Accelerators and Accelerated Systems

Mark Silberstein

mark@ee.technion.ac.il
GPU memory: scope and consistency
GPU spinlocks, producer/consumer

CPU-GPU shared memory
Types of memories on NVIDIA GPUs

- Global memory
- L2 $/$ shared m
- Registers
- SM1
- SMn
- L1 $/$ shared m
- Constant memory $/$
- Texture memory $/$
Types of memories on NVIDIA GPUs

- Compiler managed
  - Registers
  - SM1
    - L1 $/ shared m
  - SMn
    - L1 $/ shared m
  - L2 $
  - Global memory

- User addressable
  - Constant memory $
  - Texture memory $

- Hardware managed
  - Constant memory $
  - Texture memory $

- User addressable
User-visible memory
Scope

- The **scope** defines which compute elements share access to the memory
- Global: all threads in a kernel, CPU (via DMA or MMIO)
- Shared: all threads in a threadblock
- Registers: one thread
- Local: used by the compiler to spill registers, and for stack. Located in global memory
GPU Memory Consistency
(focus on NVIDIA)

Examples from Alglave et.al
“GPU concurrency: weak behaviors and programming assumptions”
Example

Thread 1:  
*var1=1;
*var2=1;

Thread 2:  
reg1=*var1;
reg2=*var2;
if (reg2==1)
reg1==????;
Example

Thread 1:   Thread 2:

*var1=1;    reg1=*var1;
*var2=1;    reg2=*var2;
   if (reg2==1)
   reg1==????;

• The question is not defined well. We need to define also scope and memory type:

  • T1 and T2 are in the same warp, different warps of the same TB, or different TBs, or different processors
  • *var is in shared memory, global memory or CPU memory
In-GPU memory consistency

- **Weak** memory consistency
  
  ```c
  *var1=1;
  *var2=1;
  reg1=*var1;
  reg2=*var2;
  if (reg2==1)
  reg1==????;
  ```

- The answer is either 0 or 1 – behavior undefined for any scope or memory type

- Must use **fences**
Fences

- Explicit consistency control
- Definition: all writes performed by a thread before fence are guaranteed to be seen by other threads in the fence's scope
  - Scope: threadblock, GPU, GPU+PCIe attached devices

```c
*var1=1;
__threadfence_<scope>
*var2=1;
reg1=*va12;
reg2=*var2;
if (reg2==1)
  reg1==1;
```
The importance of scope

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>*var1=1;</td>
<td>reg1=*va1;</td>
</tr>
<tr>
<td>__threadfence_block()</td>
<td>reg2=*var2;</td>
</tr>
<tr>
<td>*var2=1;</td>
<td>if (reg2==1)</td>
</tr>
<tr>
<td></td>
<td>reg1==1;</td>
</tr>
</tbody>
</table>

**ONLY if T1 & T2 in the same TB**

| *var1=1;                                                               | reg1=*var1;                                                             |
| __threadfence()                                                       | reg2=*var2;                                                            |
| *var2=1;                                                               | if (reg2==1)                                                           |
|                                                                         | reg1==1;                                                               |

**ONLY if T1 & T2 in the same kernel**

| *var1=1;                                                               | reg1=*var1;                                                             |
| __threadfence_system()                                                 | reg2=*var2;                                                            |
| *var2=1;                                                               | if (reg2==1)                                                           |
|                                                                         | reg1==1;                                                               |

**Even if T2 is on another GPU**
How fence works

Thread 1:
*var1=1;
__threadfence()
wait until data propagates
*var2=1;

Thread 2:
while (var2==0);

Does not block other threads!
*var1=1;

print var1
How fence works

Thread 1:
*var1=1;
__threadfence()
wait until data propagates
*var2=1;

Thread 2:
while (var2==0);

Does not block other threads!

