MadFlow: towards the automation of Monte Carlo simulation on GPU for particle physics processes

Juan M Cruz-Martinez
in collaboration with: S. Carrazza, M. Rossi, M. Zaro

25th International Conference on Computing in High-Energy and Nuclear Physics
May 2021

European Research Council
Established by the European Commission

This project has received funding from the EU's Horizon 2020 research and innovation programme under grant agreement No 740006.

Outline

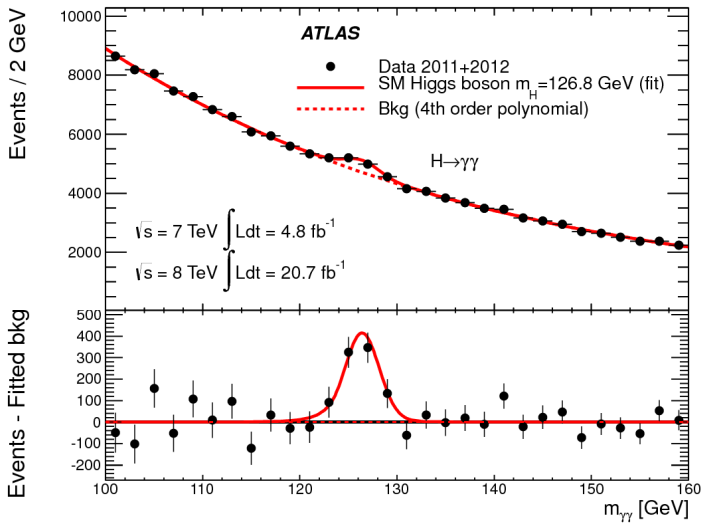
1. Motivation
   - Introduction
   - How can we do better
   - Tensors, tensors everywhere
2. The Flow suite: VegasFlow, PDFFlow, and MadFlow
   - The what, the where and the how
3. Benchmarks and examples
   - PDF interpolation
   - Automatic cross section integration
   - How to
4. Conclusions

Parton-level Monte Carlo generators

Behind most predictions for LHC phenomenology lies the numerical computation of the following integral:

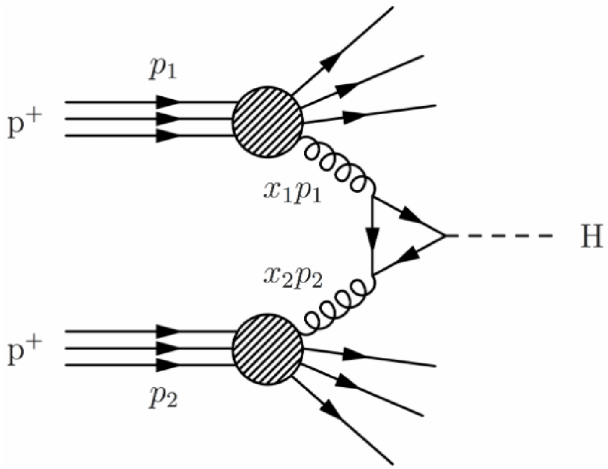
$$\int \mathrm{d}x_1\,\mathrm{d}x_2\, f_1(x_1, q^2) f_2(x_2, q^2) |M(\{p_n\})|^2 \mathcal{J}_m^n(\{p_n\})$$

- $\rightarrow$ $f(x, q)$: Parton Distribution Function
- $\rightarrow$ $|M|$: Matrix element of the process
- $\rightarrow$ $\{p_n\}$: Phase space for n particles.
- $\rightarrow$ $\mathcal{J}$: Jet function for n particles to m.

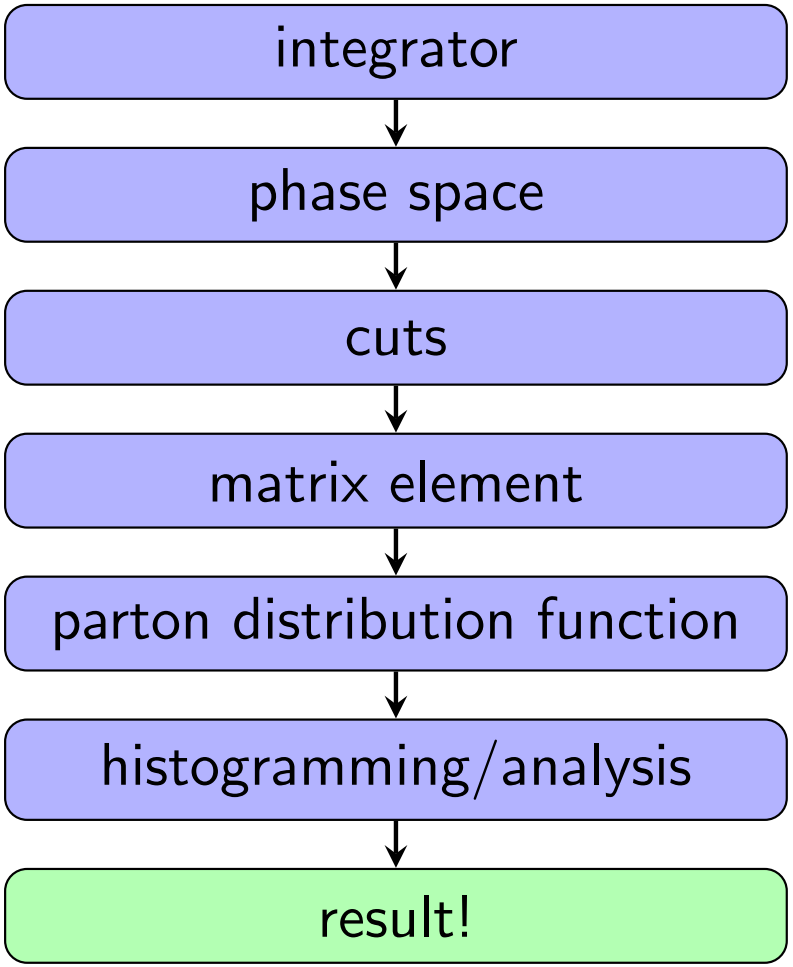


Parton-level Monte Carlo generators ingredients:

$$\int \mathrm{d}x_1\,\mathrm{d}x_2\, f_1(x_1, q^2) f_2(x_2, q^2) |M(\{p_n\})|^2 \mathcal{J}_m^n(\{p_n\})$$

