flashjet

GPU-native jet clustering with the full merge history

Chirayu Gupta[1,*] Sitian Qian[2] Alexandre De Moor[1]

[1]Vrije Universiteit Brussel

[2]Northwestern University

ML4Jets 2026 · Vienna · September 2026

Outline

Jet clustering, and flashjet

flashjet : a GPU jet clusterer

What it does

- generalised-k_t : anti-k_t, k_t, C/A
- **many jets or events per launch**
- returns jets, the **full merge history**, and substructure **as tensors**
- written in **Triton**: autotunes for whatever GPU it finds — no separate CUDA build.

$\Rightarrow$ built for **reclustering inside training loops**, on padded $(B, N, 4)$ tensors that **stay on the GPU**

Why it can be done at all

- Cacciari–Salam lemma: the smallest d_{ij} in the whole event is always some particle's **geometric** nearest neighbour
- so each particle tracks only **its own** NN, and one merge invalidates $O(1)$ of them

$\Rightarrow$ $O(N^2)$**, not** $O(N^3)$ making each step cheap and parallel

In the ML ecosystem

- Just a **PyTorch** call
- no host $\leftrightarrow$ device copy in a training loop

What everything in this talk is run on

Toys: where the answer is *known analytically*, so landing off the curve is unambiguously a bug.

Input A : toy fat jets

QCD-like and W-like fat jets, $p_T \approx 500\,\text{GeV}$. A known 2-prong vs 1-prong truth.

Input B : toy parton shower

Primary, fixed-coupling leading-log shower, uniform in the Lund triangle

Input C : ATLAS Open Data

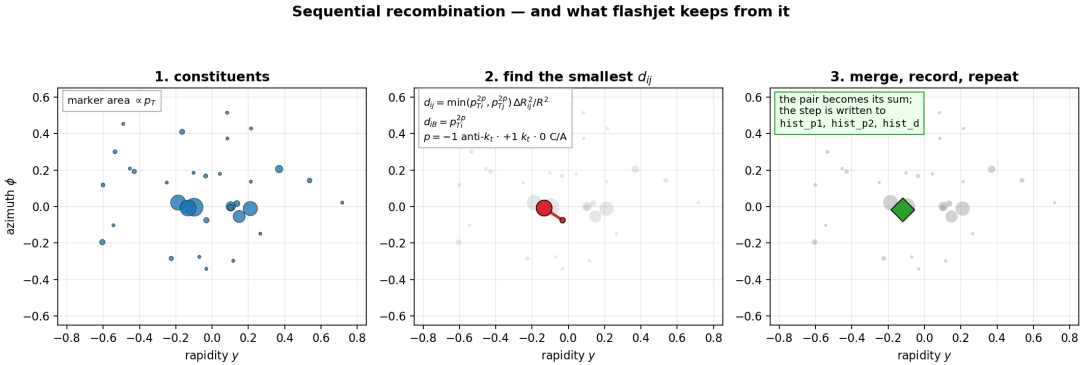
Real detector-level **calorimeter-cluster constituents**

	sample	units	$\langle N \rangle$
jet	boosted top	30 000	59.2
jet	QCD (q/g)	30 000	52.4
event	$t\bar{t}$ reco	3 000	591.6

Top Tagging Open Data `10.7483/OPENDATA.ATLAS.FG5F.96GA`; jet-reconstruction set `10.7483/OPENDATA.ATLAS.L806.5CKU`.

Every results slide carries one of these tags **in its title bar**, so it is always clear whether a plot is **generated** or **measured**. Generation details in backup.

How sequential recombination works [toy A]



Two distances, computed for every pair:

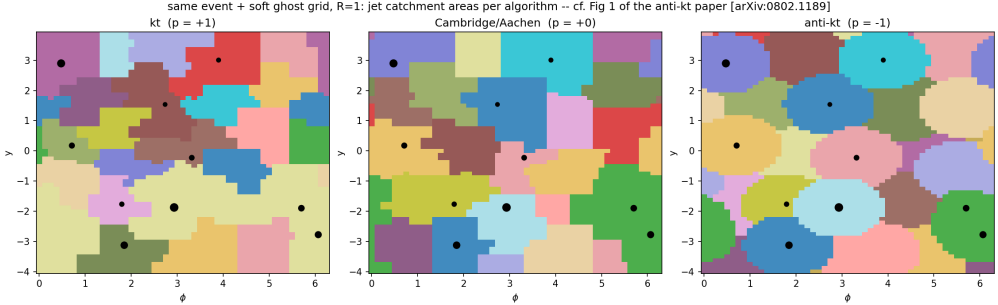
- d_{ij} — *pair* distance, particle i to particle j
- d_{iB} — *beam* distance, particle i to the beam

Then repeat until nothing is left:

1. find the smallest of all distances
2. if it is a d_{ij}: **merge** i and j
3. if it is a d_{iB}: call i a **jet**, remove it

anti-k_t gives rigid, circular jets

[toy B]



Only the exponent p changes:

p	algorithm	merges first	tree ordered by	used for	**panel above**
-1	anti-k_t	the **hardest** pairs	—	**finding jets**	circular, rigid
0	Cambridge–Aachen	the **closest** pair	**angle**	grooming, Lund plane	ragged edges
$+1$	k_t	the **softest** pair	**relative** p_T	exclusive subjets	most irregular

It returns the tree, not just the jets [ATLAS Open Data]

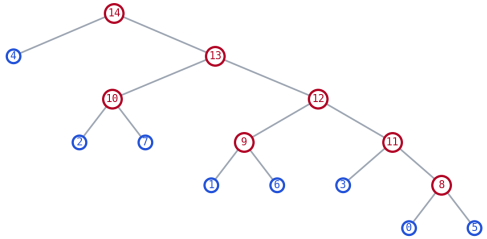
Clustering a batch returns **two** things : the jets (`jet_idx` (B, N), `n_jets` $(B,)$), and the **merge history** below: `hist_p1`, `hist_p2` (which pair merged), `hist_child` (what they merged into) and `hist_d` (the distance), each (B, N).

what the kernel writes out (step →)

hist_p1	0	1	2	3	9	10	4
hist_p2	5	6	7	8	11	12	13
hist_child	8	9	10	11	12	13	14
hist_d	0.000	0.000	0.000	0.029	0.030	0.053	0.059
step	0	1	2	3	4	5	6

each column is one merge: particles p_1, p_2 join into *child*, at distance *d*. Indices ≥ 8 are merged pseudo-particles.

the binary tree those rows encode



blue = the 8 input particles red = merged pseudo-particles

One real ATLAS jet with 8 constituents, C/A. Read column 0: particles 0 and 5 merge into 8 Seven columns, seven merges, and the tree is complete. **Everything downstream is a read of these rows**, never a second clustering.

