Fermilab | U.S. DEPARTMENT OF ENERGY | Office of Science

Support for Heterogeneous Computing in CMSSW

Matti Kortelainen for the CMS Collaboration

HSF WLCG Virtual Workshop on New Architectures, Portability, and Sustainability

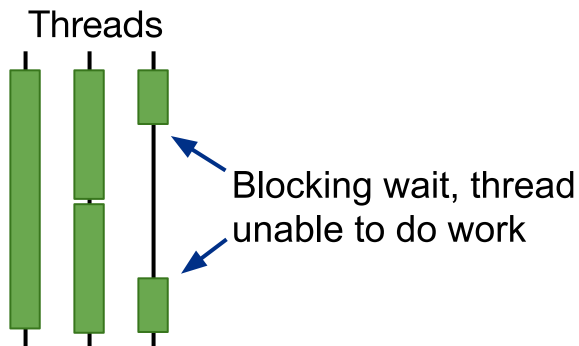
12 May 2020

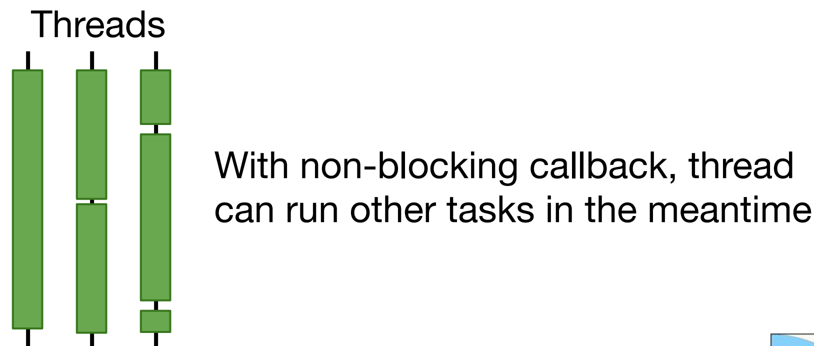
Introduction

- Co-processors or accelerators like GPUs and FPGAs are popular
  - Supercomputers
  - ALICE has used GPUS for a while, LHCb decided heterogeneous HLT1 for Run 3
  - CMS considers GPUs for High Level Trigger in Run 3
- 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
- We have also developed mechanisms for specific support for
  - Algorithms offloading to same-node GPUs with CUDA (CHEP19 talk)
  - Algorithms doing ML inference on remote services (CHEP19 talk)

Concurrent CPU/non-CPU Processing

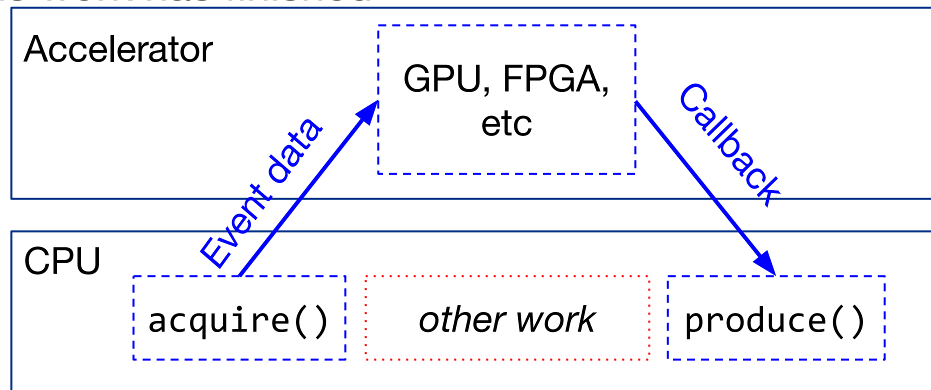
- When offloading work to non-CPU resources, the CPU needs to eventually know when that work is finished
- Could do a blocking wait
  - Then the thread would be blocked and could not do other work
- Instead, want to keep the TBB thread free to run other tasks





External worker concept

- General mechanism, can interface with *anything* external to the framework
- 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 be able 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 algorithm Event-by-Event
- Such workflows need to be validated with all technology permutations

