Optimized Inference Engine Generation for Advanced Deep Learning Models

Robin Abrahamse[*] *& Vladimir Lončar*[†]
[*]University of Technology Delft, The Netherlands
[†]European Organization for Nuclear Research (CERN)

Abstract
With the prospect of ever-increasing luminosity in the Large Hadron Collider (LHC) and particle collider experiments in general, there is a growing demand for efficient data processing and analysis tools in both online and offline settings. Among others, the HLS4ML framework has demonstrated that deep learning inference can be used effectively for efficiently executing data analysis tasks in High Energy Physics (HEP). The current project builds and improves on recent efforts to generate efficient inference engines for deep learning models with hardware-specific optimizations using the Intel oneDNN library and HLS4ML framework. A functioning HLS4ML backend was built for translating common deep learning model formats (Tensorflow, Keras, Pytorch, and ONNX) to oneDNN-accelerated inference engines, with support for advanced non-sequential models. The generated inference engines show inference latencies and peak memory footprints improving on or meeting state-of-the-art tools for the renowned ResNet50V2 and MobilenetV2 models. Moreover, this report includes a brief analysis of system feasibility and potential future work, the latter of which is already partly in progress.

Keywords
Machine Learning; Inference Engines; HLS4ML; HLS; Intel oneAPI; Intel oneDNN; Data Parallel C++

1 Introduction

The future of the Large Hadron Collider (LHC) and High Energy Physics (HEP) in general is increasingly focused on raising collider luminosity, especially with the European wide assigned priority on High-Luminosity (HL) upgrade for the LHC [3]. HL-LHC aims to increase luminosity with a factor of roughly ten, implying a similar increase in Data Acquisition (DAQ) volume, which puts extra strain on the data storage systems. Successful scaling to HL-LHC requires improved DAQ filtering at both online and offline time scales to ensure feasibility of data storage. As accelerated deep learning solutions have already shown promise to aid in improving data acquisition filtering performance [1, 4, 6], the current project aims to explore a new type of deep learning inference acceleration using Intel oneAPI Deep Neural Network Library (oneDNN) [9].

In order to accommodate the large volumes of generated data, the LHC leverages a DAQ filtering system composed of 2 trigger levels; the initial L1 trigger and the High-Level Trigger (HLT). The L1 trigger consists of FPGA-based systems that have microseconds latency requirements, feeding into the HLT that is based on CPU-powered software systems and has latency requirements of several hundreds of milliseconds [2]. The L1 trigger filters the ~40MHz LHC signal to ~100kHz and then the HLT filters that signal down to ~400Hz for permanent storage [2] (Fig. 1). Recently, FPGA-based deep learning inference using the HLS4ML framework [7] leveraging Xilinx High-Level Synthesis (HLS) has shown promise for usage in future L1 upgrades [1, 4, 6].

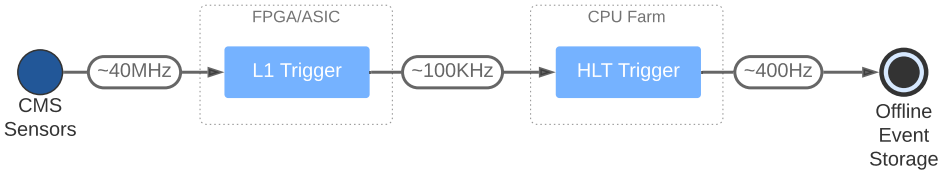


Figure 1: CMS Trigger Diagram

The current project attempts a similar effort with HLS4ML for CPU-based systems using oneDNN while in the process adding support for a variety of heterogeneous architectures. Considering that generally the HLT is situated on a CPU farm, future upgrades to the triggering system could potentially benefit from such optimized CPU-based inference approach. Moreover, oneDNN supports reduced-precision inference optimization and compilation for hetergeneous architectures (including FPGA and GPU execution), which might further improve performance and scalability of the triggering system with increasing luminosity.

