Scripting GPUs with PyOpenCL

Andreas Klöckner

Division of Applied Mathematics
Brown University

Scipy 2010 · June 29, 2010
Thanks

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

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
4. Playtime!
5. Conclusions
Outline

1 Intro: GPUs, OpenCL
   - What and Why?
   - Bird’s eye view of OpenCL

2 GPU Programming with PyOpenCL

3 Additional Topics

4 Playtime!

5 Conclusions
Outline

1 Intro: GPUs, OpenCL
   - What and Why?
   - Bird’s eye view of OpenCL

2 GPU Programming with PyOpenCL

3 Additional Topics

4 Playtime!

5 Conclusions
GPU Computing?

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

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

- GPU Computing takes a different approach:
  - Throughput matters—single threads do not
  - Hide latency through parallelism
  - Let programmer deal with “raw” storage hierarchy
Floorplan: VIA Isaiah (2008)  
65 nm, 4 SP ops at a time, 1 MiB L2.

Floorplan: AMD RV770 (2008)  
55 nm, 800 SP ops at a time.
GPU Architecture (e.g. Nvidia GT200)

- 1 GPU = 30 SIMD cores
- 1 SIMD core: 32 × 32 PCs, HW Sched + 1 ID (1/4 clock) + 8 SP + 1 DP + 16 KiB Shared + 32 KiB Reg
- Device ↔ RAM: 140 GB/s
- Device ↔ Host: 6 GB/s
- User manages memory hierarchy
## GPU Programming: Gains and Losses

<table>
<thead>
<tr>
<th>Gains</th>
<th>Losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Bandwidth</td>
<td></td>
</tr>
<tr>
<td>(140 GB/s vs. 12 GB/s)</td>
<td></td>
</tr>
<tr>
<td>Compute Bandwidth</td>
<td></td>
</tr>
<tr>
<td>(Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)</td>
<td></td>
</tr>
<tr>
<td>Data-parallel programming</td>
<td></td>
</tr>
</tbody>
</table>
## GPU Programming: Gains and Losses

<table>
<thead>
<tr>
<th>Gains</th>
<th>Losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅ Memory Bandwidth</td>
<td>✅ Tuning hardware-specific</td>
</tr>
<tr>
<td>(140 GB/s vs. 12 GB/s)</td>
<td></td>
</tr>
<tr>
<td>✅ Compute Bandwidth</td>
<td>✅ Data size ⇄ Alg. design</td>
</tr>
<tr>
<td>(Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)</td>
<td>✅ Cheap branches (i.e. ifs)</td>
</tr>
<tr>
<td>✅ Data-parallel programming</td>
<td>✅ Fine-grained malloc *)</td>
</tr>
<tr>
<td>✅) Possibly less problematic soon.</td>
<td>✅ Recursion *)</td>
</tr>
<tr>
<td>✅ Function pointers *)</td>
<td>✅ Function pointers *)</td>
</tr>
</tbody>
</table>

*) Possibly less problematic soon.
Outline

1. Intro: GPUs, OpenCL
   - What and Why?
   - Bird’s eye view of OpenCL

2. GPU Programming with PyOpenCL

3. Additional Topics

4. Playtime!

5. Conclusions
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]

- Vendor-neutral, unlike Nvidia CUDA
  - though rather similar to it

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)
OpenCL: Computing as a Service

Host (CPU)
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)
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

Platform 0 (e.g. CPUs)

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)

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

Host (CPU)

(Think “chip”, has memory interface)

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

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

Platform 0 (e.g. CPUs)
Platform 1 (e.g. GPUs)

Python
Device Language:
~C99
OpenCL: Computing as a Service

- **Host (CPU)**
  - Compute Unit (think “processor”, has insn. fetch)

- **Compute Device 0 (Platform 0)**
  - Memory

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

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

- **Compute Device 1 (Platform 1)**
  - Memory

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

(host “chip”, has memory interface)

Host (CPU)

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

Processing Element (think “SIMD lane”)

Python Device Language: ∼C99

Andreas Klöckner

Scripting GPUs with PyOpenCL
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

Python Device Language: \sim C99

Andreas Klöckner

Scripting GPUs with PyOpenCL
OpenCL: Computing as a Service

Python

Device Language: ~ C99
OpenCL: Execution Model

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

**nD Grid**
- Group (0, 0)
- Group (1, 0)
- Group (2, 0)
- Group (0, 1)
- Group (1, 1)
- Group (2, 1)

**Work Group (1, 0)**
- Item (0, 0)
- Item (1, 0)
- Item (2, 0)
- Item (3, 0)
- Item (0, 1)
- Item (1, 1)
- Item (2, 1)
- Item (3, 1)
- Item (0, 2)
- Item (1, 2)
- Item (2, 2)
- Item (3, 2)
- Item (0, 3)
- Item (1, 3)
- Item (2, 3)
- Item (3, 3)
OpenCL: Execution Model

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

$nD$ Grid

Group (0, 0) Group (1, 0) Group (2, 0)
Group (0, 1) Group (1, 1) Group (2, 1)
Work Group (1, 0)
Item (0, 0) Item (1, 0) Item (2, 0) Item (3, 0)
Item (0, 1) Item (1, 1) Item (2, 1) Item (3, 1)
Item (0, 2) Item (1, 2) Item (2, 2) Item (3, 2)
Item (0, 3) Item (1, 3) Item (2, 3) Item (3, 3)
OpenCL: Execution Model

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

- Comm/Sync only within work group
- Work group maps to compute unit

- Grid/Group $\approx$ outer loops in an algorithm

- Device Language:
  - `get_{global,group,local}_{id,size}(axis)`
Why do Scripting for OpenCL?

- Compute Devices are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  → complement each other
Why do Scripting for OpenCL?

- Compute Devices are everything that scripting languages are not.
  - Highly parallel
  - Very architecture-sensitive
  - Built for maximum FP/memory throughput
  - complement each other
- CPU: largely restricted to control tasks (∼1000/sec)
  - Scripting fast enough
Why do Scripting for OpenCL?

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

1. Intro: GPUs, OpenCL

2. GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - Dealing with Space: Memory
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3. Additional Topics

4. Playtime!

5. Conclusions
Outline

1 Intro: GPUs, OpenCL

2 GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - Dealing with Space: Memory
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3 Additional Topics

4 Playtime!

5 Conclusions
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)
```
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

Andreas Klöckner
Scripting GPUs with PyOpenCL
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
8 a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
9 cl.enqueue_write_buffer(queue, a_dev, a)
10
dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=a.nbytes)
11 cl.enqueue_write_buffer(queue, a_dev, a)
12
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()
13
14 prg.twice(queue, a.shape, (256,), a_dev)
15
16 result = numpy.empty_like(a)
17 cl.enqueue_read_buffer(queue, a_dev, result).wait()
18 import numpy.linalg as la
19 assert la.norm(result - 2*a) == 0
```

Andreas Klöckner

Scripting GPUs with PyOpenCL
Getting your feet wet

Log into your assigned machine:

1. ssh NAME@haamster.rice.edu
2. ssh teramite or ssh slate

In your home directory, find “1-intro/intro.py”. Try running it (on the right GPU).

http://tiker.net/tmp/scipy10-pyopencl-tut.tar.gz

Thinking about GPU programming

How would we modify the program to...

Andreas Klöckner
Getting your feet wet

Log into your assigned machine:

1. `ssh NAME@haamster.rice.edu`
2. `ssh teramite` or `ssh slate`

In your home directory, find “1-intro/intro.py”. Try running it (on the right GPU).

http://tiker.net/tmp/scipy10-pyopencl-tut.tar.gz

Thinking about GPU programming

How would we modify the program to...

1. …compute \( c_i = a_i b_i \)?
Getting your feet wet

Log into your assigned machine:

1. `ssh NAME@haamster.rice.edu`
2. `ssh teramite` or `ssh slate`

In your home directory, find “1-intro/intro.py”.
Try running it (on the right GPU).

http://tiker.net/tmp/scipy10-pyopencl-tut.tar.gz

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?
Getting your feet wet

Log into your assigned machine:

1. ssh NAME@haamster.rice.edu
2. ssh teramite or ssh slate

In your home directory, find “1-intro/intro.py”. Try running it (on the right GPU).

http://tiker.net/tmp/scipy10-pyopencl-tut.tar.gz

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()`.)
Outline

1 Intro: GPUs, OpenCL

2 GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - Dealing with Space: Memory
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3 Additional Topics

4 Playtime!

5 Conclusions
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
Command Queues and Events

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

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

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

✔ OK

```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_write_buffer(queue, a_dev, a)
```

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

# OK

a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev, numpy.random.rand(256**3).astype(numpy.float32), blocking=True)
```
Command Queues: A Crashy Puzzle

✅ OK

