Accelerating End-to-End Deep Learning for Particle Reconstruction using CMS open data

Michael Andrews[1], *Bjorn* Burkle[2], *Shravan* Chaudhari[3], *Davide* Di Croce[4,*], *Sergei* Gleyzer[4], *Ulrich* Heintz[2], *Meenakshi* Narain[2], *Manfred* Paulini[1], and *Emanuele* Usai[2]

[1]Department of Physics, Carnegie Mellon University, Pittsburgh, USA
[2]Department of Physics, Brown University, Providence, USA
[3]Department of Electrical and Electronics Engineering, BITS Pilani, Goa, India
[4]Department of Physics and Astronomy, University of Alabama, Tuscaloosa, USA

Abstract. Machine learning algorithms are gaining ground in high energy physics for applications in particle and event identification, physics analysis, detector reconstruction, simulation and trigger. Currently, most data-analysis tasks at LHC experiments benefit from the use of machine learning. Incorporating these computational tools in the experimental framework presents new challenges. This paper reports on the implementation of the end-to-end deep learning with the CMS software framework and the scaling of the end-to-end deep learning with multiple GPUs. The end-to-end deep learning technique combines deep learning algorithms and low-level detector representation for particle and event identification. We demonstrate the end-to-end implementation on a top quark benchmark and perform studies with various hardware architectures including single and multiple GPUs and Google TPU.

1 Introduction

One of the major challenges for high-energy physicists is to manage the large amounts of data generated by the Large Hadron Collider (LHC). With the advent of High Luminosity LHC (HL-LHC), the filtering, data processing and reconstruction of collision events will become even more challenging. To attain the main physics goals of the HL-LHC, the use of advanced machine learning techniques will be necessary to overcome significant challenges posed by increased levels of pile-up and the rarity of sought-after signals. To address this issue, researchers are applying state-of-the-art machine learning algorithms for data processing and detector reconstruction, aimed at optimising their performance and accelerating these models during training and inference.

Most machine learning-based particle identification techniques currently developed by ATLAS and CMS experiments rely on inputs provided by the Particle Flow (PF) algorithm used to convert detector level information to physics objects [1]. Despite the very high reconstruction efficiency of PF algorithms, some physics objects may fail to be reconstructed, are reconstructed imperfectly, or exist as fakes [2]. For that reason it is advantageous to consider reconstruction that allows a direct application of machine learning algorithms to low-level

[*]e-mail: davide.di.croce@cern.ch

data in the detector. The end-to-end deep learning technique [3, 4] combines deep learning algorithms and low-level detector representation of collision events.

The integration of such innovative machine learning algorithms with the data processing pipeline of experimental software frameworks is an important goal for LHC experiments [5] and forms one of the major research and development goals for CMS. Training deep neural networks is extremely time consuming and demands significant computational resources. Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) have proved to be efficient for computationally expensive deep learning tasks without significant drops in accuracy and precision. However, for large datasets and deep networks, even a GPU might fall short of the speed required by the model to achieve convergence in the desired time. Heterogeneous computing is a time tested approach which involves scaling large datasets and their processing to multiple computational resources (CPUs, GPUs) resulting in remarkable boost in the performance and energy efficiency. In this paper, we demonstrate the integration of the end-to-end deep learning framework into CMS Software Framework (CMSSW) on a simulated top quark benchmark [6] and compare the performance of realistic deep learning algorithms on single and multiple GPUs and TPUs.

2 Open Data Simulated Samples

The end-to-end deep learning technique relies on high-fidelity detector data, which in this work comes from the simulated Monte Carlo in the CMS Open Data Portal [7]. We use a sample of SM $t\bar{t}$ production where the W boson from the top quark decay is required to decay to quarks as a source of boosted top quarks. Light flavour- and gluon-jets were obtained from three samples of QCD dijet production in different ranges of the hard-scatter transverse momentum. The datasets used for this study can be found in [8–11].

