An introduction to CUDA using Python

Miguel Lázaro-Gredilla
miguel@tsc.uc3m.es

May 2013

Machine Learning Group
http://www.tsc.uc3m.es/~miguel/MLG/

Contents

Leveraging GPGPU

General-Purpose computing on the Graphics Processing Unit:

- GPUs have highly parallel architectures ($>$2000 cores)
- GPU cores are not independent, fully-featured CPUs
  - Flow-control operations: performance penalties
  - Maximum occupancy is not always satisfied
- Good for fast and cheap number crunching
- Very successful for neural network training and deep learning
- Heterogeneous programming can be tricky!
- Tooling is currently work in progress

APIs for GPGPU

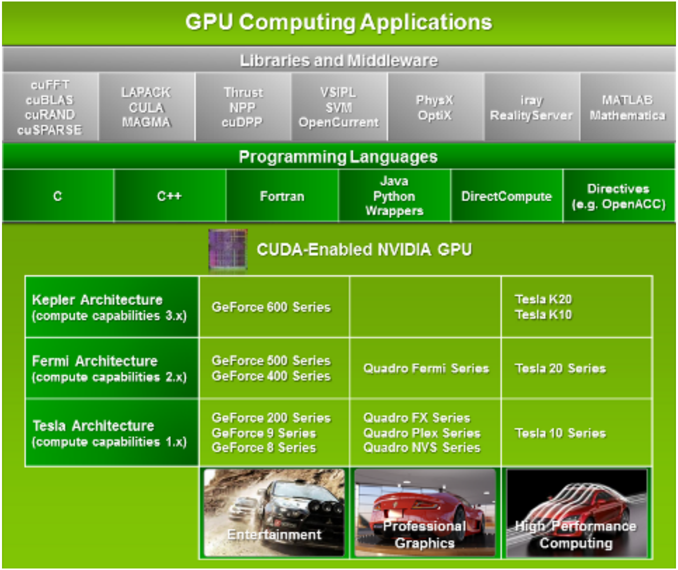
Open computing language (OpenCL)

- Many vendors: AMD, Nvidia, Apple, Intel, IBM...
  - Standard CPUs may report themselves as OpenCL capable
- Works on most devices, but
  - Implemented feature set and extensions may vary
  - For portability, only the common subset can be used...
  - ...so maximum performance can't be achieved

Compute unified device architecture (CUDA)

- One vendor: Nvidia (more mature tools)
- Better coherence across a limited set of devices

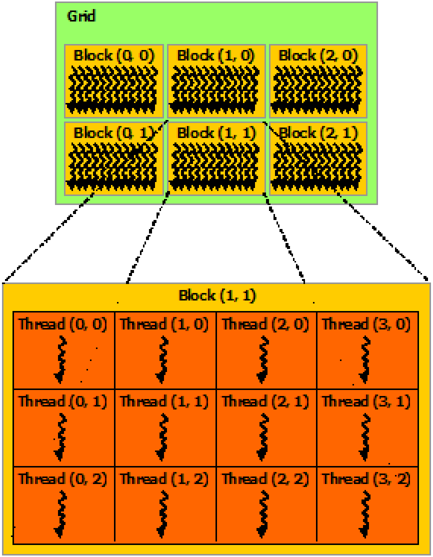
CUDA stack (source: Nvidia documentation)



Hardware concepts

- A grid is a 2D arrangement of independent blocks
  - of dimensions (gridDim.x $\times$ gridDim.y)
  - and with blocks at (blockIdx.x, blockIdx.y)
- A block is a 3D arrangement of threads
  - of dimensions (bloackDim.x $\times$ blockDim.y $\times$ blockDim.y)
  - and with threads at (threadIdx.x, threadIdx.y, threadIdx.z)
- Each thread is a unit of work and instances a *kernel*
  - Kernels are written in CUDA C (.cu, a variant of C/C++)
  - Each kernel is parametrized by its zero-based location
- The size of each arrangement is user configurable (within hardware constraints)

Process hierarchy (source: Nvidia documentation)



Compute capabilities

	Compute capability						
Specifications	1.0	1.1	1.2	1.3	2.x	3.0	3.5
64-bit in global memory	No		Yes				
Max dim of grid	2				3		
Max gridDim.x	65535					$2^{31}-1$	
Max gridDim.y/z	65535						
Max dim of block	3						
Max blockDim.x/y	512				1024		
Max gridDim.z	64						
Max threads/block	512				1024		
Warp size	32						
Max blocks/MP	8					16	
Max threads/MP	768		1024		1536	2048	

A mid-2009 macboook pro

macbookm:release miguel$ ./deviceQuery

Found 1 CUDA Capable device(s)

Device 0:	"GeForce 9400M"
CUDA Driver Version / Runtime Version	4.1 / 4.1
CUDA Capability Major/Minor version number:	1.1
Total amount of global memory:	254 MBytes
(2) Multiprocessors x (8) CUDA Cores/MP:	16 CUDA Cores
GPU Clock Speed:	1.10 GHz
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
(. . .)	

