Parallelized Kalman-Filter-Based Reconstruction of Particle Tracks on Many-Core Architectures

CMS R&D Tracking Projects --- Monday 16 Oct 2017, M. Tadel

G. Cerati[4], P. Elmer[3], 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. University of California – San Diego
2. Cornell University
3. Princeton University
4. Fermi National Accelerator Laboratory

Goal of the talk / Outline

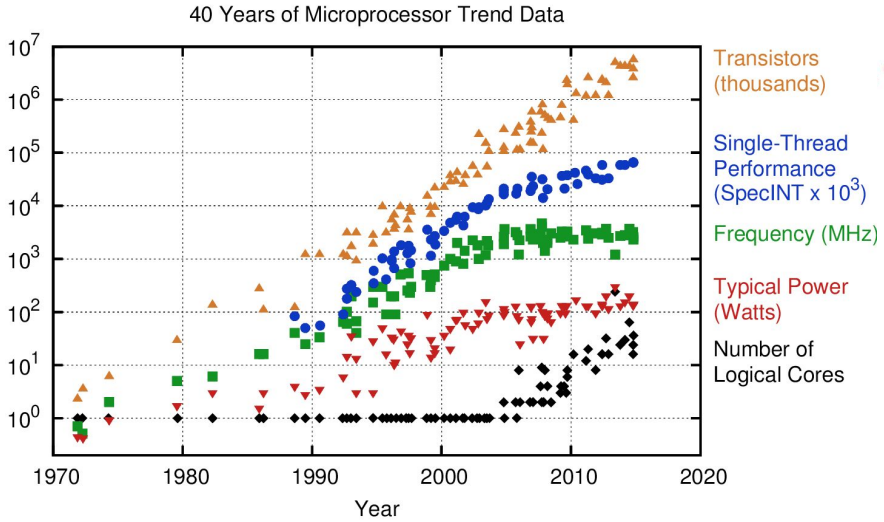
Main goal: Give an overview of what we are doing and how we are doing it

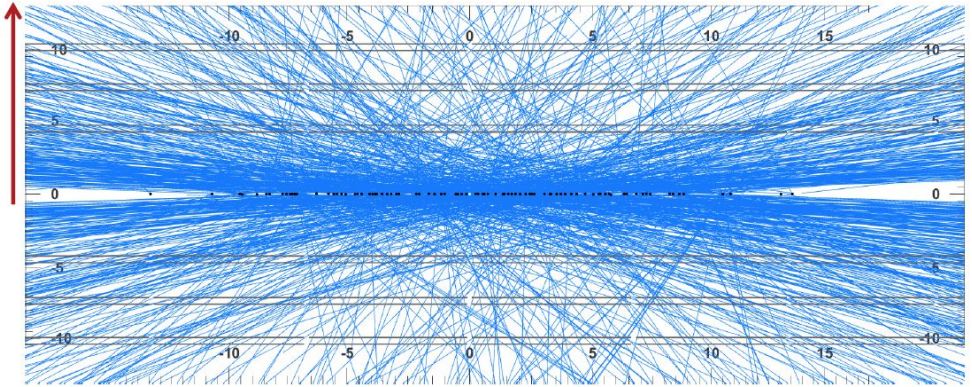
Do not focus on computational or physics performance. We plan to present detailed physics performance soon in the Tracking POG.

- Project introduction
  - Motivation for many-core Kalman filter implementation
- What we have done & tried so far
  - Geometries, event data
  - Algorithms & Data structures
  - Vectorization & Multi-threading
  - Architectures & Compilers
- Current focus & plans

Project overview

- Three institutions: Cornell, Princeton, UC San Diego (all CMS).
  - 3-year NSF grant, now in the final year
- 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





Kalman filter

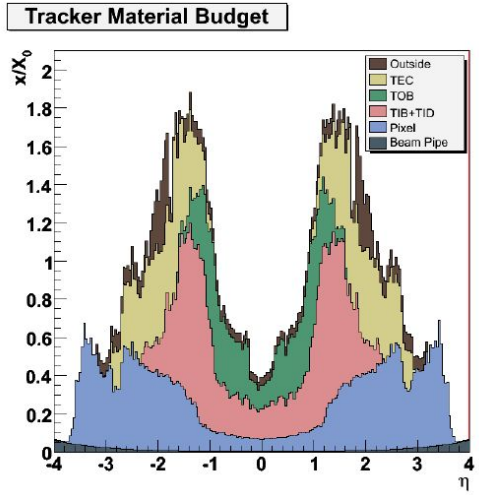
Why use Kalman filter:

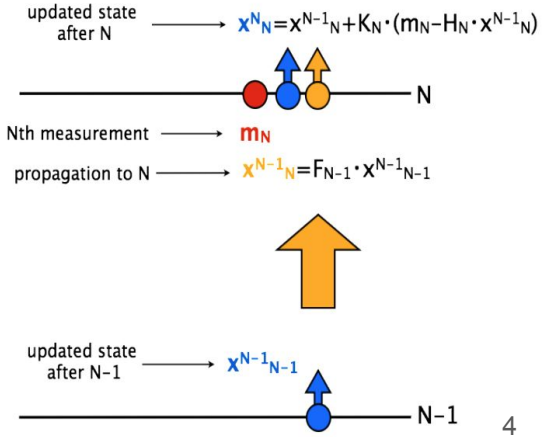
- Widely used & well understood
- Demonstrated physics performance:
  - Can handle multiple scattering and energy loss (badly needed)

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.





