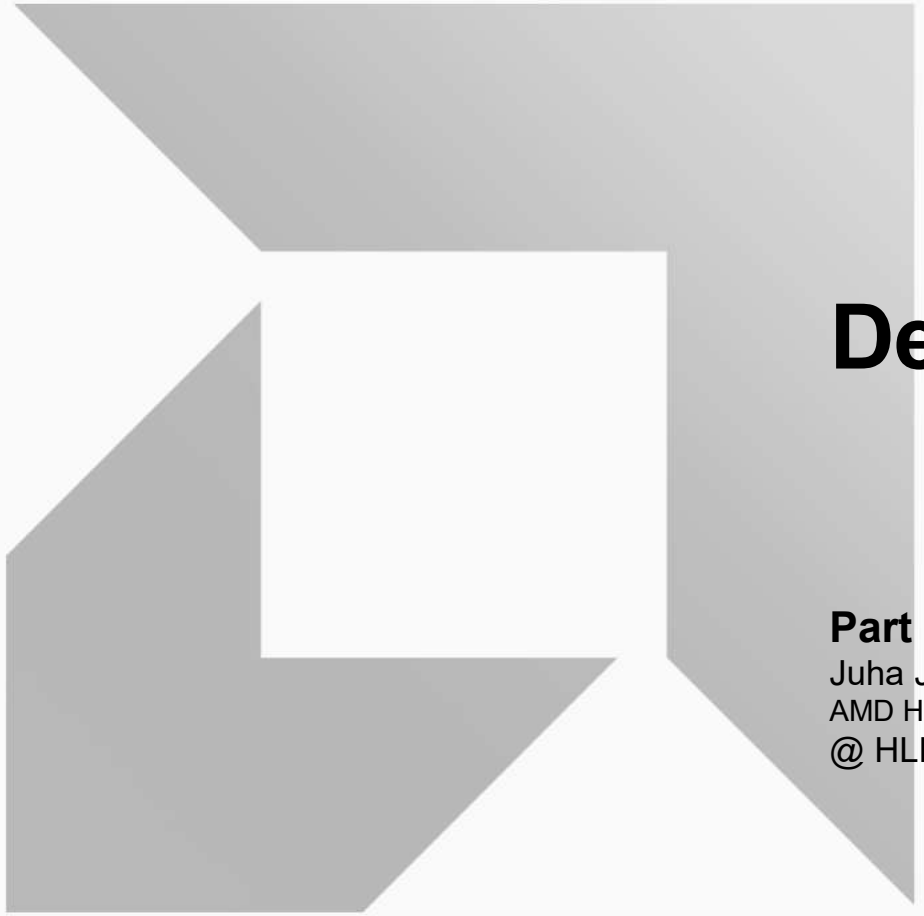


# Debugging with ROCgdb

## Part II - debugging on the GPU

Juha Jäykkä
AMD HPC & Sovereign AI Customer Enablement
@ HLRS, 14 October 2026

AMD
together we advance_

Public



# From CPU threads to GPU waves and lanes

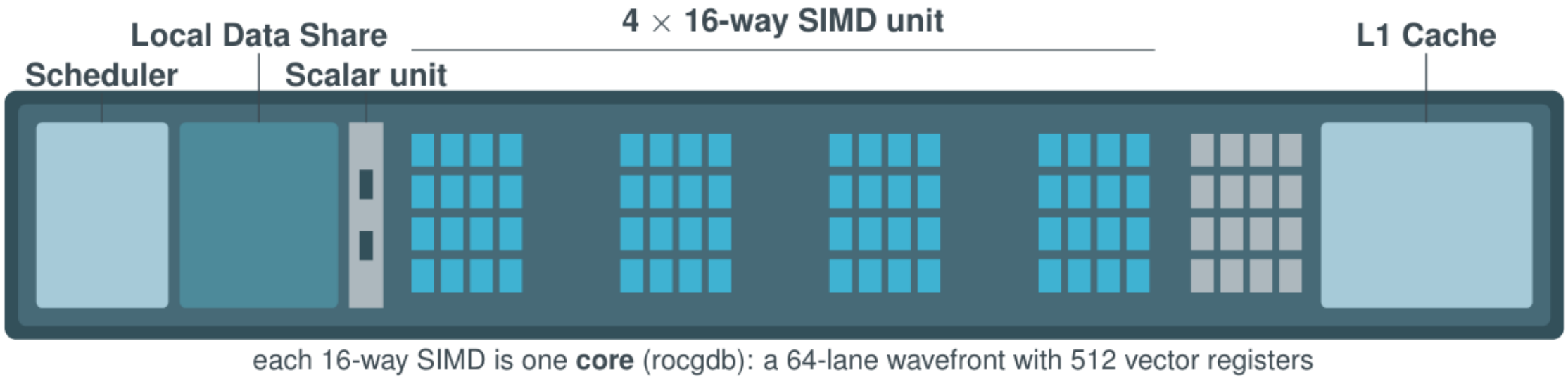
- Part I was all CPU. rocgdb debugs the *device* side with the same commands.
- A heterogeneous program has a *host* (CPU) part and *device* (GPU) kernels.
- rocgdb's vocabulary for the GPU:
  - *agent* = a GPU as the ROCm runtime enumerates it (a whole MI300A APU, or one XCD of it when the APU is partitioned in CPX mode);
  - *queue* = an HSA command queue the host submits work to;
  - *dispatch* = one kernel launch;
  - *thread* = either a CPU thread or a GPU wave
    - a group of lanes run in lock-step
    - lane = one HIP or OpenMP thread (one work-item).
- So a `thread` in rocgdb output is a wave, not a HIP or OpenMP thread.
- The wave's name encodes where it is: `agent:queue:dispatch:wave (wg-x,wg-y,wg-z)/work-group-thread-index`.
- The same workflow spans HIP, OpenMP target offload, and hipstdpar, and C/C++ and Fortran.

