programming graphics processing units

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code

MCS 572 Lecture 29
Introduction to Supercomputing
Jan Verschelde, 28 October 2016
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
OpenCL: Open Computing Language

OpenCL, the Open Computing Language, is the open standard for parallel programming of heterogeneous system.

OpenCL is maintained by the Khronos Group — a not for profit industry consortium creating open standards for the authoring and acceleration of parallel computing, graphics, dynamic media, computer vision and sensor processing on a wide variety of platforms and devices — with home page at www.khronos.org.

Another related standard is OpenGL (www.opengl.org), the open standard for high performance graphics.

The development of OpenCL was initiated by Apple.

Many aspects of OpenCL are familiar to a CUDA programmer because of similarities with data parallelism and complex memory hierarchies.

OpenCL offers a more complex platform and device management model to reflect its support for multiprocessor and multivendor portability.

OpenCL implementations exist for AMD ATI and NVIDIA GPUs as well as x86 CPUs.

The code in this lecture runs on an Intel Iris Graphics 6100, the graphics card of a MacBook Pro.
about PyOpenCL


Same benefits of PyOpenCL as PyCUDA:

- takes care of a lot of “boiler plate” code;
- focus on the kernel, with numpy typing.

Instead of a programming model tied to a single hardware vendor’s products, open standards enable portable software frameworks for heterogeneous platforms.
a sanity check on the installation

PyOpenCL can be installed with pip, just do

$ sudo pip install pyopencl

Then we launch python:

```python
>>> import pyopencl
>>> from pyopencl.tools import get_test_platforms_and_devices
>>> get_test_platforms_and_devices()
[(<pyopencl.Platform 'Apple' at 0x7fff0000>,
  [<pyopencl.Device 'Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz'
    on 'Apple' at 0xffffffff>,
  <pyopencl.Device 'Intel(R) Iris(TM) Graphics 6100'
    on 'Apple' at 0x1024500>])
>>>```

A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
matrix matrix multiplication

Our running example will be the multiplication of two matrices:

$ python matmatmulocl.py

matrix A:

[[ 0. 0. 1. 1.]
 [ 1. 1. 1. 1.]
 [ 1. 1. 1. 1.]]

matrix B:

[[ 1. 1. 0. 1. 1.]
 [ 1. 1. 0. 1. 1.]
 [ 0. 0. 1. 0. 1.]
 [ 1. 0. 1. 0. 1.]]

multiplied A*B:

[[ 1. 0. 2. 0. 2.]
 [ 3. 2. 3. 1. 4.]
 [ 3. 2. 3. 1. 4.]]

$
The script `matmatmulocl.py`

```python
import pyopencl as cl
import numpy as np
import os

# context: 0 for Apple, 1 for the graphics card
os.environ['PYOPENCL_CTX'] = '0:1'

(n, m, p) = (3, 4, 5)

a = np.random.randint(2, size=(n*m))
b = np.random.randint(2, size=(m*p))
c = np.zeros((n*p), dtype=np.float32)

a = a.astype(np.float32)
b = b.astype(np.float32)
```

Introduction to Supercomputing (MCS 572)
Introduction to OpenCL
context, queue, and buffers

```python
ctx = cl.create_some_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)
c_buf = cl.Buffer(ctx, mf.WRITE_ONLY, c.nbytes)
```
defining the kernel

prg = cl.Program(ctx, ""
__kernel void multiply(ushort n,
ushort m, ushort p, __global float *a,
__global float *b, __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = 0.0f;
    int rowC = gid/p;
    int colC = gid%p;
    __global float *pA = &a[rowC*m];
    __global float *pB = &b[colC];
    for(int k=0; k<m; k++)
    {
        pB = &b[colC+k*p];
        c[gid] += (*(pA++))*(*pB);
    }
}
""" ).build()
executing the program

```python
prg.multiply(queue, c.shape, None,
              np.uint16(n), np.uint16(m), np.uint16(p),
              a_buf, b_buf, c_buf)

a_mul_b = np.empty_like(c)
cl.enqueue_copy(queue, a_mul_b, c_buf)
# Python 3 version of print statements
print("matrix A:")
print(a.reshape(n, m))
print("matrix B:"
print(b.reshape(m, p))
print("multiplied A*B:"
print(a_mul_b.reshape(n, p))
```
running the NVIDIA OpenCL SDK

$ python matmatmulsdk.py
GPU push+compute+pull total [s]: 0.0844735622406
GPU push [s]: 0.000111818313599
GPU pull [s]: 0.0014328956604
GPU compute (host-timed) [s]: 0.0829288482666
GPU compute (event-timed) [s]: 0.08261928

GFlops/s: 24.6958693242

GPU==CPU: True

CPU time (s) 0.0495228767395

GPU speedup (with transfer): 0.586252969875
GPU speedup (without transfer): 0.59717309205
$

The operating principle of GPU code generation:

PyCUDA is installed on *kepler* and *pascal*.
checking the installation on pascal

```
[jan@pascal ~]$ python
Python 2.7.5 (default, Sep 15 2016, 22:37:39) 
[GCC 4.8.5 20150623 (Red Hat 4.8.5-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycuda
>>> import pycuda.autoinit
>>> from pycuda.tools import make_default_context
>>> c = make_default_context()
>>> d = c.get_device()
>>> d.name()
'Tesla P100-PCIE-16GB'
```
running the script

We multiply an \( n \text{-by-} m \) matrix with an \( m \text{-by-} p \) matrix with a two dimensional grid of \( n \times p \) threads. For testing we use 0/1 matrices.

\[
\text{$\begin{array}{cccc}
0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 0 \\
0 & 1 & 1 & 0
\end{array}$}
\]

\[
\text{$\begin{array}{cccc}
1 & 1 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 0
\end{array}$}
\]

multiplied \( A \times B \):

\[
\text{$\begin{array}{cccc}
0 & 0 & 1 & 1 & 0 \\
0 & 0 & 2 & 2 & 0 \\
1 & 0 & 2 & 1 & 0
\end{array}$}
\]
import pycuda.driver as cuda
import pycuda.autoinit
from pycuda.compiler import SourceModule
import numpy

(n, m, p) = (3, 4, 5)

n = numpy.int32(n)
m = numpy.int32(m)
p = numpy.int32(p)

# a = numpy.random.randn(n, m)
# b = numpy.random.randn(m, p)
a = numpy.random.randint(2, size=(n, m))
b = numpy.random.randint(2, size=(m, p))
c = numpy.zeros((n, p), dtype=numpy.float32)
allocation and copy from host to device

```python
a_gpu = cuda.mem Alloc(a.size * a.dtype.itemsize)
b_gpu = cuda.mem Alloc(b.size * b.dtype.itemsize)
c_gpu = cuda.mem Alloc(c.size * c.dtype.itemsize)

cuda.memcpy_htod(a_gpu, a)
cuda.memcpy_htod(b_gpu, b)
```
mod = SourceModule(""
    __global__ void multiply
    ( int n, int m, int p,
        float *a, float *b, float *c )
    {
        int idx = p*threadIdx.x + threadIdx.y;
        c[idx] = 0.0;
        for(int k=0; k<m; k++)
            c[idx] += a[m*threadIdx.x+k] * b[threadIdx.y+k*p];
    }
    "")
launching the kernel

```python
func = mod.get_function("multiply")
func(n, m, p, a_gpu, b_gpu, c_gpu, 
    block=(numpy.int(n), numpy.int(p), 1), 
    grid=(1, 1), shared=0)

cuda.memcpy_dtoh(c, c_gpu)

