Parallelized Kalman-Filter-Based Reconstruction of Particle Tracks with Accurate Detector Geometry

G. Cerati[4], P. Elmer[3], M. Kortelainen[4], S. Krutelyov[1], S. Lantz[2], M. Lefebvre[3], M. Masciovecchio[1], K. McDermott[2], D. Riley[2], M.Tadel[1], P.Wittich[2], F.Würthwein[1], A.Yagil[1]

1. UCSD 2. Cornell 3. Princeton 4. FNAL

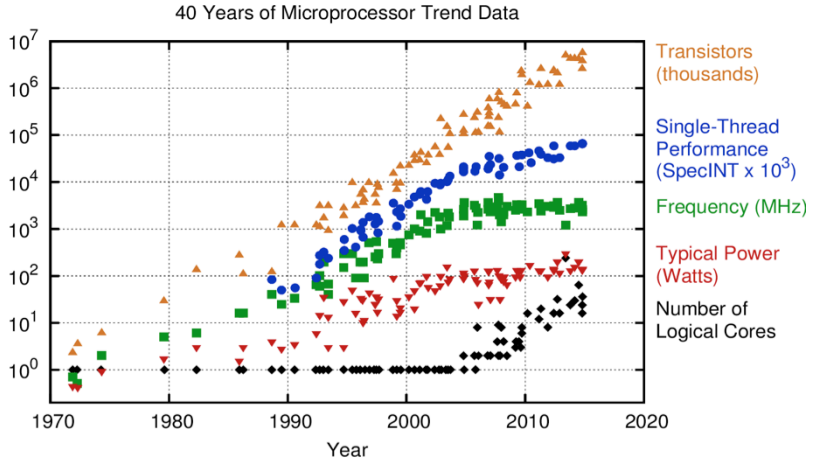
Connecting the Dots 2018, University of Washington, Seattle

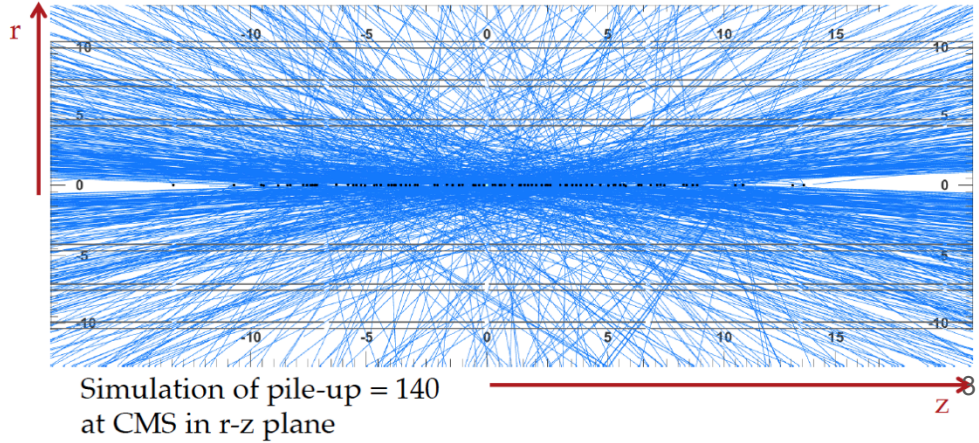
Outline

- Project introduction
  - Motivation for many-core Kalman filter implementation
- Project details
  - Geometries, event data
  - Algorithms & Data structures
  - Vectorization & Multi-threading
  - Architectures & Compilers
- Current focus & Status
  - Physics performance, scaling, GPU status
- Conclusion

Project overview

- Cornell, Princeton, UC San Diego + Fermilab (all CMS).
  - 3-year NSF grant, now in the final year + CMS R&D project
  - Fermilab and University of Oregon: 3 year DOE SciDAC4 grant (just started)
- Mission statement: Explore Kalman filter based track finding and track fitting on many-core SIMD and SIMT architectures --- because:
  - a) that's what we are getting (with reduced cache size and memory b/w per register); and
  - b) we really need the additional resources to be able to process HL-LHC data
- Goal: Run in CMS HLT for Run3 and beyond; maybe also parts of offline





Kalman filter

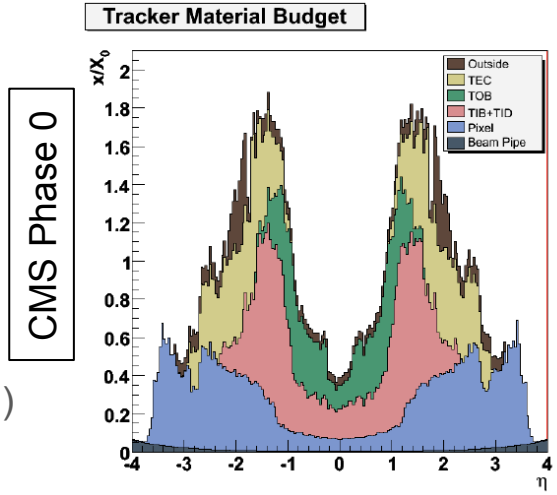
Why use Kalman filter:

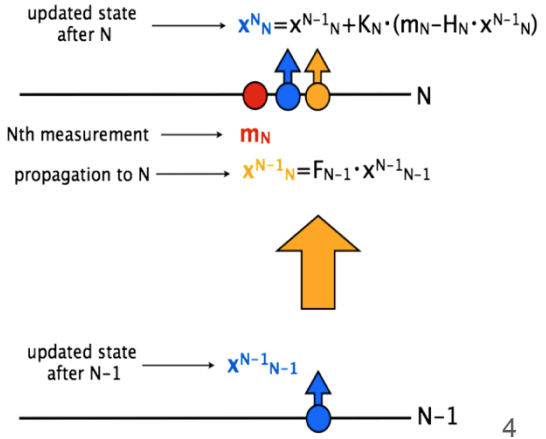
- Widely used & well understood
- Demonstrated physics performance:
  - Can handle multiple scattering and energy loss (badly needed)

Our goals for Kalman filter based track finding:

- Make effective use of parallel and vector architectures
- Maintain physics performance
- Preserve consistent systematics across platforms

Our work is complementary to tracklet-based divide and conquer algorithms.





