Parallelized Tracking

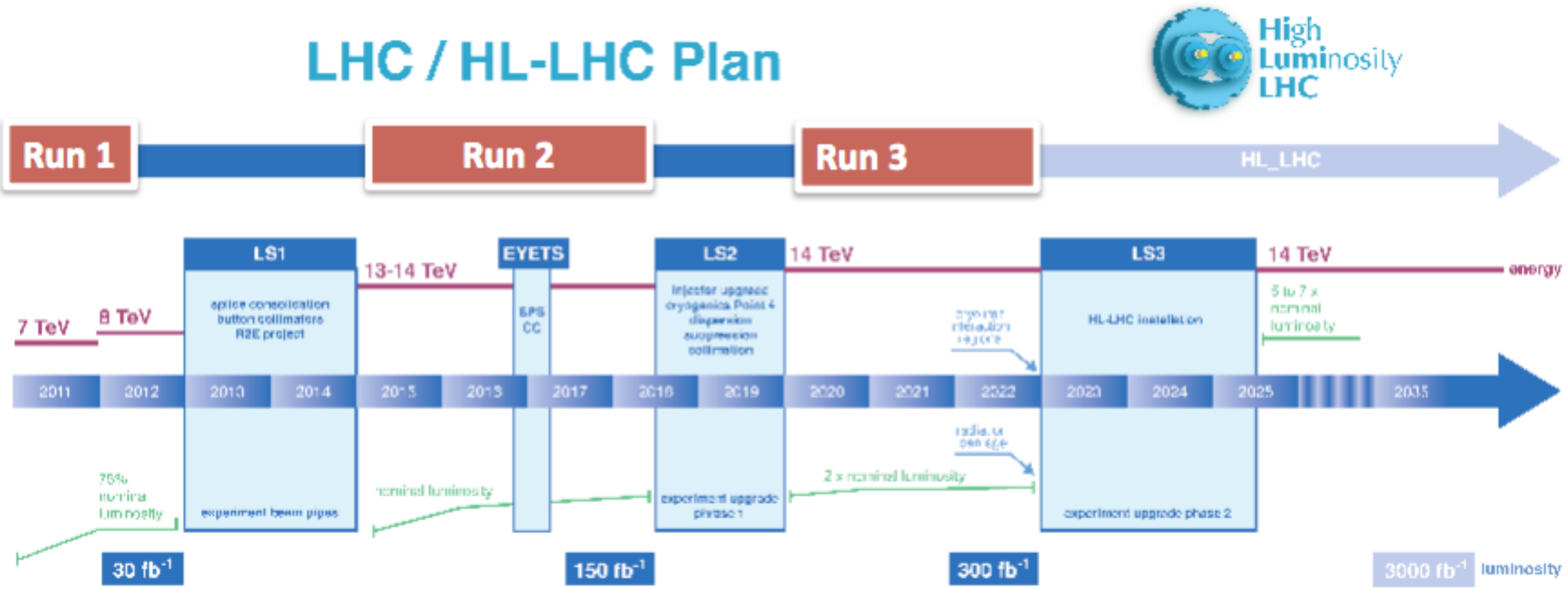
Future Trends in Nuclear Physics Computing
Mar. 16, 2016

G.Cerati (UCSD)

Preamble

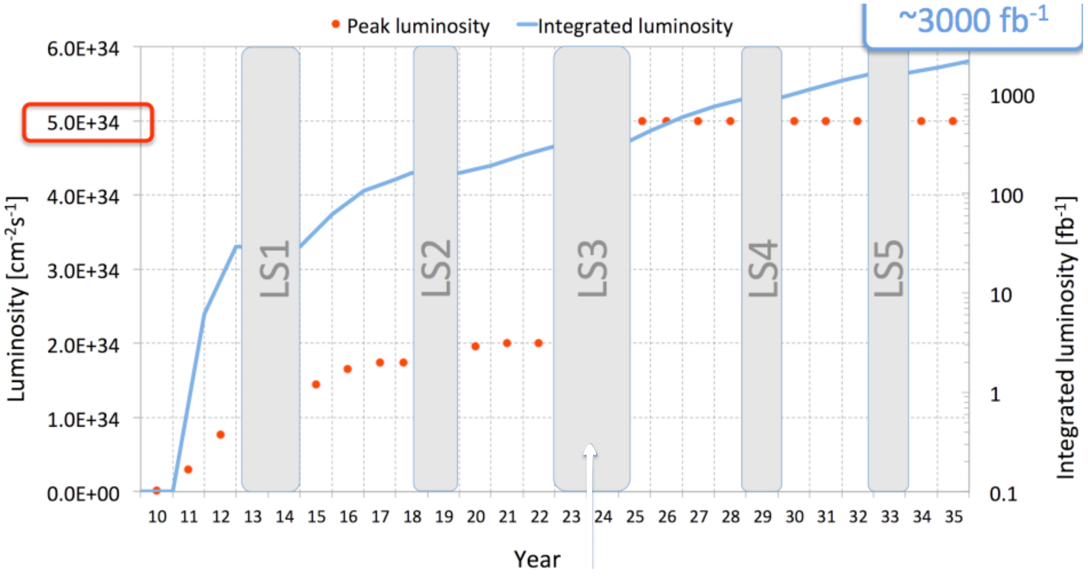
- Institutions currently involved UCSD, Cornell and Princeton
  - including PI's, ~10 people working on it (all with limited fraction of the time):
    - G.Cerati, M.Tadel, F.Würthwein, A.Yagil (UCSD)
    - S.Lantz, K.McDermott, D.Riley, P.Wittich (Cornell)
    - P.Elmer, M.~Lefebvre (Princeton)
  - recently supported by NSF with the Physics at the Information Frontier program
- R&D project is work in progress
  - this seminar is an overview and all results are intended to be preliminary/explorative studies

LHC plans

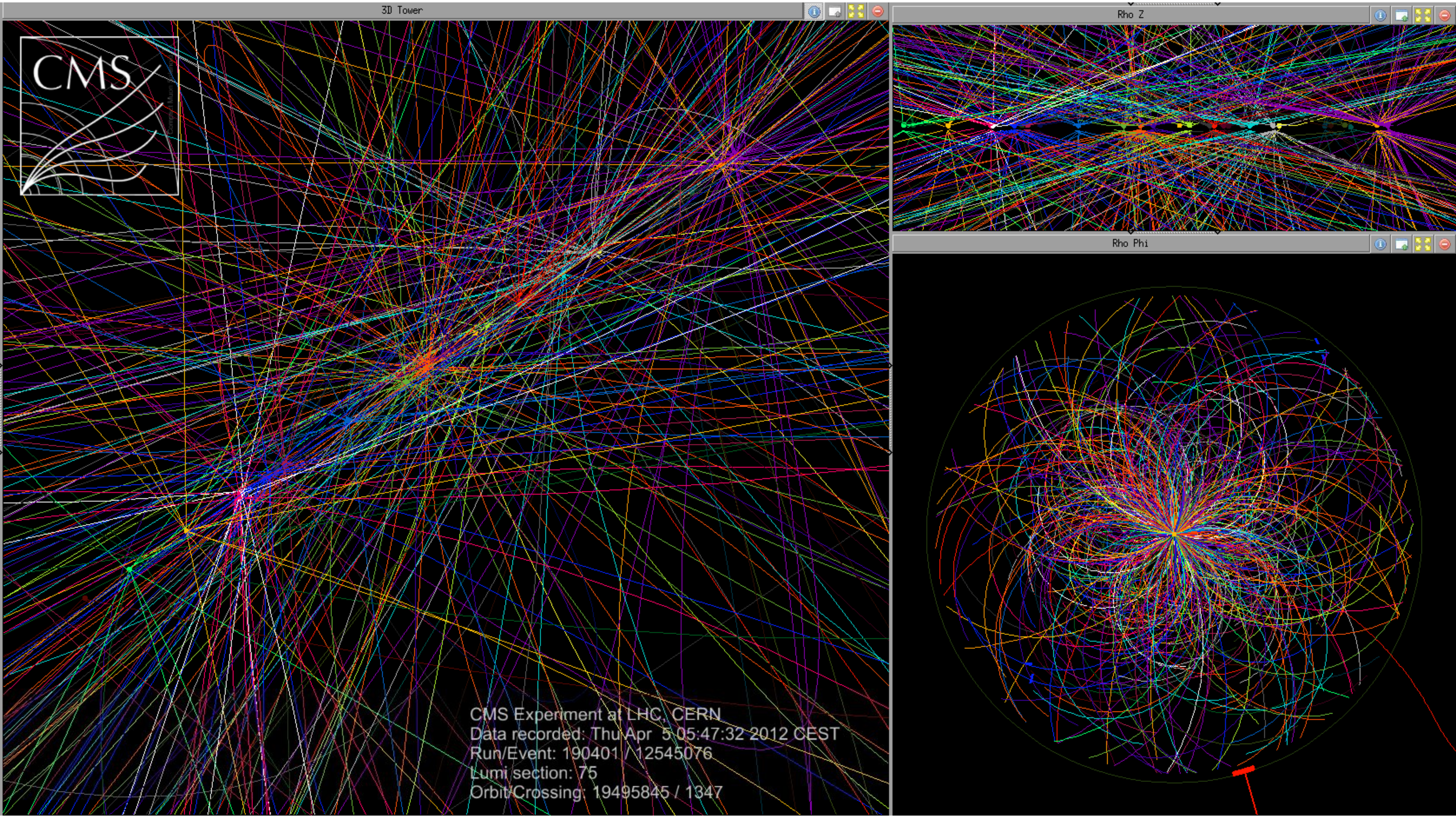


LHC is now close to its maximum collision energy. Need **as much data as possible** to see evidence of extremely rare processes accessible at this energy.

LHC is planning a smooth increase in instantaneous luminosity until the end of Run3. Then, the HL-LHC is planned with a luminosity of 5E34.

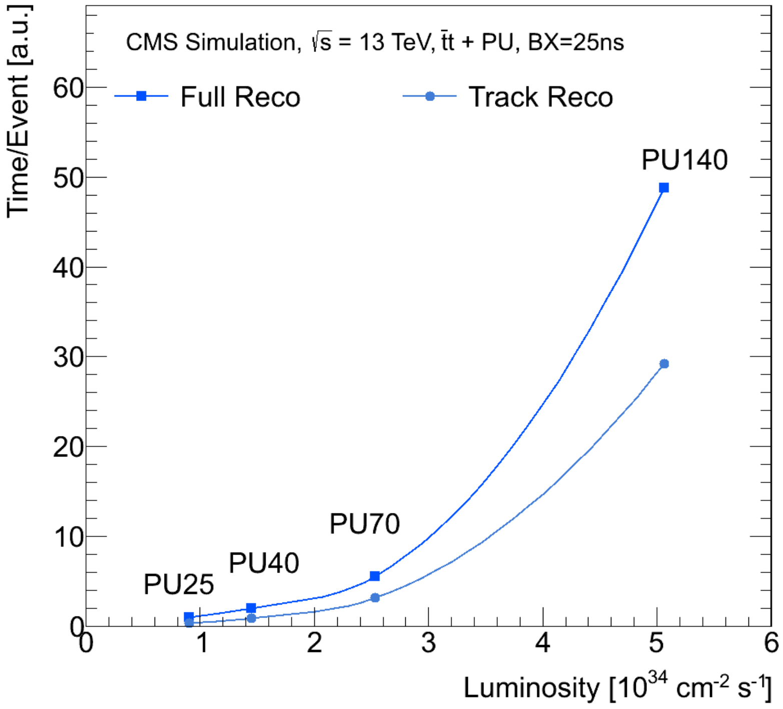


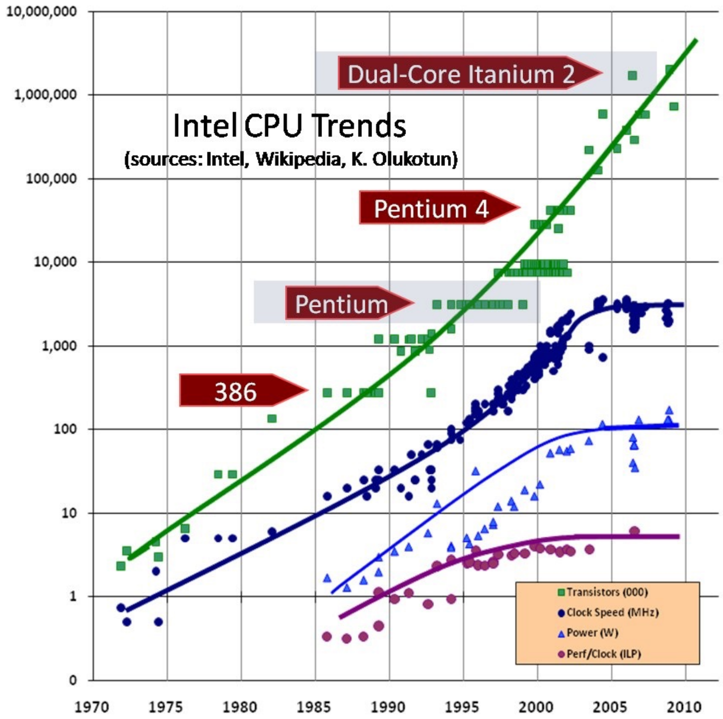
High Lumi means High Pile-up



CMS Experiment at LHC, CERN
Data recorded: Thu Apr 5 05:47:32 2012 CEST
Run/Event: 190401 / 12545076
Lumi section: 75
Orbit/Crossing: 19495845 / 1347

Reconstruction time vs PU

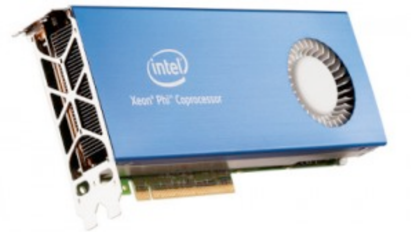




- **Reconstruction time diverges** at large (≥100) PU
  - CPU frequency does not scale with Moore’s law anymore
  - **Current model cannot be used at HL-LHC without compromises on physics!**
  - **Tracking** takes the **largest fraction** of the reconstruction time

- But Moore’s law still holds for the number of transistors
  - **Highly parallel architectures** now popular in the market,
    can we exploit them to increase the physics sensitivity?

New architectures



	Xeon E5-2670	Xeon Phi 5110P	Tesla K20X
Cores	8	60	14 SMX
Logical Cores	16 (HT)	240 (HT)	2,688 CUDA cores
Frequency	2.60GHz	1.053GHz	735MHz
GFLOPs (double)	333	1,010	1,317
SIMD width	256 Bits	512 Bits	N/A
Memory	~16-128GB	8GB	6GB
Memory B/W	51.2GB/s	320GB/s	250GB/s

Many-core architectures have lower clock frequency and lower memory than traditional multi-cores, but feature SIMD units and a large number of cores for a much larger nominal throughput

Goals

- **Need large speedup factors**, both for online and offline processing
- Online event selection
  - faster processing allows for more advanced reconstruction and selection
  - higher efficiency with respect to offline selection
  - increased purity allows decrease of thresholds for higher sensitivity
- Offline event reconstruction
  - faster reconstruction means no cuts in physics phase space to fit into time budget: more efficiency, better resolution, higher sensitivity
  - more data processed: easier reprocessing, larger MC samples, no data parking

- Eventually the full event reconstruction will have to be ported, but it is natural to start from the most time consuming algorithm, track reconstruction

- Algorithms cannot be ported in a straightforward way, need to **exploit architecture features** or will end up in slower processing
  - may need hardware-specific solutions for optimal performance
- But it's likely there will be **heterogeneous solutions**, possibly site-dependent
  - algorithm design has to be generic and applicable to different architectures

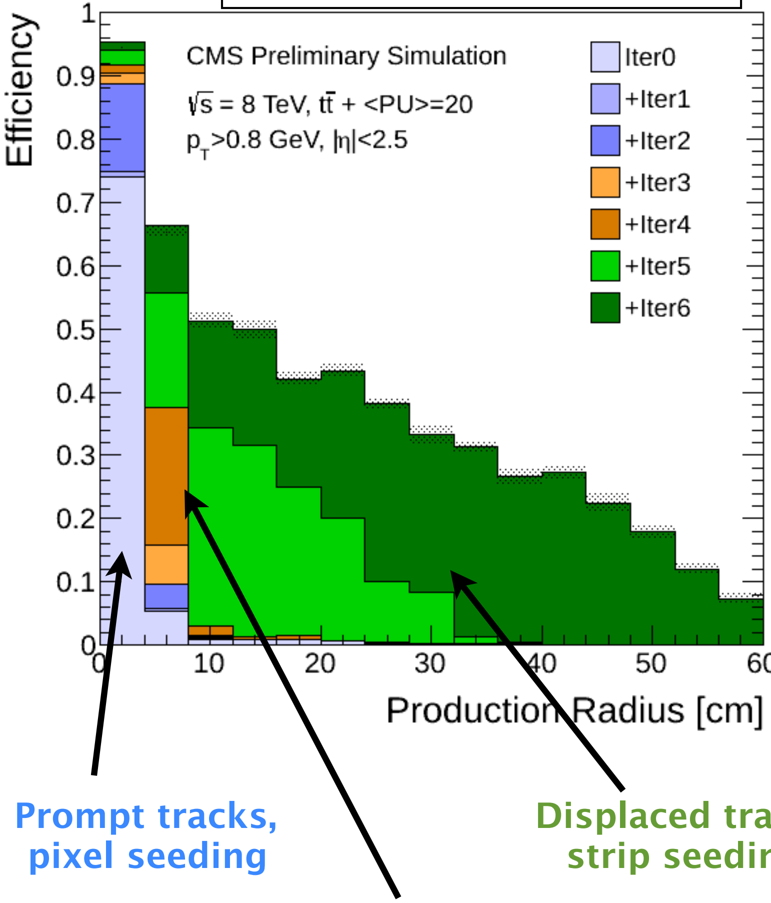
Why Xeon Phi?

- We started with no real prejudice on a specific architecture
- Xeon Phi good starting point since it is not too far from **traditional programming**
- Main features (vector units, many cores) present in smaller scale also on Xeon
  - **direct porting** of solutions/improvements across the two architectures
- But SIMD and non-SIMD processing levels are also used in GPU/CUDA programming model
  - algorithm **design** or choices can also be valid for **GPUs**
- Convenient choice given large investment for next-generation **supercomputers** based on Xeon Phi

Why Kalman Filter?

Kalman filter tracking commonly used in HEP collision experiments.
Robust treatment of material effects, so it is particularly suitable for silicon tracker.
Outstanding performance at the LHC, e.g. CMS.

PoS Vertex2014 (2015) 037



Prompt tracks, pixel seeding

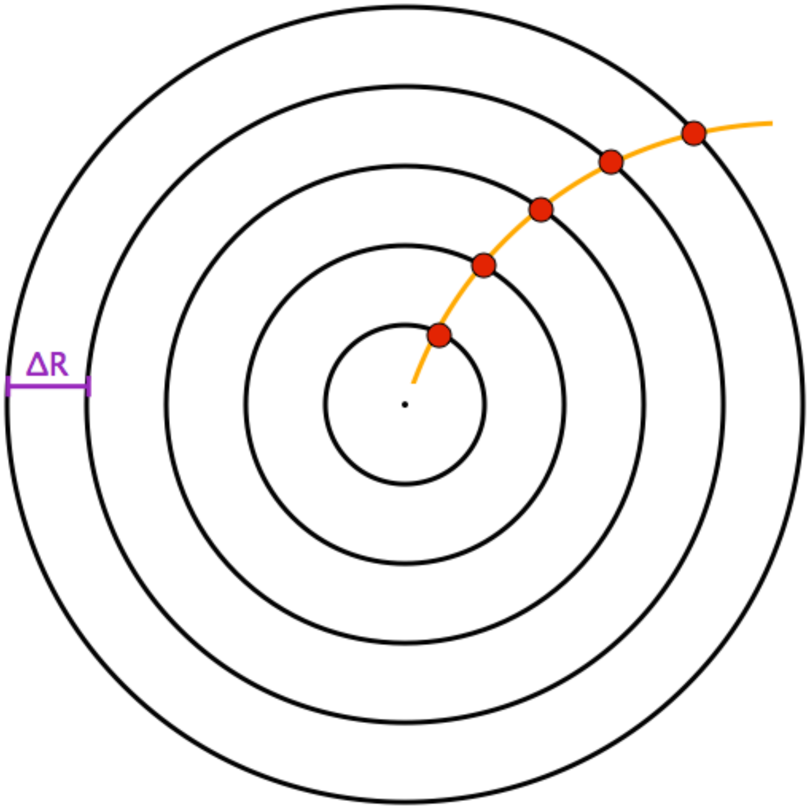
B decay tracks, mixed seeding

Displaced tracks, strip seeding

- Tracking based on Kalman Filter, divided in 3+1 main steps:
  - **seeding, pattern recognition, fitting and selection**
- Procedure repeated iteratively, removing hits associated to high quality tracks ("High Purity") to reduce combinatorics

Achieved outstanding performance:
⟹ high efficiency below 1 GeV
⟹ sizable efficiency even at R=50 cm
⟹ inclusive tracking for all collision vertices

Experimental setup



Simplified, standalone tracking code:

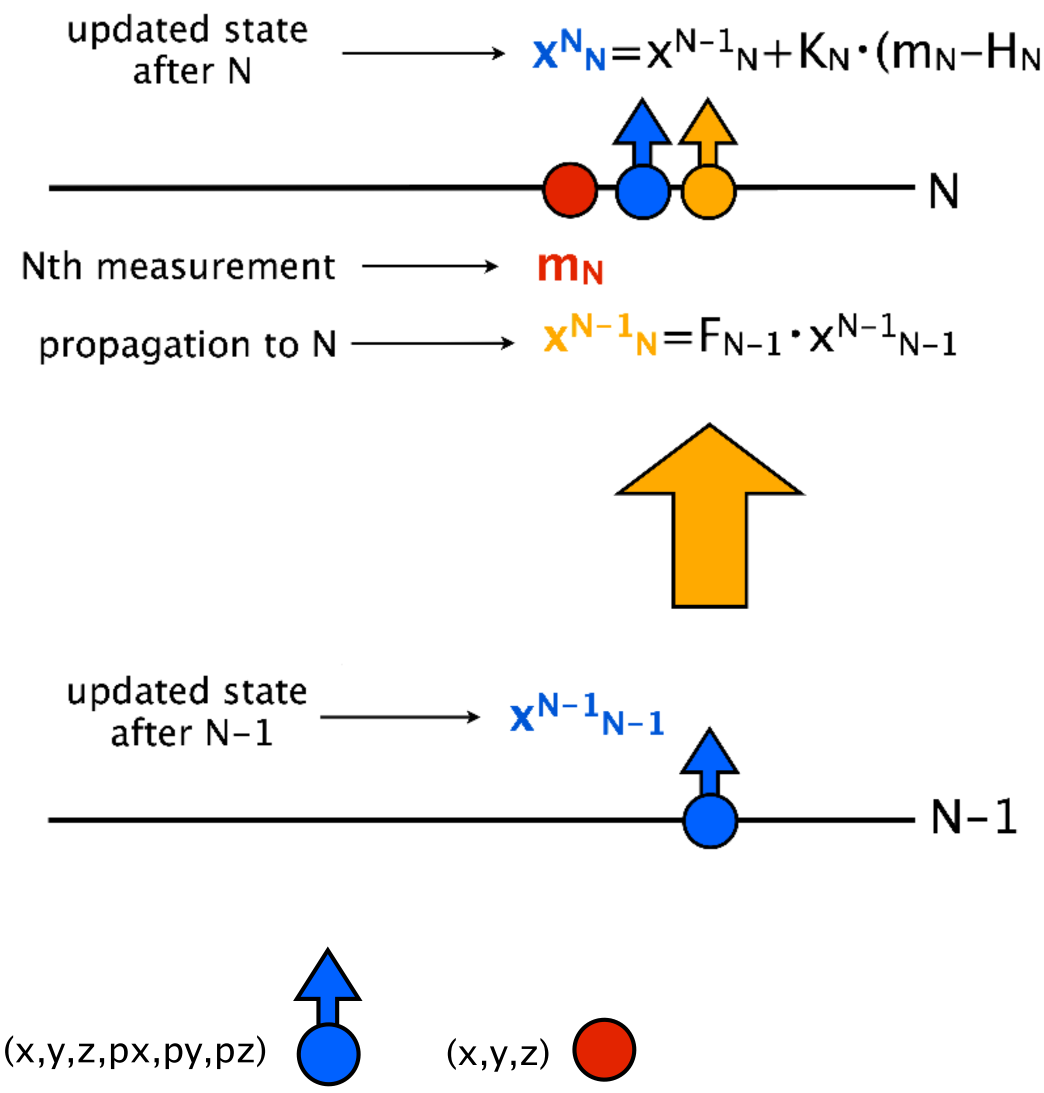
- Tracker with 10 barrel layers, perfectly cylindrical, separated by ΔR=4 cm
- Longitudinal bounds: $|\eta|<1$
- 3.8 T magnetic field, coaxial with the tracker
- Assume beam spot width 1mm in xy and 1cm in z
- Hit resolution 100μm in r-phi, 1mm in z
- No material, no inefficiencies
- Work in global coordinates

- Particle-gun generation of tracks with $0.5<p_T<10$ GeV
- Tracks uncorrelated for now, no jets, no decays
- Hits are recorded smearing the ideal crossing point by the assumed resolution, hit uncertainty set equal to resolution.

Simplified setup is the starting point,
we are gradually increasing complexity towards full simulation

Kalman Filter: basics

The Kalman filter can be seen as the iterative repetition of the same **logic unit**.



