Using HPCToolkit to Measure and Analyze the Performance of GPU-accelerated Applications

EXASCALE COMPUTING PROJECT

John Mellor-Crummey
Rice University

Tutorial
Mar-Apr 2021
NERSC and OLCF (Virtual)

National Nuclear Security Administration

U.S. DEPARTMENT OF ENERGY | Office of Science

RICE

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

Acknowledgments

- **Current funding**
  - DOE Exascale Computing Project (Subcontract 4000151982)
  - DOE Labs: ANL (Subcontract 9F-60073), Tri-labs (LLNL Subcontract B645220)
  - Industry: AMD, Total E&P Research & Technology
- **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

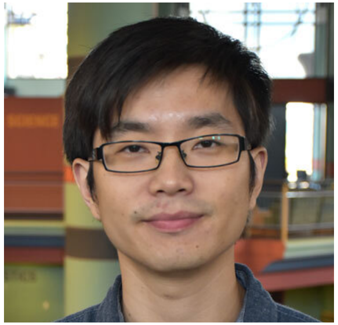
EXASCALE COMPUTING PROJECT

HPCToolkit Team at the Tutorial

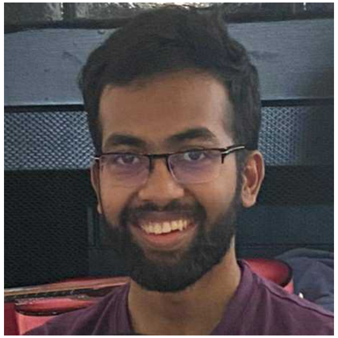
Laksono Adhianto

Mark Krentel

John Mellor-Crummey

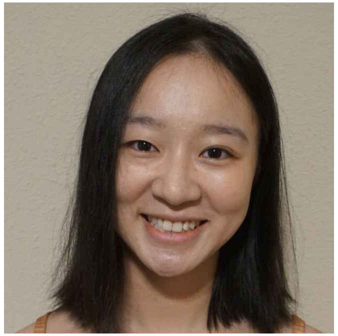


Xiaozhu Meng



Aaron Cherian

Dejan Grubisic

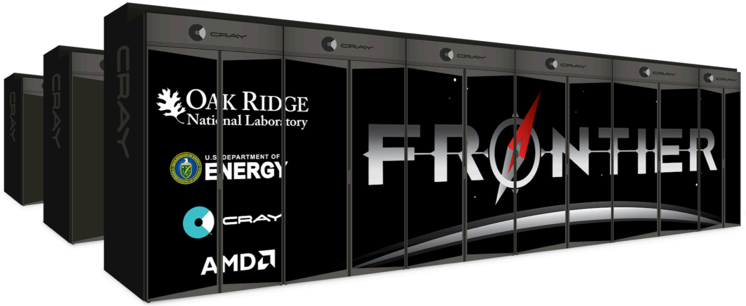


Yumeng Liu

Keren Zhou

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

DOE's NVIDIA-based Heterogeneous Supercomputers

- **Summit compute nodes (OLCF)**
  - 2 IBM Power9 processors
  - 6 NVIDIA V100 GPUs
  - Dual-rail Mellanox EDR Infiniband Fat Tree

- **Sierra compute nodes (LLNL)**
  - 2 IBM Power9 processors
  - 4 NVIDIA V100 GPUs
  - Mellanox EDR Infiniband Fat Tree

- **Perlmutter compute nodes (LBNL)**
  - AMD Milan CPU
  - 4 NVIDIA A100 GPUs
  - Cray Slingshot Interconnect

Node-level Programming Models for Heterogeneous Supercomputers

- **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

  - CUDA
  - 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

Global Programming Models

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

Sources: Frontier Spec Sheet
https://www.olcf.ornl.gov/wp-content/uploads/2019/05/frontier_specsheet.pdf
https://docs.nersc.gov/development/programming-models

Performance Analysis Challenges for GPU-accelerated Supercomputers

- **Myriad performance concerns**
  - Computation: need extreme-scale data parallelism to keep GPUs busy
  - Data movement costs within and between memory spaces
  - Internode communication
  - I/O

- **Many ways to hurt performance**
  - insufficient parallelism, load imbalance, serialization, replicated work, data copies, synchronization, lack of locality, …

- **Hardware and execution model complexity**
  - Multiple compute engines with vastly different characteristics, capabilities, and concerns
  - Multiple memory spaces with different performance characteristics
  - Asynchronous execution

Measurement Challenges for GPU-accelerated Supercomputers

- **Extreme-scale parallelism**
  - Serialization within tools will disrupt parallel performance
- **Dependent on third-party measurement interfaces**
  - Hardware
    - CPU hardware performance monitoring unit
    - GPU hardware counters and PC sampling
  - Software
    - Glibc LD_AUDIT for tracking dynamic loading of shared libraries
    - Linux perf_events for kernel measurement
    - GPU monitoring and instrumentation libraries from vendors
- **Multiple measurement modalities and interfaces**
  - Sampling on the CPU
  - Callbacks when GPU operations are launched and (sometimes) completed
  - GPU event stream, including PC sampling measurements
- **Frequent GPU kernel launches require a low-overhead measurement substrate**

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

Other 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
- **Tools**
  - GPU vendors
    - Nsight Systems, Nsight Compute, nvprof, ROCProfiler, Intel VTune
  - Third party
    - TAU, VampirTrace, ARM Map

Shortcomings of Other Tools for Complex GPU-accelerated Programs

- They lack a comprehensive profile view to analyze
  - CPU calling contexts (including inlined frames) where GPU operations are invoked
    - Understanding where, how and why GPU kernels arose from instantiation of nested templates
    - Understanding costs of GPU APIs (e.g., cudaMemcpy) invoked from many different contexts
  - Sophisticated GPU calling contexts
    - OpenMP Target, Kokkos, and RAJA generate GPU code with many small procedures
  - Loop-level performance information on CPUs and GPUs
- At best, existing tools only attribute runtime cost to a flat profile view of functions executed on GPUs

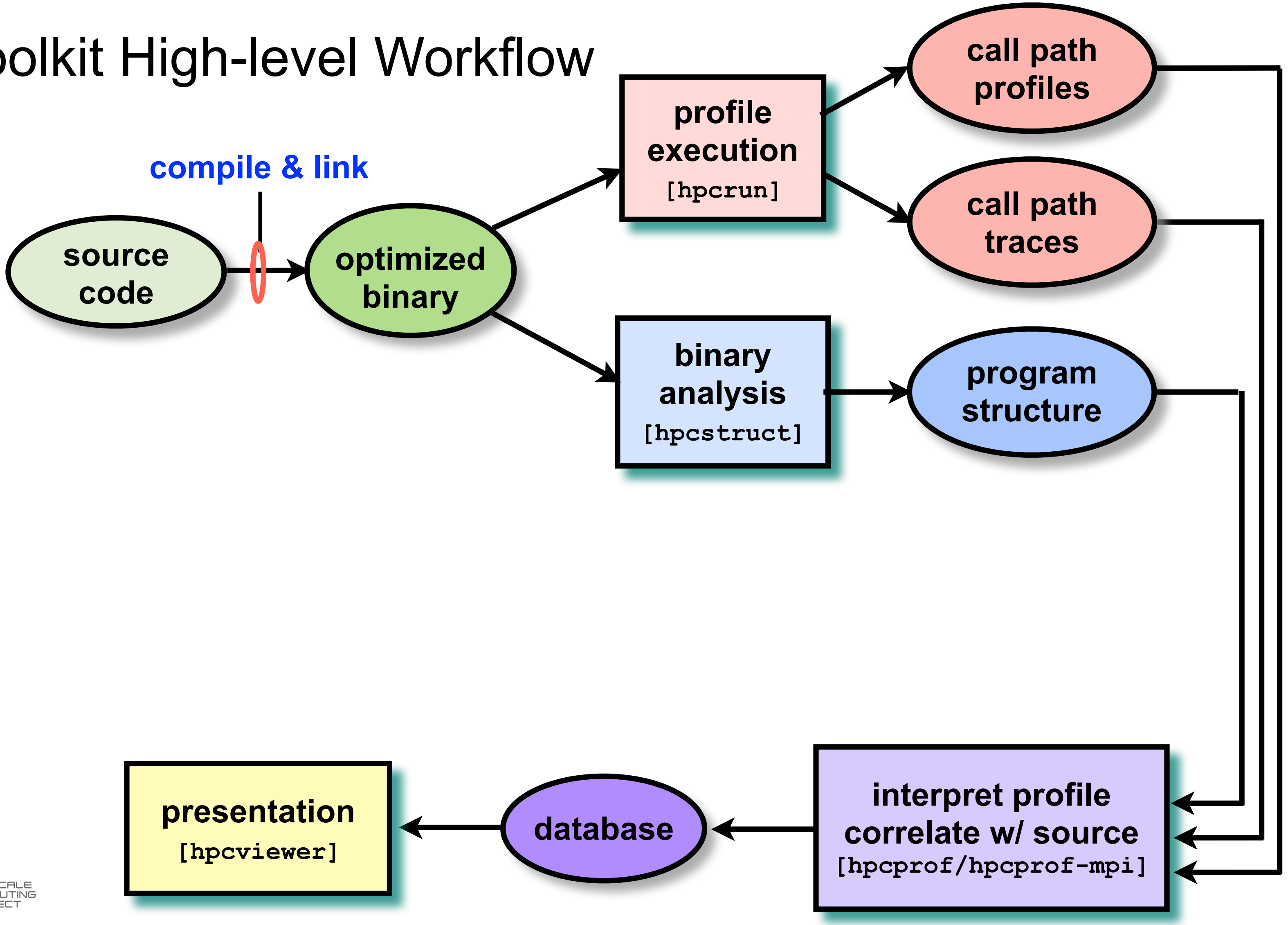
Outline

- **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

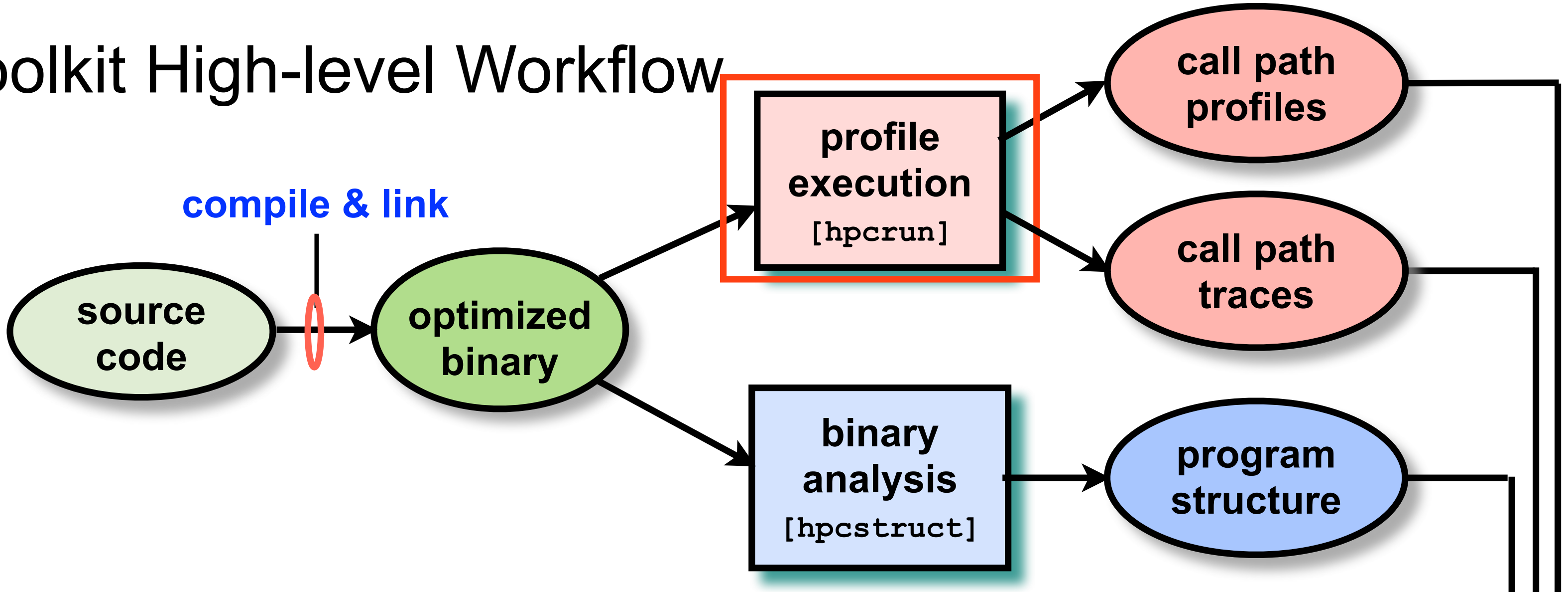
- **Employs binary-level measurement and analysis**
  - Observes executions of fully optimized, dynamically-linked parallel applications
  - Supports multi-lingual codes with external binary-only libraries
- **Collects sampling-based measurements of CPU**
  - Controllable overhead
  - Minimize systematic error and avoid blind spots
  - Enable data collection for large-scale parallelism
- **Measures GPU performance using vendor APIs**
  - Register callbacks to monitor launch/completion of GPU operations
  - Monitor asynchronous GPU operations using activity APIs from NVIDIA and AMD
  - Collect fine-grain measurements of GPU code using PC sampling (NVIDIA) and instrumentation (Intel GTPin)
- **Associates metrics with both static and dynamic context**
  - Loop nests, procedures, inlined code, calling contexts 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 High-level Workflow



