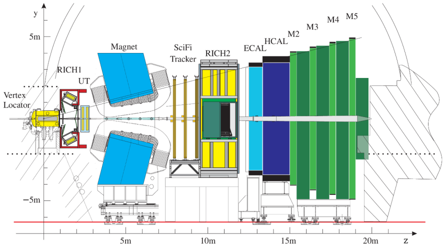
# Standalone track reconstruction in LHCb's SciFi detector for the GPU-based High Level Trigger

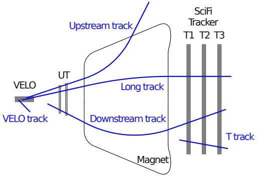
A. De Oyanguren Campos (Univ. of Valencia, CSIC), A. Hennequin (LNS, MIT), B. Kishor Jashal (Tata Inst. of Fundamental Research), C. Agapopoulou (CNRS), J. Zhuo (Univ. of Valencia, CSIC), L. Henry (CERN), L. Calefice (TUD, LPNHE)

Presenter: arthur.hennequin@cern.ch

# Track reconstruction in LHCb

- LHCb's tracking sub-detectors: Velo, UT, SciFi
- UT not available during the commissioning
- Use Velo and SciFi to reconstruct Long tracks
- This talk focuses on the SciFi tracker $\Rightarrow$ 3 stations, 2 vertical "X layers" and 2 tilted "U/V layers" each, divided into 2 parts ($y > 0/y < 0$)



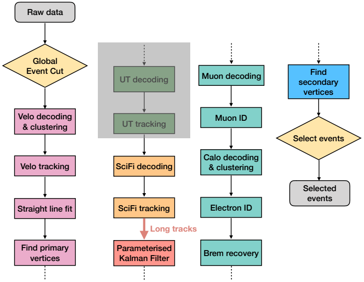



# High Level Trigger on GPUs

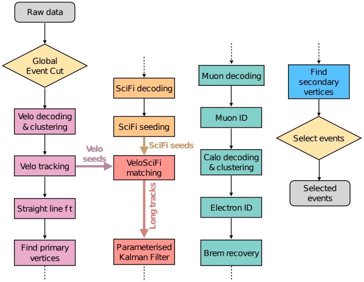
For Run 3, LHCb uses a 2 stages software HLT:

- HLT1 takes event at 30MHz and runs on O(200) GPUs
- HLT2 takes HLT1-filtered events at 1MHz and runs on CPUs

Default HLT1 sequence vs our proposed HLT1 sequence:




forward with UT (default)




seeding and matching (ours)

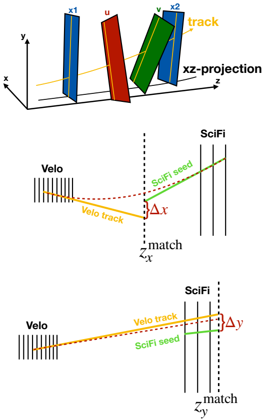
# SciFi seeding and matching

SciFi seeding:

- seed_xz: find xz-projections using only x layers
- seed_confirmTracks: augment the projections with a y component using information from tilted u/v layers

Matching:

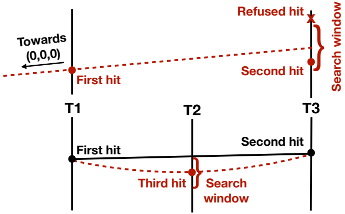
- extrapolate velo and scifi segments to a parametrized position
- measure the error and keep the best matches



# Seeding algorithm: seed_xz

For each part ($y > 0/y < 0$) independently:

- Store hits of the 6 X layers into shared memory
- Make triplets of aligned hits pointing roughly to coordinate (0,0,0)
  $\Rightarrow$ search windows in each layer are momentum dependent

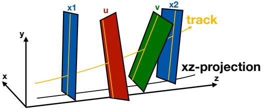


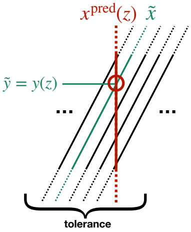

- Make 4-6 hits candidates with remaining x layers in registers
- Fit track parameters in xz (3rd order polynomial) and compute $\chi^2$
- Remove duplicates

# Seeding algorithm: seed_confirmTracks

For each part ($y > 0 / y < 0$) independently:

- Store hits of the 6 U/V layers into shared memory
- Collect hits in UV layers for each xz track in parallel, starting in 2 different layers, then using the track and first hit to collect the rest
- Fit track parameters in yz (linear) and compute $\chi^2$, on the fly



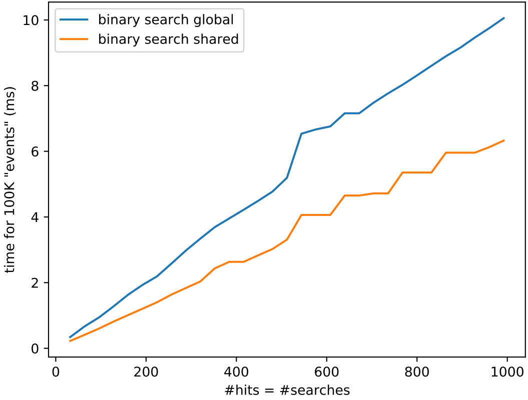
# Storing hits in Shared Memory

The seeding includes a lot of combinatorics over the hits: they are read multiple time each and most of the time in random order (non-coalesced).

But shared memory is a limited storage, how many hits do we need to store ?

- We only care about the x of the hit (4 bytes)
- We only need 6 layers and one part at a time (X layers or UV layers)
- We authorized $6 \times 300 = 1800$ hits in total: about 7.2KB
- Fallback to global memory if overflow: occurs very rarely.

