Fermilab U.S. DEPARTMENT OF ENERGY Office of Science

Performance of Heterogeneous Algorithm Scheduling in CMSSW

Andrea Bocci[1], Christopher Jones[2], **Matti Kortelainen[2]** for the CMS collaboration

[1]CERN [2]FNAL

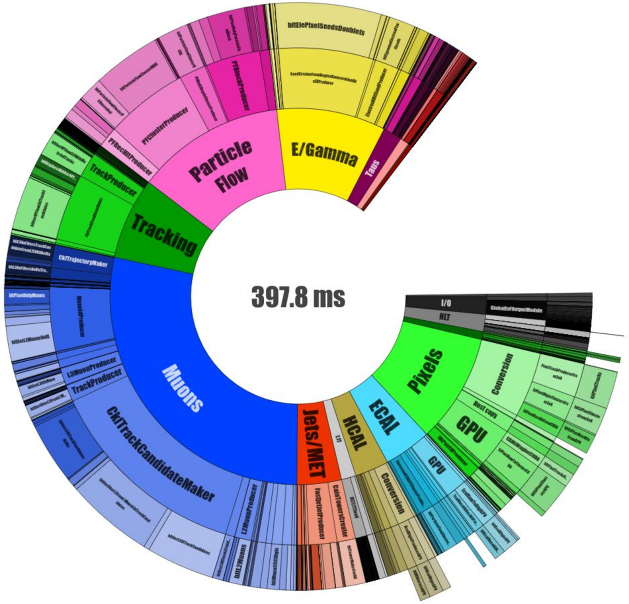
CHEP 2023 9 May 2023

Introduction

- CMS' data processing software framework, CMSSW, has a generic mechanism to interact with any work external to the framework
  - Allows the framework to utilize the CPU thread for other work
- CMSSW has a sophisticated pattern for framework modules to interact with CUDA
  - Was presented in CHEP 2019 (EPJ Web. Conf 245, 05009 (2020))
  - Is used in production in CMS' High Level Trigger (HLT) since 2022
  - CMS is in process of moving from CUDA to Alpaka (A. Bocci today Track 2 17:00)
    - Similar synchronization model underneath
- In this presentation we take a close look on the benefits of this pattern using actual HLT application that was used in 2022 data taking
- I'll start with a simplified setup and gradually add improvements towards the production setup in CMSSW

GPU reconstruction in CMS HLT 2022

- The HLT menu has total of ~4400 modules
- Offloaded parts
  - Pixel detector reconstruction: from RAW data unpacking up to tracks and vertices (11 modules)
  - ECAL local reconstruction (4 modules)
  - HCAL local reconstruction (3 modules)
- 57 unique kernels, ranging from 2 µs to 7 ms in these events
- Memory pool to amortize cost of raw memory allocations and provide asynchronous allocation interface in CUDA stream order
- All offloaded modules have CPU versions that are used for reference measurement
- More information were in [G. Parida's talk earlier in this session]()

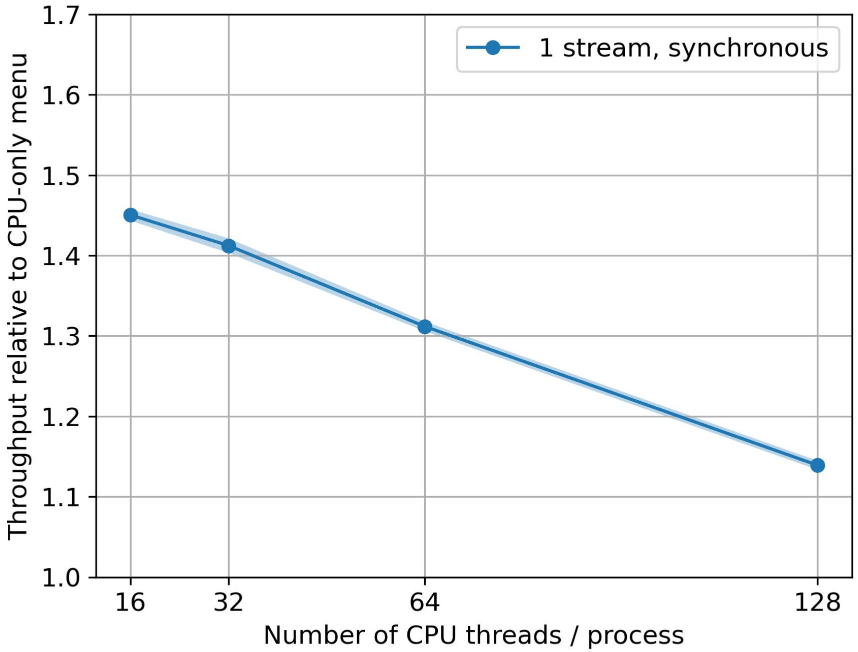


Measurements

- Use events triggered with CMS Level 1 trigger with average pileup of 65
- Measurements done on a machine like the production HLT nodes
  2x AMD EPYC 7763 (Milan) CPUs + 2x NVIDIA Tesla T4 GPUs
  - 2 sockets x 64 CPU cores / socket x 2 threads / core = 256 hardware threads on CPU
  - Aggregated throughput of N processes x M threads/process to have total of 256 threads
    - Take average of 4 executions
  - Number of concurrent events 3/4 of number of CPU threads to conserve GPU memory
    - No impact in event processing throughput
  - Measurements start at 16 CPU threads/process to fit in the 16 GB memory of T4 GPU
- Report event processing throughput relative to CPU-only menu

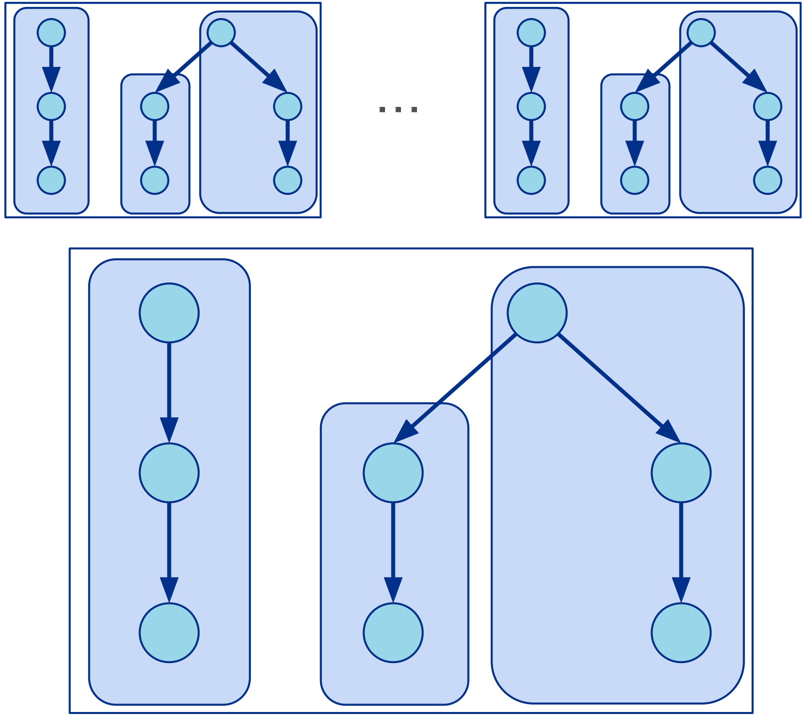
Simple starting point

- Each CUDA-using module launches their CUDA work by directly interacting with the CUDA API
- All these modules launch their work into the same CUDA stream
  - Mimics the behavior of the default CUDA stream
- Every CUDA-using module does a blocking synchronization
  - `cudaStreamSynchronize()`
- 15-45 % improvement compared to CPU-only



Add multiple CUDA streams

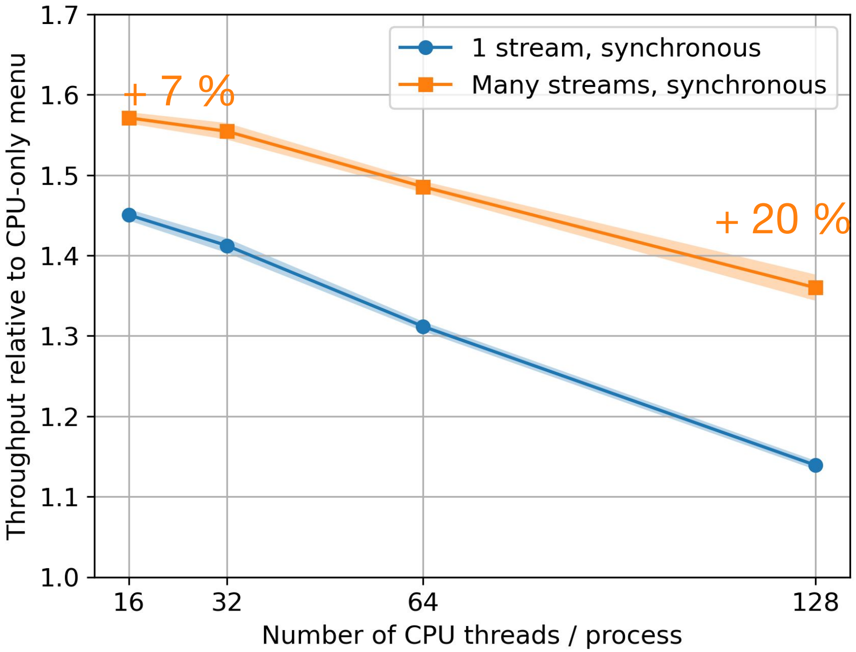
- Each non-branching chain of modules within an event uses a separate CUDA stream
  - Each concurrent event has its own chains
- Every CUDA-using module still does a blocking synchronization
  - Tested `cudaDeviceSchedule{Auto, Spin, Yield, BlockingSync}`, all gave practically the same performance
    - Reporting `cudaDeviceScheduleAuto`



