FERMILAB-SLIDES-19--069-SCD

Bringing heterogeneity to the CMS software framework

Presenter: **Oliver Gutsche[2]**

Andrea Bocci[1], W David Dagenhart[2], Vincenzo Innocente[1], Christopher D Jones[2], **Matti J Kortelainen[2]**, Felice Pantaleo[1], Marco Rovere[1]

CHEP 2019 4 November 2019

[1]CERN [2]FNAL

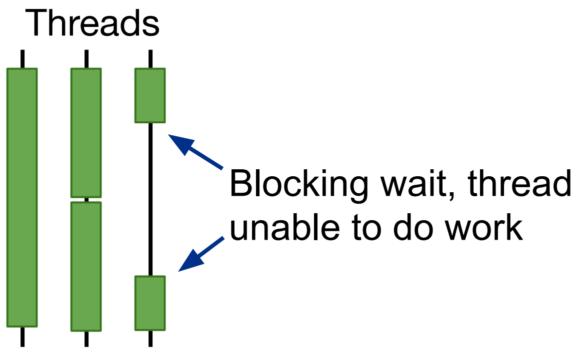
This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

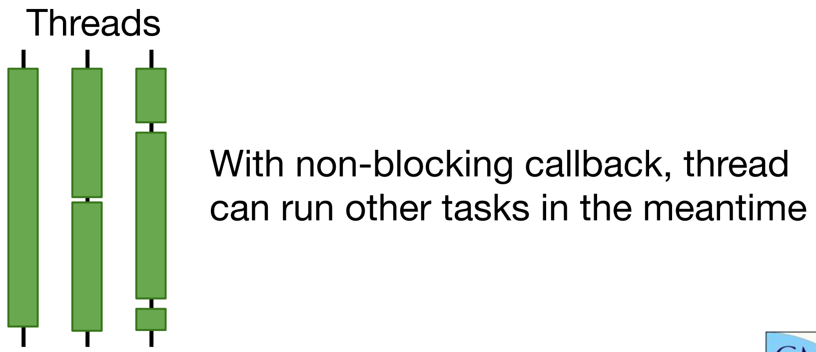
Introduction

- Co-processors or accelerators like GPUs and FPGAs are becoming more and more popular
  – Being considered for CMS High Level Trigger in Run 3 (see talk by A. Bocci on Wed Track 1)
  – Supercomputers
- CMS' data processing framework (CMSSW) implements multi-threading using Intel TBB utilizing tasks as concurrent units of work
- We have developed generic mechanisms within the CMSSW framework to
  – Interact effectively with non-CPU resources
  – Configure CPU and non-CPU algorithms in a unified way
- As a first step to gain experience, we have explored mechanisms for how algorithms could offload work to NVIDIA GPUs with CUDA

Concurrent CPU/non-CPU Processing

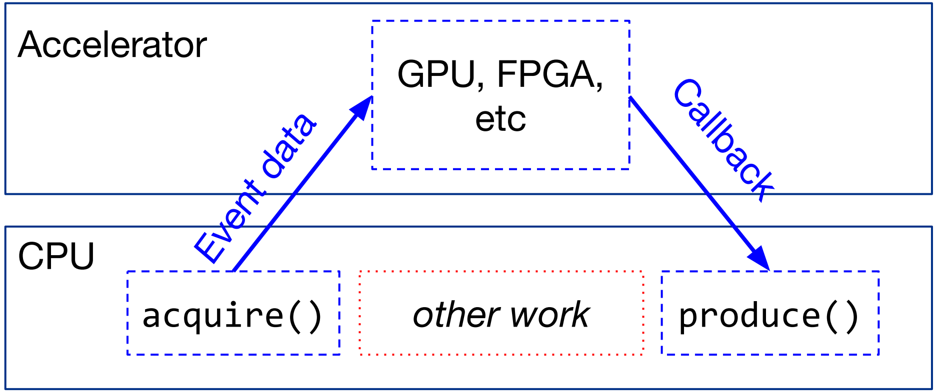
- When offloading work to non-CPU resources, the CPU needs to eventually know when that work is finished
- Could do a blocking wait
  - Then the thread would be blocked and could not do other work
- Instead, want to keep the TBB thread free to run other tasks





External worker concept

- 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



Unified configuration for CPU and non-CPU algorithms

- Want jobs for a workflow to run at any site
- Want same configuration for all jobs in a workflow
  - Be agnostic to the kind of hardware being used for a given job
  - Hash of configuration already used by framework to segregate data from different workflows
- Want to be able to keep CPU and non-CPU algorithms separate
  - No need to touch working code
  - Different hardware may want to group the work differently
    - E.g. CPU might want to spread over 3 modules while GPU wants them combined to 1
  - Not precluding having CPU and non-CPU algorithm in same module either
- Use provenance tracking to store the choice of technology along the Event
  - Framework already tracks the input data of each module Event-by-Event
- Such workflows need to be validated with all technology permutations

