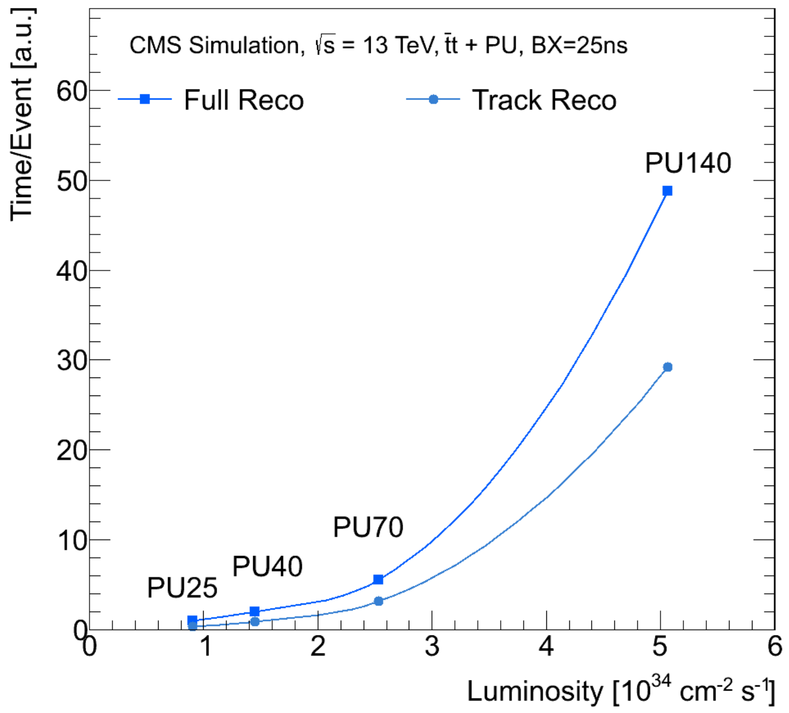
Kalman Filter Tracking on Parallel Architectures

CHEP 2015 – Apr. 13, 2015

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

Introduction



- 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!
    - both in online and offline processing
- The solution is a phase transition to **highly parallel architectures**
  - but code needs a **hardware-specific design** for optimal performance
  - algorithms cannot be ported in a straightforward way
  - initial target is the most time consuming algorithm, **track reconstruction**

Test Setup

- **Xeon Phi** as starting point, no real prejudice on architecture
  - but more direct porting of optimizations to **Xeon**
    - in fact we test performance on both
  - the name of the game is to keep the many **processors occupied** and the **vector units on sync**, performing the same calculations and thus minimizing branching points

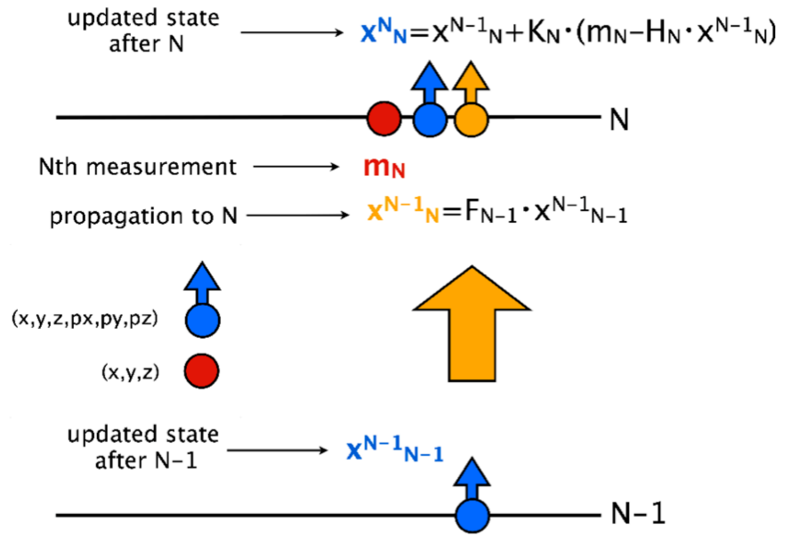
- **Standalone tracking code**
  - started with a simplified setup
    - Ideal barrel geometry, no material interaction, gaussian hit position smearing
    - Particle gun simulation, no interactions/decays
  - prepared to increase complexity along the way

Kalman Filter Tracking

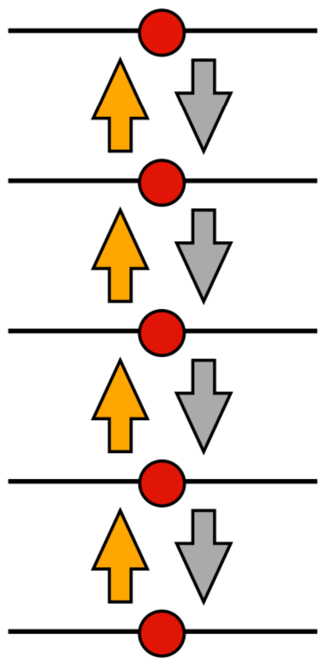
Kalman Filter tracking widely used in HEP, outstanding performance in LHC environment.

It consists in the reiteration of a **basic logic unit** for each tracker layer.

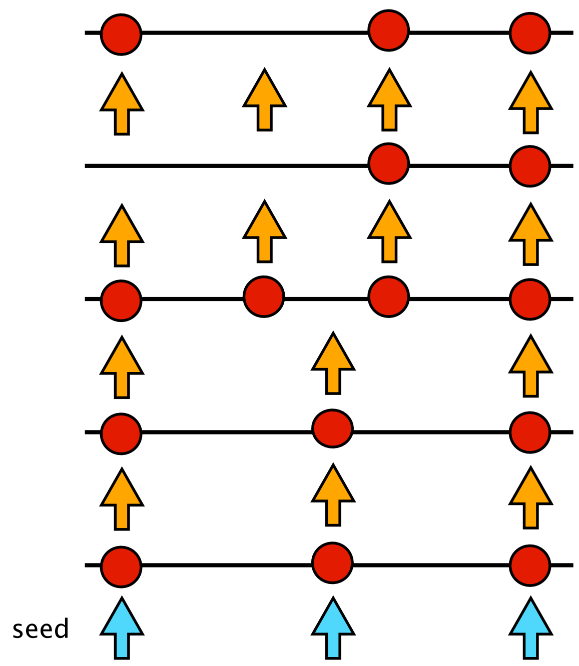
basic logic unit



track fit



track building



The track reconstruction process can be divided in 3 steps: track seeding, building and fitting.