Public



# The hardware, from rocgdb's point of view

- A *compute unit* (CU) groups several SIMD units behind a shared scheduler, with a scalar unit, local data share (LDS), and L1 cache. The exact counts are per-architecture - `info agents` reports them for the part you are on.
  - MI300A (CDNA3, gfx942): 4 SIMD units per CU, 64 KB LDS, scalar unit, L1.
- rocgdb treats each SIMD as a "core" with a fixed *vector length* (lanes per wave), room for several resident waves, and a vector-register file. These, too, differ by architecture:
  - lanes per wave: 64 on MI300A and MI355X (CDNA <= 4), 32 on all RDNA generations, MI430X and the other CDNA5 parts.
    - caveat: `rocgdb` support on RDNA tends to be incomplete
  - on MI300A in CPX mode, `info agents` reports 152 such cores and 1216 resident-wave slots (8 per core); each SIMD has 512 vector registers (256 general + 256 matrix-accumulation, a 128 KiB file).
    - CPX mode treats each XCD chiplet as a separate GPU
    - SPX mode is "whole GPU"; multiply the numbers by 6.



Public



# Compile, and the arch flag

| Flag / command | What it does |
|---|---|
| `amdclang++ -x hip -ggdb -O0 --offload-arch=gfx942` | HIP C++ debug build for MI300A (gfx942); other arches: gfx950 MI355X, etc. |
| `amdflang -g -O0 --offload-arch=gfx942` | Fortran debug build (amdflang has no `-ggdb`; `-g` is the Fortran way). |
| `rocgdb --args ./app <args>` | launch under rocgdb. |
| `set amdgpu precise-memory on` | report a memory fault at the exact faulting instruction. |

- for both `amdclang++` and `amdflang` add `-fopenmp` to use OpenMP directives (including offload to GPU)
- `hipcc` is deprecated: compile HIP C++ with `amdclang++ -x hip`, which selects the HIP language and links the runtime (plain `amdclang` is for C).
- Env tips:
  - `OMP_NUM_THREADS=1` (else you see hundreds of host threads)
  - `HSA_XNACK` controls demand paging (Hunter defaults it to 1)
- On Cray systems you can also compile through the wrappers `cc`, `CC`, and `ftn` (C, C++, Fortran), based on loaded modules
  - `craype-accel-*` - adds correct offload target etc flags automatically
  - `PrgEnv-*` - determines the underlying compiler (e.g. `PrgEnv-amd` uses `amdclang`, `PrgEnv-gnu` uses `gcc`)

Public



# Breakpoints in a kernel

- `break <kernel>` is *pending* until the code object loads
  - answer "pending? yes", or
  - `set breakpoint pending on`, or
  - `export HIP_ENABLE_DEFERRED_LOADING=0` loads HIP kernels eagerly.
- Stepping/continue switches between waves; `set scheduler-locking on` (or `tbreak`, or a breakpoint condition) stays on one - but it freezes the others, so it deadlocks if the kernel needs them (a `__syncthreads`, or the host waiting on all); turn it `off` to finish.
- When it fires, all waves halt and rocgdb switches to the *lowest-numbered* thread (`thread 6` below, `with lanes [0-63]` = all 64 lanes live)
- A *function* breakpoint (`break saxmy`) may stop the rest mid-prologue (the runtime code between your host- and device-side source codes)
- a *line* breakpoint (`break saxmy.cpp:21`) catches them all on the line (see "Waves and lanes").
- Left: the `rocgdb ./saxmy` launch + *pending* `break saxmy`; right: the `saxmy` kernel it binds to (line 21)
  - `run` is on the next slide: the next slides continue from here

```
$ rocgdb ./saxmy
[rocgdb informational cruft removed]
Reading symbols from ./saxmy...
(gdb) break saxmy
Breakpoint 1 (saxmy) pending.
```

```
16  __constant__ float a = 1.0f;
17
18  __global__
19  void saxmy(int n, float const* x, int incx, float* y, int incy)
20  {
21      int i = blockDim.x*blockIdx.x + threadIdx.x;
22      if (i < n)
23          y[i] = a*x[i]*y[i];
24  }
```