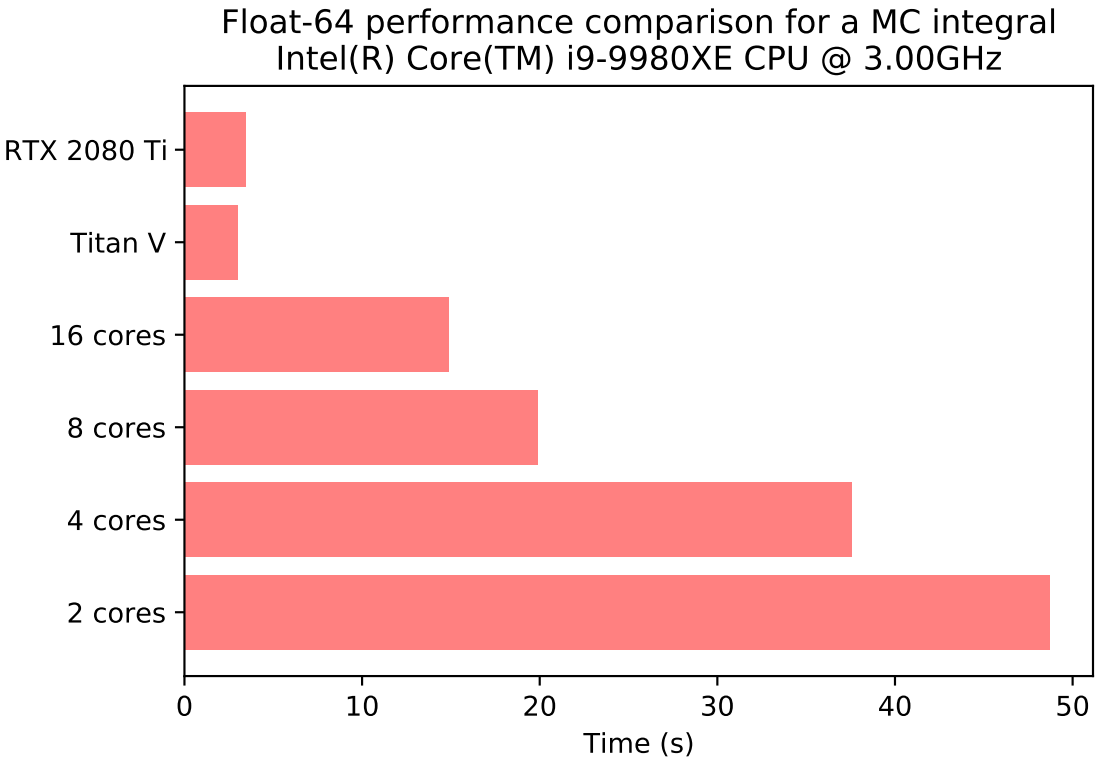


The integrals are usually computed numerically using CPU-expensive Monte Carlo generators.



GPU computing

Monte Carlo simulations are highly parallelizable, which make them a great target for GPU computation.



Quick Example:
n-dimensional gaussian function

$$I = \int \mathrm{d}x_1 \dots \mathrm{d}x_n \, e^{x_1^2 + \dots + x_n^2}$$

Every event is independent of all other events!

GPU computation can increase the performance of the integrator by more than an order of magnitude.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Why is then GPU computing not more widespread in HEP?

Most of the more advance theoretical calculations still rely exclusively on CPU.

✗ Diminishing returns
- Huge CPU-optimized Fortran 77/90 or C++ codebases.
- Publication-ready results are easily obtained expanding existing code.
- It's catch-22: porting the code becomes more and more complicated.

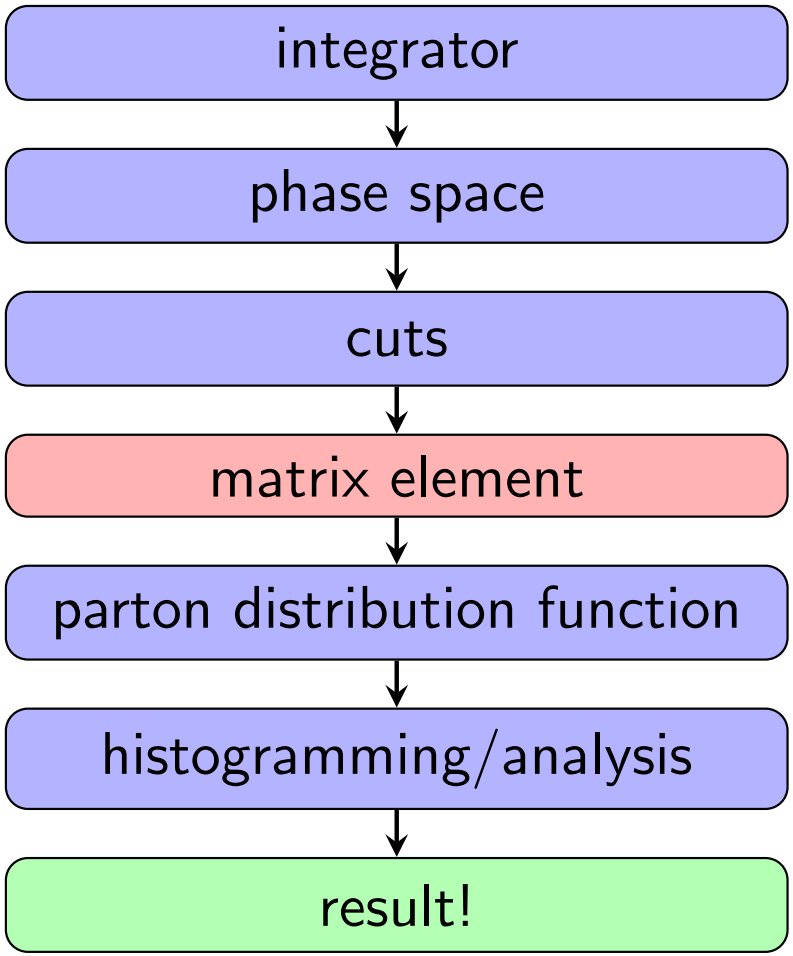
✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

✗ Lack of tools
- Many ready-made tools for CPU.
- GPUs are still decades behind in the hep-ph world.

Lack of Tools

Running on a CPU:

Worry only about what you are interested in. For instance, if we want an NNLO computation for $H \to j$ and we have a $Z \to j$ computation we only need to change the matrix elements.



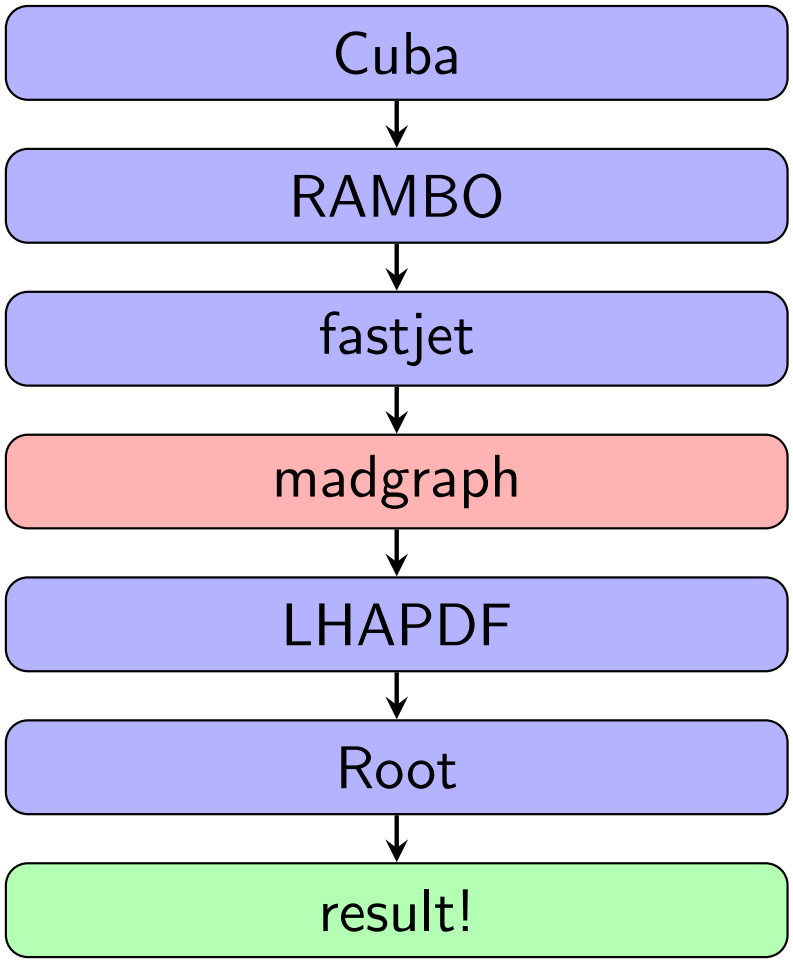
Lack of Tools

Running on a CPU:

Even if we don't already have some obscure and private fortran-based framework already built, there exists a complete tool set for producing results.

