CS 380 - GPU and GPGPU Programming
Lecture 13: GPU Compute APIs 3

Markus Hadwiger, KAUST
Reading Assignments #8, #9 (until Oct 28)

Read (required):

• Programming Massively Parallel Processors book, 3rd edition (!), Chapter 7 (*Parallel Patterns: Convolution*)

• Interpolation for Polygon Texture Mapping and Shading, Paul Heckbert and Henry Moreton
  
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.7886

• MIP-Map Level Selection for Texture Mapping
  
  http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=765326

Read (optional):

• Vulkan Tutorial
  
  https://vulkan-tutorial.com
Quiz #2: Oct 23

Organization

• First 30 min of lecture
• No material (book, notes, ...) allowed

Content of questions

• Lectures (both actual lectures and slides)
• Reading assignments
• Programming assignments (algorithms, methods)
• Solve short practical examples
Semester Project (proposal until next week!)

• Choosing your own topic encouraged! (we can also suggest some topics)
  • Pick something that you think is really cool!
  • Can be completely graphics or completely computation, or both combined
  • Can be built on CS380 frameworks, NVIDIA OpenGL SDK, or CUDA SDK
• Submit short (1-2 pages) project proposal sometime next week
  • Send email or talk briefly with Peter or me before (!) you start writing your proposal to confirm that your plan is a suitable topic
• Submit semester project and report (deadline: Dec 5)
• Present semester project (we will schedule event in final exams week)
Semester Project Ideas (1)

Some ideas 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
• Advanced image de-convolution (e.g., convex L1 optimization)
• PDE solvers (e.g., anisotropic diffusion filtering, 2D level set segmentation, 2D fluid flow)
Some ideas for topics

• Distance field computation (GPU Gems 3, chapter 34)
• Livewire (“intelligent scissors“) segmentation in CUDA
• Linear systems solvers, matrix factorization (Cholesky, ...); with/without CUBLAS
• CUDA + matlab
• Fractals (Sierpinski, Koch, ...)
• Image compression
• Bilateral grid filtering for multichannel images
• Discrete wavelet transforms
• Fast histogram computations
• Terrain rendering from height map images; clipmaps or adaptive tessellation
Launching Kernels

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(...)
```

- Execution Configuration (“<<< >>>”)
  - **dG** - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x * dG.y
  - **dB** - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x * dB.y * dB.z
  - Unspecified **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) to a global ID which can be used as array indices

Grid

```
0  1  2  3  4
0  1  2  3  4
0  1  2  3  4
```

\[
\text{blockIdx.x} \\
\text{blockDim.x = 5} \\
\text{threadIdx.x} \\
\text{blockIdx.x*blockDim.x + threadIdx.x}
\]
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

- `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**

New sync functions from compute capability 2.x: `__syncthreads_count()`, `__syncthreads_and/or()`, `__threadfence_block()`, `__threadfence_system()`, ...

And: Completely new Cooperative Thread Groups API in CUDA 9 and newer!
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 from compute capability 2.x:
  __syncthreads_count(), __syncthreads_and/or(),
  __threadfence_block(), __threadfence_system(), …

And: Completely new Cooperative Thread Groups API in CUDA 9 and newer!
Matrix-Matrix Multiplication

\[ P = MN \]
Programming Model: Square Matrix Multiplication

- $P = M \times N$ of size $\text{WIDTH} \times \text{WIDTH}$
- **Without tiling:**
  - One thread handles one element of $P$
  - $M$ and $N$ are loaded $\text{WIDTH}$ times from global memory
Multiply Using One Thread Block

- One block of threads computes matrix P
  - Each thread computes one element of P
- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block
Matrix Multiplication
Device-Side Kernel Function (cont.)

...  
for (int k = 0; k < M.width; ++k)
{
    float Melement = M.elements[ty * M.pitch + k];
    float Nelement = Nd.elements[k * N.pitch + tx];
    Pvalue += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * blockDim.x + tx] = Pvalue;
}
Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a \((\text{BLOCK\_WIDTH})^2\) sub-matrix (tile) of the result matrix
  - Each has \((\text{BLOCK\_WIDTH})^2\) threads
- Generate a 2D Grid of \((\text{WIDTH}/\text{BLOCK\_WIDTH})^2\) blocks

You still need to put a loop around the kernel call for cases where \text{WIDTH} is greater than Max grid size!
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks
Multiply Using Several Blocks - Idea

- One thread per element of P
- Load sub-blocks of M and N into shared memory
- Each thread reads one element of M and one of N
- Reuse each sub-block for all threads, i.e. for all elements of P
- Outer loop on sub-blocks
Example: Matrix Multiplication (1)

- Copy matrices to device; invoke kernel; copy result matrix back to host

```c
// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc((void**)&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);

    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc((void**)&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);
```
// Allocate C in device memory
Matrix d_C;
d_C.width = d_C.stride = C.width; d_C.height = C.height;
size = C.width * C.height * sizeof(float);
cudaMalloc((void**) &d_C.elements, size);

// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

// Read C from device memory
cudaMemcpy(C.elements, d_C.elements, size,
           cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
Example: Matrix Multiplication (3)

- Multiply matrix block-wise
- Set BLOCK_SIZE for efficient hardware use, e.g., to 16 on cc. 1.x or 16 or 32 on cc. 2.x +

- Maximize parallelism
  - Launch as many threads per block as block elements
  - Each thread fetches one element of block
  - Perform row * column dot products in parallel
Example: Matrix Multiplication (4)

```c
__global__ void MatrixMul( float *matA, float *matB, float *matC, int w )
{
    __shared__ float blockA[ BLOCK_SIZE ][ BLOCK_SIZE ];
    __shared__ float blockB[ BLOCK_SIZE ][ BLOCK_SIZE ];

    int bx = blockIdx.x; int tx = threadIdx.x;
    int by = blockIdx.y; int ty = threadIdx.y;

    int col = bx * BLOCK_SIZE + tx;
    int row = by * BLOCK_SIZE + ty;

    float out = 0.0f;
    for ( int m = 0; m < w / BLOCK_SIZE; m++ ) {
        blockA[ ty ][ tx ] = matA[ row * w + m * BLOCK_SIZE + tx ];
        blockB[ ty ][ tx ] = matB[ col + ( m * BLOCK_SIZE + ty ) * w ];
        __syncthreads();

        for ( int k = 0; k < BLOCK_SIZE; k++ ) {
            out += blockA[ ty ][ k ] * blockB[ k ][ tx ];
        }
        __syncthreads();
    }
    matC[ row * w + col ] = out;
}
```

Caveat: for brevity, this code assumes matrix sizes are a multiple of the block size (either because they really are, or because padding is used; otherwise guard code would need to be added)
Thank you.