Hardware/Software Co-Design

Lecture 5: CUDA Memories

Miaoqing Huang
University of Arkansas
Fall 2011
1. CUDA Memories
2. Atomic Operations
Hardware Implementation of CUDA Memories

Each thread can:

- Read/write per-thread registers
- Read/write per-thread local memory
- Read/write per-block shared memory
- Read/write per-grid global memory
- Read only per-grid constant memory
### CUDA Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Scope</th>
<th>Life Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>int var;</td>
<td>register</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td>int array_var[10];</td>
<td>local</td>
<td>thread</td>
<td>thread</td>
</tr>
<tr>
<td><strong>shared</strong> int shared_var;</td>
<td>shared</td>
<td>block</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong> int global_var;</td>
<td>global</td>
<td>grid</td>
<td>application</td>
</tr>
<tr>
<td><strong>constant</strong> int constant_var;</td>
<td>constant</td>
<td>grid</td>
<td>application</td>
</tr>
</tbody>
</table>

- “automatic” scalar variables without qualifier reside in registers
  - compiler will spill to thread local memory
    - thread local memory physically resides in global memory
- “automatic” array variables without qualifier reside in thread-local memory
CUDA Memory Types

- **Thread Block**
  - Per-block shared memory

- **Thread**
  - Per-thread local memory

- **Grid 0**
  - Block (0, 0)
  - Block (1, 0)
  - Block (2, 0)
  - Block (0, 1)
  - Block (1, 1)
  - Block (2, 1)

- **Grid 1**
  - Block (0, 0)
  - Block (1, 0)
  - Block (0, 1)
  - Block (1, 1)
  - Block (0, 2)
  - Block (1, 2)

- **Global memory**
### CUDA Variable Type Performance

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Memory</th>
<th>Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int var;</code></td>
<td>register</td>
<td>1×</td>
</tr>
<tr>
<td><code>int array_var[10];</code></td>
<td>local</td>
<td>100×</td>
</tr>
<tr>
<td><code>__shared__ int shared_var;</code></td>
<td>shared</td>
<td>1×</td>
</tr>
<tr>
<td><code>__device__ int global_var;</code></td>
<td>global</td>
<td>100×</td>
</tr>
<tr>
<td><code>__constant__ int constant_var;</code></td>
<td>constant</td>
<td>1×</td>
</tr>
</tbody>
</table>

- Scalar variables reside in fast, on-chip registers
- Shared variables reside in fast, on-chip memories
- Thread-local arrays & global variables reside in uncached off-chip memory
- Constant variables reside in cached off-chip memory
### CUDA Variable Type Scale

<table>
<thead>
<tr>
<th>Variable Declaration</th>
<th>Instances</th>
<th>Visibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int var;</code></td>
<td>100,000s</td>
<td>1</td>
</tr>
<tr>
<td><code>int array_var[10];</code></td>
<td>100,000s</td>
<td>1</td>
</tr>
<tr>
<td><code>__shared__ int shared_var;</code></td>
<td>100s</td>
<td>1,000s</td>
</tr>
<tr>
<td><code>__device__ int global_var;</code></td>
<td>1</td>
<td>100,000s</td>
</tr>
<tr>
<td><code>__constant__ int constant_var;</code></td>
<td>1</td>
<td>100,000s</td>
</tr>
</tbody>
</table>

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 1Ks of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads
CUDA Memories
Atomic Operations

Where to declare variables?

Can host access it?