Cost depends on the scope

print var1
Preventing weak behavior due to compiler

- Fences “push” writes from caches and prevent reordering by hardware
  but
- Compiler may store results in registers or optimize memory access!

```c
__device__ int var;
....
while (var<10){
    var++;
}
__whatever_fence_you_want_it___wont_help
```

```
T1
while(var!=10);
printf(“done”);
```

Infinite loop
Preventing weak behavior due to compiler

- **volatile** prevents compiler optimizations w.r.t. variable and forces access to memory
  - of any memory type

```c
__device__ volatile int var=0;
while (var<10){
    var++;
}
__threafence_<scope>()
```

```
T1
while(var!=10);
printf("done");
```

```
T2
```
Implicit threadfence()

- When a threadblock exits
  - maybe also on thread exit, unclear
- On kernel exit
- On __syncthreads()
Example: producer-consumer

- Thread in block 2 consumes data produced by thread in block 1

- Idea:

```cpp
__device__ int mailbox;
__device__ int mail;

TB1:
mail=10112;
mailbox=1;

TB2:
while(!mailbox);
consume(mail)
```
What can go wrong?

• Thread in block 2 consumes data produced by thread in block 1

• Idea:

  ```
  __device__ int mailbox;
  __device__ int mail;
  ```

  TB1:
  mail=10112;
  mailbox=1;

  TB2:
  while(!mailbox);
  consume(mail)

Problems?
What can go wrong?

- Thread in block 2 consumes data produced by thread in block 1

- Idea:
  ```
  __device__ int mailbox;
  __device__ int mail;
  
  TB1:
  mail=10112;
  mailbox=1;
  
  TB2:
  while(!mailbox);
  consume(mail)
  ```

- update to mailbox
  - may not be delivered until the thread block exits
  - may be reordered with update to mail
Enforcing ordering and propagation

• Thread in block 2 consumes data produced by thread in block 1

• Idea:

  ```
  __device__ int mailbox;
  __device__ int mail;

  TB1:
  mail=10112;
  __threadfence();
  mailbox=1;
  __threadfence();
  ```

  ```
  TB2:
  while(!mailbox);
  consume(mail)
  ```

This is optional, ensures the data gets written fast
Are we done?

• Thread in block 2 consumes data produced by thread in block 1

• Idea:

```c
__device__ int mailbox;
__device__ int mail;

TB1:
  mail=10112;
  __threadfence();
  mailbox=1;
  __threadfence();

TB2:
  while(!mailbox);
  consume(mail)
```

Problems?
Use volatile to prevent reordering by compiler

- Thread in block 2 consumes data produced by thread in block 1

- Idea:

  ```
  __device__ volatile int mailbox;
  __device__ volatile int mail;
  
  TB1:
  mail=10112;
  __threadfence();
  mailbox=1;
  __threadfence();
  
  TB2:
  while(!mailbox);
  consume(mail)
  ```
Atomic operations

- Problem: count total number of thread blocks
- Idea:

```cpp
__device__ int blocks = 0;
blocks++;
```

Problems?
Atomic operations

- Problem: count total number of thread blocks
- Idea:
  - First thread of each block increments a global variable

    ```
    __device__ int blocks=0;
    if (threadIdx.x == 0) blocks++;
    ```

Problems?
Atomic operations (2)

- Increment MUST be atomic

```
__device__ int blocks=0;
if (threadIdx.x == 0)
    atomicAdd(&blocks, 1);
```
Atomics memory consistency

• All atomics maintain Sequential Consistency, but only **among themselves**

• Example:

```c
while(atomicCAS(mutex, 0, 1)); // lock
shared_var++;
atomicCAS(mutex, 1, 0); // unlock
```

• What is the problem?
Question

```c
__shared__ volatile int var;
atomicInc(&var,1);
```

Compilation error!
no function atomicInc(volatile int*,int).

Why??
How to have the last threadblock do something?

