<table>
<thead>
<tr>
<th>1</th>
<th>CUDA Shared Memory &amp; Synchronization (K&amp;H Ch5, S&amp;K Ch5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Shared Memory Model example: dot product (S&amp;K Ch5)</td>
</tr>
</tbody>
</table>
CUDA SHMEM & Synchronization

(S&K, Ch5.3, K&H Ch5)
The CUDA Memory Model

- The *kernel* is executed by a batch of threads.
- Threads are organized into a *grid* of thread *blocks*.
- Each thread has its own registers, no other thread can access it.
- The *kernel* uses registers to store private thread data.
- **Shared memory:** allocated to thread blocks - promotes *thread cooperation*.
- **Global memory:** host/threads can read/write.
- **Constant and texture memory:** host/threads read only.
- Threads in same block can share memory.
- Requires synchronization – essentially communication.
- Example: Dot product: \((x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4\)

Source: NVIDIA
Programmer View of CUDA Memories

Each thread can:

- Read/write per-thread registers (1 cycle)
- Read/write per-block shared memory (5 cycles)
- Read/write per-grid global memory (500 cycles)
- Read/only per-grid constant memory (5 cycles with caching)

Source: NVIDIA
## CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>int LocalVar;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong> int SharedVar;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int GlobalVar;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong> int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables without any qualifier reside in a register
- Except per-thread arrays that reside in global memory
- All threads have access to Global Memory

Source: David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al
A Common Programming Strategy

- Global memory resides in device memory (DRAM)
- Perform computation on device by tiling the input data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

Source: David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al
CUDA Shared Memory

Each thread can:
- Compiler creates copy of var for each block launched
- low latency: var lives on GPU not off-chip DRAM
- shared memory is more effective on per-block basis
- All threads on a block have access to memory, so require synchronization to avoid race conditions.
Invocation Example

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

Source: David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al
Shared Memory Model: dot.cu (S&K Ch5)
Vector Dot Product

Dot Product is: $\vec{X} \cdot \vec{Y} = |X| |Y| \cos \theta$

Geometric interpretation: length of the projection of Vector $\vec{X}$ onto Vector $\vec{Y}$

$$\vec{X} \cdot \vec{Y} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$
Shared Memory Model: dot.cu (S&K Ch5)

- Dot product is a good example for shared memory and synchronization.
- Each thread multiplies a pair of vector points (line 10).
- Repeats for its chunk of work (line 11).
- Stores its local sum into shared mem cache entry (line 14).
- Synchronize threads (line 17).
- Reduction (lines 22-27): each thread sums 2 entries.
- Store block data into global arr (line 29).

```c
__global__ void dot(float *a, float *b, float *c) {
    // buffer of shared memory - store sum
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    // each thread computes running sum of product
    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // set the cache values in the shared buffer
    cache[cacheIndex] = temp;

    // synchronize threads in this BLOCK
    __syncthreads();

    // for reductions, threadsPerBlock must be a power of 2
    // because of the following code
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}
```
Reduction Operation

// for reductions, threadsPerBlock must be a power of 2
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex]+=cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

// only need one thread to write to global memory
if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];

Source: NVIDIA
Allocate host memory (lines 11-13)

Allocate device memory (lines 15-18)

Populate host arrays (lines 22-24)

Copy from host to device glob mem (27, 29): only need arrays \( a \) and \( b \)

```c
#include "../common/book.h"
#define imin(a,b) (a<b?a:b)
const int N = 33 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1)/threadsPerBlock );
int main( void ) {
    float *a, *b, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;
    // allocate memory on the cpu side
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c, blocksPerGrid*sizeof(float) ) );
    // fill in the host memory with data
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }
    // copy the arrays 'a' and 'b' to the GPU
    HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float), cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float), cudaMemcpyHostToDevice ) );
```
host launches kernel (line 1)

number of threads and blocks depend on $N$

smallest multiple of the \textit{threadsperblock} that is greater than $N$:
Shared Memory and Threading (2D matrix example)

- Each SM in Maxwell has 64KB shared memory (48KB max per block)
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
  - Can potentially have up to 32 Thread Blocks actively executing
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - The next TILE_WIDTH 32 would lead to 2*32*32*4B = 8KB shared memory usage per thread block, allowing 8 thread blocks active at the same time
- Using 16x16 tiling, we reduce the accesses to the global memory by a factor of 16
  - The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS

Source: David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al