Kalman Filter on Parallel Architectures

Giuseppe Cerati

HEP.TrkX kickoff meeting

Nov. 2, 2016

Collaboration

- UCSD
  - A. Yagil, F. Würthwein
  - M. Tadel, S. Krutelyov, post-doc replacing GC
- Cornell
  - P. Wittich
  - D. Riley, S. Lantz, K. McDermott
- Princeton
  - P. Elmer
  - M. Lefebvre

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

Platforms

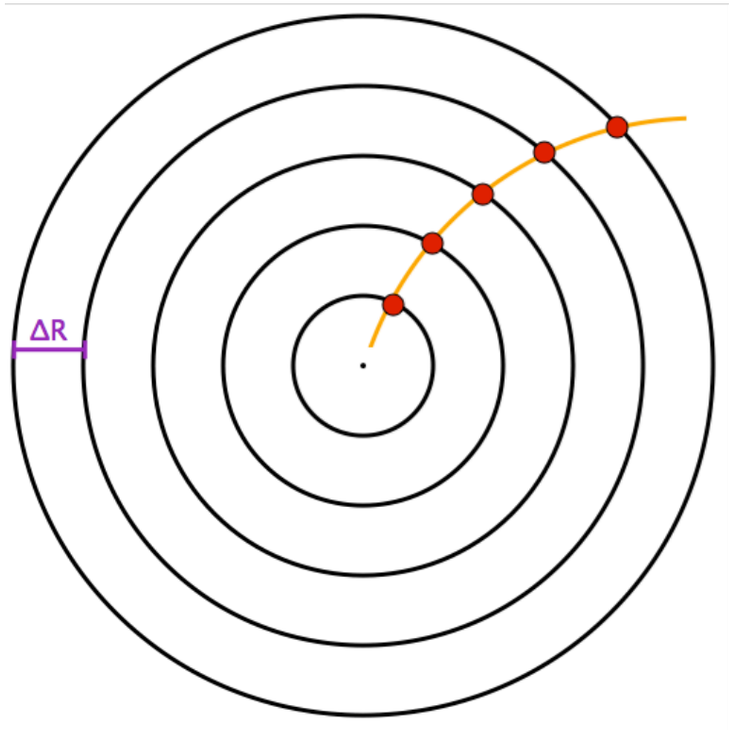
	CHEP2015 Reference Platforms		Preliminary Results	
	Xeon E5-2620 Sandy Bridge (SNB)	**Xeon Phi 7120P Knights Corner (KNC)**	**Xeon Phi 7230 Knights Landing (KNL)**	**Tesla K40**
Logical Cores	6x2x2	61x4	64x4	2880 CUDA cores
Clock rate	2.5 GHz	1.24 GHz	1.3 GHz	875 MHz
GFLOPS	120	1208	2660	1430
SIMD width	256 bits	512 bits	2x512 bits	32 thread warp
Memory	~64-384 GB	16 GB	16 & 384 GB	12 GB
Bandwidth	42.6 GB/s	352 GB/s	475 & 90 GB/s	288 GB/s

Caches typically 32+32KB (L1) and 256 or 512 KB L2.

Experimental Setup

Simple starting point:

- "Cylindrical Cow":10 barrel layers, △R = 4cm, |η|<1, 3.8T magnetic field
- Beam spot 1mm in xy, 1cm in z
- Hit resolution 100μm in r-phi, 1mm in z
- Uncorrelated tracks, no scattering



Add realism after platform issues are understood

Intel ICC compiler

Kalman Filter reconstruction

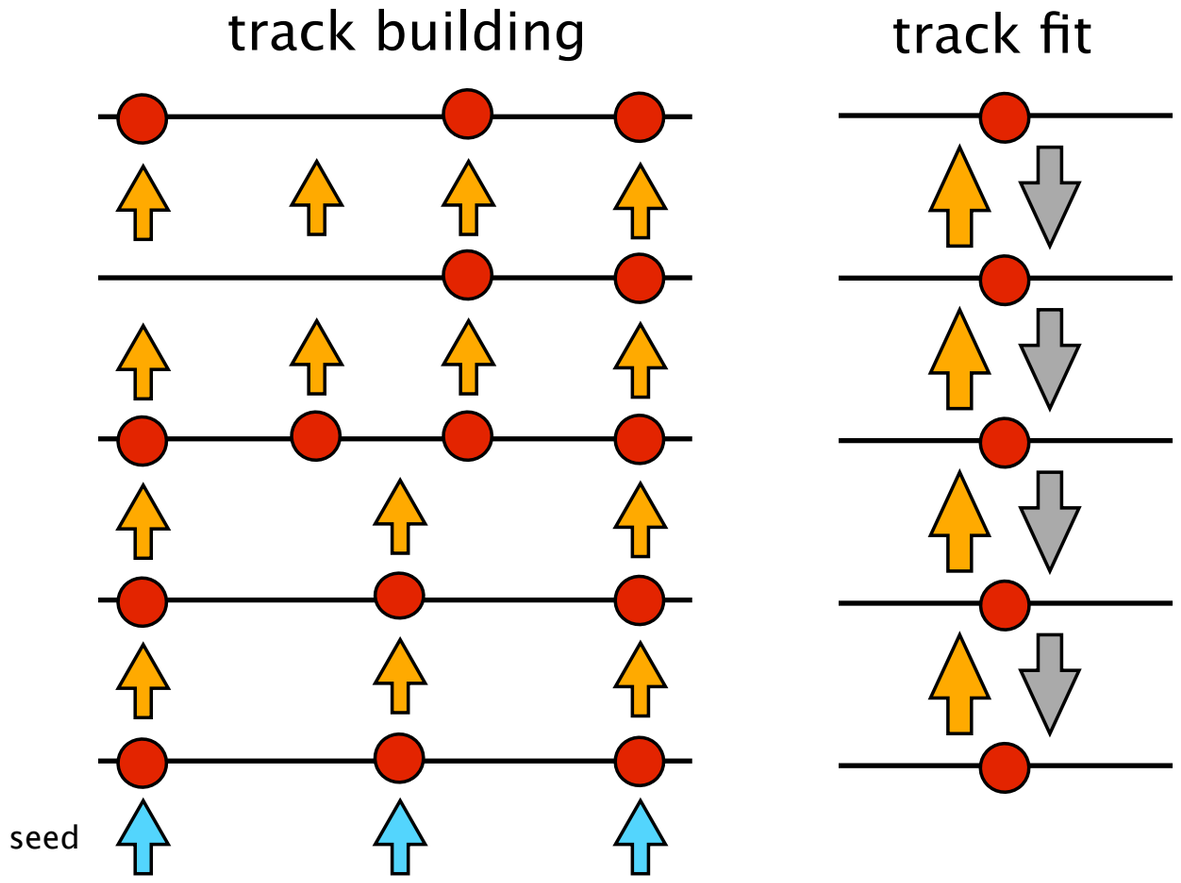
The Kalman Filter track reconstruction searches for hits along the track direction, with a search window that shrinks when more measurements are added.

