

Linaro DDT - A Parallel Debugger at Scale

Rudy Shand
Linaro Ltd

HPC development Solutions from Linaro

Build reliable and optimized code on multiple Server and HPC architectures

Linaro Forge combines



Linaro DDT

Market leading, simple to use
HPC debugger for C/C++,
Fortran and Python
applications.



Linaro MAP

Effortless performance
analysis for experts and
novices alike.



Linaro Performance Reports

At a glance, single-page, application
performance summary.

**Performance
Engineering for any
architecture, at any
scale**

gdb under the hood

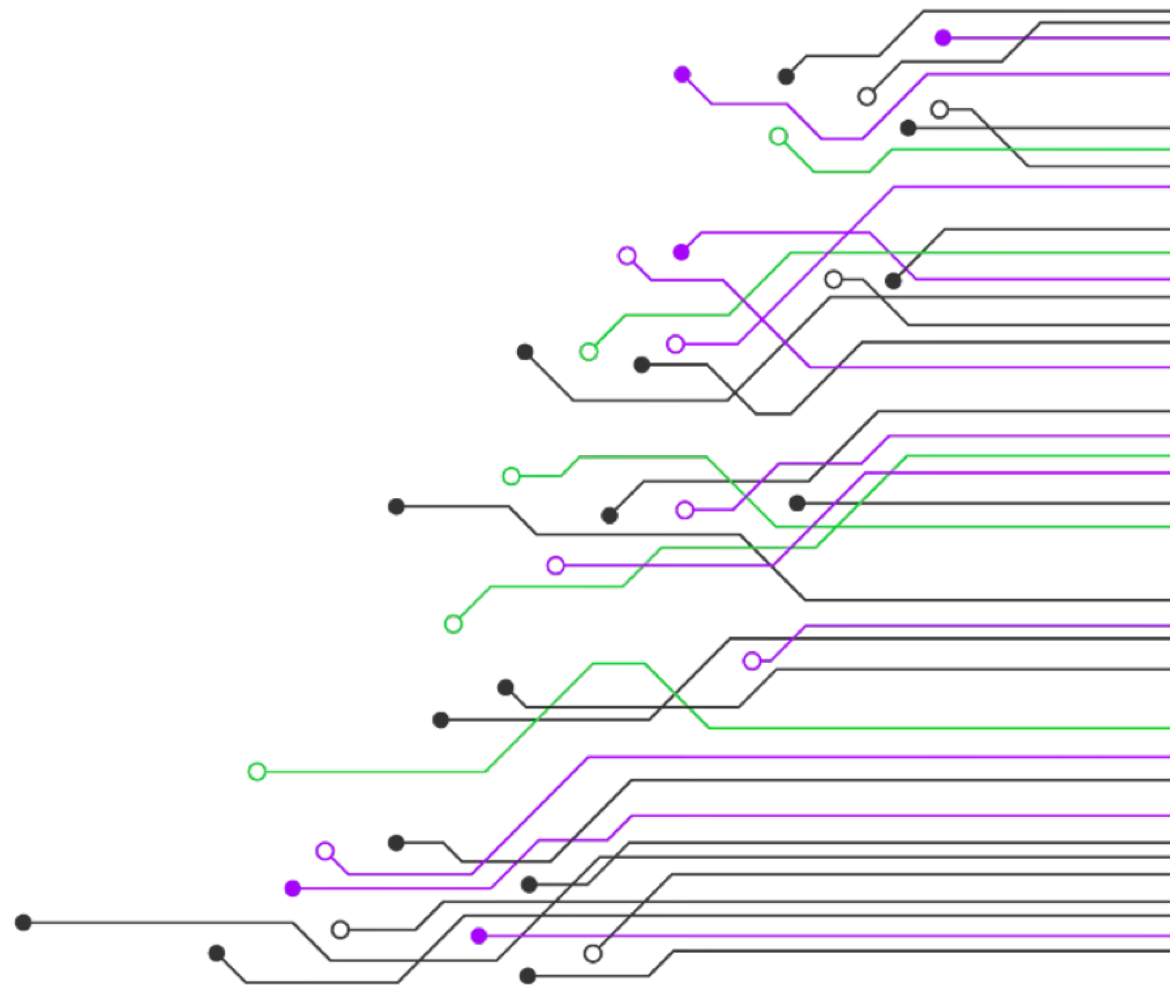
Leveraging the gdb community

GDB is the underlying technology

- Linaro upstreams patches to the gdb community
- Forge team raises and fixes gdb bugs

Nimble at supporting new technologies

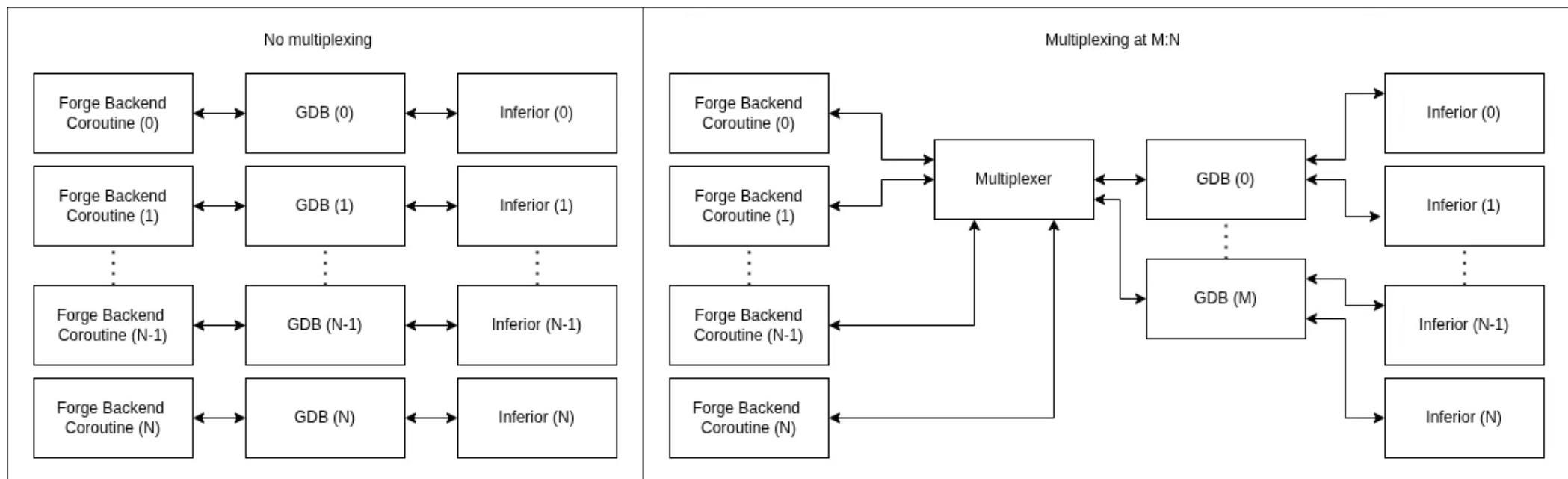
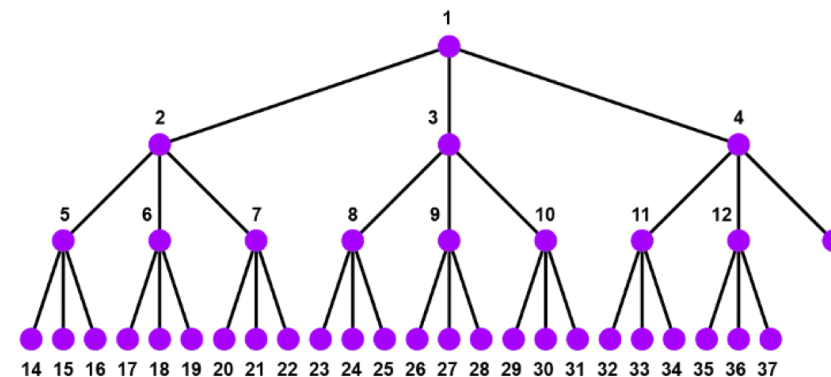
- Rely on hardware vendors to add gdb support
- Rely on software consortiums to add gdb support
- Helps us to stay current with technologies
- Provides a state of the art debugger



