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

```python
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  144.9786211783412  gigaflops on matrix size of  100  wi
got  3960.0037765538946  gigaflops on matrix size of  300  w
got  18293.769490786173  gigaflops on matrix size of  1000
got  23998.77257796772  gigaflops on matrix size of  3000  w
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 0.4908240002393723
>>> 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;
  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()));
```