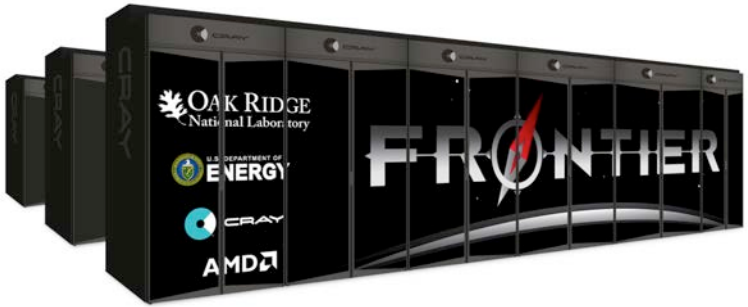
Towards Performance Tools for Emerging GPU-Accelerated Exascale Supercomputers

John Mellor-Crummey
Rice University

18th Annual Workshop on Charm++ and Its Applications
October 21, 2020
University of Illinois at Urbana-Champaign (Virtual)

DOE's Forthcoming Heterogeneous Exascale Platforms



- **Aurora compute nodes (ALCF)**
  - 2 Intel Xeon "Sapphire Rapids" processors
  - 6 Intel Xe "Ponte Vecchio" GPUs
  - 8 Slingshot endpoints
  - Unified memory architecture
- **Frontier compute nodes (OLCF)**
  - 1 AMD EPYC CPU
  - 4 purpose-built AMD Radeon Instinct GPUs
  - multiple Slingshot endpoints
  - Unified memory architecture
- **El Capitan compute nodes (LLNL)**
  - Next-generation AMD EPYC "Genoa" CPU (5nm)
  - Next-generation AMD Radeon Instinct GPUs

Node-level Programming Models for Forthcoming Exascale Systems

- **Native programming models from platform vendors**
  - Intel DPC++

    C++ **+** SYCL specification 1.2.1 **+** extensions: device queues, buffers, accessors, parallel_for, single_task, parallel_for_work_group, events
  - HIP: AMD's CUDA-like model
- **Directive-based models**
  - OpenACC
  - OpenMP

  offload structured blocks and device functions, work sharing loops, data environment, data mappings
- **C++ template-based models**
  - RAJA
  - Kokkos

  parallelism loops, iteration spaces, execution policies, traversal templates, lambda functions, n-dimensional array abstractions, and lambda functions

A Diverse Set of Global Programming Models

- **Message passing**
  - MPI
- **Partitioned global address space programming models**
  - languages
    - Coarray Fortran, Coarray C++, Chapel, UPC
  - librarys
    - UPC++, GASNet, OpenSHMEM , Global Arrays
- **Object-based**
  - Charm++

Source: Frontier Spec Sheet
https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf

Performance Analysis Challenges for GPU-accelerated Supercomputers

- **Myriad performance concerns**
  - Computation performance
    - Principal concern: keep GPUs busy and computing productively
      - need extreme-scale data parallelism!
  - Data movement costs within and between memory spaces
  - Internode communication
  - I/O
- **Many ways to hurt performance**
  - insufficient parallelism, load imbalance, serialization, replicated work, parallel overheads …
- **Hardware and execution model complexity**
  - Multiple compute engines with vastly different characteristics, capabilities, and concerns
  - Multiple memory spaces with different performance characteristics
    - CPU and GPU have different complex memory hierarchies
  - Asynchronous execution

Measurement Challenges for GPU-accelerated Supercomputers

- **Extreme-scale parallelism**
  - Serialization within tools will disrupt parallel performance
- **Multiple measurement modalities and interfaces**
  - Sampling on the CPU
  - Callbacks when GPU operations are launched and (sometimes) completed
  - GPU event stream
- **Frequent GPU kernel launches require a low-overhead measurement substrate**
- **Importance of third-party measurement interfaces**
  - Tools can only measure what GPU hardware can monitor
    - support for fine-grain measurement will be essential to diagnose GPU inefficiencies
  - Linux perf_events for kernel measurement
  - GPU monitoring libraries from vendors

Engineering Challenges for Performance Tools

- **Complex applications**
  - Compositions of programming models
  - > 100 dynamic libraries
  - Application binaries exceeding 5GB
  - HPC libraries that intercept system calls (mmap, munmap, open, close)
  - Quirky application characteristics
    - NAMD: exit initiated by a non-initial thread
    - Kull: forking non-readable helper application
- **Dynamic library loading**
  - Implicit system locks on dynamic library state
  - RUNPATH: library-specific library load path
  - Early threads in library init constructors
  - Nested dynamic library loading
- **Provisioning thread local state**
  - Implicit lock when creating or destroying thread local storage
- **Process fork**
  - atfork handlers trigger thread destructors
- **Interactions with vendor tool substrates**
  - Libraries lack documentation of their actions, e.g. creating threads
  - Callbacks for submission and completion on unspecified (and sometimes different) threads
- **Interaction between tools and software stack**
  - Interaction of signals with everything
  - Managing monitoring when forking
- **Lack of vendor tooling and documentation**
  - Non-standard GPU binary formats that lack public documentation

GPU Performance Tools

- **Features**
  - Trace view
    - A series of events that happen over time on each process, thread, and GPU stream
  - Profile view
    - A correlation of performance metrics with program contexts
- **Existing GPU performance tools**
  - GPU vendors
    - Nsight Systems, Nsight Compute, nvprof, ROCProfiler, Intel VTune
  - Third parties
    - TAU, VampirTrace, ARM Map

Tool Shortcomings for Analyzing Complex GPU-accelerated Programs

- They lack a comprehensive profile view to analyze
  - Sophisticated CPU calling contexts
    - A GPU API (e.g., cudaMemcpy) invoked in different places
  - Sophisticated GPU calling contexts
    - OpenMP Target, Kokkos, and RAJA generate code with many small procedures
- At best, existing tools only attribute runtime cost to a flat profile view of functions executed on GPUs

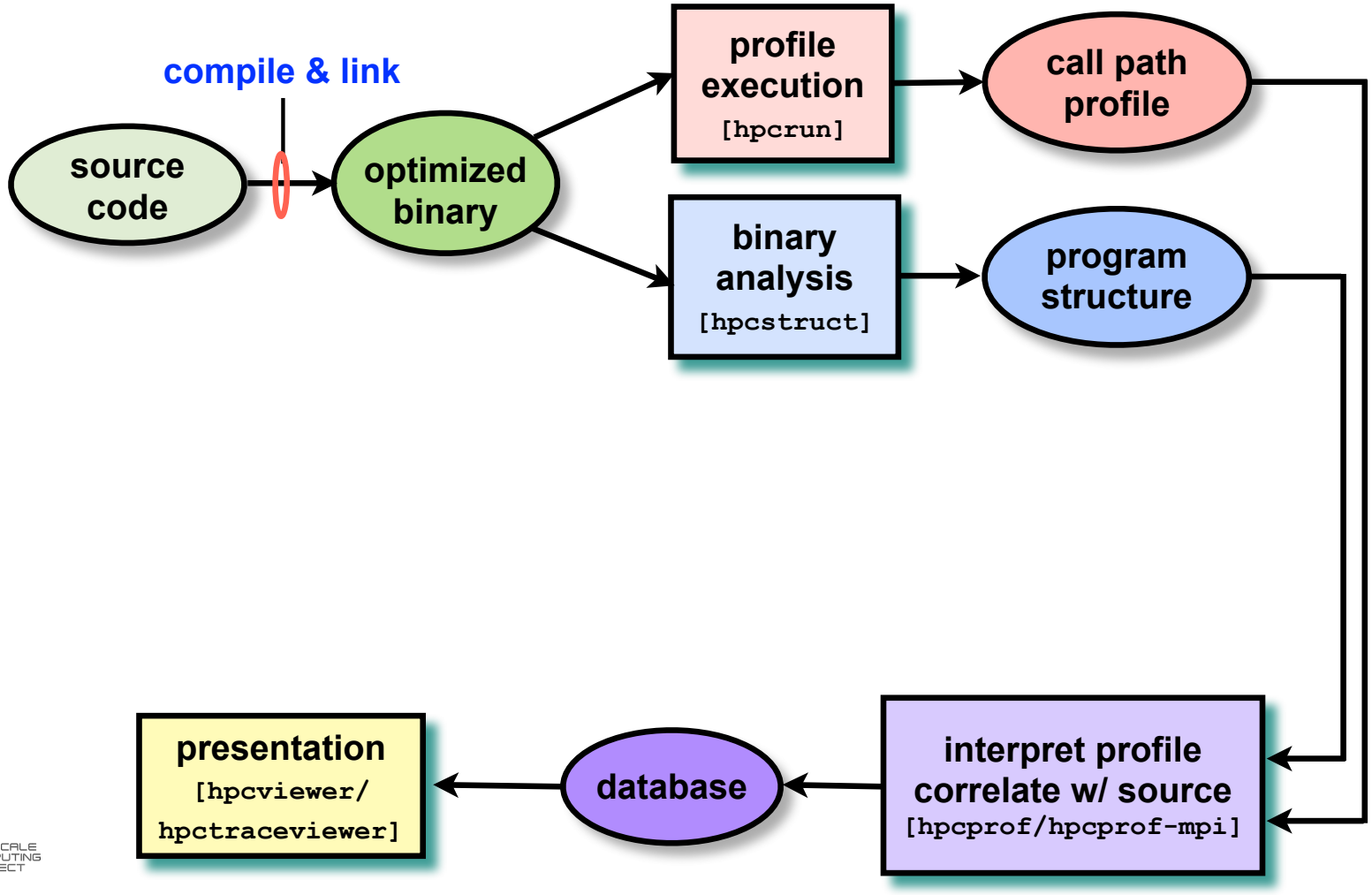
Outline

- **Performance measurement and analysis challenges for GPU-accelerated supercomputers**
- **Introduction to HPCToolkit performance tools**
  - Overview of HPCToolkit components and their workflow
  - HPCToolkit's graphical user interfaces
- **Analyzing the performance of GPU-accelerated supercomputers with HPCToolkit**
  - Overview of HPCToolkit's GPU performance measurement capabilities
  - Collecting measurements
  - Analysis and attribution
  - Scalable analysis of performance data
- **Status, ongoing work, final remarks**

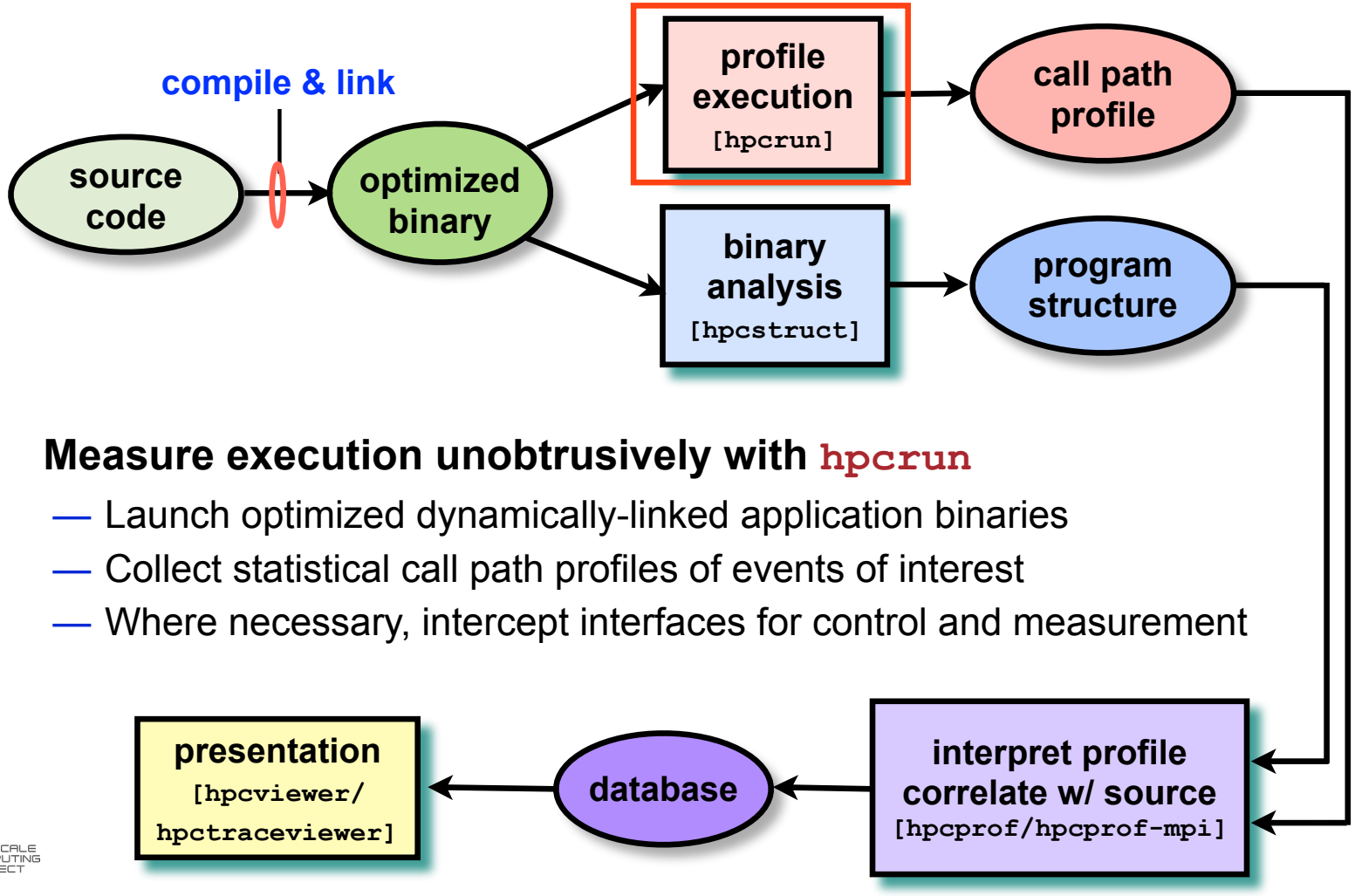
Rice University's HPCToolkit Performance Tools

- **Employs binary-level measurement and analysis**
  - Observes executions of fully optimized, dynamically-linked parallel applications
  - Supports multi-lingual codes with external binary-only libraries
- **Collects sampling-based measurements of CPU**
  - Controllable overhead
  - Minimize systematic error and avoid blind spots
  - Enable data collection for large-scale parallelism
- **GPU performance using measurement APIs provided by vendors**
  - Callbacks to monitor launch/completion of GPU operations
  - NVIDIA and AMD: activity API provides information about asynchronous operations on GPU devices
  - NVIDIA PC sampling and Intel GTPin instrumentation for fine-grain measurement
- **Associates metrics with both static and dynamic context**
  - Loop nests, procedures, inlined code, calling context on both CPU and GPU
- **Enables one to specify and compute derived CPU and GPU performance metrics of your choosing**
  - Diagnosis often requires more than one species of metric
- **Supports top-down performance analysis**
  - Identify costs of interest and drill down to causes: up and down call chains, over time

HPCToolkit Workflow



HPCToolkit Workflow



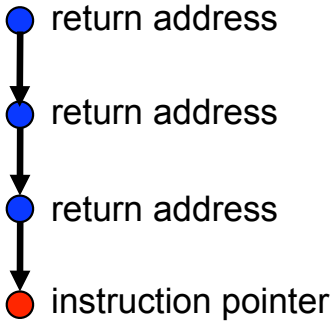
Measure execution unobtrusively with `hpcrun`

- Launch optimized dynamically-linked application binaries
- Collect statistical call path profiles of events of interest
- Where necessary, intercept interfaces for control and measurement

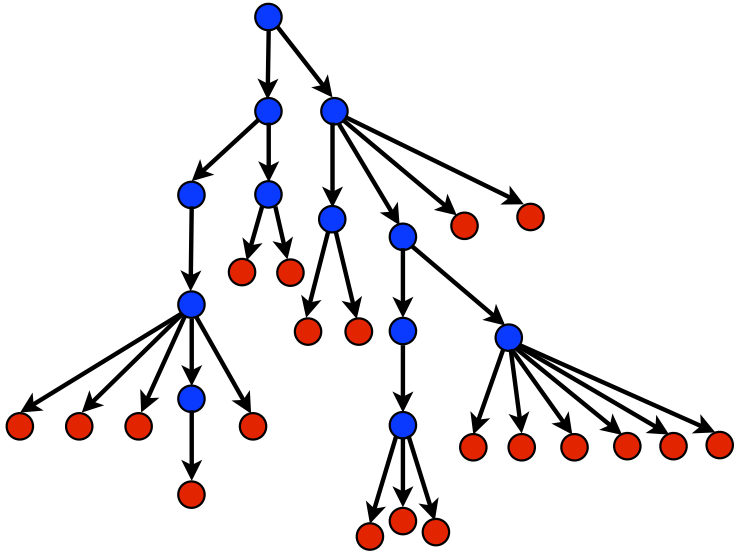
Call Path Profiling

- **Measure and attribute costs in context**
  - Sample timer or hardware counter overflows
  - Gather CPU calling context using stack unwinding