```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_write_buffer(queue, a_dev, a)
```

❌ Crash

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev, 
                        numpy.random.rand(256**3).astype(numpy.float32))
```
Command Queues: A Crashy Puzzle

✔ OK

```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_write_buffer(queue, a_dev, a)
```

✖ Crash

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

✔ OK

```python
a_dev = cl.Buffer(ctx, cl.mem_flags.READ_WRITE, size=256**3*4)
cl.enqueue_write_buffer(queue, a_dev, 
numpy.random.rand(256**3).astype(numpy.float32), is_blocking=True)
```
Command Queues: 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_write_buffer(queue, a_dev, a)
```

❌ Crash

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

✅ OK

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

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

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

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

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

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

- **src**: OpenCL device code
  - Derivative of C99
  - Functions with `_kernel` attribute can be invoked from host

```python
prg.build(options="", devices=None)
```

- **kernel**: `prg.kernel_name`

```python
kernel(queue, 
       (G_x, G_y, G_z), (S_x, S_y, S_z), 
       arg, ..., 
       wait_for=None)
```

(Note: `local_size` used to be keyword argument.)
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

Explicitly sized scalars:

✖️ Annoying, error-prone.

Better:

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

Use None for non-scalars.
OpenCL exposes two different forms of SIMD computing:

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

**Single-Instruction Multiple-Data in OpenCL**

OpenCL exposes two different forms of SIMD computing:

- **Explicit**: Use (e.g.) `float2`, `float4`, `float16`.
- **Implicit**: Adjacent work items get mapped to SIMD lanes (implemented in hardware or software).

** Implicit SIMD**: Groups of work items are scheduled together.  
→ “Work Item” \(\neq\) “Thread”!
Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

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

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

```c
if (get_global_id(0) % 2 == 0)
    do_something();
else
    do_another_thing();
do_the_rest();
```
Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

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

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

```python
if (get_global_id(0) % 2 == 0):
    do_something();
else:
    do_another_thing();
    do_the_rest();
```
Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

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

**Implicit SIMD**: Groups of work items are scheduled together.

→ “Work Item” ≠ “Thread”!

```c
if (get_global_id(0) % 2 == 0)
    do_something();
else
    do_another_thing();
    do_the_rest();
```
Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

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

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

```cpp
if (get_global_id(0) % 2 == 0)
    do_something();
else
    do_another_thing();
do_the_rest();
```
Implicit and Explicit SIMD

Single-Instruction Multiple-Data in OpenCL

OpenCL exposes two different forms of SIMD computing:

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

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

```cpp
if (get_global_id(0) % 2 == 0)
    do_something();
else
    do_anotherThing();
do_the_rest();
```
Outline

1. Intro: GPUs, OpenCL

2. GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - **Dealing with Space: Memory**
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3. Additional Topics

4. Playtime!

5. Conclusions
Example: Matrix Transpose

| C1,1 | C1,2 | C1,3 | C1,4 |
| C2,1 | C2,2 | C2,3 | C2,4 |
| C3,1 | C3,2 | C3,3 | C3,4 |
| C4,1 | C4,2 | C4,3 | C4,4 |

| C1,1 | C2,1 | C3,1 | C4,1 |
| C1,2 | C2,2 | C3,2 | C4,2 |
| C1,3 | C2,3 | C3,3 | C4,3 |
| C1,4 | C2,4 | C3,4 | C4,4 |

Andreas Klöckner
Scripting GPUs with PyOpenCL
Transpose? Simple Enough!

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

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

w, h = shape
return self.kernel(queue, (w, h), (1,1),
                   tgt, src, numpy.uint32(w), numpy.uint32(h))
```
Writing high-performance Codes

Mindset: What is going to be the limiting factor?

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

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

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

Benchmark the assumed limiting factor right away.
Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?
- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Evaluate

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

Very likely: Bound by memory bandwidth.
Performance: Matrix transpose

Very likely: Bound by memory bandwidth.

**Fantastic!** Far slower than CPU. Why?
Intra-device Work Distribution

```python
w, h = shape
return self.kernel(queue, (w, h), (1,1),
tgt, src, numpy.uint32(w), numpy.uint32(h))
```

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

Again: Work Groups

- Work group size matters. A lot.
- Determines work distribution among processors
- Optimal size? Up to experimentation
Performance: Matrix transpose

Better. $1.5 \times$ faster than CPU—not great. Why?
Aside: How does computer memory work?

