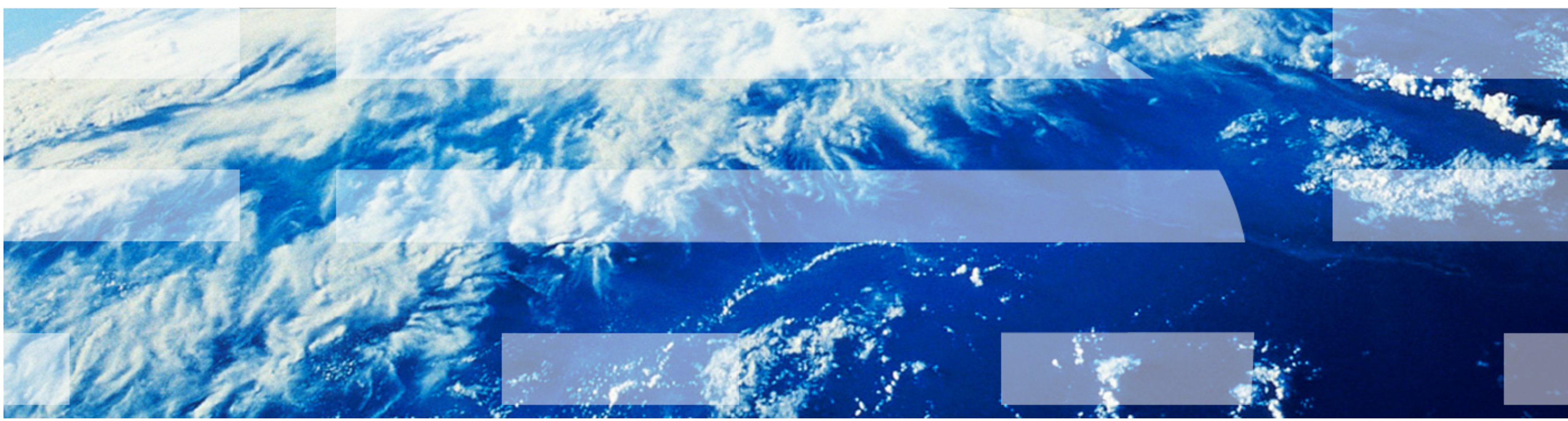
E6895 Advanced Big Data Analytics Lecture 8:

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



Demo of GPU Computing on MacBook and iPhone

Speaker: Richard Chen

CUDA on Mac OS X

NVIDIA CUDA TOOLKIT DOCUMENTATION

CUDA Toolkit v7.0

Getting Started Mac OS X

Getting Started Mac OS X (PDF) - v7.0

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. Seria
  portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existin
  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 th
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 developme

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)

GPU Architecture

GPU Architecture

built by several streaming multiprocessors (SMs)

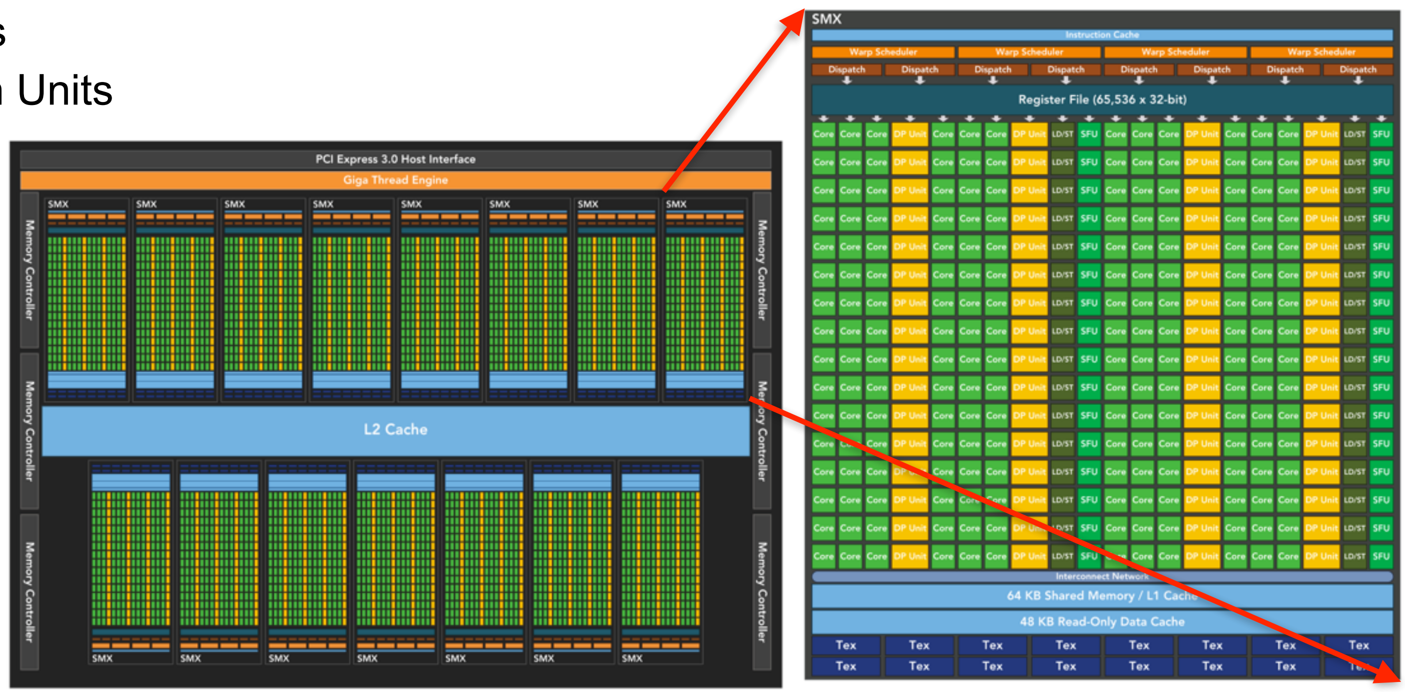
In each SM:

- CUDA cores
- Shared Memory/L1 Cache
- Register File
- Load/Store Units
- Special Function Units
- Warp Scheduler

In each device:

- L2 Cache

Kepler Architecture, K20X



Example: deviceQuery

Understand the hardware constraint via deviceQuery (in example code of CUDA toolkit)

```
Device 0: "GeForce GT 650M"
  CUDA Driver Version / Runtime Version          7.0 / 7.0
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 1024 MBytes (1073414144 bytes)
  ( 2) Multiprocessors, (192) CUDA Cores/MP:     384 CUDA Cores
  GPU Max Clock rate:                            900 MHz (0.90 GHz)
  Memory Clock rate:                             2508 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 262144 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
```

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

```
void sumArraysOnHost(float *A, float *B, float *C, const int N)
{
    for (int idx = 0; idx < N; idx++)
    {
        C[idx] = A[idx] + B[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

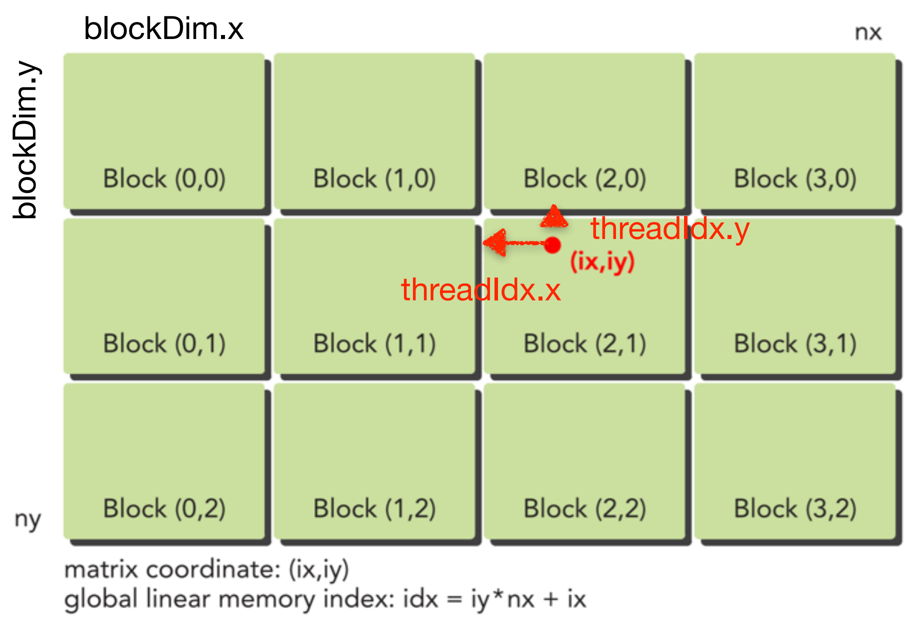
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx +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

```
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());
```

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

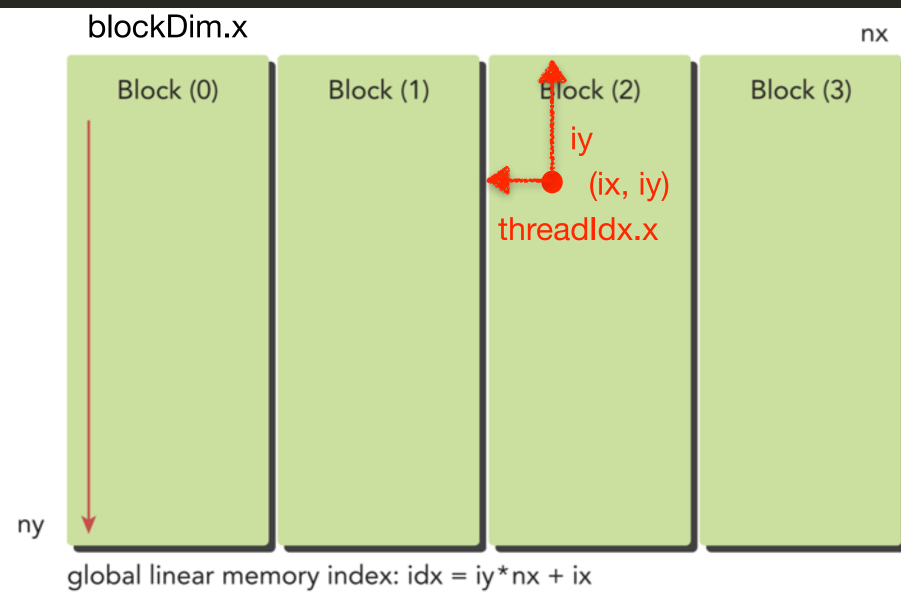
```
// grid 2D block 2D
__global__ void sumMatrixOnGPU2D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = threadIdx.y + blockIdx.y * blockDim.y;
    unsigned int idx = iy * nx +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