Call path sample

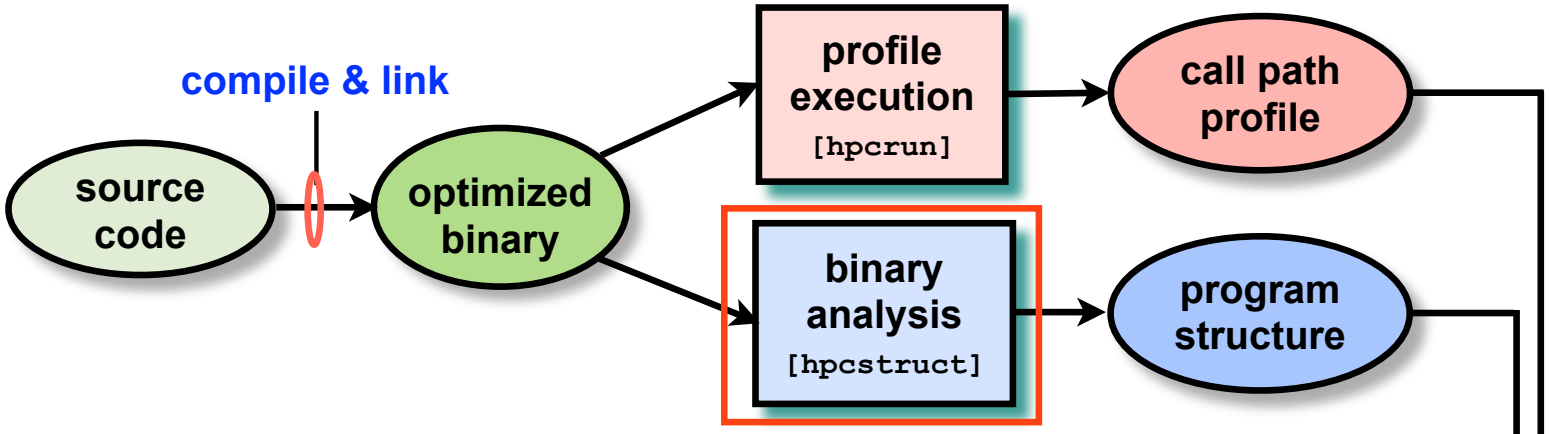


Calling context tree



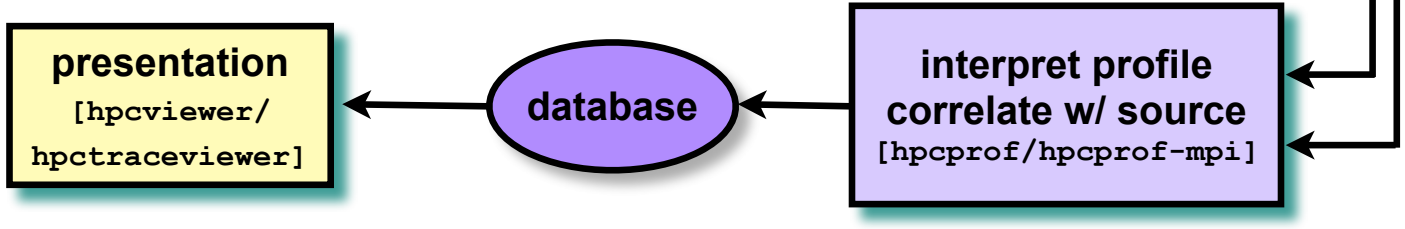
Overhead proportional to sampling frequency, not call frequency

HPCToolkit Workflow

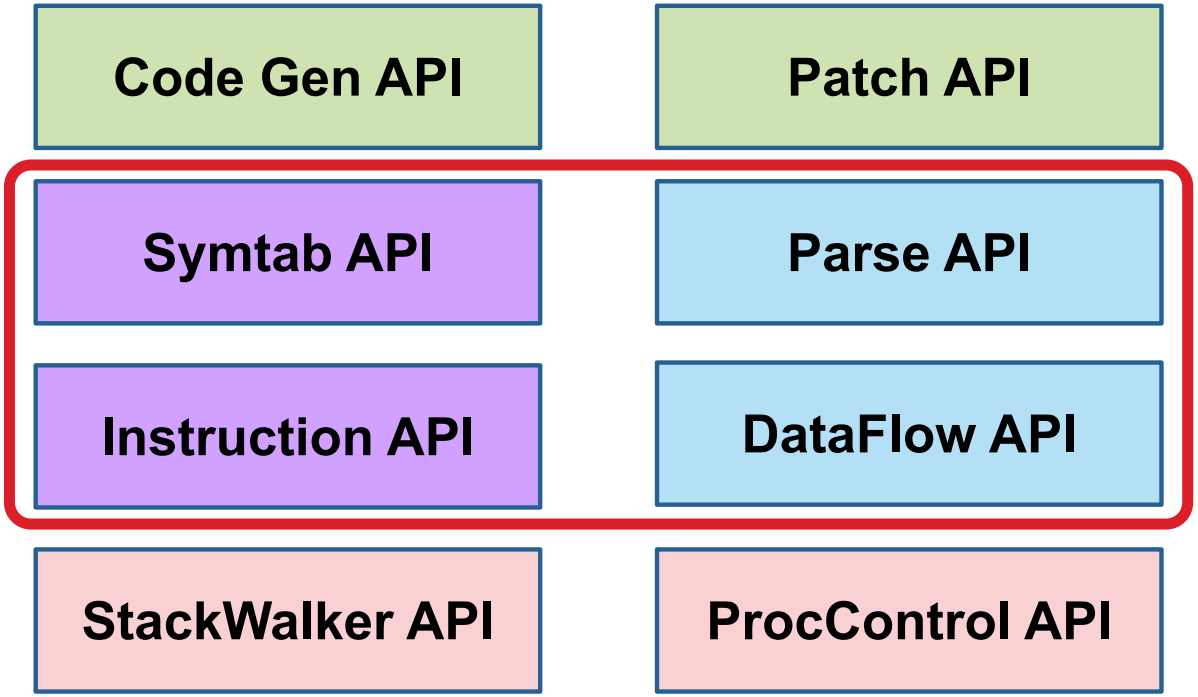


Analyze binary with `hpcstruct`: recover program structure

- Analyze machine code, line map, debugging information
- Extract loop nests & identify inlined procedures
- Map transformed loops and procedures to source



Dyninst: A Toolkit for Binary Analysis and Instrumentation



Architectures

X86_64

Power/BE

Power/LE

ARM

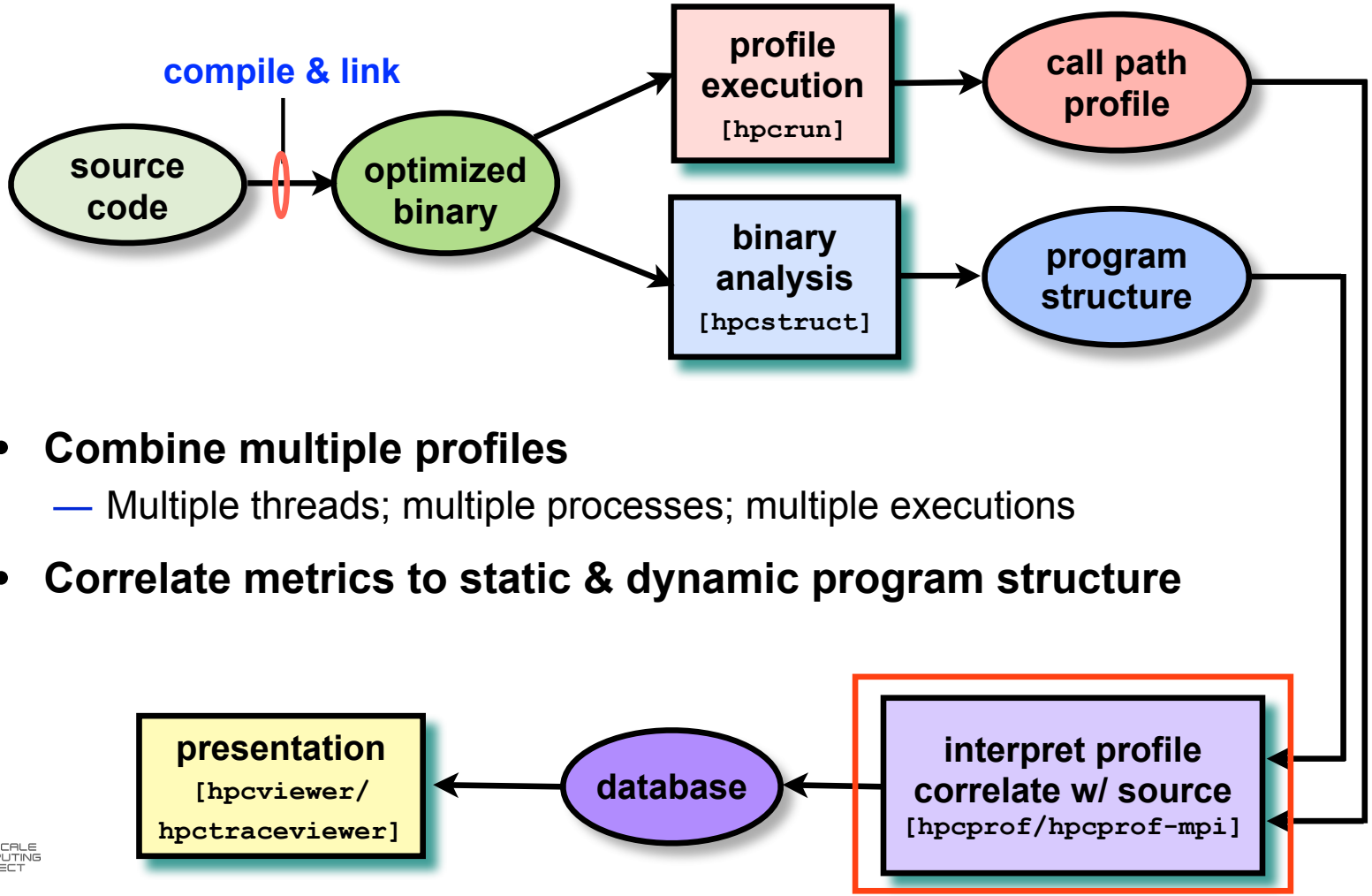
AMD Vega

CUDA

Intel GPU

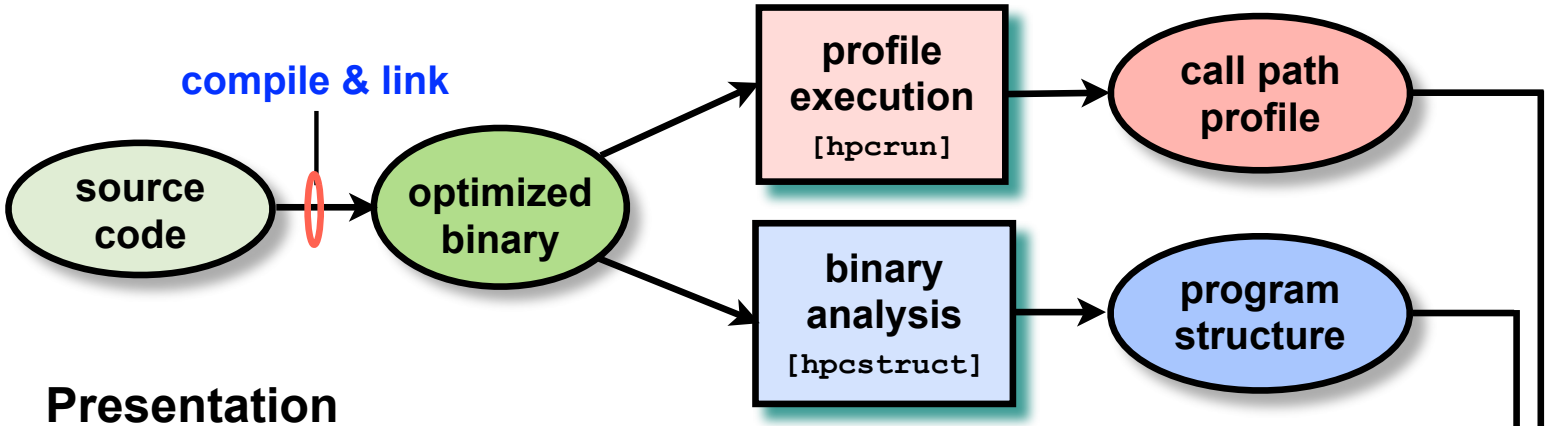
Lead Institution: University of Wisconsin – Madison

HPCToolkit Workflow



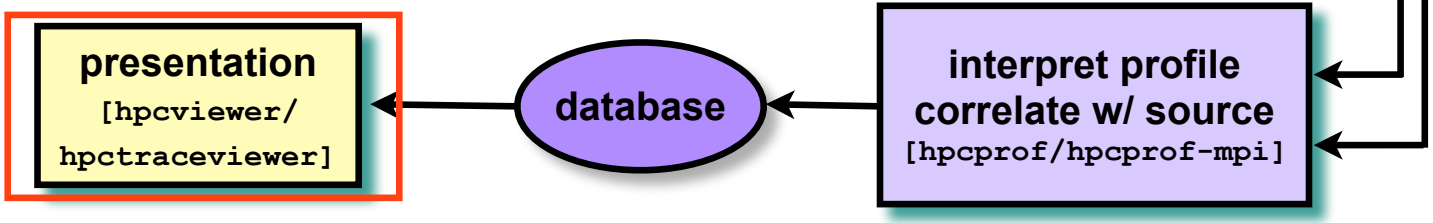
- **Combine multiple profiles**
  - Multiple threads; multiple processes; multiple executions
- **Correlate metrics to static & dynamic program structure**

HPCToolkit Workflow

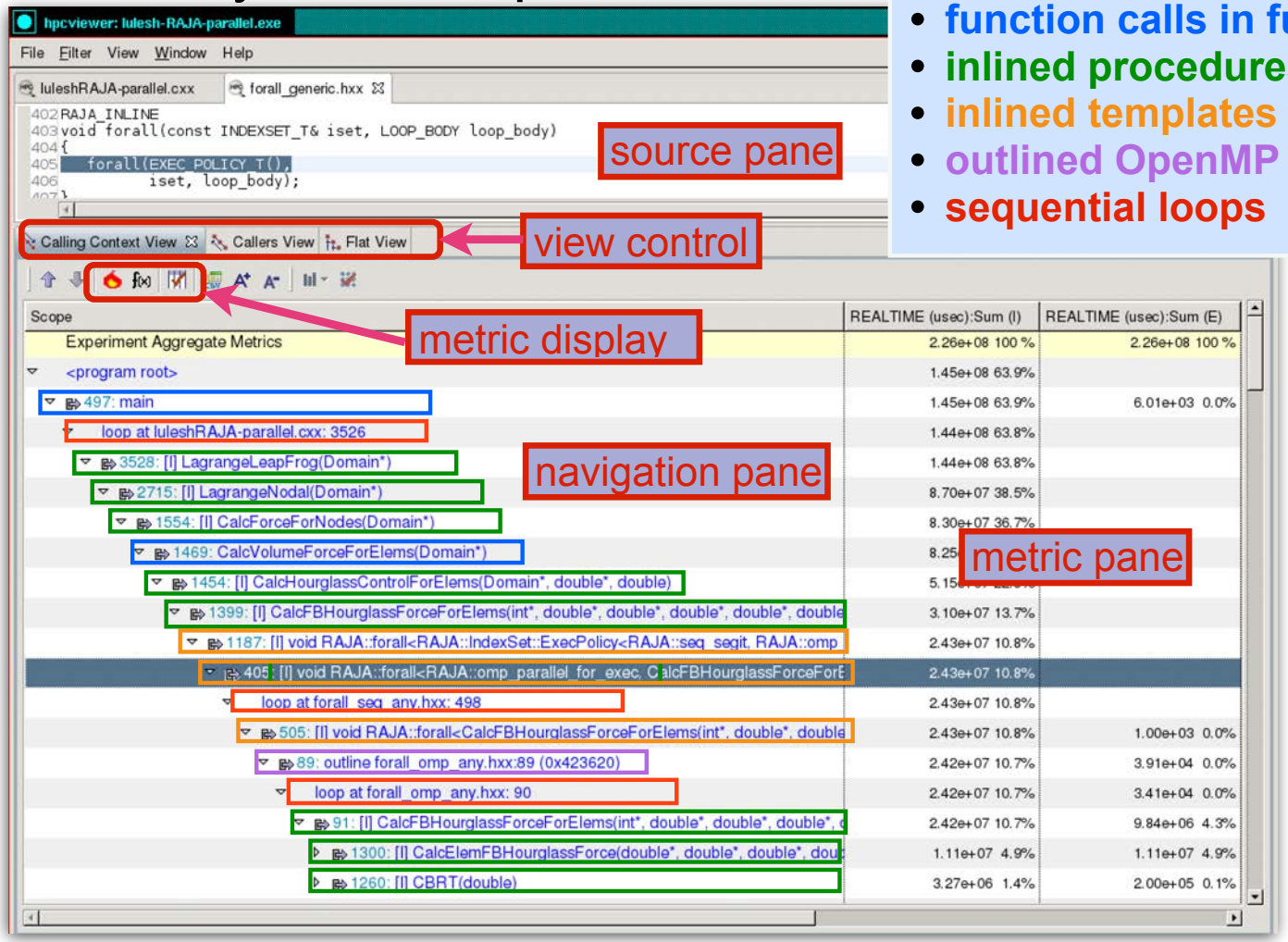


Presentation

- Explore performance data from multiple perspectives
  - Rank order by metrics to focus on what's important
    e.g., cycles, instructions, GPU instructions, GPU stalls
  - Compute derived metrics to help gain insight
    e.g. scalability losses
- Explore evolution of behavior over time



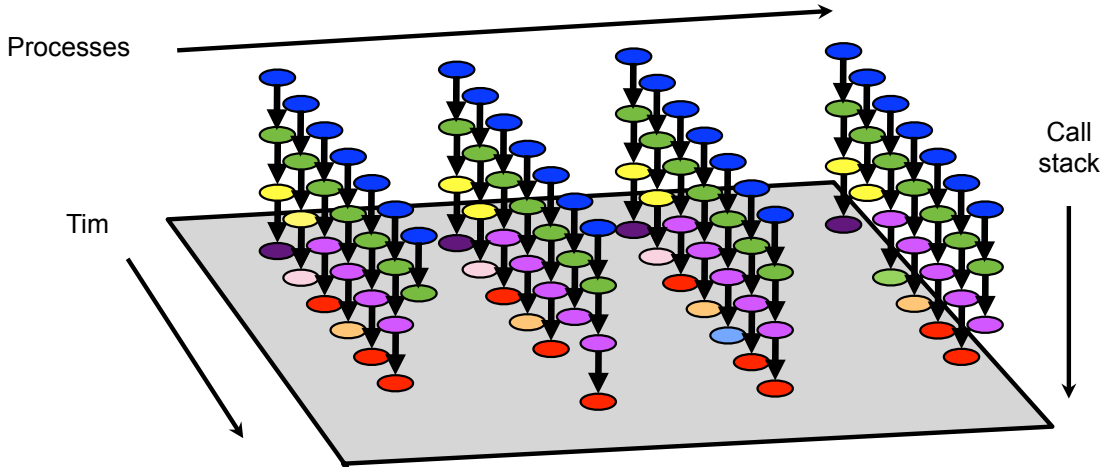
Code-centric Analysis with hpcviewer



- function calls in full context
- inlined procedures
- inlined templates
- outlined OpenMP loops
- sequential loops

