Research Project Report: Mechanisms for observing and monitoring executions of GPU accelerated programs

Dejan Grubisic

Department of Computer Science, Rice University, Houston, TX

Abstract

Graphical Processing Units (GPUs) provide over 95% of the compute performance of GPU-accelerated supercomputers such as the Summit system at Oak Ridge National Laboratory. Forthcoming supercomputers such as Frontier and Aurora will also be GPU accelerated, able to execute over a quintillion (10^{18}) instructions per second. On such architectures, programs need to offload nearly all of their computation onto GPUs for top performance. That is an unfamiliar process for application developers and supercomputers could end up largely underutilized since optimization of applications for such a large scale presents an extremely hard task. To address this challenge, performance tools are used to collect execution-based data, associate them to code origin and find out performance bottlenecks. In this paper, we describe newly added features to the HPCToolkit performance tool that includes support for OpenCL, scalable tracing, GPU hardware performance counters, and GPU idleness analysis. Development of such features for performance tools requires deep integration with vendor runtime environments, tool frameworks, programming models, and applications. How this interaction should be done, and what standard should be followed is not uniquely defined. Here, we advocate for creating a transparent set of invariants for each piece of the infrastructure, to make development easier and robust. As a motivation, we describe the set of problems and solutions we faced during development, to identify requirements and necessary APIs needed from performance tools.

1. Introduction

Computation has become an integral tool for science, medicine engineering, and national security. Each of these fields relies strongly on scientific simulations and the processing of an enormous amount of data to model complex processes from supernovae explosion to drug discovery. As computer systems become large and more powerful, they enabled researchers to create more detailed models and run larger experiments, which led to more breakthroughs in each of these fields.

Demands for extreme-scale computations have motivated GPU-accelerated computing because it can deliver significantly higher node performance, which makes systems cost-effective. The current fastest supercomputer in the US - Summit, has 6 GPUs on each node and performs more than 95% of computations on them. According to the reports from the Exascale Computing Project (ECP), [1], forthcoming exascale supercomputers will all be accelerated with GPUs as well. These systems will have several thousand nodes that will be capable of executing a quintillion (10^{18}) instructions per second; however they could end up underutilized since optimization of complex applications on large-scale is hard task.

Large-scale applications that run on supercomputers usually use interprocess communication, shared memory access, atomic primitives, concurrent data structures, GPU offloading, and synchronization which introduces a high level of complexity to program optimization. To optimize such applications, new programming models arose that enable some level of tuning and compiler directives. For multiprocessing, OpenMP uses pragmas to guide parallelization across multiple threads. For GPU programming CUDA [2] and OpenCL [3] have parameters for kernel size, while higher-level programming abstractions such as Kokkos [4], Raja [5] and SYCL [6] have parallel_for, reduction and other primitives with optimized implementations for commonly used patterns. Nevertheless, to write an optimized code programmers need to measure its performance, understand what caused performance problems, and how to fix them. This is done with performance tools.

Performance tools guide the optimization of user applications by monitoring the execution of the program and presenting performance data. Some performance tools like *GPU Advisor* [7] additionally, provide a set of optimizations and an estimate of their speedup. The 2020 report from the Department of Energy (DOE) [8] describes the state of the art tools and frameworks for optimizing large scale applications. To monitor the execution, performance tools use sampling and instrumentation to collect performance data and attribute them to their code origin.

Sampling is achieved by interrupting the execution and collecting data about the program counter, which tells us where we interrupted the program. Additionally, we can read hardware counters that can be set to count some predefined system events such as cache misses or data movement. By collecting this information we can reason about program behavior in terms of where is the program spending the most of the time, and what could be the reason for that. Sampling-based techniques usually introduce small overhead, and since we can dictate the

frequency of sampling it is possible to tune the ratio between precision and overhead.

Instrumentation-based techniques, on the other hand, provide a callback mechanism or counter increment every time a particular function is being called. Once we take over the control, we can collect time or some other system metrics which we can attribute to the given function. In contrast to sampling techniques, instrumentation enables us to collect data completely focused on the functions that we are interested in. Since we usually want to monitor the functions that introduce performance bottlenecks, and we cannot control the frequency of function calls, instrumentation generally adds get higher overhead.

As we collect metrics with sampling or instrumentation, we need to find a way to attribute recorded values to the source code. Modern profiling tools such as [9, 10, 11, 12, 13, 14, 15, 16] associate metrics directly to the function names, but often it is hard to understand why the bottleneck function is called too often and what created performance problem. To solve this problem, HPCToolkit [17] uses callstack unwinding to recover the chain of function calls for every sample. This way it is possible to locate hot paths which give us a better understanding of why the performance problem happened in the first place.

Besides associating performance metrics to the source code which is referred to as profile view, it is beneficial to order samples in timeline fashion which is known as trace view. Trace view provides a convenient visual representation that shows, each thread on the y-axis and their activity for each point in time on the x-axis. These activities could include not only what function executed at a certain time interval, but also arithmetic unit utilization, memory allocation, or any other system-wide metric. With this information, it is easy to see performance problems such as load imbalance, memory leakage, or power consumption. For applications accelerated with GPUs, trace view shows valuable information about CPU-GPU interaction, uncovers synchronization problems, and underutilization problems.

To work properly performance tools face the following challenges:

- Deciding what functions to watch
- Wrapping functions of interest
- Relying on incompletely documented vendor APIs
- Precisely attributing metrics
- Interaction between application, tool, and runtime
- Minimizing overhead and disturbance to application
- Supporting distributed execution

In this paper, we describe the development of performance tools and describe the interaction between performance tools and vendor infrastructure. Since each has their invariants which can be in conflict, careful engineering and coordination between teams are necessary to make performance monitoring work.

This paper is organized as follows. Section 2 describes APIs for collecting performance metrics on Nvidia, AMD and Intel GPUs. Section 3 describes the high-level structure of HPCToolkit, a performance tool capable of recording GPU metrics. Section 4 discusses the implementation of HPCToolkit GPU tracing infrastructure and elaborates on newly added features. Section 5 introduces the GPU blame-shifting approach, in trace analysis. Section 6 describes problems and solutions arising from an interaction between performance tools and vendor infrastructures and elaborates on invariants that need to be maintained to make performance tools work. Finally, section 7 summarizes the contribution of this research project and describes lessons learned during the development process.

