VecGeom `surface_model` profiling

Peter Heywood, Research Software Engineer

The University of Sheffield

2024-11-12

VecGeom `surface_model` & `testRaytracing`

University of Sheffield

VecGeom `surface_model`

> VecGeom is a geometry modeller library with hit-detection features as needed by particle detector simulation at the LHC and beyond

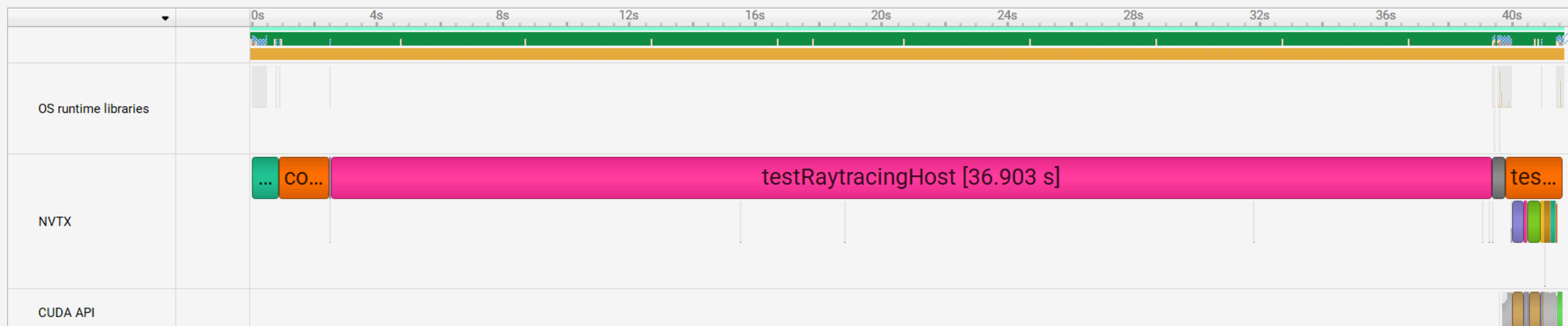
- gitlab.cern.ch/VecGeom/VecGeom
- CPU & GPU implementations
- Solid modelling / representation
  - Not ideal for GPU
- Developers are adding a Surface modelling / representation
  - `surface_model` branch
    - Similarities with Orange/Celeritas

`testRaytracing`

- Loads geometry, generates random rays, tests on CPU and GPU.
  - Solid and Surface representation
  - Validation
  - With & Without BVH
  - GPU Surface BVH using multiple kernel launches & split kernels
- `test/surfaces/testRaytracing.{h/cpp/cu}`
- Profiling tweaks:
  - NVTX ranges for profile annotation
  - `-oncpu 0` to disable cpu runs to to speed up profiling

University of Sheffield

`testRaytracing` timeline CPU & GPU

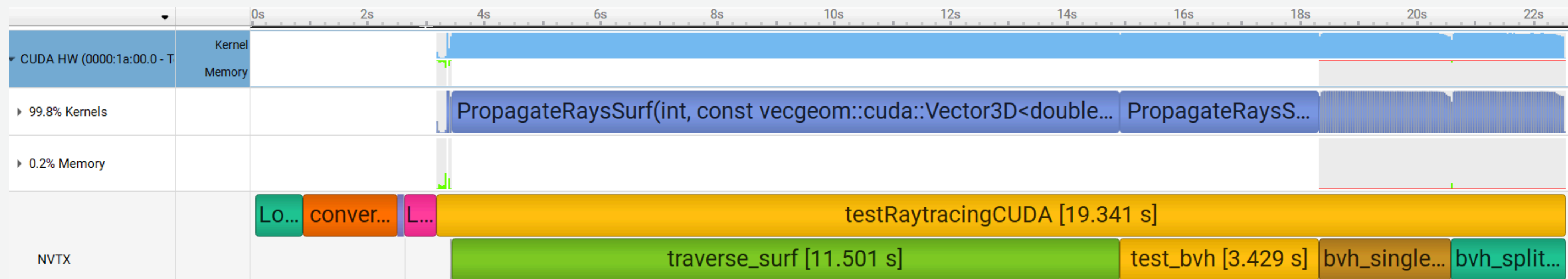


CPU & GPU Timeline (TBHGCal, `16384` rays, `-use_TB_gun 1`, V100)

```
1 testRaytracing -gdml_name TBHGCal.gdml -ongpu 1 -mmunit 0.1 -verbosity 0 \
2     -accept_zeros 1 -validate_results 1 -nrays 16384 -use_TB_gun 1       \
3     -only_surf 0 -test_bvh 1 -bvh_single_step 1 -bvh_split_step 1 -oncpu 1
```


University of Sheffield

testRaytracing timeline -oncpu 0 -only_surf 1



GPU & surface only timeline (TBHGCal, 524228 rays, `-use_tb_gun 1`, V100)

```
1 testRaytracing -gdml_name TBHGCal.gdml -ongpu 1 -mmunit 0.1 -verbosity 0   \
2     -accept_zeros 1 -validate_results 0 -nrays 524228  -use_TB_gun 1       \
3     -only_surf 1 -test_bvh 1 -bvh_single_step 1 -bvh_split_step 1 -oncpu 0
```

Hardware & Geometries

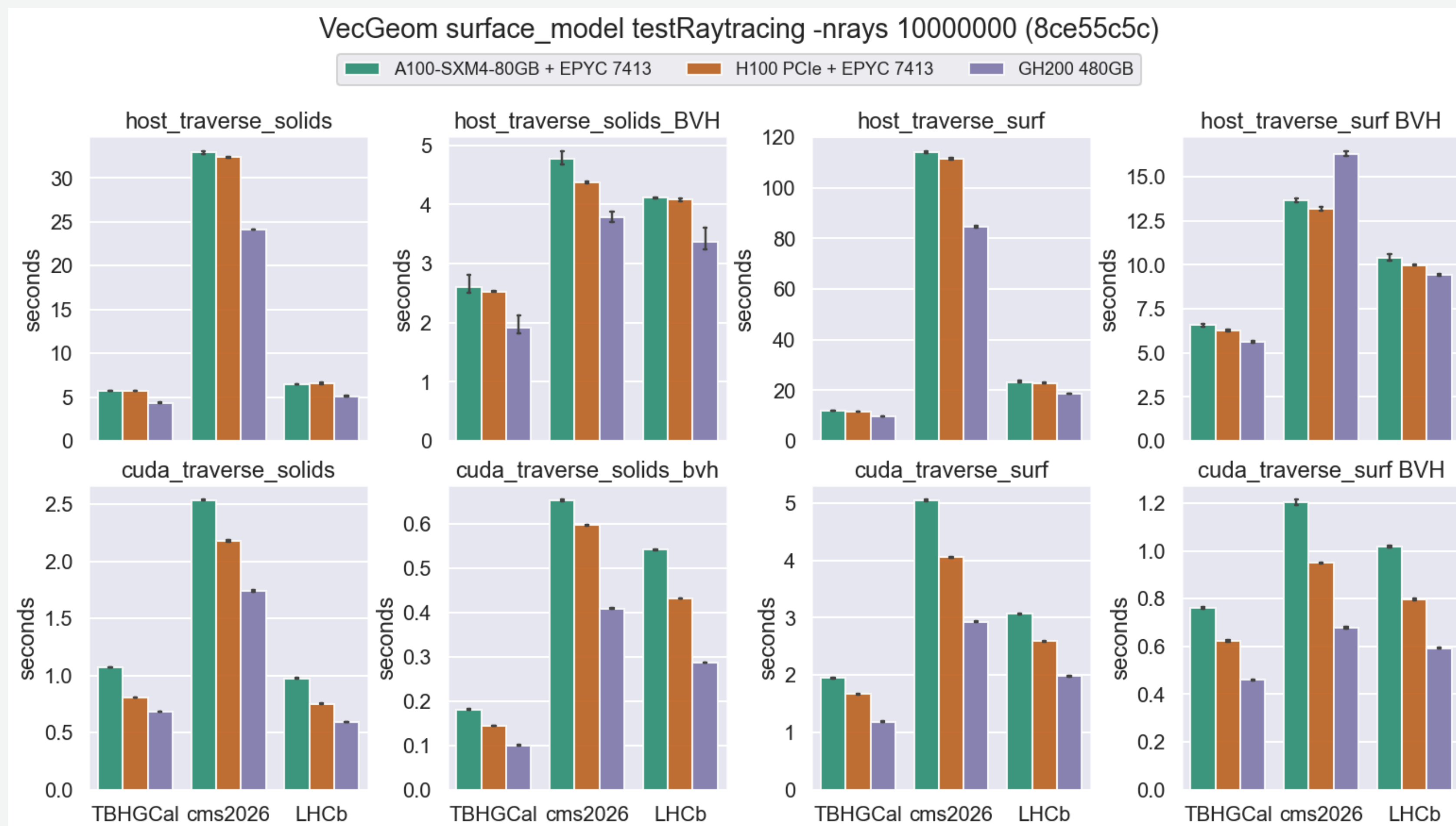
GPU	CC	CPU	Cluster	Driver
V100 SXM2	70	Intel Xeon Gold 6138	TUoS Bessemer	550.127.05
A100 SXM4	80	AMD EPYC 7413	TUoS Stanage	550.127.05
H100 PCIe	90	AMD EPYC 7413	TUoS Stanage	550.127.05
GH200	90	Nvidia Grace	N8CIR Bede	560.35.03

