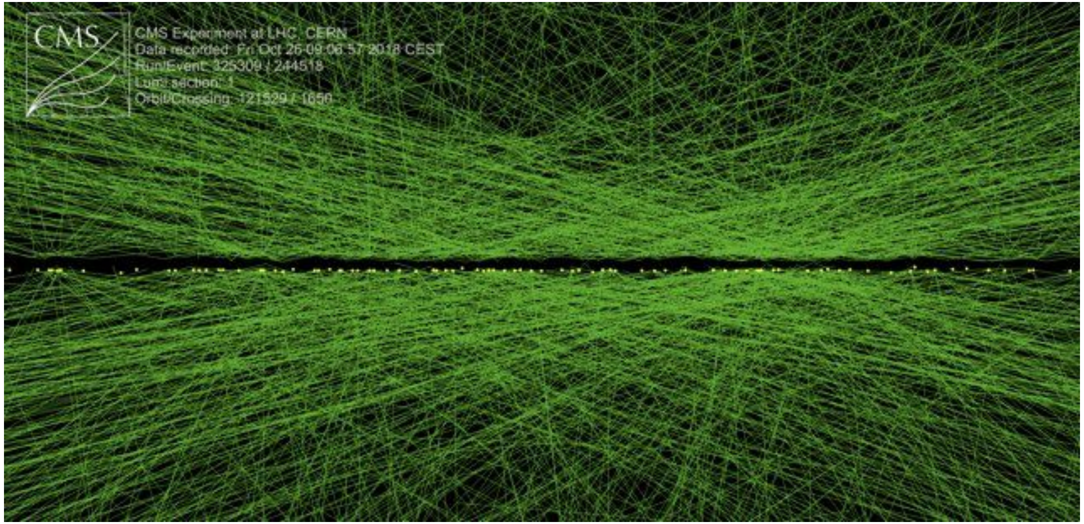
Status of Patatrack pixel tracking use case

Matti Kortelainen
HEP-CCE F2F meeting
5 November 2020

Introduction

- The overall approach of Patatrack pixel tracking
  - Reconstruct pixel-based tracks and vertices on the GPU
  - Leverage existing support in CMSSW for threads and on-demand reconstruction
    - Also explore adding support for heterogeneous computing into the framework
  - Minimize data transfers
    - Start data processing on GPU from RAW data of pixel detector
- Main goal: deployment in CMS HLT for Run 3
  - Gain experience on heterogeneous HLT farm for HL-LHC

