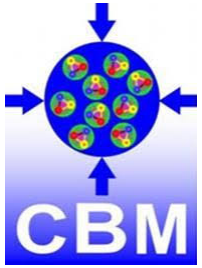


Bundesministerium
für Forschung, Technologie
und Raumfahrt

GOETHE
UNIVERSITÄT
FRANKFURT AM MAIN

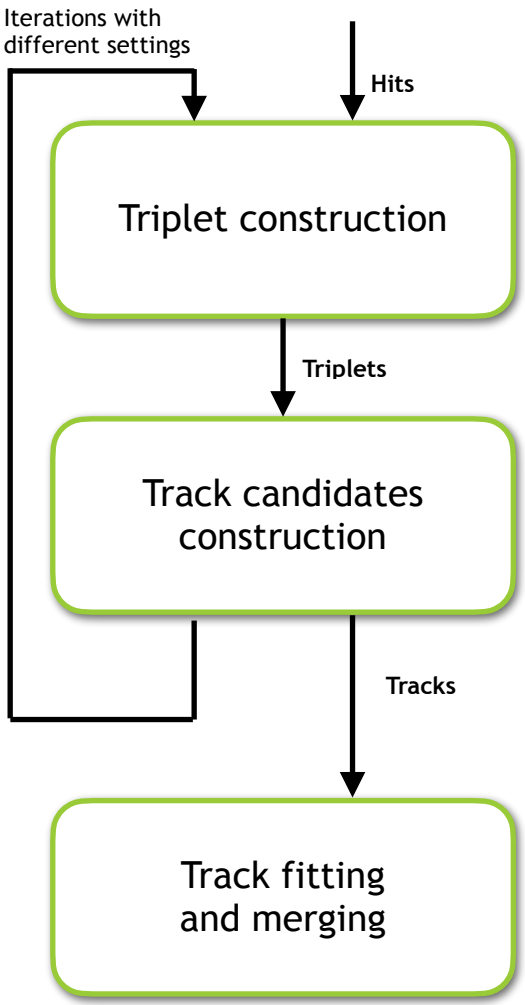
FIAS Frankfurt Institute
for Advanced Studies

Porting the CBM CA Track Finder to GPU

Grigory Kozlov[1,2]

|[1]*Goethe-Universität Frankfurt, Frankfurt am Main, Germany*
|[2]*Frankfurt Institute for Advanced Studies, Frankfurt am Main, Germany*

Cellular Automaton Track Finder in the CBM experiment



CBM CA Track Finder:

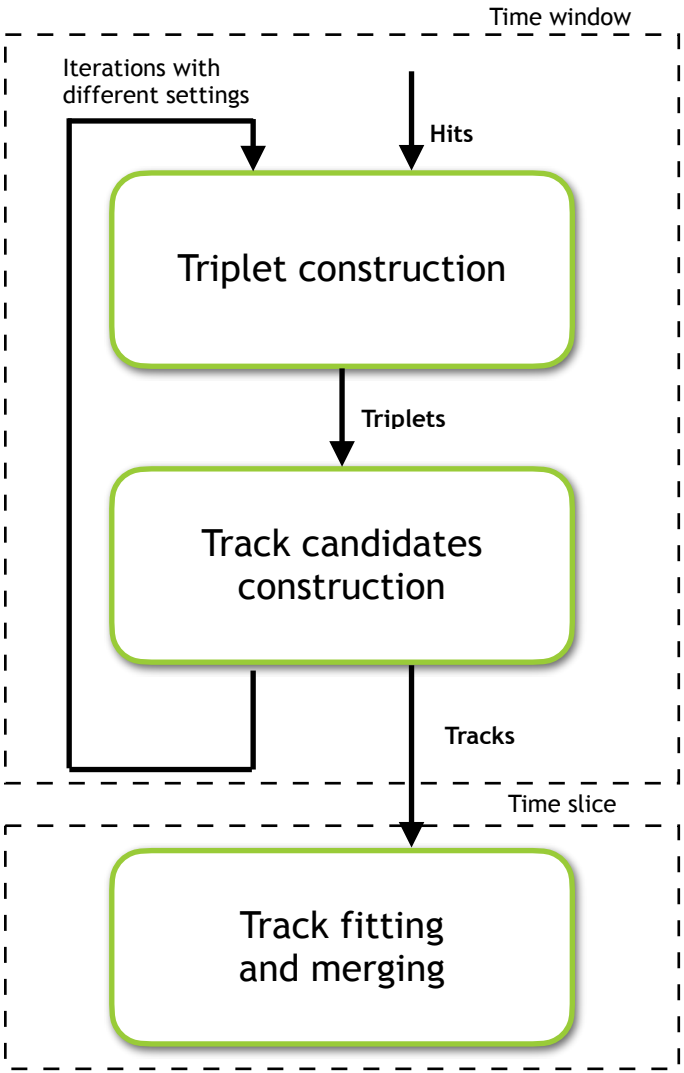
- Locality of the data and calculations.
- Fitting segments and tracks using the Kalman filter.
- Broad potential for data- and task-level parallelization.
- Code optimization both in terms of efficiency and time.
- Using time as an additional dimension when combining hits.

Modular algorithm structure:

- Triplet construction
  - Singlets — hit + vertex — track parameters estimation.
  - Doublets — two hits — approximate direction.
  - Triplets — three hits — curvature and momentum.
- Track candidates construction
  - Track candidates — connected triplets with two common hits.
- Track fitting and merging
  - Tracks — segments that match the track model the most.

Cellular Automaton Track Finder is the best solution for fast track reconstruction as it combines high efficiency and calculation speed, also providing wide opportunities for parallelization.

Porting CA tracking to GPU



Core concepts and principles:

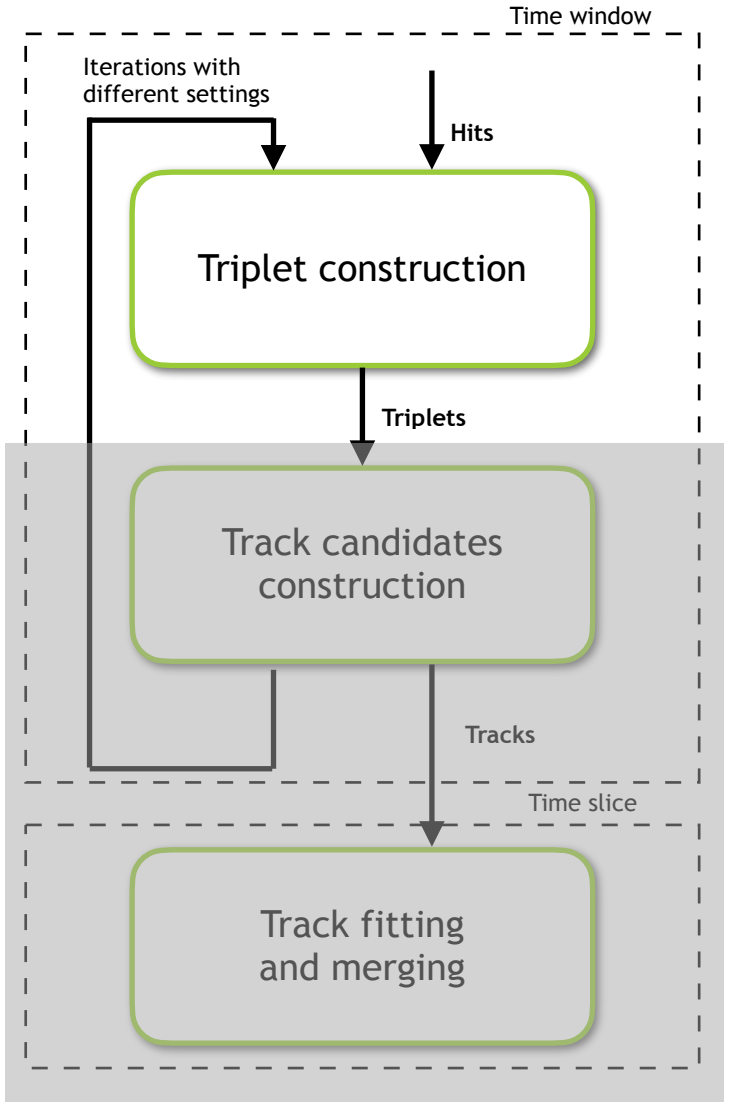
- Unified cross-platform code base;
- Leveraging platform-specific capabilities;
- Modular and maintainable architecture;
- Interchangeable modules with consistent I/O;
- Hybrid CPU-GPU computation support.

