TotalView Training - NERSC

MAY 13, 2024

LBL/NERSC Agenda – May 2024

- Introduction
- Latest TotalView Features
- TotalView Roadmap
- Remote Debugging Techniques
- Review of General Debugging Features
- TotalView Debugging at NERSC Best Practices
- GPU Debugging with TotalView on Perlmutter (10:00am)
- MPI and OpenMP debugging
- Reverse Debugging
- Memory Debugging
- Common TotalView Usage Questions
- Q&A

Introductions

- Bill Burns (Senior Director of Software Engineering and Product Manager)

  bburns@perforce.com

- John DelSignore (TotalView Chief Architect)

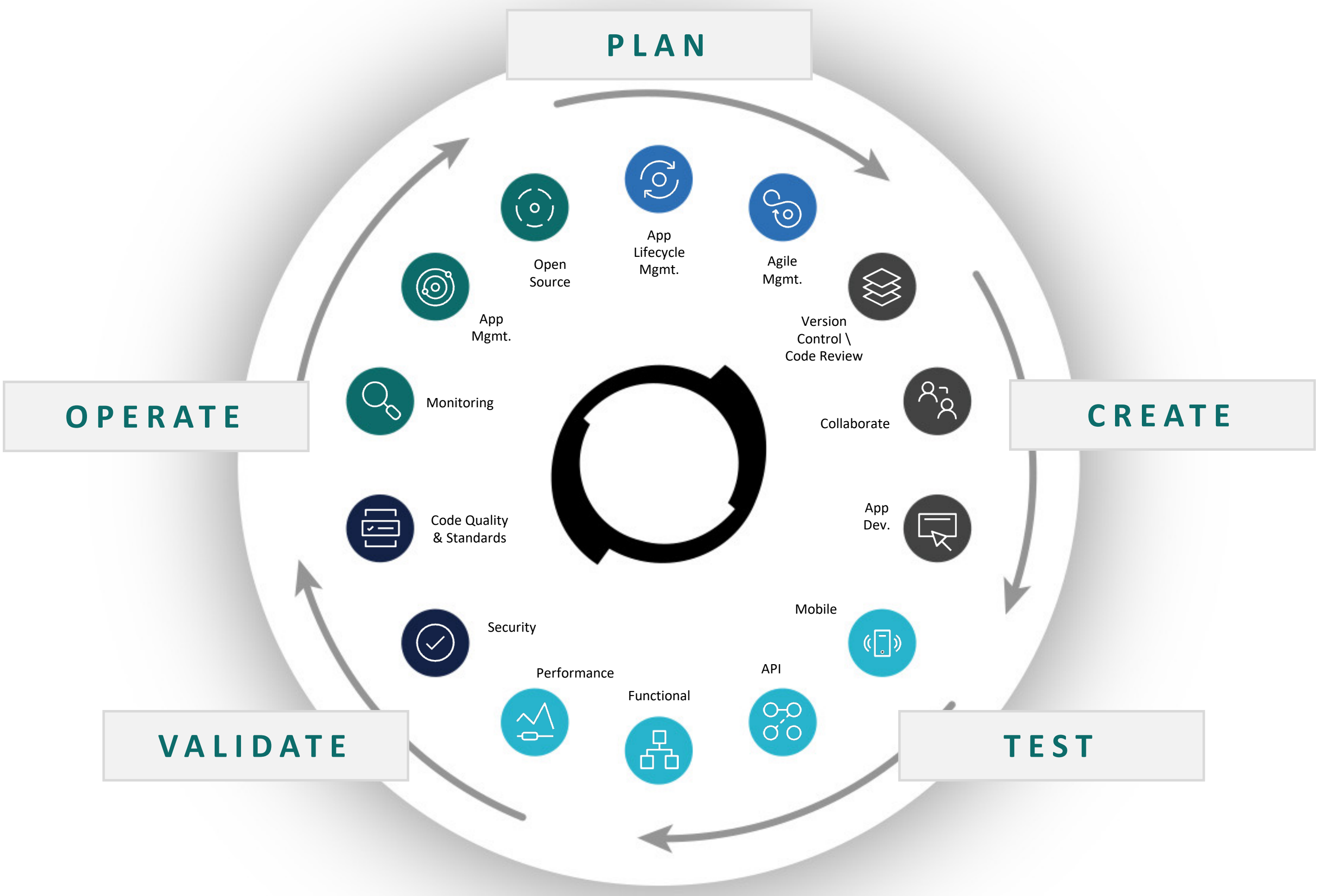
  jdelsignore@perforce.com

- Larry Edelstein (Manager, Sales Engineering, TotalView)

  ledelstein@perforce.com

PERFORCE

Solving the Hardest Challenges in DevOps



Perforce Products

Agile Management

- Helix ALM
- Hansoft
- Gliffy

Code Management & Collaboration

- Helix Core
- Methodics
- Helix4Git
- JRebel
- TotalView

Application Mgmt. & Components

- Puppet
- Akana
- OpenLogic
- Zend
- Visualization
- SourcePro
- IMSL

Automated Testing

- Helix QAC
- Klocwork
- Perfecto
- BlazeMeter

TotalView

Overview of TotalView Labs

Overview of TotalView Labs

Nine different labs and accompanying example programs

- Lab 1 - Debugger Basics: Startup, Basic Process Control, and Navigation
- Lab 2 - Viewing, Examining, Watching, and Editing Data
- Lab 3 - Examining and Controlling a Parallel Application
- Lab 4 - Exploring Heap Memory in an MPI Application
- Lab 5 - Debugging Memory Comparisons and Heap Baseline *
- Lab 6 - Memory Corruption discovery using Red Zones *
- Lab 7 - Batch Mode Debugging with TVScript
- Lab 8 - Reverse Debugging with ReplayEngine
- Lab 9 - Asynchronous Control Lab

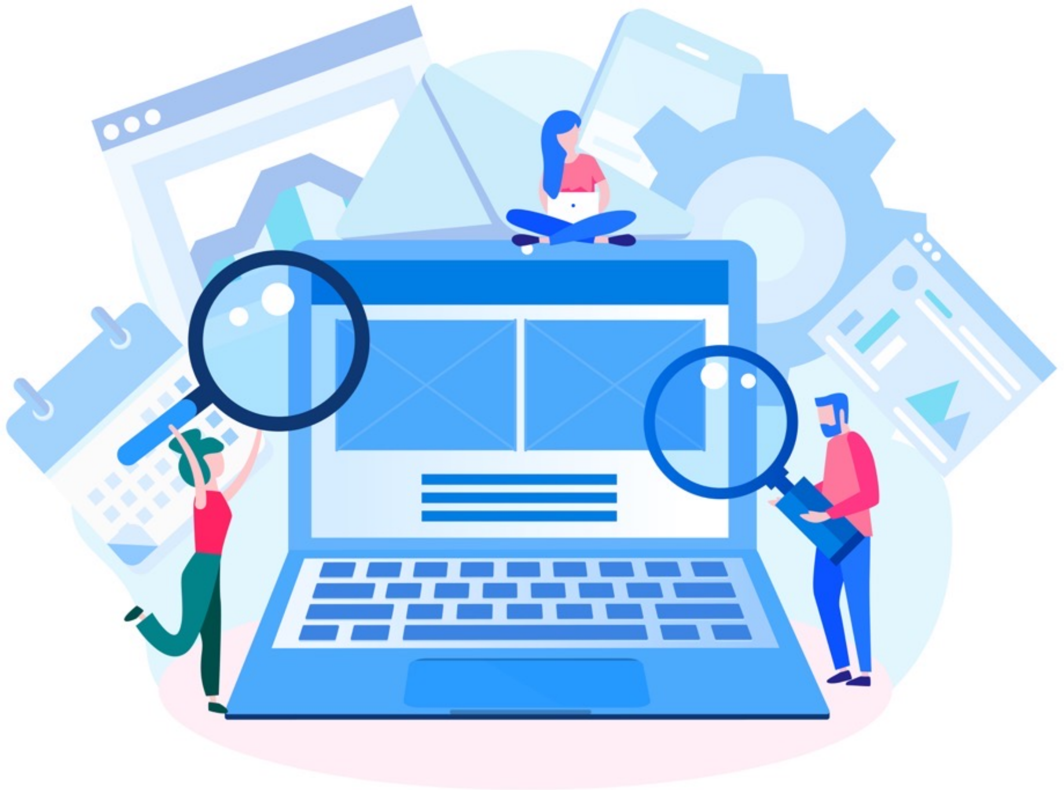
Notes

- * Labs 5 and 6 require use of TotalView's Classic UI
- Sample program breakpoint files were created with GNU compilers. If a different compiler is used, they may not load and will need to be recreated.
- Several example programs use OpenMPI so you will need to configure your environment beforehand.
- We do not have a lab specific to Python Debugging yet. There are good examples and instructions in the TotalView *totalview.<version>/<linux-x86-64>/examples/PythonExamples* directory.
- Use this slide deck for GPU specific debugging information

TotalView Features

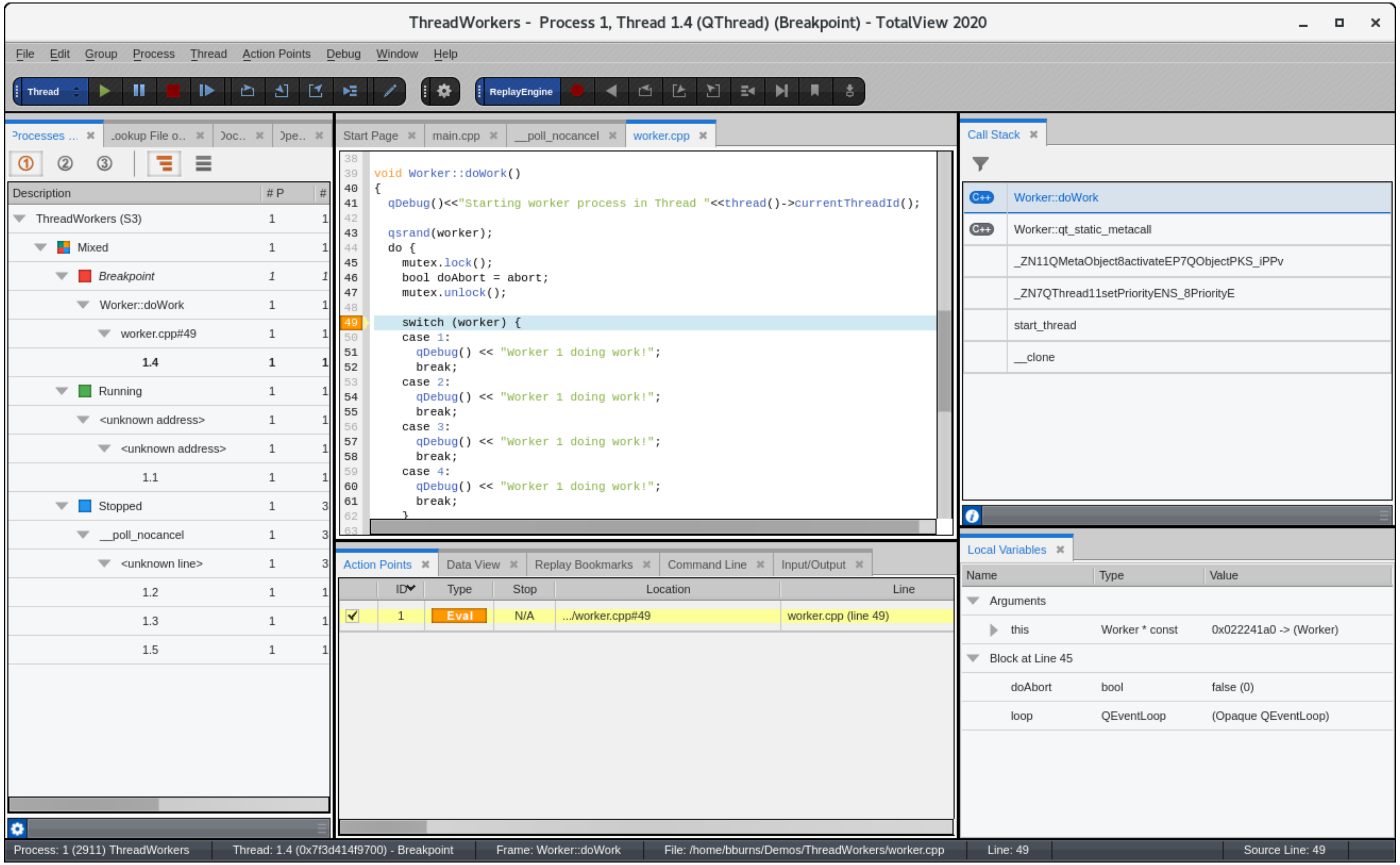
What is TotalView used for?

- Provides interactive Dynamic Analysis capabilities to help:
  - Understand complex code
  - Improve code quality
  - Collaborate with team members to resolve issues faster
  - Shorten development time
- Finds problems and bugs in applications including:
  - Program crash or incorrect behavior
  - Data issues
  - Application memory leaks and errors
  - Communication problems between processes and threads
  - CUDA application analysis and debugging
- Contains batch and Continuous Integration capabilities to:
  - Debug applications in an automated run/test environment



TotalView Features

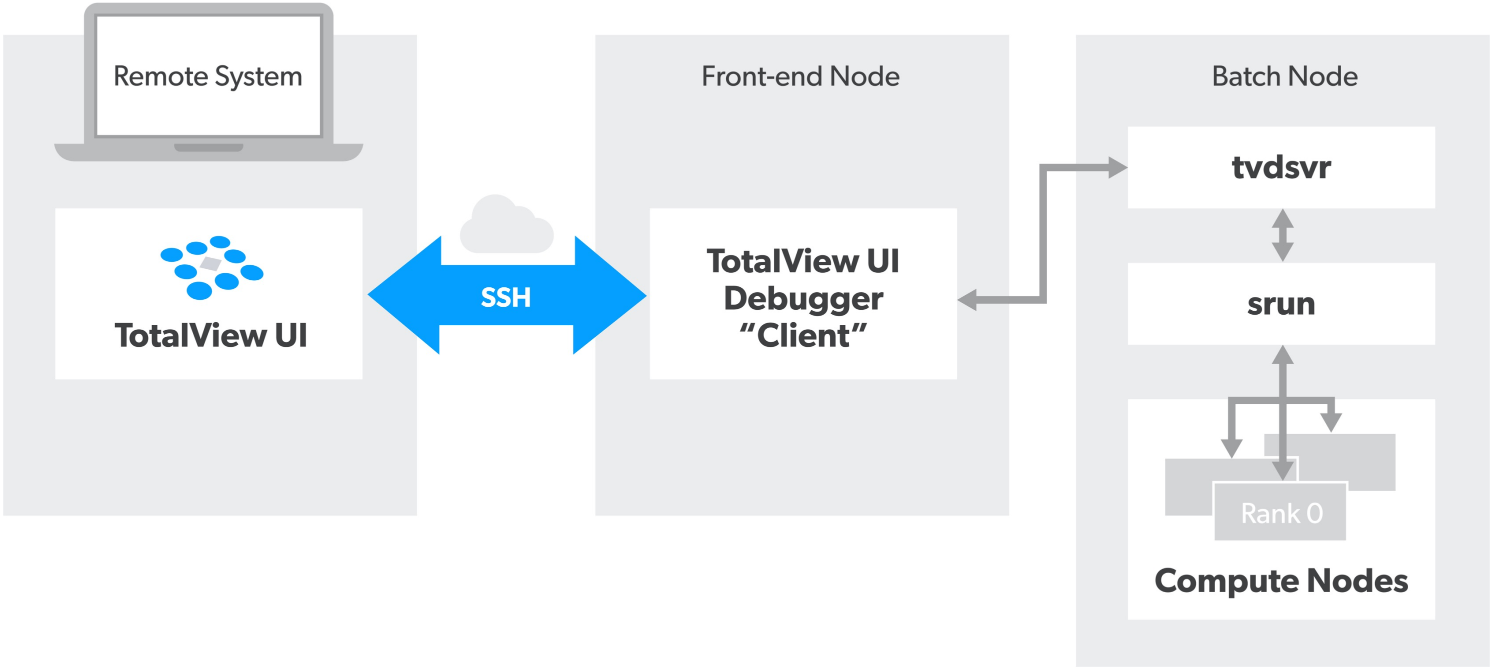
- Multi-process/thread dynamic analysis and debugging
- Comprehensive C, C++ and Fortran Support
- Thread specific breakpoints with individual thread control
  - View thread specific stack and data
- View complex data types easily
- MPI, OpenMP, Hybrid support
- NVIDIA (CUDA) and AMD (HIP) GPU support
- Convenient remote debugging
- Integrated Reverse debugging
- Mixed Language - Python C/C++ debugging
- Memory debugging
- Script debugging
- Linux, macOS and UNIX
- **More than just a tool to find bugs**
  - Understand complex code
  - Improve developer efficiency
  - Collaborate with team members
  - Improve code quality
  - Shorten development time



Recent TotalView Features

TotalView Remote Client for Windows

- TotalView 2024.1 adds native **Windows remote client support**
- Combine the convenience of establishing a remote connection to a cluster and the ability to run the TotalView GUI locally
- Front-end GUI architecture does not need to match back-end target architecture (macOS front-end -> Linux back-end)
- Secure communications
- Convenient saved sessions
- Once connected, debug as normal with access to all TotalView features
- **Windows, macOS and Linux native front-ends**



TotalView 2024.1 Platform Updates

Platform / Compiler Updates

- macOS Sonoma
- AMD GPU ROCm 6.0 and MI300

Other Updates

- Various bug fixes and other minor enhancements
- Third-party open-source package updates — security

Other Recent TotalView Updates

- Assembly and Register View
- C++ Type Transformations
  - Iterator support
  - Additional container classes
- Array Debugging
  - Array View
  - Array Visualization
- Apple ARM M1/2/3 support
- Memory Debugging Additions
  - Hoarding and Painting
  - Buffer overwrite detection

Category	Class	Iterator kind	Transformed type
Sequence	array (C++11)	Random	Array (Dense)
	vector	Random	Array (Dense)
	deque	Random	Array (Sparse)
	forward_list (C++11)	Forward	List
	list	Bidirectional	List
Associative	set	Bidirectional	Tree
	multiset	Bidirectional	Tree
	map	Bidirectional	Tree
	multimap	Bidirectional	Tree
Unordered associative	unordered_set	Forward, Local	Hashtable
	unordered_multiset	Forward, Local	Hashtable
	unordered_map	Forward, Local	Hashtable
	unordered_multimap	Forward, Local	Hashtable
Adaptors	stack (deque,list,vector)		Struct
	queue (deque,list)		Struct
	priority_queue (deque,vector)		Struct
General utilities	pair		Struct
	tuple (C++11)		Struct
Memory management	unique_ptr (C++11)		Struct
	shared_ptr (C++11)		Struct
	weak_ptr (C++11)		Struct
Numeric	complex		Struct
Strings	string		Struct

