

Scheduling for Next Generation Triggers

Eric Cano¹, Mateusz Fila¹, Attila Krasznahorkay²

¹CERN, ²University of Massachusetts

CHEP 2026, 28.05.2026

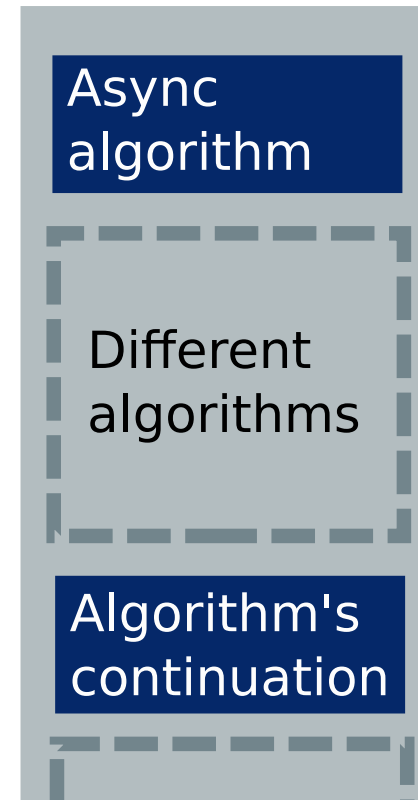


NexTGen
Next Generation Triggers

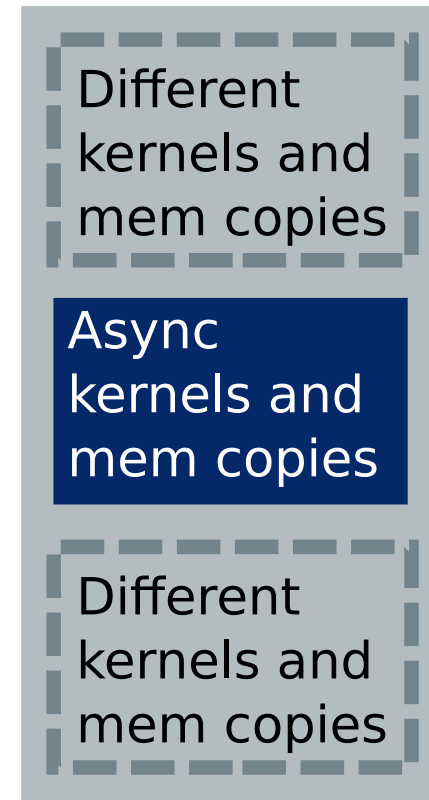
Async scheduling in frameworks

- High energy physics experiments are increasingly adopting heterogeneous computing and remote inference, introducing inherently asynchronous workloads.
- Asynchronous execution separates task progress from completion, enabling concurrency rather than blocking.
- Frameworks such as Athena and CMSSW already support asynchronous scheduling.
 - Current solutions are framework-specific and monolithic, making integration of external asynchronous libraries difficult.

CPU thread



GPU stream



External libraries with async operations



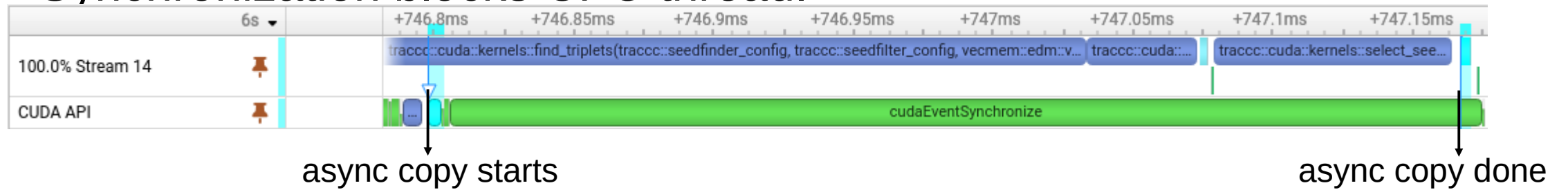
NexTGen
Next Generation Triggers

traccc

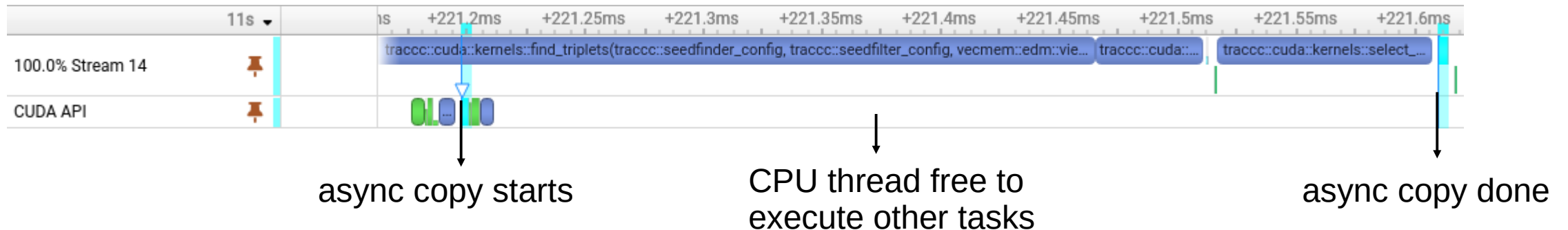
- **traccc** is a GPU tracking project (R&D line of ACTS).
- The project is meant to be experiment agnostic.
 - Cannot depend on a specific framework.
 - No schedulers or advanced scheduling logic internally.
- Prime candidate for testing different async models.

traccc synchronizations

- Multiple traccc functions require synchronizing memory copies between device and host to allocate new buffers. Single-threaded illustration
- Synchronization blocks CPU thread.