# Searching hits: global or shared memory ?




Constant speedup: $\times 1.57$ when using shared memory

# Registers

Intermediate track candidates stay on the chip $\Rightarrow$ stored in registers (the fastest kind of memory on the GPU)

There are a few rules to allow the compiler to place a variable into registers (otherwise they are placed in global memory):

- Loop sizes must be known at compile time (to allow full unrolling)
- Array indices must be known at compile time
- The maximum number of registers depends on how many threads per block and how many block per streaming multiprocessor (max 64k 32-bit registers per SM)
- Delay conditional index increments to the very end of the kernel

# Registers (Loop unrolling)

```
float x, tx; // registers
float dz[6]; // registers ?
float x_pred[6]; // registers ?
for (int i=3 ; i<6 ; i++) {
    x_pred[i-3] = x + tx * dz[i];
}

// Compiler will unroll everything:
register x_pred_0 = x + tx * dz_3;
register x_pred_1 = x + tx * dz_4;
register x_pred_2 = x + tx * dz_5;
```

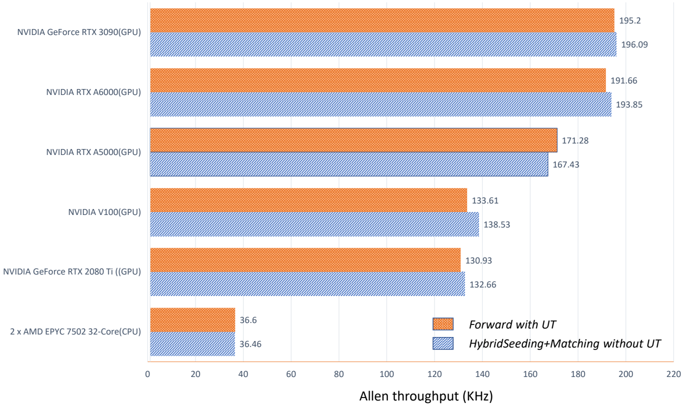
# Registers (early conditional increment)

```
int hits[6]; // local variable
int nHits = 0;
for (int i=0 ; i<6 ; i++) {
  // idx from somewhere, conditional increments:
  if (idx != -1) hits[nHits++] = idx;
}
// ...
for (int i=0 ; i<nHits ; i++) {
  tracks[threadIdx.x].idx[i] = hits[i];
}
// Gets translated to:
ld.global.u32 %r2, [%rd2];
setp.eq.s32 %p1, %r2, -1;
@%p1 bra $L__BB0_2;
st.local.u32 [%rd1], %r2;
// ...
ld.local.u32 %r42, [%rd38]; // load from local (global) memory
st.global.u32 [%rd39], %r42; // store in global memory
```

# Registers (delayed conditional increment)

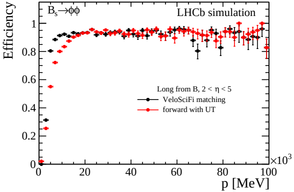
```
int hits[6]; // local variable
int nHits = 0;
for (int i=0 ; i<6 ; i++) {
  hits[i] = idx; // always store (idx may be invalid)
}
// ...
for (int i=0 ; i<6 ; i++) {
  if (hits[i] != -1) tracks[threadIdx.x].idx[nHits++] = hits[i];
}
// Gets translated to:
ld.global.u32 %r1, [%rd6];
ld.global.u32 %r2, [%rd6+4];
ld.global.u32 %r3, [%rd6+8];
ld.global.u32 %r4, [%rd6+12];
ld.global.u32 %r5, [%rd6+16];
ld.global.u32 %r6, [%rd6+20];
// ...
setp.eq.s32 %p1, %r1, -1;
@%p1 bra $L__BB1_2;
st.global.u32 [%rd9], %r1; // store register in global memory
```

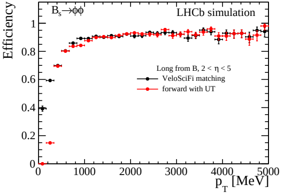
# Comparison to forward tracking with UT

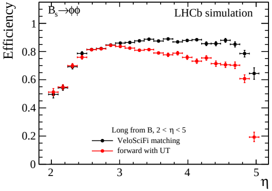



The currently installed ∼200 RTX A5000 can handle the full detector input at 30MHz

# Comparison to forward tracking with UT

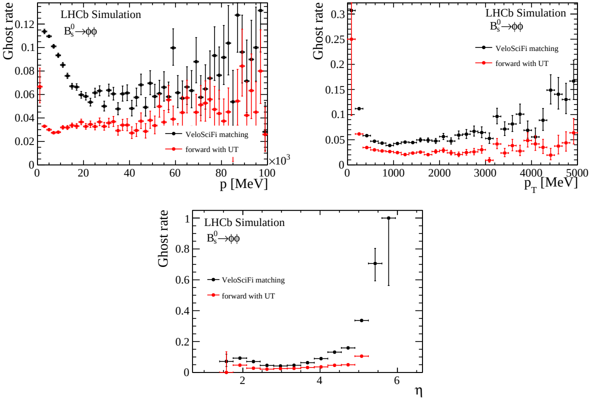



Matching is more efficient at low p / pt and at high eta, since no explicit momentum cuts are needed to meet throughput requirements.

# Comparison to forward tracking with UT




Matching has more ghosts due to the absence of UT informations. Ghosts could be killed when UT becomes available.

# Conclusion

We developed a standalone algorithm to reconstruct tracks in the SciFi detector:

- fast enough to run in real-time on GPUs
- provides an alternative way of reconstructing long tracks in abscence of UT
- as efficient as the standard forward tracking with UT
- opens possibilities to reconstruct downstream tracks when UT become available

Our algorithm is currently being used in the commissioning of the detector and the software trigger.

A standalone UT seeding and downstream matching is in development.