Update on the Track Finder benchmark

18.11.2008

Håvard Bjerke

Outline

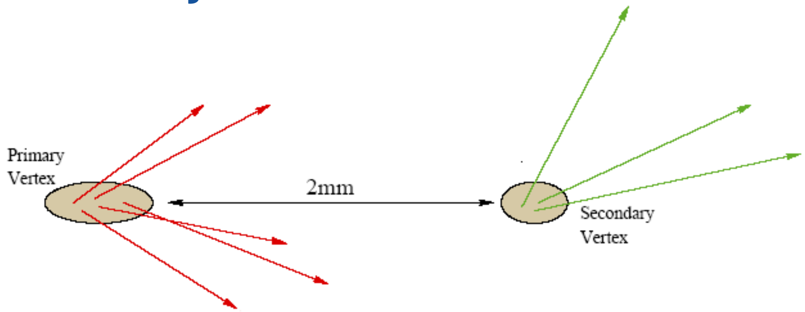
- Motivation
- Background
  - HLT
  - AliHLT framework
- Track finder
  - Visualization
  - Algorithms
- Vectorization & multi-threading

Motivation

- Make the HLT software ready for the many-core era
- Explore optimization methods
  - Vectorization (SIMD)
  - Multi-threading
- Forward-scaling for future architecture
  - Many cores
  - Wider vector registers
    - e.g. AVX: 256 bits, new instructions