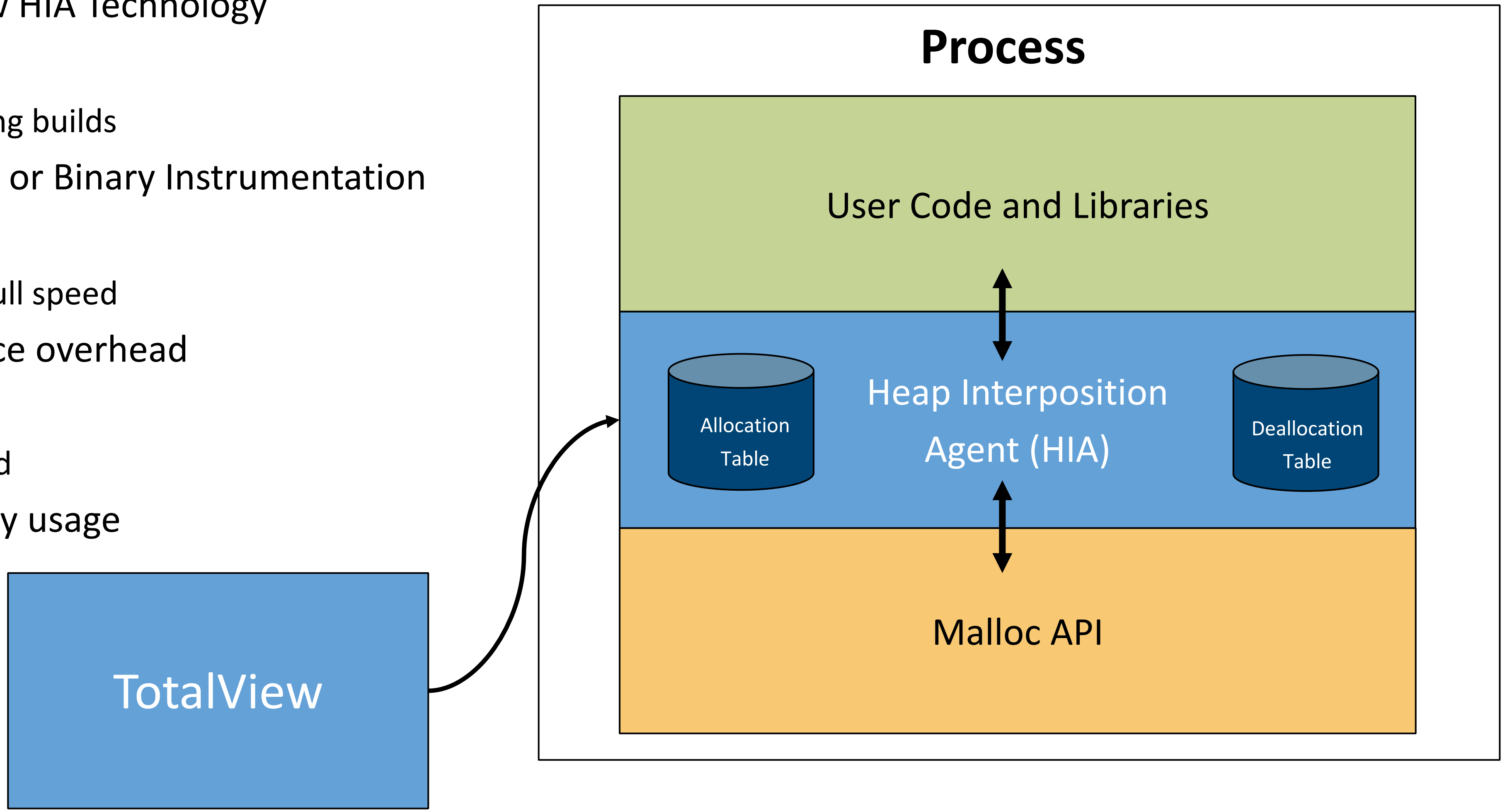
TotalView Memory Debugging

What is a Memory Bug?

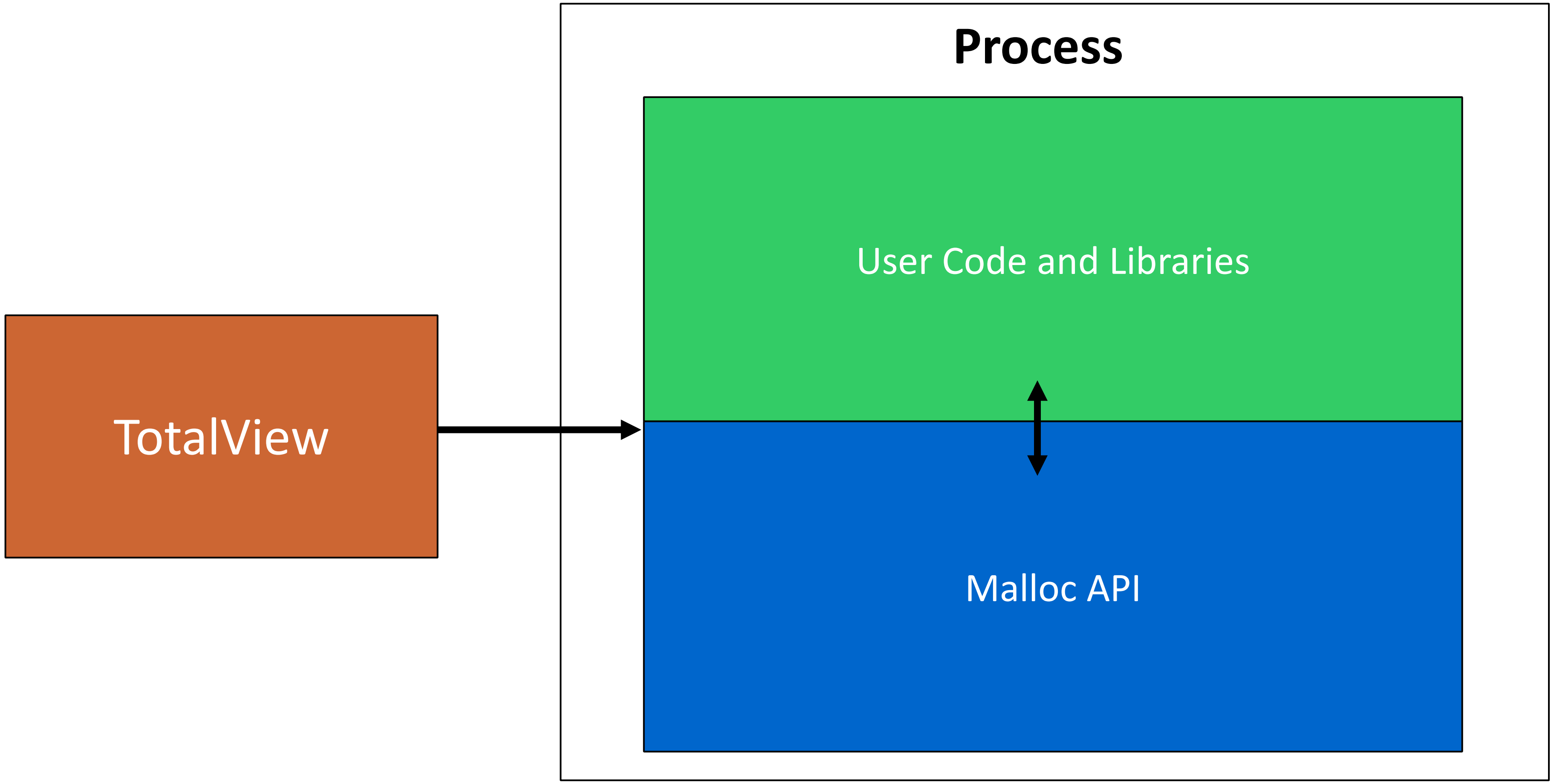
- A Memory Bug is a mistake in the management of heap memory
    - Leaking: Failure to free memory
    - Dangling references: Failure to clear pointers
    - Failure to check for error conditions
    - Memory Corruption
        - Writing to memory not allocated
        - Overrunning array bounds

TotalView Heap Interposition Agent (HIA) Technology

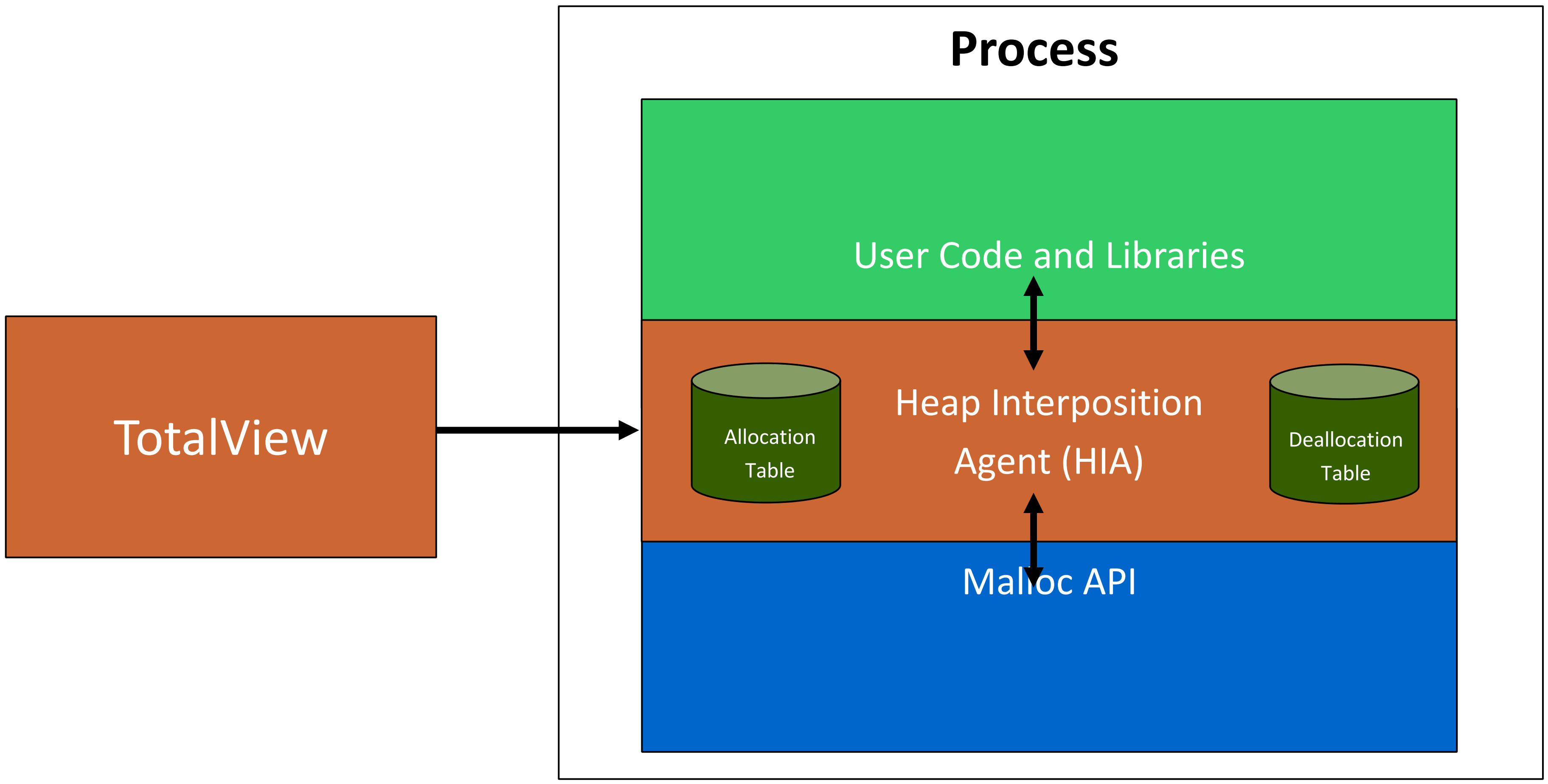
- Advantages of TotalView HIA Technology
  - Use it with your existing builds
    - No Source Code or Binary Instrumentation
  - Programs run nearly full speed
    - Low performance overhead
  - Low memory overhead
    - Efficient memory usage



The Agent and Interposition



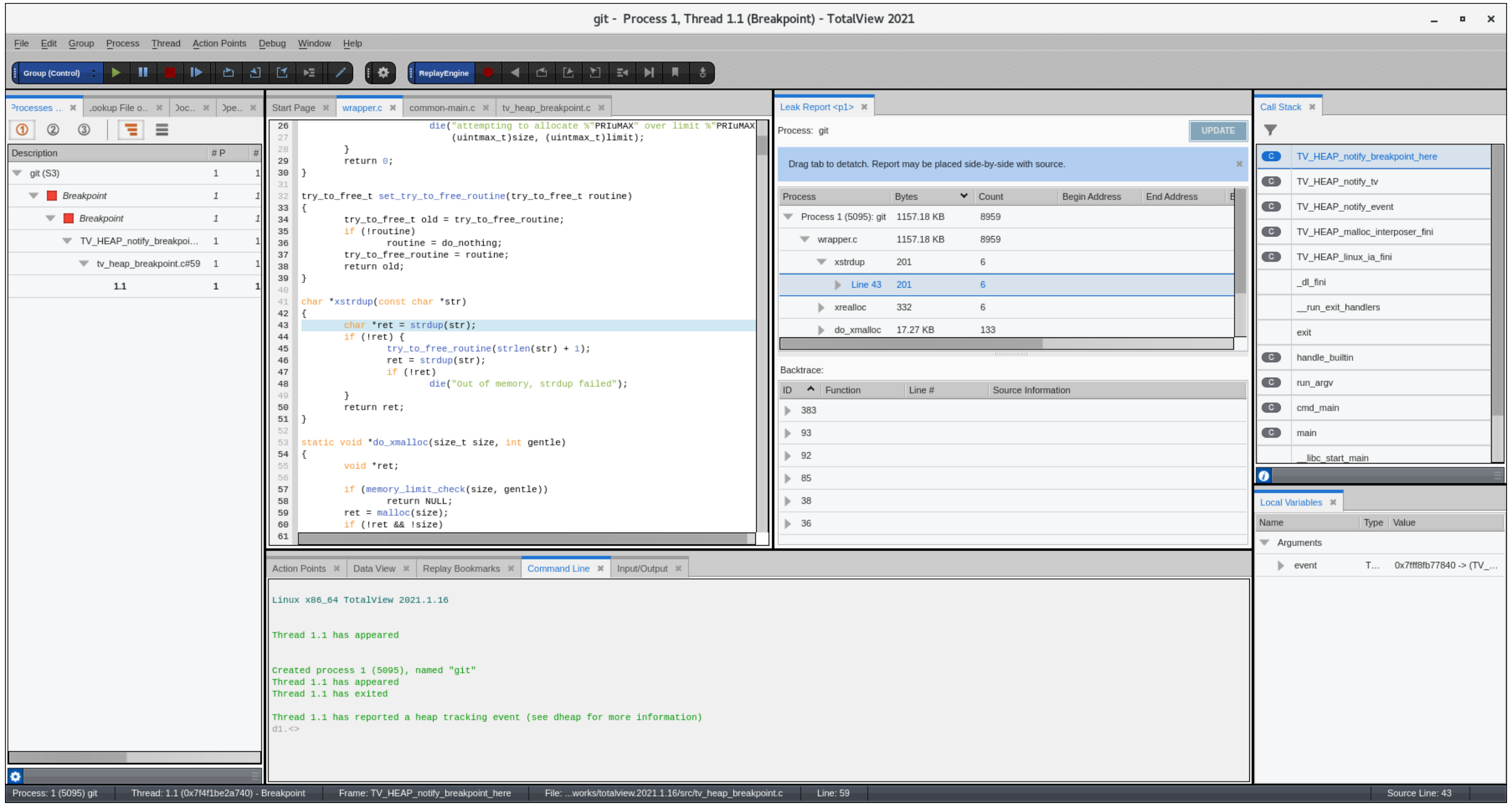
The Agent and Interposition



Memory Debugging in TotalView's New UI

TotalView 2024.1 Features

- Leak detection
- Dangling pointer detection
- Heap allocation overview
- Automatically detect allocation problems
- Memory Corruption Detection - Guard Blocks
- Memory Block Painting
- Memory Hoarding



Coming Features

- Graphical heap view
- Memory Corruption Detection - Red Zones
- Memory Comparisons between processes

Memory Debugging Demo

Demo

- Memory Debugging Demo

Debugging OpenMP Applications

TotalView Support for OpenMP

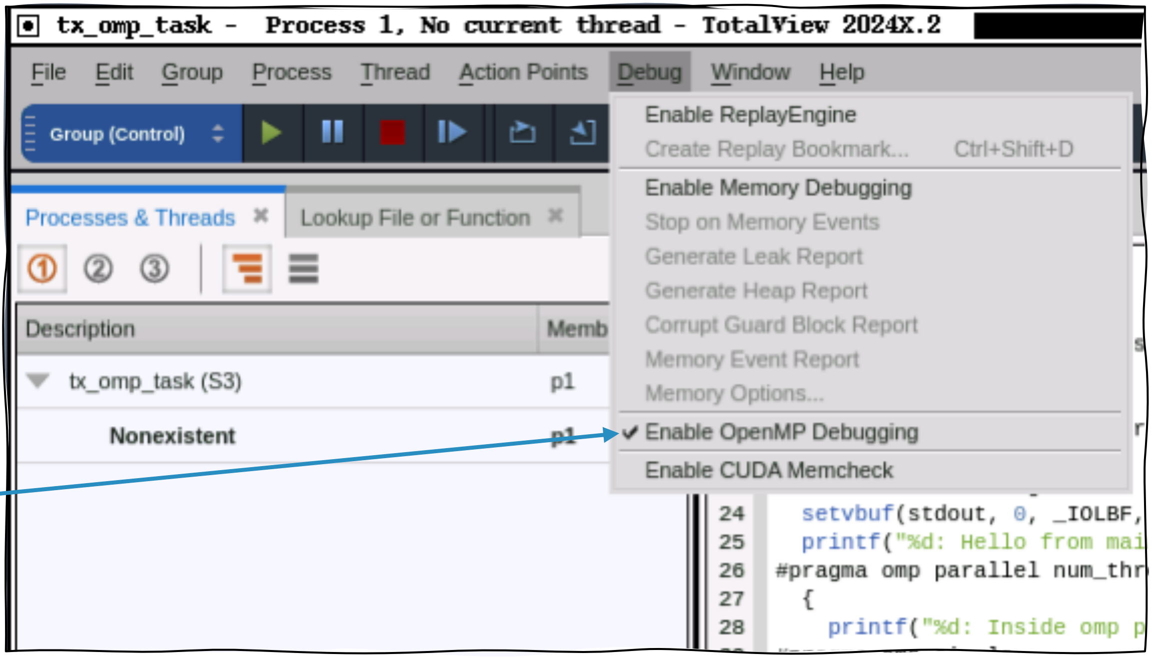
- Source-level debugging of the original OpenMP code (C, C++, and Fortran)
- Debug code inside of OMP **parallel** and **task** regions
  - Set stop-thread breakpoints, single-step, etc.
  - View OMP **shared**, **private**, and **threadprivate** variables
- Debug code inside OMP **target** regions
  - On NVIDIA GPUs, similar to debugging CUDA code
  - On AMD GPUs, similar to debugging HIP code
- CORAL-2 OMP/OMPD support (scheduled for TotalView 2024.2 release)
  - Focuses on Clang, AMD Clang/Flang, and HPE CCE compilers
  - Evaluates quality DWARF debug information produced by the compiler
  - Adds TotalView OpenMP views to display the runtime state of regions, threads, control variables, and ICVs
  - OpenMP thread-stack transformations: filter-out OMP runtime frames, annotate parallel/task regions, and insert parent links
  - Demangles OpenMP outlined function names