- Asynchronous execution frees the CPU to execute other tasks.



Async models considered – acquire & produce

- Implemented in CMSSW.
- Phase-split synchronization:
 - Asynchronous functions split into pre- and post-synchronization functions.
 - Supports any type of external work.
 - Framework provides a callback to signal that external work completed and `produce()` function can be called.
- Framework specific, non-trivial to handle variable number of synchronizations.

```
// toolkit
void Clustering::pre_sync(...){
    ...
    cudaMemcpyAsync(..., stream);
    ...
}
void Clustering::post_sync(...);

// application
void ClusteringModule::acquire(...){
    // Get input data from framework
    // launch async operation
    clustering.pre_sync(...);
    // Use appropriate signaling to
    // callback the framework
}

void ClusteringModule::produce(...){
    // continue with results
    clustering.post_sync(...);
    // register produced data with framework
}
```

Async models considered – fibers

- Fibers AKA stackful coroutines
 - Allow function suspension and resumption (be careful, resuming on different thread invalidates `thread_local` variables).
- Libraries such as `Boost.Fiber`, `TBB`.
 - `Boost.Fiber` currently used in Gaudi and Athena.
 - `TBB` is already widely used in many HEP projects for multi-threading.
- Suspending function can be called when run by corresponding execution context (`Boost.Fiber` manager, `tbb::task_arena`).
- Small changes in existing code.

```
// toolkit
Clusters clustering(...) {
    ...
    cudaMemcpyAsync(..., stream);
    ...
    // suspend until work in stream is done
    cudaStreamSynchronize(stream);
    boost::fibers::cuda::wait_for_all(stream);
    ...
    return clusters;
}

// application
StatusCode algorithm::execute(...) {
    Clusters clusters = clustering(...);
    Seeds seeds = seeding(clusters, ...);
    ...
    return StatusCode::Success;
}

boost::fibers::fiber(
    boost::fibers::launch::post, alg.execute);
```

Async models considered – beyond fibers

- **C++20 coroutines** (stackless coroutines):
 - Allow suspension and resumption (again, be careful with `thread_local` variables)
 - Only basic building blocks standardized, require 3rd party library.
- `std::execution` AKA senders/receivers (P2300):
 - Supposed to be the standard model for C++26.
 - Basic abstraction is a sender, but also comes with compatible coroutine type.
- Both require rewriting “async functions” to coroutines/senders.
- Both allow to separate execution and logic.
- Depending on specific coroutine library, bridges to senders or other coroutine libraries may be possible.

// with C++26 std::execution

// toolkit

```
std::execution::task<Clusters> clustering(...) {  
    ...  
    cudaMemcpyAsync(..., stream);  
    ...  
    co_await CustomWaitFor(stream);  
    ...  
    co_return clusters;  
}
```

// application

```
std::execution::task<StatusCode> algorithm::execute(...) {  
    Clusters clusters = co_await clustering(...);  
    Seeds seeds = co_await seeding(clusters, ...);  
    ...  
    co_return StatusCode::Success;  
}
```

```
scope.spawn(std::execution::starts_on(  
    thread_pool_adapter, alg.execute(...)));
```


