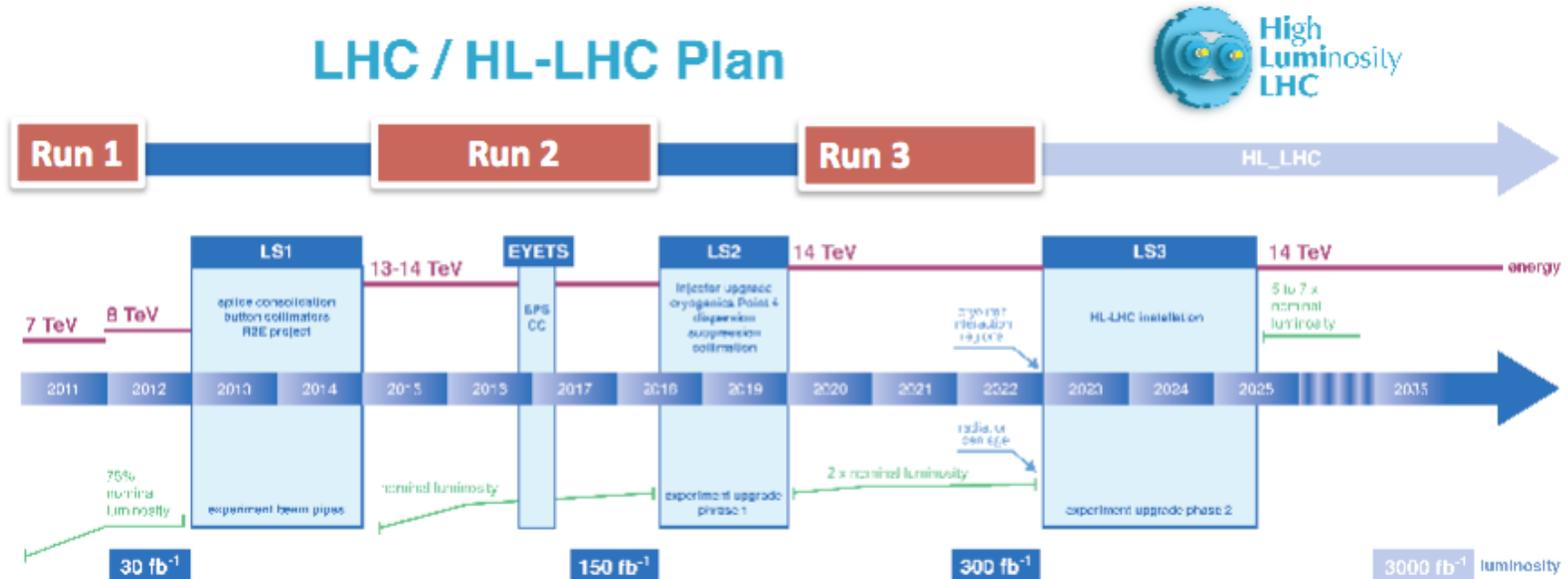


Kalman Filter Tracking on Parallel Architectures

Computing Techniques Seminar
FNAL – Jan. 13, 2016

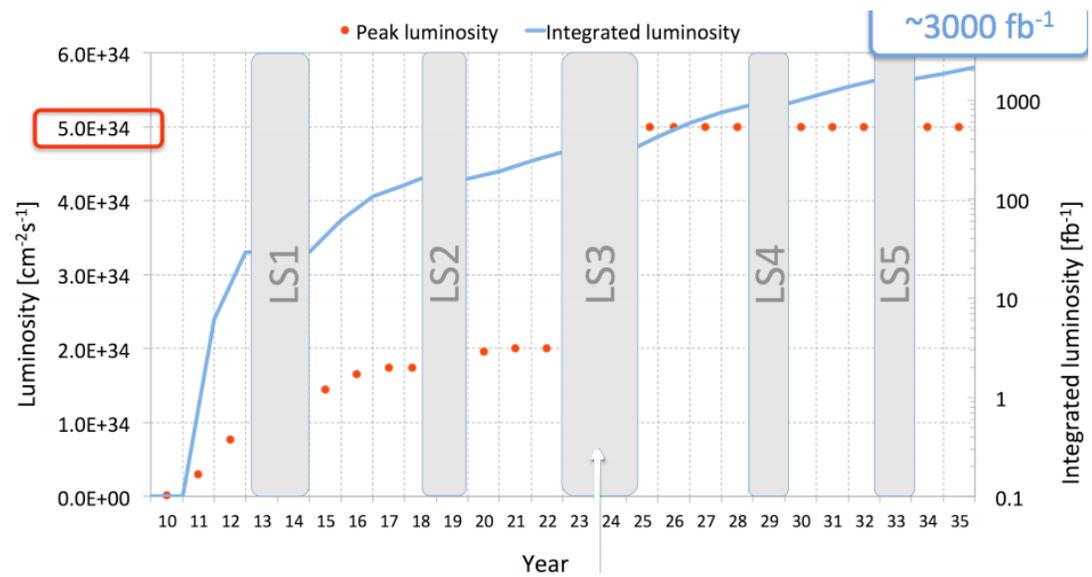
G.Cerati (UCSD)

- 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 (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

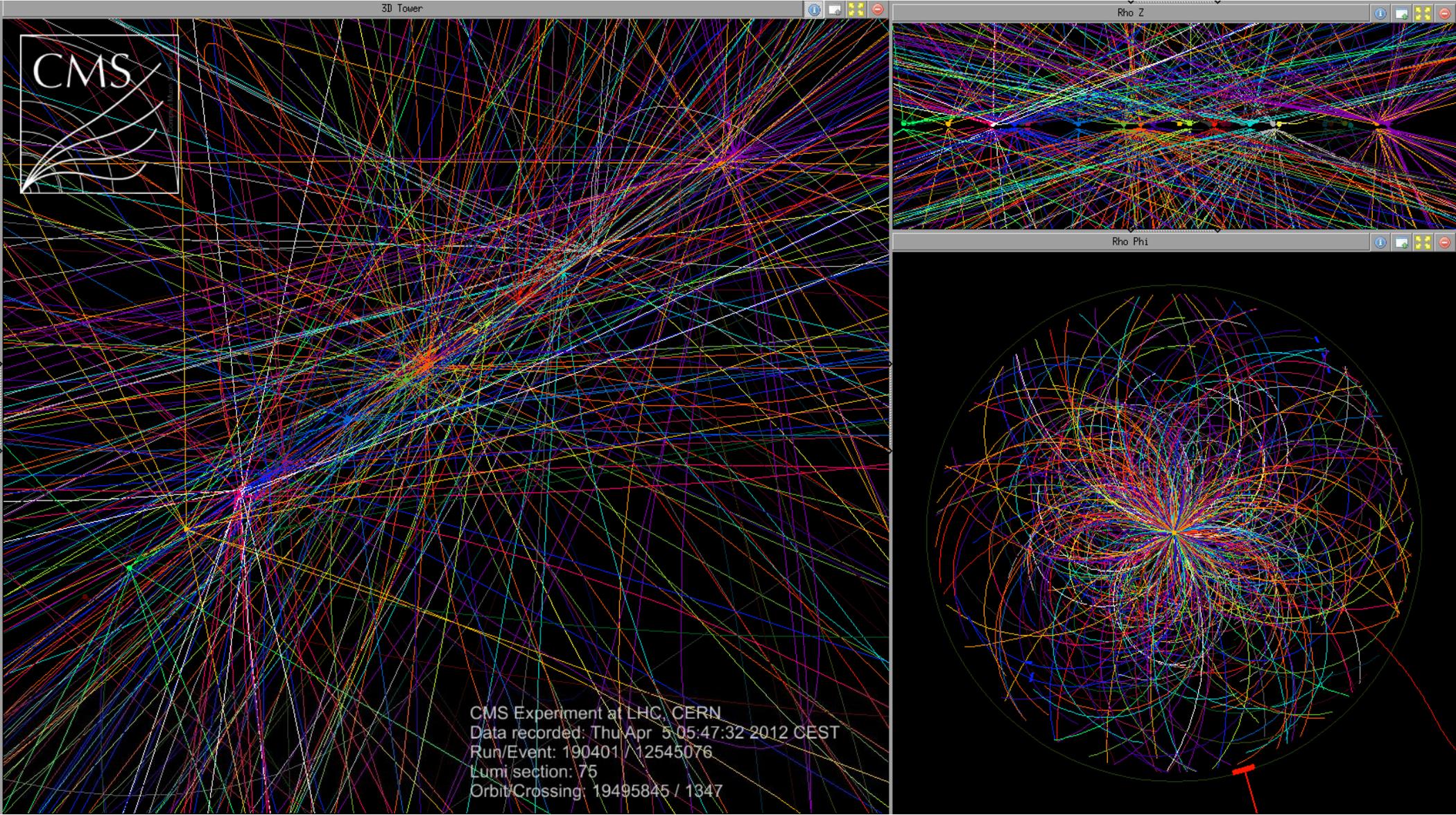


LHC is now close to its maximum collision energy. Need as much data as possible to see evidence of extremely rare processes accessible at this energy.

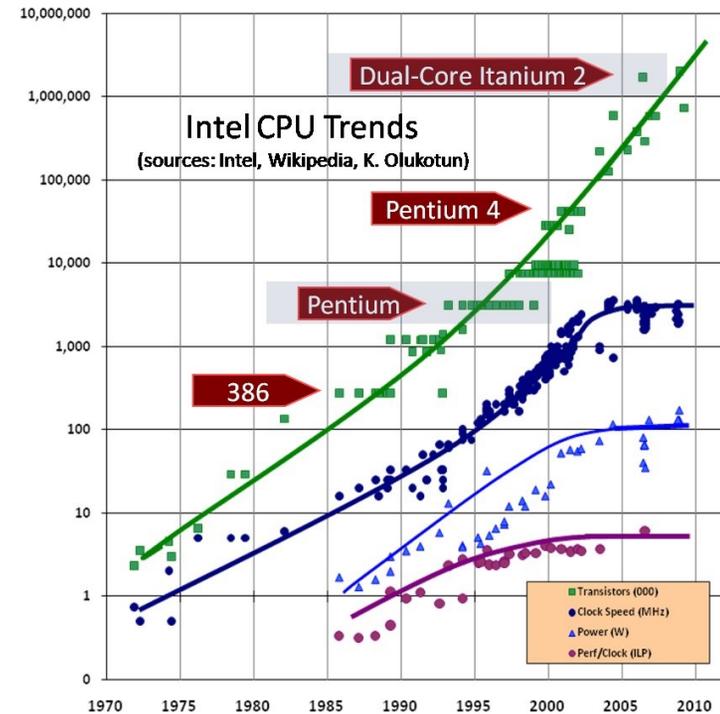
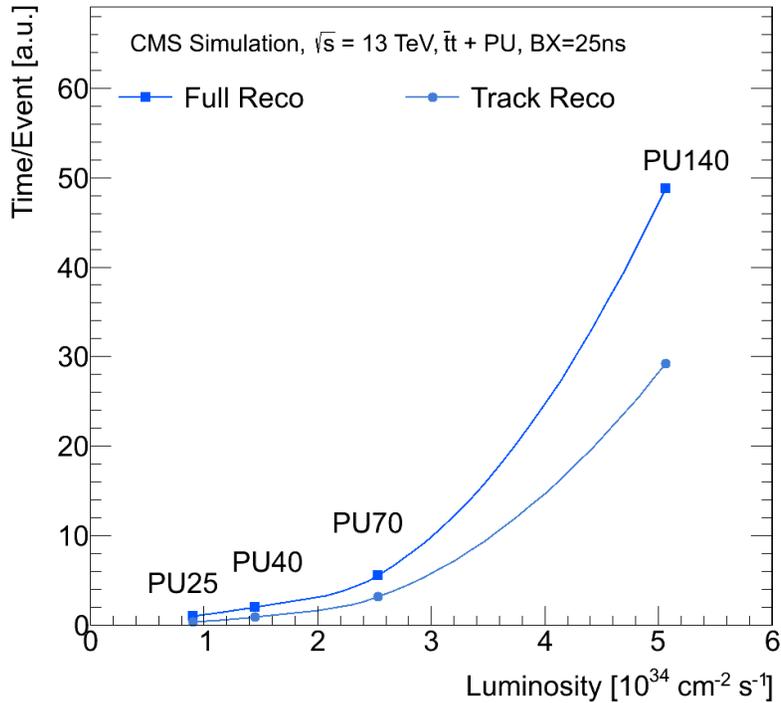
LHC is planning a smooth increase in instantaneous luminosity until the end of Run3. Then, the HL-LHC is planned with a luminosity of 5E34.



