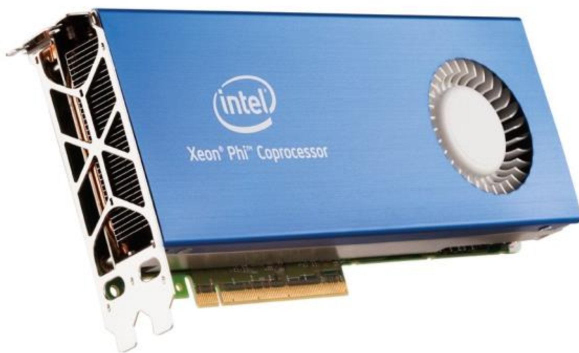
Cornell University
Center for Advanced Computing

High Performance Computing in the Manycore Era: Challenges for Applications

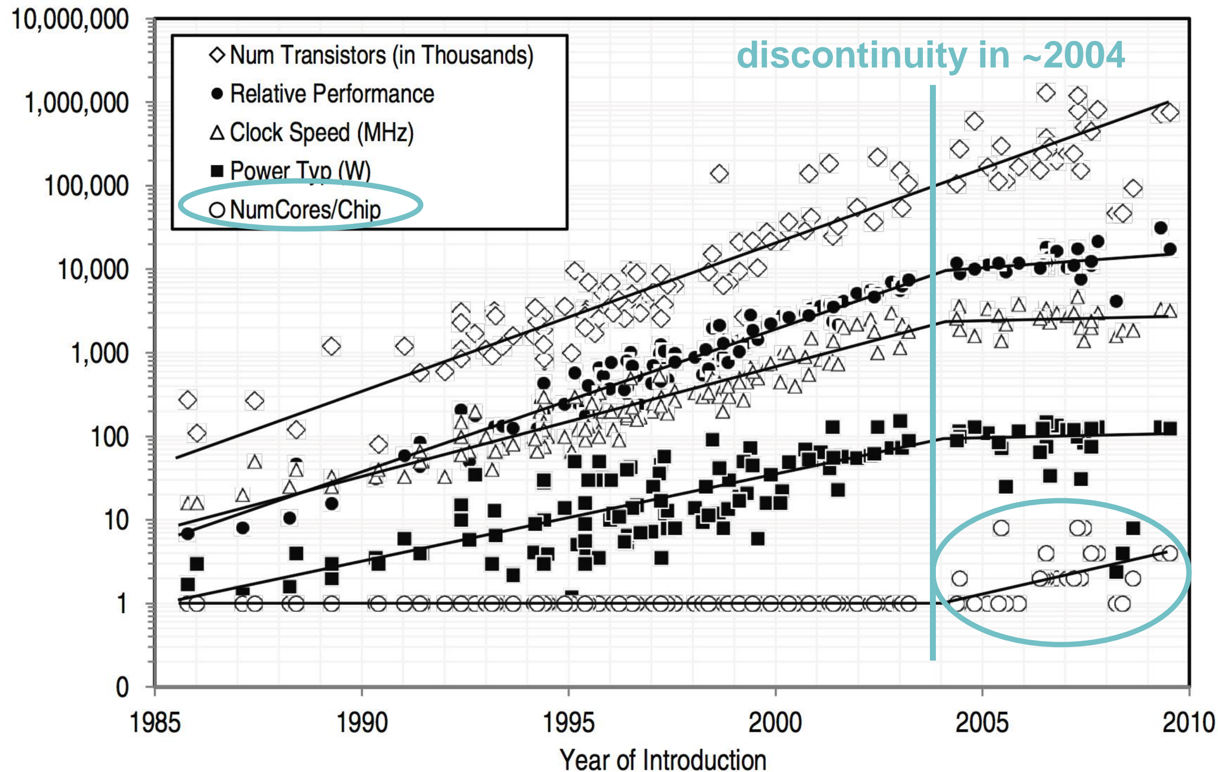
Steve Lantz
Senior Research Associate
Cornell University Center for Advanced Computing (CAC)
slantz@cac.cornell.edu

CACM Seminar at RIT, Nov. 17, 2015

Manycore, not Manticore...



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

Photo by TACC, June 2012

- 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 must move from multi-core to ***manycore***

Manycore Elements in Petaflop/s Machines

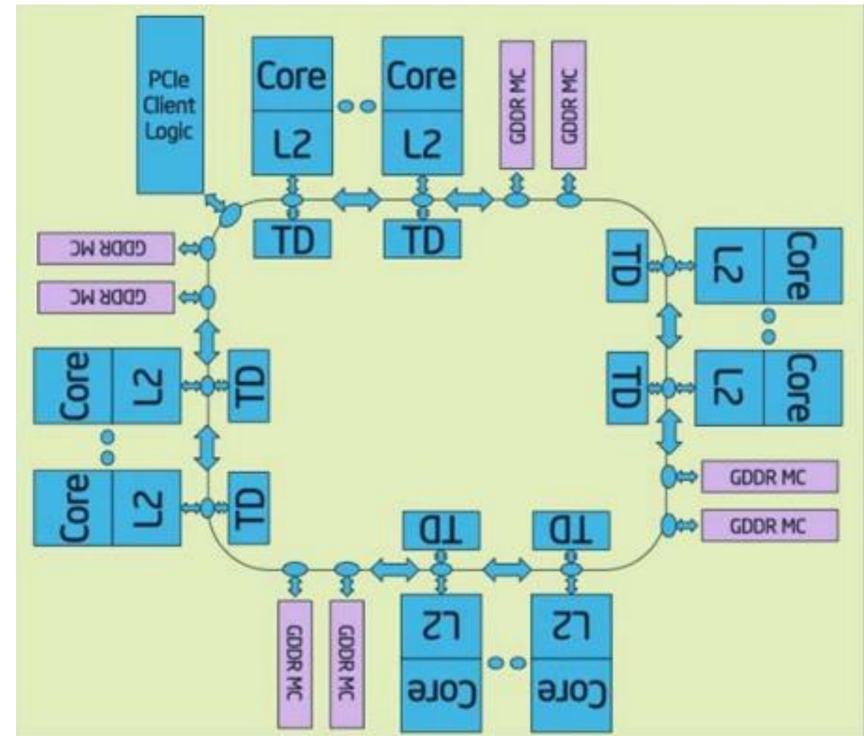
- **CPUs**: Wider vector units, more cores
  - *General-purpose* platform
  - *Single-thread* performance emphasis
  - Example, dual E5-2680 on Stampede: **0.34 Tflop/s, 260W**
- **GPUs**: Thousands of very simple stream processors
  - *Special multithreading APIs*: CUDA, OpenCL, OpenACC
  - *High floating-point throughput*
  - Example, Tesla K20 on Stampede: **1.17 Tflop/s, 225W**
- **MICs**: Dozens of CPU cores optimized for floating-point efficiency
  - *General-purpose* codes will run (like CPU)
  - *High floating-point throughput* for multithreaded code (like GPU)
  - Example, Xeon Phi SE10P on Stampede: **1.06 Tflops/s, 300W**

Xeon Phi: What Is It?

