GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

# Profiling tree species classification with synthetic data and deep learning

Hauke Kirchner

Supervisor: Dorothea Sommer

April 27, 2023

# Abstract

Recent developments in artifical inteligence (AI) (Rombach et al. 2022; OpenAI 2023) were achieved with increasingly complex models (Schwartz et al. 2019). These achievements lead to the increasing usage of AI in a broad range of industries and research (Zhang et al. 2022), which results in high demand for compute power and a growing energy consumption (Dodge et al. 2022). Especially in environmental research, such as forest science, sustainability is fundamental, and the emerging use of AI should not lead to the opposite effect by increasing the researcher's carbon footprint. Further, most researchers depend on fast training and inference. For example, the model should predict tomorrow's weather before tomorrow. Therefore, the optimization of deep learning models is essential. The efficient profiling and evaluation of profiling outputs must be done as a first step of optimization. This project will evaluate tools and metrics for profiling a PyTorch model's training process. Here the use case of pre-training the PointNet model (Qi et al. 2016) with synthetic light detection and ranging (lidar) data for tree species classification will be used. It will be shown how a simple bottleneck during data loading can slow down the training process by factor 14 and how the profilers can help to identify this bottleneck. Further, standard profiling metrics for the given example will be shown and discussed.

# Contents

# List of Tables



# List of Figures



# List of Listings

# List of Abbreviations

**AI** artifical inteligence

**API** application programming interface

**CPU** central processing unit

**FLOPS** floating point operations per second

**GPU** graphics processing unit

**HPC** high-performance computing

**MACs** multiply-accumulate operations

**SCC** scientific compute cluster

**lidar** light detection and ranging

# 1 Introduction

Recent advances in deep learning, such as image (Rombach et al. 2022) and text generation (OpenAI 2023), lead to an increase in the number of AI publications in the world (Zhang et al. 2022). Most of the accuracy gains of these models result from increasingly complex models (Schwartz et al. 2019). From 2013 to 2019, the required compute power for training deep learning models increased by a factor of $300,000$ (Amodei and Hernandez 2018). This also leads to an increase in required energy and results in significant carbon dioxide emissions. Dodge et al. 2022 showed that the training process of a 6 billion parameter transformer model (trained for approximately 13% of the total training time) consumes $\sim$ 10M $CO_2$ grams. That is equivalent to a US household's yearly home energy demands per year. Further, this massive computing capacity is exclusively accessible by a few organizations, making the developed state-of-the-art models inaccessible to many researchers. To make AI more environmental-friendly and inclusive, Schwartz et al. (2019) proposed "Green AI" as an alternative to the accuracy-focused "Red AI". Besides accuracy as a measurement of performance, "Green AI" includes the cost of this model.

Also, in forest science, deep learning approaches gained popularity for diverse tasks and data sets (Hamedianfar et al. 2022), such as tree species classification based on lidar data (Liu et al. 2021, Seidel et al. 2021). Further, network architectures optimized for point cloud classification and segmentation were proposed (Qi et al. 2016). Often training data is a bottleneck for the application of neural networks. Especially in the field of environmental research, collecting data requires substantial effort. Therefore, Helios++ was developed by Esmorís et al. (2022) to generate synthetic lidar data. This synthetic lidar data can then be used for pre-training a model. In that way, valuable field data can be used for fine-tuning. As forest science inherently focuses on sustainability, the application of "Green AI" is obvious.

Here, a PyTorch workflow for tree species classification with PointNet and synthetic lidar data will be analyzed. Different profiling tools will be evaluated and tested to identify performance bottlenecks. Therefore, the following research objectives will be considered:

- Identification of **profiling tools** that can help to **optimize an existing PyTorch workflow**.
- As this tool should help scientists, the **usibility is highly important**. The most straightforward tool for doing the job is preferred.
- Unlike training benchmark suites like MLPerf, it will not be focused on benchmarking hardware but on optimizing an existing PyTorch workflow.

The code and documents produced during this course can be found in the accompanying GitHub repository[1].

[1] `https://github.com/haukekirchner/scap/`, accessed on:31.3.2023

# 2 Methods

## 2.1 Tree species classification based on synthetic lidar data

The use case for testing profiling options for PyTorch is lidar-based tree species classification. Several applications of deep learning approaches for analyzing lidar data were recently proposed ( Qi et al. 2016; Krisanski et al. 2021). As the computation cost for 3D convolutions is high, analyzing a workflow for lidar data processing is an ideal real-world example exploring the capabilities of PyTorch profilers.

In the project ForestCare[2], a pre-training workflow based on synthetic lidar data was developed to overcome the lack of available pre-trained neural networks for tree species classification. A detailed description of this workflow can be found in Figure 1. The code for pre-training the pointnet model can be found in the accompanying GitHub repository[3].

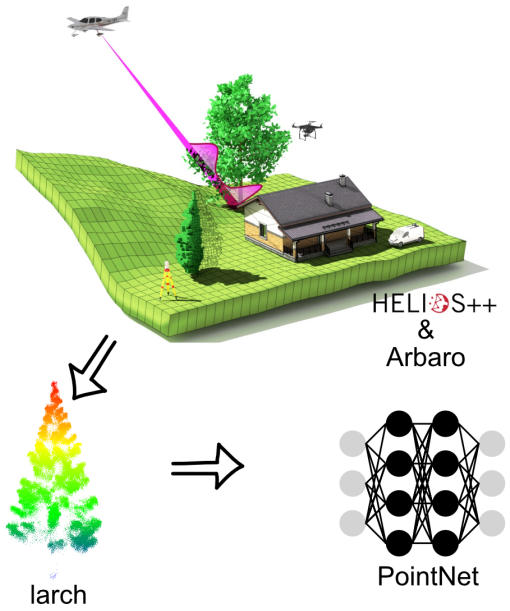



Figure 1: The workflow for generating synthetic lidar data starts with generating tree models. This is done with the software Arbaro[4] (Weber and Penn 1995). Based on these models, Helios++ (Esmorís et al. 2022) is used to simulate lidar data which is similar to data capture with the Zenmuse L1 sensor from DJI[5]. This generated data is used to pre-train a PointNet model proposed by Qi et al. (2016). The figure is adapted from Esmorís et al. (2022).

## 2.2 Fundamentals of software profiling

Software profiling is "a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes" (Wikibooks 2018). Metrics commonly of interest to identify performance bottlenecks are the number and elapsed time per function call and the memory consumption.

[2]https://hps.vi4io.org/research/projects/forestcare/start, accessed on: 13.03.2023
[3]`https://github.com/haukekirchner/scap/`, accessed on: 22.02.2023

Metrics traditionally used for software profiling are execution time and floating point operations per second (FLOPS) (Table 1). Execution time is a simple measure that reports a function call's total time. FLOPS refers to the number of floating point operations that can be executed per second and can be used to evaluate the model's efficiency.

With the advent of graphics processing unit (GPU)'s throughput and multiply-accumulate operations (MACs) gained importance (Verma and Radhakrishnan 2019). Throughput is essential as deep learning models need to analyze massive amounts of data to find generalizable patterns and can report on the model's speed. Analog to FLOPS MACs do report on the software's efficiency. "Most modern hardware architectures use FMA [fused multiply-add] instructions for operations with tensors. FMA computes $a * x + b$ as one operation. Roughly $GMACs = 0.5 * GFLOPs$"[6].

Further, profiling memory consumption is critical, as deep learning applications depend on massive amounts of data. Therefore, efficiently using GPU's internal memory is critical to the overall performance.

Other important profiling measures in deep learning, which are not considered in this report, are Time to Accuracy (TTA) and Average Time to Multiple Thresholds (ATTMT) (Verma and Radhakrishnan 2019). These metrics were out of scope as the tested tools (Section 2.3.1) do not provide this information. However, setting up a workflow to measure these metrics based on the tools tested here is possible by combining them with early stopping approaches.

| metric | purpose |
|---|---|
| Execution time<br>FLOPS | traditionally |
| Throughput: $\frac{images}{sec}$<br>MACs | with the advent of GPUs |

Table 1: Overview of performance metrics that were used for profiling.

## 2.3 Profiling tools

### 2.3.1 Overview of profiling tools

Many different profiling tools can be used to gain insights into the PyTorch workflow presented in Section 2.1. Table 2 lists some of the considered options. General profiling tools such as Intel's Vtune[7] or the open-source software LIKWID[8] can be used to collect valuable metrics of any software.

However, tools were developed specifically to the needs one needs to profile PyTorch workflows. One popular tool is the in-build PyTorch profiler, which can be combined with Tensorboard[9]. This graphical visualization can be used for beginners and experts to optimize their workflows, as it provides automatic performance recommendations and detailed views such as the call stack or the trace view. Another tool specifically designed

[6] `https://github.com/sovrasov/flops-counter.pytorch/issues/16#issuecomment-518585837` , accessed on: 22.11.2022

[7] `https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html`, accessed on: 22.11.2022

[8] `https://github.com/RRZE-HPC/likwid`, accessed on: 22.11.2022

[9] `https://www.tensorflow.org/tensorboard/`, accessed on: 16.03.2023

for PyTorch is DeepSpeed[10]. The FlopsProfiler was used here, as the Pytorch Profiler lacks the option to analyze FLOPS.

General profiling tools could not provide automatic recommendations or useful visualizations to highlight critical aspects and bottlenecks, as they do not have insights into PyTorch. Consequently, more expert knowledge is required to use tools like LIKWID or Vtune. As this project's research goal was identifying easy-to-use tools, this report focuses on the PyTorch Profiler and DeepSpeed's FlopsProfiler.

| tool | metrics | scope |
|---|---|---|
| PyTorch Profiler With TensorBoard | performance metrics (e.g. time, memory) | PyTorch |
| Deepspeed/FlopsProfiler | FLOPS | |
| Vtune | general | Intel-only |
| likwid | performance metrics | general |

Table 2: Ovierview of tools used for profiling.

### 2.3.2 TensorBoard

Tensorboard is "TensorFlow's visualization toolkit"[11] and is commonly used to visualize the loss and accuracy during the training process of a neural network. For most people starting with deep learning, Tensorboard is one of the first tools to get insights into the deep learning workflow. Unlike the name implies, this tool is not limited to using TensorFlow but can also be used with PyTorch.

### 2.3.3 PyTorch - Profiler

A more advanced option to get insights into a PyTorch Workflow is the PyTorch - Profiler, which is "a simple profiler API that is useful when user needs to determine the most expensive operators in the model"[12]. With the help of this tool, one can collect general performance metrics, identify expensive operators, and track kernel activity. The PyTorch - Profiler can be used with Tensorboard to visualize the profiling outputs[13].

### 2.3.4 Deepspeed - FlopsProfiler

DeepSpeed is a collection of tools that can help to optimize deep learning workflows[14]. This project's primary interest was on the FlopsProfiler[15]. As the name implies, the FlopsProfiler can measure the efficiency (FLOPS) for both the training and inference of a deep learning model. Further, it can profile the model's speed (latency, throughput). Contrary to the PyTorch Profiler, the output is text-based without a visualization tool.

[10] `https://www.deepspeed.ai/tutorials/flops-profiler/`, accessed on: 16.03.2023

[11] `https://www.tensorflow.org/tensorboard`, accessed on: 20.3.2023

[12] `https://pytorch.org/tutorials/recipes/recipes/profiler_recipe.html`, accessed on: 20.3.2023

[13] `https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html`, accessed on: 20.3.2023

[14] `https://www.deepspeed.ai/`, accessed on: 20.3.2023

[15] `https://www.deepspeed.ai/tutorials/flops-profiler/`, accessed on: 20.3.2023

## 2.4 Experiments

For the design of the experiments, two main questions were considered. What is the benefit of using different accelerators? What is the cost/overhead of using profilers?

Therefore, the three presented tools (Section 2.3.1) were tested on all available GPUs of the scientific compute cluster (SCC)[16]. The main focus is on comparing the PyTorch - Profiler and DeepSpeed's Flops Profiler, as these tools are promoted primarily to optimize the performance of deep learning workflows. In addition, Tensorboard was included to analyze the overhead of this relatively simple and commonly used tool and to have a baseline for simple profiling tools.

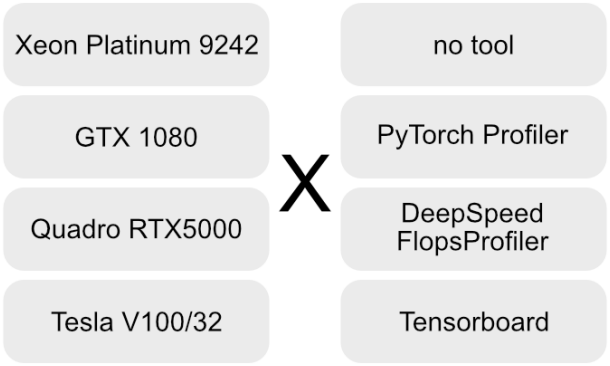



Figure 2: In total, 16 different runs were planned by testing all possible combinations of tools with available GPUs and one central processing unit (CPU). Four more runs were performed by adding experiments according to the data loader process and different batch sizes (Table 3).

Based on this experiment design, 16 different runs were executed (Table 3). Two more runs were included to test the performance recommendations of the PyTorch Profiler (runs 17 and 20). Another two runs show the effect of an unoptimized data loading process, where the raw point cloud is loaded every time, and the point sampling process happens afterward (runs 18 and 19) (more details can be found in Section 3.1).

[16] `https://www.gwdg.de/web/guest/hpc-on-campus/scc`

| run | node | tool | job_id | is_valid | experiment |
|---|---|---|---|---|---|
| 1 | scc_cpu | no-tool | 14629421 | TRUE | |
| 2 | scc_cpu | tensorboard | 14629426 | TRUE | |
| 3 | scc_cpu | profiler-torch | 14650740 | TRUE | |
| 4 | scc_cpu | deepspeed | 14617521 | FALSE | |
| 5 | scc_gtx1080 | no-tool | 14619617 | TRUE | |
| 6 | scc_gtx1080 | tensorboard | 14615343 | TRUE | |
| 7 | scc_gtx1080 | profiler-torch | 14650076 | TRUE | |
| 8 | scc_gtx1080 | deepspeed | 14615344 | TRUE | |
| 9 | scc_rtx5000 | no-tool | 14619618 | TRUE | |
| 10 | scc_rtx5000 | tensorboard | 14617172 | TRUE | |
| 11 | scc_rtx5000 | profiler-torch | 14650079 | TRUE | |
| 12 | scc_rtx5000 | deepspeed | 14617171 | TRUE | |
| 13 | scc_v100 | no-tool | 14619619 | TRUE | |
| 14 | scc_v100 | tensorboard | 14617203 | TRUE | |
| 15 | scc_v100 | profiler-torch | 14650080 | TRUE | |
| 16 | scc_v100 | deepspeed | 14617202 | TRUE | |
| 17 | scc_gtx1080 | profiler-torch | 14650758 | TRUE | batch-size-64 |
| 18 | scc_gtx1080 | profiler-torch | 14650750 | TRUE | sample-points |
| 19 | scc_cpu | profiler-torch | 14657599 | TRUE | sample-points |
| 20 | scc_gtx1080 | profiler-torch | 14650759 | TRUE | batch-size-128 |

Table 3: In total, twenty different experiments were performed. The first 16 experiments are directly derived from the experiment design presented in Figure 2. Experiments 18 and 19 were preliminary work to optimize the data loading process before all experiments were performed ( Section 3.1). Experiments 17 and 20 were done based on the PyTorch - Profiler's performance recommendations( Section 3.3.2).

# 3 Results

## 3.1 Optimization of the data loading process

As we were aware that due to the massive amount of data we want to process, the data loading process is a bottleneck, it was decided to optimize this aspect before doing all remainig runs presented in Section 2.4. This way, it was possible to run more experiments without wasting valuable GPU hours for all users on the SCC. The idea is to speed up the data-loading process by migrating the point sampling to a pre-processing workflow. Point sampling means selecting the number of points per tree the neural network architecture pointnet expects as input. Thus, it is possible to reduce the data size by a factor of 1714.55, from $931GB$ to $543MB$.

As the data loading process was the major bottleneck, reducing data that needs to be loaded during the training process significantly speeds up the whole training workflow (Figure 3). For example, while it still took 4hours to train the network on the GTX 1080 in the unoptimized version, it only took 16minutes for the optimized version. Similarly, the training process on the CPU was decreased from 14 hours and 15 minutes to 2 hours and 49 minutes. This means a speed up for the CPU by a factor of 5 and for the GPU

(GTX 1080) a speed up by a factor of 14.

So far, the optimization problem has been solved by expert knowledge about the workflow and the data. Simultaneously the PyTorch - Profiler was tested to identify the bottleneck in the data loader. Unfortunately, the overview did not help identify the issue. Only an analysis of the trace view revealed that the function for reading the data takes up a large part of the time and thus slows down the workflow (Figure 8). More details on the PyTorch - Profiler can be found in Section 3.3.2.

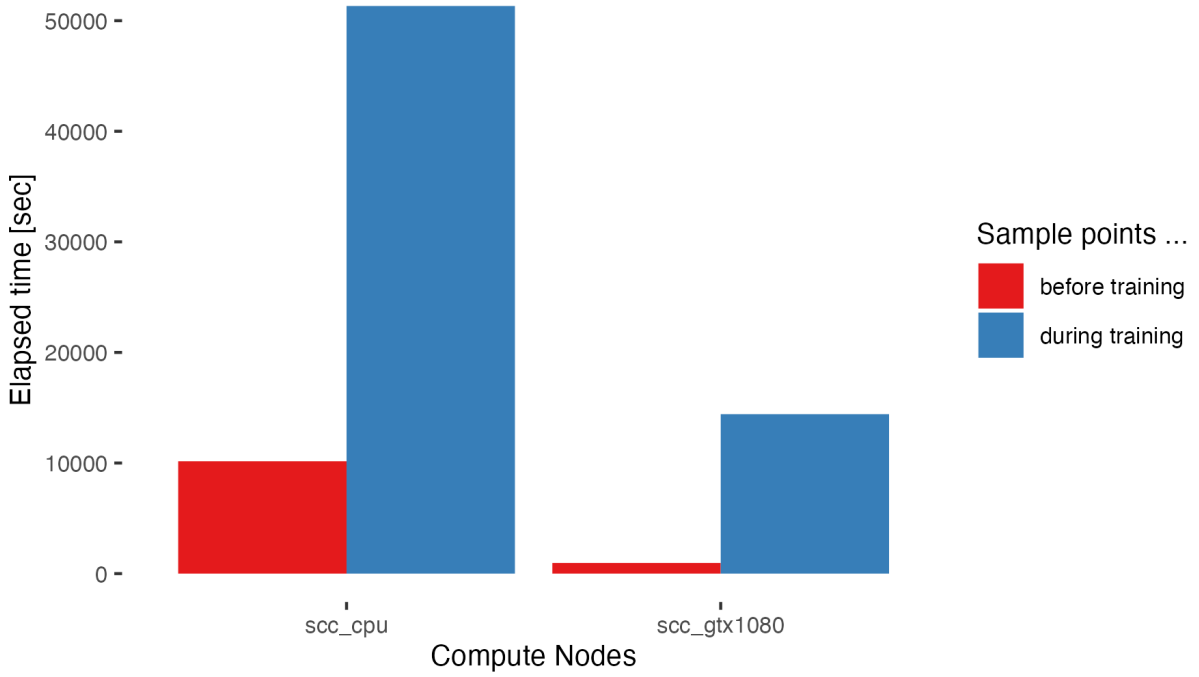



Figure 3: By migrating the point sampling process from the data loader to a pre-processing step, the elapsed time for the training runs was decreased significantly.

## 3.2 Effect of the accelerators and profiling tools on the runtime

The simplest method to get a first estimate for the performance of all different runs is to look at the elapsed time for each analysis. On the SCC jobs are executed via SLURM[17] which provides the tool `sacct`[18] for analyzing accounting information, such as elapsed time of a job. The raw output and the command to get this information can be found in the Appendix ( Listing 4, Listing 6).

Most apparent is the difference between runs on the CPU and the GPUs (Figure 4). The mean runtime for jobs on CPUs is 161.16 minutes, and for runs on the GPUs, 8.55 minutes. Further, it is notable that DeepSpeed is not compatible with CPUs. A more detailed view on runs using the GPUs (Figure 5) shows that the GTX 1080 node is slower (mean runtime of 13 minutes) than the RTX 5000 node (mean runtime of 7.65 minutes). The fastest node is the V100, with a mean runtime of 5.02 minutes.

No clear trend can be observed regarding the overhead of the different profiling tools. Generally, the PyTorch - Profiler creates the highest overhead. However, on nodes GTX

[17] `https://slurm.schedmd.com/`, accessed on: 21.3.2023
[18] `https://slurm.schedmd.com/sacct.html`, accessed on: 21.3.2023

1080 and RTX 5000, runs using DeepSpeed's FlopsProfiler or TensorBoard are faster than baseline runs without any tools.

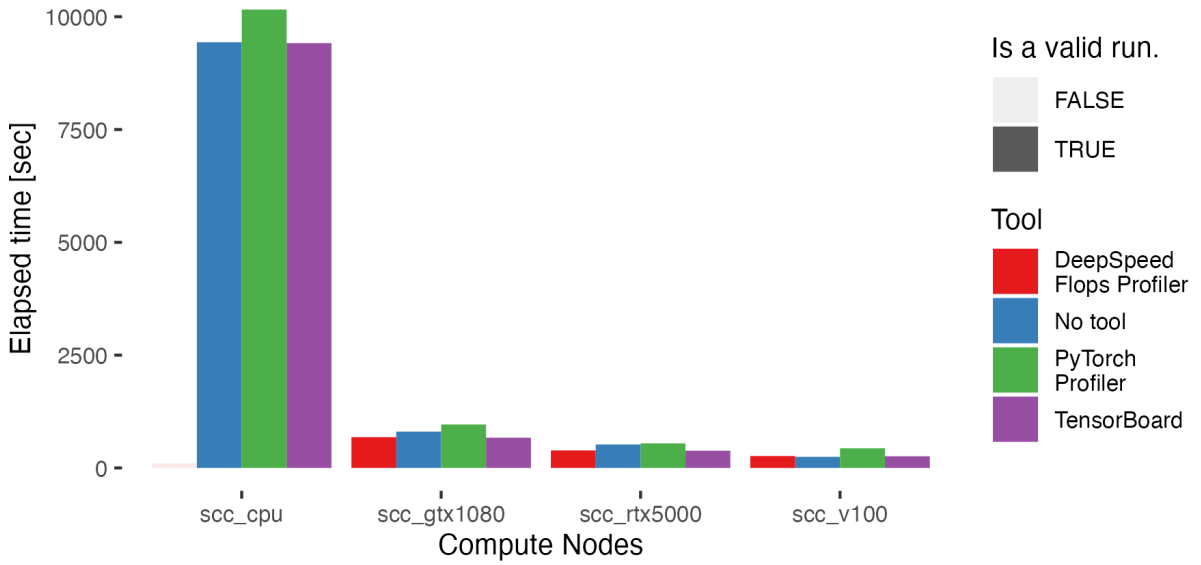



Figure 4: The runtime of all runs testing the combination of tools and GPU and CPU partitions (Figure 2).

