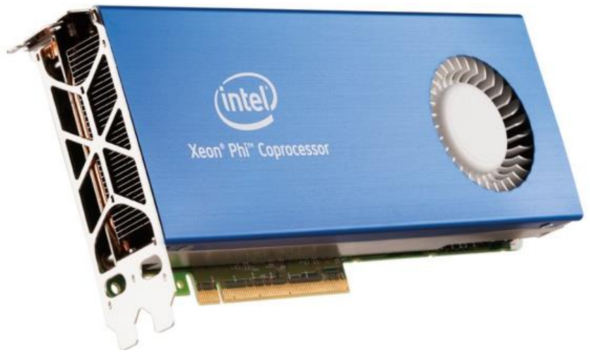
Particle Track Reconstruction for the Large Hadron Collider: Progress in Many-Core Parallel Computing

Steve Lantz
Senior Research Associate
Cornell University Center for Advanced Computing (CAC)
steve.lantz@cornell.edu

SCAN Seminar, May 9, 2016

Many-Core, not Manticore...



Many-Core Computing in High Energy Physics

Collaborators
K.McDermott,
D.Riley,
P.Wittich
(Cornell);
G.Cerati,
M.Tadel,
S.Krutelyov,
F.Würthwein,
A.Yagil
(UCSD);
P.Elmer,
M.Lefebvre
(Princeton)

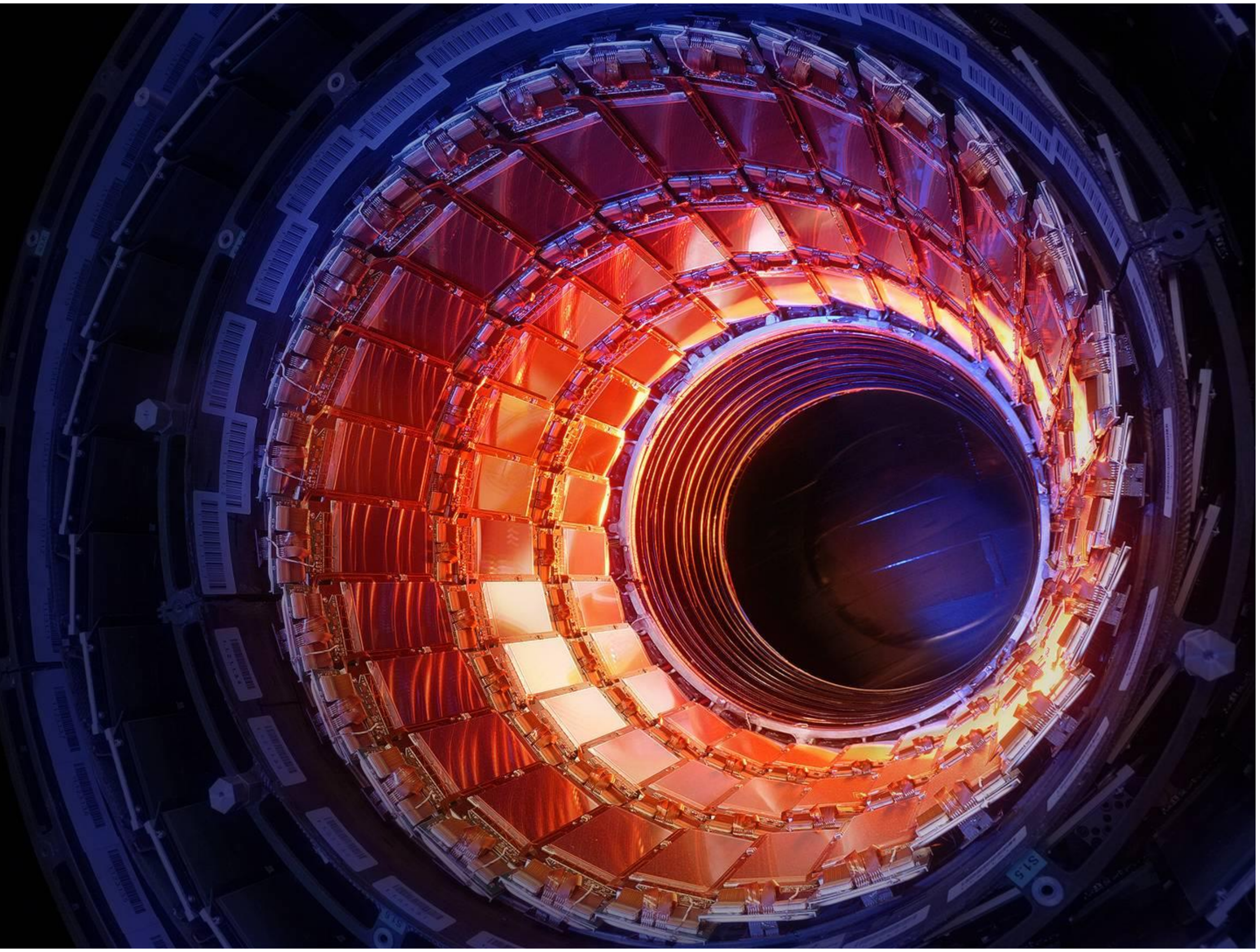
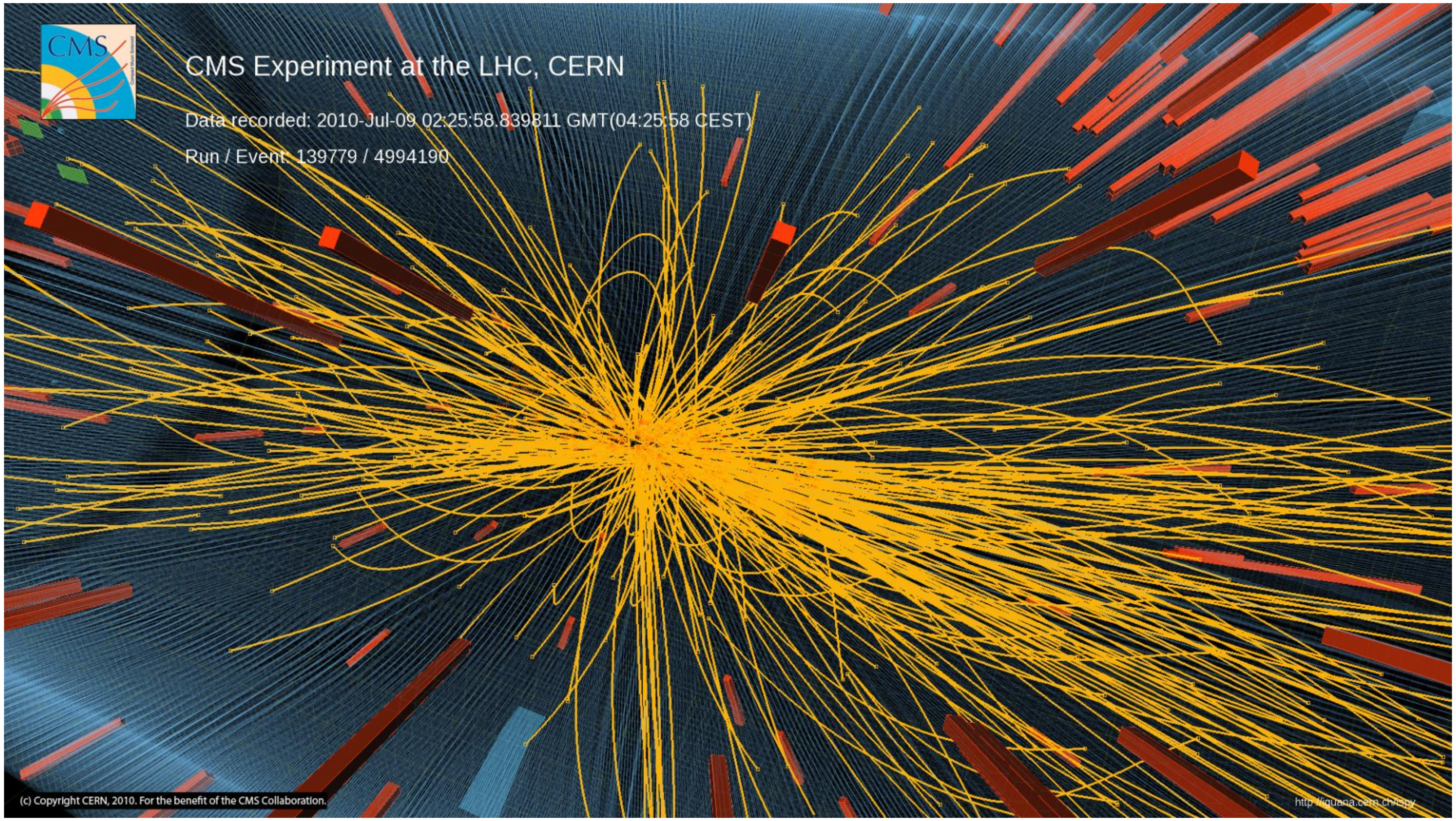


Photo: CMS detector, LHC, CERN

LHC: The *Super Collider!*

The Large Hadron Collider smashes beams of protons into each other, as they go repeatedly around a ring 17 miles in circumference at nearly the speed of light

Collision Energy Becomes Particle Masses: $E=mc^2$


CMS
CMS Experiment at the LHC, CERN
Data recorded: 2010-Jul-09 02:25:58.839811 GMT(04:25:58 CEST)
Run / Event: 139779 / 4994190
(c) Copyright CERN, 2010. For the benefit of the CMS Collaboration.
http://iguana.cern.ch/ispy

Higgs Discovery @ LHC

Big news on July 4, 2012!

theguardian

News | US | World | Sports | Comment | Culture | Business | Enviro

News › Science › Higgs boson

What is the Higgs boson?

Physicists are set to announce the latest results from the Large Hadron Collider (LHC), but what exactly is the Higgs boson, why do people call it the 'god particle' and what would its discovery mean for physics?

Ian Sample and James Randerson
guardian.co.uk, Friday 29 June 2012 09.35 EDT

FOX NEWS .com Fair & Balanced

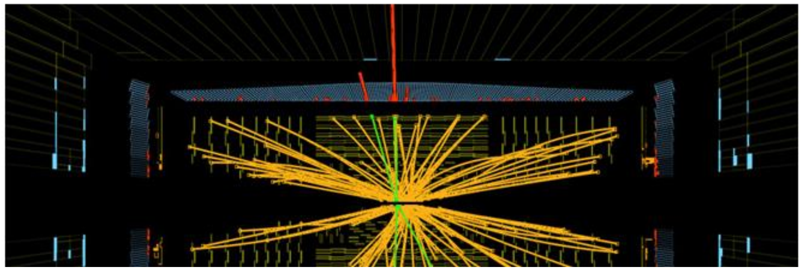
Home | Video | Politics | U.S. | Opinion

Science Home | Archaeology | Air & Space | Planet

BREAKING NEWS ISRAEL SAYS IT S

The elusive particle: Higgs Boson

Published July 05, 2012 / LiveScience



HOME PAGE | TODAY'S PAPER | VIDEO | MOST POPULAR | U.S. Edition

The New York Times Science

WORLD | U.S. | N.Y. / REGION | BUSINESS | TECHNOLOGY | SCIENCE | HEALTH | SPORTS | OPINION | ARTS | S

ENVIRONMENT SPACE & COSMOS

Physicists Find Elusive Particle Seen as Key to Universe

Pool photo by Denis Balibouse

Scientists in Geneva on Wednesday applauded the discovery of a subatomic particle that looks like the Higgs boson.

By DENNIS OVERBYE
Published: July 4, 2012 | 122 Comments

ASPEN, Colo. — Signaling a likely end to one of the longest, most expensive searches in the history of science, physicists said

FACEBOOK
TWITTER

Today'

21

nature International weekly journal of science

Home | News & Comment | Research | Careers & Jobs | Current Issue | Archive | Audio & Video | For A

News & Comment › News › 2012 › November › Article

NATURE | NEWS

Physicists declare victory in Higgs hunt

Researchers must now pin down the precise identity of their new particle.

Geoff Brumfiel

04 July 2012

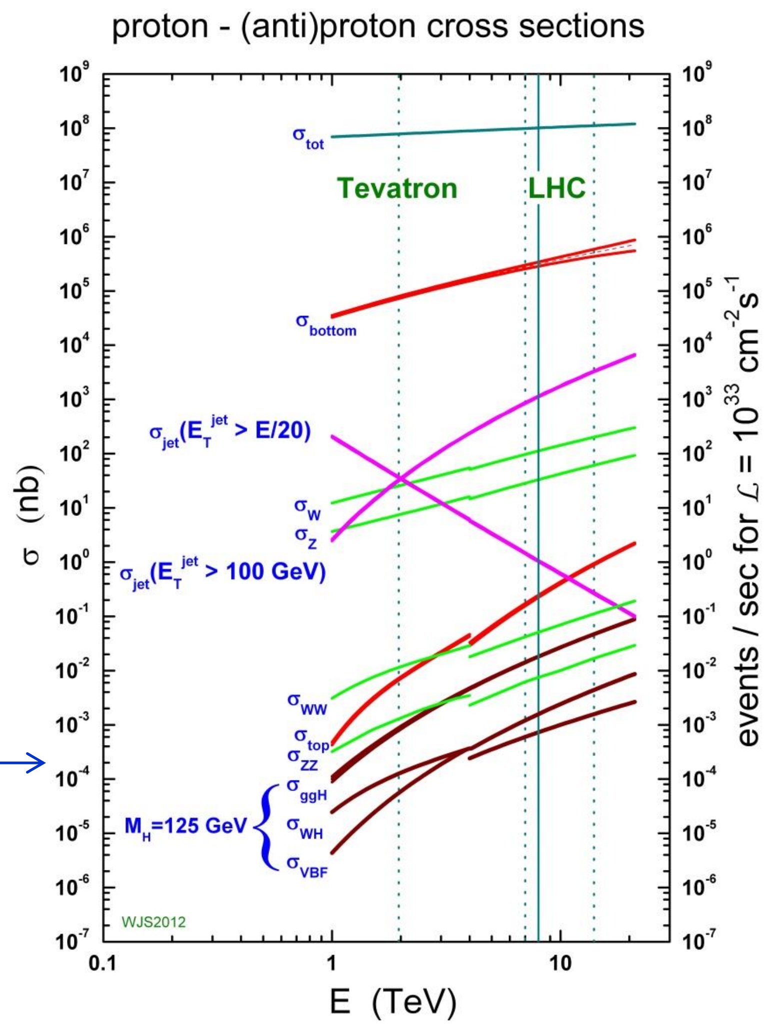
Physicists announced today that they have seen a clear signal of a Higgs boson — a key part of the mechanism that gives all particles their masses.

Two independent experiments reported their

Big Data Challenge

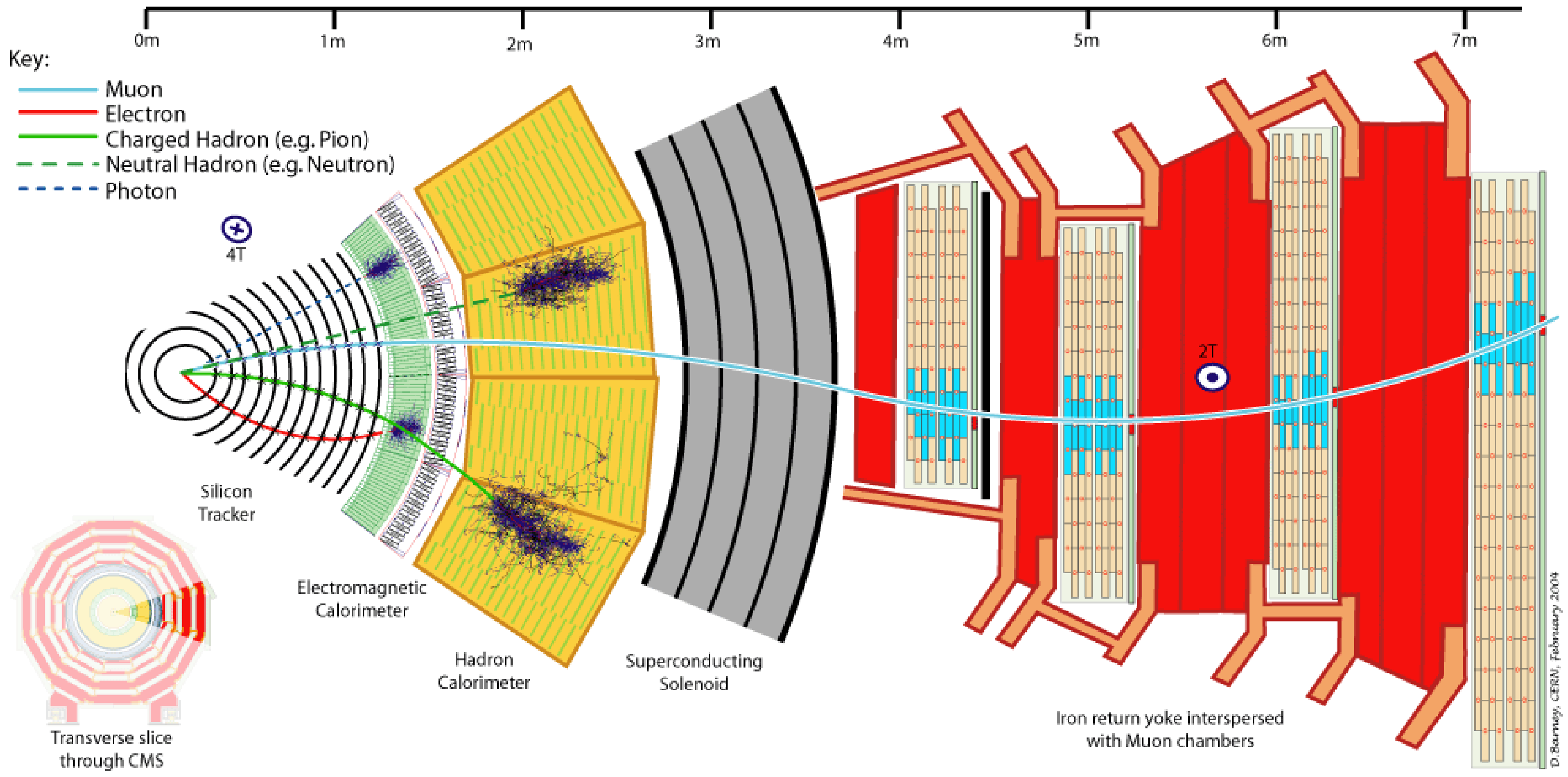
- *40 million* collisions a second
- Most are boring
  - Dropped within 3 μs
- 0.5% are interesting
  - Worthy of reconstruction...
- Higgs events: *super* rare
  - 10^{16} collisions → 10^{6} Higgs
  - Maybe 1% of these are found

- Ultimate "needle in a haystack"
- "Big Data" since before it was cool



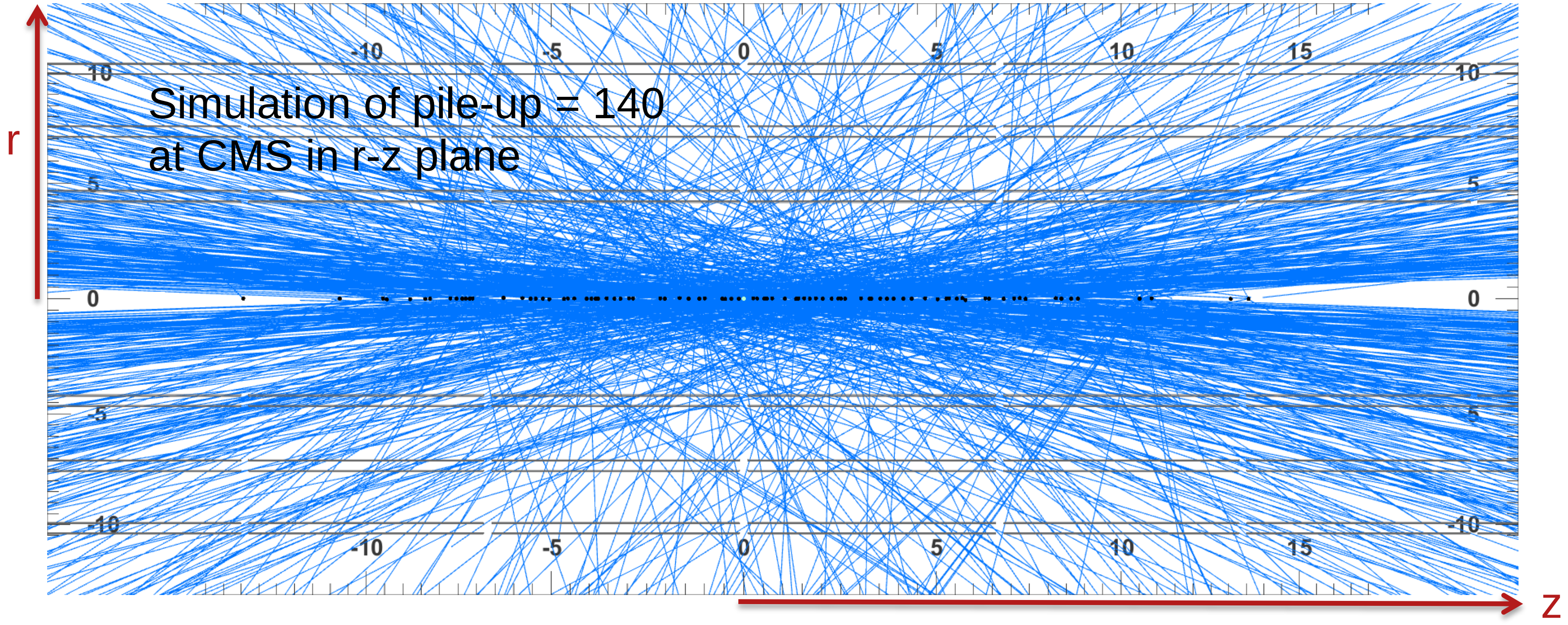
http://www.hep.ph.ic.ac.uk/~wstirlin/plots/plots.html

CMS: Like a Fast Camera for Identifying Particles


0m
1m
2m
3m
4m
5m
6m
7m
Key:
Muon
Electron
Charged Hadron (e.g. Pion)
Neutral Hadron (e.g. Neutron)
Photon
4T
2T
Silicon Tracker
Electromagnetic Calorimeter
Hadron Calorimeter
Superconducting Solenoid
Iron return yoke interspersed with Muon chambers
Transverse slice through CMS
D.Barney, CERN, February 2004

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

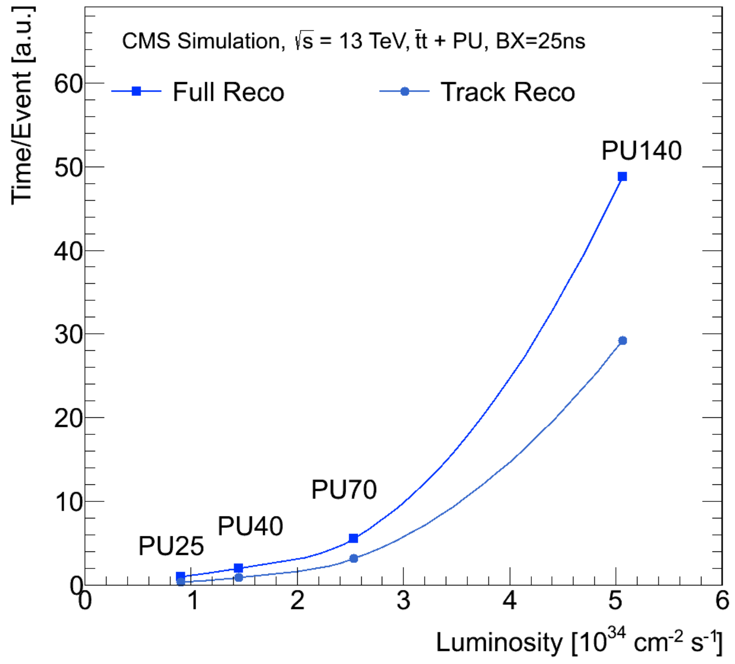
CMS Is About to Get Busier



- By 2025, 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 per event

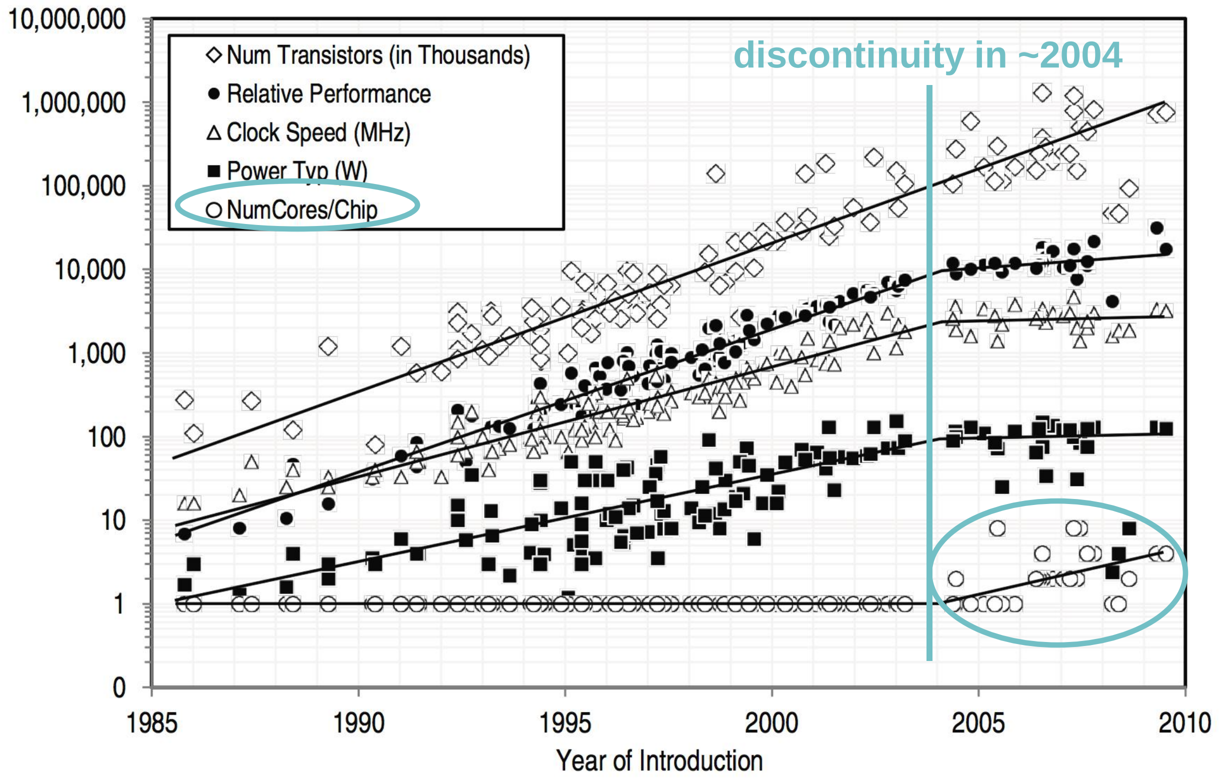
Reconstruction Will Soon Run Into Trouble

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



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

Overview of CPU Speed and Complexity Trends

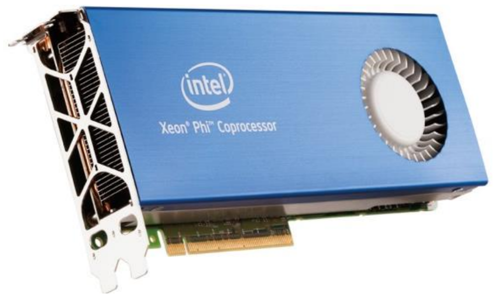


Committee on Sustaining Growth in Computing Performance, National Research Council. "What Is Computer Performance?"
In *The Future of Computing Performance: Game Over or Next Level?*
Washington, DC: The National Academies Press, 2011.

How TACC Stampede Reached ~10 Petaflop/s

- 2+ petaflop/s of Intel Xeon E5
- 7+ additional petaflop/s of Intel Xeon Phi™ SE10P *coprocessors*
- Follows the hardware trend of the last 10 years: processors gain cores (execution engines) rather than clock speed
- So is Moore's Law dead? No!
  - Transistor densities are still doubling every 2 years
  - Clock rates have stalled at < 4 GHz due to power consumption
  - Only way to increase flop/s/watt is through *greater on-die parallelism*
- Architectures are therefore moving from multi-core to ***many-core***

Xeon Phi: What Is It?



- *An x86-derived CPU featuring a large number of simplified cores*
  - Many Integrated Core (MIC) architecture
- *An HPC platform geared for high floating-point throughput*
  - Optimized for floating-point operations per second (flop/s)
- *Intel's answer to general purpose GPU (GPGPU) computing*
  - Similar flop/s/watt to GPU-based products like NVIDIA Tesla
- Just another target for the compiler; no need for a special API
  - Compiled code is not (yet) binary compatible with x86_64
- Initially, a full system on a PCIe card (separate Linux OS, RAM)...
- **KNL**: with "Knight's Landing", *Xeon Phi can be the main CPU*

Many-Core Elements in Petaflop/s Machines

- **CPUs: Wider vector units, more cores**
  - AVX instructions crunch 8 or 16 floats at a time
  - Single thread runs well; dozens are needed
  - Stampede example: peak DP, dual Xeon E5-2680 - **0.34 Tflop/s, 260W**
- **GPUs: 1000s of simple stream processors**
  - Single Instruction, Multiple Thread (SIMT): think vector units, not cores
  - Special APIs are required: CUDA, OpenCL, OpenACC
  - Stampede example: peak DP, NVIDIA Tesla K20 - **1.17 Tflop/s, 225W**
- **MICs: 60+ CPU cores, floating-point efficiency**
  - Slow clock, yet high flop/s from more/wider vectors, more cores
  - Intel compiler handles vectorization and multithreading of OpenMP code
  - Stampede example: peak DP, Xeon Phi SE10P - **1.06 Tflops/s, 300W**
  - *Next generation "Knight's Landing" (KNL): ~3 Tflop/s, ~300W*

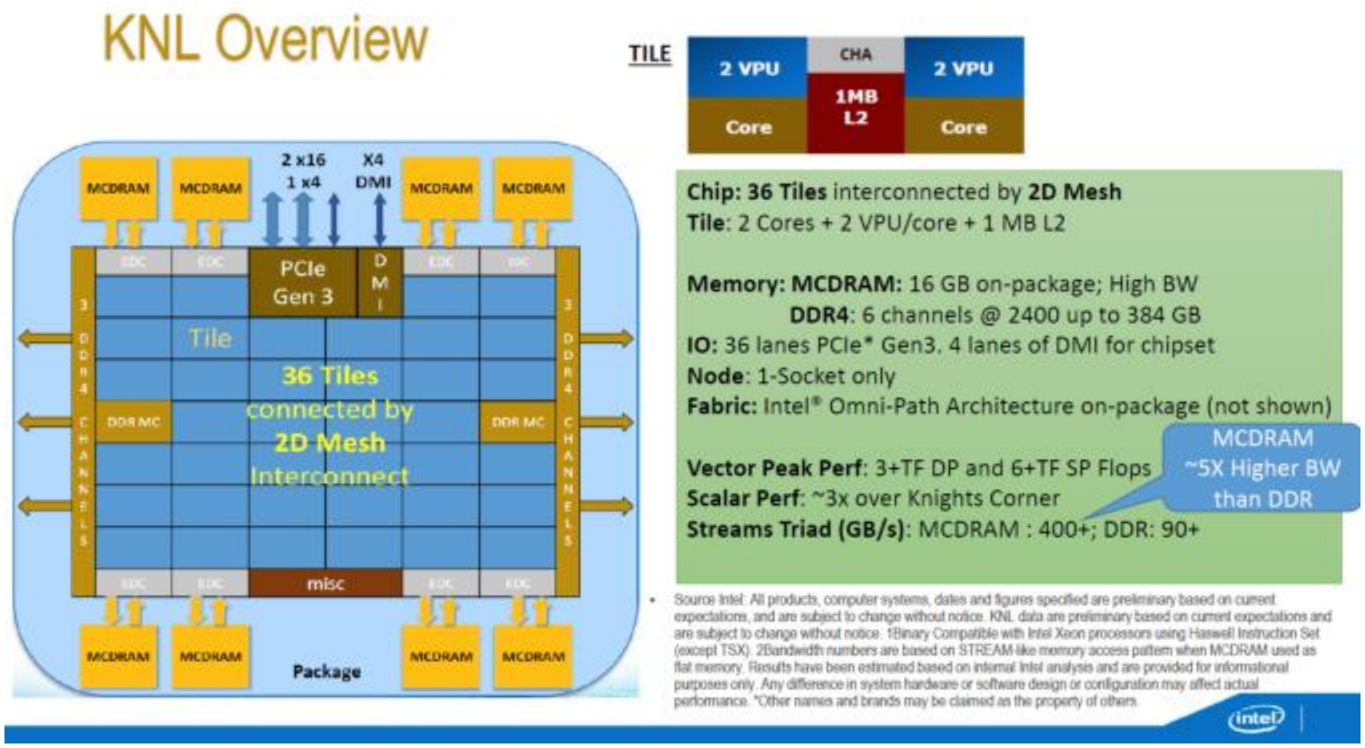
Xeon Phi vs. Xeon

	SE10P	Xeon E5	Xeon Phi is…
Number of cores	61	8	much higher
Clock speed (GHz)	1.01	2.7	lower
SIMD width (bits)	512	256	higher
DP Gflop/s/core	16+	21+	lower
HW threads/core	4	1*	higher

- Xeon designed for all workloads, high single-thread performance
- Xeon Phi also general purpose, but optimized for number crunching
  - High aggregate throughput via lots of weaker threads, more SIMD
  - Possible to achieve >2x performance compared to dual E5 CPUs

Where CPU Technology Is Headed Next: KNL

- 72 cores, 2 VPUs/core, new RAM layer (fast memory or slow cache)



Chip: 36 Tiles interconnected by **2D Mesh**
Tile: 2 Cores + 2 VPU/core + 1 MB L2

Memory: MCDRAM: 16 GB on-package; High BW
DDR4: 6 channels @ 2400 up to 384 GB
IO: 36 lanes PCIe* Gen3. 4 lanes of DMI for chipset
Node: 1-Socket only
Fabric: Intel® Omni-Path Architecture on-package (not shown)

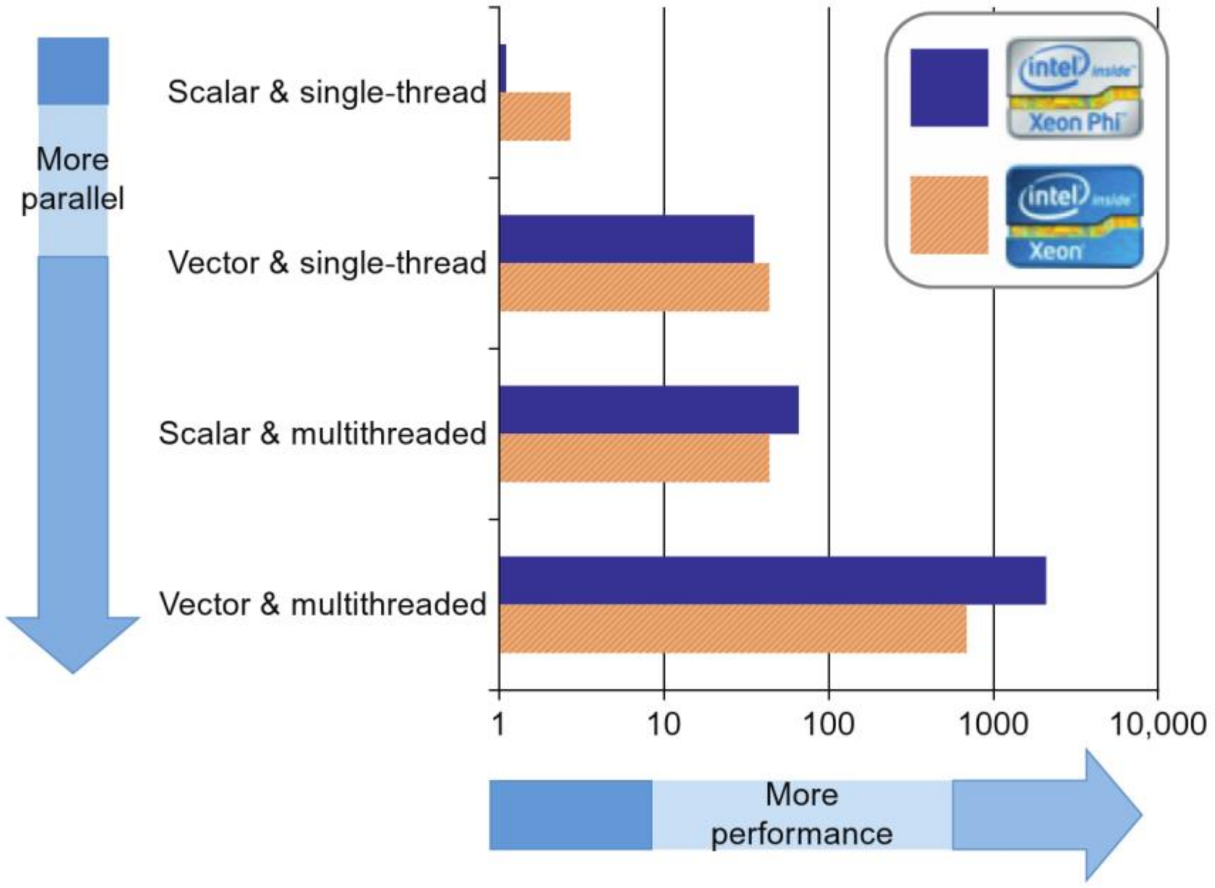
Vector Peak Perf: 3+TF DP and 6+TF SP Flops
Scalar Perf: ~3x over Knights Corner
Streams Triad (GB/s): MCDRAM : 400+; DDR: 90+

- Source Intel: All products, computer systems, dates and figures specified are preliminary based on current expectations, and are subject to change without notice. KNL data are preliminary based on current expectations and are subject to change without notice. 1Binary Compatible with Intel Xeon processors using Haswell Instruction Set (except TSX). 2Bandwidth numbers are based on STREAM-like memory access pattern when MCDRAM used as flat memory. Results have been estimated based on internal Intel analysis and are provided for informational purposes only. Any difference in system hardware or software design or configuration may affect actual performance. *Other names and brands may be claimed as the property of others.

Two Types of MIC (and CPU) Parallelism

- Threading (task parallelism)
  - OpenMP, Cilk Plus, TBB, Pthreads, etc.
  - It's all about sharing work and scheduling
- Vectorization (data parallelism)
  - "Lock step" Instruction Level Parallelization (SIMD)
  - Requires management of synchronized instruction execution
  - It's all about finding simultaneous operations
- To utilize MIC fully, both types of parallelism need to be identified and exploited
  - Need 2–4+ threads to keep a MIC core busy (in-order execution stalls)
  - Vectorized loops gain 8x or 16x performance on MIC!
  - Important for CPUs as well: gain of 4x or 8x on Sandy Bridge

Parallelism and Performance on Xeon Phi vs. Xeon

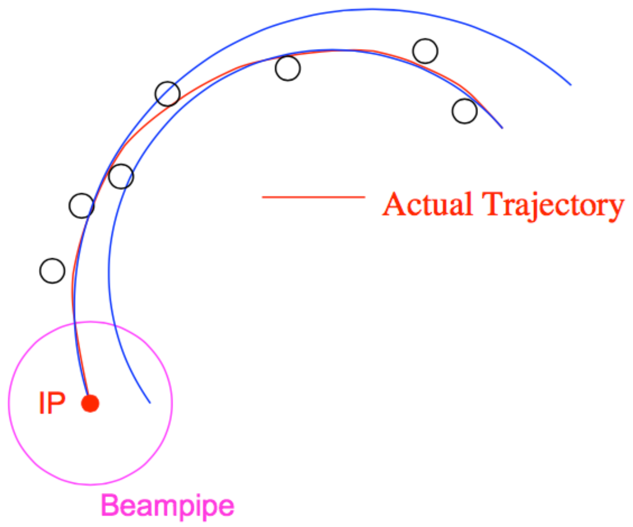


Courtesy James Reinders, Intel

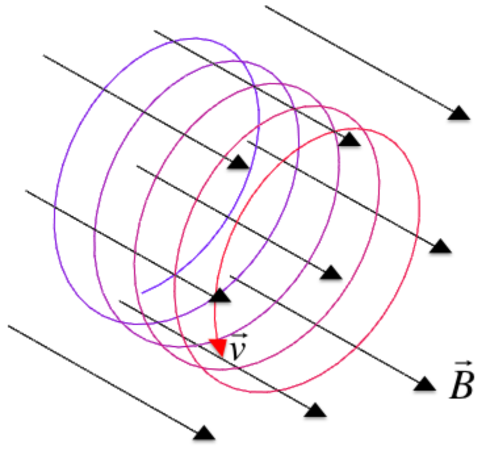
What Does the Tracking Algorithm Do?

- 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) tracks
- Trajectory may change due to interaction with materials
- Ultimately we care mainly about:
  - ***Initial track parameters***
  - ***Exit position to the calorimeters***

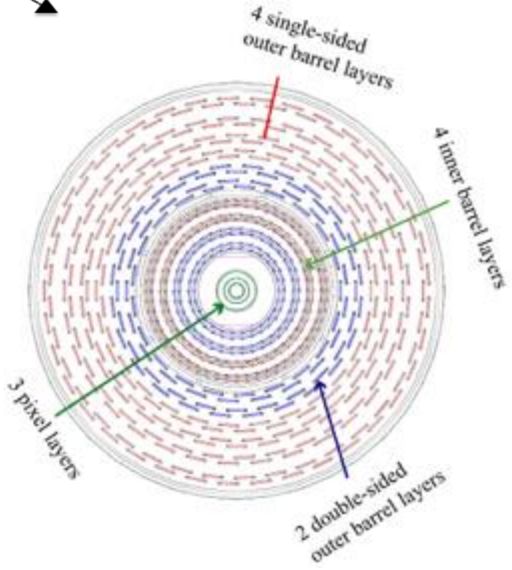
- ***We use a Kalman Filter-based technique***



Why Kalman Filter for Particle Tracking?



science fiction...

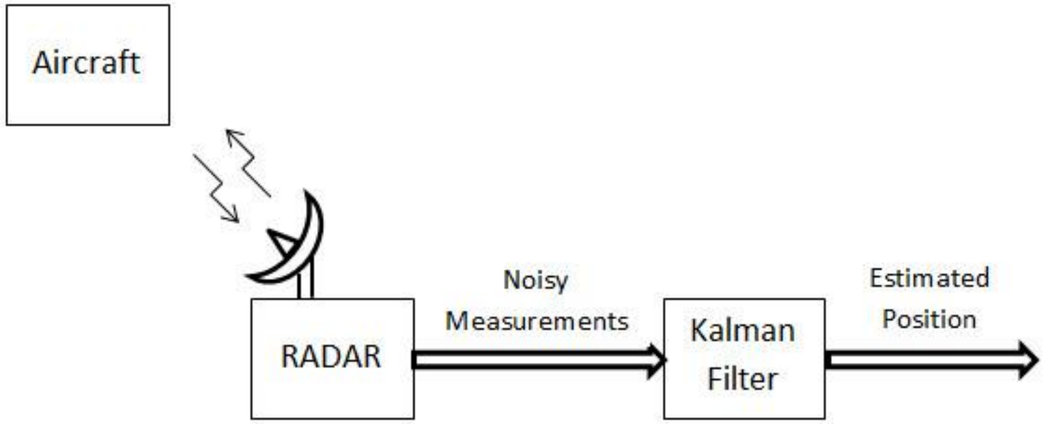

4 single-sided outer barrel layers
4 inner barrel layers
3 pixel layers
2 double-sided outer barrel layers

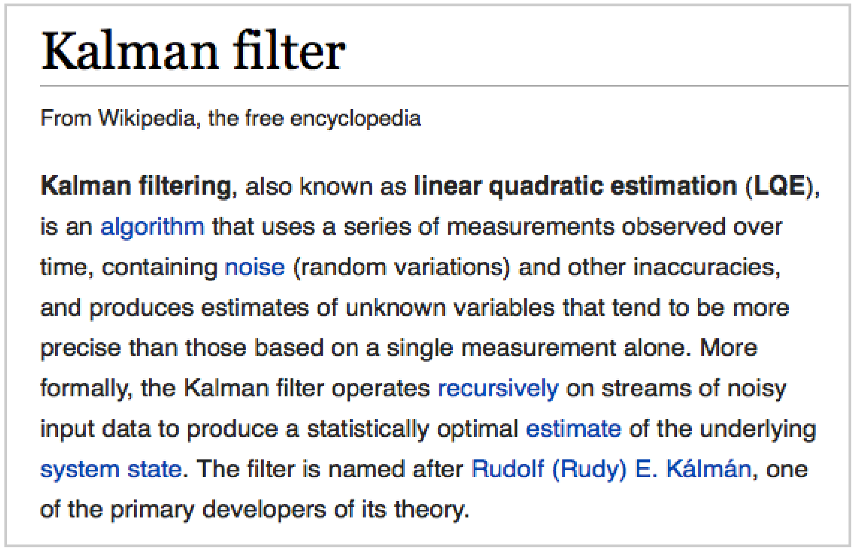
..vs. real materials

- Naively, the particle's trajectory is described by a single helix
- Forget it
  - Non-uniform B field
  - Scattering
  - Energy loss
  - ...
- Trajectory is only *locally helical*
- Kalman Filter allows us to take these effects into account, while preserving a locally smooth trajectory

Kalman Filter

- Method for obtaining best estimate of the five track parameters
- Natural way of including interactions in the material (process noise) and hit position uncertainty (measurement error)
- Used both in *pattern recognition* (i.e., determining which hits to group together as coming from one particle) and in *fitting* (i.e., determining the ultimate track parameters)





Kalman filter

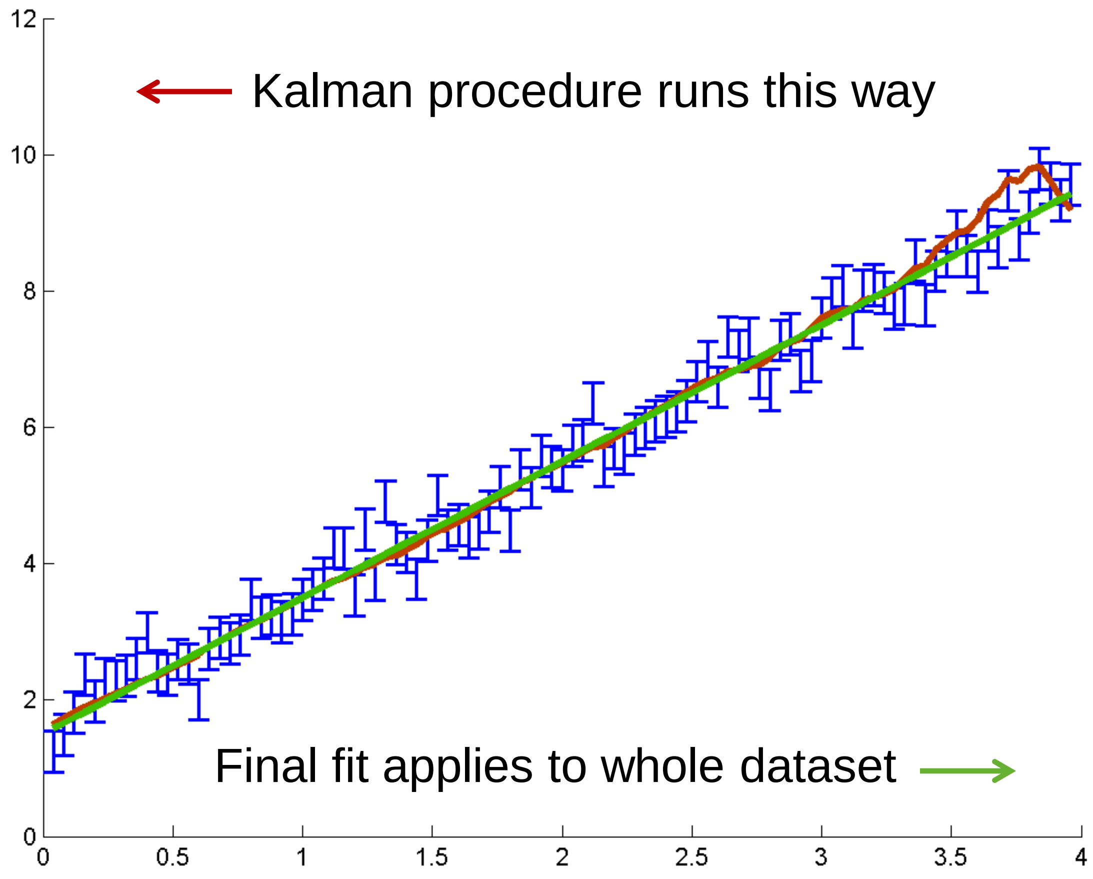
From Wikipedia, the free encyclopedia

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

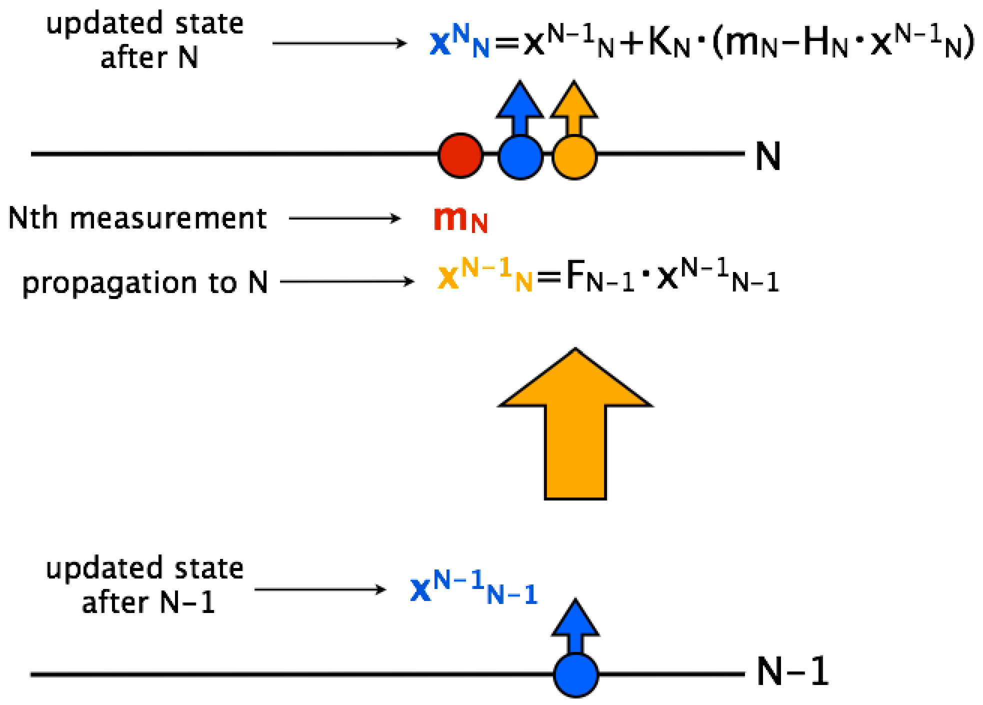
Kalman Example

- 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

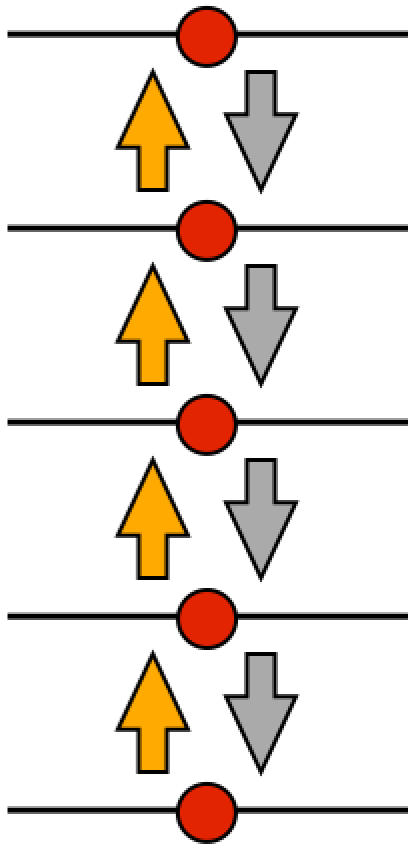
- Track reconstruction has 3 main steps: *seeding*, *building*, and *fitting*
- Building and fitting repeat the basic logic unit of the Kalman Filter...


updated state after N
$x^N_N = x^{N-1}_N + K_N \cdot (m_N - H_N \cdot x^{N-1}_N)$
N
Nth measurement
m_N
propagation to N
$x^{N-1}_N = F_{N-1} \cdot x^{N-1}_{N-1}$
updated state after N−1
x^{N-1}_{N-1}
N−1

- From current *track state* (parameters and uncertainties), track is *propagated* to next layer
- Using hit measurement information, track state is *updated (filtered)*
- Procedure is repeated until last layer is reached

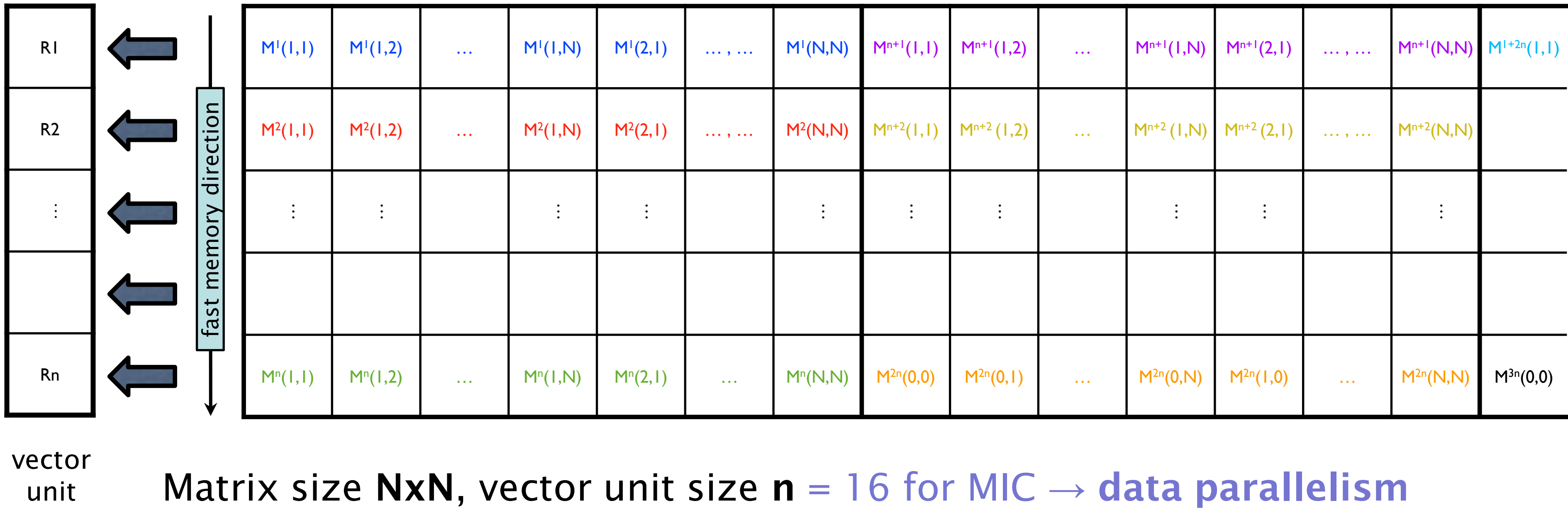
Track Fitting as Kalman Filter

- 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, many tracks can be fit *in parallel*



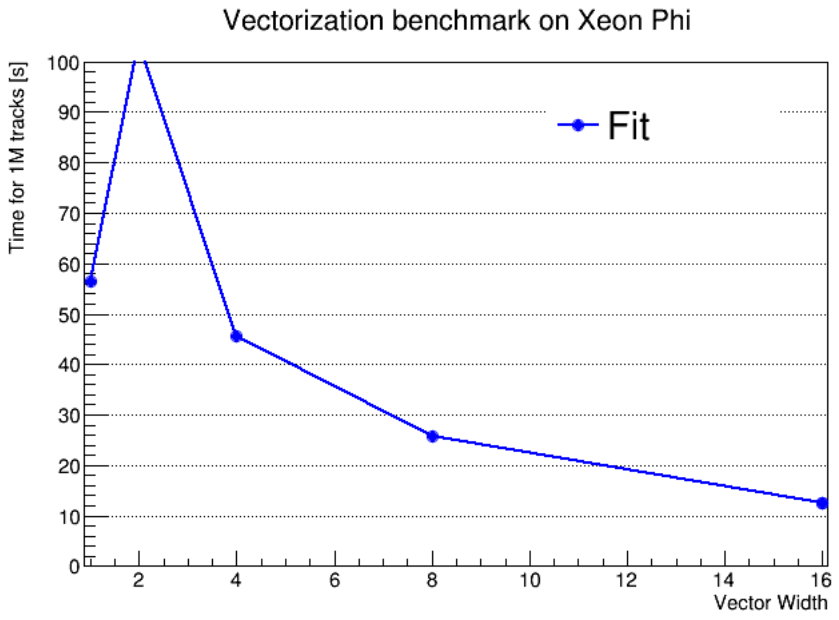
"Matriplex" Structure for Kalman Filter Operations

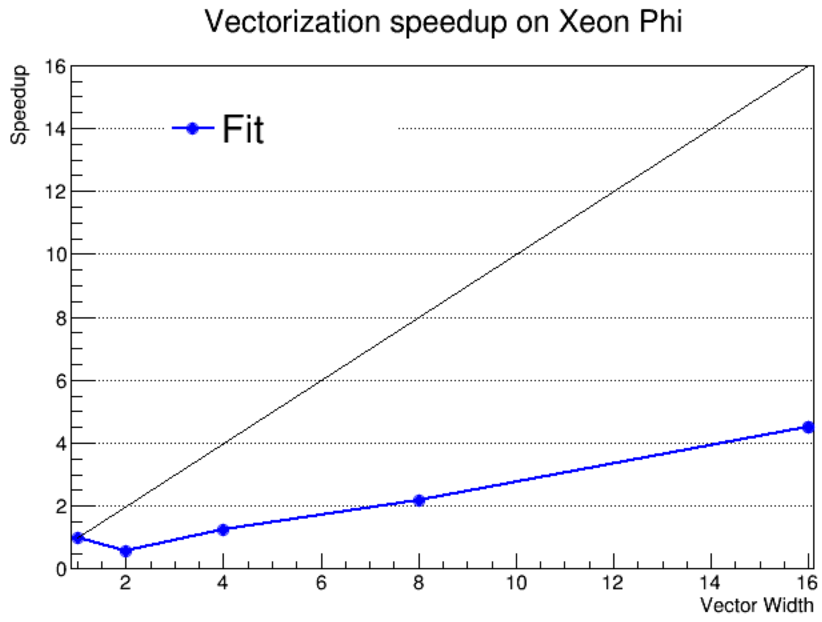
- Each individual matrix is small: 3x3 or 6x6, and may be symmetric
- Store in "matrix-major" order so 16 matrices work in sync (SIMD)
- Potential for 60 vector units on MIC to work on 960 tracks at once!


R1
R2
⋮
Rn
vector unit
fast memory direction
$M^1(1,1)$ $M^1(1,2)$ … $M^1(1,N)$ $M^1(2,1)$ …, … $M^1(N,N)$ $M^{n+1}(1,1)$ $M^{n+1}(1,2)$ … $M^{n+1}(1,N)$ $M^{n+1}(2,1)$ …, … $M^{n+1}(N,N)$ $M^{1+2n}(1,1)$
$M^2(1,1)$ $M^2(1,2)$ … $M^2(1,N)$ $M^2(2,1)$ …, … $M^2(N,N)$ $M^{n+2}(1,1)$ $M^{n+2}(1,2)$ … $M^{n+2}(1,N)$ $M^{n+2}(2,1)$ …, … $M^{n+2}(N,N)$
$M^n(1,1)$ $M^n(1,2)$ … $M^n(1,N)$ $M^n(2,1)$ … $M^n(N,N)$ $M^{2n}(0,0)$ $M^{2n}(0,1)$ … $M^{2n}(0,N)$ $M^{2n}(1,0)$ … $M^{2n}(N,N)$ $M^{3n}(0,0)$

Matrix size **NxN**, vector unit size **n** = 16 for MIC → **data parallelism**

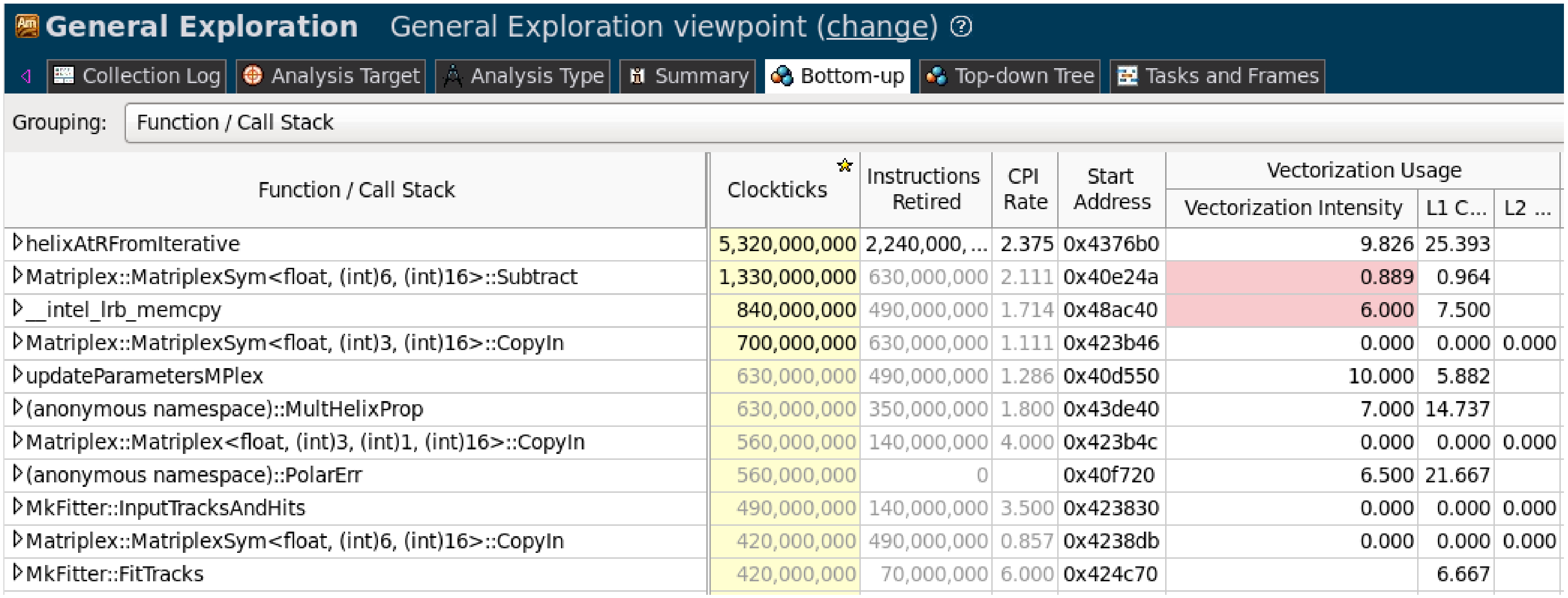
How Well Does Matriplex Work?





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

Clues From Intel’s VTune



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	

- Spending 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 calling routine...

```
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 is both input and output. But we know that is totally safe, because Subtract just runs element-wise through the arrays
- Compiler must often make conservative assumptions by default

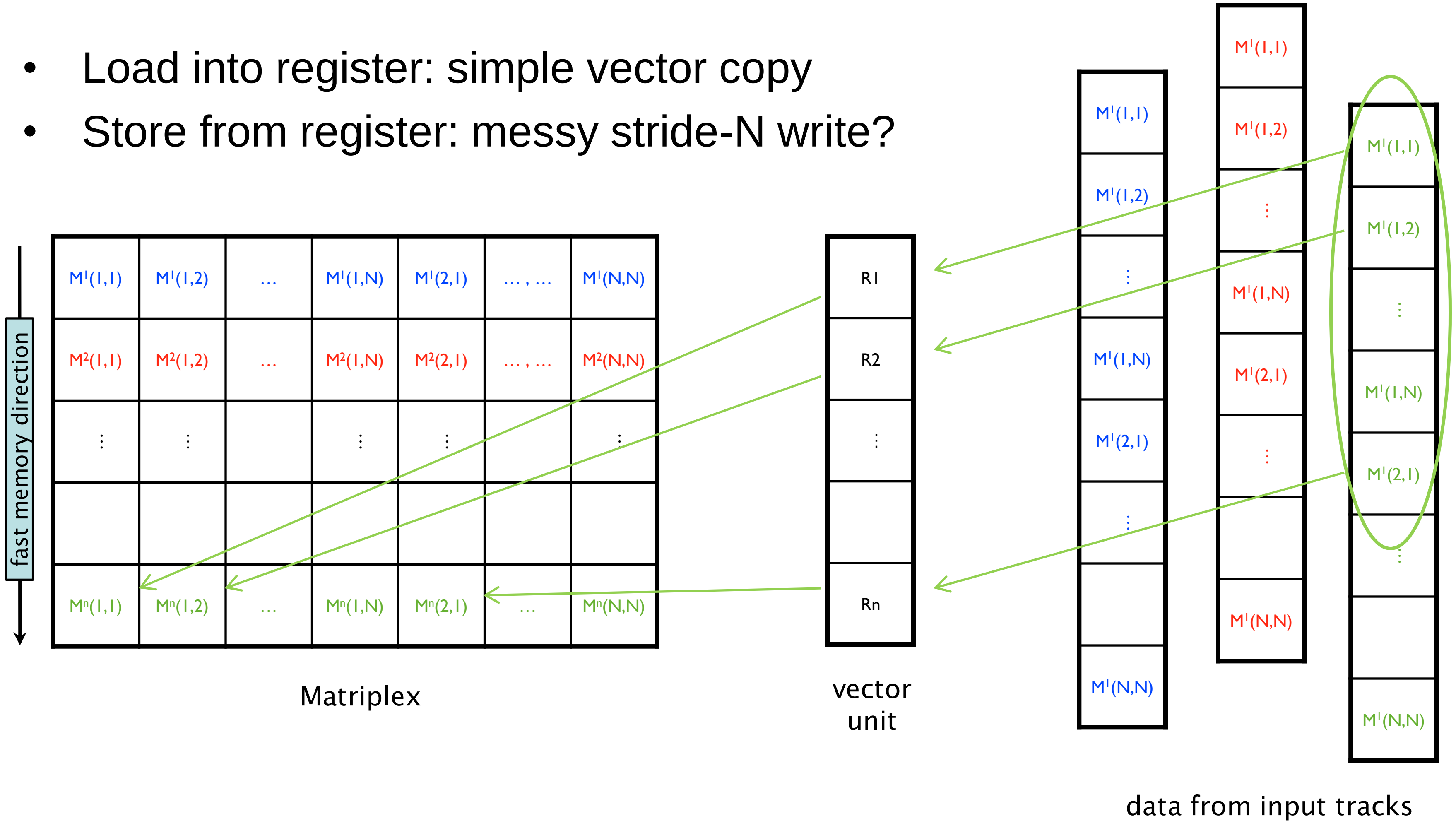
Fixing the False Vector Dependence

- Just add a pragma to **i**gnore **v**ector **dep**endence
- Single change gives ~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?



Matriplex::CopyIn

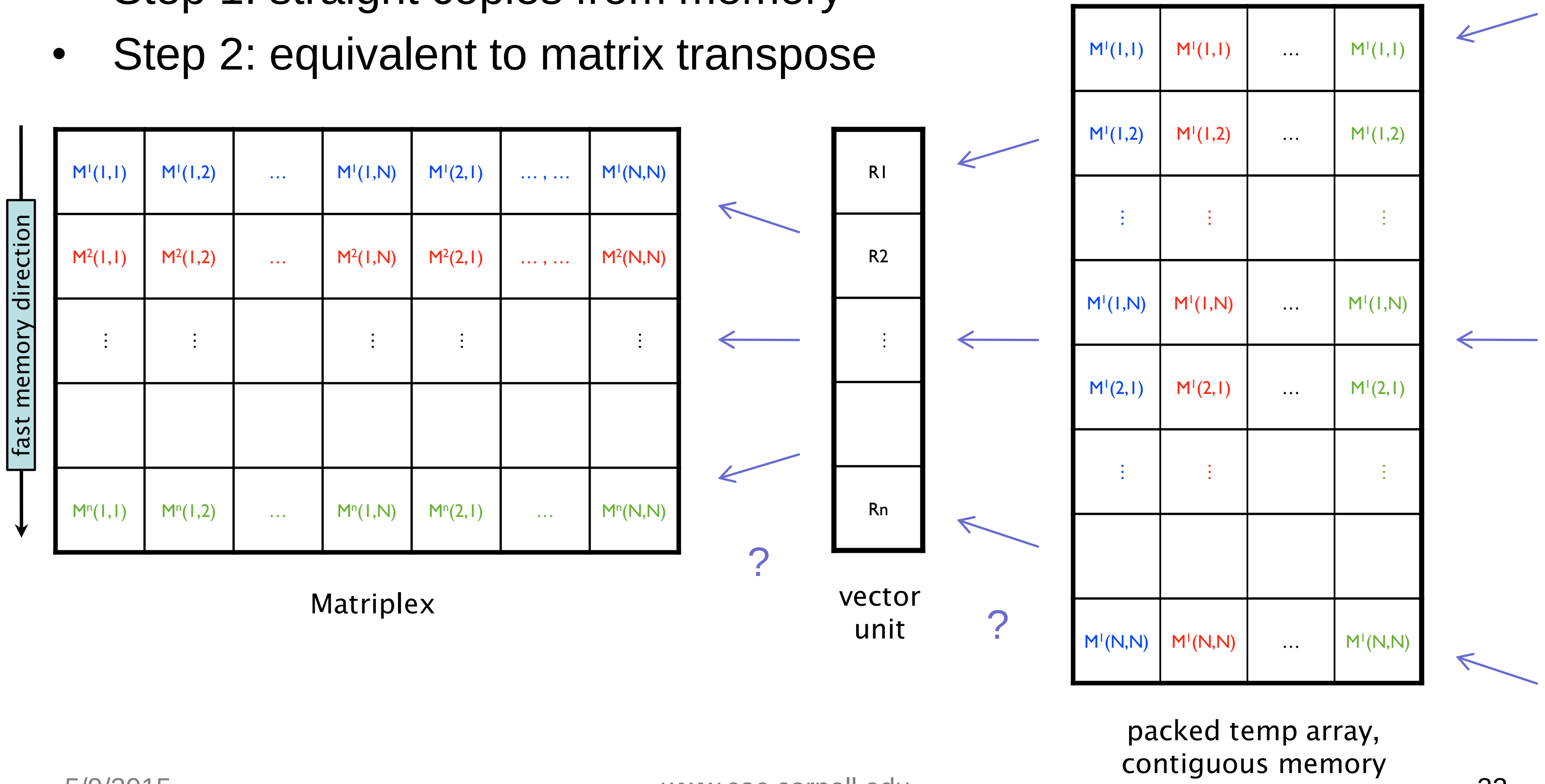
- Takes a single array as input and spreads it into fArray so that it occupies the n-th position in the Matriplex ordering (0 < n < N–1)

```
void CopyIn(idx_t n, T *arr)
{
   for (idx_t i = n; i < kTotSize; i += N)
   {
      fArray[i] = *(arr++);
   }
}
```

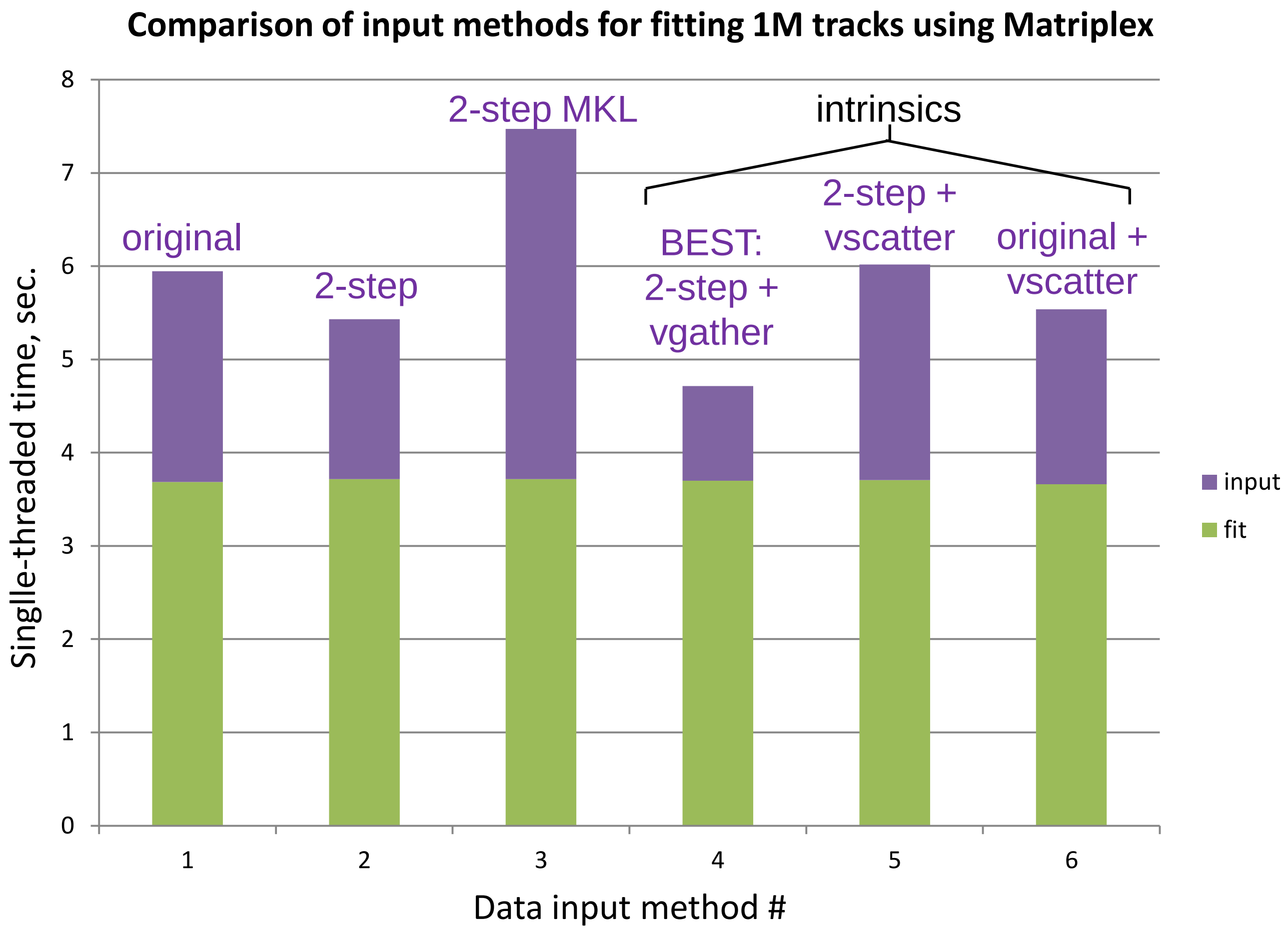
~~Will it blend?~~ **Will it vectorize?** (Answer: no!)

Redesign: Two-Step Initialization of Matriplex

- Step 1: straight copies from memory
- Step 2: equivalent to matrix transpose

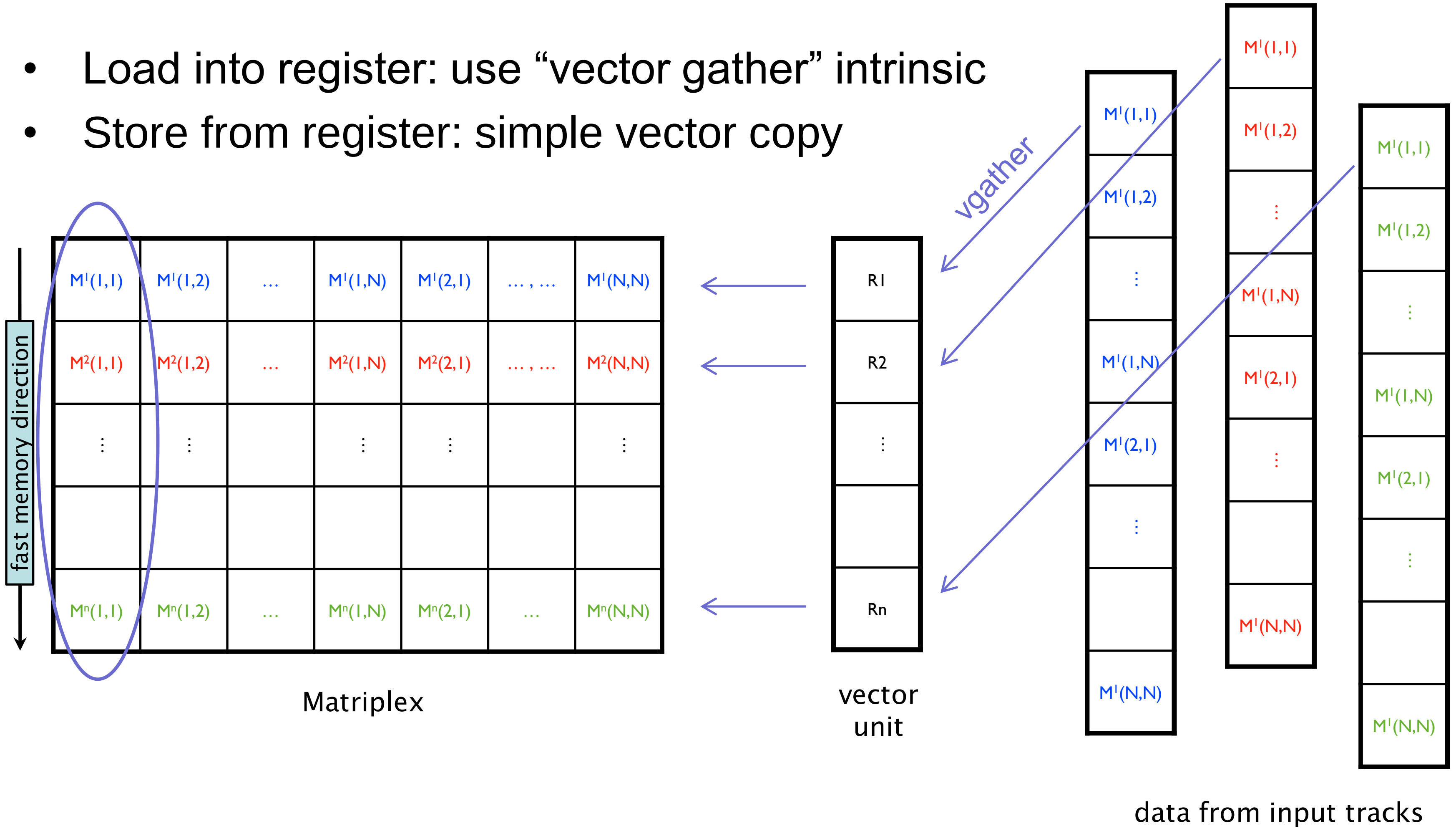


What We Already Knew : CHEP 2015 Results

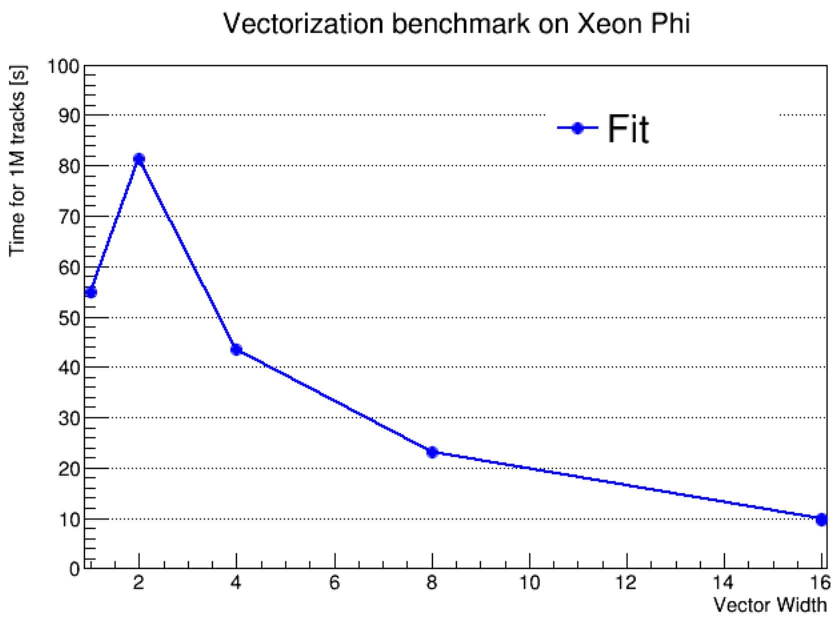


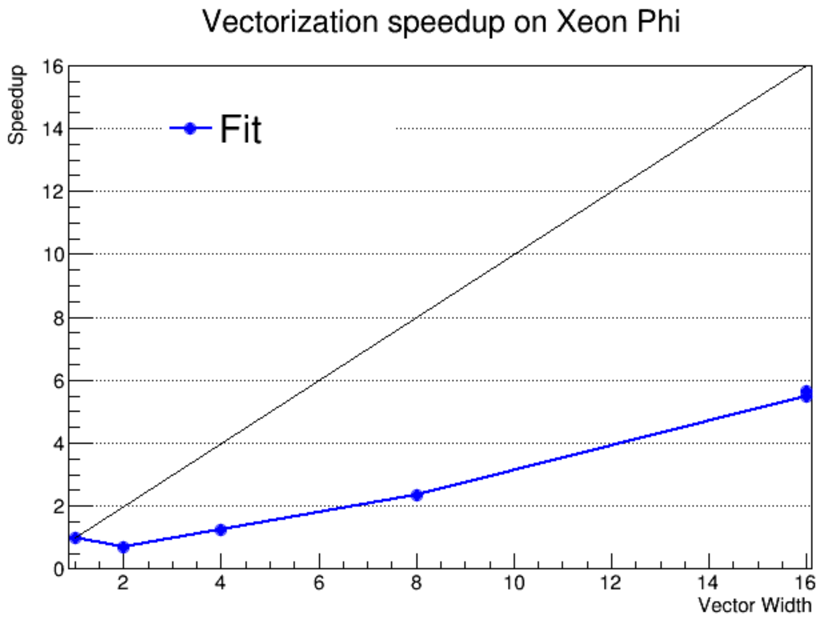
SlurpIn: Faster, One-Pass Initialization of Matriplex

- Load into register: use "vector gather" intrinsic
- Store from register: simple vector copy



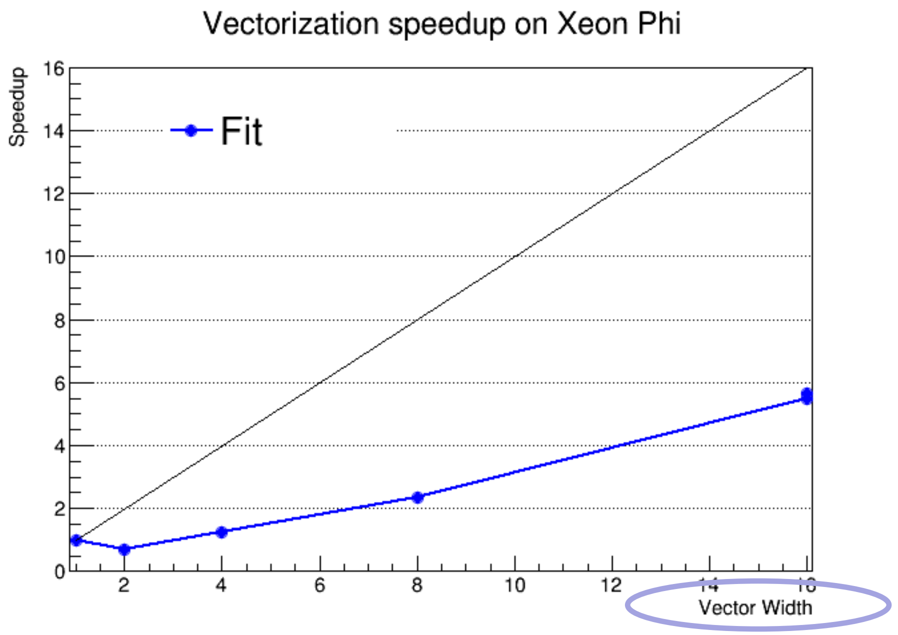
How Well Does Matriplex Work Now?

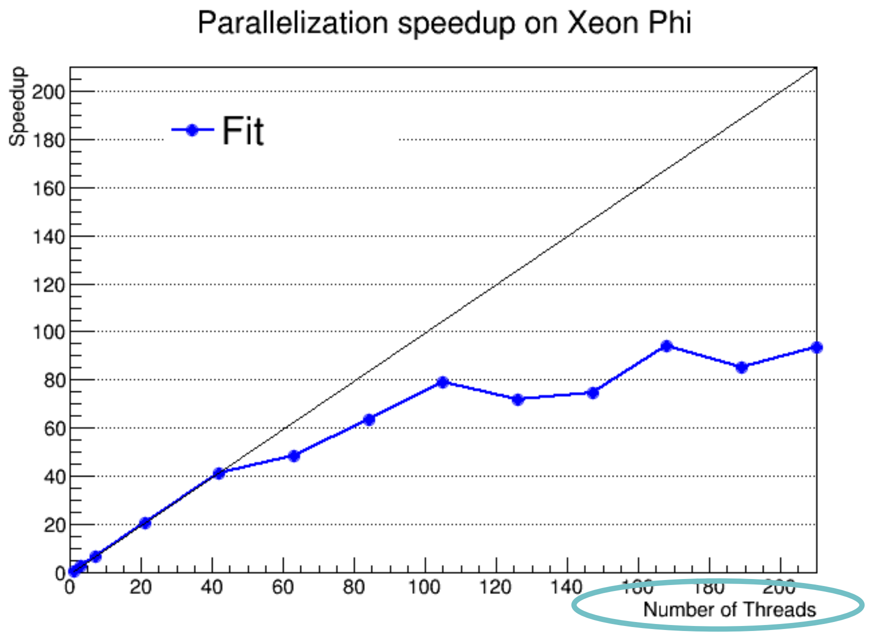




- After fixing Subtract and switching to SlurpIn, test runs 25% faster at full vector width, maximum speedup goes from ~4.4x to ~5.6x
- Amdahl's Law: *can't* get full speedup until *everything* is vectorized

Non-Ideal, But Good Use of MIC for Track Fitting!

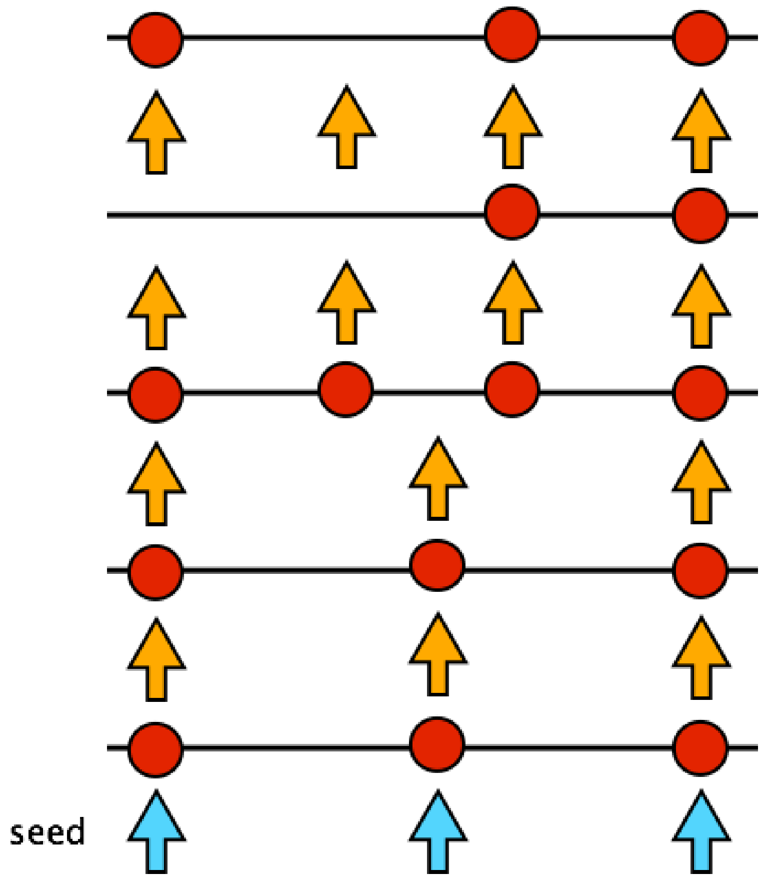




- Fitting is vectorized with Matriplex <u>and</u> parallelized using OpenMP
- Same simulated physics results as production code, but faster
  - Effective performance of vectorization is only about 40% utilization
  - Parallelization performance is close to ideal, in case of 1 thread/core

Track Building

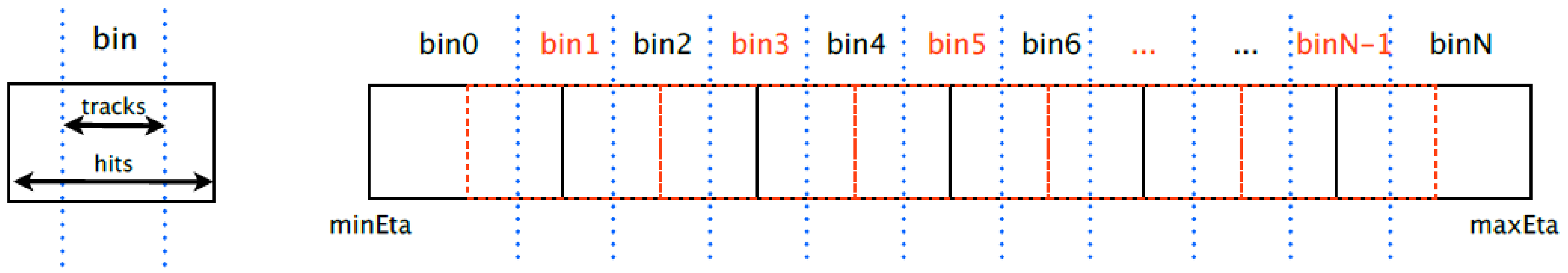
- Building is harder than fitting
- After propagating a track candidate to the next layer, hits are searched for within a compatibility window
- Track candidate needs to ***branch*** in case of multiple compatible hits
