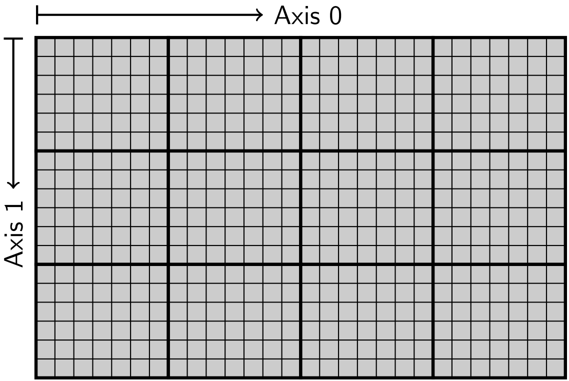
or

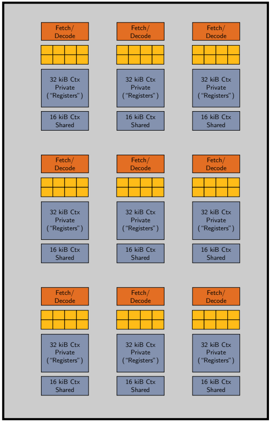
- Sequential program → parallel hardware?

Connection: Hardware ↔ Programming Model

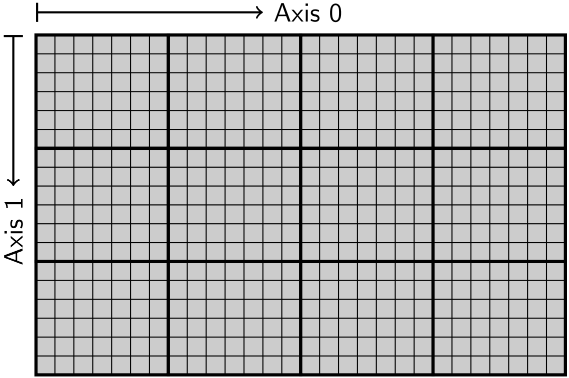


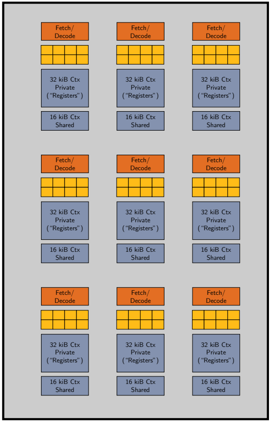
Connection: Hardware ↔ Programming Model





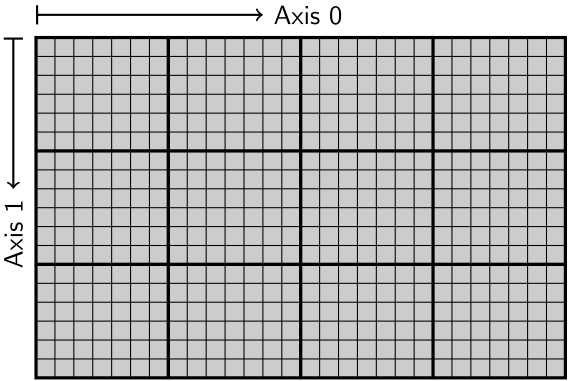
Connection: Hardware ↔ Programming Model