Yes

Outside of any function

__constant__ int constant_var;
__device__ int global_var;
int var;
int array_var[10];
__shared__ int shared_var;

No

In the kernel

int var;
int array_var[10];
__shared__ int shared_var;
```
// motivate per-thread variables with
// Ten Nearest Neighbors application
__global__ void ten_nn(float2 *result, float2 *ps, float2 *qs,
                        size_t num_qs)
{
    // p goes in a register
    float p = ps[threadIdx.x];

    // per-thread heap goes in off-chip memory
    float heap[10];

    // read through num_qs points, maintaining
    // the nearest 10 qs to p in the heap
    ...

    // write out the contents of heap to result
    ...
}
```
Examples – shared variables

// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]

__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // each thread loads two elements from global memory
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // what are the bandwidth requirements of this kernel?
        int x_i = input[i]; //Two loads
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
Examples – shared variables

// motivate shared variables with
// Adjacent Difference application:
// compute result[i] = input[i] - input[i-1]
__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // How many times does this kernel load input[i]?
        int x_i = input[i];           // once by thread i
        int x_i_minus_one = input[i-1]; // again by thread i+1

        result[i] = x_i - x_i_minus_one;
    }
}
// motivate shared variables with
// Adjacent Difference application:
// compute \( \text{result}[i] = \text{input}[i] - \text{input}[i-1] \)

__global__ void adj_diff_naive(int *result, int *input)
{
    // compute this thread’s global index
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;

    if(i > 0)
    {
        // Idea: eliminate redundancy by sharing data
        int x_i = input[i];
        int x_i_minus_one = input[i-1];

        result[i] = x_i - x_i_minus_one;
    }
}
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
    // shorthand for threadIdx.x
    int tx = threadIdx.x;
    // allocate a __shared__ array, one element per thread
    __shared__ int s_data[BLOCK_SIZE];
    // each thread reads one element to s_data
    unsigned int i = blockDim.x * blockIdx.x + tx;
    s_data[tx] = input[i];

    // avoid race condition: ensure all loads
    // complete before continuing
    __syncthreads();
    ...
    if (tx > 0) {
        result[i] = s_data[tx] - s_data[tx-1];
    }
}
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input)
{
    // shorthand for threadIdx.x
    int tx = threadIdx.x;
    // allocate a __shared__ array, one element per thread
    __shared__ int s_data[BLOCK_SIZE];
    // each thread reads one element to s_data
    unsigned int i = blockDim.x * blockIdx.x + tx;
    s_data[tx] = input[i];

    // avoid race condition: ensure all loads
    // complete before continuing
    __syncthreads();
    ...
    if (tx > 0) {
        result[i] = s_data[tx] - s_data[tx-1];
    }
    else if (i > 0) {
        // handle thread block boundary
        result[i] = s_data[tx] - input[i-1];
    }
}
Examples – shared memory

```c
// when the size of the array isn’t known at compile time...
__global__ void adj_diff(int *result, int *input)
{
    // use extern to indicate a __shared__ array that will be
    // allocated dynamically at kernel launch time
    extern __shared__ int s_data[];
    ...
}

// pass the size of the per-block array, in bytes, as the third
// argument to the triple chevrons
adj_diff<<<num_blocks, block_size, block_size*sizeof(int)>>>(r,i);
```

- Dynamic allocate the size of the shared array during the kernel launch time
Examples – global memory and constant memory

- `cudaMalloc()` and `cudaMemcpy()`
- `cudaMemcpyToSymbol()` and `cudaMemcpyFromSymbol()`

```c
// outside any function
__constant__ float constData[256];

// host side
float data[256];

// host side
cudaMemcpyToSymbol(constData, data, 256*sizeof(float));
cudaMemcpyFromSymbol(data, constData, 256*sizeof(float));
```
Yes, you can use them!
You can point at any memory space per se

```c
__device__ int my_global_variable;
__constant__ int my_constant_variable = 13;

__global__ void foo(void)
{
    __shared__ int my_shared_variable;

    int *ptr_to_global = &my_global_variable;
    const int *ptr_to_constant = &my_constant_variable;
    int *ptr_to_shared = &my_shared_variable;
    ...
    *ptr_to_global = *ptr_to_shared;
}
```
CUDA Memories

Atomic Operations

A Common Programming Strategy

- Global memory resides in device memory (DRAM)
  - Much slower access than shared memory

