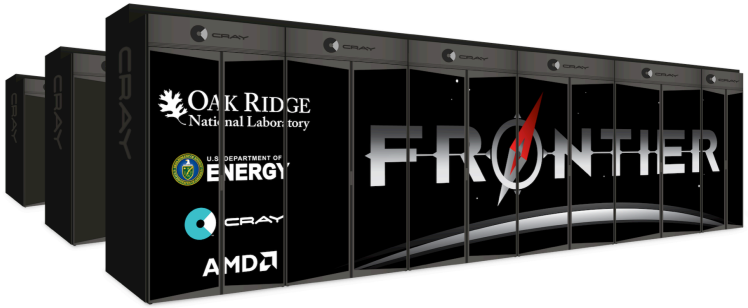
Performance Analysis of GPU-accelerated Applications with HPCToolkit

John Mellor-Crummey
Rice University

2025 Energy HPC Conference
February 28, 2025

RICE
STEP SOFTWARE TOOLS ECOSYSTEM PROJECT
U.S. DEPARTMENT OF ENERGY | Office of Science
NNSA National Nuclear Security Administration

DOE's GPU-Accelerated Exascale Platforms



- Frontier compute nodes (OLCF)
  - 1 AMD EPYC "Trento" CPU
  - 4 MI250X AMD Radeon Instinct GPUs
  - 4 Slingshot 11 endpoints
  - Unified memory architecture

- Aurora compute nodes (ALCF)
  - 2 Intel Xeon "Sapphire Rapids" processors
  - 6 Intel Data Center GPU Max 1500
  - 8 Slingshot 11 endpoints
  - Unified memory architecture

- El Capitan compute nodes (LLNL)
  - 4 AMD MI300 APU
  - 4 Slingshot 11 endpoints
  - Unified memory architecture

Outline

- Introduction to HPCToolkit performance tools
  - —Overview of HPCToolkit components and their workflow
  - —HPCToolkit's graphical user interfaces
- Analyzing the performance of GPU-accelerated codes with HPCToolkit
  - Slides: Exawind (AMReX)
  - —Slides: LAMMPS at Exascale (Kokkos)
  - —Demo: GAMESS (OpenMP)
  - —Hands-on: Quicksilver (CUDA)
  - —Hands-on: TeaLeaf

Linux Foundation's HPCToolkit Performance Tools

Collect profiles and traces of unmodified parallel CPU and GPU-accelerated applications

Understand where an application spends its time and why

- call path profiles associate metrics with application source code contexts
  - analyze instruction-level performance within GPU kernels and attribute it to your source code
- hierarchical traces to understand execution dynamics

Parallel programming models

- across nodes: MPI, SHMEM, UPC++, …
- within nodes: OpenMP, Kokkos, RAJA, HIP, DPC++, Sycl, CUDA, OpenACC, …

Languages

- C, C++, Fortran, Python, …

Hardware

- CPU cores and GPUs within a node
  - CPU: x86_64, Power, ARM
  - GPU: NVIDIA, AMD, Intel

Why HPCToolkit?

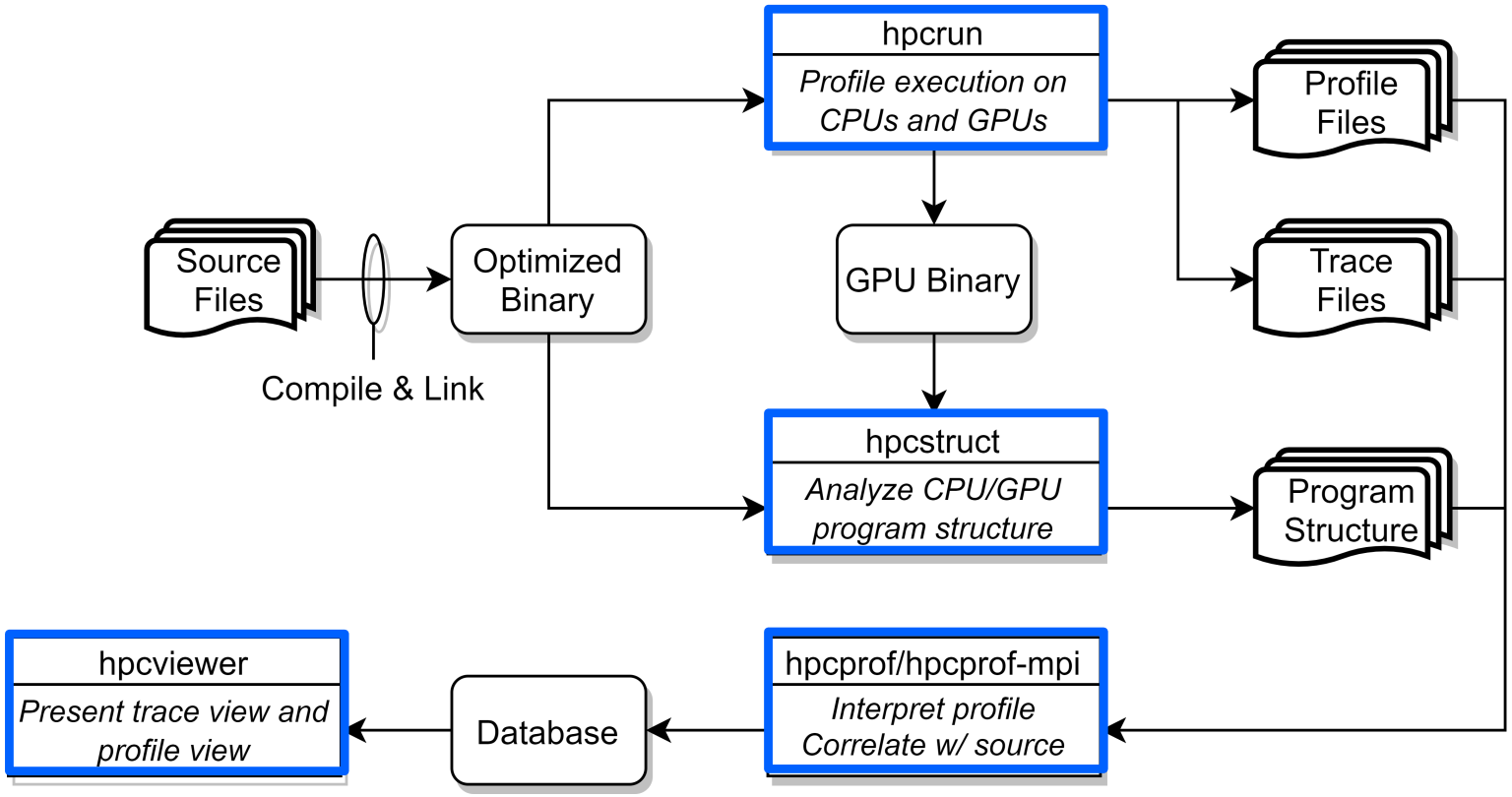
- Measure and analyze performance of CPU and GPU-accelerated applications
- Easy: profile unmodified application binaries
- Fast: low-overhead measurement
- Informative: understand where an application spends its time and why
  - —call path profiles associate metrics with application source code contexts
  - —optional hierarchical traces to understand execution dynamics
- Broad audience
  - —application developers
  - —framework developers
  - —runtime and tool developers

- Unlike vendor tools works with a wide range of CPUs and GPUs

How does HPCToolkit Differ from NVIDIA's Tools?

- NVIDIA NSight Systems
  - —tracing of CPU and GPU streams
  - —analyze traces when you open them with the GUI
    - long running traces are huge and thus extremely slow to analyze, limiting scalability
  - —designed for measurement and analysis within a node
- NVIDIA NSight Compute
  - —detailed measurement of kernels with counters and execution replay
  - —very slow measurement
  - —flat display of measurements within GPU kernels
- HPCToolkit
  - —supports more scalable tracing than Nsight Systems
    - measure exascale executions across many GPUs and nodes
  - —scalable, parallel post-mortem analysis vs. non-scalable in-GUI analysis
  - —detailed reconstruction of estimates for calling context profiles within GPU kernels

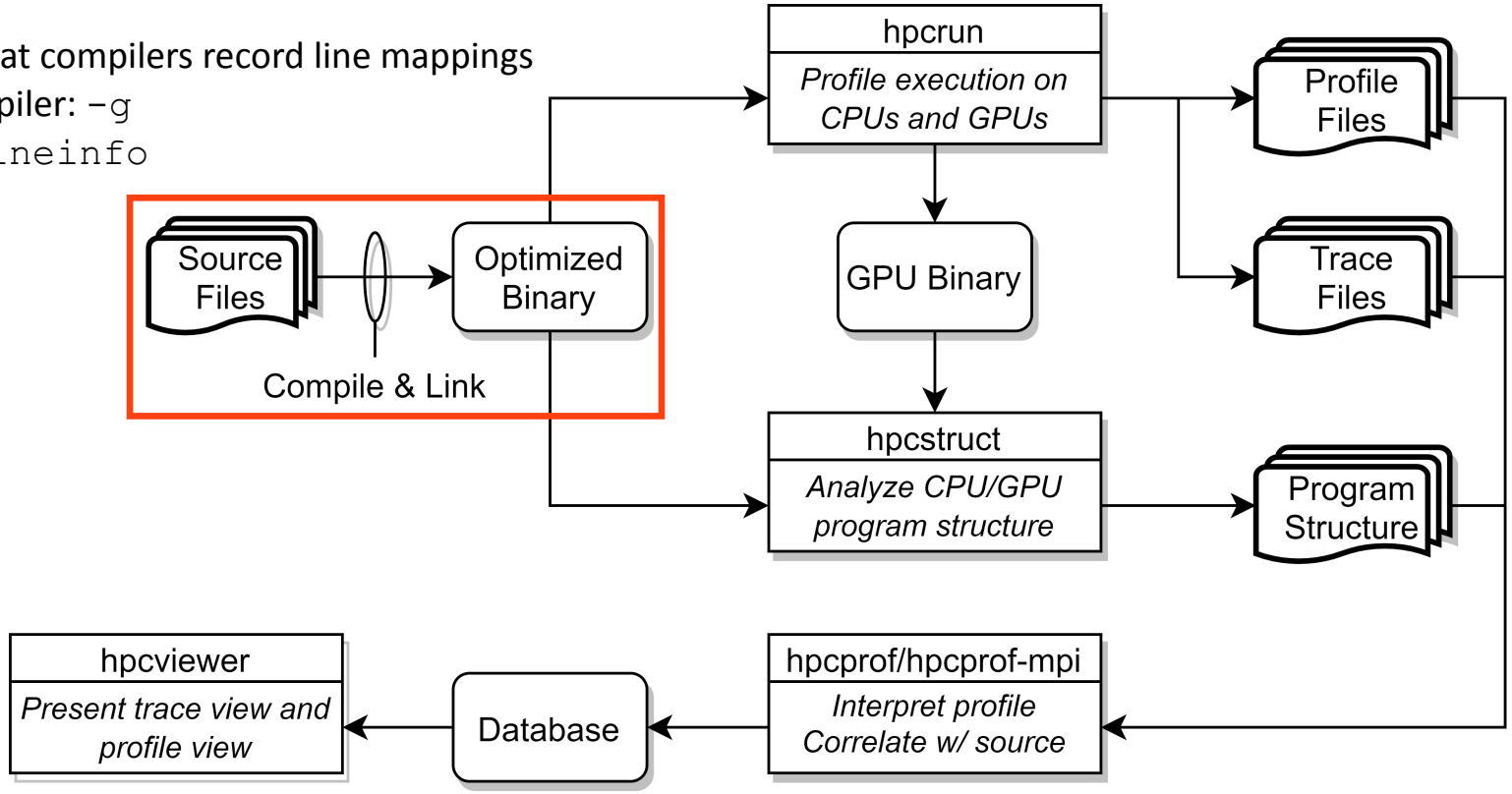
HPCToolkit's Workflow for GPU-accelerated Applications



HPCToolkit's Workflow for GPU-accelerated Applications

Step 1:

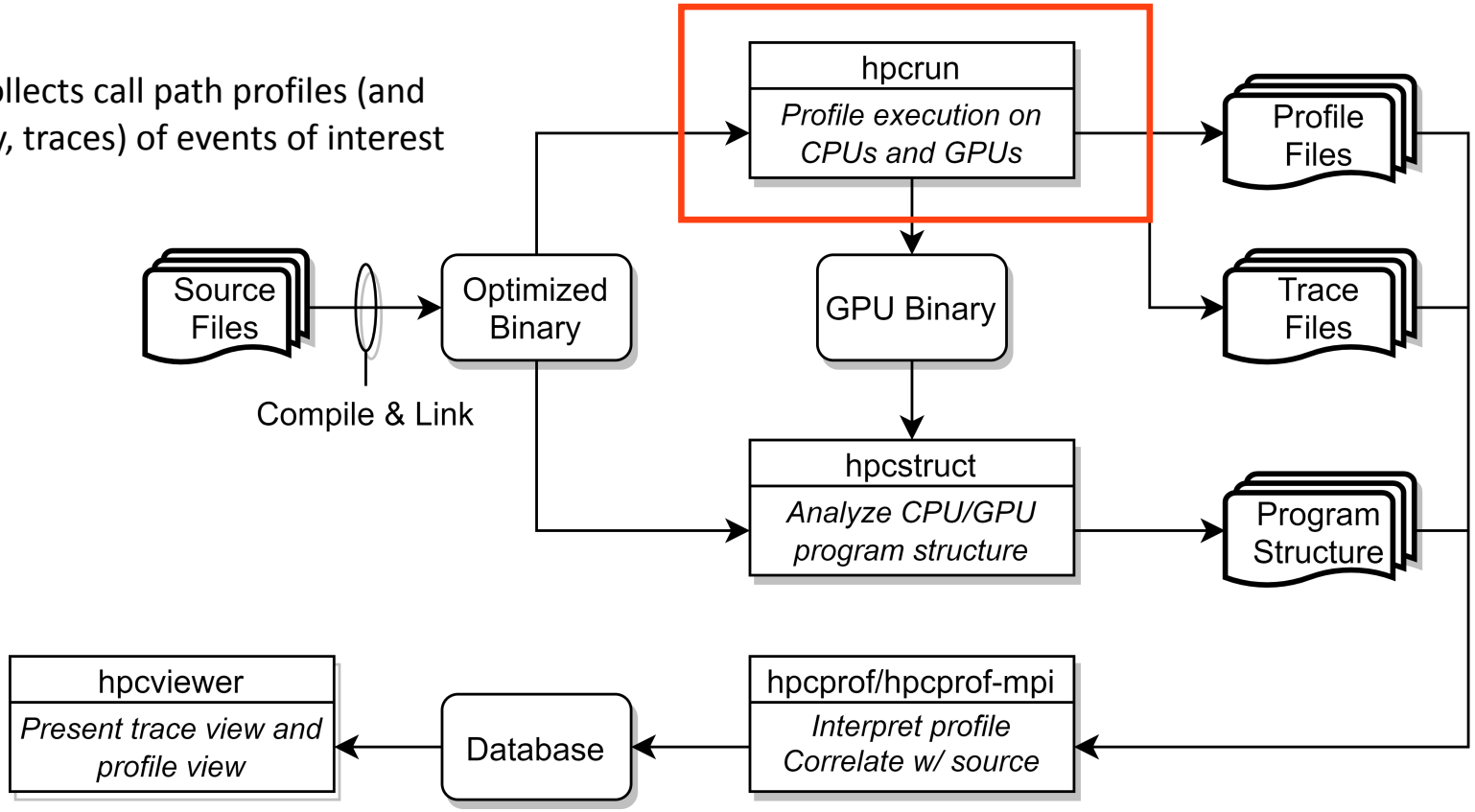
- Ensure that compilers record line mappings
- host compiler: `-g`
- nvcc: `-lineinfo`



HPCToolkit's Workflow for GPU-accelerated Applications

Step 2:

- *hpcrun* collects call path profiles (and optionally, traces) of events of interest



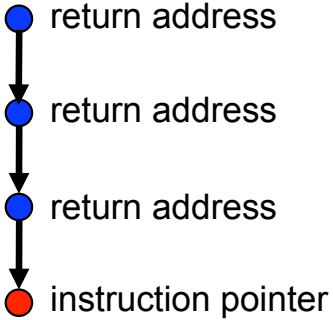
Measurement of CPU and GPU-accelerated Applications

- Sampling using Linux timers and hardware counter overflows on the CPU
- Callbacks when GPU operations are launched and (sometimes) completed
- Event stream for GPU operations
- PC Samples: NVIDIA (in progress: AMD, Intel)
- Binary instrumentation of GPU kernels on Intel GPUs for fine-grain measurement

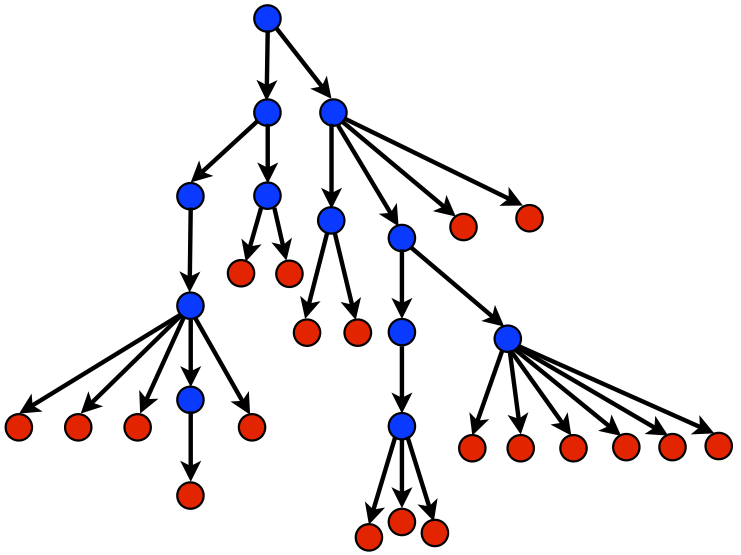
Call Stack Unwinding to Attribute Costs in Context

- Unwind when timer or hardware counter overflows
  - —measurement overhead proportional to sampling frequency rather than call frequency
- Unwind to capture context for events such as GPU kernel launches

Call path sample



Calling context tree



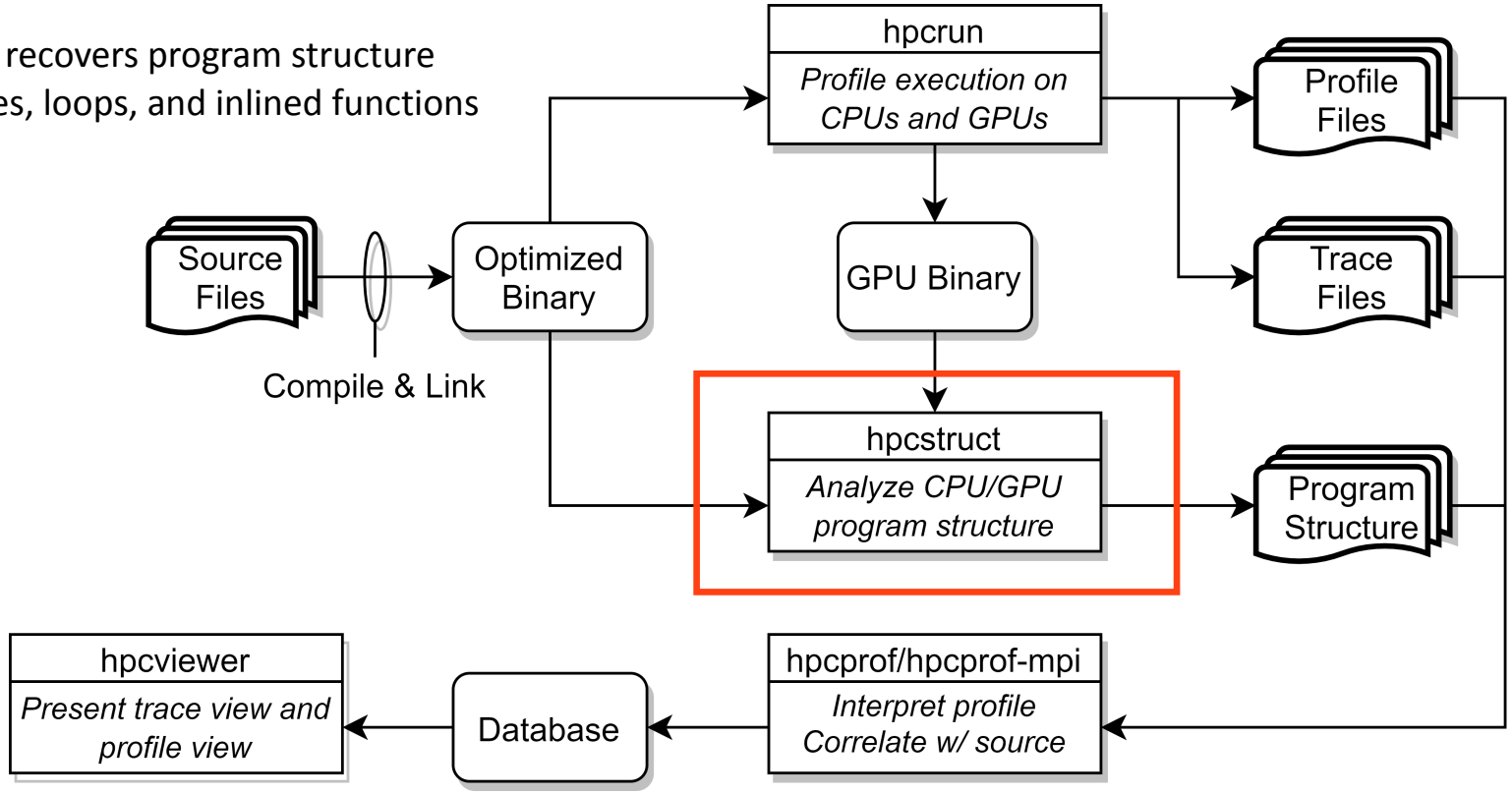
hpcrun: Measure CPU and/or GPU activity

- GPU profiling
  - `hpcrun -e gpu=xxx <app> ….` *xxx* ∈ *{nvidia,amd,opencl,level0}*
- GPU PC sampling (NVIDIA GPU only)
  - `hpcrun -e gpu=nvidia,pc <app>`
- CPU and GPU Tracing (in addition to profiling)
  - `hpcrun -e CPUTIME -e gpu=xxx -tt <app>`
- Use hpcrun with MPI on Polaris
  - `mpiexec -n <ranks> … hpcrun -e gpu=xxx <app>`