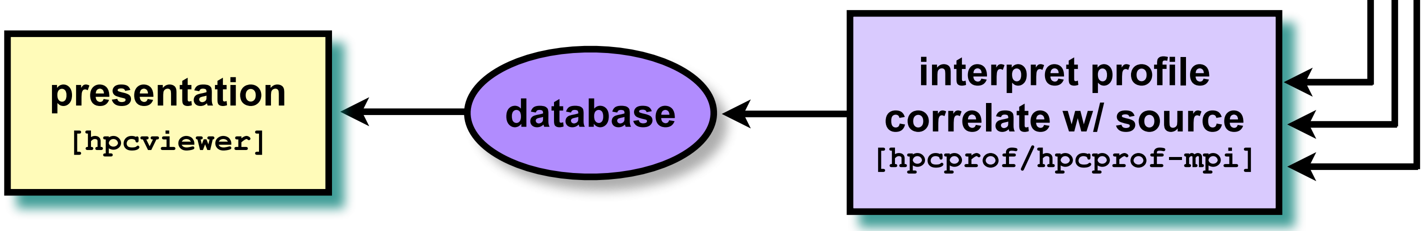
EXASCALE COMPUTING PROJECT

HPCToolkit High-level Workflow



Measure execution unobtrusively with `hpcrun`

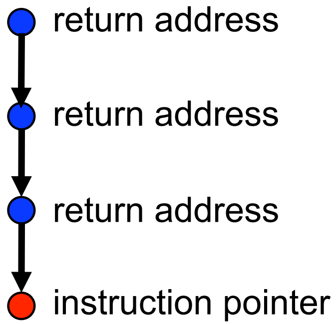
— Launch optimized dynamically-linked application binaries
— Collect call path profiles of events of interest
— Where necessary, intercept interfaces for control and measurement



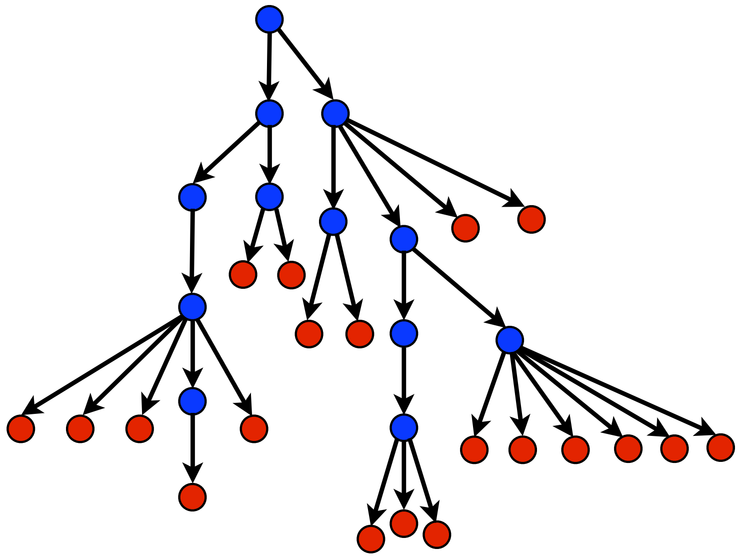
Call Path Profiling

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

Call path sample



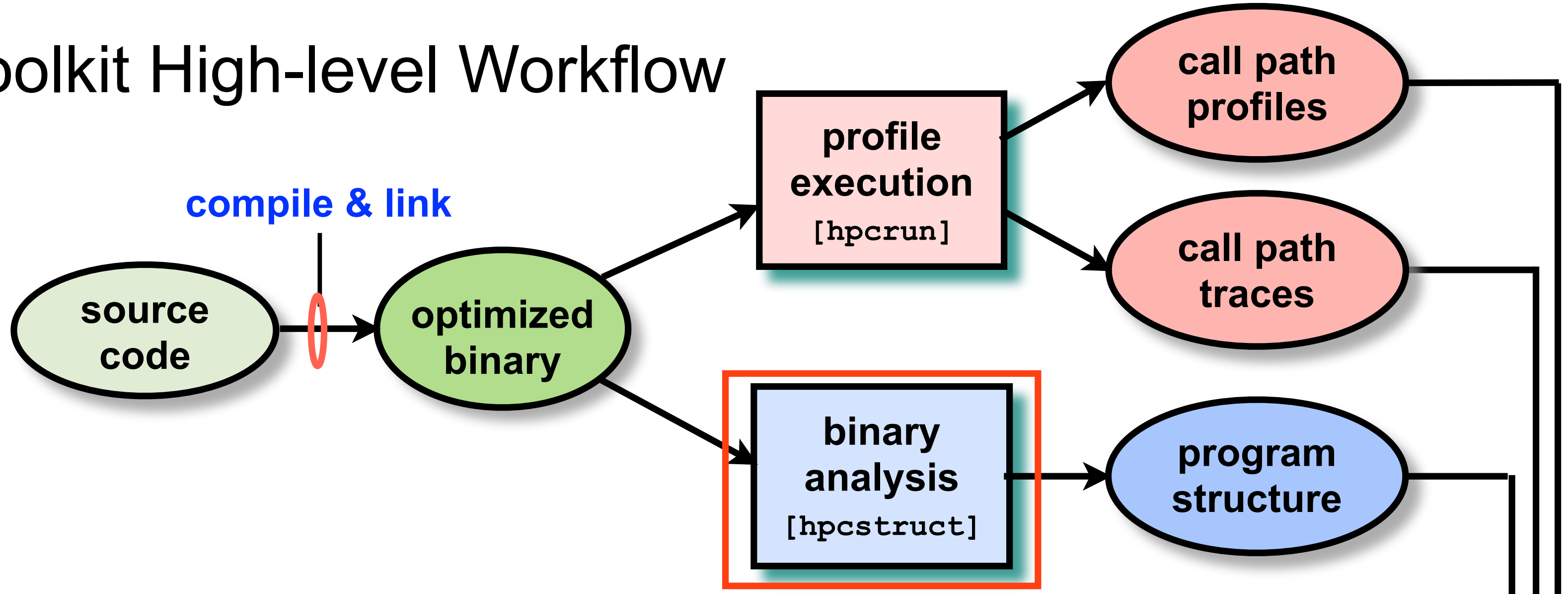
Calling context tree



Overhead proportional to sampling frequency, not call frequency

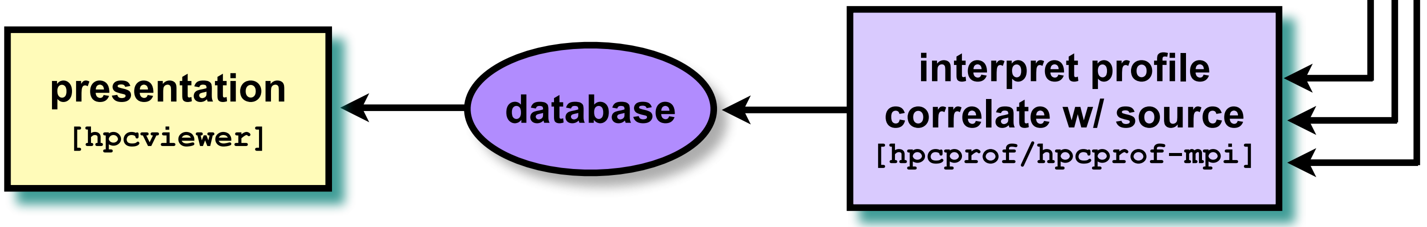
EXASCALE COMPUTING PROJECT

HPCToolkit High-level 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



EXASCALE COMPUTING PROJECT

Dyninst: A Toolkit for Binary Analysis and Instrumentation

Code Gen API	**Patch API**
Symtab API	**Parse API**
Instruction API	**DataFlow API**
StackWalker API	**ProcControl API**

