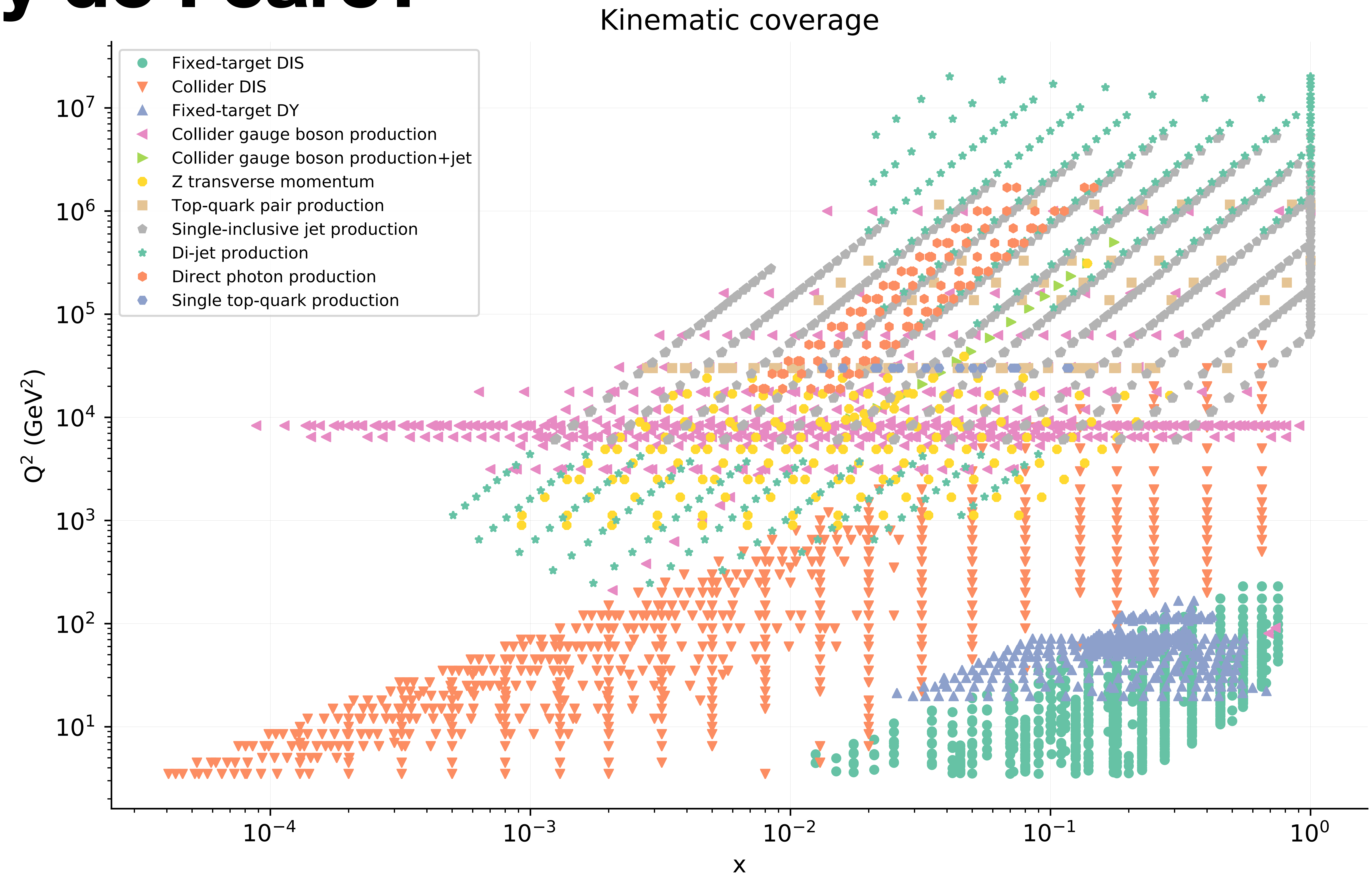
Towards a framework for GPU event generation

Juan M. Cruz-Martinez - Cern TH Department

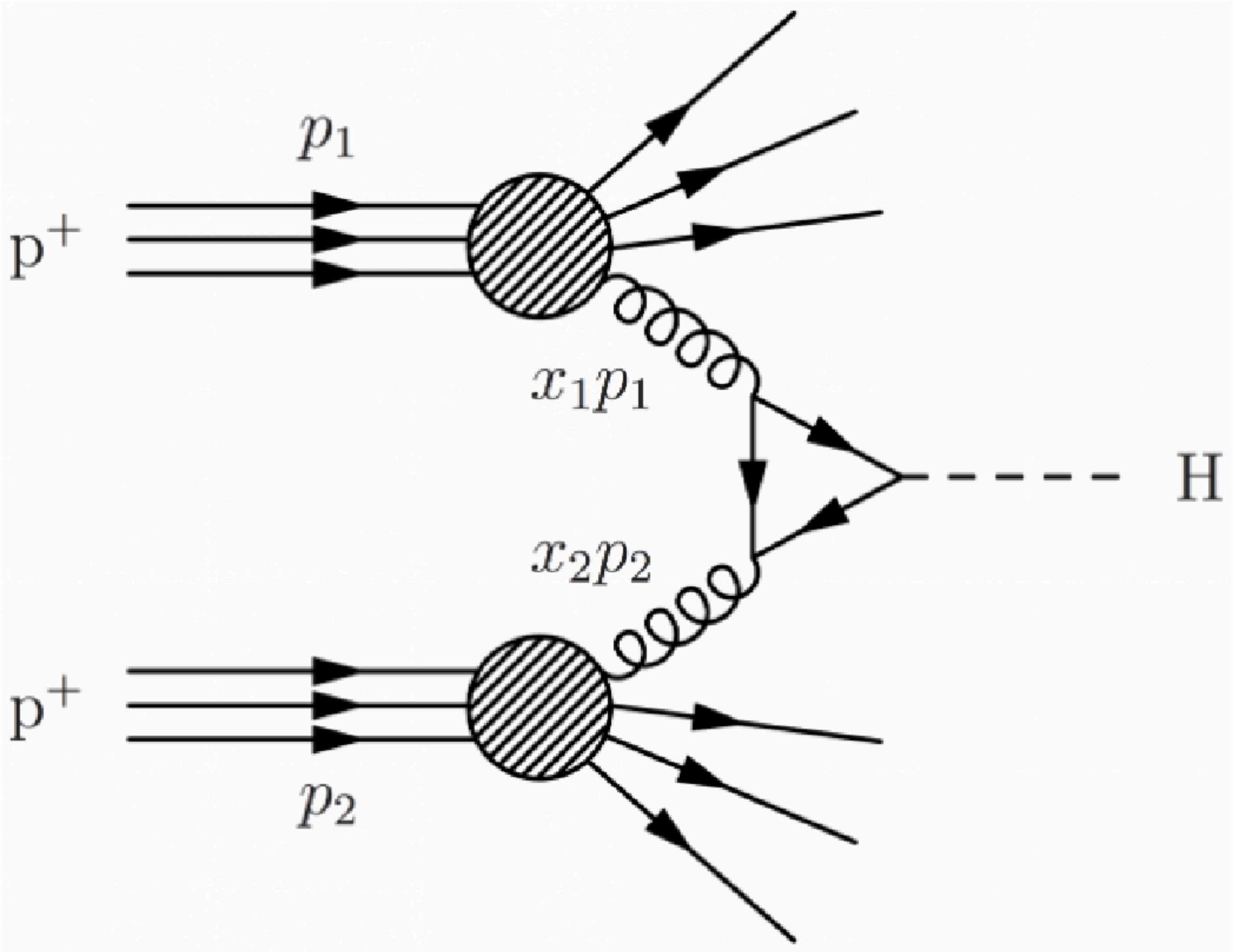
Milan Christmas Meeting, December 2023

Why do I care?


Kinematic coverage
Fixed-target DIS
Collider DIS
Fixed-target DY
Collider gauge boson production
Collider gauge boson production+jet
Z transverse momentum
Top-quark pair production
Single-inclusive jet production
Di-jet production
Direct photon production
Single top-quark production
Q^2 (GeV^2)
10^7
10^6
10^5
10^4
10^3
10^2
10^1
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^0
x

The ingredients of a Monte Carlo generator

histogramming / analysis



- Filling interpolation grids
- Loop integrals and special functions
- PDF uncertainties
- Scale variations
- Infrared subtractions
- ....

$$\mathcal{O} = \int d\Phi_n dx_1 dx_2 f_1(x_1, \mu_F^2) f_2(x_2, \mu_F^2) \,|\, M(\{p_n\}, \mu_R) \,|^2\, \mathcal{J}_m^n(\{p_n\})$$