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

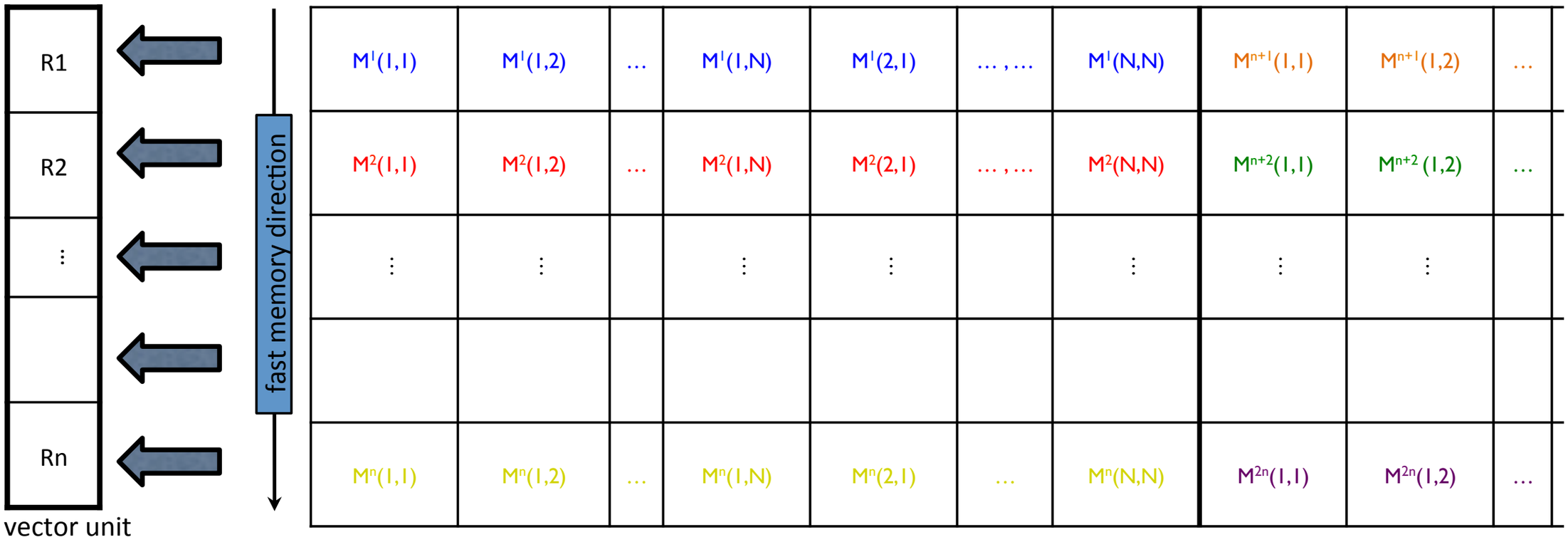
- Track fitting not affected by the first two issues: simple starting point

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**.

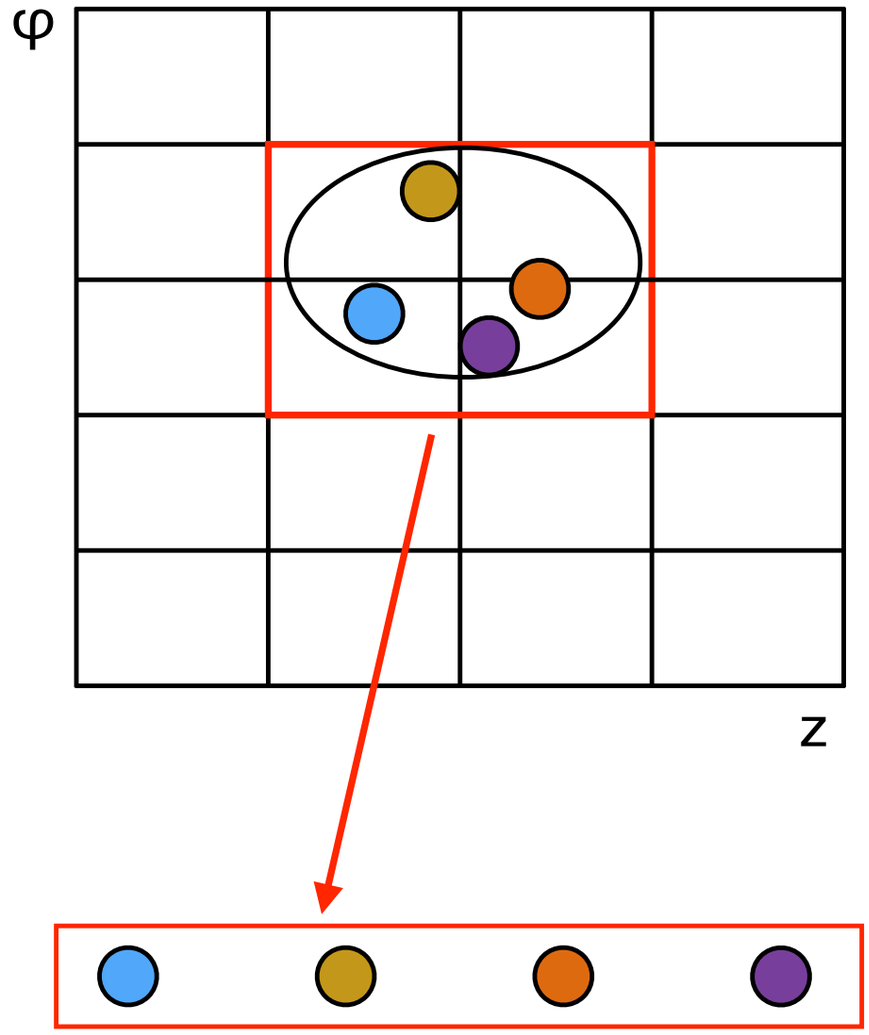
In other words, vector units are also used for SIMD parallelization (in addition to parallelization from threads in different cores)



