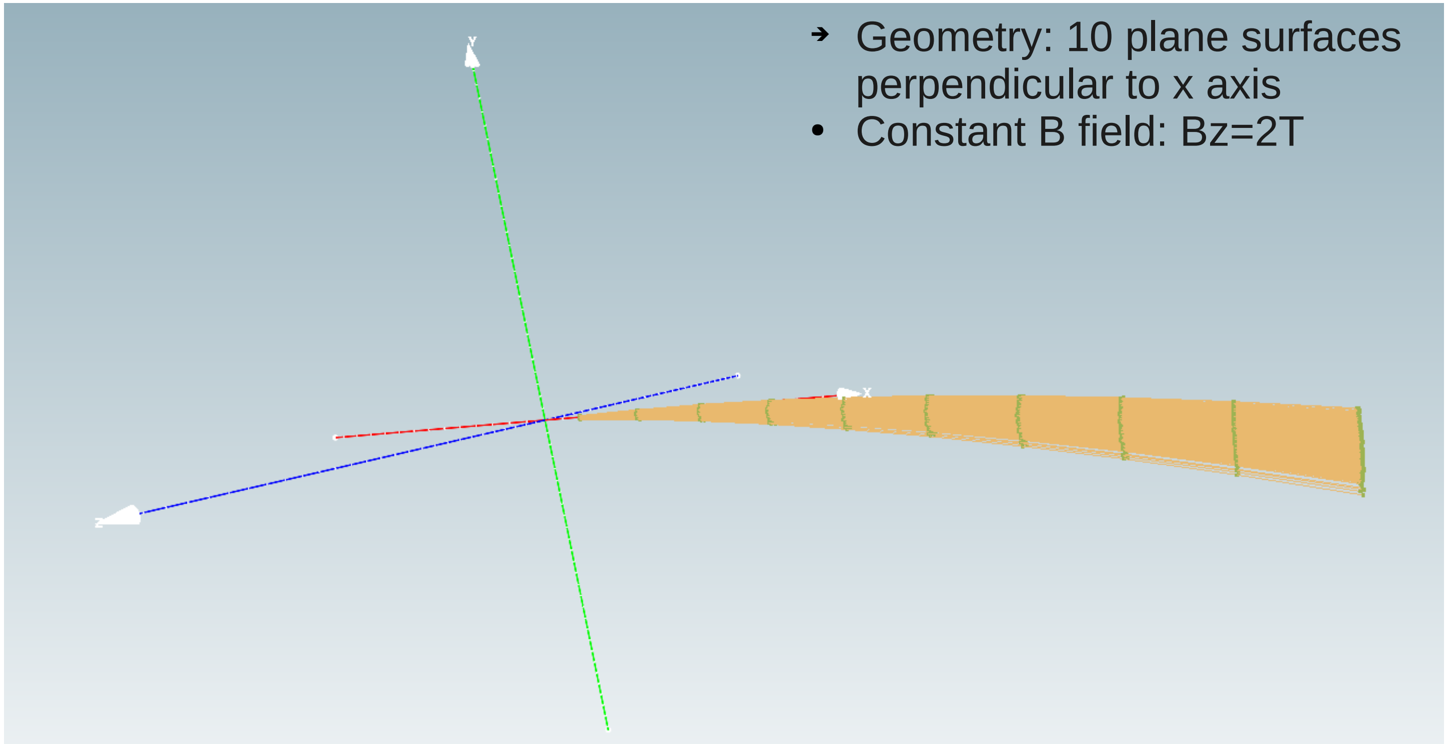
ACTS KalmanFitter on GPU

Xiaocong Ai

Sept 25, 2020

Optimizations of ACTS KalmanFitter on GPUs

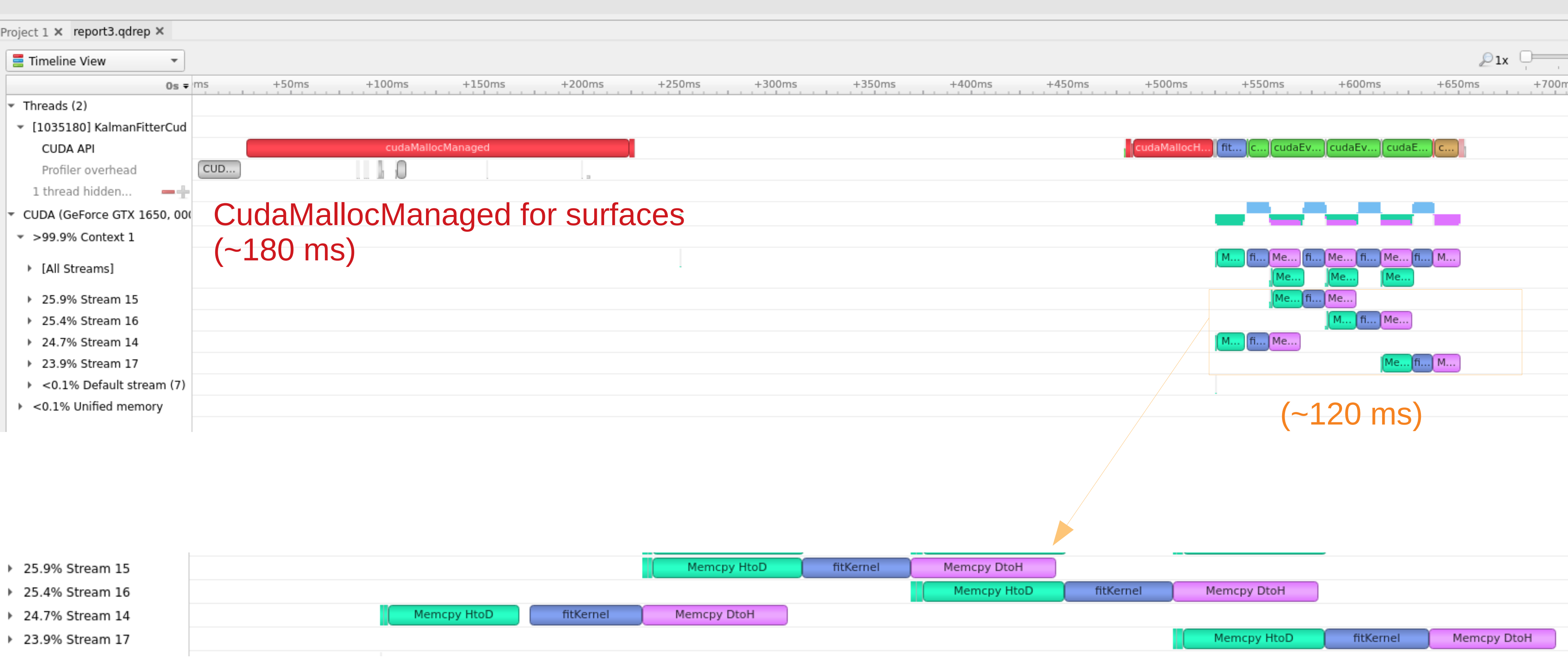
- Using paged-locked (pinned) memory for source links, starting parameters
  - Managed memory for surfaces and fitted states
- Using streams for asynchronous data transfer and kernel execution
- Currently testing the approach of using one block for one track
  - Use shared memory for propagator options, propagation state, propagation result, but this requires relevant objects to be default-constructible



➔ Geometry: 10 plane surfaces perpendicular to x axis
- Constant B field: Bz=2T

Timing profiling

The managed memory allocation and data transfer between CPU and GPU seems to be the bottleneck