Debugging OpenMP Applications

- OpenMP programs are multi-threaded programs
  - Use normal debugging techniques for multi-threaded programs
  - Use stop-thread breakpoints inside parallel regions
  - Use thread-level execution controls: hold/unhold thread, single-step threads / thread groups
- OpenMP **parallel** and **task** regions are in outlined functions
  - A single line of source code can generate multiple block of machine code
  - Outlined function names are mangled by the compiler, but TV 2024.2 will support OMP name demangling
  - TV 2024.2 will support step into and out of OMP parallel region, if the compiler supports OMPD properly
  - Otherwise, set a breakpoint inside the parallel region and let the process run to it
- OpenMP **target** regions are offloaded to the GPU
  - Use normal debugging techniques for CUDA / HIP
  - Unfortunately, OMPD information is not available for target regions

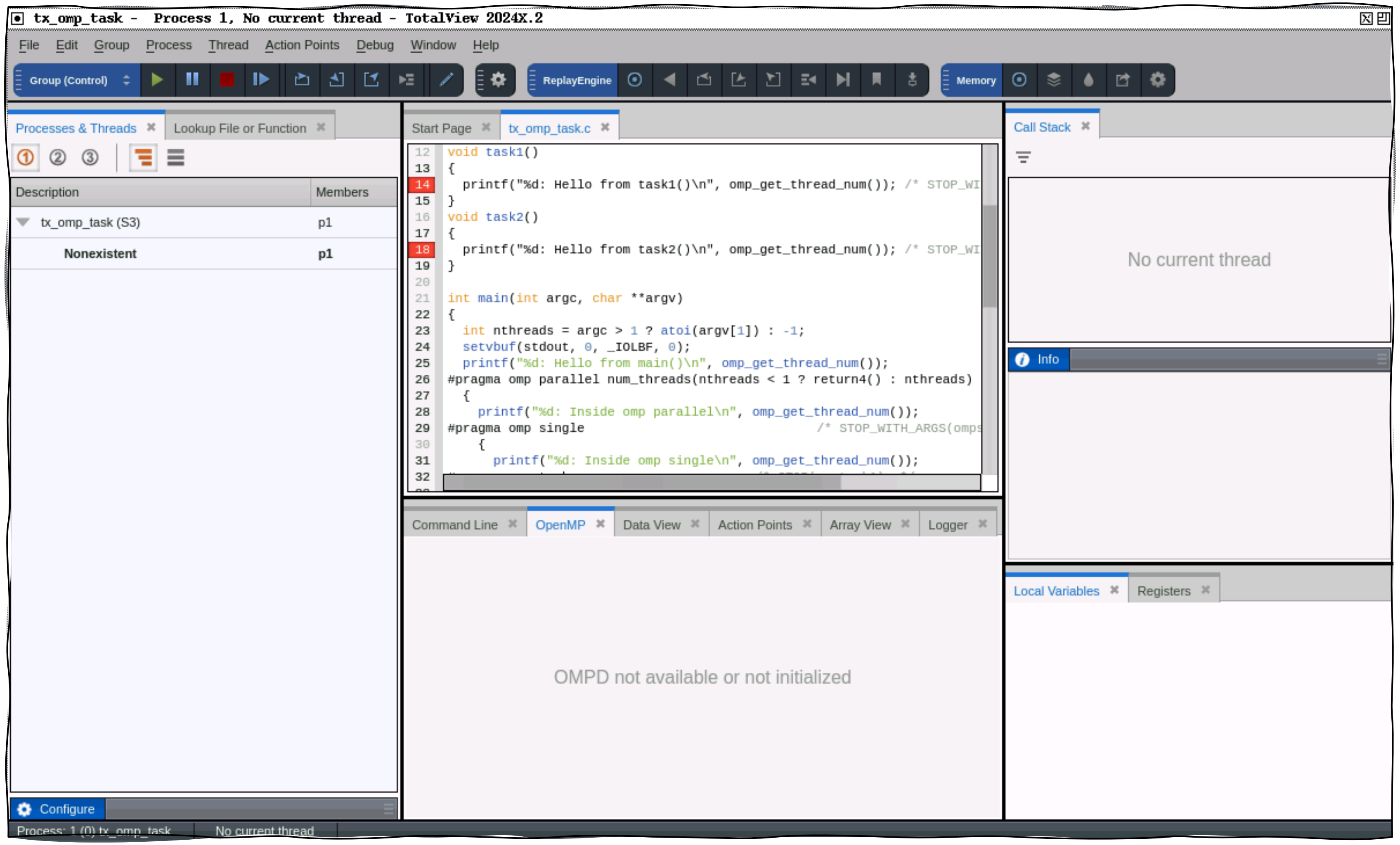
Enabling OMPD Support (TV 2024.2)

- TotalView OMPD support requires compiler support
  - Clang 15+, HPE CCE 17+, AMD Clang/Flang 17+
  - OMPD support is still maturing
  - Special linking rules might apply (check the doc)
- Set OMPD environment variable
  - **OMP_DEBUG=enabled**
  - MPI + OpenMP codes require setting the environment variable in the MPI processes
  - Use to propagate **OMP_DEBUG** setting
- Select “Enable OpenMP Debugging”
  - Good for non-MPI codes launched by the debugger
  - Does not work for processes that are attached to



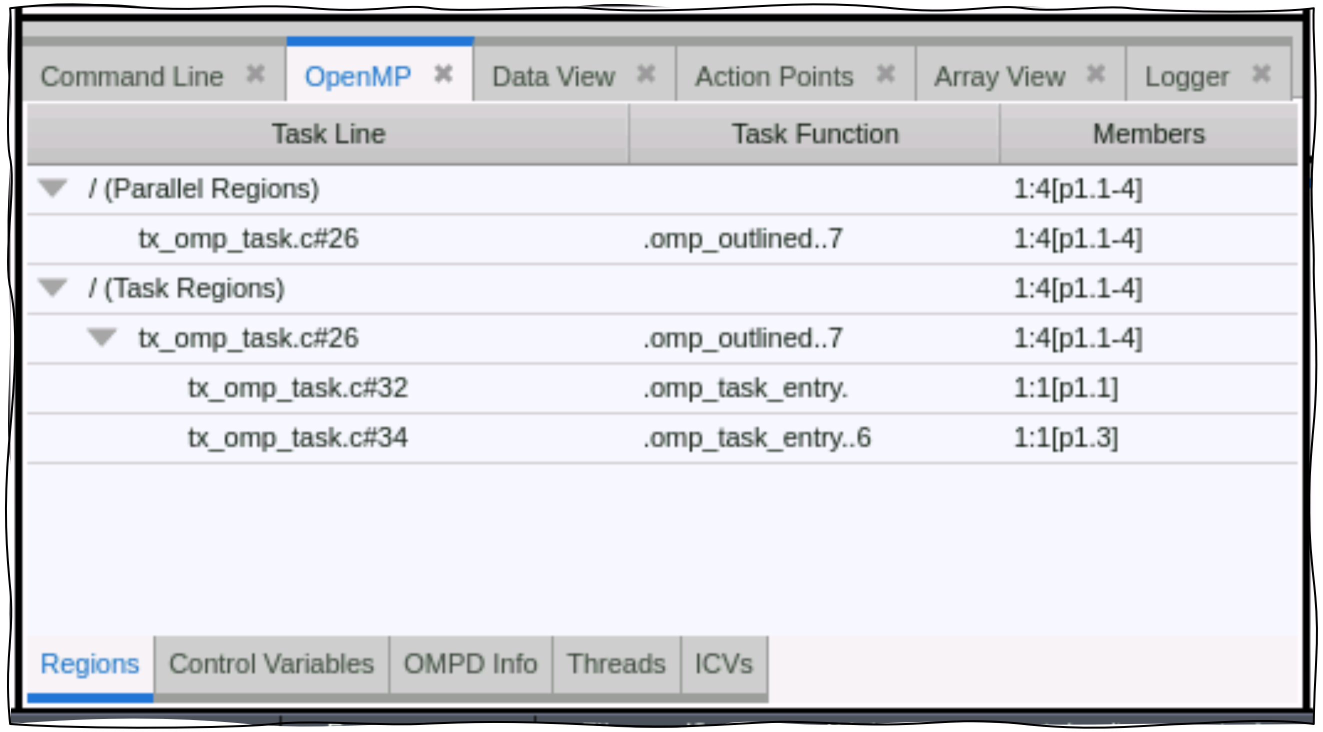
Example OpenMP Debugging Session (TV 2024.2)

- Example using AMD Clang 17
  - LLVM Clang works but has some issues
  - HPE CCE works but has some issues
  - gcc / gfortran works but OMPD support not tested
- Set stop-thread breakpoints inside parallel / task region
- Make sure the <u>whole</u> process is stopped so that the OpenMP runtime is in a consistent state



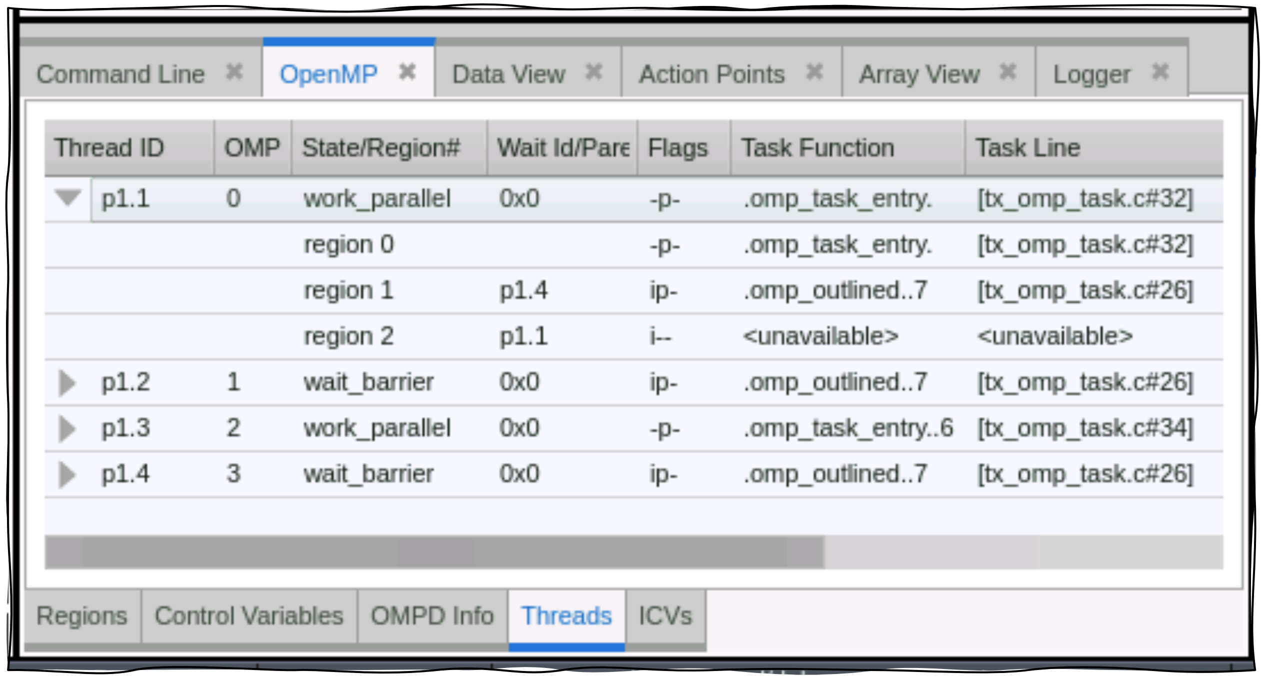
OpenMP > Regions (TV 2024.2)

- Displays **parallel** and **task** regions
  - Aggregated view of all OpenMP threads
- "Regions" tab shows
  - Source-code line-number of OMP region
  - OMP implicit or explicit task function name
  - OpenMP threads that are in the region



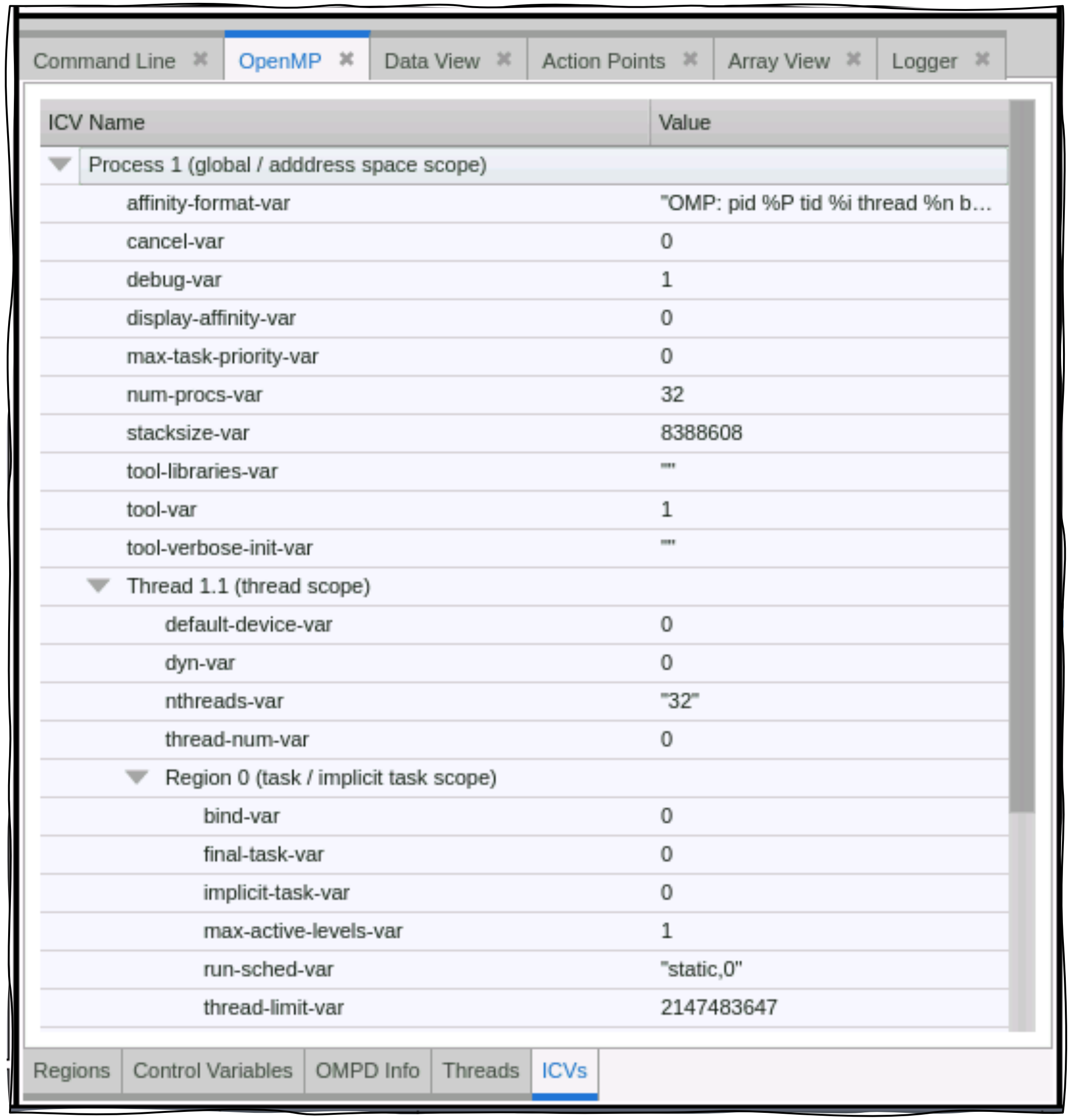
OpenMP Threads (TV 2024.2)

- Thread-oriented view of OMP threads
  - For the focus process
  - Current state plus nest of OMP generating task regions
- “Threads” tab shows
  - Debugger process/thread ID and OMP thread-num
  - Current state of OMP thread / region #
  - Wait ID / Parent (encountering thread) ID
  - Region flags
    - “i” implicit vs. explicit task
    - “p” active parallel region
    - “f” final task
  - Task function and source-code line-number
  - Runtime frame information (not shown)

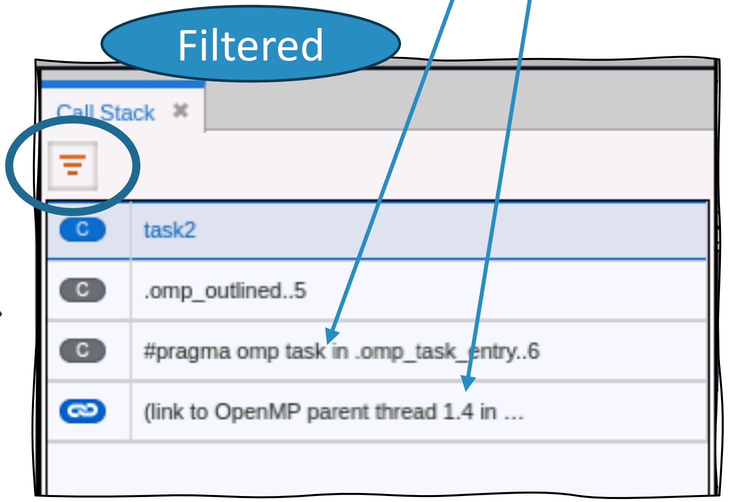


OpenMP > ICVs (TV 2024.2)

- Hierarchical view of OpenMP internal control variables
  - For the focus process
- Organized by OpenMP scope
  - Global / address space scope
  - Thread scope
  - Parallel region scope
  - Task / implicit task scope



OpenMP Stack Transformations



- Select the filter icon to filter
- OpenMP thread-stack transformations
  - Filters-out OMP runtime frames
  - Annotates parallel/task regions
  - Inserts parent thread links (click to focus on parent thread)

