Benchmarks for the ITS cluster finder

Sylvain Chapeland – ALICE DAQ / O^2

Outline

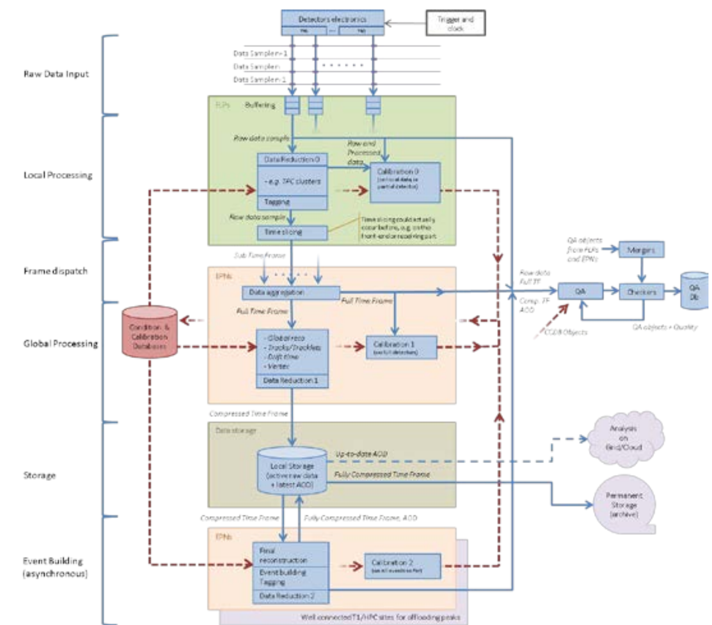
- ALICE upgrade
- ITS cluster finder algorithm
- Implementation and optimization
  - Tools, platforms, benchmarks
- Plans

ALICE upgrade

- O^2 is in design phase

  250 First Level Processors (readout) :
  Sub-event processing, ~5x compression

  1250 Event Processing Nodes:
  Global processing and event building
  (synchronous and asynchronous)
  Raw data discarded



- Benchmark required for estimation of computing resources needs and possible hardware

Benchmarks for ALICE upgrade

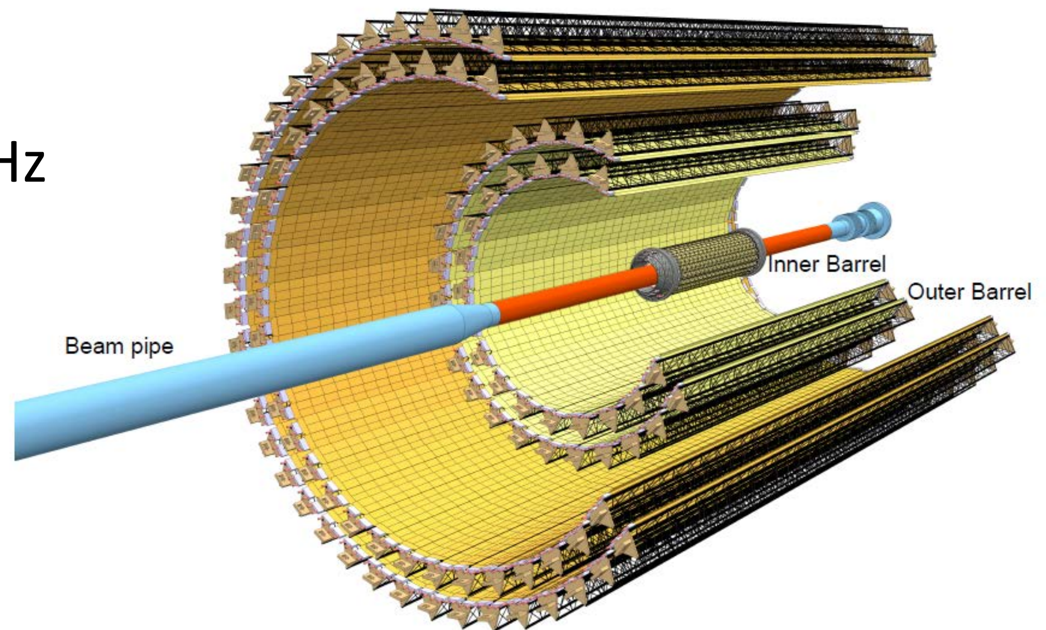
- Several ongoing activities (e.g. in CWG5)
  - raw benchmarks (cpu, memory, etc)
  - Reuse existing code (e.g. extensive experience with TPC tracking on GPUs by ALICE HLT, used in production)

- We present here one case study, with following goals:
  - Find a realistic workload
  - Implement the algorithm with various hw/sw
  - Get performance results
  - Get experience with the tools and platforms