Porting traccc

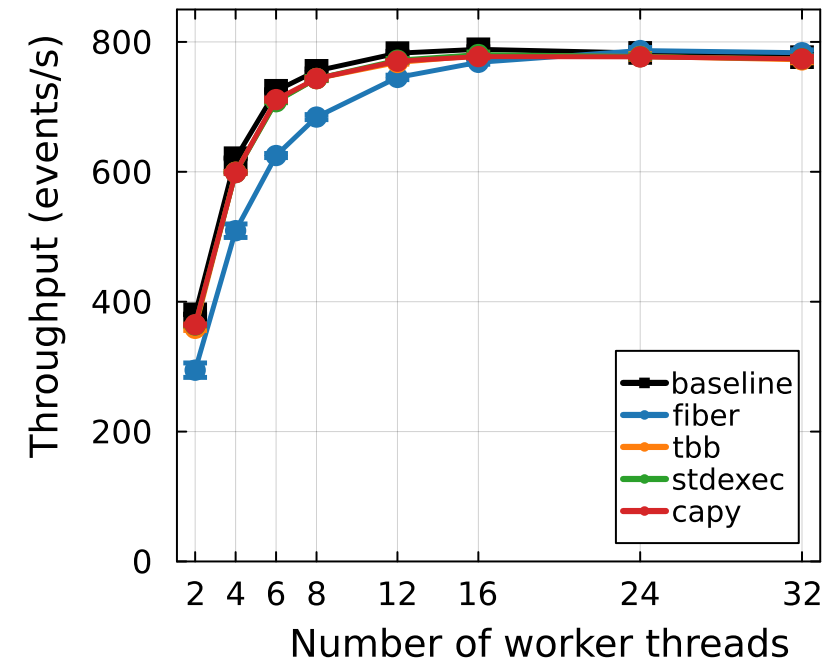
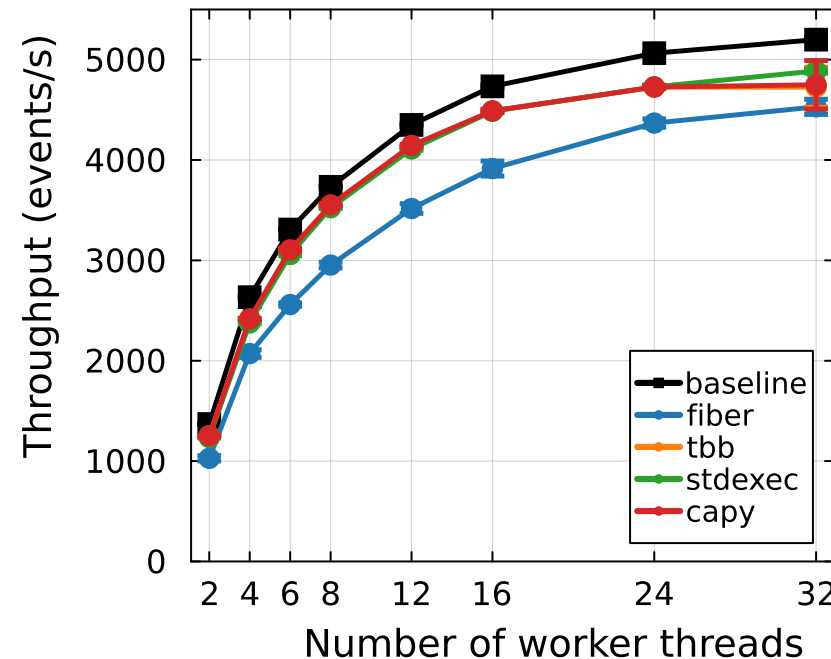
- CUDA flavour of traccc clustering and seeding stages were ported to compare different async libraries:
 - Boost.Fiber
 - TBB suspension
 - [NVIDIA/stdexec](#) – reference implementation for C++26 `std::execution`
 - [Boost.Capy](#) – reference implementation for pure coroutine-based I/O library proposed for C++29 ([P4003](#))

All async models perform similarly – almost

traccc with polling, seeding only, Nvidia L40S

ttbar_mu20

ttbar_mu200

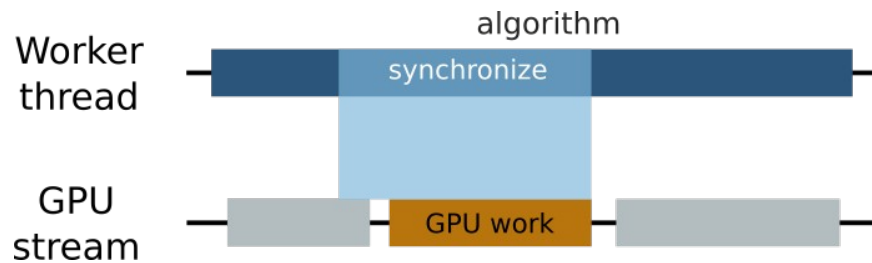


- fiber – Boost.Fiber with work sharing.
- tbb – tasks submitted to a task arena, native TBB suspension.
- stdexec – senders executed on a TBB task arena.
- capy – Boost.Capy coroutines executed on a TBB task arena.
- **TBB** is not only already used by many HEP libraries but also **offers more performant suspension than Boost.Fiber**.
- TBB, coroutines, senders/receivers have similar performance, **differentiation should come from ergonomics and capabilities**.

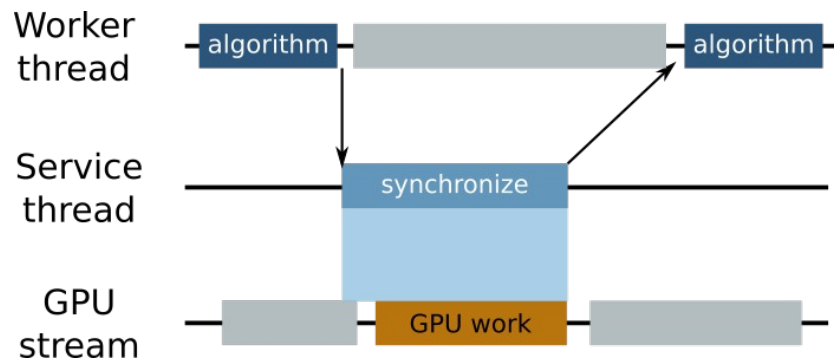
- CPU used only to drive the GPU computations.
- CUDA events and service threads configured to sleep.
- Expecting decrease of throughput since no CPU-work is scheduled in parallel.

Resumption handler strategies

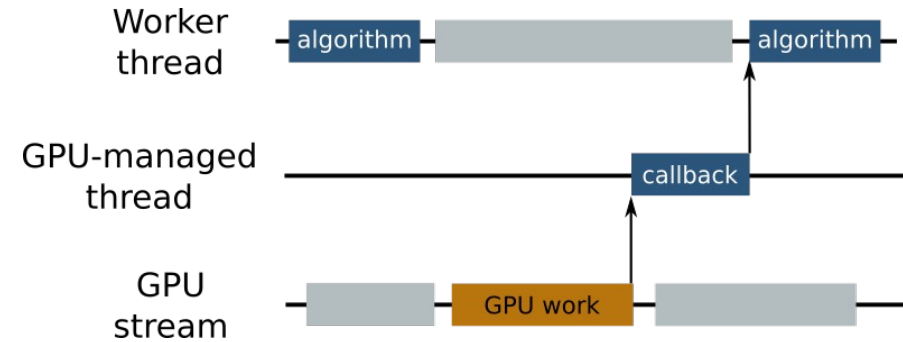
- Mechanism used for resumption may affect performance too.



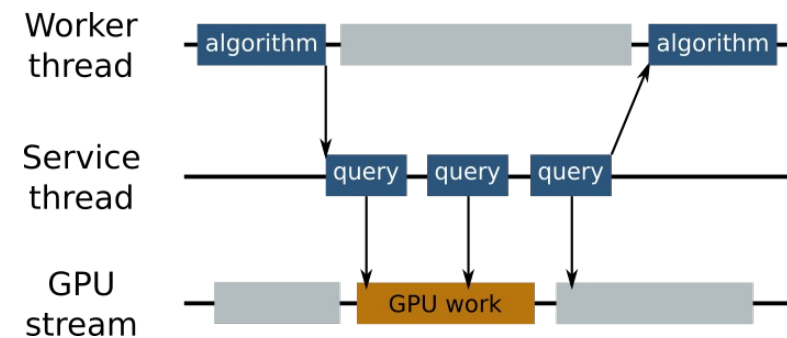
sync-event – synchronize on the current thread.



defer-sync-event – synchronize on a separate thread then request resumption.



callback – register a callback requesting resumption.

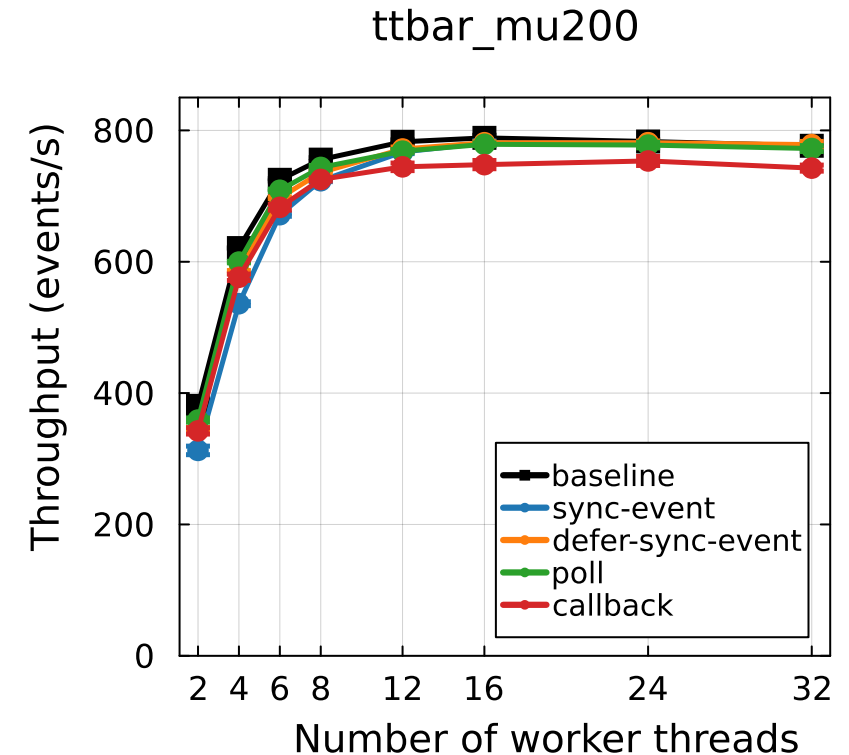
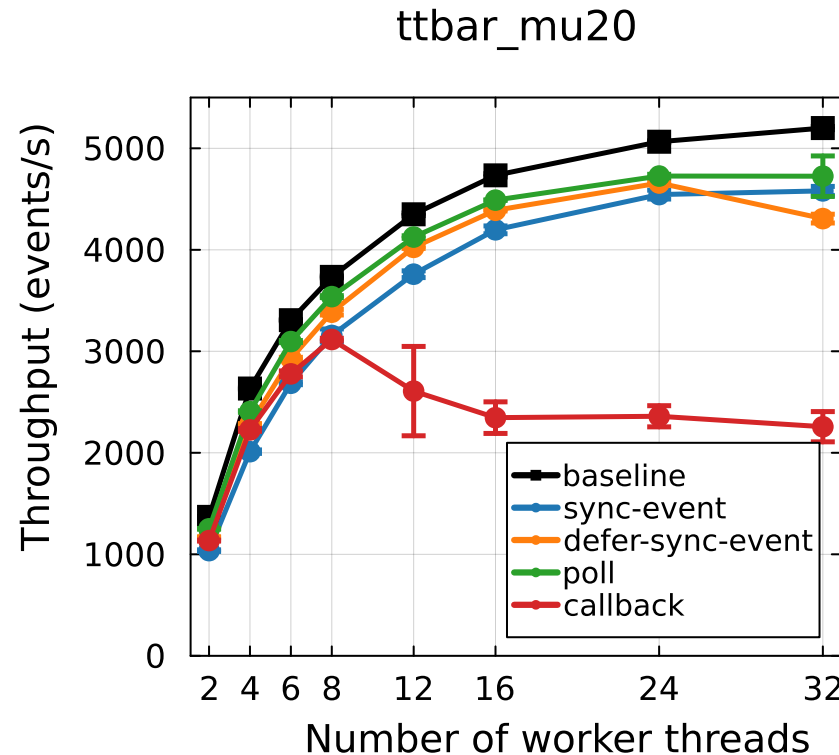


poll – query events in a loop on a separate thread, request resumption when done.

Not all handlers are created equal

- All the handlers can be implemented with any async model.
- For CUDA, `callback` is easy to implement but doesn't scale as good as other handlers.
- Impact depends on the workload.
- Use handler appropriate for your application. **Polling and deferred synchronization are good alternatives for multi-threaded jobs.**

traccc with TBB suspension, seeding only, Nvidia L40S



- CPU used only to drive the GPU computations.
- CUDA events and service threads configured to sleep.
- Expecting decrease of throughput since no CPU-work is scheduled in parallel.

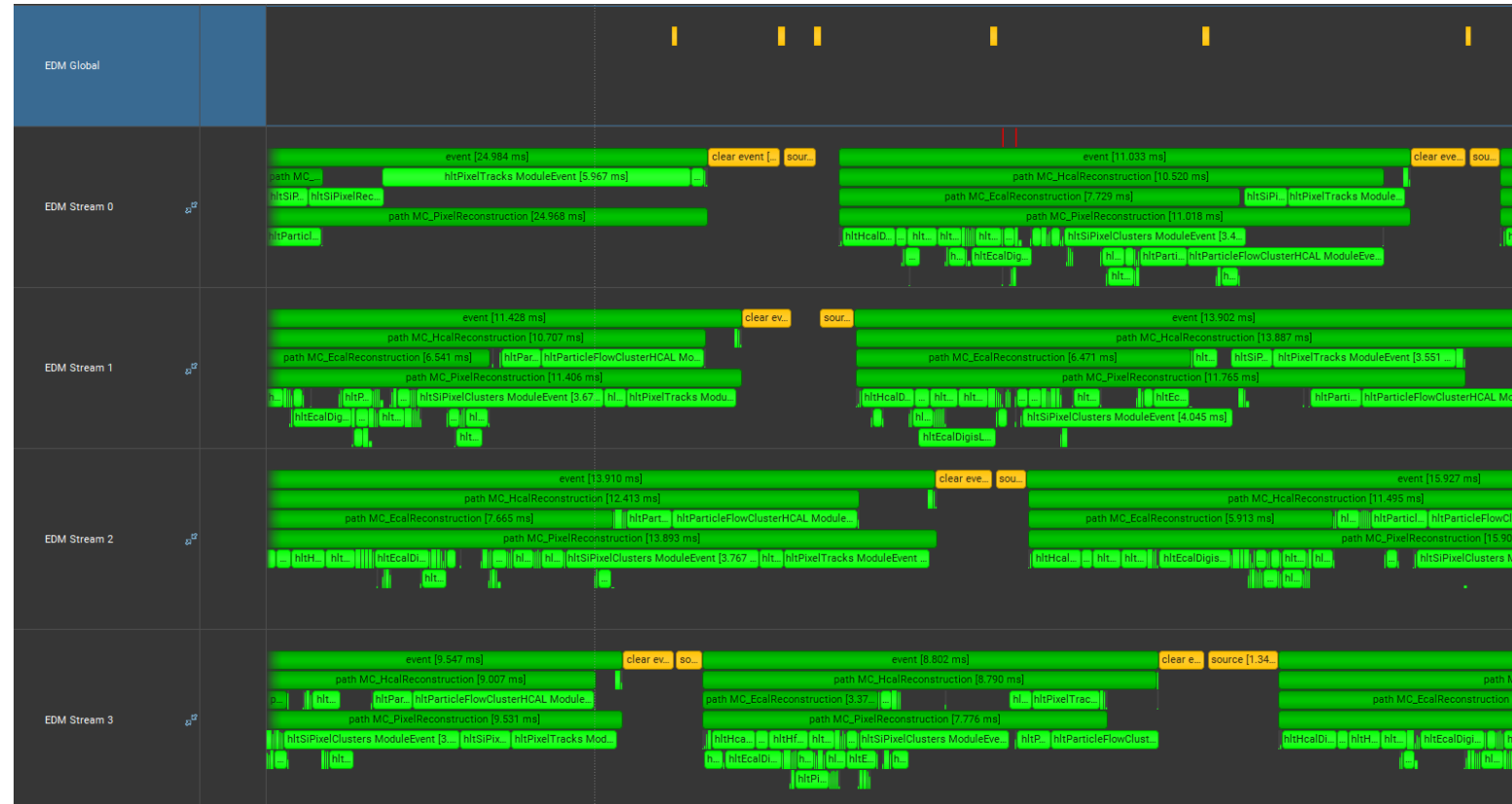
Instrumentation in frameworks



NexTGen
Next Generation Triggers

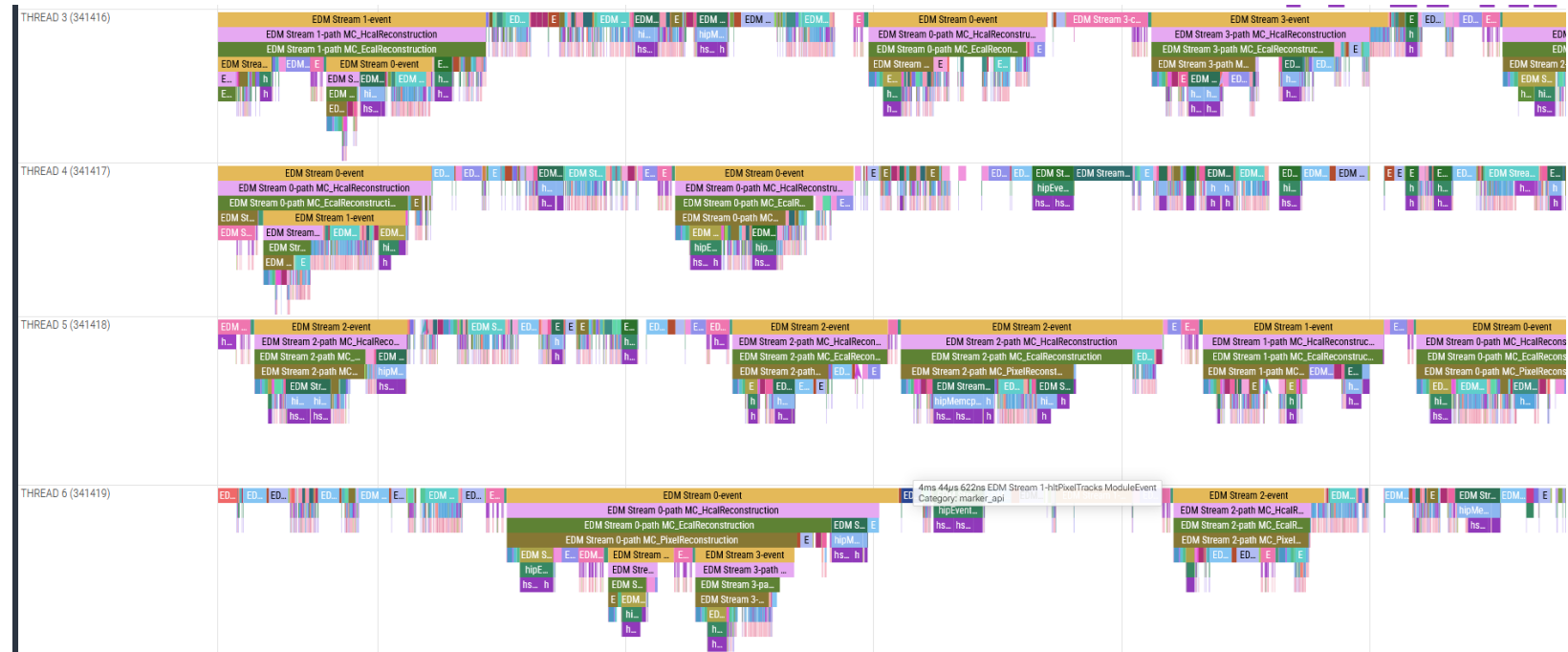
Profiling with Nvidia

- Instrumentation is essential for making framework execution visible in performance profilers, enabling the identification of bottlenecks and a better understanding of scheduling behavior.
- In CMSSW, the instrumentation was extended to annotate additional framework activity for CUDA profilers.



Profiling with AMD

- The same infrastructure was reused to add annotations for AMD profilers.
- Support for the Intel VTune profiler could also be added in the future.



Summary

- Investigating design and integration of external asynchronous libraries with HEP event-processing application frameworks.
- Ported traccc GPU reconstruction to different async models:
 - The same performance with C++20 coroutines, C++26 senders/receivers and TBB suspension.
 - C++20 coroutines and senders/receivers allow for decoupling work logic and scheduling logic.
 - CUDA callback-based handlers scale poorly for multi-threaded jobs. Event polling and deferred synchronization are both viable alternatives.
- **TBB suspension is a solid mid-term solution. In the long term, coroutines and senders/receivers appear to be the desired models** despite the higher investment required, due to their stronger integration model and their expected wider adoption in the future within the C++ community.



This work has been funded by the Eric & Wendy Schmidt Fund for Strategic Innovation through the CERN Next Generation Triggers project under grant agreement number SIF-2023-004.



NexTGen
Next Generation Triggers

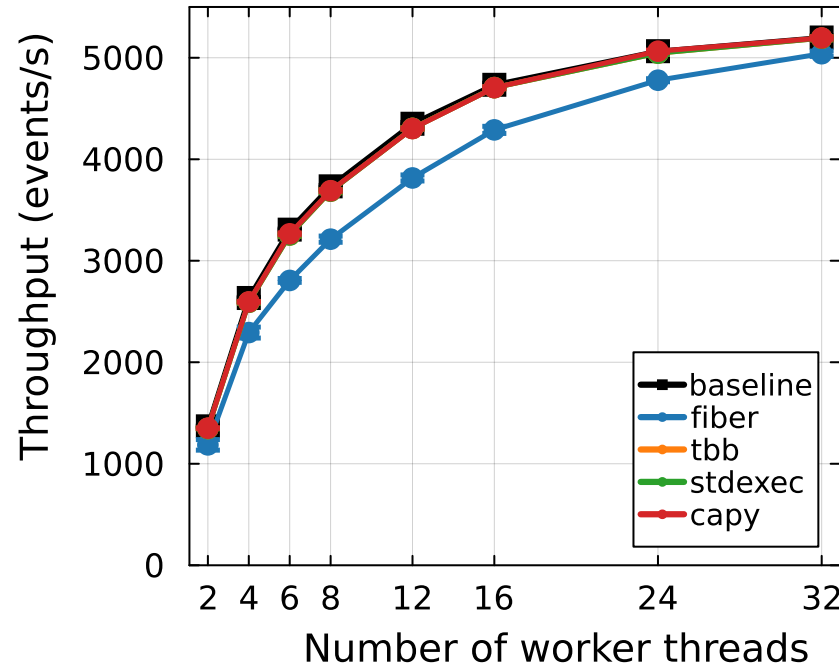
Async models (spinning)

traccc with polling, seeding only, Nvidia L40S

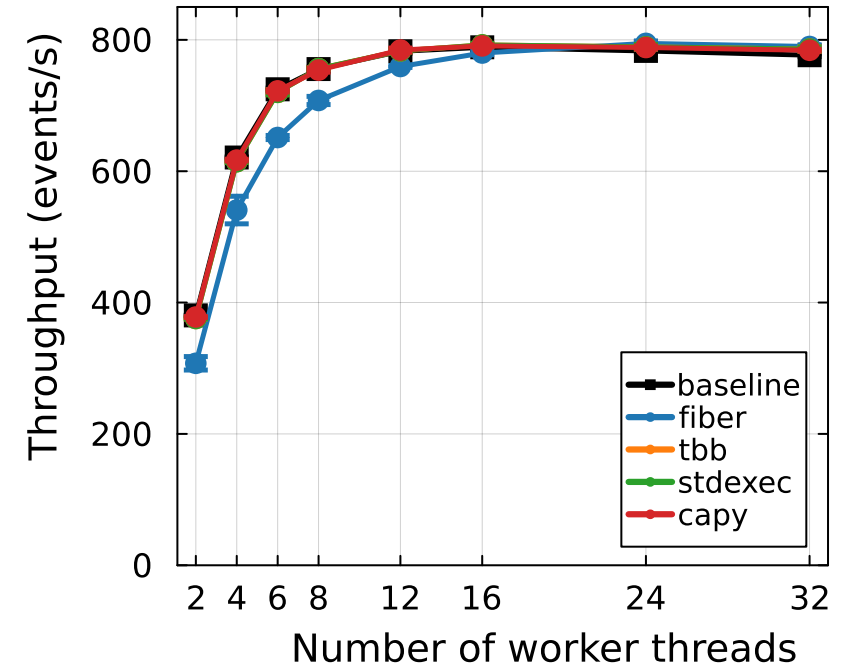
- CUDA events and service threads configured to spin to reduce latency of resumption at the cost of increased CPU-utilization.
- Expecting the same performance as synchronous baseline.
- All the async models can deliver performance of synchronous baseline at the cost of extra CPU-utilization.