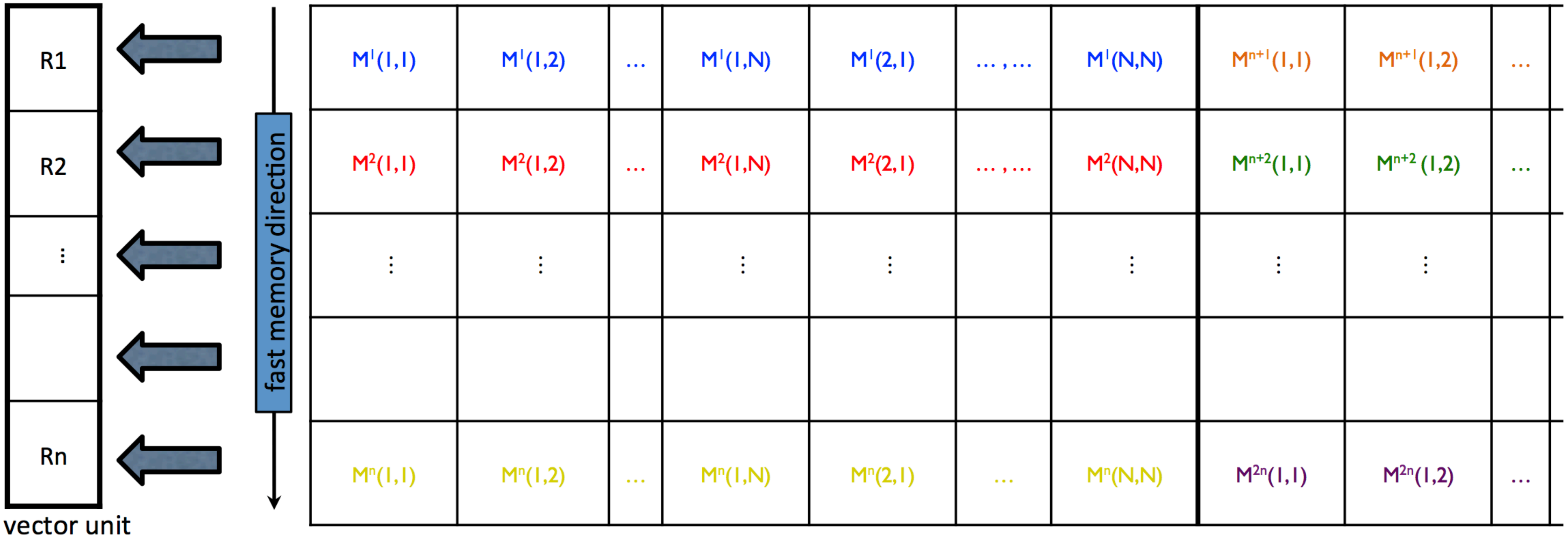
The **track fit** is the bare repetition of the basic unit, ideal as a **starting point**.
Track **building is the most time consuming part** – it involves branching points of variable size, with the simplest version degenerating into the track fit case.
Track **seeding** under development but not included yet, for now seeds are defined using MC info.

Dedicated data structures: Matriplex

Approach successful only if data structures are optimized for the specific architecture.

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?

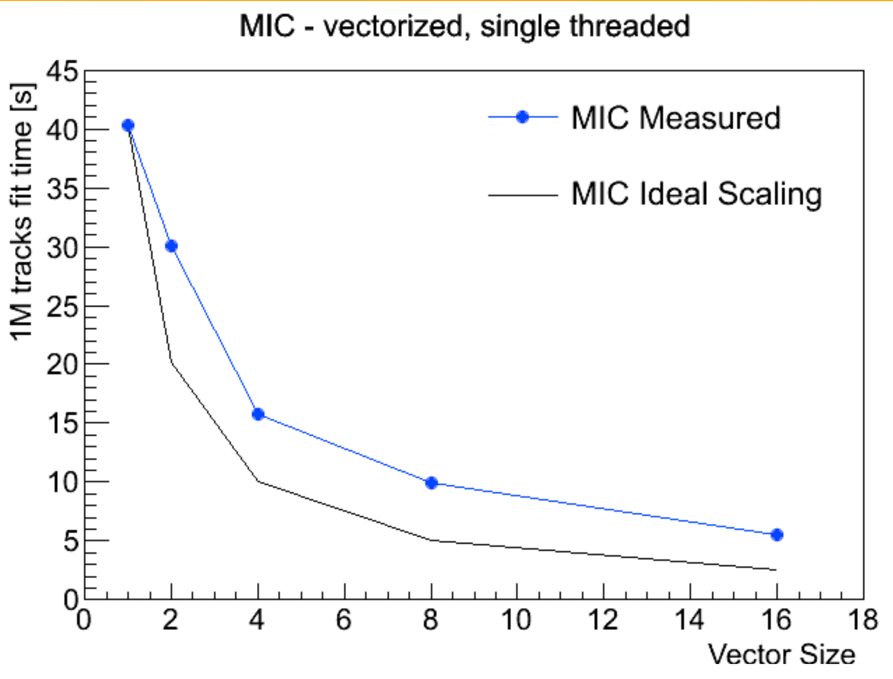
Matriplex is a "matrix-major" representation, where vector units elements are separately filled by a different matrix: **n matrices work in sync**.

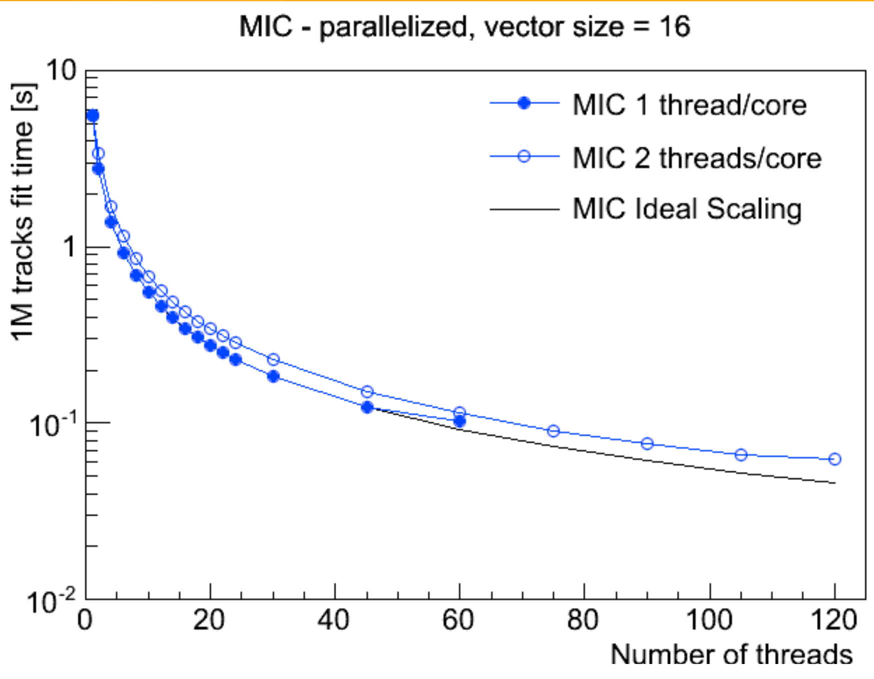


Matrix size **NxN**, vector unit size **n**

Previous Results

arXiv:1409.8213





- **Track fit** implemented using **Matriplex**
  - same physics results and faster than SMatrix even in serial case
  - tested both on **Intel Xeon and Xeon Phi** (native application) with OpenMP, **similar qualitative results**

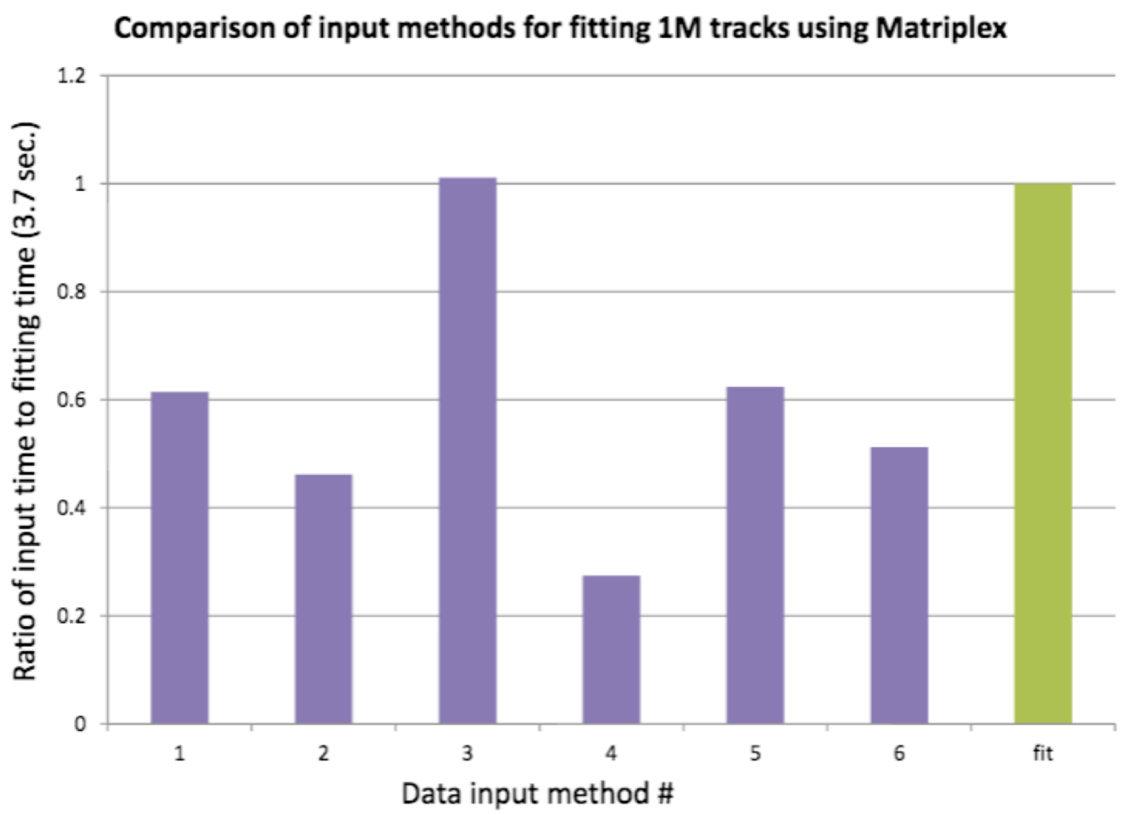
- Observe **large speedup** both from vectorization and parallelization.
  - Effective performance of vectorization is about 50% utilization efficiency.
  - Parallelization performance is close to ideal in case of 1 thread/core
    - some overhead with 2 threads/core

- Both **issues related to L1 cache**
  - Data availability and data packing in matriplex format

Lines of Development

- **Consolidate track fitting** results
  - identify bottlenecks for vectorization
  - improve simulation towards a realistic setup
- **Develop track building**
  - start with simple geometry setup
  - target full vectorization and parallelization

Improving Vectorization

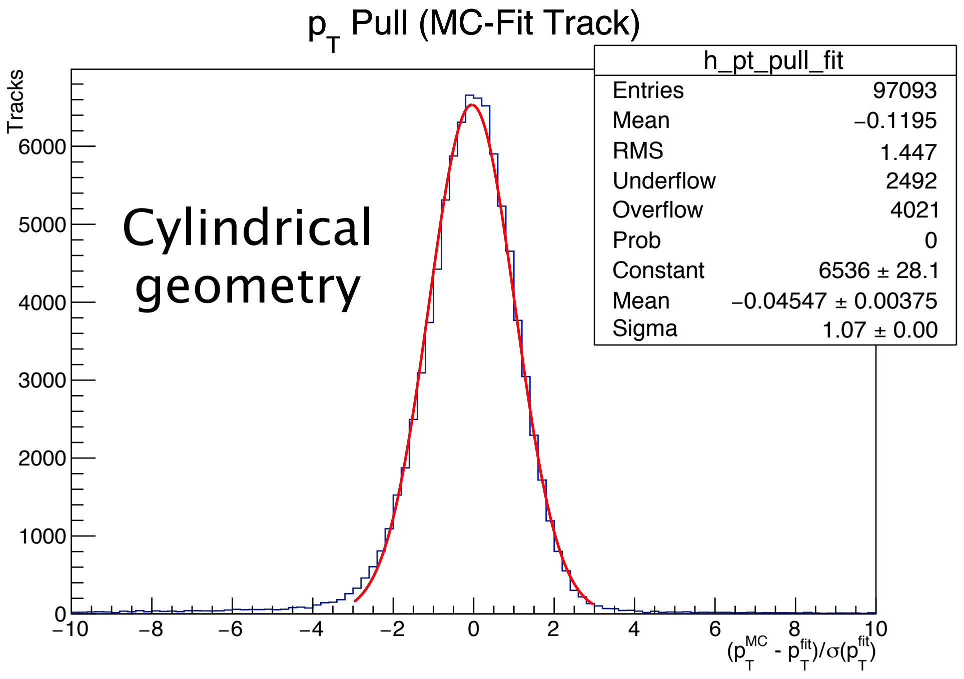


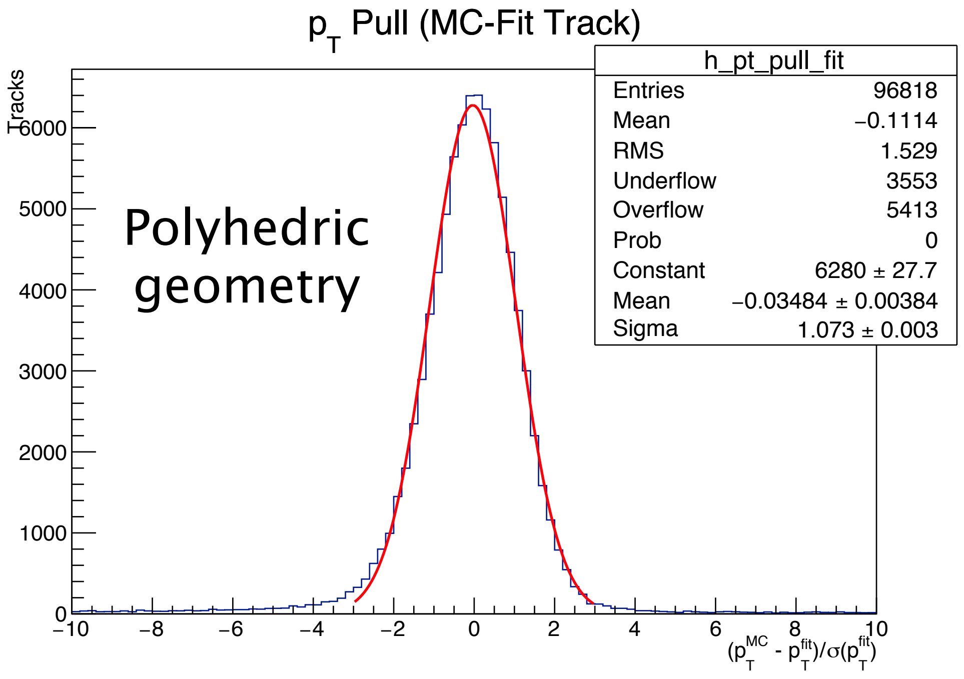
- Relative fraction of **time for data input is large** with respect to total fit time
  - input consists of data transfer and repacking into Matriplex format
  - plot shown for Xeon Phi, single-threaded, using the vector units to the extent possible

- We explored different approaches (1-3) and different intrinsics (4-6) resulting in substantially different performance
  - 1, 6: Scatter data from tracks into Matriplexes, track by track, using for-loops or vscatter intrinsic
  - 2-5: Two stages: copy data into packed temporary, then use strided loads, MKL transpose, or intrinsics
  - **Best method (#4) relies on two-stage copy and vgather intrinsic**

More Realistic Setup

- Progress towards implementation of the **full tracking chain**:
  - fit track obtained from building, not straight from simulation
- More **realistic detector** simulation:
  - Added option to include multiple scattering in simulation
  - Using USolids geometry package
    - implement different barrel-like layouts in a transparent way
    - still propagating track parameters to hit radius
    - **timing and physics performance of track fit do not change!**



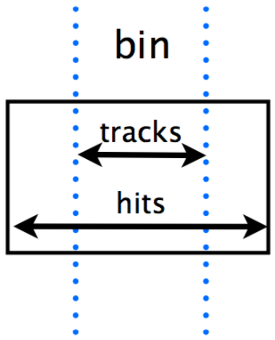


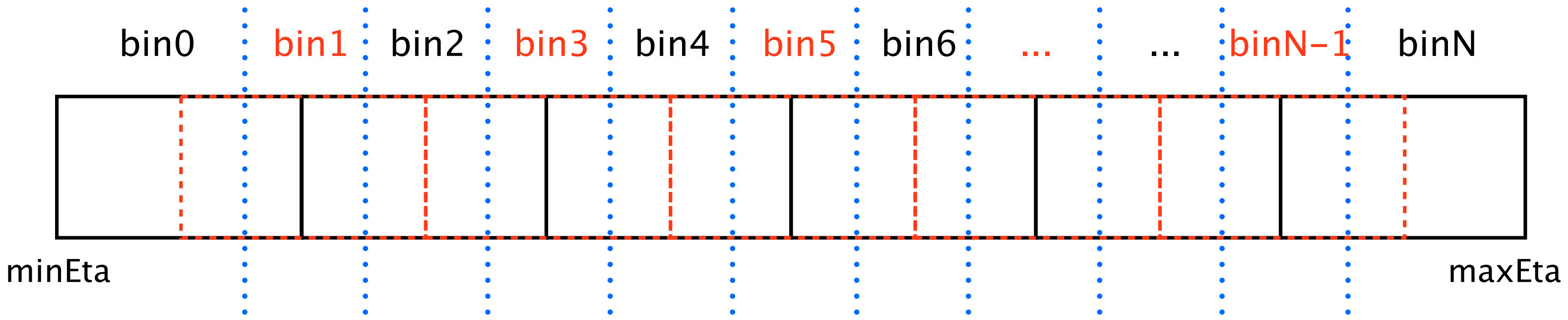
Strategy for Track Building

- 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**

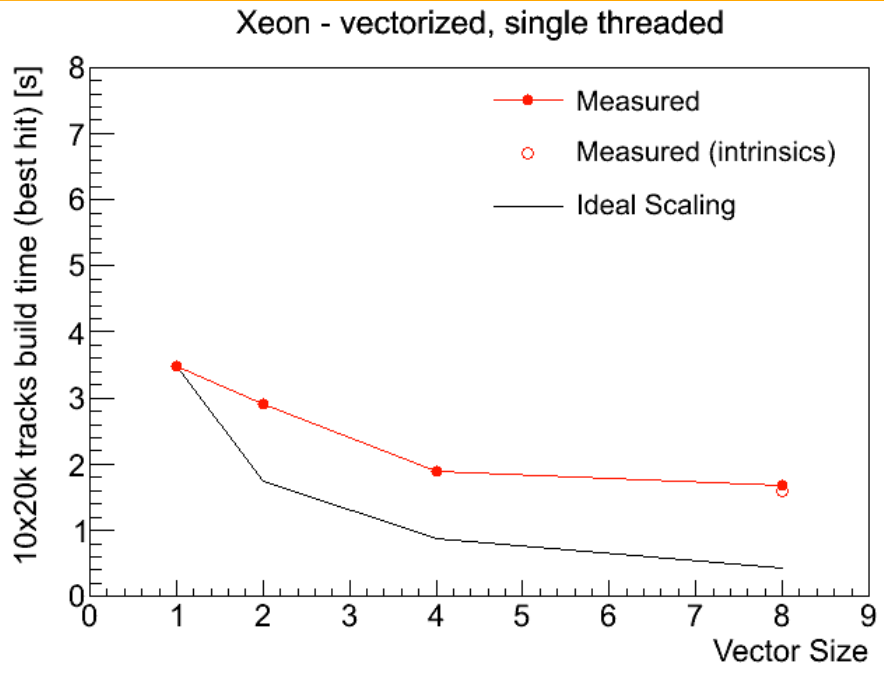
- The two issues can be **factorized** by dividing the development in two stages
  - first develop a simplified algorithm choosing only the best hit on next layer
    - deal with large number of hits, not with cloning – study vectorization in this case first
  - then full implementation with combinatorial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

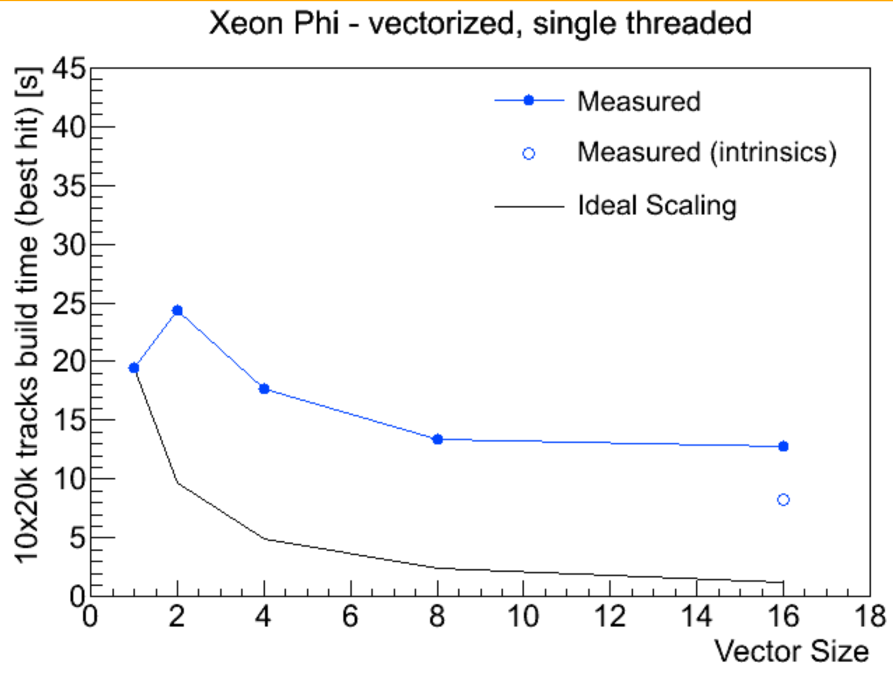
- **Data locality** is the key for reducing the Nhits problem
  - **eta partitions** are **self consistent** (no bending)
    - bins redundant in terms of hits, track candidates never search outside their eta bin
    - natural **boundary for thread definitions**
  - phi partitions give fast lookup of hits in compatibility window