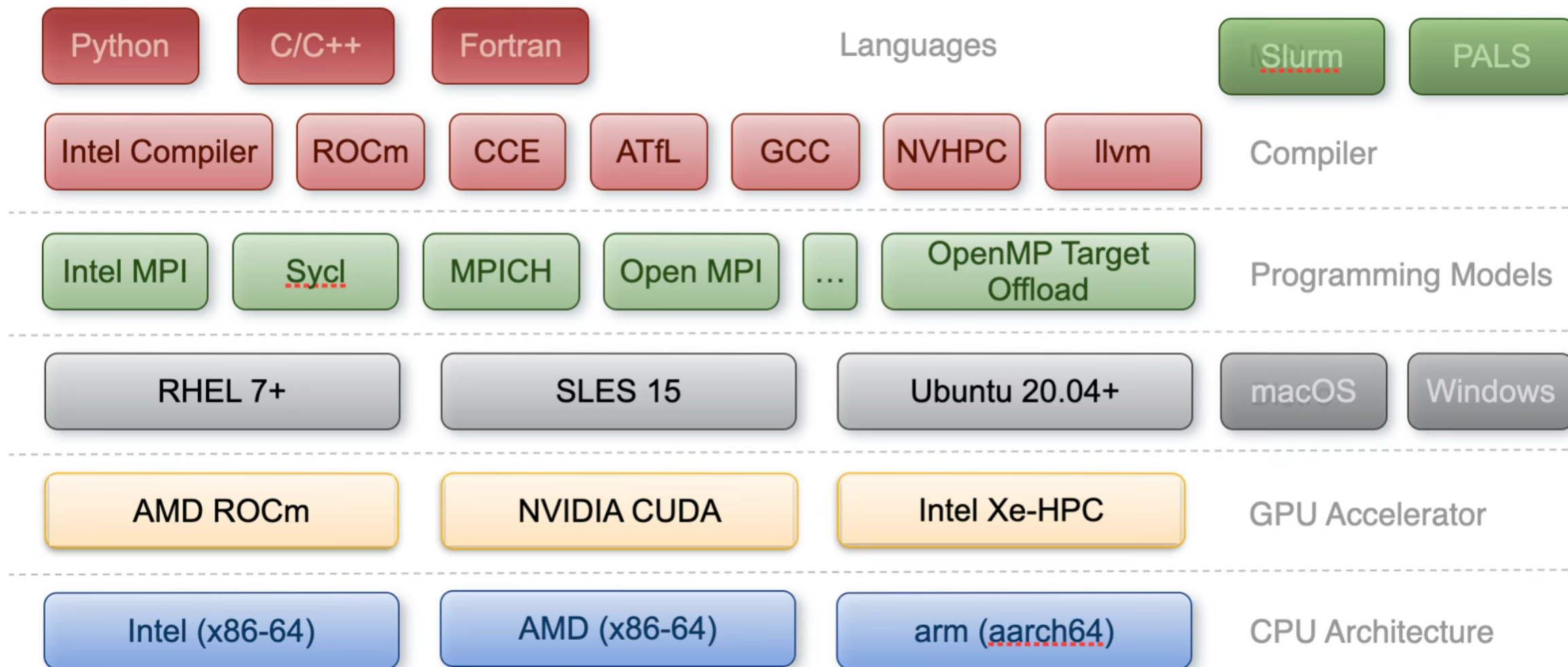
Scaling gdb

Tree network topology

- Tree server design is how Forge is able to scale
- Send bulk commands and merge responses
- Aggregate the data instead of broadcasting thousands of responses
- Multiplex for tooling memory reduction (up to 80% reduction)



DDT Supported platforms



DDT User Interface

- 1 Process controls
- 2 Process groups
- 3 Source Code view
- 4 Variables
- 5 Evaluate window
- 6 Parallel Stack
- 7 Project files
- 8 Find a file or function

Compile with -g -O0

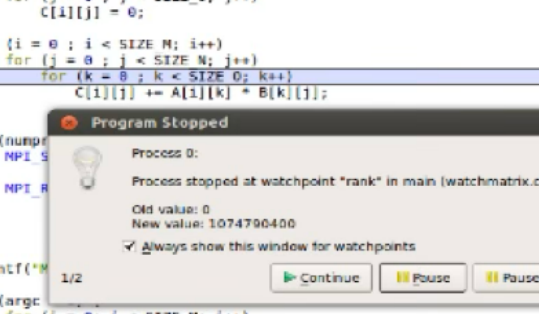
The screenshot displays the DDT (Data Display Tool) User Interface, which is used for debugging and monitoring parallel applications. The interface is divided into several panels:

- 1 Process controls:** Located at the top, it includes a toolbar with icons for running, pausing, and stepping through code, as well as a status bar showing the current group and process.
- 2 Process groups:** A table below the process controls showing the status of different process groups. It includes columns for the group name, the number of processes, and the status (Paused, Playing, Finished).
- 3 Source Code view:** The central pane showing the source code of the application being debugged. It includes a search bar and a list of project files on the left.
- 4 Variables:** A panel on the right showing the current values of variables in the scope of the selected line of code. It includes a table with columns for the variable name and its value.
- 5 Evaluate window:** A panel at the bottom right used for evaluating expressions and viewing the results. It includes a text area for input and a table for output.
- 6 Parallel Stack:** A panel at the bottom left showing the stack of processes and functions. It includes a table with columns for the process ID and the function name.
- 7 Project files:** A panel on the left showing the project files and their locations. It includes a search bar and a tree view of the project structure.
- 8 Find a file or function:** A panel on the left used for finding files and functions. It includes a search bar and a list of results.