As such, an HLS4ML backend based on oneDNN was developed to generate the hardware-optimized deep learning inference engines and their latency performance was tested for different batch sizes. The project was commissioned by the Compact Muon Solenoid (CMS) experiment at the LHC but its results should easily generalize to other parts of the LHC and even data analysis in other fields where there is demand for low-power, low-latency deep learning inference.

This report will first explain the general concepts of HLS4ML and the developed oneDNN backend in Section 2. Then, Section 3 will outline the experiment and respective results. Lastly, a brief analysis of system feasibility and future work is given in Section 4.

2 HLS4ML

HLS4ML [7] is an open-source framework purposed to translate machine learning models of the most common formats (Keras, Tensorflow, Pytorch, and ONNX) to FPGA firmware for low-latency inference. The system acts as a specialized compiler that interprets the layer structure of a deep learning model and transforms it into an internal intermediate representation. This representation can then be transformed to firmware for different kinds of FPGAs or ASICs using High-Level Synthesis (HLS). This section will elaborate on the framework itself and the newly developed oneDNN backend.

2.1 Framework

The HLS4ML framework [7] is python-based and receives as input a deep learning model in one of the aforementioned formats with its corresponding weights. Moreover, a configuration detailing the hardware target and several performance settings is input. The framework then outputs a firmware package to be loaded on the target hardware. It allows configurability of model compression and quantization for further performance and resource control [1].

As mentioned before, HLS4ML interprets the input model in terms of its layer structure. The layers are stored in an internal representation and a (extensible) series of graph-scale optimization passes are made to analyse and optimize the layer structure. Based on the supplied configuration, the framework will then generate appropriate C code using templating to be transformed into HDL with supported HLS tools (e.g. Xilinx Vivado and Intel Quartus), as displayed in Fig. 2.

Considering that FPGA-based deep learning inference is still a maturing field, no published literature was found comparing HLS4ML firmware performance to alternative tools. However, achieving up to sub-microsecond latency for small models [4, 6] and low energy usage [4] shows it can offer benefits over traditional inference approaches.

2.2 OneDNN Backend

Since HLS4ML maintains an internal representation of the input model, the framework is not limited to conversion of the deep learning models to HDL. With the inception of oneDNN as part of the new Data Parallel C++ paradigm, which aims to supply hardware-aware optimization across different (heteregeneous) platforms, a new opportunity for HLS4ML arises.

OneDNN is a deep learning library built on Data Parallel C++ with the purpose of supplying a library to built generic training and inference applications. These applications are then optimized for the specific targeted hardware (i.e. CPU, GPU, or FPGA/ASIC) by the compiler. A specialized oneDNN backend allows HLS4ML to automatically generate oneDNN inference engines, which can consequently be compiled for hardware-optimized execution, as displayed in Fig. 2 (added components in green).

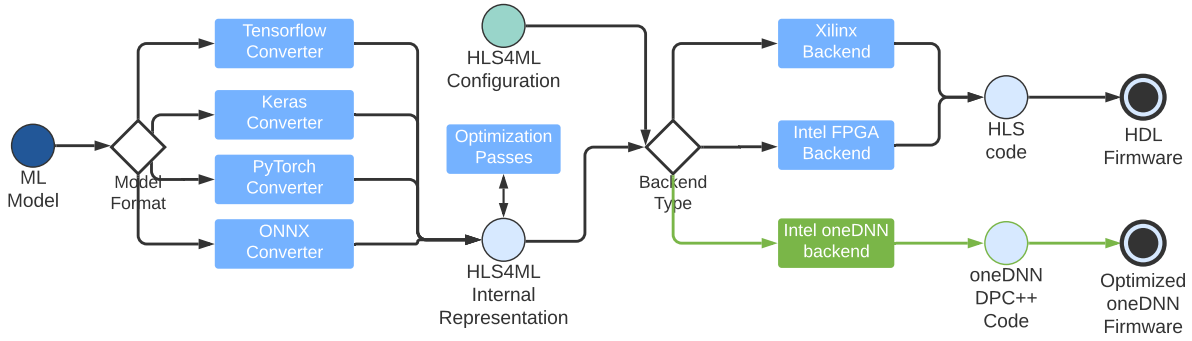
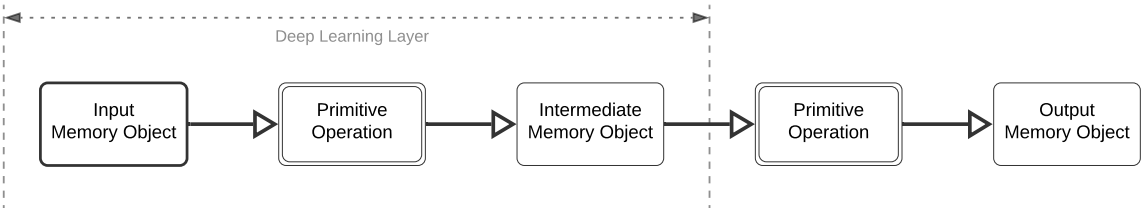