phase space

matrix element

integrator

Parton Distribution Functions

cuts

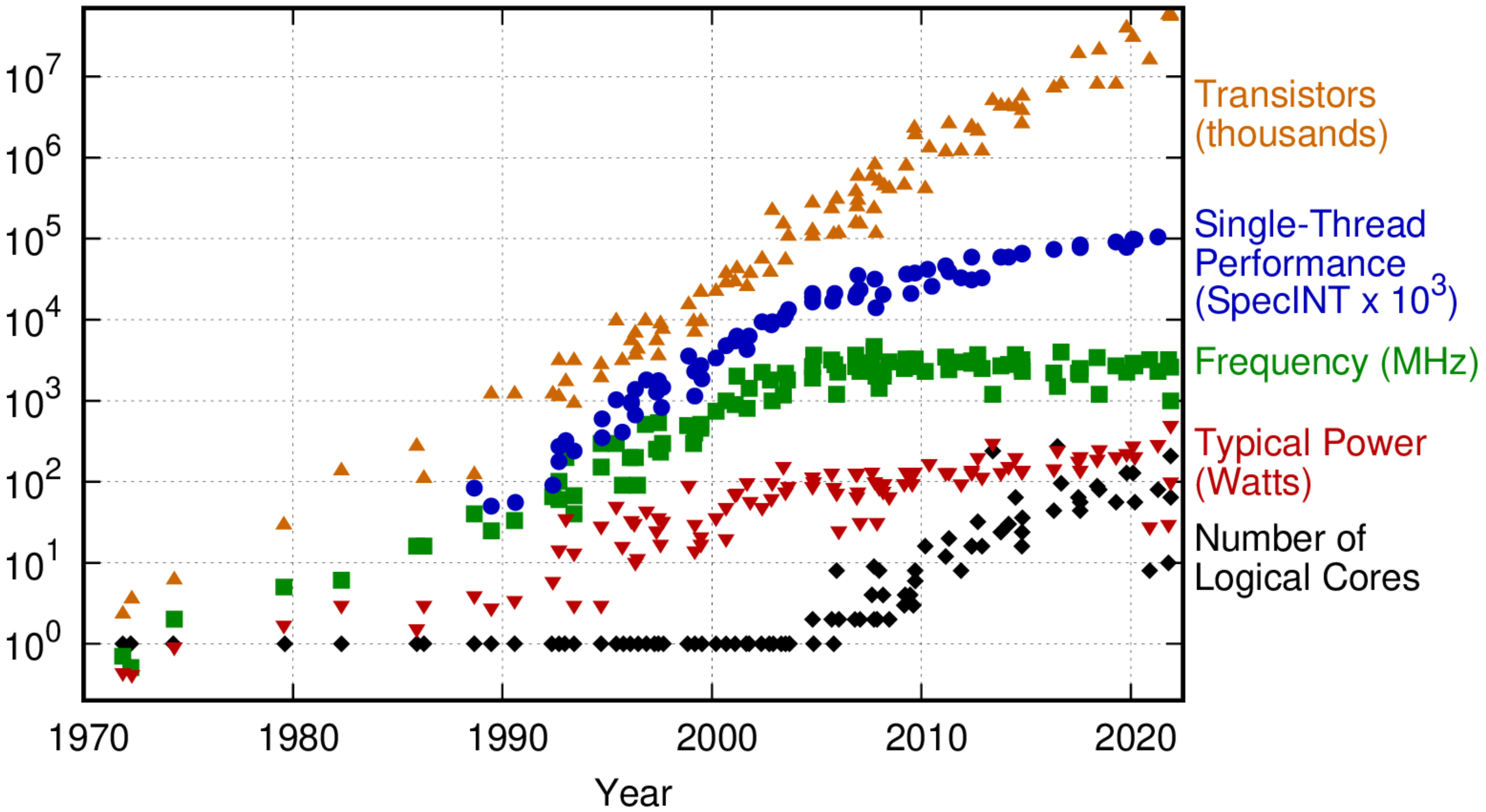
The CPU engine

As long as the performance of the CPUs increase faster than the complexity of the calculation this is not really a problem.

And when it starts being a problem… well, just build increasingly large clusters of CPUs to continue the exponential trend…

Because, luckily, Monte Carlo integrations are, in general **highly parallelizable!**

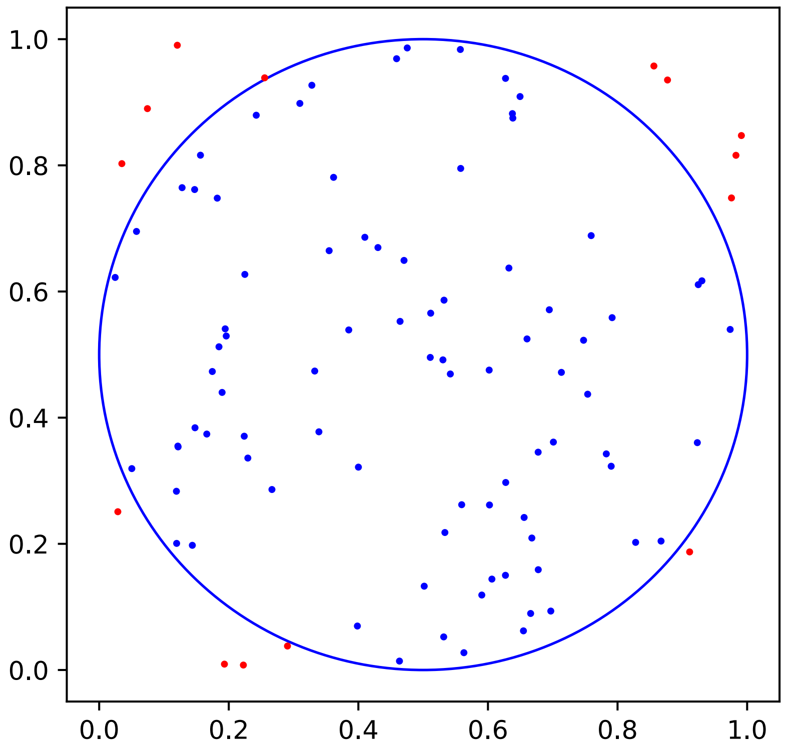
50 Years of Microprocessor Trend Data



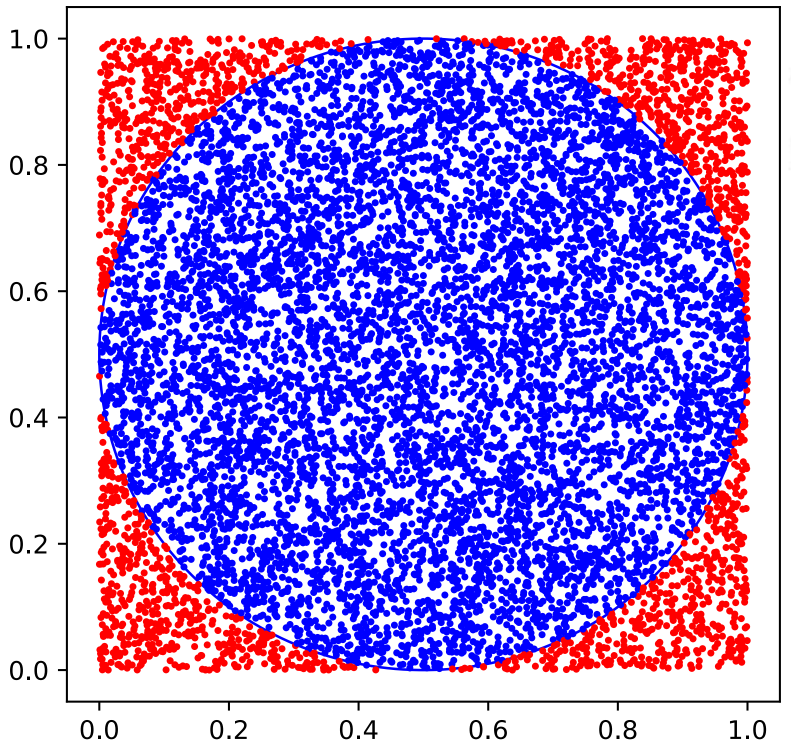
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2021 by K. Rupp

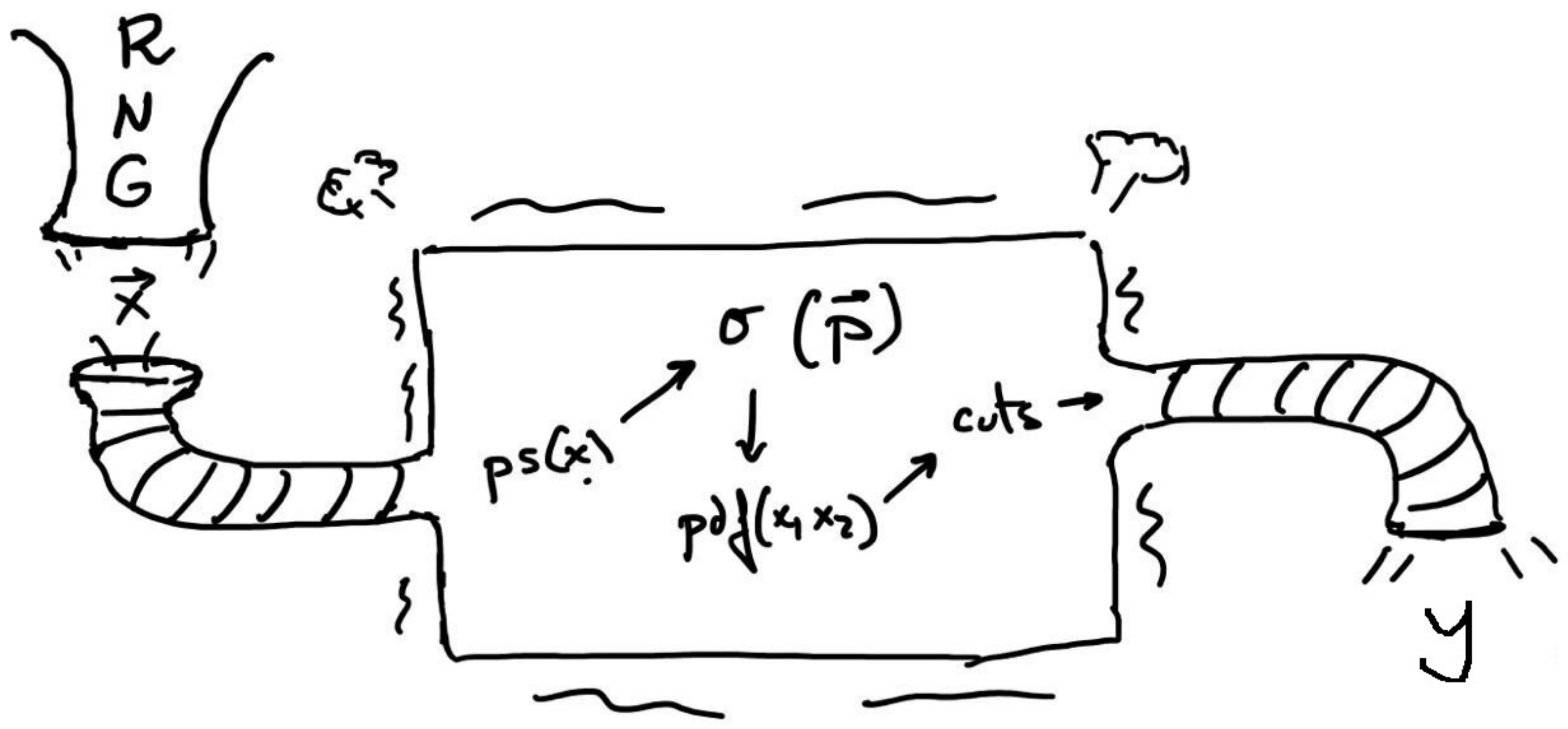
Monte Carlo integrals are highly parallelizable

