

Experiences Deploying a Hybrid PVFinder Algorithm for Primary Vertex Reconstruction in LHCb's GPU-Resident HLT1

Fast Machine Learning for Science Conference 2025

Mohamed Elashri^{1*}, Simon Akar², Conor Henderson¹, Michael David Sokoloff¹

¹University of Cincinnati (US)

²LPCA - Université Clermont Auvergne (FR)

*Corresponding author

September 5, 2025

This work is supported by IRIS-HEP under NSF Cooperative Agreement PHY-2323298



What This Talk Is (and Is Not)

Deployment focus: inference-engine integration and real-time constraints

This talk is about:

- Making a hybrid DNN run inside LHCb's Trigger under hard real-time budgets
- Throughput impact and how to reach deployment-grade performance

This talk is not about:

- Architecture search or training details of PVFinder
- Losses, hyperparameters, or physics tuning of the model

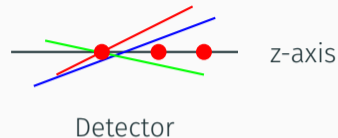
PVFinder Overview

Problem: Real-time primary vertex reconstruction at LHCb

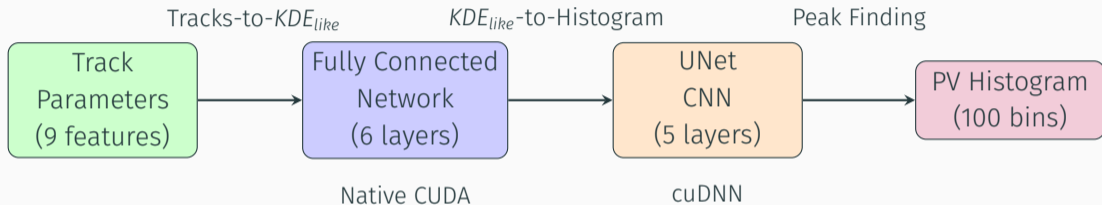
- Run 3: 5x luminosity increase \rightarrow 5.6 PVs/event
- Hardware trigger removed \rightarrow pure software trigger
- Need fast, accurate PV finding for 30 MHz data rate

Solution: Hybrid deep learning approach

- **Input:** Track parameters (9 features/track)
- **Output:** PV positions along z-axis
- **Method:** FC layers + CNN for pattern recognition + heuristic algorithm to extract pv positions from histogram output.



Architecture

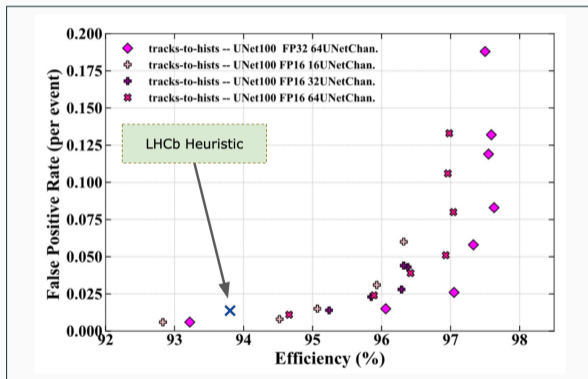


Key Innovation: End-to-end learning from tracks to vertex histograms

- **Spatial slicing:** 4000 bins → 40 intervals of 100 bins
- **Better performance:** Higher efficiency, lower false positive rate

Physics Performance

Performance



Magenta configuration (FP32 64-channel) selected for inference deployment

Key Results:

- **Efficiency:** >97% at low FP rates
- **False Positives:** 0.03/event (0.6%/PV)
- **FP16 compatibility:** Minimal performance loss

Advantages over heuristic methods:

- Better separation of nearby vertices
- Robust against varying pile-up conditions

ATLAS Adaptation:

- Adapted with good performance
- [More information](#)

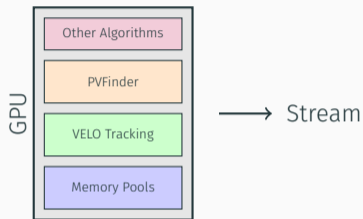
Allen Trigger Framework

Real-time constraints:

- **Throughput:** 30 MHz \rightarrow $O(2)$ kHz (HLT1)
- **Average Execution Time:** $<400 \mu\text{s}$ per event budget
- **GPU:** NVIDIA A5000 in production

