E6895 Advanced Big Data Analytics Lecture 13:

**GPU Examples and GPU on iOS devices**

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
IBM Chief Scientist, Graph Computing Research
NVIDIA CUDA Getting Started Guide for Mac OS X

1. Introduction

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables harnessing the power of the graphics processing unit (GPU).

CUDA was developed with several design goals in mind:

- Provide a small set of extensions to standard programming languages, like C, that enable With CUDA C/C++, programmers can focus on the task of parallelization of the algorithms
- Support heterogeneous computation where applications use both the CPU and GPU. Serial portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existing devices that have their own memory spaces. This configuration also allows simultaneous memory resources.

CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads in a register file and a shared memory. The on-chip shared memory allows parallel tasks running on system memory bus.

This guide will show you how to install and check the correct operation of the CUDA development environment.

1.1. System Requirements

To use CUDA on your system, you need to have:

- a CUDA-capable GPU
- Mac OS X 10.9 or later
- the Clang compiler and toolchain installed using Xcode
- the NVIDIA CUDA Toolkit (available from the CUDA Download page)
Example: deviceQuery

Understand the hardware constraint via deviceQuery (in example code of CUDA toolkit)
Example: Matrix Addition on CPU

Problem: Sum two matrices with M by N size.

\[ C_{mxn} = A_{mxn} + B_{mxn} \]

In traditional C/C++ implementation:
- A, B are input matrix, N is the size of A and B.
- C is output matrix
- Matrix stored in array is row-major fashion

```c
void sumArraysOnHost(float *A, float *B, float *C, const int N) {
    for (int idx = 0; idx < N; idx++)
    {
    }
}
```
Example: Matrix Addition on GPU - 2D grid with 2D blocks

Problem: Sum two matrices with M by N size.

\[ C_{mxn} = A_{mxn} + B_{mxn} \]

CUDA C implementation:
- matA, matB are input matrix, nx is column size, and ny is row size
- matC is output matrix

```c
#include <cuda.h>

__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}

int dimx = 32;
int dimy = 32;
dim3 block(dimx, dimy);
dim3 grid((nx + block.x - 1) / block.x, (ny + block.y - 1) / block.y);

iStart = seconds();
sumMatrixOnGPU2D<<<grid, block>>>(d_MatA, d_MatB, d_MatC, nx, ny);
CHECK(cudaDeviceSynchronize());
```
Example: Matrix Addition on GPU - 2D grid with 2D blocks

Data accessing in 2D grid with 2D blocks arrangement (one green block is one thread block)

```c
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}
```

Matrix coordinate: (ix, iy)
Global linear memory index: idx = iy * nx + ix
Example: Matrix Addition on GPU - 1D grid with 1D blocks

Data accessing in 1D grid with 1D blocks arrangement (one green block is one thread block)

```c
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int nx, int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx)
        for (int iy = 0; iy < ny; iy++) {
            int idx = iy * nx + ix;
        }
}
```

Global linear memory index: \( \text{idx} = \text{iy} \times \text{nx} + \text{ix} \)
Data accessing in 2D grid with 1D blocks arrangement (one green block is one thread block)

```c
// grid 2D block 1D
global void sumMatrixOnGPUTile(float *MatA, float *MatB, float *MatC, int nx, int ny) {

    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
}
```
Example: Matrix Transpose on CPU

Problem: Transpose one matrix with M by N to one matrix with N by M

\[ A_{mxn} = B_{nxm} \]

In traditional C/C++ implementation:

- in is input matrix, nx is column size, and ny is row size.
- out is output matrix
- Matrix stored in array is row-major fashion

```c
void transposeHost(float *out, float *in, const int nx, const int ny) {
    for (int iy = 0; iy < ny; ++iy) {
        for (int ix = 0; ix < nx; ++ix) {
            out[ix * ny + iy] = in[iy * nx + ix];
        }
    }
}
```
Example: Matrix Transpose on GPU

```c
// case 2 transpose kernel: read in rows and write in columns
__global__ void transposeNaiveRow(float *out, float *in, const int nx,
                                 const int ny) {
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[ix * ny + iy] = in[iy * nx + ix];
  }
}

// case 3 transpose kernel: read in columns and write in rows
__global__ void transposeNaiveCol(float *out, float *in, const int nx,
                                 const int ny) {
  unsigned int ix = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
  if (ix < nx && iy < ny) {
    out[iy * nx + ix] = in[ix * ny + iy];
  }
}
```
Example: Matrix Transpose on GPU

```c
__global__ void transposeUnroll4Row(float __out, float __in, const int __nx, const int __ny) {
    unsigned int ix = blockDim.x * blockIdx.x * 4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy * __nx + ix; // access in rows
    unsigned int to = ix * __ny + iy; // access in columns
    if (ix + 3 * blockDim.x < __nx && iy < __ny) {
        out[to] = in[ti];
        out[to + ny * blockDim.x] = in[ti + blockDim.x];
        out[to + ny * 2 * blockDim.x] = in[ti + 2 * blockDim.x];
        out[to + ny * 3 * blockDim.x] = in[ti + 3 * blockDim.x];
    }
}

__global__ void transposeUnroll4Col(float __out, float __in, const int __nx, const int __ny) {
    unsigned int ix = blockDim.x * blockIdx.x * 4 + threadIdx.x;
    unsigned int iy = blockDim.y * blockIdx.y + threadIdx.y;
    unsigned int ti = iy * __nx + ix; // access in rows
    unsigned int to = ix * __ny + iy; // access in columns
    if (ix + 3 * blockDim.x < __nx && iy < __ny) {
        out[ti] = in[to];
        out[ti + blockDim.x * ny] = in[to + blockDim.x * ny];
        out[ti + 2 * blockDim.x * ny] = in[to + 2 * blockDim.x * ny];
        out[ti + 3 * blockDim.x * ny] = in[to + 3 * blockDim.x * ny];
    }
}
```
Example: Concurrent Processing

Concurrent handle *data transfer* and *computation*

For NVIDIA GT 650M (laptop GPU), there is one copy engine.

For NVIDIA Tesla K40 (high-end GPU), there are two copy engines.

The latency in data transfer could be hidden during computing.

To handle two tasks, which both are matrix multiplications.

Copy two inputs to GPU, copy one output from GPU
Reference

Professional CUDA C Programming


source code are available on the above website
GPU Programming with Python + CUDA on AWS EC2

Lev E. Givon

Bionet Group
Department of Electrical Engineering
Columbia University

April 23, 2015
Outline

1. Setting Up a GPU Instance on EC2
2. Setting up Python and PyCUDA
3. GPU Programming with Python
Creating an EC2 Instance

1. Note: EC2 GPU instances are not free (on-demand hourly fee is currently $0.65/hour while the instance is running).
2. Create account on https://aws.amazon.com, sign in, and go to the AWS EC2 console (https://console.aws.amazon.com/ec2)
3. Select **Launch Instance** and select **Ubuntu Server 14.04 LTS (HVM), SSD Volume Type**.
4. Choose the **g2.2xlarge** instance type.
5. Add storage - choose something like 20 Gb.
6. Configure a security group; if you have a fixed IP address, restrict access to that address.
7. Click **Review and Launch** and generate/download an SSH key to access the instance; you need this to login to the running image.
Accessing the Instance

1. Wait until the instance is listed as running on the EC2 console.

2. Find the instance’s public IP address; if the SSH key you generated is saved as gpu.pem, login to the instance from Linux/MacOSX as follows:

   ```
   ssh -i gpu.pem ubuntu@ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com
   ```

3. Before you install CUDA, you need to update the image’s kernel and restart it, otherwise the GPU driver will not work:

   ```
   sudo -s
   apt-get update
   apt-get install linux-generic
   reboot
   ```
Installing CUDA

1. Download/install CUDA 7.0 and some other useful packages:

   ```
   wget http://developer.download.nvidia.com/compute/cuda/repos/\ubuntu1404/x86_64/cuda-repo-ubuntu1404_7.0-28_amd64.deb
   sudo -s
dpkg -i cuda-repo-ubuntu1404_7.0-28_amd64.deb
apt-get update
apt-get install cuda-7-0 build-essential
   ```