- ✓ PDF providers
- ✓ Phase space generators
- ✓ Integrator libraries...

some of which can still provide that sweet 70s' Fortran taste

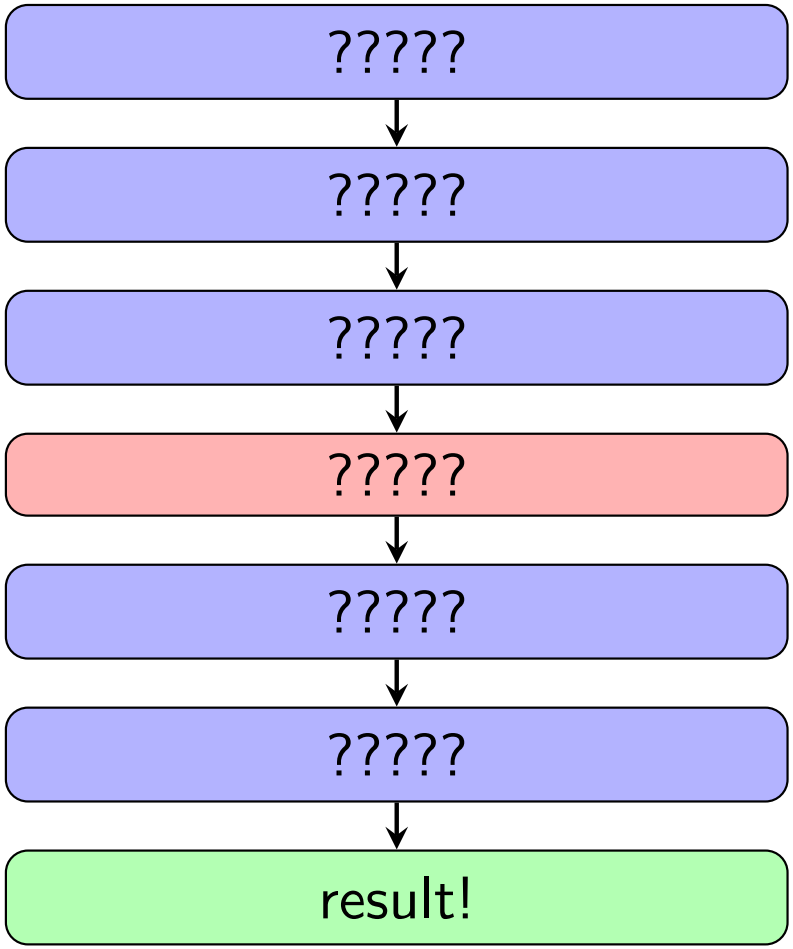


Lack of Tools

Running on a GPU:

There is no such tool set yet

so it needs to be written from scratch



Filling up the box

1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points space points.

Filling up the box

1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points space points.

2. A PDF interpolation tool to generate luminosities in parallel for many events ✓

Filling up the box

1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points space points.

2. A PDF interpolation tool to generate luminosities in parallel for many events ✓

3. An integrator framework able to send and receive batches to and from the GPU ✓

Filling up the box

1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points space points.

2. A PDF interpolation tool to generate luminosities in parallel for many events ✓

3. An integrator framework able to send and receive batches to and from the GPU ✓

4. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase (tree ✓, loop tbd)

Filling up the box

1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points space points.

2. A PDF interpolation tool to generate luminosities in parallel for many events ✓

3. An integrator framework able to send and receive batches to and from the GPU ✓

4. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase (tree ✓, loop tbd)

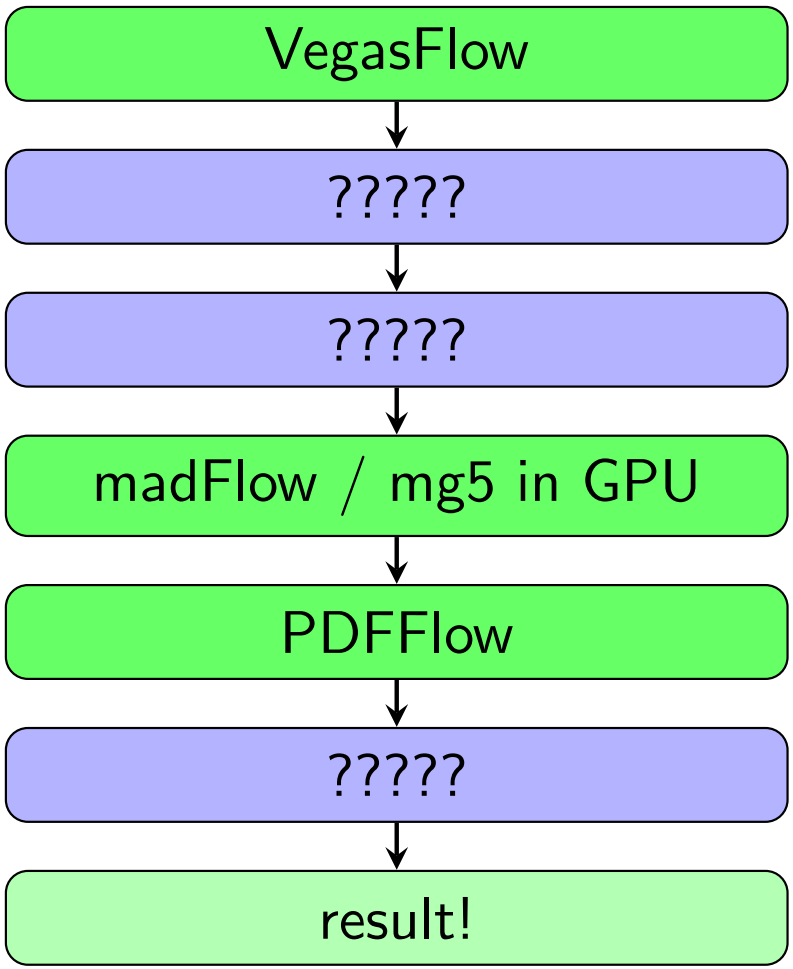
5. Analysis tools, experiment simulation, jet algorithms...

Filling up the box: moving with the flow

The flow suite focus on speed and efficiency for both the computer and the developer

- Python and TF based engine
- Compatible with other languages: Cuda, C++
- Seamless CPU and GPU computation out of the box
- Easily interfaceable with NN-based integrators

Source code available at:
`github.com/N3PDF/VegasFlow`
`github.com/N3PDF/PDFFlow`

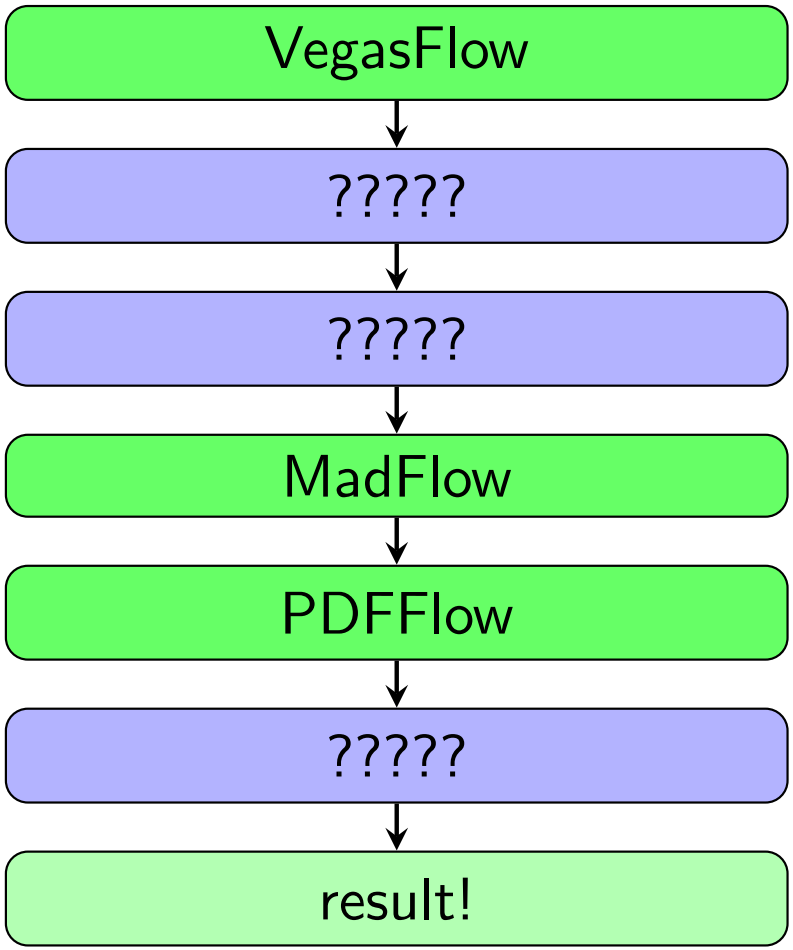


Interface with Madgraph's matrix generation

As a first step towards a full parton-level fixed-order Monte Carlo generator we interface the "flow" suite with Madgraph's matrix element generator.