Python support for CUDA

PyCUDA

- You still have to write your kernel in CUDA C
- . . . but integrates easily with numpy
- Higher level than CUDA C, but not much higher
- Full CUDA support and performance

gnumpy/CUDAMat/cuBLAS

- gnumpy: numpy-like wrapper for CUDAMat
- CUDAMat: Pre-written kernels and partial cuBLAS wrapper
- cuBLAS: (incomplete) CUDA implementation of BLAS

Contents

Exercise 1.A

Generate 10^7 random draws from a $\mathcal{N}(0,1)$ density and count how many of them lie between -1 and +1. Time it.

Disable User Module Deleter (UMD) if using Spyder.

```
import numpy as np
import time as t

x = np.random.randn(10e6).astype(np.float32)
start = t.time()
valid = np.logical_and(-1 < x, x < +1)
print 'CPU: Found %d values in %f secs' % (np.sum(valid), t.time()-start)
```

Exercise 1.B

Repeat 1.A using PyCUDA.

You can start from this illustration of PyCUDA usage.

```
import numpy as np
import pycuda.autoinit
from pycuda.compiler import SourceModule
import pycuda.driver as drv
import pycuda.gpuarray as gpuarray

kernel = SourceModule("""
__global__ void twice(float *x)
{
    const unsigned int i = threadIdx.x + threadIdx.y*blockDim.x;
    x[i] = 2*x[i];
}
""")

twice = kernel.get_function('twice')
x = np.random.randn(16).astype(np.float32)
x_gpu = gpuarray.to_gpu(x)
twice(x_gpu, block=(4, 4, 1), grid=(1,1))

print x, np.sum(x)
print x_gpu.get(), np.float32(gpuarray.sum(x_gpu).get())
```

Exercise 1.B

A useful kernel might look like this:

```
kernel = SourceModule("""
__global__ void threshold(float *x, unsigned int len)
{
    const unsigned int idx = blockIdx.x * blockDim.x + threadIdx.x;
    const unsigned int numThreads = blockDim.x * gridDim.x;
    for(int i = idx; i < len; i+=numThreads)
        x[i] = -1<x[i] && x[i]<+1? 1.0 : 0.0;
}
""")
threshold = kernel.get_function('threshold')
```

And the corresponding call would look like this:

```
start = t.time()
threshold(x_gpu, np.uint32(len(x)), block=(256, 1, 1), grid=(16,1))
print 'GPU: Found %d values in %f secs'% (gpuarray.sum(x_gpu).get(),\
                                          t.time()-start)
```

Exercise 1.B

If you're going to invoke the same kernel many times, it's faster to "prepare" the call:

```
x_gpu = gpuarray.to_gpu(x)
threshold.prepare('Pi')
start = t.time()
threshold.prepared_call((16,1),(256,1,1),x_gpu.gpudata, np.uint32(len(x)))
print 'GPU: Found %d values in %f secs (prepared call)'%\
           (gpuarray.sum(x_gpu).get(), t.time()-start)
```

Exercise 1.B

Copying from the host to the device and back can be handled automatically, but it is slower:

```
start = t.time()
threshold(drv.InOut(x), np.uint32(len(x)), block=(256 ,1, 1), grid=(16,1))
print 'GPU: Found %d values in %f secs (automatic conversion)'%\
          (np.sum(x), t.time()-start)
```

I get this output for 1.A and the three 1.B variants:

```
CPU: Found 6828501 values in 0.102769 secs
GPU: Found 6828501 values in 0.020188 secs
GPU: Found 6828501 values in 0.020282 secs (prepared call)
GPU: Found 6828501 values in 0.166021 secs (automatic conversion)
```

Contents

Exercise 1.C

Repeat 1.A using gnumpy and generating only 2×10^6 values.

- gnumpy mimicks numpy. Import it as if it was numpy.
- There is no "logical_and" function AFAIK.
- There exists the "all" function, and its specialization for boolean inputs, "all2".
- If you just used PyCUDA, better restart IPython's kernel!

```
import gnumpy as g

x_gpu = g.garray(x).reshape(-1,1)
start = t.time()
x_gpu = g.concatenate(( -1 < x_gpu, x_gpu < +1),1)
print 'GPU: Found %d values in %f secs' %\
                (x_gpu.all2(1).sum(), t.time()-start)
```

Exercise 2.A

Generate a 2000×2000 random matrix by sampling i.i.d. from $x = \exp(t)$, where $t \sim \mathcal{N}(0,1)$. Use numpy to do this. Time it.

```
start = t.time()
x = np.random.randn(2000,2000)
x = np.exp(x)
print 'CPU: Generated %d numbers in %f secs' %\
                (np.prod(np.shape(x)), t.time()-start)
```

Exercise 2.B

Repeat 2.A, this time using gnumpy. Time it.

```
start = t.time()
x_gpu = g.randn(2000,2000)
x_gpu = g.exp(x_gpu)
print 'GPU: Generated %d numbers in %f secs' %\
                (np.prod(np.shape(x_gpu)), t.time()-start)
```

Exercise 3.A

Generate a 2000×2000 random matrix by sampling i.i.d. from $x \sim \mathcal{N}(0,1)$. Square it. Then sum all of its values. Use numpy to do this. Time it.

```
x = x_gpu.asarray()
start = t.time()
print 'CPU: Matrix product, total sum is %f, computed in %f secs' %\
                (np.sum(np.dot(x,x)), t.time()-start)
```

Exercise 3.B

Repeat 3.A, this time using gnumpy. Time it.

```
x_gpu = g.randn(2000,2000)
start = t.time()
print 'GPU: Matrix product, total sum is %f, computed in %f secs' %\
                (g.sum(g.dot(x_gpu,x_gpu)), t.time()-start)
```

Exercise 4.A

Generate a 2000×2000 random matrix A by sampling i.i.d. from $x \sim \mathcal{N}(0, 1)$. Compute trace$(A^\top A)$. Use numpy to do this. Time it.

```
x = x_gpu.asarray()
start = t.time()
print 'CPU: Element-wise product, total sum is %f, computed in %f secs' %\
                (np.sum(x*x), t.time()-start)
```

Exercise 4.B

Repeat 4.A, this time using gnumpy. Time it.

```
x_gpu = g.randn(2000,2000)
start = t.time()
print 'GPU: Element-wise product, total sum is %f, computed in %f secs' %\
                (g.sum(x_gpu*x_gpu), t.time()-start)
```

Results using gnumpy

I get this output:

```
(1.A) CPU: Found 1367262 values in 0.018013 secs
(1.C) GPU: Found 1367262 values in 0.541749 secs (<= Much slower than 1.B, we're only using 2e6 values!)

(2.A) CPU: Generated 4000000 numbers in 0.299176 secs
(2.B) GPU: Generated 4000000 numbers in 0.031891 secs (<= GPU is much faster!)

(3.A) GPU: Matrix product, total sum is -81153.687500, computed in 1.705496 secs (<= GPU is slower!)
(3.B) CPU: Matrix product, total sum is -81153.645447, computed in 1.087240 secs

(4.A) GPU: Element-wise product, total sum is 4003010.00, computed in 0.024504 secs (<= GPU is faster!)
(4.B) CPU: Element-wise product, total sum is 4003008.73, computed in 0.048110 secs
```

Note that the used GPU is one of the less capable in the market and numpy was linked against the fast MKL

Contents

Exercise 5.A (finally, a bit of machine learning!

Generate samples from a Gaussian process, then find its posterior mean. Use numpy. Time the solution.

```
import numpy as np
import time as t
from matplotlib.pyplot import plot, savefig, close, title

# generate GP
x = np.arange(-5,5,0.01).reshape(-1,1); N = len(x)
K = np.exp(-0.5/0.7*(np.dot(x*x,np.ones((1,N)))\
                    +np.dot(np.ones((N,1)),(x*x).T)-2*np.dot(x,x.T)))
Kn = K + np.eye(N)
L = np.linalg.cholesky(Kn)
y = np.dot(L,np.random.randn(N))
K = K.astype(np.float32);Kn = Kn.astype(np.float32);y = y.astype(np.float32);

# solve GP with numpy
start = t.time()
alpha = np.linalg.solve(Kn,y)
mu = np.dot(K,alpha)
print 'CPU: Found solution in %f secs (using numpy.linalg.solve)' % (t.time()-start)
plot(x,y,'bx',x,mu,'k');title('Numpy')
```

Exercise 5.B

Repeat 5.A, but this time avoid matrix inversions by using CG descent. Time the solution. Plot the results.

```
def conjGrad(A,b,tol=1.0e-3):
    N = len(b)
    x = np.zeros(N).astype(np.float32)
    r = b - np.dot(A,x)
    p = r.copy()
    for i in range(N):
        z = np.dot(A,p)
        alpha = np.dot(p,r)/np.dot(p,z)
        x = x + alpha*p
        r = b - np.dot(A,x)
        if (np.sqrt(np.dot(r,r))) < tol:
            break
        else:
            beta = -np.dot(r,z)/np.dot(p,z)
            p = r + beta*p
    print 'Iterations required on CPU:', i
    return x

start = t.time()
alpha = conjGrad(Kn,y)
mu = np.dot(K,alpha)
print 'CPU: Found solution in %f secs' % (t.time()-start)
plot(x,y,'bx',x,mu,'k')
```

Exercise 5.C

Repeat 5.B, but this time use gnumpy.

```python
import gnumpy as g

def conjGradGPU(A,b,tol=1.0e-3):
    N = len(b)
    x = g.zeros(N)
    r = b - g.dot(A,x)
    p = r.copy()
    for i in range(N):
        z = g.dot(A,p)
        alpha = g.dot(p,r)/g.dot(p,z)
        x = x + alpha*p
        r = b - g.dot(A,x)
        if (g.sqrt(g.dot(r,r))) < tol:
            break
        else:
            beta = -g.dot(r,z)/g.dot(p,z)
            p = r + beta*p
    print 'Iterations required on GPU:', i
    return x

K = g.garray(K); Kn = g.garray(Kn); y = g.garray(y)
start = t.time()
alpha = conjGradGPU(Kn,y)
mu = g.dot(K,alpha)
print 'GPU: Found solution in %f secs' % (t.time()-start)
plot(x,y.as_numpy_array(),'bx',x,mu.as_numpy_array(),'r');title('Gnumpy')
```

Results for GP solution using gnumpy

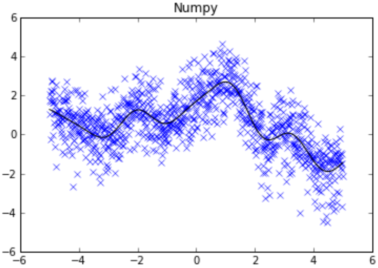
I get this output:

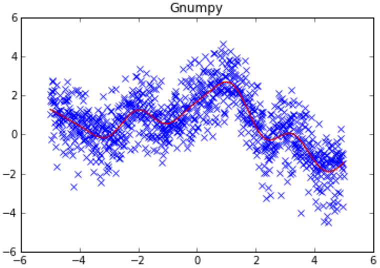
```
(5.A) CPU: Found solution in 0.073970 secs (using numpy.linalg.solve)

(5.B) Iterations required on CPU: 30
(5.B) CPU: Found solution in 0.070011 secs

(5.C) Iterations required on GPU: 29
(5.C) GPU: Found solution in 1.299504 secs

(Using CUDAMat directly) Iterations required on GPU: 30
(Using CUDAMat directly) GPU: Found solution in 0.442285 secs
```





Python support for CUDA not covered here

- Accelerate, within NumbaPro
- scikits.cublas
- (. . .)

Final remarks

- Some operations are much faster on the GPU, even on low-end ones
- Proper timing should take into account
  - Language (for loops are very slow in Python)
  - Precision (32 bits operations are also faster on CPU)
- gnumpy/CUDAMat/cuBLAS are not fully optimized
- Larger matrices result in bigger speedups (if they fit!)

References

- [nVIDIA] CUDA C programming guide. From nVIDIA documentation. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`
- [PyCUDA] PyCUDA documentation. `http://documen.tician.de/pycuda/`
- [AK] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*. (38)3:157-174, 2012.
- [CUDAMat] CUDAMat documentation. `http://www.cs.toronto.edu/~vmnih/docs/cudamat_tr.pdf`
- [gnumpy] gnumpy documentation. `http://www.cs.toronto.edu/~tijmen/gnumpyDoc.html`