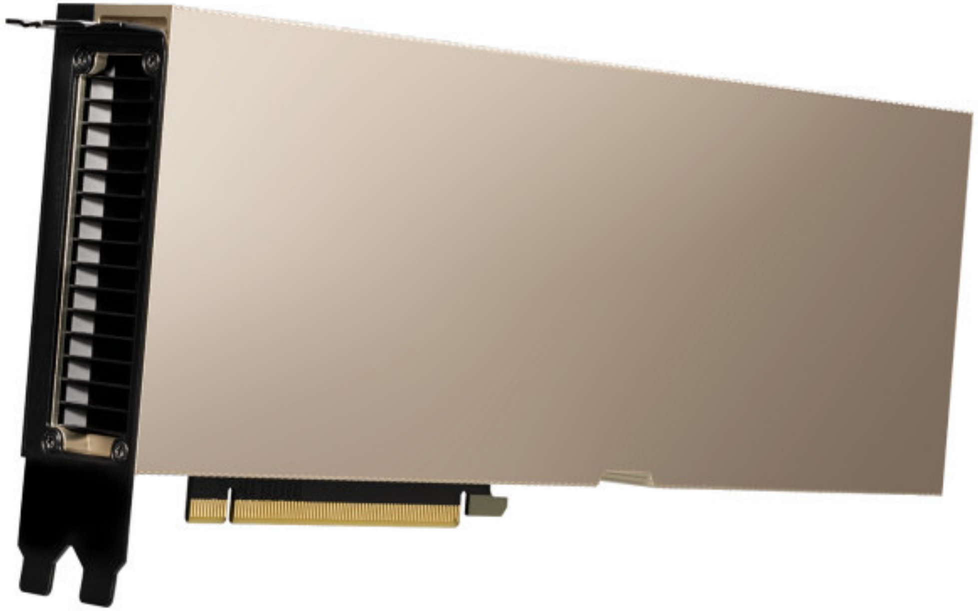
OpenMP Debugging Caveats

- OpenMP support prior to TotalView 2024.2 is a *prototype*
  - Not fully supported w/ limited compiler support
  - Has bugs and other problems with OpenMP displays
- LLVM/Clang-based compilers do <u>not</u> do a good at generating DWARF debug information
  - Program variables inside regions are all marked artificial, so TotalView does not display them
  - Use TotalView "**-compiler_vars**" option to display program variables, but compiler-generated variables are also displayed
  - Parallel "for" loop variables do not have correct values
  - Many other DWARF debug information problems exist
- GNU compilers seem to do a much better job in general
- Linking applications with OMPD support varies by compiler
  - Check the documentation

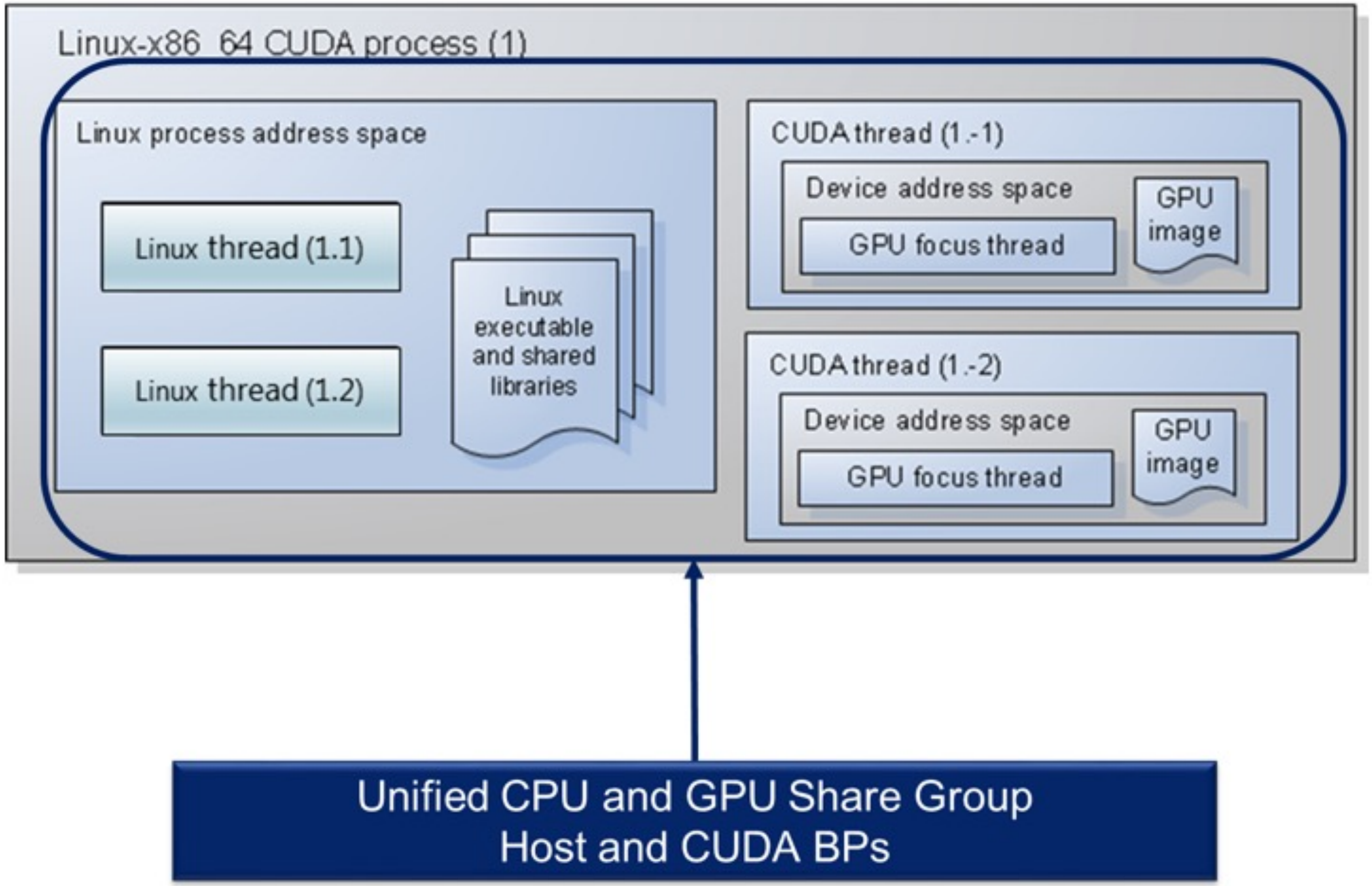
Debugging NVIDIA GPUs and CUDA with TotalView

TotalView for the NVIDIA ® GPU Accelerator

- NVIDIA Tesla, Fermi, Kepler, Pascal, Volta, Turing, Ampere, Hopper
- NVIDIA CUDA 9.2, 10, 11 and 12
  - With support for Unified Memory
- NVIDIA and Cray OpenACC support
- Features and capabilities include
  - Support for dynamic parallelism
  - Support for MPI based clusters and multi-card configurations
  - Flexible Display and Navigation on the CUDA device
    - Physical (device, SM, Warp, Lane)
    - Logical (Grid, Block) tuples
  - Support for types and separate memory address spaces
  - GPU Status view reveals what is running where

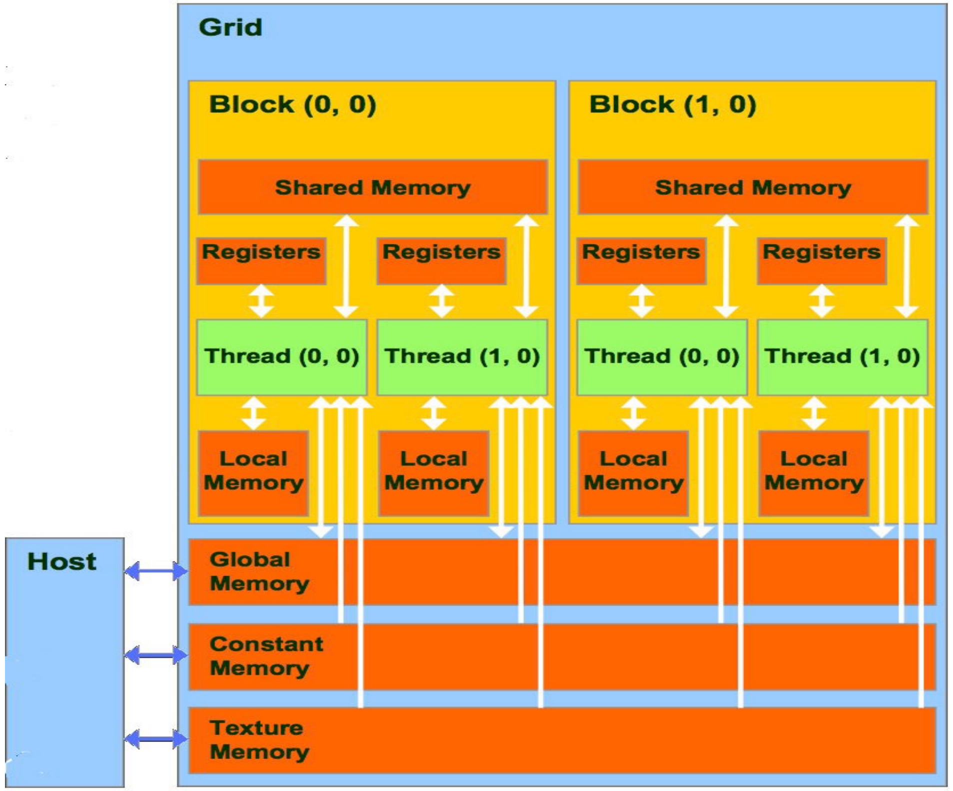


TotalView CUDA Debugging Model



GPU Memory Hierarchy

- Hierarchical memory
  - Local (thread)
    - Local
    - Register
  - Shared (block)
  - Global (GPU)
    - Global
    - Constant
    - Texture
  - System (host)



Supported Type Storage (aka, Address Space) Qualifiers

@generic	An offset within generic storage
@frame	An offset within frame storage
@global	An offset within global storage
@local	An offset within local storage
@parameter	An offset within parameter storage
@iparam	Input parameter
@oparam	Output parameter
@shared	An offset within shared storage
@surface	An offset within surface storage
@texsampler	An offset within texture sampler storage
@texture	An offset within texture storage
@rtvar	Built-in runtime variables
@register	A PTX register name
@sregister	A PTX special register name

Control of Threads and Warps

- Warps advance synchronously
  - They share a PC
- Single step operation advances all GPU threads in the same warp
- Stepping over a __syncthreads() call will advance all relevant threads
- To advance more than one warp
  - Continue, possibly after setting a new breakpoint
  - Select a line and “Run To”

NVIDIA GPU and CUDA Parallelization

- CUDA uses the single instruction multiple thread (SIMT) model of parallelization.
- CUDA GPUs made up of many computing units called cores
  - Cores includes an arithmetic logic unit (ALU) and a floating-point unit (FPU).
- Cores collected into groups called streaming multiprocessors (SMs).
- Computing tasks are parallelized by breaking them into numerous subtasks called threads.
- Threads are organized into blocks.
- Blocks are divided into warps whose size matches the number of cores in an SM.
- Each warp gets assigned to a particular SM for execution. GPUs have one or more SMs.
- SM control unit directs each of its cores to execute the same instructions simultaneously for each thread in the assigned warp.

Compiling for CUDA debugging

When compiling an NVIDIA CUDA program for debugging, it is necessary to pass the **-g -G** options to the nvcc compiler driver. These options disable most compiler optimization and include symbolic debugging information in the driver executable file, making it possible to debug the application.

```
% /usr/local/bin/nvcc -g -G -c tx_cuda_matmul.cu -o tx_cuda_matmul.o

% /usr/local/bin/nvcc -g -G -Xlinker=-R/usr/local/cuda/lib64 \
tx_cuda_matmul.o -o tx_cuda_matmul

% ./tx_cuda_matmul
A:
[ 0][ 0] 0.000000
...output deleted for brevity...
[ 1][ 1] 131.000000
```

Compiling for a specific GPU architecture (avoids JIT’ing from PTX)

Compiling for Ampere
-gencode arch=compute_80,code=sm_80

Compiling for Volta
-gencode arch=compute_70,code=sm_70

Compiling for Pascal
-gencode arch=compute_60,code=sm_60

Compiling for Kepler
-gencode arch=compute_35,code=sm_35

Compiling for Fermi and Tesla
-gencode arch=compute_20,code=sm_20 –gencode arch=compute_10,code=sm_10

Compiling for Fermi
-gencode arch=compute_20,code=sm_20

A TotalView Session with CUDA

A standard TotalView installation supports debugging CUDA applications running on both the host and GPU processors.

TotalView dynamically detects a CUDA install on your system. To start the TotalView GUI or CLI, provide the name of your CUDA host executable to the totalview or totalviewcli command.

For example, to start the TotalView GUI on the sample program, use the following command:

% totalview tx_cuda_matmul

* This example is just a single node, no MPI application

Source View Opened on CUDA host code

Start Page | tx_cuda_matmul.cu

```
  Matrix A;
  A.width = width_;
  A.height = height_;
  A.stride = width_;
  A.elements = (float*) malloc(sizeof(*A.elements) * width_ * height_);
  for (int row = 0; row < height_; row++)
    for (int col = 0; col < width_; col++)
      A.elements[row * width_ + col] = row * 10.0 + col;
  return A;
}

static void
print_Matrix (Matrix A, const char *name)
{
  printf("%s:\n", name);
  for (int row = 0; row < A.height; row++)
    for (int col = 0; col < A.width; col++)
      printf ("[%5d][%5d] %f\n", row, col, A.elements[row * A.stride + col]);
}

// Multiply an m*n matrix with an n*p matrix results in an m*p matrix.
// Usage: tx_cuda_matmul [ m [ n [ p ] ] ]
// m, n, and p default to 1, and are multiplied by BLOCK_SIZE.
int main(int argc, char **argv)
{
//  cudaSetDevice(0);
  const int m = BLOCK_SIZE * (argc > 1 ? atoi(argv[1]) : 1);
  const int n = BLOCK_SIZE * (argc > 2 ? atoi(argv[2]) : 1);
  const int p = BLOCK_SIZE * (argc > 3 ? atoi(argv[3]) : 1);
  Matrix A = cons_Matrix(m, n);
  Matrix B = cons_Matrix(n, p);
  Matrix C = cons_Matrix(m, p);
  MatMul(A, B, C);
  print_Matrix(A, "A");
  print_Matrix(B, "B");
  print_Matrix(C, "C");
  return 0;
}

/*
 * Update log
 *
 * Feb 25 2015 NYP: Removed __forceinline__, it is making cli too fast
```

Set Breakpoints in CUDA Kernel Code Before Launch

Set breakpoints in the CUDA or OpenMP TARGET region code before you start the process.

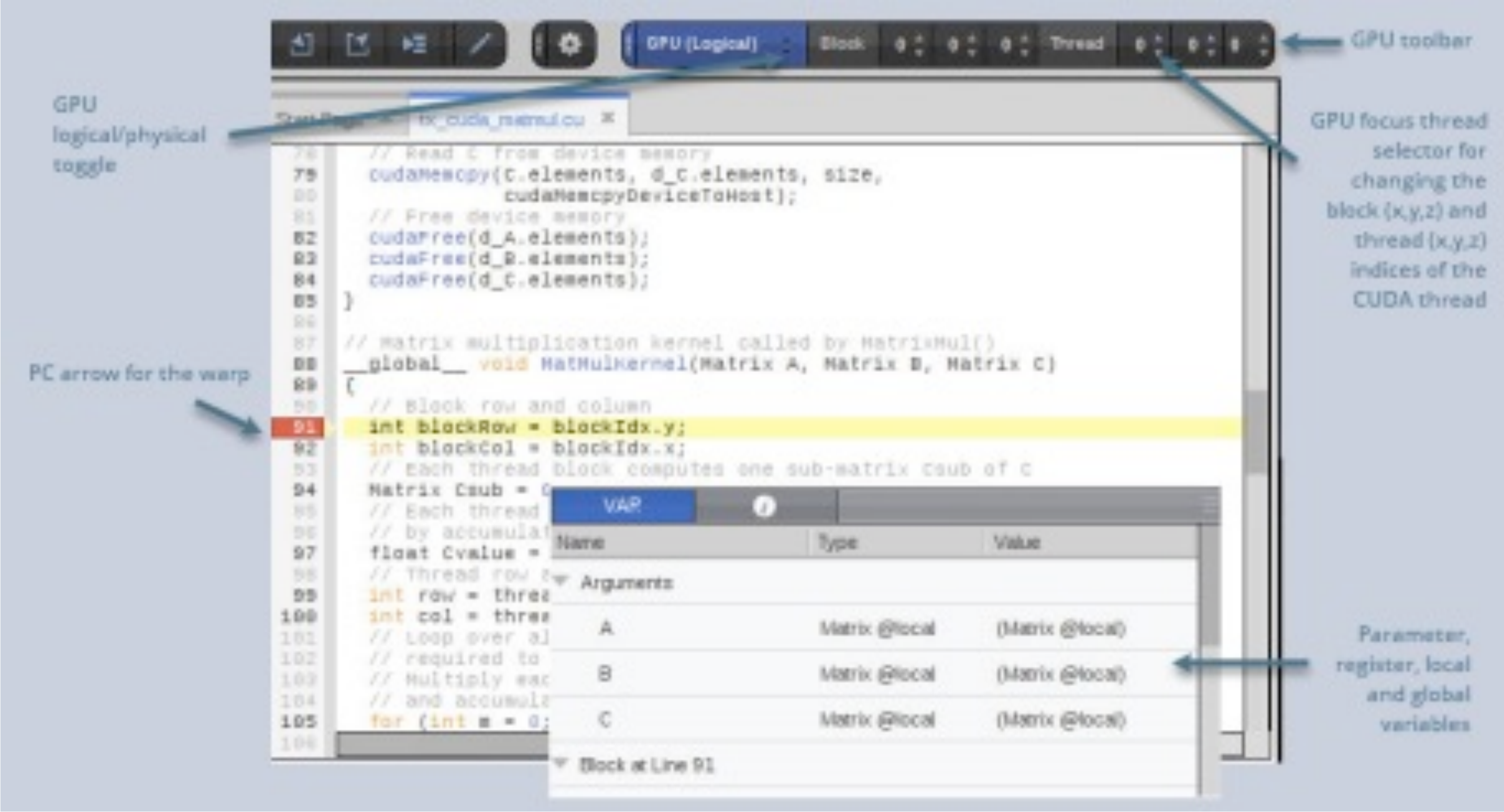
Hollow breakpoint indicates a breakpoint will be set when the code is loaded onto the GPU.

Start Page   tx_cuda_matmul.cu

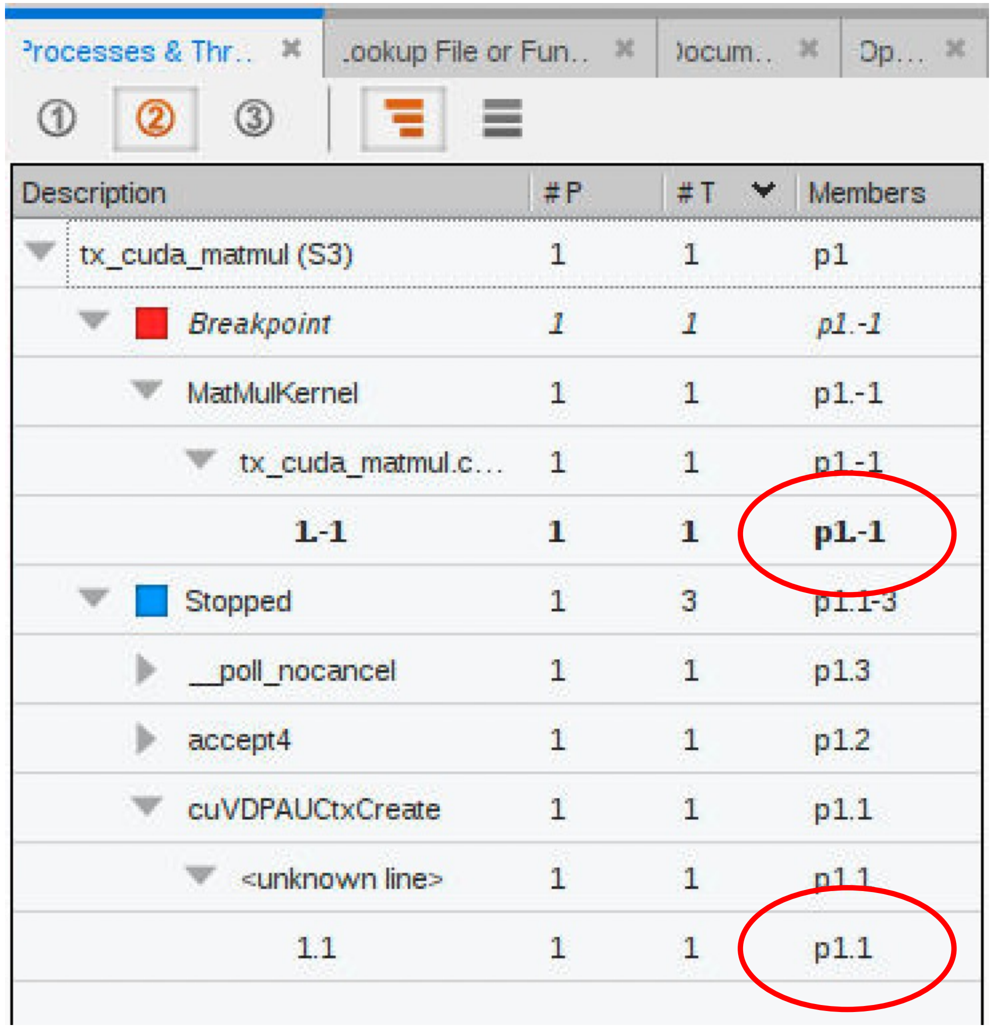
```
// Matrix multiplication kernel called by MatrixMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Block row and column
  int blockRow = blockIdx.y;
  int blockCol = blockIdx.x;
  // Each thread block computes one sub-matrix Csub of C
  Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
  // Each thread computes one element of Csub
  // by accumulating results into Cvalue
  float Cvalue = 0;
  // Thread row and column within Csub
  int row = threadIdx.y;
  int col = threadIdx.x;
  // Loop over all the sub-matrices of A and B that are
  // required to compute Csub
  // Multiply each pair of sub-matrices together
  // and accumulate the results
  for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {
    // Get sub-matrix Asub of A
    Matrix Asub = GetSubMatrix(A, blockRow, m);
    // Get sub-matrix Bsub of B
    Matrix Bsub = GetSubMatrix(B, m, blockCol);
    // Shared memory used to store Asub and Bsub respectively
    __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
    // Load Asub and Bsub from device memory to shared memory
    // Each thread loads one element of each sub-matrix
    As[row][col] = GetElement(Asub, row, col);
    Bs[row][col] = GetElement(Bsub, row, col);
    // Synchronize to make sure the sub-matrices are loaded
    // before starting the computation
    __syncthreads();
    // Multiply Asub and Bsub together
    for (int e = 0; e < BLOCK_SIZE; ++e)
      Cvalue += As[row][e] * Bs[e][col];
    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();
  }
  // Write Csub to device memory
```

Stopped at a Breakpoint in CUDA Kernel Code

- Bold line numbers indicate source code lines where the compiler generated code, which are good places to set breakpoints



CUDA thread IDs and Coordinate Spaces



Host thread IDs have a positive thread ID (p1.1)

CUDA thread IDs have a negative thread ID (p1.-1)

GPU Physical and Logical Focus Toolbars

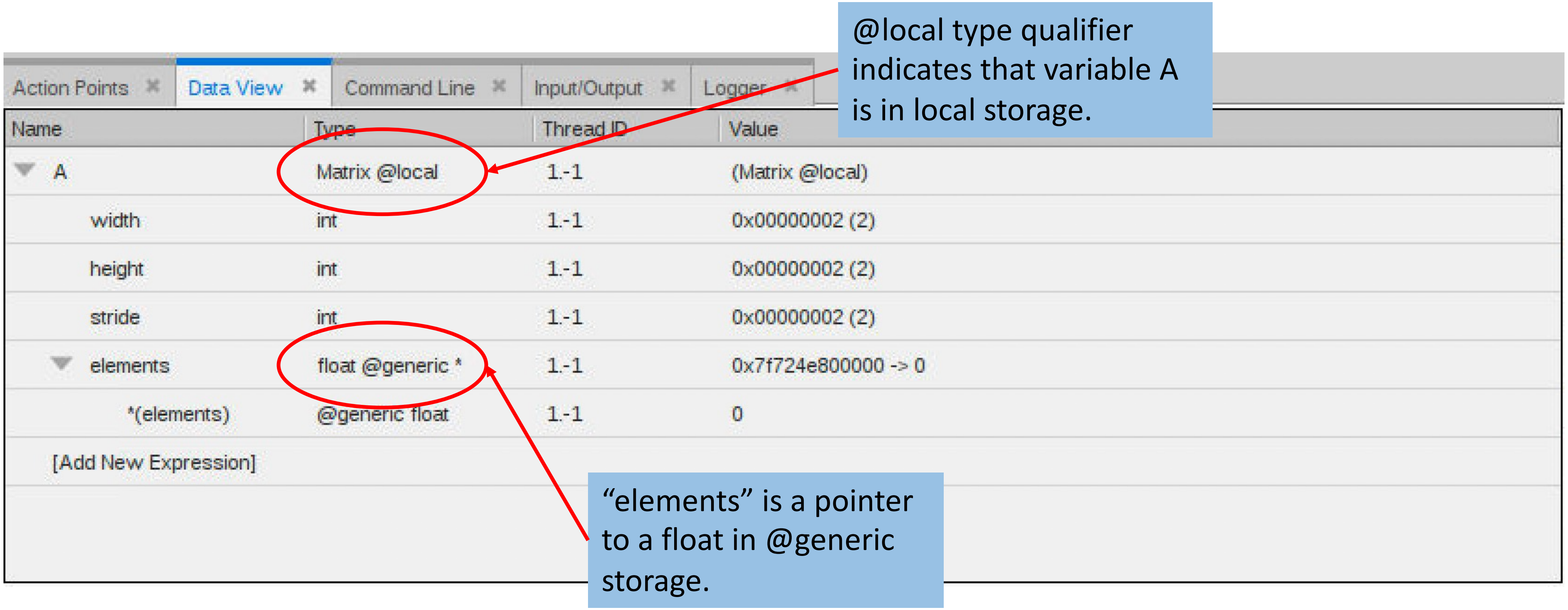
Logical toolbar displays the Block and Thread coordinates.

Physical toolbar displays the Device number, Streaming Multiprocessor, Warp and Lane.

To view a CUDA host thread, select a thread with a positive thread ID in the Process and Threads view.

To view a CUDA GPU thread, select a thread with a negative thread ID, then use the GPU focus controls in the logical or physical toolbar to focus on a specific GPU thread or lane.

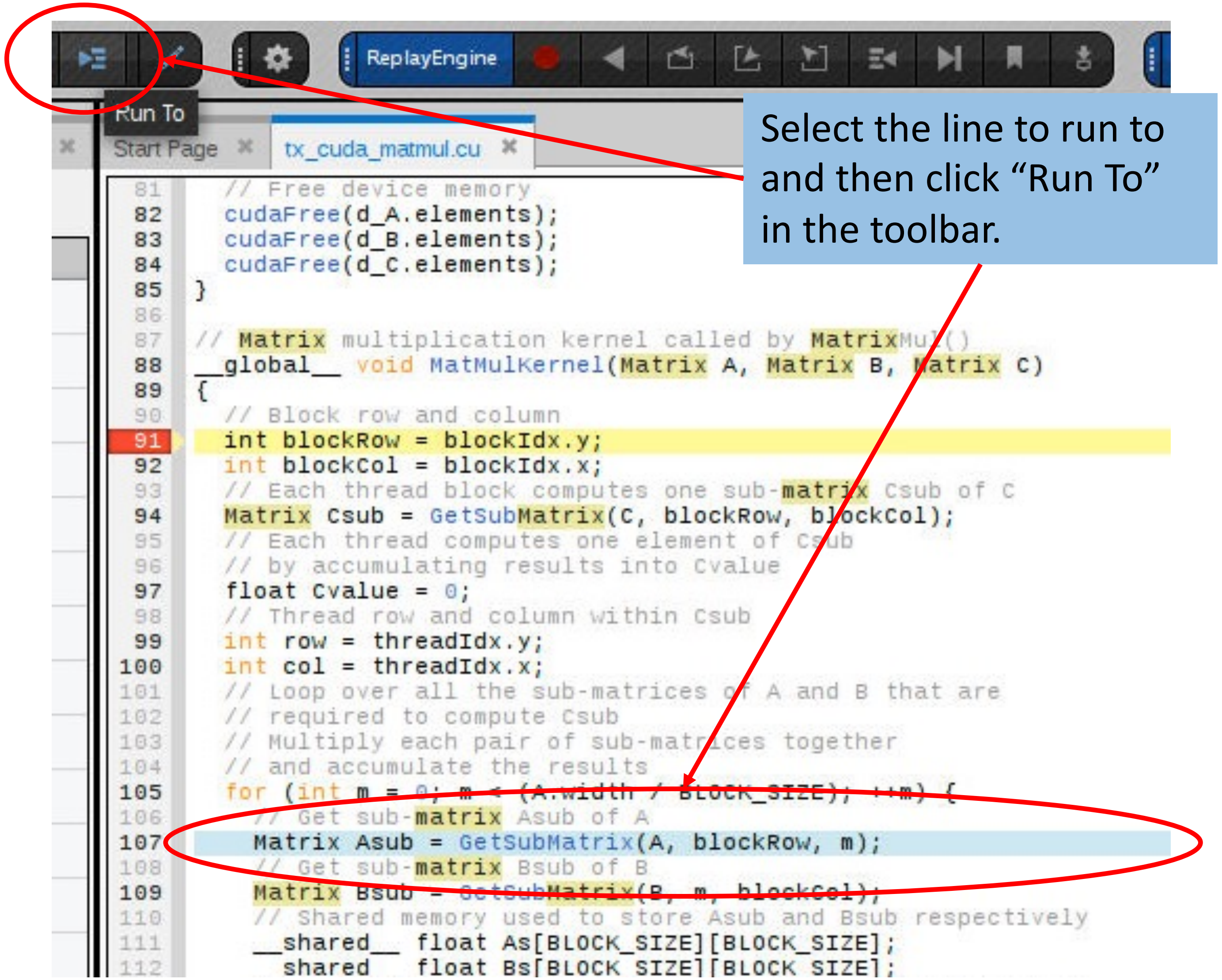
Displaying CUDA Program Variables



- The identifier @local is a TotalView built-in type storage qualifier that tells the debugger the storage kind of "A" is local storage.
- The debugger uses the storage qualifier to determine how to locate A in device memory

Stepping GPU Code

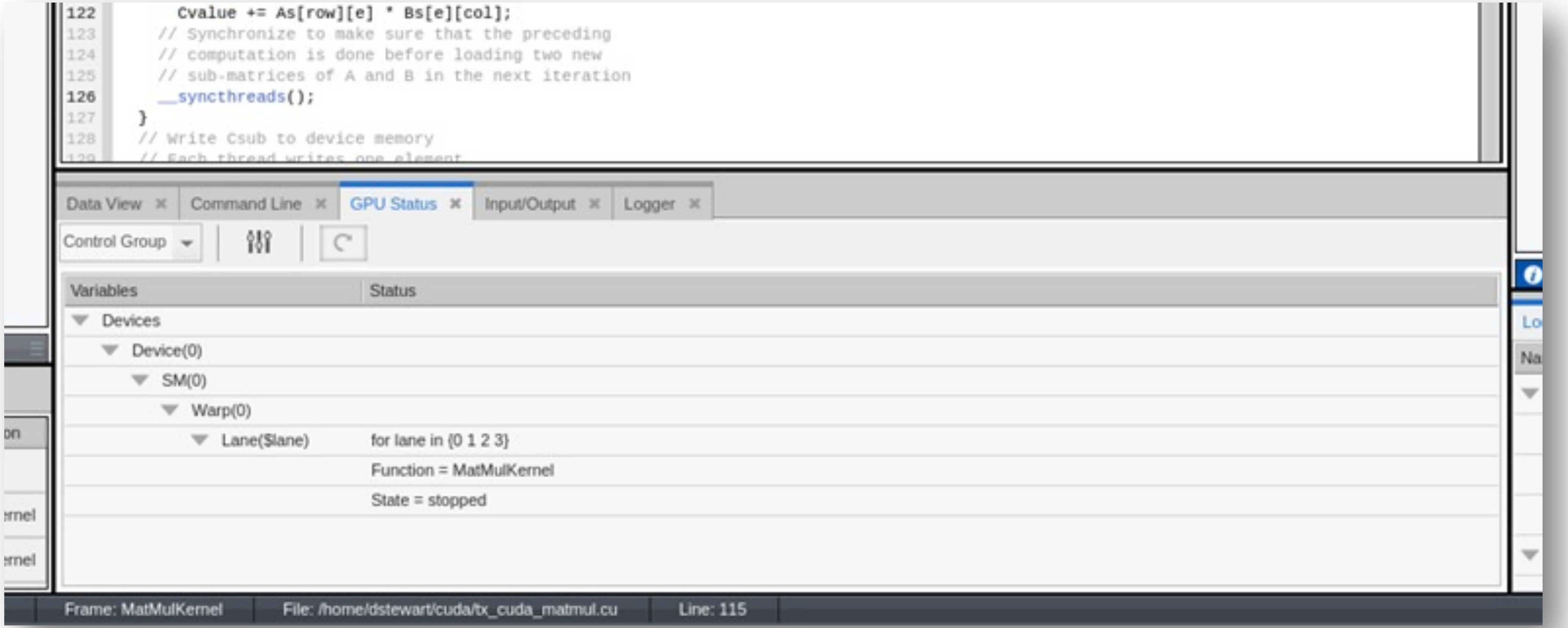
- Single-step operations advance all the GPU hardware lanes in the same warp
- Note that stepping operations Step and Next are slow in GPU code; the following is faster…
- To advance the execution of more than one warp, you may either:
  - Set a breakpoint and continue the process, or
  - Select a line number in the source pane and select “Run To”.



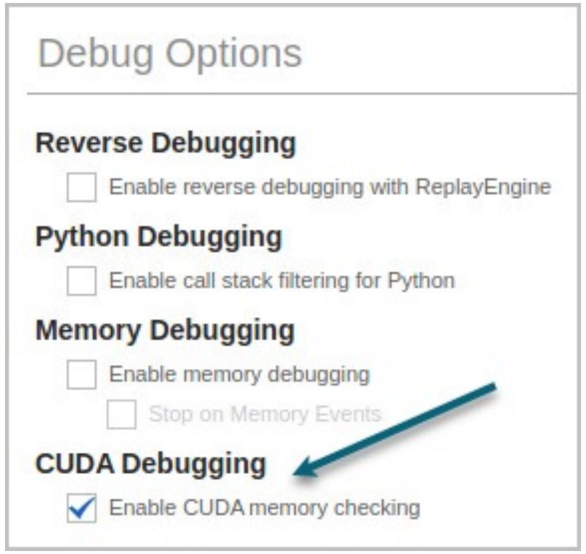
GPU Status View

Displays the state of all the GPUs being debugged.

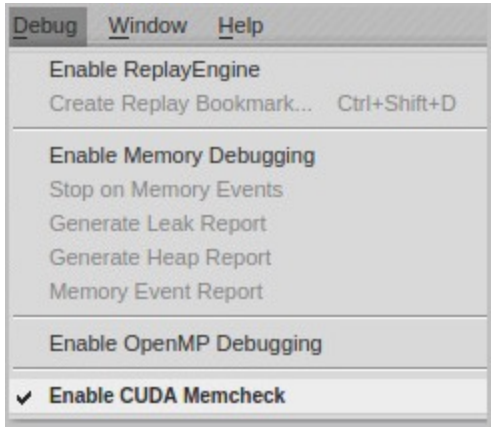
Fully configurable to allow aggregating, sorting and filtering based on physical or logical attributes.



Enabling CUDA Memory Checker Feature

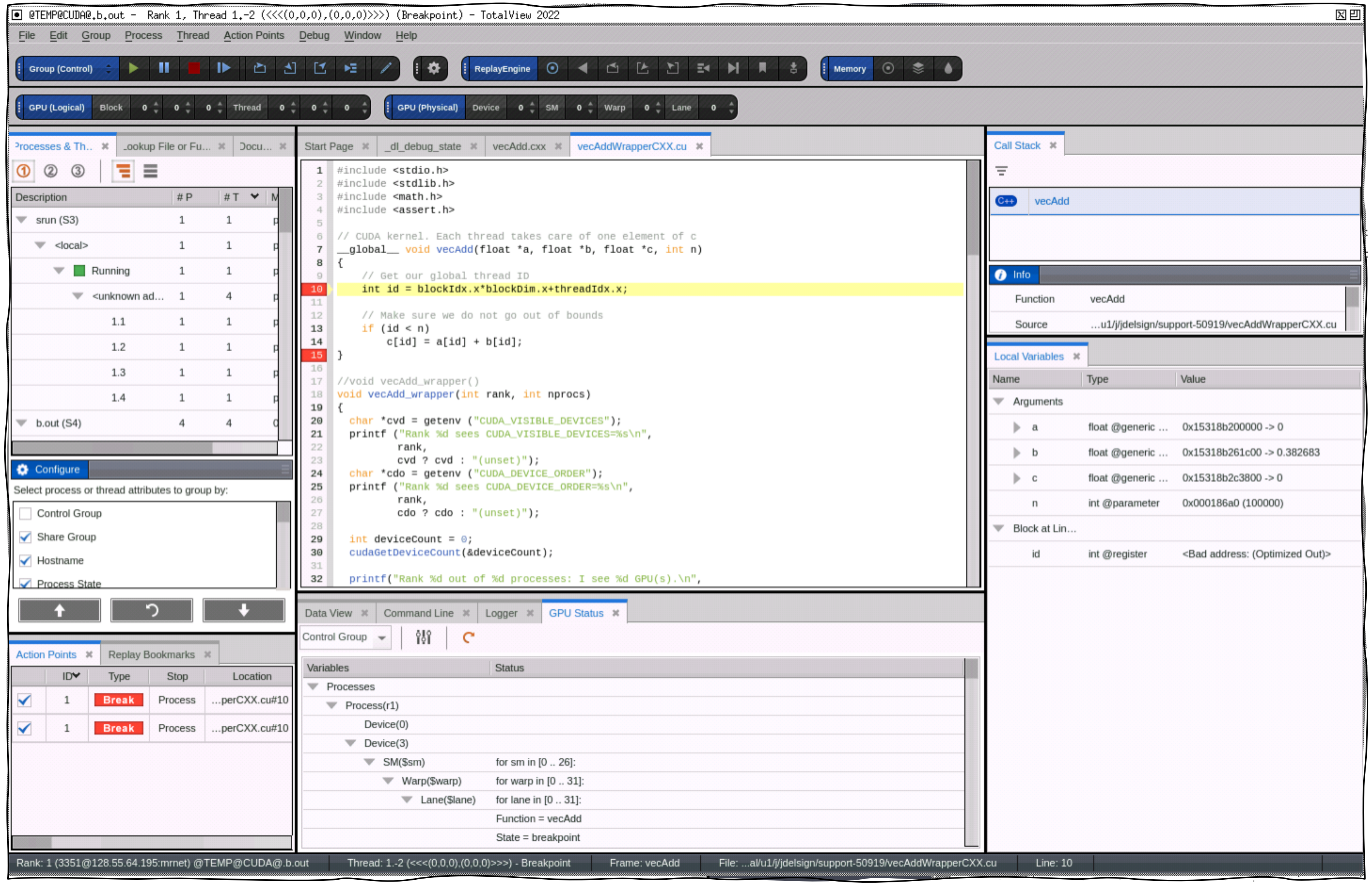


From the Program Session Dialog



From the Debug Menu

Demo



Debugging on Perlmutter

Debugging on Perlmutter (Things to Know)

- If you bind processes to GPUs using srun, the debugger cannot determine which GPUs the processes are using
  - SLURM’s use of Linux control groups make it impossible
  - Workaround – Do not use the “--gpu-bind” option when debugging
- Watchpoints in GPU memory are not support on NVIDIA GPUs, but CPU watchpoints are supported
- On Perlmutter, the environment variable “TVD_DISABLE_CRAY=1” must be set to disable using Cray CTI
  - “module load totalview” sets TVD_DISABLE_CRAY=1 on Perlmutter
  - SSH is used to instantiate the TV/MRNet tree
  - Requires passwordless SSH between nodes

Debugging on Perlmutter (Things to Know)

- Using SSH to between NERSC nodes can generate a lot of terminal output
  - Each SSH generates a long “NOTICE TO USERS” message
- The messages can be suppressed by adding the following lines to your “$HOST/.ssh/config” file:

```
# The "LogLevel quiet" option stops the "NOTICE TO USERS" messages
Host *
  LogLevel quiet
```

- The above is <u>not</u> necessary, but it does reduce terminal output

Debugging on Perlmutter (Supported Start-ups)

- TotalView supports interactive and batch debugging sessions
- Interactive debugging sessions
  - Use **salloc** to allocate interactive nodes
  - Start TotalView on **srun** within the allocation
  - Allows restarting **srun** multiple times within the same allocation
- Batch debugging sessions
  - Use **sbatch** to submit a batch job
  - Batch script uses **tvconnect srun …** to request a “reverse connect” to TotalView
  - Start TotalView on a login node and accept the “reverse connect” request
  - To restart srun multiple times, invoke **tvconnect srun** in a loop in the script

Debugging on Perlmutter (Interactive Start-up)

- Load the "totalview" module
  - **`module load totalview`**
- Allocate some nodes, for example
  - **`salloc -A ntrain7 -C gpu -N 2 -G 8 -t 60 -q interactive_ss11`**
- An interactive shell (bash, csh, etc.) will start inside the allocation
- Start **totalview** on **srun**, for example
  - **`totalview –args srun -n 8 -G 8 -c 32 --cpu-bind=cores ./b.out`**
    - Remember, "**`--gpu-bind`**" does not work, so do <u>not</u> use it while debugging

Debugging on Perlmutter (Batch Start-up)

- Example batch script using **tvconnect**

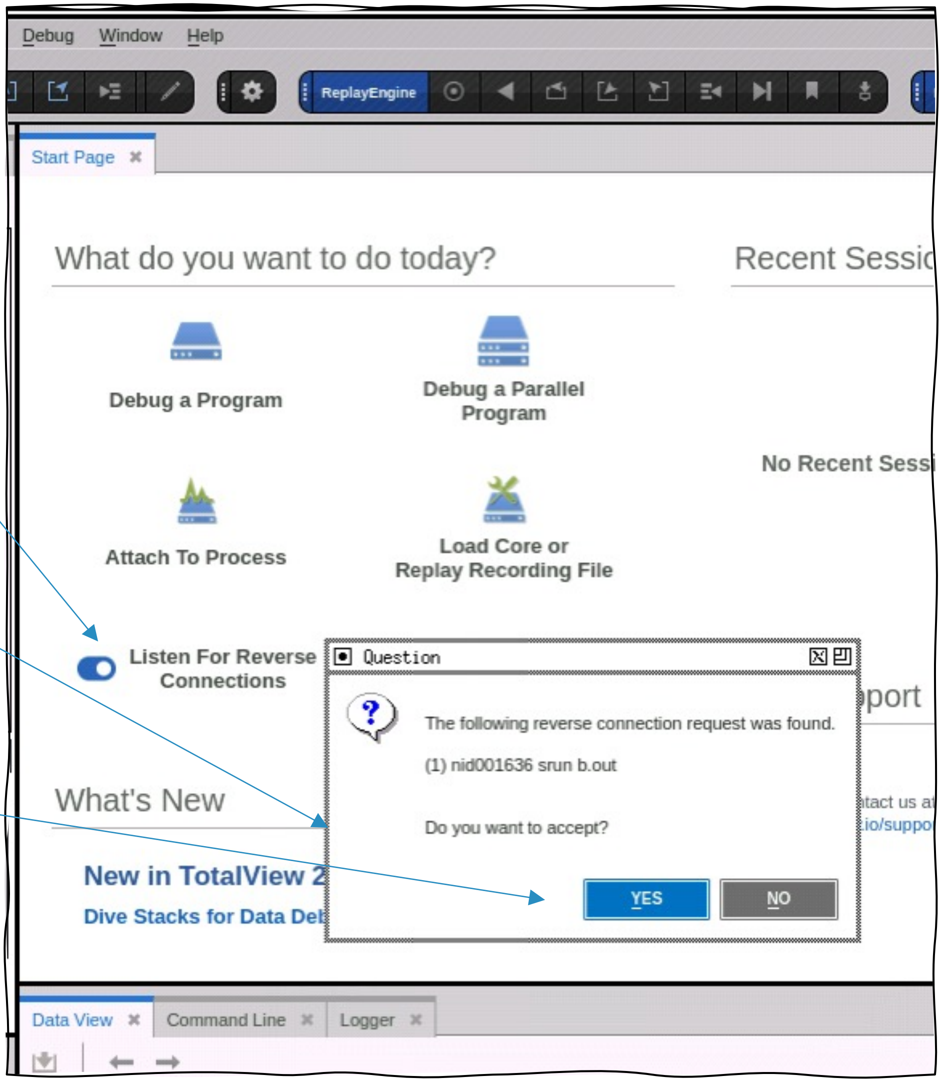
```
#!/bin/bash -x
#SBATCH -A nvendor
#SBATCH -C gpu
#SBATCH -N 2
#SBATCH -G 8
#SBATCH -t 30
#SBATCH --qos=debug

module load totalview
tvconnect srun b.out
```

- When the batch script starts, **tvconnect** blocks until a **totalview** accepts the reverse connect request
- On the login node, load the “totalview” module and start **totalview**

```
module load totalview
totalview
```

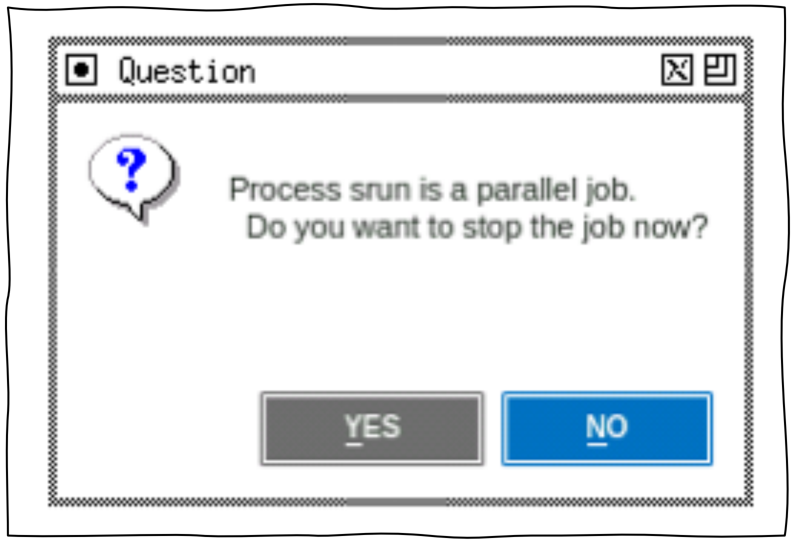
Debugging on Perlmutter (Batch Start-up)

- TotalView will “Listen For Reverse Connections” by default, but make sure the option is enabled
- When the batch script executes the **tvconnect** command, TotalView will post a dialog
- Select “Yes” to connect TotalView to the batch job



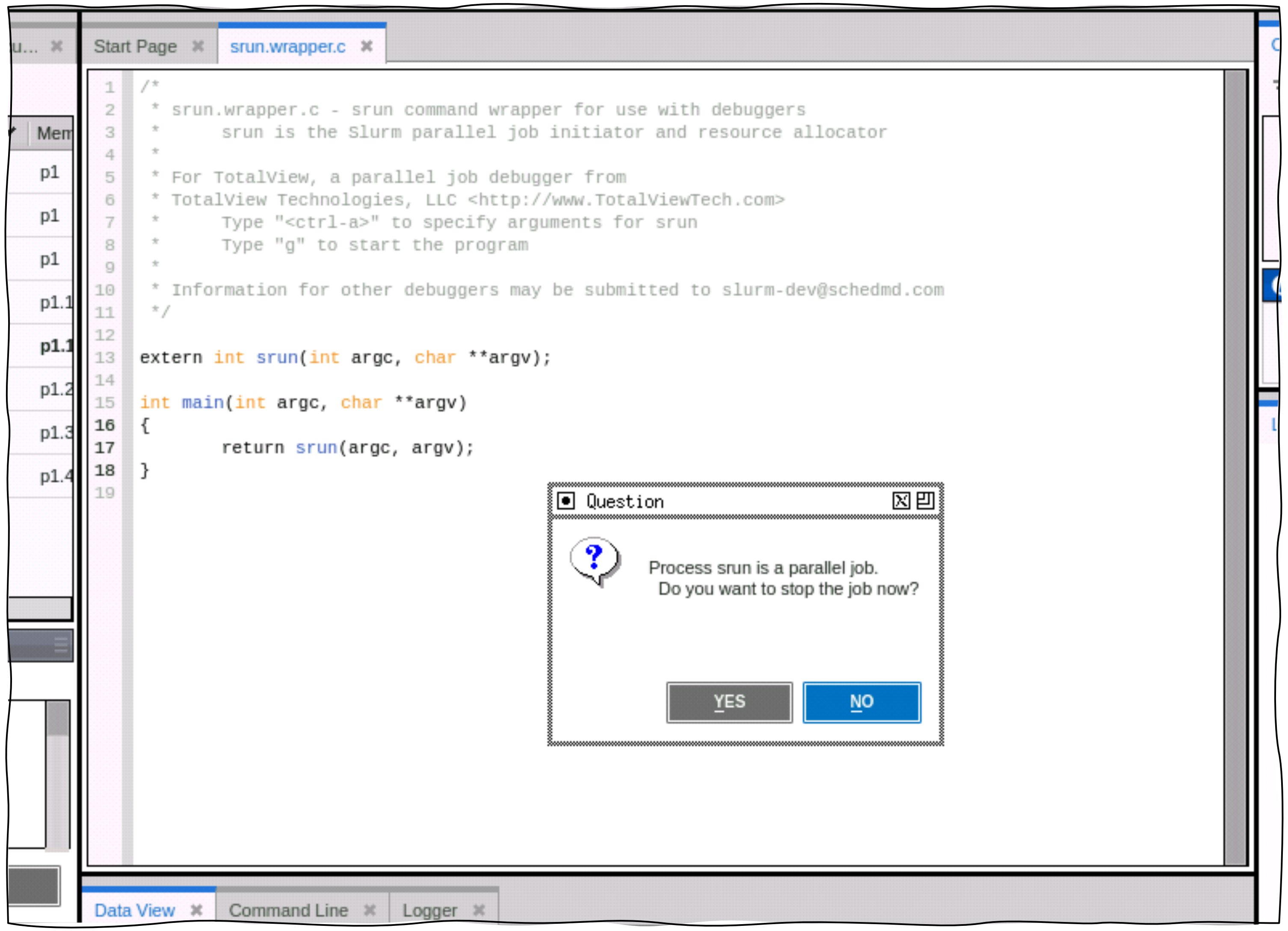
Debugging on Perlmutter (Common to Interactive/Batch)

- Once TotalView starts-up on **srun**, the following steps are common to interactive / batch debugging
- Typically
  - Select “**Go**” to start **srun**
  - **srun** will launch the parallel program
  - TotalView detects the parallel program launch and attaches to the MPI processes
- When the jobs goes parallel,
  TotalView will post a dialog



Stop the job when it goes parallel?

- Click “Yes” to stop the parallel job, which is useful if you want to
  - Navigate to source files / functions
  - Set breakpoints
- Click “No” to allow the job to run, which is useful if you
  - Have saved breakpoints from a previous session
  - Know the program is going to crash (SEGV, etc.)

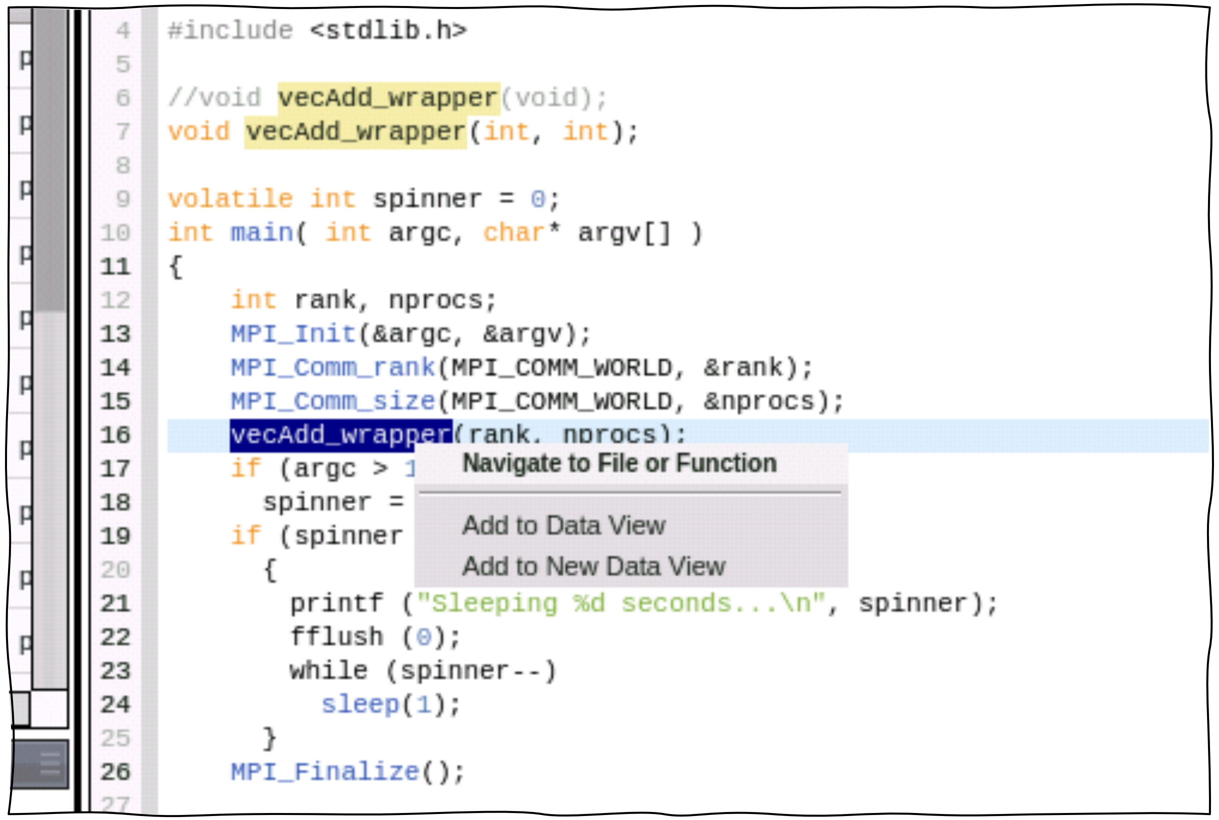


TotalView will focus on main() in rank 0

Start Page | _dl_debug_state | vecAdd.cxx

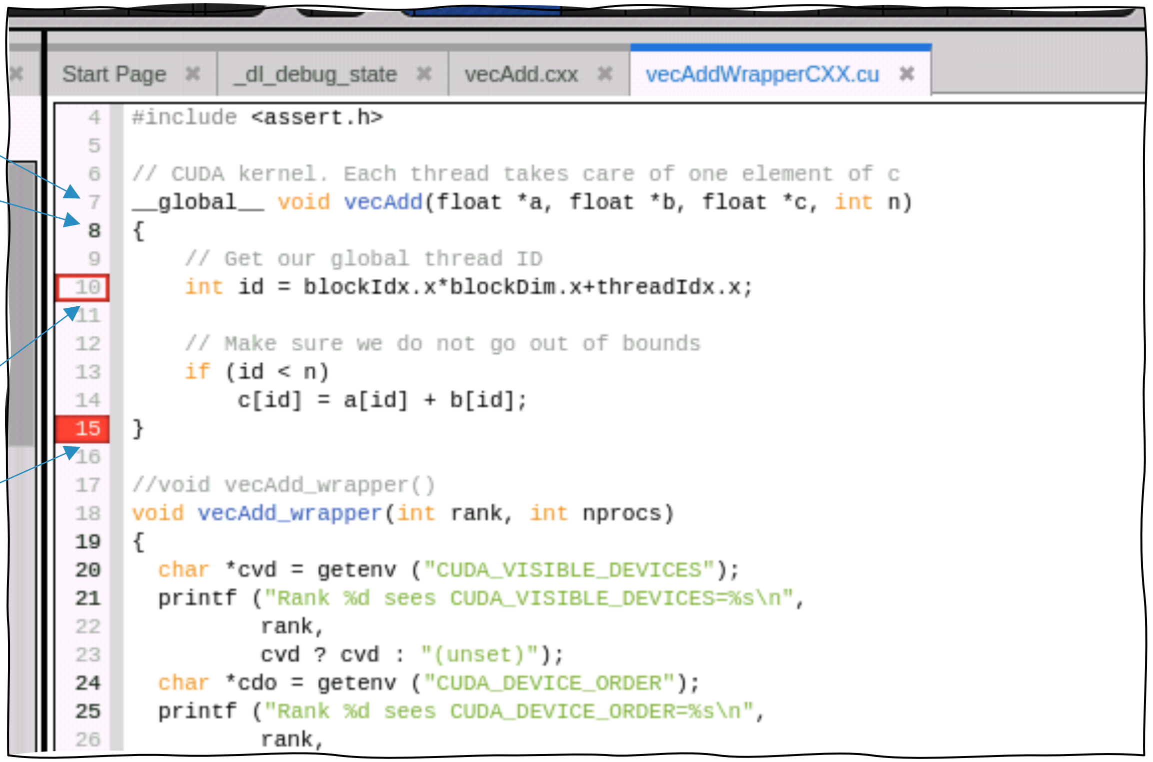
```
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>
#include <stdlib.h>

//void vecAdd_wrapper(void);
void vecAdd_wrapper(int, int);

volatile int spinner = 0;
int main( int argc, char* argv[] )
{
    int rank, nprocs;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    vecAdd_wrapper(rank, nprocs);
    if (argc > 1)
      spinner = atoi (argv[1]);
    if (spinner > 0)
      {
        printf ("Sleeping %d seconds...\n", spinner);
        fflush (0);
        while (spinner--)
          sleep(1);
      }
    MPI_Finalize();

    return 0;
}
```