Geometry	Touchables
trackML.gdml	18790
TBHGCal181Oct_fixdup.gdml	61802
cms2026D110DD4hep_fix.gdml	13133900
LHCb_Upgrade_onlyECALandHCAL.gdml	18429884

University of Sheffield

Initial Benchmarking

- 3 geometries
- 10 million rays
  - not using TB gun
  - TBHGCal unrealistic
- 3 machines
  - A100
  - H100 pcie
  - GH200

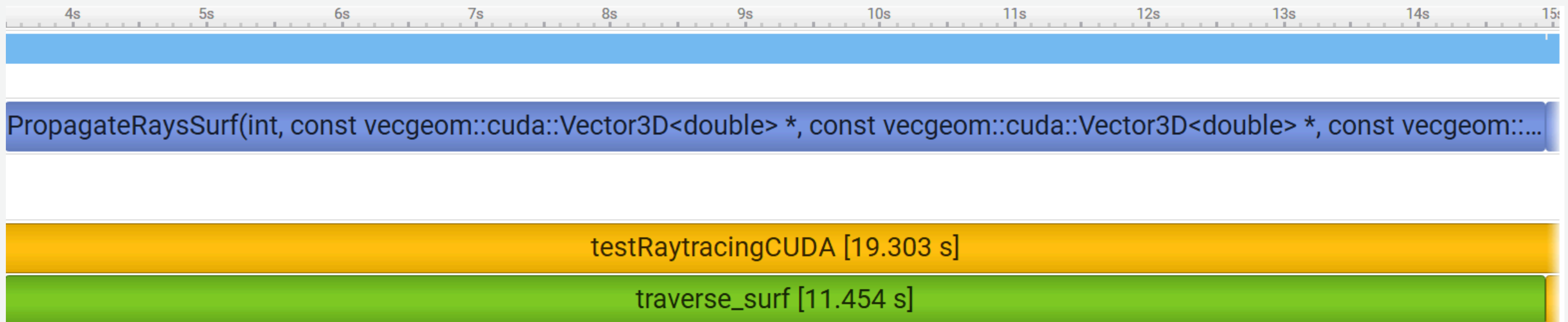


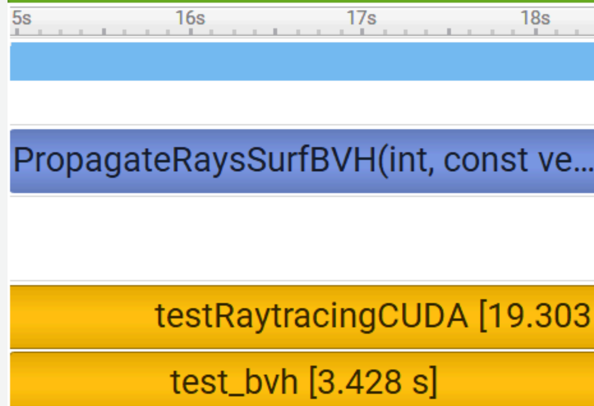
Initial `testRaytracing` benchmarking with 10 Million Rays

Initial surface model profiling

`PropagateRaysSurf` & `PropagateRaysSurfBVH`

- Single kernel launch for the full batch of rays.
- Grid-stride over rays, steps until the ray is outside the geometry
- *Bounding Volume Hierarchy (BVH)* improves work-efficiency
- TBHGCal, `-nrays 524228 -use_TB_gun 1` on V100



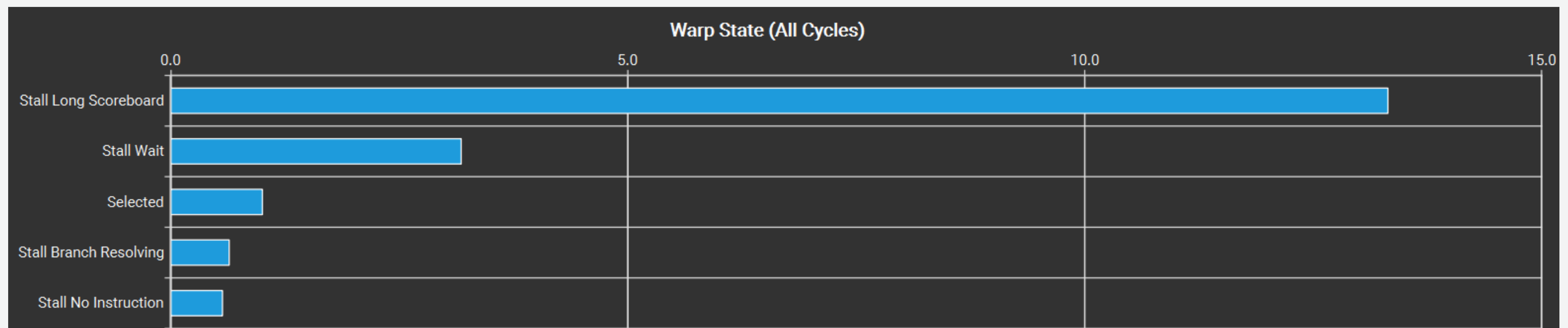


Method	Duration (s)
`PropagateRaysSurf`	11.454
`PropagateRaysSurfBVH`	3.428

`PropagateRaysSurf` & `PropagateRaysSurfBVH` kernel profiling

- `ncu --set full -o report.ncu-rep ./testRaytracing ...`
- For all 4 geometries on V100 and GH200 highlighted issues are:
  - Very low occupancy
  - Long scoreboard (memory) stalls
  - Scattered memory access

Theoretical Occupancy [%]	12.50
Theoretical Active Warps per SM [warp]	8
Achieved Occupancy [%]	12.03
Achieved Active Warps Per SM [cycle]	7.70



Nvidia GPU Structure

- NVIDIA GPUs are made up of many *Streaming Multiprocessors* (SMs)
  - GH100 die contains 144 SMs (8 GPCs of 18 SMs)

Full GH100.

© NVIDIA Corporation (source)

Nvidia Streaming Multiprocessor (SM)

- Each SM contains:
  - Compute units (Int32, FP32, FP64)
  - **Register file (64K 32-bit registers for Hopper)**
  - Instruction Cache
  - L1 Caches & Shared memory
- Latency to Resources outside the SM is higher
  - L2 Cache
  - Global memory (and local memory)

GH100 SM.
© NVIDIA Corporation (source)

University of Sheffield

CUDA Terminology

- *Kernels* are executed by **grid** of (*clusters of*) **blocks** of **threads**
- Blocks are issued to an SM, and they become **resident**
  - Remain on the SM until all threads in the block return
- A **warp** is the group of `32` threads which execute in lock-step
  - **Active warps** are resident in an SM and have not executed their last instruction
  - **Stalled warps** are not ready to execute their next instruction
  - **Maximum number of resident warps & threads per SM**
    - `64` warps, `2048` threads for V100

Occupancy

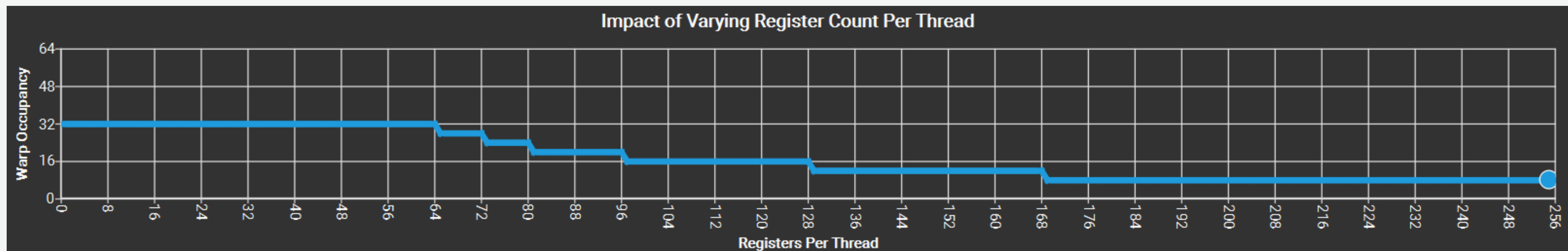
- **Occupancy** - ratio of *active warps* on an SM to the *maximum warps per SM*
  - **Theoretical Occupancy** - occupancy based on hardware and kernel constraints
  - **Achieved Occupancy** - observed occupancy during execution