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)

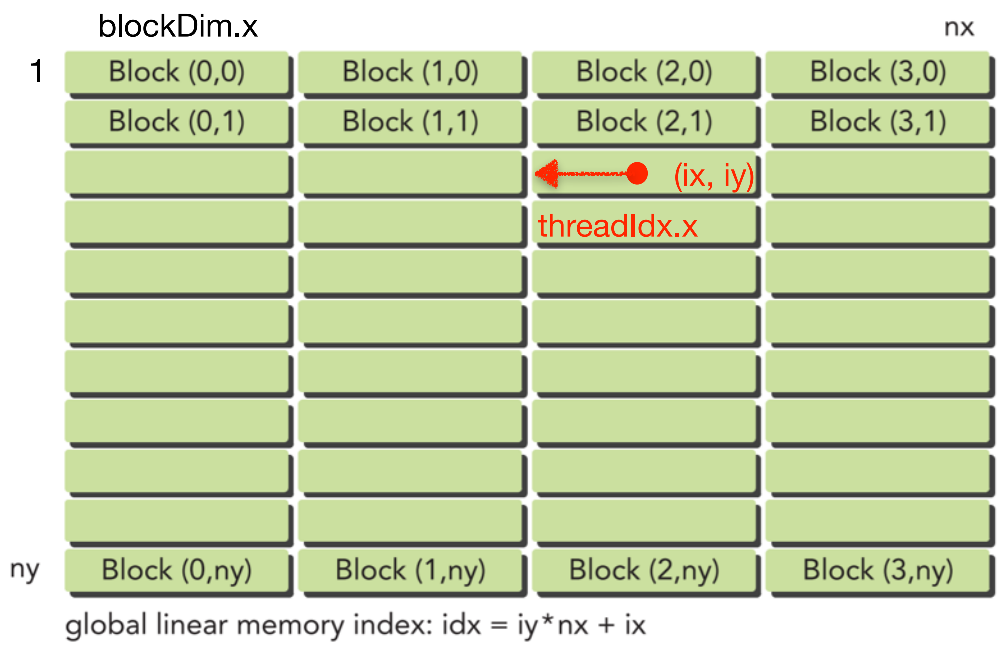
```
// grid 1D block 1D
__global__ void sumMatrixOnGPU1D(float *MatA, float *MatB, float *MatC, int nx,
                                 int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    if (ix < nx )
        for (int iy = 0; iy < ny; iy++) {
            int idx = iy * nx + ix;
            MatC[idx] = MatA[idx] + MatB[idx];
        }
}
```



Example: Matrix Addition on GPU - 2D grid with 1D blocks

Data accessing in 2D grid with 1D blocks arrangement (one green block is one thread block)

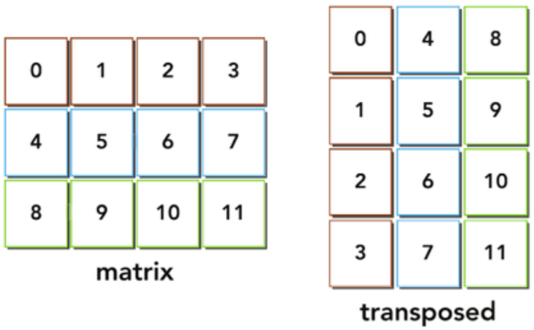
```
// grid 2D block 1D
__global__ void sumMatrixOnGPUMix(float *MatA, float *MatB, float *MatC, int nx,
                                  int ny) {
    unsigned int ix = threadIdx.x + blockIdx.x * blockDim.x;
    unsigned int iy = blockIdx.y;
    unsigned int idx = iy * nx + ix;
    if (ix < nx && iy < ny)
        MatC[idx] = MatA[idx] + MatB[idx];
}
```



Example: Matrix Transpose on CPU

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

$$A_{mxn} = B_{nxm}$$



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

```
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

```
// 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

```
// case 4 transpose kernel: read in rows and write in columns + unroll 4 blocks
__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];
    }
}
```

```
// case 5 transpose kernel: read in columns and write in rows + unroll 4 blocks
__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] = in[to +   blockDim.x * ny];
        out[ti + 2 * blockDim.x] = in[to + 2 * blockDim.x * ny];
        out[ti + 3 * blockDim.x] = in[to + 3 * blockDim.x * ny];
    }
}
```