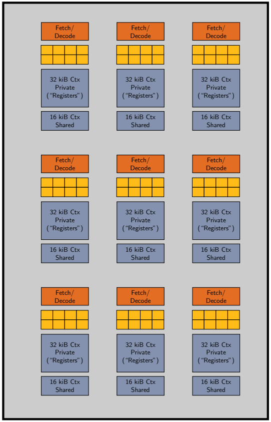


Hardware

Connection: Hardware ↔ Programming Model

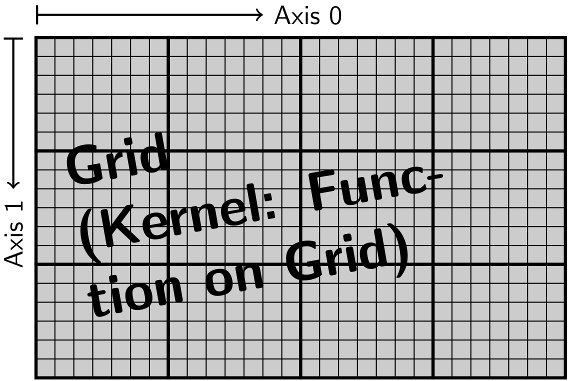


Software representation

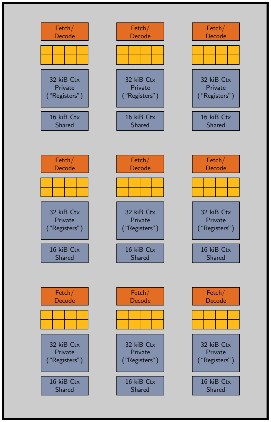


Hardware

Connection: Hardware ↔ Programming Model

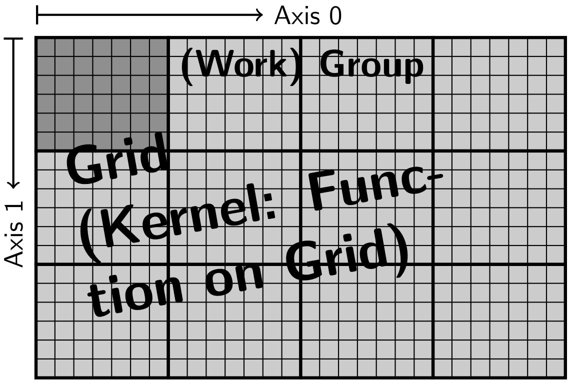


Software representation

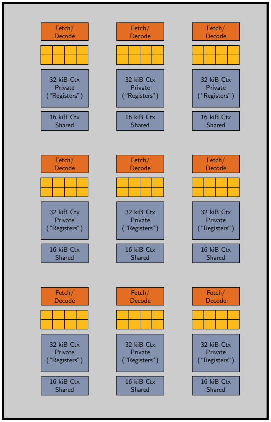


Hardware

Connection: Hardware ↔ Programming Model

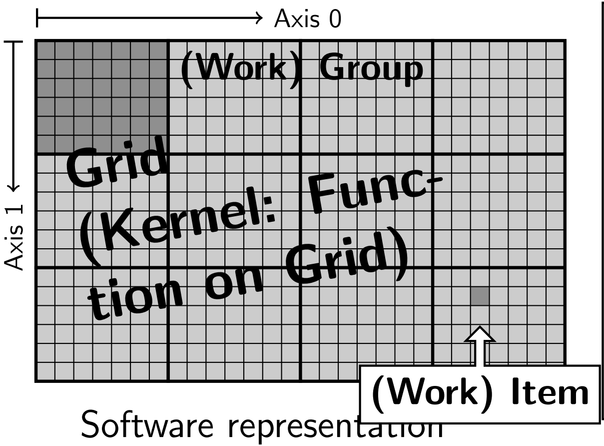


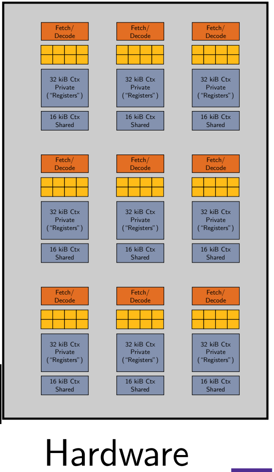
Software representation



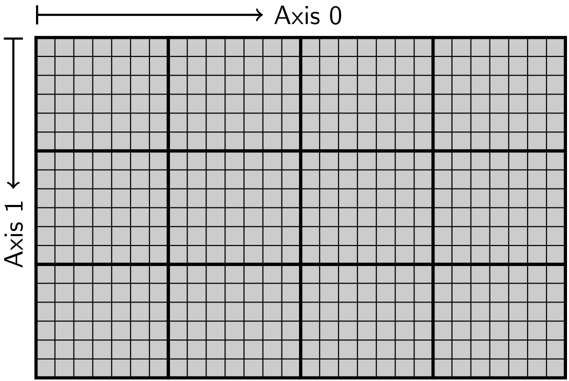
