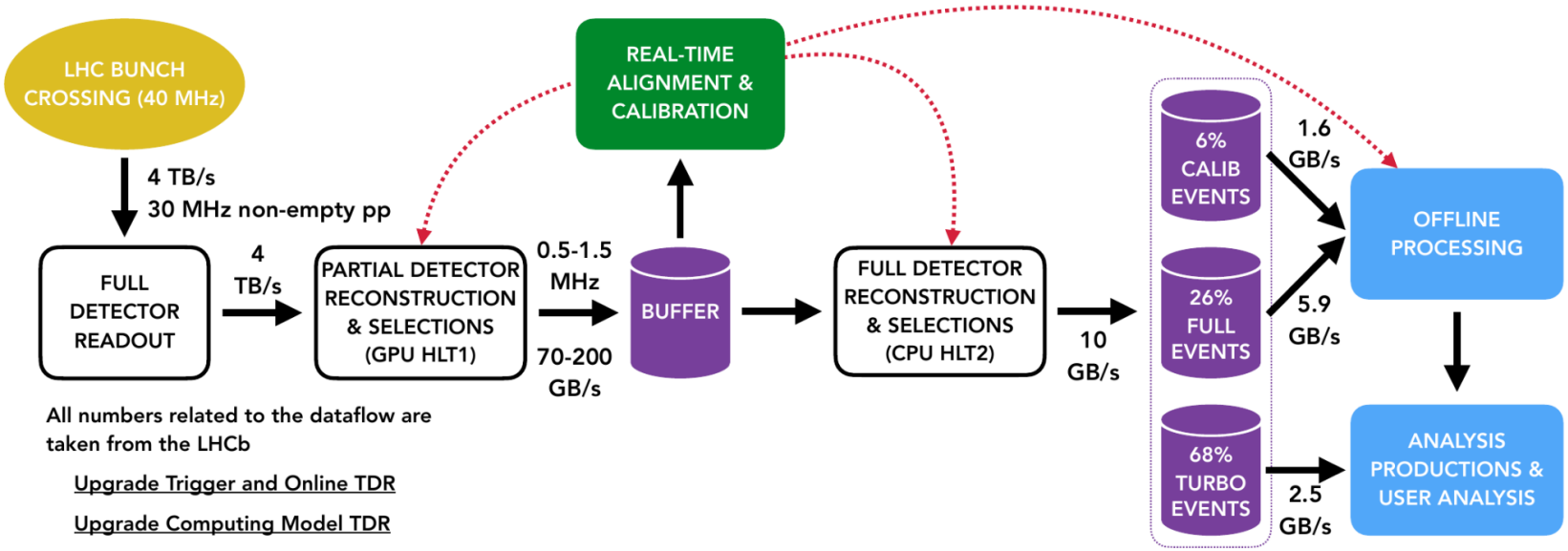
LHCb: Allen

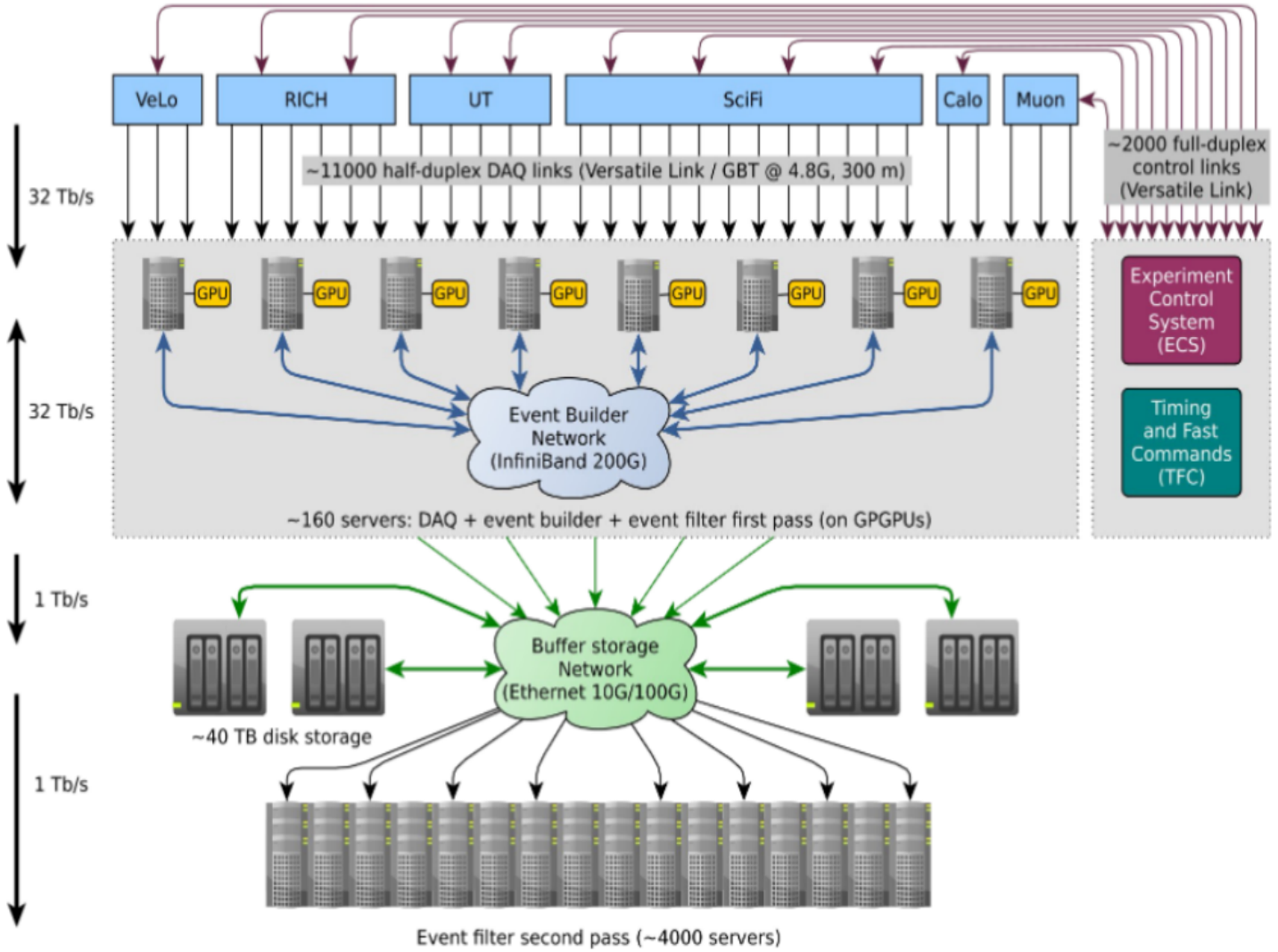
Arthur Hennequin – arthur.hennequin@cern.ch

LHCb Upgrade Dataflow (Run 3)



HLT1 challenge: reduce 4 TB/s to 100 GB/s in real-time with high physics efficiency

LHCb Upgrade Trigger and DAQ (Run 3)



Allen: Key Design Principles

Minimal Host Intervention

- GPUs share CPUs with Event Builder servers
- Host only coordinates, doesn't process data
- Raw data transferred on GPU
- Decisions sent back with minimal overhead
- Transient data stay on GPU

Maximized GPU Utilization

- Every algorithm hand-optimized in CUDA
- Tuned specifically for NVIDIA A5000 GPUs
- Throughput (events/s) as primary metric
- Focus on GPU occupancy and thread efficiency

Primary Constraint

Everything must run on GPU - Host steering kept to absolute minimum

Multi-Event Processing Architecture

Stream-Based Parallelism

- Each GPU divided into **streams**
- Each stream processes a **batch of events**
- Streams operate independently and concurrently
- Maximizes GPU resource utilization

Static Scheduling

- Reconstruction sequence configured in Python
- Algorithms topologically sorted at configuration time
- Data dependencies determine execution order
- No runtime scheduling overhead

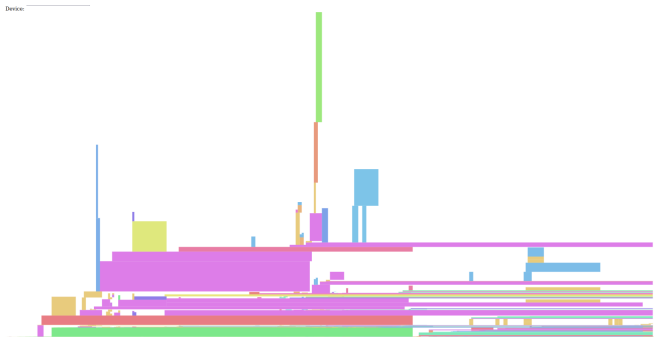
Memory Management Strategy

Custom Allocator Design

- GPU memory is a scarce resource (24 GB per A5000)
- Preallocated memory blob per stream (500-1000 MB)
- Only Transient Event Data, geometry / monitoring is shared between streams and managed separately.

Host-Managed Allocation

- All allocations managed on host side
- Separarate allocators for host and device memory
- Buffer lifetimes computed statically at configuration time, based on data dependencies



Cross-Platform Portability

The "Allen Way" of Writing Code

- Algorithms written in **backend-agnostic C++** that looks like CUDA
- Lightweight header files define CUDA keywords for other backends
- Block-stride loops in every kernel (on CPU, blockDim hardcoded to 1)
- Same source code runs on GPU and CPU