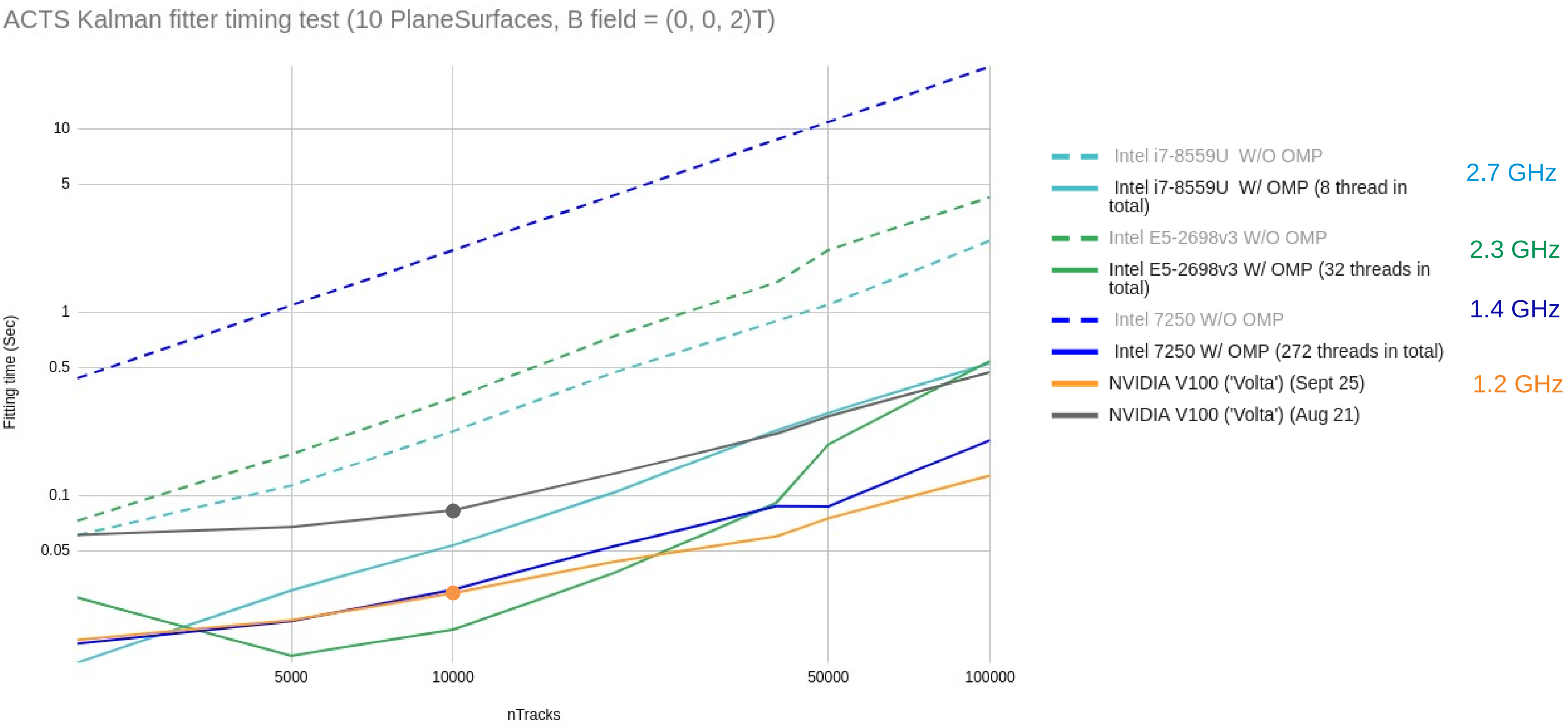


CudaMallocManaged for surfaces (~180 ms)

(~120 ms)

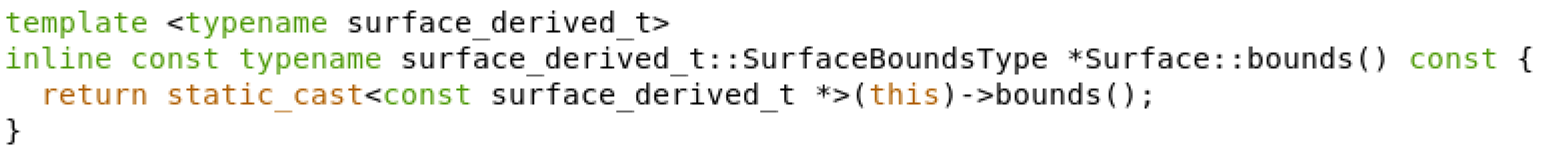
ACTS KalmanFitter performance on GPUs

- Comparable execution time between GPU (excluding surfaces allocation and transfer time) and CPU+openMP parallelization



