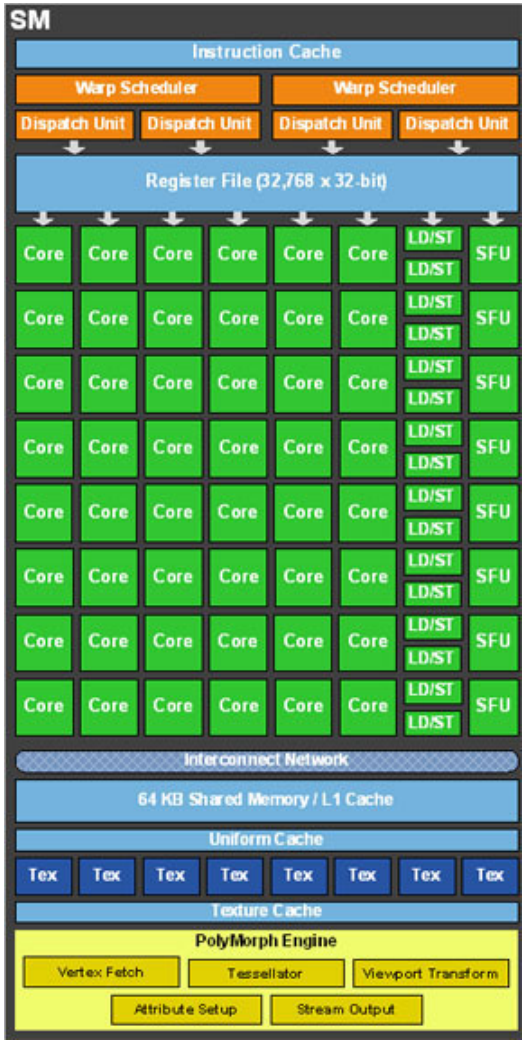
CS195V Week 11

CUDA Part 2

Shared Memory and Thread Communication

- Recall shared memory from last lecture
  - Fast, equivalent to L1 cache (~64kb)
  - Shared between threads in a block
- Best way to communicate between threads
  - You can synchronize across threads in a block to ensure r/w is complete

Global memory requires you to use multiple kernel invocations (since you don't know when a r/w op is done)



Reductions

- A commonly used operation is the reduction operation, where some function (ie. sum) is used to sum up the values of an array
- The bad but easy way:

```
extern __shared__ float cache[];
cache[i] = threadIdx.x;
__syncthreads(); //ensure all threads done writing to
cache
if(thread.Idx == 0){
    for(int i=0; i<N; i++){
        cache[0] += cache[i];
    }
    printf("%f\n", cache[0]);
}
```

Reductions

- The better way:

```
extern __shared__ float cache[];
cache[i] = threadIdx.x;
__syncthreads();

for(int i = blockDim.x; i>0; i >>= 1) {
    int halfPoint = (i >> 1);
    if(threadIdx.x < halfPoint)
        cache[threadIdx.x]+=cache[threadIdx. x +
halfPoint];
    __syncthreads();
}

printf("%f\n", cache[0]);
```

CUDA Random Number Generation (RNG)

- Historically it has been very difficult to create random numbers on the GPU
  - Had to sample from random textures / implement your own PRNG (ie. LCG, Mersenne, etc.)
  - Implementing a parallel PRNG isn't trivial (one option is to give each PRNG thread a different seed, but you loose some randomness guarantees)!
  - And you wonder why GLSL noise() doesn't work...
- Compute 2.0 added built in functionality for RNG

Random headers can be found in <curand.h> and <curand_kernel.h>

CUDA Random Number Generation (RNG)

- We will give a randState to each CUDA thread, from which it can sample from
- On the host, create a device pointer to hold the randomStates
- Malloc number of states equal to number of threads
- Pass the device pointer to your function
- Init the random states
- Call a random function (ie. curand_uniform) with the state given to that thread
- Free the randomStates

CUDA Random Number Generation (RNG)

```
__global__ void init_stuff(curandState *state) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    curand_init(1337, idx, 0, &state[idx]);
}
__global__ void make_rand(curandState *state, float
*randArray)  {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    randArray[idx] = curand_uniform(&state[idx]);
}
void host_function() {
    curandState *d_state;
    cudaMalloc(&d_state, nThreads * nBlocks);
    init_stuff<<<nblocks, nthreads>>>(d_state);
    make_rand<<<nblocks, nthreads>>>(d_state, randArray);
    cudaFree(d_state);
}
```

CUDA Random Number Generation (RNG)

- The total state space of the PRNG before you start to see repeats is about 2^190
- CUDA's RNG is designed such that given the same seed in each thread, it will generate random numbers spaced 2^67 numbers away in the PRNG's sequence
  - When calling curand_init with a seed, it scrambles that seed and then skips ahead 2^67 numbers (this is kind of expensive but has some nice properties)
  - This even spacing between threads guarantees that you can analyze the randomness of the PRNG and those results will hold no matter what seed you use

CUDA Random Number Generation (RNG)