DDT Highlights

Input/Output	Breakpoints	Watchpoints	Tracepoints	Tracepoint Output	Stacks (All)
Tracepoint Output					
Tracepoint	Processes	Values logged			
vhone f90 85	976, ranks 12, 14-17, 22-23, 12...	mtype  2172-3527	jcol:  2-83	mod 	pey 
vhone f90 81	960, ranks 12, 14-17, 22-23, 12...	ls  1 kmax 	pez 		
vhone f90 85	942, ranks 12, 14-17, 22-23, 12...	mtype  2172-3527	jcol:  2-83	mod 	pey 
vhone f90 81	929, ranks 12, 14-17, 22-23, 12...	ls  1 kmax 	pez 		
vhone f90 85	919, ranks 12, 14-17, 22-23, 12...	mtype  2172-3527	jcol:  2-83	mod 	pey 
vhone f90 81	898, ranks 12, 14-17, 22-23, 12...	ls  1 kmax 	pez 		

The scalable print alternative



The screenshot shows a C program with nested loops and a watchpoint. The code is as follows:

```

for (i = 0; i < SIZE_M; i++)
    for (j = 0; j < SIZE_O; j++)
        C[i][j] = 0;

for (i = 0; i < SIZE_M; i++)
    for (j = 0; j < SIZE_N; j++)
        for (k = 0; k < SIZE_O; k++)
            C[i][j] += A[i][k] * B[k][j];

```

The program is running in a debugger, and a watchpoint has been set on the variable `rank` in the file `watchmatrix.c` at line 45. The watchpoint has triggered, and the program has stopped at the line `Process stopped at watchpoint "rank" in main (watchmatrix.c:45).`

The **Program Stopped** dialog box shows the following information:

- Process 0:** Process stopped at watchpoint "rank" in main (watchmatrix.c:45).
- Old value:** 0
- New value:** 1074790400
- ☒ Always show this window for watchpoints

The dialog box also has buttons for **Continue**, **Pause**, and **Pause All**.

Stop on variable change

The screenshot shows a C program in a debugger. The code defines a function `func3` that takes a `void*` argument, increments it, and then dereferences it. A warning is displayed, stating that the behavior is undefined because the pointer is of type `void*`. The warning text is: "portability" is of type 'void *'. When using void pointers in calculations, the behaviour is undefined. Below the warning, a message says: "Left click to add a breakpoint on line 50". The code is as follows:

```

hello.c
43     else
44         test=-1;
45     }
46
47 void func3()
48 {
49     void* i = (void*) 1;
50     while(i++ || !i)
51         free((void*)i);
52
53     return test;
54 }
55 {
56     typeThree test;
57     typeThree* t2;
58     int i;

```

Static analysis warnings on code errors

Memory Debugging

```
# Enable reading of debug environment variables:
host:~/demo> export NEOReadDebugKeys=1

# Disable RTLD_DEEPPBIND, so that malloc/free calls bind to
# our memory debug library, instead of glibc:
host:~/demo> export NEODisableDeepBind=1
```

The screenshot shows a C program in a text editor. The code is as follows:

```
if (argv[i] && !strcmp(argv[i], "crash")) {  
    argv[i] = 0;  
    printf("%s", *(char **)argv[i]);  
    /* we shall see you later */  
}  
  
func1();  
  
func2();  
fprintf(stderr, "I  
beingWatched = 1;  
  
test.anotherList.s  
test.c = 'p';  
beingWatched = 0;
```

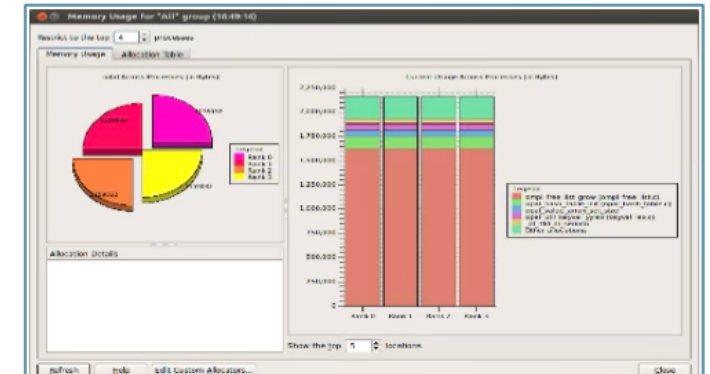
A Windows error dialog box titled "Program Stopped" is overlaid on the code. It contains the following text:

Processes 0-3:
Memory error detected in main (hello.c:118):
null pointer dereference or unaligned memory access

Note: the latter may sometimes occur spuriously if guard pages are enabled
Tip: Use the stack list and the local variables to explore your program's current state and identify the source of the error.

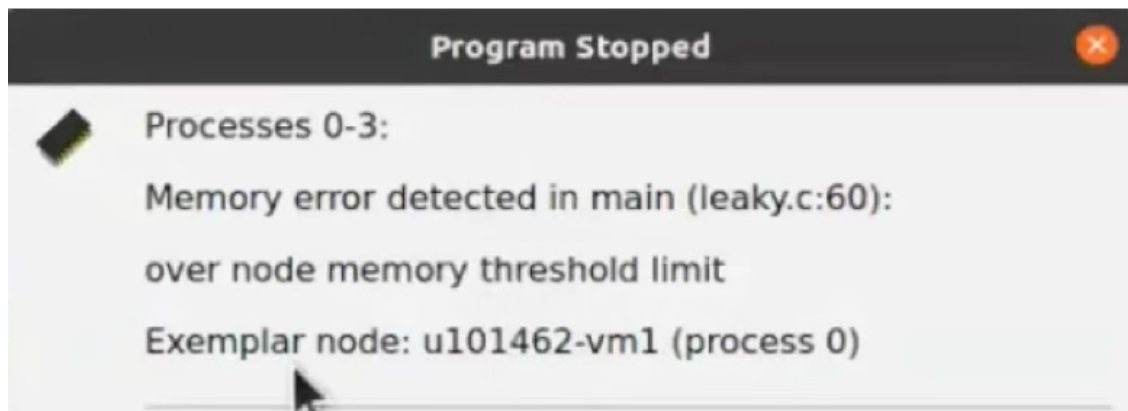
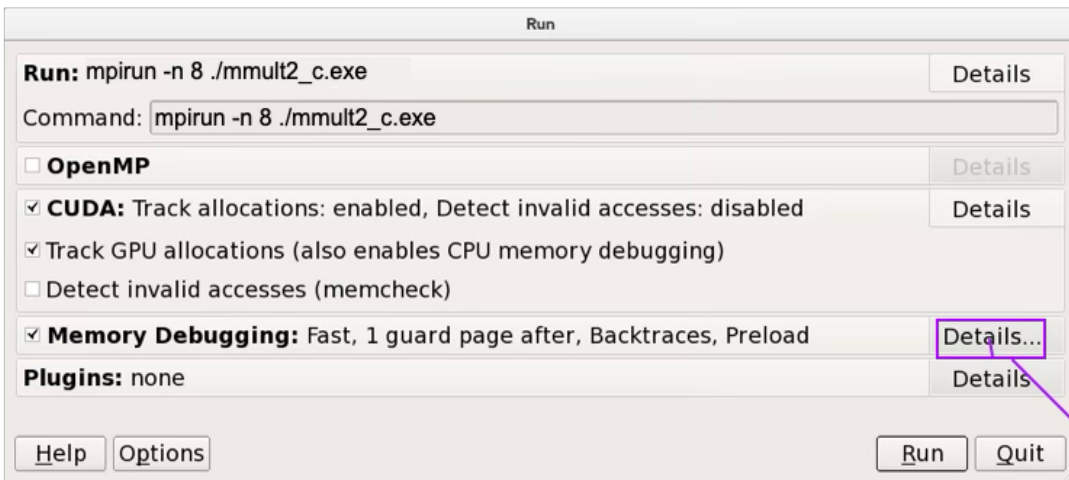
At the bottom of the dialog are two buttons: "Continue" and "Pause".