- **Tile data** to take advantage of fast shared memory
  - Divide and conquer

- Carefully partition data according to access patterns
  - Read-only $\rightarrow$ **_constant_** memory (fast)
  - R/W & shared within block $\rightarrow$ **_shared_** memory (fast)
  - R/W within each thread $\rightarrow$ registers (fast)
  - Indexed R/W within each thread $\rightarrow$ local memory (slow)
  - R/W inputs/results $\rightarrow$ `cudaMalloc`’ed global memory (slow)
1. Partition data into subsets that fit into shared memory
2. Handle each data subset with one thread block
3. Load the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
Perform the computation on the subset from shared memory
A Common Programming Strategy – Demo

5 Copy the result from shared memory back to global memory
Outline

1. CUDA Memories
2. Atomic Operations
Race Conditions

```c
//all items in vector are initialized to zeros
__device__ int vector[NUMBER_VECTOR];
__global__ void race(void)
{
    ...
    a = func_a();
    vector[0] += a;
}
```
Race Conditions

//all items in vector are initialized to zeros
__device__ int vector[NUMBERVECTOR];
__global__ void race(void)
{
    ...
    a=func_a();
    vector[0] += a;
}

threadId:0
    ...
    vector[0] += 5;

threadId:1917
    ...
    vector[0] += 1;

- What is the value of vector[0] in thread 0?
- What is the value of vector[0] in thread 1917?
Race Conditions

```cuda
// all items in vector are initialized to zeros
__device__ int vector[NUMBER_VECTOR];
__global__ void race(void)
{
    ...
    a = func_a();
    vector[0] += a;
}
```

<table>
<thead>
<tr>
<th>threadId:0</th>
<th>threadId:1917</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>vector[0] += 5;</td>
<td>vector[0] += 1;</td>
</tr>
</tbody>
</table>

- What is the value of `vector[0]` in thread 0?
- What is the value of `vector[0]` in thread 1917?
- Answer: not defined by the programming model, can be arbitrary
  - Thread 0 could have finished execution before thread 1917 started
  - Or the other way around
  - Or both are executing at the same time
Race Conditions

// all items in vector are initialized to zeros
__device__ int vector[NUMBER_VECTOR];
__global__ void race(void)
{
  ...
  a=func_a();
  vector[0] += a;
}

threadId:0
  ...
  vector[0] += 5;

threadId:1917
  ...
  vector[0] += 1;

- What is the value of vector[0] in thread 0?
- What is the value of vector[0] in thread 1917?
- Answer: not defined by the programming model, can be arbitrary
  - Thread 0 could have finished execution before thread 1917 started
  - Or the other way around
  - Or both are executing at the same time
- CUDA provides atomic operations to deal with this problem
Atomics

- An atomic operation guarantees that only a single thread has access to a piece of memory until an operation completes.
- The name atomic comes from the fact that it is uninterruptable.
- No dropped data, but ordering is still arbitrary.
- Different types of atomic instructions:
  - \texttt{atomic}\{Add, Sub, Exch, Min, Max, Inc, Dec, CAS, And, Or, Xor\}
- More types in Fermi.
Example: Histogram

```c
// Determine frequency of colors in a picture;
// Colors have already been converted into ints;
// Each thread looks at one pixel and increments a counter atomically;

__global__ void histogram(int* color, int* buckets)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

- Atomics are slower than normal load/store
- You can have the whole machine queuing on a single location in memory
- Atomics unavailable on G80!
Example: Global Min/Max

// Naive solution
// If you require the maximum across all threads in a grid, you could
// do it with a single global maximum value, but it will be VERY slow

__global__ void global_max(int* values, int* gl_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(gl_max,val); // return the previous max
}
Example: Global Min/Max

// Naive solution
// If you require the maximum across all threads in a grid, you could
// do it with a single global maximum value, but it will be VERY slow
__global__ void global_max(int* values, int* gl_max)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int val = values[i];
    atomicMax(gl_max, val); // return the previous max
}

// Better solution
// Introduce intermediate maximum results, so that
// most threads do not try to update the global max
__global__ void global_max(int* values, int* gl_max,
                          int *reg_max, int num_regions)
{
    // i and val as before ...
    int region = i % num_regions;
    if (atomicMax(&reg_max[region], val) < val) {
        atomicMax(gl_max, val);
    }
}
Inspiration from Global Min/Max

- Single value causes serial bottleneck
  - Create hierarchy of values for more parallelism
- Performance will still be slow, so use atomic operations judiciously
  - Cannot use normal load/store for inter-thread communication because of race conditions
  - Use atomic instructions for sparse and/or unpredictable global communication
  - Decompose data (very limited use of single global sum/max/min/etc.) for more parallelism
Atomic operations are not cheap!

They imply serialized access to a variable
Atomic operations are not cheap!

They imply serialized access to a variable