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

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
  - regular 2D grid in z-phi
    - bin spacing based on measured track parameter errors (about 3sigma, layer dependent)
  - within a single z bin, hits still sorted in phi
  - non overlapping bins, no assumption on self-containing
    - hit information only duplicated
  - hit search done looping over bins in compatible z-phi window
    - precompute array of indices of hits in window
- Advantages:
  - no more problems with eta bin migrations
  - less redundancy of hit data (only one copy)
  - flexibility in definition of hit processing order within the array of indices
    - not bound to go from -phi to +phi, order of indices can be sorted (not done for now)
    - to avoid sorting and overwriting of output candidates
  - prerequisite for direct access to hits in original hit collection
    - no copy of hit data, but random access (hits not sorted in our code)
  - grid can be used for mapping material



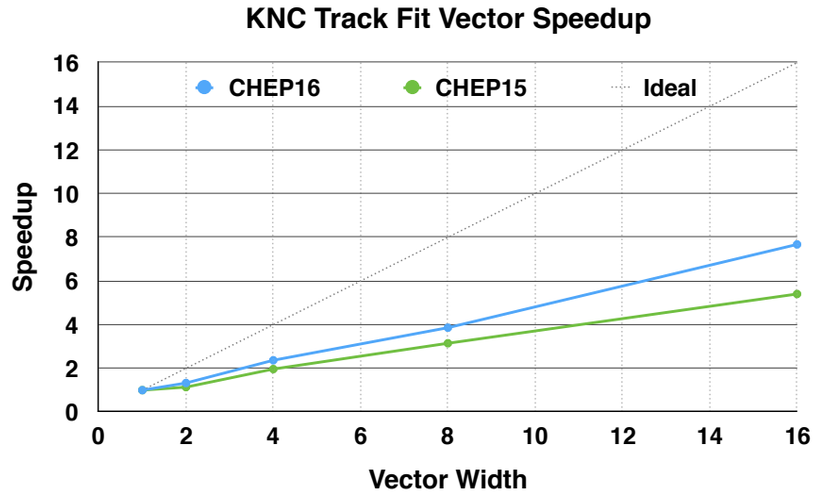
Vectorization: Track Fitting

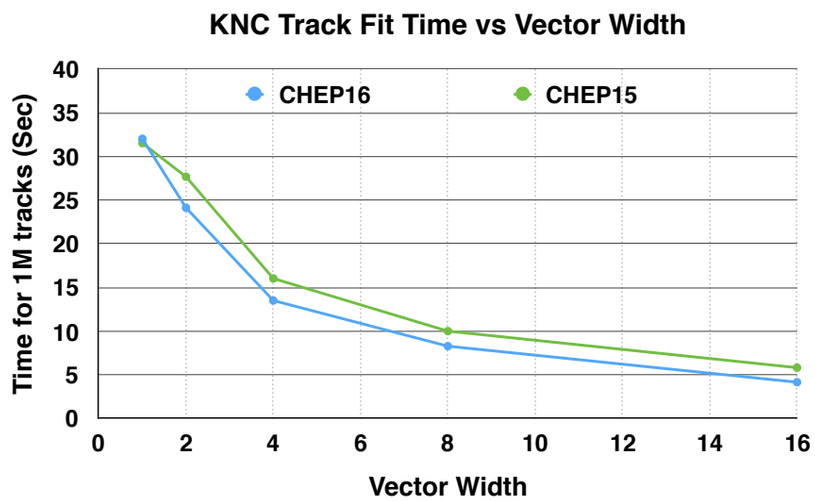
Improved somewhat from CHEP2015

- Physics improvements reduced performance
- Careful optimization restored scalar performance, improved vectorization ~33%
- SNB: 4x speedup with vector width 8, consistent with KNC at vector width 8

Subtle errors can lead to poor vectorization

- References to unaligned locations in aligned arrays
- Not using prefetching, scatter/gather, or intrinsics
- Conversions to double precision
- Re-using variables that could be const with smaller scope

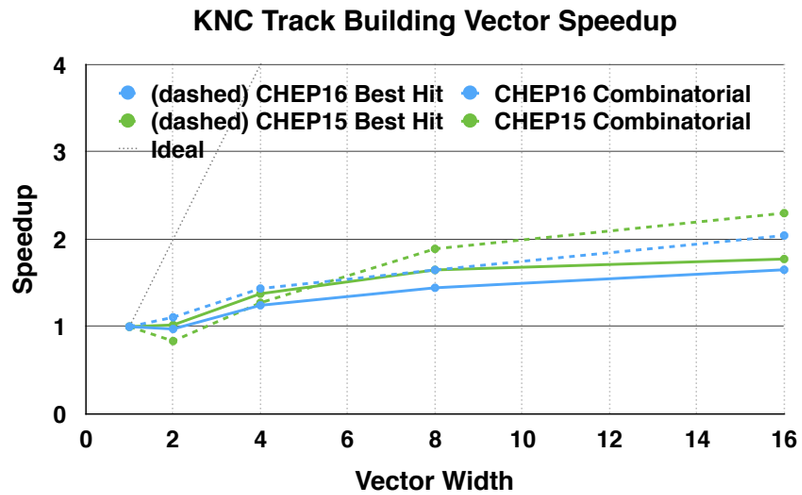




Vectorization: Track Building

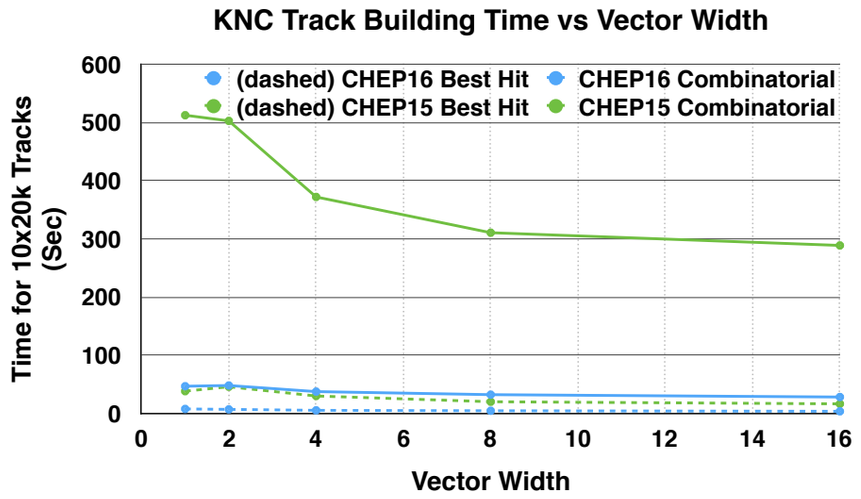
Much more challenging:

- Branches to select candidates impairs vectorization
- Adding multiple candidates at each layer leads to frequent data repacking
- More complicated data structures and poorer data locality stress cache size and memory bandwidth



Results are better understood, but performance is no better than CHEP2015

- ~2x speedup (SNB: also ~2x speedup)
- Improving this becomes more critical as number of vector registers increases



Combinatorial track builder *much* faster than CHEP2015

Track Building Optimizations

Data locality is critical (w/speedups relative to CHEP2015):

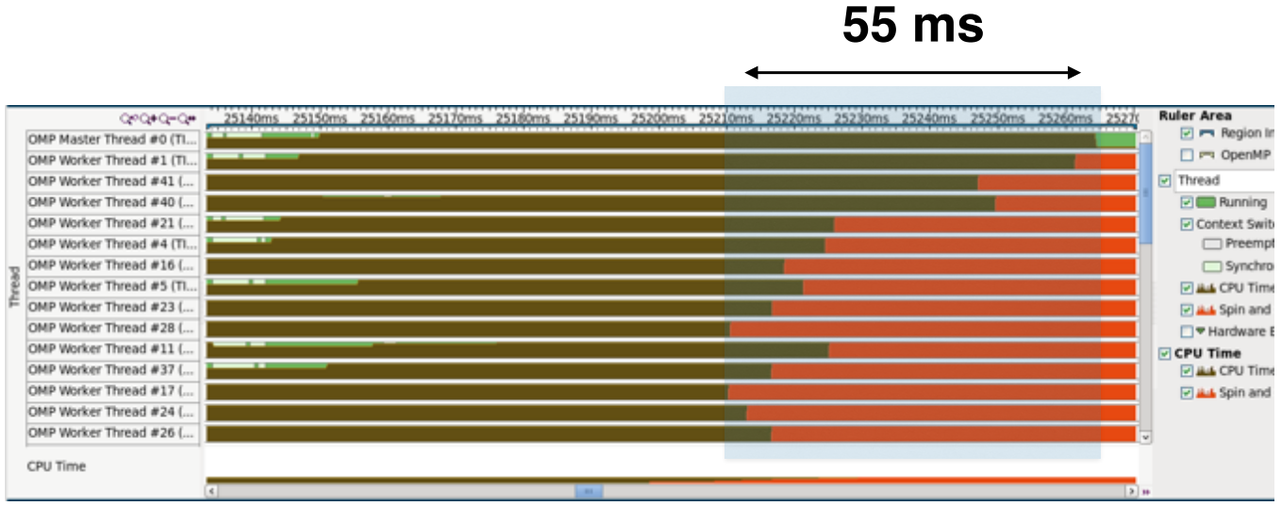
- Optimize/vectorize copying of tracks into Matriplex (+20%)
- Minimize dynamic memory allocations (+45%)
- Avoid unnecessary object instantiations, copies (+25%)
- Minimize size of data structures, smarter low-level algorithms (+30%)

Parallelization switched from OpenMP to Threading Building Blocks (TBB)

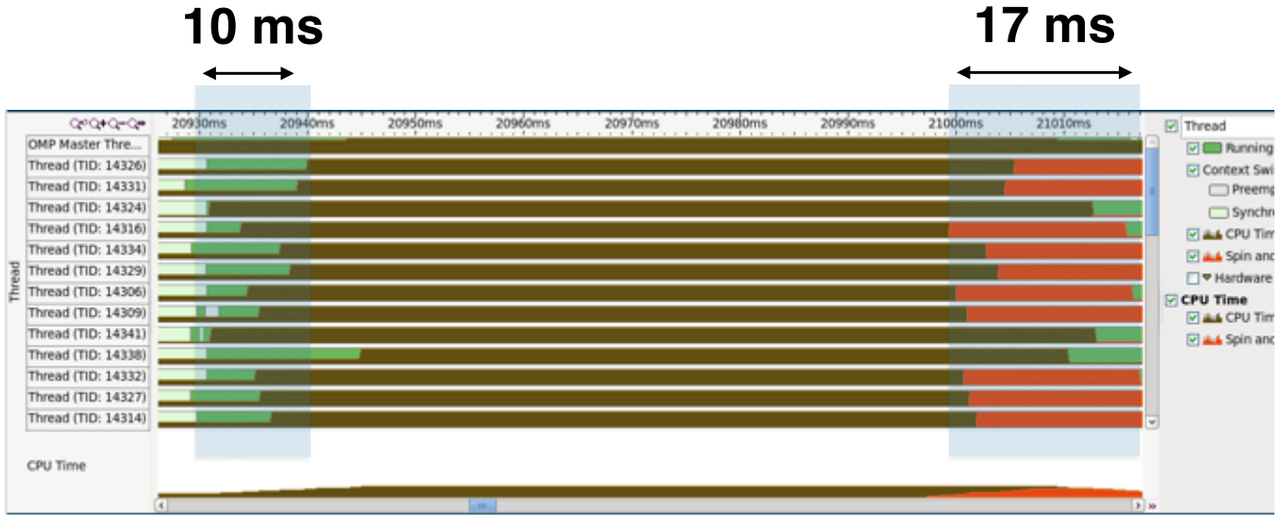
- Static binning with OpenMP led to "tail effects" due to variable distribution of work
- TBB work-stealing provided an easy way to even out load variability
- Optimizing work partition size still critical—too large doesn't allow enough balancing, too small has high over head costs

vTune: OpenMP vs. TBB

OpenMP shows large tail effects due to uneven distribution of work from static partitioning



TBB work stealing, wth smaller units of work and dynamic partitioning, reduces tail effects



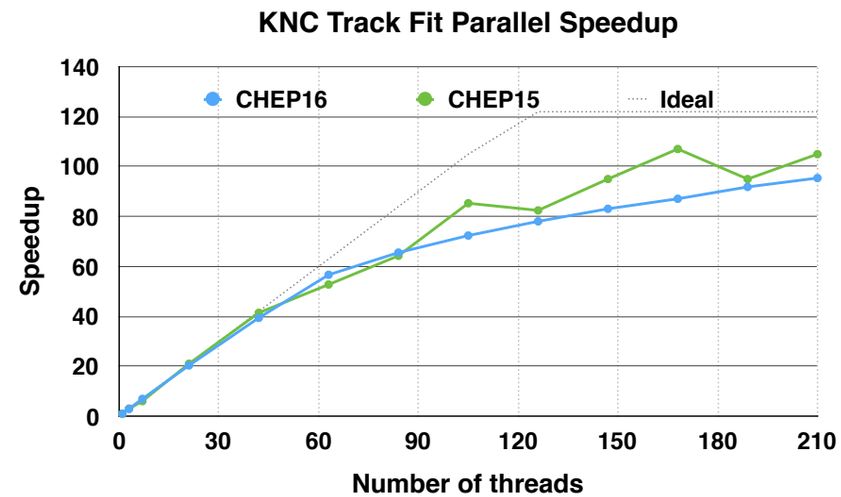
Parallelization: Track Fitting

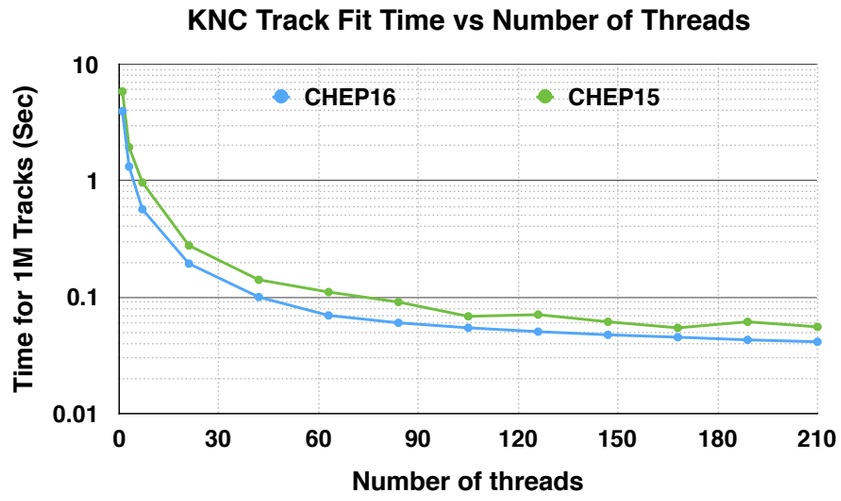
Scaling similar to CHEP2015

- Parallelization near ideal up to 61 threads
- Reach ~100x speedup at ~200 threads
- Ideally ≥122x to occupy available instruction slots
- CHEP2016 faster due to better vectorization

SNB: ideal up to 12 threads, 14x at 24 threads

- Some boost from hyper-threading filling idle slots

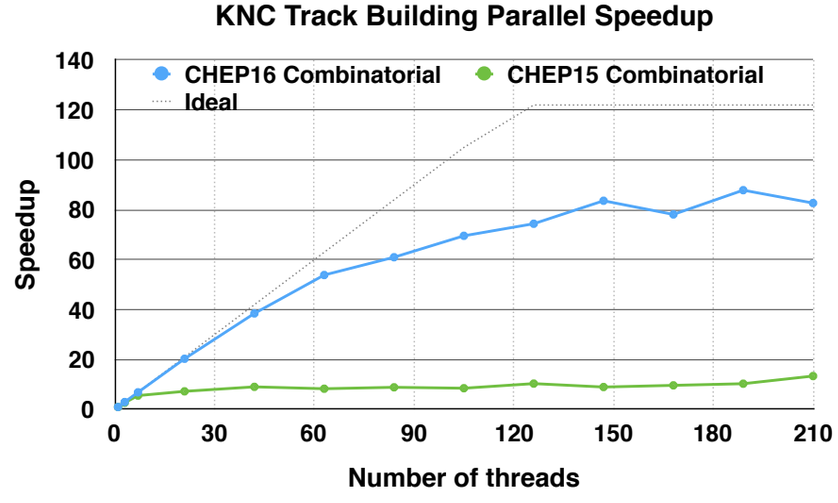




Parallelization: Track Building

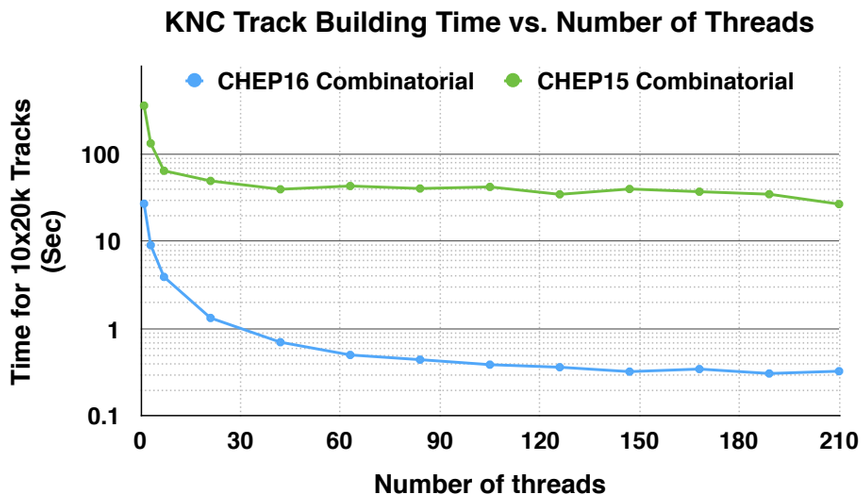
Scaling much better than CHEP2015

- Within 15% of ideal up to 61 threads
- ~85x speedup at ~200 threads
- SNB: near ideal up to 12 threads, 14x at 24 threads



Improvements in both single-thread track building and parallel scaling

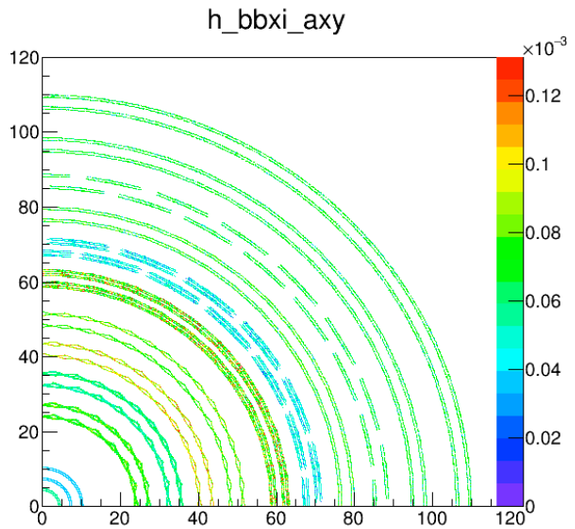
- Single-thread performance ~10x better
- Parallel scaling ~10x better
- Improved physics performance



Adding Realism

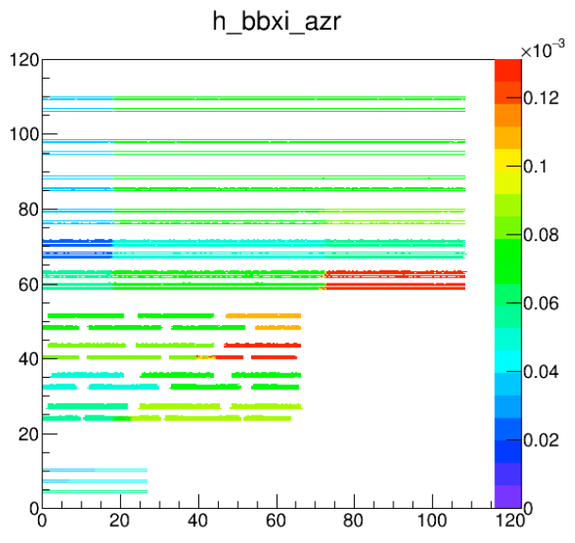
Began with a simplified setup as proof-of-concept as we understood the issues with vectorization and parallelization of the KF

- Ultimately need to include realistic geometry, material effects, inefficiencies, overlaps, etc.
- Use CMS simulation, add complexity in incremental steps



Two step propagation to avoid using the full geometry

- Simple parameterization of CMS geometry and material
- Step 1: propagate to the average radius of the layer
- Step 2: propagate to the exact hit radius

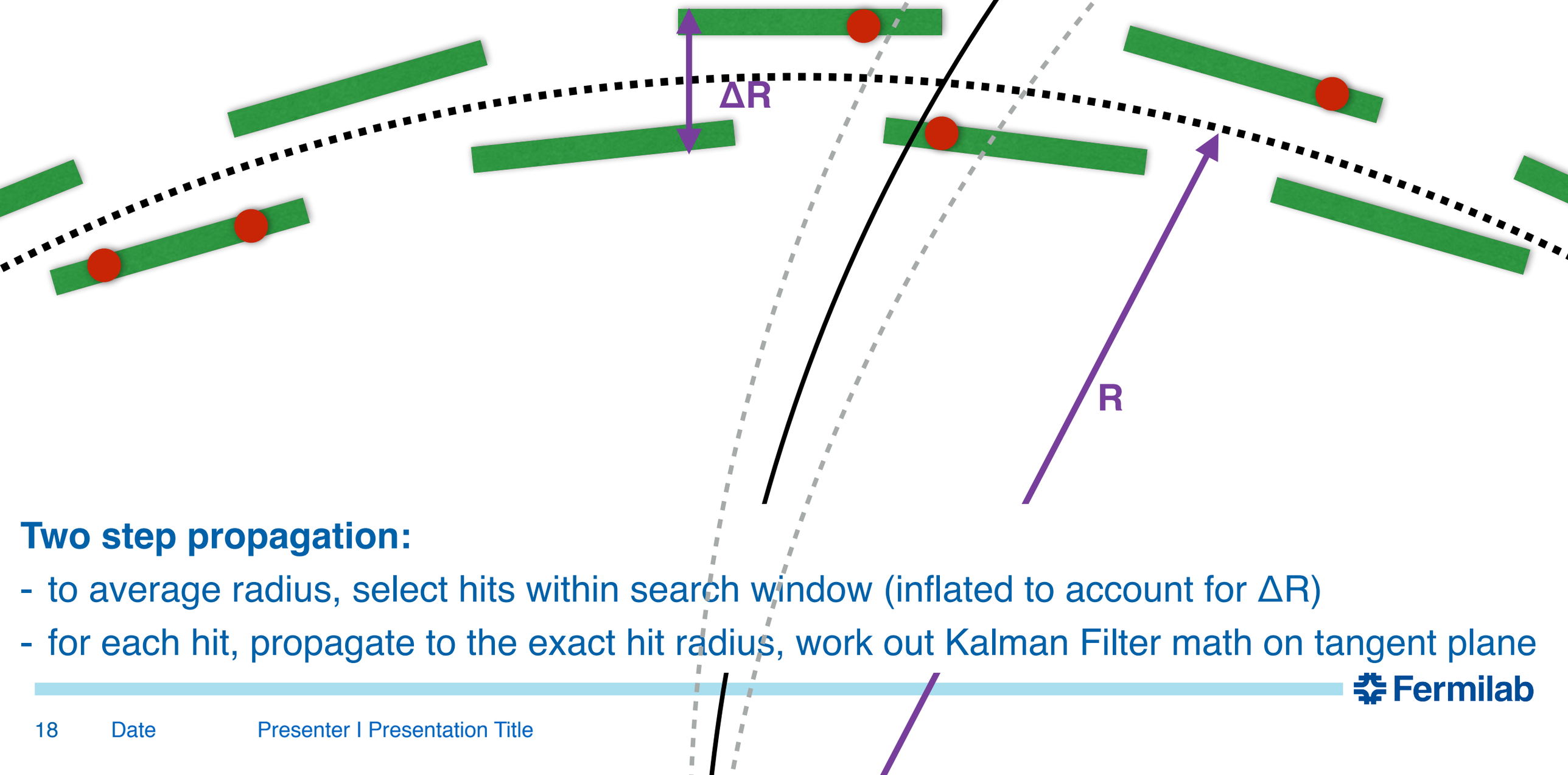


Endcap/Disks

- Propagate to z, similar handling of material and propagation

Barrel layer representation

Barrel layer representation uses only 3 parameters:
average radius R, radius spread ΔR, length Z
=> avoid filling memory with complex geometry structures (position, rotation of each module)
=> almost detector independent



Two step propagation:

- to average radius, select hits within search window (inflated to account for ΔR)
- for each hit, propagate to the exact hit radius, work out Kalman Filter math on tangent plane

CMS Simulation (preliminary)

Early tests with CMS simulation data

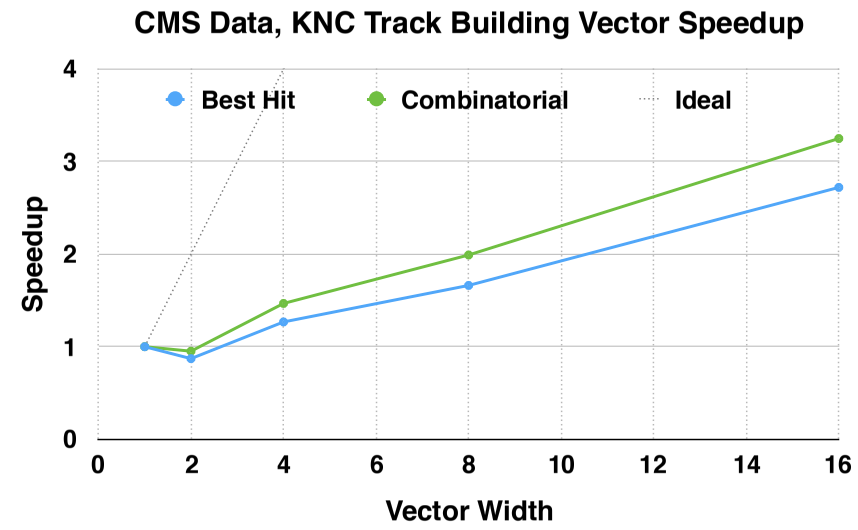
- Hits from full CMS simulation
- Parameterized geometry & material effects