Navigator on GPU

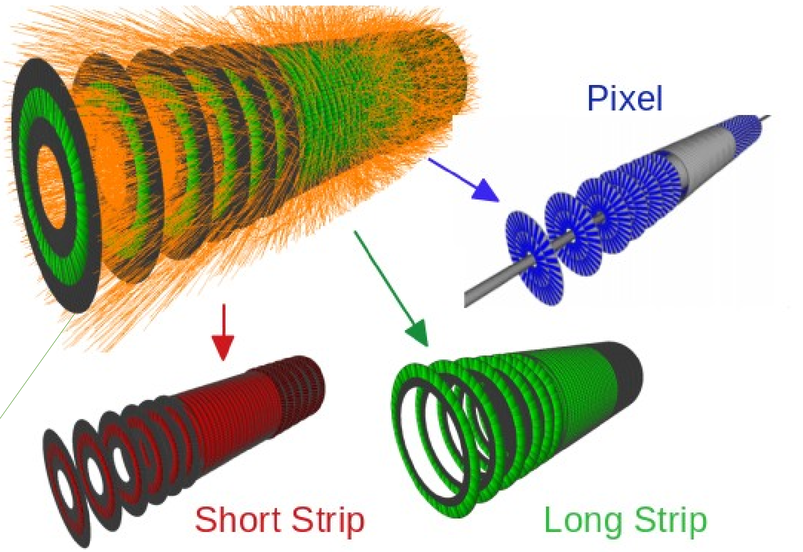
- Reminder:
  - Virtual functions couldn’t be called inside CUDA kernels (unless objects are constructed inside kernels)
  - All ACTS surface types have a polygonal representation allowing for triangular mesh of all surfaces
- Polymorphism is necessary if we doesn’t want two sets of code for CPU and GPU
  - CRTP (Curiously Recurring Template Pattern) is used for the surface class, i.e. either the whole base class or some of its `virtual` functions are templated on the derived class
    - But not real polymorphism, i.e. the objects couldn’t be put in one container any more and/or the derived class type must be statically known when those `virtual` functions are called

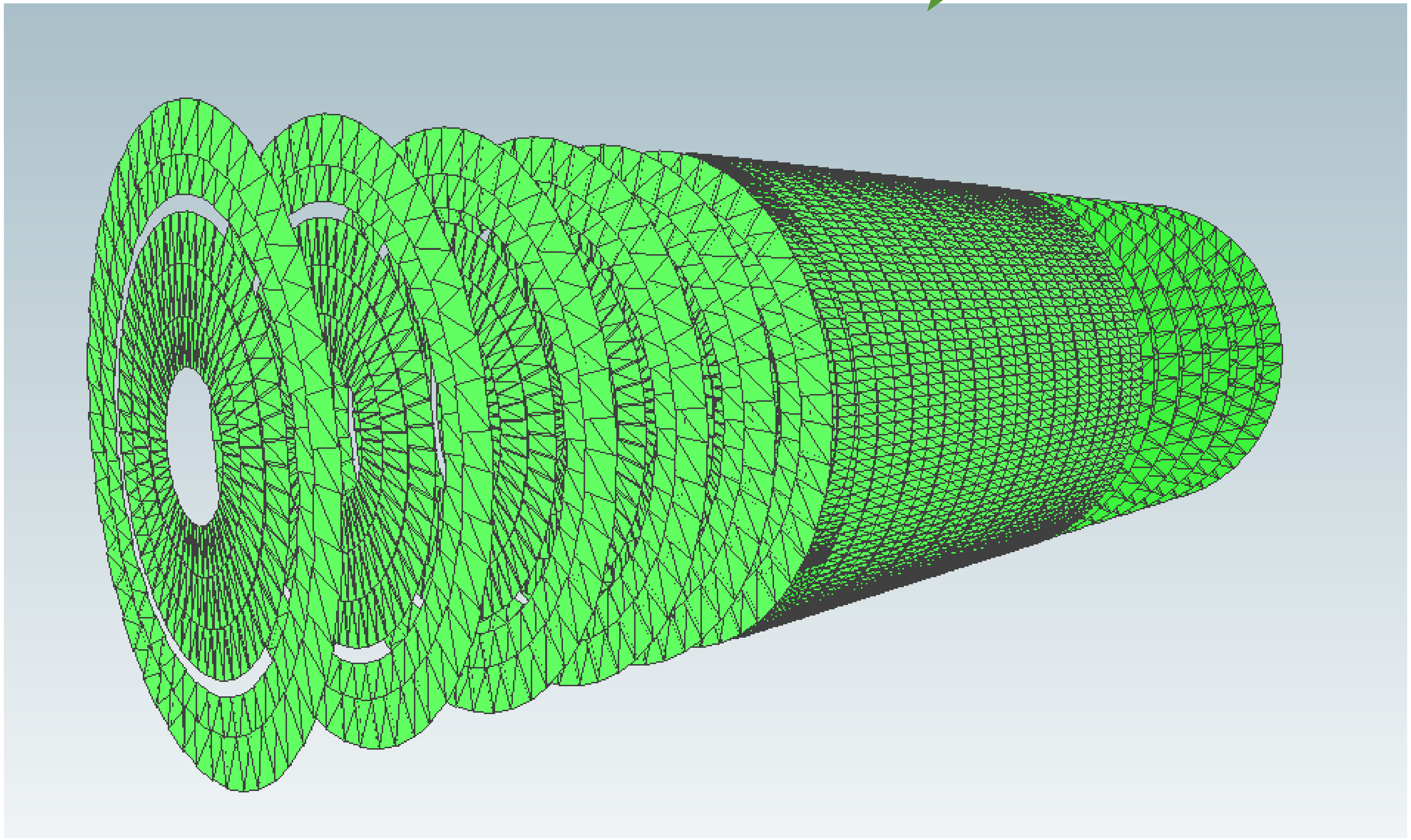


```
template <typename surface_derived_t>
inline const typename surface_derived_t::SurfaceBoundsType *Surface::bounds() const {
  return static_cast<const surface_derived_t *>(this)->bounds();
}
```