Architectures

X86_64

Power/BE

Power/LE

ARM

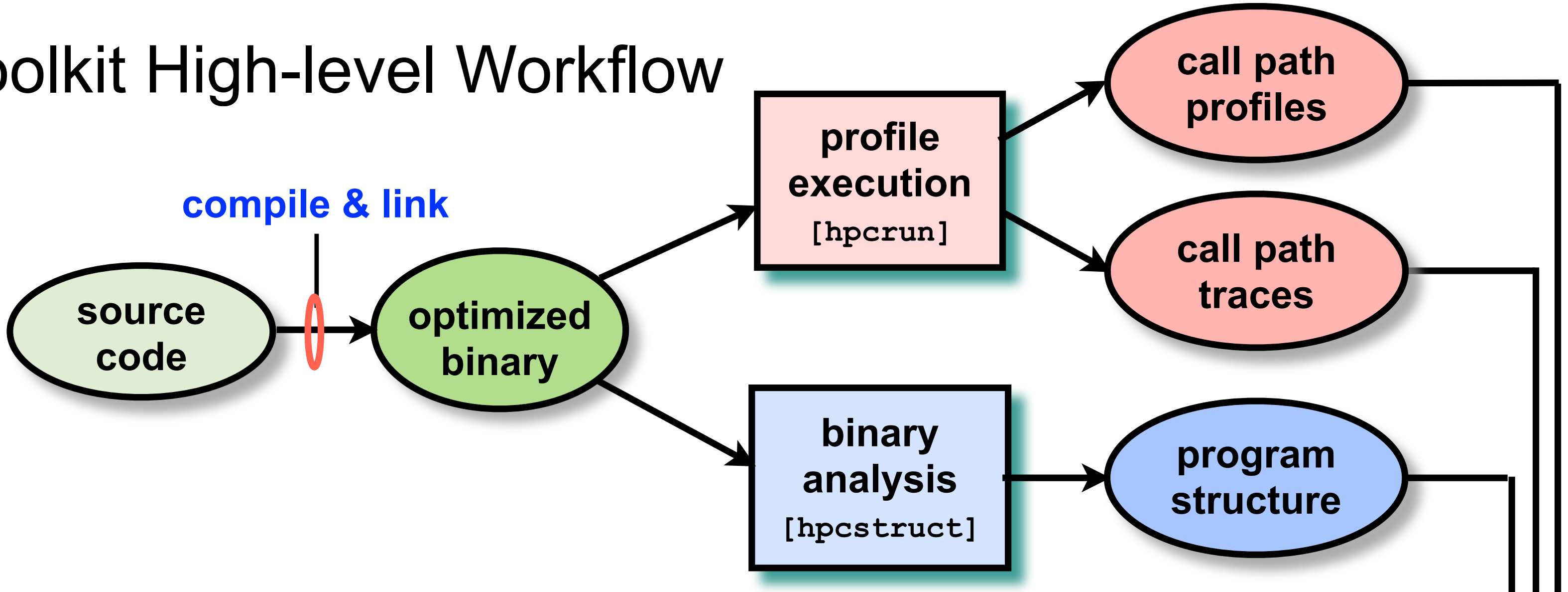
AMD Vega

CUDA

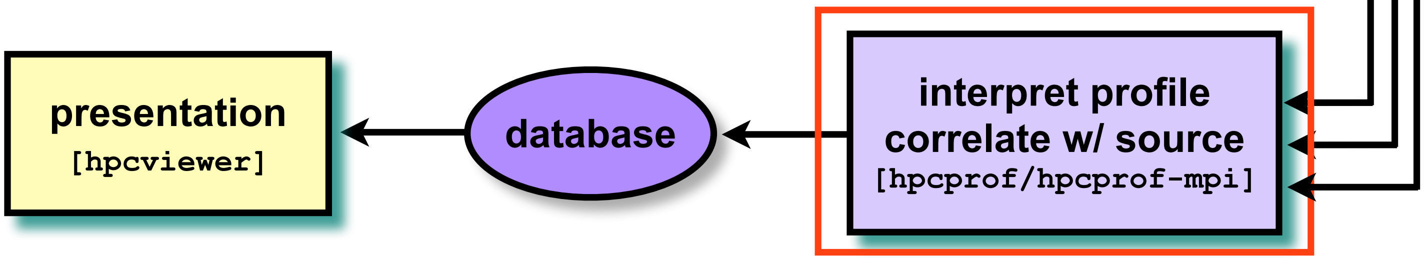
Intel GPU

Lead Institution: University of Wisconsin – Madison

HPCToolkit High-level Workflow

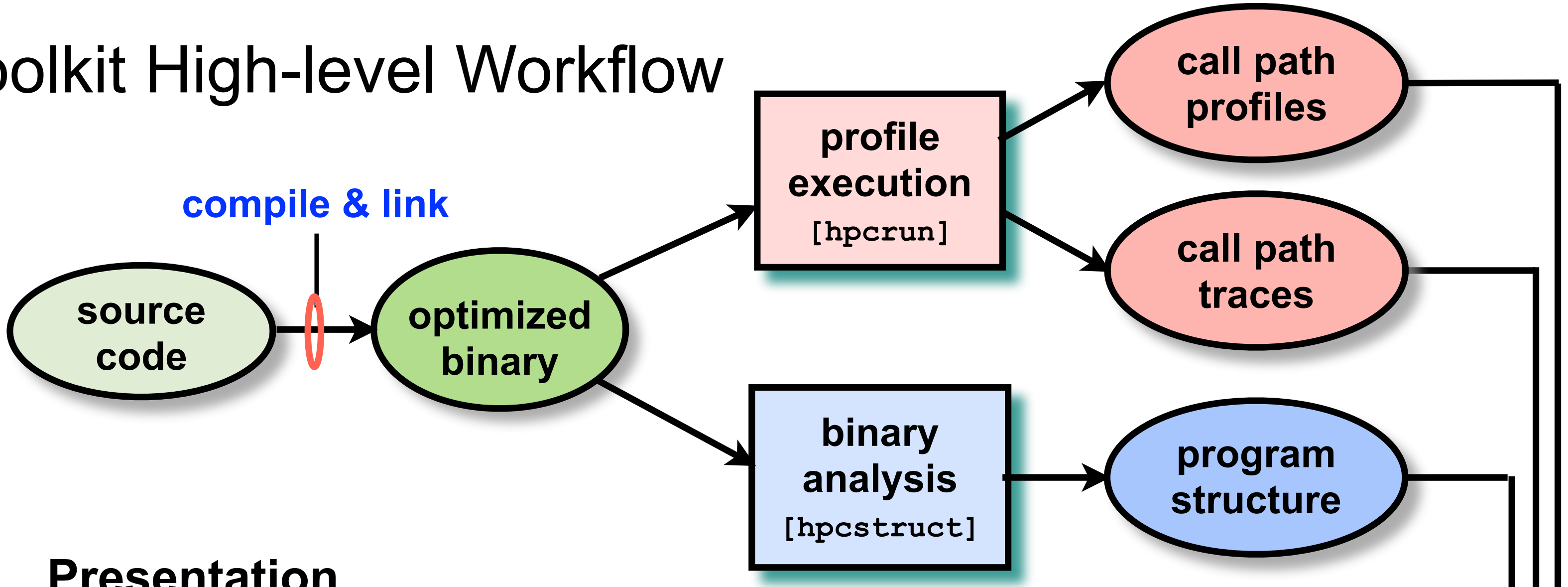


- **Combine multiple profiles**
  - Multiple threads; multiple processes; multiple executions
- **Correlate metrics to static & dynamic program structure**



