Lecture plan

- Last time: Graph Algorithms
- This time: GPU programming

Graph Algorithms: Summary

- Core algorithms use DFS and BFS
- Look for BFS solutions: easier to parallelize
- Look for familiar sub-algorithms (planarity, maximum independent subsets, etc...)

Early history (redux)

- Late 80s-early 90s: "golden age" for supercomputing
  - Companies: Thinking Machines, MasPar, Cray
  - Relatively fast processors (vs memory)
  - Lots of academic interest and development
  - But got hard to compete with commodity hardware: Scientific computing is not a market driver!
- 90s-early 2000s: age of the cluster
  - Beowulf, grid computing, etc.
  - "Big iron" also uses commodity chips (better interconnect)
- Past few years
  - CPU producers move to multicore
  - High-end graphics becomes commodity hardware
  - Gaming is a market driver!
  - GPU producers realize their many-core designs can apply to general purpose computing

Thread design points

- Threads on desktop CPUs
  - Implemented via lightweight processes (for example)
  - General system scheduler
  - Thrashing when more active threads than processors
- An alternative approach
  - Hardware support for many threads / CPU
  - Modest example: hyperthreading, More extreme: Cray MTA-2 and XMT
  - Hide memory latency by thread switching
  - Want many more independent threads than cores
- GPU programming
  - Thread creation / context switching are basically free
  - Want lots of threads

General-purpose GPU programming

- Old GPU model: use texture mapping interfaces
- CUDA (Compute Unified Device Architecture)
  - More natural general-purpose programming model
  - Initial release in 2007; now in version 11.8
- OpenCL
  - In Apple's Snow Leopard release
  - Open standard: includes NVidia, ATI, etc
- OpenACC
  - Introduced in 2012
  - Open standard: Includes Cray, Nvidia (but not Intel!)
  - Up to version 2.17
- TBB (Threading Building Block) introduced 2016
  - Competing Intel standard
  - C++ template library, runtime

What exactly is CUDA?

- Compute Unified Device Architecture
- Exposes GPU architecture for general purpose computing
- Does so using standard c/c++ library
- Relatively small set of extensions
- Wrappers for other languages (ex: python, java)

Compiling CUDA

- `nvcc` is the driver
- Builds on top of g++ or other compilers
- `nvcc` driver produces CPU and PTX code
- Must Load kernel module: `module load cuda`: `nvcc filename.cu`

Programming in CUDA

1. Copy data from CPU Memory to GPU Memory
2. Load GPU Code and execute on GPU hardware
3. Copy data from GPU Memory to CPU Memory

CUDA programming in specific

```
do_something_on_cpu();
some_kernel<<<nBlk, nTid>>>(args);
do_something_else_on_cpu();
cudaThreadSynchronize();
```

- Highly parallel kernels run on device
- Vaguely analogous to parallel sections in OpenMP code
- Rest of the code on host (CPU)
- C++ extensions to program both host code and kernels

Thread blocks

- Monolithic thread array partitioned into blocks
  - Blocks have 1D or 2D numeric identifier
  - Threads within blocks have 1D, 2D, or 3D identifier
  - Identifiers help figure out what data to work on
- Blocks cooperate via shared memory, atomic operations, barriers
- Threads in different blocks cannot cooperate... except for implied global barrier from host

CUDA memory model

- *Registers* are registers; per thread
- *Shared* memory is small, fast, on-chip; per block
- *Global memory* is large uncached off-chip: Also accessible by host
- Support for texture memory and constant memory

Basic outline

1. Perform any needed allocations
2. Copy data from host to CPU memory
3. Invoke kernel
4. Copy results from GPU to host
5. Clean up allocations

GPU memory management

```
h_data = malloc(size);
... Initialize h_data on host ...
cudaMalloc((void**) &d_data, size);
cudaMemcpy(d_data, h_data, size, cudaMemcpyHostToDevice);
... invoke kernel ...
cudaMemcpy(h_data, d_data, size, cudaMemcpyDeviceToHost);
cudaFree(d_data);
free(h_data);
```

- Don't dereference h_data on device or d_data on host!
- Can also copy host-to-host, device-to-device
- Kernel invocation is asynchronous with CPU; cudaMemcpy is synchronous (can synchronize kernels with cudaThreadSynchronize)

CUDA function declarations

```
__device__ float device_func();
__global__ void kernel_func();
__host__ float host_func();
```

- `__global__` for kernel (must return void)
- `__device__` functions called and executed on device (GPU)
- `__host__` functions called and executed on host
- `__device__` and `__host__` can be used together

Restrictions on device functions

- No taking the address of a `__device__` function
- No recursion
- No static variables inside the function
- No varargs

Kernel invocation with execution configuration

```
__global__ void kernel_func(...);
dim3 dimGrid(100, 50); // 5000 thread blocks
dim3 dimBlock(4, 8, 8); // 256 threads per block
size_t sharedMemBytes = 64;
kernel_func<<dimGrid, dimBlock, sharedMemBytes>>(...);
```

- Can write integers (1D layouts) for first two arguments
- Third argument is optional (defaults to zero)
- Optional fourth argument for stream of execution
- Used to specify asynchronous execution across kernels
- Kernel can fail if you request too many resources