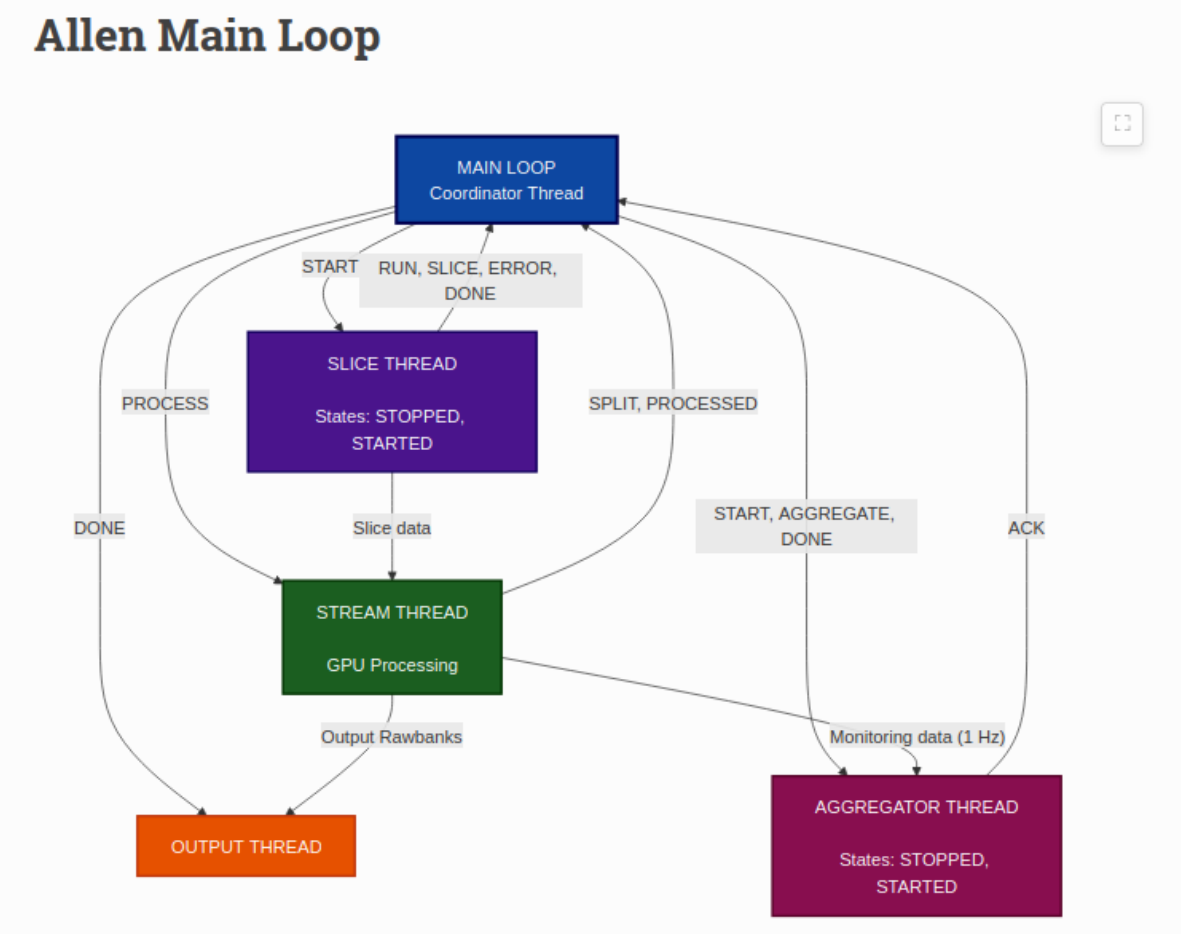
CPU Fallback for Simulation

- Must run anywhere for simulation workflows
- Speed not critical for simulation use cases
- Enables validation and testing without GPUs

Code Example

```
// Block-stride loop pattern
for (unsigned i=threadIdx.x; i<total_items; i+=blockDim.x) {
    // Process item i
}
```

Allen Threads

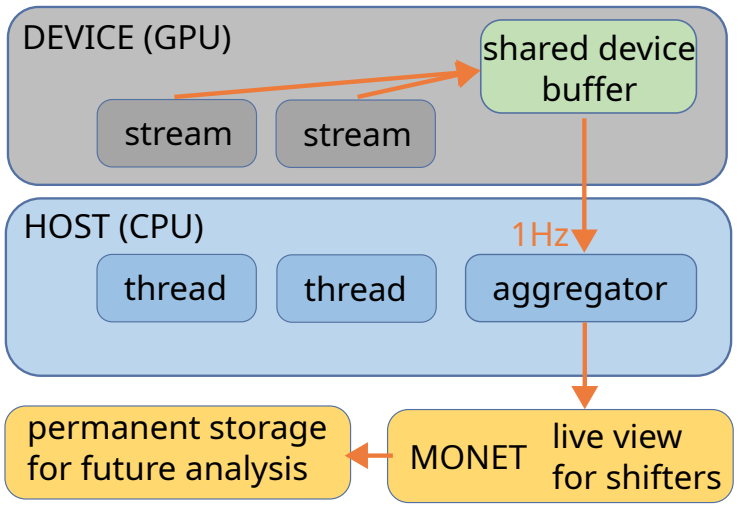


Real-Time Monitoring System

Critical for Trigger Understanding

- Only 1/30 of data saved - monitoring plots are essential to understand system performances
- Provides real-time system performance assessment: enables immediate detection of issues

- **Shared histograms** and counters: Aggregation across all streams on device with atomics
- Double buffering and **Low-frequency copying** to host for no throughput impact



Production Integration

Online Control System

- Steered by LHCb Online Control System
- Monitored and controlled like any detector system
- Handles run transitions, error recovery, configuration

Performance in Production

- Processes full 30 MHz input rate, with some margin
- 100+ algorithms in reconstruction sequence