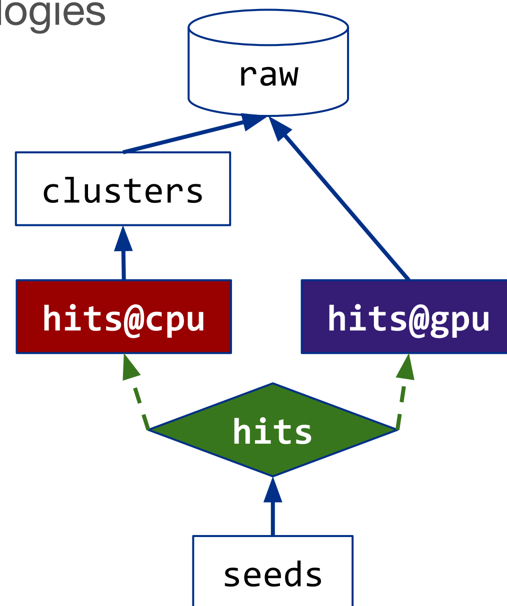
Switch mechanism for producers

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

```
clusters = Producer(“ClusterProducer”,
    input = “raw”
)

hits = SwitchProducer(
    cpu = Producer(“HitProducer”,
        input = “clusters”),
    gpu = Producer(“HitProducerGPU”,
        input = “raw”)
)

seeds = Producer(“SeedProducer”,
    input = “hits”
)
```



Considerations for supporting CUDA algorithms

- CMS code has a relatively flat time profile
  - Some functions take more time than others
  - Speeding up a single function has a small impact at best
- CMS code tends to process "large" amount of data with low arithmetic intensity
  - "Large" as larger than caches
  - But "small" as in not enough to fully utilize the GPU with data of one event
- Offloading "random algorithms here and there" may get limited by transfers
- To maximize the event processing throughput
  - Run kernels concurrently, overlap with data transfers
  - Offload long chains (or DAGs) of algorithms
    - Even if some of them would be a poor fit for the GPU by themselves
  - Minimize data transfers between CPU and GPU

Specific support for CUDA algorithms

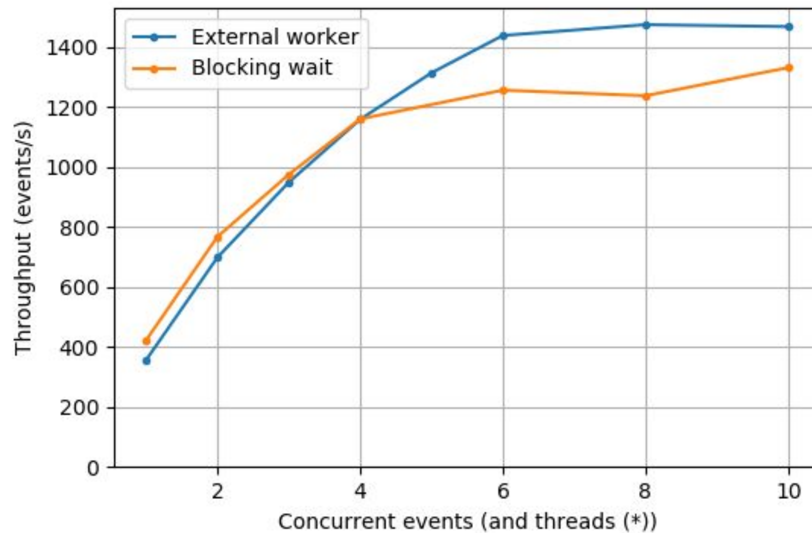
- Allow CPU to do other work while the GPU is running an algorithm
  - Use only asynchronous API and CUDA streams (=concurrent work queues)
    - CPU does not wait for the GPU to finish
  - Use CUDA callbacks to notify when CPU can proceed
    - Only after issuing a transfer from GPU to CPU memory
- Mechanism for a chain of algorithms to share a resource (e.g. GPU memory)
  - Introduced a wrapper for event products that holds CUDA device ID and CUDA stream
  - Re-use the same CUDA stream for subsequent work when possible
- Transfer data between the GPU and the CPU only when necessary
  - Framework runs a producer only if some other module consumes the product
    - I.e. if no-one asks for the product in CPU, do not transfer
- Extendable to multiple device types, and multiple devices per type

Experience from the CUDA support

- The overall model appears to work pretty well (CHEP19 talk)
- Some limiting factors experienced
  - CUDA API is thread safe, but protected with a single mutex
    - With many small concurrent CUDA operations the mutex becomes contended
  - Each CUDA operation has a ~constant overhead
    - With many small CUDA operations the relative cost of the overhead becomes noticeable
- An often proposed solution for “many small operations” is to batch event data
  - Would have non-trivial consequences on e.g. EDM and overall work scheduling
    - Large work imbalance between CMS events, e.g. pipeline performs poorly
  - We have started to explore scheduling strategies for algorithm DAGs where only some algorithms batch event data

Performance of external worker vs blocking wait

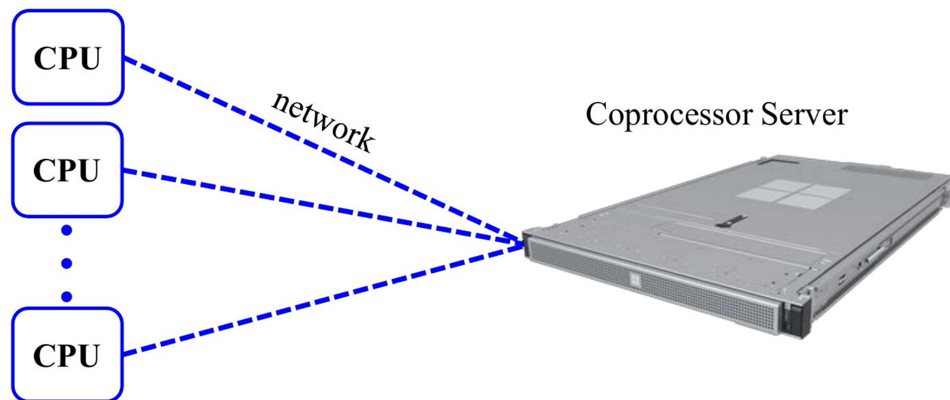
- Compare event processing throughput of a simulation of the Patatrack pixel tracking
  - Transfers SoA results back to CPU
  - Measured data transfer sizes, kernel times, and CPU times
  - Repeat with toy transfers and kernel/CPU operations taking the same amount of time
- Dual-socket Xeon Gold 6130
  - 2x16 cores (2x32 threads)
- Single NVIDIA Tesla T4
- Throughput of one job, node filled with CPU consuming work up to 64 threads
  - Number of CPU threads set equal to number of concurrent events
    - (*) External work with 1 concurrent event runs with 2 threads in practice



CMS

Specific support for remote services

- Services for Optimized Network Inference on Coprocessors (SONIC)
- Convert experimental data to neural network input, send to coprocessor using communication protocol
- Uses “external worker” mechanism for asynchronous, non-blocking requests
- Currently supports
  - gRPC with TensorFlow
    - Tested with Microsoft Brainwave (FPGA)
    - Compatible with any service using gRPC and TensorFlow
  - NVIDIA Triton for GPUs
    - Can serve models also from other ML frameworks than TensorFlow



CMS

Conclusions and outlook

- CMSSW has generic building blocks to continue exploring the use of non-CPU resources
- We have also developed specific support for CUDA and for ML inference on external services using the generic building blocks

- We are exploring performance portability technologies like Kokkos, Alpaka, SYCL
  - Aiming for single-source approach for “GPU capable” algorithms
  - Need to understand how the CUDA support could be evolved for those technologies
  - Goal for Run 3: portability between CPU and CUDA
    - To support algorithm development for the HLT
  - Goal for Run 4: “as portable as possible”

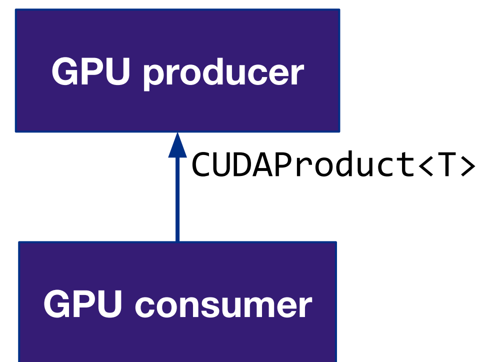
Backup

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