MULTI-GPU TRAINING AND PARALLEL CPU COMPUTING FOR THE MACHINE LEARNING EXPERIMENTS USING ARIADNE LIBRARY

P. Goncharov[1], A. Nikolskaia[2], G. Ososkov[1], E. Rezvaya[1], D. Rusov[1] and E. Shchavelev[2,a]

[1] *Joint Institute for Nuclear Research, 6 Joliot-Curie street, 141980, Dubna, Moscow region, Russia*

[2] *Saint Petersburg State University, 7-9 Universitetskaya emb., Saint Petersburg, 199034, Russia*

E-mail: [a] egor.schavelev@gmail.com

Modern machine learning (ML) tasks and neural network (NN) architectures require huge amounts of GPU computational facilities and demand high CPU parallelization for data preprocessing. At the same time, the Ariadne library, which aims to solve complex high-energy physics tracking tasks with the help of deep neural networks, lacks multi-GPU training and efficient parallel data preprocessing on the CPU.

In our work, we present our approach for the Multi-GPU training in the Ariadne library. We will present efficient data-caching, parallel CPU data preprocessing, generic ML experiment setup for prototyping, training, and inference deep neural network models. Results in terms of speed-up and performance for the existing neural network approaches are presented with the help of GOVORUN computing resources.

Keywords: Machine Learning, Tracking, Python library, CPU optimizations, GPU optimizations

Pavel Goncharov, Anastasiia Nikolskaia, Gennady Ososkov,
Ekaterina Rezvaya, Daniil Rusov, Egor Shchavelev

1. Motivation

Modern high-energy physics (HEP) experiments produce large amounts of data and require specific computer software to operate. Particle tracking is an important part of software of HEP experiments and there are many algorithms for performing such tasks and one of the most well-proven tracking approach is based on Kalman filter. Unfortunately, it does not scale sufficiently to perform efficient computations on modern hardware such as graphics processing units (GPU). At the same time, studies [1,2] indicate that machine learning (ML) and deep neural networks (NN) can be an efficient replacement for the well-known tracking algorithms. Their authors achieve competitive results in terms of track reconstruction accuracy, and they are orders of magnitude faster in terms of processing speed. Modern ML approaches are mostly developed in the Python programming language and use specific tensor-based libraries to implement NN models and deploy them to the GPU. Considering the novelty of the ML tracking there are no generally known Python library which goal is to study deep learning in HEP tracking tasks. Considering all the above mentioned we decided to start the development of the Ariadne [3] library – the first Python open-source library for particle tracking based on deep learning methods. The goal of Ariadne is to help researchers investigate their ML-based tracking methods with a simple but standardized setup. Ariadne is still in development but has already provided great benefits for our tasks. The initial Ariadne description and motivation one can find in [3].

2. Current state of Ariadne

Current Ariadne application programming interface (API) from the researcher point of view is shown in Figure 1.

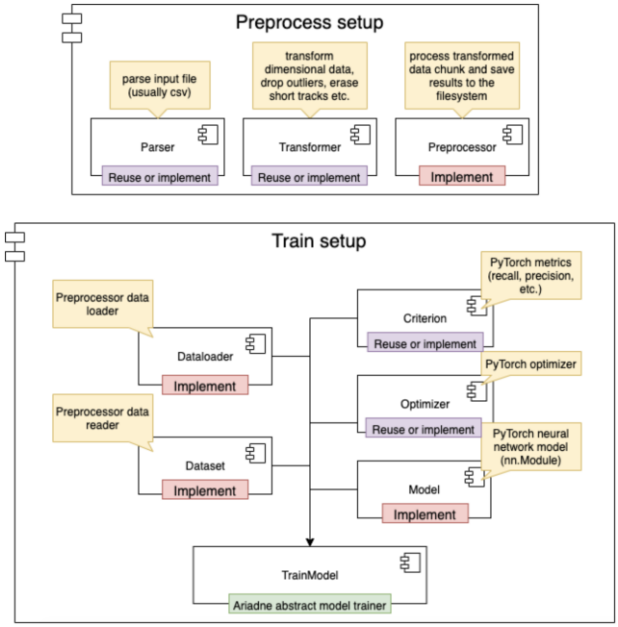

Figure 1. Ariadne API

For an experimental run, researcher implements such following components, as preprocessor, model, and dataset, while then he can override already implemented components such as parse, transforms, criterion and optimizer.

After the implementation, the user should run the ‘prepare’ phase which computes needed preprocessing steps. Initial data processing steps are shown in Figure 2a. Later, one can train his NN model with the preprocessed data.

3. Caching and Multi-CPU prepare

After the previous work [3] there were already 5 different NN approaches developed with the help of Ariadne. Every approach shares the common library API but implements its own preprocessing and training components. During the implementation and investigation of a potential approach researchers often run parsing, preprocessing, and training phases sequentially one-by-one in a single Python process. So, for example, after any change in preprocessing algorithms all training data (which can occupy hundreds of gigabytes of disk space) must be recomputed from scratch. Running such scripts as a single-process Python is a huge time-consumer and cannot scale well with a hardware computing facility. In this work we reimplemented 'prepare' core scripts with the help of multi-processing. The comparison of old and new implementations is shown in Figure 2. Implementation consists of 3 main parts:

- Caching module – realtime memorization of any processing unit (such as parsing, coordinate transformations, and any other data mutation procedure)
- Multiprocessing of target preprocessing routine (a preprocessor is being run in worker pool in parallel with the help of Python multiprocessing framework)
- Data serialization – with the help of HDF5 format [4] data could be efficiently read & write to the disk.

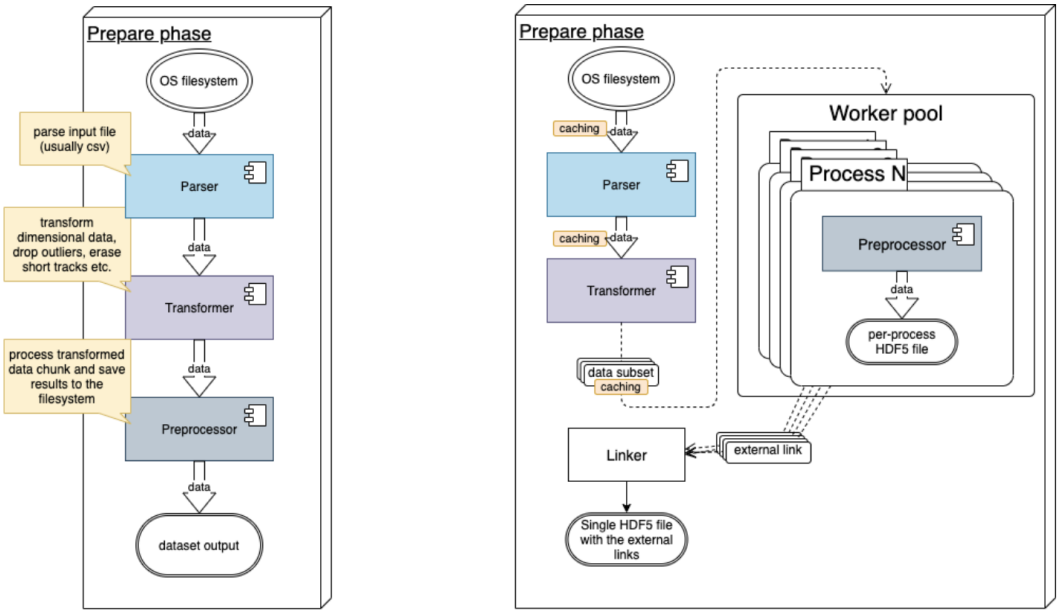

Figure 2. Comparison of the old (a) and new (b) 'prepare' core implementation. For the new implementation, worker pool creates as many parallel processing processes as a count of cores on the target hardware.

4. Batch bucketing and Multi-GPU training

With the help of a new caching module, we implemented the batch bucketing routine. Batch bucketing routine is a common algorithm[5] for effective dataset data processing which allows placing the NN input with the equal dimensions to the same training batch. Such routine can reasonably speed-up training time on a single GPU device and allow to use the batch sizes which would not fit in GPU memory without such approach. We also enabled the Multi-GPU training with the help of PyTorch Lightning [6] library. Now researchers can run their NN training on up to 8 GPUs in parallel which also greatly reduces model training times.

5. Measured performance impact

After applying new functionalities described above, we measured the typical researcher workflows on 2 target hardware:

- Laptop (MacBook Pro 13) // Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz (8 cores)
- Hybrilit (JINR HOVORUN) // Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz (80 cores)

In Table 1, one can observe more than x25 event processing speed-up compared to the initial implementation and up to 6 times faster 'prepare' phase. In Figure 3 one can observe a great NN training speed improvement for the f1_score metric for the revised implementation compared to the original. (for the same 1-hour training on the same data, the same GraphNet model converges much more rapidly with the multi-GPU or batch bucketing training).

Table 1. Processing speed and time preparation for the initial and revised implementation.

Machine	**Processing speed, events per second**		**Full dataset (250k events) preparation, minutes**	
	Old	New	Old	New
MacBook	~17	~120 (**x7 speed-up**)	n/a	n/a
Hybrilit	~26	~630 (**x25 speed-up**)	396 minutes	62 minutes (**x6 speed-up**)

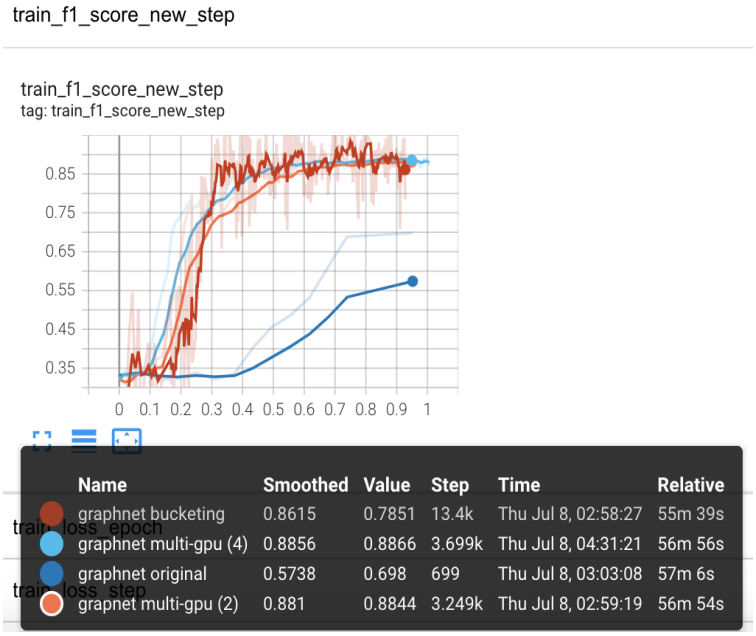

Figure 3. F1 score metrics for the original (*graphnet original*) implementation and revised implementation (*graphnet bucketing* – training with the batch bucketing routine, *graphnet multi-gpu (2)* – training on the 2 GPU units in parallel, *graphnet multi-gpu (4)* – training on the 4 GPU units in parallel).

6. Conclusion

In our work, we successfully implemented a new 'prepare' module for Ariadne. The module now can run in parallel utilizing all CPU cores on the target hardware which led up to 25x faster event processing for the GraphNet NN model. For the 'training' module we enabled the multi-GPU training and batch bucketing algorithm which greatly reduces training time for the existing NN model

implementation. Such results show great potential for the future implementations of the other NN approaches within the Ariadne library – users can now utilize more hardware resources, therefore, increasing processing capacity for more complex neural network models and preprocessing routines. Source code is available at [7].

7. Acknowledgment

The reported study was funded by RFBR according to the research project № 18-02-40101.

The calculations were carried out on the basis of the HybriLIT heterogeneous computing platform (LIT, JINR) [8].

References

[1] Goncharov P., Shchavelev E., Ososkov G. and Baranov D. BM@N Tracking with Novel Deep Learning Methods // EPJ Web Conf., 226 (2020) 03009 /DOI: https://doi.org/10.1051/epjconf/202022603009

[2] Farrell S. et al., "Novel deep learning methods for track reconstruction," in 4th International Workshop Connecting The Dots 2018 (CTD2018) Seattle, Washington, USA, March 20-22, 2018 (2018), arXiv:1810.06111 [hep-ex].

[3] Goncharov P. et al. Ariadne: PyTorch library for particle track reconstruction using deep learning / P. Goncharov, E. Schavelev, A. Nikolskaya, and G. Ososkov //AIP Conference Proceedings. – AIP Publishing LLC, 2021. – Vol. 2377. – No. 1. – pp. 040004.

[4] The HDF Group. Hierarchical Data Format, version 5, 1997-NNNN. https://www.hdfgroup.org/HDF5/.

[5] Khomenko V., Shyshkov O., Radyvonenko O., and Bokhan K. (2016). Accelerating recurrent neural network training using sequence bucketing and multi-GPU data parallelization. 10.1109/DSMP.2016.7583516.

[6] Falcon, W., & The PyTorch Lightning team. (2019). PyTorch Lightning (Version 1.4) [Computer software]. https://doi.org/10.5281/zenodo.3828935

[7] Ariadne, Github: https://github.com/t3hseus/ariadne

[8] G. Adam et al., CEUR Workshop Proc., Vol. 2267, 638-644 (2018)