Asynchronous and event-driven PyOpenCL programming

Tomasz Rybak
tomasz.rybak@post.pl

PyConPL, 2012-09-16

Table of content

About me

- Debian Maintainer, maintaining PyOpenCL and PyCUDA
- Helping to develop PyOpenCL and PyCUDA
- Currently working at CodiLime
- Worked at University of Geneve
- Worked at Bialystok University of Technology
- Contact
  - Blog: http://tomaszrybak.wordpress.com/
  - Email: tomasz.rybak@post.pl

Increasing hardware parallelism

- Moore's law: increasing transistor density
- Power wall
- Chips' frequency does not increase anymore
- We get more cores instead
- No more automatic performance improvements
- Different programming models
- OpenCL as one of the standards intended to help with programming

OpenCL

- Standard managed by Khronos
- Similar to OpenGL
  - Extensions
  - Different models (embedded, etc.) needed for different devices
- Compiled or binary kernels run on cores separate from CPU
- Based on C, with manual memory management, function pointers, etc.
- Events and asynchronous execution

OpenCL

- Execution units (hardware) hierarchy
  1. Host
  2. Platforms
  3. Computing Devices
  4. Computing Units
  5. Processing Elements
- Memory hierarchy
  1. Global memory
  2. Constant memory
  3. Local memory
  4. Private memory
- Relaxed consistency of memory access
- Cache

Execution (run-time) hierarchy

- Context
- Queue
- Work-group
- Work-item

Execution (run-time) hierarchy

- Context
  - can contain many devices
- Queue

- Work-group
- Work-item

Execution (run-time) hierarchy

- Context
  - can contain many devices
- Queue
  - for one device
- Work-group
- Work-item

Execution models

- Task parallelism
  - One thread running computations
  - Possibility of running many threads at the same time
  - Require out-of-order queue or many queues
- Computation parallelism
  - Many threads running the same computations
- “Memory wall”
- Memory transfer and memory access are often limiting performance

PyOpenCL

- . . . and PyCUDA
- Python wrapper for OpenCL
- Not only wrapper
  - Pythonic
  - Object-oriented
- Stable, but still work in progress
  - extensions
  - high-level programming

OpenCL programming

1. Compile kernels
2. Prepare data
3. Transfer data to device
4. Run computations
5. After finishing computations transfer results from device
6. Free resources

Sample program template

```
platform = pyopencl.get_platforms()[0]
device = platform.get_devices(device_type=GPU)[0]
context = pyopencl.Context(devices=[device])
queue = pyopencl.CommandQueue(context, device=device)

a = numpy.zeros(1000).astype(numpy.float32)
a_gpu = pyopencl.array.Array(context, shape=a.shape)
program = pyopencl.Program(context, """
    __kernel void increase(__global float *a)
    {
        int gid = get_global_id(0);
        a[gid] = a[gid]+1.0;
    }
""").build()

pyopencl.enqueue_copy(queue, a, a_gpu)
program.increase(queue, a.shape, None, a_gpu)
```

Events

- Instruct OpenCL to run some computations
- Do not wait for finishing
- Get Information when computation is finished

```
event = pyopencl.enqueue_copy(queue, a, a_gpu)
event.wait()

event = program.increase(queue, a.shape, None, a_gpu)
while event.get_info(COMMAND_EXECUTION_STATUS) != COMPLETE:
    pass

event = pyopencl.enqueue_copy(queue, a_gpu, b)
event.wait()
```

More events

- Build workflow, instruct OpenCL what about dependencies to determine order in which computations need to be performed
- Similar to Makefile
- Possibility to run computations in parallel if hardware allows

```
event0 = pyopencl.enqueue_copy(queue, a, a_gpu)
eventX = pyopencl.enqueue_copy(queue, b, b_gpu)
event1 = program.increase(queue, a.shape, None, a_gpu,
        wait_for=[event0])
event2 = pyopencl.enqueue_copy(queue, a_gpu, b,
        wait_for=[event1])
event2.wait()
```

Events and queues

- One queue
  - In-order
  - Out-of-order execution
- Many queues
- Need to use events for many queues or for out-of-order execution

```
queue0 = pyopencl.CommandQueue(context)
queue1 = pyopencl.CommandQueue(context)
event0 = pyopencl.enqueue_copy(queue0, a, a_gpu)
event1 = program.increase(queue1, a.shape, None, a_gpu,
        wait_for=[event0])
event2 = pyopencl.enqueue_copy(queue0, a_gpu, b,
        wait_for=[event1])
event2.wait()
```

Many devices

- Create multi-device context
  - Queues still can only serve one device
- Events are necessary
- Memory transfers
  - Implicit and explicit memory transfers

```
devices = platform.get_devices()
context = pyopencl.Context(devices)
queue0 = pyopencl.CommandQueue(context, device=devices[0])
queue1 = pyopencl.CommandQueue(context, device=devices[1])
event0 = pyopencl.enqueue_copy(queue0, a, a_gpu)
event1 = pyopencl.enqueue_copy(queue1, b, b_gpu)
event2 = program.increase(queue0, a.shape, None, a_gpu,
        wait_for=[event0])
event3 = program.increase(queue1, b.shape, None, a_gpu,
        wait_for=[event1])
event4 = pyopencl.enqueue_copy(queue0, a_gpu, a,
        wait_for=[event2])
event5 = pyopencl.enqueue_copy(queue1, b_gpu, b,
```

Event-related objects

- Not all PyOpenCL functions and methods accept list of events to wait for
- We may manually wait for those events
- . . . or we can create marker or barrier
  - pyopencl.enqueue_barrier(queue, wait_for)
  - pyopencl.enqueue_marker(queue, wait_for)
- Wait-for list may be empty

Fission

- Splitting one physical device into many logical
- Can be used to reserve some computation power
- Solution similar to CPU virtualisation
- No problems with device-to-device memory transfers

```
device = platform.get_devices()[0]
devices = device.create_sub_devices([[EQUALLY, 3]])
context = pyopencl.Context(devices)
queue0 = pyopencl.CommandQueue(context, device=devices[0])
queue1 = pyopencl.CommandQueue(context, device=devices[1])
```

High-level programming

- Traditional programming — write kernels by hand
- Similar code in different kernels
- PyOpenCL help
  - Array
  - Random number generators
  - Single-pass element-wise expressions
  - Reduction (does not return Event!)
  - Parallel scan (does not return Event!)
- Long development history
  - Initial implementations: limited functionality (prefix sum)
  - Later more generic parallel prefix scan
  - Now very generic and offering much features
    - Segmented
    - Look-behind on input and output
    - Map-scan (transformed scan)
- Customisable with code snippets

Element-wise kernel

```
copy_depth = pyopencl.elementwise.ElementwiseKernel(context,
    "freenect_data *output, unsigned short *input",
    "output[i].d = input[i]",
    "copy_depth", preamble=c_decl)
copy_depth(a_gpu, b_gpu)
```

Extensions

- Allowing to add features by hardware vendors
- List of extensions supported by platform
- OpenCL returns function pointer for given extension function
- Each extension must be added manually
  - PyOpenCL implements fission extension

OpenGL

- Cooperation between OpenGL and OpenCL
- Passing events
  - Only through extensions, not implemented in PyOpenCL yet
- Sharing memory
- Creating objects from memory belonging to the other side
- Implemented, works in both ways

PyOpenCL future

- Intention to share code between PyOpenCL and PyCUDA (compyte)
- Small set of surrounding libraries (e.g. pyfft)
- Some of those could be added to PyOpenCL, some will remain separate
- Problems
  - Adding extensions
  - Supporting additional libraries (rather PyCUDA problem)

OpenCL support

- Problems with various OpenCL libraries and versions

OpenCL support

- Problems with various OpenCL libraries and versions
- NVIDIA seems hostile to OpenCL (and Linux?)
- They are in Khronos management board
- But have not yet implemented OpenCL 1.2
- Removed OpenCL examples from SDK
- Problems with using NVIDIA laptop GPUs on Linux

Hardware future

- Can be already observed in mobile chips
- Many different cores
  - SIMD, Hyper Threading — cheap ways of increasing parallel performance
  - Few full featured cores
  - Many small slower cores
  - Maybe specialised parts
- More sophisticated hardware (NVIDIA Kepler can run kernels from inside kernels)
- AMD Bulldozer — sharing floating point units among cores
-
- S. Borkar and A. A. Chien (Intel), “The Future of Microprocessors”, CACM 2011-05
- T. Ungerer et al., “A survey of processors with explicit multithreading”, ACM Computing Survey 2003-03

Q & A

Thank you for your attention.

Questions?

Contact me

- Blog: http://tomaszrybak.wordpress.com/
- Email: tomasz.rybak@post.pl
- Skype: rybakthomas
- GTalk: tomasz.rybak@gmail.com
- Phone: +48-661-972-777