print "matrix A:"
print a
print "matrix B:"
print b
print "multiplied A*B:"
print c
```
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
grids, blocks, and threads

The code that runs on the GPU is defined in a function, the kernel. A kernel launch

- creates a grid of blocks, and
- each block has one or more threads.

The organization of the grids and blocks can be 1D, 2D, or 3D.

During the running of the kernel:

- Threads in the same block are executed simultaneously.
- Blocks are scheduled by the streaming multiprocessors.

The NVIDIA Tesla C2050 has 14 streaming multiprocessors and threads are executed in groups of 32 (the warp size). This implies: \(14 \times 32 = 448\) threads can run simultaneously.

For the K20c the numbers are respectively 13, 192, and 2496.
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
OpenCL and CUDA concepts

After launching a kernel, its code is executed by *work items*. Work items form *work groups*, which correspond to CUDA blocks.

An index space defines how data are mapped to work items.

<table>
<thead>
<tr>
<th>OpenCL concept</th>
<th>CUDA equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>kernel</td>
</tr>
<tr>
<td>host program</td>
<td>host program</td>
</tr>
<tr>
<td>NDRange (index space)</td>
<td>grid</td>
</tr>
<tr>
<td>work group</td>
<td>block</td>
</tr>
<tr>
<td>work item</td>
<td>thread</td>
</tr>
</tbody>
</table>
mapping memory types

Like CUDA, OpenCL exposes a hierarchy of memory types. The mapping of OpenCL memory types to CUDA is:

<table>
<thead>
<tr>
<th>OpenCL memory type</th>
<th>CUDA equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>global memory</td>
<td>global memory</td>
</tr>
<tr>
<td>constant memory</td>
<td>constant memory</td>
</tr>
<tr>
<td>local memory</td>
<td>shared memory</td>
</tr>
<tr>
<td>private memory</td>
<td>local memory</td>
</tr>
</tbody>
</table>

Local memory in OpenCL and shared memory in CUDA are accessible respectively to a work group and thread block.

Private memory in OpenCL and local memory in CUDA is memory accessible only to individual threads.
## dimensions and indices

All work items have their own unique global index values.

<table>
<thead>
<tr>
<th>OpenCL API call</th>
<th>CUDA equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_global_id(0)</code></td>
<td><code>blockIdx.x \times blockDim.x + threadIdx.x</code></td>
</tr>
<tr>
<td><code>get_local_id(0)</code></td>
<td><code>threadIdx.x</code></td>
</tr>
<tr>
<td><code>get_global_size(0)</code></td>
<td><code>gridDim.x \times blockDim.x</code></td>
</tr>
<tr>
<td><code>get_local_size(0)</code></td>
<td><code>blockDim.x</code></td>
</tr>
</tbody>
</table>

Replacing 0 in `get_global_id(0)` by 1 and 2 gives the values for the $y$ and $z$ dimensions respectively.
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
overview of the OpenCL parallel execution model

from Chris Lamb, *NVIDIA: OpenCL and other interesting stuff.*
the OpenCL device architecture

from Chris Lamb, *NVIDIA: OpenCL and other interesting stuff.*
### basic OpenCL program structure

<table>
<thead>
<tr>
<th>host program</th>
<th>platform layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>query platform</td>
<td></td>
</tr>
<tr>
<td>query compute devices</td>
<td></td>
</tr>
<tr>
<td>create contexts</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>runtime</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>create memory objects associate the contexts</td>
<td></td>
</tr>
<tr>
<td>compile and create kernel program objects</td>
<td></td>
</tr>
<tr>
<td>issue commands to command queue</td>
<td></td>
</tr>
<tr>
<td>synchronization of commands</td>
<td></td>
</tr>
<tr>
<td>clean up OpenCL resources</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>language</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>kernels</td>
<td></td>
</tr>
<tr>
<td>OpenCL C code</td>
<td></td>
</tr>
</tbody>
</table>

---

L-29  28 October 2016  30 / 45
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
hello world example by Apple

A simple Hello World uses OpenCL to compute the square for a buffer of floating point values.

Compiling and running:

```
$ gcc -o /tmp/hello hello_opencl.c -framework OpenCL

$ /tmp/hello
Computed '1024/1024' correct values!
```

const char *KernelSource = "\n" \n"__kernel void square(\n" __global float* input, \n" __global float* output, \n" const unsigned int count) \n{" \n" int i = get_global_id(0); \n" if(i < count) \n" output[i] = input[i] * input[i]; \n"\n"\n";
A Brief Introduction to OpenCL

1. PyOpenCL and PyCUDA
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

2. Thread Organization
   - grids, blocks, and threads

3. Data Parallelism Model
   - dictionaries between OpenCL and CUDA
   - the OpenCL parallel execution model

4. Writing OpenCL Programs
   - hello world example by Apple
   - looking at the code
the code