What is implemented

Clustering

- generalised-k_t: anti-k_t, k_t, C/A
- merge history recorded in-kernel
- many jets clustered in one batched call
- scales from single jets to whole events

Substructure reads of the history

- **F1** exclusive k_t subjets
- **F2** soft drop / mMDT / mass-drop
- **F3** primary Lund coordinates

All three are pure-torch post-reads: no kernel changes, CPU/GPU identical, negligible cost.

Exclusive subjets from the k_t history [toy A]

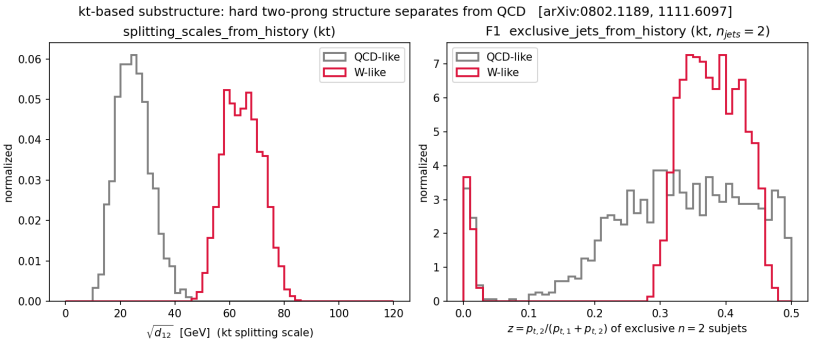
What it does. Undo the last merges of the sequence to expose exactly n subjets or stop at a d_{cut}.

How. The history is already a binary tree. Walk back up it $n-1$ steps; no reclustering.

Why k_t. k_t merges the softest pair first, so the *last* merges are the hard prongs which is exactly what a 2-prong decay leaves behind.

The splitting scales come straight from `hist_d`:

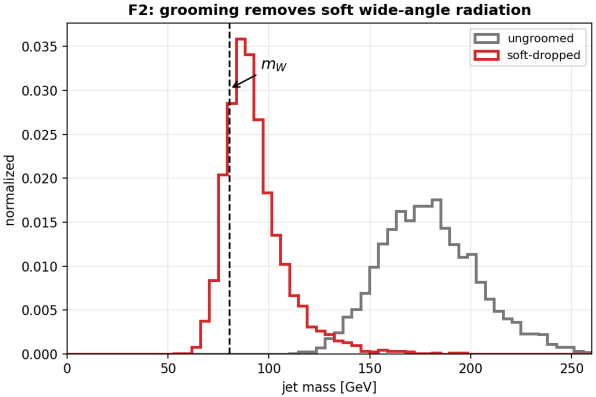
$$\sqrt{d_{12}},\ \sqrt{d_{23}},\ \ldots$$

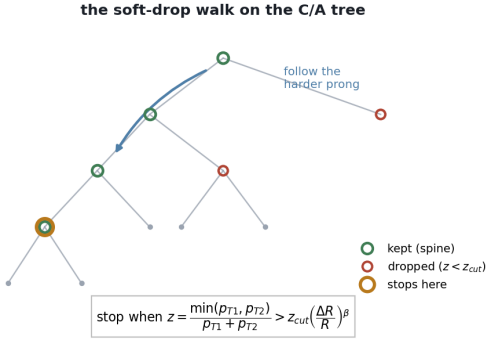


Toy W-like (2-prong) vs QCD-like fat jets. **Left:** the k_t splitting scale $\sqrt{d_{12}}$ separates cleanly. **Right:** the momentum balance z of the two exclusive subjets: the W sits near $z \sim 0.4$, QCD spreads to small z.

Soft drop / mMDT grooming

[toy A]





Walk down the C/A tree along the harder branch, discarding the softer prong until that condition is met. $\beta = 0$ recovers mMDT.

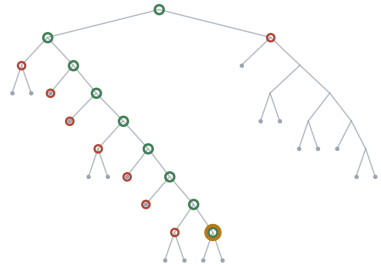
The walk is $O(\log_2 n)$ on the stored tree and it never touches the constituents again, so grooming is essentially free once clustered.

What the tree looks like on real jets

[ATLAS Open Data]

vertical = declustering depth · horizontal = angular ordering

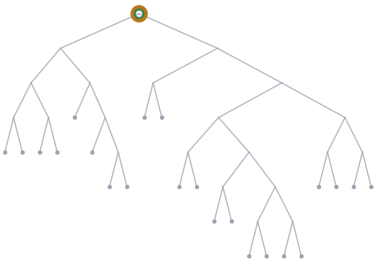
Light quark / gluon jet



a long spine: soft prong after soft prong is stripped, and the mass collapses

n_{drop} = 8 `m: 80 → 1 GeV`

Boosted top — a clean two-prong core



the very first split is already balanced, so nothing is groomed away

n_{drop} = 0 `m: 80 → 80 GeV`

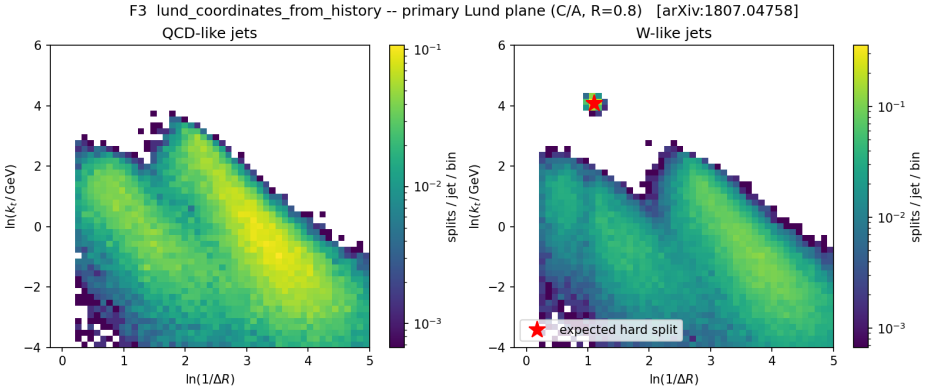
Boosted top — the full $t \to bW$ system

also stops at once, but on a wider split, so the whole decay is kept

n_{drop} = 0 `m: 170 → 170 GeV`

soft-drop spine · dropped prong · grooming stops here · constituent

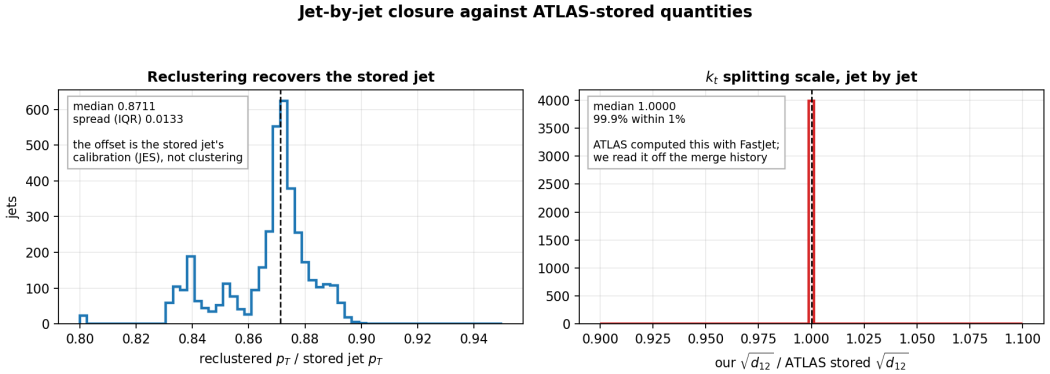
Primary Lund coordinates [toy A]



Every primary splitting emits $z = \frac{p_{T2}}{p_{T1}+p_{T2}}$, ΔR, $k_t = p_{T2}\Delta R$, and the plane coordinates $\left(\ln \frac{1}{\Delta R},\ \ln k_t\right)$. QCD fills the soft-collinear region smoothly; the W-like sample adds a **localised hard splitting** exactly where the decay is predicted to sit (star).

Correctness

Jet by jet, against ATLAS's own stored values [ATLAS Open Data]



Left. Reclustering the stored constituents recovers the stored jet with a **0.013 spread**. The 0.87 offset is the stored jet's **calibration**, applied after clustering and it is present before we touch anything.

Right. The k_t splitting scale $\sqrt{d_{12}}$ against the value **ATLAS computed with FastJet** and stored: median **1.0000**, **99.9 %** within 1 %. We read ours straight off the merge history.

It gives the same jets everywhere we looked

[ATLAS Open Data]

vs FastJet

3 algorithms $\times$ 5 radii $\times$ 5 multiplicity bins $\times$ 2 samples, 20 000 jets each.

100.000 % jet-count agreement at **every** point.

Leading-jet p_T within 10^{-4} at **148/150**; median difference $\sim 7 \times 10^{-8}$ attributed to float32 precision.

Per-point tables and the R/multiplicity scans are in backup.

vs ATLAS's own reconstructed jets

Cluster the ∼600 stored calorimeter clusters ourselves; compare to the jets ATLAS reconstructed and published.

100.0 % n_{jets} match, **10.96 vs 10.96** jets/event.

The event regime, where the problem is $O(N^2)$.

Speed

The benchmark in two regimes

[ATLAS Open Data]

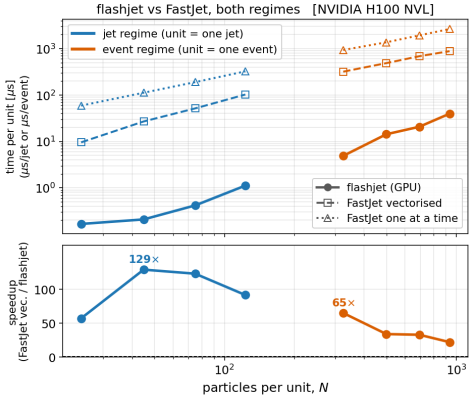
Two ways the clusterer gets used

Jet regime : one *jet* per unit

Recluster a single jet's constituents to read substructure off the tree. **Small N, huge batch**, thousands of independent jets per launch. This is the ML use case: features inside a training loop.

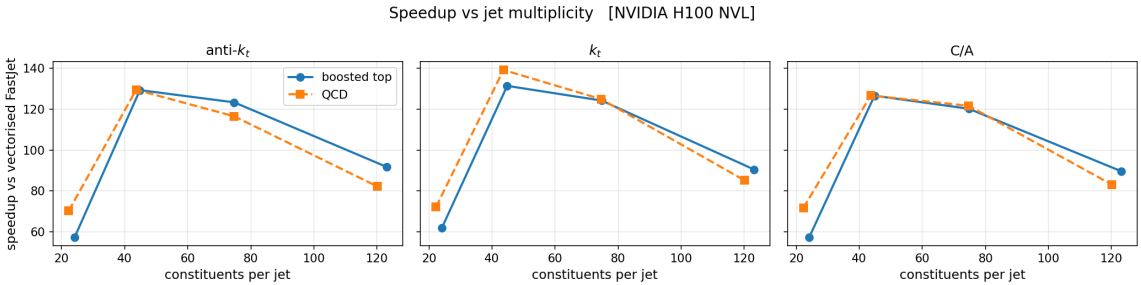
Event regime : one *event* per unit

Cluster a whole event's calorimeter clusters to *find* the jets. **Large N, modest batch** which is the classic reconstruction task.



vectorised = one FastJet call over the whole batch (its awkward-array interface); **one at a time** = a separate `ClusterSequence` per jet. Both run FastJet on a **single CPU thread**.

How the speedup varies by jet regime [ATLAS Open Data]



54–143× over vectorised FastJet (median **89**×), every point at **100 %** n_{jets} agreement.

All three algorithms, both samples, $R = 1.0$.

Using it

```
import flashjet

out = flashjet.cluster(p4, mask, R=1.0, algorithm="antikt")
```

argument	type	
`p4`	$(B, N, 4)$ tensor	(p_x, p_y, p_z, E); CPU or CUDA
`mask`	(B, N) bool	marks real constituents in a padded batch
`R`	float	jet radius
`algorithm`	str	`"antikt"` | `"kt"` | `"cambridge"`
`p`	float	generalised-k_t exponent (overrides `algorithm`)

```
out.n_jets                                  # (B,)     jets found
out.jets_p4(p4)                             # (B,J,4)  jet four-momenta
out.splitting_scales()                      #          sqrt(d12), sqrt(d23), ...
out.exclusive_jets(n_jets=3)                # F1       exclusive kt subjets
out.groomed_jets(p4, R, z_cut=0.1, beta=0.0) # F2      soft drop / mMDT
out.lund_coordinates(p4, R)                 # F3       primary Lund plane
```

Downstream: the clustering tree as tagger input

Does the tree help a tagger?