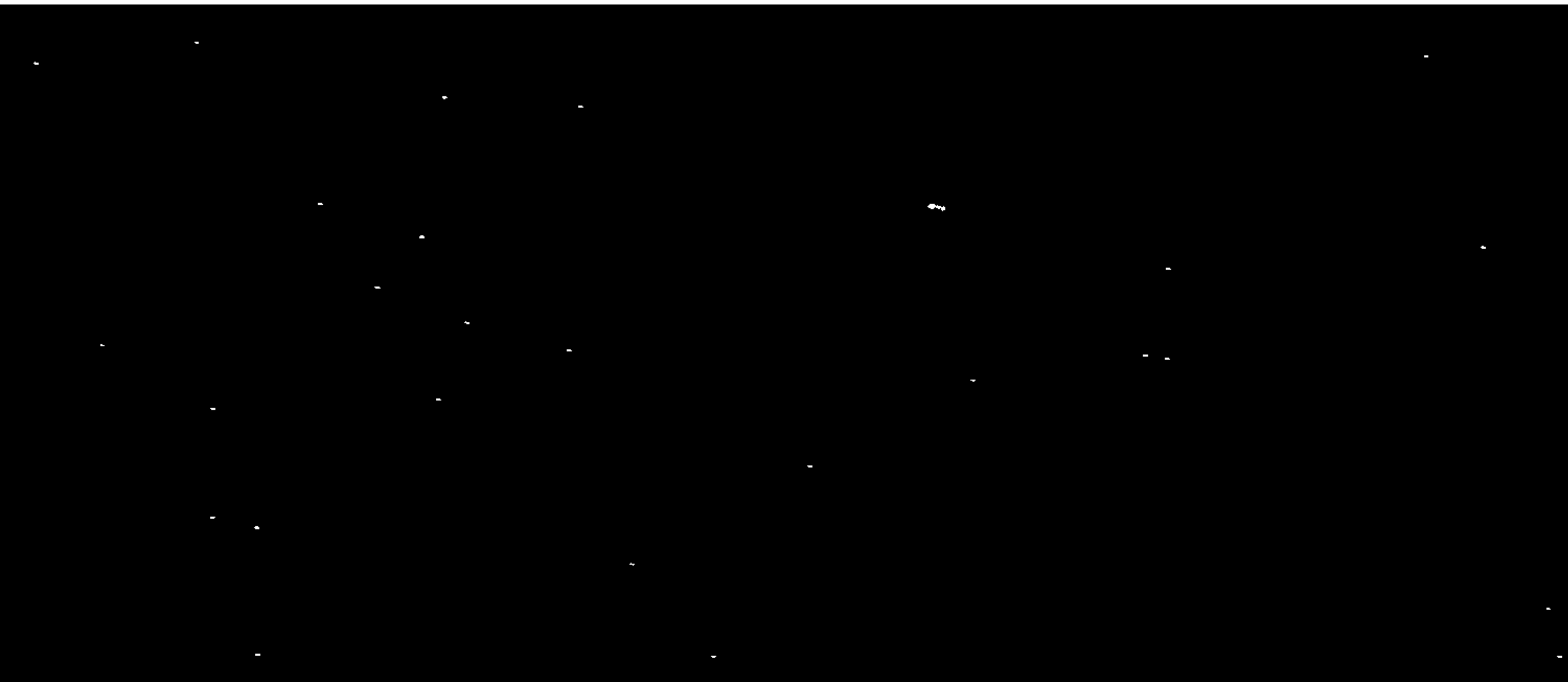
ALICE ITS detector upgrade

- Inner Tracking System, silicon detector
  - Chip size 1650x500 pixels (inner layer)
  - 24120 chips in 7 layers

- Readout 50-400 kHz
- 40GB/s for Pb-Pb@50kHz

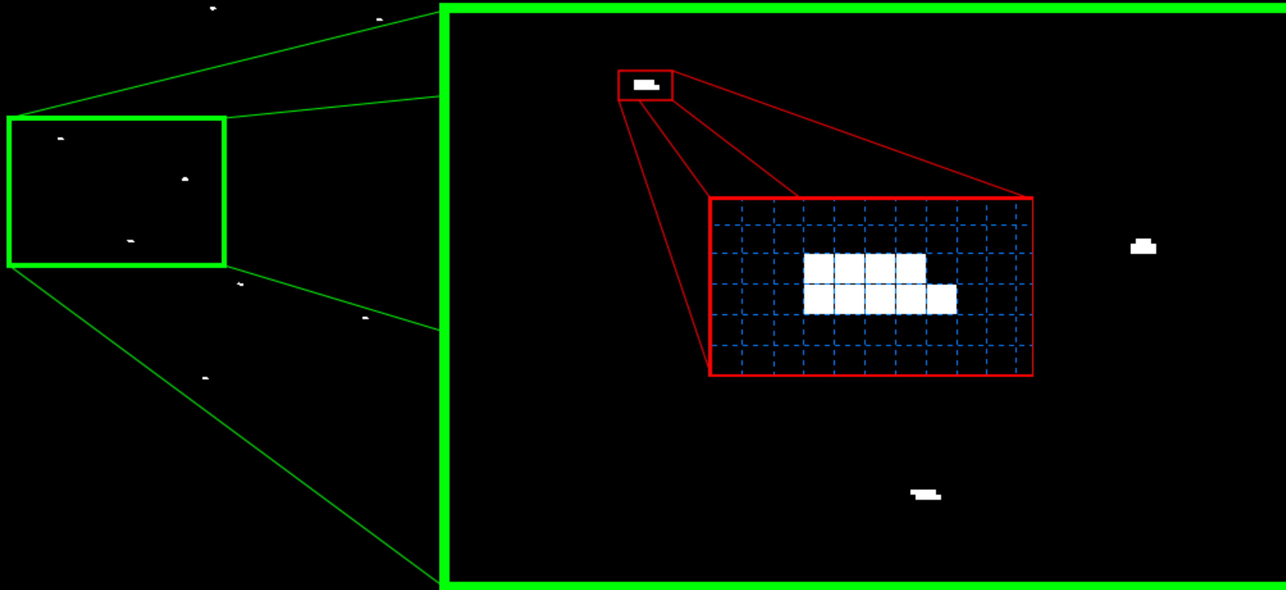


Example simulated event on a chip



Mostly empty

Simulated event (close-up)



Each white pixel is a hit, and a group of adjacent hits is a cluster

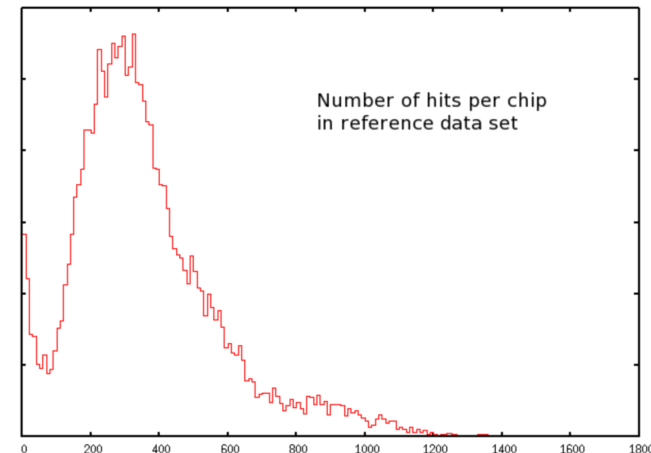
ITS cluster finder algorithm

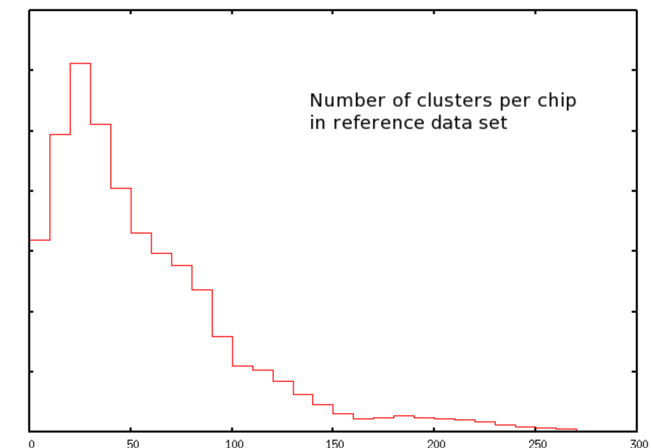
- Simple to understand, no physics involved
  - Identify group of adjacent pixels
  - Compute their center of gravity