2. Performance Measurement Infrastructure on GPUs

In recent years, vendors have developed proprietary APIs for monitoring their GPU activities. In this section, we describe performance APIs for tracing Nvidia, AMD, and Intel GPUs, and best practices for their use. Besides proprietary APIs, there are also vendor-independent tools such as PAPI, that provide unique APIs for collecting metrics from performance counters on all mentioned GPUs.

2.1. Nvidia

Nvidia developed the CUPTI API [18] which provides an instrumentation layer through subscriber callback, which intercepts every kernel launch on GPU. Besides this, CUPTI provides a completion callback, that reports GPU activities by sending them to the CPU in a batch fashion. Once records are delivered to the CPU, the user can process collected data and attribute them to code. As one more special feature, Nvidia added fine-grained measurement on GPU by using Program Counter(PC) sampling that enables collection of stall reason of threads on GPU.

2.2. AMD

AMD developed Rocm API that resembles Nvidia CUPTI API with limited capabilities. At this point, Rocm consists of ROCProfiler and ROCTracer libraries. ROCProfiler library provides the methods to open/close profiling context, to start, stop and read HW performance counters and traces, to intercept kernel dispatches to collect per-kernel profiling data [19]. ROCTracer provides subscriber and completion callback functionalities described above, while PC sampling is under development.

2.3. Intel - OpenCl

Intel uses Level Zero API [20] and GT-Pin [21] to provide callback API to instrument kernel launch and collect fine-grain instruction-level measurement. Level Zero is designed to support direct access to the lowest level metric for any accelerator type such as GPU, FPGAs, and TPUs. Level zero API should provide APIs for tracing, collecting performance metrics, program instrumentation, and program debugging.

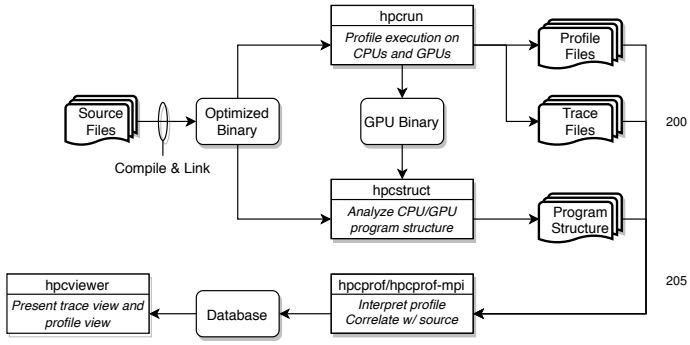


Figure 1: HPCToolkit's workflow for analysis of GPU-accelerated applications.

2.4. The PAPI

Performance Application Programming Interface (PAPI) [22] is the University of Tennessee's vendor-independent API for collecting the metrics from performance counters across the entire computing system (eg. CPUs, GPUs, on/off-chip memory, interconnects, I/O system, energy/power). For CPUs, PAPI returns a callback when a certain threshold is reached for the specified CPU event or it can use start/stop commands to measure the value of performance counters for code that executes between two.

For GPU, PAPI first groups events to event sets that are measured in parallel on the GPU. Conveniently, we can use again the same start/stop functionality to turn on the monitoring around kernels we are interested in. Additionally, PAPI can use read functionalities to access the current values of the counters.

On Nvidia GPUs PAPI collects per kernel metrics, enforcing serialization. On AMD GPUs PAPI can read asynchronously counter values for the GPU for a particular point in time and if we want to get per kernel metric, we need to enforce serialization ourselves. On Intel GPUs, both kernel-mode and time-mode are supported independently.

3. HPCToolkit Overview

In this section, we provide a brief overview of the high-level infrastructure for Rice University's HPCToolkit, which is described in detail in the manuscript under review [23]. Figure 1 shows HPCToolkit's workflow to analyze programs running on GPUs. HPCToolkit's *hpcrun* measurement tool collects GPU performance metrics using profiling APIs from GPU vendors or custom hooks with `LD_PRELOAD`.

hpcrun can measure programs that employ one or more GPU programming models, including OpenMP [24], OpenACC [25], CUDA [2], HIP [26], OpenCL [27], and DPC++ [28]. As GPU binaries are loaded into memory, *hpcrun* records them for later analysis. For GPUs that provide APIs for fine-grained measurement, *hpcrun* can collect instruction-level characterizations of GPU kernels using hardware support for sampling or binary instrumentation. *hpcrun*'s output includes profiles and optionally traces. Each profile contains a calling context tree in which each node is associated with a set of metrics. Each trace file contains a sequence of events on a CPU thread or a GPU stream with their timestamps.

hpcstruct analyzes CPU and GPU binaries to recover static information about procedures, inlined functions, loop nests, and source lines. There are two aspects to this analysis: (1) recovering information about line mappings and inlining from compiler-recorded information in binaries, and (2) analyzing machine code to recover information about loops.

hpcprof and *hpcprof-mpi* correlate performance metrics for GPU-accelerated code with program structure. *hpcprof* employs a multithreaded streaming aggregation algorithm to quickly aggregate profiles, reconstruct a global calling context tree, and relate measurements associated with machine instructions back to CPU and GPU source code. To accelerate the analysis of performance data from extreme-scale executions, *hpcprof-mpi* additionally employs distributed-memory parallelism for greater scalability. Both *hpcprof* and *hpcprof-mpi* write sparse representations of their analysis results in a database.

Finally, *hpcviewer* interprets and visualizes the database. In its profile view, *hpcviewer* presents a heterogenous 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. In its trace view, *hpcviewer* identifies each CPU or GPU trace line with a tuple of metadata about the hardware (e.g., node, core, GPU) and software constructs (e.g., rank, thread, GPU stream) associated with the trace. Automated analysis of traces can attribute GPU idleness to CPU code.

4. GPU Tracing Infrastructure

In HPCToolkit the process of collecting GPU data could be described in three steps:

1. Intercepting GPU invocation launch and unwinding CPU callstack
2. Collecting and processing GPU performance metrics associated with invocation
3. Attribute GPU metrics to invocation CPU callstack

These three steps are achieved with the interaction between application threads, monitor thread, and tracing threads shown in Figure 2. Threads exchange data through bidirectional single producer single consumer, wait-free, channels described in [29]. Application threads are responsible for unwinding CPU callstack and attribution of GPU metrics to it (steps 1, 3). The monitor thread is responsible for getting GPU activities from profiling APIs, processing obtained data and sending data to the application, and tracing threads for recording.

When an application thread performs an invocation $\mathcal{I}$ of a GPU operation (e.g., a kernel or a data copy), *hpcrun* gets subscriber callback before and after the invocation launch on GPU.

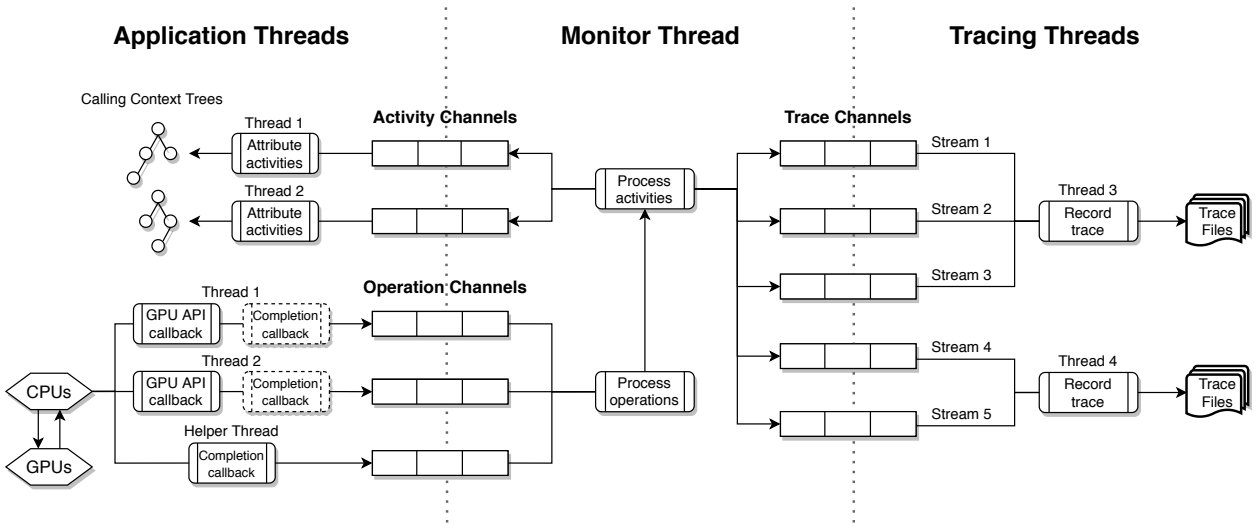


Figure 2: HPCToolkit's infrastructure for coordinating application threads, monitor thread, and tracing threads.

The operations in subscriber callback consist of the following actions:

- Updating calling context tree with GPU activity record from the previous invocation (step 3)
- Unwinding the application thread's call stack to determine the CPU calling context of $\mathcal{I}$ and inserting a placeholder $\mathcal{P}$ in callstack representing the GPU $\mathcal{I}$ type (eg. sync, copy, kernel) (step 1)
- Creating *correlation id* c_{id} that uniquely defines invocation $\mathcal{I}$ and sending it to monitoring thread
- Recording current time as invocation submit time t_{su}
- Creating one activity channel $C_{\mathcal{A}}$ for the current thread
- Sending c_{id}, $\mathcal{P}$, t_{su} and $C_{\mathcal{A}}$ to the monitor thread
- Optionally, turning on/off instrumentation using APIs such as PAPI

Once subscriber callback is executed, GPU operation is submitted to the launching queue and it will be executed on the GPU. During execution vendor APIs record GPU activities and send them to a performance tool for further processing. In the case of CUPTI and Rocm, GPU activities are asynchronously logged in vendor activity buffer and when the buffer is full, completion callback is triggered which allows performance tools to read GPU activities. Completion callback is executed from the monitor thread which is created from CUPTI and Rocm. In the case of OpenCL, each GPU activity triggers a callback to one application thread that needs to send data to the monitor thread created by the HPCToolkit. This is done through operation channels which transmit information about GPU activity and pointer to activity channel associated to the application thread where it should be attributed.

Every time the GPU monitor thread receives a buffer completion callback, it drains vendor activity buffer or operation channels and processes received GPU activities. The GPU monitor thread matches each GPU activity $\mathcal{A}$, tagged with its correlation id c_{id}, with its associated operation tuple (c_{id}, $\mathcal{P}, C_{\mathcal{A}}$). The monitor thread enqueues a pair $(\mathcal{A}, \mathcal{P})$ into activity channel $C_{\mathcal{A}}$ to attribute the GPU activity back to CPU-GPU calling context of launching thread $\mathcal{T}$.

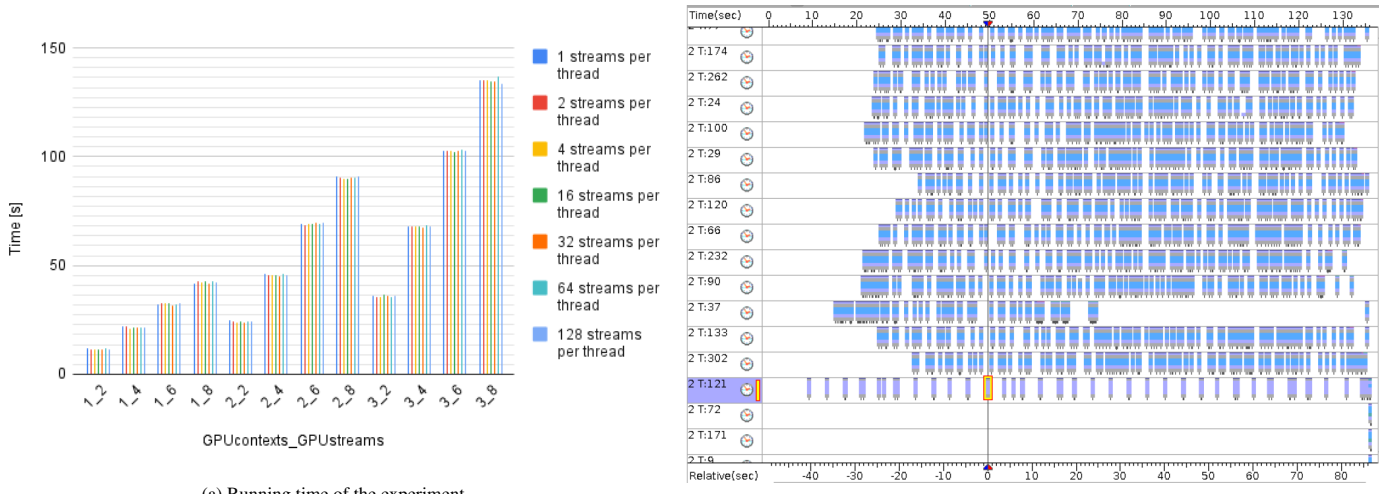
When tracing is enabled, the monitor thread separates GPU activities by their associated stream and sends each stream of activities to a tracing thread. Each tracing thread records one or more GPU streams of activities and their timestamps into trace files. For efficient inter-thread communication, HPCToolkit uses bidirectional channels, each consisting of a pair of wait-free single-producer and single-consumer queues [30]. The precise instantiation of HPCToolkit's monitoring infrastructure is tailored to each GPU vendor's software for monitoring GPU computations.

4.1. Support for OpenCL Metrics Attribution

When using OpenCL and Level Zero, depending upon the GPU operation invoked, either an application thread or a runtime thread will receive a completion callback providing measurement data. At each GPU API invocation $\mathcal{I}$ by an application thread $\mathcal{T}$, *hpcrun* provides a `user_data` parameter [31], which includes a placeholder node $\mathcal{P}$ for the invocation $\mathcal{I}$) and $\mathcal{T}$'s activity channel $C_{\mathcal{A}}$. The OpenCL or Level Zero runtime will pass `user_data` to the completion callback associated with $\mathcal{I}$.

At each completion callback, some thread receives measurement data about a GPU activity $\mathcal{A}$. Using information from its `user_data` argument, the completion callback correlates $\mathcal{A}$ with placeholder $\mathcal{P}$ and then enqueues an operation of $(\mathcal{A}, \mathcal{P}, C_{\mathcal{A}})$ for the monitor thread in its operation channel C_O. The monitor thread enqueues an $(\mathcal{A}, \mathcal{P})$ pair in $\mathcal{T}$'s activity channel $C_{\mathcal{A}}$.

If the thread receiving the callback enqueued $(\mathcal{A}, \mathcal{P})$ pairs directly into $\mathcal{T}$'s activity channel $C_{\mathcal{A}}$, $C_{\mathcal{A}}$ would need to be a



(a) Running time of the experiment

(b) Trace view of application with 3-contexts-80-streams and 1 tracing thread

Figure 3: Tracing performance for multi-contexts-multi-streams-application on gpu.cs.rice.edu x86_64 arch, 76 cores

multi-producer queue since more than one thread may receive completion callbacks for $\mathcal{T}$. To make the design simpler, we replace the need for a multi-producer queue for each $C_{\mathcal{A}}$ with processing callback data with monitor thread, creating just one wait-free multi-producer queue between callback threads and monitor thread, which we call operation multiplexer.

Operation multiplexer is an array of pointers to bidirectional wait-free channels. When the first thread gets a completion callback, it creates a shared array of pointers and initializes atomic variable operation_channels_count to zero, which tells how many channels are registered. At this point, the callback thread creates a monitor thread that can access the array of pointers. To register a channel each thread increments an atomic variable and saves unique channel_id which it uses for sending data to the queue. To read from operation multiplexer, monitor thread spins on channels with channel_id less than operation_channels_count.

4.2. Support for Scalable Tracing

When tracing is enabled, the monitor thread checks the GPU stream id $\mathcal{S}$ of each GPU activity and enqueues the activity and its placeholder $\mathcal{P}$ into a trace channel for $\mathcal{S}$. In the previous implementation, a new thread was created for each trace channel. Since some GPU applications could have hundreds of streams, we would create hundreds of tracing threads, which would create high memory footprint and sometimes wouldn't be allowed by execution policy.

To overcome this problem, we created a single-producer multi-consumer data structure called trace_demultiplexer that enables tracing of custom number of streams from a tracing thread. The first time the monitor encounters GPU tracing data, it associates stream_id with it, allocates the set of user-defined number trace channels, and creates a tracing thread that periodically polls a set of channels and writes them in trace files. For each new stream, we add a new channel to the channel set in a similar way like in operation multiplexer. Channel sets are connected with a linked list, and tracing threads are created only when previous channel sets are fully utilized. This approach enables us to monitor applications with an unlimited number of GPU streams.

To determine what is the best number of streams per thread, we run a multi-context multi-streams application with a variable number of contexts (from 1 to 3) and streams (from 20 to 80) and monitor the wall-clock time of the execution. The experiment was executed on 2 socket Power9 with 72 cores in total and results are shown in Figure 3.

In each experiment, we fix the number of GPU streams in the application and monitor the execution time for different values of the control knob (number_of_streams_per_thread) ranging from 1 to 128. From Figure 3a, we can see that the execution time stays the same, not depending on the number_of_streams_per_thread and therefore number of tracing threads. To get more insight we used Oracle Performance Analyzer [32] to show the activity of tracing threads.

Figure 3b shows the trace view and of the application with 3 contexts and 80 streams (total 240 streams) and number_of_streams_per_thread set to 256. Even in this scenario, just one tracing thread (marked in the figure), is capable of handling all streams. The reason for this is that GPU activities are delivered in batch to monitoring thread which sends data to the tracing thread. Base on this, we set default value of number_of_streams_per_thread to 256.

4.3. Support for Measuring Performance with Hardware Counters

HPCToolkit uses hardware performance counters to observe how an application interacts with a compute node. On the CPU, HPCToolkit accesses hardware counters using the Linux `perf_event` interface [33] and uses asynchronous sampling to collect a call path each time a hardware counter reaches a specified threshold. With appropriately chosen event thresholds, such measurement has a low overhead.

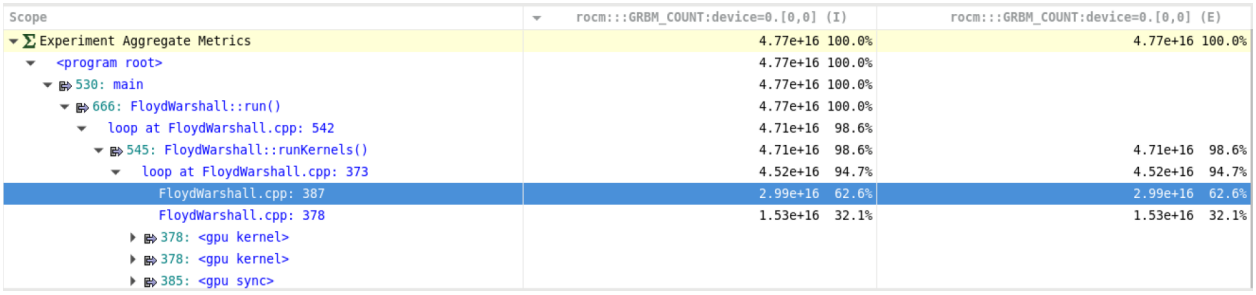


Figure 4: Hardware counters metric for AMD GPU for Rocm::GRBM_COUNT:device=0 Tie High Count number of clocks

To measure GPU performance with hardware performance counters, HPCToolkit uses the University of Tennessee's PAPI as a vendor-independent interface to hardware performance counters. PAPI has emerging support for using hardware counters on NVIDIA, AMD, and Intel GPUs. GPUs typically support the order of 100 hardware counters that characterize various aspects of functional unit utilization, device utilization, memory hierarchy activity, and more.

To associate hardware counter measurements with GPU kernels, existing APIs require reading data from counters before and after kernel execution in subscriber callback. Accurate attribution of hardware performance metrics to kernels requires kernel serialization. The synchronization needed to enforce serialization in an application that employs asynchronous kernels and concurrent kernel execution may slow an application program under study.

HPCToolkit associates performance counter metrics with calling context tree and presented in the profile view in HPCViewer in Figure 4. In this example, we were collecting the number of clocks with PAPI event *Rocm:GRBM_COUNT:device=0* for AMD GPU. For this application, we can see that 94.7% were spent in the loop at FloydWarshall.cpp on line 373 where 62.6% were spent on line 387 and 32% were spent on line 378.

5. Analysis of GPU accelerated applications

HPCToolkit's trace viewer provides a time-centric user interface for the interactive examination of a sample-based time series (hereafter referred to as a trace) view of program execution. It is designed to interactively present traces of large-scale execution across both CPUs and GPUs, relating activity to both hardware contexts (eg. nodes, GPUs, cores) and software contexts (e.g., MPI ranks, threads, GPU contexts, and streams).

As shown in Figure 5, the trace viewer's main pane shows ⟨profile, time⟩ dimensions, for each available call-stack depth. By changing call-stack depth, a user can change the granularity of trace lines, and gain insight into execution at different levels of abstraction. For the routine pointed by the cursor, functions from the call stack are listed on the right to the main pane. Each routine is uniquely identified with a specific color, while idle activity is assigned the color white.

The trace viewer's *Statistic* and *GPU Idleness Blame* tabs analyze the information in traces and offer some high-level characterizations of what the traces show. The *Statistic* tab calculates the percentage of the area occupied by each routine in the main pane and lists routines in descending order according to their percentage of the area. The *GPU Idleness Blame* tab employs blame analysis in an attempt to help application developers understand the causes of GPU idleness. To do so, it identifies times when all GPU streams are idle and at least one CPU thread is active. In such cases, it partitions the cost of GPU idleness among routines being executed by active CPU threads. The *GPU Idleness Blame* tab then presents normalized blame associated with each CPU function is sorted in descending order. CPU routines associated with high GPU idleness may be candidates for optimization.

5.1. GPU Blame shifting analysis

Figure 5 shows the trace view of the VectorAdd application on 8 CPU threads and 8 GPU streams. In this application the most of the time CPU spends in functions marked with braon(*cuMemcpyDtoHAsync*, 20.15%) and green (*cuMemFree_v2*, 10.69%) on the lowest level. Optimizing these functions wouldn't be beneficial since the application wouldn't finish until the GPU part is done. On the other hand, GPU blame analysis points to functions *cuMemAlloc_v2* and *gomp_team_start* which cause the most of idleness on GPU and their optimization would provide the most benefit. If we want to see what upper-level functions are responsible for GPU idleness blame, it is enough to change stack depth.

GPU blame analysis is based on counting pixels, that HPCToolkit draws from sampling CPU in real-time or from reconstructing intervals from GPU activities. Since all CPU samples describe just points in time, we have no information about what happened between the two samples, and we have to approximate them. Currently, HPCToolkit paints the space between an interval with the color of one endpoint. To increase precision we can increase the frequency of the sampling or we can use tools like *perfmon* which records the intervals when a thread is blocking or unblocking rather than just collecting points in time.

5.2. Nyx case study

Figure 6 shows a trace view of Nyx executing on Summit using 640 streams across 128 GPUs. The trace view shows that this execution consists of five phases. Phase 1 (0-10s)

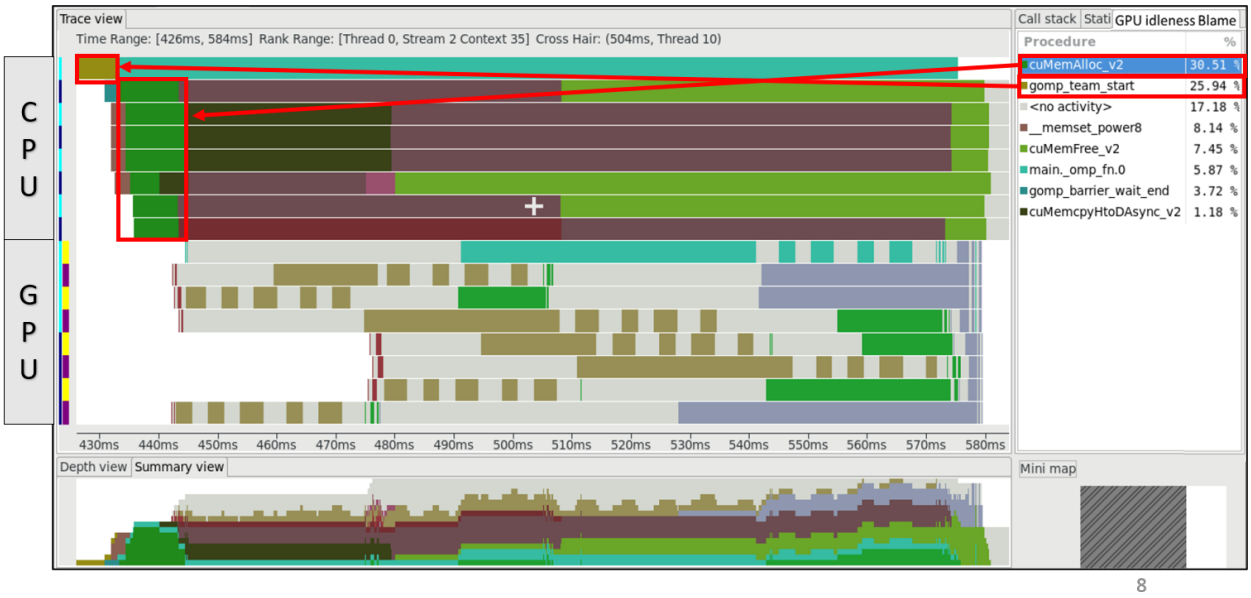


Figure 5: Application VectorAdd (8 CPU threads, 8 GPU streams)

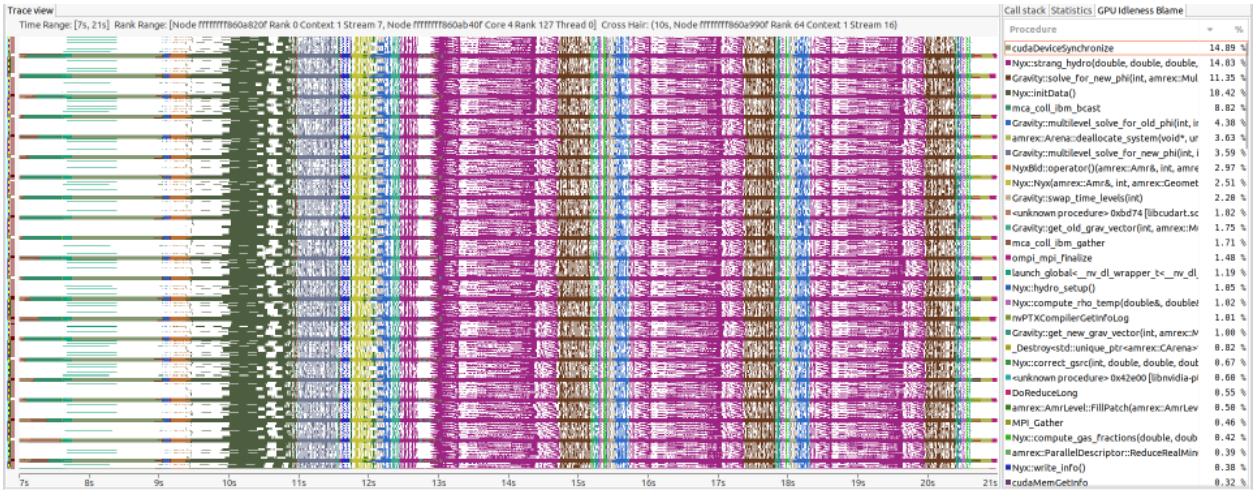


Figure 6: Nyx' trace view running across 22 nodes using 640 GPU streams and 128 processes.

corresponds to an initialization phase with little GPU activity. Phase 2 (10-13s) includes initialization of particle and dark matter data (shown in green) and calculation of gravity (shown in grey). Phases 3, 4, and 5 (13-20.6s) illustrate the same pattern that consists of the hydrodynamic (shown in magenta) and gravity calculations (shown in brown).

For each phase, we employed blame analysis to understand why GPUs are idle. In the first phase, 58.01% of idleness is caused by a call to `cuCtxSynchronize`, which synchronizes all streams on the GPU. Because only a single stream is used in this phase and a synchronized memory copy always follows the call to `cuCtxSynchronize`, we can safely remove the synchronization call and reduce the running time by 0.6s.

In the second phase, we easily identified that the major cause of idleness is JIT compilation at runtime. By providing the specific GPU architecture flag to the compiler and recompiling the program, we reduced the running time by 0.2s with this single optimization. In the following three phases, we identified that idleness is caused by calls to `MPI_Waitall`, which suggests that there maybe opportunities for improving performance by optimizing communication.

For the GPU intensive part of the application (10-20.6s), the insights provided by blame-shifting analysis reduced the GPU running time of Nyx from 10.6s to 9.8s, achieving a 1.08× speedup with 640 GPU streams. While the problems and improvements that we describe here are small, the important part is that HPCToolkit provides developers insights to identify even small problems.

6. Tooling requirements

In this section, we discuss problems in developing performance tools and best practices we derived during development. Performance tools must interact with complex infrastructure from vendor software stack and application runtimes without changing its behavior. Since many different teams are working on each of these parts at the same time and thinking from different perspectives, defining the set of invariants is paramount.

The initial set of invariants for HPCToolkit:

- HPCToolkit will be initialized as soon as a thread is created or right before the call to the main
- HPCToolkit expects to be informed about all system calls
- HPCToolkit doesn't monitor vendor infrastructure not accessible to the user

During our development, we needed to understand and relax our invariants in some cases and require other interacting tools to relax their invariants as well. For instance, vendor libraries could create threads too early during execution, and tracing that thread could be difficult for the tools. Instrumentation of function calls through LD_PRELOAD, doesn't work if the function called through function pointer obtain by using *dlopen* and *dlsym*. In the next subsections, we describe a set of problems and solutions that we faced during development.

6.1. Failing to observe the creation of a thread

For one application that uses HPE's SGI MPI library, we observed a segmentation fault, because we were trying to deallocate the tool data of the thread we never observed being created. This happened in the finalization step when we were freeing memory and we assumed that we were able to monitor all system calls during the execution. Since our first invariant was wrong, we needed to relax it and gracefully handle everything we overlooked. Now let's see why this problem happened.

To be able to monitor an application, HPCToolkit uses libmonitor to intercept the creation threads, processes, and system calls with LD_PRELOAD when dynamic linking is used. Libmonitor preloads a shared library libmonitor.so that overrides the application calls to __libc_start_main, _exit, fork, pthread_create, and uses dlopen to find the real versions of these functions. This way, each time a thread or process is created through these calls, we are able to use the callback functions *monitor_init_process()* and *monitor_init_thread()*, to turn on profiling and *monitor_fini_process()* and *monitor_fini_thread()* to turn it off. In this case, libmonitor separates the job of hooking the application program from the profiling tasks.

Unfortunately, LD_PRELOAD is not able to intercept the calls when *dlopen* and *dlsym* are used to call the procedure through the pointer. HPE's SGI MPI obtains a pointer to *pthread_create* using dlsym and calls it through a function pointer. Similarly, UCX does this with sigaction. This approach enables an application to enter critical functions without passing through wrappers introduced by a performance tool.

One way to handle this situation is to establish a convention that calls such as *pthread_create* and system calls are always called directly, and not through *dlopen* and *dlsym*.

We also experimented with using *Gotcha*, which has a mechanism that can be used to intercept *pthread_create*, but needs to be initialized before any thread is created. Since, LD_PRELOAD cannot observe every thread creating, we cannot reliably initialize *Gotcha* before all threads, and we have the same problem.

This problem may be solved with a careful wrapping of *LD_PRELOAD* around dlsym and detecting the call to *pthread_create*. However, this approach is error-prone, and the best way to solve this problem is to use the LD_AUDIT layer which should be able to guarantee that we cannot miss such events.

