Generalizing mkFit and its Application to HL-LHC

Giuseppe Cerati[4], *Peter* Elmer[2], *Patrick* Gartung[4], *Leonardo* Giannini[1], *Matti* Kortelainen[4], *Vyacheslav* Krutelyov[1], *Steven* Lantz[3], *Mario* Masciovecchio[1], *Tres* Reid[3], *Allison* Reinsvold Hall[5], *Daniel* Riley[3], *Matevž* Tadel[1,*], *Emmanouil* Vourliotis[1], *Peter* Wittich[3], and *Avi* Yagil[1] on behalf of the CMS collaboration

[1]UC San Diego, La Jolla, CA, USA 92093
[2]Princeton University, Princeton, NJ, USA 08544
[3]Cornell University, Ithaca, NY, USA 14853
[4]Fermilab, Batavia, IL, USA 60510-5011
[5]US Naval Academy, Annapolis, MD, USA 21402

Abstract. mkFit is an implementation of the Kalman filter-based track reconstruction algorithm that exploits both thread- and data-level parallelism. In the past few years the project transitioned from the R&D phase to deployment in the Run-3 offline workflow of the CMS experiment. The CMS tracking performs a series of iterations, targeting reconstruction of tracks of increasing difficulty after removing hits associated to tracks found in previous iterations. mkFit has been adopted for several of the tracking iterations, which contribute to the majority of reconstructed tracks. When tested in the standard conditions for production jobs, speedups in track pattern recognition are on average of the order of 3.5x for the iterations where it is used (3-7x depending on the iteration). Multiple factors contribute to the observed speedups, including vectorization and a lightweight geometry description, as well as improved memory management and single precision. Efficient vectorization is achieved with both the icc and the gcc (default in CMSSW) compilers and relies on a dedicated library for small matrix operations, Matriplex, which has recently been released in a public repository. While the mkFit geometry description already featured levels of abstraction from the actual Phase-1 CMS tracker, several components of the implementations were still tied to that specific geometry. We have further generalized the geometry description and the configuration of the run-time parameters, in order to enable support for the Phase-2 upgraded tracker geometry for the HL-LHC and potentially other detector configurations. The implementation strategy and high-level code changes required for the HL-LHC geometry are presented. Speedups in track building from mkFit imply that track fitting becomes a comparably time consuming step of the tracking chain. Prospects for an mkFit implementation of the track fit are also discussed.

1 Introduction

The mkFit project was started in 2014 with the goal of exploring how the traditional Kalman filter based track fitting and track finding [1] can be rethought and optimized in the age of – at

*e-mail: mtadel@ucsd.edu

that time, novel – many-core, vectorized computing architectures. After initial positive results on simplified detector geometries the focus was shifted to applying the mkFit algorithm to silicon detector track finding for the CMS experiment [2], resulting in a viable prototype in 2018 and culminating in a final one-year integration and validation campaign in 2021. Since the beginning of LHC Run3 in 2022 [3], mkFit is used by CMS to reconstruct five out of twelve tracking iterations covering 90% of found tracks with $p_T > 0.5\,\mathrm{GeV/c}$. An in-depth review of mkFit, including detailed motivation and algorithm description, has been published [4]. An overview of early work with further references can be found in [5] and physics performance of mkFit in CMS Run3 is available as a CMS Detector Performance note [6].

This paper focuses on improvements and extensions of mkFit that were required to support running of multiple tracking iterations in *CMS software* (CMSSW) as well as to prepare it for implementation of tracking after the Phase-2 detector upgrades expected around 2030, in the *High Luminosity LHC* (HL-LHC) era. Section 2 introduces how mkFit is structured and run within CMSSW. A detailed presentation of required generalizations of geometry description, configuration and steering systems is given in section 3. Currently ongoing and planned or possible future work is discussed in section 4.

2 mkFit in CMSSW

mkFit was initially developed as a standalone tracking library and at first included into CMSSW as an external package. This mode of operation was used for development, physics performance tuning and benchmarking. However, one of the conditions for using mkFit in production was for the code to be incorporated into the core CMSSW distribution, to make the software building, configuration, and patching for online and offline use compatible with CMS's requirements for computing operations. This section discusses the high-level code structure of mkFit in CMSSW; outlines steps performed by mkFit in a typical CMSSW reconstruction job; and, finally, presents some highlights of physics and computing performance.