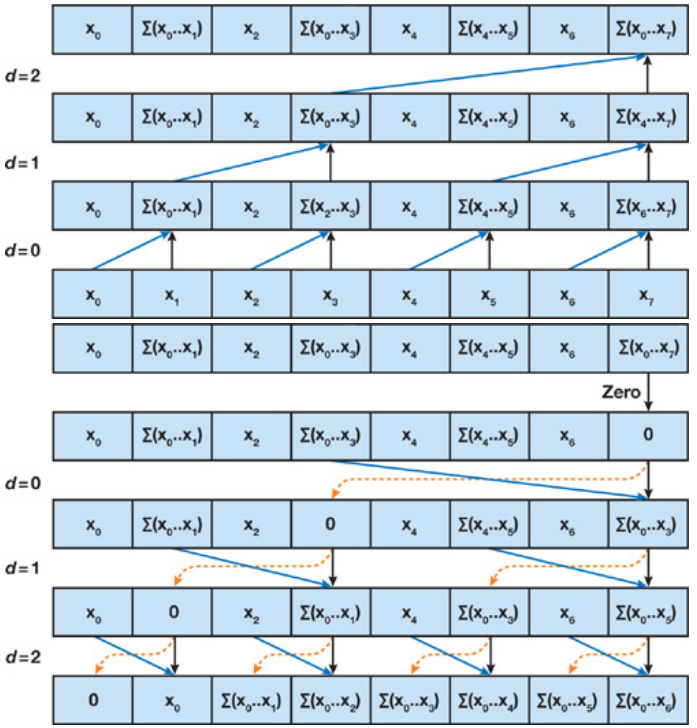
Algorithm Building Blocks: Prefix Sum

Use cases

- Variable-sized allocations (convert counts to offsets)
- Used everywhere in reconstruction:
  - $O(10^3)$ events per slice
  - $O(10^5)$ tracks per slice
  - $O(10^6)$ hits per slice

Implementation

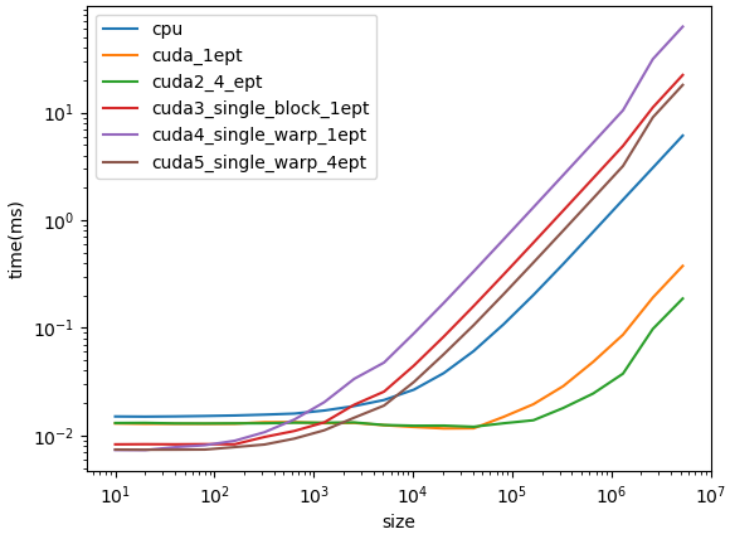
- Blelloch's scan algorithm
- Optimized for GPU memory hierarchy
- Avoids bank conflicts
- Up to 8% throughput gain compared to naive CPU prefix sum



Prefix Sum: Performance Optimization

Why GPU Beats CPU ?

- Common belief: prefix sum inherently sequential so is faster on CPU
- Reality: GPU faster for large arrays, and even small arrays if the data is already on device
- Avoids PCIe transfers
- Different algorithms for different sizes



Throughput comparison of different prefix sum algorithms

Implementation Choice

- Small arrays (< 512): Single block
- Large arrays: Hierarchical approach

Algorithm Building Blocks: Segmented Sort

The Challenge

- Sort many small segments independently
- Old approach: $O(N^2)$ per segment
- Wasteful: threads idle for small segments
- Multiple custom implementations

