CS 380 - GPU and GPGPU Programming
Lecture 12+13: Shading and Compute APIs 3+4

Markus Hadwiger, KAUST
Reading Assignment #7 (until March 26)

Read (required):

- Programming Massively Parallel Processors book, Chapter 5 (*CUDA Memories*)

- **CUDA C Programming Guide 5.5**
  Appendix F: Compute Capabilities

  Study the different memory access requirements for different compute capabilities
Semester Project (Proposal until March 26)

Try to find your own topic of interest

• Pick something that you think is really cool
• Can be completely graphics or completely computation or both combined
• Browse GPU Gems books (online/in library) for ideas, browse SDK examples, look online
• Can be built on frameworks, NVIDIA OpenGL SDK or NVIDIA CUDA SDK

Write project proposal

• 1-2 pages (pdf), just overview of plan
• Talk to us before you start writing! (regarding content and complexity)
  – Hand in proposal (to Peter) until March 26

Project presentations in final exam week; write final report before that!
Semester Project Ideas

Some hints for topics

• Procedural shading with noise + marble etc. (GPU Gems 2, chapter 26)
• Procedural shading with noise + bump mapping (GPU Gems 2, chapter 26)
• Subdivision surfaces (GPU Gems 2, chapter 7)
• Ambient occlusion, screen space ambient occlusion
• Shadow mapping, hard shadows, soft shadows
• Deferred shading
• Particle system rendering + CUDA particle sort
• Advanced image filters: fast bilateral filtering, Gaussian kD trees
• PDE solvers (e.g., anisotropic diffusion filtering, 2D level set segmentation, 2D fluid flow)
Semester Project Ideas

Some hints for topics

• Distance field computation (GPU Gems 3, chapter 34)
• Livewire (“intelligent scissors“) in CUDA
• Comparison of parallel sorting algorithms
• Parallel computation of summed area table / multires pyramid, ... use in rendering
• Linear systems solvers, matrix factorization (Cholesky, ...); with/without CUBLAS
• CUDA + matlab
• Fractals (Sierpinski, Koch, ...)
• Image compression

• Look at GPU Gems / GPU Computing Gems books for more ideas!
• “Compute Unified Device Architecture”
• Extensions to C(++) programming language
  • `__host__`, `__global__`, and `__device__` functions
  • Heavily multi-threaded
  • Synchronize threads with `__syncthreads()`, ...
  • Atomic functions
    (before compute capability 2.0 only integer, now also float)

• Compile `.cu` files with NVCC
• Uses general C compiler (Visual C, gcc, ...)
• Link with CUDA run-time (`cudart.lib`) and cuda core (`cuda.lib`)
CUDA Multi-Threading

- CUDA model groups threads into blocks; blocks into grid

- Execution on actual hardware:
  - Block assigned to SM (up to 8 blocks per SM)
  - 32 threads grouped into warp
• Identify work of thread via
  – threadIdx
  – blockIdx

threadIdx Thread Block 0 Thread Block 1 Thread Block N - 1
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7

... float x = input[threadIdx];
    float y = func(x);
    output[threadIdx] = y;
...

blockIdx == 0 blockIdx == 1
CUDA Memory Model and Usage

- `cudaMalloc()`, `cudaFree()`
- `cudaMallocArray()`, `cudaMalloc2DArray()`, `cudaMalloc3DArray()`
- `cudaMemcpy()`
- `cudaMemcpyArray()`
- `Host ↔ host`
  - Host ↔ device
  - Device ↔ device
- Asynchronous transfers
CUDA Software Development

- CUDA Optimized Libraries: math.h, FFT, BLAS, ...
- Integrated CPU + GPU C Source Code
- NVIDIA C Compiler
- NVIDIA Assembly for Computing (PTX)
- CPU Host Code
- Standard C Compiler
- CPU
- GPU
- CUDA Driver
- Profiler
Compiling CUDA Code

C/C++ CUDA Application

NVCC

PTX Code

Virtual

PTX to Target Compiler

G80

Target code

GPU

Physical

CPU Code
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as **kernels**
  - One **kernel** is executed at a time
  - Many threads execute each **kernel**

- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

**Definitions**

- **Device** = GPU
- **Host** = CPU
- **Kernel** = function that runs on the device
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

```c
float x = input[threadID];
float y = func(x);
output[threadID] = y;
...```

```
threadID  0  1  2  3  4  5  6  7
...```
Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to transparently scale to different GPUs
Transparent Scalability

- Hardware is free to schedule thread blocks on any processor
- A kernel scales across parallel multiprocessors
Execution Model

**Software**
- Thread

**Hardware**
- Thread Processor
  - Threads are executed by thread processors
- Multiprocessor
  - Thread blocks are executed on multiprocessors
  - Thread blocks do not migrate
  - Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)

**Grid**
- A kernel is launched as a grid of thread blocks

**Device**
- Only one kernel can execute on a device at one time
CUDA Programming Model

- **Kernel**
  - GPU program that runs on a thread grid

- **Thread hierarchy**
  - Grid: a set of blocks
  - Block: a set of warps
  - Warp: a SIMD group of 32 threads
  - Grid size * block size = total # of threads
CUDA Memory Structure

- Memory hierarchy
  - PC memory: off-card
  - GPU global: off-chip / on-card
  - GPU shared/register/cache: on-chip
- The host can read/write global memory
- Each thread communicates using shared memory
Kernel Memory Access

- **Per-thread**
  - Thread
  - Registers
  - On-chip
  - Local Memory
  - Off-chip, uncached
    - cached on Fermi/Kepler

- **Per-block**
  - Block
  - Shared Memory
  - On-chip, small
  - Fast

- **Per-device**
  - Global Memory
  - Off-chip, large
  - Uncached
  - Persistent across kernel launches
  - Kernel I/O
    - cached on Fermi/Kepler
# Memory Architecture

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in a block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Application</td>
</tr>
</tbody>
</table>

* cached on Fermi/Kepler
Unified memory space can be enabled on Fermi / CUDA 4.x and newer

- **CPU and GPU have separate memory spaces**
- **Host (CPU) code manages device (GPU) memory:**
  - Allocate / free
  - Copy data to and from device
  - Applies to *global* device memory (DRAM)
CUDA Memory Allocation / Release

- `cudaMalloc(void ** pointer, size_t nbytes)`
- `cudaMemset(void * pointer, int value, size_t count)`
- `cudaFree(void* pointer)`

```c
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```
Data Copies

`cudaMemcpy(void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);`

- `direction` specifies locations (host or device) of `src` and `dst`
- Blocks CPU thread: returns after the copy is complete
- Doesn’t start copying until previous CUDA calls complete

`enum cudaMemcpyKind`
- `cudaMemcpyHostToDevice`
- `cudaMemcpyDeviceToHost`
- `cudaMemcpyDeviceToDevice`
Data Movement Example

```c
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
Data Movement Example

```c
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
Data Movement Example

```c
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy(void **) &a_d, nBytes);
    cudaMemcpy(void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);

    for (i=0; i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i ;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
Data Movement Example

```c
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
```
int main(void)
{
    float *a_h, *b_h; // host data
    float *a_d, *b_d; // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMalloc((void **) &a_d, nBytes);
    cudaMalloc((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.0f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
int main(void)
{
    float *a_h, *b_h;  // host data
    float *a_d, *b_d;  // device data
    int N = 14, nBytes, i;

    nBytes = N*sizeof(float);
    a_h = (float *)malloc(nBytes);
    b_h = (float *)malloc(nBytes);
    cudaMemcpy((void **) &a_d, nBytes);
    cudaMemcpy((void **) &b_d, nBytes);

    for (i=0, i<N; i++) a_h[i] = 100.0f + i;

    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

    for (i=0; i<N; i++) assert( a_h[i] == b_h[i] );
    free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
    return 0;
}
Executing Code on the GPU

- **Kernels are C functions with some restrictions**
  - Cannot access host memory
  - Must have `void` return type
  - No variable number of arguments ("varargs")
  - Not recursive except: `__device__` functions on comp.cap. 2.x
  - No static variables

- **Function arguments** automatically copied from host to device
Function Qualifiers

- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void

- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code

- __host__
  - Function called from host and executed on host (default)
  - __host__ and __device__ qualifiers can be combined to generate both CPU and GPU code
Variable Qualifiers (GPU code)

---

**__device__**
- Stored in global memory (large, high latency, no cache)
- Allocated with `cudaMalloc` (**device** qualifier implied)
- Accessible by all threads
- Lifetime: application

**__shared__**
- Stored in on-chip shared memory (very low latency)
- Specified by execution configuration or at compile time
- Accessible by all threads in the same thread block
- Lifetime: thread block

**Unqualified variables:**
- Scalars and built-in vector types are stored in registers
- What doesn’t fit in registers spills to “local” memory
Launching Kernels

- Modified C function call syntax:

  \[ \text{kernel} \langle\langle \text{dim3} \ dG, \ \text{dim3} \ dB \rangle\rangle(...) \]

- Execution Configuration ("\langle\langle >>=\rangle\rangle")
  - \(dG\) - dimension and size of grid in blocks
    - Two-dimensional: \(x\) and \(y\)
    - Blocks launched in the grid: \(dG.x \times dG.y\)
  - \(dB\) - dimension and size of blocks in threads:
    - Three-dimensional: \(x, y,\) and \(z\)
    - Threads per block: \(dB.x \times dB.y \times dB.z\)
  - Unspecified \text{dim3} fields initialize to 1
CUDA Built-in Device Variables

- All `__global__` and `__device__` functions have access to these automatically defined variables

  - `dim3 gridDim;`
    - Dimensions of the grid in blocks (at most 2D)
  - `dim3 blockDim;`
    - Dimensions of the block in threads
  - `dim3 blockIdx;`
    - Block index within the grid
  - `dim3 threadIdx;`
    - Thread index within the block
Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
  - Map from local thread ID (threadIdx.x) to a global ID which can be used as array indices

\[
\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}
\]
Increment Array Example

CPU program

```c
void inc_cpu(int *a, int N)
{
    int idx;
    for (idx = 0; idx<N; idx++)
        a[idx] = a[idx] + 1;
}

int main()
{
    ...
    inc_cpu(a, N);
}
```

CUDA program

```c
__global__ void inc_gpu(int *a, int N)
{
    int idx = blockIdx.x * blockDim.x
              + threadIdx.x;
    if (idx < N)
        a[idx] = a[idx] + 1;
}

int main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float)blocksize ) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```
Thread Cooperation

- The Missing Piece: threads may need to cooperate

- Thread cooperation is valuable
  - Share results to avoid redundant computation
  - Share memory accesses
    - Drastic bandwidth reduction

- Thread cooperation is a powerful feature of CUDA

- Cooperation between a monolithic array of threads is not scalable
  - Cooperation within smaller **batches** of threads is scalable
Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete

CUDA 4.x:
cudaDeviceSynchronize() and cudaStreamSynchronize()
Host Synchronization Example

// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);

// execute the kernel
inc_gpu<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);

// run independent CPU code
run_cpu_stuff();

// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
Device Runtime Component: Synchronization Function

```c
void __syncthreads();
```

Synchronizes all threads in a block

- Once all threads have reached this point, execution resumes normally
- Used to avoid RAW / WAR / WAW hazards when accessing shared

Allowed in conditional code only if the conditional is uniform across the entire thread block
Synchronization

- Threads in the same block can communicate using shared memory
- No HW global synchronization function yet
- __syncthreads()
  - Barrier for threads only within the current block
- __threadfence()
  - Flushes global memory writes to make them visible to all threads

New sync functions on compute capability 2.x: __syncthreads_count(), __syncthreads_and/or(), __threadfence_block(), __threadfence_system(), …
## Basic Limits (CUDA C Programming Guide, Appendix F.1)

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum y- or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td>3</td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
</tbody>
</table>
Basic Limits (CUDA C Programming Guide, Appendix F.1)

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>Compute Capability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for texture memory</td>
<td></td>
</tr>
</tbody>
</table>
Thank you.