Distributed and Parallel Computing

Lecture 06

Alan P. Sexton

University of Birmingham

Spring 2016

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread

Most common CUDA programming errors so far

- All malloc, cudaMalloc and cudaMemcpy in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

  ```
  __global__ kern(int *A, int len)
  {
      int i = blockDim.x * blockIdx.x + threadIdx.x;
      if (i > 1 && i < len)
          A[i] += A[i-1];
  }
  ```

- Accessing past the end (or before the beginning) of an allocated array

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations

Most common CUDA programming errors so far

- All malloc, cudaMalloc and cudaMemcpy in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

```
__global__ kern(int *A, int len)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i > 1 && i < len)
        A[i] += A[i-1];
}
```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations
- Confusion in cudaMemcpy: always copy h_A to or from h_A, never to or from h_B

Most common CUDA programming errors so far

- All `malloc`, `cudaMalloc` and `cudaMemcpy` in bytes, all array allocation, indexing in type size (usually 4 byte words)
- Reading kernel code as if it were only going to be executed by a single thread
- Read/Write race conditions e.g. a kernel that is run with multiple threads:

  ```
  __global__ kern(int *A, int len)
  {
      int i = blockDim.x * blockIdx.x + threadIdx.x;
      if (i > 1 && i < len)
          A[i] += A[i-1];
  }
  ```

- Accessing past the end (or before the beginning) of an allocated array
- Omitting necessary barrier synchronisations
- Confusion in cudaMemcpy: always copy `h_A` to or from `h_A`, never to or from `h_B`
- Round-off errors

Coalesced Global Memory Access

Global memory accesses occur in *memory transactions* or *bursts* of size 32, 64 or 128 bytes.

- Each memory transaction takes nearly the same amount of time
- Thus reading or writing 8, 16 or 32 words, assuming those reads or writes are appropriately aligned, take approximately the same amount of time as reading a single word.
- So long as the consecutive threads in a warp read consecutive words, only 1 memory transaction is required.
- If consecutive threads read non-consecutive words, then each read requires a separate memory transaction ⇒ strided access is much worse than consecutive access
- Array of Structs (AoS) vs Struct of Arrays (SoA)
- Specially important for 2- or 3- dimensional arrays

```
struct {int a; int b} X[LEN] ;       // X[0].a = X[0].b
struct {int a[LEN]; int b[LEN]} X ; // X.a[0] = X.b[0]
```

Shared Memory Banks

Shared memory accesses are approximately 2 orders of magnitude faster than global memory accesses

- Shared Memory in GPUs of compute capability 2.0 or better is divided into 32 equally sized banks
- Shared memory is organised so that 32 consecutive memory word accesses are spread over all 32 banks, one word from each
- On devices of compute capability 3.0 or higher, the banks can be configured to be organised by double, instead of single word.
- Simultaneous access (by different threads in the same warp) to different banks can be serviced simultaneously (4 cycles for a read or write)
- Simultaneous access to the same bank must be serialised
- **Exception:** simultaneous read of the same *address* by all threads in the warp can be serviced simultaneously (broadcast)
- **Exception:** simultaneous read of the same address by some number of threads in the warp can be serviced simultaneously (compute capability 2.0+ multicast)

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
    A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
    A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
    A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`
- Atomic operations *serialise* access to the memory location ⇒ limits parallelism

Atomic operations

We frequently need to implement a read-modify-write operation in parallel:

```
A[index] += 1 ;
```

If multiple threads might be trying to do such an operation on the same memory location, then we have to avoid read/write races. There are a number of options:

- Restructure our code using `__syncthreads()` to enforce serialised access to the memory location
- Restructure our code so that different threads collect updates locally until complete and then a single thread collates the results and updates the target memory location.
- We use an *atomic operation* `atomicAdd(&(A[index]),1)`
- Atomic operations *serialise* access to the memory location ⇒ limits parallelism
- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions`