Distributed and Parallel Computing

Lecture 04

Alan P. Sexton

University of Birmingham

Spring 2019

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 `d_A`, never to or from `d_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 `d_A`, never to or from `d_B`
- Round-off errors

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 (consider a warp executing it)

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 $\Rightarrow$ limits parallelism (consider a warp executing it)
- `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#atomic-functions`

Coalesced Global Memory Access

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

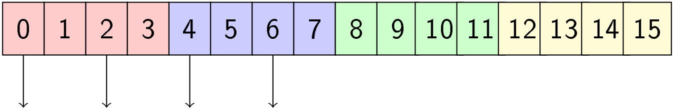
- Each memory transaction takes nearly the same amount of time
- Thus reading or writing 8, 16 or 32 words, assuming those reads or writes are appropriately aligned, take approximately the same amount of time as reading a single word.
- So long as the threads in a warp read a set of consecutive words, 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. Let `i` be the thread id:

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

Global Memory Coalescing — Details

Global memory is partitioned in **burst sections**

- Whenever a location in global memory is accessed, all other locations in the same section are also delivered
- Burst sections can be 128 bytes or more
- When a warp executes a load or store, the number of dram requests issued (and serialised) is the number of different burst sections addressed
- For example: warp size 4, burst size 16 bytes (4 words), stride=2: 2 memory transactions required



- Order of access doesn't matter

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)

Shared Memory Bank Conflicts — Details

Shared memory is stuctured into **banks**

- Modern GPUs have 32 4-byte word banks but can be configured as 32 8-byte double word banks
- The bank used for a word address is the remainder when you divide the word address by the number of banks
- Shared memory can deliver/accept 1 word simultaneously from each bank in a single read/write transaction
- Multiple accesses to the same bank are serialized
- For example: warp size 4, 4 banks, array of 32 words:
  - Warp accesses (00, 01, 02, 03) or (00, 05, 10, 15) in 1 op
  - Warp accesses (00, 02, 04, 06) in 2 ops, (00, 04, 08, 12) in 4

00	01	02	03
04	05	06	07
08	09	10	11
12	13	14	15

Usual Shared Memory Allocation

We have been using one approach to shared memory allocation in our GPU kernels:

- The call to the kernel is executed as follows:

```
kernel1<<<gridDim,blockDim>>>(in,out,len);
```

- The kernel itself is written as:

```
__global__ kernel1(int[] in, int[] out, int len)
{
    __shared__ int XY[BLOCK_SIZE];
    ...
}
```

Usual Shared Memory Allocation

We have been using one approach to shared memory allocation in our GPU kernels:

- The call to the kernel is executed as follows:

```
kernel1<<<gridDim,blockDim>>>(in,out,len);
```

- The kernel itself is written as:

```
__global__ kernel1(int[] in, int[] out, int len)
{
    __shared__ int XY[BLOCK_SIZE];
    ...
}
```

- But `BLOCK_SIZE` needs to be known at **compile time** (i.e. a macro or literal, not a variable)

Alternative Shared Memory Allocation

An alternative approach lets you add an extra *runtime* parameter to the kernel invocation:

- ```
  kernel2<<<gridDim,blockDim,sharedBytes>>>(in,out,len);
  ```

  where the kernel itself now looks like:

  ```
  __global__ kernel1(int[] in, int[] out, int len)
  {
      extern __shared__ int XY[];
      ...
  }
  ```

- This allocates, at kernel invocation time, a certain number of shared bytes for your shared memory data structures
  ```

Alternative Shared Memory Allocation

An alternative approach lets you add an extra *runtime* parameter to the kernel invocation:

- ```
  kernel2<<<gridDim,blockDim,sharedBytes>>>(in,out,len);
  ```

  where the kernel itself now looks like:

  ```
  __global__ kernel1(int[] in, int[] out, int len)
  {
      extern __shared__ int XY[];
      ...
  }
  ```

- This allocates, at kernel invocation time, a certain number of shared bytes for your shared memory data structures
- Advantage: can choose the amount of shared memory per block at runtime
  ```

Alternative Shared Memory Allocation

- You can only specify one block of shared memory per block in the kernel invocation, so if you want multiple shared items dynamically allocated, you have to do something like

```
__global__ kernel2(int[] in, int[] out, int len)
{
    extern __shared__ double data[]
    // if you want d1 to 8 floats long (32 bytes)
    // d2 to be 8 doubles long (64 bytes)
    // d3 to be the rest of the shared memory block

    float *d1 = (float *) &data[0];
    double *d2 = (double *) &data[4];
    int *d2 = (int *) &data[12];

    d1[0] = 1.0f ; d2[0] = 1.0; d3[0] = 1;
    ...
}
```

- Watch out for memory alignment issues!