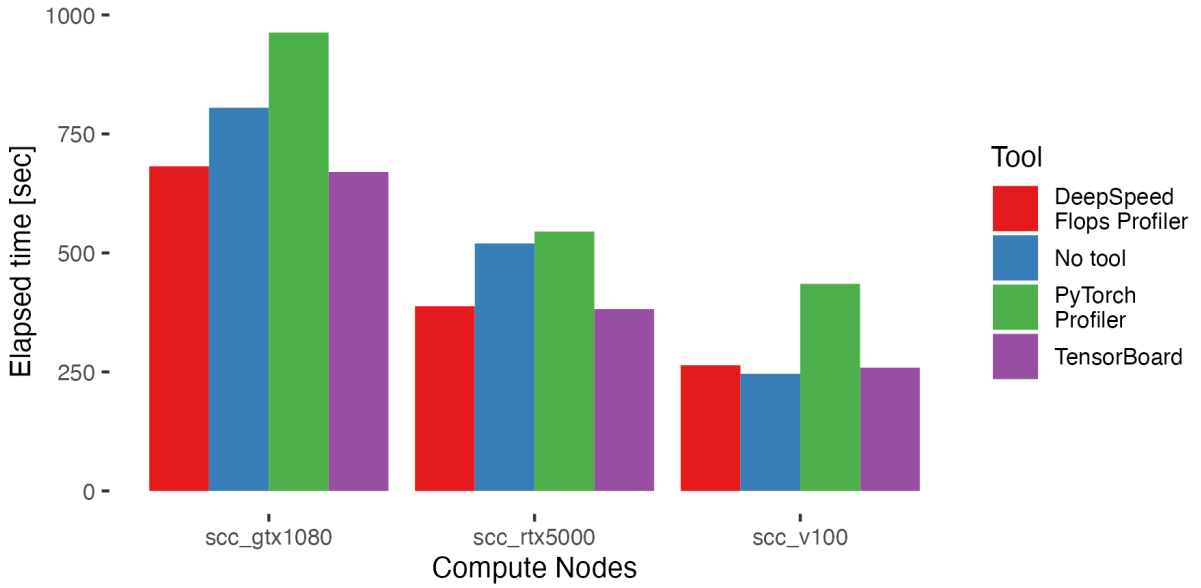



Figure 5: The runtime of all runs testing the combination of tools and GPU partitions - excluding CPU partitions (Figure 2).

## 3.3 Results of the profiling tools

Following the results and outputs from all tested profilers are described individually.

### 3.3.1 TensorBoard

The minimal example for using a TensorBoard to keep track of the model's training process provides information about the model's accuracy, the loss per epoch, and the loss per minibatch (Figure 10). It can be seen that over the training time, the accuracy constantly increases, and the loss decreases.

### 3.3.2 PyTorch - Profiler

When visualizing the profiling results of the PyTorch Profiler with Tensorboard, the first view visible is the "Overview". This view is one of six different views, showing different details of the profiler output. If not labeled differently here, the output of the PyTorch - Profiler run on the GTX 1080 with optimized batch size of 64 (run 17 Table 3) will be discussed.

The "Overview" (Figure 6) gives "a high-level summary of the model performance"[19]. It can be subdivided into five different panes. On top, there are the panes "Configuration", "GPU Summary", and "Execution Summary". In "Configuration", information about the number(s) of workers and device type (CPU or GPU) are listed. The "GPU Summary" provides information about the GPU utilization by showing different metrices[20], such as "GPU Utilization" and the "Estimated Stream Multiprocessor Efficiency". Details on the metrics can be found in the corresponding Github repository[21]. Here the metrics indicate that the utilization of the GPU is relatively low for the tested workflow. The same can be seen in the "Execution Summary", where the time spent on the kernel is low. Here, the profiler misses detecting the DataLoader operations and sums up most of the time spent for the workflow in the category "Other". The "Step Time Breakdown" visualizes the same information as the "Execution Summary" per step. However, in the tested example, just the first step was visualized. The last pane, "Performance Recommendation", provides hints for optimizing the PyTorch workflow. Here, the profiler detected a low GPU utilization where a common strategy is to increase the batch size. Testing this recommendation showed that increasing the batch size from 32 to 64 improved the GPU utilization by 10% (Figure 7). As the utilization is still low, the performance recommendation remains the same. Nevertheless, increasing the batch size to 64 lead to a decrease in GPU utilization.

[19] `https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html`, accessed on: 20.3.2023

[20] `https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html`, accessed on: 22.3.2023

[21] `https://github.com/pytorch/kineto/blob/main/tb_plugin/docs/gpu_utilization.md`, accessed on: 27.22023

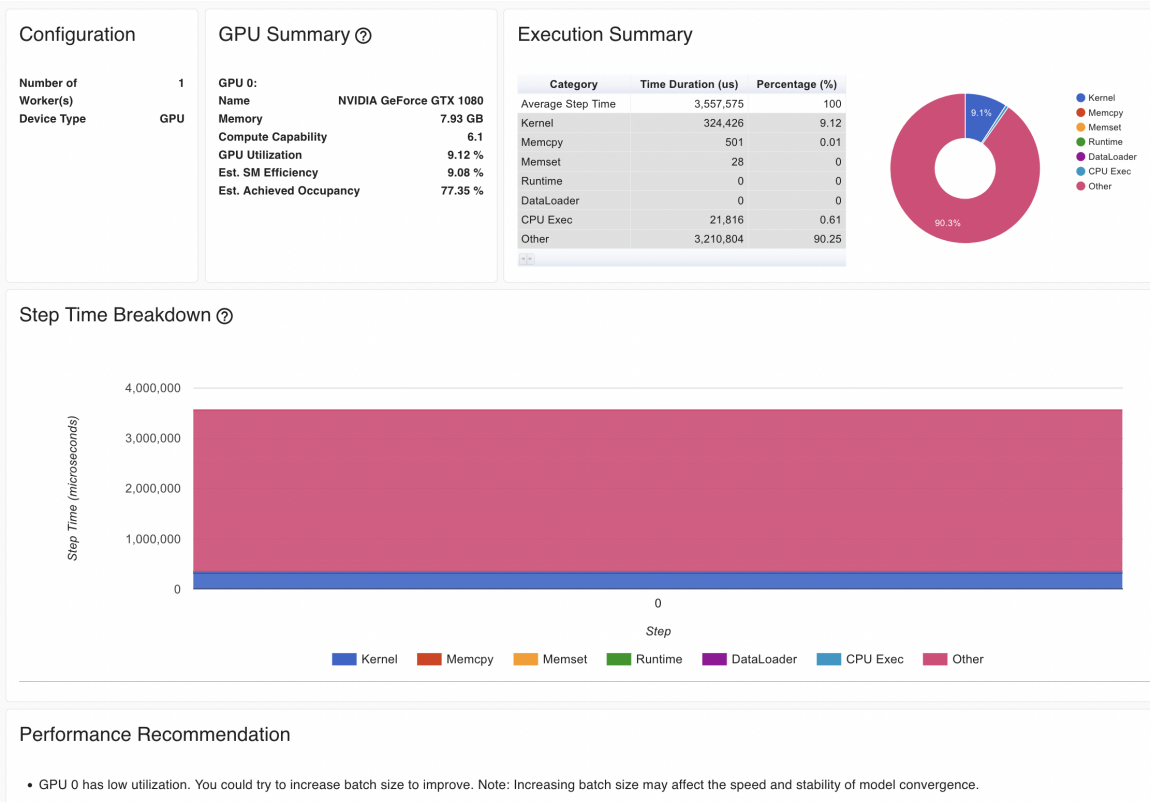



Figure 6: Visualizing the profiling results of the PyTorch - Profiler with Tensorboard provides an overview page. This page summarizes important metrics of the workflow, such as GPU utilization. Further, it offers performance recommendations on how to adress potential bottlenecks. This "Overview" shows the result for run 17 in Table 3.

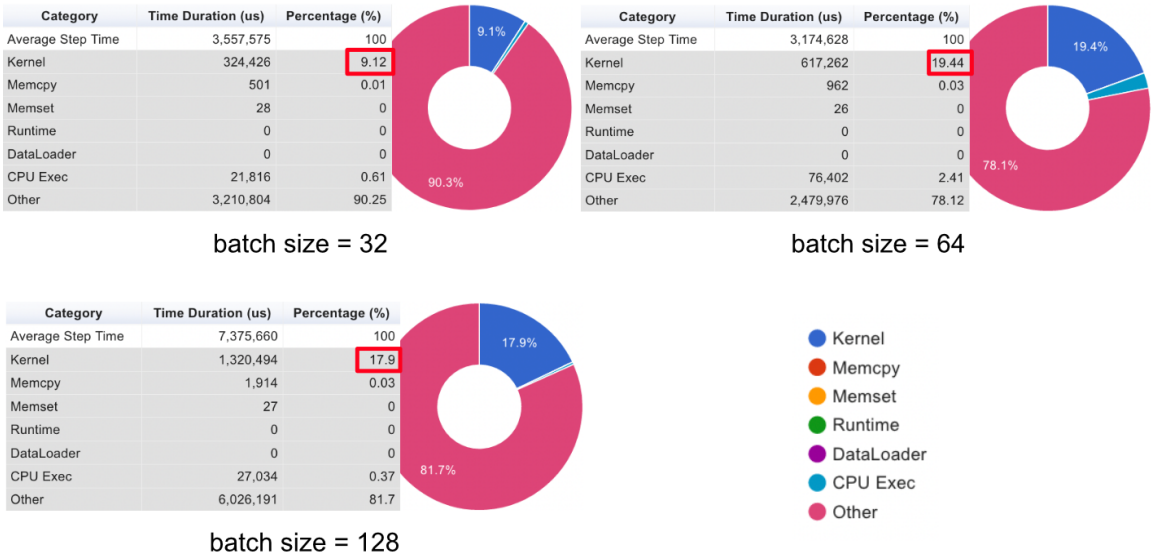



Figure 7: For the tested training workflow the performance recommendations of the PyTorch - Profiler suggest to increase the batch size. The effect of the batch size on the GPU utilization can be seen for a batch size of 32 (run 7 in Table 3), 64 (run 17 in Table 3), 128 (run 20 in Table 3). The full view for these experiments can be seen in Figure 6, Figure 11, and Figure 12.

The "Trace View" of the PyTorch - Profiler shows the usage of operators and GPU

kernels on a timeline (Figure 13). By clicking on operators, details, such as start time or end time, can be seen (Figure 8). Further, this view provides insights into operator connections ("incoming flow"). As already stated in the "Overview", it can be seen that the GPU is most of the time waiting for other operators on the CPU to finish. That way, a significant part of the GPU time is wasted. The operator identified as a bottleneck is the data reading operation. This is even worse for the not optimized workflow (run 18 Table 3) where the raw point cloud is loaded and afterward, the point sampling is done (Figure 9). For one reading operation, the optimized data loading strategy takes 3,023 milliseconds, and the unoptimized variation requires 58,064 milliseconds.