Navigate to a file or function you want to debug

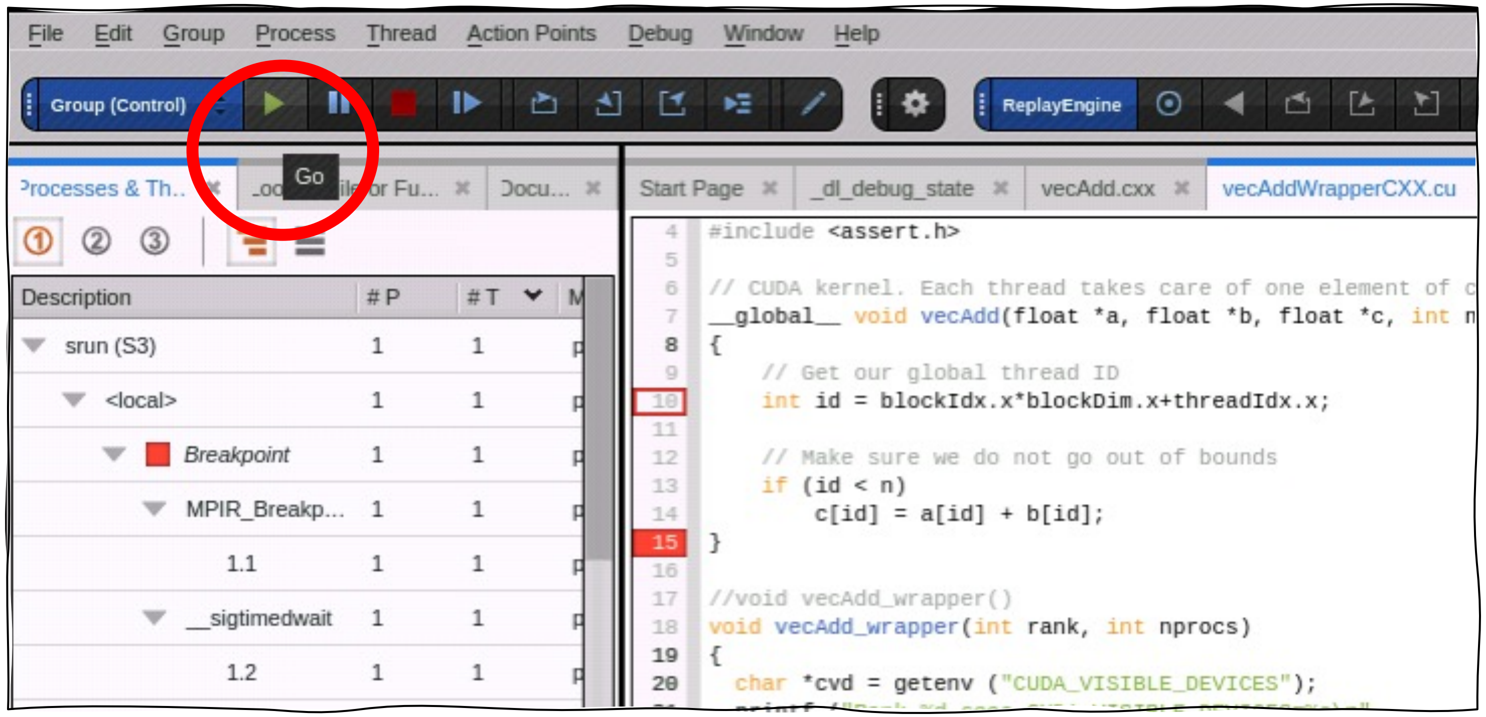


Find the CUDA kernel and select a line number to plant a breakpoint

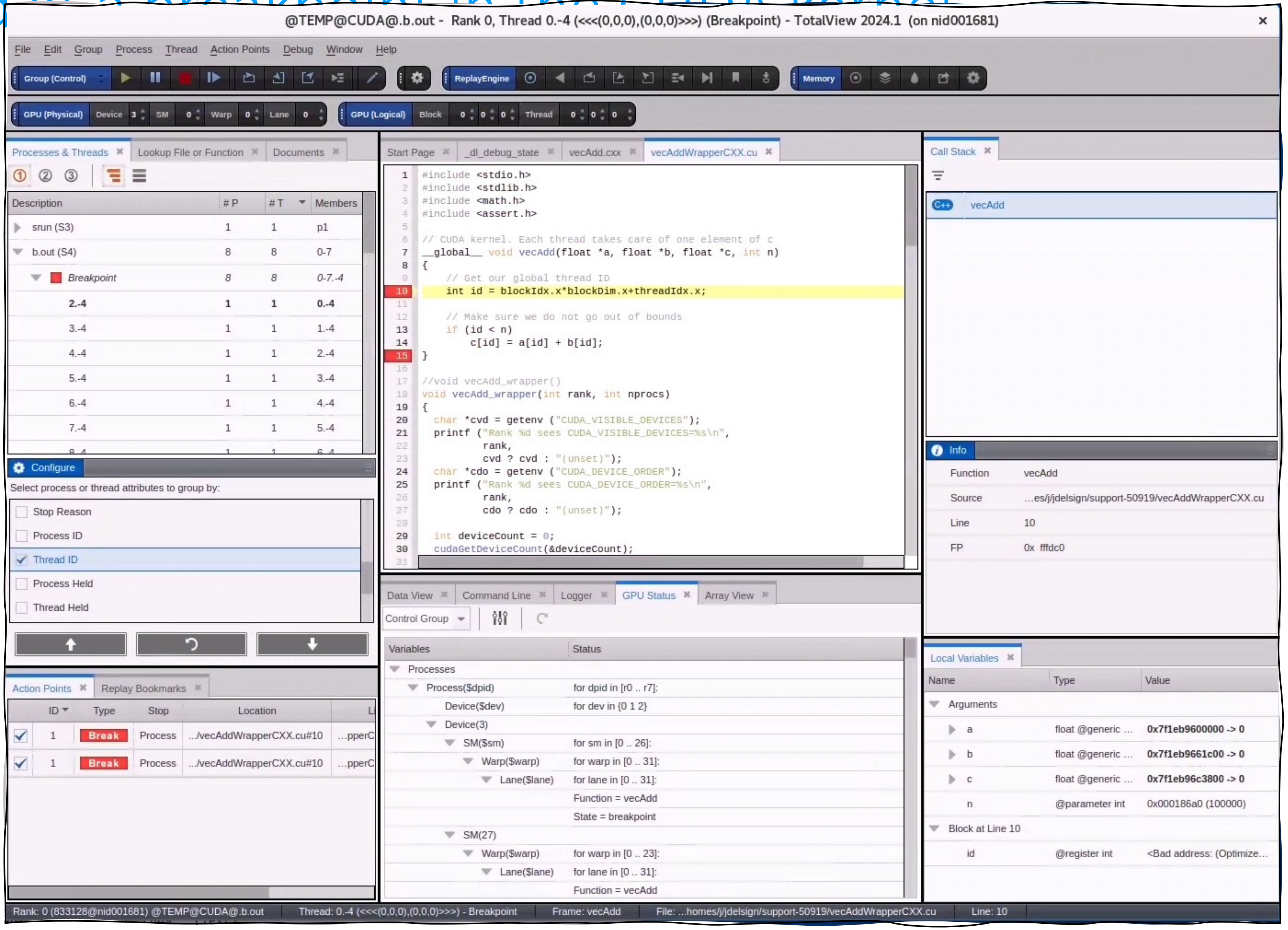
- Line numbers indicate if there's code at that line
  - Pale line numbers indicate no code (yet)
  - Bold line numbers indicate code
- CUDA code is *dynamically* loaded at runtime, so TotalView does not have any debug information *until* the CUDA kernel is launched
- Select a line number in the CUDA kernel that will have CUDA code loaded
  - Hollow breakpoint markers indicate no code *yet*
  - Solid breakpoint markers indicate code
- Source line information for a source file is *unified* for both GPU and CPU code



Click the “Go” button to run the application and launch the kernel

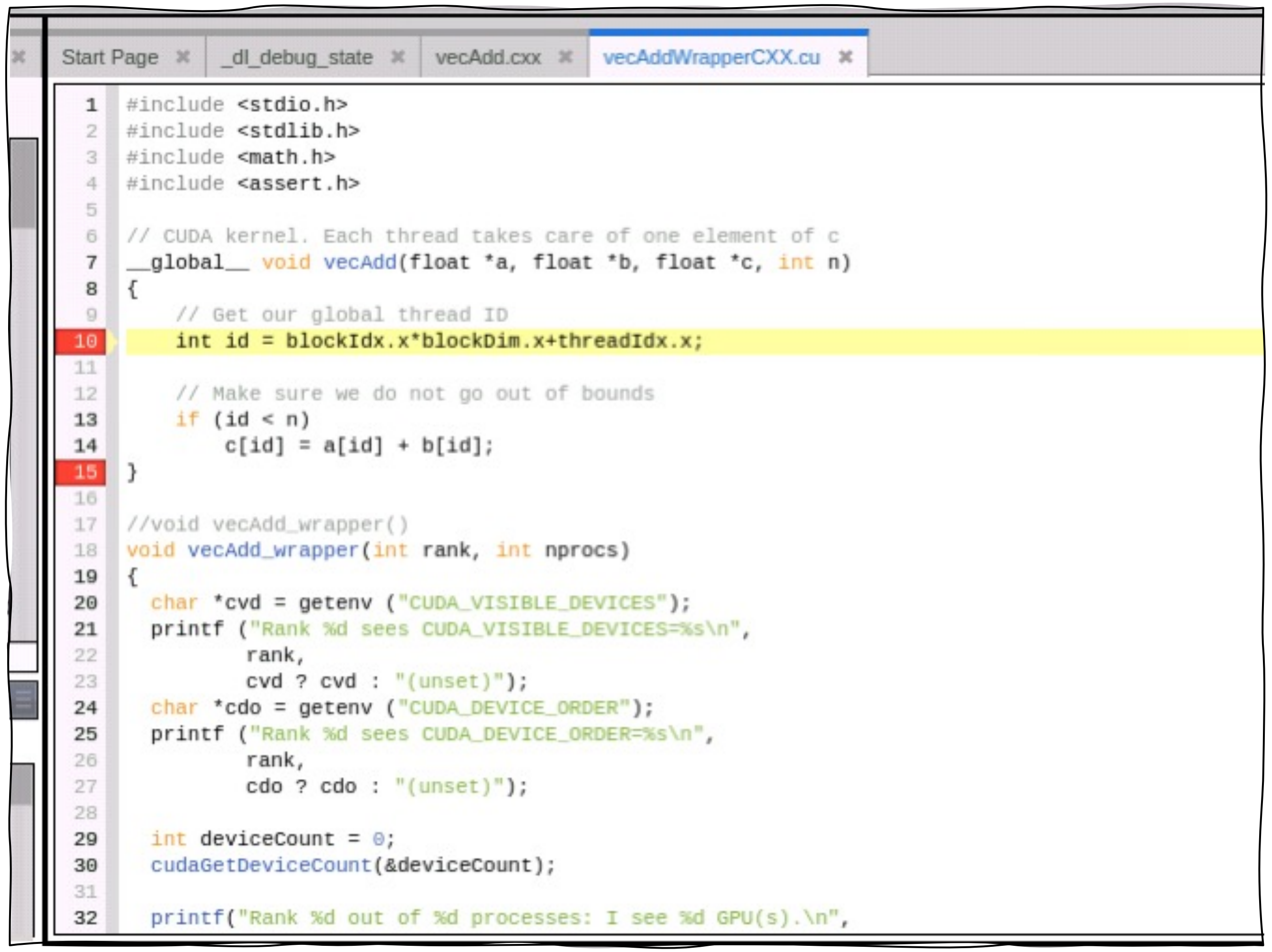


Stopped at a breakpoint in the CUDA kernel



Source view stopped in a CUDA kernel

- Line number information for the GPU code is *unified* with the CPU code
- The hollow breakpoint marker turns solid, indicating that there is now code at that line
- The PC arrow and highlighted source line indicates where the warp is stopped



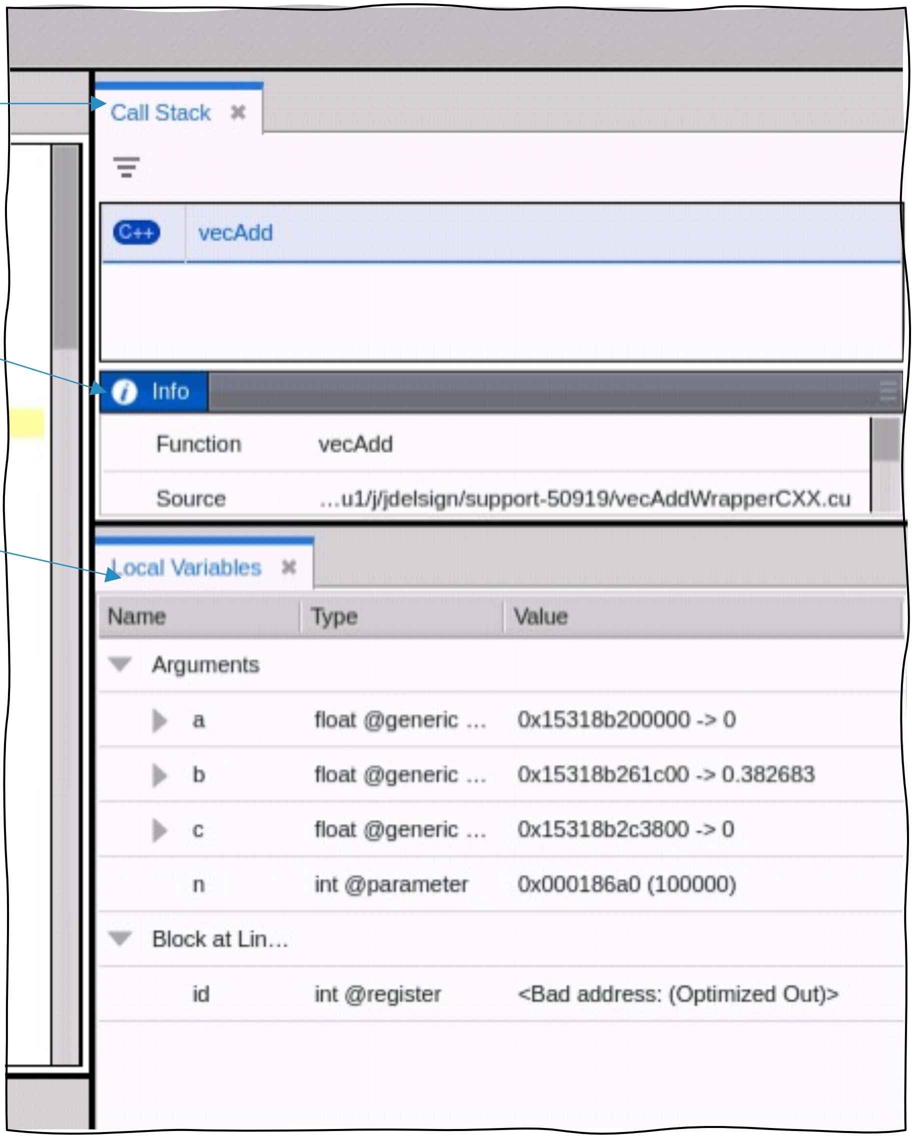
GPU thread focus and navigation controls

- “GPU (Logical)” control displays and allows focusing on a specific Block and Thread

- “GPU (Physical)” control displays and allows focusing on a specific Device, SM, Warp, and Lane

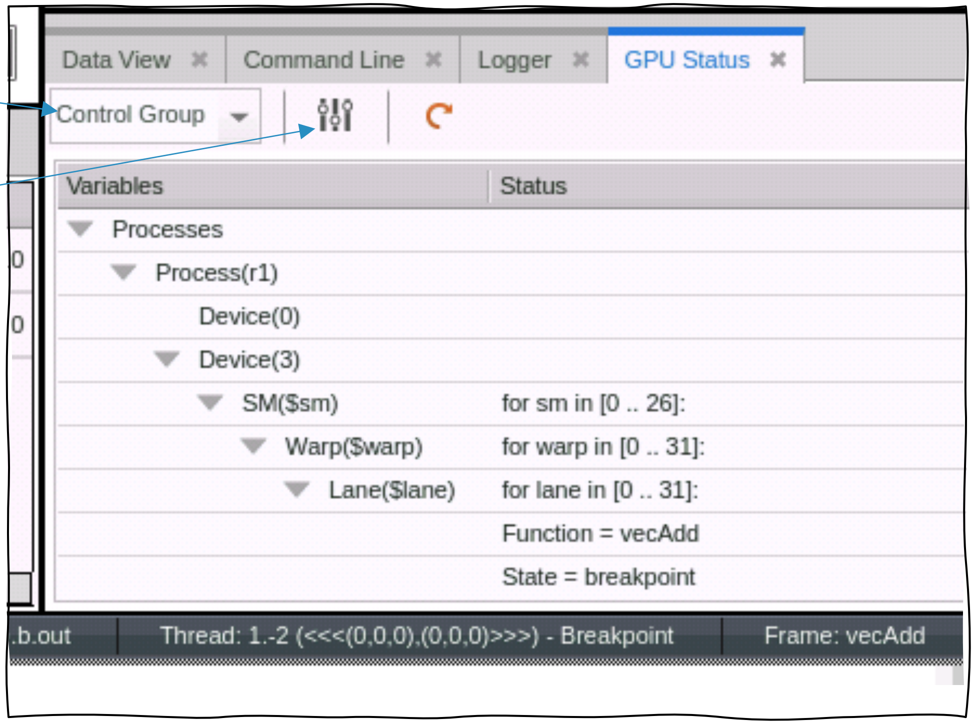
CUDA stack backtrace and local variables