High Lumi means High Pile-up

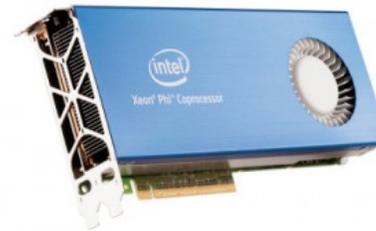


Reconstruction time vs PU



- 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 <u>SMX</u>
Logical Cores	16 (<u>HT</u>)	240 (<u>HT</u>)	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

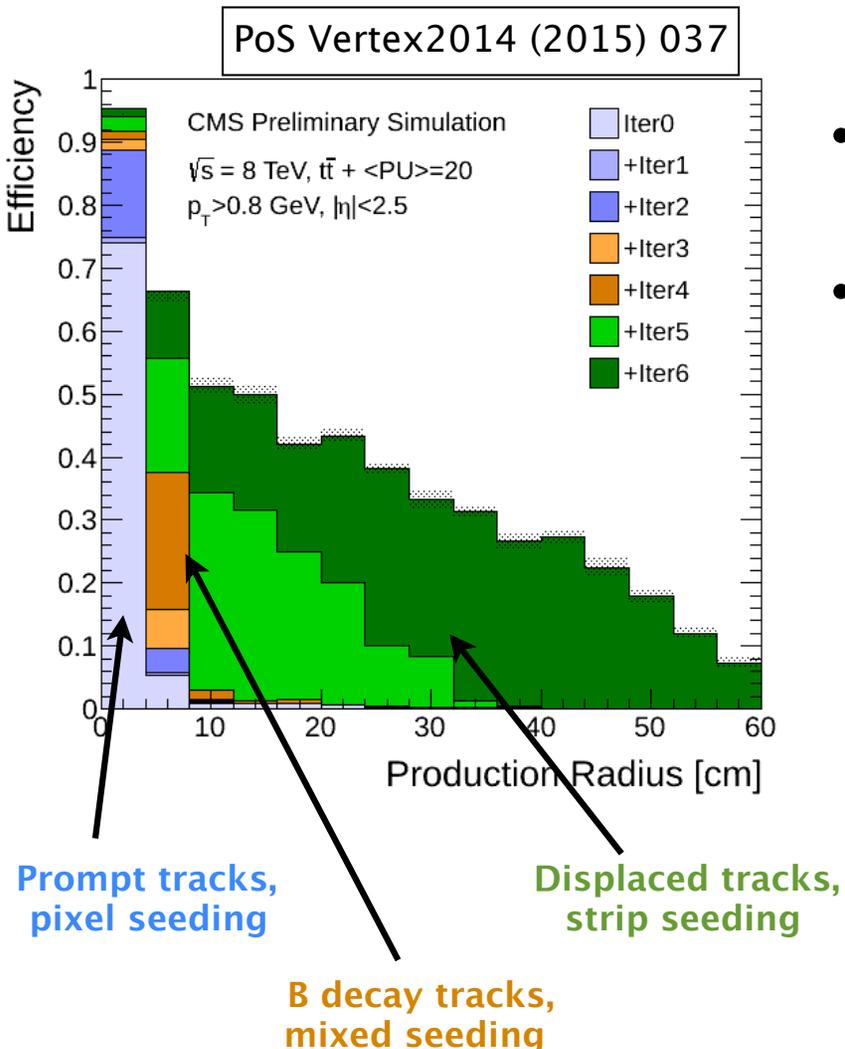
Many-core architectures have lower clock frequency and lower memory than traditional multi-cores, but feature SIMD units and a large number of cores for a much larger nominal throughput

- 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

- 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 may also be valid for GPU
- Convenient choice given large investment for next-generation supercomputers based on Xeon Phi

Why Kalman Filter?

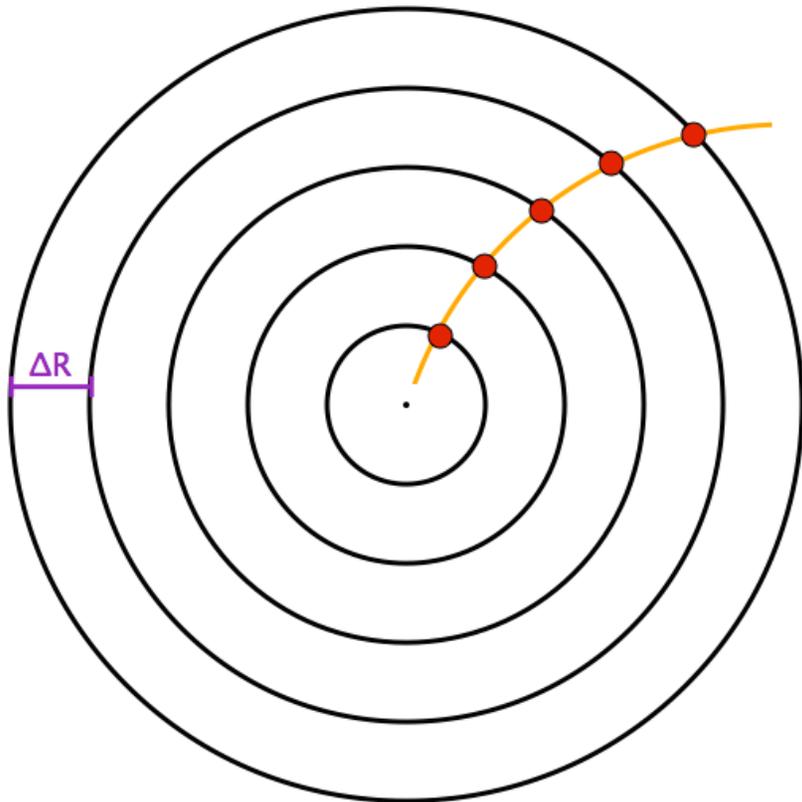
Kalman filter tracking commonly used in HEP collision experiments.
 Robust treatment of material effects, so it is particularly suitable for silicon tracker.
 Outstanding performance at the LHC, e.g. CMS.



- 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 $R=50 \text{ cm}$
- ⇒ inclusive tracking for all collision vertices



Simplified, standalone tracking code:

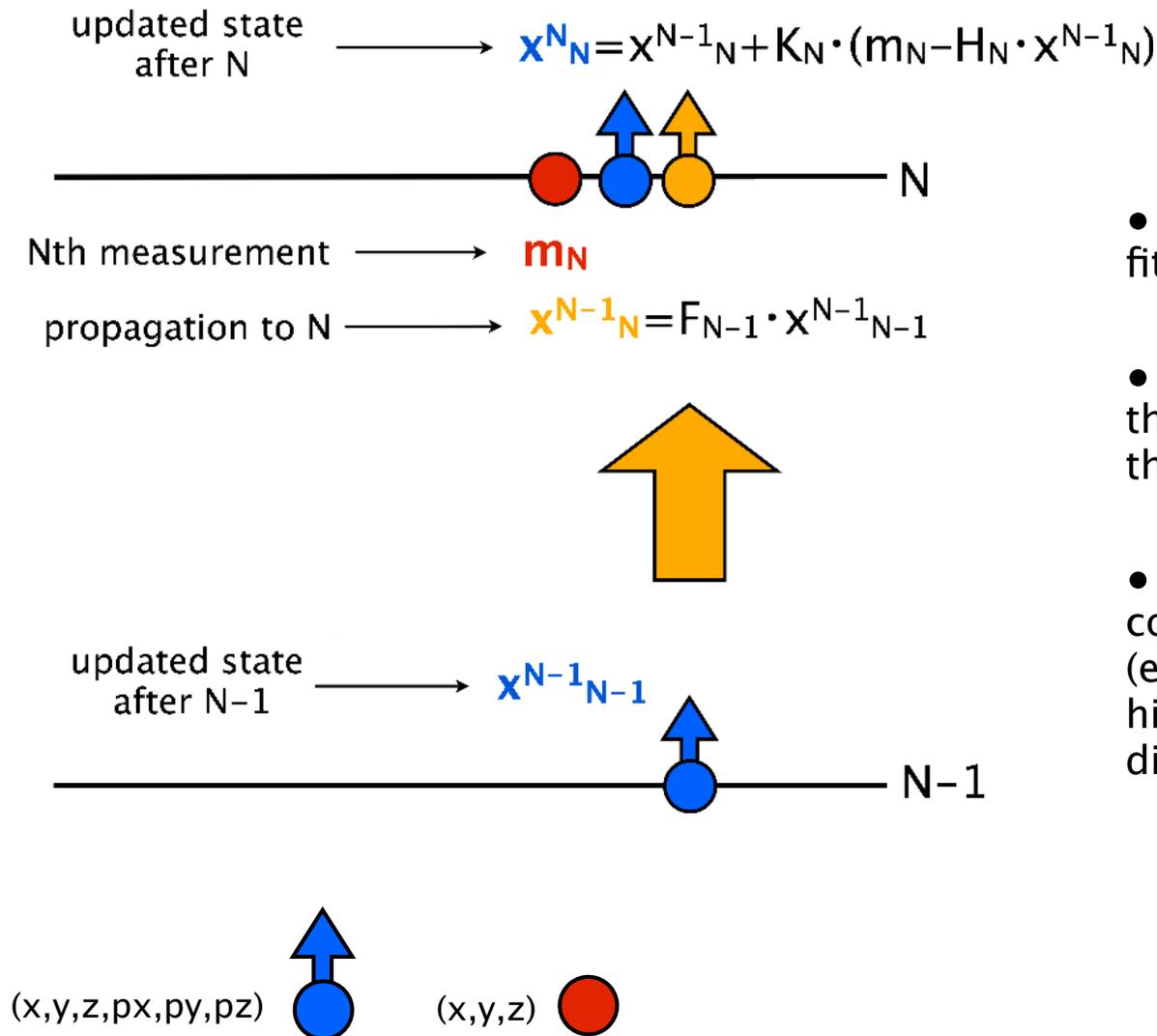
- Tracker with 10 barrel layers, perfectly cylindrical, separated by $\Delta R=4$ cm
- Longitudinal bounds: $|\eta|<1$
- 3.8 T magnetic field, coaxial with the tracker
- Assume beam spot width 1mm in xy and 1cm in z
- Hit resolution 100 μ m in r-phi, 1mm in z
- No material, no inefficiencies
- Work in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