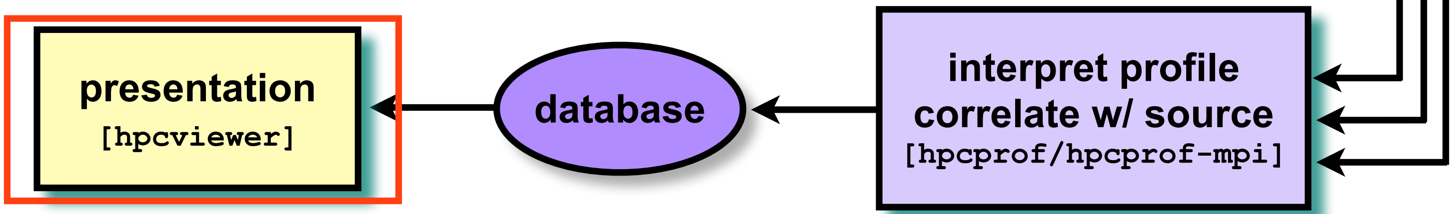
EXASCALE COMPUTING PROJECT

HPCToolkit High-level Workflow



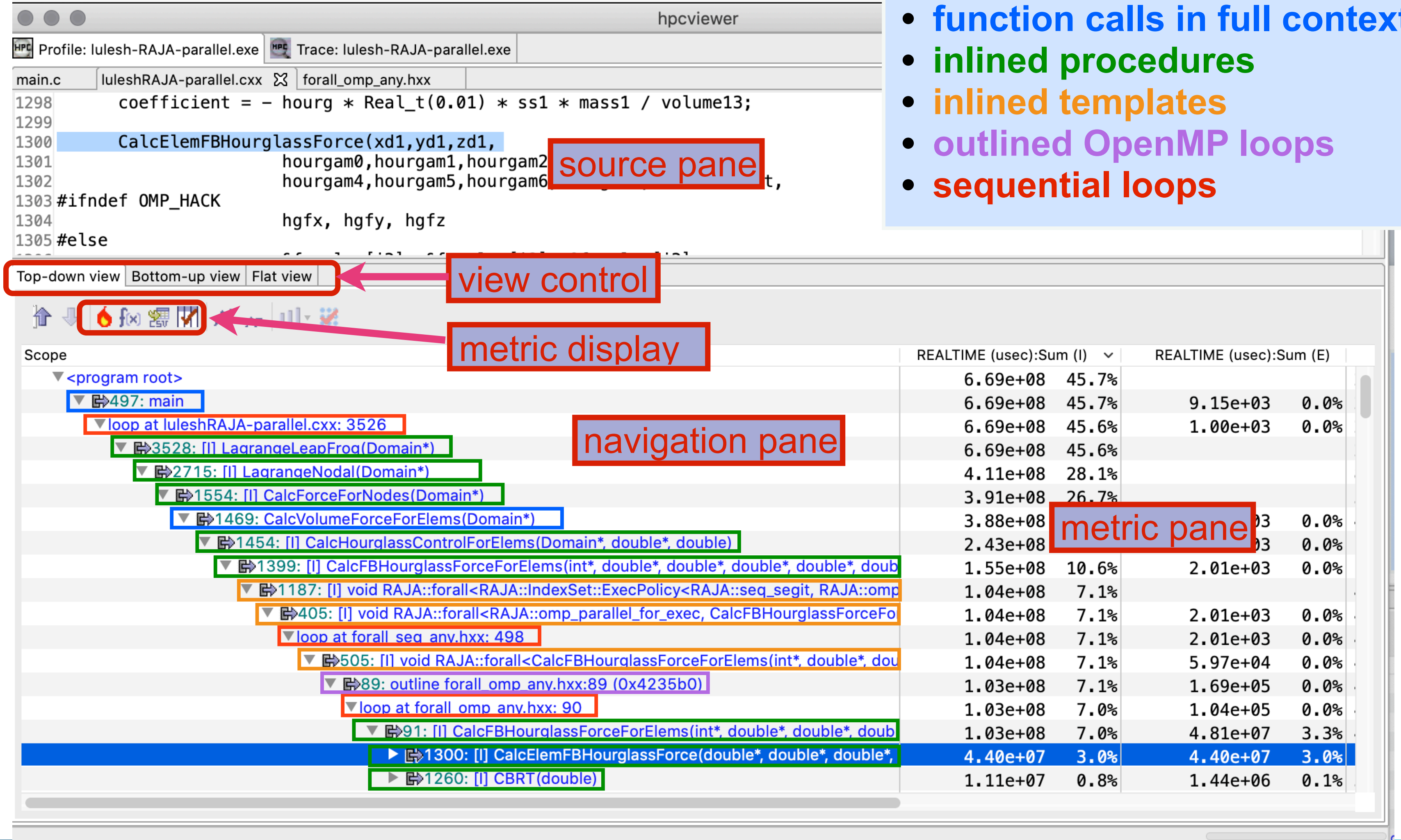
Presentation

- 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



EXASCALE COMPUTING PROJECT

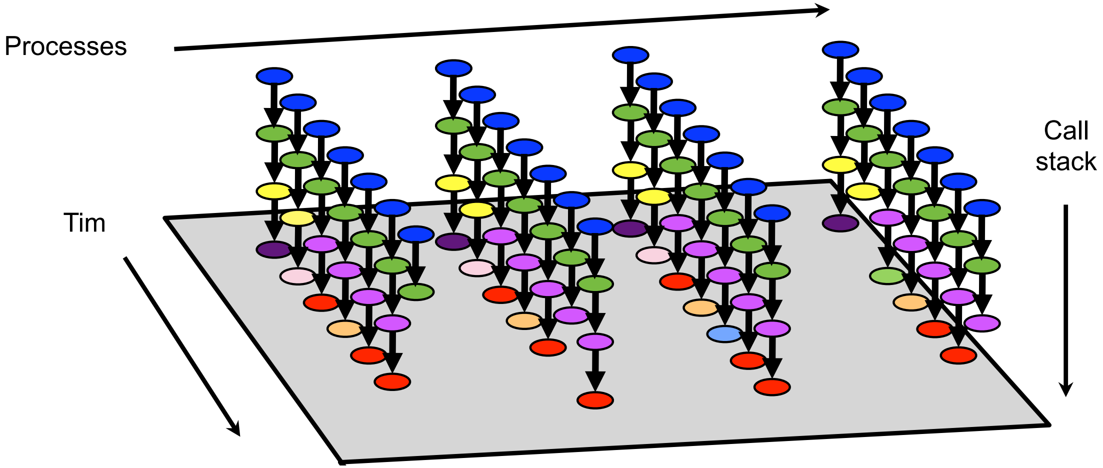
Code-centric Analysis with hpcviewer



- 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

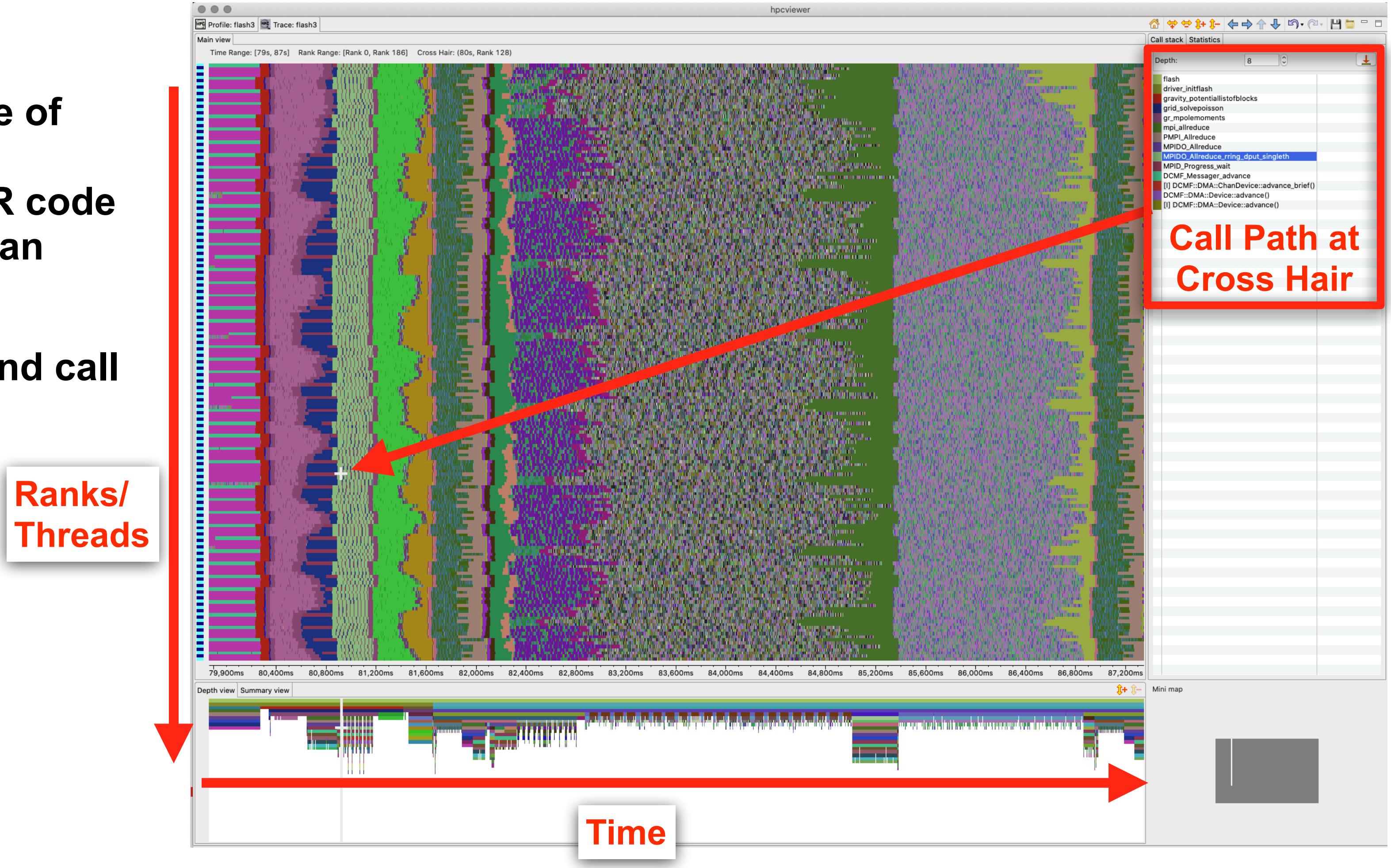


Time-centric Analysis of Call Path Traces

Detail of a trace of Flash3 - block structured AMR code written in Fortran

- 256 ranks

Depth, trace, and call path views

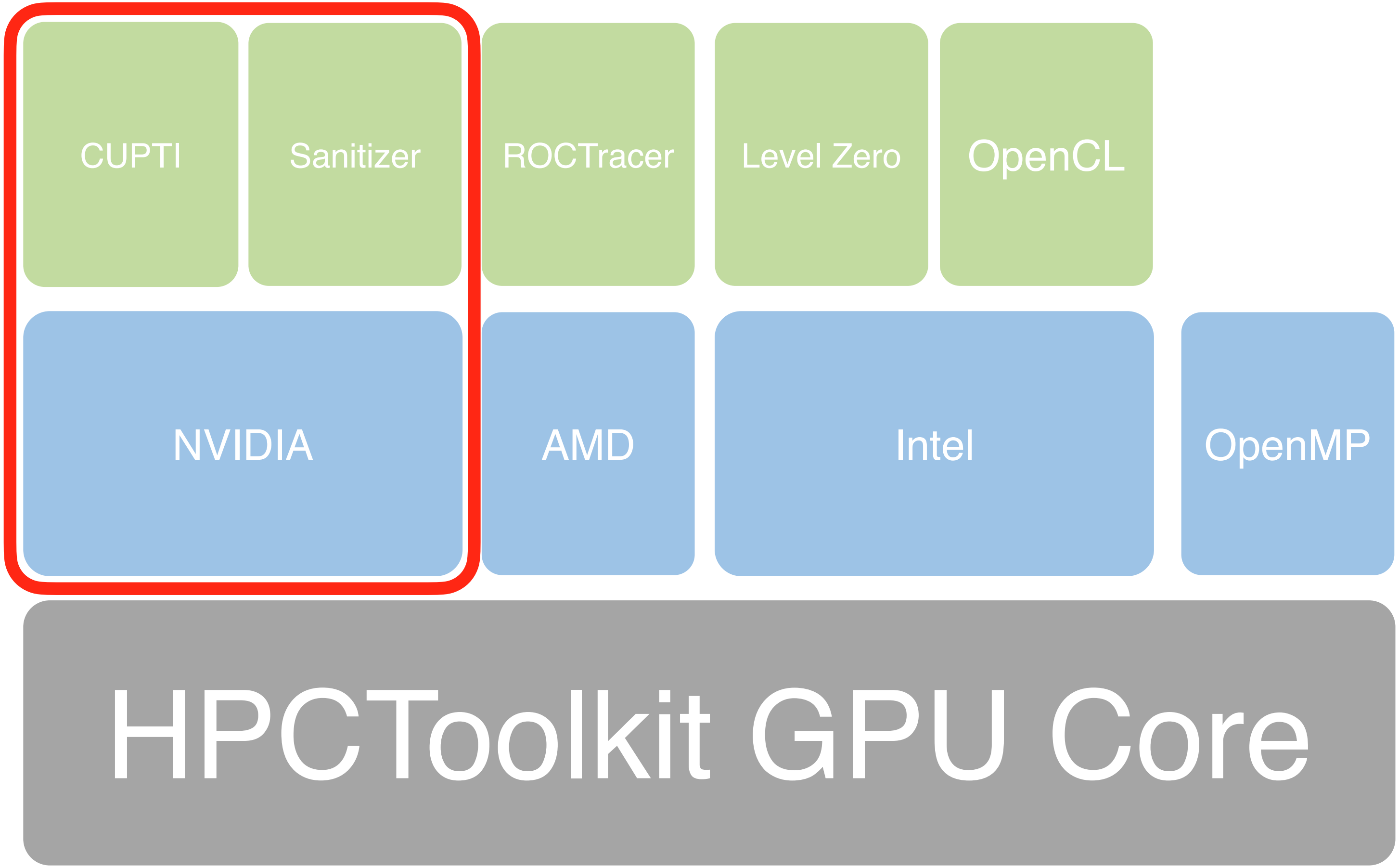


EXASCALE COMPUTING PROJECT

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**

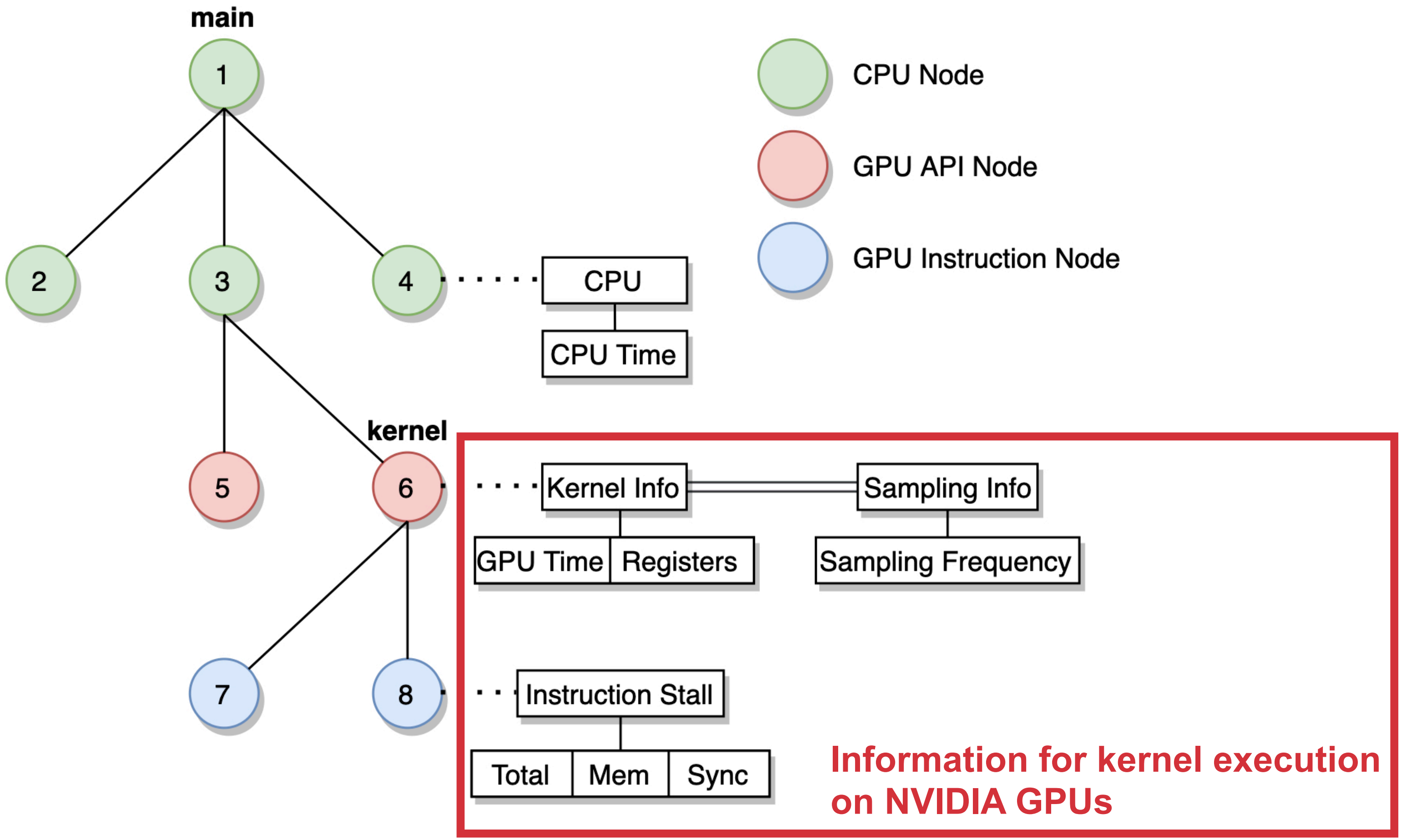
HPCToolkit for GPU-accelerated Computations



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 novel data structures for fast and non-blocking 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: derives 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 Sparse Representation of Measurements at Run-time



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

Profile: PeleC3d.gnu.CUDA.ex

main.cpp | mechanism.cpp | cuda_runtime.h

```
1911     /*convert to chemkin units */
1912     for (id = 0; id < 9; ++id) {
1913         C[id] *= 1.0e-6;
1914         wdot[id] *= 1.0e-6;
1915     }
1916 }
```

