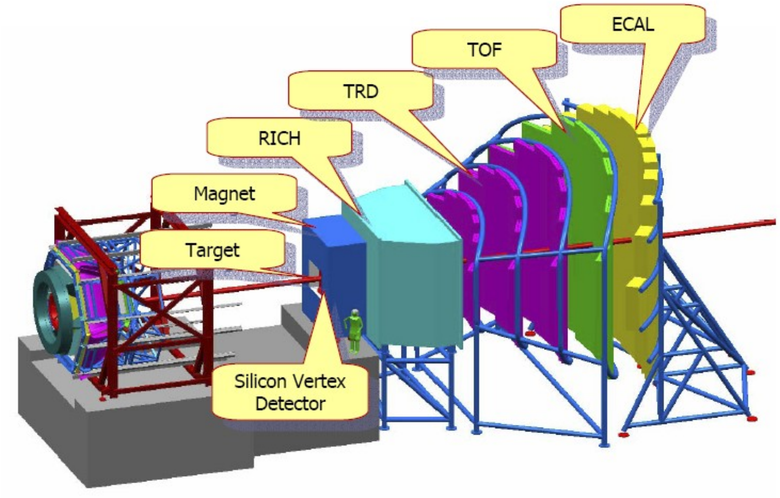
Review of a parallel High Level Trigger benchmark (using multithreading and/or SSE)

19.02.2008

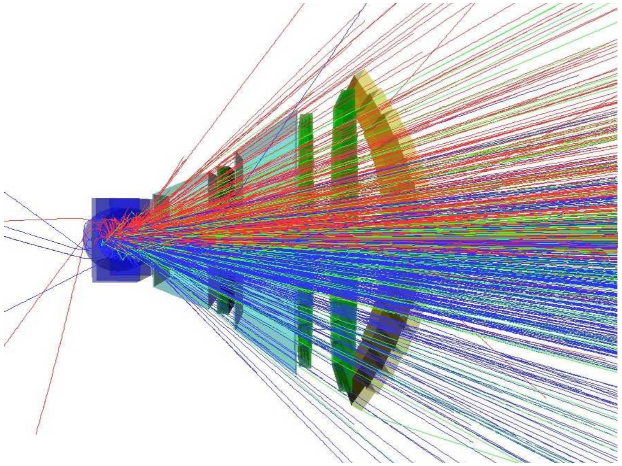
Håvard Bjerke

Background

- Reconstruction of events

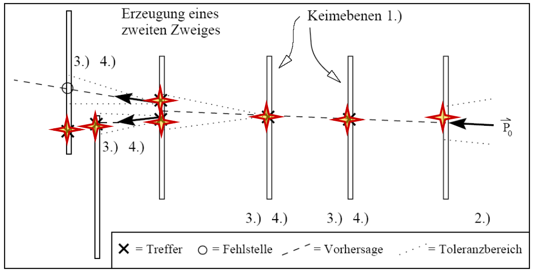


Fixed target detector



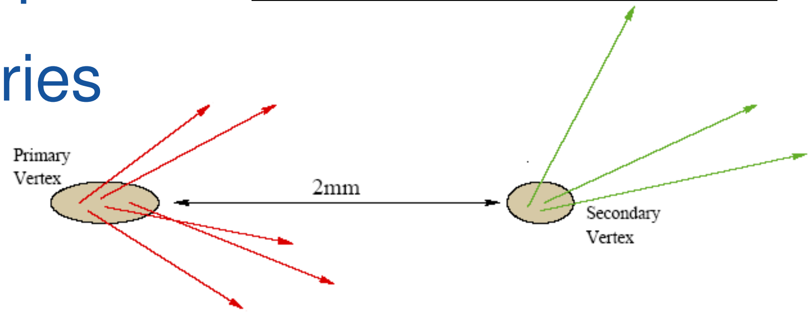
Reconstruction Challenges

1. Track finding
   - High frequency of collisions (LHC: 40 MHz)
   - A lot of irrelevant particle noise
   - Needs to be filtered in order to concentrate on most important data



2. Track fitting
   - Measurements are imprecise
   - Estimate real trajectories

3. Find vertices

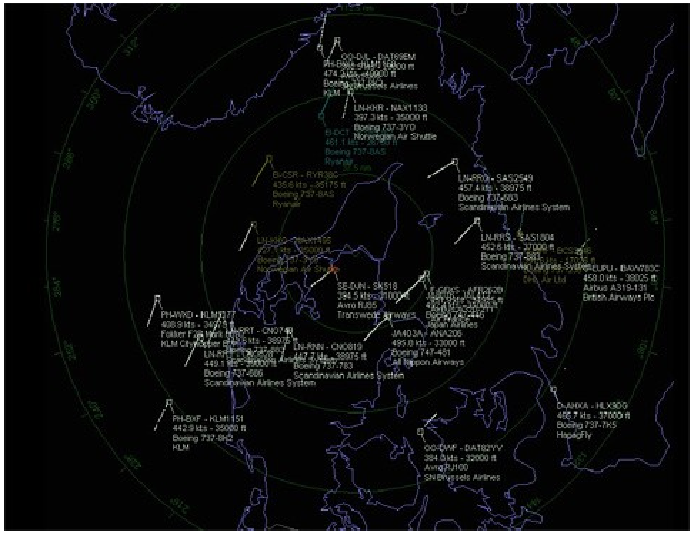


Track finding

- Filtering: Remove particle tracks that are not interesting
- Example filter rule: remove all particles that are not, for instance, muons
- Some simple rules can be applied already at the hardware level, with dedicated chips
- More advanced rules are applied in an on-line compute farm

Track fitting

- Find the real trajectory of particles from imprecise observations of the particles



- Kalman filter
  - Estimates real trajectory from imprecise measurements
  - Computes trajectory based on error correcting feedback

High-Level Trigger code

- Tracing particles through a magnetic field
- Each track is calculated independently
  - Embarrassingly parallel
- Optimization
  - Step 1: use vectors instead of scalars
    - Allows exploitation of SIMD instructions
  - Step 2: use multithreading
    - Allows exploitation of multiple cores

Explicit vectorisation

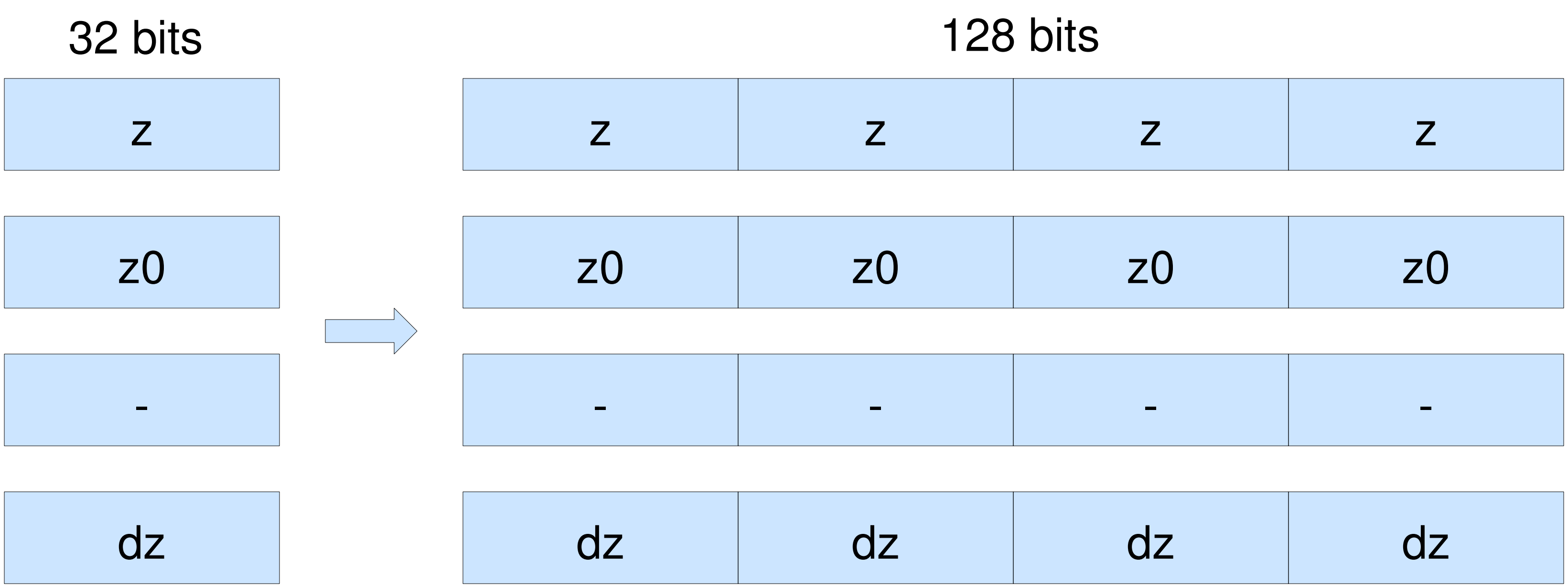
- Operator overloading allows seamless change of data types, even between primitives (e.g. float) and classes
- Two classes
  - P4_F32vec4 – packed single
    - ✓ operator + = _mm_add_ps
    - ✓ rcp = _mm_rcp_ps
  - P4_F64vec2 – packed double
    - ✓ operator + = _mm_add_pd
    - ✗ rcp: No _mm_rcp_pd! Use 1. / a

Vectorised calculation

dz = z - z0

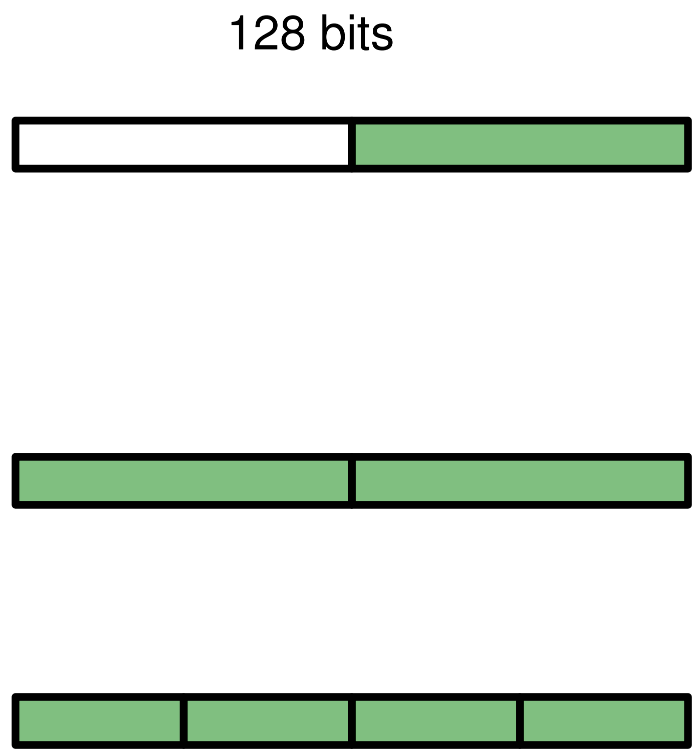
```
for #tracks {
        dz[#] = z[#] – z0[#]
}
```

```
for #tracks/4 {
        vdz[#] = vz[#] - vz0[#]
}
```



SIMD implementation

- Three modes
  - Scalar (SISD) double, "x87"
    - 1 scalar double precision calculation per instruction
  - Packed double
    - 2 scalar double precision calculations per instruction
  - Packed single
    - 4 scalar single precision calculations per instruction



Performance measurements

- Woodcrest @ 2.4 GHz using ICC 9.1

	Calculation time per track / us	Incremental speedup	Total speedup from scalar
scalar	2.6	1	1
double	1.6	1.6	1.6
single	0.7	2.3	3.7

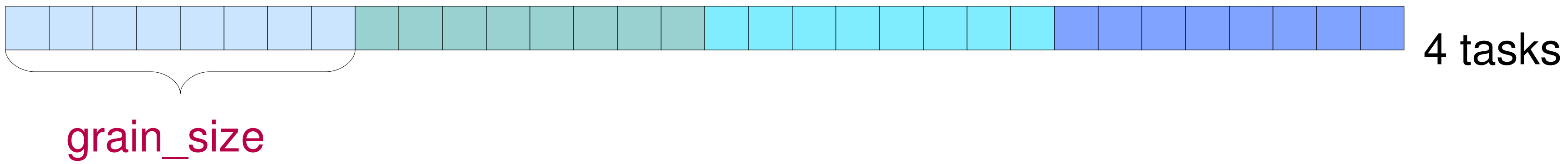
Performance counters

instruction type	scalar	double	single
computational scalar double	١٠.٦	٠	٠
computational packed double	٠	٥.٥	٠
total packed double	٠	٥.٥	٠
computational packed single	٠	٠	٢.٨
total packed single	٠	٣.٧	٤.٦
total SIMD	١٦.٩	٩.٥	٤.٧
total	٢٤.٧	١٧.٢	١٠.٩

Multithreading

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

loops = n_tracks / 4



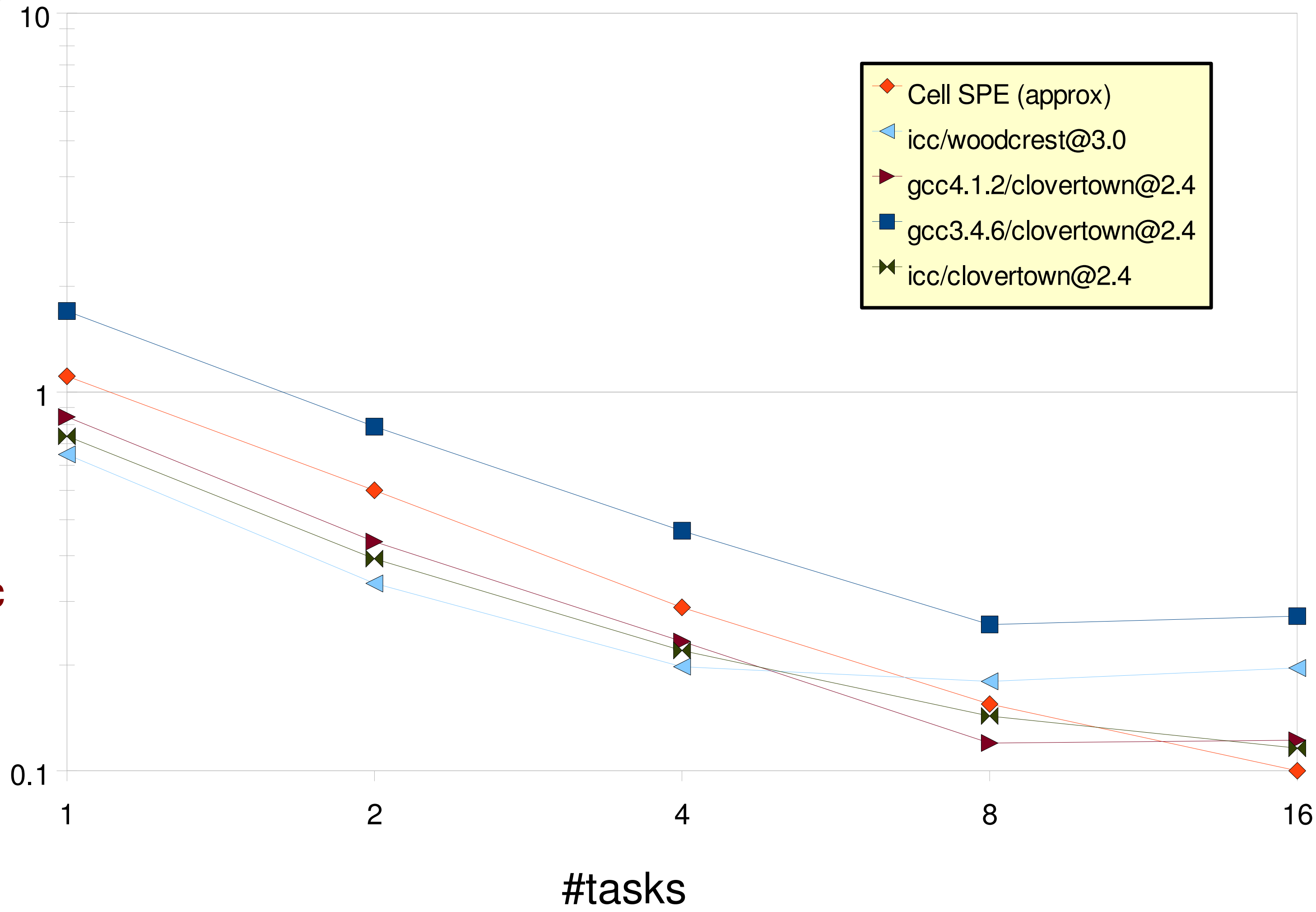

```
for(int i = 0; i < n_tracks / 4; i++){
      Fit(track_vector[i], ...);
}
```

```
parallel_for(blocked_range<int>(
      0, n_tracks / 4, grain_size),
      ApplyFit(track_vectors, ...));
```

Measurements

Real fit time/track (us)

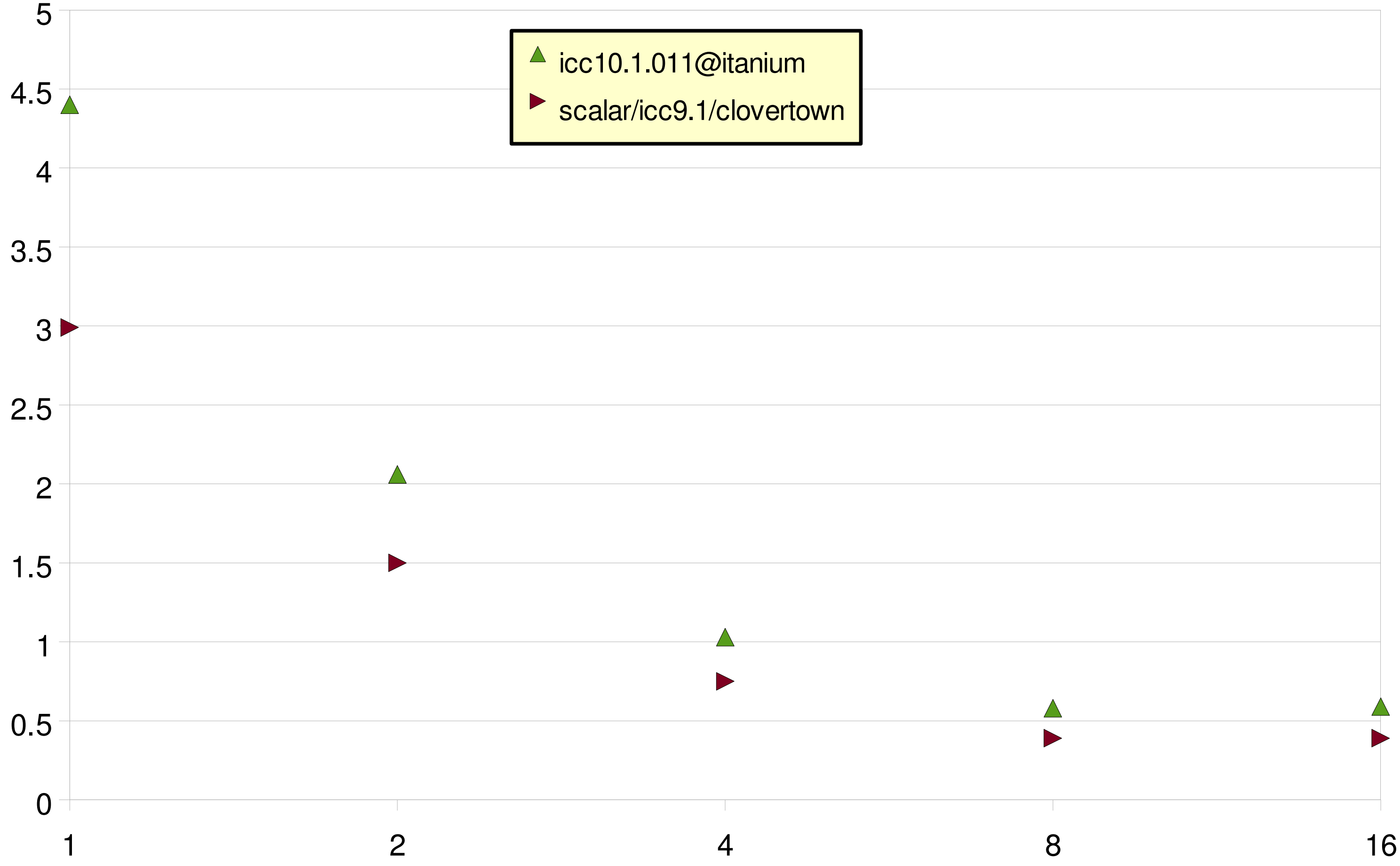
Logarithmic scale!



Observations

- Cell (16-way) and Clovertown w/ 8 cores have highest throughput
- Woodcrest w/ 4 cores has best per-core performance
- GCC 4.1.2 has doubled vectorised code performance of 3.4.6

Measurements - Itanium



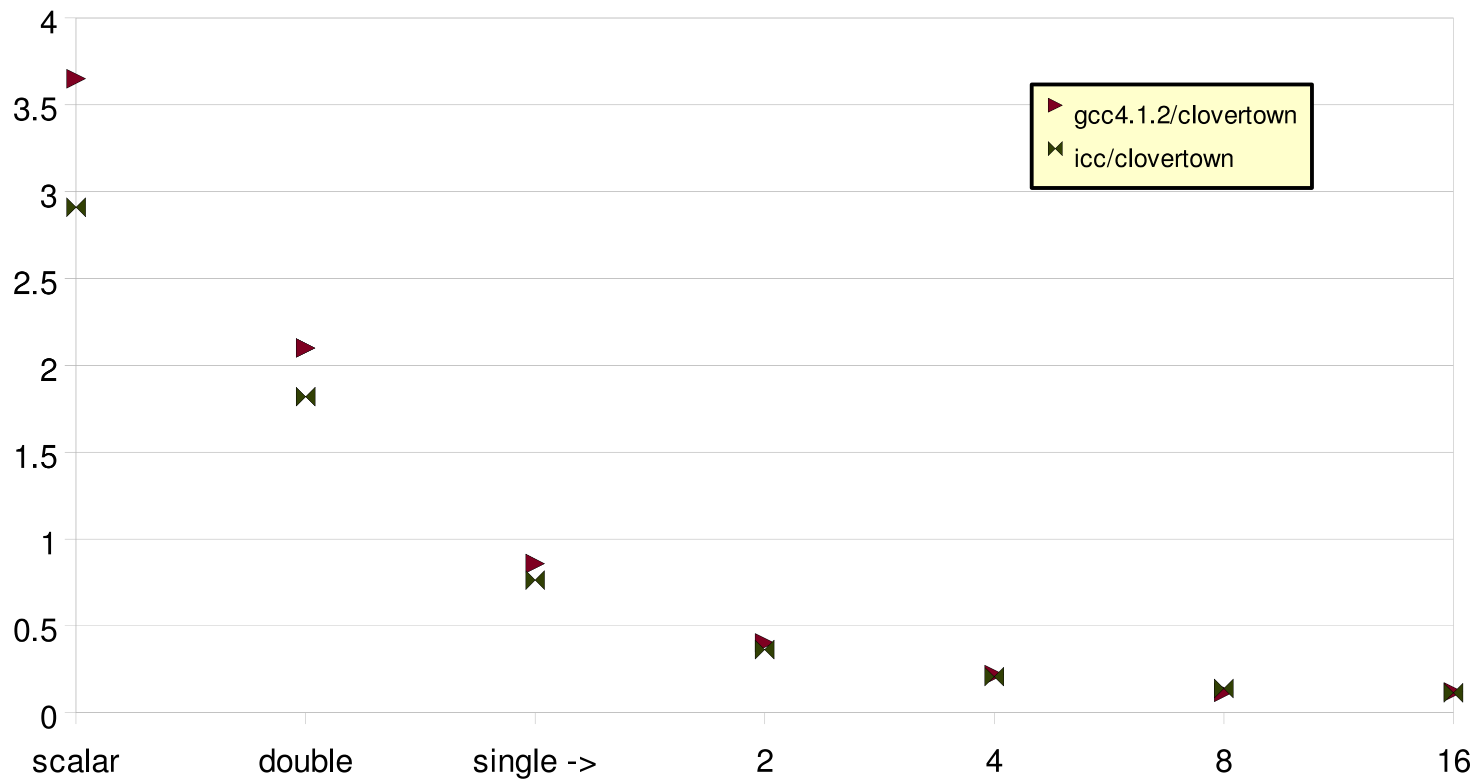
Real fit time/track (us)

#tasks

Total speedup

- Total speedup w/ both optimizations:
  3.7 / 0.12 = 30

Real fit time/track (us)



Conclusion

- Track fitting with the Kalman Filter is embarrassingly parallel and scales well over multiple cores
- A lot of time (= money) can be saved by properly optimizing parallel reconstruction code
  - Example vectorisation speedup from scalar double to packed single: 3.7
  - Example multithreading speedup on 8 cores: 7.2
  - Proportional speedup increase can be expected with future architectures