HPCToolkit's Workflow for GPU-accelerated Applications

Step 3:

- *hpcstruct* recovers program structure about lines, loops, and inlined functions



hpcstruct: Analyze CPU and GPU Binaries Using Multiple Threads

- Usage

```
hpcstruct [--gpucfg yes] <measurement-directory>
```

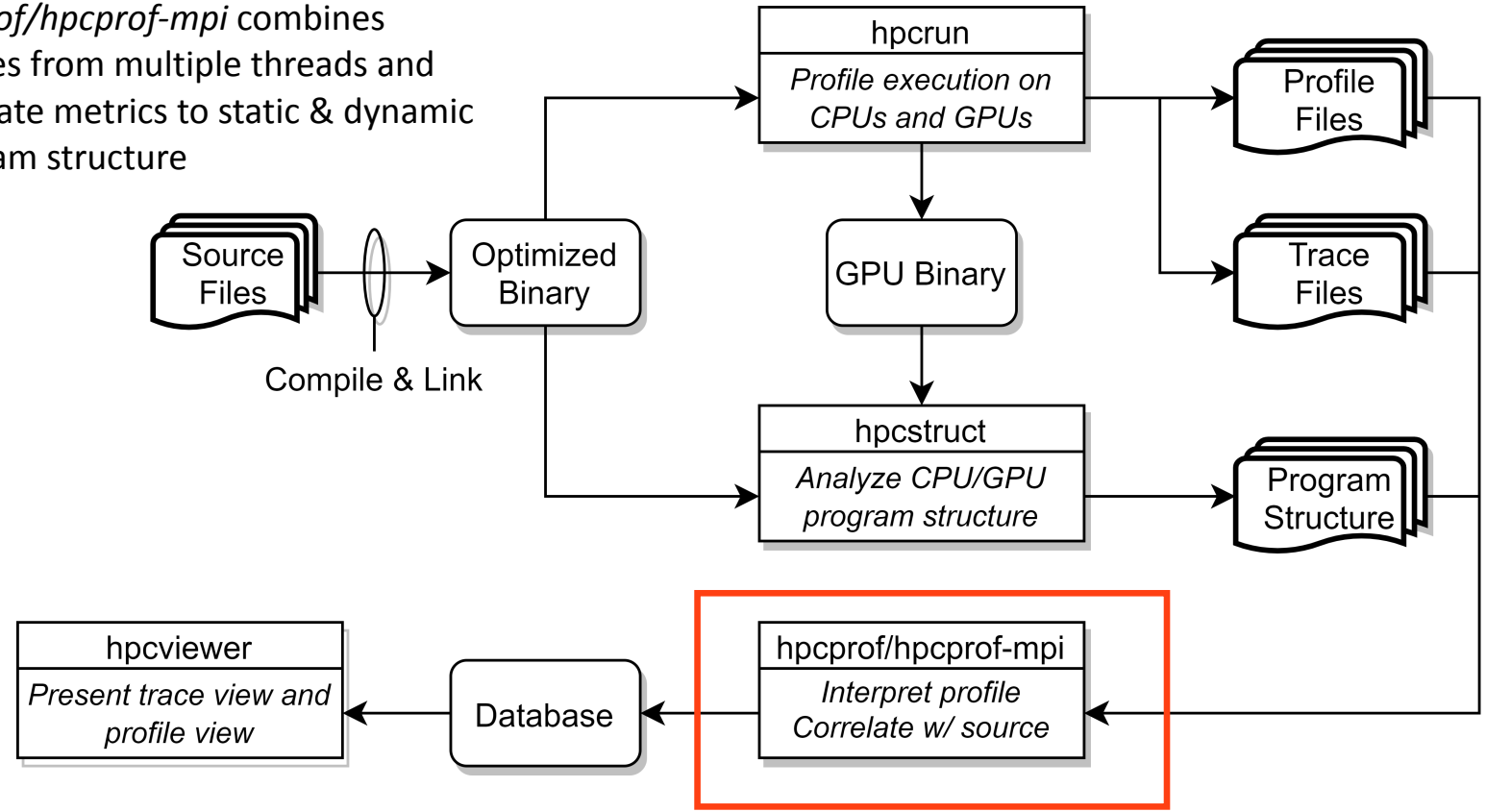
- What it does
  - Recover program structure information
    - Files, functions, inlined templates or functions, loops, source lines
  - In parallel, analyze all CPU and GPU binaries that were measured by HPCToolkit
    - —typically analyze large application binaries with 16 threads
    - —typically analyze multiple small application binaries concurrently with 2 threads each
  - Cache binary analysis results for reuse when analyzing other executions

NOTE: `--gpucfg yes` needed only for analysis of GPU binaries for interpreting PC samples on NVIDIA GPUs

HPCToolkit's Workflow for GPU-accelerated Applications

Step 4:

- *hpcprof/hpcprof-mpi* combines profiles from multiple threads and correlate metrics to static & dynamic program structure



hpcprof/hpcprof-mpi: Associate Measurements with Program Structure

- Analyze data from modest executions with multithreading (moderate scale)

```
hpcprof <measurement-directory>
```

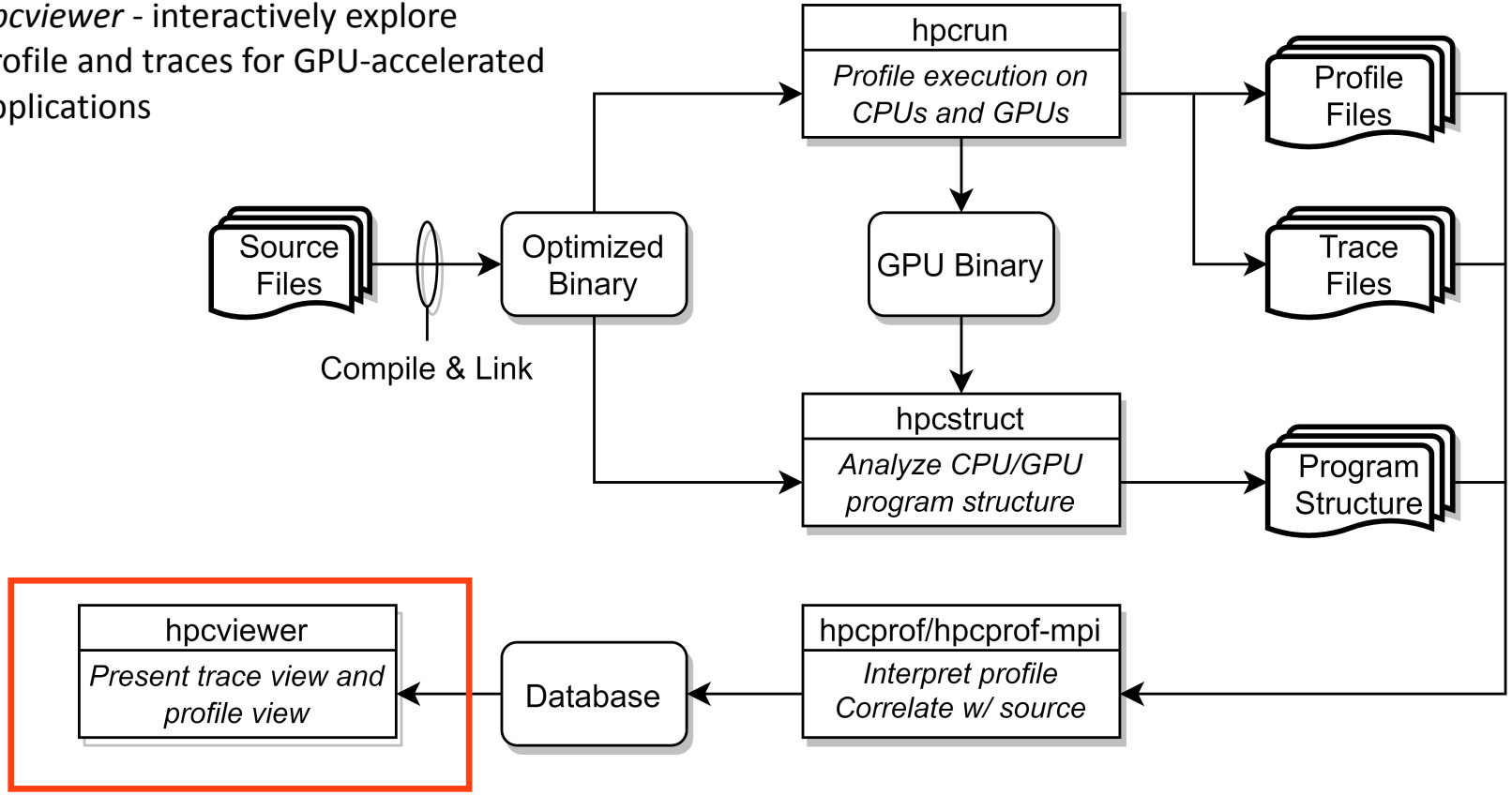
- Analyze data from large executions with distributed-memory parallelism + multithreading (large scale)

```
mpiexec -n ${NODES} --ppn 1 –depth=128 \
    hpcprof-mpi <measurement-directory>
```

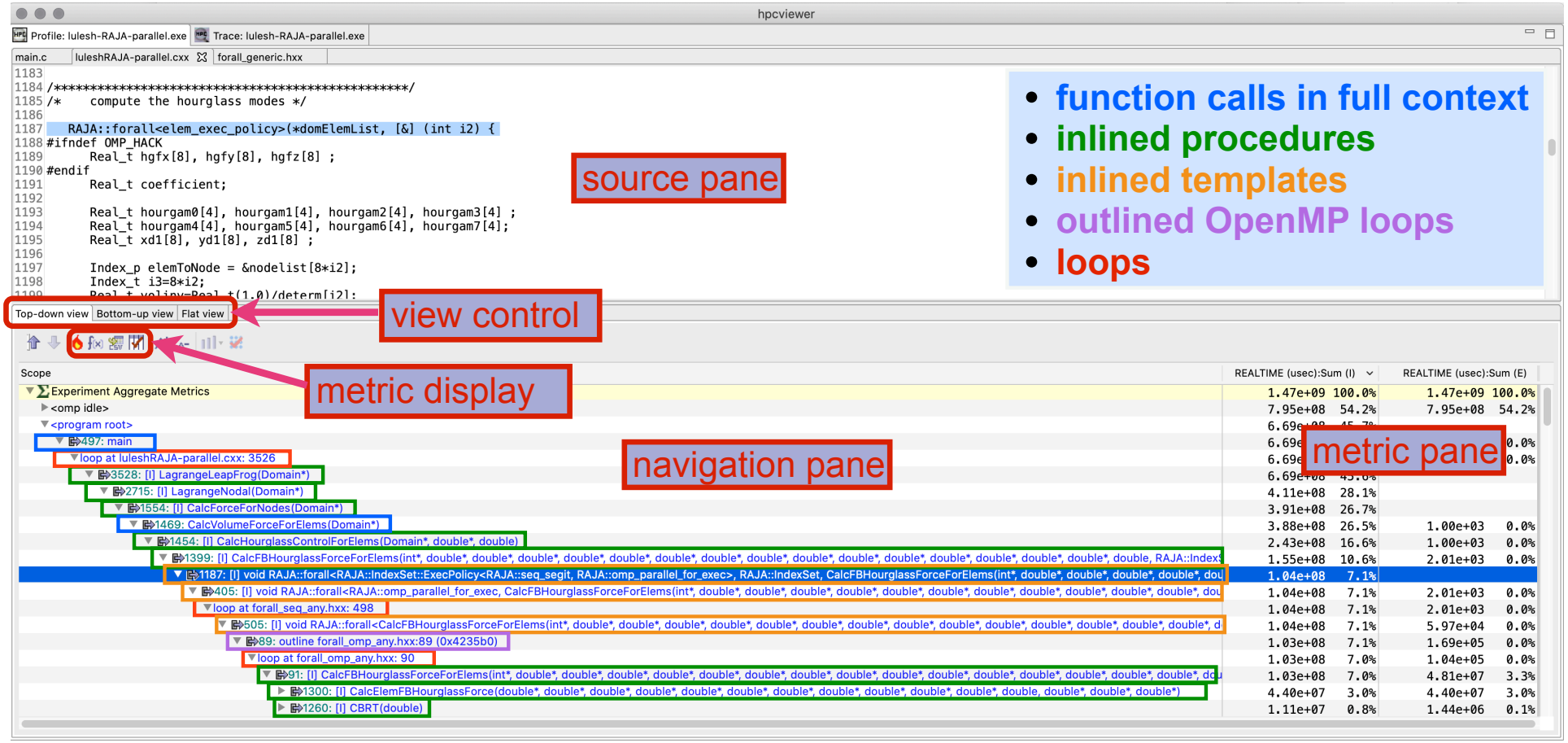
HPCToolkit's Workflow for GPU-accelerated Applications

Step 4:

- *hpcviewer* - interactively explore profile and traces for GPU-accelerated applications

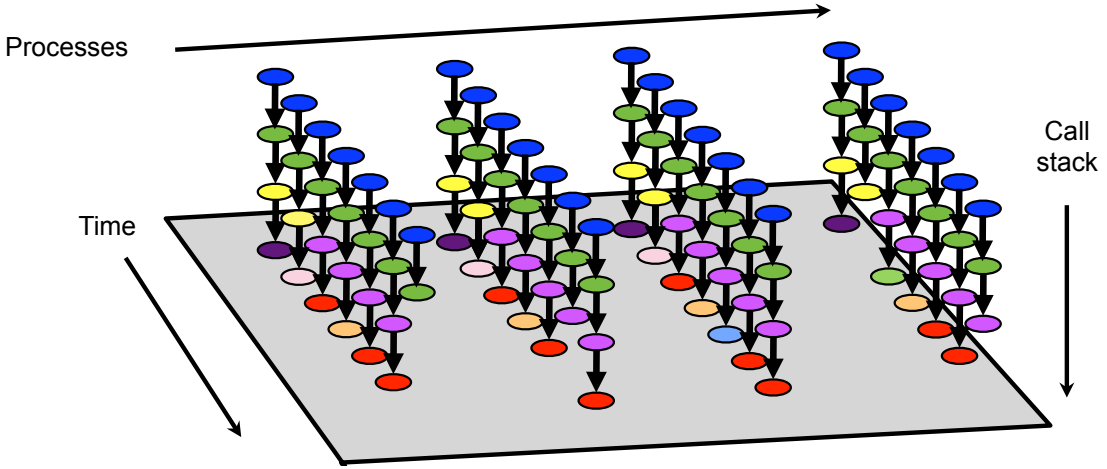


Code-centric Analysis with hpcviewer

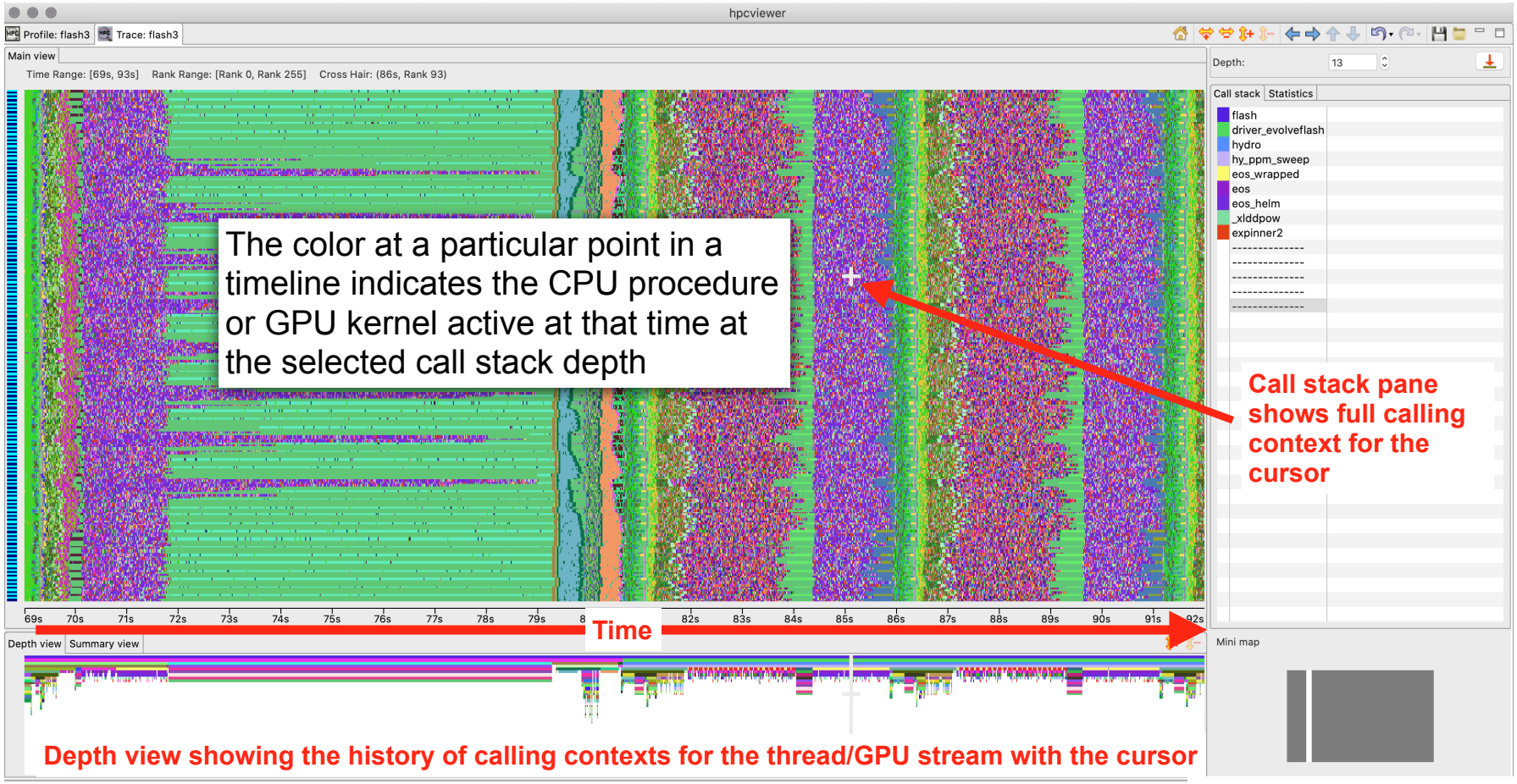


Understanding Temporal Behavior

- Profiling compresses out the temporal dimension
  - —Temporal patterns, e.g. serial sections and dynamic load imbalance are invisible in profiles
- What can we do? Trace call path samples
  - —N times per second, take a call path sample of each thread
  - —Organize the samples for each thread along a time line
  - —View how the execution evolves left to right
  - —What do we view? assign each procedure a color; view a depth slice of an execution