2.1 Code structure

mkFit code is structured into three CMSSW packages:

- **`MkFitCore`** holds the central components of mkFit, including all computational algorithms, internal data formats and geometry description, as well as configuration structures and related processing code. This core package is independent of any experiment or geometry details. It does not depend on or interact directly with any CMSSW modules or data formats.
- **`MkFitCMS`** contains helper algorithms, called *standard functions*, that perform specific tasks during track finding: seed pre-processing, candidate scoring, candidate filtering, and duplicate removal. These codes use mkFit internal data formats and still do not depend on CMSSW.
- **`MkFit`** is the actual bridge between mkFit and CMSSW. It defines the CMSSW producer modules for both configuration and data-processing. It uses CMSSW specific mechanisms to pull in configuration and event-data, transforms them into mkFit internal structures and calls appropriate steering functions. It depends both on `MkFitCore` (data-formats and geometry) and `MkFitCMS` (steering and standard functions) packages.

Standalone operation of mkFit is still possible and is frequently used for validation, tuning, development, and debugging. To support this mode, packages `MkFitCore` and `MkFitCMS` contain additional code and makefiles in sub-directory `standalone/` that is not used by

CMSSW build or touched by CMSSW code managements tools. This allows for keeping all mkFit related files stored in a single repository. A minimal additional repository with external packages that would otherwise be used from CMSSW or CMSSW's external software still needs to be maintained separately for standalone builds.[1]

2.2 Track finding algorithm

Details of CMS track reconstruction and iterative tracking can be found in [7]. Here we are concerned with processing as it occurs for every iteration after the seed tracks have been found. Input to track finding is a vector of seed tracks, each consisting of a list of associated hits and the initial estimate of the track parameters at the final, outermost point. After that, mkFit processing steps are as follows:

1. *Seed cleaning* is performed. As mkFit processes seeds in parallel it can not rely on hit masking in order to exclude seeds whose hits have already been consumed by previously found tracks.
2. *Seed partitioning* reshuffles the seeds into *tracking regions* (barrel, transition, and endcap). Those define the sequence in which detector layers will be visited. Additionally, the seeds are sorted in η, φ-space to improve hit access coherency during later steps.
3. *Forward search* proceeds through the detector layers for the given tracking region going outwards from the seeding layers. For each seed, combinatorial search with a limit on the maximum concurrent number of candidates is performed, adding new hits on each layer while allowing for a limited number of missed layers and a single additional "detector overlap" hit. At the end, either because the edge of the tracker is reached or because no more new hits are found, the best-scoring candidate is chosen as the representative. Optionally, a quality filter can be applied before the next step.
4. *Backward fit* re-traverses each found track backwards, refitting the track parameters. If the seeding region for the current iteration does not extend all the way to the vertex region, a combinatorial *backward search* can also be performed, going inwards from the first known hit.[2] Again, the search is stopped when the innermost layers of the detector have been reached or if no new hits are found for a given seed. If the search has been performed, the best candidate is chosen as the final representative.
5. *Quality filtering & duplicate removal* are performed on the resulting tracks.

After the track finding for each iteration is complete, two more steps are performed by other CMSSW modules: final fit including outlier rejection; and final quality selection, based on a multivariate algorithm.

In standalone operation, the same mkFit processing steps are performed. Pre-processed input seeds and hit data are read from a custom binary file. Final fit and quality selection are not performed, but there is a standalone version of validation comparing the found tracks to simulated ones.

2.3 Physics and computational performance

Here we present two highlights from the CMS Detector Performance note [6].[3] Both cases compare relevant quantities before and after the inclusion of mkFit in the standard Run3 track reconstruction of simulated $t\bar{t}$ events with an average pile-up of 65.

[1] https://github.com/trackreco/mkFit-external

[2] Optionally, some or all of the seed-region hits can be dropped and new hits searched for in the seeding layers as well.

