Special Issue: Post ECP

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2025, Vol. 39(2) 251–268

DOI: 10.1177/10943420241303884
journals.sagepub.com/home/hpc
Sage

Advancements of PAPI for the exascale generation

Heike Jagode[1], Anthony Danalis[1], Giuseppe Congiu[1], Daniel Barry[1], Anthony Castaldo[2] and Jack Dongarra[1]

Abstract

The Performance Application Programming Interface (PAPI) serves as a coherent, operating-system-independent interface for accessing performance counter data across a wide range of hardware and software components. PAPI can operate autonomously as a performance monitoring library and tool for application analysis. However, its true value emerges when it functions as a middleware for numerous third-party profiling, tracing, and sampling toolkits, establishing itself as a universal interface for hardware counter analysis. In this role, PAPI manages the intricacies of each hardware component, presenting a streamlined API to higher-level toolkits. Within the Exascale Computing Project (ECP), PAPI has expanded its capabilities in performance counter monitoring and incorporated support for power management across cutting-edge hardware and software technologies. This includes performance and power monitoring for AMD GPUs through integration with AMD ROCm and ROCm-SMI, Intel Ponte Vecchio GPUs via Intel's oneAPI Level Zero, and NVIDIA GPUs through the CUPTI Profiling API. Additionally, PAPI is compatible with interconnects, the latest CPUs, and ARM chips. These enhancements have been implemented while preserving the standard PAPI interface and methodology for utilizing low-level performance counters in CPUs, GPUs, on/off-chip memory, interconnects, and the I/O system, encompassing energy and power management. To strengthen PAPI's sustainability, ECP has facilitated its integration into Spack and E4S, ensuring software robustness through continuous integration and continuous deployment. In addition to hardware counter-based data, PAPI now supports the registration and monitoring of Software-Defined Events. This feature exposes the internal behavior of runtime systems and libraries like PaRSEC, SLATE, Magma, to applications utilizing those libraries, broadening the scope of performance events to include software-based information. Additionally, PAPI has been expanded with the Counter Analysis Toolkit, aiding in native performance counter disambiguation through micro-benchmarks. These micro-benchmarks probe various essential aspects of modern chips, contributing to the classification of raw performance events. In summary, ECP has enabled PAPI to include comprehensive counter analysis capabilities, advanced performance and power monitoring support for exascale hardware components, and broadened the scope of performance events to encompass not only hardware-related metrics but also software-based information.

Keywords

Performance monitoring, power monitoring, energy efficiency, hardware performance counters

1. Introduction

Modern computer systems need to integrate with a variety of computational applications, software and hardware technologies, each presenting unique characteristics and resource requirements. On the software side, it is imperative that applications and software layers execute efficiently and at high performance. On the hardware side, while many modern compute environments are built using commodity components, there is a need to develop and incorporate new technologies to support forthcoming advancements in software and hardware. Central to these efforts—whether it is ensuring the efficient execution of applications and software layers, or the development and integration of new hardware technologies—is the crucial role played by performance monitoring tools and solutions. The objective of these tools is to continuously track metrics at the lowest

[1]Innovative Computing Laboratory (ICL), University of Tennessee, Knoxville, TN, USA
[2]Synopsys Inc, Sunnyvale, CA, USA

Corresponding author:
Heike Jagode, Innovative Computing Laboratory (ICL), University of Tennessee, 1122 Volunteer Blvd, Knoxville, TN 37996, USA.
Email: jagode@icl.utk.edu

hardware level, helping to identify potential inefficiencies or bottlenecks, and facilitating the operation of high-performance, and energy-efficient software and hardware.

The Performance Application Programming Interface (PAPI) plays a central role in this, as it serves as a universal interface for accessing performance counter data across a wide range of hardware and software components (Jagode et al., 2016; Jagode-McCraw et al., 2014; Terpstra et al., 2010). Specifically, PAPI provides performance counter monitoring and power management (reading and capping) support (Haidar et al., 2017, 2018) for the latest CPUs from various hardware vendors, as well as for AMD and NVIDIA GPUs. It also offers monitoring support for low-level performance metrics on Intel GPUs. Additionally, PAPI provides insights into network congestion for various interconnects, such as Infiniband, Aris, Gemini, and Slingshot, as well as monitoring I/O activities, including Lustre filesystem operations (Jagode and Hein, 2008; Jagode-McCraw et al., 2013). Last but not least, the PAPI software-defined-events (SDE) component helps uncover essential software-critical events from different libraries and runtimes (Danalis et al., 2019; Jagode et al., 2019).

Accessing all this information uniformly through the PAPI library is crucial for maintaining productivity. Without it, application developers and researchers would be forced to use multiple APIs to access all available performance events, which could significantly decrease efficiency. Further, monitoring and analyzing the performance and energy efficiency of applications running on increasingly complex, heterogeneous, multi-vendor systems would become exceedingly difficult.

This paper highlights the development and advancements delivered for PAPI within the Exascale Computing Project (ECP), including novel performance counter monitoring capabilities and support for power management across cutting-edge hardware from various vendors. In addition to providing support for the latest hardware and software layers, ECP also improved PAPI's sustainability by enabling integration into the Spack package manager (Gamblin et al., 2015) and the Extreme-scale Scientific Software Stack (E4S) (Willenbring et al., 2023), and ensuring software robustness through continuous integration and deployment. With the ongoing integration of new monitoring capabilities for advanced hardware and software technologies, PAPI is well-positioned to meet the emerging needs of the high-performance computing community, continuing to make an impact well beyond the ECP era.

2. Performance and power monitoring capabilities for GPUs

Within ECP, the PAPI library has been significantly advanced with functionalities by integrating comprehensive performance counter monitoring capabilities and embracing support for power management across the latest hardware and software technologies. This broadened scope of support encompasses detailed performance metrics and power usage insights for diverse hardware platforms, facilitating a deeper understanding of application behavior and energy efficiency. The subsections below detail the monitoring capabilities and mechanisms used to provide comprehensive access to performance and power metrics for GPUs from various vendors, including AMD, Intel, and NVIDIA.

Overall, these PAPI integrations provide a unified and cross-platform approach to performance and power monitoring, crucial for optimizing high-performance computing applications on exascale systems. This cross-vendor support ensures consistency and ease of use across different architectures, providing users with the necessary tools to optimize their applications for AMD's, Intel's, and NVIDIA's latest GPU architectures by offering insights into computational efficiency and energy usage.

2.1. AMD

For AMD GPUs, PAPI has been integrated with AMD's vendor-specific Radeon Open Compute Platform (ROCm) (AMD, 2024a) and the ROCm System Management Interface (ROCm-SMI) (AMD, 2024b), enabling the capture of detailed performance metrics such as execution cycles, floating-point operations, memory accesses, and power consumption. The new PAPI components, "`rocm`" and "`rocm_smi`", provide users with a portable solution to access performance counters and configure hardware parameters, including power caps, on AMD GPU devices.

2.1.1. Performance monitoring capabilities. The following list, presented without any particular order, highlights key monitoring features frequently requested and available on the latest AMD GPU devices:

- LDS/GDS: Local/Global Data Store events
- TCP/TA: L1 Cache-related counts
- TCC: L2 Cache-related counts
- SQ: Sequencer events for fetch, decode, schedule instructions
- FLAT: Instructions using flat address space for Video, Sys, LDS, and Scratch, which are slower
- VMEM: Vector Memory storage events
- VMem: Video Memory events
- SMEM: Scalar Memory storage events
- SFetch: Scalar fetch events
- VFetch: Vector fetch events (excluding FLAT instructions)

- VALU: Vector Arithmetic Logic Unit events, including FMA floating-point operations for various data types
- SALU: Scalar Arithmetic Logic Unit events

The PAPI "rocm" component provides two modes for monitoring hardware performance metrics on AMD GPU devices. While previous AMD GPU monitoring was limited to "sampling mode" only, the latest PAPI 7.1.0 release introduces support for both "sampling" and "intercept mode" for the "rocm" component. To switch between the two monitoring modes for AMD GPUs, users must set the environment variable ROCP_HSA_INTERCEPT (1 or 2 for selecting intercept mode, 0 for sampling mode, which is the default). Regardless of the collection mode selected, the usage of the PAPI interface remains identical.

This approach mirrors the PAPI support for Intel GPUs, which is covered in 2.2. As is common, each vendor uses its own unique terminology. To provide a brief overview, "sampling mode" on AMD GPUs corresponds to "time-based mode" on Intel GPUs; both refer to performance counter values that can be read at any time, independent of the kernels running on the GPUs. In contrast, "intercept mode" on AMD GPUs is equivalent to "kernel-based mode" on Intel GPUs. This mode refers to performance counter monitoring on a per-kernel basis, where counter values can only be read after a kernel has finished executing. It is important to note that in this mode, the respective GPU runtime serializes kernels that overlap in execution. Essentially, even if two or more kernels could be executed concurrently on the GPU, AMD's ROCm and Intel's oneAPI serialize their execution when "intercept" (AMD) or "kernel-based" (Intel) collection mode is used. This ensures that event counts are accurately associated with individual kernels.

