Debugging CUDA programs

General Interest webinar 2016

Sergey Mashchenko

SHARCNET™

Outline

- Introduction
- DDT debugger
- Examples (live demo)
- Summary: tips and tricks

Common Bugs

- Arithmetic
  - infinities, out of range
- Logic
  - infinite loop
- Syntax
  - wrong operator, arguments
- Resource starvation
  - memory leak
- Parallel
  - race conditions
  - deadlock
- Misuse
  - wrong initial conditions / insufficient checking / variable initialization

Parallel bugs

- In addition to usual, “serial” bugs, parallel programs can have “parallel-only” bugs, such as
  - Race conditions
    - When results depend on specific ordering of commands, which is not enforced
  - Deadlocks
    - When task(s) wait perpetually for a message/signal which never come

Race condition

- Race condition manifests itself as wrong and variable code results. (You get different results every time you run the code, or only for some runs, and when you change the number of threads.)
- As only shared variables are at risk of creating race conditions, use them sparingly (only when truly necessary), and pay a lot of attention to them during debugging.

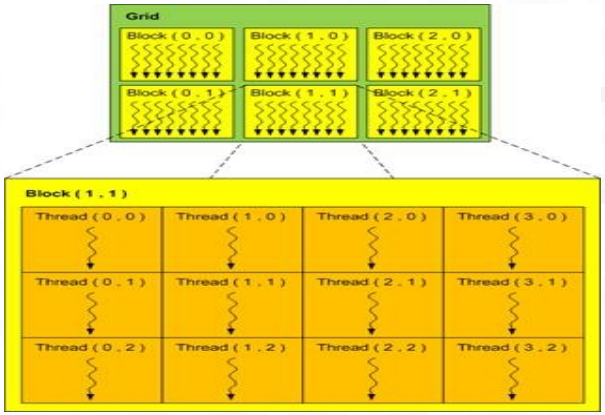
Deadlocks

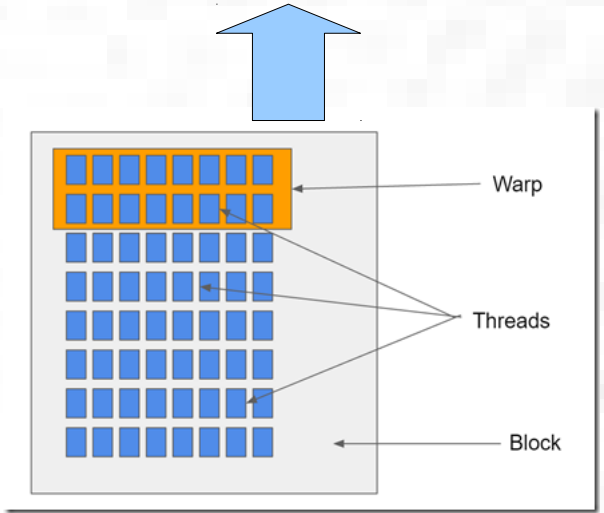
- It happens when ranks (MPI) or threads (OpenMP/CUDA) lock up while waiting on a locked resource that will never become available.
- The sign of a deadlock: the program hangs (always or sometimes) when reaching a certain point in the code.

CUDA bugs

- CUDA is a substantially more complicated parallel platform than say MPI and OpenMP.
- This stems from the complex hierarchical structure of CUDA, which is a mixture of serial, vector, shared memory, and distributed memory models.
- Shared memory levels are prone to race conditions bugs.
- Both shared and distributed memory levels can have deadlock bugs.
- Let's consider these levels in more detail.

