Particle Track Reconstruction for the Large Hadron Collider: A Case Study in Intra-Processor Parallelism

Steve Lantz

Senior Research Associate

Cornell University Center for Advanced Computing (CAC)

https://www.cac.cornell.edu/slantz

steve.lantz@cornell.edu

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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. 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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions

High Performance Computing in High Energy Physics

<u>Collaborators</u>

K. McDermott, 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).

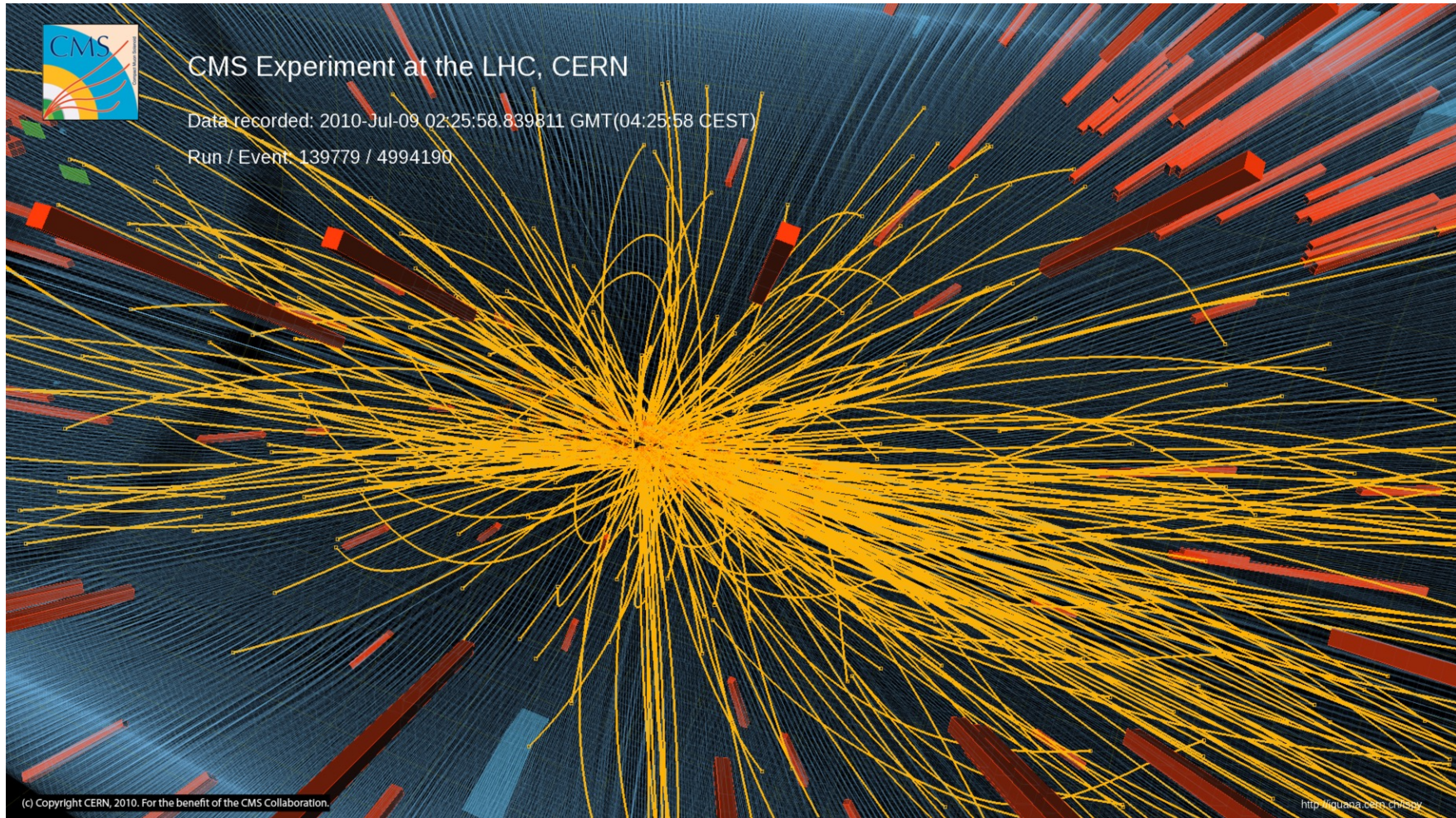
Photo: CMS detector, LHC, CERN

LHC: The *Super Collider*

The **Compact Muon Solenoid** (CMS) is one of the detectors in the LHC (actual photo)

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

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



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

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

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

FOX NEWS .com Fair & Balanced

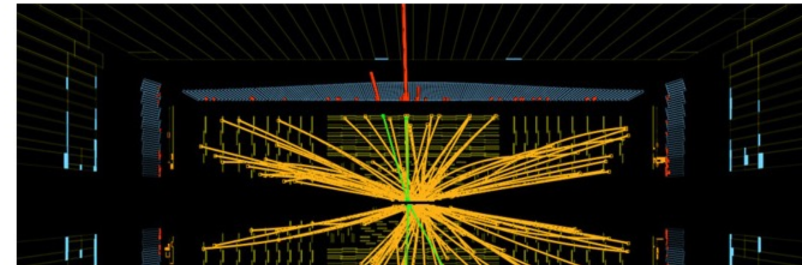
Home | Video | Politics | U.S. | Opinion

Science Home | Archaeology | Air & Space | Planet E

BREAKING NEWS | ISRAEL SAYS IT SH

The elusive particle: Higgs Boson

Published July 05, 2012 / LiveScience



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

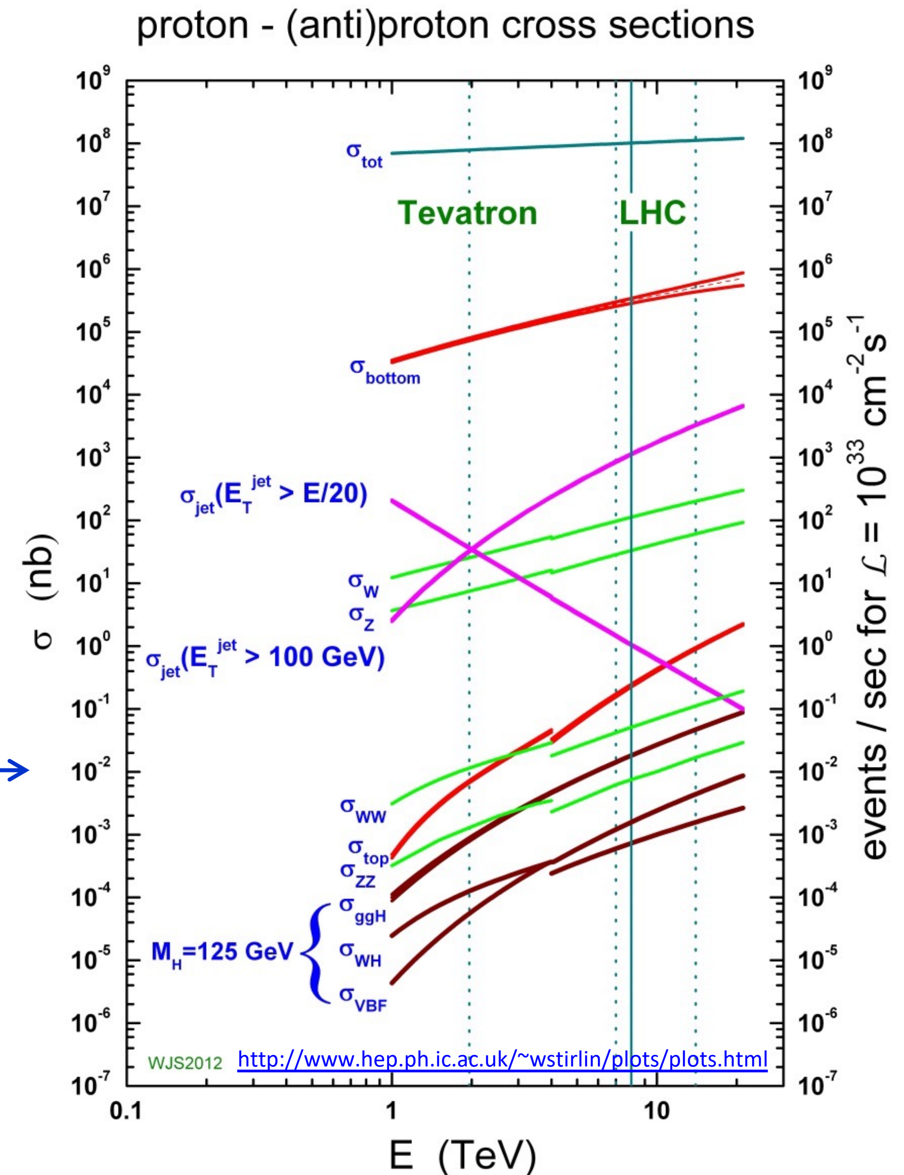
04 July 2012

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.

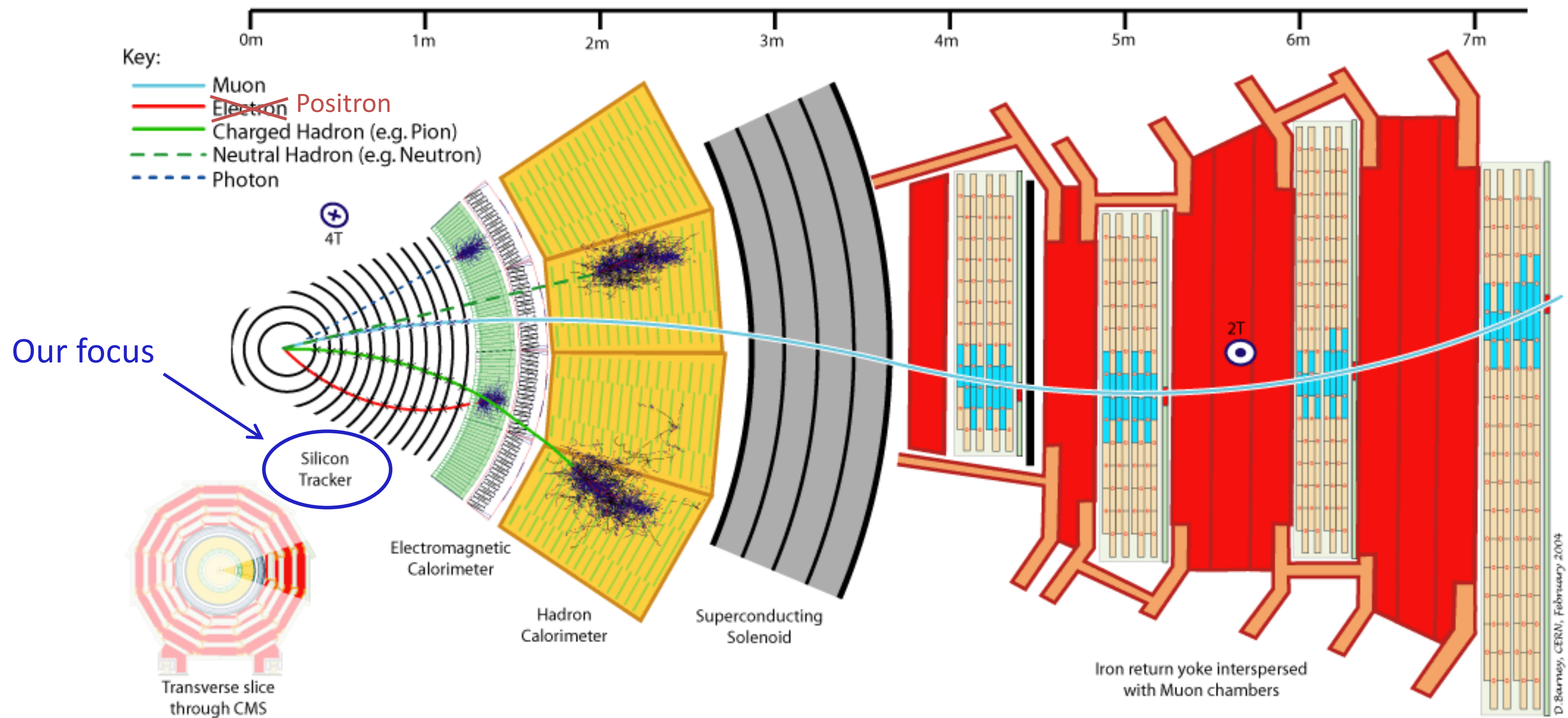
Two independent experiments reported their

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



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

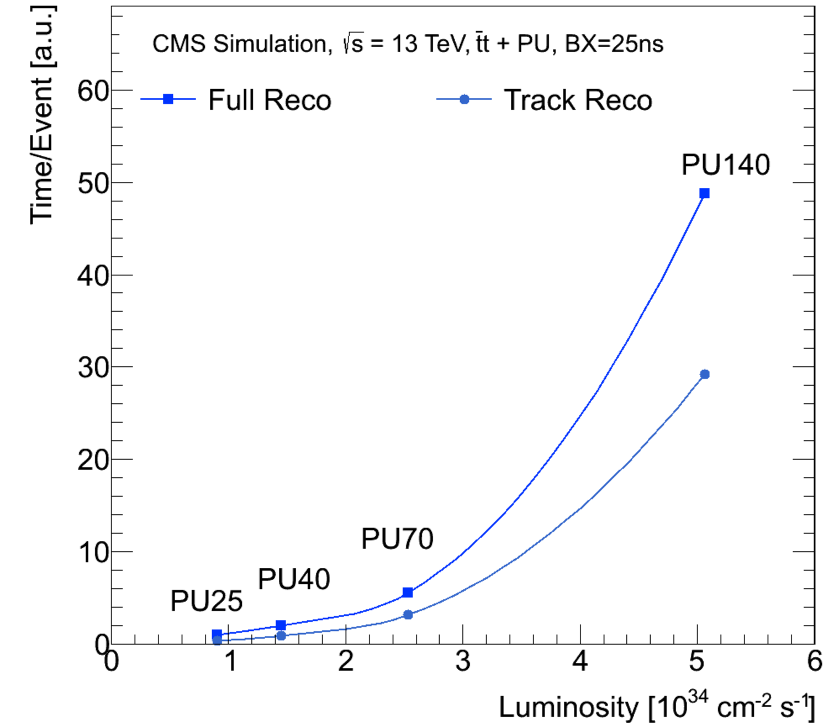
CMS Is About to Get Busier

Simulation of pile-up = 140 at CMS in r-z plane

- 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

48 Years of Microprocessor Trend Data

discontinuity in ~2005

Transistors (thousands)

Single-Thread Performance (SpecINT x 10^3)

Frequency (MHz)

Typical Power (Watts)

Number of Logical Cores

10^0 10^1 10^2 10^3 10^4 10^5 10^6 10^7

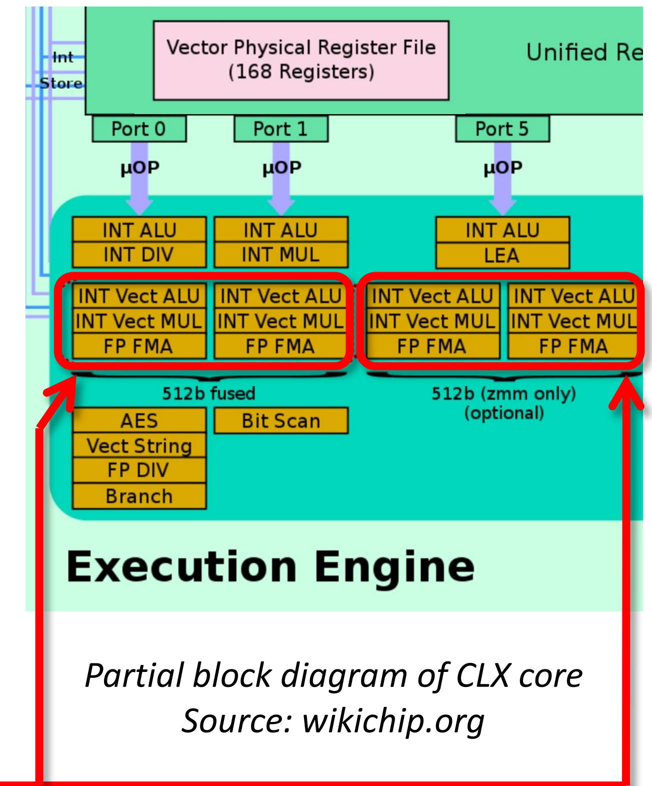
1970 1980 1990 2000 2010 2020

Year

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

Vector Hardware on CPUs

- SIMD = Single Instruction, Multiple Data
  - Part of commodity CPUs (x86, x64, PowerPC) since late ’90s
- Goal: parallelize computations on vector arrays
  - Line up operands, execute one op on all simultaneously
- SIMD instructions have gotten speedier over time
  - Initially: several cycles for execution on small vectors
  - Intel AVX introduced pipelining of some SIMD instructions
  - Now: multiply-and-add large vectors on every cycle
- Intel’s latest: Cascade Lake, Ice Lake, Sapphire Rapids…
  - 2 VPUs (vector processing units) available per core
  - 2 ops/VPU if they do FMAs (Fused Multiply-Add) every cycle



Partial block diagram of CLX core
Source: wikichip.org

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?)

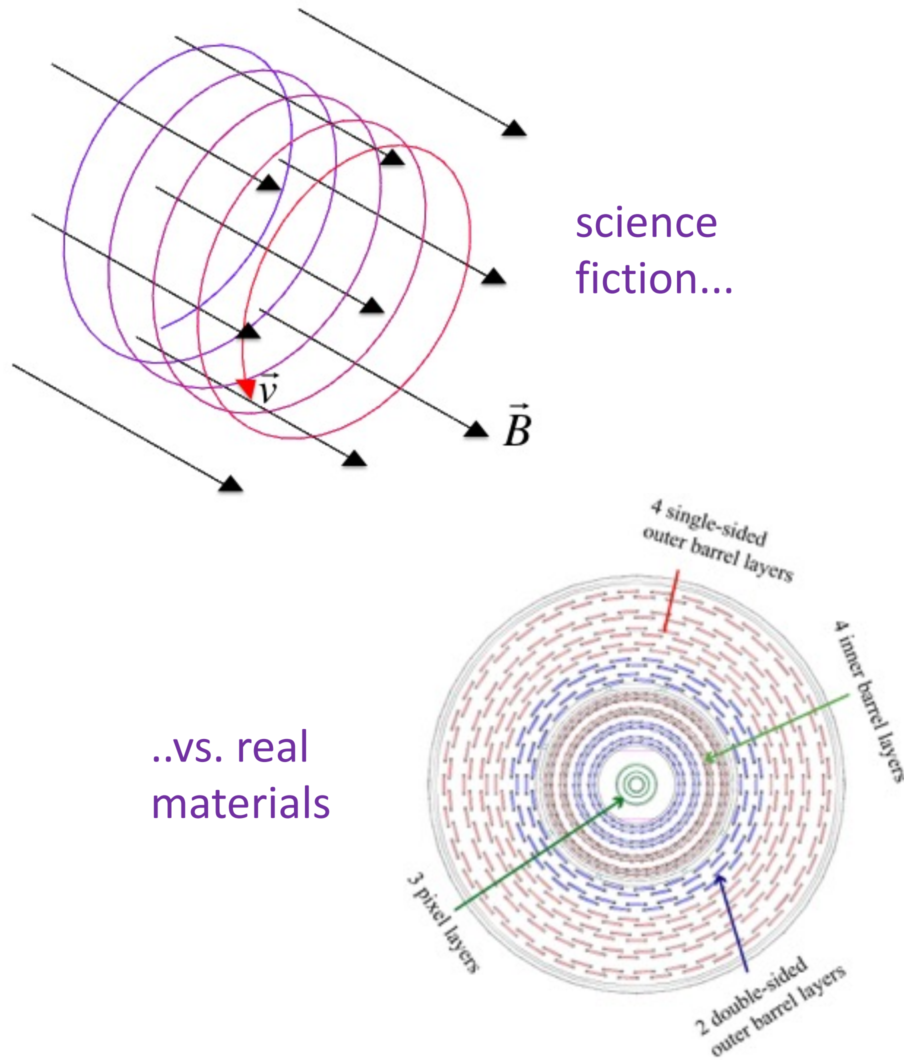
Outline

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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions

History of the Trackreco/mkFit Project

- 2015 NSF PIF (Physics at the Information Frontier) grant: *“Particle Tracking at High Luminosity on Heterogeneous, Parallel Processor Architectures”*
  - Cornell, Princeton, UCSD ➛ all CMS
  - HL-LHC: high pile-up, 200 interactions per bunch crossing
  - New (at the time) computer architectures: MIC / AVX-512, GPUs, ARM-64
  - Goal: make tracking software more general and faster!
- Proposal: enhance the parallelism of existing, production tracking algorithms based on **Kalman Filter**:
  - Keep well-known physics performance – efficiencies, fake rates
  - Make code amenable to vectorization and multithreading, through new data structures and generalized algorithms

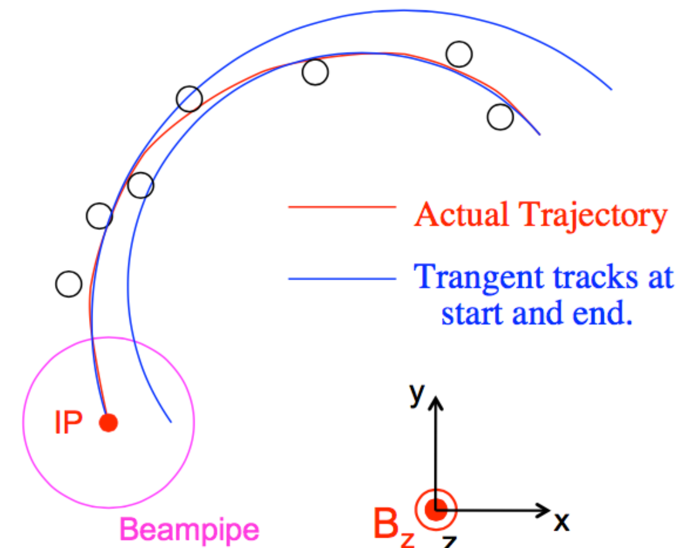
Why Kalman Filter for Particle Tracking?



- Naively, each 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

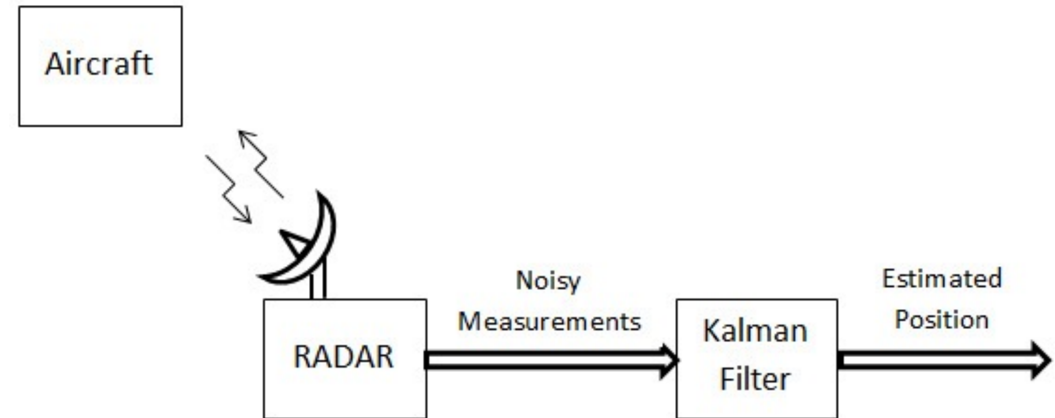
What Does the Tracking Algorithm Do?

- 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* (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

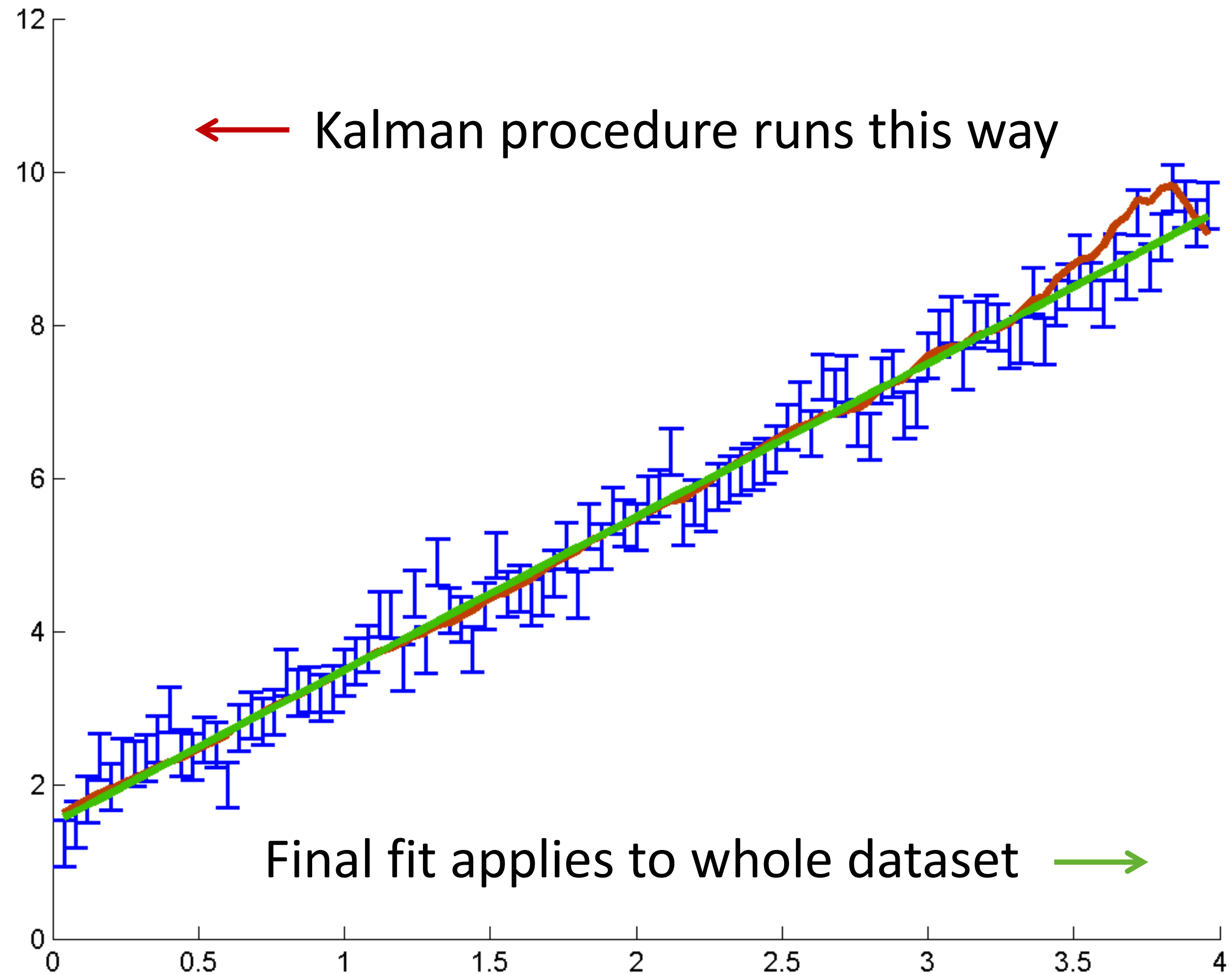
From Wikipedia, the free encyclopedia

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

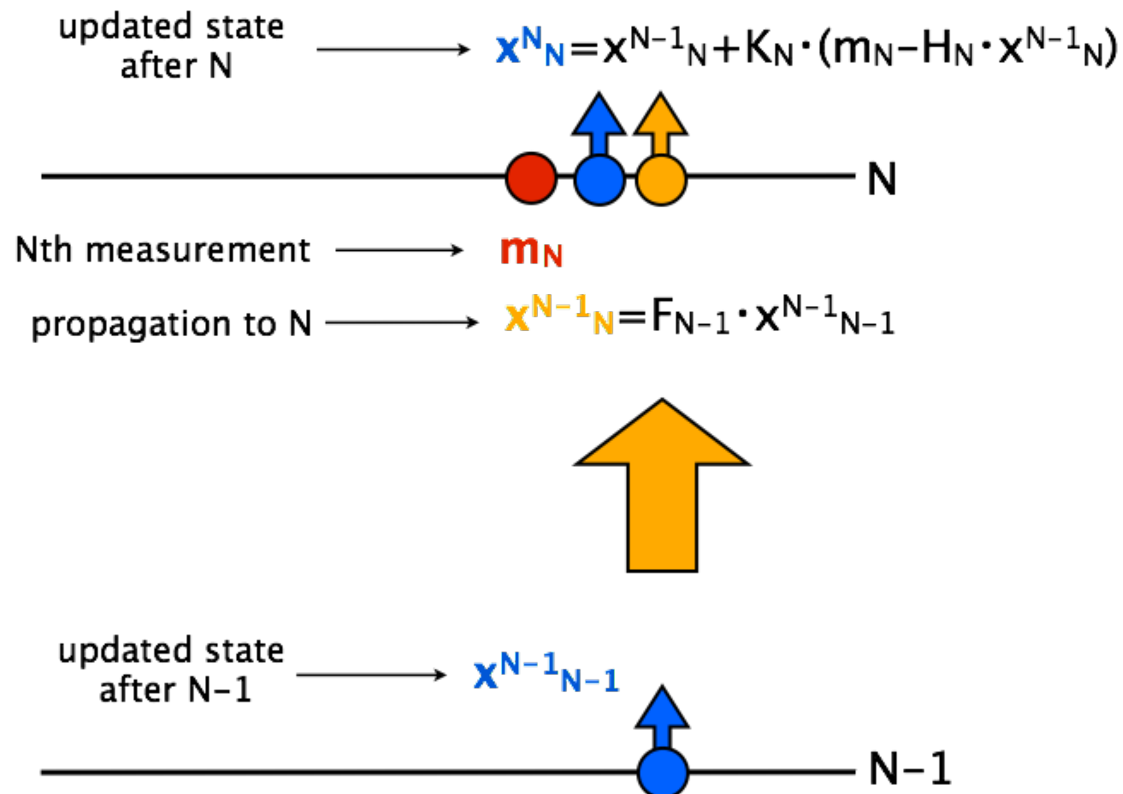
Kalman Example

- 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

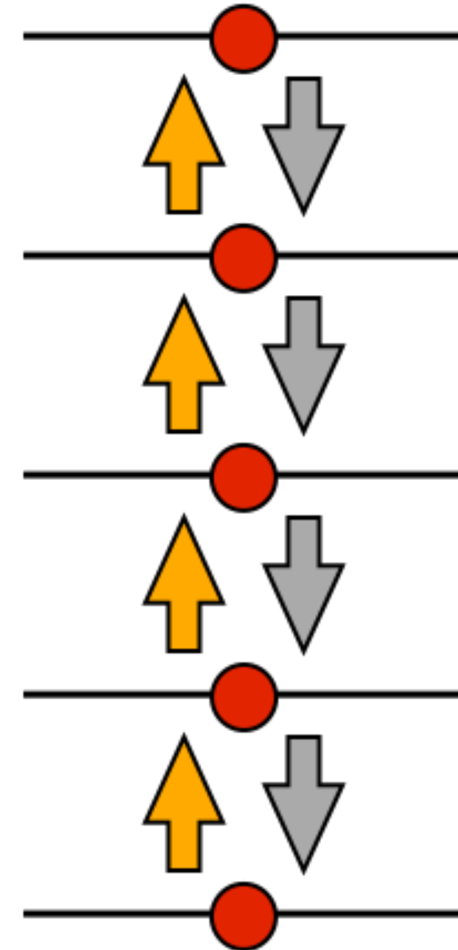
- Track reconstruction has 3 main steps: *seeding*, *building*, and *fitting*
- Building and fitting repeat the basic logic unit of the Kalman Filter...



- 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

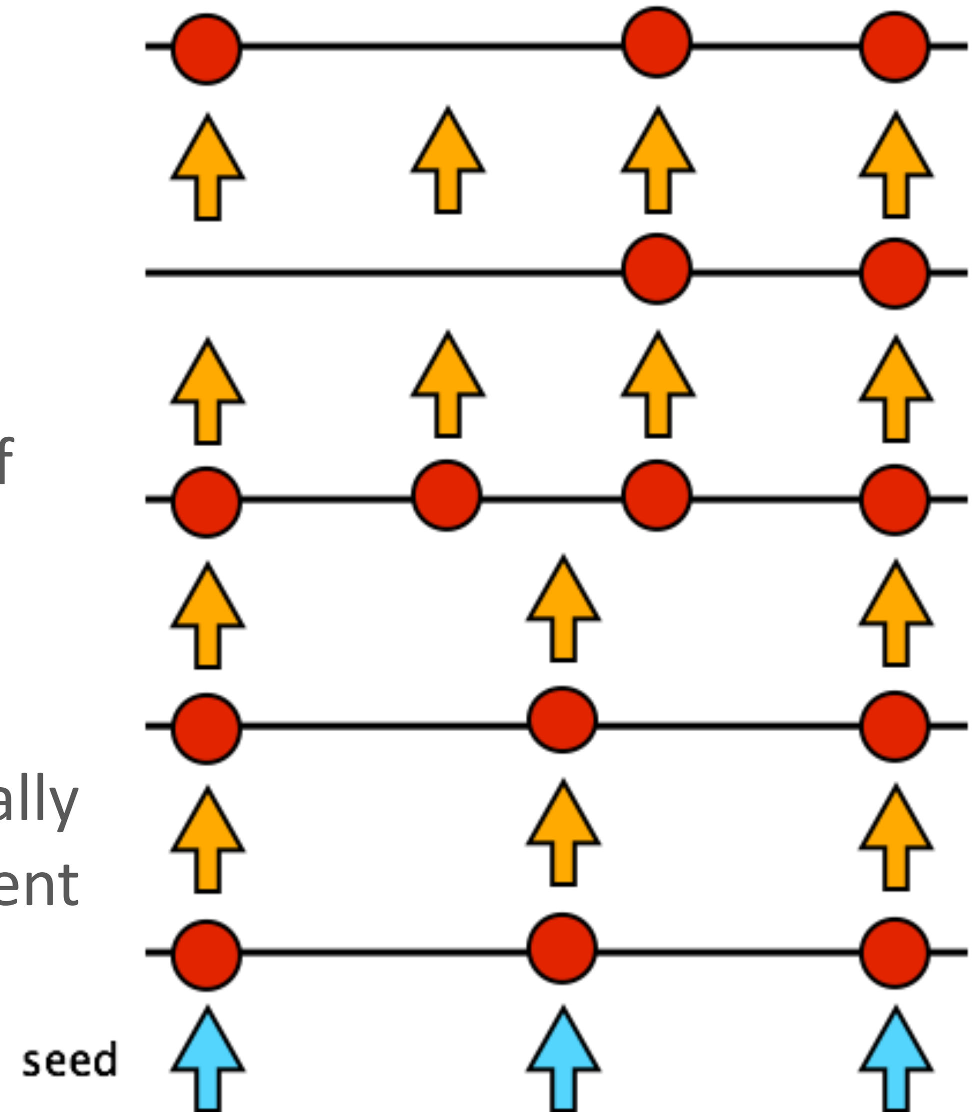
Track Fitting as Kalman Filter

- 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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions

How Do We Get Vector Speedup?

- Program the key routines in assembly…
  - Ultimate performance potential, but only for the brave
- Program the key routines using SIMD intrinsics...
  - Step up from assembly; useful in spots, but risky
- Link to an optimized library that does the heavy lifting…
  - Intel MKL, e.g., written by people who know all the tricks
  - BLAS is the portable interface for doing fast linear algebra
- Let the compiler figure it out
  - Relatively “easy” for user, “challenging” for compiler
  - Compiler may need some guidance through directives
  - Programmer can help by using simple loops and arrays

All these were tried! (refers to the last three items)

Objects in Track Finding and Fitting

- Hit: 3-vector of position, 3x3 symmetric covariance matrix, label
  - 40 bytes, a bit less than a 64-byte cache line
- Track: 6-vector of position and momentum, 6x6 symm. cov. matrix, hit indices
  - Not the most compact representation: helix has 5 parameters, 5x5 symm. cov. matrix
  - But with 6x6, the covariance matrix is block diagonal, one can do sparse matrix tricks
  - Keep just the indices of assigned hits – 256 bytes – 4 cache lines
- Kalman Filter: a set of operations using the above objects
  - Mostly multiplications; intermediate results are 6x3 matrices
  - Similarity operations that transform between measurement basis, parameter basis
  - 3x3 matrix inversion
  - Be careful, the product of symmetric matrices is not symmetric

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

R1	$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	… , …	$M^1(N,N)$	$M^{n+1}(1,1)$	$M^{n+1}(1,2)$	…	$M^{n+1}(1,N)$	$M^{n+1}(2,1)$	… , …	$M^{n+1}(N,N)$	$M^{2n+1}(1,1)$
R2	$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	… , …	$M^2(N,N)$	$M^{n+2}(1,1)$	$M^{n+2}(1,2)$	…	$M^{n+2}(1,N)$	$M^{n+2}(2,1)$	… , …	$M^{n+2}(N,N)$	$M^{2n+2}(1,1)$
⋮	⋮	⋮		⋮	⋮		⋮	⋮	⋮		⋮	⋮		⋮	
Rn	$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$	$M^{2n}(1,1)$	$M^{2n}(1,2)$	…	$M^{2n}(1,N)$	$M^{2n}(2,1)$	…	$M^{2n}(N,N)$	$M^{3n}(1,1)$

fast memory direction

vector unit

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

Matriplex Templates in C++

```
template <typename T, idx_t D1, idx_t D2, idx_t N>
class Matriplex {  // Covers also vectors with D2 = 1 and scalars with D1 = D2 = 1.
public:
  typedef T value_type;
  static constexpr int kRows = D1;
  static constexpr int kCols = D2;
  static constexpr int kSize = D1 * D2;
  static constexpr int kTotSize = N * kSize;

  T fArray[kTotSize] __attribute__((aligned(64)));
...
template <typename T, idx_t D, idx_t N>
class MatriplexSym {
public:
  typedef T value_type;
  static constexpr int kRows = D;
  static constexpr int kCols = D;
  static constexpr int kSize = (D + 1) * D / 2;
  static constexpr int kTotSize = N * kSize;

  T fArray[kTotSize] __attribute__((aligned(64)));
```

Packed into fArray are N matrices of type T, dimension D1 x D2, in “matrix-major” order, aligned on a 64-byte boundary in RAM

N-way SIMD with 3x3 Matrices

```
static void multiply(const MPlex<T, 3, 3, N>& A,
                     const MPlex<T, 3, 3, N>& B,
                     MPlex<T, 3, 3, N>& C)
{
   const T *a = A.fArray; ASSUME_ALIGNED(a, 64);
   const T *b = B.fArray; ASSUME_ALIGNED(b, 64);
         T *c = C.fArray; ASSUME_ALIGNED(c, 64);
#pragma omp simd
   for (int n = 0; n < N; ++n)
   {
      c[ 0*N+n] = a[ 0*N+n]*b[ 0*N+n] + a[ 1*N+n]*b[ 3*N+n] + a[ 2*N+n]*b[ 6*N+n];
      c[ 1*N+n] = a[ 0*N+n]*b[ 1*N+n] + a[ 1*N+n]*b[ 4*N+n] + a[ 2*N+n]*b[ 7*N+n];
      c[ 2*N+n] = a[ 0*N+n]*b[ 2*N+n] + a[ 1*N+n]*b[ 5*N+n] + a[ 2*N+n]*b[ 8*N+n];
      c[ 3*N+n] = a[ 3*N+n]*b[ 0*N+n] + a[ 4*N+n]*b[ 3*N+n] + a[ 5*N+n]*b[ 6*N+n];
      c[ 4*N+n] = a[ 3*N+n]*b[ 1*N+n] + a[ 4*N+n]*b[ 4*N+n] + a[ 5*N+n]*b[ 7*N+n];
      c[ 5*N+n] = a[ 3*N+n]*b[ 2*N+n] + a[ 4*N+n]*b[ 5*N+n] + a[ 5*N+n]*b[ 8*N+n];
      c[ 6*N+n] = a[ 6*N+n]*b[ 0*N+n] + a[ 7*N+n]*b[ 3*N+n] + a[ 8*N+n]*b[ 6*N+n];
      c[ 7*N+n] = a[ 6*N+n]*b[ 1*N+n] + a[ 7*N+n]*b[ 4*N+n] + a[ 8*N+n]*b[ 7*N+n];
      c[ 8*N+n] = a[ 6*N+n]*b[ 2*N+n] + a[ 7*N+n]*b[ 5*N+n] + a[ 8*N+n]*b[ 8*N+n];
   }
}
```

Compiler should convert each line in the loop into a single vector instruction

What About SIMD Intrinsics?

- Initial versions of the fitting code relied heavily on C++ intrinsic functions
- Improvements in compilers have largely removed the need for them
  - They are still used for packing Matriplexes from input matrices
- Intrinsics for multiplying *symmetric* matrices are still generated using Perl
  - Vectorization is otherwise tricky because only lower triangular parts are held in memory
  - To account for FMA latencies, elements are not written immediately after computation
  - Macros enable switching among SIMD intrinsics for AVX, AVX2, AVX512 →
  - The FMA instruction must be emulated for AVX, as it came in with AVX2

```
#if defined(__AVX512F__)

#define LD(a, i)      _mm512_load_ps(&a[i * N + n])
#define ST(a, i, r)  _mm512_store_ps(&a[i * N + n], r)
#define ADD(a, b)    _mm512_add_ps(a, b)
#define MUL(a, b)    _mm512_mul_ps(a, b)
#define FMA(a, b, v) _mm512_fmadd_ps(a, b, v)
```

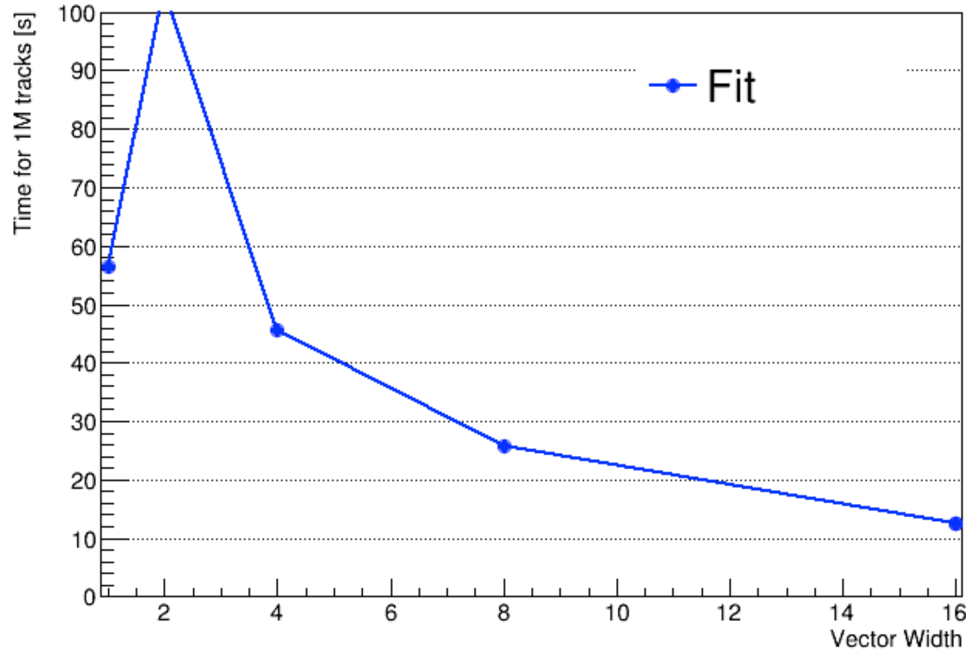
Outline

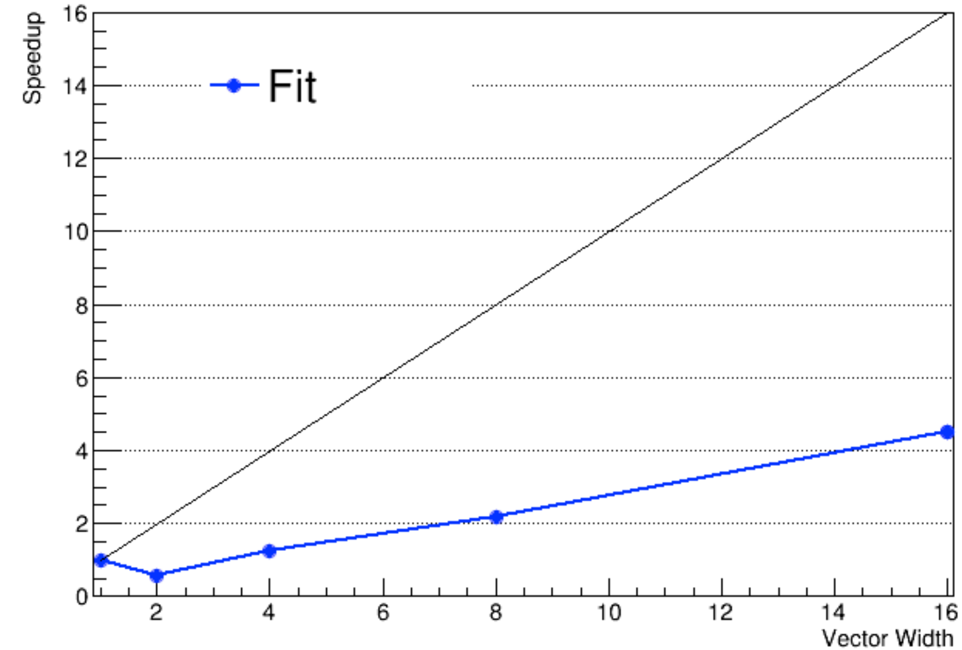
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. 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

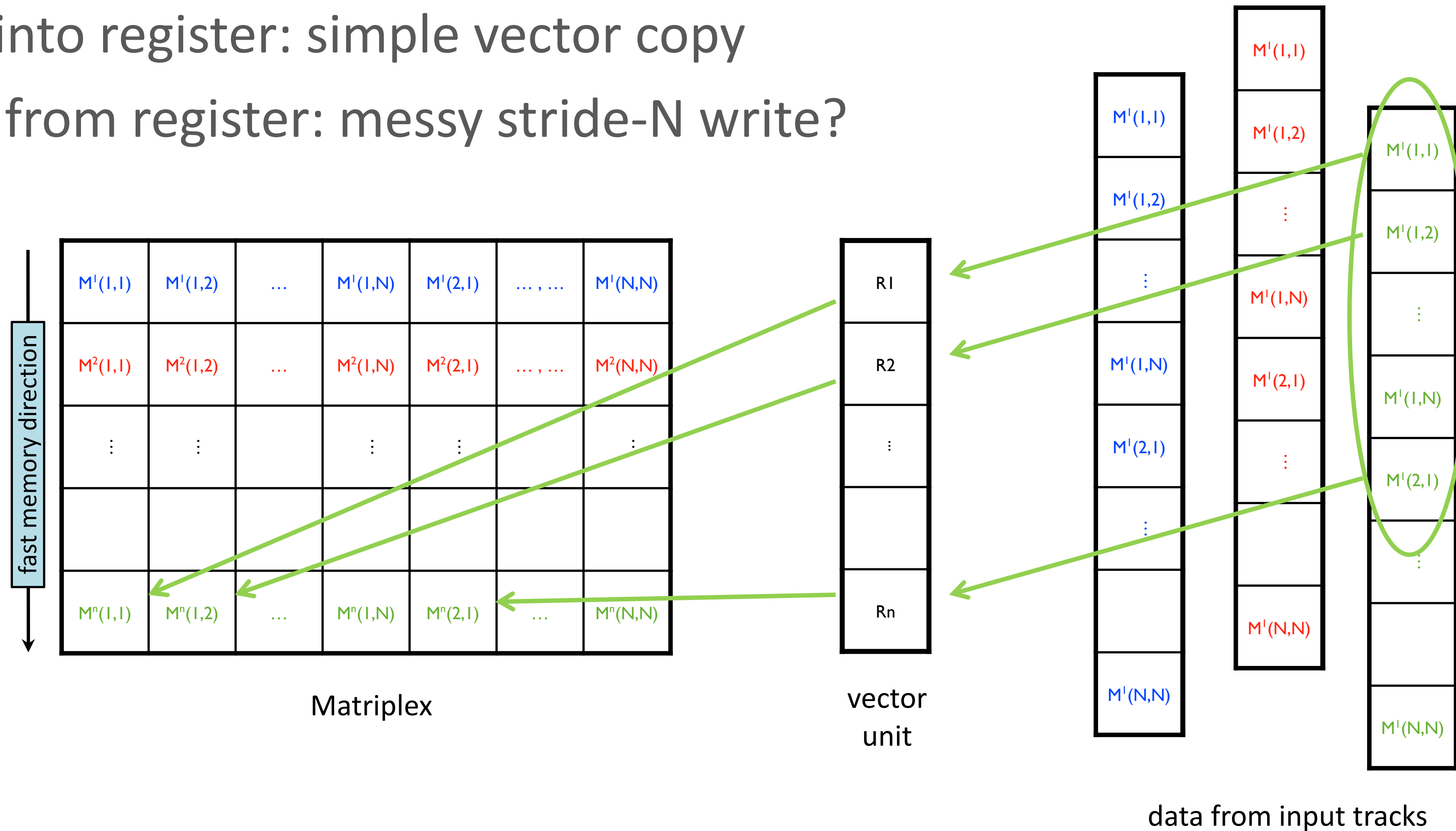
Fixing the False Loop-Carried Dependence

- 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?



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 vectorize? (Answer: no!)

SlurpIn: Faster, One-Pass Initialization of Matriplex

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

fast memory direction

Matriplex

R1
R2
Rn

vector unit

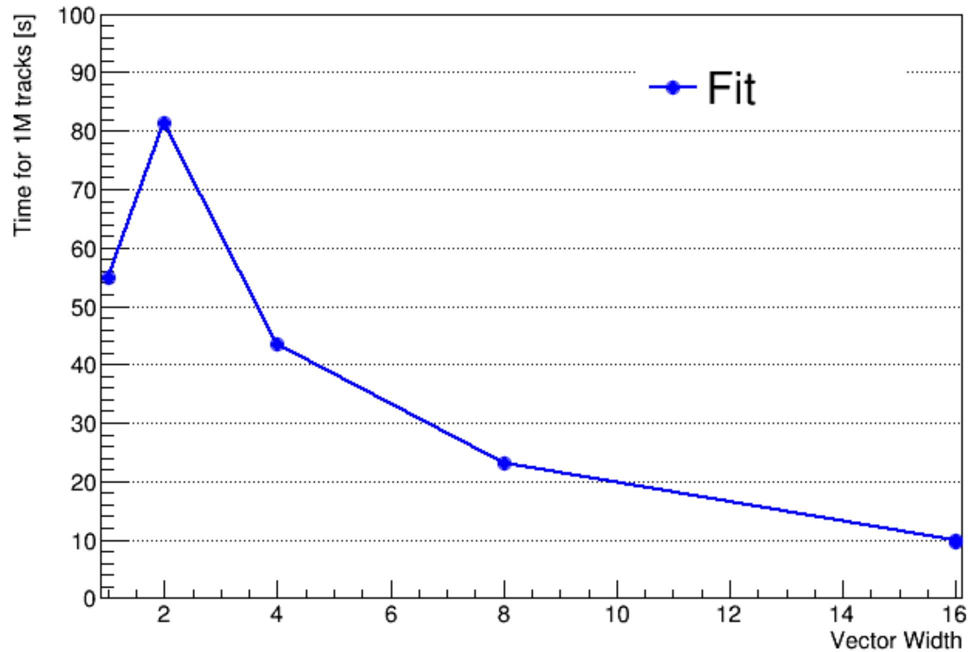
vgather

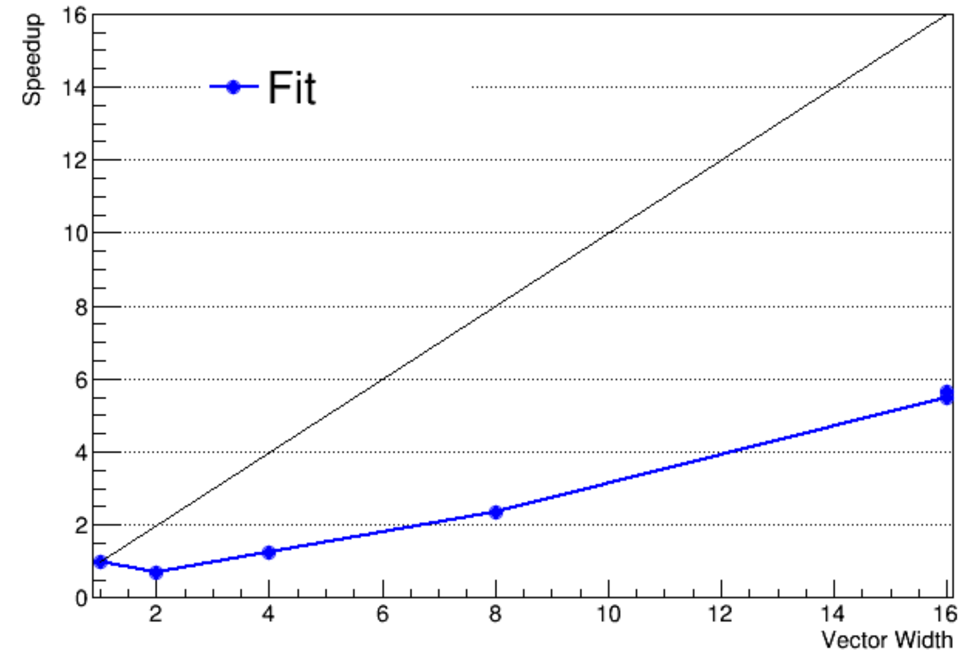
data from input tracks

Getting Data into and out of Matriplexes

- CopyIn
  - Take one data array and distribute it into the Matriplex.
- SlurpIn
  - Build the Matriplex by taking elements (i,j) of all data arrays.
  - AVX-512 includes a special *gather* instruction for input matrices that are addressable from a common address base.
- CopyOut – populate output matrix
  - Jumps over 8 or 16 floats (16 floats is a cache line) – yikes.
  - CopyOut is done infrequently and often only for selected parts.
  - It hasn't shown up on the radar of things to fix yet.
  - CopyIn did and that's why we have SlurpIn ☺

Retest of Track Fitting in a Simplified Detector

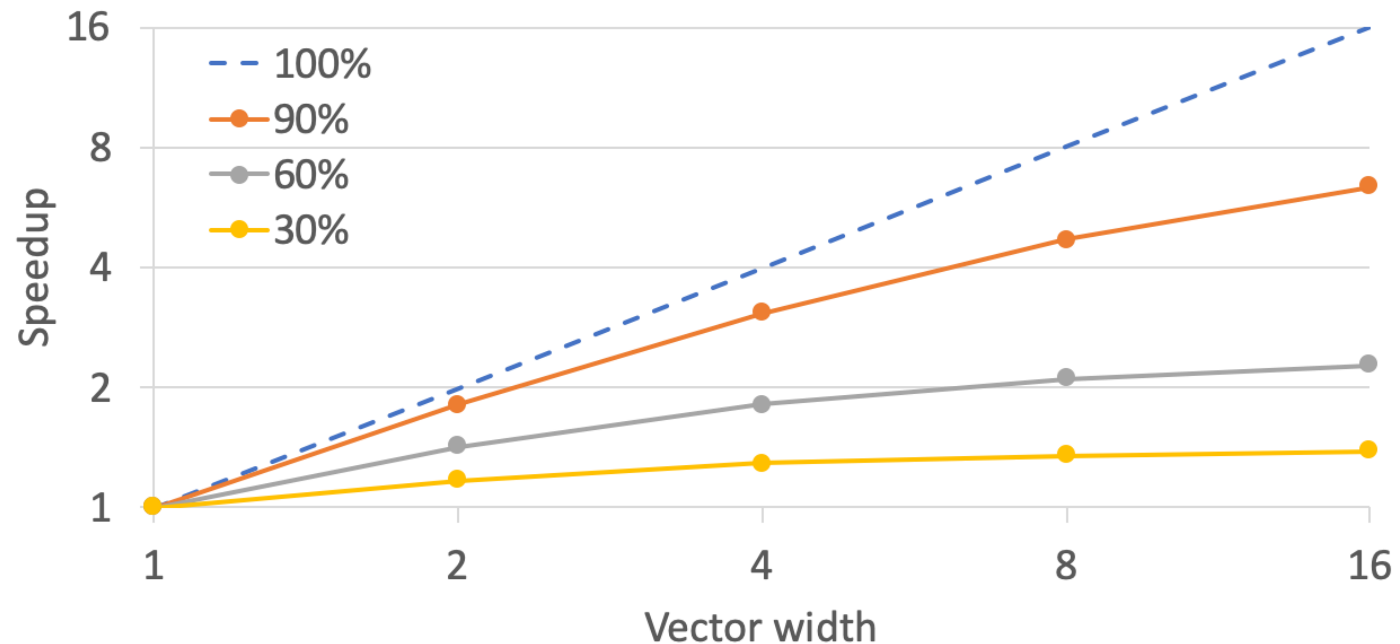




- After fixing Subtract and switching to SlurpIn, test runs 25% faster at full vector width, maximum speedup goes from ~4.4x to ~5.6x
- Amdahl's Law: *can't* get full speedup until *everything* is vectorized

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



Summary: Improving Vectorization

- The compiler “automatically” vectorizes tight loops
- Write code that is vector-friendly
  - Innermost loop accesses arrays with stride one
  - Loop bodies consist of simple multiplications and additions
  - Data in cache are reused; loads are stores are minimized
- Write code that avoids the potential issues
  - No loop-carried dependencies, branching, aliasing, etc.
- This means you know where vectorization should occur
- Optimization reports will tell you if expectations are met
  - See whether the compiler’s failures are legitimate
  - Fix code if the compiler is right; use #pragma if it is not

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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions

Memory Performance and Vectorization

- We have mostly been focusing on faster flop/s, but flop/s don't happen unless data are present
  - Moving data from memory is often the rate-limiting step!
- Data (including scalar data + neighbors) travel between RAM and caches in groups called "cache lines" that are the exact same size as vectors
- But if data movement is "vectorized", like adds and multiplies are vectorized, then everything gets the same speedup, right?
  - No. The data rate for RAM is slow, even if it is always "vectorized" in a sense
  - Well… *loads* from L1 data cache to registers, and *stores* from registers to L1d, are truly vector operations. But that's just the final short step, if the data start way out in RAM

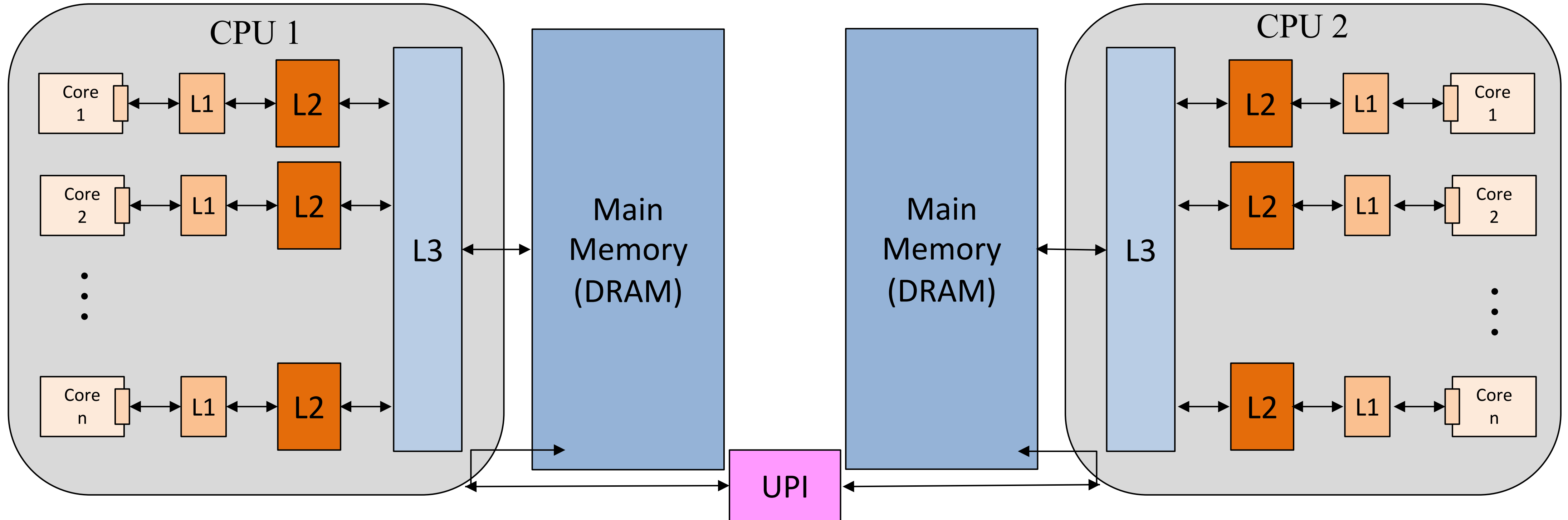
Stride-One Access

- Fastest usage pattern is “stride 1”: perfectly sequential
  - Cache lines arrive in L1d as full, ready-to-load vectors
- Stride-1 constructs:
  - Storing data in structs of arrays vs. arrays of structs
  - Looping through arrays so their “fast” dimension is innermost
    - C/C++: stride 1 on last index (columns)
    - Fortran: stride 1 on first index (rows)

```
for(j=0;j<n;j++) {
   for(i=0;i<n;i++) {
      a[j][i]=b[j][i]*s;
   }
}
```

```
do j=1,n
   do i=1,n
      a(i,j)=b(i,j)*s
   end do
end do
```

Memory Hierarchy



	Registers	L1 Cache	L2 Cache	L3 Cache	DRAM	Disk
Speed	1 cycle	~4 cycles	~10 cycles	~30 cycles	~200 cycles	10ms
Size	< KB per core	~32 KB per core	~256 KB per core	~35 MB per socket	~100 GB per socket	TB

Laptop Vector Utilization as a Function of Array Size

Vector utilization

sum2
sum3
mul2
mul3
div2
div3

Utilization

0.45
0.4
0.35
0.3
0.25
0.15
0.1
0.05
0

1
10
10^2
10^3
10^4
10^5
10^6
10^7
N_{array}

L1 drop off, 32KB

?

L2 drop off, 256KB

Loop overhead

L3 drop off (6MB), too soon?
Output matters, too!

Vector too small

Division – cache does not matter

“mtorture” code by Matevž Tadel, UCSD

HPC Vector Utilization as a Function of Matriplex Array Size

Vector utilization

Utilization

0.7
0.6
0.5
0.4
0.3
0.2

1 10 10^2 10^3 10^4

Matriplex vector size

3_8
3_16
3_32
6_8
6_16
6_32

- Square matrix multiplications
- First number is dimension
- Second is Matriplex width

Sandy Bridge, Xeon, AVX

“mtorture” code by Matevž Tadel, UCSD

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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. Conclusions and future directions

Loops That the Compiler Can Vectorize

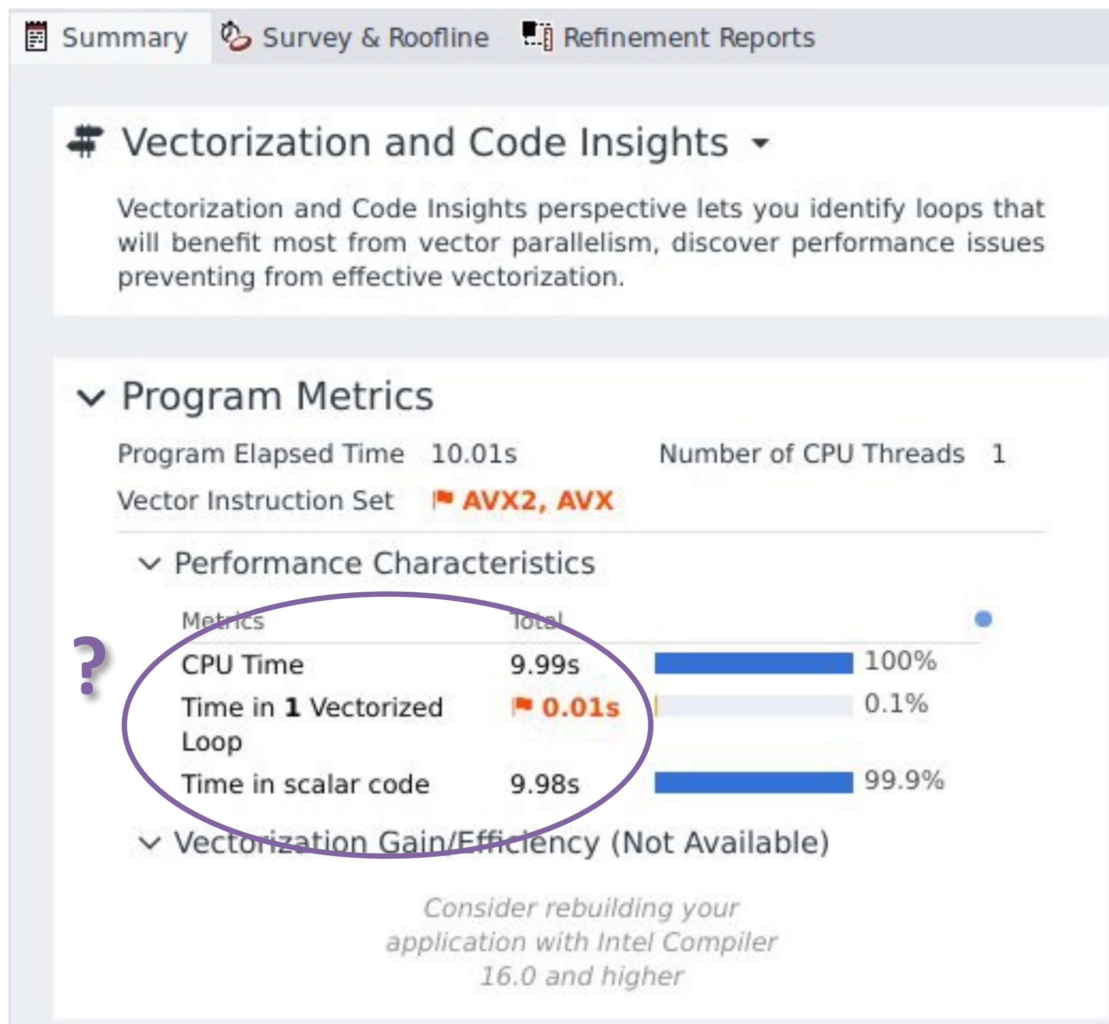
Basic requirements of auto-vectorizable loops:

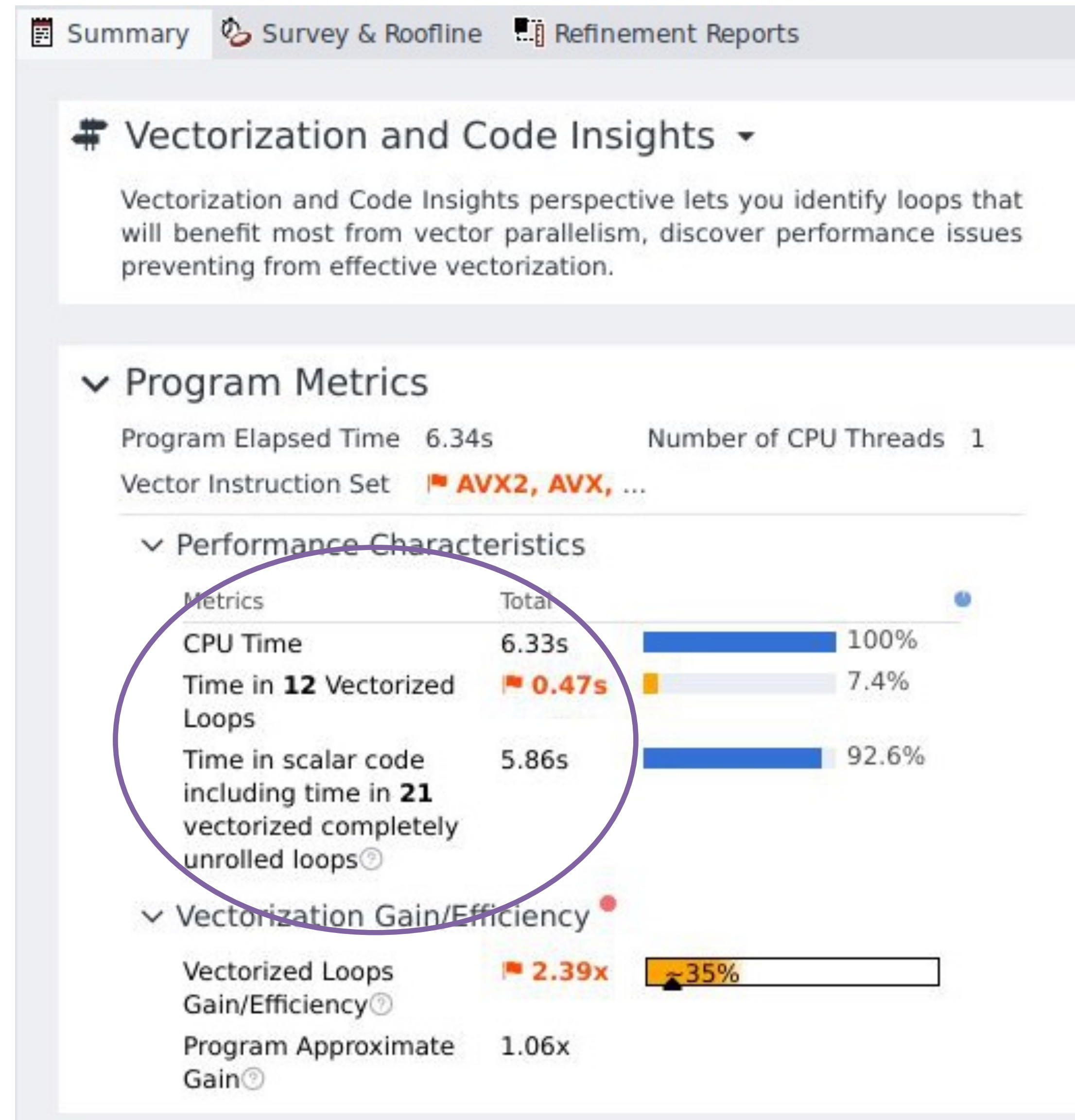
- Number of iterations is known on entry
  - No conditional termination (“break” statements, while-loops)
- Single control flow; no “if” or “switch” statements
  - Note, the compiler may convert “if” to a masked assignment!
- Must be the innermost loop, if nested
  - Note, the compiler may reorder loops as an optimization!
- No function calls but basic math: pow(), sqrt(), sin(), etc.
  - Note, the compiler may inline functions as an optimization!
- All loop iterations must be independent of each other

Compiler Options and Optimization

- GCC vectorizes with `-O2 -ftree-vectorize` or `-O3`
  - Default for x86_64 is SSE (see output from `gcc -v`)
  - To tune vectors to the host machine: `-march=native`
  - To optimize across objects (e.g., to inline): `-flto`
  - For AVX-512, you *must* add `-mprefer-vector-width=512`
- Intel Classic Compilers vectorize with simply `-O2`
  - Default is SSE instructions, 128-bit vector width (4 floats)
  - To tune vectors to the host machine: `-xHost`
  - To optimize across objects (e.g., to inline functions): `-ipo`
  - For AVX-512, you *must* add `-qopt-zmm-usage=high`
  - Says AVX-512 isn't a great default (AMD added it only recently)

Intel Advisor's Vectorization Report: gcc vs. icc





work with Patrick Gartung, Fermilab

Recent 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 did not come with libmvec, an equivalent vector math library, 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
  - 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

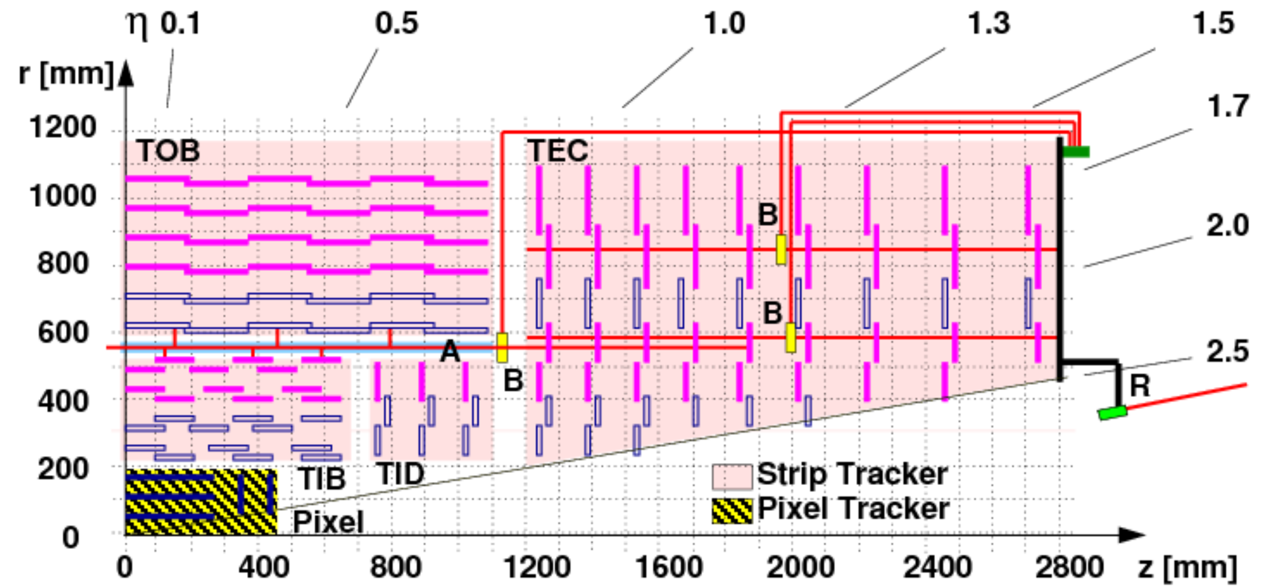
1. Introduction to particle colliders and the tracking problem
2. Reconstructing particle tracks with a Kalman Filter algorithm
3. Vectorization of the basic Kalman Filter operations
4. Tuning Matriplex methods to improve vectorization
5. Checking the cache performance of Matriplex
6. Using compilers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

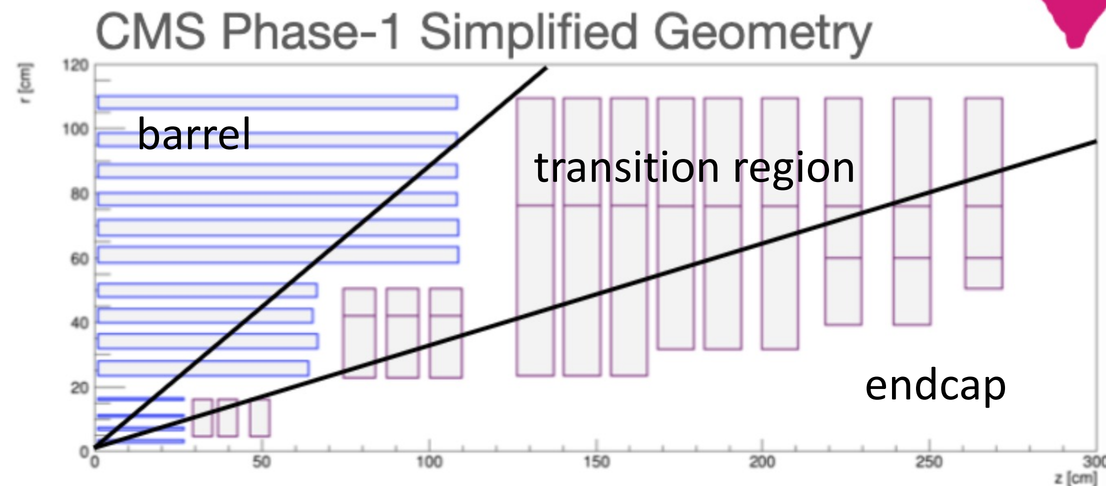
Strategy for Track Building with “mkFit”

- 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

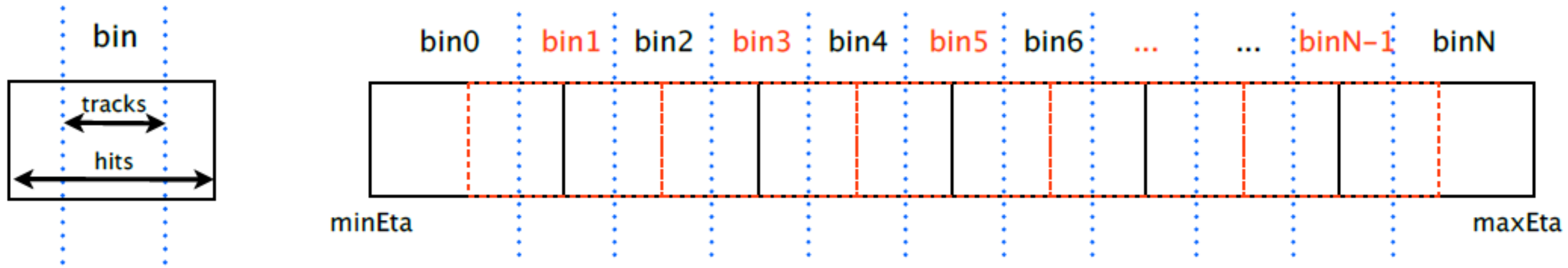
Simplifying the Geometry

- 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)

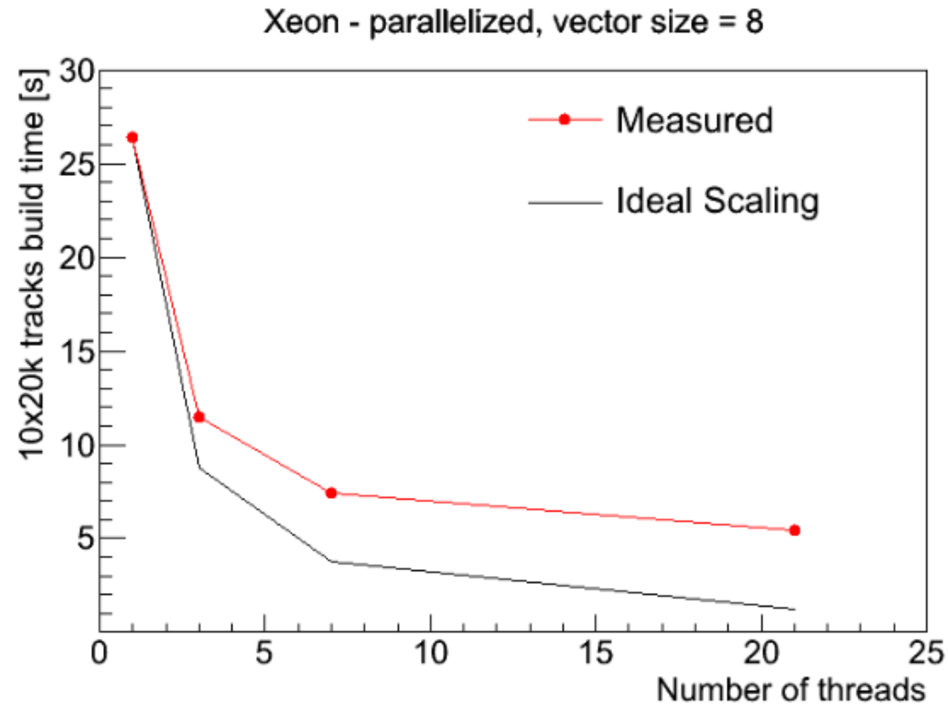
Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

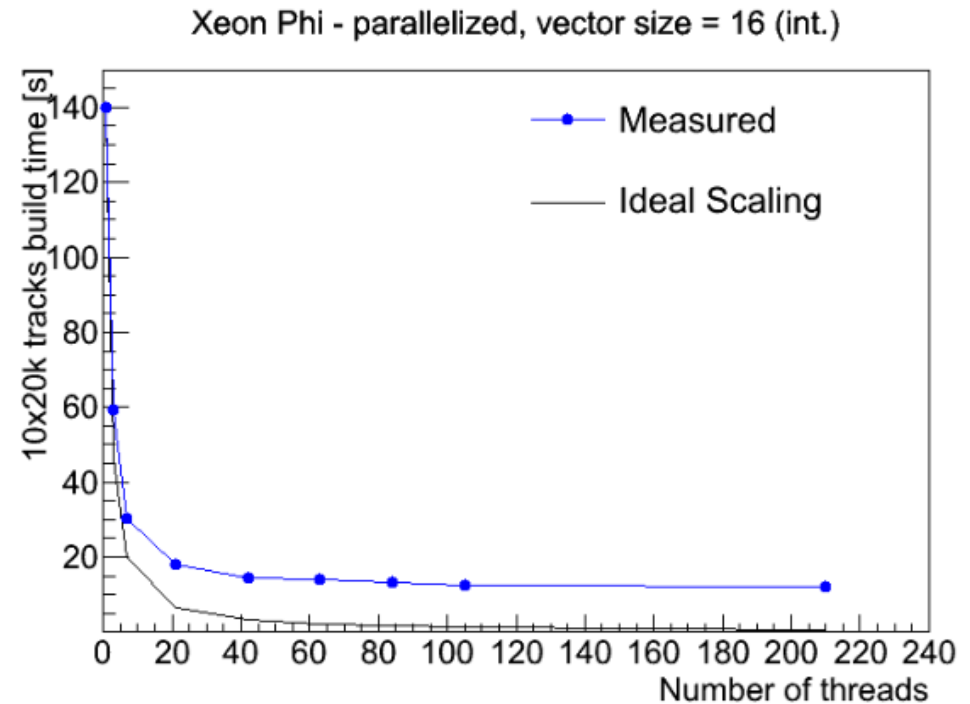
Grouping: Function / Call Stack

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>>>, l	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 assigning threads to 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 Again

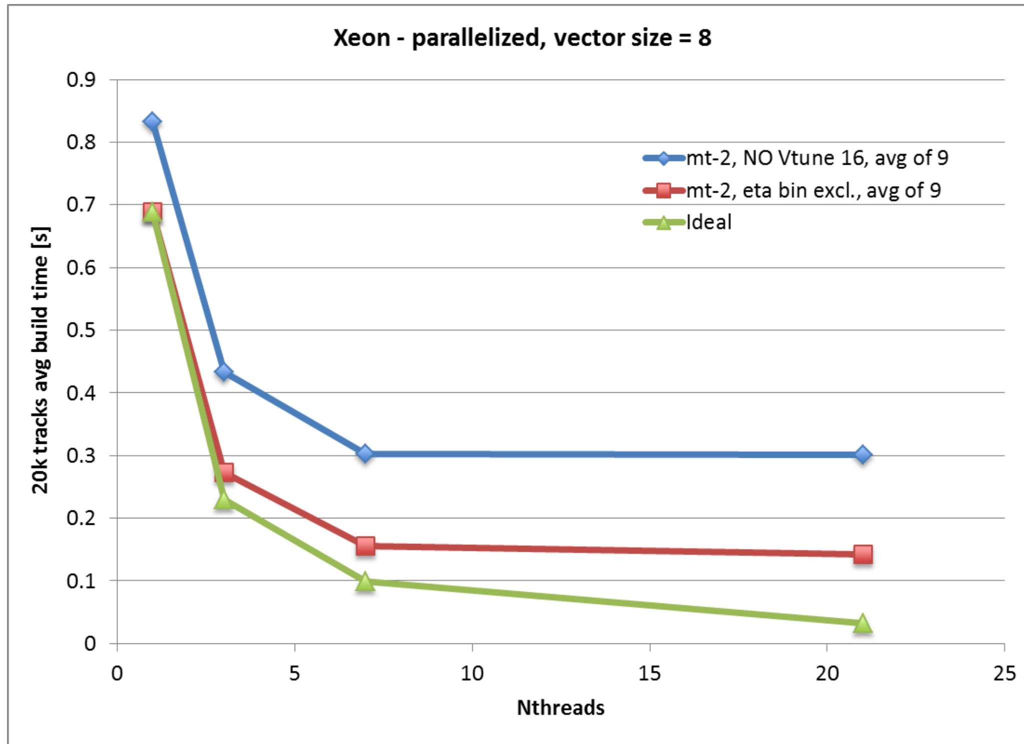
- Possible explanation: some fraction B of work is a serial bottleneck
- If so, the minimum time for n threads is set by Amdahl's Law:

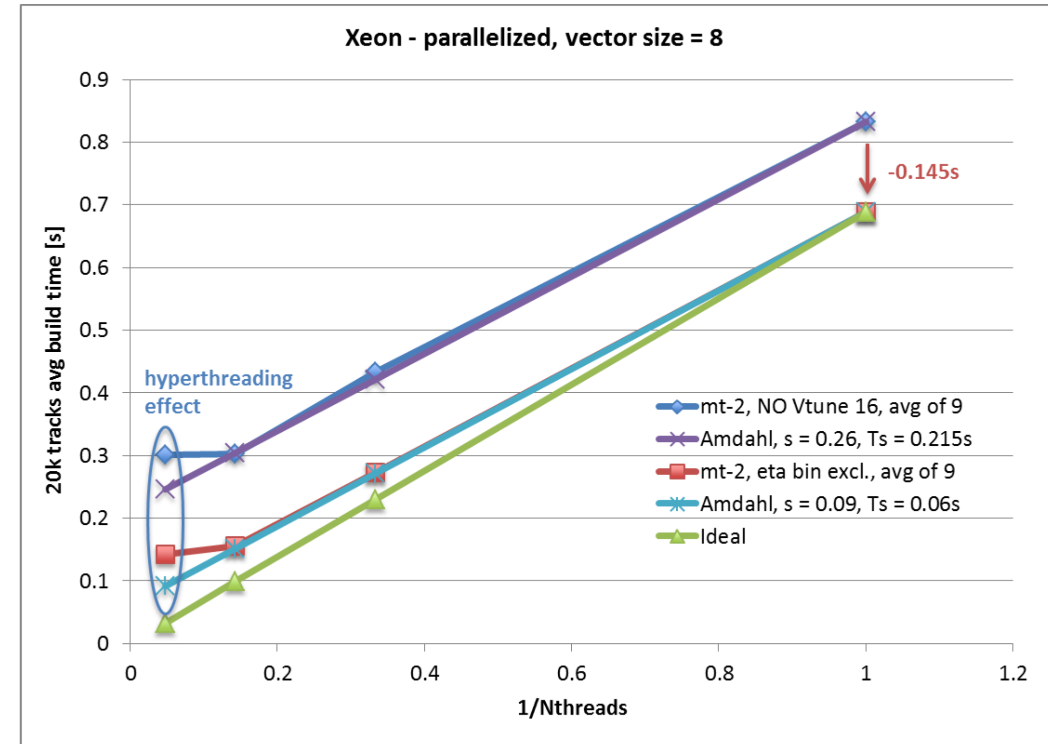
$$T(n) = T(1)\ [(1-B)/n + B]$$

parallelizable… *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!





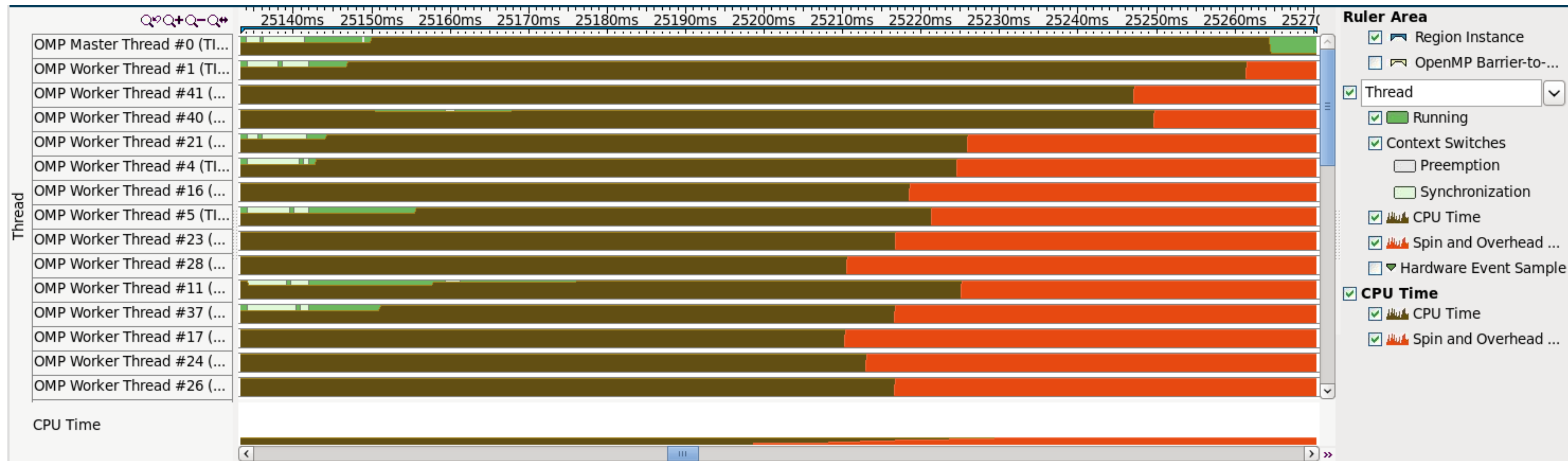
- Huge improvement from excluding *one code line* creating eta bins

```
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

Intel VTune Shows Another Issue

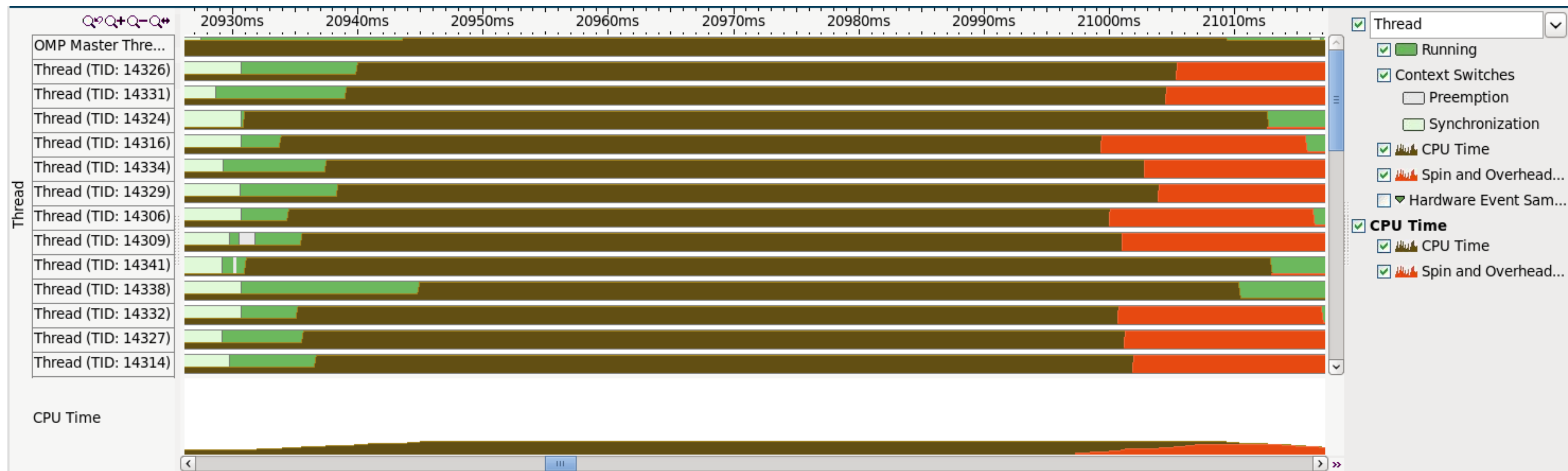
- VTune reveals non-uniformity of occupancy within OpenMP threads
  - Some threads take far longer than others: *load imbalance*
  - Worsens as threads increase: test below uses 42 threads on Xeon Phi



- 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

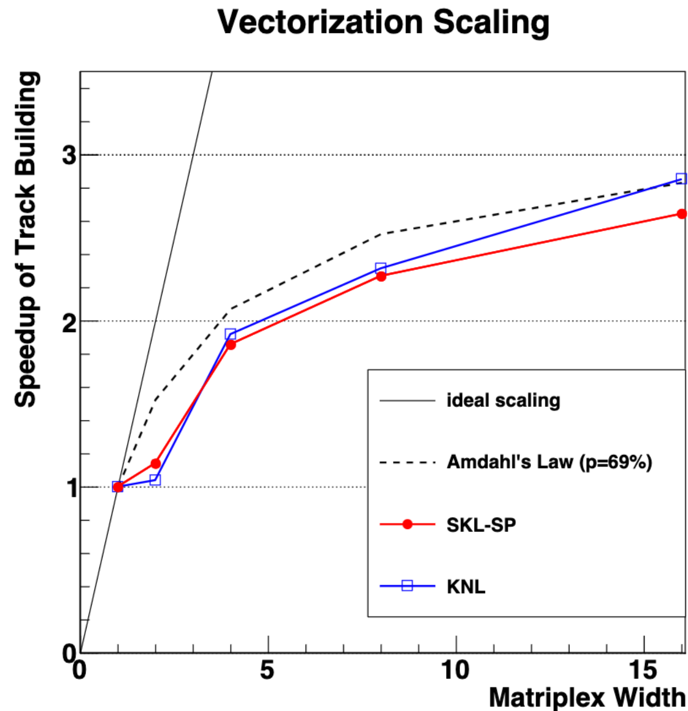
- Nested levels of parallel tasks for track building:
  1. Loop over different events;
  2. Loop over different η-regions;
  3. Loop over z-/r- and φ-sorted groups of seeds.
- Parallel tasks scheduled through Intel TBB
  - Dynamic task stealing to balance workloads
- Basic parallel task includes simplified 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

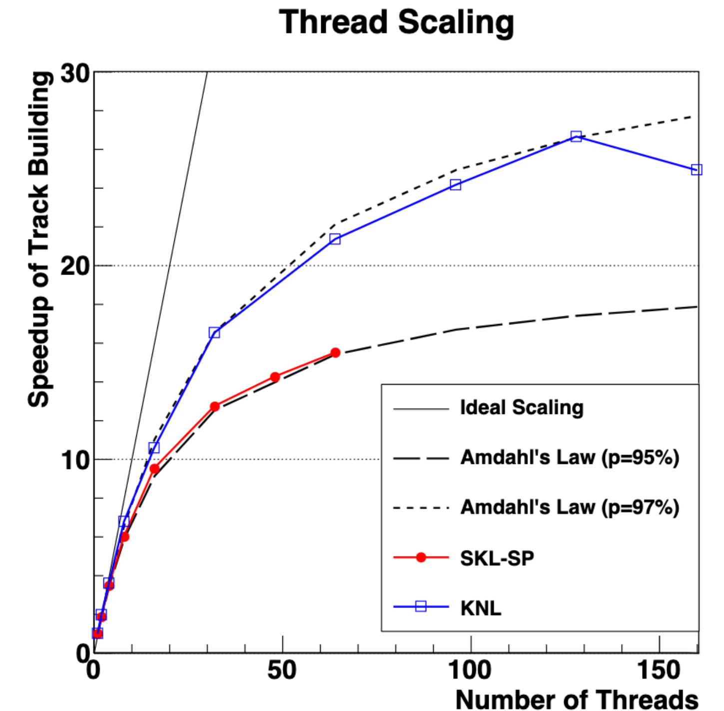
Outline

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. Checking the cache performance of Matriplex
6. Using compilers to auto-vectorize track propagation
7. The multithreaded framework for building tracks
8. 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

Conclusions: Vectorized Track Finding

- Vectorization can give a significant boost depending on:
  - The ability to express the problem in vector form
  - The problem size and problem complexity
  - The ability to move data through cache hierarchy to VPUs
- Understanding performance at the simplest level is vital
- Memory issues can blur vectorization effects:
  - Multithreading – cache sharing, locking
  - Swapping algorithms or data blocks – cache thrashing
- Use of code-line level profiling is crucial!
  - Don't be afraid of the assembler view, it is often revealing
- Experiment – being frustrated is part of the game ☺

Conclusions: HPC in the Era of Many Cores

- HPC has moved beyond clusters that rely on coarse-grained, MPI parallelism
  - *Coarse-grained*: big tasks are parceled out to a cluster
  - *MPI*: tasks pass messages to each other over a local network
- HPC now involves fine-grained parallelism and SIMD within shared memory
  - *Fine-grained*: threads run subtasks on numerous cores within a processor
  - *SIMD*: subtasks also act upon multiple sets of operands simultaneously
  - It is now the norm to have many of these SIMD engines 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

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