



CMS Patatrack project

A. Bocci¹, V. Innocente¹, **M. Kortelainen**², F. Pantaleo¹, M. Rovere¹
CERN¹, FNAL²

2019 Joint HSF/OSG/WLCG Workshop

March 19, 2019

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.



The Patatrack group



- Patatrack was formed by people with common interest and a varied pool of expertise
 - Software optimisation
 - Heterogeneous architectures
 - Track reconstruction
 - High Level Trigger
- Work started in 2016 with the participation to the EuroHack 2016 event, sponsored by NVIDIA
- And continued through 2017 to 2019 with self-organized Hackathons at CERN, collaboration with Openlab, training and working with students, and so on



The Patatrack demonstrator



- Goal is demonstrate that part of the HLT reconstruction can be efficiently offloaded
 - Running on a single machine equipped with GPUs
- Focus on a $\sim 10\%$ slice of HLT time consumption
 - Pixel local reconstruction
 - Pixel-only track reconstruction
 - Vertex reconstruction
- Other groups have started to work on
 - Calorimeters local reconstruction
 - Full track reconstruction
- For more details see closeby talks in
 - ACAT 2019, 10–15 March, Saas-Fee (Switzerland)
 - CDT/WIT 2019, 2–5 April, Valencia (Spain)



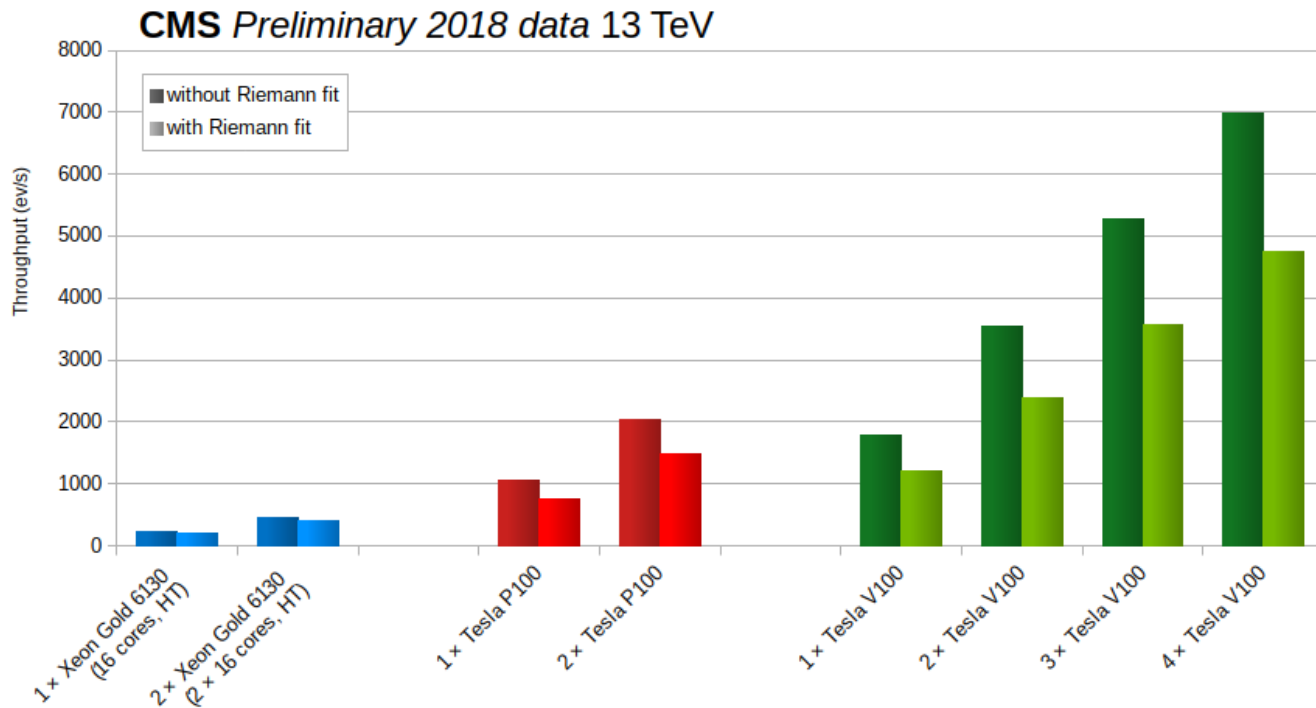
The Patatrack demonstrator workflow



- Copy the pixel raw data to the GPU
- Pixel local reconstruction
 - Decode the raw data
 - Clustering
 - Calibrations
- Pixel-only tracking
 - Form hit doublets
 - Form hit quadruplets with Cellular automaton algorithm
- Optionally
 - Full track fit (Riemann, Broken-line fits)
- Some GPU algorithms are same, others different wrt. (legacy) CPU
 - Implementations are currently different
 - Bitwise or statistically identical physics performance
- Organized as a chain of 3 GPU producer modules
 - Pass GPU data from one producer to the next
 - Use the CMSSW's "external worker" mechanism



The Patatrack demonstrator (2018)



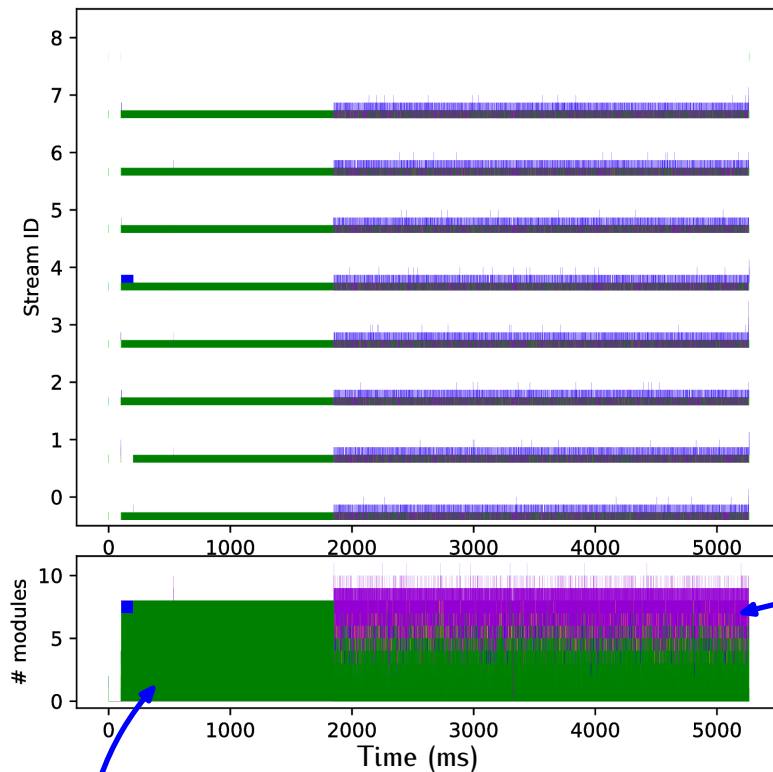
- 2018 data: average pileup 50
- HLT-like configuration, optimised for maximal throughput
- One Tesla V100 is 5×–7× faster than one Xeon Gold 6130



CPU utilization



modules running event stalled module running read from input
modules running other multiple modules running external work



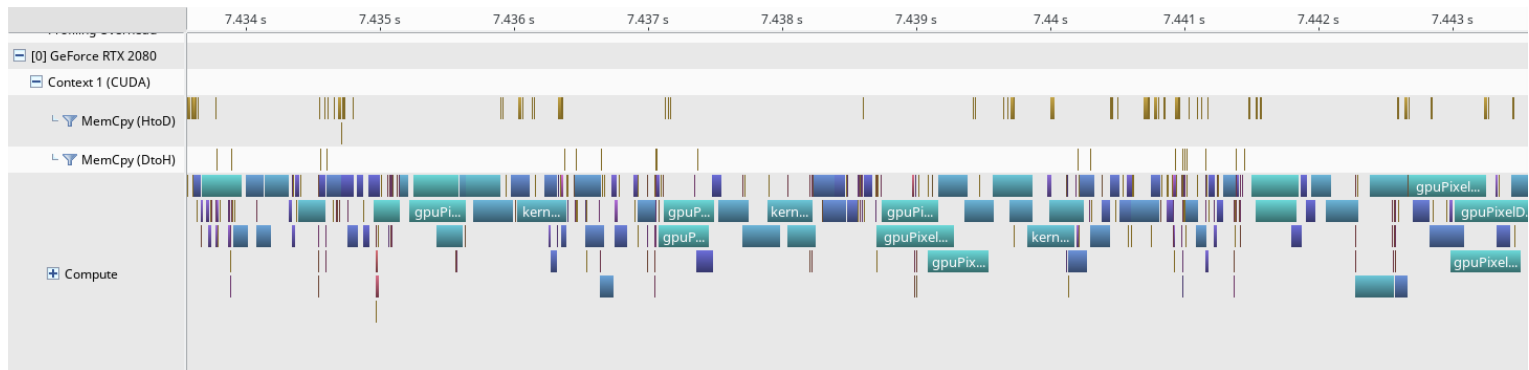
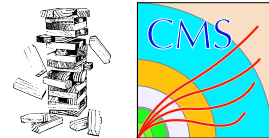
Number of running modules

- Caveat: different machine (i7-4771, GeForce RTX 2080)
 - 8 threads and 8 concurrent events
- After the initialization
 - CPU utilization is roughly 50 %
 - There are roughly 4–5 external workers scheduled in parallel
- NB: this workflow is “artificially” tuned to minimize the CPU utilization

Number of scheduled external workers



GPU utilization



- Screenshot of NVIDIA Visual Profiler for a random 10 ms period
- Kernels and data transfers being run in parallel



Lessons learned: design principles



- For optimal performance, follow a Data Oriented Design
 - Memory operations are costly, computations are almost free
 - Design the data structure for maximal efficiency (SOA vs ... vs. AOS)
 - Implement the algorithms around the data structure
 - Avoid object-oriented patterns in critical code e.g. data formats
 - ★ inheritance, virtual functions, etc
- Most (all?) GPU operations (memory copies, running “kernels”, etc) should be asynchronous
 - The “kernels” run on the GPU while the CPU is doing other work
 - The GPU can transfer data to and from the host while both the CPU and the GPU are working
- Memory transfer, and especially data format conversions, between CPU and GPUs are costly
 - In some cases, almost as much as running the original algorithm itself



Lessons learned: tools and architectures



- CUDA and CMSSW support different sets of compilers and C++ features
 - CUDA 10.1 supports
 - ★ C++ 14
 - ★ GCC 8, CLANG 7
 - ▷ CUDA 10.0 supported GCC 7, CLANG 6
 - CMSSW 10.6.X supports
 - ★ C++ 17
 - ★ GCC 7 and GCC 8, CLANG 7
 - ★ CUDA 10.1 in latest pre-release (was 10.0 before)
- Unfortunately, we need to keep the host and device code somewhat separate
 - Host code can use C++ 17 features
 - Device code (and common code) is limited to C++14 features
 - You do not want to `#include` framework (or ROOT) headers in device code!



Lessons learned: what about CMSSW?



- Redesign dedicated data formats for use on GPUs
 - In fact, they might be more efficient also on traditional CPUs
- Design a chain of algorithms (framework modules) that work on the GPU
 - Without copying data back and forth
- Take advantage of the “external worker” approach in CMSSW
 - Launch the work on the GPU, schedule other work in parallel on the CPU
- Split GPU modules in two parts
 - The part that deals with the framework and the rest of the CMSSW
 - The part that deals with the GPU data structures and kernels
- Split the GPU-related work in two (or more) modules, e.g.
 - Copy data from CPU to GPU, launch kernels
 - Copy data from GPU to CPU
 - ★ ran only if another modules consumes the CPU SOA
 - Transform CPU SOA to CPU legacy data format
 - ★ ran only if another module consumes