- Low occupancy
  - Reduces latency hiding (i.e. *long scoreboard stalls*)
  - Cannot exploit all of the GPU if too low
- Higher occupancy does not guarantee higher performance

- Theoretical Occupancy can be limited by
  - **Registers per thread**
  - Threads per Block
  - Shared Memory per thread

Occupancy: registers per thread

- **Registers per thread** for kernel selected by optimiser at compile time
  - Maximum of `255` 32-bit registers per thread in recent HPC GPUs
  - 64K 32-bit registers per SM
  - `PropagateRaysSurf` uses 255 reg/thread: `12.5%` occupancy

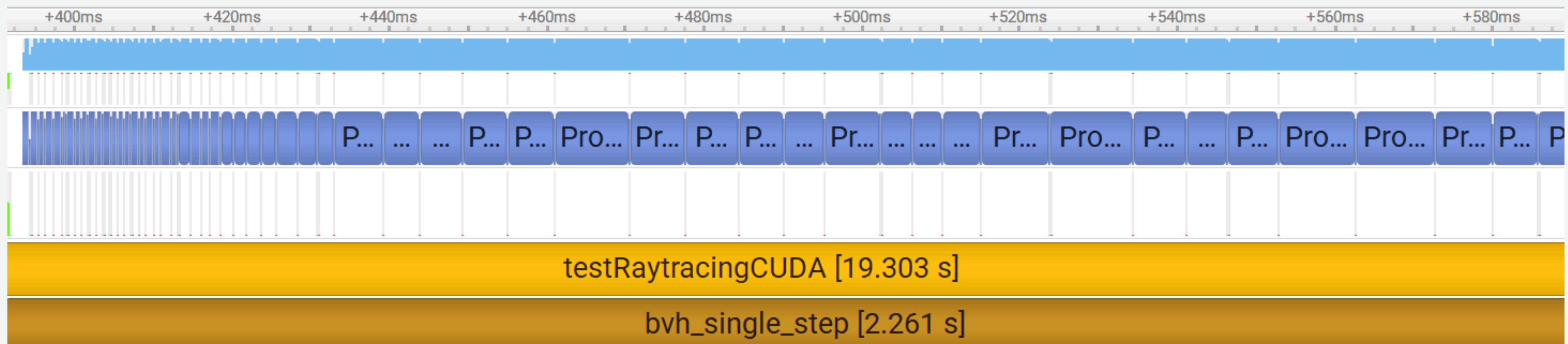


- Attempt to improve by:
  1. Split monotlithic kernel
  2. Force the compiler to use less registers per thread
  3. Lower precision reals

Optimisation attempts

Split monolithinc kernel: `-bvh_single_step`

- Split the single kernel launch into a loop of 2 kernels:
  - `PropagateRaysSurfBVHSingle` - traverse a single step
  - `filterAliveRays` - compact the alive/inside rays for the next iteration



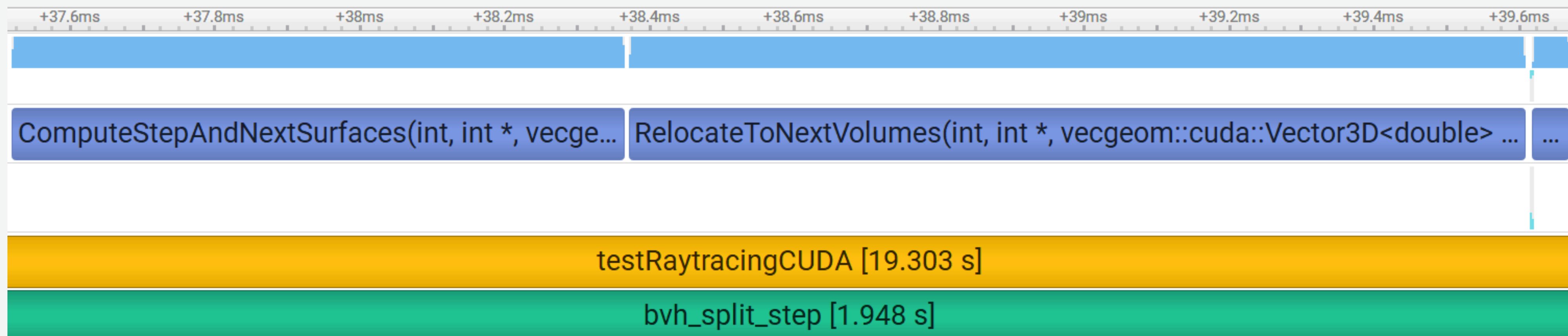
- 🎉 `250` registers per thread
- 😞 Still `12.5%` occupancy
- 😄 Shorter runtime
- ❌ Some runtime errors on some hardware to debug

