programming graphics processing units in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy

MCS 507 Lecture 14
Mathematical, Statistical and Scientific Software
Jan Verschelde, 11 February 2022
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
Thanks to the industrial success of video game development graphics processors became faster than general CPUs.

General Purpose Graphic Processing Units (GPGPUs) are available, capable of double floating point calculations.

Accelerations by a factor of 10 with one GPGPU are not uncommon.

Comparing electric power consumption is advantageous for GPGPUs.

Thanks to the popularity of the PC market, millions of GPUs are available – every PC has a GPU. This is the first time that massively parallel computing is feasible with a mass-market product.

Example: Actual clinical applications on magnetic resonance imaging (MRI) use some combination of PC and special hardware accelerators.
kepler versus pascal versus volta

NVIDIA Tesla K20 “Kepler” C-class Accelerator
- 2,496 CUDA cores, $2,496 = 13 \text{ SM} \times 192 \text{ cores/SM}$
- 5GB Memory at 208 GB/sec peak bandwidth
- peak performance: 1.17 TFLOPS double precision

NVIDIA Tesla P100 16GB “Pascal” Accelerator
- 3,586 CUDA cores, $3,586 = 56 \text{ SM} \times 64 \text{ cores/SM}$
- 16GB Memory at 720GB/sec peak bandwidth
- peak performance: 5.3 TFLOPS double precision

NVIDIA Tesla V100 32GB “Volta” Accelerator
- 5,120 CUDA cores, 640 Tensor cores
- 32GB Memory at 870GB/sec peak bandwidth
- peak performance: 7.9 TFLOPS double precision
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
Programming model: Single Instruction Multiple Data (SIMD).

- Data parallelism: blocks of threads read from memory, execute the same instruction(s), write to memory.
- Massively parallel: need 10,000 threads for full occupancy.

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.
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
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.

about OpenCL

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 multiplatform and multivendor portability.

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

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

The current version for python3 is installed on pascal.math.uic.edu.

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

```bash
$ pip3 install pyopencl
```

Then we launch python:

```python
$ python3
>>> import pyopencl
>>> from pyopencl.tools import get_test_platforms_and_devices
>>> get_test_platforms_and_devices()
[(<pyopencl.Platform 'NVIDIA CUDA' at 0x21dd450>,
  [<pyopencl.Device 'Tesla P100-PCIE-16GB' on 'NVIDIA CUDA' at 0x219cd00>,
   <pyopencl.Device 'Quadro K420' on 'NVIDIA CUDA' at 0x220df10>]),
]```
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
matrix matrix multiplication

Our running example will be the multiplication of two matrices:

```bash
$ 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.  1.  0.  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.]]
```

import pyopencl as cl
import numpy as np

import os
os.environ['PYOPENCL_COMPILER_OUTPUT'] = '1'
# 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)
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
cdef 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))
```
$ 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
$
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy

The operating principle of GPU code generation:

PyCUDA is installed on `pascal.math.uic.edu`. 
checking the installation on \texttt{pascal}


code

$ python
>>> 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'

$
checking the installation on a windows laptop

```
$ python3
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
>>> import pycuda
>>> import pycuda.autoinit
>>> from pycuda.tools import make_default_context
>>> c = make_default_context()
>>> d = c.get_device()
>>> d.name()
'GeForce RTX 2080 with Max-Q Design'
```
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
running the script

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

```
$ python matmatmul.py
matrix A:
[[ 0.  0.  1.  0.]
 [ 0.  0.  1.  1.]
 [ 0.  1.  1.  0.]]
matrix B:
[[ 1.  1.  0.  1.  1.]
 [ 1.  0.  1.  0.  0.]
 [ 0.  0.  1.  1.  0.]
 [ 0.  0.  1.  1.  0.]]
multiplied A*B:
[[ 0.  0.  1.  1.  0.]
 [ 0.  0.  2.  2.  0.]
 [ 1.  0.  2.  1.  0.]]
```

$
headers and type declarations

```python
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)
a = a.astype(numpy.float32)
b = b.astype(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)
```
definition of the kernel

```python
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];
    }
"""
```

Scientific Software (MCS 507)  GPU Acceleration in Python  L-14  11 February 2022  27/31
launching the kernel

```
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)
```
GPU Accelerations in Python

1. Graphics Processing Units
   - introduction to general purpose GPUs
   - data parallelism

2. PyOpenCL
   - parallel programming of heterogeneous systems
   - matrix matrix multiplication

3. PyCUDA
   - about PyCUDA
   - matrix matrix multiplication

4. CuPy
   - about CuPy
CuPy

CuPy implements the multi-dimensional array of numpy on CUDA.

CuPy has a large community of developers (on github), under the direction by the company Preferred Networks.

CuPy uses on-the-fly kernel synthesis:

- for a required kernel call, it compiles the code of the kernel, optimizes for shapes and dtypes of the arguments;
- sends the compiled code to the GPU device; and
- executes the kernel.

The kernel code is cached, so the second call executes faster.

Accelerates Chainer, a deep learning framework.
Summary and Exercises

We can accelerate our computations in Python.

Exercises :

1. Investigate the capabilities of the graphics card on your computer. Install PyOpenCL and/or PyCUDA, depending on the type of GPU.
2. Examine the possibilities to extend the code for matrix multiplication to work with 3-dimensional matrices.
3. Compare the theoretical peak performance on the CPU and GPU on your computer.
4. For sufficiently large enough dimension of the matrices, compare the performance of the matrix matrix multiplication on the CPU and GPU on your computer.