Offline Track Reconstruction on GPUs for the ATLAS Experiment

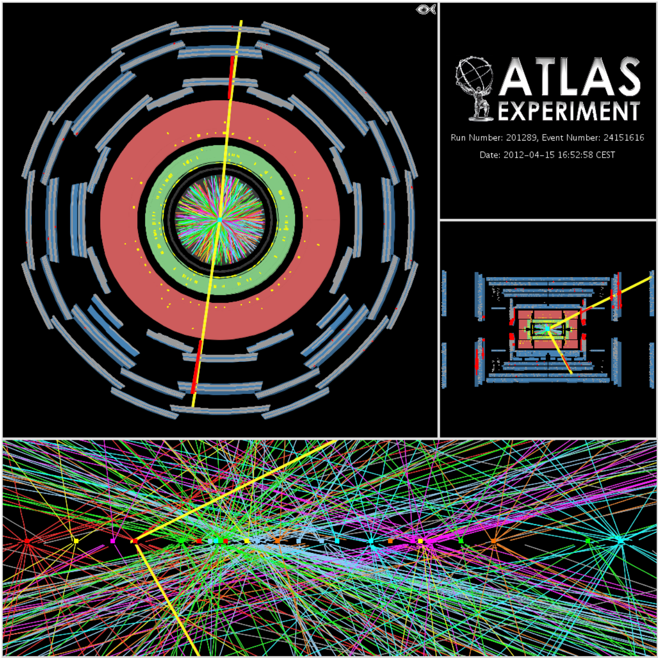
Sebastian Artz Johannes Mattmann Christian Schmitt

Johannes Gutenberg-Universität Mainz, Institut für Physik

Graphics Processing Units (GPUs) in High Energy Physics
Workshop, DESY, 15.04.2013

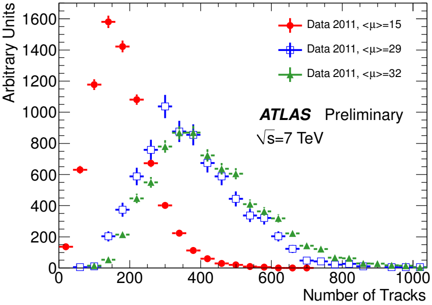
JOHANNES GUTENBERG UNIVERSITÄT MAINZ
PRISMA Cluster of Excellence
FSP 101 ATLAS
Bundesministerium für Bildung und Forschung

Content


ATLAS EXPERIMENT
Run Number: 201289, Event Number: 24151616
Date: 2012-04-15 16:52:58 CEST

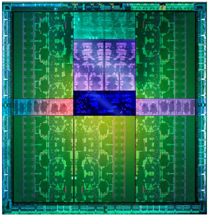
Motivation: Why using GPUs for track reconstruction?

- reconstruction time per event around 15-20 s
- ca. 400 events recorded per second
- high pile-up causes tremendous combinatorial complexity
- necessity of raising the p_T cut



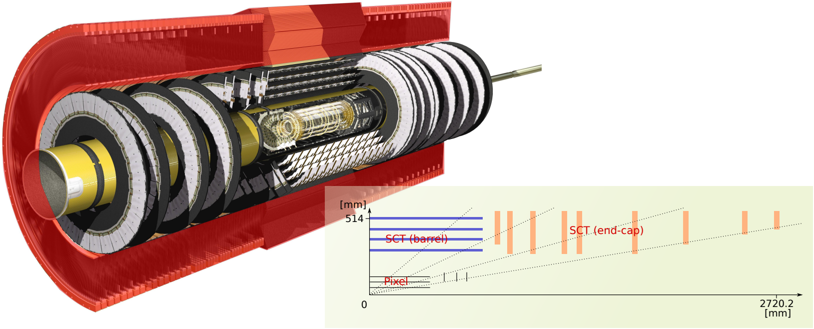
tracks do not have mutual dependencies → algorithm predestined for parallel implementation

The GPUs used for performance measurement



	GTX 460	GT 520	GTX 680
CUDA cores	336	48	1536
global memory (MB)	768	2048	2048
graphics clock (MHz)	650	810	1006 - 1058
memory clock (MHz)	1700	900	1502
GPU type	GF104	GF119	GK104

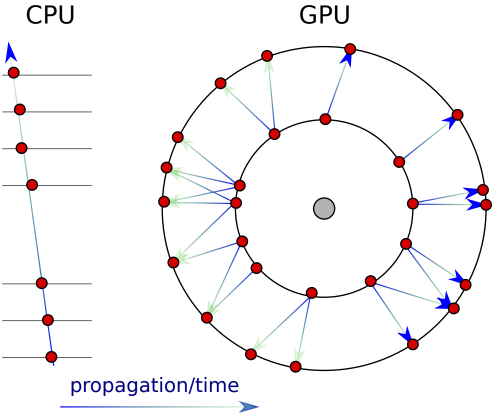
ATLAS Inner Detector: Pixel and Silicon Strip Detector