Time-centric Analysis with hpcviewer



A multi-level call stack based view of execution over time

Minimap indicates part of execution trace shown

Case Studies

- ExaWind
- GAMESS (OpenMP)
- Quicksilver (CUDA)
- PeleC (AMReX)
- LAMMPS (Kokkos) at exascale

ExaWind: Wakes from Three Turbines over Time

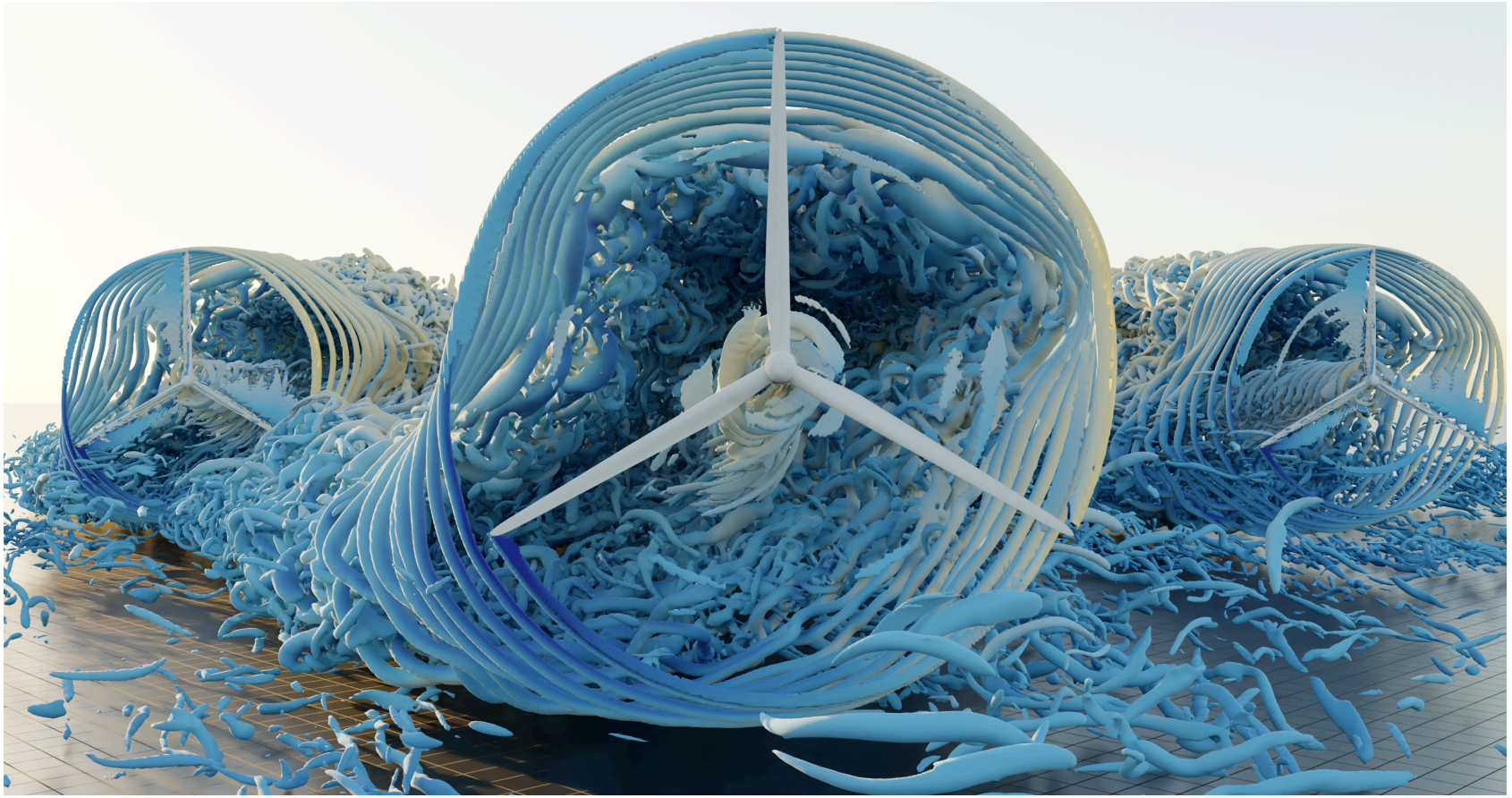


Figure credit: Jon Rood, NREL

ExaWind: Visualization of a Wind Farm Simulation

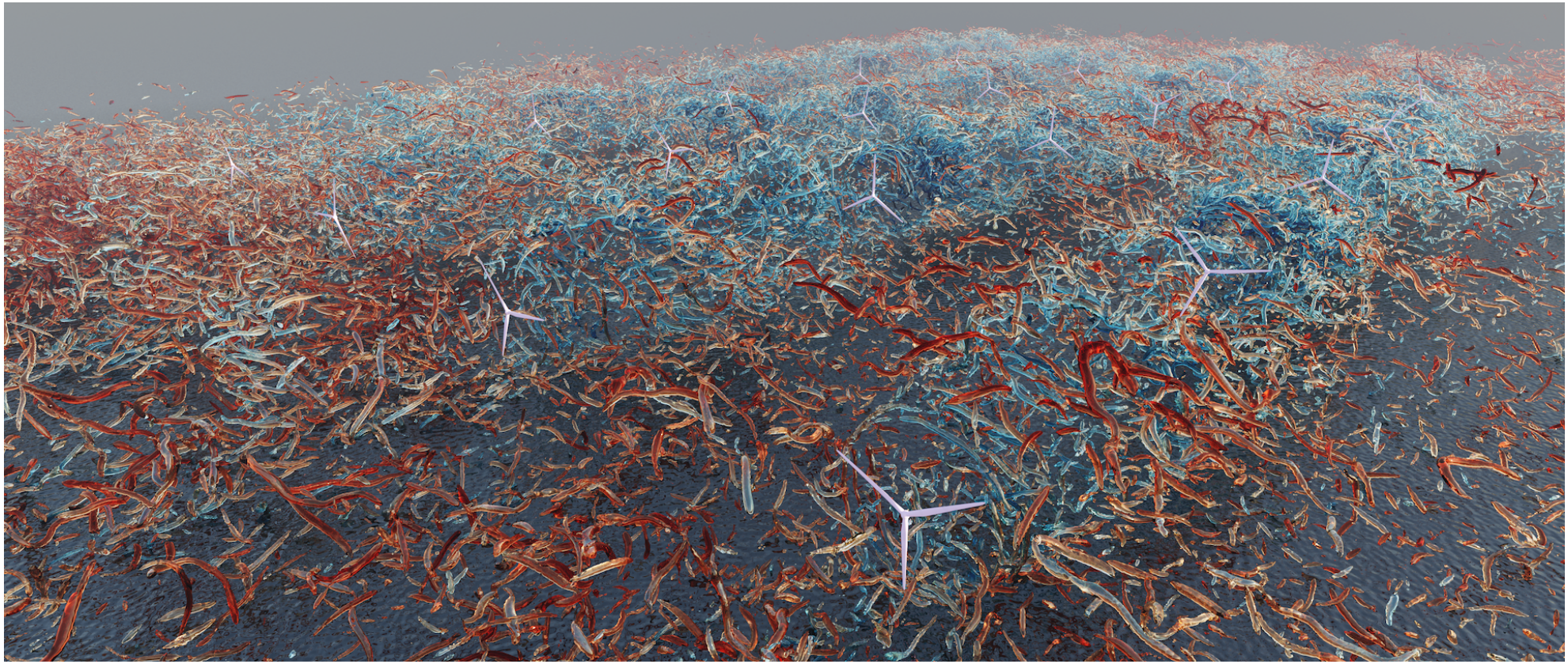


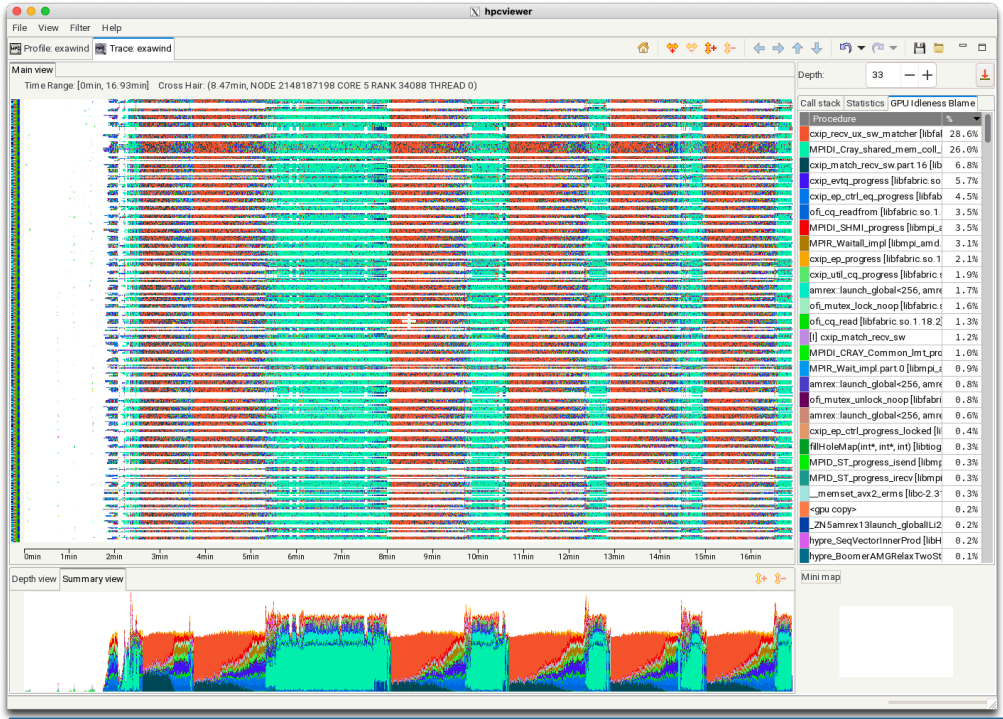
Figure credit: Jon Rood, NREL

ExaWind: Execution Traces on Frontier Collected with HPCToolkit

Traces on roughly ~70K MPI ranks for ~17minutes

Before: MPI waiting (bad), shown in red

After: MPI overhead negligible*



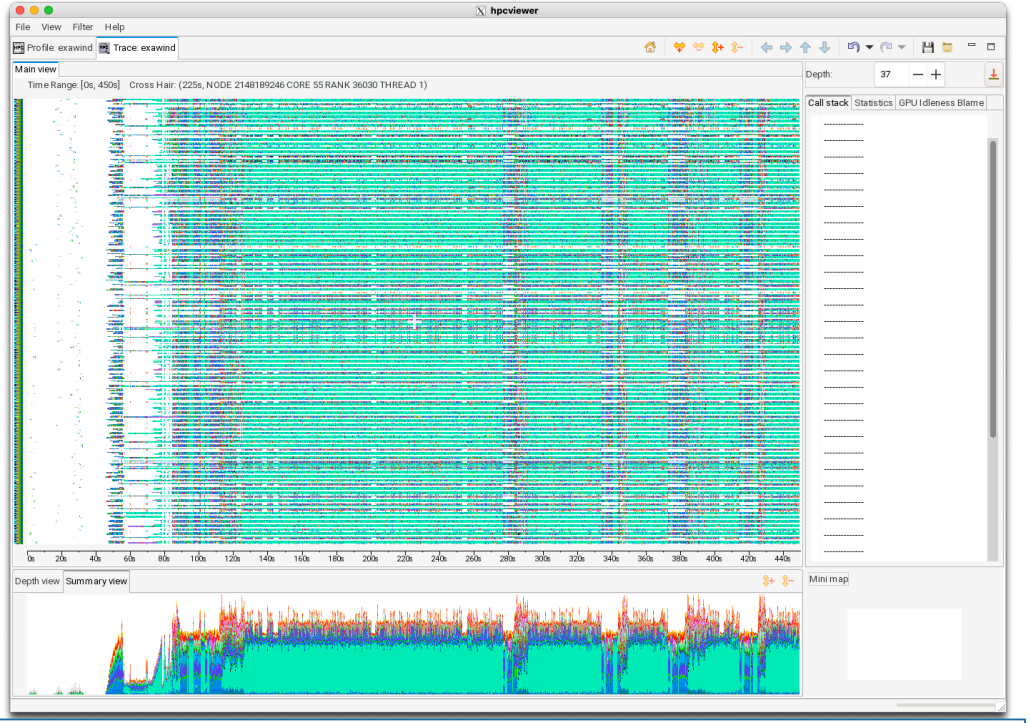


Figure credits: Jon Rood, NREL

*replaced non-blocking send/recv with ialltoallv

ExaWind Testimonials for HPCToolkit

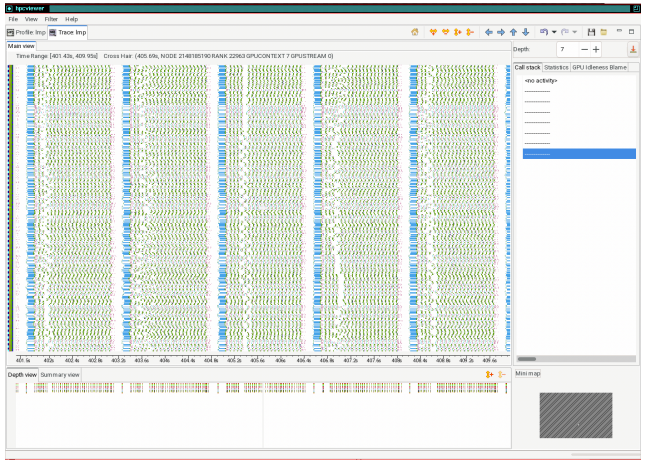
I just wanted to mention we've been using HPCToolkit a lot for our ExaWind application on Frontier, which is a hugely complicated code, and your profiler is one of the only ones we've found that really lets us easily instrument and then browse what our application is doing at runtime including GPUs. As an example, during a recent hackathon we had, we improved our large scale performance by 24x by understanding our code better with HPCToolkit and running it on 1000s of nodes while profiling. We also recently improved upon this by 10% for our total runtime.

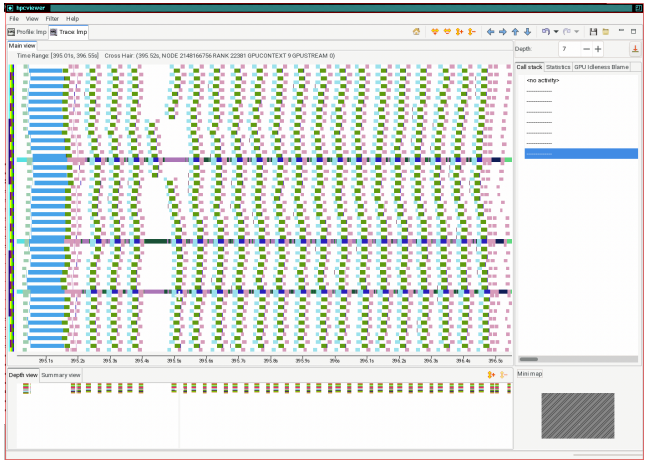
- Jon Rood NREL (5/31/2024)

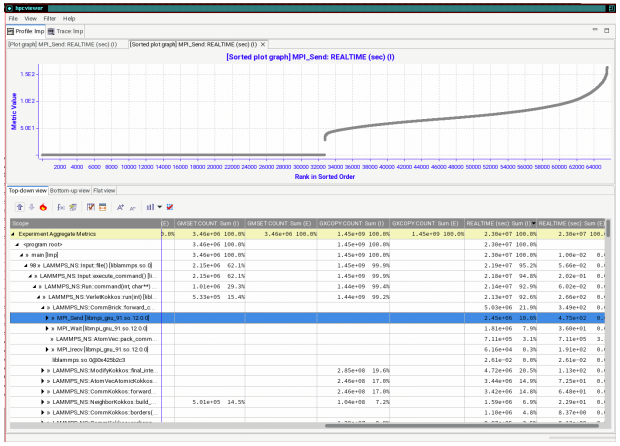
One big thing for us is that we can't overstate how complicated ExaWind is in general, and how complicated it is to build, so finding out that HPCToolkit could easily profile our entire application without a ton of instrumentation during the build process, and be able to profile it on a huge amount of Frontier with line numbers and visualizing the trace was really amazing to us.

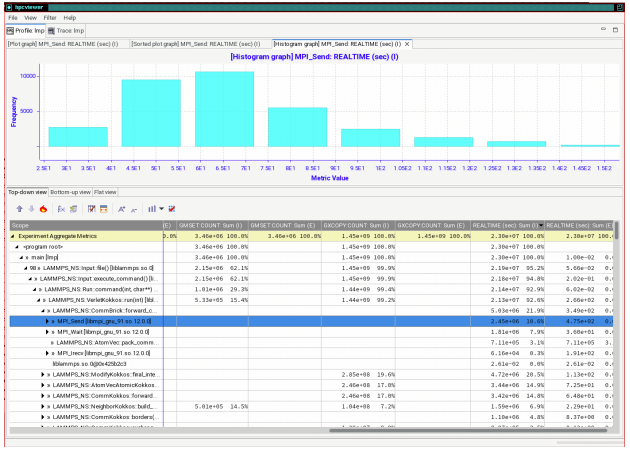
- Jon Rood NREL (6/3/2024)

LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds

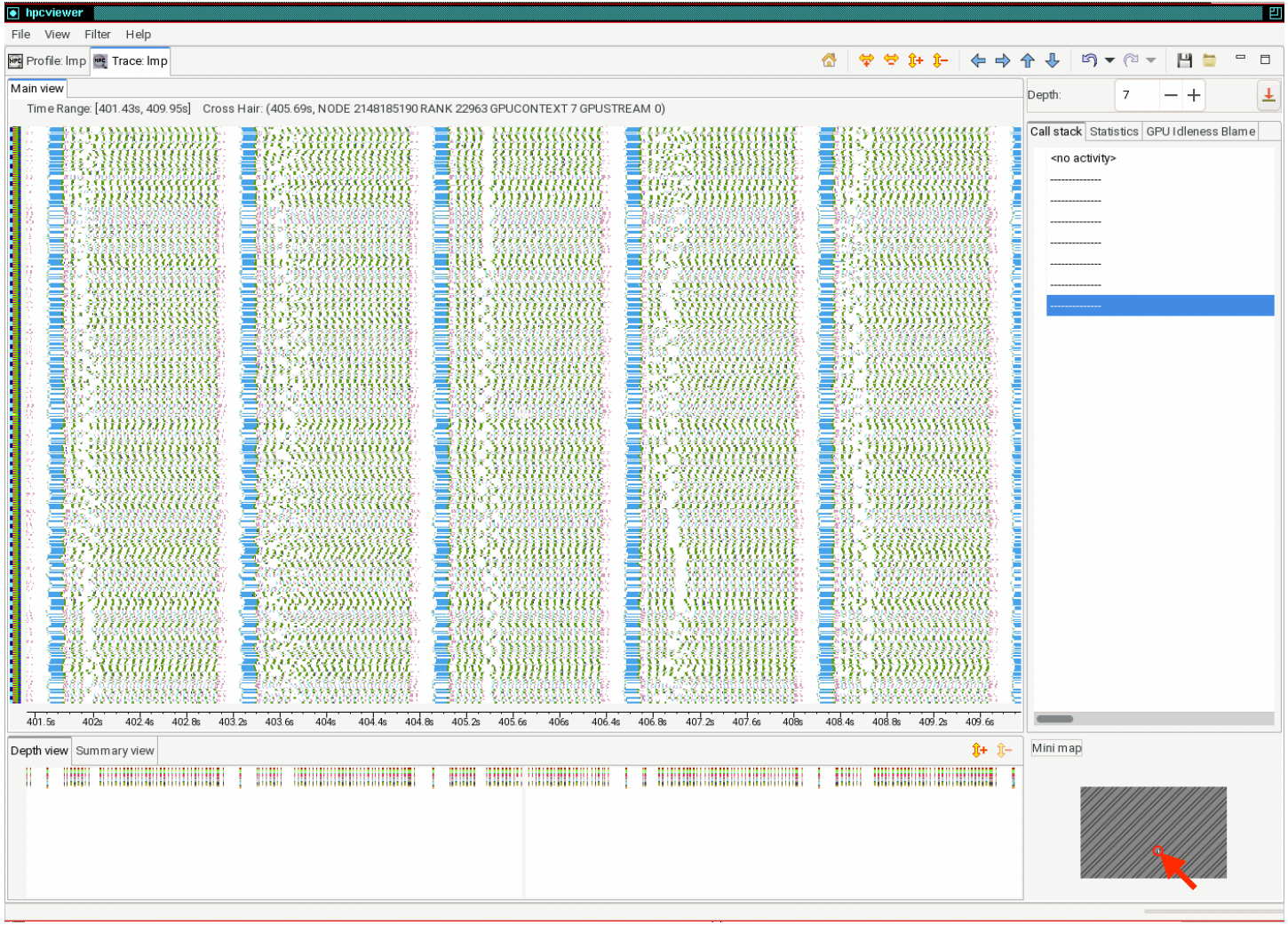




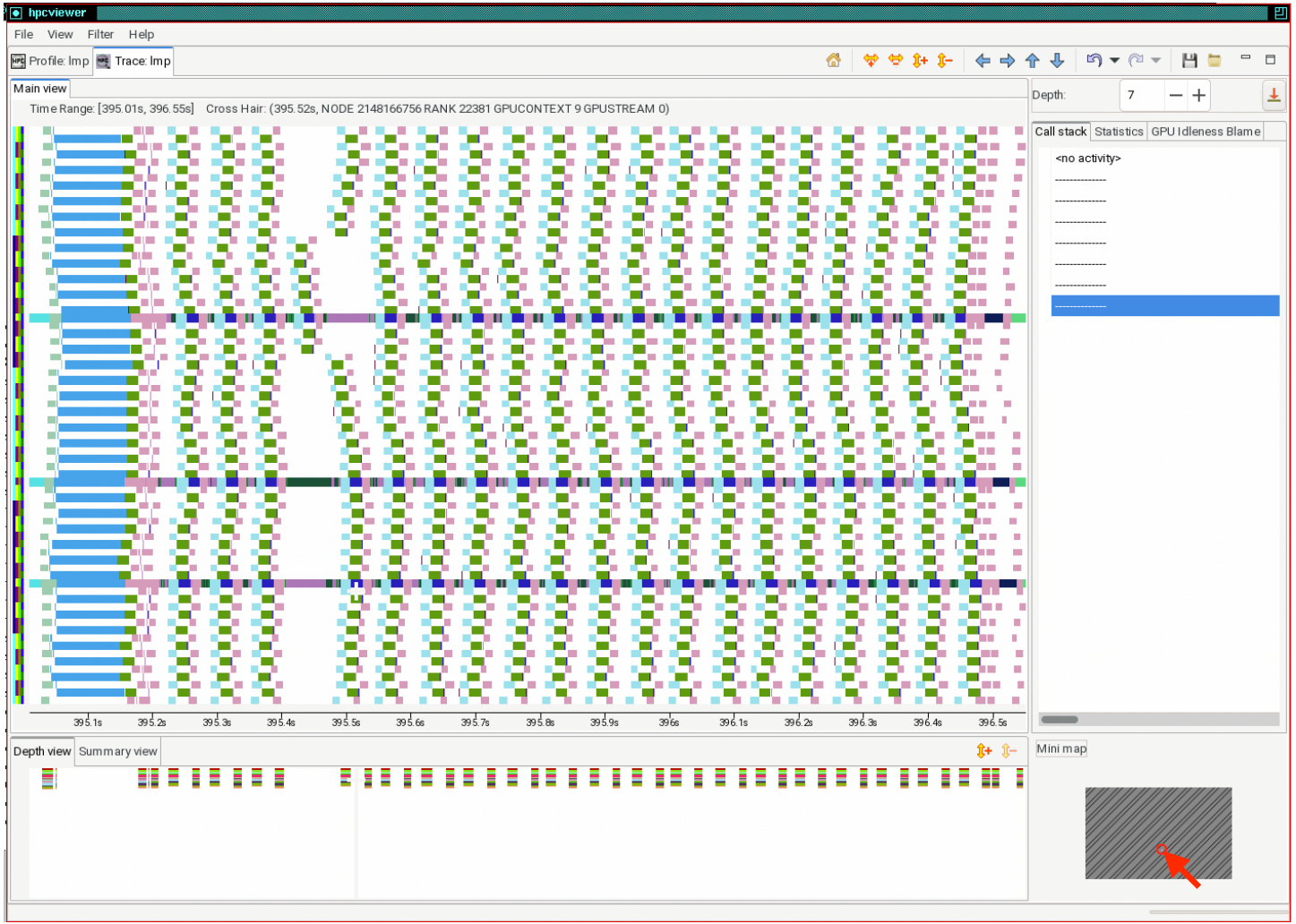




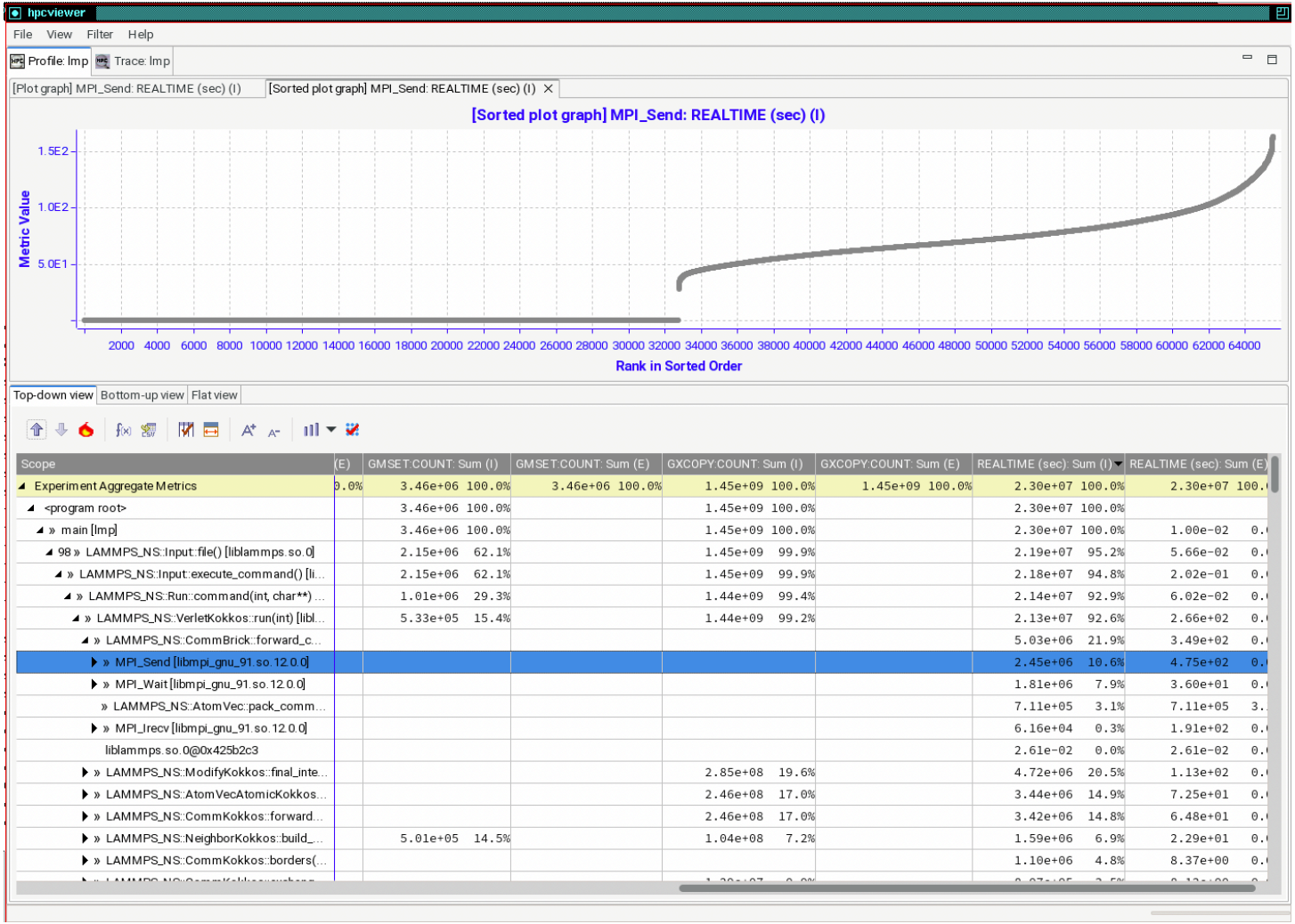
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds



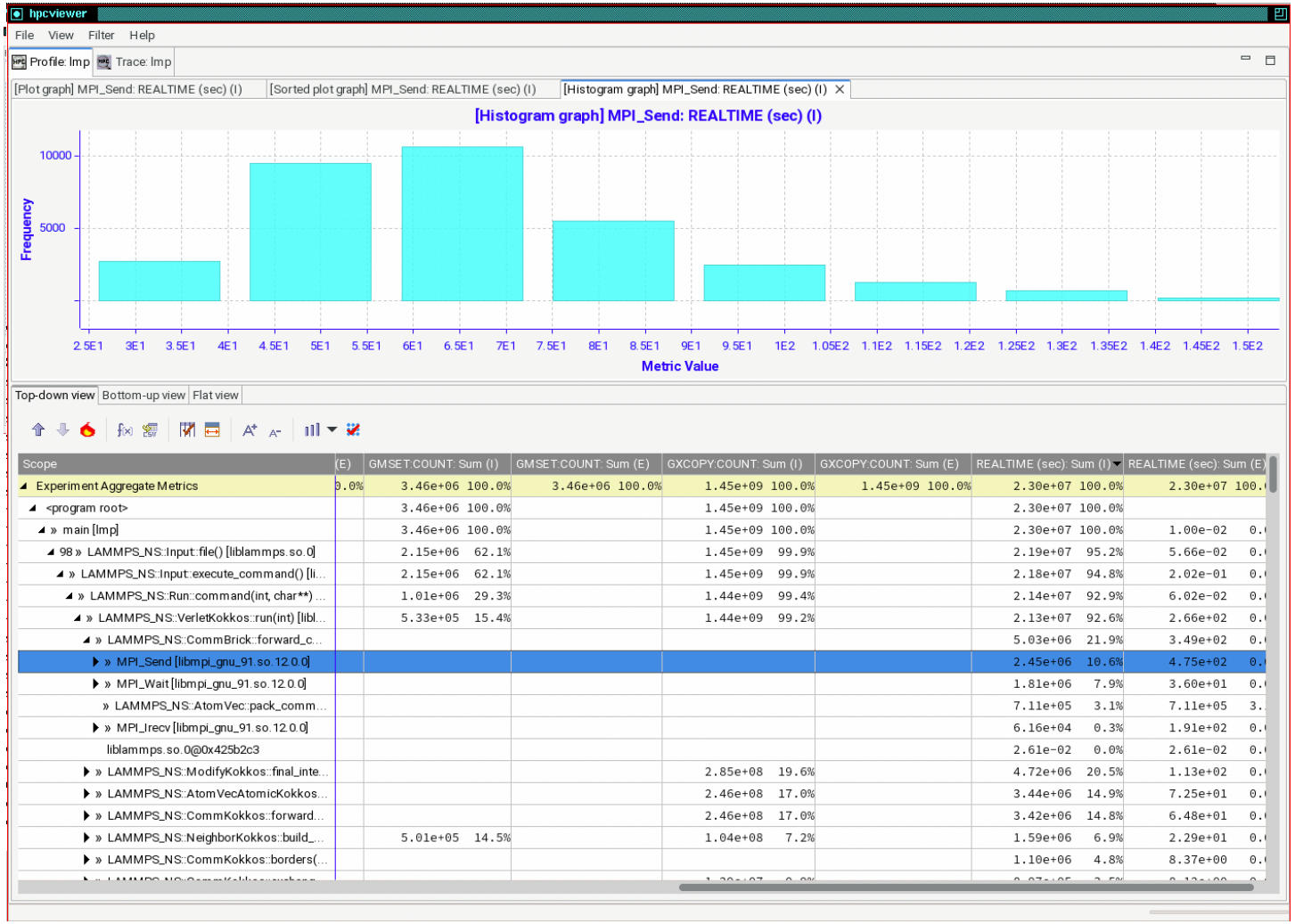
LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds



LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds

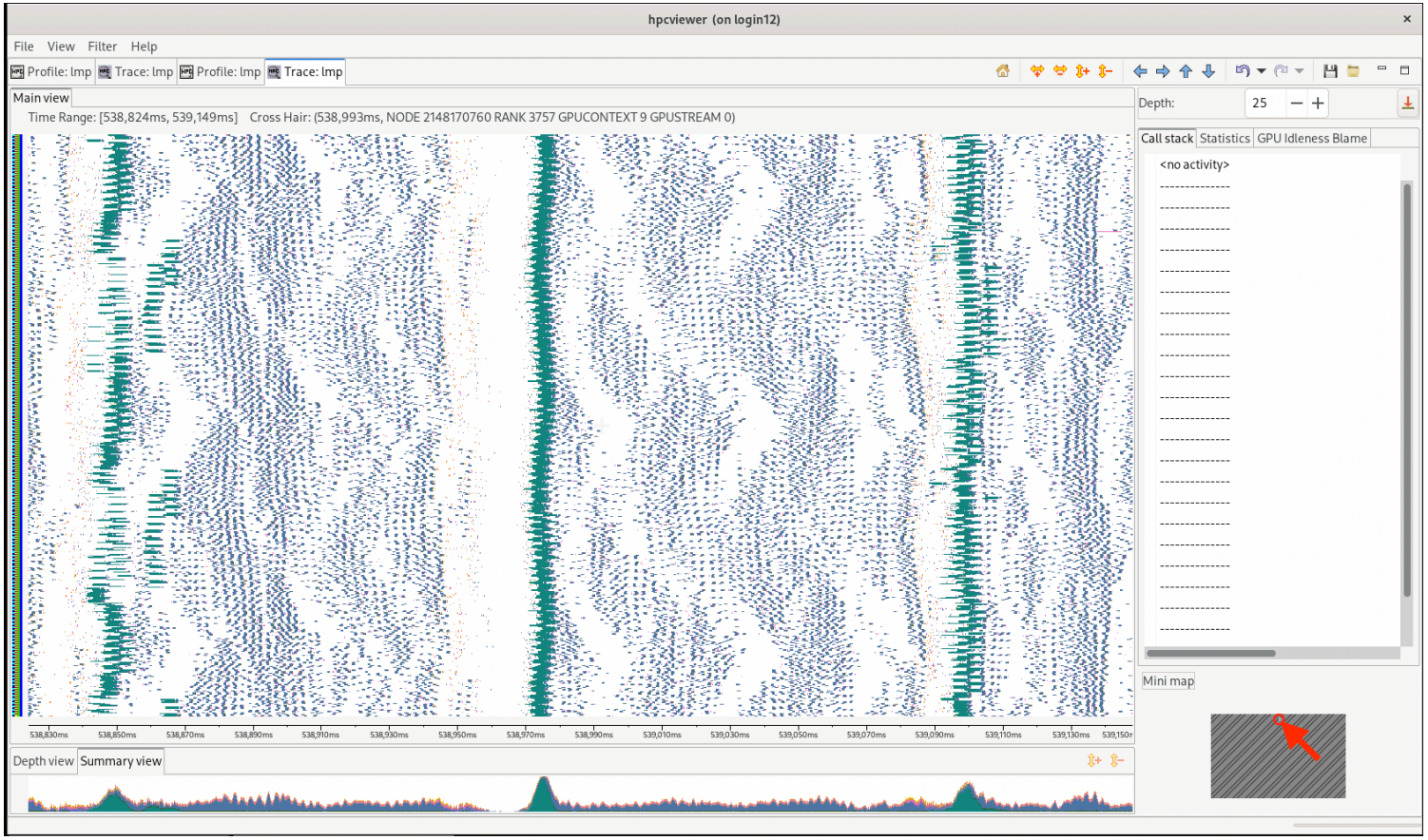


LAMMPS on Frontier: Executions with Kernel Duration of Milliseconds



LAMMPS on Frontier: 8K nodes, 64K MPI ranks + 64K GPU tiles

Kernel duration of microseconds



Case Study: GAMESS

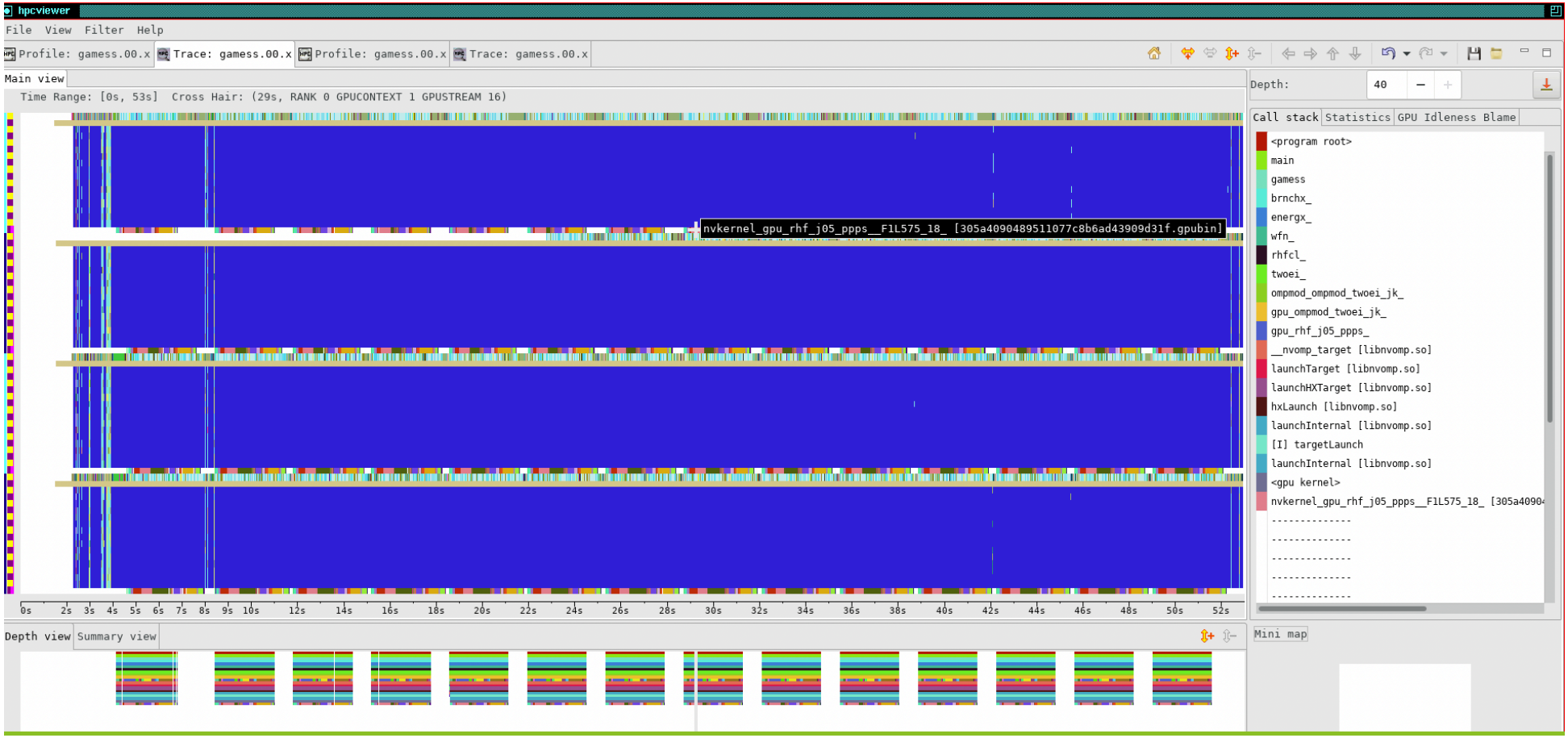
- General Atomic and Molecular Electronic Structure System (GAMESS)
  —general *ab initio* quantum chemistry package
- Calculates the energies, structures, and properties of a wide range of chemical systems

- Experiments
  - GPU-accelerated nodes at a prior Perlmutter hackathon
    - Single node with 4 GPUs
    - Five nodes with 20 GPUs