2.1.2. GPU-to-GPU connectivity. The high-performance configurations of AMD's XGMI architecture enable high-speed, direct GPU-to-GPU connectivity, and optimizing its use is crucial for application performance. PAPI now integrates new monitoring capabilities for AMD's XGMI-Link GPU-to-GPU interconnect-related metrics, including the volume of transmitted and received data, as well as the throughput for both transmitted and received data, among other network-monitoring capabilities. PAPI leverages vendor-specific low-level APIs to access GPU-to-GPU monitoring capabilities and XGMI-related performance metrics on AMD GPUs from user space. Moreover, capabilities for monitoring XGMI interconnect activities, crucial for performance monitoring of multi-threaded and multi-GPU applications, are integrated into the "rocm_smi" component of PAPI. The example below demonstrates that the PAPI event rocm_smi:::min_xgmi_internode_bw:device=6:target=0 indicates a minimum bandwidth utilization of 50,000 bytes per second between devices 6 and 0, while event rocm_smi:::max_xgmi_internode_bw:device=0:target=1 reveals a maximum bandwidth utilization of 200,000 bytes per second between devices 0 and 1. Additionally, and the zero result for event rocm_smi:::max_xgmi_internode_bw:device=0:target=3 indicates that devices 0 and 3 are not connected directly.

Example: $ ./papi_command_line

```
rocm_smi:::min_xgmi_internode_bw:device=6:target=0
rocm_smi:::max_xgmi_internode_bw:device=0:target=1
rocm_smi:::max_xgmi_internode_bw:device=0:target=3
```

Output:

```
rocm_smi:::min_xgmi_internode_bw:device=6:target=0:
   50,000
rocm_smi:::max_xgmi_internode_bw:device=0:target=1:
   200,000
rocm_smi:::max_xgmi_internode_bw:device=0:target=3:
   0
```

2.1.3. Power usage. The PAPI "rocm_smi" component provides support for power management on AMD GPUs, including monitoring power consumption, fan speed, temperature, and enabling power capping, which allows users to modify run profiles to potentially reduce energy consumption. Below are some of the monitoring features available for AMD GPUs:

- Power: Monitoring and power capping capabilities.
- Temperature: Monitoring current temperature, maximum critical value, and temporary emergency temperature.
- Fan: Measuring fan speed in rotations per minute, maximum speed, and read/write speed.
- Memory: Monitoring used and total memory for: VRAM (Video RAM, or graphics memory), visible VRAM (CPU-accessible video memory on the device), and Graphics Translation Table (GTT).
- PCI: Monitoring throughput sent, received, and maximum packet size.
- Busy Percent: Assessing the percentage of time the device is busy processing.

Figure 1 illustrates the monitoring of PAPI "rocm_smi" events within the TAU profiler (Shende and Malony, 2006) to observe power usage for a hipBLAS (AMD, 2024c) GEMM (General Matrix Multiply) kernel on AMD GPUs. It demonstrates that the PAPI "rocm_smi" component is fully integrated, allowing all its monitoring capabilities to be utilized by third-party performance analysis tools, like TAU, that already leverage PAPI as middleware.

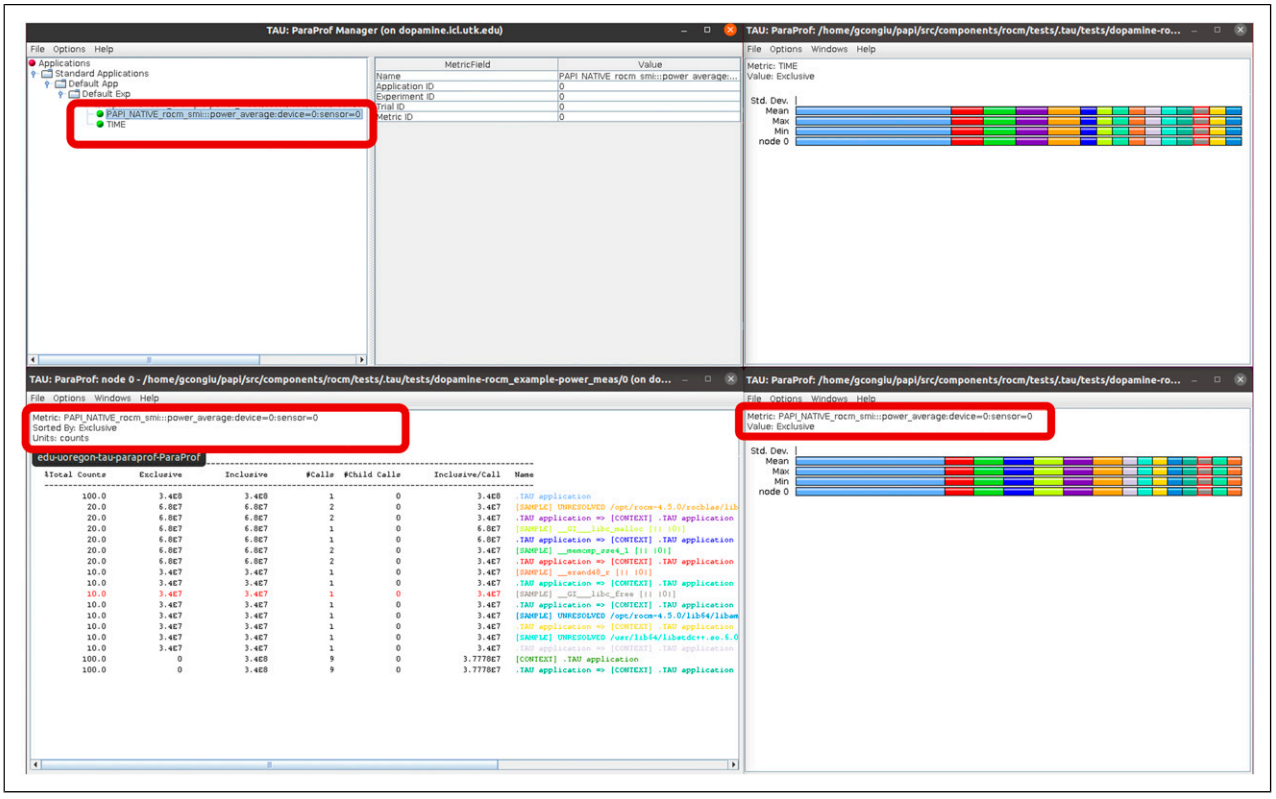

Figure 1. PAPI power monitoring of a hipBLAS GEMM kernel using TAU on AMD GPUs.

2.2. *Intel*

In the realm of Intel hardware, PAPI now supports Intel's Ponte Vecchio GPUs through integration with Intel's oneAPI Level Zero interface (Intel, 2024b). This allows for the monitoring of a wide range of performance counters specific to Intel's latest GPU architecture, including metrics on compute operations, memory bandwidth utilization, and others. By leveraging these insights via the PAPI "`intel_gpu`" component, developers can tailor their applications to better utilize Intel's GPU resources, enhancing both performance and throughput efficiency.

The PAPI "`intel_gpu`" component offers two collection modes for monitoring Intel GPU hardware events and memory performance metrics, such as bytes read, written, or transferred from/to the L3. On one hand, the "Time-based Collection Mode" aggregates counter values in a buffer, allowing them to be read at any time, making performance counter monitoring completely independent of the kernels running on the Intel GPUs. On the other hand, the "Kernel-based Collection Mode" monitors performance on a per-kernel basis, with counter data available only after a kernel has finished executing. In this mode, Intel oneAPI uses internal barriers. This means that if two or more kernels overlap in execution, the GPU runtime serializes their execution to ensure that counter values are correctly associated with individual kernels. Figure 2(a) and (b) illustrate both modes, respectively.

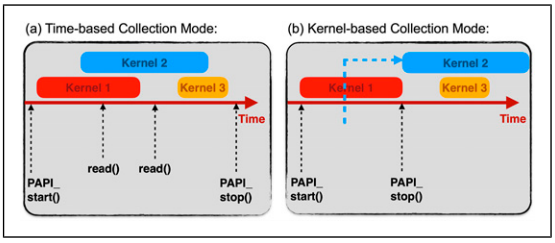

Figure 2. Collection modes for monitoring Intel GPU metrics.

The examples (a) and (b) below confirm that PAPI effectively supports both collection modes, which can be alternated by setting the environment variable `ZE_ENABLE_TRACING_LAYER` (1 for kernel-based, 0 for time-based mode (default)). For illustration purposes, a matrix-matrix multiplication (GEMM) was instrumented using PAPI, and each GEMM computation was executed four times. In example (a), performance counter data is collected in "kernel-based mode," and the results for the three selected PAPI counters are reported only after the kernel execution has finished on the Intel device. Notably,

PAPI counter 1, which tracks the GPU-time event, matches the total execution time (real time) reported from the system.

Example (a): Kernel-based Collection Mode:

```
$ ./gemm_papi
Matrix-Matrix Multiplication (repeats 4 times)
Target device: Intel(R) GPU

Matrix multiplication time: 3.25726 sec
Matrix multiplication time: 3.25611 sec
Matrix multiplication time: 3.25520 sec

Matrix multiplication time: 3.25601 sec

  PAPI Counter 1: GPU Time (ns):   13,419,165,548
  PAPI Counter 2: Memory Reads:    10,208,287,486
  PAPI Counter 3: L3 Cache Misses: 7,020,866,902

Total execution time (real time): 13.6682 sec
```

In Example (b), performance counter data is gathered in "time-based mode," where a separate thread executes `PAPI_read()` every 200 ms. This method allows users to obtain more frequent counter updates during a kernel execution on the device, rather than single counter values post-execution.

The outputs from Counter 1 (GPU-time) confirms that the interval between reported values is approx. 200 ms, which aligns with the chosen `PAPI_read()` sampling frequency.

Example (b): Time-based Collection Mode:

```
$ ./gemm_papi
Matrix-Matrix Multiplication (repeats 4 times)
PAPI_read() every 200ms
Target device: Intel(R) GPU

  PAPI Counter 1: GPU Time (ns): 314,367,691
  PAPI Counter 1: GPU Time (ns): 524,628,819
  PAPI Counter 1: GPU Time (ns): 729,428,619
...
  PAPI Counter 1: GPU Time (ns): 13,579,592,059

Total execution time (real time): 13.7564 sec
```

The provided examples serve primarily as validation cases to demonstrate PAPI's newly introduced monitoring capabilities for the latest Intel GPU technologies.

2.3. NVIDIA

For NVIDIA GPUs, PAPI has incorporated support via the NVIDIA Nsight Perf SDK (NVIDIA, 2024d), enabling detailed performance (via the PAPI "`cuda`" component) and power (via the PAPI "`nvml`" component) monitoring. This includes tracking of CUDA kernel executions, memory transfers, and GPU power usage, among others. The integration with Nsight Perf SDK equips developers with the tools to fine-tune their applications on NVIDIA GPUs with compute capability 7.0 and higher, focusing on maximizing computational throughput while minimizing energy consumption.

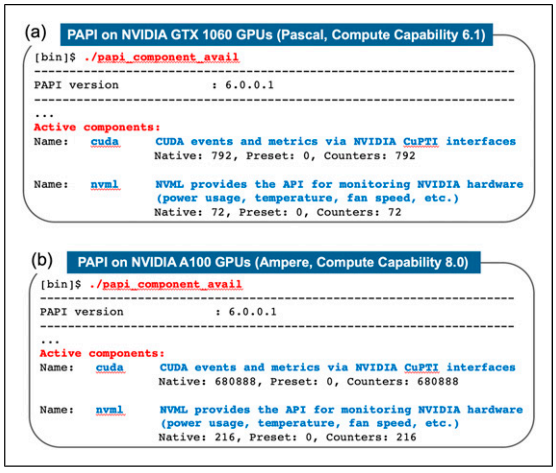

Figure 3. PAPI on NVIDIA compute capabilities < and > 7.0.

2.3.1. Performance monitoring capabilities. The PAPI "`cuda`" component, designed for hardware counter monitoring on NVIDIA GPUs, has been a staple in performance analysis for several years (Malony et al., 2011). However, with the deprecation of NVIDIA's CUPTI (NVIDIA, 2024c) "Event API" for compute capabilities greater than 7.0—which served as the primary mechanism for hardware counter monitoring on earlier architectures—PAPI faced the challenge of maintaining support for newer NVIDIA architectures.

In response, the PAPI "`cuda`" component was significantly refactored to ensure uniform support across NVIDIA compute capabilities, both below and above 7.0. This integration required leveraging the advanced features of CUPTI's "Profiling API" and the Nsight Perf SDK to facilitate performance monitoring for NVIDIA compute capability 7.0 and beyond, including Turing, Ampere, and Hopper. This effort not only ensured PAPI's compatibility with NVIDIA's evolving architecture but also expanded its applicability for performance analysis across a broader spectrum of computing environments, all while maintaining a consistent interface.

Figure 3(a) and (b) show the output of the PAPI utility called "`papi_component_avail`" when run on NVIDIA GPUs with compute capabilities 6.1 (Pascal GPUs) and 8.0 (Ampere GPUs), respectively. This utility provides a list of the supported and enabled PAPI components. In this particular case, both the "`cuda`" and "`nvml`" components are active on both systems. Additionally, it displays the number of events available for monitoring. This

demonstrates that the same version of PAPI operates uniformly, regardless of the underlying APIs used for different NVIDIA compute capabilities.

2.3.2. Power usage. The PAPI "`nvml`" component has been updated to support power management for the latest NVIDIA GPUs, including monitoring of power consumption, fan speed, temperature, and power capping. This enhancement means that power monitoring and capping are now available for NVIDIA GPUs with compute capability 7.0 and higher, extending to Turing, Ampere, Hopper, and beyond, in addition to previously supported GPUs with compute capability less than 7.0.

Figures 4 and 5 illustrate power readings obtained with PAPI for A100 GPUs during the execution of two GEMM kernels using the numerical linear algebra libraries MAGMA (Agullo et al., 2009) and cuBLAS (NVIDIA, 2024a), respectively. These figures present power usage data (without power caps) while various GEMM operations are performed. Figure 4 provides fine-grained power monitoring with a 10 ms sampling rate for a single GEMM kernel, whereas Figure 5 reports the final power usage at the completion of different GEMM computations with both MAGMA and cuBLAS.

Figure 6(a) and (b) present graphs of power readings and capping capabilities utilizing PAPI on A100 GPUs. The emphasis here is on demonstrating the power capping feature, which requires elevated privileges. The same GEMM kernels, using both MAGMA and cuBLAS, were executed as in the previous examples. In Figure 6(a), the power was capped at 250 W, while in Figure 6(b), the power limit was set at 150 W. These examples primarily serve to demonstrate PAPI's power

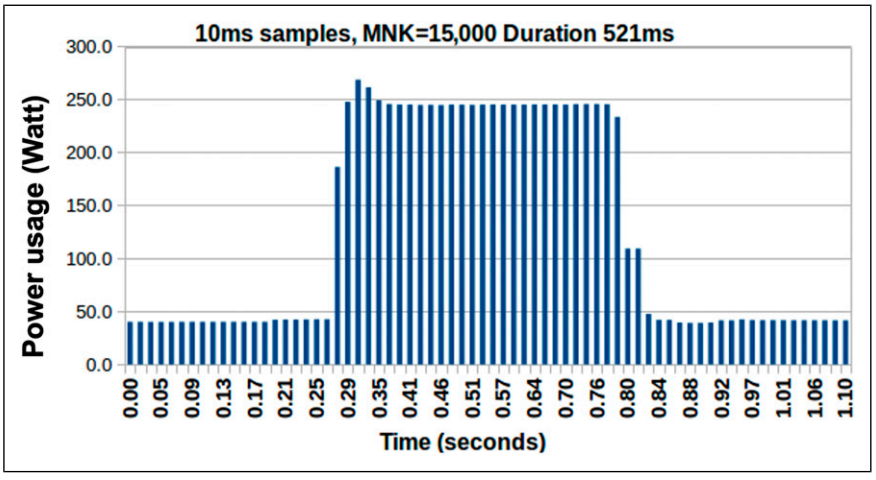

Figure 4. Fine-grained power monitoring of a single GEMM.

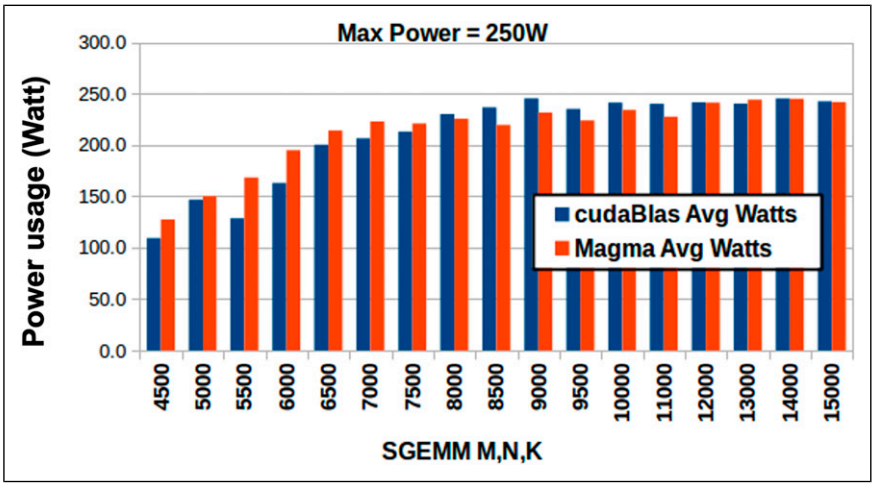

Figure 5. Power monitoring on NVIDIA GPUs.

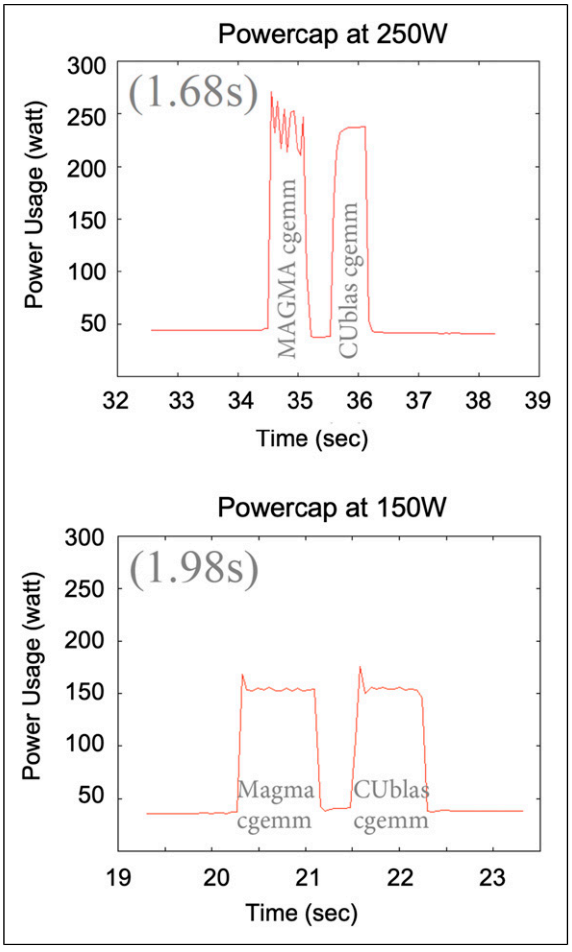

Figure 6. Power capping on NVIDIA GPUs.

monitoring and emphasize its capping capabilities on the latest NVIDIA GPUs.

Figure 7 illustrates the monitoring of PAPI "`cuda`" and "`nvml`" metrics using Vampir (Brunst and Knüpfer, 2011) and Score-P (Schlütter et al., 2014) to observe the performance as well as power usage for a Kokkos application on NVIDIA GPUs. It shows that the PAPI "`cuda`" and "`nvml`" components are fully integrated, enabling the use of all their monitoring capabilities by third-party performance analysis tools, such as Vampir, that already use PAPI as middleware.

3. Detection of hardware topology features

Through its various components, PAPI can monitor counters across a wide range of hardware. To achieve this, each component interfaces with the appropriate low-level vendor API, such as CUPTI, ROCm, OneAPI, etc. A new development, which took place during the ECP project, was the creation of the "`sysdetect`" component. This component goes beyond monitoring counters on a specific type of hardware. Instead, it leverages the APIs from different vendor-specific PAPI components to provide a universal system detection infrastructure.

This new functionality has been implemented through a new user API as well as a command line tool, both of which are designed to support a broad range of hardware, both current and forthcoming. Through the command line tool, `papi_hardware_avail`, users can directly access information collected by the PAPI "`sysdetect`" component to probe the hardware elements of their machine. Additionally, applications like NWChemEx (Kowalski et al., 2021), which have a complex structure designed to adapt to the specifics of heterogeneous systems, or third-party tools such as sys-sage (Vanecek and Schulz, 2023) that aim to capture information about the dynamic environments of HPC systems, can access this information via the API.

3.1. "sysdetect" user API

The new user interface includes the following functions:

- `PAPI_enum_dev_type`: This function enumerates all devices in the system and returns a handle that can be used to access device information through the other two functions.
- `PAPI_get_dev_type_attr`: Returns device type attributes, including the device vendor, the number of devices of that type and vendor in the system, and more.
- `PAPI_get_dev_attr`: Returns device attributes, such as the number of hardware threads for a CPU, warps for NVIDIA GPUs, wavefronts for AMD GPUs, among others.

For instance, for applications like NWChemEx, it is crucial to obtain detailed information about the hardware topology that is exposed to the various algorithms, in order to achieve optimal performance on heterogeneous, multi-vendor systems. The NWChemEx team expressed the desire for the discovery of hardware topology features, occurring at runtime, to be facilitated through a consistent interface. The PAPI interface offered an ideal solution, and the latest version ships with the new user API for accessing comprehensive platform details. Requested features of the hardware topology include the number and type of GPUs on a node, number and type of CPUs on a node, cache sizes and attributes for computational devices, details on which nodes can access shared memory or are located on the same rack, among others.

In summary, NWChemEx now has the capability to utilize PAPI for accessing performance counter data, architectural information about the system, and resource details, which can be leveraged to improve the parallel and performance environment of NWChemEx.

3.2. "sysdetect" command Line tool

An example of the information obtainable from the papi_hardware_avail utility is shown below:

```
$ ./papi_hardware_avail
Device Summary ------------------------------------
Vendor           DevCount
AuthenticAMD       (1)
NVIDIA             (0)
AMD/ATI            (2)

Device Information --------------------------------
Vendor               : AuthenticAMD (2,0x2)
Id                   : 0
Name                 : AMD EPYC 7413 24-Core Proc.
CPUID                : Family/Model/Step. 25/1/1
Sockets              : 2
Numa regions         : 2
Cores per socket     : 24
Cores per NUMA region: 48
SMT threads per core : 2
                     : Size/LineSize/Lines/Assoc
L1i Cache            : 32KB/64B/512/8
L1d Cache            : 32KB/64B/512/8
L2 Cache             : 512KB/64B/8192/8
L3 Cache             : 131072KB/64B/8192
Numa Node 0 Memory   : 257833MB
Numa Node 0 Threads  : 0 1 2 3 4 5 6 7 8 9 10 ...
Numa Node 1 Memory   : 257982MB
Numa Node 1 Threads  : 24 25 26 27 28 29 30 31 ...

Vendor                    : AMD/ATI
Id                        : 0
Name                      : gfx90a
Wavefront size            : 64
SIMD per compute unit     : 4
Max threads per workgroup : 1024
Max waves per compute unit: 32
Max shared memory per wgp : 65536
Max workgroup dim x       : 1024
Max workgroup dim y       : 1024
Max workgroup dim z       : 1024
Max grid dim x            : 4294967295
Max grid dim y            : 4294967295
Max grid dim z            : 4294967295
Compute unit count        : 104
Compute capability        : 1.1
```

4. Software defined events (SDEs)

Software Defined Events (SDEs) are a mechanism provided by PAPI that allows developers of third-party software to expose internal information about their software. PAPI does not dictate what information should be exported; rather, it offers a flexible mechanism for exporting and accessing arbitrary data. This allows the developers, who have the most in-depth knowledge of their software, to decide what information should be exported as an SDE.

To support this functionality, a two-sided approach is necessary. On one side there is the software package that exports the new events and on the other side, there is the entity that reads the events. Both sides can be arbitrary layers of the software stack, such as libraries, runtime systems, complete applications, or external performance monitoring tools. In the rest of this document we will refer to the layer that exports SDEs as the "producer" and we will refer to the layer that reads the SDEs as the "consumer".

The API for exporting SDEs is implemented as a standalone library (libsde), which is distributed as part of the PAPI project and is installed alongside the PAPI library (libpapi). On the other hand, reading SDEs from a consumer does not require any new API. The reading functionality has been implemented as a PAPI component and therefore SDEs can be accessed by a consumer using the same PAPI_start()/PAPI_read()/PAPI_stop() API used for accessing hardware events.

The following subsections describe the different types of events that can be exported using the SDE mechanism, along with examples of their use in third-party libraries and runtime environments. Further details about SDEs, which are outside the scope of this document, can be found in (Danalis et al., 2019; Jagode et al., 2019).

4.1. Registered counters

The simplest and lowest overhead type of SDE is a native program variable registered as a counter. There is only one step required for achieving this. The producer must use the SDE API to declare that a variable will act as the counter for an event and pass a pointer to that variable, along with metadata such as the name of the event, etc. After this step, the producer does not need to call any other SDE API. Since registered counters are native program variables whose values are being modified as part of the producer's normal execution without requiring any SDE-specific API calls, they incur zero overhead. Natural candidates for this type of SDE would be anything that a library already counts but has no standardized way to export to the outside world. For example, the number of iterations executed by an iterative solver until convergence is attained, as found in the numerical linear algebra package MAGMA. Other examples could be the number of elements in a queue, the number of messages sent or received by a communication library, the number of years simulated by a climate simulation application, or the number of insertions, deletions, collisions, etc., in a hash table data-structure.

4.2. Created counters

PAPI supports the notion of *overflow*. That is, a consumer can register a callback function with PAPI and request that this function be invoked when an event counter reaches a specific value. For hardware counters, PAPI can implement this by asking the hardware to call an interrupt handler when the counter reaches the desired value, or, in the case of hardware that does not support this functionality PAPI can periodically poll the value. Similarly, PAPI can poll the value of SDEs implemented as registered counters to allow for overflowing. In addition, we offer a different type of

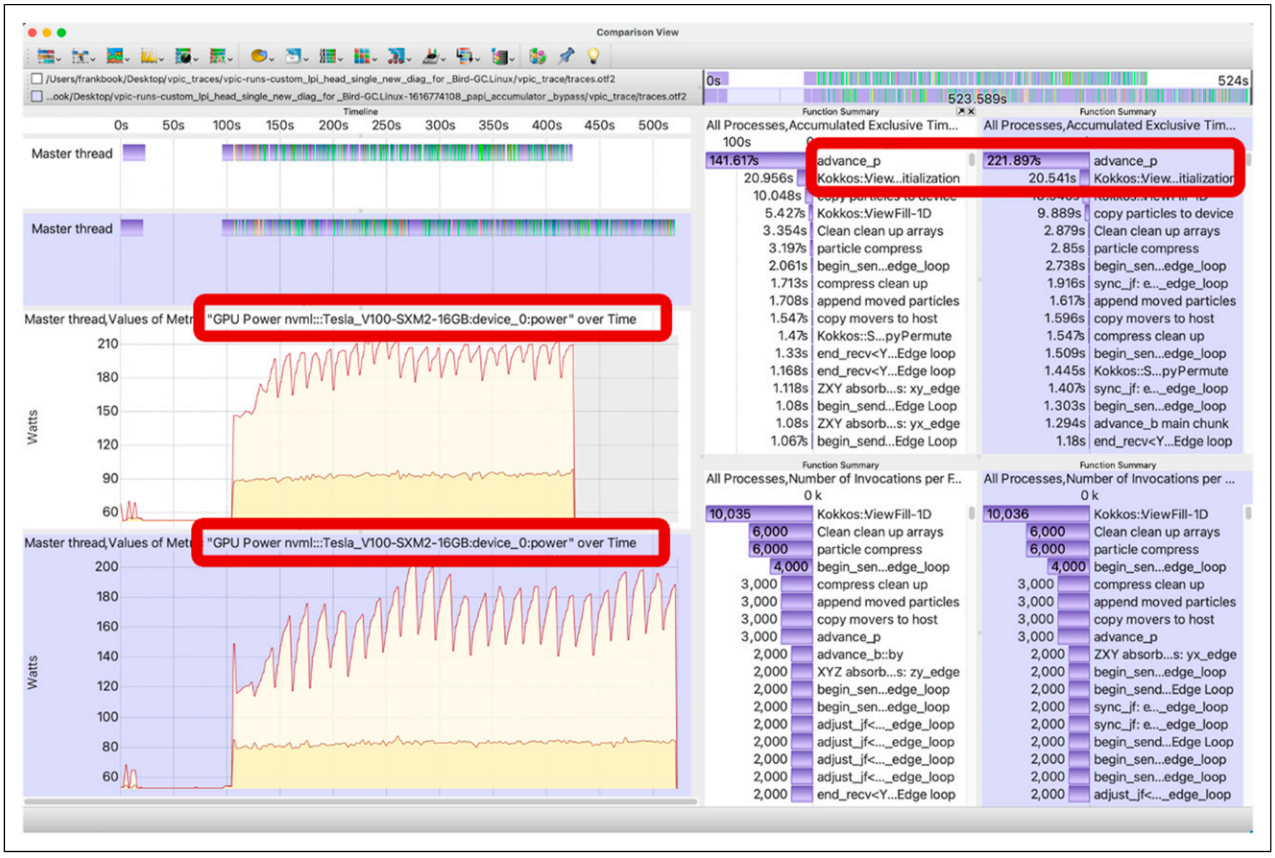

Figure 7. PAPI performance and power measurements of a Kokkos application using Vampir/Score-P on NVIDIA GPUs.

