HPCToolkit

Release 2026.1.0-alpha

The HPCToolkit Developers

Sep 13, 2026

CONTENTS

HPCToolkit is an integrated suite of tools for measurement and analysis of program performance on computers ranging from multicore desktop systems to the world's largest GPU-accelerated supercomputers. HPCToolkit can measure a program's work, resource consumption, and inefficiency on both CPUs and GPUs (Adhianto et al. 2010; Zhou et al. 2020; Adhianto et al. 2024). HPCToolkit correlates such metrics with the program's source code, works with multilingual, fully optimized binaries, has very low measurement overhead, and scales to large parallel systems. HPCToolkit's measurements provide support for analyzing a program execution cost, inefficiency, and scaling characteristics both within and across nodes of a parallel system.

CHAPTER

ONE

INTRODUCTION

HPCToolkit supports performance measurement and analysis on ARM, x86_64, and IBM Power processors as well as AMD, Intel, and NVIDIA GPUs.

On CPUs, HPCToolkit principally monitors an execution of a multithreaded and/or multiprocess program using asynchronous sampling. When an asynchronous sample interrupts a thread, HPCToolkit unwinds the thread's call stack and attributes the metric value associated with the sample event to the calling context in which the sample was delivered. Asynchronous sampling is typically triggered by the expiration of a Linux timer or reaching a threshold value for a hardware performance counter. Sampling has several advantages over instrumentation for measuring program performance: it requires no modification of source code, it avoids potential blind spots due to uninstrumented code, and it has lower overhead. HPCToolkit typically adds measurement overhead of only a few percent to an execution for reasonable sampling rates (Tallent, Mellor-Crummey, and Fagan 2009). Sampling enables fine-grain measurement and attribution of costs in both serial and parallel programs.

On GPUs, HPCToolkit collects profiles and traces of GPU operations, such as data copies or kernel launches, using vendor-provided libraries, such as NVIDIA's CUPTI, AMD's Rocprofiler-sdk, and Intel's Level Zero. HPCToolkit correlates GPU measurements to application source code by using call stack unwinding to attribute metrics associated with a GPU operation back to the calling context where the operation was initiated.

For parallel programs, one can use HPCToolkit to measure the fraction of time threads are idle, working, or communicating. To obtain detailed information about a program's computation performance, one can collect samples using a processor's built-in performance monitoring units to measure metrics such as operation counts, pipeline stalls, cache misses, and data movement between processor sockets. Such detailed measurements are essential to understand the performance characteristics of applications on modern multicore microprocessors that employ instruction-level parallelism, out-of-order execution, and complex memory hierarchies. With HPCToolkit, one can also easily compute derived metrics such as cycles per instruction, waste, and relative efficiency to provide insight into a program's shortcomings.

A unique capability of HPCToolkit is its ability to unwind the call stack of a thread executing highly optimized code to attribute time, hardware counter metrics, as well as software metrics (e.g., context switches) to a full calling context. Call stack unwinding is often difficult for highly optimized code (Tallent, Mellor-Crummey, and Fagan 2009). For accurate call stack unwinding, HPCToolkit employs two strategies: interpreting compiler-recorded information in DWARF Frame Descriptor Entries (FDEs) and binary analysis to compute unwind recipes directly from an application's machine instructions. On ARM processors, HPCToolkit uses `libunwind` exclusively. On Power processors, HPCToolkit uses binary analysis exclusively. On x86_64 processors, HPCToolkit employs both strategies in an integrated fashion.

HPCToolkit assembles performance measurements into a call path profile that associates the costs of each function call with its full calling context. In addition, HPCToolkit uses binary analysis to attribute program performance metrics with detailed precision – full dynamic calling contexts augmented with information about call sites, inlined functions and templates, loops, and source lines. Measurements can be analyzed in a variety of ways: top-down in a calling context tree, which associates costs with the full calling context in which they are incurred; bottom-up in a view that apportions costs associated with a function to each of the contexts in which the function is called; and in a flat view that aggregates all costs associated with a function independent of calling context. This multiplicity of code-centric perspectives is essential to understanding a program's performance for tuning under various circumstances. HPCToolkit also supports

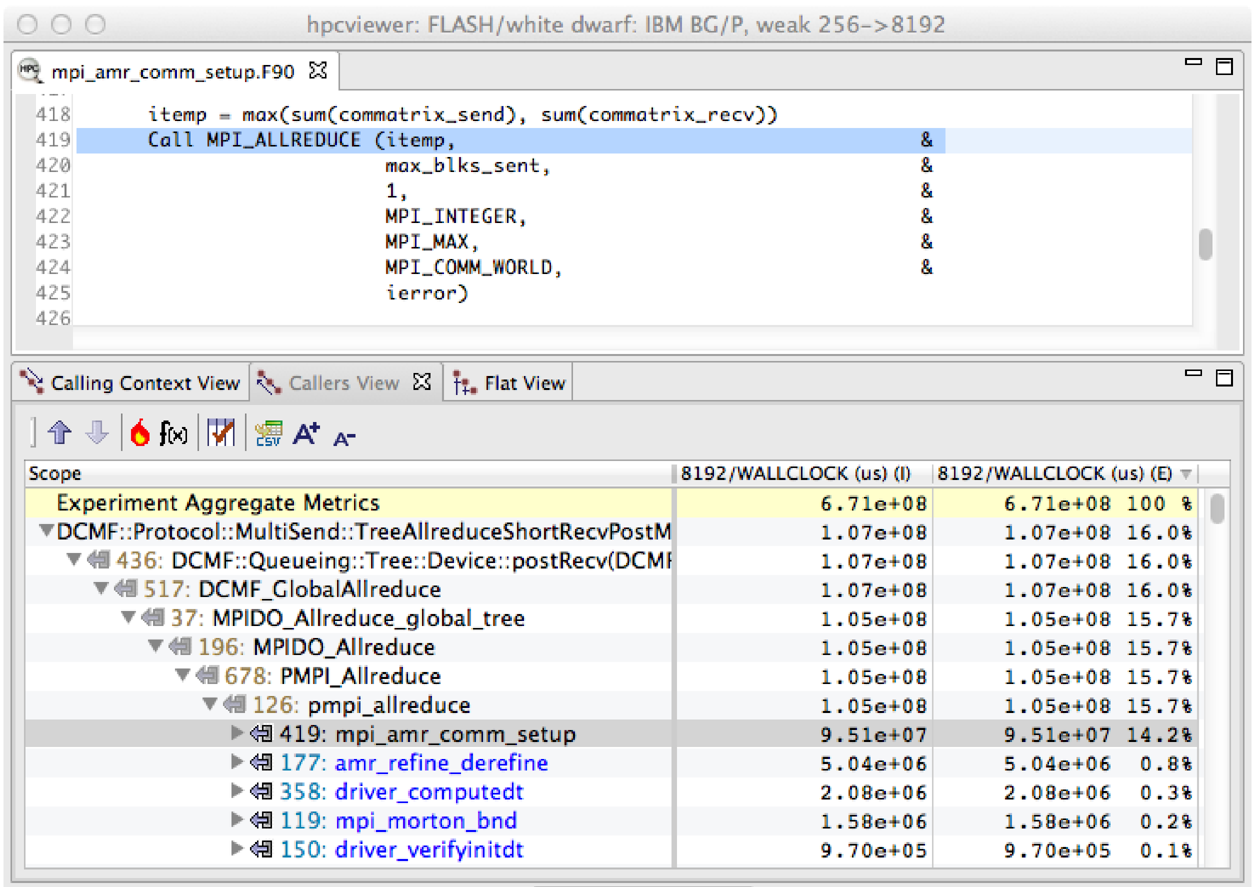


Fig. 1: Figure 1.1: A code-centric view of an execution of the University of Chicago's FLASH code executing on 8192 cores of a Blue Gene/P. This bottom-up view shows that 16% of the execution time was spent in IBM's DCMF messaging layer. By tracking these costs up the call chain, we can see that most of this time was spent on behalf of calls to `pmpi_allreduce` on line 419 of `amr_comm_setup`.

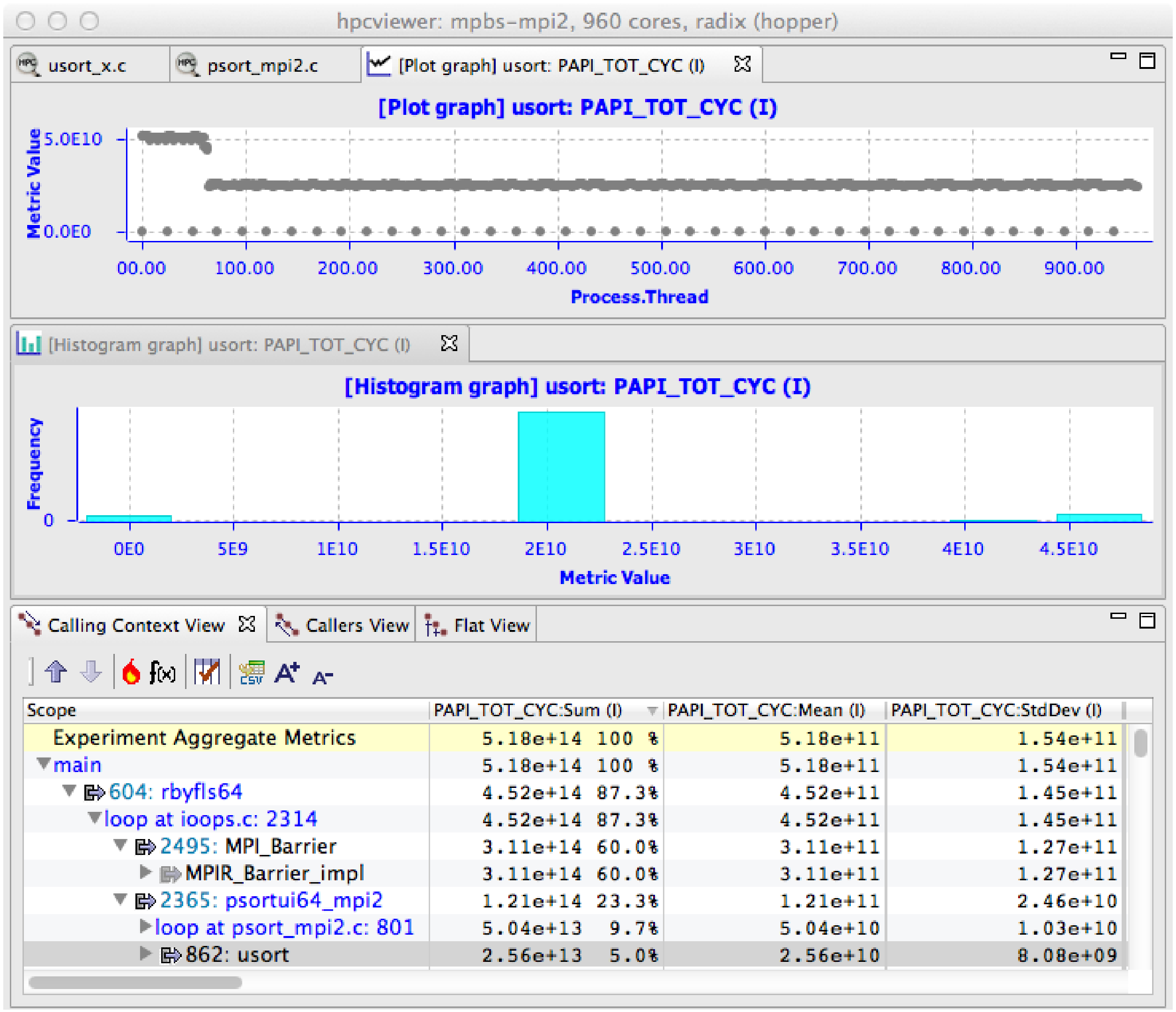


Fig. 2: Figure 1.2: A thread-centric view of the performance of a parallel radix sort application executing on 960 cores of a Cray XE6. The bottom pane shows a calling context for `usort` in the execution. The top pane shows a graph of how much time each thread spent executing calls to `usort` from the highlighted context. On a Cray XE6, there is one MPI helper thread for each compute node in the system; these helper threads spent no time executing `usort`. The graph shows that some of the MPI ranks spent twice as much time in `usort` as others. This happens because the radix sort divides up the work into 1024 buckets. In an execution on 960 cores, 896 cores work on one bucket and 64 cores work on two. The middle pane shows an alternate view of the thread-centric data as a histogram.

a thread-centric perspective, which enables one to see how a performance metric for a calling context differs across threads, and a time-centric perspective, which enables a user to see how an execution unfolds over time. Figures *1.1*–*1.3* show samples of HPCToolkit's code-centric, thread-centric, and time-centric views.

By working at the machine-code level, HPCToolkit accurately measures and attributes costs in executions of multilingual programs, even if they are linked with libraries available only in binary form. HPCToolkit supports performance analysis of fully optimized code. It measures and attributes performance metrics to shared libraries that are dynamically loaded at run time. The low overhead of HPCToolkit's sampling-based measurement is particularly important for parallel programs because measurement overhead can distort program behavior.

HPCToolkit is especially good at pinpointing scaling losses in parallel codes, both within multicore nodes and across the nodes in a parallel system. Using differential analysis of call path profiles collected on different numbers of threads or processes, one can quantify scalability losses and pinpoint their causes to individual lines of code executed in particular calling contexts (Coarfa et al. 2007). We have used this technique to quantify scaling losses in leading science applications across thousands of processor cores on Cray and IBM Blue Gene systems, associate them with individual lines of source code in full calling context (Tallent, Mellor-Crummey, and Fagan 2009; Tallent, Adhianto, and Mellor-Crummey 2010), and quantify scaling losses in science applications within compute nodes at the loop nest level due to competition for memory bandwidth in multicore processors (Tallent et al. 2008). We have also developed techniques for efficiently attributing the idleness in one thread to its cause in another thread (Tallent, Mellor-Crummey, and Fagan 2009; Tallent, Mellor-Crummey, and Porterfield 2010).

HPCToolkit is deployed on many DOE supercomputers, including the El Capitan exacale supercomputer (AMD MI300A APUs) at Lawrence Livermore National Laboratory, the Frontier exascale supercomputer (AMD Trento CPUs + MI250X GPUs) at Oak Ridge National Laboratory, the Aurora exascale supercomputer (Intel Sapphire Rapids CPUs + Intel Max GPUs) at Argonne's Leadership Computing Facility, the Crossroads supercomputer (Intel Sapphire Rapids) at Los Alamos National Laboratory, and the Kestrel HPC platform, which includes both GPU-accelerated nodes (AMD Genoa + NVIDIA H100) and CPU-only nodes (Intel Sapphire Rapids) at the National Renewable Energy Laboratory.

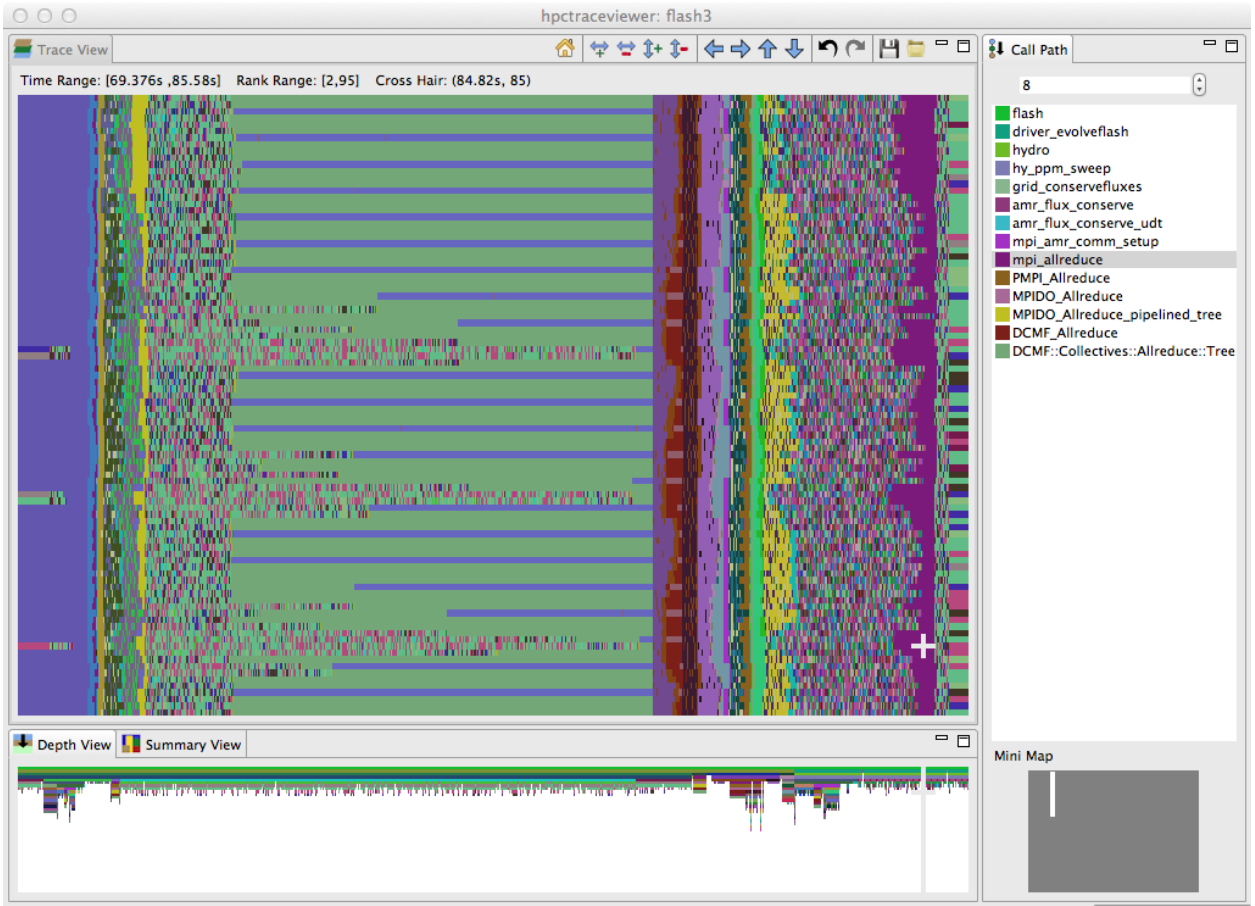


Fig. 3: Figure 1.3: A time-centric view of part of an execution of the University of Chicago's FLASH code on 256 cores of a Blue Gene/P. The figure shows a detail from the end of the initialization phase and part of the first iteration of the solve phase. The largest pane in the figure shows the activity of cores 2–95 in the execution during a time interval ranging from 69.376s–85.58s during the execution. Time lines for threads are arranged from top to bottom and time flows from left to right. The color at any point in time for a thread indicates the procedure that the thread is executing at that time. The right pane shows the full call stack of thread 85 at 84.82s into the execution, corresponding to the selection shown by the white crosshair; the outermost procedure frame of the call stack is shown at the top of the pane and the innermost frame is shown at the bottom. This view highlights that even though FLASH is an SPMD program, the behavior of threads over time can be quite different. The purple region highlighted by the cursor, which represents a call by all processors to `mpi_allreduce`, shows that the time spent in this call varies across the processors. The variation in time spent waiting in `mpi_allreduce` is readily explained by an imbalance in the time processes spend a prior prolongation step, shown in yellow. Further left in the figure, one can see differences among ranks executing on different cores in each node as they await the completion of an `mpi_allreduce`. A rank executing on one core of each node waits in `DCMF_Messager_advance` (which appears as blue stripes) while ranks executing on other cores in each node wait in a helper function (shown in green). In this phase, ranks await the delayed arrival of a few of their peers who have extra work to do inside `simulation_initblock` before they call `mpi_allreduce`.

CHAPTER

TWO

HPCTOOLKIT OVERVIEW

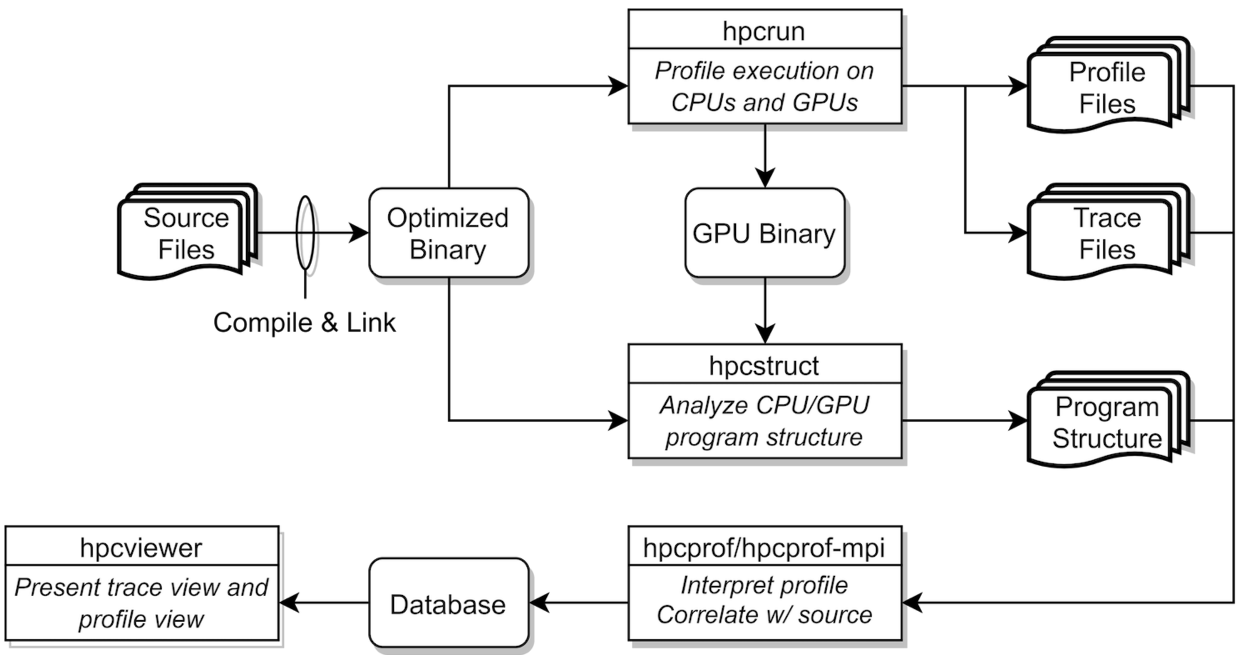


Fig. 1: Figure 2.1: Overview of HPCToolkit's tool work flow.

HPCToolkit's work flow is organized around four principal capabilities, as shown in Figure *2.1*:

1. *measurement* of context-sensitive performance metrics using call-stack unwinding while an application executes;
2. *binary analysis* to recover program structure from the application binary and the shared libraries and GPU binaries used in the run;
3. *attribution* of performance metrics by correlating dynamic performance metrics with static program structure; and
4. *presentation* of performance metrics and associated source code.

To use HPCToolkit to measure and analyze an application's performance, one first compiles and links the application for a production run, using *full* optimization and including debugging symbols.[1] Second, one launches an application

[1] For the most detailed attribution of application performance data using HPCToolkit, one should ensure that the compiler includes line mappings in the object code it generates. While HPCToolkit does not need this information to function, it can be helpful to users trying to interpret the results. Since compilers can usually provide line map information for fully optimized code, this requirement need not require a special build process. While adding a `-g` flag is the right thing to do for many compilers, some compilers provide other flags that provide line mapping and inlining information for profiling without compromising optimization. For instance, when compiling CUDA for NVIDIA GPUs with nvcc, one should use `-lineinfo` rather than `-g` to avoid inadvertently disabling optimization.

with HPCToolkit's measurement tool, `hpcrun`, which uses asynchronous sampling to profile and trace CPU activity and vendor-provided libraries to profile and trace GPU activity. Third, one invokes `hpcstruct`, HPCToolkit's tool for analyzing an application binary and any shared objects and GPU binaries used in a measured execution. `hpcstruct` recovers information about source files, procedures, loops, and inlined code that are found in CPU and GPU binaries. Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's `hpcviewer` graphical user interface.

The rest of this chapter briefly discusses unique aspects of HPCToolkit's measurement, analysis and presentation capabilities.

2.1 Call Path Profiling

Without accurate measurement, performance analysis results may be of questionable value. As a result, a principal focus of work on HPCToolkit has been the design and implementation of techniques to provide accurate fine-grain measurements of production applications running at scale. For tools to be useful on production applications on large-scale parallel systems, large measurement overhead is unacceptable. For measurements to be accurate, performance tools must avoid introducing measurement error.

Both source-level and binary instrumentation can distort application performance through a variety of mechanisms (Mytkowicz et al. 2009). Frequent calls to small instrumented procedures can lead to considerable measurement overhead. Furthermore, source-level instrumentation can distort application performance by interfering with inlining and template optimization. To avoid these effects, many instrumentation-based tools intentionally refrain from instrumenting certain procedures. Ironically, the more this approach reduces overhead, the more it introduces *blind spots*, i.e., intervals of unmonitored execution. For example, a common selective instrumentation technique is to ignore small frequently executed procedures; however, those routines may be mutex lock operations that introduce performance bottlenecks! Sometimes, a tool unintentionally introduces a blind spot. A typical example is that source code instrumentation suffers from blind spots when source code is unavailable, a common condition for vendor-provided math and communication libraries.

To avoid these problems, HPCToolkit eschews instrumentation and favors the use of *asynchronous sampling* to measure and attribute performance metrics. During a program execution, sample events are triggered by periodic interrupts induced by an interval timer or overflow of hardware performance counters. One can sample metrics that reflect work (e.g., instructions, floating-point operations), consumption of resources (e.g., cycles, bandwidth consumed in the memory hierarchy by data transfers in response to cache misses), or inefficiency (e.g., stall cycles). For reasonable sampling frequencies, the overhead and distortion introduced by sampling-based measurement is typically much lower than that introduced by instrumentation (Froyd, Mellor-Crummey, and Fowler 2005).

For all but the most trivially structured programs, it is important to associate the costs incurred by each procedure with the contexts in which the procedure is called. Knowing the context in which each cost is incurred is essential for understanding why the code performs as it does. This is particularly important for code based on application frameworks and libraries. For instance, costs incurred for calls to communication primitives (e.g., `MPI_Wait`) or code that results from instantiating C++ templates for data structures can vary widely depending how they are used in a particular context. Because there are often layered implementations within applications and libraries, it is insufficient either to insert instrumentation at any one level or to distinguish costs based only upon the immediate caller. For this reason, HPCToolkit uses call path profiling to attribute costs to the full calling contexts in which they are incurred.

HPCToolkit's `hpcrun` call path profiler uses call stack unwinding to attribute execution costs of optimized executables to the full calling context in which they occur. Unlike other tools, to support asynchronous call stack unwinding during execution of optimized code, `hpcrun` uses on-the-fly binary analysis to locate procedure bounds and compute an unwind recipe for each code range within each procedure (Tallent, Mellor-Crummey, and Fagan 2009). These analyses enable `hpcrun` to unwind call stacks for optimized code with little or no information other than an application's machine code.

The output of a run with `hpcrun` is a *measurements directory* containing the data, and the information necessary to recover the names of all shared libraries and GPU binaries.

2.2 Recovering Program Structure

To enable effective analysis, call path profiles for executions of optimized programs must be correlated with important source code abstractions. Since measurements refer only to instruction addresses within the running application, it is necessary to map measurements back to the program source. The mappings include those of the application and any shared libraries referenced during the run, as well as those for any GPU binaries executed on GPUs during the run. To associate measurement data with the static structure of fully-optimized executables, we need a mapping between object code and its associated source code structure.[2] HPCToolkit constructs this mapping using binary analysis; we call this process *recovering program structure* (Tallent, Mellor-Crummey, and Fagan 2009).

HPCToolkit focuses its efforts on recovering source files, procedures, inlined functions and templates, as well as loop nests as the most important elements of source code structure. To recover program structure, HPCToolkit's `hpcstruct` utility parses a binary's machine instructions, reconstructs a control flow graph, combines line map and DWARF information about inlining with interval analysis on the control flow graph in a way that enables it to relate machine code after optimization back to the original source.

One important benefit accrues from this approach. HPCToolkit can expose the structure of and assign metrics to the code is actually executed, *even if source code is unavailable*. For example, `hpcstruct`'s program structure naturally reveals transformations such as loop fusion and scalarization loops that arise from compilation of Fortran 90 array notation. Similarly, it exposes calls to compiler support routines and wait loops in communication libraries of which one would otherwise be unaware.

2.3 Attributing Performance to Code

HPCToolkit combines (post-mortem) the recovered static program structure with dynamic call paths to expose inlined frames and loop nests. This enables us to attribute the performance of samples in their full static and dynamic context and correlate it with source code.

HPCToolkit's `hpcprof` utility employs multithreading to quickly aggregate performance measurements from multiple threads and processes and attribute them to application and/or library CPU and GPU source code.

One uses `hpcprof` by invoking it on the *measurements directory* recorded by `hpcrun` and augmented with program structure information by `hpcstruct`. From the measurements and structure, `hpcprof` generates a *database directory* containing performance data presentable by `hpcviewer`.

In most cases `hpcprof` is able to complete the reduction in minutes, however for especially large experiments (e.g. thousands of threads or GPU streams (Anderson, Liu, and Mellor-Crummey 2022)) its multi-node sibling `hpcprof-mpi` may be substantially faster. `hpcprof-mpi` is an MPI application identical to `hpcprof`, except that employs distributed-memory parallelism in addition to multithreading to aggregate and attribute performance measurements for many CPUs and GPUs. In our experience, exploiting 8-10 compute nodes via `hpcprof-mpi` can be as much as 5x faster than `hpcprof` for analyzing performance measurements from thousands of CPUs and GPUs.

2.4 Interactive Performance Analysis

To enable an analyst to rapidly pinpoint and quantify performance bottlenecks, tools must present the performance measurements in a way that engages the analyst, focuses attention on what is important, and automates common analysis subtasks to reduce the mental effort and frustration of sifting through a sea of measurement details.

To enable rapid analysis of an execution's performance bottlenecks, we have carefully designed HPCToolkit's `hpcviewer` graphical user interface. `hpcviewer` provide code-centric profile views (Adhianto, Mellor-Crummey, and Tallent 2010), thread-centric graphing of metrics, and time-centric views of CPU and GPU activity (Tallent et al. 2011).

[2] This object to source code mapping should be contrasted with a binary's line map, which (if present) only maps machine instructions to source lines.

`hpcviewer` combines a relatively small set of complementary presentation techniques that, taken together, rapidly focus an analyst's attention on performance bottlenecks rather than on unimportant information. To facilitate the goal of rapidly focusing an analyst's attention on performance bottlenecks `hpcviewer` extends several existing presentation techniques. In its code-centric view, `hpcviewer` (1) synthesizes and presents top-down, bottom-up, and flat views of calling-context-sensitive metrics; (2) treats a procedure's static structure as first-class information with respect to both performance metrics and constructing views; (3) enables a large variety of user-defined metrics to describe performance inefficiency; and (4) automatically expands hot paths based on arbitrary performance metrics — through calling contexts and static structure — to rapidly highlight important performance data.

`hpcviewer`'s time-centric trace tab enables an application developer to visualize how a parallel execution unfolds over time. This view makes it easy to spot key inefficiencies such as serialization and load imbalance.

CHAPTER

THREE

QUICK START

This chapter provides a rapid overview of analyzing the performance of an application using HPCToolkit. It assumes an operational installation of HPCToolkit.

3.1 Guided Tour

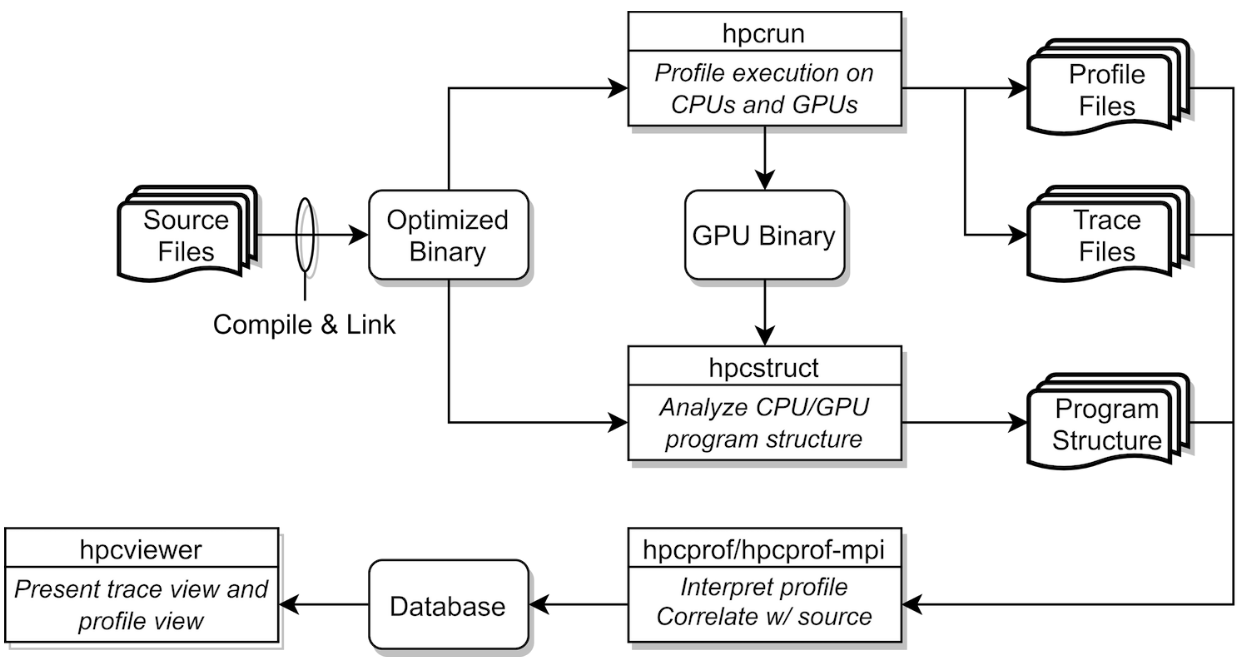


Fig. 1: Figure 3.1: Overview of HPCToolkit tool's work flow.

HPCToolkit's work flow is summarized in Figure 3.1 and is organized around four principal capabilities:

1. *measurement* of context-sensitive performance metrics while an application executes;
2. *binary analysis* to recover program structure from CPU and GPU binaries;
3. *attribution* of performance metrics by correlating dynamic performance metrics with static program structure; and
4. *presentation* of performance metrics and associated source code.

To use HPCToolkit to measure and analyze an application's performance, one first compiles and links the application for a production run, using *full* optimization. Second, one launches an application with HPCToolkit's measurement tool, `hpcrun`, which uses statistical sampling to collect a performance profile. Third, one applies `hpcstruct` to an application's measurement directory to recover program structure information from any CPU or GPU binary that was measured. Program structure, which includes information about files, procedures, inlined code, and loops, is used to relate performance measurements to source code. Fourth, one uses `hpcprof` to combine information about an application's structure with dynamic performance measurements to produce a performance database. Finally, one explores a performance database with HPCToolkit's graphical user interface: `hpcviewer` which presents both a code-centric analysis of performance metrics and a time-centric (trace-based) analysis of an execution.

The following subsections explain HPCToolkit's work flow in more detail.

3.1.1 Compiling an Application

For the most detailed attribution of application performance data using HPCToolkit, one should compile so as to include with line map information in the generated object code. This usually means compiling with options similar to `-g -O3`. Check your compiler's documentation for information about the right set of options to have the compiler record information about inlining and the mapping of machine instructions to source lines. We advise picking options that indicate they will record information that relates machine instructions to source code without compromising optimization. Additionally, specifying flags that cause a compiler to record mappings of machine instructions to inlined code are particularly useful for performance analysis of codes that employ C++ templates.

While HPCToolkit does not need information about the mapping between machine instructions and source code to function, having such information included in the binary code by the compiler can be helpful to users trying to interpret performance measurements. Since compilers can usually provide information about line mappings and inlining for fully-optimized code, this requirement usually involves a one-time trivial adjustment to the an application's build scripts to provide a better experience with tools. Such mapping information enables tools such as HPCToolkit to attribute performance metrics to source code constructs within procedures rather than only at the procedure level.

3.1.2 Measuring Application Performance

Today, HPCToolkit is designed to measure executions of dynamically-linked applications. To monitor a dynamically linked application, simply use `hpcrun` to launch the application. An application may be sequential, multithreaded or based on MPI. The commands below give examples for an application named `app`.

- Dynamically linked applications:

  Simply launch your application with `hpcrun`:

  ```
  [<mpi-launcher>] hpcrun [hpcrun-options] app [app-arguments]
  ```

  Of course, `<mpi-launcher>` is only needed for MPI programs and is sometimes a program like `mpiexec` or `mpirun`, or a workload manager's utilities such as Slurm's `srun` or IBM's Job Step Manager utility `jsrun`.

hpcrun will produce a measurements database that contains separate measurement information for each MPI rank and thread in the application. The database is named according the form:

```
hpctoolkit-app-measurements[-<jobid>]
```

If the application `app` is run under control of a recognized batch job scheduler (such as Slurm, Cobalt, or IBM's Job Manager), the name of the measurements directory will contain the corresponding job identifier `<jobid>`. Currently, the database contains measurements files for each thread that are named using the following templates:

```
app-<mpi-rank>-<thread-id>-<host-id>-<process-id>.<generation-id>.hpcrun
app-<mpi-rank>-<thread-id>-<host-id>-<process-id>.<generation-id>.hpctrace
```

Specifying CPU Sample Sources

HPCToolkit primarily monitors an application using asynchronous sampling. Consequently, the most common option to `hpcrun` is a list of sample sources that define how samples are generated. A sample source takes the form of an event name `e` and `howoften`, specified as `e@howoften`. The specifier `howoften` may be a number, indicating a period, e.g. `CYCLES@4000001` or it may be `f` followed by a number, `CYCLES@f200` indicating a frequency in samples/second. For a sample source with event `e` and period `p`, after every `p` instances of `e`, a sample is generated that causes `hpcrun` to inspect the and record information about the monitored application.

To configure `hpcrun` with two samples sources, `e1@howoften1` and `e2@howoften2`, use the following options:

```
--event e1@howoften1 --event e2@howoften2
```

Measuring GPU Computations

One can simply profile and optionally trace computations offloaded onto AMD, Intel, and NVIDIA GPUs by using one of the following event specifiers:

- `-e gpu=cuda` is used with CUDA and OpenMP on NVIDIA GPUs
- `-e gpu=rocm` is used with HIP and OpenMP on AMD GPUs
- `-e gpu=level0` is used with Intel's Level Zero runtime for Data Parallel C++ and OpenMP
- `-e gpu=opencl` can be used on any of the GPU platforms.

Adding a `-t` to `hpcrun`'s command line when profiling GPU computations will trace them as well. `hpcrun` also supports a `-tt` option for boosted resolution tracing of GPU-accelerated programs. Using this flag causes the calling context of CPU threads to be recorded in their traces each time they launch a GPU operation. This option is particularly helpful when tracing codes that frequently launch GPU operations.

More information about monitoring GPU-accelerated applications, including information about how to monitor activity within GPU kernels, is available in a later chapter *Measurement and Analysis of GPU-accelerated Applications*.

3.1.3 Recovering Program Structure

When `hpcrun` measures the performance of an application, it associates performance information with machine code addresses. To relate performance information associated with machine code addresses back to source code locations, HPCToolkit provides a tool `hpcstruct` that analyzes CPU and GPU binaries associated with an execution to *recover program structure*. Program structure for a binary includes information about its source files, procedures, inlined code, loop nests, and statements. Program structure for a binary is recovered by combining source code mapping information recorded by compilers (when apprpropriate compile time arguments, e.g. `-g`, are used) with information about loops that `hpcstruct` gleans from analyzing the control flow between machine instructions in the binary.

Typically, one launches `hpcstruct` without any options and one argument that specifies a HPCToolkit *measurement directory*.

```
hpcstruct hpctoolkit-app-measurements
```

From a measurement directory, `hpcstruct` identifies the application, any shared libraries it loads, and any GPU binaries that it invokes. `hpcstruct` processes each of CPU and GPU binary and records information about its program structure into the *measurements directory*. Program structure for a binary includes information about its source files, procedures, inlined code, loop nests, and statements.

When applied to a measurements directory, `hpcstruct` analyzes multiple binaries concurrently. The amount of parallelism used to analyze each binary is proportional to the size of the binary. By default, `hpcstruct` uses a pool of threads equal to half of the available hardware threads. If you want to adjust this value (higher or lower), use the `-j|--jobs` option.

```
hpcstruct -j <num-jobs> measurements-directory
```

Although not usually necessary, one can apply `hpcstruct` to recover program structure information for a single CPU or GPU binary. To recover static program structure for a single binary `b`, use the command:

```
hpcstruct b
```

This command analyzes the binary and saves this information in a file named `b.hpcstruct`.

Tip

On rare occasions, there may be a module file for which `hpcstruct` takes an excessive amount of time to analyze.

When this happens, if the module file is not essential to understanding the application's performance, you can skip that module with the `-x|--exclude` option. Specify the module's path name or file name. For example:

```
hpcstruct -x /path/to/file-name measurements-directory
hpcstruct -x file-name measurements-directory
```

By default, `hpcstruct` skips certain known files. If you need detailed analysis of a binary that would otherwise be skipped, use the `-i|--include` option.

```
hpcstruct -i file-name measurements-directory
```

You can see a list of all binaries associated with an execution's measurements directory with the `--show-files` option.

```
hpcstruct --show-files measurements-directory
```

In practice, `hpcstruct` analyzes most CPU or GPU binaries in less than a minute, so these options are normally not needed.

Caching Structure Results

`Hpcstruct` can cache the results of the structure files. This can be useful if you profile your application multiple times but most of its module files are unchanged.

The simplest way to use the cache is to set the `HPCTOOLKIT_HPCSTRUCT_CACHE` environment variable to a directory where you want to store the cache. For example (using `bash`),

```
export HPCTOOLKIT_HPCSTRUCT_CACHE=/path/to/cache/directory
```

While caching structure results is useful to avoid repeating analysis of common files and we recommend this for all users, note that `hpcstruct` does not have a default location for `HPCTOOLKIT_HPCSTRUCT_CACHE`. This is to avoid inadvertently capturing information about applications with restricted access (e.g. export controlled codes) and saving it to an inappropriate location in the filesystem.

You can also specify the cache directory on the command line with the `-c|--cache` option. This option might be attractive if you are analyzing measurement data for an application with restricted access and want to control where information about it is saved.

```
hpcstruct -c /path/to/cache/directory measurements-directory
```

Warning

Caching the structure files involves copying the CPU and GPU binaries and libraries into the cache directory. If all or part of your application has restricted access, be careful where you put the cache directory and be sure to restrict the directory permissions.

3.1.4 Attributing Performance to Code

To analyze HPCToolkit's measurements and attribute them to the application's source code, use `hpcprof`, typically invoked as follows:

```
hpcprof hpctoolkit-app-measurements
```

This command will produce an HPCToolkit performance database with the name `hpctoolkit-app-database`. If this database directory already exists, `hpcprof` will form a unique name by appending a random hexadecimal qualifier.

`hpcprof` performs this analysis in parallel using multithreading. By default all available threads are used. If this is not wanted (e.g. using sharing a single machine), the thread count can be specified with `-j <threads>`.

`hpcprof` usually completes this analysis in a matter of minutes. For especially large experiments (applications using thousands of threads and/or GPU streams), the sibling `hpcprof-mpi` may produce results faster by exploiting additional compute nodes[3]. Typically `hpcprof-mpi` is invoked as follows, using 8 ranks and compute nodes:

```
<mpi-launcher> -n 8 hpcprof-mpi hpctoolkit-app-measurements
```

Note that additional options may be needed to grant `hpcprof-mpi` access to all threads on each node, check the documentation for your scheduler and MPI implementation for details.

If possible, `hpcprof` will copy the sources for the application and any libraries into the resulting database. If the source code was moved since or was mounted at a different location than when the application was compiled, the resulting database may be missing some important source files. In these cases, the `-R/--replace-path` option may be specified to provide substitute paths based on prefixes. For example, if the application was compiled from source at `/home/joe/app/src/` but it is mounted at `/extern/homes/joe/app/src/` when running `hpcprof`, the source files can be made available by invoking `hpcprof` as follows:

```
hpcprof -R `/home/joe/app/src/=/extern/homes/joe/app/src/' \
  hpctoolkit-app-measurements
```

Note that on systems where MPI applications are restricted to a scratch file system, it is the users responsibility to copy any wanted source files and make them available to `hpcprof`.

3.1.5 Interactive Performance Analysis

To interactively view and analyze an HPCToolkit performance database, use `hpcviewer`. `hpcviewer` may be launched from the command line or by double-clicking on its icon on MacOS or Windows. The following is an example of launching from a command line:

```
hpcviewer hpctoolkit-app-database
```

Additional help for `hpcviewer` can be found in a help pane available from `hpcviewer`'s *Help* menu.

[3] We recommend running `hpcprof-mpi` across 8-10 compute nodes. More than this may not improve or may degrade the overall speed of the analysis.

3.1.6 Effective Analysis Strategies

To effectively analyze application performance, consider using one of the following strategies, which are described in more detail in Chapter *4*.

- A waste metric, which represents the difference between achieved performance and potential peak performance is a good way of understanding the potential for tuning the node performance of codes (Section *11.3*). `hpcviewer` supports synthesis of derived metrics to aid analysis. Derived metrics are specified within `hpcviewer` using spreadsheet-like formula. See the `hpcviewer` help pane for details about how to specify derived metrics.
- Scalability bottlenecks in parallel codes can be pinpointed by differential analysis of two profiles with different degrees of parallelism (Section *11.4*).

3.2 Additional Guidance

For additional information, consult the rest of this manual and other documentation: First, we summarize the available documentation and command-line help:

Command-line help:
Each of HPCToolkit's command-line tools can generate a help message summarizing the tool's usage, arguments and options. To generate this help message, invoke the tool with `-h` or `--help`.

Man pages:
Man pages are available online or in a local HPCToolkit installation (`<hpctoolkit-installation>/share/man`).

Manuals:
Manuals are available either online or from a local HPCToolkit installation (`<hpctoolkit-installation>/share/doc/hpctoolkit/documentation.html`).

Papers:
Papers that describe various aspects of HPCToolkit's measurement, analysis, attribution and presentation technology can be found on the HPCToolkit website.

CHAPTER

FOUR

INSTALLING WITH SPACK

This chapter outlines how to build and install HPCToolkit and Hpcviewer with Spack. If you are unfamiliar with Spack, see the section below *Running Spack for the First Time*.

HPCToolkit proper (`hpcrun`, `hpcstruct` and `hpcprof`) is used to measure and analyze an application's performance and then produce a database for `hpcviewer`. We distribute HPCToolkit from source on GitLab and support building on 64-bit Linux on x86_64, little-endian powerpc (power8, 9 and 10) and ARM (aarch64) with spack.

We provide binary distributions for `hpcviewer` on Linux (x86_64, ppc64/le and aarch64), Windows (x86_64) and MacOS (x86_64, M1, M2 and M3). Although `hpcviewer` is normally included as part of HPCToolkit, we provide `hpcviewer` as a separate package to install on platforms, such as MacOS and Windows laptops, where HPCToolkit proper is not available. HPCToolkit databases are platform-independent and it is common to run `hpcrun` on one machine and then view the results on another machine.

Building `hpctoolkit` requires a fairly recent C/C++ compiler that supports C++17 and common GNU extensions (including LLVM). GNU gcc/g++ 12.x or later should do fine. Spack requires Python 3.8 or later plus various build utilities: bash, git, curl, patch, tar, bzip2, unzip, etc.

`Hpcviewer` requires Java 17 or later. Without any configuration, Spack will automatically install an appropriate version of Java OpenJDK to support `hpcviewer`.

When installing `hpctoolkit` using Spack, `hpcviewer` will be installed automatically by default.

Gitlab repositories:

- https://gitlab.com/hpctoolkit/hpctoolkit
- https://gitlab.com/hpctoolkit/hpcviewer

Spack Documentation:

- https://spack.readthedocs.io/en/latest

4.1 Config Files

Spack uses a variety of config files for setting paths, options, etc. The simplest way to set or change a value is to copy the default file from `spack/etc/spack/defaults` one directory up and make the change there.

A complete discussion of this topic is available in Spack's documentation about configuration files. Here, we only mention a few key configuration details that you might want to adjust when installing HPCToolkit with Spack.

4.1.1 Config.yaml

In `config.yaml`, you may want to set the path to the install tree.

```yaml
config:
  # This is the path to the root of the Spack install tree.
  install_tree:
    root:  /path/to/root/of/install/tree
```

4.1.2 Modules.yaml

If you are using modules (TCL or Lmod), you need to `enable` the module type and provide the path to the modules directory.

```yaml
modules:
  ...
  default:
    # Normally only need one of these
    roots:
      tcl:   /path/to/tcl/modules/directory
      lmod:  /path/to/lmod/modules/directory
    # What type of modules to use ("tcl" and/or "lmod")
    enable:
     - tcl
     - lmod
```

Important

You should disable the module `autoload` feature for `hpctoolkit`. HPCToolkit does not need its dependency modules loaded and loading them may interfere with your application's dependencies. Do this for both `tcl` and `lmod` modules (whichever one you are using).

```yaml
modules:
  ...
  default:
    ...
    tcl:
      hpctoolkit:
        autoload: none
      all:
        autoload: direct
    lmod:
      hpctoolkit:
        autoload: none
      all:
        autoload: direct
```

Tip

If you forget to turn off `autoload` for `hpctoolkit`, you don't have to start over from scratch. You can uninstall `hpctoolkit`, fix the `modules.yaml` file and then reinstall `hpctoolkit`. Or, you could just hand edit the module file.

4.2 Installing HPCToolkit

4.2.1 Basic Installation

Here we explain how to install a basic version of HPCToolkit with Spack. The directions immediately below build and install a version of HPCToolkit that profiles only a program's CPU activity. To install a GPU-aware version of HPC-Toolkit or configure HPCToolkit for post-mortem analysis of thousands of profiles, see Section *4.2.2* for a description of how to enable optional HPCToolkit features when building with Spack.

Although spack can install packages with a single `spack install` command, we recommend that you always run `spack spec` first to verify what spack is going to install.

```
spack spec hpctoolkit
```

Check that the version and variants for `hpctoolkit` and `dyninst` and the compiler and `arch` type are what you want. If you are adding support for a GPU or `hpcprof-mpi`, check that they are included and are the version that you intend. Then, install HPCToolkit with:

```
spack install hpctoolkit
```

Important

Normally, Spack builds HPCToolkit for the specific architecture (skylake, sapphirerapids, zen3, etc) on which `spack install hpctoolkit` is run. Some care is required when installing HPCToolkit on a system that contains multiple kinds of x86_64 processors. A default Spack build of HPCToolkit on one kind of x86_64 processor and running on another may cause `illegal instruction` errors.

It is worth noting that in some cases, processor heterogeneity is less than obvious. For example, on Argonne's Aurora supercomputer, login nodes use Intel Icelake processors while compute nodes use Intel Sapphire Rapids processors.

You can avoid problems when multiple kinds of x86_64 processors are present by configuring HPCToolkit for a generic x86_64 processor as follows:

```
spack install hpctoolkit target=x86_64
```

HPCToolkit can also be configured for a generic x86_64 target in `packages.yaml` with:

```
packages:
  all:
    require: "target=x86_64"
```

You can see a list Spack's targets and families with `spack arch`.

```
spack arch --known-targets
```

For further information, see Spack's documentation about architecture specifiers.

Tip

Configuring HPCToolkit for debugging By default, Spack will build a `release` version of `hpctoolkit` (full optimization and no debug info). If you want a `debug` version, use the `buildtype` variant.

`spack install hpctoolkit buildtype=debug`

Possible values for `buildtype` are `release` (default), `debug`, `debugoptimized`, `minsize` and `plain`.

Tip

If you encounter problems with the latest release of HPCToolkit, you might trying the in-development versions of HPCToolkit and Dyninst; namely, `hpctoolkit@develop` and `dyninst@master`:

`spack install hpctoolkit@develop ^dyninst@master`

4.2.2 Configuration Options

HPCToolkit supports a variety of configuration options. Multiple (or even all) options can be enabled in a single installation of HPCToolkit on a platform with the necessary hardware and/or software prerequisites.

CUDA (+cuda)

HPCToolkit supports profiling NVIDIA GPUs through the CUDA interface. You can either use an existing CUDA module or let Spack build one.

To use an existing CUDA module (recommended), load the `cuda` module and run `spack external find`. For example:

```
module load cuda/12.9
spack external find cuda
```

This should create an entry in `packages.yaml` similar to the following. If not, then add an entry manually.

```
packages:
  cuda:
    externals:
    - spec: cuda@12.9
      prefix: /usr/local/cuda-12.9
```

Then, build `hpctoolkit` with `+cuda`.

```
spack install hpctoolkit +cuda
```

Level Zero (+level_zero)

Beginning with 2025.0, HPCToolkit has basic support for Intel GPUs (start and stop times for GPU kernels) through the Intel Level Zero interface. Build `hpctoolkit` with `+level_zero`.

```
spack install hpctoolkit +level_zero
```

Advanced support to see inside the GPU kernels requires the `oneapi-igc` (Intel Graphics Compiler) package. This is an externals only package, Spack does not download or install it. Normally, this is installed in `/usr` and you should manually add a Spack externals entry for it.

```
packages:
  oneapi-igc:
    externals:
    - spec: oneapi-igc@1.0.10409
      prefix: /usr
```

Then build `hpctoolkit` with both `+level_zero` and `+gtpin`.

```
spack install hpctoolkit +level_zero +gtpin
```

We recommend always building with gtpin if possible and then deciding at runtime which options to use.