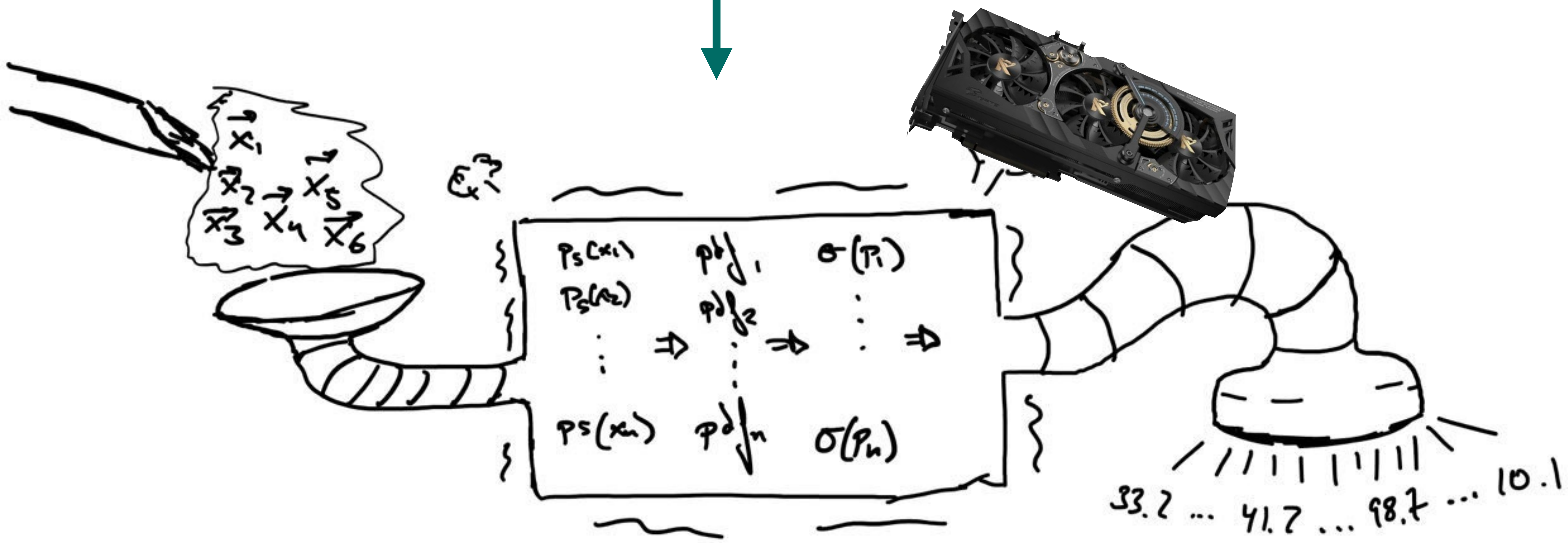
$\pi \simeq 3.2$

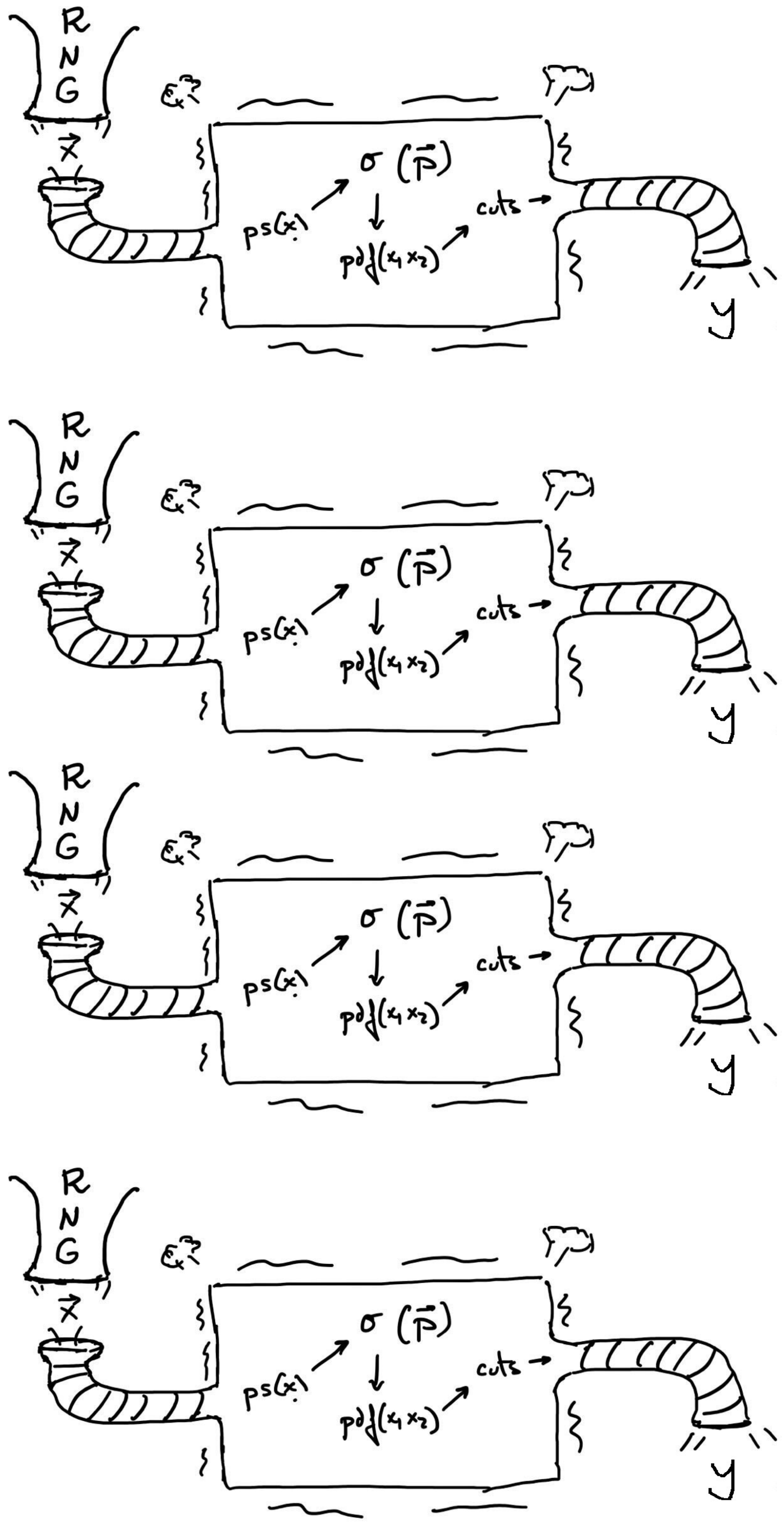


$\pi \simeq 3.16$





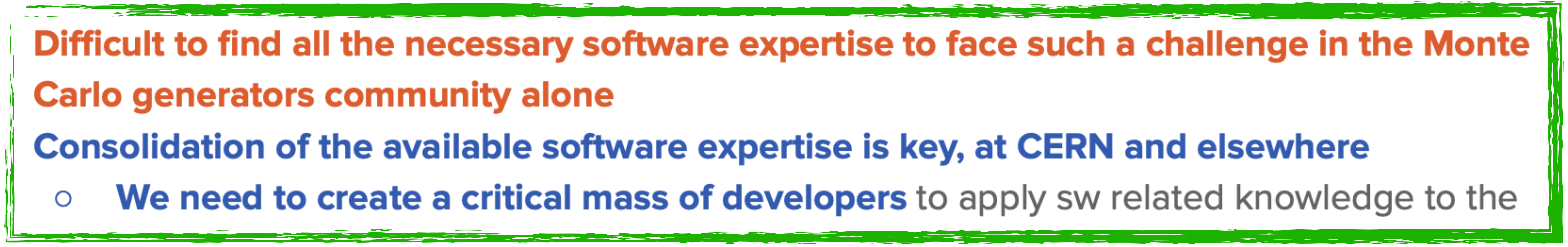




If they are so good, why are we not using them?

- Huge codebases optimized for running on CPU clusters, not necessarily in GPU-friendly languages
- Papers are needed, porting code is “wasted time”
- Existing expertise not always translate cleanly to a completely different architecture/language
- It’s a catch-22 situation!

Summary from Danilo Piparo’s talk at the Event generators' acceleration workshop in November:



Difficult to find all the necessary software expertise to face such a challenge in the Monte Carlo generators community alone

Consolidation of the available software expertise is key, at CERN and elsewhere

- **We need to create a critical mass of developers** to apply sw related knowledge to the

But, in addition, we also need suffer from a lack of tools.

We need to create a critical mass of **GPU based tools** so that development can seamlessly move towards hardware accelerators.

So you extend your current code, which allows you to get a paper out, which makes you more proficient in whatever language that code was written in, which in turns makes the codebase even larger and with it the barrier to implement it on a GPU has also grown.

The phenomenologist (CPU-based) toolbox

integrator → myriad of Vegas implementations / cuba

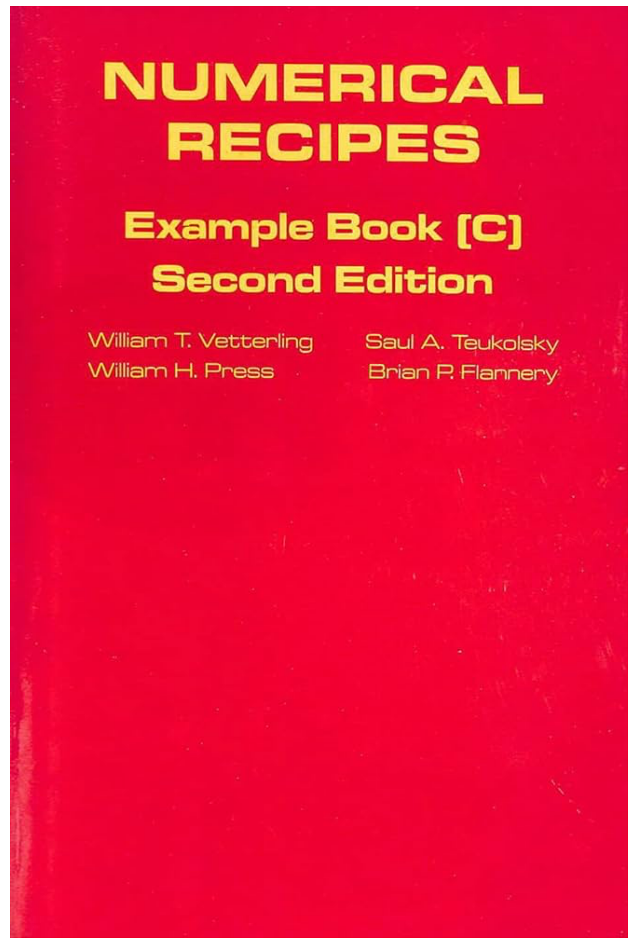
phase space → RAMBO / many algorithms available