SDE that supports overflowing internally without the need for periodic polling. This type is called *created counter*, and it is implemented as variables created and managed internally by libsde. The main difference compared to registered counters is that these counters are not native variables of the producer. Modifying their value requires libsde API calls. Consequently, updating a created counter incurs some overhead for the producer. On the other hand, since libsde is involved in changing the value of created counters, overflow notifications can be delivered to a consumer accurately and without the much higher overhead of periodic polling. This type of SDE is best suited for events that do not occur very frequently and can benefit from the immediate reaction of an external software layer. For example, the size of a runtime's work queue dropping to zero (signaling idle time), or the receive queue of a communication library being empty, etc.

4.3. Callback counters

The previous two categories associated a program variable with an event counter. *Callback counters* allow libraries to associate a callback function with a counter. This means that the value that will be returned when a consumer reads this type of SDE does not have to exist anywhere in the program's memory. Instead, upon reading the SDE, the callback function will be invoked, allowing the producer to generate complex dynamic values that might not reside in any particular variable. An example would be an event that counts a value that spans multiple threads, with each thread having a partial count of the total number. When the consumer tries to access this SDE, the callback will be invoked, which will compute the sum of the partial values across all threads and return the result to the consumer. This type of event is used by the task scheduling runtime PaRSEC (Bosilca et al., 2013a, 2013b) to export the number of tasks that reside in its "ready" queues. By using a callback counter, instead of a simple registered counter, PaRSEC avoids adding to its critical path costly atomic operations that would be needed for keeping the counters accurate in the presence of task stealing among threads. Instead, the overhead of synchronizing the information kept across different threads is only incurred if a consumer calls `PAPI_read()` to obtain the current value of the counter. Figure 8 shows the values of various SDEs exported by PaRSEC measured at different times during the execution of a program (Danalis et al., 2019).

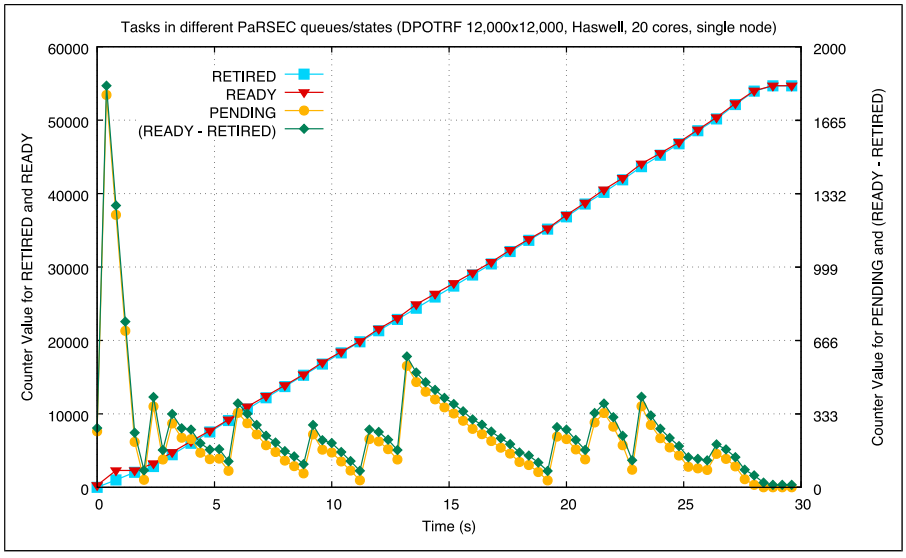

Figure 8. SDEs exported by the task runtime PaRSEC.

4.4. Recorders

Unlike the previous types of SDEs, which all hold one value, *recorders* are multi-value counters. This type of SDE enables libraries to record an arbitrarily long sequence of data. The storage of this data is managed internally by libsde, so the producer can simply "record" the desired data and forget about it. Furthermore, the type of the elements that will be recorded is defined by the producer (i.e., not dictated by libsde), giving the producer maximum flexibility. Similarly to created counters, recorders incur overhead for the producer every time data is recorded. On the other hand, they offer producers the ability to keep track of arbitrary amounts of arbitrary types of data with a simple API call. The mechanism that enables this flexibility involves a hierarchical data-structure that can dynamically allocate additional memory on demand. This approach keeps the amount of unused allocated memory to a minimum, while avoiding the performance overhead of memory copying associated with mechanisms such as `realloc()`. This type of SDE is best suited for data whose evolution over time holds meaning and is worth examining by a consumer. For example, the residual of a matrix across the different iterations of an iterative solver is a natural candidate for a recorder, as is showcased by its use in MAGMA. Figure 9 shows the values stored by a MAGMA-sparse iterative solver using SDE recorders (Jagode et al., 2019).

4.5. CountingSets

A traditional *set* data structure, common in many languages and utility libraries, enables the storage of objects, where each object is unique. A libsde *counting set* also stores unique objects, but it also keeps track of the number of times each object was added to the counting set. In other words, counting sets enable libraries to create histograms of library-specific objects with minimal effort. This functionality is being integrated into the linear algebra package SLATE (Gates et al., 2019). Another possible use of counting sets is for correctness verification. Specifically, a third-party code could use this type of SDE to ensure that all transient objects created by the code have been properly destroyed. For example, if a program adds an entry to a counting set every time it allocates memory (e.g., with "`malloc()`") and removes the entry from the counting set when the memory is released (e.g., with "`free()`"), then at the end of the program the counting set should be empty. The existence of any entries can be used to detect memory leaks.

4.6. Counter groups

Single-value SDEs can be grouped to form new SDEs. This enables a consumer to read the sum, minimum, or maximum value of all counters in a group by treating the counter group as a new counter. Counter groups are first-class citizens and can be recursively combined with other counters or groups into larger groups. This type of SDE is useful when a producer keeps track of information at fine levels of granularity, e.g., per thread, and wishes to expose different levels of granularity to the consumers by grouping individual elements together. An example usage can be found in the task runtime PaRSEC. Specifically, the runtime exports SDEs that hold the number of ready tasks per thread—as we mentioned in Section 4.3. However, this level of granularity is probably too fine for most users. Therefore, these SDE belong to multiple hierarchical groups so that a consumer

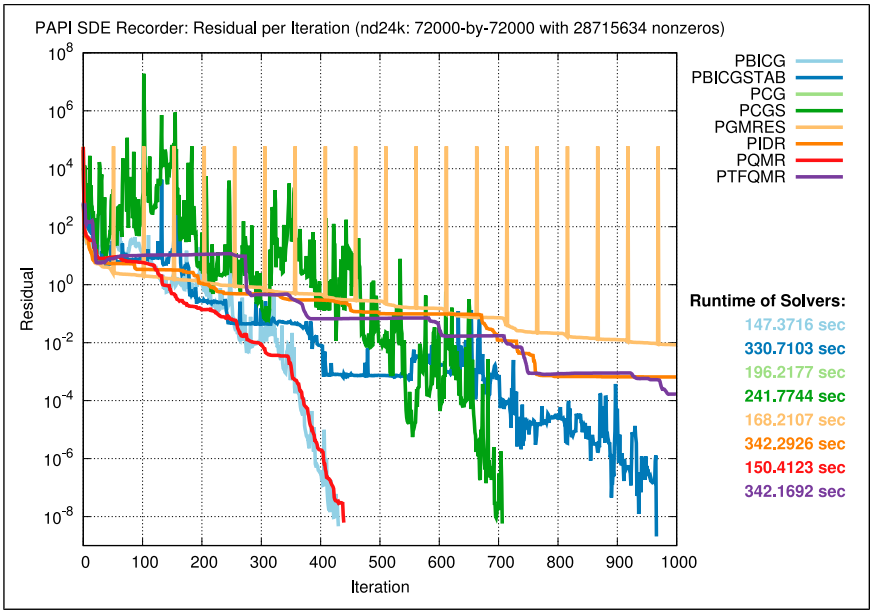

Figure 9. SDEs exported by the linear algebra library MAGMA.

can query the number of ready tasks at a single core by accessing the SDEs directly, the number of ready tasks across all cores of a socket by reading a group of SDEs, or even the total number of ready tasks across all cores in a node by reading the group of all socket-level groups.

5. Counter analysis toolkit (CAT)

The hardware components that comprise Exascale platforms have complex internal structure. As a result, they contain very large numbers of low-level performance events, many of which differ from each other in subtle ways. Furthermore, the names and descriptions of many low-level events are often unclear and sometimes misleading or wrong. To address these challenges, as part of the Exa-PAPI project, we extended the functionality and applicability of the Counter Analysis Toolkit (CAT) (Barry et al., 2021, 2023; Jagode et al., 2016) and released it as part of the PAPI repository. CAT consists of a collection of microbenchmark kernels that are designed to satisfy the following three goals.