Public



# System and device state

- `info agents` - the GPUs (agents) and their properties (gfx version, CUs, max waves).
- `info queues` - the HSA queues
- `info dispatches` - the kernel launches (grid, workgroup, kernel).
- `show architecture` - `amdgcn:gfx942` once you are stopped in device code.
- The *dispatch* row maps straight to the launch: `Grid` is the *total* work-items (`num_groups x group_size`), `Workgroup` the per-group size (both as x,y,z). The kernel below launched `<<<2, 128>>>`, hence `[256,1,1]` / `[128,1,1]`.

```
(gdb) run
Thread 6 "saxmy" hit Breakpoint 1, with lanes [0-63], saxmy (n=256, x=0x0, incx=1, y=0x0, incy=1) at saxmy.cpp:21
(gdb) info agents
  Id State Target Id                  Architecture Device Name         Cores Threads
* 1  A     AMDGPU Agent (GPUID 23908) gfx942       AMD Instinct MI300A 152   1216
(gdb) info queues
  Id   Target Id                Type Read Write Size     Address
  1    AMDGPU Queue 1:1 (QID 0) HSA  2    2     4096     0x00007ffff6418000
* 2    AMDGPU Queue 1:2 (QID 1) HSA  1    3     1048576  0x00007fffe1000000
(gdb) info dispatches
  Id   Target Id                      Grid      Workgroup Fence   Kernel Function
* 1    AMDGPU Dispatch 1:2:1 (PKID 1) [256,1,1] [128,1,1] B|As|Rs saxmy(...)
(gdb) show architecture
The target architecture is set to "auto" (currently "amdgcn:gfx942").
```

```
26  int main()
27  {
28      int n = 256;
29      std::size_t size = sizeof(float)*n;
30
31      float* d_x = nullptr;
32      float* d_y = nullptr;
33
34      int num_groups = 2;
35      int group_size = 128;
36      saxmy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
37      hipDeviceSynchronize();
```

Public



# Waves and lanes

- `info threads` lists host threads *and* GPU waves
- `info lanes` the lanes of the current wave.
- `thread <id>` selects a wave; `lane <n>` a lane within it.
- `set scheduler-locking on` isolates one wave when single-stepping (races).
- `info threads` shows four GPU waves (threads):
  - 128 work-items pack into two 64-lane waves, times two work-groups
  - only the *current* wave is at the breakpoint line; the others sit a few prologue instructions behind (here still in the inlined `blockDim.x` accessor) - waves advance independently
  - to catch every wave on the *same* line, set a *line* breakpoint (`break saxmy.cpp:21`) rather than the *function* breakpoint (`break saxmy`), which may stop mid-prologue; or inspect at the fault (example on a later slide)

```
(gdb) info threads
  Id   Target Id                                  Frame
  1    Thread 0x... "saxmy"  WaitRelaxed () from libhsa-runtime64.so
  2    Thread 0x... "saxmy"  __GI___ioctl () at ioctl.c:36
* 6    AMDGPU Wave 1:2:1:1 (0,0,0)/0 "saxmy"  saxmy (...) at saxmy.cpp:21
  7    AMDGPU Wave 1:2:1:2 (0,0,0)/1 "saxmy"  __hip_get_block_dim_x () at amd_hip_runtime.h:258
  8    AMDGPU Wave 1:2:1:3 (1,0,0)/0 "saxmy"  __hip_get_block_dim_x () at amd_hip_runtime.h:258
  9    AMDGPU Wave 1:2:1:4 (1,0,0)/1 "saxmy"  __hip_get_block_dim_x () at amd_hip_runtime.h:258
(gdb) info lanes                               # lanes of the current wave (6)
  Id   State Target Id                                Frame
* 0    A     AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0]   saxmy (...) at saxmy.cpp:21
  1    A     AMDGPU Lane 1:2:1:1/1 (0,0,0)[1,0,0]   saxmy (...) at saxmy.cpp:21
  ...
  63   A     AMDGPU Lane 1:2:1:1/63 (0,0,0)[63,0,0] saxmy (...) at saxmy.cpp:21
```