One memory transaction (simplified):

Processor → Memory

- D0..15
- A0..15
- R/\bar{W}
- CLK

Observation: Access (and addressing) happens in bus-width-size "chunks".
Aside: How does computer memory work?

One memory transaction (simplified):

- Processor
- Memory
- D0..15
- A0..15
- R/\text{W}
- CLK

Observation: Access (and addressing) happens in bus-width-size "chunks".
Aside: How does computer memory work?

One memory transaction (simplified):

- **Processor**
  - D0..15
  - A0..15
  - R/W
  - CLK

- **Memory**

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

Andreas Klöckner
Scripting GPUs with PyOpenCL
Aside: How does computer memory work?

One memory transaction (simplified):
Aside: How does computer memory work?

One memory transaction (simplified):

Processor

\[\begin{array}{c}
D0..15 \\
A0..15 \\
R/W \\
CLK
\end{array}\]

Memory
Aside: How does computer memory work?

One memory transaction (simplified):

Processor

<table>
<thead>
<tr>
<th>D0..15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0..15</td>
</tr>
<tr>
<td>R/W</td>
</tr>
<tr>
<td>CLK</td>
</tr>
</tbody>
</table>

Memory
Aside: How does computer memory work?

One memory transaction (simplified):

- Processor
  - D0..15
  - A0..15
  - R/W
  - CLK
- Memory

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

Andreas Klöckner
Scripting GPUs with PyOpenCL
Problem

Memory chips have only one data bus.

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

Memory chips have only one data bus.

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

Solutions: Parallel Access to Memory

- Split a really wide data bus, but have only one address bus
- Have many “small memories” (“banks”) with separate data and address busses, select by address LSB.
Naive: Using Global Memory

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

    a_t[write_idx] = a[read_idx];
}
"""
).build().transpose
```
Naive: Using Global Memory

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

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

Reading from global mem:

![Reading from global mem]

```
...     ...     ...
```

stride: 1
Naive: Using Global Memory

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

Reading from global mem:

```
|     |     |     |     |     |
```

stride: 1 → one mem.trans.
Naive: Using Global Memory

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

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

Reading from global mem:
```
| | | | | ...
```

stride: 1 → one mem.trans.

Writing to global mem:
```
| | | | | |
```

stride: 16
Naive: Using Global Memory

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

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

Reading from global mem:
```
...  ...
```

stride: 1 → one mem.trans.

Writing to global mem:
```
...  ...
```

stride: 16 → 16 mem.trans!
Local Memory: Banking

- Nvidia hardware has 16 banks.
- Work item access local memory in groups of 16.

Andreas Klöckner

Scripting GPUs with PyOpenCL
Local Memory: Banking

- **Bank 0**: 0, 4, 8, 12, 16, 20, ...
- **Bank 1**: 1, 5, 9, 13, 17, 21, ...
- **Bank 2**: 2, 6, 10, 14, 18, 22, ...
- **Bank 3**: 3, 7, 11, 15, 19, 23, ...

- **Work Item Access**: Accessing local memory in groups of 16.
- **Nvidia Hardware**: Has 16 banks.

- **Address**
- **OK**: Variables accessed within a single cycle.
- **Bad**: Variables accessed requiring multiple cycles.

- **Andreas Klöckner**: Scripting GPUs with PyOpenCL
Local Memory: Banking

- Work item access local memory in groups of 16.
- Nvidia hardware has 16 banks.

0 4 8 12 16 20

3 7 11 15 19 23

2 6 10 14 18 22

1 5 9 13 17 21

0 2 4 6 8 10

OK: `local variable[get local id(0)]`, (Single cycle)
Bad: `local variable[BANK COUNT*get local id(0)]`, (BANK COUNT cycles)
OK: `local variable[(BANK COUNT+1)*get local id(0)]`, (Single cycle)
OK: `local variable[ODD NUMBER*get local id(0)]`, (Single cycle)
Bad: `local variable[2*get local id(0)]`, (BANK COUNT/2 cycles)
OK: `local variable[f(blockIdx)]`, (Broadcast–single cycle)
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

Bank

Work Item

Address

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

Nvidia hardware has 16 banks.
Work item access local memory in groups of 16.

Andreas Klöckner
Scripting GPUs with PyOpenCL
Nvidia hardware has 16 banks. Work item access local memory in groups of 16.
Transposing: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn’t mind.
Transpose: Idea

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

Idea

- Don’t transpose element-by-element.
- Transpose block-by-block instead.
Transpose: Idea

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

Idea

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

1. Read untransposed block from global and write to local
2. Read block transposed from local and write to global
Illustration: Blockwise Transpose

$C_{1,1}$ $C_{1,2}$ $C_{1,3}$ $C_{1,4}$

$C_{2,1}$ $C_{2,2}$ $C_{2,3}$ $C_{2,4}$

$C_{3,1}$ $C_{3,2}$ $C_{3,3}$ $C_{3,4}$

$C_{4,1}$ $C_{4,2}$ $C_{4,3}$ $C_{4,4}$

$C^T_{1,1}$ $C^T_{2,1}$ $C^T_{3,1}$ $C^T_{4,1}$

$C^T_{1,2}$ $C^T_{2,2}$ $C^T_{3,2}$ $C^T_{4,2}$

$C^T_{1,3}$ $C^T_{2,3}$ $C^T_{3,3}$ $C^T_{4,3}$

$C^T_{1,4}$ $C^T_{2,4}$ $C^T_{3,4}$ $C^T_{4,4}$
Improved: With Local Memory

Part 1/3:

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

__kernel void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
```
Improved: With Local Memory

