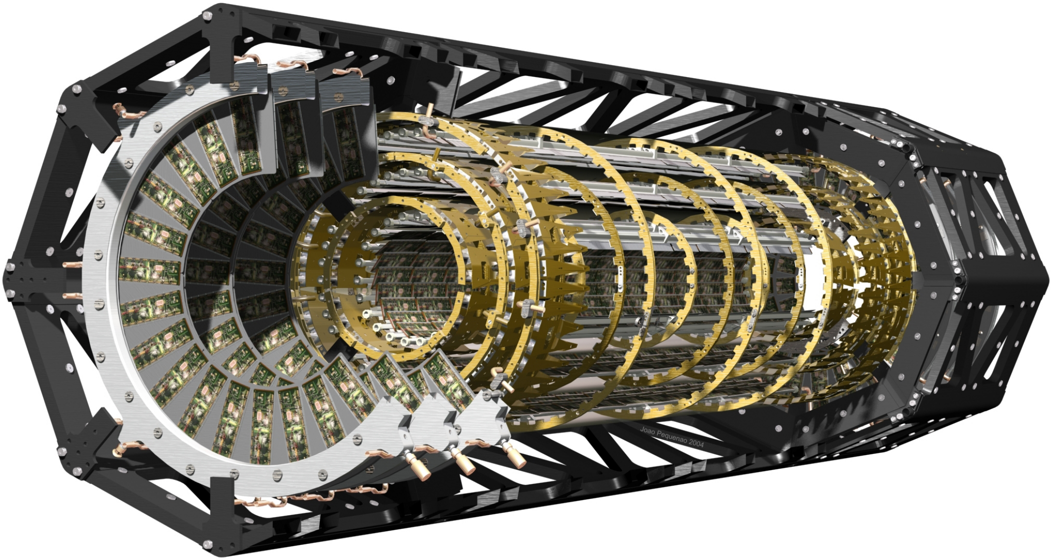
Performance Case Study: Charge Clusterization

Louis-Guillaume Gagnon (UC Berkeley)

CoDaS-HEP 2024
2024/07/24

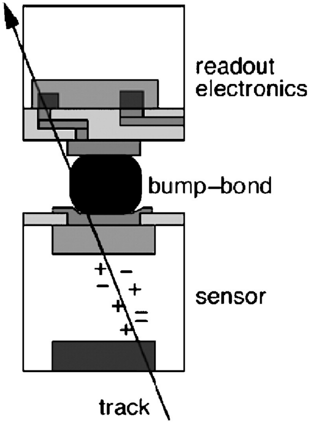
Introduction: What is charge clusterization?

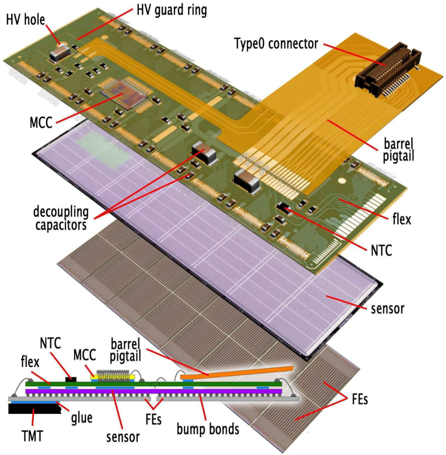


Introduction: What is charge clusterization?

- Charged particle ionizes Si sensor
- Charge detected via bump bond to readout

- Si sensor *not* segmented
- 2D matrix defined by bump bonds

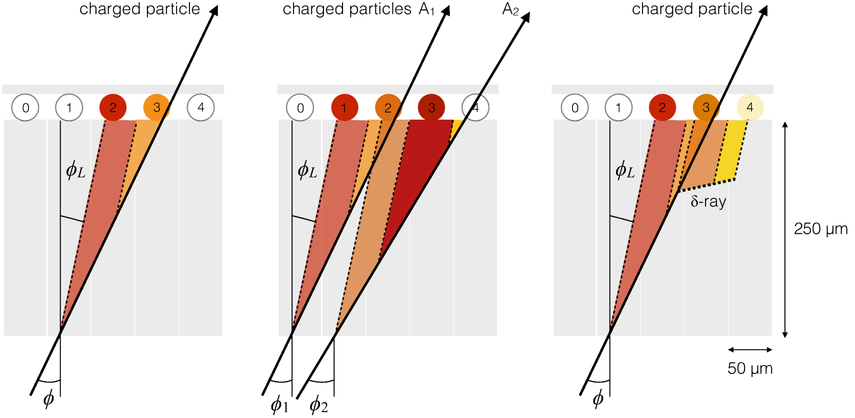




source

ATLAS ID pixel chip. source

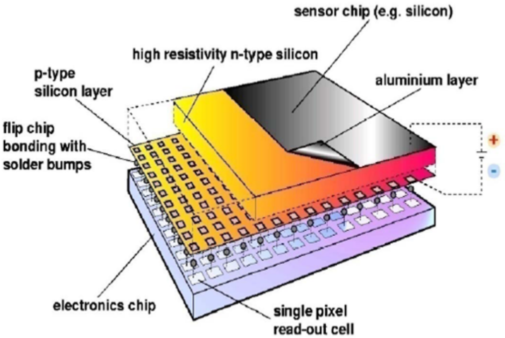
Introduction: What is charge clusterization?

