Filter and Automaton for Silicon Tracking (FASTrack)

A1. Authors

Dmitry Emeliyanov

Rutherford Appleton Laboratory

d.emeliyanov@outlook.com

A2. Team in the competition

Competition name: TrackML Throughput Phase

Team name: fastrack

Leaderboard place: 2

A3. Summary

This document provides a general description of the FASTrack algorithm which won the second place in the throughput phase of the TrackML competition. The accuracy was 0.944, time per event 1.11 seconds and overall score 1.1145. The memory consumption of the algorithm itself is about 0.6 GB and when the algorithm is run in the trackml docker environment the overall memory consumption is 1.4 GB.

The algorithm is based on the following key ideas and techniques:

1) using hit clusters shape (numbers of cells in u- and v-directions) to predict the intervals of track inclination angles and save CPU time by avoiding hit combinations with parameters incompatible with the prediction;
2) using the track segment-based track following network with an embedded Kalman filter for fast discovery of track candidates;
3) limited usage of the Kalman filter-based combinatorial track following for missing hits search and track extension to areas not covered by the track following network such as long strip volumes 16, 17, and 18.

This document accompanies the code available on Github (https://github.com/demelian/fastrack). Please note that the code in the above repository contains several important bugfixes that were added after the end of the competition and before the public release of the code. These additions resulted in overall improvement of the algorithm performance so that the accuracy is 0.948 and the projected time per event is 0.8 seconds.

A4. High-level code description

The algorithm is implemented in C++ and integrated with model.py using ctypes. The algorithm shared library (libModel.so) is built using Makefile located in /lib sub-directory together with the C++ source code. The C++ compiler should support C++11 standard and OpenMP for parallelization.

The main tracking workflow is implemented in TrackFinder.cpp which provides a class used by ModelClass.cpp. The ModelClass imports hits and cells from model.py via ctypes-wrapped numpy arrays into the SoAs in Event.h. The track finding is organized as a multi-stage process. In the current solution there are 3 stages: the 1st finds higher momentum central tracks (coming from the interaction region along z-axis), the 2nd finds lower momentum central tracks, and the 3rd stage targets all the rest. For each stage, the ModelClass calls the TrackFinder instance and specifies which stage needs to be run, the output vector to place the reconstructed tracks, and the minimum number of not-shared hits per track. Once all stages are completed the output track collections are concatenated and hit labels are generated. A hit is assigned a track Id if the hit is not assigned to another track with a smaller track Id in order to create a unique "hit-to-track" assignment which is then passed back to model.py.

A5. Scientific details

The TrackFinder starts with arranging input hits into ring-like "bins" in all the detector layers. The bin widths are calculated in accordance with a uniform pseudorapidity (eta) binning to guarantee more or less the same number of hits per bin. The value of eta bin is 0.2. All hits in the bins are sorted along increasing value of phi (azimuthal angle).

After that the hits in each layer are clustered into nodes by the Cluster Maker class implemented in ClusterMaker.cpp. The idea is to collect together hits which likely belong to the same track but are located in different modules on the same layer. The nodes are used for the actual track finding while hits are used for more precise track fitting. After the clustering, the nodes are preselected for subsequent track segment creation on the basis on their cells parameters (number of cells along u- and v-directions). For each selected node an interval measurement (i.e. min/max) of the track inclination cotangent (tau) is calculated using a lookup table in HitLayer.cpp which relates the min/max values of tau to the number of cells in v-direction.

The TrackFinder connects nodes and creates track segments in accordance with the layer linking scheme trained on data. The scheme can be seen in connection.csv file. For example, the record

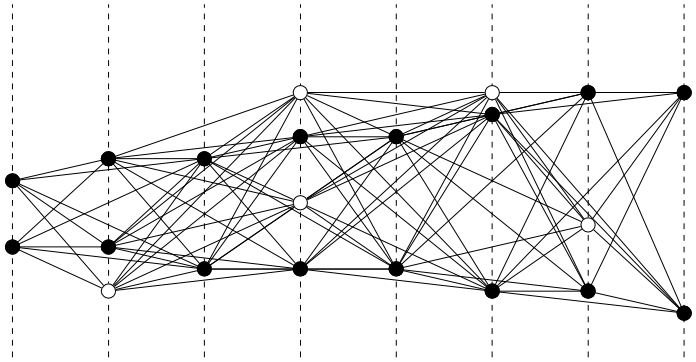
8004,8002,0.876002

means that the layer 2 of volume 8 is connected to layer 4 of volume 8 and the average "flow" of track score carried through this connection is 0.876002. The last parameter is used to characterize the importance of layer connections.

For the sake of speed, the layer linking scheme was converted to connection.bin input binary file. This file is loaded by the Layer Linker class (LayerLinker.cpp) during the configuration phase of the TrackFinder initialization together with the Geometry initialization. The geometry file (geometry.bin) is also converted from the original CSV format (detectors.csv).

To facilitate parallel processing by OpenMP the track segments are created and stored in 3 separate arrays (Segment Banks). The segment building algorithm operates on node collections from pairs of eta-bins (rings). The bin pairing was trained on data and stored in a look-up table (the binTable in the TrackFinder.cpp).

The next step of track finding connects track segments which share the same nodes and creates the track following network which is illustrated below.



The network is a graph in which the vertices are the nodes containing hits and the edges are connections between the nodes, i.e. track segment. For each vertex, there are two collection of edges: incoming and outgoing. The sense of direction is determined towards the z-axis of the detector. The algorithm selects all the vertices with non-empty “In” and “Out” collections and for each “In” edge finds possibly connected “Out” edges satisfying cuts on differences in azimuthal angle phi, pseudorapidity (eta), and track curvature values. The maximum allowed number of connections is set to 6.

Once the network is built the segments start interacting with their neighbours in the “Out” direction. The aim is to calculate the maximum number (called level) of connections which can be traced from the segment and identify the segments which are likely to be the starting points of long tracks. The TrackFinder implementation of this algorithm employs a cellular automaton (CA). The CA is parallelized using OpenMP and operates in synchronous mode: first all segments calculate the proposal for the new level (e.g. if a segment with level = 1 has a neighbour with the same level then the proposal for the next CA iteration is 1+1 = 2) and then all segments with proposals which differ from their current states gets updated. For more details on the CA-based track finding please see [1] (slides 21-26) and [2].

After the network evolution stops (no more segment level updates in the whole network) the TrackFinder proceeds with the extraction of track candidates from the track following network. The track extraction starts with the segments with the highest level and continues with extraction until the level drops below the stage-dependent threshold (4 for the 1st stage, 3 and 2 for the 2nd and 3rd, respectively). The track extraction is basically a segment-by-segment track following process which is implemented as the recursive “depth-first” graph traversal. In order to reduce quickly the number of traversed combinations a simplified Kalman track fit (TrackingFilter.cpp) is embedded in the recursion. The Tracking Filter algorithm estimates the track cotangent in the r-z projection and track phi and dphi/dr or dphi/dz in the r-phi projection. To speed things up, the Tracking Filter does not use any magnetic field description and instead, models the track evolution in r-z as a random walk (caused by the detector material effects) and as the Ornshtein-Uhlenbeck (AR(1)) process in the r-phi

projection which emulates gradual, trend-like, change in the track azimuthal direction under the influence of the magnetic field.

The extracted track candidates get fitted by the FastTrackFitter algorithm which employs the 3rd order Runge-Kutta track parameter and Jacobian extrapolator (see [3]) and a fast approximation for the non-ideal solenoidal magnetic field based on the approach from [4]. The parameters of the solenoid (field in the center, half-length, and the aspect ratio = radius/half-length) were learned from data by the tracking efficiency maximization during the hyper-parameter scan.