LD_AUDIT is Glibc's monitoring layer designed to provide an interface for tools to observe operations performed by the dynamic linker such as library lookup and symbol resolution. To employ the auditing interface, the environment variable LD_AUDIT must be defined to contain a colon-separated list of shared libraries, each of which can implement the auditing API. When an auditable event occurs, the corresponding function is invoked in each library, in the order that the libraries are listed.

From our experience, the auditor has a significant bug that requires fixes. We found that when dlmopen is used to open a library in the new namespace, which crashes the auditor and the only way to fix it is to use dlopen instead. If you tell the linker to bind now, the audit library *la_symbind()* isn't called. On some versions of Linux, Glibc adds the overhead of about 10x for auditing every PLT calls for small procedures, even when PLT auditing is off. Glibc incorrectly saves registers when auditing on aarch64 ARM architecture. These bugs are documented on [34] and we are working with the RedHat team on their resolution. Although LD_AUDIT seems unstable at the moment, it is the best way to enable performance tools to monitor system calls.

6.2. Escaping monitoring ourselves

To be successful, performance tools need to monitor and present only performance data associated with the application only. This task can be challenging since it is not enough to ignore threads created by our tool, but also all other threads created by any performance API that we are using. Otherwise, we would present the performance data of the infrastructure that the user cannot change. Although such transparency could be beneficial for learning about the system, it is usually not for optimizing the application. We can get a deadlock if we monitor threads that are writing performance data to the files since their activity will create new performance data all the time.

To avoid this, HPCToolkit uses a mechanism to avoid monitoring for all threads created from libraries that contain predefined functions by using *module_ignore_map* data structure. This data structure enables us to check if the thread that is being created, belongs to a blacklisted library and to ignore it if this is true. By using this approach we ignore all libraries that contain cuLaunchKernel, cuptiActivityEnable, roctracer_set_properties, hsa_init, etc.

6.3. Delaying initialization until the clean point

HPCToolkit used to perform one complete initialization the first time a thread is created or the main function is reached. This turns out to be a problem since the creation of a thread could happen at an unstable point when the runtime is not fully initialized. In our case, Rocm was spawning a thread during the initialization of *hsa_init* while holding a lock. Libmonitor intercepted thread creation and HPCToolkit started complete initialization and ended up in a deadlock. It turned out that HPCToolkit started the initialization of the PAPI library that called *hsa_init* as well to initialize their Rocm component waiting for the same lock.

Formally speaking, the problem happened because AMD's invariant that nobody should intercept their initialization while holding a lock was broken by HPCToolkit. To fix this, we designed a two-step initialization that postpones the critical part of HPCToolkit's initialization until the stable point. The current HPCToolkit initialization consists of two parts - core initialization and measurement subsystem initialization.

Core initialization is called only the first time some thread is created and it is used to set up shared libraries, unwind infrastructure, and function bounds. First, we initialize the load map that will hold all shared libraries. Next, we initialize x86 unwinding infrastructure and module_ignore_map which we use to predicate libraries which calls we don't want to monitor. Finally, we use function bounds to specify address offsets that we use during unwind process.

We initialize the measurement subsystem at the stable point right before the main function starts executing, or when a new application thread is created. First, we initialize all sampling sources (eg. PAPI_library_init) and user-defined control-knobs(eg. *streams_per_tracing_thread*). Then we initialize a thread data structure that holds all information about collected metrics for a given thread. We parse user-defined monitoring events and set up parameters for sampling frequency and interrupt threshold. Finally, we combine sampling events into sets of events that will be collected at the same time on the CPU/GPU and we turn on sampling for each component.

Similar to AMD invariant, HPCToolkit also has an invariant about uninterruptability during initialization. This invariant was broken when AMD creates thread when Rocprofiler library is loaded, which happens in the middle of HPCToolkit initialization, and triggers initialization of measurement subsystem which breaks. As a resolution of this conflict, AMD fixed the issue by spawning a thread when the rocprofiler component is turned on rather than on library loading.

6.4. Problem with PAPI for the applications that fork

Applications that forks require some careful engineering from performance tools. Every time fork happens, HPCToolkit stop and shutdown all sample sources and reinitialize sample sources, figure out event sets and turn on the sampling. This is done because every thread needs to have sampling sources initialized separately, and they cannot share data structures.

We tested the Spectrum MPI application on the Summit supercomputer and found one more invariants conflict. PAPI library doesn't expect a fork after the Cuda component is being initialized. In our example, Spectrum MPI has integrated a pmpix daemon that forks at the beginning of the application. At that point, HPCToolkit has already initialized sampling sources including the PAPI Cuda component, which conflicts with the PAPI invariant. The application fails when HPCToolkit restarts PAPI sampling after the fork happens. The reason for this that the shutdown is not done properly and both processes are still sharing some internal representation of the initial state. Since the PAPI Cuda component is not set properly the collected values from hardware counters are garbage.

PAPI team reported that after a fork() follows PAPI_init(), besides PAPI failure, the bare Cuda functions such as cudaGetDevice(), cudaMalloc(), and cudaMemcpy() fail as well just for the child process while succeeding for the parent process. This indicates that the problem lays in the initialization of the child thread. To solve this problem it is necessary to synchronize invariants between PAPI and Nvidia tools and clearly define them to avoid future confusion.

7. Conclusion

In this paper we described the process of building performance tools, adding new features to HPCToolkit, and explaining the problems and best practices we derived during development.

The first step in the development process is to understand monitoring APIs available for collecting performance data. In principle, there are two common approaches which are sampling and instrumentation. Sampling incurs small overhead and provides the bigger picture of the execution, while instrumentation incurs higher overhead, but enables focus on the specific parts. For GPU tracing, the vendors provide specific APIs for tracing GPU activities through asynchronous delivery of GPU activity records. Besides this, they provide a callback for instrumenting every interaction between CPU and GPU. Additionally, we can use vendor-independent tools like PAPI to collect data from performance counters through provided instrumentation callbacks from vendor APIs. In HPCToolkit, we use all of these capabilities to collect performance metrics from Nvidia, AMD, and Intel GPUs and attribute them to the source-line from where they originate.