Example: Concurrent Processing

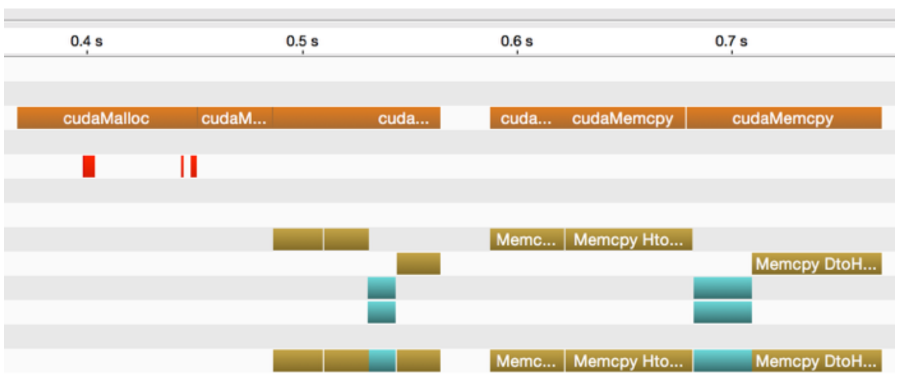
Concurrent handle ***data transfer*** and ***computation***

- For NVIDIA GT 650M (laptop GPU), there is one copy engine.
- For NVIDIA Tesla K40 (high-end GPU), there are two copy engines

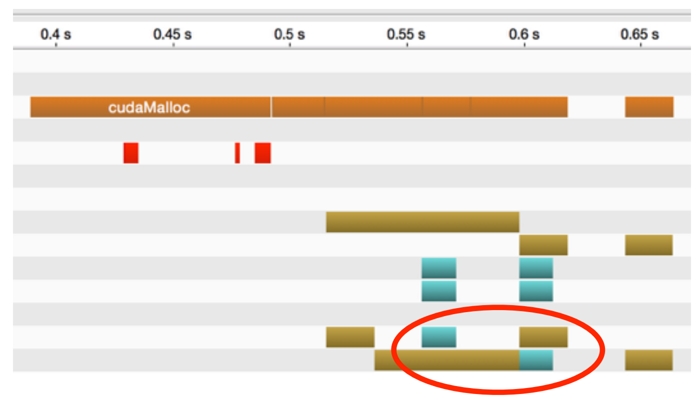
The latency in data transfer could be hidden during computing

To handle two tasks, which both are matrix multiplications.

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



No concurrent processing



Concurrent processing

GPU on iOS devices

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

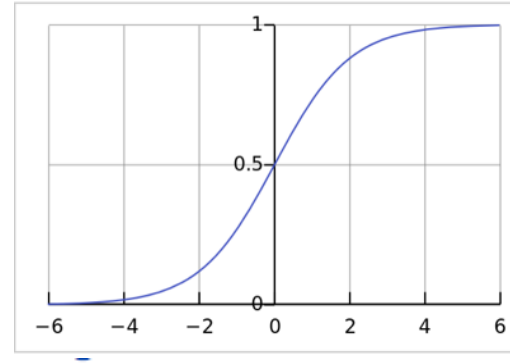
General data-parallel processing ➔ open CL and CUDA

*: https://developer.apple.com/metal/

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:

$$S(t) = \frac{1}{1 + e^{-t}}$$

kernel code

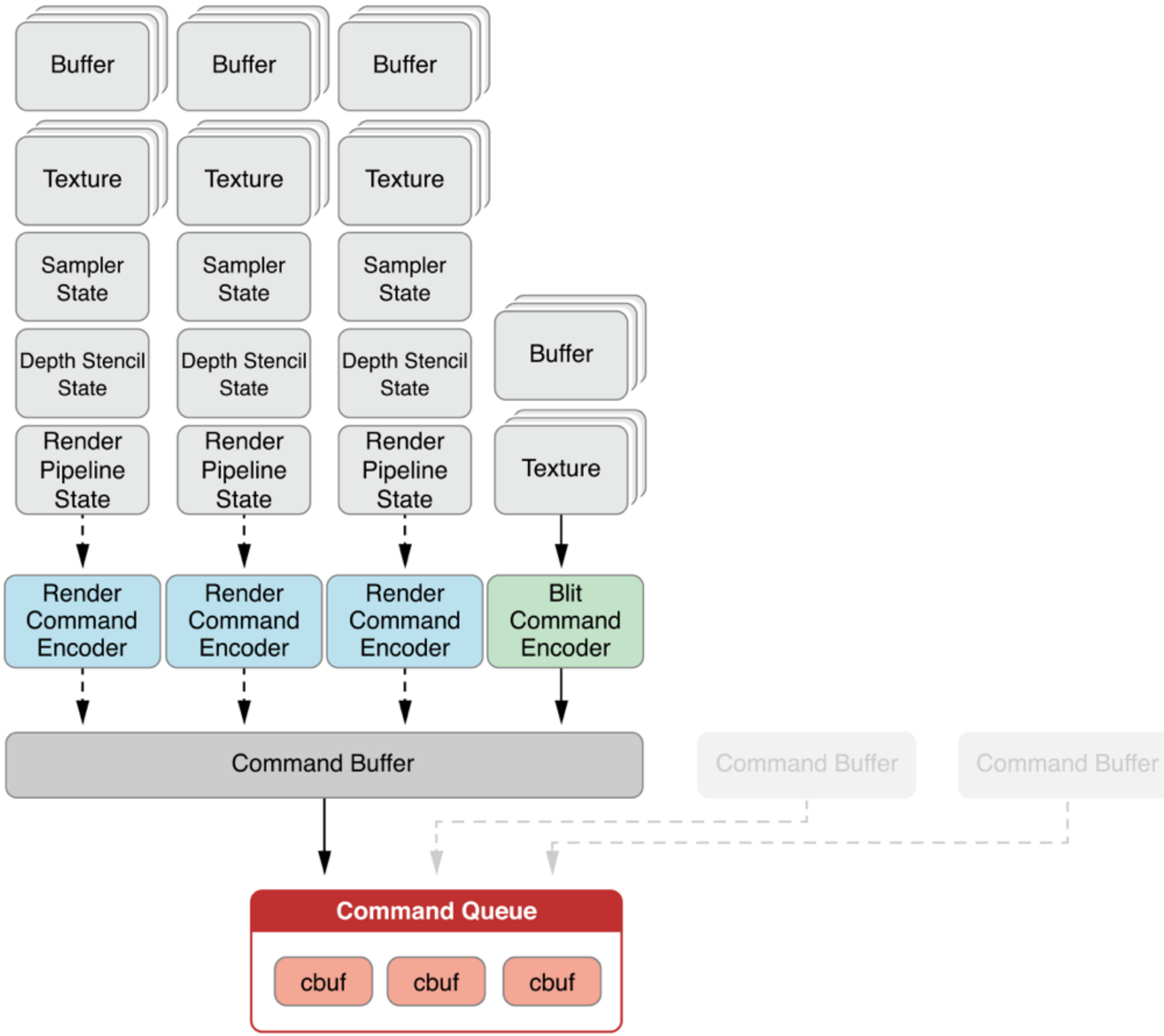
device memory

thread_id for data parallelization

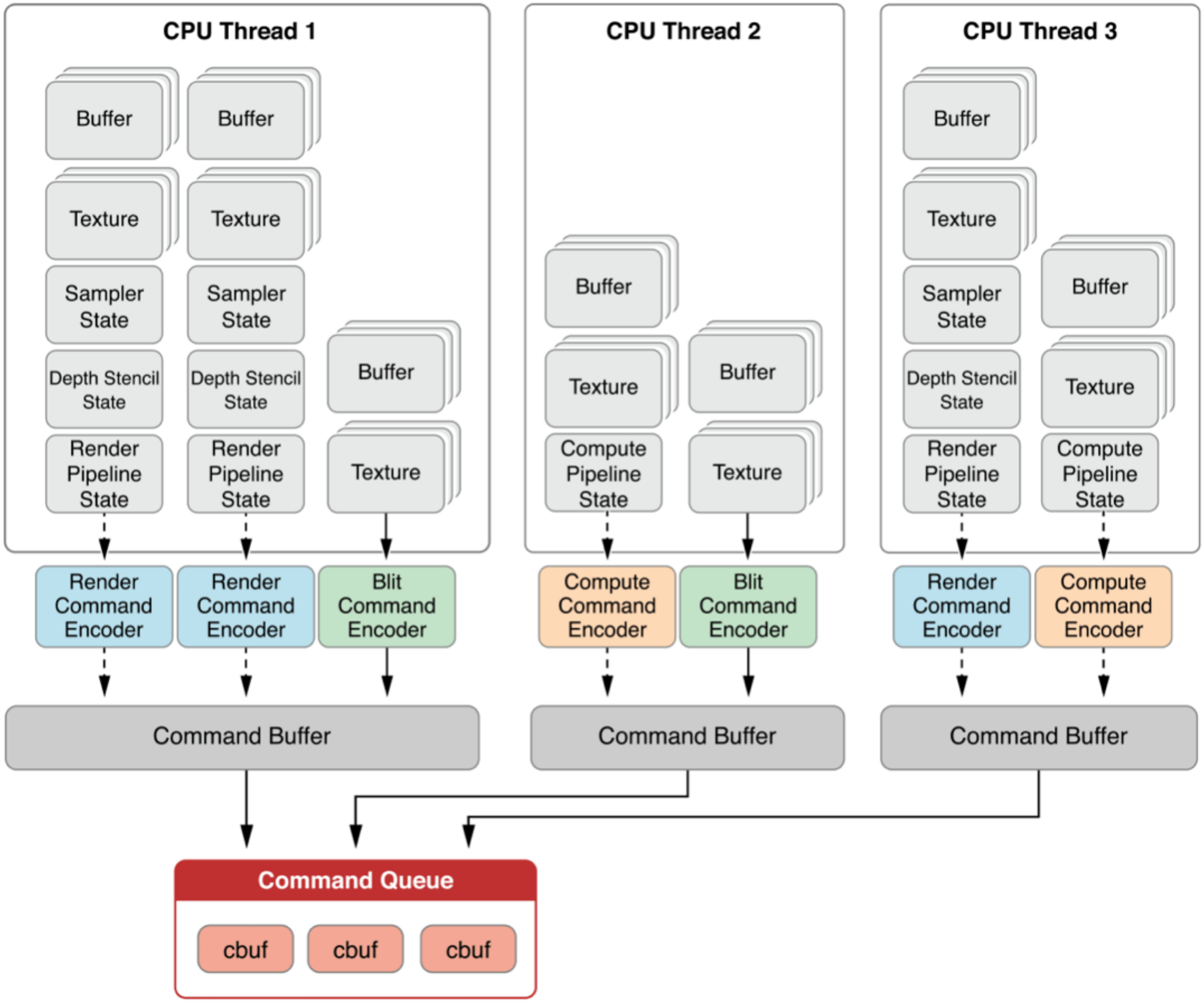
```
kernel void sigmoid(const device float *inVector [[ buffer(0) ]],
                    device float *outVector [[ buffer(1) ]],
                    uint id [[ thread_position_in_grid ]]) {
    // This calculates sigmoid for _one_ position (=id) in a vector per call on the
GPU
    outVector[id] = 1.0 / (1.0 + exp(-inVector[id]));
}
```