  - The algorithm needs to be robust against missing/outlier hits
- Due to branching,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



Strategy for Track Building

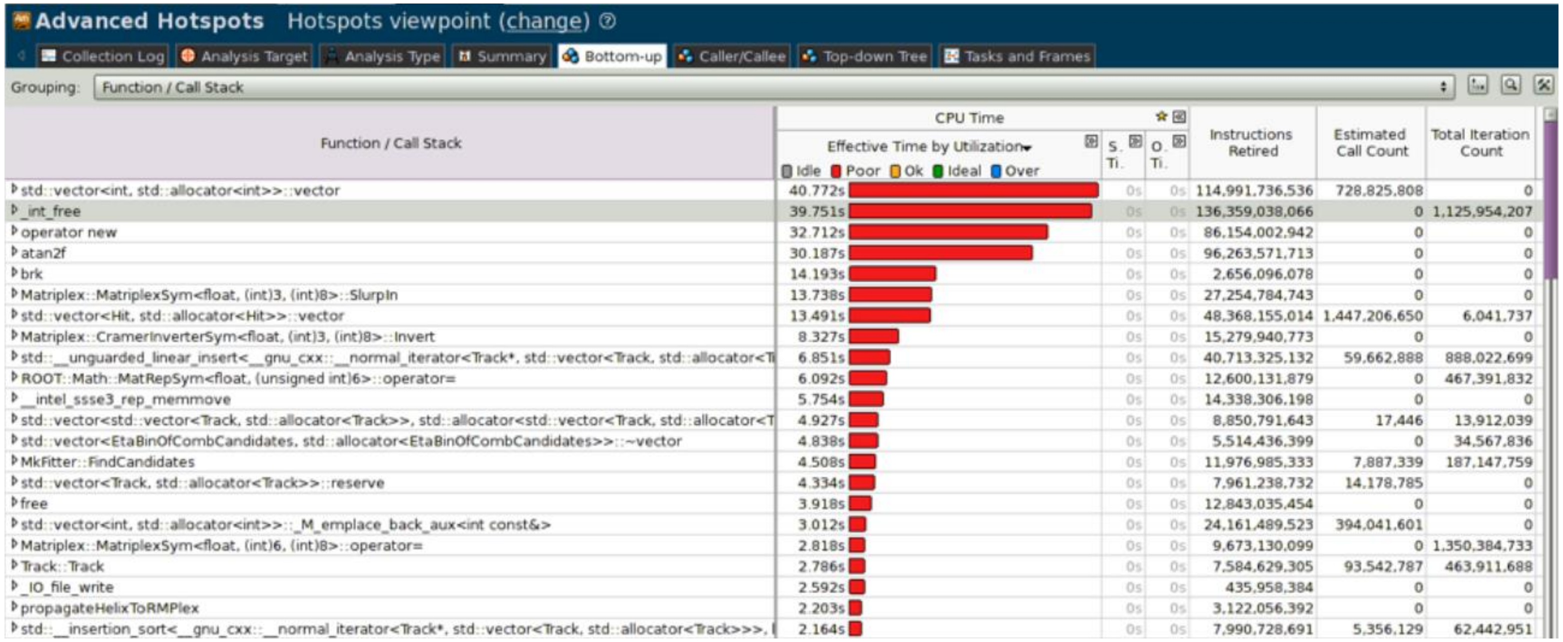
- Keep the same goal of vectorizing and multithreading all operations
  - Vectorize by continuing to use Matriplex, just as in fitting
  - Multithread by binning tracks in eta (related to angle from axis)
- Add two big complications
  - *Hit selection:* hit(s) on next layer must be selected from ~10k 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

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**

Memory Access Problems

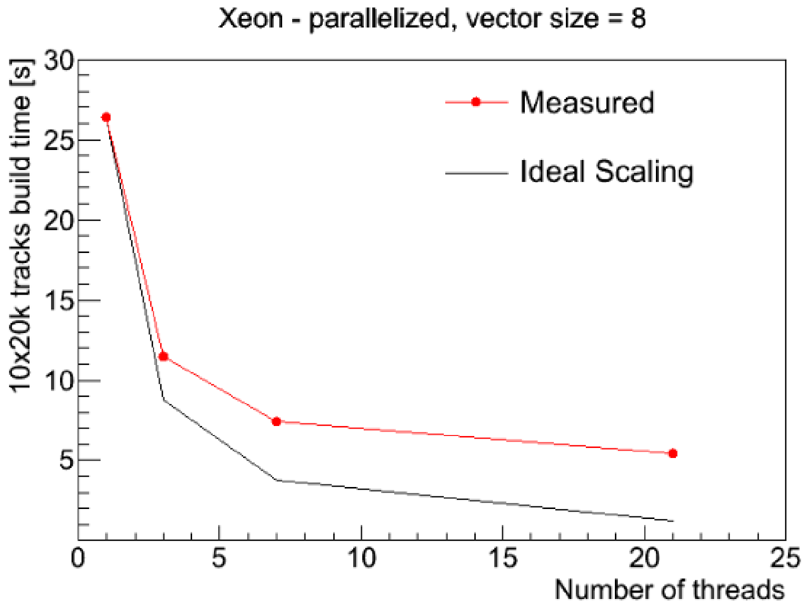

Advanced Hotspots Hotspots viewpoint (change)

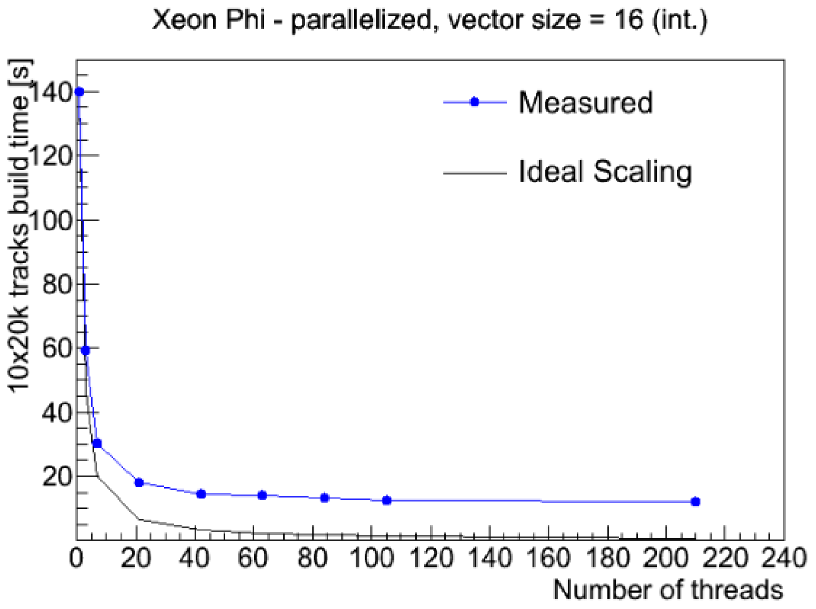
Grouping: Function / Call Stack

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

Scaling Problems





- Test parallelization by distributing threads across 21 eta bins
  - For nEtaBin/nThreads = j > 1, assign j eta bins to each thread
  - For nThreads/nEtaBin = j > 1, assign j threads to each eta bin
- Observe poor scaling and saturation of speedup

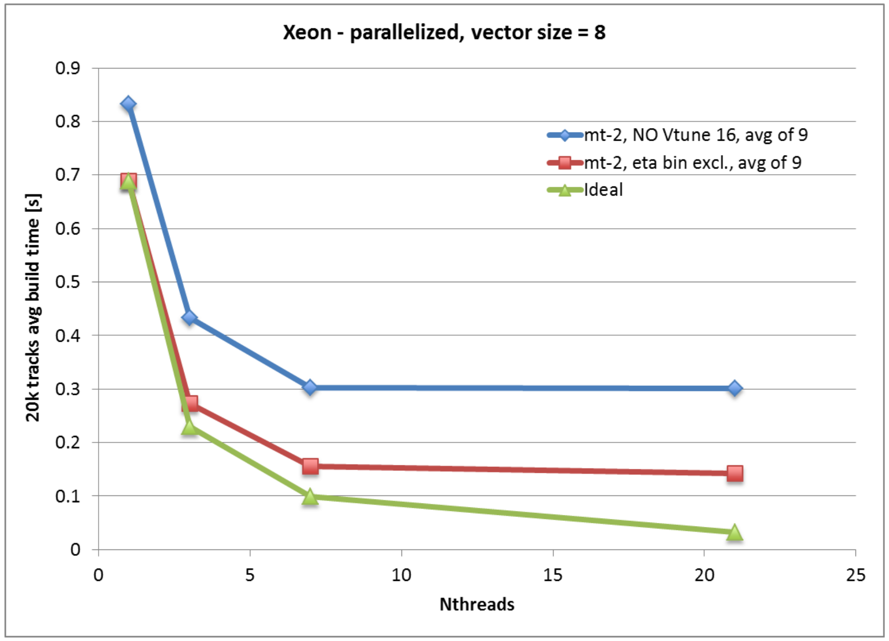
Amdahl's Law

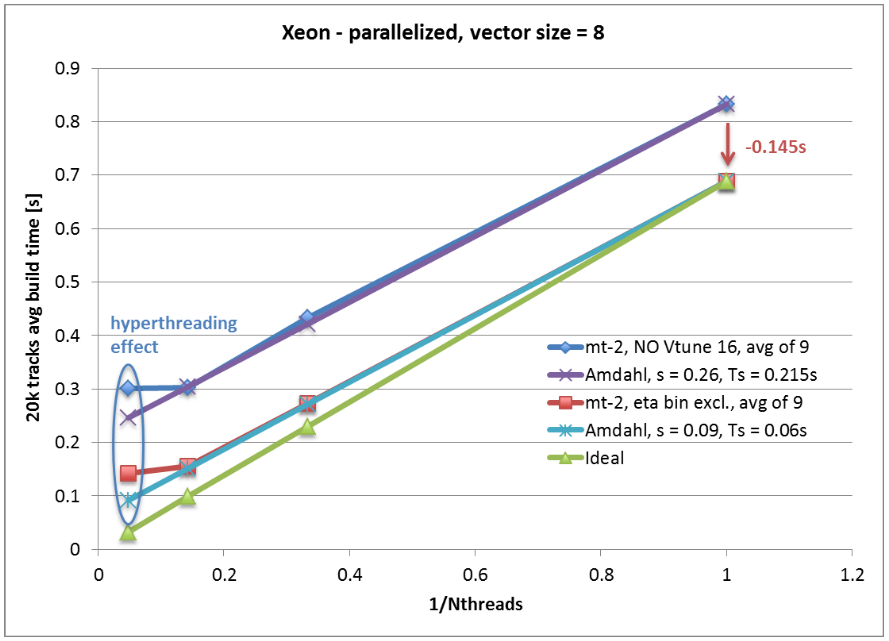
- Possible explanation: some fraction *B* of work is a serial bottleneck
- If so, the minimum time for *n* threads is set by Amdahl's Law

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

- Note, asymptote as $n \to \infty$ is not zero, but $T(1)B$
- Idea: plot the scaling data 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!





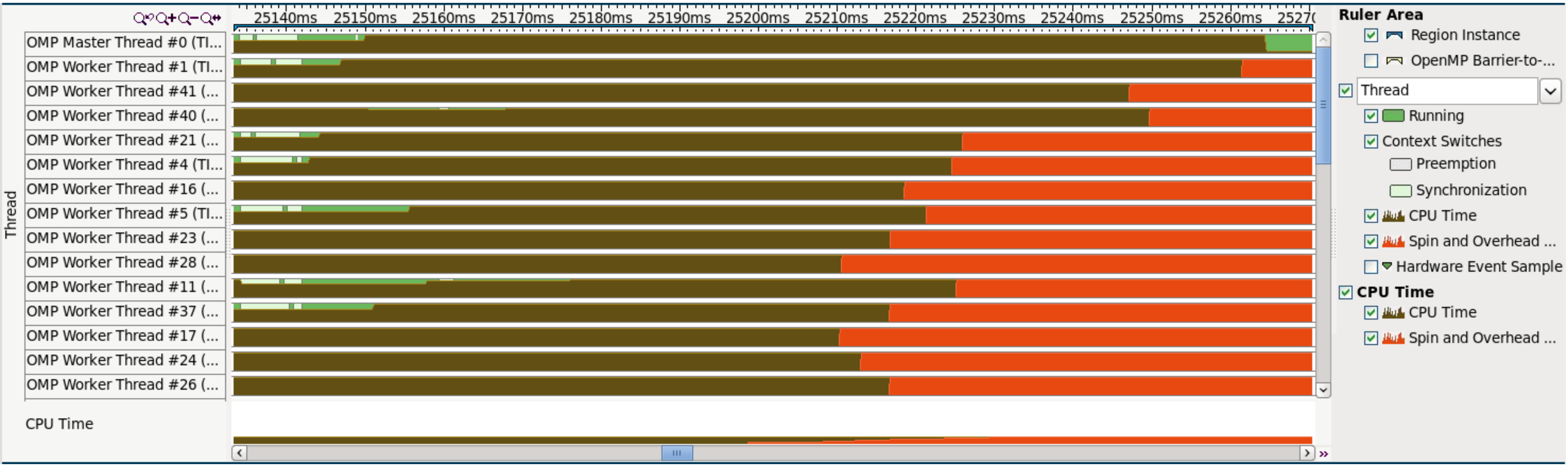
- Huge improvement from excluding *one code line* creating eta bins

