

Debugging and Optimization strategies

Philip Blakely

Laboratory for Scientific Computing, Cambridge

Outline

1 Correctness

2 Warp-based functions

3 Atomics

4 Debugging

Writing a correct CUDA code

- You should start with a functional CPU/serial code
- If possible, port small parts to GPU at a time
- Codes should give exactly the same results at all times
- If possible, display intermediate results to test individual kernels
- A lot of the difficulty with CUDA is thinking about parallelism in the right way, and working out which threads and blocks should do what work.

- Always make use of error-return codes
- Use `cudaGetLastError()`
- Common errors:
 - Mixing up host and device pointers - use e.g. `hData[]`, `dData[]` naming convention
 - Reversing direction of copying in `cudaMemcpy()`

```
cudaMemcpy(dest, src, numBytes, cudaMemcpyKind)
```
 - Allocating too little memory - `cudaMalloc` needs no. of bytes - use `sizeof()`

```
float3 *a;  
a = cudaMalloc((void **) &a, sizeof(float3) * N);
```
- Most of these will be flagged by `cuda-gdb` or `cuda-memcheck`.

Floating point mismatch

- Moving from CPU code to GPU code *will* lead to changes in floating-point results
- Recall that floating point addition is non-associative
- Upshot: If you are summing in parallel, result may be different from serial result
- Extra FMA (Floating-Multiply-Add) instructions on GPUs can also affect result. See `Floating_Point_on_NVIDIA_GPU.pdf`
- Consider whether you need single-precision or double-precision arithmetic. Former is faster (by at least factor of 8) but may not be accurate enough for scientific computing.
- If any of this leads to stability problems - rethink your algorithm

Indexing errors

- Always check correct calculation of indexes
- Usually `blockIdx.x * blockDim.x + threadIdx.x`
- May have different expressions depending on distribution of data between threads/thread-blocks
- Always check for out-of-bounds access
- It's common to have redundant threads that shouldn't access data - make sure they have an appropriate `if` statement to make them do nothing

Race conditions

If your code occasionally gives errors or gives different results for the same input, you've probably got a race condition, i.e. the final state of code is dependent on order of execution of threads/blocks.

Avoiding race conditions

- Never assume thread blocks are executed in a particular order
- Distinct thread blocks should not depend on each other's results
- Best approach is for a kernel to take data from one global array and compute a result into another array.
- Do not use the same array for input and output.

Race conditions may also emerge when moving to a larger GPU as there are more SMs which can run more thread-blocks simultaneously.

Global memory race conditions

- Usually occur when two or more threads update global memory value independently
- No guarantee in which order they will occur
- No guarantee which will succeed
- General rule: Avoid updating same global memory location from distinct threads
- (Unless you use atomic instructions, in which case you may have performance problems instead.)

Shared memory race conditions

- Occur when distinct threads within a thread-block update single shared memory location independently
- These can be overcome by correct use of `__syncthreads()`
- Ensures that all threads in block have reached same point in kernel
- General advice: When using shared memory, check use of `__syncthreads()`
- It *may* be possible to avoid some of this due to threads in the same warp advancing in lock-step.

Outline

- 1 Correctness
- 2 Warp-based functions**
- 3 Atomics
- 4 Debugging

Warp vote functions

- Recall that a warp of 32 threads is guaranteed to run in lock-step.
- We may need a logical reduction operation (AND/OR) between threads in a warp.
- We could use shared memory for this, but NVIDIA have provided specialised warp-functions that apply to sets of 32 threads only.
- In the following, note that an `unsigned int` is assumed to have 32 bits.

Warp vote functions

The following functions allow communication between all threads in a single warp (32 threads):

```
int __all_sync(unsigned int mask, int predicate);
```

returns non-zero value if and only if **predicate** is non-zero on all active threads for which the relevant bit of **mask** is 1.

(This is a logical AND across all threads in **mask**.)

```
int __any_sync(unsigned int mask, int predicate);
```

returns non-zero value iff **predicate** is non-zero on at least one (active) thread in the warp, for which the relevant bit of **mask** is 1.

(This is a logical OR across all threads in **mask**.)

Warp vote functions ctd.

```
unsigned int __ballot_sync(unsigned int mask, int predicate);
```

returns an integer whose N th bit is set iff `predicate` is non-zero on thread N and bit N of `mask` is set.

```
unsigned int __activemask();
```

returns an integer whose N th bit is set iff thread N is currently active.

Warp shuffle functions

- When implementing a tree-based reduction operation, we may need to communicate more data efficiently between threads in a warp.
- Here, thread N needs to combine its data with that of thread $N + 16$.
- (Then thread N combines data with thread $N + 8$, $N + 4$, etc.)
- For this, we have warp shuffle functions.

Warp shuffle functions

```
T __shfl_up_sync(unsigned int mask, T var, unsigned int delta);
```

returns the value of `var` from the thread `delta` below the current one in the warp. If the current lane is less than `delta`, then `var` is returned unchanged.

The exchange is only carried out for active threads set in `mask`.