- What if you're running millions of threads and each thread needs RNs?
  - Not completely uncommon
  - You could run out of state space per thread and start seeing repeats... ((2^190) / (10^6)) / (2^67) = 1.0633824 × 10^31
- Can seed each thread with a different seed (ex. theadIdx.x), and then set the state to zero (ie. don't advance each thread by 2^67)
  - This may introduce some bias / correlation, but not many other options
  - Don't have the same assurance of statistical properties remaining the same as seed changes
  - It's also faster (by a factor of 10x or so)

CUDA Random Number Generation (RNG)

- Why do we lose some statistical guarantees of randomness?
- Suppose we choose seeds equal to the threadIdx.x (ie. 0,1,2...)
- Now suppose the seed scrambler (essentially a hash function) has a collision between threads 0 and 4
  - This means threads 0 and 4 will be generating the same sequence of numbers
- There could also be bias introduced by the choice of hash function itself...

CUDA Random Number Generation (RNG)

- The take home message:
  - Depending on your problem you may need to be careful when using CUDA's RNG (ie. crypto)
  - If you're making pretty pictures (ie. graphics) it probably doesn't matter

CUDA Libraries

- CUDA has a lot of libraries that you can use to make things much easier
  - Lots of unofficial libraries, but we'll cover some of the main included libraries here
- Not used by our projects, but if you pursue CUDA in the future, you will most certainly make use of these

CUFFT

- Based on the successful FFTW library for C++
- Adds FFT (Fast Fourier Transform) functionality to CUDA, as you might expect
- 1D, 2D, 3D, complex and real data
- 1D transform size of up to 128 million elements
- Order of magnitude speedup from multi-core CPU implementations

CUFFT Example (3D)

```
#include <cufft.h>
#define NX 64
#define NY 64
#define NZ 128
cufftHandle plan;
cufftComplex *data1, *data2;
cudaMalloc((void**)&data1, sizeof(cufftComplex)*NX*NY*NZ);
cudaMalloc((void**)&data2, sizeof(cufftComplex)*NX*NY*NZ);
/* Create a 3D FFT plan. */
cufftPlan3d(&plan, NX, NY, NZ, CUFFT_C2C);
/* Transform the first signal in place. */
cufftExecC2C(plan, data1, data1, CUFFT_FORWARD);
/* Transform the second signal using the same plan. */
cufftExecC2C(plan, data2, data2, CUFFT_FORWARD);
/* Destroy the cuFFT plan. */
cufftDestroy(plan);
cudaFree(data1); cudaFree(data2);

// note that the cufft code takes the place of your cuda kernel
```

CUBLAS

- CUDA Basic Linear Algebra Subroutines
- Lets you do linear algebra-y things easily
  - 152 standard BLAS (Basic Linear Algebra Subprogram) operations
- Easy way to do matrix multiplication, etc.
- Execution is very similar to CUFFT
  - You get a handle and call built in functions on it with your data
  - Kernel launches replaced by library functions

Other CUDA Libraries

- CURAND
  - Generate large batches of random numbers
- CUDA math library
  - normal math functionality, exactly like C/C++ (same #include <math.h>)
- CUSPARSE
  - Sparse matrix manipulation
- NVIDIA Performance Primitives (NPP)
  - Image, signal processing primitives
- Thrust
  - Parallel algorithms and data structures

Ant Colony Optimization

http://www.csse.monash.edu.au/~berndm/CSE460/Lectures/cse460-9.pdf

Parallelizing ACO on the GPU

- We can have thousands or even millions of ants
  - Easy way of splitting up the work is to give each worker ant a thread
- Recall that each ant must be able to add its tour to the pheromone graph before proceeding to the next time step
  - Need to synchronize across ants (ie. after each ant has constructed their tour, they must add pheromone to the graph before we can compute the next time step's traversal probabilities)

Parallelizing ACO on the GPU

- We don't have any easy way to synchronize across all threads in the GPU
- In fact the only way is to call multiple kernel invocations with a cudaThreadSynchronize()

```
for(int i=0; i<numIterations; i++) {
      tsp<<<1, numAnts>>>();
      cudaThreadSynchronize();
}
```

  - This is expensive (we're not reaching full occupancy)
- Recall that we *do* have built in functionality to synchronize threads in a block without exiting the kernel...

Multi Colony Optimization on the GPU

- An easy way of parallelizing ACO is to create multiple colonies of ants
  - Each colony of ants stores their own pheromone graph and creates multiple iterations of tours
  - Every so often, communicate between the different colonies (ie. pass best paths / share pheromone information)
- Intuitive subdivision!
  - Number of blocks = Number of colonies
  - Number of ants per block = Number of threads per block
  - Total number of threads = Number of ants x Number of colonies

Multi Colony Optimization on the GPU

- Now when all ants have created their tours, we can sum up their total contribution for each edge across all threads in a block by using __syncthreads() and a reduction
- Similarly, we can update the pheromone evaporation by just having each thread in a block update some set of edges, and then __syncthreads() again before the next tour construction step