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

(x,y,z,px,py,pz)

(x,y,z)

- Smallest logic unit is the base both of track fitting and track building

- After updating with the hit measurement, the state at layer N has smaller uncertainty than at layer N-1

- In reality, it is actually a bit more complicated than this picture (energy loss, multiple scattering, hit position re-evaluated using track direction)

Kalman Filter reconstruction

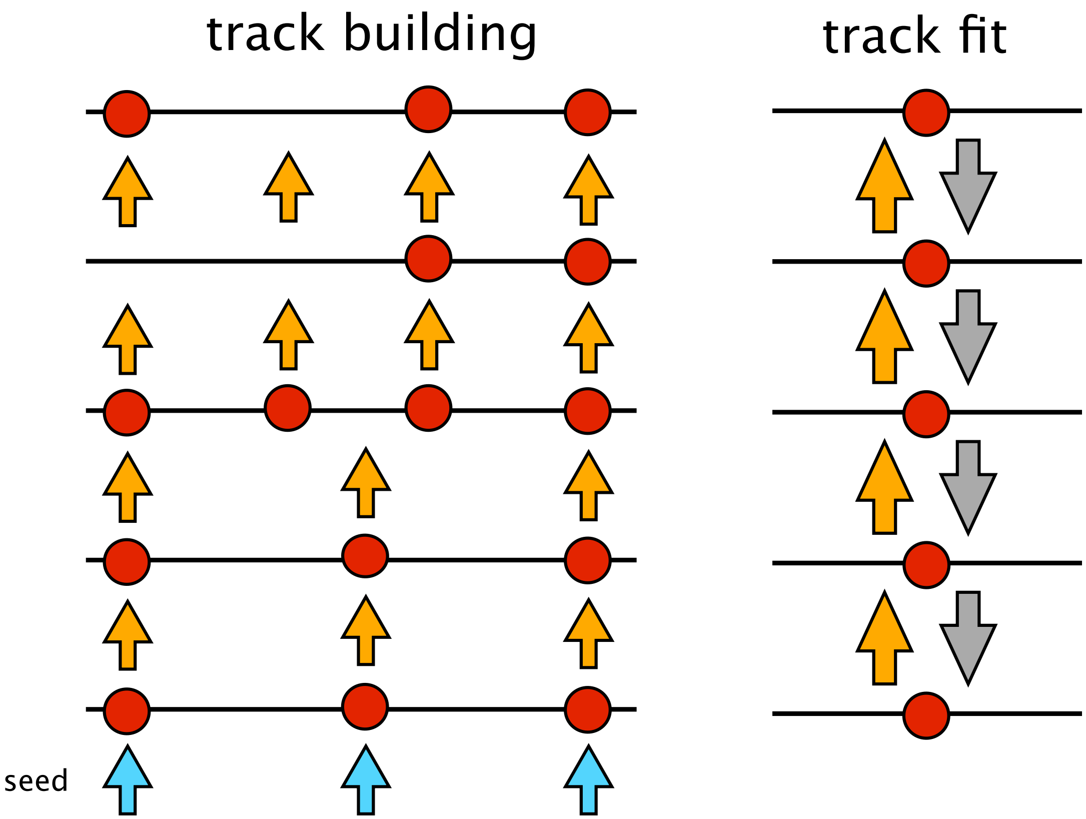
The Kalman Filter track reconstruction searches for hits along the track direction, with a search window that shrinks when more measurements are added.

The track reconstruction process can be divided in 3 steps: track seeding (initial track prototype), building (hit finding) and fitting (final parameter estimate).

The **track fit** is the bare repetition of the basic unit, ideal as a **starting point**.

Track **building is the most time consuming part** – it involves branching points of variable size, with the simplest version degenerating into the track fit case.

Track **seeding** not fully implemented yet, for now seeds are defined using MC info.



Challenges for parallelization/vectorization

- The current incarnation of the Kalman Filter track building cannot be successfully parallelized and vectorized in a straightforward way

- Each track lives in a different **micro-environment**
  - non-homogeneous workload per track
  - difficult for thread balancing

- **Branching points** (decisions) at each layer
  - hardly predictable variable number of branches are created
  - intrinsically non-SIMD

- Large **use of memory** to access geometry, magnetic field, alignment, conditions

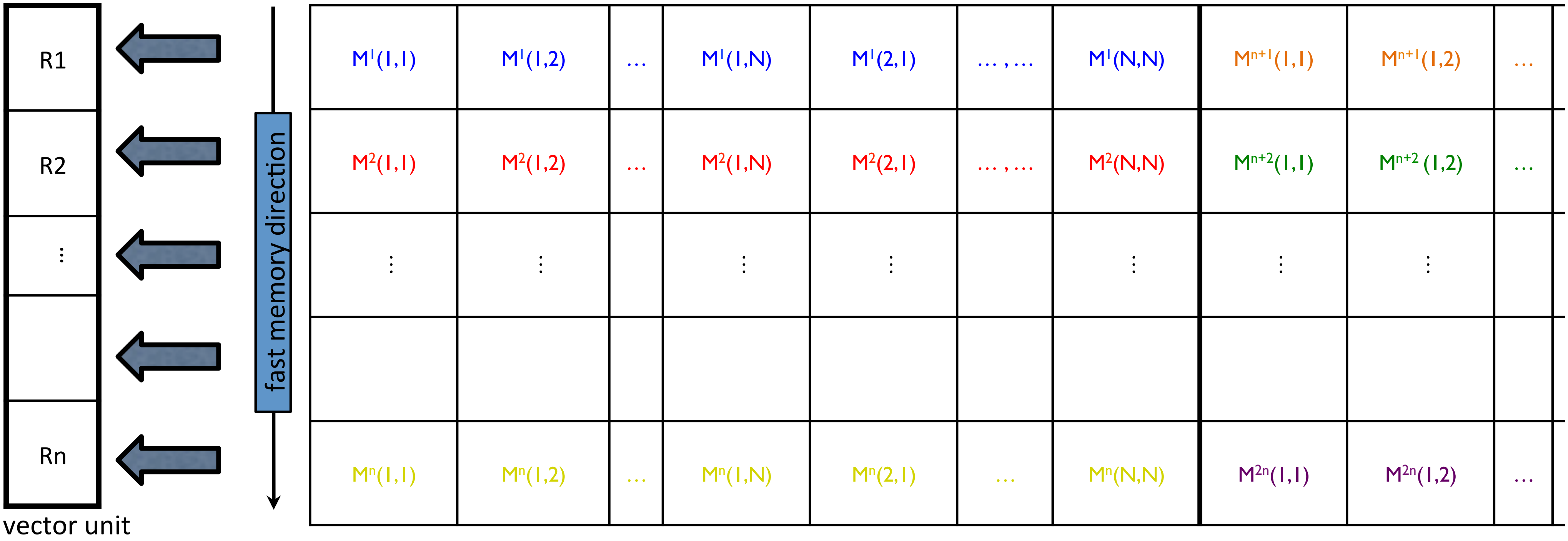
- Track fitting not affected by the first two issues: simple starting point

Matriplex

Kalman filter calculations based on small matrices.
Intel Xeon and Xeon Phi have **vector units** with size 8 and 16 floats respectively.
How can we efficiently exploit them?

Matriplex is a “matrix-major” representation, where vector units elements are separately filled by a different matrix: **n matrices work in sync**.

In other words, vector units are also used for SIMD parallelization
(in addition to parallelization from threads in different cores)

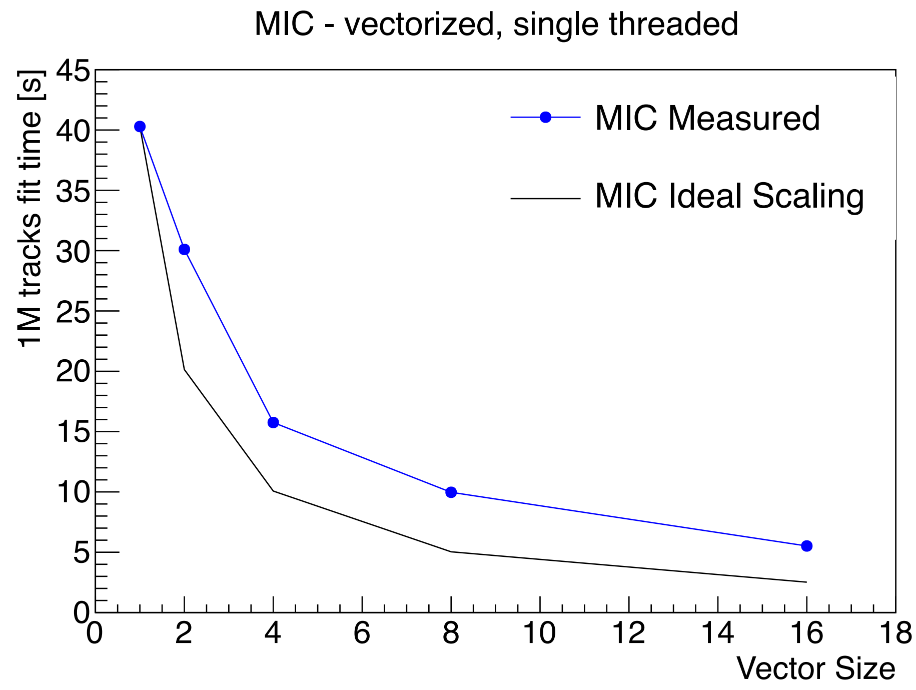


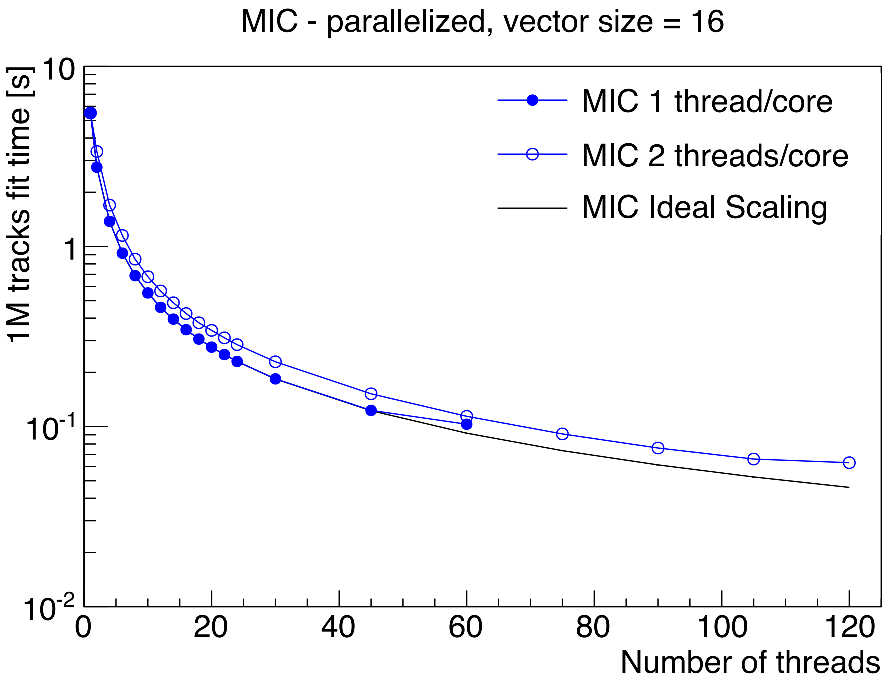
$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	… , …	$M^1(N,N)$	$M^{n+1}(1,1)$	$M^{n+1}(1,2)$	…
$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	… , …	$M^2(N,N)$	$M^{n+2}(1,1)$	$M^{n+2}(1,2)$	…
⋮	⋮		⋮	⋮		⋮	⋮	⋮	
$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$	$M^{2n}(1,1)$	$M^{2n}(1,2)$	…

Matrix size **NxN**, vector unit size **n**

Fitting time and speedup

arXiv:1409.8213





Track fit implemented and tested both on Intel Xeon and Xeon Phi (native application) with OpenMP and got similar qualitative results.

Observe **significant speedup both from vectorization and parallelization.**

Effective performance of vectorization is about 50% utilization efficiency.

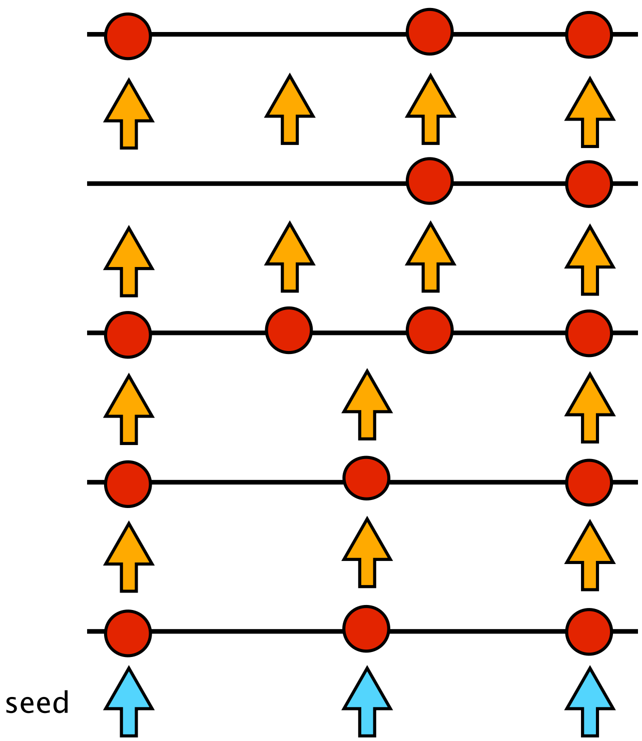
Parallelization performance is close to ideal in case of 1 thread/core, while with 2 threads/core an overhead is observed. Problems are related to L1 cache issues.

Demonstrated feasibility on the fitting case, track building is the next target.

Track Building

- Same core calculations as in track fitting but adding two big complications
  - **Hit set is not defined**: hit on next layer to be chosen between O(10k) hits
  - For >1 compatible hit, combinatorial problem requires **cloning of candidates**

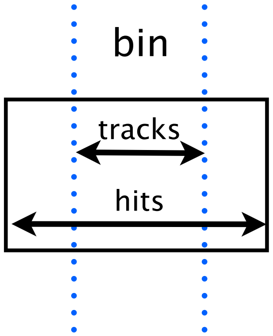
- The two issues can be **factorized** by dividing the development in two stages
  - first develop a simplified algorithm choosing only the **best hit** on next layer
    - deal with large number of hits, not with cloning
    - study vectorization in this case first
  - then full implementation with **combinatorial** expansion
    - parallelization already using this version!

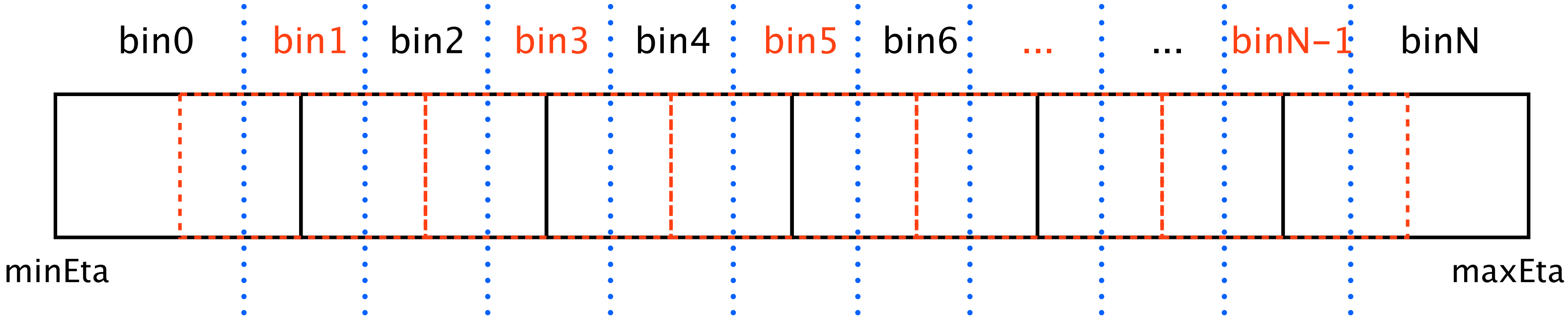


Space Partitioning for Track Building

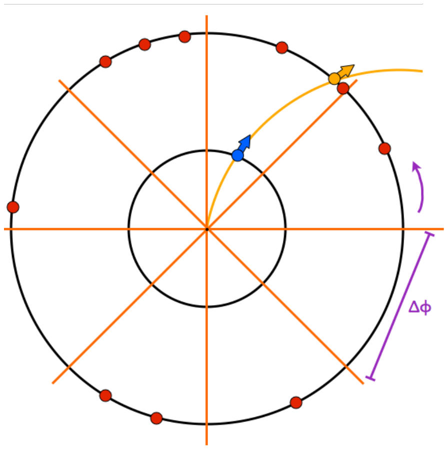
- **Data locality** is the key for reducing the Nhits problem
  - partition the space without any detailed knowledge of the detector geometry structures
  - **eta partitions** are **self consistent** (no bending)
    - bins redundant in terms of hits, track candidates never search outside their eta bin
    - simple boundary for **thread definitions**
  - **phi partitions** give **fast lookup** of hits in compatibility window

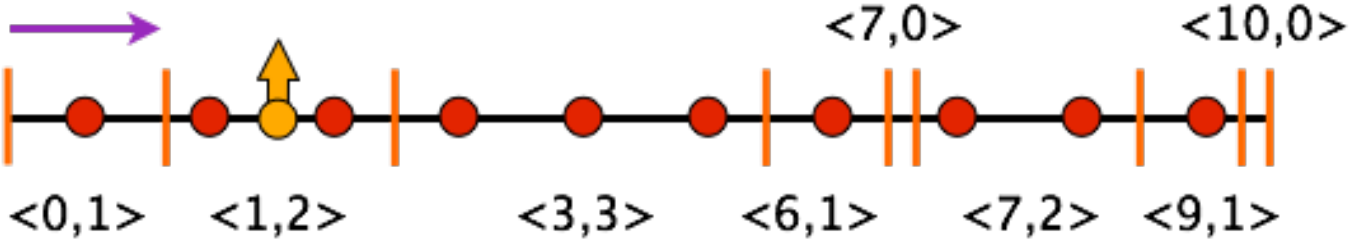
eta partitions:



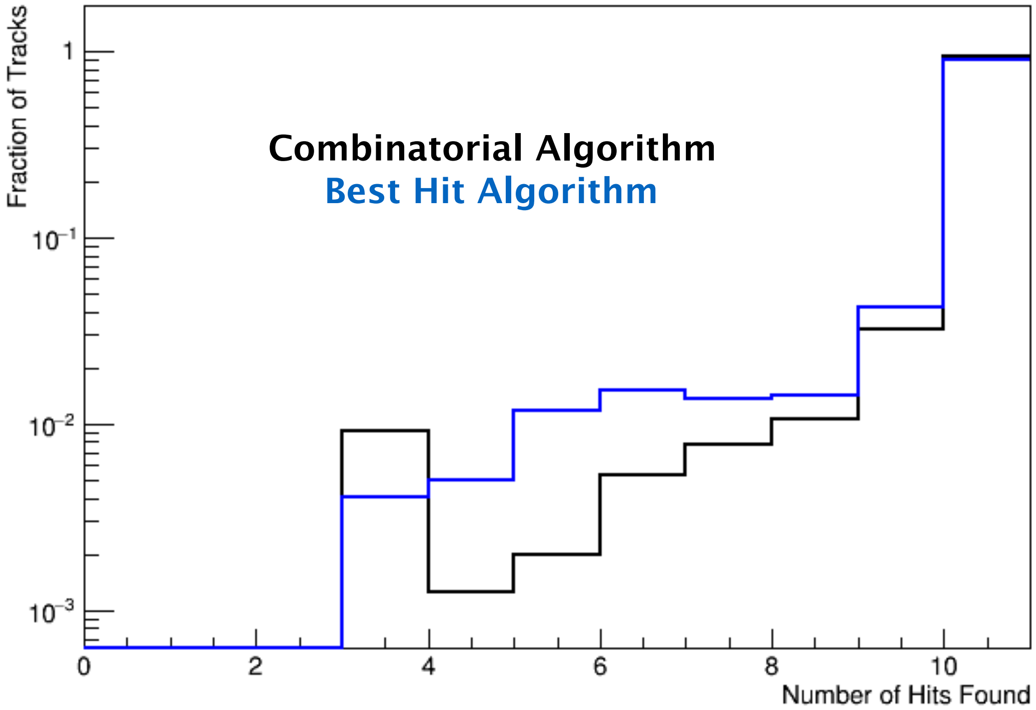


phi partitions:





Hit Finding Performance



- Simulate events with 20k tracks per event on simplified setup
  - reconstruct using seeds taken from MC truth

- **Combinatorial Algorithm**: 96% (99%) of tracks found with ≥90% (60%) of the hits.

- **BestHit Algorithm**: 94% (98%) of tracks found with ≥90% (60%) of the hits.
  - BestHit version not expected to behave so well with fully realistic setup

Reducing Impact of Memory Operations

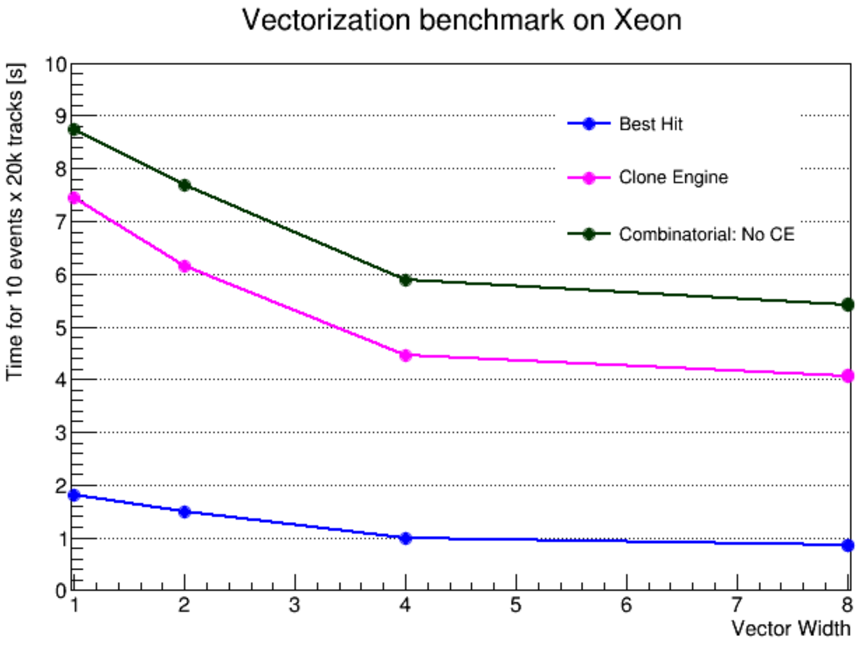
- First results on speedup for track building presented at CHEP2015 (arXiv: 1505.04540)
- Analysis with VTune Hotspots revealed **bottlenecks from memory** operations

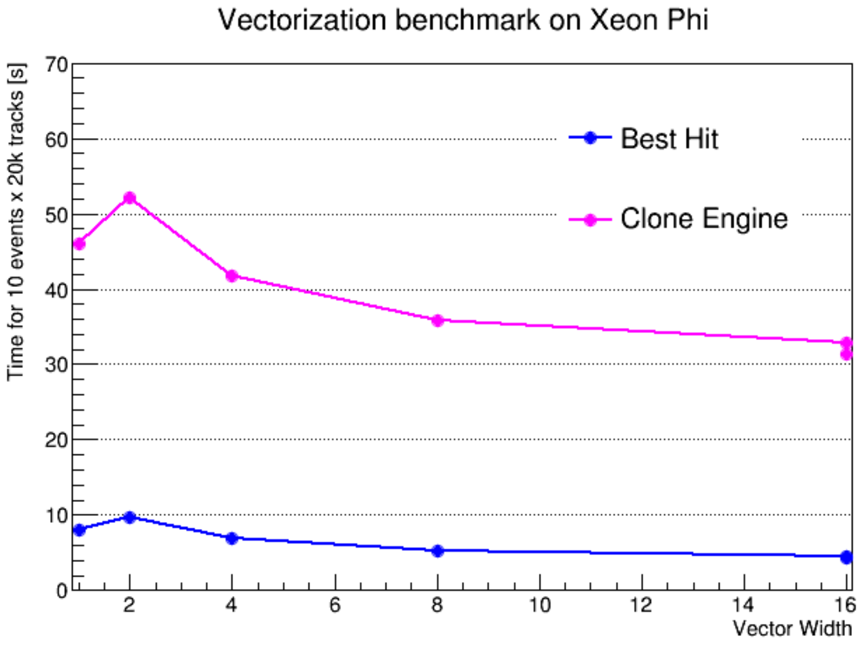
- Several **technical improvements** to the memory management
  - Avoid resizing of hit indices vector in track object
  - Reduce size of Hit and Track objects to minimum (objects with heaviest impact on memory)
  - Recycle partitioning data structures every event (reset instead of new instantiation)
  - Reduce number of conversions to and from Matriplex format

- **Algorithmic changes**: mitigate impact from serial work of copying track candidates in combinatorial approach by moving copying outside of vectorized operations: **Clone Engine**
  - fill a bookmarking list inside vectorized code
  - process it at the end of the vectorized loop minimizing the memory operations

- New results presented at Connecting the Dots workshop
  - Proceedings in preparation

Vectorization Results (time)

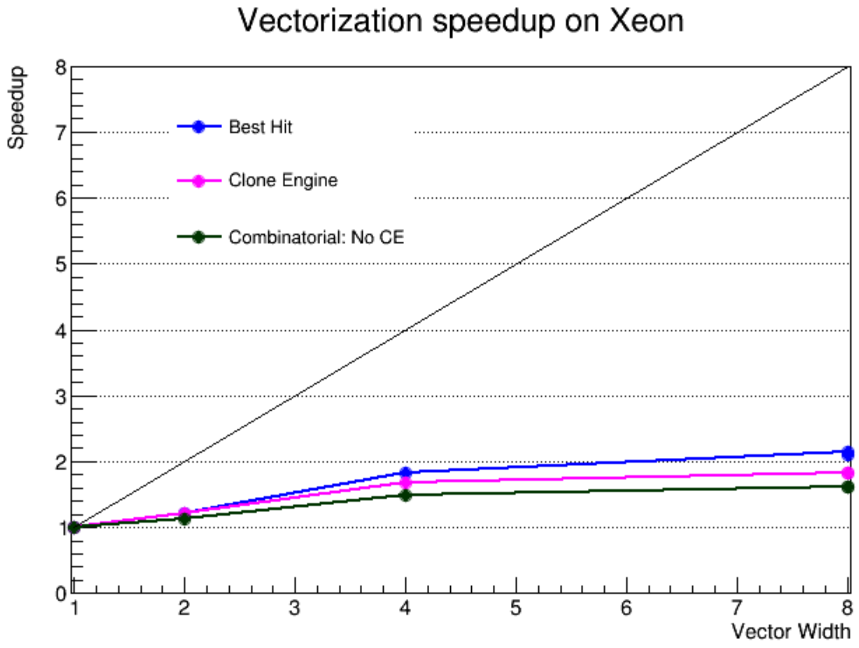


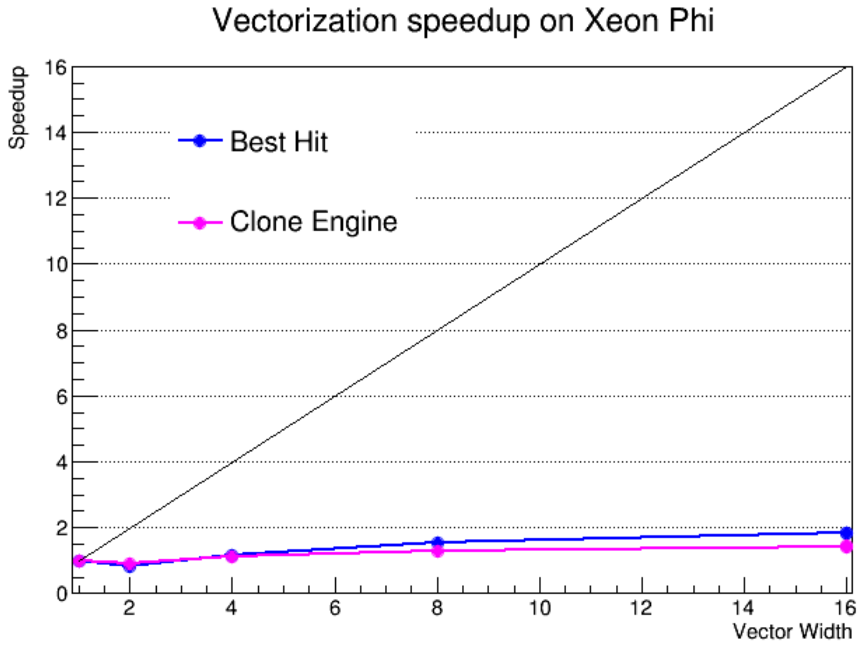


- Expect worse results than simple fitting case
- **BestHit**: no cloning, faster but worse hit finding performance
- CloneEngine version faster than **standard combinatorial** algorithm

- Use single-threaded processing for vectorization studies
- Results show a **maximum speedup of ~2x** both on Xeon and Xeon Phi
  - reasonable scaling on Xeon
  - overhead observed when enabling vectorization on Xeon Phi, then speedup

Vectorization Results (speedup)

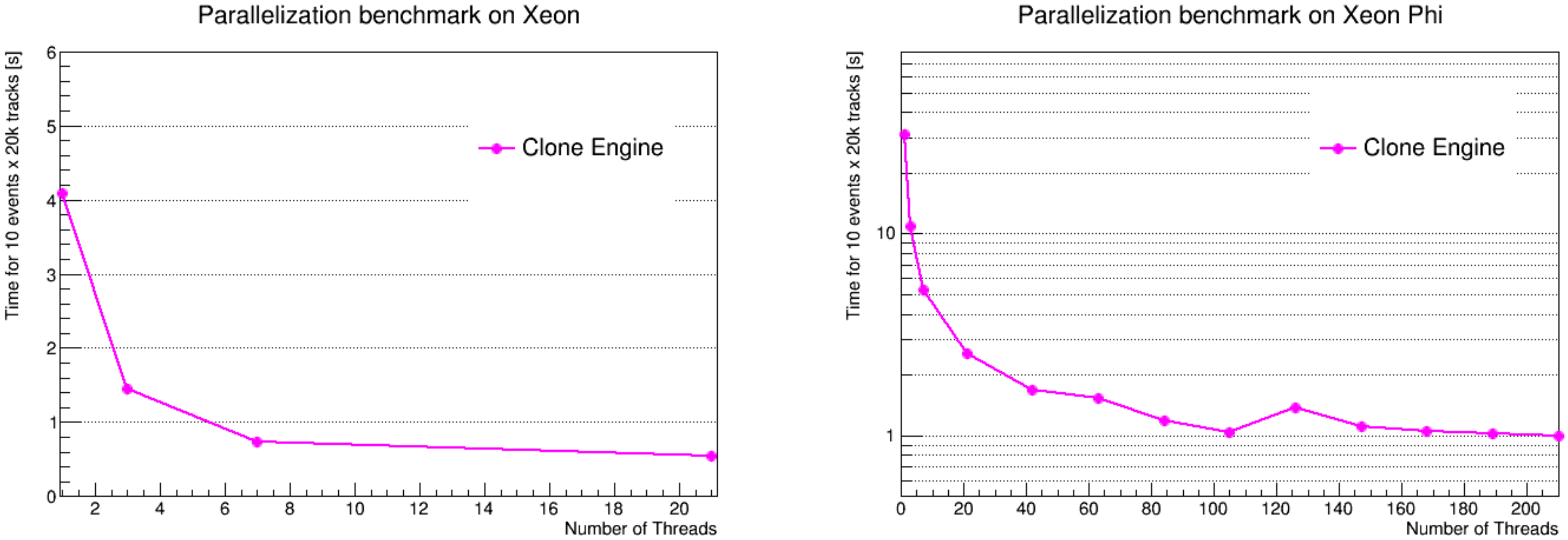

Vectorization speedup on Xeon


Vectorization speedup on Xeon Phi

- Expect worse results than simple fitting case
- **BestHit**: no cloning, faster but worse hit finding performance
- **CloneEngine** version faster than **standard combinatorial** algorithm

- Use single-threaded processing for vectorization studies
- Results show a **maximum speedup of ~2x** both on Xeon and Xeon Phi
  - reasonable scaling on Xeon
  - overhead observed when enabling vectorization on Xeon Phi, then speedup

Parallelization Results (time)

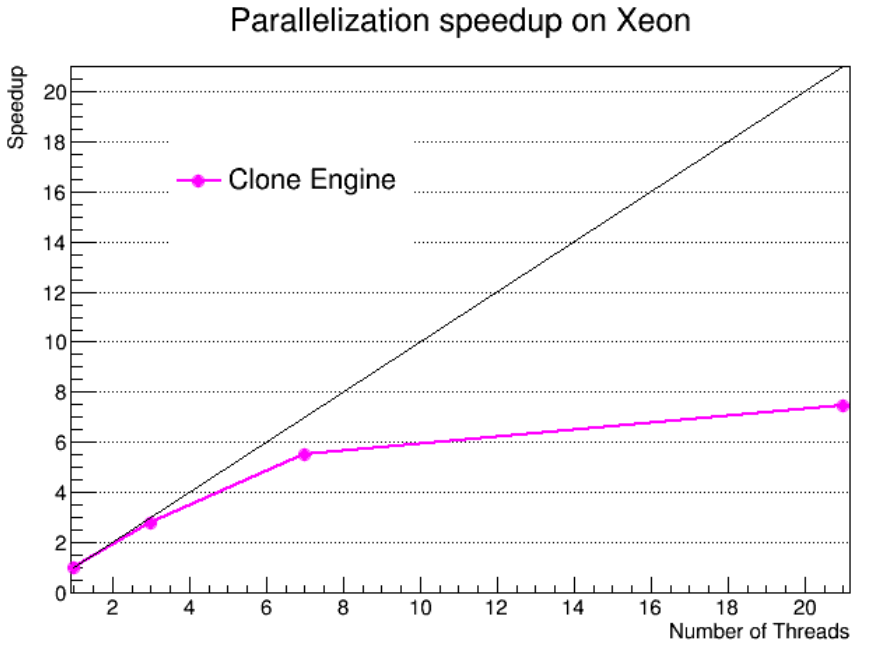


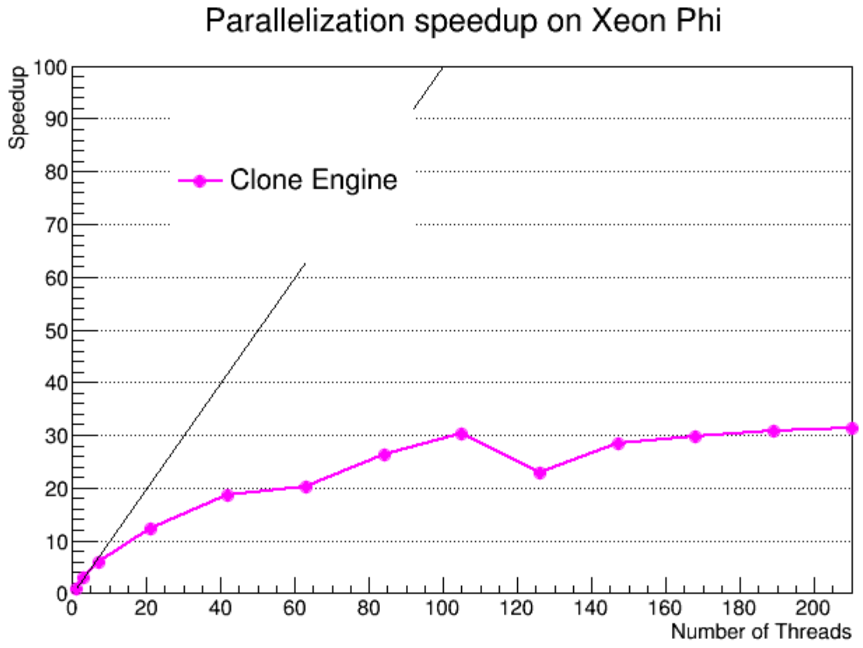
- Use fully vectorized processing for parallelization studies
- Focus on **CloneEngine**

- Parallelization is implemented by **distributing threads across 21 eta bins**
  - for nEtaBin multiple of nThreads, split eta bins in threads
  - for nThreads multiple of nEtaBin, split seeds in bin across nThreads/nEtaBin threads

- Large **speedup** achieved, both on Xeon and Xeon Phi
  - up to **>7x on Xeon and >30x Xeon Phi**

Parallelization Results (speedup)





- Use fully vectorized processing for parallelization studies
- Focus on **CloneEngine**

- Parallelization is implemented by **distributing threads across 21 eta bins**
  - for nEtaBin multiple of nThreads, split eta bins in threads
  - for nThreads multiple of nEtaBin, split seeds in bin across nThreads/nEtaBin threads

- Large **speedup** achieved, both on Xeon and Xeon Phi
  - up to **>7x on Xeon and >30x Xeon Phi**

Ideas to improve speedup

- Further analysis revealed that the main bottleneck for non ideal speedup performance are **thread and vector imbalance**

- We process 8/16 candidates in the same vector unit on Xeon/XeonPhi
- **Different number of probed hits** per candidate lead to dead time
  - in case the search window is very different
  - in case the local occupancy is very different
  - in case there is a track that goes crazy
- Idea: fill vector units after **sorting** track candidates in a smart way
  - sort by position on next layer, sort by curvature, sort by chi2, …

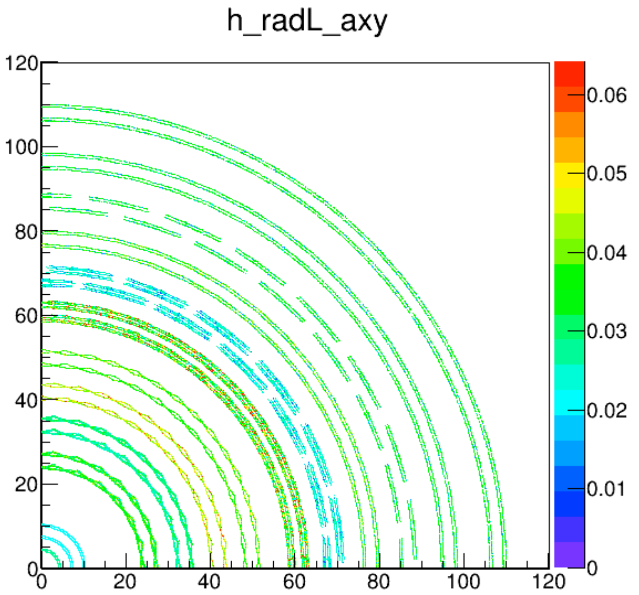
- **Simple division of threads** in eta partitions is too naive
  - some partitions take more time, will be even more evident when processing events with jets
- Main idea is to move to a "next-in-line" approach
  - as opposed to an "upfront" distribution of the work
- Work ongoing to move from OpenMP to **TBB** for more flexibility in the workload definition

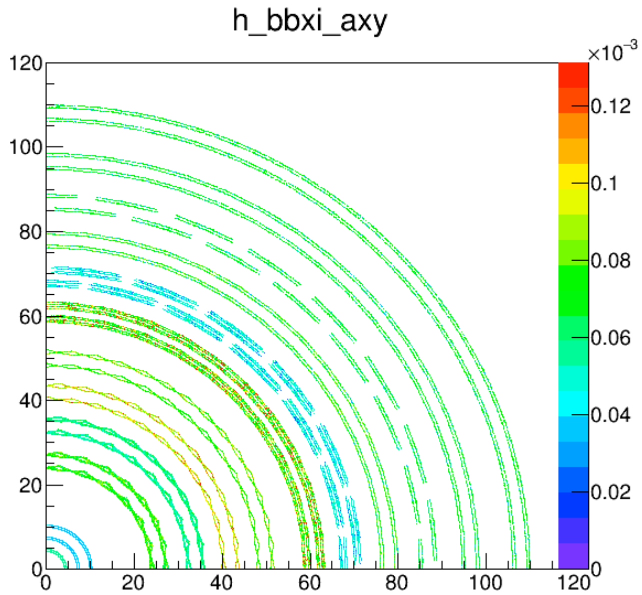
Full simulation results

- We are working towards reconstructing tracks from a **full simulation** or from real **detector data**
  - non ideal geometry, material effects
  - detector inefficiencies, non-gaussian tails in hit position
  - particle clustering in jets, particle decays

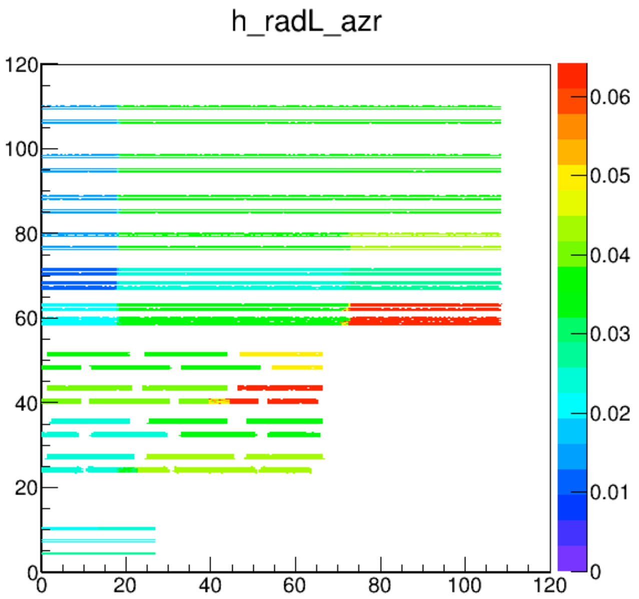
- The simplest way to do this is to **interface with CMSSW**
  - indirect way, dumping and reading from an ntuple
  - save and link information from all tracking-related collections
    - hits, seed, tracks – both simulated and reconstructed
  - maximal flexibility:
    - use simulation, local reconstruction or steps in global reconstruction as our starting point
    - allow direct comparison of same events with CMSSW reconstruction (but this is not so straightforward)
  - now used also for other tracking studies in CMS

Material in CMSSW reconstruction



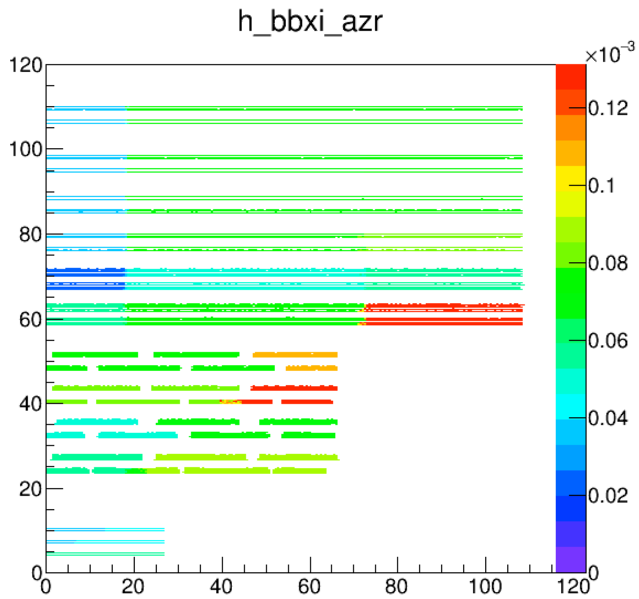


In CMSSW reconstruction the tracker material is described in terms for **radiation length** (radL) and **energy loss** constant ($\xi = Kz^2Z/A$ in Bethe-bloch)

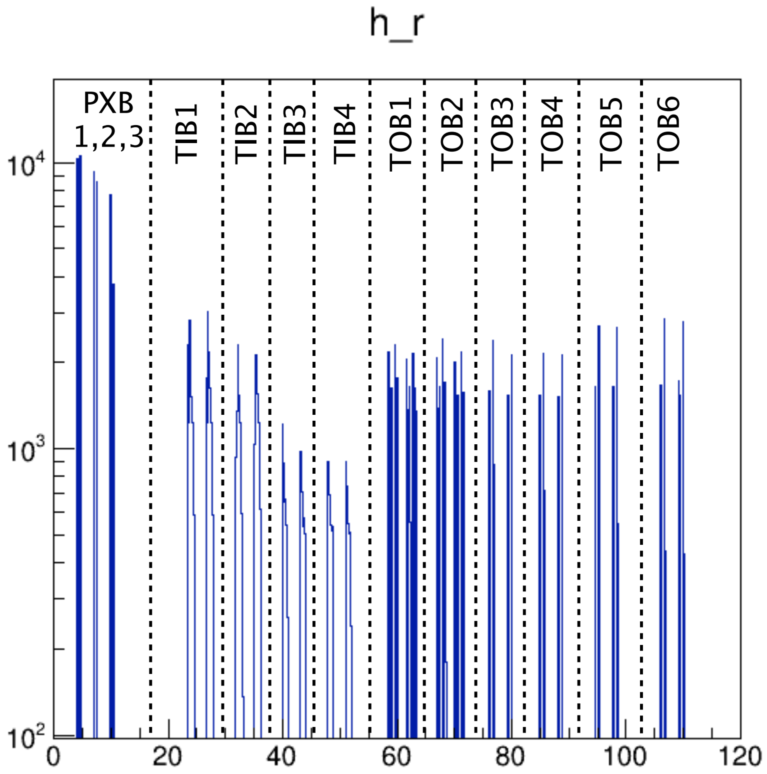


All material (including services) is assumed on the detector module, so the effect is that the parameters are flat in phi but vary significantly vs r and z

We **parametrize** these values vs |z| for each layer.



Detector geometry model



Average radii [cm]:

PXB1	=	4.42
PXB2	=	7.31
PXB3	=	10.17
TIB1	=	25.65
TIB2	=	33.81
TIB3	=	41.89
TIB4	=	49.67
TOB1	=	60.95
TOB2	=	69.11
TOB3	=	78.19
TOB4	=	86.84
TOB5	=	96.78
TOB6	=	108.10

Build reco geometry with cylindric barrel layer at **average radii**.
First propagation step to average radii, find hits in compatibility window.

For hits in window, perform second propagation step:
compute chi2 and update parameters at exact hit radius.

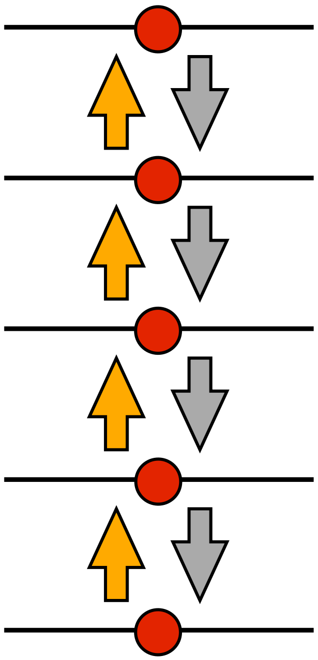
Advantage: work with a **simplified geometry**, no need to store in memory geometry structure details. Easy to readapt to different detectors with similar structure.
Disadvantage: compatibility window has to be inflated to correct for spread in R.

Conclusions

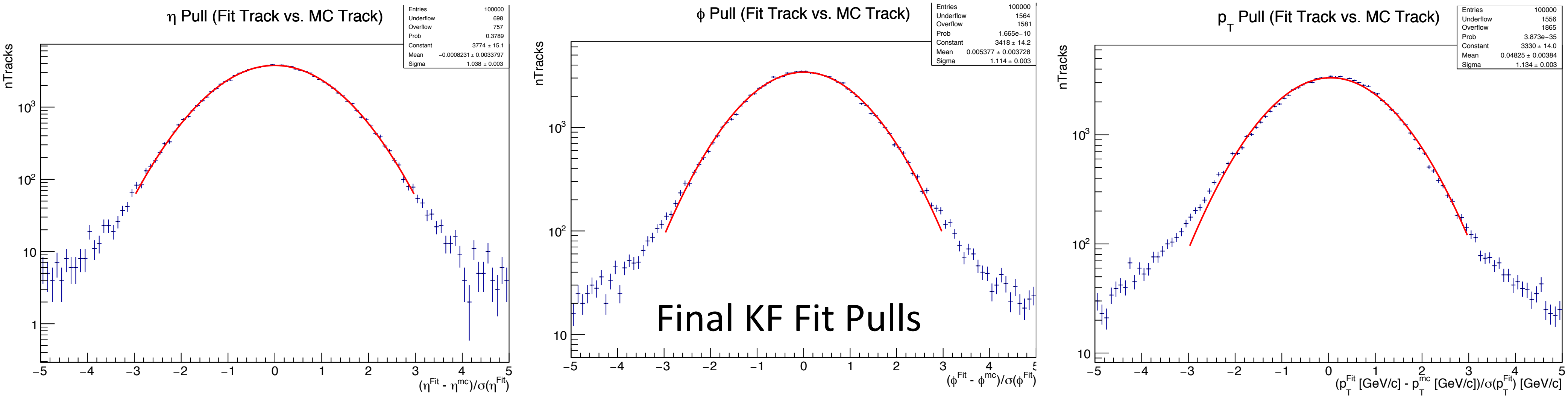
- R&D for tracking on parallel architectures gives promising preliminary results on Xeon/XeonPhi
  - large speedup both from vectorization and parallelization
  - Main bottlenecks are identified, further improvements are being explored
- Code interfaced to CMS full simulation, geometry and material properties
  - keep simple detector description to save memory, use parametrizations for same precision
- Recently started porting the code to Cuda and tested on GPU: encouraging results
- Next steps:
  - improve performance using full simulation input, including collision events (with PU)
  - complete fully vectorized/parallelized tracking chain: seeding+building+fitting
  - consolidation/implementation of new ideas
  - …
  - comparison with current reconstruction model
  - porting and deployment in official reconstruction

backup

Track Fitting

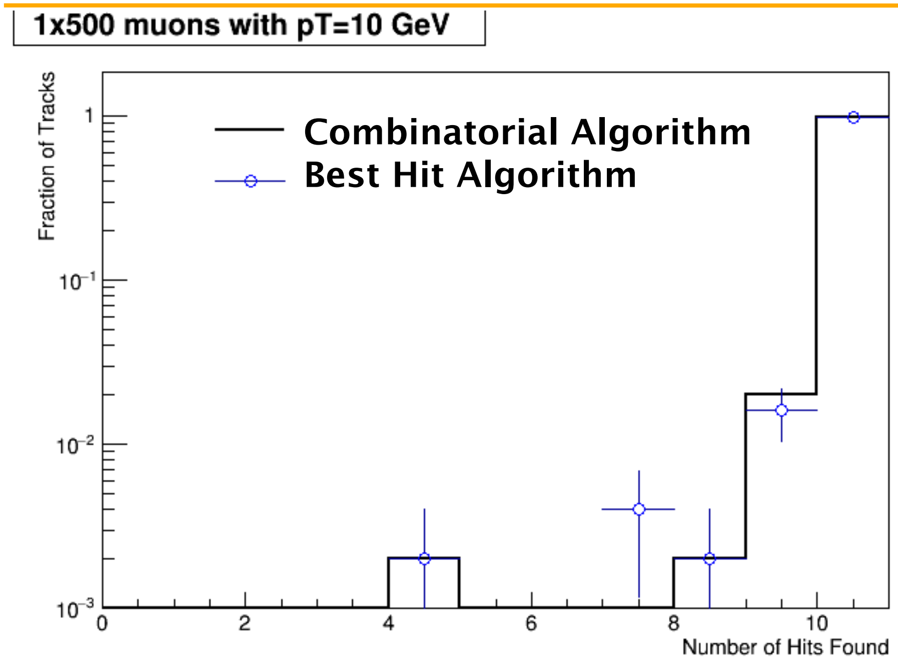


- Kalman Filter fitting is the repetition of propagation and update of parameters through the pre-determined list of hits
- Fit can be forward, backward or a combination of both for ultimate precision
  - Kalman Filter needs an initial state, in our setup it can be both from simulation or from a fast parabolic fit of 3 points
- Resulting performance are very good, with ~unitary width pulls and p_T resolution 0.5×p_T [%]

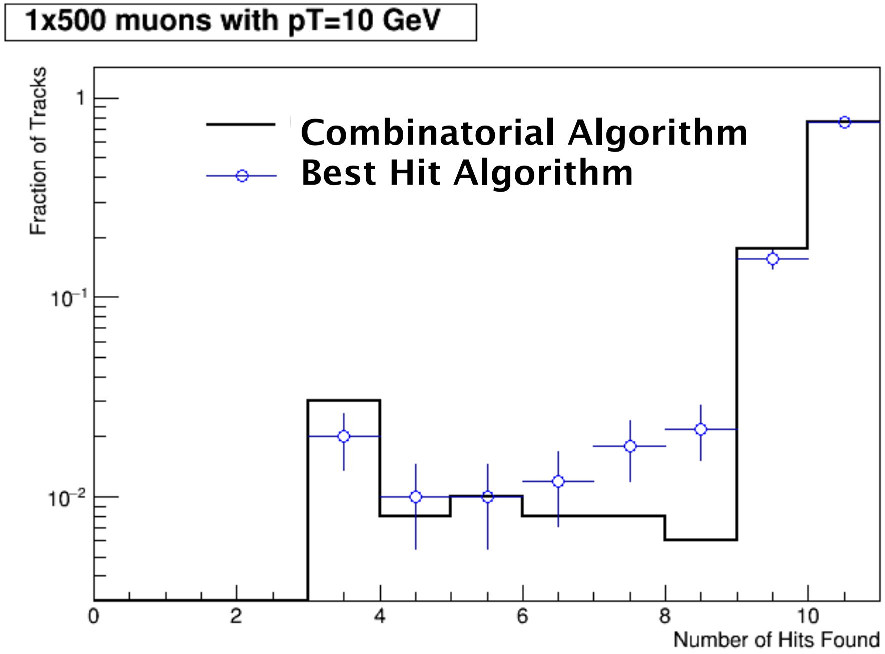


Final KF Fit Pulls

Hit Finding Performance



Sim Hits from CMSSW

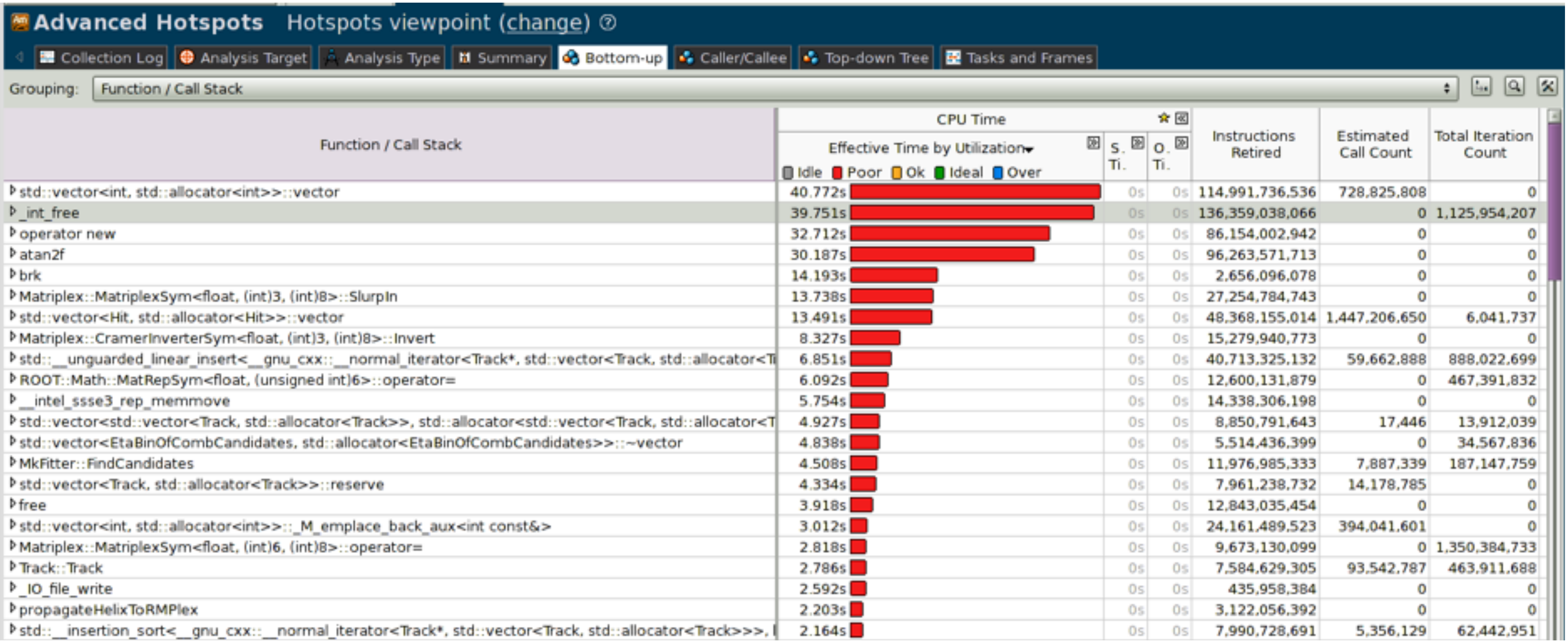


Rec Hits from CMSSW

- Switch to single event with 500 tracks, only one eta bin
  - select tracks without inefficiencies in CMSSW
  - hit smearing done in our code when starting from Sim Hits
- Hit finding for 10 GeV muons looks very good
  - 99.6% of the tracks have at least 9 found hits when starting from Sim Hits
  - 93% of the tracks have at least 9 found hits (95% at least 6) when starting from Rec Hits
- Worse performance for low p_T tracks, currently under investigation

Bottlenecks of single threaded running

Hotspots Analysis from VTune Intel profiler



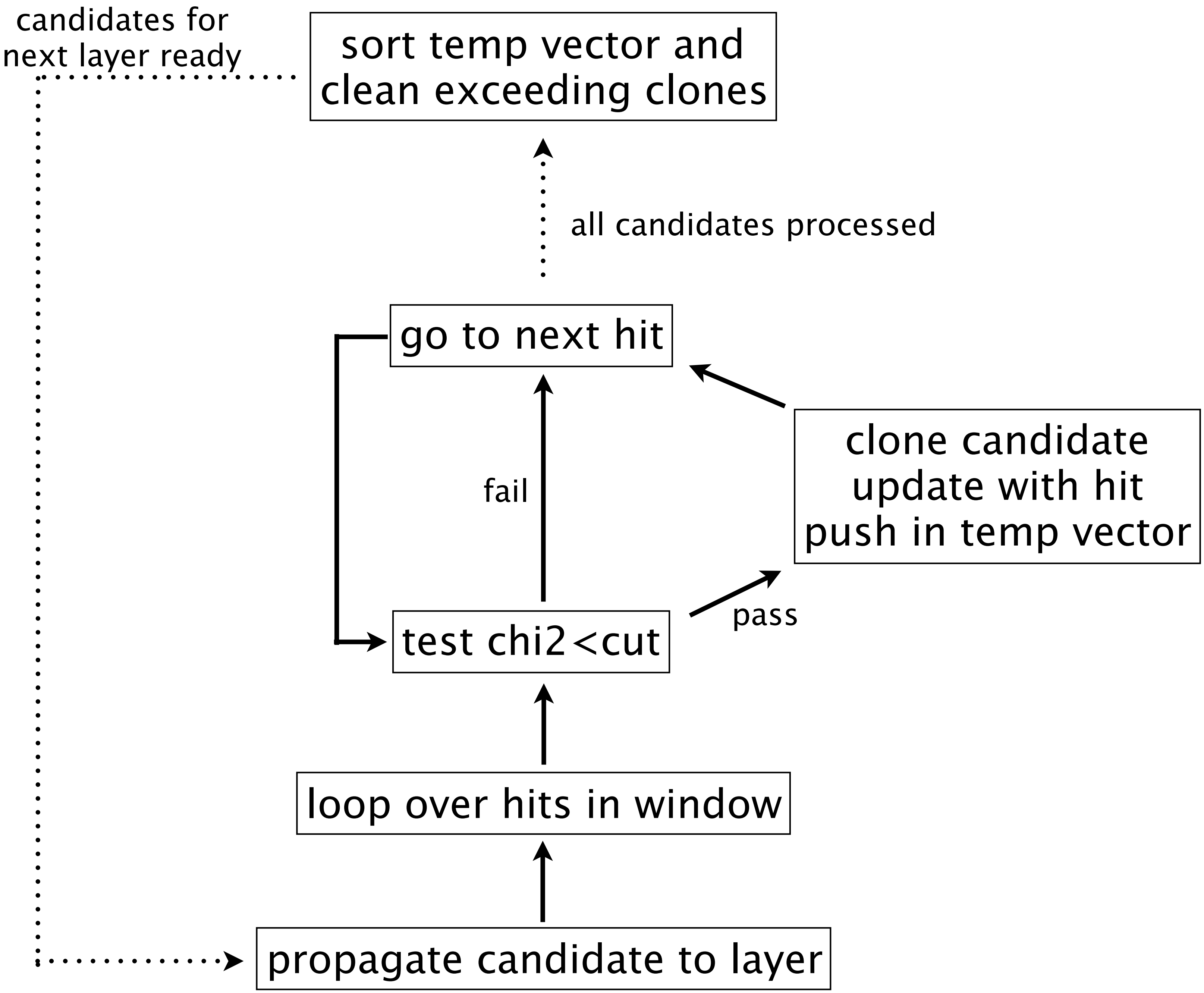
Grouping: Function / Call Stack

Function / Call Stack	CPU Time			Instructions Retired	Estimated Call Count	Total Iteration Count
	Effective Time by Utilization ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	S. Ti.	O. Ti.			
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

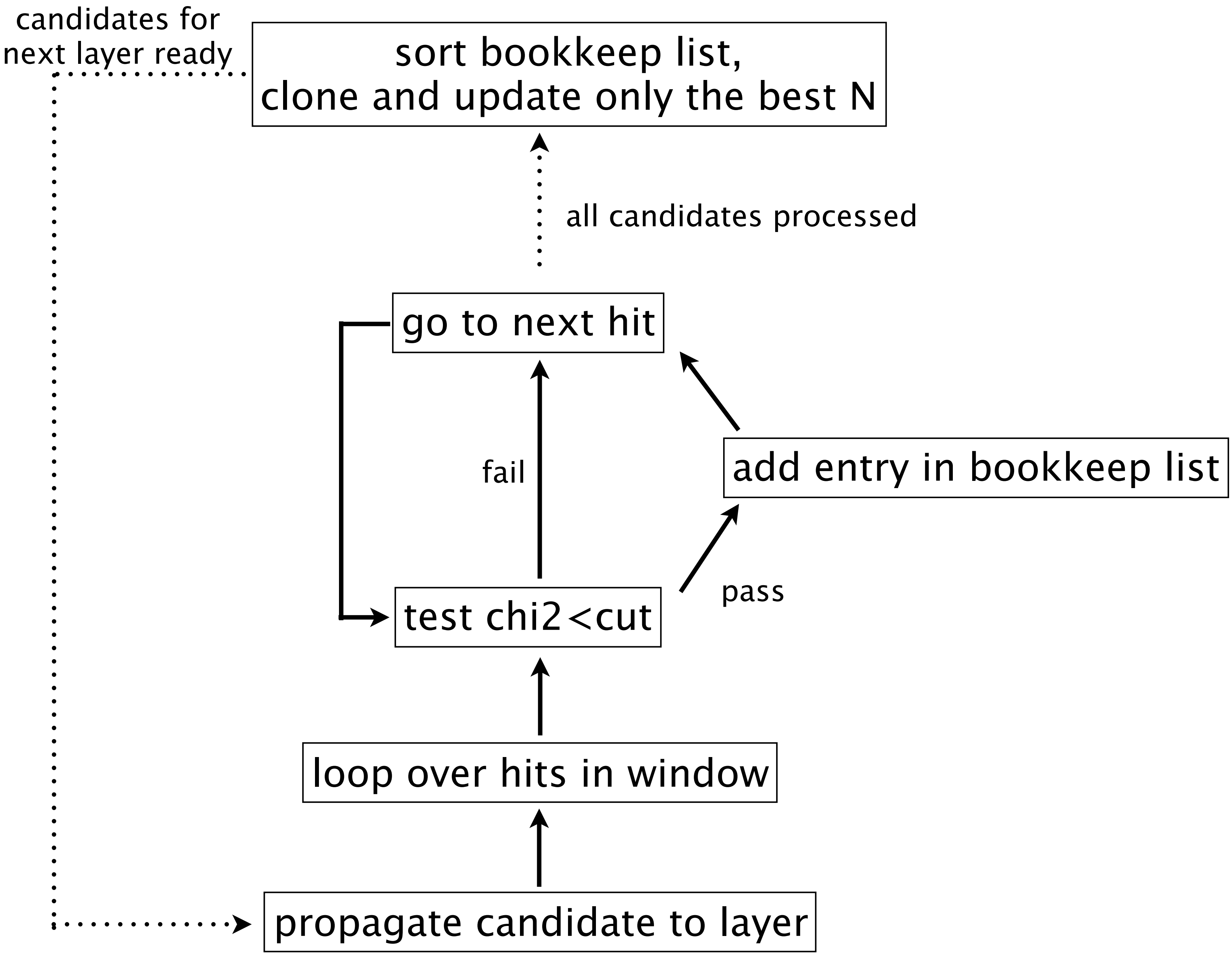
Leading functions are all memory operations!
Cloning of candidates and loading of hits in cache are the bottlenecks.

(note that atan2f is mainly in event preparation – not counted in timing tests)