source: http://memkite.com

Metal Object Relationships



Metal Command Buffers with Multiple Threads



Metal Programming Model

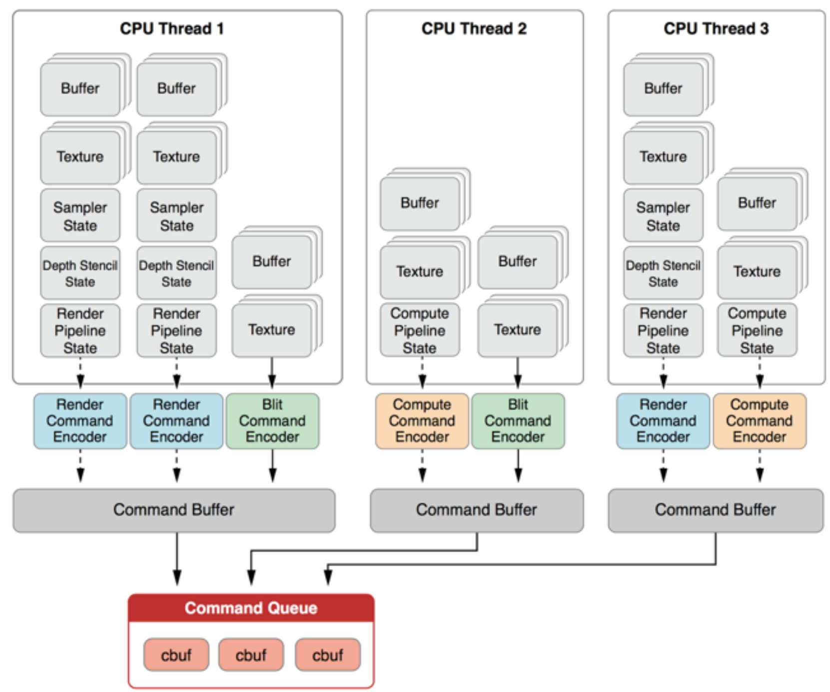
It integrates the support for both ***graphics*** and ***compute*** operations.

- Three command encoder:
  - Graphics Rendering: Render Command Encoder
  - Data-Parallel Compute Processing: Compute Command Encoder
  - Transfer Data between Resource: Blitting Command Encoder

Multi-threading in encoding command is supported

Typical flow in compute command encoder

- Prepare data
- Put your function into pipeline
- Command encoder
- Put command into command buffer
- Commit it to command queue
- Execute the command
- Get result back

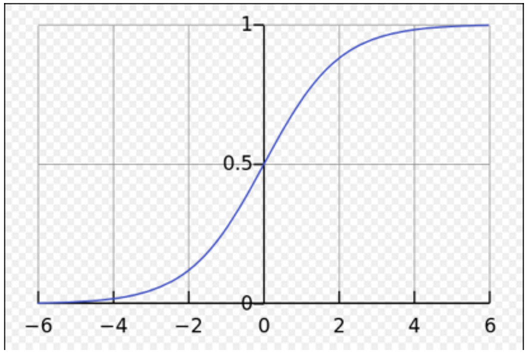


Metal Programming, Kernel Function

Compute command

- Two parameters, threadsPerGroup and numThreadgroups, determines number of threads. ➔ equivalent to grid and thread block in CUDA. They are all 3-D variable.
- The total of all threadgroup memory allocations must not exceed 16 KB.