Project details – What we do and How

Code name: mkFit – Matriplex Kalman Fitter / Finder

Tasks related to track finding

1. Seed finding: have basic implementation but it is not actively developed
   a) Now we either use CMSSW seeds or MC truth seeding for development
   b) For CMSSW seeds we do cleaning prior to track finding
2. **Track finding**: this is our primary focus. Several algorithms:
   a) Best hit: take the best hit on every layer
   b) Standard: on every layer check all compatible hits, select N best candidates for each seed
   c) Minimize copy: apply cleverness to b. to reduce data copying and unnecessary cloning of Tracks
   d) Full vector: different cleverness in handling & management of candidates belonging to same seed

   a) and b) are reference implementations

   In c) and d) are variations of b) where we are trying to do better …
3. **Track fitting**: secondary focus, is actually much easier
   a) Starting with a vector of found hits and initial track parameters, use Kalman filter on all the hits.
   b) This was the first piece we developed, it gave us great results and encouragement :)
   c) Simple cases saw x8 vectorization speedup on KNC and good multi-thread scaling
4. **Validation**: physics & computational performance

Geometries & Events

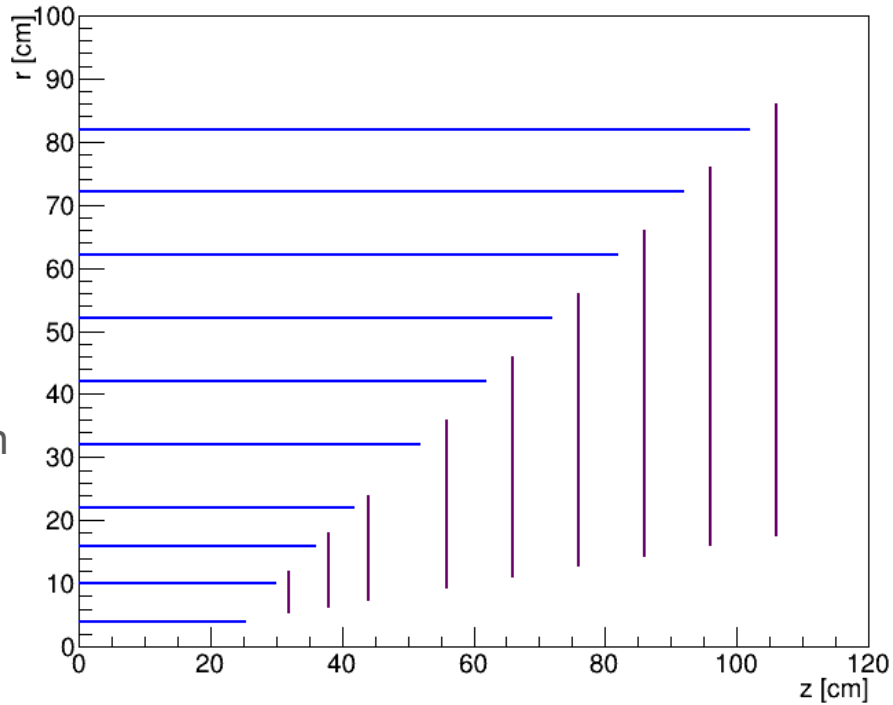
- Current work is focusing on CMS-2017 geometry, Iteration 0 tracking
  - Iteration 0 = Starting from pixel seeds having 4 hits with beam spot constraint
  - Using CMSSW generated events:
    - 10 muon events for development (barrel/endcap/transition region, low pT)
    - ttbar, ttbar + 35 or 70 PU
  - use a simple event data format, basically a memory dump of our structures
    - hits, seeds, sim-tracks, reco-tracks (for phys. performance comparison)
    - when run within CMSSW, the data will be pulled from Event record (in progress).

We can run track finding on full detector, iteration 0, physics performance comparable to CMSSW.

- Early on we developed a simple standalone tracker geometry (the Cow):
  - Early prototyping and development done with Cylindrical Cow – barrel only.
  - When addressing endcaps and transition region: Cylindrical Cow With Lids.
  - Includes simulation with multiple scattering and energy loss.

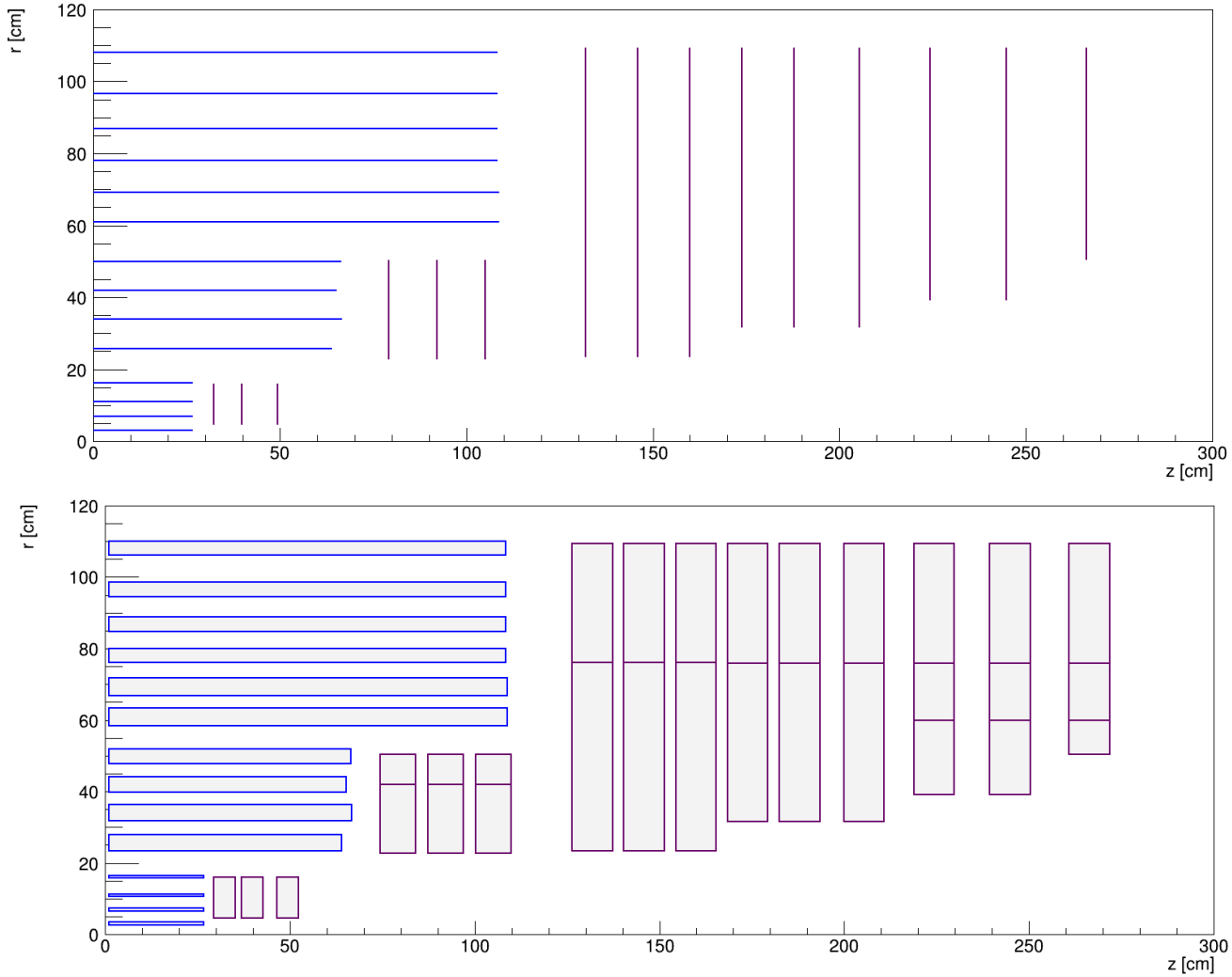
Cylindrical Cow with Lids

- Simple basic geometry
  - transition region |eta| 1 to 1.3
  - “long” pixels on all layers
- Supporting several geometries keeps tracking algorithms independent of actual geometry!
  - And points to required generalizations
- Geometries are implemented as a plugin / code that runs during program initialization and sets up geometry and algorithm steering structures.

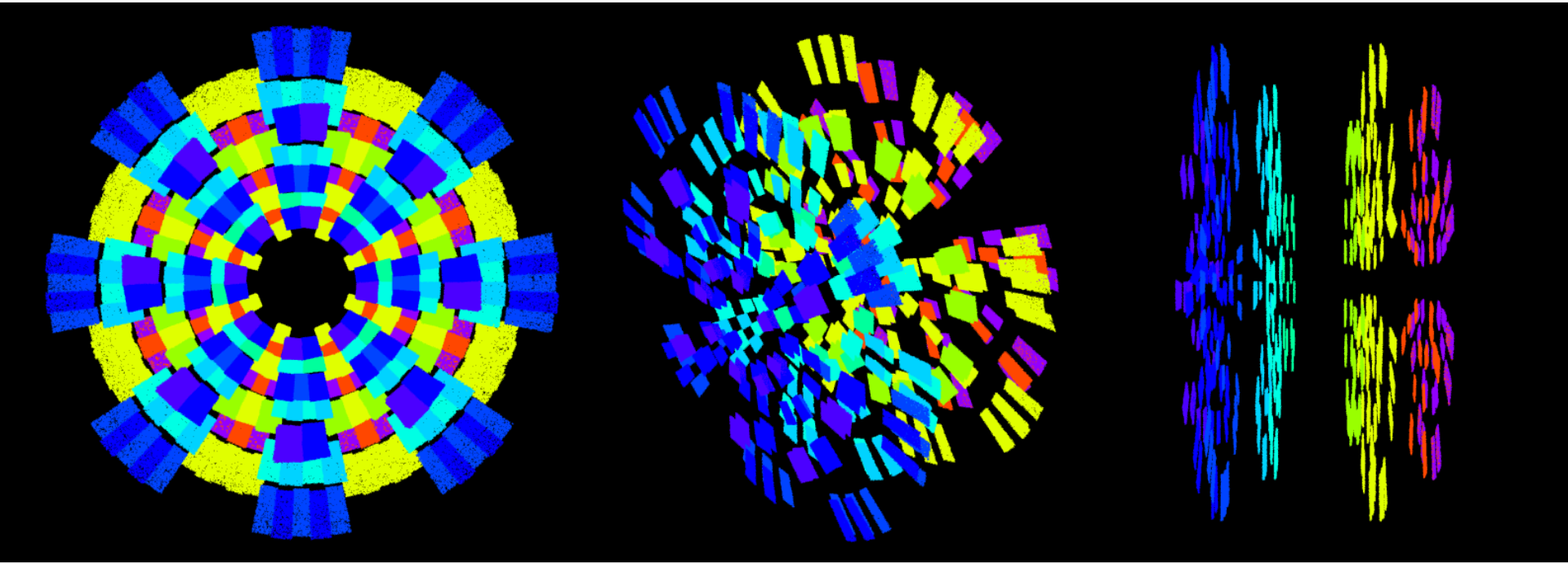


CMS-2017

- Top – what is usually shown.
  - Lines at layer centroids
- Bottom – actual size of layers accounted for.
  - This is actual geometry used by mkFit.
  - Extracted automatically from CMS sim hit data.
  - Note: stripes on endcap disks are results of partial stereo layer coverage



CMS, example of an endcap disk



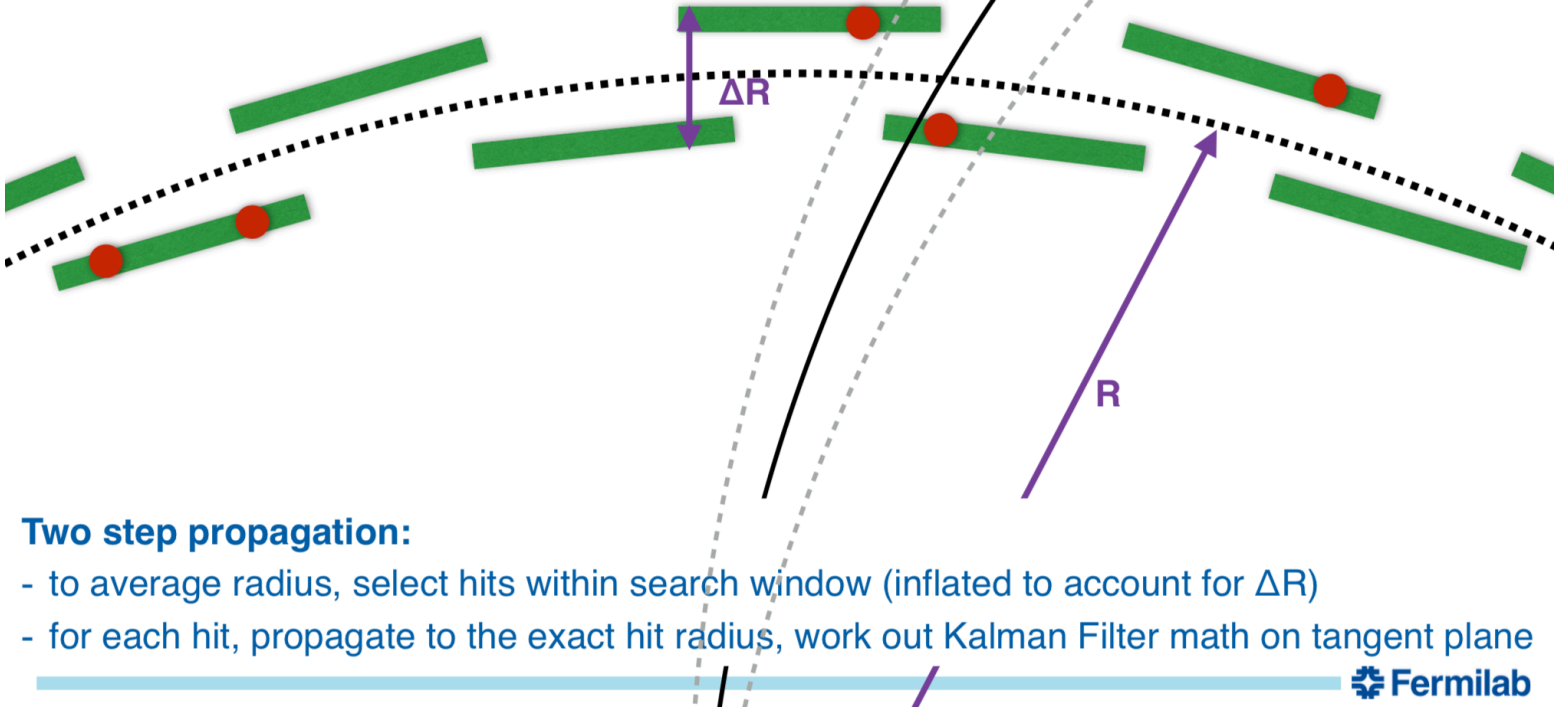
Geometry description & approximation

Unlike CMSSW, we DO NOT deal with detector modules! We use layers only:

- Propagate to the center of a layer and perform hit pre-selection.
- Requires additional propagation step for every compatible hit!
  - But this really vectorizes well. [And we do not have to propagate to a module.]
- Stereo: mono / stereo modules are put into separate layers.
- Can *only* pick up one hit per layer on outward propagation.
  - Could pickup overlap hits during backward fit, or after, for layers where it matters.
- **Simplifies track steering code and minimizes candidate specific code.**

Barrel layer representation

Barrel layer representation uses only 3 parameters:
average radius R, radius spread ΔR, length Z
=> avoid filling memory with complex geometry structures (position, rotation of each module)
=> almost detector independent



Two step propagation:

- to average radius, select hits within search window (inflated to account for ΔR)
- for each hit, propagate to the exact hit radius, work out Kalman Filter math on tangent plane

High level overview of track finding - Steering code

This is more or less similar for all track finding methods.

- Process seeds, assign them into regions: barrel, endcap +/-, transition +/-
- parallel_for (TBB) every region
  - parallel_for over seeds from current region (typically in bunches of 16 or 32)
    - loop over layers, according to a “layer plan” for current region
      - propagate to current layer
      - select matching hits
      - process matching hits, calculate chi2
        - update is either done here (standard) or after selection (minimize copy)
      - select best candidate(s) for further processing
    - select best final Track for each seed & do backward fit

Data structures & Algorithms

- Keep Hit and Track as small as possible, no heap allocated member data
  - Position: global x, y, z
  - Momentum: 1/pT, phi, theta
  - This gives us 6 dimensional track state and errors
- LayerOfHits: container for hits belonging to the same layer
  - Sorted and indexed into a 2D binned structure giving fast lookup of compatible hit indices
    - Use Radix sort on phi/z (barrel) or phi/r (endcap)
  - Hit pre-selection does not vectorize well and is not cache friendly
- MkFitter/MkFinder: encapsulation of fitting / finding algorithms (sort of toolbox)
  - This is intermediate level between steering code and low-level vectorized code.
    - Requires copying from Hit/Track classes into Matriplex - the vectorization friendly format
  - Also, that's where barrel / endcap separation happens
    - propagate to R / Z, compute Chi2 & update parameters Barrel / Endcap

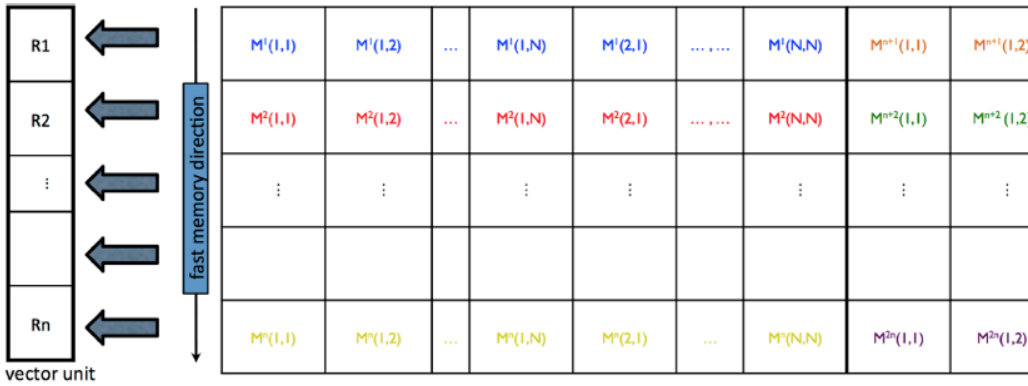
Matriplex - Vectorization of small matrix operations

"Matrix-major" matrix representation designed to fill a vector unit with **n** small matrices operated on in synch

Use vector-unit width on Xeons

- With or without intrinsics
- Shorter vector sizes w/o intrinsics
- For GPUs, use the same layout with very large vector width

Interface template common to Xeon and GPU versions



Matriplex - GenMul code generator

GenMul.pm - Generate matrix Multiplication code for given matrix dimensions
Features:

- Generate C++ code or Intrinsics (AVX, MIC, AVX-512)
  - Output is then included into a function.
  - For intrinsics it takes into account instruction latencies
- Can be told about known 0 and 1 elements in input and output matrices:
  - This reduces number of operations by more than 40%!
- Can do on-the-fly transpose of input matrices
  - Avoids transposition for similarity transformation.

We use this for vectorizing all Kalman filter related operations.

For propagation we rely on compiler vectorization (#pragma simd for the outer propagation loop over track candidates).

Multi-threading, Architectures & compilers

For multi-threading we use TBB:

- Two parallel_fors over tracking regions and seeds (shown in steering code)
- parallel_for over events - multiple events in flight
  - This is crucial for plugging the gaps arising from unequal load in track finding tasks!

We actually started with OpenMP but it is hard to do dynamic problem partitioning. TBB is already used in CMSSW.

Architectures & compilers:

- x86_64 (AVX, AVX-512), KNC (MIC), KNL (AVX-512)
  - icc, gcc; we use --std-c++11
- Nvidia / CUDA
  - Have implementations of track fitting and track finding (best hit and minimize copy)

Current focus & Status

What we are working on now

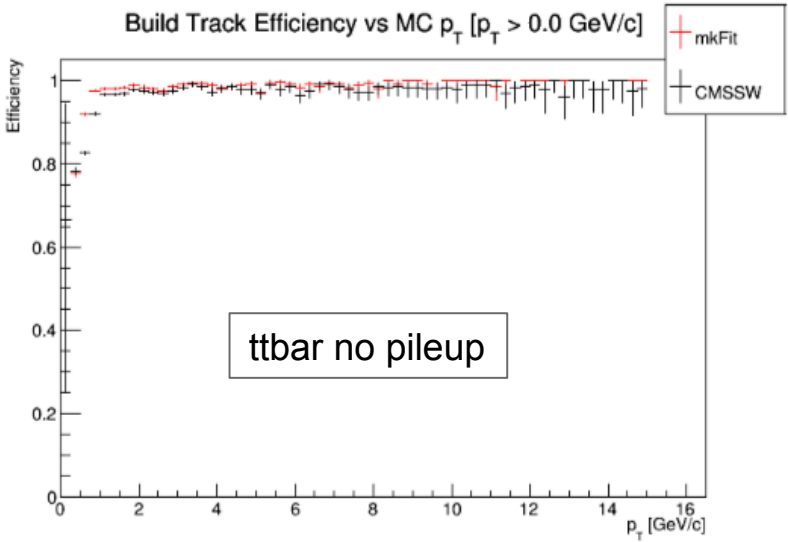
- Meaningful comparison of track finding with CMSSW for Iteration 0
  - Physics performance – almost there
    - Polishing the edges, tuning of track finding parameters
    - Use cluster charge information to remove hits due to out of time pileup
    - Still need to implement cleaning / merging of resulting tracks
      - While we do seed cleaning, we get duplicates & ghosts, especially in the endcaps where there are a lot of module overlaps within layers.
  - Computational performance, i.e. speed, scaling, and memory footprint
    - x86_64 (Skylake Silver vs. Gold), KNL
- Consolidation of complete work-chain, including fitting
- Still have some ideas to further improve vectorization speedup and overall performance.