Cloning: standard approach



Clone Engine approach



candidates for next layer ready

sort bookkeep list, clone and update only the best N

all candidates processed

go to next hit

fail

add entry in bookkeep list

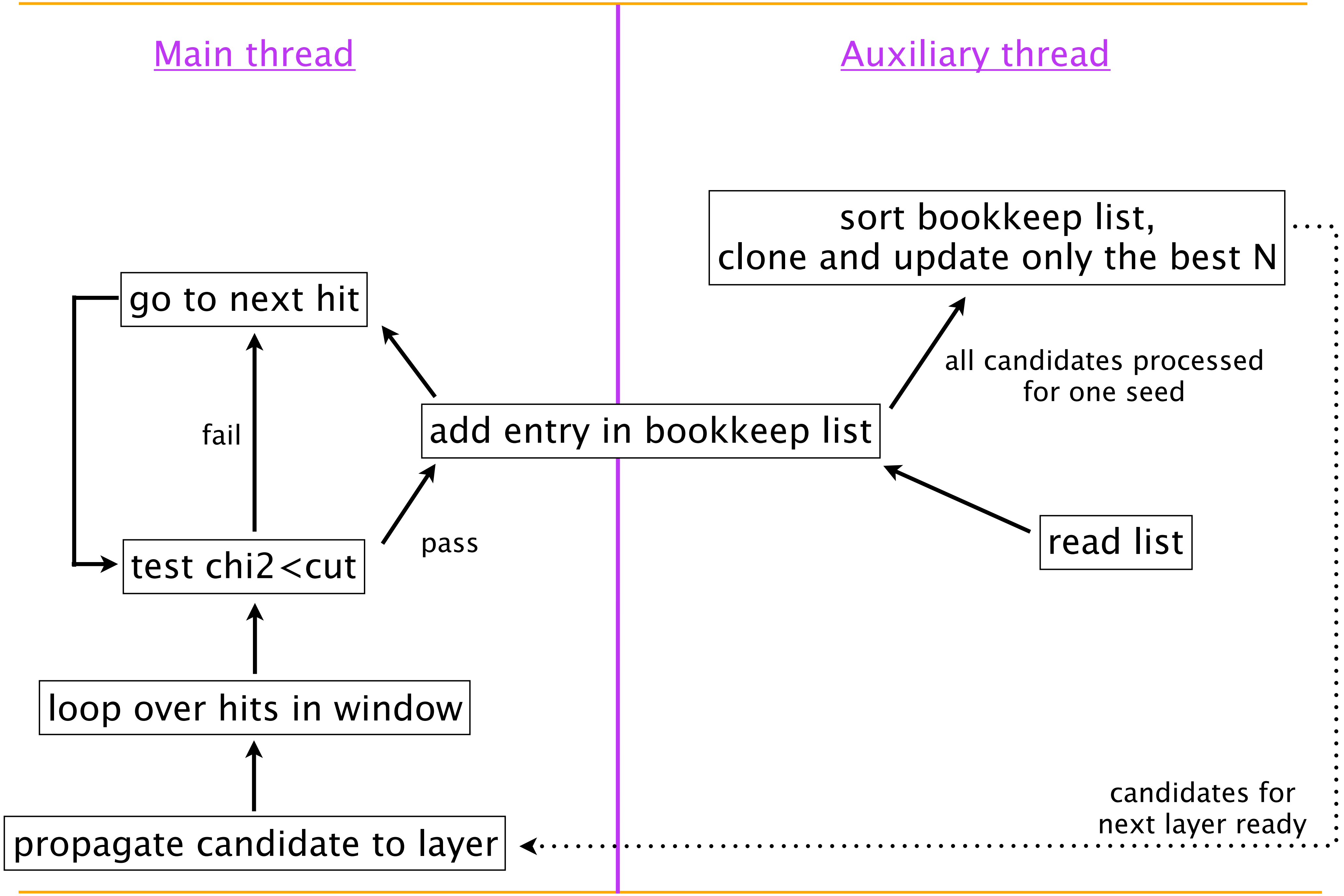
pass

test chi2<cut

loop over hits in window

propagate candidate to layer

Threaded cloning engine approach



Main thread

Auxiliary thread

sort bookkeep list,
clone and update only the best N

go to next hit

all candidates processed
for one seed

fail

add entry in bookkeep list

pass

read list

test chi2<cut

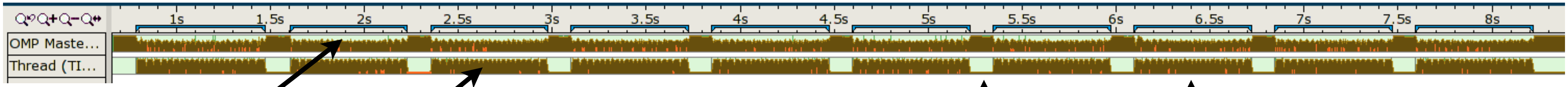
loop over hits in window

candidates for
next layer ready

propagate candidate to layer

Cloning Engine Performance

Cloning engine gives larger speedup from vectorization.
Threaded cloning engine gives significant speedup over serial cloning engine: 25–35%
(full utilization of parallel threads would be 50%).



main thread

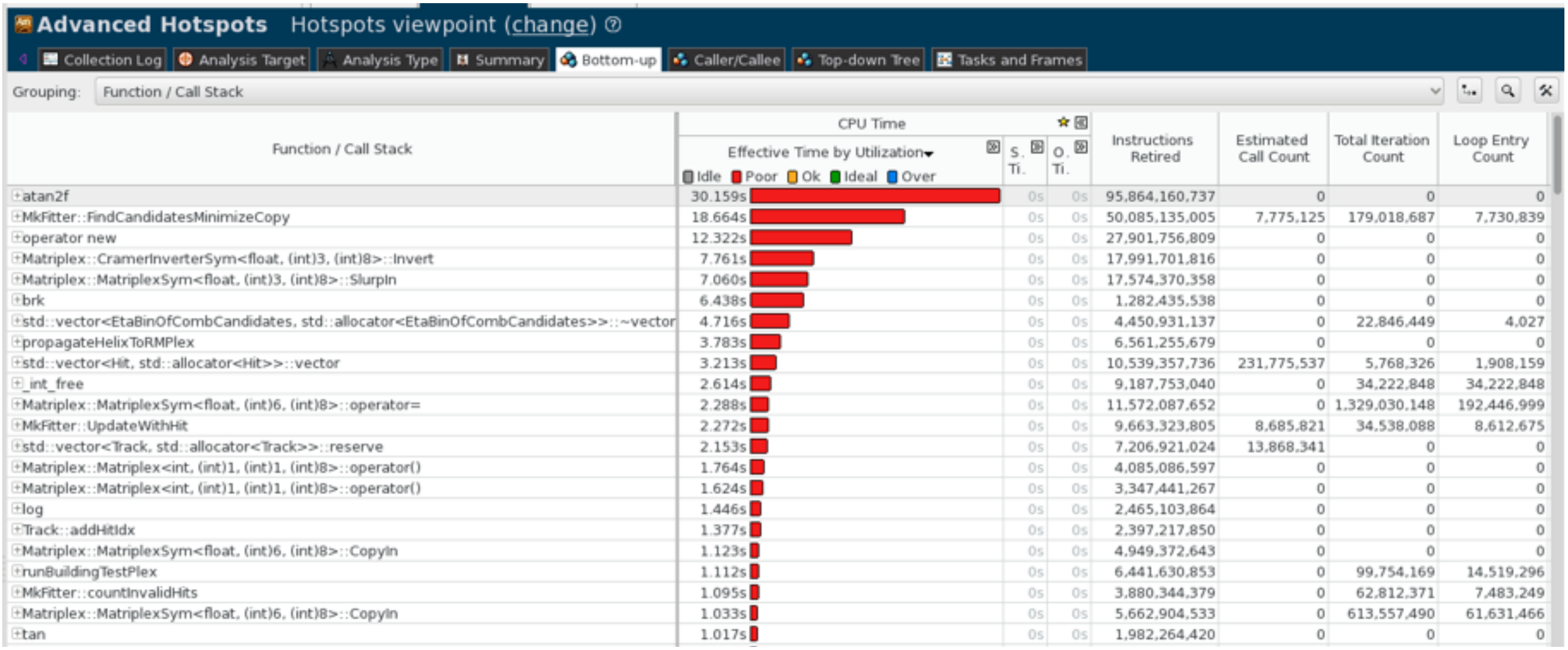
auxiliary thread

event preparation

event processing (repeated 10x)

Impact of cloning engine smaller when using reduced data formats,
but the two approaches are not exclusive.

VTune report after all memory improvements: many calculation functions now at the top!

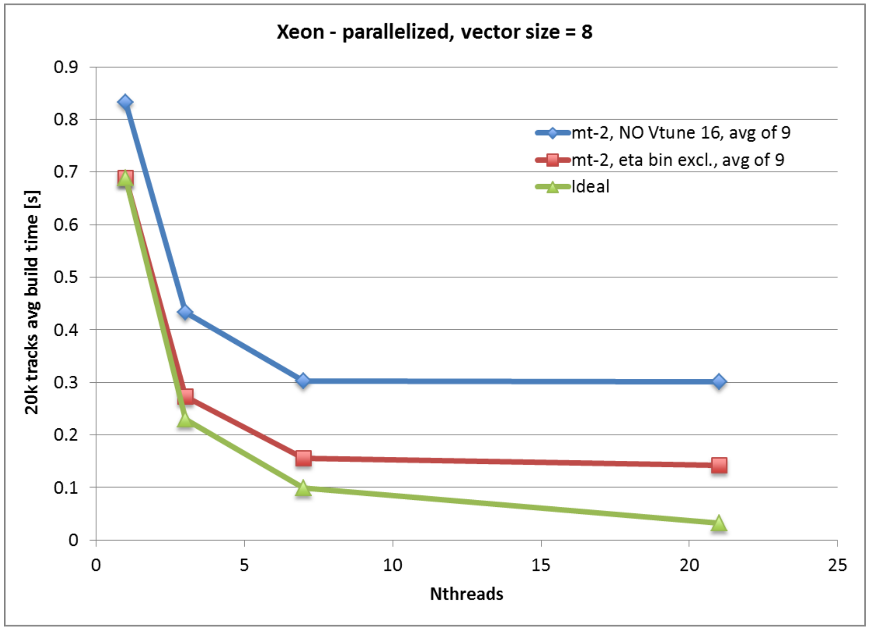


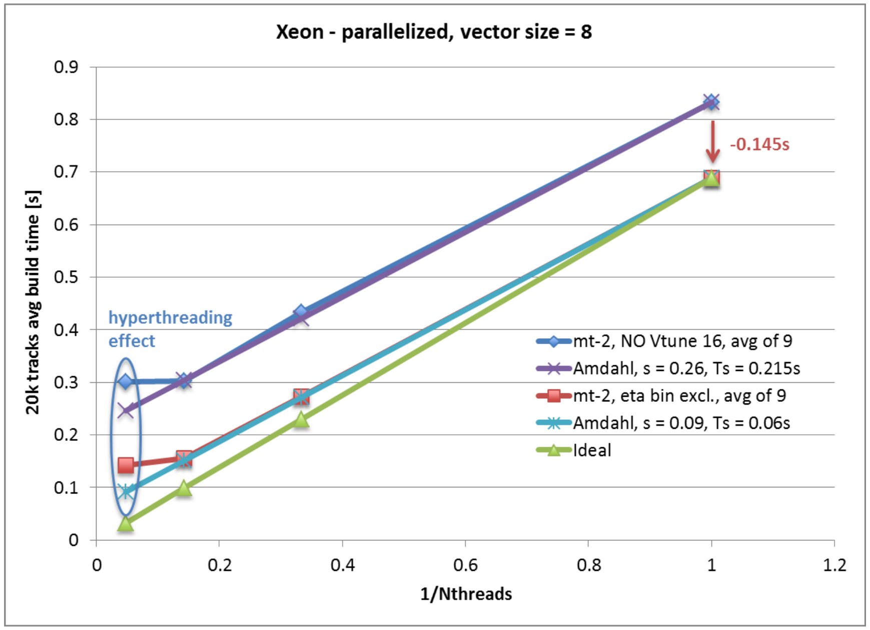
Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization ▨ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over	S. Ti.	O. Ti.	Instructions Retired	Estimated Call Count	Total Iteration Count	Loop Entry Count
atan2f	30.159s	0s	0s	95,864,160,737	0	0	0
MkFitter::FindCandidatesMinimizeCopy	18.664s	0s	0s	50,085,135,005	7,775,125	179,018,687	7,730,839
operator new	12.322s	0s	0s	27,901,756,809	0	0	0
Matriplex::CramerInverterSym<float, (int)3, (int)8>::Invert	7.761s	0s	0s	17,991,701,816	0	0	0
Matriplex::MatriplexSym<float, (int)3, (int)8>::SlurpIn	7.060s	0s	0s	17,574,370,358	0	0	0
brk	6.438s	0s	0s	1,282,435,538	0	0	0
std::vector<EtaBinOfCombCandidates, std::allocator<EtaBinOfCombCandidates>>::~vector	4.716s	0s	0s	4,450,931,137	0	22,846,449	4,027
propagateHelixToRMPlex	3.783s	0s	0s	6,561,255,679	0	0	0
std::vector<Hit, std::allocator<Hit>>::vector	3.213s	0s	0s	10,539,357,736	231,775,537	5,768,326	1,908,159
_int_free	2.614s	0s	0s	9,187,753,040	0	34,222,848	34,222,848
Matriplex::MatriplexSym<float, (int)6, (int)8>::operator=	2.288s	0s	0s	11,572,087,652	0	1,329,030,148	192,446,999
MkFitter::UpdateWithHit	2.272s	0s	0s	9,663,323,805	8,685,821	34,538,088	8,612,675
std::vector<Track, std::allocator<Track>>::reserve	2.153s	0s	0s	7,206,921,024	13,868,341	0	0
Matriplex::Matriplex<int, (int)1, (int)1, (int)8>::operator()	1.764s	0s	0s	4,085,086,597	0	0	0
Matriplex::Matriplex<int, (int)1, (int)1, (int)8>::operator()	1.624s	0s	0s	3,347,441,267	0	0	0
log	1.446s	0s	0s	2,465,103,864	0	0	0
Track::addHitIdx	1.377s	0s	0s	2,397,217,850	0	0	0
Matriplex::MatriplexSym<float, (int)6, (int)8>::CopyIn	1.123s	0s	0s	4,949,372,643	0	0	0
runBuildingTestPlex	1.112s	0s	0s	6,441,630,853	0	99,754,169	14,519,296
MkFitter::countInvalidHits	1.095s	0s	0s	3,880,344,379	0	62,812,371	7,483,249
Matriplex::MatriplexSym<float, (int)6, (int)8>::CopyIn	1.033s	0s	0s	5,662,904,533	0	613,557,490	61,631,466
tan	1.017s	0s	0s	1,982,264,420	0	0	0

Understanding parallelization issues

- Previous results on Xeon consistent with a serial workload of ~25% of T1 execution
  - Fit to Amdahl's Law: T = T1 * (0.74/Nthreads + 0.26)
- Largest contribution coming from re-instantiation of a data structure at each event
- Replacing deletion/creation with simple reset gave large improvement
  - Amdahl still fits: T = T1 * (0.91/Nthreads + 0.09)





- Significant residual contribution to non-ideal scaling is due to non-uniformity of occupancy within threads, i.e. some threads take longer than others
  - clear limitation of distributing the thread work among eta bins
  - also eta bins are problematics in case of a beam spot with large longitudinal width
- Work ongoing to define strategies for an efficient 'next in line' approach or a dynamic reallocation of thread resources

Summary of recent improvements

- Large speedup even in serial case
  - contributions from many fixes at 10-50% speedup level
- Vectorization is improved from reduction of impact of memory operations:
  - better data structures
  - cloning engine
- Parallelization is improved by identifying and fixing the serial code

- Further ideas mostly related to reduce the imbalance for threads and vector units
  - plus improvements for the cloning engine for parallelization: use it with hyper-threading?

- Overall achieved a good understanding of the Xeon (Phi) features with our standalone/simplified setup: how far are we from reality?