Simple famous example

```
__global__ void mykernel(void) {
printf("Hello World from GPU!\n");
}
int main(void) {
mykernel<<<1,1>>>();
printf("Hello World from CPU!\n");
return 0;
}
```

Comments on famous example

- Executed on GPU: Triple brackets mark a call from host code to device
- Called from CPU
- nvcc separates GPU and CPU code
- Code for GPU are compiled to that device
- main and other function compiled for host system

Vector Addition

```
__global__ void VecAdd(float *a, float *b, float *c) { ...
```

- `add` runs on the device.
- So a, b, and c must be pointers in device memory, *not* the CPU
- We must allocate memory on the GPU and copy the data

Memory Management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - May be passed to/from host code
    - May not be dereferenced in host code
  - Host pointers point to CPU memory
    - May be passed to/from device code
    - May not be dereferenced in device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`

Vector Addition Memory Management

```
// Allocate on "device"
cudaMalloc((void**)&d_A, size);
cudaMalloc((void**)&d_B, size);
cudaMalloc((void**)&d_C, size);
// Copy from CPU memory to GPU memory
cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
.
.
.
cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);
```

Shared memory

Size known at compile time:

```
__global__ void kernel(...)
{
__shared__ float x[256];
...
}
kernel<<<nb,bs>>>(...);
```

Size known at kernel launch:

```
__global__ void kernel(...)
{
extern __shared__ float x[];
...
}
kernel<<<nb,bs,bytes>>>(...)
```

Vector Addition Parallelism

```
int threadsPerBlock = 256;
int blocksPerGrid = (N+255) / threadsPerBlock;
VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A,d_B,d_C,N);
```

Synchronization

```
void __syncthreads();
```

- Synchronizes all threads within a block
- Used to prevent data hazards
- All threads must reach the barrier
- In conditional code, the condition must be uniform across the block

Coordinating Host and Device

- Kernel launches are asynchronous
- Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results
- `cudaMemcpy()` Blocks the CPU until the copy is complete. Copy begins when all preceding CUDA calls have completed
- `cudaMemcpyAsync()` Asynchronous, does not block the CPU.
- `cudaDeviceSynchronize()` Blocks the CPU until all preceding CUDA calls have completed

CUDA errors

- All CUDA API calls return an error code with type `cudaError_t`
- Error in: the API call itself, an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error: `cudaError_t cudaGetLastError(void)`
- Get a string to describe the error:
  `char *cudaGetErrorString(cudaError_t)`
  `printf("%s", cudaGetErrorString(cudaGetLastError()));`

Error checking function

```
void CheckCudaError() {
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
    printf("Error: %s\n", cudaGetErrorString(err));
    exit(-1);
    }
}
```

Summary: CUDA extensions

- Type qualifiers:
  - `global`
  - `device`
  - `shared`
  - `local`
  - `constant`
- Keywords (`threadIdx`, `blockIdx`)
- Intrinsics (`__syncthreads`)
- Runtime API (memory, symbol, execution management)
- Function launch

Libraries

- CUBLAS, CUFFT, CUDA LAPACK bindings (commercial)
- CUDA-accelerated libraries
- Bindings to CUDA from Python, Java, etc...

Hardware example (G80)

- 128 processors execute threads
- Thread Execution Manager issues threads
- Parallel data cache / shared memory per processor
- All have access to device memory
  - Partitioned into global, constant, texture spaces
  - Read-only caches to texture and constant spaces

Hardware threads

- Single Instruction, Multiple Thread (SIMT)
- A warp of threads executes physically in parallel (one warp $==$ 32 parallel threads)
- Blocks are partitioned into warps by consecutive thread ID
- Best efficiency when all threads in warp do same operation
- Conditional branches reduce parallelism: serially execute all paths taken

G80 memory architecture

- Memory divided into 16 banks of 32-byte words
- Each bank services one address per cycle
- Conflicting accesses are serialized
- Stride 1 (or odd stride): no bank conflicts

Memory coalescing

- Coalescing is a coordinated read by half-warp
- Read contiguous region (64, 128, or 256 bytes)
- Starting address for region a multiple of region size
- Thread k in half-warp accesses element k of blocks
- Not all threads need to participate

OpenACC

- new Nvidia standard like OpenMP but for GPUs
- Directives via #pragmas
- new runtime API

OpenACC directives

Computation:

- `#pragma acc parallel`: Parallel execution block
- `#pragma acc serial`: Serial execution block
- `#pragma acc kernels`: Compile into sections for GPU, each loop is a separate kernel

Data:

- `#pragma acc data`: copy data to and from the accelerator (done automagically upon request with `copyin, copyout`)
- `#pragma acc host_data`: makes the address of data in device memory available on the host
- `#pragma acc atomic`

Loop annotation:

- `#pragma acc loop`: Mark loop for parallelization
  - vector - execute in SIMD mode
  - tile(size) - Split into subloops
  - reduction(o) - reduction operator

Runtime API

- Synchronization: `acc_async_wait()`, `acc_async_wait_all()`
- Memory allocation: `acc_malloc()`, `acc_free()`
- Information: `acc_get_num_devices`