- Kernel function: 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]));
}
```

Example

```
func initMetal() -> (MTLDevice, MTLCommandQueue, MTLLibrary, MTLCommandBuffer,
    MTLComputeCommandEncoder){
        // Get access to iPhone or iPad GPU
        var device = MTLCreateSystemDefaultDevice()

        // Queue to handle an ordered list of command buffers
        var commandQueue = device.newCommandQueue()

        // Access to Metal functions that are stored in Shaders.metal file, e.g. sigmoid()
        var defaultLibrary = device.newDefaultLibrary()

        // Buffer for storing encoded commands that are sent to GPU
        var commandBuffer = commandQueue.commandBuffer()

        // Encoder for GPU commands
        var computeCommandEncoder = commandBuffer.computeCommandEncoder()

        return (device, commandQueue, defaultLibrary!, commandBuffer, computeCommandEncoder)
}
```

```
// set up a compute pipeline with Sigmoid function and add it to encoder
let sigmoidProgram = defaultLibrary.newFunctionWithName("sigmoid")
var pipelineErrors = NSErrorPointer()
var computePipelineFilter = device.newComputePipelineStateWithFunction(sigmoidProgram!, error: pipelineErrors)
computeCommandEncoder.setComputePipelineState(computePipelineFilter!)
```

```
//   set the input vector for the Sigmoid() function, e.g. inVector
//    atIndex: 0 here corresponds to buffer(0) in the Sigmoid function
var inVectorBufferNoCopy = device.newBufferWithBytesNoCopy(memory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(inVectorBufferNoCopy, offset: 0, atIndex: 0)

// d. create the output vector for the Sigmoid() function, e.g. outVector
//    atIndex: 1 here corresponds to buffer(1) in the Sigmoid function
var outVectorBufferNoCopy = device.newBufferWithBytesNoCopy(outmemory, length: Int(size), options: nil, deallocator: nil)
computeCommandEncoder.setBuffer(outVectorBufferNoCopy, offset: 0, atIndex: 1)
```

Example

```
// hardcoded to 32 for now (recommendation: read about threadExecutionWidth)
var threadsPerGroup = MTLSize(width:32,height:1,depth:1)
var numThreadgroups = MTLSize(width:(Int(maxcount)+31)/32, height:1, depth:1)
computeCommandEncoder.dispatchThreadgroups(numThreadgroups, threadsPerThreadgroup: threadsPerGroup)
computeCommandEncoder.endEncoding()
```

```
commandBuffer.commit()
commandBuffer.waitUntilCompleted()
```

Speaker: Eric Johnson

GPU Programming with Python + CUDA on AWS EC2

Lev E. Givon

Bionet Group
Department of Electrical Engineering
Columbia University

amazon webservices EC2 python NVIDIA CUDA

Lev E. Givon GPU Programming with Python + CUDA on AWS EC2

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
```

Checking that Things Work

1. Sample code from the CUDA SDK is installed in /usr/local/cuda/samples. Try to build one of the simple examples as follows:

```
cd ~/
cp -rp /usr/local/cuda/samples .
cd samples/1_Utilities/deviceQuery
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) 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:

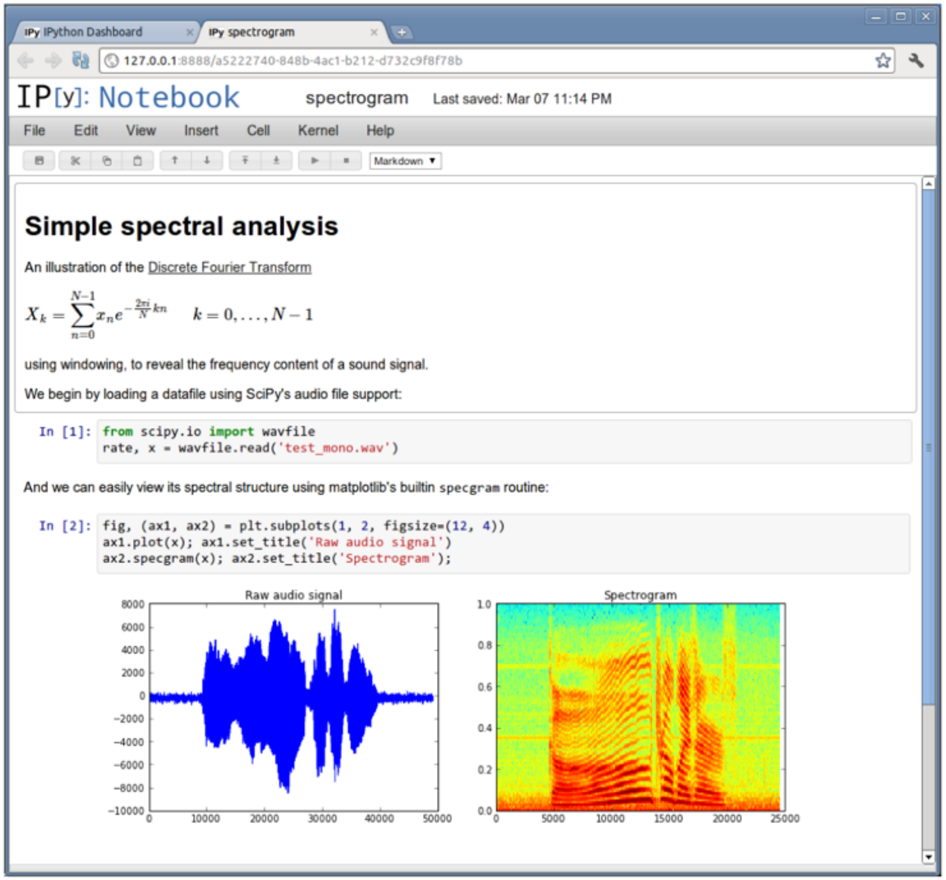
```
conda install ipython pyzmq jinja2 tornado mistune \
jsonschema pygments terminado
ipython notebook --no-browser
```

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
```