Method	Duration (s)
`PropagateRaysSurf`	11.454
`PropagateRaysSurfBVH`	3.428
`bvh_single_step`	**2.261**

University of Sheffield

Split further: `-bvh_split_step`

- Splits `PropagateRaysSurfBVHSingle` into
  - `ComputeStepAndNextSurfaces`
  - `RelocateToNextVolumes`



🎉 `153` reg / thread for `ComputeStepAndNextSurfaces`

😄 `18.75%` occupancy

🎉 `218` reg / thread for `RelocateToNextVolumes`

😞 Still `12.5%` occupancy

😄 Shorter runtime

❌ Some runtime errors on some hardware to debug

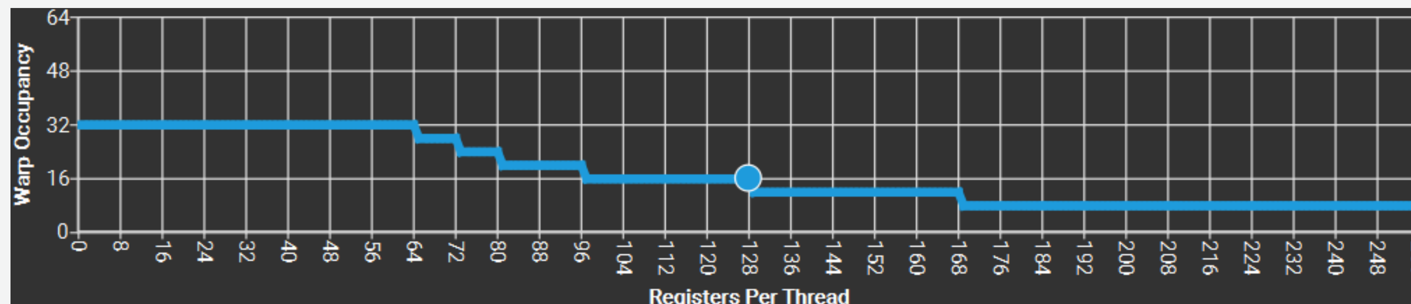
Method	Duration (s)
`PropagateRaysSurf`	`11.454`
`PropagateRaysSurfBVH`	`3.428`
`bvh_single_step`	`2.261`
`bvh_split_step`	**`1.948`**

Increase Occupancy: Set maximum registers per thread

- Force NVCC to limit registers per thread
  - For all kernels via `--maxrregcount=N`
  - Per kernel via qualifiers
    - `__maxnreg__` for CUDA >= 12.4
    - `__launch_bounds__` (less intuitive)
- Often hurts more than it helps
  - Increased Occupancy
  - Forces register spills to high-latency local memory

- TBHGCal, `-nrays 524228`, `-use_TB_gun 1` , V100
- `-maxrregcount=128`
- `25%` theoretical occupancy
- ~75% increase in global memory transfer for `PropagateRaysSurfBVH`

Strategy	Reference	128reg/thread
`PropagateRaysSurf`	11.456	**9.224**
`PropagateRaysSurfBVH`	**3.430**	3.515
`bvh_single_step`	**2.263**	2.323
`bvh_split_step`	**1.948**	2.145



Increase Occupancy: Set maximum registers per thread

Surface approach runtimes for `-nrays 524228` on V100 with maximum register counts of 255 and 128