Part 2/3:

```c
{  
    __local float a_local [BLOCK_SIZE][BLOCK_SIZE];
    int base_idx_a =
        get_group_id (0) * BLOCK_SIZE +
        get_group_id (1) * A_BLOCK_STRIDE;
    int base_idx_a_t =
        get_group_id (1) * BLOCK_SIZE +
        get_group_id (0) * A_T_BLOCK_STRIDE;

    int glob_idx_a =
        base_idx_a + get_local_id (0)
        + a_width * get_local_id (1);
    int glob_idx_a_t =
        base_idx_a_t + get_local_id (0)
        + a_height * get_local_id (1);
}
```
Improved: With Local Memory

Part 3/3:

```c
    a_local[ get_local_id (1)][ get_local_id (0)] = a[glob_idx_a ];
    barrier (CLK_LOCAL_MEM_FENCE);
    a_t[glob_idx_a_t ] = a_local[ get_local_id (0)][ get_local_id (1)];
```
Improved: With Local Memory

Launch Code:

```python
w, h = shape

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

Transpose example is `2-transpose/transpose.py` in your home directory. Spot any bank conflicts? Tinker away!
Performance: Matrix transpose

Much better. Not peak, but good enough.
Outline

1. Intro: GPUs, OpenCL

2. GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - Dealing with Space: Memory
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3. Additional Topics

4. Playtime!

5. Conclusions
What is a Barrier?
What is a Barrier?
Synchronization

What is a Barrier?
Synchronization

What is a Barrier?
What is a Barrier?
Synchronization

What is a Barrier?
Synchronization

What is a Barrier?
Synchronization

What is a Memory Fence?

17
Synchronization

What is a Memory Fence?

write 18

17
Synchronization

What is a Memory Fence?

![Diagram showing memory fence](image)

17 write 18 read
What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

17
Synchronization

What is a Memory Fence?

write 18
What is a Memory Fence?
What is a Memory Fence? An ordering restriction for memory access.
Synchronization

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

write 18

17
Synchronization

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

write 18
write 17
What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
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-block:**
  - `barrier(...)`,
  - `mem_fence(...)`
  
  
  \[
  \ldots = \text{CLK}_\{\text{LOCAL,GLOBAL}\}_\text{MEM_FENCE}
  \]

- **Inter-block:**
  - 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
Outline

1. Intro: GPUs, OpenCL

2. GPU Programming with PyOpenCL
   - First Contact
   - A more Detailed Look
   - Dealing with Space: Memory
   - Dealing with Time: Synchronization
   - What PyOpenCL brings to the Table

3. Additional Topics

4. Playtime!

5. Conclusions
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:

- OpenCL 1.1
- Every GetInfo() query
- Images and Samplers
- Memory Maps
- Profiling and Synchronization
- GL Interop (example in source)
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.
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(100000).astype(numpy.float32)
b = numpy.random.rand(100000).astype(numpy.float32)
ctx = cl.Context()
queue = cl.CommandQueue(ctx)
mf = cl.mem_flags
a_buf = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=a)
b_buf = cl.Buffer(ctx, mf.READ_WRITE | mf.COPY_HOST_PTR, hostbuf=b)
dest_buf = cl.Buffer(ctx, mf.WRITE_ONLY, b.shape)
prog = cl.Program(ctx, ...
    __kernel void sum(const float *a, const float *b, float *c)
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
...).build()
prog.sum(queue, a.shape, a_buf, b_buf, dest_buf)
a_plus_b = numpy.empty_like(a)
c1 = numpy.random.randn(len(queue), dest_buf, a_plus_b, wait() print la.norm(a_plus_b - (a+b))
```