- Example: reduction -- sum 10K numbers
- Idea:
  - Partial sum of 512 numbers in one thread block
  - Last thread block sums the partial sums
result[blockIdx.x] =
    reduce(&input[start], blockDim.x);

if (blockIdx.x == blockDim.x - 1)
    return reduce(result, blockDim.x)

Problems?
Take 2

__device__ int blockCount=0;
result[blockIdx.x]=
    reduce(&input[start],blockDim.x);
if (threadIdx.x == 0 ){
    if (atomicAdd(&blockCount,1)==gridDim.x-1)
        reduce(result,gridDim.x)
}

Problems?
Take 3 (from CUDA SDK)

// PHASE 1: Process all inputs assigned to this block

reduceBlocks<blockSize, nIsPow2>(g_idata, g_odata, n);

// PHASE 2: Last block finished will process all partial sums

if (gridDim.x > 1)
{
    const unsigned int tid = threadIdx.x;
    __shared__ bool amLast;
    extern float __shared__ smem[];

    // wait until all outstanding memory instructions in this thread are finished
    __threadfence();

    // Thread 0 takes a ticket
    if( tid==0 )
    {
        unsigned int ticket = atomicInc(&retirementCount, gridDim.x);
        // If the ticket ID is equal to the number of blocks, we are the last block!
        amLast = (ticket == gridDim.x-1);
    }
    __syncthreads();

    // The last block sums the results of all other blocks
Take 3 (from CUDA SDK)

:// PHASE 1: Process all inputs assigned to this block
//
reduceBlocks<blockSize, nIsPow2>(g_idata, g_odata, n);

// PHASE 2: Last block finished will process all partial sums
//
if (gridDim.x > 1)
{
    const unsigned int tid = threadIdx.x;
    __shared__ bool amLast;
    extern float __shared__ smem[];

    // wait until all outstanding memory instructions in this thread are finished
    __threadfence();

    // Thread 0 takes a ticket
    if (tid==0)
    {
        unsigned int ticket = atomicInc(&retirementCount, gridDim.x);
        // If the ticket ID is equal to the number of blocks, we are the last block!
        amLast = (ticket == gridDim.x-1);
    }
    __syncthreads();

    // The last block sums the results of all other blocks

// Make sure all writes make it BEFORE the semaphore is released
Question: what is wrong here
Used to be an official spinlock in CUDA

bool leaveLoop = false;

while(!leaveLoop) {

    int lockValue = atomicCAS(lockAddr,0,1);

    if(lockValue == 0) {

        leaveLoop = true;

        ACTUAL_CRITICAL_VAR++;

        *lockAddr = 0;
        __threadfence();
    }
}
Question: what is wrong here:

```java
bool leaveLoop = false;
while(!leaveLoop) {
    int lockValue = atomicCAS(lockAddr,0,1);
    if(lockValue == 0) {
        leaveLoop = true;
        ACTUAL_CRITICAL_VAR++;
        *lockAddr = 0;
        __threadfence();
    }
}
```

This can be reordered
Question: what is wrong here: 2

```c
bool leaveLoop = false;

while(!leaveLoop) {
    int lockValue = atomicCAS(lockAddr,0,1);
    if(lockValue == 0) {
        leaveLoop = true;
        ACTUAL_CRITICAL_VAR++;
        __threadfence();
        *lockAddr = 0;
    }
}
```
Question: what is wrong here 2

```c
bool leaveLoop = false;
while(!leaveLoop) {
  int lockValue = atomicCAS(lockAddr,0,1);
  if(lockValue == 0) {
    leaveLoop = true;
    ACTUAL_CRITICAL_VAR++;
    __threadfence();
    atomicExch(lockAddr,0);
  }
}
```

Without atomics
the value
might not be
observable
to other threads
Question: what is wrong here 3

bool leaveLoop = false;

while(!leaveLoop) {

    int lockValue = atomicCAS(lockAddr,0,1);

    if(lockValue == 0) {

        leaveLoop = true;

        __threadfence();

        ACTUAL_CRITICAL_VAR++;

        __threadfence();

        atomicExch(lockAddr,0);

    }

}
Caution!

Spinlocks are deadlock-prone!
Why?
What is the main difference b/w CPU mutex vs. GPU spinlocks?
CPU-GPU memory consistency
How CPU can share memory with GPU (in CUDA)?

