Performance Case Study: the mkFit Particle Tracking Code

Steve Lantz, Cornell University

CoDaS-HEP Summer School, July 24, 2024

High Performance Computing in High Energy Physics

Collaborators

K. McDermott, G. Niendorf, M. Reid, D. Riley, P. Wittich (Cornell);
S. Berkman, G. Cerati, P. Gartung, M. Kortelainen (Fermilab);
B. Wang (NVIDIA);
P. Elmer (Princeton);
L. Giannini, S. Krutelyov, M. Masciovecchio, M. Tadel, E. Vourliotis, F. Würthwein, A. Yagil (UCSD);
B. Gravelle, B. Norris (U. Oregon);
A. R. Hall (USNA).

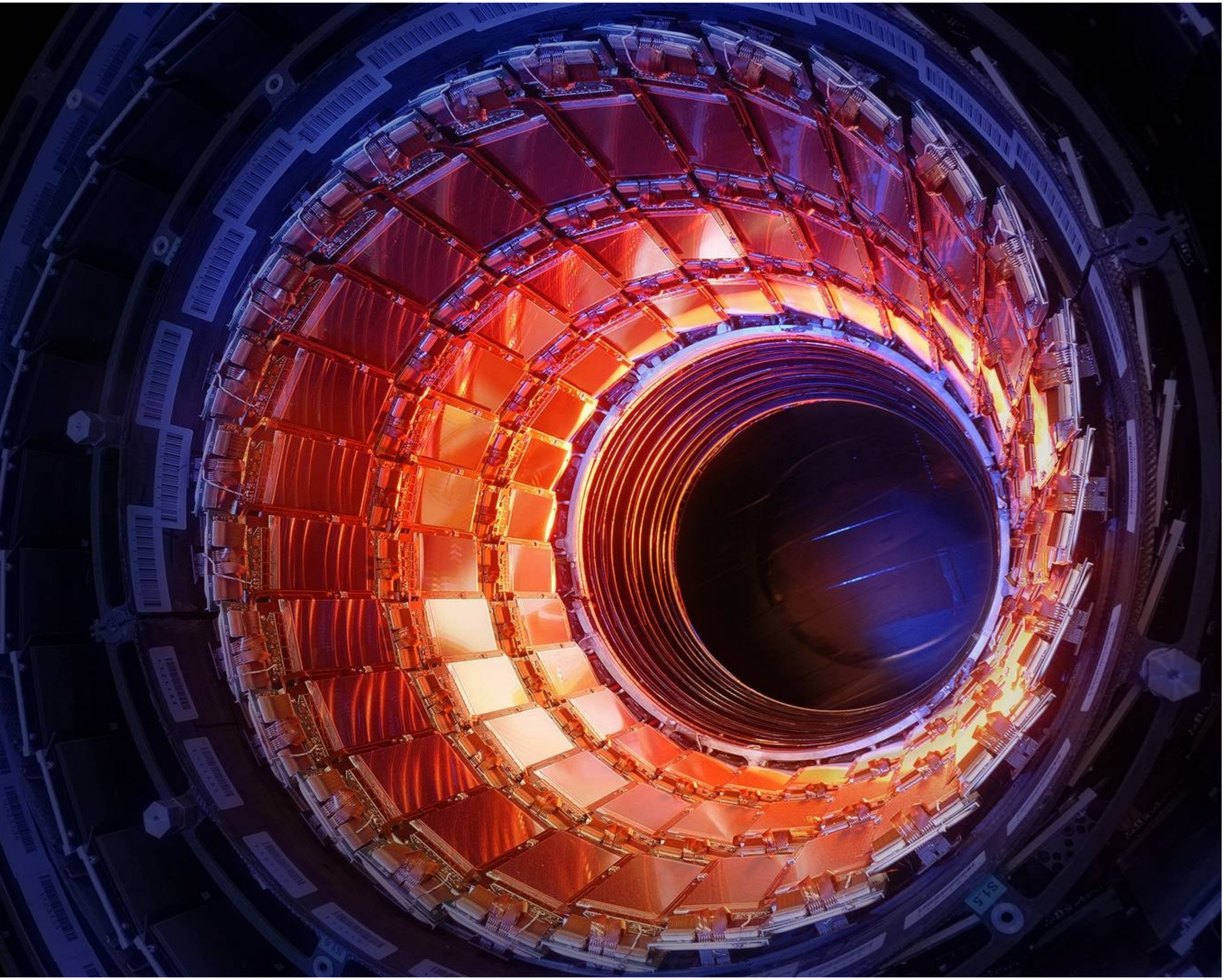


Photo: CMS detector, LHC, CERN

Key reference: S. Lantz et al., *J. Inst.* 15 P09030 (2020)

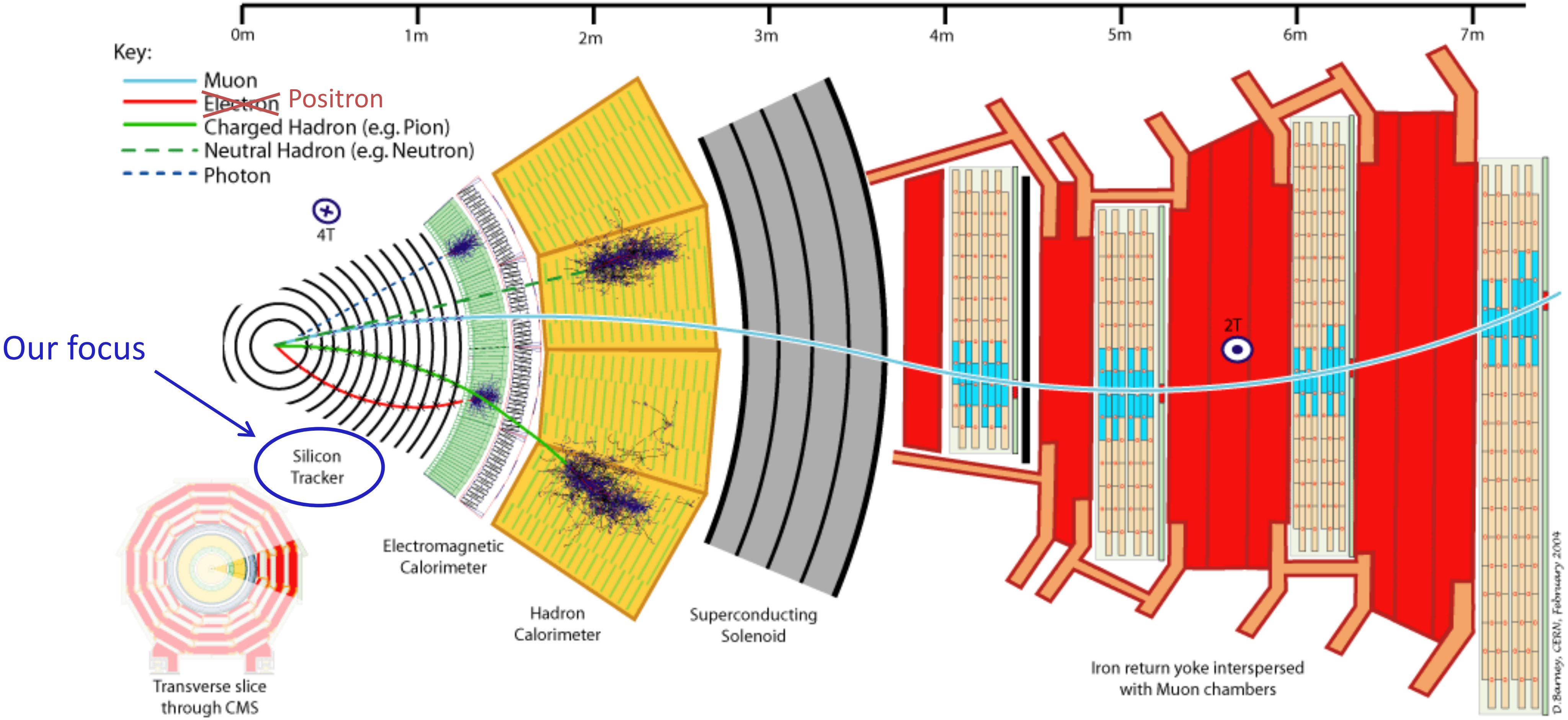
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

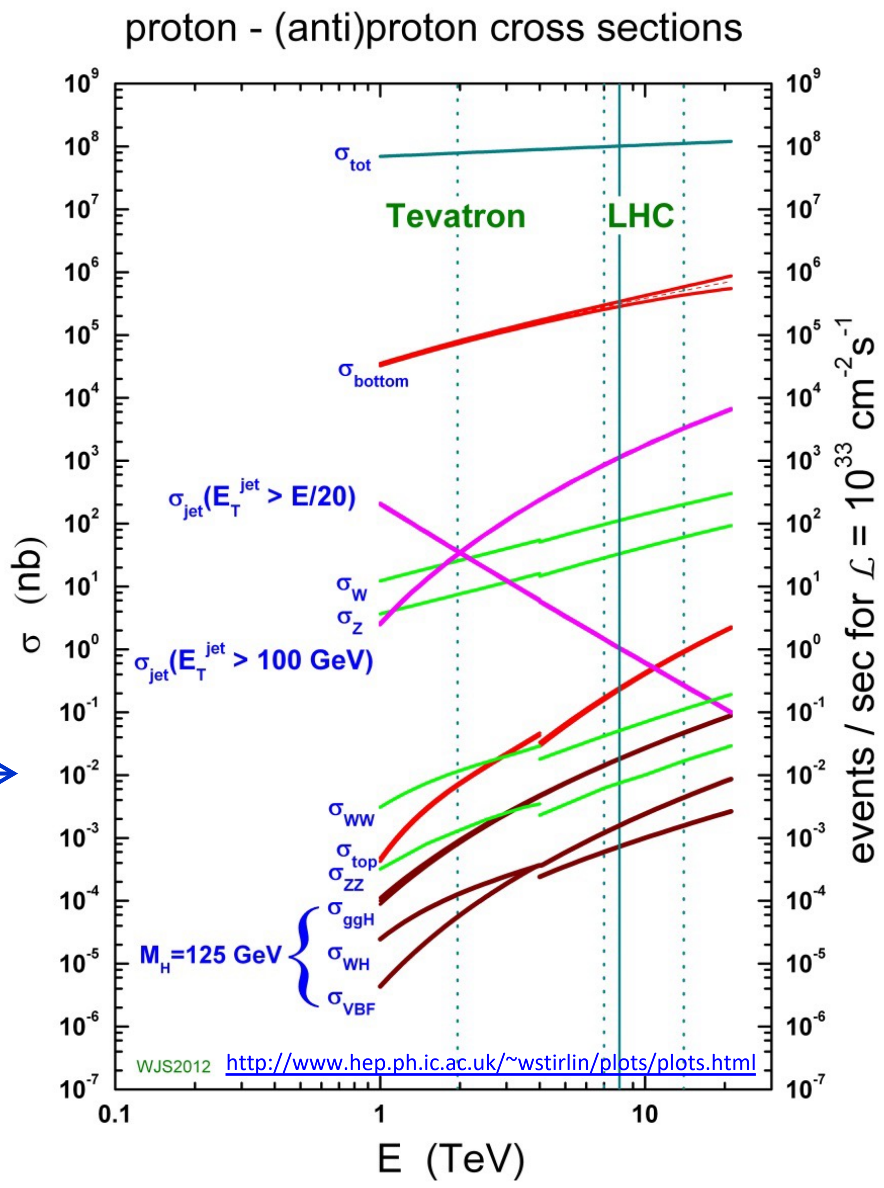
CMS: Like a Fast Camera for Identifying Particles



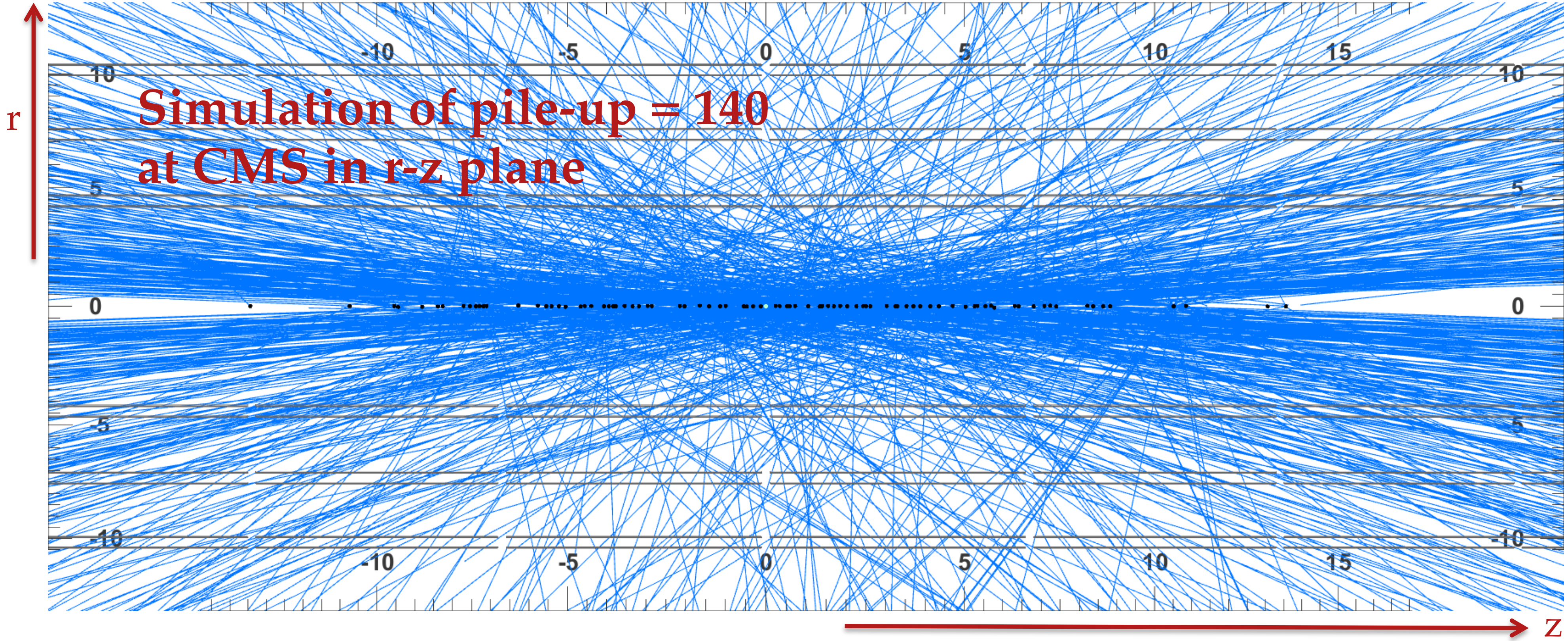
Particles interact differently, so CMS is a detector with different layers to identify the decay remnants of Higgs bosons and other unstable particles

Big Data Challenge

- *40 million* collisions a second
- Most are boring
  - Dropped within 3 μs
- 0.5% are interesting
  - Worthy of reconstruction...
- Higgs events: *super* rare
  - 10^{16} collisions → 10^6 Higgs
  - Maybe 1% of these are found
- Ultimate "needle in a haystack"
- "Big Data" since before it was cool



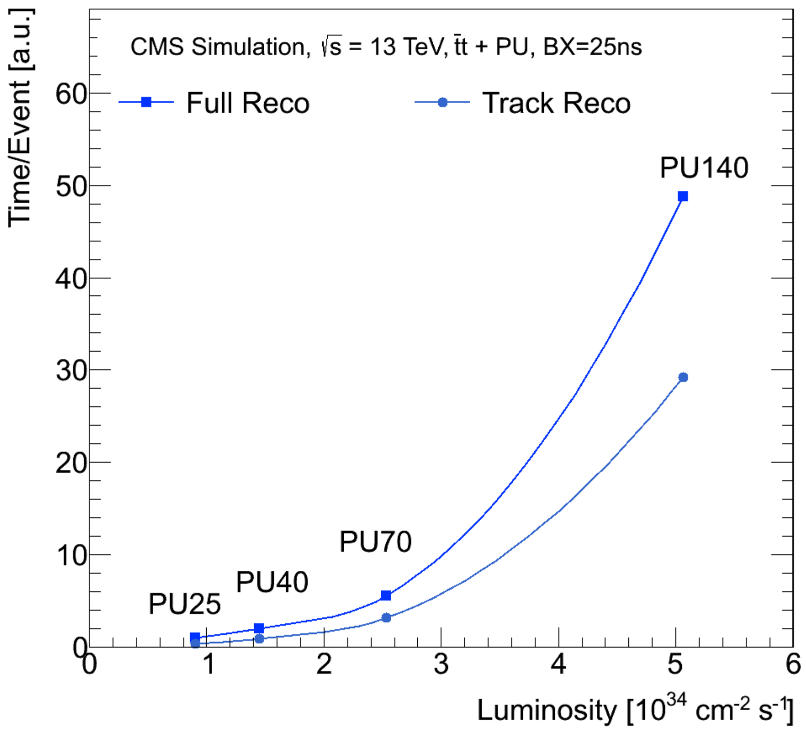
CMS Is About to Get Busier



- By ~~2025~~ 2029, the instantaneous luminosity of the LHC will increase by a factor of 2.5, transitioning to the High Luminosity LHC (HL-LHC)
- Significant increase in number of interactions per bunch crossing, i.e., "pile-up", on the order of 140–200 interactions per *event*

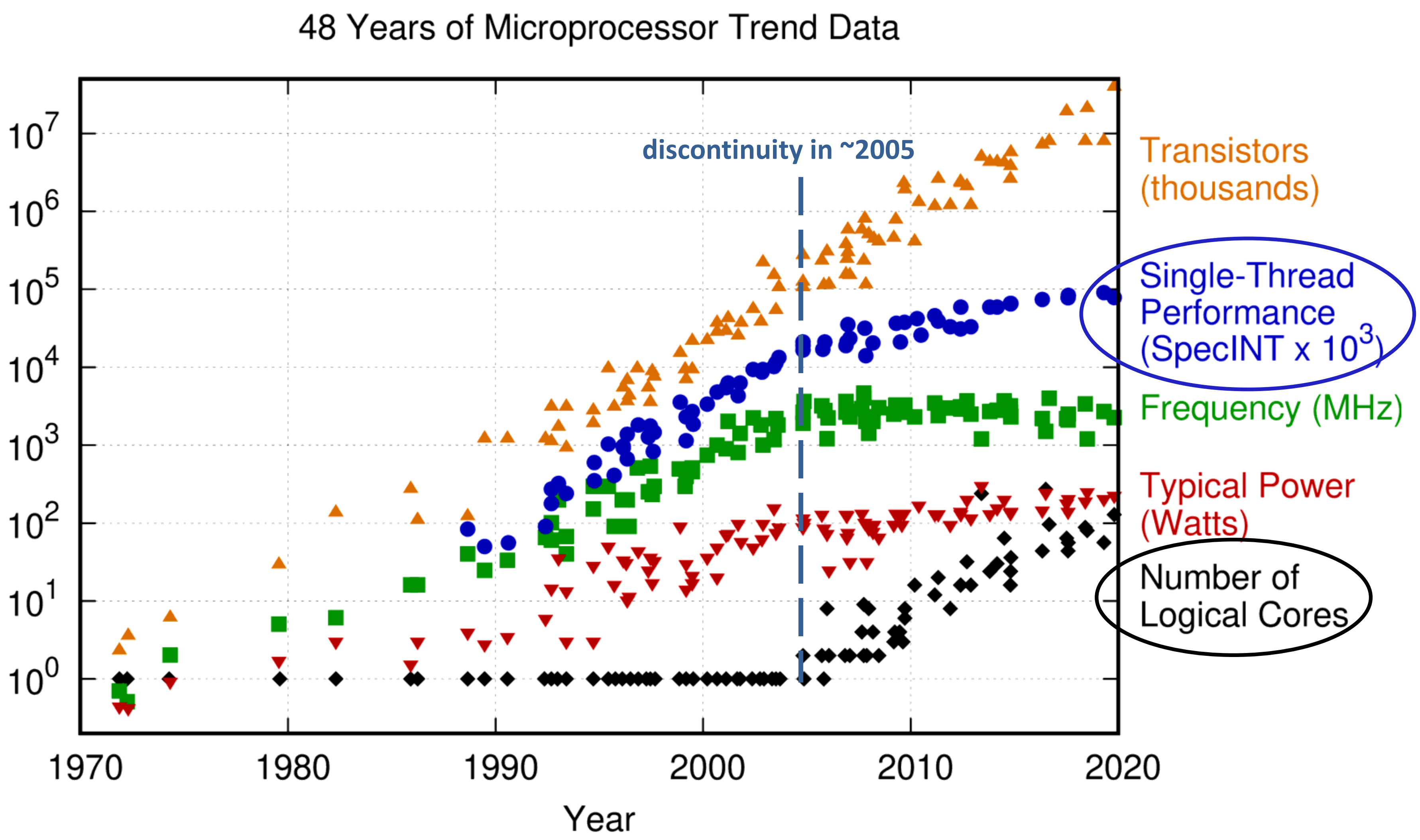
Reconstruction Will Soon Run Into Trouble

- Higher detector occupancy puts a strain on read-out, selection, and event ***reconstruction***
- A slow step in reconstruction is combining ~10^6 energy deposits ("hits") in the tracker to form charged-particle trajectories – ***tracking***
- Tracking is typically the biggest contributor to reconstruction time per event in CMS, and for high pile-up, it ***diverges***



- We can no longer rely on Moore's Law scaling of CPU frequency to keep up with growth in reconstruction time – we need a new solution
- Can we make the tracking algorithm ***concurrent*** to gain speed?

Overview of CPU Speed and Complexity Trends



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2019 by K. Rupp [GitHub link]

Two Types of Intra-Processor Parallelism

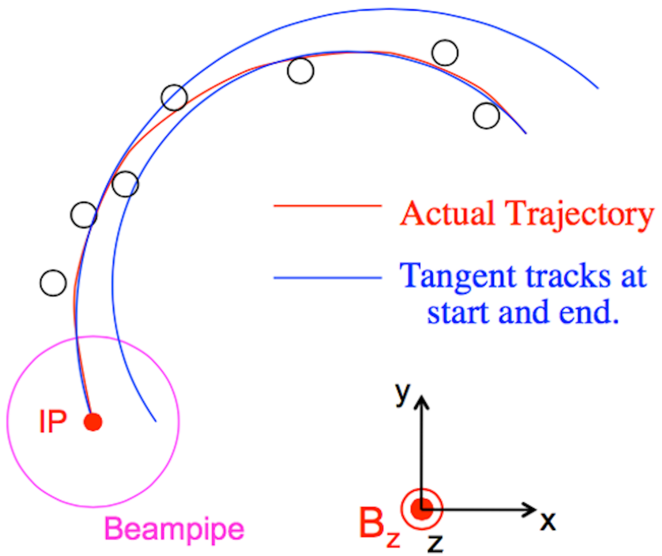
- **Vectorization (data parallelism)**
  - "Lock step" Instruction Level Parallelization: SIMD = Single Instruction, Multiple Data
  - Requires minimization of branching and efficient memory utilization
  - It's all about finding simultaneous operations, on well-aligned data
- **Multithreading (task parallelism)**
  - OpenMP, Threading Building Blocks, Pthreads, etc., to use multiple cores
  - It's all about sharing work and balancing the load, with minimal overhead
- To occupy a processor fully, both types need to be identified and addressed
  - Vectorized loops (not the whole code) gain 8x or 16x performance on CPUs
  - Multithreading offers a further Mx speedup on M cores
- Prior tracking algorithms did not do this at the *event* level—can we? (How?)

Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

What Does the Tracking Algorithm Do?

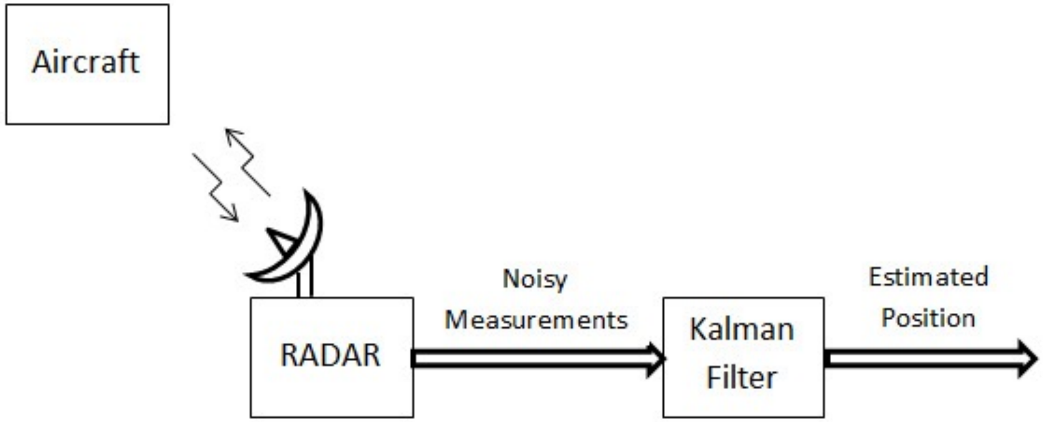
- Goal is to reconstruct the trajectory (track) of *each* charged particle
- Solenoidal B field bends the trajectory in one plane (“transverse”)
- Trajectory is a helix described by 5 parameters, p_T, η, φ, z_0, d_0
- We are most interested in high-momentum (high-p_T), low-curvature tracks
- But trajectories may change due to interaction with materials…
- Ultimately we care mainly about:
  - ***Initial track parameters***
  - ***Exit position to the calorimeters***



- ***Kalman Filter is well suited for this job***

Kalman Filter

- Method for obtaining best estimate of the parameters of a trajectory
- For particle tracking: a natural way of including interactions in the material (process noise) and hit position uncertainty (measurement error)
- Used both in *pattern recognition* (e.g., track building, determining which hits belong to the track of one particle) and in *fitting* (e.g., determining the ultimate track parameters)



Kalman filter

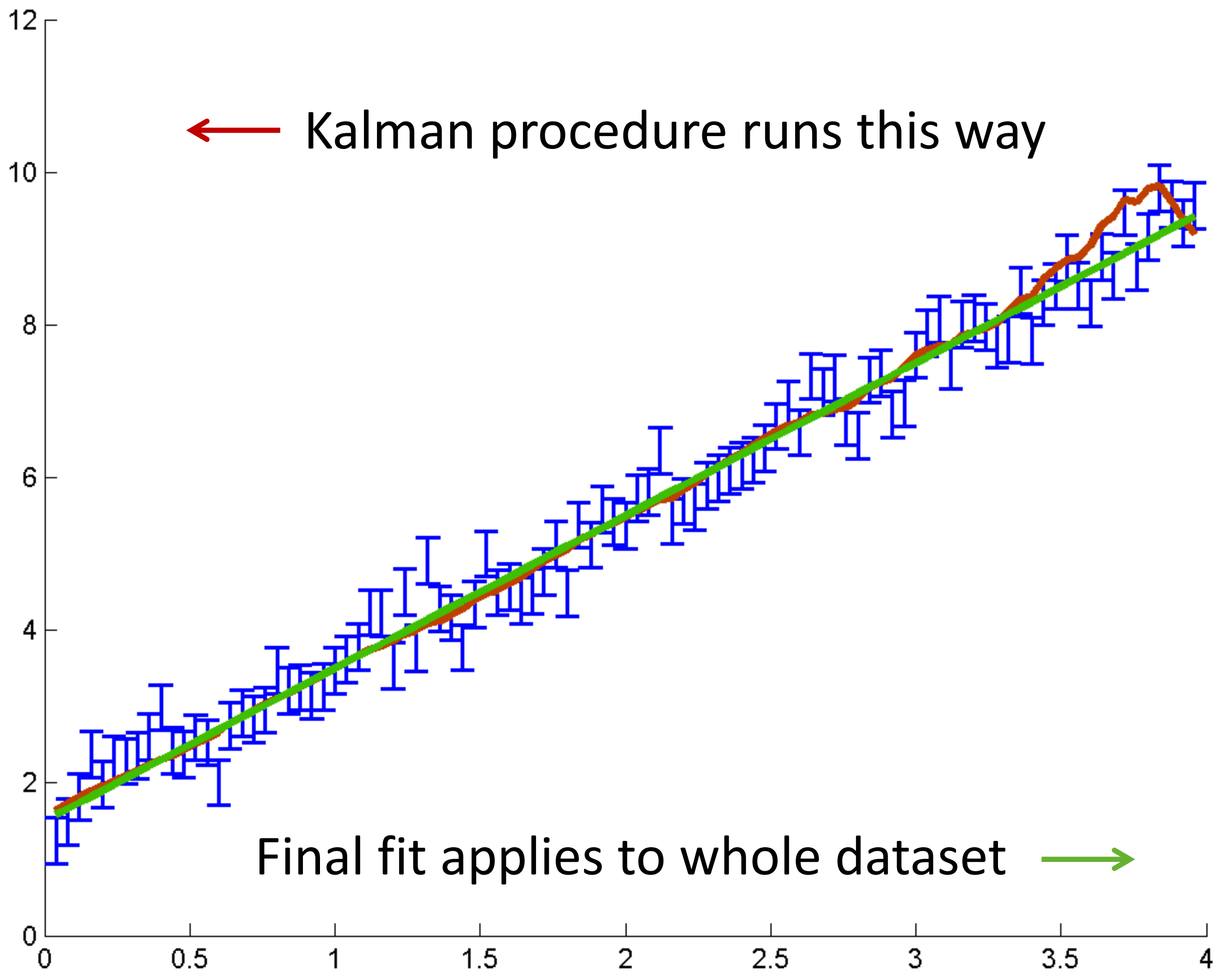
From Wikipedia, the free encyclopedia

Kalman filtering, also known as **linear quadratic estimation** (**LQE**), is an algorithm that uses a series of measurements observed over time, containing noise (random variations) and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone. More formally, the Kalman filter operates recursively on streams of noisy input data to produce a statistically optimal estimate of the underlying system state. The filter is named after Rudolf (Rudy) E. Kálmán, one of the primary developers of its theory.

R. Frühwirth, *Nucl. Instr. Meth. A* **262**, 444 (1987), DOI:10.1016/0168-9002(87)90887-4; http://www.mathworks.com/discovery/kalman-filter.html

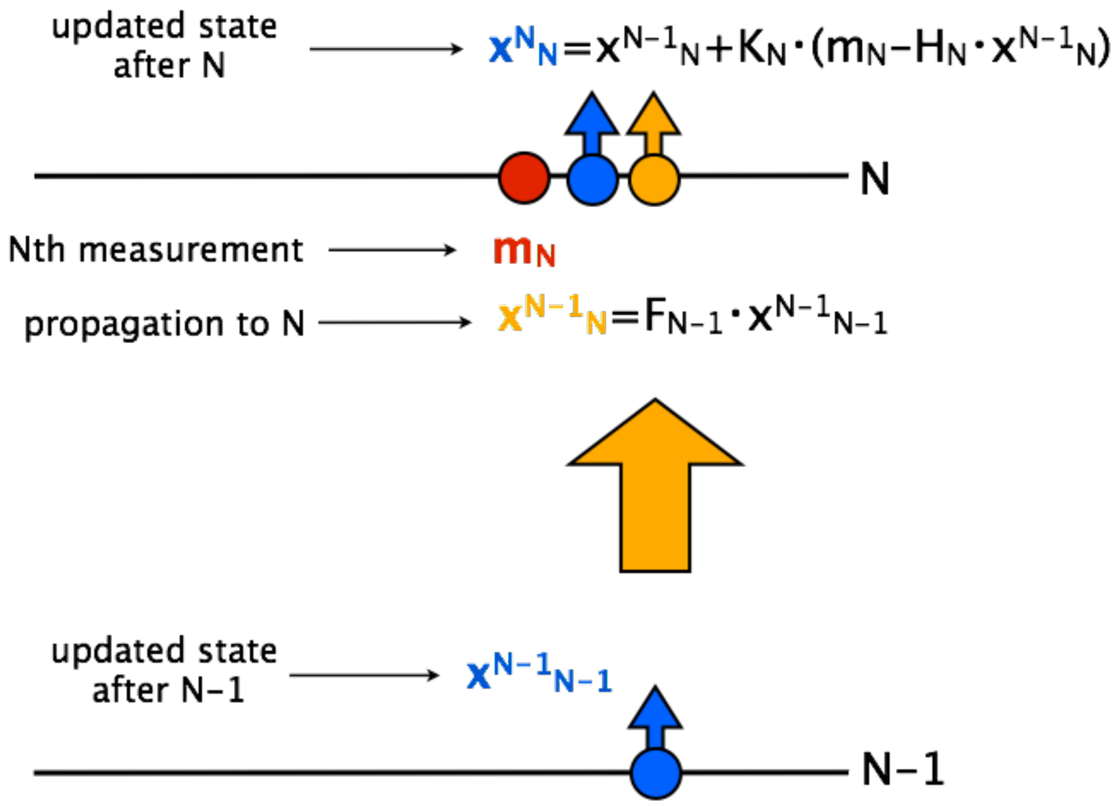
Kalman Example

- Use Kalman procedure to estimate slope and y-intercept of a straight-line fit to noisy data
- Parameter values improve as data points are added
- 30-line script in MATLAB



Tracking as Kalman Filter

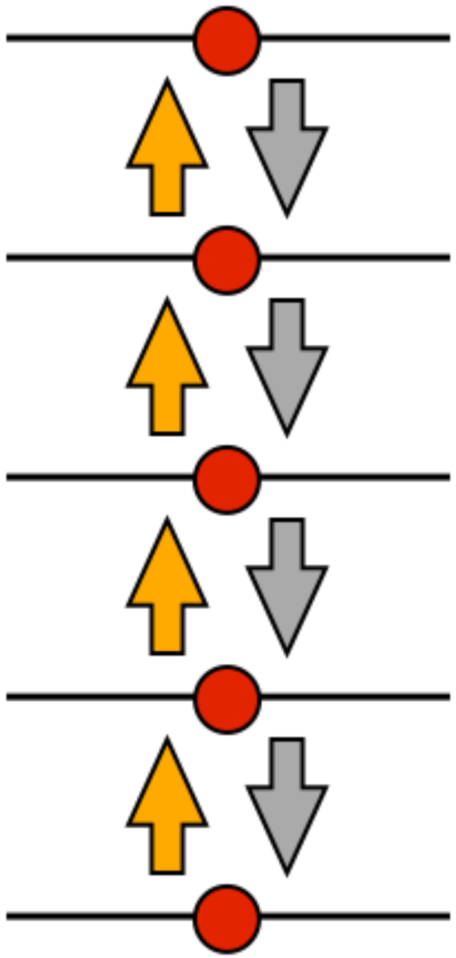
- Track reconstruction has 3 main steps: *seeding*, *building*, and *fitting*
- Building and fitting repeat the basic logic unit of the Kalman Filter...



- From current *track state* (parameters and uncertainties), track is *propagated* to next layer
- Using hit measurement data, track state is *updated (filtered)*
- Amount of correction is inversely weighted by hit uncertainty
- Procedure is repeated until last layer is reached

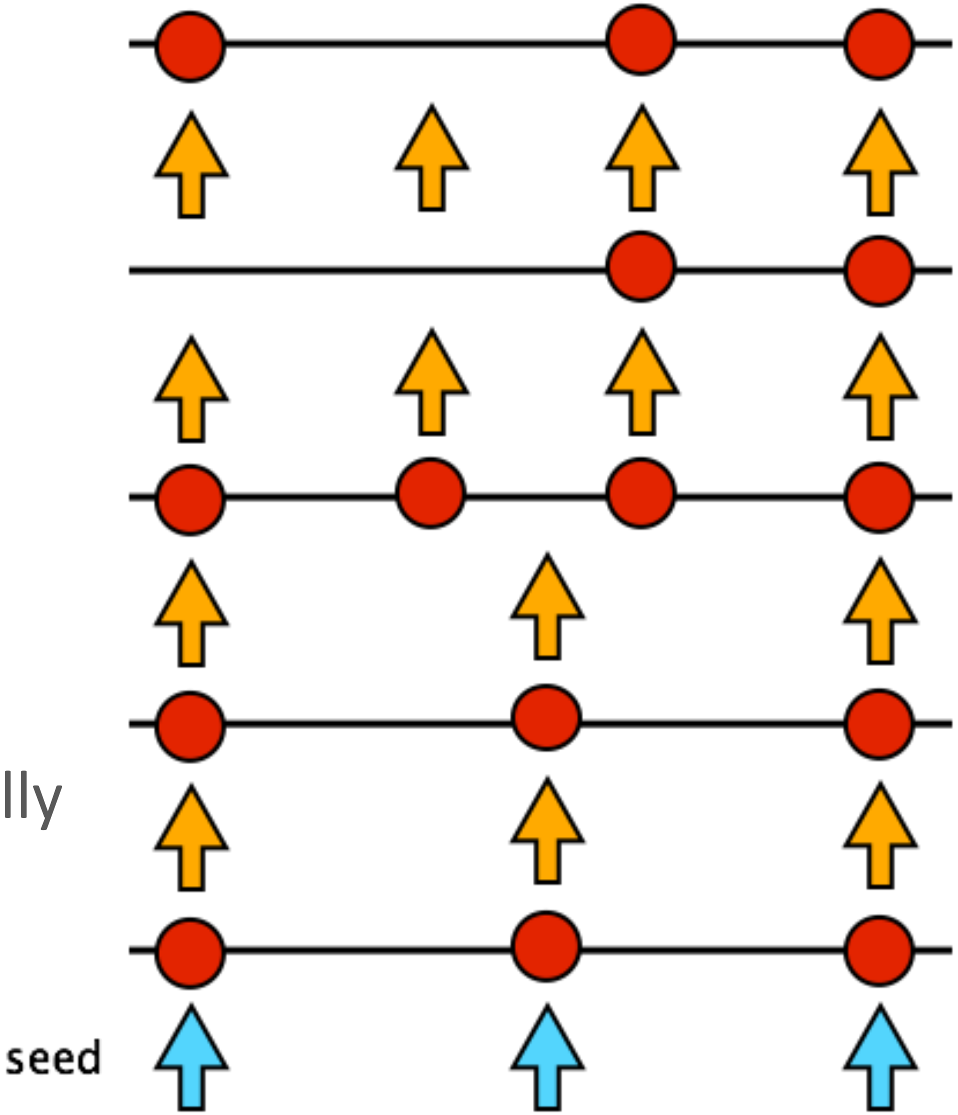
Track Fitting as Kalman Filter

- The track *fit* consists of the simple repetition of the basic logic unit for hits that are *already determined* to belong to the same track
- Divided into two stages
  - Forward fit: best estimate at collision point
  - Backward smoothing: best estimate at face of calorimeter
- Computationally, the Kalman Filter is a sequence of matrix operations with *small matrices* (dimension 6 or less)
- But, every single track can be fit *in parallel*



Track Building

- Building is harder than fitting!
- After propagating a track candidate to the next layer, hits are searched for within a compatibility window
- Track candidate needs to ***branch*** in case of multiple compatible hits
  - The algorithm needs to be robust against missing/outlier hits
- Due to branching, track building has typically been the ***most time-consuming step*** in event reconstruction, by far



Parallelization Plan for CPUs

1. Partition the tracks (or track candidates) into SIMD-size bunches
   – Assign bunches to different CPU threads
   – Try to vectorize operations within each bunch
2. Propagate bunches to next detector layer
   – Rely on automatic vectorization by compiler, here
   – Costliest part: computing derivatives for error propagation
3. Select one or more compatible hits in the layer (building only)
   – This is hard! Depends on space-partitioning the data structures containing hits
   – Combinatorial explosion! Need to cap the number of track candidates per seed
4. Perform Kalman updates on track parameters and errors
   – But auto-vectorization doesn't work well for small matrices… **must focus efforts here**

```
# multithread this loop...
For b in [ bunches ]
  #pragma omp simd
  for t in [ track bunch b ]
    # ~80 lines of calculations
```

Outline

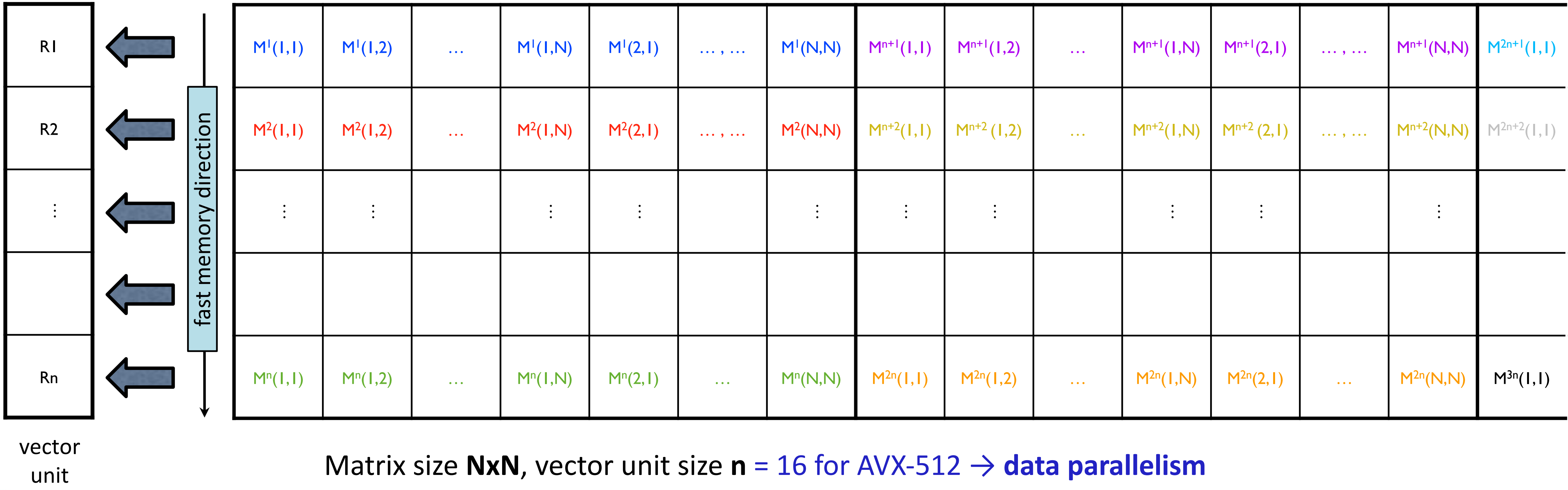
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

Matriplex – The Key Idea

- Nearly impossible to vectorize small matrix/vector ops individually
  - Many multiplications and additions, but pattern of access and operations is inconsistent
- Expand identical operations by doing V_W (8 or 16) matrices simultaneously!
  - **Matriplex** is a library that helps you do it in optimal fashion
  - Effectively, creates V_W-way SIMD operations from V_W matrix multiplications
  - Input data are repacked so that loading vector registers is trivial
- But vectorization hardly matters if the data aren't in cache memory…
  - Best if all matrices are present in L1 data cache together (L1d size: 32-64 kB)
  - Can be done, but puts pressure on both cache and registers
    - » 6x6 floats * 4 Bytes * 3 operands * 8 = 3456 Bytes
    - » 6x6 floats * 4 Bytes * 3 operands * 16 = 6912 Bytes

Matriplex Structure for Kalman Filter Operations

- Store in "matrix-major" order so 16 matrices work in sync (SIMD)
  - Potential for 60 vector units in Intel Xeon SP to work on 960 tracks at once!
  - Each individual matrix is small: 3x3 or 6x6, and may be symmetric



Matrix size **NxN**, vector unit size **n** = 16 for AVX-512 → **data parallelism**

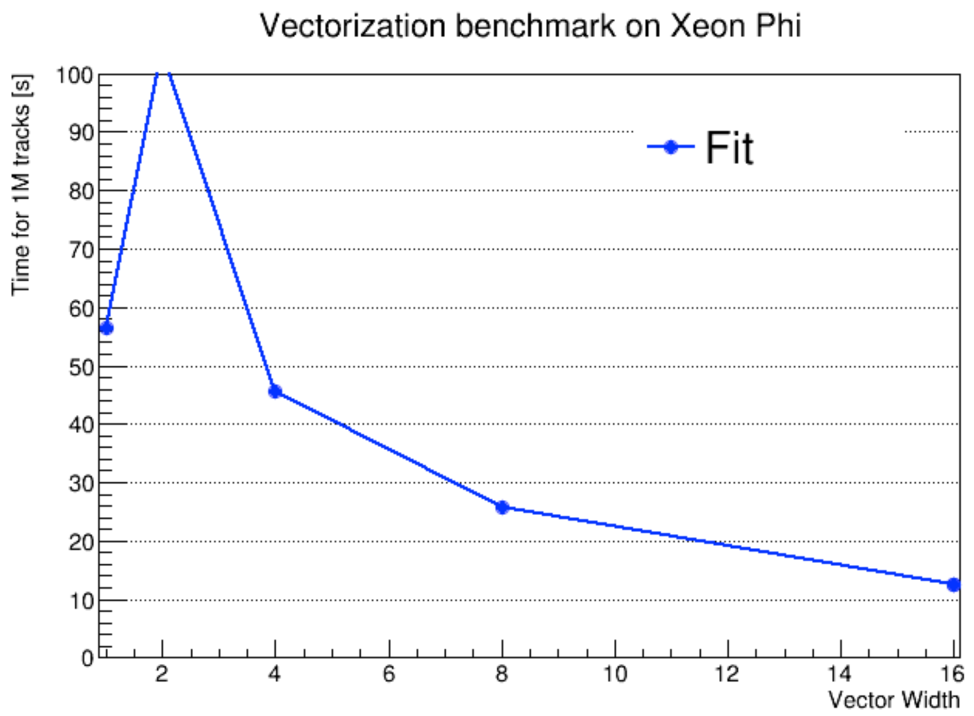
Outline

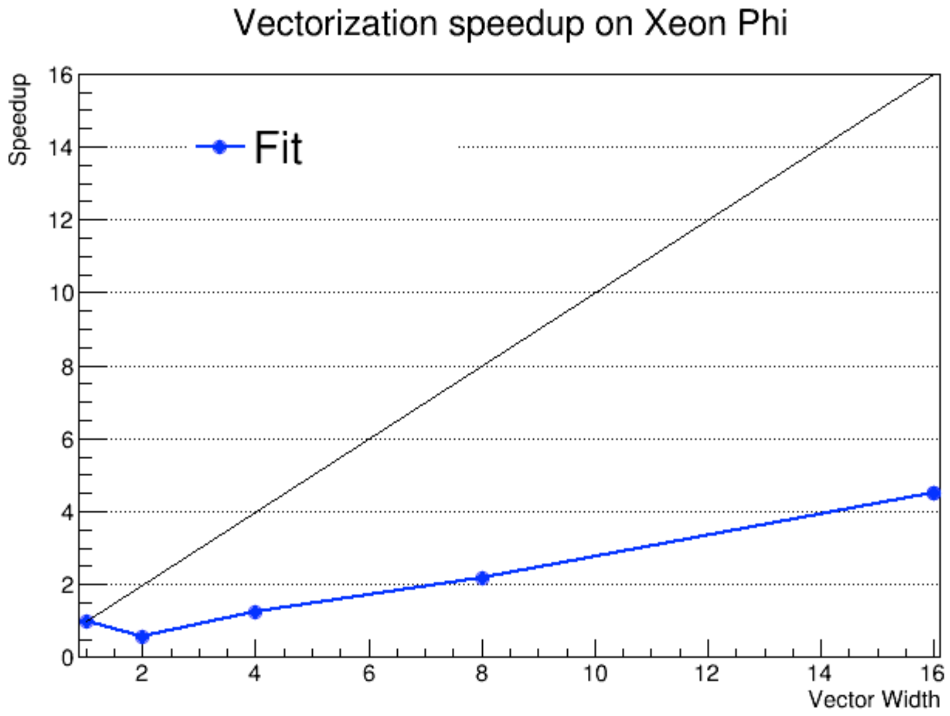
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

Vector-Aware Coding and Performance Tuning

- Know what makes codes vectorizable at all
  - The “for” loops (C) or “do” loops (Fortran) that meet constraints
- Know where vectorization ought to occur
- Arrange vector-friendly data access patterns (unit stride)
- Study compiler reports: do loops vectorize as expected?
- Implement fixes: directives, compiler flags, code changes
  - Remove constructs that hinder vectorization
  - Encourage/force vectorization when compiler fails to do it
  - Engineer better memory access patterns
- Turn to performance tools, if further speedup is desired

Initial Speed Test of Track Fitting in a Simplified Detector





- Fit benchmark: average of 10 events, 10^6 tracks each, single thread
- Matriplex width varies from 1 (quasi-unvectorized) to 16 (full)
- Maximum speedup is only ~4.4x. What's wrong?

Clues from Intel Advisor

General Exploration General Exploration viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	Clockticks	Instructions Retired	CPI Rate	Start Address	Vectorization Usage: Vectorization Intensity	L1 C...	L2 ...
helixAtRFromIterative	5,320,000,000	2,240,000,...	2.375	0x4376b0	9.826	25.393	
Matriplex::MatriplexSym<float, (int)6, (int)16>::Subtract	1,330,000,000	630,000,000	2.111	0x40e24a	0.889	0.964	
__intel_lrb_memcpy	840,000,000	490,000,000	1.714	0x48ac40	6.000	7.500	
Matriplex::MatriplexSym<float, (int)3, (int)16>::CopyIn	700,000,000	630,000,000	1.111	0x423b46	0.000	0.000	0.000
updateParametersMPlex	630,000,000	490,000,000	1.286	0x40d550	10.000	5.882	
(anonymous namespace)::MultHelixProp	630,000,000	350,000,000	1.800	0x43de40	7.000	14.737	
Matriplex::Matriplex<float, (int)3, (int)1, (int)16>::CopyIn	560,000,000	140,000,000	4.000	0x423b4c	0.000	0.000	0.000
(anonymous namespace)::PolarErr	560,000,000	0		0x40f720	6.500	21.667	
MkFitter::InputTracksAndHits	490,000,000	140,000,000	3.500	0x423830	0.000	0.000	0.000
Matriplex::MatriplexSym<float, (int)6, (int)16>::CopyIn	420,000,000	490,000,000	0.857	0x4238db	0.000	0.000	0.000
MkFitter::FitTracks	420,000,000	70,000,000	6.000	0x424c70		6.667	

- Taking lots of time in routines that are unvectorized (or nearly so)
- Ideal vectorization intensity should be 16
- Subtract and CopyIn appear to be the top offenders

More Clues From Optimization Reports

- Intel compilers have an option to generate vectorization reports
- One report showed a problem in a call to a Matriplex method…

```
remark #15344: loop was not vectorized: vector dependence
prevents vectorization. First dependence is shown below...

remark #15346: vector dependence: assumed FLOW dependence
between outErr line 183 and outErr line 183
```

```
outErr.Subtract(propErr, outErr);
```

- OK! – so outErr (a reference) is both input and output. But we know that is totally safe, because Subtract just runs element-wise through fArray
- Compiler must often make conservative assumptions by default

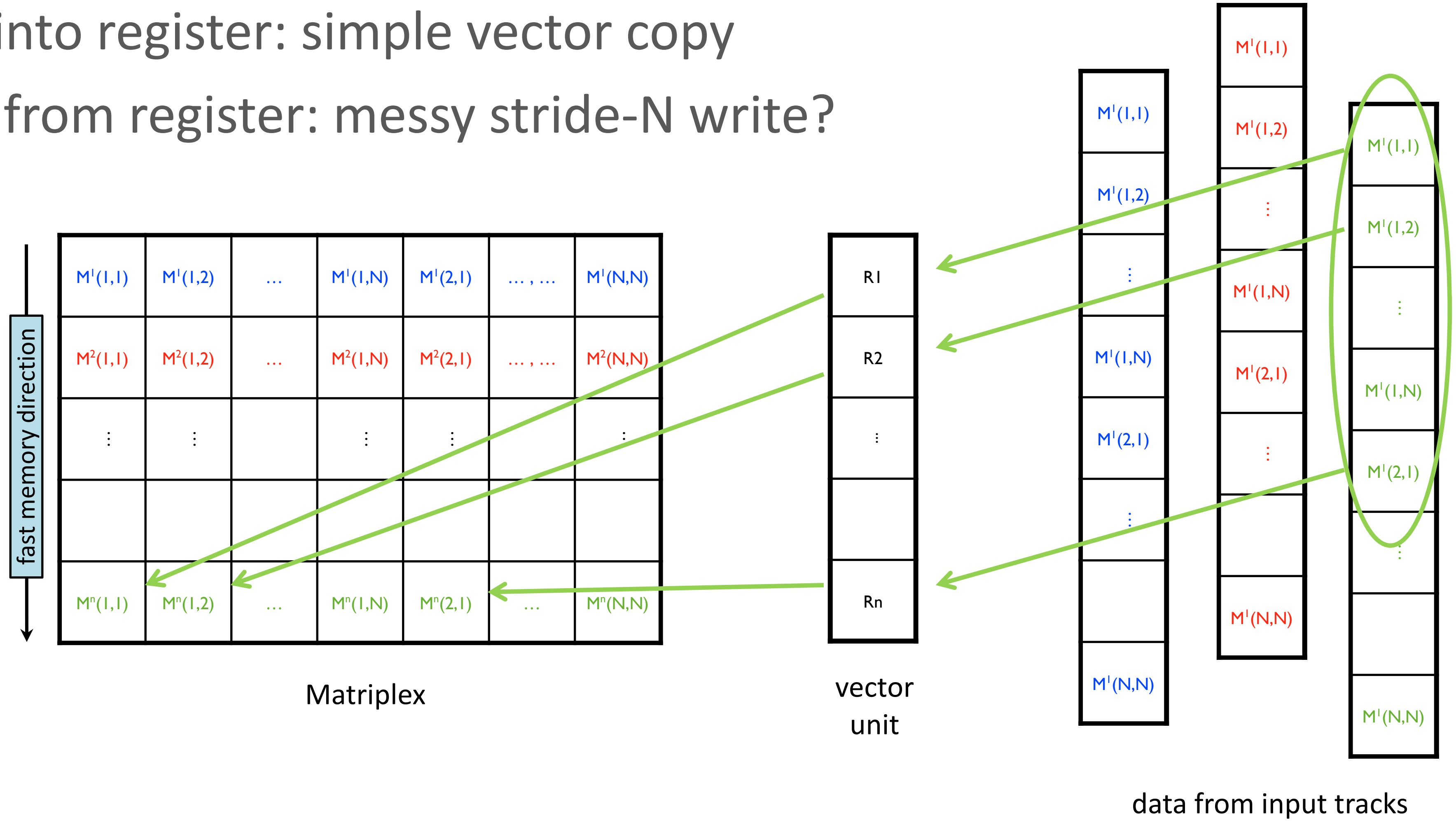
Fixing the False Loop-Carried Dependence

- Just add a pragma to **i**gnore **v**ector **dep**endence
  - Later this was changed to the even stronger `#pragma omp simd`
- Single change gave ~10% performance gain! (at full vector width)

```
MatriplexSym& Subtract(const MatriplexSym& a,
                       const MatriplexSym& b)
{
#pragma ivdep
   for (idx_t i = 0; i < kTotSize; ++i)
   {
      fArray[i] = a.fArray[i] - b.fArray[i];
   }
}
```

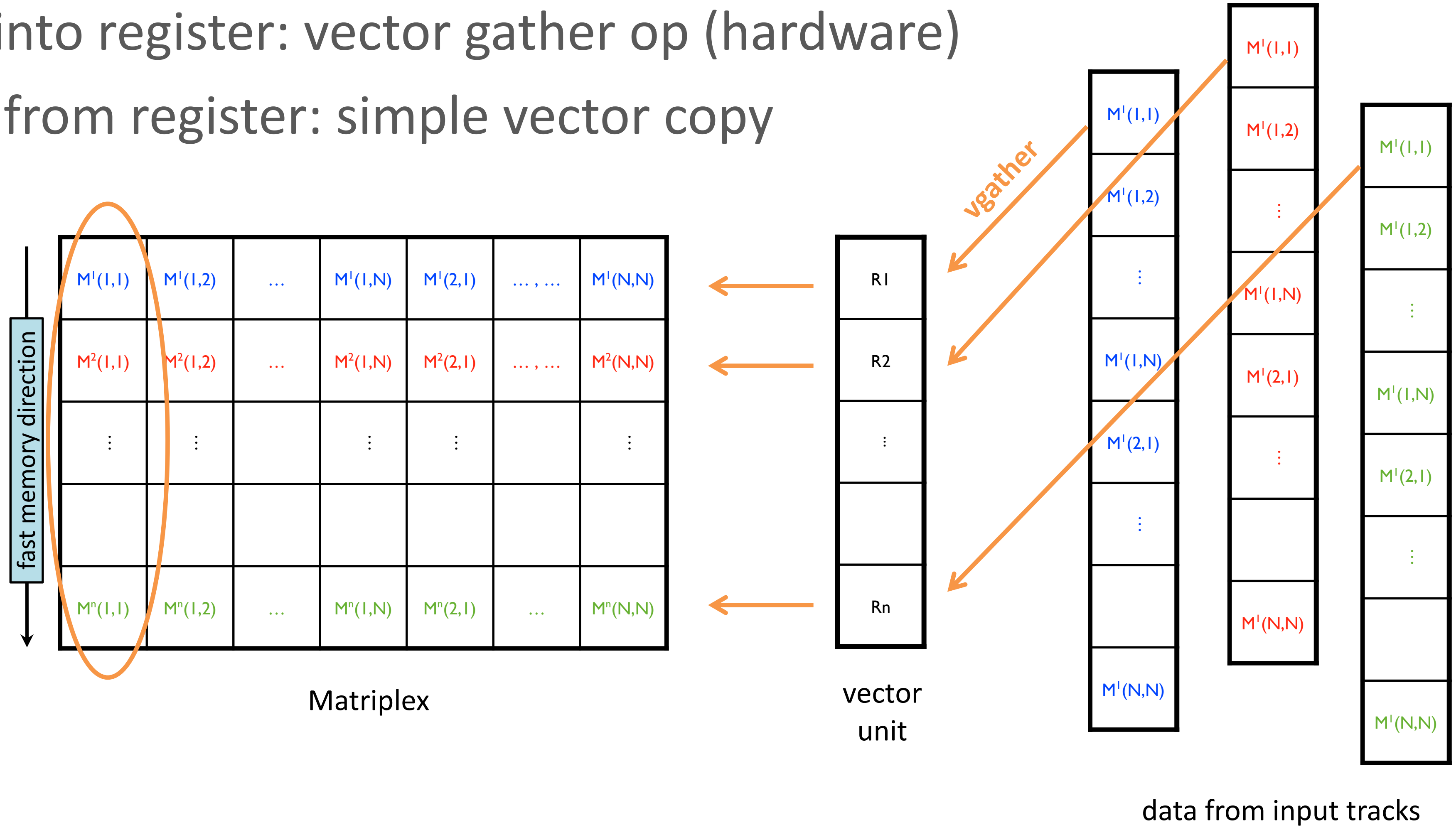
CopyIn: Initialization of Matriplex from Track Data

- Load into register: simple vector copy
- Store from register: messy stride-N write?

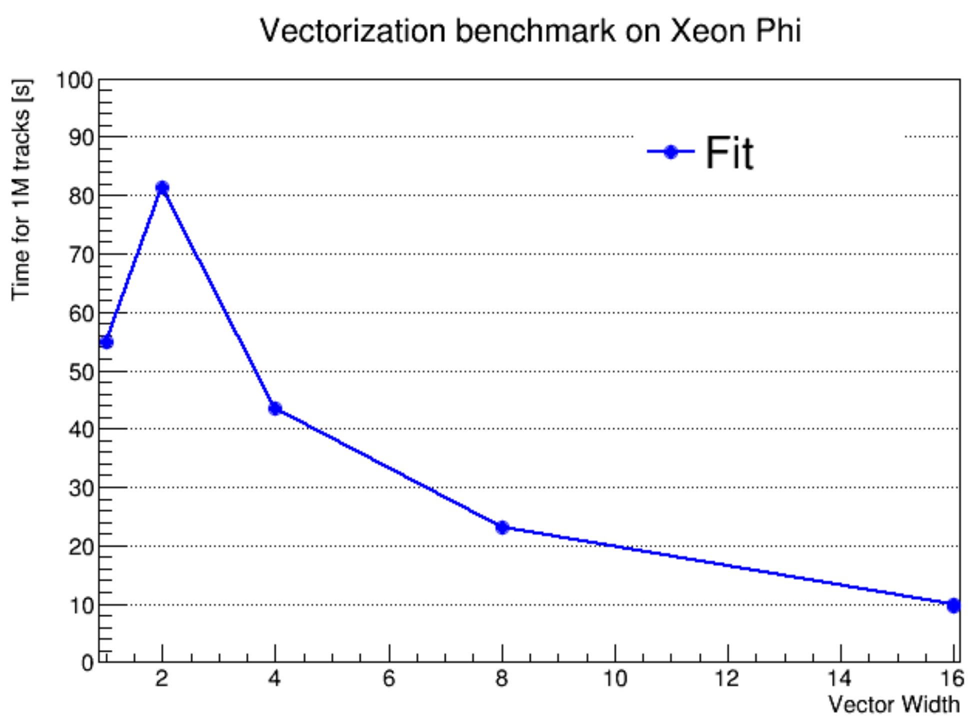


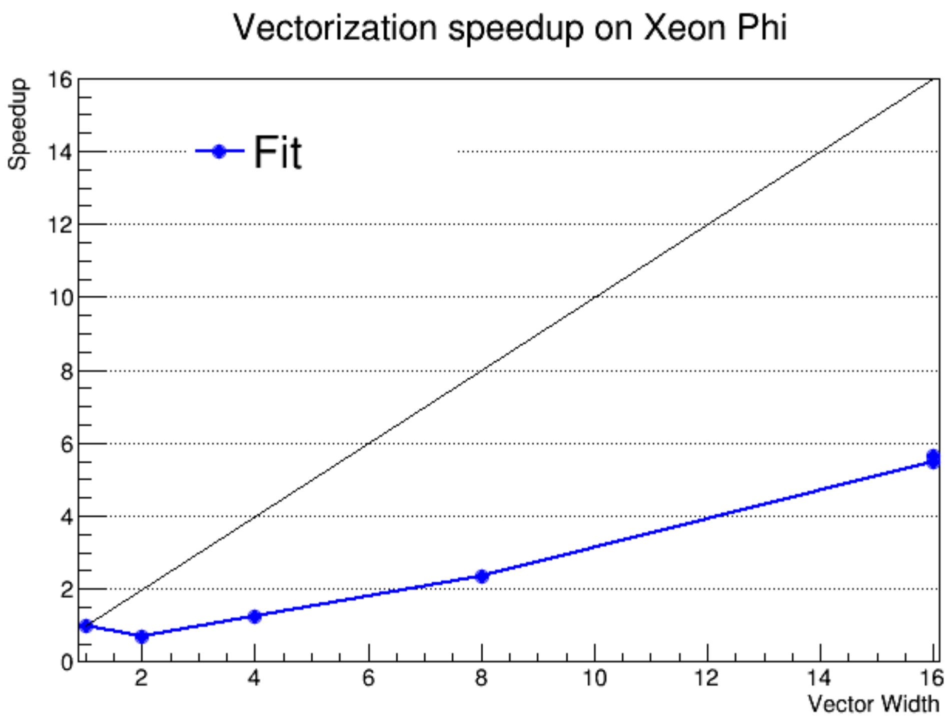
SlurpIn: Faster, One-Pass Initialization of Matriplex

- Load into register: vector gather op (hardware)
- Store from register: simple vector copy



Retest of Track Fitting in a Simplified Detector

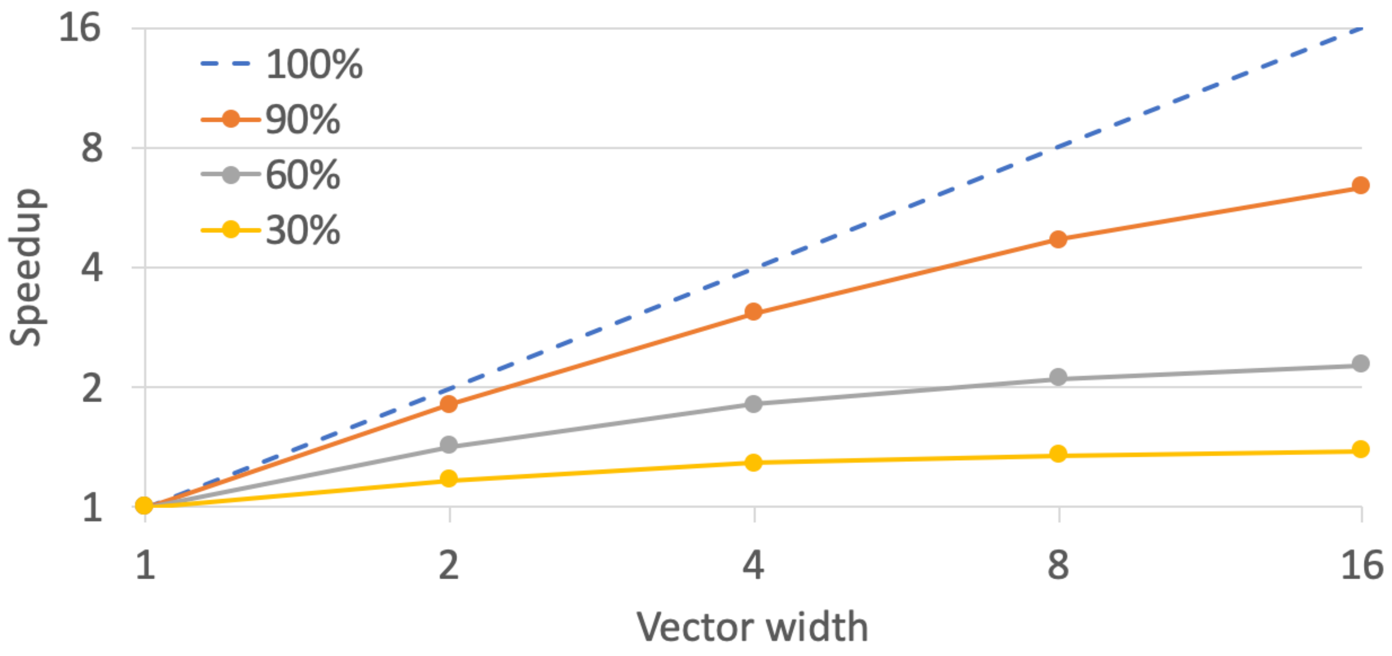




- After fixing Subtract and switching to SlurpIn, test runs 25% faster at full vector width, maximum speedup goes from ~4.4x to ~5.6x
- Amdahl's Law: *can't* get full speedup until *everything* is vectorized

A Quick Word on Amdahl's Law

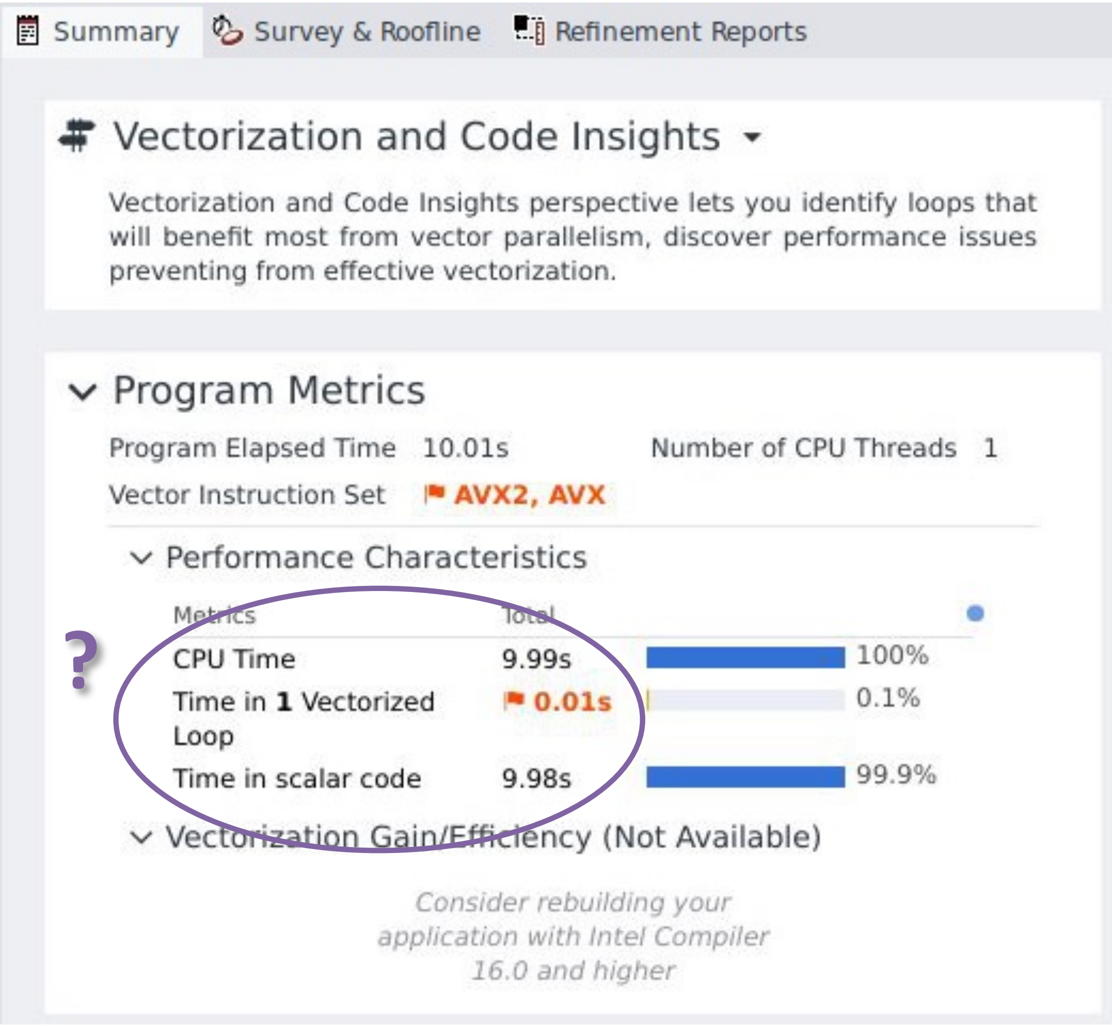
- SIMD means parallel, so Amdahl's Law is in effect!
  - Linear speedup is possible only for *perfectly* parallel code
  - Amdahl's asymptote of the speedup curve is 1/(serial fraction)
  - Speedup of 16x is unattainable even if 99% of work is vector

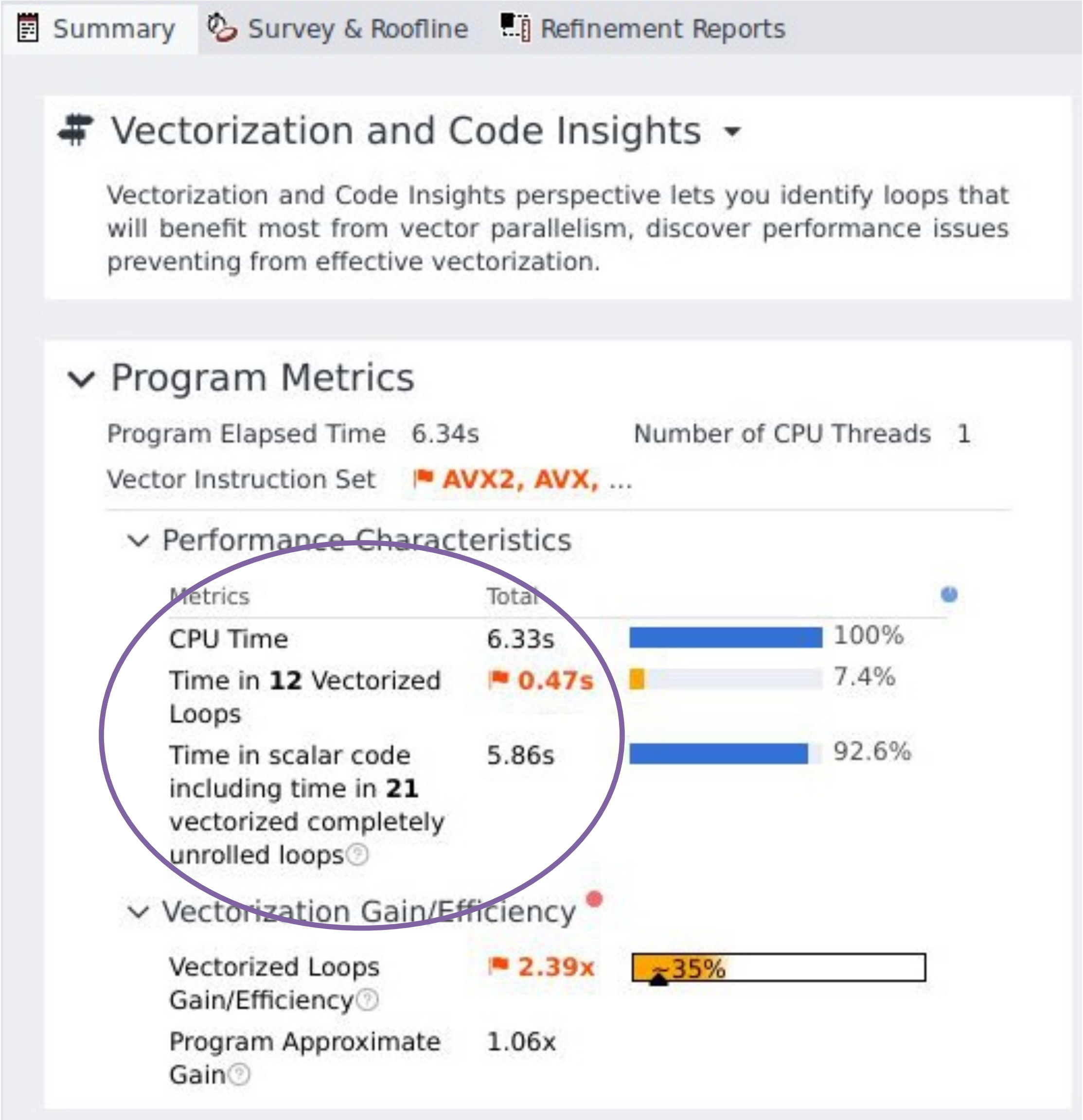


Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

Intel Advisor's Vectorization Report: gcc vs. icc





work with Patrick Gartung, Fermilab

Resolution of a Long-Term Mystery!