(You can find this example as `examples/sum.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, Boost C++, Python 2.4+.
- Support via mailing list.
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
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
```
Avoiding extra store-fetch cycles for elementwise math:

```
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]"
)

import numpy.linalg as la

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

1 Intro: GPUs, OpenCL

2 GPU Programming with PyOpenCL

3 Additional Topics
   - Code Generation
   - Other GPU Gadgetry
   - GPU Architectures in more Detail
   - Automatic GPU Programming

4 Playtime!

5 Conclusions
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
   - Code Generation
   - Other GPU Gadgetry
   - GPU Architectures in more Detail
   - Automatic GPU Programming
4. Playtime!
5. Conclusions
The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards
OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

Devices differ by

- Memory Types, Latencies, Bandwidths
- Vector Widths
- Units of Scheduling
The OpenCL Ecosystem: One Language, Many Devices

OpenCL generalizes over many types of devices:

- Multicore CPUs
- Various GPU architectures
- Accelerator boards

Devices differ by

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

Optimally tuned code will (often) be different for each device
In GPU scripting, GPU code does not need to be a compile-time constant.
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

(Key: Code is data—it wants to be reasoned about at run time)
In GPU scripting, GPU code does \textit{not} need to be a compile-time constant.

(Key: Code is data—it \textit{wants} to be reasoned about at run time)
In GPU scripting, GPU code does not need to be a compile-time constant.

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

(Good for code generation)

**In PyOpenCL**

*Python Code* 

*GPU Code*

*GPU Compiler*

*GPU Binary*

*GPU*

*Result*
Machine-generated Code

Why machine-generate code?

- Automated Tuning (cf. ATLAS, FFTW)
- Data types
- Specialize code for given problem
- Constants faster than variables (→ register pressure)
- Loop Unrolling
PyOpenCL: Support for Metaprogramming

Three (main) ways of generating code:

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

Many ways of evaluating code—most important one:

- Exact device timing via events
RTCG via Templates

```python
from jinja2 import Template

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

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

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

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

Andreas Klöckner  
Scripting GPUs with PyOpenCL
RTCG via AST Generation

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

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

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

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
   - Code Generation
   - Other GPU Gadgetry
   - GPU Architectures in more Detail
   - Automatic GPU Programming
4. Playtime!
5. Conclusions
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible! → Interruptible!

How?

```c
atomic {
    add, inc, cmpxchg, ...
} (int *global, int value);
```
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Atomic Global Memory Update:
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

Protected

Andreas Klöckner
Scripting GPUs with PyOpenCL
Atomic Operations

Collaborative (inter-block) Global Memory Update:

- Read
- Increment
- Write

Interruptible!

Atomic Global Memory Update:

- Read
- Increment
- Write

Protected

How?

```
atomic_{add,inc,cmpxchg,...}(int *global, int value);
```
Even more GPU Gadgetry

Available in GPU code:

- Floating point intrinsics
  - `native_sin(x)`, `native_cos(x)`, etc.
  - Very fast
  - Less accurate, limited domains

- Vector types
  - `int/float n` for `n` in `1,2,3,4,8,...`
  - Plus functions: load/store/sum/dot
  - Much saner than SSE intrinsics

- Images (r/w through texture units)
  - Can do filtering
  - Has some cache
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
   - Code Generation
   - Other GPU Gadgetry
   - GPU Architectures in more Detail
   - Automatic GPU Programming
4. Playtime!
5. Conclusions
1 GPU = 30 SIMDs
1 SIMD = 1 ID (1/4 clock)
+ 8 SP + 1 DP +
16 KiB Shared +
32 KiB Reg + HW Sched
Scalar cores, deep pipeline
32 scheduling slots
DDR3 RAM (140 GB/s)
PCle2 Host DMA (6 GB/s)
Limited Caches
GPU Architecture (e.g. ATI RV870)

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

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
   - Code Generation
   - Other GPU Gadgetry
   - GPU Architectures in more Detail
   - Automatic GPU Programming
4. Playtime!
5. Conclusions
Automating GPU Programming

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

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

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

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

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

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