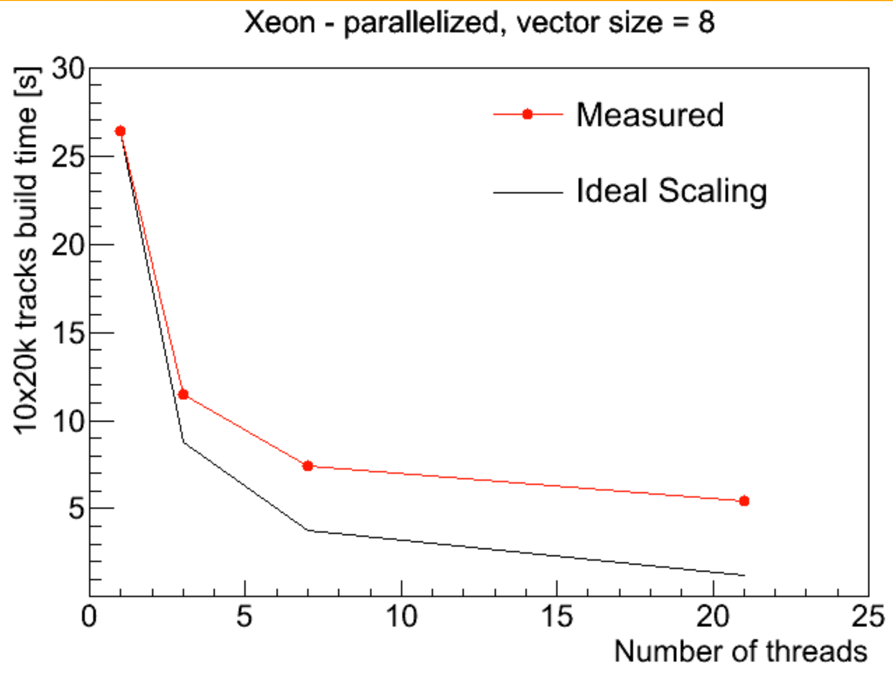
Vectorization Results

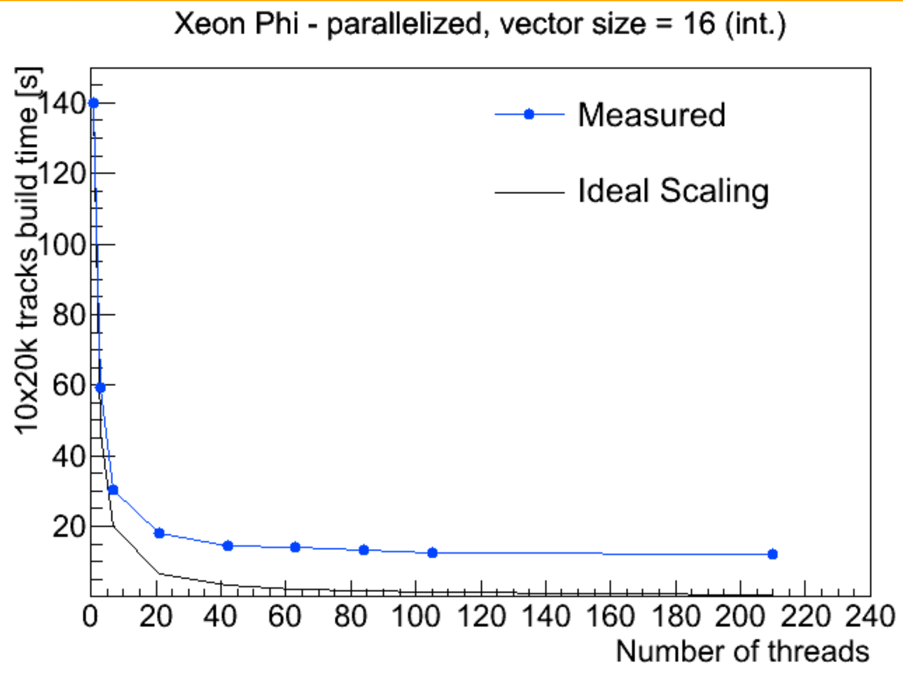




- Run **simplified track building** (best hit option) on 10 events with 20k tracks each
  - pick hit in compatibility window with lowest chi2 at each layer
  - 70% (93%) of tracks found with ≥90% (60%) of the hits
- Already much **more difficult than fitting case**, expect worse results:
  - test multiple (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** instrinsics, but data input still takes a large fraction of the time!

Parallelization Results





- Run **full track building** with combinatorial expansion of candidates
  - ultimate physics performance, slower
  - 85% (95%) of tracks found with ≥90% (60%) of the hits
- 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

Conclusions

- First version of vectorized and parallelized track building implemented!
- **Significant speedup** achieved both on **Xeon and Xeon Phi**
  - 2x from vectorization
  - 5x on Xeon and 10x on Xeon Phi from parallelization

- Ideal scaling indicates a large **margin for further improvements**
- **More studies** and developments are needed:
  - identify and solve bottlenecks in vectorization and parallelization performance
  - performance studies at different track occupancy values
  - improve data locality: optimize partition size, smart sorting of candidates
  - complete transition to more advanced tools for parallelization (TBB)
- While at the same time progressing towards a **fully realistic setup**
  - use full event simulation
  - complete the chain starting from track seeding

backup

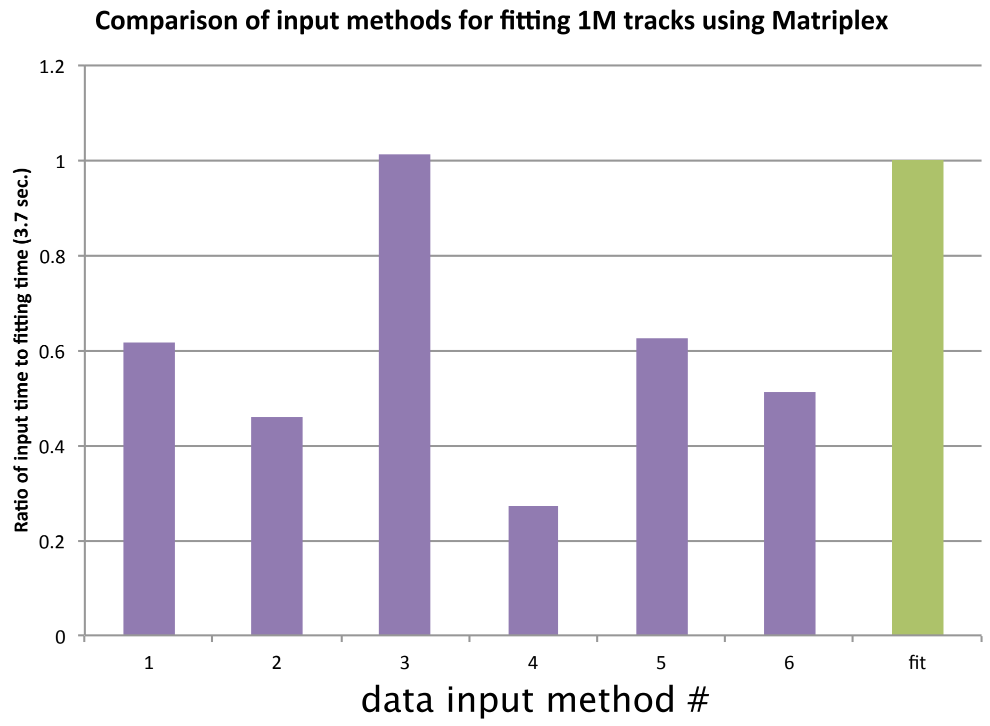
Specs Sheet

- Xeon
  - CentOS 6.6, 2x6 core Xeon E5-2620 @ 2GHz, 64 GB RAM, turbo off, hyperthreading on
- Xeon Phi
  - Xeon Phi 7150, 16 GB RAM, 61 cores @ 1.24GHz, MPSS 3.4.3
- Intel compiler
  - icc (ICC) 15.0.2 20150121 with gcc-4.8.2 libstdc++/c++-11 support
- Open MP 4.0
- TBB 4.3.0
- Intel VTune 2015.2.0

Vectorization of CopyIn

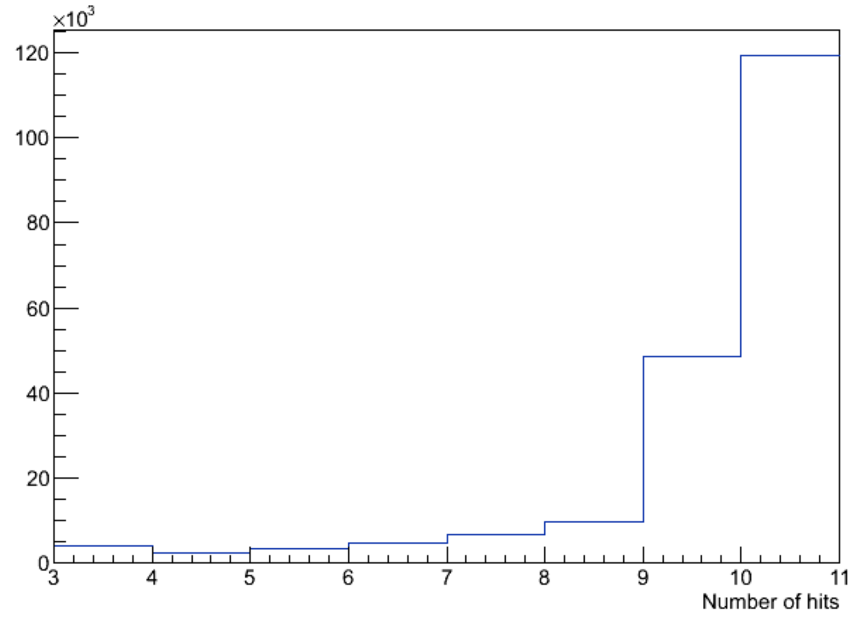
- Intel VTune's metrics revealed that CopyIn, which distributes input data into a Matriplex, was underperforming
  - Assembler code showed lack of vectorization on Xeon Phi
  - Compiler was not converting strided for-loops into vectorized stores
  - Better: copy all data into a packed temp array, do strided loads
- Fastest Xeon Phi code uses Intel's _mm512 vector intrinsics
  - Vgather gave additional gains for strided loads
  - Vscatter was not as helpful with strided stores
- Underlying operation is equivalent to a matrix transpose
  - Intel MKL includes an optimized out-of-place transpose routine
  - It didn't work well; probably it is targeted for large-matrix operations

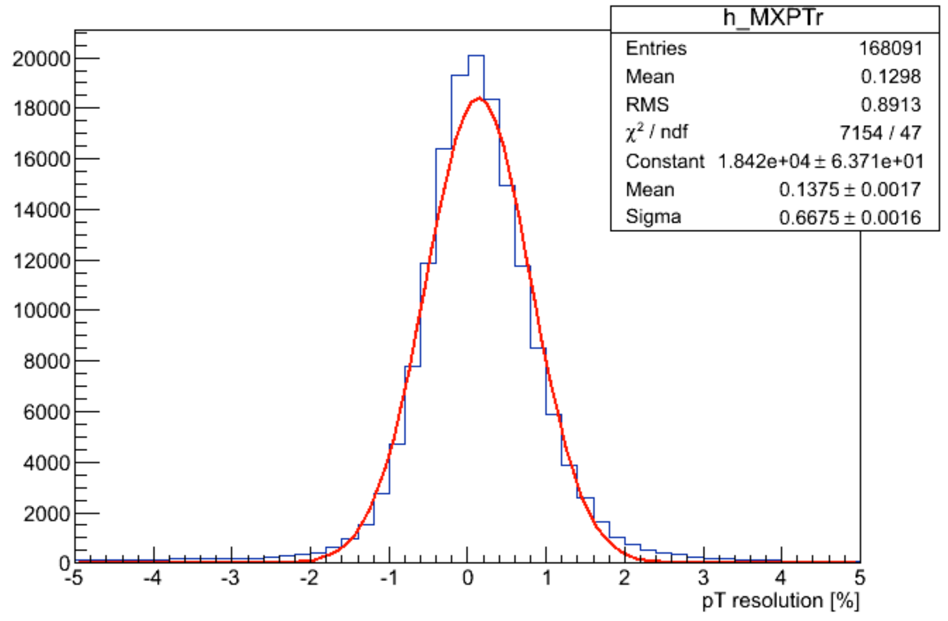
Details of Input Methods



- 1. CopyIn: scatter data from tracks into Matriplexes, track by track, using simple for-loops
- 2. CopyInContig + Plexify: copy 16 tracks into contiguous memory, transpose into Matriplexes via loops
- 3. CopyInContig + PlexifyMKLOut: copy 16 tracks into memory, use Intel MKL to transpose each Mplex
- 4. CopyInContig + PlexifyIntr (vgather): copy 16 tracks into memory, vgather into Matriplexes by rows
- 5. CopyInContig + PlexifyIntr2 (vscatter): copy 16 tracks into memory by rows, vscatter into Matriplexes
- 6. CopyInIntr (vgather + vscatter): scatter data into Matriplexes, track by track, using Intel intrinsics

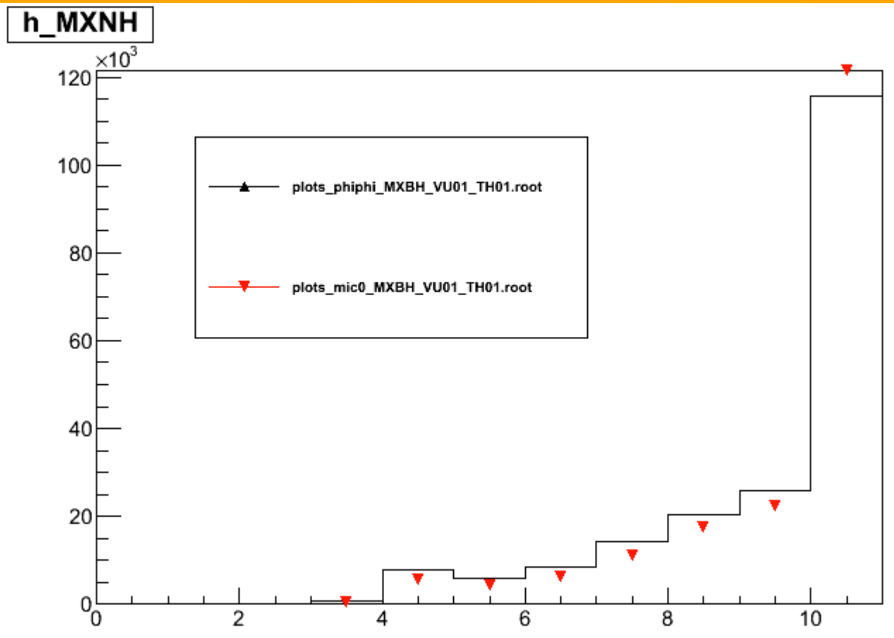
Performance of Track Builiding

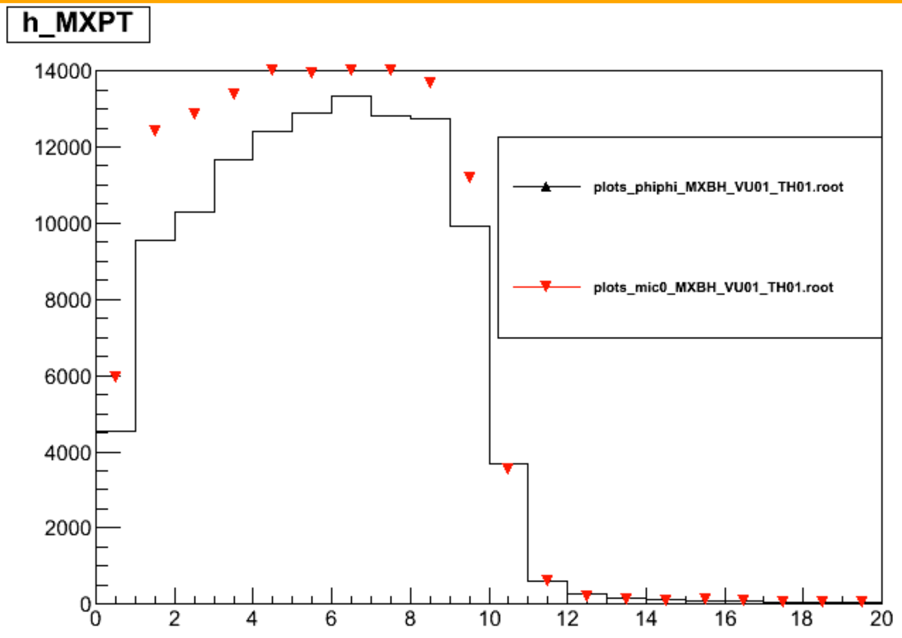




- Algorithm parameters not tuned for ultimate physics performance yet
- But results already in very good shape
  - large hit collection efficiency
  - tracks with ≥ 9 hits have $<1\%$ resolution

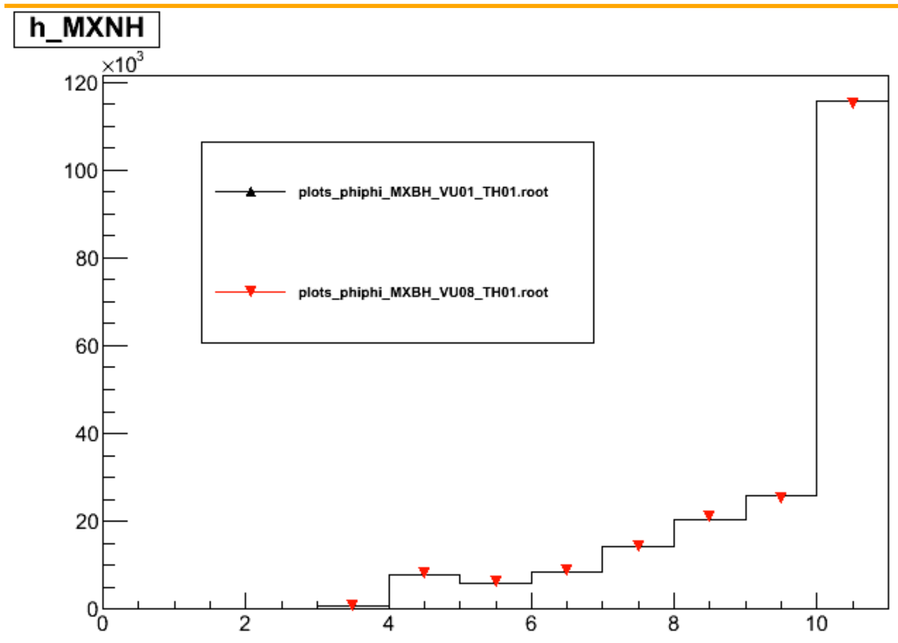
phiphi vs mic0

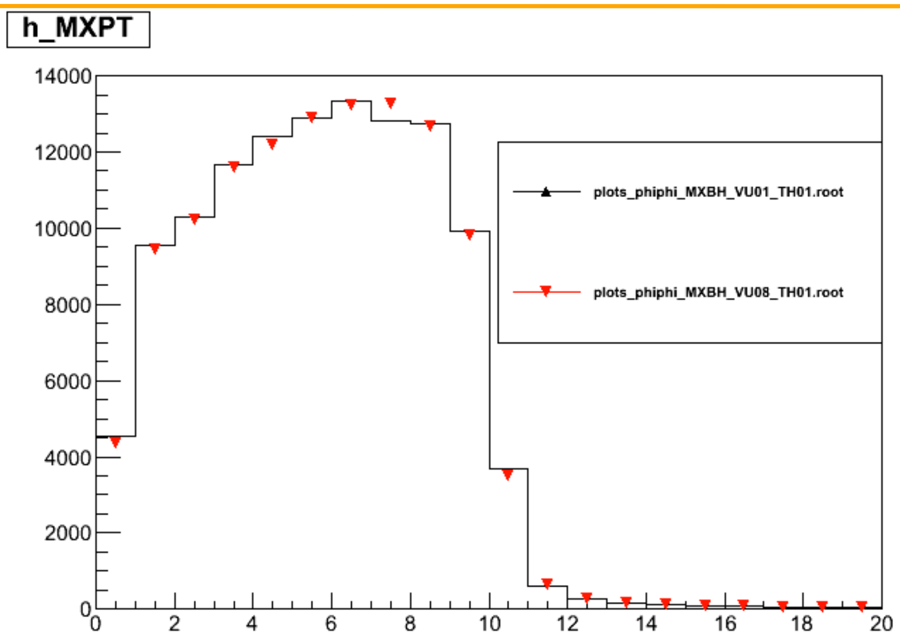




- looks like we do better on mic0?
- how come code is not the same?

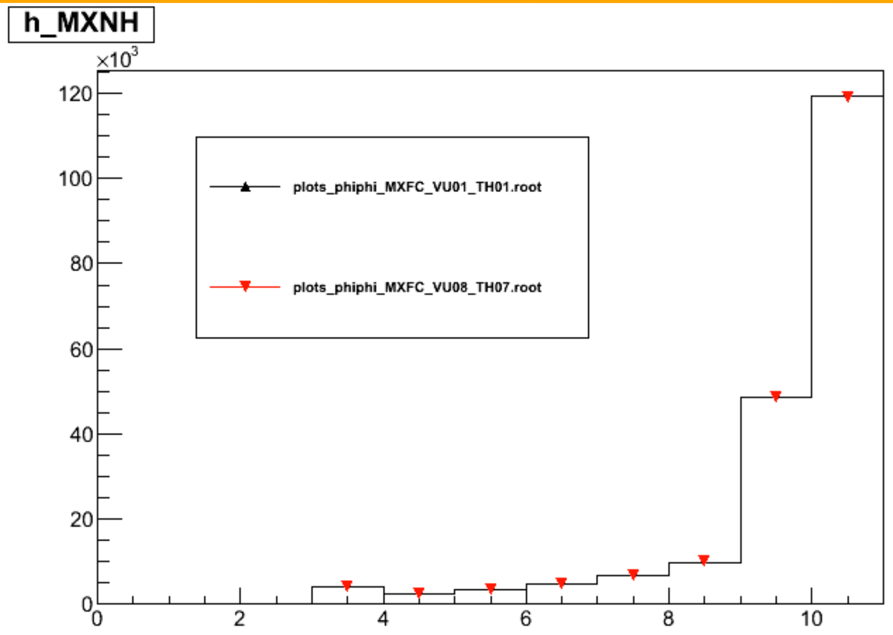
BestHit on phiphi

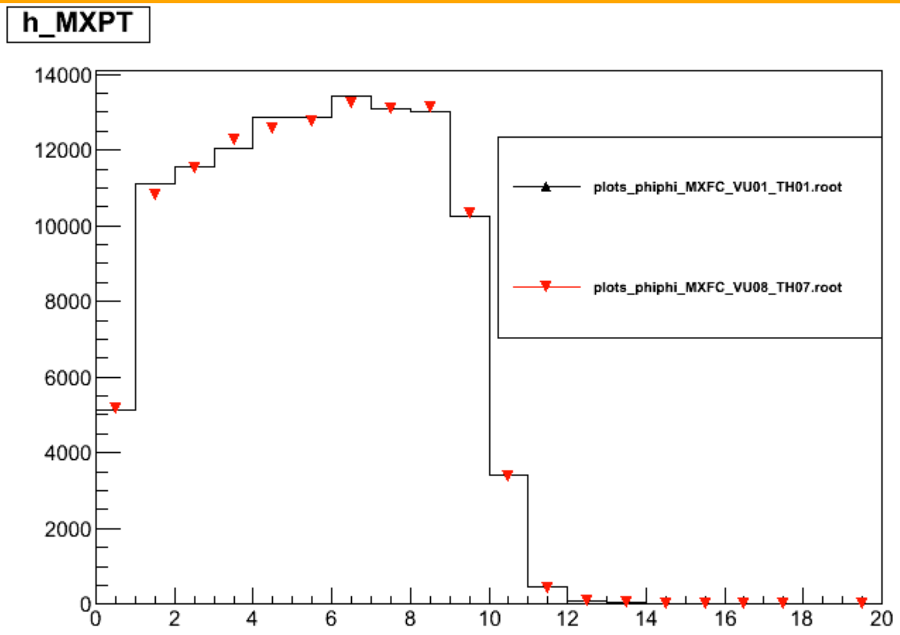




- change vector unit size from 1 to 8
- not identical but very close

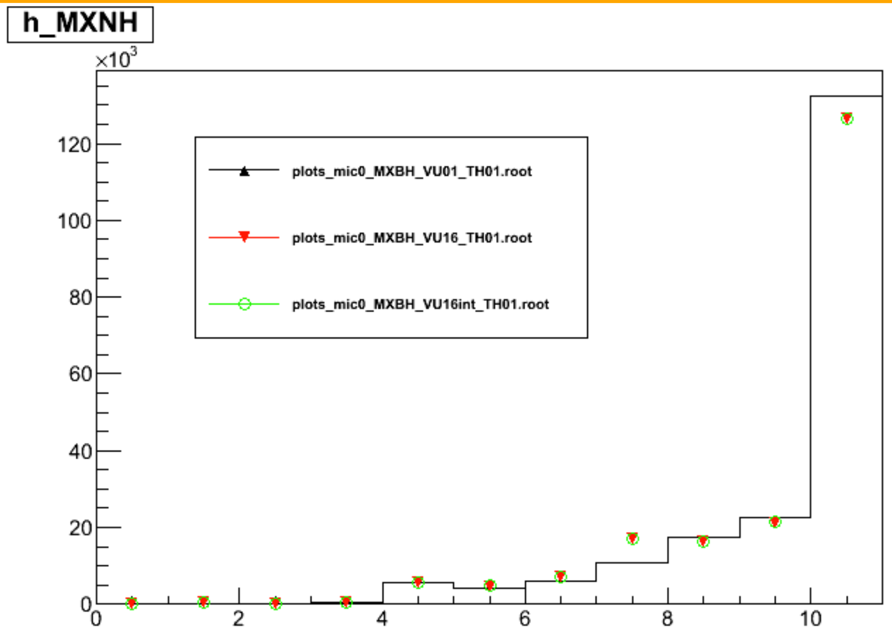
FindCandidates on phiphi

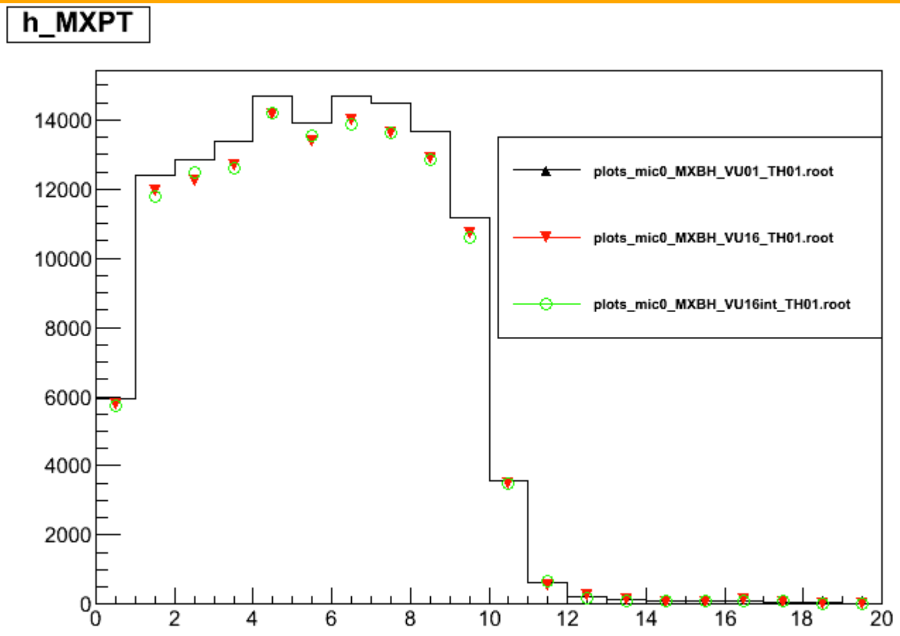




- change both vector unit size and number of threads
- not identical but very close

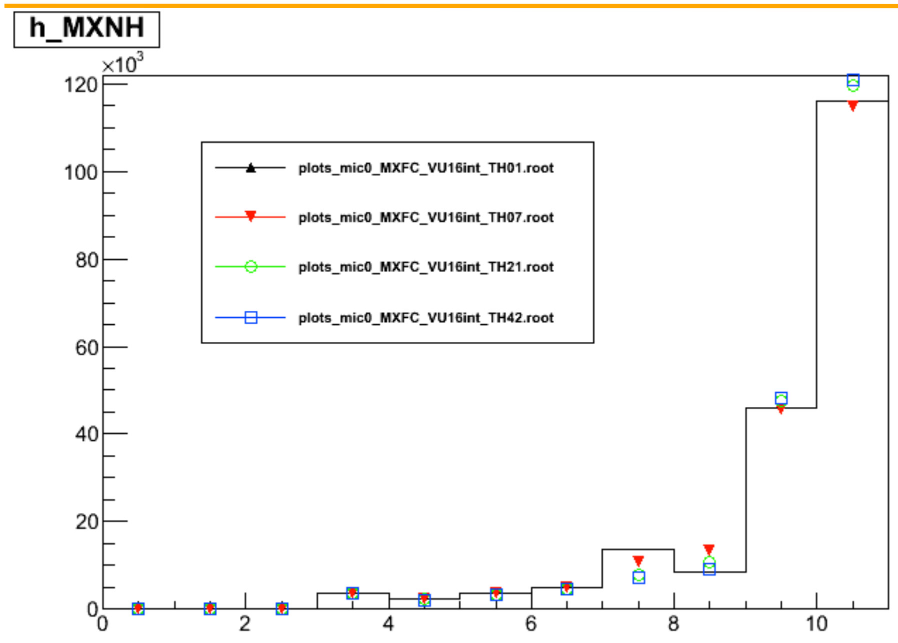
BestHit on mic0

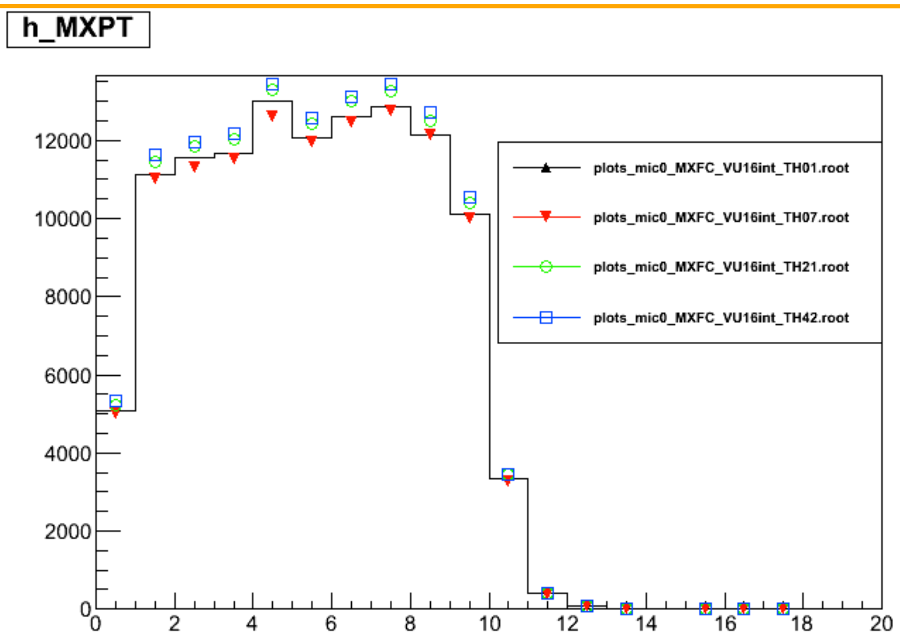




- change vector unit size from 1 to 16 (with and without intrinsics)
- large difference going from 1 to 16
- small difference when turning on intrinsics

FindCandidates on mic0





- change number of threads with fixed vector unit size
- nthreads=1 and 7 are similar (nthreads<nEtaBins)
- nthreads=21 and 42 are similar (nthreads>=nEtaBins)

Prefetching in AddBestHit

	phiphi	phiphi noPF	mic0	mic0 noPF
1	3.49	3.44	18.47	19.49
2	2.90	2.78	23.11	24.32
4	1.90	1.84	22.46	17.63
8	1.68	1.58	11.89	13.43
8 int	1.60	1.61		
16			11.13	12.83
16 int			8.28	10.24