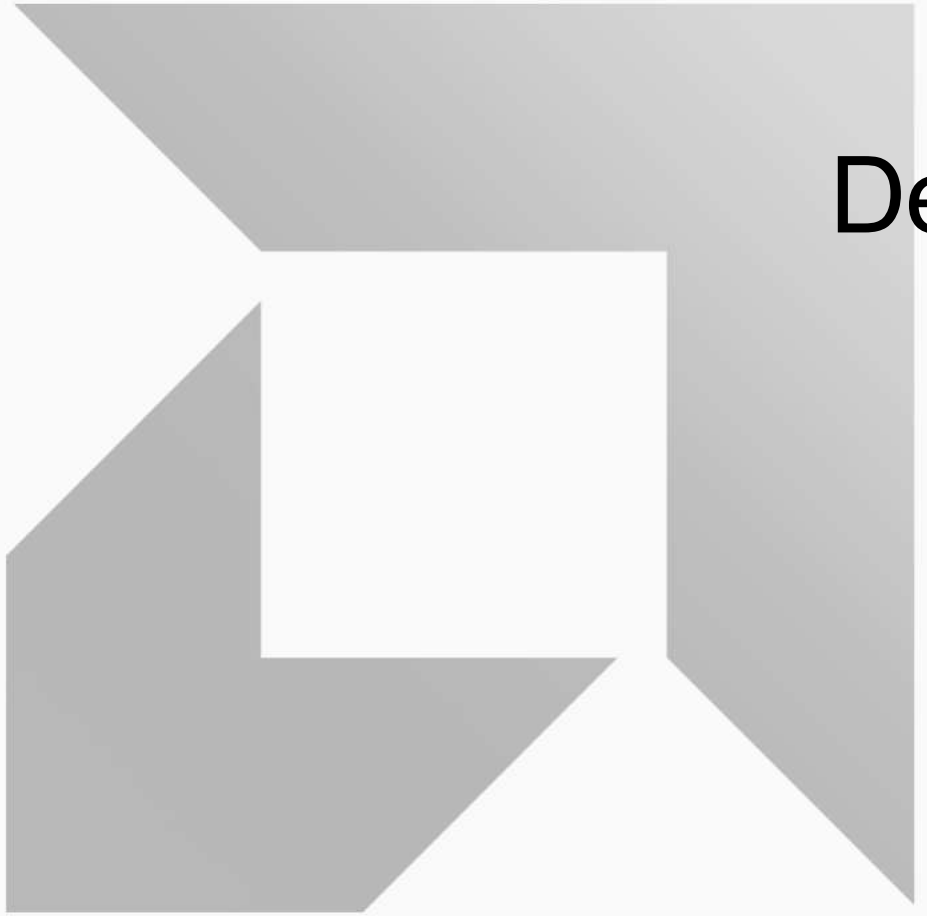


Debugging with ROCgdb

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

AMD
together we advance_

Public

What rocgdb is

- `rocgdb`: AMD's fork of the GNU source-level debugger (GDB)
  - attach to a program (this stops program execution until debugger allows it to continue)
  - inspect where it is and what its data are, and
  - change either.
- The same commands debug the CPU (host) and the GPU (device) parts of a heterogeneous program.
- Why a debugger beats `print` or `printf` debugging:
  - move around a stopped program (`step`, `jump`, re-run a piece) without editing and re-compiling;
  - print variables you never thought to print when you wrote the code;
  - catch a crash (segfault, `SIGBUS` , an exception thrown by C++) and see exactly where and how it happened;
  - attach to an already-running job and look inside it live.
- `rocgdb` is based on GDB, the GNU Debugger (which is CPU-only)
  - `rocgdb` is a superset of GDB
  - it adds GPU-specific capabilities that have no CPU counterpart (Part II).

Public

Building for debugging: with debug info, minimal optimization

Flag	What it does
-ggdb	emit debug information in the most expressive format for GDB/rocgdb, GDB extensions included; use -ggdb with rocgdb.
-g	emits the platform-default format, includes less info than -ggdb, please **use -ggdb** when available
-O0	no optimization at all, so machine code follows the source most literally - the easiest to single-step, and the debug default here (but it can hide bugs that only surface once the optimizer runs).

- Release builds optimize (commonly optimization levels `-O2` , `-O3` , or `-O3 -ffast-math`).
- Optimization can hide a local (`<optimized out>`) or move a breakpoint to a line you never asked for.
- some bugs appear only when certain optimization options are active
  - these can be activated directly or by choosing a particular optimization level
  - reproduce with those options or at that level
- some compilers have a recommended debug-friendly optimization level `-Og` , use it when available
- on this course, we use `amdclang` , `amdclang++` , and `amdflang`
- on Cray systems, you may use the wrappers `cc` / `CC` / `ftn` (they wrap the vendor compiler; the same `-ggdb` / `-O0` flags apply)

Public

Starting rocgdb

Invocation	What it does
rocgdb --args <program> <args>	load the program, ready to run; --args also records the program's own arguments.
rocgdb --pid <PID>	attach to an already-running process by PID.
rocgdb <program> <core>	open a core dump for post-mortem inspection.
rocgdb -help	the option list.

- `--args` : the next word following it is the program to run and anything remaining are its arguments.
  - `rocgdb` remembers `<args>` , so a bare `run` re-uses them.
- Shorthands: `rocgdb --args <program>` and `rocgdb <program>`
  - load the program, record no arguments to it
  - must retype `<args>` - if any - on every `run`
- the file `~/.gdbinit` is read at `rocgdb` start
  - put any `set` commands etc that you want to *always* set here

Public

A note before we go on: much HPC debugging is non-interactive

- MPI programs can be hard to debug without specialized tools
  - attaching to one rank is a cheap way in
    - … but can cause or hide races, hangs, timeout-exits, perturbed timings.
  - Usable if you know the limits; other ranks usually wait at an `MPI_Barrier` (or similar).
  - there's a slide about a free interactive MPI debugger later on
- often batch systems with no interactive prompt.
  - makes even non-MPI debugging more challenging
- `rocgdb` can run entirely from a script – see Section "Batch, non-interactive `rocgdb`"

Public

Commands at a glance

Command	What it does
run / r	start the program; it runs freely until something stops it.
run <args>	start it with these arguments.
start	run, but stop at the first line of main.
starti	stop at the very first instruction (before main).
continue / c	resume until the next stop.
Ctrl-c	interrupt a running program (sends SIGINT to rocgdb, not the program).
kill	stop the running program but stay in rocgdb.
quit	leave rocgdb.

- Most commands: a short form or any unambiguous prefix (`r = run` , `c = continue`).
- `Enter` on an empty line repeats the last command
  - so a run of `continue` (or, later, of steps) is just the `Enter` key.

Public

Starting and continuing execution

- Compile with `-ggdb` (see Section "Building for debugging"), launch under `rocgdb`, drive from the `(gdb)` prompt.
- Three ways to begin:
  - `run` - let it go freely: until a breakpoint, a fault, or it finishes.
  - `start` - a one-shot breakpoint at `main` , then `run` (control from your first line).
  - `starti` - stop at the very first instruction, before the run-time reaches `main` .
- `continue` resumes, again freely, to the next breakpoint.
- `continue` is not `run` : `run` restarts from scratch, `continue` carries on from here.
- `Ctrl-c` interrupts a run; `kill` ends the run without leaving `rocgdb`.

Public

Launching a program, then run

- `rocgdb --args` : everything after it becomes the program's `argv`; `run` lets it go.
- Here `alpha`, `beta` arrive as `argv[1]` , `argv[2]` :

```
$ amdclang -ggdb args.c -o args
$ rocgdb --args ./args alpha beta
[gdb informational cruft removed]
Reading symbols from ./args...
(gdb) run
got 2 argument(s):
 argv[1] = alpha
 argv[2] = beta
[Inferior 1 (process 268) exited normally]
```

```
5  #include <stdio.h>
6
7  int main(int argc, char **argv)
8  {
9      printf("got %d argument(s):\n", argc - 1);
10     for (int i = 1; i < argc; i++)
11         printf(" argv[%d] = %s\n", i, argv[i]);
12     return 0;
13 }
```

Public

Attaching to a program that is already running

- Debug a job that is already running: `attach` by PID, don't launch it.
- Start it (backgrounded, PID in `$!`), then point `rocgdb` at that PID:

```
$ ./sim & # a long-running job, now in the background
[1] 48213 # ...its PID (also available as $!)
$ rocgdb --pid 48213 # or: start rocgdb, then
(gdb) attach 48213 ... # rocgdb stops it wherever it currently is
(gdb) continue # resume it, or 'detach' to leave it running
```

- Attaching pauses it wherever it is; `continue` resumes, `detach` leaves it running without `rocgdb`.
- You can only attach to your own processes, and the system may restrict it further
  - attaching to your own jobs on a compute node is usually fine

Public

Stopping at the first line: start, then continue

- `start` = a one-shot breakpoint at main, then `run`
  - this is before any of your code runs
- continue then carries on to the end. The same program, now in Fortran, via `start` :

```
$ rocgdb --args ./argsf alpha beta
[gdb informational cruft removed]
Reading symbols from ./argsf...
(gdb) start
Temporary breakpoint 1 at 0x22364b: file argsf.f90, line 9.
Temporary breakpoint 1, ARGS () at argsf.f90:9
9           nargs = command_argument_count()
(gdb) continue
got 2 argument(s):
 arg(1) = alpha
 arg(2) = beta
[Inferior 1 (process 282) exited normally]
```

```
5   program args
6   implicit none
7   integer :: i, nargs
8   character(len=64) :: arg
9   nargs = command_argument_count()
10  print '(a,i0,a)', "got ", nargs, " argument(s):"
11  do i = 1, nargs
12      call get_command_argument(i, arg)
13      print '(a,i0,a,a)', " arg(", i, ") = ", trim(arg)
14  end do
15  end program args
```

Public

Commands at a glance

Command	What it does
next / n	run the current line, stepping OVER any calls.
step / s	run the current line, stepping INTO a call if there is one.
until / u	like `next`, but won't single-step a loop twice: at a loop's back-edge it runs the body out to the line after it.
until <line>	run to that line, or (if it is never reached) until the frame returns.
finish	run until the current function returns, and print the return value.
return [value]	force the current function to return now, skipping the rest of it.
stepi / si / nexti / ni	the same, but one machine instruction at a time.

Public

Line-level stepping

- One source line at a time; `next` and `step` differ only at a call:
  - `next` - run the call whole, stop on the next line.
  - `step` - go into the callee, stop on its first line.
  - note that macros can hide calls
- Landed in library code by mistake? (a common slip)
  - `finish` runs the function to its return , drops you back in the caller.
- `until` - like `next` , but runs a loop out in one step (won’t single-step it twice)
- `until <line>` : run to a line, or out of the current frame

Public

next: run the call and move on

- Demo: `add(a,b)` called from `main` ; `start` stops at its first line; comments mark where `next` and `step` lead.
- Walk lines 15-18, without entering the function on line 17:

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) start
Temporary breakpoint 1 at 0x20175f: file step.c, line 15.
Temporary breakpoint 1, main () at step.c:15
15            int x = 3; /* start stops here */
(gdb) next
16            int y = 4;
(gdb) next
17            int z = add(x, y); /* next: over, step: into */
(gdb) next
18            printf("z = %d\n", z);
```

```
5   #include <stdio.h>
6
7   int add(int a, int b)
8   {
9       a = a + b; /* step lands here */
10      return a;
11  }
12
13  int main(void)
14  {
15      int x = 3; /* start stops here */
16      int y = 4;
17      int z = add(x, y); /* next: over, step: into */
18      printf("z = %d\n", z);
19      return 0;
20  }
```

Public

step: into a function

- Same as above (the same program in Fortran), but enter the function, too:

```
$ rocgdb ./stepf
[gdb informational cruft removed]
Reading symbols from ./stepf...
(gdb) start
Temporary breakpoint 1 at 0x2217f8: file stepf.f90, line 15.
Temporary breakpoint 1.1, STEP () at stepf.f90:15
15              x = 3 ! start stops here
(gdb) next
16              y = 4
(gdb) next
17              z = add(x, y) ! next: over, step: into
(gdb) step
add (a=3, b=4) at stepf.f90:8
8               add = a + b ! step lands here
```

```
5   integer function add(a, b)
6       implicit none
7       integer, intent(in) :: a, b
8       add = a + b ! step lands here
9   end function add
10
11  program step
12      implicit none
13      integer :: x, y, z
14      integer :: add
15      x = 3 ! start stops here
16      y = 4
17      z = add(x, y) ! next: over, step: into
18      print *, "z =", z
19  end program step
```

Public

until: run out of a loop

- `until` (no arg): like `next` , but at a loop's back-edge it runs the whole loop body out, to the next line past it.
- Step the first pass (lines 10 → 11 → 12 → 11); the next `until` runs the rest of the loop out, straight to line 13 - then `continue` lets `main` print the sum:

```
$ rocgdb ./loop
[gdb informational cruft removed]
Reading symbols from ./loop...
(gdb) start
Temporary breakpoint 1 at 0x2017c6: file loop.c, line 18.
Temporary breakpoint 1, main (argc=1, argv=0x7fffffffd148) at loop.c:18
18            int n = (argc > 1) ? atoi(argv[1]) : 5;
(gdb) next
19            long total = sum_to(n); /* step here to enter sum_to */
(gdb) step
sum_to (n=5) at loop.c:10
10            long s = 0;
(gdb) until
11            for (int i = 1; i <= n; i++)
(gdb) until
12                s += i;
(gdb) until
11            for (int i = 1; i <= n; i++)
(gdb) until
13            return s;
(gdb) continue
sum 1..5 = 15
[Inferior 1 (process 324) exited normally]
```

```
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  long sum_to(int n)
9  {
10     long s = 0;
11     for (int i = 1; i <= n; i++)
12         s += i;
13     return s;
14 }
15
16 int main(int argc, char **argv)
17 {
18     int n = (argc > 1) ? atoi(argv[1]) : 5;
19     long total = sum_to(n); /* step here to enter sum_to */
20     printf("sum 1..%d = %ld\n", n, total);
21     return 0;
22 }
```

Public

finish: run out of a function

- After stepping into `add` , `finish` runs it to return and lands back in `main` - printing the value returned:

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) start
Temporary breakpoint 1 at 0x20175f: file step.c, line 15.
Temporary breakpoint 1, main () at step.c:15
15          int x = 3; /* start stops here */
(gdb) next
16          int y = 4;
(gdb) next
17          int z = add(x, y); /* next: over, step: into */
(gdb) step
add (a=3, b=4) at step.c:9
9           a = a + b; /* step lands here */
(gdb) finish
0x0000000000201778 in main () at step.c:17
17          int z = add(x, y); /* next: over, step: into */
Value returned is $1 = 7
```

```
5   #include <stdio.h>
6
7   int add(int a, int b)
8   {
9       a = a + b; /* step lands here */
10      return a;
11  }
12
13  int main(void)
14  {
15      int x = 3; /* start stops here */
16      int y = 4;
17      int z = add(x, y); /* next: over, step: into */
18      printf("z = %d\n", z);
19      return 0;
20  }
```

Public

Commands at a glance

Command	What it does
break <where> / b	set a breakpoint (see the variants below).
info breakpoints / i b	list breakpoints, with their numbers and hit counts.
delete <n> / d <n>	remove breakpoint number n (delete alone: all).
disable <n> / enable <n>	switch a breakpoint off / back on without deleting it.
clear	remove the breakpoint at a location.
tbreak / tb	a temporary breakpoint - it fires once, then removes itself.

Public

break (b) takes a location

Location	Breakpoint at
b <file>:<line>	at a specific source line.
b <function>	at the entry of a function.
b +<n> / b -<n>	n lines after / before the current line.
b <label>	at an assembly label.

- `b <function>` (e.g. `b add`): the function's first line, in whatever file.
- `b <file>:<line>` : a specific line - the everyday case.
- Optimized builds: a breakpoint can bind to a different line than you asked – see Section "What optimization does to debugging".
- Not-yet-loaded code (a shared library, a GPU kernel): pending breakpoints – see Section "Breakpoints on not-yet-loaded symbols".
- `tbreak main` then `run` is exactly what `start` does: a one-shot stop at `main` .

Public

Set a few, then list them

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) tbreak main
Temporary breakpoint 1 at 0x20175f: file step.c, line 15.
(gdb) break add
Breakpoint 2 at 0x20173a: file step.c, line 9.
(gdb) break step.c:18
Breakpoint 3 at 0x20177b: file step.c, line 18.
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     del  y   0x000000000020175f in main at step.c:15
2       breakpoint     keep y   0x000000000020173a in add at step.c:9
3       breakpoint     keep y   0x000000000020177b in main at step.c:18
(gdb) run
Temporary breakpoint 1, main () at step.c:15
15            int x = 3; /* start stops here */
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x000000000020173a in add at step.c:9
3       breakpoint     keep y   0x000000000020177b in main at step.c:18
```

```
5  #include <stdio.h>
6
7  int add(int a, int b)
8  {
9      a = a + b; /* step lands here */
10      return a;
11 }
12
13 int main(void)
14 {
15     int x = 3; /* start stops here */
16     int y = 4;
17     int z = add(x, y); /* next: over, step: into */
18     printf("z = %d\n", z);
19     return 0;
20 }
```

Public

Listing, disabling, and removing breakpoints

- `info breakpoints` (`i b`): each breakpoint, its number, and - if non-zero - its hit count.
- The number is the handle - `delete` , `disable` , `enable` all take it.
- `disable` : silence one but keep it (and its conditions); `enable` brings it back.
- `delete` : remove it for good (`delete` alone removes all).
- `tbreak` : one-shot - fires once, then removes itself.

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) break main
Breakpoint 1 at 0x20175f: file step.c, line 15.
(gdb) break add
Breakpoint 2 at 0x20173a: file step.c, line 9.
(gdb) break step.c:18
Breakpoint 3 at 0x20177b: file step.c, line 18.
(gdb) run
Breakpoint 1, main () at step.c:15
15            int x = 3; /* start stops here */
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000020175f in main at step.c:15
        breakpoint already hit 1 time
2       breakpoint     keep y   0x000000000020173a in add at step.c:9
3       breakpoint     keep y   0x000000000020177b in main at step.c:18
(gdb) disable 1
(gdb) delete 2
(gdb) info breakpoints
Num     Type           Disp Enb Address            What
1       breakpoint     keep n   0x000000000020175f in main at step.c:15
        breakpoint already hit 1 time
3       breakpoint     keep y   0x000000000020177b in main at step.c:18
```

```
5  #include <stdio.h>
6
7  int add(int a, int b)
8  {
9      a = a + b; /* step lands here */
10     return a;
11 }
12
13 int main(void)
14 {
15     int x = 3; /* start stops here */
16     int y = 4;
17     int z = add(x, y); /* next: over, step: into */
18     printf("z = %d\n", z);
19     return 0;
20 }
```

Public

Commands at a glance

Command	What it does
backtrace / bt / where	the call stack: who called whom to get here.
frame <n> / up / down	select a stack frame to inspect.
info args / info locals	the selected frame’s arguments and local variables.
print <expr> / p	evaluate and print an expression.
p/<fmt> <expr>	print in a format (x hex, d dec, t binary, c char, f float, . . .).
print *<addr>@<count>	print count consecutive values from an address.
x/<n><f><u> <addr>	examine memory (n items, format f, unit size u).

Public

The call stack: where execution is, and how it got there

- Each active function call is a frame on the stack: its arguments, its local variables, and the line it will return to.
- `backtrace` (`bt`): the chain of calls that got here, innermost frame first.
- `frame n` (or `up` / `down`): select a frame to inspect.
- `info args` / `info locals` always show the selected frame.
- Below: the fault is a `NULL`-pointer dereference in `deref` (`p` is `0x0`)
  - `bt` shows `main` called it
  - selecting `frame 1` (`main`) shows `main`'s locals - its line 15 has not finished, so nothing there is meaningful yet

```
$ rocgdb ./crash
[gdb informational cruft removed]
Reading symbols from ./crash...
(gdb) run
Program received signal SIGSEGV, Segmentation fault.
0x000000000020177c in deref (p=0x0) at crash.c:9
9               return *p; /* crashes when p is NULL */
(gdb) bt
#0  0x000000000020177c in deref (p=0x0) at crash.c:9
#1  0x0000000000201750 in main () at crash.c:15
(gdb) frame 1
#1  0x0000000000201750 in main () at crash.c:15
15              int x = deref(p);
(gdb) info locals
p = 0x0
x = 32767
```

```
5  #include <stdio.h>
6
7  static int deref(int *p)
8  {
9      return *p; /* crashes when p is NULL */
10 }
11
12 int main(void)
13 {
14     int *p = NULL;
15     int x = deref(p);
16     printf("%d\n", x);
17     return 0;
18 }
```

Public

Printing variables and expressions

- `print` (`p`): evaluate a C expression in the current frame - variable, field, arithmetic, even a call.
- `print arg@entry` : the value on entry, even after it is overwritten.
  - needs an optimized build that keeps `add` callable (here `-O1 -fno-inline`); `-O0` keeps no entry record.
  - `add` does `a = a + b` : `a` is `7` but `a@entry` is `3`.
- Expressions have side effects: `p x = 5` assigns - see Section Changing the program state.

```
$ rocgdb ./step0g
[gdb informational cruft removed]
Reading symbols from ./step0g...
(gdb) break add
Breakpoint 1 at 0x201720: file step.c, line 9.
(gdb) run
Breakpoint 1, add (a=3, b=4) at step.c:9
9             a = a + b; /* step lands here */
(gdb) info args
a = 3
b = 4
(gdb) next
10            return a;
(gdb) print a
$1 = 7
(gdb) print a@entry
$2 = 3
```

```
5  #include <stdio.h>
6
7  int add(int a, int b)
8  {
9      a = a + b; /* step lands here */
10      return a;
11  }
12
13  int main(void)
14  {
15      int x = 3; /* start stops here */
16      int y = 4;
17      int z = add(x, y); /* next: over, step: into */
18      printf("z = %d\n", z);
19      return 0;
20  }
```

Public

Output formats

- Add `/<code>` to `print` (or `x`) to force a representation:

Format	Prints as
p/x	hexadecimal.
p/d / p/u	signed / unsigned decimal.
p/t	binary (mnemonic: "t" for base *t*wo).
p/c	character.
p/f	floating point.
p/a	address (symbol + offset).

Public

Output formats

- Same bits, different lens. For numerical work: `f` (float), and `x/gf` to read memory as 8-byte doubles (Section Arrays and raw memory):

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) break add
Breakpoint 1 at 0x20173a: file step.c, line 9.
(gdb) run
Breakpoint 1, add (a=3, b=4) at step.c:9
9 a = a + b; /* step lands here */
(gdb) print a
$1 = 3
(gdb) print/x a
$2 = 0x3
(gdb) print/t a
$3 = 11
(gdb) print/c 65
$4 = 65 'A'
```

- `rocgdb` numbers each printed value `$1` , `$2` , . . . (its value history); reuse one later like a variable, e.g. `print $1 + 1` .

Public

Arrays and raw memory

- `print *ptr@count` : count elements from `ptr` .
  - `print` follows the type - counts elements (C's `ptr[n]` , not bytes); `print p` alone: just the address.
- `x/<count><f><u> <addr>` ("examine"): count items, format `f` , unit `u` .
  - units: `b` 1, `h` 2, `w` 4, `g` 8 bytes (formats as for `print`).
- `x` does not follow the variable - reuses the last format and size (demonstration on the next slide)
  - `x/3fg da` (doubles), then a bare `x/3 fa` : 4-byte floats read as 8-byte doubles - rubbish.
  - always give `x` a format and a size (count is safe to omit).
- Why: when you have an address but no typed variable - a raw buffer, a pointer into an array, device memory.
- GPU: an `x` variant reads LDS (local data share) memory – see Part II.

Public

Arrays and raw memory

```
$ rocgdb ./mem
[gdb informational cruft removed]
Reading symbols from ./mem...
(gdb) break mem.c:12
Breakpoint 1 at 0x201791: file mem.c, line 12.
(gdb) run
Breakpoint 1, main () at mem.c:12
12          printf("%g %g\n", dp[0], fa[0]);
(gdb) print da
$1 = {1.5, 2.5, 3.5}
(gdb) print dp
$2 = (double *) 0x7fffffffd000
(gdb) print *dp@3
$3 = {1.5, 2.5, 3.5}
(gdb) x/3fg da
0x7fffffffd000: 1.5     2.5
0x7fffffffd010: 3.5
(gdb) x/3 fa
0x7fffffffcff4: 8.0000018998980522      5.3360734001323986e-315
0x7fffffffd004: 5.3023991505199126e-315
(gdb) x/3fw fa
0x7fffffffcff4: 1.5     2.5     3.5
```

```
5   #include <stdio.h>
6
7   int main(void)
8   {
9       double da[3] = {1.5, 2.5, 3.5};
10      float  fa[3] = {1.5f, 2.5f, 3.5f};
11      double *dp = da; /* a pointer into da */
12      printf("%g %g\n", dp[0], fa[0]);
13      return 0;
14  }
```

Public

Source and disassembly

Command	What it does
list / l	show source around the current line (or a given function/line).
disassemble	show the machine code of the current (or named) function.
x/<n>i $pc	disassemble n instructions from the program counter.

- For example. . . (on the next slide)

Public

What optimization does to debugging

- With `run` , `break` and `info locals` now familiar, here is why the build's optimization level matters.
- Built with optimization (`-O2` or `-O3`), several things bite - and a debug build (`-O0`) settles them.

Public

Symptom 1: variables read as "optimized out"

- At `-O2` `rocgdb` often cannot show you a local - it prints `<optimized out>` .
- Here `sum` and `avg` are folded into registers or the return value: no storage left, so `info locals` has nothing to print.

```
$ rocgdb ./optim
[gdb informational cruft removed]
Reading symbols from ./optim...
(gdb) break average3
Breakpoint 1 at 0x201720: file optim.c, line 11.
(gdb) run
Breakpoint 1, average3 (a=2, b=4, c=9) at optim.c:11
11            int sum = a + b + c;
(gdb) info locals
sum = <optimized out>
avg = <optimized out>
```

```
5  #include <stdio.h>
6
7  /* noinline so the function survives -O2
8  and we can stop inside it. */
9  __attribute__((noinline)) int average3(int a, int b, int c)
10 {
11     int sum = a + b + c;
12     int avg = sum / 3;
13     return avg;
14 }
15
16 int main(void)
17 {
18     printf("%d\n", average3(2, 4, 9));
19     return 0;
20 }
```

Public

Symptom 2: a breakpoint binds to the wrong line

- At `-O2` the compiler fuses adjacent statements: several source lines can compile to a single machine instruction.
- A breakpoint on any fused line then lands on the same spot, not where you typed - here three different lines all resolve to one address:

```
$ rocgdb ./fuse
[gdb informational cruft removed]
Reading symbols from ./fuse...
(gdb) break fuse.c:12
Breakpoint 1 at 0x201720: file fuse.c, line 15.
(gdb) break fuse.c:13
Breakpoint 2 at 0x201720: file fuse.c, line 15.
(gdb) break fuse.c:14
Breakpoint 3 at 0x201720: file fuse.c, line 15.
```

```
5  #include <stdio.h>
6
7  /* At -O2 the optimizer fuses these assignments (a=7; a=a+7 -> a=14; b=28),
8     so several source lines share ONE machine instruction - a breakpoint set
9     on any of them lands on the same place. */
10 __attribute__((noinline)) int compute(int x)
11 {
12     int a = 7;
13     a = a + 7;
14     int b = a * 2;
15     int c = b + x;
16     return c;
17 }
18
19 int main(int argc, char **argv)
20 {
21     printf("%d\n", compute(argc)); /* argc is a run-time value */
22     return 0;
23 }
```

Public

Symptom 2: a breakpoint binds to the wrong line

- The same in Fortran - ask for line 10, 11 or 12 and all three bind to the same machine address (each reported as "`3 locations`"):

```
$ rocgdb ./fusef
[gdb informational cruft removed]
Reading symbols from ./fusef...
(gdb) break fusef.f90:10
Breakpoint 1 at 0x221c40: fusef.f90:10. (3 locations)
(gdb) break fusef.f90:11
Breakpoint 2 at 0x221c40: fusef.f90:11. (3 locations)
(gdb) break fusef.f90:12
Breakpoint 3 at 0x221c40: fusef.f90:12. (3 locations)
```

```
5  integer function compute(x)
6  implicit none
7  !GCC$ ATTRIBUTES noinline :: compute
8  integer, intent(in) :: x
9  integer :: a, b, c
10  a = 7
11  a = a + 7
12  b = a * 2
13  c = b + x
14  compute = c
15  end function compute
16
17  program fuse
18  implicit none
19  integer :: compute
20  print '(i0)', compute(command_argument_count())
21  end program fuse
```

Public

Symptom 3: an inlined function vanishes

- The optimiser often inlines a small function into its caller and drops the out-of-line copy from the execution path.
- With constant arguments it can fold the whole call to a number.
- Here `scale_add` is a small loop compiled without `__attribute__((noinline))` , so at `-O2` it inlines into `main` and folds to `10` .
- `break scale_add` still finds an out-of-line copy - but nothing calls it, so the breakpoint never fires

```
$ rocgdb ./inln
[gdb informational cruft removed]
Reading symbols from ./inln...
(gdb) break scale_add
Breakpoint 1 at 0x201740: file inln.c, line 12.
(gdb) run
result = 10
[Inferior 1 (process 528) exited normally]
```

```
5   #include <stdio.h>
6
7   /* misbp.c's scale_add, minus __attribute__((noinline)):
8      at -O2 it inlines into main and folds to a constant. */
9   double scale_add(int n, double a)
10  {
11      double y = 0.0;
12      for (int i = 0; i < n; i++)
13          y += a;
14      return y;
15  }
16
17  int main(void)
18  {
19      int count = 4;
20      double step = 2.5;
21      double r = scale_add(count, step); /* inlined at -O2 */
22      printf("result = %g\n", r);
23      return 0;
24  }
```

Public

Symptom 3: an inlined function vanishes

- `rocgdb` still knows about inlined code - it was the constant-folding, not inlining itself, that made the breakpoint useless above.
- Rebuilt with a run-time argument, `scale_add` is inlined but not folded:
- the compiler's `-Rpass=inline` report (an `amdclang` flag, not a `rocgdb` command) confirms the inline, and `main` 's disassembly has no call to it;
- yet `break scale_add` does fire - at the inlined copy, hence "`2 locations`";
- `bt` shows the inlined frame (its `#1 main` carries no address of its own).
- At `-O0` , or with `__attribute__((noinline))` , `scale_add` stays a real, callable function and the breakpoint works normally.

```
$ amdclang -ggdb -O2 -Rpass=inline inln2.c -o inln2 2>&1 |\
  grep -v libstdc

inln2.c:19:16: remark: 'scale_add' inlined into 'main' […]
   19 | double r = scale_add(argc + 3, 2.5); /* run-time arg */
      |            ^

$ rocgdb ./inln2
[gdb informational cruft removed]
Reading symbols from ./inln2...
(gdb) break scale_add
Breakpoint 1 at 0x201740: scale_add. (2 locations)
(gdb) run

Breakpoint 1.2, scale_add (n=4, a=2.5) at inln2.c:12
12            for (int i = 0; i < n; i++)
(gdb) bt
#0 scale_add (n=4, a=2.5) at inln2.c:12
#1 main (argc=1, argv=0x7fffffffd148) at inln2.c:19
```

```
5   #include <stdio.h>
6
7   /* inln.c but with a RUN-TIME argument (argc): at -O2 scale_add is
8      inlined but not folded, so gdb can still stop inside it. */
9   double scale_add(int n, double a)
10  {
11      double y = 0.0;
12      for (int i = 0; i < n; i++)
13          y += a;
14      return y;
15  }
16
17  int main(int argc, char **argv)
18  {
19      double r = scale_add(argc + 3, 2.5); /* run-time arg */
20      printf("result = %g\n", r);
21      return 0;
22  }
```

Public

Commands at a glance

Command	What it does
set var <v> = <e>	write a new value into a variable (or through it into memory).
set {<type>}<addr> = <v>	a typed write straight to an address (the write-side of x).
jump <location>	resume execution at <location>: moves the program counter, then continues from <location>, it is not a step.
return	force the current function to return now (optionally with a value).

Public

Setting variables

- `set var <name> = <expr>` : write a new value into a variable.
- A `print` that assigns (`p i = 5`) does the same, as a side effect.
- Uses: force a code path, or repair state and carry on.
- Prefer `set var` over a bare `set` (`set` also changes `rocgdb`'s own settings).
- `set var n = 3` mid-run makes `sum_to` return `6`, not `15`:

```
$ rocgdb ./loop
[gdb informational cruft removed]
Reading symbols from ./loop...
(gdb) break sum_to
Breakpoint 1 at 0x201777: file loop.c, line 10.
(gdb) run
Breakpoint 1, sum_to (n=5) at loop.c:10
10            long s = 0;
(gdb) print n
$1 = 5
(gdb) set var n = 3
(gdb) finish
0x00000000002017f6 in main (argc=1, argv=0x7fffffffd148) at loop.c:19
19            long total = sum_to(n); /* step here to enter sum_to */
Value returned is $2 = 6
```

```
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  long sum_to(int n)
9  {
10      long s = 0;
11      for (int i = 1; i <= n; i++)
12      s += i;
13      return s;
14  }
15
16  int main(int argc, char **argv)
17  {
18      int n = (argc > 1) ? atoi(argv[1]) : 5;
19      long total = sum_to(n); /* step here to enter sum_to */
20      printf("sum 1..%d = %ld\n", n, total);
21      return 0;
22  }
```

Public

Writing memory

- `set var <name> = <expr>` : write a value through a variable.
- `set {<type>}<addr> = <value>` : a typed write straight to an address
- Danger: nothing checks you - easy to corrupt state.
- Fortran arrays are 1-based: `da(1)` is the first element (C's `da[0]`). Below, `da(1)` and `da(2)` become `9` and `7`:

```
$ rocgdb ./memf
[gdb informational cruft removed]
Reading symbols from ./memf...
(gdb) break memf.f90:9
Breakpoint 1 at 0x2233b8: file memf.f90, line 9.
(gdb) run
Breakpoint 1, MEM () at memf.f90:9
9           total = sum(da)
(gdb) print da
$1 = (1.5, 2.5, 3.5)
(gdb) set var da(1) = 9
(gdb) set var da(2) = 7
(gdb) print da
$2 = (9, 7, 3.5)
```

```
5  program mem
6  implicit none
7  real(8) :: da(3) = [1.5d0, 2.5d0, 3.5d0]
8  real(8) :: total
9  total = sum(da)
10  print '(a, f6.2)', "total = ", total
11  end program mem
```

Public

Altering control flow

- `jump <location>` : resume at a different line
  - It does not adjust the stack - easy to misuse; deliberate experiments only.
- Uses:
  - skip a bad call, or
  - re-run a block
- `jump 18` skips the `add` call, so `z` is never set - the program prints garbage:

```
$ rocgdb ./step
[gdb informational cruft removed]
Reading symbols from ./step...
(gdb) break step.c:17
Breakpoint 1 at 0x20176d: file step.c, line 17.
(gdb) run
Breakpoint 1, main () at step.c:17
17              int z = add(x, y); /* next: over, step: into */
(gdb) jump 18
z = -12032
[Inferior 1 (process 608) exited normally]
```

```
5   #include <stdio.h>
6
7   int add(int a, int b)
8   {
9       a = a + b; /* step lands here */
10      return a;
11  }
12
13  int main(void)
14  {
15      int x = 3; /* start stops here */
16      int y = 4;
17      int z = add(x, y); /* next: over, step: into */
18      printf("z = %d\n", z);
19      return 0;
20  }
```

Public

Commands, flags, and the batch/interactive difference

Flag or command	What it does
-batch	run the given commands, then exit.
-ex '<cmd>'	supply one command on the command line (repeatable).
-x <file>	read commands from a file.
set pagination off	do not stop every screenful (essential for logs).
set logging file <f>	choose a log file. . .
set logging enabled on	. . . and start writing output to it (older rocgdb: set logging on).
commands <n> ... end	attach an action list to breakpoint n.
set breakpoint pending on	accept breakpoints on not-yet-loaded code.

- Batch `rocgdb` is just scripted interactive `rocgdb`: the same commands, fed from `-ex` flags or an `-x` file, then it exits
- The one real difference is ordering: set every breakpoint, watchpoint, and action before `run` (or anything that runs past them).
  - there is no prompt to pause at, so you cannot wait and decide later;
  - a breakpoint set after execution has already passed that point simply never fires.
- Put `set pagination off` first - otherwise long output stops for a `--More--` prompt, but as standard input is unconnected in batch mode, it is not possible to respond to the prompt.
- This is why *conditional* breakpoints (see Section "*Advanced topics: more breakpoints, watch, and automation*") matter so much in batch: the filtering you would do by hand at the prompt must be set up in advance in batch mode.

Public

Driving rocgdb without a prompt

- Everything you type interactively can be given up front - on the command line (`-ex '<cmd>'`) or in a file (`-x <file>`).
- Here we catch a crash and print a backtrace, fully non-interactively, on a program that dereferences a null pointer:

```
DEBUGINFOD_URLS= rocgdb -batch -ex run -ex 'bt' \
    --args ./crash 2>&1 | grep -v libthread_db
Program received signal SIGSEGV, Segmentation fault.
0x000000000020177c in deref (p=0x0) at crash.c:9
9             return *p; /* crashes when p is NULL */
#0  0x000000000020177c in deref (p=0x0) at crash.c:9
#1  0x0000000000201750 in main () at crash.c:15
```

```
5   #include <stdio.h>
6
7   static int deref(int *p)
8   {
9       return *p; /* crashes when p is NULL */
10  }
11
12  int main(void)
13  {
14      int *p = NULL;
15      int x = deref(p);
16      printf("%d\n", x);
17      return 0;
18  }
```

- Reading the backtrace: the fault is `SIGSEGV` in `deref` (dereferencing `p = 0x0`), called from `main` at line 15 - the crash site and how it got there, no interaction.
- `-ex '<cmd>'` and `-x <file>` are interchangeable; the same commands in a file (`rocgdb -batch -x commands.txt --args ./crash`) are the basis of the MPI wrapper below.
  - The `grep -v libthread_db` just drops `rocgdb`'s thread-db start-up notice
- to capture everything instead, use two commands
  - `set logging file <f>` to name the log, then
  - `set logging enabled on` to start writing it, and
  - read it offline

Public

Scripted actions at a breakpoint

- `commands <n> ... end` attaches an action list to breakpoint `n` , run automatically each time it fires.
- Shape: start with `silent` (drop the default stop banner), print what you want, end with `continue` so the program carries on.
- With `set pagination off` and logging, this dumps exactly the state you want at each hit - no interaction.

```
DEBUGINFOD_URLS= rocgdb -batch -x dump_loop.gdb \ --args
./loop 2>&1 | grep -v libthread_db


Breakpoint 1 at 0x20178e: file loop.c, line 12.
 i=1, running s=0
 i=2, running s=1
 i=3, running s=3
 i=4, running s=6
 i=5, running s=10
sum 1..5 = 15
[Inferior 1 (process 94597) exited normally]
```

```
 5  break loop.c:12
 6  commands
 7    silent
 8    printf " i=%d, running s=%ld\n", i, s
 9  continue
10  end
11  run
```

Public

Breakpoints on not-yet-loaded symbols

- A breakpoint on code that is not loaded yet
  - a shared library, or a GPU kernel
  - prompts "`make breakpoint pending?`", which you cannot answer in batch
- `set breakpoint pending on` up front accepts the breakpoint now and binds it later, when the code loads
- Or set the environment variable `HIP_ENABLE_DEFERRED_LOADING` to `0` before launching (sh-family: `export HIP_ENABLE_DEFERRED_LOADING=0`):
  - this loads all GPU kernels eagerly, so their symbols exist earlier - simpler, but start-up is slower and uses more memory.
- Gotcha: a breakpoint on an inlined function may report "`cannot evaluate`" or bind at several locations
  - see the inline discussion in Section What optimization does to debugging

Public

Conditional breakpoints and ignore counts

Command	What it does
break <loc> if <cond>	stop only when the condition holds.
condition <n> <cond>	add/replace the condition on breakpoint n.
ignore <n> <count>	skip the next count hits of breakpoint n.

- These shine in batch, where you cannot wait and stop by hand: a condition fires the breakpoint only on the iteration that matters.
- On the summing loop, stop only when the loop counter reaches 3:

```
$ rocgdb ./loop
[gdb informational cruft removed]
Reading symbols from ./loop...
(gdb) break loop.c:12 if i == 3
Breakpoint 1 at 0x20178e: file loop.c, line 12.
(gdb) run
Breakpoint 1, sum_to (n=5) at loop.c:12
12              s += i;
(gdb) print i
$1 = 3
(gdb) print s
$2 = 3
```

```
5   #include <stdio.h>
6   #include <stdlib.h>
7
8   long sum_to(int n)
9   {
10      long s = 0;
11      for (int i = 1; i <= n; i++)
12      s += i;
13      return s;
14  }
15
16  int main(int argc, char **argv)
17  {
18      int n = (argc > 1) ? atoi(argv[1]) : 5;
19      long total = sum_to(n); /* step here to enter sum_to */
20      printf("sum 1..%d = %ld\n", n, total);
21      return 0;
22  }
```

Public

Conditional breakpoints and ignore counts

- `condition <n> <cond>` adds or changes the test on an existing breakpoint; `condition <n>` alone clears it.
- `ignore <n> <count>` skips the next count hits - the quick way to reach "the 1000th iteration" without writing a counter.
- Cost: `rocgdb` re-evaluates the condition in its own interpreter on every hit, so a breakpoint that fires a million times to match once is slow - narrow the location first, or use `ignore` when you just need the Nth hit.

Public

Watchpoints

Command	What it does
watch <expr>	stop when the value of expr changes.
rwatch <expr>	stop when expr is read.
awatch <expr>	stop on read or write.

- A watchpoint finds where a value changes when you do not know which line is to blame - the tool for "who is clobbering this variable?".
- Watch a plain variable `total` , then `awatch` an array element `hist[2]` - `rocgdb` stops on each change, reporting the old and new value:
- Hardware watchpoints (the default where the CPU supports them) are fast but few, and limited in the size they cover; `rocgdb` falls back to software watchpoints, which single-step the program and are far slower.
- Scope: a watchpoint on a local dies when its frame returns (the storage is gone) - `watch` a global, or re-arm it, to follow a value across calls.

Public

Watchpoints

```
$ rocgdb ./watch
[gdb informational cruft removed]
Reading symbols from ./watch...
(gdb) tbreak watch.c:13
Temporary breakpoint 1 at 0x201783: file watch.c, line 13.
(gdb) run
Temporary breakpoint 1, main () at watch.c:13
13                total += i + 1; /* a plain variable that
changes */
(gdb) watch total
Hardware watchpoint 2: total
(gdb) continue
Hardware watchpoint 2: total
Old value = 0
New value = 1 main () at watch.c:14
14                hist[i] = total; /* ...and an array element */
(gdb) delete
(gdb) awatch hist[2]
Hardware access (read/write) watchpoint 3: hist[2]
(gdb) continue
Hardware access (read/write) watchpoint 3: hist[2]
Old value = 0
New value = 6 main () at watch.c:11
11            for (int i = 0; i < 5; i++)
```

```
5   #include <stdio.h>
6
7   int main(void)
8   {
9       int total = 0;
10      int hist[5] = {0};
11      for (int i = 0; i < 5; i++)
12      {
13          total += i + 1; /* a plain variable that changes */
14          hist[i] = total; /* ...and an array element */
15      }
16      printf("total = %d\n", total);
17      return 0;
18  }
```

Public

Catchpoints, display, and tracepoints

Command	What it does
catch <event>	stop on an event: throw, syscall, fork, signal, library load.
display <expr>	re-print expr automatically at every stop.
trace / collect / actions	record data at a point without stopping (tracepoints).

- A catchpoint stops on an event rather than a code location. `catch signal SIGSEGV` stops the instant a fault is raised - here on the null-deref program:
- `catch throw` stops on a thrown C++ exception; `catch syscall` , `catch fork` , `catch load` do the same for syscalls, forks, and library loads.
- `display <expr>` re-prints `<expr>` automatically at every stop - set it once, then just `step` or `continue` and watch the value evolve.
- Tracepoints (`trace` a location, `collect` the data, `tstart`) record without stopping the program - for "cannot afford to pause it" cases; they need `gdbserver` and are beyond this course.

```
$ rocgdb ./crash
[gdb informational cruft removed]
Reading symbols from ./crash...
(gdb) catch signal SIGSEGV
Catchpoint 1 (signal SIGSEGV)
(gdb) run
Catchpoint 1 (signal SIGSEGV),
0x000000000020177c in deref (p=0x0) at crash.c:9
9             return *p; /* crashes when p is NULL */
```

```
5   #include <stdio.h>
6
7   static int deref(int *p)
8   {
9       return *p; /* crashes when p is NULL */
10  }
11
12  int main(void)
13  {
14      int *p = NULL;
15      int x = deref(p);
16      printf("%d\n", x);
17      return 0;
18  }
```

Public

The TUI: a full-screen source view

- `rocgdb` has a terminal UI: a split-screen mode showing source (or registers, or disassembly) above the command prompt.
- A different interface, not new functionality - the same `rocgdb` underneath (which is why it sits here, at the end).
- Terminal-native - it works over a plain SSH session; GUI front-ends that need X11 forwarding are out of scope for this course.

Command	What it does
rocgdb --tui / tui enable	a split terminal view: source on top, prompt below.
layout src / asm / split / regs	choose the TUI panes.

- Toggle it live with `Ctrl-x a` ; `Ctrl-x o` moves focus between panes; `Ctrl-L` redraws if program output corrupts the screen.
- Being interactive, it is best seen live: try `rocgdb --tui ./demo` yourself.

Public

Scripting rocgdb in Python (provisional)

- `rocgdb` embeds a full Python interpreter: code in `~/.gdbinit` (or loaded with `source <file>`) can define new commands and walk frames, variables, and registers.
- Teaser: `python print(gdb.parse_and_eval("s"))` reads a program variable from Python; a `gdb.Command` subclass turns a script into a new `(gdb)` command you can call like any built-in (the Python module stays `gdb` in `rocgdb`).
- We only introduce the idea here - the natural demonstration is register-level decoding, a GPU topic deferred to Part II (targeting MI300A).

Public

Exercises

- The exercises live in AMD's HPCTrainingExamples repository - clone it and work through the `README` in each exercise directory:

```
$ git clone https://github.com/amd/HPCTrainingExamples.git
$ cd HPCTrainingExamples/Rocgdb/rocgdb_training/exercises # exact path TBD
$ cd common && make # build the shared sample programs once
$ cd ../01_running && less README.md # then work through each exercise
```

- Each exercise is self-contained and independent - do them in any order, and skip any you like.
- On Hunter, `git clone` and `pip install` (exercise `06_mdb`) need the site HTTPS proxy: set `https_proxy` first (see the HLRS docs at https://kb.hlrs.de/platforms/index.php/SSH_Tunnel_with_Proxy), or clone elsewhere and copy the tree across.

Public

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.

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


AMD