Highlighted regions:

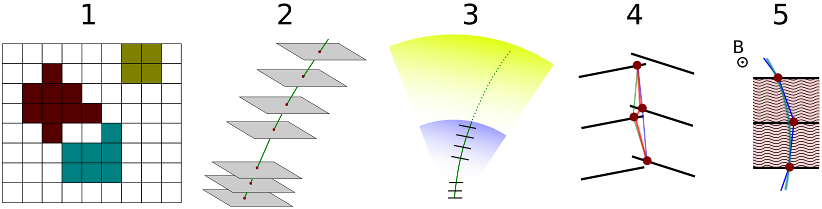
- Si pixel detectors in barrel (center region) & endcaps: 3 layers
- Si strip detectors:
  - barrel region: 4 double layers (with stereo angle)
  - endcap region: 9 discs

Basic parallelisation approach



- current CPU implementation: sequential process for each possible track
- parallel GPU implementation: parallel process for all (or subset of) tracks in one state

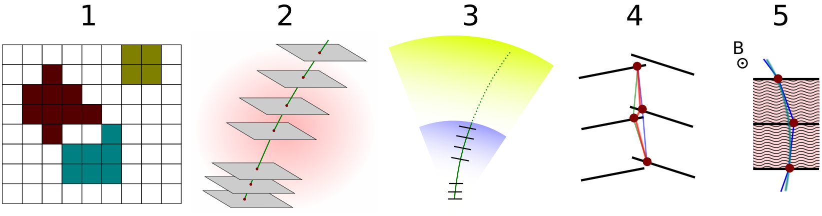
Track reconstruction chain



1. Hit calibration and clustering
2. Seed search and track extrapolation in silicon detector region
3. Track extension into tracking chamber region
4. Track revision (ambiguity solving)
5. Final track fit based on track candidates using detailed magnetic field map and material effects

First approach: focus on step with high combinatorics

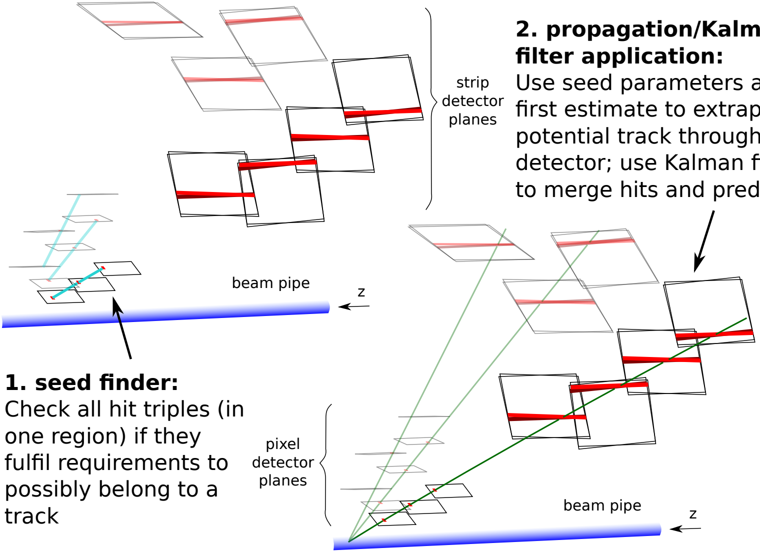
Track reconstruction chain



1. Hit calibration and clustering
2. Seed search and track extrapolation in silicon detector region
3. Track extension into tracking chamber region
4. Track revision (ambiguity solving)
5. Final track fit based on track candidates using detailed magnetic field map and material effects

First approach: focus on step with high combinatorics

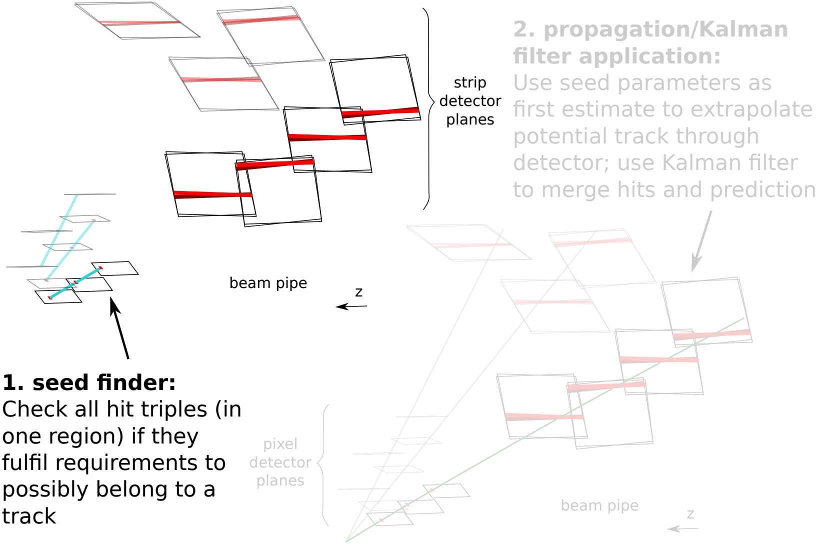
Reconstruction steps implemented on GPU



2. propagation/Kalman filter application:
Use seed parameters as first estimate to extrapolate potential track through detector; use Kalman filter to merge hits and prediction

