Chapter 3.3: Debugging Page Not Present Errors

ROCm Tutorial | AMD 2020

Table of Contents

Chapter 3.3: Debugging Page Not Present Errors

This hands on tutorial shows how we can identify and debug page fault errors which happen due to incorrect memory allocation when using ROCm

Preparation

1. First in the tutorial repository go to the directory

```
cd Chapter3/03_HIP_Page_Fault_Debug
```

2. As noted in the accompanying ppt, this application is part of the GPU stream benchmark suite that has been slightly modified for this tutorial

Compiling and Executing

1. Run make

   Execute ./gpu_stream-hip

2. You will see the program run for a while before it crashes with a page not present error as shown in Figure 1

3. This means some kernel in the application tried to access memory that was not allocated to it

```
GPU-STREAM
Version: 1.0
Implementation: HIP
GridSize: 26214400 work-items
GroupSize: 1024 work-items
Operations/Work-item: 1
Precision: double

Running kernels 10 times
Array size: 200.0 MB (=0.2 GB) 0 bytes padding
Total size: 1000.0 MB (=1.0 GB)
Using HIP device Vega 20 (compute_units=60)
Driver: 4
d_a=0x7fd62a600000
d_b=0x7fd61dc00000
d_c=0x7fd611200000
d_d=0x7fd604800000
d_e=0x7fd5f7e00000
Memory access fault by GPU node-1 (Agent handle: 0x1b818c0) on address 0x7fd5f7cde000. Reason: Page not present or supervisor p
rivilege.
Aborted (core dumped)
```

Figure 1: Page not present error in the add4 benchmark

Kernel Trace Dumping

1. The page not present error means we tried to access something in a kernel which was beyond the memory that was allocated for that data structure
2. The error does not tell us which kernel caused the problem
3. So first we need to identify the fault source
4. To do this we will set the environment variable AMD_LOG_LEVEL=4
    a. This shows a full trace of the API calls that were made by the application
5. Let us re-run:

    AMD_LOG_LEVEL=4 ./gpu-stream-hip

6. You will see an output as shown in Figure 2:

```
:3:rocvirtual.cpp          :2167: 9924687635979 us: [7f4a62f57700]!    ShaderName : _Z5triadIdEvPT_PKS0_S3_

:4:rocvirtual.cpp          :511 : 9924687635988 us: [7f4a62f57700] HWq=0x7f4a62f06000, Dispatch Header = 0x302 (type=2, barrie
r=1, acquire=1, release=0), setup=3, grid=[9049088, 1, 1], workgroup=[1024, 1, 1], private_seg_size=0, group_seg_size=0, kernel
_obj=0x7f484520e5c0, kernarg_address=0x7f4a62e00000, completion_signal=0x7f4a62f7c880
:4:commandqueue.cpp        :161 : 9924687635997 us: command is submitted: 0x1ce2500
:4:command.cpp             :192 : 9924687635944 us: waiting for event 0x1dd9d40 to complete, current status 3
:4:rocvirtual.cpp          :594 : 9924687636008 us: [7f4a62f57700] HWq=0x7f4a62f06000, BarrierAND Header = 0x1503 (type=3, bar
rier=1, acquire=2, release=2), dep_signal=[0x0, 0x0, 0x0, 0x0, 0x0], completion_signal=0x7f4a62dbea80
Memory access fault by GPU node-1 (Agent handle: 0x1d08e10) on address 0x7f4848a26000. Reason: Page not present or supervisor p
rivilege.
Aborted (core dumped)
```

Figure 2: Trace of kernel execution shown using AMD_LOG_LEVEL=4

Locating Fault Source

1. From the API trace, we can see the program faulted after the kernel "_Z5triadIdEvPT_PKS0_S3"
   - The kernel name is mangled
   - So, we will need to find the actual name of the kernel
2. Run " c++filt _Z5triadIdEvPT_PKS0_S3_"
   - This will give us the detangled kernel time which is

     void triad<double>(double*, double const*, double const*)

Understanding the Faulty Array Allocation

1. Looking at the "triad" kernel, we see that the kernel uses three arrays:
   - triad_e, b and c
   - These arrays correspond to triad_e, triad_b and triad_c as can be observed from the kernel launch call:

     hipLaunchKernelGGL(triad<double>, dim3(ARRAY_SIZE_2/groupSize),

     dim3(groupSize), 0, 0, (double*)triad_e, (double*)triad_b, (double*)triad_c);

2. Therefore, we need to find out the faulty array

3. Inside the kernel all these arrays are indexed by “i” which runs from 0 to ARRAY_SIZE_2 - 1
    - This is because each thread is responsible for calculating one element of the output array which is the “triad_e”
4. We want to verify if all our arrays used here have been allocated with “ARRAY_SIZE_2” elements
5. Let us take a look at the allocation code shown in the Figure 3(Line 287 to Line 310)

```
hipMalloc(&triad_b, ARRAY_SIZE_2*DATATYPE_SIZE + ARRAY_PAD_BYTES);
triad_b += ARRAY_PAD_BYTES;
check_cuda_error();
hipMalloc(&triad_c, ARRAY_SIZE_2*DATATYPE_SIZE + ARRAY_PAD_BYTES);
triad_c += ARRAY_PAD_BYTES;
check_cuda_error();
hipMalloc(&triad_e, ARRAY_SIZE*DATATYPE_SIZE + ARRAY_PAD_BYTES);
triad_e += ARRAY_PAD_BYTES;
check_cuda_error();
```

Figure 3: Array Allocation for the faulty kernel

Fixing the Incorrect Array Allocation

1. Ooops. Looks like we allocated triad_b and triad_c with ARRAY_SIZE_2 but triad_e with ARRAY_SIZE

2. The kernel launches threads that operate on elements from 0 to ARRAY_SIZE_2 -1
    - However triad_e has only ARRAY_SIZE elements
    - ARRAY_SIZE < ARRAY_SIZE_2(Line 42 and 43 in common.cpp)
    - Therefore, when a thread tries to access triad_e[i] where i > ARRAY_SIZE it results in a page fault

3. Let us apply the simple fix to correct the allocation in line 308:

```
hipMalloc(&triad_e, ARRAY_SIZE_2*DATATYPE_SIZE + ARRAY_PAD_BYTES);
```

4. Run make and ./hip_stream

5. The program will run without the page fault error