PyOpenCL - unleash your GPU with the help of Python

Tomasz Rybak
tomasz.rybak@post.pl

PyConUA, 2012-10-21

Simple program for the beginning

```
import pyopencl
import pyopencl.array
import numpy

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

a = numpy.random.rand(10000).astype(numpy.float32)
a_gpu = pyopencl.array.to_device(queue, a)
b_gpu = 2*a_gpu+1
print (a_gpu, b_gpu)
```

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

Hardware scenery

- Moore's law: increasing transistor density
  - More transistors
  - Higher frequency
  - "Power wall"
  - Chips' frequency does not increase anymore
  - No more automatic performance improvements
- Speed of light — ultimate limit
  - Limits the size of the single chip
  - Limits the latency when accessing memory
  - "Memory wall"
- More cores instead of single faster core
- Harder to keep cache consistency amongst many cores
- Different programming models

GPU

- Highly parallel computations
- Mathematical primitives
  - Vertex
  - Matrix
  - Texture
- Large memory
- Different available memory types

Programming GPU

- Shaders
- CUDA
  - Managed by NVIDIA
- OpenCL
  - Standard managed by Khronos
  - Similar to OpenGL
  - Compiled or binary kernels run on cores separate from CPU
  - Based on C, with manual memory management, function pointers, etc.
- PTX

OpenCL

- Hardware execution hierarchy
  1. Processing Elements
     - Warp or wave-front
  2. Computing Units
  3. Computing Devices
  4. Platforms
  5. Host

- Runtime execution hierarchy
  - Work-item
    - Warp or wave-front
  - Work-group
  - Queue
    - for one device
  - Context
    - can contain many devices

Memory hierarchy

1. Global memory
2. Constant memory
3. Local memory
4. Private memory

- Texture memory, Image memory
- Relaxed consistency of memory access
- Cache

Execution models

- Task-level parallelism
  - One thread running computations
  - Possibility of running many threads at the same time
  - Require out-of-order queue or many queues
- Thread-level parallelism
  - Many threads running the same computations

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)
a = a_gpu.get()
```

OpenCL programming

PyOpenCL is compatible with Python3!

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

Event-based programming

- Build workflow, instruct OpenCL 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()

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

Performance considerations

- Massive parallelism
  - Limited memory per core
  - e.g. GTX460: 1GB for 336 cores: 3MB per core
- Locality and Hierarchy
  - "Memory wall"
  - Memory transfer and memory access are often limiting performance
- Load Balancing
  - Hardware occupancy
  - Task and computation parallelism

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,
```

Problems with traditional GPU programming

- Writing kernels by hand
- Similar code in different kernels
- Similar code to prepare data, execute kernel, etc.

High-level programming

- PyOpenCL can help here
  - 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

```
compute = pyopencl.elementwise.ElementwiseKernel(context,
    "float *a, int b, float *c, float d, float *e",
    "a[i] = c[i]/b + e[i]*e[i]*d",
    "compute_this")
compute(a_gpu, 10, b_gpu, 3.14, c_gpu)
```

Reduction kernel

```
compute = pyopencl.reduction.ReductionKernel(context,
    numpy.float32, neutral="0",
    reduce_expr="a+b", map_expr="x[i]*x[i]-sin(y[i])",
    arguments="float *x, float *y")
result = compute(a_gpu, b_gpu)
```

Prefix-scan kernel

```
compute = pyopencl.elementwise.ElementwiseKernel(context,
    numpy.float32, neutral="0",
    arguments="float *a, float *out",
    input_exp="(a[i] > 0) ? a[i] : 0",
    scan_expr="a+b",
    output_statement="out[i] = item")
compute(a_gpu, b_gpu)
```

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

Open problems

- Developers' feedback
  - standards
  - libraries
- Hardware changes
  - Multi-GPU or GPU-CPU support are still in infancy
  - model confusion (Kepler and GTX 6XX)
  - need for manual optimisation
- GPGPU seen as niche

Links

- http://www.khronos.org/opencl/
- http://mathema.tician.de/software/pyopencl
- http://mathema.tician.de/software/pycuda
- http://developer.amd.com/RESOURCES/HC/OPENCLZONE/Page
- http://developer.nvidia.com/category/zone/cuda-zone

Q & A

Thank you for your attention.

Questions?

Contact me

- Blog: http://tomaszrybak.wordpress.com/
- Email: tomasz.rybak@post.pl