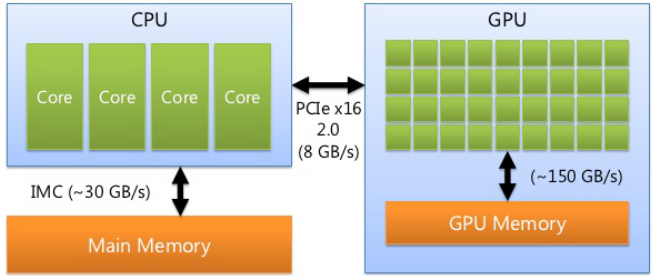
CUDA model





CPU-GPU Relationship

- GPU: coprocessor with its own memory

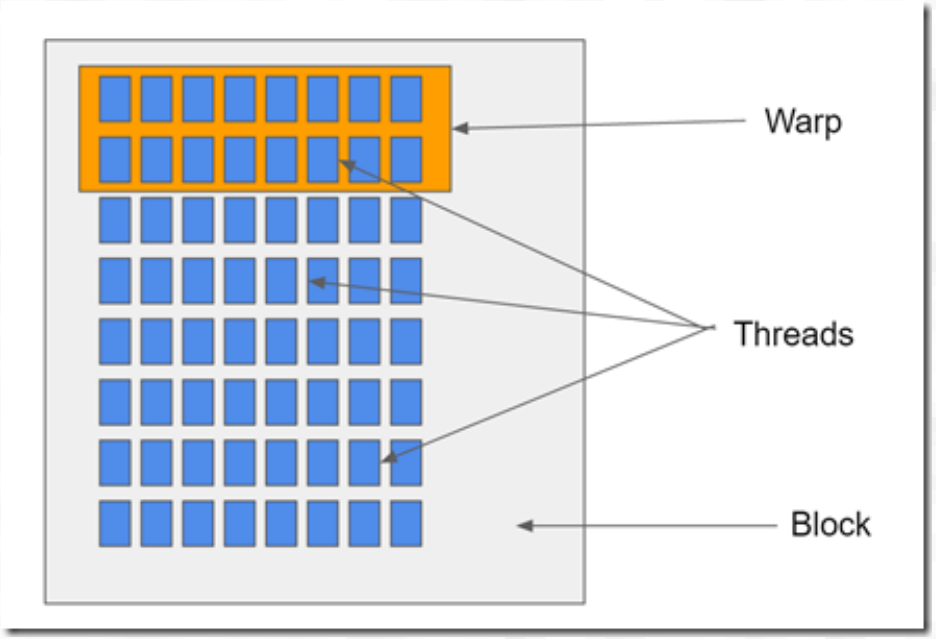


SHARCNET™

Race condition bug

```
__shared__ float A[BSIZE];
int i = threadIdx.x + blockDim.x * blockIdx.x;
A[threadIdx.x] = d_B[i];

// Forgetting to put this will create a bug:
__syncthreads();
// Each thread needs all A elements initialized:
if (threadIdx.x==0) {
   float sum = 0;
   for (int j=0; j<BSIZE; j++)
      sum = sum + A[j];
   }
```



Race condition bug

```
__device__ int d_sum;
__global__ void MyBuggyKernel ()
{
  int block_result;

  /* Computing block_result */
  ...

  if (threadIdx.x == 0)
  // The race condition bug:
     d_sum = d_sum + block_result;
}
```

- The race condition bug is triggered when multiple blocks concurrently read and update the shared variable, d_sum.
- The solution is to protect the updates with `atomicAdd()`.

Dealing with CUDA bugs

- Avoid introducing CUDA bugs in the first place.
  - Follow good CUDA programming practices.
- Catch bugs early by using a proper CUDA error capturing mechanism.
  - E.g. use macro error capture functions ERR() and Is_GPU_present() in ~syam/CUDA_debugging/cuda_errors.h
- Use a parallel debugger like DDT (installed on monk).

Allinea software

- In this webinar, I will focus on advanced parallel debugging tool developed by Allinea and installed on multiple SHARCNET clusters (orca, monk, kraken etc.), DDT. The version installed on monk is CUDA capable.
- For detailed information on how to use DDT on our clusters, check this wiki page:

  https://www.sharcnet.ca/help/index.php/DDT

Using DDT

- DDT can be used interactively on orca and monk development nodes (orc-dev1 ... orc-dev4; mon-dev1).
- Use “-Y” switch with all ssh commands.
- Compile your CUDA code with low or zero optimization (-O0), and use “-g -G” switches with nvcc to add symbolic information.
- module load ddt
- Simply prepend “ddt” in front of your code + command line arguments:

```
$ ddt ./my_cuda_code  arg1  arg2
```

Live Demo

- All examples are on SHARCNET systems, in /home/syam/CUDA_debugging directory.

Summary: tips and tricks

- CUDA debugging is complicated, so use a macro error capturing function with every CUDA function – helps to capture ~1/2 of all bugs.
- Do a query on a presence and capability of the GPU(s) at the beginning of your code: exceeding the GPU resources limits result in nasty (hard to catch) bugs.

Summary: tips and tricks (cont.)

- Each level with shared memory access (all except for the CPU-GPU level) is prone to race condition bugs, which are hard to catch. Pay close attention to all shared variables writes.
- DDT cannot capture race condition bugs caused by the warp execution model. (-G switch apparently causes all threads in a block to execute each line in a kernel, one line after another.)

Summary: tips and tricks (cont.)

- Many CUDA errors (in kernels and asynchronous memory copies) are reported later.
- Always use explicit device/host suffixes/prefixes when naming pointers.
- In CUDA “Symbol” memcopy functions, remember that the device residing variable is provided as a symbol (name), not as a pointer.

The end