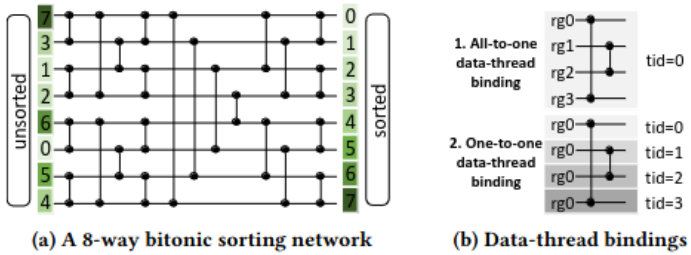
Optimized Solution

- Group segments by size
- Specialized kernels per size bucket
- Register sort for sizes < 512
- Global memory fallback for larger sizes
- Unified API for all use cases

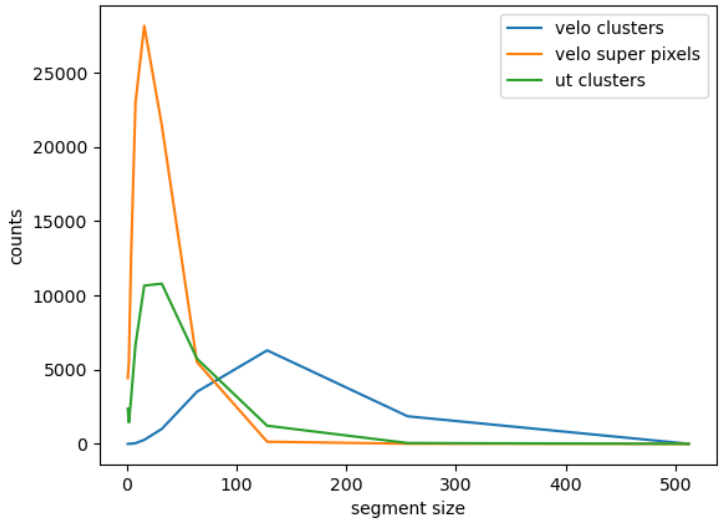
Segmented Sort: Implementation Details

Key Optimizations

- **Register sorts**: Data stays on-chip, uses warp shuffles
- **Size bins**: Power-of-2 buckets up to 512
- **Code generation**: Generate optimal sort for each size bin
- **No Dynamic parallelism**: Bins are dispatched manually with load balancing techniques, within a single mega-kernel (non-nvidia backends compatibility)



Register sort using sorting networks



Segment size distribution (log scale)

User-Friendly Building Blocks

Prefix Sum API

```
#include <PrefixSum.cuh>

// In-place prefix sum
PrefixSum::prefix_sum<dev_offsets_t>(
  *this, arguments, context);

// With host total
PrefixSum::prefix_sum<
  dev_offsets_t, host_total_t>(
  *this, arguments, context);
```

Segmented Sort API

```
#include <SegSort.h>

SegSort::segsort<int64_t>(
  *this, arguments, context,
  data<dev_keys_t>(arguments),
  data<dev_offsets_t>(arguments),
  size<dev_offsets_t>(arguments)-1,
  data<dev_permutations_t>(arguments));
```

Impact on Development

- **Hide complexity**: Developers focus on physics
- **Consistent APIs**: Easy to learn and use
- **Performance by default**: Optimized implementations
- **Maintainable**: Single implementation to optimize

Lessons Learned from past Allen Development

Technical Insights

- Memory access patterns critical
- Specialized kernels usually outperform general ones
- Register usage requires careful coding and compiler understanding
- Monitoring can't be an afterthought