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

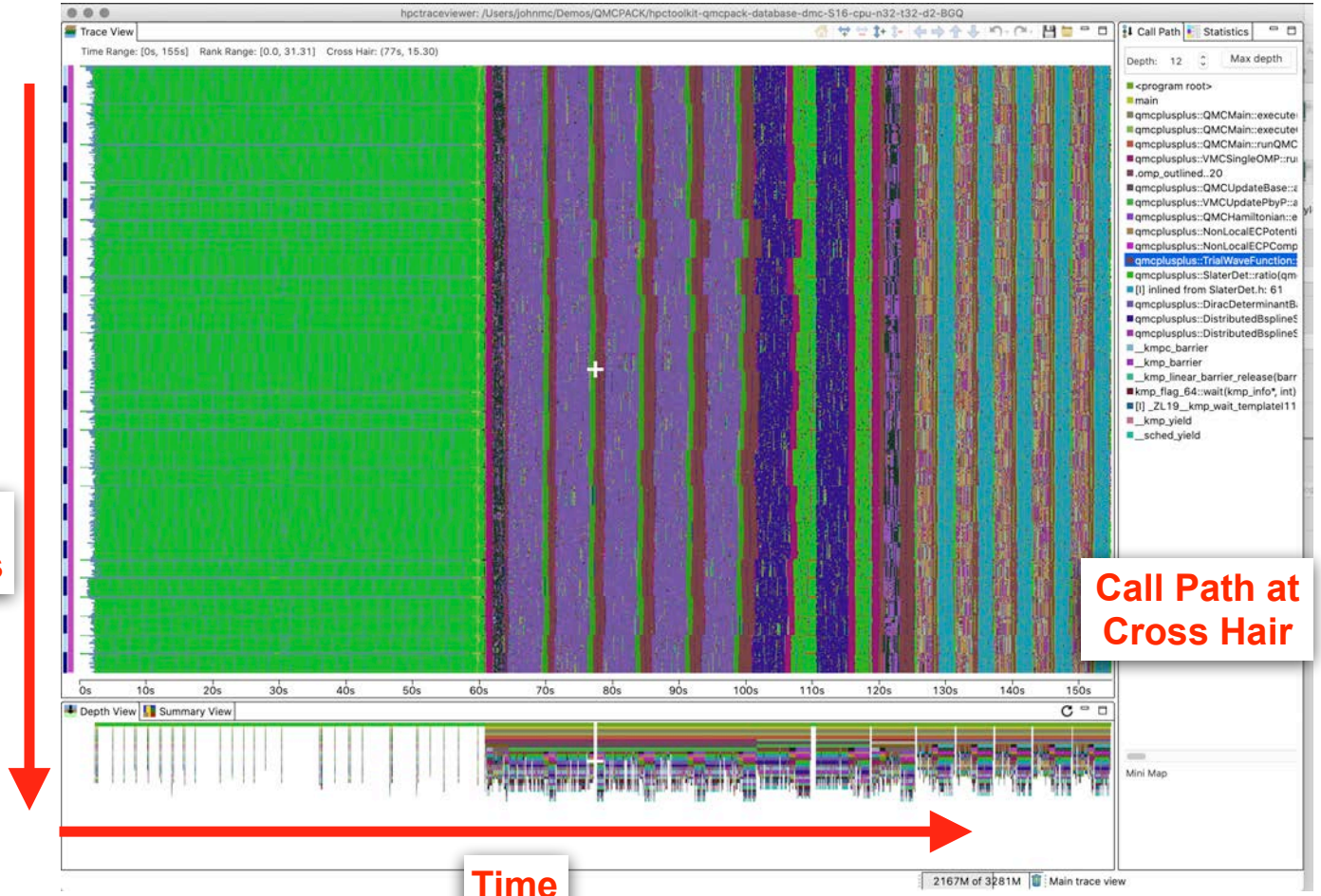


Time-centric Analysis with hpctraceviewer

Experimental version of QMCPack on Blue Gene Q

- 32 ranks
- 32 threads each

Call stack view

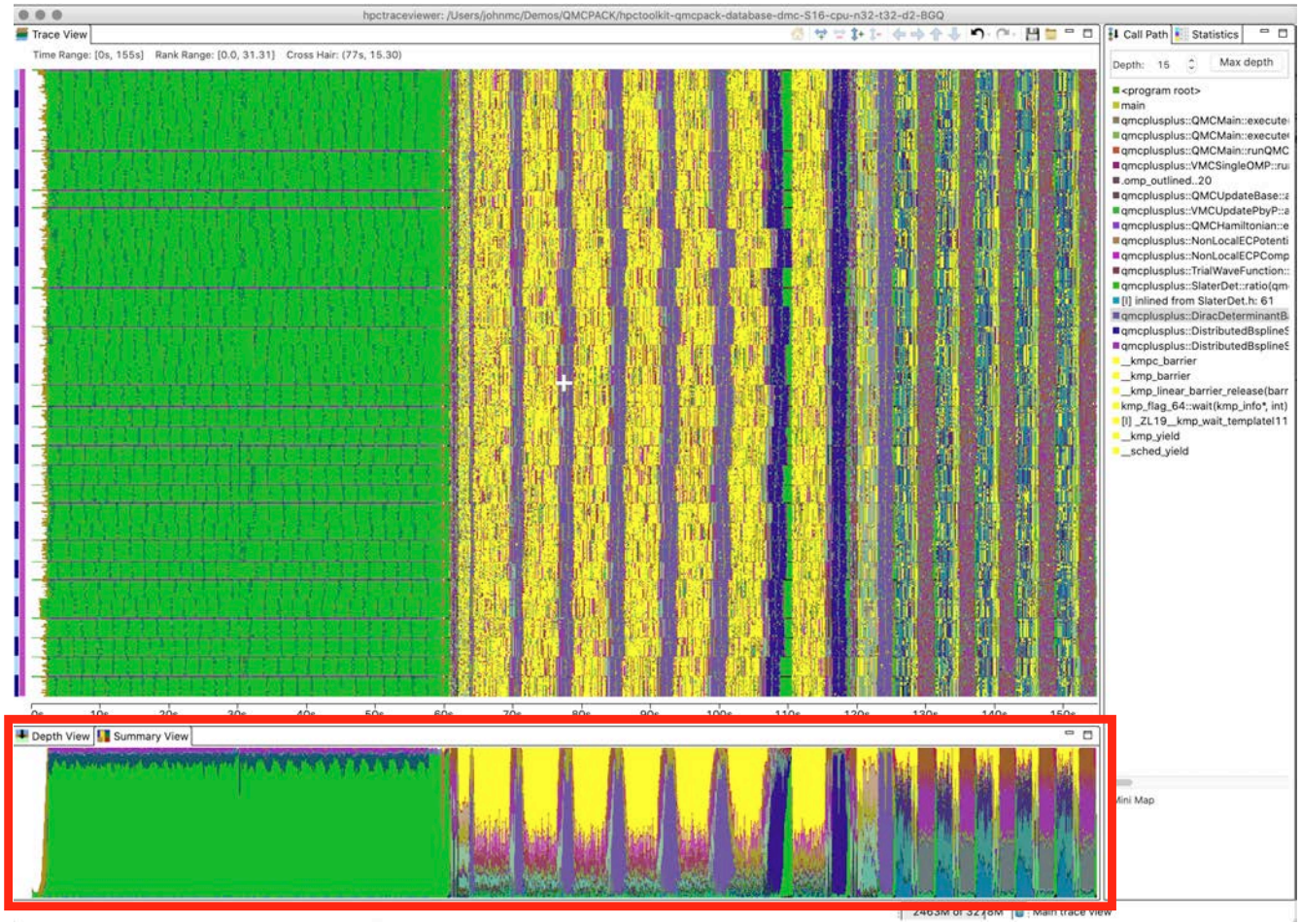


Time-centric Analysis with hpctraceviewer

Experimental version of QMCPack on Blue Gene Q

- 32 ranks
- 32 threads each

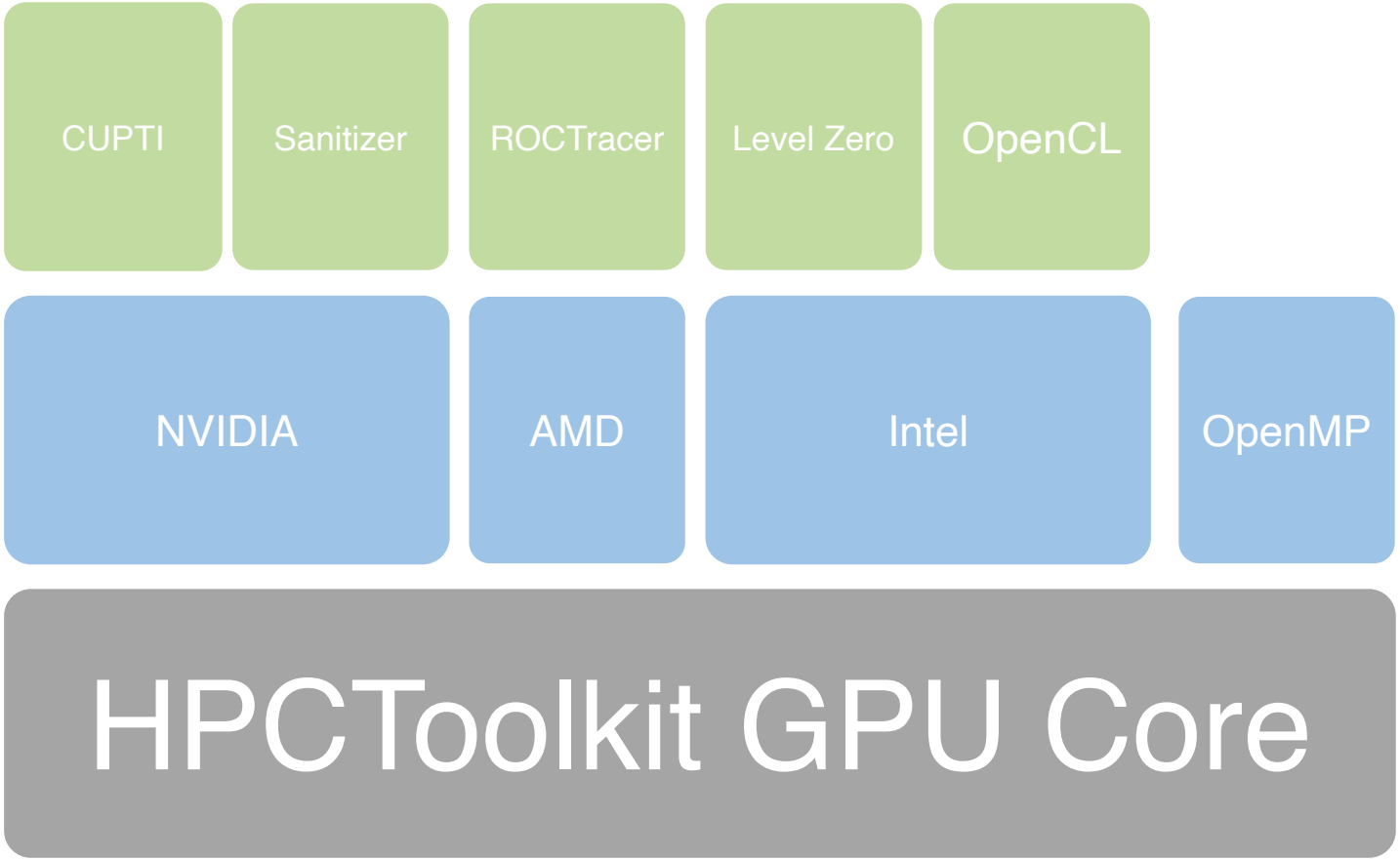
Summary view modeled after Projections



Outline

- **Performance measurement and analysis challenges for GPU-accelerated supercomputers**
- **Introduction to HPCToolkit performance tools**
  - Overview of HPCToolkit components and their workflow
  - HPCToolkit's graphical user interfaces
- **Analyzing the performance of GPU-accelerated supercomputers with HPCToolkit**
  - Overview of HPCToolkit's GPU performance measurement capabilities
  - Collecting measurements
  - Analysis and attribution
  - Scalable analysis of performance data
- **Status, ongoing work, final remarks**

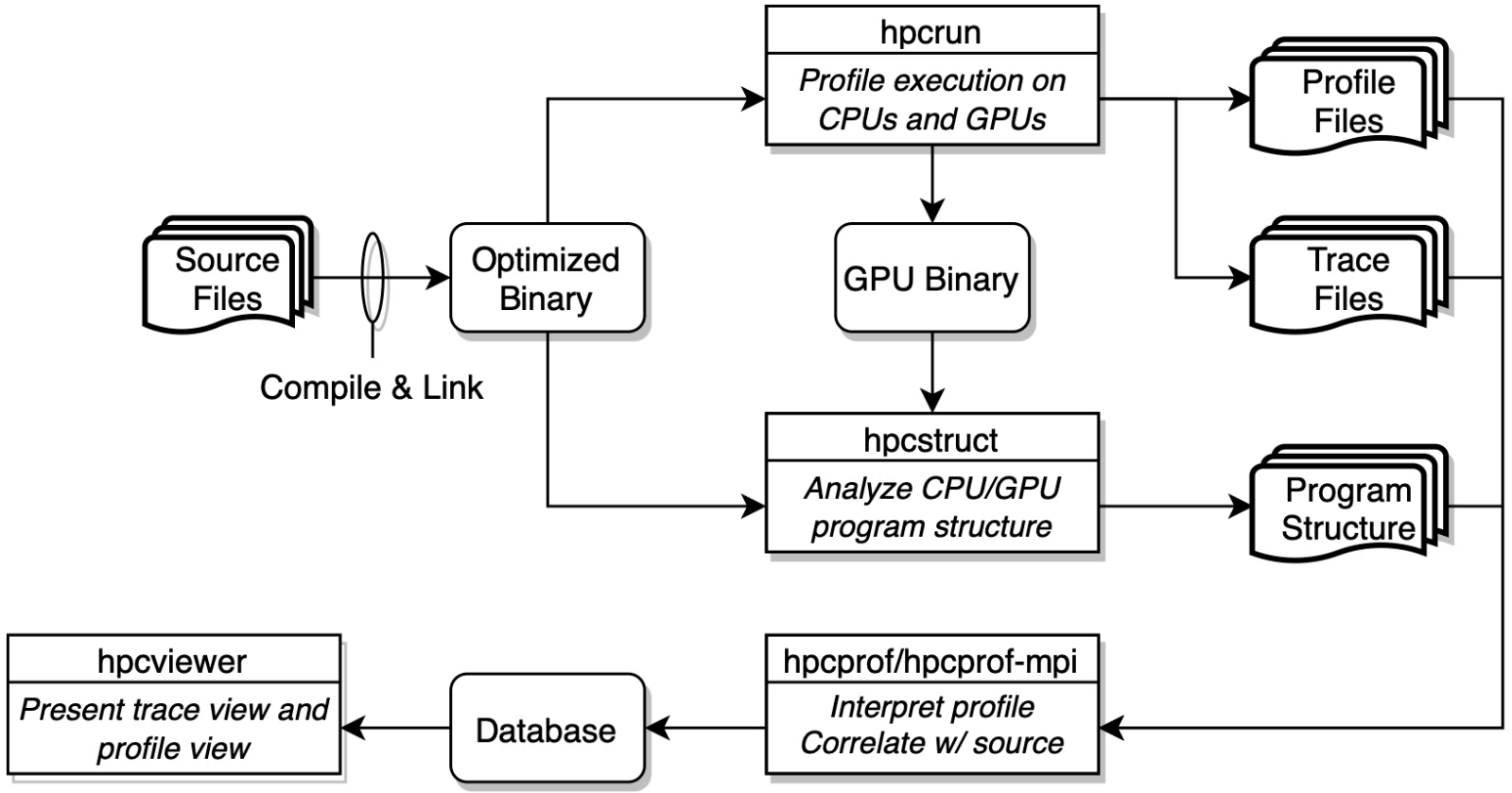
HPCToolkit for GPU-accelerated Computations