1. seed finder:
Check all hit triples (in one region) if they fulfil requirements to possibly belong to a track

Reconstruction steps implemented on GPU



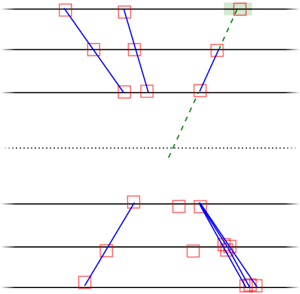
1. seed finder:
Check all hit triples (in one region) if they fulfil requirements to possibly belong to a track

2. propagation/Kalman filter application:
Use seed parameters as first estimate to extrapolate potential track through detector; use Kalman filter to merge hits and prediction

Testing the hit combinations

seed criteria

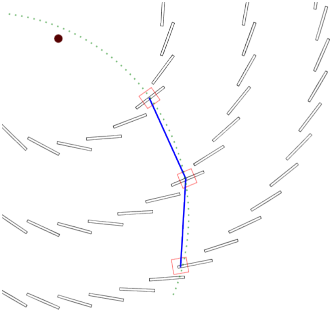
- hits on a straight line in rz-plane?
- $\eta < 2.5$ and origin within barrel region?
- does the track bend?
- minimal circle radius ($\rightarrow p_{T,min}$)?
- distance between circle and assumed impact parameter?



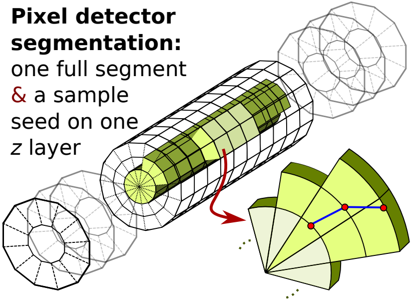
Testing the hit combinations

seed criteria

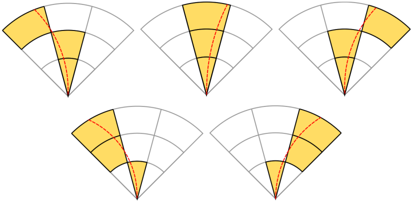
- hits on a straight line in rz-plane?
- $\eta < 2.5$ and origin within barrel region?
- does the track bend?
- minimal circle radius ($\rightarrow p_{T,min}$)?
- distance between circle and assumed impact parameter?



Seed finder implementation

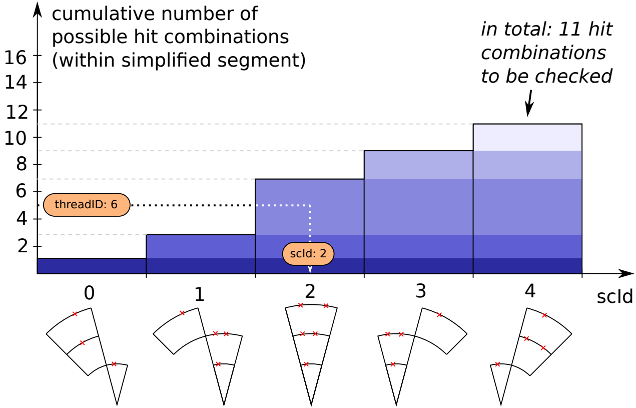


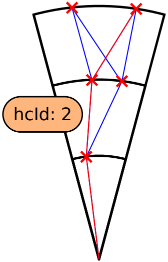
Pixel detector segmentation: one full segment & a sample seed on one z layer



- optimization: no need to check any hit triple, constraints from $p_{t,\mathrm{min}}$, polar angle etc. implied in spatial segmentation
- CPU: sort *global* hits in segments (array + indices)
- save reasonable segment combinations in lookup table
- check hit combinations: 1 thread $\hat{=}$ 1 hit triple