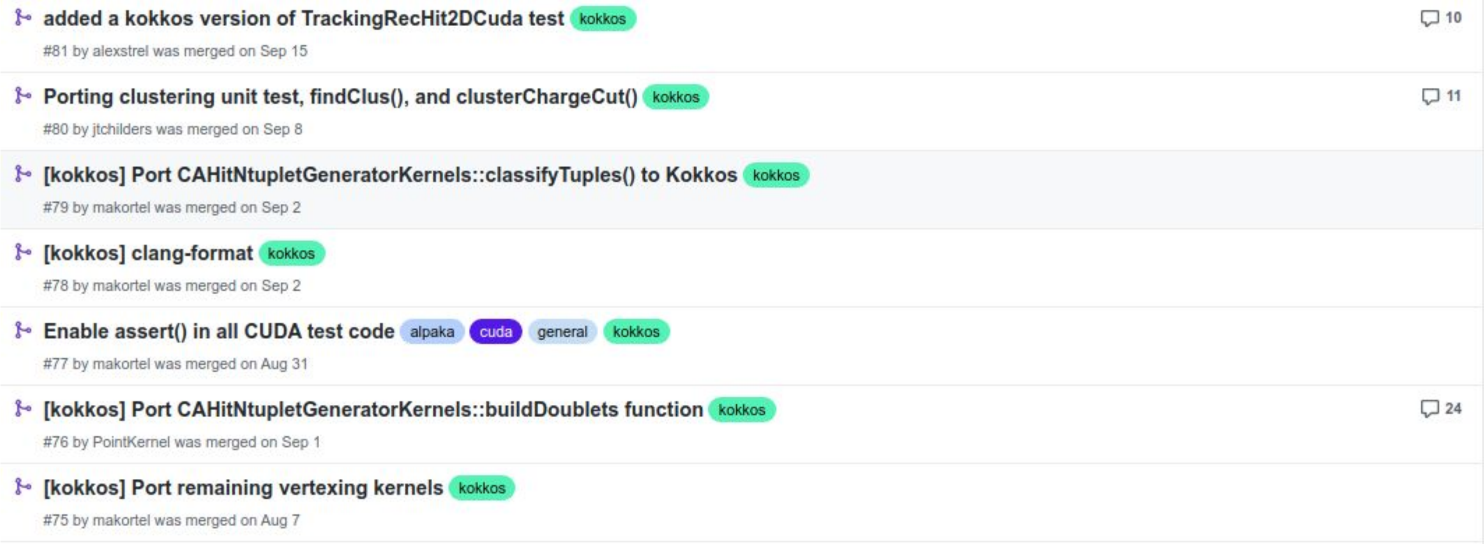


Summary of year 1

- Patatrack pixel tracking code extracted from CMSSW into a standalone code
  - https://github.com/cms-patatrack/pixeltrack-standalone/
  - Infrastructure specifically designed for experimentation of different frameworks
  - In retrospect this was an excellent choice by making the exploration much easier
- Initial porting of the original CUDA program to Kokkos is complete
  - Physics (clusters, hits, tracks, vertices) is validated against the original CUDA program
  - Able to run on CPU and GPU
    - Single build with a run-time choice of what code to run
- Now working on performance improvements
  - Kokkos port was expected to be slower "by construction"

Scale of the porting effort

- So far 71 merged Pull Requests, ~7 months of wall clock time
  - By myself, Yunsong, Taylor, Alexei
- ~13kSLOC of code in the Kokkos port
  - Much of that is copy-paste from original CUDA program



added a kokkos version of TrackingRecHit2DCuda test kokkos
#81 by alexstrel was merged on Sep 15
10

Porting clustering unit test, findClus(), and clusterChargeCut() kokkos
#80 by jtchilders was merged on Sep 8
11

[kokkos] Port CAHitNtupletGeneratorKernels::classifyTuples() to Kokkos kokkos
#79 by makortel was merged on Sep 2

[kokkos] clang-format kokkos
#78 by makortel was merged on Sep 2

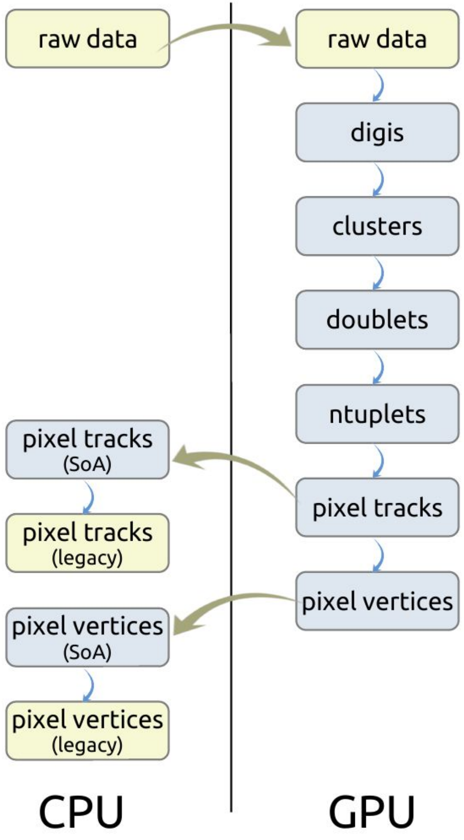
Enable assert() in all CUDA test code alpaka cuda general kokkos
#77 by makortel was merged on Aug 31

[kokkos] Port CAHitNtupletGeneratorKernels::buildDoublets function kokkos
#76 by PointKernel was merged on Sep 1
24

[kokkos] Port remaining vertexing kernels kokkos
#75 by makortel was merged on Aug 7

Reminder of the Patatrack program

- Copy the raw data to the GPU (~250 kB/event)
- Run multiple kernels (39) to perform the various steps
  - Decode the raw data
  - Cluster the pixel hits
  - Form hit doublets
  - Form hit ntuplets (triplets and quadruplets) with a Cellular Automaton algorithm
  - Clean up duplicates
  - Vertexing
- Copy only the final results back to the host
  - Optimized SoA format
  - ~4 MB/event for tracks, ~90 kB/event for vertices
  - Convert to legacy format if requested



Reminder of the Patatrack program (2)