Only Boost.Fiber has a lower performance ceiling.

ttbar_mu20



ttbar_mu200

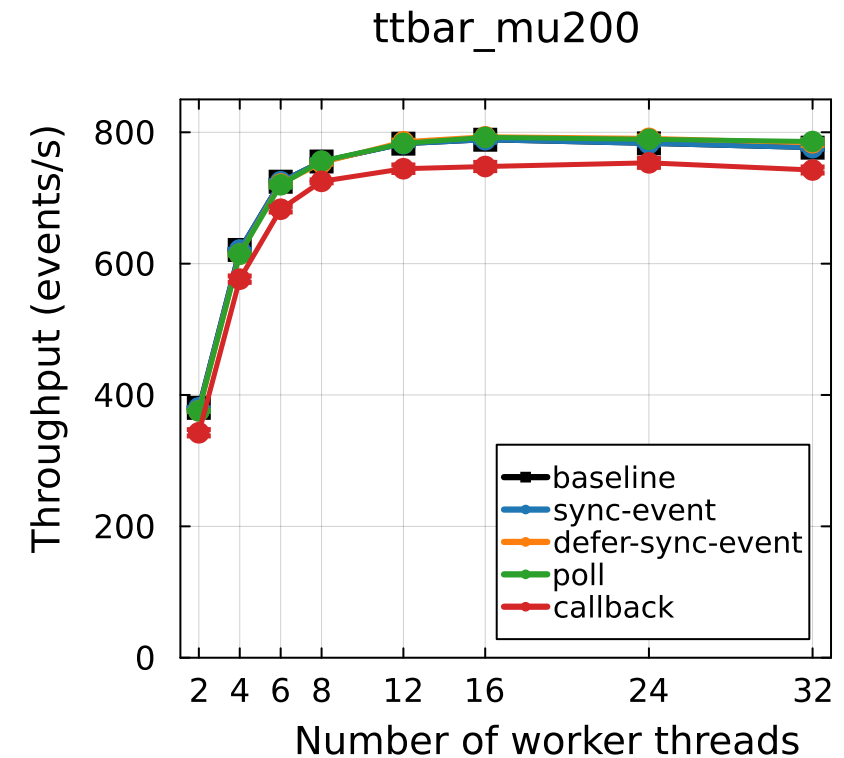
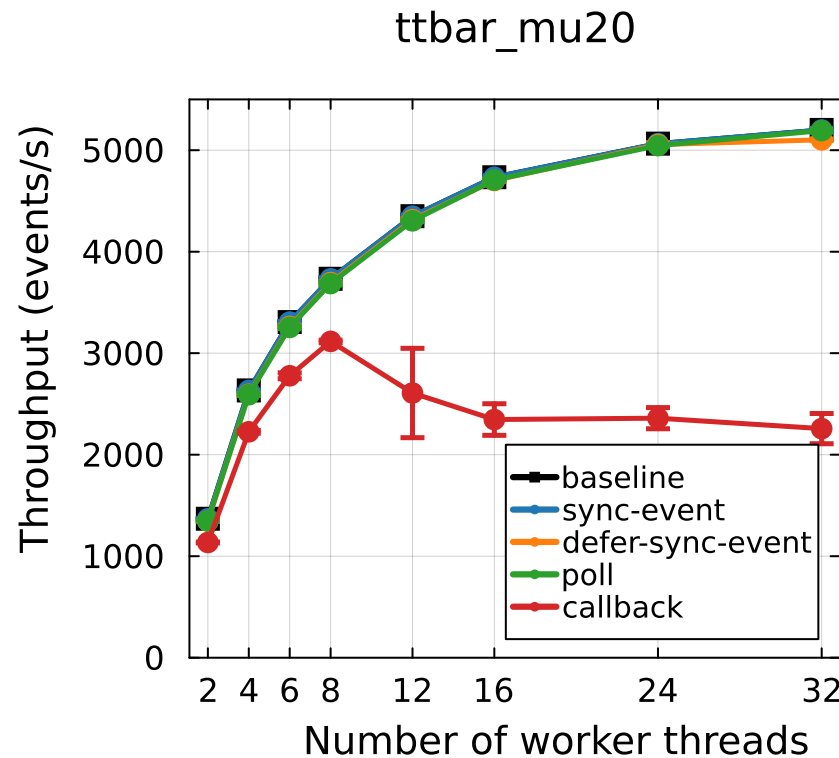


- CPU used only to drive the GPU computations.

Handlers (spinning)

traccc with TBB suspension, seeding only, Nvidia L40S

- CUDA events and service threads configured to spin to reduce latency of resumption at the cost of increased CPU-utilization.
- Expecting the same performance as synchronous baseline.
- All the handlers can deliver performance of synchronous baseline at the cost of extra CPU-utilization. **Only CUDA-callback does not-scale in this setup.**



- CPU used only to drive the GPU computations.

Framework hooks

- Task lifecycle (start, suspension, resumption, finish) should be communicated to a framework to enable instrumentation or influence scheduling decisions.
- Boost.Fiber and TBB do not expose suspension/resumption state, and only allow notifications if explicitly injected into task code.
 - This makes them less suitable for clean, standalone instrumentation in large frameworks.
- Sender/receiver and C++20 coroutine models separate work definition from scheduling concerns.
 - Sender schedulers describe how and where the work is executed. A framework specific scheduler be used to notify framework components about task lifecycle.
- **Coroutines and senders/receivers offer cleaner integration model with frameworks than fibers.**