3. Open http://localhost:8888 in your web browser.

What it Looks Like



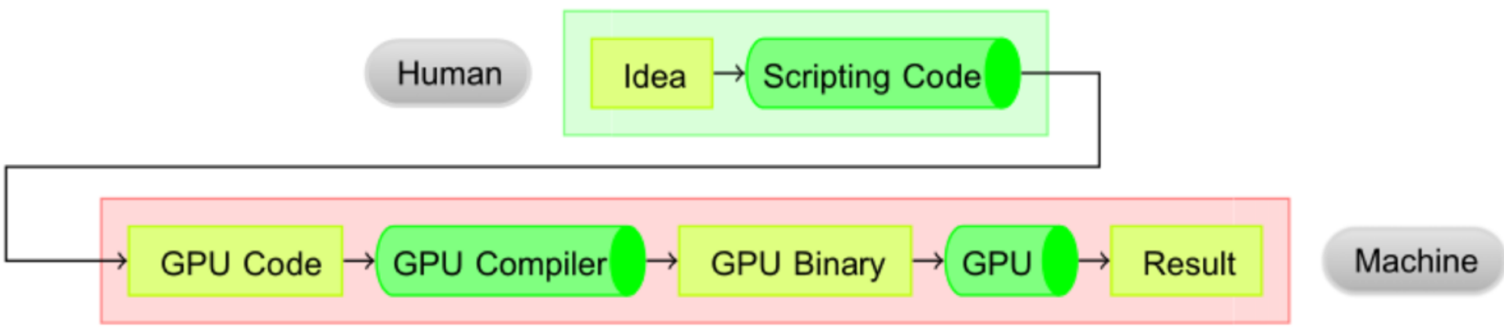
Outline

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

General Problem

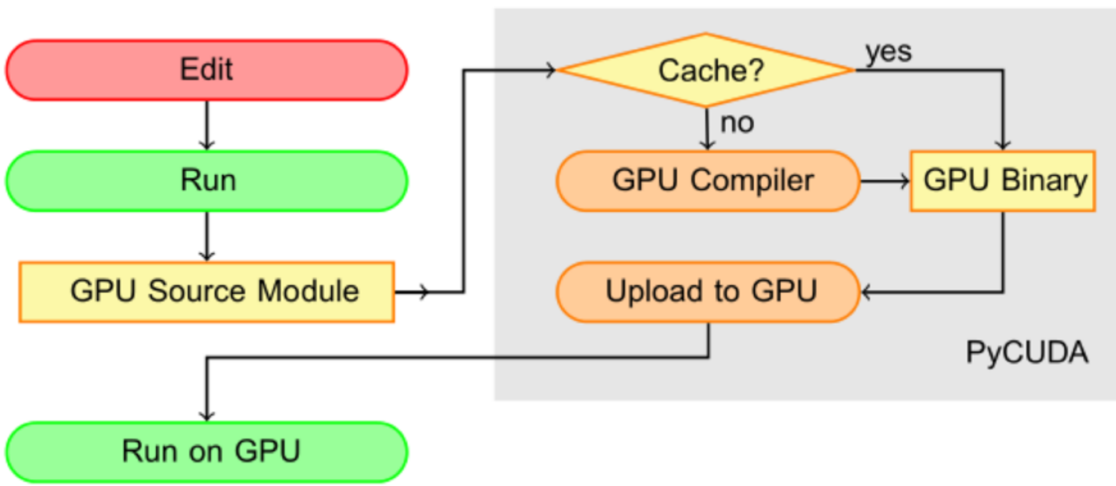
- Many problem scenario parameters to address: different types, array dimensions, etc.
- Many possible hardware scenarios: number of threads, blocks, compute capability, etc.
- Python reduces (but *does not* eliminate) the need to think about computer architecture and hardware. Can it do the same for GPUs?

Solution


Human
Idea
Scripting Code
GPU Code
GPU Compiler
GPU Binary
GPU
Result
Machine

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`:

```
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.gpuarray 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 = gpuarray.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)

Elementwise Operations

```
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
from pycuda.elementwise import ElementwiseKernel
from pycuda.curandom 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())
```

Reductions

```
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)
```

Scans

```
import numpy as np
import pycuda.autoinit
import pycuda.gpuarray as gpuarray
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.gpuarray as gpuarray
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.

CUDA SciKit Example

```
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