```c
__global__ void sum(int *input, int *result)
{
    atomicAdd(result, input[threadIdx.x]);
}
...
// how many threads will contend
// for exclusive access to result?
sum<<<1,N>>>(input, result);
```
Hierarchical Summation

- **Divide & Conquer**
  - Per-thread `atomicAdd` to a __shared__ partial sum
  - Per-block `atomicAdd` to the total sum
Hierarchical Summation

```c
__global__ void sum(int *input, int *result)
{
  __shared__ int partial_sum;

  // thread 0 is responsible for initializing partial_sum
  if(threadIdx.x == 0) {
    partial_sum = 0;
  }
  __syncthreads();

  // each thread updates the partial sum
  int tx = threadIdx.x;
  atomicAdd(&partial_sum, input[tx]);
  __syncthreads();

  // thread 0 updates the total sum
  if(threadIdx.x == 0) {
    atomicAdd(result, partial_sum);
  }
}

// launch the kernel
sum<<<B,N/B>>>(input, result);
```
__global__ void sum(int *input, int *result)
{
    __shared__ int partial_sum;

    // thread 0 is responsible for initializing partial_sum
    if (threadIdx.x == 0) {
        partial_sum = 0;
    }
    __syncthreads();

    // each thread updates the partial sum
    int tx = threadIdx.x;
    atomicAdd(&partial_sum, input[tx]);
    __syncthreads();

    // thread 0 updates the total sum
    if (threadIdx.x == 0) {
        atomicAdd(result, partial_sum);
    }
}

// launch the kernel
sum<<<B,N/B>>>(input,result);
Hierarchical Summation

```c
__global__ void sum(int *input, int *result)
{
    __shared__ int partial_sum;

    // thread 0 is responsible for initializing partial_sum
    if(threadIdx.x == 0) {
        partial_sum = 0;
    }
    __syncthreads();

    // each thread updates the partial sum
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    atomicAdd(&partial_sum, input[tx]);
    __syncthreads();

    // thread 0 updates the total sum
    if(threadIdx.x == 0) {
        atomicAdd(result, partial_sum);
    }
}

// launch the kernel
sum<<<B,N/B>>>(input, result);
```
Use barriers such as `__syncthreads` to wait until `__shared__` data is ready

Prefer barriers to atomics when data access patterns are regular or predictable

Prefer atomics to barriers when data access patterns are sparse or unpredictable

Atomics to `__shared__` variables are much faster than atomics to global variables

Do not synchronize or serialize unnecessarily
Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase throughput

Use `__shared__` memory to eliminate redundant loads from global memory
- Use `__syncthreads` barriers to protect `__shared__` data
- Use atomics if access patterns are sparse or unpredictable

Prefer atomics to barriers when data access patterns are sparse or unpredictable

Atomics to `__shared__` variables are much faster than atomics to global variables

Optimization comes with a development cost

Memory resources ultimately limit parallelism
### Technical Specification of GPU Architectures

<table>
<thead>
<tr>
<th>Technical Specification</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max x- or y-dimension of a grid</td>
<td>65,535</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Max x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
</tr>
<tr>
<td>Max z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of resident blocks per SM</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max number of resident warps per SM</td>
<td>24</td>
<td>32</td>
<td>48</td>
</tr>
<tr>
<td>Max number of resident threads per SM</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
</tr>
<tr>
<td>Number of 32-bit register per SM</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
</tr>
<tr>
<td>Max amount of shared memory per SM</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>