Hardware

Connection: Hardware ↔ Programming Model

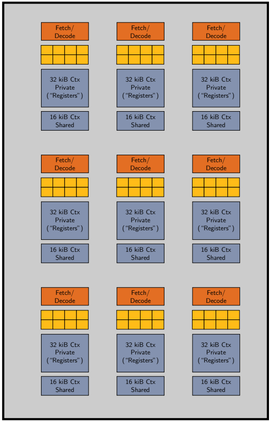




Connection: Hardware ↔ Programming Model

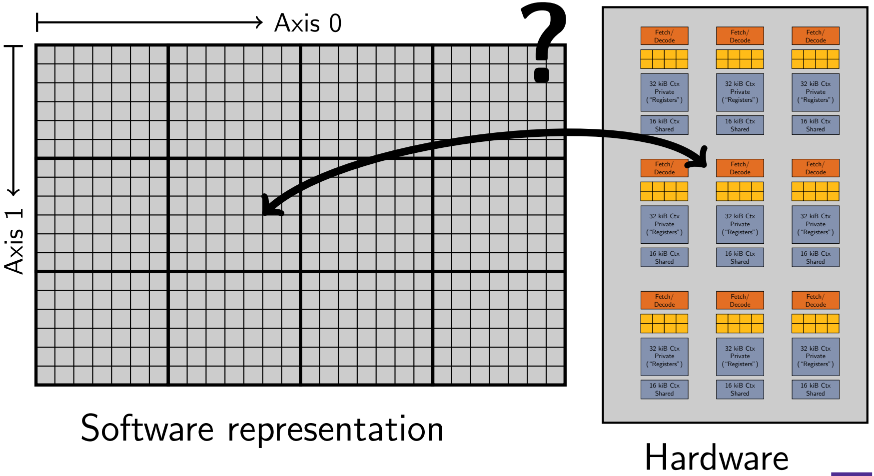


Software representation

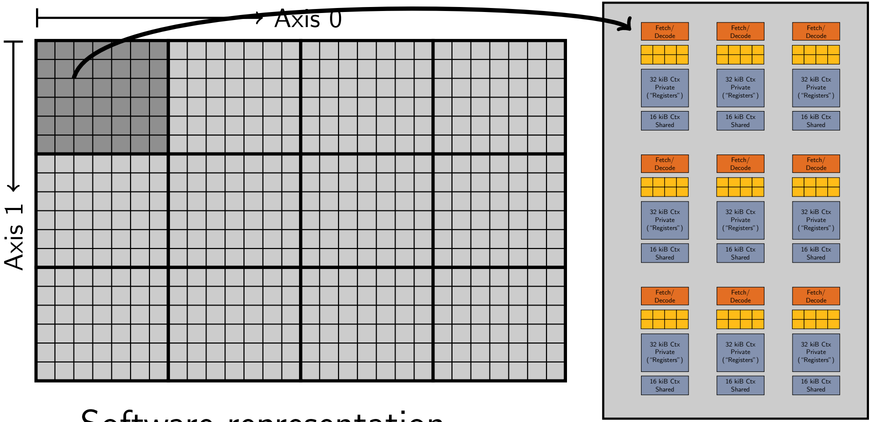


Hardware

Connection: Hardware ↔ Programming Model



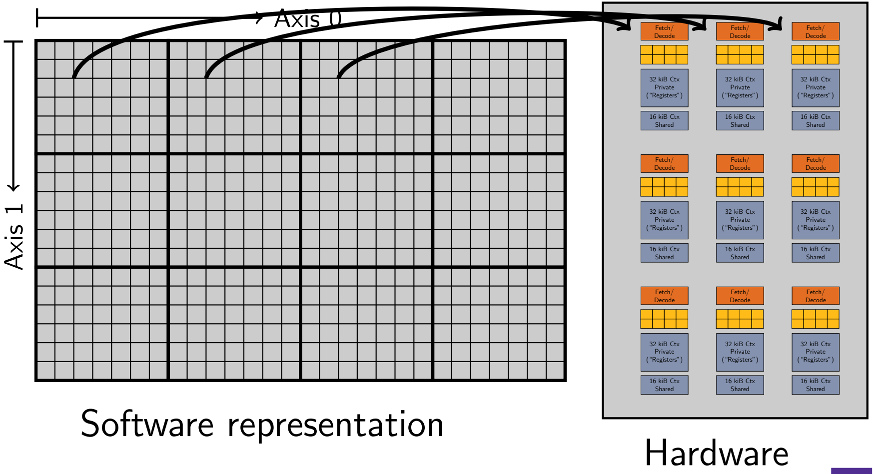
Connection: Hardware ↔ Programming Model



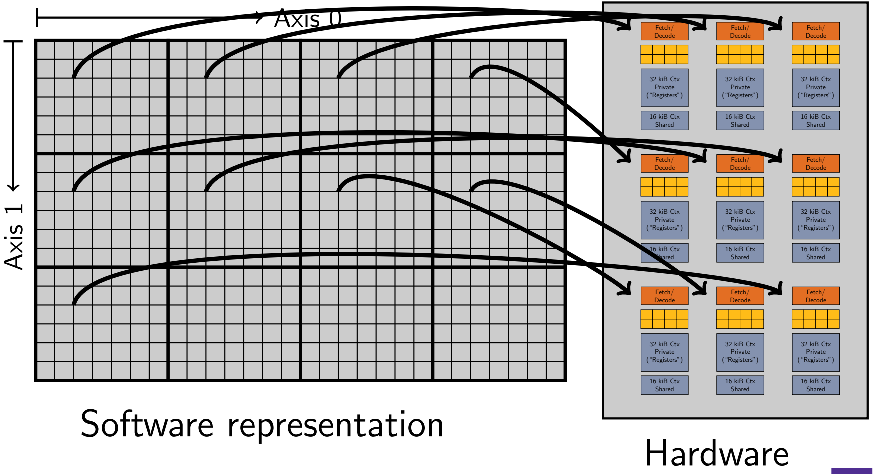
Software representation

Hardware

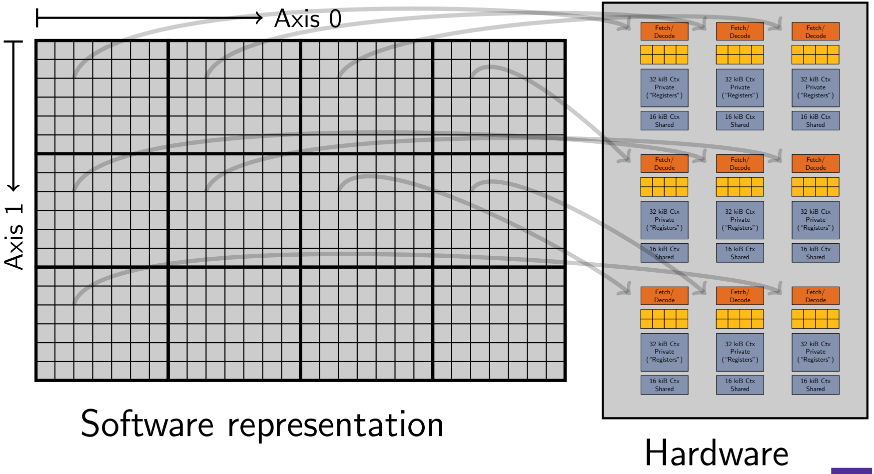
Connection: Hardware ↔ Programming Model