- The benchmarks stress-test **specific features** of modern hardware, **each in isolation.** That is, each kernel aims to maximize the use of a specific function of the hardware (e.g., Level-2 cache, Branch Prediction Unit, AVX-256 Floating Point Unit, etc) while minimizing the use of all others. As a result, if we monitor a low-level event E_i while executing a kernel that exercises a particular part of the hardware, we can derive whether the event E_i relates to that part of the hardware.
- The structure of each microbenchmark and the parameters that we use when executing it are chosen such that the **expected behavior** of each kernel is **well understood.** For example, we expect the memory benchmark to cause a specific number of hits and misses on the different levels of the cache hierarchy based on the run parameters we use. Also, the branch kernels are expected to lead to very specific amounts of retired conditional branches and mispredictions, and the floating point kernels are expected to execute specific amounts of vector instructions. Comparing these expectations against measurements of low-level events, we can verify whether an event accurately measures what it is supposed to measure and opens up the way for automatic correlation between low-level events and performance concepts.
- The kernels are written in C. The motivation behind this choice is not only portability but also readability. We made this design decision to enable application developers to read and understand the code so they can understand how parts of their code might affect different hardware components. Also, keeping the code of our kernels simple and readable enables code sharing with other performance analysis experts and promotes collaboration.

5.1. Kernel categories

CAT as a tool compiles into a single executable, `cat_collect`. By selecting the appropriate flags a user can

execute one or more of the different kernel categories that are described below.

5.1.1. Data cache. The core kernel of the data cache benchmark performs a very simple pointer-chasing operation: `p = (uintptr_t *)*p`. However, modern caches have powerful internal logic for performing effective prefetching and replacement policies. As a result, the benchmark's nuance is in the patterns used to create the pointer chain that the core kernel will operate on, and the careful use of threads to apply the correct amount of pressure on the memory subsystem. Furthermore, we have parameterized the access patterns that we use to create the pointer chains so that we can achieve different memory access patterns for each set of parameters. This, in turn, exercises the different levels of the cache hierarchy and the prefetch units in different ways.

Figure 10 shows the measurements collected when monitoring two different low-level events[1] when running the CAT data cache kernel with three different sets of parameters. By modifying the parameter "Block", we modify the working set of the benchmark, which has a direct effect on the behavior of the L2-prefetcher (and therefore an effect on L2-hits for large buffer sizes, as shown in the figure). Similarly, by modifying the stride, we can defeat the "Adjacent line" prefetcher, which also affects L2-hits for large buffer sizes. The exact interplay of these parameters and the measured value are outside the scope of this paper, and more information can be found in (Barry et al., 2021). However, even without these details, the reader can easily see the vastly different pattern of the two events across all three subplots, and the difference across subplots. These complex but deliberate response curves allow us to identify what each event measures and verify that it measures it accurately.

5.1.2. Instruction cache. In the data cache benchmark, discussed above, the number of data elements accessed (and therefore the pressure on the different cache levels) can be programmatically selected and modified during the execution. However, this cannot be done in a reasonable way in the case of the benchmark designed to exercise the instruction cache, because the number of instructions in an executable cannot be modified at run-time as readily as a data buffer. To achieve varying pressure on the instruction cache, our benchmark must contain multiple kernels with different amounts of code each. This way, we can choose which kernel to execute each time. Note that we cannot simply use a loop that executes a kernel multiple times because the instructions contained in the loop will be stored in the instruction cache once, regardless of the number of loop iterations. Instead, we need kernels that contain different numbers of instructions. The kernels of this benchmark are generated via a script and contain different numbers of copies of a basic pattern. This approach has an upper limit however, because kernels with more than tens of thousands of lines of code require a prohibitively high compilation time and quite often lead to a failed compilation. To address this we use dynamic loading. Namely, we build the instruction cache kernels into a dynamic library and we create twelve copies of the library. At run-time, we load all twelve libraries and execute the largest kernel (which has 5000 copies of the basic block) from each of the twelve libraries. This way the instruction cache is presented with kernels that go from 10 copies of the basic block to 12x5000 copies. Consequently, the response curves that come out of executing this benchmark resemble those of the data cache benchmark, with distinct regions of hits and misses in the different levels of the instruction cache hierarchy.

5.1.3. Floating point operations (FLOPs). Modern hardware contains vector units of different lengths. These units contain instructions that perform multiple operations each. This presents a challenge in defining high-level events that measure concepts such as "total floating point operations". The reason is that in order to count the total number of floating point operations we need to measure all the different types of floating point instructions and scale each one appropriately, based on the depth of each vector instruction. This is complicated further by the fact that some vendors do not provide a full set of basic events. For example, the events in

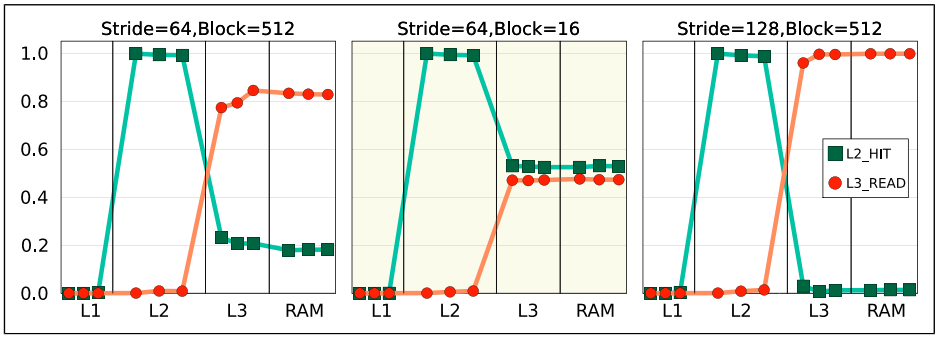

Figure 10. Multi-parameter Data Cache benchmark runs.

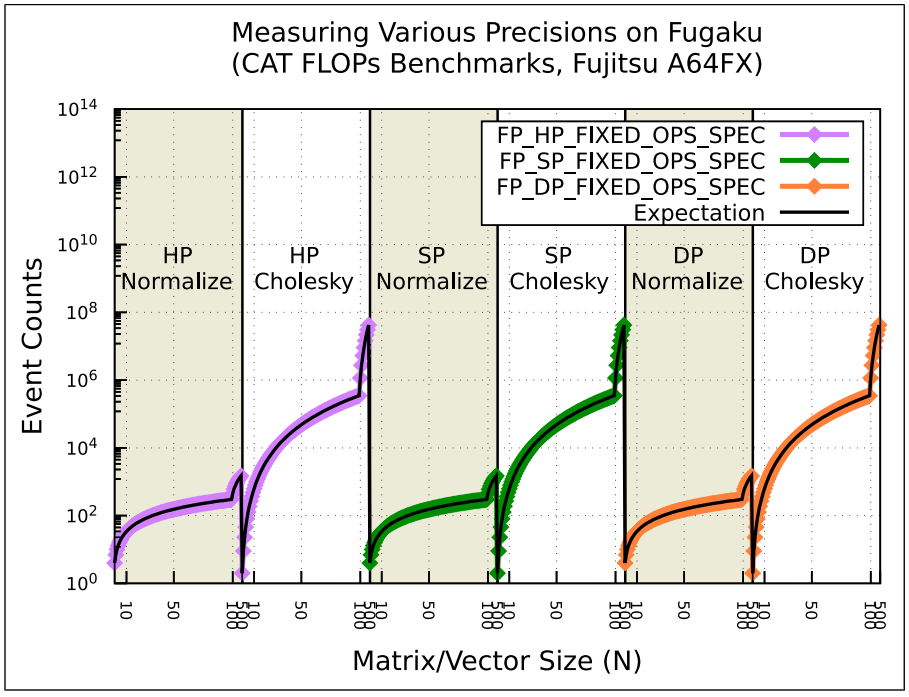

Figure 11. FLOPs events of different precisions.

some AMD processors do not differentiate between single-precision and double-precision instructions, and the events in some Intel processors do not differentiate between FMA (Fused Multiply-Add) and non-FMA instructions.

There are two categories of kernels that exercise this part of the hardware. The first category implements well known linear algebra operations that perform a known number of floating point operations. The second category of kernels utilizes intrinsics that guide the compiler to generate the appropriate vector instructions. In our current version of CAT we support intrinsics for x86, ARM, and POWER architectures. The benchmark includes kernels that span the following orthogonal dimensions:

- Scalar instructions, AVX-128, AVX-256, AVX-512
- FMA and non-FMA
- Single-precision (SP) and double-precision (DP).

In other words, the space, S, covered by our kernels is the following:

S={scalar, 128, 256, 512}× {FMA, non-FMA}× {SP, DP}

In Figure 11 we show the results of the flops benchmark run on the Fugaku system, which enabled us to create preset events for the A64FX architecture.

