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

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
Outline

1. Intro: GPUs, OpenCL
   - What and Why?
   - Intro to OpenCL

2. GPU Programming with PyOpenCL
Outline

1. Intro: GPUs, OpenCL
   - What and Why?
   - Intro to OpenCL

2. GPU Programming with PyOpenCL
CPU Chip Real Estate

65 nm, 4 SP ops at a time, 1 MiB L2.
“CPU-style” Cores

Fetch/Decode
ALU (Execute)
Execution Context
Out-of-order control logic
Fancy branch predictor
Memory pre-fetcher
Data cache (A big one)

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)
 Intro PyOpenCL

...and again

Credit: Kayvon Fatahalian (Stanford)

→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Saving Yet More Space

**Idea #2**

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

Credit: Kayvon Fatahalian (Stanford)
Saving Yet More Space

Idea #2

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

→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Idea #2

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

→ SIMD
Idea #2

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

→ SIMD
Gratuitous Amounts of Parallelism!

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

Credit: Kayvon Fatahalian (Stanford)
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!
Intro PyOpenCL

What and Why? OpenCL

Remaining Problem: Slow Memory

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?

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/

Hiding shader stalls

Time (clocks)

Frag 1 … 8

Idea #3
Even more parallelism
+ Some extra memory
= A solution!

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Remaining Problem: Slow Memory

... but we've removed most of the hardware that helps us deal with that.

- Caches
- Branch prediction
- Out-of-order execution

So what now?

Idea #3

Even more parallelism

+ Some extra memory

= A solution!
Core Ideas:

1. Many slimmed down cores → 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

- Fetch/Decode
- 32 kiB Ctx Private ("Registers")
- 16 kiB Ctx Shared

Who cares how many cores?

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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
Sequential program → parallel hardware?

Axis 0
Axis 1
Hardware Software representation

Really: Group provides pool of parallelism to draw from.
X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.
Connection: Hardware ↔ Programming Model

Fetch/Decode

32 kiB Ctx
Private ("Registers")

16 kiB Ctx
Shared

Who cares how many cores?

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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0

Axis 1

Hardware
Software representation

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.
Connection: Hardware ↔ Programming Model

32 kiB Ctx
Private ("Registers")

16 kiB Ctx
Shared

Who cares how many cores?

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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0
Axis 1

Hardware
Software representation

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Connection: Hardware ↔ Programming Model

Fetch/Decode

32 kiB Ctx Private ("Registers")

16 kiB Ctx Shared

Who cares how many cores?

Idea:

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

Consider: Which is easy to do automatically?

Parallel program → sequential hardware
Sequential program → parallel hardware?

Axis 0
Axis 1

Hardware
Software representation?

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0 Hardware
Axis 1 Software representation

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.
Who cares how many cores?

Idea:
- Program as if there were “infinitely” many cores
- Program as if there were “infinitely” many ALUs per core
Who cares how many cores?

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?
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
Consider: Which is easy to do automatically?
Parallel program → sequential hardware or Sequential program → parallel hardware?

Axis 0
Axis 1
Hardware
Software representation?

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Connection: Hardware ↔ Programming Model

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
<th>Fetch/Decode</th>
<th>Fetch/Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
</tr>
<tr>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
</tr>
<tr>
<td>Fetch/Decode</td>
<td>Fetch/Decode</td>
<td>Fetch/Decode</td>
</tr>
<tr>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
</tr>
<tr>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
</tr>
<tr>
<td>Fetch/Decode</td>
<td>Fetch/Decode</td>
<td>Fetch/Decode</td>
</tr>
<tr>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
<td>32 kB Ctx Private (&quot;Registers&quot;)</td>
</tr>
<tr>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
<td>16 kB Ctx Shared</td>
</tr>
</tbody>
</table>

Who cares how many cores?

Idea: Program as if there were "infinitely" many cores
Program as if there were "infinitely" many ALUs per core
Consider: Which is easy to do automatically?
Parallel program → sequential hardware or Sequential program → parallel hardware?