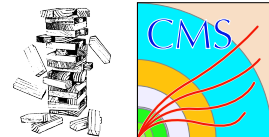
Model for CUDA Producers



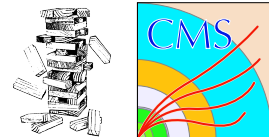
- Aim to avoid blocking synchronization as much as possible
- A helper object gives the CUDA device and stream to use for the algorithms
- Memory management
 - Raw CUDA allocations and frees should be avoided within the event loop
 - Preallocating memory buffers as module member data leads to unnecessarily high GPU memory use
 - We went for a caching allocator for device and pinned-host memory that amortizes the cost of raw CUDA allocations
 - ★ Currently based on the caching allocator of cub
- GPU event products are like regular EDM products, but enclosed in a wrapper that holds also the CUDA device and the CUDA stream
 - Allows the consumer to set the device, and queue more work to the same CUDA stream
 - Allows also the TBB-flowgraph `streaming_node` style operation
 - ★ Module in the middle of the chain may only queue more asynchronous work
 - ★ Later module in the chain synchronizes (with “external worker”)



Conclusions



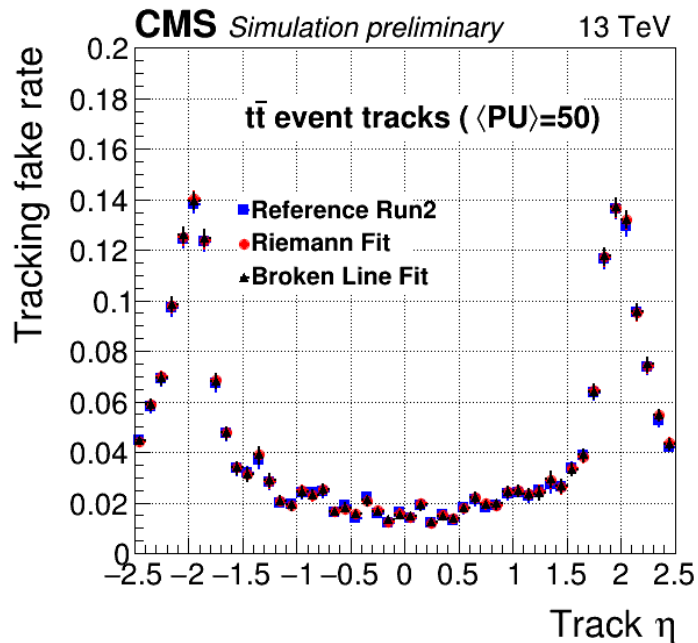
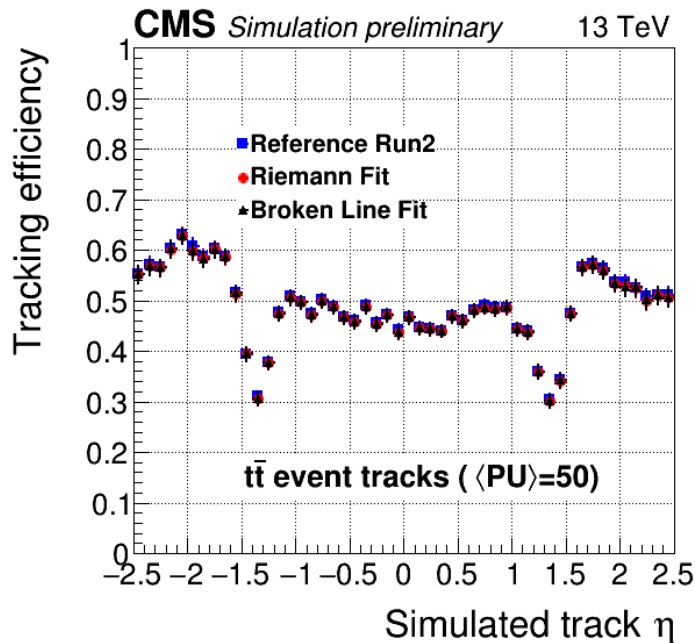
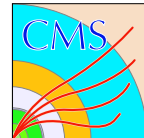
- We have demonstrated that GPUs are an efficient alternative to traditional CPUs
 - For complex tasks like track reconstruction
- Next steps
 - Integrate the developments in the official CMSSW
 - Continue evolving the framework to make it easier to leverage GPUs
 - Focus on code portability and avoiding code duplication as much as possible
 - Study how more algorithms and data structures could benefit from GPUs
 - Study local vs. remote offloading to GPUs



