Profiling & Monitoring Deep Learning Training Tasks

Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, Pınar Tözün
IT University of Copenhagen, Copenhagen, Denmark
ehyo,titr,pito@itu.dk

Abstract

The embarrassingly parallel nature of deep learning training tasks makes CPU-GPU co-processors the primary commodity hardware for them. The computing and memory requirements of these tasks, however, do not always align well with the available GPU resources. It is, therefore, important to monitor and profile the behavior of training tasks on co-processors to understand better the requirements of different use cases. In this paper, our goal is to shed more light on the variety of tools for profiling and monitoring deep learning training tasks on server-grade NVIDIA GPUs. In addition to surveying the main characteristics of the tools, we analyze the functional limitations and overheads of each tool by using a both light and heavy training scenario. Our results show that monitoring tools like `nvidia-smi` and `dcgm` can be integrated with resource managers for online decision making thanks to their low overheads. On the other hand, one has to be careful about the set of metrics to correctly reason about the GPU utilization. When it comes to profiling, each tool has its time to shine; a framework-based or system-wide GPU profiler can first detect the frequent kernels or bottlenecks, and then, a lower-level GPU profiler can focus on particular kernels at the micro-architectural-level.

CCS Concepts: • **Computing methodologies** → *Artificial intelligence*; *Machine learning*; • **Hardware**; • **Computer systems organization** → **Parallel architectures**;

Keywords: deep learning, co-processors, monitoring, profiling, optimization

ACM Reference Format:
Ehsan Yousefzadeh-Asl-Miandoab, Ties Robroek, Pınar Tözün. 2023. Profiling & Monitoring Deep Learning Training Tasks. In *3rd Workshop on Machine Learning and Systems (EuroMLSys '23), May 8, 2023, Rome, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3578356.3592589

ACM ISBN 979-8-4007-0084-2/23/05…$15.00
https://doi.org/10.1145/3578356.3592589

1 Introduction

Deep learning training requires high computing and memory resources and is a highly parallel process. This has naturally lead to accelerating the training processes with hardware architectures such as GPUs that can exploit these traits. On the other hand, matching the computing and memory requirements of deep learning training to the capabilities of modern GPUs is not straightforward for all deep learning applications and GPU types. This mismatch results in slowdowns and resource underutilization [14, 19, 23, 36]. To find solutions to these challenges, it is essential to characterize the interaction between deep learning systems and their underlying hardware. This is obtainable by profiling systems and monitoring hardware utilization.

Profiling provides the developers with insights in how the application behaves in terms of computing and memory patterns and requirements. Afterwards, the data and trace plots can assist in finding and addressing the bottlenecks. Monitoring tools, in contrast, reveal how specific hardware resources react to the execution of applications. One can find out whether the current configuration of the model training saturates the hardware resources. The workload can be scaled up or more applications may be run simultaneously to optimize for both high utilization and training performance.

Using profiling and monitoring tools effectively is an art and can be time-consuming for beginners. Furthermore, in the field of deep learning, one has to understand the tools for not only CPUs but also accelerators like GPUs. While there are many works utilizing tools for CPUs (e.g., top, perf, Intel VTune) for workload characterization [13, 15, 17, 21, 22, 32, 34, 35], tools for accelerators are less mature and rapidly evolving, and relatively unexplored. To address this challenge, this paper reviews the most relevant profiling and monitoring tools for deep learning workloads. We investigate the strengths and limitations of the profiling tools offered by NVIDIA, Nsight Systems and Compute, in addition to the monitoring tools `nvidia-smi` and `dcgm`. We do this by (1) surveying the functionality offered by these tools, (2) studying the metrics reported and showing the shortcomings of widely used high-level utilization metrics, and (3) measuring profiling and monitoring tools' overheads while running both light and heavy deep learning training scenarios.

Our investigation demonstrates the following:

- The negligible overhead of the monitoring tools make them ideal candidates to be integrated into task schedulers and resource managers for online decision-making.

- On the other hand, the GPU utilization and GRACT metrics offered by `nvidia-smi` and `dcgm`, respectively, are too high-level and unrepresentative for actual GPU utilization. More concrete metrics such as SMACT and SMOCC from `dcgm` may help to overcome this issue.
- The profiling tools are effective for targeted code optimizations, but their overheads make them unsuitable for online decision making. The profiling mode of Nsight Compute in particular heavily disrupts a training run.
- Each profiling tool has their time to shine. Profiling tools integrated into deep learning frameworks and Nsight Systems offer a way to detect bottlenecks with application-specific and system-wide views, respectively. However, to further optimize individual kernels, a tool like Nsight Compute offer deeper insights at the micro-architectural level of a GPU. Thus, one can create a pipeline of profiling stages using a mix of tools.

2 Tools

This section surveys the most relevant profiling and monitoring tools for deep learning training on NVIDIA GPUs.