matrix element → Madgraph / Comix / OpenLops

Parton Distribution Functions → LHAPDF

cuts → Fastjet

histogramming / analysis → Thousands of ROOT scripts



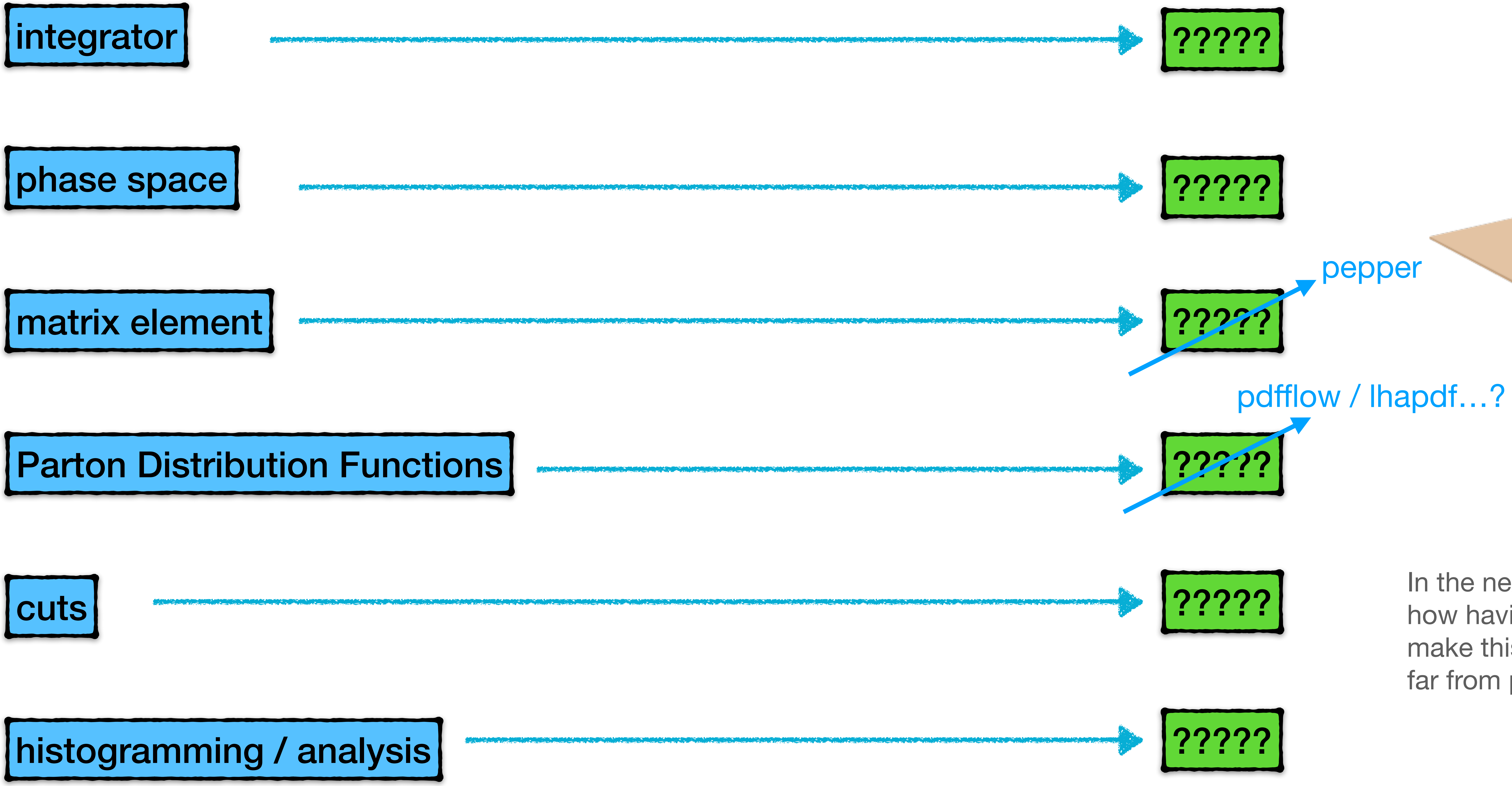
NUMERICAL RECIPES

Example Book [C]
Second Edition

William T. Vetterling
William H. Press

Saul A. Teukolsky
Brian P. Flannery

The phenomenologist (GPU-based) lack thereof



integrator → ?????

phase space → ?????

matrix element → ????? → pepper

Parton Distribution Functions → ????? → pdfflow / lhapdf…?

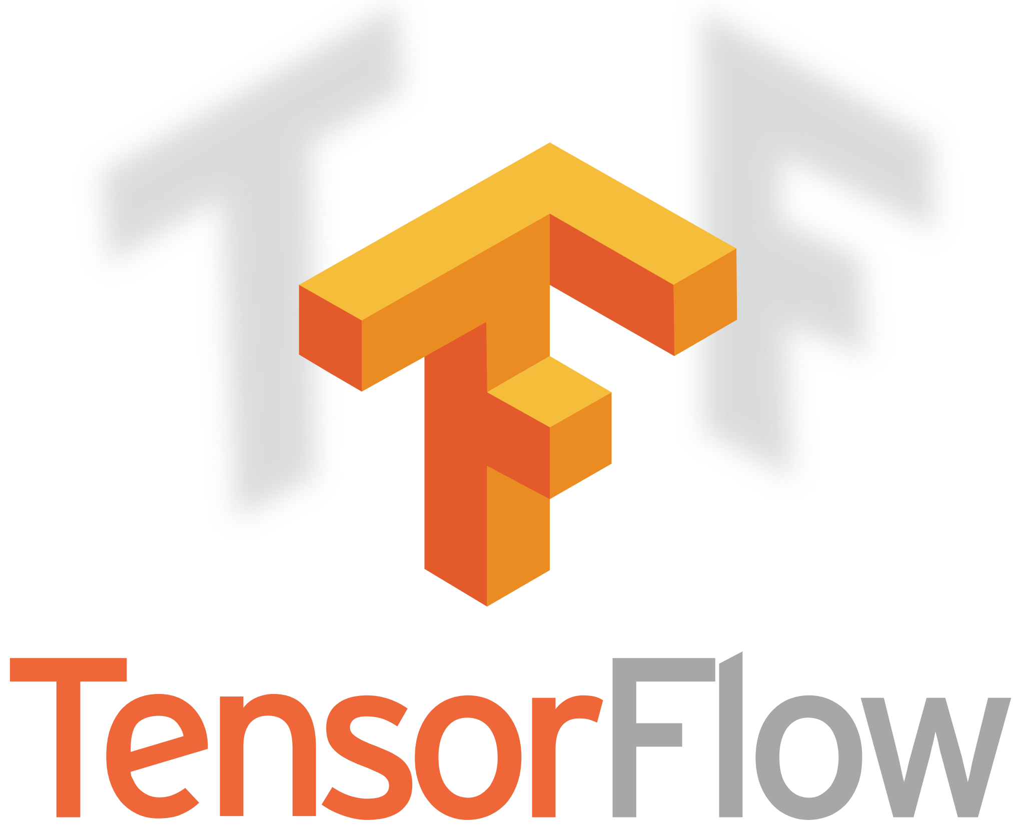
cuts → ?????

histogramming / analysis → ?????

In the next few slides I’ll try to motivate how having some kind of framework can make this jump much easier even if it is far from perfect.

Filling up the box with some new tools

In order to create an environment in which we could start moving forward we have written some of these tools using TensorFlow



Our goals:

- ☑ Exactly the same code base for CPU and GPU (no matter the brand!)
- ☑ A lot of mathematical functions already available
- ☑ Not adding extra dependencies to our existing codebase (mostly python and, yes, TF)
- ☑ Easily extensible and **interfaceable with other languages (C++, Cuda, Fortran, Rust)**

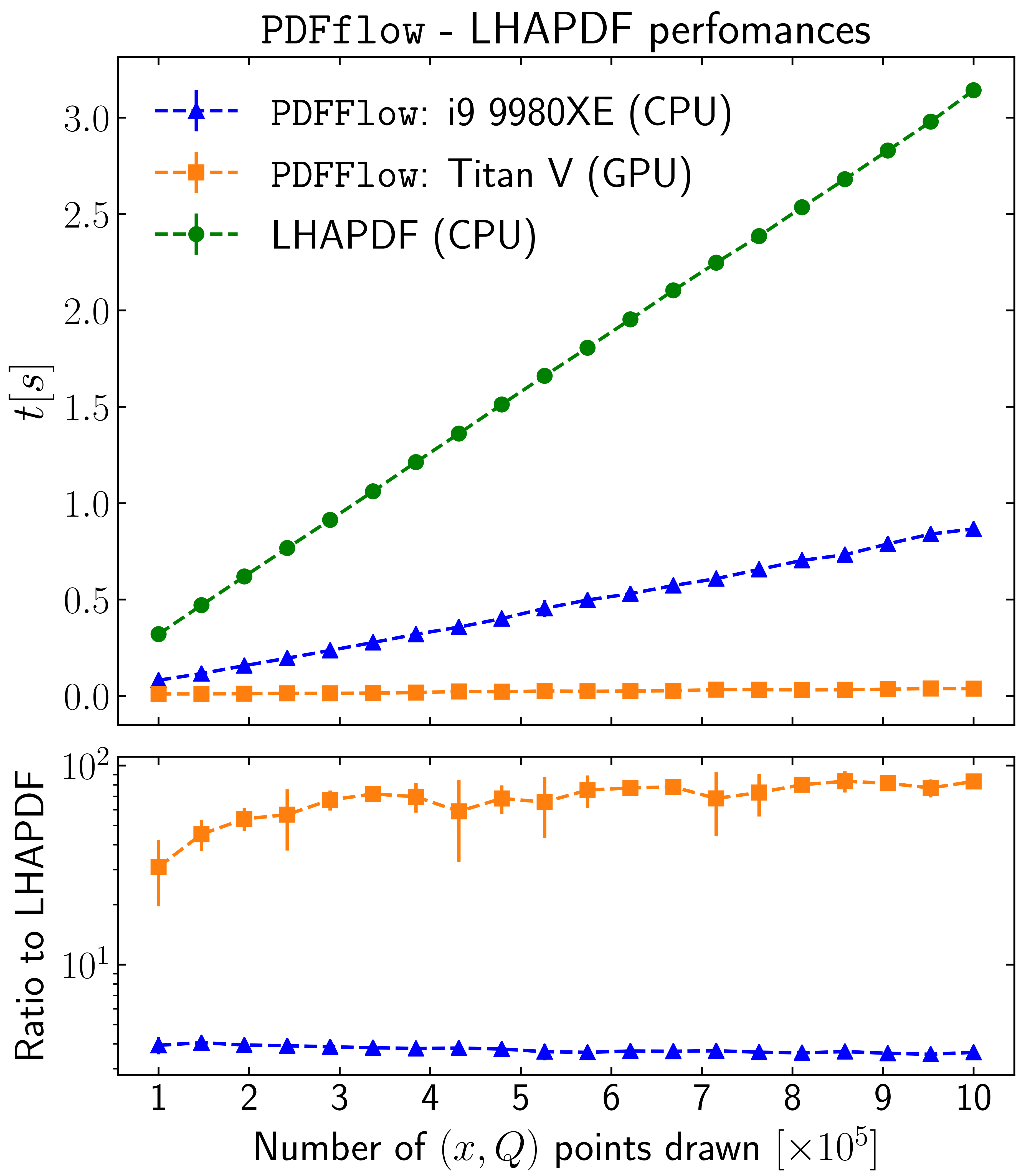
Some caveats and disadvantages:

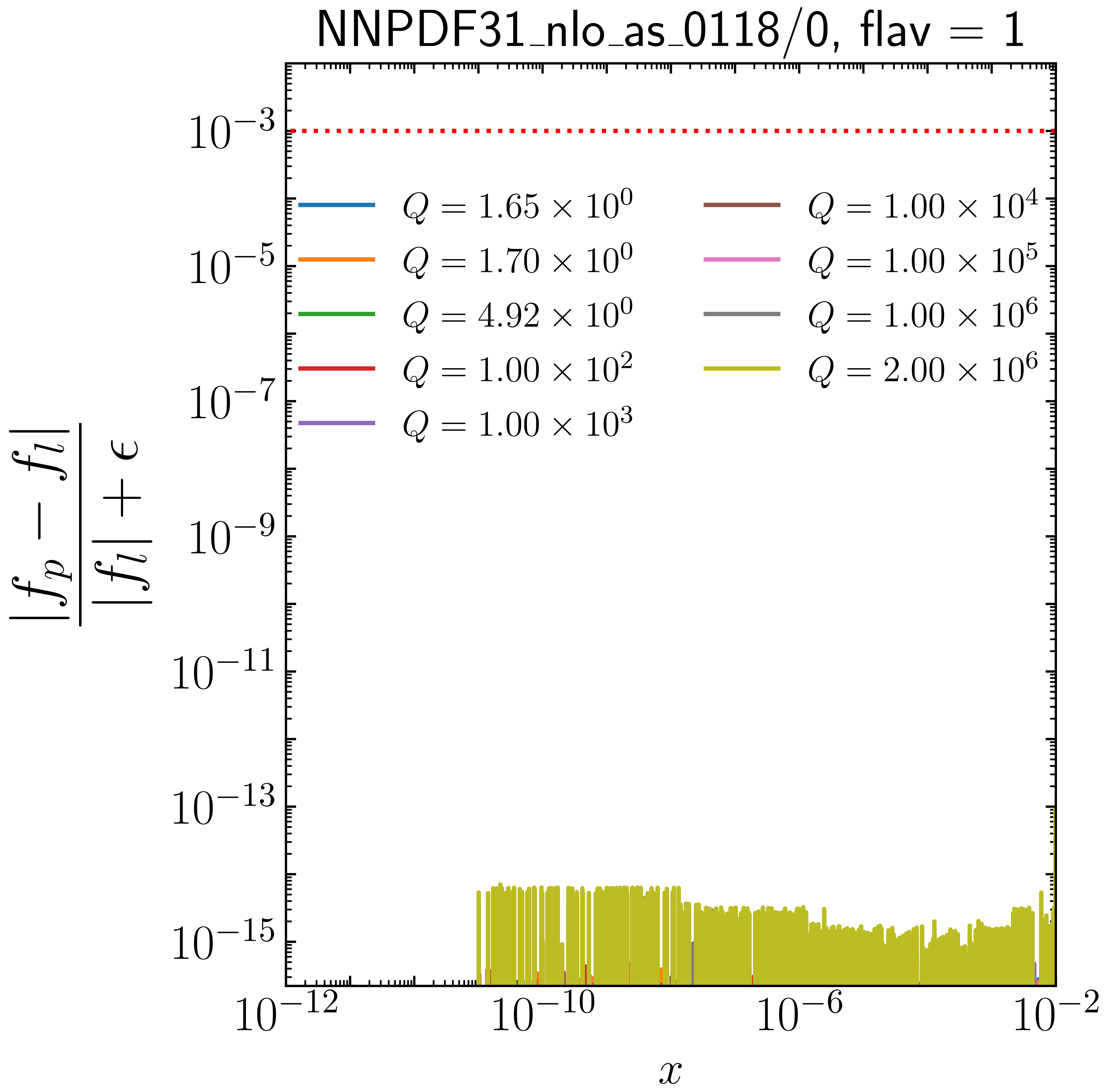
- □ It’s an external Machine Learning library, their goals are not always aligned with ours
- □ The above is most obvious on some overheads in (mostly) memory and execution time]
- □ The *easily* in “easily extensible” is subject to opinion

PDF Interpolation Library: PDFFlow

[hep-ph] 2009.06635

github.com/N3PDF/pdfflow



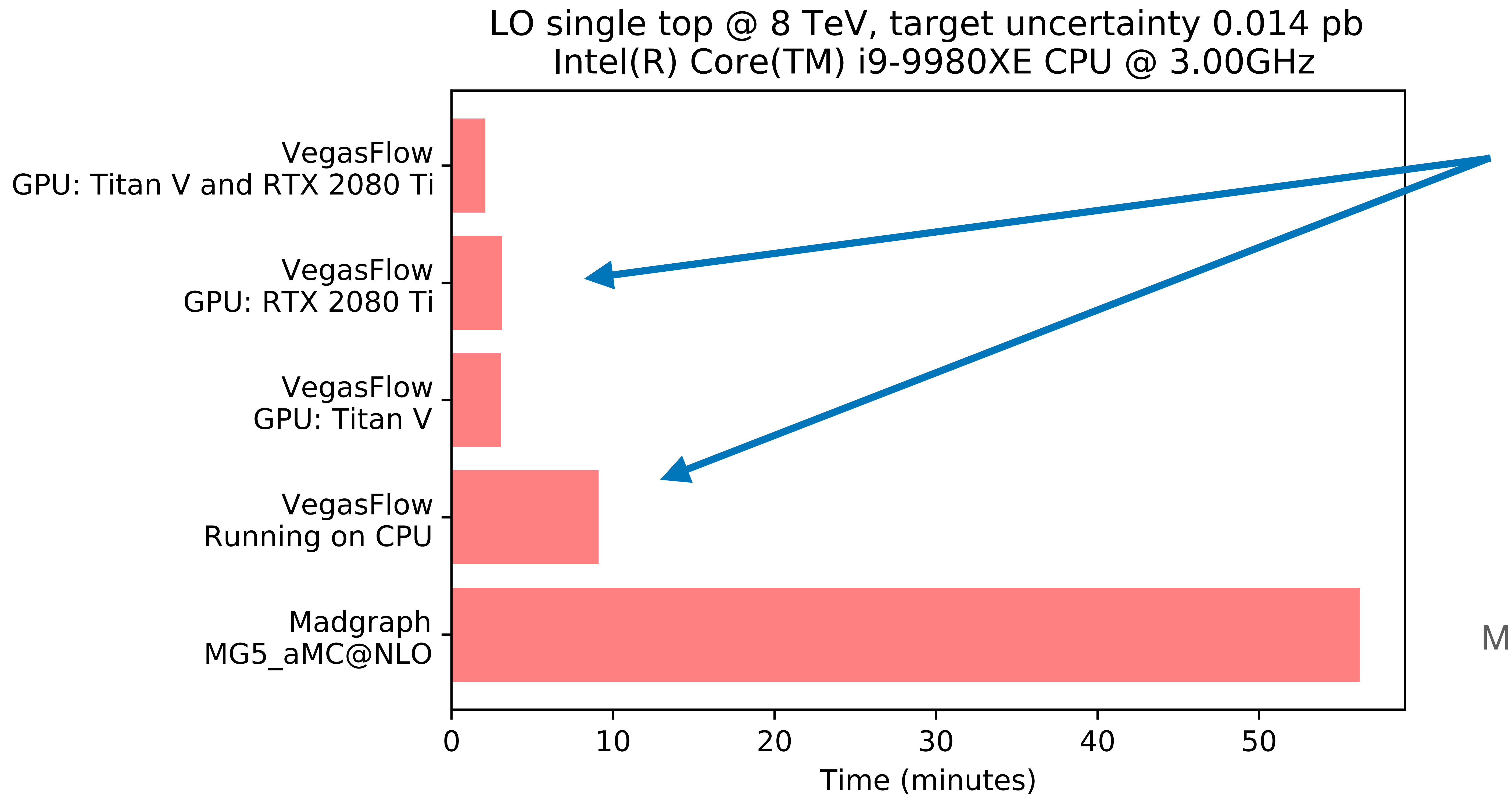


GPU-aware integration wrapper: VegasFlow

[hep-ph] 2002.12921 github.com/N3PDF/vegasflow

VegasFlow implements some widely used importance sampling algorithms and knows how to dispatch integrands to one (or multiple) GPUs.

The first real-life test we can do is a simple Leading Order process with easy expressions and a not too complicated phase space



At this point we have a framework which we can use to run in different kind of hardware with relatively little added effort.

We can start building from here!

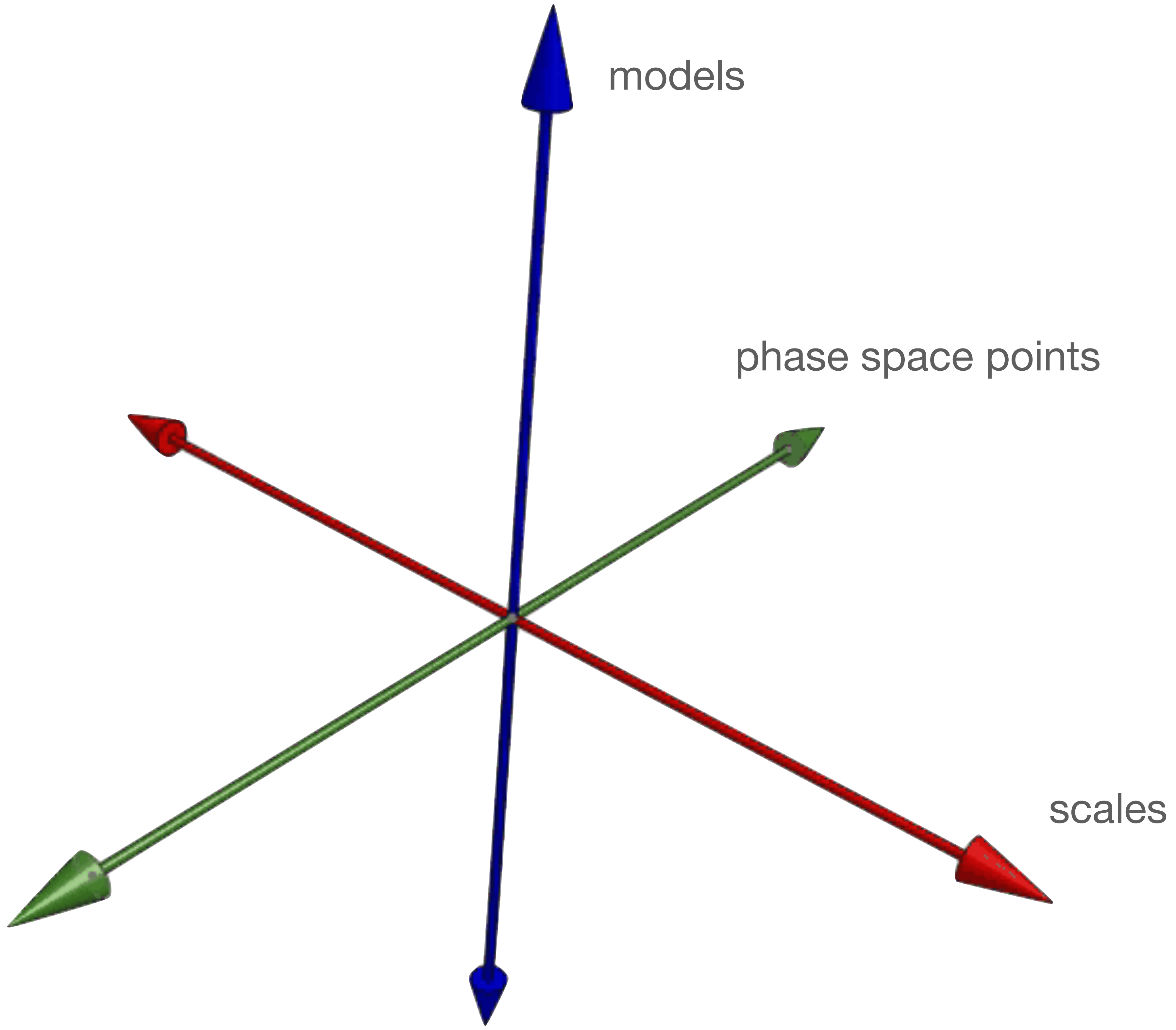
Madgraph timing here for reference, it’s not an apple-to-apples comparison

Parallelising outside the box

While we are often thinking about vectorising in the “event-axis”, that’s only one of the options

Maybe -for whatever reason- the strategy doesn’t allow for non-sequential running.