5.1.4. Branches. The part of the hardware responsible for branches contains some sophisticated logic for dealing with branch prediction. Still, it is among the most straightforward to exercise in ways that yield clean results. This part of CAT

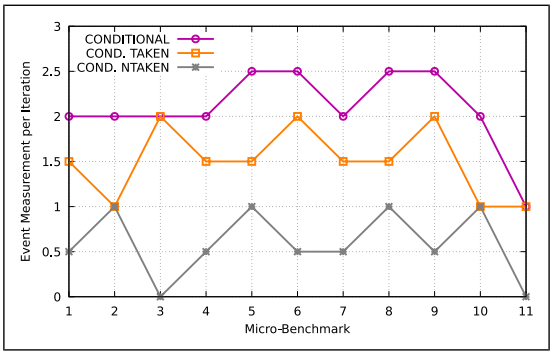

Figure 12. Branch event verification on Sapphire Rapids.

contains eleven kernels, each different from the others in at least one aspect related to branches. Since the kernels are different, if we monitor a low-level event while executing all kernels in sequence, the resulting measurements will form a unique signature that will allow us to differentiate between the concepts of:

- Branches Executed
- Branches Retired
- Branches Taken
- Branches Mispredicted
- Direct Branches

Furthermore, by combining the signatures, we can create composite events, or verify that the event measurements are

```
for(i=0; i<M; i++){
    for(j=0; j<N; j++){
        f64_00 += f64_03;
        f64_01 += f64_00;
        f64_02 += f64_01;
        f64_03 += f64_02;
        ...
    }
}
```

```
for(i=0; i<M; i++){
    for(j=0; j<N; j++){
        f64_00 += f64_100;
        f64_01 += f64_100;
        f64_02 += f64_100;
        f64_03 += f64_100;
        ...
    }
}
```

Figure 13. Kernels with different level of ILP.

accurate. For example, as we see in Figure 12, the "*total number of conditional branches*" matches the sum of "*conditional branches taken*" and "*conditional branches not-taken*".

5.1.5. Instructions. The last benchmark in CAT contains kernels that test different types of instructions. Specifically, the kernels exercise integer, floating point, memory read, and memory write instructions. Some of the operations performed by these kernels overlap with functionality tested by the previously mentioned benchmarks. However, the structure of this benchmark is different from the previous ones. As a result, there are some subtle differences in what is being tested. For example, this benchmark contains kernels with different levels of Instruction-Level Parallelism (ILP). Figure 13 shows two simplified code snippets from our kernels. The code on the left has data dependencies between subsequent instructions, so it cannot be parallelised. In contrast, the code on the right has no dependencies between instructions and can utilize the maximum level of ILP found in a given architecture.

The kernels in this benchmark differ from those in the FLOPs benchmark in one more way. Namely, these kernels do not utilize intrinsics but instead rely on the compiler to detect that the structure of the kernel allows for vectorization. This means that the code of these kernels is more portable and easier to read, but the resulting assembler code is not guaranteed to contain vector instructions. However, the structure of our kernels is chosen so that it is simple enough to be well within the ability of modern compilers to detect as "vectorizable". Figure 14 shows a snippet of code that the compiler can vectorize by using vector instructions twice as wide as the original double-precision variables (such as AVX-128). Underneath the code we show a graphic representation of the memory access pattern when using double-precision and AVX-128. Using kernels like this and controlling the offset of the data dependency ("2" in this example), we force the compiler to use vector instructions of specific widths.

```
for(int j=2; j<BUFFER_SIZE; j++){
    a[j] = a[j-2] + b[j];
}
```

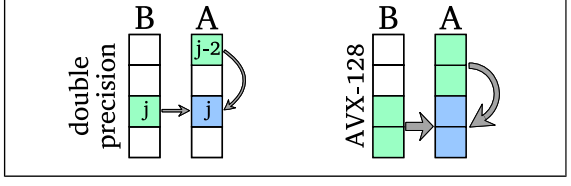

Figure 14. Kernel with loop that can be vectorized.

5.2. Data analysis

If we monitor a native hardware event E_i during the execution of one of the kernels we mentioned above, we will obtain a value. Repeating the measurement for multiple kernels will result in a series of values, such as those shown by each curve in Figures 10 and 12. Such a series of values can be viewed as the measurement vector of E_i. If we repeat this process for all the native events in a target architecture and concatenate the resulting vectors, we produce a measurement matrix A. At the same time, we can handcraft a vector whose values correspond to a performance metric that does not exist on the target architecture. For example, while most modern systems provide native events that count floating-point instructions, they do not provide an event for counting floating-point operations. Nevertheless, since we know how many operations our benchmark kernels perform, we can handcraft a vector b that contains the measurements that a hypothetical floating-point operations event would produce. Solving the linear algebra problem $A \cdot x = b$ would result in a vector x that would tell us which columns of A need to be combined (and scaled) to produce vector b. Therefore, this would tell us which native events in the target architecture to combine (and scale) to construct the missing floating-point operations event.

However, this problem cannot be solved directly, for multiple reasons. First, the matrix A is singular, i.e., it contains columns that are linear combinations of other

columns. Second, the measurements contain noise, which masks linear dependencies and creates bogus dependencies where none exist. Finally, the scale between different events is substantial. Consider, for example, the column containing the values of level-3 cache misses compared to the column with values of idle cycles when running the same memory-intensive kernel. Such large-scale differences would cause traditional matrix "cleaning" (orthogonalization) algorithms to pick irrelevant events in the resulting matrix.

To address these problems in analyzing the measurements that result from our kernels, we deploy several noise suppression techniques, both in our data collection and during post-processing, and utilize specialized orthogonal matrix factorizations that produce the most relevant result for our particular problem. The details of these data analysis steps are outside the scope of this paper, but can be found in (Barry et al., 2024). Using this analysis, we can determine higher level performance metrics, such as PAPI's preset events, in a rigorous and systematic way.

6. Related work

The LIKWID lightweight performance tool suite (Treibig et al., 2010) allows for accessing performance counters by bypassing the Linux kernel and directly accessing hardware performance monitoring registers. While PAPI provides a universal API for accessing hardware performance counters and power management capabilities across various platforms, allowing for fine-grained performance monitoring through integration into applications, LIKWID offers a complementary approach with command-line tools that enable users to collect performance data without the need for user-level instrumentation of applications.

The perf tool (Molnar, 2009) utilizes the perf_event API, which is part of the Linux kernel. While perf_event attempts to provide a generic interface for accessing hardware counters on CPUs on Linux platforms, it is very low-level. As a result, the information returned often requires significant interpretation to be useful for tool developers or end users.

Different vendors provide tools for performance analysis, including NVIDIA Nsight Systems (NVIDIA, 2024b), AMD Omnitrace (AMD, 2024d), and Intel VTune Profiler (Intel, 2024a). These tools are tailored to the specific architectures of their respective vendors and do not offer a standard API for accessing performance data.

A wide range of third-party performance tools rely on PAPI for performance counter and power usage measurements by using PAPI internally as middleware to fetch hardware performance and power monitoring data. They then apply and visualize this information as part of their event records. Some of these tools include Caliper (Böhme et al., 2016), CrayPat (Kaufmann and Homer, 2003), Vampir (Brunst and Knüpfer, 2011), TAU (Shende and Malony, 2006), Scalasca (Geimer et al., 2010), Score-P (Schlütter et al., 2014), and APEX (Huck, 2022).

7. Conclusions

Through the initiatives of ECP, the PAPI product has significantly enhanced its functionality. Notably, PAPI has significantly expanded its performance monitoring capabilities to include support for the latest AMD, Intel, and NVIDIA GPUs, alongside modern interconnects, CPU architectures, and ARM chips. Additionally, PAPI now extends beyond traditional performance counter monitoring to include: (1) support for power management across a variety of state-of-the-art hardware platforms, (2) a new API and a new command line tool for detecting and exposing details of the available hardware, (3) the introduction of SDEs, enabling users to register and monitor new metrics that reveal the internal behavior of libraries to applications using those libraries, and (4) semantic analysis of hardware counters, assisting researchers in interpreting the ever-growing list of events.

These advancements have been seamlessly integrated into the existing PAPI framework, maintaining its traditional interface and methodologies for collecting low-level performance counters across CPUs, GPUs, memory, interconnects, and I/O systems, as well as power management.

Furthermore, PAPI's integration with Spack and E4S not only improves its accessibility and usability on HPC systems but also significantly enhances its reliability due to the implementation of continuous integration and deployment practices. Every new PAPI component ships with Spack "smoke tests" to quickly verify basic functionalities. These tests ensure that the components configured within PAPI are fully operational in the version installed via Spack.

In summary, ECP's support has enabled PAPI to significantly enhance its community impact by serving as a portability layer that consolidates monitoring of hardware counters, power consumption, and software-defined events into a unified open-source software package. By offering a single, portable interface, PAPI facilitates the monitoring of metrics across advanced hardware and software technologies. Because of PAPI, application developers and researchers can avoid the cumbersome necessity of using multiple APIs to access all available counters on a system, thus significantly improving productivity.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This research was supported by the Exascale Computing Project (17-

SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

ORCID iDs

Heike Jagode https://orcid.org/0000-0002-8173-9434
Anthony Danalis https://orcid.org/0009-0002-9846-0066
Giuseppe Congiu https://orcid.org/0009-0008-7165-7591

Note

1. L2 HIT = L2 RQSTS:DEMAND DATA RD HIT, and L3 READ = OFFCORE REQUESTS:DEMAND DATA RD.

References

Agullo E, Demmel J, Dongarra J, et al. (2009) Numerical linear algebra on emerging architectures: the PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180: 012037.

AMD (2024a) AMD's Radeon open compute platform. https://rocm.docs.amd.com/en/latest/.

AMD (2024b) AMD's ROCm system management interface. https://rocm.docs.amd.com/projects/rocm_smi_lib/en/latest/.

AMD (2024c) hipBLAS: basic linear algebra on AMD GPUs. https://rocm.docs.amd.com/projects/hipBLAS/en/latest/.

AMD (2024d) Omnitrace: application profiling, tracing, and analysis. https://github.com/AMDResearch/omnitrace.

Barry D, Danalis A and Jagode H (2021) Effortless monitoring of arithmetic intensity with PAPI's counter analysis toolkit. In: Tools for High Performance Computing 2018/2019, Dresden, Germany, 195–218.

Barry D, Jagode H, Danalis A, et al. (2023) Memory traffic and complete application profiling with PAPI multi-component measurements. In: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), St. Petersburg, FL, USA, 15–19 May 2023, pp. 393–402. DOI: 10.1109/IPDPSW59300.2023.00070.