ROCm (`+rocm`)

HPCToolkit supports profiling computations on AMD GPUs atop the ROCm runtime. HPCToolkit's support for ROCm can be built using an existing ROCm installation on your system or you can have Spack build a version of ROCm components that are compatible with firmware and kernel drivers installed on your system.

> **Important**
>
> This release of HPCToolkit is based on AMD's Rocprofiler-sdk library, which is available only for ROCm 6.2+.
>
> Support for older versions of ROCm requires a release of HPCToolkit prior to 2025.1.0.
>
> Light testing of compatibility across ROCm versions indicates that a version of HPCToolkit built with Rocprofiler-sdk 7.0+ is usable with ROCm versions 6.3+.
>
> ROCm 6.2 needs its own build of HPCToolkit because of incompatible changes to Rocprofiler-sdk in later ROCm versions.
>
> In our experience, Rocprofiler-sdk is not yet fully portable across ROCm versions. To use PC sampling on AMD GPUs for instruction-level performance measurement within GPU kernels, we have found that it is best to build HPCToolkit with the Rocprofiler-sdk from the ROCm release used by your application. If you encounter issues that seem related to cross-version compatibility, you can use Spack to build a copy of HPCToolkit with the same ROCm version as your application.

For ROCm support, HPCToolkit uses two Spack packages: `hip` and `rocprofiler-sdk`. Rather than having Spack build these dependences, we recommend that you use an existing ROCm installation on your platform. These are typically found in paths like `/opt/rocm-x.y.z` where `x.y.z` represents the major version, minor version, and patch version.

We recommend manually adding dependences on a particular version of `hip` and `rocprofiler-sdk` into your Spack `packages.yaml` file as shown below, where `7.1.0` in each of the dependencies would be replaced with a ROCm version installed on your platform.

```
packages:
  hip:
    buildable: false
    externals:
    - spec: hip@7.1.0
      prefix: /opt/rocm-7.1.0
  rocprofiler-sdk:
    buildable: false
    externals:
```

(continues on next page)

(continued from previous page)

```
  - spec: rocprofiler-sdk@7.1.0
    prefix: /opt/rocm-7.1.0
```

Note

Recent versions of AMD's Rocprofiler-sdk library support instruction-level performance monitoring within GPU kernels using PC sampling. Rocprofiler-sdk supports two different implementations of PC sampling. One is software based; the other is hardware based. The availability of PC sampling on a platform depends on the GPU model, the GPU firmware versions installed, and the version of ROCm installed.

Software-based PC sampling is only supported by HPCToolkit on MI200+ GPUs and APUs for ROCm 6.4+. Hardware-based PC sampling is only available on MI300+ GPUs and APUs for ROCm 7+.

With those dependencies in place, you can then build `hpctoolkit` with `+rocm`.

```
spack install hpctoolkit +rocm
```

OpenCL (+opencl)

For all three GPU types, an application can access the GPU through the native interface (CUDA, ROCm, Level Zero) or through the OpenCL interface. To add support for OpenCL, use the `+opencl` variant in addition to the native interface.

For example, with CUDA:

```
spack install hpctoolkit +cuda +opencl
```

We recommend adding opencl support for all GPU types.

MPI (+mpi)

HPCToolkit always supports profiling MPI applications. The Spack variant `+mpi` is for building `hpcprof-mpi`, the MPI version of `hpcprof`. If you want to build `hpcprof-mpi`, then you need to supply an installation of MPI.

```
spack install hpctoolkit +mpi
```

Normally, for systems with compute nodes, you should use an existing MPI module that was built for the correct interconnect for your system and add this to `packages.yaml`. The MPI module should be built with the same C/C++ compiler used to build `hpctoolkit` (to keep the C++ libraries in sync). For example:

```
packages:
  mpi:
    require: "mpich@4.1.2"
  mpich:
    externals:
    - spec: mpich@4.1.2
      modules:
      - mpich/4.1.2
```

Note

This version of MPI is for `hpcprof-mpi` only and is entirely separate from any MPI in your application.

PAPI (+papi)

HPCToolkit allows a developer to access a processor's Hardware Performance Counters with the University of Tennessee's Performance Application Programming Interface, known as PAPI. PAPI runs on top of the perfmon library (`libpfm4`) and uses its own, internal (but slightly out of date) copy of perfmon. Building with `+papi` (default) supports access to counters with either PAPI or perfmon events.

If you want to disable PAPI and use the latest perfmon instead, then build hpctoolkit with `~papi`.

```
spack install hpctoolkit ~papi
```

Python (+python)

HPCToolkit supports profiling Python scripts and attributing samples to Python source functions and not the Python interpreter. Use the `+python` variant.

```
spack install hpctoolkit +python
```

You should use python 3.10 or later and use the same version as the application.

4.3 Installing Hpcviewer

HPCToolkit's `hpcviewer` user interface is installed by default when installing `hpctoolkit` with Spack. However, `hpcviewer` is also available as a separate package. If you want to run `hpcviewer` on a platform (e.g., a laptop) without installing all of `hpctoolkit`, then you can install just the `hpcviewer` package.

```
spack install hpcviewer
```

We provide binary distributions for HPCToolkit's `hpcviewer` graphical user interface on Linux (x86_64, ppc64/le and aarch64), Windows (x86_64) and MacOS (x86_64 and Apple M-series processors). `Hpcviewer` is implemented using Java. Rather than requiring a compatible Java installation on a platform, `hpcviewer` is bundled with a Java runtime environment that meets its needs.

Note

HPCToolkit databases are platform-independent. It is common to use `hpcrun` to measure an application's performance on one machine and then view performance analysis results with `hpcviewer` on another machine.

4.4 Spack for Beginners

If you have never run Spack before, then follow these steps to set up your system. Some of these steps may take several minutes for the first time while Spack initializes its files.

Start by cloning Spack from GitHub.

```
git clone https://github.com/spack/spack.git
```

Source the Spack setup script for your shell. This adds `spack` to your `PATH` and sets up support for `spack load`. For example, for `bash`:

```
cd spack/share/spack
. setup-env.sh
```

If /usr/bin/python3 is older than 3.8 or 3.9, then find or install a later version and set SPACK_PYTHON to that path. This is the version of python3 used to run the Spack scripts. If a package uses python as a dependency, then that is separate. For example:

```
export SPACK_PYTHON=/usr/bin/python3.11
```

Run spack info on a small package. This will trigger cloning the Spack packages repository.

```
spack info zlib
```

Run spack compiler find to search for available compilers on your system.

```
spack compiler find
```

Run spack solve or spack spec on a small package. This will cause Spack to install the concretizer, Spack's main engine for resolving specs.

```
spack solve zlib
```

Then you're ready to start installing packages. Edit config.yaml and modules.yaml as described above, run spack external find as needed, and then install hpctoolkit.

If your system compiler is too old, you can have Spack build a later one. For example, on one RHEL 8.x system, /usr/bin/gcc is 8.5.0. I used that to build gcc 14.3.0 and then used gcc 14.3.0 to build hpctoolkit.

```
spack install gcc@14.3.0 %gcc@8.5.0
spack load gcc@14.3.0    (or module load)
spack compiler find
spack install hpctoolkit %gcc@14.3.0
```

For more information about using Spack, consult Spack's Getting Started documentation.

CHAPTER

FIVE

MESON FOR DEVELOPERS

This chapter describes how to build HPCToolkit from source using Meson. This approach is recommended for HPC-Toolkit developers.

5.1 Quickstart

Install:

- `meson` >=1.6.0 and a working C/C++ compiler (GCC, Clang, etc.)
- Boost >=1.71.0
- `make`, `awk`, and `sed` for `hpcstruct` measurement directory support (see #704).
- (Optional but highly recommended:) ccache
- (Optional:) system-wide *installation of HPCToolkit's dependencies as root*

Then run:

```
$ meson setup builddir/
$ cd builddir/
$ meson compile  # -OR- ninja
$ meson test     # -OR- ninja test
```

Do not use `meson install` or `ninja install`. Instead, working versions of the tools themselves are available using `meson devenv`. One can use `meson devenv` to either run an individual command, or create a subshell environment in which HPCToolkit's commands are available, as shown below:

```
$ meson devenv hpcrun --version  # -OR-
$ meson devenv
[builddir] $ hpcrun --version
```

5.2 Configuration

Configuration arguments can be passed to the initial `meson setup` or later `meson configure` invocations with invocations as follows `meson setup [arguments] builddir/` or `meson configure [arguments]`. Arguments that control the build of HPCToolkit are listed below:

- `-Dtests=(disabled|auto|enabled)`: Enable unit tests with `meson test`. (`disabled` only disables tests that require additional dependencies.)
- `-Dmanpages=(disabled|auto|enabled)`: Generate and build man pages. Requires Docutils.
- `-Dmanual=(disabled|auto|enabled)`: Generate and build user manual. Requires Sphinx.

- `-Dhpcprof_mpi=(disabled|auto|enabled)`: Build `hpcprof-mpi` in addition to `hpcprof`. Requires MPI.
- `-Dpython=(disabled|auto|enabled)`: Enable an (experimental) Python unwinder. Requires Python headers.
- `-Dpapi=(disabled|auto|enabled)`: Enable PAPI metrics. Requires PAPI.
- `-Dcuda=(disabled|auto|enabled)`: Enable CUDA metrics (`-e gpu=nvidia`). Requires CUDA.
- `-Dlevel0=(disabled|auto|enabled)`: Enable Level Zero metrics (`-e gpu=level0`). Requires Level Zero.
- `-Dgtpin=(disabled|auto|enabled)`: Also enable Level Zero instrumentation metrics (`-e gpu=level0,inst`). Requires Level Zero, IGC/IGA and GTPin.
- `-Dopencl=(disabled|auto|enabled)`: Enable OpenCL metrics (`-e gpu=opencl`). Requires OpenCL headers.
- `-Drocm=(disabled|auto|enabled)`: Enable ROCm metrics (`-e gpu=amd`). Requires ROCm.
- `-Dvalgrind_annotations=(false|true)`: Inject annotations for debugging with Valgrind.

Note that many of the features above require additional optional dependencies when enabled.

5.3 Installing Dependencies without Root

5.3.1 Meson Wraps

If core dependencies needed to build and run HPCToolkit can be found on your system, Meson will use those. To assist you in manually providing dependencies to Meson, see the discussion about how Meson *looks for dependencies* later on this page.

If Meson cannot find HPCToolkit's dependencies on your system, `meson (setup|configure)` will use Meson wraps to build HPCToolkit's core dependencies. No root privileges are needed to use Meson wraps.

If you let Meson wraps provide HPCToolkit's dependencies, then building HPCToolkit requires little more than Meson and a C/C++ compiler; any missing dependencies will be downloaded and built as subprojects. Below, we list Meson wraps that HPCToolkit will download and build if they are needed by your build configuration and they are not found on your system:

```
Subprojects
  bzip2          : YES
  dyninst        : YES 347 warnings
  elfutils       : YES 2 warnings
  libiberty      : YES (from dyninst)
  liblzma        : YES (from elfutils)
  libpfm         : YES
  libunwind      : YES
  jemalloc       : YES
  opencl-headers: YES
  tbb            : YES
  xed            : YES
  xerces-c       : YES
  xxhash         : YES
  yaml-cpp       : YES
  zstd           : YES (from elfutils)
```

Meson wraps are downloaded and built only as needed. In some cases, a wrap will not be considered unless you enable a particular feature of HPCToolkit. For instance, `opencl-headers` will not be downloaded unless your `meson (setup|configure)` command includes the configuration option `-Dopencl=enabled`. If the internet is not available on the system running `meson setup`, you may also need to run `meson subprojects download` first on a system with internet to download the required sources.

If wraps are not desired, they can be disabled with `meson (setup|configure) --wrap-mode=nofallback`. With this setting, all dependencies must be installed or otherwise loaded into the build environment, except for force-fallback dependencies `meson (setup|configure) --force-fallback-for=dep1,dep2,...`.

See the official Meson documentation about wraps for more configuration options and implementation details.

5.3.2 Dev Containers (BETA)

Another way to manage dependencies without root permission is to simply build and run HPCToolkit in container.

Multiple Dev Container configurations are provided for common distros and vendor software:

- `ubuntu20.04`, `rhel8`, `leap15`, `fedora39`: CPU-only configurations for common distros
- `cuda*`: Ubuntu 20.04 with NVIDIA CUDA toolkit installed
- `rocm*`: Ubuntu 20.04 with AMD ROCm installed
- `intel`: Ubuntu 20.04 with Intel OneAPI Base Toolkit
- `bare`: Bare minimum dependencies to build HPCToolkit

5.4 Installing Dependencies as Root

5.4.1 Debian/Ubuntu and Derivatives

For Debian Sid/Ubuntu 23.10:

```
sudo apt-get install git build-essential ccache ninja-build meson cmake pkg-config␣
↪python3-venv libboost-all-dev libbz2-dev libtbb-dev libelf-dev libdw-dev libunwind-dev␣
↪libxerces-c-dev libiberty-dev libyaml-cpp-dev libpfm4-dev libxxhash-dev zlib1g-dev
pipx install 'meson>=1.6.0'
```

For older versions:

```
sudo apt-get install git build-essential ccache ninja-build cmake pkg-config pipx␣
↪python3 python3-pip python3-venv libboost-all-dev libbz2-dev libtbb-dev libelf-dev␣
↪libdw-dev libunwind-dev libxerces-c-dev libiberty-dev libyaml-cpp-dev libpfm4-dev␣
↪libxxhash-dev zlib1g-dev
pipx install 'meson>=1.6.0'
```

Note that some dependencies may be *built from wraps* by default, either because they aren't packaged in Debian/Ubuntu (e.g. Dyninst) or because the version is too old. Additional packages can be installed for optional features:

Package	Feature option	Notes
`mpi-default-dev`	`-Dhpcprof_mpi=enabled`	
`libpapi-dev`	`-Dpapi=enabled`	
`nvidia-cuda-toolkit`	`-Dcuda=enabled`	
`level-zero-dev`	`-Dlevel0=enabled`	
`libigdfcl-dev`	`-Dlevel0=enabled`	
`libigc-dev`	`-Dgtpin=enabled`	
`opencl-headers`	`-Dopencl=enabled`	*Optional, available as wrap*
`rocm-hip-libraries`	`-Drocm=enabled`	

5.4.2 Fedora/RHEL and Derivatives

For RHEL 8:

```
sudo dnf install git gcc gcc-c++ ccache ninja-build cmake pkg-config python39 python39-pip boost-devel bzip2-devel tbb-devel elfutils-devel xerces-c-devel binutils-devel yaml-cpp-devel libpfm-devel xxhash-devel zlib-devel
python3.9 -m pip install pipx
pipx install 'meson>=1.6.0'
```

For RHEL 9:

```
sudo dnf install git gcc gcc-c++ ccache ninja-build cmake pkg-config python3 pipx boost-devel bzip2-devel tbb-devel elfutils-devel xerces-c-devel binutils-devel yaml-cpp-devel libpfm-devel libunwind-devel xxhash-devel zlib-devel
pipx install 'meson>=1.6.0'
```

For Fedora:

```
sudo dnf install git gcc gcc-c++ ccache ninja-build cmake pkg-config python3 meson pipx boost-devel bzip2-devel tbb-devel elfutils-devel xerces-c-devel binutils-devel yaml-cpp-devel libpfm-devel libunwind-devel xxhash-devel zlib-devel
pipx install 'meson>=1.6.0'  # Optional but recommended
```

Note that some dependencies may be *built from wraps* by default, either because they aren't packaged in Fedora/RHEL (e.g. Xed) or because the version is too old. Additional packages can be installed for optional features:

Package	Feature option	Notes
`openmpi-devel` or `mpich-devel` or `mvapich2-devel`	`-Dhpcprof_mpi=enabled`	
`papi-devel`	`-Dpapi=enabled`	
`cuda-toolkit`	`-Dcuda=enabled`	
`level-zero-devel`	`-Dlevel0=enabled`	
`intel-igc-opencl-devel`	`-Dlevel0=enabled`	
`opencl-headers`	`-Dopencl=enabled`	*Optional, available as wrap*
`rocm`	`-Drocm=enabled`	

5.4.3 SUSE Leap/SLES 15 and Derivatives

```
sudo zypper install git gcc12 gcc12-c++ ccache ninja-build cmake pkg-config python311 python311-pip libboost_atomic-devel libboost_chrono-devel libboost_date_time-devel libboost_filesystem-devel libboost_graph-devel libboost_system-devel libboost_thread-devel libboost_timer-devel libbz2-devel tbb-devel libxerces-c-devel binutils-devel yaml-cpp-devel libpfm-devel xxhash-devel zlib-devel
python3.11 -m pip install pipx
pipx install 'meson>=1.6.0'
```

Note that some dependencies may be *built from wraps* by default, either because they aren't packaged in SUSE Leap (e.g. Xed) or because the version is too old. Additional packages can be installed for optional features:

Package	Feature option	Notes
openmpi-devel or mpich-devel	-Dhpcprof_mpi=enabled	
papi-devel	-Dpapi=enabled	
cuda-toolkit	-Dcuda=enabled	
level-zero-devel	-Dlevel0=enabled	
intel-igc-opencl-devel	-Dlevel0=enabled	
opencl-headers	-Dopencl=enabled	*Optional, available as wrap*
rocm	-Drocm=enabled	

5.5 Custom Dependencies

It is also possible to build and run HPCToolkit using custom versions of dependencies, e.g. one you have modified to fix a bug. However, to expose such dependencies to Meson, you will need some basic understanding of how Meson finds dependencies.

This section is intended as a "peek behind the curtain" of how Meson understands dependencies, and an abridged version of [Meson's `dependency()` docs] for developers.

Meson fetches all dependencies from the "build environment" aka the "machine" (technically two, the `build_machine` and the `host_machine`, which are only different when cross-compiling). Whenever Meson configures (i.e. during `meson setup` or automatically during `meson compile/install/test/etc.`), queries the "machine" as defined by various environment variables:

- Finding programs (`find_program()`) is done by searching the `PATH`,
- The C/C++ compilers (`CC`/`CXX`), which are also affected by `CFLAGS`, `CXXFLAGS`, `LDFLAGS`, `INCLUDE_PATH`, `LIBRARY_PATH`, `CRAY_*`, . . .
- Dependencies are found via `pkg-config`, which is affected by `PKG_CONFIG_PATH` (and other `PKG_CONFIG_*` variables),
- Dependencies are found via CMake, which is affected by `CMAKE_PREFIX_PATH` (and other `CMAKE_*`, `*_ROOT` and `*_ROOT_DIR` variables),
- Dependencies with Meson built-in functionality have their own quirks:
  - Boost is found under `BOOST_ROOT`,
  - The CUDA Toolkit is found under `CUDA_PATH`.

These variables are frequently altered by commands designed to load software into a shell environment, for example `module un/load`. Meson assumes the "machine" does not change between `meson` invocations, however if it does by running the above commands it is possible Meson's cache will not match reality. In these cases reconfiguration or later compilation may fail due to the inconsistent state between dependencies. In some cases this can be fixed by clearing Meson's cache (`meson configure --clearcache`), Meson will efficiently avoid rebuilding files if the compile command did not actually change. If build errors persist it may be necessary to wipe the build directory completely (`meson setup --wipe`), it is recommended to use `ccache` to accelerate the subsequent rebuild.

If you want to use a custom version of a dependency library, install it first and then adjust one or more of the variables mentioned above or corresponding Meson options:

- If the dependency installs a `pkg-config` file (usually `lib/pkgconfig/*.pc`), append the directory containing these files to `PKG_CONFIG_PATH` or `-Dpkg_config_path`.
- In all other cases, append the install prefix (i.e. the directory with `include/` and `lib/` or `lib64/`) to `CMAKE_PREFIX_PATH` or `-Dcmake_prefix_path`.

- For select dependencies (i.e. if the dependency doesn't install `pkg-config` or CMake files and there is no matching `share/cmake-*/Modules/Find*.cmake` script in the CMake installation), you may also opt to configure the compiler instead, by:
  - Extending `-Dc_args` with appropriate `-I` include flags and `-Dc_link_args` with `-L` link flags, or
  - Extending `CPATH` with `include/` directories and `LIBRARY_PATH` with `lib/` or `lib64/` directories.

Some of these variables can be persisted as Meson built-in options (e.g. `-Dpkg_config_path`) or more generally [native files][meson native file]. For example, a native file to use GCC 11 with extra flags and a custom dependency search flags would look like:

```
# .../my-env.ini
[binaries]
c = ['ccache', 'gcc-11']
cpp = ['ccache', 'g++-11']

[built-in options]
c_args = ['-fstack-protector']
cpp_args = ['-fhardened']
cmake_prefix_path = ['/.../path/to/custom/install/', '/.../path/to/other/install/']
pkg_config_path = ['/.../path/to/custom/install/lib/pkgconfig', '/.../path/to/other/
↪install/lib/pkgconfig']
```

And then can be used in a build directory by passing additional arguments to `meson setup`:

```
$ meson setup --native-file .../my-elf-dyn.ini builddir/
```

See [`doc/developers/meson.ini`] for a more complete template listing the settings available in a native file.

5.6 Links to Meson Documentation

- Meson
- Meson native file
- Meson subprojects
- Meson wraps
- Meson's `dependency()` docs

CHAPTER

SIX

MONITORING APPLICATIONS

This chapter describes the mechanics of using hpcrun to profile a single process application and collect performance data. For advice on how to choose events, perform scaling studies, etc., see Chapter *12 Strategies for Insight*.

6.1 Using hpcrun

The hpcrun launcher is used to run an application and collect call path profiles and call path traces data for *dynamically linked* binaries. For dynamically linked programs, this requires no change to the program source and no change to the build procedure. You should build your application natively with full optimization. hpcrun inserts its profiling code into the application at runtime via LD_AUDIT (preferred in most situations) or LD_PRELOAD (still available since most versions of glibc LD_AUDIT have significant bugs in corner cases).

hpcrun monitors the execution of applications on a CPU using asynchronous sampling and, upon request, monitors GPU activity using vendor runtime capabilities. If hpcrun is used without any arguments to measure a program

```
hpcrun app arg ...
```

it will the measure the program's execution by sampling its CPUTIME and collect a call path profile for each thread in the execution. More about the CPUTIME metric can be found in Section *7.1*.

In addition to a call path profile, hpcrun can collect a call path trace of an execution if the -t (or --trace) option is used turn on tracing. The following use of hpcrun will collect both a call path profile and a call path trace of CPU execution using the default CPUTIME sample source.

```
hpcrun -t app arg ...
```

The -t will collect a call path trace based on asynchronous sampling of each CPU thread using a time-based sampling metric. Besides the default CPUTIME metric, other time-based metrics include REALTIME and CYCLES. How to use these metrics is described a bit later.

The -t option also traces any GPU operations if a -e gpu=* option is used to enable measurement of GPU activities.

Traces are most useful for understanding the execution dynamics of multithreaded or multi-process applications; however, you may find a trace of a single-threaded application to be useful to understand how an execution unfolds over time.

For programs that launch many GPU operations per second, the -tt will collect a more useful trace than -t. Like the -t option, -tt records a call path trace based on asynchronous sampling of each CPU thread using a time-based sampling metric. Unlike -t, -tt also records a sample for a thread as it launches each GPU operation, if any. Without -tt, the activity seen on a CPU trace line at the time it launches a GPU operation is often from far earlier in the execution, which can make it difficult to relate to concurrent CPU and GPU activity. Since additional non-sample elements are added, any statistical properties of the CPU traces are disturbed.

While CPUTIME is used as the default sample source if no other sample source is specified, many other sample sources are available. Typically, one uses the `-e` (or `--event`) to specify a sample source and sampling rate.[7] Sample sources are specified as '`event@howoften`' where `event` is the name of the source and `howoften` is either a number specifying the period (threshold) for that event, or `f` followed by a number, e.g., `@f100` specifying a target sampling frequency for the event in samples/second.[8] Note that a higher period implies a lower rate of sampling. The `-e` option may be used multiple times to specify that multiple sample sources be used for measuring an execution.

The basic syntax for profiling an application with `hpcrun` is:

```
hpcrun -t -e event@howoften ... app arg ...
```

For example, to profile an application using hardware counter sample sources provided by Linux `perf_events` and sample cycles 300 times/second (the default sampling frequency) and sample every 4,000,000 instructions, you would use:

```
hpcrun -e CYCLES -e INSTRUCTIONS@4000000 app arg ...
```

The units for timer-based sample sources (`CPUTIME` and `REALTIME` are microseconds, so to sample an application with tracing every 5,000 microseconds (200 times/second), you would use:

```
hpcrun -t -e CPUTIME@5000 app arg ...
```

`hpcrun` stores its raw performance data in a *measurements* directory with the program name in the directory name. On systems with a batch job scheduler (eg, PBS) the name of the job is appended to the directory name.

```
hpctoolkit-app-measurements[-jobid]
```

It is best to use a different measurements directory for each run. So, if you're using `hpcrun` on a local workstation without a job launcher, you can use the '`-o dirname`' option to specify an alternate directory name.

For programs that use their own launch script (eg, `mpirun` or `mpiexec` for MPI), put the application's run script on the outside (first) and `hpcrun` on the inside (second) on the command line. For example,

```
mpirun -n 4 hpcrun -e CYCLES mpiapp arg ...
```

Note that `hpcrun` is intended for profiling dynamically linked *binaries*. It will not work well if used to profile a shell script. At best, you would be profiling the shell interpreter, not the script commands, and sometimes this will fail outright. Profiling statically-linked binaries is no longer supported by HPCToolkit.

6.1.1 If `hpcrun` causes your application to fail

`hpcrun` can cause applications to fail in certain circumstances. Here, we describe two kind of failures that may arise and how to sidestep them.

`hpcrun` causes failures related to loading or using shared libraries

Unfortunately, the Glibc implementations used today on most platforms have known bugs monitoring loading and unloading of shared libraries and calls to a shared library's API. While the best approach for coping with these problems is to use a system running Glibc 2.35 or later, for most people, this is not an option: the system administrator picks the operating system version, which determines the Glibc version available to developers.

To understand what kinds of problems that you may encounter with shared libraries and how you can work around them, it is helpful to understand how HPCToolkit monitors shared libraries. On Power and `x86_64` architectures, by default `hpcrun` uses `LD_AUDIT` to monitor an application's use of dynamic libraries. Use of `LD_AUDIT` is the only

[7] GPU and OpenMP measurement events don't accept a rate.

[8] Frequency-based sampling and the frequency-based notation for `howoften` is only available for sample sources managed by Linux `perf_events`. For Linux `perf_events`, HPCToolkit uses a default sampling frequency of 300 samples/second.

strategy for monitoring shared libraries that will not cause a change in application behavior when libraries contain a `RUNPATH`. However, Glibc's implementation of `LD_AUDIT` has a number of bugs that may crash the application:

- Until Glibc 2.35, most applications running on ARM will crash. This was caused by a fatal flaw in Glibc's PLT handler for ARM, where an argument register that should have been saved was instead replaced with a junk pointer value. This register is used to return C/C++ `struct` values from functions and methods, including some C++ constructors.
- Until Glibc 2.35, applications and libraries using `dlmopen` will crash. While most applications do not use `dlmopen`, an example of a library that does is Intel's GTPin, which `hpcrun` uses to instrument Intel GPU code.
- Applications and libraries using significant amounts of static TLS space may crash with the message "`cannot allocate memory in static TLS block`." This is caused by a flaw in Glibc causing it to allocate insufficient static TLS space when `LD_AUDIT` is enabled. For Glibc 2.35 and newer, setting the environment variable

  ```
  export GLIBC_TUNABLES=glibc.rtld.optional_static_tls=0x1000000
  ```

  will instruct Glibc to allocate 16MB of static TLS memory per thread, in our experience this is far more than any application will use (however the value can be adjusted freely). For older Glibc, the only option is to disable `hpcrun`'s use of `LD_AUDIT`.

The following options direct `hpcrun` to adjust the strategy it uses for monitoring dynamic libraries. We suggest that you don't consider using any of these options unless your program fails using `hpcrun`'s defaults.

`--disable-auditor`
: This option instructs hpcrun to track dynamic library operations by intercepting `dlopen` and `dlclose` instead of using `LD_AUDIT`. Note that this alternate approach can cause problem with libraries and applications that specify a `RUNPATH`.

`--enable-auditor`
: This option is default, except on ARM or when Intel GTPin instrumentation is enabled. Passing this option instructs `hpcrun` to use `LD_AUDIT` in all cases.

`--disable-auditor-got-rewriting`
: When using an `LD_AUDIT`, Glibc unnecessarily intercepts every call to a function in a shared library. `hpcrun` avoids this overhead by rewriting each shared library's global offset table (GOT). Such rewriting is tricky. This option can be used to disable GOT rewriting if it is believed that the rewriting is causing the application to fail.

`--namespace-single`
: `dlmopen` may load a shared library into an alternate namespace, which crashes on Glibc until 2.35. This option instructs `hpcrun` to override `dlmopen` to instead load all shared libraries within the application namespace. This may significantly change application behavior, but may be helpful to avoid crashing. This option is default when Intel GTPin instrumentation is enabled.

`--namespace-multiple`
: This option is the opposite of `--namespace-single`, and will instruct `hpcrun` to *not* override `dlmopen` and thus retain its normal function. This option is default except when Intel GTPin instrumentation is enabled.

If your code fails to find libraries when it is monitoring your code by wrapping `dlopen` and `dlclose` rather than using `LD_AUDIT`, you can sidestep this problem by adding any library paths listed in the `RUNPATH` of your application or library to your `LD_LIBRARY_PATH` environment variable before launching `hpcrun`.

`hpcrun` causes your application to fail when `gprof` instrumentation is present

When an application has been compiled with the compiler flag `-pg`, the compiler adds instrumentation to collect performance measurement data for the `gprof` profiler. Measuring application performance with HPCToolkit's measurement subsystem and `gprof` instrumentation active in the same execution may cause the execution to abort. One can detect the presence of `gprof` instrumentation in an application by the presence of `__monstartup` and `_mcleanup` symbols in a executable. One can disable `gprof` instrumentation when measuring the performance of a dynamically-linked application by using the `--disable-gprof` argument to `hpcrun`.

6.2 Experimental Python Support

This section provides a brief overview of how to use HPCToolkit to analyze the performance of Python-based applications. Normally, `hpcrun` will attribute performance to the CPython implementation, not to the application Python code, as shown in Figure *6.2*. This usually is of little interest to an application developer, so HPCToolkit provides experimental support for attributing to Python callstacks.

NOTE: Python support in HPCToolkit is in early days. If you compile HPCToolkit to match the version of Python being used by your application, in many cases you will find that you can measure what you want. However, other cases may not work as expected; crashes and corrupted performance data are not uncommon. For the aforementioned reasons, use at your own risk.

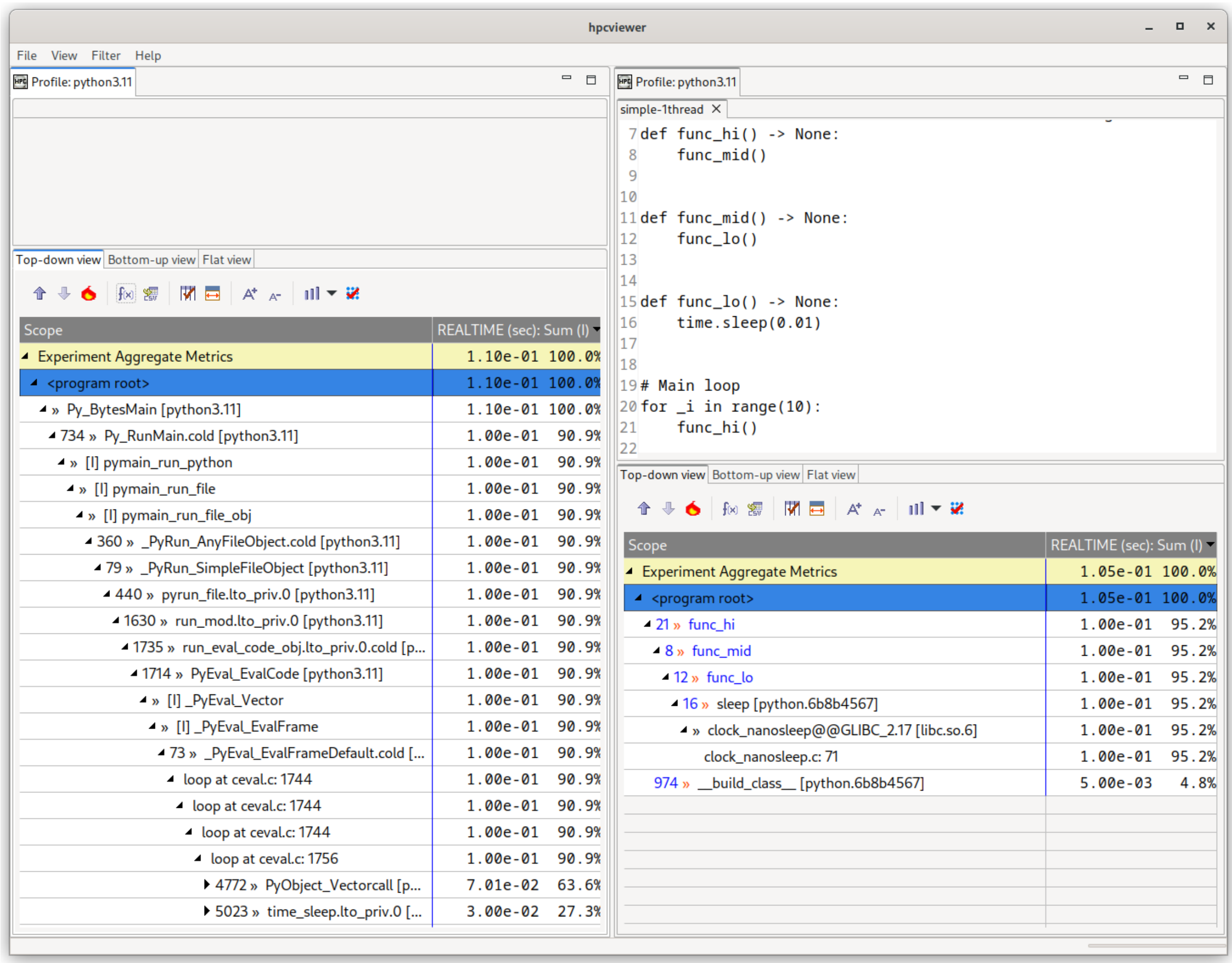


Fig. 1: Figure 6.2: Example of a simple Python application measured without (left) and with (right) Python support enabled via `hpcrun -a python`. The left database has no source code, since sources were not provided for the CPython implementation.

If HPCToolkit has been compiled with Python support enabled, `hpcrun` is able to replace segments of the C callstacks with the Python code running in those frames. To enable this transformation, profile your application the additional `-a python` flag:

```
hpcrun -a python -e event@howoften python3 app arg ...
```

As shown in Figure *6.2*, passing this flag removes the CPython implementation details, replacing it with the much smaller Python callstack. When Python calls an external C library, HPCToolkit will report both the name of the Python function object and the C function being called, in this example `sleep` and Glibc's `clock_nanosleep` respectively.

6.2.1 Known Limitations

This section lists a number of known limitations with the current implementation of the Python support. It is recommended that users are aware of these limitations before attempting to use the Python support in practice.

1. Pythons older than 3.8 are not supported by HPCToolkit. Please upgrade any applications and Python extensions to use a recent version of Python before attempting to enable Python support.
2. The application should be run with the same Python that was used to compile HPCToolkit. The CPython ABI can change between patch versions and due to certain build configuration flags. To ensure `hpcrun` will not unwittingly crash the application, it is best to use a single Python for both HPCToolkit and the application.

   Despite this recommendation, it is worth noting that we have had some minor success with cross-version compatibility (e.g. building HPCToolkit with Python 3.11.8 support and using it to measure a program running under Python 3.11.6) However, you should be surprised if cross-version compatibility works rather than expecting it to work, even across patch releases.
3. The bottom-up and flat views of `hpcviewer` may not correctly present Python callstacks, particularly those that call C/C++ extensions. Some Python functions may be missing, and the metrics attributed to them may be suspect. In these cases, refer to the top-down view as the known-good source of truth.
4. Threads spawned by Python's `threading` and `subprocess` modules are not fully supported. Only the main Python thread will attribute performance to Python callstacks, all others will attribute performance to the CPython implementation. If Python `threading` is a performance bottleneck, consider implementing the parallelism in a C/C++ extension instead of in Python to avoid contention on the GIL.
5. Applications using signals and signal handlers, for example Python's `signal` module, will experience crashes when run under `hpcrun`. The current implementation fails to process the non-sequential modifications to the Python stack that take place when Python handles signals.

6.3 Process Fraction

Although `hpcrun` can profile parallel jobs with thousands or tens of thousands of processes, there are two scaling problems that become prohibitive beyond a few thousand cores. First, `hpcrun` writes the measurement data for all of the processes into a single directory. This results in one file per process plus one file per thread (two files per thread if using tracing). Unix file systems are not equipped to handle directories with many tens or hundreds of thousands of files. Second, the sheer volume of data can overwhelm the viewer when the size of the database far exceeds the amount of memory on the machine.

The solution is to sample only a fraction of the processes. That is, you can run an application on many thousands of cores but record data for only a few hundred processes. The other processes run the application but do not record any measurement data. This is what the process fraction option (`-f` or `--process-fraction`) does. For example, to monitor 10% of the processes, use:

```
hpcrun -f 0.10 -e event@howoften app arg ...
hpcrun -f 1/10 -e event@howoften app arg ...
```

With this option, each process generates a random number and records its measurement data with the given probability. The process fraction (probability) may be written as a decimal number (0.10) or as a fraction (1/10) between 0 and 1. So, in the above example, all three cases would record data for approximately 10% of the processes. Aim for a number of processes in the hundreds.

6.4 Start and Stop API

HPCToolkit supports an API for the application to start and stop sampling. This is useful if you want to profile only a subset of a program and ignore the rest. The API supports the following functions.

```
void hpctoolkit_sampling_start(void);
void hpctoolkit_sampling_stop(void);
```

For example, suppose that your program has three major phases: it reads input from a file, performs some numerical computation on the data and then writes the output to another file. And suppose that you want to profile only the compute phase and skip the read and write phases. In that case, you could stop sampling at the beginning of the program, restart it before the compute phase and stop it again at the end of the compute phase.

This interface is process wide, not thread specific. That is, it affects all threads of a process. Note that when you turn sampling on or off, you should do so uniformly across all processes, normally at the same point in the program. Enabling sampling in only a subset of the processes would likely produce skewed and misleading results.

And for technical reasons, when sampling is turned off in a threaded process, interrupts are disabled only for the current thread. Other threads continue to receive interrupts, but they don't unwind the call stack or record samples. So, another use for this interface is to protect syscalls that are sensitive to being interrupted with signals. For example, some Gemini interconnect (GNI) functions called from inside `gasnet_init()` or `MPI_Init()` on Cray XE systems will fail if they are interrupted by a signal. As a workaround, you could turn sampling off around those functions.

Also, you should use this interface only at the top level for major phases of your program. That is, the granularity of turning sampling on and off should be much larger than the time between samples. Turning sampling on and off down inside an inner loop will likely produce skewed and misleading results.

To use this interface, put the above function calls into your program where you want sampling to start and stop. Remember, starting and stopping apply process wide. For C/C++, include the following header file from the HPCToolkit `include` directory.

```
#include <hpctoolkit.h>
```

Compile your application with `libhpctoolkit` with `-I` and `-L` options for the include and library paths. For example,

```
gcc -I /path/to/hpctoolkit/include app.c ... \
    -L /path/to/hpctoolkit/lib/hpctoolkit -lhpctoolkit ...
```

The `libhpctoolkit` library provides weak symbol no-op definitions for the start and stop functions. For dynamically linked programs, be sure to include `-lhpctoolkit` on the link line (otherwise your program won't link).

To run the program, set the `LD_LIBRARY_PATH` environment variable to include the HPCToolkit `lib/hpctoolkit` directory.

```
export LD_LIBRARY_PATH=/path/to/hpctoolkit/lib/hpctoolkit
```

Note that sampling is initially turned on until the program turns it off. If you want it initially turned off, then use the `-ds` (or `--delay-sampling`) option for `hpcrun`.

```
hpcrun -ds -e event@howoften app arg ...
```

6.5 Environment Variables

For most systems, `hpcrun` requires no special environment variable settings. There are two situations, however, where `hpcrun`, to function correctly, *must* refer to environment variables. These environment variables, and corresponding situations are:

`HPCTOOLKIT`
: To function correctly, `hpcrun` must know the location of the HPCToolkit top-level installation directory. The `hpcrun` script uses elements of the installation `lib` and `libexec` subdirectories. On most systems, the `hpcrun` can find the requisite components relative to its own location in the file system. However, some parallel job launchers *copy* the `hpcrun` script to a different location as they launch a job. If your system does this, you must set the `HPCTOOLKIT` environment variable to the location of the HPCToolkit top-level installation directory before launching a job.

Note to system administrators: if your system provides a module system for configuring software packages, then constructing a module for HPCToolkit to initialize these environment variables to appropriate settings would be convenient for users.

CHAPTER SEVEN

MONITORING CPU

`hpcrun` monitors CPU activity by periodically interrupting execution and recording the program's call path at each interrupt. The interrupt source — called a sample source — determines what triggers each sample and what is measured. HPCToolkit supports the following sample sources:

- Linux timers (`REALTIME`, `CPUTIME`) — sample at regular wall-clock or CPU-time intervals, independent of hardware counter support.
- I/O tracing (`IO`) — record I/O operations to identify time spent in read/write calls.
- Memory leak detection (`MEMLEAK`) — track heap allocations and deallocations to identify memory that is allocated but never freed.
- Hardware performance counters — measure microarchitectural events such as cache misses, branch mispredictions, and pipeline stalls, via either **PAPI** or **Linux perf_events**.

The remainder of this section describes each sample source in detail.

7.1 REALTIME and CPUTIME

HPCToolkit supports two timer-based sample sources: `CPUTIME` and `REALTIME`. The unit for periods of these timers is microseconds.

Before describing this capability further, it is worth noting that the CYCLES event supported by Linux perf_events or PAPI's `PAPI_TOT_CYC` are generally superior to any of the timer-based sampling sources.

The `CPUTIME` and `REALTIME` sample sources are based on the POSIX timers `CLOCK_THREAD_CPUTIME_ID` and `CLOCK_REALTIME` with the Linux `SIGEV_THREAD_ID` extension. `CPUTIME` only counts time when the CPU is running; `REALTIME` counts real (wall clock) time, whether the process is running or not. Signal delivery for these timers is thread-specific, so these timers are suitable for profiling multithreaded programs. Sampling using the `REALTIME` sample source may break some applications that don't handle interrupted syscalls well. In that case, consider using `CPUTIME` instead.

The following example, which specifies a period of 5000 microseconds will sample each thread in `app` at a rate of approximately 200 times per second.

```
hpcrun -e REALTIME@5000 app arg ...
```

do not use more than one timer-based sample source to monitor a program execution. When using a sample source such as `CPUTIME` or `REALTIME`, we recommend not using another time-based sampling source such as Linux perf_events `CYCLES` or PAPI's `PAPI_TOT_CYC`. Technically, this is feasible and `hpcrun` won't die. However, multiple time-based sample sources would compete with one another to measure the execution and likely lead to dropped samples and possibly distorted results.

7.2 IO

The IO sample source counts the number of bytes read and written. This displays two metrics in the viewer: “IO Bytes Read” and “IO Bytes Written.” The IO source is a synchronous sample source. It overrides the functions `read`, `write`, `fread` and `fwrite` and records the number of bytes read or written along with their dynamic context synchronously rather than relying on data collection triggered by interrupts.

To include this source, use the IO event (no period). For example,

```
hpcrun -e IO app arg ...
```

The IO source is mainly used to find where your program reads or writes large amounts of data. However, it is also useful for tracing a program that spends much time in `read` and `write`. The hardware performance counters do not advance while running in the kernel, so the trace viewer may misrepresent the amount of time spent in syscalls such as `read` and `write`. By adding the IO source, `hpcrun` overrides `read` and `write` and thus is able to more accurately count the time spent in these functions.

7.3 MEMLEAK

The MEMLEAK sample source counts the number of bytes allocated and freed. Like IO, MEMLEAK is a synchronous sample source and does not generate asynchronous interrupts. Instead, it overrides the malloc family of functions (`malloc`, `calloc`, `realloc` and `free` plus `memalign`, `posix_memalign` and `valloc`) and records the number of bytes allocated and freed along with their dynamic context.

MEMLEAK allows you to find locations in your program that allocate memory that is never freed. But note that failure to free a memory location does not necessarily imply that location has leaked (missing a pointer to the memory). It is common for programs to allocate memory that is used throughout the lifetime of the process and not explicitly free it.

To include this source, use the MEMLEAK event (no period). For example,

```
hpcrun -e MEMLEAK app arg ...
```

If a program allocates and frees many small regions, the MEMLEAK source may result in a high overhead. In this case, you may reduce the overhead by using the memleak probability option to record only a fraction of the mallocs. For example, to monitor 10% of the mallocs, use:

```
hpcrun -e MEMLEAK --memleak-prob 0.10 app arg ...
```

It might appear that if you monitor only 10% of the program’s mallocs, then you would have only a 10% chance of finding the leak. But if a program leaks memory, then it’s likely that it does so many times, all from the same source location. And you only have to find that location once. So, this option can be a useful tool if the overhead of recording all mallocs is prohibitive.

7.4 PAPI

PAPI, the Performance API, is a library for providing access to the hardware performance counters. PAPI aims to provide a consistent, high-level interface that consists of a universal set of event names that can be used to measure performance on any processor, independent of any processor-specific event names. In some cases, PAPI event names represent quantities synthesized by combining measurements based on multiple native events available on a particular processor. For instance, in some cases PAPI reports total cache misses by measuring and combining data misses and instruction misses. PAPI is available from the University of Tennessee at https://icl.cs.utk.edu/papi.

PAPI focuses mostly on in-core CPU events: cycles, cache misses, floating point operations, mispredicted branches, etc. For example, the following command samples total cycles and L2 cache misses.

```
hpcrun -e PAPI_TOT_CYC@15000000 -e PAPI_L2_TCM@400000 app arg ...
```

The precise set of PAPI preset and native events is highly system dependent. Commonly, there are events for machine cycles, cache misses, floating point operations and other more system specific events. However, there are restrictions both on how many events can be sampled at one time and on what events may be sampled together and both restrictions are system dependent. Table *below* contains a list of commonly available PAPI events.

To see what PAPI events are available on your system, use the `papi_avail` command from the `bin` directory in your PAPI installation. The event must be both available and not derived to be usable for sampling. The command `papi_native_avail` displays the machine's native events. Note that on systems with separate compute nodes, you normally need to run `papi_avail` on one of the compute nodes.

Table 1: Some commonly available PAPI events. The exact set of available events is system dependent.

Name	Description
PAPI_BR_INS	Branch instructions
PAPI_BR_MSP	Conditional branch instructions mispredicted
PAPI_FP_INS	Floating point instructions
PAPI_FP_OPS	Floating point operations
PAPI_L1_DCA	Level 1 data cache accesses
PAPI_L1_DCM	Level 1 data cache misses
PAPI_L1_ICH	Level 1 instruction cache hits
PAPI_L1_ICM	Level 1 instruction cache misses
PAPI_L2_DCA	Level 2 data cache accesses
PAPI_L2_ICM	Level 2 instruction cache misses
PAPI_L2_TCM	Level 2 cache misses
PAPI_LD_INS	Load instructions
PAPI_SR_INS	Store instructions
PAPI_TLB_DM	Data translation lookaside buffer misses
PAPI_TOT_CYC	Total cycles
PAPI_TOT_IIS	Instructions issued
PAPI_TOT_INS	Instructions completed

When selecting the period for PAPI events, aim for a rate of approximately a few hundred samples per second. So, roughly several million or tens of million for total cycles or a few hundred thousand for cache misses. PAPI and `hpcrun` will tolerate sampling rates as high as 1,000 or even 10,000 samples per second (or more). However, rates higher than a few hundred samples per second will only increase measurement overhead and distort the execution of your program; they won't yield more accurate results.

Beginning with Linux kernel version 2.6.32, support for accessing performance counters using the Linux perf_events performance monitoring subsystem is built into the kernel. perf_events provides a measurement substrate for PAPI on Linux.

On modern Linux systems that include support for perf_events, PAPI is only recommended for monitoring events outside the scope of the perf_events interface.

7.4.1 Proxy Sampling

HPCToolkit supports proxy sampling for derived PAPI events. For HPCToolkit to sample a PAPI event directly, the event must not be derived and must trigger hardware interrupts when a threshold is exceeded. For events that cannot trigger interrupts directly, HPCToolkit's proxy sampling sample on another event that is supported directly and then reads the counter for the derived event. In this case, a native event can serve as a proxy for one or more derived events.

To use proxy sampling, specify the `hpcrun` command line as usual and be sure to include at least one non-derived PAPI event. The derived events will be accumulated automatically when processing a sample trigger for a native event. We recommend adding `PAPI_TOT_CYC` as a native event when using proxy sampling, but proxy sampling will gather data as long as the event set contains at least one non-derived PAPI event. Proxy sampling requires one non-derived PAPI event to serve as the proxy; a Linux timer can't serve as the proxy for a PAPI derived event.

For example, on newer Intel CPUs, often PAPI floating point events are all derived and cannot be sampled directly. In that case, you could count FLOPs by using cycles a proxy event with a command line such as the following. The period for derived events is ignored and may be omitted.

```
hpcrun -e PAPI_TOT_CYC@6000000 -e PAPI_FP_OPS app arg ...
```

Attribution of proxy samples is not as accurate as regular samples. The problem, of course, is that the event that triggered the sample may not be related to the derived counter. The total count of events should be accurate, but their location at the leaves in the Calling Context tree may not be very accurate. However, the higher up the CCT, the more accurate the attribution becomes. For example, suppose you profile a loop of mixed integer and floating point operations and sample on `PAPI_TOT_CYC` directly and count `PAPI_FP_OPS` via proxy sampling. The attribution of flops to individual statements within the loop is likely to be off. But as long as the loop is long enough, the count for the loop as a whole (and up the tree) should be accurate.

7.5 Linux `perf_events`

Linux perf_events provides a powerful interface that supports measurement of both application execution and kernel activity. Using perf_events, one can measure both hardware and software events. Using a processor's hardware performance monitoring unit (PMU), the perf_events interface can measure an execution using any hardware counter supported by the PMU. Examples of hardware events include cycles, instructions completed, cache misses, and stall cycles. Using instrumentation built in to the Linux kernel, the perf_events interface can measure software events. Examples of software events include page faults, context switches, and CPU migrations.

7.5.1 Frequency-based sampling

The Linux perf_events interface supports frequency-based sampling. With frequency-based sampling, the kernel automatically selects and adjusts an event period with the aim of delivering samples for that event at a target sampling frequency.[9] Unless a user explicitly specifies an event count threshold for an event, HPCToolkit's measurement interface will use frequency-based sampling by default. HPCToolkit's default sampling frequency is `min(300, M-1)`, where `M` is the value specified in the system configuration file `/proc/sys/kernel/perf_event_max_sample_rate`.

For circumstances where the user wants to use frequency-based sampling but HPCToolkit's default sampling frequency is inappropriate, one can specify the target sampling frequency for a particular event using the notation *event*`@f`*rate* when specifying an event or change the default sampling frequency. When measuring a dynamically-linked executable using `hpcrun`, one can change the default sampling frequency using `hpcrun`'s `-c` option.

[9] The kernel may be unable to deliver the desired frequency if there are fewer events per second than the desired frequency.

Example

The following command specifies a target sampling frequency of `f100` which means 100 samples per second for the `CYCLES` event.

```
hpcrun -e CYCLES@f100 my_program
```

7.5.2 Multiplexing

Multiplexing enables one to monitor more events in a single execution than the number of hardware counters a processor can support for each thread. The number of events that can be monitored in a single execution is only limited by the maximum number of concurrent events that the kernel will allow a user to multiplex using the perf_events interface.

When more events are specified than can be monitored simultaneously using a thread's hardware counters,[10] the kernel will employ multiplexing and divide the set of events to be monitored into groups, monitor only one group of events at a time, and cycle repeatedly through the groups as a program executes.

For applications that have very regular, steady state behavior, e.g., an iterative code with lots of iterations, multiplexing will yield results that are suitably representative of execution behavior. However, for executions that consist of unique short phases, measurements collected using multiplexing may not accurately represent the execution behavior. To obtain more accurate measurements, one can run an application multiple times and in each run collect a subset of events that can be measured without multiplexing. Results from several such executions can be imported into HPCToolkit's `hpcviewer` and analyzed together.

7.5.3 Profiling with `perf_events`

When profiling with native events, by default hpcrun will access them directly using the perf_events API. To force HPCToolkit to use PAPI to manage monitoring of a native event rather than using perf_events directly (assuming that HPCToolkit has been configured to include support for PAPI), one must prefix the native event name with '`papi::`' as follows:

```
hpcrun -e papi::CYCLES
```

For PAPI presets, there is no need to prefix the event with '`papi::`'. For instance it is sufficient to specify `PAPI_TOT_CYC` event without any prefix to profile using PAPI. For more information about using PAPI, see Section *6.3.2*.

Below, we provide some examples of various ways to measure `CYCLES` and `INSTRUCTIONS` using HPCToolkit's perf_events measurement substrate:

To sample an execution 100 times per second (frequency-based sampling) counting `CYCLES` and 100 times a second counting `INSTRUCTIONS`:

```
hpcrun -e CYCLES@f100 -e INSTRUCTIONS@f100 ...
```

To sample an execution every 1,000,000 cycles and every 1,000,000 instructions using period-based sampling:

```
hpcrun -e CYCLES@1000000 -e INSTRUCTIONS@1000000
```

By default, hpcrun uses frequency-based sampling with the rate 300 samples per second per event type. Hence the following command causes HPCToolkit to sample `CYCLES` at 300 samples per second and `INSTRUCTIONS` at 300 samples per second:

[10] How many events can be monitored simultaneously on a particular processor may depend on the events specified.

```
hpcrun -e CYCLES -e INSTRUCTIONS ...
```

One can specify a different default sampling period or frequency using the `-c` option. The command below will sample `CYCLES` and `INSTRUCTIONS` at 200 samples per second each:

```
hpcrun -c f200 -e CYCLES -e INSTRUCTIONS ...
```

7.5.4 Notes

- Linux perf_events uses one file descriptor for each event to be monitored. Furthermore, since `hpcrun` generates one hpcrun file for each thread, and an additional hpctrace file if traces is enabled. Hence for `e` events and `t` threads, the required number of file descriptors is:

  ```
  t * e + t (+t if trace is enabled)
  ```

  For instance, if one profiles a multi-threaded program that executes with 500 threads using 4 events, then the required number of file descriptors is

  ```
  500 threads * 4 events + 500 hpcrun files + 500 hpctrace files = 3000 file␣
  ↪descriptors
  ```

  If the number of file descriptors exceeds the number of maximum number of open files, then the program will crash. To remedy this issue, one needs to increase the number of maximum number of open files allowed.

- When a system is configured with suitable permissions, HPCToolkit will sample call stacks within the Linux kernel in addition to application-level call stacks. This feature can be useful to measure kernel activity on behalf of a thread (e.g., zero-filling allocated pages when they are first touched) or to observe where, why, and how long a thread blocks. For a user to be able to sample kernel call stacks, the configuration file `/proc/sys/kernel/perf_event_paranoid` must have a value <=1. To associate addresses in kernel call paths with function names, the value of `/proc/sys/kernel/kptr_restrict` must be 0 (number zero). If these settings are not configured in this way on your system, you will need someone with administrator privileges to change them for you to be able to sample call stacks within the kernel.

- Due to a limitation present in all Linux kernel versions currently available, HPCToolkit's measurement subsystem can only approximate a thread's blocking time. At present, Linux reports when a thread blocks but does not report when a thread resumes execution. For that reason, HPCToolkit's measurement subsystem approximates the time a thread spends blocked using sampling as the time between when the thread blocks and when the thread receives its first sample after resuming execution.

- Users need to be cautious when considering measured counts of events that have been collected using hardware counter multiplexing. Currently, it is not obvious to a user if a metric was measured using a multiplexed counter. This information is present in the measurements but is not currently visible in `hpcviewer`.

7.5.5 Thread blocking

When a program executes, a thread may block waiting for the kernel to complete some operation on its behalf. For instance, a thread may block waiting for data to become available so that a `read` operation can complete. On systems running Linux 4.3 or newer, one can use the `perf_events` sample source to monitor how much time a thread is blocked and where the blocking occurs. To measure the time a thread spends blocked, one can profile with `BLOCKTIME` event and another time-based event, such as `CYCLES`. The `BLOCKTIME` event shouldn't have any frequency or period specified, whereas `CYCLES` may have a frequency or period specified.

Example

The following command profiles `my_program` using `BLOCKTIME` to how long a thread is blocked and `CYCLES` events the total cycles spent.

```
hpcrun -e BLOCKTIME -e CYCLES my_program
```

7.6 CPU Top-down Profiling

HPCToolkit supports Intel's Top-Down Microarchitecture Analysis (TMA) methodology through a set of special measurement events recognized by `hpcrun`. TMA provides a hierarchical framework for diagnosing CPU pipeline bottlenecks by categorizing pipeline slots into a two-root hierarchy: **Active** (slots where the CPU is doing useful or attempted work) and **Stall** (slots where the CPU is unable to issue micro-operations due to resource limitations or delays). Each root is further subdivided into increasingly specific sub-metrics at Levels 2 through 4.

HPCToolkit's TMA support is currently available on **Intel Sapphire Rapids** and newer microarchitectures that provide the `PERF_METRICS` MSR (Model Specific Register) and the `TOPDOWN.SLOTS` fixed counter (Fixed Counter 3). Support requires `perf_event_paranoid` set to 1 or lower, or the `CAP_PERFMON` capability.

7.6.1 TMA Metric Tree

The TMA metric tree is rooted at **Total Slots**, which represents the total number of pipeline slot opportunities available to the CPU over a measurement interval — computed as the dispatch width of the processor multiplied by the number of elapsed cycles. Every pipeline slot is accounted for by exactly one of two mutually exclusive categories:

- **Active**, where the CPU was doing useful or attempted work
- **Stall**, where the CPU was blocked and unable to issue micro-operations

Each of these two roots is further subdivided into a hierarchy of increasingly specific bottleneck categories. **Active** comprises **Retiring** (slots consumed by instructions that successfully committed) and **Bad Speculation** (slots wasted on instructions that were later squashed). **Stall** comprises **Frontend Bound** (slots lost because the frontend could not deliver instructions) and **Backend Bound** (slots lost because the backend could not accept new work). These four Level 1 categories form the foundation of TMA diagnosis: at any given measurement scope, the dominant category identifies which subtree to investigate next, drilling down through Levels 2, 3, and 4 until the root cause of the bottleneck is isolated.

```
Total Slots
├── Active
│   ├── Retiring
│   │   ├── Heavy Operations
│   │   │   ├── Uops Sequencer
│   │   │   └── Few Operations
│   │   └── Light Operations
│   │       ├── FP Arithmetic
│   │       │   ├── X87 Use
│   │       │   ├── FP Scalar
│   │       │   └── FP Vector
│   │       ├── Integer Operations
│   │       └── Memory Operations
│   └── Bad Speculation
│       ├── Misprediction
```

(continues on next page)

(continued from previous page)

```
│       └── Machine Clear
└── Stall
    ├── Frontend Bound
    │   ├── Fetch Latency
    │   │   ├── ICache Misses
    │   │   ├── ITLB Misses
    │   │   ├── LCP
    │   │   ├── DSB Switches
    │   │   ├── Branch Resteers
    │   │   └── MS Switches
    │   └── Fetch Bandwidth
    └── Backend Bound
        ├── Memory Bound
        │   ├── L1 Bound
        │   ├── L2 Bound
        │   ├── L3 Bound
        │   ├── DRAM Bound
        │   │   ├── Memory Bandwidth
        │   │   └── Memory Latency
        │   └── Store Bound
        └── Core Bound
            ├── Divider
            ├── Port Utilization
            └── Serialization
```

A full TMA analysis from Level 1 to Level 4 requires dozens of hardware performance counters simultaneously, far exceeding the 4–8 programmable counters available on a typical Intel core. HPCToolkit addresses this through **subtree selection**: rather than attempting a full TMA traversal in a single run, users request the specific branch of the TMA hierarchy they wish to investigate. HPCToolkit then schedules only the counters required for that subtree, keeping the counter budget within hardware limits (if possible) and reducing costly multiplexing.

The recommended workflow is to start with a Level 1 and 2 overview (`topdown`), identify the dominant bottleneck category, and then re-profile with the corresponding subtree event (`topdown:retiring`, `topdown:frontend`, or `topdown:backend`) to get deeper attribution.

HPCToolkit provides the following TMA measurement events. Each event name is passed to `hpcrun` via the `-e` flag.

7.6.2 Event: `topdown`

Collects only **TMA Level 1 and Level 2** metrics using `PERF_METRICS` MSR and `CYCLES` as the trigger. This event is the recommended starting point for TMA analysis. It provides the two root categories (Active and Stall), their Level 1 sub-categories:

- **Retiring**: slots utilized by useful work, i.e., issued uops that eventually get retired.
- **Bad Speculation**: slots utilized by uops representing speculative computation that is found to be unnecessary and is discarded.
- **Frontend**: stalled slots due to instruction fetch and decode issues; the backend is ready to accept work but the frontend cannot supply it.
- **Backend**: slots stalls due to lack of backend resources; the frontend is supplying uops but the backend cannot accept or execute them.

and their Level 2 sub-categories, which provide a high-level characterization of pipeline efficiency without requiring many hardware counters.

Metrics collected:

Root	L1	L2	Description
Active	Retiring	Heavy Operations	Multi-uop instruction retirement
Active	Retiring	Light Operations	Single-uop instruction retirement
Active	Bad Speculation	Misprediction	Slots lost to branch misprediction recovery
Active	Bad Speculation	Machine Clear	Slots lost to full pipeline flushes
Stall	Frontend Bound	Fetch Latency	Sustained frontend delivery stalls
Stall	Frontend Bound	Fetch Bandwidth	Partial frontend throughput reduction
Stall	Backend Bound	Memory Bound	Backend stalls due to memory subsystem pressure
Stall	Backend Bound	Core Bound	Backend stalls due to execution resource contention

7.6.3 Event: `topdown:retiring`

Collects **TMA Level 1, Level 2, and the full Retiring subtree** (Levels 3 and 4 under Retiring). Use this event after a `topdown` run shows Retiring as the dominant Active category, indicating the application is compute-bound and optimization opportunities lie in instruction mix, vectorization, and code generation quality. High Retiring is generally positive but Heavy Operations and Uops Sequencer indicate inefficient instruction encoding.

Additional metrics collected beyond `topdown`:

L2	L3	L4	Description
Heavy Operations	Uops Sequencer	—	Uops fetched by the Microcode Sequencer
Heavy Operations	Few Operations	—	Instructions decoding to 2+ uops (non-MS)
Light Operations	FP Arithmetic	X87 Use	Legacy x87 FP uops
Light Operations	FP Arithmetic	FP Scalar	Arithmetic FP scalar uops retired
Light Operations	FP Arithmetic	FP Vector	Arithmetic FP vector uops retired (SSE/AVX)
Light Operations	Integer Operations	—	Integer uops retired
Light Operations	Memory Operations	—	Load and store uops retired

7.6.4 Event: `topdown:frontend`

Collects **TMA Level 1, Level 2, and the full Frontend Bound subtree** (Levels 3 and 4 under Frontend Bound). Use this event after a `topdown` run shows Frontend Bound as the dominant Stall category, indicating the CPU pipeline is starved of instructions from the fetch and decode stage. Common causes include instruction cache pressure from large code footprints, ITLB misses in code with many call sites, and branch misprediction-driven resteers.

Additional metrics collected beyond `topdown`:

L2	L3	Description
Fetch Latency	ICache Misses	Cycles stalled waiting for instruction cache fills
Fetch Latency	ITLB Misses	Cycles stalled waiting for instruction TLB walks
Fetch Latency	LCP	Cycles stalled due to Length Changing Prefixes in legacy decode
Fetch Latency	DSB Switches	Cycles stalled on transitions from DSB (uop cache) to MITE
Fetch Latency	Branch Resteers	Cycles stalled due to pipeline redirections after branch resolution
Fetch Latency	MS Switches	Cycles stalled on transitions to Microcode Sequencer delivery
Fetch Bandwidth	—	Slots lost to sustained below-peak frontend throughput

7.6.5 Event: `topdown:backend`

Collects **TMA Level 1, Level 2, and the full Backend Bound subtree** (Levels 3 and 4 under Backend Bound). Use this event after a `topdown` run shows Backend Bound as the dominant Stall category. Backend Bound is the most common bottleneck in HPC workloads. Memory Bound typically indicates cache hierarchy or DRAM pressure; Core Bound indicates execution unit saturation or serialization.

Additional metrics collected beyond `topdown`:

L2	L3	L4	Description
Memory Bound	L1 Bound	—	Stalls on L1 cache hits with long latency (bank conflicts, forwarding)
Memory Bound	L2 Bound	—	Stalls on loads satisfied from L2 cache
Memory Bound	L3 Bound	—	Stalls on loads satisfied from L3 cache
Memory Bound	DRAM Bound	Memory Bandwidth	DRAM stalls from bandwidth saturation
Memory Bound	DRAM Bound	Memory Latency	DRAM stalls from individual long-latency accesses
Memory Bound	Store Bound	—	Stalls from store buffer pressure or load-store forwarding blocks
Core Bound	Divider	—	Cycles occupied by non-pipelined integer or FP divide
Core Bound	Port Utilization	—	Cycles lost to uneven or saturated execution port loading
Core Bound	Serialization	—	Cycles stalled on serializing instructions (CPUID, MFENCE, I/O)

7.6.6 Event: `topdown:all`

Collects the **complete TMA hierarchy from Level 1 to Level 4** across all subtrees simultaneously. This requires the largest counter budget and may involve counter multiplexing on hardware with fewer than the required programmable counters. Use `topdown:all` when you want a comprehensive profile in a single run and are willing to accept the additional measurement overhead and potential multiplexing error.

For most workflows, using `topdown` first and then re-profiling with a targeted subtree event is preferable to `topdown:all`, as it produces more accurate measurements per subtree with lower overhead.

7.6.7 Combining Events

TMA events can be combined with each other and with conventional `perf_events` hardware or software events in the same `hpcrun` invocation. Each event is specified with a separate `-e` flag.

Combining two TMA subtrees:

```
hpcrun -e topdown:frontend -e topdown:backend ./myapp
```

This collects Level 1, Level 2, the full Frontend Bound subtree, and the full Backend Bound subtree in a single run. This is equivalent to running the two subtree events separately and merging results, but with higher overhead and more multiplexing inaccuracies.

Combining TMA with conventional events:

```
hpcrun -e topdown -e MEM_LOAD_RETIRED:L3_MISS ./myapp
```

This collects TMA Level 1 and 2 alongside explicit L3 miss sampling, allowing TMA's Backend Bound → Memory Bound attribution to be cross-validated with direct cache miss counts attributed to call paths.

Note on counter budget: When combining events, Linux kernel checks whether the total required hardware counters fit within the available PMU slots. If the combination exceeds the counter budget, it may fall back to time-division multiplexing, which can introduce sampling error.

7.6.8 Workflow Example

The following example demonstrates the recommended iterative TMA workflow using a naive matrix multiplication implementation:

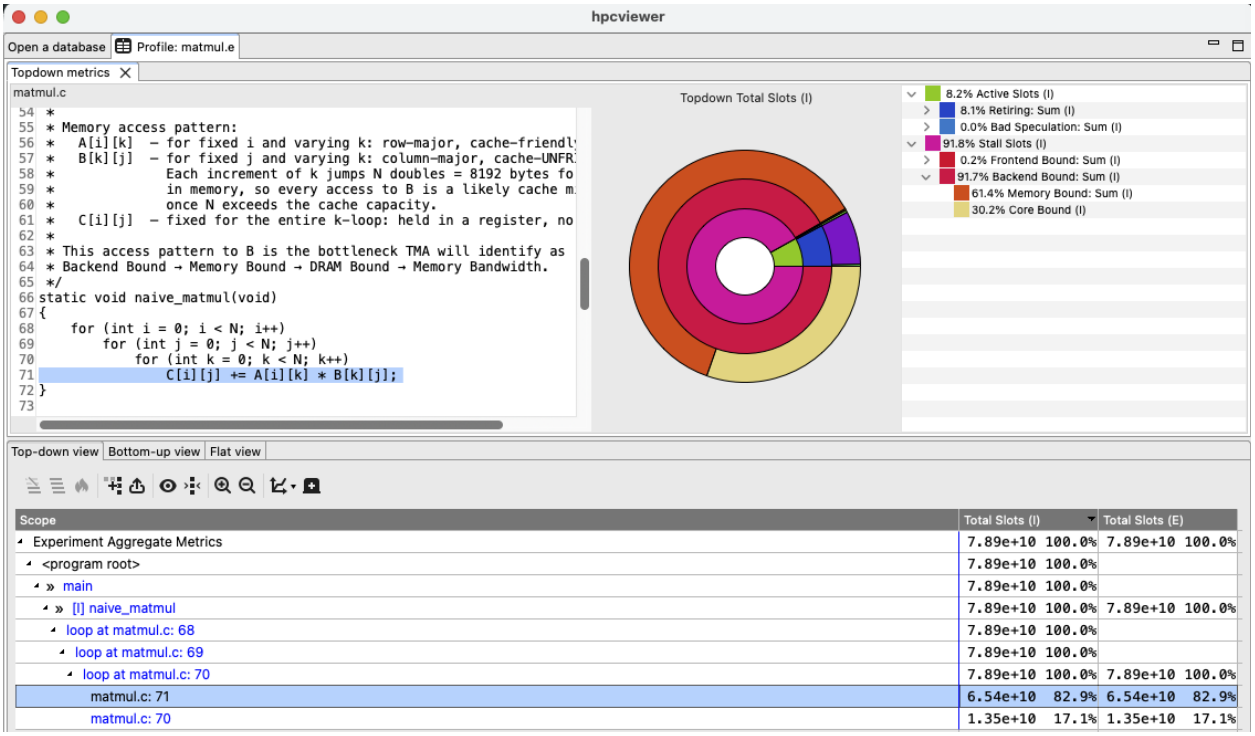
```
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            for (int k = 0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];
```

Step 1 — Performance Overview

Run the following commands to profile the matrix multiplication program (called `./matmul`) with the basic `topdown` event and examine the snapshot of performance metrics in `hpcviewer`.

```
hpcrun -e topdown ./matmul
hpcstruct hpctoolkit-matmul-measurements
hpcprof hpctoolkit-matmul-measurements
hpcviewer hpctoolkit-matmul-database
```

The first command is to profile `./matmul` with the `topdown` event, which collects CPU topdown metrics for level 1 and 2. Since we do not know the exact cause of the performance bottleneck, we use the `topdown` event to get a snapshot of the performance metrics with low overhead. The fourth command, we examine the top-down metrics in `hpcviewer` as shown below.



Note

In `hpcviewer`'s top-down metrics pane, the circle at the center of the initial donut chart represents the root *Total Slots* metric. The innermost ring of the donut chart represents the children of the metric at the graph's center. Each subsequent ring represents the children of the metrics in the ring it encloses. Child metrics in a ring occupy the same arc as their parent metric in the ring it encloses. In the tree-based representation of the metric hierarchy, each node in the tree contains a metric name and the percentage of the total slots associated with the metric.

Selecting any cell in an inclusive or exclusive *Total Slots* column of `hpcviewer`'s metric pane will refresh the top-down metrics pane to show the breakdown of *Total Slots* for the scope represented by the line containing the selected cell. Section *11.2.1.4* describes how to explore the data presented by the top-down metric hierarchy in `hpcviewer`'s top-down pane.

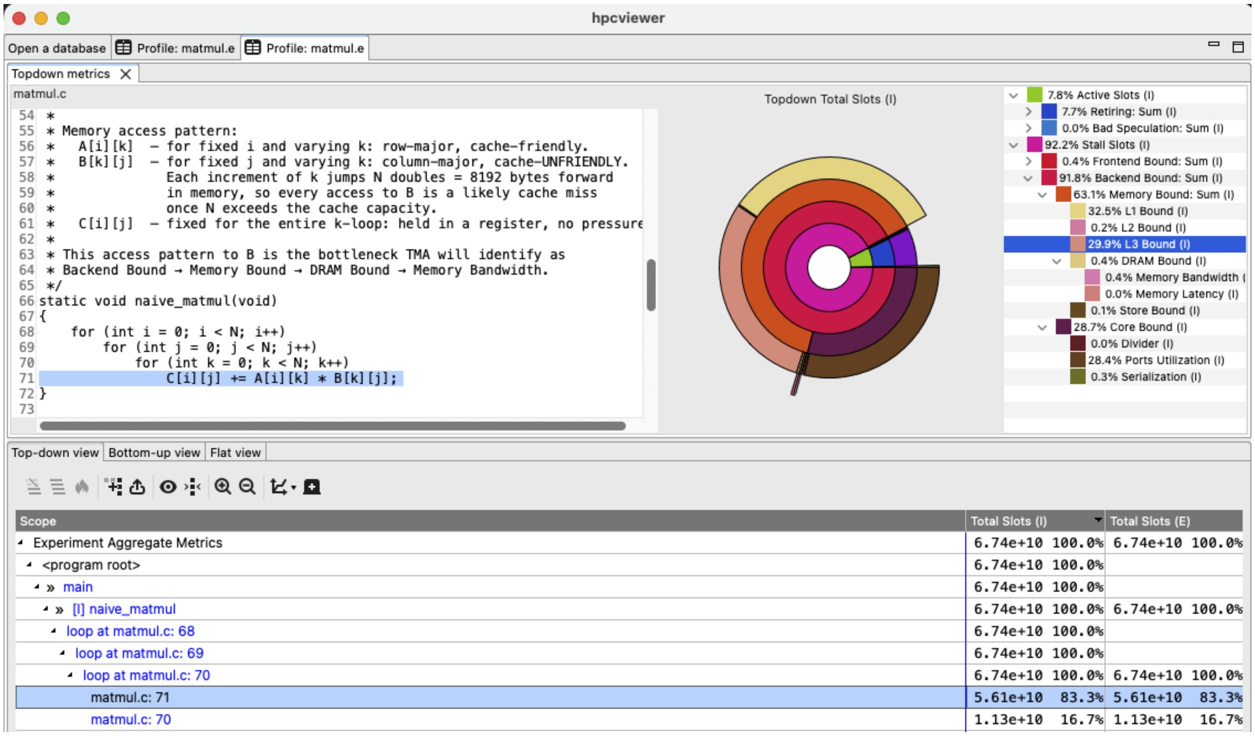
Both the donut chart (top-middle) and the tree (top-right) shows that **Stall Slots** dominates, with **Backend Bound** accounting for 92% of slots at the top-level loop and **Memory Bound** dominant at Level 2 (62% of the total slots). This indicates the naive column-major access pattern is creating memory pressure. The next step is to profile the backend bound subtree to get more detailed metrics.

Step 2 — Backend Bound Subtree

We profile with the `topdown:backend` event to get more detailed metrics, followed by running `hpcstruct` and `hpcprof` to generate the database.

```
hpcrun -e topdown:backend ./matmul
... # Run hpcstruct, hpcprof and hpcviewer as usual
```

As shown in hpcviewer snapshot below, the Level 3 and 4 metrics reveal that **L3 Bound** is one of the dominant sub-metrics (around 30% of the total time), confirming cache saturation from the strided access pattern. This diagnosis guides the optimization: loop interchange to row-major access, then cache blocking.

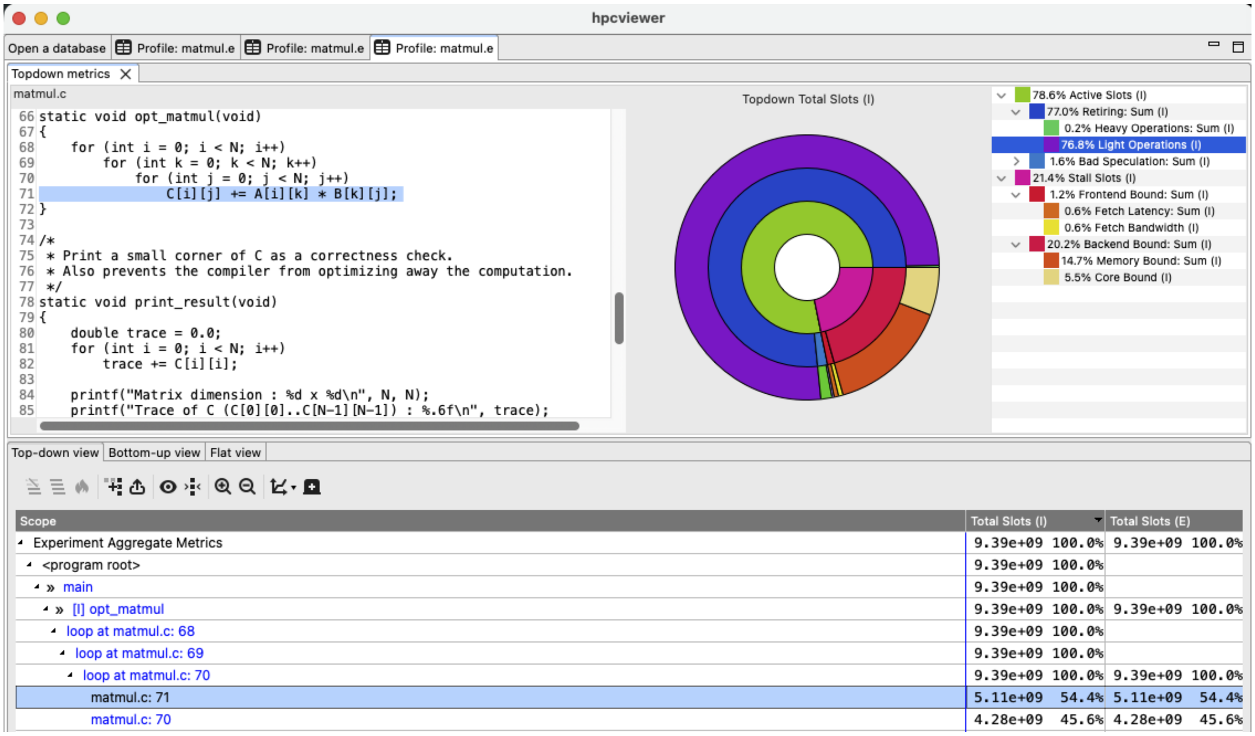


After recompiling the code, we verify the performance improvement by profiling with the `topdown` event again.

Step 3 — Verify After Optimization

```
hpcrun -e topdown -o hpctoolkit-matmul-opt-measurements ./matmul_optimized
... # Run hpcstruct, hpcprof and hpcviewer as usual
```

After optimization, `Backend Bound` decreases substantially to just 15%. The `Active` root grows to %79 from just %8, with `Retiring` increasing to almost 77%, and `Light Operations` of almost 77% reflecting improved cache-resident load/store throughput.



7.6.9 Interpretation Notes

Top-down measurement uses proxy sampling. HPCToolkit collects TMA metrics through *proxy sampling*: a normal hardware counter such as `CYCLES` acts as the sampling trigger and fires a periodic interrupt. At each interrupt, `hpcrun` reads a snapshot of the TMA hardware counters — specifically the `PERF_METRICS` MSR and `TOPDOWN.SLOTS` — which have been accumulating pipeline slot fractions since the previous sample. The trigger event and the measured events (`PERF_METRICS`) are therefore different: The trigger event determines *when* a sample is taken, while `PERF_METRICS` provides *what* is measured. Because the snapshot reflects an inter-sample interval of thousands of cycles rather than a single instruction, TMA attribution is not accurate at the instruction level but is reliable and meaningful at the granularity of loops and functions.

Multiplexing and accuracy. When the number of required hardware counters exceeds the available PMU slots, Linux kernel time-multiplexes counters across the measurement period and scales the results. Multiplexed measurements are less accurate than direct measurements. The `topdown` event are each designed to fit within a typical hardware counter budget without multiplexing. `topdown:retiring`, `topdown:frontend`, and `topdown:backend`, `topdown:all` and combinations of multiple subtree events are more likely to require multiplexing.

Hardware requirements. TMA events require:

- Intel Sapphire Rapids or newer microarchitecture
- `perf_event_paranoid` <= 1, or `CAP_PERFMON` capability

7.6.10 Summary of Events

Event	TMA Coverage	Recommended Use
`topdown`	Active + Stall roots, L1–L2 all subtrees	First-pass bottleneck identification
`topdown:retiring`	L1–L2 all + L3–L4 Retiring subtree	Diagnose instruction-mix and vectorization issues
`topdown:frontend`	L1–L2 all + L3–L4 Frontend Bound subtree	Diagnose instruction fetch and decode stalls
`topdown:backend`	L1–L2 all + L3–L4 Backend Bound subtree	Diagnose memory or execution resource bottlenecks
`topdown:all`	L1–L4 all subtrees	Comprehensive single-run profile (higher overhead)

Tip

Events may be combined with `-e` flags, and its sampling frequency can be adjusted with `@` suffix, such as:

```
hpcrun -e topdown:frontend@f100 -e topdown:backend@f150 my_program
```

CHAPTER

EIGHT

MONITORING MPI

HPCToolkit's measurement subsystem can measure each process and thread in an execution of an MPI program. HPC-Toolkit supports C, C++, Fortran, and Python MPI programs. HPCToolkit can be used with pure MPI programs as well as hybrid programs that use multithreading, e.g., OpenMP or Pthreads, within MPI processes.

A single installation of HPCToolkit should work with any MPI implementation. You don't need to provide an `mpi.h` include path when building HPCToolkit and you don't need to compile multiple versions of HPCToolkit, one for each MPI implementation. It has been tested and used with MPICH, MVAPICH and OpenMPI.

Hint

A thoughtful reader might wonder how one installation of HPCToolkit can interoperate with MPI implementations that use different representations for `MPI_COMM_WORLD`. Instead of calling `MPI_Comm_rank` directly, `hpcrun` waits for the application to call `MPI_Comm_rank` and captures the returned rank index.

8.1 Measuring MPI Ranks

For a dynamically linked application binary `app`, use a command line similar to the following example:

```
<mpi-launcher> hpcrun -e <event>:<period> ... app [app-arguments]
```

Observe that the MPI launcher (e.g., `srun`, `mpiexec`, or `mpirun`) is used to launch `hpcrun`, which then launches the application program.

In this example, `s3d_f90.x` is the Fortran S3D program compiled with OpenMPI's `mpif90` and run with the command line

```
mpiexec -n 4 hpcrun -e cycles ./s3d_f90.x
```

This produced 12 files in the following abbreviated `ls` listing:

```
krentel 1889240 Feb 18  s3d_f90.x-000000-000-72815673-21063.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000000-001-72815673-21063.hpcrun
krentel 1914680 Feb 18  s3d_f90.x-000001-000-72815673-21064.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000001-001-72815673-21064.hpcrun
krentel 1908030 Feb 18  s3d_f90.x-000002-000-72815673-21065.hpcrun
krentel    7974 Feb 18  s3d_f90.x-000002-001-72815673-21065.hpcrun
krentel 1912220 Feb 18  s3d_f90.x-000003-000-72815673-21066.hpcrun
krentel    9848 Feb 18  s3d_f90.x-000003-001-72815673-21066.hpcrun
krentel  147635 Feb 18  s3d_f90.x-72815673-21063.log
krentel  142777 Feb 18  s3d_f90.x-72815673-21064.log
```

(continues on next page)

(continued from previous page)

```
krentel  161266 Feb 18  s3d_f90.x-72815673-21065.log
krentel  143335 Feb 18  s3d_f90.x-72815673-21066.log
```

The measurement files show that the execution consisted of four processes with two threads per process. Looking at the file names, `s3d_f90.x` is the name of the program binary, `000000-000` through `000003-001` are the MPI rank and thread numbers, and `21063` through `21066` are the process IDs.

We see from the file sizes that OpenMPI is spawning one helper thread per process. Technically, the smaller `.hpcrun` files imply only a smaller calling-context tree (CCT), not necessarily fewer samples. But in this case, the helper threads are not doing much work.

For the best results with HPCToolkit, an application's first call to `MPI_Comm_rank` should use communicator `MPI_COMM_WORLD`. Nearly all MPI programs already do this, so this is rarely a problem. For example, in C, a program might begin with:

```
int main(int argc, char **argv)
{
  int size, rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  ...
}
```

The call to `MPI_Comm_rank` should be unconditional, that is all processes should make this call. In the code above, the call to `MPI_Comm_size` is not necessary for `hpcrun`, although most MPI programs normally call both `MPI_Comm_size` and `MPI_Comm_rank`.

If the first call to `MPI_Comm_rank` is not passed `MPI_COMM_WORLD`, HPCToolkit should be able to measure the application anyway, although it may not properly identify MPI ranks.

CHAPTER

NINE

MONITORING OPENMP

HPCToolkit includes an implementation of the OpenMP Tools API, known as OMPT, initially defined in OpenMP 5.0. The OMPT interface enables HPCToolkit to extract enough information to reconstruct user-level calling contexts from implementation-level measurements.

No special switches are needed to monitor OpenMP threads, tasks, and offloading with HPCToolkit. HPCToolkit enables the OMPT interface for monitoring OpenMP by default.

In the unlikely event that there is a bad interaction between HPCToolkit's support for the OMPT interface and an OpenMP runtime, OMPT support may be disabled when measuring your code with HPCToolkit by setting an environment variable, as shown below:

```
export OMP_TOOL=disabled
```

9.1 Monitoring OpenMP on the Host

Support for OpenMP 5.0 and OMPT is available in OpenMP runtimes for LLVM, AMD, Intel, and IBM compilers. Support in these implementations is mostly complete, although there are some quirks with OMPT support for tracking computation offloaded on TARGET devices.

As of this writing, a popular OpenMP runtime that lacks OMPT support is the GCC compiler suite's `libgomp`. Fortunately, the LLVM OpenMP runtime, which supports OMPT, is compatible with `libgomp`, at least on the host.

Tip

For applications not using OpenMP's TARGET construct, one can typically use LLVM's ABI-compatible OpenMP runtime (`libomp.so`) as a replacement for `libgomp` by (1) creating a link to LLVM's `libomp` runtime whose name matches the `libgomp` dependence in your binary, and (2) putting the directory containing the link on your `LD_LIBRARY_PATH` as shown below:

```
ln -s /path/to/libomp.so libgomp.so.1
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

In OpenMP implementations without support for OMPT, HPCToolkit records and reports implementation-level measurements of program executions. At this level, work is typically partitioned between a primary (master) thread and one or more worker threads. Work executed on the master thread can be associated with its full user-level calling context and is reported under `<program root>`. However, OpenMP regions and tasks executed by worker threads typically can't be associated with the calling context in which regions or tasks were launched. Instead, this work is attributed to a worker thread outer context that polls for work, finds the work, and executes the work. HPCToolkit reports such work under `<thread root>`.

When an OpenMP runtime supports OMPT, HPCToolkit can gather information that enables it to reconstruct a global, user-level view of the parallelism. Using OMPT, HPCToolkit can attribute performance for parallel regions incurred by worker threads back to the calling contexts in which those parallel regions were invoked. In such cases, most or all work performance is attributed back to global user-level calling contexts that are descendants of `<program root>`.

> **Note**
>
> Even when an OpenMP runtime supports OMPT, there may be some costs that cannot be attributed back to a global user-level calling context in an OpenMP program. For instance, costs associated with idle worker threads (that can't be associated with any parallel region) may be attributed to `<omp idle>`, and small costs associated with runtime startup may be attributed to `<thread root>`.

9.2 Monitoring OpenMP Offloading on GPUs

HPCToolkit can measure computation offloaded onto GPUs via an OpenMP TARGET construct and attribute performance for offloaded computation back to the host calling context that launched it. This functionality is enabled when

1. The OpenMP runtime supports OMPT, and
2. The appropriate GPU monitoring flags are passed to `hpcrun` as described in Section *9.1*.

The sections below detail known support and quirks for vendor implementations of OpenMP.

9.2.1 AMD OpenMP

The OpenMP runtime included with AMD's ROCm 5.1 and later releases contains OMPT support for monitoring and attributing host computations as well as computations offloaded to AMD GPUs using OpenMP TARGET. This support is available for applications compiled with the `amdclang` and `amdclang++` compilers included in ROCm.

> **Note**
>
> As of writing, AMD only provides an idiosyncratic strategy for reporting offloaded OpenMP computations that does not follow the OpenMP standard. As a result, calling contexts for computations offloaded onto AMD GPUs using OpenMP TARGET may include implementation details of AMD's OpenMP and ROCm runtimes.

9.2.2 Intel OneAPI OpenMP

Intel's OneAPI `ifx`, `icx`, and `icpx` compilers and runtime support OpenMP offloading atop Intel's latest GPU-enabled Level Zero runtime. The OneAPI OpenMP runtime provides support for the OMPT tools interface.

The implementation of host-side OMPT callbacks in the OneAPI OpenMP runtime is sufficient for attributing both CPU and GPU work to global, user-level calling contexts rooted at `<program root>`. However, at this writing, the OneAPI OpenMP runtime lacks up-to-date OMPT support for monitoring OpenMP offloading. As a result, calling contexts for computations offloaded onto Intel GPUs using OpenMP TARGET may include implementation details of OneAPI OpenMP and Intel's Level Zero runtime.

9.2.3 NVIDIA OpenMP

As of writing, NVIDIA's OpenMP runtime for its CUDA and HPC SDK compilers (`nvc`, `nvcc`, `nvc++`, `nvfortran`) lacks support for OMPT. Without OMPT support, HPCToolkit attributes performance measurements of GPU computations offloaded using OpenMP TARGET to implementation-level calling contexts.

9.2.4 HPE's Cray OpenMP

HPE's Cray compilers `craycc`, `craycxx`, and `crayftn` have host-side support for the OMPT interface, which enables HPCToolkit to reconstruct user-level calling contexts from implementation-level measurements.

We have not investigated whether the implementation of OpenMP offloading for NVIDIA GPUs in the Cray OpenMP runtime supports the OMPT interface.

As of this writing, AMD's Rocprofiler-sdk doesn't report the performance of computations offloaded onto AMD GPUs by the Cray OpenMP runtime.

9.2.5 GCC OpenMP

It appears that GCC's support for OpenMP offloading can only be used with `libgomp`. Since we lacked GCC compilers configured for OpenMP offloading on the systems we are using, we didn't experiment with with GCC's OpenMP offloading to see whether vendor tool libraries observe GPU computations offloaded by GCC.

CHAPTER

TEN

MONITORING GPUS

HPCToolkit can measure both CPU and GPU performance of GPU-accelerated applications. It measures CPU performance using asynchronous sampling, as described in Section *6.1*, and it measures GPU operations using tool libraries or interfaces provided by GPU vendors. A single build of HPCToolkit can support GPUs from AMD, NVIDIA, and Intel.

In this chapter, we describe HPCToolkit's capabilities for profiling, tracing, and PC sampling of GPU-accelerated applications that use a variety of vendor runtimes and tool interfaces, including ROCm (AMD), CUDA (NVIDIA), Level Zero (Intel), OpenCL (all GPUs), and/or OpenMP (all GPUs).

Section *9.1* provides a quick start guide for using HPCToolkit to measure GPU-accelerated applications. Subsequent sections describe profiling (Section *9.2*), tracing (Section *9.3*), PC sampling (Section *9.4*), top-down microarchitectural analysis (Section *9.5*), and associated platform-independent metrics. The chapter concludes with sections on AMD GPUs (Section *9.6*), NVIDIA GPUs (Section *9.7*), and Intel GPUs (Section *9.8*) that discuss platform-specific measurement details or metrics.

10.1 GPU Measurement Quickstart

HPCToolkit provides a vendor-independent monitoring substrate for measuring the performance of GPU-accelerated applications. This substrate interfaces with NVIDIA's CUPTI (CUDA Performance Tools Interface) and AMD's Rocprofiler-sdk. It also supports intercepting calls to the OpenCL API and Intel's Level Zero API because OpenCL and Level Zero lack a measurement API like CUPTI or Rocprofiler-sdk.

For each of the supported GPU runtimes (ROCm, CUDA, Level Zero, and OpenCL), HPCToolkit supports several monitoring options. All of the GPU runtimes support profiling and tracing of GPU operations. ROCm, CUDA, and Level Zero support instruction-level measurement within GPU kernels using PC sampling. Level Zero also supports instruction-level performance measurement within GPU kernels using binary instrumentation with Intel's GTPin.

Table *9.1* shows arguments to `hpcrun` that can be used to monitor the performance of GPU operations offloaded by HIP or OpenMP on AMD GPUs.

Table 1: Table 9.1: Monitoring performance on AMD GPUs when using AMD's HIP and OpenMP programming models atop AMD's ROCm runtime.

Argument to `hpcrun`	What is monitored
`-e gpu=rocm`	coarse-grain profiling of AMD GPU operations
`-e gpu=rocm -t`	coarse-grain profiling and tracing of AMD GPU operations
`-e gpu=rocm -tt`	coarse-grain profiling and high-resolution tracing of AMD GPU operations
`-e gpu=rocm,pc[={sw,hw}][@k]`	coarse-grain profiling of GPU operations; fine-grain profiling within GPU kernels using PC sampling every 2^k cycles (hw) or 2^k ns (sw).

AMD GPUs support two types of PC sampling: hardware (`hw`) and software-only (`sw`). `sw` PC sampling is the default mode because it is available on MI200+; `hw` PC sampling is available only on MI300+. For`sw` PC sampling, HPCToolkit's default period is 2^20 ns. For `hw` PC sampling, the default period is 2^20 cycles.

Normal tracing (`-t`) collects CPU trace entries using asynchronous sampling and GPU trace entries at the beginning and end of every GPU operation. High-resolution tracing (`-tt`) additionally records a CPU trace entry each time a CPU thread launches a GPU operation, if any. For programs that launch many GPU operations per second, `-tt` will collect a more useful trace than `-t`. Without `-tt`, the activity seen on a CPU trace line at the time it launches a GPU operation is often from far earlier in the execution, which can make it difficult to relate to concurrent CPU and GPU activity.

Tip

For all GPU events, you may combine CPU and GPU profiling events in the same run. For example, `hpcrun -e REALTIME -e gpu=rocm -t ...` would profile and trace activity on the CPU using the default CPU sampling period while profiling and tracing GPU operations as well.

Tip

You can use `-t` or `-tt` while collecting PC samples to see *qualitative* timings; however, the overhead of processing PC samples is likely to distort timings in traces. For accurate timings in traces, we recommend collecting PC samples and traces in different executions.

Table 9.2 shows arguments to `hpcrun` that can be used to monitor the performance of GPU operations offloaded by CUDA or OpenMP on NVIDIA GPUs. On NVIDIA GPUs, HPCToolkit's default period for PC sampling is 2^20 cycles.

Table 2: Table 9.2: Monitoring performance on NVIDIA GPUs when using CUDA or OpenMP programming models.

Argument to `hpcrun`	What is monitored
`-e gpu=cuda`	coarse-grain profiling of GPU operations
`-e gpu=cuda -t`	coarse-grain profiling and tracing of GPU operations
`-e gpu=cuda -tt`	coarse-grain profiling and high-resolution tracing of GPU operations
`-e gpu=cuda,pc[@k]`	coarse-grain profiling of GPU operations; fine-grain profiling within GPU kernels using PC sampling every 2^k cycles

Table 9.3 shows available options for using HPCToolkit that can be used to monitor the performance of GPU operations offloaded by SYCL or OpenMP to Intel's Level Zero runtime. On Intel GPUs, HPCToolkit's default period for PC sampling is 2^20 ns.

Table 3: Table 9.3: Monitoring performance on Intel GPUs when using SYCL or OpenMP programming models atop Intel's Level Zero runtime.

Argument to `hpcrun`	What is monitored
`-e gpu=level0`	coarse-grain profiling of GPU operations runtime
`-e gpu=level0 -t`	coarse-grain profiling and tracing of GPU operations
`-e gpu=level0 -tt`	coarse-grain profiling and high-resolution tracing of GPU operations
`-e gpu=level0,inst=count`	coarse-grain profiling of GPU operations; fine-grain measurement within kernels using binary instrumentation to count instructions
`-e gpu=level0,pc[@k]`	coarse-grain profiling of GPU operations; fine-grain profiling within GPU kernels using PC sampling every 2^k ns

Table 9.4 shows the possible command-line arguments to `hpcrun` for monitoring OpenCL programs.

Table 4: Table 9.4: Monitoring performance on GPUs when using the OpenCL programming model.

Argument to `hpcrun`	What is monitored
`-e gpu=opencl`	coarse-grain profiling of GPU operations using a platform's OpenCL runtime
`-e gpu=opencl -t`	coarse-grain profiling and tracing of GPU operations using a platform's OpenCL runtime
`-e gpu=opencl -t`	coarse-grain profiling and high-resolution tracing of GPU operations using a platform's OpenCL runtime

If OpenMP is implemented using the OMPT interface, then no special arguments are needed to turn on monitoring.

10.2 Profiling GPU Operations

In most cases, HPCToolkit reports performance metrics for GPU operations in a vendor-neutral way.

Coarse-grain profiling attributes to each calling context the total time of all GPU operations initiated in that context. Table *9.5* shows the classes of GPU operations for which timings are collected. For AMD and NVIDIA GPUs, HPCToolkit also reports GPU kernel characteristics, including including register usage, thread count per block, and theoretical occupancy as shown in Table *9.6*. HPCToolkit derives a theoretical GPU occupancy metric as the ratio of the actual number of threads in a block to be scheduled on an AMD compute unit or an NVIDIA streaming multiprocessor to the maximum number of threads supported in a block.

In addition, HPCToolkit records metrics for operations performed including memory allocation and deallocation (Table *9.7*), memory set (Table *9.8*), explicit memory copies (Table *9.9*), and synchronization (Table *9.10*). These operation metrics are available for GPUs for AMD, Intel, and NVIDIA GPUs. While summary metrics in Table *9.1* and Table *9.2* are shown by default in hpcviewer, metrics marked with an asterisk in Tables *9.7*, *9.8*, *9.9*, and *9.10* are hidden by default to avoid overwhelming users with many columns of metrics. In `hpcviewer`, one can reveal any hidden metric by opening its metric pane marking the metric as visible.

Table 5: Table 9.5: GPU operation timings.

Metric	Description
GKER (s)	GPU time: kernel execution (seconds)
GMEM (s)	GPU time: memory allocation/deallocation (seconds)
GMSET (s)	GPU time: memory set (seconds)
GXCOPY (s)	GPU time: explicit data copy (seconds)
GSYNC (s)	GPU time: synchronization (seconds)
GPUOP (s)	Total GPU operation time: sum of all metrics above

Table 6: Table 9.6: GPU kernel characteristic metrics.

Metric	Description
GKER:STMEM (B)	GPU kernel: static memory (bytes)
GKER:DYMEM (B)	GPU kernel: dynamic memory (bytes)
GKER:LMEM (B)	GPU kernel: local memory (bytes)
GKER:WARP_ACT	GPU kernel: SM warp parallelism, actual
GKER:WARP_AVL	GPU kernel: SM warp parallelism, available
GKER:THR_REG	GPU kernel: thread register count
GKER:BLK_THR	GPU kernel: thread count per grid block
GKER:BLKS_AVG	GPU kernel: average grid block count per launch
GKER:BLK_SM (B)	GPU kernel: block local memory (bytes)
GKER:COUNT	GPU kernel: launch count
GKER:OCC_THR	GPU kernel: theoretical occupancy 100 * (WARP_ACT / WARP_AVL)

Table 7: Table 9.7: GPU memory allocation and deallocation. Note: metrics marked with * are hidden by default.

Metric	Description
GMEM:UNK (B)*	GPU memory alloc/free: unknown memory kind (bytes)
GMEM:PAG (B)*	GPU memory alloc/free: pageable memory (bytes)
GMEM:PIN (B)*	GPU memory alloc/free: pinned memory (bytes)
GMEM:DEV (B)*	GPU memory alloc/free: device memory (bytes)
GMEM:ARY (B)*	GPU memory alloc/free: array memory (bytes)
GMEM:MAN (B)*	GPU memory alloc/free: managed memory (bytes)
GMEM:DST (B)*	GPU memory alloc/free: device static memory (bytes)
GMEM:MST (B)*	GPU memory alloc/free: managed static memory (bytes)
GMEM:COUNT	GPU memory alloc/free: count

Table 8: Table 9.8: GPU memory set metrics. Note: metrics marked with * are hidden by default.

Metric	Description
GMSET:UNK (B)*	GPU memory set: unknown memory kind (bytes)
GMSET:PAG (B)*	GPU memory set: pageable memory (bytes)
GMSET:PIN (B)*	GPU memory set: pinned memory (bytes)
GMSET:DEV (B)*	GPU memory set: device memory (bytes)
GMSET:ARY (B)*	GPU memory set: array memory (bytes)
GMSET:MAN (B)*	GPU memory set: managed memory (bytes)
GMSET:DST (B)*	GPU memory set: device static memory (bytes)
GMSET:MST (B)*	GPU memory set: managed static memory (bytes)
GMSET:COUNT	GPU memory set: count

Table 9: Table 9.9: GPU explicit memory copy metrics. Note: metrics marked with * are hidden by default.

Metric	Description
GXCOPY:UNK (B)*	GPU explicit memory copy: unknown kind (bytes)
GXCOPY:H2D (B)*	GPU explicit memory copy: host to device (bytes)
GXCOPY:D2H (B)*	GPU explicit memory copy: device to host (bytes)
GXCOPY:H2A (B)*	GPU explicit memory copy: host to array (bytes)
GXCOPY:A2H (B)*	GPU explicit memory copy: array to host (bytes)
GXCOPY:A2A (B)*	GPU explicit memory copy: array to array (bytes)
GXCOPY:A2D (B)*	GPU explicit memory copy: array to device (bytes)
GXCOPY:D2A (B)*	GPU explicit memory copy: device to array (bytes)
GXCOPY:D2D (B)*	GPU explicit memory copy: device to device (bytes)
GXCOPY:H2H (B)*	GPU explicit memory copy: host to host (bytes)
GXCOPY:P2P (B)*	GPU explicit memory copy: peer to peer (bytes)
GXCOPY:COUNT	GPU explicit memory copy: count

Table 10: Table 9.10: GPU synchronization metrics. Note: metrics marked with * are hidden by default.

Metric	Description
GSYNC:UNK (s)*	GPU synchronizations: unknown kind
GSYNC:EVT (s)*	GPU synchronizations: event
GSYNC:STRE (s)*	GPU synchronizations: stream event wait
GSYNC:STR (s)*	GPU synchronizations: stream
GSYNC:CTX (s)*	GPU synchronizations: context
GSYNC:COUNT	GPU synchronizations: count

Figure 9.1 shows a screenshot of a profile view of Quicksilver - a proxy application from Lawrence Livermore National Laboratory. The figure shows a top-down view of heterogeneous calling contexts that span both CPU and GPU. In the middle of the figure is a placeholder `<gpu kernel>` that is inserted by HPCToolkit to mark the transition between CPU and GPU code in the call stack. Above the placeholder is a CPU calling context where the GPU kernel was invoked. Below the `<gpu kernel>` placeholder, we see `CycleTrackingKernel` - the only non-trivial kernel in this code. Columns in the figure's metric table show metrics associated with `CycleTrackingKernel`. From these we see that on average, the kernel uses 12 of 64 possible warps per SM (`WARP_ACT` vs. `WARP_AVL`). The kernel uses 158 registers (`GKER:SREG`), uses 128 threads per grid block (`GKER:BLK_THR`), was launched with an average of 5 grid blocks (`GKER:BLKS_AVG`), was launched 195 times (`GKER:COUNT Sum(E)`), with an average occupancy 18.8% (`GKER:OCC_THR`), calculated as 100 * (`WARP_ACT` / `WARP_AVL`).

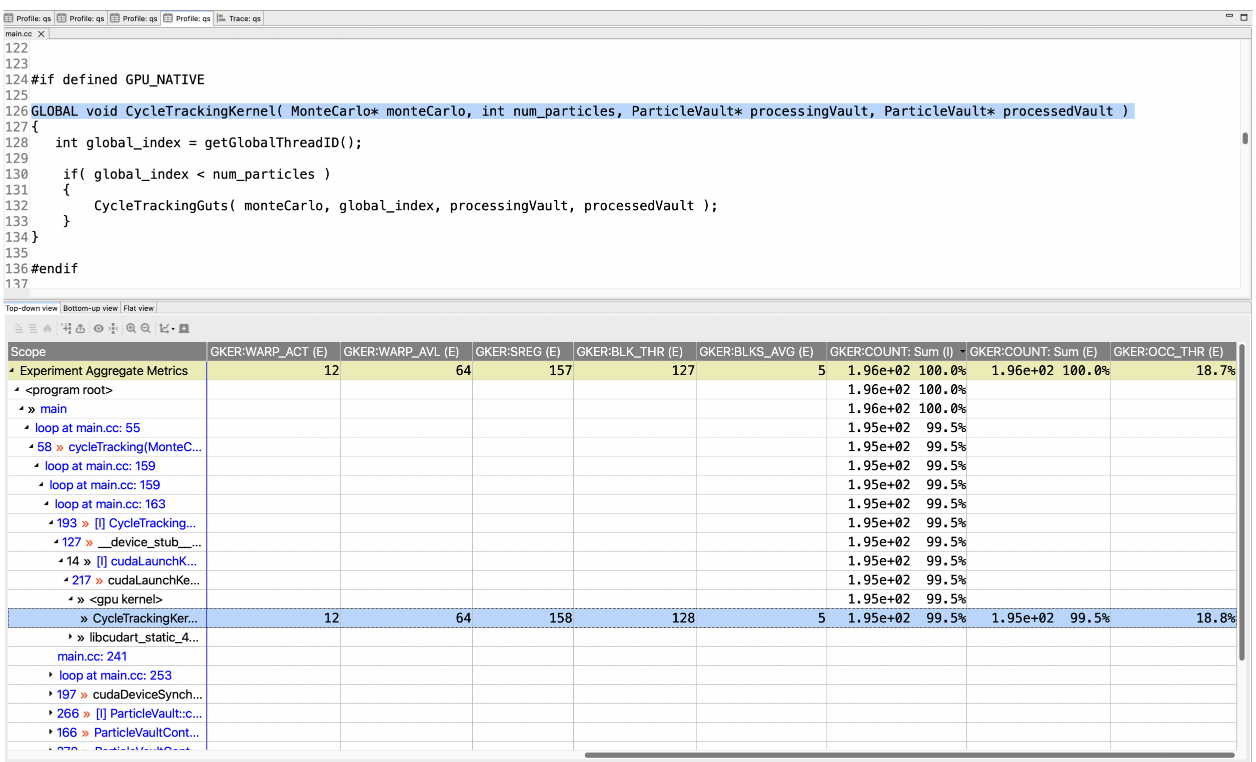


Fig. 1: Figure 9.1: A screenshot of `hpcviewer` for the GPU-accelerated Quicksilver proxy app with kernel metrics.

Important

The cost of profiling and tracing GPU operations using monitoring libraries provided by GPU vendors can't be controlled by HPCToolkit. Unlike CPU monitoring based on asynchronous sampling, GPU performance monitoring uses callback interfaces provided by vendor monitoring libraries to monitor the initiation and completion of each GPU operation. Accordingly, the overhead of GPU performance monitoring depends upon how frequently GPU operations are initiated.

For example, when monitoring CUDA applications on NVIDIA GPUs with NVIDIA's CUPTI interface, we have seen the execution time double when profiling and tracing a GPU-accelerated application that launches kernels very frequently. Measures of GPU activity are accurate; however, processing the torrent of measurement data from one or more GPUs can add considerable CPU overhead. For that reason, you should expect some CPU overhead associated with profiling and tracing of GPU operations and account for it in your analysis.

For instance, if a GPU-accelerated program runs in 1000 seconds without HPCToolkit monitoring GPU activity but slows to 2000 seconds when GPU profiling and tracing is enabled, then if GPU profiles and traces show that the GPU is active for 25% of the execution time, one should mentally re-scale the accurate measurements of GPU activity by considering the `2x` dilation that occurs when monitoring GPU activity. Without monitoring, one would expect the same level of GPU activity, but the host time would be twice as fast. Thus, without monitoring, the ratio of GPU activity to host activity would be roughly double.

10.3 Tracing GPU Operations

HPCToolkit also supports tracing of activities on GPU streams on AMD, Intel, and NVIDIA GPUs when using ROCm, Level Zero, CUDA, OpenCL or OpenMP offloading. Tracing of GPU activities will be enabled any time GPU monitoring is enabled and `hpcrun`'s tracing is enabled with `-t|--trace`.

For tracing GPU-accelerated workloads, HPCToolkit offers a better tracing option `-tt|--ttrace`, which collects a boosted resolution trace of CPU threads. Like `-t`, `-tt` records a call path trace for CPU threads based on asynchronous sampling of a time-based sampling metric, such as `CPUTIME`, `REALTIME`, or `cycles`. Unlike `-t`, `-tt` also records a sample for a CPU thread as it launches each GPU operation (if any). Without `-tt`, the activity seen on a CPU trace line at the time a kernel is launched may be from long ago, which makes it hard to understand how CPU activity relates to GPU activity. Note that using `-tt` disturbs the statistical properties of CPU traces since it adds non-sample events whenever a thread launches a GPU operation.

Important

The next section describes instruction-level measurement within GPU kernels using PC sampling. Unless you are using a very large sampling period, collecting and attributing PC samples may add significant CPU overhead to a GPU-accelerated code. You can assess PC sampling overhead by comparing the execution time of a GPU-accelerated code with and without PC sampling. Unless the PC sampling frequency you are using is known to measure executions with low overhead, tracing using either `-t` or `-tt` is not recommended when PC sampling is enabled as the overhead of PC sampling will affect timings in traces.

Important

During execution `hpcrun` creates CPU tool threads to record traces of GPU activities. The work performed by such tool threads is not reported in profiles and traces shown by HPCToolkit. By default, `hpcrun` creates one tracing thread per 256 GPU streams. To adjust the number of GPU streams per tracing thread, see the settings for `HPCRUN_CONTROL_KNOBS` in Section *14*. When mapping a GPU-accelerated node program onto a node, you may need to consider provisioning additional hardware threads or cores to accommodate these tracing threads; otherwise, they may compete with application threads for CPU resources, which may degrade the performance of your program.

10.4 PC Sampling on GPUs

HPCToolkit collects Program Counter (PC) samples to measure instruction-level performance metrics on AMD, Intel and NVIDIA GPUs. Measurement of PC samples is supported for CUDA on NVIDIA GPUs, HIP on AMD GPUs, SYCL/DPC++ on Intel GPUs, and OpenMP on all GPUs. None of the GPU vendors support PC sampling for OpenCL.

All GPU implementations of PC sampling periodically interrupt execution of a kernel warp or wavefront running on a GPU to record the value of its program counter to identify the instruction that an AMD compute unit, an Intel execution unit, or an NVIDIA streaming multiprocessor is trying to execute. HPCToolkit attributes much of the information gathered from sample records in a vendor-neutral way. HPCToolkit also supports instruction-level measurement on Intel GPUs using binary instrumentation.

PC sampling on Intel and NVIDIA GPUs is supported by hardware. AMD GPUs have two strategies for collecting PC samples: one supported entirely in software and one supported by hardware. Each of these implementations collect instruction-level metrics within GPU kernels.

Metrics collected with PC sampling are estimates for two reasons:

First, when sampling with a period of `N` instructions, each time a sample is recorded, HPCToolkit charges `N` cycles to the machine instruction indicated by the program counter. This strategy has the advantage that the measurement results should be independent of the sampling period. NVIDIA GPUs and AMD's hardware-supported stochastic sampling both use sampling periods specified using a number of instructions. When measuring a program using PC sampling, one can specify a sampling period by adding an `@k` to the end of the argument enabling PC sampling; this will set the sampling period to `2^k`.

Second, Intel's Level Zero runtime and AMD's software implementation of PC sampling for MI200+ GPUs specify the sampling period as time rather than the number of instructions. Like the aforementioned implementations with instruction-level periods, HPCToolkit's implementation of PC sampling for Intel GPUs and AMD's software implementation of PC sampling use `@k` to indicate a sampling period of 2^k; however, the period for these implementations is in nanoseconds rather than cycles.

For both Intel and AMD GPUs, base clock frequencies are approximately 1GHz. While boosted clock frequencies can be as much as twice as fast, HPCToolkit doesn't attempt to correct for that. Instead, HPCToolkit treats nanoseconds and clock cycles as interchangeable. Even when using time-based periods, HPCToolkit reports GPU measurements as cycles. With clock scaling, this may be off by as much as a factor of two. However, for the purpose of identifying which source lines within kernels consume resources, metrics will accumulate at the proper locations with the proper relative costs for different locations regardless of whether the metric actually is a measure of GPU cycles or nanoseconds.

When using AMD's software support for PC sampling, cycles is the only metric that can be collected; when processing a PC sample at runtime, no information is available about whether the sampled instruction issued or stalled. In contrast, hardware support for PC sampling on AMD, Intel, and NVIDIA GPUs reports whether the sampled instruction was issued or whether the sampled instruction stalled instead. All GPU instruction-level metrics measured (e.g. cycles, issues, stalls) are scaled by multiplying the actual count of samples by the sample period.

HPCToolkit considers a PC sample to represent a stall only if the latency of the instruction's stall is exposed. On an NVIDIA GPU, HPCToolkit only considers a sampled instruction to be a stall if NVIDIA's CUPTI reports the instruction as a latency sample, which means that no instruction issued on the GPU in the cycle when the sample was recorded. On an AMD GPU, AMD's Rocprofiler-sdk reports the type of instruction that issued. HPCToolkit only considers an instruction of type T to be an exposed stall if instruction T should issue to pipeline P, and pipeline P did not issue any instruction in the current cycle.

Each of the GPUs report when a sampled instruction is ready to execute but was `NOT_SELECTED`. HPCToolkit doesn't generally consider `NOT_SELECTED` as an exposed stall. An instruction that is `NOT_SELECTED` stalled only because another instruction is executing instead. When the latency of a stalled instruction is overlapped with execution of another instruction, a developer need not be concerned about the reason for the stall. GPUs are designed to hide the latency of stalled instructions by interleaving the execution of instructions from different warps or wavefronts and overlapping one warp or wavefront's stall with execution of an instruction from another warp or wavefront.

Besides reporting a summary metric of exposed stalls, HPCToolkit additionally measures and attributes reasons for exposed stalls. HPCToolkit maps stall reasons from AMD, Intel, and NVIDIA GPUs into a (mostly) vendor-neutral taxonomy of stall reasons. To facilitate the interpretation of the myriad of instruction-level metrics that HPCToolkit collects using PC sampling, HPCToolkit presents metrics in the context of a top-down metric hierarchy. Section *9.5* describes HPCToolkit's support for top-down performance analysis on GPUs.

The next section describes how HPCToolkit attributes metrics to heterogeneous calling contexts that span CPU and GPU.

10.4.1 Attributing PC Samples to Source Code

Using PC sampling, HPCToolkit attributes GPU metrics to heterogeneous calling contexts that include the CPU calling context in which a GPU kernel was launched as a prefix and measurements within the GPU kernel as the suffix.

When collecting PC samples for a GPU-accelerated applications, GPU runtimes notify `hpcrun` every time they load a binary into a GPU. The first time a GPU binary is loaded, `hpcrun` computes a cryptographic hash of the GPU binary's contents and records the binary inside a `gpubins` subdirectory of `hpcrun`'s measurement directory in a file with a name based on the hex representation of its cryptographic hash, e.g., `984ef7dea.gpubin`.

To attribute PC samples collected during an execution of a GPU-accelerated application back to the application's code, one simply follows the traditional HPCToolkit workflow. Invoking `hpcstruct` on the execution's measurement directory analyzes any GPU binaries used in the course of an execution along with the executable and CPU shared libraries. Program structure information recovered by `hpcstruct` is used to map PC samples associated with machine instructions back to source code.

By default, when applied to a measurements directory, `hpcstruct` performs only lightweight analysis of the GPU functions in each GPU binary, enabling PC samples to be mapped to functions, loops, and source lines. It is important to understand that AMD, Intel, and NVIDIA GPU measurement APIs for PC sampling collect "flat" PC samples without any information about GPU call stacks.

In our experience, calling contexts within GPU kernels are essential for developers to understand the performance of sophisticated kernels that employ many device functions. The GPU-accelerated Quicksilver proxy application from Lawrence Livermore National Laboratory illustrates this problem. Figure *9.2* shows a screenshot of `hpcviewer` displaying PC sampling measurements for Quicksilver without reconstruction of a calling context tree within `CycleTrackingKernel`. The figure shows a top-down view of heterogeneous calling contexts that span both CPU and GPU. In the middle of the figure is a placeholder `<gpu kernel>` that is inserted by HPCToolkit to mark the transition between CPU and GPU code in the call stack. Above the placeholder is a CPU calling context where the GPU kernel was invoked. Below the `<gpu kernel>` placeholder, `hpcviewer` shows more than a dozen GPU device functions that were executed on behalf of `CycleTrackingKernel`.

Since GPU measurement APIs don't provide information about device-function call stacks within GPU kernels for PC samples, we designed a method to reconstruct GPU calling contexts during post-mortem analysis. This analysis is only performed when (1) an execution has been monitored using PC sampling, and (2) the measurement directory containing an execution's GPU binaries has been analyzed in detail by invoking `hpcstruct` on the directory with the option `--gpucfg yes`. Reconstructing calling contexts within GPU kernels is optional because it may be expensive for kernels with large calling context trees.

To reconstruct calling context trees for GPU computations, HPCToolkit uses information about call sites identified by `hpcstruct` in conjunction with PC samples measured for each `call` instruction in a GPU binary.

Apportioning costs among two or more calls to the same device function in a GPU calling context is a heuristic process. Without the ability to measure each device function invocation, HPCToolkit assumes that each invocation of a particular device function incurs the same costs. The costs of each device function are apportioned among its caller or callers using the following rules:

- If a device function G can only be invoked from a single call site, all of the measured cost of G will be attributed to its single call site.

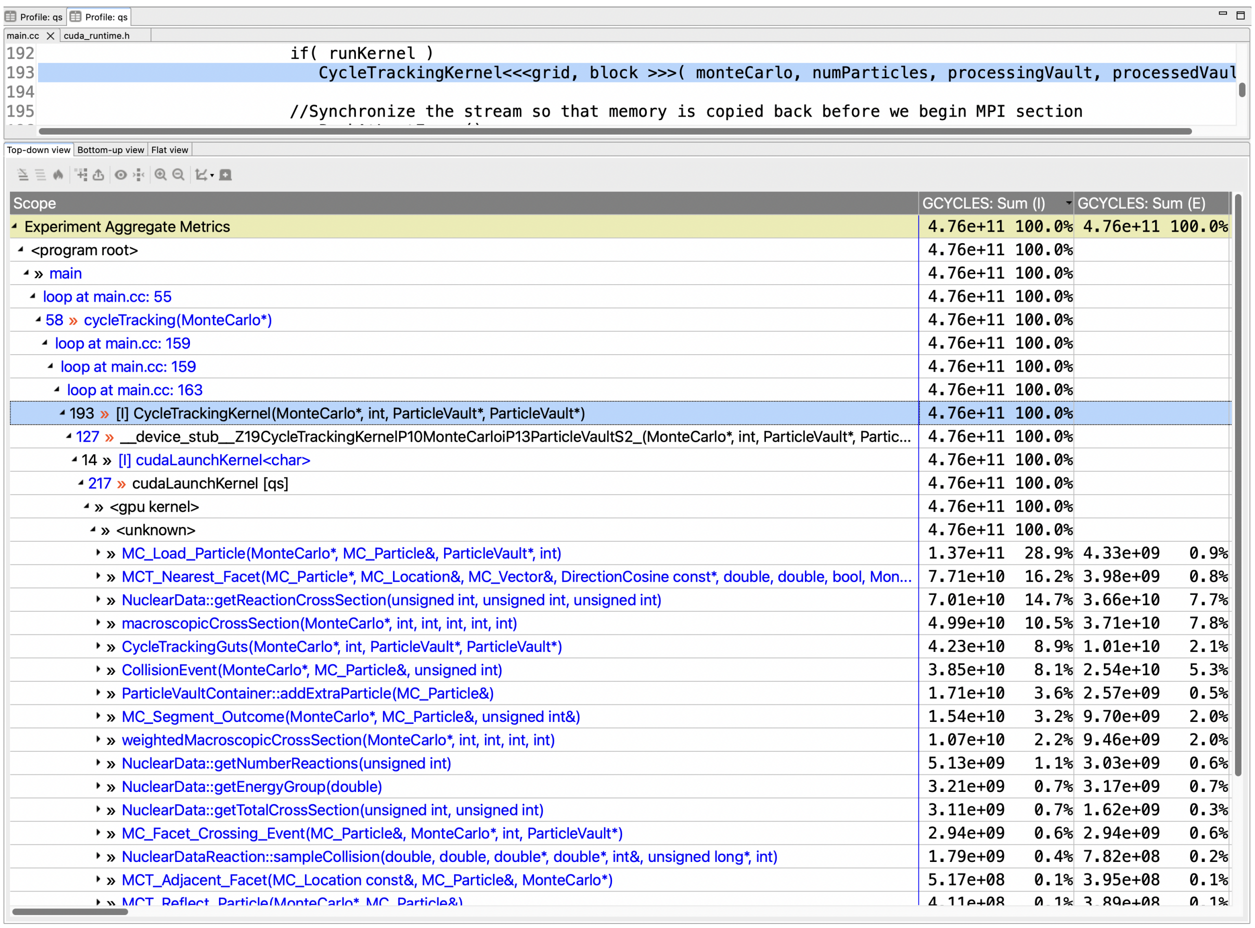


Fig. 2: Figure 9.2: A screenshot of `hpcviewer` for the GPU-accelerated Quicksilver proxy app without GPU CCT reconstruction.

- If a device function G can be called from multiple call sites and PC samples have been collected for one or more of the call instructions for G, the costs for G are proportionally divided among G's call sites according to the distribution of PC samples for calls that invoke G. For instance, consider the case where there are three call sites where G may be invoked, 5 samples are recorded for the first call instruction, 10 samples are recorded for the second call instruction, and no samples are recorded for the third. In this case, HPCToolkit divides the costs for G among the first two call sites, attributing 5/15 of G's costs to the first call site and 10/15 of G's costs to the second call site.
- If no call instructions for a device function G have been sampled, the costs of G are apportioned evenly among each of G's call sites.

HPCToolkit's `hpcprof` analyzes the static call graph associated with each GPU kernel. If the static call graph for a GPU kernel contains cycles, which arise from recursive or mutually-recursive calls, `hpcprof` replaces each cycle with a strongly connected component (SCC). In this case, `hpcprof` unlinks call graph edges between vertices within the SCC and adds an SCC vertex to enclose the set of vertices in each SCC. The rest of `hpcprof`'s analysis treats an SCC vertex in the call graph like a device function. The process of reconstructing calling context trees for GPU kernels using static call sites and sample counts is described elsewhere.

Figure *9.3* shows an `hpcviewer` screenshot for the GPU-accelerated Quicksilver code following reconstruction of GPU calling contexts. Notice that after the reconstruction, one can see that `CycleTrackingKernel` calls `CycleTrackingGuts`, which calls `CollisionEvent`, which eventually calls `macroscopicCrossSection` and `NuclearData::getNumberOfReactions`. The the rich GPU calling context tree reconstructed by `hpcprof` also shows loop nests and inlined code.

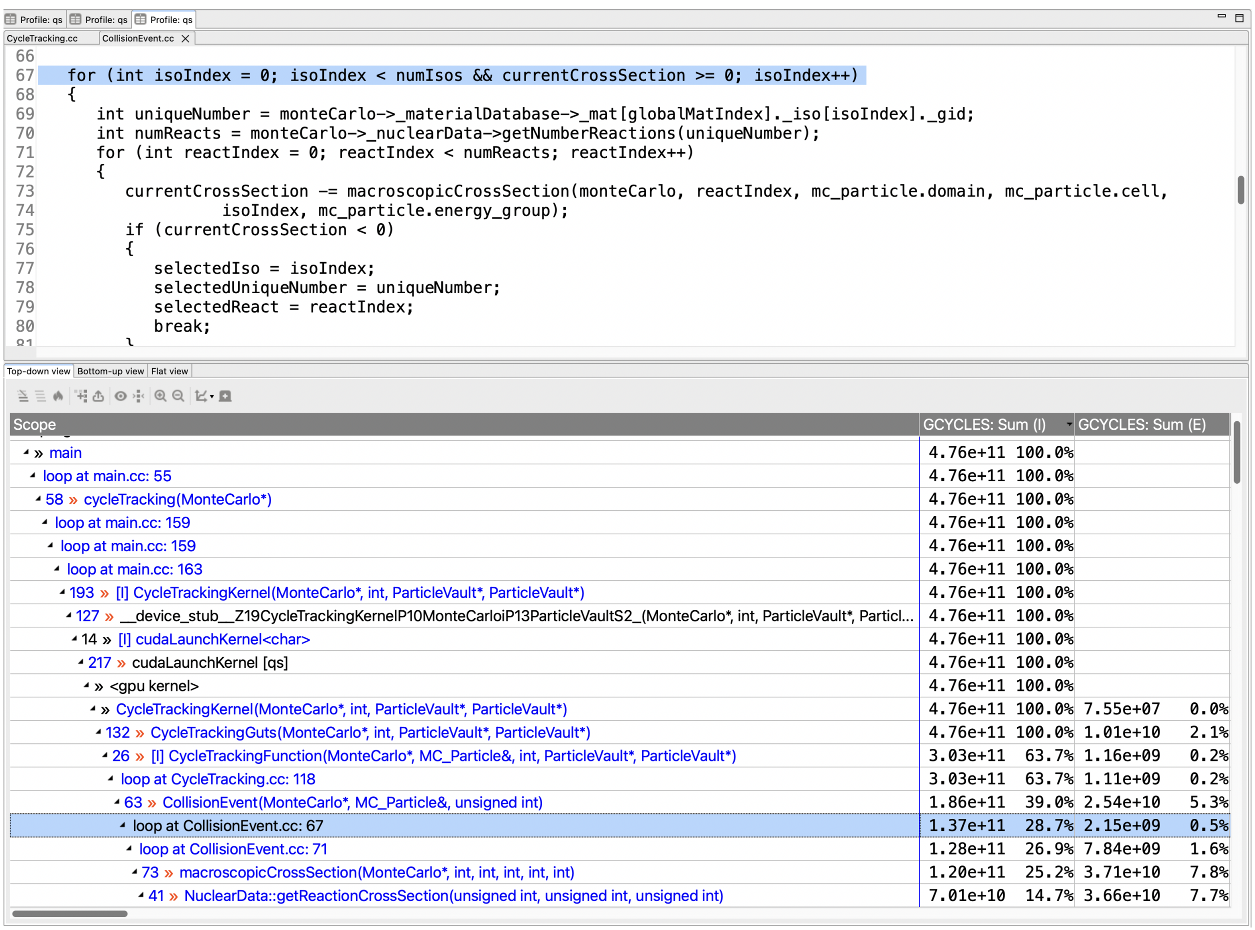


Fig. 3: Figure 9.3: A screenshot of `hpcviewer` for the GPU-accelerated Quicksilver proxy app with GPU CCT reconstruction.

Tip

If all PC samples in a kernel map to line 0, it is likely that the GPU binaries used by your application don't contain line mapping information. Consult the appropriate GPU-specific section below for information about the compiler option required to record line mappings that you need for performance analysis.

10.5 GPU Top-down Analysis

Inspired by Intel's top-down method for microarchitectural performance analysis on CPUs, HPCToolkit organizes instruction-level measurements of GPU kernels collected using PC sampling into a *top-down hierarchy* based on GPU instruction stall reasons. HPCToolkit constructs a top-down hierarchy automatically whenever a developer uses PC samples to measure an execution of a GPU-accelerated application.

HPCToolkit's top-down hierarchy classifies every GPU cycle estimated using PC sampling, enabling a developer to begin with a whole-execution profile view of the breakdown of how GPU cycles were spent. As we explain below, a developer can use HPCToolkit's support for top-down analysis of GPU execution costs to examine the breakdown of GPU cycles at any level of an execution's heterogeneous calling contexts, including the whole program, a callsite, a CPU function, a GPU kernel, a loop, an inlined function, or an individual source line in a GPU kernel.

Tip

Top-down analysis of GPU kernels reports the breakdown of cycles when a GPU is *active*. The breakdown of cycles in a kernel is only important if

- one or more GPUs are active for a significant fraction of a program's execution,
- the program spends a significant fraction of its execution time in the kernel, and
- the latency of the kernel's execution is *exposed*, i.e. not fully overlapped with the execution of code on a CPU.

If these conditions are not met, tuning efforts that increase the fraction of the program's execution in which the GPU is active are likely to be more effective. Traces are often helpful for gaining insight into what is happening on CPU cores when GPUs are idle.

10.5.1 Metrics Hierarchy

At the root of HPCToolkit's top-down hierarchy for GPU analysis is *GPU Cycles* – the total GPU cycles estimated using PC sampling. Each cycle is classified into one of three top-level categories:

- *Issues*: Cycles in which a sampled instruction issued. On AMD GPUs, hardware-based stochastic sampling (described in section Section *9.6.1.1*) reports the type of instruction issued (e.g., compute, memory, communication, control flow, or synchronization). In this case, HPCToolkit classifies instruction issues by instruction type.
- *Hidden Stalls*: Cycles in which a sampled instruction stalled but its latency was hidden because another instruction issued in the same cycle. Hidden stalls represent instructions that were ready to execute but were *not selected* in favor of another instruction. Hidden stalls are not evidence of a performance problem because GPUs are designed to hide latencies of stalled instructions by overlapping them with the execution of instructions from other warps or wavefronts.
- *Exposed Stalls*: Cycles in which a sampled instruction stalled and no other instruction issued to hide its latency. Exposed stalls represent lost pipeline cycles. These are subdivided by stall reasons to indicate their causes.

HPCToolkit's full top-down hierarchy for GPUs is shown in Figure *9.4*. On any particular GPU, HPCToolkit reports only the subset of the hierarchy that can be measured or synthesized.

```
GPU Cycles
├── Issues
│   ├── Compute
│   │   ├── Vector
│   │   │   ├── Matrix
│   │   │   ├── Dual
│   │   │   └── Single
│   │   └── Scalar
│   ├── Memory
│   │   ├── Flat
│   │   ├── LDS
│   │   ├── LDS Direct
│   │   └── Texture
│   ├── Communication
│   │   ├── Message
│   │   └── Export
│   ├── Control Flow
│   │   ├── Jump
│   │   └── Branch
│   │       ├── Taken
│   │       └── Not Taken
│   ├── Synchronization
│   └── Other
├── Exposed Stalls
│   ├── Front End Stalls
│   │   └── Instruction Fetch Stalls
│   └── Back End Stalls
│       ├── Compute Stalls
│       │   └── Instruction Dependence Stalls
│       ├── Memory Stall
│       │   ├── Latency Stalls
│       │   │   ├── Global Memory Stalls
│       │   │   ├── Constant Memory Stalls
│       │   │   └── Texture Memory Stalls
│       │   └── Request Issue Stalls
│       │       ├── Flat Request Issue Stalls
│       │       ├── Texture Request Issue Stalls
│       │       └── LDS Request Issue Stalls
│       ├── Resource Stalls
│       │   ├── Branch Unit Stalls
│       │   └── Pipeline Stalls
│       ├── Synchronization Stalls
│       └── Miscellaneous Stalls
│           ├── Internal Instruction Stalls
│           ├── Sleep Stalls
│           └── Other Stalls
└── Hidden Stalls
    ├── Not Selected
    └── Other Hidden
```

Figure 9.4 The top-down hierarchy of GPU metrics used by HPCToolkit.

The mapping of measurements onto the hierarchy is vendor specific because AMD, Intel, and NVIDIA GPUs report different information (e.g. stall reasons, pipeline status, activity mask) with each PC sample. Leaves of the hierarchy typically correspond to metrics measured using PC sampling, such as the issue and stall metrics described in Section 9.4. Sometimes, metrics at leaves are synthesized from measured metrics, as we describe below. Typically, the value of an interior node is computed as the sum of the values of its children. In other cases, it is measured or synthesized directly when no breakdown of its cost is available. Below, we describe how HPCToolkit computes the values of key top-down metrics for GPUs by different vendors.

- On NVIDIA and Intel GPUs, issued instructions are reported only in aggregate under *Issues*. In contrast, stochastic sampling on AMD GPUs reports the type of each issued instruction, so any issue can be classified and reported in a subcategory of compute, memory access, communication, or control flow.
- On AMD and Intel GPUs, memory latency stalls are reported only in aggregate under memory *Latency Stalls* because there is no information about the kind of access whose response is being awaited. In contrast, NVIDIA GPUs attribute a memory latency stall to global memory, constant memory, or texture memory.
- HPCToolkit attributes `SEND` stalls on Intel GPUs and `MEMORY THROTTLING` stalls on NVIDIA GPUs as *Request Issue Stalls*. However, AMD GPUs lack any stall reason that corresponds to *Request Issue Stalls*. HPCToolkit handles this by synthesizing *Request Issue Stalls* by integrating information about instruction types and stalls for various pipelines associated with memory accesses.
  - Instruction type information in a PC sample on an AMD GPU indicates whether a memory access instruction will access flat memory, texture memory, or local data store (LDS). The pipeline on which a memory access instruction will be dispatched is a function of the kind of memory it will access.
  - HPCToolkit reports a `PIPE BUSY` stall for a memory access instruction *A* as a subtype of *Request Issue Stalls* based on the kind of memory being accessed by *A*.
  - HPCToolkit also reports a `NOT SELECTED` memory access instruction *A* as a memory request issue stall if another memory access *B* attempted to issue to the pipeline associated with *A* but the pipeline stalled. In this case, the pipeline associated with *A* is stalled and the latency of *A*'s stall is not being hidden by *B*'s execution because *B* is stalled as well.

Important

AMD, Intel and NVIDIA GPUs each may report a stall reason of `NOT SELECTED` for a sampled instruction. This indicates that one or more other instructions were selected for execution instead. Since GPUs are designed to use multithreading to hide instruction stalls, instruction stalls that are overlapped with execution of other instructions are often not of interest. To help users avoid being needlessly concerned about `NOT SELECTED` stalls, HPCToolkit typically reports such stalls as *Hidden Stalls*.

It is important to note that HPCToolkit computes hidden stalls differently on Intel and NVIDIA GPUs than it does on AMD GPUs. On Intel and NVIDIA GPUs, HPCToolkit classifies a cycle as a hidden stall whenever the GPU reports a stall reason `NOT SELECTED` for the sampled instruction.

On AMD GPUs, stochastic PC samples include information about the state of individual GPU instruction pipelines (e.g., scalar, vector, flat memory). HPCToolkit uses this pipeline state information to assess whether an instruction that is not selected should be considered a hidden stall or not. Specifically, on an AMD GPU, HPCToolkit only considers a cycle as a hidden stall when two conditions are satisfied

- the sampled instruction with a stall reason `NOT SELECTED` maps to a particular pipeline `P` (e.g. vector), and
- another instruction associated with pipeline Pwas selected for execution.

Thus, on AMD GPUs HPCToolkit only considers a `NOT SELECTED` stall for a sampled instruction associated with pipeline `P` as hidden if another instruction issued on pipeline `P` in the same cycle.

The top-down hierarchy turns raw stall information reported by GPU hardware into actionable guidance. Two examples:

- A large fraction of cycles attributed to memory *Request Issue Stalls* often indicates that memory accesses are not being coalesced. Accesses are coalesced only when adjacent threads access adjacent memory locations.

To improve coalescing, ensure that `threadIdx.x` (CUDA or HIP), `get_local_id(0)` (OpenCL or SYCL), or `team_rank()` (Kokkos) is used as the innermost array index.

- If *Scalar* issues dominate *Vector* issues on an AMD GPU, wavefronts are spending more issue slots updating index variables, performing address arithmetic, evaluating predicates, and issuing uniform operations than performing vectorized computation.

10.5.2 Presentation

In `hpcviewer`, the top-down metric hierarchy showing the breakdown of *GPU Cycles* is visualized using a *donut chart*. When you open the profile view of a performance database containing instruction-level measurements of GPU cycles collected using hardware support for PC sampling, `hpcviewer` displays a donut chart that represents the breakdown of GPU cycles in the program execution along with a tree-based representation of the top-down metric hierarchy. Figure 9.5 shows an example of a *GPU Cycles* donut chart visualization for a sample profile.

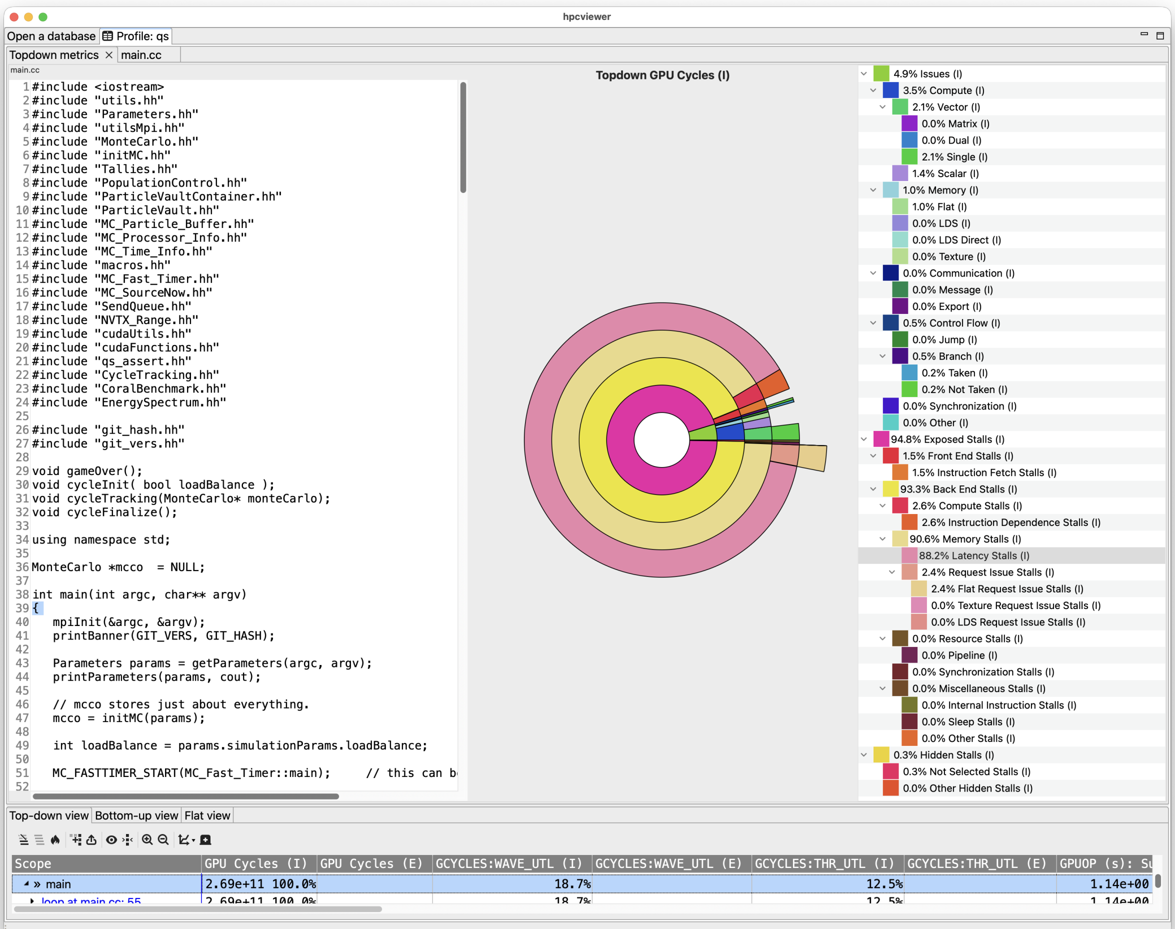


Fig. 4: Figure 9.5: An screenshot of `hpcviewer` showing donut-chart and tree-based representation of *GPU Cycles* for measurements of an application on an AMD MI300A.

As shown in Figure 9.5, the circle at the center of the initial donut chart represents the root *GPU Cycles* metric. The innermost ring of the donut chart represents the children of the metric at the graph's center. Each subsequent ring represents the children of the metrics in the ring it encloses. Child metrics in a ring occupy the same arc as their parent metric in the ring it encloses. In the tree-based representation of the metric hierarchy, each node in the tree contains a metric name and the percentage of the total GPU cycles associated with the metric.

Selecting any cell in an inclusive or exclusive *GPU Cycles* column of `hpcviewer`'s metric pane will refresh the top-down metrics pane to show the breakdown of *GPU cycles* for the scope represented by the line containing the selected cell.

Section *11.2.1.4* describes how to explore the data presented by the top-down metric hierarchy in `hpcviewer`'s top-down pane.

10.5.3 A Case Study

As a concrete example of the utility of top-down analysis, consider Minimod, a seismic modeling kernel that applies a star-shaped 25-point stencil to a 3D array. Figure *9.6* shows an annotated `hpcviewer` screenshot of top-down measurements of a HIP implementation of Minimod on an AMD MI300A (synthesized by Claude Sonnet) collected using stochastic PC sampling. HPCToolkit's top-down hierarchy attributes 91% of the GPU cycles spent in the stencil kernel to exposed stalls that occur while attempting to issue flat memory accesses. Such an overwhelming fraction of memory issue request stalls can be a sign that memory accesses are not being coalesced. Inspection of the kernel's indexing confirmed that values derived from `threadIdx.x` were not used to index the innermost dimension of the arrays, which causes uncoalesced accesses.

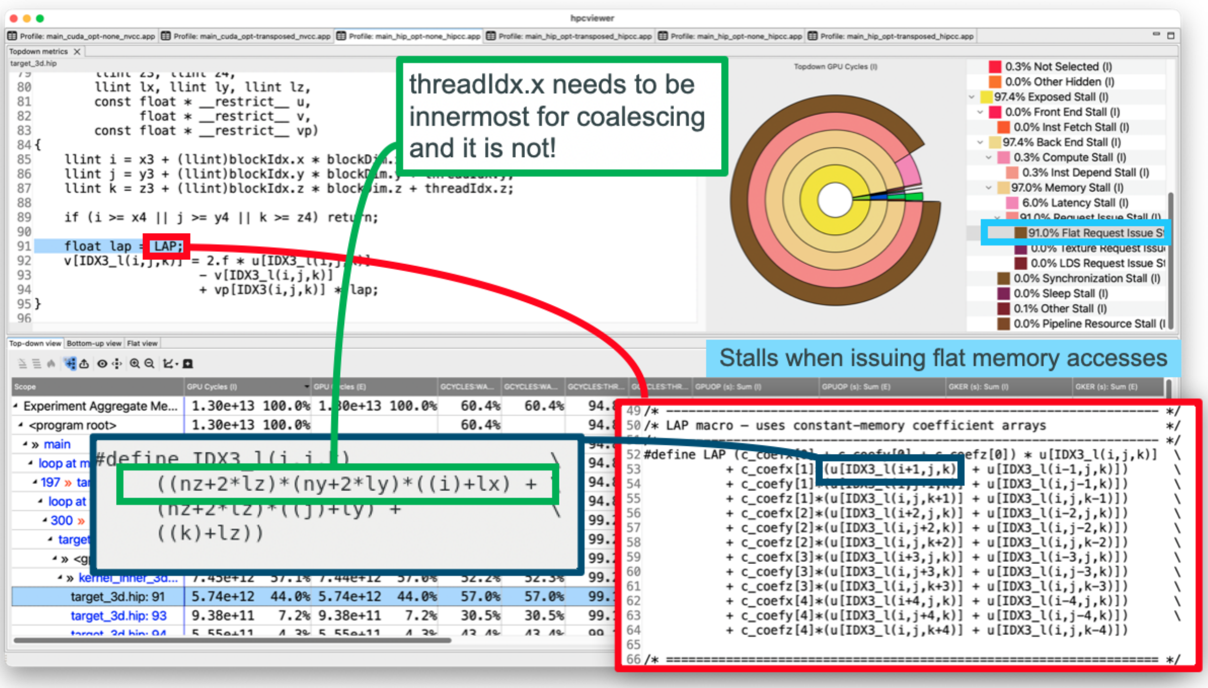


Fig. 5: Figure 9.6: An annotated screenshot of `hpcviewer` showing top-down measurements of Minimod's stencil kernel on an AMD MI300A. Most GPU cycles are exposed stalls when issuing flat memory accesses, a sign that memory accesses are not coalesced.

Figure *9.7* shows top-down measurements of a transposed variant of the kernel that indexes the innermost dimension with values derived from `threadIdx.x`. The transposed variant dramatically reduced memory request issue stalls and ran over 6x faster.

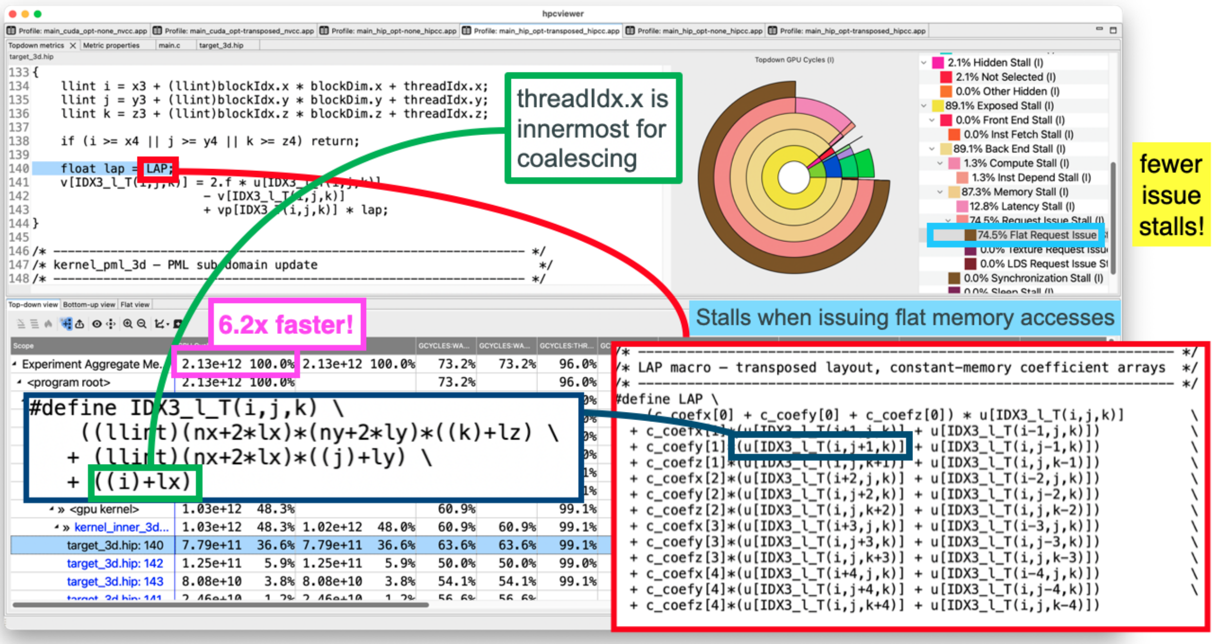


Fig. 6: Figure 9.7: An annotated screenshot of `hpcviewer` showing top-down measurements of a transposed variant of Minimod's stencil kernel on an AMD MI300A. With coalesced memory accesses, memory request issue stalls dropped dramatically and the kernel ran over 6x faster.

10.5.4 Inspecting Metrics

By default, `hpcviewer` only displays *GPU Cycles* - the root of the top-down metric hierarchy - as a column in its metric pane. If desired, columns for other metrics in the top-down hierarchy can be displayed.

Figure 9.8 shows a version of the top-down GPU metric hierarchy in which each metric is annotated with a name that encodes its position in the hierarchy. Displaying a metric column using the name that encodes its position in the hierarchy makes it easier to remember what a metric column represents.

```
GPU Cycles (GCYCLES)
├── Issues (GCYCLES:ISU)
│   ├── Compute (GCYCLES:ISU:COMP)
│   │   ├── Vector (GCYCLES:ISU:COMP:VECT)
│   │   │   ├── Matrix (GCYCLES:ISU:COMP:VECT:MATR)
│   │   │   ├── Dual (GCYCLES:ISU:COMP:VECT:DUAL)
│   │   │   └── Single (GCYCLES:ISU:COMP:VECT:SGL)
│   │   └── Scalar (GCYCLES:ISU:COMP:SCLR)
│   ├── Memory (GCYCLES:ISU:MEM)
│   │   ├── Flat (GCYCLES:ISU:MEM:FLAT)
│   │   ├── LDS (GCYCLES:ISU:MEM:LDS)
│   │   ├── LDS Direct (GCYCLES:ISU:MEM:LDSD)
│   │   └── Texture (GCYCLES:ISU:MEM:TEX)
│   ├── Communication (GCYCLES:ISU:COMM)
│   │   ├── Message (GCYCLES:ISU:COMM:MESG)
│   │   └── Export (GCYCLES:ISU:COMM:XPRT)
│   ├── Control Flow (GCYCLES:ISU:CFLOW)
│   │   ├── Jump (GCYCLES:ISU:CFLOW:JMP)
│   │   └── Branch (GCYCLES:ISU:CFLOW:BR)
│   │
```

(continues on next page)

(continued from previous page)

```
│   │       ├── Taken (GCYCLES:ISU:CFLOW:BR:TKN)
│   │       └── Not Taken (GCYCLES:ISU:CFLOW:BR:NTKN)
│   ├── Synchronization (GCYCLES:ISU:SYNC)
│   └── Other (GCYCLES:ISU:OTHR)
├── Exposed Stalls (GCYCLES:STL)
│   ├── Front End Stalls (GCYCLES:STL:FE)
│   │   └── Instruction Fetch Stalls (GCYCLES:STL:FE:IFET)
│   └── Back End Stalls (GCYCLES:STL:BE)
│       ├── Compute Stalls (GCYCLES:STL:BE:COMP)
│       │   └── Instruction Dependence Stall (GCYCLES:STL:BE:COMP:IDEP)
│       ├── Memory Stall (GCYCLES:STL:BE:MEM)
│       │   ├── Latency Stalls (GCYCLES:STL:BE:MEM:LAT)
│       │   │   ├── Global Memory Stalls (GCYCLES:STL:BE:MEM:LAT:GMEM)
│       │   │   ├── Constant Memory Stalls (GCYCLES:STL:BE:MEM:LAT:CMEM)
│       │   │   └── Texture Memory Stalls (GCYCLES:STL:BE:MEM:LAT:TMEM)
│       │   └── Request Issue Stalls (GCYCLES:STL:BE:MEM:REQ)
│       │       ├── Flat Request Issue Stalls (GCYCLES:STL:BE:MEM:REQ:FLAT)
│       │       ├── Texture Request Issue Stalls (GCYCLES:STL:BE:MEM:REQ:TEX)
│       │      └── LDS Request Issue Stalls (GCYCLES:STL:BE:MEM:REQ:LDS)
│       ├── Resource Stalls (GCYCLES:STL:BE:RES)
│       │   ├── Branch Unit Stalls (GCYCLES:STL:BE:RES:BUNIT)
│       │   └── Pipeline Stalls (GCYCLES:STL:BE:RES:PIPE)
│       ├── Synchronization Stalls (GCYCLES:STL:BE:SYNC)
│       └── Miscellaneous Stalls (GCYCLES:STL:BE:MISC)
│           ├── Internal Instruction Stalls (GCYCLES:STL:BE:MISC:INT)
│           ├── Sleep Stalls (GCYCLES:STL:BE:MISC:SLP)
│           └── Other Stalls (GCYCLES:STL:BE:MISC:OTHR)
└── Hidden Stalls (GCYCLES:HSTL)
    ├── Not Selected (GCYCLES:HSTL:NSEL)
    └── Other Hidden (GCYCLES:HSTL:OTHR)
```

Figure 9.8: The top-down hierarchy of GPU metrics annotated with names that encode their position in the hierarchy.

By clicking on `hpcviewer`'s metric display control icon one show any hierarchically-named metric in Figure *9.8* that is available in a database by adding a check mark next to the metric that you want to show.

In the metric pane, `hpcviewer` displays the measured value of a top-down metric for each scope along with the percentage that value represents relative to the total value of that metric measured in the execution. The percentage associated with a metric for a scope in the metric pane is incomparable to the percentage for that metric rendered for the scope in its top-down donut chart and tree representations. In the top-down donut chart and tree, the percentage shown for any of the scope's top-down metrics is relative to the aggregate GPU cycles measured for the program scope.

10.5.5 Under the Hood

For those interested in how the top-down hierarchy is implemented in an architecture-independent fashion, the tree is described by `gpu_topdown_info` – a vector of `mt_node_info_t` structures – defined in `src/hpcrun/gpu/gpu-metrics.c`. Each entry in the table indicates its parent (if any); the root *GPU Cycles* entry has no parent. Descriptions associated with stall reasons in the top-down hierarchy are defined in the `gpu_topdown_help` table also in `src/hpcrun/gpu/gpu-metrics.c`. The two tables are related by the type code in the `index` field of each structure. At runtime, when PC sampling is enabled, `hpcrun` links the raw metrics available on the GPU being measured into the corresponding nodes of the hierarchy, binds a derived metric to each interior node that sums the metrics of its children, and elides any subtree for which no measurements are available.

10.6 Vendor-specific Details

10.6.1 AMD GPUs

On AMD GPUs, HPCToolkit supports coarse-grain profiling of GPU-accelerated applications that offload GPU computation using AMD's HIP programming model, OpenMP, and OpenCL. In this section, we focus on a few important details about monitoring applications on AMD GPUs with AMD's new Rocprofiler-sdk.

AMD PC Sampling

AMD GPUs support two kinds of PC sampling: host-trap (software-based) sampling and stochastic (hardware-based) sampling. Support for host-trap based PC sampling first became available with ROCm 6.4 and is available on AMD GPUs MI200 and newer. Support for stochastic sampling first became available with ROCm 7.0 and it requires hardware support that first became available with AMD's MI300 series. Both kinds of sampling need the GPU to be running modern firmware.

You can check whether one or both kinds of PC sampling are available for your GPUs with AMD's `rocprofv3-avail` utility. Running `rocprofv3-avail info --pc-sampling` will report the list of PC sampling configurations supported for each GPU agent in your system.

Tip

If neither form of PC sampling is available, your GPU is an MI200+, and ROCm 6.4+ is installed, you might want to discuss with your system administrator whether PC sampling can be made available with a firmware update. Some information about what firmware versions are necessary or recommended for PC sampling on various GPU versions can be found in the Rocprofiler-sdk documentation. If you don't find what you need for your GPU, check with AMD by submitting a Github issue for AMD's documentation.

PC sampling on AMD GPUs is a device-wide activity for both host-trap (software-based) sampling and stochastic (hardware-based) sampling.

When using host-trap (software-based) PC sampling, a background thread periodically interrupts running waves on a GPU to read the program counter. Samples collected using host-trap sampling may suffer from "skid", where a sample may be attributed to an instruction up to two instructions away from a source of latency. As described in AMD's documentation about host-trap sampling, one must be careful interpreting host-trap PC samples because the costly instructions may be nearby rather than the ones where costs are attributed.

When using stochastic (hardware-based) PC sampling, each compute unit periodically chooses an active wave on the compute unit and records its PC. Over time, each compute unit samples waves in a round robin fashion.

To select software-based host-trap PC sampling, specify `-e gpu=rocm,pc=sw`. To select hardware-based stochastic PC sampling, specify `-e gpu=rocm,pc=hw`. Specifying simply `-e gpu=rocm,pc` will default to software-based host-trap sampling.

Periods for hardware stochastic sampling are measured in GPU cycles and the minimum is 256; periods must be a power of 2. HPCToolkit's default period for AMD stochastic sampling is currently set to `2^20` cycles. HPCToolkit enables a user to specify the desired period by adding an `@k` to the end of a PC sampling specification, e.g. `-e gpu=rocm,pc=hw@22` will set the period for stochastic sampling to `2^22` cycles. Periods for host-trap based sampling are measured in microseconds. The minimum period is 1 us, which HPCToolkit enforces as `2^10` ns. HPCToolkit's default period for AMD host-trap sampling is set to `2^20` ns. Using `-e gpu=rocm,pc=sw@21` will set the period for software-based host-trap sampling to `2^21` ns.

Important

At present, setting the sampling period for `hw` PC sampling to a period significantly shorter than `2^20` has been observed to cause AMD's driver to fail. As a result, HPCToolkit uses the AMD-recommended default PC sampling period of `2^20` cycles for stochastic sampling.

For both kinds of sampling, HPCToolkit reports GPU cycles (`GCYCLES`). The default frequency for AMD GPUs is approximately 1GHz, so 1 cycle is approximately 1 ns. While frequency adjustments may affect the GPU clock, assuming that the frequency is constant at 1GHz should be sufficient for measurements to be adequate for performance tuning.

AMD's stochastic sampling hardware records a wealth of information with each PC sample. A sampled instruction will either issue or not. In most cases, the stochastic sampling hardware in an AMD MI300 reports the kind of a wavefront's sampled instruction. Table *9.13* shows the kinds of issued instructions that the MI300 tracks and reports. In some cases, a wavefront's instruction may not be known if the wavefront is stalled fetching the next instruction after a branch.

Table 11: Table 9.13: GPU issue metrics. Issue metrics are hidden by default.

Metric	Description
GCYCLES:ISU:COMP:VECT:MATR	GPU issue cycles: issued a sampled matrix instruction
GCYCLES:ISU:COMP:VECT:DUAL	GPU issue cycles: issued dual vector instructions
GCYCLES:ISU:COMP:VECT:SGL	GPU issue cycles: issued a sampled vector instruction
GCYCLES:ISU:COMP:SCLR	GPU issue cycles: issued a sampled scalar instruction
GCYCLES:ISU:MEM:TEX	GPU issue cycles: issued a sampled texture instruction
GCYCLES:ISU:MEM:LDS	GPU issue cycles: issued a sampled Local Data Store instruction
GCYCLES:ISU:MEM:LDSD	GPU issue cycles: issued a sampled Local Data Store direct instruction
GCYCLES:ISU:MEM:FLAT	GPU issue cycles: issued a sampled flat instruction
GCYCLES:ISU:COMM:XPRT	GPU issue cycles: issued a sampled export instruction
GCYCLES:ISU:COMM:MESG	GPU issue cycles: issued a sampled message instruction
GCYCLES:ISU:SYNC:BAR	GPU issue cycles: issued a sampled barrier instruction
GCYCLES:ISU:CFLOW:BR:TKN	GPU issue cycles: issued a sampled taken branch instruction
GCYCLES:ISU:CFLOW:BR:NTKN	GPU issue cycles: issued a sampled not taken branch instruction
GCYCLES:ISU:CFLOW:JMP	GPU issue cycles: issued a sampled jump instruction
GCYCLES:ISU:OTHR	GPU issue cycles: issued a sampled 'other' instruction

AMD GPUs support multiple execution pipelines and multiple instruction issue. As a result, more than one pipeline may report an issue in a cycle. AMD's stochastic sampling hardware tracks all instructions that issue in a sampled cycle, not just the sampled instruction. Different models of AMD GPUs may have different pipelines. HPCToolkit reports AMD GPU pipeline issues with a separate metric for each GPU pipeline, as shown in Table *9.14*. Note that the matrix pipeline is only used for matrix instructions on low-precision numbers. Matrix operations on 32 or 64-bit numbers issue to the vector pipeline.

Table 12: Table 9.14: GPU pipeline issue metrics.

Metric	Description
GPIPE:ISU:MATR	GPU pipeline issue status: low precision matrix operation
GPIPE:ISU:DUAL	GPU pipeline issue status: dual vector dual
GPIPE:ISU:SGL	GPU pipeline issue status: single vector vector
GPIPE:ISU:SCLR	GPU pipeline issue status: scalar ALU or memory
GPIPE:ISU:LDS	GPU pipeline issue status: Local Data Store
GPIPE:ISU:LDSD	GPU pipeline issue status: Local Data Store direct
GPIPE:ISU:TEX	GPU pipeline issue status: texture
GPIPE:ISU:FLAT	GPU pipeline issue status: flat
GPIPE:ISU:XPRT	GPU pipeline issue status: export
GPIPE:ISU:BMSG	GPU pipeline issue status: branch or message
GPIPE:ISU:MISC	GPU pipeline issue status: miscellaneous

A GPU pipeline may stall waiting for resources that are not available in the current cycle, e.g., register ports. HPCToolkit reports AMD GPU pipeline stalls with a separate metric for each GPU pipeline, as shown in Table *9.15*. By comparing issue and stall counts with the GCYCLES metric, one can compute the fraction of cycles in which each pipeline issues, stalls, or is idle.

Table 13: Table 9.15: GPU pipeline stall metrics.

Metric	Description
GPIPE:STL:MATR	GPU pipeline stall status: low precision matrix operation
GPIPE:STL:DUAL	GPU pipeline stall status: dual vector
GPIPE:STL:SGL	GPU pipeline stall status: single vector
GPIPE:STL:SCLR	GPU pipeline stall status: scalar ALU or memory
GPIPE:STL:LDS	GPU pipeline stall status: Local Data Store
GPIPE:STL:LDSD	GPU pipeline stall status: Local Data Store direct
GPIPE:STL:TEX	GPU pipeline stall status: texture
GPIPE:STL:FLAT	GPU pipeline stall status: flat
GPIPE:STL:XPRT	GPU pipeline stall status: export
GPIPE:STL:BMSG	GPU pipeline stall status: branch or message
GPIPE:STL:MISC	GPU pipeline stall status: miscellaneous

Computation on AMD's GPUs is organized as wavefronts. AMD GPUs hide latency by overlapping stalls of one wavefront with the execution of another. Having multiple wavefronts available to schedule is important for hiding latency. On AMD GPUs, HPCToolkit measures how many wavefronts are active in each sampled cycle and computes the metrics shown in Table *9.16*. `GCYCLES:WAVE_ACT` reports the total number of wavefronts active aggregated over all sampled cycles. `GCYCLES:WAVE_AVL` reports the total number of wave slots (today 32) aggregated over all sampled cycles. From these two metrics, HPCToolkit computes `GCYCLES:WAVE_UTL` – the percent utilization of the available wave slots across all sampled cycles. Note that this measure of wave utilization only represents the utilization of wave slots in active CUs. You may have a high wave utilization even if you are only using a small fraction of the available compute units. Wavefront utilization metrics are attributed to code within GPU kernels at all levels of granularity: source lines, loops, inlined functions, and call chains.

Table 14: Table 9.16: GPU wave utilization metrics.

Metric	Description
GCYCLES:WAVE_ACT	GPU waves aggregate active
GCYCLES:WAVE_AVL	GPU waves aggregate available
GCYCLES:WAVE_UTL	GPU waves utilization (actual occupancy): 100 * (WAVE_ACT /WAVE_AVL)

Today, AMD's GPUs typically use wavefronts of 64 threads. Using hardware support for stochastic sampling on AMD MI300+ GPUs, HPCToolkit measures how many threads are active and computes the thread activity metrics shown in Table *9.17*. When a vector instruction is sampled, HPCToolkit determines how many threads are active in the sampled cycle by counting the number of bits in the execution mask. `GCYCLES:THR_ACT` reports the total number of bits in the execution mask summed over all sampled vector instructions. `GCYCLES:THR_AVL` reports the wavefront size x the number of sampled vector instructions. From these two metrics, HPCToolkit computes `GCYCLES:THR_UTL` – the percent utilization of the available SIMD lanes by vector instructions. Thread utilization metrics are attributed to code within GPU kernels at all levels of granularity: source lines, loops, inlined functions, and call chains.

Table 15: Table 9.17: GPU thread utilization metrics.

Metric	Description
GCYCLES:THR_ACT	GPU SIMD lanes (threads) aggregate active
GCYCLES:THR_AVL	GPU SIMD lanes (threads) aggregate available
GCYCLES:THR_UTL	GPU SIMD lanes (threads) utilization: 100 * (THR_ACT / THR_AVL)

Source Attribution

On AMD GPUs, adding a `-g` to your normal optimization flags when compiling with AMD's compilers, e.g. `hipcc`, will ensure that your GPU binaries include all of the information for mapping measurements collected with PC sampling to source code.

Note

After optimization by AMD's compilers, some code affected by optimization may map to line 0. This is a problem/feature of upstream LLVM and can't currently be avoided. If all code maps to line 0, that is a sign that you didn't compile your GPU code using `-g`.

HW Counters

AMD GPUs now support a variety of hardware counters. Using AMD's Rocprofiler-sdk, HPCToolkit can configure a GPU hardware counter to count the number of events that occur during a kernel execution. Multiple counters can be used in the same execution. To see the list of available GPU hardware counters, run the `hpcrun -L` command and scan for counters listed with the prefix `rocm::`. HPCToolkit exposes the set of counters exposed by AMD's Rocprofiler-sdk.

When using HPCToolkit on a system with multiple GPUs, any hardware counters specified on `hpcrun`'s command line will be configured for each of the GPUs specified in `ROCR_VISIBLE_DEVICES` or all GPUs if `ROCR_VISIBLE_DEVICES` is not specified.

If not all of a node's GPUs are not the same kind, they may support different sets of counters. If you encounter a situation where `hpcrun -L` indicates that certain counters are available but you have trouble using them to monitor an execution on multiple GPUs, you can use AMD's utility `rocprofv3-avail` to check whether a set of counters (e.g., `pmc1`, `pmc2`, `pmc3`) are available for particular GPU device and whether they can be measured together using `rocprofilerv3-avail pmc-check pmc1 pmc2 pmc3`.

Note that the ROCm counter names you use with HPCToolkit have the prefix `rocm::`; when you pass counter names to `rocprofilerv3-avail`, omit the `rocm::` prefix.

10.6.2 NVIDIA GPUs

HPCToolkit supports profiling, tracing, and PC sampling of GPU operations on NVIDIA GPUs for programs written in CUDA, OpenMP, OpenCL. Profiling, tracing, and PC sampling of GPU-accelerated applications have already been in this chapter. In this section, we focus on a few things specific to the measurement of applications on NVIDIA GPUs.

NVIDIA PC Sampling

NVIDIA's GPUs have supported PC sampling since Maxwell. Instruction samples are collected separately on each active streaming multiprocessor (SM) and merged in a buffer returned by NVIDIA's CUPTI. In each sampling period, one warp scheduler of each active SM samples the next instruction from one of its active warps. Sampling rotates through an SM's warp schedulers in a round robin fashion. When an instruction is sampled, its stall reason (if any) is recorded. If all warps on a scheduler are stalled when a sample is taken, the sample is marked as a latency sample, meaning no instruction will be issued by the warp scheduler in the next cycle. Figure *9.9* shows a PC sampling example on an SM with four warp schedulers. Among the six collected samples, four are latency samples, so the estimated stall ratio is 4/6.

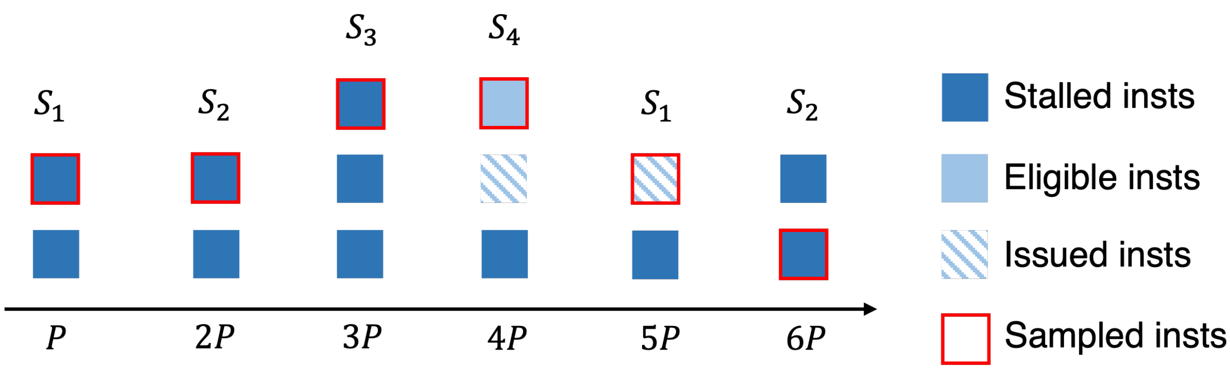


Fig. 7: Figure 9.9: NVIDIA's GPU PC sampling example on an SM. `P-6P` represent six sample periods P cycles apart. `S_1-S_4` represent four warp schedulers on an SM.

Table *9.18* shows summary statistics recorded by HPCToolkit with collecting PC samples on NVIDIA GPUs. Of particular note is the metric `GSAMP:UTIL`. HPCToolkit computes approximate GPU utilization using information gathered using PC sampling. Given the average clock frequency and the sampling rate, if all SMs are active, then HPCToolkit knows how many instruction samples would be expected (`GSAMP:EXP`) if the GPU was fully active for the interval when it was in use. HPCToolkit approximates the percentage of GPU utilization by comparing the measured samples with the expected samples using the following formula: `100 * (GSAMP:TOT) / (GSAMP:EXP)`.

Table 16: Table 9.18: GPU utilization statistics computed from PC Samples on NVIDIA GPUs.

Metric	Description
GSAMP:DRP	GPU PC samples: dropped
GSAMP:EXP	GPU PC samples: expected
GSAMP:TOT	GPU PC samples: measured
GSAMP:PER (cyc)	GPU PC samples: period (GPU cycles)
GSAMP:UTIL (%)	GPU utilization computed using PC sampling

Important

To collect PC samples on NVIDIA GPUs, at present HPCToolkit uses an older CUPTI interface that serializes the execution of GPU kernels. HPCToolkit uses this interface so that it can precisely attribute GPU PC samples to CPU calling contexts that invoke GPU kernels. This interface blocks concurrent execution of GPU kernels, which may slow execution. Furthermore, because of kernel serialization, HPCToolkit's PC sampling measurements will only reflect the GPU activity of kernels running in isolation.

Our experience with CUPTI's serialization-based API for PC sampling is that the overhead is less than `5x`. The overhead of GPU monitoring is principally on the host side. The time spent in GPU operations or in kernels as measured by PC samples is expected to be accurate. However, since execution as a whole is slowed while measuring GPU operations and collecting PC samples, one must be careful about drawing conclusions about overall program performance. Any traces collected while collecting PC samples on NVIDIA GPUs will be dilated and should be viewed as providing qualitative information only.

Source Attribution

To attribute performance within GPU kernels, one must add an appropriate flag to NVIDIA's compilers to record mappings between machine code and source code. `nvcc` and `nvc++` provide a `-lineinfo` option that records precise mappings from machine code to source code, including information about inlined functions.

Important

Don't use `nvcc`'s `-G` option when measuring performance. While it records precise mappings from machine code to source code, it also disables *all* compiler optimizations and generates GPU code that may be vastly slower.

Binary analysis

NVIDIA does not provide an API for analysis of GPU binaries. For that reason, HPCToolkit invokes NVIDIA's `nvdisasm` command line tool to analyze control flow in NVIDIA GPU binaries. `nvdisasm` produces Control Flow Graphs (CFGs) in DOT with assembly code annotations for each node in a CFG. For large CUDA binaries, `nvdisasm`'s ASCII output files can be huge; however, that is the least of the problems. The biggest problems are that (1) `nvdisasm` fails to produce a CFG for GPU functions that have certain features, e.g. contain `sib calls`, and (2) when a GPU binary contains a function that `nvdisasm` can't process, it gives up and doesn't process the remainder of a GPU binary.

To compensate, HPCToolkit inspects the symbol table of an NVIDIA GPU binary and then invokes `nvdisasm` once for each function in the GPU binary to extract that function's CFG. With this approach, HPCToolkit is resilient against `nvdisasm` failures on individual functions. In cases where `nvdisasm` fails to extract a CFG for a function, HPCToolkit attributes costs for any samples collected to the function as a whole.

Caution

For large binaries with many functions, the cost of invoking `nvdisasm` to extract a CFG for just one function is not proportional to only the size of the function of interest. That may make the aforementioned approach of using multiple executions of `nvdisasm` to extract CFGs one function at a time impractical for very large binaries. When we applied this approach to an NVIDIA GPU binary almost a gigabyte in size containing roughly 40K functions, `hpcstruct` analyzed the binary for two days without completing before we killed it.

If `hpcstruct` takes too long to analyze anything, it can be interrupted with `^C`. Any analysis already complete, will be saved and not recomputed. Restarting `hpcstruct` will show the first binary where analysis is being reinitiated. A problematic binary can be excluded from analysis by `hpcstruct` using `hpcstruct`'s `-x|--exclude` option. Without program structure information from `hpcstruct`, HPCToolkit's `hpcprof` will be unable to attribute costs to calling contexts within a GPU kernel, loops, or inlined code. However, `hpcprof` will still be able to attribute PC samples to source lines in kernels and device functions. Usage of the exclude option is discussed elsewhere in a tip in Section *3.1.3*.

10.6.3 Intel GPUs

HPCToolkit supports profiling, tracing, and PC sampling of GPU-accelerated applications that offload computation onto Intel GPUs using Intel's Data-parallel C++ programming model supported by Intel's `icpx` compiler, OpenMP computations offloaded with Intel's `ifx`, `icx`, or `icpx` compilers, or OpenCL. At program launch, a user can select whether Intel's Data-parallel C++ programming model is to execute atop Intel's OpenCL runtime or Intel's Level Zero runtime.

Collection of profiles, traces, and PC samples for GPU-accelerated applications was previously described in this chapter. On Intel GPUs, HPCToolkit also supports instrumentation-based measurement of GPU kernels using Intel's GTPin binary instrumentation tool in conjunction with Intel's Level Zero runtime.

At present, HPCToolkit only supports instrumentation-based collection of dynamic instruction counts within GPU kernels. Previously, HPCToolkit also supported approximate measurement of memory latency and a variety of statistics for SIMD instructions. However, problems experienced over the years with various versions of GTPin have led us to disable collection of latency and SIMD measurements in the current release of HPCToolkit. These capabilities may work with the most recent version of GTPin but they are untested. Instrumentation can be combined with profiling and tracing in the same execution.

Intel PC Sampling

Hardware support for PC sampling on Intel GPUs report stall reasons for sampled instructions. Table *9.19* shows the mapping of stall reasons reported by Intel's GPUs into HPCToolkit's top-down GPU metrics.

Since memory stalls tend to dominate everything else, we attribute all stalls that contain memory dependencies to memory stalls. On Intel GPUs, we report `SCOREBOARD ID` stalls to `GCYCLES:STL:BE:MEM:LAT` to indicate a stall waiting for unspecified outstanding memory requests, and `SEND` stalls to `GCYCLES:STL:BE:MEM:REQ` since SEND instructions are used to send messages to other components or write to device memory.

Table 17: Table 9.19: Intel GPU stall reasons, their descriptions, and their mappings into HPCToolkit's top-down GPU metric hierarchy.

Intel GPU Stall Reason	Stall Reason Description	Mapping to HPCToolkit top-down GPU metric
ACTIVE	Actively executing in at least one pipeline	GCYCLES:ISU
INST_FETCH	Stalled due to an instruction fetch operation	GCYCLES:STL:FE:IFET
DIST or ACC	Stalled due to internal pipeline dependency	GCYCLES:STL:FE:IDEPEND
SCOREBOARD ID	Stalled due to memory dependency or internal XVE pipeline dependency	GCYCLES:STL:BE:MEM:LAT
SEND	Stalled due to memory dependency or pipeline dependency for send	GCYCLES:STL:BE:MEM:REQ
SYNC	Stalled due to sync operation	GCYCLES:STL:BE:SYNC
CONTROL	Stalled because one or more threads is waiting for a branch unit	GCYCLES:STL:BE:RES:BR
PIPESTALL	Stalled due to XVE pipeline	GCYCLES:STL:BE:RES:PIPE
OTHER	Stalled due to any other reason	GCYCLES:STL:BE:MISC:OTHR

Source Attribution

To attribute performance within GPU kernels, one must add appropriate flags to Intel's compilers to record mappings between machine instructions and source code. Intel recommends using compiler options `-gline-tables-only` and `-fdebug-info-for-profiling` rather than `-g`. These options can be added to your normal optimization flags.

CHAPTER

ELEVEN

HPCVIEWER

11.1 Overview

HPCToolkit provides the `hpcviewer` (Adhianto, Mellor-Crummey, and Tallent 2010; Tallent et al. 2011) graphical user interface for interactive examination of performance databases. `hpcviewer` presents a heterogeneous calling context tree that spans both CPU and GPU contexts, annotated with measured or derived metrics to help users assess code performance and identify bottlenecks.

The database generated by `hpcprof` consists of 4 dimensions:

- *Execution context* (also called *execution profile*), which includes any logical threads (such as OpenMP, pthread, and C++ threads), MPI processes, and GPU streams.
- *Time*, which represents the timeline of the program's execution. This time dimension is only available if the application is profiled with traces enabled (`hpcrun -t` option).
- *Call-path*, which depicts a top-down path in a calling-context tree.
- *Metric*, which constitutes program measurements performed by `hpcrun` such as cycles, number of instructions, stall percentages, and derived metrics such as ratio of idleness.

To simplify performance data visualization, `hpcviewer` restricts the display of two dimensions at a time: the Profile View displays pairs of (call-path, metric) or (execution context, metric) dimensions and the Trace View visualizes the behavior of execution contexts over time. The table below summarizes views supported by `hpcviewer`.

View	Dimension	Note
Profile - *Metric view*	Call-path x Metrics	display the tree and its associated metrics
Profile - *Thread view*	Call-path x Metrics	display the tree and its metrics for a set of execution contexts
Profile - *Graph view*	Execution contexts x Metric	display a metric of a specific tree node for all execution contexts
Trace - *Main view*	Execution contexts x Time	display execution context behavior over time
Trace - *Depth view*	Call-path x Time	display call stacks over time of an execution context

Note that in the *Profile view*, GPU stream execution contexts are not shown in this view; metrics for a GPU operation are associated with the calling context in the thread that initiated the GPU operation (Section *Thread View*). In the *Trace view*, GPU streams have their trace lines independently from their host, allowing for the traces between hosts and devices to be separated.

11.1.1 Downloading

While you can build your own copy of HPCToolkit's `hpcviewer` graphical user interface, we recommend downloading a binary release at https://hpctoolkit.org/download.html. Pre-built packages are available for Linux (x86_64, aarch64, ppcle), MacOS (aarch64, x86_64), and Windows (x86_64).

> **Note**
>
> Trace views previously provided by HPCToolkit's `hpctraceviewer` interface have been integrated into `hpcviewer` since its 2020.12 release.

11.1.2 Building from Source

Source code for HPCToolkit's `hpcviewer` graphical user interface is available at https://gitlab.com/hpctoolkit/hpcviewer.git. Clone a copy with `git`. Building `hpcviewer` requires `maven`. You can download a copy of `maven` with Spack. To build `hpcviewer` on your platform of choice, follow the directions in its `README.md` on Gitlab.

11.1.3 Launching

Requirements to launch `hpcviewer`:

- On all platforms: Java 17 or newer (up to Java 21).
- On Linux: GTK 3.20 or newer.

`hpcviewer` can be launched from a command line (Linux platforms) or by clicking the `hpcviewer` icon (for Windows, Mac OS X, and Linux platforms). The command line syntax is as follows:

```
hpcviewer [options] [<hpctoolkit-database>]
```

Here, `<hpctoolkit-database>` is an optional argument to load a database automatically. Without this argument, `hpcviewer` will prompt for the location of a database. Possible options for `hpcviewer` are shown below:

`-h, --help`
Print a help message.

`-jh, --java-heap` *size*
Set the JVM maximum heap size for this execution of `hpcviewer`. The value of *size* must be in megabytes (M) or gigabytes (G). For example, one can specify a *size* of 3 gigabytes as either 3076M or 3G.

`-v, --version`
Print the current version

On Linux, when `hpcviewer` is installed using its `install.sh` script, one can specify the maximum size for the Java heap on the current platform. When analyzing measurements for large and complex applications, it may be necessary to use the `--java-heap` option to specify a larger heap size for `hpcviewer` to accommodate many metrics for many contexts.

On macOs and Windows, the value of JVM maximum heap size is stored in `hpcviewer.ini` file, specified with `-Xmx` option. On macOS, this file is located at `hpcviewer.app/Contents/Eclipse/hpcviewer.ini`.

11.1.4 Menus

`hpcviewer` provides four main menus: **File**, **Filter**, **View**, and **Help**.

File

This menu includes several menu items for controlling basic viewer operations.

- **New window**
  Open a new `hpcviewer` window that is independent of the existing one. However, filtering CCT node operation (see *Filtering Tree Nodes*) will also affect all `hpcviewer` windows.
- **Open database**
  Open a database without replacing the existing one. This menu can be used to compare two databases.
- **Open remote database** *(experimental feature)*
  Open a database located at the remote host via a secured SSH tunnel. One can use the private key option to connect without typing a password. See *Remote database* page for further details.
- **Switch database**
  Load a performance database into the current `hpcviewer` window replacing the existing opened databases.
- **Close database**
  Unloading an open database.
- **Merge databases**
  Merging two databases that are currently in the viewer. Currently, it doesn't support storing a merged database in a file.
  – **Top-down tree**: Merging the top-down tree of the databases.
  – **Flat tree**: Merging the flat (static) tree of the databases.
- **Preferences**
  Display the settings window, which consists of three sections:
  – **Appearance**: Change the fonts for tree and metric columns and source viewer.
  – **Traces**: Specify settings for Trace view, such as the rendering option, the number of working threads to be used and the tooltip's delay.
  – **Debug**: Enable/disable debug mode and experimental feature.
- **Exit**
  Quit the `hpcviewer` application.

Filter

- **Filter CCT nodes**
  Open a filter property window that enables one to filter which calling context tree nodes will be shown or hidden. Why node filtering is useful and how to filter tree nodes is discussed in Section *Filtering Tree Nodes*.
- **Filter execution contexts** (Trace view only)
  Open a window to select which trace lines to show or hide.

View

- **Zoom in**
  Zoom in increases the font size and line spacing used in all views.
- **Zoom out**
  Zoom out decreases the font size and line spacing used in all views.
- **Show metrics** *(Profile view only)*
  Display the list of metrics including their name, description, and visibility. For GPU metrics, the descriptions are useful for explaining what the short and somewhat cryptic metric names mean. From this list, one can use

the `Edit` button to rename or modify a derived metric, including the formula. Once the change is confirmed, it will be propagated to the list and all views (Top-down, Bottom-up, and Flat) in the current database.

- **Show color map** *(Trace view only)*
  Open a window showing the customized mapping between a procedure pattern and a color. See the *Color map* Section.
- **Debug** *(if debug mode is enabled)*
  - **View experiment.xml file** *(For databases created with HPCToolkit 2021.05.15 and earlier)*
    Display HPCToolkit's XML representation of performance data.

Help *(Linux only)*

- **About**
  Displays brief information about `hpcviewer` including version information and funding acknowledgments.

11.1.5 Limitations

Some important `hpcviewer` limitations are listed below:

- **Limited number of metric columns**.
  With a large number of metric columns, `hpcviewer`'s response time may become sluggish as this requires a large amount of memory.
- **Experimental Windows 11 platform**.
  The Windows version of `hpcviewer` is mainly tested on Windows 10. Support for Windows 11 is still experimental.
- **No support for dark themes**.
  We received reports that `hpcviewer` is not very visible on Linux with a dark theme. It is still an ongoing work.
- **Linux TWM window manager is not supported**.
  Reason: This window manager is too ancient.

11.2 Profile View

Profile view is the default view, and it interactively presents context-sensitive performance metrics correlated to program structure and mapped to a program's source code, if available. It can show an arbitrary collection of performance metrics gathered during one or more runs.

Profile View: An annotated screenshot of hpcviewer's interface

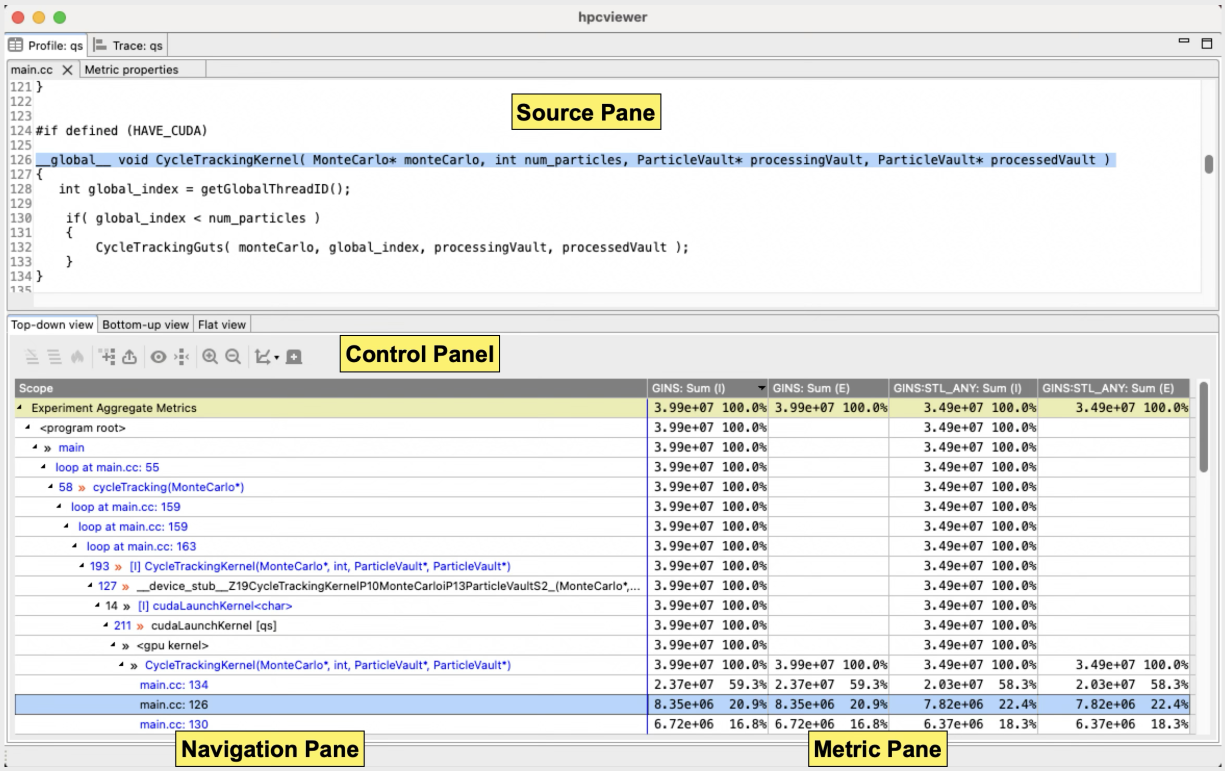


Figure *Profile View* above shows an annotated screenshot of `hpcviewer`'s user interface presenting a call path profile. The annotations highlight `hpcviewer`'s principal window panes and key controls.

The Profile view is conceptually divided into two principal panes. The top pane is primarily to display source code associated with performance metrics, display metric properties, and plot graphs. The bottom pane contains a control panel, a navigation pane, and a metric pane, which associates performance metrics with code contexts displayed in the navigation pane. These panes are discussed in more detail in Section *Panes*.

`hpcviewer` displays calling-context-sensitive performance data in three views: *Top-down*, *Bottom-up*, and *Flat View*. One selects the desired view by clicking on the corresponding view control tab. We briefly describe the three views and their corresponding purposes.

- **Top-down View**. This top-down view shows the dynamic calling contexts (call paths) in which costs were incurred. Using this view, one can explore performance measurements of an application in a top-down fashion to understand the costs incurred by calls to a procedure in a particular calling context. We use the term *cost* rather than simply *time* since `hpcviewer` can present a multiplicity of metrics, such as cycles, cache misses, or derived metrics (e.g., cache miss rates or bandwidth consumed) that are other indicators of execution cost.

  A calling context for a procedure `f` consists of the stack of procedure frames active when the call was made to `f`. Using this view, one can readily see how much of the application's cost was incurred by `f` when called from a particular calling context. If finer detail is of interest, one can explore how the costs incurred by a call to `f` in a particular context are divided between `f` itself and the procedures it calls. HPCToolkit's call path profiler `hpcrun` and the `hpcviewer` user interface distinguish calling context precisely by individual call sites; this means that if a procedure `g` contains calls to procedure `f` in different places, these represent separate calling contexts.

- **Bottom-up View**. This bottom-up view enables one to look upward along call paths. The view apportions a procedure's costs to its callers and, more generally, its calling contexts. This view is particularly useful for understanding the performance of software components or procedures used in multiple contexts. For instance, a message-passing program may call `MPI_Wait` in many different calling contexts. The cost of any particular call will depend upon the structure of the parallelization in which the call is made. Serialization or load imbalance may cause long waits in some calling contexts, while other parts of the program may have short waits because computation is balanced and communication is overlapped with computation.

When several levels of the Bottom-up View are expanded, saying that it apportions metrics of a callee on behalf of its callers can be confusing. More precisely, the Bottom-up View apportions the metrics of a procedure on behalf of the various *calling contexts* that reach it.

- **Flat View**. This view organizes performance measurement data according to the static structure of an application. All costs incurred in any calling context by a procedure are aggregated in the Flat View. This complements the Top-down View, in which the costs incurred by a particular procedure are represented separately for each call to the procedure from a different calling context.

11.2.1 Panes

`hpcviewer`'s browser window is divided into three panes: the *Source pane*, *Navigation pane*, and the *Metrics pane*. We briefly describe the role of each pane.

Source Pane

The source pane displays the source code associated with the current entity selected in the navigation pane. When a performance database is first opened with `hpcviewer`, the source pane is initially blank because no entity has been selected in the navigation pane. Selecting any entity in the navigation pane will cause the source pane to load the corresponding file and highlight the line corresponding to the selection. Switching the source pane to display a different source file is accomplished by making another selection in the navigation pane.

Navigation Pane

The navigation pane presents a hierarchical tree-based structure used to organize the presentation of an applications's performance data. Entities in the navigation pane's tree include load modules, files, procedures, procedure activations, inlined code, loops, and source lines. Selecting any of these entities will cause its corresponding source code (if any) to be displayed in the source pane. In this view, one can reveal or conceal children in this hierarchy by 'opening' or 'closing' any non-leaf (i.e., individual source line) entry.

The node in the tree will have the color blue if the source code is available. Otherwise, its color is black. If a node is a call statement, it will have the line number of the call statement (if the information is available) and the callee procedure as follows:

> [`line_number`] » `callee`

The symbol » represents a call to a procedure. For instance, the following node means that at line `1234` there is a call to function `foo()`:

> `1234` » `foo()`

Clicking the line number will display the source code at the call site of the function. Clicking the name of the called function will display the function's first line.

The nature of the entities in the navigation pane's tree structure depends upon whether one is exploring the Top-down View, the Bottom-up View, or the Flat View of the performance data.

- In the **Top-down View**, entities in the navigation tree represent procedure activations, inlined code, loops, and source lines. While most entities link to a single location in source code, procedure activations link to two: the call site from which a procedure was called and the procedure itself.
- In the **Bottom-up View**, entities in the navigation tree are procedure activations. Unlike procedure activations in the top-down view, in which call sites are paired with the called procedure, in the bottom-up view, call sites are paired with the calling procedure to allow to attribute the costs of a called procedure to multiple different call sites and callers. The node in this view has the following notation:

`[line_number] « caller`

Where the symbol « represents a call by a procedure.

- In the **Flat View**, entities in the navigation tree correspond to source files, procedure call sites (rendered the same way as procedure activations), loops, and source lines.

Control Panel

The header above the navigation pane contains some controls for the navigation and metric view. In Figure *Profile View*, they are labeled as "navigation control."

- **Flatten / Unflatten** (Flat View only): Enabling to flatten and unflatten the navigation hierarchy. Clicking on the flatten button (the icon that shows a tree node with a slash through it) will replace each top-level scope shown with its children. If a scope has no children (i.e., it is a leaf), the node will remain in the view. This flattening operation is useful for relaxing the strict hierarchical view so that peers at the same level in the tree can be viewed and ranked together. For instance, this can be used to hide procedures in the Flat View so that outer loops can be ranked and compared to one another. The inverse of the flatten operation is the unflatten operation, which causes an elided node in the tree to be made visible once again.
- **Zoom-in / Zoom-out** : Depressing the up arrow button will zoom in to show only information for the selected line and its descendants. One can zoom out (reversing a prior zoom operation) by depressing the down arrow button.
- **Hot call path** : This button automatically reveals and traverses the hot call path rooted at the selected node in the navigation pane with respect to the selected metric column. Let `n` be the node initially selected in the navigation pane. A hot path from `n` is traversed by comparing the values of the selected metric for `n` and its children. If one child accounts for `T%` or more (where `T` is the threshold value for a hot call path) of the cost at `n`, then that child becomes `n` and the process repeats recursively.
- **Add derived metric** : Create a new metric by specifying a mathematical formula. See Section *Derived Metrics* for more details.
- **Hide/show metrics** : Show or hide metric columns. A metric property view will appear, and the user can select which metric columns should be displayed.
- **Resizing metric columns** / : Resize the metric columns based on either the width of the data or the width of both of the data and the column's label.
- **Export into a CSV format file** : Export the current metric table into a comma separated value (CSV) format file. This feature only exports all metrics that are currently shown. Metrics not shown in the view (whose scopes are not expanded) will not be exported (we assume these metrics are not significant).
- **Increase font size / Decrease font size** : Increase or decrease the size of the navigation and metric panes.
- **Show a graph of metric values** (Top-down view only): Show a graph (a plot, a sorted plot, or a histogram) of metric values associated with the selected node in CCT for all processes or threads (Section *Plot Graphs*).
- **Show the metrics of a set of threads** (Top-down view only): Show the CCT and the metrics of a selected threads (Section *Thread View*).

Context menus

Navigation control also provides several context menus by clicking the right-button of the mouse.

- **Copy**: Copy into the clipboard the selected line in the navigation pane, which includes the name of the node in the tree and the values of visible metrics in the *metric pane*. The values of hidden metrics will not be copied.

- **Find**: Search for text within the Scope column of the current table. The search has several options, such as case sensitivity, whole word search, and using regular expressions.

Metric Pane

The metric pane displays one or more performance metrics associated with entities to the left in the navigation pane. Entities in the tree view of the navigation pane are sorted at each level of the hierarchy by the metric in the selected column. When `hpcviewer` is launched, the leftmost metric column is the default selection, and the navigation pane is sorted according to the values of that metric in descending order. One can change the selected metric by clicking on a column header. Clicking on the header of the selected column toggles the sort order between descending and ascending.

During analysis, one often wants to consider the relationship between two metrics. This is easier when the metrics of interest are in adjacent columns of the metric pane. One can change the order of columns in the metric pane by selecting the column header for a metric and then dragging it left or right to its desired position. The metric pane also includes scroll bars for horizontal scrolling (to reveal other metrics) and vertical scrolling (to reveal other scopes). Vertical scrolling of the metric and navigation panes is synchronized.

Top-down Pane

As described elsewhere in this manual, HPCToolkit builds top-down metric hierarchies to present an integrated view of CPU performance assembled from measurements collected using many hardware counters (Section *7.6*), and and GPU performance measurements collected using hardware support for PC sampling on AMD, Intel, and NVIDIA GPUs (Section *10.5*).

Whether top-down metrics represent CPU performance or GPU performance, `hpcviewer`'s profile view presents them in a *Top-down metrics* pane using the same visualizations. These include

- a tree-based representation of the metric hierarchy,
- a donut chart that visualizes distribution of costs among each node of the metric hierarchy, and
- an embedded source code pane that shows the source code context associated with the measurement data shown in the metric hierarchy and donut chart.

Figure *topdown* shows an example of the top-down metrics pane for GPU performance measured on an AMD MI300A.

As shown in Figure *topdown*, the circle at the center of the initial donut chart represents the root *GPU Cycles* metric. The innermost ring of the donut chart represents the children of the metric at the graph's center. Each subsequent ring represents the children of the metrics in the ring it encloses. Child metrics in a ring occupy the same arc as their parent metric in the ring it encloses. In the tree-based representation of the metric hierarchy, each node in the tree contains a metric name and the percentage of the total GPU cycles associated with the metric.

Hovering over a ring segment in the donut chart will highlight the corresponding node in the tree-based representation of the metric hierarchy. After a few seconds, hovering over a ring segment of a donut chart or a node in the tree-based representation representing a metric will bring up a tooltip with information about the metric. For many of the metrics that represent stalls, a metric's tooltip often includes suggestions for how you might change your program to reduce that kind of stalls. Clicking on a segment of a ring in the donut chart, which represents a metric at some level in the hierarchy, will zoom the donut chart so that the selected metric becomes the center of the donut and its children occupy the innermost ring. Clicking on the center of the donut chart will zoom out one level so the node's parent in the hierarchy, if any (the root metric has no parent), becomes the center of the donut and its child metrics are then shown in the innermost ring.

Showing the Top-down Metrics Pane

For a GPU-accelerated application measured using hardware-based GPU PC sampling, selecting a cell in the *GPU Cycles* column of `hpcviewer`'s metric pane will refresh the donut chart and the tree-based representation of the metric hierarchy to display the breakdown of *GPU cycles* for the scope represented by the line containing the selected cell.

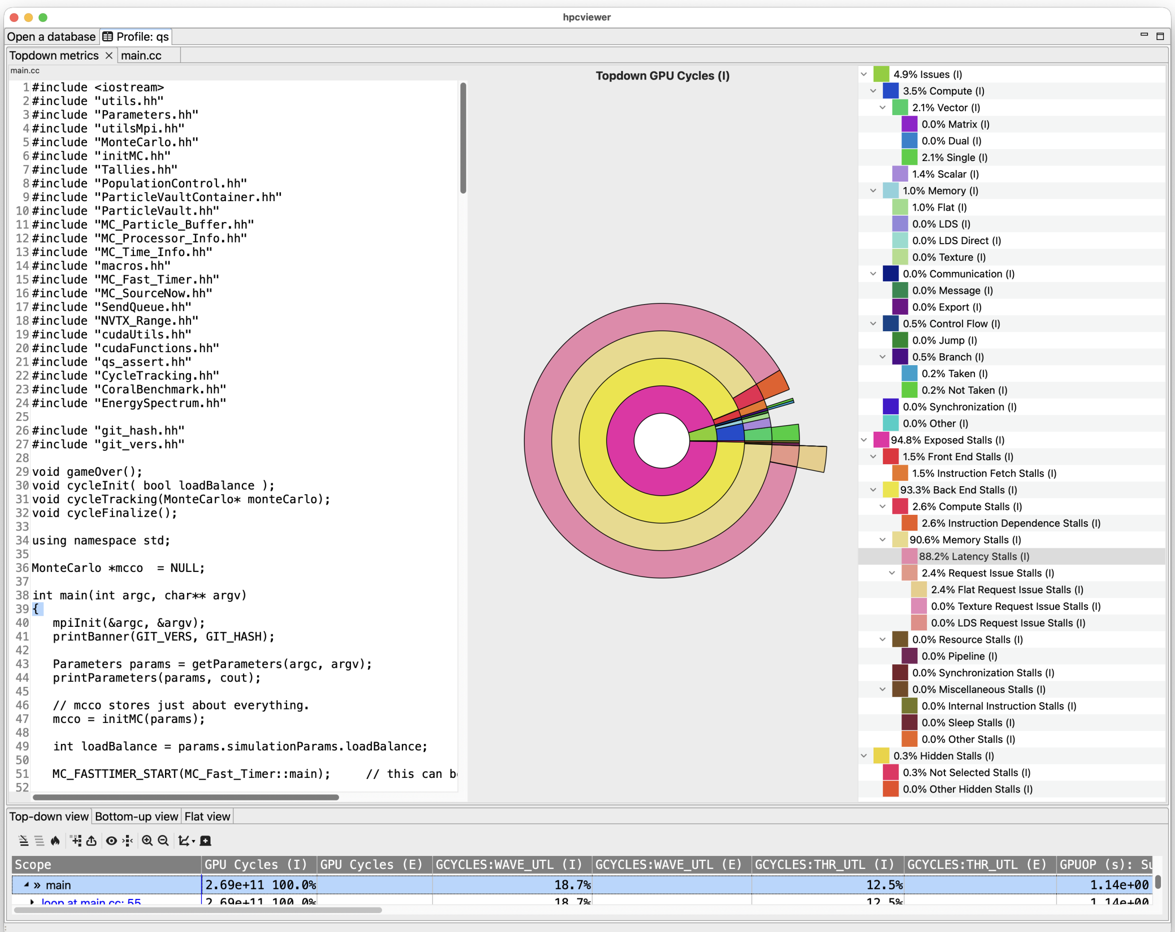


Fig. 1: Figure topdown: An screenshot of `hpcviewer` showing donut-chart and tree-based representation of *GPU Cycles* for measurements of an GPU-accelerated application on an AMD MI300A.

On an Intel Sapphire Rapids CPU (currently the only CPU for which HPCToolkit supports top-down analysis), measuring a CPU application with any top-down option (-topdown, -topdown:retiring, -topdown:frontend, -topdown:backend, or -topdown:all) enables the *top-down metrics view* in `hpcviewer`. Selecting a cell in the *Total Slots* column of the metric pane refreshes both the donut chart and the tree-based metric hierarchy to show how total slots — a measure of the CPU's total issue slots during execution — breaks down for the scope in that row.

11.2.2 Understanding Metrics

`hpcviewer` can present an arbitrary collection of performance metrics gathered during one or more runs, or compute derived metrics expressed as formulae. A derived metric may be specified with a formula that typically uses one or more existing metrics as terms in an expression.

For any given scope in `hpcviewer`'s three views, `hpcviewer` computes both *inclusive* and *exclusive* metric values. First, consider the Top-down View. Inclusive metrics reflect costs for the entire subtree rooted at that scope. Exclusive metrics are of two flavors, depending on the scope. For a procedure, exclusive metrics reflect all costs within that procedure but exclude callees. In other words, for a procedure, costs are exclusive with respect to dynamic call chains. For all other scopes, exclusive metrics reflect costs for the scope itself; i.e., costs are exclusive with respect to a static structure. The Bottom-up and Flat Views contain inclusive and exclusive metric values relative to the Top-down View. This means, e.g., that inclusive metrics for a particular scope in the Bottom-up or Flat View are with respect to that scope's subtree in the Top-down View.

How Metrics are Computed

Call path profile measurements collected by `hpcrun` correspond directly to the Top-down View. `hpcviewer` derives all other views from exclusive metric costs in the Top-down View. For the Bottom-up View, `hpcviewer` collects the cost of all samples in each function and attribute that to a top-level entry in the Bottom-up View. Under each top-level function, `hpcviewer` can look up the call chain at all of the contexts in which the function is called. For each function, `hpcviewer` apportions its costs among each of the calling contexts in which they were incurred. `hpcviewer` computes the Flat View by traversing the calling context tree and attributing all costs for a scope to the scope within its static source code structure. The Flat View presents a hierarchy of nested scopes for load modules, files, procedures, loops, inlined code, and statements.

Example

Assume one has a simple recursive program divided into two source files `file1.c` and `file2.c` as follows:

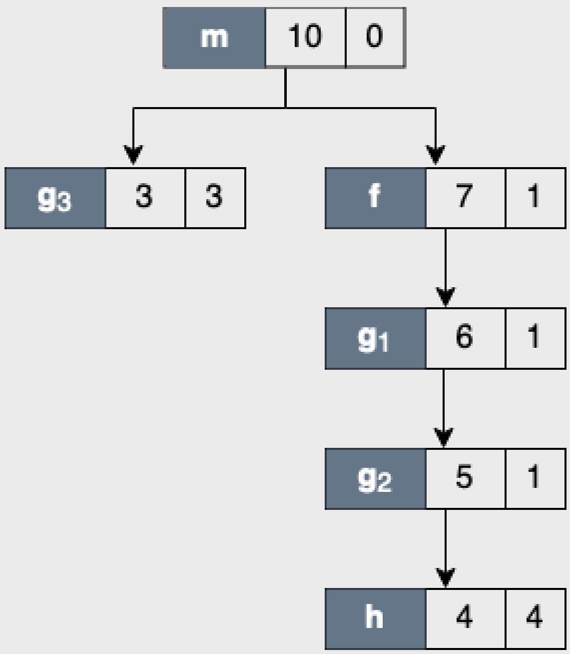
```
// file1.c
f () {
  g ();
}
// m is the main routine
m () {
  f ();
  g ();
}
```

```
// file2.c
// g can be a recursive function
g () {
  if ( . . ) g ();
  if ( . . ) h ();
}

h () {
}
```

Figure *Top-down View* below shows the top-down representation of the above program where procedure `m` is the main entry program, which then calls two procedures: `f` and `g`. Routine `g` can behave as a recursive function depending on the value of the condition branch (lines 3–4). In this figure, we use numerical subscripts to distinguish between different instances of the same procedure. In contrast, in *Bottom-up View* and *Flat View*, we use alphabetic subscripts. We use different labels because there is no natural one-to-one correspondence between the instances in the different views.

Top-down View



Each node of the tree has three boxes: the left-most is the name of the node (or in this case the name of the procedure, the center is the inclusive metric cost, and on the right is the exclusive cost.

Figure *Top-down View* above shows an example of the call chain execution of the program annotated with both inclusive and exclusive costs. Computation of inclusive costs from exclusive costs in the Top-down View involves simply summing up all of the costs in the subtree below.

In this figure, we can see that on the right path of the routine `m`, routine `g` (instantiated in the diagram as `g_1`) performed a recursive call (`g_2`) before calling routine `h`. Although `g_1`, `g_2` and `g_3` are all instances from the same routine (i.e., `g`), we attribute a different cost for each instance. This separation of cost can be critical to identify which instance has a performance problem.

Bottom-up View

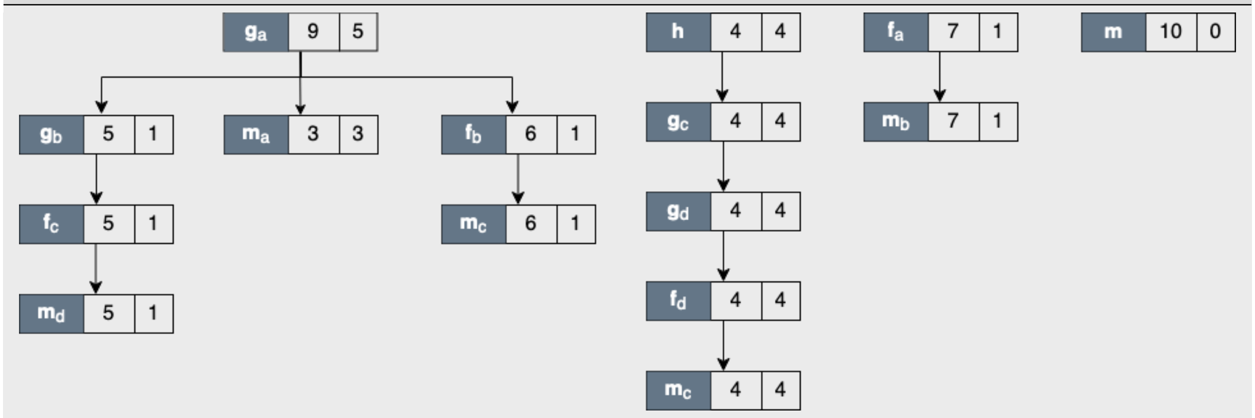
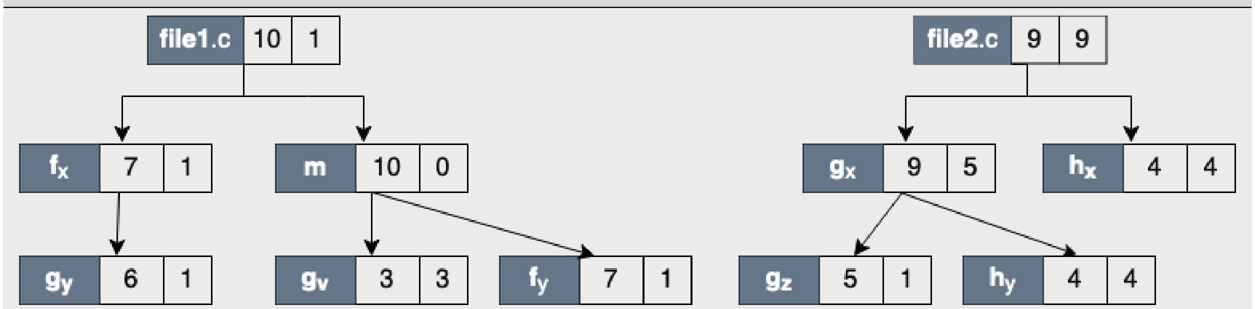


Figure *Bottom-up View* shows the corresponding scope structure for the Bottom-up View and the costs we compute for this recursive program. The procedure `g` noted as `g_a` (which is a root node in the diagram), has different cost to `g` as a callsite as noted as `g_b`, `g_c` and `g_d`. For instance, on the first tree of this figure, the inclusive cost of `g_a` is `9`, which is the sum of the highest cost for each path in the *Top-down View* that includes `g`: the inclusive cost of `g_3` (which is `3`) and `g_1` (which is `6`). We do not attribute the cost of `g_2` here since it is a descendant of `g_1` (in other term, the cost of `g_2` is included in `g_1`).

Flat View



Inclusive costs need to be computed similarly in the Flat View. The inclusive cost of a recursive routine is the sum of the highest cost for each branch in calling context tree. For instance, in Figure *Flat View*, The inclusive cost of `g_x`, defined as the total cost of all instances of `g`, is `9`, and this is consistently the same as the cost in the bottom-up tree. The advantage of attributing different costs for each instance of `g` is that it enables a user to identify which instance of the call to `g` is responsible for performance losses.

11.2.3 Derived Metrics

Frequently, the data become useful only when combined with other information such as the number of instructions executed or the total number of cache accesses. While users don't mind a bit of mental arithmetic and frequently compare values in different columns to see how they relate to a scope, doing this for many scopes is exhausting. To address this problem, `hpcviewer` provides a mechanism for defining metrics. A user-defined metric is called a "derived metric." A derived metric is defined by specifying a spreadsheet-like mathematical formula that refers to data in other columns in the metric table by using `$n` to refer to the value in the `nth` column.

Formulae

The formula syntax supported by `hpcviewer` is inspired by spreadsheet-like in-fix mathematical formulae. Operators have standard algebraic precedence.

Examples

Suppose the database contains information from five executions, where the same two metrics were recorded for each:

1. Metric 0, 2, 4, 6 and 8: total number of cycles
2. Metric 1, 3, 5, 7 and 9: total number of floating point operations

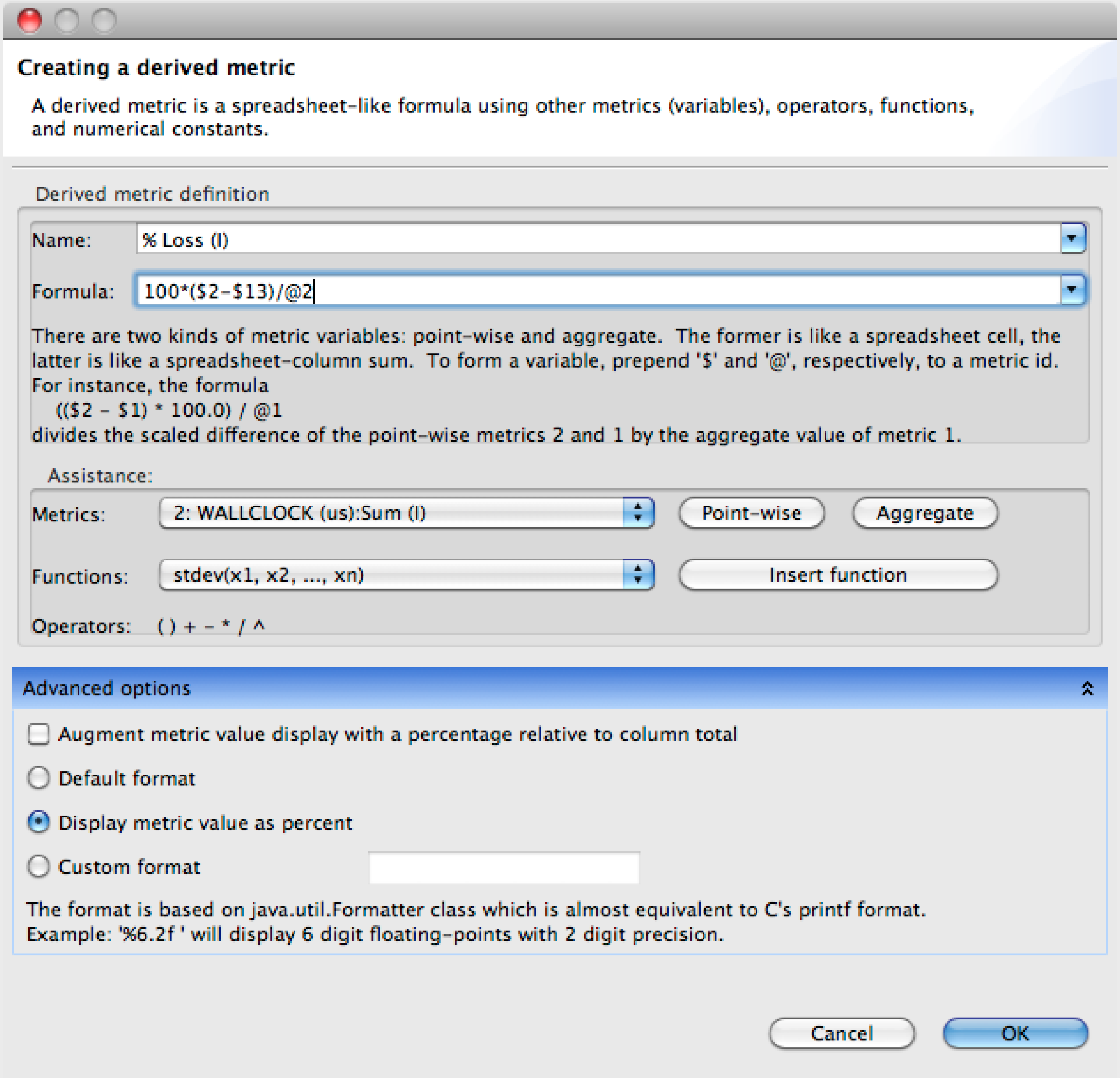
To compute the average number of cycles per floating point operation across all of the executions, we can define a formula as follows:

```
avg($0, $2, $4. $6. $8) / avg($1, $3, $5, $7, $9)
```

Creating Derived Metrics

A derived metric can be created by clicking the **Derived metric** tool item in the navigation/control pane. A derived metric window will then appear as shown in Figure *Derived Metric* below.

Derived metric dialog box



The window has two main parts:

Derived metric definition:

- *New name for the derived metric*. Supply a string that will be used as the column header for the derived metric.
- *Formula definition field*. In this field, the user can define a formula with a spreadsheet-like mathematical formula. A user can type a formula into this field, or use the buttons in *Metrics* pane below to help insert metric terms or function templates.
- *Metrics*. This is used to find the *ID* of a metric. For instance, in this snapshot, the metric `WALLCLOCK` has the ID `2`.
  - `Point-wise` button will insert the metric ID with a "dollar" character (`$`) in the formula field. This dollar prefix refers to the metric value at an individual node in the calling context tree (point-wise) or the value at

the root of the calling context tree (aggregate).

  – `Aggregate` button will insert the metric ID with prefix "at" character (`@`) in the formula field. This prefix refers to the aggregate (root) value of the metric.

- *Functions*. This is to guide the user who wants to insert functions in the formula definition field. Some functions require only one metric as the argument, but some can have two or more arguments. For instance, the function `avg()` which computes the average of some metrics, needs at least two arguments.

Advanced options:

- *Augment metric value display with a percentage relative to column total*. When this box is checked, each scope's derived metric value will be augmented with a percentage value, which for scope *s* is computed as the 100 * (*s*'s derived metric value) / (the derived metric value computed by applying the metric formula to the aggregate values of the input metrics for the entire execution). Such a computation can lead to nonsensical results for some derived metric formulae. For instance, if the derived metric is computed as a ratio of two other metrics, the aforementioned computation that compares the scope's ratio with the ratio for the entire program won't yield a meaningful result. To avoid a confusing metric display, think before you use this button to annotate a metric with its percent of the total.
- *Default format*. This option will display the metric value using scientific notation with three digits of precision, which is the default format.
- *Display metric value as percent*. This option will display the metric value formatted as a percent with two decimal digits. For instance, if the metric has a value `12.3415678`, with this option, it will be displayed as `12.34%`.
- *Custom format*. This option will present the metric value with your customized format. The format is equivalent to Java's Formatter class, or similar to C's printf format. For example, the format "`%6.2f`" will display six-digit floating-points with two digits to the right of the decimal point.

Note that the entered formula and the metric name will be stored automatically. One can then review again the formula (or metric name) by clicking the small triangle of the combo box.

11.2.4 Plotting Metrics for Execution Contexts

An *Execution context* is an abstract concept of a measurable code execution. For example,

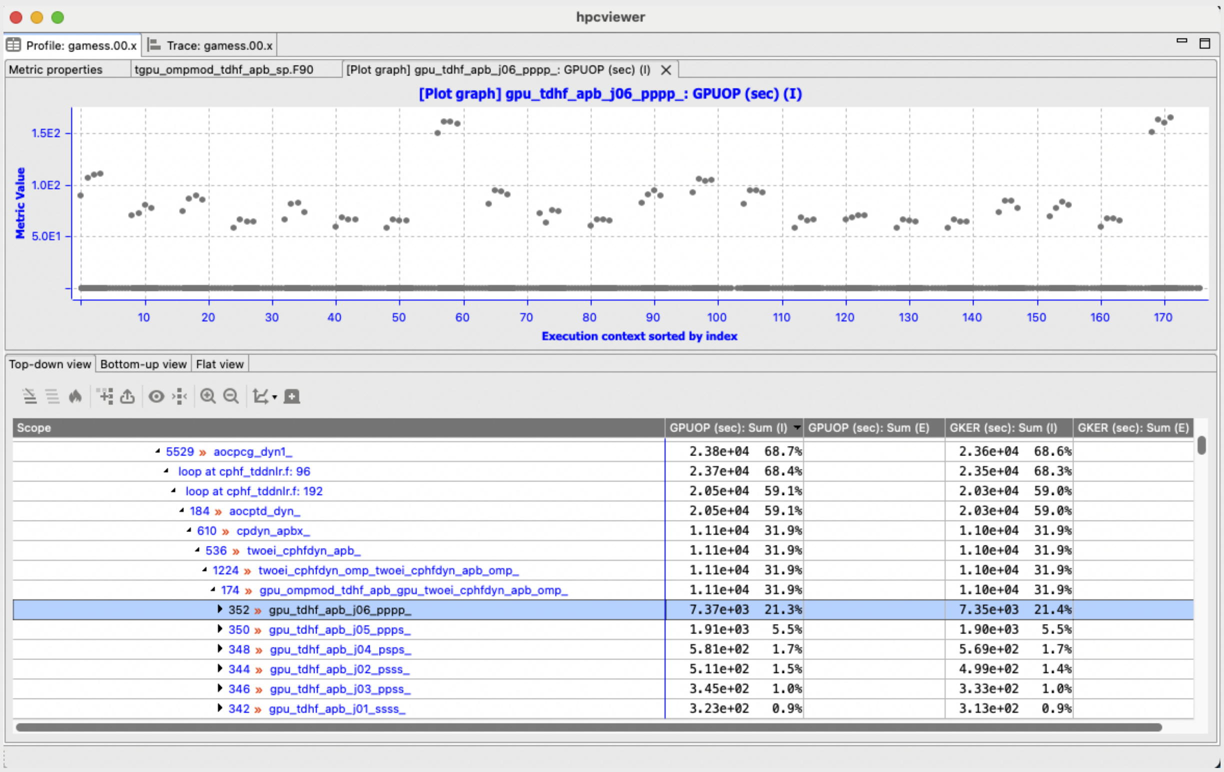
- in a pure MPI application, an execution context is an MPI rank,
- in an OpenMP application, an execution context is an OpenMP thread,
- in a GPU-accelerated application, an execution context may be a CPU thread or a GPU stream, and
- in a hybrid MPI+OpenMP application, an execution context is an MPI rank / thread pair.

There are two types of execution context:

- *Physical* which represents the hardware ID of the execution, such as:
  – `NODE`: the ID of the compute node.
  – `CORE`: the CPU core to which the application thread is bound.
- *Logical* which represents any non-physical entity of the program execution, like:
  – `RANK`: the rank of the process (like the MPI process),
  – `THREAD`: the application CPU thread (such as OpenMP thread),
  – `GPUCONTEXT`: a hardware or software context for a GPU or GPU tile, and
  – `GPUSTREAM`: a stream or queue used to issue operations on a GPU.

Plot Graphs

Plot graph



Plot graph view of a procedure in GAMESS MPI+OpenMP application showing an imbalance where a group of execution contexts have much higher GPU operations than others.

HPCToolkit Experiment databases that have been generated by `hpcprof` can be used by `hpcviewer` to plot graphs of metric values for each execution context. This is particularly useful for quickly assessing load imbalance *in context* across the several threads or processes of an execution. Figure *Plot graph* shows `hpcviewer` rendering such a plot. The horizontal axis shows the application execution context sorted by index (in this case, it's MPI rank and OpenMP thread). The vertical axis shows metric values for each execution context. Because `hpcviewer` can generate scatter plots for any node in the Top-down View, these graphs are calling-context sensitive.

To create a graph, first select a scope in the Top-down View; in the Figure *Plot graph*, the procedure `gpu_tdhf_apb_j06_pppp_` is selected. Then, click the graph button to show the associated sub-menus. At the bottom of the sub-menu is a list of non-empty metrics for the selected scope. Each metric contains a sub-menu that lists the three different types of graphs:

- **Plot graph**: This standard graph plots metric values ordered by their execution context.
- **Sorted plot graph**: This graph plots metric values in ascending order.
- **Histogram graph**: This graph is a histogram of metric values. It divides the range of metric values into a small number of sub-ranges. The graph plots the frequency of the metric that falls into a particular sub-range.

Remark that the label of execution context for a mixed programming model program like a hybrid MPI and OpenMP (Figure *Plot graph*) is

```
PROCESS . THREAD
```

Hence, if the application is a hybrid MPI and OpenMP program, and the execution contexts are 0.0, 0.1, … 31.0, 31.1 it means MPI process 0 has two threads: thread 0 and thread 1 (similarly with MPI process 31).

The viewer also provides additional functionalities by right-clicking on the graph to display the context menus:

- **Adjust Axes Ranges**: to reset the axis X, Y, or both.
- **Zoom In/Out**: to zoom-in or zoom-out the current graph.
- **Save As**: to save the current graph to a file in `*.png` or `.jpeg` format.
- **Properties**: to change the settings of the current graph. This setting is not persistent.
- **Display ...**: this menu is only enabled if one right-click on the dot of the graph. It allows to display the *Thread view* of the specific execution context.

In a graph of metrics for a calling context, hovering over a dot in the graph will display the metric value and the name of its associated execution context. In the Profile view, metrics for a GPU operation are attributed to either the main thread under `<program root>` or aggregated under `<thread root>` for operations launched by a thread with an id > 0. In the graphical thread view, each thread displays its value for the GPU metric associated with the calling context.

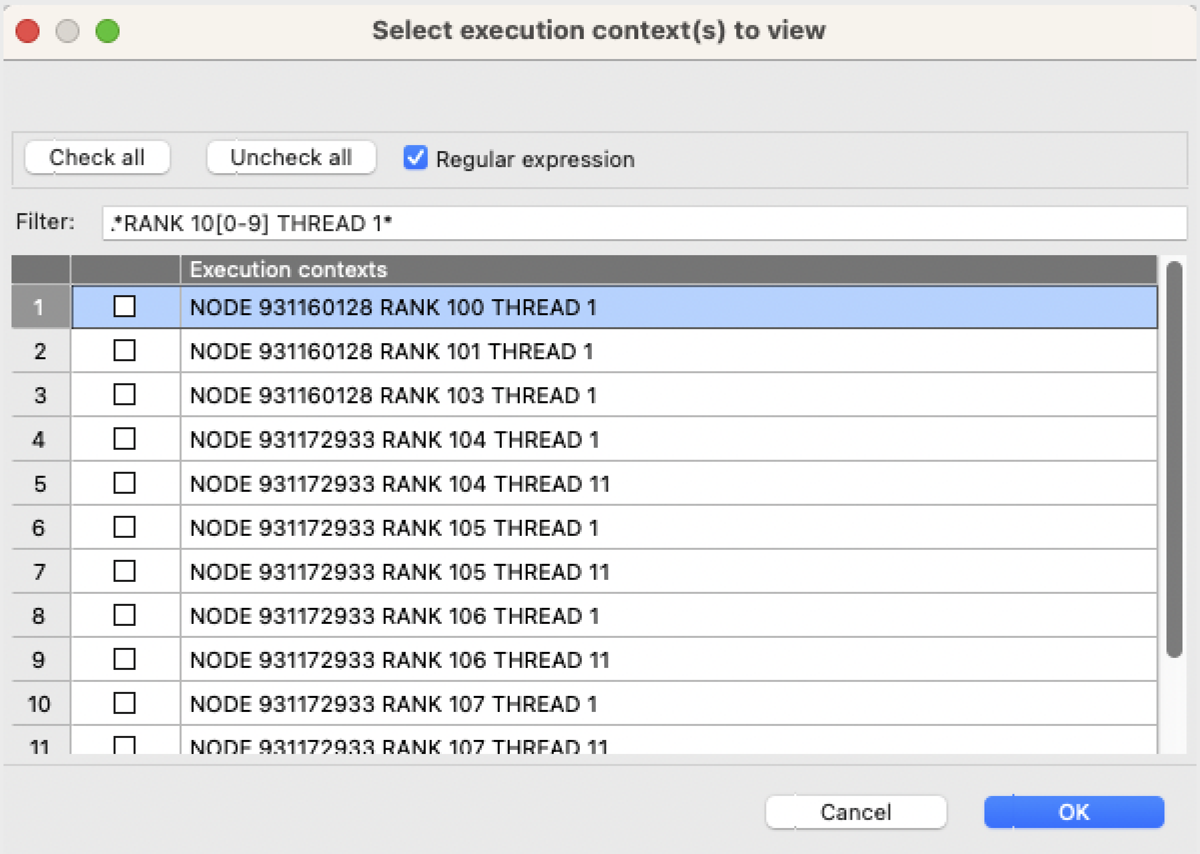
Note:

- Currently, it is only possible to generate scatter plots for metrics directly collected by `hpcrun`, which excludes derived metrics created within `hpcviewer`.

Thread View

`hpcviewer` can also displays the metrics associated with certain execution contexts (i.e., threads and/or processes) named **Thread View**. To select a set of execution contexts, one needs to use the execution-context selection window by clicking button from the *control panel* in the Top-down view.

Selecting execution contexts



A snapshot of an execution-contexts selection window. One can refine the list of execution contexts using regular expression by selecting the Regular expression checkbox.

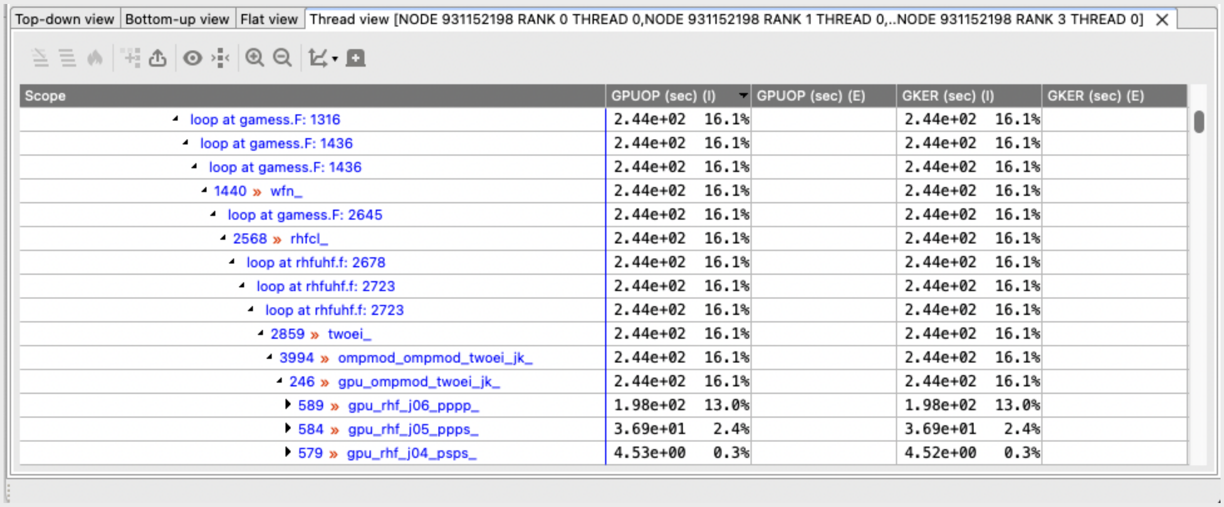
On the *Execution-context selection* window, one needs to select the checkbox of the execution context of interest. To narrow the list, one can specify the thread name on the filter part of the window. For instance for a hybrid MPI and OpenMP application, to display just the main thread (thread zero), one can type:

```
THREAD 0
```

on the filter, and the view only lists all threads 0, such as `RANK 1 THREAD 0`, `RANK 2 THREAD 0`, and `RANK 3 THREAD 0`.

Once execution contexts have been selected, one can click **OK**, and the *Thread View* will be activated. The tree of the view is the same as the tree from the Top-down view, with the metrics only from the selected execution contexts. If there is more than one execution context, the metric value is the sum of the values of the selected execution contexts.

Threads view contains a CCT and metrics of a set of threads



Example of Thread View which displays a set of execution contexts. The first column is a calling-context tree (CCT) equivalent to the CCT in the Top-down View. The next columns represent the metrics from the selected execution contexts (in this case, they are the sum of metrics from rank 0 threads 0 to rank 3 thread 0)

11.2.5 Filtering Tree Nodes

It is often useful to hide *uninteresting* nodes of an execution's Calling Context Tree (CCT) to reduce clutter. For instance, you may want to suppress display of CCT nodes for contexts in stripped libraries that lack correlations to source code, or contexts associated with implementation details of MPI primitives. To do this, `hpcviewer` provides a *filtering* capability that enables a user to incicate names of nodes that should be hidden or whose children should be hidden.

`hpcviewer` enables users to specify multiple filters, each of which is associated with a glob pattern and a filter type. There are three types of filters:

- *self only*: hide each matched node,
- *descendants only*: hide their descendants of each matched node, and
- *self and descendants*: hide each matched node and its descendants.

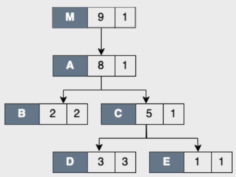
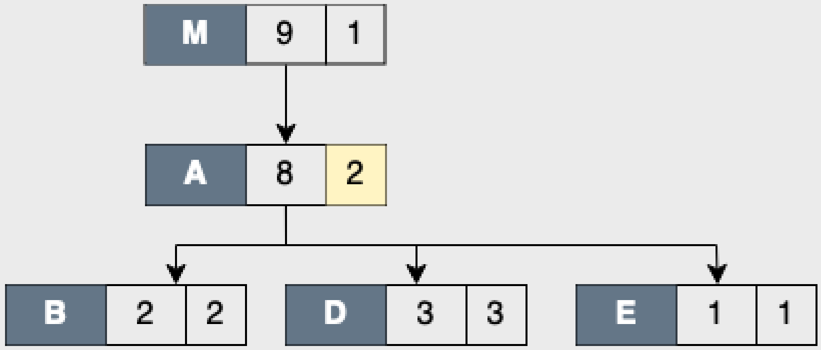
Below we explain the purpose of each type of filter and show an example of each filter's effect on the display of a CCT and its metrics. In the illustrations of CCTs below, each tree node is shown as three boxes: the node label (left), the inclusive cost (middle) and the exclusive cost (right).

Filter Types

Self only

A self-only filter is used to hide nodes that are not of interest. Each node that matches one or more filter patterns will be removed and its children will be displayed as if they were children of the node's closest non-elided ancestor. For example, a self-only filter could be used to hide call sites inside a stripped runtime library that would otherwise appear with names such as `libamdhip64.so.7.0.70001@0x2b23b3`. All call sites in stripped code could be elided with a self-only filter with a glob `*@*` pattern since the only time `@` appears in a node name is as a separator between a library name and virtual address when no symbol name is available.

The nearest non-elided ancestor of an elided node will have its metric costs augmented with the inclusive and exclusive costs of any elided children. Figure *Filter self* shows the result of filtering node `C` of the CCT from Figure *Original CCT*. After filtering, node `C` is elided and its exclusive cost is augmented into the exclusive cost of its parent (node `A`). The children of node `C` (nodes `D` and `E`) are now the children of node `A`.

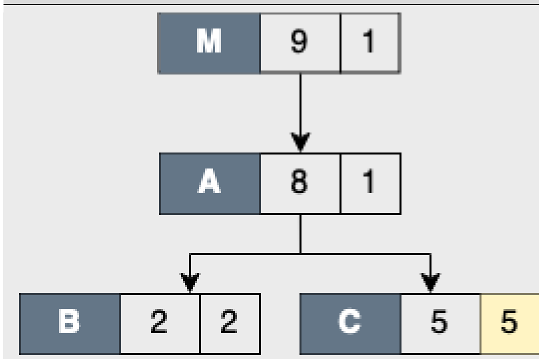
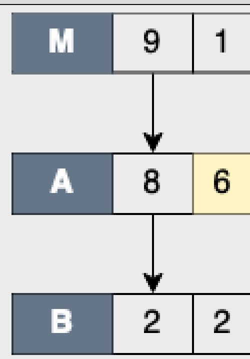
Original CCT	Filtering with "self only"
	
The original top-down tree (or CCT) without any filters.	The result of applying a *self only* filter on node `C` from CCT in Figure *Original CCT*. Node `C` is elided and its children (nodes `D` and `E`) are shown as children of `A` - the first non-elided ancestor of `C`. The yellow highlight shows the updated exclusive cost of node `A`.

Descendants only

This filter elides the descendants of any matched node; the matched node itself is displayed. A common use of this filter is to exclude descendants of MPI functions. Such a filter will show nodes representing the MPI functions themselves, but elide their descendants, which represent details of their implementations. As shown in Figure *Filter descendants*, filtering node `C` causes its descendants `D` and `E` to be elided and `C` and the inclusive cost of `C` is updated to include the costs of its elided descendants.

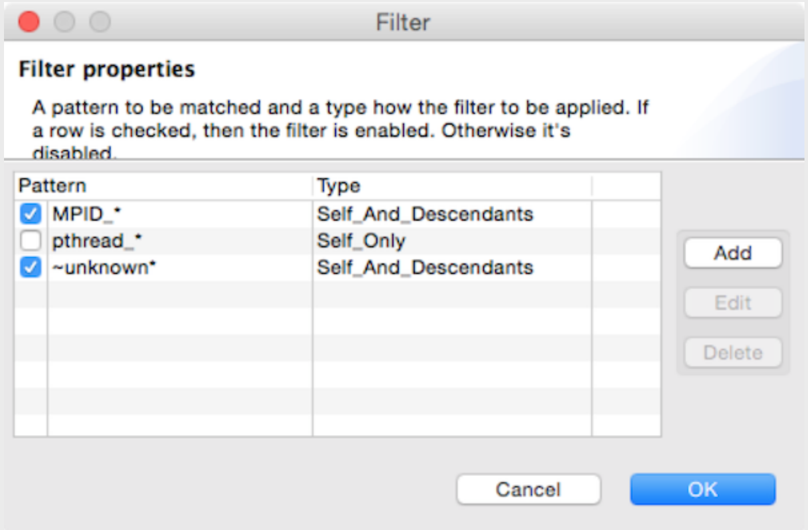
Self and descendants

This filter elides both the matched node and any of its descendants. Such a filter could be used compiler-inserted calls to a runtime function and its implementation details. All of the costs of such elided nodes would be added to the nearest non-elided ancestor of the outlined function. Figure *Filter self and descendants* shows that filtering node `C` will elide the node and its children (nodes `D` and `E`). The total of the exclusive cost of the elided nodes is augmented to the exclusive cost of node `A`.

Filtering with "descendants only"	Filtering with "self and descendants"
	
The result of applying *Descendants only* filter on node `C` from CCT in Figure *Original CCT*. All the children of node `C` (nodes `D` and `E`) are elided, and the total of their exclusive cost is added to node `C`.	The result of applying *Self and descendants* filter on node `C` from CCT in Figure *Original CCT*. Nodes `C` and all the descendants are elided, and their exclusive cost is augmented to node `A`, which is the parent of node `C`.

The filter feature can be accessed by clicking the menu "Filter" and then submenu "Show filter property", which will then show the *Filter window* as shown below:

Filter window



The window consists of a table of filters and a group of action buttons: *add* to create a new filter, *edit* to modify a selected filter, and *delete* to remove a set of selected filters. The table comprises two columns: the left column is to display a filter's switch whether the filter is enabled or disabled, and a glob-like filter pattern; and the second column is to show the type of pattern (self only, children only or self and children). If a checkbox is checked, it signifies the filter is enabled; otherwise, the filter is disabled.

Cautious is needed when using the filter feature since it can change the tree's shape, thus affecting the interpretation of performance analysis. Furthermore, if the filtered nodes are children of a "root" node (such as `<program root>` and `<thread root>`), the exclusive metrics in Bottom-up and flat view can be misleading.

Note that the filter set is global: it affects all open databases in all windows, and it is persistent that it will also affect across `hpcviewer` sessions.

11.2.6 Convenience Features

In this section we describe some features of `hpcviewer` that help improve productivity.

Source Code Pane

The pane is used to display *a copy* of your program's source code or HPCToolkit's performance data in XML format; for this reason, it does not support editing of the pane's contents. To edit a program source code, one should use a favorite editor to edit *the* original copy of the source, not the one stored in HPCToolkit's performance database. Thanks to built-in capabilities in Eclipse, `hpcviewer` supports some useful shortcuts and customization:

- **Find**. To search for a string in the current source pane, `ctrl-f` (Linux and Windows) or `command-f` (Mac) will bring up a find dialog that enables you to enter the target string.
- **Copy**. To copy a selected text into the system's clipboard by pressing `ctrl-c` on Linux and Windows or `command-c` on Mac.

Metric Pane

For the metric pane, `hpcviewer` has some convenient features:

- **Sorting the metric pane contents by a column's values**. First, select the column to sort. If no triangle appears next to the metric, click again. A downward pointing triangle means that the rows in the metric pane are sorted in descending order according to the column's value. Additional clicks on the header of the selected column will toggle back and forth between ascending and descending.

- **Changing column width**. To increase or decrease the width of a column, first put the cursor over the right or left border of the column's header field. The cursor will change into a vertical bar between a left and right arrow. Depress the mouse and drag the column border to the desired position.
- **Changing column order**. If it would be more convenient to have columns displayed in a different order, they can be permuted. Depress and hold the mouse button over the header of column to move and drag the column right or left to its new position.
- **Copying selected metrics into a clipboard**. To copy selected lines of scopes/metrics, one can right-click on the metric pane or navigation pane and then select the menu **Copy**. The copied metrics can then be pasted into any text editor.
- **Hiding or showing metric columns**. Sometimes, it may be more convenient to suppress the display of metrics that are not of current interest. When there are too many metrics to fit on the screen at once, it is often useful to suppress the display of some. The icon above the metric pane will bring up the metric property pane on the source pane area.

  The pane contains a list of metrics sorted according to their order in HPCToolkit's performance database for the application. Each metric column is prefixed by a check box to indicate if the metric should be *displayed* (if checked) or *hidden* (unchecked). To display all metric columns, one can click the **Check all** button. A click to **Uncheck all** will hide all the metric columns. The pane also allows to edit the name of the metric or change the formula of a derived metric. If the metric has no cost, it will be marked with a grey color, and it isn't editable.

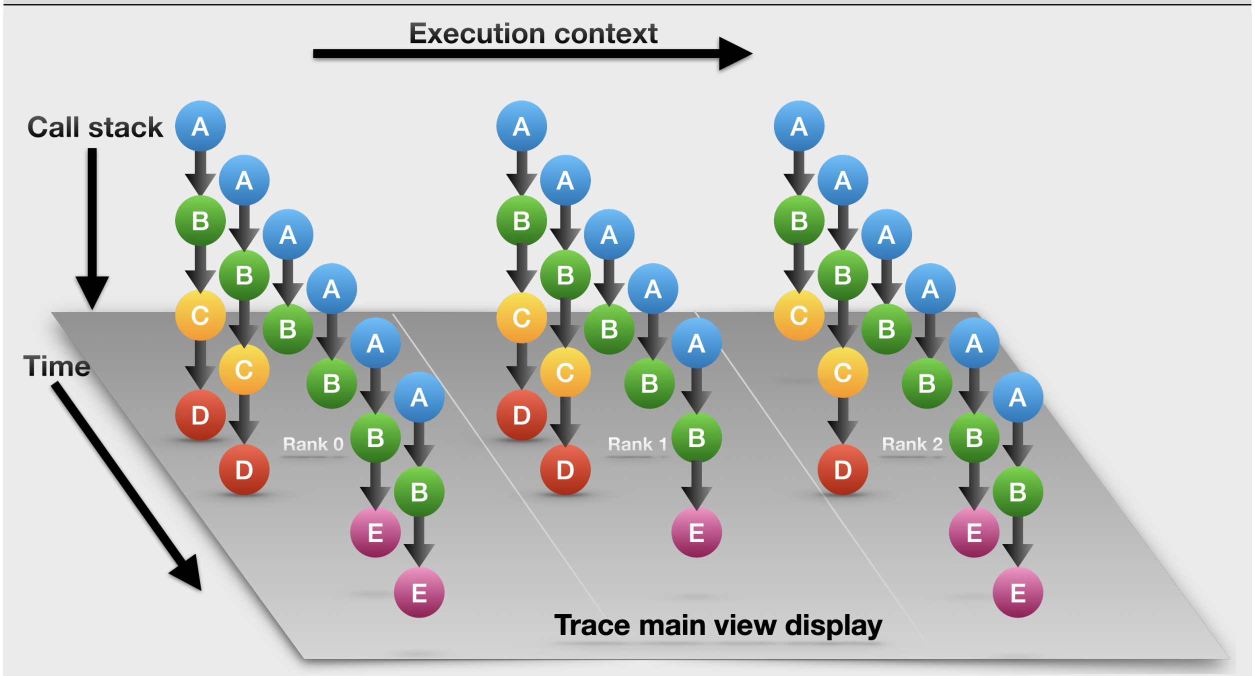
  Finally, an option `Apply to all views` will set the configuration into all views (Top-down, Bottom-up, and Flat views) when checked. Otherwise, the configuration will be applied only on the current view.

11.3 Trace view

Trace view (Tallent et al. 2011) is a time-centric user interface for interactive examination of a sample-based time series (hereafter referred to as a trace) view of a program execution. Trace view can interactively present a large-scale execution trace without concern for the scale of parallelism it represents.

To collect a trace for a program execution, one must instruct HPCToolkit's measurement system to collect a trace. When launching a dynamically-linked executable with `hpcrun`, add the `-t` flag to enable tracing. When collecting a trace, one must also specify a metric to measure. The best way to collect a useful trace is to asynchronously sample the execution with a time-based metric such as `REALTIME`, `CYCLES`, or `CPUTIME`.

Trace dimensions



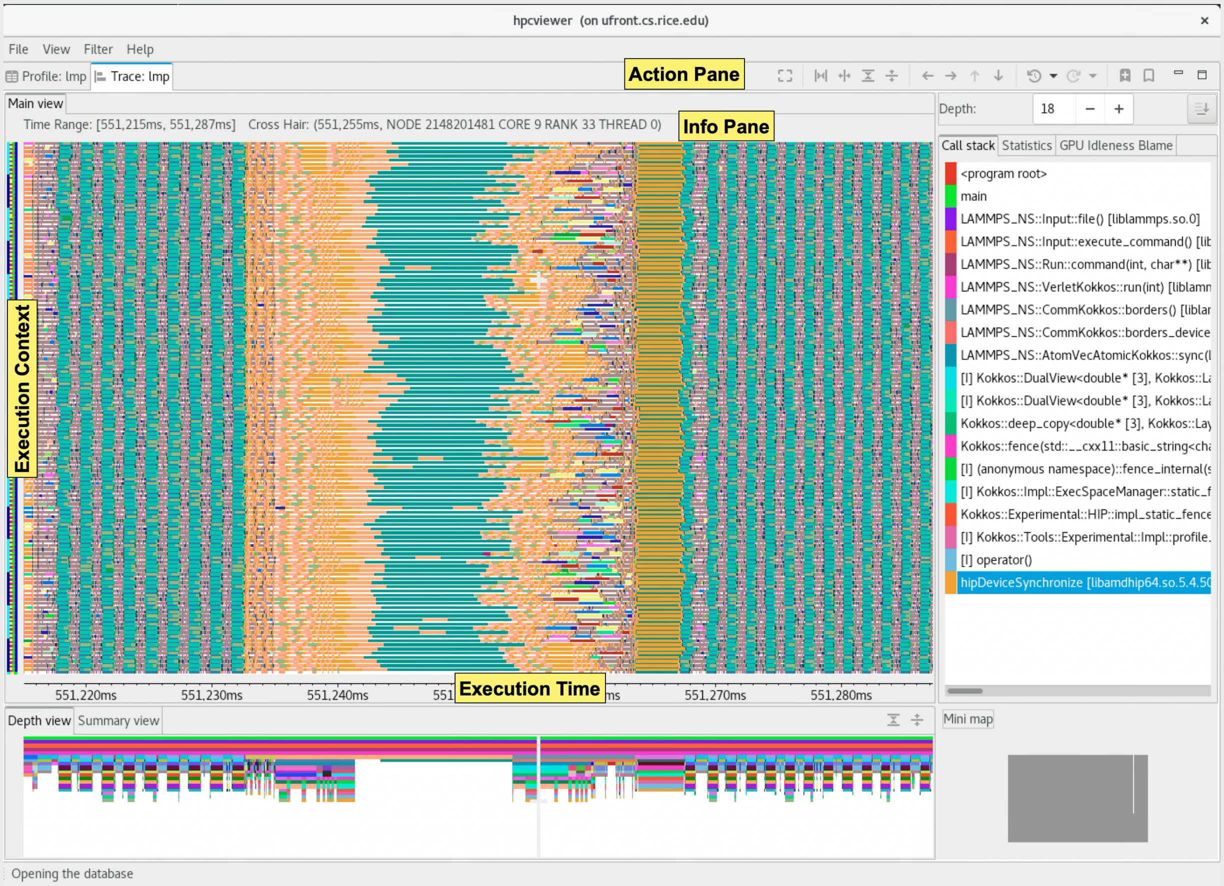
The logical view of traces of call-path samples on three dimensions: **time**, **execution context** (rank/thread/GPU), and **call stack** (also known as call-path).

As shown in the *Trace dimensions* figure above, call-path traces consist of data in three dimensions: *execution context* (also called *profile*) representing a process or a thread rank or a GPU stream, *time*, and *call stack*. A *crosshair* in Trace view is defined by a triplet `(p,t,d)` where `p` is the selected process/thread rank, `t` is the selected time, and `d` is the selected call stack depth.

Trace view renders a view of processes and threads over time. The Depth View shows the call stack depth over time for the thread selected by the cursor. Trace view's Call-stack View shows the call stack associated with the thread and time pair specified by the cursor. Each of these views plays a role in understanding an application's performance.

In the Trace view, each procedure is assigned a specific color. The *Trace dimensions* figure above shows that at depth 1, each call stack has the same color: blue. This node represents the main program that serves as the root of the call chain in all processes at all times. At depth 2, all processes have a green node, which indicates another procedure. At depth 3, in the first time step, all processes have a yellow node; in subsequent time steps, they have purple nodes. This might indicate that the processes are first observed in an initialization procedure (represented by yellow) and later observed in a solve procedure (represented by purple). The pattern of colors that appears in a particular depth slice of the Main View enables a user to visually identify inefficiencies such as load imbalance and serialization.

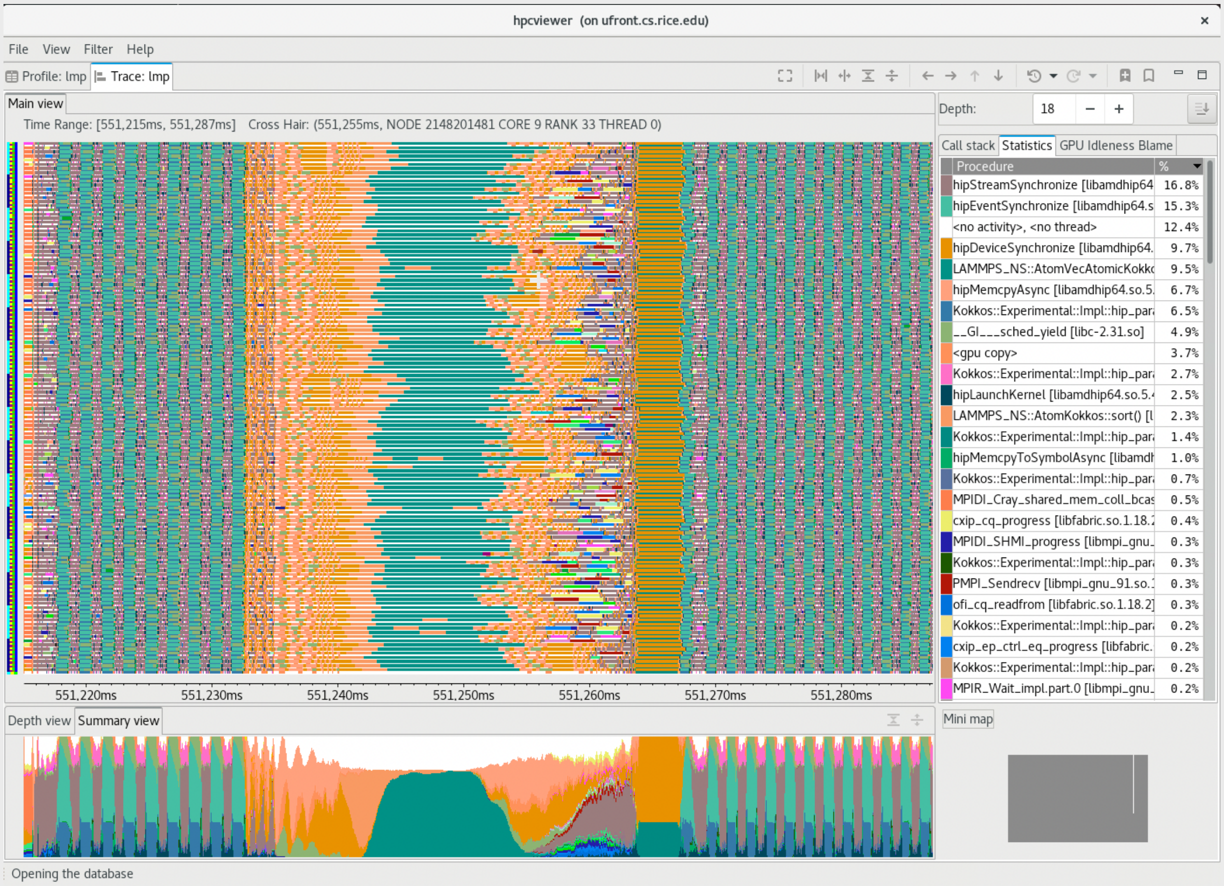
Trace view with legends



A snapshot of traces of an MPI+OpenMP program. The main view shows the *Rank* or *Execution context* as the Y-axis and the program execution *time* as the X-axis.

The above *Trace view* figure highlights Trace view's four principal window panes: *Main View*, *Depth View*, *Call Stack View*, and *Mini Map View*, while the *Trace view* figure below shows two additional window panes: *Summary View* and Statistics View:

Trace view with the Summary View and Statistics View



A screenshot of `hpcviewer`'s Trace view showing the Summary View (tab in bottom, left pane) and Statistics View (tab in top, right pane)

- **Main View** (top tab, left pane): This is the Trace view's primary view. This view shows time on the horizontal axis and the execution context (rank, thread, GPU stream) on the vertical axis; time moves from left to right. Compared to typical process/time views, there is one key difference. The view is a user-controllable slice of the execution-context/time/call-stack space to show the call stack hierarchy (see the *Trace dimensions* figure). Given a call-stack depth, the view shows the color of the currently active procedure at a given time and process rank. (If the requested depth is deeper than a particular call stack, then Trace view simply displays the deepest procedure frame and, space permitting, overlays an annotation indicating the fact that this frame represents a shallower depth.)

  Trace View assigns colors to procedures based on (static) source code procedures. Thus, the same color within the Main and Depth views refers to the same procedure.

  The Main view has a white crosshair representing a selected time and process space. For this selected point, the *Call Stack View* shows the corresponding call stack, while the Depth View shows the selected process.

- **Depth View** (bottom tab, left pane): The view presents a <call-path, time> dimension for the current execution context selected by the Main view's crosshair. It shows for each virtual time along the horizontal axis a stylized call stack along the vertical axis, where 'main' is at the top and leaves (samples) are at the bottom. In other words, this view shows for the whole time range, in a qualitative fashion, what the Call Path View shows for a selected point. The horizontal time axis aligns exactly with the Trace View's time axis; and the colors are consistent across both views. This view has a crosshair corresponding to the currently selected time and call stack depth. One can specify a new crosshair time and a new time range:

   - Selecting a new crosshair time `t` can be done by clicking a pixel within Depth View. This will update the crosshair in Main View and the call path in Call Stack View.
   - Selecting a new time range `[t_m,t_n] = {t | t_m <= t <= t_n}` is performed by first clicking the position of `t_m` and dragging the cursor to the position of `t_n`. A new content in Depth View and Main View is then updated. Note that this action will not update the call path in Call Stack View since it does not change the position of the crosshair.
- **Summary View** (bottom tab, left pane): The view shows the proportion of each subroutine within the current time range. Similar to the Depth view, the Summary view's time range reflects the Trace view's time range…
- **Call Stack View** (top tab, right pane): This view shows two things:
   - the current call stack depth that defines the hierarchical slice shown in the Trace view, and
   - the actual call stack for the point selected by the Trace view's crosshair. To easily coordinate the call stack depth value with the call path, the Call Stack View currently suppresses details such as loop structure and call sites; we may use indentation or other techniques to display this in the future. In this view, the user can select the depth dimension of the Main view by either typing the depth in the depth editor or selecting a procedure in the Call stack view.
- **Statistics View** (top tab, right pane, not shown): This view shows the list of procedures active in the space-time region shown in the Main view at the current call stack depth. Each procedure's percentage in the Statistics view indicates the percentage of pixels in the Main view pane filled with this procedure's color at the current Call stack depth. When the Main view is navigated to show a new time-space interval or the call-stack's depth is changed, the statistics view will update its list of procedures and the percentage of execution time to reflect the new space-time interval or depth selection.
- **GPU Idleness Blame View** (top tab, right pane, not shown): The view is only available if the database contains information on GPU traces. It shows the list of procedures that cause GPU idleness displayed in the trace view. If the trace view displays one CPU thread and multiple GPU streams, then the CPU thread will be blamed for the idleness of those GPU streams. If the view contains more than one CPU thread and multiple GPU streams, then the cost of idleness is shared among the CPU threads.
- **Mini Map View** (bottom tab, right pane): The Mini Map shows, relative to the process/time dimensions, the portion of the execution shown by the Trace View. The Mini Map enables one to zoom and to move from one close-up to another quickly. The user can also move the current selected region to another region by clicking the white rectangle and dragging it to the new place.

11.3.1 Action and Information Pane

Main View is divided into two parts: the top part, which contains *action* and *information* panes, and the main canvas, which displays the traces.

The buttons in the action pane are the following:

- **Home** : Resetting the view configuration into the original view, i.e., viewing traces for all times and processes.
- **Horizontal zoom in / out** : Zooming in/out the time dimension of the traces.
- **Vertical zoom in / out** : Zooming in/out the process dimension of the traces.
- **Navigation buttons** , , , : Navigating the trace view to the left, right, up and bottom, respectively. It is also possible to navigate with the arrow keys in the keyboard. Since Main View does not support scroll bars, the only way to navigate is through navigation buttons (or arrow keys).
- **Undo** : Canceling the action of zoom or navigation and returning back to the previous view configuration.
- **Redo** : Redoing of previously undo change of view configuration.

- **Save / Open a view configuration** : Saving/loading a saved view configuration. A view configuration file contains information about the process/thread and time ranges shown, the selected depth, and the position of the crosshair. It is recommended that the view configuration file be stored in the same directory as the database to ensure that it matches the database since a configuration does not store its associated database. Although it is possible to open a view configuration file associated with a different database, it is not recommended since each database has different time/process dimensions and depth.

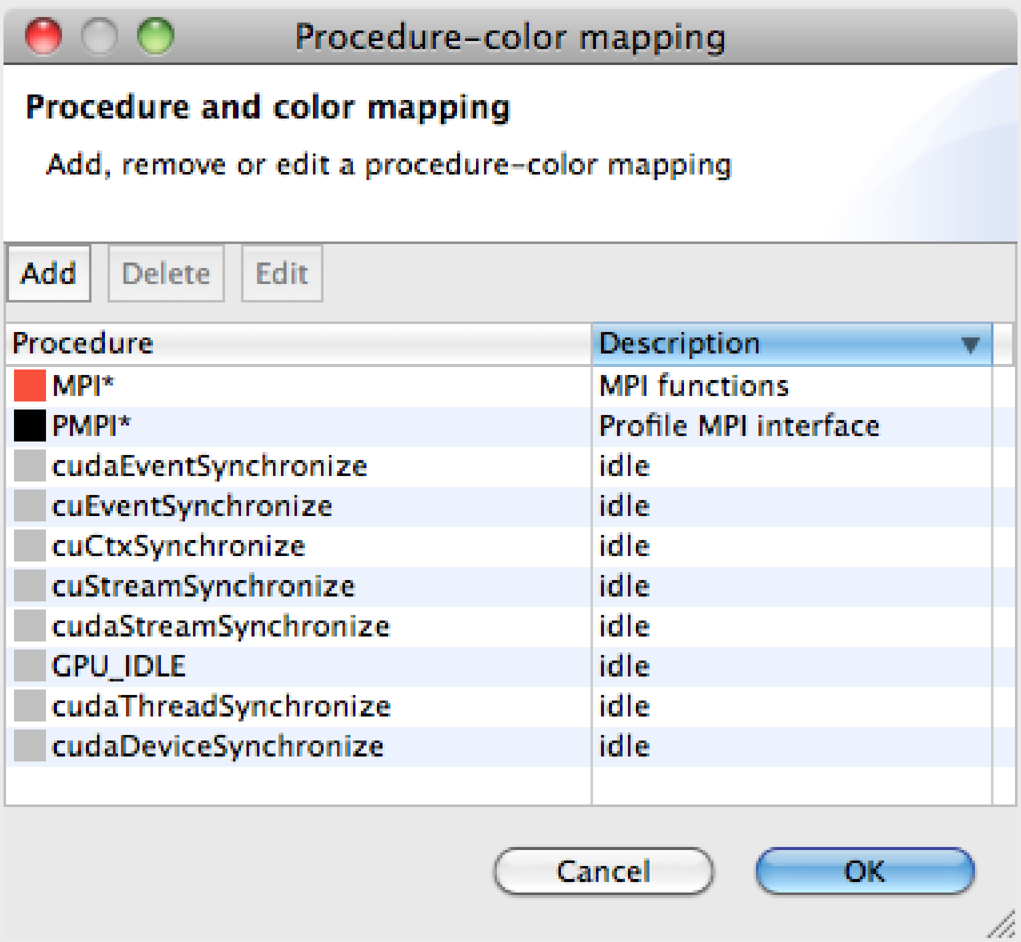
At the top of an execution's Main View pane is information about the data shown in the pane.

- **Time Range**. It shows the time interval of the Main view along the horizontal dimension.
- **Cross Hair**. It indicates the current cursor position in the time and execution-context dimensions.

11.3.2 Customizing the Color Map

Trace view allows users to customize the color of a specific procedure or a group of procedures. To do that, one can select the `View - Color map` menu, and `Color Map` window will appear as shown below:

Color map window



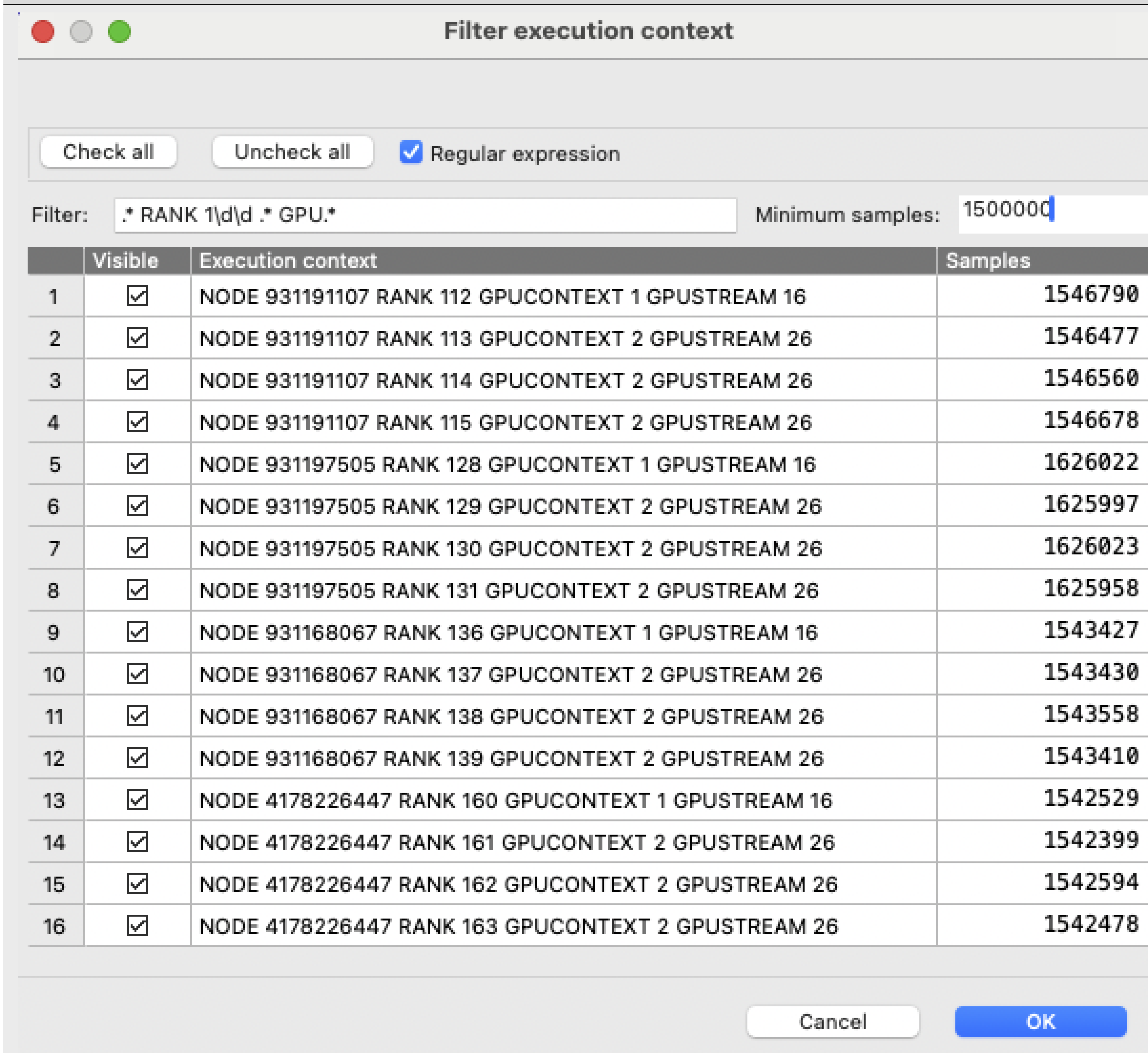
This snapshot shows that any procedure names that match with "`MPI*`" pattern are assigned with red, while procedures that match with "`PMPI*`" pattern are assigned with black.

To add a new procedure-color map, click the `Add` button and a *Color map* window will appear. In this window, one can specify the procedure's name or a *glob* pattern of procedure, and then specify the color to be associated by clicking the color button. Clicking `OK` will close the window and add the new color map to the global list. Note that this map is persistent across sessions, and will apply to other databases as well.

11.3.3 Filtering Execution Contexts

One can select which execution contexts (ranks, threads or GPU streams) to be displayed in Trace View, by selecting the `Filter - execution contexts` menu. This will display a filter window that allows to select which execution contexts to show/hide:

Filter execution context window



An example of narrowing the list of execution contexts using both a regular expression and the minimum number of samples criteria

Similar to *Profile View's thread selection*, one can narrow the list by specifying the name of the execution context on the filter part of the window. In addition, one can also narrow the list based on the minimum number of trace samples (the third column of the table), as shown by the above figure.

Filtering Suggestions

- Sometimes, it is helpful to associate a group of procedures (such as `MPI_*`) to a specific color to approximate its statistic percentage.
- Filtering GPU traces: use the filter menu to select what execution traces to see
  - CPU only, GPU, or a mix: type a string or a regular expression in the chooser to select or unselect the a set of execution contexts
  - 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: e.g., MPI implementation or implementation of CUDA primitives
- Trace view also provides a context menu by right-clicking on the view to save the current display. This context menu is also available on the Depth view and the Summary view.

11.4 Accessing Remote Databases

`hpcviewer` can open performance databases located on remote hosts. Access to performance data on a remote host is supported by running a utility known as `hpcserver` on the remote host to communicate with an instance of `hpcviewer` running elsewhere. With the help of an instance of `hpcserver`, a user running `hpcviewer` can interact with a remote database in real-time, viewing and analyzing performance metrics.

Protecting against unauthorized access was a principal design goal of `hpcviewer`'s capability for remote access to performance data. Several aspects of the design work together to prevent unauthorized access to data on a remote system.

- Before accessing data on a remote host, a local instance of `hpcviewer` must authenticate with an instance of `hpcserver` running on the remote host using standard mechanisms (e.g. password, ssh keys).
- Communication between `hpcviewer` and an instance of `hpcserver` occurs over an SSH tunnel. This guarantees that all communication between `hpcviewer` and `hpcserver` is encrypted, safeguarding it from unauthorized access or tampering.
- To further safeguard communication, an instance of `hpcserver` launched by an authenticated user uses a UNIX domain socket only accessible to that user when communicating with an instance of `hpcviewer`. This prevents access to the socket by other users.
- `hpcviewer` doesn't save a copy of remote performance data to the local file system, which reduces the risk of unauthorized access to the data.

To enable remote access to data on a host, one needs to build and install `hpcserver` on the host, as described in the next section.

11.4.1 Installing `hpcserver`

`hpcserver` supports secure and efficient communication between a server containing an HPCToolkit database and an instance of `hpcviewer` running elsewhere. `hpcserver` is written entirely in Java and is designed for cross-platform compatibility. Below are the requirements and instructions for building and installing `hpcserver`.

To install `hpcserver`, you need Java 17 or newer. You can download a JDK at Adoptium or Oracle JDK

Steps to Install `hpcserver`

1. Go to database release page: https://gitlab.com/hpctoolkit/database/-/releases
2. Download the latest release binary package `server-<version>-bin.tar.gz`

3. Go to the root installation directory. For instance:

```
cd /opt/hpctoolkit/
```

4. Unpack the downloaded binary file

```
tar xf /path/download/server-<version>-bin.tar.gz
```

5. Optionally create a soft link to the server's directory

```
ln -s server-<version>-bin  server
```

   Users will then enter `/opt/hpctoolkit/server` into hpcviewer as the path to the server.

6. It is highly recommended to bind hpcserver to a Java installation directory

```
server/share/hpcserver/bind-java.sh -j  <java_path>
```

Tip

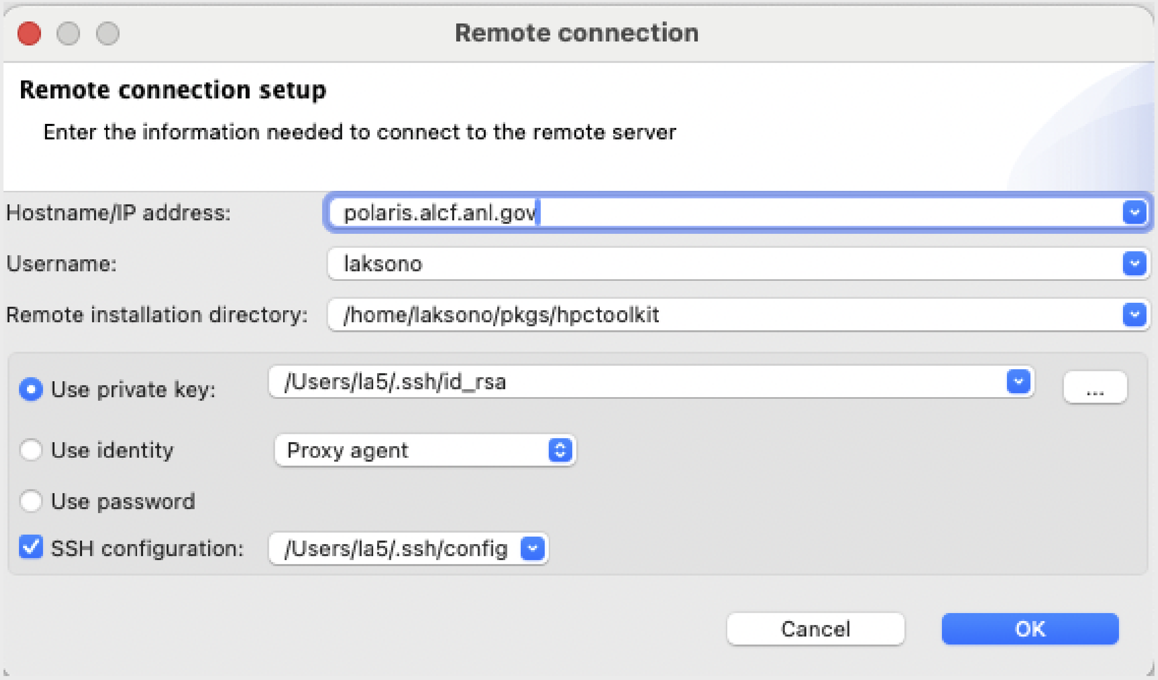
- Always use the `share/hpcserver/bin-java.sh -j /path/to/java-jdk/root` when installing `hpcserver`. Just because `java` is on your path doesn't guarantee that it will be on the path of anyone trying to launch an instance of `hpcserver`.
- When installing `hpcserver`, make sure that the `/path/to/java-jdk/root` is accessible to anyone who will be connecting to the platform with `hpcviewer` to launch an instance of `hpcserver`
- For the convenience of users, `/path/where/to/install/hpcserver` should be short because remote users will need to type the path into a remote instance of `hpcviewer`. If convenient, `/path/where/to/install/hpcserver` can be `/path/to/hpctoolkit-installation`

11.4.2 Opening a Remote Database

Steps to Open a Remote Database

1. Click the menu `File` - `Open remote database`
2. On **Remote connection** window, type the required fields:

Remote connection window



- **Hostname/IP address**: the name of the remote host where `hpcserver` is installed
- **Username**: the username at the remote host
- **Remote installation directory**: the absolute path of `hpcserver` installation. In the above case, it's `/path/hpctoolkit`

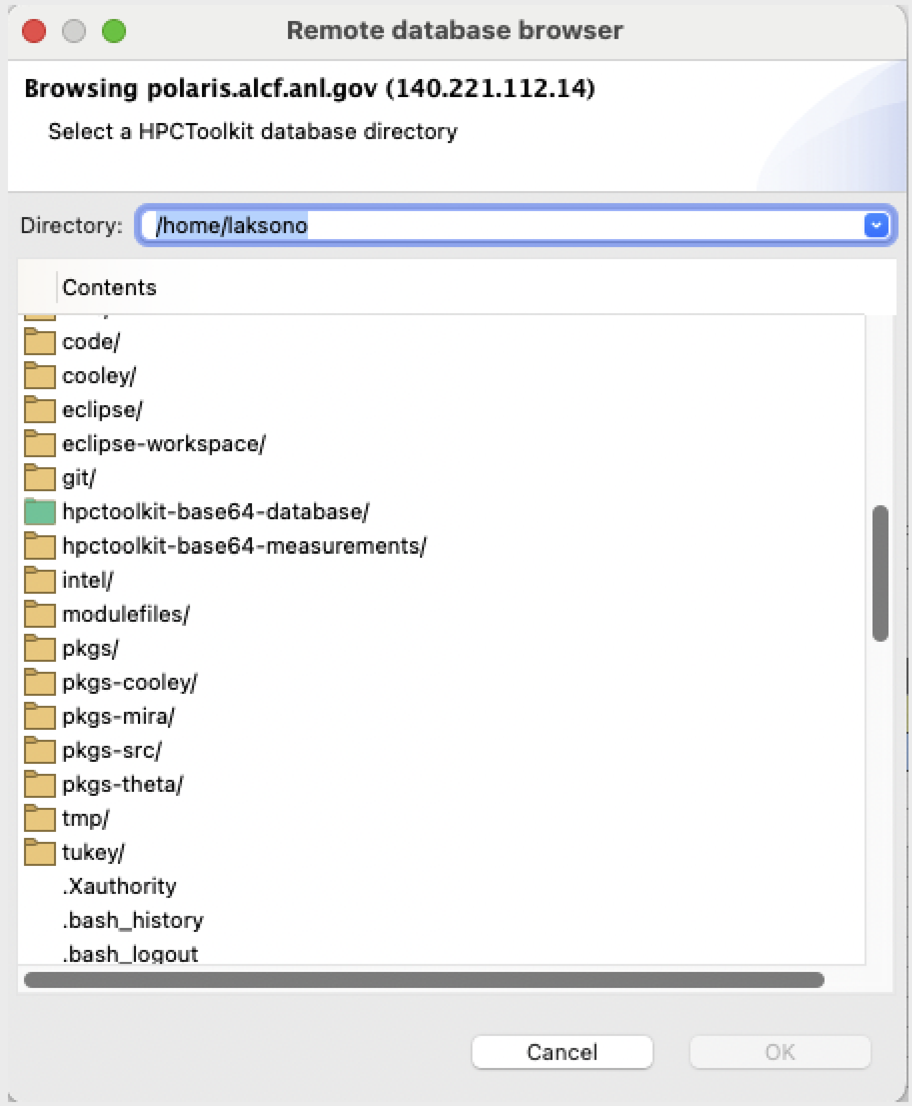
To facilitate the connection, `hpcviewer` allows to connect via three options:

- **Use private key**: use SSH private key whenever possible. This is the recommended way to avoid typing the password all the time.
- **Use identity**: use SSH identity. This is an experimental feature to connect via user's SSH identity.
- **Use password**: use a password to connect. One can check this option if using a private key is problematic.
- **SSH configuration**: use the user's SSH configuration to simplify the connection to a remote host that requires multiple hops or *proxy jump*. Usually the configuration file is located at `$HOME/.ssh/config` on most POSIX platforms.

Once the configuration is set, one needs to click the `OK` button to start the connection.

3. If the connection succeeds, one has to choose an HPCToolkit database from the **Remote database browser** window. The window shows the current remote directory at the top, and the list of its content:

Remote database browser window



- icon represents a regular directory, and one can access it via double-click at the icon or the name of the folder.
- icon represents an HPCToolkit database. Selecting this item will enable the OK button.
- A no icon item represents a regular file.

Note: the OK button will remain disabled until one selects an HPCToolkit database.

CHAPTER

TWELVE

STRATEGIES FOR INSIGHT

This chapter describes some proven strategies for using performance measurements to identify performance bottlenecks in both serial and parallel codes.

12.1 Monitoring High-Latency Events

A very simple and often effective methodology is to profile with respect to cycles and high-latency penalty events. If HPCToolkit attributes a large number of penalty events with a particular source-code statement, there is an extremely high likelihood of significant exposed stalling. This is true even though (1) modern out-of-order processors can overlap the stall latency of one instruction with nearby independent instructions and (2) some penalty events "over count".[4] If a source-code statement incurs a large number of penalty events and it also consumes a non-trivial amount of cycles, then this region of code is an opportunity for optimization. Examples of good penalty events are last-level cache misses and TLB misses.

12.2 Computing Derived Metrics

Modern computer systems provide access to a rich set of hardware performance counters that can directly measure various aspects of a program's performance. Counters in the processor core and memory hierarchy enable one to collect measures of work (e.g., operations performed), resource consumption (e.g., cycles), and inefficiency (e.g., stall cycles). One can also measure time using system timers.

Values of individual metrics are of limited use by themselves. For instance, knowing the count of cache misses for a loop or routine is of little value by itself; only when combined with other information such as the number of instructions executed or the total number of cache accesses does the data become informative. While a developer might not mind using mental arithmetic to evaluate the relationship between a pair of metrics for a particular program scope (e.g., a loop or a procedure), doing this for many program scopes is exhausting. To address this problem, `hpcviewer` supports calculation of derived metrics. `hpcviewer` provides an interface that enables a user to specify spreadsheet-like formula that can be used to calculate a derived metric for every program scope.

Figure *4.1* shows how to use `hpcviewer` to compute a *cycles/instruction* derived metric from measured metrics `PAPI_TOT_CYC` and `PAPI_TOT_INS`; these metrics correspond to *cycles* and *total instructions executed* measured with the PAPI hardware counter interface. To compute a derived metric, one first depresses the button marked `f(x)` above the metric pane; that will cause the pane for computing a derived metric to appear. Next, one types in the formula for the metric of interest. When specifying a formula, existing columns of metric data are referred to using a positional name `$n` to refer to the `n`th column, where the first column is written as `$0`. The metric pane shows the formula `$1/$3`. Here, `$1` refers to the column of data representing the exclusive value for `PAPI_TOT_CYC` and `$3` refers to the column of data representing the exclusive value for `PAPI_TOT_INS`.[5] Positional names for metrics you use in your formula can be determined using the *Metric* pull-down menu in the pane. If you select your metric of choice using the pull-down,

[4] For example, performance monitoring units often categorize a prefetch as a cache miss.

[5] An *exclusive* metric for a scope refers to the quantity of the metric measured for that scope alone; an *inclusive* metric for a scope represents the value measured for that scope as well as costs incurred by any functions it calls. In `hpcviewer`, inclusive metric columns are marked with "(I)" and exclusive metric columns are marked with "(E)."

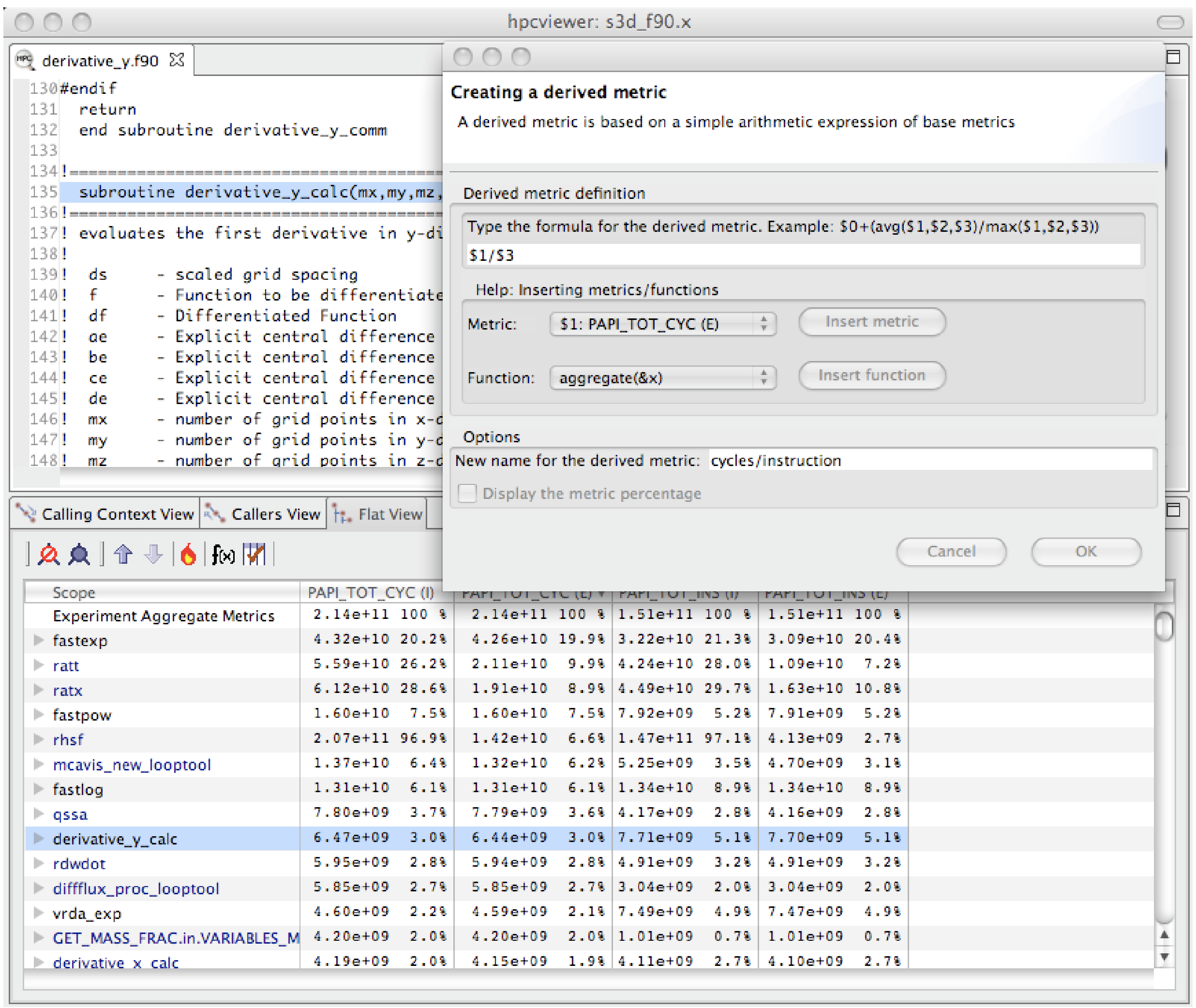


Fig. 1: Computing a derived metric (cycles per instruction) in `hpcviewer`.

you can insert its positional name into the formula using the *insert metric* button, or you can simply type the positional name directly into the formula.

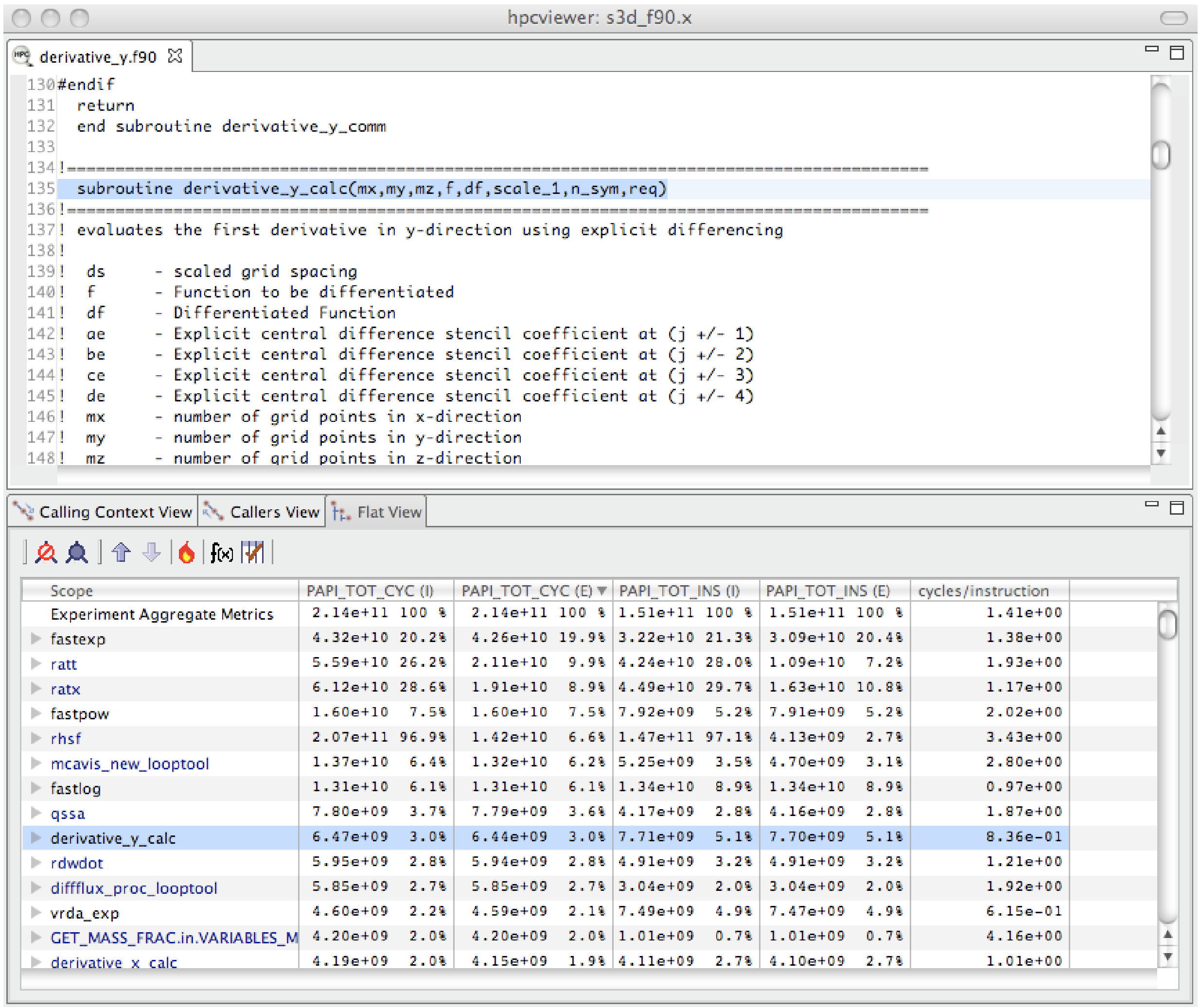


Fig. 2: Displaying the new *cycles/ instruction* derived metric in `hpcviewer`.

At the bottom of the derived metric pane, one can specify a name for the new metric. One also has the option to indicate that the derived metric column should report for each scope what percent of the total its quantity represents; for a metric that is a ratio, computing a percent of the total is not meaningful, so we leave the box unchecked. After clicking the OK button, the derived metric pane will disappear and the new metric will appear as the rightmost column in the metric pane. If the metric pane is already filled with other columns of metric, you may need to scroll right in the pane to see the new metric. Alternatively, you can use the metric check-box pane (selected by depressing the button to the right of `f(x)` above the metric pane) to hide some of the existing metrics so that there will be enough room on the screen to display the new metric. Figure *4.2* shows the resulting `hpcviewer` display after clicking OK to add the derived metric.

The following sections describe several types of derived metrics that are of particular use to gain insight into performance bottlenecks and opportunities for tuning.

12.3 Pinpointing and Quantifying Inefficiencies

While knowing where a program spends most of its time or executes most of its floating point operations may be interesting, such information may not suffice to identify the biggest targets of opportunity for improving program performance. For program tuning, it is less important to know how much resources (e.g., time, instructions) were consumed in each program context than knowing where resources were consumed *inefficiently*.

To identify performance problems, it might initially seem appealing to compute ratios to see how many events per cycle occur in each program context. For instance, one might compute ratios such as FLOPs/cycle, instructions/cycle, or cache miss ratios. However, using such ratios as a sorting key to identify inefficient program contexts can misdirect a user's attention. There may be program contexts (e.g., loops) in which computation is terribly inefficient (e.g., with low operation counts per cycle); however, some or all of the least efficient contexts may not account for a significant amount of execution time. Just because a loop is inefficient doesn't mean that it is important for tuning.

The best opportunities for tuning are where the aggregate performance losses are greatest. For instance, consider a program with two loops. The first loop might account for 90% of the execution time and run at 50% of peak performance. The second loop might account for 10% of the execution time, but only achieve 12% of peak performance. In this case, the total performance loss in the first loop accounts for 50% of the first loop's execution time, which corresponds to 45% of the total program execution time. The 88% performance loss in the second loop would account for only 8.8% of the program's execution time. In this case, tuning the first loop has a greater potential for improving the program performance even though the second loop is less efficient.

A good way to focus on inefficiency directly is with a derived *waste* metric. Fortunately, it is easy to compute such useful metrics. However, there is no one *right* measure of waste for all codes. Depending upon what one expects as the rate-limiting resource (e.g., floating-point computation, memory bandwidth, etc.), one can define an appropriate waste metric (e.g., FLOP opportunities missed, bandwidth not consumed) and sort by that.

For instance, in a floating-point intensive code, one might consider keeping the floating point pipeline full as a metric of success. One can directly quantify and pinpoint losses from failing to keep the floating point pipeline full *regardless of why this occurs*. One can pinpoint and quantify losses of this nature by computing a *floating-point waste* metric that is calculated as the difference between the potential number of calculations that could have been performed if the computation was running at its peak rate minus the actual number that were performed. To compute the number of calculations that could have been completed in each scope, multiply the total number of cycles spent in the scope by the peak rate of operations per cycle. Using `hpcviewer`, one can specify a formula to compute such a derived metric and it will compute the value of the derived metric for every scope. Figure *4.3* shows the specification of this floating-point waste metric for a code.[6]

Sorting by a waste metric will rank order scopes to show the scopes with the greatest waste. Such scopes correspond directly to those that contain the greatest opportunities for improving overall program performance. A waste metric will typically highlight loops where

- a lot of time is spent computing efficiently, but the aggregate inefficiencies accumulate,
- less time is spent computing, but the computation is rather inefficient, and
- scopes such as copy loops that contain no computation at all, which represent a complete waste according to a metric such as floating point waste.

Beyond identifying and quantifying opportunities for tuning with a waste metric, one can compute a companion derived metric *relative efficiency* metric to help understand how easy it might be to improve performance. A scope running at very high efficiency will typically be much harder to tune than running at low efficiency. For our floating-point waste metric, we one can compute the floating point efficiency metric by dividing measured FLOPs by potential peak FLOPs and multiplying the quantity by 100. Figure *4.4* shows the specification of this floating-point efficiency metric for a code.

Scopes that rank high according to a waste metric and low according to a companion relative efficiency metric often make the best targets for optimization. Figure *4.5* shows the specification of this floating-point efficiency metric for

[6] Many recent processors have trouble accurately counting floating-point operations accurately, which is unfortunate. If your processor can't accurately count floating-point operations, a floating-point waste metric will be less useful.

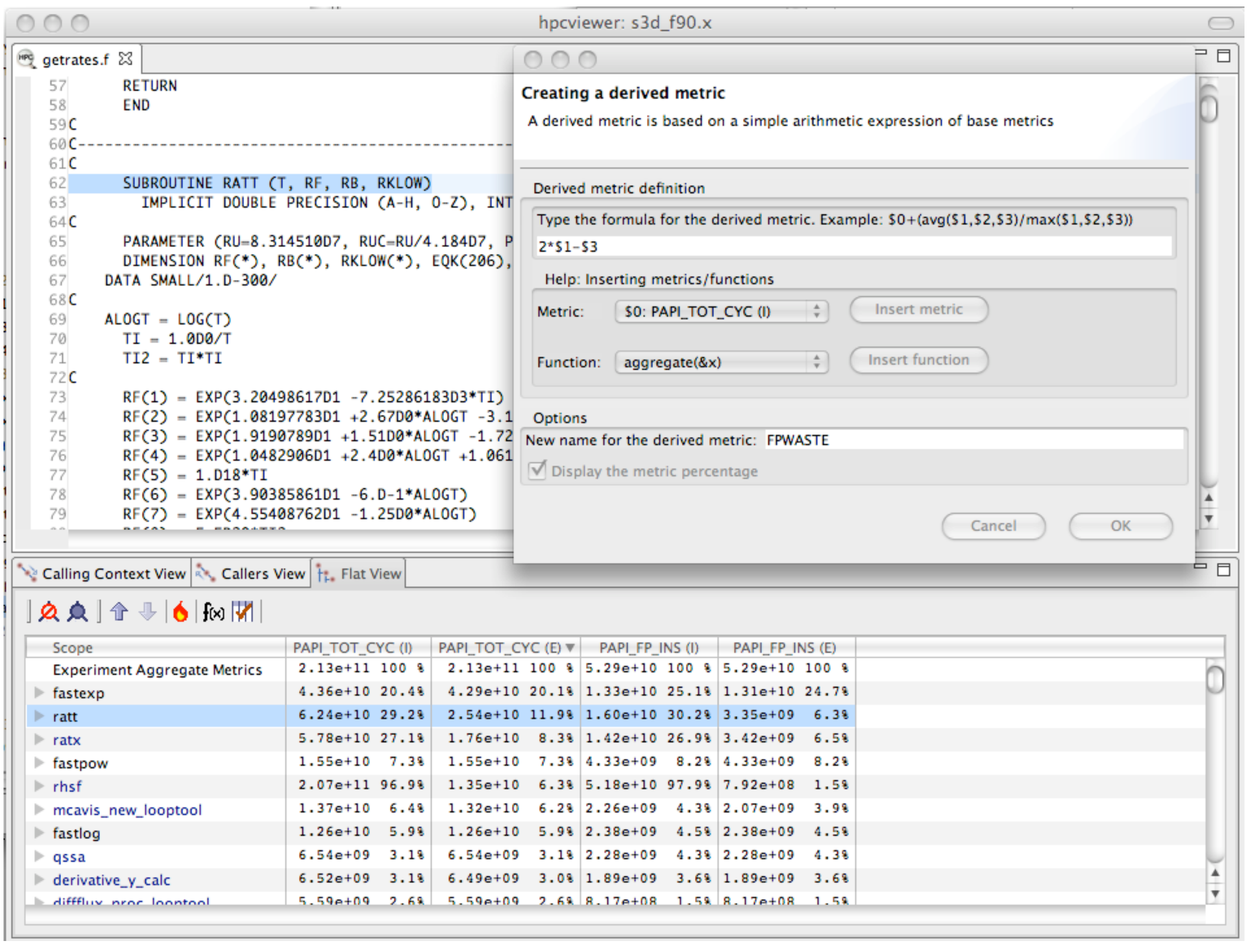


Fig. 3: Computing a floating point waste metric in `hpcviewer`.

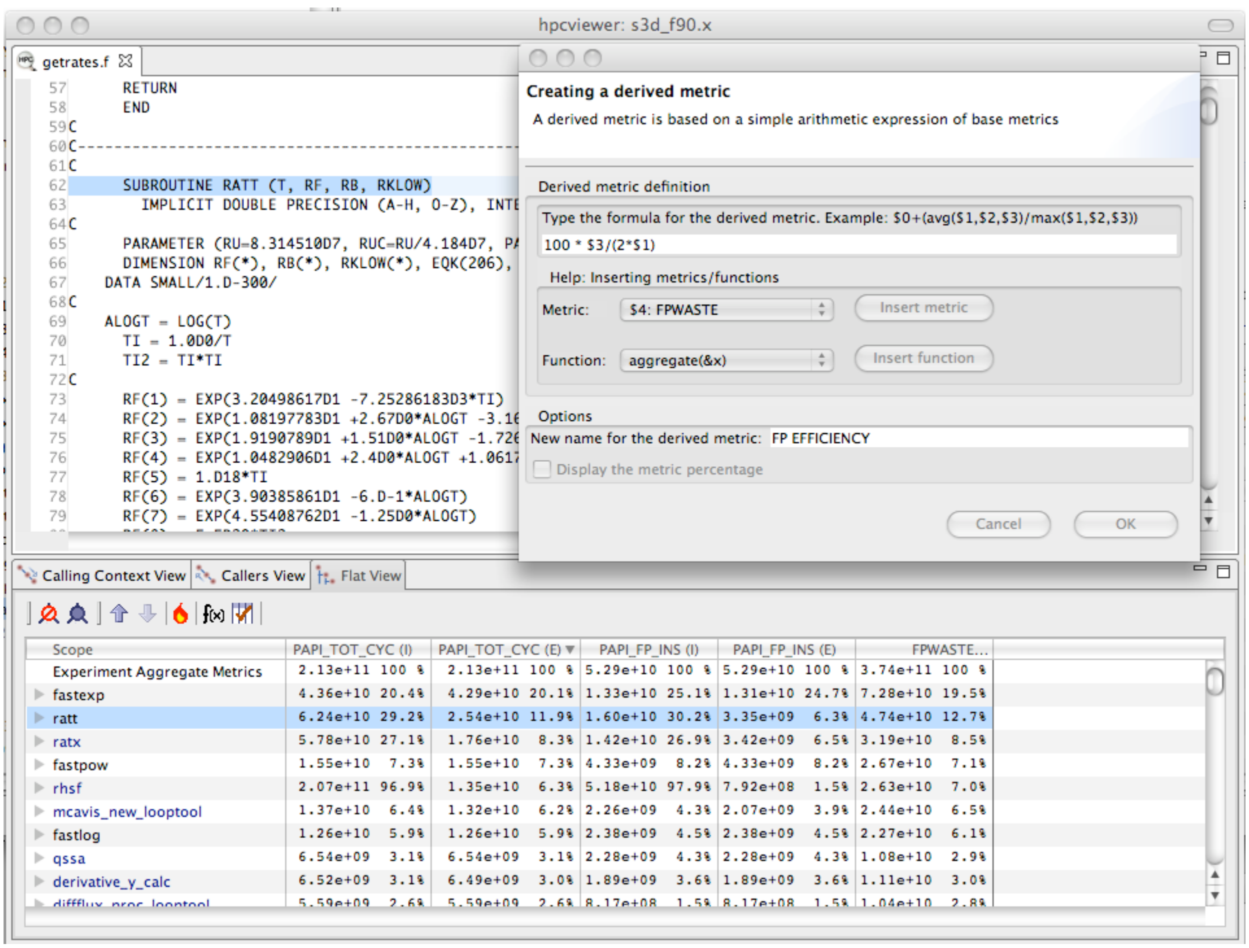


Fig. 4: Computing floating point efficiency in percent using `hpcviewer`.

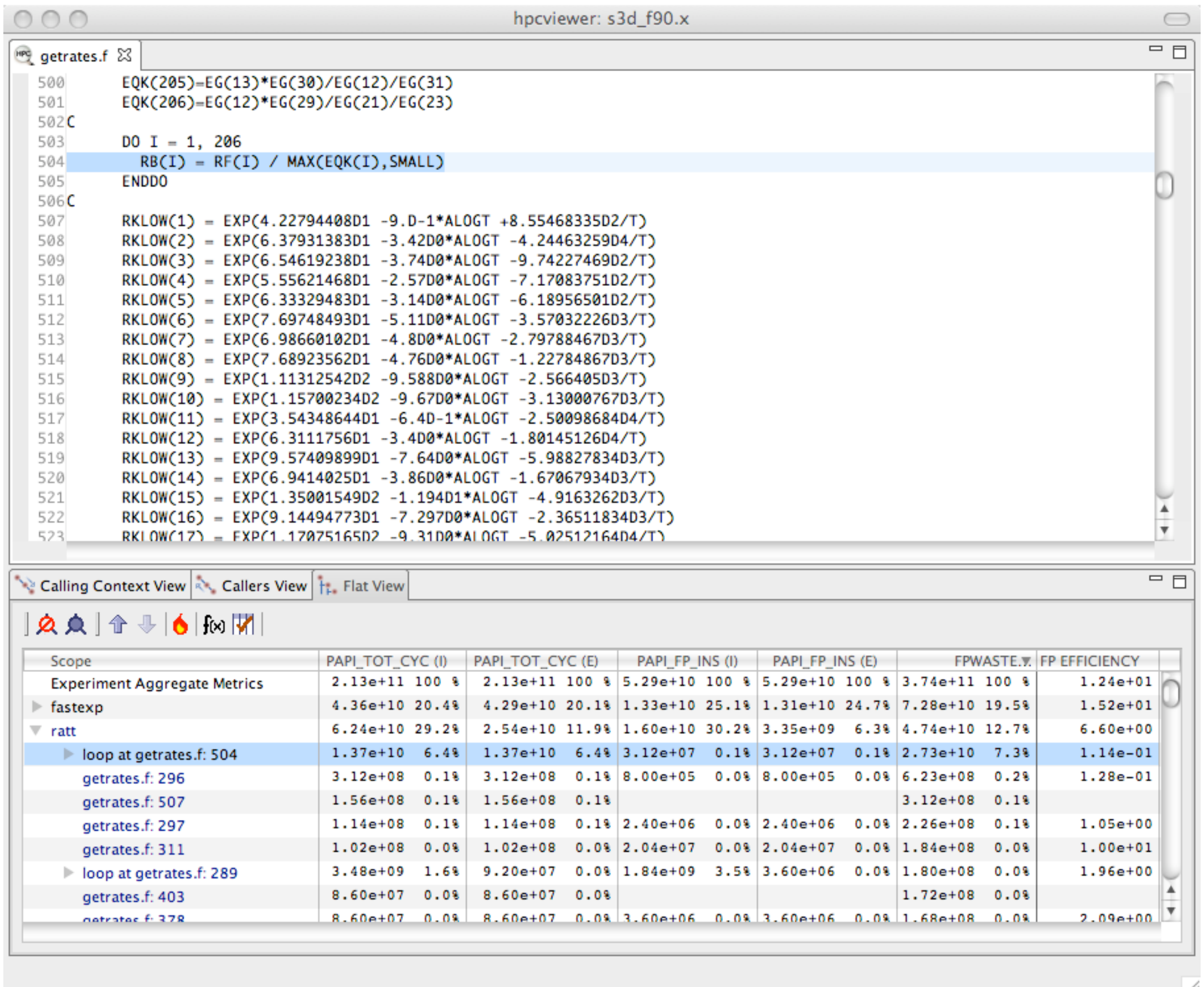


Fig. 5: Using floating point waste and the percent of floating point efficiency to evaluate opportunities for optimization.

a code. Figure *4.5* shows an `hpcviewer` display that shows the top two routines that collectively account for 32.2% of the floating point waste in a reactive turbulent combustion code. The second routine (`ratt`) is expanded to show the loops and statements within. While the overall floating point efficiency for `ratt` is at 6.6% of peak (shown in scientific notation in the `hpcviewer` display), the most costly loop in `ratt` that accounts for 7.3% of the floating point waste is executing at only 0.114% efficiency. Identifying such sources of inefficiency is the first step towards improving performance via tuning.

12.4 Analyzing Scalability Bottlenecks

On large-scale parallel systems, identifying impediments to scalability is of paramount importance. On today's systems fashioned out of multicore processors, two kinds of scalability are of particular interest:

- scaling within nodes, and
- scaling across the entire system.

HPCToolkit can be used to readily pinpoint both kinds of bottlenecks. Using call path profiles collected by `hpcrun`, it is possible to quantify and pinpoint scalability bottlenecks of any kind, *regardless of cause*.

To pinpoint scalability bottlenecks in parallel programs, we use *differential profiling* — mathematically combining corresponding buckets of two or more execution profiles. Differential profiling was first described by McKenney (McKenney 1999); he used differential profiling to compare two *flat* execution profiles. Differencing of flat profiles is useful for identifying what parts of a program incur different costs in two executions. Building upon McKenney's idea of differential profiling, we compare call path profiles of parallel executions at different scales to pinpoint scalability bottlenecks. Differential analysis of call path profiles pinpoints not only differences between two executions (in this case scalability losses), but the contexts in which those differences occur. Associating changes in cost with full calling contexts is particularly important for pinpointing context-dependent behavior. Context-dependent behavior is common in parallel programs. For instance, in message passing programs, the time spent by a call to `MPI_Wait` depends upon the context in which it is called. Similarly, how the performance of a communication event scales as the number of processors in a parallel execution increases depends upon a variety of factors such as whether the size of the data transferred increases and whether the communication is collective or not.

12.4.1 Scalability Analysis Using Expectations

Application developers have expectations about how the performance of their code should scale as the number of processors in a parallel execution increases. Namely,

- when different numbers of processors are used to solve the same problem (strong scaling), one expects an execution's speedup to increase linearly with the number of processors employed;
- when different numbers of processors are used but the amount of computation per processor is held constant (weak scaling), one expects the execution time on a different number of processors to be the same.

In both of these situations, a code developer can express their expectations for how performance will scale as a formula that can be used to predict execution performance on a different number of processors. One's expectations about how overall application performance should scale can be applied to each context in a program to pinpoint and quantify deviations from expected scaling. Specifically, one can scale and difference the performance of an application on different numbers of processors to pinpoint contexts that are not scaling ideally.

To pinpoint and quantify scalability bottlenecks in a parallel application, we first use `hpcrun` to a collect call path profile for an application on two different numbers of processors. Let `E_p` be an execution on `p` processors and `E_q` be an execution on `q` processors. Without loss of generality, assume that `q > p`.

In our analysis, we consider both *inclusive* and *exclusive* costs for CCT nodes. The inclusive cost at `n` represents the sum of all costs attributed to `n` and any of its descendants in the CCT, and is denoted by `I(n)`. The exclusive cost at `n` represents the sum of all costs attributed strictly to `n`, and we denote it by `E(n)`. If `n` is an interior node in a CCT, it represents an invocation of a procedure. If `n` is a leaf in a CCT, it represents a statement inside some procedure. For leaves, their inclusive and exclusive costs are equal.

It is useful to perform scalability analysis for both inclusive and exclusive costs; if the loss of scalability attributed to the inclusive costs of a function invocation is roughly equal to the loss of scalability due to its exclusive costs, then we know that the computation in that function invocation does not scale. However, if the loss of scalability attributed to a function invocation's inclusive costs outweighs the loss of scalability accounted for by exclusive costs, we need to explore the scalability of the function's callees.

Given CCTs for an ensemble of executions, the next step to analyzing the scalability of their performance is to clearly define our expectations. Next, we describe performance expectations for weak scaling and intuitive metrics that represent how much performance deviates from our expectations. More information about our scalability analysis technique can be found elsewhere (Coarfa et al. 2007; Tallent et al. 2009).

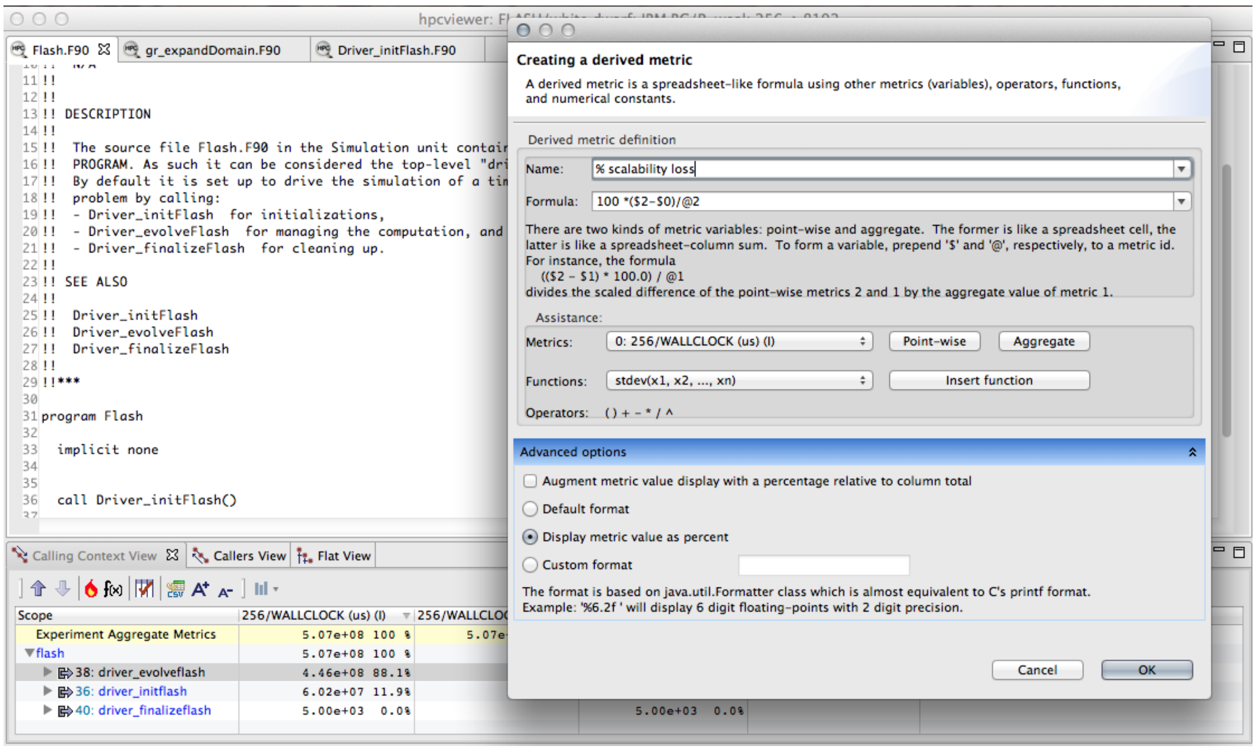


Fig. 6: Computing the scaling loss when weak scaling a white dwarf detonation simulation with FLASH3 from 256 to 8192 cores. For weak scaling, the time on an MPI rank in each of the simulations will be the same. In the figure, column 0 represents the inclusive cost for one MPI rank in a 256-core simulation; column 2 represents the inclusive cost for one MPI rank in an 8192-core simulation. The difference between these two columns, computed as `$2-$0`, represents the excess work present in the larger simulation for each unique program context in the calling context tree. Dividing that by the total time in the 8192-core execution `@2` gives the fraction of wasted time. Multiplying through by 100 gives the percent of the time wasted in the 8192-core execution, which corresponds to the % scalability loss.

Weak Scaling

Consider two weak scaling experiments executed on `p` and `q` processors, respectively, `p < q`. In Figure *4.6* shows how we can use a derived metric to compute and attribute scalability losses. Here, we compute the difference in inclusive cycles spent on one core of a 8192-core run and one core in a 256-core run in a weak scaling experiment. If the code had perfect weak scaling, the time for an MPI rank in each of the executions would be identical. In this case, they are not. We compute the excess work by computing the difference for each scope between the time on the 8192-core run and the time on the 256-core core run. We normalize the differences of the time spent in the two runs by dividing then by the total time spent on the 8192-core run. This yields the fraction of wasted effort for each scope when scaling from 256 to 8192 cores. Finally, we multiply these results by 100 to compute the % scalability loss. This example shows how one can compute a derived metric to that pinpoints and quantifies scaling losses across different node counts of a

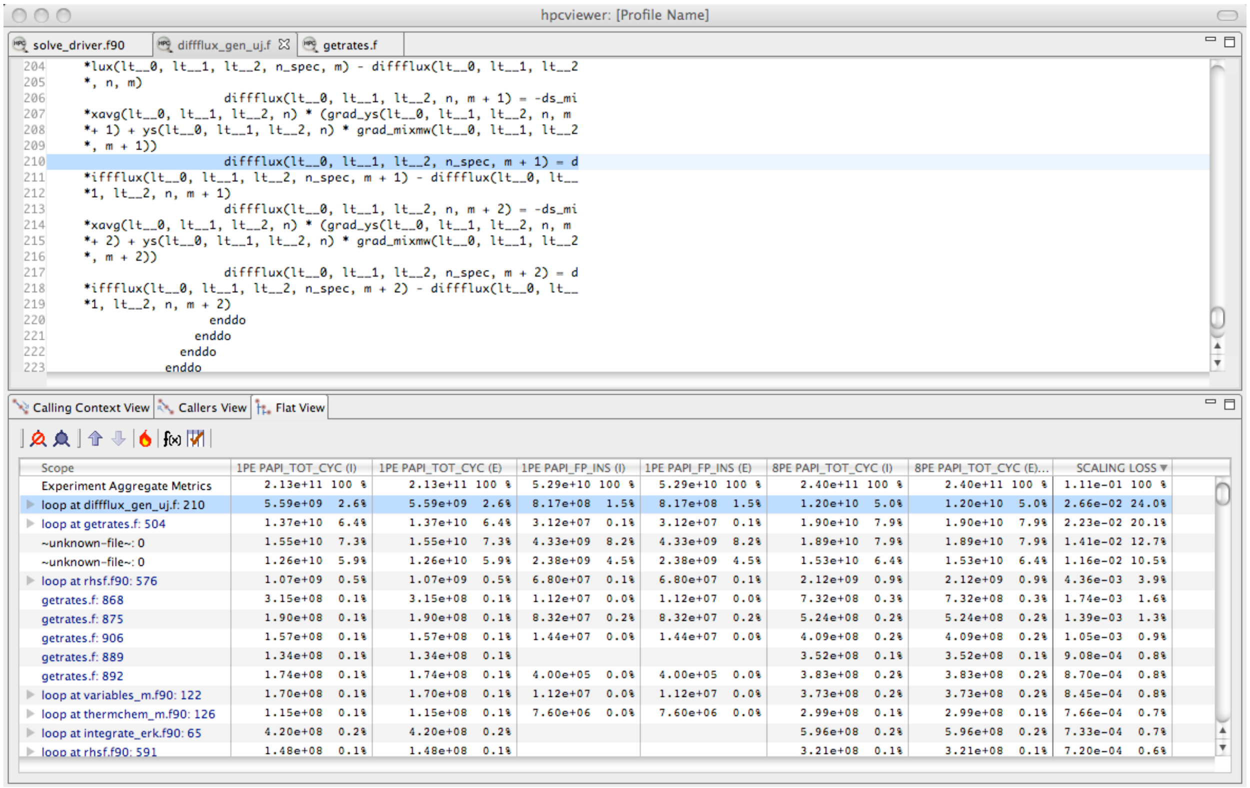


Fig. 7: Using the fraction the scalability loss metric of Figure *4.6* to rank order loop nests by their scaling loss.

Blue Gene/P system.

A similar analysis can be applied to compute scaling losses between jobs that use different numbers of core counts on individual processors. Figure *4.7* shows the result of computing the scaling loss for each loop nest when scaling from one to eight cores on a multicore node and rank order loop nests by their scaling loss metric. Here, we simply compute the scaling loss as the difference between the cycle counts of the eight-core and the one-core runs, divided through by the aggregate cost of the process executing on eight cores. This figure shows the scaling lost written in scientific notation as a fraction rather than multiplying through by 100 to yield a percent. In this figure, we examine scaling losses in the flat view, showing them for each loop nest. The source pane shows the loop nest responsible for the greatest scaling loss when scaling from one to eight cores. Unsurprisingly, the loop with the worst scaling loss is very memory intensive. Memory bandwidth is a precious commodity on multicore processors.

While we have shown how to compute and attribute the fraction of excess work in a weak scaling experiment, one can compute a similar quantity for experiments with strong scaling. When differencing the costs summed across all of the threads in a pair of strong-scaling experiments, one uses exactly the same approach as shown in Figure *4.6*. If comparing weak scaling costs summed across all ranks in `p` and `q` core executions, one can simply scale the aggregate costs by `1/p` and `1/q` respectively before differencing them.

Exploring Scaling Losses

Scaling losses can be explored in `hpcviewer` using any of its three views.

- *Top-down view.* This view represents the dynamic calling contexts (call paths) in which costs were incurred.
- *Bottom-up view.* This view enables one to look upward along call paths. This view is particularly useful for understanding the performance of software components or procedures that are used in more than one context, such as communication library routines.

- *Flat view.* This view organizes performance measurement data according to the static structure of an application. All costs incurred in *any* calling context by a procedure are aggregated together in the flat view.

`hpcviewer` enables developers to explore top-down, bottom-up, and flat views of CCTs annotated with costs, helping to quickly pinpoint performance bottlenecks. Typically, one begins analyzing an application's scalability and performance using the top-down calling context tree view. Using this view, one can readily see how costs and scalability losses are associated with different calling contexts. If costs or scalability losses are associated with only a few calling contexts, then this view suffices for identifying the bottlenecks. When scalability losses are spread among many calling contexts, e.g., among different invocations of `MPI_Wait`, often it is useful to switch to the bottom-up of the data to see if many losses are due to the same underlying cause. In the bottom-up view, one can sort routines by their exclusive scalability losses and then look upward to see how these losses accumulate from the different calling contexts in which the routine was invoked.

Scaling loss based on excess work is intuitive; perfect scaling corresponds to a excess work value of 0, sublinear scaling yields positive values, and superlinear scaling yields negative values. Typically, CCTs for SPMD programs have similar structure. If CCTs for different executions diverge, using `hpcviewer` to compute and report excess work will highlight these program regions.

Inclusive excess work and exclusive excess work serve as useful measures of scalability associated with nodes in a calling context tree (CCT). By computing both metrics, one can determine whether the application scales well or not at a CCT node and also pinpoint the cause of any lack of scaling. If a node for a function in the CCT has comparable positive values for both inclusive excess work and exclusive excess work, then the loss of scaling is due to computation in the function itself. However, if the inclusive excess work for the function outweighs that accounted for by its exclusive costs, then one should explore the scalability of its callees. To isolate code that is an impediment to scalable performance, one can use the *hot path* button in `hpcviewer` to trace a path down through the CCT to see where the cost is incurred.

CHAPTER THIRTEEN

KNOWN ISSUES

This section lists some known issues and potential workarounds. Other known issues can be seen in the project's Gitlab issues pages:

- For HPCToolkit, see https://gitlab.com/HPCToolkit/HPCToolkit/issues
- For hpcviewer, see https://gitlab.com/HPCToolkit/HPCViewer/issues

13.1 Limited cross-version compatibility with ROCm

While a build of HPCToolkit with ROCm 7.0+ has proven to be compatible with ROCm 6.3+, ROCm 6.2 requires its own build of HPCToolkit 2025.1. ROCm versions prior to 6.2 don't support AMD's Rocprofiler-sdk used in HPCToolkit 2025.1+ and require an earlier release of HPCToolkit.

13.2 No support for PC sampling with CUDA 13+

In CUDA 13, NVIDIA removed a deprecated PC sampling API used by HPCToolkit for instruction-level performance measurement. While HPCToolkit can be used with CUDA 13+, HPCToolkit does not yet support PC sampling for CUDA 13+. If you want to use HPCToolkit to perform instruction-level performance measurement using PC sampling on NVIDIA GPUs, we recommend using CUDA 12.

13.3 When monitoring applications that use ROCm, using LD_AUDIT in `hpcrun` may cause it to fail to elide OpenMP runtime frames

Description:

When an application provides a runtime that supports the OpenMP tools API known as OMPT, normally in the OpenMP runtime frames between user code on call stacks are elided. However, we have observed that when using Glibc's `LD_AUDIT` as part of HPCToolkit's measurement infrastructure in conjunction with ROCm's Rocprofiler and Roctracer, an application's TLS storage is incorrectly reinitialized during HPCToolkit's initialization; this clears some important HPCToolkit state information from thread local variables. As a result, the primary thread is not recognized as an OpenMP thread, which is necessary to elide runtime frames.

This bug was reported to Red Hat (https://sourceware.org/bugzilla/show_bug.cgi?id=31717) and fixed in Glibc 2.41, which is considerably newer than the Glibc on almost all installed systems.

Workaround:

Use the `--disable-auditor` option to `hpcrun`.

13.4 When using instrumentation on Intel GPUs, hpcrun may report that substantial time is spent in a partial call path consisting of only an unknown procedure

Description:
Binary instrumentation on Intel GPUs uses Intel's GTPin. GTPin runs in its own private namespace. Asynchronous samples collected in response to Linux timer or hardware counter events may often occur when GTPin is executing. With GTPin in a private namespace, its code and symbols are invisible to `hpcrun`, which causes a degenerate unwind consisting of only an unknown procedure.

Workaround:
Don't collect Linux timer or hardware counter events on the CPU when using binary instrumentation to collect instruction-level performance measurements of kernels executing on Intel GPUs.

13.5 hpcrun reports partial call paths for code executed by a constructor prior to entering main

Description:
At present, all samples of code executed by constructors are reported as a partial call paths even if they are full unwinds. This occurs because HPCToolkit wasn't designed to attribute code that executes in constructors.

Workaround:
Don't be concerned by partial call paths that unwind through `__libc_start_main` and `__lib_csu_init`. The samples are fully attributed even though HPCToolkit does not recognize them as such.

Development Plan:
A future version of HPCToolkit will recognize that these unwinds are indeed full call paths and attribute them as such.

13.6 hpcrun may fail to measure a program execution on a CPU with hardware performance counters

Description:
We observed a problem using Linux `perf_events` to measure CPU performance using hardware performance counters on an `x86_64` cluster at Sandia. An investigation determined that the cluster was running Sandia's LDMS (Lightweight Distributed Metric Service)—a low-overhead, low-latency framework for collecting, transferring, and storing metric data on a large distributed computer system. On this cluster, the LDMS daemon had been configured to use the `syspapi_sampler` (https://github.com/ovis-hpc/ovis/blob/OVIS-4/ldms/src/sampler/syspapi/syspapi_sampler.c), which uses the Linux `perf_events` subsystem to measure hardware counters at the node level. At present, the LDMS `syspapi_sampler`'s use of the Linux `perf_events` subsystem for data collection at the node level conflicts with native use of use the Linux `perf_events` subsystem by HPCToolkit for process-level measurement.

Workaround:
Surprisingly, measurement using HPCToolkit's PAPI interface atop Linux `perf_events` works even though using HPCToolkit directly atop Linux `perf_events` yields no measurement data. For instance, rather than measuring `cycles` using Linux `perf_events` directly with `-e cycles`, one can measure cycles through HPCToolkit's PAPI measurement subsystem using `-e PAPI_TOT_CYC`. Of course, one can configure PAPI to measure other hardware events, such as graduated instructions and cache misses.

Development Plan:
Identify why the use of the Linux `perf_events` subsystem by the LDMS `syspapi_sampler` conflicts with the

use of the direct use of Linux `perf_events` HPCToolkit and the Linux `perf` tool but not with the use of Linux `perf_events` by PAPI.

13.7 hpcrun may associate several profiles and traces with rank 0, thread 0

Description:

On HPE systems, we have observed that `hpcrun` associates several profiles and traces with rank 0, thread 0. This results from the fact that a PMI daemon gets forked from the application in a constructor and there is no exec. Initially, each process gets tagged with rank 0, thread 0 until the real rank and thread is determined later in the execution. That determination never happens for the PMI daemon.

Workaround:

In our experience, the hpcrun files in the measurement for the daemon tagged with rank 0 thread 0 are very small. In experiments we ran, they were about 2K. You can remove these profiles and their matching trace files before processing a measurement database with `hpcprof`. The correspondence between a profile and trace can be determined because they only differ in their suffix (hpcrun or hpctrace).

13.8 `hpcrun` sometimes enables writing of read-only data in memory

If an application or shared library contains a `PT_GNU_RELRO` segment in its program header, the runtime loader `ld.so` will mark all data in that segment readonly after relocations have been processed at runtime. As described in Section *6.1.1.1* of the manual, on `x86_64` and Power architectures, `hpcrun` uses `LD_AUDIT` to monitor operations on dynamic libraries. For `hpcrun` to properly resolve calls to functions in shared libraries, the Global Offset Table (GOT) must be writable. Sometimes, the GOT lies within the `PT_GNU_RELRO` segment, which may cause it to be marked readonly after relocations are processed. If `hpcrun` is using `LD_AUDIT` to monitor shared library operations, it will enable write permissions on the `PT_GNU_RELRO` segment during execution. While this makes some data writable that should have read-only permissions, it should not affect the behavior of any program that does not attempt to overwrite data that should have been readonly in its address space.

13.9 A confusing label for GPU theoretical occupancy

Affected architectures:

NVIDIA GPUs

Description:

When analyzing a GPU-accelerated application that employs NVIDIA GPUs, HPCToolkit estimates percent GPU theoretical occupancy as the ratio of active GPU threads divided by the maximum number of GPU threads available. In multi-threaded or multi-rank programs, HPCToolkit reports GPU theoretical occupancy with the label