Public



# Registers

- rocgdb reads the kernel's builtins directly: `print threadIdx.x` is the work-item index (per *lane*); `print blockIdx.x` is the work-group index - the same for every lane *and* wave of a group. Here each 128-lane group is two waves, so `blockIdx.x` is `0` on both waves of group 0 and `1` on both of group 1: it changes per *work-group*, not per *wave* (the two coincide only when a group is a single wave, i.e. group size \le wavefront). Switch with `lane <n>` / `thread <id>`.
- `info registers` / `info reg exec` / `info reg vcc` dump the raw vector and scalar registers, the execute mask, and the condition code.

```
(gdb) lane 0
(gdb) print threadIdx.x
$1 = 0
(gdb) lane 5
(gdb) print threadIdx.x
$2 = 5
(gdb) lane 63
(gdb) print threadIdx.x
$3 = 63
(gdb) info reg exec vcc
exec           0xffffffffffffffff  18446744073709551615
vcc            0xffffffffffffffff  18446744073709551615
```

Public



# Reading the ISA: source and instructions interleaved

- `disassemble /s` prints each *source* line directly above the instructions it generated - no need to use external tools.
- `amd_hip_runtime.h` lines are the inlined `blockDim.x` accessor and arg base-pointer loads; `/s` reprints the `file:` header on each switch. Addresses shown as offsets.

```
(gdb) disassemble /s
Dump of assembler code for _Z5saxmyiPKfiPfi:
amd_hip_runtime.h:
258  __DEVICE__ unsigned int ...
=> <+0>:   s_load_dword s3, s[0:1], 0x34
   <+8>:   s_load_dword s4, s[0:1], 0x0
   <+16>:  s_waitcnt lgkmcnt(0)
   <+20>:  s_and_b32 s3, s3, 0xffff

saxmy.cpp:
21     int i = blockDim.x*blockIdx.x + threadIdx.x;
   <+28>:  s_mul_i32 s2, s2, s3
   <+32>:  v_add_u32_e32 v0, s2, v0

22     if (i < n)
   <+36>:  v_cmp_gt_i32_e32 vcc, s4, v0
   <+40>:  s_and_saveexec_b64 s[2:3], vcc
   <+44>:  s_cbranch_execz 29  # +164

amd_hip_runtime.h:
258  __DEVICE__ unsigned int ...
   <+48>:  s_load_dwordx2 s[2:3], s[0:1], 0x8
   <+56>:  s_load_dwordx2 s[4:5], s[0:1], 0x18
```

```
saxmy.cpp:
23         y[i] = a*x[i]*y[i];
   <+64>:  v_ashrrev_i32_e32 v1, 31, v0
   <+68>:  v_lshlrev_b64 v[0:1], 2, v[0:1]
   <+76>:  s_getpc_b64 s[0:1]
   <+80>:  s_add_u32 s0, s0, 0xffffeeb0
   <+88>:  s_addc_u32 s1, s1, lit(0xffffffff)
   <+96>:  s_waitcnt lgkmcnt(0)
   <+100>: v_lshl_add_u64 v[2:3], s[2:3], 0, v[0:1]
   <+108>: global_load_dword v2, v[2:3], off
   <+116>: v_lshl_add_u64 v[0:1], s[4:5], 0, v[0:1]
   <+124>: global_load_dword v3, v[0:1], off
   <+132>: s_load_dword s0, s[0:1], 0x0
   <+140>: s_waitcnt vmcnt(1) lgkmcnt(0)
   <+144>: v_mul_f32_e32 v2, s0, v2
   <+148>: s_waitcnt vmcnt(0)
   <+152>: v_mul_f32_e32 v2, v2, v3
   <+156>: global_store_dword v[0:1], v2, off

24     }
   <+164>: s_endpgm
End of assembler dump.
```

