Reading Assignment #6 (until Mar. 17)

Read (required):

- Programming Massively Parallel Processors book, Chapter 4 (*CUDA Threads*)

- GLSL book: material needed for programming assignment #3
Next Lecture Dates

Lecture 11: Mar. 10 16:00
Lecture 12: Mar. 11 16:00, seminar room GMSV
Lecture 13: Mar. 12 16:00, seminar room GMSV

Lecture 14: Mar. 17 16:00 (Peter Rautek)

Lecture 15: Mar. 24 16:00
Lecture 16: Mar. 27 16:00
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

blockIdx.x
blockDim.x = 5
threadIdx.x

blockIdx.x*blockDim.x + threadIdx.x

0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
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
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(), …
Matrix-Matrix Multiplication

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

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)
- Without tiling:
  - One thread handles one element of \( P \)
  - \( M \) and \( N \) are loaded \( WIDTH \) 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**

Parallel08 – Memory Access

Hendrik Lensch and Robert Strzodka
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 = N.d.elements[k * N.pitch + tx];
    P.value += Melement * Nelement;
}
// Write the matrix to device memory;
// each thread writes one element
P.elements[ty * blockDim.x + tx] = P.value;
}
Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a (BLOCK_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (BLOCK_WIDTH)^2 threads
- Generate a 2D Grid of (WIDTH/BLOCK_WIDTH)^2 blocks

You still need to put a loop around the kernel call for cases where 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 current NVIDIA hw (or 32 on Fermi)

- 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
__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;
}
Thank you.