#1: shared memory

- `cudaHostRegister()`
- `cudaHostGetDevicePointer()`
Shared memory: weak consistency explicit sync required

- CPU writes:
  - only through volatile vars
  - followed by write barrier
    - __sync_synchronize() -- gcc memory barrier intrinsics
- GPU writes:
  - only through volatile vars
  - followed by write barrier
    - __threadfence_system() in CUDA
- No atomics over PCI!!
How CPU can share memory with GPU (in CUDA)?

#2: DMA – read/write

cudaMemcpyAsync()
DMA-read/write

- CPU reads/writes
  - in a separate CUDA stream
  - officially unsupported while kernel is running, but works in practice (guaranteed not to deadlock now)
  - requires another protocol for synchronization
- GPU must read/write from/to L2
  - *volatile* suffices
  - L1 is not kept coherent with L2
  - L2 coherent with DMA reads/writes
Producer-consumer

- Simple queue
- Producer: q. enqueue(job)
- Consumer: q.dequeue(job)

- Queue implementations: locks or lock-free
- Both require atomic operations
CPU-GPU producer-consumer

• Queue will NOT work across PCIe – no atomics!
CPU-GPU producer-consumer
One-shot shared buffer

• Idea: single-producer/single-producer

• No need for atomics if only each processor updates its own variable
CPU-GPU producer-consumer
One-shot shared buffer

- Idea: single-producer/single-producer
- No need for atomics if only each processor updates its own variable

schematic implementation

```c
struct buffer{
    int tail=0; int head=0; JOBS q[size];
    produce(j){ if (head<size) q[head]=j; head++;}
    consume(*r) if (tail<head)  *r=q[tail]; tail++;}
}
```
CPU-GPU producer-consumer
One-shot shared buffer

- Idea: single-producer/single-producer
- No need for atomics if only each processor updates its own variable

**schematic implementation**

```c
struct buffer{
    int tail=0; int head=0; JOBS q[size];
    produce(j){ if (head<size) q[head]=j; head++;}
    consume(*r) if (tail<head) *r=q[tail]; tail++;}
}
```

Problems?
CPU-GPU producer-consumer

```c
struct buffer{
    int tail=0; int head=0; JOBS q[size];

    CPU produce(j){ if (head<size) q[head]=j; head++;}

    GPU consume(*r) if (tail<head) *r=q[tail]; tail++;}
}
```

CPU: produce(j) if (head<size) q[head]=j; head++;

GPU: consume(*r) if (tail<head) *r=q[tail]; tail++;
CPU-GPU producer-consumer

```
struct buffer{
    int tail=0; int head=0; JOBS q[size];

CPU produce(j){ if (head<size) q[head]=j; head++;}

GPU consume(*r) if (tail<head) *r=q[tail]; tail++;}
}
```

**Good news:** it is safe!

**Bad news:** liveness is not guaranteed without fences
Summary

• Several memory types
  • Global/Shared
• L1/L2 caches available
• **WEAK** memory consistency
Global memory

- Main bulk (2-12 GB)
- Slow (100s cycles), Wide (280-350GB/s)
- Lifetime & scope: application, stack, per-thread local memory
- CUDA declaration: __device__
- Allocation:
  - before kernel: from CPU via cudaMalloc
  - during kernel: from GPU via malloc
- Access from CPU: cudaMemcopy/cudaMemcpyToSymbol
L1 cache/shared memory

- 64 (128)KB of L1/scratchpad memory combined
  - Split can be configured at runtime: 48(96)K/16(32)K
- L1 – hardware cache
  - line = 128 bytes
  - Can be *partially* disabled at compile time
    - Stack/local memory may use L1
Shared memory

- Scratchpad
  - Looks like regular memory, but **fast**: ~1TB/s
  - **User-managed cache** (so not cached)
- Lifetime & Scope: **thread block**
- CUDA declaration: `__shared__`
- Allocation:
  - during kernel: static allocation
  - before kernel: as a part of kernel invocation
- Access from CPU: none
Special purpose read-only memories

- Constant, Read-only
- Texture
Guidelines

● Global memory – standard RAM
● Shared memory – user-managed caching