[3] In this note mkFit has also been used for PixelLess iteration that was later removed due to poor performance for low-momentum highly displaced tracks.

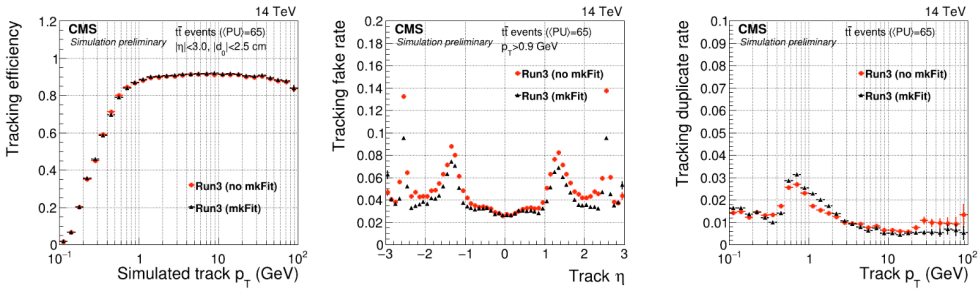

Figure 1. Comparison of physics performance of legacy track reconstruction (red markers) and the new Run3 tracking configuration where mkFit is used for 6 of twelve tracking iterations (black markers): From left-to-right: a) tracking efficiency, b) fake rate, and c) duplicate rate.

Figure 1 shows comparisons of basic physics performance markers. Tracking efficiency is comparable overall; efficiency vs. η (not shown) indicates small gains in the endcap region ($2.4 < |\eta| < 2.8$). Fake rate is improved overall with reduction improving with increasing $|\eta|$. Duplicate rate is slightly increased but has been subsequently improved with further iteration-specific tuning of the duplicate removal algorithm.

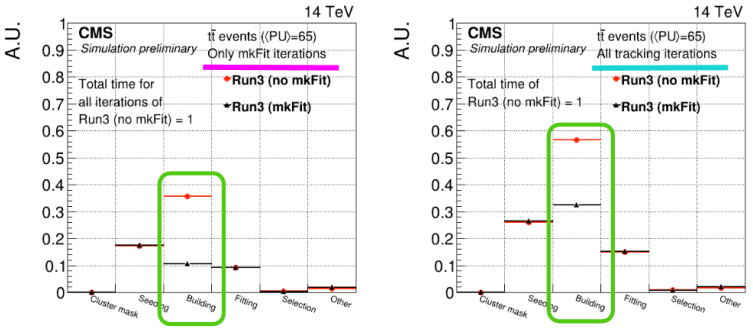

Figure 2. Comparison of computational performance of legacy track reconstruction (red markers) and the new Run3 tracking configuration where mkFit is used for 6 of twelve tracking iterations (black markers). Plots show relative times of tracking steps for (left) iterations that use mkFit and (right) all iterations.

Vectorization and threading scaling tests for initial iteration imply that, according to Amdahl's Law, ~70% of operations are vectorized and that more than 95% of code is effectively parallelized. Computational speedups when using mkFit are shown in figure 2. For all iterations where mkFit is used, the observed track building time is reduced by ~3.5x (the best observed reduction for one mkFit iteration is 6.7x). Note that track building with mkFit takes less time than seeding, and about the same time as the final fit.

When all iterations, including non-mkFit based ones, are considered the building time is reduced by ~1.7x. This translates to a 25% reduction of total tracking time (including seeding, and final fit) and, overall, results in a 10-15% increase of Run3 reconstruction job event throughput.

3 Generalizations for iterative tracking & HL-LHC

CMSSW is a multi-threaded, module-based event processing framework that instantiates and runs modules as well as manages data sources (both event data and longer-lived data products) according to the dependencies generated by the modules themselves when the job configuration is processed. As such, each module can be instantiated multiple times and associated with different configurations and data-sources, including parallel processing of several events. This requires complete separability of configuration and module instance state as well as complete absence of any non-constant global state. Further, in the mkFit case, each iteration has its own set of parameters that control and steer the functioning of core tracking algorithms as well as separate implementations of standard functions, as mentioned in section 2.1.