- The Intel C/C++ Compiler Classic always produced much faster code than GCC
- The reason could be traced to **sin/cos functions** needed during propagation
  - **icc** vectorized these from its **SVML**, enabling vectorization of a larger loop
  - **gcc** has an equivalent vector math library, **libmvec**, but it did not come until glibc 2.22
  - Thus, older operating systems such as CentOS 7 did not include libmvec
- The full solution did not arrive until last year…
  - AlmaLinux 8 (and similar CentOS 8 replacements) shipped with libmvec
  - For gcc to link to it, `-ffast-math` (or at least a subset of it) must also be specified
  - But still, gcc found the propagation loop too complicated to vectorize
  - The main loop had to be broken into many subloops that were obviously vectorizable

work with Patrick Gartung, Fermilab

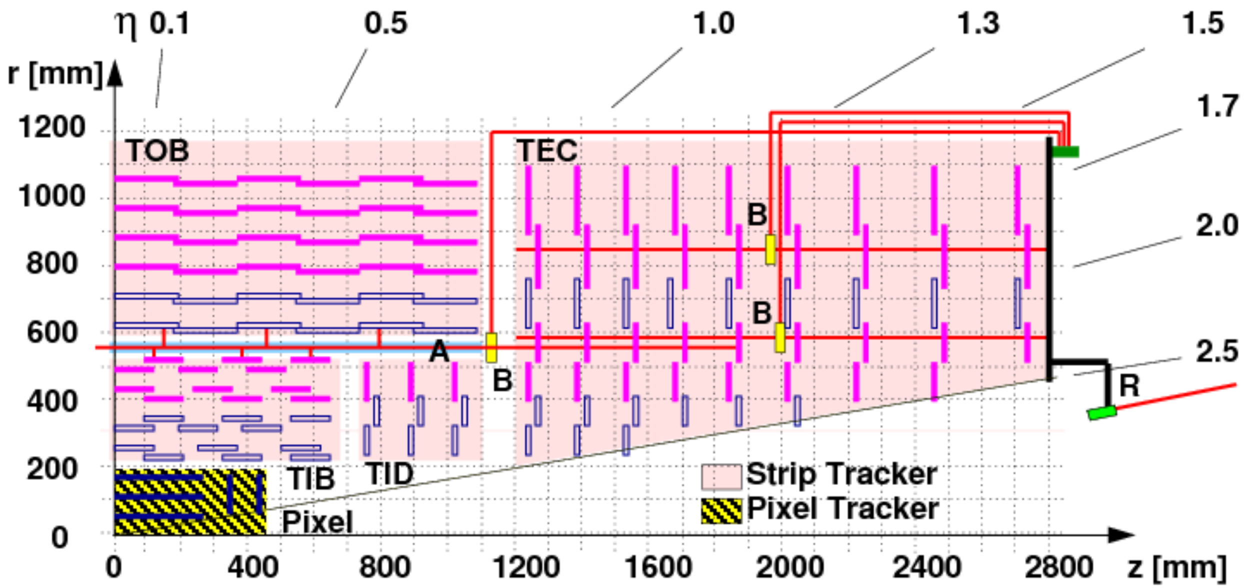
Outline

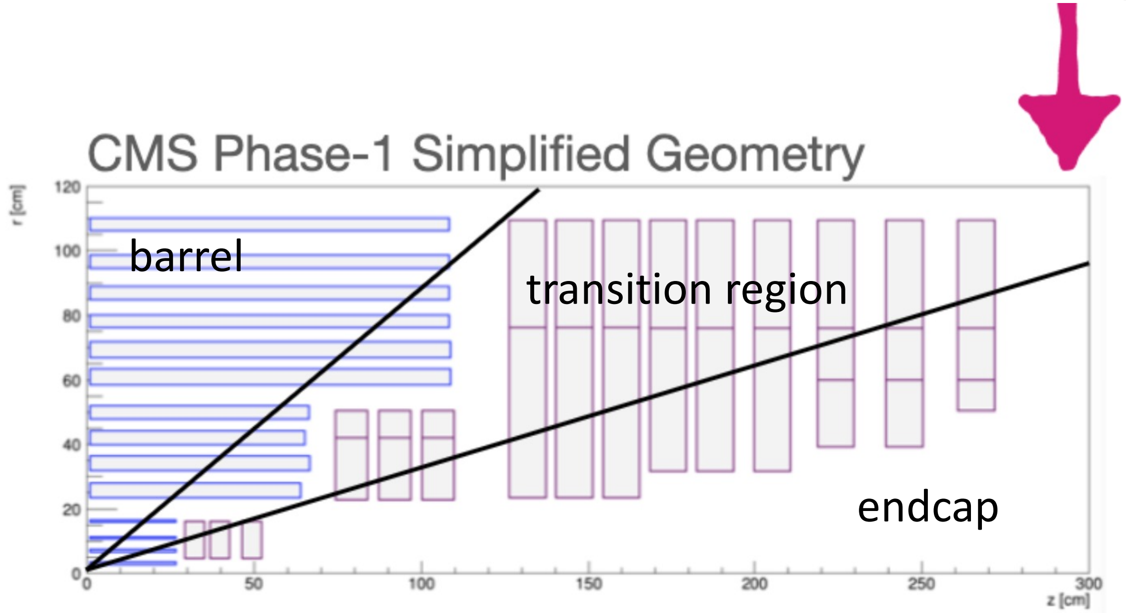
Strategy for Track Building with mkFit

- Keep the same goal of vectorizing and multithreading all operations
  - Vectorize by continuing to use Matriplex, just as in fitting
  - Multithread by binning tracks in eta (related to angle from axis)
- Add two big complications
  - *Hit selection:* hit(s) on next layer must be selected from ~10k hits
  - *Branching:* track candidate must be cloned for >1 selected hit
- Speed up *hit selection* by binning hits in both eta and phi (azimuth)
  - Faster lookup: compatible hits for a given track are found in a few bins
- Limit *branching* by putting a cap on the number of candidate tracks
  - Sort the candidate tracks at the completion of each layer
  - Keep only the best candidates; discard excess above the cap

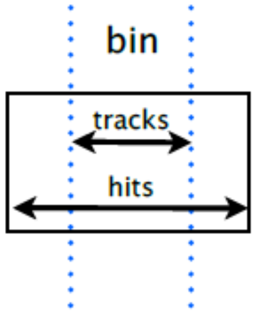
Simplifying the Geometry

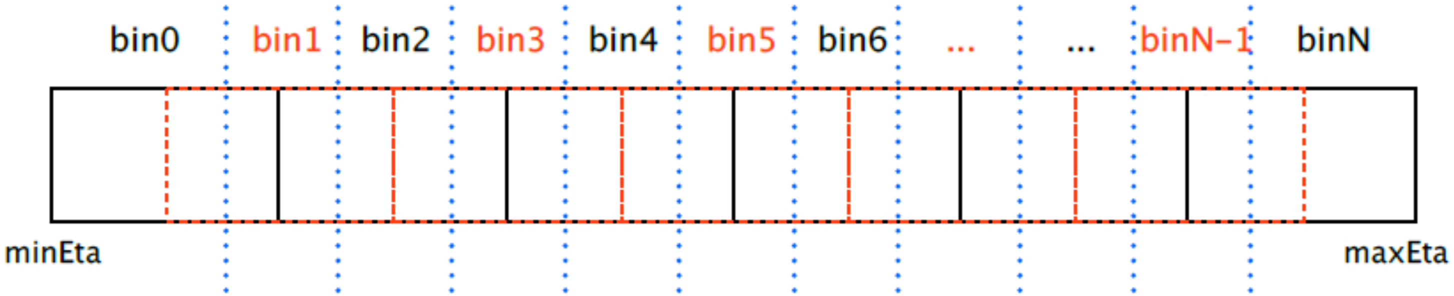
- Don't propagate to one of the tiled, overlapping modules in CMS; instead, SIMD-propagate bunches of tracks to an average r (barrel) or z (disk/endcap)
- Search for nearby hits in a global coordinate space
- Pay one-time, up-front cost (per event) to transform all hits into global coordinates





Eta Binning





- Eta binning is natural for both track candidates and hits
  - Tracks don’t curve in eta
- Form overlapping bins of hits, 2x wider than bins of track candidates
  - Track candidates never need to search beyond one extra-wide bin
- Associate threads with distinct eta bins of track candidates
  - Assign 1 thread to j bins of track candidates, or vice versa (j can be 1)
  - Threads work entirely independently → **task parallelism**

Intel Advisor: Lots of Time in Memory Operations

Advanced Hotspots Hotspots viewpoint (change)

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: S. Ti.	CPU Time: O. Ti.	Instructions Retired	Estimated Call Count	Total Iteration Count
std::vector<int, std::allocator<int>>::vector	40.772s	0s	0s	114,991,736,536	728,825,808	0
_int_free	39.751s	0s	0s	136,359,038,066	0	1,125,954,207
operator new	32.712s	0s	0s	86,154,002,942	0	0
atan2f	30.187s	0s	0s	96,263,571,713	0	0
brk	14.193s	0s	0s	2,656,096,078	0	0
Matriplex::MatriplexSym<float, (int)3, (int)8>::SlurpIn	13.738s	0s	0s	27,254,784,743	0	0
std::vector<Hit, std::allocator<Hit>>::vector	13.491s	0s	0s	48,368,155,014	1,447,206,650	6,041,737
Matriplex::CramerInverterSym<float, (int)3, (int)8>::Invert	8.327s	0s	0s	15,279,940,773	0	0
std::__unguarded_linear_insert<__gnu_cxx::__normal_iterator<Track*, std::vector<Track, std::allocator<T	6.851s	0s	0s	40,713,325,132	59,662,888	888,022,699
ROOT::Math::MatRepSym<float, (unsigned int)6>::operator=	6.092s	0s	0s	12,600,131,879	0	467,391,832
__intel_ssse3_rep_memmove	5.754s	0s	0s	14,338,306,198	0	0
std::vector<std::vector<Track, std::allocator<Track>>, std::allocator<std::vector<Track, std::allocator<T	4.927s	0s	0s	8,850,791,643	17,446	13,912,039
std::vector<EtaBinOfCombCandidates, std::allocator<EtaBinOfCombCandidates>>::~vector	4.838s	0s	0s	5,514,436,399	0	34,567,836
MkFitter::FindCandidates	4.508s	0s	0s	11,976,985,333	7,887,339	187,147,759
std::vector<Track, std::allocator<Track>>::reserve	4.334s	0s	0s	7,961,238,732	14,178,785	0
free	3.918s	0s	0s	12,843,035,454	0	0
std::vector<int, std::allocator<int>>::_M_emplace_back_aux<int const&>	3.012s	0s	0s	24,161,489,523	394,041,601	0
Matriplex::MatriplexSym<float, (int)6, (int)8>::operator=	2.818s	0s	0s	9,673,130,099	0	1,350,384,733
Track::Track	2.786s	0s	0s	7,584,629,305	93,542,787	463,911,688
_IO_file_write	2.592s	0s	0s	435,958,384	0	0
propagateHelixToRMPlex	2.203s	0s	0s	3,122,056,392	0	0
std::__insertion_sort<__gnu_cxx::__normal_iterator<Track*, std::vector<Track, std::allocator<Track>>>,	2.164s	0s	0s	7,990,728,691	5,356,129	62,442,951

- Profiling showed the busiest functions were memory operations!
- Cloning of candidates and loading of hits were major bottlenecks
  - This was alleviated by reducing sizes of Track by 20%, Hit by 40%
  - Track now references Hits by index, instead of carrying full copies