Obtaining the seed candidate for each thread

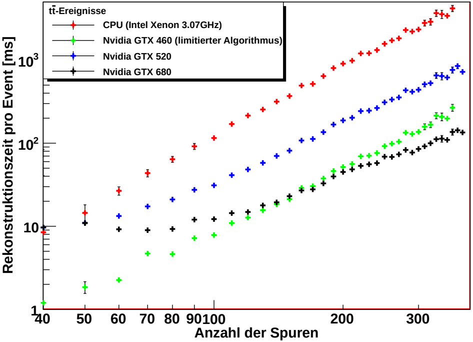




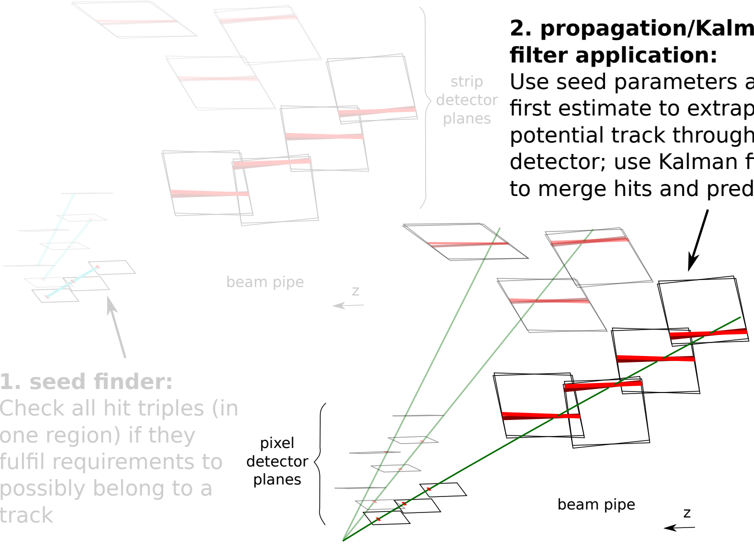
- **left (1):** lookup process to map threadID to segment combination, $\mathcal{O}(\log N)$)
- **right (2):** enumerate all hit combinations within segment combination → 2nd reverse lookup (cf. 3D array indexing with varying dimensions via integer division and modulo operation)

Seed finder results

- Performance gain compared to *our own* CPU version (∼ factor 40)
- Performance gain compared to previous, limited GPU version



Reconstruction steps implemented on GPU



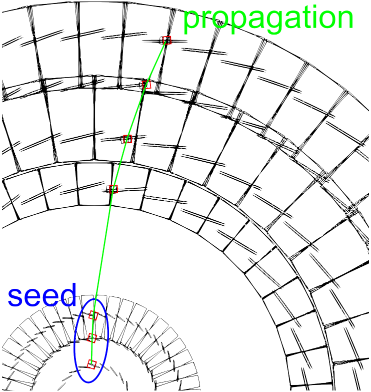
1. seed finder:
Check all hit triples (in one region) if they fulfil requirements to possibly belong to a track

2. propagation/Kalman filter application:
Use seed parameters as first estimate to extrapolate potential track through detector; use Kalman filter to merge hits and prediction

The propagation

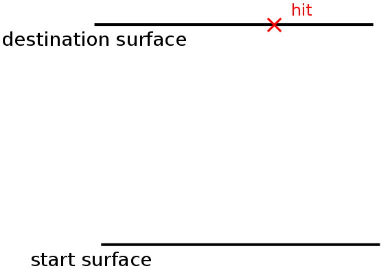
Propagation from one layer to the next

- parallel: find intersection point with subsequent layer
- parallel: Propagation of parameters and covariance matrix plus Kalman filter processing

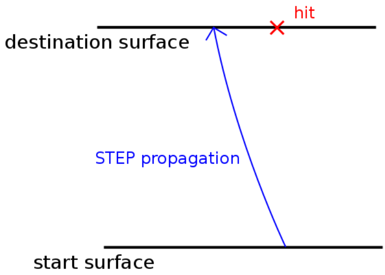


View along the beam axis

Propagation and Kalman filter

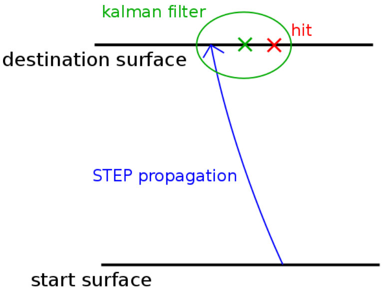


Propagation and Kalman filter



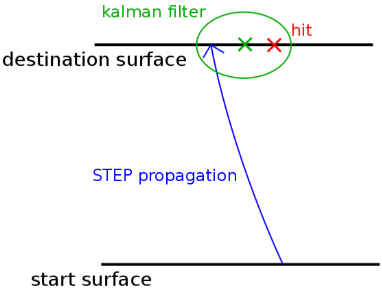
- Propagation using an adaptive Runge-Kutta-Nyström algorithm

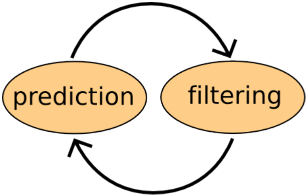
Propagation and Kalman filter



- Propagation using an adaptive Runge-Kutta-Nyström algorithm
- Kalman filter: merge propagation result and hit information, weighted by their respective errors