```c
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <OpenCL/opencl.h>

int main(int argc, char** argv)
{
    int err; // error code returned from api calls
    float data[DATA_SIZE]; // original data given to device
    float results[DATA_SIZE]; // results returned from device
    unsigned int correct; // number of correct results
    size_t global; // global domain size for our calculation
    size_t local; // local domain size for our calculation
```
OpenCL types

cl_device_id device_id;       // compute device id
cl_context context;           // compute context
cl_command_queue commands;    // compute command queue
cl_program program;           // compute program
cl_kernel kernel;             // compute kernel
cl_mem input;                 // device memory used for the input
cl_mem output;                // device memory used for the output

// Fill our data set with random float values

int i = 0;
unsigned int count = DATA_SIZE;
for(i = 0; i < count; i++)
    data[i] = rand() / (float)RAND_MAX;
connect and create context

// Connect to a compute device

int gpu = 1;
err = clGetDeviceIDs(NULL, gpu ?
   CL_DEVICE_TYPE_GPU : CL_DEVICE_TYPE_CPU,
   1, &device_id, NULL);
if(err != CL_SUCCESS)
{
    printf("Error: Failed to create a device group!\n");
    return EXIT_FAILURE;
}

// Create a compute context

c context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
if (!context)
{
    printf("Error: Failed to create a compute context!\n");
    return EXIT_FAILURE;
}
commands and program

// Create a command commands

commands = clCreateCommandQueue(context, device_id, 0, &err);
if (!commands)
{
    printf("Error: Failed to create a command commands!\n");
    return EXIT_FAILURE;
}

// Create the compute program from the source buffer

program = clCreateProgramWithSource(context, 1,
    (const char **) & KernelSource, NULL, &err);
if (!program)
{
    printf("Error: Failed to create compute program!\n");
    return EXIT_FAILURE;
}
building the executable

// Build the program executable

err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
if (err != CL_SUCCESS)
{
    size_t len;
    char buffer[2048];

    printf("Error: Failed to build program executable!\n");
    clGetProgramBuildInfo(program, device_id,
                  CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
    printf("%s\n", buffer);
    exit(1);
}
creating kernel and data

// Create the compute kernel in the program we wish to run

kernel = clCreateKernel(program, "square", &err);
if (!kernel || err != CL_SUCCESS)
{
    printf("Error: Failed to create compute kernel!\n");
    exit(1);
}

// Create the input and output arrays in device memory

input = clCreateBuffer(context, CL_MEM_READ_ONLY,
    sizeof(float) * count, NULL, NULL);
output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(float) * count, NULL, NULL);
if (!input || !output)
{
    printf("Error: Failed to allocate device memory!\n");
    exit(1);
}
write data and kernel arguments

// Write our data set into the input array in device memory

err = clEnqueueWriteBuffer(commands, input, CL_TRUE, 0, sizeof(float) * count, data, 0, NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to write to source array!\n");
    exit(1);
}

// Set the arguments to our compute kernel

err = 0;
err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &input);
err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &output);
err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &count);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to set kernel arguments! %d\n", err);
    exit(1);
}
configuring the execution

// Get the maximum work group size for executing the kernel

err = clGetKernelWorkGroupInfo(kernel, device_id,
    CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to retrieve kernel work group info! %d\n", err)
    exit(1);
}

// Execute the kernel over the entire range of
// our 1d input data set // using the maximum number
// of work group items for this device

global = count;
err = clEnqueueNDRangeKernel(commands, kernel, 1, NULL,
    &global, &local, 0, NULL, NULL);
if (err)
{
    printf("Error: Failed to execute kernel!\n");
    return EXIT_FAILURE;
}
finishing and reading results

// Wait for the command commands to get serviced
// before reading back results

clFinish(commands);

// Read back the results from the device for verification

er = clEnqueueReadBuffer(commands, output, CL_TRUE, 0, sizeof(float) * count, results, 0, NULL, NULL);
if (err != CL_SUCCESS)
{
    printf("Error: Failed to read output array! %d\n", err);
    exit(1);
}
validation and cleanup

correct = 0;
for(i = 0; i < count; i++)
{
    if(results[i] == data[i] * data[i])
        correct++;
}

printf("Computed '%d/%d' correct values!\n", correct, count);

clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(program);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);

return 0;
We started chapter 11 (first edition) of the book of Kirk and Hwu, see chapter 14 in the second edition of the book.

Some programming guides available online: