Intro to CUDA/CUPY

GPUs are Fast

- Graphics cards have to pump a lot of pixels, with little pixels (mostly) independent. Usually doing basic matrix ops to render.
- Same hardware turned out to be great for massively parallel numerical computations.
- NVidia's CUDA was phase change - made power of GPUs accessible for general purpose programming (GPGPU). See also OpenCL, others…
- Modern GPUs have order 10k cores, ~1 TB/s memory bandwith, 50 TFLOPS (single precision), ~1 PFLOP (tensor cores)
- Memory bandwidth large: ~1 TB/s. Latency very large (often many hundreds of clock cycles).
- Double precision often much, much slower than single. Pay close attention to datatypes.

If You Write Yourself

- You must understand how to split up your job into (tens of) thousands of streams.
- You can, and probably must, handle memory yourself.
- Threads come in blocks, and some resources shared within block. You can manually manage shared memory yourself.
- Reasoning about thread behavior when sharing memory can get very tricky.

Fortunately…

- Many basic libraries ported to CUDA. NVidia supports linear algebra, FFTs. As always, steal if possible.
- High level languages offer GPU support. e.g. cupy, Jax work in python, see also tensorflow, pytorch.
- We'll stick with cupy - very close to drop-in replacement for numpy. Often same code will work for both.

Memory Transfers

- CPU can’t direclty see memory on GPU. You have to transfer data to/from GPU.
- Bandwidth between CPU,GPU much slower than internal bandwidth for both.
- If it takes longer to send your data to GPU than to do problem on CPU, no point in using GPU.

Profiling

- In general, GPU calls are asynchronous. You tell GPU to do something, it says “OK, I’ll go do that”, and returns immediately.
- This is a good thing, but if you aren’t careful you may be confused about run time.
- cupy comes with built-in profiler (cupyx.profiler.benchmark). Suggest you use it.

Matrix Benchmark Revisited

- We'll time our matrix benchmark using cupy.
- Many numpy calls are directly supported. Just replace numpy with cupy
- Cupy allocated directly on gpu. If you want to transfer from gpu, cupy.asnumpy(cupy_array) will transfer to numpy array on CPU.
- cupy.asarray(numpy_array) will transfer to GPU

```
from cupyx.profiler import benchmark
import cupy as cp

n=[100,300,1000,3000,5000,7000,10000,15000]
nrep=[1000,200,20,3,3,1,1,1]

for i,nn in enumerate(n):
    x=cp.random.rand(nn,nn,dtype='float32')
    #x=cp.random.rand(nn,nn) #changed one character!
    t1=time.time()
    for j in range(nrep[i]):
        y=x@x
    print(y[0,0])
    t2=time.time()
    nop=2*nn**3*nrep[i]
    gflops=nop/(t2-t1)/1e9
    print('got ',gflops,' gigaflops on matrix size of ',nn,
    print(benchmark(cp.matmul,(x,x),n_repeat=nrep[i]))
```

```
>>> exec(open("time_matrix_mult_gpu.py").read())
got  50.739185135004355  gigaflops on matrix size of  100
got  269.8426353726098  gigaflops on matrix size of  300
got  484.7687431195046  gigaflops on matrix size of  1000
got  543.5530987963836  gigaflops on matrix size of  3000
got  533.7326663982502  gigaflops on matrix size of  5000
got  548.506438242777  gigaflops on matrix size of  7000
got  551.3213171786251  gigaflops on matrix size of  10000
got  554.5790350019305  gigaflops on matrix size of  15000
```

Above: double precision matrix times.
Below: single precision matrix time.
Factor of ~50 faster.

```
>>> exec(open("time_matrix_mult_gpu.py").read())
got  144.97862117834 12  gigaflops on matrix size of  100  wi
got  3960.0037765538946  gigaflops on matrix size of  300  v
got  18293.769490786173  gigaflops on matrix size of  1000
got  23998.77257796772  gigaflops on matrix size of  3000  v
got  26182.762495318155  gigaflops on matrix size of  5000
got  25521.487883626043  gigaflops on matrix size of  7000
got  25363.916185408034  gigaflops on matrix size of  10000
got  25545.671098791634  gigaflops on matrix size of  15000
```

Laplace Kernel

```
import numpy as np;print("using numpy")
#import cupy as np;print("using cupy")
from cupyx.profiler import benchmark

def kernel(V):
    return V-0.25*(np.roll(V,1,0)+np.roll(V,-1,0)+np.roll(V,1,1)+np.roll(V,-1,1))

n=2048
V=np.zeros([n,n],dtype='float32')
x0=n//2
V[x0,x0]=1
stats=benchmark(kernel,(V,),n_repeat=100)
print("cpu/gpu times (msec): ",np.mean(stats.cpu_times)*1000,np.mean(stats.gpu_times)*1000)
```

```
>>> exec(open("laplace_roll.py").read())
using cupy
cpu/gpu times (msec):  0.2452143561094999 0.490824000239372 3
>>> exec(open("laplace_roll.py").read())
using numpy
cpu/gpu times (msec):  13.675076654180884 13.708930616378783
```

CUDA Kernel

- Under the hood, CUDA is “almost” c++
- Much more control if you use.
- Much, much harder to get right/debug in general. Lots of idiosyncrasies with hardware/threads getting in each other’s way.
- Often required for peak performance.

Adding Arrays

```
__global__
void add_vecs_simple(float *out, float *in1, float *in2, long n)
{
  /* Simple way to add vectors where each element gets one thread.  works great
     for small arrays.*/
  long idx=threadIdx.x+blockDim.x*blockIdx.x;
  if (idx<n)
    out[idx]=in1[idx]+in2[idx];
}
```

```
extern "C" {
void add(float *out, float *in1,float *in2, long n)
{
  long bs=256;  //Set a block size for threads per block
  long nblock=n/bs;
  if ((nblock*bs)<n)
    nblock++;
  add_vecs_simple<<<nblock,bs>>>(out,in1,in2,n);
  printf("err is currently %s\n",cudaGetErrorString(cudaGetLastError()));
```

Adding Arrays

```
__global__
void add_vecs_simple(float *out, float *in1, float *in2, long n)
{
  /* Simple way to add vectors where each element gets one thread.  works great
     for small arrays.*/
  long idx=threadIdx.x+blockDim.x*blockIdx.x;
  if (idx<n)
    out[idx]=in1[idx]+in2[idx];
}
```

```
extern "C" {
void add(float *out, float *in1,float *in2, long n)
{
  long bs=256;  //Set a block size for threads per block
  long nblock=n/bs;
  if ((nblock*bs)<n)
    nblock++;
  add_vecs_simple<<<nblock,bs>>>(out,in1,in2,n);
  printf("err is currently %s\n",cudaGetErrorString(cudaGetLastError()));
```

__global__ keyword says function will be executed on the GPU.
each thread gets a unique ID with threadIdx.x(,y,z) specifying location in thread block, and blockIdx.x(,y,z) specifying which block it's in. blockDim.x(,y,z) says how big thread block is (and gridDim.x(,y,z) says how many blocks there are).

Adding Arrays

```
__global__
void add_vecs_simple(float *out, float *in1, float *in2, long n)
{
  /* Simple way to add vectors where each element gets one thread.  works great
     for small arrays.*/
  long idx=threadIdx.x+blockDim.x*blockIdx.x;
  if (idx<n)
    out[idx]=in1[idx]+in2[idx];
}
```

```
extern "C" {
void add(float *out, float *in1,float *in2, long n)
{
  long bs=256;  //Set a block size for threads per block
  long nblock=n/bs;
  if ((nblock*bs)<n)
    nblock++;
  add_vecs_simple<<<nblock,bs>>>(out,in1,in2,n);
  printf("err is currently %s\n",cudaGetErrorString(cudaGetLastError()));
```

inside add_vecs_simple, idx calculates a global position. If this position is inside our array, then set the output for that index equal to the sum of the inputs.

Adding Arrays

```
__global__
void add_vecs_simple(float *out, float *in1, float *in2, long n)
{
  /* Simple way to add vectors where each element gets one thread.  works great
     for small arrays.*/
  long idx=threadIdx.x+blockDim.x*blockIdx.x;
  if (idx<n)
    out[idx]=in1[idx]+in2[idx];
}
```

```
extern "C" {
void add(float *out, float *in1,float *in2, long n)
{
  long bs=256;  //Set a block size for threads per block
  long nblock=n/bs;
  if ((nblock*bs)<n)
    nblock++;
  add_vecs_simple<<<nblock,bs>>>(out,in1,in2,n);
  printf("err is currently %s\n",cudaGetErrorString(cudaGetLastError()));
```

can call CUDA kernels from C, including within same file. The wrapper decides how many blocks it needs, then configures the CUDA call by specifying how many thread blocks, how many threads per block in <<<>>> clause.
Can also check errors, with e.g. cudaGeLastError, and cudaGetErrorString
extern “C” says to not name-mangle, makes it easier to find correct function in library.

Calling from Python

```
mylib=ctypes.cdll.LoadLibrary("libaddVecs.so")
add_cuda_c=mylib.add
add_cuda_c.argtypes=[ctypes.c_void_p,ctypes.c_void_p,ctypes.c_void_p,ctypes.c_long]
```

inside python, we'll use ctypes again (many other options exist). GPU/CPU pointers look the same to system, it's only when you try to use contents that difference is clear.

```
def add_cuda_simple(c,a,b):
    n=len(a)
    add_cuda_c(c.data.ptr,a.data.ptr,b.data.ptr,n)
```

Often convenient to have a wrapper function that handles measuring sizes, pulling pointers etc.

```
n=2048
a=cp.ones(n,dtype='float32')
b=cp.ones(n,dtype='float32')
c=cp.zeros(n,dtype='float32')
add_cuda(c,a,b)
```

Routines we wrote expected data on GPU (note lack of any host/device transfers). We can add arrays on GPU directly.

Slightly More Complicated Add

```
__global__
void add_vecs(float *out, float *in1, float *in2, long n)
{
  //slightly more complicated version that should work with arbitrarily large arrays*/
  for (long i=threadIdx.x+blockDim.x*blockIdx.x;i<n;i+=gridDim.x*blockDim.x)
    out[i]=in1[i]+in2[i];
```

For large arrays, we can't create enough threads to assign one thread per element. We have to loop. One way is for each thread to start as per usual, but then take steps of total # of threads. For 1D grids, that step is gridDim.x*blockDim.x

Calling routine from C looks the same, except you put a cap on # of thread blocks.

```
extern "C" {
void add2(float *out, float *in1,float *in2, long n)
{
  long bs=256;
  long nblock=n/bs;
  if ((nblock*bs)<n)
    nblock++;
  long nblock_max=128;
  if (nblock>nblock_max)
    nblock=nblock_max;
  add_vecs<<<nblock,bs>>>(out,in1,in2,n);
```

2D Kernel

```
__global__
void apply_stencil_cuda(float *out, float *in, long n, long m)
{
  for (long i=threadIdx.x+blockDim.x*blockIdx.x;i<n-1;i+=blockDim.x*gridDim.x)
    if (i>0) {
      for (long j=threadIdx.y+blockDim.y*blockIdx.y;j<m-1;j+=blockDim.y*gridDim.y)
        if (j>0) {
          long ind=i*m+j;
          float left=in[ind-1];
          float right=in[ind+1];
          float bot=in[ind-m];
          float top=in[ind+m];
          out[ind]=in[ind]-0.25*(left+right+top+bot);
        }
```

Example of a 2D kernel. Now threads have x,y indices, that we use to access a 2D array, where A[i,j]-> A[i*m+j]
Our top/bottom neighbors are m away from us. NB - we need to not access out of bounds, hence i>0, j>0 calls.

```
extern "C"
{
void apply_stencil(float *out, float *in, long n, long m)
{
  dim3 bs(16,16);
  dim3 nb(16,16);
  apply_stencil_cuda<<<nb,bs>>>(out,in,n,m);
```

And from Python

```
mylib=ctypes.cdll.LoadLibrary("liblaplace.so")
apply_stencil=mylib.apply_stencil
apply_stencil.argtypes=[ctypes.c_void_p,ctypes.c_void_p,ctypes.c_long,ctypes.c_long]
```

```
n=4096
V=cp.zeros([n,n],dtype='float32')
V2=cp.zeros([n,n],dtype='float32')
x0=n//2
V[x0,x0]=1
apply_stencil(V2.data.ptr,V.data.ptr,n,n)
```

python calls look like they have in the past. We could have (but I did not) write a handy python wrapper to hide the pointer access.

ctypes call from python