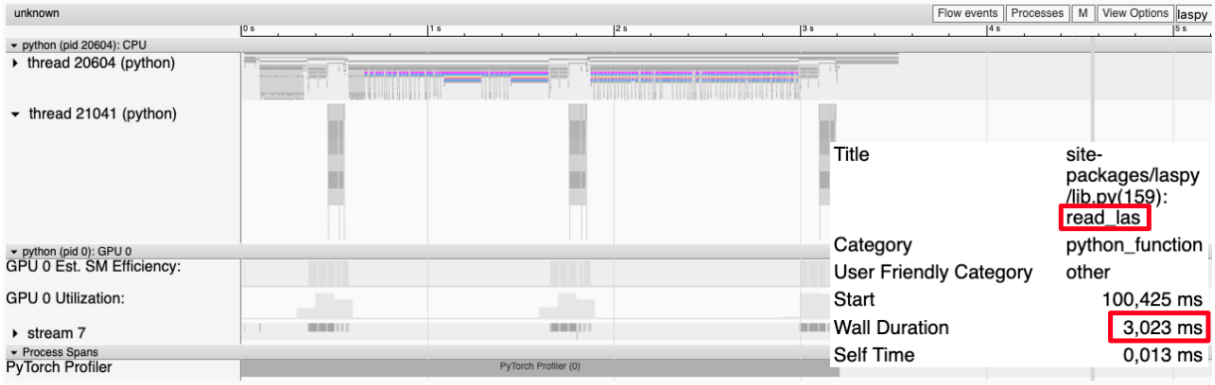



Figure 8: The "Trace View" provides insights into the operators and GPU kernels over time. Here the read operation of the lidar data based on pre-sampled point clouds is analyzed.

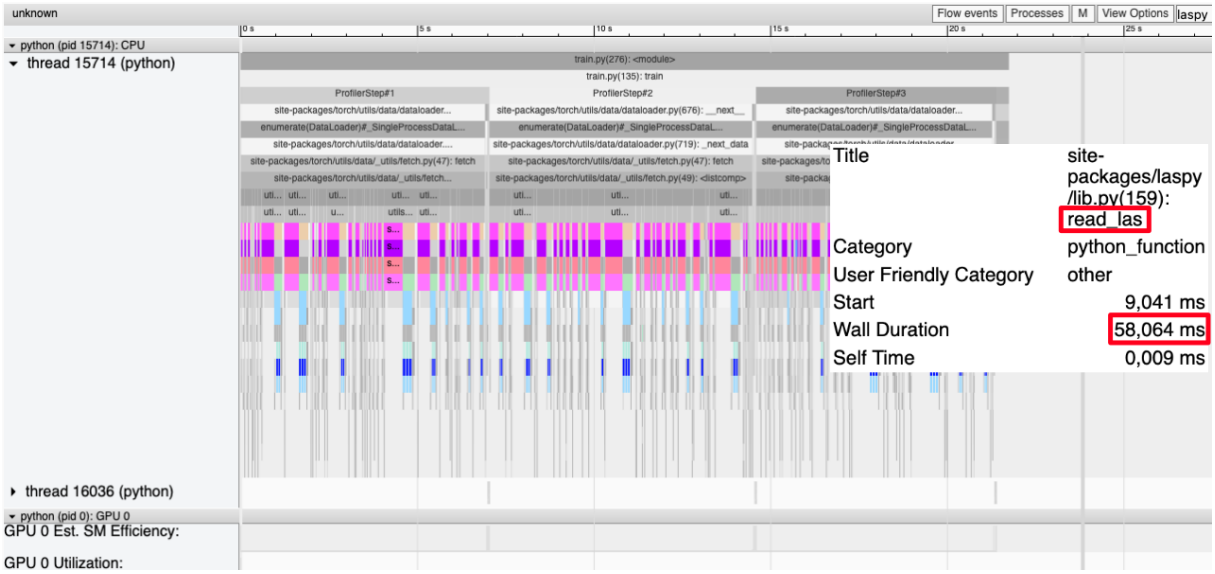



Figure 9: The "Trace View" of the unoptimized workflow (run 18 Table 3), loading the raw lidar data, shows that a significant part of the time is spend on the reading opertions.

Further views provided by the PyTorch - Profiler are "Operator", "GPU Kernel", "Memory", and "Module". The "Operator" view provides insights into the profiling results of PyTorch's operators (Figure 14). Self-time refers to the duration of the operator without child operators, which are included in the total time. For example, it can be seen

that the operator `aten::copy_` is expensive. Also, the View CallStack can be shown in this view (Figure 15). The "GPU Kernel" view shows for all kernels the time that was spent on the GPU (Figure 16). More information about the metrics used in this view can be found on the official tutorials website[22]. The "Memory" view gives information about memory usages, such as allocation and release events. This view consists of a "memory curve graph, memory events table, and memory statistics table"[23] (Figure 17). The "Module" view shows profiling results (occurrences, time) per PyTorch module, such as the whole PointNet or the submodule Transform (Figure 18).

### 3.3.3 DeepSpeed - FlopsProfiler

The output of the DeepSpeed - Flops Profiler is a textfile (Listing 5), which provides metrics on different levels of the network. The first part summarizes metrics for the whole neural network, such as the model parameters, MACs, number of floating-point operations (flops), floating-point operations per second (FLOPS), and latency. Precise definitions of these metrics can be found in the output file (Listing 5). For this example run (run 8 Table 3), a total of 14.07 GMACs (29.04 Gflops) was measured. Dividing this by the forward propagation latency shows that for the whole model, the floating-point operations per second are 349.1 GFLOPS.

```
Profile Summary at step 5:
Notations:
data parallel size (dp_size), model parallel size(mp_size),
number of parameters (params), number of multiply-accumulate operations(MACs),
number of floating-point operations (flops), floating-point operations per second (FLOPS),
fwd latency (forward propagation latency), bwd latency (backward propagation latency),
step (weights update latency), iter latency (sum of fwd, bwd and step latency)

params per gpu:                                                 3.46 M
params of model = params per GPU * mp_size:                     3.46 M
fwd MACs per GPU:                                               14.07 GMACs
fwd flops per GPU:                                              29.04 G
fwd flops of model = fwd flops per GPU * mp_size:               29.04 G
fwd latency:                                                    83.19 ms
fwd FLOPS per GPU = fwd flops per GPU / fwd latency:            349.1 GFLOPS
```

Listing 1: DeepSpeed - FlopProfiler: Summary

The second part of the DeepSpeed - FlopsProfiler's output (Listing 2) reports on the top modules in different depths of the model. Top modules are defined based on the number of parameters, MACs, and forward propagation latency. In-depth 0 the whole

[22] https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html, accessed on: 20.3.2023

[23] https://pytorch.org/tutorials/intermediate/tensorboard_profiler_tutorial.html, accessed on: 20.3.2023

PointNet model is considered so that the profiling results are the same as in the first part of the output (Listing 1). In depth 1 it can be seen that most of the resources are spent on the Transform module. For example, 14.05 GMACs out of 14.07 GMACs are spent on this module. The top module in depth 2 is the Tnet with 9.34 GMACs.

```
Top 1 modules in terms of params, MACs or fwd latency at different model depths:
depth 0:
    params      - {'PointNet': '3.46 M'}
    MACs        - {'PointNet': '14.07 GMACs'}
    fwd latency - {'PointNet': '83.19 ms'}
depth 1:
    params      - {'Transform': '2.8 M'}
    MACs        - {'Transform': '14.05 GMACs'}
    fwd latency - {'Transform': '82.08 ms'}
depth 2:
    params      - {'Tnet': '2.66 M'}
    MACs        - {'Tnet': '9.34 GMACs'}
    fwd latency - {'Tnet': '58.43 ms'}
```

Listing 2: DeepSpeed - FlopProfiler: Aggregated Profile per GPU

In the detailed profile, the metrics are reported for all submodules of the model (Listing 3). Besides the metrics given for higher-level summarization before, e.g., metrics for the whole PointNet, all metrics for submodules are listed. The order of the metrics is params, percentage of total params, MACs, percentage of total MACs, fwd latency, percentage of total fwd latency, fwd FLOPS (Listing 5). For example, it can be seen that the second convolution of the Tnet is a 1D convolution[24] with 268.44 MMACs.

```
PointNet(
  3.46 M, 100.00% Params, 14.07 GMACs, 100.00% MACs, 83.19 ms, 100.00% latency, 349.1 GFLOPS,
  (transform): Transform(
    2.8 M, 80.94% Params, 14.05 GMACs, 99.85% MACs, 82.08 ms, 98.67% latency, 353.28 GFLOPS,
    (input_transform): Tnet(
      803.08 k, 23.19% Params, 4.59 GMACs, 32.63% MACs, 27.06 ms, 32.53% latency, 350.87 GFLOPS,
      (conv1): Conv1d(256, 0.01% Params, 6.29 MMACs, 0.04% MACs, 313.28 us, 0.38% latency, 46.86 GFLOPS, 3, 64, kernel_size=(1,), stride=(1,))
      (conv2): Conv1d(8.32 k, 0.24% Params, 268.44 MMACs, 1.91% MACs, 374.32 us, 0.45% latency, 1.45 TFLOPS, 64, 128, kernel_size=(1,),
      stride=(1,))
```

Listing 3: DeepSpeed - FlopProfiler: Detailed Profile per GPU

[24] https://pytorch.org/docs/stable/generated/torch.nn.Conv1d.html, accessed on: 23.3.2023

### 3.3.4 Usability of the different tools

The tools provide different application programming interface (API)s, such as a context manager, a decorater, or a Python module[25]. For example, the DeepSpeed Flops Profiler can be activated in DeepSpeed's runtime, which would not require to adapt the Python code at all. However, for this project all tools were used as Python packages. This requires to insert a few lines of code. The important parts of the code and their placement in the training code is shown in Listing 7, Listing 8, and Listing 9.

# 4 Discussion

Generally, the used profilers, PyTorch Profiler and DeepSpeed's FlopsProfiler, provide interesting and valuable insights into the training process of a PyTorch model, which can be used for optimization purposes. Nevertheless, also with the beginner-friendly TensorBoard approach, tracking the accuracy and loss, details of the training process can be analyzed.

The data loading process was the most apparent bottleneck for the PointNet workflow analyzed for this report. This bottleneck could be seen in the PyTorch Profiler's logging output or using expert knowledge about the size of the input data and the point data sampling technique within the data loader. In Figure 3, two essential aspects for optimizing the PointNet workflow can be seen. First, the speedup for each compute node by migrating the point sampling process to a pre-processing step of the workflow. Second, there is a substantial difference between the runtimes of the CPU and GPU. Here it can be seen how important the use of accelerators, such as GPUs, is for training deep neural networks. Further, Figure 5 shows the performance gain one can get by using faster GPU nodes on the cluster and the compute overhead each tool is generating due to its measurement activities and the writing of logging files. The overhead created by the PyTorch Profiler seems to be the highest. Interestingly, the runs without any tool on the GTX 1080 and RTX 5000 are slower than those profiled with DeepSpeed's FlopsProfiler and TensorBoard. However, the mean of several identical runs would be required for a reliable performance comparison. Additionally, a performance comparison between nodes on the SCC is more complicated because the compute nodes are shared resources, and computations by other users could affect the measurements.

An easy starting point for profiling the training process of a PyTorch model is TensorBoard. Most will be familiar with this tool as it is commonly used in beginner introductions to deep learning. In addition, the user-friendly visualization as a dashboard and the interactivity increases the accessibility. For example, the loss and accuracy curves shown in TensorBoard make overfitting visible, and the training process could be stopped manually or by early stopping rules. Additionally, checkpoints can prevent the re-training of models in such a case. These easy strategies for reducing the cost of training neural networks are also recommended by the authors of the paper Green AI (Schwartz et al. 2019) in their interview at the Practical AI podcast[26].