- Simple data set
  - Input: list of hits coordinates

    int[], ordered by (row,column) in detectors electronics. In average 360 hits per chip, i.e. ~3kB
  - Output: list of cluster coordinates

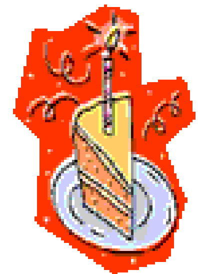
    float[], in millimeters from chip center). In average 60 clusters per chip, i.e. ~0.5 kB





ITS cluster finder algorithm

- Good candidate for benchmark
  - This is a good representative of one demanding type of computation to be done online, similar things done in other detectors
  - Simulated data available for input
  - Reference algorithm already available in the offline framework for output crosscheck
  - Can be easily re-implemented standalone, no external libs required
  - Fine level of parallelism for free (event or chip level)
  - Small data size should fit most architectures



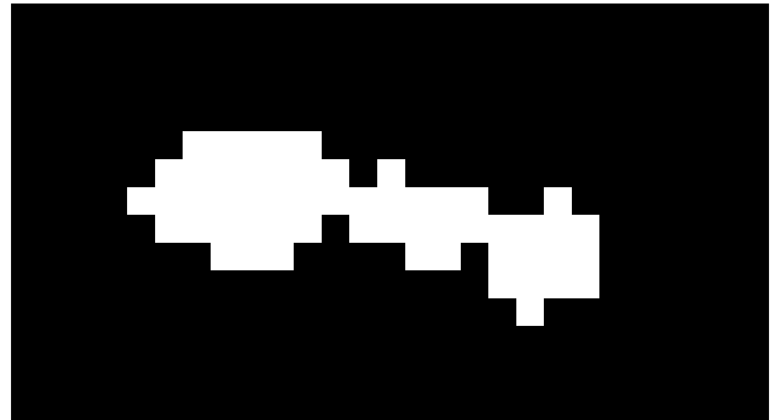
C implementation for x86

- Loop over (ordered) list of hits
- Create and assign cluster ID
  - We keep current and previous pixel line in memory, with id of cluster
  - Neighborhood hits check by array indexing + bitmask (no loop)
- Group clusters
- Compute CoG

- No threading in “processEvent” code, rather 1 thread per event basis or per module
- -O3 flag with gcc 4.4.7 & icc 14.0.2

Development cycle

- Ref data: 50 events 430 modules
- Implement as simple as possible
  - started with plain C
- Verify validity of result
  - Several iterations needed
  - Some nasty use-cases
  - Added PNG debug function
- Optimize 1 thread
- Try multithread (1 thread per event/module)



Profiling

- Valgrind / callgrind
  - Good enough to find hotspots
  - Readily available
  - Heavy execution time
  - Kcachegrind GUI (kdesdk RPM) to check results

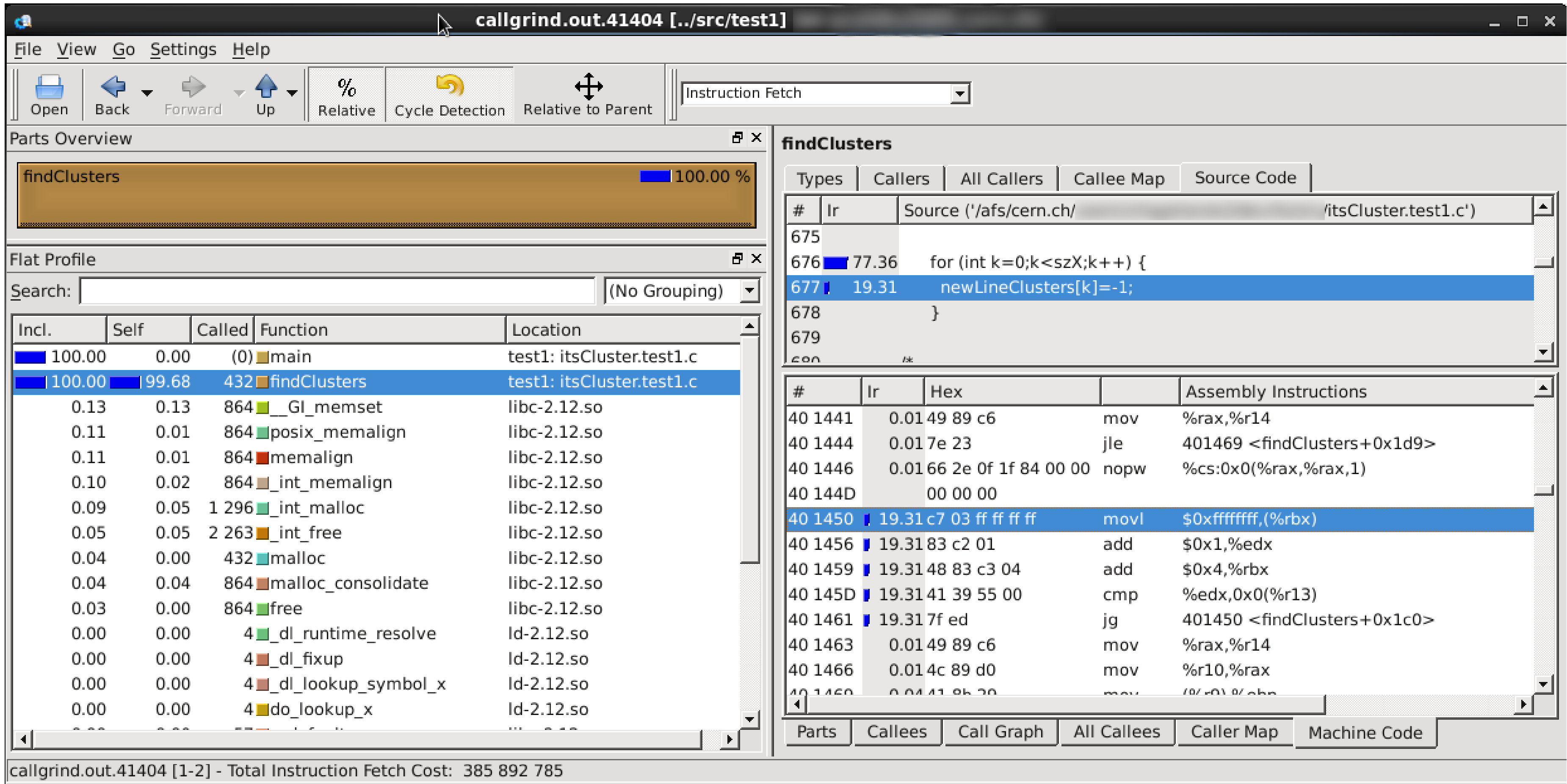
```
#include <valgrind/callgrind.h>
CALLGRIND_START_INSTRUMENTATION
    myCode()
CALLGRIND_STOP_INSTRUMENTATION
```