Detect read/write beyond array bounds

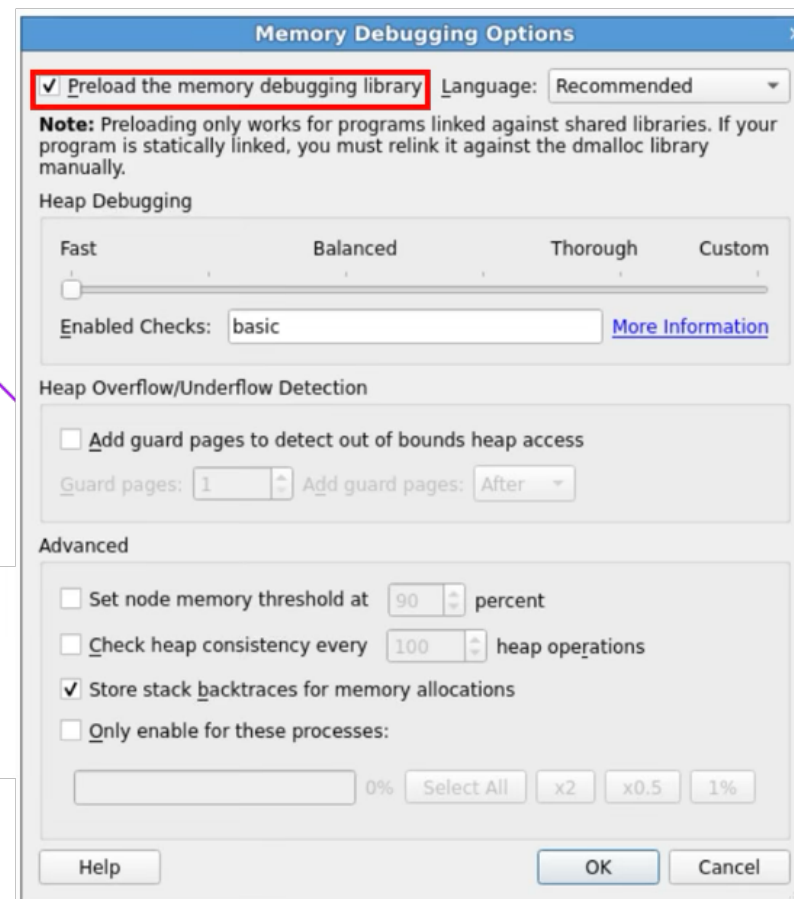


Detect stale memory allocations

Memory Debugging



When manual linking is used,
untick "Preload" box



MDA Viewer

What does your data look like at runtime?

View arrays

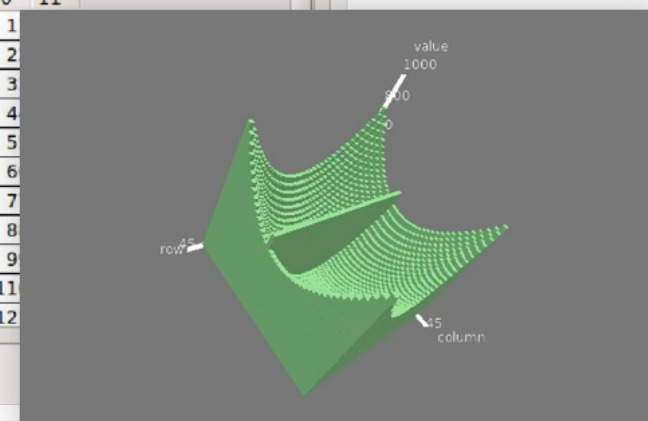
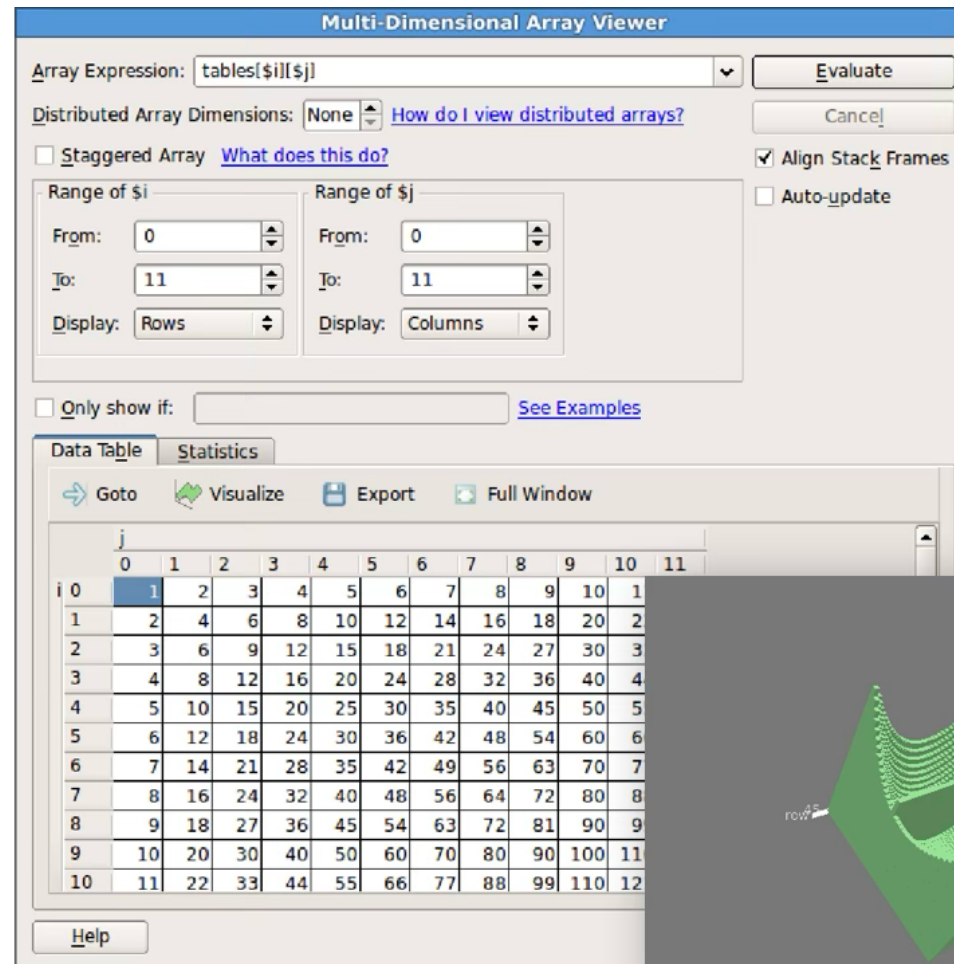
- On a single process
- Or distributed on many ranks

Use metavariables to browse the array

- Example: \$i and \$j
- Metavariables are unrelated to the variables in your program
- The bounds to view can be specified
- Visualise draws a 3D representation of the array

Data can also be filtered

- “Only show if”: \$value>0 for example \$value being a specific element of the array



GPU Debugging Terminology

	NVIDIA GPU	AMD GPU	Intel GPU
Name	CUDA	ROCm	Xe
Language	CUDA C/C++ and Fortran	HIP	SYCL
Execution Unit	Warp	Wavefront	Sub-group
EU Size	32 Threads	64 Threads	8, 16 or 32 Threads
EU GDB	...	GDB Thread	GDB Thread
EU Thread	...	Lane	Lane (work-item)
Forge GPU Thread	Lane	Lane	Lane

Debugging Intel Xe GPUs

Using Linaro DDT

Debug code simultaneously on the GPU and the CPU

Controlling the GPU execution:

- All active threads in a Sub-group will execute in lockstep. Therefore, DDT will step 16 threads at a time.
- Play/Continue runs all GPU threads
- Pause will pause a running kernel

Key (additional) GPU features:

- Kernel Progress View
- GPU thread in parallel stack view
- GPU Thread Selector
- GPU Device Pane

Kernels must be compiled with the -g and -O0 flags

The screenshot displays the Linaro DDT interface for debugging Intel Xe GPUs. The top panel shows a grid of GPU threads (0-61) and a 'Show processes' button. Below this, the 'Project Files' pane on the left lists source files, with 'matrix_mul_sycl.cpp' selected. The central pane shows the C++ source code for a matrix multiplication kernel, with line 66 highlighted. The right pane shows the 'GPU Devices' section, listing 'Intel(R) Data Center GPU Max 1550' with 1 ID and 448 Cores. The bottom pane features the 'Kernel Progress View' with a progress bar for the 'main::(lambda(auto:...))' kernel, and an 'Evaluate' section showing variable values: M=150, N=300, P=600.