Top-down view | Bottom-up view | Flat view

Fine-grained Metrics — **Coarse-grained Metrics** — **Derived Metrics**

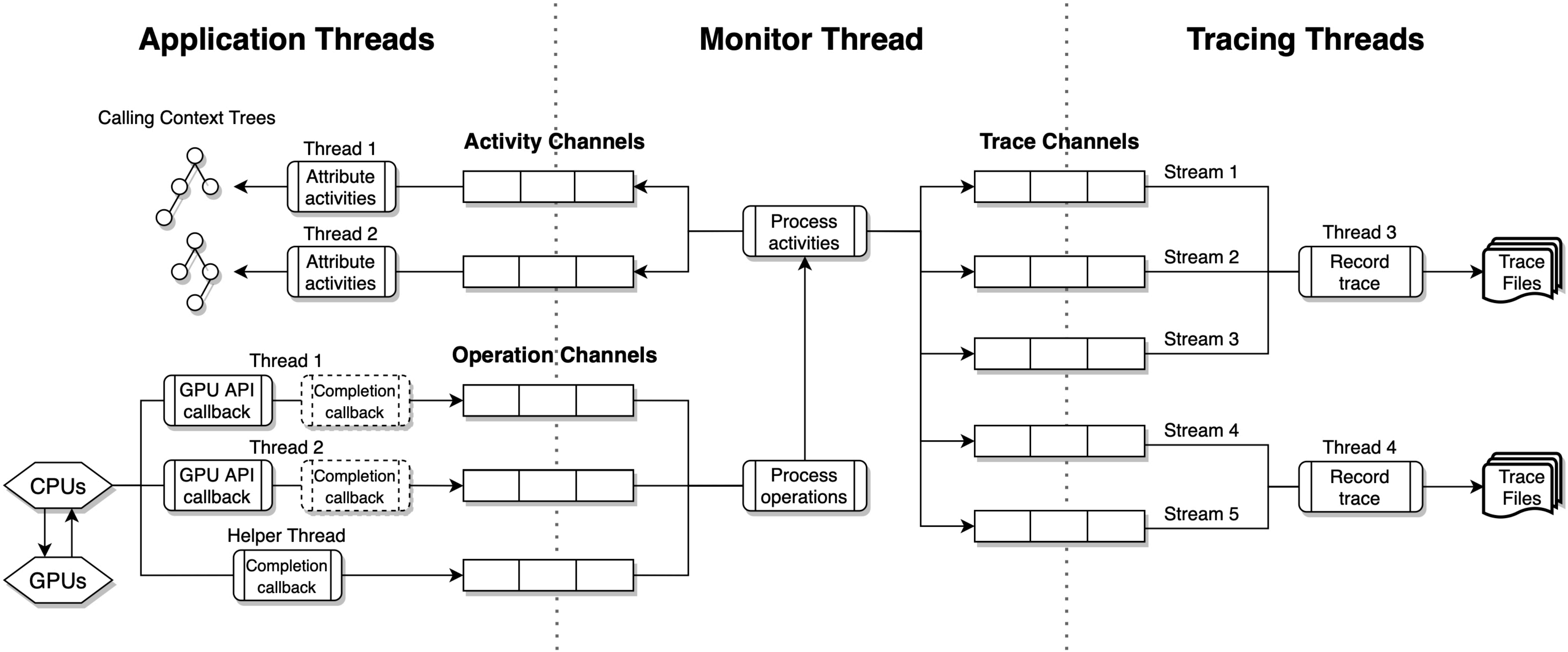
	Scope	GINS:Sum (I)		GINS:Sum (E)		GKER (sec):Sum (I)		GKER (sec):Sum (E)		GPU UTIL
	1325: ParallelFor<__nv_dl_wrapper_t<__nv_dl_tag<void (PeleC::*)(double, c	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
	797: [I] amrex::launch_global<__nv_dl_wrapper_t<__nv_dl_tag<void (*)(	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
	11: [I] __wrapper__device_stub_launch_global<__nv_dl_wrapper_t<_	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
	114: [I] __device_stub__ZN5amrex13launch_globalIZNS_11Paralle	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
	109: [I] cudaLaunchKernel<char>	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
CPU Calling Context	209: cudaLaunchKernel [PeleC3d.gnu.CUDA.ex]	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
GPU API Node	<gpu kernel>	3.30e+09	1.0%			2.99e-01	16.5%			2.49 %
	_ZN5amrex13launch_globalIZNS_11ParallelForIZN5P	3.08e+09	1.0%	2.57e+08	0.1%	2.99e-01	16.5%	2.99e-01	16.5%	
	loop at AMReX_GpuLaunchFunctsG.H: 797	3.08e+09	1.0%	2.56e+08	0.1%					
	[I] inlined from React.H: 163	3.08e+09	1.0%	2.56e+08	0.1%					
	loop at React.H: 163	3.08e+09	1.0%	2.53e+08	0.1%					
	[I] inlined from EOS.H: 59	2.88e+09	0.9%	5.28e+07	0.0%					
	168: CKWC	4.57e+08	0.1%	4.56e+08	0.1%					
GPU Calling Context	mechanism.cpp: 1914	2.13e+07	0.0%	2.13e+07	0.0%					

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)

EXASCALE COMPUTING PROJECT

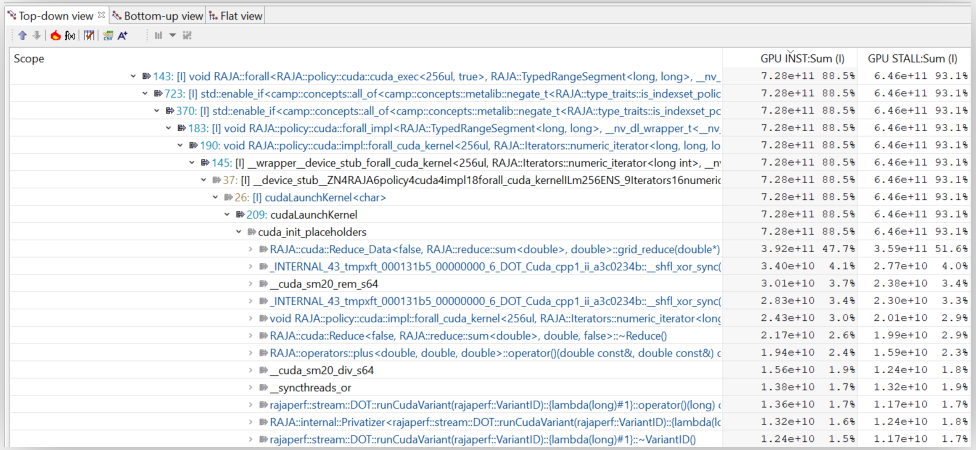
HPCToolkit’s Runtime Monitoring Infrastructure for OpenCL

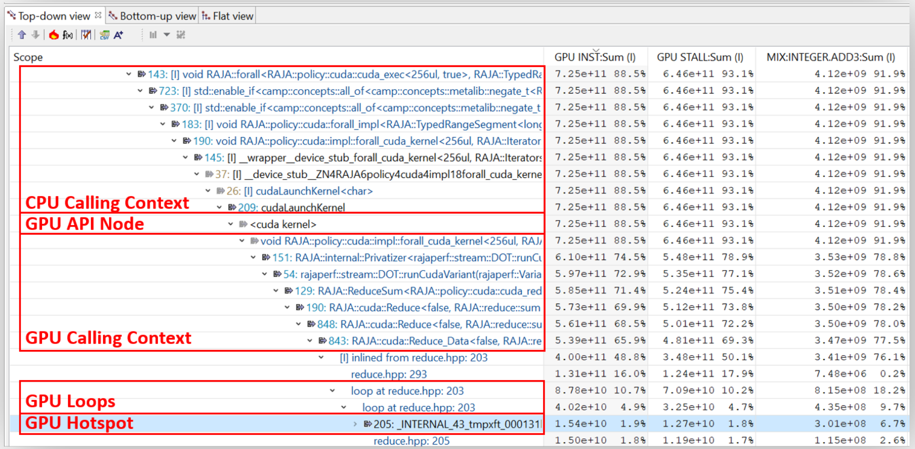


Approximation of GPU Calling Contexts to Understand Performance

- 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





EXASCALE COMPUTING PROJECT

Approximation of GPU Calling Contexts to Understand Performance

Top-down view | Bottom-up view | Flat view

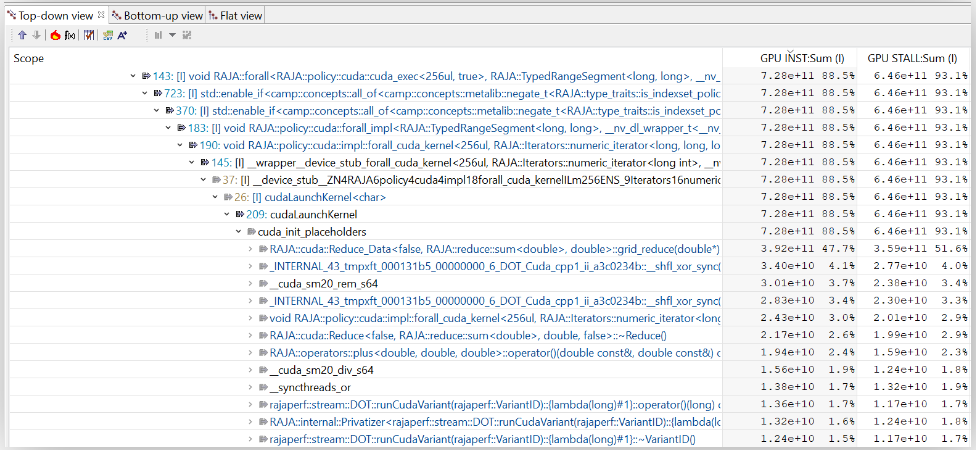
Scope	GPU INST:Sum (I)		GPU STALL:Sum (I)	
143: [I] void RAJA::forall<RAJA::policy::cuda::cuda_exec<256ul, true>, RAJA::TypedRangeSegment<long, long>, __nv_	7.28e+11	88.5%	6.46e+11	93.1%
723: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t<RAJA::type_traits::is_indexset_polic	7.28e+11	88.5%	6.46e+11	93.1%
370: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t<RAJA::type_traits::is_indexset_pc	7.28e+11	88.5%	6.46e+11	93.1%
183: [I] void RAJA::policy::cuda::forall_impl<RAJA::TypedRangeSegment<long, long>, __nv_dl_wrapper_t<__nv_	7.28e+11	88.5%	6.46e+11	93.1%
190: void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<long, long, lo	7.28e+11	88.5%	6.46e+11	93.1%
145: [I] __wrapper__device_stub_forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<long int>, __n	7.28e+11	88.5%	6.46e+11	93.1%
37: [I] __device_stub__ZN4RAJA6policy4cuda4impl18forall_cuda_kernelILm256ENS_9Iterators16numeric	7.28e+11	88.5%	6.46e+11	93.1%
26: [I] cudaLaunchKernel<char>	7.28e+11	88.5%	6.46e+11	93.1%
209: cudaLaunchKernel	7.28e+11	88.5%	6.46e+11	93.1%
cuda_init_placeholders	7.28e+11	88.5%	6.46e+11	93.1%
RAJA::cuda::Reduce_Data<false, RAJA::reduce::sum<double>, double>::grid_reduce(double*)	3.92e+11	47.7%	3.59e+11	51.6%
_INTERNAL_43_tmpxft_000131b5_00000000_6_DOT_Cuda_cpp1_ii_a3c0234b::__shfl_xor_sync(	3.40e+10	4.1%	2.77e+10	4.0%
__cuda_sm20_rem_s64	3.01e+10	3.7%	2.38e+10	3.4%
_INTERNAL_43_tmpxft_000131b5_00000000_6_DOT_Cuda_cpp1_ii_a3c0234b::__shfl_xor_sync(	2.83e+10	3.4%	2.30e+10	3.3%
void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJA::Iterators::numeric_iterator<lonc	2.43e+10	3.0%	2.01e+10	2.9%
RAJA::cuda::Reduce<false, RAJA::reduce::sum<double>, double, false>::~Reduce()	2.17e+10	2.6%	1.99e+10	2.9%
RAJA::operators::plus<double, double, double>::operator()(double const&, double const&) c	1.94e+10	2.4%	1.59e+10	2.3%
__cuda_sm20_div_s64	1.56e+10	1.9%	1.24e+10	1.8%
__syncthreads_or	1.38e+10	1.7%	1.32e+10	1.9%
rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(long)#1}::operator()(long) c	1.36e+10	1.7%	1.17e+10	1.7%
RAJA::internal::Privatizer<rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(lc	1.32e+10	1.6%	1.24e+10	1.8%
rajaperf::stream::DOT::runCudaVariant(rajaperf::VariantID)::{lambda(long)#1}::~VariantID()	1.24e+10	1.5%	1.17e+10	1.7%

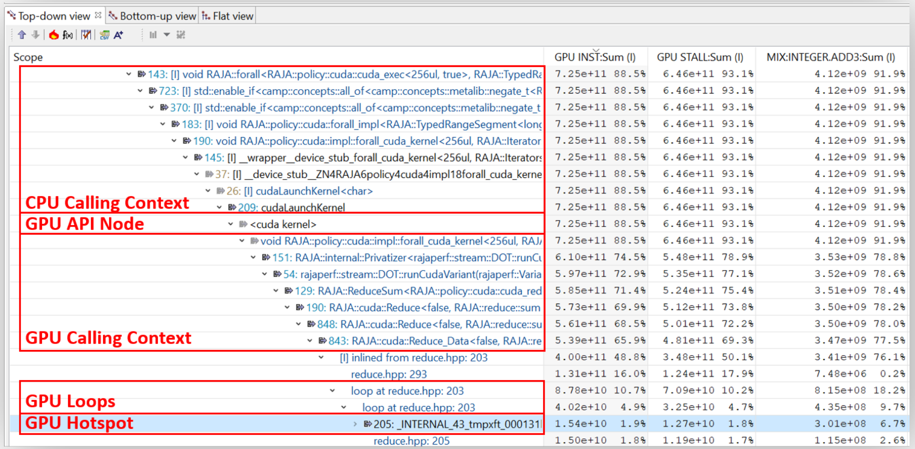
flat profile of functions called by a GPU kernel