```python
a, b, c, i, j, k = [var(s) for s in "abcijk"]
n = 500
k = make_loop_kernel([
    LoopDimension("i", n),
    LoopDimension("j", n),
    LoopDimension("k", n),
], [
    (c[i+n*j], a[i+n*k]*b[k+n*j])
])
gen_kwargs = {
    "min_threads": 128,
    "min_blocks": 32,
}
```

→ Ideal case: Finds 160 GF/s kernel without human intervention.
Limited scope:
- Require input/output separation
- Kernels must be expressible using “loopy” model (i.e. indices decompose into “output” and “reduction”)
- Enough for DG, LA, FD, ...
Loo.py Status

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

- Kernel compilation limits trial rate
- Non-Goal: Peak performance
- Good results currently for dense linear algebra and (some) DG subkernels
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
4. Playtime!
   - Fun with Reduction
5. Conclusions
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
4. Playtime!
   - Fun with Reduction
5. Conclusions
Parallel Reduction

- Tree-based approach used within each thread block
  - 3 1 7 0 4 1 6 3
  - 4 7 5 9 11 14 25
  - But how do we communicate partial results between thread blocks?

- Need to be able to use multiple thread blocks
  - To process very large arrays
  - To keep all multiprocessors on the GPU busy
  - Each thread block reduces a portion of the array

Slides by M. Harris (Nvidia Corp.)
Solution: Kernel Decomposition

- Avoid global sync by decomposing computation into multiple kernel invocations

- In the case of reductions, code for all levels is the same
  - Recursive kernel invocation

Slides by M. Harris (Nvidia Corp.)
Interleaved Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2</td>
</tr>
</tbody>
</table>

Step 1
Thread IDs

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2</td>
</tr>
</tbody>
</table>

Step 2
Thread IDs

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

Step 3
Thread IDs

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>

Step 4
Thread IDs

<table>
<thead>
<tr>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2</td>
</tr>
</tbody>
</table>
Interleaved Addressing

Values (shared memory):

<table>
<thead>
<tr>
<th>Step 1</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 1</td>
<td>0, 2, 4, 6, 8, 10, 12, 14</td>
<td>10, 1, 8, -2, 3, 5, -2, 2, 7, 0, 11, 0, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 2</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 2</td>
<td>0, 4, 8, 12</td>
<td>11, 1, 7, -2, 8, 5, -5, 9, 7, 11, 11, 2, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 3</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 4</td>
<td>0, 8</td>
<td>18, 1, 7, -1, 6, -2, 8, 5, 4, -3, 9, 7, 13, 11, 2, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 8</td>
<td>0</td>
<td>24, 1, 7, -1, 6, -2, 8, 5, 17, -3, 9, 7, 13, 11, 2, 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Step 4</th>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stride 8</td>
<td>0</td>
<td>41, 1, 7, -1, 6, -2, 8, 5, 17, -3, 9, 7, 13, 11, 2, 2</td>
</tr>
</tbody>
</table>

**Issue:** Divergence

Slides by M. Harris (Nvidia Corp.)
Sequential Addressing

Values (shared memory)

<table>
<thead>
<tr>
<th>Thread IDs</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>-2</td>
</tr>
<tr>
<td>8</td>
<td>-3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
</tr>
</tbody>
</table>

Step 1
Stride 8

Values:
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 2
Stride 4

Values:
8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3
Stride 2

Values:
8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4
Stride 1

Values:
21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Values:
41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Sequential addressing is conflict free.
Sequential Addressing

Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2

Step 1 Thread IDs
Stride 8 0 1 2 3 4 5 6 7
Values 8 -2 10 6 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 2 Thread IDs
Stride 4 0 1 2 3
Values 8 7 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 3 Thread IDs
Stride 2 0 1
Values 21 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Step 4 Thread IDs
Stride 1 0
Values 41 20 13 13 0 9 3 7 -2 -3 2 7 0 11 0 2

Better!

Slides by M. Harris (Nvidia Corp.)
Reduction: Further Strategies

Further Strategies:

- Exploit SIMD synchronicity
  - Eliminate a few `barrier()`s
- Amortize cost of index calculation/preparation
  - Not just one item per thread!
- Do as much as possible at compile time
  - Unroll loops
  - Exploit compile-time knowledge of block size, etc.
    (→ metaprogramming: PyCUDA or C++ templates)
Try for yourself: Performance of GPU Reduction

1. In your home directory, find and run 3-reduction/reduction.py.

2. Add event-based timing. Compute memory throughput in GiB/s for a number of vector sizes.
   (e.g. $2^k$ for $k \in \{12, \ldots, 25\}$)

3. Implement and benchmark the improvements discussed previously.

4. What else is missing for peak performance? (Google?)

PyOpenCL docs: [http://documen.tician.de/pyopencl](http://documen.tician.de/pyopencl)

These slides: [http://tiker.net/tmp/scipy10-pyopencl.pdf](http://tiker.net/tmp/scipy10-pyopencl.pdf)
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
4. Playtime!
5. Conclusions
   - A Brief Look at PyCUDA
   - Summary
Outline

1 Intro: GPUs, OpenCL
2 GPU Programming with PyOpenCL
3 Additional Topics
4 Playtime!
5 Conclusions
   ■ A Brief Look at PyCUDA
   ■ Summary
### OpenCL ↔ CUDA: A dictionary

<table>
<thead>
<tr>
<th><strong>OpenCL</strong></th>
<th><strong>CUDA</strong></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>image&lt;nd_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.xxyz</td>
</tr>
<tr>
<td>get_global_id(012)</td>
<td>– (reimplement)</td>
</tr>
</tbody>
</table>
### PyOpenCL ↔ PyCUDA: A (rough) dictionary

<table>
<thead>
<tr>
<th>PyOpenCL</th>
<th>PyCUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>Context</td>
</tr>
<tr>
<td>CommandQueue</td>
<td>Stream</td>
</tr>
<tr>
<td>Buffer</td>
<td>mem_alloc / DeviceAllocation</td>
</tr>
<tr>
<td>Program</td>
<td>SourceModule</td>
</tr>
<tr>
<td>Kernel</td>
<td>Function</td>
</tr>
<tr>
<td>Event (eg. enqueue_marker)</td>
<td>Event</td>
</tr>
</tbody>
</table>
Whetting your appetite

