COMP 605: Introduction to Parallel Computing Lecture : CUDA Shared Memory

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 04/25/17
Last Update: 04/25/17

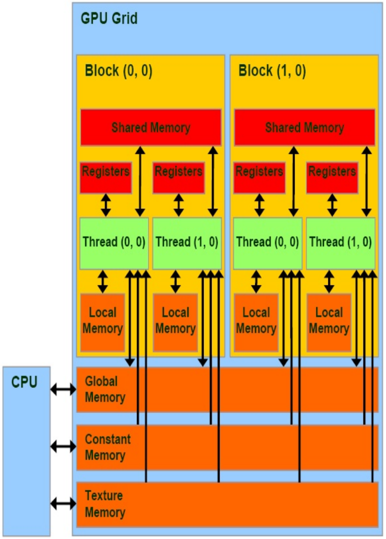
Table of Contents

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: $(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1 y_1 + x_2 y_2 + x_3 y_3 + x_4 y_4$

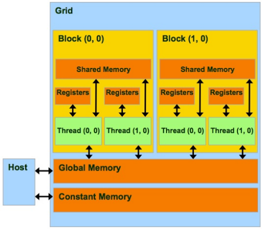


Source: NVIDIA

Programmer View of CUDA Memories

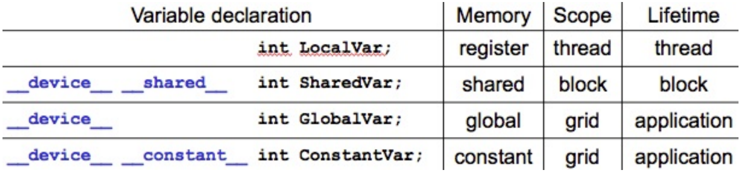
Each thread can:

- 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



Variable declaration	Memory	Scope	Lifetime
`int LocalVar;`	register	thread	thread
`__device__ __shared__ int SharedVar;`	shared	block	block
`__device__ int GlobalVar;`	global	grid	application
`__device__ __constant__ int ConstantVar;`	constant	grid	application

- __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

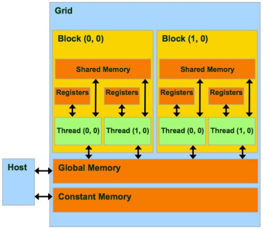
- Global memory resides in device memory (DRAM)
- Perform computation on device by tiling the input data to take advantage of fast shared memory:
    - Partition data into subsets that fit into shared memory
    - Handle each data subset with one thread block:
        - Loading the subset from global 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.



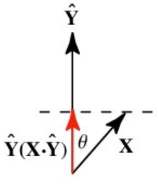
Invocation Example

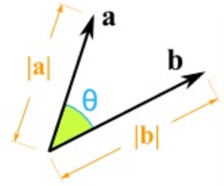
```
__global__ void MatrixMulKernel(float* M, float* N,
                                float* P, int Width) {

    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```

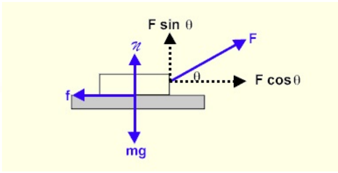
Source: David Kirk/NVIDIA and Wen-mei W. Hwu, ECE408/CS483/ECE498al

Shared Memory Model: dot.cu (S&K Ch5)

Vector Dot Product







Dot Product is: $\vec{X} \bullet \vec{Y} = |X|\,|Y|\, cos\theta$
Geometric interpretation: length of the projection of Vector $\vec{X}$ onto Vector $\vec{Y}$ $\vec{X} \bullet \vec{Y} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$

Shared Memory Model: dot.cu (S&K Ch5)

- Dot product is 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)