Similarly, `__shfl_down_sync` exists to get data from the thread `delta` above the current one. So, a tree reduction to find a maximum could be performed using:

```
var = max(var, __shfl_down_sync(0xFFFFFFFF, var, 16));  
var = max(var, __shfl_down_sync(0xFFFFFFFF, var, 8));  
var = max(var, __shfl_down_sync(0xFFFFFFFF, var, 4));  
var = max(var, __shfl_down_sync(0xFFFFFFFF, var, 2));  
var = max(var, __shfl_down_sync(0xFFFFFFFF, var, 1));
```

Warp shuffle functions

```
T __shfl_sync(unsigned int mask, T var, int src);
```

returns the value of `var` from thread `srcLane`.

The exchange is only carried out for active threads set in `mask`.

This is effectively a broadcast from thread `src` to all other active threads in the warp.

Programming Guide

If applications have warp-synchronous codes, they will need to insert the `__syncwarp()` warp-wide barrier synchronization instruction between any steps where data is exchanged between threads via global or shared memory. Assumptions that code is executed in lockstep or that reads/writes from separate threads are visible across a warp without synchronization are invalid.

Outline

- 1 Correctness
- 2 Warp-based functions
- 3** **Atomics**
- 4 Debugging

- In machine-code, an instruction such as `i += 1` where `i` is in global or shared memory requires three instructions:
 - Read `i` into a register
 - Increment `i`
 - Write `i` back to memorywhich results in a potential race condition.
- CUDA hardware provides atomic instructions that perform some simple operations on shared or global data in one instruction.

Atomic functions

```
T atomicAdd(T* address, T val);
```

adds `val` to the value stored at `address` and returns the original value of `*address`.

T can be any 16-,32-,64-bit type (subject to hardware - see Appendix K).

Other atomic instructions exist: `atomicSub()`, `atomicMax()`, `atomicMin()`, `atomicExch()`, `atomicInc()`, `atomicDec()`, `atomicAnd()`, `atomicOr()`, `atomicXor()`

but there are various restrictions on what types these apply to.

Atomic Compare-And-Swap:

```
T atomicCAS(T* old, T compare, T val);  
// equivalent to  
*old = (*old == compare)? val : *old;
```

Atomic instructions notes

- Example use: Keep a count of how many thread-blocks have completed using `atomicInc`.
- Avoids race condition issue of using global memory as a index - but you still don't know what order updates will occur in
- Note that due to the extra locking overhead, using atomic instructions from hundreds of threads may cause poor performance. However, one atomic instruction per block (for example) should be OK.
- Most atomic operations only available for integer and 32-bit float operations except for `atomicAdd()` for 64-bit floats on Pascal architecture (CC \geq 6.0).
- See the CUDA C Programming Guide for full details.

Outline

- 1 Correctness
- 2 Warp-based functions
- 3 Atomics
- 4 Debugging**

Basic debugging

- Often the easiest form of debugging is `printf` statements
- Output for all `printf` statements encountered will be emitted on host
- For full `print` syntax, use `man printf`:

```
printf("Thread %d v = %lf\n", threadIdx.x, v);
```
- Alternatively, use `assert(i)` which will abort the program if `i` is zero on any thread.
- No guarantee about which order print statements will occur in from different threads

Basic debugging

- The output buffer is of limited size (1MB), and is dumped to screen at kernel launch or at `cudaDeviceSynchronize()`
- If the buffer fills up, further output will overwrite the earliest output
- The buffer size can be increased using `cudaDeviceSetLimit()`
- This feature should therefore only be used as a quick-and-dirty debugging tool.
- Upshot: `printf` may be helpful but absence of output does not imply absence of thread reaching a particular point.

Use of a debugger

- The `cuda-gdb` debugger is an extension of `gdb`
- Need to compile with `-g -G` options.
- Note that `-G` turns off optimization. For code profiling, use `-lineinfo`.
- The debugger allows explicit switching between threads/blocks and checking of values
- Allows stepping through kernels line-by-line
- Not part of the practicals, as I've had problems using it in the past, and it locks up the compute card on which a kernel is being debugged.
- See `/lsc/opt/cuda-11.3/doc/pdf/cuda-gdb.pdf`

In a similar way to `valgrind`, `cuda-memcheck` can check for simple out-of-bounds memory access.

Ill-formed CUDA program

```
__global__ void f(float* a, int N)
{
    int idx = blockDim.x * blockIdx.x + threadIdx.x;

    a[idx] = sinf(idx);
}
```

cuda-memcheck output

```
==== CUDA-MEMCHECK  
==== Invalid __global__ write of size 4  
====       at 0x00000498 in f  
====       by thread (32,0,0) in block (2,0)  
====       Address 0x00201080 is out of bounds  
====  
==== ERROR SUMMARY: 1 error
```

See /lsc/opt/cuda-11.3/doc/pdf/CUDA_Memcheck.pdf

`memcheck` is capable of a large number of checks:

- Out-of-bounds access check
- Malloc/free error-checks
- CUDA API checks (although you should be doing this anyway)
- Memory leaks
- Race condition checks for shared memory (`tool=--racecheck`)
- Uninitialized data usage (`tool=--initcheck`)
- Synchronization errors (`tool=--synccheck`)

Finally, setting the environment variable `CUDA_LAUNCH_BLOCKING=1` will cause CUDA kernels to be performed synchronously.