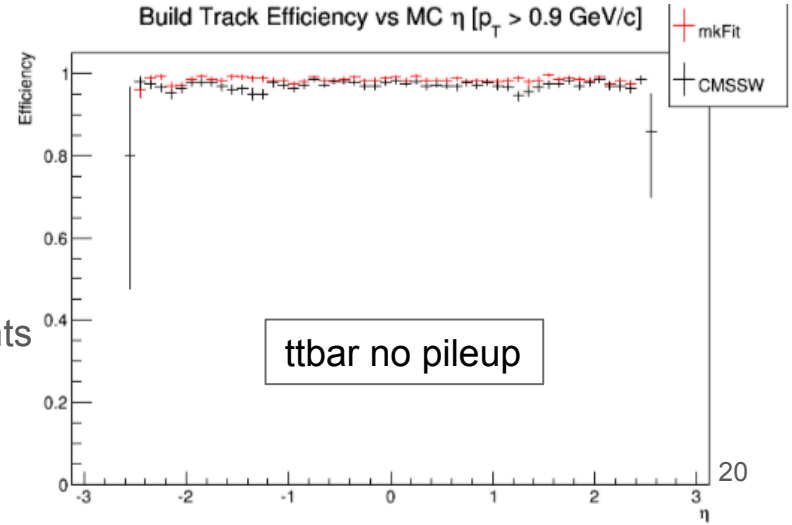
Muon gun & ttbar no pileup

- Efficiency denominator: findable sim-tracks with a matching seed.
  - Remember – this is iteration 0 / initial step using pix quadruplets as seeds

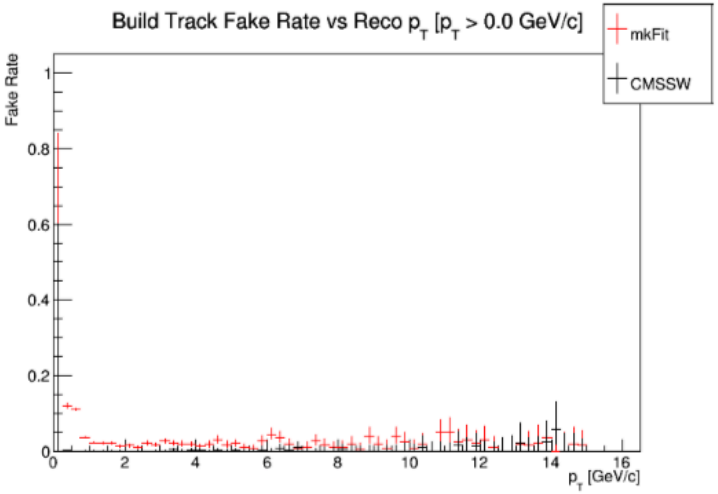
A. 10 mu per event, pt from 0.5 to 10 GeV
   - Practically fully efficient, zero fake rate
   - Duplicate rate spikes to ~50% in endcaps
     - Direct consequence of seed duplicates
     - Should go away once we implement cleaning and merging

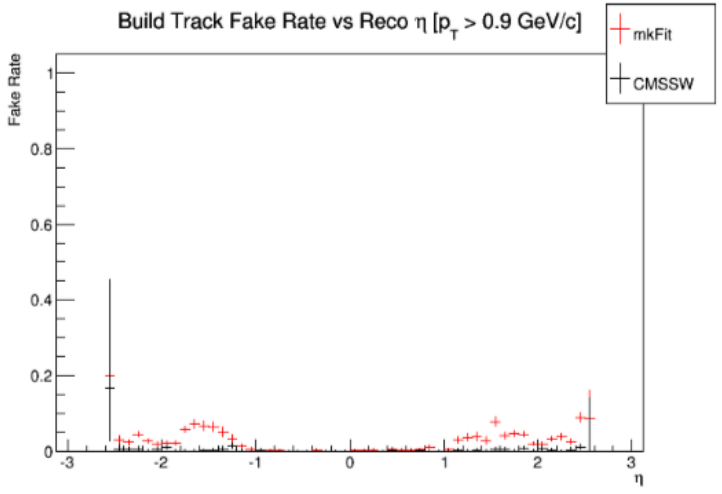
B. ttbar no pileup - basically the same as 10 muon events
   - Some fakes in transition region (~5% eta 1.2 to 1.7)
     - Cleaning / merging can reduce this