Figure 2: System Diagram for HLS4ML with oneDNN Backend

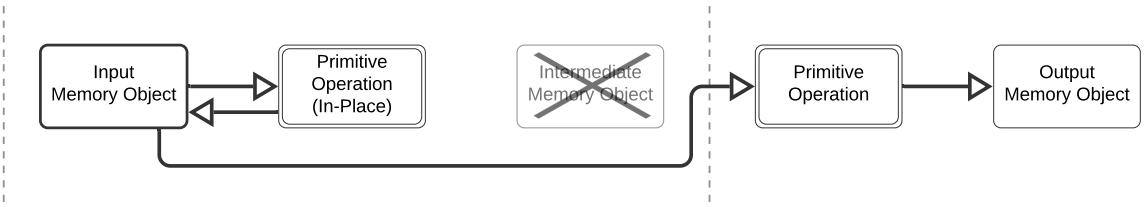
The developed oneDNN backend builds upon a previous effort, which is detailed in Ref. [14]. This previous effort developed the general framework for generating main deep learning operations (e.g. convolution and dense layers) and provided a proof-of-concept. The generated code still required manual modification for successful compilation and suffered from robustness issues. The current project improves the robustness and extends the functionality of the previous effort such that complex and non-sequential models are now supported and manual intervention is no longer required. Additionally, a few simple optimization strategies are applied.

The backend is essentially a templating engine, generating code that fully complies with the oneDNN programming model. At the core, it builds a pipeline with a series of oneDNN memory objects – starting with the input memory and ending in the output memory, with intermediate memory objects and weights memory objects in between – that are connected by deep learning operation 'primitives'. The primitives are oneDNN objects that execute the layer operations of the input model and route the output from the previous layer to the next layer by using memory objects (Fig. 3a).

Some primitives – most operations that maintain the input dimensions – support in-place execution, reducing the number of required memory objects. As such, an optimization was implemented for sequential models which reuses memory objects for in-place primitives and thus reduces memory footprint (Fig. 3b). This is currently disabled for non-sequential models as the intermediate memory objects may be input for other primitives and memory reuse may overwrite the contained data. An additional optimization pass can be implemented to apply this optimization in layers of non-sequential models that do not suffer of this overwrite issue as well. It should be noted that the effect of in-place operations on latency was not tested but disabling the optimization is trivial.



(a) Default Pipeline



(b) Optimized Pipeline

Figure 3: Memory-Reuse Pipeline Optimization

Additionally, an optimization pass was implemented to leverage logical zero padding rather than physical, meaning that the zero pads are not physically contained by memory objects but calculated logically in the primitives instead. This means that zero padding layers and corresponding memory objects can be eliminated, thus reducing memory footprint and potentially execution latency as well.