BACKUP MATERIAL



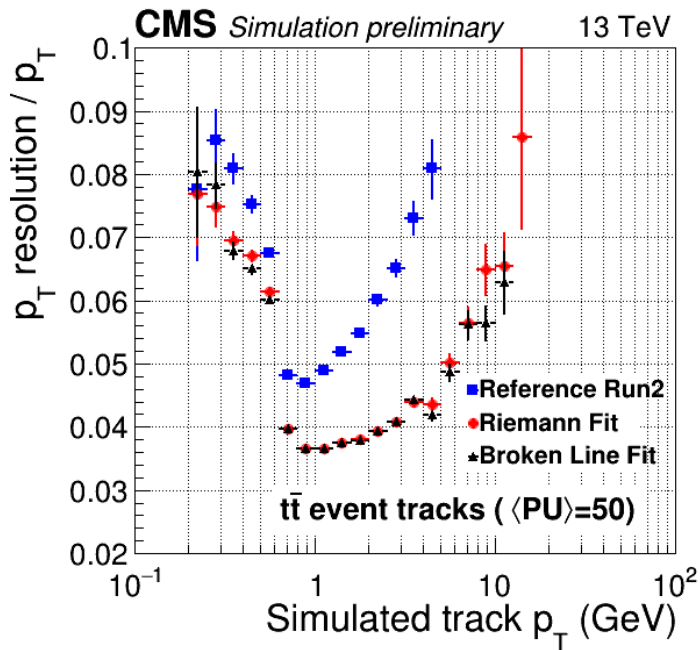
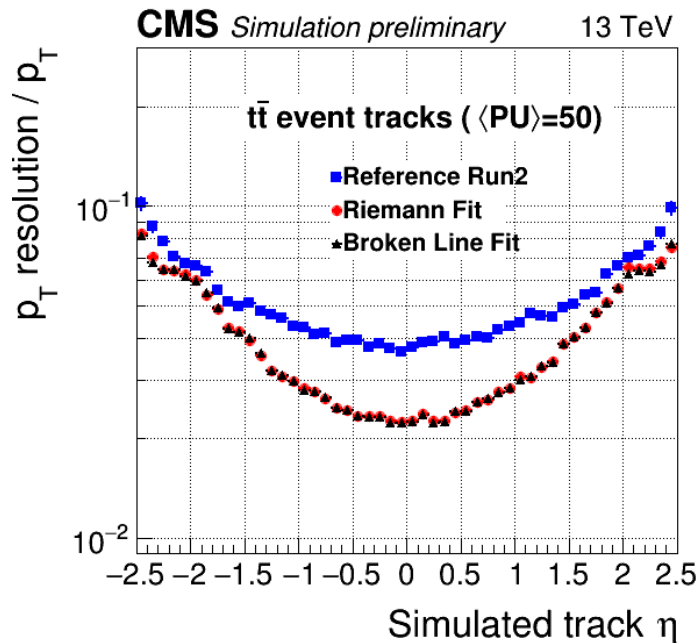
The Patatrack demonstrator (2018)



- Similar efficiency and fake rate as with legacy CPU algorithm
- More information: [CMS Detector Performance Note DP-2018/059](#)



The Patatrack demonstrator (2018)



- Proper fits improve resolution significantly
- More information: [CMS Detector Performance Note DP-2018/059](#)