```
 1  __global__ void dot( float *a, float *b, float *c ) {
 2   // buffer of shared memory - store sum
 3   __shared__ float cache[threadsPerBlock];
 4      int tid = threadIdx.x + blockIdx.x * blockDim.x;
 5      int cacheIndex = threadIdx.x;
 6
 7   // each thread computes running sum of product
 8      float   temp = 0;
 9          while (tid < N) {
10          temp += a[tid] * b[tid];
11          tid += blockDim.x * gridDim.x;
12      }
13      // set the cache values in the shared buffer
14      cache[cacheIndex] = temp;
15
16  // synchronize threads in this BLOCK
17     __syncthreads();
18
19     // for reductions, threadsPerBlock must be a power of 2
20     // because of the following code
21     int i = blockDim.x/2;
22     while (i != 0) {
23          if (cacheIndex < i)
24              cache[cacheIndex] += cache[cacheIndex + i];
25          __syncthreads();
26          i /= 2;
27      }
28
29      if (cacheIndex == 0)
30          c[blockIdx.x] = cache[0];
31  }
32
```

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];
}
```

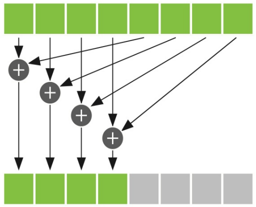
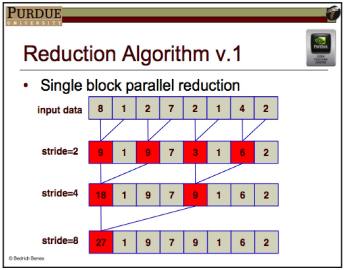


Figure 5.4 One step of a summation reduction



Reduction Algorithm v.1

- Single block parallel reduction

input data	8	1	2	7	2	1	4	2
stride=2	9	1	9	7	3	1	6	2
stride=4	18	1	9	7	9	1	6	2
stride=8	27	1	9	7	9	1	6	2

© Bedrich Benes

Source: NVIDIA

Shared Memory Model: dot.cu (S&K Ch5)

- 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*

```
 1  #include "../common/book.h"
 2  #define imin(a,b) (a<b?a:b)
 3  const int N = 33 * 1024;
 4  const int threadsPerBlock = 256;
 5  const int blocksPerGrid =
 6       imin( 32, (N+threadsPerBlock-1)/threadsPerBlock );
 7  int main( void ) {
 8      float   *a, *b, c, *partial_c;
 9      float   *dev_a, *dev_b, *dev_partial_c;
10      // allocate memory on the cpu side
11      a = (float*)malloc( N*sizeof(float) );
12      b = (float*)malloc( N*sizeof(float) );
13      partial_c = (float*)malloc(blocksPerGrid*sizeof(float));
14      // allocate the memory on the GPU
15      HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
16                                N*sizeof(float) ) );
17      HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
18                                N*sizeof(float) ) );
19      HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
20                                blocksPerGrid*sizeof(float) ) );
21      // fill in the host memory with data
22      for (int i=0; i<N; i++) {
23          a[i] = i;
24          b[i] = i*2;     }
25
26     // copy the arrays 'a' and 'b' to the GPU
27      HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
28                                   cudaMemcpyHostToDevice ) );
29      HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
30                                   cudaMemcpyHostToDevice ) );
```

Shared Memory Model: dot.cu (S&K Ch5)

- host launches kernel (line 1)
- number of threads and blocks depend on N
- smallest multiple of *threadsperblock* that is greater than N:

```
1   dot<<<blocksPerGrid,threadsPerBlock>>>(
2                   dev_a, dev_b,dev_partial_c );
3
4   // copy the array 'c' back from the GPU to the CPU
5   HANDLE_ERROR( cudaMemcpy( partial_c,
6            dev_partial_c, blocksPerGrid*sizeof(float),
7           cudaMemcpyDeviceToHost ) );
8
9   // finish up on the CPU side
10  c = 0;
11  for (int i=0; i<blocksPerGrid; i++) {
12          c += partial_c[i];
13  }
14
15  #define sum_squares(x)  (x*(x+1)*(2*x+1)/6)
16  printf( "Does GPU value %.6g = %.6g?\n", c,
17               2 * sum_squares( (float)(N - 1) ) );
18
19  // free memory on the gpu side
20  HANDLE_ERROR( cudaFree( dev_a ) );
21  HANDLE_ERROR( cudaFree( dev_b ) );
22  HANDLE_ERROR( cudaFree( dev_partial_c ) );
23
24  // free memory on the cpu side
25  free( a );
26  free( b );
27  free( partial_c );
```

```
const int blocksPerGrid =  imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

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