Cross-platform flexibility with XPU:

- Lightweight C++ library for GPU development;
- Abstraction layer over CUDA, HIP, and OpenMP backends;
- Unified code compiled for each backend, selected at runtime;
- Optimized GPU algorithms with CPU fallback;
- Integrated timing and memory management.

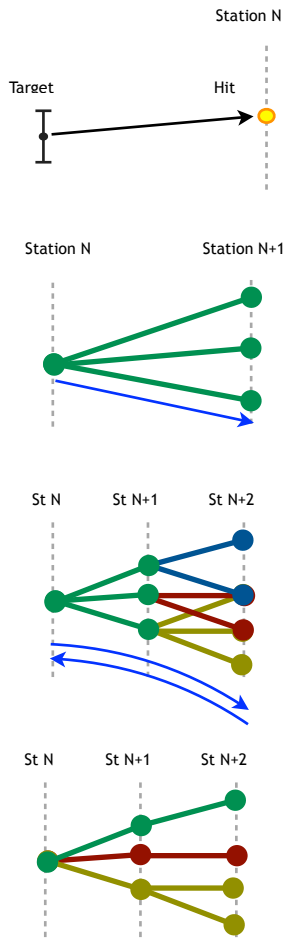
***Concept*: Hybrid CPU-GPU computing with interchangeable modules and consistent I/O.**
***Implementation*: Achieved through an abstraction layer provided by the XPU library.**

Porting CA tracking to GPU - triplets



Triplets reconstruction on CPU

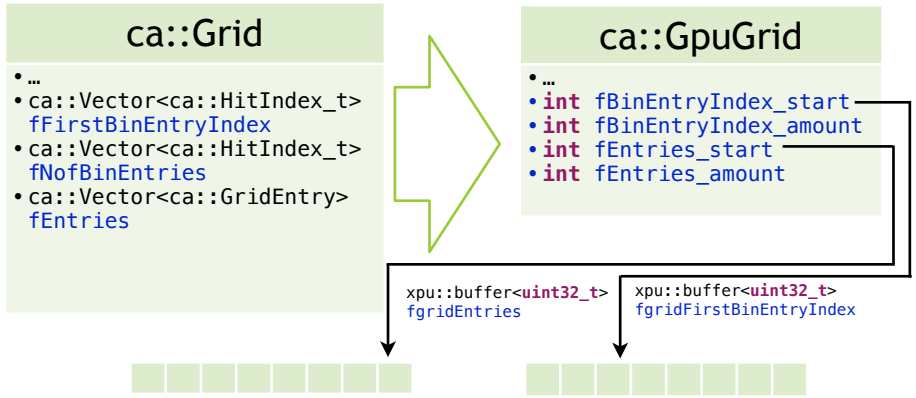
- for loop by stations
  - for loop by hits on station
    - FindSinglets()
    - FindDoublets()
    - FitDoublets()
    - FindTriplets()
    - FitTriplets()
    - StoreTriplets()
  - End of for loop by hits on station
- End of for loop by stations

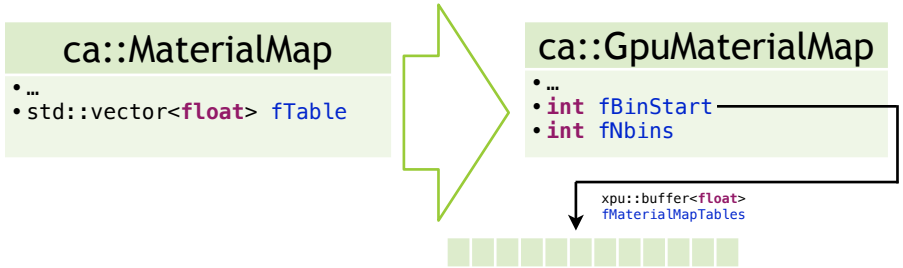


- Creating initial track parameters estimation. Fit a straight line through the target and the (left) hit taking into account magnetic field
- Search for possible combinations of hit pairs
- Fitting forward, adding a second (middle) hit to the parameter estimation
- Search for possible 3-hit segments
- Fitting forward and backward several times to get a final parameters estimation of the triplets
- Checking triplets, saving triplets whose parameters satisfy specified conditions

Sequentially combines hits into doublets and then into triplets. Optional triplet fitting improves track parameter quality but can be skipped.

Data structures on GPU






```
xpu::buffer<hit_doublets> fHitDoublets
• ca::HitIndex_t hit1;
• ca::HitIndex_t hit2;
```

8 bytes per element

```
xpu::buffer<hit_triplets> fHitTriplets
• ca::HitIndex_t hit1;
• ca::HitIndex_t hit2;
• ca::HitIndex_t hit3;
```

12 bytes per element

Considerations for GPU data structures (XPU):

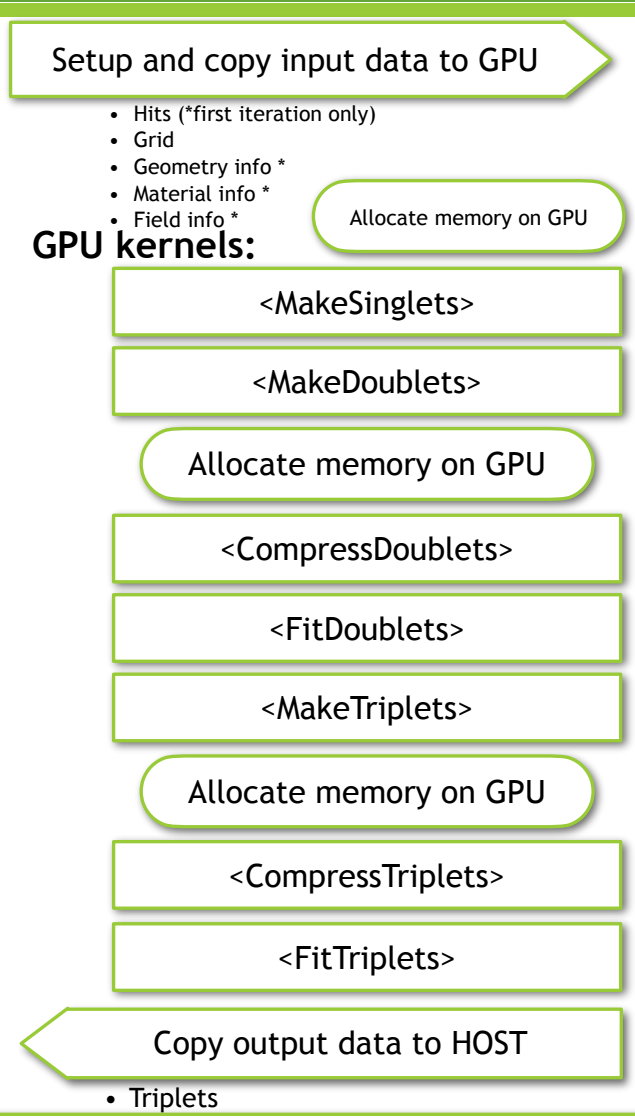
- `std::vector` is not supported;
- Classes with internal dynamic memory allocation cannot be used as storage buffers;
- Dynamic memory allocation for data buffers is not feasible.