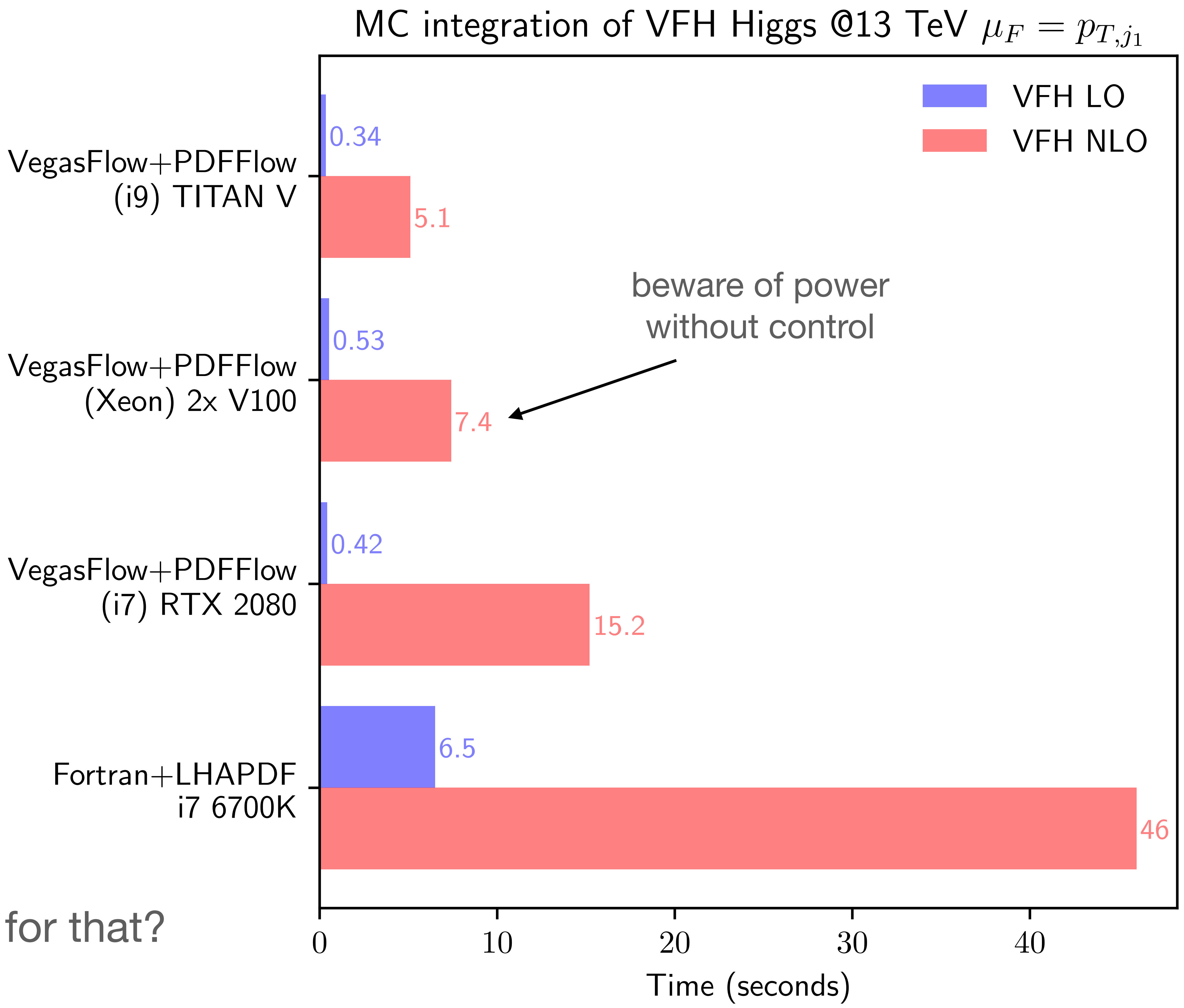
- Running different models at once
- The goal of the game is to keep as much of the calculation along the parallelisation axis constant (luminosity channels, for instance, are probably not a good candidate for this)
- Maybe thinking about a more general tensorization instead of vectorization



Beyond Leading Order

Let’s introduce a more realistic scenario: a NLO calculation with non trivial cuts and 4 particles in the final state

- Infrared subtraction are subtracted locally with antenna subtraction.
- The Phase Space was manually written with this process in mind.
- The whole batch of events is generated and cuts immediately applied before continuing the calculation.
- Phase Space points are then reorganized to eliminate any kind of branching in the more expensive parts of the calculation
- For this process, at NLO the cancellation of infrared divergences works very decently, otherwise care needs to be put on that as well.
- We kept an index of each event, its phase space and its weight in order to fill histograms at the end, in this case we were trading memory for performance

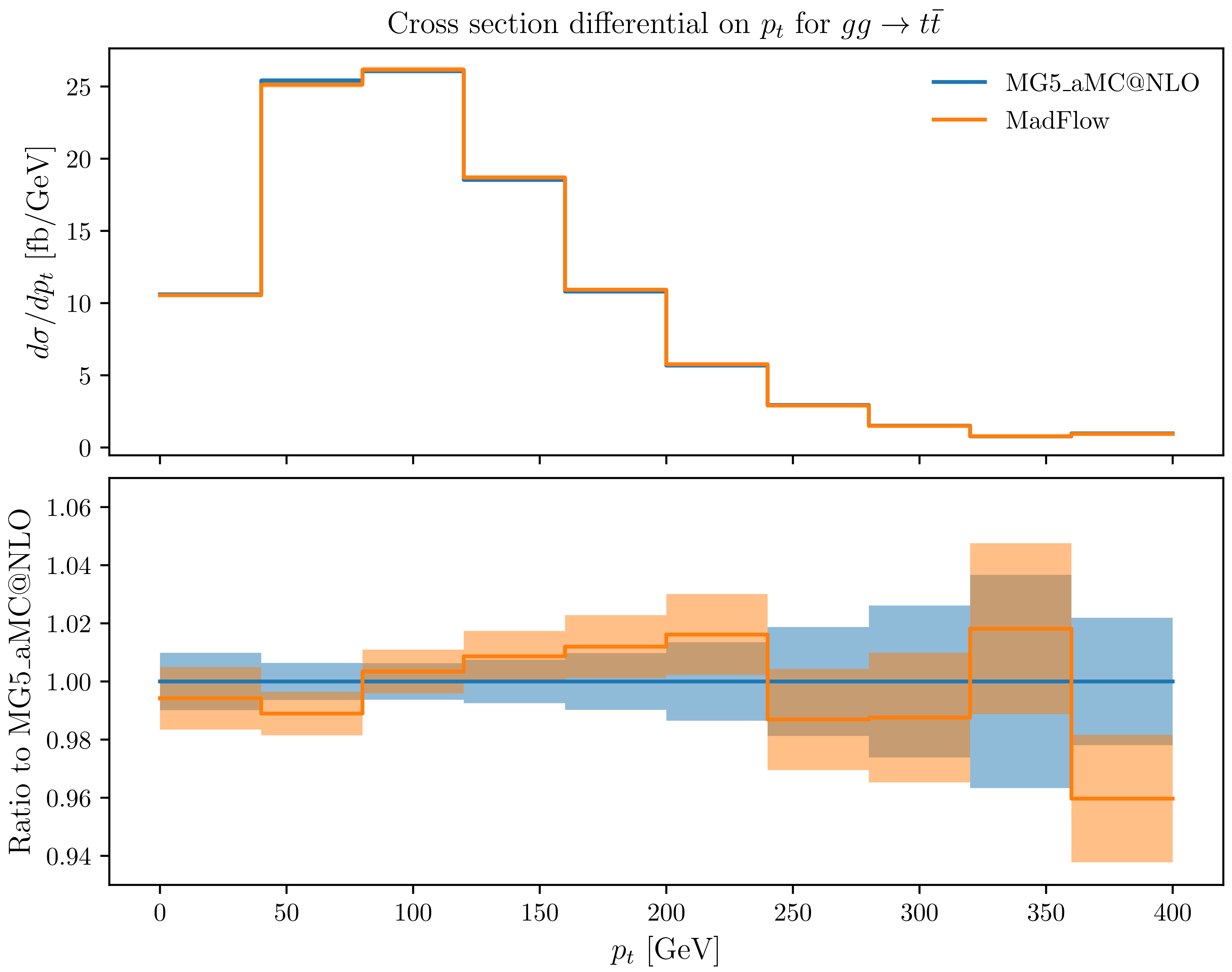


beware of power without control

Can we generalize this to any process? What would we need for that?