What we have done & tried so far

Tasks related to track finding

1. Seed finding: on ice for more than a year
   a. Now we either use CMSSW seeds or MC truth seeding for development
   b. For CMSSW CA seeds we do cleaning prior to track finding
2. **Track finding**: this is our primary focus. We have three main algorithms (b. is the reference implementation):
   a. *Best hit*: take the best hit on every layer
   b. *Standard*: on every layer check all hits, select N best candidates for each seed
   c. *Minimize copy*: apply cleverness to b. to reduce data copying and unnecessary cloning of Tracks

   Missing hits are added as long as the resulting candidates are viable.
3. **Track fitting**: secondary focus, is actually much easier
   a. Starting with a vector of known 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; currently it needs some TLC to catch up
4. **Validation**: a necessary side-theme that helps us stay on the straight & narrow
   a. Physics performance

Geometries & Events

- Current work is focusing on CMS-2017 geometry
  - We are using CMSSW generated events:
    - 10 muon events for development (barrel/endcap/transition region, low pT)
    - ttbar, ttbar + 70 PU
  - We use a simple event data format, basically a memory dump of our structures
    - CMSSW data is extracted from the Tracking Ntuple
      - hits, seeds, sim-tracks, reco-tracks (for phys. performance comparison)
  - *We can run track finding on full detector, iteration 0, with comparable physics performance.*
- Early on we developed a simple standalone tracker geometry (the Cow):
  - Early prototyping and development (Cylindrical Cow, barrel only)
  - Includes simulation with multiple scattering and energy loss
  - When starting the work on CMSSW, it “helped” us keep tracking algorithms independent of actual geometry! (Cylindrical Cow With Lids - includes endcap).

Geometry description & approximation

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

- 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.
- Requires additional propagation step for every hit matching!
  - But this really vectorizes well. [And we do not have to propagate to a module.]
- **Simplifies track steering code and minimizes candidate specific code.**

Geometry description:

- TrackerInfo: a vector of LayerInfo structures + basic parameters
- LayerInfo: r/z extent of the layer; hit binning parameters and hit search windows
  - For CMSSW, extents are determined from Tracking Ntuple using Sim hits

High level overview of track finding - Steering code

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

- Process seeds, assign them into regions: barrel, endcap +/-, transition +/-
- parallel_for every region
  - parallel_for over seeds from current region (typically in bunches of 16 or 32)
    - loop over layers, according to a LayerPlan 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

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 structure giving fast lookup of compatible hit indices
    - Does not vectorize well and is not cache friendly
      - Hit pre-selection is one of bottlenecks in track finding!
    - Use Radix sort on phi/z (barrel) or phi/r (endcap)
- 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

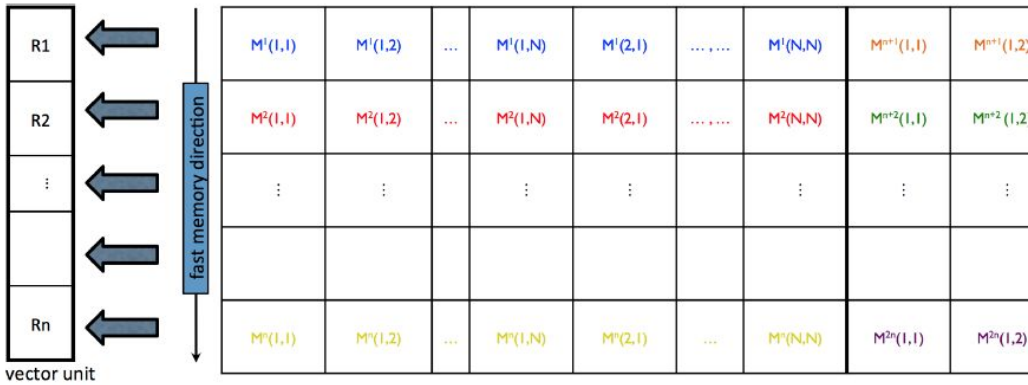
Matriplex - Vectorization of small matrix operations

"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 all Kalman filter related operations.

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

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!

