CMPSCI 691AD – General Purpose Computation on the GPU

Spring 2009

Lecture 4: CUDA Programming Basics 2

Rui Wang

CUDA API

- Minimal extension to the C programming language
- A runtime library split into:
  - Host component: runs on host.
  - Device component: runs on device.
  - Common component: subset of C library that are supported in both host and device.

Extensions to C

- Four extensions:
  - Function type qualifiers
  - Variable type qualifiers
  - Kernel calling directive
  - 5 built-in variables

Extensions to C

- Four extensions:
  - Function type qualifiers
  - Variable type qualifiers
  - Kernel calling directive
  - 5 built-in variables

Function Type Qualifiers

- Indicate where a function executes, and where can a function be called from.
- __global__
  - Declares a function as kernel function
  - Callable from host only
  - Executes on the device
  - Must return void
  - Must be called using directive: <<< … >>>
  - Executes asynchronously

Function Type Qualifiers

- __device__
  - Executes on the device
  - Callable from device functions only
- __host__
  - Executes on the host, callable from the host only
  - This qualifier is optional
- __device__ and __host__ can be used together, in which cast the function is compiled for both the host and the device.

Function Type Qualifiers

- Restrictions
  - __device__ and __global__ do not support recursion.
  - __device__ and __global__ cannot have a variable number of arguments.
  - Cannot take pointers to __device__ functions.

Extensions to C

- Four extensions:
  - Function type qualifiers
  - Variable type qualifiers
  - Kernel calling directive
  - 5 built-in variables

Variable Type Qualifiers

- Indicate where a variable resides in device memory.
  - Lifetime and scope of the variable
- __device__
  - Resides in global memory space
  - Has the lifetime of an application
  - Accessible from all threads, and the host (through the runtime library)

Variable Type Qualifiers

- __constant__
  - Similar to __device__ but read-only and resides in constant memory space
  - Has the lifetime of an application
  - Accessible from all threads, and the host (through the runtime library)
  - Must be assigned from the host through runtime library.

Variable Type Qualifiers

- __shared__
  - Resides in the shared memory space of a block
  - Has the lifetime of a block
  - Only accessible from from all threads within the block
  - Writes to shared memory only guaranteed to be visible to other threads after __syncthreads().

Variable Type Qualifiers

- __shared__
  - Declared as

    extern __shared__ float shared[];
  - Cannot be initialized on declaration.
  - All variables declared this way have the same address in memory.
    - Absolute addressing.
  - Access speed very fast, can be thought of as user managed L1 cache.

Variable Type Qualifiers

- Notes
  - Qualifiers not allowed on formal parameters and local variables within a function that executes on the host.
  - Variables without any qualifier generally resides in register space, unless for large structures, which may be placed in local memory by compiler.
    - Inspect -ptx code will tell you where they are placed.

Extensions to C

- Four extensions:
  - Function type qualifiers
  - Variable type qualifiers
  - Kernel calling directive
  - 5 built-in variables

Kernel Calling Directive

- For any call to a __global__ function.
- Specifies:
  - Configuration of threads and blocks
  - Amount of shared memory to be allocated per block (optional)
  - Streams (optional)

Kernel Calling Directive

- In the form of <<< Dg, Db, Ns, S >>> (parameters)
- Dg:
  - of type dim3
  - specifies grid dimension
    - # of blocks = Dg.x * Dg.y
- Db: of type dim3
  - of type dim3
  - specifies block dimension
    - # of threads/block = Db.x * Db.y * Db.z

Kernel Calling Directive

- Example:

```
__global__ void Func(float* parameter) {
    ...
}

void main()
{
    dim3 dimBlock(16, 16);
    dim3 dimGrid(4, 4);

    Func<<<dimGrid, dimBlock>>>(parameter);
}
```

- Parameters are passed in shared memory.
- Call fails if configuration exceeds device limit.

Extensions to C

- Four extensions:
  - Function type qualifiers
  - Variable type qualifiers
  - Kernel calling directive
  - 5 built-in variables

5 Built-in Variables

- gridDim
  - of type dim3
  - contains the dimensions of the grid

```
__global__ void Func(float* parameter) {

    int dgx = gridDim.x;
    int dgy = gridDim.y;
}
```

- blockDim
  - similar, contains the dimensions of the block

5 Built-in Variables

- blockIdx
  - of type uint3
  - contains block index
- threadIdx
  - similar, contains thread index within the block
- warpSize
  - of type int
  - Contains # threads in a warp (32)

Common Runtime Component

- Can be used in both host and device functions
- Built-in vector types

```
char1, uchar1, char2, uchar2, char3,
uchar3, char4, uchar4, short1, ushort1,
short2, ushort2, short3, ushort3, short4,
ushort4, int1, uint1, int2, uint2, int3,
uint3, int4, uint4, long1, ulong1, long2,
ulong2, long3, ulong3, long4, ulong4,
float1, float2, float3, float4, double2
```

- Vector components accessible through x,y,z,w fields
- Default constructor: make_xxxx(...)