CMS is a multi-purpose detector composed of several cylindrical subdetector layers, with both barrel and endcap sections, encasing a primary interaction point. It features a large $B = 3.8$ T solenoid magnet to bend the trajectories of charged particles that aid in p_{T} measurement [2]. At the innermost layers, close to the beamline, there is a silicon tracker used to reconstruct the trajectory of charged particles and find their interaction vertices. The tracker can be divided in two parts the silicon pixel detector and silicon strip detector [2]. The first silicon pixel detector is the inner most part and composed of three layers in the barrel region (BPIX) and three disks in the endcap region (FPIX). The pixel detector provides crucial information for vertexing and track seeding. The outer part of the tracking system is composed of silicon strip detectors. These provide a precise position in the ϕ coordinate, but not in the η coordinate. This is followed by the electromagnetic calorimeter (ECAL), made of lead-tungstate crystals, to measure the energy of electromagnetically interacting particles, then the hardonic calorimeter (HCAL), made of brass towers, to measure the energy of hadrons [2]. These are surrounded by the solenoid magnet which is finally encased by the muon chambers to detect the passage of muons [2].

For all samples, the detector response is simulated using Geant4 with the full CMS geometry and is processed through the CMS PF reconstruction algorithm using CMSSW release 5_3_32 [12]. An average of ten additional background collisions or pileup (PU) interactions are added to the simulated hard-scatter event, which are sampled from a realistic distribution of simulated minimum bias events. For this study, we additionally use a custom CMS data format which includes the low-level tracker detector information, specifically, the reconstructed clusters from the pixel and silicon strip detectors [13].

For jet selection, we take reconstructed jets clustered using the anti-k$_t$ algorithm [14] with a radius parameter R of 0.8, and require $p_{\mathrm{T}} > 400\,\mathrm{GeV}$ and $|\eta| < 1.37$ for our event selection. For top jets we also require the generator-level top quark, its bottom quark and

Category	Top quark jets	QCD jets	Total Jets
Train	1280830	1279170	2560000
Validation	47859	48141	96000
Test	319819	320181	640000

Table 1. Number of jets used for training, validation, and testing in all categories

W-boson daughters, and W-boson daughters to be within an angular separation of $\Delta R = \sqrt{\Delta\eta^2 + \Delta\phi^2} < 0.8$ from the reconstructed AK8 jet axis, where ϕ is the azimuthal angle of the CMS detector. The total number of jets used in the training, validation, and testing of our networks can be found in Table 1.

We construct the jet images using low-level detector information where each subdetector is projected onto an image layer, or several layers in the case of the tracker, in a grid of 125 x 125 pixels with the image centered around the most energetic HCAL deposit of the jet. Each pixel corresponds to the span of an ECAL barrel crystal which covers a 0.0174×0.0174 in the $\eta-\phi$ plane, giving our images an effective ΔR of 2.175. For the ECAL and HCAL images, each crystal or tower is directly mapped to one or more image pixels containing the energy deposited in that crystal or tower, as described in [4]. Reconstructed particle tracks are used to construct three separate image layers.

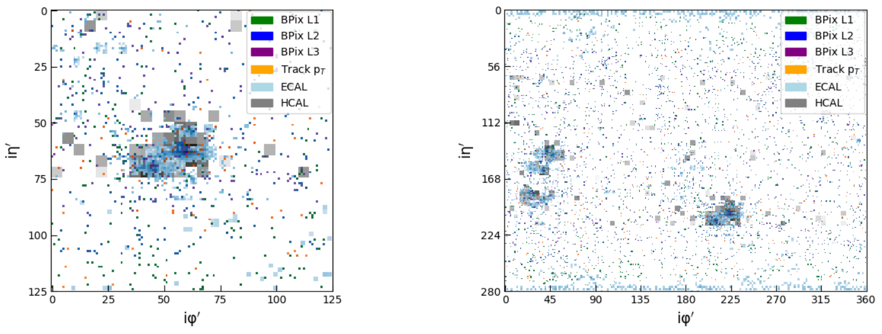

Figure 1. Composite image of a simulated boosted top quark jet (left). Image is constructed at the ECAL resolution.

For each layer, tracks weighted by their reconstructed p_T, their transverse ($d0$) impact parameter (IP) significance, or their longitudinal (dZ) IP significance and have their location projected to the nearest ECAL crystal. We also include layers produced from charge clusters in the silicon tracker. These reconstructed cluster hits (RecHits) are traditional used for track reconstruction and seeding.