Mixed precision mode

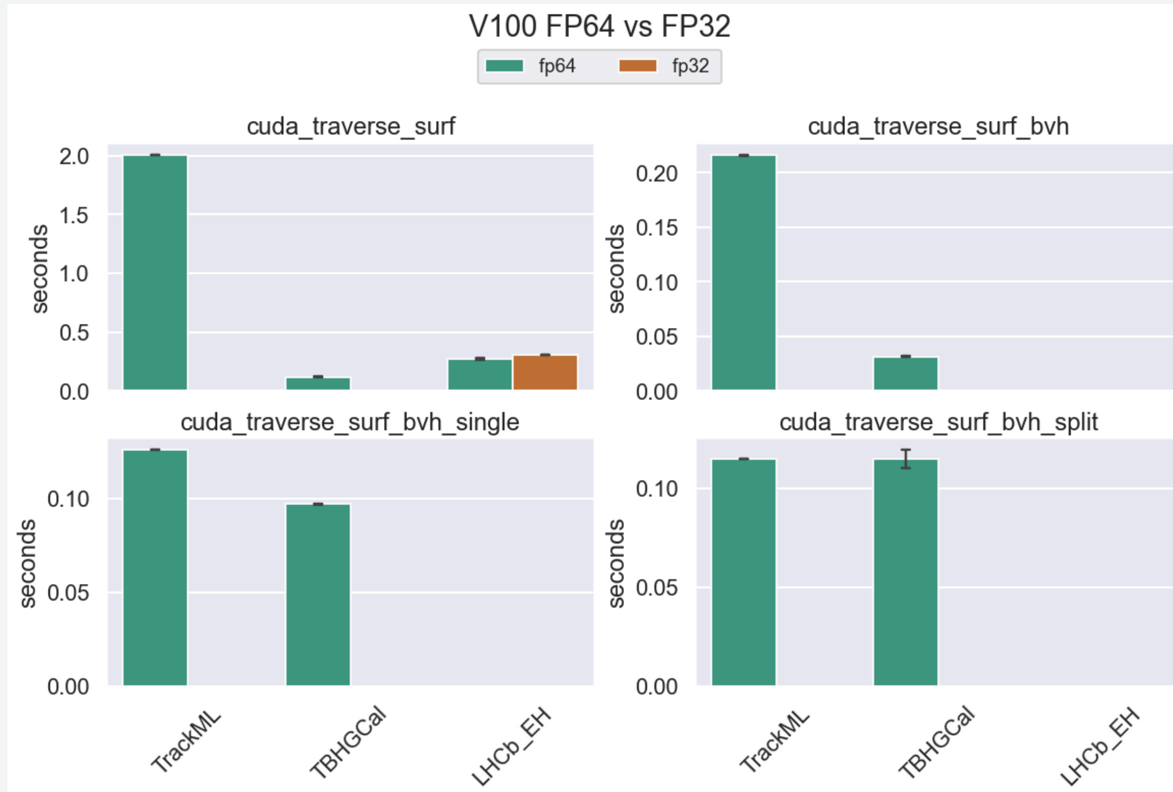
- Single precision for some but not all `Real`
- Reduces register pressure
- Reduces volume of data movement
- 2 FP32 for each FP64 unit on HPC GPUs
  - 32:1 or 64:1 on most other NVIDIA GPUs

```
// testRaytracing.h
using Real_t = float;
```

- ❌ *“not stable on most geometries”*
  - Assertions triggered by many geometries
  - Launch failures, incomplete profile reports
  - `bvh_single_step` and `bvh_split_step` run indefinitely
    - For some geometries, some of the time

Unsuccessful FP32 runs on V100

❌ Very few successful runs / configurations



Very partial FP64 vs FP32 results

`ProjectRaysSurf` on V100

Precision	Reg/Thread	LHCb 524228 rays
FP64	255	243ms
FP32	208	298ms

- Unintentionally created a 1.3TB log file…

```
1 $ du -sh slurm-842405.out
2 1.3T    slurm-842405.out
```

- Needs investigation

Increased block size

- `testRaytracing.cu` uses a fixed number of threads per block of `32`
  - Different block sizes may impact performance

```
1 // testRaytracing.cu
2   constexpr int initThreads = 32;
3   int initBlocks            = (nrays + initThreads - 1) / initThreads;
```

- Alternatively, can use a per-kernel occupancy API method to maximise occupancy
  - i.e. `cudaOccupancyMaxPotentialBlockSize`
  - Specialises for the target GPU architecture.

```
1   int minGridSize = 0;
2   int blockSize = 0;
3   int gridSize = 0;
4   // ...
5   cudaOccupancyMaxPotentialBlockSize(&minGridSize, &blockSize, PropagateRaysSurf);
6   gridSize = (nrays + blockSize - 1) / blockSize;
7   PropagateRaysSurf<<<gridSize, blockSize>>>(nrays, ...);
```

Increased block size

- `trackML.gdml`, `524228` rays, V100

Strategy	Reference Time(s)	Time(s)	Selected Blocksizes
`PropagateRaysSurf`	`2.006`	`2.206`	`256`
`PropagateRaysSurfBVH`	`0.215`	`0.229`	`256`
`bvh_single_step`	`0.127`	`0.131`	`256` & `1024`
`bvh_split_step`	`0.116`	`0.174`	`384`, `256` & `1024`

- Not an improvement on V100
  - Try on other architectures?
  - Try other values between `32` and `256`?