Common Runtime Component

- Type dim3
  - Based on uint3, uninitialized values default to 1
- Math functions
  - Appendix B of the programming guide
  - Some functions (e.g. sinf) have less accurate but very fast version (__sinf), only available on device
  - Compiler option: -use_fast_math
- Timing function
  - clock_t clock();

Synchronization of Host and Device

- Some runtime functions are asynchronous: control is returned to host immediately after call
  - __global__ function calls
  - Memory copy functions with suffix Async
  - Functions that perform device↔device memory copies
  - Functions that set memory.
- Hence host and device can run simultaneously.
- On the other hand, new kernel calls, memory set or copy functions execute only after all preceding device operations are completed.

Synchronization of Host and Device

- cudaMemcpy() is synchronous
  - Control returns to host after copy is completed
  - Copy starts after all previous CUDA calls are completed.
- cudaThreadSynchronize()
  - Blocks until all previous CUDA calls are completed.
  - Useful for timing on the host side

Synchronization of Host and Device

- cudaThreadSynchronize()
  - Blocks until all previous CUDA calls are completed.
  - Useful for timing on the host side

```
// start timer
kernel <<<...>>> (… …)
cudaThreadSynchronize();
// stop timer
// calculate timer difference
```

Synchronization of Host and Device

- Note: cudaThreadSynchronize vs. __syncthreads
  - __syncthreads() is invoked from *device* code and synchronizes all threads *in a block*
  - cudaThreadSynchronize() is invoked from *host* code and synchronizes *all threads*

Device Runtime Component

- Atomic Functions

  An atomic function performs a Read-Modify-Write atomic operation on one 32-bit or 64-bit word residing in global or shared memory.

  - Such as: `count[i]++;`
  - No other thread can access the element being modified until the operation is complete.
  - Appendix C in the programming guide.

Device Runtime Component

- Warp Vote Functions

```
int __all(int predicate);
```

The value of 'predicate' is evaluated for all threads of a warp, and returns non-zero if and only if all evaluations are non-zero. (Analogous to AND)

```
int __any(int predicate);
```

Similar, but performs an OR

Host Runtime Component

- Device Management
- Context Management
- Module Management
- Execution Control
- Memory Management
- Texture Reference Management
- Interoperability with OpenGL and D3D

Memory Management

- Device Memory
  - 32-bit address space, allocated as either
    - *Linear memory*
      Analogous to CPU memory allocation, 1D only
    - or *CUDA array*

      Memory layout optimized for texture fetching, up to 3D, each element can have 1,2, or 4 components. Associated with textures only.

Device Memory

- Linear Memory
  - Allocated using cudaMalloc() o cudaMallocPitch()
  - Freed using cudaFree()
  - Example:

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256*sizeof(float));
```

  - Pitch is recommended for allocating 2D arrays → padding to meet memory alignment requirements.

Device Memory

- Linear Memory
  - Allocated using cudaMalloc() o cudaMallocPitch()
  - Freed using cudaFree()
  - Example:

Good naming convention is important!

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256*sizeof(float));
```

  - Pitch is recommended for allocating 2D arrays → padding to meet memory alignment requirements.

Device Memory

- CUDA Array
  - Allocated using cudaMallocArray()
  - Freed using cudaFreeArray()
  - Requires a format descriptor

```
cudaChannelFormatDesc desc =
        cudaCreateChannelDesc<float>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &desc, width,height);
```

Memory Management

- Host Memory
    - CUDA runtime can allocate page-locked host memory using cudaMallocHost.
    - The physical address is locked and cannot be swapped out to disk.
    - This provides much higher bandwidth for transfer between CPU and GPU, compared to page-able memory allocated using malloc.
    - However, don't be too greedy!

Memory Management

- Transfer Between Device and Host
  - Basic function: `cudaMemcpy(dst, src, count, kind)`
  - 'Kind' maybe: host-to-host, host-to-device, device-to-host, device-to-device.
  - Make sure your pointers match the kind!
  - There is also a `cudaMemcpyAsync(dst, src, count, kind)`
  - Lots of other variations, read the CUDA reference manual.