Triangular-meshed detector

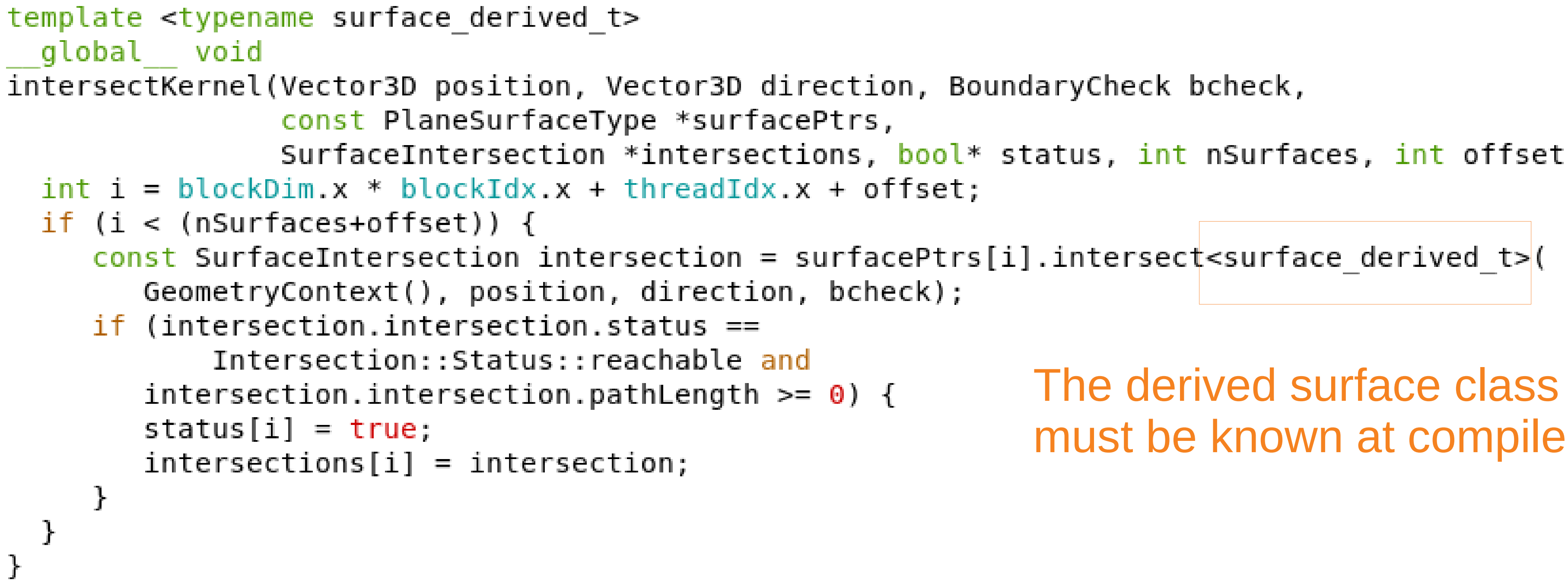
- Senstive surfaces of TrackML detector are meshed to 37456 triangles (PlaneSurface with ConvexPolygonBounds):
    - Surface bounds: ~1.8 MB
    - Surfaces: ~6 MB
    - Intersections: ~ 3.6 MB