- 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

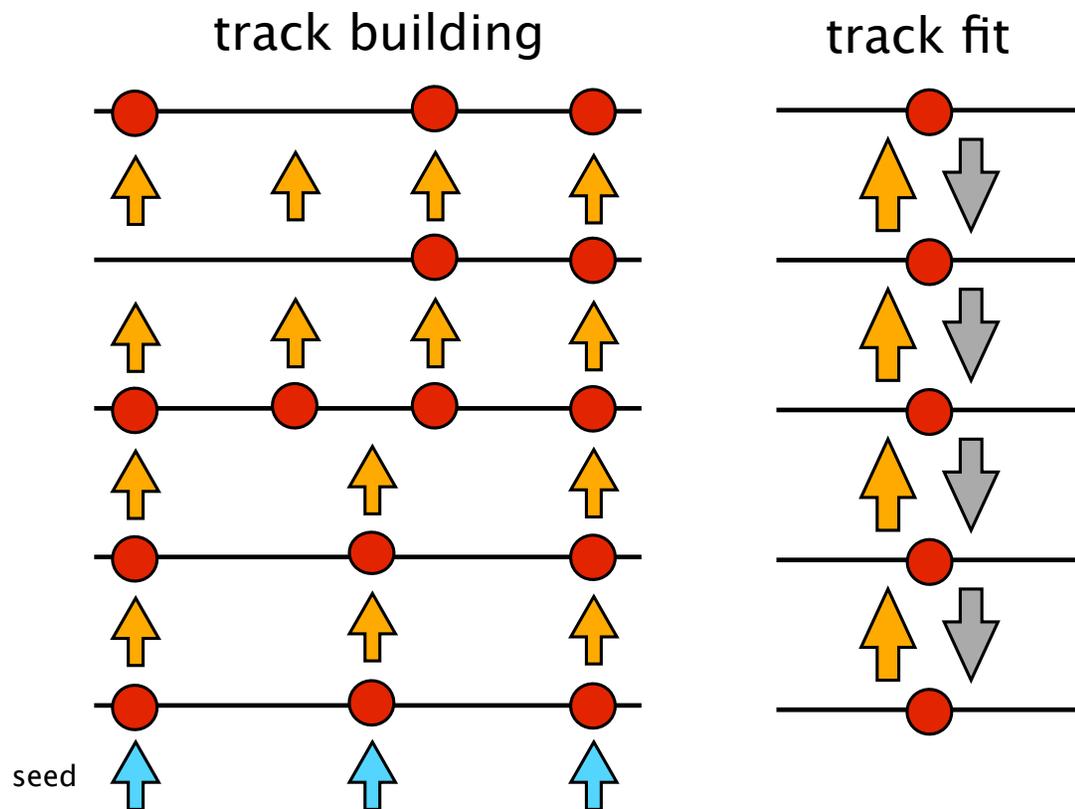
The Kalman Filter track reconstruction searches for hits along the track direction, with a search window that shrinks when more measurements are added.

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**.

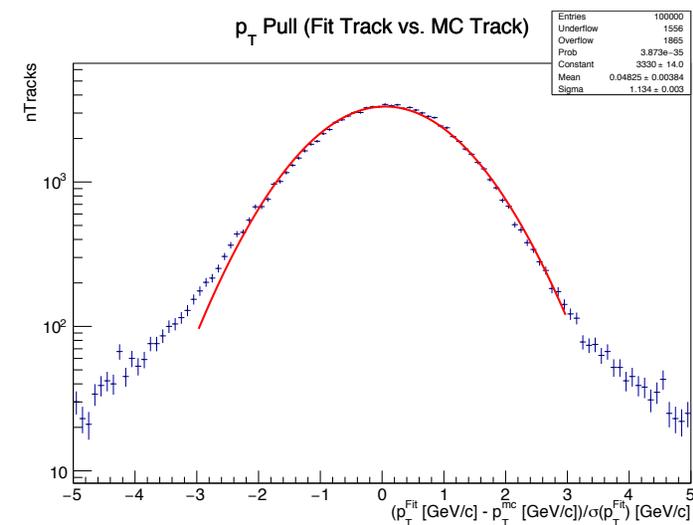
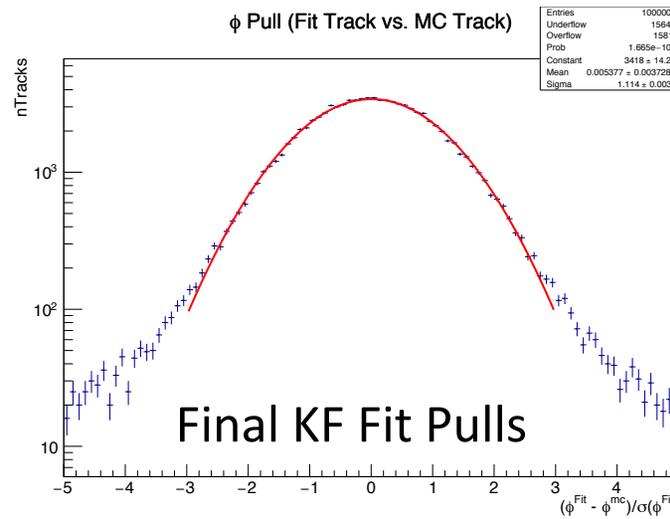
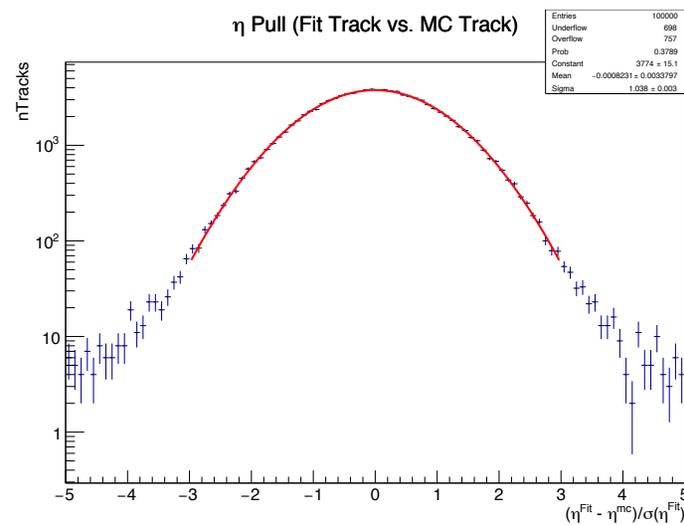
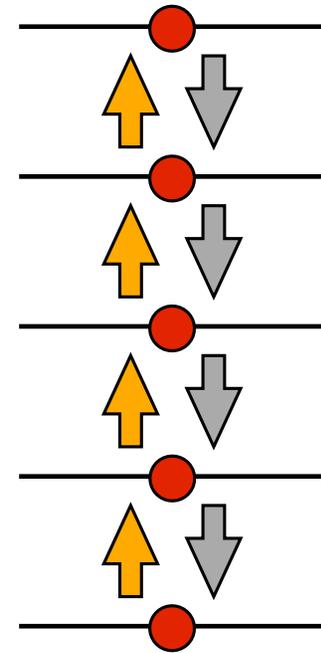
Track **building is the most time consuming part** – it involves branching points of variable size, with the simplest version degenerating into the track fit case.

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



- 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

- 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 \sim unitary width pulls and p_T resolution $0.5 \times p_T$ [%]

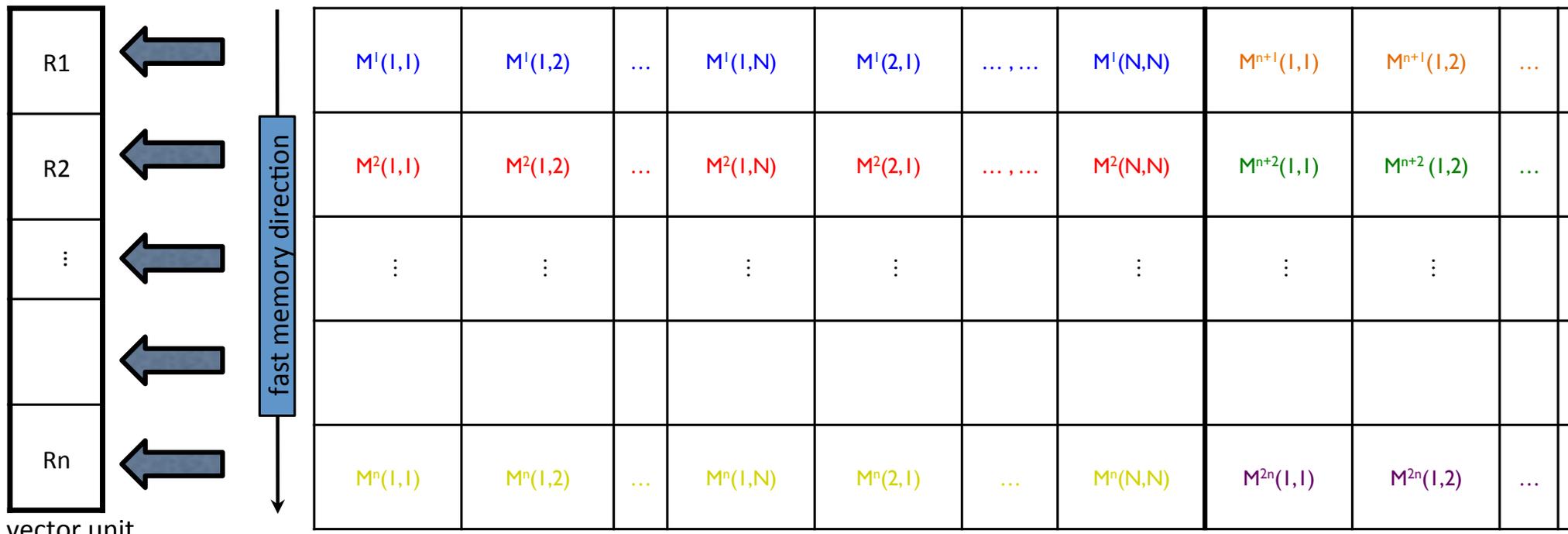


Matriplex

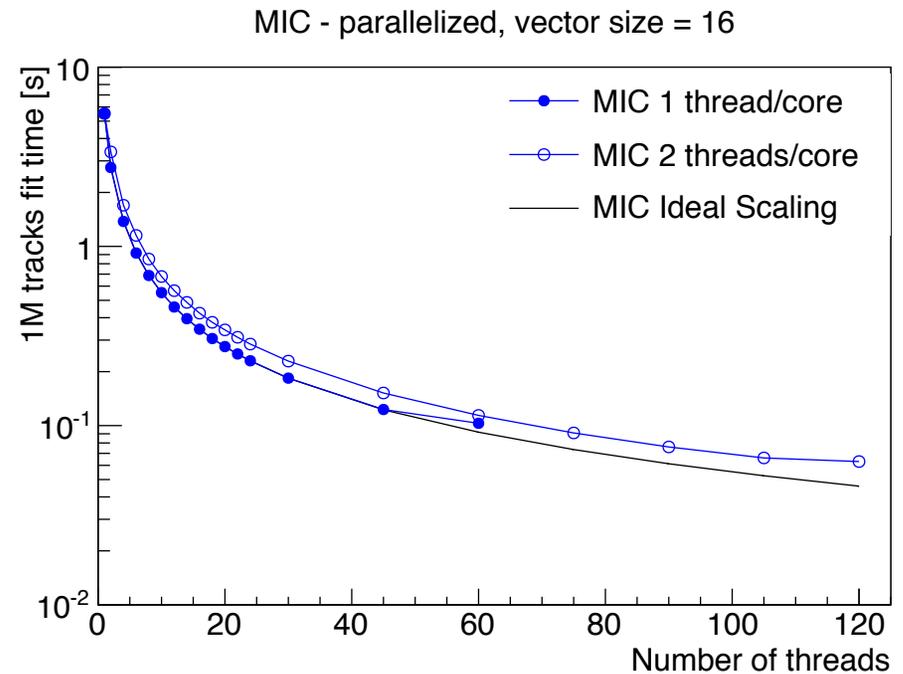
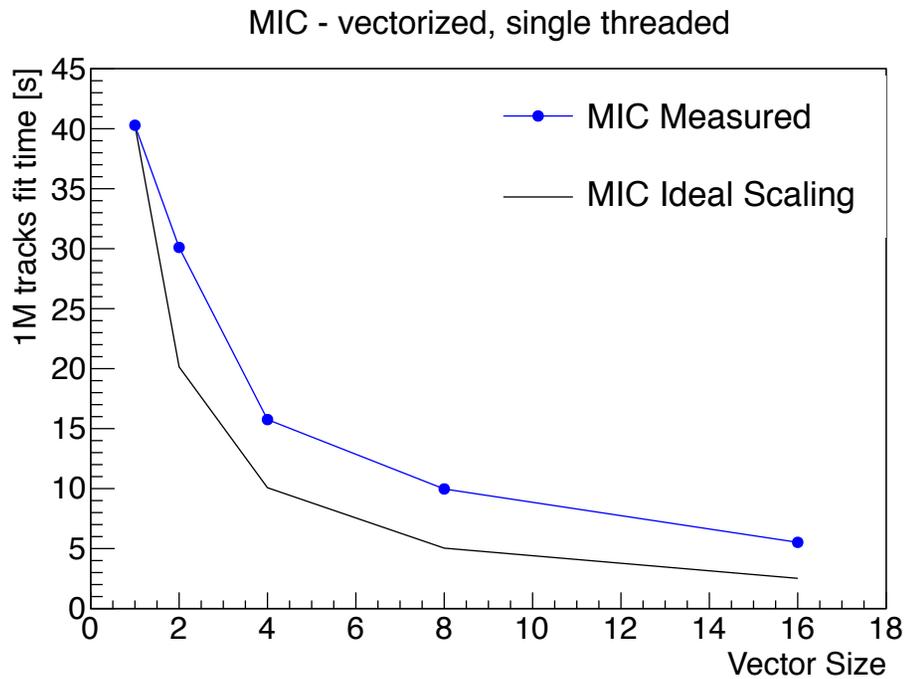
Kalman filter calculations based on small matrices.
 Intel Xeon and Xeon Phi have **vector units** with size 8 and 16 floats respectively.
 How can we efficiently exploit them?

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)



Matrix size $N \times N$, vector unit size n



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**.

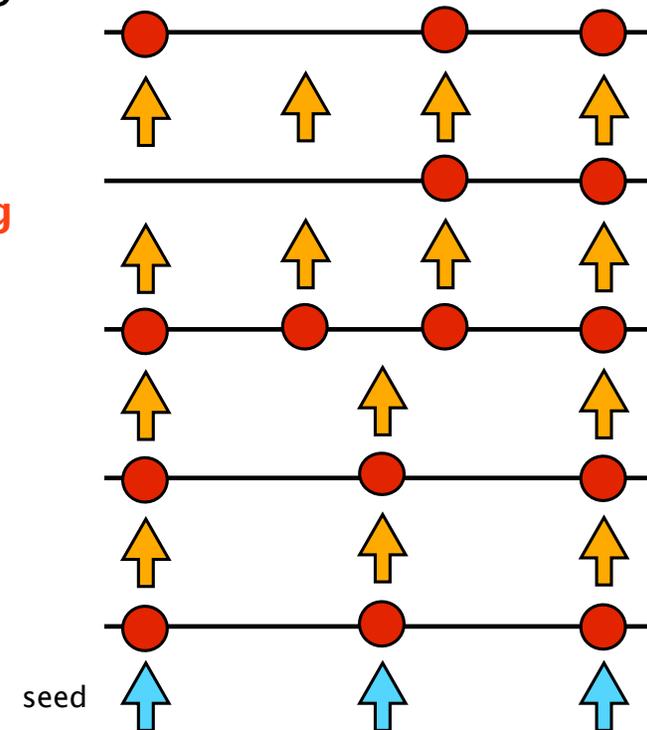
Effective performance of vectorization is about 50% utilization efficiency.

Parallelization performance is close to ideal in case of 1 thread/core, while with 2 threads/core an overhead is observed.

Both problems related to L1 cache issues.

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

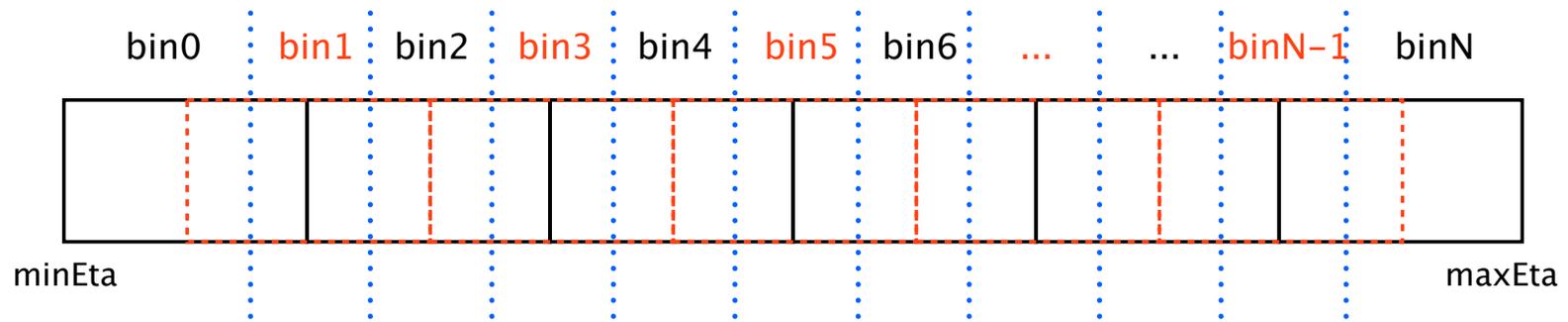
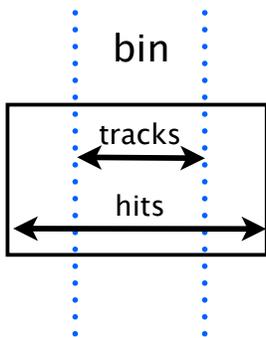
- Same core calculations as in track fitting but adding two big **complications**
 - **Hit set is not defined**: hit on next layer to be chosen between $O(10k)$ hits
 - For >1 compatible hit, combinatorial problem requires **cloning of candidates**
- The two issues can be **factorized** by dividing the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



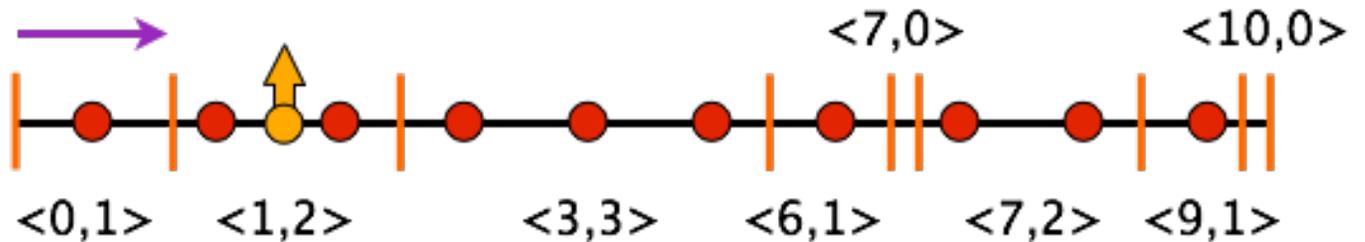
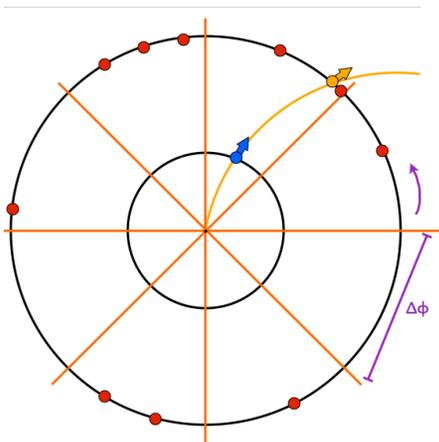
Space Partitioning for Track Building

- **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

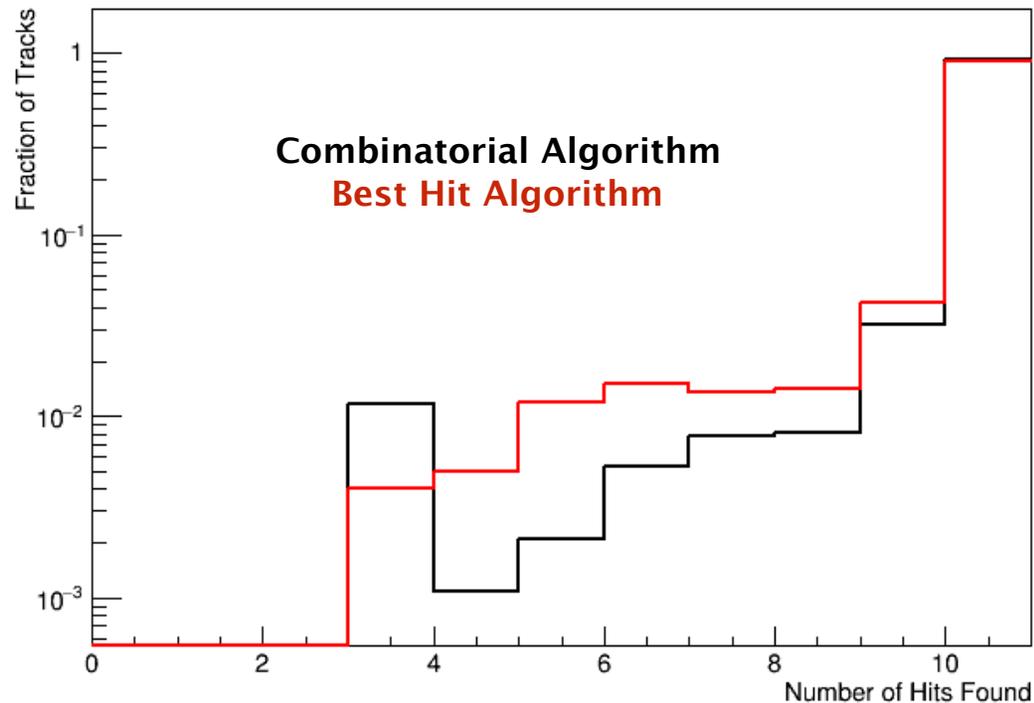
eta partitions:



phi partitions:

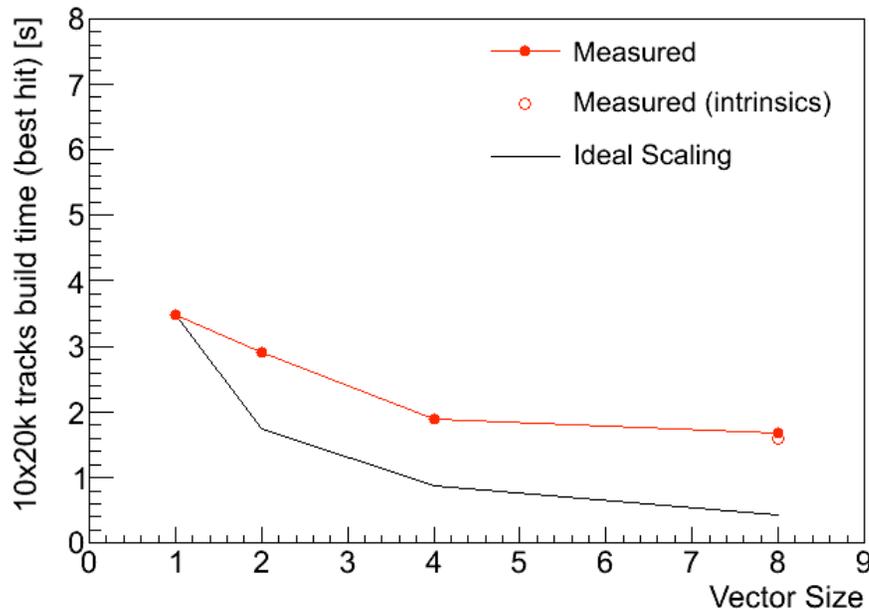


Hit Finding Performance

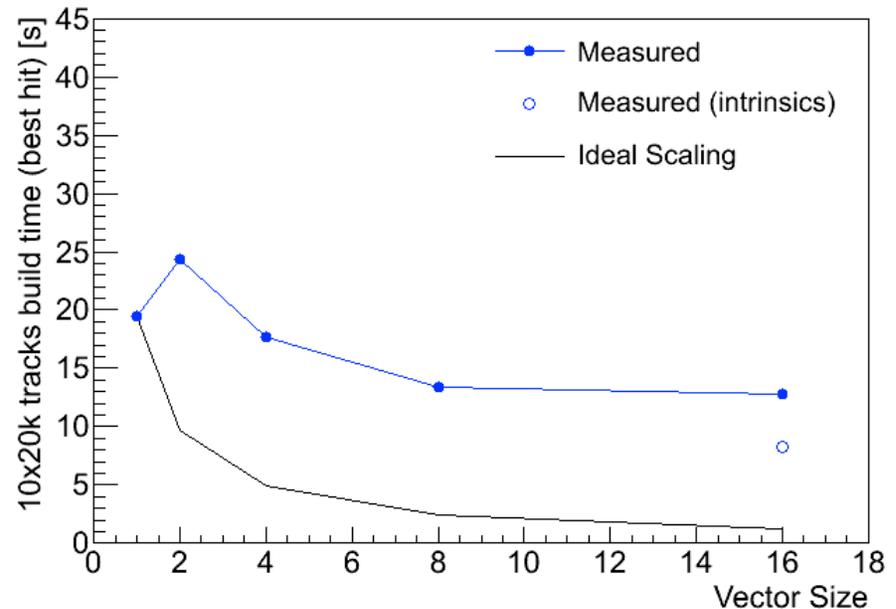


- Simulate events with 20k tracks per event
 - ▶ reconstruct using seeds taken from MC truth
- Combinatorial Algorithm: 96% (99%) of tracks found with $\geq 90\%$ (60%) of the hits.
- BestHit version: 94% (98%) of tracks found with $\geq 90\%$ (60%) of the hits.
 - ▶ BestHit algorithm not expected to behave so well with fully realistic setup

Xeon - vectorized, single threaded

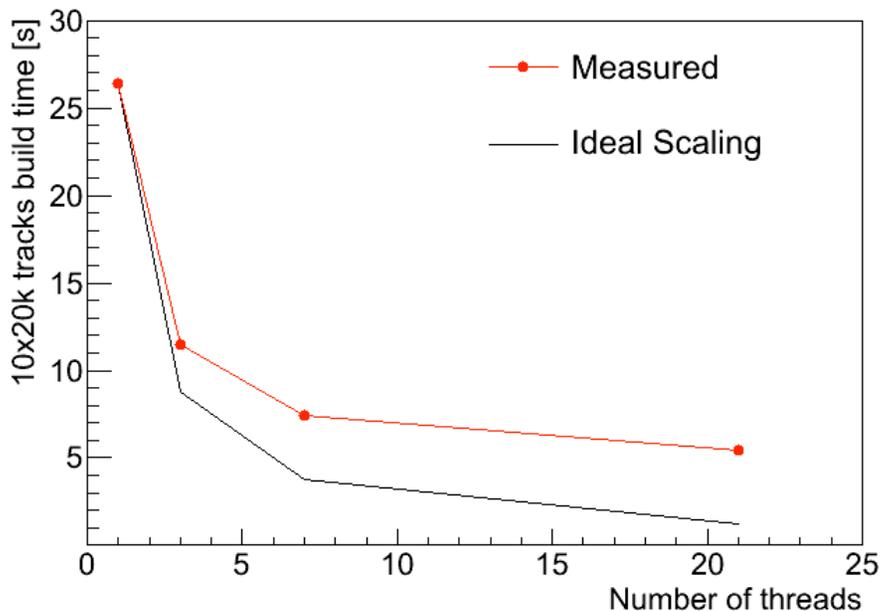


Xeon Phi - vectorized, single threaded

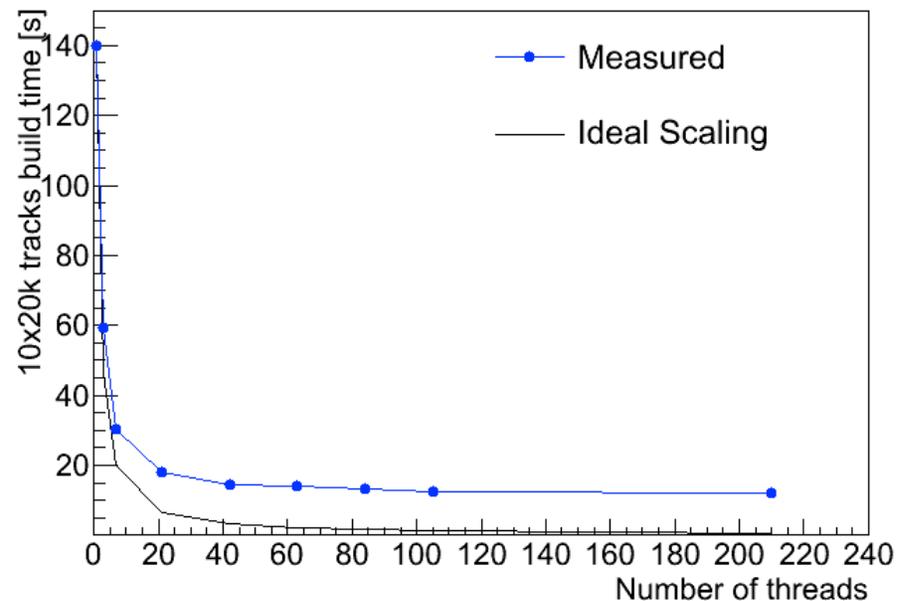


- Run **simplified track building** (best hit) on 10 events with 20k tracks each
 - ▶ pick hit in compatibility window with lowest chi2 at each layer
- Already much **more difficult than fitting case**, expect worse results:
 - ▶ test multiple (non pre-determined) hits per track
 - compatibility window and hits to process are not fully defined until propagation to layer
- 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
 - further **gain from using prefetching and gathering** intrinsics, but data input still takes large fraction of time!

Xeon - parallelized, vector size = 8



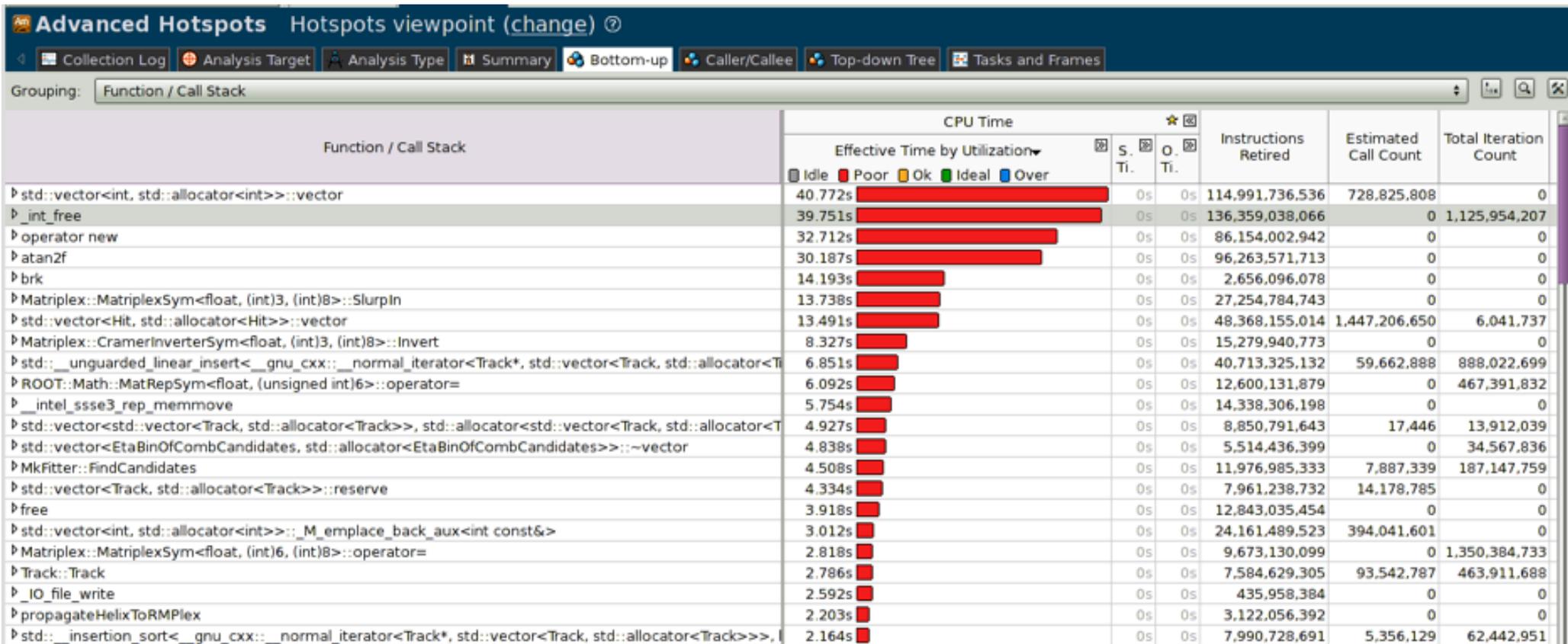
Xeon Phi - parallelized, vector size = 16 (int.)