3 Integration of End-to-End Deep Learning with `CMSSW` Framework

`CMSSW` is a C++-based framework built around the Event Data Model (EDM) [12]. In the EDM format, each entry (`edm::Event`) represents a single collision event able to hold multiple attributes on top of which various modules can be run to perform simulation, digitization, and reconstruction. These modules are categorized by function into `CMSSW` base classes for analyzing data collections (`edm::EDAnalyzer`) or producing new ones (`edm::EDProducers`), among others. The end-to-end framework (`E2EFW`) is designed to be highly modular and flexible in order to support customizable workflows relevant for end-to-end ML training and inference. The `E2EFW` can be optionally run to produce EDM-format files for further downstream processing by other `CMSSW` modules for a production workflow, or configured to produce `ROOT`-ntuples, for rapid prototyping.

3.1 Typical End-to-End Framework Pipeline

`E2EFW` consists of three main package categories: `DataFormats`, `FrameProducers`, and `Taggers`. The `DataFormats` package consists of all the objects and classes needed for running the `E2EFW` modules and for storing any output back into EDM-format files. These consist of convenience classes for handling inputs and defining association with other pertinent collections. In particular, the association maps for linking object-level detector inputs to their associated reconstructed physics objects are defined here.

The `FrameProducer` package is primarily responsible for extracting detector data—either as whole event data or as object-level data—and auxilliary functions aiding in this regard. To cast the whole event data to desired images or graphs, the data producers are provided for reading hit information from various CMS subdetectors. A number of modules are further provided that interface with the output of detector data producers for creating localized windows or crops around the coordinates of a desired reconstructed physics object. For reconstructed jet objects, such as used in this paper, the object level data producers allow to produce multi-subdetector data like images or graphs associated with each reconstructed jet. The `E2EFW` includes user-configurable parameters to control which subdetectors to include, as well as which jet clustering algorithm to use for determining their centers. For electrons and photons, a separate module is provided. This way, different combinations of the producers can be used for different tasks depending on user requirements.

The typical physics application will require more detailed analyses to be performed on the reconstructed objects. For this purpose, a third, Taggers package category is included to facilitate such studies. These can interface with the output of any of the previous `FrameProducer` packages allowing for modular, highly-customizable workflows. While most production-level analyses will have their own dedicated analysis workflow, the Taggers provide a quick and convenient avenue for rapid prototyping and analysis, such as is desirable during the ML algorithm development, but can also be used for running inference in a production-like workflow. We include a number of template modules for these purposes. For instance, we implemented the `TopTagger` package which contains the basic modules necessary for the selection of boosted hadronic $t\bar{t}$ decays. Additional template tagging packages, such as `QGTagger` and `EGTagger`, for the selection of quark/gluon jets [4] and electron/photon showers, respectively. For rapid prototyping during the development phase, the included Tagger packages provide the option for outputting directly to ROOT-ntuples for ease of conversion to an ML-friendly file format. For running inference in a production-like EDM-format workflow, implementations for running ML model inference within these Taggers packages and for storing the predictions into an EDM-format output are also provided.

In order to summarize a typical `E2EFW` pipeline, we take the workflow used in this boosted top quark study as a use case, as illustrated in Figure 2. The detector level data producer is first run to extract whole detector images or graphs corresponding to the ECAL, HCAL, and Track layers and storing these back into an EDM-format file in vector shape suitable for object-level cropping. Then, object level data producers are run to crop and process these whole detector vectors into jet level data around the coordinates of the reconstructed AK8 jets and again push back these vectors into the EDM-format file. Lastly, `TopTagger` is run to select only reconstructed AK8 jets passing the boosted top selection criteria. Running

