Efficient Data Movement for Machine Learning Inference in Heterogeneous CMS Software

C. Zeh [1, 5] L. Michalski [2, 5] L. Beltrame [3, 5] D. Valsecchi [4] F. Pantaleo [5] E. Cano [5]

On behalf of the CMS Collaboration

[1]Technical University Vienna [2]Wroclaw University of Science and Technology [3]Polytechnic of Milan

[4]ETH Zurich [5]European Organization for Nuclear Research (CERN)

LinkedIn: Christine Zeh Lukasz Michalski

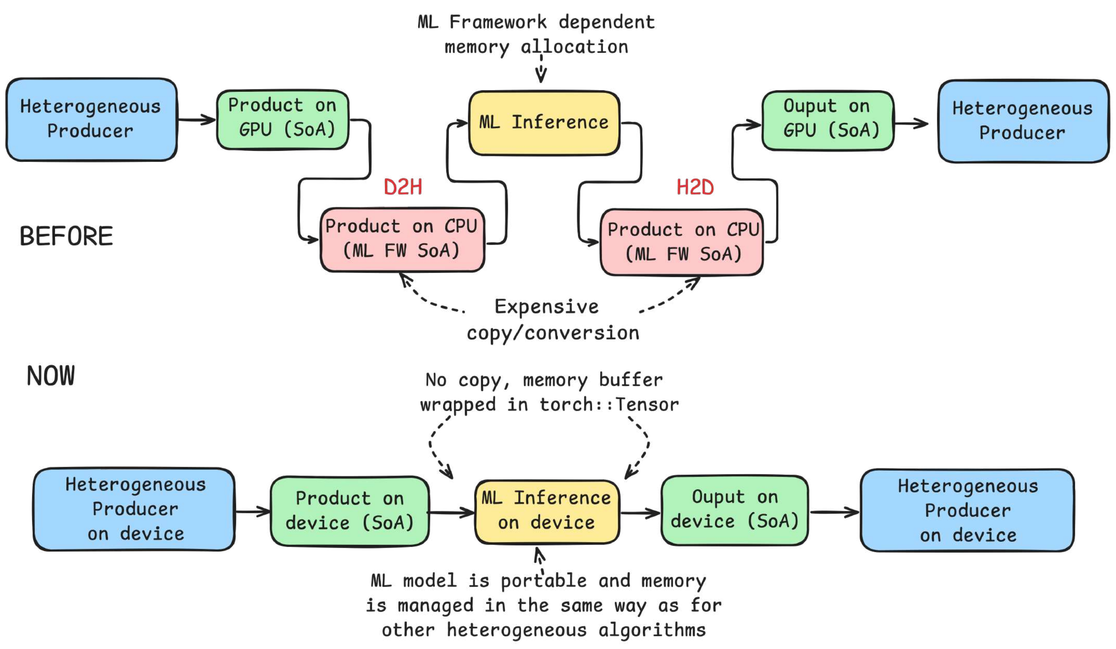
Abstract

Machine learning (ML) on GPUs is often slowed by redundant data movement between GPU memory and tensor formats. We present a new interface that converts Structure-of-Arrays (SoA) data directly into PyTorch tensors. This avoids explicit transfers, computes strides for different data types, and uses metadata to define tensor layouts for direct GPU access.

Our approach eliminates unnecessary memory copies and accelerates model execution. By integrating with the alpaka library, it also supports heterogeneous environments. The result is improved GPU efficiency and simplified integration of ML models into high-performance workflows. It enables linear conversion time and depending on the size of the input data, this saved up to 500 ms in the benchmarks.

Motivation

- Adoption of efficient memory structures like SoAs and PortableCollections.
- Lack of a streamlined interface for ML model inference in (alpaka-based) workflows.
- ONNX requires multiple copies and tensor conversions, leading to inefficiency.
- TensorFlow has limited GPU support for SoA usage.
- Users rely on suboptimal memory management, such as tensor conversions.
- Opportunity to directly interact with memory using `torch::from_blob()` (not readily available in TensorFlow/ONNX).

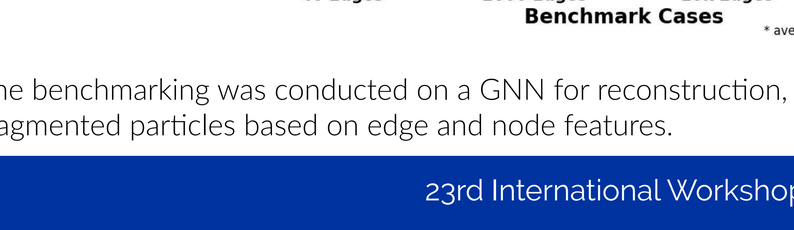


Copy-Efficient Memory Buffer Manipulation

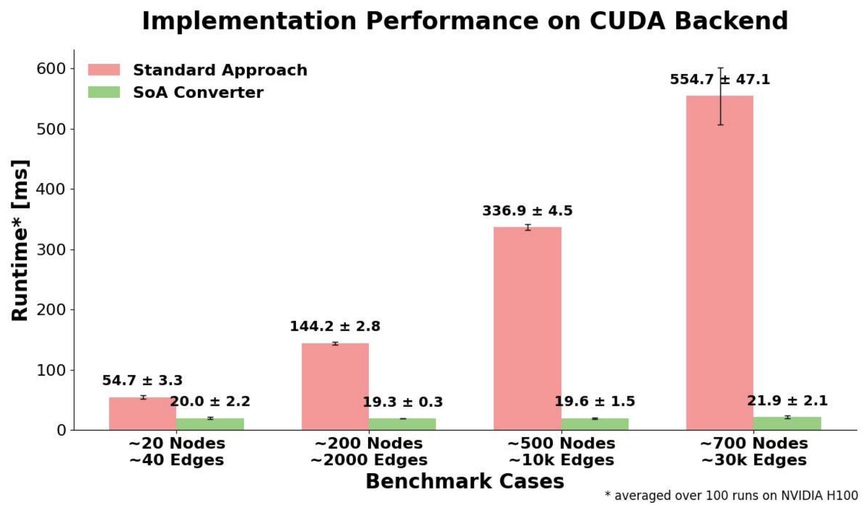
Optimizing data movement is essential for efficient ML on heterogeneous hardware, such as Nvidia or AMD GPUs. We introduce a flexible interface that converts structured physics data into PyTorch tensors without copying.