```
Sum over rank/thread of exclusive 'GPU kernel: theoretical occupancy 100*(WARP_ACT/
↪WARP_AVL)'
```

rather than its correct label

```
GPU kernel: theoretical occupancy 100*(WARP_ACT/WARP_AVL)
```

The metric is computed correctly by summing the fine-grain parallelism used in each kernel launch across all threads and ranks and dividing it by the sum of the maximum fine-grain parallelism available to each kernel launch across all threads and ranks, and presenting the value as a percent.

Explanation:

This metric is unlike others computed by HPCToolkit. Rather than being computed by `hpcprof`, it is computed by having `hpcviewer` interpret a formula.

Workaround:

Pay attention to the metric value, which is computed correctly and ignore its awkward label.

Development Plan:

Add additional support to `hpcrun` and `hpcprof` to understand how derived metrics are computed and avoid spoiling their labels.

CHAPTER

FOURTEEN

FAQ AND TROUBLESHOOTING

14.1 General Measurement Failures

14.1.1 Profiling `setuid` programs

`hpcrun` uses preloaded shared libraries to initiate profiling. For this reason, it cannot be used to profile `setuid` programs.

14.1.2 Problems loading dynamic libraries

On most platforms, `hpcrun` uses Glibc's `LD_AUDIT` subsystem to monitor an application's use of dynamic libraries. Use of `LD_AUDIT` is needed to properly track loaded libraries when a `RUNPATH` is set in the application or libraries. Due to design flaws in Glibc's implementation, `hpcrun`'s use of `LD_AUDIT` may cause an application to crash. See Section *6.1.1.1* for details about problems caused by `hpcrun`'s use of `LD_AUDIT` and how to avoid them.

14.1.3 Problems using allocator replacement libraries

On most platforms, `hpcrun` uses Glibc's `LD_AUDIT` subsystem to monitor an application's use of dynamic libraries. Use of `LD_AUDIT` is needed to properly track loaded libraries when a `RUNPATH` is set in the application or libraries. However, bugs in older Glibc implementations may cause an application to crash when measured with `hpcrun` if the application uses an allocator replacement library, e.g., `jemalloc`, and `hpcrun`'s `LD_AUDIT` support is enabled. Crashes have been observed with Glibc versions 2.28 and earlier when using an allocator replacement library and `LD_AUDIT`. `LD_AUDIT` in Glibc 2.42 didn't introduce an allocation-related SEGV when monitoring an application that uses a `jemalloc` allocator replacement library. `hpcrun`'s `LD_AUDIT` support hasn't been exhaustively tested against Glibc versions between 2.29 and 2.42 and a variety of allocator replacement libraries. If you encounter crashes with an allocator replacement library and `hpcrun`'s use of `LD_AUDIT`, we recommend using `hpcrun`'s `--disable-auditor` option to avoid the crash. See Section *6.1.1.1* for additional details about problems caused by `hpcrun`'s use of `LD_AUDIT` and how to avoid them.

14.1.4 Problems caused by `gprof` instrumentation

When an application has been compiled with the compiler flag `-pg`, the compiler adds instrumentation to collect performance measurement data for the `gprof` profiler. Measuring application performance with HPCToolkit's measurement subsystem and `gprof` instrumentation active in the same execution may cause the execution to abort. One can detect the presence of `gprof` instrumentation in an application by the presence of the `__monstartup` and `_mcleanup` symbols in a executable. You can recompile your code without the `-pg` compiler flag and measure again. Alternatively, you can use the `--disable-gprof` argument to `hpcrun` to disable `gprof` instrumentation while measuring performance with HPCToolkit.

To cope with `gprof` instrumentation in dynamically-linked programs, you can use `hpcrun`'s `--disable-gprof` option.

14.2 Measurement Failures using NVIDIA GPUs

14.2.1 Deadlock while monitoring a program that uses IBM Spectrum MPI and NVIDIA GPUs

IBM's Spectrum MPI uses a special library `libpami_cudahook.so` to intercept allocations of GPU memory so that Spectrum MPI knows when data is allocated on an NVIDIA GPU. Unfortunately, the mechanism used by Spectrum MPI to do so (wrapping `dlsym`) interferes with performance tools that use `dlopen` and `dlsym`. This interference causes measurement of a GPU-accelerated MPI application using HPCToolkit to deadlock when an application uses both Spectrum MPI and CUDA on an NVIDIA GPU while not using `hpcrun`'s `LD_AUDIT` support. `LD_AUDIT` support is typically enabled, although on some platforms (e.g. Aurora), it is not. If `LD_AUDIT` is disabled, it can be enabled using `--enable-auditor`.

If `LD_AUDIT` cannot be used, e.g. because an application uses `dlmopen` which causes `LD_AUDIT` to fail prior to glibc 2.35, when launching a program that uses Spectrum MPI with `jsrun`, one can use `--smpiargs="-x PAMI_DISABLE_CUDA_HOOK=1 -disable_gpu_hooks"` to disable the PAMI CUDA hook library. These flags cannot be used with the `-gpu` flag.

Note however that disabling Spectrum MPI's CUDA hook will cause trouble if CUDA device memory is passed into the MPI library as a send or receive buffer. An additional restriction is that memory obtained with a call to `cudaMallocHost` may not be passed as a send or receive buffer. Functionally similar memory may be obtained with any host allocation function followed by a call the `cudaHostRegister`.

14.2.2 Ensuring permission to use GPU performance counters

Your Administrator or a recent NVIDIA driver installation may have disabled access to GPU Performance due to Security Notice: NVIDIA Response to "Rendered Insecure: GPU Side Channel Attacks are Practical" https://nvidia.custhelp.com/app/answers/detail/a_id/4738 - November 2018. If that is the case, HPCToolkit cannot access NVIDIA GPU performance counters when using a Linux 418.43 or later driver. This may cause an error message when you try to use PC sampling on an NVIDIA GPU.

A good way to check whether GPU performance counters are available to non-root users on Linux is to execute the following commands:

1. `cd /etc/modprobe.d`
2. `grep NVreg_RestrictProfilingToAdminUsers *`

Generally, if non-root user access to GPU performance counters is enabled, the `grep` command above should yield a line that contains `NVreg_RestrictProfilingToAdminUsers=0`. Note: if you are on a cluster, access to GPU performance counters may be disabled on a login node, but enabled on a compute node. You should run an interactive job on a compute node and perform the checks there.

If access to GPU hardware performance counters is not enabled, one option you have is to use `hpcrun` without PC sampling, i.e., with the `-e gpu=nvidia` option instead of `-e gpu=nvidia,pc`.

If PC sampling is a must, you have two options:

1. Run the tool or application being profiled with administrative privileges. On Linux, launch HPCToolkit with `sudo` or as a user with the `CAP_SYS_ADMIN` capability set.
2. Have a system administrator enable access to the NVIDIA performance counters using the instructions on the following web page: https://developer.nvidia.com/ERR_NVGPUCTRPERM.

14.2.3 Avoiding the error `cudaErrorUnknown`

When monitoring a CUDA application with `REALTIME` or `CPUTIME`, you may encounter a `cudaErrorUnknown` return from many or all CUDA calls in the application.[18] This error may occur non-deterministically. We have observed that this error occurs regularly at very fast periods such as `REALTIME@100`. If this occurs, we recommend using `CYCLES` as a working alternative similar to `CPUTIME`, see Section *13.3.1* for more detail on HPCToolkit's `perf_events` support.

14.2.4 Avoiding the error `CUPTI_ERROR_NOT_INITIALIZED`

`hpcrun` uses NVIDIA's CUDA Performance Tools Interface known as CUPTI to monitor computations on NVIDIA GPUs. In our experience, this error occurs when the version of CUPTI used by HPCToolkit is incompatible with the version of CUDA used by your program or CUDA kernel driver installed on your system. You can check the version of the CUDA kernel driver installed on your system using the `nvidia-smi` command. Table 3 *CUDA Application Compatibility Support Matrix* at the following URL https://docs.nvidia.com/cuda/cuda-toolkit-release-notes specifies what versions of the CUDA kernel driver match each version of CUDA and CUPTI. Although the table indicates that some drivers can support newer versions of CUDA than the one that they were designed for, in our experience that does not necessarily mean that the driver will support performance measurement of CUDA programs using a newer CUPTI. We believe that best way to avoid the `CUPTI_ERROR_NOT_INITIALIZED` error is to ensure that (1) HPCToolkit is compiled with the version of CUDA that your installed CUDA kernel driver was designed to support, and (2) your application uses the version of CUDA that matches the one your kernel driver was designed to support or a compatible older version.

14.2.5 Avoiding the error `CUPTI_ERROR_HARDWARE_BUSY`

When trying to use PC sampling to measure computation on an NVIDIA GPU, you may encounter the following error: 'function `cuptiActivityConfigurePCSampling` failed with error `CUPTI_ERROR_HARDWARE_BUSY`'.

For all versions of CUDA to date (through CUDA 11), NVIDIA's CUPTI library only supports PC sampling for only one process per GPU. If multiple MPI ranks in your application run CUDA on the same GPU, you may see this error.

You have two alternatives:

1. Measure the execution in which multiple MPI ranks share a GPU using only `-e gpu=nvidia` without PC sampling.
2. Launch your program so that there is only a single MPI rank per GPU.
    1. `jsrun` advice: if using `-g1` for a resource set, don't use anything other than `-a1`.

14.2.6 Avoiding the error `CUPTI_ERROR_UNKNOWN`

When trying to use PC sampling to measure computation on an NVIDIA GPU, you may encounter the following error: 'function `cuptiActivityEnableContext` failed with error `CUPTI_ERROR_UNKNOWN`'.

For all versions of CUDA to date (through CUDA 11), NVIDIA's CUPTI library only supports PC sampling for only one process per GPU. If multiple MPI ranks in your application run CUDA on the same GPU, you may see this error. You have two alternatives:

1. Measure the execution in which multiple MPI ranks share a GPU using only `-e gpu=nvidia` without PC sampling.
2. Launch your program so that there is only a single MPI rank per GPU.
    1. `jsrun` advice: if using `-g1` for a resource set, don't use anything other than `-a1`.

[18] We have observed this error on ORNL's Summit machine, running Red Hat Enterprise Linux 8.2.

14.3 General Measurement Issues

14.3.1 How do I choose sampling periods?

When using sample sources for hardware counter and software counter events provided by Linux `perf_events`, we recommend that you use frequency-based sampling. The default frequency is 300 samples/second.

Statisticians use samples sizes of approximately 3500 to make accurate projections about the voting preferences of millions of people. In an analogous way, rather than measuring and attributing every action of a program or every runtime event (e.g., a cache miss), sampling-based performance measurement collects "just enough" representative performance data. You can control `hpcrun`'s sampling periods to collect "just enough" representative data even for very long executions and, to a lesser degree, for very short executions.

For reasonable accuracy (+/- 5%), there should be at least 20 samples in each context that is important with respect to performance. Since unimportant contexts are irrelevant to performance, as long as this condition is met (and as long as samples are not correlated, etc.), HPCToolkit's performance data should be accurate enough to guide program tuning.

We typically recommend targeting a frequency of hundreds of samples per second. For very short runs, you may need to collect thousands of samples per second to record an adequate number of samples. For long runs, tens of samples per second may suffice for performance diagnosis.

Choosing sampling periods for some events, such as Linux timers, cycles and instructions, is easy given a target sampling frequency. Choosing sampling periods for other events such as cache misses is harder. In principle, an architectural expert can easily derive reasonable sampling periods by working backwards from (a) a maximum target sampling frequency and (b) hardware resource saturation points. In practice, this may require some experimentation.

See also the `hpcrun` man page.

14.3.2 Why do I see partial unwinds?

Under certain circumstances, HPCToolkit can't fully unwind the call stack to determine the full calling context where a sample event occurred. Most often, this occurs when `hpcrun` tries to unwind through functions in a shared library or executable that has not been compiled with `-g` as one of its options. The `-g` compiler flag can be used in addition to optimization flags. On Power and `x86_64` processors, `hpcrun` can often compensate for the lack of unwind recipes by using binary analysis to compute recipes itself. However, since `hpcrun` lacks binary analysis capabilities for ARM processors, there is a higher likelihood that the lack of a `-g` compiler option for an executable or a shared library will lead to partial unwinds.

One annoying place where partial unwinds are somewhat common on `x86_64` processors is in Intel's MKL family of libraries. A careful examination of Intel's MKL libraries showed that most but not all routines have compiler-generated Frame Descriptor Entries (FDEs) that help tools unwind the call stack. For any routine that lacks an FDE, HPCToolkit tries to compensate using binary analysis. Unfortunately, highly-optimized code in MKL library routines has code features that are difficult to analyze correctly.

There are two ways to deal with this problem:

- Analyze the execution using information from partial unwinds. Often knowing several levels of calling context is enough for analysis without full calling context for sample events.
- Recompile the binary or shared library causing the problem and add `-g` to the list of its compiler options.

14.3.3 Measurement with HPCToolkit has high overhead! Why?

For reasonable sampling periods, we expect `hpcrun`'s overhead percentage to be in the low single digits, e.g., less than 5%. The most common causes for unusually high overhead are the following:

- Your sampling frequency is too high. Recall that the goal is to obtain a representative set of performance data. For this, we typically recommend targeting a frequency of hundreds of samples per second. For very short runs, you may need to try thousands of samples per second. For very long runs, tens of samples per second can be quite reasonable. See also Section *13.4.1*.

- `hpcrun` has a problem unwinding. This causes overhead in two forms. First, `hpcrun` will resort to more expensive unwind heuristics and possibly have to recover from self-generated segmentation faults. Second, when these exceptional behaviors occur, `hpcrun` writes some information to a log file. In the context of a parallel application and overloaded parallel file system, this can perturb the execution significantly. To diagnose this, you can grep the log files in a measurement directory for large counts of "Errant Samples", which appear after the string "errant:".
- You have very long call paths where long is in the hundreds or thousands. On x86-based architectures, try additionally using `hpcrun`'s `RETCNT` event. This has two effects: It causes `hpcrun` to collect function return counts and to memoize common unwind prefixes between samples.
- Currently, on very large runs the process of writing profile data can take a long time. However, because this occurs after the application has finished executing, it is relatively benign overhead. (We plan to address this issue in a future release.)
- At runtime, `hpcrun` analyzes CPU binaries loaded into an application's address space. This analysis occurs when libraries are loaded. Most libraries are loaded at program launch. This analysis might take seconds for your program. For short-running programs, this can lead to high overhead. However, time for this analysis is not actually considered part of the execution time measured by `hpcrun`.

14.3.4 Some of my syscalls return EINTR

When profiling a threaded program, there are times when it is necessary for `hpcrun` to signal another thread to take some action. When this happens, if the thread receiving the signal is blocked in a syscall, the kernel may return EINTR from the syscall. This would happen only in a threaded program and mainly with "slow" syscalls such as `select()`, `poll()` or `sem_wait()`.

14.3.5 My application spends a lot of time in C library functions with names that include `mcount`

If performance measurements with HPCToolkit show that your application is spending a lot of time in C library routines with names that include the string `mcount` (e.g., `mcount`, `_mcount` or `__mcount_internal`), your code has been compiled with the compiler flag `-pg`, which adds instrumentation to collect performance measurement data for the `gprof` profiler. If you are using HPCToolkit to collect performance data, the `gprof` instrumentation is needlessly slowing your application. You can recompile your code without the `-pg` compiler flag and measure again. Alternatively, you can use the `--disable-gprof` argument to `hpcrun` to disable `gprof` instrumentation while measuring performance with HPCToolkit.

14.4 Problems Recovering Loops in NVIDIA GPU binaries

- When using the `--gpucfg yes` option to analyze control flow to recover information about loops in CUDA binaries, `hpcstruct` needs to use NVIDIA's `nvdisasm` tool. It is important to note that `hpcstruct` uses the version of `nvdisasm` that is on your path. When using the `--gpucfg yes` option to recover loops in CUBINs, you can improve `hpcstruct`'s ability to recover loops by having a newer version of `nvdisasm` on your path. Specifically, the version of `nvdisasm` in CUDA 11.2 is much better than `nvdisasm` in CUDA 10.2. It will recover loops for more procedures and faster.
- While NVIDIA has improved the capability and speed of `nvdisasm` in CUDA 11.2, it may still be too slow to be usable on large CUDA binaries. Because of failures we have encountered with `nvdisasm`, `hpcstruct` launches `nvdisasm` once for each procedure in a GPU binary to maximize the information it can extract. With this approach, we have seen `hpcstruct` take over 12 hours to analyze a CUBIN of roughly 800MB with 40K GPU functions. For large CUDA binaries, our advice is to skip the `--gpucfg yes` option at present until we adjust `hpcstruct` launch multiple copies of `nvdisasm` in parallel to reduce analysis time.

14.5 Graphical User Interface Issues

14.5.1 `hpcviewer` fails to launch

`hpcviewer` saves settings from your preferences. Typically, this information is recorded in `$HOME/.hpctoolkit/hpcviewer`. Often, removing this directory and relaunching `hpcviewer` will solve this problem. A future version of `hpcviewer` will tag recorded state with a version number, gracefully fail, and alert you to the mismatch. With this approach, you will have a choice to use a version of `hpcviewer` that matches your saved state or remove the saved state so that you can use a different version of `hpcviewer`.

14.5.2 Fail to run `hpcviewer`: executable launcher was unable to locate its companion shared library

Although this error mostly incurrs on Windows platform, but it can happen in other environment. The cause of this issue is that the permission of one of Eclipse launcher library (org.eclipse.equinox.launcher.*) is too restricted. To fix this, set the permission of the library to 0755, and launch again the viewer.

14.5.3 Launching `hpcviewer` is very slow on Windows

There is a known issue that Windows Defender significantly slow down Java-based applications. See the github issue at https://github.com/microsoft/java-wdb/issues/9.

A temporary solution is to add `hpcviewer` in the Windows' exclusion list:

1. Open Windows settings.
2. Search for "Virus and threat protection" and open it.
3. Now click on "Manage settings" under "Virus and threat protection settings" section.
4. Now click "Add or remove exclusions" under "Exclusions" section.
5. Now click "Add an exclusion" then select "Folder"
6. Point to `hpcviewer` directory and press "Select Folder"

14.5.4 When executing `hpcviewer`, it complains cannot create "Java Virtual Machine"

If you encounter this problem, we recommend that you edit the `hpcviewer.ini` file which is located in HPCToolkit installation directory to reduce the Java heap size. By default, the content of the file on Linux `x86` for `hpcviewer 2025.02` is as follows:

```
-startup
plugins/org.eclipse.equinox.launcher_1.6.800.v20240513-1750.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.gtk.linux.x86_64_1.2.1000.v20240506-2123
-clearPersistedState
-vmargs
-Xmx8G
-Dosgi.locking=none
-Dslf4j.provider=ch.qos.logback.classic.spi.LogbackServiceProvider
-Dosgi.requiredJavaVersion=17
```

You can decrease the maximum size of the Java heap from 8G to 2GB by changing the `Xmx` specification in the `hpcviewer.ini` file as follows:

```
-Xmx2GB
```

14.5.5 `hpcviewer` fails to launch due to `java.lang.NoSuchMethodError` exception.

The root cause of the error is due to a mix of old and new hpcviewer binaries. To solve this problem, you need to remove your hpcviewer workspace (usually in your `$HOME/.hpctoolkit/hpcviewer` directory), and run `hpcviewer` again.

14.5.6 `hpcviewer` fails due to `java.lang.OutOfMemoryError` exception.

If you see this error, the memory footprint that `hpcviewer` needs to store and the metrics for your measured program execution exceeds the maximum size for the Java heap specified at program launch. On Linux, `hpcviewer` accepts a command-line option `--java-heap` that enables you to specify a larger non-default value for the maximum size of the Java heap. Run `hpcviewer --help` for the details of how to use this option.

14.5.7 `hpcviewer` writes a long list of Java error messages to the terminal!

The Eclipse Java framework that serves as the foundation for `hpcviewer` can be somewhat temperamental. If the persistent state maintained by Eclipse for `hpcviewer` gets corrupted, `hpcviewer` may spew a list of errors deep within call chains of the Eclipse framework.

On MacOS and Linux, try removing your `hpcviewer` Eclipse workspace with default location `$HOME/.hpctoolkit/hpcviewer` and run `hpcviewer` again.

14.5.8 `hpcviewer` attributes performance information only to functions and not to source code loops and lines! Why?

Most likely, your application's binary either lacks debugging information or is stripped. A binary's (optional) debugging information includes a line map that is used by profilers and debuggers to map object code to source code. HPCToolkit can profile binaries without debugging information, but without such debugging information it can only map performance information (at best) to functions instead of source code loops and lines.

For this reason, we recommend that you always compile your production applications with optimization *and* with debugging information. The options for doing this vary by compiler. We suggest the following options:

- GNU compilers (`gcc`, `g++`, `gfortran`): `-g`
- IBM compilers (xlc, xlf, xlC): `-g`
- Intel compilers (`icc`, `icpc`, `ifort`): `-g -debug inline_debug_info`.

We generally recommend adding optimization options *after* debugging options — e.g., '`-g -O2`' — to minimize any potential effects of adding debugging information. Also, be careful not to strip the binary as that would remove the debugging information. (Adding debugging information to a binary does not make a program run slower; likewise, stripping a binary does not make a program run faster.)

Please note that at high optimization levels, a compiler may make significant program transformations that do not cleanly map to line numbers in the original source code. Even so, the performance attribution is usually very informative.

14.5.9 `hpcviewer` hangs trying to open a large database! Why?

The most likely problem is that the Java virtual machine is low on memory and thrashing. The memory footprint that `hpcviewer` needs to store and the metrics for your measured program execution is likely near the maximum size for the Java heap specified at program launch.

On Linux, `hpcviewer` accepts a command-line option `--java-heap` that enables you to specify a larger non-default value for the maximum size of the Java heap. Run `hpcviewer --help` for the details of how to use this option.

14.5.10 `hpcviewer` runs glacially slowly! Why?

There are three likely reasons why `hpcviewer` might run slowly. First, you may be running `hpcviewer` on a remote system with low bandwidth, high latency or an otherwise unsatisfactory network connection to your desktop. If any of these conditions are true, `hpcviewer`'s otherwise snappy GUI can become sluggish if not downright unresponsive. The solution is to install `hpcviewer` on your local system, copy the database onto your local system, and run `hpcviewer` locally. We almost always run `hpcviewer` on our local desktops or laptops for this reason.

Second, the HPCToolkit database may be very large, which can cause the Java virtual machine to run short on memory and thrash. The memory footprint that `hpcviewer` needs to store and the metrics for your measured program execution is likely near the maximum size for the Java heap specified at program launch. On Linux, `hpcviewer` accepts a command-line option `--java-heap` that enables you to specify a larger non-default value for the maximum size of the Java heap. Run `hpcviewer --help` for the details of how to use this option.

14.5.11 `hpcviewer` does not show my source code! Why?

Assuming you compiled your application with debugging information (see Issue *12.6.8*), the most common reason that `hpcviewer` does not show source code is that `hpcprof/mpi` could not find it and therefore could not copy it into the HPCToolkit performance database.

An explanation how HPCToolkit finds source files

`hpcprof/mpi` obtains source file names from your application binary's debugging information. If debugging information is unavailable, such as is often the case for system or math libraries, then source files are unknown.

Two things immediately follow from this. First, in most normal situations, there will always be some functions for which source code cannot be found, such as those within system libraries.[19] Second, to ensure that `hpcprof/mpi` has file names for which to search, make sure as much of your application as possible (including libraries) contains debugging information.

If debugging information is available, source files can come in two forms: absolute and relative. `hpcprof/mpi` can find source files under the following conditions:

- If a source file path is absolute and the source file can be found on the file system, then `hpcprof/mpi` will find it.
- If a source file path is relative, `hpcprof/mpi` can only find it if the source file can be found from the current working directory.
- Finally, if a source file path is absolute and cannot be found by its absolute path, `hpcprof/mpi` uses a special search mode. Let the source file path be `p/f`. If the path's base file name `f` is found within a search directory, then that is considered a match. This special search mode accommodates common complexities such as: (1) source file paths that are relative not to your source code tree but to the directory where the source was compiled; (2) source file paths to source code that is later moved; and (3) source file paths that are relative to file system that is no longer mounted.

Note that given a source file path `p/f` (where `p` may be relative or absolute), it may be the case that there are multiple instances of a file's base name `f` within one search directory, e.g., `p_1/f` through `p_n/f`, where `p_i` refers to the `i`th path to `f`. Similarly, with multiple search-directory arguments, `f` may exist within more than one search directory. If this is the case, the source file `p/f` is resolved to the first instance `p'/f` such that `p'` best corresponds to `p`, where instances are ordered by the order of search directories on the command line.

For any functions whose source code is not found (such as functions within system libraries), `hpcviewer` will generate a synopsis that shows the presence of the function and its line extents (if known).

Hypothetically, let's say that your HPCToolkit database is missing source code from PetSC and you linked your program against a copy of PetSC provided by a module provided by your system administrators. You can check if that library contains line map information by running `readelf --debug-dump=decodedline`. In the `readelf` output, if you

[19] Having a system administrator download the associated `devel` package for a library can enable visibility into the source code of system libraries.

see that the source file paths begin with `/path/to/petsc`, then you can download a matching version of PetSC to a location of your choosing `/my/path/to/petsc`. Then, you can rerun `hpcprof/mpi` with the `-R` option, which is used to replace path prefixes when searching for source files. In this example, you would use `-R /path/to/petsc=/my/path/to/petsc` to instruct `hpcprof/mpi` to consider all path prefixes of `/path/to/petsc` as `/my/path/to/petsc` so `hpcprof/mpi` will find the copies of source that you downloaded.

14.5.12 `hpcviewer`'s reported line numbers do not exactly correspond to what I see in my source code! Why?

To use a cliché, "garbage in, garbage out". HPCToolkit depends on information recorded in the symbol table by the compiler. Line numbers for procedures and loops are inferred by looking at the symbol table information recorded for machine instructions identified as being inside the procedure or loop.

For procedures, often no machine instructions are associated with a procedure's declarations. In that case, thae function might be mapped back to the first statement in the function that had machine code associated with it.

Inlined functions may occasionally lead to confusing data for a procedure. Machine instructions mapped to source lines from the inlined function appear in the context of other functions. While `hpcprof`'s methods for handling incline functions are good, some codes can confuse the system.

For loops, the process of identifying what source lines are in a loop is similar to the procedure process: what source lines map to machine instructions inside a loop defined by a backward branch to a loop head. For some compilers, that may cause a loop to be mapped back to its closing brace rather than beginning of the loop.

14.5.13 `hpcviewer` claims that there are several calls to a function within a particular source code scope, but my source code only has one! Why?

In the course of code optimization, compilers often replicate code blocks. For instance, as it generates code, a compiler may peel iterations from a loop or split the iteration space of a loop into two or more loops. In such cases, one call in the source code may be transformed into multiple distinct calls that reside at different code addresses in the executable.

When analyzing applications at the binary level, it is difficult to determine whether two distinct calls to the same function that appear in the machine code were derived from the same call in the source code. Even if both calls map to the same source line, it may be wrong to coalesce them; the source code might contain multiple calls to the same function on the same line. By design, HPCToolkit does not attempt to coalesce distinct calls to the same function because it might be incorrect to do so; instead, it independently reports each call site that appears in the machine code. If the compiler duplicated calls as it replicated code during optimization, multiple call sites may be reported by `hpcviewer` when only one appeared in the source code.

14.5.14 hpcviewer's Trace view shows lots of white space on the left. Why?

At startup, Trace view renders traces for the time interval between the minimum and maximum times recorded for any process or thread in the execution. The minimum time for each process or thread is recorded when its trace file is opened as HPCToolkit's monitoring facilities are initialized at the beginning of its execution. The maximum time for a process or thread is recorded when the process or thread is finalized and its trace file is closed. When an application uses the `hpctoolkit_start` and `hpctoolkit_stop` primitives, the minimum and maximum time recorded for a process/thread are at the beginning and end of its execution, which may be distant from the start/stop interval. This can cause significant white space to appear in Trace view's display to the left and right of the region (or regions) of interest demarcated in an execution by start/stop calls.

14.6 Debugging

14.6.1 How do I debug HPCToolkit's measurement?

Assume you want to debug HPCToolkit's measurement subsystem when collecting measurements for an application named `app`.

14.6.2 Tracing HPCToolkit's Measurement Subsystem

Broadly speaking, there are two levels at which a user can test `hpcrun`. The first level is tracing `hpcrun`'s application control, that is, running `hpcrun` without an asynchronous sample source. The second level is tracing `hpcrun` with a sample source. The key difference between the two is that the former uses the `--event NONE` or `HPCRUN_EVENT_LIST="NONE"` option (shown below) whereas the latter does not (which enables the default CPUTIME sample source). With this in mind, to collect a debug trace for either of these levels, use commands similar to the following:

```
[<mpi-launcher>] \
  hpcrun --monitor-debug --dynamic-debug ALL --event NONE \
    app [app-arguments]
```

Note that the `*debug*` flags are optional. The `--monitor-debug/MONITOR_DEBUG` flag enables `libmonitor` tracing. The `--dynamic-debug/HPCRUN_DEBUG_FLAGS` flag enables `hpcrun` tracing.

14.6.3 Using a debugger to inspect an execution being monitored by HPCToolkit

If HPCToolkit has trouble monitoring an application, you may find it useful to execute an application being monitored by HPCToolkit under the control of a debugger to observe how HPCToolkit's measurement subsystem interacts with the application.

HPCToolkit's measurement subsystem is easiest to debug if you configure and build HPCToolkit for debugging when building by Spack or Meson. See the Spack and Meson sections in this manual for how to configure debugging for your build. Note: if configuring HPCToolkit for debugging using Spack, you probably want to install hpctoolkit with the `--keep-stage` option, which instructs Spack not to remove source code (e.g. a copy of HPCToolkit) after compiling it.

One can debug a dynamically-linked applications being measured by HPCToolkit's measurement subsystem. For a single-process program, you can use `gdb hpcrun`, set breakpoints inside `hpcrun`'s code where you want them, and then launch the application you are measuring with the gdb `run` command.

> **Important**
>
> `hpcrun` launches the program it is measuring with `exec`. As a result, in `gdb` before you issue the `run` command again, you will need to use the `gdb` command `exec-file hpcrun` to tell `gdb` that you want to launch `hpcrun` with the `run` command and not the application launched by `hpcrun` using `exec`. If you forget, you will find that none of the breakpoints in `hpcrun` will be encountered and your application will run to completion unmonitored.

Alternatively, you can launch an application directly with `hpcrun` and pass the `--debug` flag on its command line. Then, from a different terminal, you can attach a debugger to the copy of your application which will be spin waiting for you to attach.

To debug hpcrun with a debugger use the following approach.

1. Launch your application. To debug `hpcrun` without controlling sampling signals, launch normally. To debug `hpcrun` with controlled sampling signals, launch as follows:

   ```
   hpcrun --debug --event REALTIME@0 app [app-arguments]
   ```

2. Attach a debugger. The debugger should be spinning in a loop whose exit is conditioned by the `HPCRUN_DEBUGGER_WAIT` variable.
3. Set any desired breakpoints. To send a sampling signal at a particular point, make sure to stop at that point with a *one-time* or *temporary* breakpoint (`tbreak` in GDB).
4. Call `(void) hpcrun_continue()` or set the `HPCRUN_DEBUGGER_WAIT` variable to 0 and continue.

5. To raise a controlled sampling signal, raise a SIGPROF, e.g., using GDB's command `signal SIGPROF`.

CHAPTER

FIFTEEN

ENVIRONMENT VARIABLES

HPCToolkit's measurement subsystem decides what and how to measure using information it obtains from environment variables. This chapter describes all of the environment variables that control HPCToolkit's measurement subsystem.

When using HPCToolkit's `hpcrun` script to measure the performance of dynamically-linked executables, `hpcrun` takes information passed to it in command-line arguments and communicates it to HPCToolkit's measurement subsystem by appropriately setting environment variables. Measurement of statically-linked programs is no longer supported by HPCToolkit.

Section *14.1* describes environment variables of interest to users. Section *14.3* describes environment variables designed for use by HPCToolkit developers. In some cases, HPCToolkit's developers will ask a user to set some of the environment variables described in Section *14.3* to generate a detailed error report when problems arise.

15.1 Environment Variables for Users

HPCRUN_EVENT_LIST

This environment variable is used provide a set of (event, period) pairs that will be used to configure HPCToolkit's measurement subsystem to perform asynchronous sampling. The HPCRUN_EVENT_LIST environment variable must be set otherwise HPCToolkit's measurement subsystem will terminate execution. If an application should run with sampling disabled, HPCRUN_EVENT_LIST should be set to NONE. Otherwise, HPCToolkit's measurement subsystem expects an event list of the form shown below.

```
event1[@period1];...;eventN[@periodN]
```

As denoted by the square brackets, periods are optional. The default period is 1 million.

Flags to add an event with `hpcrun`: `-e/--event event1@period1`

Multiple events may be specified using multiple instances of `-e/--event` options.

HPCRUN_TRACE

If this environment variable is set, HPCToolkit's measurement subsystem will collect a trace of sample events as part of a measurement database in addition to a profile. HPCToolkit's hpctraceviewer utility can be used to view the trace after the measurement database are processed with either HPCToolkit's hpcprof or hpcprofmpi utilities.

Flags to enable tracing with `hpcrun`: `-t/--trace`

HPCRUN_OUT_PATH

If this environment variable is set, HPCToolkit's measurement subsystem will use the value specified as the name of the directory where output data will be recorded. The default directory for a command *command* running under control of a job launcher with as job ID *jobid* is hpctoolkit-*command*-measurements[-*jobid*]. (If no job ID is available, the portion of the directory name in square brackets will be omitted. Warning: Without a *jobid* or an output option, multiple profiles of the same *command* will be placed in the same output directory.

Flags to set output path with `hpcrun`: `-o/--output <directoryName>`

HPCRUN_PROCESS_FRACTION

If this environment variable is set, HPCToolkit's measurement subsystem will measure only a fraction of an execution's processes. The value of HPCRUN_PROCESS_FRACTION may be written as a a floating point number or as a fraction. So, '0.10' and '1/10' are equivalent. If HPCRUN_PROCESS_FRACTION is set to a value with an unrecognized format, HPCToolkit's measurement subsystem will use the default probability of 0.1. For each process, HPCToolkit's measurement subsystem will generate a pseudo-random value in the range [0.0, 1.0). If the generated random number is less than the value of HPCRUN_PROCESS_FRACTION, then HPCToolkit will collect performance measurements for that process.

Flags to set process fraction with `hpcrun`: `-f/-fp/--process-fraction <frac>`

HPCRUN_DELAY_SAMPLING

If this environment variable is set, HPCToolkit's measurement subsystem will initialize itself but not begin measurement using sampling until the program turns on sampling by calling `hpctoolkit_sampling_start()`. To measure only a part of a program, one can bracket that with `hpctoolkit_sampling_start()` and `hpctoolkit_sampling_stop()`. Sampling may be turned on and off multiple times during an execution, if desired.

Flags to delay sampling with `hpcrun`: `-ds/--delay-sampling`

HPCRUN_CONTROL_KNOBS

`hpcrun` has some settings, known as control knobs, that can be adjusted by a knowledgeable user to tune the operation of `hpcrun`'s measurement subsystem. Names and default values of the control knobs are shown in Table *13.1*

Table 1: Control knob names and default values.

Name	Default Value	Description
`MAX_COMPLETION_CALLBACK_THREADS`	1000	See Note 1.
`STREAMS_PER_TRACING_THREAD`	256	See Note 2.
`HPCRUN_CUDA_DEVICE_BUFFER_SIZE`	8388608	See Note 3.
`HPCRUN_CUDA_DEVICE_SEMAPHORE_SIZE`	65536	See Note 4.

Note 1: OpenCL may execute callbacks on helper threads created by the OpenCL runtime. This knob specifies the maximum number of helper threads that can be handled by `hpcrun`'s OpenCL tracing implementation.

Note 2: GPU stream traces are recorded by tracing threads created by `hpcrun`. Reducing the number of streams per `hpcrun` tracing thread may make monitoring faster, though it will use more resources.

Note 3: Value used as `CUPTI_ACTIVITY_ATTR_DEVICE_BUFFER_SIZE`. See https://docs.nvidia.com/cuda/cupti/group__CUPTI__ACTIVITY__API.html.

Note 4: Value used as `CUPTI_ACTIVITY_ATTR_PROFILING_SEMAPHORE_POOL_SIZE`. See https://docs.nvidia.com/cuda/cupti/group__CUPTI__ACTIVITY__API.html

Flags to set a control knob for `hpcrun`: `-ck/--control-knob` name=setting.

HPCRUN_MEMLEAK_PROB

If this environment variable is set, HPCToolkit's measurement subsystem will measure only a fraction of an execution's memory allocations, e.g., calls to `malloc`, `calloc`, `realloc`, `posix_memalign`, `memalign`, and valloc. All allocations monitored will have their corresponding calls to free monitored as well. The value of HPCRUN_MEMLEAK_PROB may be written as a a floating point number or as a fraction. So, '0.10' and '1/10' are equivalent. If HPCRUN_MEMLEAK_PROB is set to a value with an unrecognized format, HPCToolkit's measurement subsystem will use the default probability of 0.1. For each memory allocation, HPCToolkit's measurement subsystem will generate a pseudo-random value in the range [0.0, 1.0). If the generated random number is less than the value of HPCRUN_MEMLEAK_PROB, then HPCToolkit will monitor that allocation.

Flags to set process fraction with `hpcrun`: `-mp/--memleak-prob <prob>`

HPCRUN_RETAIN_RECURSION
Unless this environment variable is set, by default HPCToolkit's measurement subsystem will summarize call chains from recursive calls at a depth of two. Typically, application developers have no need to see performance attribution at all recursion depths when an application calls recursive procedures such as quicksort. Setting this environment variable may dramatically increase the size of calling context trees for applications that employ bushy subtrees of recursive calls.

Flags to retain recursion with `hpcrun`: `-r/--retain-recursion`

HPCRUN_MEMSIZE
If this environment variable is set, HPCToolkit's measurement subsystem will allocate memory for measurement data in segments using the value specified for HPCRUN_MEMSIZE (rounded up to the nearest enclosing multiple of system page size) as the segment size. The default segment size is 4M.

Flags to set memsize with `hpcrun`: `-ms/--memsize <bytes>`

HPCRUN_LOW_MEMSIZE
If this environment variable is set, HPCToolkit's measurement subsystem will allocate another segment of measurement data when the amount of free space available in the current segment is less than the value specified by HPCRUN_LOW_MEMSIZE. The default for low memory size is 80K.

Flags to set low memsize with `hpcrun`: `-lm/--low-memsize <bytes>`

HPCTOOLKIT_HPCSTRUCT_CACHE
If this environment variable contains the name of a Linux directory that is readable and writable to you, `hpcstruct` will cache any program structure files it computes in this directory. When invoked to analyze a binary, `hpcstruct` will check if program structure information for the binary exists in the cache. If so, `hpcstruct` will return the cached copy. If not, `hpcstruct` will compute program structure information for the binary and record it in the cache.

15.2 Environment Variables that May Avoid a Crash

HPCRUN_AUDIT_FAKE_AUDITOR
By default, `hpcrun` will use `libc`'s `LD_AUDIT` feature to monitor dynamic library operations. For cases where using `LD_AUDIT` is problematic (e.g. with applications or libraries that require the use of `dlmopen`) , `hpcrun` supports an alternative *fake auditor* that monitors shared library operations by wrapping `dlopen` and `dlclose` instead. This variable will be set to 1 if a *fake auditor* is used. If `LD_AUDIT` is not causing your program to crash, we don't recommend using the fake auditor as it may cause your application or shared libraries it loads to ignore any `RUNPATH` set in their binaries.

Flag to select the fake auditor with `hpcrun`: `--disable-auditor`.

HPCRUN_AUDIT_DISABLE_PLT_CALL_OPT
By default, `hpcrun` will use `libc`'s `LD_AUDIT` feature to monitor dynamic library operations. The `LD_AUDIT` facility has the unfortunate behavior of intercepting each call to a shared library. Each call to a shared library is dispatched through the *Procedure Linkage Table* (PLT). We have observed that allowing the `LD_AUDIT` facility to intercept each call to a shared library is costly: on `x86_64` we measured a slowdown of 68x for a call to an empty shared library routine.

To avoid this overhead, `hpcrun` sidesteps `LD_AUDIT`'s monitoring of a load module's calls to a shared library routine by allowing the address of the routine to be cached in the load module's Global Offset Table (GOT). The mechanism for this optimization is complex. If you suspect that this optimization is causing your program to crash, this optimization can be disabled. If your program is not crashing, don't even consider adjusting this!

Flag to disable optimization of PLT calls when using `LD_AUDIT` to monitor shared library operations with `hpcrun`: `--disable-auditor-got-rewriting`.

15.3 Environment Variables for Developers

HPCRUN_WAIT

If this environment variable is set, HPCToolkit's measurement subsystem will spin wait for a user to attach a debugger. After attaching a debugger, a user can set breakpoints or watchpoints in the user program or HPCToolkit's measurement subsystem before continuing execution. To continue after attaching a debugger, use the debugger to set the program variable DEBUGGER_WAIT=0 and then continue. Note: Setting HPCRUN_WAIT can only be cleared by a debugger if HPCToolkit has been built with debugging symbols. Building HPCToolkit with debugging symbols requires configuring HPCToolkit with –enable-develop.

HPCRUN_DEBUG_FLAGS

HPCToolkit supports a multitude of debugging flags that enable a developer to log information about HPCToolkit's measurement subsystem as it records sample events. If HPCRUN_DEBUG_FLAGS is set, this environment variable is expected to contain a list of tokens separated by a space, comma, or semicolon. If a token is the name of a debugging flag, the flag will be enabled, it will cause HPCToolkit's measurement subsystem to log messages guarded with that flag as an application executes. The complete list of dynamic debugging flags can be found in HPCToolkit's source code in the file src/tool/hpcrun/messages/messages.flag-defns. A special flag value "ALL" enables all flags. Note: not all debugging flags are meaningful on all architectures.

Caution: turning on debugging flags will typically result in voluminous log messages, which will typically will dramatically slow measurement of the execution under study.

Flags to set debug flags with `hpcrun`: `-dd/--dynamic-debug <flag>`

HPCRUN_ABORT_TIMEOUT

If an execution hangs when profiled with HPCToolkit's measurement subsystem, the environment variable HPCRUN_ABORT_TIMEOUT can be used to specify the number of seconds that an application should be allowed to execute. After executing for the number of seconds specified in HPCRUN_ABORT_TIMEOUT, HPCToolkit's measurement subsystem will forcibly terminate the execution and record a core dump (assuming that core dumps are enabled) to aid in debugging.

Caution: for a large-scale parallel execution, this might cause a core dump for each process, depending upon the settings for your system. Be careful!

CHAPTER SIXTEEN

GETTING HELP

The HPCToolkit project team operates the 'HPCToolkit Users' Discord server. Discord offers a Slack-like experience for interacting developers and other users. We welcome questions, problem reports, or feature requests.

New members: Join HPCToolkit Users Discord server

Current members: Access HPCToolkit Users Discord server

CHAPTER

SEVENTEEN

INFORMATION FOR EXPERTS

17.1 Hardware Counter Event Names

HPCToolkit uses libpfm4 (Libpfm4 2008) to translate from an event name string to an event code recognized by the kernel. An event name is case insensitive and is defined as followed:

```
[pmu::][event_name][:unit_mask][:modifier|:modifier=val]
```

- **pmu**. Optional name of the PMU (group of events) to which the event belongs to. This is useful to disambiguate events in case events from difference sources have the same name. If no pmu is specified, the first match event is used.
- **event_name**. The name of the event. It must be the complete name, partial matches are not accepted.
- **unit_mask**. Some events can be refined using sub-events. A `unit_mask` designates an optional sub-event. An event may have multiple unit masks and it is possible to combine them (for some events) by repeating `:unit_mask` pattern.
- **modifier**. A modifier is an optional filter that restricts when an event counts. The form of a modifier may be either `:modifier` or `:modifier=val`. For modifiers without a value, the presence of the modifier is interpreted as a restriction. Events may allow use of multiple modifiers at the same time.
  - **hardware event modifiers**. Some hardware events support one or more modifiers that restrict counting to a subset of events. For instance, on an Intel Broadwell EP, one can add a modifier to `MEM_LOAD_UOPS_RETIRED` to count only load operations that are an `L2_HIT` or an `L2_MISS`. For information about all modifiers for hardware events, one can direct HPCToolkit's measurement subsystem to list all native events and their modifiers as described in Section *6.1*.
  - **precise_ip**. For some events, it is possible to control the amount of skid. Skid is a measure of how many instructions may execute between an event and the PC where the event is reported. Smaller skid enables more accurate attribution of events to instructions. Without a skid modifier, hpcrun allows arbitrary skid because some architectures don't support anything more precise. One may optionally specify one of the following as a skid modifier:
    * `:p` : a sample must have constant skid.
    * `:pp` : a sample is requested to have 0 skid.
    * `:ppp` : a sample must have 0 skid.
    * `:P` : autodetect the least skid possible.

    NOTE: If the kernel or the hardware does not support the specified value of the skid, no error message will be reported but no samples will be recorded.

17.1.1 Examples

Assume running hpcrun with the following arguments:

```
$ hpcrun -e UOPS_RETIRED:MS:c=1:e=1:pp  my_program
```

The above command will run `my_program` with the `UOPS_RETIRED:MS` event enabled with precise IP and no skid. The hardware profiling event `UOPS_RETIRED:MS:c=1:e=1` counts the number of times a block of code stalls or falls back to the processor's Microcode Sequencer (MSROM) to execute complex instructions or handle performance faults (assists).

Event Breakdown

- `UOPS_RETIRED`: Tracks the micro-operations (uops) that successfully complete execution. Modern Intel processors break down complex software instructions into simpler, hardware-level uops.
- `:MS`: Filters the uops specifically to those issued by the Microcode Sequencer.
- `:c=1`: A condition (or duration threshold) that counts the number of cycles where at least one MS uop was retired.
- `:e=1`: An edge trigger that counts the actual occurrences or transitions of this condition, preventing a continuous stream of MS uops from being counted as a single prolonged event.
- `:pp`: Request precise IP and no skid.