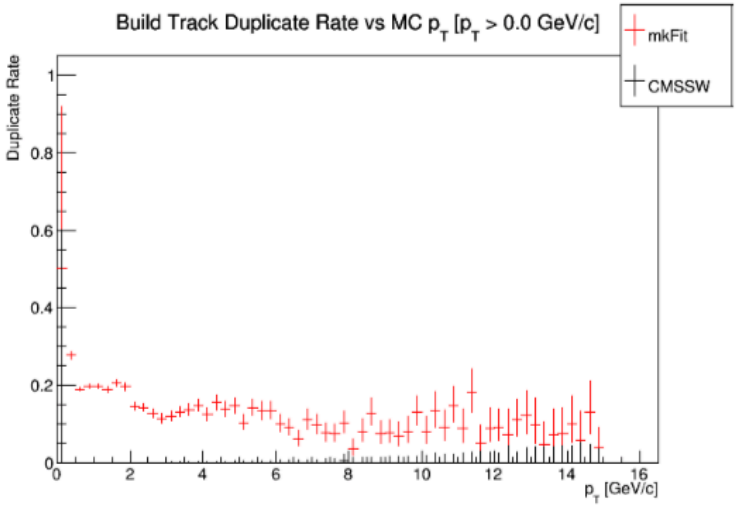


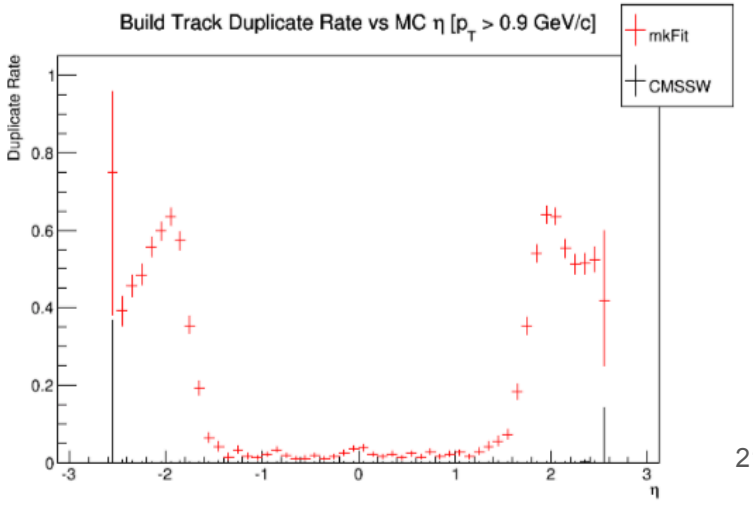


ttbar, no pileup



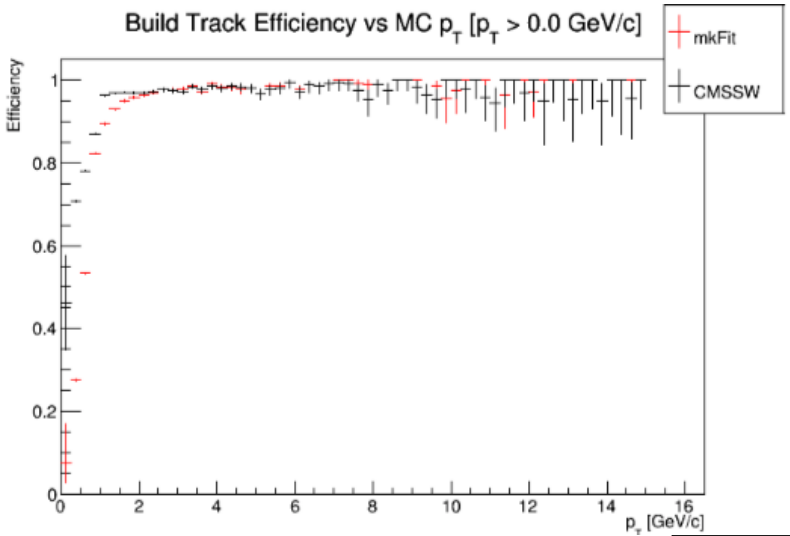


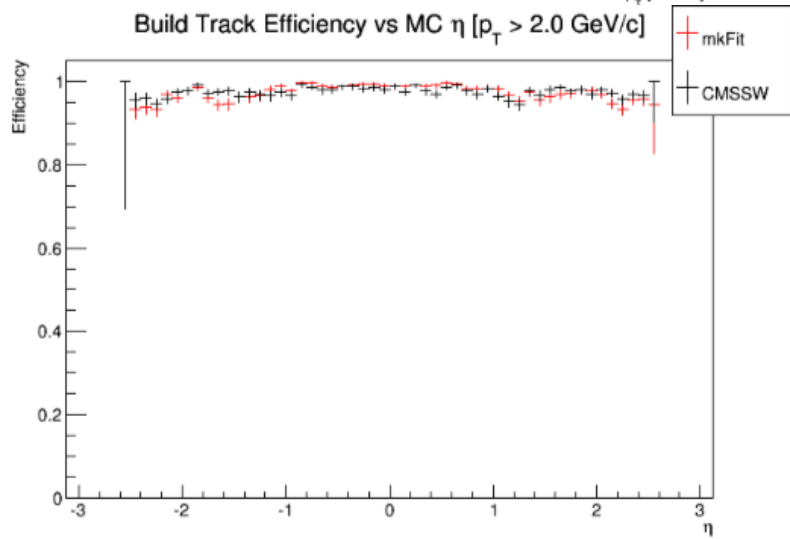




ttbar + 70 PU

- Efficiency comparable for pt > 2 GeV
  - Exploration of low pt inefficiency is ongoing
- Fake rate is more significant
  - Final cleaning should help
  - Investigate quality criteria
- Duplicate rate similar to no pileup / muon case
  - Which means it has the same origin – duplicates in input seed collection.
  - Post-build cleaning / merging will get this down to CMSSW levels