Axis 0

Axis 1

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?

Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Hardware

Grids can be 1, 2, 3-dimensional.
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Who cares how many cores?
Idea: Program as if there were "infinitely" many cores
Program as if there were "infinitely" many ALUs per core
Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0
Axis 1

32 kB Ctx Private ("Registers")
16 kB Ctx Shared

Fetch/Decode

32 kB Ctx Private ("Registers")
16 kB Ctx Shared

Fetch/Decode

32 kB Ctx Private ("Registers")
16 kB Ctx Shared

Fetch/Decode
Connection: Hardware ↔ Programming Model

Grid

(Kernel: Function on Grid)

Software representation

Hardware
Connection: Hardware ↔ Programming Model

Grid
(Work) Group
(Kernel: Function on Grid)

Software representation

Hardware

Who cares how many cores?
Idea:
Program as if there were "infinitely" many cores
Program as if there were "infinitely" many ALUs per core
Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Intro PyOpenCL

What and Why? OpenCL

Connection: Hardware ↔ Programming Model

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?

Parallel program → sequential hardware
- Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware
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

Consider: Which is easy to do automatically?

Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Software representation

Axis 0

Axis 1

Hardware

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware or Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation Hardware

Andreas Klöckner GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Software representation

Hardware
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware
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
Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0
Axis 1

Software representation
Hardware

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
Sequential program → parallel hardware?

Axis 1

Axis 0

Software representation

Hardware

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
Connection: Hardware ↔ Programming Model

Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)
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
Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Software representation

Hardware

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
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

Consider: Which is easy to do automatically?
Parallel program → sequential hardware
or
Sequential program → parallel hardware?

Axis 0
Axis 1

Software representation
Hardware
Introduction to PyOpenCL

What and Why? OpenCL

Connection: Hardware ↔ Programming Model

Fetch/Decode
32 kiB Context (Private)
16 kiB Context (Shared)

Who cares how many cores?
Idea: Program as if there were "infinitely" many cores
Program as if there were "infinitely" many ALUs per core
Consider: Which is easy to do automatically?
Parallel program → sequential hardware or Sequential program → parallel hardware?

Axis 0
Axis 1

Hardware
Software representation

- `get_local_id(axis)`?/`size(axis)`?
- `get_group_id(axis)`?/`num_groups(axis)`?
- `get_global_id(axis)`?/`size(axis)`?
- `axis=0,1,2,...`

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
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

Axis 1

Software representation

Hardware

Who cares how many cores?

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

Consider: Which is easy to do automatically?

Parallel program → sequential hardware
- Sequential program → parallel hardware?

Axis=0,1,2,...

Really: Group provides pool of parallelism to draw from.

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

get local id(axis)?/size(axis)?
get group id(axis)?/num groups(axis)?
get global id(axis)?/size(axis)?
Outline

1 Intro: GPUs, OpenCL
   - What and Why?
   - Intro to OpenCL

2 GPU Programming with PyOpenCL
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
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?

OpenCL is a programming framework for heterogeneous compute resources.

Credit: Khronos Group
**CUDA source code:**

```c
__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:**

```c
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

<table>
<thead>
<tr>
<th>OpenCL</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>Grid</td>
</tr>
<tr>
<td>Work Group</td>
<td>Block</td>
</tr>
<tr>
<td>Work Item</td>
<td>Thread</td>
</tr>
<tr>
<td>__kernel</td>
<td><strong>global</strong></td>
</tr>
<tr>
<td>__global</td>
<td><strong>device</strong></td>
</tr>
<tr>
<td>__local</td>
<td><strong>shared</strong></td>
</tr>
<tr>
<td>__private</td>
<td><strong>local</strong></td>
</tr>
<tr>
<td>imagednd_t</td>
<td>texture&lt;type, n, ...&gt;</td>
</tr>
<tr>
<td>barrier(LMF)</td>
<td>__syncthreads()</td>
</tr>
<tr>
<td>get_local_id(012)</td>
<td>threadIdx.xyz</td>
</tr>
<tr>
<td>get_group_id(012)</td>
<td>blockIdx.xyz</td>
</tr>
<tr>
<td>get_global_id(012)</td>
<td>(reimplement)</td>
</tr>
</tbody>
</table>
OpenCL: Computing as a Service
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Python Device Language: \sim C99