We take advantage of ALOHA to produce tensorized versions of the matrix elements that can be efficiently run in GPU.

We aim to be modular enough that different ME providers can be used or even combined (such as Madgraph's CUDA output, see Andrea's talk from 2 hours ago!)

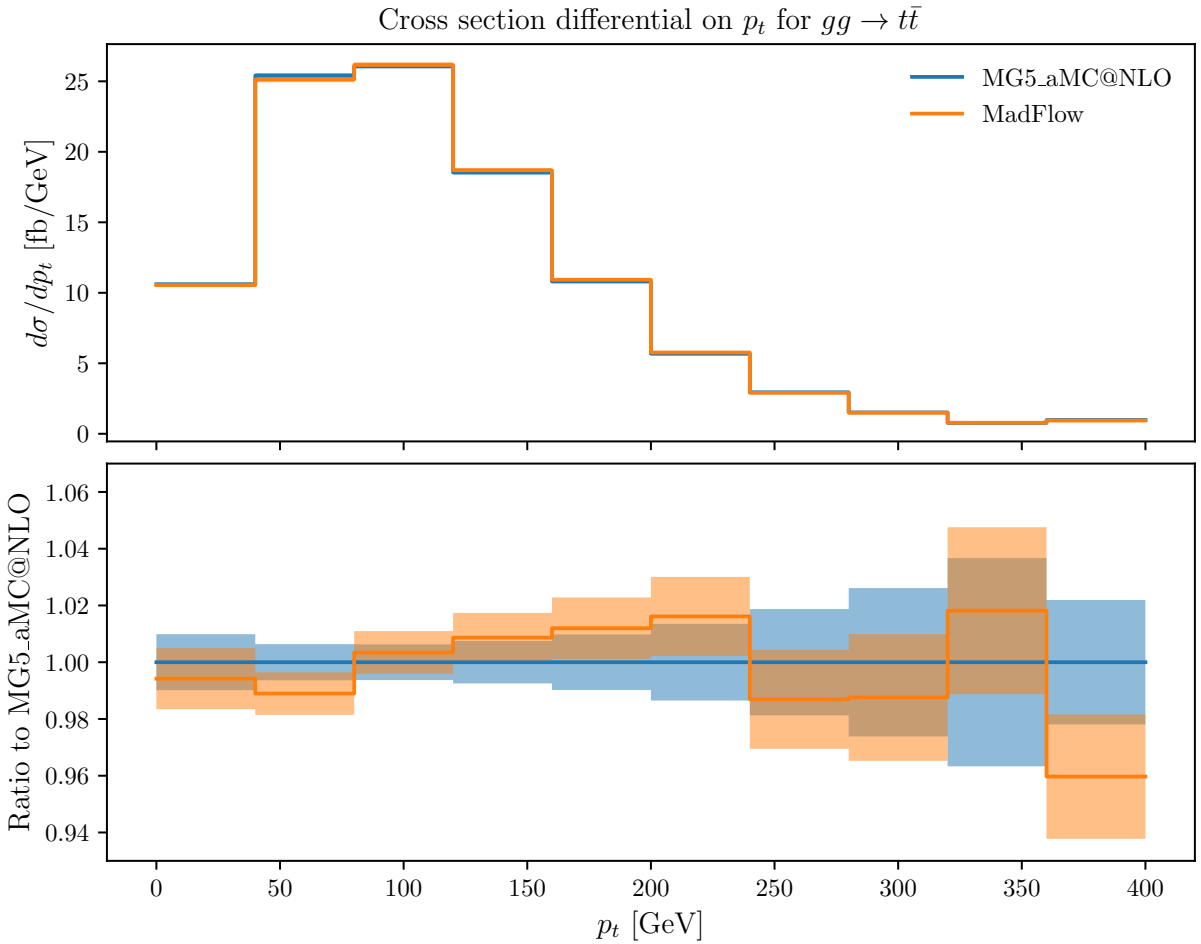


Preliminary Results

Exact same ME and feynman diagrams ✓

RamboFlow phase space ✗

Perfect compatibility ✓

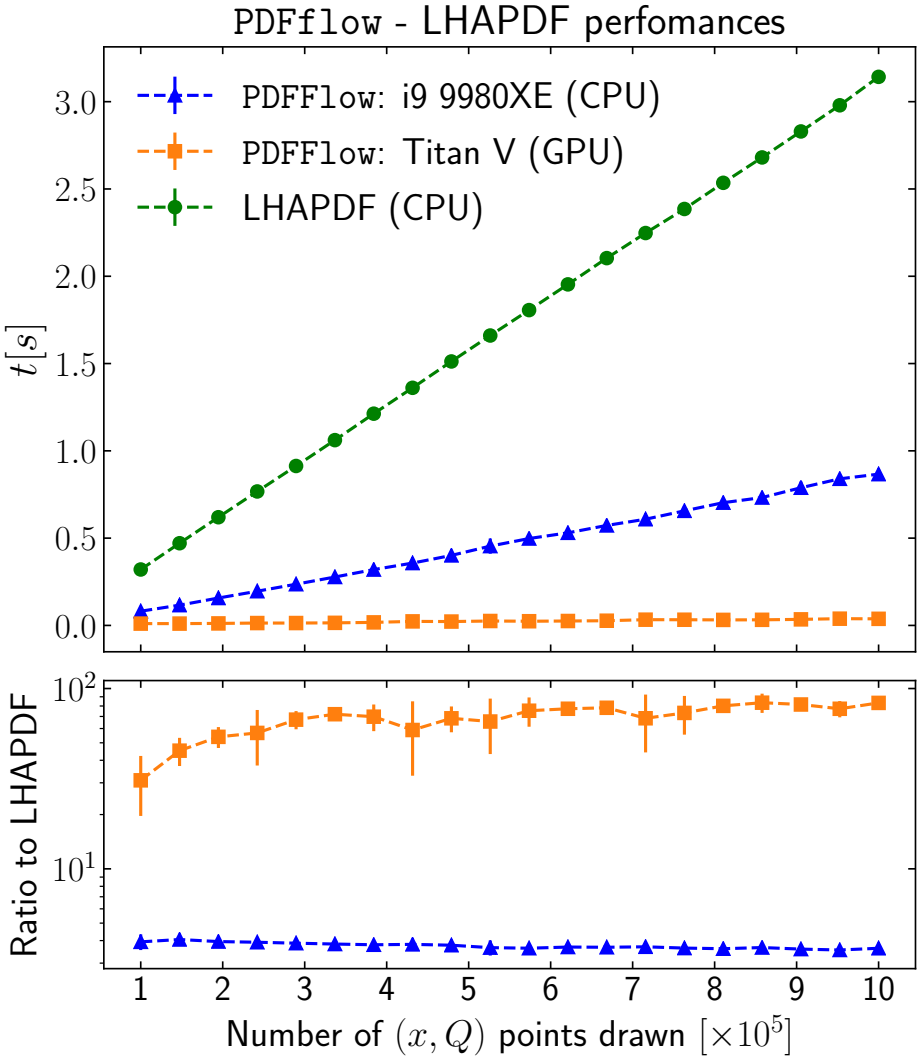


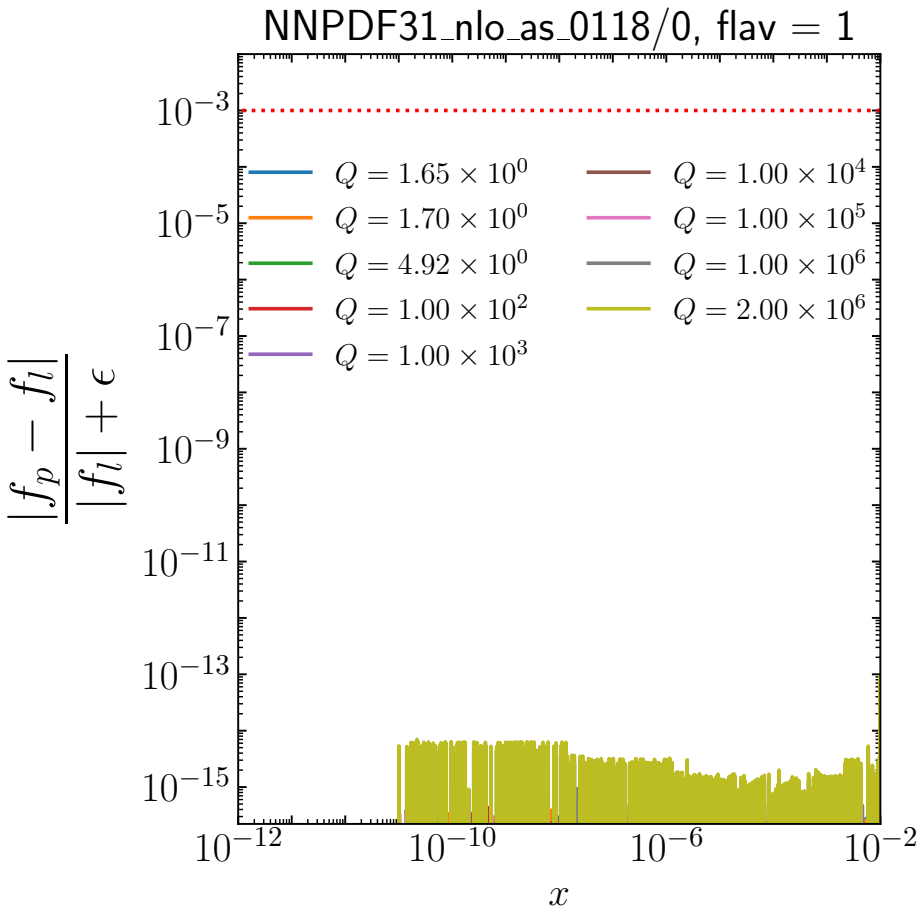
Benchmarks and examples

To wrap it up, we will see some examples and benchmarks that show how the parallelization (and tensorization!) of calculations can speed them up enormously.

- ✓ Parallel PDF interpolation
- ✓ A completely automatic LO calculation, CPU vs GPU
- ✓ Generation of unweighted events

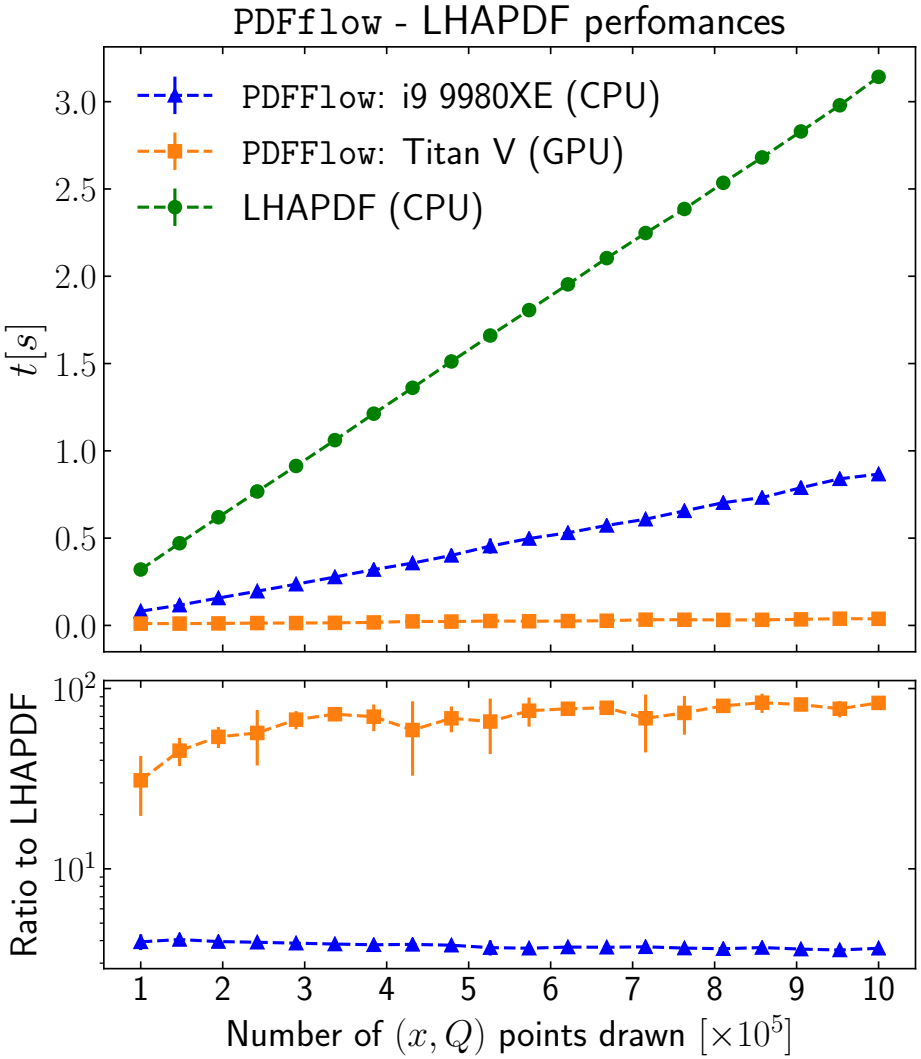
LHAPDF vs PDFFlow

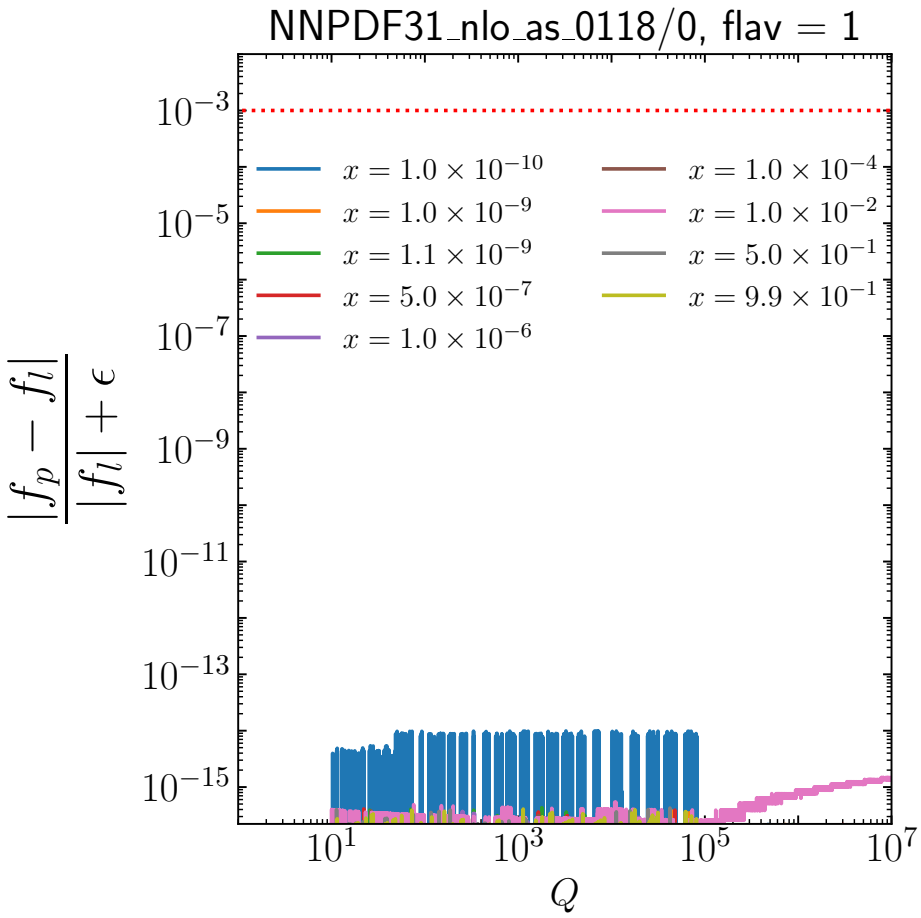




Interpolation in x for fixed q.