- Vtune
  - Ampl-xe gui nice, extensive threading support
  - Kernel module easy to recompile
  - Need root access to load module
  - Disable NMI: echo 0 > /proc/sys/kernel/nmi_watchdog
  - Fine details in results, many counters/metrics
  - Need most recent HW for all perf counters

```
#include <ittnotify.h>
__itt_resume()
    myCode()
__itt_pause()
```

- For both: easy to isolate code to be measured
- Missing tool: bookkeeping (run test, keep ref code, document findings and results)

Callgrind / kcachegrind



96.5% in this loop!

Vtune – amplxe-gui

amplxe/projects/test7 - Intel VTune Amplifier (on pcaldbuild05.cern.ch)

Advanced Hotspots — Hotspots viewpoint (change) — Intel VTune Amplifier XE 2013

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames | itsCluster.te...

Source | Assembly | Assembly grouping: Address

```
789
790        int maxk;
791        maxk=maxXnew/4+1;
792        //maxk=szL/4;
793
794        #if (PACKING_MODE==1)
795        maxk=maxXnew/8+1;                          2.667ms
796        #endif
797
798  //      #pragma prefetch p_fastinit:0:256
799        for (int k=0;k<maxk;k+=8) {                15.542ms
800          p_fastinit[k]=v_fastinit;                1.708ms
801          p_fastinit[k+1]=v_fastinit;              15.458ms
802          p_fastinit[k+2]=v_fastinit;              11.667ms
803          p_fastinit[k+3]=v_fastinit;              15.000ms
804          p_fastinit[k+4]=v_fastinit;              14.583ms
805          p_fastinit[k+5]=v_fastinit;              14.167ms
806          p_fastinit[k+6]=v_fastinit;              13.250ms
807          p_fastinit[k+7]=v_fastinit;              13.000ms
808        }
809
810        //printf("maxX=%d, maxk=%d\n",maxXnew,maxk);
811  /*      #pragma unroll(8)
812        for (int k=0;k<maxk;k++) {
813          //p_fastinit[k]=_mm_set_epi32(-1,-1,-1,-1);
814          p_fastinit[k]=v_fastinit;
815        }
816  */
817  /*
818        for (int k=0;k<maxk;k+=8) {
819          _mm_store_si128(&p_fastinit[k], v_fastinit);
820          _mm_store_si128(&p_fastinit[k+1], v_fastinit);
821          _mm_store_si128(&p_fastinit[k+2], v_fastinit);
822          _mm_store_si128(&p_fastinit[k+3], v_fastinit);
823          _mm_store_si128(&p_fastinit[k+4], v_fastinit);
824          _mm_store_si128(&p_fastinit[k+5], v_fastinit);
825          _mm_store_si128(&p_fastinit[k+6], v_fastinit);
826          _mm_store_si128(&p_fastinit[k+7], v_fastinit);
827        }
```

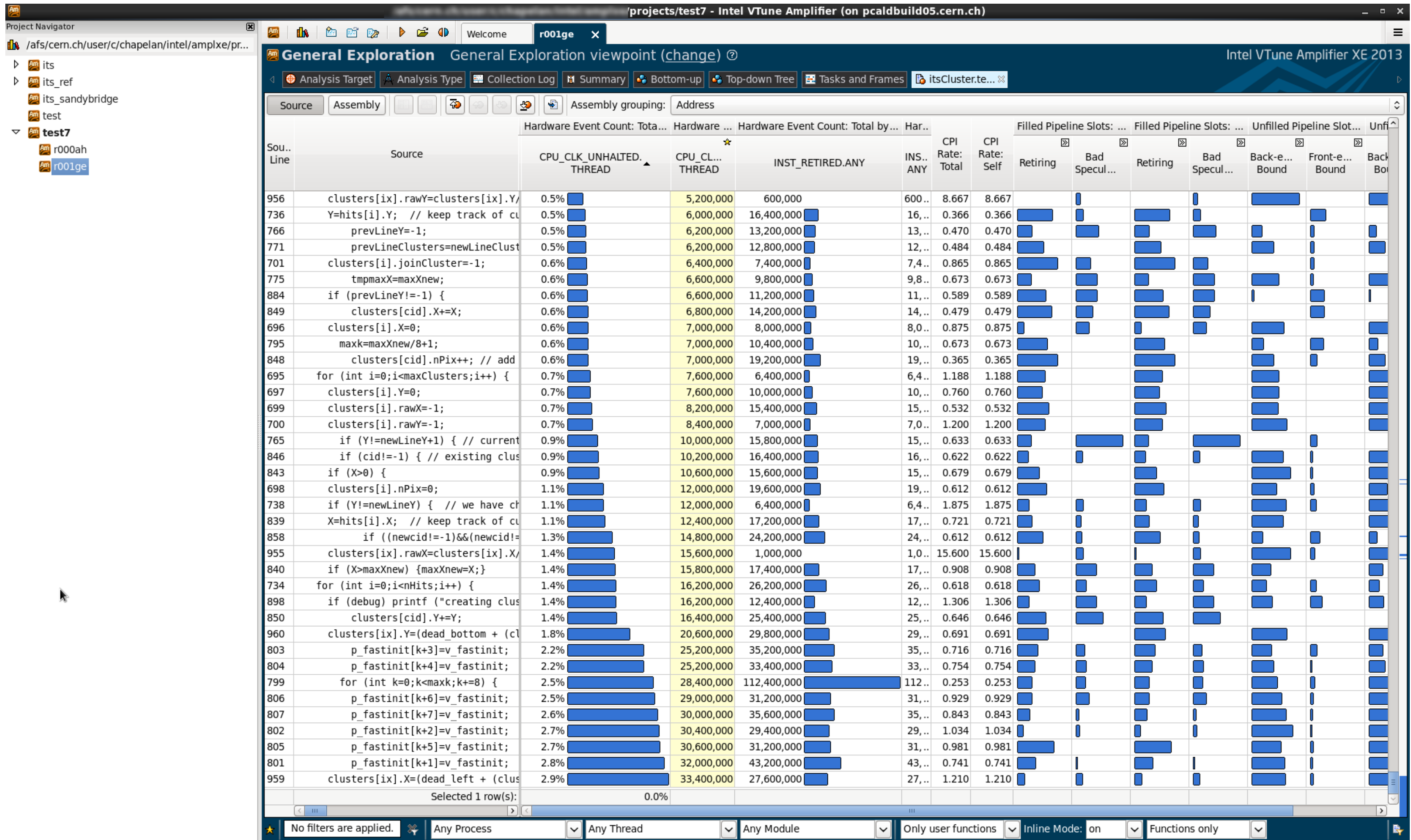
Selected 1 row(s): 1.708ms

```
0x40509a  795  shr $0x1d, %r10d
0x40509e  795  add %r15d, %r10d
0x4050a1  795  sar $0x3, %r10d
0x4050a5  795  movsxd %r10d, %r10
0x4050a8  795  inc %r10
0x4050ab  799  jle 0x4050e8 <Block 27>
0x4050ad       Block 25:
0x4050ad  799  pcmpeqd %xmm0, %xmm0
0x4050b1       Block 26:
0x4050b1  799  add $0x8, %r9
0x4050b5  800  movdqax %xmm0, (%rsi)
0x4050b9  801  movdqax %xmm0, 0x10(%rsi)
0x4050be  802  movdqax %xmm0, 0x20(%rsi)
0x4050c3  803  movdqax %xmm0, 0x30(%rsi)
0x4050c8  804  movdqax %xmm0, 0x40(%rsi)
0x4050cd  805  movdqax %xmm0, 0x50(%rsi)
0x4050d2  806  movdqax %xmm0, 0x60(%rsi)
0x4050d7  807  movdqax %xmm0, 0x70(%rsi)
0x4050dc  799  add $0x80, %rsi
0x4050e3  799  cmp %r10, %r9
0x4050e6  799  jl 0x4050b1 <Block 26>
0x4050e8       Block 27:
0x4050e8  833  xor %r15d, %r15d
0x4050eb  835  mov %r8d, %r14d
0x4050ee       Block 28:
0x4050ee  839  movl (%r13), %r9d
0x4050f2  840  cmp %r15d, %r9d
0x4050f5  840  cmovnl %r9d, %r15d
0x4050f9  843  test %r9d, %r9d
0x4050fc  843  jle 0x4053dc <Block 49>
0x405102       Block 29:
0x405102  845  lea -0x1(%r9), %eax
0x405106  845  mov %eax, %r10d
0x405109  845  sar $0x1, %r10d
0x40510c  845  movsxd %r10d, %r10
0x40510f  845  movl (%rbx,%r10,4), %r10d
0x405113  845  mov %r10d, %r11d
0x405116  845  sar $0x10, %r11d
0x40511a  845  test $0x1, %al
```

Highlighted 1 row(s)

No filters are applied. | Any Process | Any Thread | Any Module | Any Utilization | Only user functions | Inline Mode: on | Functions only

Vtune – amplxe-gui



Code optimization (1 thread)

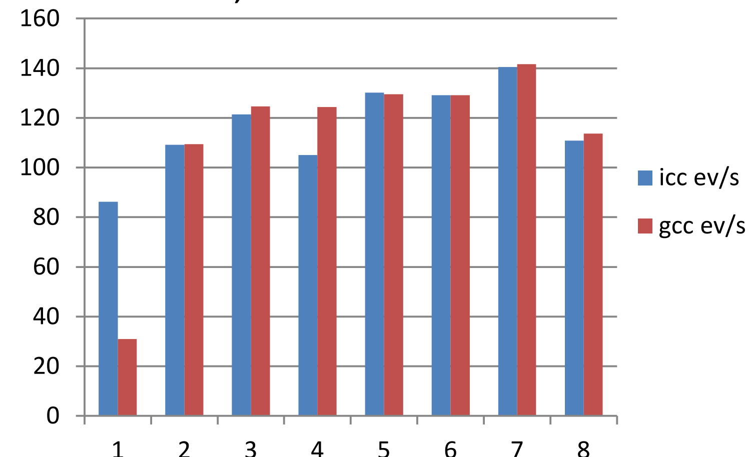
- Hotspot analysis shows that most time is spent in clearing index line

  (id of cluster for each pixel in current line)

```
for (int k=0;k<szX;k++) {
  newLineClusters[k]=-1;
}
```

- Optimization
  1. Dummy loop
  2. + Don't reset full line (keep max index updated)
  3. Use memset /* works only because (int)-1 = char[4] {-1,-1,-1,-1} */
  4. Use 128bits intrinsics => movdqa
  5. + Manual loop unroll (8)
  6. + Pack data manually (1 coord per int)
  7. + Pack data manually (2 coords per int)
  8. + Pack data manually (3 coords per int)

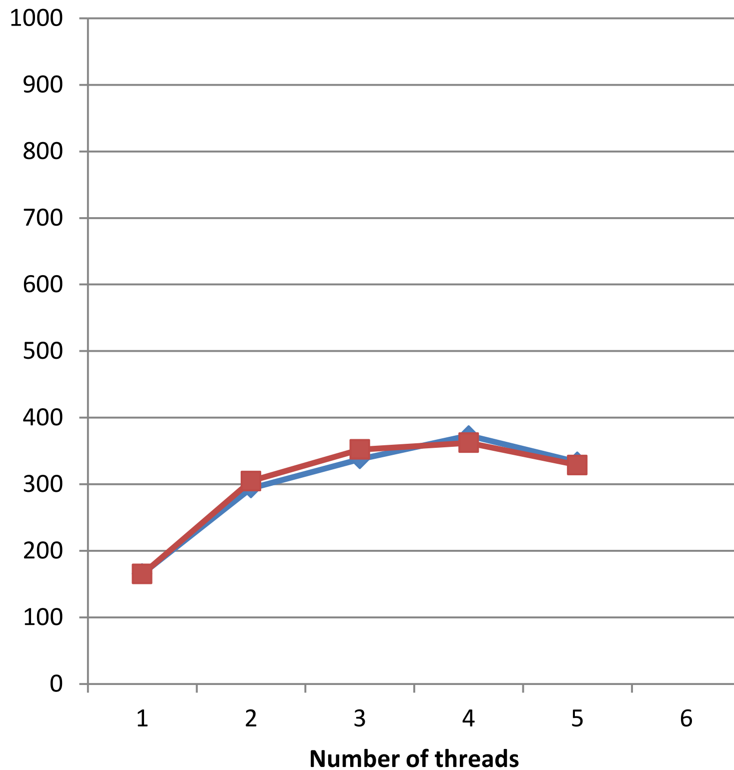
```
#include <immintrin.h>
__m128i v_fastinit;
v_fastinit=_mm_set_epi32(-1,-1,-1,-1);
for (int k=0;k<maxX/4+1;k++) {
  ((__m128i *)(newLineClusters))[k]=v_fastinit;
}
```

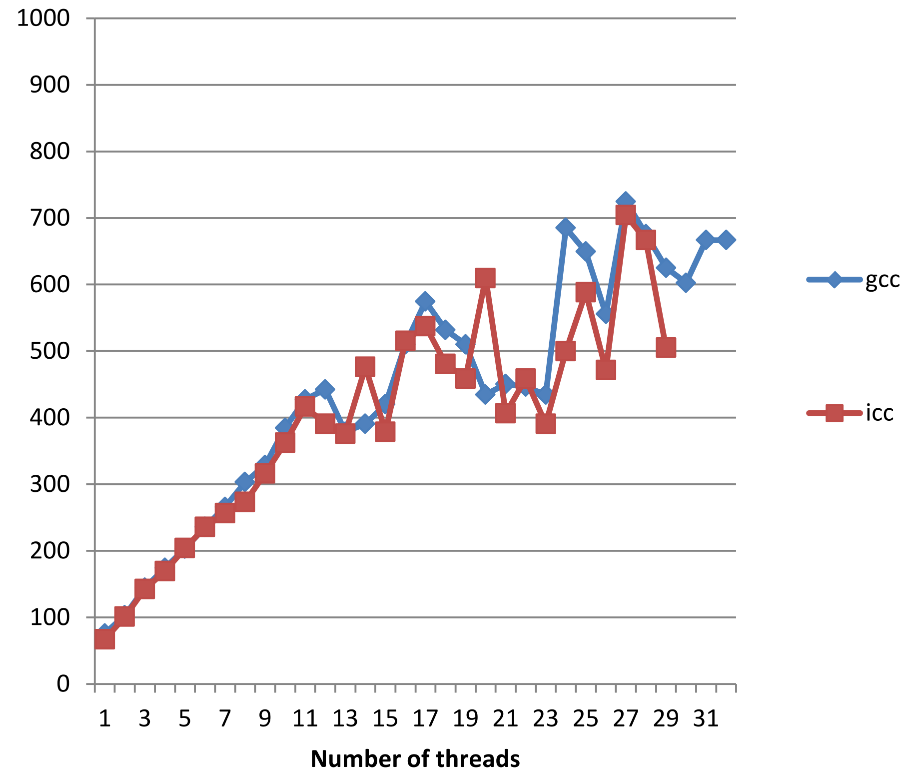


Code optimization (1 thread)

- Observed so far:
  - Pack data = improve performance, because algorithm is memory I/O bound
  - Importance of data structs (e.g. aligned to use vector instructions)
  - Assembly useful (e.g. to check the good AVX instructions are used in the end)
  - Some obvious and simple things might still be worth optimizing “by hand”
- Among ideas tried:
  - Automatic loop unroll with icc pragma
  - Pack 3 coords in one int – but loose time on division
  - Use short instead of int does not help (data not packed automatically)
  - Walk back array faster than resetting indices
  - Excellent performance and ease of implementation with c++11 vector class
  - Use one array for cluster index + 1 bitmask for upper row neighbor check is the best solution so far
  - Optimization of CoG computation helpful (another 10-15%)
- Demanding process…
  - Algorithm was looking like a piece of cake, but reality is different
  - Requires effort and iterations
  - Quite addictive

Scalability (04/2014)



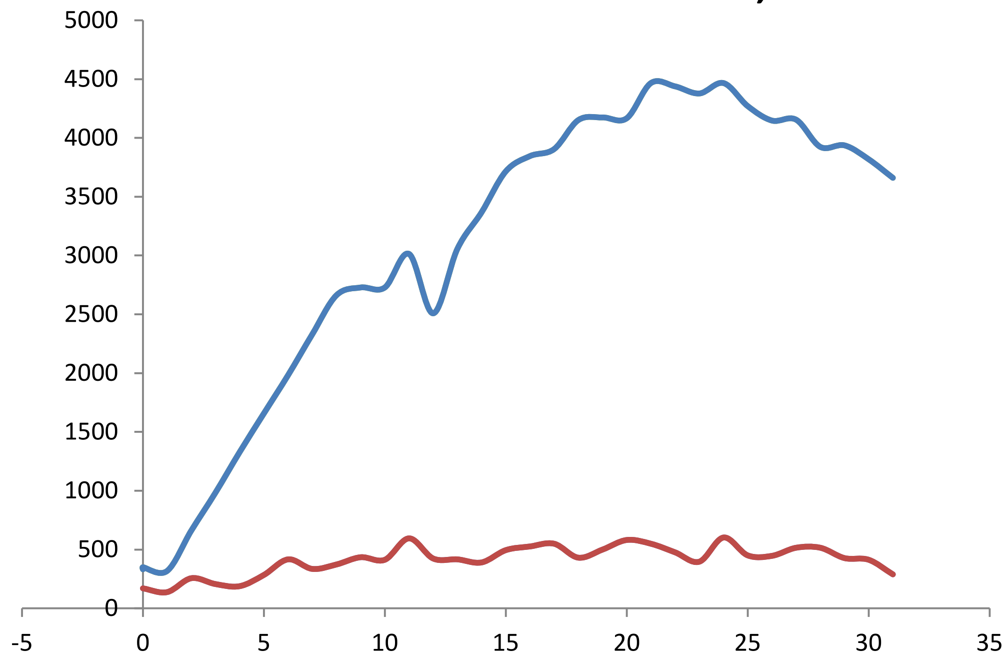


1 thread per event, 50 events

Scalability (observations 04/2014)

- Observations
  - Suspect bottleneck in memory access, high back-end usage
  - Was not expecting good results because of lightweight computing task (rather memory bound)
  - Effects on turbo mode affects the measurements with low thread count (and the linear baseline for scalability check)
  - Fluctuations with high thread count because of small data set and variance in processing time per event
- Algorithm scales decently
  - Suspect locks in memory allocation from 1-thread hotspot analysis
  - Overhead of threading / workload distribution
  - Need to deal with streaming data in NUMA node (data/CPU affinity)

Scalability (06/2014)



Optimized version: c++11 vectors, NO numa tuning
neighbor search: dual-row indexing + bitmask

Initial version: c,
neighbor search: dual-row indexing , 2 pixels per int32

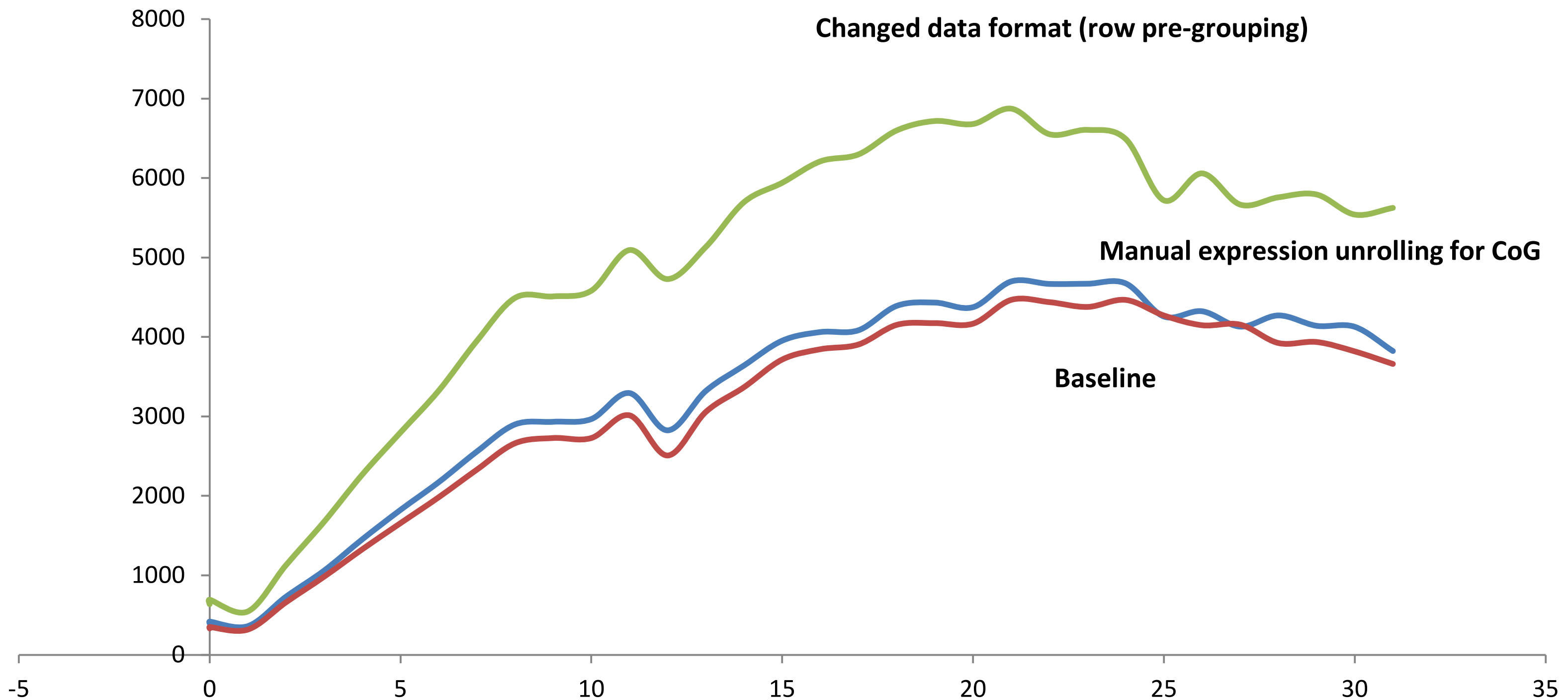
1 thread per event (or chip?)

What changed (04->06 / 2014)

- Bitmask for neighbor check
  - i.e. 24*64 bit in addition to 1500 x 32 bit index
- C++ 11
  - Std::vector
  - Threading

- Still to do:
  - NUMA affinity handling (standalone experiments with libnuma are promising)
  - Redo profiling on ivy bridge (compared to sandybridge, gives finer details on back-end, higher level metrics and hints for memory bottlenecks)
  - Plot processing time per chip versus number of hits on chip (probably linear)

Latest performance (yesterday)



Proposal on data format

- Now: hit1 (X,Y) hit2 (X,Y) … [order by row,col]
- What about grouping consecutive pixels in a row?
- Proposed: hitgroup1 (X1,X2,Y) hitgroup (X1,X2,Y) … [still ordered by row, col]
  - Should be easy implement in FEE

- On reference data set, this represents a 2.5x compression, which in turns also improves throughput in algorithm

inaccurate & provocative **cost** observations

CPU	CPU Price $	CPU mark	Perf event/s	Price/perf $/(event/sec)
I5-680 2c@3.6 GHz (desktop Q2 2010)	320	3539	373	0.86
E5-2665 16c@2.4 GHz (server Q2 2012)	1420	12452	724	1.96

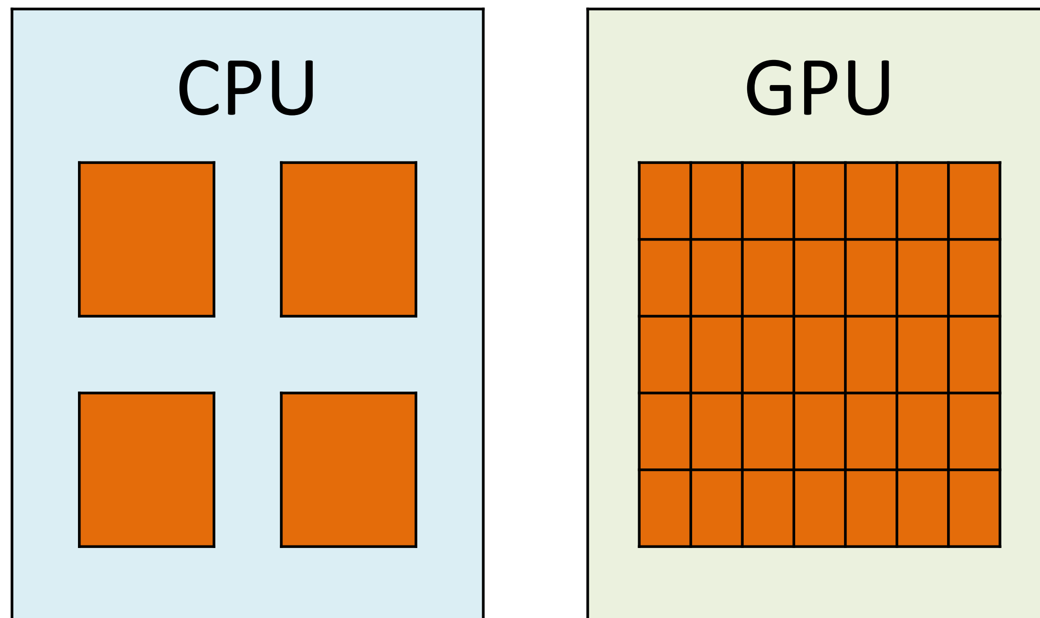
- Higher clock speed is still one easy way to get perf
  - to be considered for specific low-latency needs ?
- For some workloads, a desktop might still be a good contender and more than twice cheaper at same perf level

X86 summary

- We can now process 430 inner chips @ 400Hz for 1 thread / 5KHz for 1 machine (6 GB/s with 32bit coords)
- How does this extrapolate to full detector? (need full data set) 10-50 times more?
- We can still progress on local scalability
  - NUMA
  - Threading framework
  - Data format
  - Wider vectors
- 10KHz/machine (for 400 chips) looks reasonably reachable, probably more depending on data format
- Still far from 50KHz for 20000 chips… would need 50-250 nodes just for cluster finding?
- Algorithm mature enough to clarify estimate, but depends much on realistic data format. We should focus on this now.

ITS Cluster Finding on GPU

- Work from Boonyarit Changaival @ KMUTT
- Exploit multicore architecture
- Use one thread to process one hit (in most kernel)



GPU

- Preliminary results
  - i5-3470 @ 3.2GHz
  - Nvidia Geforce GTX780

- 30 events/second (same ref. data as for x86, i.e. 430 modules)
  - 1 CPU thread
  - Show full occupancy over all kernels (>90%)
  - Very low serialization portion (in most kernels)

GPU Runtime Overview

- Spend 71% of total runtime on GPU
- Overhead is 29% of the total runtime
  - Initialization
  - Host and device memory allocation
  - Moving data between host and device
- Why slower than x86?
  - Copying memory between Host and Device
  - Loops in GPU decrease performance
  - Not many rich instructions
- My personal analysis: this algorithm does not fit well GPU, too little floating point math & vector, too much I/O

- Still some ideas to try: multiple GPU cards, multiple CPU threads
- We will still work a bit on this in coming weeks, but unlikely that we gain 2 orders of magnitude…

Xeon Phi

- Initial tests promising for code portability
  - Ok to compile, doc not so easy to start with e.g. TBB
  - Simple programming paradigm to adapt the code: C++11 / TBB / parallel_for

```
#pragma offload_attribute (push,target(mic))
#include "tbb/tbb.h"
#pragma offload_attribute(pop)

__attribute__((target(mic))) void ComputeClusters(dataIn dIn, dataOut *dOut) {…}

void ParallelApplyComputeClusters( dataIn* in, dataOut *out, size_t n ) {
   parallel_for( blocked_range<size_t>(0,n),
      [=](const blocked_range<size_t>& r) {
                  for(size_t i=r.begin(); i!=r.end(); ++i)
                     ComputeClusters(in[i],&out[i]);
               }
    );
}
```

- Work paused, need to clarify scalability first
- Then tune code with hw specific features (e.g. wider vectors)
- However it might be that we need next generation, with out of order execution able to handle the many branching we have in cluster finding code. I/O bandwidth also to be tested.
- HW made available by CERN Techlab

What's next

- Get full detector data set
- Continue platform implementation
  - x86
    - Vector/bitmask
    - Haswell (new 256bit int instructions, BSR & gather/scatter)
    - NUMA affinity / memory pre-allocate / data streaming
      - i.e. "framework", not algorithm
  - MIC – 512bit instructions, 70 "slow" cores
  - GPU – any good idea?
  - FPGA (many implementations, c.f. RT14) ?

Conclusion

One need a real use case to progress!

- Find a realistic workload
- Implement the algorithm with various hw/sw
- Get performance results
- Get experience with the tools and platforms

- *Exercise was fruitful in all aspects*