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





Approximate Performance Attribution to GPU Calling Contexts

Top-down view | Bottom-up view | Flat view

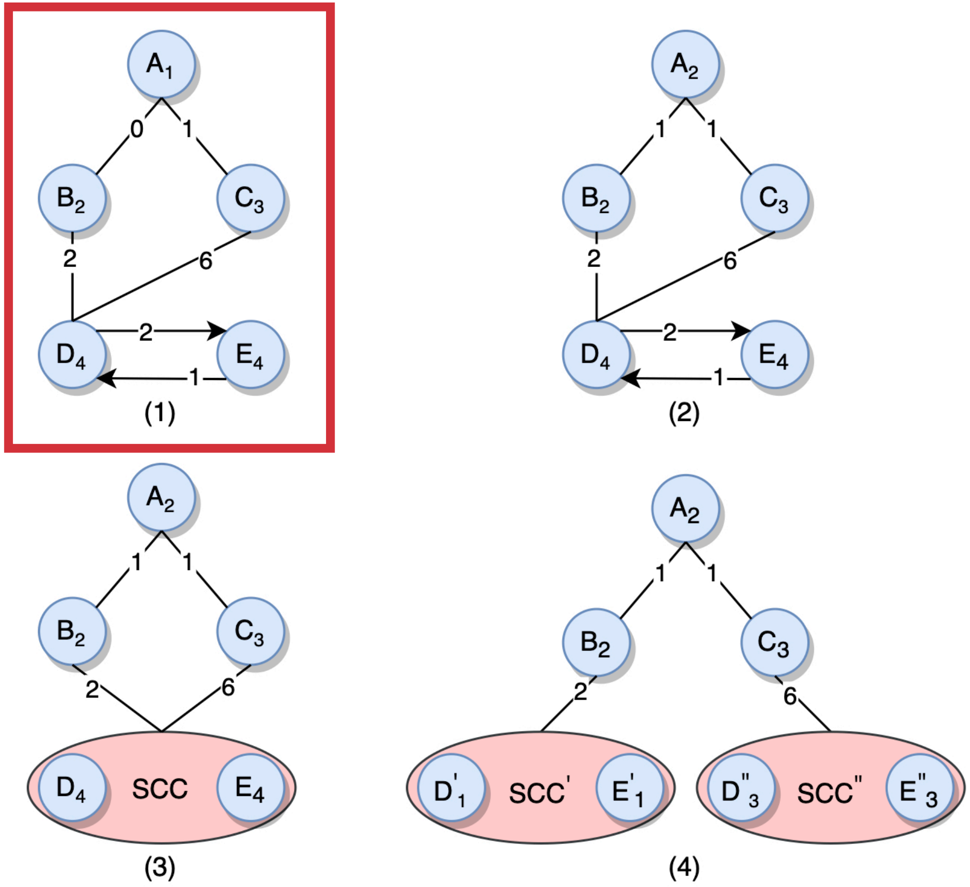
	Scope	GPU INST:Sum (I)		GPU STALL:Sum (I)		MIX:INTEGER.ADD3:Sum (I)	
	143: [I] void RAJA::forall<RAJA::policy::cuda::cuda_exec<256ul, true>, RAJA::TypedRa	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	723: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t<R	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	370: [I] std::enable_if<camp::concepts::all_of<camp::concepts::metalib::negate_t	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	183: [I] void RAJA::policy::cuda::forall_impl<RAJA::TypedRangeSegment<long	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	190: void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJA::Iterator	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	145: [I] __wrapper__device_stub_forall_cuda_kernel<256ul, RAJA::Iterators	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	37: [I] __device_stub__ZN4RAJA6policy4cuda4impl18forall_cuda_kerne	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	26: [I] cudaLaunchKernel<char>	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
CPU Calling Context	209: cudaLaunchKernel	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
GPU API Node	<cuda kernel>	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	void RAJA::policy::cuda::impl::forall_cuda_kernel<256ul, RAJ/	7.25e+11	88.5%	6.46e+11	93.1%	4.12e+09	91.9%
	151: RAJA::internal::Privatizer<rajaperf::stream::DOT::runCu	6.10e+11	74.5%	5.48e+11	78.9%	3.53e+09	78.8%
	54: rajaperf::stream::DOT::runCudaVariant(rajaperf::Varia	5.97e+11	72.9%	5.35e+11	77.1%	3.52e+09	78.6%
	129: RAJA::ReduceSum<RAJA::policy::cuda::cuda_red	5.85e+11	71.4%	5.24e+11	75.4%	3.51e+09	78.4%
	190: RAJA::cuda::Reduce<false, RAJA::reduce::sum	5.73e+11	69.9%	5.12e+11	73.8%	3.50e+09	78.2%
	848: RAJA::cuda::Reduce<false, RAJA::reduce::su	5.61e+11	68.5%	5.01e+11	72.2%	3.50e+09	78.0%
GPU Calling Context	843: RAJA::cuda::Reduce_Data<false, RAJA::re	5.39e+11	65.9%	4.81e+11	69.3%	3.47e+09	77.5%
	[I] inlined from reduce.hpp: 203	4.00e+11	48.8%	3.48e+11	50.1%	3.41e+09	76.1%
	reduce.hpp: 293	1.31e+11	16.0%	1.24e+11	17.9%	7.48e+06	0.2%
GPU Loops	loop at reduce.hpp: 203	8.78e+10	10.7%	7.09e+10	10.2%	8.15e+08	18.2%
	loop at reduce.hpp: 203	4.02e+10	4.9%	3.25e+10	4.7%	4.35e+08	9.7%
GPU Hotspot	205: _INTERNAL_43_tmpxft_000131	1.54e+10	1.9%	1.27e+10	1.8%	3.01e+08	6.7%
	reduce.hpp: 205	1.50e+10	1.8%	1.19e+10	1.7%	1.15e+08	2.6%

Reconstruction of GPU Calling Context Trees

- **Problem**
  - Vendor GPU monitoring APIs don’t collect call paths inside GPU kernels
- **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 Performance Attribution to GPU Calling Contexts

1. Construct a GPU static call graph based on functions and call instructions. Initialize call edge 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.



EXASCALE COMPUTING PROJECT

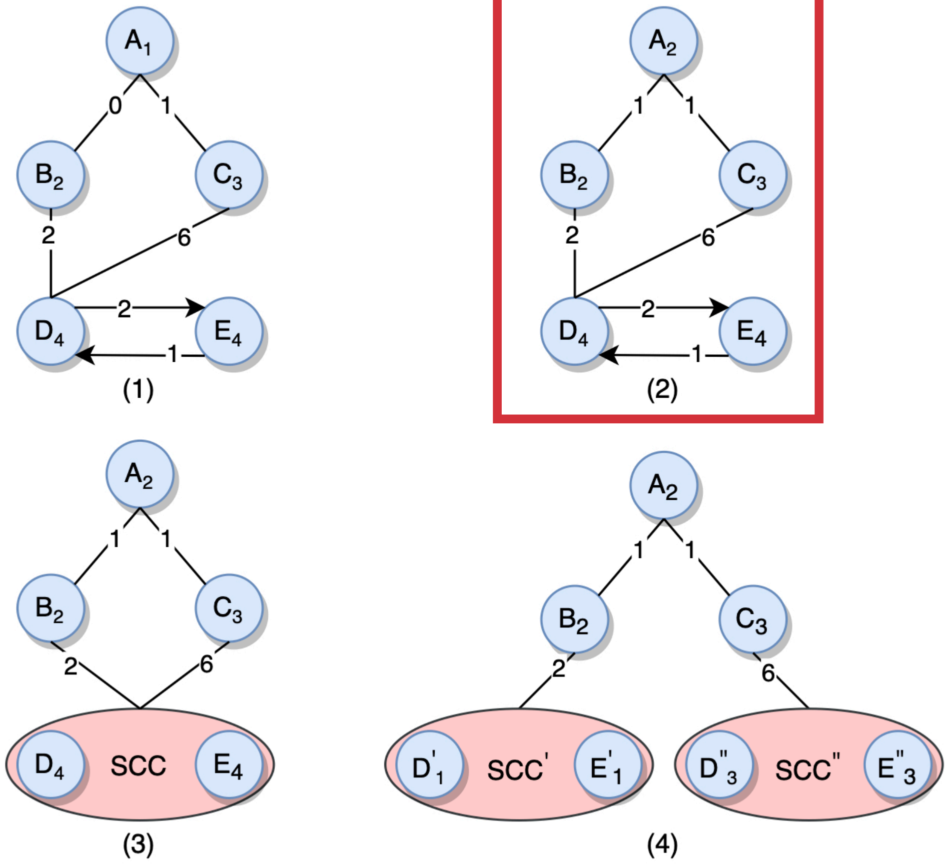
Approximate Performance Attribution to GPU Calling Contexts

1. Construct a GPU static call graph based on functions and call instructions. Initialize call edge 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.



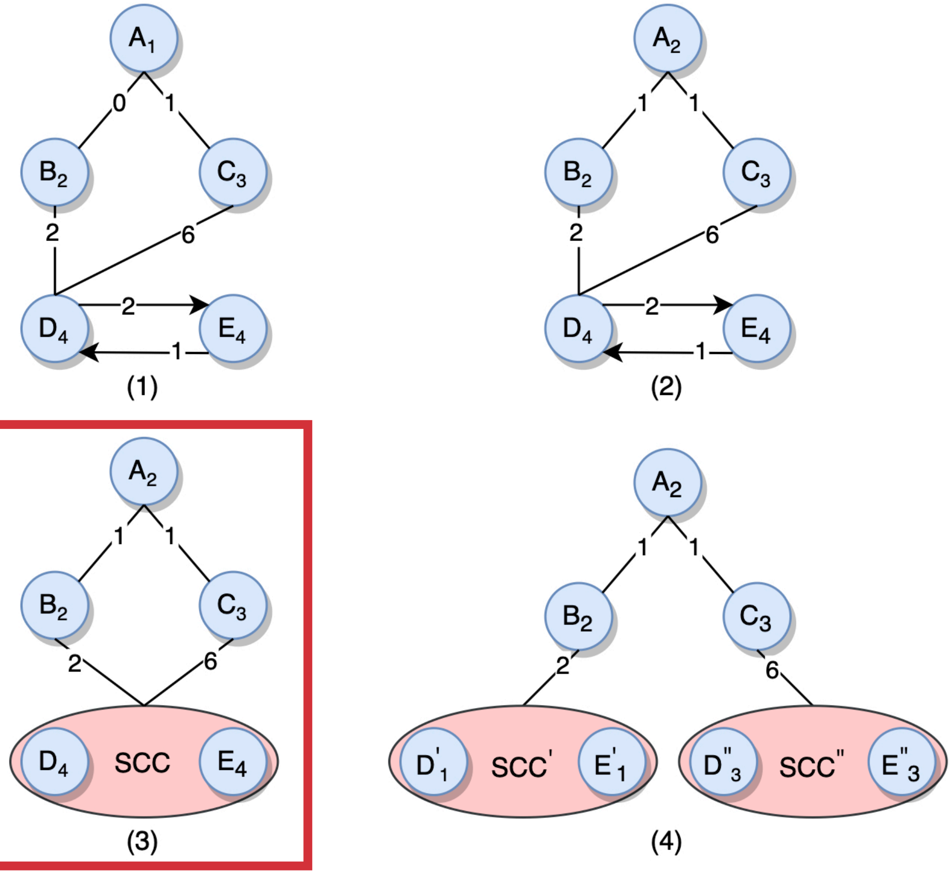
Approximate Performance Attribution to GPU Calling Contexts

1. Construct a GPU static call graph based on functions and call instructions. Initialize call edge 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.



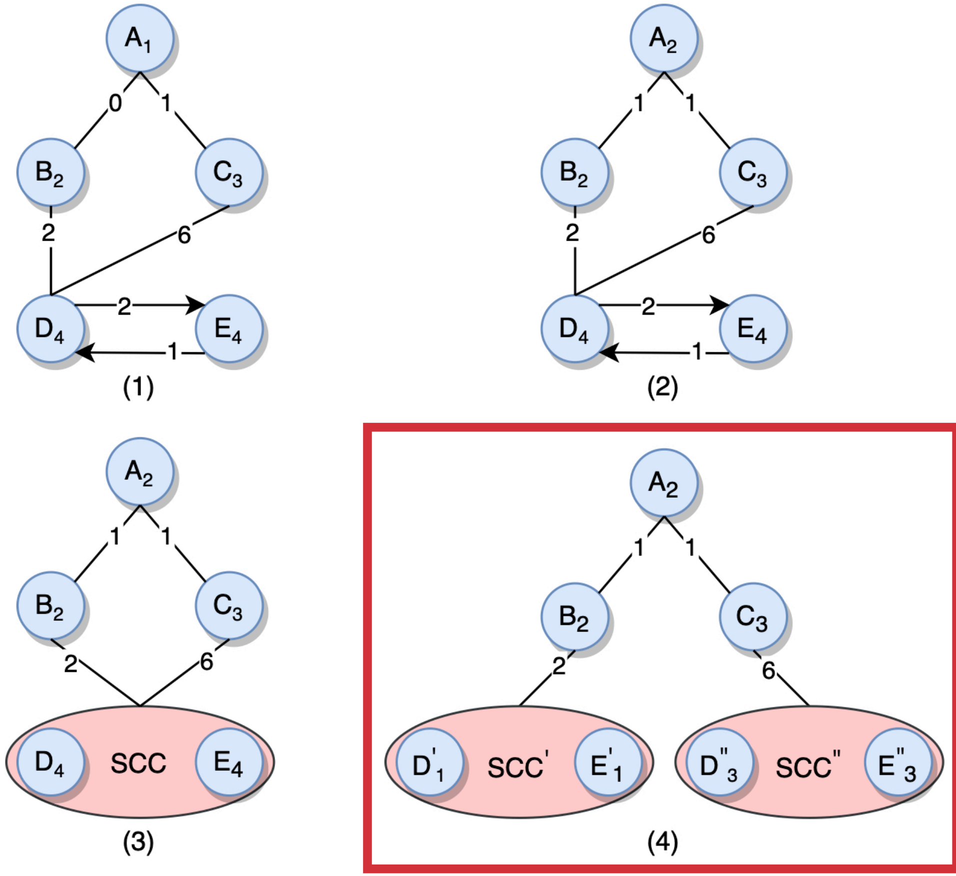
Approximate Performance Attribution to GPU Calling Contexts

1. Construct a GPU static call graph based on functions and call instructions. Initialize call edge 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.