- SoA Layout: Structure-of-Arrays format for fast, parallel access on GPUs.
- Zero-Copy Tensors: Direct conversion to PyTorch tensors, no data movement.
- Metadata-Driven: Compile-time field mapping using C++ templates.
- User-Friendly: High performance without low-level memory management.

Check out the SoA Poster:



Benchmarking Results



The benchmarking was conducted on a GNN for reconstruction, merging fragmented particles based on edge and node features.

Heterogeneous Inference

Integrating PyTorch prioritizes tight control over threading, device streams, and resource management to ensure safe and scalable ML inference.

- Thread-Safe: PyTorch's internal threading is disabled.
- Scoped Scheduling: all execution is bound to CMSSW's threads pool, ensuring no unmanaged parallelism.
- Stream-Aware Execution: ML operations run on CMSSW-managed CUDA/HIP streams using queue-based resource control
- Scalable Design: Efficient for both large models and lightweight preprocessing steps.
- Multi-Device Ready: Concurrent execution across multiple GPUs and CPU, with models deployed on all required devices.
- Hardware-Aware Scheduling: Models are deployed just-in-time on the target device, triggered by the first scheduled event.

Model Compilation for Low-Latency Inference

To enable deep learning in high-energy physics workflows, integrate both *ahead-of-time* (AOT) and *just-in-time* (JIT) compilation strategies.

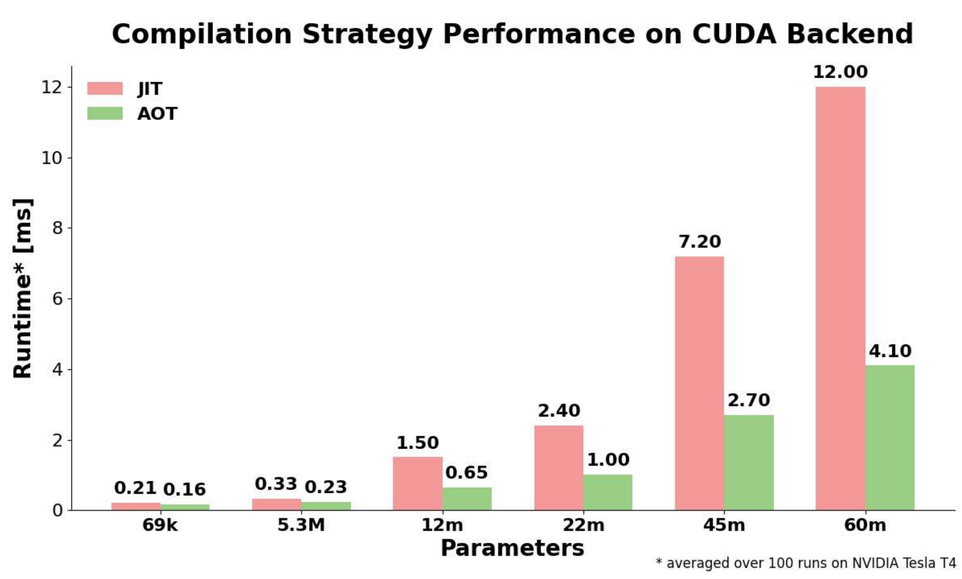
just-in-time

- Compiles models dynamically at runtime.
- Ideal for flexible development and quick iteration.
- Supports on-the-fly execution without build-time preparation.

ahead-of-time

- Uses `TorchInductor` and `Triton` to compile models into optimized binaries for specific hardware (e.g., NVIDIA GPUs).
- Produces shared library, loaded at runtime using standard C++ interfaces

AOT should be always used for production due to reduced runtime overhead, reproducibility, and maximum performance.



Inference Code

From the user's perspective, integrating a PyTorch model into an alpaka-based heterogeneous CMSSW producer is straightforward.

```
using namespace cms::torch::alpaka;

// structs
PortableCollection<SoAInputs, Device> inputs_device(batch_size, alpaka_device);
PortableCollection<SoAOutputs, Device> outputs_device(batch_size, alpaka_device);

// instantiate model
auto model = Model<CompilationType::kJustInTime>(m_path);
model.to(queue);

// metadata for automatic tensor conversion
auto input_records = inputs_device.view().records();
auto output_records = outputs_device.view().records();
SoAMetadata<SoAInputs> inputs_metadata(batch_size);
inputs_metadata.append_block("features", input_records.x(), input_records.y(),
↪  input_records.z());
SoAMetadata<SoAOutputs> outputs_metadata(batch_size);
outputs_metadata.append_block("preds", output_records.m(), output_records.n());
ModelMetadata<SoAInputs, SoAOutputs> metadata(inputs_metadata, outputs_metadata);

// inference
model.forward(metadata);
```

Partners

Supported by the Eric & Wendy Schmidt Fund for Strategic Innovation (grant agreement SIF-2023-004)