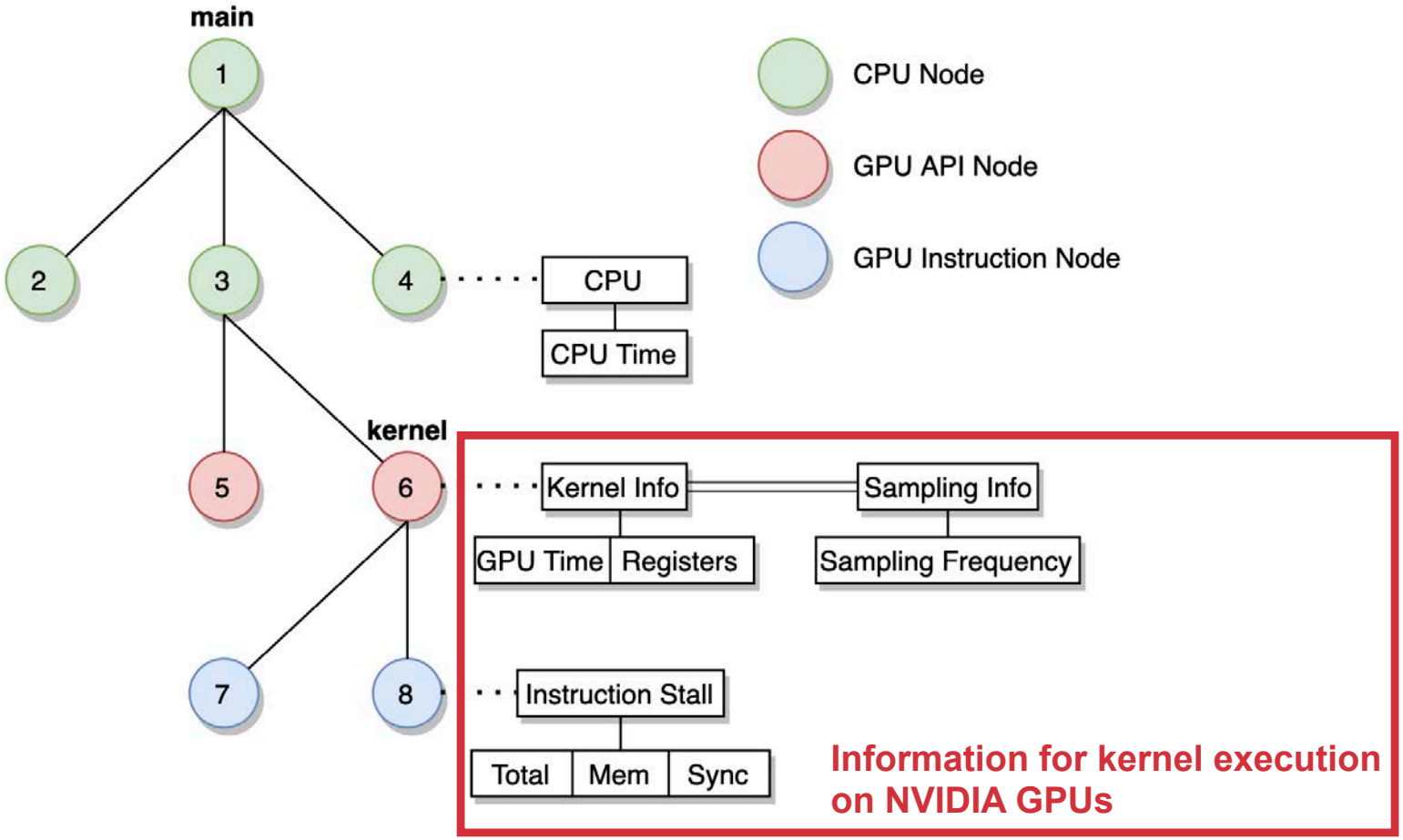
Highlights of HPCToolkit's Support for GPU-accelerated Codes

- **It unwinds the CPU call stack to identify the CPU calling context for each GPU API invocation**
- **It employs a novel and fast wait-free data structure for inter-thread communication**
- **It employs binary analysis of GPU code to attribute fine-grain performance measurements to functions, inlined functions and templates, loops, and statements**
  - NVIDIA, Intel, and AMD GPU binaries
- **It uses a novel technique to reconstruct an approximate GPU calling context tree for computations from instruction-level measurements**
- **On NVIDIA GPUs: derive a rich set of metrics from PC samples from a single execution**
- **It performs scalable analysis of sparse representations of performance measurements and produces sparse representations tailored for graphical user interfaces**

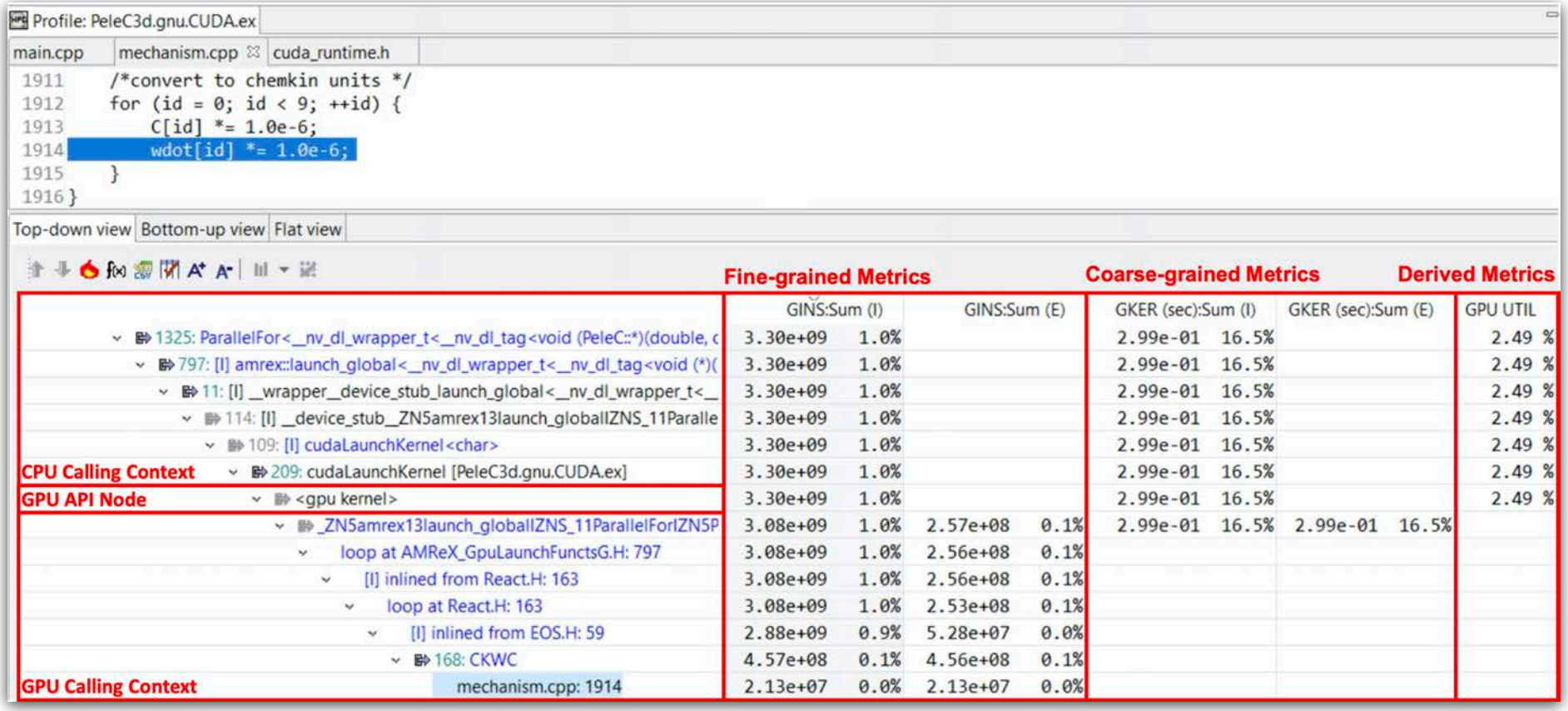
HPCToolkit's Workflow for GPU-accelerated Applications



HPCToolkit's Sparse Representation of Measurements at Run-time



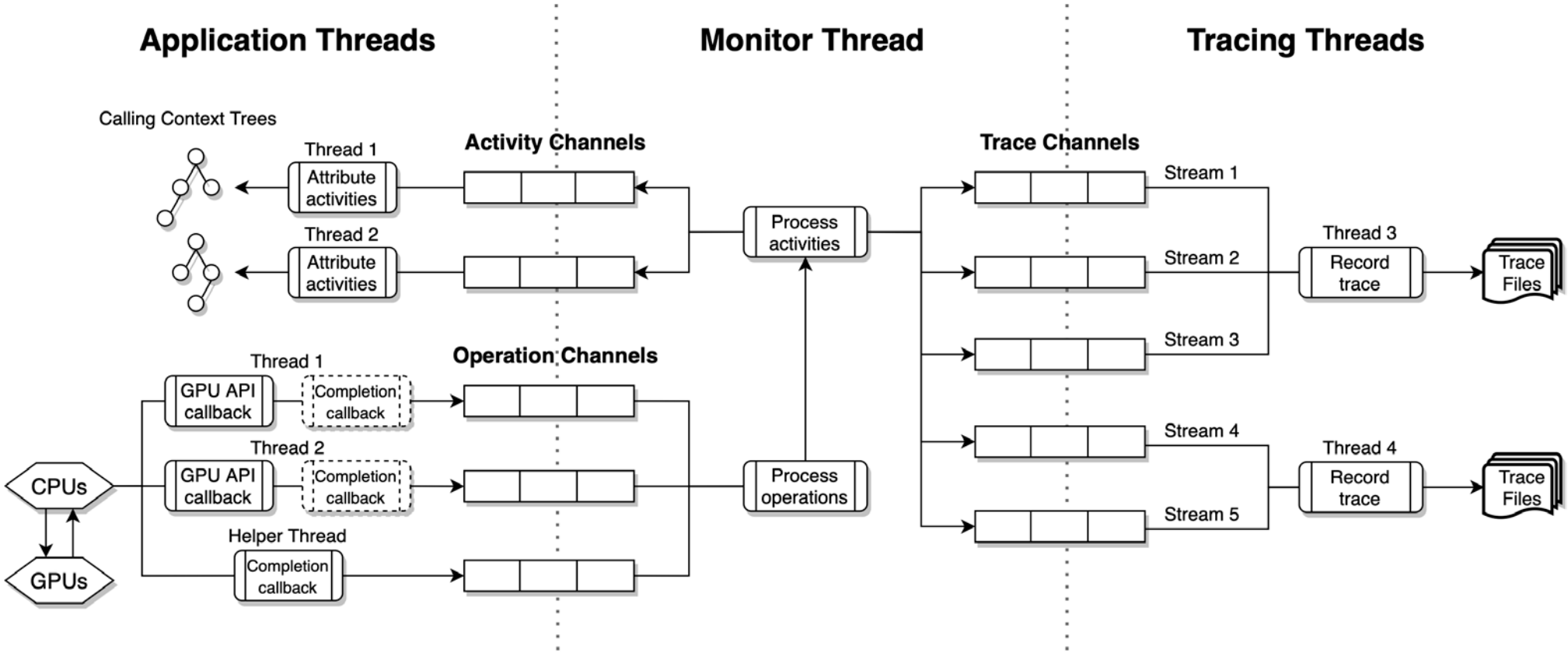
HPCToolkit's Code-Centric Profiles of GPU-accelerated Code



GPU Performance Measurement

- Three categories of threads
  - Application Threads (*N* per process)
    - Launch kernels, move data, and synchronize GPU calls
  - Monitor Thread (*1* per process)
    - Monitor GPU events and collect GPU measurements
  - Tracing Threads (*1* for every K GPU streams)
- Interactions
  - **Create correlation:** An application thread *T* creates a correlation record when it launches a kernel and tags the kernel with a correlation ID *C*, notifying the monitor thread that *C* belongs to *T*
  - **Attribute measurements**: The monitor thread collects measurements associated with *C* and communicates measurement records back to thread *T*
  - **Record traces:** The monitor thread sends activity traces to tracing threads to record in a separate trace file per GPU stream (NVIDIA, AMD) or device queue (Intel, AMD)

HPCToolkit's Runtime Monitoring Infrastructure for OpenCL



Wait-free Channels for Inter-thread Transfer of Measurement Data

- Unidirectional wait-free channel between a pair of threads
  - implemented by a pair of stacks: one private, one shared
- Operations
  - **Producer PUSH**(*CAS*): push an item on a shared stack
  - **Consumer STEAL**(*XCHG*): steal the contents of the shared stack, push the contents onto a private stack
  - **Consumer POP**: pop an item from the private stack
- Wait-free because **PUSH** fails at most once when a concurrent thread **STEALs** contents of the shared stack
- Bi-directional channel: pair of wait-free unidirectional channels

Correlating GPU API Invocations with CPU Calling Context

- Unwind a call stack from each API invocation
  - Kernel launch, memory copy, and synchronization
- Initial approach
  - Identify the function enclosing each call site in the call stack using a global shared map
- Problem
  - Applications have deep call stacks and large codebase
    - Nyx: up to 60 layers and 400k calls
    - Laghos: up to 40 layers and 100k calls

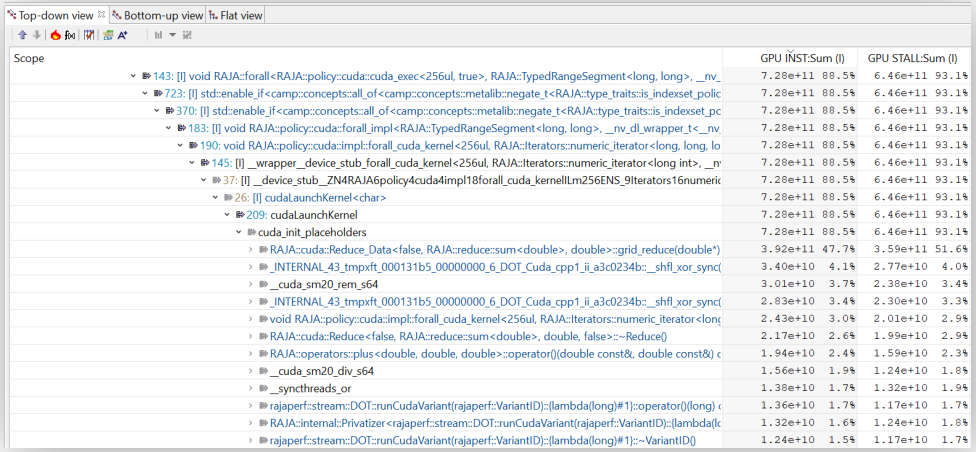
Fast Unwinding

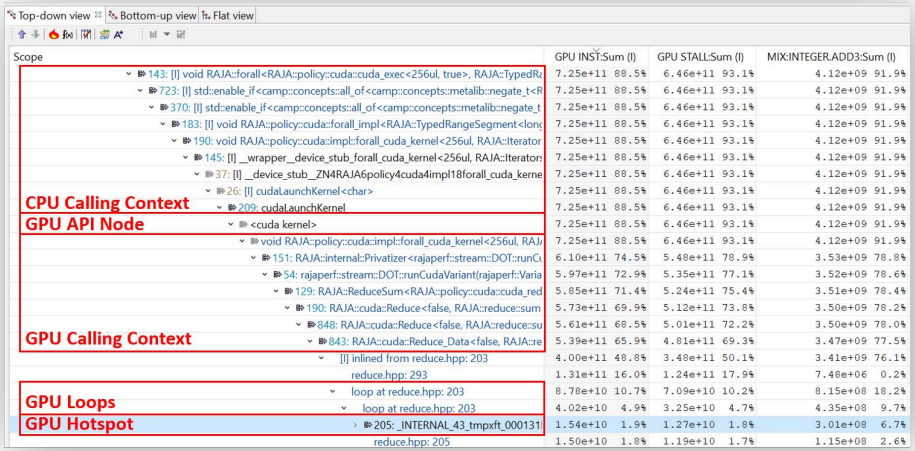
- Memoize common call path prefixes
  - Temporally-adjacent samples in often share common call path prefixes
  - Employ eager (mark bits) or lazy (tramopoline) marking to identify LCA of stack unwinds
- Avoid costly access to mutable concurrent data
  - Cache unwinding recipes in a per thread hash table
- Avoid duplicate unwinds
  - Filter CUDA Driver APIs within CUDA Runtime APIs

Approximation of GPU Calling Contexts to Understand Performance

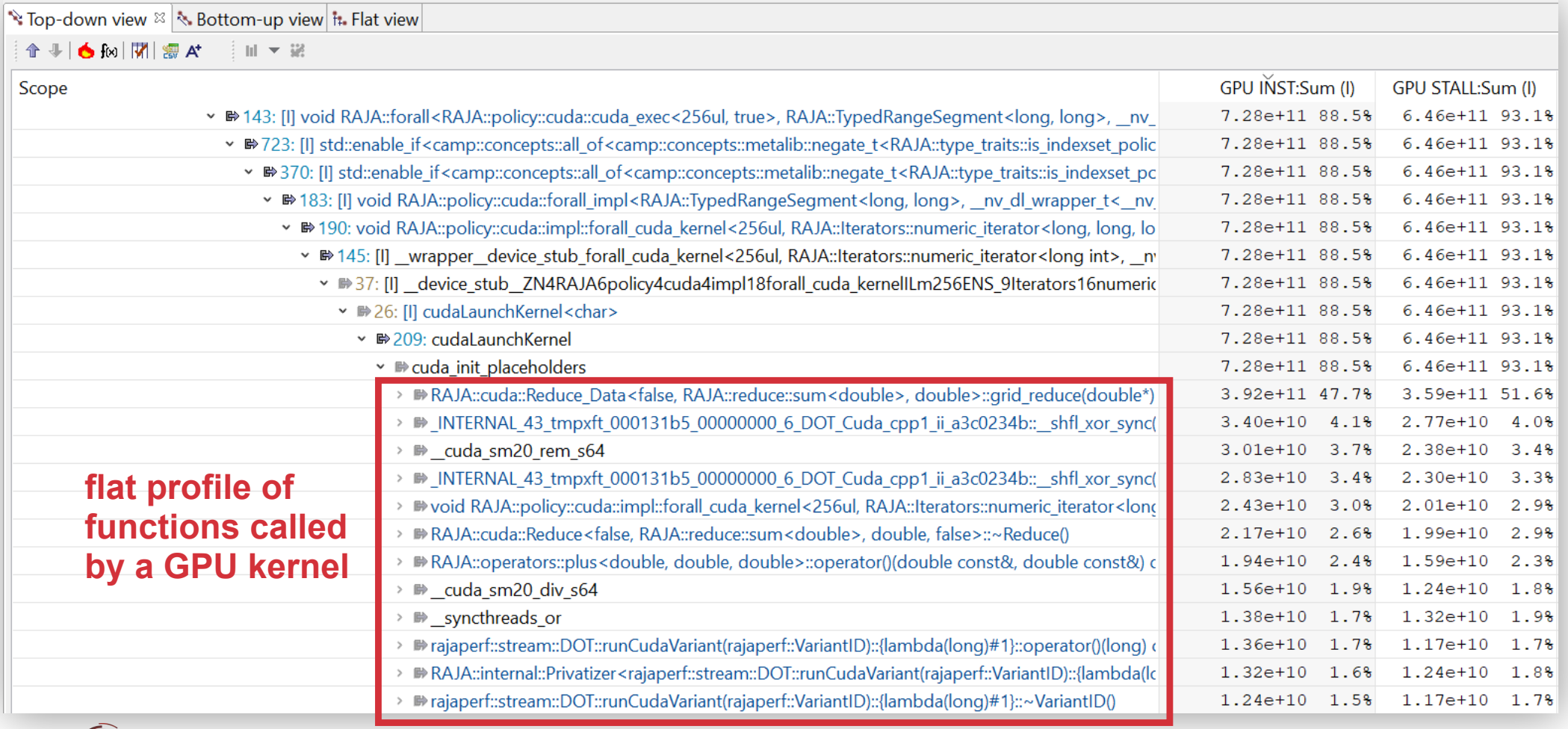
- GPU code from C++ template-based programming models is complex
- NVIDIA GPUs collect flat PC samples
- Flat profiles for instantiations of complex C++ templates are inscrutable