The question. Gouskos & Maier [PLuM, 2605.26821] report that explicit Lund-plane splittings help a transformer tagger. We test that on a **full 10-class** training.

Model. Particle Transformer [2202.03772] : 8 encoder + 2 class-attention blocks, 8 heads, embed 128; **2.144 M** parameters.

Data. JetClass, 10 classes, 100 M training jets, ∼**20.05 M** test jets, 128 particles/jet.

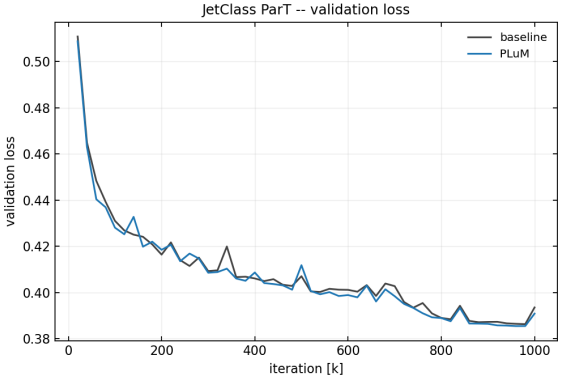
Training. Ranger, lr 10^{-3}, batch 512, 10^6 **iterations** to convergence. All arms read **identical shards** in the same order; features are computed **live** by flashjet inside the loop.

Our baseline reproduces the ParT paper

	ours	published
test accuracy	**86.21 %**	∼86.1 %
parameters	2,193,930	2.19 M

Shown here: PLuM : 48 k_t-ordered Lund splittings $[\ln k_t, \ln 1/\Delta R, \ln z]$, concatenated as extra tokens (176 vs 128). Two other encodings were tried — **subjet** (n-body mass, p_T fraction, n_{const}) and **CA5** (per-particle C/A branch points); both null-or-worse (backup).

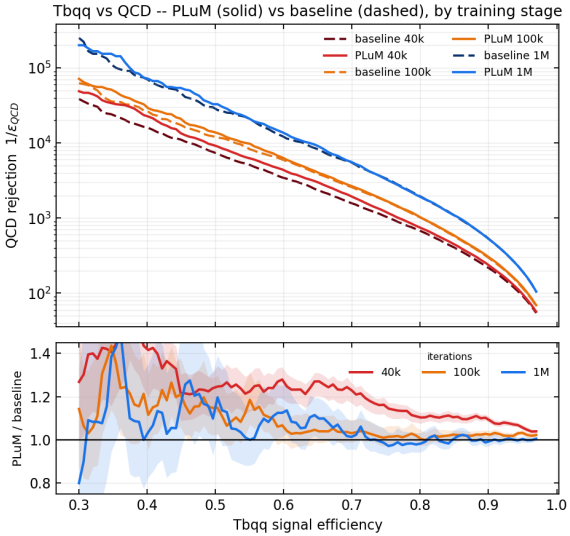
Both arms train identically



The curves are **on top of each other**: the feature block neither destabilises training nor accelerates it. Both converge to $\sim$**86.2 %**.

Validation loss over the full 10^6 iterations.

Top tagging: the gain closes with training



$t \to bqq$ **vs QCD.**

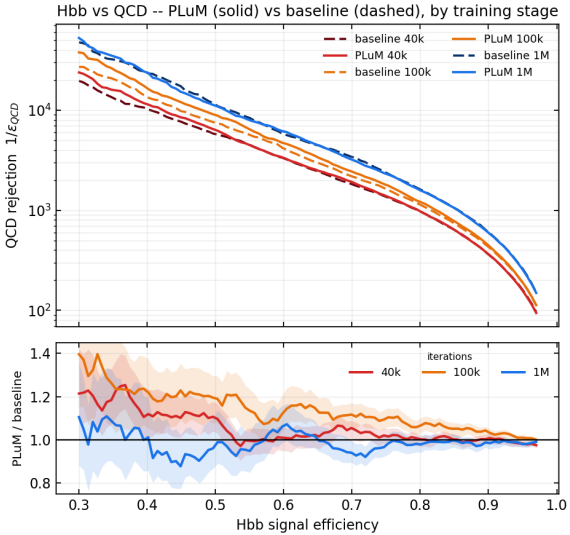
Top panel: full ROC at three training stages.
Bottom: **PLuM / baseline** rejection ratio.

At **40k** the ratio sits **10–25 % above 1**. By **1M** it has collapsed onto the line.

Bands: unpaired bootstrap of the test set, 16–84th percentile. The low-efficiency end is noisy because rejection there is $\sim 10^5$.

Top right = BETTER

$H \to b\bar{b}$: the paper's best channel



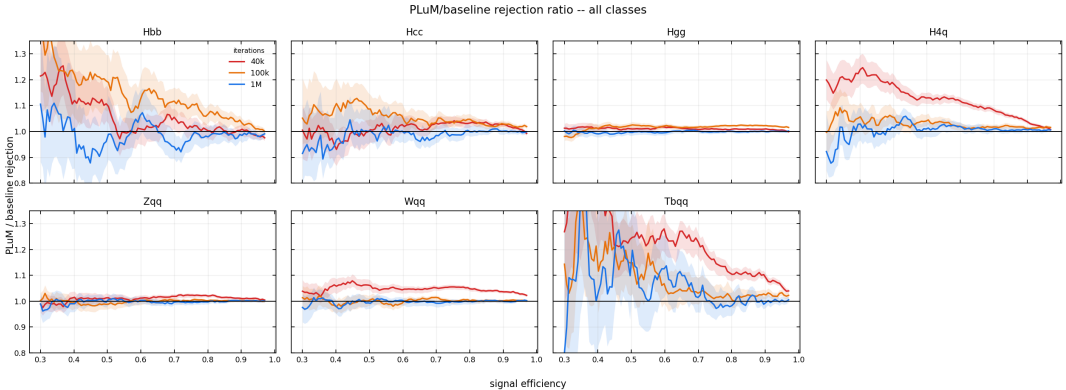
$H \to b\bar{b}$ **vs QCD** where PLuM report their largest gain.

We **reproduce it early**: $+20\,\%$ at 100k. At **1M** the ratio is **at or below 1**.