Public



# Source and ISA together: the TUI

- `rocgdb --tui` with `layout split` stacks the *source* (top) and *ISA* (bottom) in one screen; the current source line (`>`) and the current instruction (`B+>`) are both marked. The panes do *not* interleave row-by-row the way `disassemble /s` does so for the per-line source-to-asm map use `/s` (previous slide). `focus src` / `focus asm` switch panes.
- `stepi` is very useful in this layout: watch `B+>` walk the ISA one instruction at a time, with the `>` source line tracking alongside - the live counterpart to the static `/s` dump of the previous slide.

```
┌─saxmy.cpp──────────────────────────────────────────────────────┐
│   18 __global__                                                │
│   19 void saxmy(int n, float const* x, int incx, float* y,     │
│   20            int incy) {                                    │
│ > 21     int i = blockDim.x*blockIdx.x + threadIdx.x;          │
│   22     if (i < n)                                            │
│   23         y[i] = a*x[i]*y[i];                               │
│   24 }                                                         │
└────────────────────────────────────────────────────────────────┘

┌────────────────────────────────────────────────────────────────┐
│B+> <+0>   s_load_dword s3, s[0:1], 0x34                        │
│    <+32>  v_add_u32_e32 v0, s2, v0                             │
│    <+36>  v_cmp_gt_i32_e32 vcc, s4, v0                         │
│    <+40>  s_and_saveexec_b64 s[2:3], vcc                       │
│    ...                                                         │
│    <+108> global_load_dword v2, v[2:3], off                    │
│    <+124> global_load_dword v3, v[0:1], off                    │
│    <+144> v_mul_f32_e32 v2, s0, v2                             │
│    <+152> v_mul_f32_e32 v2, v2, v3                             │
│    <+156> global_store_dword v[0:1], v2, off                   │
└────────────────────────────────────────────────────────────────┘
 amd-dbgapi AMDGPU Wave 1:1:1:1 (asm) In: saxmy    L21  PC:<+0>
```

Public



# Reading LDS (local data share) memory

- *Local data share* (LDS) is inspected just like inspects global memory: use `x` or `print`
  - the address is tagged `local#<offset>` to denote it is *local* address space, not a global VA
- example here is a small kernel that fills a `__shared__` array `s` (which lives in LDS)

```
(gdb) x/4fw s          # s is a __shared__ array -> lives in LDS
local#0x0:  0.00843411591  0.0702233389  -0.0454526655  0.00581426499
(gdb) print &s
$5 = (float (*)[64]) local#0x0
```

```
__global__ void
lds_demo(float* out)
{
  __shared__ float s[64];  // in LDS
  int t = threadIdx.x;
  s[t] = t * 1.5f;
  __syncthreads();
  out[t] = s[t];
}
```

Public



# Page fault: from the faulting wave to the host stack

- bug: the two device pointers are null (no `hipMalloc`)
- kernels are asynchronous, host may learn about a problem much later
  - often leads to user misattributing blame to a non-buggy kernel
- ensure errors are reported immediately
  - use `hipDeviceSynchronize` - but it will decimate your performace
  - `export HIP_LAUNCH_BLOCKING=1; export AMD_SERIALIZE_KERNEL=3; export AMD_SERIALIZE_COPY=3` if you cannot
  - `export OMPX_FORCE_SYNC_REGIONS=1` for OpenMP codes
- `run` faults in a *GPU wave* (it has no CPU stack). `info threads` lists the host threads AND the four waves.
- `thread 1` then `where` crosses to the host stack: the CPU is parked in `hipDeviceSynchronize`, waiting on the device.

```
$ rocgdb ./saxmy
[rocgdb informational cruft removed]
Reading symbols from ./saxmy...
(gdb) set amdgpu precise-memory on
(gdb) run

Thread 6 "saxmy" received signal SIGSEGV, Segmentation fault.
[Switching to thread 6, lane 0 (AMDGPU Lane 1:2:1:1/0 (0,0,0)[0,0,0])]
saxmy (n=256, x=0x0, incx=1, y=0x0, incy=1) at saxmy.cpp:23
23          y[i] = a*x[i]*y[i];
(gdb) info threads
  Id   Target Id                                Frame
  1    Thread ... "saxmy"  WaitRelaxed () from libhsa-runtime64.so
  2    Thread ... "saxmy"  __GI___ioctl () at ioctl.c:36
* 6    AMDGPU Wave 1:2:1:1 (0,0,0)/0 "saxmy"  saxmy (...) at saxmy.cpp:23
  7    AMDGPU Wave 1:2:1:2 (0,0,0)/1 "saxmy"  saxmy (...) at saxmy.cpp:23
  8    AMDGPU Wave 1:2:1:3 (1,0,0)/0 "saxmy"  saxmy (...) at saxmy.cpp:23
  9    AMDGPU Wave 1:2:1:4 (1,0,0)/1 "saxmy"  saxmy (...) at saxmy.cpp:23
(gdb) thread 1
[Switching to thread 1 ...]
(gdb) where
#0  WaitRelaxed () from libhsa-runtime64.so
...   (HSA and HIP runtime frames)
#12 hipDeviceSynchronize () from libamdhip64.so
#13 main () at saxmy.cpp:37
```

```
16  __constant__ float a = 1.0f;
17
18  __global__
19  void saxmy(int n, float const* x, int incx, float* y, int incy)
20  {
21      int i = blockDim.x*blockIdx.x + threadIdx.x;
22      if (i < n)
23          y[i] = a*x[i]*y[i];
24  }
25
26  int main()
27  {
28      int n = 256;
29      std::size_t size = sizeof(float)*n;
30
31      float* d_x = nullptr;
32      float* d_y = nullptr;
33
34      int num_groups = 2;
35      int group_size = 128;
36      saxmy<<<num_groups, group_size>>>(n, d_x, 1, d_y, 1);
37      hipDeviceSynchronize();
```