Architecture requirements:

- Structure-of-Arrays (SoA) data layout
- Manual memory pool management
- Stream-based execution model
- Predictable, deterministic performance



Challenge: Integrate ML inference while maintaining Allen's constraints

Integration Constraints

Allen's rigid requirements:

- Fixed memory ownership
- No dynamic allocation at runtime
- One event = one GPU block
- No global synchronizations

Why this matters:

- Any violation breaks HLT1 stability
- Must respect existing data flow
- Cannot disrupt other algorithms

Current environment constraints:

- **GPU hardware:** NVIDIA A5000 specifications
- **Memory bandwidth:** Limited by current architecture
- **Compute resources:** Shared with other algorithms



Hybrid Implementation Challenge

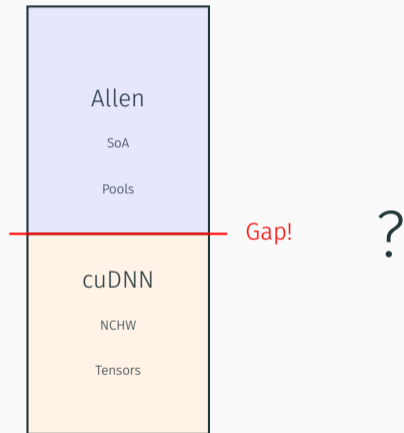
Two different worlds:

FC Stage (Native CUDA):

- Hand-written matrix operations
- Zero external dependencies
- Native Allen SoA support
- Direct memory pool integration

CNN Stage (cuDNN):

- Industry-standard deep neural network library
- Optimized convolution kernels
- Requires NCHW tensor format
- Different memory model



The Problem: How to bridge these different approaches?

Translation Layer Solution

Three-phase approach:

1. Prepare:

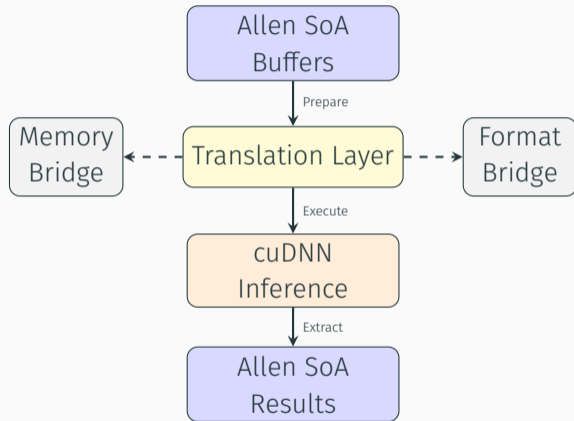
- Create tensor views on Allen buffers
- Pre-allocate workspace arena
- Map batch intervals

2. Execute:

- Run cuDNN on Allen stream
- Preserve event-parallel semantics
- No synchronization violations

3. Extract:

- Map outputs back to Allen buffers
- Minimal data movement
- Recycle staging buffers



Technical Implementation

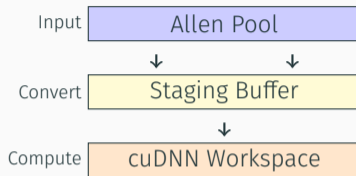
Key innovations:

- **View-based mapping:** No deep copies
- **Fused FC output:** Direct CNN layout
- **Single staging buffer:** For rare padding cases
- **Fixed workspace:** Pre-allocated, reused
- **Stream isolation:** All ops on Allen stream

Benefits:

- Isolates cuDNN from Allen codebase
- Preserves deterministic memory model

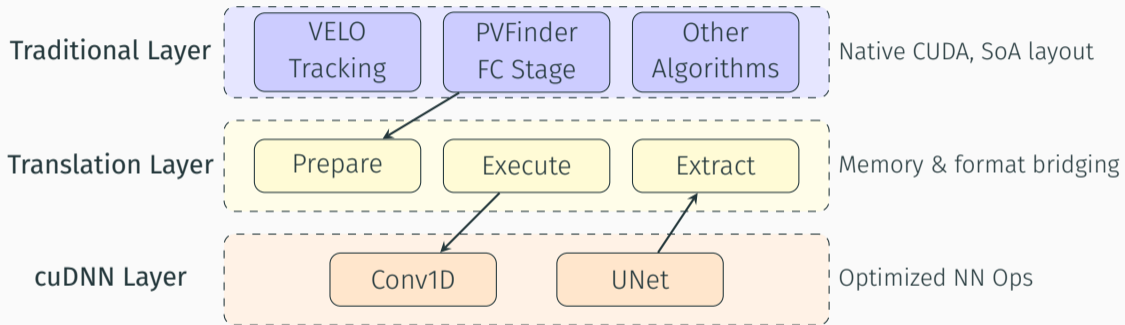
Memory flow:



Stream hygiene:

- All GPU work on Allen stream
- No default stream leakage
- Event-based synchronization only

Complete Architecture



Key principle: Each layer speaks its native language, translation layer handles all conversions

Standalone Performance

Standalone Performance (RTX 2080 Ti):

Configuration	Throughput
FC stage only	300 kHz
Full hybrid model	110 kHz

Translation layer breakdown:

- Prepare: 11% of layer overhead
- Execute: 79% of layer overhead
- Extract: 10% of layer overhead

PRELIMINARY RESULTS

These results are preliminary.
We anticipate an **order of magnitude** improvement with optimizations.

FC optimizations:

- Warp-level reductions
- Per-warp sub-histograms
- Optimized atomic operations
- Currently running on **CUDA cores**

Expected optimizations:

- **32 vs 64 UNet channels:** 4x speedup (quadratic scaling)
- **FP16 vs FP32:** Additional 2x speedup
- **Tensor cores:** Future utilization

Current Config
FP32 + 64 UNet
Channels
(CUDA cores only)

HLT1 Integration Performance

Relative Throughput Impact:

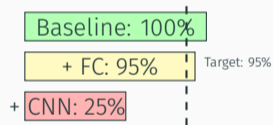
Configuration	Relative Throughput
Baseline HLT1	100%
+ FC stage	~95%
+ Full hybrid	~25%

Current constraints:

- CNN stage dominates runtime overhead
- Hardware: A5000 GPU limitations
- Target: maintain <5% regression at scale

Challenge analysis:

- Memory bandwidth saturation
- Cache interference with other kernels
- Algorithm consolidation required



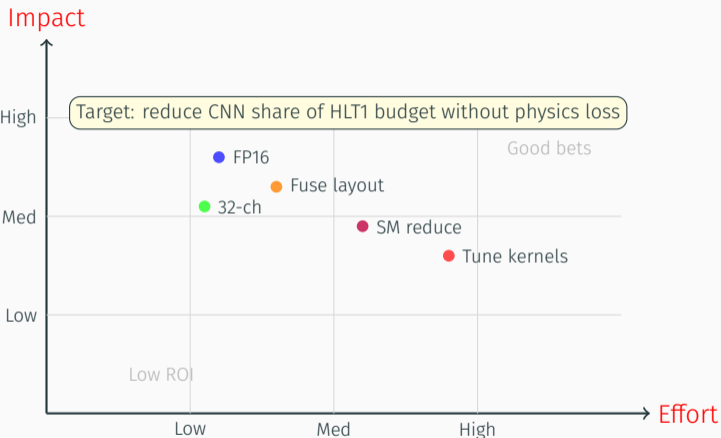
Performance Roadmap: High-Impact Next Steps

Numerics & Model

- FP16 end-to-end with tight validation
- 32-channel UNet; prune safe paths

Memory & Layout

- Keep $[B, C, L]$ contiguous; avoid transposes
- Reuse fixed workspace; coalesced access
- Fuse format-bridging where it removes traffic



Order-of-magnitude improvement expected

Abstraction Layer for LHCb ML: Status and Next Steps

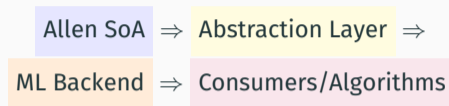
What we integrated

- CNN + FC run inside the GPU trigger via a small abstraction layer.
- Data stays in Allen buffers; shapes fixed up front; one GPU stream.
- Clear boundary between trigger code and the inference backend.

Optimization first

- Prioritize PVFinder: mixed precision, leaner network, less data movement.
- Evaluate on the integrated path; keep physics performance while reducing latency.