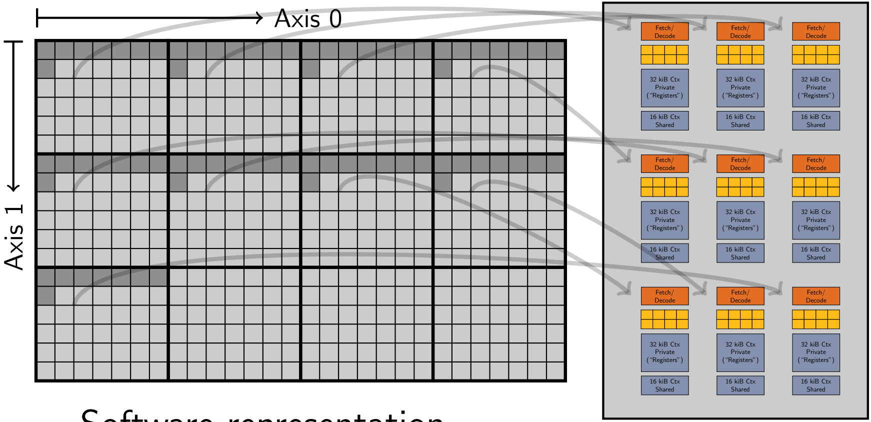
Connection: Hardware ↔ Programming Model



Connection: Hardware ↔ Programming Model



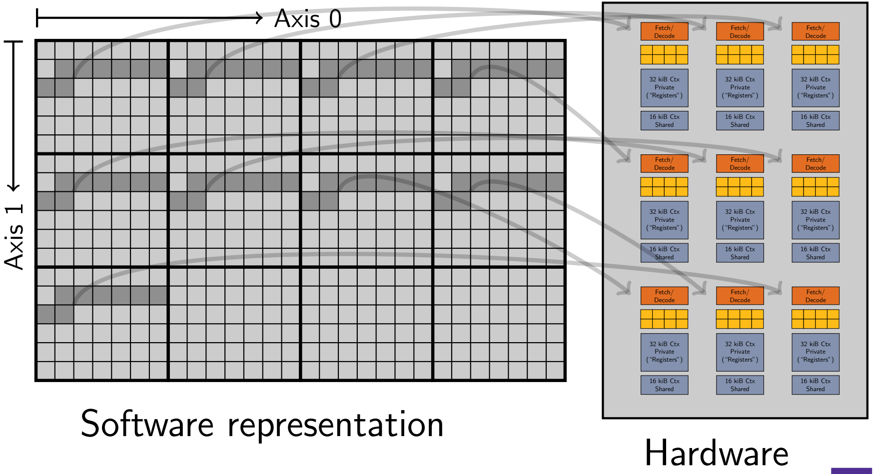
Connection: Hardware ↔ Programming Model



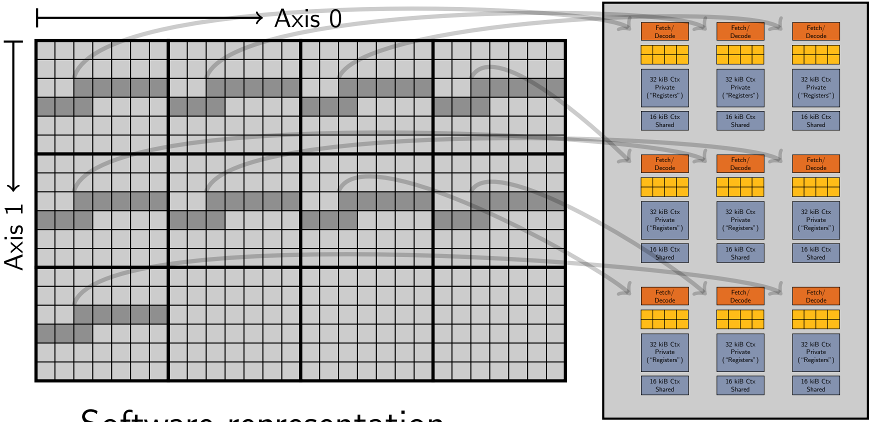
Software representation

Hardware

Connection: Hardware ↔ Programming Model



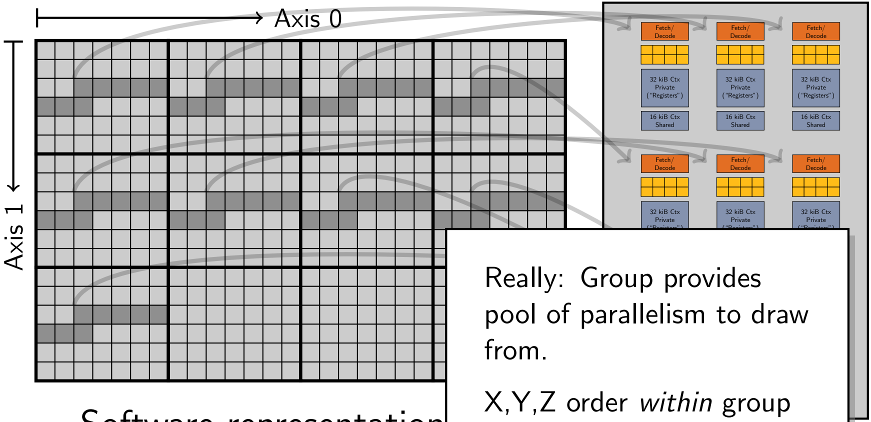
Connection: Hardware ↔ Programming Model