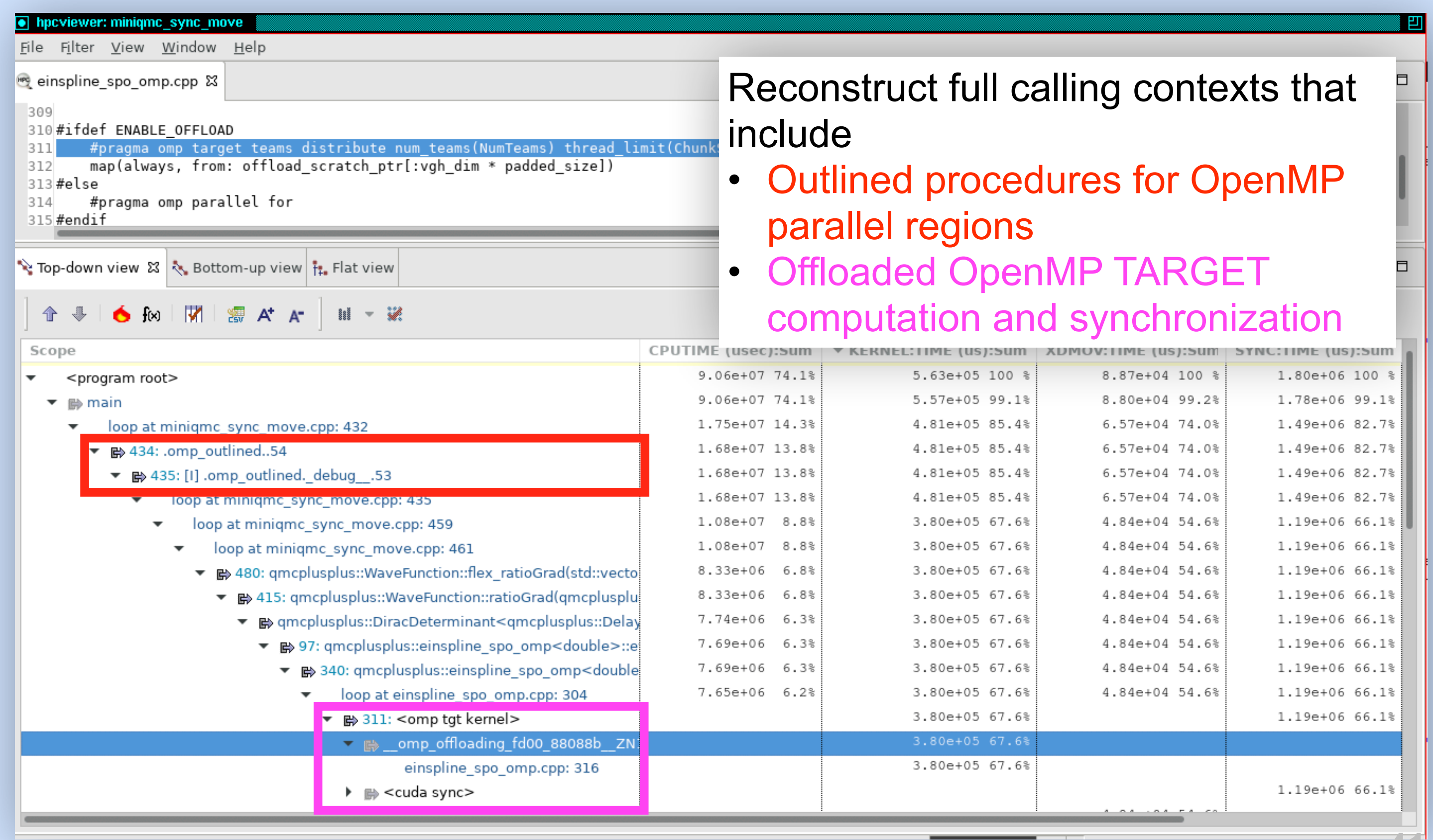
4. Build CCT by splitting call graph. Like gprof, assume that every call to a function has equal cost. Apportion costs of each function among its call sites according to ratios of calls from each call site.



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)



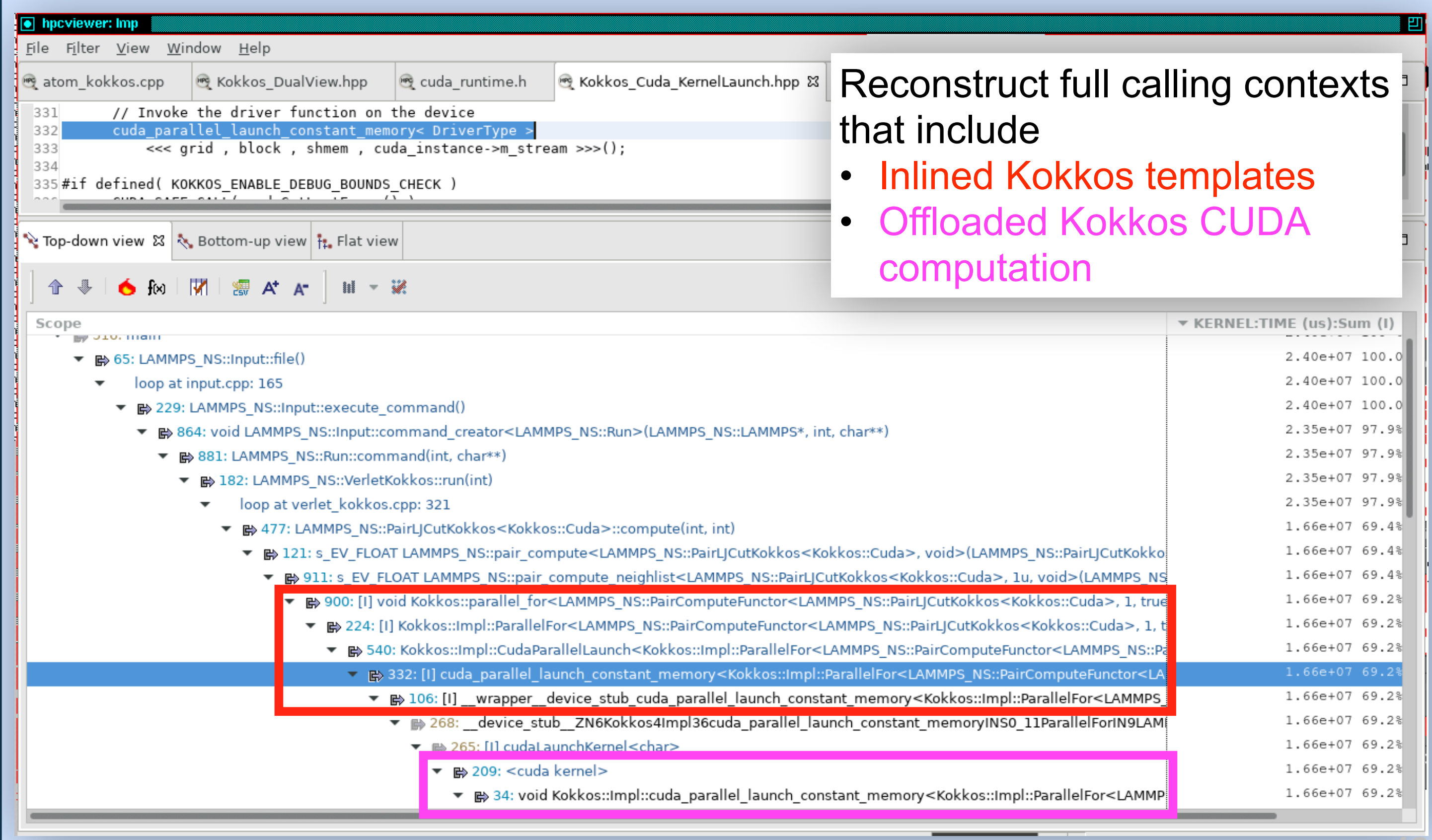
Reconstruct full calling contexts that include
- Outlined procedures for OpenMP parallel regions
- Offloaded OpenMP TARGET computation and synchronization

EXASCALE COMPUTING PROJECT

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)



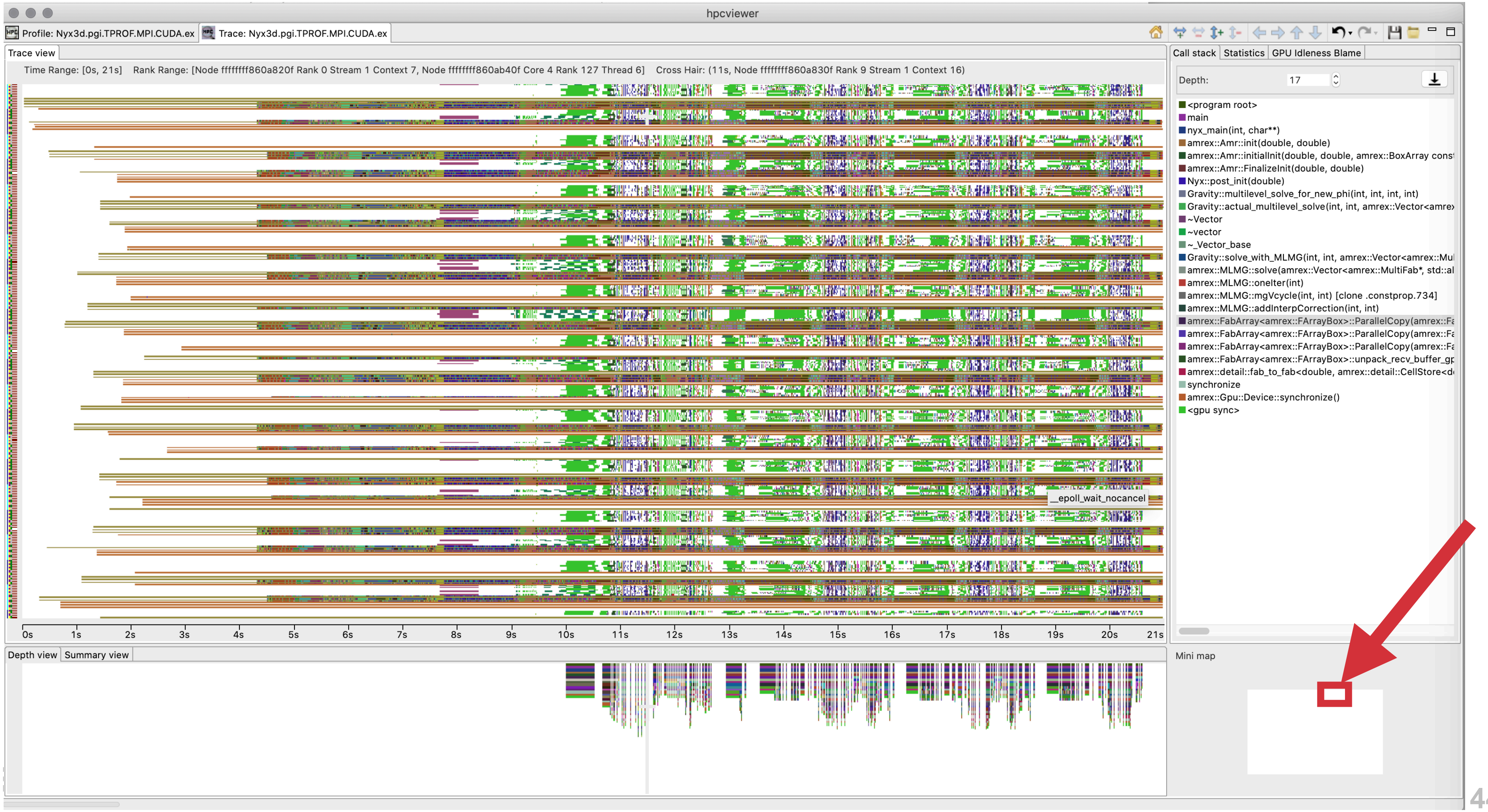
Reconstruct full calling contexts that include
- Inlined Kokkos templates
- Offloaded Kokkos CUDA computation