While basic, algorithmic modifications had to be made to make mkFit conform to the iterative tracking of CMSSW — i.e., to support forward search, backward fit, and backward search — the majority of these changes amounted to generalizations of the algorithms, along with mechanisms for expressing different modes of behavior through configuration structures and intermediate-level code that steers the algorithms. The previous section dealt with how the code is structured and how it operates; this section addresses the design of configuration structures and associated processing that allows the core of the code, which is independent of experiment and geometry, to run in accordance with the given detector description, algorithm tuning, and required standard functions.

3.1 Geometry & detector description

In mkFit terminology a *layer* denotes an $r-z$ bounding box in global cylindrical space where hits belonging to the said layer are expected to be found. It usually corresponds in some way to detector construction or readout layers, but it does not have to:[4] its main purposes are to aggregate the hits, provide an easy way to specify layer crossing sequences for each tracking region (called a *layer plan*), and allow track search to proceed uniformly among a set of tracks. This reduces complexity and allows for the vectorization of certain key computations, including track candidate propagation, hit selection, and Kalman filter calculations.

Prior to CMSSW integration, logically dividing CMS into nested layers was sufficient to allow mkFit to roughly reproduce the physics performance of CMSSW legacy tracking. However, as mkFit was being considered a drop-in replacement for the existing tracking implementation, additional detector-module identification had to be included in mkFit's geometry description to enable it to pick up multiple hits from overlapping modules within the same layer. Moreover, to support the Phase-2 upgrade geometry, which includes axially tilted detector modules, further information had to be provided (module position, normal and φ-direction vectors).

The layer boundaries, module details, and material properties are all extracted during the CMSSW job setup by traversing all inner tracker modules. For standalone usage this information gets exported into a binary file.

3.2 Configuration structures

As mentioned in the introduction to this section, mkFit code needs to run concurrently within the main process, where each execution module is configured for its specific tracking iteration. As there can be no static or global data, the required configuration (or the relevant fragments) needs to be passed down the execution stack or stored in local objects. It is therefore important that the configuration data are structured in a way to facilitate such usage.

[4] E.g., mono and stereo hits from the same silicon-strip detector layer in CMS are split into separate mkFit layers.

The top-level configuration for each tracking iteration is represented by the class `IterationConfig`. It contains flags that control which steps of the track finding algorithm (see section 2.2) need to be performed, the standard functions that are to be used for this iteration (described in more detail in the next subsection), some high-level parameters for seed and duplicate cleaning, and the following structures.

- Layer traversal plans for all tracking regions.
- Tracking parameters (e.g., maximum number of missed layers, χ^2 cuts, quality filter parameters) encapsulated in class `IterationParams`, with two separate instances for forward and backward search.
- Iteration-specific layer information, stored in class `IterationLayerConfig`, which holds parameters guiding hit search and selection algorithms. These are stored in a vector, with one instance per layer.

The CMSSW module system is typically configured via Python scripts; this requires a rather tight coupling at the level of `C++` code to parse incoming data. As the above mkFit configuration is rather elaborate, it was accepted as a compromise that all mkFit configuration can be loaded from (and saved into) JSON files. Each iteration's configuration is stored in a separate file (stored as a part of CMSSW release) and the name of this file is then passed to the mkFit CMSSW module during instantiation.

To allow for an easy modification of a small number of parameters, reading of partial JSON overrides is fully supported: the default base configuration is read from the CMSSW release and then existing in-memory representation gets patched or overridden via simple additional JSON files or strings. Some frequently used parameters can also be set via the Python interface, e.g., to tune mkFit performance for heavy-ion operations.

Plugin-style configuration is still supported in standalone mode and is, in fact, used to generate the default JSON files for the CMSSW operation.

3.3 Standard function catalogs

While adding support for multiple iterations and for Phase-2 tracking it became obvious that using a single standard function and putting additional parameters into `IterationConfig` structure does not scale and that a more flexible configuration mechanism for standard functions is required for the following tasks:

- seed cleaning & partitioning – defined per iteration;
- candidate filters, pre- and post-backward fit – defined per iteration;
- duplicate cleaning – defined per iteration; and
- candidate scoring – defined per iteration with a possible override for each tracking region.