LHAPDF vs PDFFlow

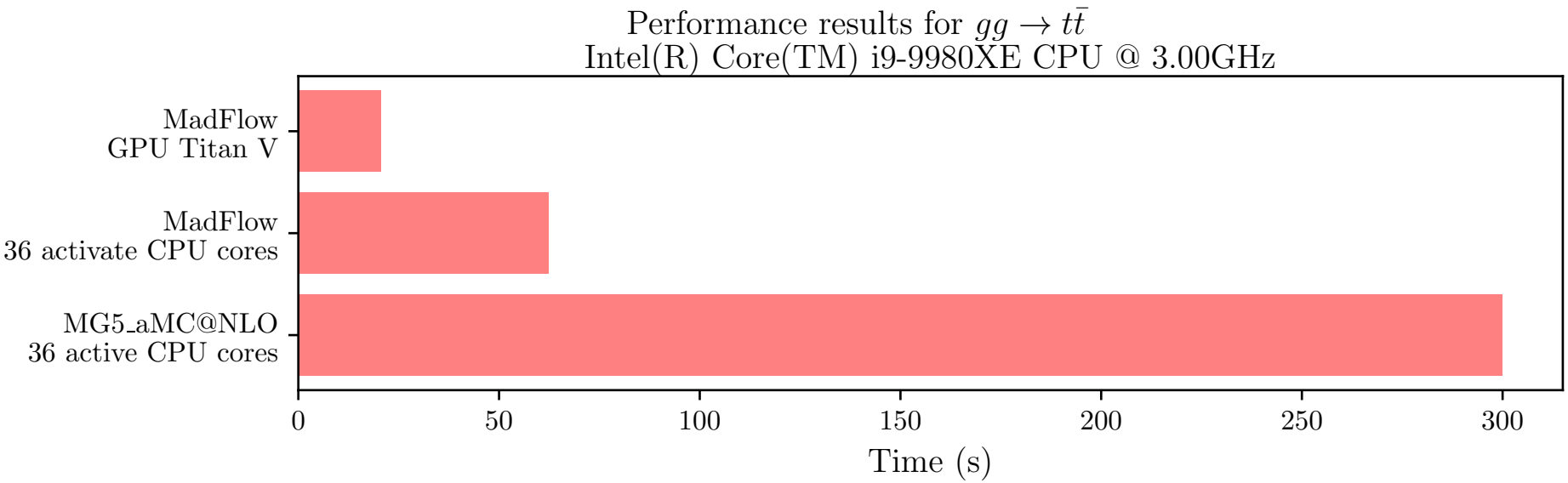




Interpolation in q for fixed x.

MadFlow Vs plain Madgraph LO

Figure: Plain Madgraph Vs MadFlow VegasFlow-based implementation



- ✓ PhaseSpace for VegasFlow: a GPU version of RAMBO
- ✓ There's even room for improvement if a clever phase space were to be used!

Open source for HEP

Where to obtain the code

The entire Flow suite is open source and can be found at the N3PDF organization repository `github.com:N3PDF`

How to install

VegasFlow and PDFFlow can be installed from the repository or directly with `pip`:

```
~$ pip install vegasflow pdfflow
```

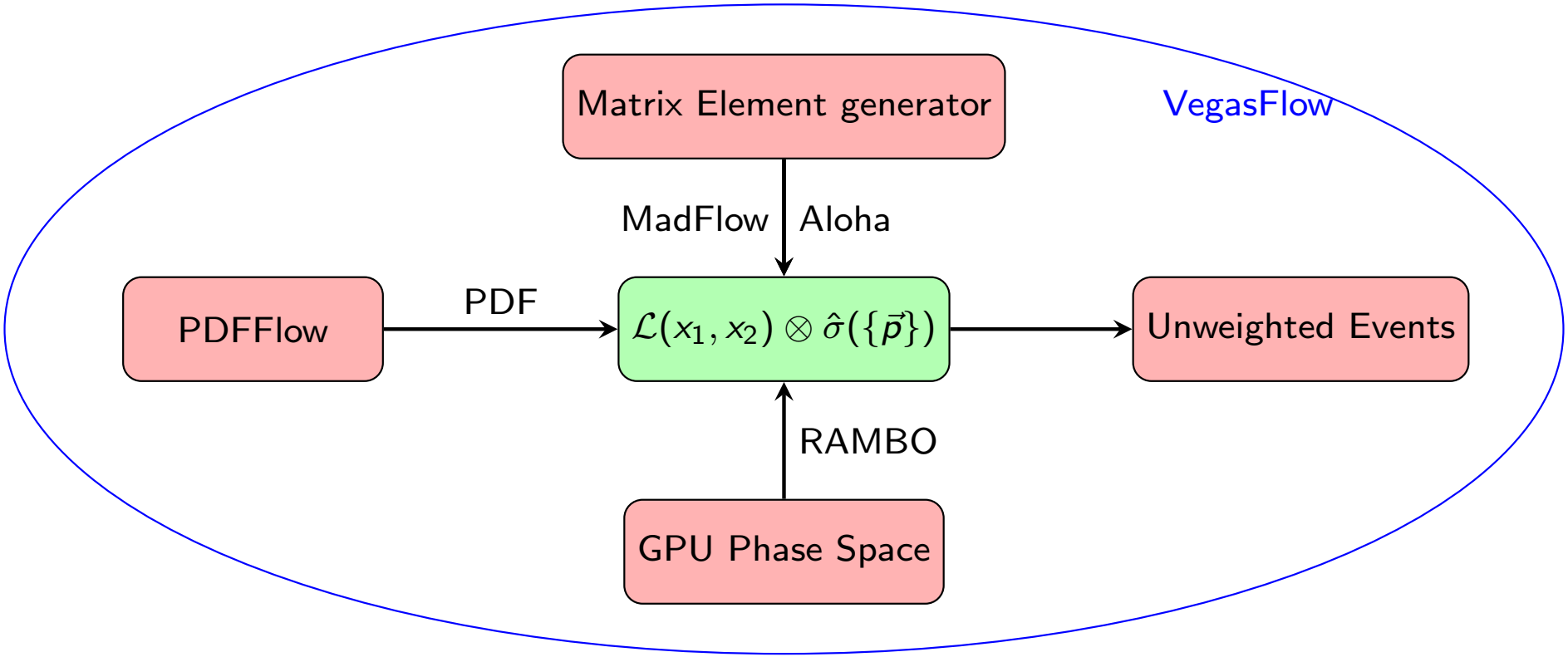
Documentation

The documentation for these tools is accessible at:
VegasFlow: `vegasflow.rtfd.io`
PDFFlow: `pdfflow.rtfd.io`

The MadFlow prototype

To be released

We expect to release a working prototype for "MadFlow" with unweighted events by the Offshell conference in July



Summary

- GPU computation is increasingly gaining traction in many areas of science but it is still not heavily used in particle physics phenomenology.

→ Is competitive with CPU for MC simulations.

→ A lot of effort on GPU-based computations.

✓ VegasFlow, PDFFlow and MadFlow provide a framework to run in any device.

✓ Generate all the different pieces (ME, PS, PDFs, integration algorithm) needed for fixed order calculations.