Software representation

Hardware

Connection: Hardware ↔ Programming Model

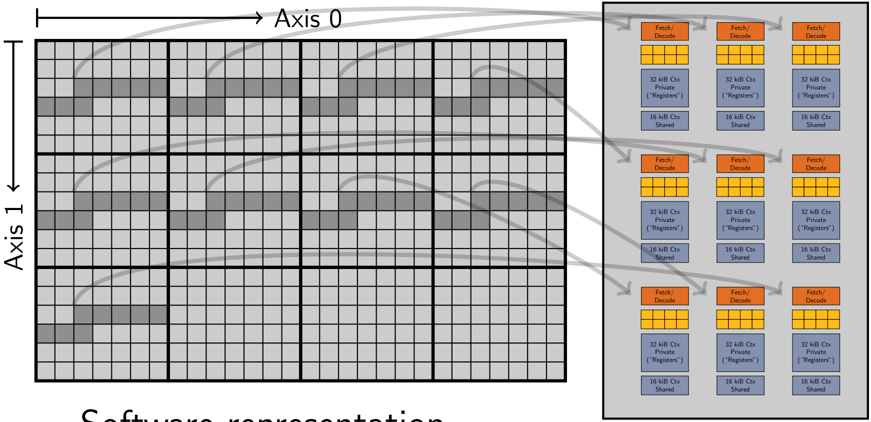


Software representation

Really: Group provides pool of parallelism to draw from.

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

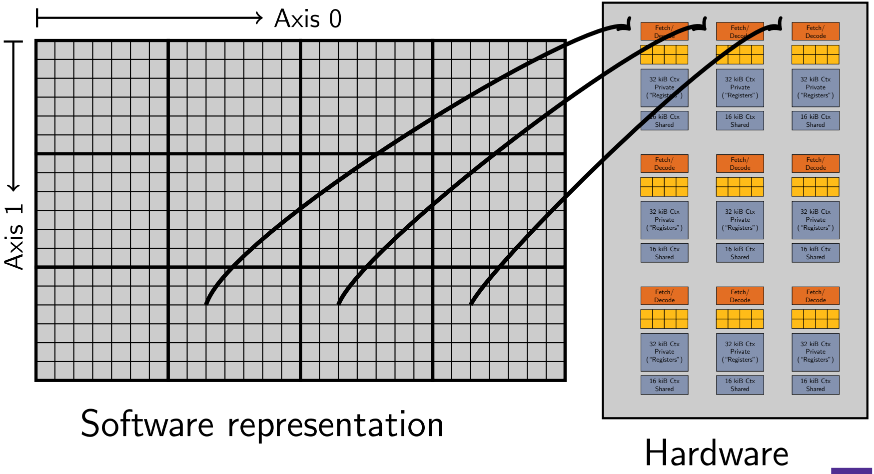
Connection: Hardware ↔ Programming Model



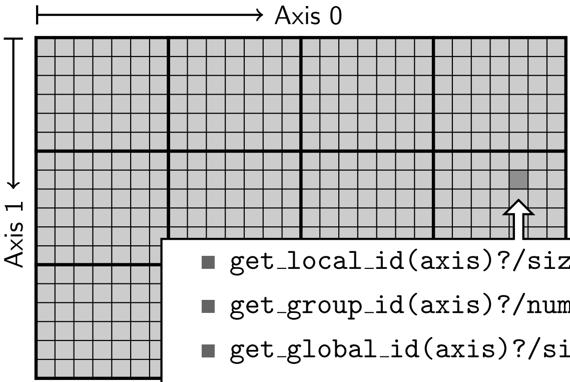
Software representation

Hardware

Connection: Hardware ↔ Programming Model



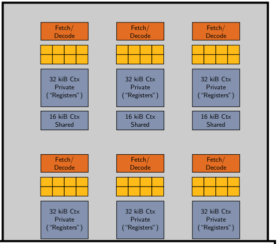
Connection: Hardware ↔ Programming Model



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

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

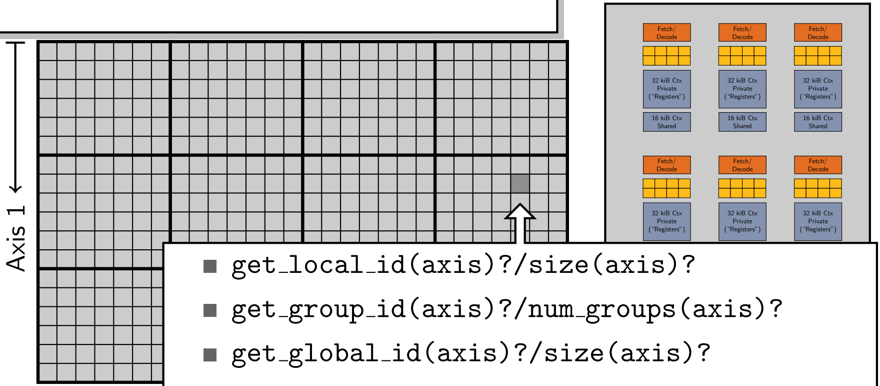
Software representation



Hardware

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,...`

Software representation

Hardware

Outline

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

Jun08 — Apple proposes OpenCL working group and contributes draft specification to Khronos