EXASCALE COMPUTING PROJECT

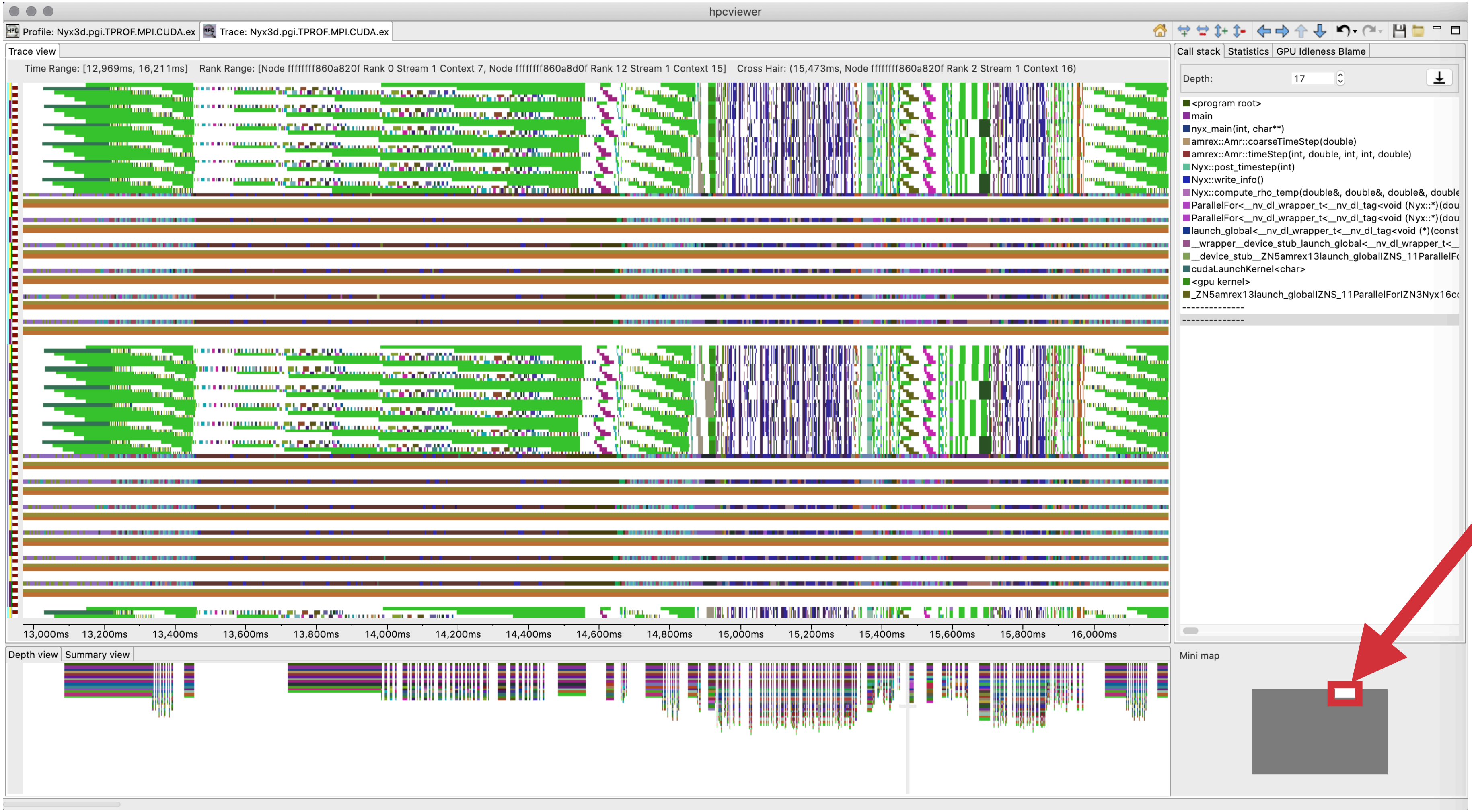
Deriving GPU Metrics

- **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**
  - Measure a single pass of an execution and collect PC samples
  - Derive multiple metrics using PC samples and other activity records
    - e.g., GPU SM utilization, GPU occupancy, …

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

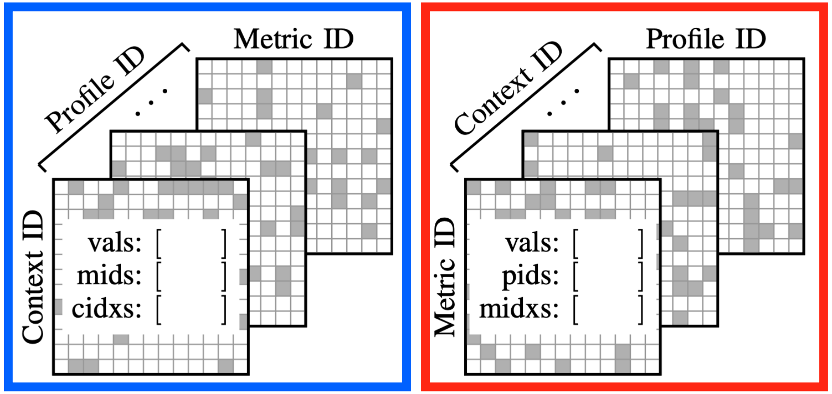


Nyx with CUDA: Trace of Multi-rank Multi-GPU Executions



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



Is Using Sparse Formats Important?

- **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

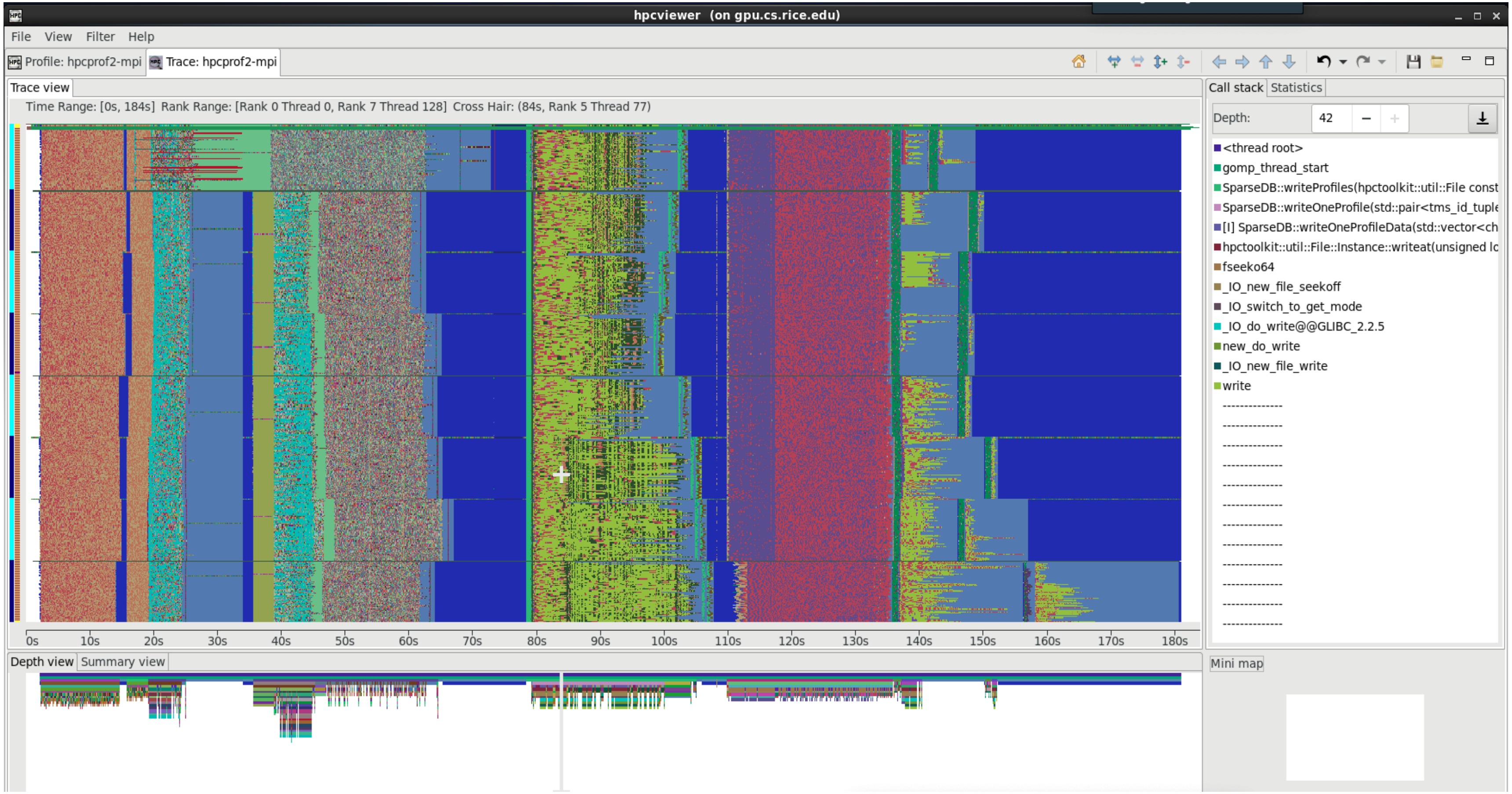
Dataset		Size (MiB)		
		Dense	Sparse	Ratio
AMG2006 (1)	M	659.0	911.0	0.723×
	D	7370.0	836.0	8.819×
AMG2006 (9)	M	21.7	11.1	1.956×
	D	2290.0	33.5	68.34 ×
Nyx	M	5890.0	278.0	21.14 ×
	D	130 GiB	601.0	221.5 ×
LAMMPS	M	85.5	5.23	16.35 ×
	D	8250.0	24.5	336.9 ×

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



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	emerging Dyninst instrumenter	Roctracer	emerging Dyninst decoder

Detailed Performance Analysis Requires Support at Many Levels

Hardware and Software Stack Components	Partners
• **Hardware must include support for fine-grain measurement and attribution** • performance counters are not enough; NVIDIA’s PC sampling approximates our needs	GPU vendors

• **System software must provide appropriate interfaces for introspection and analysis**
  • e.g. Linux perf_events supports sample-based performance monitoring even in the kernel
  • e.g. dynamic loader (ld.so) provides LD_AUDIT interface for monitoring and control of dynamic library operations
  • elfutils must support NVIDIA’s extended line maps in CUDA 11.2+ GPU binaries
	Red Hat

• GPU vendor software stacks (kernel driver, runtime, tools API)	GPU vendors
• **Compiler must compute high-quality DWARF information** • associate each machine instruction with full call chains involving inlined templates and functions	Vendors and LLVM community
• **Runtime must maintain information needed to map computations to a source-level view** • OpenMP’s OMPT helps bridge the vast gap between the implementation and user-level view	OpenMP Language Committee and LLVM Community

• **Performance tools must gather measurements using multiple modalities and map them to source**
  • precise attribution when possible
  • reconstruct approximate attribution when precise attribution is unavailable
    • GPU calling context
    • loops in CPU and GPU code
    • attribute inefficiencies from where they are observed back to their causes
	Wisconsin’s Dyninst Project

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
  - Integrated user interface that supports both profiles and traces
    - Automated serialization analysis of CPU and GPU traces
- **Internals**
  - Collecting GPU hardware counters, which will support Roofline analysis
  - Updating measurement and analysis support for NVIDIA GPUs (emerging CUPTI, more info about inlining)
  - Extending HPCToolkit to support analysis of machine learning frameworks: Pytorch, Tensorflow
  - Improving scalability of measurement and analysis
  - Developing instrumentation to assess performance on Intel GPUs
  - Refining implementation of monitoring for Intel’s Level 0
  - Improving binary analysis of AMD GPU binaries

Final Remarks

- **Nice to work with national labs and have early involvement in big procurements**
  - 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
    - instability and 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