To provide a mechanism for registering different, specialized implementations of these standard functions, and to be able to choose them at configuration time, *standard function catalogs* have been introduced. For each type of function above, a thread-safe catalog with string keys and `std::function` value type is provided. The catalogs are populated via static object initializers in source files that contain the standard function codes. As `std::function` objects are exported, the functions themselves can be hidden in anonymous namespaces. Further, function templates can be used to inject compile-time parameters and, in simple cases, the registered functions can be direct lambda expressions.

With this infrastructure in place, JSON files can simply specify the names (strings) associated in the catalog with the desired function. After configuration loading and setup is complete the names get resolved into `std::function` objects for fast access and become available through the `IterationConfig` structure.

4 Ongoing & future work

At present, the described changes are being used to further tune Phase-1 CMS iterations. Algorithmic improvements in the processing of layers and multi-layer scoring are being investigated, with the goal of extending the usage of mkFit beyond the current five tracking iterations, as well as improving computational performance for currently supported use-cases. In parallel, Phase-2 tracking is being developed, currently still focusing on a single, initial tracking iteration, while the specifics of track propagation and Kalman updates required for the support of tilted modules are being worked out.

The final fit is now the most time-consuming tracking task in iterations using mkFit. With the latest additions to geometry description, it should be feasible to effectively use mkFit for this task as well, and we are investigating the required developments in this area, along with possible improvements to existing backward-fit and backward-search algorithms.

There has been a recent intense development of CMS Phase-2 Line Segment Tracking (LST) [8, 9], a highly parallelizable algorithm that can run efficiently on GPUs, which is showing great promise for both offline and high-level trigger usage. We are planning to explore possible synergetic development with the LST project, aiming for a hybrid approach where LST performs the initial, fast track finding and mkFit provides final steps such as backward fit, overlap hit search, and final fitting.

5 Conclusion

mkFit is in production mode for the CMS experiment since Run3 of the LHC, used as a drop-in replacement for the legacy tracking code for five out of twelve iterations, with equivalent physics performance and with overall tracking time reduction of ~25%. Work has started to support CMS Phase-2 tracking geometry and events with increased pileup, where some of required changes are described in this paper, namely: generalizations of geometry description, multi-iteration configuration, and introduction of catalogs of standard functions. Exploration of extending mkFit to also cover the final-fit and to operate synergistically with other track finding algorithms is in progress.

Acknowledgements

This work was supported by the U.S. National Science Foundation under Cooperative Agreements OAC-1836650 and PHY-2121686 and grant NSF-PHY-1912813.

References

[1] R. Frühwirth, *Application of Kalman filtering to track and vertex fitting,* Nucl. Instrum. Meth. **A262**, 440–450 (1987)

[2] CMS Collaboration, *The CMS experiment at the CERN LHC* JINST **3**, S08004 (2008)

[3] CMS Collaboration, *Development of the CMS detector for the CERN LHC Run 3,* arXiv:2309.05466 [physics.ins-det] (2023) https://arxiv.org/abs/2309.05466

[4] S. Lantz *et al.*, *Speeding up particle track reconstruction using a parallel Kalman filter algorithm,* JINST **15**, P09030 (2020)

[5] G. Cerati *et al.*, *Parallelized and Vectorized Tracking Using Kalman Filters with CMS Detector Geometry and Events,* EPJ Web of Conferences **214**, 02002 (2019)

[6] CMS Collaboration, *Performance of Run 3 track reconstruction with the mkFit algorithm,* CERN-CMS-DP-2022-018 (2022) https://cds.cern.ch/record/2814000

[7] CMS collaboration, *Description and performance of track and primary-vertex reconstruction with the CMS tracker,* JINST **9**, P10009 (2014) [arXiv:1405.6569]

[8] P. Chang *et al.*, *Line Segment Tracking in the High-luminosity LHC,* in these proceedings

[9] CMS Collaboration, *Performance of Line Segment Tracking algorithm at HL-LHC,* CERN-CMS-DP-2023-019 (2023) https://cds.cern.ch/record/2857438