Debugging Nvidia GPUs

Using Linaro DDT

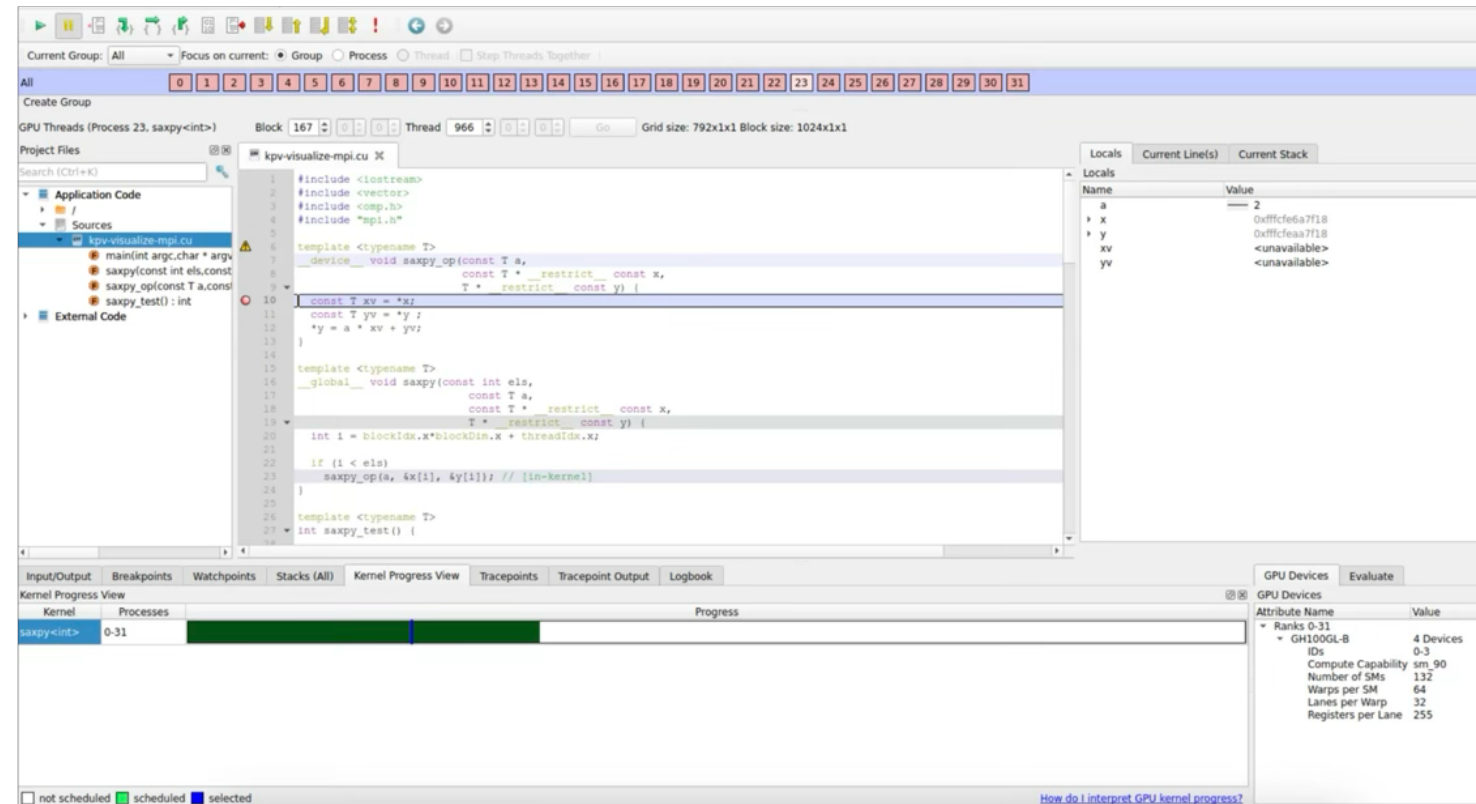
Controlling the GPU execution:

- All active threads in a warp will execute in lockstep
Therefore, DDT will step 32 threads at a time.
- Play/Continue runs all GPU threads
- Pause will pause a running kernel

Key (additional) GPU features:

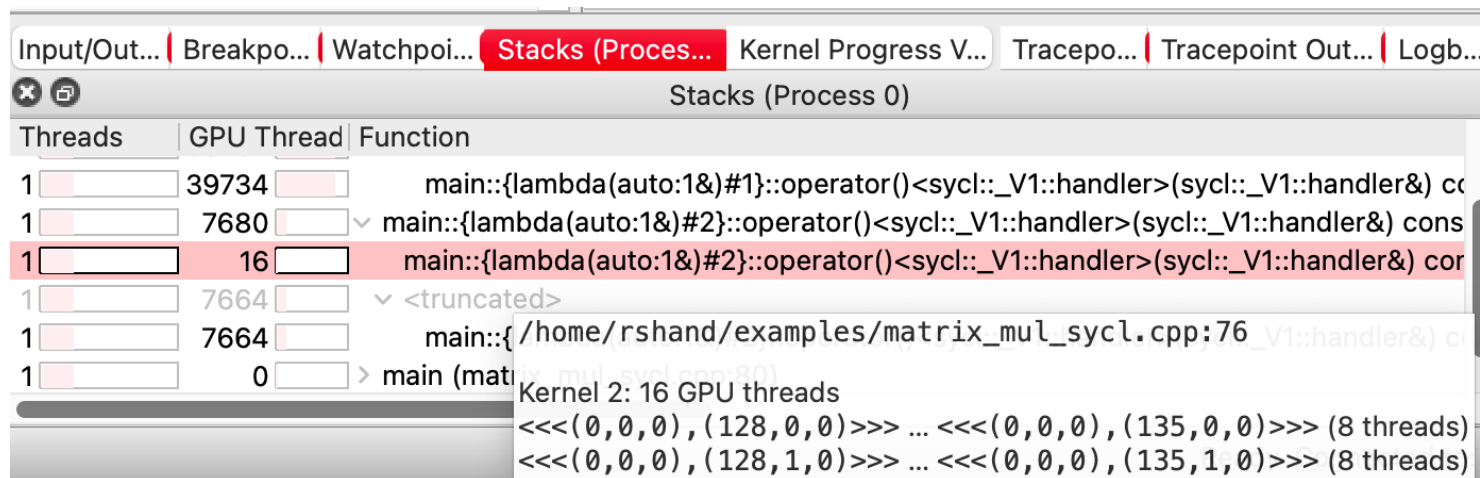
- Kernel Progress View
- GPU thread in parallel stack view
- GPU Thread Selector
- GPU Device Pane