Key code modifications:

- Grid data moved to flat arrays; base class keeps only start indices and total count;
- Station material maps organized similarly;
- Doublet memory pre-allocated: nDoublets = nSinglets × maxNDoubletsPerHit (150);
- Triplet memory pre-allocated: nTriplets = nDoubletsFound × maxNTripletsPerDoublet (15);

GPU computations require data structures adapted to the device memory model. Flat arrays in global memory are used instead of nested vectors of variable length.

Triplets reconstruction on GPU



Common GPU/CPU solution for triplet construction based on the XPU framework

- The overall code style follows GPU programming, with an OpenMP-based CPU fallback;
- The procedure is split into separate kernels in GPU style;
- Simultaneous processing of all objects of the same type (singlets, doublets, triplets) instead of sequential doublet and triplet construction for each hit.
- Operations that require GPU-specific functionality are duplicated for CPU;
- Each kernel in the chain can be easily replaced by an alternative, provided the logic and data structures handling are preserved;
- Additional data compression steps are added to improve GPU utilization;
- After computation, results can either be transferred to the host or kept on the GPU for further processing.

The algorithm adopts a unified GPU-oriented design with an OpenMP-based CPU fallback. All objects of the same type are processed simultaneously, ensuring flexible execution and efficient GPU utilization.

Testing and benchmarking

Hardware:

- CPU: 2x Intel Xeon Gold 6130
- GPU: AMD Radeon VII

Dataset:

- 8 STS stations
- Au+Au@10*A*Gev
- Realistic data: mbias (5500 hits), central (11300 hits)
- Stress testing: TS with 5 events (54000 hits), TS with 10 events (92000 hits)

Algorithm:

- 1 iteration - clean results, no cumulative effects
- Old tracking - standard algorithm, 1 thread, SIMDized track fitting
- New tracking - new code, common CPU/GPU (XPU) implementation, running on CPU (1, 4, 8, 16, 32 threads) and GPU
    - Triplet construction - full results
    - Track candidates construction - benchmarking only, no comparison with old tracking
    - Track fitting and clone merging - full results

The benchmarking setup provides a consistent environment for CPU-GPU comparison and evaluates algorithm scalability and execution speed under varying data loads.

Triplet construction speed

1 central collision, ~5500 hits

	Old version	New CPU(XPU) 1 thread	New CPU(XPU) 32 threads	New GPU (XPU)
MakeSinglets (ms)		25.687	6.328	0.083
MakeDoublets (ms)		3.470	1.325	0.553
CompressDoublets (ms)		2.192	3.728	0.313
FitDoublets (ms)		11.221	1.569	0.077
MakeTriplets (ms)		95.903	11.787	0.232
CompressTriplets (ms)		0.892	1.242	0.215
FitTriplets (ms)		239.652	20.263	0.547
Total (ms)	**254.108**	**379.017**	**46.242**	**2.02**

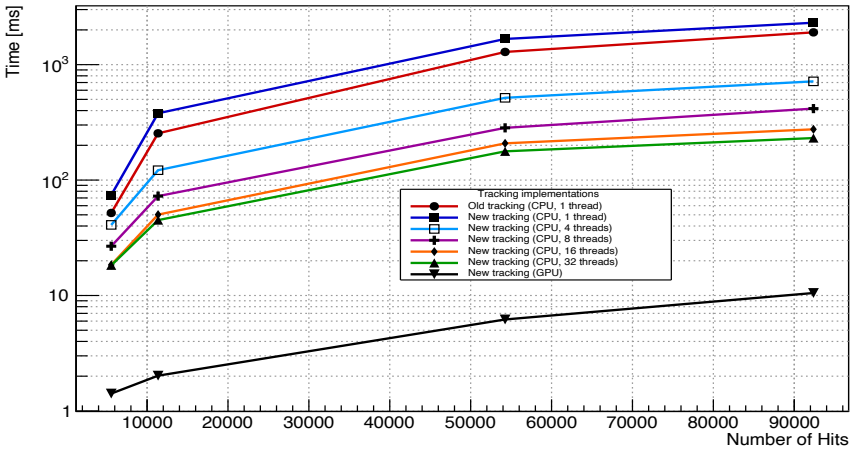
New triplet construction on CPU:

- More time-consuming in single-thread mode than the original implementation;
- Scales with the number of threads, but performance gain saturates at higher thread counts;
- Triplet fitting remains computationally expensive.

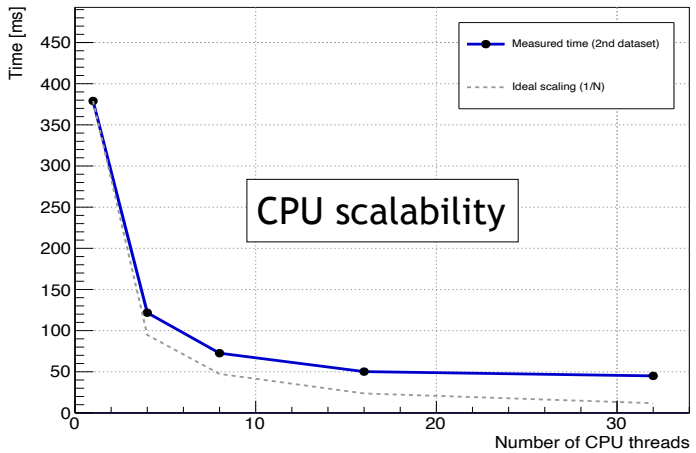
New triplet construction on GPU:

- Achieves multi-fold speedup at every stage of the algorithm;
- Provides up to ×150 improvement over the original version and ×25 over the 32-thread CPU implementation.