The oneDNN backend currently supports the following (HLS4ML) layers (missing layers are detailed in Section 4):

- Input
- Dense
- Conv1D, SeparableConv1D[1]
- Conv2D, DepthwiseConv2D, SeparableConv2D[1]
- Pooling1D, GlobalPooling1D
- Pooling2D, GlobalPooling2D
- ZeroPadding1D, ZeroPadding2D
- Activation, ParameterizedActivation
- Softmax
- BatchNormalization[1]
- Merge (Add & Subtract operations only)

It should be noted that in some models race conditions in the generated inference engines resulting in sporadically inconsistent outputs were identified (using Intel Inspector). Since the generated code generally complies with the oneDNN programming model, it appears unlikely that this affects performance considerably. Similarly, the experimental layers do not consistently output the desired results but they follow the oneDNN programming model and thus should yield valid for latency and memory measurements. Both these issues fall outside of the scope of the current project due to time constraints but they require further scrutiny nevertheless.

[1]Experimentally, see Section 4

3 Experiment & Results

The oneDNN backend was developed to meet or exceed the performance of current state-of-the-art tools in terms of latency primarily and resource usage secondarily. As such, the performance of generated code was tested on a modern CPU-based server system and compared to several different existing solutions. This section will elaborate on the benchmark experiment and the retrieved results.

Considering that the main goal of the oneDNN backend is to generate low-latency inference engines, latency is the main metric for benchmarking. The oneDNN backend was leveraged to generate inference engines, which were used for measuring inference latency of deep learning models. All benchmarks were conducted using the same system on Intel DevCloud [8], details of the system can be found in Table 1.

Hardware	Intel® Xeon® Gold 6128 CPU @ 3.40GHz, 192GB RAM
Operating System	Ubuntu 18.04
Library Versions	Intel oneDNN 2.4.1 Intel-Tensorflow 2.3.0 Tensorflow 2.3.0 ONNX Runtime 1.8.1

Table 1: Measurement System Details

The performance of HLS4ML oneDNN inference engines was benchmarked against several state-of-the-art inference engine tools. Due to temporary HLS4ML constraints at the time of the project, only Tensorflow Keras models could be tested, meaning that the comparisons were also restricted to tools supporting Keras models; however, in the past CPU inference performance of common inference engines has been comparable to Tensorflow for sufficiently parallel systems [13]. Ultimately, Intel optimized Tensorflow (intel-tf in Fig. 4), regular Tensorflow (vanilla-tf in Fig. 4), and ONNX Runtime (ONNX in Fig. 4) were compared to HLS4ML oneDNN (HLS4ML OneAPI in Fig. 4) performance.

The Intel optimized version of Tensorflow requires setting parameters to account for specific system parallelism. All recommended settings as described by Intel in Ref. [15] were adopted to achieve this, resulting in the following example bash command:

```
$ OMP_NUM_THREADS=6 python test.py --num_intra_threads=6
--num_inter_threads=2 --kmp_affinity=granularity=fine,compact,1,0
--kmp_blocktime=0
```

Additionally, Tensorflow and Intel-Tensorflow inference was executed using '`model.predict()`' as well as '`model()`' (the `__call__` attribute). This distinction was made to map the performance difference that Tensorflow Keras models exhibit with these respective inference functions.

The deep learning models included in the benchmark measurements are *ResNet50V2* and *MobileNetV2*. These models were selected due to their image classification performance in terms of both accuracy and latency in combination with their modern, non-sequential architectures. These characteristics make that the latency performance of such models is interesting for purposes of usage in the LHC, such as in particle classification tasks.

The measurements consisted of 20 times repeated runs of inference on a set of 12 224x224 images, arbitrarily picked from Ref. [5]. Due to the layered nature of deep learning models, the input images' contents should not matter for inference latency or memory performance; however, a set of images was selected to eliminate potential cache warming over multiple measurements. The measurements were repeated with batch sizes ranging from 1 to 12 – based on the maximum number of available system processing cores. Latency was measured using the python *time* module and C++ *std::chrono::high_resolution_clock* for oneDNN.