For NVIDIA's nvcc compiler, kernels must be compiled with the -g and -G flags



Parallel Stack view

Display location and number of threads



The screenshot shows the 'Stacks (Process 0)' window with the following table:

Threads	GPU Thread	Function
1	39734	main::{lambda(auto:1&)#1}::operator()<sycl::_V1::handler>(sycl::_V1::handler&) cc
1	7680	main::{lambda(auto:1&)#2}::operator()<sycl::_V1::handler>(sycl::_V1::handler&) cons
1	16	main::{lambda(auto:1&)#2}::operator()<sycl::_V1::handler>(sycl::_V1::handler&) cor
1	7664	<truncated>
1	7664	main::{ /home/rshand/examples/matrix_mul_sycl.cpp:76 _V1::handler&) c
1	0	> main (mat

Below the table, the following text is displayed:

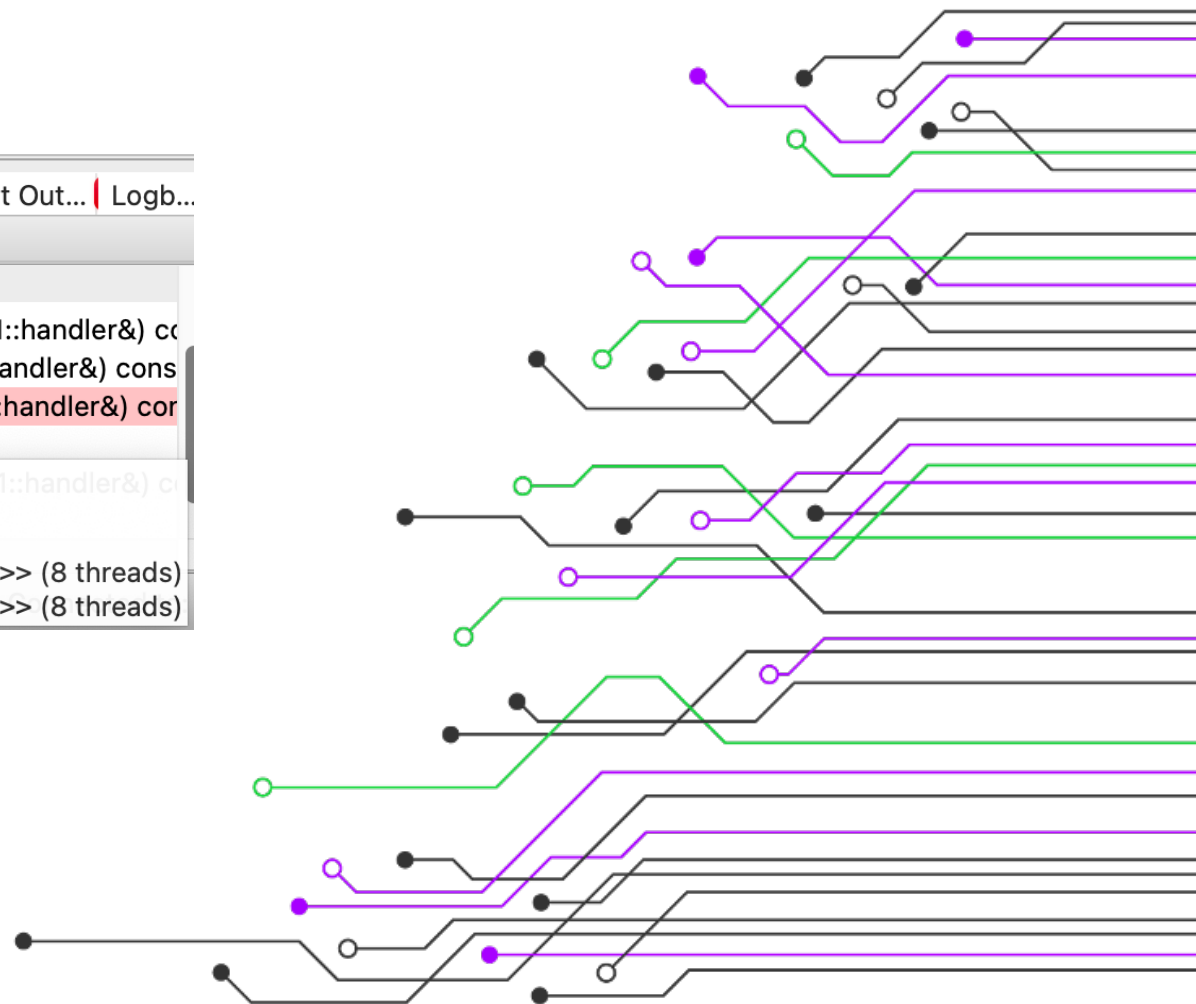
Kernel 2: 16 GPU threads
<<<(0,0,0),(128,0,0)>>> ... <<<(0,0,0),(135,0,0)>>> (8 threads)
<<<(0,0,0),(128,1,0)>>> ... <<<(0,0,0),(135,1,0)>>> (8 threads)

Click stack item

- Select GPU Thread
- Update variable display
- Move Source Code Viewer

Tooltip displays

- GPU Thread Ranges
- Size of each range



Case Study: Vienne Supercomputing Center (VSC)

Developing a GPU version of an already existing serial algorithm (Marching Tetrahedra). <https://github.com/HliasGit/tetra>

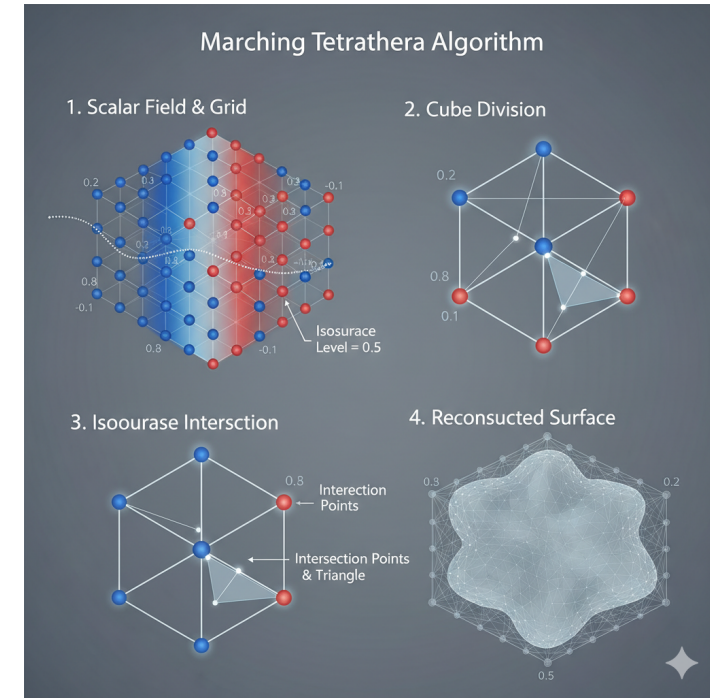
- The aim of the algorithm is to extract the approximation of an iso-surface from a scalar field.
- The scalar field (whose dimensions are DX,DY,DZ) is represented as a 3D tensor (of floats)
- Linearise (flatten) the scalar field to use it in a simpler way on GPU

Used three kernels to develop the GPU version:

- The first one is a preprocessing step to reduce the computational complexity.
- Second and the third ones do the needed computation for the algorithm to work.