Public



# Keep the evidence: a GPU core dump

- A GPU core dump freezes the faulting wave state for a post-mortem on *any* machine - no GPU needed to reload it. There are two ways to get one.
- *Without a debugger* (below): run the program normally with `ulimit -c unlimited` set; a device fault leaves a host `core` and a GPU `core.gpu`. `roccoremerge` combines the two into one core rocgdb can load - then `rocgdb ./saxmy <core>` and the four waves are all there (`where`, `info threads`, `thread apply all bt`).
- *Under rocgdb*, at the fault: `generate-core-file` (`gcore`) writes a combined core directly - handy when a support engineer asks for a core you did not already keep.

```
$ ulimit -c unlimited
$ ./saxmy                                  # a plain run, no debugger
Memory access fault by GPU node-1 on address (nil). Reason: Unknown.
Aborted (core dumped)                      # leaves host "core" + GPU "core.gpu"

$ roccoremerge saxmy.core core core.gpu    # combine the host + GPU cores
Creating saxmy.core
 Analysing core ...
 Copying segment from core.gpu  vma=0x782444800000   (combined core ~300 MB)

$ rocgdb ./saxmy saxmy.core
Core was generated by `./saxmy'.
Program terminated with signal SIGABRT, Aborted.
Thread 5 (AMDGPU Wave 1:1:1:1 (0,0,0)/0) stopped because of SIGSEGV.
Thread 6 (AMDGPU Wave 1:1:1:2 (0,0,0)/1) stopped because of SIGSEGV.
Thread 7 (AMDGPU Wave 1:1:1:3 (1,0,0)/0) stopped because of SIGSEGV.
Thread 8 (AMDGPU Wave 1:1:1:4 (1,0,0)/1) stopped because of SIGSEGV.
[Current thread is 8, lane 0 (AMDGPU Lane 1:1:1:4/0 (1,0,0)[64,0,0])]
(gdb) where
#0  saxmy (n=256, x=0x0, incx=1, y=0x0, incy=1) at saxmy.cpp:23
```

Public



# Fortran with OpenMP target offload

- The OpenMP-offload twin of the HIP page fault, in Fortran: the same `saxmy`, but `x` is never allocated, so the `!$omp target` region reads a *null pointer* on the device. `OMP_TARGET_OFFLOAD=MANDATORY` forces the kernel onto the GPU; `set amdgpu precise-memory on` pins the stop to the faulting load.
- `run` stops with `SIGSEGV` at `saxmy.f90:24`, the `y(i)=a*x(i)*y(i)` line that reads `x`. `info threads` shows all eight waves halted there and `info dispatches` the launch grid. (The C sibling prints `x=0x0` in the frame outright; the Fortran `x` is a descriptor whose data pointer is null.)

```
17    real :: a = 2.0
18    real, allocatable :: x(:), y(:)
19    integer :: i
20    allocate(y(n))                ! BUG: x is never allocated -> null pointer
21    y = 2.0
22    !$omp target teams distribute parallel do map(tofrom:y)
23    do i = 1, n
24       y(i) = a*x(i)*y(i)         ! device reads x(i) from a null pointer -> fault
25    end do
26    !$omp end target teams distribute parallel do
27    print *, 'y(1) =', y(1), '  y(n) =', y(n)
28    deallocate(y)
29  end program saxmy_omp
```

Public



# Fortran with OpenMP target offload

- Note the launch differs from the HIP `saxmy.cpp`: `info dispatches` below shows a *workgroup of only 32* (`[32]`), not 128 - so each wave here is half-full (32 of 64 lanes), and the OpenMP runtime chose a different grid for the `target` region.

```
(gdb) set amdgpu precise-memory on
(gdb) run

Thread 6 "__omp_offlo-67fd" received signal SIGSEGV, Segmentation fault.
[Switching to thread 6, lane 0 (AMDGPU Lane 1:1:1:1/0 (0,0,0)[0,0,0])]
0x00007fffed12dd14 in __omp_offloading_...QQmain_l22..omp_par () at saxmy.f90:24
24          y(i) = a*x(i)*y(i)
(gdb) info threads
  Id   Target Id                                        Frame
  1    Thread 0x7ffff02e9f40 "saxmyf"  __GI___ioctl () at ioctl.c:36
  ...
* 6    AMDGPU Wave 1:1:1:1 (0,0,0)/0 "__omp_offlo-67fd"  ... at saxmy.f90:24
  7    AMDGPU Wave 1:1:1:2 (1,0,0)/0 "__omp_offlo-67fd"  ... at saxmy.f90:24
  ...
  13   AMDGPU Wave 1:1:1:8 (7,0,0)/0 "__omp_offlo-67fd"  ... at saxmy.f90:24
(gdb) backtrace
#0  ... __omp_offloading_...QQmain_l22..omp_par () at saxmy.f90:24
#4  ... __omp_offloading_...QQmain_l22 (..., x=<error reading variable: Invalid
        address for address space 0: 0xffffe4202000>) at saxmy.f90:22
(gdb) info dispatches
  Id   Target Id                      Grid  Workgroup Fence  Kernel Function
* 1    AMDGPU Dispatch 1:1:1 (PKID 1) [256] [32]      As|Rs  __omp_offloading_10302_3bc000c__QQmain_l22
```

Public



# GPU memory-fault triage in an MPI batch job

- The Part-I per-rank wrapper, now with MPI-GPU code: it writes a command file per rank (`catch signal SIGSEGV`, `set amdgpu precise-memory`, `run`, `thread apply all bt`, `info dispatches`) and one `gdb_out_<rank>.txt` per rank - no interleaving.
- Launch it across the ranks (`mpirun -np N ./gdb_mpi_wrapper_all_ranks.sh ./mpi_saxmy`), then read each log *from the end*. Here the last rank passed a null `y`, so only `gdb_out_1.txt` shows the GPU fault (below); the other ranks' logs just say "done"
- The wrapper `gdb_mpi_wrapper_all_ranks.sh` ships in the AMD `HPCTrainingExamples` repo (and in this course's `exercises/cpu/06_batch_mpi/`).
- For real applications, backtraces can be very long
  - AI tools are really good at interpreting them

```
$ cat gdb_out_1.txt       # the faulting rank (the others just say "done")
Catchpoint 1 (signal SIGSEGV)
[Switching to thread 9, lane 0 (AMDGPU Lane 1:2:1:1/0 ...)]
Thread 9 "saxmy" hit Catchpoint 1 (SIGSEGV), saxmy (n=256, a=2, x=0x..., y=0x0) at mpi_saxmy.cpp:20
20          y[i] = a * x[i] * y[i];
(gdb) thread apply all bt
Thread 12 (AMDGPU Wave 1:2:1:4) #0 saxmy (... y=0x0) at mpi_saxmy.cpp:20
Thread 11 (AMDGPU Wave 1:2:1:3) #0 saxmy (... y=0x0) at mpi_saxmy.cpp:20
Thread 10 (AMDGPU Wave 1:2:1:2) #0 saxmy (... y=0x0) at mpi_saxmy.cpp:20
Thread  9 (AMDGPU Wave 1:2:1:1) #0 saxmy (... y=0x0) at mpi_saxmy.cpp:20
```

Public



# AMD_LOG_LEVEL

- `AMD_LOG_LEVEL` is an *environment variable* you set before launching the program (`AMD_LOG_LEVEL=3 ./saxmy`). It makes the ROCm runtime report what it is doing - agent enumeration, queue and dispatch setup, memory allocations - to `stderr`. Higher means more detail: 0 none, 1 error, 2 warning, 3 info, 4 debug.
- No debugger needed: it is a fast first look when a run misbehaves. (Timestamps trimmed below; the `Initalizing` spelling is the runtime's own.)
- *It is verbose:* level 4 (`debug`) floods the terminal; even level 3 runs to thousands of lines on a real workload. This toy run prints 41 lines at level 3 (74 at level 4) - the five below are just the head. Start low, `grep` for what you want, or redirect (`2> log`); raise the level only once you know what you are chasing.

```
$ AMD_LOG_LEVEL=3 ./saxmy    (head of 41 lines)
:3:rocdevice.cpp :434 :  Initalizing runtime stack, Enumerated GPU agents = 1
:3:rocdevice.cpp :181 :  Numa selects cpu agent[0]=... for gpu agent=... CPU<->GPU XGMI=1
:3:comgrctx.cpp  :89  :  Loaded COMGR library version 3.3.
:3:rocdevice.cpp :1661:  addressableNumVGPRs=512, totalNumVGPRs=512, vGPRAllocGranule=8
:3:rocdevice.cpp :1672:  imageSupport=0
```