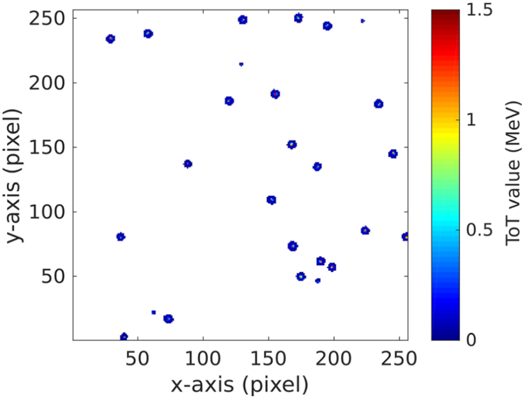


(a) (b) (c)

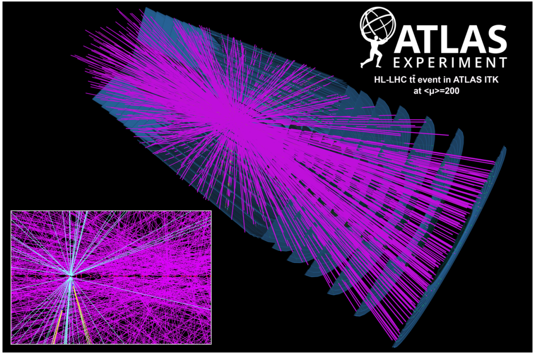
- Charge can be deposited in > 1 pixel: Incident angle, drift in B field, cluster merging, δ-rays, . . .
- Pixel chip will typically readout *individual* pixels
- Clusterization: Forming charge clusters out of individual pixels (& estimate crossing position)

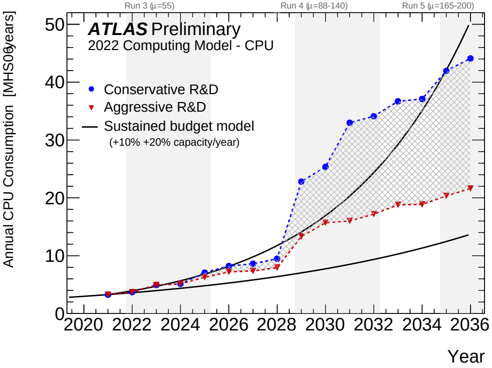
- Example: Timepix detector module
- Note that module is *sparsely activated*





Introduction: Why do we need to optimize this?





- Run 4: Circa 2027, first run with HL-LHC
- Luminosity increase: very challenging for track reconstruction!

- Luminosity increase strains CPU budget
- Tracking is a large contribution: Needs R&D
- Must speedup *every* part of the tracking chain!

Introduction: ACTS

- ACTS: experiment-independent toolkit for track reconstruction
- Emphasis on **long-term maintainable code** and **optimized computing and physics performance**
- Funded by IRIS-HEP!
- ACTS used in published physics results: **ATLAS**, **FASER**
- ACTS integration in progress: ALICE, CEPC, ePIC, LDMX, Lohengrin, NA60+, sPhenix, STFC, . . .
- ATLAS in process of migrating tracking code to ACTS
- More information:
  - Overview paper: [2106.13593]
  - Project webpage: acts.readthedocs.io
  - Code repository: github.com/acts-project/acts

The ACTS clusterisation algorithm, pre-2022

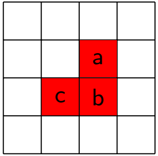
Algorithm 1 createClusters

Input: *pixels*, unordered vector of activated pixels

$map \leftarrow hashMap(pixels)$ // $index \rightarrow pixel$
for all *pixel* in *map* **do**
  if not $pixel.used()$ **then**
    $fillCluster(\{pixel\}, pixel, map)$
  end if
end for

Algorithm 2 fillCluster

for *i* in $neighbourIndices(pixel)$ **do**
  if $pixel' \leftarrow map.find(i)$ & not $pixel'.used()$ **then**
    $cluster \leftarrow cluster + \{pixel'\}$
    $fillCluster(cluster, pixel', map)$
  end if
end for



- $fillCluster(\{a\}$, a, $map)$
  - $fillCluster(\{a, b\}$, b, $map)$
    - $fillCluster(\{a, b, c\}$, c, $map)$
    - ...
  - ...
- ...
- $\implies \{a, b, c\}$

The ACTS clusterisation algorithm, pre-2022

This algorithm has many desirable characteristics! E.g.