For each point in the scalar field, it needs to form a "cube" by accessing its 8 neighbours. Due to the Z-major linearization, only the neighbor in the Z-direction (index + 1) is adjacent in memory. Accessing neighbors in the X-direction (e.g., index + $DZ \cdot DY$) or Y-direction (e.g., index + DZ) requires calculated memory offsets.

This approach led to "CUDA illegal memory access" errors. The problem was a failure to correctly handle boundary conditions. When a thread processed a point on the "edge" of the 3D domain (like at $x = DX-1$ or $y = DY-1$), these offset calculations were trying to access memory outside the allocated array. This boundary-checking logic was not intuitive, used Linaro DDT debugger to pinpoint these out-of-bounds access.



Spotting illegal memory access with Linaro DDT

The screenshot displays the Linaro DDT (Data Display Tool) interface, which is used for debugging GPU applications. The main window shows the source code of a CUDA program, `test_optimized.cu`, with a breakpoint set at line 40. The program is running on a GPU with a grid size of 51222x1x1 and a block size of 1024x1x1. The current thread is 704. A "Program Stopped" dialog box is open, indicating that the program has stopped at a breakpoint in the `remove_unnecessary_cubes_SoA_kernel` function. The dialog box provides details about the thread and the breakpoint location.

Program Stopped

Process 0:
Kernel 1 Thread <<<(0,0,0).(704,0,0)>>> stopped at breakpoint in
remove_unnecessary_cubes_SoA_kernel<<<(51222,1,1).(1024,1,1)
>>> (marching_tetrahedron_optimized.cu:40).

☒ Always show this window for user-defined breakpoints

Locals

Name	Value
grid	0x1554f8000000
counter	0x1554f6000000
size	52450896
threshold	0.000399999998989515007
d_relevant_cubes	0x1554f6000200
j	2
s_mem_2	0x155536001004
all_out	false
k	86
s_mem_3	0x155536002008
i	704
mem_1	0x155536000000
mem_4	0x15553600300c
in	false

Stacks

Threads	GPU Thread	Function
1	0	main (test_optimized.cu:67)
1	86016	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:7)
1	256	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:36)
1	1632	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:39)
1	12256	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:40)
1	288	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:41)
1	50336	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:44)
1	2560	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:45)
1	992	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:49)
1	17696	remove_unnecessary_cubes_SoA_kernel (marching_tetrahedron_optimized.cu:51)

Python Debugging

- Debug Features

- Sparklines for Python variables
- Tracepoints
- MDA viewer
- Mixed language support

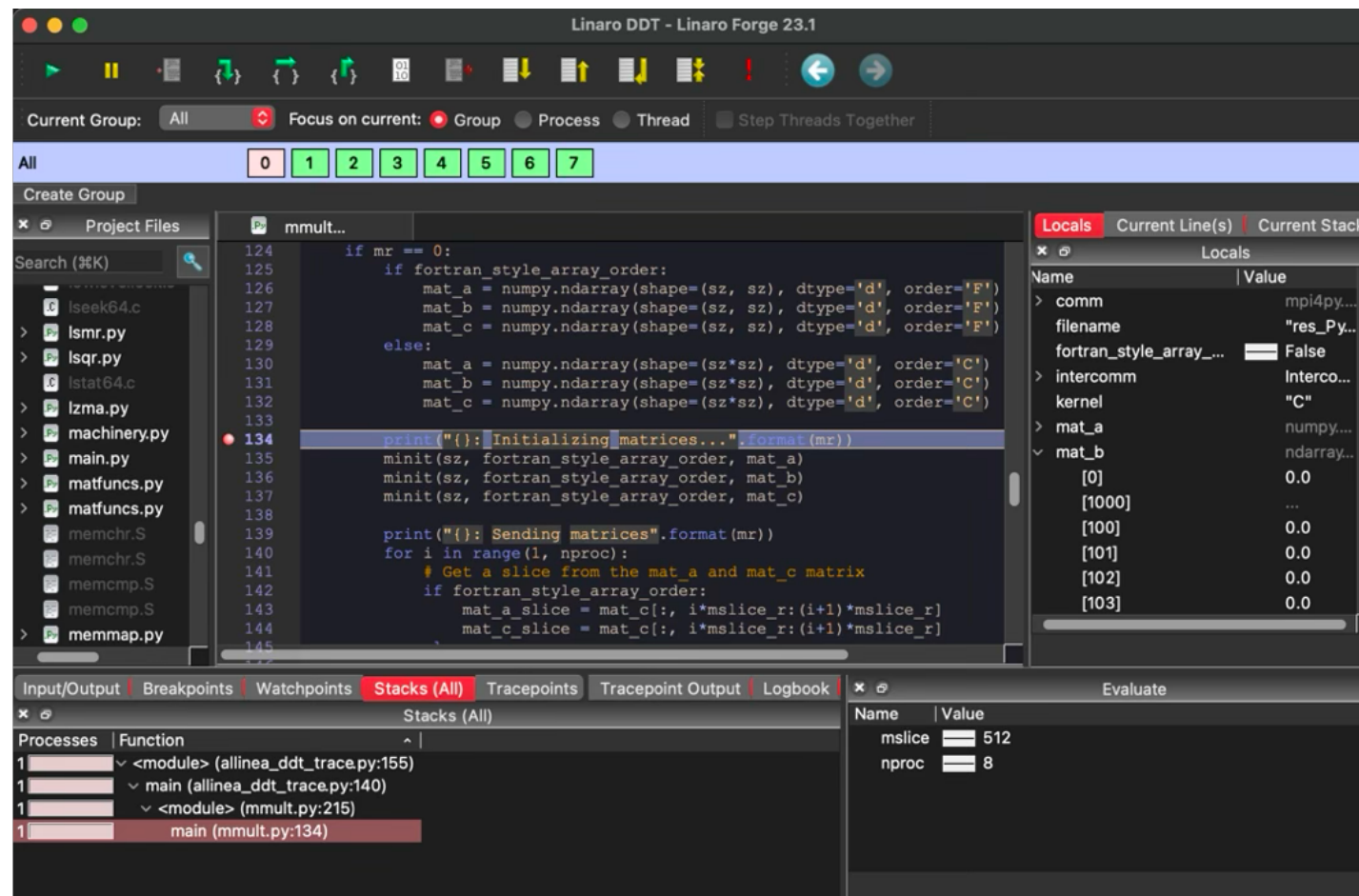
- Improved Evaluations:

- Matrix objects
- Array objects
- Pandas DataFrame
- Series objects

- Python Specific:

- Stop on uncaught Python exception
- Show F-string variables in “Current Line” display
- Mpi4py, NumPy, SciPy

```
ddt --connect mpirun -n 8 python3  
%allinea_python_debug% ./mmult.py
```



DDT offline mode

Run the application under DDT and halt or report when a failure occurs

You can run the debugger in non-interactive mode

- For long-running jobs / debugging at very high scale
- For automated testing, continuous integration...