Perlmutter node at a glance

AMD Milan CPU
4 NVIDIA A100 GPUs
256 GB memory

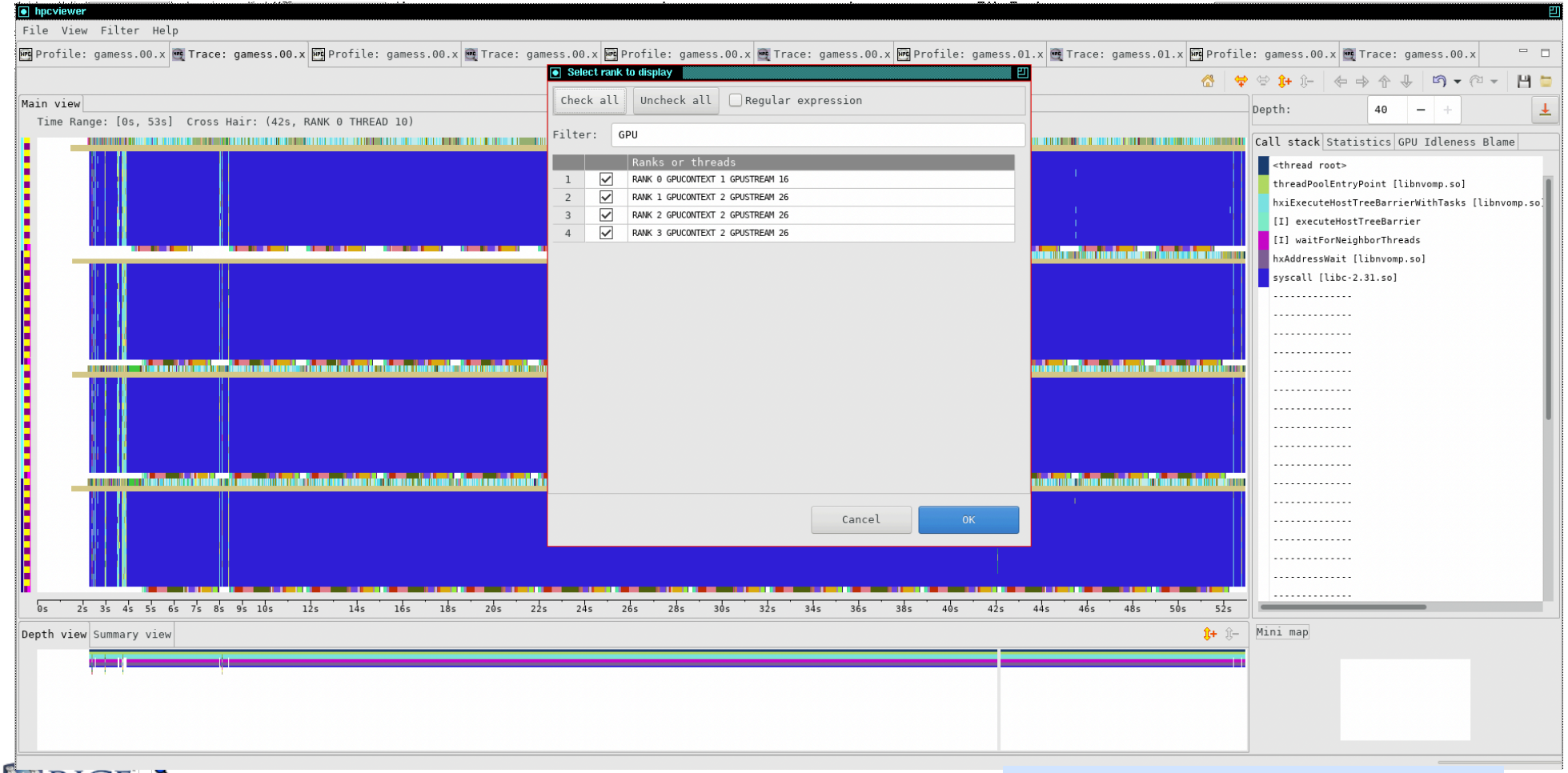
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter



GAMESS original

All CPU threads and GPU streams

Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter



GAMESS original

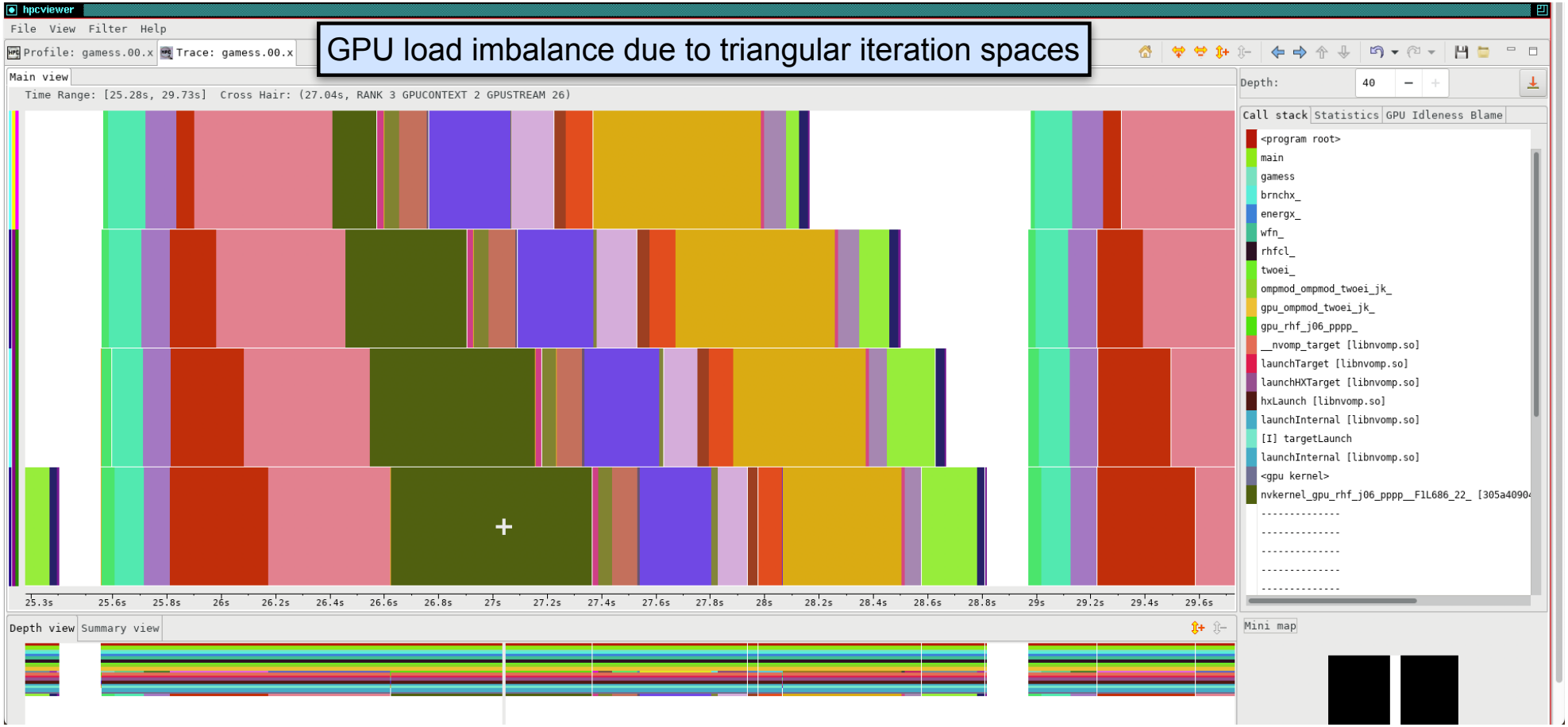
All CPU threads and GPU streams

Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter

GAMESS original

All GPU streams, whole execution

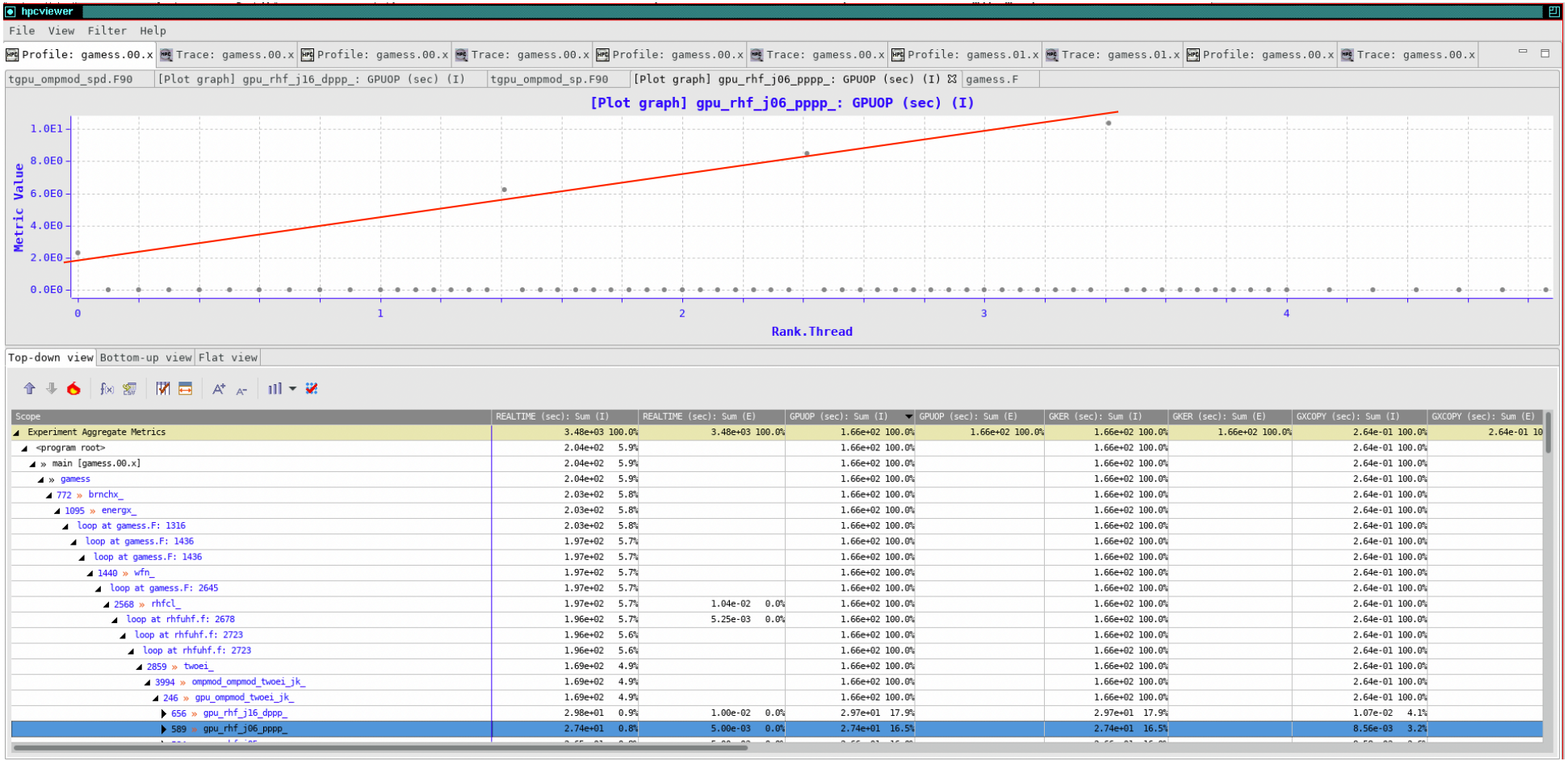
Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter



GAMESS original

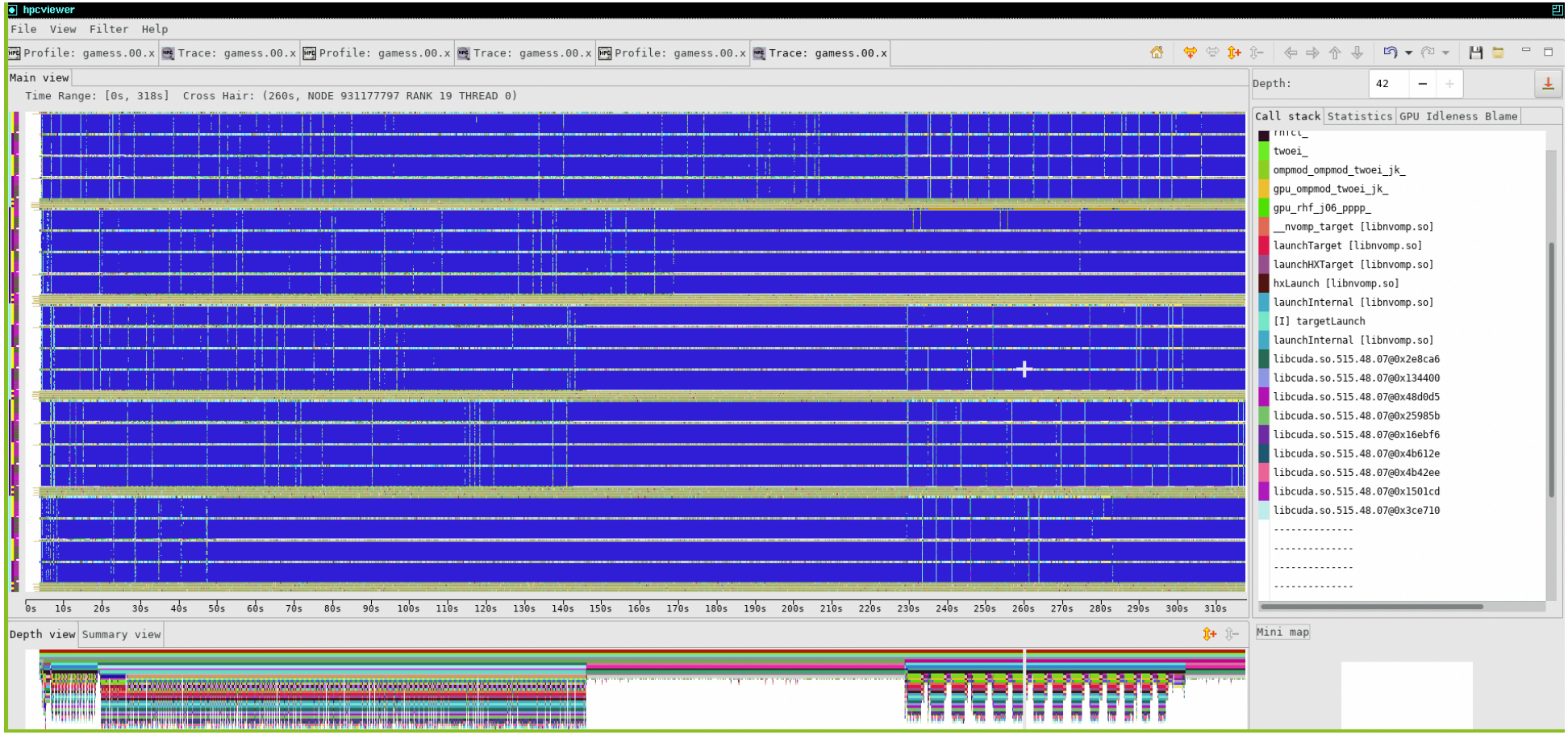
GPU streams: 1 iteration

Time-centric Analysis: GAMESS 4 ranks, 4 GPUs on Perlmutter



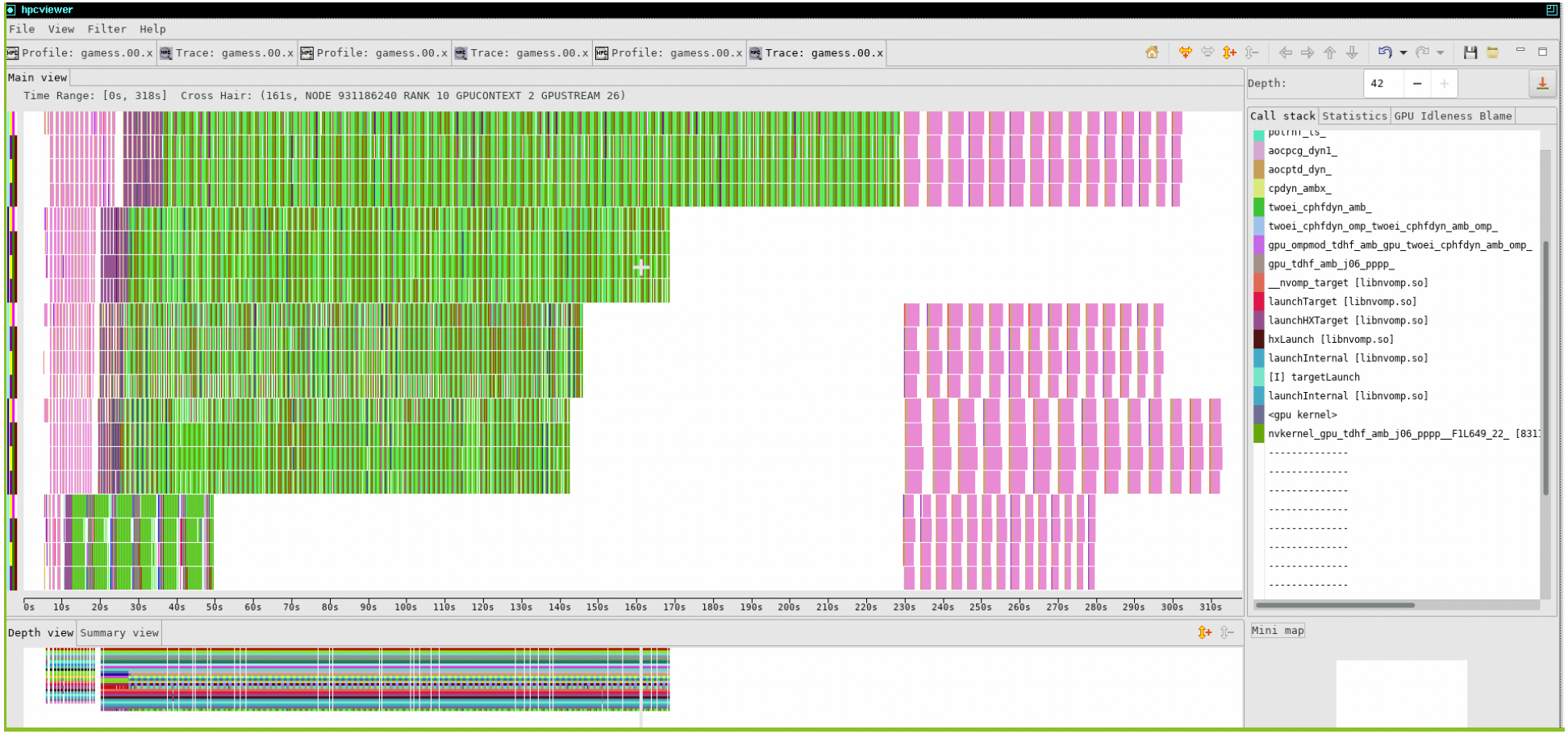
GAMESS original

Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



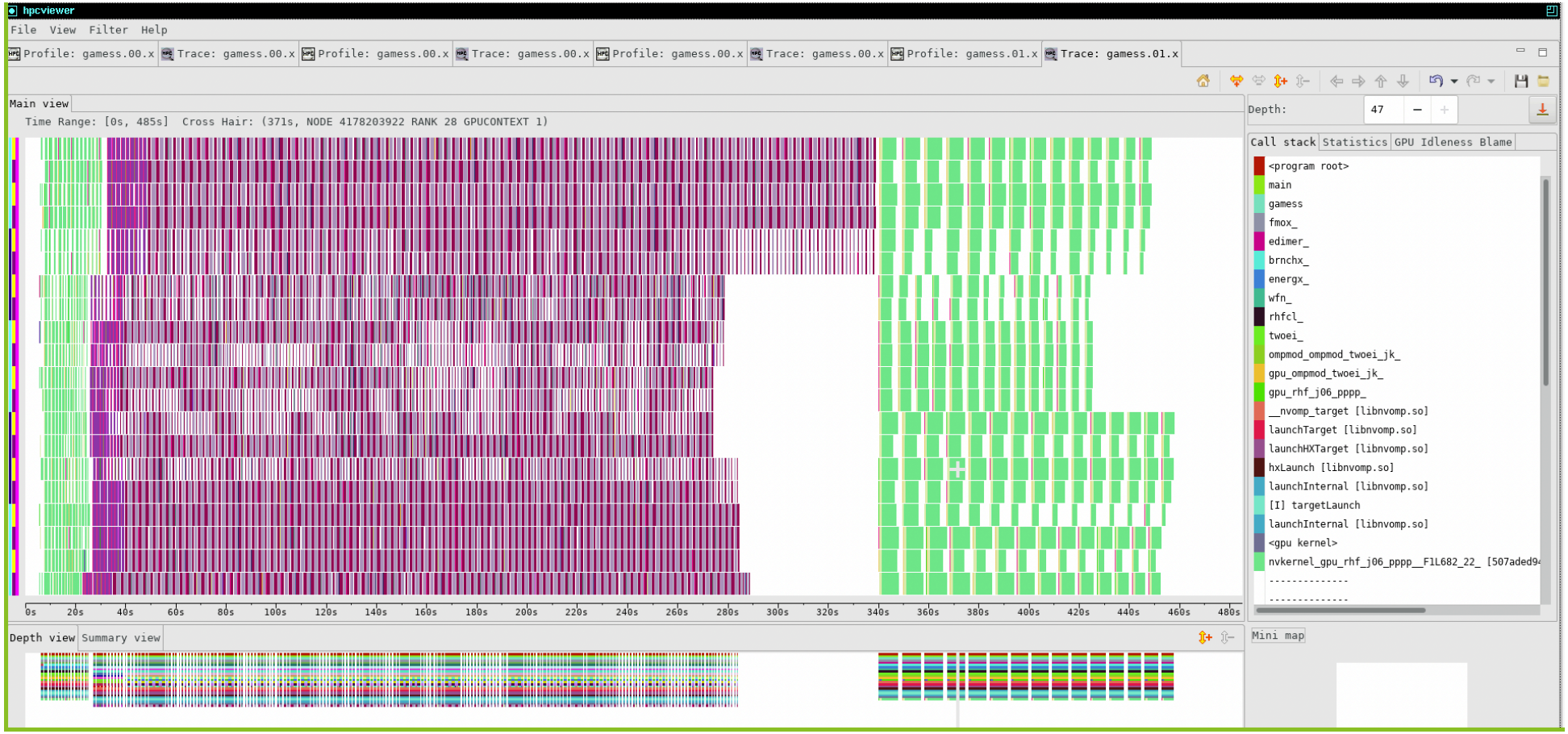
GAMESS improved CPU Threads and GPU Streams

Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



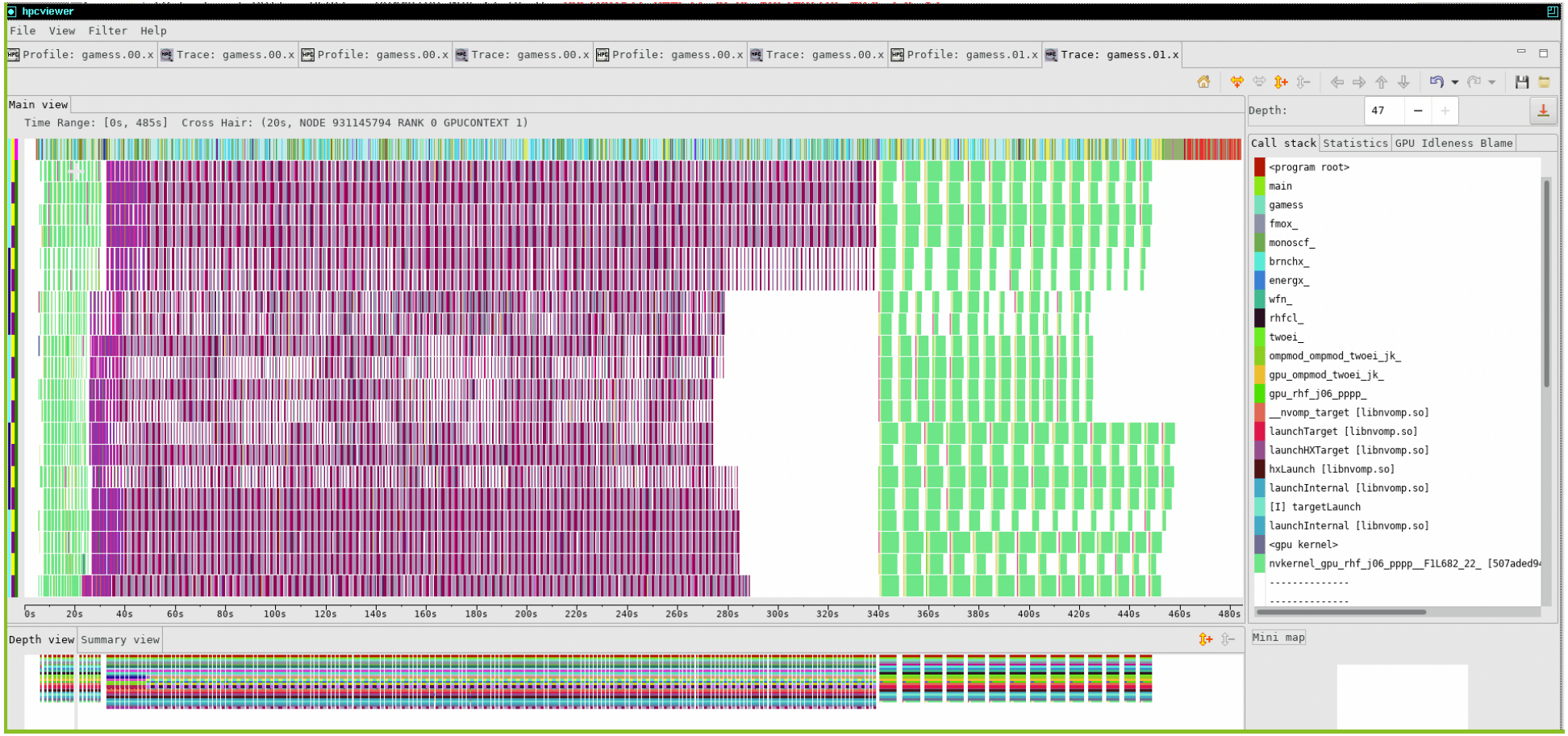
GAMESS improved

Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



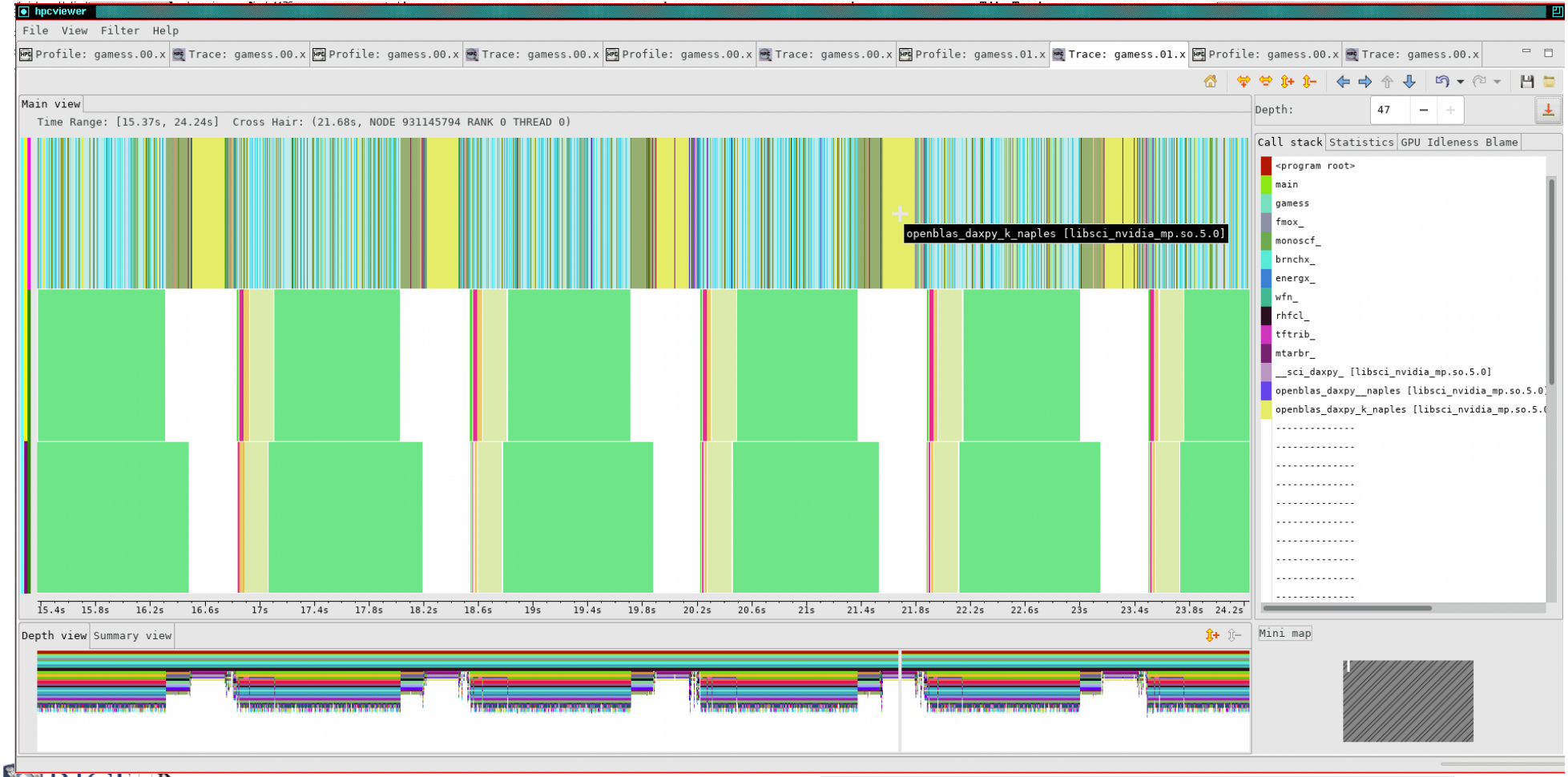
GAMESS improved with better manual distribution of work in input

Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



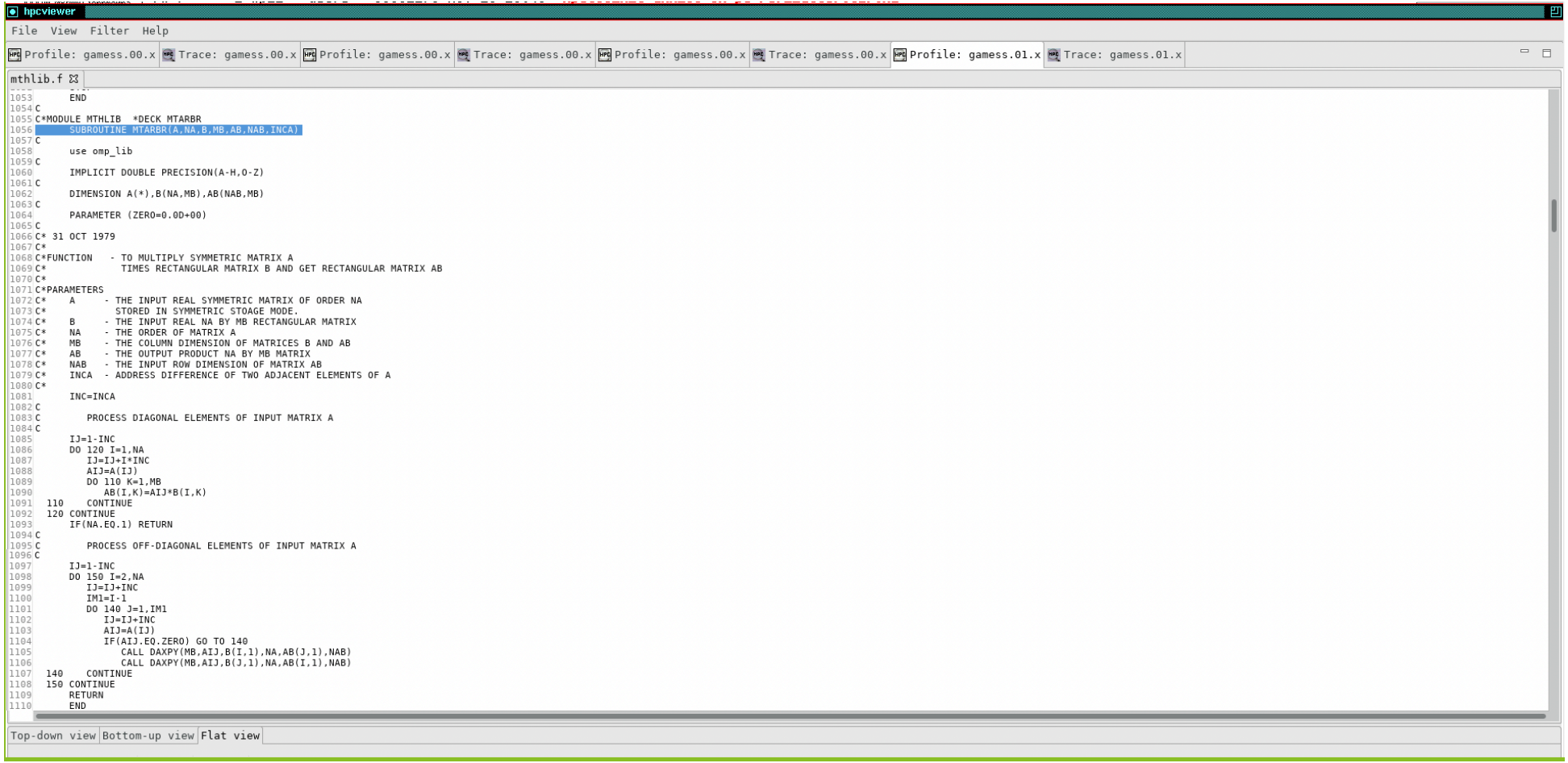
GAMESS improved adding Rank 0 Thread 0 to GPU streams

Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter

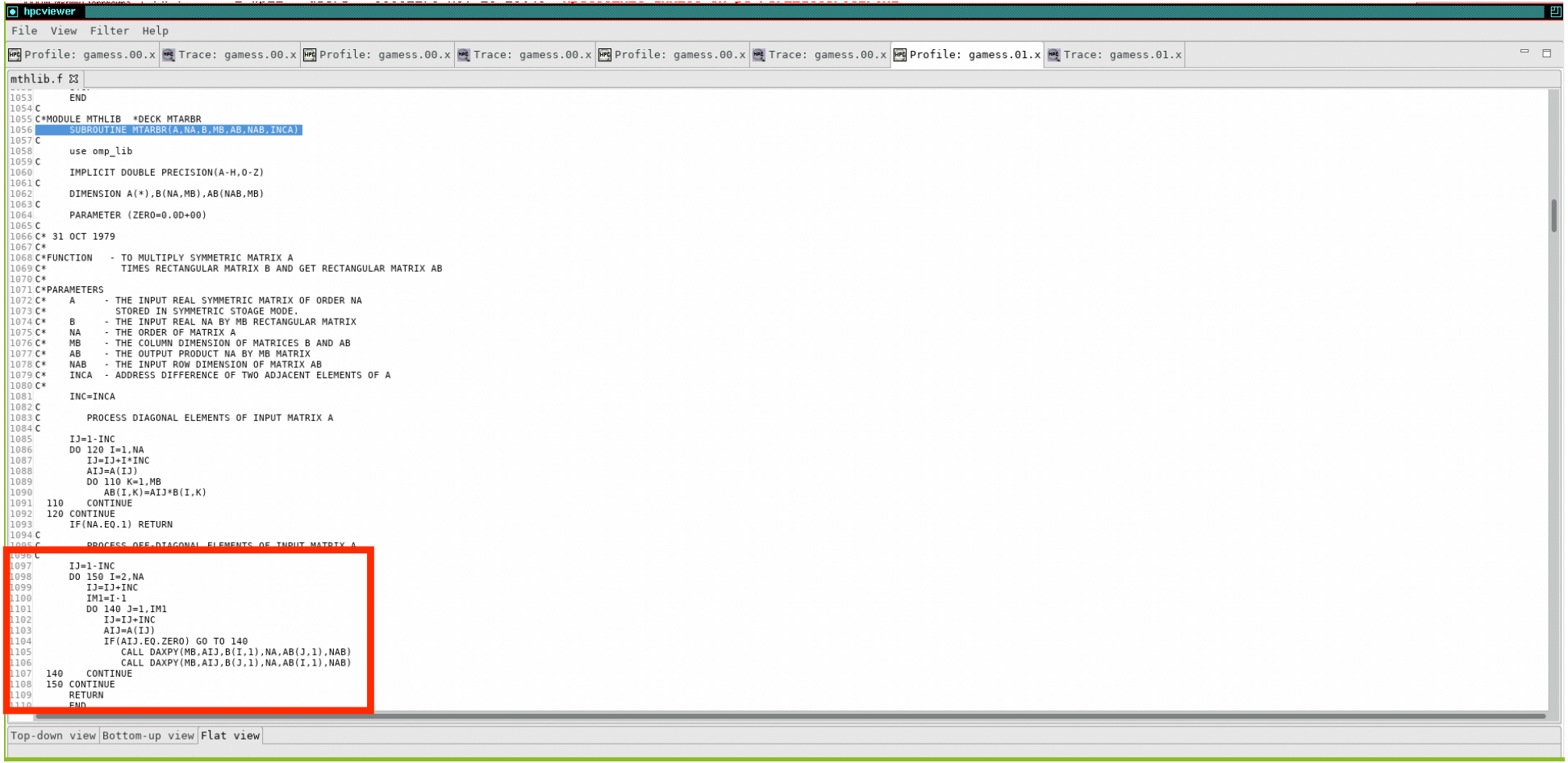


1 CPU Stream, 2 GPU Streams: 6 Iterations

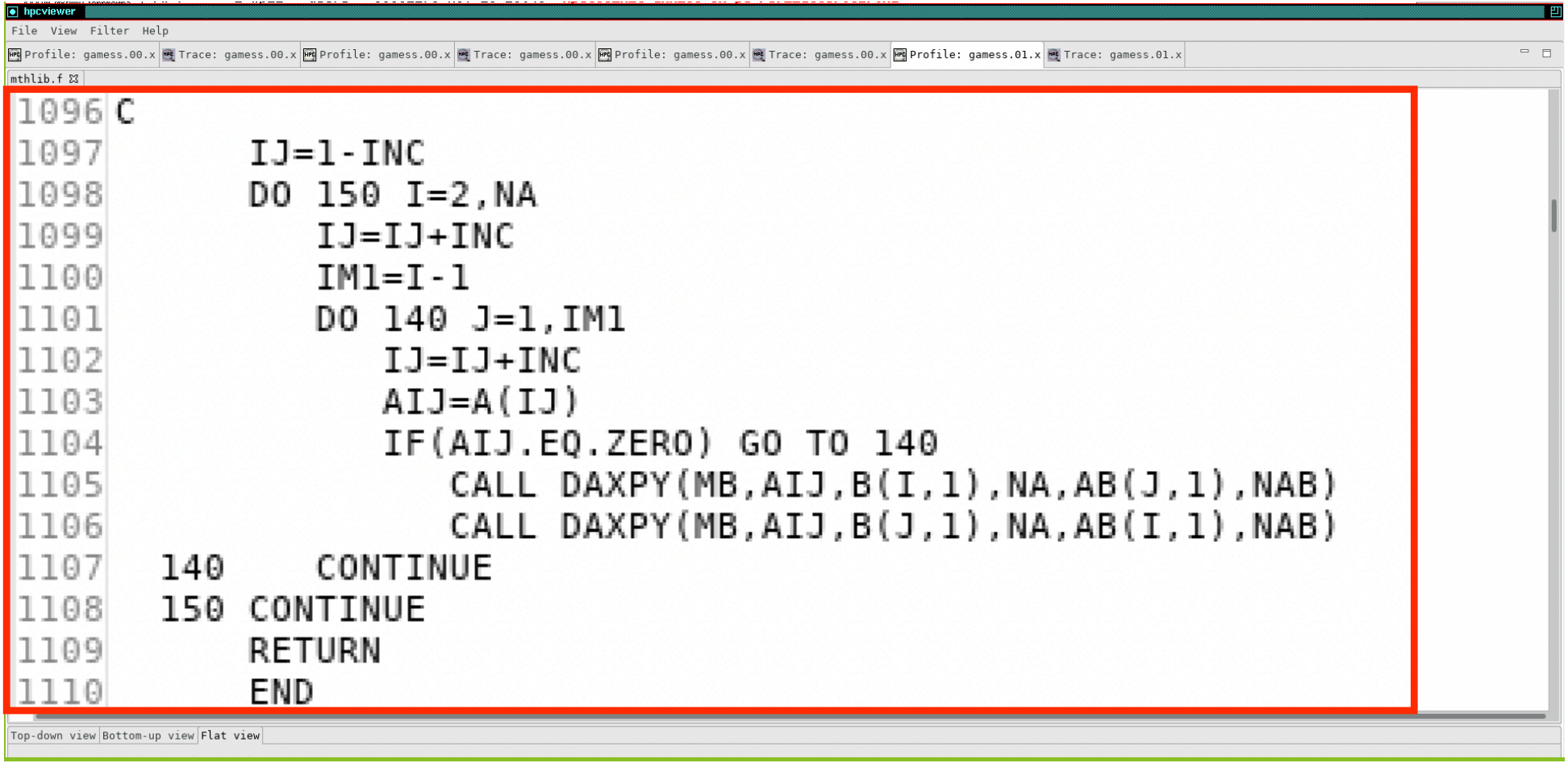
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