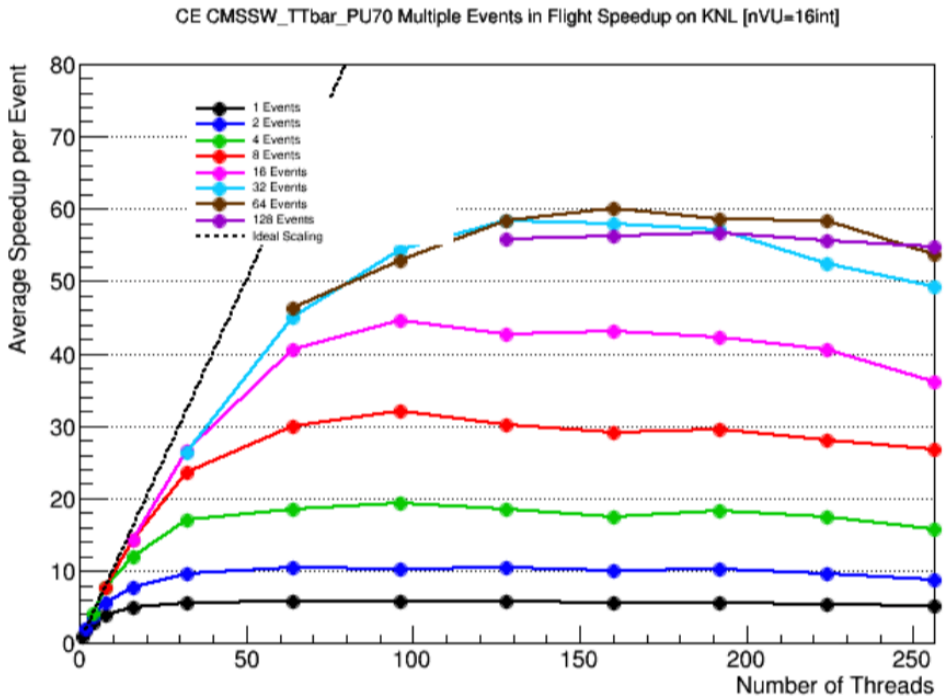




Computational performance

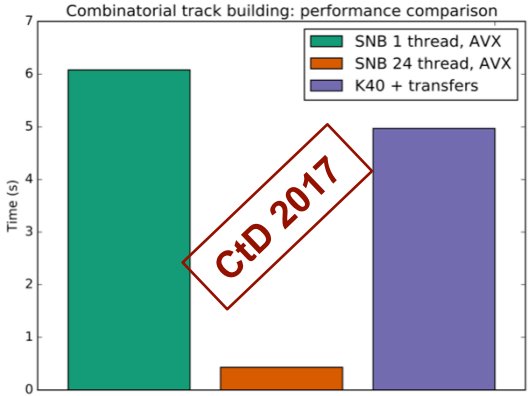
- Vectorization (building only) gives about 2 to 3x speedup (AVX, AVX-512)
- For multi-threading, having multiple events in flight is crucial!
  - Currently cleaning up “administrative” tasks we didn’t care much about before, e.g., loading of hits, seed cleaning.
- Compared to CMSSW, mkFit is about 10x faster (both single-thread).
  - Intentionally vague as this is work in progress.
  - icc significantly boosts mkFit performance
- ttbar + 70 PU @ KNL: 80 events / s

KNL



GPU results: Barrel-only combinatorial search

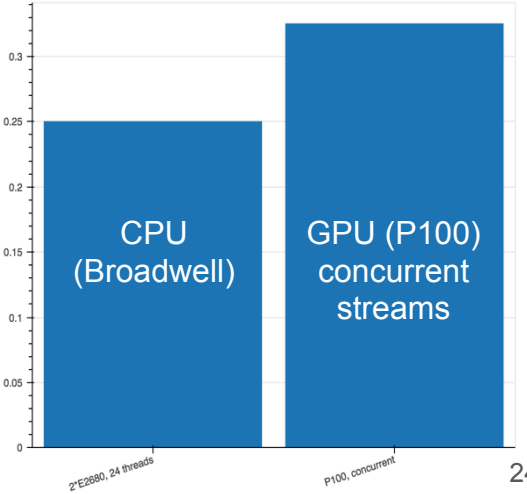
- Results from CtD 2017: the GPU code is slower than the optimized x86 version
  - Data transfers and global memory movements are penalizing



20 events, 10k tracks each

Things done since then:

- Process multiple events concurrently
  - Hide data transfers, fill the GPU better
- Detailed bottleneck study
  - Latency and memory dependency issues: why and where?
  - Profiling with nvprof (hard, no separate analysis by function).
  - Talking to NVidia guys.
- Result: 15x faster than CtD-17 (5s vs 0.33s)
  - x5 – code improvements + events in flight, x3 – K40 vs. P100



We are doing roofline analysis on a set of simple 6x6 matrix kernels to help us decide what to do next.

Conclusion

Conclusion

- mkFit is getting ready to be used in testing environment of CMS HLT
  - investigate inefficiency for low-pT tracks in high pileup data
  - implement post-build cleaning to reduce duplicate rate
  - improve scaling – optimization of code that was considered “out of scope” until now
    - hit preprocessing, seed cleaning, etc
- With all this, we are approaching our first production release.
  - Opportunity to do some deep cleaning of the code.
- Code is in principle quite general … but mkFit is not a ready to use tracking package
  - We will continue to make efforts in that direction.