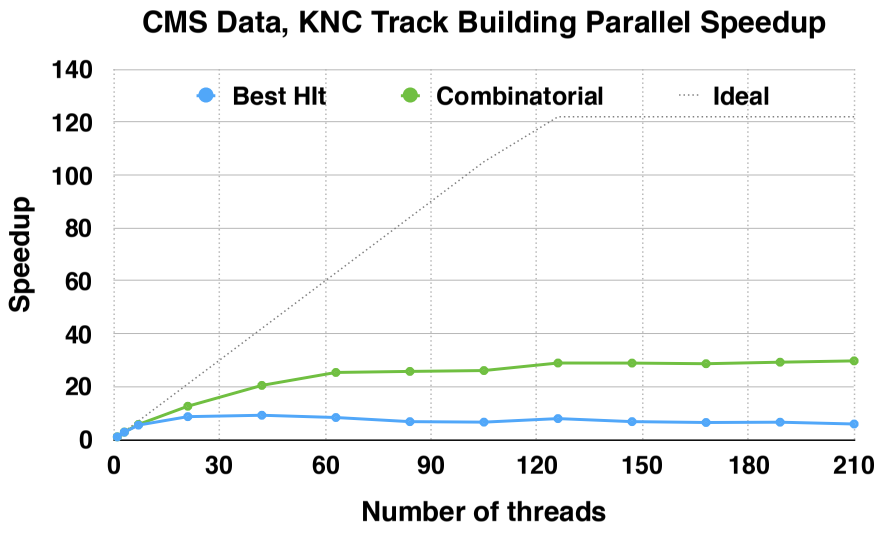
Vectorization is better

- Two-step propagation results in more time spent in well-vectorized routines

Parallelization speedup is worse than toy setup

- Events are smaller than toy events, increasing parallelization overhead
- May need multiple events in flight
- Possibly other effects from more complex geometry





GPU Progress

Implemented fitting routines on Tesla K40/K20

- Best Hit track building in progress

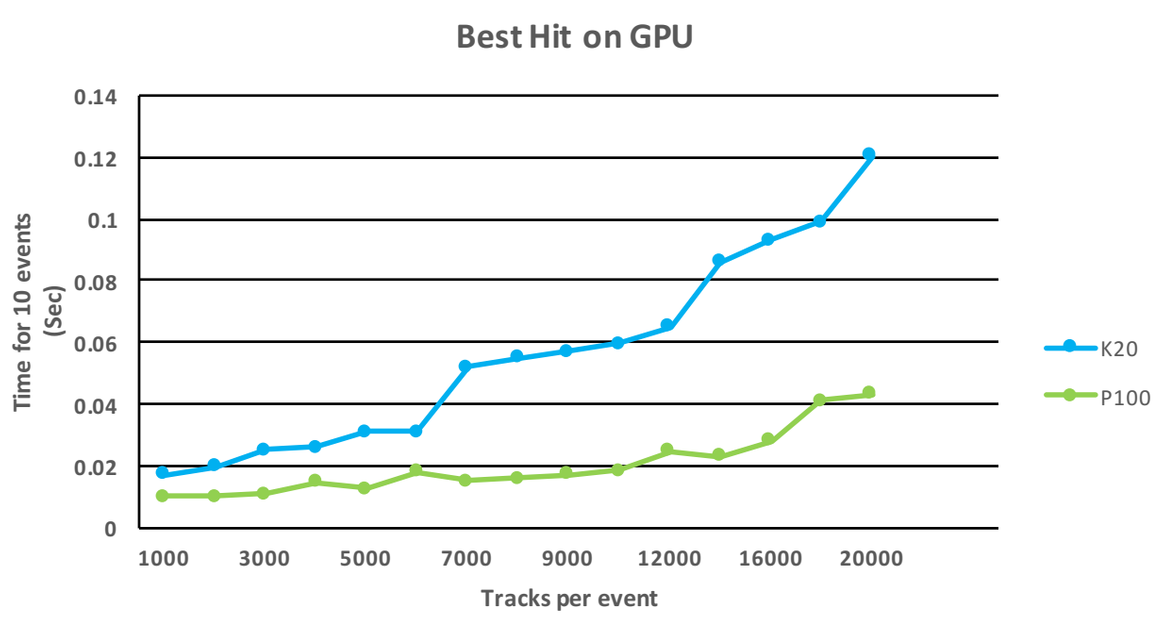
Investigating code sharing

- Define a GPlex class with same interfaces as Matriplex
- Template low-level routines, e.g. propagation, KF

```
template<typename Tf, typename Ti, typename TfLL1,
         typename Tf11, typename TfLLL>
#ifdef __CUDACC__
__device__
#endif
static inline void
helixAtRFromIterative_impl(const Tf& __restrict__ inPar,
                           const Ti& __restrict__ inChg,
                           TfLL1& __restrict__ outPar,
                           const Tf11& __restrict__ msRad,
                           TfLLL& __restrict__ errorProp,
                           int nmin, int nmax)
```

Preliminary tests with new Pascal P100 GPU

- Better scaling behavior
- Simpler memory management



Q&A

- Is the problem of tracking on parallel architectures solved?
  - Only partially
  - Very large parallelization speedup once event size sufficiently large (will need multi-event processing for real life)
  - Moderate vectorization speedup, room for improvement but not easy task

- Is this project ready to reconstruct a full event end to end?
  - Not yet, but the path is well defined
  - Pattern recognition working on barrel or endcap only, transition region being addressed
  - Seeding to be finalized

- Will it work only for CMS or can be easily ported to Atlas/other experiments?
  - because of memory limitations, detector description kept extremely simple, at the level of parametrized detector
  - it should be straightforward to define a different set to parameters to make it work for any other barrel+endcap detector (but ultimate optimizations may require specific tunes)

- How do we want to tackle the problem of the reference algorithm?