2. Make sure `/usr/local/cuda/bin` is in your PATH:

   ```
   echo $PATH
   ```

   If not, add the following line to your `~/.bashrc` file, logout, and login:

   ```
   export PATH=/usr/local/cuda/bin:$PATH
   ```
Sample code from the CUDA SDK is installed in /usr/local/cuda/samples. Try to build one of the simple examples as follows:

1. cd ~/
2. cp -rp /usr/local/cuda/samples .
3. cd samples/1.Utilities/deviceQuery
4. make

The compilation should complete without any errors. Run the compiled program in the above directory as ./deviceQuery; it should print information about the GPU.
Outline

1. Setting Up a GPU Instance on EC2
2. Setting up Python and PyCUDA
3. GPU Programming with Python
Local Python Environment Management

1. Ubuntu ships many Python packages that can be installed directly using its package manager.

2. Installing newer versions (and their dependencies) manually can be a hassle.

3. One solution (others exist): conda (http://conda.pydata.org/)

4. Enables local (non-root) installation/management of prebuilt Python packages and their dependencies.

5. Can create multiple independent Python environments - useful for trying things out without breaking one’s environment.
Installing/Using Miniconda

1. Miniconda ([http://conda.pydata.org/miniconda.html](http://conda.pydata.org/miniconda.html)) is a minimal combination of conda and Python.

2. Download and install as follows; let the installer use miniconda as the installation directory and let it add miniconda/bin to your PATH:

   ```
   cd ~/
   wget http://repo.continuum.io/miniconda/\Miniconda-latest-Linux-x86_64.sh
   ./Miniconda-latest-Linux-x86_64.sh
   ```

3. Logout, log back in, and try installing some Python packages:

   ```
   conda install ipython
   ```
Non-Conda Packages

1. If a package hasn’t been built for conda, you can try installing it from the Python Package Index (http://pypi.python.org):
   
   ```
   conda install pip
   pip install some_interesting_package
   ```

2. You can examine both your conda and pip packages as follows:

   ```
   conda list
   ```

3. Other useful commands available - see conda help and the package website.
Installing PyCUDA

1. Download the latest PyCUDA tarball from PyPI (don’t try to install it using pip), unpack, configure it to look for CUDA in the right location, build, and install:

```
wget https://pypi.python.org/packages/source/p/
pycuda/pycuda-2014.1.tar.gz
tar zxf pycuda-2014.1.tar.gz
cd pycuda-2014.1
./configure.py --cuda-root=/usr/local/cuda
python setup.py install
```

2. Try running one of the included examples:

```
cd ~/pycuda-2014.1/examples
python dump_properties.py
```

This should print information about the instance’s GPU.
IPython Notebook

1. Don’t like working at the command line? Try the IPython Notebook (http://ipython.org/notebook.html). Install the following packages in your instance and then start the notebook server. Make note of the port used by the server:

   - conda install ipython pyzmq jinja2 tornado mistune
   - jsonschema pygments terminado
   - ipython notebook --no-browser

2. Create an ssh tunnel to your EC2 instance that forwards to the above port (e.g., 8888):

   - ssh -i gpu.pem -L 8888:localhost:8888
   - ubuntu@ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com

Simple spectral analysis

An illustration of the Discrete Fourier Transform

\[ X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i k n}{N}} \quad k = 0, \ldots, N - 1 \]

using windowing, to reveal the frequency content of a sound signal.

We begin by loading a dataset using SciPy’s audio file support:

```
In [1]: from scipy.io import wavfile
rate, x = wavfile.read('test_mono.wav')
```

And we can easily view its spectral structure using matplotlib’s built-in `specgram` routine:

```
In [2]: fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
   ...
   ...
ax1.plot(x); ax1.set_title('Raw audio signal')
ax2.specgram(x); ax2.set_title('Spectrogram');
```
Outline

1. Setting Up a GPU Instance on EC2
2. Setting up Python and PyCUDA
3. GPU Programming with Python
Many problem scenario parameters to address: different types, array dimensions, etc.