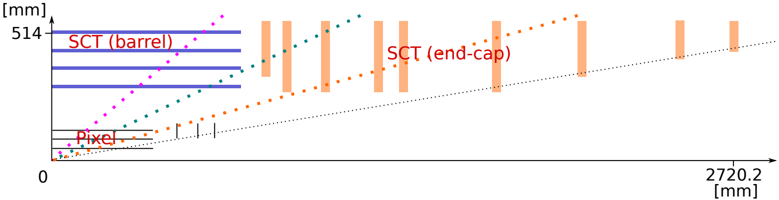
Propagation and Kalman filter





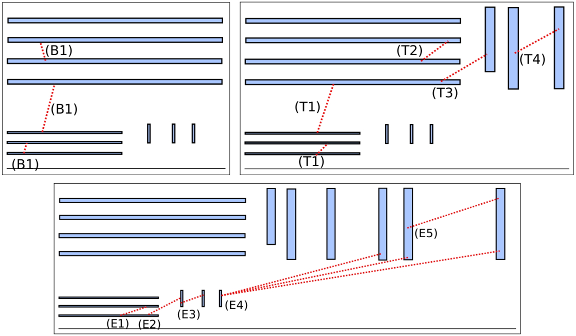
- Propagation using an adaptive Runge-Kutta-Nyström algorithm
- Kalman filter: merge propagation result and hit information, weighted by their respective errors
- Repeat the propagation and Kalman filter application up to the outermost layer

Finding the right intersection



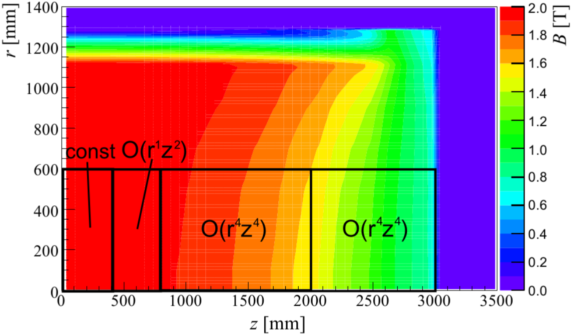
- find an intersection with barrel/disc layer
- heuristic method to find target layer
- problem: faster and slower intersections are calculated in parallel → loss of efficiency expected (*warp divergency*)
- solution: presort tracks in the seedfinding algorithm (pure barrel tracks, certain endcap tracks, transition area tracks)

Finding the right intersection



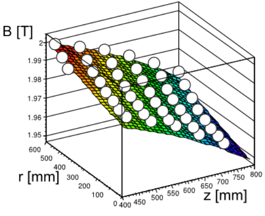
- find an intersection with barrel/disc layer
- heuristic method to find target layer
- problem: faster and slower intersections are calculated in parallel → loss of efficiency expected (*warp divergency*)
- solution: presort tracks in the seedfinding algorithm (pure barrel tracks, certain endcap tracks, transition area tracks)

Handling the inhomogeneous magnetic field

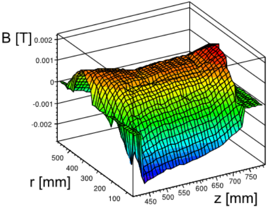


- Magnetic field in the endcaps region is not constant
- GPU memory is limited → trying to avoid transferring the magnetic field map
- Polynomial fit of the B_z component in the rz plane (assuming ϕ dependency to be small)

Handling the inhomogeneous magnetic field



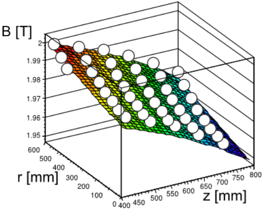
data (dots) + fit



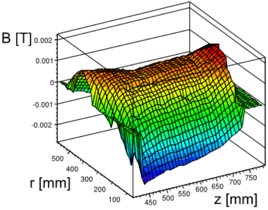
fit - data

- Magnetic field in the endcaps region is not constant
- GPU memory is limited → trying to avoid transferring the magnetic field map
- Polynomial fit of the B_z component in the rz plane (assuming ϕ dependency to be small)

Handling the inhomogeneous magnetic field



data (dots) + fit



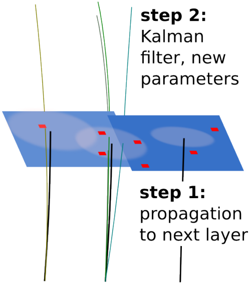
fit - data

- Magnetic field in the endcaps region is not constant
- GPU memory is limited → trying to avoid transferring the magnetic field map
- Polynomial fit of the B_z component in the *rz* plane (assuming ϕ dependency to be small)
- inhomogeneous field → numerical propagation

Propagation: handling track multiplicity changes

- **challenge:** in each step tracks can end (maximum number of 'holes' reached) or split up (more than one hit could possibly belong to the current track)
- **approach:** each track should be handled by one track → rearrange tracks once hits on subsequent layer have been 'seen'
- **implementation:** basically two GPU kernel calls: rough search for potential track hits (step 1), further processing of hit data and propagated original track parameters (step 2) with matched track numbers