Barry D, Danalis A and Dongarra J (2024) Automated data analysis for defining performance metrics from raw hardware events. In: IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), San Francisco, CA, USA.

Böhme D, Gamblin T, Beckingsale D, et al. (2016) Caliper: performance introspection for hpc software stacks. In: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 13–18 November 2016, pp. 550–560. DOI: 10.1109/SC.2016.46.

Bosilca G, Bouteiller A, Danalis A, et al. (2013a) PaRSEC: exploiting heterogeneity to enhance scalability. *IEEE Computing in Science Engineering* 15(6): 36–45.

Bosilca G, Bouteiller A, Danalis A, et al. (2013b) Scalable dense linear algebra on heterogeneous hardware. *Advances in Parallel Computing* 24: 65–103. DOI: 10.3233/978-1-61499-324-7-65.

Brunst H and Knüpfer A (2011) Vampir. In: Padua D (ed) *Encyclopedia of Parallel Computing*. New York, NY: Springer US, pp. 2125–2129. DOI: 10.1007/978-0-387-09766-4_60.

Danalis A, Jagode H, Herault T, et al. (2019) Software-defined events through PAPI. In: 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Rio de Janeiro, Brazil, 20–24 May 2019, pp. 363–372. DOI: 10.1109/IPDPSW.2019.00069.

Gamblin T, LeGendre M, Collette M, et al. (2015) The Spack package manager: bringing order to HPC software chaos. In: SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis, Los Alamitos, CA, USA, 15–20 November 2015, pp. 1–12. IEEE Computer Society. DOI: 10.1145/2807591.2807623.

Gates M, Kurzak J, Charara A, et al. (2019) Slate: design of a modern distributed and accelerated linear algebra library. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–18.

Geimer M, Wolf F, Wylie B, et al. (2010) The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience* 22(6): 702–719. DOI: 10.1002/cpe.1556.

Haidar A, Jagode H, YarKhan A, et al. (2017) Power-aware computing: measurement, control, and performance analysis for intel Xeon Phi. In: 2017 IEEE High Performance Extreme Computing Conference (HPEC '17), Waltham, MA, USA, 12–14 September 2017.

Haidar A, Jagode H, Vaccaro P, et al. (2018) Investigating power capping toward energy-efficient scientific applications. *Concurrency Computation: Practice and Experience (CCPE): Special Issue on Power-Aware Computing* 31: e4485. DOI: 10.1002/cpe.4485.

Huck K (2022) Broad performance measurement support for asynchronous multi-tasking with APEX. In: 2022 IEEE/ACM 7th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2), Dallas, TX, USA, 13–18 November 2022, pp. 20–29. DOI: 10.1109/ESPM256814.2022.00008.

Intel (2024a) Intel VTune profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

Intel (2024b) Intel's oneAPI level zero interface. https://spec.oneapi.io/level-zero/latest/index.html.

Jagode H and Hein J (2008) Custom assignment of MPI ranks for parallel multi-dimensional FFTs: evaluation of BG/P versus BG/L. In: 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications, Sydney, NSW, Australia, 10–12 December 2008, pp. 271–283. DOI: 10.1109/ISPA.2008.136.

Jagode H, YarKhan A, Danalis A, et al. (2016) Power management and event verification in PAPI. In: *Tools for High Performance Computing 2015: Proceedings of the 9th International*

Workshop on Parallel Tools for High Performance Computing, September 2015, Dresden, Germany. Cham: Springer International Publishing, pp. 41–51. DOI:10.1007/978-3-319-39589-0_4.

Jagode H, Danalis A, Anzt H, et al. (2019) PAPI software-defined events for in-depth performance analysis. *The International Journal of High Performance Computing Applications* 33(6): 1113–1127. DOI: 10.1177/1094342019846287.

Jagode-McCraw H, Terpstra D, Dongarra J, et al. (2013) Beyond the CPU: hardware performance counter monitoring on blue gene/Q. In: *Proceedings of the International Supercomputing Conference 2013*. Heidelberg: Springer, pp. 213–225. ISC'13.

Jagode-McCraw H, Ralph J, Danalis A, et al. (2014) Power monitoring with PAPI for Extreme scale architectures and dataflow-based programming Models. In: Workshop on Monitoring and Analysis for High Performance Computing Systems Plus Applications (HPCMASPA 2014), Madrid, Spain, 22–26 September 2014, pp. 385–391. IEEE Cluster 2014.

Kaufmann S and Homer B (2003) CrayPat-cray X1 performance analysis tool. In: *Proceedings of the Cray User Group 2003*, pp. 1–32. https://cug.org/5-publications/proceedings_attendee_lists/2003CD/S03_Proceedings/Pages/Authors/Kaufmann.pdf.

Kowalski K, Bair R, Bauman N, et al. (2021) From NWChem to NWChemEx: evolving with the computational chemistry landscape. *Chemical Reviews* 121(8): 4962–4998. DOI: 10.1021/acs.chemrev.0c00998.

Malony A, Biersdorff S, Shende S, et al. (2011) Parallel performance measurement of heterogeneous parallel systems with gpus. In: *Proceedings of the 2011 International Conference on Parallel Processing, ICPP '11*. Washington, DC, USA: IEEE Computer Society, pp. 176–185. DOI: 10.1109/ICPP.2011.71.

Molnar I (2009) perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/.

NVIDIA (2024a) cuBLAS: basic linear algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas.

NVIDIA (2024b) NVIDIA Nsight systems. https://developer.nvidia.com/nsight-systems.

NVIDIA (2024c) NVIDIA's CUDA profiling tools interface. https://developer.nvidia.com/cupti.

NVIDIA (2024d) NVIDIA's Nsight perf SDK. https://developer.nvidia.com/nsight-perf-sdk.

Schlütter M, Philippen P, Morin L, et al. (2014) Profiling hybrid HMPP applications with Score-P on heterogeneous hardware. In: *Parallel Computing: Accelerating Computational Science and Engineering (CSE), Advances in Parallel Computing*. Amsterdam: IOS Press, Vol. 25, pp. 773–782. DOI: 10.3233/978-1-61499-381-0-773.

Shende S and Malony A (2006) The Tau parallel performance system. *International Journal of High Performance Computing Applications* 20(2): 287–311. DOI: 10.1177/1094342006064482.

Terpstra D, Jagode H, You H, et al. (2010) Collecting performance data with papi-c. In: Müller MS, Resch MM, Schulz A, et al. (eds) *Tools for High Performance Computing 2009*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 157–173.

Treibig J, Hager G and Wellein G (2010) LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: Proc. Of the First International Workshop on Parallel Software Tools and Tool Infrastructures.

Vanecek S and Schulz M (2023) Sys-sage: a fresh view on dynamic topologies & attributes of HPC systems. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Poster Session.

Willenbring J, Shende S, Spear W, et al. (2023) E4S: extreme-scale scientific software stack. https://www.osti.gov/biblio/2432176.

Author biographies

Heike Jagode is a Research Associate Professor at the Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville (UTK). She has led the ICL Performance Group since 2013 and specializes in high-performance computing, with a focus on performance analysis, tuning, and energy efficiency of parallel scientific applications. She holds a Ph.D. in Computer Science from UTK, where she was advised by Jack Dongarra, an M.S. in High Performance Computing from the University of Edinburgh, UK, and an M.S. in Applied Mathematics from the University of Applied Sciences Mittweida, Germany.

Anthony Danalis is a Research Assistant Professor at the University of Tennessee, Knoxville. His research interests include performance measurement and optimization, systems, compiler analysis, MPI, and accelerators. He holds a Ph.D. in Computer Science from the University of Delaware, an M.S. in Computer Science from the University of Delaware, and an M.S. in Computer Science from the University of Crete, Greece.

Giuseppe Congiu is a consultant Research Scientist at Innovative Computing Laboratory (ICL) at the University of Tennessee, Knoxville, where he works on performance measurement and modeling. Before joining ICL he was a Postdoctoral Fellow at Argonne National Laboratory, where he worked on Programming Models and Runtime Systems for large scale supercomputing clusters. He holds a Ph.D. in Computer Science from the Johannes Gutenberg University of Mainz, Germany, and a B.Sc. and M.Sc. in Electrical and Electronic Engineering from the University of Cagliari, Italy.

Daniel Barry is a Data Science and Engineering Ph.D. student at the University of Tennessee, Knoxville (UTK), advised by Jack Dongarra. He conducts research with the Innovative Computing Labs Performance Group. Daniel received a Bachelor of Science in Computer Engineering from UTK. His research interests include performance

monitoring technology, benchmarking methodologies, application optimization, and numerical methods.

Anthony Castaldo holds an M.S. in Mathematics (2005) from Texas A&M University, an M.S. in Computer Science (2007) from the University of Texas, and a Ph.D. in Computer Science (2010) from the University of Texas, where he was advised by R. Clint Whaley. His research focused on developing new high-performance computing algorithms.

Jack Dongarra specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced computer architectures, programming methodology, and tools for parallel computers. He holds appointments at the University of Manchester, Oak Ridge National Laboratory, and the University of Tennessee. In 2019, he received the ACM/SIAM Computational Science and Engineering Prize. In 2020, he received the IEEE-CS Computer Pioneer Award. In 2022, he received the ACM A.M. Turing Award for his pioneering contributions to numerical algorithms and software that have driven decades of extraordinary progress in computing performance and applications. He is a Fellow of the AAAS, ACM, IEEE, and SIAM; a foreign member of the British Royal Society and a member of the U.S. National Academy of Sciences and the U.S. National Academy of Engineering.