Also, the PyTorch Profiler profits from intuitive visualizations in Tensorboard. Especially the Overview page, with its performance recommendations, can be handy for

[25] `https://www.alcf.anl.gov/sites/default/files/2022-08/Profiling.pdf`, accessed on: 23.3.2023

[26] `https://changelog.com/practicalai/124`, accessed on: 30.3.2023

beginners. Nevertheless, the effect of these recommendations should still be checked, as shown in Figure 7. Further, the Trace view provided essential insights into the training workflow, highlighting the most resource intense operators. This helped identify the data reading as a bottleneck of the workflow. More complicated is the evaluation of the other views, which report on the model's performance on a low level and sometimes require knowledge about the implementation of PyTorch and the model's architecture to make good use of it.

As the PyTorch Profiler lacks the feature to measure FLOPS, DeepSpeed's FlopsProfiler was additionally tested. The text-based logging file is easy to interpret for high-level summarizations of the model's performance (Listing 5). The "Detailed Profile per GPU" is less intuitive, which includes the measurements for all submodules in PyTorch. However, the "Aggregated Profile per GPU" already highlights the most expensive modules in different depths of the model. The analysis of the details view can then be limited to those modules. Compared to example outputs in the official tutorial[27], some metrics are unfortunately not reported, such as the model's throughput. Further, there were issues about the compatibility with 3D convoultions[28], which shows the importance of testing these tools with models one is interested in and not only the provided examples.

All tested tools were easy to use, and the effort to get them running was relatively low. However, they have some bugs or unexpected behaviors (just one step in the PyTorch Profiler, missing metrics in the FlopsProfiler), which could be reasoned by a wrong usage or actual bugs in the software. Unfortunately, figuring this out was out of the scope of this report.

# 5 Conclusion

It is well known that profiling tools help to identify bottlenecks in software. The same was proven true for the tested tools in profiling the training process of PyTorch models. By identifying the major bottleneck in the data loading process, it was possible to optimize this step easily and to achieve a speed up by a factor of 14 on the GTX 1080. Achieving higher performance requires substantial expert knowledge of PyTorch and the network architecture. This can be both, a barrier for practitioners or a motivation to learn more about the technical backgrounds. Because of the given workflow we were interested in, the focus was on the training process. However, inference can also be measured with these tools, and due to the potentially higher number of inference runs, this can be even more important in a real-world example. Further, the DeepSpeed framework provides many more options for optimizing PyTorch workflows, which are very interesting to test in future projects.

[27] `https://www.deepspeed.ai/tutorials/flops-profiler/`, accessed on: 24.3.2023
[28] `https://github.com/microsoft/DeepSpeed/issues/1467`, accessed on: 24.3.2023

# References


Amodei, Dario and Danny Hernandez (2018). *Introduction to Software Engineering — Wikibooks, The Free Textbook Project*. [Online; accessed 30-March-2023]. URL: https://openai.com/research/ai-and-compute.

Dodge, Jesse et al. (2022). *Measuring the Carbon Intensity of AI in Cloud Instances*. DOI: 10.48550/ARXIV.2206.05229. URL: https://arxiv.org/abs/2206.05229.

Esmorís, Alberto M. et al. (2022). "Virtual LiDAR Simulation as a High Performance Computing Challenge: Toward HPC HELIOS++". In: *IEEE Access* 10, pp. 105052–105073. DOI: 10.1109/ACCESS.2022.3211072.

Hamedianfar, Alireza et al. (Feb. 2022). "Deep learning for forest inventory and planning: a critical review on the remote sensing approaches so far and prospects for further applications". In: *Forestry: An International Journal of Forest Research* 95.4, pp. 451–465. ISSN: 0015-752X. DOI: 10.1093/forestry/cpac002. eprint: https://academic.oup.com/forestry/article-pdf/95/4/451/45293980/cpac002.pdf. URL: https://doi.org/10.1093/forestry/cpac002.

Krisanski, Sean et al. (2021). "Sensor Agnostic Semantic Segmentation of Structurally Diverse and Complex Forest Point Clouds Using Deep Learning". In: *Remote Sensing* 13.8. ISSN: 2072-4292. DOI: 10.3390/rs13081413. URL: https://www.mdpi.com/2072-4292/13/8/1413.

Liu, Maohua et al. (2021). "Tree species classification of LiDAR data based on 3D deep learning". In: *Measurement* 177, p. 109301. ISSN: 0263-2241. DOI: https://doi.org/10.1016/j.measurement.2021.109301. URL: https://www.sciencedirect.com/science/article/pii/S0263224121003043.

OpenAI (2023). *GPT-4 Technical Report*. arXiv: 2303.08774 [cs.CL].

Qi, Charles Ruizhongtai et al. (2016). "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: *CoRR* abs/1612.00593. arXiv: 1612.00593. URL: http://arxiv.org/abs/1612.00593.

Rombach, Robin et al. (2022). "High-Resolution Image Synthesis with Latent Diffusion Models". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. URL: https://github.com/CompVis/latent-diffusionhttps://arxiv.org/abs/2112.10752.

Schwartz, Roy et al. (2019). "Green AI". In: *CoRR* abs/1907.10597. arXiv: 1907.10597. URL: http://arxiv.org/abs/1907.10597.

Seidel, Dominik et al. (2021). "Predicting Tree Species From 3D Laser Scanning Point Clouds Using Deep Learning". In: *Frontiers in Plant Science* 12. ISSN: 1664-462X. DOI: 10.3389/fpls.2021.635440. URL: https://www.frontiersin.org/articles/10.3389/fpls.2021.635440.

Verma, Snehil and Ramesh Radhakrishnan (2019). *METRICS FOR MACHINE LEARNING WORKLOAD BENCHMARKING*. Available online: https://snehilverma41.github.io/Metrics_ML_FastPath19.pdf (Accessed on: 22.02.2023).

Weber, Jason and Joseph Penn (1995). "Creation and Rendering of Realistic Trees". In: *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: Association for Computing Machinery, pp. 119–128. ISBN: 0897917014. DOI: 10.1145/218380.218427. URL: https://doi.org/10.1145/218380.218427.

Wikibooks (2018). *Introduction to Software Engineering — Wikibooks, The Free Textbook Project*. [Online; accessed 16-March-2023]. URL: https://en.wikibooks.org/w/index.php?title=Introduction_to_Software_Engineering&oldid=3457910.

Zhang, Daniel et al. (2022). *The AI Index 2022 Annual Report*. arXiv: 2205.03468 [cs.AI].

```
JobID|Elapsed
14615343|00:11:10
14615344|00:11:22
14617171|00:06:28
14617172|00:06:22
14617202|00:04:24
14617203|00:04:19
14617521|00:01:40
14618941|00:20:02
14619617|00:13:25
14619618|00:08:40
14619619|00:04:06
14629421|02:37:14
14629426|02:36:55
14650076|00:16:03
14650079|00:09:05
14650080|00:07:15
14650740|02:49:19
14650750|04:00:18
14650758|00:14:47
14650759|00:15:11
14657599|14:15:29
```

Listing 4: Output produced by Listing 4, which provides the elapsed time for each job on the SCC.

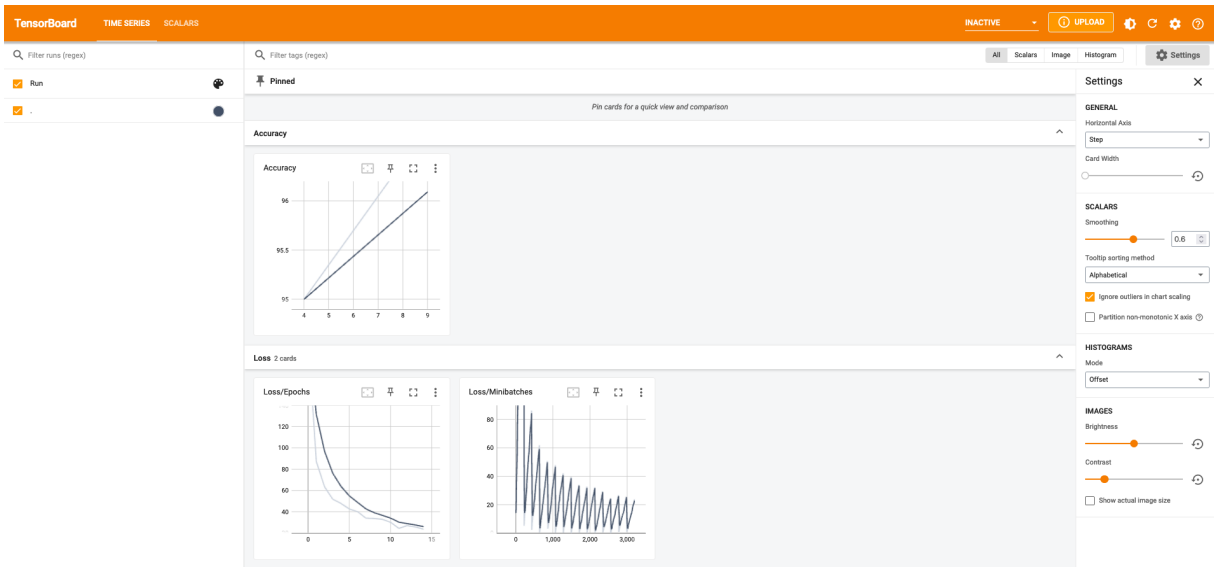



Figure 10: TensorBoard - Accuracy and Loss for run 6 on the GTX 1080 (Table 3)

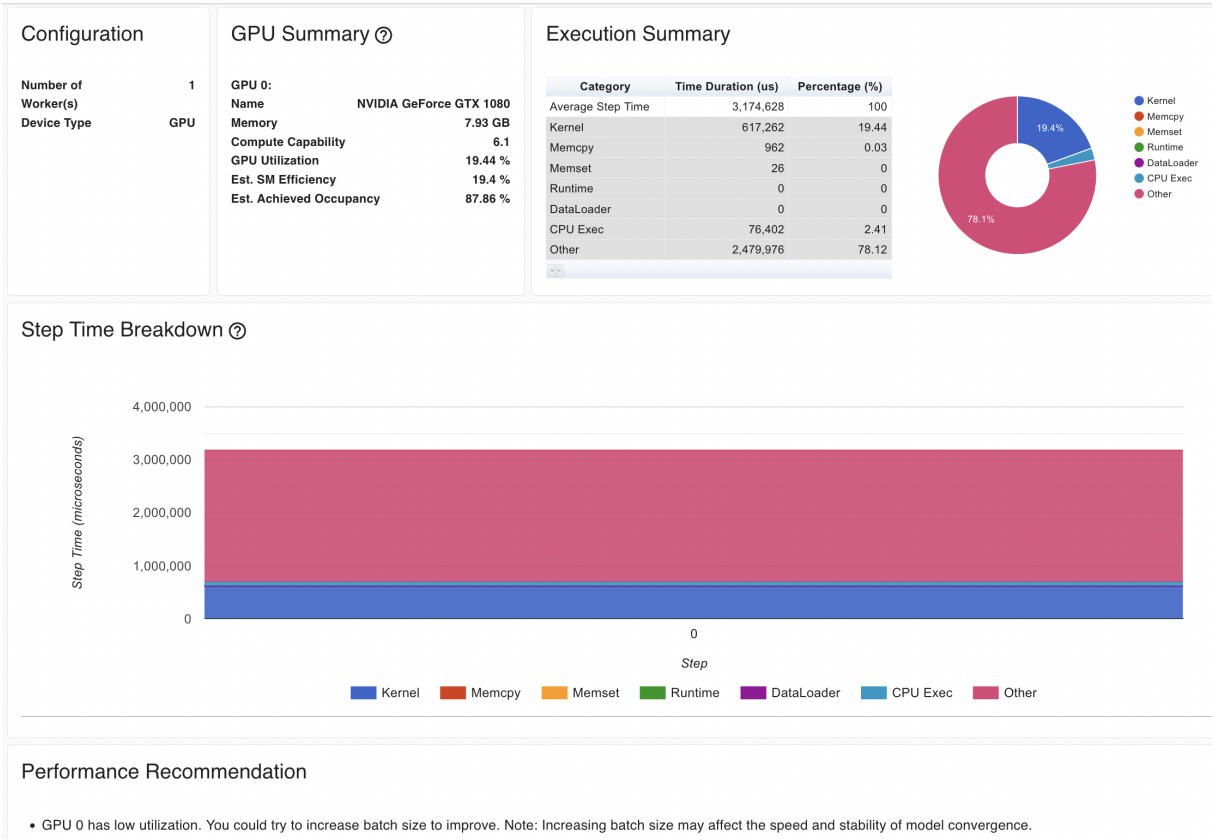



Figure 11: PyTorch - Profiler: Overview - Increased batch size (64)

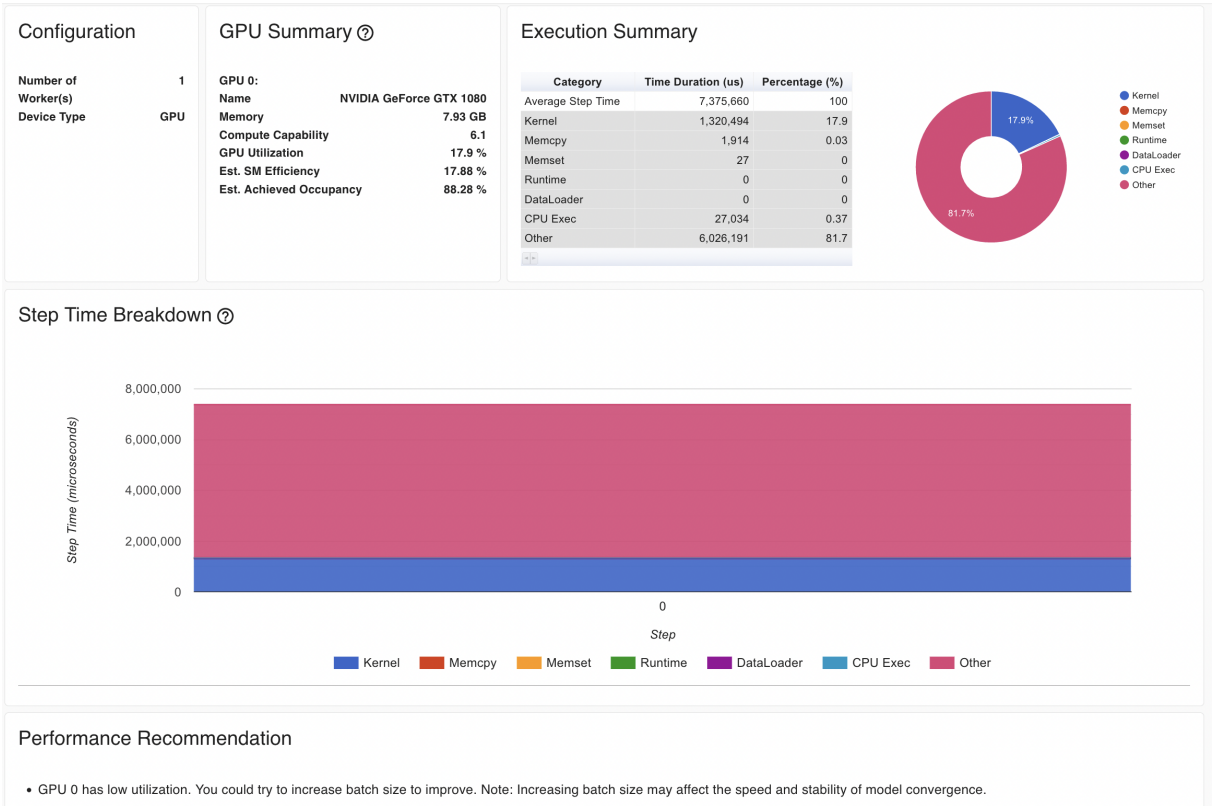



Figure 12: PyTorch - Profiler: Overview - Increased batch size (128)

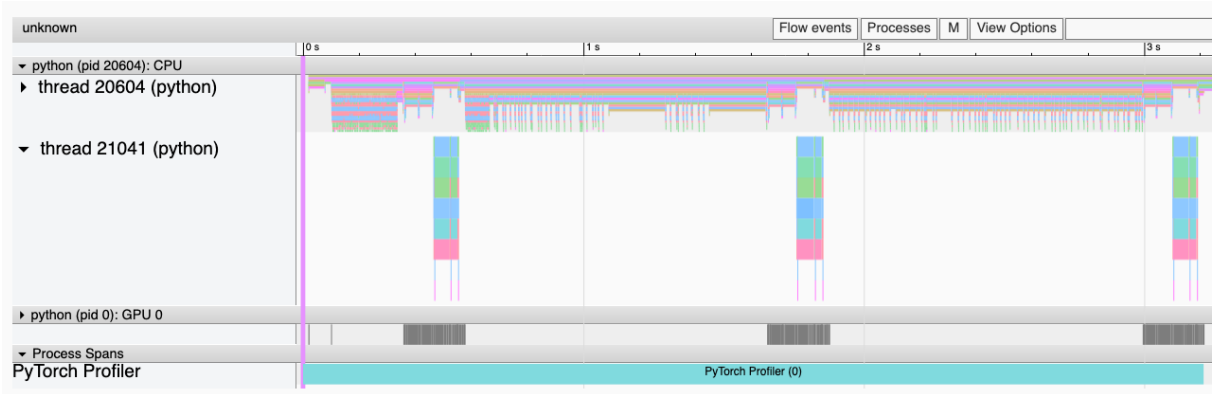
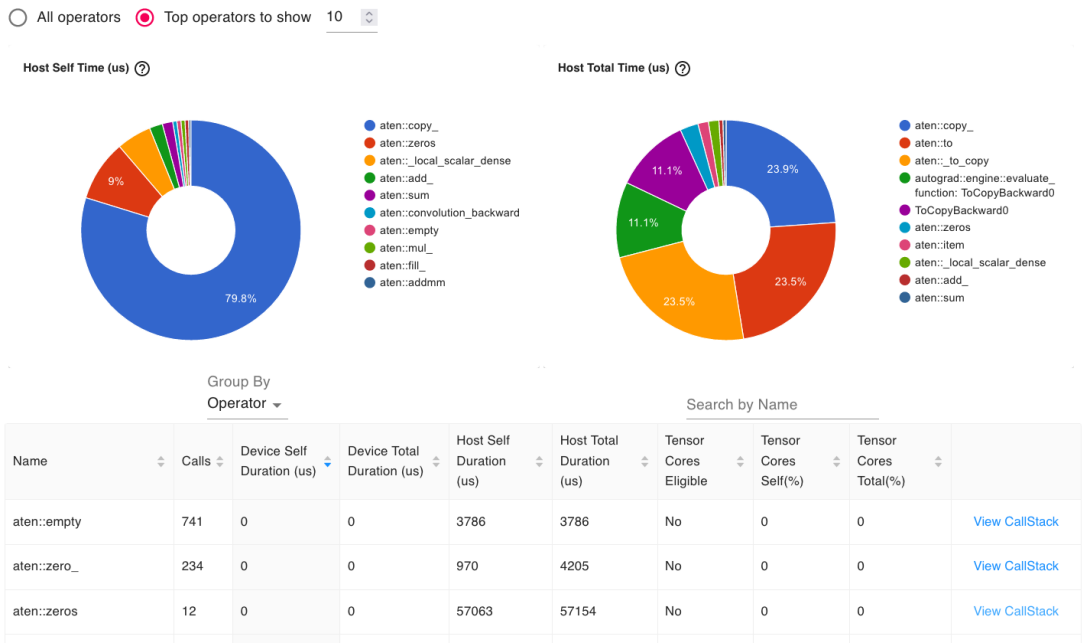
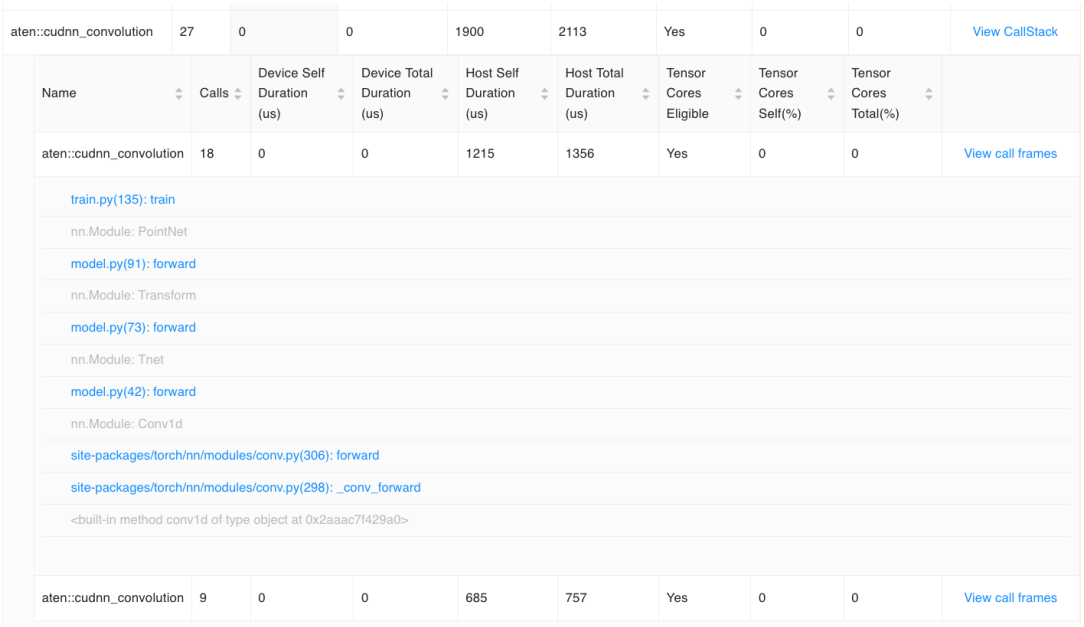



Figure 13: PyTorch - Profiler: Trace View




| Name | Calls | Device Self Duration (us) | Device Total Duration (us) | Host Self Duration (us) | Host Total Duration (us) | Tensor Cores Eligible | Tensor Cores Self(%) | Tensor Cores Total(%) | |
|---|---|---|---|---|---|---|---|---|---|
| aten::empty | 741 | 0 | 0 | 3786 | 3786 | No | 0 | 0 | View CallStack |
| aten::zero_ | 234 | 0 | 0 | 970 | 4205 | No | 0 | 0 | View CallStack |
| aten::zeros | 12 | 0 | 0 | 57063 | 57154 | No | 0 | 0 | View CallStack |

Figure 14: PyTorch - Profiler: Operator View



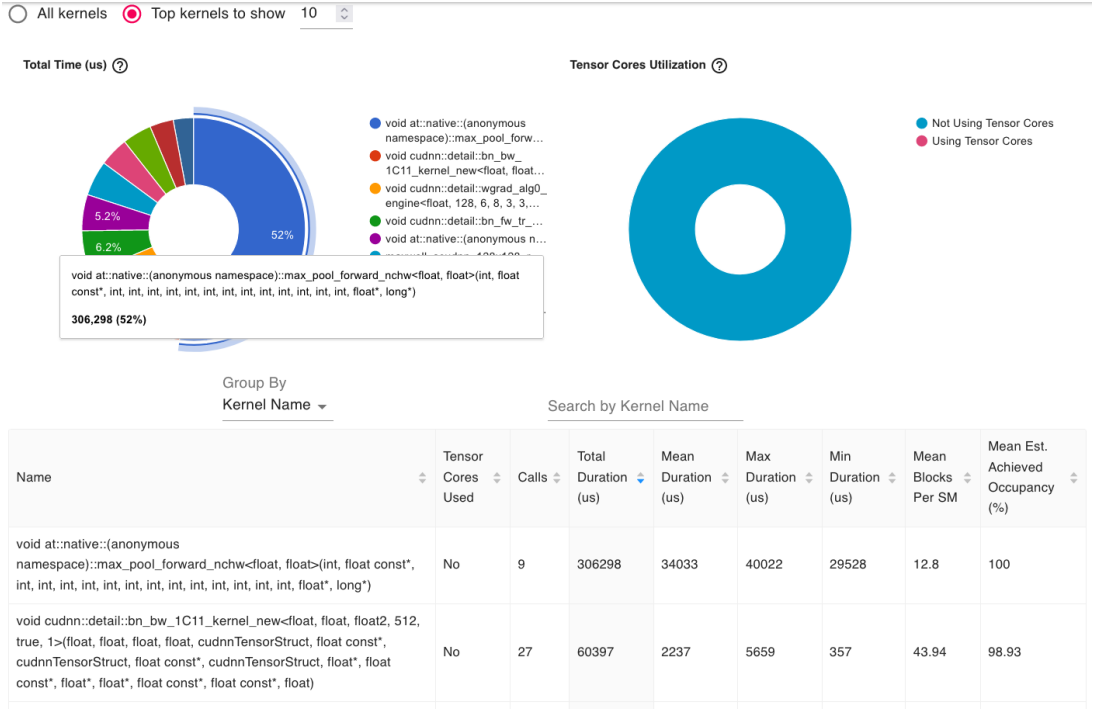
| aten::cudnn_convolution | 27 | 0 | 0 | 1900 | 2113 | Yes | 0 | 0 | View CallStack |
|---|---|---|---|---|---|---|---|---|---|

| Name | Calls | Device Self Duration (us) | Device Total Duration (us) | Host Self Duration (us) | Host Total Duration (us) | Tensor Cores Eligible | Tensor Cores Self(%) | Tensor Cores Total(%) | |
|---|---|---|---|---|---|---|---|---|---|
| aten::cudnn_convolution | 18 | 0 | 0 | 1215 | 1356 | Yes | 0 | 0 | View call frames |

train.py(135): train
nn.Module: PointNet
model.py(91): forward
nn.Module: Transform
model.py(73): forward
nn.Module: Tnet
model.py(42): forward
nn.Module: Conv1d
site-packages/torch/nn/modules/conv.py(306): forward
site-packages/torch/nn/modules/conv.py(298): _conv_forward
<built-in method conv1d of type object at 0x2aaac7f429a0>

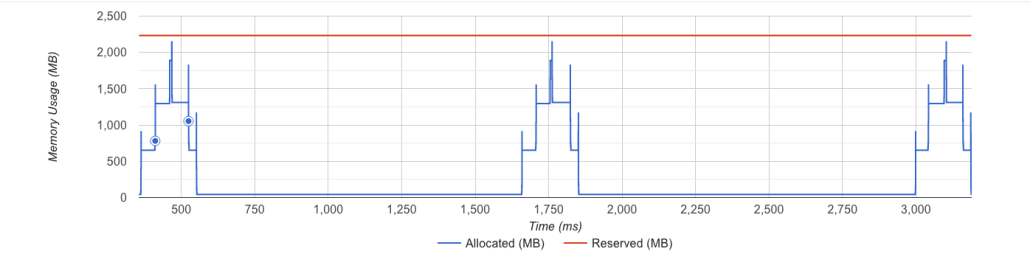
| aten::cudnn_convolution | 9 | 0 | 0 | 685 | 757 | Yes | 0 | 0 | View call frames |
|---|---|---|---|---|---|---|---|---|---|

Figure 15: PyTorch - Profiler: Operator View (CallStack)




| Name | Tensor Cores Used | Calls | Total Duration (us) | Mean Duration (us) | Max Duration (us) | Min Duration (us) | Mean Blocks Per SM | Mean Est. Achieved Occupancy (%) |
|---|---|---|---|---|---|---|---|---|
| void at::native::(anonymous namespace)::max_pool_forward_nchw<float, float>(int, float const*, int, int, int, int, int, int, int, int, int, int, int, int, float*, long*) | No | 9 | 306298 | 34033 | 40022 | 29528 | 12.8 | 100 |
| void cudnn::detail::bn_bw_1C11_kernel_new<float, float, float2, 512, true, 1>(float, float, float, float, cudnnTensorStruct, float const*, cudnnTensorStruct, float const*, cudnnTensorStruct, float*, float const*, float*, float*, float const*, float const*, float) | No | 27 | 60397 | 2237 | 5659 | 357 | 43.94 | 98.93 |

Figure 16: PyTorch - Profiler: GPU Kernel View




Search by Name

Min Size(KB) 65536

Max Size(KB) 262144

| Operator | Size (KB) | Allocation Time (ms) | Release Time (ms) | Duration (ms) |
|---|---|---|---|---|
| aten::empty_like (aten::empty) | 262144 | 362.91 | 363.04 | 0.13 |
| aten::empty_like (aten::empty) | 262144 | 411.38 | 411.5 | 0.13 |
| aten::empty_like (aten::empty) | 262144 | 460.62 | 467.57 | 6.94 |
| aten::max_pool2d_with_indices_backward | 262144 | 467.51 | 467.69 | 0.18 |
| aten::cudnn_convolution | 262144 | 460.36 | 467.69 | 7.33 |
| aten::cudnn_batch_norm_backward (aten::empty) | 262144 | 467.63 | 468.03 | 0.4 |
| aten::max_pool2d_with_indices_backward | 262144 | 524.27 | 524.4 | 0.13 |
| aten::clamp_min | 262144 | 411.47 | 524.4 | 112.93 |
| aten::threshold_backward | 262144 | 524.38 | 524.5 | 0.12 |
| aten::cudnn_convolution | 262144 | 411.14 | 524.5 | 113.36 |

Figure 17: PyTorch - Profiler: Memory View

Module View

| Module Name | Occurences | Operators | Host Total Time | Host Self Time | Device Total Time | Device Self Time |
|---|---|---|---|---|---|---|
| NLLLoss | 1 | 1 | 148 | 73 | 0 | 0 |
| NLLLoss | 1 | 1 | 101 | 45 | 0 | 0 |
| NLLLoss | 1 | 1 | 98 | 42 | 0 | 0 |
| + PointNet | 3 | 12 | 302303 | 420 | 0 | 0 |
| + PointNet | 3 | 12 | 302303 | 420 | 0 | 0 |
| + PointNet | 3 | 12 | 302303 | 420 | 0 | 0 |
| Linear | 3 | 3 | 356 | 66 | 0 | 0 |
| BatchNorm1d | 3 | 6 | 722 | 231 | 0 | 0 |
| Linear | 3 | 3 | 342 | 84 | 0 | 0 |
| Dropout | 3 | 3 | 385 | 147 | 0 | 0 |
| BatchNorm1d | 3 | 6 | 644 | 229 | 0 | 0 |
| Linear | 3 | 3 | 338 | 76 | 0 | 0 |
| LogSoftmax | 3 | 3 | 296 | 124 | 0 | 0 |
| + Transform | 3 | 24 | 298343 | 1827 | 0 | 0 |

Figure 18: PyTorch - Profiler: Module View

```
1  -------------------------- DeepSpeed Flops Profiler
   ↪  --------------------------
2  Profile Summary at step 5:
3  Notations:
4  data parallel size (dp_size), model parallel size(mp_size),
5  number of parameters (params), number of multiply-accumulate
   ↪  operations(MACs),
6  number of floating-point operations (flops), floating-point operations per
   ↪  second (FLOPS),
7  fwd latency (forward propagation latency), bwd latency (backward propagation
   ↪  latency),
```

```
step (weights update latency), iter latency (sum of fwd, bwd and step latency)

params per gpu:                                                 3.46 M
params of model = params per GPU * mp_size:                     3.46 M
fwd MACs per GPU:                                               14.07 GMACs
fwd flops per GPU:                                              29.04 G
fwd flops of model = fwd flops per GPU * mp_size:               29.04 G
fwd latency:                                                    83.19 ms
fwd FLOPS per GPU = fwd flops per GPU / fwd latency:            349.1 GFLOPS

----------------------------- Aggregated Profile per GPU -----------------------------
Top 1 modules in terms of params, MACs or fwd latency at different model depths:
depth 0:
    params      - {'PointNet': '3.46 M'}
    MACs        - {'PointNet': '14.07 GMACs'}
    fwd latency - {'PointNet': '83.19 ms'}
depth 1:
    params      - {'Transform': '2.8 M'}
    MACs        - {'Transform': '14.05 GMACs'}
    fwd latency - {'Transform': '82.08 ms'}
depth 2:
    params      - {'Tnet': '2.66 M'}
    MACs        - {'Tnet': '9.34 GMACs'}
    fwd latency - {'Tnet': '58.43 ms'}

------------------------------ Detailed Profile per GPU ------------------------------
Each module profile is listed after its name in the following order:
params, percentage of total params, MACs, percentage of total MACs, fwd latency, percentage of total fwd latency, fwd FLOPS

Note: 1. A module can have torch.nn.module or torch.nn.functional to compute logits (e.g. CrossEntropyLoss). They are not counted as submodules, thus not to be printed out. However they make up the difference between a parent's MACs (or latency) and the sum of its submodules'.
2. Number of floating-point operations is a theoretical estimation, thus FLOPS computed using that could be larger than the maximum system throughput.
3. The fwd latency listed in the top module's profile is directly captured at the module forward function in PyTorch, thus it's less than the fwd latency shown above which is captured in DeepSpeed.

PointNet(
  3.46 M, 100.00% Params, 14.07 GMACs, 100.00% MACs, 83.19 ms, 100.00% latency, 349.1 GFLOPS,
  (transform): Transform(
    2.8 M, 80.94% Params, 14.05 GMACs, 99.85% MACs, 82.08 ms, 98.67% latency, 353.28 GFLOPS,
```

```
45      (input_transform): Tnet(
46        803.08 k, 23.19% Params, 4.59 GMACs, 32.63% MACs, 27.06 ms, 32.53%
          ↪ latency, 350.87 GFLOPS,
47        (conv1): Conv1d(256, 0.01% Params, 6.29 MMACs, 0.04% MACs, 313.28 us,
          ↪ 0.38% latency, 46.86 GFLOPS, 3, 64, kernel_size=(1,), stride=(1,))
48        (conv2): Conv1d(8.32 k, 0.24% Params, 268.44 MMACs, 1.91% MACs, 374.32
          ↪ us, 0.45% latency, 1.45 TFLOPS, 64, 128, kernel_size=(1,),
          ↪ stride=(1,))
49        (conv3): Conv1d(132.1 k, 3.81% Params, 4.29 GMACs, 30.53% MACs, 2.78
          ↪ ms, 3.34% latency, 3.1 TFLOPS, 128, 1024, kernel_size=(1,),
          ↪ stride=(1,))
50        (fc1): Linear(524.8 k, 15.15% Params, 16.78 MMACs, 0.12% MACs, 124.45
          ↪ us, 0.15% latency, 269.61 GFLOPS, in_features=1024,
          ↪ out_features=512, bias=True)
51        (fc2): Linear(131.33 k, 3.79% Params, 4.19 MMACs, 0.03% MACs, 92.27 us,
          ↪ 0.11% latency, 90.92 GFLOPS, in_features=512, out_features=256,
          ↪ bias=True)
52        (fc3): Linear(2.31 k, 0.07% Params, 73.73 KMACs, 0.00% MACs, 94.18 us,
          ↪ 0.11% latency, 1.57 GFLOPS, in_features=256, out_features=9,
          ↪ bias=True)
53        (bn1): BatchNorm1d(128, 0.00% Params, 0 MACs, 0.00% MACs, 238.42 us,
          ↪ 0.29% latency, 43.98 GFLOPS, 64, eps=1e-05, momentum=0.1,
          ↪ affine=True, track_running_stats=True)
54        (bn2): BatchNorm1d(256, 0.01% Params, 0 MACs, 0.00% MACs, 325.2 us,
          ↪ 0.39% latency, 64.49 GFLOPS, 128, eps=1e-05, momentum=0.1,
          ↪ affine=True, track_running_stats=True)
55        (bn3): BatchNorm1d(2.05 k, 0.06% Params, 0 MACs, 0.00% MACs, 1.81 ms,
          ↪ 2.17% latency, 92.85 GFLOPS, 1024, eps=1e-05, momentum=0.1,
          ↪ affine=True, track_running_stats=True)
56        (bn4): BatchNorm1d(1.02 k, 0.03% Params, 0 MACs, 0.00% MACs, 153.3 us,
          ↪ 0.18% latency, 534.37 MFLOPS, 512, eps=1e-05, momentum=0.1,
          ↪ affine=True, track_running_stats=True)
57        (bn5): BatchNorm1d(512, 0.01% Params, 0 MACs, 0.00% MACs, 130.41 us,
          ↪ 0.16% latency, 314.07 MFLOPS, 256, eps=1e-05, momentum=0.1,
          ↪ affine=True, track_running_stats=True)
58      )
59      (feature_transform): Tnet(
60        1.86 M, 53.62% Params, 4.75 GMACs, 33.78% MACs, 31.37 ms, 37.71%
          ↪ latency, 312.94 GFLOPS,
61        (conv1): Conv1d(4.16 k, 0.12% Params, 134.22 MMACs, 0.95% MACs, 247.48
          ↪ us, 0.30% latency, 1.09 TFLOPS, 64, 64, kernel_size=(1,),
          ↪ stride=(1,))
62        (conv2): Conv1d(8.32 k, 0.24% Params, 268.44 MMACs, 1.91% MACs, 371.46
          ↪ us, 0.45% latency, 1.46 TFLOPS, 64, 128, kernel_size=(1,),
          ↪ stride=(1,))
63        (conv3): Conv1d(132.1 k, 3.81% Params, 4.29 GMACs, 30.53% MACs, 2.77
          ↪ ms, 3.33% latency, 3.12 TFLOPS, 128, 1024, kernel_size=(1,),
          ↪ stride=(1,))
64        (fc1): Linear(524.8 k, 15.15% Params, 16.78 MMACs, 0.12% MACs, 115.87
          ↪ us, 0.14% latency, 289.58 GFLOPS, in_features=1024,
          ↪ out_features=512, bias=True)
```

```
65          (fc2): Linear(131.33 k, 3.79% Params, 4.19 MMACs, 0.03% MACs, 92.51 us,
            ↪  0.11% latency, 90.68 GFLOPS, in_features=512, out_features=256,
            ↪  bias=True)
66          (fc3): Linear(1.05 M, 30.39% Params, 33.55 MMACs, 0.24% MACs, 118.97
            ↪  us, 0.14% latency, 564.08 GFLOPS, in_features=256,
            ↪  out_features=4096, bias=True)
67          (bn1): BatchNorm1d(128, 0.00% Params, 0 MACs, 0.00% MACs, 216.01 us,
            ↪  0.26% latency, 48.54 GFLOPS, 64, eps=1e-05, momentum=0.1,
            ↪  affine=True, track_running_stats=True)
68          (bn2): BatchNorm1d(256, 0.01% Params, 0 MACs, 0.00% MACs, 312.09 us,
            ↪  0.38% latency, 67.2 GFLOPS, 128, eps=1e-05, momentum=0.1,
            ↪  affine=True, track_running_stats=True)
69          (bn3): BatchNorm1d(2.05 k, 0.06% Params, 0 MACs, 0.00% MACs, 1.82 ms,
            ↪  2.18% latency, 92.43 GFLOPS, 1024, eps=1e-05, momentum=0.1,
            ↪  affine=True, track_running_stats=True)
70          (bn4): BatchNorm1d(1.02 k, 0.03% Params, 0 MACs, 0.00% MACs, 144.0 us,
            ↪  0.17% latency, 568.87 MFLOPS, 512, eps=1e-05, momentum=0.1,
            ↪  affine=True, track_running_stats=True)
71          (bn5): BatchNorm1d(512, 0.01% Params, 0 MACs, 0.00% MACs, 133.51 us,
            ↪  0.16% latency, 306.78 MFLOPS, 256, eps=1e-05, momentum=0.1,
            ↪  affine=True, track_running_stats=True)
72        )
73        (conv1): Conv1d(256, 0.01% Params, 6.29 MMACs, 0.04% MACs, 231.27 us,
          ↪  0.28% latency, 63.48 GFLOPS, 3, 64, kernel_size=(1,), stride=(1,))
74        (conv2): Conv1d(8.32 k, 0.24% Params, 268.44 MMACs, 1.91% MACs, 452.52
          ↪  us, 0.54% latency, 1.2 TFLOPS, 64, 128, kernel_size=(1,),
          ↪  stride=(1,))
75        (conv3): Conv1d(132.1 k, 3.81% Params, 4.29 GMACs, 30.53% MACs, 2.77 ms,
          ↪  3.34% latency, 3.11 TFLOPS, 128, 1024, kernel_size=(1,), stride=(1,))
76        (bn1): BatchNorm1d(128, 0.00% Params, 0 MACs, 0.00% MACs, 208.14 us,
          ↪  0.25% latency, 50.38 GFLOPS, 64, eps=1e-05, momentum=0.1,
          ↪  affine=True, track_running_stats=True)
77        (bn2): BatchNorm1d(256, 0.01% Params, 0 MACs, 0.00% MACs, 319.24 us,
          ↪  0.38% latency, 65.69 GFLOPS, 128, eps=1e-05, momentum=0.1,
          ↪  affine=True, track_running_stats=True)
78        (bn3): BatchNorm1d(2.05 k, 0.06% Params, 0 MACs, 0.00% MACs, 1.82 ms,
          ↪  2.18% latency, 92.32 GFLOPS, 1024, eps=1e-05, momentum=0.1,
          ↪  affine=True, track_running_stats=True)
79      )
80      (fc1): Linear(524.8 k, 15.15% Params, 16.78 MMACs, 0.12% MACs, 111.82 us,
        ↪  0.13% latency, 300.08 GFLOPS, in_features=1024, out_features=512,
        ↪  bias=True)
81      (fc2): Linear(131.33 k, 3.79% Params, 4.19 MMACs, 0.03% MACs, 93.7 us,
        ↪  0.11% latency, 89.53 GFLOPS, in_features=512, out_features=256,
        ↪  bias=True)
82      (fc3): Linear(2.57 k, 0.07% Params, 81.92 KMACs, 0.00% MACs, 89.88 us,
        ↪  0.11% latency, 1.82 GFLOPS, in_features=256, out_features=10,
        ↪  bias=True)
83      (bn1): BatchNorm1d(1.02 k, 0.03% Params, 0 MACs, 0.00% MACs, 145.2 us,
        ↪  0.17% latency, 564.2 MFLOPS, 512, eps=1e-05, momentum=0.1, affine=True,
        ↪  track_running_stats=True)
```

```
    (bn2): BatchNorm1d(512, 0.01% Params, 0 MACs, 0.00% MACs, 133.28 us, 0.16%
    ↪ latency, 307.33 MFLOPS, 256, eps=1e-05, momentum=0.1, affine=True,
    ↪ track_running_stats=True)
    (dropout): Dropout(0, 0.00% Params, 0 MACs, 0.00% MACs, 87.02 us, 0.10%
    ↪ latency, 0.0 FLOPS, p=0.3, inplace=False)
    (logsoftmax): LogSoftmax(0, 0.00% Params, 0 MACs, 0.00% MACs, 66.04 us,
    ↪ 0.08% latency, 0.0 FLOPS, dim=1)
)
------------------------------------------------------------------------------
```

Listing 5: DeepSpeed - FlopProfiler: Full output

# A Code samples

```
sacct --format=jobid,elapsed -P -j 14657599, 14617521, 14615343,
↪ 14650076, 14615344, 14617172, 14650079, 14617171, 14618941,
↪ 14617203, 14619619, 14650080, 14619618,14619617, 14617202, 14650750,
↪ 14650758, 14629421, 14629426, 14650740, 14650759 -X
```

Listing 6: The command that is used to collect accounting information for the jobs executed on the SCC for this project (Table 3). The output of this command can be found in Listing 4.

```
from torch.utils.tensorboard import SummaryWriter
[...]
loss_idx_value = 0
for epoch in range(num_training_epochs):
    model.train(); running_loss = 0.0
    for i, batch in enumerate(train_loader):
        [...] # Load data, classify data, calculate loss
        loss.backward(); optimizer.step()
        running_loss += loss.item()
        writer.add_scalar("Loss/Minibatches", running_loss, loss_idx_value)
        loss_idx_value += 1
    writer.add_scalar("Loss/Epochs", running_loss, epoch)
    model.eval()
    if epoch % 5 == 4: # get validation accuracy every 5 epochs
        [...] # calculate accuracy
        writer.add_scalar("Accuracy", accuracy, epoch)
```

Listing 7: Highlighted parts of the code are required to use Tensorboard for profiling the training process of the neural network. The implementation can be found in the accompanying GitHub repository: `https://github.com/haukekirchner/scap/blob/main/code/pointnet/train.py`

```
[...]
for epoch in range(num_training_epochs):
    [...]
    prof = torch.profiler.profile(
            schedule=torch.profiler.schedule(wait=1, warmup=1, active=3,
            ↪  repeat=2),
            on_trace_ready=torch.profiler.tensorboard_trace_handler(logdir +
            ↪  "/profiler"),
            record_shapes=True,
            profile_memory=True,
            with_stack=True)
    prof.start()
    for i, batch in enumerate(train_loader):
        [...] # Load data, classify data, calculate loss
        loss.backward(); optimizer.step()
        prof.step()
    prof.stop()
```

Listing 8: Highlighted parts of the code are required to use the PyTorch Profiler for profiling the training process of the neural network. The implementation can be found in the accompanying GitHub repository: `https://github.com/haukekirchner/scap/blob/main/code/pointnet/train.py`

```
from deepspeed.profiling.flops_profiler import FlopsProfiler
[...]
flop_prof = FlopsProfiler(model)
profile_step = 5; print_profile= True
for epoch in range(num_training_epochs):
    [...]
    for i, batch in enumerate(train_loader):
            if i == profile_step:
                flop_prof.start_profile()
            [...] # Load data, classify data, calculate loss
            if i == profile_step: # end profiling and print output
                flop_prof.stop_profile()
                flops = flop_prof.get_total_flops()
                macs = flop_prof.get_total_macs()
                params = flop_prof.get_total_params()
                if print_profile:
                    flop_prof.print_model_profile(profile_step=profile_step)
                flop_prof.end_profile()
            loss.backward(); optimizer.step()
```

.

Listing 9: Highlighted parts of the code are required to use Deepspeed - FLOPSProfiler for profiling the training process of the neural network. The implementation can be found in the accompanying GitHub repository: `https://github.com/haukekirchner/scap/blob/main/code/pointnet/train.py`