- Run **full track building** with combinatorial expansion of candidates
 - ▶ ultimate physics performance, slower
- 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 **~5x on Xeon and >10x Xeon Phi**
 - ▶ speedup saturates above nThreads=42

Bottlenecks of single threaded running

Hotspots Analysis from VTune Intel profiler



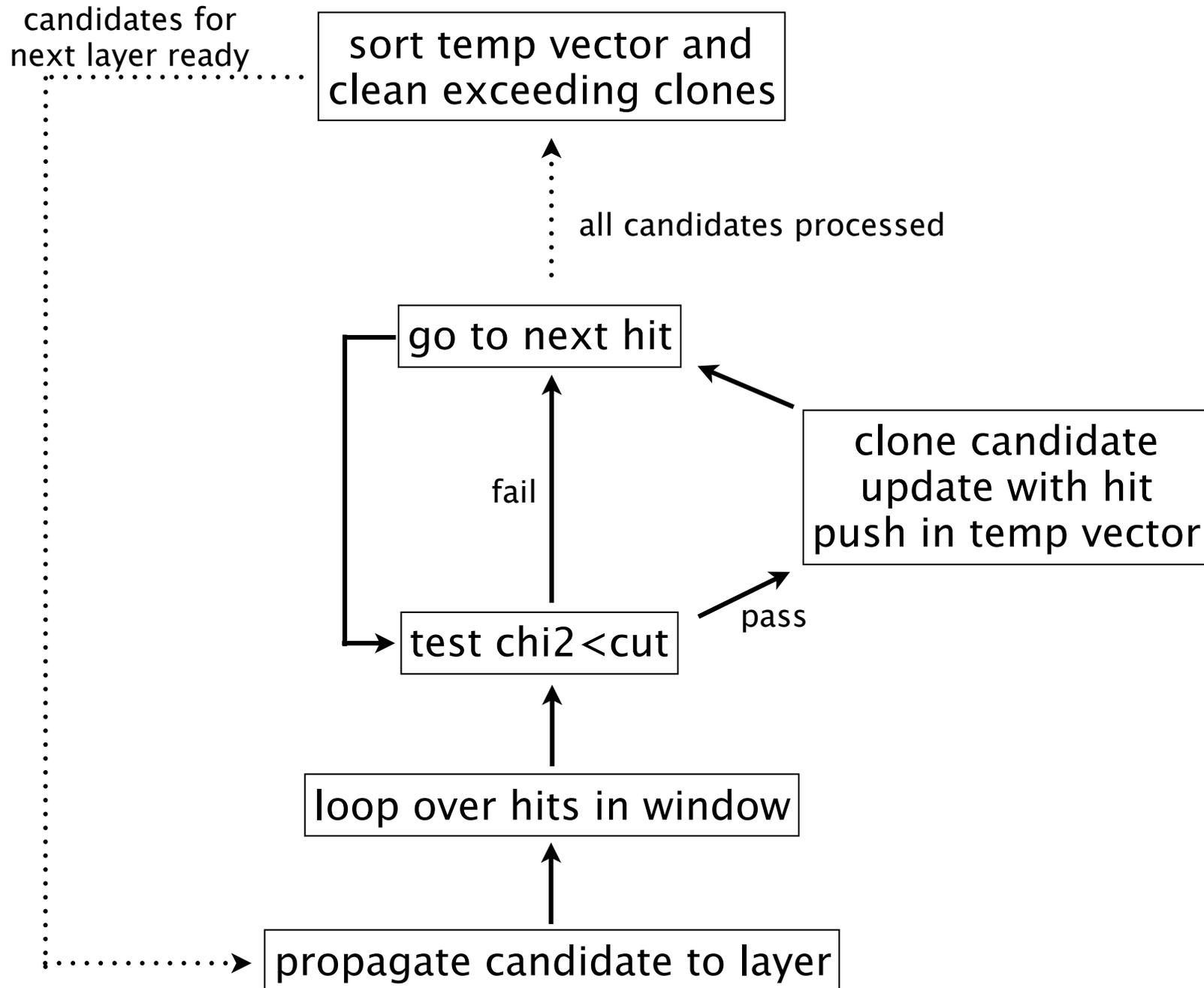
Function / Call Stack	CPU Time		Instructions Retired	Estimated Call Count	Total Iteration Count
	Effective Time by Utilization	S. Ti.			
std::vector<int, std::allocator<int>>::vector	40.772s	0s	114,991,736,536	728,825,808	0
_int_free	39.751s	0s	136,359,038,066	0	1,125,954,207
operator new	32.712s	0s	86,154,002,942	0	0
atan2f	30.187s	0s	96,263,571,713	0	0
brk	14.193s	0s	2,656,096,078	0	0
Matriplex::MatriplexSym<float, (int)3, (int)8>::SlurpIn	13.738s	0s	27,254,784,743	0	0
std::vector<Hit, std::allocator<Hit>>::vector	13.491s	0s	48,368,155,014	1,447,206,650	6,041,737
Matriplex::CramerInverterSym<float, (int)3, (int)8>::Invert	8.327s	0s	15,279,940,773	0	0
std::_unguarded_linear_insert<_gnu_cxx::_normal_iterator<Track*, std::vector<Track, std::allocator<Track>>::iterator>, std::vector<Track, std::allocator<Track>>::iterator>	6.851s	0s	40,713,325,132	59,662,888	888,022,699
ROOT::Math::MatRepSym<float, (unsigned int)6>::operator=	6.092s	0s	12,600,131,879	0	467,391,832
_intel_ssse3_rep_memmove	5.754s	0s	14,338,306,198	0	0
std::vector<std::vector<Track, std::allocator<Track>>, std::allocator<std::vector<Track, std::allocator<Track>>>	4.927s	0s	8,850,791,643	17,446	13,912,039
std::vector<EtaBinOfCombCandidates, std::allocator<EtaBinOfCombCandidates>>::~vector	4.838s	0s	5,514,436,399	0	34,567,836
MkFitter::FindCandidates	4.508s	0s	11,976,985,333	7,887,339	187,147,759
std::vector<Track, std::allocator<Track>>::reserve	4.334s	0s	7,961,238,732	14,178,785	0
free	3.918s	0s	12,843,035,454	0	0
std::vector<int, std::allocator<int>>::_M_emplace_back_aux<int const&>	3.012s	0s	24,161,489,523	394,041,601	0
Matriplex::MatriplexSym<float, (int)6, (int)8>::operator=	2.818s	0s	9,673,130,099	0	1,350,384,733
Track::Track	2.786s	0s	7,584,629,305	93,542,787	463,911,688
_IO_file_write	2.592s	0s	435,958,384	0	0
propagateHelixToRMPlex	2.203s	0s	3,122,056,392	0	0
std::_insertion_sort<_gnu_cxx::_normal_iterator<Track*, std::vector<Track, std::allocator<Track>>::iterator>	2.164s	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.

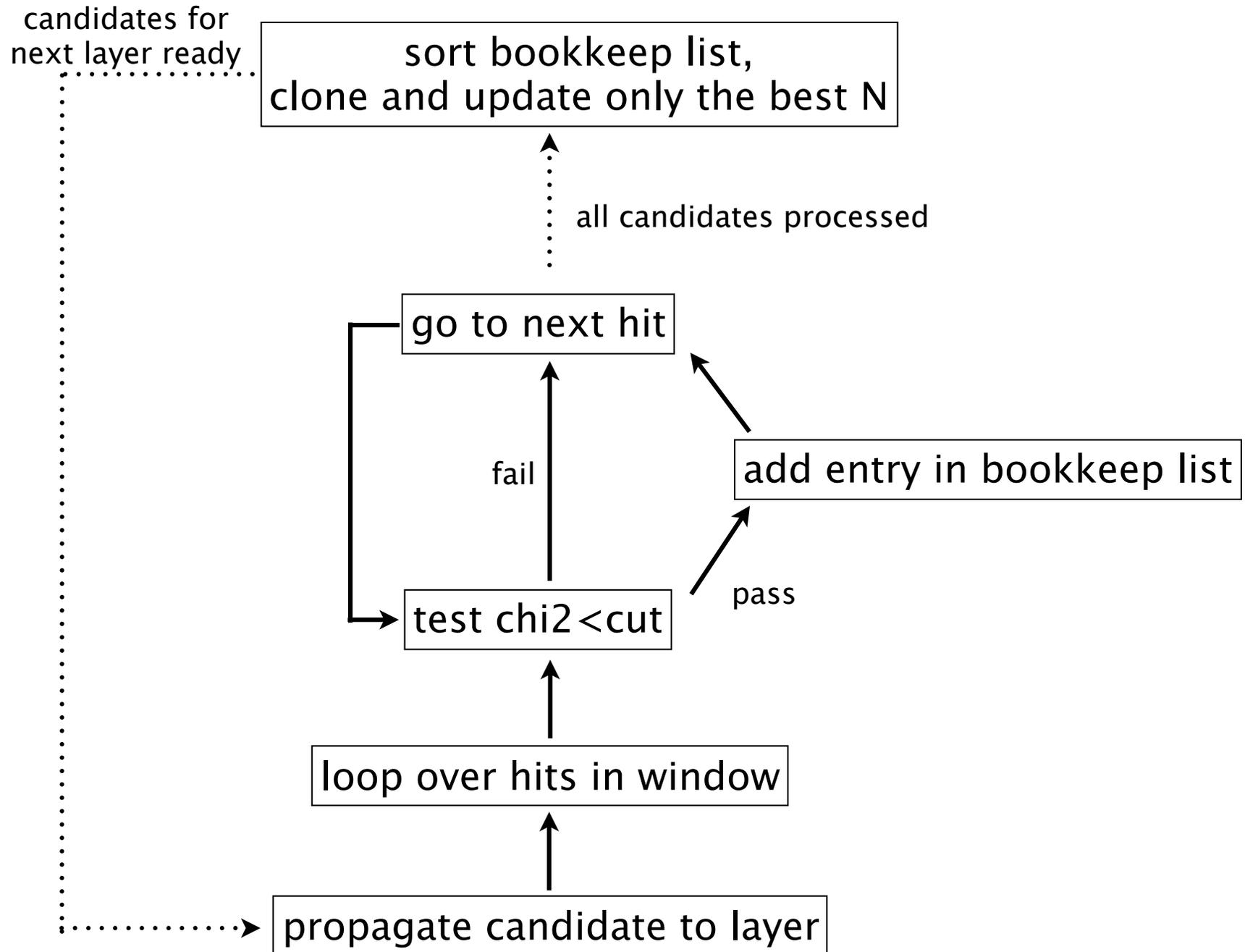
(note that atan2f is mainly in event preparation – not counted in timing tests)

- First, avoid resizing of hit indices vector in track object: get 45% speedup
 - reserve did not help, move from `std::vector` to fixed size array
- Then, reduce size of data formats to minimum
 - Size of Hit and Track objects is crucial since they have heaviest impact on memory
- Current versions carry data members that are not necessarily needed
 - MC truth information, copy of hit vector
 - parameters and errors stored as SMatrix objects which are heavier than just the array of floats
- Reduce size of Track by 20% and size of Hit by 40%: get additional 30% speedup
 - improve also vectorization speedup by ~20%
- At the same time, we tested a new strategy to reduce the impact of memory operations: the cloning engine
 - basically the idea is to avoid/minimize memory operations in vectorized loops
 - delegate them to the cloning engine

Cloning: previous approach



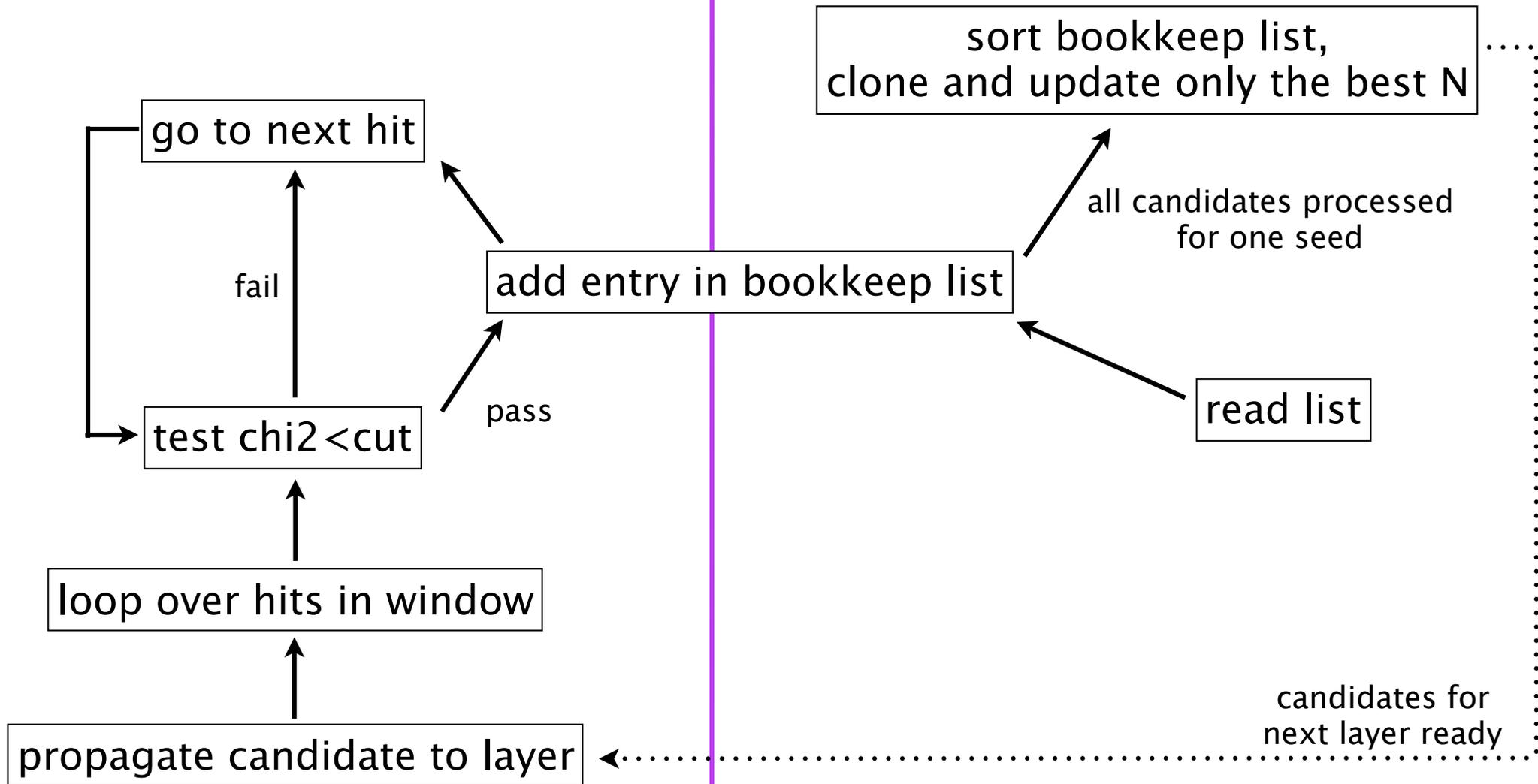
Cloning engine approach



Threaded cloning engine approach

Main thread

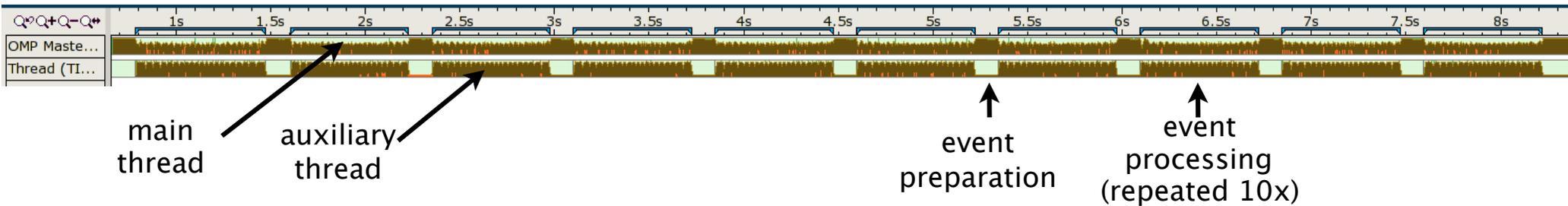
Auxiliary thread



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%).



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

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

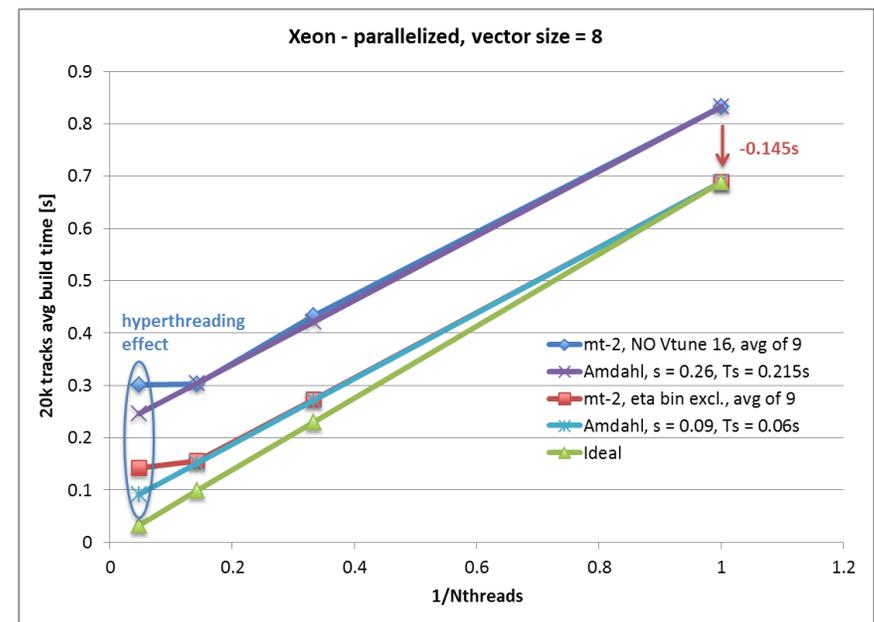
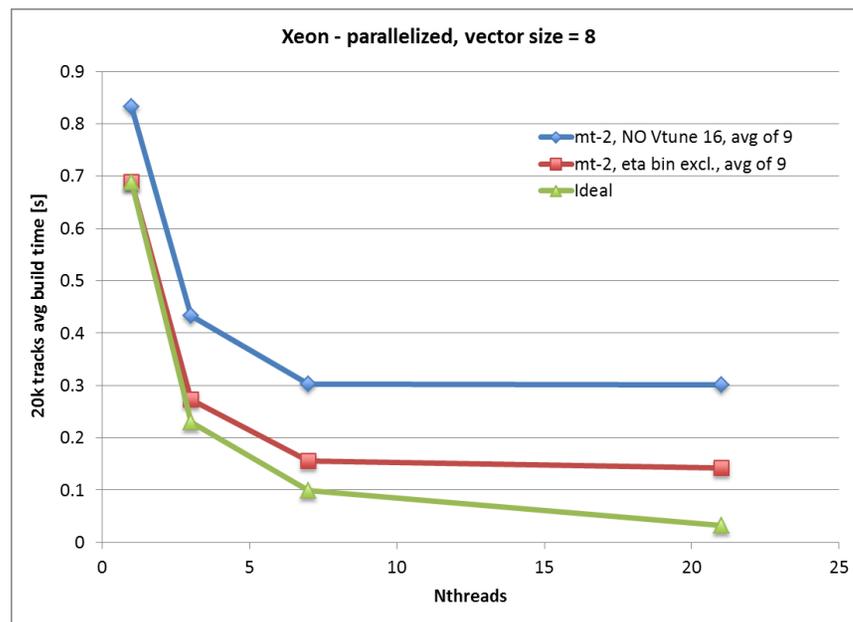
Function / Call Stack	CPU Time				S. Ti.	O. Ti.	Instructions Retired	Estimated Call Count	Total Iteration Count	Loop Entry Count
	Effective Time by Utilization	Idle	Poor	Ok						
+atan2f	30.159s				0s	0s	95,864,160,737	0	0	0
+MkFitter::FindCandidatesMinimizeCopy	18.664s				0s	0s	50,085,135,005	7,775,125	179,018,687	7,730,839
+operator new	12.322s				0s	0s	27,901,756,809	0	0	0
+Matrplex::CramerInverterSym<float, (int)3, (int)8>::Invert	7.761s				0s	0s	17,991,701,816	0	0	0
+Matrplex::MatrplexSym<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
+Matrplex::MatrplexSym<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
+Matrplex::Matrplex<int, (int)1, (int)1, (int)8>::operator()	1.764s				0s	0s	4,085,086,597	0	0	0
+Matrplex::Matrplex<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
+Matrplex::MatrplexSym<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
+Matrplex::MatrplexSym<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

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

- Further analysis revealed one bottleneck for non ideal vectorization performance
- 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
- Main idea for further improvement is to sort track candidates in a smart way
 - sort by position on next layer, sort by curvature, sort by χ^2 , ...

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 problematic 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

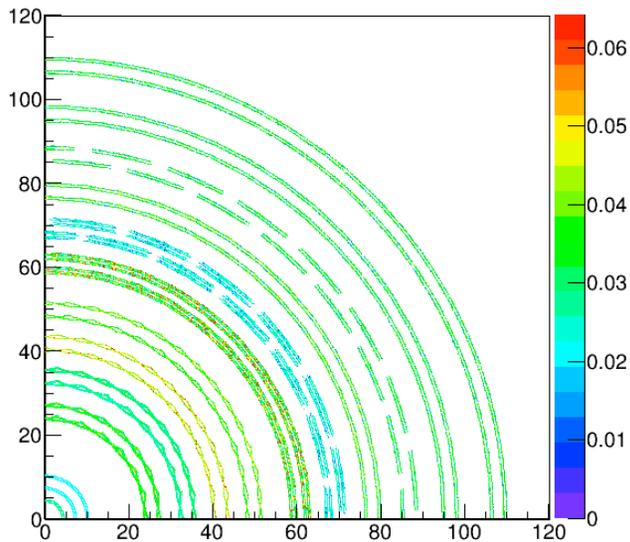
Summary of recent improvements

- 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?

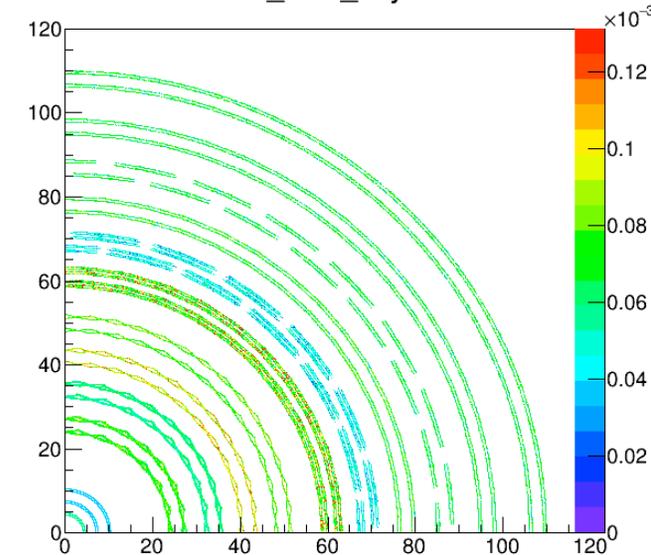
- We are working towards reconstructing tracks from a full simulation or from real detector data
 - non ideal geometry, material effects
 - detector inefficiencies, non-gaussian tails in hit position
 - particle clustering in jets, particle decays
- The simplest way to do this is to interface with CMSSW
 - 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)
 - can be useful for all sort of tracking studies in CMS

Material in CMSSW reconstruction

h_radL_axy

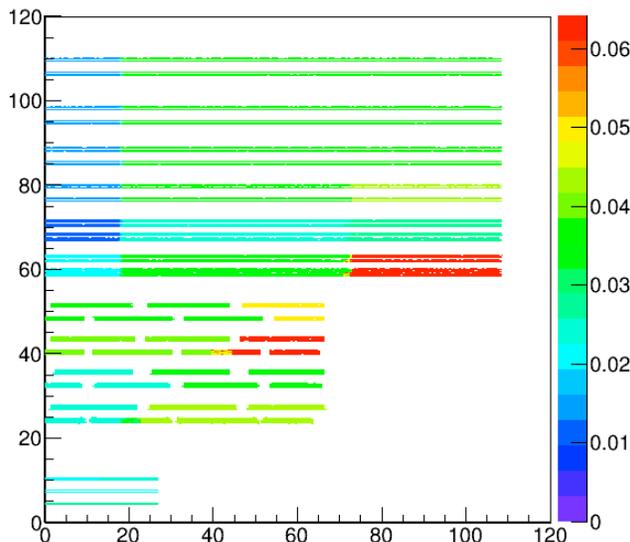


h_bbxix_axy

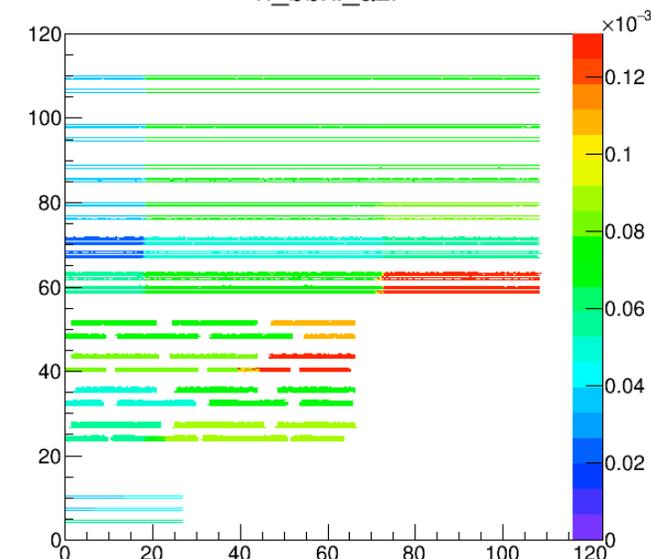


In CMSSW the tracker material is parametrized in terms for radiation length (radL) and energy loss ($\xi = Kz^2Z/A$ term in Bethe-bloch formula)

h_radL_azr

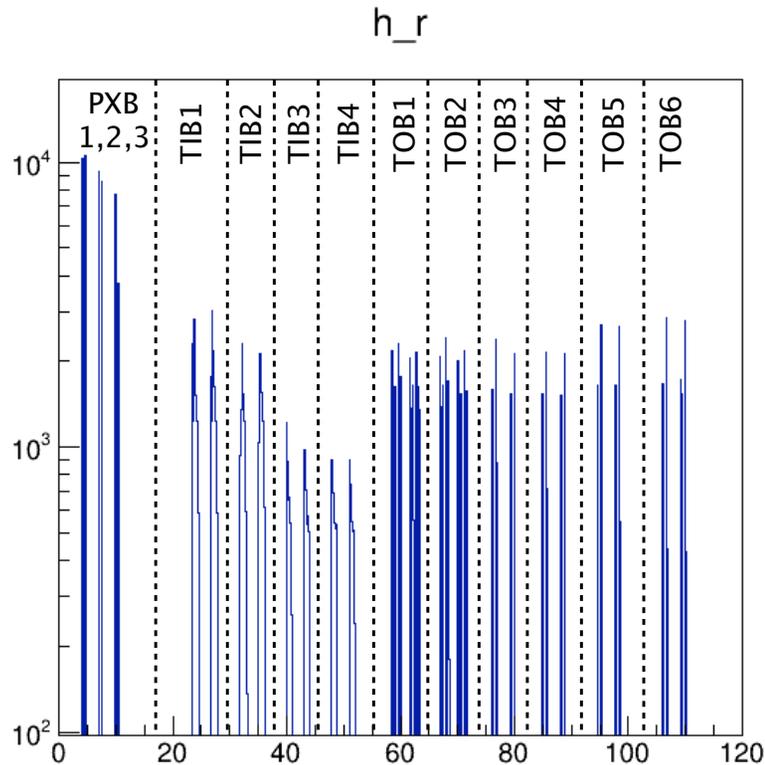


h_bbxiazr



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.



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

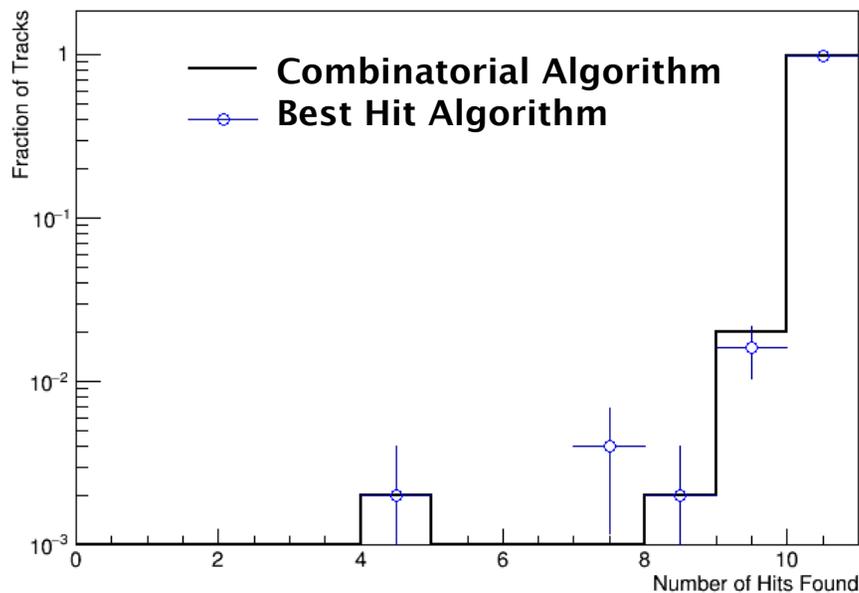
Build reco geometry with cylindric barrel layer at average radii.
 First propagation step to average radii, find hits in compatibility window.

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.

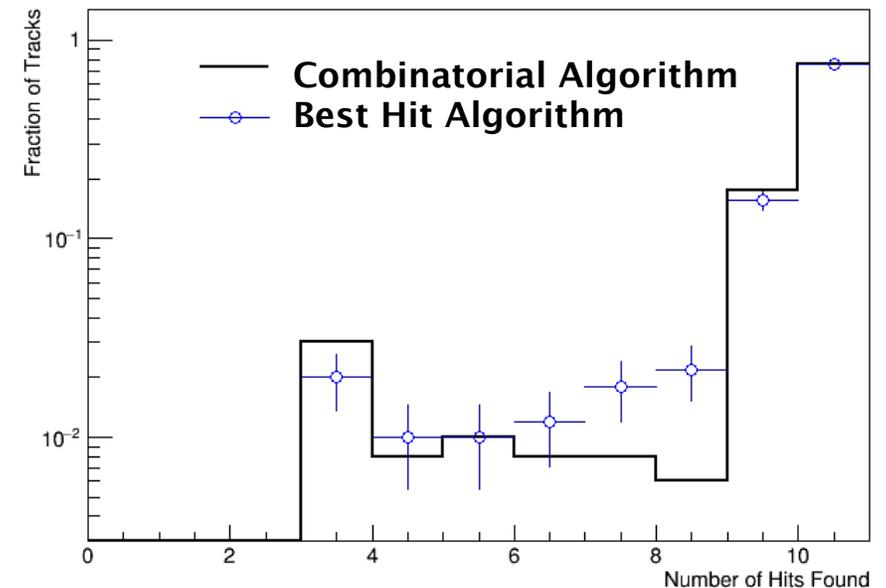
Hit Finding Performance

1x500 muons with $p_T=10$ GeV



Sim Hits from CMSSW

1x500 muons with $p_T=10$ GeV



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

- R&D project for tracking on Xeon Phi gives promising preliminary results
 - large speedup both from vectorization and parallelization
- Main bottlenecks are identified
- Further improvements are being explored, many already implemented
- Code interfaced to CMS full simulation, geometry and material properties
 - already good performance, especially for high p_T tracks
- 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
 - ...
 - explore other architectures (GPU already started)
 - comparison with current reconstruction model
 - porting and deployment in official reconstruction