- The program is in the limit of “many small GPU operations”
  - All overheads matter
  - At the peak throughput the CUDA runtime mutex is hammered at O(100 kHz)
- Key elements for performance
  - Process multiple events concurrently by using CUDA streams
  - Asynchronous execution: CPU does other work while the GPU is processing
    - “Continuation passing” with callback functions instead of blocking synchronization calls
- Caching allocator (“memory pool”) to amortize the cost of CUDA memory allocation functions
- More information
  - arXiv:2008.13461 for the algorithms
  - arXiv:2004.04334 for the framework side

Standalone program

- Flexible GNU Make-based build system
- Simple framework
  - Uses TBB tasks similarly to CMSSW (event loop, asynchronous external work)
- I/O is ignored
  - Data of all 1000 input events is fully read into memory before the event loop
    - From CMS Open Data of TTbar + pileup-50 simulation
    - When processing more than 1000 events, the events in the set are recycled
  - No output
- Performance is measured as the event processing throughput over the event loop
  - Simple to measure, exactly the quantity that matters in the end
  - Incorporates all overheads that would be there in an event loop of an experiment framework
- https://github.com/cms-patatrack/pixeltrack-standalone/

Impact on building

- Kokkos requires a runtime library
- Available backends are chosen at the build configuration time of the library
  - Can have one host **serial** backend
  - Can have one host parallel backend: OpenMP, **pthreads**
    - Chris Jones has a private prototype of TBB backend
  - Can have one device parallel backend: **CUDA**, HIP, (HPX), (SYCL in `develop`)
- Supporting multiple device parallel backends requires a separate runtime library for each backend
  - In fact worse, need a separate library for each CUDA major architecture: Pascal (6.x), Volta/Turing (7.x), Ampere (8.x)
  - Same goes for vector architectures on CPU side if one wants to make use of Kokkos' optimizations on those
- One host backend is always needed

Impact on building (2)

- If CUDA backend is enabled in the runtime library, all source files including any(?) Kokkos header must be compiled with CUDA-capable compiler (`nvcc` or `clang`)
  - Even if the source file would not use any CUDA functionality
  - I assume the same holds for HIP and SYCL as well
- `nvcc` is unable to link device code from shared objects, consequences:
  - Kokkos runtime library must be built as a static library
  - Can not use relocatable device code
    - A source file must contain (directly or `#include`) all device code called from that source file
      - I.e. can not link to device code in another object file
    - Can not use CUDA dynamic parallelism

Backend choice

- Set of available backends chosen at build configuration time of Kokkos runtime
- Actual backend to be used is chosen at compile time
  - By default the "most advanced" backend is used (CUDA > OpenMP > Serial)
  - Can choose explicitly with a template argument
- In Patatrack Kokkos port, separate versions of a "framework module" are compiled for all available backends
  - The versions to be used are chosen at run time (command line)
  - With a Kokkos runtime built for Serial+CUDA, it is possible to run the Serial-only versions on a machine without GPU
    - Requires some care, also on the algorithm implementation side

Writing algorithms

- High-level API: `parallel_for`, `parallel_reduce`, `parallel_scan`
  - Can be nested (with some restrictions)
- Details of the iteration and operations are controlled with a policy
  - `RangePolicy`: 1-dimensional range, all elements are independent
  - `MDRangePolicy`: 1-6 dimensional range, all elements are independent
  - `TeamPolicy`: thread teams / hierarchical parallelism (more on next slide)
    - Corresponds to CUDA’s grid of blocks of threads
- `RangePolicy` and `MDRangePolicy` are simple to use when hierarchical parallelism is not needed
  - Developer does not have to think about distribution of work to threads

Hierarchical parallelism

- In CUDA one has a grid of blocks of threads
  - Threads of a block can synchronize (e.g. barrier) and have common scratch space (shared memory)
  - Blocks themselves are independent (no synchronization available)
- Kokkos supports this model via thread teams ("league of teams of threads")
  - Barrier synchronization, scratch space
  - Can do reduction over the threads of a team
- Number of threads in a team is not exactly portable
  - Serial backend must have exactly 1, pthread backend can use at most the number of CPU threads, CUDA backend has the same limits as CUDA itself (128/256 are typical)
  - Can be mostly mitigated by letting Kokkos decide with `Kokkos::AUTO()`