Surfaces intersecting on GPU

- Intersections of a seed with detectors are computed using CUDA Kernels

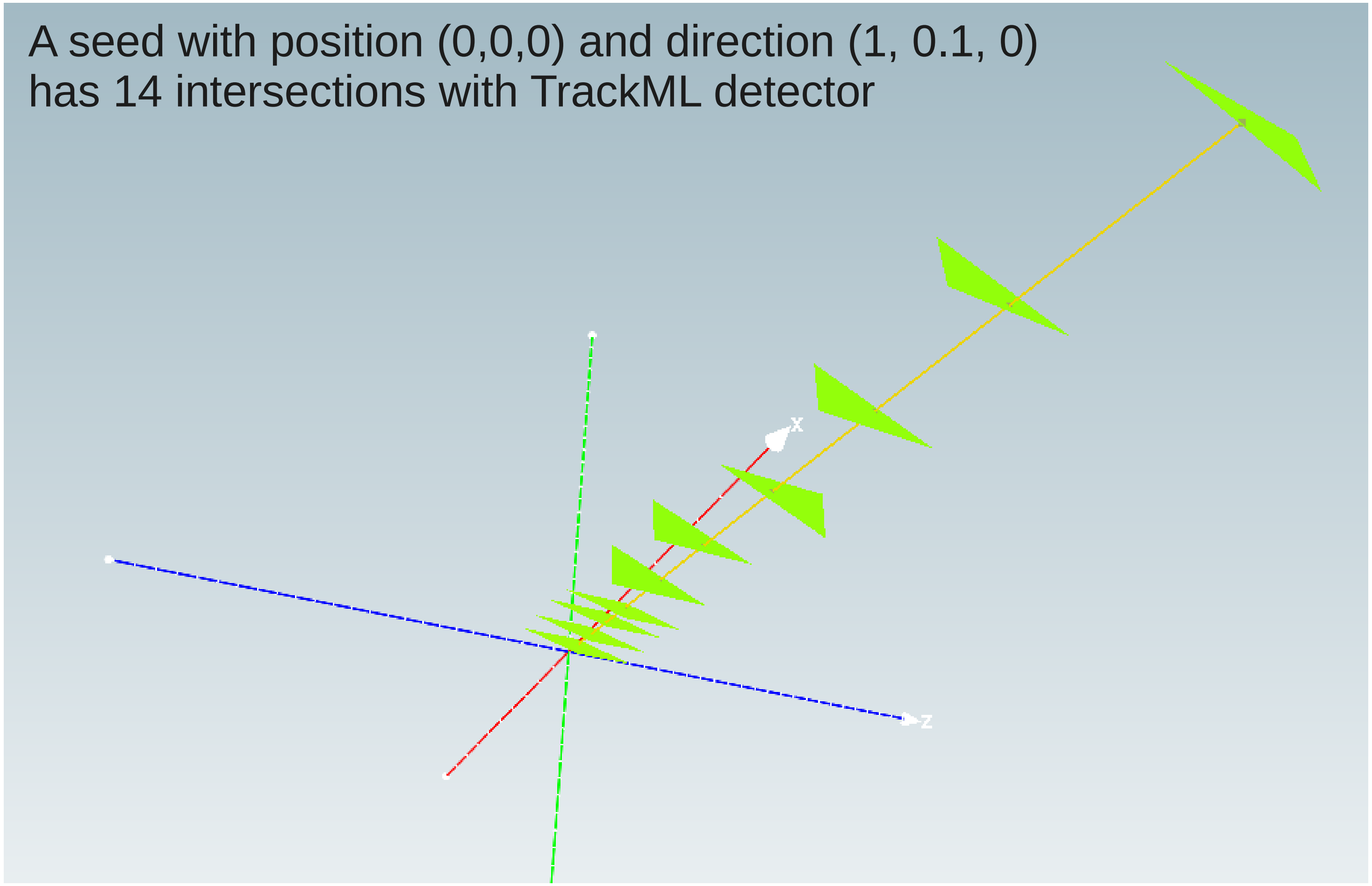


```
template <typename surface_derived_t>
__global__ void
intersectKernel(Vector3D position, Vector3D direction, BoundaryCheck bcheck,
                const PlaneSurfaceType *surfacePtrs,
                SurfaceIntersection *intersections, bool* status, int nSurfaces, int offset) {
  int i = blockDim.x * blockIdx.x + threadIdx.x + offset;
  if (i < (nSurfaces+offset)) {
     const SurfaceIntersection intersection = surfacePtrs[i].intersect<surface_derived_t>(
        GeometryContext(), position, direction, bcheck);
     if (intersection.intersection.status ==
            Intersection::Status::reachable and
        intersection.intersection.pathLength >= 0) {
        status[i] = true;
        intersections[i] = intersection;
     }
  }
}
```