Additionally, the peak memory footprint was measured for all benchmarks with memtime [10] and averaged over 10 repeated runs. Memtime polls '`/proc/$PID/stat`' in Linux to gather statistics regarding the peak resident set size of a process, that is used as a measure of peak memory footprint here. Due to the aforementioned issues with the current oneDNN backend, memory measurements of HLS4ML oneDNN inference engines with batch sizes larger than 1 were highly distorted and therefore excluded.

The results of the measurements are presented in Figs. 4 and 5, with measurements averaged per sample and mean±standard deviation (SD) plotted. It shows that the oneDNN inference engines consistently outperform Tensorflow with matching batch size. The inference engine latencies are also very similar to ONNX Runtime for batch size bigger than 1. Moreover, for batch size 1 the oneDNN inference engines show a smaller peak memory footprint than the other tools for all set sizes. Measurements for batch sizes larger than 1 remain to be performed.

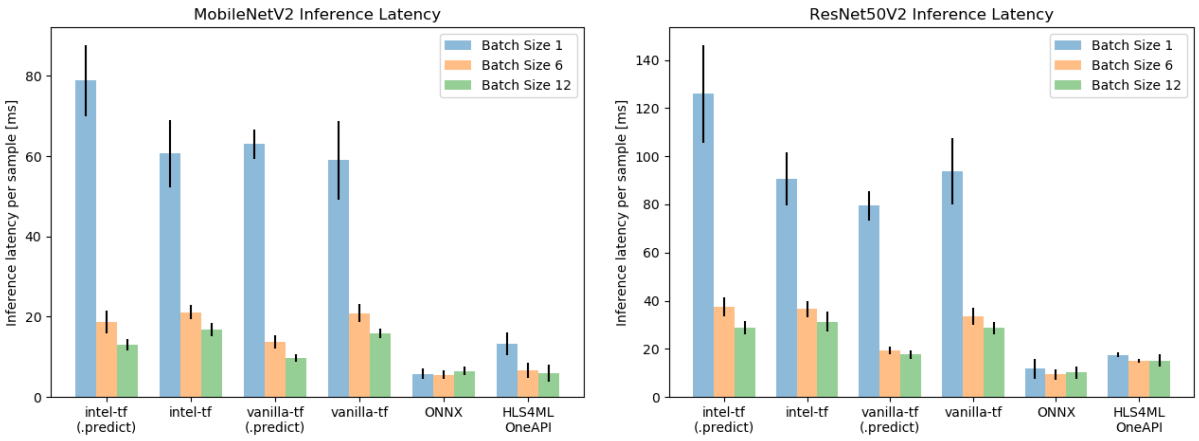


Figure 4: Inference Engine Latencies per Sample (Mean±SD)

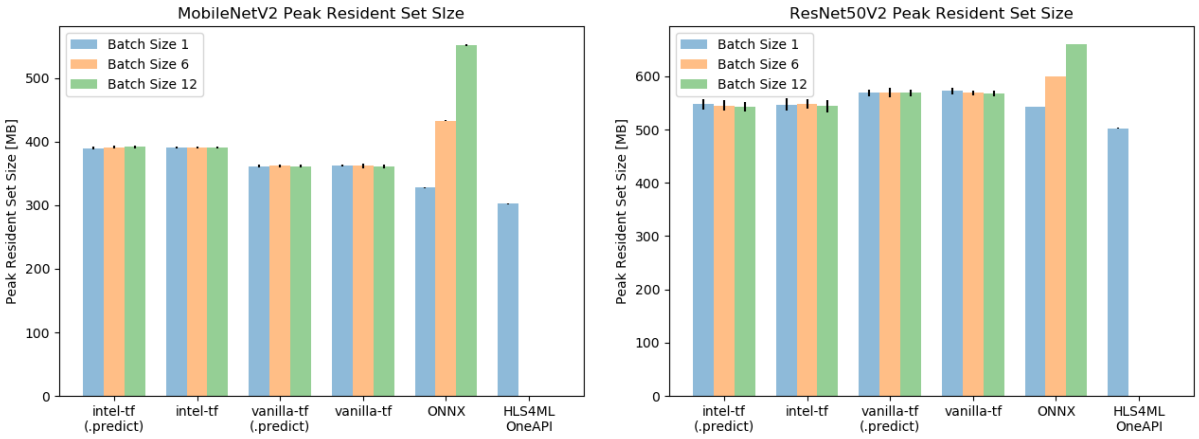


Figure 5: Inference Engine Peak Resident Set Sizes (Mean±SD)