To be released

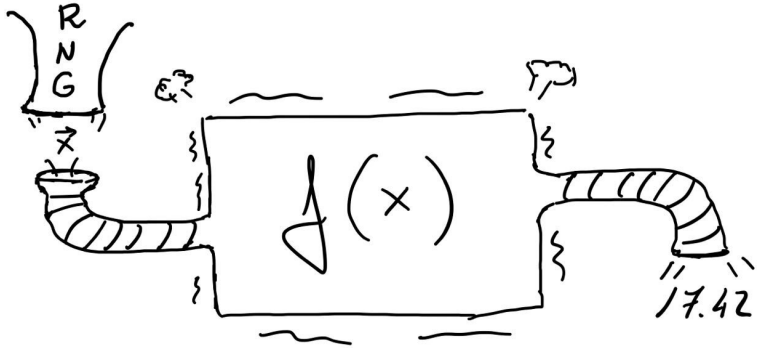
Working prototype to be ready in the next few months (we aim to release by July Offshell conference)

Thanks!

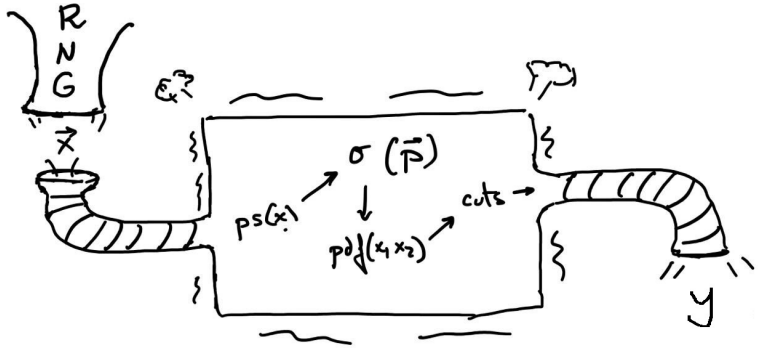
Act in parallel: CPU

The way we do Monte Carlo calculations in CPU already allows for a certain degree of parallelization

$$I = \frac{1}{N} \sum f(\vec{x}_i)$$



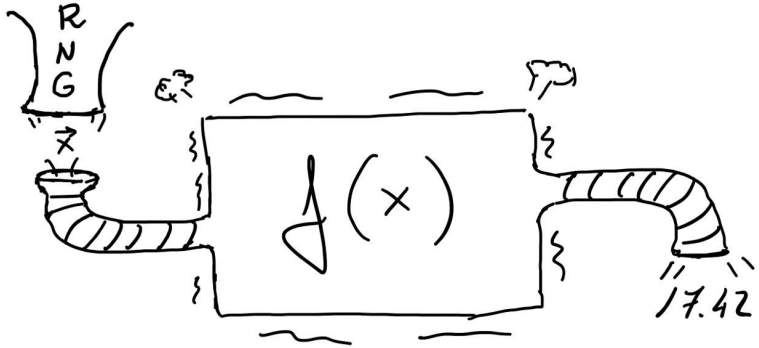
(the function $f(\vec{x})$ might be arbitrarily complicated)



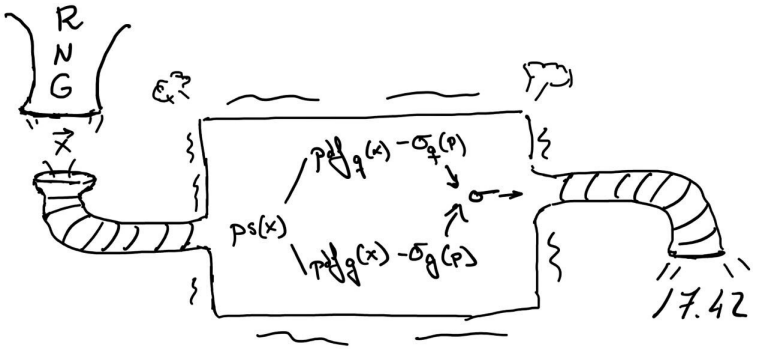
Act in parallel: CPU

The way we do Monte Carlo calculations in CPU already allows for a certain degree of parallelization

$$I = \frac{1}{N} \sum f(\vec{x}_i)$$

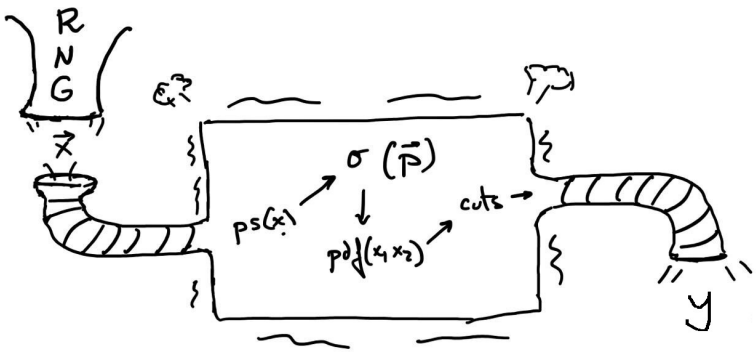


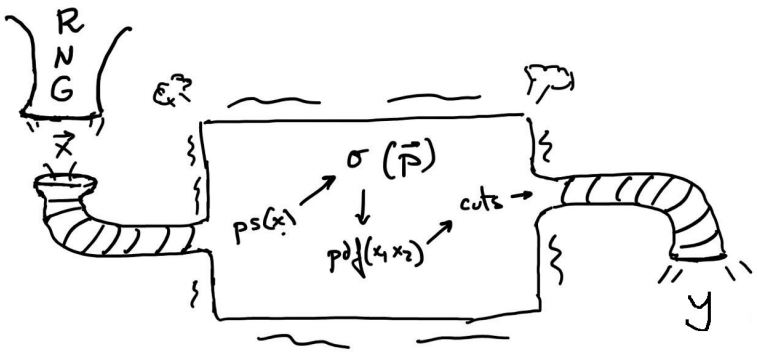
(the function $f(\vec{x})$ might be arbitrarily complicated)

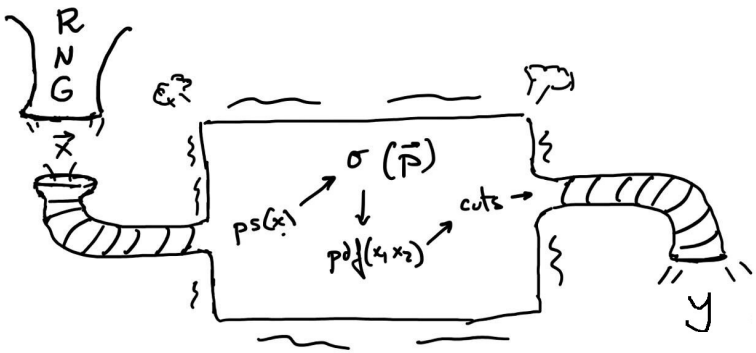


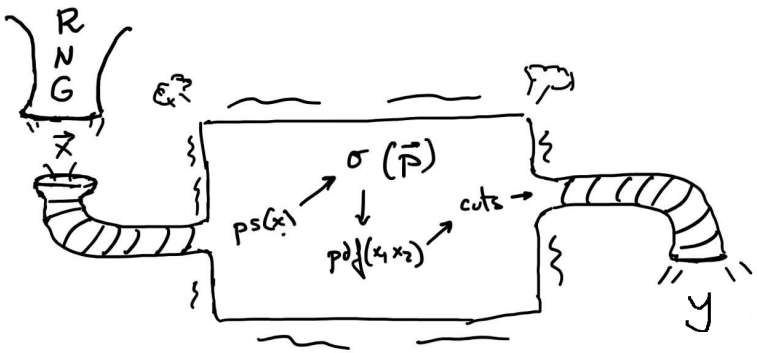
Act in parallel: CPU

The way we do Monte Carlo calculations in CPU already allows for a certain degree of parallelization