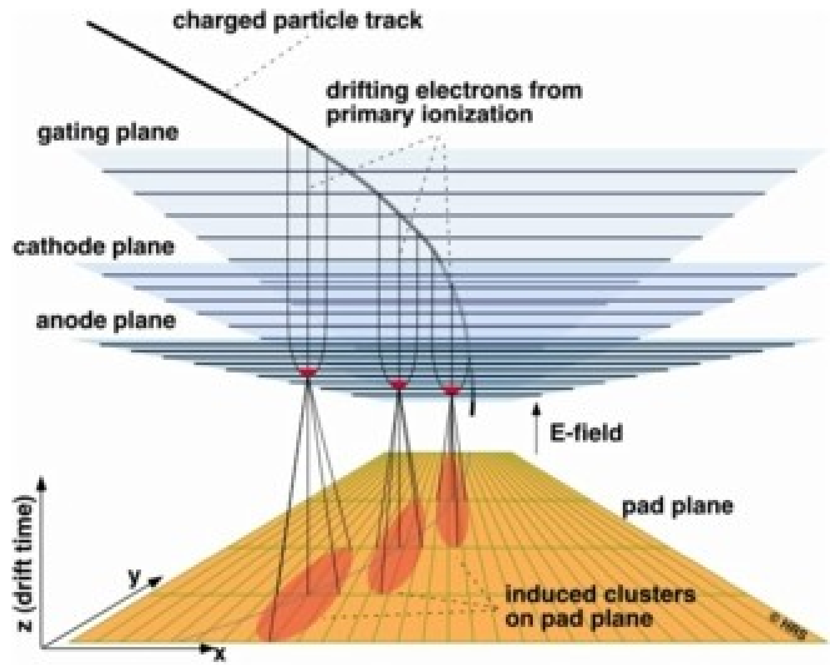
Reconstruction Challenges

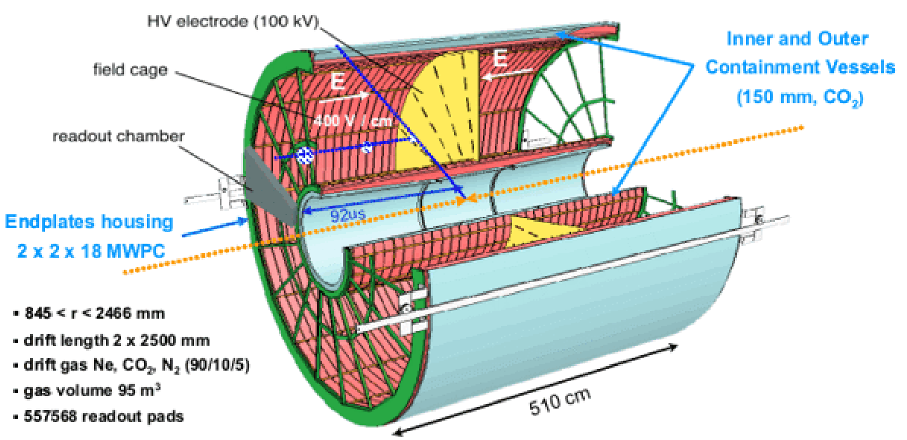
1. Track finding
   - High frequency of collisions
   - A lot of irrelevant particle noise
   - Needs to be filtered in order to concentrate on most important data
2. Track fitting
   - Estimate the real particle trajectories
3. Find vertices



HLT Tracking Benchmarks

- Collaboration with Intel Bruhl and KIP
- Track Finder: *Reconstructing particle tracks from events*
  - Under development
- Track Fitter: *Fit track parameters*
  - Highly thread and SIMD parallel benchmark



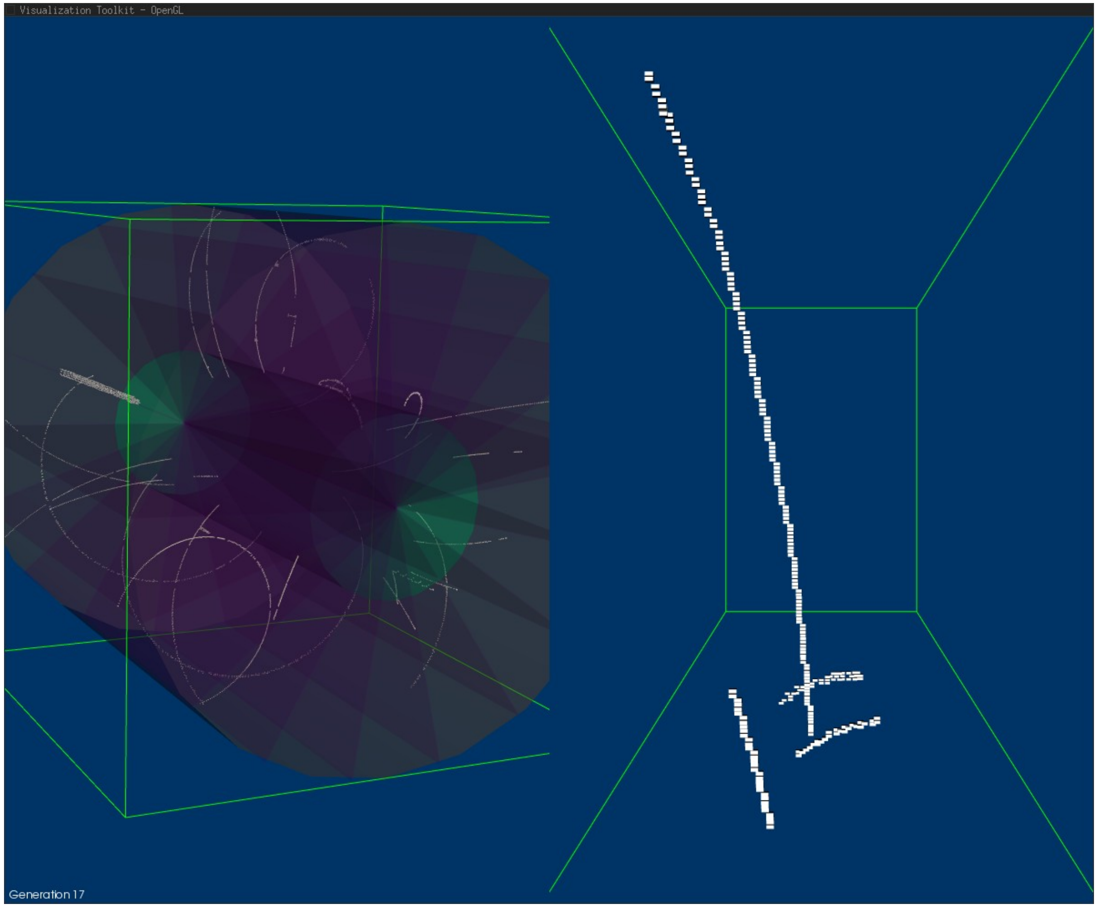


AliHLT Framework

- Already contains
  - MC simulations
  - Event reader
  - Tracker
  - Fitter
  - Merger
  - Track writer
  - Performance calculation
- Objectives
  - Replace Tracker with faster and more efficient code
  - Exploit parallel hardware
  - Build a standalone benchmark
  - Better visualization
  - Process Pb events

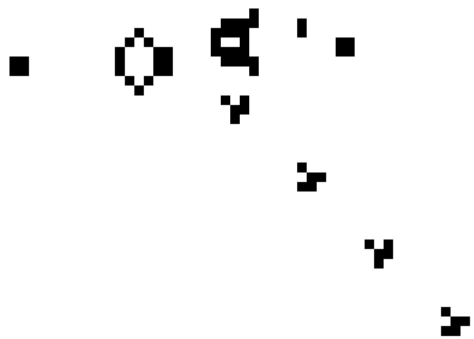
Interactive 3D visualization

- With help from Intel



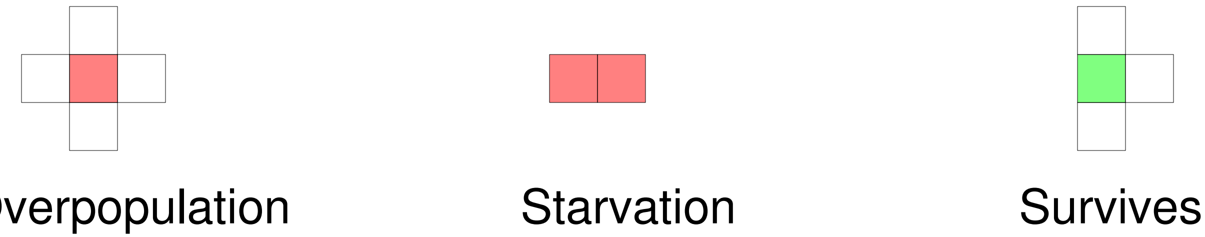
Track finder algorithm

- Cellular automaton, based on Conway's Game of Life

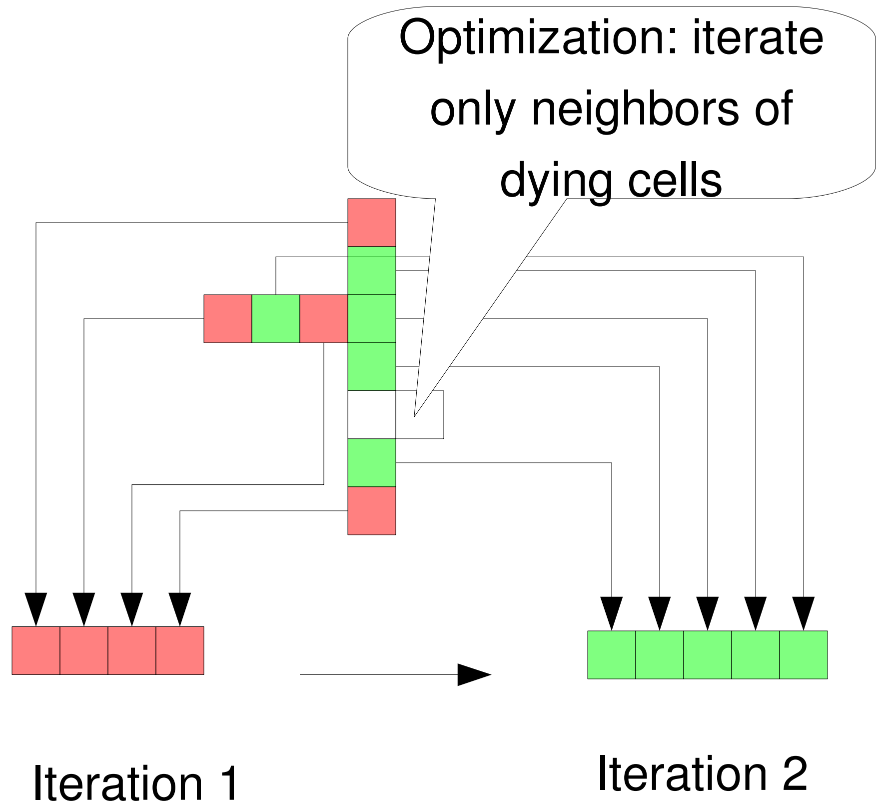


Interesting CA properties:
- Simple
- Local
- Parallel

Game of Life data parallelism

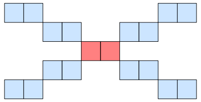


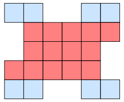
- Data parallelism *within* iterations
- Dependency *between* iterations

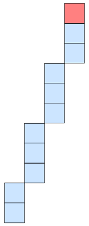


Cellular Automaton Track Finder

- Overpopulation: (difficult to discern tracks)
  - More than two neighbours
  - An overlap of more than two cells
- Starvation [changed]: No top neighbour (track endings)





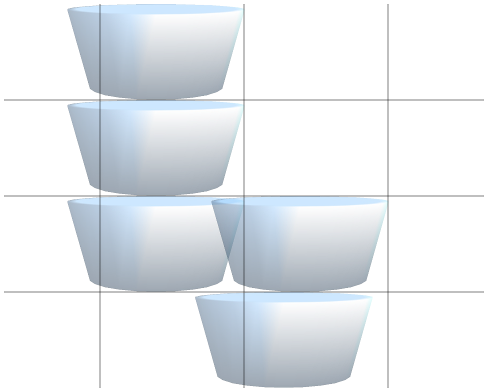


Deviation from original algorithm

- Suggest hybrid (but equivalent) solution:
  - CA is expensive – execute only one iteration
  - Phase 1: Use CA to kill “noisy” clusters – only one iteration
  - Phase 2: Iterate through the alive cells, since we know that these are easy to find tracks
- Results are encouraging
  - Reconstruction efficiency is as good or better
  - Order of magnitude speedup

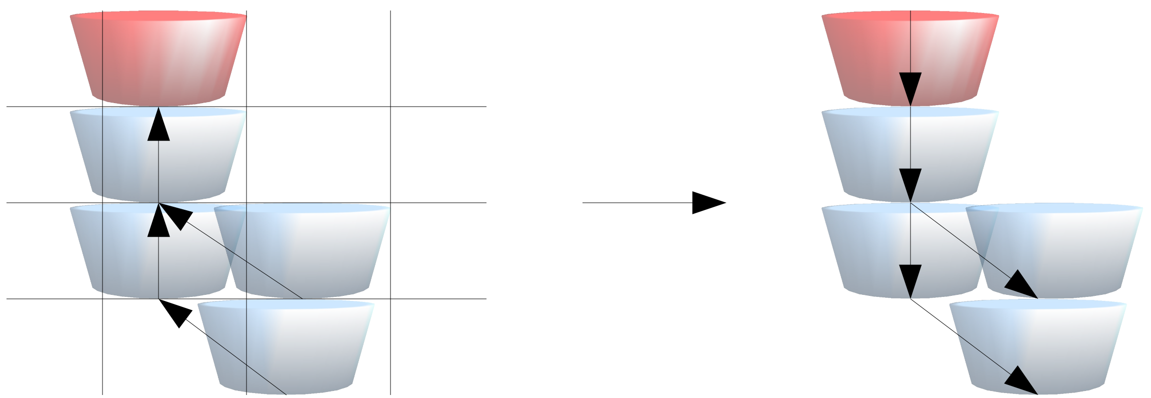
Simplification of the problem (1)