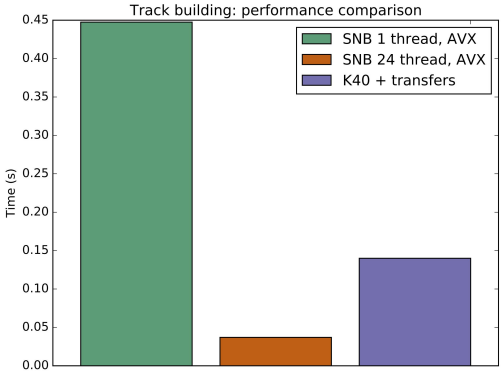
We actually started with OpenMP but it is hard to do dynamic problem partitioning. Also, TBB is good at task stealing.

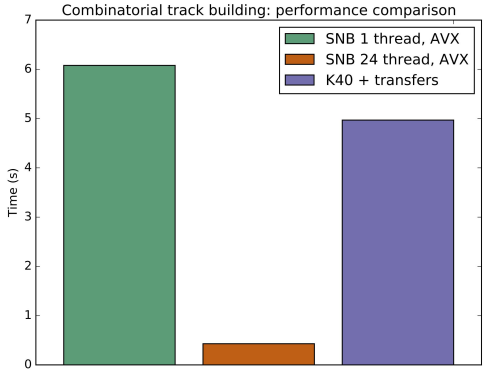
Architectures & compilers:

- 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)

Porting Track Reconstruction to GPU

- The strategy was to increase complexity by step
  - 1) Fitting; 2) Building only one track per seed; 3) Combinatorial building
  - Only ToyMC & barrel until performance gets better
- Memory-aligned data structures are critical to GPUs
  - Designed a “Matriplex”-like data structure for the GPU
- Global memory movements are penalizing
  - Try to avoid them by using a bookkeeping approach to avoid building irrelevant track candidates
- Data transfers slow things down
- Results from CtD 2017: the GPU code is slower than the optimized x86 version
  - Especially for the combinatorial version





Ongoing Improvements to the GPU Track Reco.

- Process multiple events concurrently
  - Hide data transfers, fill the GPU better
  - Close some gap with the x86 code
- More detailed bottleneck study
  - Latency and memory dependencies are an issue, why and where?
  - Profiling (with nvprof) a single kernel is hard, no separate analysis by function
- Eventually try a brute force approach
  - Building all track candidates with a satisfying goodness of fit
  - Only relevant if the bottleneck study shows the critical part to be the track candidate selection at the end of each layer

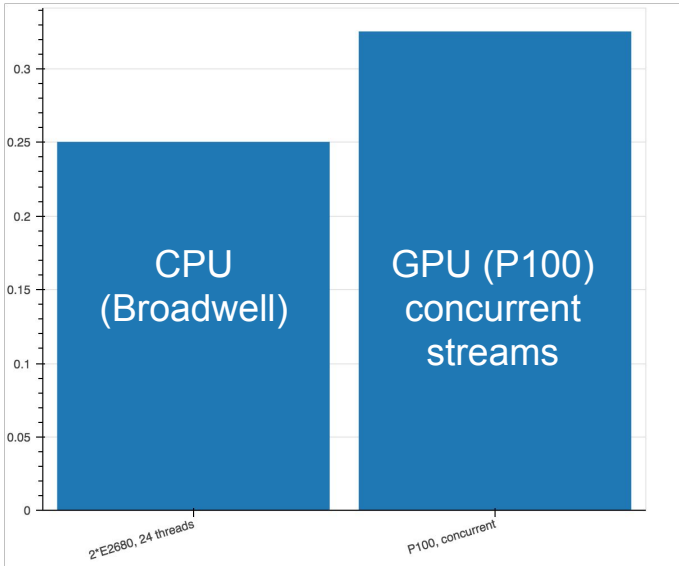


Fig. Time to reconstruct 20 events with 10k tracks each.

Current focus & Plans

What we are working on now and plan to do soon

- Meaningful comparison of track finding with CMSSW for Iteration 0
  - We are getting comparable physics performance already
    - Polishing the edges, will be ready to present soon!
  - Computational performance, i.e. speed, scaling, and memory footprint
    - x86_64, KNL
- Tuning of track finding parameters
  - Number of candidates to keep for each seed
  - Search window limits on every layer
- Consolidation of complete work-chain, including fitting
- Try out several ideas to further reduce memory operations during track finding

The End

Profiling tools

- VTune is our most frequently used tool to understand performance
  - Top-down view - functions that take the most of time
  - Bottom-up view / hotspots - pieces of code that can bring most significant improvements
    - Understanding of cache misses - this is really hard
      - Usually requires assembler view, good understanding of what is going on and some creativity
  - Vectorization performance (compiler's optimization reports can help here, too)
  - Also tried Intel Advisor with limited success
- Internal stopwatch
  - We used to measure time of “relevant section of code”, excluding parts we considered
  - This is becoming increasingly useless with
    - a) multiple events in flight
    - b) comparison against CMSSW
- Wallclock an perf