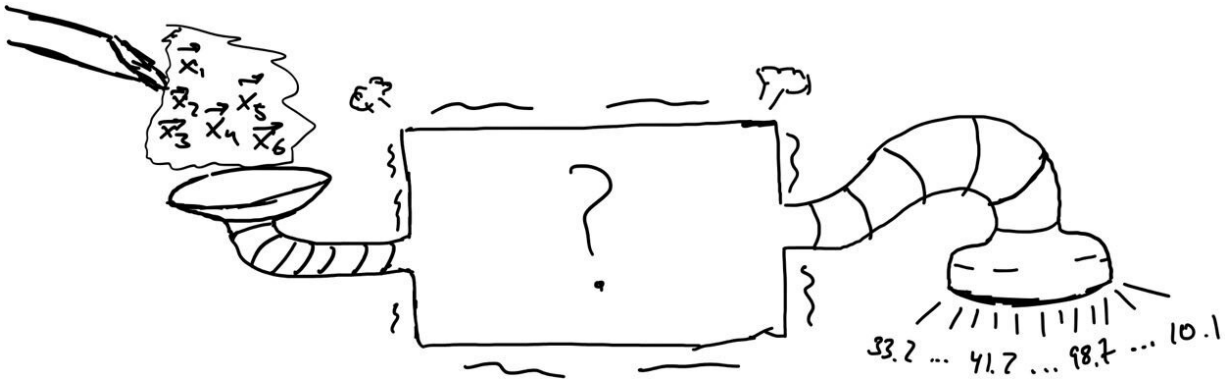






Act in parallel: GPU

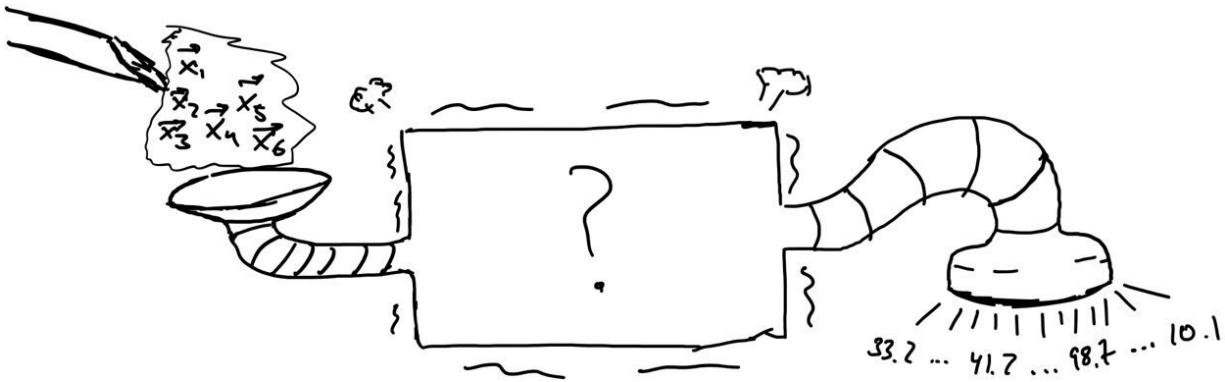
What can we do then in these machines?



We need a completely different machine, which takes a different input and a different output

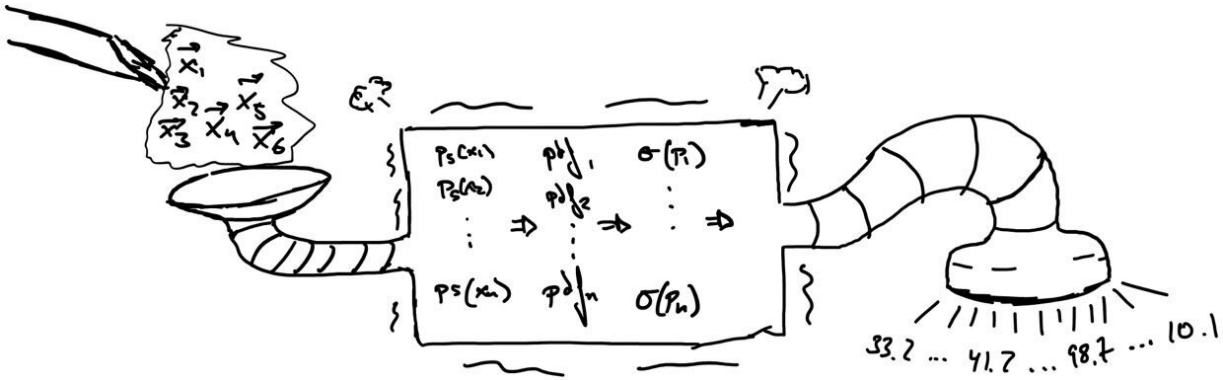
Act in parallel: GPU

What can we do then in these machines?



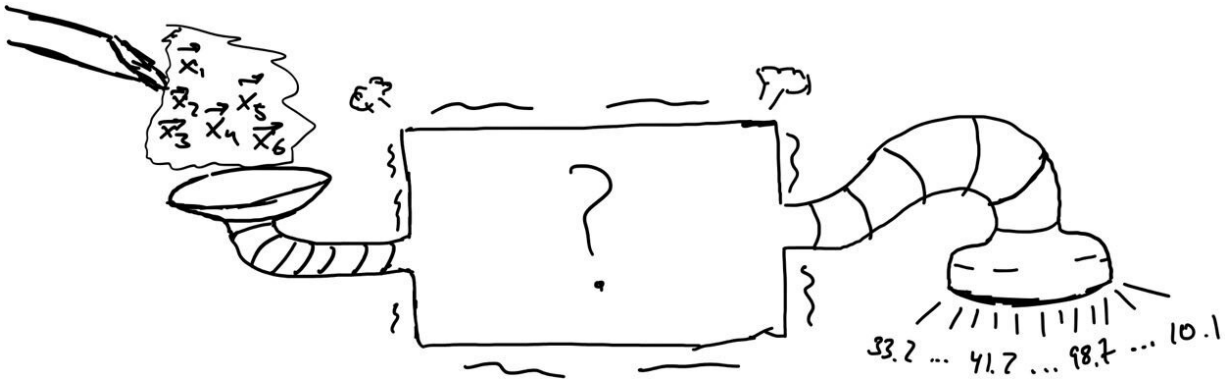
We need a completely different machine, which takes a different input and a different output

All operations must act on all inputs at once!



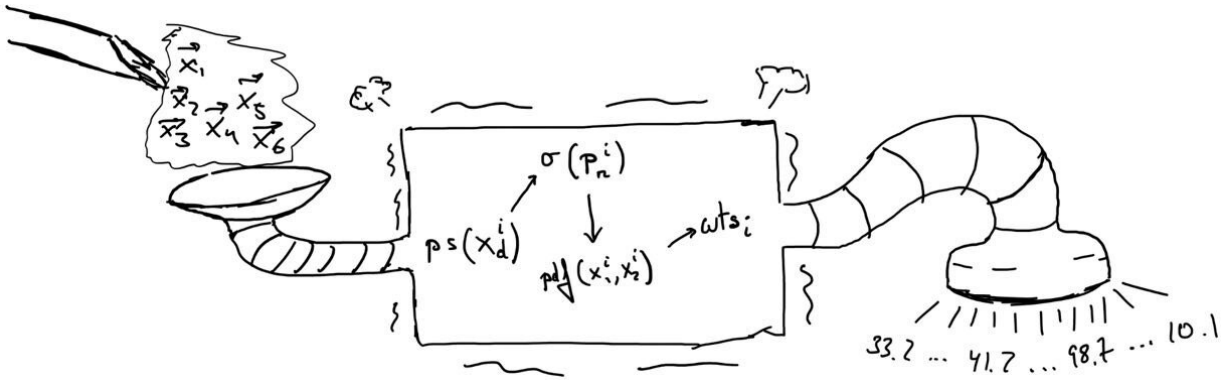
Act in parallel: GPU

What can we do then in these machines?



We need a completely different machine, which takes a different input and a different output

All operations must act on all inputs at once!



So far so good, but how can we do it?