Triplet reconstruction total time

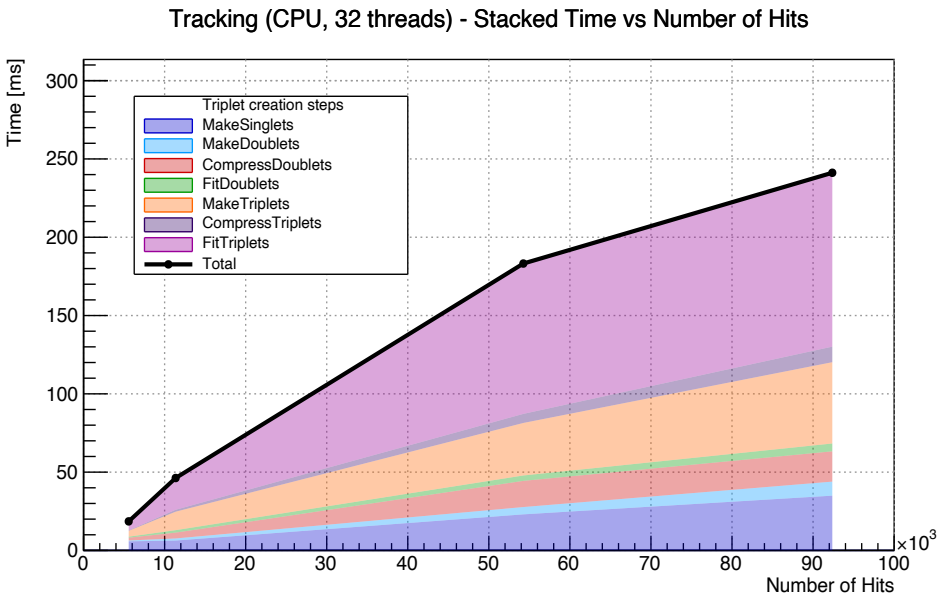


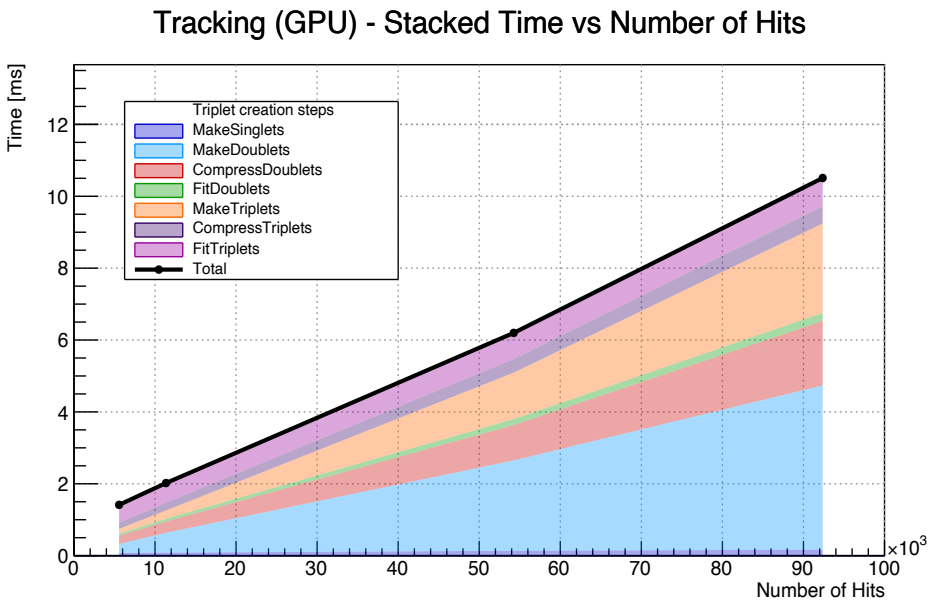
Triplet reconstruction total (2nd dataset)



The GPU version of the algorithm demonstrates a speedup of about ×150 compared to the original implementation and around ×25 relative to the multithreaded CPU fallback.

Triplet construction speed - detailed structure





New triplet construction on CPU:

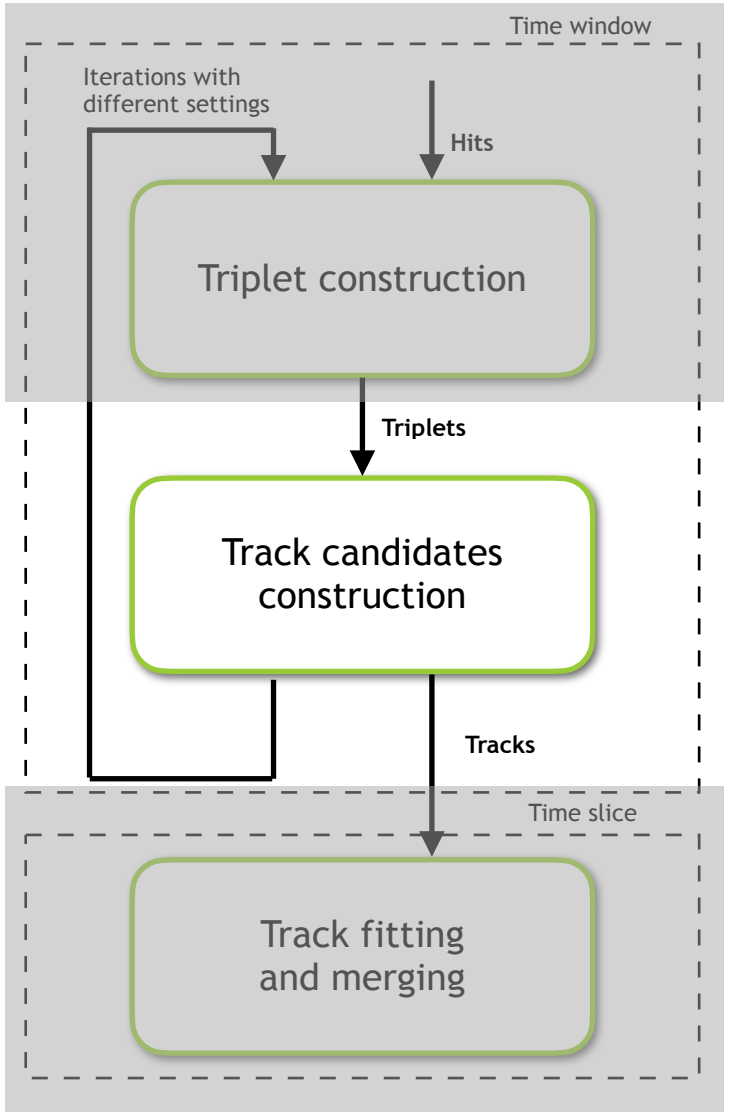
- Triplet fitting is the most computationally expensive stage;
- Singlet and triplet creation also require a significant amount of time;
- The main performance bottleneck is the track parameter extrapolation between stations.

New triplet construction on GPU:

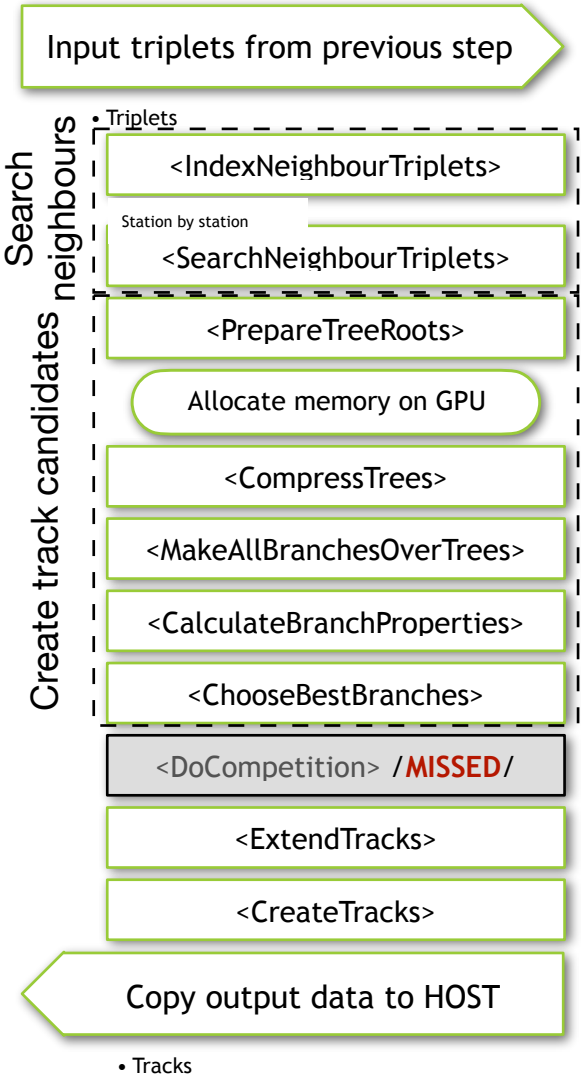
- Triplet fitting is extremely cheap and can be freely used whenever it improves efficiency;
- The main performance bottlenecks are combinatorial steps and long iterative loops;
- Combinatorial operations are still faster than on the CPU.

For the CPU version, the main bottleneck is track parameter extrapolation between stations (KF fitting).
For the GPU version, the dominant bottlenecks are combinatorial operations and long iterative loops.

Porting CA tracking to GPU - track candidates



Track candidates construction



Common GPU/CPU solution for track reconstruction based on the XPU framework

- The kernel structure is more complex than in the pure CPU implementation, as a single recursive function must be replaced by multiple GPU kernels;
- The DoCompetition kernel on GPU is still under development;
- Functions from the "Search Neighbours" and "Create Track Candidates" stages provide only marginal performance gains on GPU due to low computational intensity and irregular memory access patterns;
- The ExtendTracks kernel on GPU demonstrates a significant speedup , as it relies on parallelized fitting;
- The number of track candidates per event remains too small to fully utilize the GPU.

The GPU version replaces recursion with multiple kernels, enabling parallelized track-candidate search. However, the small number of tracks results in low computational intensity and prevents full GPU utilization.

Track candidate construction speed - benchmark

1 central collision, ~5500 hits

	New CPU(XPU) 1 thread	New CPU(XPU) 32 threads	New GPU (XPU)
IndexNeighbourTriplets (ms)	0.357	0.072	0.047
SearchNeighbourTriplets (ms)	0.850	0.175	0.465
Create track candidates (4 kernels) (ms)	0.412	0.181	1.122
ExtendTracks (ms)	5.807	1.442	0.405
Total (ms)	**7.426**	**1.870**	**2.039**

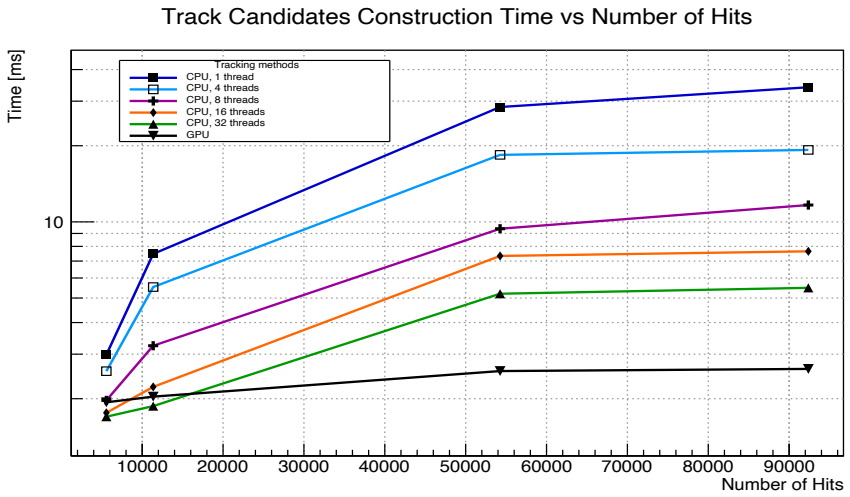
- Benchmark run without DoCompetition stage.
- The number of tracks is critically insufficient for effective GPU utilization.

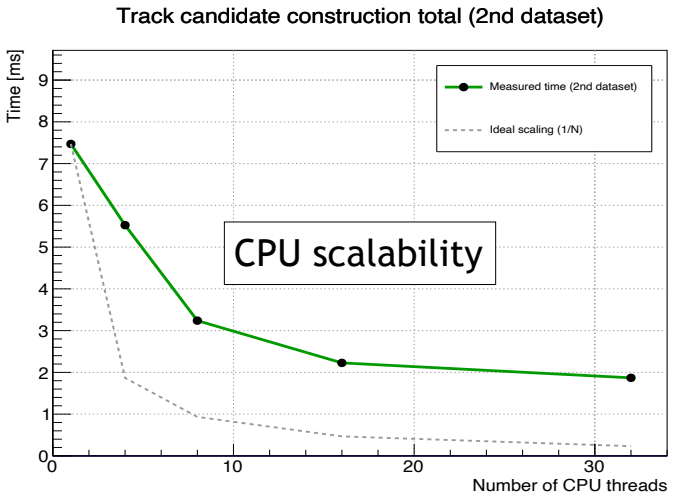
New track candidate construction on CPU:

- Scales with the number of threads, but performance gain remains well below the theoretical maximum;
- Outperforms the GPU in combinatorial operations with random memory access.

New track candidate construction on GPU:

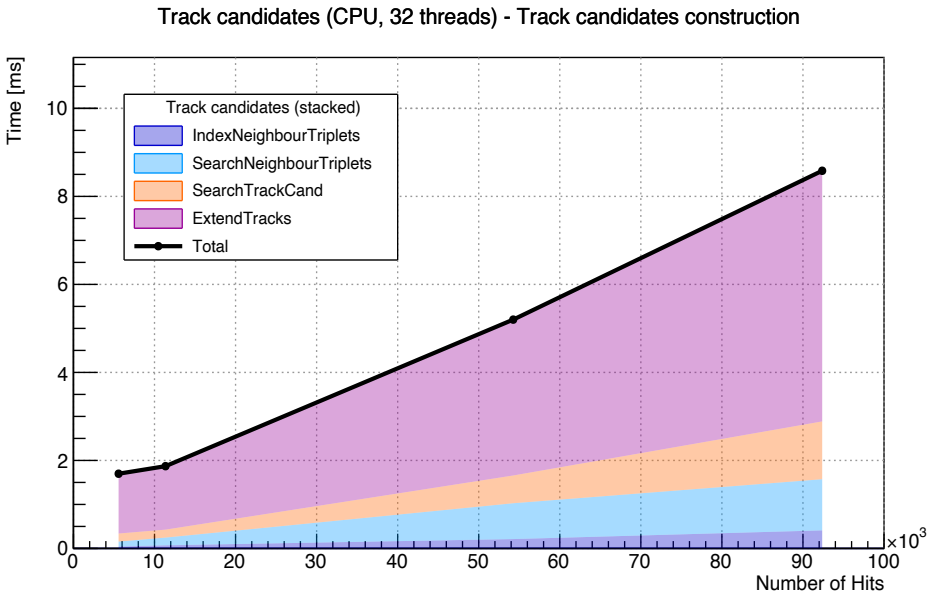
- Outperforms the CPU in track-candidate extension, as this step relies on parallelized fitting.

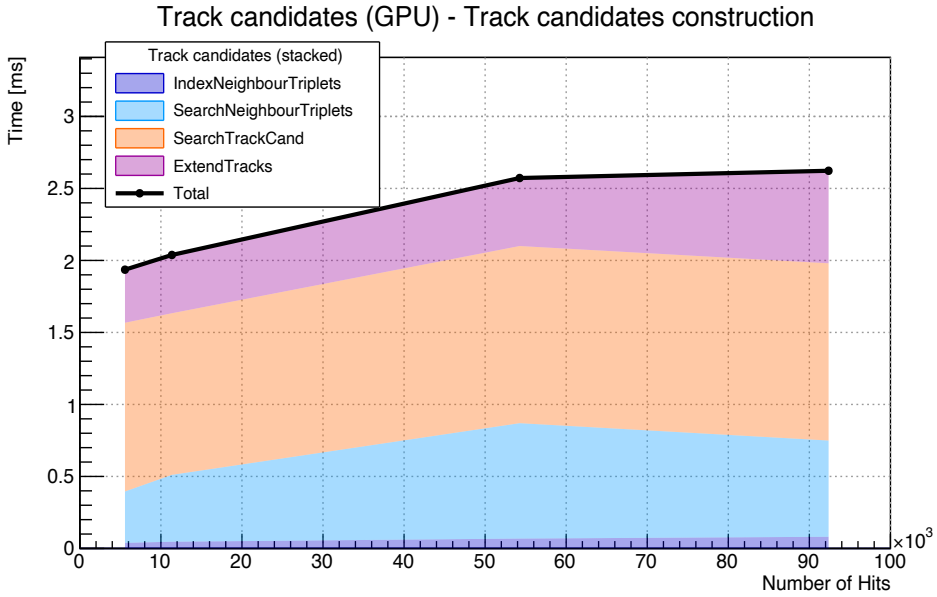




Track-candidate creation **is inefficient on GPU for small data sets.** ***Track extension*** **performs faster on GPU than on CPU, as it is based on parallelized fitting, which constitutes the core part of this function.**

Track candidate construction - detailed structure





New track candidate construction on CPU:

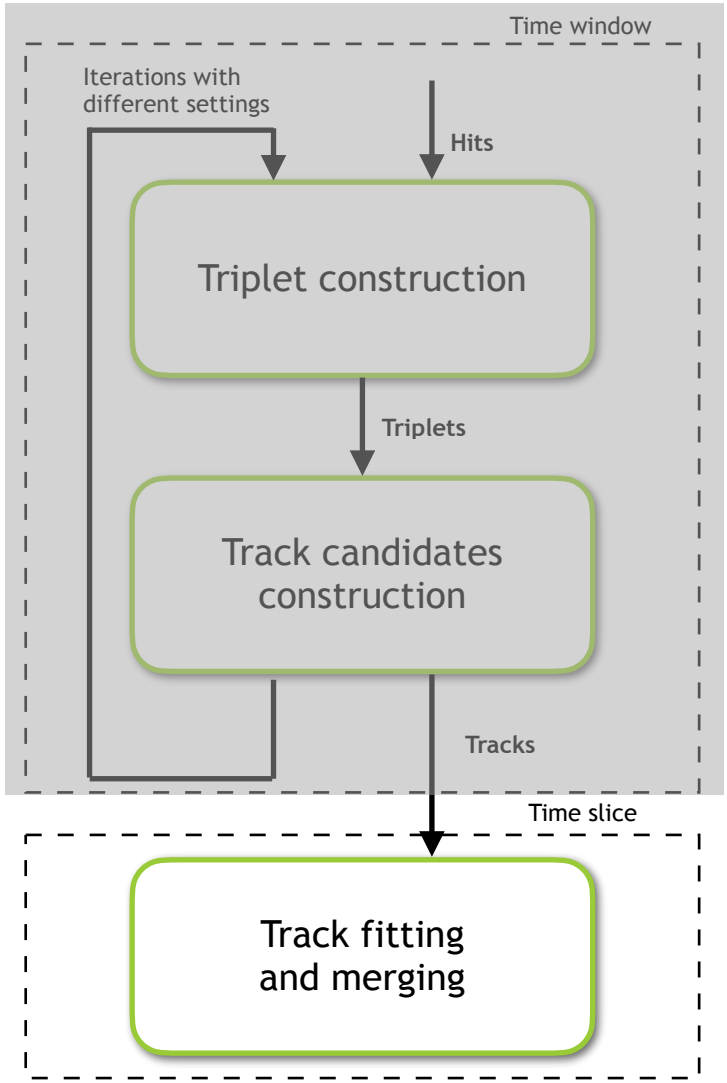
- Extension of track candidates takes the largest share of the total execution time;
- Execution time of all stages increases steadily with the number of triplets and track candidates.

New track candidate construction on GPU:

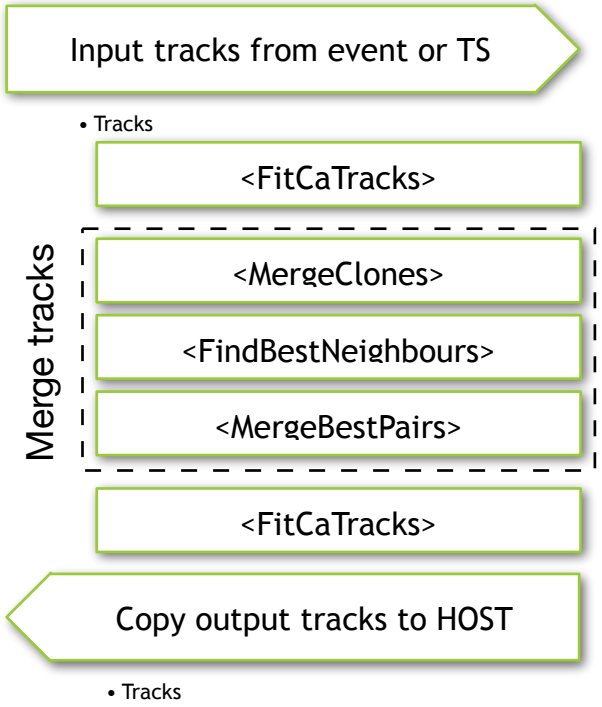
- Execution time of each stage changes only slightly with increasing numbers of triplets and track candidates;
- Track candidates extension is particularly efficient, as it is based on parallelized fitting;
- The track count is too low to fully utilize GPU resources, making full GPU usage for this task inefficient.

On CPU, execution time grows steadily with data size, with track extension dominating the total runtime.
On GPU, it grows more slowly since the track count is too low to fully load the device.

Porting CA tracking to GPU - tracks



Tracks fitting and merging



Common GPU/CPU solution for track fitting amd merging based on the XPU framework

- The Kalman filter-based track fitter is highly suitable for GPU parallelization and demonstrates excellent computational performance;
- Unlike the pure CPU implementation, the CPU fallback of the GPU code loses the benefits of SIMD vectorization and may run slower in single-threaded mode;
- The track-merging procedure is divided into three kernels to fully utilize GPU resources, which can make the CPU fallback slower than the pure CPU version;
- Each candidate-pair combination is processed by a separate thread instead of a one-to-many loop.

The Kalman filter-based track fitting efficiently uses GPU parallelism, achieving high performance. Track merging benefits from GPU execution, though its CPU fallback runs slower due to GPU-oriented design.

Track fitting and merging speed

1 central collision, ~5500 hits

	Old version	New CPU(XPU) 1 thread	New CPU(XPU) 32 threads	New GPU (XPU)
Fit Tracks (ms)	4.676	18.841	1.751	0.789
Merge Clones (ms)	4.746	9.295	2.788	0.366
Fit Tracks (ms)	4.519	18.650	1.645	0.557
Total (ms)	**13.941**	**46.786**	**6.184**	**1.712**

- The number of tracks is critically insufficient for effective GPU utilization. Processing multiple events at once improves resource utilization.

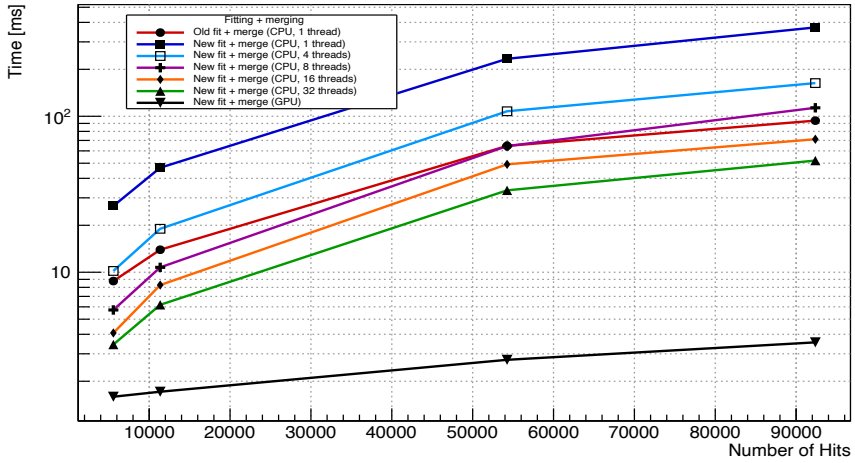
New fitting and merging on CPU:

- Fitting is slower than in the original version, as it is not SIMDized;
- Merging is slower than in the original version, as it is optimized for GPU execution.

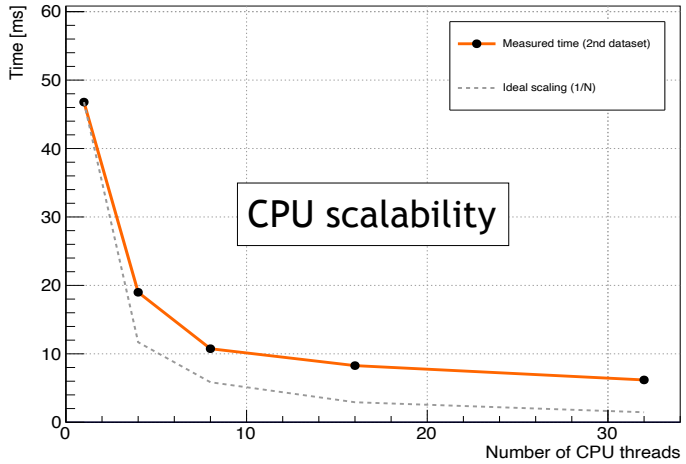
New fitting and merging on GPU:

- Demonstrates a speed advantage even for a small number of tracks.

Fitting + Clone merging total time

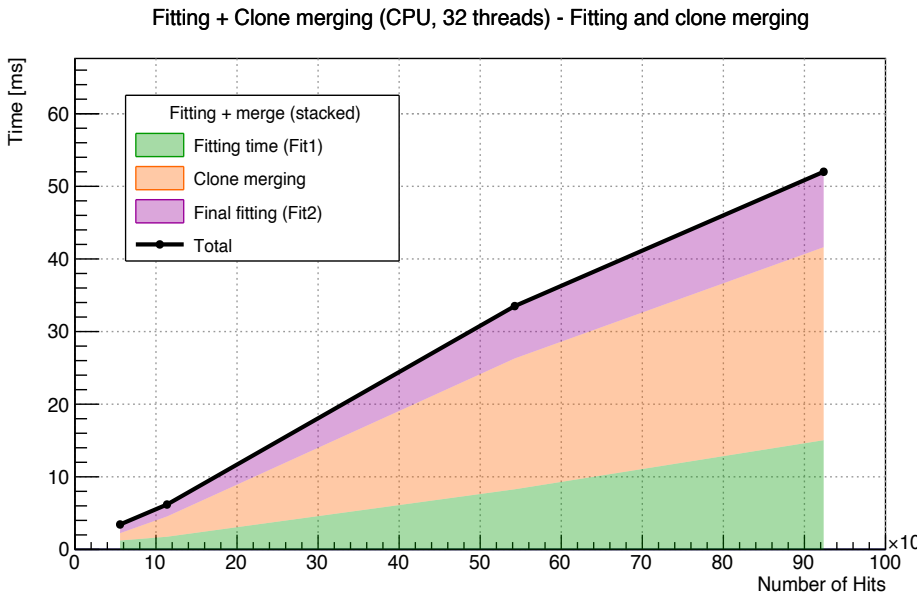


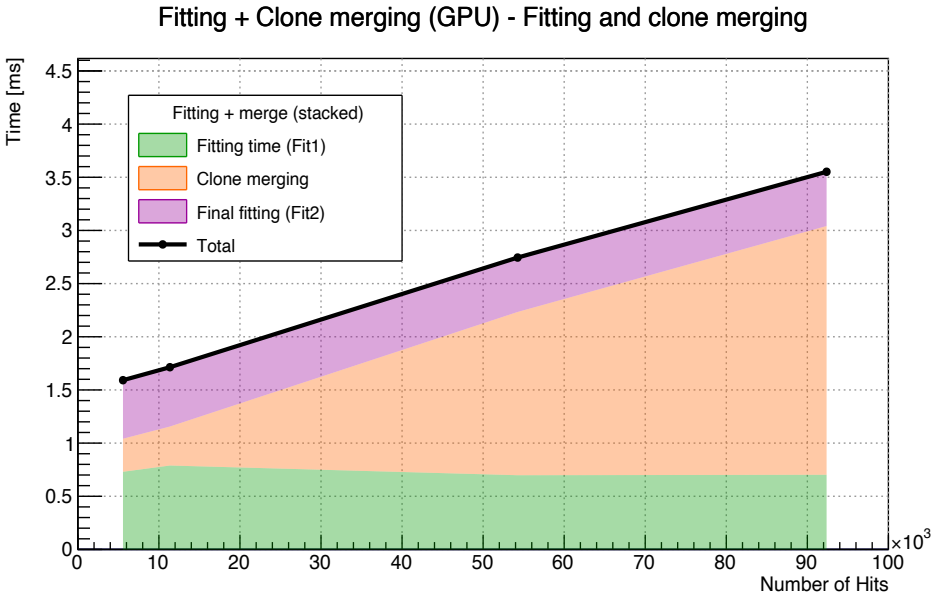
Fitting + clone merging total (2nd dataset)



The GPU version shows a clear speed advantage even with few tracks.
Processing multiple events simultaneously would allow better utilization of GPU resources.

Track fitting and merging speed - detailed structure





New track fitting and merging on CPU:

- Execution time of all stages increases steadily with the number of tracks.
- Merging shows a less favorable performance trend, as the function is optimized for GPU rather than CPU execution.

New track fitting and merging on GPU:

- Fitting speed remains nearly constant, as the number of tracks is insufficient to fully utilize GPU resources;
- The CloneMerger efficiently utilizes available GPU resources even with a small number of tracks, processing each track pair in a separate thread instead of long, costly loops.