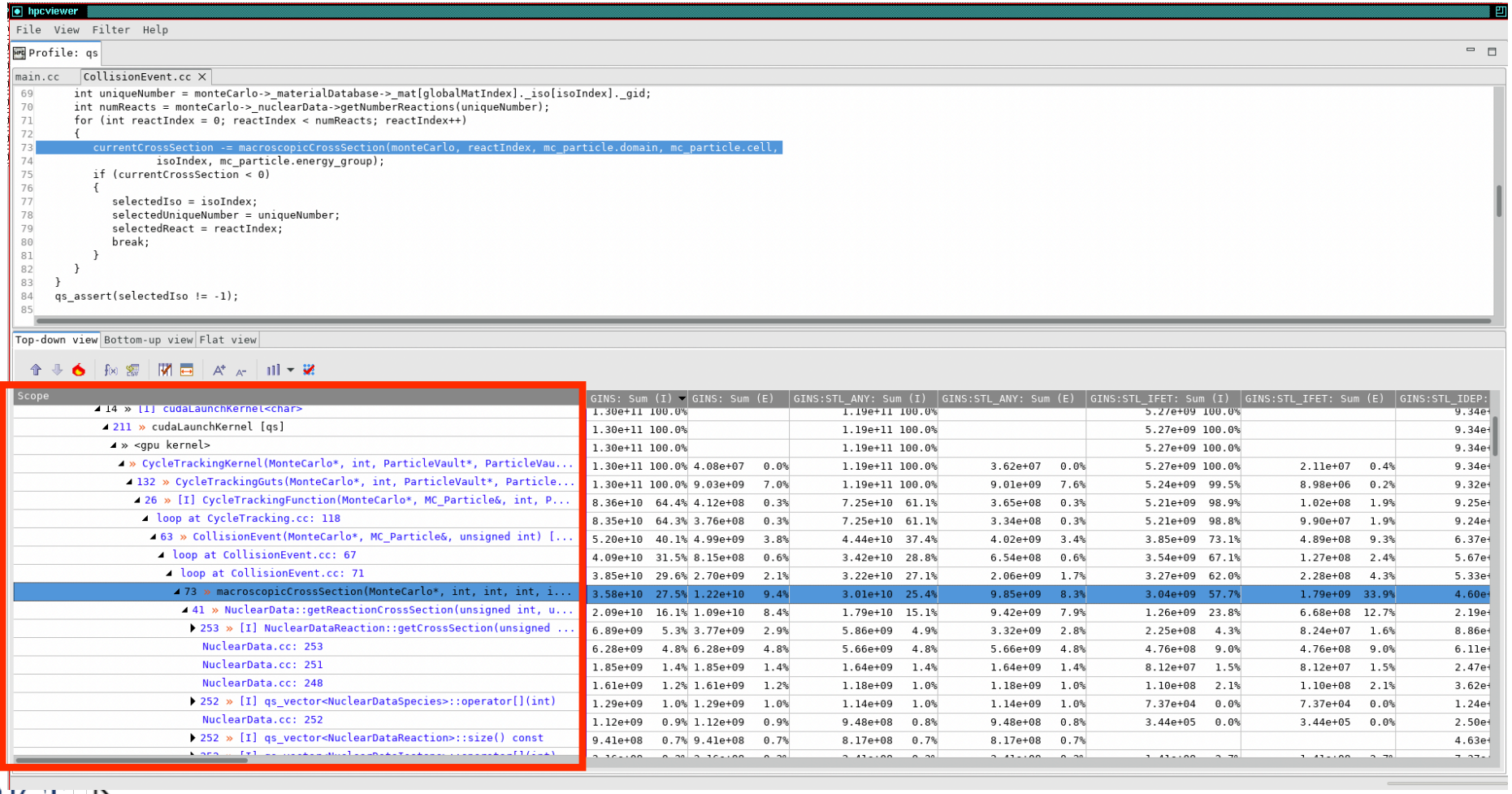
Time-centric Analysis: GAMESS 5 nodes, 40 ranks, 20 GPUs on Perlmutter



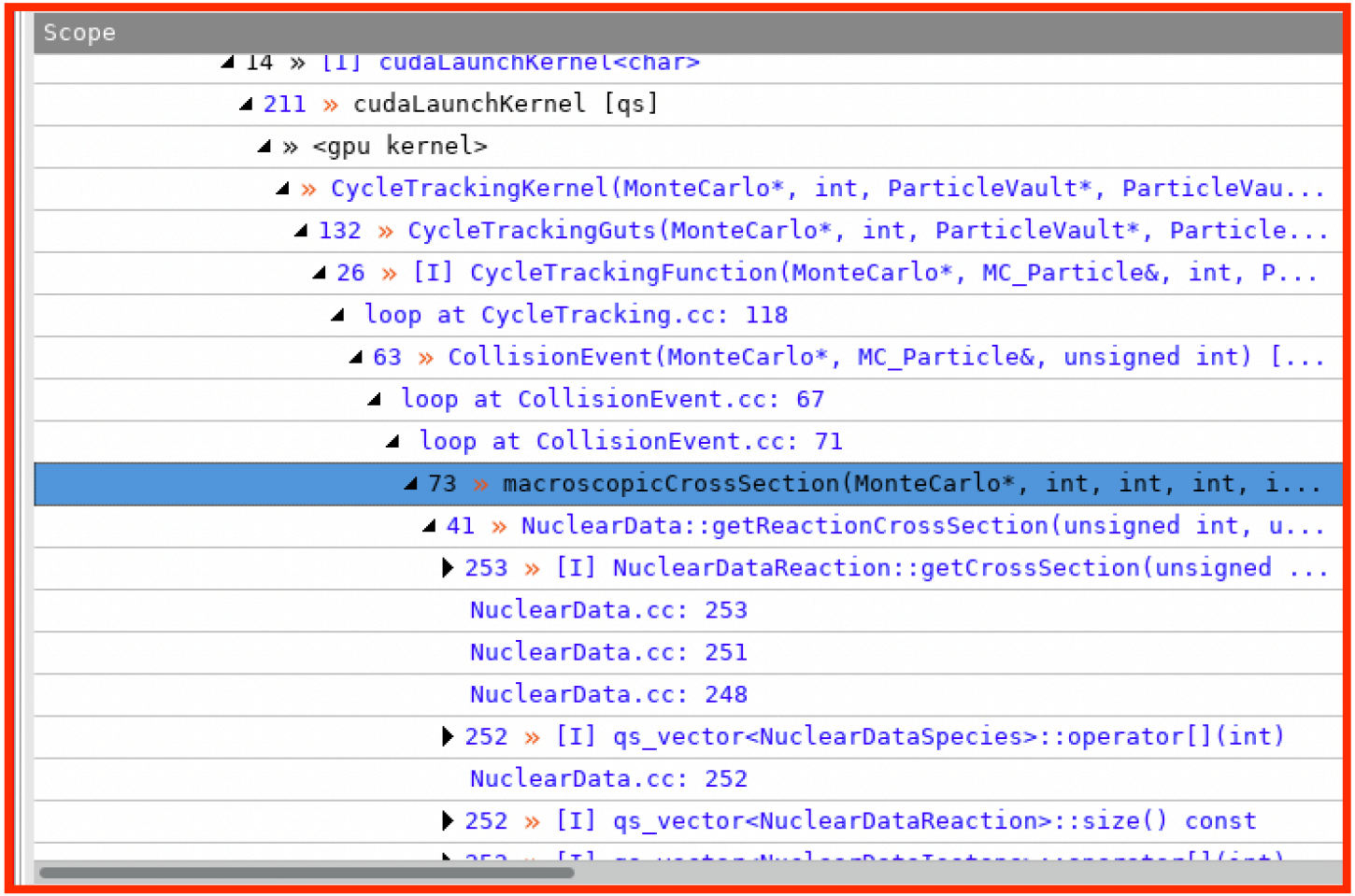
Case Study: Quicksilver

- Proxy application that represents some elements of LLNL's Mercury workload
- Solves a simplified dynamic Monte Carlo particle transport problem
  - Attempts to replicate memory access patterns, communication patterns, and branching or divergence of Mercury for problems using multigroup cross sections
- Parallelization: MPI, OpenMP, and CUDA
- Performance Issues
  - load imbalance (for canned example)
  - latency bound table look-ups
  - a highly branchy/divergent code path
  - poor vectorization potential

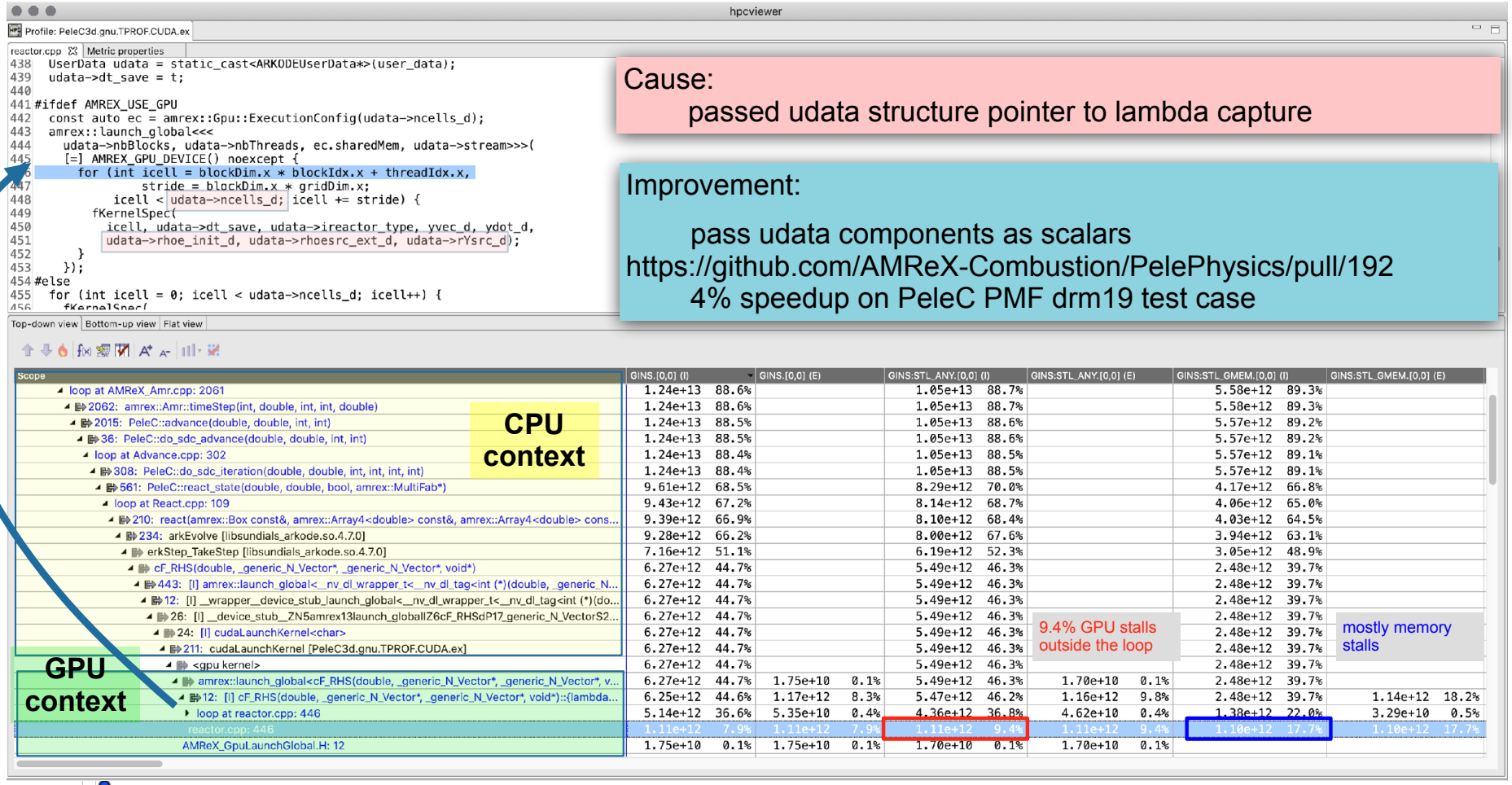
Quicksilver: Detailed analysis within a Kernel using PC Sampling



Quicksilver: Detailed analysis within a Kernel using PC Sampling



Analysis of PeleC using PC Sampling on an NVIDIA GPU



Scope	GINS.[0,0] (I)		GINS.[0,0] (E)		GINS:STL_ANY.[0,0] (I)		GINS:STL_ANY.[0,0] (E)		GINS:STL_GMEM.[0,0] (I)		GINS:STL_GMEM.[0,0] (E)	
loop at AMReX_Amr.cpp: 2061	1.24e+13	88.6%			1.05e+13	88.7%			5.58e+12	89.3%		
2062: amrex::Amr::timeStep(int, double, int, int, double)	1.24e+13	88.6%			1.05e+13	88.7%			5.58e+12	89.3%		
2015: PeleC::advance(double, double, int, int)	1.24e+13	88.5%			1.05e+13	88.6%			5.57e+12	89.2%		
36: PeleC::do_sdc_advance(double, double, int, int)	1.24e+13	88.5%			1.05e+13	88.6%			5.57e+12	89.2%		
loop at Advance.cpp: 302	1.24e+13	88.4%			1.05e+13	88.5%			5.57e+12	89.1%		
308: PeleC::do_sdc_iteration(double, double, int, int, int)	1.24e+13	88.4%			1.05e+13	88.5%			5.57e+12	89.1%		
561: PeleC::react_state(double, double, bool, amrex::MultiFab*)	9.61e+12	68.5%			8.29e+12	70.0%			4.17e+12	66.8%		
loop at React.cpp: 109	9.43e+12	67.2%			8.14e+12	68.7%			4.06e+12	65.0%		
210: react(amrex::Box const&, amrex::Array4<double> const&, amrex::Array4<double> cons...	9.39e+12	66.9%			8.10e+12	68.4%			4.03e+12	64.5%		
234: arkEvolve [libsundials_arkode.so.4.7.0]	9.28e+12	66.2%			8.00e+12	67.6%			3.94e+12	63.1%		
erkStep_TakeStep [libsundials_arkode.so.4.7.0]	7.16e+12	51.1%			6.19e+12	52.3%			3.05e+12	48.9%		
cF_RHS(double, _generic_N_Vector*, _generic_N_Vector*, void*)	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
443: [I] amrex::launch_global<__nv_dl_wrapper_t<__nv_dl_tag<int (*)(double, _generic_N...	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
12: [I] __wrapper__device_stub_launch_global<__nv_dl_wrapper_t<__nv_dl_tag<int (*)(do...	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
26: [I] __device_stub__ZN5amrex13launch_globalIZ6cF_RHSdP17_generic_N_VectorS2...	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
24: [I] cudaLaunchKernel<char>	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
211: cudaLaunchKernel [PeleC3d.gnu.TPROF.CUDA.ex]	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
<gpu kernel>	6.27e+12	44.7%			5.49e+12	46.3%			2.48e+12	39.7%		
amrex::launch_global<cF_RHS(double, _generic_N_Vector*, _generic_N_Vector*, v...	6.27e+12	44.7%	1.75e+10	0.1%	5.49e+12	46.3%	1.70e+10	0.1%	2.48e+12	39.7%		
12: [I] cF_RHS(double, _generic_N_Vector*, _generic_N_Vector*, void*)::{lambda...	6.25e+12	44.6%	1.17e+12	8.3%	5.47e+12	46.2%	1.16e+12	9.8%	2.48e+12	39.7%	1.14e+12	18.2%
loop at reactor.cpp: 446	5.14e+12	36.6%	5.35e+10	0.4%	4.36e+12	36.8%	4.62e+10	0.4%	1.38e+12	22.0%	3.29e+10	0.5%
reactor.cpp: 446	1.11e+12	7.9%	1.11e+12	7.9%	1.11e+12	9.4%	1.11e+12	9.4%	1.10e+12	17.7%	1.10e+12	17.7%
AMReX_GpuLaunchGlobal.H: 12	1.75e+10	0.1%	1.75e+10	0.1%	1.70e+10	0.1%	1.70e+10	0.1%				

Key Metrics for GPU Kernels

- GPUOP: GPU operation time (kernel launch, copies, etc.)
- GXCOPY:* GPU copies of various kinds
- GKER: GPU kernel time
- GKER:FGP_ACT: fine grain parallelism actual (active warps per SM)
- GKER:FGP_MAX: maximum possible fine-grain parallelism (max warps per SM)
- GKER:BLK_THR: threads per block
- GKER:BLK_SM: block shared memory
- GKER:OCC_THR: theoretical thread occupancy

Metrics for GPU Kernels with PC Samples

- GINS: GPU instructions
- GINS:STL_ANY: GPU instruction stalls for any reason
- GINS:STL_IFET: GPU instruction stalls for instruction fetch
- GINS:STL_GMEM: GPU instruction stalls for global memory
- GINS:STL_CMEM: GPU instruction stalls for constant memory
- GINS:STL_IDEP: GPU instruction stalls for instruction dependences
- GINS:STL_PIPE: GPU instruction pipeline stalls
- GINS:STL_MTHR: GPU instruction stalls for memory throttling

- GSAMP:EXP: expected number of samples
- GSAMP:TOT: total number of samples recorded
- GSAMP:UTIL: GPU utilization = (PC samples expected) / (PC samples total)

HPCToolkit Resources

- Documentation
  - —User manual for HPCToolkit: http://hpctoolkit.org/manual/HPCToolkit-users-manual.pdf
  - —Cheat sheet: https://gitlab.com/hpctoolkit/hpctoolkit/-/wikis/HPCToolkit-cheat-sheet
  - —User manual for hpcviewer: https://hpctoolkit.gitlab.io/hpcviewer
  - —Tutorial videos
    - http://hpctoolkit.org/training.html
    - recorded demo of GPU analysis of Quicksilver: https://youtu.be/vixa3hGDuGg
    - recorded tutorial presentation including demo with GPU analysis of GAMESS: https://vimeo.com/781264043
- Software
  - —Download hpcviewer GUI binaries for your laptop, desktop, cluster, or supercomputer
    - OS: Linux, Windows, MacOS
    - Processors: x86_64, aarch64, ppc64le
    - http://hpctoolkit.org/download.html
  - —Install HPCToolkit on your Linux desktop, cluster, or supercomputer using Spack
    - http://hpctoolkit.org/software-instructions.html