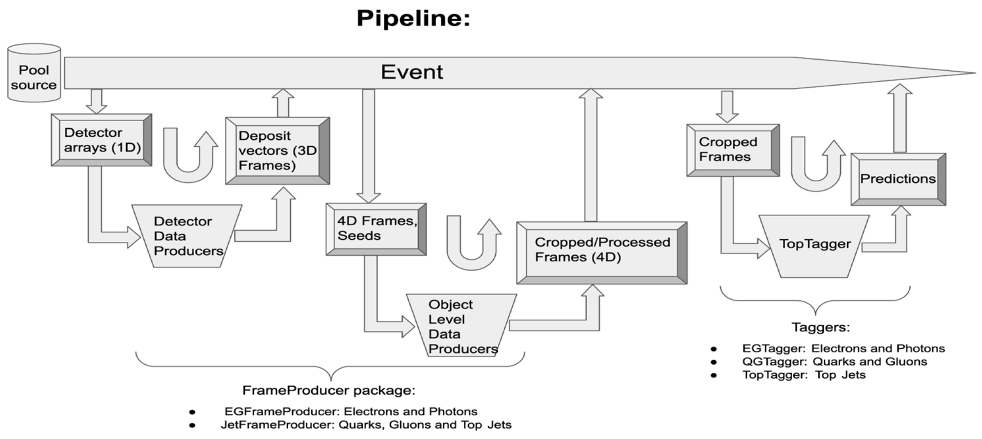
Pipeline:
Pool source
Event
Detector arrays (1D)
Deposit vectors (3D Frames)
Detector Data Producers
4D Frames, Seeds
Cropped/Processed Frames (4D)
Object Level Data Producers
Cropped Frames
Predictions
TopTagger
Taggers:
EGTagger: Electrons and Photons
QGTagger: Quarks and Gluons
TopTagger: Top Jets
FrameProducer package:
EGFrameProducer: Electrons and Photons
JetFrameProducer: Quarks, Gluons and Top Jets

Figure 2. The end-to-end framework (`E2EFW`) pipeline as used in the $t\bar{t}$ study

`TopTagger` on inference mode, the jet level data associated with the selected jets are then passed to the `Tensorflow` API for inference. The resulting predictions on these selected jets are then pushed back into the EDM-format file for further downstream analysis.

`Tensorflow` C++ API is used for inference on `Tensorflow` models, although we expect to support other deep learning packages in the future. Currently, `Tensorflow` C++ API is compatible with `Tensorflow` versions $\leq$ 1.6. For models trained and saved using later `Tensorflow` versions, there may arise issues of version incompatibility. The `E2EFW` is optimized to ML model inferences on CPUs as well as GPUs, if available.

While we have described jet-level workflow, the `E2EFW` can be appropriately adapted to process other objects and full events, by suitable definition of the process workflow. For instance, if a user wishes to perform event-level analysis and inference on whole detector inputs, an appropriate Tagger package that applies event-level selection can be defined that interfaces directly with the output of detector data producer and bypasses object level data producers. The `E2EFW` thus allows a high degree of flexibility for pursuing an assortment of end-to-end ML studies dictated by the user. For the benchmark study, we used Convolutional Neural Network (CNN) architectures for reconstruction of detector inputs as images for the deep learning model. The `E2EFW` can be easily configured to use other algorithms like Graph Neural Networks (GNNs) with graphs inputs, which is beyond the scope of this paper.

3.2 Network & Jet ID Results

The network architecture and hyperparameters used in this work closely follow what was previously used in [3, 4], making use of a ResNet-15 CNN [15] trained with the ADAM optimizer [16]. The full network infrastructure is outlined in Table 2 and it was constructed and trained using the TensorFlow library [17]. The network was trained for 40 epochs on a training set consisting of 2.56M jets. The initial learning rate was set to 5×10^{-4} and explicitly reduced by half every 10 epochs. When evaluated on a separate sample of 200k jets, we obtained an ROC-AUC value of 0.9824±0.0013 and a signal efficiency of 66.41% at 1% misidentification.

Block	Layer Type	Extra Parameters
Input Node	Conv 2D	filter size 7, stride 2, 16 channels, ReLU
	Global Pool 2D	2×2 pool size
Resblock 1	Conv 2D	filter size 3, stride 1, 16 channels, ReLU
	Conv 2D	filter size 3, stride 1, 16 channels ReLU
Resblock 2	Conv 2D	filter size 3, stride 2, 32 channels, ReLU
	Conv 2D	filter size 3, stride 1, 32 channels ReLU
Resblock 3	Conv 2D	filter size 3, stride 1, 32 channels, ReLU
	Conv 2D	filter size 3, stride 1, 32 channels, RELU
Output Node	Max Pooling 2D	global pool size
	Dense	size 32×2
	Activation	Sigmoid

Table 2. Model architecture used in this study. All convolutional layers used same padding

4 Scaling Deep Learning Training to Multiple GPUs

Deep Neural Networks (DNNs) are often computationally demanding. The training of the neural network often requires processing a large amount of data and can take days or even weeks to finish. Distributed training and data parallelism are widely used for training deep neural networks on multiple GPUs. For our purpose, we used data parallelism whereby the training data is divided into multiple subsets and fed to the computing nodes to be trained on the same replicated model in the nodes. The synchronisation of the model parameters across the nodes thus becomes an important step achieved by aggregating the model errors of all the devices and calculating the parameter updates to achieve synchronised update of the model parameters in the respective nodes. This process accelerates the optimisation of the model parameters by scaling with the amount of input data.

We applied several optimisation strategies to train the network on a cluster of commodity CPUs and GPUs with the goal of minimizing the training time without compromising the accuracy and efficiency. The computing architectures used for this study were Nvidia Tesla V100 (16GB). Further details about the hardware accelerators are given in the Table 3. The training data consisted of 2.56 million jet images, each consisting of 8 channels: track p_T + $d0$ + dZ + ECAL + HCAL + BPIX RecHits. The complexity of this data significantly strains the loading process, thus the training time of a single epoch exceeds 2 hours. The TensorFlow dataset object offers map transformation for facilitating the mapping of input elements according to a user defined function. With the help of map transformation functionality, the parallelism of the input pre-processing across multiple CPU cores can be achieved by specifying the number of parallel calls to it. In addition, the number of parallel reads of the records of the input data was set to auto-tune, in order to mitigate the data loading bottleneck.

For the scaling study, we used the Horovod framework [18] as it provides flexibility of scaling the training of the network to multiple GPUs. Horovod takes advantage of the inter-GPU and inter-node communication methods such as NCCL (Nvidia Collective Communications Library) and MPI (Message Passing Interface) to distribute the deep learning model parameters between various workers and aggregate them accordingly. As the data parallelism prototype in Horovod is not complicated, it was possible to run multiple copies of the training model on the computing nodes, wherein each node reads a specified chunk of data, performs forward propagation step of the respective model copies on it and then computes the model gradients accordingly. Horovod uses Baidu's Algorithm to average gradients and communicate those gradients to all the computing devices following the ring-allreduce decentralized strategy.

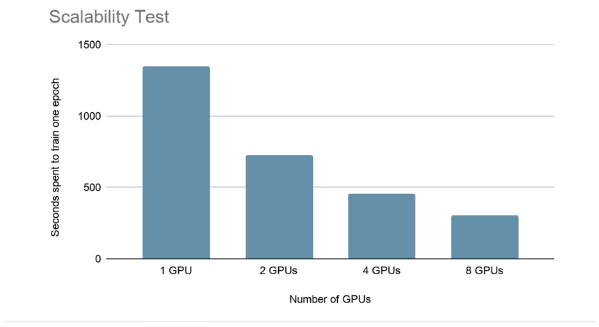

Figure 3. Scaling end-to-end deep learning training on multiple GPUs.

To increase the data loading speed, 20 CPU cores were used for processing. Further optimisation strategies such as using mixed precision and optimisation of batch size led to additional training speed increases. Figure 3 shows the training time when the final optimised model was scaled to multiple GPUs.

5 Timing Performance Comparison

We additionally performed timing studies comparing the I/O and training time of classifiers on different hardware platforms, whose specifications are summarized in Table 3: two different graphical processing units (GPU) and a tensor processor unit (TPU). Each device was accessed via different computing clusters with varying I/O architectures described below. The floating point operations per second speed (FLOPS) are quoted based on the data type and architecture setup during training.

Processor	Manufacturer	HBM Memory	Performance
Tesla P100	NVIDIA	16 GB	9.3 Single-Precision TeraFLOPS
Tesla V100	NVIDIA	32 GB / 16 GB	125 Mixed-Precision TeraFLOPS
TPUv3-8	Google	128 GB	420 Mixed-Precision TeraFLOPS

Table 3. Comparison of NVIDIA Tesla P100 [19], Tesla V100 [20], and Google TPUv3-8 [21]

The NVIDIA Tesla P100 is a GPU that utilizes the Pascal architecture [19]. The Tesla P100 GPU was accessed on a shared cluster at Fermilab via a dedicated GPU worker node through a 12 GB/s PCIe connection using the CUDA Toolkit v9.1 drivers. During training, data was stored and read from an HGST 1W10002 hard drive [22] located on the GPU machine. Images were uncompressed, pre-processed, and provided to the GPU using a single Intel(R) Xeon(R) Silver 4110 8-core CPU [23].

The NVIDIA Tesla V100 uses the Volta architecture, incorporating eight tensor cores and an all-around higher performance than the Pascal architecture [20]. Unlike the P100, the V100 is able to make use of mixed precision operations, which were utilized for this comparison, to drastically increase the number of floating point operations per second (FLOPS). During training, images were read from a solid state drive (SSD) with better random input/output operations per second. The computing node ran Cuda v11.0.2 drivers and used four Intel(R) Xeon(R) Gold 5118 12-core CPUs [24] to perform data loading and pre-processing tasks, which were parallelized using the NVIDIA DGX-1 architecture [25]. Of the 48 available CPU cores, 20 were used for data loading.

The TPU is a type of AI-accelerated architecture designed by Google with the purpose of training and running inference on machine learning models [26], and can be accessed through the Google Cloud platform [27]. TPUs boast a high number of FLOPS, made possible by dedicated matrix multiplication units which make use of the bfloat16 data type [28]. Unlike GPU based architectures, the CPU cores used to fetch and pre-process batches all live on TPU board creating a low latency I/O pipeline which does not have to compete for CPU resources. For this work, a TPUv3-8 running TPU software v1.14 was run using a type n1-standard-1 Google Cloud virtual machine node. Data used to train the networks was stored in Google Cloud buckets located in the same region as the VM and TPU to decrease time associated with transferring data during the training. This gives the cloud storage buckets comparable performance to persistent HDs in the local VM storage area [29].

Category	**Tesla P100**	**Tesla V100**	**TPUv3-8**
Training Batch Size	32 tf examples	64 tf examples	64 tf examples
Batches Per Epoch	80k batches	40k batches	40k batches
I/O Time (single batch)	0.119 s	0.0180 s	0.0176 s
Train Time (single batch)	0.481 s	0.0290 s	0.0202 s
Train Time (one epoch)	321 min	19 min	14 min

Table 4. I/O and training time comparison for different architectures

Table 4 shows a timing comparison for training machine learning models using the different architectures. A larger batch size was used when training on the Tesla V100 and TPUv3-8. Training times were found to vary by approximately 10% between epochs. The trainings performed on the P100 were drastically longer, primarily stemming from low I/O speeds. These low speeds come from random read inefficiencies associated with disk HDs [30] as well as a weaker CPU utilizing fewer CPU nodes when fetching batches and sending them to the GPU. The Tesla V100 and TPUv3-8 provide much stronger performance. When training on the Tesla V100 we took advantage of SSD storage, leading to improved random read speeds. We found that the I/O speeds for Tesla V100 and TPUv3-8 were almost identical. By subtracting the single batch I/O time from the single batch train time, we obtain an approximate computation time. From this, we see that the TPUv3-8 spends approximately 2.6 ms to perform forwards and backwards propagation calculations, which is a factor of four faster than the 11 ms required by the Tesla V100.

Comparison of all of the architectures studied shows that one of the largest training speed increases comes from improving the hardware associated with reading, decompressing, and pre-processing data. The training pipelines used to access the Tesla V100 and the TPUv3-8 used optimized data storage and highly parallelized CPU architectures to meet I/O speeds that were 10-20x faster than what was used to train on the Tesla P100. For future studies I/O speeds can be further reduced by improving how data is stored. To better optimize the balance between read and pre-processing speeds, a well-tuned compression factor can be used when storing the tf examples. When training on TPUs data can be stored as a bfloat16 to decrease storage space, the memory usage, decreasing decompression speeds while producing no overhead associated with type conversions during data loading.

6 Conclusions

In this work, we discuss the implementation and benchmarking of end-to-end deep learning application within the `CMSSW` framework. Identical copies of the benchmark model were trained on a single NVIDIA Tesla P100 GPU, a single NVIDIA Tesla V100 GPU, and a TPUv3-8 accessed through the Google Cloud platform. An improvement on the single largest training speed was observed with the optimization of I/O infrastructures. Both the Tesla V100 GPU and the TPUv3-8 offer a significant training time improvement over a Tesla P100 GPU when training on end-to-end jet images. Scaling the training of the deep learning model to multiple GPUs resulted in significant reduction in the time required with the help of various optimisation and data parallelism techniques. The training speed was boosted by more than 15 times due to the various heterogeneous computing strategies deployed without compromising the performance of the algorithm.

Acknowledgments

We would like to thank the CMS Collaboration and the CERN Open Data group for releasing their simulated data under an open access policy. We strongly support initiatives to provide such high-quality simulated datasets that can encourage the development of novel but also realistic algorithms, especially in the area of machine learning. We believe their continued availability will be of great benefit to the high energy physics community in the long run. We would also like to thank Google and NVIDIA for access to their servers which helped us speed up the training of machine learning models used in this study.

References

[1] JINST **12**, P10003. 82 p (2017)
[2] *CMS Physics: Technical Design Report Volume 1: Detector Performance and Software*, Technical Design Report CMS (CERN, Geneva, 2006), `https://cds.cern.ch/record/922757`
[3] M. Andrews, M. Paulini, S. Gleyzer, B. Poczos, Computing and Software for Big Science **4** (2020), `1807.11916`
[4] M. Andrews, J. Alison, S. An, B. Burkle, S. Gleyzer, M. Narain, M. Paulini, B. Poczos, E. Usai, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment **977**, 164304 (2020)
[5] K. Albertsson et al., arXiv e-prints (2018), `1807.02876`
[6] M. Andrews, B. Burkle, D. DiCroce, S. Gleyzer, U. Heintz, M. Narain, M. Paulini, N. Pervan, E. Usai, Submitted to VCHEP2021 (2021)
[7] CMS Collaboration, *CMS data preservation, re-use and open access policy* (2014), `http://opendata.cern.ch/record/411`
[8] CMS Collaboration (2019), `http://opendata.cern.ch/record/12200`
[9] CMS Collaboration (2019), `http://opendata.cern.ch/record/12201`
[10] CMS Collaboration (2019), `http://opendata.cern.ch/record/12202`
[11] CMS Collaboration (2019), `http://opendata.cern.ch/record/12203`
[12] *Cms software version 5_3_32 (cmssw_5_3_32)* (2016), `http://opendata.cern.ch/record/221`
[13] E. Usai, M. Andrews, B. Burkle, S. Gleyzer, M. Narain, CERN Open Data Portal (2019)
[14] M. Cacciari, G.P. Salam, G. Soyez, Journal of High Energy Physics **2008**, 063–063 (2008)

[15] K. He, X. Zhang, S. Ren, J. Sun, *Deep residual learning for image recognition* (2015), `1512.03385`

[16] D.P. Kingma, J. Ba, *Adam: A method for stochastic optimization* (2014), `1412.6980`

[17] M.A. et al., *TensorFlow: Large-scale machine learning on heterogeneous systems* (2015), software available from tensorflow.org, `http://tensorflow.org/`

[18] A. Sergeev, M.D. Balso, *Horovod: fast and easy distributed deep learning in tensorflow* (2018), `1802.05799`

[19] *NVIDIA Tesla P100: The Most Advanced Data Center Accelerator*, accessed: 28 August 2020, `https://www.nvidia.com/en-us/data-center/tesla-p100/`

[20] *Nvidia v100 | nvidia*, accessed: 16 September 2020, `https://www.nvidia.com/en-us/data-center/v100/`

[21] *Cloud tensor processing units (tpus)*, accessed: 30 August 2020, `https://cloud.google.com/tpu/docs/tpus`

[22] *Western Digital DC HA210 Datasheet, 3.5 Inch Data Center Hard Drives*

[23] *Intel Xeon Silver 4110 Processor (11M Cache, 2.10 GHz) Product Specifications*, accessed: 16 September 2020

[24] *Intel xeon gold 5118 processor (16.5m cache, 2.30 ghz) product specifications*, accessed: 16 September 2020

[25] *Nvidia dgx-1: Deep learning server for ai research*, accessed: 16 September 2020, `https://www.nvidia.com/en-us/data-center/dgx-1/`

[26] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al. (2017), `1704.04760`

[27] *Google Cloud Computing Services*, `https://cloud.google.com/`

[28] *BFloat16: The secret to high performance on Cloud TPUs*, `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus`

[29] *Storage classes | google cloud*, accessed: 29 September 2020, `https://cloud.google.com/compute/docs/disks`

[30] *Advances in Flash Memory SSD Technology for Enterprise Database Applications*, SIGMOD '09 (Association for Computing Machinery, New York, NY, USA, 2009), ISBN 9781605585512, `https://doi.org/10.1145/1559845.1559937`