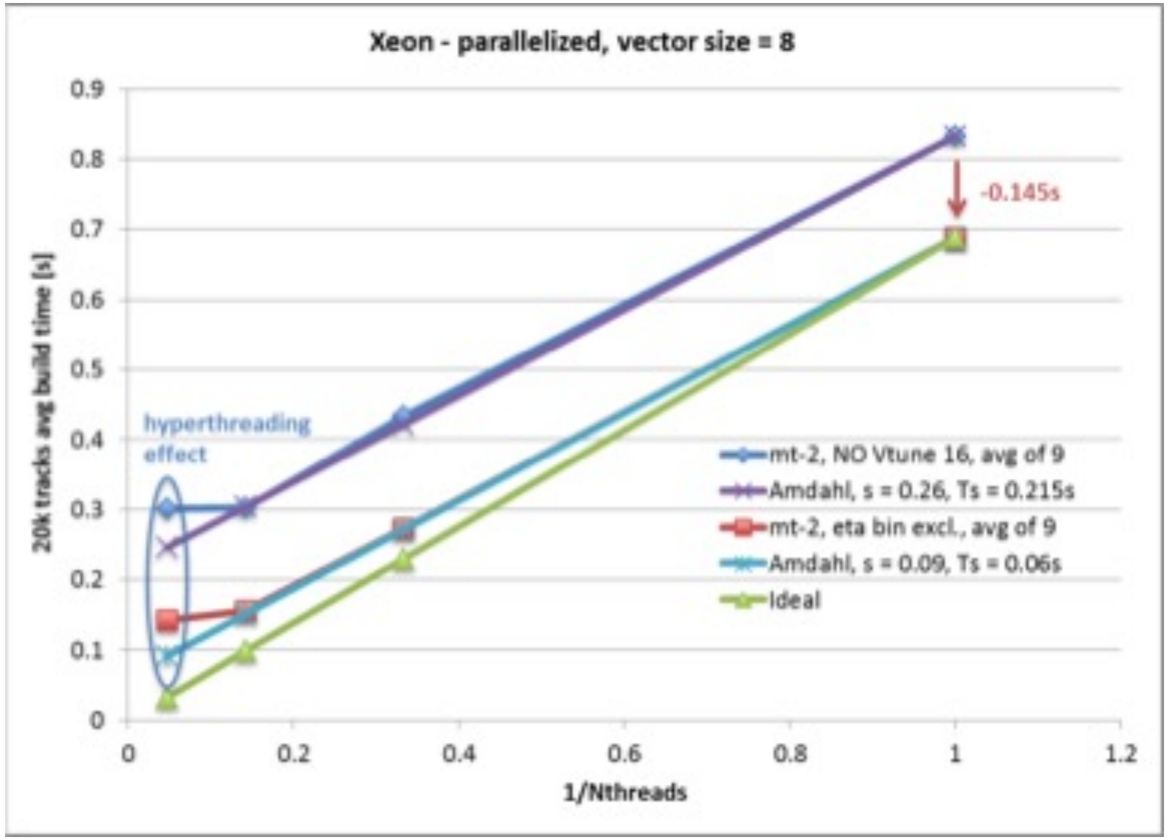
Amdahl's Law Again

- Possible explanation: some fraction B of work is a serial bottleneck
- If so, the minimum time for n threads is set by Amdahl's Law:

$$T(n) = T(1)\ [\underbrace{(1-B)/n}_{\text{parallelizable...}} + \underbrace{B}_{\textit{not!}}]$$

- Note, asymptote as $1/n \rightarrow 0$ is not zero, but $T(1)B$
- Idea: plot the scaling data vs. $1/n$ to see if it fits the above functional form
  - If it does, start looking for the source of B
  - **Progressively exclude any code not in an OpenMP parallel section**
  - Trivial-looking code may actually be a serial bottleneck…

Busted!



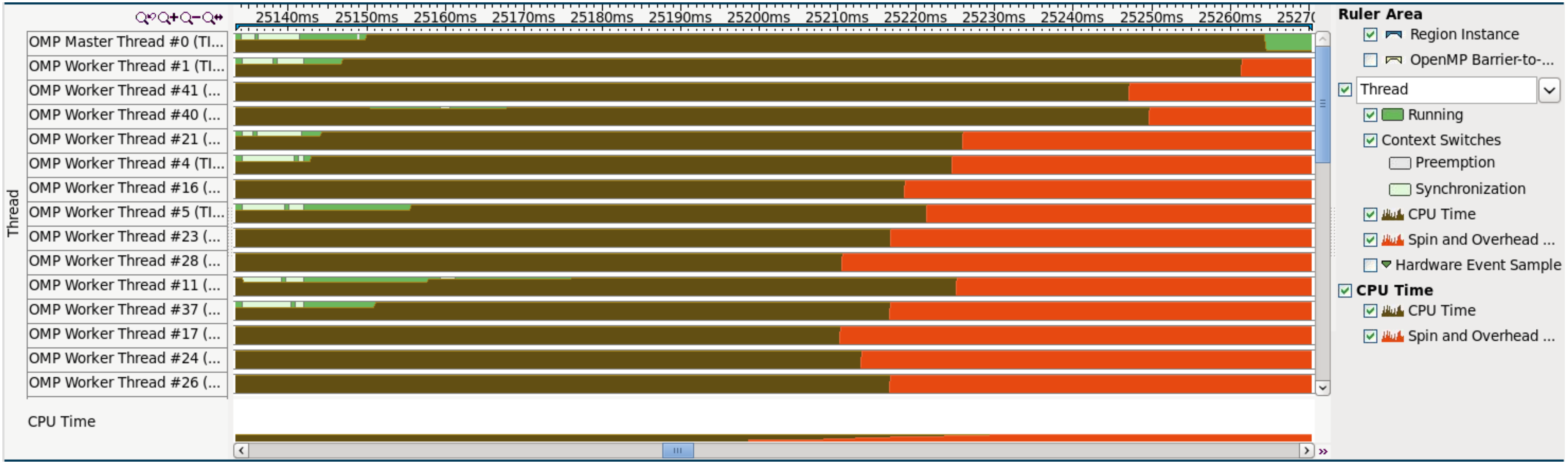
- Huge improvement from excluding *one code line* creating eta bins

```
EventOfCombCandidates event_of_comb_cands;
// constructor triggers a new std::vector<EtaBinOfCandidates>
```

- Accounts for 0.145s of serial time (0.155s)... scaling is still not ideal

Intel VTune Shows Another Issue

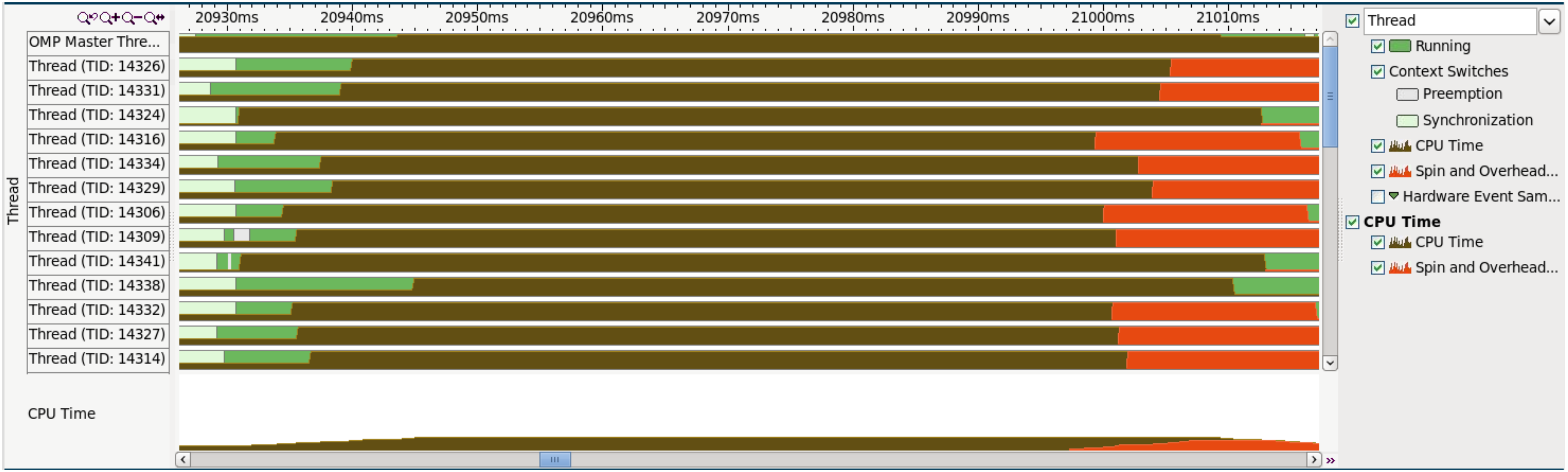
- VTune reveals non-uniformity of occupancy within OpenMP threads
  - Some threads take far longer than others: *load imbalance*
  - Worsens as threads increase: test below uses 42 threads on Xeon Phi



- Need dynamic reallocation of thread resources, e.g., task queues

Improvement with Intel Threading Building Blocks

- TBB allows eta bins to be processed by varying numbers of threads
- Allows idle threads to steal work from busy ones



- Much better load balance

Summary: Building Tracks in Parallel with mkFit

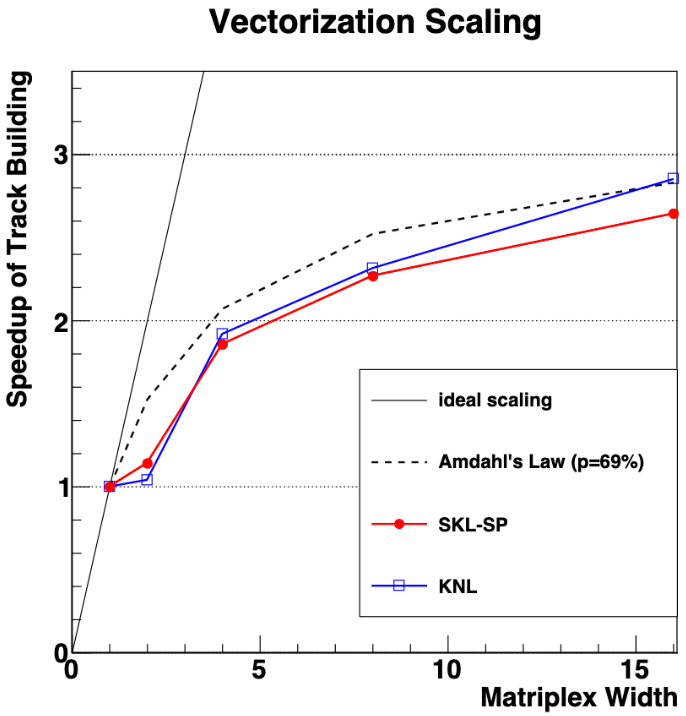
- Nested levels of parallel tasks for track building:
  1. Loop over different events;
  2. Loop over different η-regions;
  3. Loop over z-/r- and φ-sorted groups of seeds.
- Parallel tasks scheduled through Intel TBB
  - Dynamic task stealing to balance workloads
- Basic parallel task includes simplified two-step propagation
  - Propagate to average r or z of detector layer, compute compatibility window
  - Propagate to each hit in window, select which hit(s) to add to track based on χ^2
  - Kalman calculations include multiple scattering and energy loss in detector layer

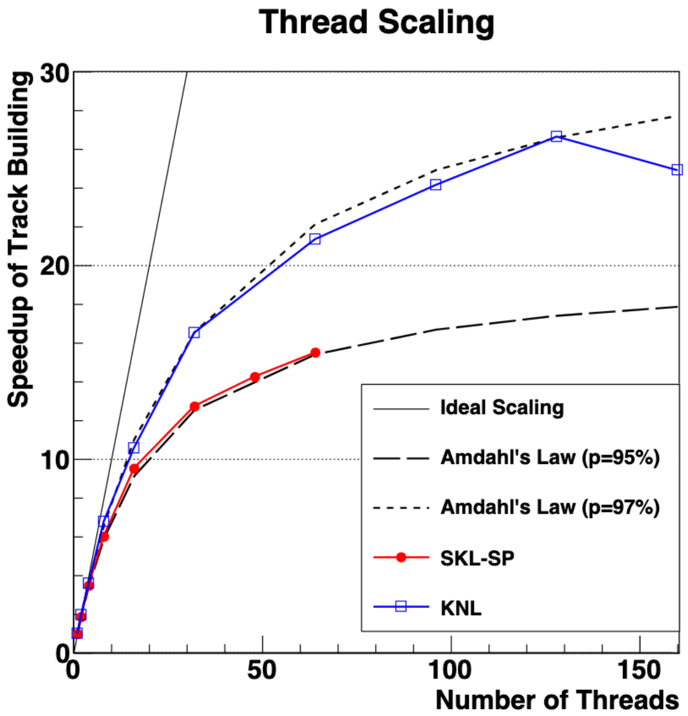
Outline

1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Using compilers to auto-vectorize track propagation
6. The multithreaded framework for building tracks
7. Conclusions and future directions

mkFit Code Performance

- Estimates of parallelization based on Amdahl's Law
  - ~70% vectorized
  - 95%+ multithreaded
- Up to 6.7x faster building time where mkFit is used
  - Reduction of 25% in total tracking time
  - Event throughput increase of 10–15% in LHC Run 3





"KNL" — 64 cores:
Intel Xeon Phi 7210 @ 1.30 GHz
"SKL-SP" — 2-socket x 16 cores:
Intel Xeon Gold 6130 @ 2.10 GHz

CMS is now using mkFit by default for computing most tracks

Future Directions

- Extend the mkFit paradigm to more applications
  - Example: extend to more complex track building steps for further speed-up
- Apply to track fitting
  - Time for fitting is now comparable to track building
- Build tracks for the High Level Trigger
  - The HLT computes on the raw data *in real time* and decides which events to keep
- Modify for CMS Phase-2 geometry and configuration
  - Optimize and tune for the new detector
  - Look for synergies with other algorithms