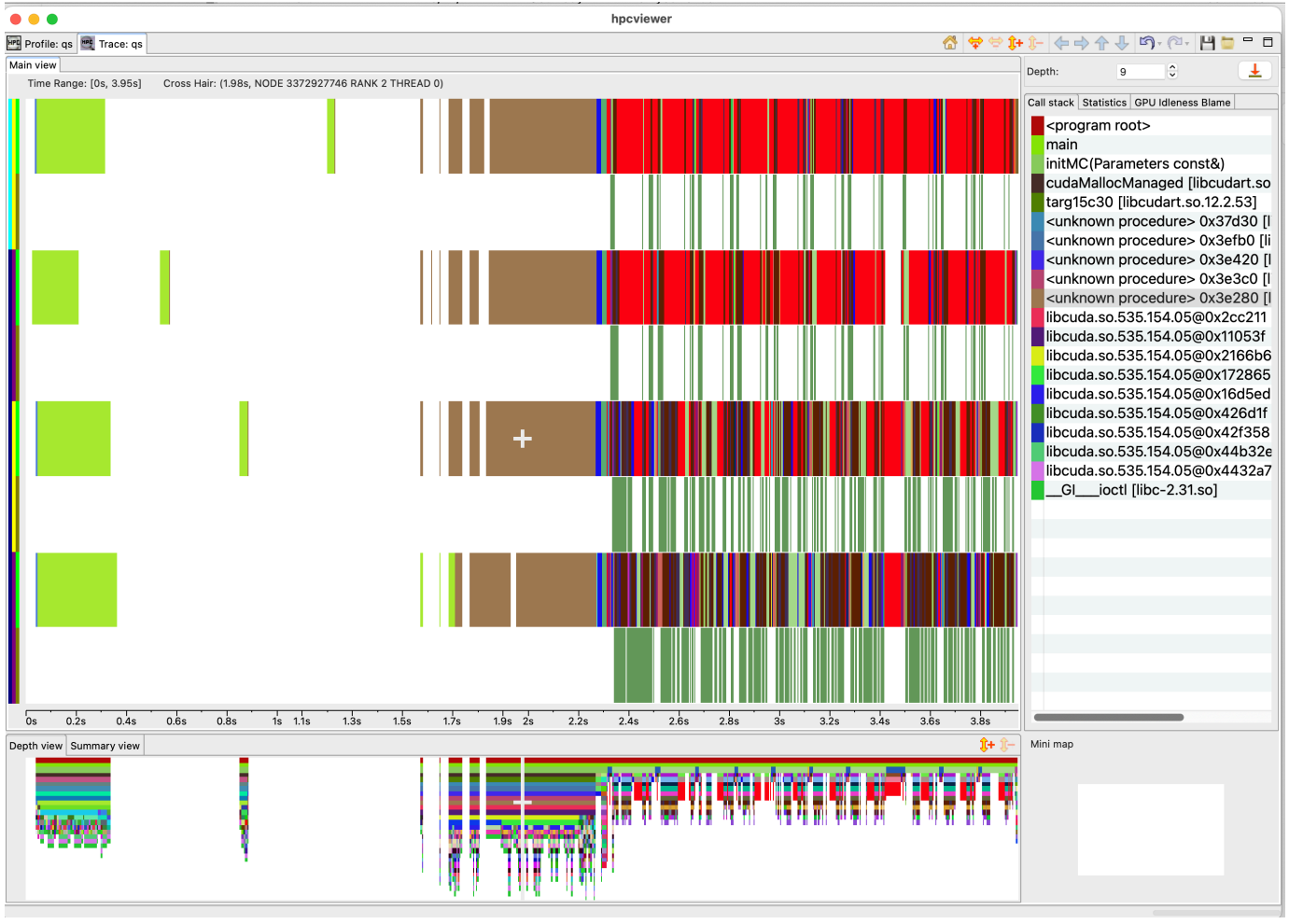
Some Hpcviewer Tips

Information for Using Hpcviewer

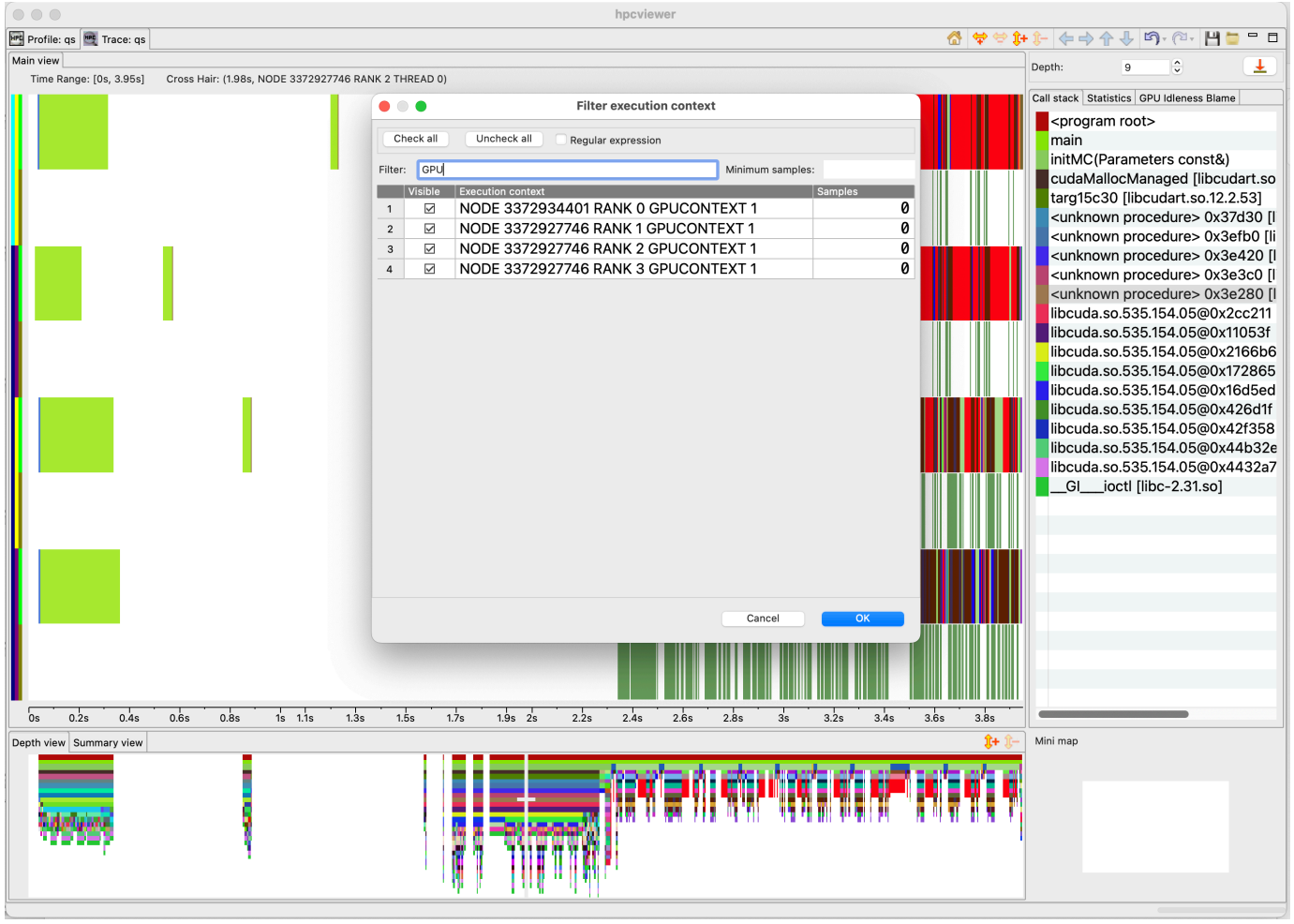
- Filtering GPU traces
  - Can use the filter menu to select what execution traces you want to see
    - cpu only, gpu, a mix
    - type a string or a regular expression in the chooser select or unselect the new set
    - only traces that exceed a minimum number of samples
- Filtering GPU calling context tree nodes to hide clutter
  - hide individual CCT nodes: e.g. lines that have no source code mapping library@0x0f450
  - hide subtrees: MPI implementation, implementation of CUDA primitives
- When inspecting GPU activity, be aware that hpcviewer has two modes
  - expose GPU traces or not
    - means: when displaying GPU trace lines, don't just show GPU activity if the time in the middle of a pixel is in a GPU operation. instead, show the first (if any) GPU operation between the time in the middle of the pixel and the middle of the next pixel
    - why? GPU activity is so short, it may be hard to find if we don't "expose" where it is
    - downside: makes the GPU appear more active than it is
      - can correct the statistics by turning the mode off
  - mode can be selected from <File>:<Preferences>:<Traces>

Quicksilver Hand-on Notes

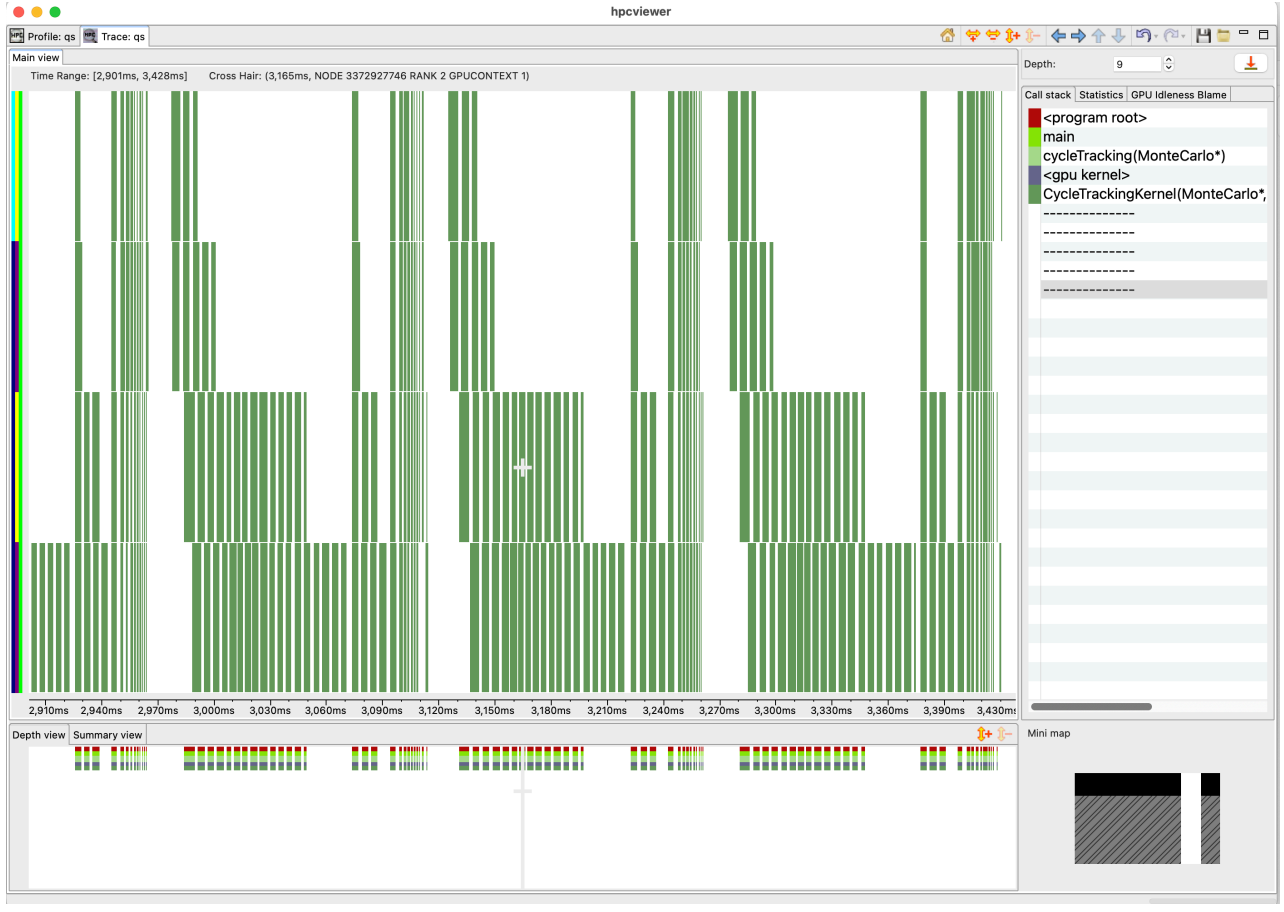
Select the Tab “Trace: qs”



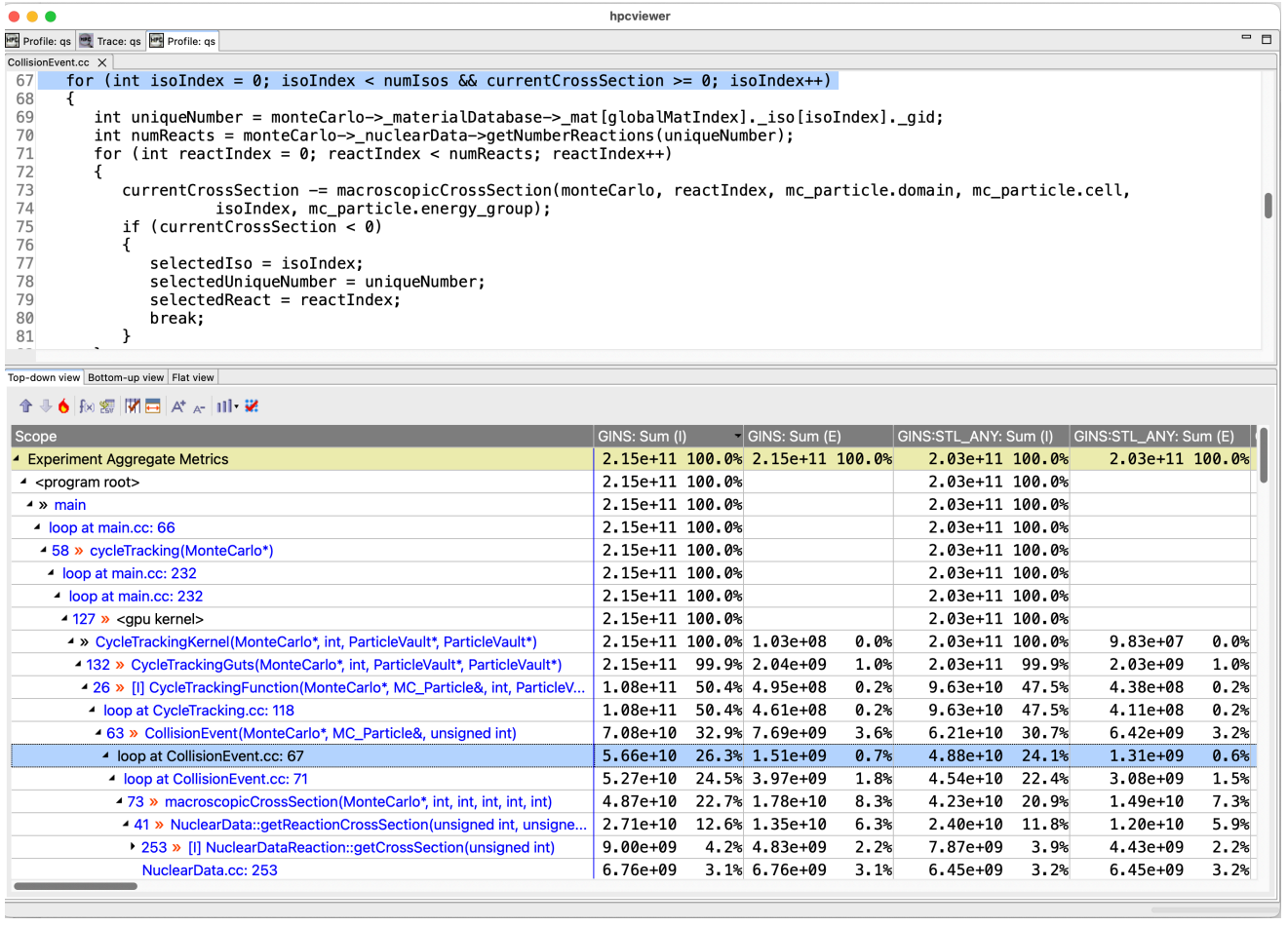
Use the Filter to “Uncheck all” and Check “GPU” streams



See Load Imbalance Across the Four GPUs



The Profile View in the other “PC Sampling” Database



Hands-on Tutorial Examples

```
%git clone https://github.com/hpctoolkit/hpctoolkit-tutorial-
  examples
%cd hpctoolkit-tutorial-examples/gpu/nvidia
%ls
%    arborx.kokkos    lammps.kokkos    quicksilver.cuda
```

Analyzing Quicksilver Traces

Using a measurement database with profiles and traces

- Select the Trace tab “Trace: qs”
- Identifying the traces
  - Select a pixel on a trace line
  - Look at legend on the top of the display, which reports the location of the “cross hair”
  - Is this a CPU or GPU trace line?
  - Repeat this a few times to identify what each of the trace lines represents
- Notice that each time you select a colored pixel on a trace line, you will be shown the function call stack in the rightmost pane
- At the top of the pane is a “depth” indicator, that indicates what level in the call stack you are viewing. The selected level will also be highlighted
- You can change the depth of your view by using the depth up/down, typing a depth, or simply selecting a frame in the call stack at the desired depth
- You can select  above the call stack frame to show the call stacks at the deepest depth
  - If a sample doesn’t have an entry at the selected depth, its deepest frame will be shown

Analyzing Quicksilver Traces

Using a measurement database with profiles and traces

- Zoom in on a region in a trace by selecting it in the trace display
- Use the back button  to undo a zoom
- Use the control buttons  at the top of the trace pane to
  - expand or contract the pane
  - move left, right, up, or down
- Keep an eye on the minimap in the lower right corner of the display to know what part of the trace you are viewing
- Use the home button  to reset the trace view to show the whole trace

Analyzing Quicksilver Traces

Using a measurement database with profiles and traces

- Select the Trace tab “Trace: qs”
- Configure filtering
  - Use the Filter menu to select Filter Execution Contexts
  - In the filtering menu, select "Uncheck all"
  - Now, in the empty box preceded by "Filter:", type "GPU" and then click "Check all”
  - Select "OK".
  - Now, the Trace View will show only trace lines for the GPUs.
- Inspect the trace data
  - Is the work load balanced across the GPUs? How can you tell?
  - Bring up the filter menu again. Select "Uncheck all". Type in "RANK 3" in the Filter box. Select thread 0 and the GPU context. Select “OK”.
  - Move the call stack to depth 2
    - What CPU function is Rank 3 thread 0 executing when the GPU is idle?
    - Does this suggest any optimization opportunities?

Analyzing the Quicksilver Summary Profile

Using a measurement database with profiles and traces

- Select the Profile Tab “Profile: qs”
- Use the column selector  to deselect and hide the two REALTIME columns
- Select the GPU OPS column, which represents time spent in all GPU operations
- Select the  button to show the “hot path” according to the selected column
  - the hot path of parent will continue into a child as long as the child accounts for 50% or more of the parent’s cost
- The hot path will select “CycleTrackingKernel” — a GPU kernel that consumes 100% of the GPU cost in this profile
- Use the  button to graph “GPU OPS (I)” — inclusive GPU operations across the profiles
  - Are the GPU operations balanced or not across the execution contexts (ranks)?

Analyzing the Quicksilver Summary Profile

- You will notice that for quicksilver, HPCToolkit doesn't report any data copies between the host and device
  - The quicksilver code uses "unified memory" so that all of the data movement occurs between CPU and GPU using page faults rather than explicit copies
  - Today's GPU hardware doesn't support attribution of page faults to individual instructions
    - We could profile them, but not attribute them to code

Analyzing Quicksilver PC Samples

Using a measurement database with traces that was collected *with* PC sampling enabled

Using the default top-down view of the profile

- Select the column “GINS (I)” to focus on the measurement of inclusive GPU Instructions
- Select use the flame button to look at where the instructions are executed
- In the call stack revealed, you will <gpu kernel> placeholder that separates CPU activity (above) from GPU kernel activity (below)
- Below the <gpu kernel> placeholder you will see the function calls, inlined functions, loops and statements in HPCToolkit’s reconstruction of calling contexts within the CycleTrackingKernel

- Using the bottom-up view of the profile
  - Select the bottom-up tab of above the control pane
  - Select the GINS STL_ANY (E) column, which will sort the functions by the exclusive GPU instruction stalls within that function
  - Scroll right to see which of the types of contributing types of stalls accounts for most of the STL_ANY amount
  - Select the function that has the most exclusive stalls
  - Select the the hot path to see where this function is called from.
    - Where do the calls to the costly function come from?
    - Does there appear to be an opportunity to reduce the number of calls to this function?

Filtering Tips to Hide Unwanted Implementation Details

- Filter “descendants-only” of CCT nodes with names *MPI* to hide the details of MPI implementation in profiles and traces
- Filter internal details of RAJA and SYCL templates to suppress unwanted detail using a “self-only” filter

Other Databases to Inspect

Other Performance Databases

See /tmp/hpctoolkit-databases for each of the following

- Quicksilver PC Sampling
- QMCPACK CPU prototype: a Quantum Monte Carlo code
  - inspect performance data from an early prototype on 32 MPI ranks x 32 threads per rank
- Deepware-rtm: Reverse time migration using Pytorch
  - trace data shows forward and backward phases utilizing a GPU
  - PC sampling measurements shows GPU stencil computations
- PeleC: a turbulent combustion code being used to simulate jet aircraft engines
  - large data from 2025 NERSC hackathon run on 16 CPUs + 16 GPU