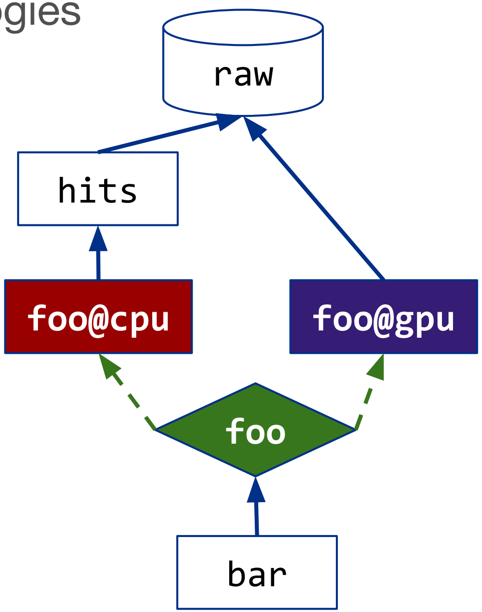
Switch mechanism for producers

- SwitchProducer added to configuration
  - Allows specifying multiple modules associated to same module label
  - At runtime picks one to be run based on available technologies
    - Consumers dictate which producers are run

```
hits = Producer("HitsProducer",
    input = "raw"
)

foo = SwitchProducer(
    cpu = Producer("FooProducer",
        input = "hits"),
    gpu = Producer("FooProducerGPU",
        input = "raw")
)

bar = Producer("BarProducer",
    input = "foo"
)
```



Goals for the pattern to interact with CUDA

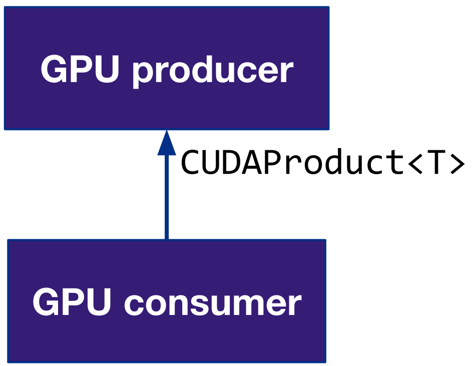
- Allow CPU to do other work while the GPU is running an algorithm
  - Asynchronous execution, i.e. CPU does not wait for the GPU to finish
- Minimize data movements between the CPU and the GPU
  - Transfer data only when necessary
- Mechanism for a chain of modules to share a resource
  - Resource being e.g. GPU memory or a CUDA stream
- Extendable to multiple device types, and multiple devices per type

CUDA pattern: asynchronous execution

- Use only asynchronous CUDA API calls during event processing
  - Mainly memory transfers and memsets
  - Kernel launches are asynchronous by construction
- Asynchronous CUDA API calls require the use of CUDA streams
  - Work items queued in a CUDA stream execute serially, but concurrently wrt other streams
  - Each parallel branch in the module DAG gets its own CUDA stream
- Avoid synchronization points
  - “Raw” memory allocations
    - Amortize their cost with a memory pool, currently based on cub `CachingDeviceAllocator`
  - `cudaDeviceSynchronize()`/`cudaStreamSynchronize()`
    - Instead use external worker to signal framework that the work is done without blocking
  - `assert()` in kernel code

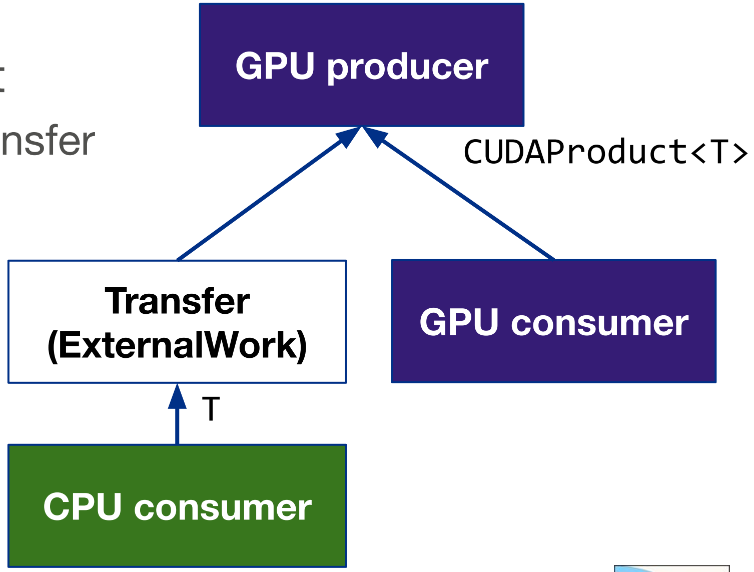
CUDA pattern: sharing resources between modules

- We introduced a wrapper template CUDAProduct<T> for a product of type T
  - Product T is partly or fully in GPU memory
- Wrapper holds the devide id and CUDA stream used to produce the product
  - Also CUDA event to mark the completion of asynchronous processing in case that was not finished when the module ended
- Consumer module uses
  - The same device
  - Either the same CUDA stream, or another that synchronizes with the input CUDA stream
- Two types of modules
  - Normal: launch work without synchronization
  - External worker: if need to transfer anything back to CPU and synchronize



CUDA pattern: minimizing data movements

- Add additional modules to do the transfers
- Output product type is different anyway between CPU and GPU
  - At minimum `T` vs. `CUDAProduct<T>`
- Exploit framework's behavior to run a producer only if some other module consumes the product
  - I.e. if no-one asks for the product in CPU, do not transfer



Conclusions and outlook

- CMSSW has generic building blocks to continue exploring the use of non-CPU resources

- We are exploring the performance characteristics of the described CUDA pattern

- We are exploring performance portability technologies like Kokkos, Alpaka, SYCL
  - Aiming for single-source approach for CPU and GPU capable algorithms
  - Need to understand how the pattern with CUDA could be evolved for those technologies