On CPU, execution time grows with track count, and merging is slower due to GPU-oriented optimization.
On GPU, fitting time stays nearly constant, while CloneMerger efficiently uses resources even with few tracks.

Conclusion

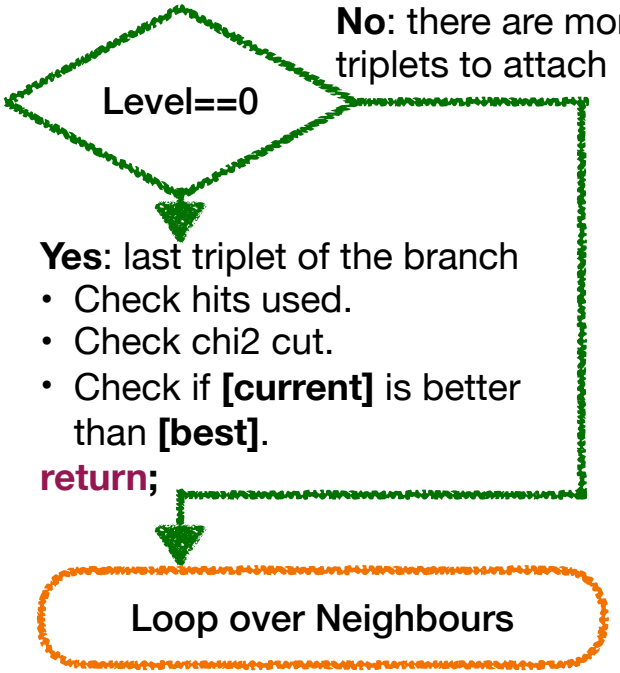
- Within the GPU porting of the CA Track Finder, almost all algorithm stages have received GPU implementations with parallelized CPU fallbacks based on OpenMP;
- The algorithm is structurally divided into three logical parts, each of which can run on either CPU or GPU, allowing mixed-mode execution;
- Porting of the track-candidate competition stage is in progress;
- Triplet construction on GPU demonstrated a significant performance improvement: about ×150 compared to the original implementation and around ×25 relative to the multithreaded CPU fallback;
- Track-candidate reconstruction on GPU is currently inefficient for small track counts;
- Fitting and merging show good performance even with a limited number of tracks.
- **Next step**: porting the Kalman Filter Particle Finder to GPU.

Backup

Track candidates reconstruction on CPU and GPU

CPU

CAFindTrack(**[best]**, **[current]**, **[new]**)



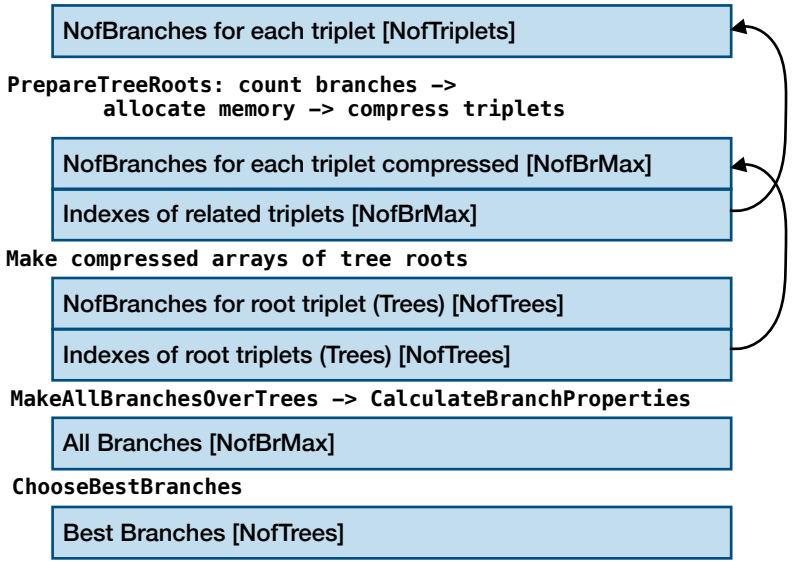
CAFindTrack(**[best]**, **[new]**, **[new]**)

GPU

No recursion —> Build all candidates, get best:

- No std::vector —> preallocate memory!
- How many candidates? Count it!
- How to identify branches from the same root? Add indexes!
- How to traverse trees and build all branches without recursion? Use Iterative stack!

Memory model to replace std::vector

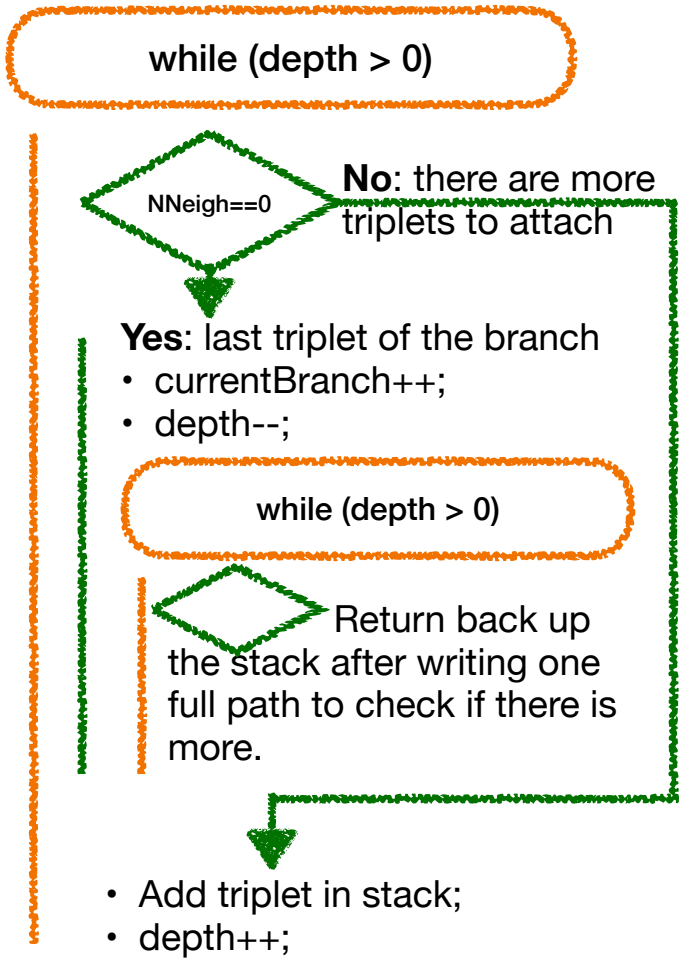


NofTrees < NofBranchesMax < NofTriplets

Track candidates reconstruction on GPU

Iterative Stack

```
int stack[constants::gpu::MaxHitsPerTrack - 2];
stack[0] = rootIndex;
int depth = 1;
```



Predefined data

Number of possible branches for each triplet.
- Calculated during neighbour triplets searcing.

Prepare memory and indexing

`PrepareTreeRoots`
- Prepare list of roots for triplet trees.
- Count ammount of trees and max branches.

Allocate memory for data and index arrays.

`ExclusiveScan [4 kernels]`
- Compress NofBranch data (GPU).

`ExclusiveScanTree [3 kernels]`
- Prepare branch to trees mapping data (GPU).

Calculate branches

`MakeAllBranchesOverTrees`
- Traverse trees with Iterative Stack (LIFO).

`CalculateBranchProperties`
- Calculate properties of all branches and apply cuts.

`ChooseBestBranches`
- Choose best branche in every tree.