- Digitization - binning clusters into a discrete grid allows O(1) access
  - But the grid is large, at least 35 MB
  - Especially expensive with low density
  - Precision is lost – tradeoff between size (granularity) and precision
- Compromise
  - Bin only by rows
  - Local cluster IDs



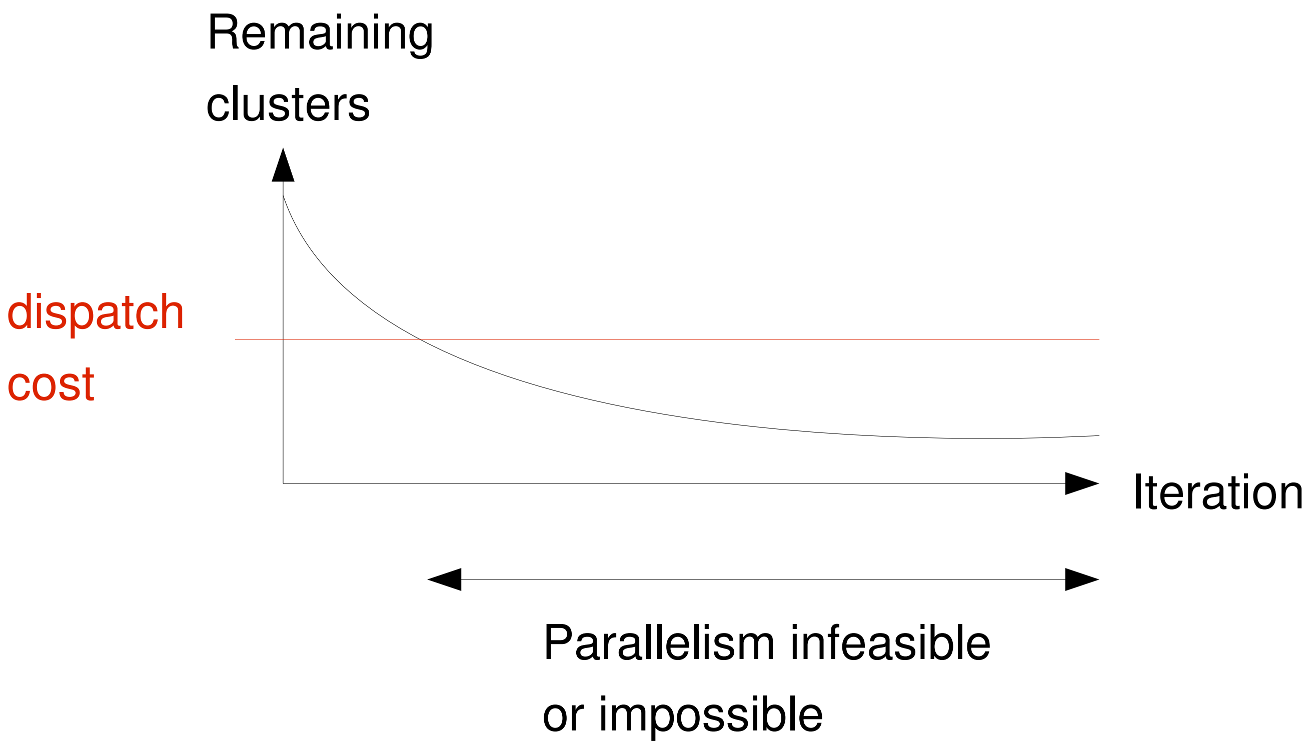
Simplification of the problem (2)

- Only one CA iteration
  - CA is expensive
- Tracks then found iteratively
  - But very little work per iteration
  - Easily parallelizable