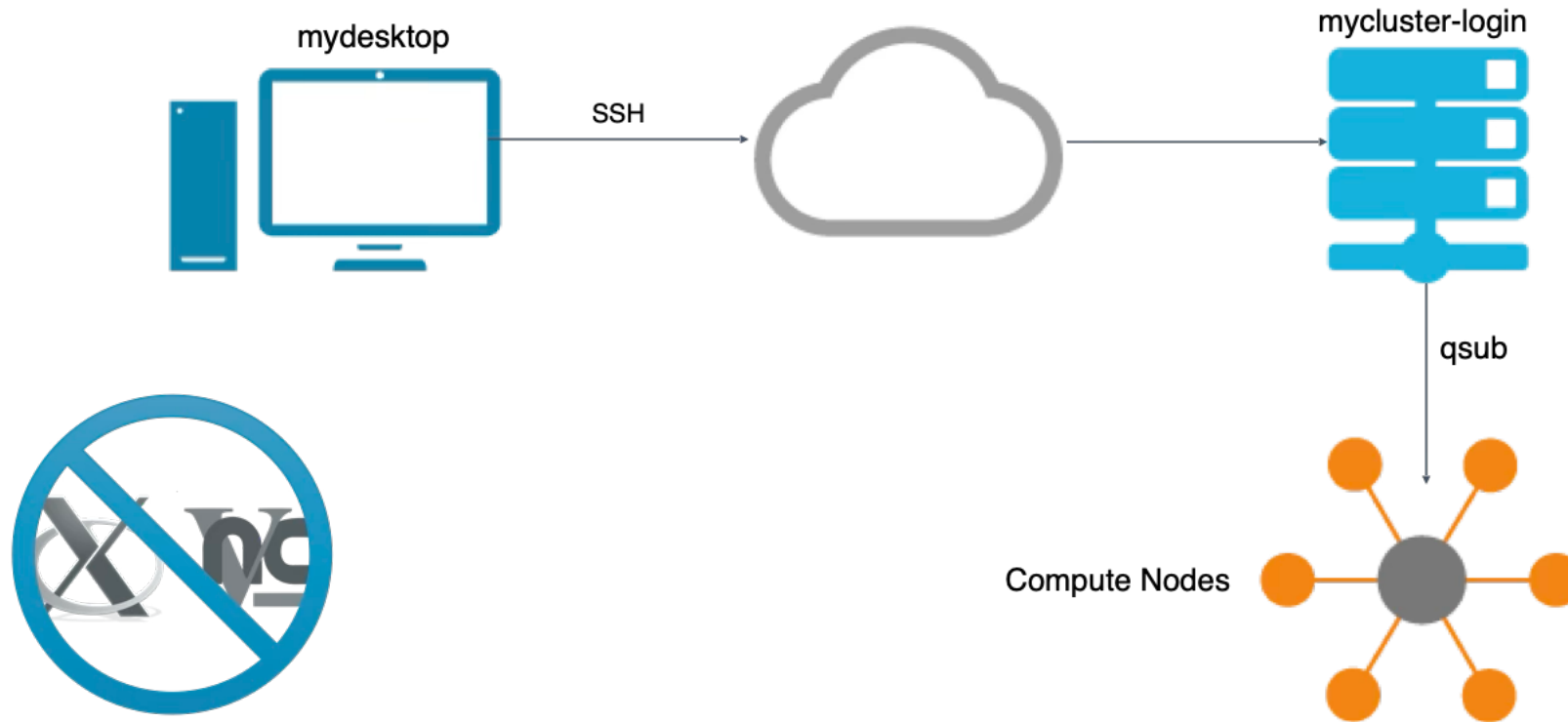
To do so, use following arguments:

- `$ ddt --offline --output=report.html mpirun ./jacobi_omp_mpi_gnu.exe`
 - **--offline** enable non-interactive debugging
 - **--output** specifies the name and output of the non-interactive debugging session (HTML or Txt)
 - Add **--mem-debug** to enable memory debugging **and memory leak detection**

```
ddt --offline -o jacobi_omp_mpi_gnu_debug.txt \  
    --trace-at _jacobi.F90:83,residual \  
    mpirun ./jacobi_omp_mpi_gnu.exe
```

The Forge GUI and where to run it

Forge provides a powerful GUIs that can be run in a variety of configurations



Remote connection to Aurora

The image shows the Linaro Forge application window and a 'Remote Launch Settings' dialog box.

Linaro Forge Interface:

- Top bar: Linaro DDT - Linaro Forge 23.1
- Left sidebar: Linaro Forge logo, Linaro DDT icon, Linaro MAP icon, and links for 'Get trial licence', 'Support', 'linaroforge.com', and 'Remote Client ?'.
- Main area: A list of actions: 'RUN' (Run and debug a program.), 'ATTACH' (Attach to an already running program.), 'OPEN CORE' (Open a core file from a previous run.), and 'MANUAL LAUNCH (ADVANCED)' (Manually launch the backend yourself.). Below these is an 'OPTIONS' section with a 'Remote Launch:' button labeled 'Configure...' which is highlighted with a red rectangle.
- Bottom bar: 'QUIT' button.

Remote Launch Settings Dialog:

- Connection Name: `aurora`
- Host Name: `rshand@login.aurora.alcf.anl.gov` (with a dropdown arrow)
- Remote Installation Directory: `/lus/flare/projects/Tools/jkwack/Linaro/forg/25.0.1`
- Remote Script: Optional
- Private Key: Optional (with a folder icon)
- KeepAlive Packets: ☐ Always look for source files locally
- Interval: 30 seconds
- ☒ Proxy through login node
- Buttons: 'Test Remote Launch', 'Help', 'OK', and 'Cancel'.

Cheat sheet

Training material

1. Getting the examples

```
cp /flare/ATPESC2026/EXAMPLES/linaro-ddt/linaro-forge-training.tar.gz  
tar -xf linaro-forge-training.tar.gz
```

2. Set the path to the forge training folder

```
export FORGE_TRAINING=<path_to_training_folder>
```

Forge Client (On local machine)

Install Forge client <https://www.linaroforge.com/downloadForge>

Running with a batch script

```
qsub $FORGE_TRAINING/submit.sh
```

Interactive Session

```
qsub -l -l select=1 -l walltime=01:00:00 -l  
filesystems=home:flare -A ATPESC2026 -q ATPESC
```

Forge commands

<code>ddt --connect</code>	<i># Reverse connect</i>
<code>ddt --offline</code>	<i># Run DDT without GUI</i>
<code>map --profile</code>	<i># Profile without GUI</i>
<code>perf-report</code>	<i># Generate Performance Report</i>

Guides

[*DDT userguide*](#)

[*Debugging on Aurora*](#)

[*Hands-on*](#)

Debugging Intel Xe GPUs on Aurora

```
cd $FORGE_TRAINING/correctness/gpu-intel-mmult
```

```
mpicxx --intel -fsycl -I$ONEAPI_ROOT/dev-utilities/latest/include -g -O0 matrix_mul_sycl.cpp -o matrixmul
```

```
qsub -l -l select=1 -l walltime=01:00:00 -l filesystems=home:flare -A ATPESC2026 -q ATPESC
```

<https://docs.alcf.anl.gov/aurora/debugging/ddt-aurora/#invoking-the-ddt-server-from-aurora>

```
ddt --np=24 --connect --mpi=generic --mpiargs="--ppn 12 --envall" ./matrixmul
```

ARGONNE ATPESC2026 EXTREME - SCALE COMPUTING

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.

The U.S. Government retains for itself and others acting on its behalf a nonexclusive, royalty-free license in this video, with the rights to reproduce, to prepare derivative works, and to display publicly.