- HPCToolkit reconstructs approximate GPU calling contexts
  - Reconstruct call graph from machine code
  - Infer calls at call sites
    - PC samples of call instructions indicate calls
      - Use call counts to apportion costs to call sites
    - PC samples in a routine





Approximation of GPU Calling Contexts to Understand Performance

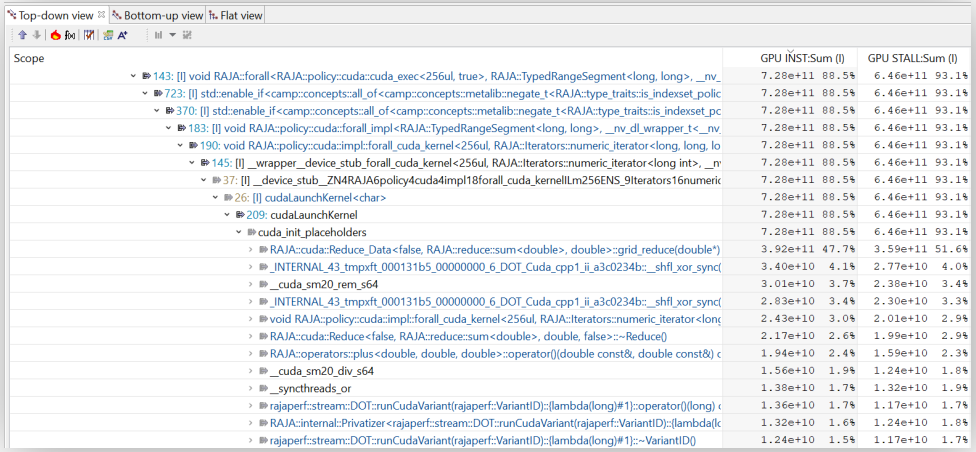


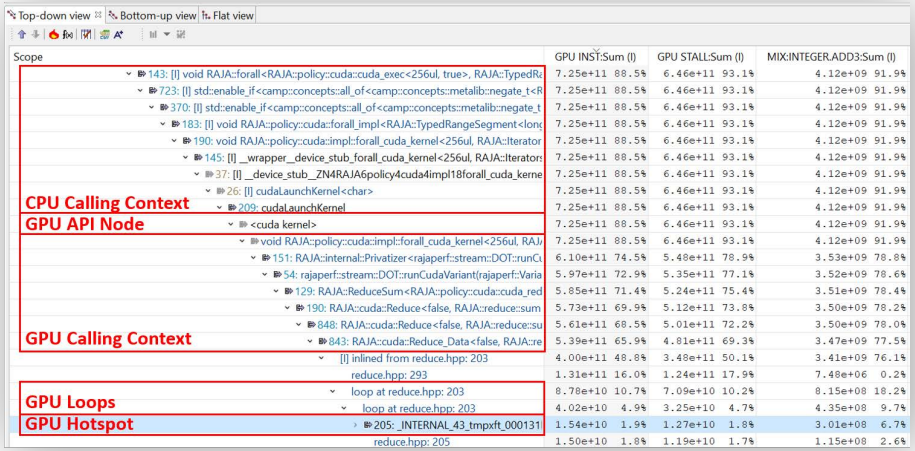
flat profile of functions called by a GPU kernel

Approximation of GPU Calling Contexts to Understand Performance

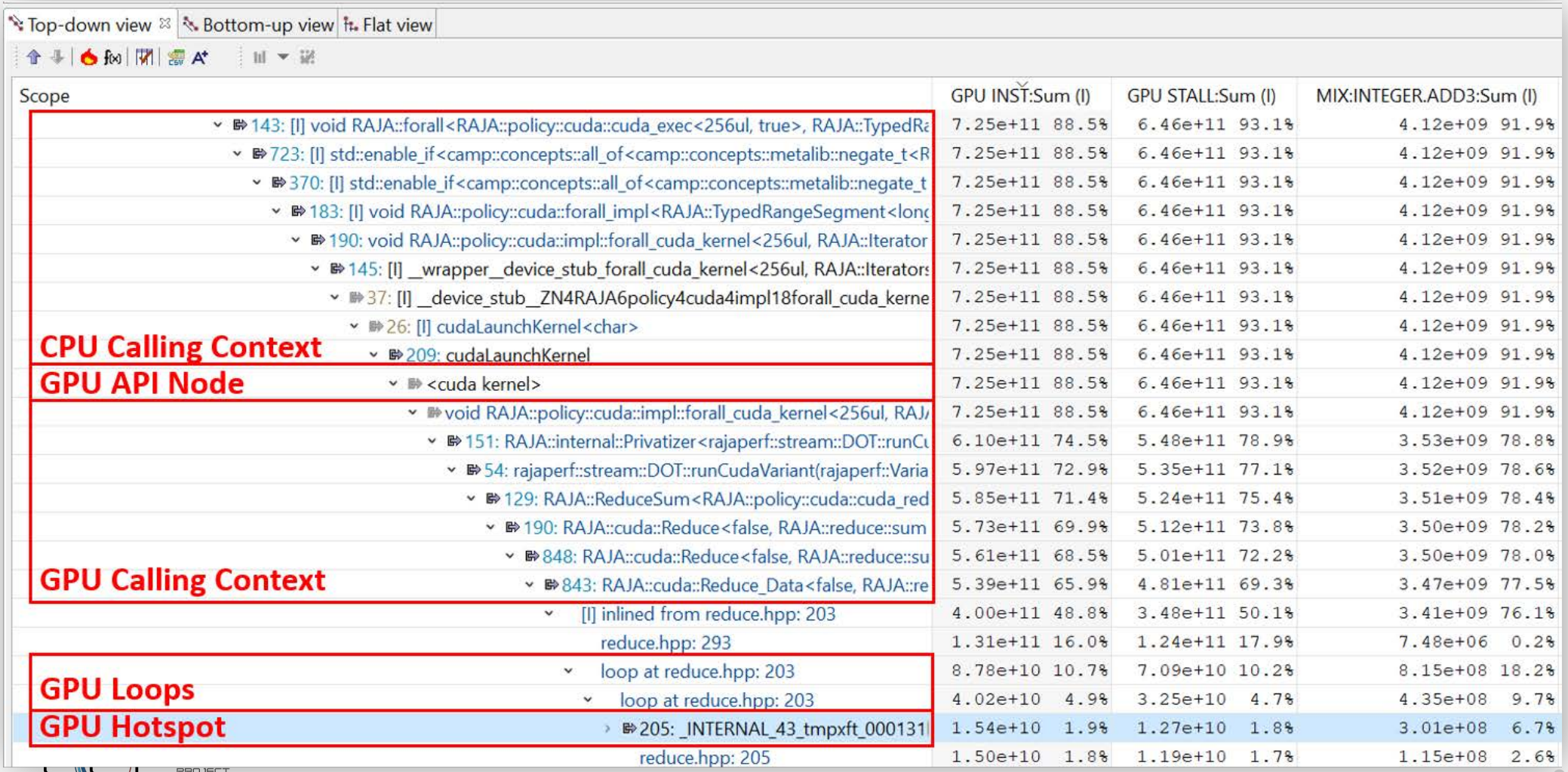
- GPU code from C++ template-based programming models is complex
- NVIDIA GPUs collect flat PC samples
- Flat profiles for instantiations of complex C++ templates are inscrutable

- HPCToolkit reconstructs approximate GPU calling contexts
  - Reconstruct call graph from machine code
  - Infer calls at call sites
    - PC samples of call instructions indicate calls
      - Use call counts to apportion costs to call sites
    - PC samples in a routine





Approximation of GPU Calling Contexts to Understand Performance



Approximate Reconstruction of GPU Calling Context Trees

- **Problem**
  - Unwinding call stacks on GPU is costly for each GPU thread
  - NVIDIA's CUPTI does not provide an unwinding API
- **Challenges**
  - GPU functions may be invoked from different call sites
  - Need to decide how to attribute costs to each call site
- **Solution**
  - Reconstruct GPU calling context tree from flat instruction samples and static GPU call graph

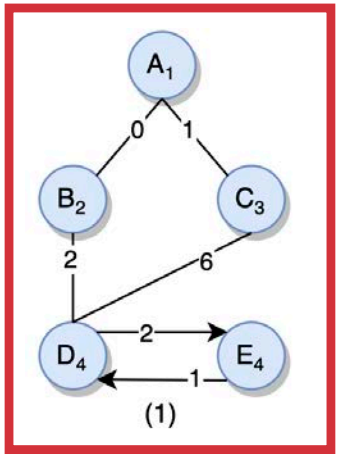
Approximate Reconstruction of GPU Calling Context Trees

1. Construct a GPU static call graph based on functions and call instructions. Initialize call counts using counts or samples of call instructions.

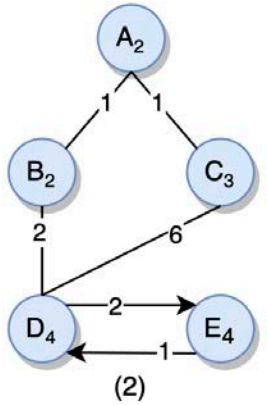
2. *For call graphs based on samples:* if a function has samples and no incoming call edge has a non-zero weight, assign each of its incoming call edges a weight of 1; repeat for call edges of callers until at least one incoming call edge has samples.

3. Identify strongly connected components (SCCs) using Tarjan's algorithm. Rewire call graph, removing SCC internal structure and linking external calls to SCC.

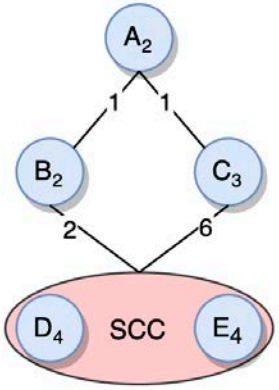
4. Build CCT by splitting call graph. Like gprof, assume that every call to a function has equal cost. Apportion costs of each function among its call sites according to ratios of calls from each call site.



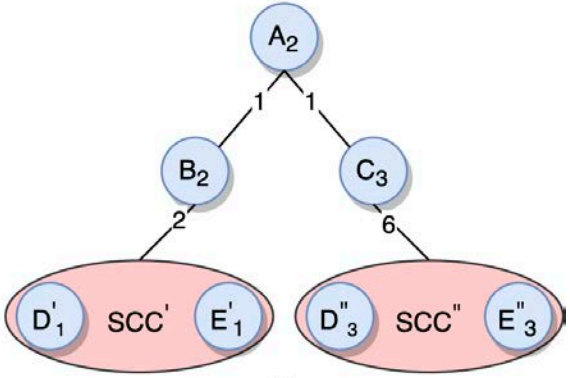
(1)



(2)



(3)



(4)

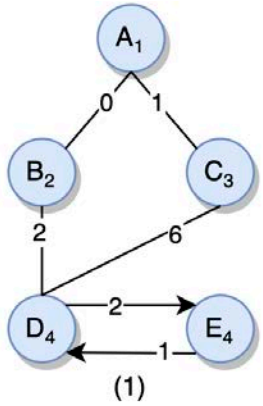
Approximate Reconstruction of GPU Calling Context Trees

1. Construct a GPU static call graph based on functions and call instructions. Initialize call counts using counts or samples of call instructions.

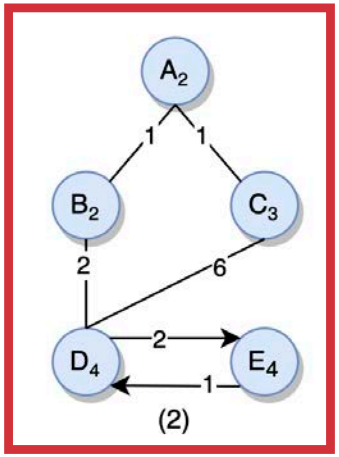
2. *For call graphs based on samples:* if a function has samples and no incoming call edge has a non-zero weight, assign each of its incoming call edges a weight of 1; repeat for call edges of callers until at least one incoming call edge has samples.

3. Identify strongly connected components (SCCs) using Tarjan's algorithm. Rewire call graph, removing SCC internal structure and linking external calls to SCC.

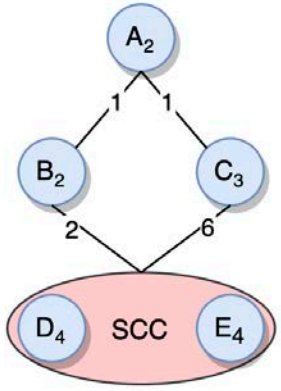
4. Build CCT by splitting call graph. Like gprof, assume that every call to a function has equal cost. Apportion costs of each function among its call sites according to ratios of calls from each call site.



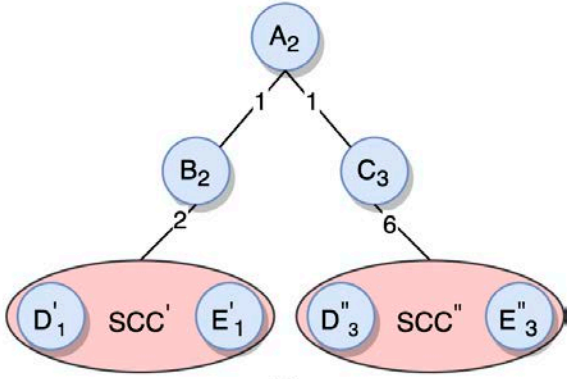
(1)



(2)



(3)



(4)

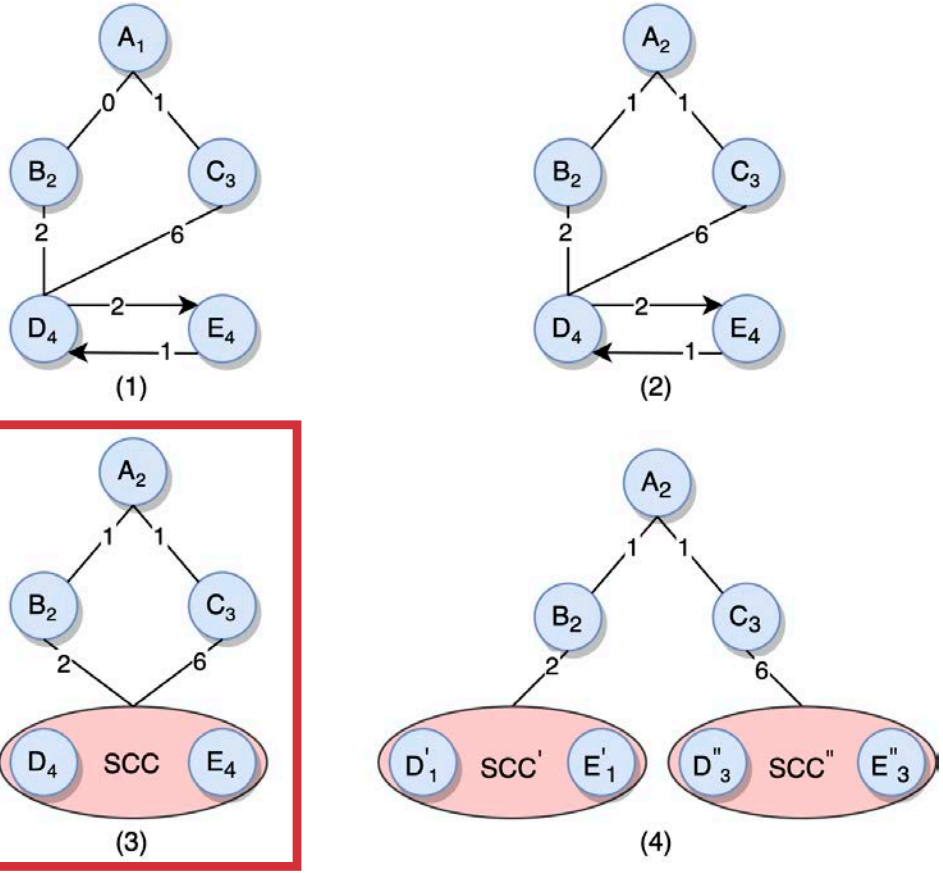
Approximate Reconstruction of GPU Calling Context Trees

1. Construct a GPU static call graph based on functions and call instructions. Initialize call counts using counts or samples of call instructions.

2. *For call graphs based on samples:* if a function has samples and no incoming call edge has a non-zero weight, assign each of its incoming call edges a weight of 1; repeat for call edges of callers until at least one incoming call edge has samples.