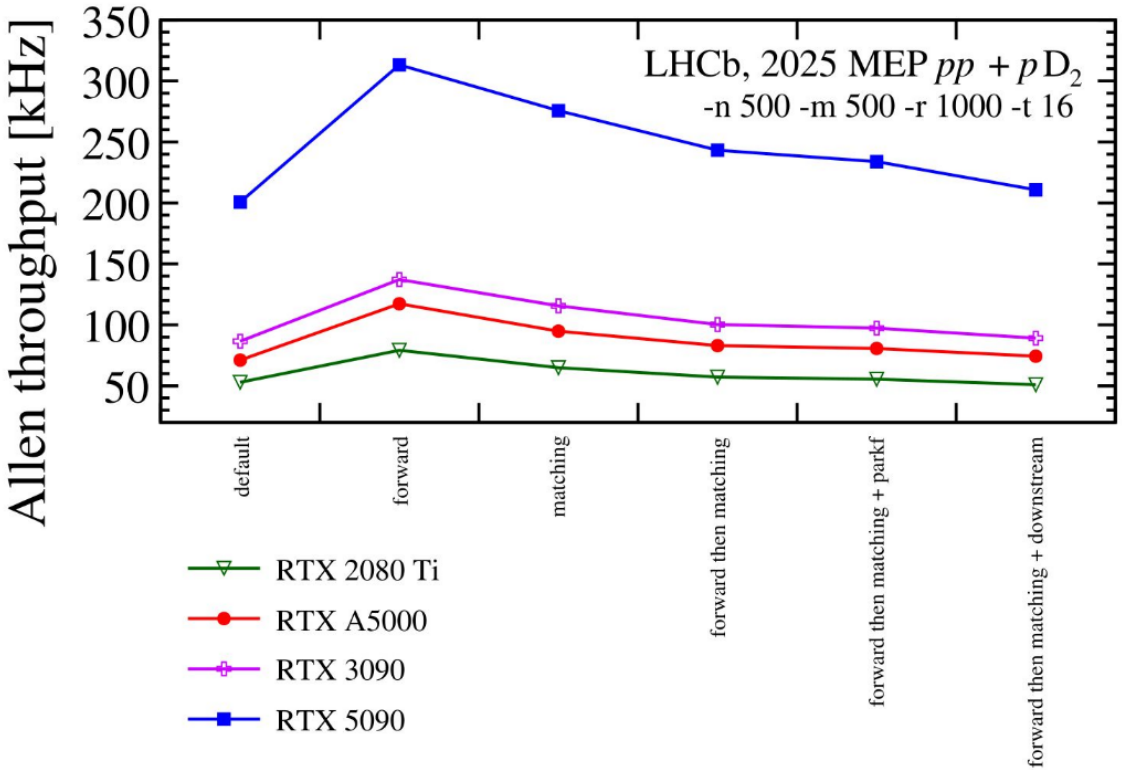
Development Practices

- Writing performant algorithms need deep GPU architecture understanding
- Profiling tools are essential
- Continuous integration critical
- Framework design affects your algorithms
- Hard to have one-size-fits-all solutions

Key Takeaway

Writing real-time 30 MHz analysis software requires both GPU expertise and careful attention to every detail of system design and implementation.

Allen Throughput Today



O(500) RTX A5000 GPUs enabling 30MHz HLT1

Ongoing developments

Framework

- AllenCore: split the non-LHCb specific part of Allen to its own repository for use in other experiments (See Gonzalo's talk next)
- Non-NVIDIA GPU backends: add support for AMD, SYCL, ...
- Rework memory manager: allow hierarchical data structures as algorithm parameters (eg. structs), implicit data dependencies

Physics

- Port of RICH reconstruction to GPU
- Evaluation of HLT1-HLT2 differences (eg. PV finding)
- Forward tracking / hybrid seeding tuning studies
- Parametrized Kalman filter improvements

Can we get to HLT2 efficiency levels ?

Conclusion

Allen: Real-Time GPU Processing for LHCb HLT1

- **Successfully deployed**: Processing 30 MHz in production
- **Innovative architecture**: Multi-event GPU framework
- **Performance driven**: Every optimization matters at scale
- **Proven reliability**: Mission critical component of LHCb data acquisition

Looking Ahead

- Framework convergence with Gaudi (HLT1 / HLT2)
- Multi-vendor GPU support (AMD, SYCL, ...)
- Enhanced physics performance
- Preparation for upgrade II

Thank You