Track level parallelism

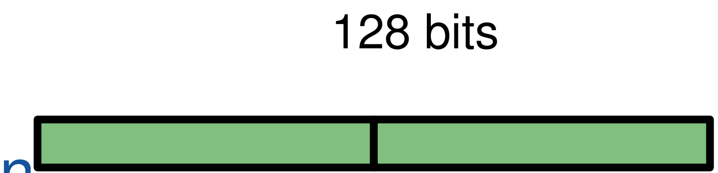
- Some tracks are longer than others



Opportunities for data parallelism

- Vectorization
  - Experiments with Game of Life show good speedup from vectorization
  - Compute neighbours of a vector of cells in parallel
- Multi-threading
  - Compute sets of vectors in parallel
  - Find linked tracks in parallel

- EPI64
  - 2 64-bit integer calculations per instruction



- EPI32
  - 4 32-bit integer calculations per instruction

…

- EPI8
  - 16 8-bit integer calculations per instruction

Cluster ID precision

- Cluster IDs are needed to calculate efficiency
  - Should be at least 32 bit
- Lower precision needed locally to discern tracks in the grid
- Example: finding neighboring cluster
  - If cluster ID is global, 32 bit integer → 4 clusters in parallel
  - If cluster ID is local to row, 16 bit integer → 8 clusters in parallel
    - map needed between <local_id, row> and global_id

Vectorization techniques

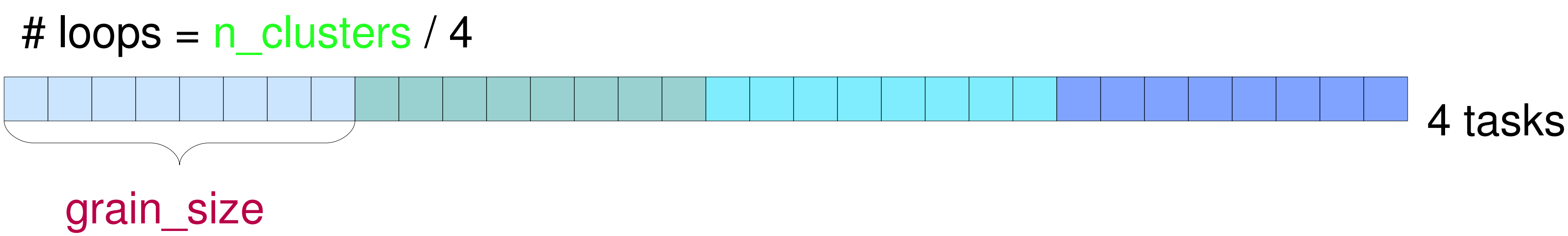
- From game of life
  - _mm_loadu_si128 - load n neighbors from grid
  - _mm_slli_si128 - shift through neighbors
- Masks
  - SSE is “streaming” calculation, so we also calculate invalid results
    - _mm_and_si128(result, mask)
- Optimization: Avoid checking bounds
  - Put zeros on the edges of the grid

TBB Multi-threading (1)

- Uses Intel Threading Building Blocks for multithreading
- parallel_for - iterates over a vector of active clusters in parallel
- concurrent_vector - allows concurrently pushing objects

TBB Multi-threading (2)

- Intel Threading Building Blocks: parallel_for
  - #tasks = #clusters / grain_size
  - #threads <= #tasks




```
for(int i = 0; i < n_clusters / 4; i++){
      exec_active_cluster(cluster_vector[i], ...);
}
```

```
parallel_for(blocked_range<int>(
      0, n_clusters / 4, grain_size),
      ApplyNextGen(cluster_vectors, ...));
```

Debugging tools

- Mix of tools needed in order discover hotspots and bugs
- Pfmon – Precise, but doesn't give trace
- Gprof – Gives trace, but imprecise
- Valgrind – Useful to debug memory, but doesn't show where the biggest flux is
  - Memcheck – discover memory leaks
  - Massif – heap profiler
- Intel Thread Profiler
  - TBB gives false positives

Conclusion

- The track finder has been integrated with the AliHLT framework
  - Processes heavy ion events efficiently
  - Interactive 3D visualization developed
  - Also standalone benchmark
- CA algorithm exhibits data parallelism
  - Partial parallelization with TBB
  - SSE techniques from game of life can be reused
    - Low precision integer calculation can give high throughput