#threads: 4
1 thread/target hit
= filtered parameter set



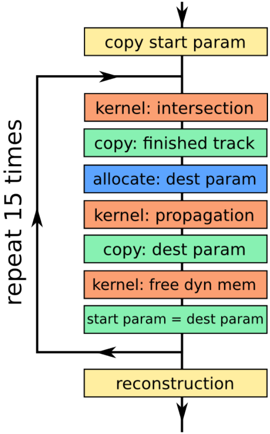
step 2: Kalman filter, new parameters

step 1: propagation to next layer

#threads: 3
1 thread/incoming track
= initial parameter set

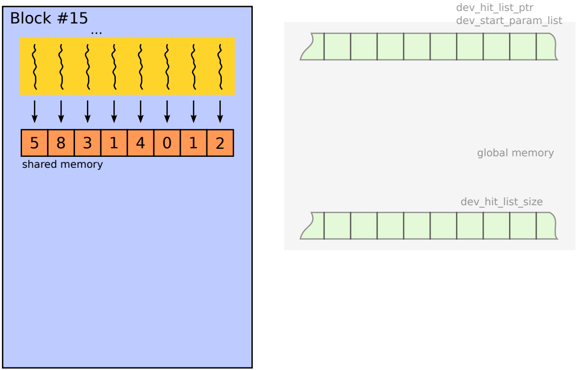
Overview of the parallel GPU algorithm

- copy start parameters to GPU
- for each layer
    - find intersections & save them dynamically
    - copy finished track informations to main memory
    - allocate memory for parameters on destination surface
    - propagate to destination surface
    - copy parameters to main memory
    - free dynamic memory from intersection step
    - use resulting parameters as new start parameters
- reconstruct tracks from last layer parameters



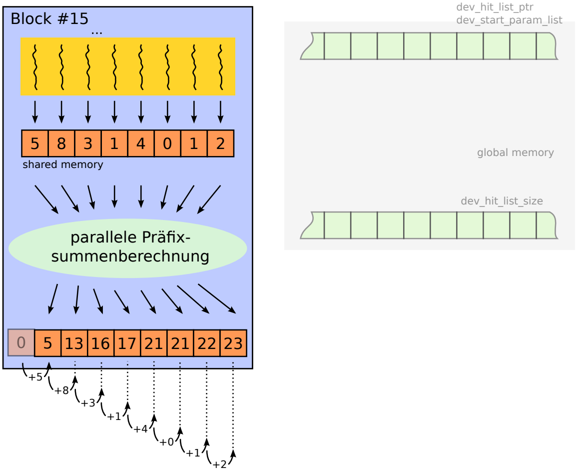
Thread data mapping between kernel calls

preparePropagation(..) [Kernel 1]



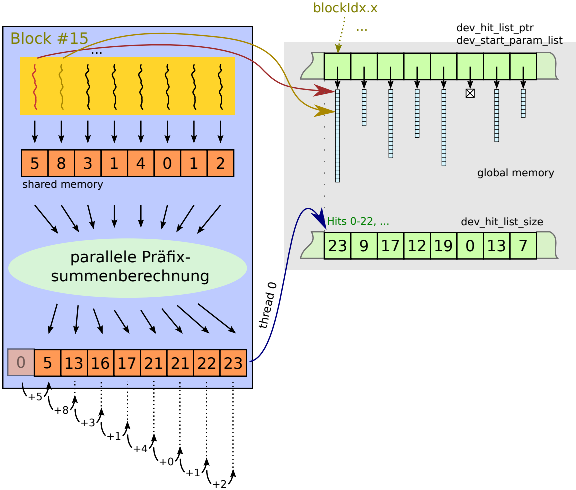
Thread data mapping between kernel calls

preparePropagation(..) [Kernel 1]

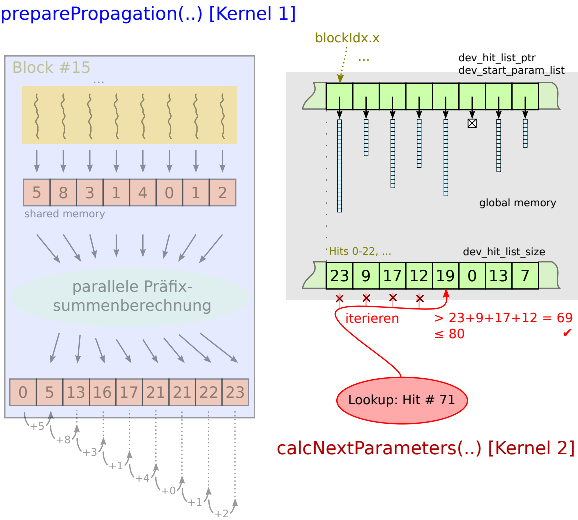


Thread data mapping between kernel calls

prepareProgagation(..) [Kernel 1]



Thread data mapping between kernel calls



preparePropagation(..) [Kernel 1]

Block #15

...