In this paper, we described GPU tracing infrastructure with newly added support for OpenCL, scalable tracing, collection of hardware counters on GPU, and create GPU idleness analysis. To associate GPU activities with the launching CPU threads, we designed a multi-producer single-consumer data structure to enable effective communication between callback threads and the monitoring thread that processes and associates GPU activity to the right CPU threads. Additionally, reused this data structure to design a scalable tracing infrastructure that enables a user to specify the number of GPU streams per tracing thread. This way we make possible tracing of applications with an unlimited number of GPU streams. We added support for monitoring performance counters through the PAPI interface, which enables us to monitor the order of 100 events on all three

GPUs. Finally, we developed a GPU idleness analysis that pinpoints the parts of CPU code responsible for GPU idleness.

During development, we several problems and derived the best practices in developing performance tools. First, we noticed that the LD_PRELOAD mechanism doesn't notify us about important system calls if it is called through the pointer by using the open/dlsym mechanism. As a replacement, we discussed using LD_AUDIT and we found a set of issues that needs to be addressed before it becomes appropriate for performance tools. We explained the mechanism of how we avoid monitoring ourselves and we designed deferred initialization to avoid conflict with AMD's thread creation on unstable points. Finally, we explained the forking problem that happens when using the PAPI Cuda component.

To avoid future problems between the interacting parts of performance tool architecture it is necessary that vendors and tool developers specify the set of invariants for each piece and to maintain it. Each developing team has its perspective and often it is hard to communicate the problem effectively between teams. Even worse, untransparent development and poorly written documentation could prevent performance tools for developing new capabilities, and therefore limit the final user's application efficiency.

References

[1] M. García and T. Munson, "Exascale computing project report 2020," *2020 Exascale Computing Project Annual Meeting*, vol. 34, no. 5, pp. B606–B641, 2012.

[2] L. Braun and H. Fröning, "CUDA Flux: A lightweight instruction profiler for CUDA applications," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS) Workshop, collocated with Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC2019)*, 2019.

[3] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[4] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling many-core performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[5] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM Intl. Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[6] B. Johnston, "Evaluating the performance and portability of contemporary sycl implementations," *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020.

[7] R. S. D. G. Keren Zhou, Xiaozhu Meng and J. Mellor-Crummey, "An automated tool for analysis and tuning of gpu-accelerated code in hpc applications," *Journal of Transactions on Parallel and Distributed Systems*, no. 12, pp. 0–12, 2021.

[8] M. Garcia and T. Munson, "2020 exascale computing project annual meeting (executive summary report)." [Online]. Available: https://www.osti.gov/biblio/1649172

[9] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir performance analysis toolset," in *Tools for High Performance Computing*. Springer, 2008, pp. 139–155.

[10] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The Intl. Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[11] C. January *et al.*, "Allinea MAP: Adding energy and OpenMP profiling without increasing overhead," in *Tools for High Performance Computing 2014*. Springer, 2015, pp. 25–35.

[12] M. Schulz, J. Galarowicz, D. Maghrak, W. Hachfeld, D. Montoya, and S. Cranford, "Open|SpeedShop: An open source infrastructure for parallel performance analysis," *Scientific Programming*, vol. 16, no. 2-3, pp. 105–121, 2008.

[13] J. Reinders, "VTune performance analyzer essentials," *Intel Press*, 2005.

[14] NVIDIA Corporation. (2020) NVIDIA Nsight Systems. [Accessed Jan. 1, 2021]. [Online]. Available: https://developer.nvidia.com/nsight-systems

[15] ——. (2020) NVIDIA Nsight Compute. [Accessed Jan. 1, 2021]. [Online]. Available: https://developer.nvidia.com/nsight-compute

[16] ——, *Profiler User's Guide DU-05982-001_v11.2*, December 2020. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf

[17] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of GPU-accelerated applications," in *Proc. of the 34th ACM Intl. Conf. on Supercomputing*, ser. ICS '20. New York, NY, USA: ACM, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392752

[18] NVIDIA Corporation, *CUPTI User's Guide DA-05679-001_v11.2*, 2020, https://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf.

[19] Advanced Micro Devices, Inc. AMD ROCm ROCProfiler. [Online]. Available: https://rocmdocs.amd.com/en/latest/ROCm_Tools/ROCm-Tools.html

[20] Intel Corporation, "oneAPI Level Zero: 1.0.4.46," https://spec.oneapi.com/level-zero/index.html, [Accessed Oct. 24, 2020].

[21] M. Kambadur *et al.*, "Fast computational GPU design with GT-Pin," in *2015 IEEE Intl. Symp. on Workload Characterization*. IEEE, 2015, pp. 76–86.

[22] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[23] K. Z. et al., "Measurement and analysis of gpu-accelerated applications with hpctoolkit," *Journal of Parallel Computing*, no. 14, pp. 0–14, 2020.

[24] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, Feb 2021. [Online]. Available: https://www.openmp.org/spec-html/5.1/openmp.html

[25] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12. Berlin, Heidelberg: Springer-Verlag, 2012, p. 859–870. [Online]. Available: https://doi.org/10.1007/978-3-642-32820-6_85

[26] AMD Corporation. (2021) HIP Programming Guide. [Accessed June 21, 2021]. [Online]. Available: https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html

[27] G. S. John E. Stone, David Gohara, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering Journal*, vol. 12, no. 3, pp. 66 – 73, 2010.

[28] Intel Corporation, "DPC++," https://spec.oneapi.com/versions/latest/elements/dpcpp/source/index.html, [Accessed Oct. 24, 2020].

[29] K. Zhou and J. Mellor-Crummey, "A tool for performance analysis of gpu-accelerated applications," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 282.

[30] K. Zhou, M. W. Krentel, and J. Mellor-Crummey, "Tools for top-down performance analysis of gpu-accelerated applications," in *Proceedings of the 34th ACM International Conference on Supercomputing*, ser. ICS '20. New York, NY, USA: Association for Computing Machinery, 2020.

[31] Khronos Group, "clSetEventCallback(3) manual page," https://www.khronos.org/registry/OpenCL/sdk/2.2/docs/man/html/clSetEventCallback.html.

[32] Oracle. (2016) Oracle® Developer Studio 12.5: Performance Analyzer. [Accessed June 21, 2021]. [Online]. Available: https://docs.oracle.com/cd/E60778_01/html/E60749/index.html

[33] V. M. Weaver, "Linux perf_event features and overhead," in *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, vol. 13, 2013, p. 5.

[34] John Mellor-Crummey, Xiaozhu Meng, Jonathon Anderson, Mark Krentel. (2021) Glibc support for LD_AUDIT tests. [Accessed June 21, 2021]. [Online]. Available: https://github.com/HPCToolkit/auditor-tests