Flow (at a glance)



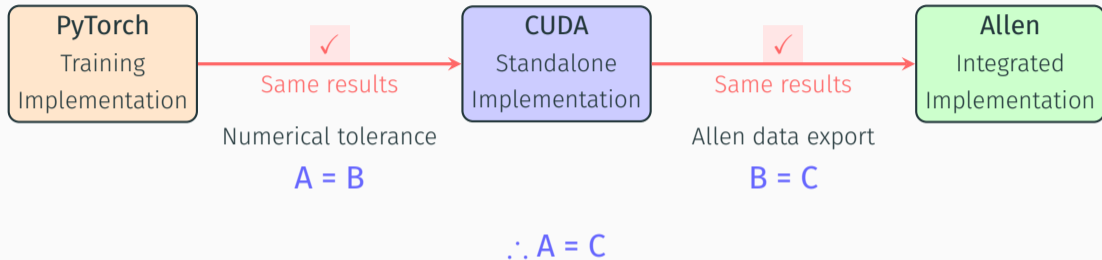
Standardize & open up (exploration)

- Publish a minimal contract for LHCb (shapes, memory ownership, stream rules) with a small adapter API.
- After optimization and review, open-source the layer and add generic backends (e.g., ONNX Runtime).

Backup Slides



Implementation Verification: Chain of Truth



Verification Strategy:

- Port PyTorch → CUDA standalone
- Validate with same datasets
- Export Allen inputs → CUDA
- Verify Allen implementation

Translation layer: abstraction and guarantees

What this layer is

- Zero-copy bridge: SoA buffers -> backend tensors.
- Deterministic: fixed shapes, fixed workspace, one stream.
- No runtime allocation, no device/global syncs.

```
# Pseudocode API (abstract)
type TLSpec   = {batch, channels, length,
                layout_in, layout_out}
type Stream   = opaque
type Workspace = {ptr, bytes}
type View     = {ptr, shape, strides}
type Handle   = opaque

fn TL.init(spec:TLSpec, s:Stream, ws:Workspace)
    -> TL
fn TL.execute(tl:TL, in:View, out:View, weights:
    Handle) -> Status
```

Translation layer: initialization (freeze-time)

Lifecycle, part 1

- Validate spec; record shapes and strides.
- Bind stream and workspace; freeze algorithm choice.
- Precompute descriptor state; nothing dynamic later.

```
# called once at setup
spec = {batch:B, channels:C, length:L,
        layout_in: "SoA->[B,C,L]",
        layout_out: "[B,1,L]"}
ws  = {ptr:pool_ptr, bytes:limit}
tl  = TL.init(spec, stream, ws)

# freeze internal mappings/algo ids
assert tl.fixed_shapes()
assert tl.workspace.bytes >= tl.required_ws()
```

Translation layer: per-event usage (single stream)

Lifecycle, part 2

- Producer writes input in agreed layout (no copies).
- Execute on the same stream; output is already consumable.

```
# per event on the same stream
in_view = {ptr:x_ptr, shape:[B,C,L], strides:sx}
out_view = {ptr:y_ptr, shape:[B,1,L], strides:sy}

ok = TL.execute(tl, in_view, out_view,
               weights_handle)
if not ok: bypass_and_continue()

# downstream reads out_view directly
consume(out_view)
```

Translation layer: cuDNN adapter

Bridge

- TL calls a backend interface.
- cuDNN implements it (1D as 2D, H=1).
- Fixed shapes: B=40, L=100, C in {32,64}.
- No allocs; single stream; fixed workspace.

```
# Backend interface
trait TLBackend {
  fn required_ws() -> Bytes
  fn bind(in_view, out_view) -> Bool
  fn forward(stream, ws, weights) -> Status
}

# Init (once)
tl.backend = Cudnn.make(spec)
tl.ws      = {ptr:pool_ptr, bytes:limit}
tl.stream  = allen_stream

# TL.execute(): binds + workspace check +
              forward
fn TL.execute(tl, in_view, out_view, weights):
  if !tl.backend.bind(in_view, out_view): return
    BYPASS
  if tl.backend.required_ws() > tl.ws.bytes: return
    BYPASS
  return tl.backend.forward(tl.stream, tl.ws, weights
    )
```