Public



# Command cheat-sheet: gdb and rocgdb

- **Start**
  - `rocgdb <prog> [core]` — open a program (+ core)
  - `rocgdb --args <prog> a b` — with arguments
  - `rocgdb -p PID` — attach to a process
- **Run**
  - `set amdgpu precise-memory on` — fault at the exact insn
  - `r[un]` — start, to a breakpoint/fault
  - `c[ontinue]` — resume
  - `n[ext]` — step over calls
  - `s[tep]` — step into calls
  - `fin[ish]` — run out of this frame
  - `u[ntil] N` — run to line N
  - `stepi` (`si`) — one instruction
  - `q[uit]` — leave rocgdb
- **Breakpoints and catch**
  - `b[reak] <where>` — at `f` or `file:line`
  - `b <where> if i==63` — conditional
  - `tb[reak]` — one-shot
  - `set breakpoint pending on` — for unloaded kernels
  - `catch signal SIGSEGV` — stop on the fault
  - `i[nfo] b` — list breakpoints
  - `dis[able] N` — disable one
  - `en[able] N` — enable one
  - `d[elete] N` — remove one
- **Environment (before the run)**
  - `AMD_LOG_LEVEL=3` — runtime log
  - `HSA_XNACK=1` — demand paging
  - `HIP_ENABLE_DEFERRED_LOADING=0` — eager kernel load
- **Stack and data**
  - `backtrace` (`bt`) — the call stack
  - `f[rame] N` — select a frame
  - `up` — move up a frame
  - `down` — move down a frame
  - `i[nfo] args` — function arguments
  - `i[nfo] locals` — local variables
  - `p[rint] x` — value of x
  - `p/x x` — value in hex
  - `i[nfo] registers` — register values
- **GPU: waves and lanes**
  - `i[nfo] agents` — the GPUs
  - `i[nfo] queues` — HSA queues
  - `i[nfo] dispatches` — kernel launches
  - `i[nfo] threads` — host threads + waves
  - `i[nfo] lanes` — lanes of this wave
  - `thread N` — select a wave
  - `lane N` — select a lane
- **GPU: kernel state**
  - `p threadIdx.x` — per-lane builtin
  - `p blockIdx.x` — per-work-group builtin (a group's waves share it)
  - `set scheduler-locking on` — freeze other waves
  - `disas[semble]` — the kernel ISA
  - `x/8i $pc` — next instructions
  - `x/4fw &s` — LDS (`local#`) memory
  - `generate-core-file` (`gcore`) — save a GPU core
- **Combine CPU and GPU cores (outside `rocgdb`)**
  - `roccoremerge combined_core core core.gpu`

Public



# Thanks and references

- To the HLRS team for the course, the invitation, and system access.
- To the AMD HPC & Sovereign AI Customer Enablement team for the material this course builds on - in particular Paul, Bob, Giacomo, Marius, and Johanna for the decks, comments, and reviews.
- And to you - questions and feedback are welcome, during and after.
- Where to look next:
  - In-tool: `help <cmd>`, `apropos <word>`, `man rocgdb`, tab completion.
  - Bundled docs: `/opt/rocm-<ver>/share/doc/rocgdb/` → `rocgdb.pdf`, `rocrefcard.pdf`.
  - Online: the ROCgdb docs portal (https://rocm.docs.amd.com/projects/ROCgdb/).
  - HLRS: https://kb.hlrs.de/platforms/index.php/ROCgdb

Public



# The exercises in HPCTrainingExamples (Part I CPU + Part II GPU)

- All the exercises live in AMD's HPCTrainingExamples repository - clone it and work through the `README` in each exercise directory. They are self-contained and independent - do them in any order, and skip any you like:

```
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/Rocgdb/rocgdb_training/exercises
```

- Part I - CPU (`cpu/`, no GPU needed): `cd common && make` to build the shared sample programs once, then `cd ../01_running && less README.md` and work through them.
- Part II - GPU (`gpu/`): grab a GPU-node allocation first (`rocgdb` attaches to the device there), then work through each `README`:
  - `01_pagefault` - find the bad device pointer behind a HIP page fault.
  - `02_omp_offload` - a null-pointer page fault in Fortran/C OpenMP target offload.
  - `03_mpi_pagefault` - find the rank that faults on the device (per-rank wrapper).
- On Hunter, `git clone` needs the site HTTPS proxy: set `https_proxy` first (see the HLRS proxy guide), or clone elsewhere and copy the tree across.

# Disclaimer

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED ``AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD Arrow logo, ROCm, Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies. The OpenMP® name and the OpenMP® logo are registered trademarks of the OpenMP® Architecture Review Board. Linux® is the registered trademark of Linus Torvalds in the U.S. and other countries. Siemens® is a registered trademark of Siemens Product Lifecycle Management Software Inc., or its subsidiaries or affiliates, in the United States and in other countries.

© 2026 Advanced Micro Devices, Inc. All rights reserved.


AMD