```python
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4).astype(numpy.float32)
gpu = cuda.mem_alloc(a.nbytes)
cuda.memcpy_htod(gpu, a)
```

[This is examples/demo.py in the PyCUDA distribution.]
Whetting your appetite

```python
mod = cuda.SourceModule('"

    __global__ void twice(float *a)
    {
        int idx = threadIdx.x + threadIdx.y*4;
        a[idx] *= 2;
    }
"

func = mod.get_function('"twice"
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```
Whetting your appetite

```python
mod = cuda.SourceModule('"
  __global__ void twice(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
"
')

func = mod.get_function('"twice"
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Compute kernel
Whetting your appetite, Part II

Did somebody say “Abstraction is good”? 
```python
import numpy
import pycuda.autoinit
import pycuda.gpuarray as gpuarray

a_gpu = gpuarray.to_gpu(
    numpy.random.randn(4,4).astype(numpy.float32))
da_doubled = (2*a_gpu).get()

print a_doubled
print a_gpu
```
Outline

1. Intro: GPUs, OpenCL
2. GPU Programming with PyOpenCL
3. Additional Topics
4. Playtime!
5. Conclusions
   - A Brief Look at PyCUDA
   - Summary
Concluding Remarks

- GPU Computing is maturing.
  Now is a great time to start looking at GPUs.
- First factor of 5-10 is usually easy to reach.
- Second factor of 5-10 is a bit harder
  - Usually involves rethinking the algorithm
- Fun time to be in computational science
- Python makes GPUs even more fun
  - With no compromise in performance
- OpenCL presents a huge opportunity:
  - A JIT compiler in a library
  - CPU backends exist (AMD, Apple)
  - → Like weave/codepy/Cython’s pyximport, but un-hacky
Questions?

Thank you for your attention!

http://mathema.tician.de/software/pyopencl
Image Credits

- Fighting chips: flickr.com/oskay [cc]
- Isaiah die shot: VIA Technologies
- RV770 die shot: AMD Corp.
- Nvidia Tesla Architecture: Nvidia Corp.
- C870 GPU: Nvidia Corp.
- Context: sxc.hu/svilen001
- Queue: sxc.hu/cobrasoft
- RAM stick: sxc.hu/gobran11
- CPU: sxc.hu/dimshik
- Onions: flickr.com/darwinbell [cc]
- 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]
- Apples and Oranges: Mike Johnson - TheBusyBrain.com [cc]
- Machine: flickr.com/13521837@N00 [cc]
- Circuitry: flickr.com/oskay [cc]
- Nvidia Tesla Architecture: Nvidia Corp.
- RV870 Architecture: AMD Corp.
- Dictionary: sxc.hu/topfer