Example module chains where 3 CUDA streams are used

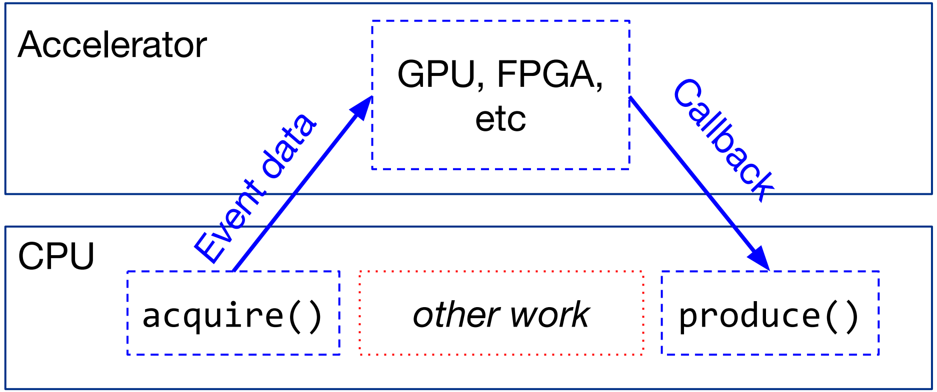
Add multiple CUDA streams

- Each non-branching chain of modules within an event uses a separate CUDA stream
  - Each concurrent event has its own chains
- Every CUDA-using module still does a blocking synchronization
  - Tested `cudaDeviceSchedule{Auto, Spin, Yield, BlockingSync}`, all gave practically the same performance
    - Reporting `cudaDeviceScheduleAuto`



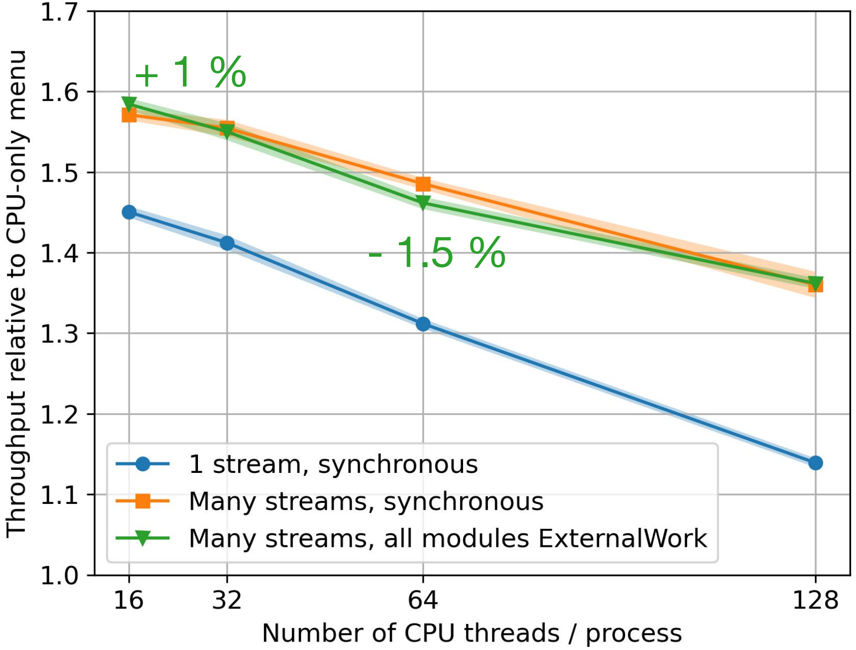
External worker mechanism

- Replace blocking waits with a callback-style solution
- Traditionally the algorithms have one function called by the framework, `produce()`
- That function is split into two stages
  - `acquire()`: Called first, launches the asynchronous work
  - `produce()`: Called after the asynchronous work has finished
- `acquire()` is given a reference-counted smart pointer to the task that calls `produce()`
  - Decrease reference count when asynchronous work has finished
  - Capable of delivering exceptions