- Uses efficient hash map datastructure
  - Creation is $\mathcal{O}(n)$
  - Lookups are $\mathcal{O}(1)$
- Elegant implementation based on recursive algorithm
- Single map traversal that yields all clusters
- Unordered traversal: no need to sort the input

The ATLAS clusterization algorithm

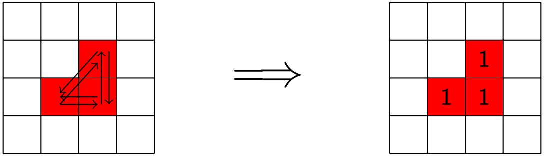
Algorithm 3 createConnectionsGraph

Input: *pixels*, unordered vector of activated pixels

```
pixels ← sorted(pixels) // sort by col., then row
graph ← emptyGraph()
for all pixel in pixels do
    for all pixel′ in pixels.forwardOf(pixel) do
        graph.connect(pixel, pixel′)
    end for
end for
```

Algorithm 4 createClusters

```
label ← 1
for vertex in graph do
    if not vertex.labeled() then
        labelAllConnected(vertex, label)
    end if
    label ← label + 1
end for
clusters ← createClusters(pixels) // ...
```

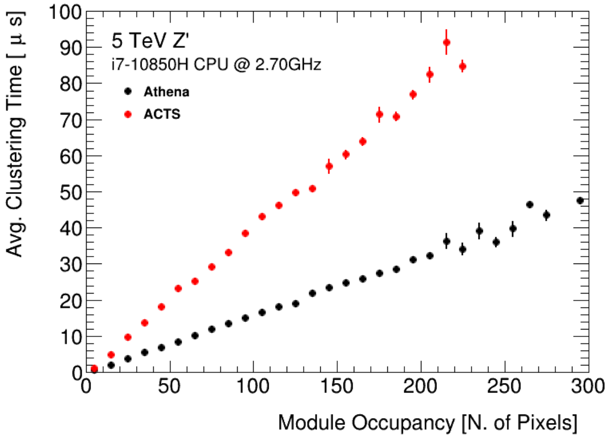


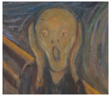
The ATLAS clusterisation algorithm, pre-Run 4

This algorithm has many question marks E.g.

- Uses a graph datastructure: creation is non-trivial
- Algorithm relies on ordered traversal to create graph: needs sorting
- Algorithm now mix of non-trivial imperative loop & recursion
- Two passes needed to create clusters: Record connections, then walk the graph

Which is faster?


5 TeV Z'
i7-10850H CPU @ 2.70GHz
Athena
ACTS
Avg. Clustering Time [μ s]
Module Occupancy [N. of Pixels]
0
10
20
30
40
50
60
70
80
90
100
0
50
100
150
200
250
300



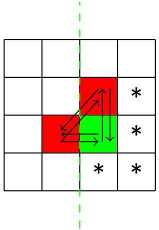
Why?

- Naively, I first thought it would be the other way around! (c.f. my notes at the time)
- Why? Two main reasons I can think of:

1. The single-pass strategy is suboptimal
2. Input data is sparse but algorithm not taking full advantage

1. The single-pass strategy is suboptimal

- Counter-intuitively, it can be faster to solve an intermediate problem before solving the main one!
- In this case, single-pass algo is unable to create partial clusters and reconcilse later!
- Time is wasted checking *every* neighbors (which ensure creation of whole clusters)
- Better algorithm: Record connections first, *then* create clusters
  $\Longrightarrow$ Only need to check for connections on one side of pixel: Less work!



Key insight: Pick the right algorithm!

Input data is **sparse** but algorithm not taking full advantage

- Remember: on average, pixel detector modules are sparsely activated
- With sparse data, optimal data representation is usually different from dense case!
- Note that ACTS *is* using a sparse representation: an index map!
  - But it queried every neighbor indices, as if the module was densely activated!
- Better representation: simple vector of activated pixels, sorted by position

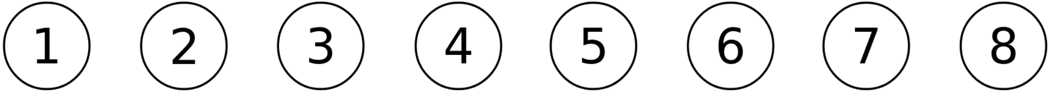
- In a nutshell, with sorted list you can ask:
  "Give me the closest cell, I will check if it's a neighbor"
- For sparse data, it is better than asking:
  "give me the neighboring cell in DIRECTION if it exists"

Key insight: Know your data!

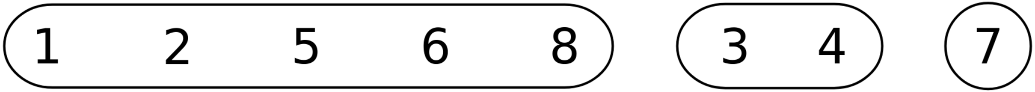
Can we do better?

- I will now make a bold claim: The Athena algorithm solves the **wrong** problem!
- Do we *really* care about the exact way the pixels are connected?
- What if, instead, the algorithm would:

1. Assign a label to each pixel, e.g.



2. Keep track of relationships *between those labels*



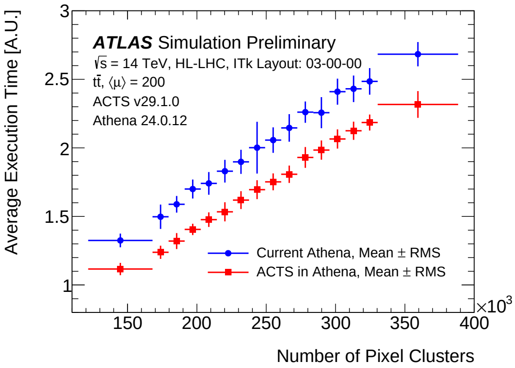
$\Longrightarrow$ Can use "Union Find", a.k.a "Disjoit Set Forest" datastructure

The *new* ACTS algorithm

The Hoshen-Kopelman algorithm

- For each active pix, search backward neighbors
- If there are none, allocate a new cluster label
- If there are connections:
  1. re-use one of the label
  2. mark all connected labels as equivalent

- Second pass: "Merge" labels based on result
- **These operations are efficiently supported by the disjoint set forest!**



Key insight: Pick the right datastructure!

Future Outlook: Charge Clusterization on GPU?

- Promising results from traccc project
- Implementation of a similar algorithm `FastSV`
- Table: Scaling vs N. of Si sensor modules to process

Scale	1	2	4	8	16	32	64
N	2500	5000	10 000	20 000	40 000	80 000	160 000
CPU time (ms)	44.9	84.4	170.9	340.6	691.8	1353.5	2755.0
GPU time (ms)	3.8	8.0	14.8	29.8	54.9	109.9	221.3
CPU to base	1.00	1.88	3.81	7.59	15.40	30.10	61.40
GPU to base	1.00	2.11	3.89	7.84	14.40	28.90	58.20
GPU speedup (est.)	1.48	1.32	1.44	1.43	1.57	1.54	1.55

credit: Stephen Swatman, ACTS Workshop 2023

Conclusion

$\Longrightarrow$ **Know your data**

$\Longrightarrow$ **Pick the right algorithms**

$\Longrightarrow$ **Pick the right datastructures**

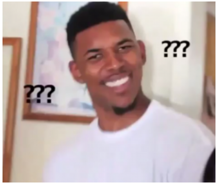
$\Longrightarrow$ **Always benchmark**

It’s time for . . .

Things you didn’t know you needed

Benchmarking!?

- My laptop: 6-core i7-10850H CPU @ 2.70 GHz with hyperthreading
- Turned off turbo boost for reproducible results
  - `echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo`
- Set performance mode for CPU governor
  - `cpupower frequency-set -g performance`
- Pick a core, disable hyperthreading for it
  - `cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list`
  - `echo 0 > /sys/devices/system/cpu/cpu6/online`
- Run jobs with minimum niceness to avoid yielding to other thread
  - `nice -20 <command>`
- Run jobs with maximum CPU affinity to avoid context switches
  - `taskset -c 0 <command>`
- Monitor temperature sensors
  - `acpi -t`
- Close potential resource-hungry programs, do nothing else while job is running
- Check timing distributions for outliers
- Verify that results hold over multiple runs



Benchmarking!?

- Very easy to get this wrong. . .
- Check out: LIKWID
  - Probe the hardware topology of your device
  - Microbenchmark suite to characterize your device
  - Enforce thread/core affinity of a program
  - Control CPU-level settings, e.g. frequencies, hyperthreading, . . .
  - Measure performance metrics (Can use other programs like `perf` as backend!)
  - Helpers for benchmarking openMP/MPI applications
  - Helpers for making performance plots
  - Extensive documentations
  - . . . and more!