As many track candidates share the same hits some of the tracks are merged and removed after the next step of processing. This is done in the removeClones and assignCloneFlags methods of the TrackFinder. All tracks are sorted in accordance with their fit likelihood (the weighed number of layers with associated hits minus penalty on the chi^2 contributions of hits) and then hits are labelled by the track index starting from the best track. In this way, the shared hits are identified and, depending on the fraction of shared hits and the number of competing tracks, the decision is made whether to merge a track with a better one or to delete it. This decision is actioned upon in the removeClones method.

The merged tracks are re-fitted again and extended towards the interaction region and towards the long-strip volumes 16, 17, and 18, as they are not used in the segment creation and network building process, by the runTrackExtensions method of the TrackFinder. The track extension procedure consists of the prediction of track trajectory by extrapolation from the first (last) hit on the track, collection of hits around the trajectory crossing points on detector layers and track update. Any ambiguity in the “hit-to-track” assignment is resolved via the branching track propagation which also employs the Kalman track fit. The number of simultaneously propagated “best” branches is 1 for the “inside” track extension and 3 for the “outside” propagation. The track extension procedure can add up to 3 new hits per layer to account for situations when more than one hit per layer is produced in the overlapping detector modules.

Since the track extension can result in some additional hit sharing the removeClones is called again. After that the extended and possibly merged tracks are refitted and the “hit-to-track” assignments are reviewed in the reassignHits method of the TrackFinder. Any missing hits found in the vicinity of estimated track positions on the detector layers are added to a track. The hit addition algorithm applies the constraint that a track can have at most one hit per module.

Finally, all reconstructed tracks checked for the number of shared hits. If this number exceeds the stage-dependent threshold (e.g. 7 for the 1st stage) such a track is discarded. Otherwise, the track is accepted and all the hits on it are marked as assigned so that they cannot be used in the following stages of the track finding.

A6. Interesting findings

The most surprising finding is how efficient is the cluster shape-based prediction of track direction for reducing CPU time and improving the reconstruction efficiency. For instance, the accuracy score for this algorithm in the first phase of the TrackML challenge was 0.87 (4th place). As turned out this was mainly the result of not using the cluster shape-based prediction at all.

The most distinguishing feature of the algorithm is using the fast Kalman filter with approximate track model in the graph traversal part of the track candidate discovery. Similar algorithms in the

literature (cf. [2]) make use of track fit but only for the track accept/reject decision after the graph traversal is finished.

A7. Simple feature and methods

As the algorithm employs the multi-staged approach for track finding 90% of the final accuracy could be achieved by using only the 1st stage. This should also make the algorithm at least 20% faster.

A8. Model training and execution time

N/A as the algorithm does not use any ML training infrastructure.

A9. Outlook

The execution time of the algorithm can be improved by the following techniques:

1) Massive parallelization on GPGPUs. Most parts of the algorithm are already implemented in a thread-safe manner and accelerated using OpenMP directives. By exploiting the track-level parallelism the track fitting parts of the algorithms can be efficiently executed on a GPU as the fast and compact magnetic field model can be implemented as a GPU device code.
2) Currently, the "In" and "Out" collections of track segments are created independently. It might make sense to group layer pairs in such a way that, firstly, all "In" collections for a particular layer are formed. Then these collections can be analysed and the predictions for the "Out" segments made (for example, min/max of segment inclination angle compatible with the segments in the "In" collection). By using such predictions one could avoid creating segments which cannot be connected at their common nodes.

A10. References

1. https://www.ppd.stfc.ac.uk/Pages/ppd_seminars_170215_talks_dmitry_emeliyanov.pdf
2. https://indico.cern.ch/event/658267/contributions/2813731/attachments/1621890/2580935/20-02-2018_CDT_TrickTrack.pdf
3. https://iopscience.iop.org/article/10.1088/1748-0221/4/04/P04001
4. https://arxiv.org/abs/1003.3720