GENERAL CUDA TIPS

CS179 2021 – ETHAN JASZEWSKI

NVIDIA LIBRARIES

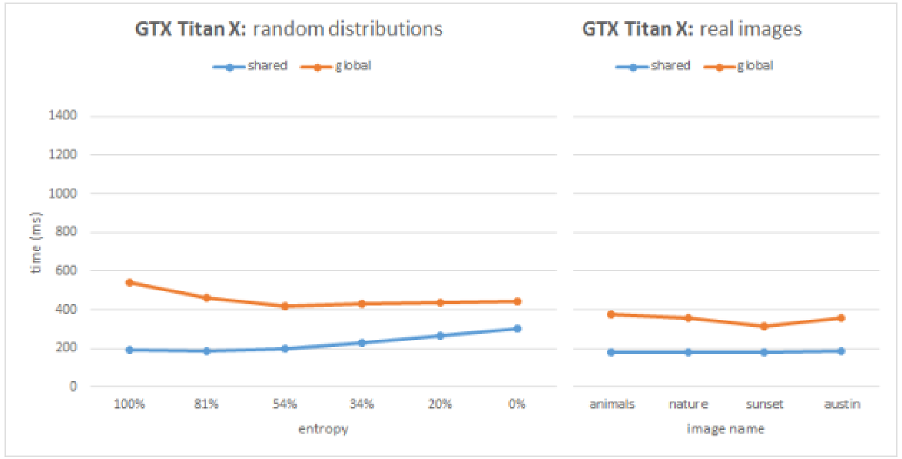
- cuBLAS
  - Generalized BLAS package that runs on GPU
  - Highly performance tuned (**never** write your own BLAS kernels)
- cuSPARSE
  - Performant sparse matrix multiplication library
  - Supports fully sparse and mixed (i.e., sparse and dense) operations
- cuSOLVER
  - Performant solvers for dense and sparse matrices
- cuRAND
  - Performant GPU-oriented random number generation
  - Callable both in-kernel and via host code
- cuFFT
  - Performant GPU FFT with FFTW-style interface
  - Supports multi-gpu and asynchronous computation
- cuDNN
  - GPU-accelerated library of ML primitives
  - Highly performance tuned

USEFUL OPERATIONS

- Warp Shuffle
  - Fast way to exchange data within a warp
  - Helps avoid `__syncthreads()` calls
  - Helps simplify code significantly

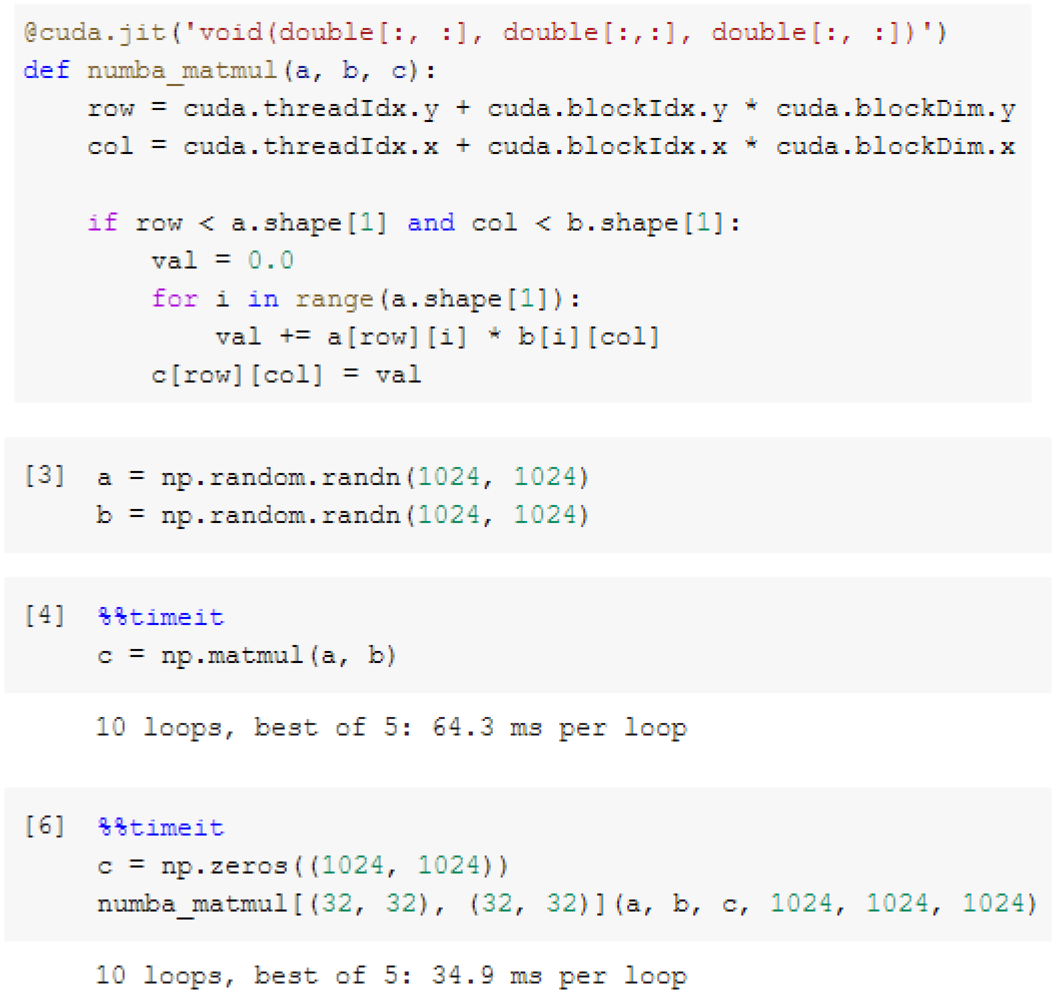
```
__inline__ __device__
int warpAllReduceSum(int val) {
    for (int mask = warpSize/2; mask > 0; mask /= 2)
        val += __shfl_xor(val, mask);
    return val;
}
```

- Shared Memory Atomics
  - Normal atomic operations – just to shared memory
  - Much faster than global memory atomics
  - Available on Maxwell and later GPUs



NUMBA

- Essentially just CUDA in Python
- Easy interop with Numpy
- Can be used through Jupyter (i.e., Google Colab)
- Provides bindings to Nvidia APIs through pyculib
  - pyculib provides:
    - cuBLAS
    - cuSPARSE
    - cuRAND
    - cuFFT



```
@cuda.jit('void(double[:, :], double[:,:], double[:, :])')
def numba_matmul(a, b, c):
    row = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y
    col = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

    if row < a.shape[1] and col < b.shape[1]:
        val = 0.0
        for i in range(a.shape[1]):
            val += a[row][i] * b[i][col]
        c[row][col] = val
```

```
[3]  a = np.random.randn(1024, 1024)
     b = np.random.randn(1024, 1024)
```

```
[4]  %%timeit
     c = np.matmul(a, b)

     10 loops, best of 5: 64.3 ms per loop
```

```
[6]  %%timeit
     c = np.zeros((1024, 1024))
     numba_matmul[(32, 32), (32, 32)](a, b, c, 1024, 1024, 1024)

     10 loops, best of 5: 34.9 ms per loop
```

MISC. TIPS

- Pay attention to block and grid dimensions
  - Well-dimensioned kernels can be a lot faster
  - Different kernels might need different block shapes
- Avoid atomic operations when possible
  - Atomic operations and `__syncthreads()` are slow!
- Pay attention to memory
  - Most kernels are memory bandwidth limited
  - Try to do a “good amount” of work for the input data
- Use the Nvidia APIs
  - They are highly optimized for most operations
- Avoid premature optimization
  - Start simple, then improve iteratively
  - Don’t optimize things that don’t matter
- Enable –O3 on your compiler!

QUESTIONS?