3. Identify strongly connected components (SCCs) using Tarjan's algorithm. Rewire call graph, removing SCC internal structure and linking external calls to SCC.

4. Build CCT by splitting call graph. Like gprof, assume that every call to a function has equal cost. Apportion costs of each function among its call sites according to ratios of calls from each call site.



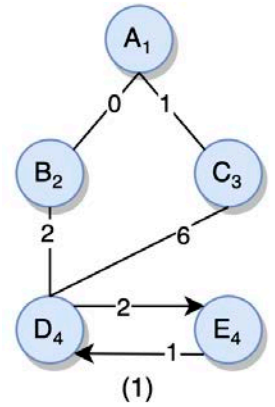
Approximate Reconstruction of GPU Calling Context Trees

1. Construct a GPU static call graph based on functions and call instructions. Initialize call counts using counts or samples of call instructions.

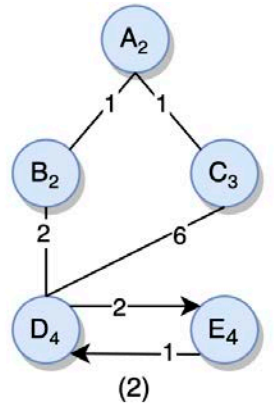
2. *For call graphs based on samples:* if a function has samples and no incoming call edge has a non-zero weight, assign each of its incoming call edges a weight of 1; repeat for call edges of callers until at least one incoming call edge has samples.

3. Identify strongly connected components (SCCs) using Tarjan's algorithm. Rewire call graph, removing SCC internal structure and linking external calls to SCC.

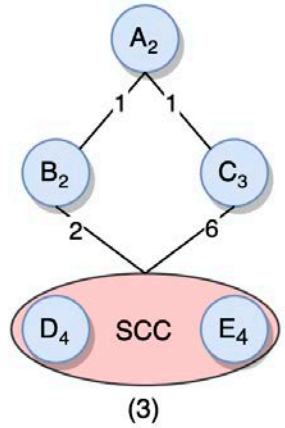
4. Build CCT by splitting call graph. Like gprof, assume that every call to a function has equal cost. Apportion costs of each function among its call sites according to ratios of calls from each call site.



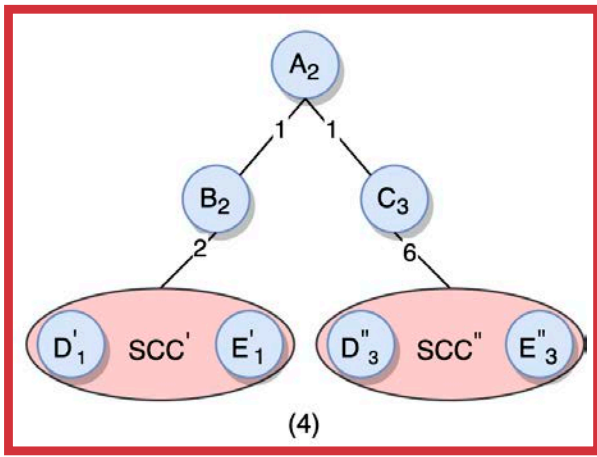
(1)



(2)



(3)

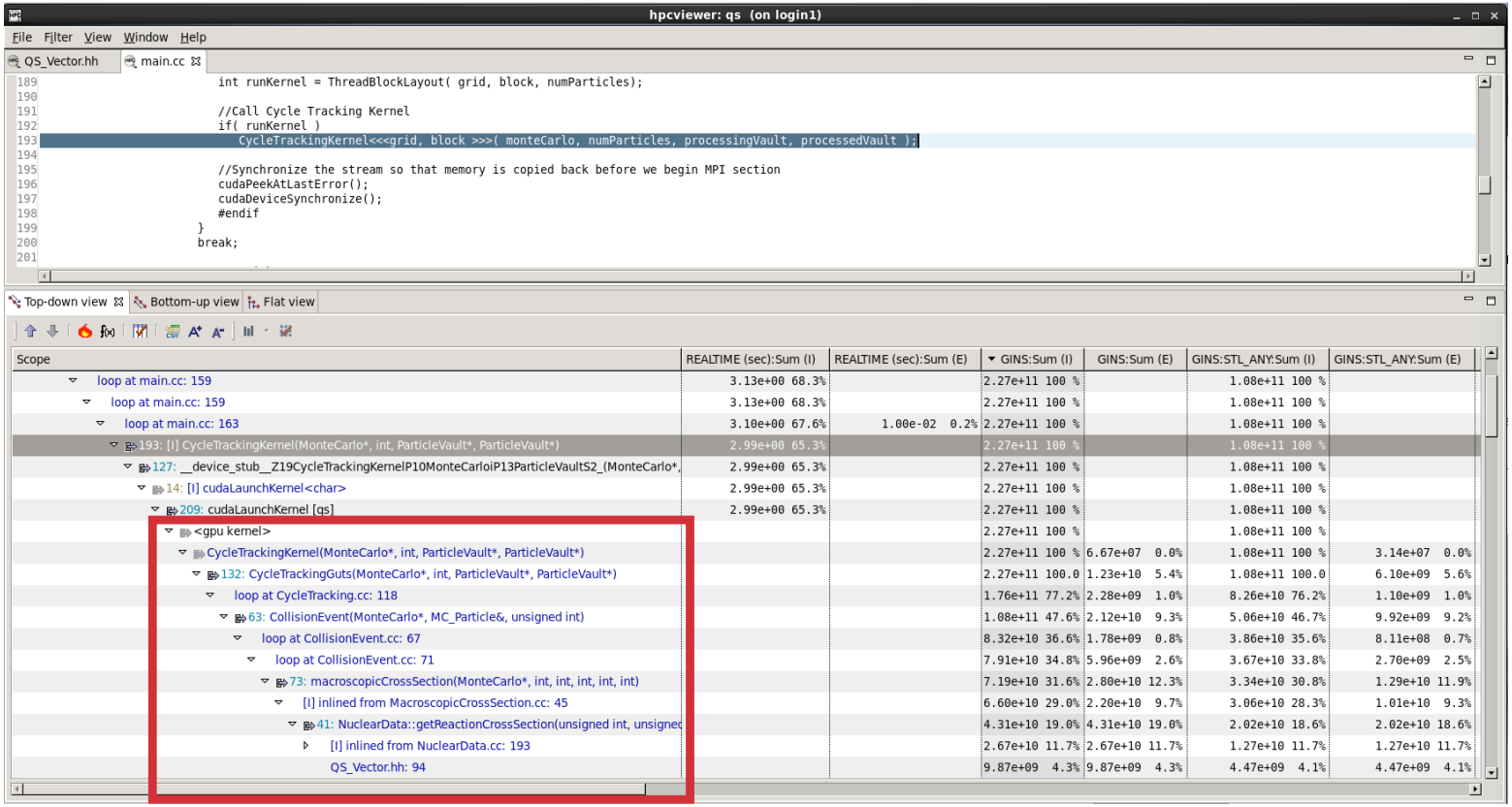


(4)

ECP Quicksilver GPU Proxy Application: Detailed Profile View

Compute Node
- 2 Power9
- 6xNVIDIA GPU

- **Optimized (-O2) compilation with nvcc**
- **1 GPU stream**
- **Detailed measurement and attribution using PC sampling**
- **Reconstruct approximate call graph on GPU from flat PC samples**
- **Attribute information to heterogeneous calling context including**
  - CPU calling context
  - GPU kernel
  - GPU calling context
  - GPU loops
  - GPU statements
- **Metrics**
  - instructions executed
  - instruction stalls and reasons
  - GPU utilization

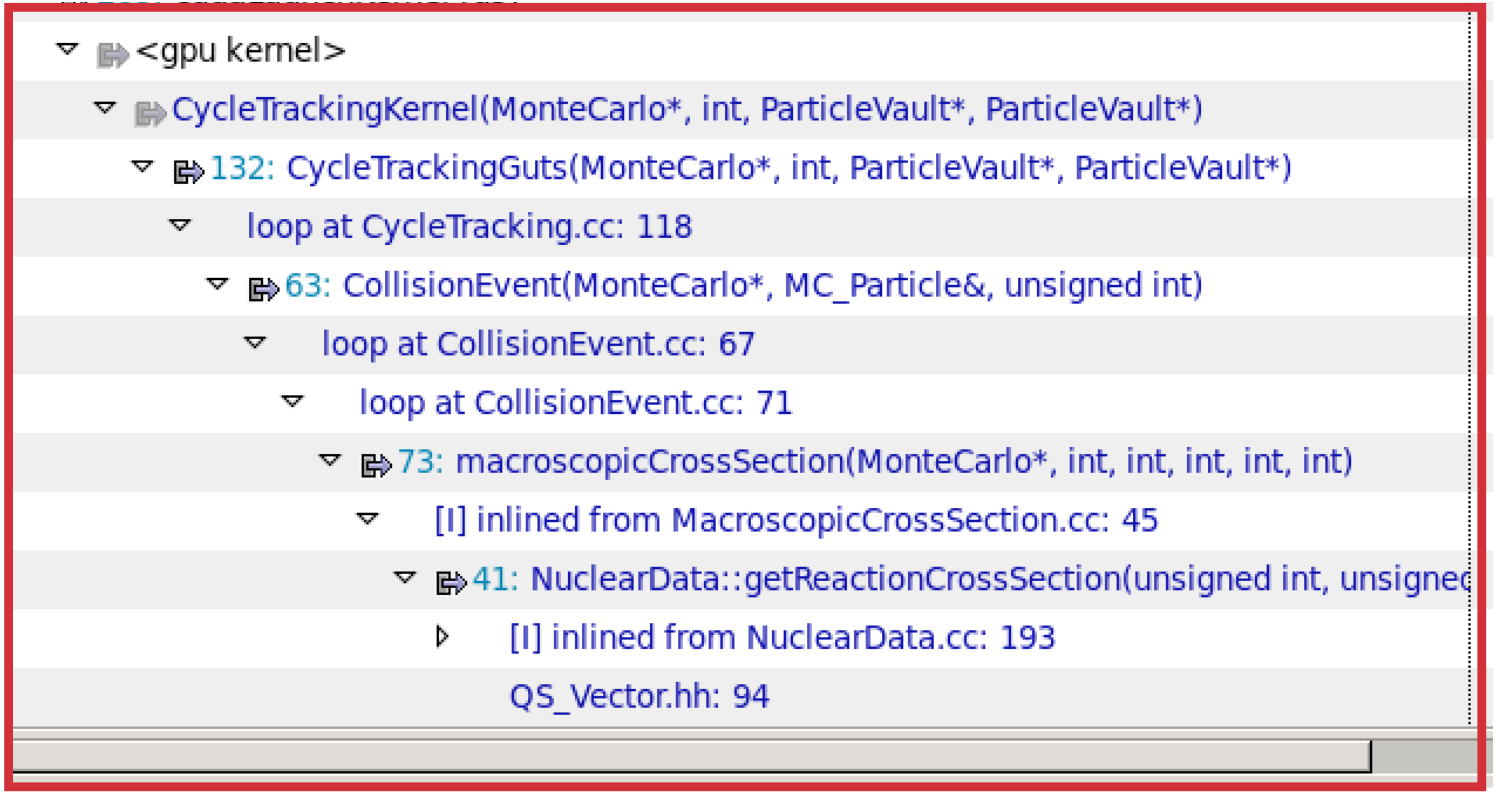


hpctoolkit-tutorial-examples/examples/gpu/quicksilver

Quicksilver GPU CUDA Example: Detailed Profile View

- **Optimized (-O2) compilation with nvcc**
- **1 GPU stream**
- **Detailed measurement and attribution using PC sampling**
- **Reconstruct approximate call graph on GPU from flat PC samples**
- **Attribute information to heterogeneous calling context including**
  - CPU calling context
  - GPU kernel
  - GPU calling context
  - GPU loops
  - GPU statements
- **Metrics**
  - instructions executed
  - instruction stalls and reasons
  - GPU utilization

Detailed Attribution on GPUs

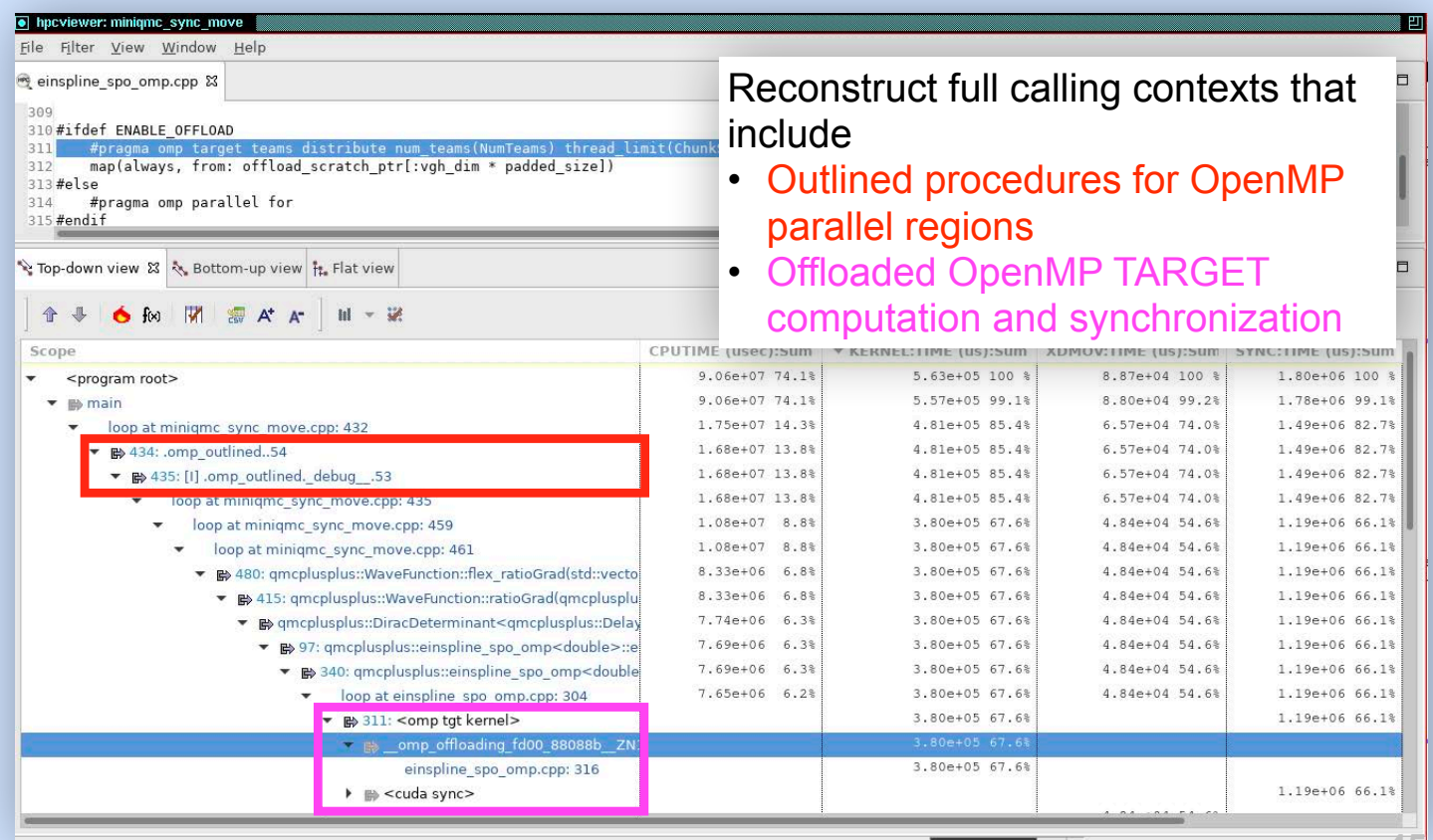


hpctoolkit-tutorial-examples/examples/gpu/quicksilver

Support for OpenMP TARGET

- HPCToolkit implementation of OMPT OpenMP API
  - host monitoring
    - leverages callbacks for regions, threads, tasks
    - employs OMPT API for call stack introspection
  - GPU monitoring
    - leverages callbacks for device initialization, kernel launch, data operations
  - reconstruction of user-level calling contexts
- Leverages implementation of OMPT in LLVM OpenMP and libomptarget

ECP QMCPACK Project: miniqmc using OpenMP TARGET (Power9 + NVIDIA V100)