University of Sheffield

Thank you

Additional Slides

CMake Configuration

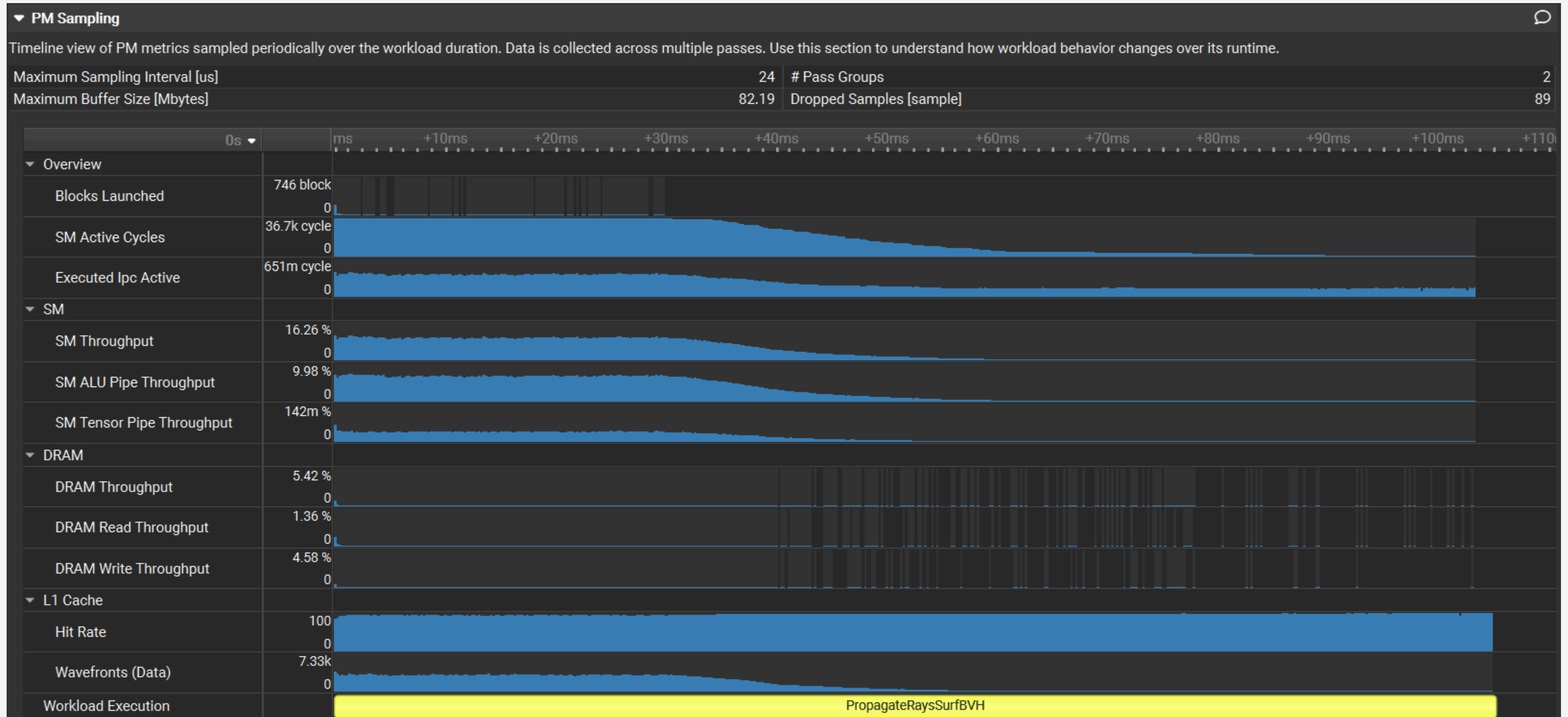
```
1 cmake -S . -B build \
2       -DCMAKE_BUILD_TYPE=Release \
3       -DCMAKE_CUDA_ARCHITECTURES="70;80;90" \
4       -DVECGEOM_ENABLE_CUDA=ON -DVECGEOM_GDML=ON \
5       -DBACKEND=Scalar -DVECGEOM_USE_NAVTUPLE=ON \
6       -DVECGEOM_BVH_SINGLE=ON -DVECGEOM_BUILTIN_VECCORE=ON
```

Surface model construction timeline

- Larger/more complex geometries would benefit from solid -> surface conversion optimisation
- 524228 rays on GH200 in FP64

University of Sheffield

Workload imbalance



PM Sampling report showing workload imbalance for LHCb with 524228 rays on GH200