The derived surface class type must be known at compile-time

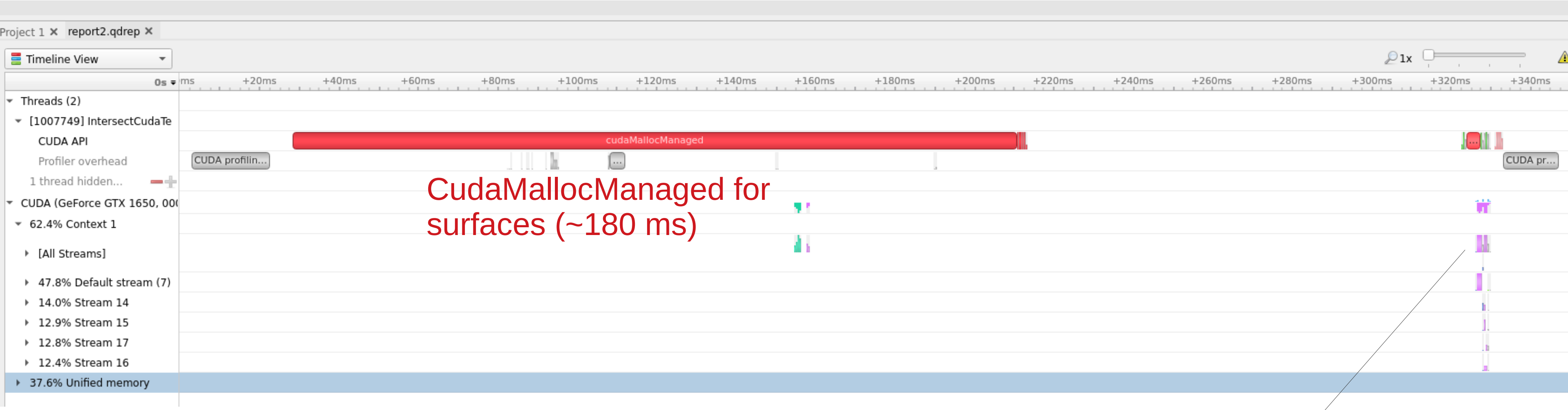
Navigation timing on GPU

- One intersection call @NVIDIA Volta 100: 0.2 ms (excluding the surface allocation and transfer time)
- Latency might be hidden by running thousands of seeds simultaneously
  - For one track propagation, do we need to run the intersection call multiples times?



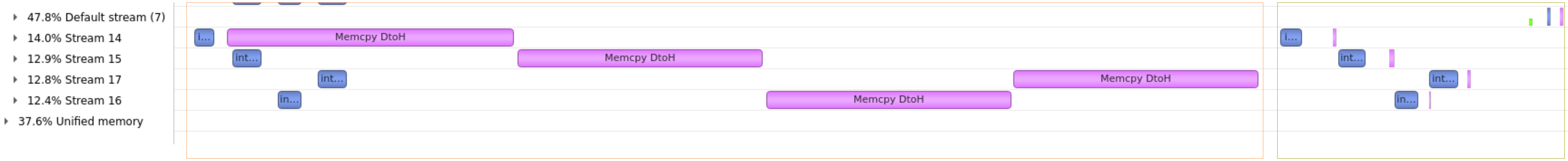
A seed with position (0,0,0) and direction (1, 0.1, 0) has 14 intersections with TrackML detector

How to minimize D2H data transfer



CudaMallocManaged for surfaces (~180 ms)

Test 1: D2H copy all intersections (valid or not)



(~0.2 ms)

Test 2: D2H copy only valid intersections
1) First D2H copy valid intersection indices
2) Alloc memory for valid intersections on device
3) Copy valid intersection from full intersections array to the valid intersections array
4) D2H valid intersections

Summary

- Cuda streams could allow overlapping data transfer and kernel execution
- Objects to be stored as pointers could use unified memory to avoid deep copy (but expensive)
- Polymorphism on GPU is difficult
- Non-default-constructible class can’t be readily used on GPU
- Kernel execution might be further optimized, but the real bottleneck seems to be the (unified) memory allocation and data transfer
  - Needs enough parallelism to hide the memory allocation and data transfer latency