Many possible hardware scenarios: number of threads, blocks, compute capability, etc.

Python reduces (but does not eliminate) the need to think about computer architecture and hardware. Can it do the same for GPUs?
Solution
Runtime Code Generation

- With PyCUDA, code does *not* need to be fixed at compile time.
- Kernels may be constructed and tuned as *Python strings* before being launched.
- Classes for facilitating construction of certain types of kernels included.
ndarray - Multidimensional Arrays in Python

- Unlike MATLAB, Python contains no native vector data type.
- `numpy.ndarray`: can be used to define vectors, matrices, tensors, etc.
- Binds multidimensional data with information about `dtype`, `shape`, and `strides`.
- Supports operation broadcasting, e.g., `A+B`, `\sin(A)`, `A**2`
- Serves as basis for other scientific computing packages: scipy, matplotlib, etc.
GPUArray - Multidimensional Arrays in GPU Memory

- `pycuda.gpuarray.GPUArray` - ndarray-like class for managing GPU memory.
- Array info resides in PC memory, data in GPU memory.
- Similar attributes to `ndarray`: `dtype`, `shape`, `strides`
- Compatible with `ndarray`:

```python
import pycuda.gpuarray as gpuarray
x_gpu = gpuarray.to_gpu(numpy.random.rand(3))
y = x_gpu.get()
```

- `print x_gpu` works automatically.
- Implicit generation of kernels for vectorized (elementwise) operations, e.g., `x_gpu+y_gpu`.
Example

```python
import atexit
import numpy as np
import pycuda.driver as drv
import pycuda.gpudarray as gpuarray

drv.init()
dev = drv.Device(0)  # initialize GPU 0
ctx = dev.make_context()
atexit.register(ctx.pop)  # clean up on exit

x = np.random.rand(2, 3).astype(np.double)
x_gpu = gpudarray.to_gpu(x)
```
Single-Pass Expressions

- Implicitly generated kernels are cached to improve performance. However..
- Elementwise expressions involving GPUArray instances (e.g., \(x+y*z\)) compile/launch new kernels for each intermediate step.
- To improve efficiency, PyCUDA enables construction of complex single-pass elementwise expressions computed using a single kernel.
- Classes also provided that facilitate construction of other types of kernels:
  - Reductions (e.g., sum, product, dot product)
  - Scans (e.g., cumulative sum)
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.elementwise import ElementwiseKernel
from pycuda.curandom import import rand

x_gpu = rand(10, np.double); y_gpu = rand(10, np.double)
z_gpu = gpuarray.empty_like(x_gpu)
func = ElementwiseKernel("double x*, double y*, double z*",
                        "z[i] = 2*x[i]+3*y[i]"
                        )
func(x_gpu, y_gpu, z_gpu)
print 'Success: ', np.allclose(2*x_gpu.get()+3*y_gpu.get(),
                              z_gpu.get())
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.reduction import ReductionKernel
from pycuda.curandom import rand

x_gpu = rand(10, np.double); y_gpu = rand(10, np.double)
func = ReductionKernel(dtype_out=np.double,
    neutral="0",
    reduce_expr="a+b",
    map_expr="x[i]*y[i]",
    arguments="double *x, double *y")
result = func(x_gpu, y_gpu).get()
print 'Success: ', np.allclose(np.dot(x_gpu.get(), y_gpu.get()),
    result)
```python
import numpy as np
import pycuda.autoinit
import pycuda.gpudarray as gpudarray
from pycuda.scan import InclusiveScanKernel
from pycuda.curandom import rand

x_gpu = rand(10, np.double); x = x_gpu.get()
func = InclusiveScanKernel(np.double, "a+b")
result = func(x_gpu).get()
print 'Success: ', np.allclose(np.cumsum(x), result)
```
Creating Your Own Kernels

```python
import numpy as np
import pycuda.autoinit
import pycuda.gpudarray as gpudarray
from pycuda.compiler import.SourceModule
from pycuda.curandom import rand

x_gpu = rand(10, np.double); x = x_gpu.get()
mod = SourceModule('"
__global__ void func(double *x) {
    int idx = threadIdx.x;
    x[idx] *= 3;
}
"
')
func = mod.get_function('func')
func(x_gpu, block=(10, 1, 1))
print 'Success: ', np.allclose(3*x, x_gpu.get())
```
Using GPU-based Libraries