Dec08 — Khronos publicly releases OpenCL 1.0 as royalty-free specification

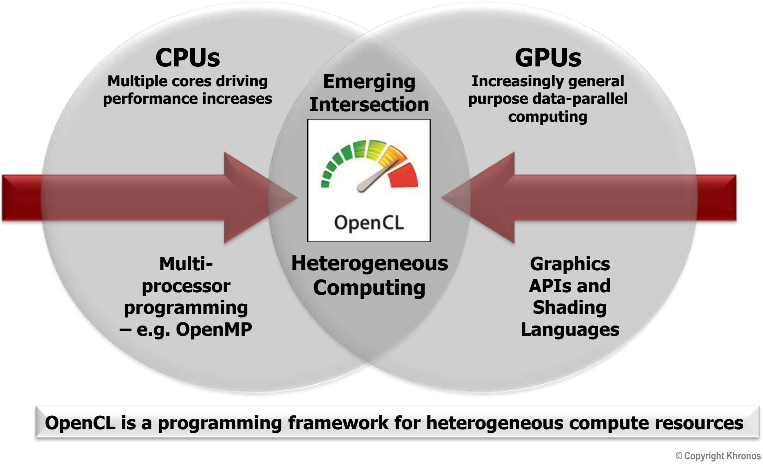
May09 — Khronos releases OpenCL 1.0 conformance tests to ensure high-quality implementations

2H09 — Multiple conformant implementations ship across diverse OS and platforms

Jun10 — OpenCL 1.1 Specification released and first implementations ship

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];
}
```

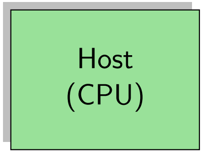
OpenCL source code:

```
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)];
}
```

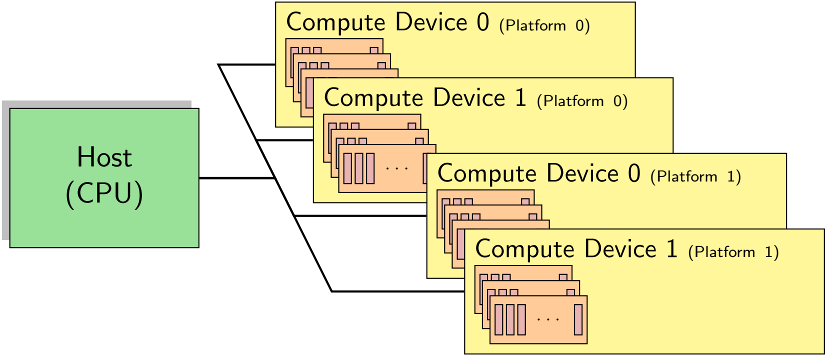
OpenCL $\leftrightarrow$ CUDA: A dictionary

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)

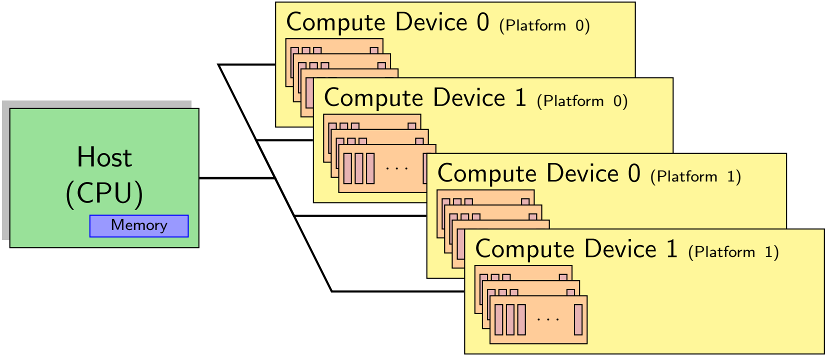
OpenCL: Computing as a Service



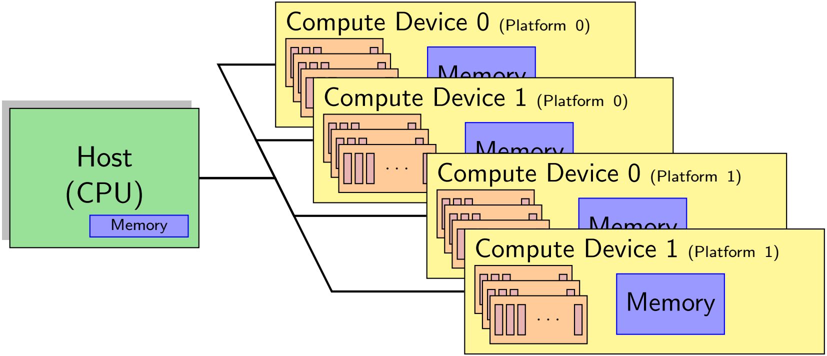
OpenCL: Computing as a Service



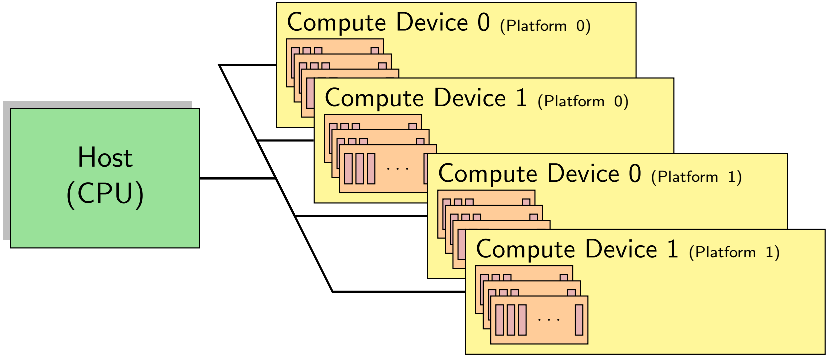
OpenCL: Computing as a Service



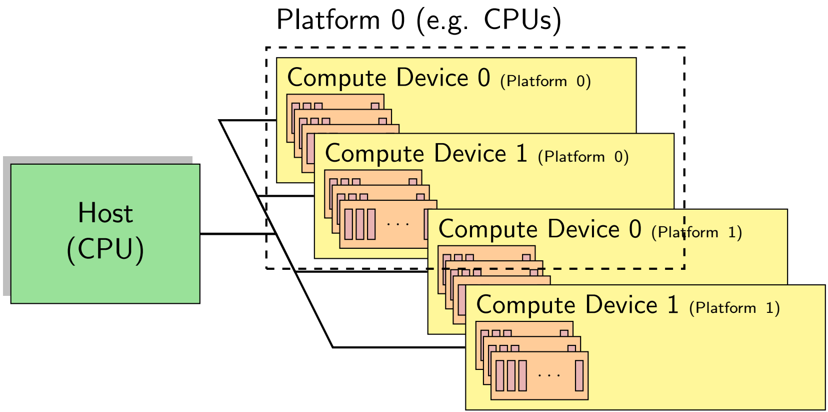
OpenCL: Computing as a Service



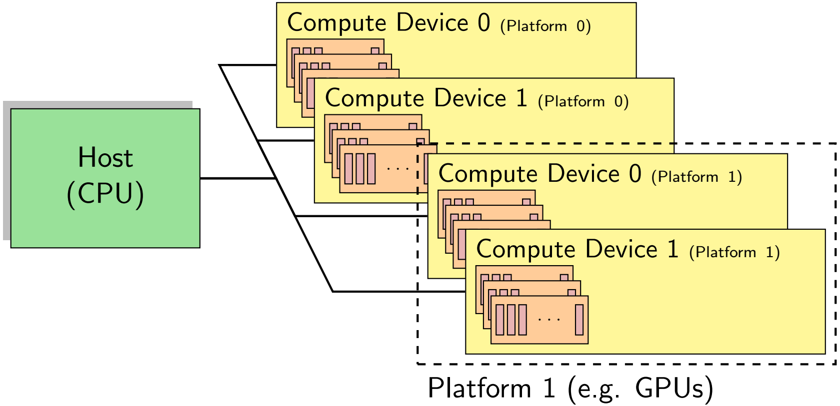
OpenCL: Computing as a Service



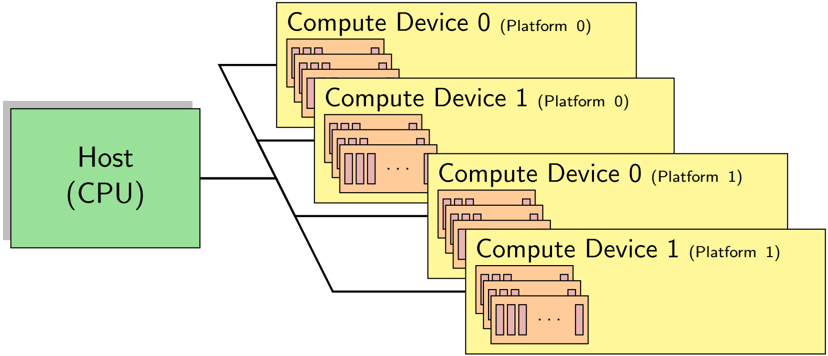
OpenCL: Computing as a Service



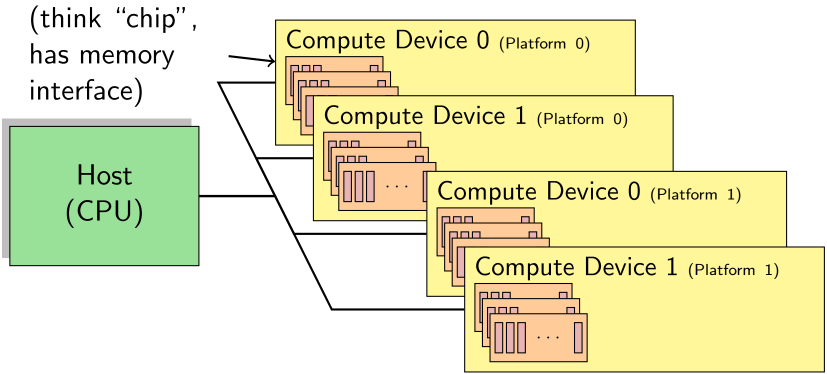
OpenCL: Computing as a Service



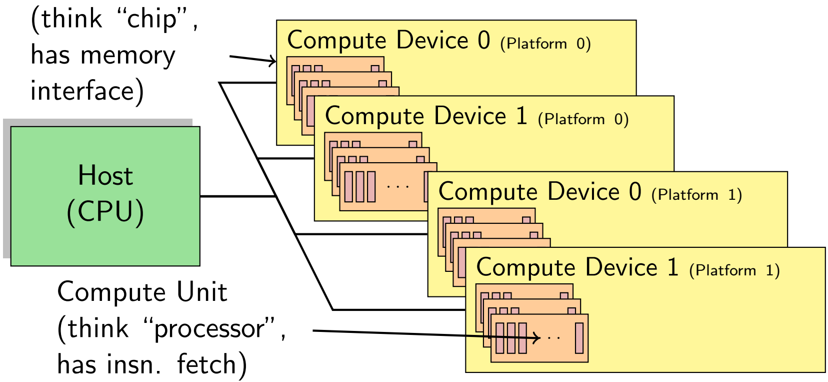
OpenCL: Computing as a Service



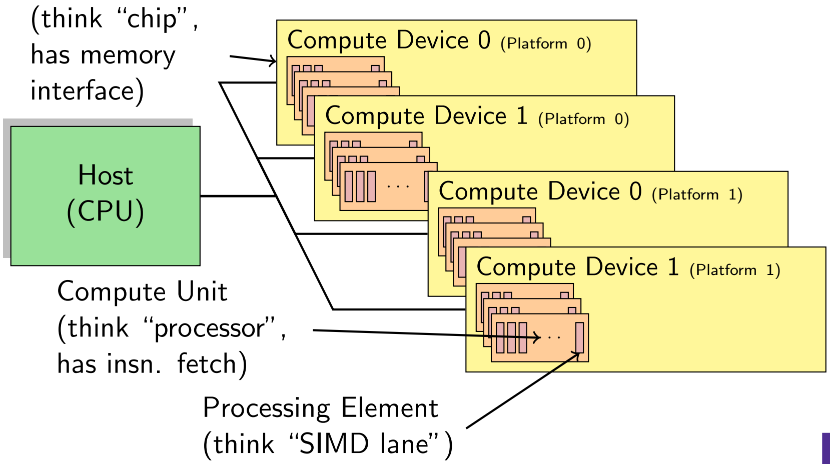
OpenCL: Computing as a Service



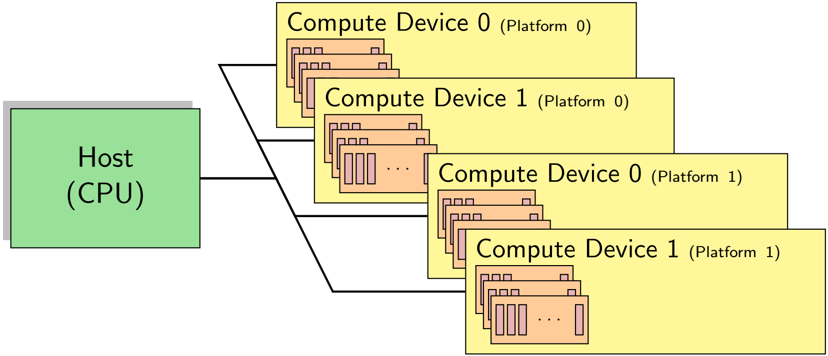
OpenCL: Computing as a Service



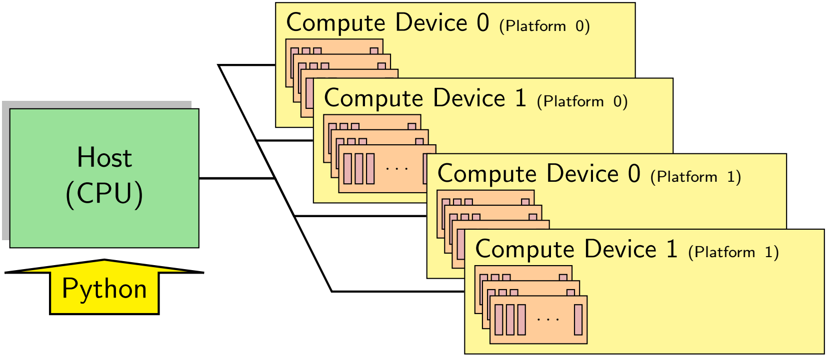
OpenCL: Computing as a Service



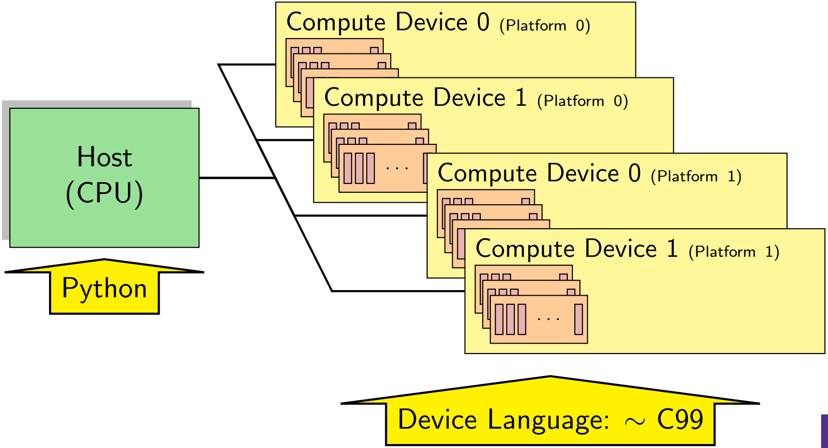
OpenCL: Computing as a Service



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

Outline

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_write_buffer(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_write_buffer(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_write_buffer(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_read_buffer(queue, a_dev, result).wait()
import numpy.linalg as la
assert la.norm(result - 2*a) == 0
```

Dive into PyOpenCL: Grouping

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
cl.enqueue_write_buffer(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_read_buffer(queue, a_dev, result).wait()
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

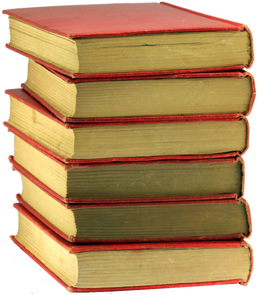
Thinking about GPU programming

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

PyOpenCL Philosophy



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

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

PyOpenCL: Vital Information

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

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
```

NYU

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)