Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2020 SCPE. Volume 21, Issues 4, pp. 689–699, DOI 10.12694:/scpe.v21i4.1807

NVIDIA GPU PERFORMANCE MONITORING USING AN EXTENSION FOR DYNATRACE ONEAGENT

TOMASZ GAJGER *

Abstract. This work presents a Dynatrace OneAgent extension for gathering NVIDIA GPU metrics using NVIDIA Management Library (NVML). The extension integrates GPU metrics into an industry-leading platform for Application Performance Management extending its capability of monitoring important business workloads to the GPU-oriented computational nodes. A practical approach for acquiring and processing NVML metrics via Python bindings is described. The work also proposes and discusses implementation of helper applications for convenient simulation of performance problems in a multi-tier web application. These applications are then used in combination with OneAgent-based monitoring and appropriate configuration of Dynatrace platform for web application monitoring. Next, an end-to-end production-like scenarios are presented, which exemplify extension usefulness in test setup resembling a real world implementation. The extension has been released on GitHub under MIT license.

Key words: Application Performance Management, GPU Performance Monitoring, Dynatrace, GPGPU, CUDA, NVML

AMS subject classifications. 68M20, 68W10

1. Introduction. The drive towards using GPUs for various, general-purpose tasks (GPGPU paradigm) has been growing stronger over recent years. Not only are the GPUs utilized in HPC applications, but an increasing number of software companies start to leverage them for running computational workloads in their backend systems [30]. This evolution is fueled by large enterprises leveraging machine learning methods to provide service to their customers or analyze business-relevant data [7], and backed by extensive research on improving GPUs performance and efficiency [22]. Examples of such workloads are: machine learning, big data, blockchain processing, database queries acceleration, and more. As the GPUs become a vital piece of modern IT infrastructure, a challenge arises to have a reliable and convenient approach for monitoring their performance. While most of the Application Performance Management (APM) vendors offer a robust support for resources like CPUs, RAM, network, hard disks, etc. in their portfolio, same cannot be said for GPUs. Some integrations and dedicated tools exist (see Section 3), still there is yet much to uncover and develop in this area.

This work presents a Dynatrace OneAgent extension for gathering NVIDIA GPU metrics using NVML[1]. Dynatrace is one of the industry leaders in APM [17, 15, 21]. The extension augments Dynatrace platform by feeding it with GPU data for processing by its AI engine for continuous performance tracking and automatic *root cause analysis* (RCA).

Considering NVIDIA's dominance in the field of GPGPU, it was decided to focus initial efforts on said hardware. For example, 27% of supercomputers on the TOP500 list [2] are using these, up from 20% in 2018 or 13% in 2016. Industry has also adopted NVIDIA GPUs for various purposes [30]. In future, the extension could be modified to cover additional types of devices, e.g. these offered by AMD or Intel.

The main contribution of this paper is in extending Dynatrace platform with GPU monitoring capability and making the source code for the extension freely available. Proposed extension is a valuable addition, enabling the platform to support novel use cases in an emerging, dynamically-expanding market. Considering existing APM tools landscape, this is an innovative approach and can serve as a base for further research and development.

The remainder of this paper is organized as follows: Section 2 explains important aspects related to the GPU performance monitoring and briefly describes the Dynatrace product. Section 3 lists related work, while in

*Department of Computer Architecture, Faculty of Electronics, Telecommunications and Informatics, Gdańsk University of Technology, Narutowicza 11/12, 80-233 Gdańsk, Poland (tomasz.gajger@pg.edu.pl)

[1] https://github.com/tomix86/oneagent-nvml-extension

Section 4 proposed extension design and implementation details are explained. Section 5 describes the testbed, testing methodology, and presents the results. Finally, Section 6 summarizes results and outlines the proposed direction of future research.

2. Background.

2.1. GPU Performance Monitoring. Application execution can be monitored either from the outside (*shallow monitoring*) or from within (*deep monitoring*) the process running in the system. In case of **shallow monitoring** the performance metrics are gathered using system, driver or programmatic counters (offered by software framework), or alternatively a logic for exposing them needs to be incorporated up-front into the application during development. For CPU-centric applications, the monitoring focuses on observing measurable effects of the application execution, e.g. CPU usage, RAM consumption. For NVIDIA GPU-based applications, such metrics are collected by CUDA Runtime and are queryable via NVML API [31].

Deep monitoring methods work by instrumenting the application code and extracting execution (performance) metrics from it. For NVIDIA GPUs this type of API is provided by CUDA Profiling Tools Interface (CUPTI) [29]. They usually allow for collection of more fine-grained metrics at the cost of increased overhead, that varies depending on the type of metric being collected. We can distinguish two basic paradigms for instrumenting application code: static (compile-time) and dynamic (run-time). **Static methods** require the instrumentation code to be incorporated into the application during its development, either via direct calls to the API of interest (e.g. CUPTI) or in a form of SDK (e.g. OneAgent SDK for C++ [8]). Applying them requires the monitored application to be recompiled. **Dynamic methods** work after the application was already developed and compiled, often by intercepting certain library (e.g. `ltrace`) or system (e.g. `strace`) calls and/or inserting specialized instrumentation code directly into the application being monitored (e.g. deep monitoring provided by OneAgent). In case of languages that are compiled just-in-time (e.g. CUDA PTX into SASS) it is possible to recompile application code in runtime and insert instrumentation code this way. This is something in-between static and dynamic instrumentation, technically it is dynamic since no explicit code changes are required, but on the other hand the application code representation changes significantly. Hence, two subtypes of dynamic methods can be named: **recompiling** and **automatic**.

The challenge with *static* and *dynamic recompiling* methods is that frameworks or toolkits often ship with already compiled kernels, what does not allow for code changes. Even with access to framework's code, modifying it is not really feasible for any serious production use case, it is simply too cumbersome and time consuming for anyone to bother. Use of *dynamic recompiling* method would incur additional overhead during initial kernel launch as it would have to be compiled from PTX into SASS (see Section 2.1.1 of [16]). The significant benefit of *shallow* and *dynamic automatic* methods is that they do not require this recompilation step and thus provide out of the box support for existing applications. *Static* methods are inferior to other types when it comes to ease of use and coverage for existing applications. While it is apparent that *dynamic automatic* methods are best in terms of usability and provided level of detail, it comes at a cost of difficult development process and increased overhead. *Shallow* methods are good because of their lack of overhead, ease of use, easy development, and best applicability compared to others. Their only drawback is that they offer a considerably limited view into the application behavior.

2.2. NVML. NVML [31] is a C-based API for monitoring and managing NVIDIA GPUs. It enables developers to build applications on top of it and has bindings to several other languages, including Python [28]. NVML, among others, allows to retrieve: list of processes running on a GPU, global memory utilization and current clock rates. It does not allow to retrieve detailed information related to the performance of the kernel, occupancy, SM utilization, and alike.

2.3. Application Performance Management. Application Performance Management (or Monitoring) [19] encompasses software products comprising digital experience monitoring (DEM), application discovery, topology mapping, tracing, diagnostics (identifying faults and aiding in resolving them [3]), integrations into CD pipeline, business analytics, and purpose-built artificial intelligence for IT operations and application developers alike.

Dynatrace is one of vendors delivering APM solutions. It is not only the company name but also the name of the product that they offer - a software platform [11] for monitoring and managing performance of

applications, digital experience and business analytics. **OneAgent** [10] component is a piece of the platform that users install on their operating systems. The agent gathers infrastructure metrics, monitors log files, detects processes and instruments them. The performance data is then reported to the **Dynatrace Cluster**. **OneAgent SDK** [8] may be used to add process-level request tracing to applications not supported by the OneAgent natively. **OneAgent Extensions** offer a mechanism to extend the collection of metrics OneAgent gathers by writing scripts in Python. An SDK [9] is provided that is a base for writing custom extensions that developers may use to collect, process, and send the data to the Dynatrace Cluster. Within the cluster a component called **Davis** operates, which is a deterministic AI causation engine. It analyzes all incoming data, including but not limited to: process and infrastructure metrics (also the ones from custom extensions), application topology, log files, performance and availability events. By leveraging automatic baselining, it detects anomalies in metrics and if such occur, or in case an unexpected event is encountered, it reports a *problem* informing the system operator about the issue at hand. On top of that, using a deterministic algorithm it performs an RCA for each given problem, pointing at the chain of events that led to it, its root cause and impact on end user. **Synthetic and real user monitoring** [12] are key elements for the DEM offered by the Dynatrace platform. Both components are used to monitor actual performance of web pages, the former one by configuring and sending pre-created web requests to analyze their behavior based on received responses. The latter one gathers statistics from real user visits on webpage in question as they traverse throughout the application stack.

3. Related work and software. This section presents related work relevant to the paper subject, a differentiation is made between software build purposely to gather GPU metrics and support for these in scope of generic APM products.

3.1. GPU metrics gathering software. Two basic types of GPU metrics gathering software can be recognized, one that operates in scope of a single host, and distributed solutions providing a centralized access to metrics coming from an arbitrary number of hosts.

Single host can be monitored using NVIDIA-made utility called `nvidia-smi` [27], which wraps NVML in a convenient command-line interface. There are various similar tools, most popular of which seem to be `gputstat` [5] and `NVTOP` [32].

Distributed monitoring is offered by NVIDIA via `DCGM` [25], also a wrapper over NVML, but able to work in a clustered environment. It offers storage for historical data, batch queries, healthchecks, and diagnostics. Integrations for following third-party products are available: `Prometheus`, `Grafana`, `IBM Spectrum LSF`, and `collectd`. `Bright Cluster Manager` [4] is a proprietary product that leverages DCGM underneath. On the other hand, `Ganglia` [1], builds directly on top of NVML, leveraging its Python bindings [28], while `Influxdata Telegraf` [20] uses `nvidia-smi` to collect the metrics. The authors of [14] propose an extension for existing `Integrated Performance Monitoring` toolset to monitor GPUs by dynamically instrumenting application code to gather timing metrics, and have proven its usefulness in a multi-node GPU cluster.

3.2. GPU monitoring support in APM products. In current APM offering, the following products offer a built-in (or opt-in via extension) support for GPU monitoring. `Google Cloud operations suite` available on Google Cloud Platform (GCP) [18]. `New Relic` provides support to a limited extent by using metrics coming from GCP Kubernetes Engine [24]. The case is similar for `Datadog` [6], which is able to use metrics coming from other providers (GCP, Kubernetes, Mesos, Amazon SageMaker, Alibaba Cloud), there are also user-made plugins for it [23] - an approach similar to the one proposed in this paper.

4. Extension design and implementation. The extension's implementation leverages Python bindings for NVML [28]. It is is capable of monitoring multiple GPUs, the metrics coming from all the devices are aggregated and sent as combined time series. There is no support for sending separate time series per device.

4.1. Metrics. Once Dynatrace OneAgent is installed and extension is deployed, NVML metrics are periodically gathered and sent to the Dynatrace Server. *HOST* metrics are reported for the host entity, while *PGI* metrics are reported for process entity. PGI is an acronym for *Process Group Instance* and, in simplification, denotes a process running in a system. The list includes:

- `gpu_mem_total` (HOST) - total available global memory,

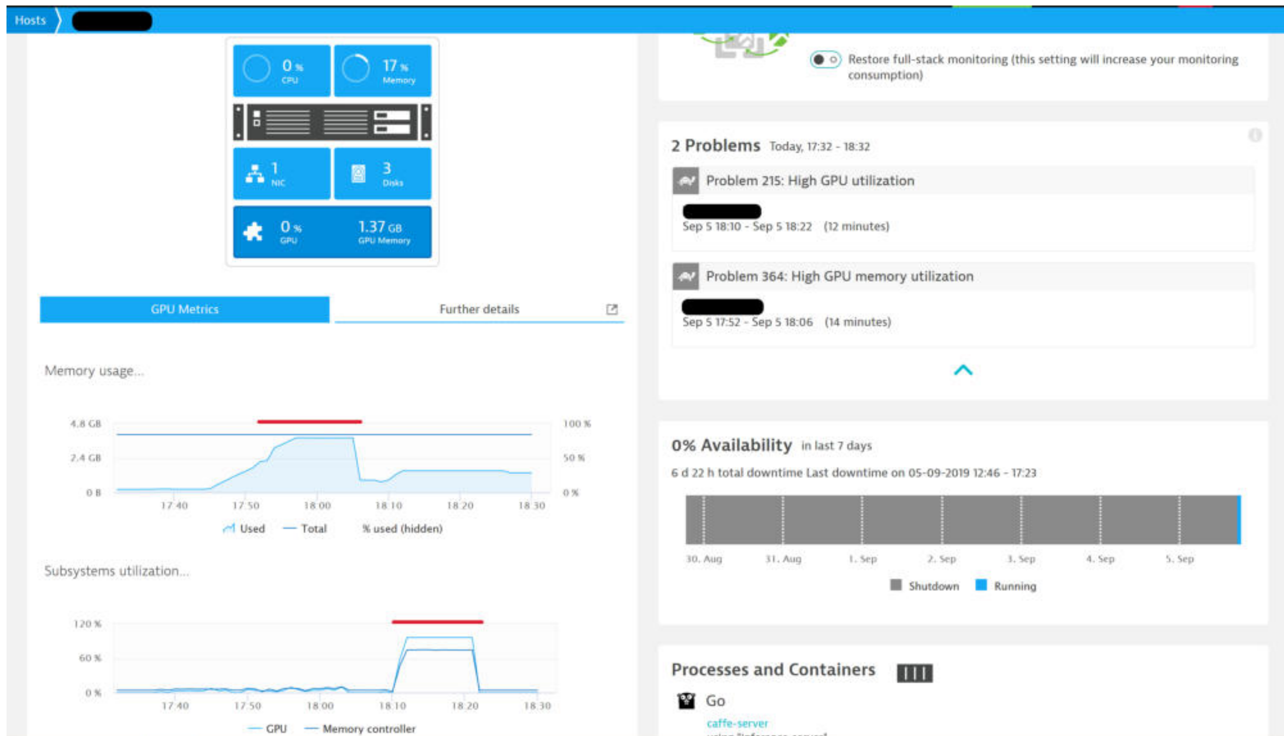

Fig. 4.1. *Host metrics reported by the extension*

- `gpu_mem_used` (HOST) - device (global) memory usage,
- `gpu_mem_used_by_pgi` (PGI) - global memory usage per process,
- `gpu_mem_percentage_used` (HOST) - artificial metric for raising *High GPU memory* alert,
- `gpu_utilization` (HOST) - percent of time over the past sample period (within CUDA driver) when a kernel was executing on the GPU,
- `gpu_memory_controller_utilization` (HOST) - percent of time over the past sample period (within CUDA driver) when global memory has been accessed,
- `gpu_processes_count` (HOST) - number of processes making use of the GPU.

If there are multiple GPUs present, the metrics are displayed in a joint fashion:
- `gpu_mem_total` is a sum of all the devices' global memory,
- `gpu_mem_used` and `gpu_mem_used_by_pgi` is the total memory usage across all the devices,
- `gpu_utilization` and `gpu_memory_controller_utilization` is an average from per-device usage metrics,
- `gpu_processes_count` shows unique count of processes using any of the GPUs. That is, if a single process is using two GPUs, it is counted as one.

These metrics can be used to observe GPU performance on the system in question:
- GPU core utilization, see Fig. 4.1,
- GPU memory subsystem utilization, see Fig. 4.1,
- Per-process GPU memory usage, see Fig. 4.2,
- Automatic notifications about performance problems (e.g. in case GPU usage exceeds predefined threshold), see Fig. 4.3.

Metrics are automatically correlated with particular hosts and processes, and thus are used by the Davis for evaluating root cause of performance problems. See Fig. 4.4 for an example where metrics anomalies are reported.

In the current extension version (v0.2.0), the following NVML device queries are utilized to gather metrics:

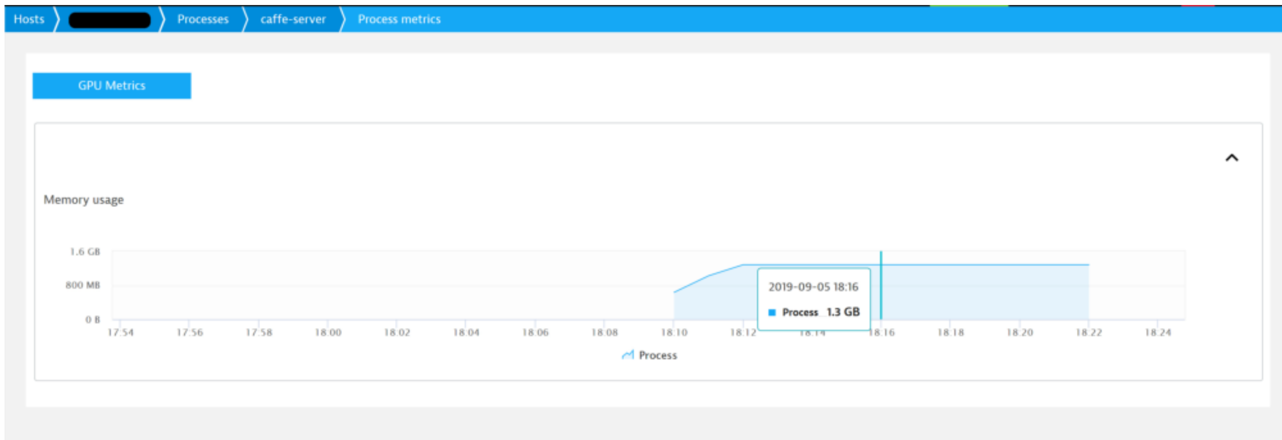

FIG. 4.2. *Process metrics reported by the extension*

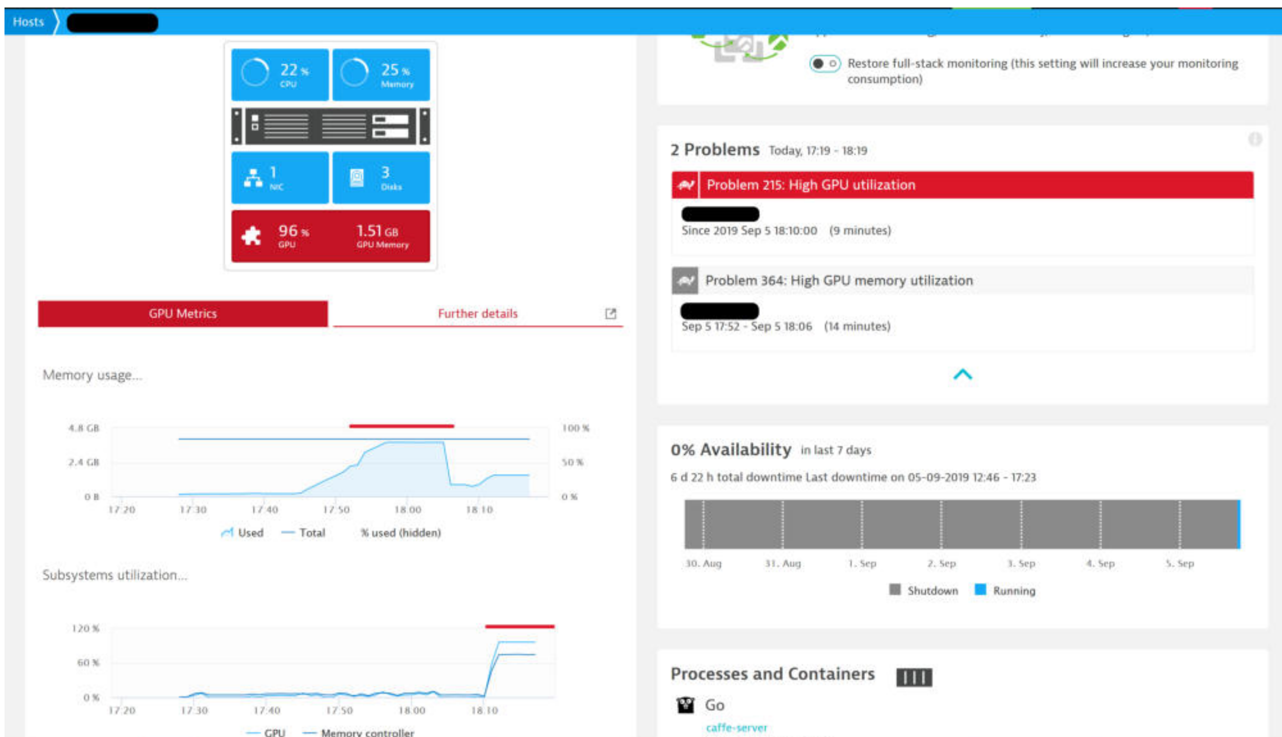

FIG. 4.3. *Alerts as seen on the host screen*

- `nvmlDeviceGetComputeRunningProcesses` - get information about processes with a compute context (non-graphics) on a device,
- `nvmlDeviceGetGraphicsRunningProcesses` - get information about graphics-based processes,
- `nvmlDeviceGetMemoryInfo` - get the amount of used, free and total memory available on the device,
- `nvmlDeviceGetUtilizationRates` - get the current utilization rates for the device's major subsystems: graphics unit and global memory.

Gathering metrics externally via NVML (contrary to CUPTI) does not incur any additional overhead on the observed applications. Internally, the extension collects several data samples and aggregates them before passing them on to the framework execution engine. By default, 5 samples in 2 second intervals are collected.

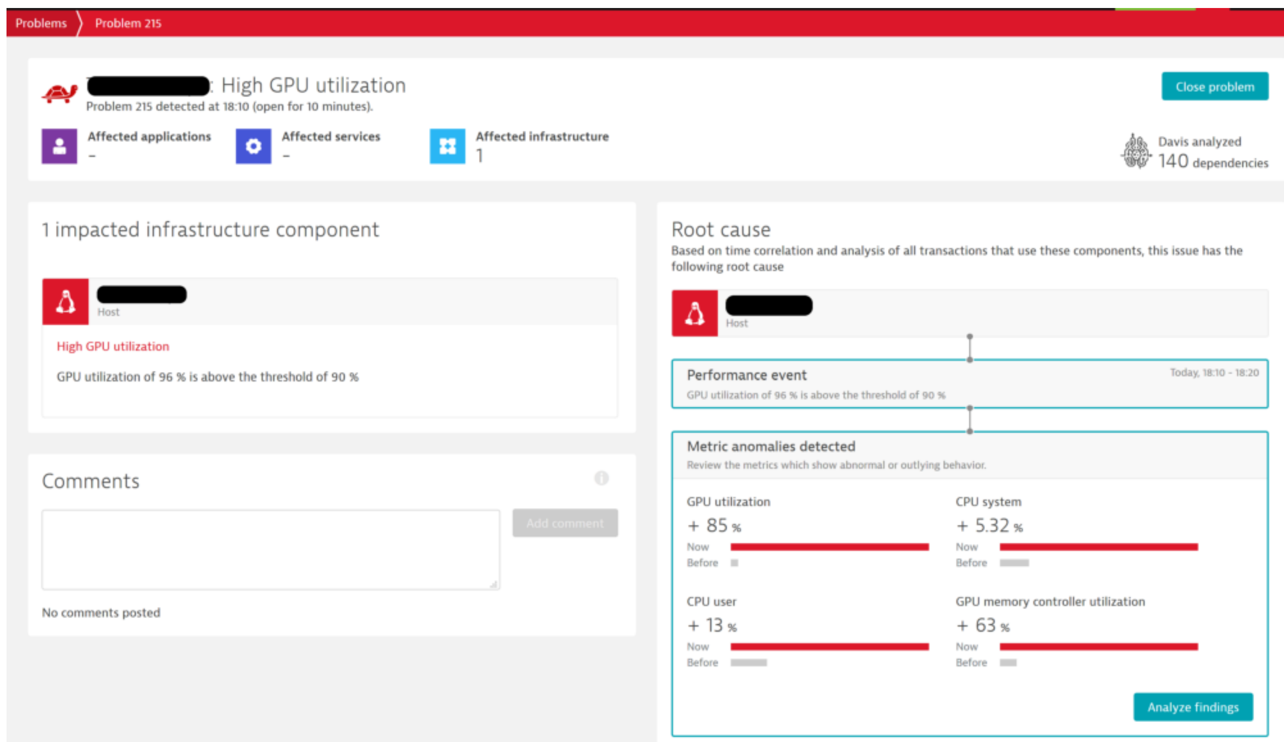

Fig. 4.4. *Problem screen with high GPU utilization alert*

These values are customizable.

Concerning per-PGI memory usage, on Windows this metric won't be available if the card is managed by WDDM driver, the card needs to be running in TCC (WDM) mode. This mode is not supported by GeForce series cards prior to Volta architecture [26].

4.2. Alerting. Three alerts are predefined in the extension, all three are generated by Davis when metrics exceed certain threshold values. These alerts are reported for the host entity and are visible on the host screen (see Fig. 4.3):

- *High GPU utilization* alert - raised when `gpu_utilization` exceeds predefined threshold (default: 90%) in given time period,
- *High GPU memory controller utilization* alert - raised when `gpu_memory_controller_utilization` exceeds predefined threshold (default: 90%) in given time period,
- *High GPU memory utilization* alert - raised when `gpu_mem_percentage_used` exceeds predefined threshold (default: 90%) relative to `gpu_mem_total` in given time period.

Alerts thresholds are customizable on the WebUI. Note that high GPU memory utilization alert is based on two separate metrics, due to current extension framework limitations, it is not possible to define such alert server-side. Thus, an artificial metric that is hidden on the memory usage chart, representing percentage usage of the GPU memory had to be introduced.

5. Experiments and results.

5.1. Helper applications. Following helper applications were used to aid with testing, two of which were developed for the purpose of this work:

- A C++ application simulating heat distribution in a two dimensional solid body[2], further referred to as *Backend Application*,

[2] `https://github.com/tomix86/webserver-test-app`

- A load generator[3] capable of using arbitrary amount of GPU global memory with control over how the usage progresses in time,
- glmark2 [13], launched to run infinitely: `$ glmark2 --run-forever`.

Backend Application is a simple webserver, exposing REST endpoints, that responds to queries with image depicting computed result. A sample query looks as shown in Listing 1, where subsequent GET parameters stand for: CUDA block size X, CUDA block size Y, 2D body mesh side length, algorithm iterations.

```
<address>/heat-distrib?16&32&100&1000
```

LISTING 1
Example query

Internal implementation leverages CUDA, NVTX and C++ REST SDK. It simulates heat distribution in a solid, where finite difference method was used to discretize steady state differential equation describing heat diffusion in the object. Environment temperature, which is a boundary condition, was assumed to be constant. Additionally, the object is assumed to be a two-dimensional square mesh, with a side of size N, so there are $N \times N$ distinct points on the mesh for which the value of the temperature has to be updated during each simulation step. The final equation, applied to each cell is a five-point 2D stencil.

5.2. Testbed setup. To showcase extension capabilities, a testbed was prepared that exemplifies, in a simplified form, a production-like environment. It consists of:

1. Host running Apache2, with a public IP address, acting as both a *Frontend Server* and reverse proxy that exposes the webpage to the internet, monitored by OneAgent,
2. *Backend Server*, monitored by OneAgent with NVML extension installed, running *Backend Application* instrumented via OneAgent SDK,
3. Machine with *ActiveGate* installed and *Synthetic monitoring* (a private location [12]) enabled:
   - *Browser monitor* [12] configured, by recording a clickpath, to query http://<*Frontend Server* IP>/heatpage.html. This is a webpage that upon load fetches additional resource from `/heat-distrib` endpoint exposed by *Backend Server*,
   - *HTTP monitor* [12] configured to query http://<*Frontend Server* IP>/heat-distrib, which goes via reverse proxy to the *Backend Server*.

Browser monitor and *HTTP monitor* are jointly referred to as *Synthetic monitoring*.

Above setup contains 3 essential components of a modern web application: a frontend page, a webserver serving said page, and a backend host running computations. Hence, they can act as a minimal, but still an accurate setup where extension usefulness can be proven. It is not very different from, let's say, a webpage with an interactive voice assistant, which needs to query a backend underneath that performs natural language recognition using machine learning methods on GPU-based computational nodes. If performance of such assistant would be impaired, it would affect real users of said webpage and could thus lead to profit loss and customer dissatisfaction.

5.3. Tests. Two tests were conducted, both of which present an end-to-end monitoring scenario with causality analysis when a problem occurs. *Synthetic monitoring* is used to measure *Frontend Server* response times, while NVML extension feeds GPU data into Davis. The problems are generated by impairing the GPU performance on *Backend Server* causing **increased response times** and **request processing failures** respective for test cases one and two. For both scenarios an RCA for performance degradation measured via *Synthetic monitoring* is shown.

5.3.1. Test case 1. This case is an end-to-end monitoring scenario, where *Backend Application* is responding slower due to high (close to 100%) GPU core utilization, with load generated by `glmark2`.

As shown in Fig. 5.1, the problem screen displays a causality analysis and pinpoints the root cause to be a malfunctioning computational node. Note the metric anomalies being detected, these are coming from instrumentation via OneAgent SDK. One can then drill-down to the problem analysis (Fig. 5.2) and see that the extension has detected the issue, with the cause clearly identified to be high GPU utilization, as shown on the host screen (Fig. 5.3). The scenario presented here shows data from host GPU metrics reported by extension

[3] `https://github.com/tomix86/cuda-load-generator`

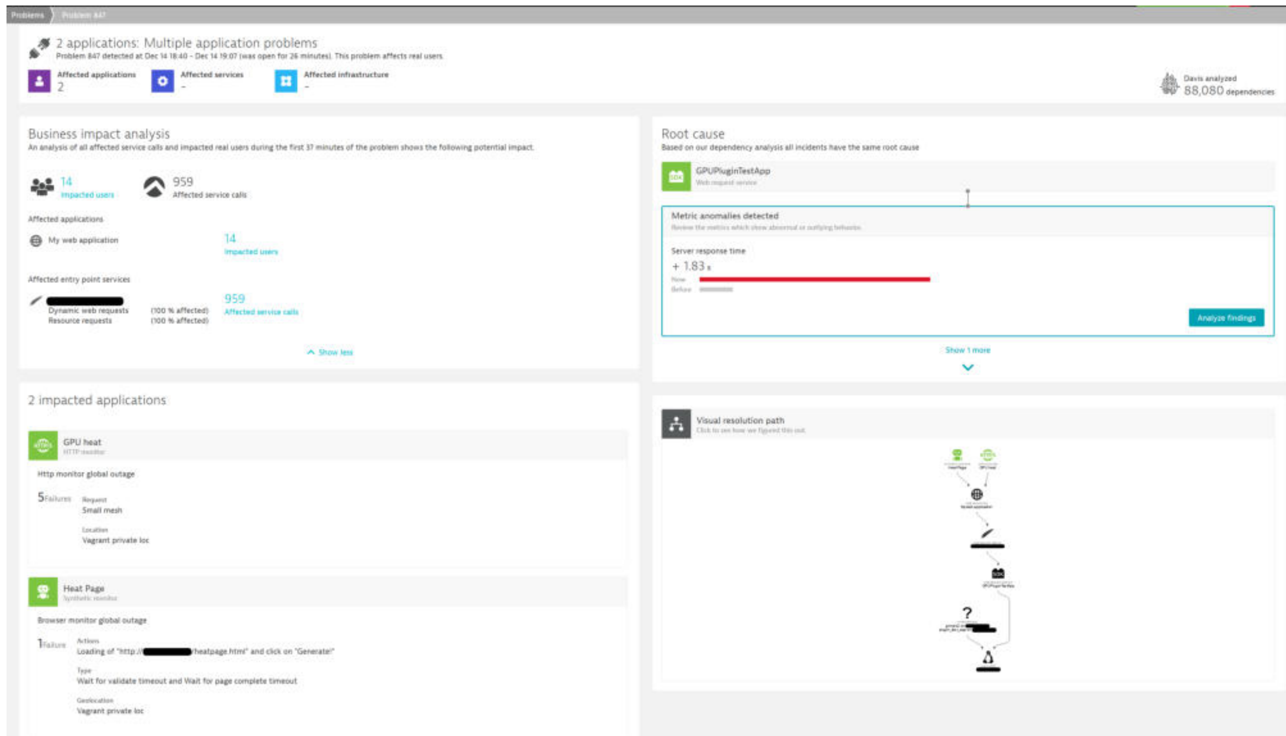

Fig. 5.1. *Detected performance problem (left) and RCA (right)*

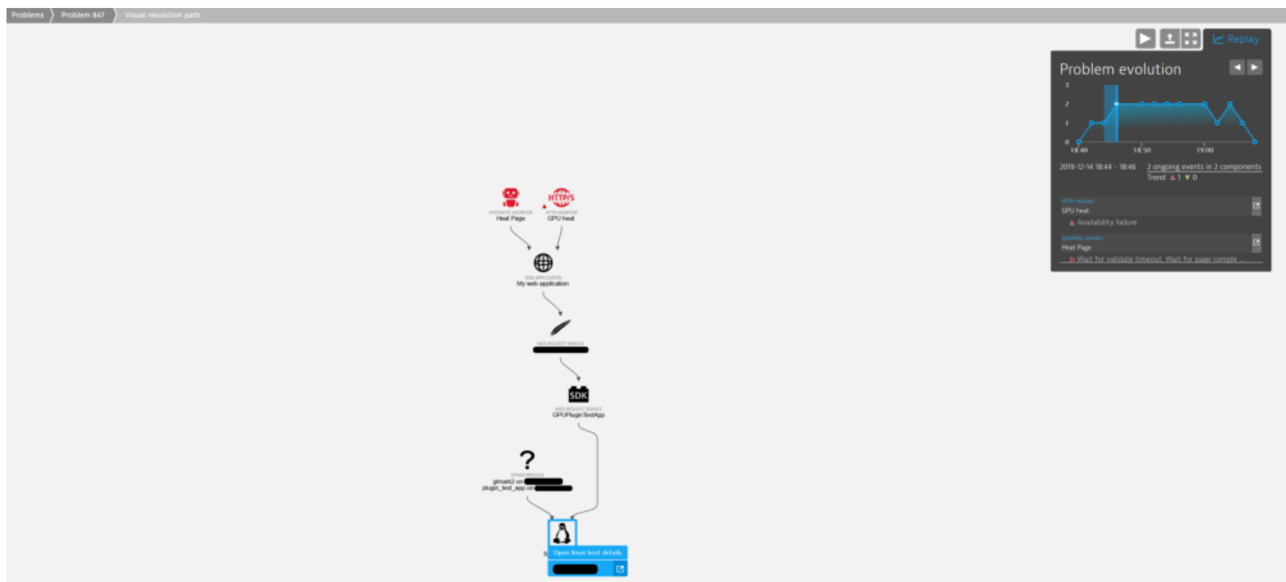

Fig. 5.2. *Problem analysis screen with visual resolution path*

being automatically correlated to availability issues in web application. In future version of the Dynatrace platform an improvement could be made such that the problem screen (Fig. 5.1) would directly display the metrics from extension, effectively providing accurate RCA without a need to perform additional steps - such as host screen inspection.

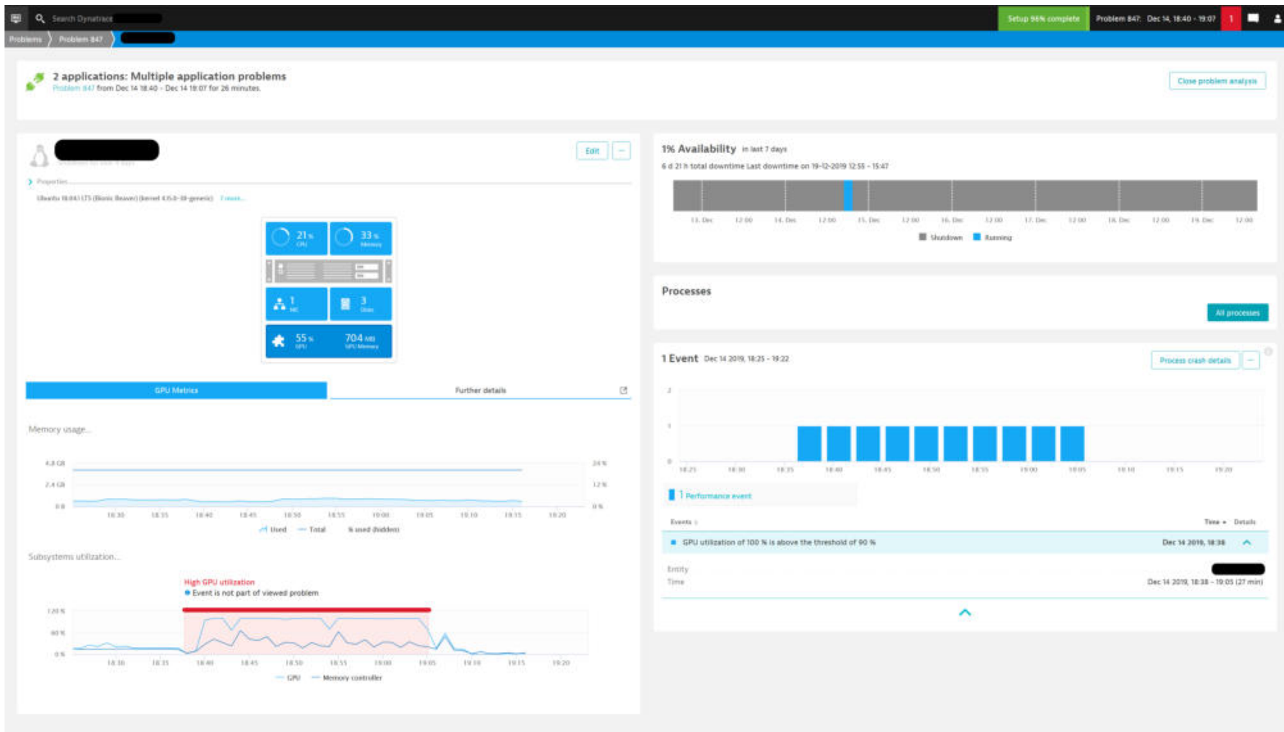

Fig. 5.3. *Host screen with high GPU utilization alert*

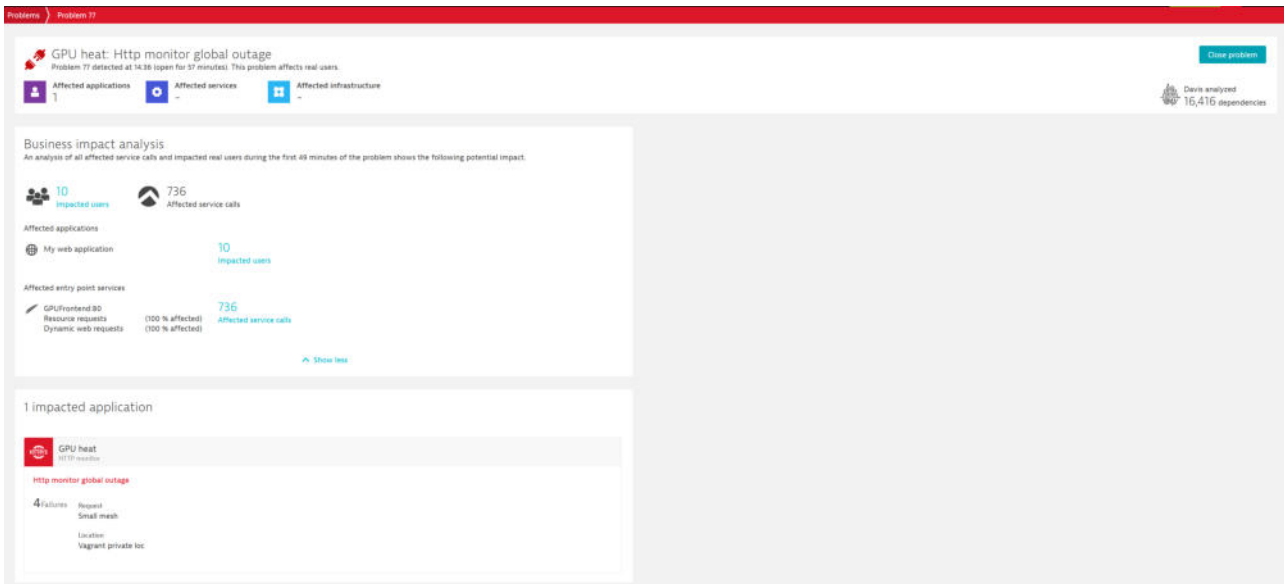

Fig. 5.4. *Detected availability problem (left) and missing RCA (right)*

5.3.2. Test case 2. In this scenario, *Backend Application* failures due to inability to allocate sufficient memory are simulated, the memory is occupied by CUDA Load Generator. Contrary to the previous test case, Dynatrace fails to identify the root cause (Fig. 5.4), even though an alert for high GPU memory occupancy is raised for the host in question. By viewing the host screen in context of the mentioned problem, one can see an

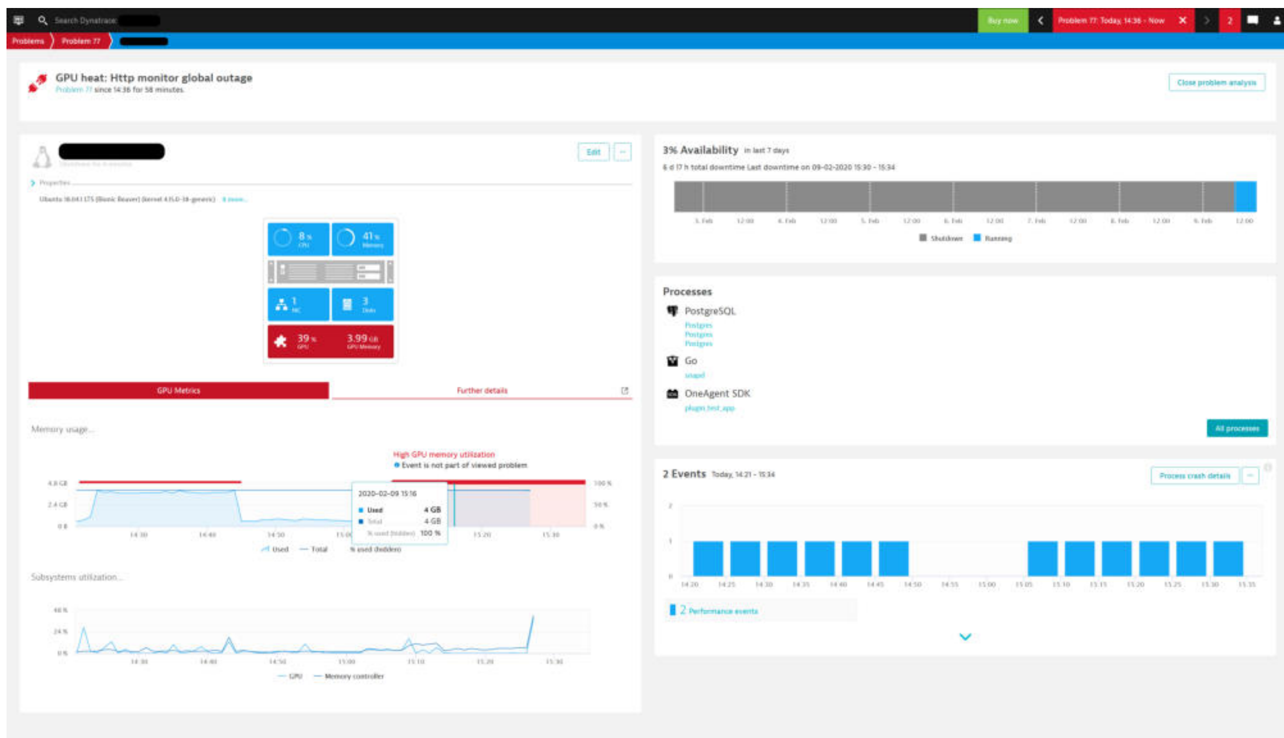

FIG. 5.5. *Host screen in context of the availability problem*

explicit tooltip indicating lack of correlation (Fig. 5.5). Even though there is no clear RCA, the data provided by extension is still a valuable source of information and shows that GPU ran out of memory resources. On this basis one can easily infer that this had most likely caused request processing failures of the web service in question.

6. Summary and future work. This paper presented an extension for Dynatrace OneAgent that enabled it to monitor GPU performance by gathering NVML metrics. It was shown how to leverage existing tools and software libraries to monitor the performance of a GPU and alert about performance problems. Extension usefulness was proven using two production-like scenarios, where it helped to quickly identify the root cause for performance degradation and availability problem in a customer-facing web application. During solution validation few shortcomings of the Dynatrace platform were identified, namely lack of out-of-the-box support for including extension metrics in RCA and deficiencies in extension SDK. Nevertheless, the end result was satisfying.

The author believes that the extension will be useful for existing users of Dynatrace, who manage infrastructure that includes GPU-enabled nodes, and that the implementation details, plus the freely available source code would help other developers to extend rest of the APM tools with similar support for GPU monitoring, which is very limited at the time of writing.

In future, several improvements are planned. Scope of metrics gathered should be broadened to include: CUDA properties, ECC errors count, GPU core and memory clock speeds, power cap, and throttling events. By leveraging SDK-like approach, where users would need to modify their application's code, the author plans to also collect kernel execution and memory transfer durations to identify kernels that violate an automatically predetermined baseline for a given set of input parameters. Improvements in available alerting profiles should also be considered to identify compound problem patterns, e.g. alert on unexpected underutilization of a GPU (device being idle, while it is expected to process data). The extension should also be validated on a wider set of GPU architectures, especially Volta or higher to verify if per-process memory occupancy can be reported

on Windows. Lastly, there are plans to develop another extension, that leverages CUPTI for collection of code-level metrics.

REFERENCES

[1] *Ganglia monitoring system*. https://developer.nvidia.com/ganglia-monitoring-system.
[2] *Top 500 list*. https://top500.org/lists/top500/2020/06/.
[3] T. M. Ahmed, C. Bezemer, T. Chen, A. E. Hassan, and W. Shang, *Studying the effectiveness of application performance management (apm) tools for detecting performance regressions for web applications: An experience report*, 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), (2016), pp. 1–12.
[4] Bright Computing, *Bright cluster manager*. https://www.brightcomputing.com/documentation.
[5] J. Choi, *gpustat*. https://github.com/wookayin/gpustat.
[6] Datadog, *Documentation*. https://docs.datadoghq.com.
[7] S. Dutta, *An overview on the evolution and adoption of deep learning applications used in the industry*, Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 8 (2017).
[8] Dynatrace, *Oneagent sdk documentation*, October 2019. https://github.com/Dynatrace/OneAgent-SDK-for-C.
[9] ———, *Extension sdk documentation*, September 2020. https://www.dynatrace.com/support/help/shortlink/oneagent-extensions-tutorial.
[10] ———, *Oneagent documentation*, September 2020. https://www.dynatrace.com/support/help/shortlink/oneagent-hub.
[11] ———, *The software intelligence platform*, September 2020. https://www.dynatrace.com/platform.
[12] ———, *Synthetic monitoring documentation*, September 2020. https://www.dynatrace.com/support/help/shortlink/synthetic-hub.
[13] A. Frantzis and J. Barker, *glmark2 - an opengl 2.0 and es 2.0 benchmark*, June 2015. https://github.com/glmark2/glmark2.
[14] K. Fürlinger, N. Wright, and D. Skinner, *Comprehensive performance monitoring for gpu cluster systems*, IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, (2011), pp. 1377 – 1386.
[15] G2, *Best application performance monitoring (apm) software*, September 2020. https://www.g2.com/categories/application-performance-monitoring-apm.
[16] T. Gajger and P. Czarnul, *Modelling and simulation of gpu processing in the merpsys environment*, Scalable Computing: Practice and Experience, 19 (2018), pp. 401–422.
[17] Gartner, *Application performance monitoring market*, September 2020. https://www.gartner.com/reviews/market/application-performance-monitoring.
[18] Google, *Compute engine - monitoring gpu performance*. https://cloud.google.com/compute/docs/gpus/monitor-gpus.
[19] C. Heger, A. van Hoorn, M. Mann, and D. Okanović, *Application performance management: State of the art and challenges for the future*, Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, (2017), pp. 429–432.
[20] InfluxData, *Telegraf*. https://www.influxdata.com/time-series-platform/telegraf/.
[21] IT Central Station, *Best application performance monitoring & management (apm) tools*, September 2020. https://www.itcentralstation.com/categories/application-performance-management-apm.
[22] M. Khairy, A. Wassal, and M. Zahran, *A survey of architectural approaches for improving gpgpu performance, programmability and heterogeneity*, Journal of Parallel and Distributed Computing, 127 (2019).
[23] T. Nagai, *datadog_nvml*. https://github.com/ngi644/datadog_nvml.
[24] New Relic, *Google kubernetes engine monitoring integration*. https://docs.newrelic.com/docs/integrations/google-cloud-platform-integrations/gcp-integrations-list/google-kubernetes-engine-monitoring-integration.
[25] NVIDIA Corporation, *Dcgm*. https://developer.nvidia.com/dcgm.
[26] ———, *Supported Compute Debugger Configurations*. https://developer.nvidia.com/nsight-visual-studio-edition-supported-gpus-full-list#SupportedComputeConfigs.
[27] ———, *nvidia-smi - nvidia system management interface*, July 2016. http://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.
[28] ———, *Python bindings for the nvidia management library*, June 2017. https://pypi.org/project/nvidia-ml-py3.
[29] ———, *Cupti user's guide*, August 2020. https://docs.nvidia.com/cuda/pdf/CUPTI_Library.pdf.
[30] ———, *Gpu applications catalog*, September 2020. https://www.nvidia.com/en-us/gpu-accelerated-applications/.
[31] ———, *Nvml api reference guide*, June 2020. https://docs.nvidia.com/pdf/NVML_API_Reference_Guide.pdf.
[32] M. Schmitt, *nvtop*. https://github.com/Syllo/nvtop.

Edited by: Dana Petcu
Received: Sep 28, 2020
Accepted: Dec 7, 2020