- Complete system on PCIe card (Linux OS, processor, memory)
- x86-derived processor featuring large number of simplified cores
  - Many Integrated Core (MIC) architecture
- Optimized for floating point throughput
  - Lots of floating-point operations per second (flop/s) for HPC
- Modified 64-bit x86 instruction set
  - Code compatible (C, C++, Fortran) after re-compile
  - Not binary compatible with x86_64
- Intel's answer to general purpose GPU (GPGPU) computing
  - Similar flop/s/watt to GPU-based products like NVIDIA Tesla

Power-Saving Choices in the Xeon Phi Design

- *Reduce* clock speed
- *Omit* power-hungry features such as branch prediction, out-of-order execution
- *Simplify* instruction decoder, but maintain high instruction rate via 2–4 threads per core
- *Eliminate* a shared L3 cache in favor of coherent L2 caches
- *And add...* lots of cores!
- These factors tend to degrade single-thread performance, so multithreading is essential



MIC vs. CPU

	MIC (SE10P)	CPU (E5)	MIC 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

- CPUs designed for all workloads, high single-thread performance
- MIC also general purpose, though optimized for number crunching
  - Focus on high aggregate throughput via lots of weaker threads
  - Possible to achieve >2x performance compared to dual E5 CPUs

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 fully utilize MIC, 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 MIC and CPU

More parallel

Scalar & single-thread

Vector & single-thread

Scalar & multithreaded

Vector & multithreaded

Xeon Phi

Xeon

1 10 100 1000 10,000

More performance

Courtesy James Reinders, Intel

Typical Configuration of a Stampede Node

- Host with dual Intel Xeon "Sandy Bridge" (CPU)
- PCIe card with Intel Xeon Phi™ (MIC)

Access from network:
ssh <host> (OS)
ssh <coprocessor> (μOS)

Linux OS

HCA

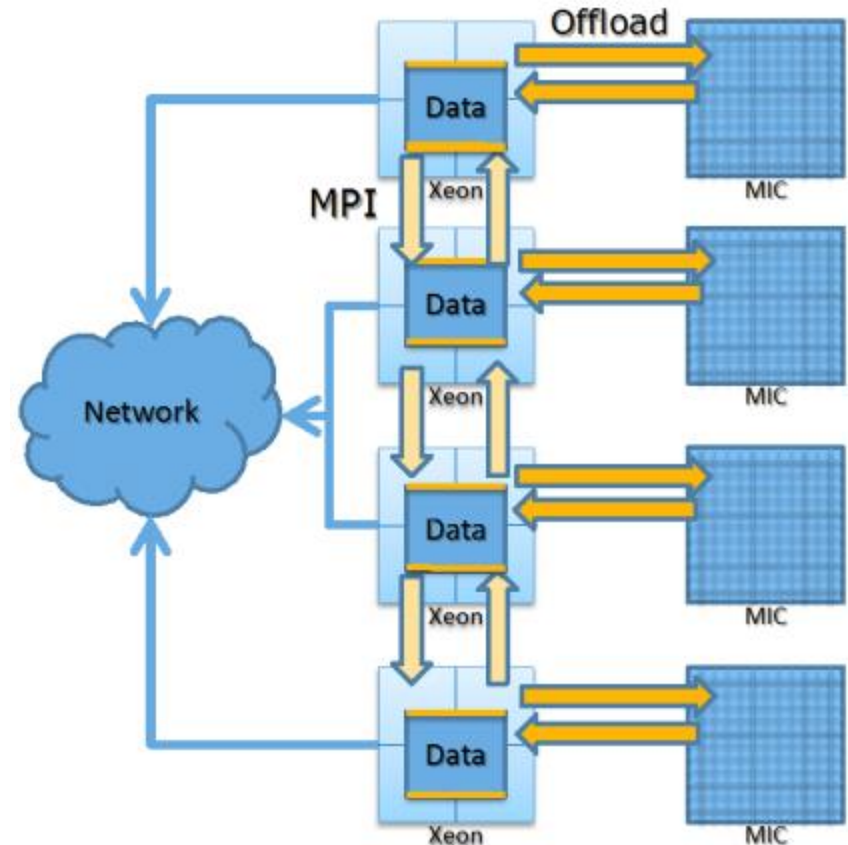
PCIe

Virtual IP* service for MIC

Linux micro OS

Offload Execution Model

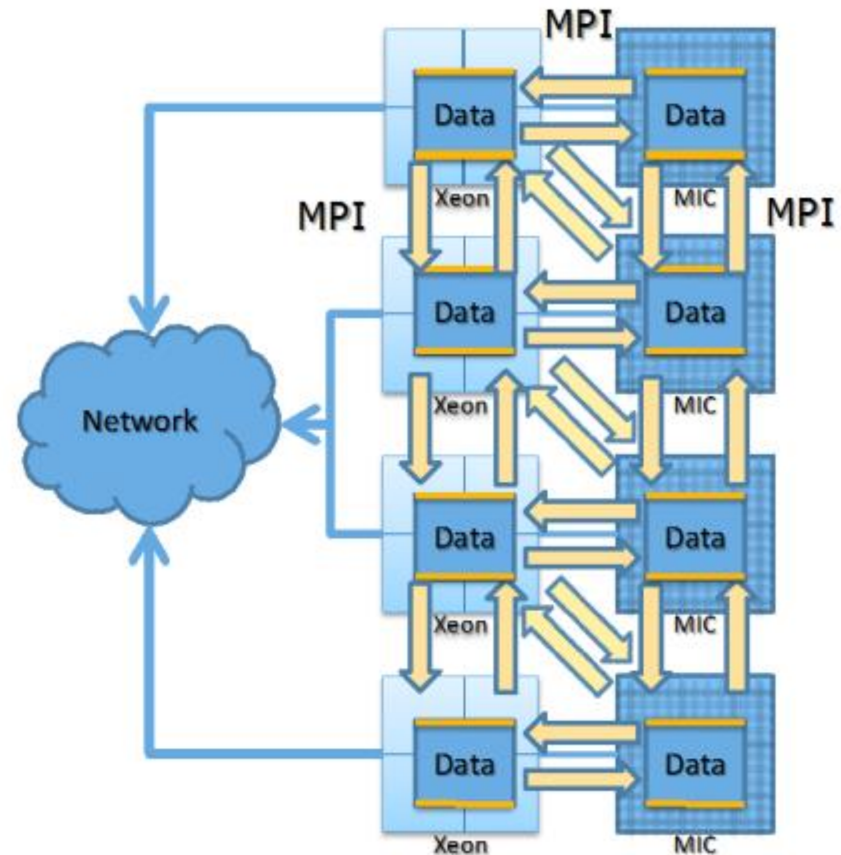
- OpenMP-like directives indicate which data and functions to send from CPU to MIC for execution
- Unified source code
- Code modifications required
- Compile once
- Run in parallel using MPI (Message Passing Interface) and/or scripting, if desired



Courtesy Scott McMillan, Intel

"Symmetric" Execution Model

- Message passing (MPI) on CPUs and MICs alike
- Unified source code
- Code modifications advisable
  - Multithread with OpenMP or Threaded Building Blocks
  - Assign different work to CPUs vs. MICs
- Compile twice, 2 executables
  - One native to host
  - One native to MIC
- Run in parallel using MPI



Courtesy Scott McMillan, Intel

Application: High Energy Physics

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

Photo
CMS detector,
LHC, CERN

Cornell University
Center for Advanced Computing

LHC: It's a 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

Cornell University
Center for Advanced Computing

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

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

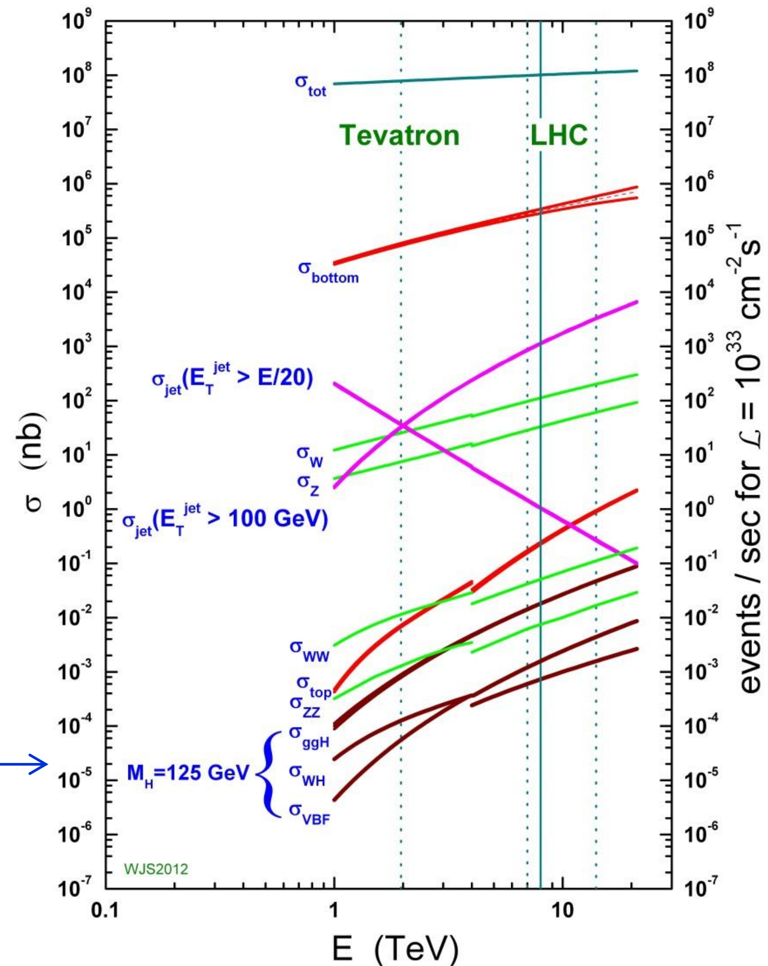
Transverse slice through CMS

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

Big Data Challenge

- 40 million collisions a second
- Most are boring
  - Dropped within 3 μs
- 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

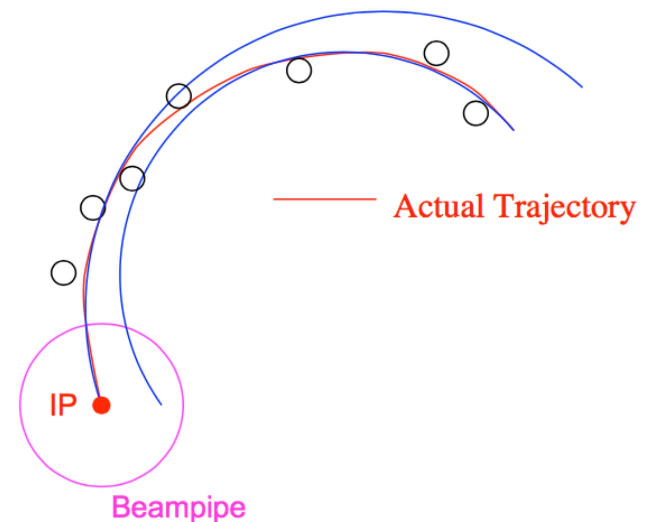


http://www.hep.ph.ic.ac.uk/~wstirlin/plots/plots.html

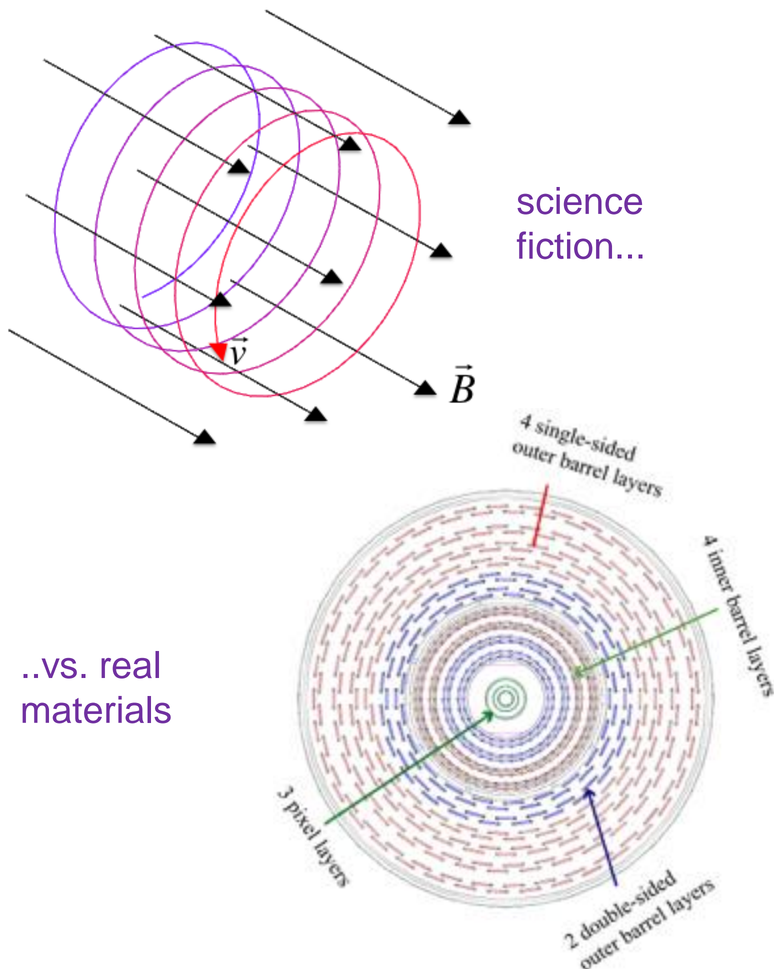
For Interesting Events: Tracking

- 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***



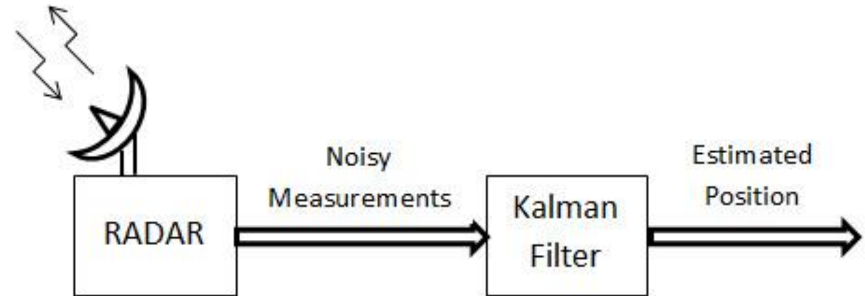
Why Kalman Filter for Particle Tracking?



- 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

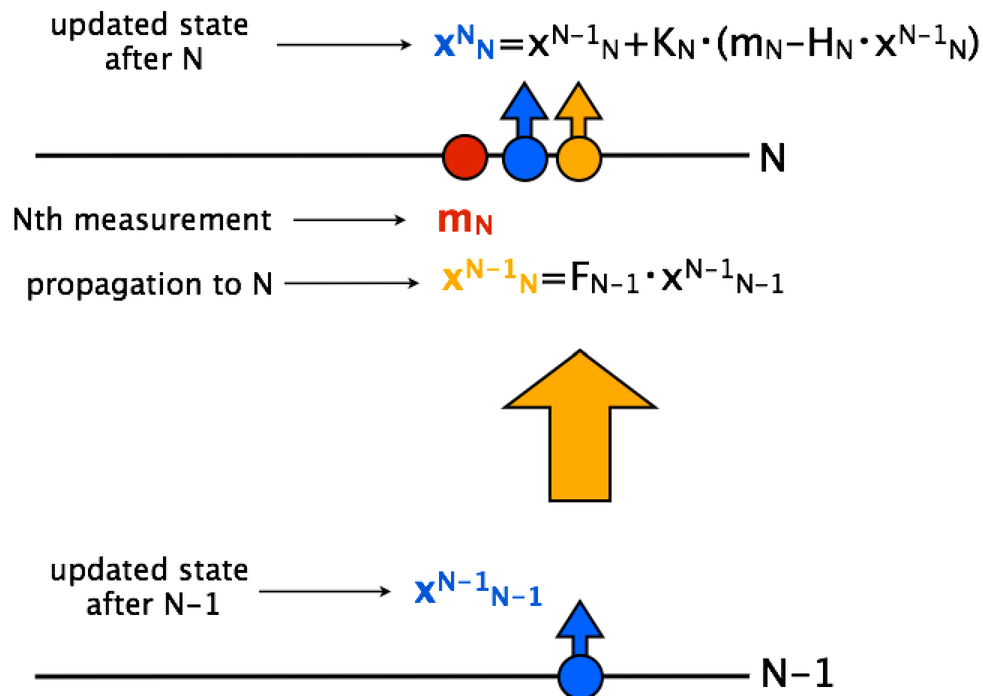
From Wikipedia, the free encyclopedia

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

Tracking as Kalman Filter

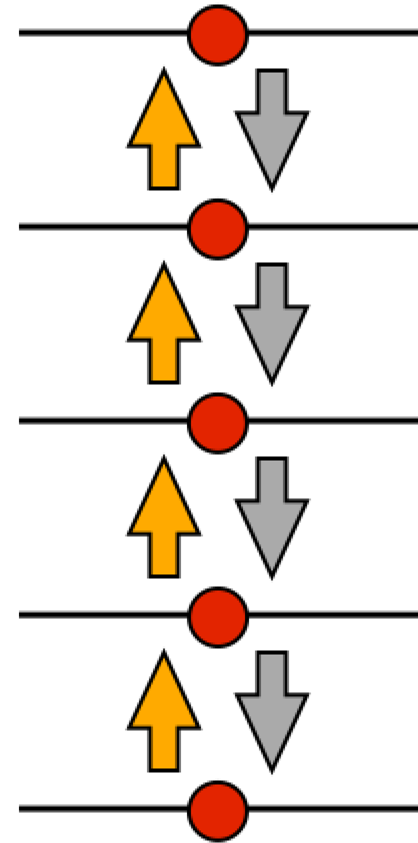
- Track reconstruction has 3 main steps: *seeding*, *building*, and *fitting*
- Building and fitting repeat the basic logic unit of the Kalman Filter...