Data structures

- `Kokkos::View<T>` is an N-dimensional array of type `T`
  - Reference-counted, works similar to `std::shared_ptr`
  - Can be passed to device functions by value (recommended pattern)
  - Layout can be controlled with template parameters
    - **Different default layout for host and device backends**
    - Possibility for custom layouts
  - Template parameters to enable optimizations based on intent
    - E.g. very easy to use CUDA texture access for random-access constant data
      - Could be useful for calibration data, not tested yet though
- Generic way to construct a `View` on the optimal host memory space for a `View` of device memory space
  - CUDA device memory -> CUDA pinned host memory
  - Host memory -> host memory (shortcut, no actual copy)

Data structures (2)

- By default the `View` initializes the content
  - On GPU memory runs a kernel with the default constructor
  - Can be very expensive if initialization is not needed (first access is write)
- Out of the box `View` does not use memory pool
- Patatrack code has not much use for arrays of > 1 dimension
  - Track parameters and covariance matrix is one exception
  - We have not tried (yet?) change the current approach based on Eigen
- Works about as well as any other smart pointer
  - Also as painful for constructing Structures-of-Arrays

Asynchronous execution

- The operations (`parallel_X()`, `deep_copy()`) should be thought to be asynchronous wrt. the calling host thread
  - Details depend on the backends
- Must synchronize explicitly in a way that blocks the host thread
  - I.e. no direct support for continuation passing
- There is some support for fine-grained task parallelism
  - Internally does provide continuation passing style chaining of tasks
  - But the host thread must do a blocking wait in the end
  - So far we have not tested Kokkos tasking

Concurrent events

- The simple framework implements concurrent events with separate “lanes” that run the algorithms for their events
  - Similar to “streams” in CMSSW
- In CUDA concurrent kernels require the use of CUDA streams
- Kokkos does not provide (yet) a portable mechanism for concurrent kernels, but allows the use of CUDA streams
  - The concept has been tested, but not at the scale of the full application
- No direct support for continuation passing
  - On the other hand using CUDA callbacks does not add much non-portability on top of the stream management
- Not tested much yet
  - Simple tests work for both Serial and CUDA backends
  - Very first attempt with Serial backend in the full application lead to assertion failures

(Unfair) performance comparison

- Processing for ~5 min wall clock time
  - Tracks and vertices **not** transferred back to host
- All tests run on the same Cori GPU node
  - Exclusive access
  - Pinned to 1 NUMA node
  - Node otherwise empty
- Kokkos uses 1 concurrent event
  - Serial uses 1 CPU thread
- Throughput from one execution

	Test	Throughput
Original CUDA program	10 concurrent events (GPU saturation point	1700 events/s
	1 concurrent event	760 events/s
	1 concurrent event, caching allocator disabled	**140 events/s**
Kokkos port	**CUDA backend**	**92 events/s**
	Threads backend (peak at 7 threads)	26 events/s
	Serial backend	13 events/s

Reasons for poor performance

- Kokkos port runs more kernels than the original CUDA program
  - Kokkos does not have team-wide `parallel_scan()` yet
    - Currently calling a `parallel_scan()` for the entire league and post-processing the result to mimic team-wide `parallel_scan()`
  - Kokkos does not have a sort function callable from device
- By default `Kokkos::View` does not use any memory pool
  - `cudaMalloc()+cudaFree()` are expensive (and synchronize)
- Only limited use of asynchronous execution
- No event-level concurrency

Summary of our Kokkos experience

- Nice high-level API
  - Some functionality missing that reduces performance
    - Team-wide prefix scan
    - Sort that could be called from device code
- Works nicely for a code that is fully compiled for a specific machine
- Works less nicely for a code that tries to “compile once, run everywhere”
  - Need one Kokkos runtime library for each pair of (CPU vector architecture, GPU architecture)
- Kokkos::View does not seem very useful for Structure-of-Arrays
  - Use for N-dimensional arrays has not been tested
- Kokkos::View gives easy way to use texture caches (not tested though)

Next steps

- Continue to understand finer details of Kokkos
    - Understand better the performance difference wrt. CUDA (i.e. improve)
    - Try out HIP and SYCL backends
    - Try out some not-yet-tested features
        - Concurrent execution, tasks
- Update reference CUDA implementation to latest, Kokkos to 3.2, CUDA to 11
- Eventually move to the next technology
    - SYCL(2020)/DPC++(/oneAPI) ?
        - Effort has actually started already outside CCE (at CERN)
    - In principle can be started concurrently with finishing the Kokkos port
- Priorities can be discussed