Their result is obtained in *binary* mode ; ours is a 10-class training. The early-training agreement is real , the disagreement is about what survives to convergence.

Top right = BETTER

All seven signal classes



Test accuracy ($\sim$20.05 M jets)

	40k	100k	1M
baseline	83.431	84.632	**86.212**
PLuM	83.428	84.697	86.206
Δ	-0.003	$+0.065$	-0.006

An **early gain in every class**, **gone by 1 M**. Our conclusions agree with what Sitian presented yesterday(link) from a **Jacobian lens** perspective

Summary

1. **Clusters on the GPU and keeps the full merge history**, so substructure like exclusive subjets, soft drop, Lund coordinates comes free as array reads. Built for **reclustering inside a training loop**, on tensors that never leave the GPU.
2. **Gives the same jets as FastJet.** 150/150 grid points at 100 % n_{jets} agreement, across anti-k_t/k_t/C-A, $R = 0.4$–1.2 and 24–123 constituents per jet.
3. **Reproduces the experiment's own reconstructed jets** exactly, at ∼600 clusters per event.
4. **54–143× faster** than vectorised FastJet
5. The tree as tagger input gave no lasting gain.
6. Still a **prototype**, with meaningful headroom left.
7. **Public**, as part of **jet-universe**: `github.com/jet-universe/FlashJet`

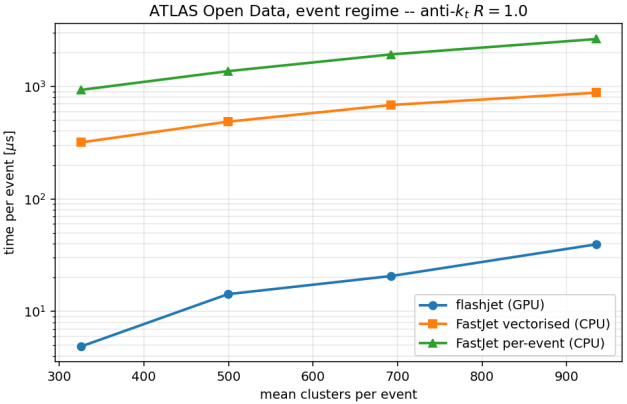
Thank you

Questions?

Backup

Backup — the event regime

[ATLAS Open Data]

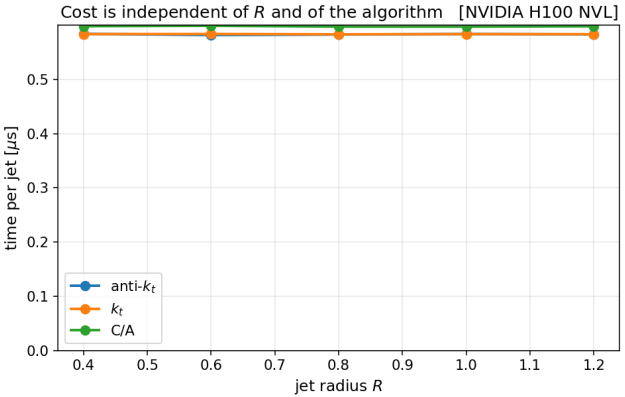


One timed unit = one whole event ($\sim$590 ATLAS calorimeter clusters), 3 000 events, same method as the jet regime.

$\langle N \rangle$	μs/event	speedup
325	4.9	**65**$\times$
499	14.3	34$\times$
693	20.6	33$\times$
935	39.5	22$\times$

22–65$\times$ over vectorised FastJet. Jet-count agreement: **60/75 grid points exactly 100 %**, worst **99.785 %** (busiest bin); leading-jet p_T within 10^{-4} at **all 75**. **H100 NVL** — the same card as the jet-regime plots, so the two are directly comparable.

Backup — cost is independent of R and of the algorithm [ATLAS Open Data]



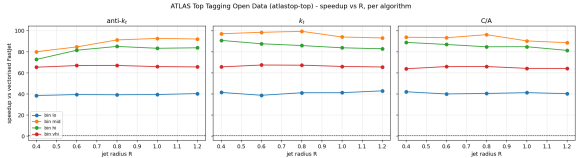
Across $R = 0.4 \to 1.2$: a **0.3 % spread** over a $3\times$ range in R.

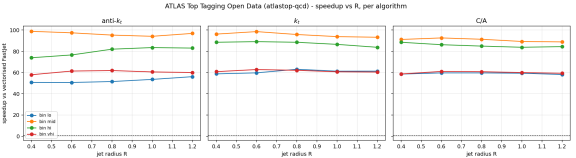
R changes *which* pairs merge, not *how many* distances are computed.

Across algorithms: anti-k_t, k_t and C/A agree to within $\sim$4 % at every multiplicity.

k_t and C/A **cost the same as anti-k_t.**

Backup — speedup vs R, per algorithm (jet regime) [ATLAS Open Data]



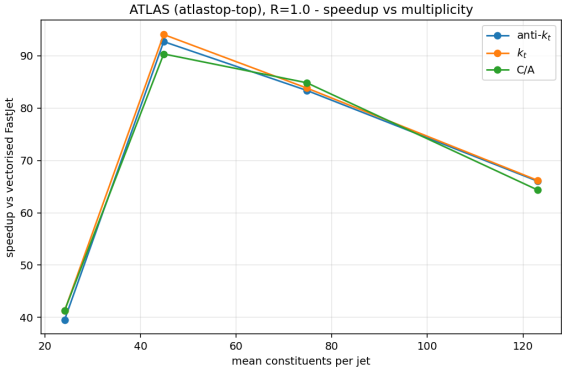


boosted top

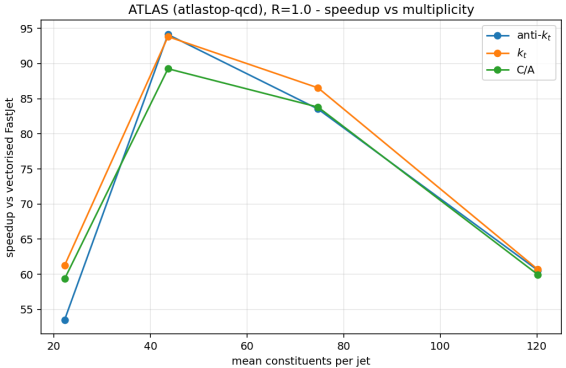
QCD (light q/g)

Flat in R for all three algorithms; ordering is by multiplicity bin.

Backup — speedup vs multiplicity (jet regime) [ATLAS Open Data]



boosted top



QCD (light q/g)

Backup — the numbers (jet regime)

sample	alg	$\langle n \rangle$	flashjet [μs/jet]	FastJet vec. [μs/jet]	speedup
top	anti-k_t	24.1	0.427	16.5	$39\times$
top	anti-k_t	44.8	0.464	43.0	$\mathbf{93}\times$
top	C/A	74.7	1.000	84.8	$85\times$
top	C/A	123.1	2.565	165.1	$64\times$
qcd	anti-k_t	22.2	0.346	17.5	$51\times$
qcd	anti-k_t	43.6	0.463	45.7	$\mathbf{99}\times$
qcd	anti-k_t	74.7	1.215	89.8	$74\times$
qcd	anti-k_t	120.2	2.673	154.9	$58\times$

54–143$\times$ over vectorised FastJet (median $89\times$), with 100 % jet-count agreement at all 150 points. **One timed unit = one jet**, batched. Benchmarked on an **NVIDIA H100 NVL**.

$R = 1.0$ rows shown. Against the per-jet FastJet interface the gap is roughly *two orders of magnitude*.

Backup — where the time goes

Nsight Systems — share of GPU time

kernel	% GPU	avg/call
`_cluster_large_kernel`	**97.6**	5.47 ms
`_decode_kernel`	1.5	82 μs
torch glue	< 1	—

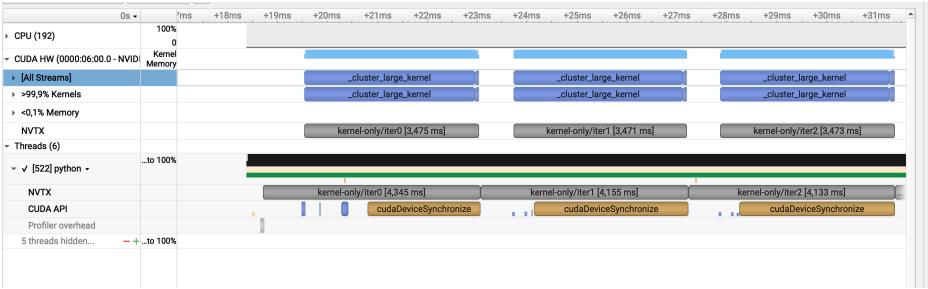
No hidden overhead — essentially all the time is real clustering work, and the memory row of the timeline is empty.

Nsight Compute — hardware counters

metric	value
DRAM throughput	**0.02 %**
Compute (SM) throughput	15.4 %
Registers / thread	168
Theoretical occupancy	18.75 %
Achieved occupancy	**6.25 %**
Waves per SM	**0.32**

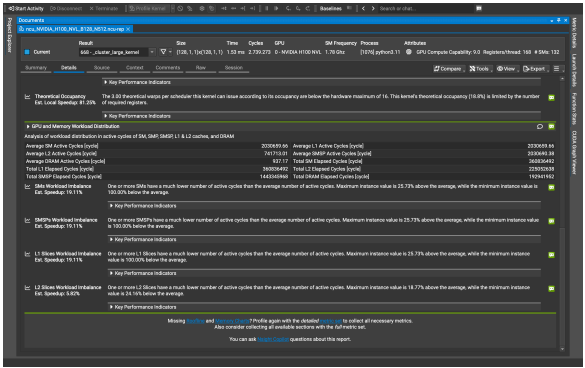
Not memory- or compute-bound — the GPU is simply **not filled**.

Backup — the GPU timeline



Nsight Systems, three consecutive iterations. Each block is one clustering call; the tool labels the rows **“>99.9 % Kernels”** and **“<0.1 % Memory”**. The kernels run back to back with nothing in between and the memory row is empty — no host/device shuffling while clustering runs. That is what makes the speedup real rather than an artefact of where the timer was started.

Backup — the profiler names the bottleneck



The tool's own speedup estimates:

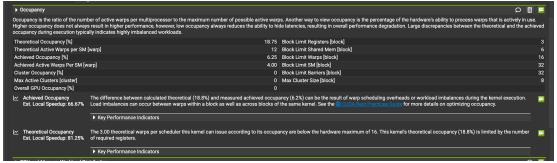
finding	est. speedup
Theoretical occupancy (registers)	**81 %**
Achieved occupancy	67 %
SM workload imbalance	19 %

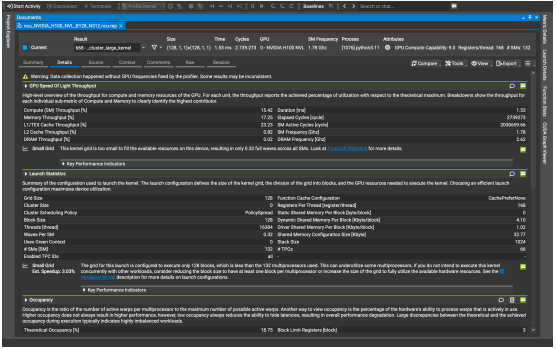
"This kernel grid is too small to fill the available resources on this device, resulting in only 0.32 full waves across all SMs."

Grid size **128 blocks** on a **132-SM** device — fewer blocks than the GPU has processors.

$\sim$**1.7–1.8**$\times$ more available from launch configuration and register budget — **not** from the algorithm.

Backup — occupancy detail

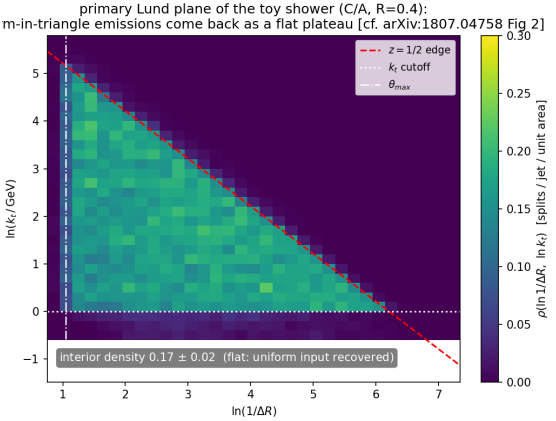




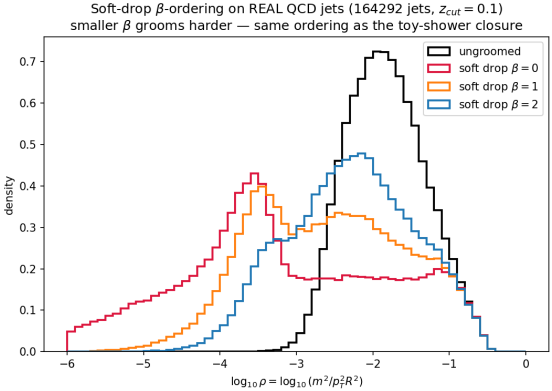
Register pressure (168/thread) caps theoretical occupancy at 18.75 %; achieved is 6.25 %. With 0.32 waves/SM the kernel does not fill the GPU even once, while DRAM throughput is 0.02 % — this is parallelism starvation, not a bandwidth problem. Both are configuration inherited from smaller GPUs, not properties of the algorithm.

Backup — more toy closures

[toy B]



The Lund plane is populated **uniformly inside the kinematic triangle** — the leading-log expectation.



The soft-drop β-family orders as predicted: larger β grooms less.

Independent NumPy tree-walks pin every decoder, and the d-channel of the Lund output ties *exactly* to the splitting scales computed separately.

Backup — Input A: the toy fat jets

Purpose. QCD vs W is a *known* truth, so any observable that fails to separate them — or lands off the analytic curve — is a real bug.

Two single-fat-jet samples at $p_T \approx 500\,\text{GeV}$, $R = 0.8$:

- **QCD-like**: one hard collinear core (92 % of p_T, $\sigma_{y,\phi} = 0.06$) + soft wide-angle radiation (8 %, $\sigma = 0.30$)
- **W-like**: two hard prongs with pair mass $m = \sqrt{z(1-z)}\, p_T\, \Delta R = 80.4\,\text{GeV}$, $z \in [0.30, 0.45]$, random orientation, + 6 % soft contamination

Each "spray" is n massive pions with exponential p_T fractions:

```
frac = rng.exponential(1., n); frac /= frac.sum()
pt   = pt_total * frac
y, phi = rng.normal(y0, sigma, n), rng.normal(phi0, sigma, n)
mt   = np.sqrt(M_PI**2 + pt**2)
```

1500 events per sample. flashjet has **no event generator** — it clusters four-vectors; these toys live in the plot scripts, outside the library.

Backup — Input B: the toy parton shower

A **primary, fixed-coupling, leading-log** shower: emissions sampled **uniformly in the Lund triangle** with density $\bar{\alpha}$ per unit $(\ln 1/\theta,\ \ln k_t)$ area, off a hard spine.

$$\theta = e^{-u}, \qquad k_t = e^{v}, \qquad z = \frac{k_t}{p_{T0}\,\theta}$$

$$\text{keep } v \le \ln(p_{T0}/2) - u \quad (\text{i.e. } z \le \tfrac{1}{2}), \quad k_t > k_t^{\min}$$

Why this is the right test

The generator's density is *uniform in the triangle by construction*, so the leading-log predictions — the $1/z$ form of z_g, the β-ordering of the groomed mass — are unambiguous. Landing on them tests the **decoders**, not the shower.

Jet areas use the anti-k_t-paper construction: one hard event plus a dense grid of **infinitely soft ghosts** ($p_T = 10^{-8}$); the ghosts each algorithm sweeps up trace its catchment area.

$\bar{\alpha} = 0.25$, $p_{T0} = 1$ TeV, $R = 0.4$, 20 000 showers; fixed seed, fully re-runnable.

Backup — cost of each feature arm

Wall clock per training step, relative to the baseline:

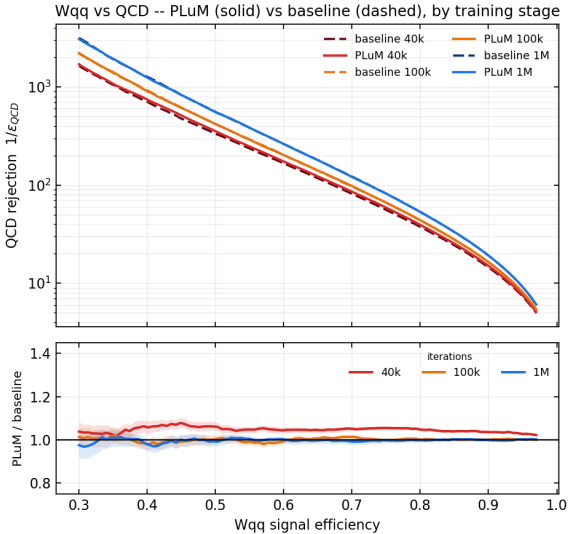
arm	cost
PLuM	**1.59**×
subjet	1.42×
CA5	1.22×

Features computed live by flashjet inside the loop, on the batch already resident on the GPU. Parameters: 2.144 M baseline → 2.194 M with the PLuM feature block ($+0.051$ M).

Measured before the GPU-side collation change; the clustering itself is a small fraction of a step, so these are upper bounds on the true feature cost.

The cost is the reason a null result still matters: if a feature is free, a null is merely uninteresting — at **1.59**× it is a reason not to use it.

Backup — $W \to qq$, the fewest-splitting channel



$W \to qq$ has the **fewest** Lund splittings of any signal class (21.6/jet), yet shows the same early gain.

The effect is **not** confined to complex or b-containing jets — which is what a decay-structure explanation would predict.

Backup — two kinds of validation

First: toys, where truth is known

On generated inputs the *answer* is known analytically — so any observable landing off the predicted curve is a real bug, with nothing to hide behind.

Then: real data, against FastJet

On detector-level jets there is no analytic answer — so the reference is **FastJet on the same events**, and the experiment's own reconstructed jets.

Input A : toy fat jets — QCD-like and W-like fat jets

Input B : toy parton shower — leading-log parton shower

Input C : ATLAS Open Data — large-R jets and full events

The order matters: **toys first** pins the algorithm against a known answer; **real data second** shows it survives detector-level input. A failure at either stage means something different.

Backup — the validation ladder

rung	input	reference	result
unit tests	—	independent NumPy tree-walks	129 pass, incl. GPU paths
anti-k_t jet areas	toy B	the anti-k_t paper's own construction	rigid cones, $\to \pi R^2$
k_t substructure	toy A	known 2-prong vs QCD truth	clean separation
soft-drop z_g	toy B	leading-log $1/z$ prediction	on the curve
groomed mass vs β	toy B	Soft Drop Figs. 3–4	ordering reproduced
Lund plane	toy A	primary-Lund construction	hard split where predicted
jet-by-jet kinematics	real C	FastJet, same events	150/150 points, 100 % n_{jets}
full-event clustering	real C	the experiment's own jets	100 % match, ∼600 clusters/ev

Each rung is closed against something independent — an analytic prediction, a separate NumPy implementation, or FastJet itself. No rung is validated against another part of flashjet.

Backup — references

- H. Qu, C. Li, S. Qian, *Particle Transformer for Jet Tagging*, arXiv:**2202.03772**. (also the JetClass dataset; Zenodo 10.5281/zenodo.6619768)
- L. Gouskos, B. Maier, *Particle-Lund Multimodality in Jet Taggers*, arXiv:**2605.26821**.
- H. Qu, L. Gouskos, *Jet tagging via particle clouds* (ParticleNet), Phys. Rev. D **101** (2020) 056019, arXiv:1902.08570.
- F. A. Dreyer, G. P. Salam, G. Soyez, *The Lund Jet Plane*, JHEP **12** (2018) 064, arXiv:1807.04758.
- F. A. Dreyer, H. Qu, *Jet tagging in the Lund plane with graph networks*, JHEP **03** (2021) 052, arXiv:2012.08526.
- P. D. Patel, S. Ganguly, *Explainable AI for Jet Tagging in the Lund Jet Plane*, arXiv:**2604.25885**.
- S. Rai, S. Ganguly, *Dissecting Jet-Tagger Through Mechanistic Interpretability*, arXiv:**2605.09881**.
- A. J. Larkoski, S. Marzani, G. Soyez, J. Thaler, *Soft Drop*, JHEP **05** (2014) 146, arXiv:1402.2657.
- M. Cacciari, G. P. Salam, G. Soyez, *The anti-k_t jet clustering algorithm*, JHEP **04** (2008) 063, arXiv:0802.1189; *FastJet user manual*, Eur. Phys. J. C **72** (2012) 1896, arXiv:1111.6097.
- S. Catani, Yu. L. Dokshitzer, M. H. Seymour, B. R. Webber, Nucl. Phys. B **406** (1993) 187 (k_t); Yu. L. Dokshitzer *et al.*, hep-ph/9707323 and M. Wobisch, T. Wengler, hep-ph/9907280 (Cambridge/Aachen).
- J. de Favereau *et al.*, *DELPHES 3*, JHEP **02** (2014) 057, arXiv:1307.6346.

Backup — the datasets

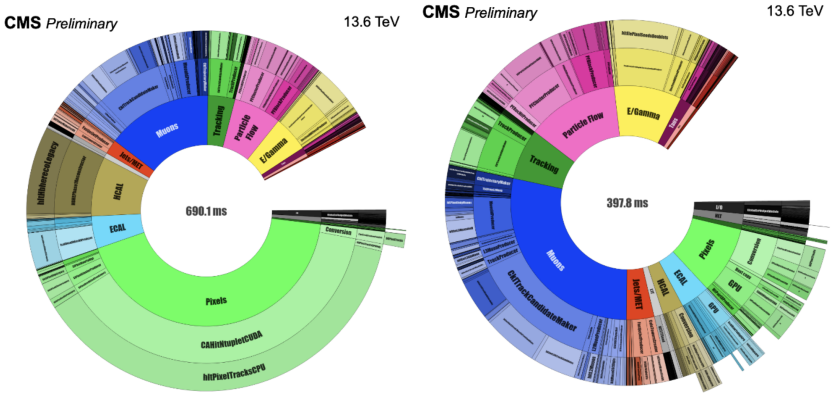
All benchmark data is **ATLAS Open Data** (CC0), read over the public redirector — no grid certificate required.

dataset	used for	DOI
ATLAS Top Tagging Open Data	timing + FastJet agreement	`10.7483/OPENDATA.ATLAS.FG5F.96GA`
(version with systematics)		`10.7483/OPENDATA.ATLAS.SOAY.LABE`
2020 ATLAS Jet Reconstruction	closure vs the experiment's own jets	`10.7483/OPENDATA.ATLAS.L806.5CKU`

All three resolve at `opendata.cern.ch`; served over `root://eospublic.cern.ch`. The toy generators are not a public dataset — they are a few lines in the plot scripts, reproduced in full in the two backup slides that follow, with a fixed seed.

Constituents are calorimeter clusters, massless. Physics closures use toy showers compared against leading-log analytic predictions, so the prediction being tested is unambiguous.

Reconstruction is moving to accelerators — clustering has not

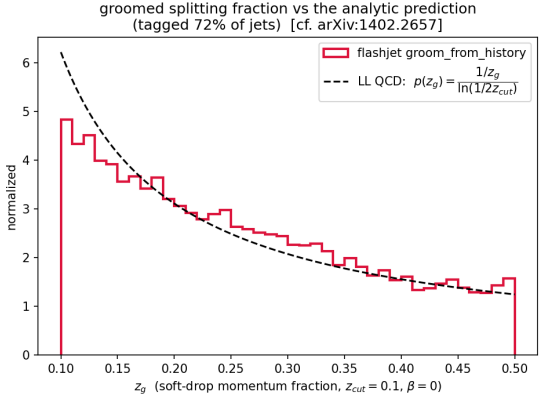


CMS HLT per-event time, CPU-only (left) vs with GPU offload (right) — ∼**30 %** of the reco offloaded buys ∼**40 %** less HLT time. **Jets/MET barely moves**: it is still on the CPU. [CMS, EPJ Web Conf. **295** (2024) 11020]

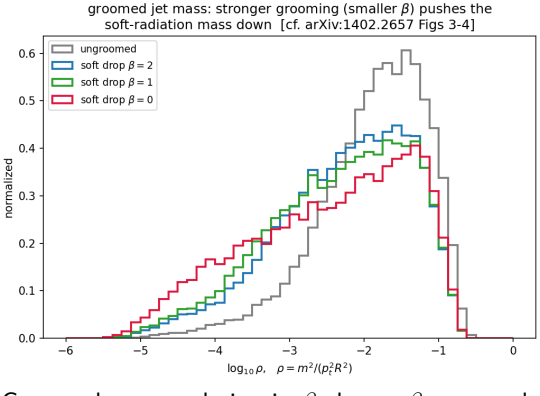
Why clustering stayed on the CPU

- sequential recombination is **IRC safe** — that is why it won
- but each merge **depends on the previous one** — serial dependence is what a GPU does worst

Closures against analytic predictions [toy B]



Soft-drop z_g follows the $1/z$ form predicted at leading-log, recovered from the C/A history.



Groomed-mass ordering in β: larger β grooms less, so the mass peak moves up.