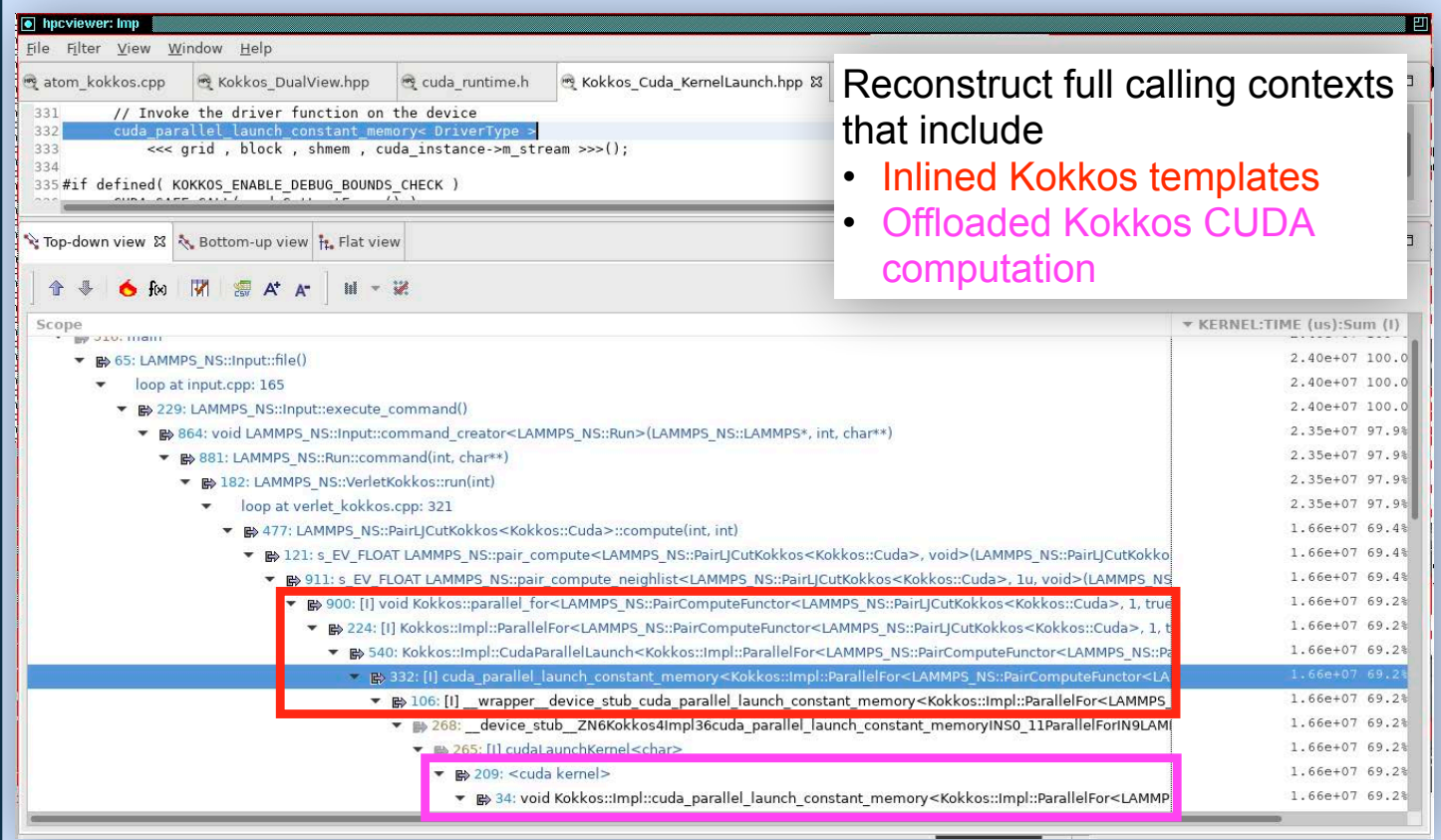
Reconstruct full calling contexts that include

- Outlined procedures for OpenMP parallel regions
- Offloaded OpenMP TARGET computation and synchronization

Support for RAJA and and Kokkos C++ Template-based Models

- RAJA and Kokkos provide portability layers atop C++ template-based programming abstractions
- HPCToolkit employs binary analysis to recover information about procedures, inlined functions and templates, and loops
  - Enables both developers and users to understand complex template instantiation present with these models

ECP EXAALT Project: LAMMPS using Kokkos over CUDA (Power9 + NVIDIA V100)

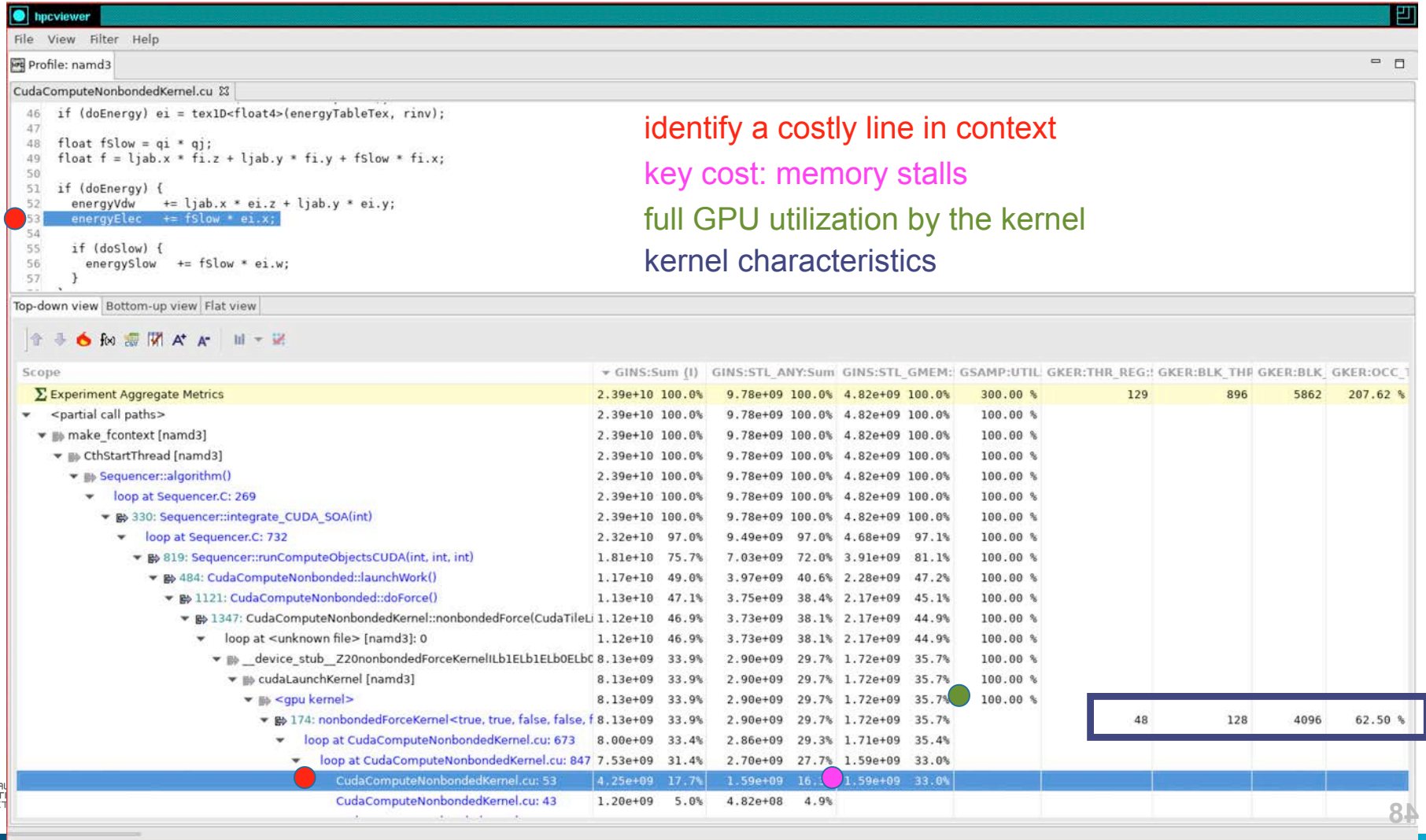


Reconstruct full calling contexts that include
- Inlined Kokkos templates
- Offloaded Kokkos CUDA computation

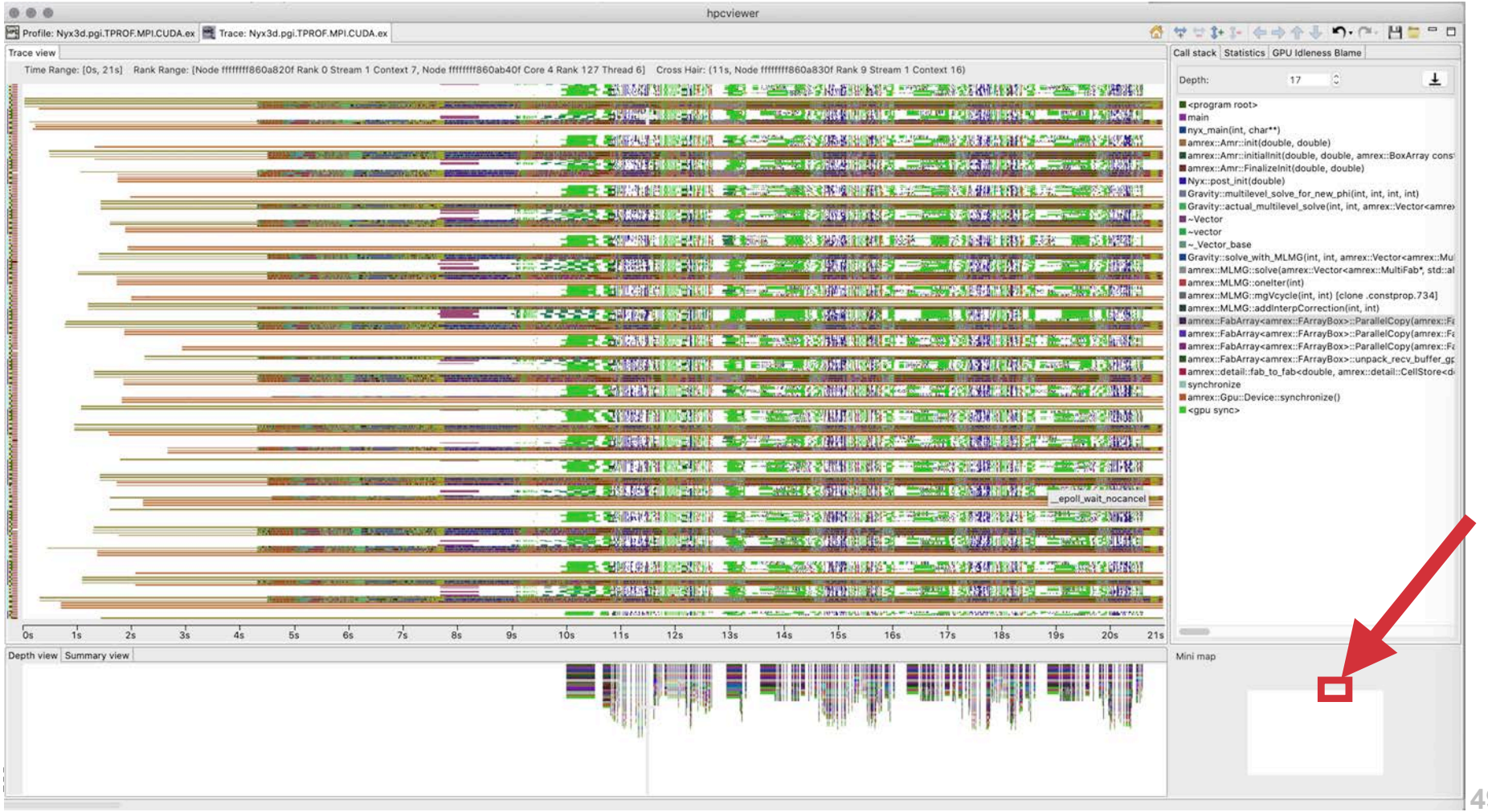
Deriving GPU Metrics

- **Problem**
  - GPU PC sampling cannot be used in the same pass with metric collection
  - Nsight-compute runs nine passes to collect multiple metrics for kernels
- **Our approach**
  - Derive multiple metrics using PC sampling and other activity records
    - e.g., GPU SM utilization, GPU occupancy, …

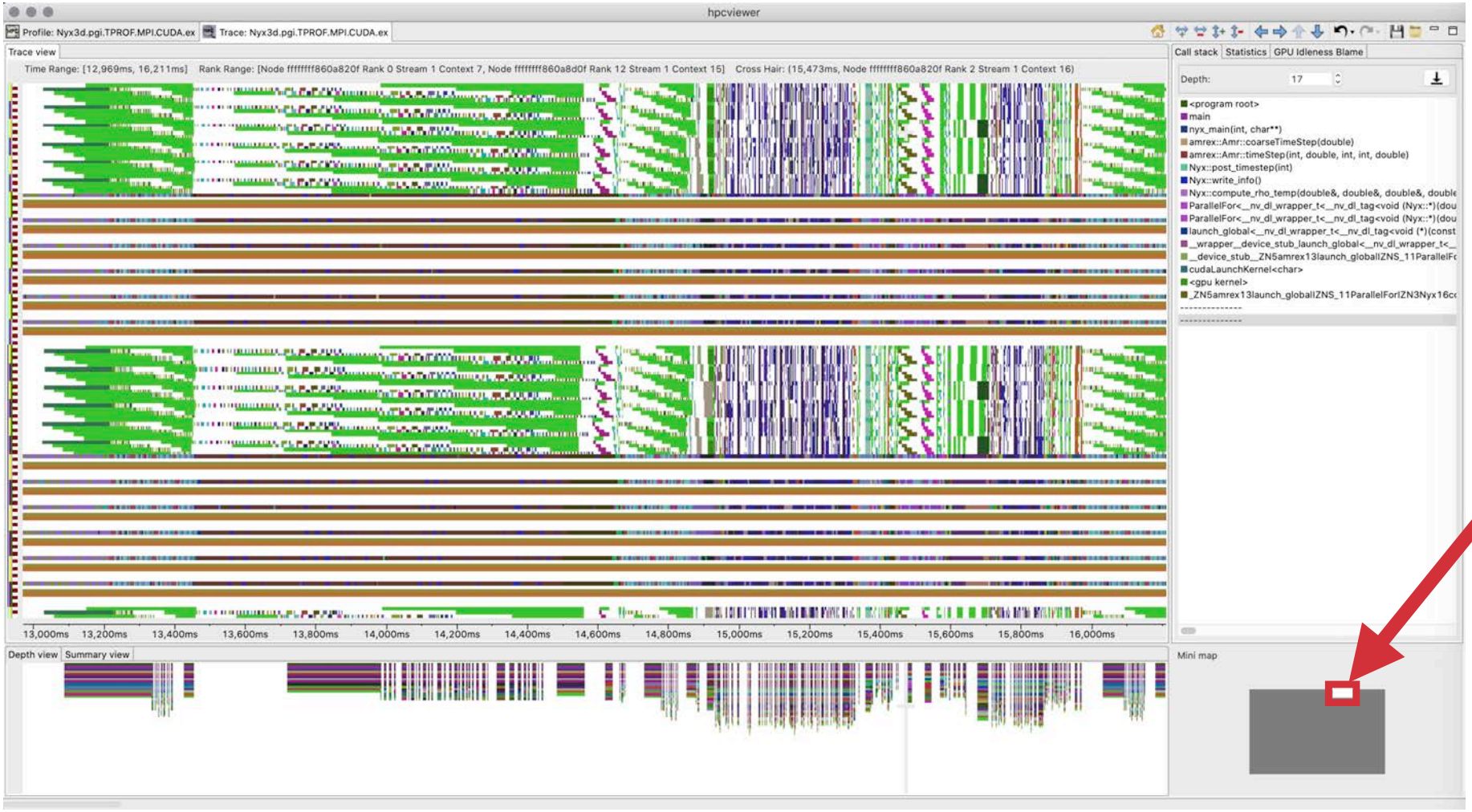
Measurement of GPU-accelerated NAMD3 using Charm++



Nyx with CUDA: Trace of Multi-rank Multi-GPU Executions

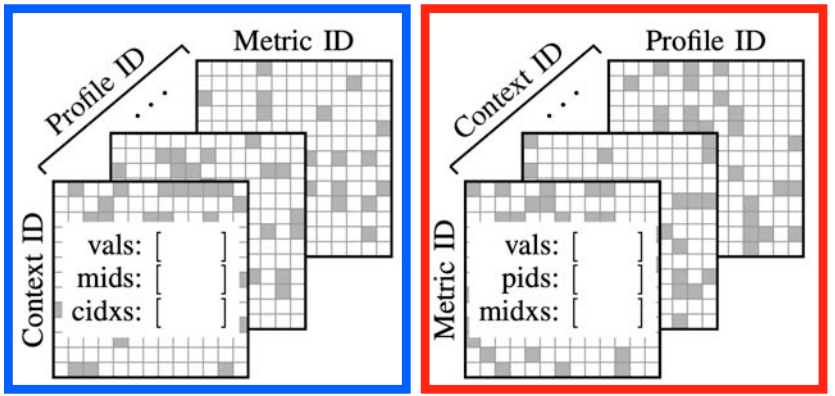


Nyx with CUDA: Trace of Multi-rank Multi-GPU Executions



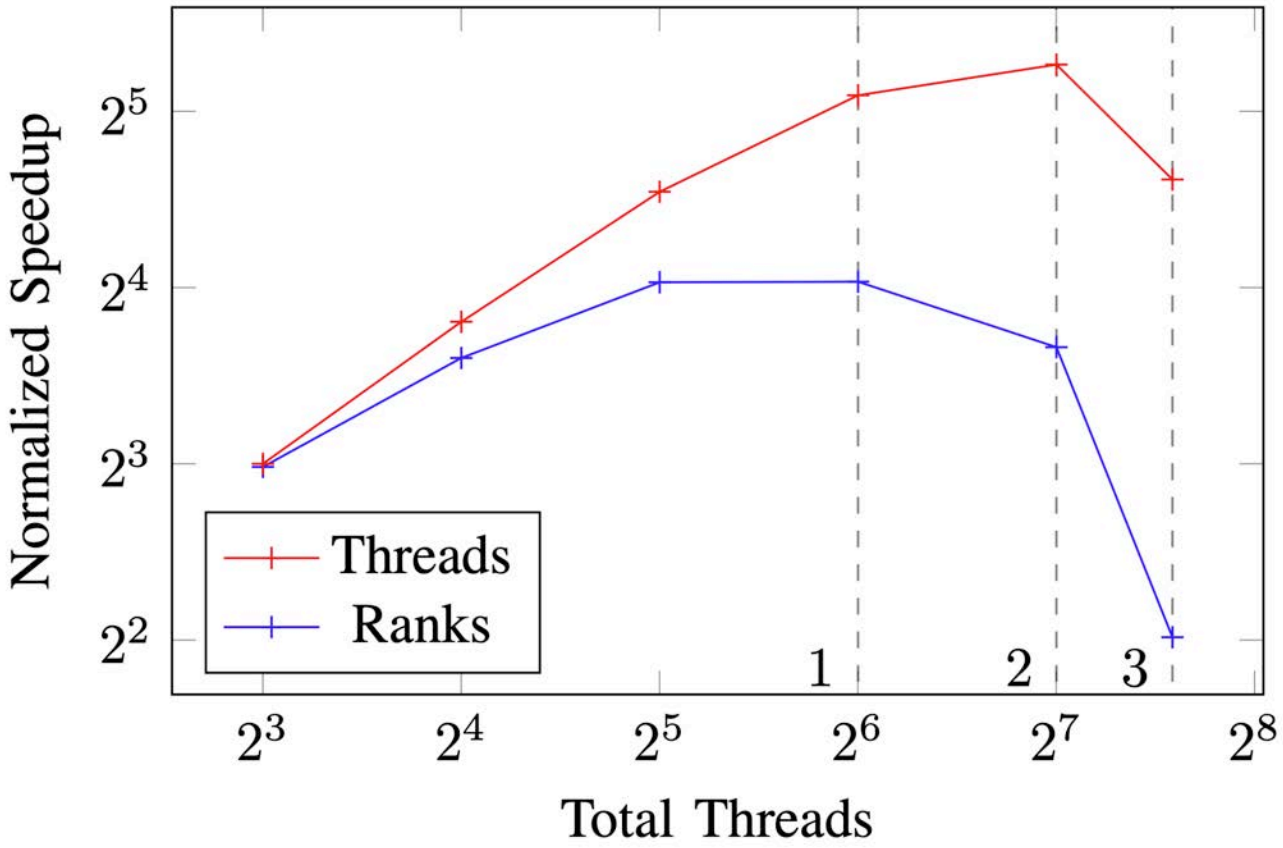
Scalable Analysis of Performance Data

- **When to reduce profile data?**
  - After termination: Linux perf, NVIDIA nvvp, and Paraver record detailed traces
  - At termination
    - Scalasca, Tau, Vampir use MPI to unify profile data into CUBE format
    - HPCToolkit saves separate profiles and traces per thread
- **Scalable analysis of performance data using out-of-core algorithms**
  - Inspect profiles and balance across ranks by aggregate size
  - Unify call stacks from all threads
  - Overlay static information on calling context trees: procedures, inline functions, loops, stmts
  - Generate computed statistics: aggregate and per profile
  - Write out two sparse outputs
    - profile-major-sparse database
    - calling-context-major-sparse database
  - Implementation: MPI + OpenMP