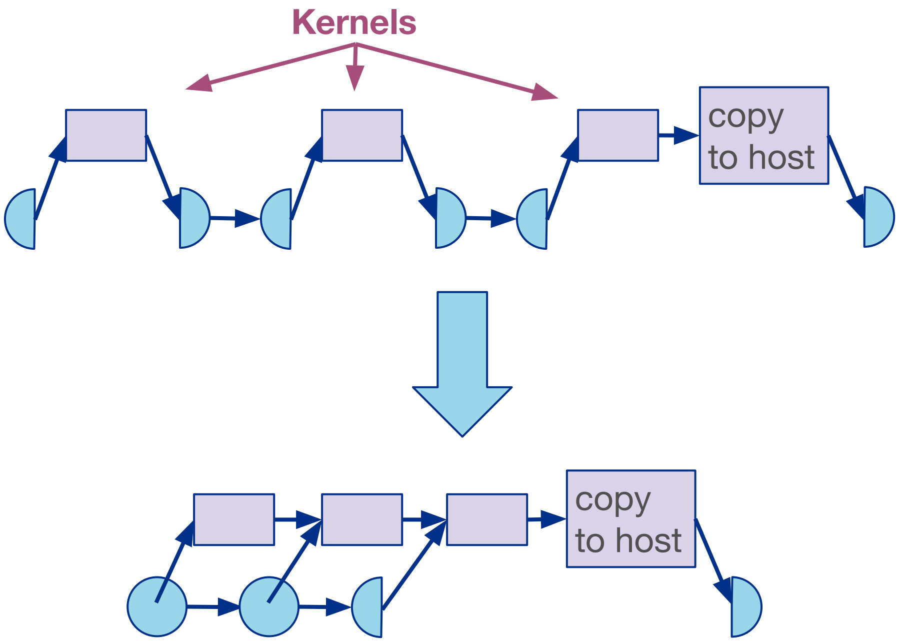
Make each CUDA module external worker

- Use of CUDA streams stays the same
- Every CUDA module does a non-blocking synchronization
  - It follows that the modules depending on the data of the CUDA-using module are scheduled to be run only after the GPU work has finished
  - We use `cudaStreamAddCallback()` to queue a host-side callback function that notifies the CMSSW framework of the completion of the GPU work
    - `cudaStreamAddCallback()` is deprecated, `cudaLaunchHostFunc()` gave same performance



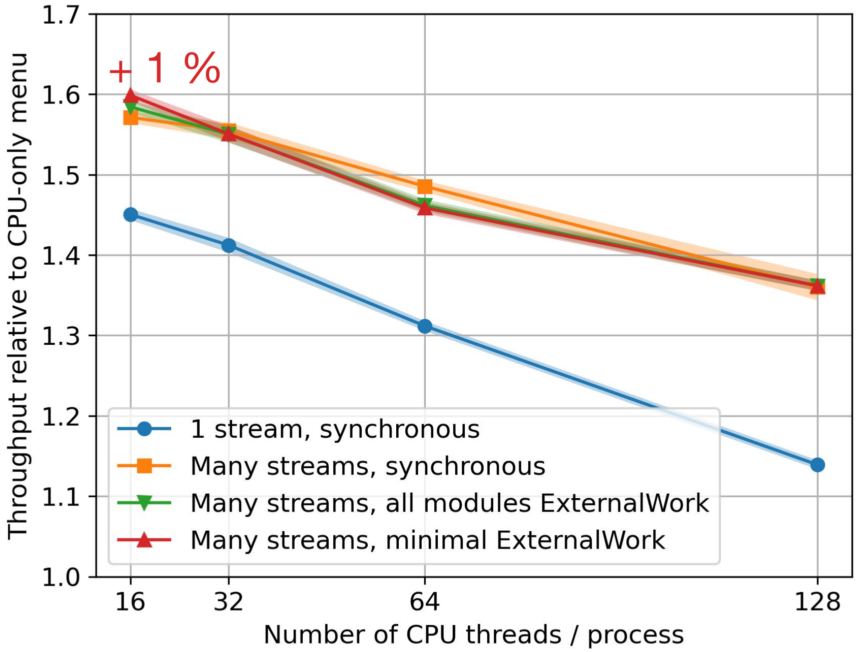
Minimize the external worker use

- Use of CUDA streams stays the same
- Modules that produce only "device-side" information do not really need synchronization with host
  - Instead we make the consuming module to call `cudaStreamWaitEvent()` in case it would use a different stream
  - Now framework can schedule the consuming modules without waiting their GPU work to finish
- This is the setup used in CMSSW



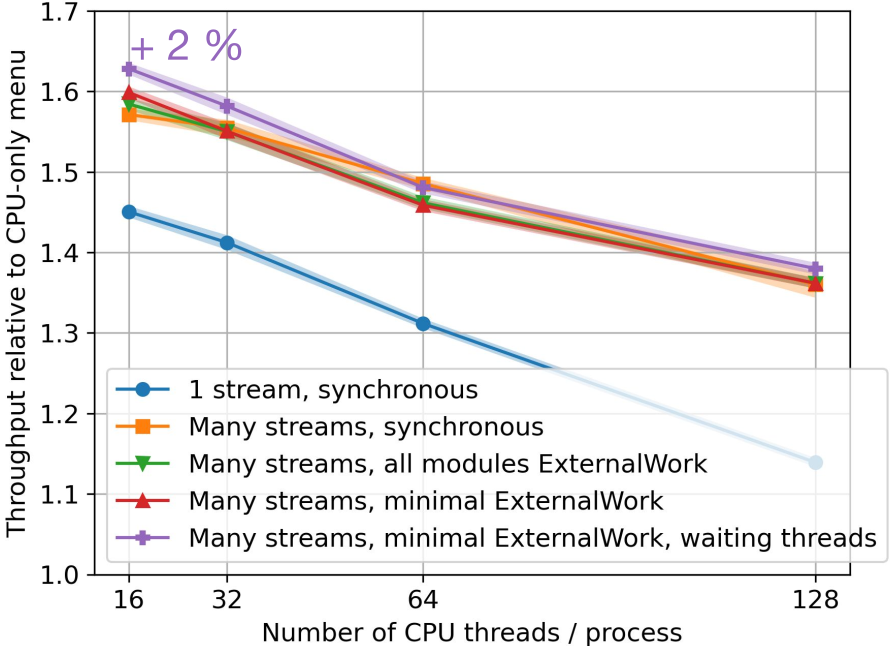
Minimize the external worker use

- Use of CUDA streams stays the same
- Modules that produce only "device-side" information do not really need synchronization
  - Instead we make the consuming module to call `cudaStreamWaitEvent()` in case it would use a different stream
  - Now framework can schedule the consuming modules without waiting their GPU work to finish
- This is the setup used in CMSSW
  - But can we do better?



More performant way to synchronize in CUDA

- After trying out various options, replacing the `cudaStreamAddCallback()` with a separate waiting thread that calls `cudaEventSynchronize()` gave about 2 % higher throughput
  - Required creating the CUDA events with flag `cudaEventBlockingSync`



Conclusions

- Demonstrated the performance impact of the design decisions of the CUDA module pattern in CMSSW
  - Using production High Level Trigger menu from 2022 as a test bed
- Good speedup (15-45 %) already from a simple single-stream with blocking synchronization approach
- Multiple CUDA streams improved the throughput by 7-20 %
- Making the synchronizations non-blocking in every module had mixed impact +1 .. -1.5 %
- Minimizing the synchronizations gave 1 % improvement for 16 threads
- Highest throughput with our own pool of threads waiting on `cudaEventSynchronize()`: ~2 % better than `cudaStreamAddCallback()`
  - 0-4 % better than blocking synchronization
- Expect these improvements be larger for longer-running kernels

Related contributions

- G. Parida: “Run-3 Commissioning of CMS Online HLT reconstruction using GPUs” earlier (14:30) in this session
- A. Bocci: “Adoption of the alpaka performance portability library in the CMS software”, Track 2 today 17:00
- M. Kortelainen: “Evaluating Performance Portability with the CMS Heterogeneous Pixel Reconstruction code”, Track X Thursday 11:45
- C. Jones: “CMSSW Scaling Limits on Many-Core Machines”, Tuesday poster session
- P. Gartung: “Vectorization in CMSSW applications”, poster

Spares

Impact of memory pool

- Same setup as on slide 5, but memory allocated with directly `cudaMalloc()` etc.
- Abysmal performance (as expected)