```
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

What Else Is Going On?

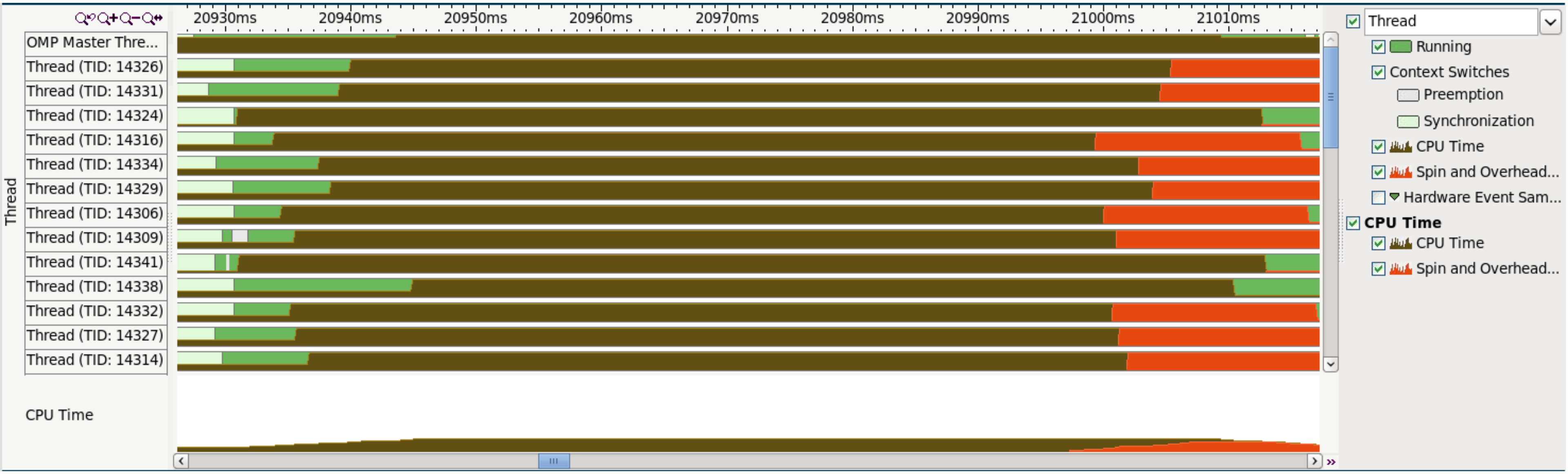
- VTune reveals non-uniformity of occupancy within threads, i.e., some threads take far longer than others: *load imbalance*
  - Worsens as threads increase: test below uses 42 threads on MIC


25140ms 25150ms 25160ms 25170ms 25180ms 25190ms 25200ms 25210ms 25220ms 25230ms 25240ms 25250ms 25260ms
Thread
OMP Master Thread #0 (TI...
OMP Worker Thread #1 (TI...
OMP Worker Thread #41 (...
OMP Worker Thread #40 (...
OMP Worker Thread #21 (...
OMP Worker Thread #4 (TI...
OMP Worker Thread #16 (...
OMP Worker Thread #5 (TI...
OMP Worker Thread #23 (...
OMP Worker Thread #28 (...
OMP Worker Thread #11 (...
OMP Worker Thread #37 (...
OMP Worker Thread #17 (...
OMP Worker Thread #24 (...
OMP Worker Thread #26 (...
CPU Time
Ruler Area
Region Instance
OpenMP Barrier-to-...
Thread
Running
Context Switches
Preemption
Synchronization
CPU Time
Spin and Overhead ...
Hardware Event Sample
CPU Time
CPU Time
Spin and Overhead ...

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

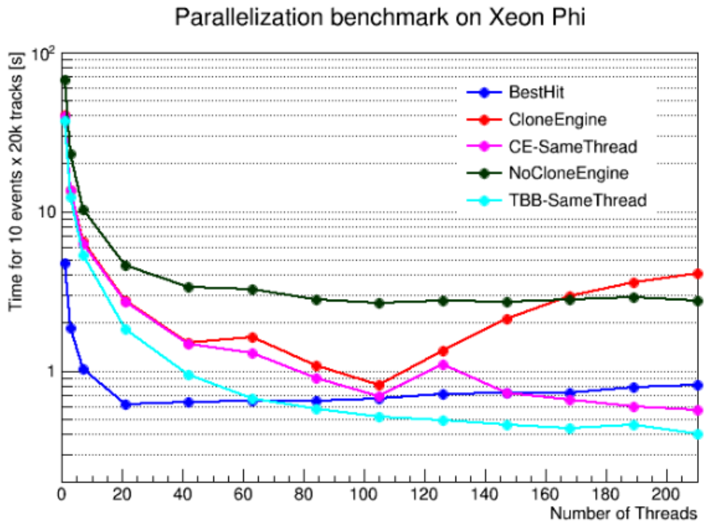
Improvement with Intel Threading Building Blocks

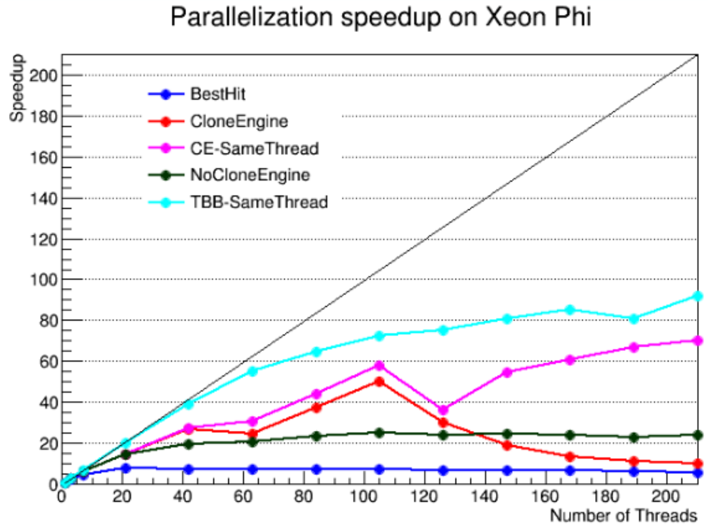
- TBB allows eta bins to be processed by varying numbers of threads
- Allows idle threads to steal work from busy ones
- Much better load balance



Scaling Tests

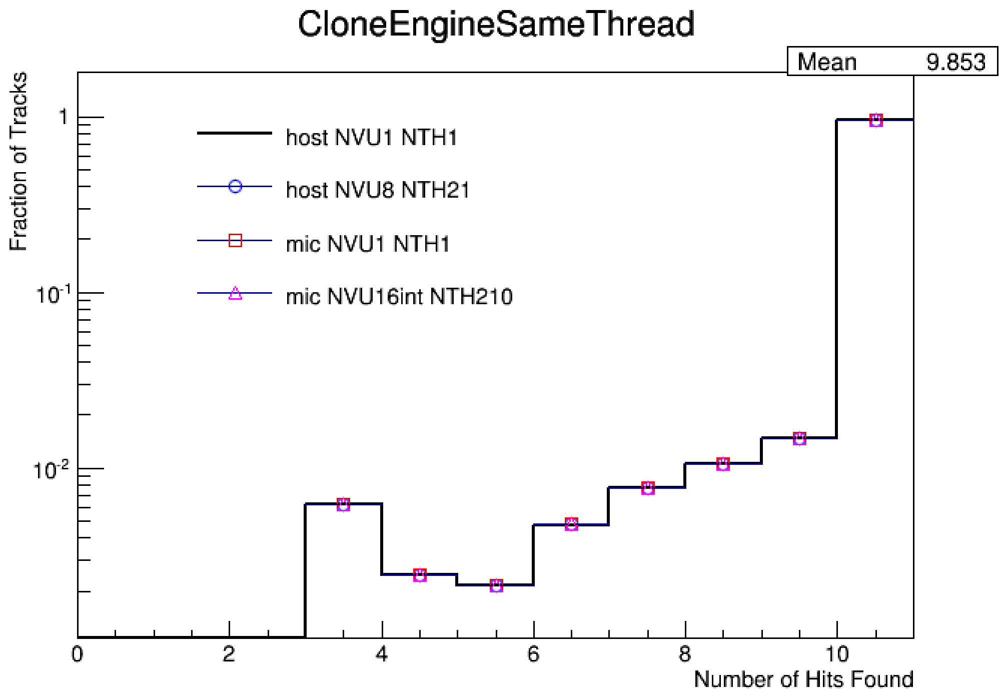
- Benchmark for the building test is the average time to perform tracking for 10 events emitting 20k charged particles each
- TBB appears to be much better at keeping all the Xeon Phi cores busy





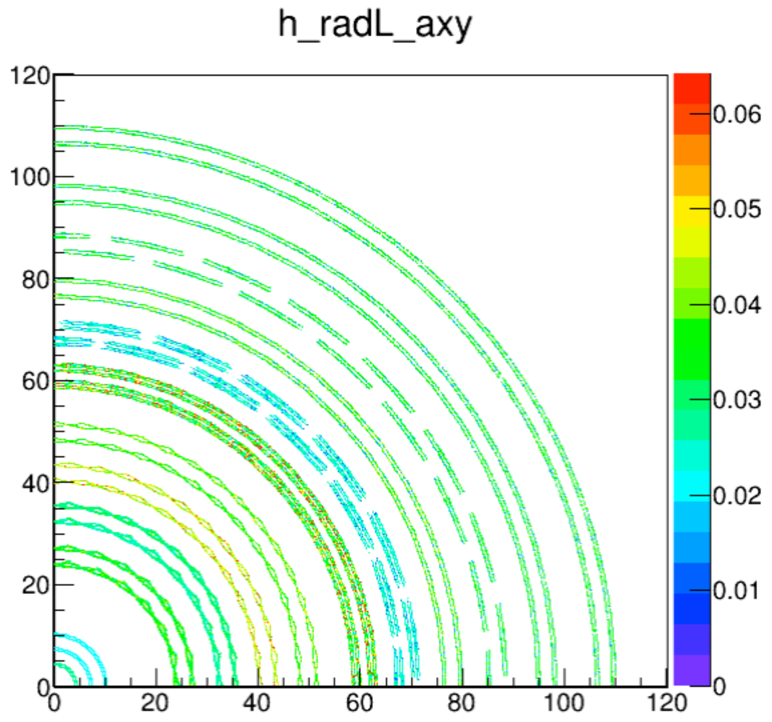
Track Building Test Actually Works, Too

- Each simulated track should have hits in all 10 detector layers
- On average, track builder finds 9.85 hits per track



Conclusions: Tracking R&D

- Significant progress in creating parallelized and vectorized tracking software on Xeon/Xeon Phi
  - Good understanding of bottlenecks
  - Intel VTune has become a key tool
  - Started a port to GPUs (CUDA)
- Better physics results, too
  - Transform momentum into *curvature* at each detector layer to get a better error estimate and find more tracks
- Encouraging tests on realistic data
- Still need to incorporate realistic geometry and materials



The project is solid and promising but we still have a long way to go

Conclusions: HPC in the Many-Core Era

- HPC has moved beyond giant clusters that rely on coarse-grained parallelism and MPI (Message Passing Interface) communication
  - *Coarse-grained*: big tasks are parceled out to a cluster
  - *MPI*: tasks pass messages to each other over a local network
- HPC now also involves many-core engines that rely on fine-grained parallelism and SIMD within shared memory
  - *Fine-grained*: threads run numerous subtasks on low-power cores
  - *SIMD*: subtasks act upon multiple sets of operands simultaneously
- Many-core is quickly becoming the norm in laptops, other devices
- *Programmers who want their code to run fast must consider how each big task breaks down into smaller parallel chunks*
  - Multithreading must be enabled explicitly through OpenMP or an API
  - Compilers can vectorize loops automatically, if data are arranged well