Beyond process-dependent code: MadFlow

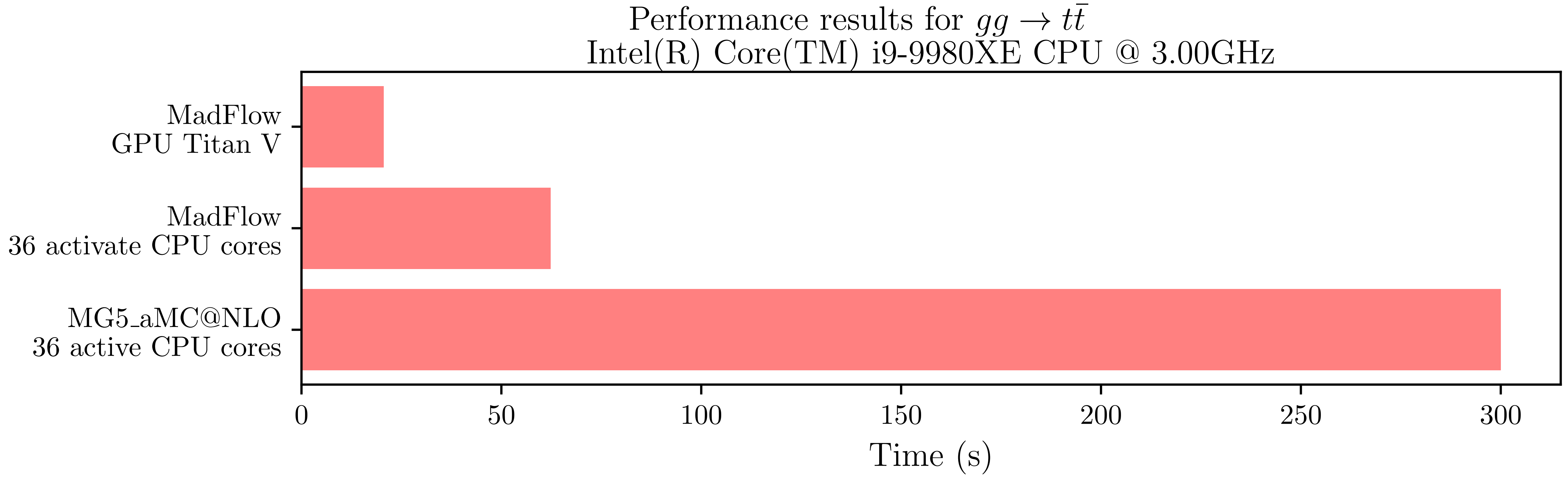
[hep-ph] 2106.10279 github.com/N3PDF/madflow



☑ Exploit MadGraph interface to write the diagrams in python, extended to write them in a vectorized way and using tensorflow-friendly routines

☑ Write a phase space generator that's completely general (vectorized version of Rambo). We gave it the very original name of RamboFlow.

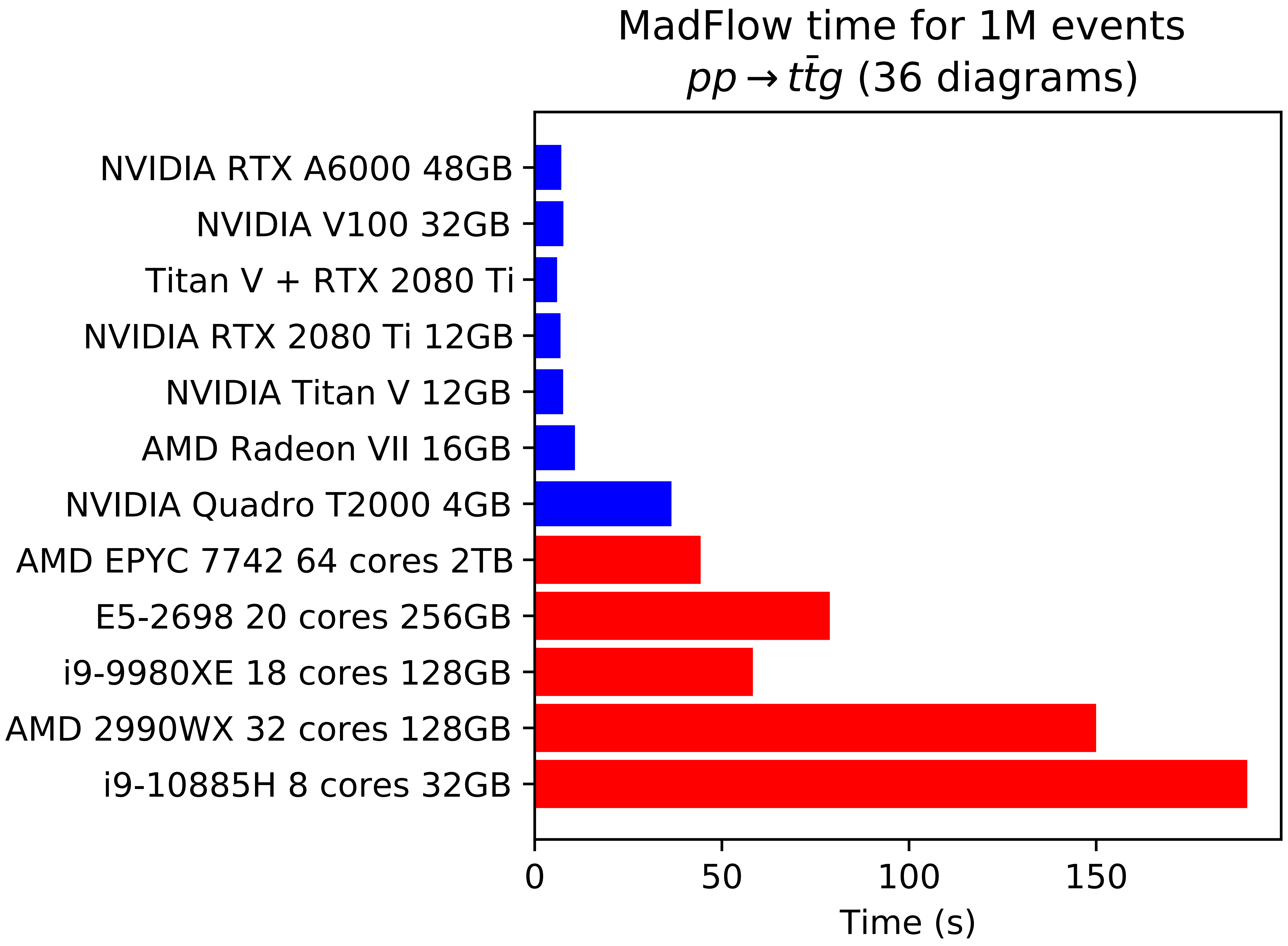
☑ We can then modify our previous example to use this Madgraph interface to *automagically* generate the matrix elements. Only at Leading Order.