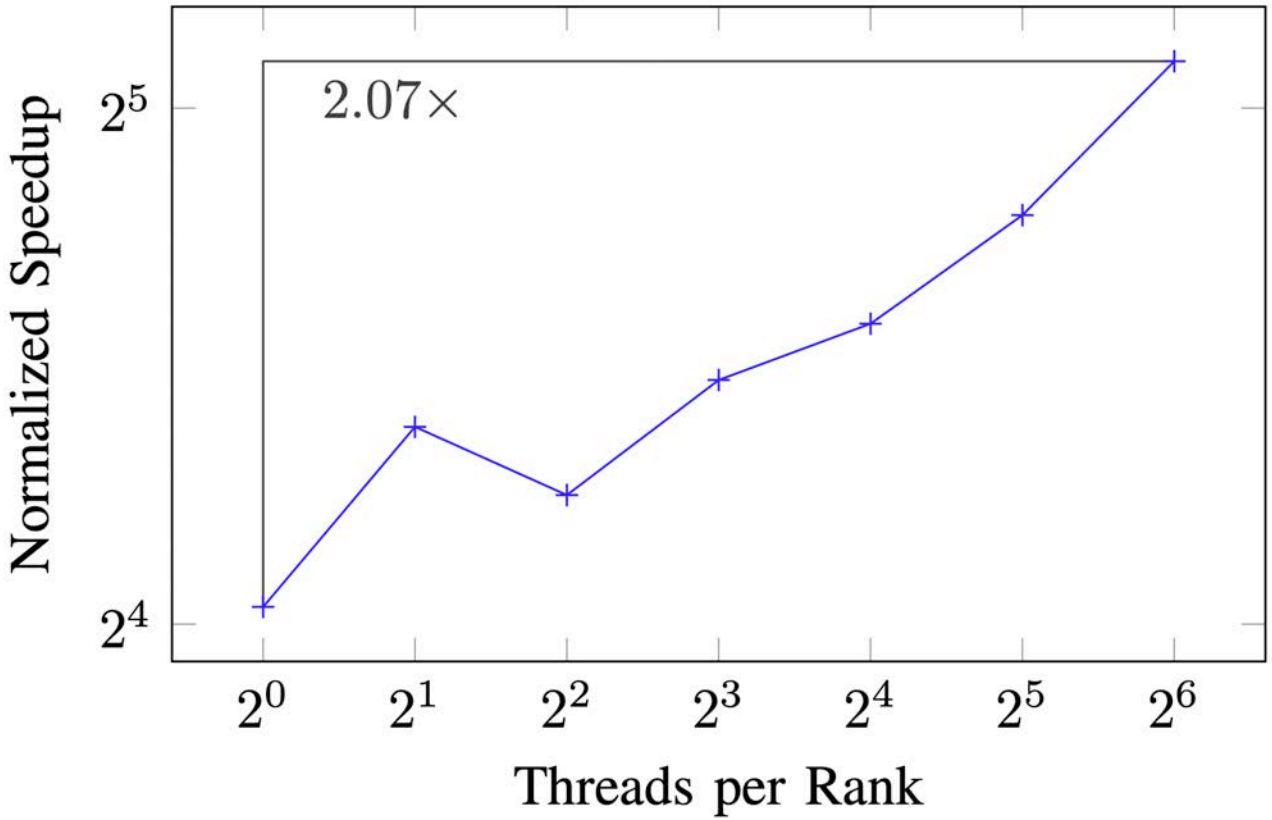
Scalable Analysis of Performance Data: MPI vs. OpenMP

- **Compare relative speedup of all MPI ranks vs. all OpenMP threads**
  - Input: 8K profiles
  - use all OpenMP threads at various scales
  - use all MPI ranks at various scales
- **Findings**
  - OpenMP has superior scaling and speedup
  - MPI speedup degrades using more than one thread per core
  - OpenMP speedup improves from 64 to 128 threads
  - Note: need 2+ threads per core to use all memory B/W on KNL



Scalable Analysis of Performance Data: Threads vs. Ranks

- **Use 64 cores and threads**
- **Compare balance of ranks vs. threads**
  - 64 ranks
  - 32 ranks, 2 threads/rank
  - …
  - 1 rank, 64 threads/rank
- **Findings**
  - Performance increases by trading threads for ranks



Scalable Analysis of Performance Data: Value of Sparsity

- **Assess the space savings of sparse profiles**
  - AMD2006 CPU
    - 1 metric
    - 9 metrics, including some rare metrics
  - Nyx GPU
  - LAMMPS GPU
- **Findings**
  - as much as 21x space reduction for measurements
  - as much 337x reduction for output data

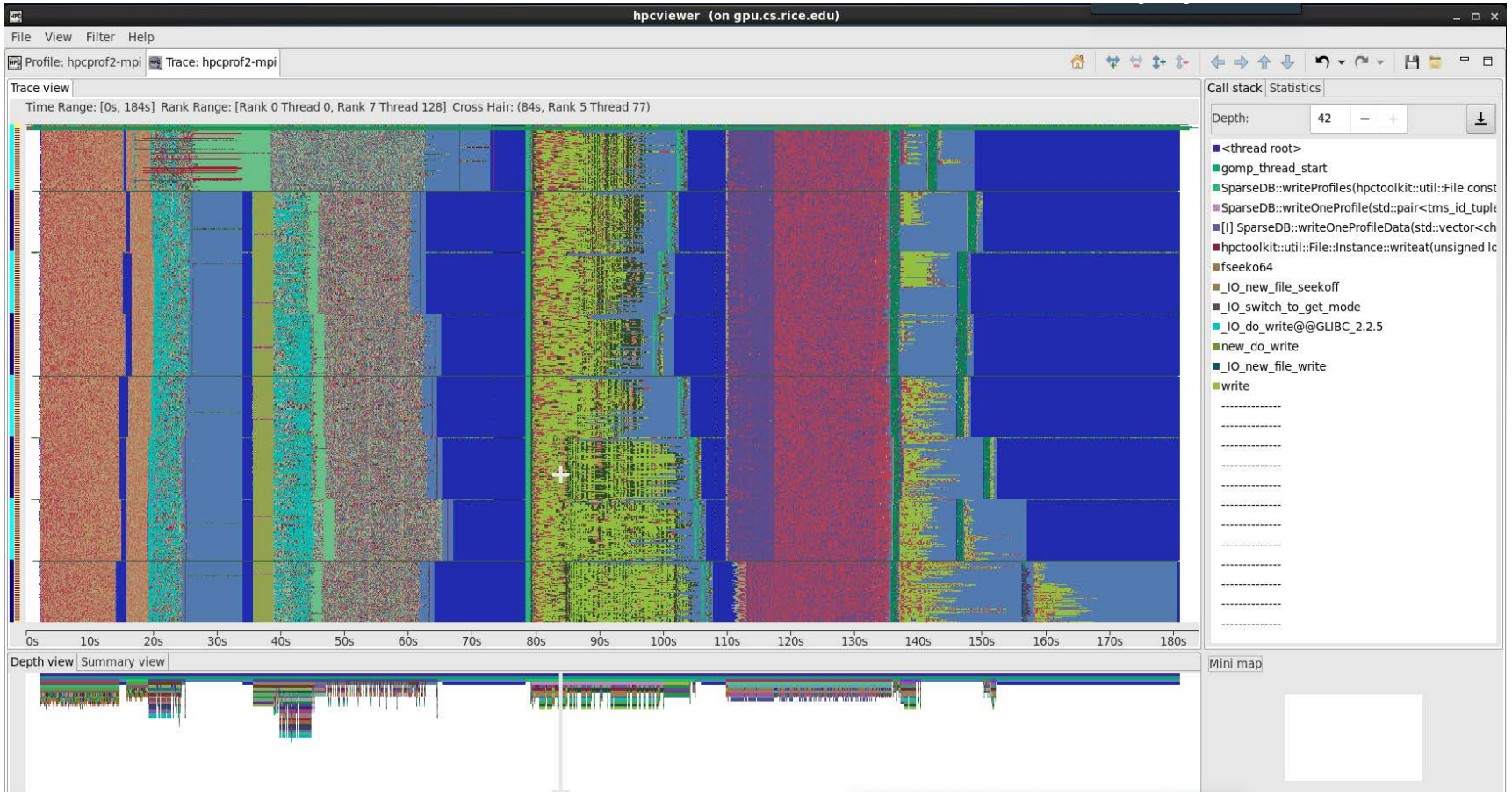
Dataset		Size (MiB) Dense	Sparse	Ratio
AMG2006 (1)	M	659.0	911.0	0.723×
	D	7370.0	836.0	8.819×
AMG2006 (9)	M	21.7	11.1	1.956×
	D	2290.0	33.5	68.34 ×
Nyx	M	5890.0	278.0	21.14 ×
	D	130 GiB	601.0	221.5 ×
LAMMPS	M	85.5	5.23	16.35 ×
	D	8250.0	24.5	336.9 ×

Scalable Analysis of Performance Data: 64K profiles of AMG2006

Input
- 5GB profiles
- 225GB traces

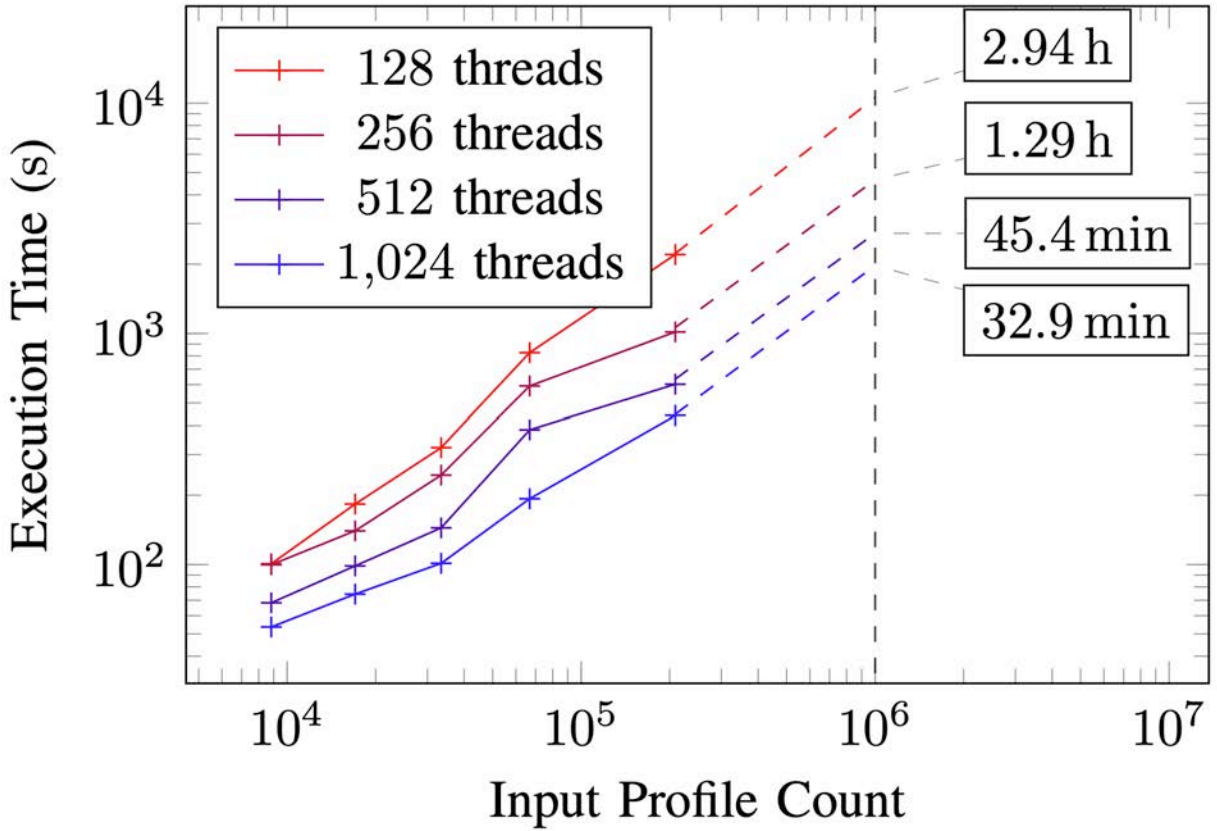
Analysis
- 8 KNL nodes
- 1 rank / node
- 128T / rank

Execution time
- 184s



Scalable Analysis of Performance Data

- **Assess scalability vs. profile counts**
- **Explore scaling using 128-1024 threads**
  - 128 threads per rank
- **Extrapolate to 1M profiles**
- **Findings**
  - Good scaling with number of profiles
  - Relative efficiency drops to 67% when scaling from 128 to 1024 threads
    - 200K profiles (9GB total) is a small problem for 1024 threads



Outline

- **Performance measurement and analysis challenges for GPU-accelerated supercomputers**
- **Introduction to HPCToolkit performance tools**
  - Overview of HPCToolkit components and their workflow
  - HPCToolkit's graphical user interfaces
- **Analyzing the performance of GPU-accelerated supercomputers with HPCToolkit**
  - Overview of HPCToolkit's GPU performance measurement capabilities
  - Collecting measurements
  - Analysis and attribution
  - Scalable analysis of performance data
- **Status, ongoing work, final remarks**

Status for Various GPUs

Vendor	Coarse-grain measurement	Fine-grain measurement	Tracing	Binary analysis: loops, inlined code
NVIDIA	CUPTI	PC sampling	CUPTI	nvdisasm + dyninst
Intel	OpenCL and Level 0	GTPin instrumentation	OpenCL callbacks	IGA + dyninst
AMD	Roctracer	—-	Roctracer	dyninst + emerging native decoder

Many of these capabilities are on different branches

Ongoing Work

- **Interface**
  - Emerging GPU Performance Advisor tool for NVIDIA GPUs
    - attributes instruction stalls with backward slicing, analyzes code, offers advice about most promising improvements
  - New integrated user interface supports both profiles and traces
    - Automated serialization analysis of CPU and GPU traces in hpctraceviewer GUI
  - Performance analysis of machine learning frameworks: Pytorch, Tensorflow
- **Internals**
  - Refinement of implementation atop Intel's Level 0
  - Binary analysis of AMD GPU instructions
  - Refining scalable aggregation

Final Remarks

- **Nice to work with national labs and have early involvement in a big procurement**
  - Amplifies our ability to affect vendor hardware and software in the near term
- **Software development challenges are myriad**
  - Developing tools for three GPU software stacks at the same time is ridiculous
  - Building capabilities ahead of current vendor hardware and software
  - AMD and Intel software is a work in progress
    - Intel: unstable with significant flaws
    - Both: API-breaking changes are common
  - Relying on vendor closed-source components is a challenge
    - standards specify only an API, but internals matter for tools that see all
    - undocumented behaviors about things that matter
    - missing capabilities, e.g. need excellent DWARF mappings for optimized GPU code
    - NVIDIA serializes kernels to facilitate measurement with PC sampling

Acknowledgments

- **Current funding**
  - DOE Exascale Computing Project (Subcontract 4000151982)
  - NSF Software Infrastructure for Sustained Innovation (Collaborative Agreement 1450273)
  - DOE Labs: ANL (Subcontract 9F-60073), Tri-labs (LLNL Subcontract B633244)
  - Industry: AMD
- **Team**
  - Rice University
    - HPCToolkit PI: Prof. John Mellor-Crummey
    - Research staff: Laksono Adhianto, Mark Krentel, Xiaozhu Meng, Scott Warren
    - Contractor: Marty Itzkowitz
    - Students: Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Yumeng Liu, Keren Zhou
    - Recent summer interns: Vladimir Indjic, Tijana Jovanovic, Aleksa Simovic
  - University of Wisconsin – Madison
    - Dyninst PI: Prof. Barton Miller