4 Discussion

From the results it appears that ONNX Runtime is indisputably the fastest inference engine of the ones compared. In some of the measurements, the oneDNN inference engines generated with HLS4ML practically match ONNX Runtime in terms of latency and with batch size bigger than 1, latency performance

is generally very close and peak memory footprint lower. This section will discuss the implications of the current project and identify future work.

Even though no peer-reviewed literature was found on the performance of ONNX Runtime on CPU compared to other tools, several sources affirm a relative speed-up of around 2x over more conventional tools [11, 12]. The fact that the developed oneDNN backend inference engines closely follow ONNX Runtime in latency – even with little graph-scale optimization – indicates a potential value of the backend. With more effort on graph-scale optimization and the potential added benefits of reduced-precision support in oneDNN, the backend could demonstrate significant added value to the field of efficient deep learning inference.

Furthermore, peak memory footprint measurements indicate that the oneDNN inference engines use memory more sparingly than the compared tools. Future memory footprint measurements with larger batch sizes should provide additional insight into the scalability as well. Similarly to latency improvements, memory footprint could potentially improve with reduced-precision inference and more elaborate graph-scale optimization, an example is mentioned in Section 2.2.

The current project provides merely the initial steps in the research of oneDNN deep learning compilers. It provides evidence for a significant performance potential HLS4ML oneDNN compilation but more research is needed to further scrutinize its benefits and drawbacks. Moreover, inference performance is not the only measure which underlies the future of this effort. The rest of this section will elaborate on the project feasibility considerations and future work.

4.1 Feasibility

The inference performance measurements indicate a significant value of HLS4ML oneDNN compilation, but are not the only feasibility measures of interest. For example flexibility and extensibility as well as ease of maintenance are important factors in development of this tool.

It appears that flexibility and extensibility in development are limited with oneDNN. For example, the Keras ReLU layer supports 3 input arguments (maximum value, negative slope, and threshold), whereas no such operation currently exists or can be constructed in oneDNN, obstructing proper code generation. Additionally, not all operations for merging parallel layers in non-sequential models – such as the Keras Mult, Dot, Max, and Min layers – are supported by oneDNN. Primitives are currently limited to support the Keras Add, Subtract, Average and Concatenate layers only. However, oneDNN is an open-source library and extending its functionality is possible but that requires a significantly deeper understanding of the underlying programming model by developers.

Furthermore, the abstract nature of oneDNN objects and optimizations currently results in obscure error messages. This means that debugging issues with generating or generated code – and thus code maintenance – becomes significantly harder. For example, oneDNN may display the following message in the highest verbosity setting:

```
terminate called after throwing an instance of 'dnnl::error'
  what(): could not create a memory object
Aborted
```

This message does not explicitly mention any (potential) reasons for failure or locations in the code where this error occurred, which quickly becomes tedious in larger models. Moreover, availability of debugging tools for oneDNN in general is currently minimal.

Considering that oneAPI and therefore oneDNN is still in a maturing phase, the aforementioned aspects could be subject to rapid improvement in the near future. However, they should be considered in assessing feasibility of maintaining and progressing the current project.

4.2 Future Work

To continue the research of HLS4ML oneDNN compilation, several next steps are identified. Potential future work can be subdivided into three types; (1) completion of existing work, (2) addition of functionality, and (3) measurement and comparison of additional statistics.

The highest priority on this list is the completion of the existing work. It is important to solve existing bugs and to complete support for remaining layers before gathering new statistics in order to assure valid results. This work consists of the following:

- Identify and fix race conditions.
- Fix known bugs:
  - BatchNormalization does not consider provided weights.
  - Pointwise convolution causes NaN.
- Support missing (HLS4ML) layers:
  - Conv2DBatchnorm
  - PReLU, Activation, ParameterizedActivation (ThresholdedReLU, ELU, etc.)
  - GarNet, GarNetStack
  - Dot[2], Concatenate, Merge (Multiply[2], Maximum[2], Minimum[2], Average)
  - Resize[2], Transpose

Furthermore, addition of features supported by oneDNN could provide interesting results based on latency and/or memory footprint statistics:

- Reduced-precision inference (bfloat16, uint8, and int8).
- Memory-reuse optimization in non-sequential models (as outlined in Section 2.2).
- Channels-last support in code generation (to investigate potential performance impact).

Lastly, additional measurements could provide essential context and potential insight for the performance of HLS4ML oneDNN inference engines:

- Comparison against more existing tools (e.g. PyTorch, Caffe, etc.) with more diverse models
- Performance measurement of oneDNN execution on heterogeneous architectures (e.g. FPGA, GPU)
- Performance comparison between HLS4ML Xilinx FPGA execution, Intel FPGA execution, and oneDNN FPGA execution.

Efforts are already being undertaken to complete the existing work and add support for reduced-precision inference. The current work can be found on `https://github.com/RobinAbrahamse/hls4ml/tree/onednn`.

Bibliography

[1] Aarrestad, T., Loncar, V., Ghielmetti, N., Pierini, M., Summers, S., Ngadiuba, J., ... & Hoang, D. (2021). Fast convolutional neural networks on FPGAs with hls4ml. *arXiv preprint arXiv:2101.05108*.

[2] CMS collaboration. (2016). The CMS trigger system. *arXiv preprint arXiv:1609.02366*.

[3] CERN Council. (2013). The European Strategy for Particle Physics Update 2013. *https://cds.cern.ch/record/1567258/files/esc-e-106.pdf*.

[4] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., ... & Wu, Z. (2018). Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation, 13*(07), P07027.

[2] Currently not supported by oneDNN

[5] Google Landmark Retrieval 2020 Train [224x224]. *Accessed August 2021 on* https://www.kaggle.com/mikelgr500/google-landmark-retrieval-2020-train-224x224.

[6] Heintz, A., Razavimaleki, V., Duarte, J., DeZoort, G., Ojalvo, I., Thais, S., ... & Wu, Z. (2020). Accelerated charged particle tracking with graph neural networks on FPGAs. *arXiv preprint arXiv:2012.01563*.

[7] HLS4ML. https://fastmachinelearning.org/hls4ml.

[8] Intel® DevCloud for oneAPI. https://devcloud.intel.com/oneapi

[9] Intel® oneAPI Deep Neural Network Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onednn.html.

[10] Bengtsson, J. Memtime. http://www.update.uu.se/~johanb/memtime/.

[11] Pulavarthi, P. (2019). ONNX Runtime: a one-stop shop for machine learning inferencing. *Accessed September 2021 on* https://cloudblogs.microsoft.com/opensource/2019/05/22/onnx-runtime-machine-learning-inferencing-0-4-release.

[12] Alluin, M. (2021). An empirical approach to speedup your BERT inference with ONNX/Torchscript. *Accessed September 2021 on* https://towardsdatascience.com/an-empirical-approach-to-speedup-your-bert-inference-with-onnx-torchscript-91da336b3a41.

[13] Shi, S., Wang, Q., Xu, P., & Chu, X. (2016). Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)* (pp. 99-104). IEEE.

[14] Świniarski, M. (2020). Inference engine for custom neural networks with oneAPI. *Zenodo*. https://doi.org/10.5281/zenodo.4047756.

[15] Xu, J., Venkatesh, P., & Tsai, H. (2021). Maximize TensorFlow* Performance on CPU: Considerations and Recommendations for Inference Workloads. *Accessed August 2021 on* https://www.intel.com/content/www/us/en/developer/articles/technical/maximize-tensorflow-performance-on-cpu-considerations-and-recommendations-for-inference.html