Beyond process-dependent code: MadFlow

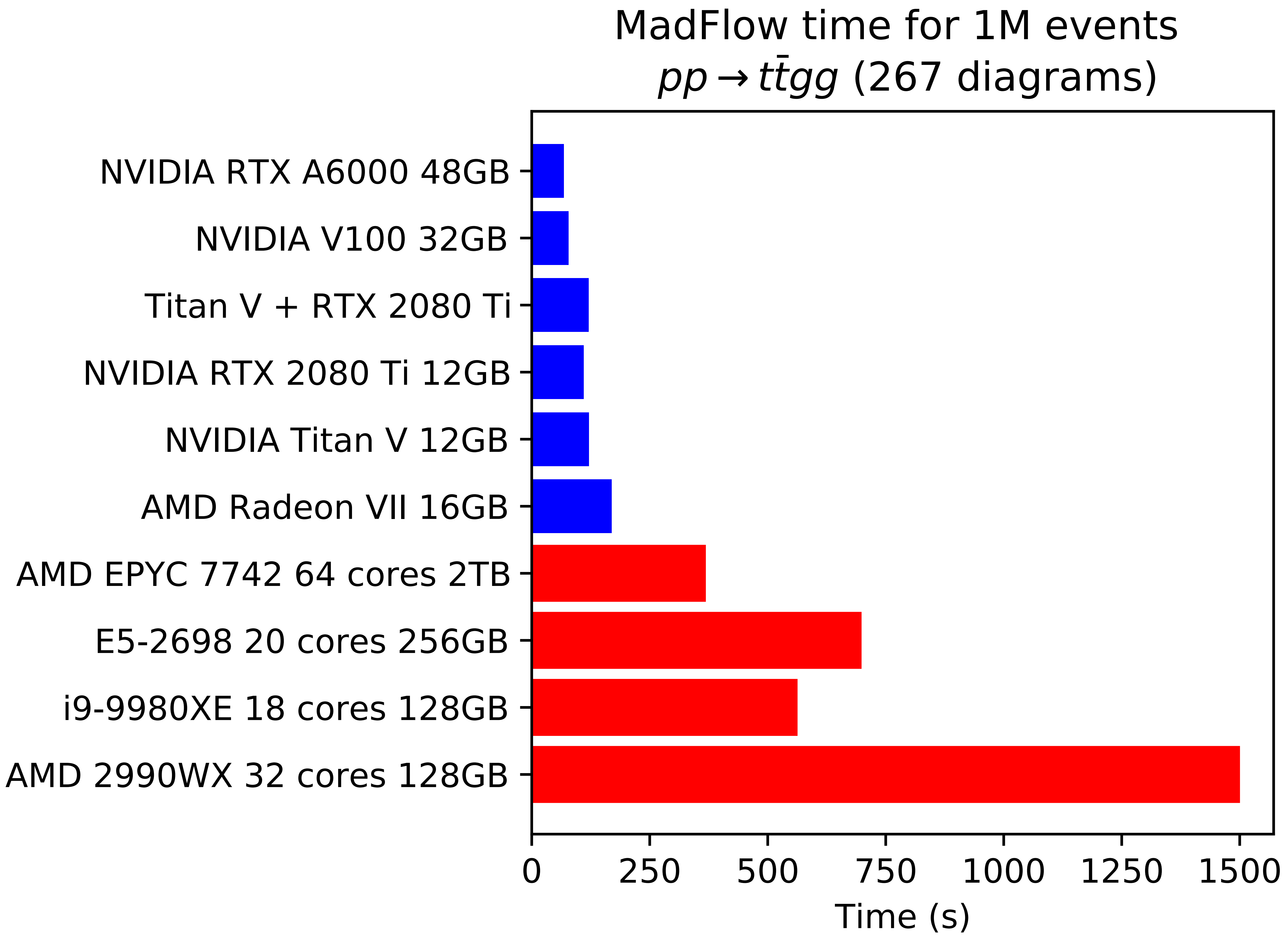
[hep-ph] 2106.10279

github.com/N3PDF/madflow



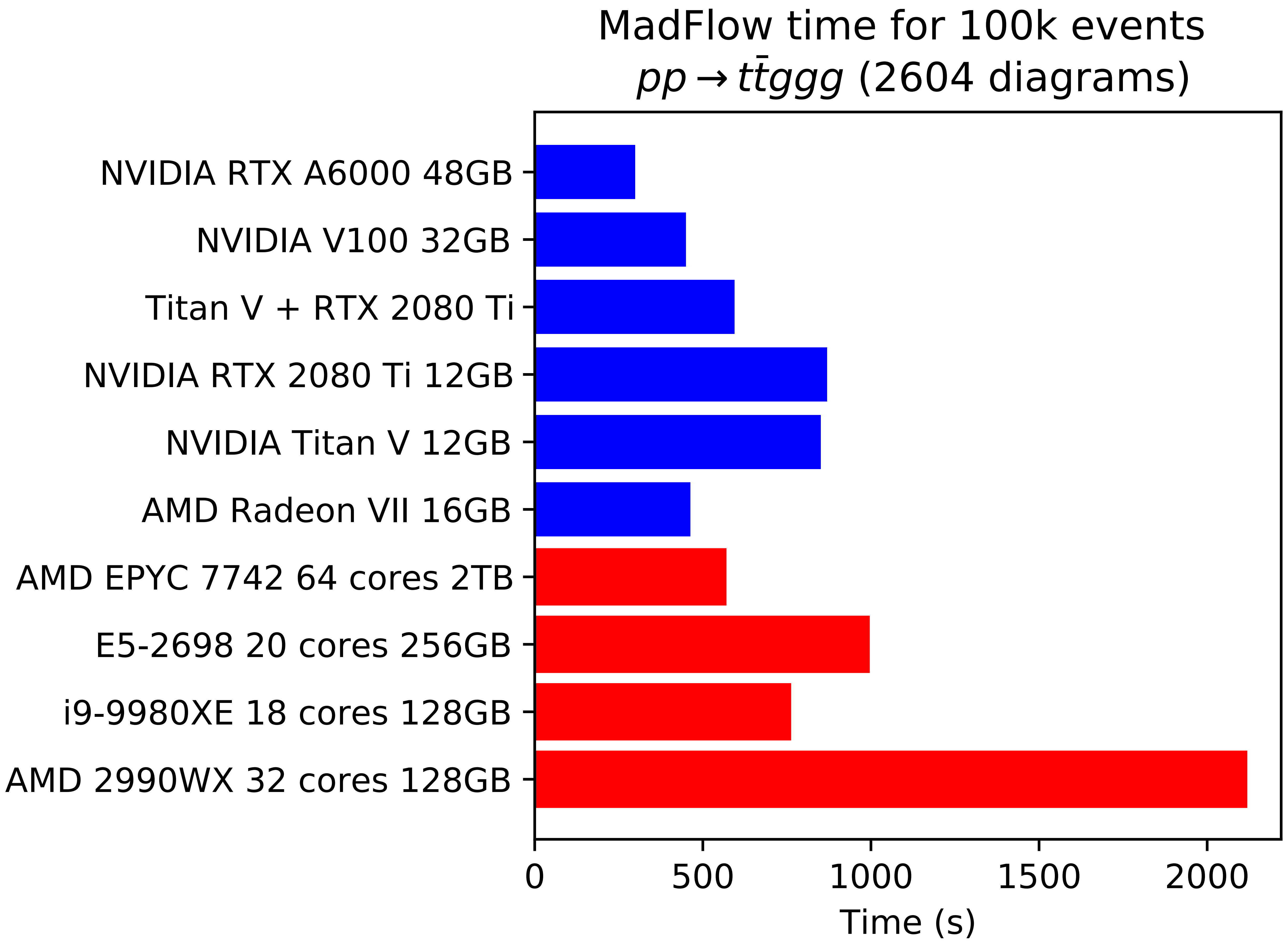
Beyond process-dependent code: MadFlow

[hep-ph] 2106.10279 github.com/N3PDF/madflow



Beyond process-dependent code: MadFlow

[hep-ph] 2106.10279 github.com/N3PDF/madflow



Beyond hardware agnostic code: overoptimization

Now we have to pay the debt that we bypassed at the beginning.

Until now we have been programming with tools that allowed us to use our existing codebase and that could run in any kind of hardware. As announced at the beginning, this introduced an overhead. This overhead is now explicitly visible.

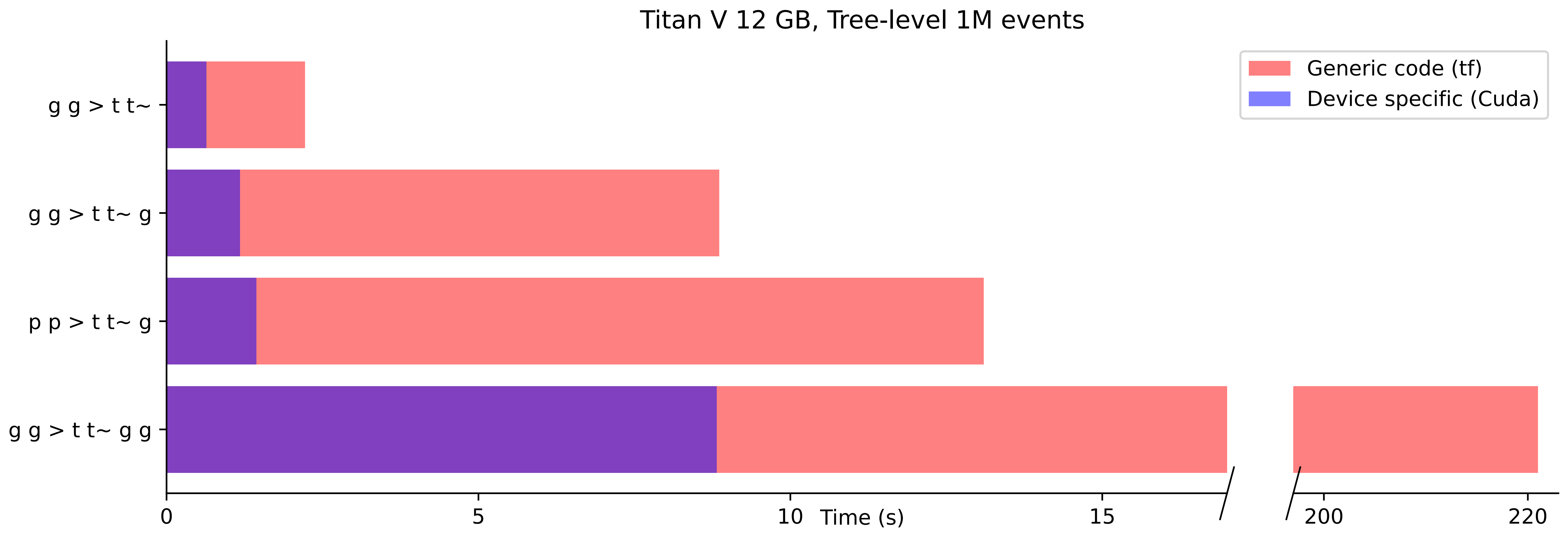
At the same time we can remember one of the advantages that we mentioned at the beginning

☑ Easily extensible and **interfaceable with other languages (C++, Cuda, Fortran, Rust)**

Let´s now use that power.

Since in this case the bottleneck is created by the sheer amount of diagrams, we can write a transpiler so that we can convert them in CUDA code that gets compiled before running the process.

The gains are mostly on memory, but that translates to a gain also in running time



Beyond hardware agnostic code: overoptimization

Now we have to pay the debt that we bypassed at the beginning.

Until now we have been programming with tools that allowed us to use our existing codebase and that could run in any kind of hardware. As announced at the beginning, this introduced an overhead. This overhead is now explicitly visible.

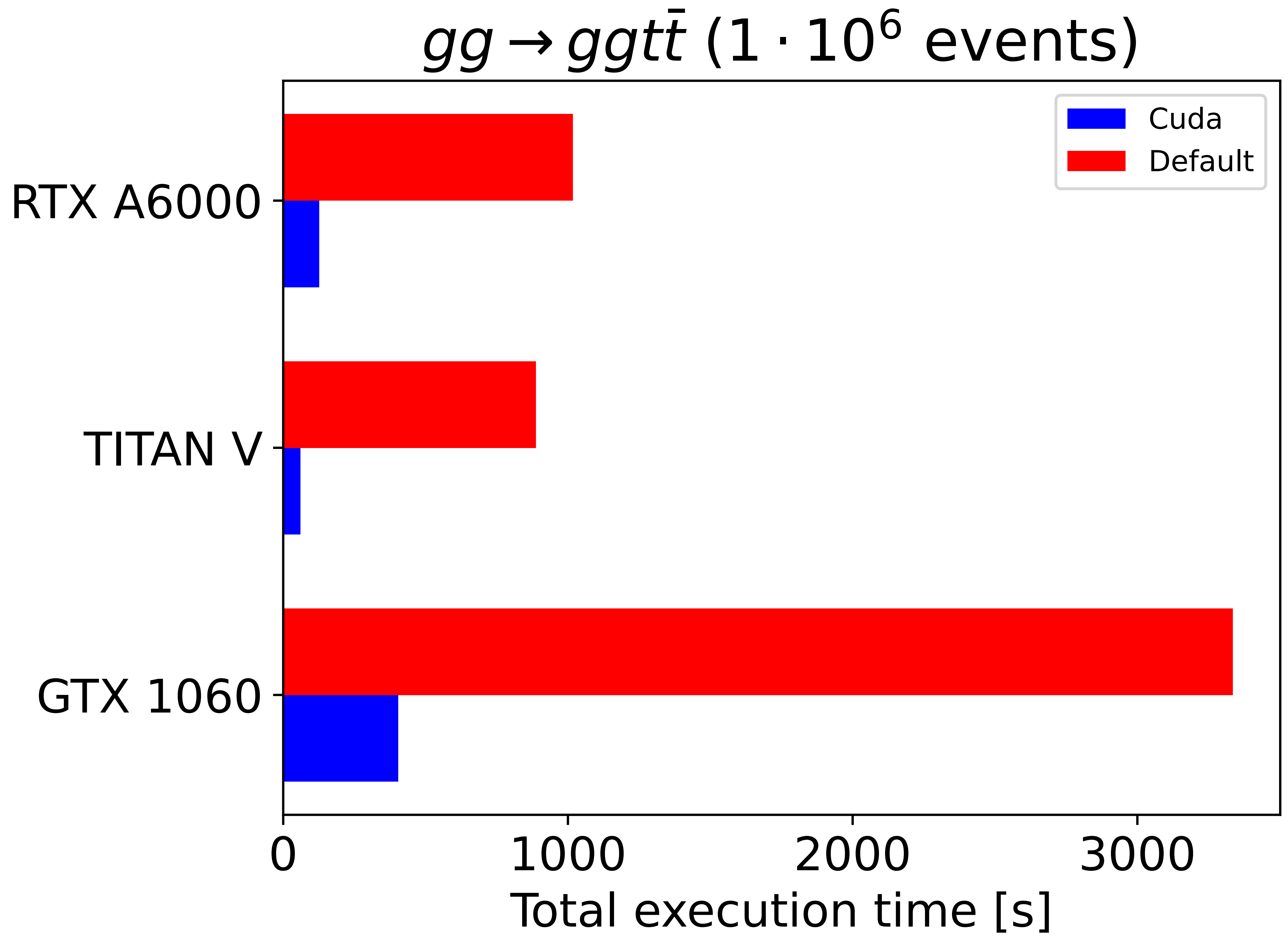
At the same time we can remember one of the advantages that we mentioned at the beginning

☑ Easily extensible and **interfaceable with other languages (C++, Cuda, Fortran, Rust)**

Let´s now use that power.

Since in this case the bottleneck is created by the sheer amount of diagrams, we can write a transpiler so that we can convert them in CUDA code that gets compiled before running the process.

The gains are mostly on memory, but that translates to a gain also in running time



$gg \rightarrow gg t\bar{t}$ $(1 \cdot 10^6$ events)

Queridos Reyes Magos…

- Full-fledged GPU Monte Carlo event generators required dedicated efforts
- Incremental improvements might not seem worth it…
- However… slow and steady wins the race

* Vectorize new implementations whenever possible, even if doesn't seem that useful
* Train students and future researchers on these new techonologies
* Release and share these tools and make them available!