2.1 Profiling Tools

There is a range of tools available to profile deep learning workloads. Tools integrated with the deep learning frameworks, such as the TensorFlow and PyTorch profilers [8, 10], are immediately available to those using their respective frameworks. Alternatively, NVIDIA provides the profiling tools Nsight Systems and Nsight Compute. This section goes over the PyTorch profiler, as a representative framework tool, and the NVIDIA profiling tools.

The PyTorch Profiler [8] is a trace-based profiling tool that can automatically collect a range of performance metrics during both deep learning training and inference. As it is integrated into the deep learning framework itself, running the profiler is just a matter of adding a few lines of Python code. It requires less setup than other monitoring or profiling tools due to being specific to PyTorch and deep learning. Being integrated into the code, the profiler allows for extensive control of which iterations are profiled. This prevents the profiling data from growing out of hand. It is in fact recommended that the users profile one or more iterations in an epoch rather than whole epochs, since the behavior of the iterations over each batch tends to be repetitive. In Section 3, we highlight this while discussing the overheads of the PyTorch profiler.

NVIDIA Nsight Systems [4], `nsys`, is a trace-based profiler similar to the PyTorch profiler. It constructs a timeline of CPU and GPU events. This notably includes different compute and memory access streams on the GPU, yielding valuable information such as data movement bottlenecks and most frequently used kernels. `nsys` is framework-independent and can effectively profile a variety of software. Furthermore, it offers a system-wide view, including more insights to interactions with the operating system and network compared to the more application-focused view given by the framework profilers. `nsys`, thus, does not annotate the deep learning traces out of the box. NVTX, NVIDIA Tools Extension [7], provides an API to enable annotating the training code itself. Multiple deep learning libraries, including PyTorch [28], support NVTX annotations in their code.

`nsys` runs as a separate process while profiling an application. Applications can be profiled both online/interactive and offline. The profiling is done either via a GUI or a command line with the level of detail specified by the user. For example, a user can launch `nsys` to track the GPU memory usage by kernels, enable the collection of backtraces, and collect metrics from network interface cards.

Bottlenecks can be detected by viewing the timeline of computing and memory operations. For example, a timeline detailing that 90% of the time is spent on compute indicates that the workload is compute-intensive and that compute-side optimizations might improve the application. Conversely, when there are a lot of data access stalls, the workload is memory-intensive and improved data orchestration may greatly improve runtime.

It should be noted that `nsys` does not work when multi-instance GPU (MIG) mode [27, 29], which divides a GPU into smaller instances, is enabled on any GPU on the server. While this is a current functional limitation as MIG technology is relatively new and has been maturing, it may be fixed over time. In addition, carelessly specifying more and more profiling options to get more details can result in longer post-processing times after the profiling is over and bigger trace files that are harder to render in the tool's GUI.

NVIDIA Nsight Compute [3], `ncu`, allows for in-depth GPU analysis. It disrupts the regular run of a program and reruns the kernels of a program multiple times to trace the micro-architectural behavior.

Similar to `nsys`, `ncu` has a GUI-based and command line interface, where the users can specify the amount information to trace. As the nature of the profiling is disruptive, it is often run as an online interactive profiler and debugger, though offline mode is also supported. Compared to the PyTorch Profiler and `nsys`, the main strength of `ncu` is the degree of detail and granularity it provides when profiling. It illustrates the data movement behavior across the different levels of the GPU memory hierarchy and helps to identify data stalls in kernels. It also maps the metrics to the individual lines of code that contribute to them by connecting assembly (SASS) code with parallel thread execution instruction set architecture (PTX or NVPTX), an architecture independent intermediate representation for CUDA, and with high-level code (e.g., CUDA, C/C++, Fortran, OpenACC, Python). Additionally, it can export CUDA execution graphs and allow the profiling of individual nodes in these graphs.

While ncu is good to investigate things at a microscopic level, it does impact application behavior. Its profiling depends on the principle of rerunning kernels multiple times either one kernel at a time (*kernel mode*) or via iterating over the application multiple times (*application mode*) as specified by the user. In each iteration, additional data for the target kernel(s) is collected. Application replay requires the program execution to be deterministic.

While ncu provides extremely detailed information at the kernel level, it is often difficult to map the information to the application level. Furthermore, as a result of the repetitive kernel runs, the profiling overhead on the application is very high. Therefore, ncu should mainly be used to optimize individual kernels, not application-level scheduling behavior. Since it supplies the users with GPU architecture and micro-architecture-related information, it is extremely useful for computer architects and low-level library developers.

2.2 Monitoring Tools

There are a variety of monitoring tools for servers to observe their utilization behavior such as how many CPU cores are in use, how many GPU streaming multiprocessors are active, what the CPU/GPU memory consumption is, etc. Such observations can aide cluster administration, hardware resource management, and workload scheduling decisions, even in real-time thanks to the low-overhead of the monitoring tools (as quantified in Section 3.2.2). On the other hand, in contrast to the profiling tools, these tools cannot be used for coming up with optimization ideas for a specific application's internals or kernels. In this section, we cover the two monitoring tools offered by NVIDIA: `nvidia-smi` and `dcgm`.

NVIDIA System Management Interface [5], `nvidia-smi`, provides monitoring and management capabilities for NVIDIA GPUs. Users can interact with it via command line to (1) configure a GPU's performance parameters like changing the frequency, setting power cap, etc., (2) set the preferred multi-instance GPU (MIG) partitions, and (3) track a range of performance metrics such as GPU utilization, size of GPU memory usage, performance state and temperature of a GPU, etc. One can view the metrics tracked via standard output or write them to a CSV or XML file. These metrics can be tracked system-wide, for a GPU, and for an application.

Underneath, `nvidia-smi` uses the NVIDIA Management Library (NVML) [2], which provides an API for monitoring and managing various states of NVIDIA GPUs. NVML provides direct access to the queries and commands that enables the monitoring done by `nvidia-smi`. If users want to customize the monitoring, they can write a custom program using NVML instead of using what is exposed by `nvidia-smi`.

While `nvidia-smi` helps with basic system monitoring, it is still limited in terms of the metrics it provides. For example, it doesn't track the interactions between the CPU and the GPU. Furthermore, on a MIG-enabled GPU, it tracks metrics from the whole GPU and not from individual MIG instances.

Table 1. Specifications of an A100 GPU - 40GB

Property	**Value**
GPU Architecture	NVIDIA Ampere
Compute Capability	8.0
#SMs	108
FP32 per SM	64
Tensor Cores per SM	4
Share Memory and L1 cache	192KB combined, Shared Memory is configurable up to 164KB
Max 32-bit Registers per SM	64KB
L2 cache	40MB
Memory	40 GB of high-speed HBM2 memory
Max Threads per Warp	32
Max Thread Blocks per SM	32
Max Warps per SM	64
Max Thread Block Size	1024
Max Registers per Thread	255

NVIDIA Data Center GPU Manager [1], `dcgm`, provides more detailed information about hardware utilization on CPU-GPU co-processors compared to `nvidia-smi`. `dcgm` can ease the management and configuration of GPUs in a cluster by providing features such as GPU grouping. Furthermore, it can track not just high-level GPU utilization, but also occupation and activity of streaming multiprocessors and utilization of tensor cores. It can give more detailed insights on energy consumption of the GPU and the data movement across CPU and GPU and different GPUs by reporting how much the PCIe / NVLink bandwidth is used. Finally, it can also monitor the utilization of the individual MIG instances. On the other hand, since both ncu and `dcgm` use the same hardware counters underneath, they cannot be used simultaneously.

3 Experiments

After the qualitative overview of the tools of interest in Section 2, this section quantitatively analyzes them. We aim at answering the following questions with our experiments:

- What is the granularity of information reported by the different GPU utilization metrics?
- How intrusive are these tools on the execution of a deep learning training process?
- How much hardware resources do these tools need?
- How does the relative impact of these tools change based on the size and complexity of the deep learning training?

3.1 Setup

All the experiments are run on a DGX Station A100, which is composed of an AMD EPYC 7742 CPU with 512GB of main

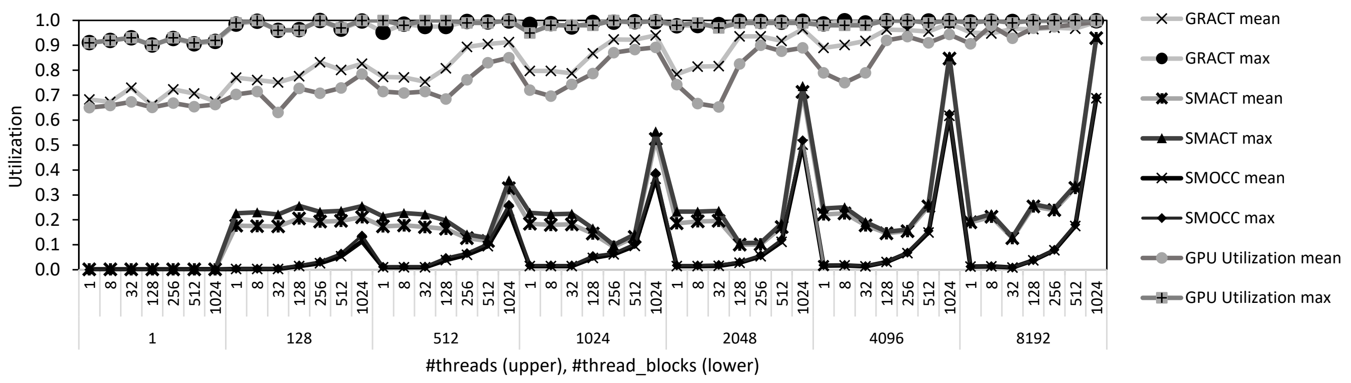


Figure 1. Different GPU utilization metrics as the load on the GPU varies.

memory and four A100 GPUs with 40GB of memory each. Table 1 details the specifications of the A100 GPU.

The system runs *DGX OS*, a variant of *Ubuntu 20.04.4 LTS*, and the installed CUDA version is 11.6.1.

Except for Section 3.2.1, the experiments are based on two types of training runs: (1) a *light* use-case with a simple CNN-model [25] trained on the MNIST [16] dataset, (2) a more *heavy* use-case in both computation and memory consumption with ResNet50 [18, 37] trained on the ImageNet dataset [30]. All models use the PyTorch framework, version 1.13.1, and are trained for 5 epochs. For the experiments in Section 3.2.1, where the different GPU utilization metrics are investigated, we create our custom micro-benchmark, which is described in the corresponding section.

We scoped down the experiments to a single light and heavy training scenario rather than experimenting with a wider variety of model training. When it comes to identifying the overhead caused by the tools, there can be two types of overhead: (1) fixed one such as fixed startup or shutdown overhead and fixed background resource usage, and (2) the one that vary based on the program complexity such as increased resource consumption due to more data being collected. The former would be more pronounced for the *light* training scenario, whereas it would be amortized or insignificant for the *heavy* scenario. The latter would be more significant for the *heavy* scenario. We argue that the overall conclusions for the respective behavior for these two types of overhead do not change across different light and heavy scenarios. This aligns with our experience using different training use cases with these profiling and monitoring tools.

For the profilers, we run our experiments offline with the default settings. The PyTorch Profiler, `pytorch`, records both CUDA and CPU activity by default, while all of the extra options, such as flop estimation, are disabled. Running this with the 5-epochs of ResNet50 leads to prohibitive tracing information. Therefore, we only report `pytorch` results for the light training scenario. The traces collected by `nsys`, version 2022.1.3, are for CPU, CUDA, NVTX, OSRT, and OpenGL calls, as well as high-level resource utilization, but without CUDA backtracing. Finally, as `ncu`, version 2022.2.1, is by design a disruptive profiler (Section 2.1), we decided that it is not insightful to report its overheads.

The raw experimental data can be found in our repository.[1]

3.2 Results

Among the questions listed above, Section 3.2.1 answers the first one, and Section 3.2.2 answers the remaining three.

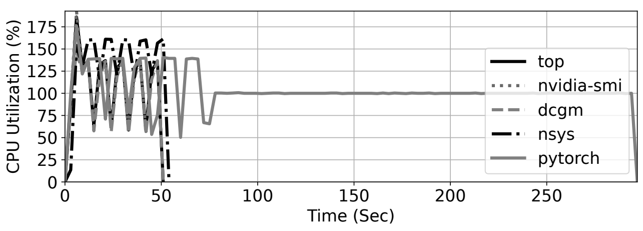
3.2.1 GPU utilization. The popular metrics of interest while monitoring GPUs tend to be compute utilization, memory consumption, data movement, and energy consumption. Especially the GPU utilization can potentially be confusing due to the different ways for measuring this activity.

The GPU monitoring tools `nvidia-smi` and `dcgm` both have a *GPU utilization* metric, which is roughly defined as the % of time one or more kernels were executing on the GPU over the past sampling period. In addition, `dcgm` has multiple metrics to track the utilization of streaming multiprocessors (SMs). Most notable ones among these metrics are `GRACT`, `SMACT`, and `SMOCC`, which we investigate in this section.

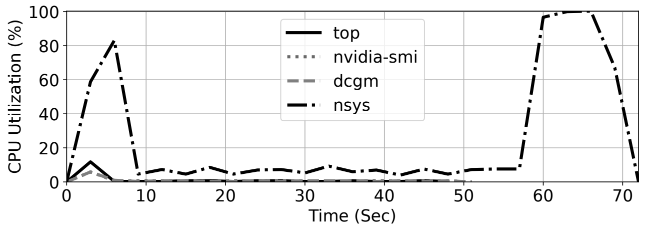
`GRACT`, *graphics engine activity*, is the fraction of time during which any portion of the graphics (e.g., ray tracing units) or compute engines were active. While `GRACT` tends to closely follow *GPU utilization* in values, in practice it is measured differently (sampling, hardware counters, etc.). Therefore, its over time values aren't exactly the same as what *GPU utilization* reports. `SMACT`, *SM activity*, refers to the fraction of active time on an SM, averaged over all SMs. Finally, `SMOCC`, *SM occupancy*, is the degree of parallelism within an SM (calculated by the unit of a warp, which typically occupies 32 threads in a thread block) relative to the maximum degree of parallelism supported by the SM.

For our investigation, we devise a micro-benchmark in which we vary the number of thread blocks and threads within a thread block in a kernel.[2] Each thread fetches a data

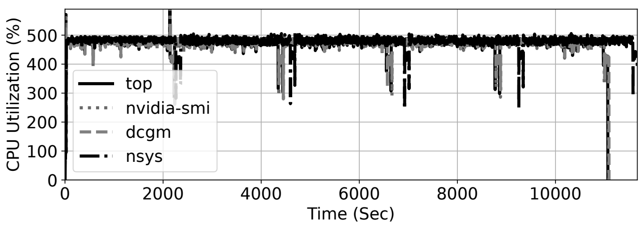
[1]https://github.com/Resource-Aware-Data-systems-RAD/PMDLT
[2]https://github.com/Resource-Aware-Data-systems-RAD/PMDLT/blob/main/benchmark/square.cu



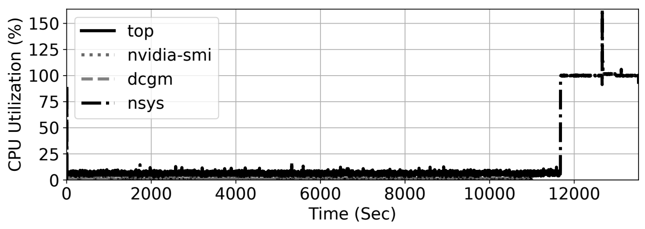
(a) Training processes while running light training.



(b) Tool processes while running light training.



(c) Training processes while running heavy training.



(d) Tool processes while running heavy training.

Figure 2. CPU Utilization.

item and calculates its square. Figure 1 shows the results. As the figure highlights, even when there is only one thread within a single thread block, GRACT can show a utilization of 90%. This is extremely misleading when one is interested in whether the SMs are in use. In contrast, SMACT and SMOCC reveal substantially more information on SM utilization.

The GPU used in our experiments (Table 1) have support for up to 3456 thread blocks in total, where each thread block has support for up to 1024 threads in total. We see that SMACT and SMOCC reach higher values as we get closer to the limits of parallelism offered by the GPU, reflecting the actual load on the SMs. Due to possible overheads in orchestrating the threads, though, neither SMACT and SMOCC reach 100%.

In conclusion, one must be careful about the GPU utilization metrics to monitor depending on the goal of monitoring. For example, for a task scheduler that aims to decide which tasks to collocate on a GPU, plainly looking at GPU utilization or GRACT *will miss the opportunities for utilizing the GPU better.*

3.2.2 Tool Overheads. To quantify the overheads of the tools described in Section 2, we measure the epoch execution time, the size of the information produced by each tool, and utilization of CPU and GPU resources with and without using a particular tool. Since top [11] is used to collect CPU utilization information, we also include it in the results.

Execution time. To reason about the runtime impact of each tool, Table 2 reports the average epoch time. The execution time for each epoch is taken from PyTorch. As Table 2 shows, while the monitoring tools have negligible impact on execution time, the profiling tools lead to a noticeable overhead. Figure 2 reveals that there is bigger runtime overhead for the profiling tools after the training is over, which is for post-processing the gathered information, omitted from

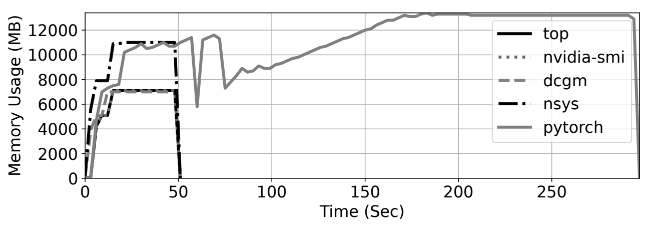
Table 2. Average epoch time w/o profiling and monitoring, and size of the information collected by the tools.

Tools	**simple CNN**		**ResNet50**	
	Time	Space	Time	Space
no tool	9.61sec	NA	37.06min	NA
top	9.66sec	~20KB	37.11min	~2MB
nvidia-smi	9.61sec	~20KB	37.04min	~2MB
dcgm	9.68sec	~85KB	37.19min	~8MB
nsys	9.88sec	~40MB	39.13min	~5GB
pytorch	13.65sec	~1.4GB	NA	

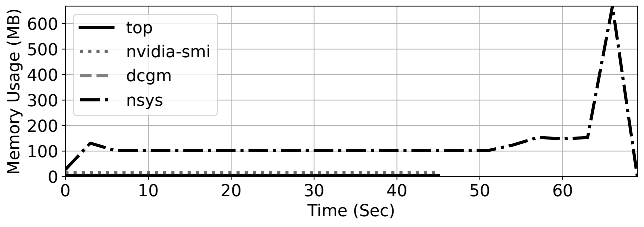
Table 2. The overhead of the pytorch profiler is larger, since it collects more data by default compared to nsys.

Overall, while the monitoring tools can be integrated into online decision making, the profiling tools should be used for deliberate targeted investigations and optimizations.

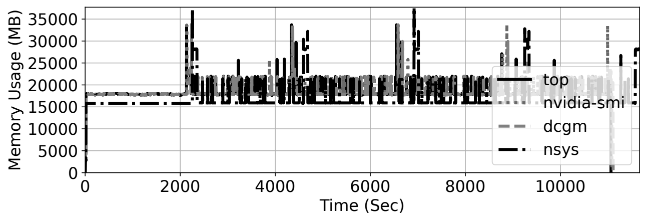
Size of data files. Table 2 also reports the size of the information collected by each tool. For the monitoring tools, we manage how this information is stored, we simply write the information to an output file. As expected, the information collected by these tools have negligible overhead, and since dcgm offers more metrics to collect, it accumulates more data. The size of the information is larger for the profiling tools, since they collect more information. *pytorch* logs the actions that are part of the framework in great detail by default. In particular, the detailed stack traces of PyTorch functions and libraries contribute greatly to the scale of information. Additionally, the files are saved in the Chrome JSON format, which is not optimized for space at all. In comparison, *nsys* defaults to logging more generic system parameters as it is an application-independent solution. In addition, the files are saved in a compressed binary format. However, increasing



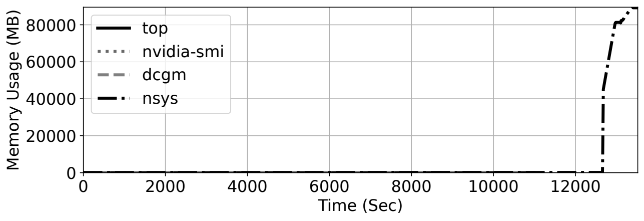
(a) Training processes while running light training.



(b) Tool processes while running light training.



(c) Training processes while running heavy training.



(d) Tool processes while running heavy training.

Figure 3. CPU memory usage.

the information collection in nsys would naturally increase the complexity and size of the trace files as well.

We highlight that while building a platform for systematically benchmarking the interaction between the deep learning applications and hardware, keeping all the monitoring and profiling information from various experiments may become a scalability challenge that has to be addressed.

CPU utilization. Figure 2 shows the CPU utilization for both the training process itself while running a variety of profiling and monitoring tools and the processes created by the tools. We use top to report CPU utilization for each tool. Therefore, the line marked as top represents the baseline, and the rest of the tools run in parallel with top.

In Figure 2a, we see that the monitoring tools have no visible impact on the CPU usage of the training process, since they don't increase the CPU utilization beyond the baseline. That is why the lines for top, nvidia-smi, dcgm, and pytorch overlap completely till around 50 seconds. The 50 seconds mark the training time for 5 epochs (Table 2), where both the training and monitoring stops. On the other hand, nsys increases the CPU utilization slightly, ~18%, while pytorch keeps it similar to the baseline training. However, pytorch has its post-processing phase performed by the main training process as well, which is why Figure 2a depicts a lengthy 1-core utilization after the 5-epoch training is over for pytorch. As we can see in Figure 2b, nsys launches a separate process for this purpose, which goes through the phases of (1) initialization (initial jump to ~ 80%), (2) waiting for the training to be over (low utilization), (3) post-processing (brief ~ 100% utilization). The post-processing time is shorter for nsys compared to pytorch. This is likely due to the larger trace gathering performed by pytorch with its default settings compared to nsys. (see Table 2). Finally, the monitoring tools also spawn their own helper processes with negligible CPU utilization as we see in Figure 2b.

Figure 2c and Figure 2d show that in a heavier training scenario the impact of all tools on CPU utilization is insignificant, while the impact on the total execution time and post-processing is still visible with nsys finishing later.

Overall, when it comes to the profiling tools, one has to be mindful about the hardware resource consumption and the execution time of the post-processing phase of the profilers.

CPU memory usage. Figure 3 shows the CPU memory usage for both the training process itself while running a variety of profiling and monitoring tools and the processes created by the tools. We use top once again to report the CPU memory consumption for each tool. Thus, the top-line represents the baseline similar to Figure 2.

In Figure 3a, we see that while the monitoring tools have no visible impact on the CPU memory usage of the training process, the profiling tools have an impact. Both nsys and pytorch increase the CPU memory usage (~55%), and pytorch's post-processing increases the total memory consumption further. In Figure 3b, the nsys helper process has the same three-stage behavior as found in Figure 2b. Finally, for the heavy training scenario, Figure 3c reveals that the impact on resource usage is negligible for all of the tools during the training epochs. However, the impact of post-processing of the profiling tools is still considerable.

Overall, Figure 3 exhibits similar trends to Figure 2.

GPU resources. Figure 4 shows the impact of the tools on the GPU resource usage. None of these tools create a separate helper process on the GPU. Therefore, the results are only for the training process. We retrieve these metrics from dcgm, which makes the bars for dcgm our baseline. As the GPU compute utilization metrics we report SMACT and

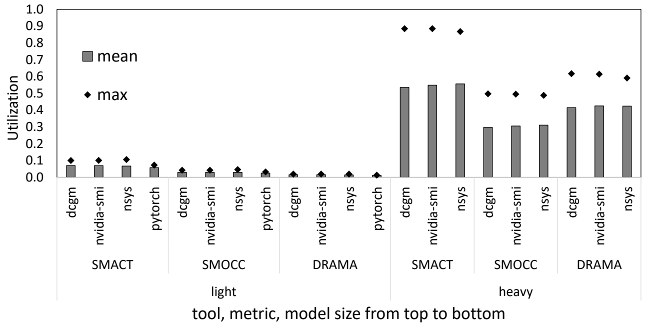


Figure 4. GPU utilization.

SMOCC based on the results of Section 3.2.1. For the GPU memory utilization we use DRAMA, showing the frequency of memory accesses.

In general, when it comes to the impact on the GPU resource usage, all of the tools have negligible impact.

4 Related Work

While we study the main monitoring tools of NVIDIA GPUs, AMD offers similar tools such as ROCm-smi [9]. In addition, there are tools built on top of existing NVIDIA libraries. For example, nvtop [6] is a wrapper around NVML that provides visualization for NVML metrics, and Moneo [24] is a monitoring system that specifically targets AI applications.

Orthogonal to Nsight Systems and Nsight Compute, there has also been efforts to build profiling tools using the NVIDIA CUDA Profiling Tools Interface (CUPTI API [26]) [12, 20, 24, 31, 38, 39]. Furthermore, there are tools that have a stronger focus on profiling the data movement, such as [20], which is also built on top of CUPTI in addition to OSU INAM [33]. Finally, nvprof was also a tool for profiling on NVIDIA GPUs, but it has been deprecated by the release of Nsight Systems and Nsight Compute.

In this paper, we scoped our study to the most relevant tools provided by NVIDIA, while using the PyTorch profiler as a point of comparison, but a similar investigation can be done for the aforementioned tools using our methodology.

5 Conclusion

Deep learning models have become essential in many application domains but are expensive to train. It is thus important to understand the behavior of the training software and the underlying hardware. In this paper, we have analyzed the impact of monitoring and profiling tools on deep learning training. We have found that monitoring tools have negligible overhead and can be used for online decision making. In contrast, profiling tools offer more detailed information but incur time, space, and hardware resource consumption overheads. Additionally, one should be careful with their choice of metrics to monitor, as some paint a clearer picture than others, especially in the case of GPU utilization.

Profiling and monitoring tools have a fast and an ever-evolving nature. For instance, in the recent past, dcgm didn't report metrics for the 4g.20GB MIG instance, but now it does. Similarly, the NVML library, which underlies nvidia-smi, recently added finer-grained dcgm metrics like SMACT and SMOCC. However, these additions are supported only on the newer NVIDIA GPUs such as the ones based on the Hopper architecture; one generation later than the Ampere architecture used in this study. This addition means that one can simply collect metrics using nvidia-smi if the point of interest is overall GPU utilization on the latest and emerging NVIDIA GPUs, removing the dependency on multiple tools. Therefore, one should pay attention to using the up-to-date version of the tools on a given processor to determine the most effective subset of tools for a particular study.

Acknowledgements

This work is funded by the Independent Research Fund Denmark's (Danmarks Frie Forskningsfond; DFF) Sapere Aude program under grant agreement number 0171-00061B. We also thank DASYA lab members at IT University of Copenhagen for their support, and the reviewers of EuroMLSys for their constructive feedback.

References

[1] [n. d.]. NVIDIA Data Center GPU Manager. https://github.com/NVIDIA/DCGM.
[2] [n. d.]. NVIDIA Management Library (NVML). https://developer.nvidia.com/nvidia-management-library-nvml.
[3] [n. d.]. NVIDIA Nsight Compute. https://developer.nvidia.com/nsight-compute.
[4] [n. d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.
[5] [n. d.]. NVIDIA System Management Interface. https://developer.download.nvidia.com/compute/DCGM/docs/nvidia-smi-367.38.pdf.
[6] [n. d.]. NVTOP: Neat Videocard TOP. https://github.com/Syllo/nvtop.
[7] [n. d.]. NVTX (NVIDIA Tools Extension Library). https://nvidia.github.io/NVTX/.
[8] [n. d.]. PyTorch Profiler. https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html.
[9] [n. d.]. ROCm Documentation. https://sep5.readthedocs.io/en/latest/ROCm_System_Managment/ROCm-System-Managment.html.
[10] [n. d.]. TensorBoard: TensorFlow's visualization toolkit. https://www.tensorflow.org/tensorboard.
[11] [n. d.]. top(1) — Linux manual page. https://man7.org/linux/man-pages/man1/top.1.html.
[12] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.
[13] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *VLDB*. 266–277.
[14] Sebastian Baunsgaard, Sebastian Benjamin Wrede, and Pınar Tözün. 2020. Training for Speech Recognition on Coprocessors. In *ADMS*.
[15] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *ASPLOS*. 127–144.

[16] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[17] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*. 37–48.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*. 770–778.

[19] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. 2019. Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads. In *USENIX ATC*. 947–960.

[20] Yuting Jiang, Yifan Xiong, Lei Qu, Cheng Luo Luo, Chen Tian, Peng Cheng, and Yongqiang Xiong. 2022. Moneo: Monitoring Fine-Grained Metrics Nonintrusively in AI Infrastructure. *ACM SIGOPS Oper. Syst. Rev.* 56, 1 (2022), 18–25.

[21] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-Scale Computer. In *ISCA*. 158–169.

[22] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker. 1998. Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads. In *ISCA*. 15–26.

[23] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. 2019. Crossbow: Scaling Deep Learning with Small Batch Sizes on Multi-GPU Servers. *PVLDB* 12, 11 (2019), 1399–1412.

[24] Pouya Kousha, Bharath Ramesh, Kaushik Kandadi Suresh, Ching-Hsiang Chu, Arpan Jain, Nick Sarkauskas, Hari Subramoni, and Dhabaleswar K. Panda. 2019. Designing a Profiling and Visualization Tool for Scalable and In-depth Analysis of High-Performance GPU Clusters. In *IEEE HiPC*. 93–102.

[25] Bryan McCann and contributors. 2022. PyTorch Example. https://github.com/pytorch/examples/tree/main/mnist.

[26] NVIDIA. [n. d.]. CUDA Profiling Tools Interface (CUPTI). https://docs.nvidia.com/cupti/.

[27] NVIDIA. 2020. *NVIDIA Multi-Instance GPU and NVIDIA Virtual Compute Server - GPU Partitioning - Technical Brief.* Technical Report. NVIDIA. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Technical-Brief-Multi-Instance-GPU-NVIDIA-Virtual-Compute-Server.pdf.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS*. 8026–8037.

[29] Ties Robroek, Ehsan Yousefzadeh-Asl-Miandoab, and Pınar Tözün. 2022. An Analysis of Collocation on GPUs for Deep Learning Training. *CoRR* (2022).

[30] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *IJCV* 115, 3 (2015), 211–252.

[31] Sameer S Shende and Allen D Malony. 2006. The TAU parallel performance system. *IJHPCA* 20, 2 (2006), 287–311.

[32] Utku Sirin, Ahmad Yasin, and Anastasia Ailamaki. 2017. A methodology for OLTP micro-architectural analysis. In *DaMoN @ ACM SIGMOD*. 1:1–1:10.

[33] Hari Subramoni, Albert Mathews Augustine, Mark Arnold, Jonathan Perkins, Xiaoyi Lu, Khaled Hamidouche, and Dhabaleswar K Panda. 2016. INAM2: InfiniBand network analysis and monitoring with MPI. In *High Performance Computing*. 300–320.

[34] Pınar Tözün, Brian Gold, and Anastasia Ailamaki. 2013. OLTP in Wonderland: Where Do Cache Misses Come from in Major OLTP Components?. In *DaMoN @ ACM SIGMOD*. Article 8, 6 pages.

[35] Pınar Tözün, Ippokratis Pandis, Cansu Kaynak, Djordje Jevdjic, and Anastasia Ailamaki. 2013. From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored. In *EDBT*. 17–28.

[36] Shang Wang, Peiming Yang, Yuxuan Zheng, Xin Li, and Gennady Pekhimenko. 2021. Horizontally Fused Training Array: An Effective Hardware Utilization Squeezer for Training Novel Deep Learning Models. *MLSys* 3 (2021), 599–623.

[37] Ross Wightman. 2019. PyTorch Image Models. https://github.com/rwightman/pytorch-image-models.

[38] Hui Zhang and Jeffrey Hollingsworth. 2019. Understanding the performance of GPGPU applications from a data-centric view. In *ProTools*. 1–8.

[39] Keren Zhou, Mark Krentel, and John Mellor-Crummey. 2020. A Tool for Top-down Performance Analysis of GPU-Accelerated Applications. In *PPoPP*. 415–416.