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Computing as a Service

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Platform 0 (e.g. CPUs)

- Compute Device 0 (Platform 0)
- Compute Device 1 (Platform 0)

Platform 1 (e.g. GPUs)

- Compute Device 0 (Platform 1)
- Compute Device 1 (Platform 1)

Host (CPU)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Platform 1 (e.g. GPUs)
OpenCL: Computing as a Service

- Host (CPU)
- Compute Device 0 (Platform 0)
  - Compute Device 1 (Platform 0)
  - Compute Device 0 (Platform 1)
  - Compute Device 1 (Platform 1)

OpenCL: Computing as a Service
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)
Compute Device 1 (Platform 0)

···

Compute Device 0 (Platform 1)
Compute Device 1 (Platform 1)

(think “chip”, has memory interface)
OpenCL: Computing as a Service

Host (CPU)

Compute Unit (think “processor”, has insn. fetch)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Processing Element (think “SIMD lane”)

Compute Unit (think “processor”, has insn. fetch)

Host (CPU) (think “chip”, has memory interface)

Python Device Language: \( \sim \) C99

Andreas Klöckner
GPU-Python with PyOpenCL and PyCUDA
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)
Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)
Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Python

Compute Device 0 (Platform 0)

···

Compute Device 1 (Platform 0)

···

Compute Device 0 (Platform 1)

···

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Python

Compute Device 0 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 0)

Compute Device 1 (Platform 1)

Device Language: ~ C99
Why do Scripting for GPUs?

- GPUs are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  → complement each other
- CPU: largely restricted to control tasks (≈1000/sec)
  - Scripting fast enough
- Python + CUDA = **PyCUDA**
- Python + OpenCL = **PyOpenCL**
Outline

1 Intro: GPUs, OpenCL

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

1 Intro: GPUs, OpenCL

2 GPU Programming with PyOpenCL
   ■ First Contact
   ■ About PyOpenCL
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)

c.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

```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_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

```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_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)
c.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)
c.enqueue_read_buffer(queue, a_dev, result).wait()
import numpy.linalg as la
assert la.norm(result - 2*a) == 0
```
Thinking about GPU programming

How would we modify the program to...

1. Compute \( c_i = a_i b_i \)?
2. Use groups of 16 \( \times 16 \) work items?
3. Benchmark 1 work item per group against 256 work items per group? (Use `time.time()` and `.wait()`.}

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Dive into PyOpenCL: Thinking on your feet

Thinking about GPU programming

How would we modify the program to...

1. ...compute \( c_i = a_i b_i \)?
Thinking about GPU programming

How would we modify the program to...

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

Andreas Klöckner

GPU-Python with PyOpenCL and PyCUDA
Outline

1 Intro: GPUs, OpenCL

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

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

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

a = numpy.random.randn(500000).astype(numpy.float32)
b = numpy.random.randn(500000).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.shape)

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

prog.sum(queue, a.shape, (1,), dest_buf)
a_plus_b = numpy.empty_like(a)
cl.enqueue_nd_range_buffer(queue, dest_buf, a_plus_b, 1).wait()

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

(You can find this example as `example/demop.py` in the PyOpenCL source distribution.)
PyOpenCL: Vital Information

- [http://mathema.tician.de/software/pyopencl](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:

```python
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
```
Questions?
Image Credits

- Isaiah die shot: VIA Technologies
- Dictionary: sxc.hu/topfer
- C870 GPU: Nvidia Corp.
- Old Books: flickr.com/ppdigital
- OpenCL Logo: Apple Corp./Ars Technica
- OS Platforms: flickr.com/aOliN.Tk
- Floppy disk: flickr.com/ethanhein