- Call Stack
- Open the drawer for details
- Local Variables

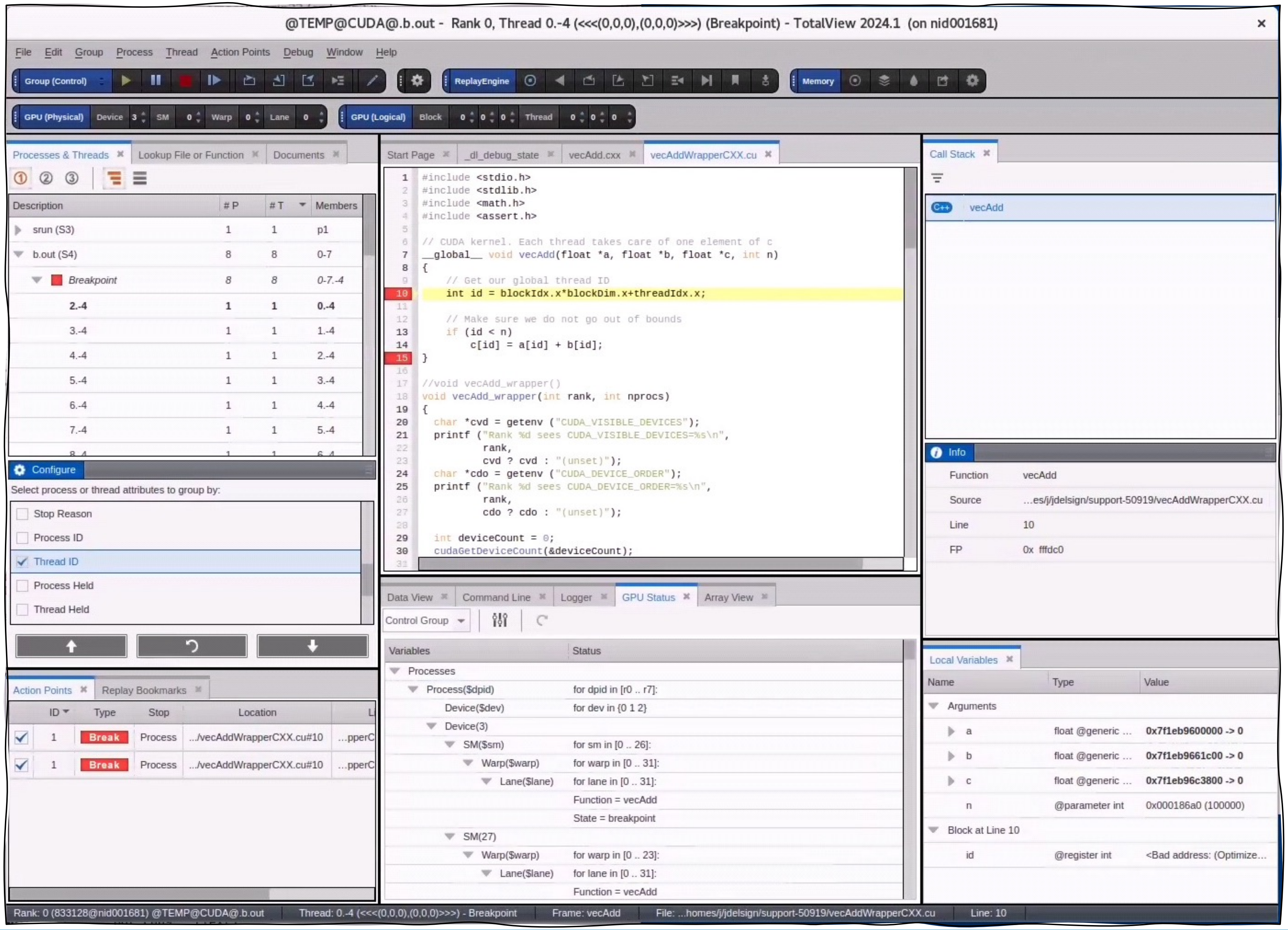


GPU Status view

- The "GPU Status" view displays an aggregated overview of one or more of the GPUs in the whole job, in a single process, or on a single GPU
- The "GPU Status" view controls allow
  - Selecting the set of properties to display
  - Aggregation by the selected properties
  - Sorting by the selected properties
  - Creating compound filters to include/exclude properties that are equal, not equal, greater, etc.
- Allows you to get a "big picture" of the state of one or more of the GPUs in your job



Demo



@TEMP@CUDA@.b.out - Rank 0, Thread 0.-4 (<<<(0,0,0),(0,0,0)>>>) (Breakpoint) - TotalView 2024.1 (on nid001681)
File Edit Group Process Thread Action Points Debug Window Help
Group (Control)
ReplayEngine
Memory
GPU (Physical) Device 3 SM 0 Warp 0 Lane 0
GPU (Logical) Block 0 0 0 Thread 0 0 0
Processes & Threads
Lookup File or Function
Documents
Description # P # T Members
srun (S3) 1 1 p1
b.out (S4) 8 8 0-7
Breakpoint 8 8 0-7.-4
2.-4 1 1 0.-4
3.-4 1 1 1.-4
4.-4 1 1 2.-4
5.-4 1 1 3.-4
6.-4 1 1 4.-4
7.-4 1 1 5.-4
Configure
Select process or thread attributes to group by:
Stop Reason
Process ID
Thread ID
Process Held
Thread Held
Action Points
Replay Bookmarks
ID Type Stop Location
1 Break Process .../vecAddWrapperCXX.cu#10 ...pperC
1 Break Process .../vecAddWrapperCXX.cu#10 ...pperC
Start Page
_dl_debug_state
vecAdd.cxx
vecAddWrapperCXX.cu
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
// Get our global thread ID
int id = blockIdx.x*blockDim.x+threadIdx.x;
// Make sure we do not go out of bounds
if (id < n)
c[id] = a[id] + b[id];
}
//void vecAdd_wrapper()
void vecAdd_wrapper(int rank, int nprocs)
{
char *cvd = getenv ("CUDA_VISIBLE_DEVICES");
printf ("Rank %d sees CUDA_VISIBLE_DEVICES=%s\n",
rank,
cvd ? cvd : "(unset)");
char *cdo = getenv ("CUDA_DEVICE_ORDER");
printf ("Rank %d sees CUDA_DEVICE_ORDER=%s\n",
rank,
cdo ? cdo : "(unset)");
int deviceCount = 0;
cudaGetDeviceCount(&deviceCount);
Data View
Command Line
Logger
GPU Status
Array View
Control Group
Variables Status
Processes
Process($dpid) for dpid in [r0 .. r7]:
Device($dev) for dev in {0 1 2}
Device(3)
SM($sm) for sm in [0 .. 26]:
Warp($warp) for warp in [0 .. 31]:
Lane($lane) for lane in [0 .. 31]:
Function = vecAdd
State = breakpoint
SM(27)
Warp($warp) for warp in [0 .. 23]:
Lane($lane) for lane in [0 .. 31]:
Function = vecAdd
Call Stack
vecAdd
Info
Function vecAdd
Source ...es/j/jdelsign/support-50919/vecAddWrapperCXX.cu
Line 10
FP 0x fffdc0
Local Variables
Name Type Value
Arguments
a float @generic ... 0x7f1eb9600000 -> 0
b float @generic ... 0x7f1eb9661c00 -> 0
c float @generic ... 0x7f1eb96c3800 -> 0
n @parameter int 0x000186a0 (100000)
Block at Line 10
id @register int <Bad address: (Optimize...
Rank: 0 (833128@nid001681) @TEMP@CUDA@.b.out
Thread: 0.-4 (<<<(0,0,0),(0,0,0)>>>) - Breakpoint
Frame: vecAdd
File: ...homes/j/jdelsign/support-50919/vecAddWrapperCXX.cu
Line: 10

Batch Debugging with TVScript

tvscript

- A straightforward language for unattended and/or batch debugging with TotalView and/or MemoryScape
- Usable whenever jobs need to be submitted or batched
- Can be used for automation
- A more powerful version of printf, no recompilation necessary between runs
- Schedule automated debug runs with *cron* jobs
- Expand its capabilities using TCL

tvscript

tvscript [*options*] [*filename*] [-a *program_args*]

options

TotalView and tvscript command-line options.

filename

The program being debugged.

-a *program_args*

Program arguments.

tvscript

- All of the following information is provided by default for each print
  - Process id
  - Thread id
  - Rank
  - Timestamp
  - Event/Action description
- A single output file is written containing all of the information regardless of the number of processes/threads being debugged

Supported tvscript events

Event Type	Event	Definition
General event	**any_event**	A generated event occurred.
Memory debugging event	**addr_not_at_start**	Program attempted to free a block using an incorrect address.
	alloc_not_in_heap	The memory allocator returned a block not in the heap; the heap may be corrupt.
	alloc_null	An allocation either failed or returned NULL; this usually means that the system is out of memory.
	alloc_returned_bad_alignment	The memory allocator returned a misaligned block; the heap may be corrupt.
	any_memory_event	A memory event occurred.
	bad_alignment_argument	Program supplied an invalid alignment argument to the heap manager.
	double_alloc	The memory allocator returned a block currently being used; the heap may be corrupt.
	double_dealloc	Program attempted to free an already freed block.
	free_not_allocated	Program attempted to free an address that is not in the heap.
	guard_corruption	Program overwrote the guard areas around a block.

Supported tvscript events

Event Type	Event	Definition
	hoard_low_memory_threshold	Hoard low memory threshold crossed.
	realloc_not_allocated	Program attempted to reallocate an address that is not in the heap.
	rz_overrun	Program attempted to access memory beyond the end of an allocated block.
	rz_underrun	Program attempted to access memory before the start of an allocated block.
	rz_use_after_free	Program attempted to access a block of memory after it has been deallocated.
	rz_use_after_free_overrun	Program attempted to access memory beyond the end of a deallocated block.
	rz_use_after_free_underrun	Program attempted to access memory before the start of a deallocated block.
	termination_notification	The target is terminating.
Source code debugging event	**actionpoint**	A thread hit an action point.
	error	An error occurred.
Reverse debugging	**stopped_at_end**	The program is stopped at the end of execution and is about to exit.

Supported tvscript actions

Action Type	Action	Definition
Memory debugging actions	**check_guard_blocks**	Checks all guard blocks and write violations into the log file.
	list_allocations	Writes a list of all memory allocations into the log file.
	list_leaks	Writes a list of all memory leaks into the log file.
	save_html_heap_status_source_view	Generates and saves an HTML version of the Heap Status Source View Report.
	save_memory_debugging_file	Generates and saves a memory debugging file.
	save_text_heap_status_source_view	Generates and saves a text version of the Heap Status Source View Report.
Source code debugging actions	**display_backtrace** [**-level***level-num*] [*num_levels*] [*options*]	Writes the current stack backtrace into the log file. **-level** *level-num* sets the level at which information starts being logged. *num_levels* restricts output to this number of levels in the call stack. If you do not set a level, **tvscript** displays all levels in the call stack. *options* is one or more of the following: **-[no]show_arguments** **-[no]show_f**p **-[no]show_fp_registers** **-[no]show_image** **-[no]show_locals** **-[no]show_pc** **-[no]show_registers**

Supported tvscript actions

Action Type	Action	Definition
	print [**-slice** {*slice_exp*}] {*variable* | *exp*}	Writes the value of a variable or an expression into the log file. If the variable is an array, the **-slice** option limits the amount of data defined by slice_exp. A slice expression is a way to define the slice, such as **var[100:130]**in C and C++. (This displays all values from **var[100]**to **var[130]**.) To display every fourth value, add an additional argument; for example,**var[100:130:4]**. For additional information, see *"Examining Arrays"*in the *TotalView for HPC User Guide*.
Reverse debugging actions	**enable_reverse_debugging**	Turns on ReplayEngine reverse debugging and begins recording the execution of the program.
	save_replay_recording_file	Saves a ReplayEngine recording file. The filename is of the form `<ProcessName>-<PID>_<DATE>.<INDEX>.recording`.

tvscript examples

Simple example

```
tvscript \
-create_actionpoint "method1=>display_backtrace -show_arguments" \
-create_actionpoint "method2#37=>display_backtrace \
             -show_locals -level 1" \
-event_action "error=>display_backtrace -show_arguments \
             -show_locals" \
-display_specifiers "noshow_pid,noshow_tid" \
-maxruntime "00:00:30" \
~/work/filterapp /filterapp -a 20
```

MPI example

```
tvscript -mpi "Open MPI" -tasks 4 \
-create_actionpoint \
"hello.c#14=>display_backtrace" \
~/tests/MPI_hello
```

tvscript examples

Memory Debugging example

```
tvscript -maxruntime "00:00:30" \
-event_action "any_event=save_memory_debugging_file" \
-guard_blocks -hoard_freed_memory -detect_leaks \
~/work/filterapp -a 20
```

ReplayEngine example

```
tvscript \
-create_actionpoint "main=>enable_reverse_debugging" \
-event_action "stopped_at_end=>save_replay_recording_file" \
filterapp
```

Demo

- TVScript demo (tvscript –script_file file tvscript_example.tvd ex2)

Common TotalView Usage Hints

Common TotalView Usage Hints

- TotalView can’t find the program source
  - Did you compile with -g ?
  - How to adjust the TotalView search paths? Preferences -> Search Path
- Python Debugging
  - Making sure proper system debug packages are installed for Python
- X11 forwarding performance
  - If users are forwarding X11 displays through ssh TotalView UI performance can be bad
- Understanding different ways to stop program execution with TotalView Action Points
  - Using a watchpoint on a local variable
- Focus
  - Diving on a variable that is no longer in scope. Check the Local Variables window for in scope variables
  - TotalView doesn’t change focus to the thread hitting a breakpoint. Set Action Point Preferences to “Automatically focus on threads/processes at breakpoint”

Common TotalView Usage Hints (cont.)

- MPI Debugging
  - Differences in launching MPI job from within the TotalView UI vs the command line.
  - TotalView runs an MPI program without stopping. Set the Parallel Preferences to “Ask What To Do” in After Attach Behavior
  - Using wrong attributes in processes and threads view
- Reverse Debugging
  - Running out of memory by not setting the maximum memory allocated to ReplayEngine
  - Defer turning on reverse debugging until later in program execution to avoid slow initialization phases
  - Adjust reverse debugging circular buffer size to reduce resources
- Memory Debugging
  - Starting with All memory debugging options enabled rather than Low
  - Not setting a size restriction for Red Zones
  - Issues with getting memory debugging turned on in an MPI job. May have to set LD_PRELOAD environment variable or worst case, prelink HIA

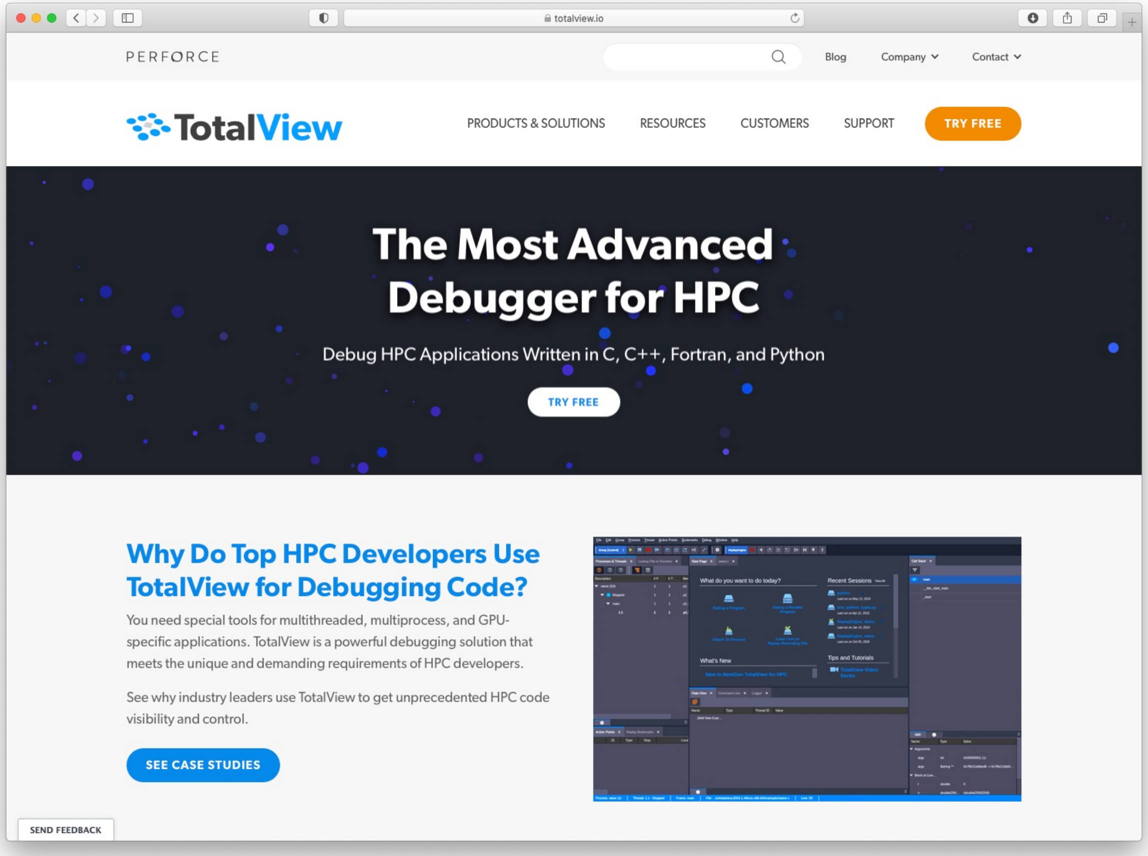
Common TotalView Usage Hints (cont.)

- Differences in functionality between new UI and classic UI
  - How to switch between them. Preferences -> Display or totalview –newUI and totalview -oldUI
  - Where the gaps still are in functionality
- Reverse Connect with tvconnect
  - When I use Reverse Connect I get the following obscure message: myProgram *is an invalid or incompatible executable file format for the target platform*
  - The message indicates an incompatible file format but most often this occurs if the program provided to tvconnect for TotalView to debug cannot be found. The easiest way to resolve problem is to provide the full path to the target application, e.g., *tvconnect /home/usr/myProgram*
- How do I get help?
  - How to submit a support ticket? techsupport@roguewave.com
  - Where is TV documentation (locally and on the internet). https://help.totalview.io/
  - Are there videos I can watch to learn how to use TotalView? https://totalview.io/support/video-tutorials

TotalView Resources and Documentation

TotalView Resources and Documentation

- TotalView website:
  https://totalview.io
- TotalView documentation:
  - https://help.totalview.io
  - User Guides: Debugging, Memory Debugging and Reverse Debugging
  - Reference Guides: Using the CLI, Transformations, Running TotalView
- Blog:
  https://totalview.io/blog
- Video Tutorials:
  https://totalview.io/support/video-tutorials



Q&A

Contact us

- Bill Burns (Senior Director of Software Engineering and Product Manager)

  bburns@perforce.com

- John DelSignore (TotalView Chief Architect)

  jdelsignore@perforce.com

- Scot Halverson (NVIDIA Solutions Architect)

  shalverson@nvidia.com

- Peter Thompson (Senior Support Engineer)

  pthompson@perforce.com

- Bruce Ryan (Senior Account Executive)

  bryan@perforce.com

- Ken Hill (Senior Sales Engineer)

  khill@perforce.com