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

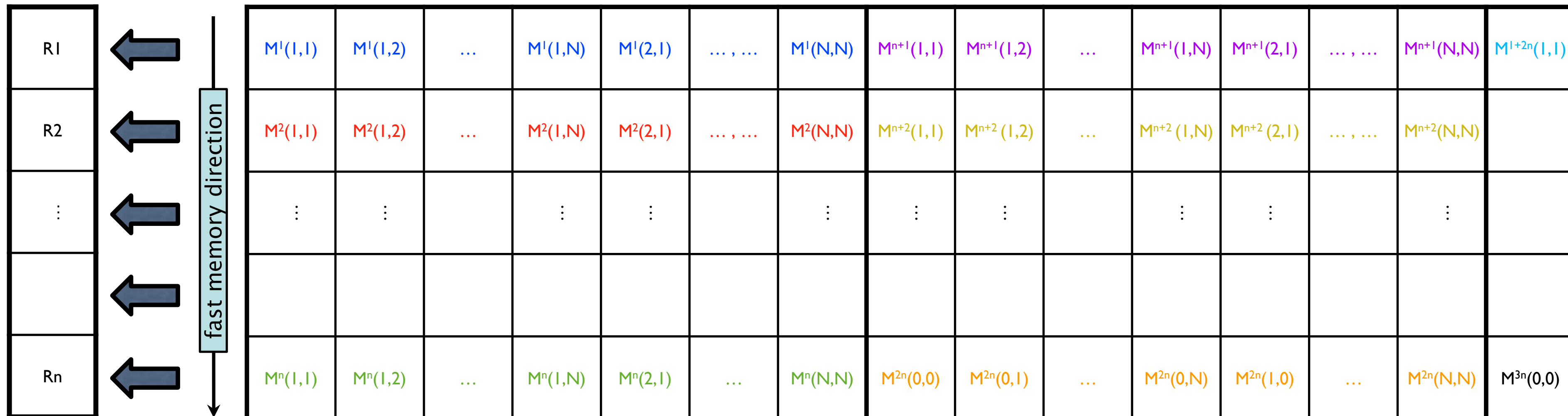
Track Fitting as Kalman Filter

- 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!



vector unit

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

Initialization of Matriplex from Track Data

- This must **vectorize** to perform well!
- It must also minimize cache misses

fast memory direction

$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	… , …	$M^1(N,N)$
$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	… , …	$M^2(N,N)$
⋮	⋮		⋮	⋮		⋮
$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$

Matriplex

R1
R2
⋮
Rn

vector unit

$M^1(1,1)$
$M^1(1,2)$
⋮
$M^1(1,N)$
$M^1(2,1)$
⋮
$M^1(N,N)$

separate input tracks

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)

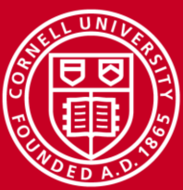
```
void CopyIn(idx_t n, T *arr)
{
   for (idx_t i = n; i < kTotSize; i += N)
   {
      fArray[i] = *(arr++);
   }
}
```

Intel VTune Analysis of L1 Cache Misses

Function / Call Stack	CPU Time	Clockticks	Instructions Retired	CPI Rate	L1 Misses	L1 Hit Ratio	Estimated Latency Impact	L2_DATA_READ_MISS_MEM_FILL
Matriplex::MatriplexSym<float, (int)3, (int)16>::**CopyIn** **[1]**	0.888531	1.1E+09	9.5E+08	1.1579	**27750000**	0.864634	**39.5002**	500000
Matriplex::MatriplexSym<float, (int)6, (int)16>::**CopyIn** **[2]**	0.565429	7E+08	1E+08	**7.00001**	**3750000**	0.75	0	2000000
MkFitter::InputTracksAndHits	0.161551	2E+08	1E+08	2	0	1	0	0
Matriplex::Matriplex<float, (int)3, (int)1, (int)16>::CopyIn	0.379645	4.7E+08	3.5E+08	1.34286	0	1	0	1000000
MultHelixProp	0.484653	6E+08	2E+08	3	0	1	0	0
Matriplex::MatriplexSym<float, (int)6, (int)16>::Subtract	0.145396	1.8E+08	50000000	3.60001	0	1	0	0

[1] equivalent to MPlexLS; called from MkFitter::InputTracksAndHits; likely to be the initialization of Err
[2] equivalent to MPlexHS; called from MkFitter::InputTracksAndHits; likely to be the initialization of msErr

*All analysis is restricted to the final part of the run; only the fitting performance is relevant, so the simulation is skipped



A Faster, Two-Step Initialization of Matriplex

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

fast memory direction

$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	… , …	$M^1(N,N)$
$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	… , …	$M^2(N,N)$
⋮	⋮		⋮	⋮		⋮
$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$

Matriplex

R1
R2
⋮
Rn

vector unit

$M^1(1,1)$	$M^1(1,1)$	…	$M^1(1,1)$
$M^1(1,2)$	$M^1(1,2)$	…	$M^1(1,2)$
⋮	⋮		⋮
$M^1(1,N)$	$M^1(1,N)$	…	$M^1(1,N)$
$M^1(2,1)$	$M^1(2,1)$	…	$M^1(2,1)$
⋮	⋮		⋮
$M^1(N,N)$	$M^1(N,N)$	…	$M^1(N,N)$

packed temp array,
contiguous memory

Full Vectorization Is Crucial to Performance…

Comparison of input methods for fitting 1M tracks using Matriplex

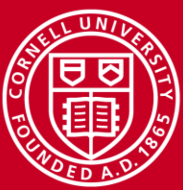
original
2-step
MKL
BEST: 2-step + vgather
2-step + vscatter
original + vscatter

input
fit

Singlle-threaded time, sec.
8
7
6
5
4
3
2
1
0

1
2
3
4
5
6

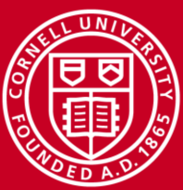
Data input method #



Cornell University
Center for Advanced Computing

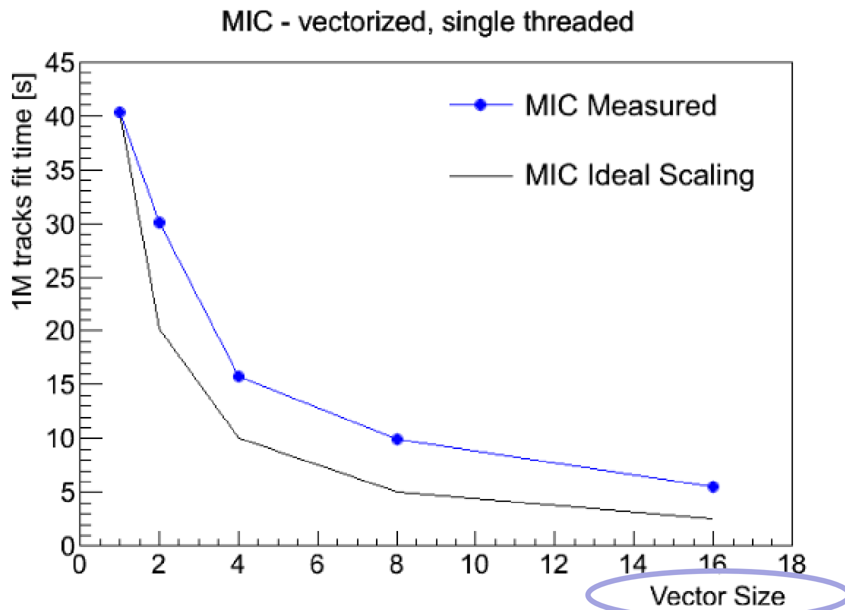
Vectorization of CopyIn: Summary

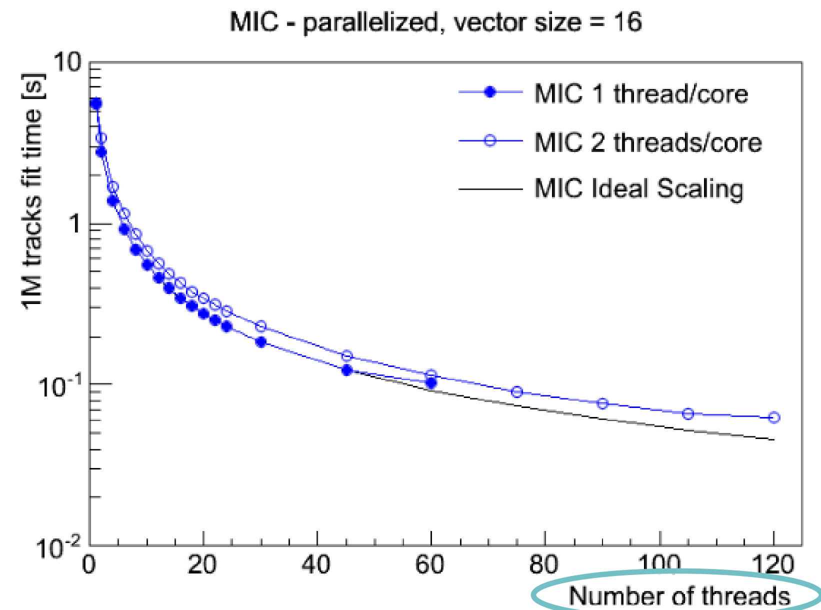
- Intel VTune's metrics revealed that CopyIn, which distributes input data into a Matriplex, was underperforming
  - Assembler code showed lack of vectorization on Xeon Phi
  - Compiler was not converting *strided* for-loops into *vectorized* stores
- Underlying operation is equivalent to a matrix transpose
  - Intel MKL didn't work well; it's best at doing large-matrix operations
- Fastest Xeon Phi code uses Intel's _mm512 vector intrinsics
  - To do a matrix transpose, either a load or a store must be *strided*
  - Intel provides *intrinsics* (low-level function calls) that can do this
  - Strided loads (vgather) work better than strided stores (vscatter)
  - Best: copy all data into a packed temp array, **vgather** into Matriplex
- Big gain from recoding *one* routine for best SIMD performance



Cornell University
Center for Advanced Computing

Good Utilization of MIC for Track Fitting!

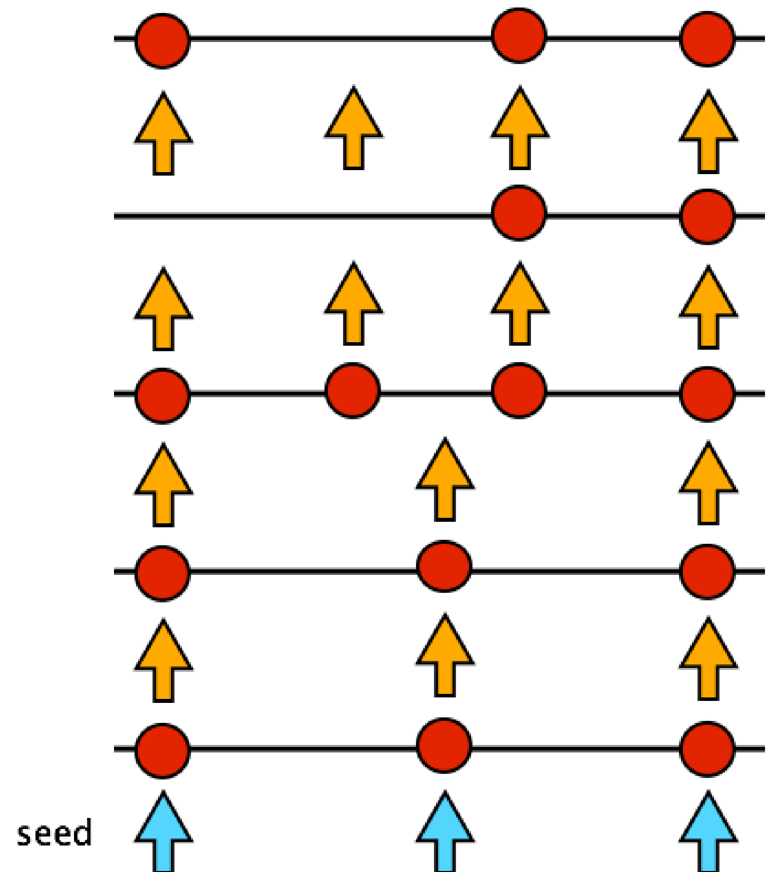




- Fitting is vectorized with Matriplex and parallelized using OpenMP
- Same simulated physics results as production code, but faster
  - Effective performance of vectorization is about 50% utilization
  - Parallelization performance is close to ideal in case of 1 thread/core

Track Building

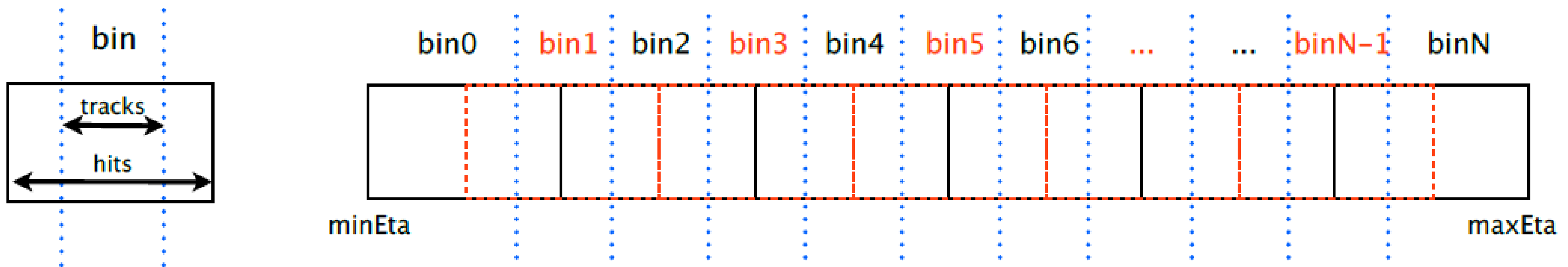
- Building is harder than fitting
- After propagating a track candidate to the next layer, hits are searched for within a compatibility window
- Track candidate needs to ***branch*** in case of multiple matches
  - The algorithm needs to be robust against missing/outlier hits
- Due to branching, track 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)

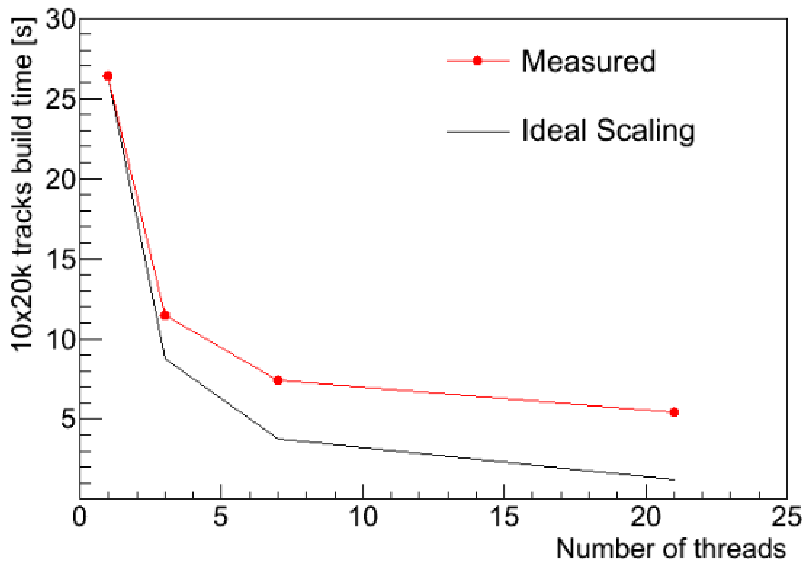
Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

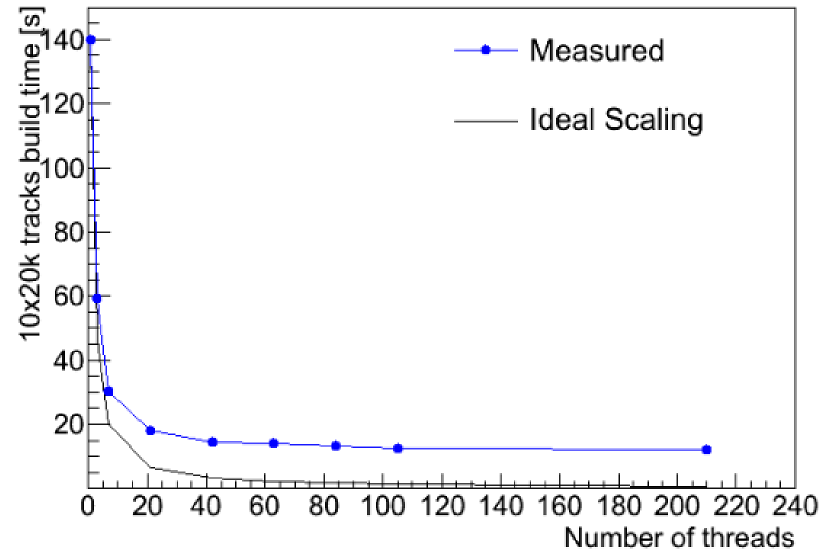
Grouping: Function / Call Stack

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

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

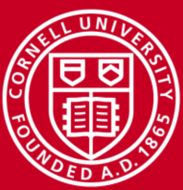
Amdahl's Law

- 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

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

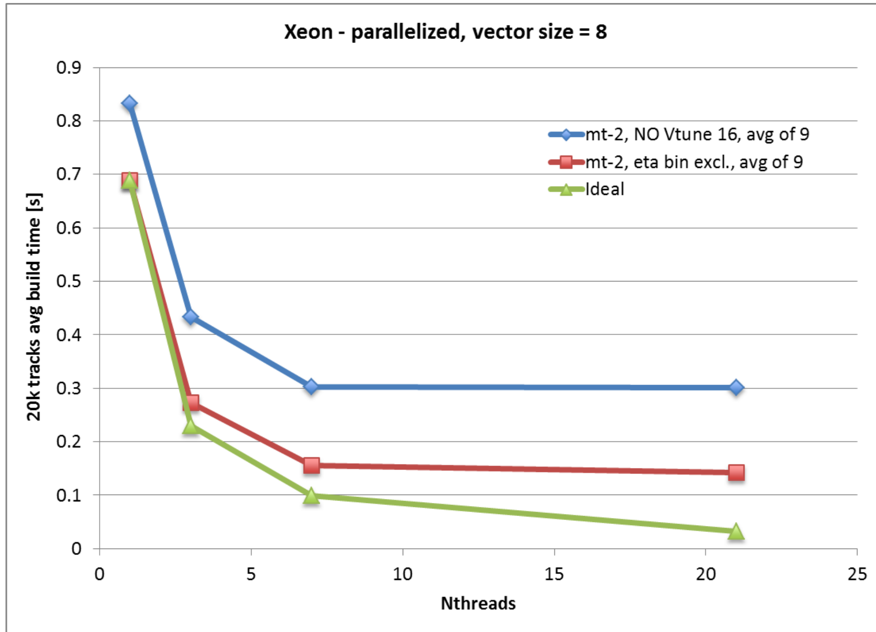
parallelizable… *not!*

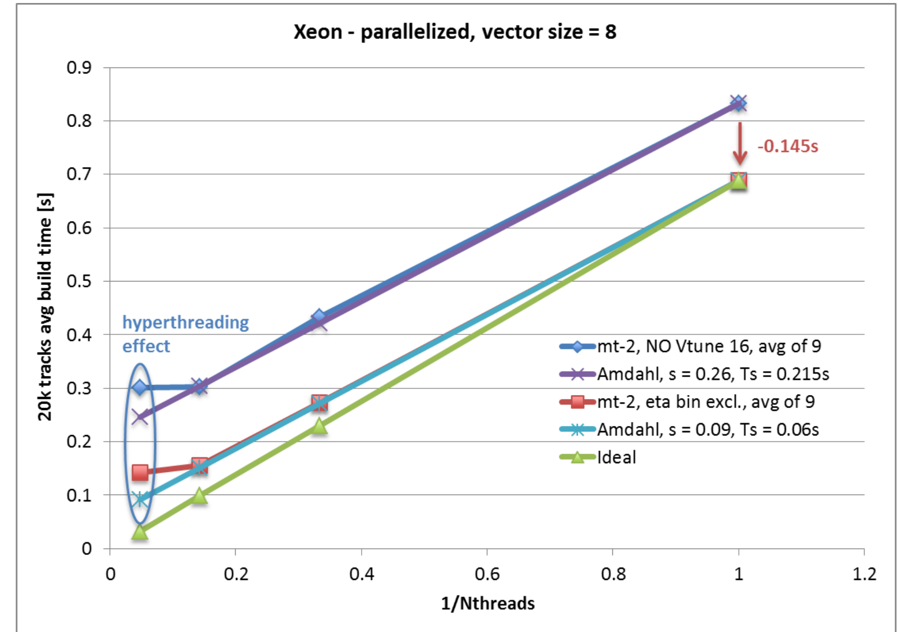
- Note, asymptote as $n \to \infty$ is not zero, but $T(1)B$
- Idea: plot the scaling data to see if it fits the above functional form
  - If it does, start looking for the source of B
  - Progressively exclude any code not in an OpenMP parallel section
  - Trivial-looking code may actually be a serial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



Cornell University
Center for Advanced Computing

Busted!





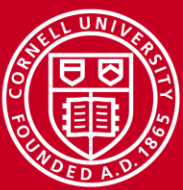
- Huge improvement from excluding *one code line* creating eta bins

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

- Accounts for 0.145s of serial code time (0.155s)

What's Going On?

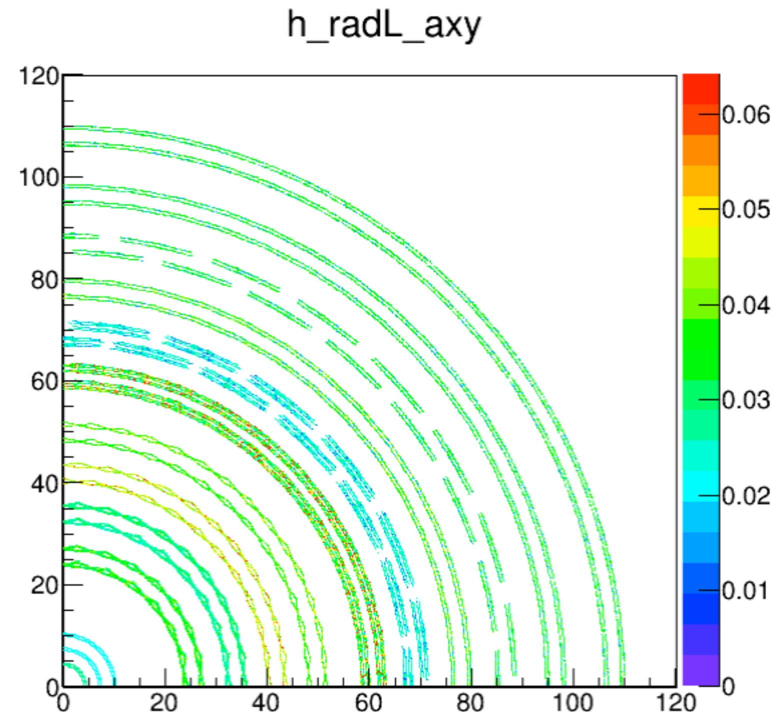
- Did a fit to the timing results on Xeon: $T(n) = T(1) * (0.74/n + 0.26)$
  - Serial fraction *B* was unacceptably large!
- Soon found that most of *B* came from re-instantiating a big data structure when starting up track-building for a new event
  - Fixed the issue by replacing deletion/creation with a simple reset
- After the fix, Amdahl *still* fits: $T(n) = T(1) * (0.91/n + 0.09)$
  - Still have some remaining *B*, or maybe there's another cause…
- Can explain residual non-ideal scaling by non-uniformity of occupancy within threads, i.e., some threads take longer than others
  - Need to define strategies for an efficient "next in line" approach
  - Need to implement dynamic reallocation of thread resources
- Work is ongoing!



Cornell University
Center for Advanced Computing

Conclusions: Tracking R&D

- Significant progress in creating parallelized and vectorized tracking software on Xeon/Xeon Phi
  - Among next steps: consider GPUs
- Good understanding of bottlenecks and limitations
  - Recent versions of the code are faster and scale better
  - Future improvements are on the way
- Have begun to process realistic data, preliminary results are encouraging
- 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 Manycore 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 manycore 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
- Manycore is quickly becoming the norm in laptops and 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