- Optimizing common algorithms for GPUs can be nontrivial - why reinvent the wheel?
- Increasing number of mathematical libraries available for GPUs: linear systems (CUBLAS, CUSOLVER), signal processing (CUFFT, CULA), sparse data (CUSPARSE) etc.
- Most of these libraries only have C/C++ interfaces, however.
- Can we use them from Python?
- Solution: CUDA SciKit
  (http://scikit-cuda.readthedocs.org)
- Provides both low level (C-like) and high level (numpy-like) interfaces to libraries.
import pycuda.gpuarray as gpuarray
import pycuda.autoinit
import numpy as np
import scikits.cuda.linalg as linalg

linalg.init()
a = np.random.randn(9, 6) + 1j*np.random.randn(9, 6)
a = np.asarray(a, np.complex64)
a_gpu = gpuarray.to_gpu(a)
u_gpu, s_gpu, vh_gpu = linalg.svd(a_gpu, 'S', 'S')
print 'Success: ', np.allclose(a, np.dot(u_gpu.get(),
np.dot(np.diag(s_gpu.get()), vh_gpu.get())), 1e-4)
PyCUDA Resources

- http://mathema.tician.de/software/pycuda
- http://lists.tiker.net/listinfo/pycuda
- http://wiki.tiker.net/PyCuda
- http://scikit-cuda.readthedocs.org
GPU on iOS devices
PowerVR GPU has been used.
A4 => SGX 535 (1.6 GFLOPS)
A5 => SGX 543 MP2 (12.8 GLOPS)
A6 => SGX 543 MP3 (25.5 GFLOPS)
A7 => G6430 (quad core)
(230.4 GFLOPS)
A8 => GX6450 (quad core)
(332.8 GLOPS)

FLOPS = sockets × cores/socket × clock × FLOPs/cycle

Most microprocessors today can carry out 4 FLOPs per clock cycle; thus a single-core 2.5 GHz processor has a theoretical performance of 10 billion FLOPS = 10 GFLOPS.
GPU in Apple A8 SoC

A8 — iPhone 6 and iPhone 6 Plus

GPU: PowerVR Quad-core GX6450

4 Unified Shading Cluster (USC)

# of ALUs: 32 (FP32) or 64 (FP16) per USC

GFLOPS: 166.4 (FP32)/ 332.8 (FP16) @ 650 MHz

Supports OpenCL 1.2

source: http://www.imgtec.com/powervr/sixxt.asp
GPU Programming in iPhone/iPad - Metal

*Metal* provides the lowest-overhead access to the GPU, enabling developers to maximize the graphics and compute potential of *iOS 8 app.*

Metal could be used for:

- Graphic processing ➔ openGL
- General data-parallel processing ➔ open CL and CUDA

Fundamental Metal Concepts

- Low-overhead interface
- Memory and resource management
- Integrated support for both graphics and compute operations
- Precompiled shaders
GPU Programming in iPhone/iPad - Metal

Programming flow is similar to CUDA

- Copy data from CPU to GPU
- Computing in GPU
- Send data back from GPU to CPU

Example: kernel code in Metal, sigmoid function:

```
kernel void sigmoid(const device float *inVector [(buffer(0))],
                    device float *outVector [(buffer(1))],
                    uint id [(thread_position_in_grid)]) {

  // This calculates sigmoid for _one_ position (=id) in a vector per call on the GPU
  outVector[id] = 1.0 / (1.0 + exp(-inVector[id]));
}
```

source: [http://memkite.com](http://memkite.com)
Metal Object Relationships

Diagram showing the relationships between different components in a system, including Buffers, Textures, Sampler States, Depth Stencil States, Render Pipeline States, Render Command Encoders, Blit Command Encoders, Command Buffers, and a Command Queue.
Metal Command Buffers with Multiple Threads