5 | 8 | 3 | 1 | 4 | 0 | 1 | 2

shared memory

parallele Präfix-summenberechnung

0 | 5 | 13 | 16 | 17 | 21 | 21 | 22 | 23

+5 +8 +3 +1 +4 +0 +1 +2

blockIdx.x

...

dev_hit_list_ptr
dev_start_param_list

global memory

Hits 0-22, ...

dev_hit_list_size

23 | 9 | 17 | 12 | 19 | 0 | 13 | 7

iterieren

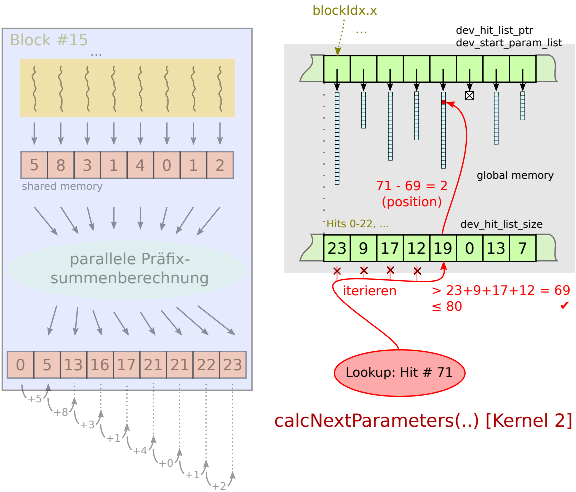
> 23+9+17+12 = 69
≤ 80 ✔

Lookup: Hit # 71

calcNextParameters(..) [Kernel 2]

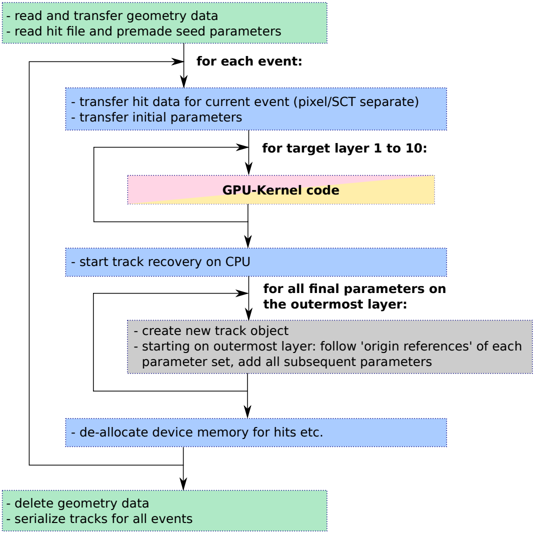
Thread data mapping between kernel calls

preparePropagation(..) [Kernel 1]



calcNextParameters(..) [Kernel 2]

Reconstruction “frame” on CPU



- read and transfer geometry data
- read hit file and premade seed parameters

for each event:

- transfer hit data for current event (pixel/SCT separate)
- transfer initial parameters

for target layer 1 to 10:

GPU-Kernel code

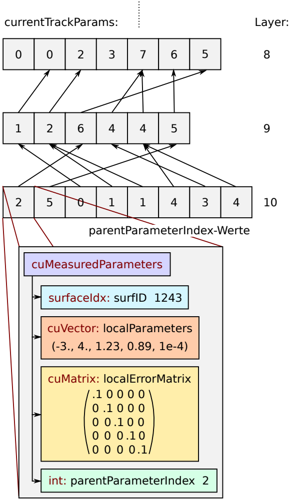
- start track recovery on CPU

for all final parameters on the outermost layer:

- create new track object
- starting on outermost layer: follow 'origin references' of each parameter set, add all subsequent parameters

- de-allocate device memory for hits etc.

- delete geometry data
- serialize tracks for all events



currentTrackParams:

Layer:

0 0 2 3 7 6 5 — 8

1 2 6 4 4 5 — 9

2 5 0 1 1 4 3 4 — 10

parentParameterIndex-Werte

cuMeasuredParameters

surfaceIdx: surfID 1243

cuVector: localParameters (-3., 4., 1.23, 0.89, 1e-4)

cuMatrix: localErrorMatrix

$$\begin{pmatrix} .1 & 0 & 0 & 0 & 0 \\ 0 & .1 & 0 & 0 & 0 \\ 0 & 0 & .1 & 0 & 0 \\ 0 & 0 & 0 & .1 & 0 \\ 0 & 0 & 0 & 0 & .1 \end{pmatrix}$$

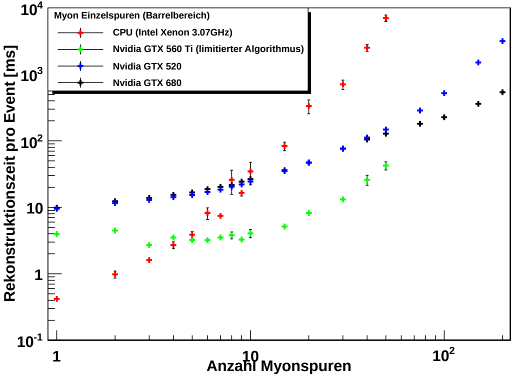
int: parentParameterIndex 2

Matrix and vector operations on the GPU

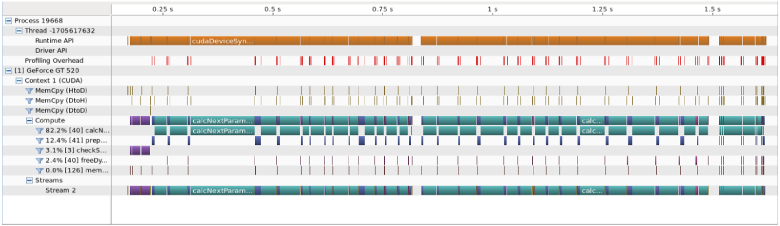
- set of matrix/vector operations required for
    - geometry (3D vectors, 3 × 4 homogeneous transformation matrices)
    - 'parameter space' (parameter representation, covariance matrix, parameter transformation, Kalman filter calculations)
- CUDA contains CUBLAS but unsuitable (and was only accessible from host), commercial library available but little documentation
- **therefore:** implemented an own subset of CLHEP-like methods to ease porting of existing code and increase readibility (including operator overloading etc.)
- **but:** only limited set of functions implemented, optimized for specific requirements (i.e. 5 × 5 and 2 × 2 matrices)

Performance of the propagation progress

- small overhead of the GPU version
- performace gain already achieved for low track numbers
- again comparing GPU version to *our* CPU version
- around 2 orders of magnitude speedup (preliminary)

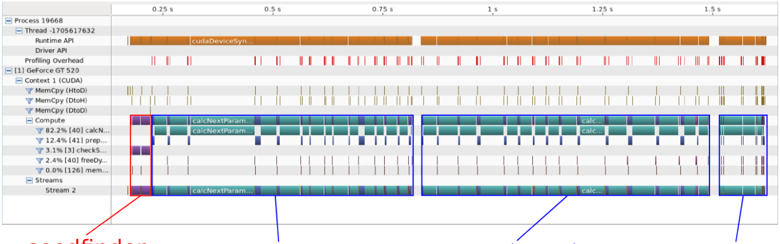


Profiling with Nsight



profiling one muon event with 500 tracks

Profiling with Nsight



seedfinder

propagation transition area seeds

propagation endcap seeds

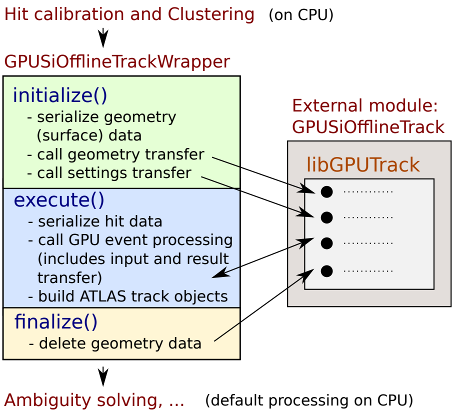
propagation barrel seeds

profiling one muon event with 500 tracks

Integration into the ATLAS software framework

- Basic approach: optional module within the ATLAS software framework to replace the corresponding CPU module if GPU/CUDA available
- Input data access: Wrapper module as part of the framework obtaining input data from 'storage' system and returning converted results
- Special compiler (NVCC) needed, special linking options required etc. → actual CUDA code in external library, no changes to the framework build system necessary

Integration into the ATLAS software framework - details



Hit calibration and Clustering (on CPU)

GPUSiOfflineTrackWrapper

initialize()
- serialize geometry (surface) data
- call geometry transfer
- call settings transfer

execute()
- serialize hit data
- call GPU event processing (includes input and result transfer)
- build ATLAS track objects

finalize()
- delete geometry data

External module: GPUSiOfflineTrack

libGPUTrack

Ambiguity solving, ... (default processing on CPU)

- Runtime properties enable/disable GPU processing path
- algorithm structure: 3 steps - initialization (once per run), execution (per event), finalization (once per run), identical structure for the external module
- conversion between full-featured framework classes and slim 'C struct' arrays

Conclusion and outlook

Results

- small and mostly constant overhead from memory transfer and lookup table creation
- seedfinder: factor ∼40 speedup (w.r.t. own CPU version)
- propagation: about 2 orders of magnitude speedup (w.r.t. own CPU version, preliminary)
- technical implemetation finished

Outlook

- incluce first SCT layer in the seed search
- performance optimizations
- further performance measurements
- testing/verification of framework module

Thanks for your attention!