Facilitating GPU acceleration for Monte Carlo simulations

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

[physics.comp-ph/2106.10279] Eur.Phys.J.C 81 (2021) 7, 656

Freiburg July 2022

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

Outline

1. Introduction
   - Integrating with Monte Carlo methods
   - Motivation for new technologies
   - Techniques for parallelization
2. GPU-enabled tools
   - VegasFlow, PDFFlow, MadFlow
   - Benchmarks
   - Examples
3. Conclusions
   - Open source in HEP

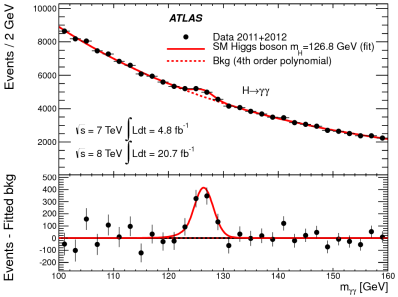
Parton-level Monte Carlo generators

Predictions for observables (for instance for LHC phenomenology) often require the numerical computation of the following integral:

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

where:

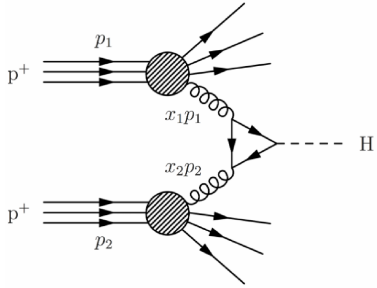
- $f(x, q)$: Parton Distribution Function
- $|M|$: Matrix element of the process
- $\{p_n\}$: Phase space for n particles.
- $\mathcal{J}$: Jet function for n particles to m.



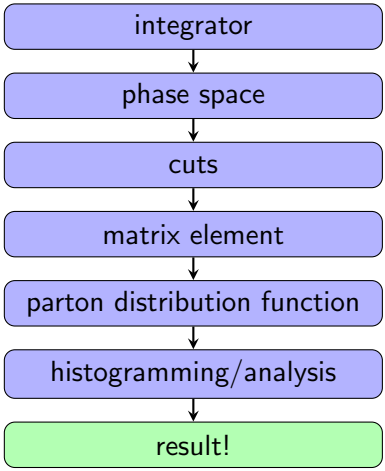
Parton-level Monte Carlo generators ingredients

In practice that requires a number of (mostly independent) ingredients:

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



The numerics being handled numerically by CPU-expensive Monte Carlo (MC) generators.



The phase space integral

Most of the difficulty of the previous equation is hidden in the differential phase space $\mathrm{d}\Phi_n$

$$\mathrm{d}\Phi_n = \prod_{i=1}^{n} \left(\frac{\mathrm{d}^3 p_i}{2E_i(2\pi)^3} \right) (2\pi)^4 \, \delta^4 \left(p_a + p_b - \sum p_i \right).$$

Where the dimensionality of the integration grows with the number of particles in the final state n as $3n-4$ $(+2)$. The resulting integral presents the following issues:

- $\rightarrow$ High dimensionality ($2 \rightarrow 2$ is at least dimension 4)
- $\rightarrow$ Complicated integration limits: the physical space we are integrating over (the detector geometry) doesn't necessarily match in a clean way the particles momenta.
- $\rightarrow$ Divergences: to add insult to injury, the $\mathrm{M}_{i \rightarrow f}$ quantity might be divergent for single events even if the integral converges. Theses divergences are either cancelled numerically or by playing with the integration limits.

The age of precision

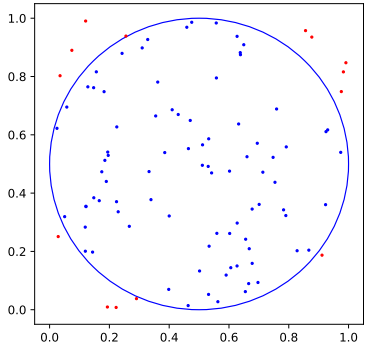
MC algorithms are very convenient as they provide an estimate of the integral and of the error while imposing almost no assumptions on the integrand.

$$\int \mathrm{d}^d x f(\vec{x}) \simeq \frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i) = I \qquad \text{var} = \frac{1}{N} \left(\frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i)^2 - I^2 \right)$$

However, the error decreases with the number of shots only as $\frac{1}{\sqrt{N}}$.

Despite techniques to reduce the number of shots necessary, the $\mathcal{O} \simeq \frac{1}{\sqrt{N}}$ holds.

$\pi \simeq 3.2$



The age of precision

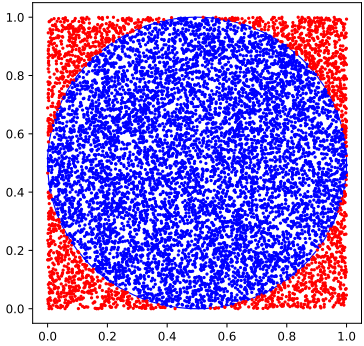
MC algorithms are very convenient as they provide an estimate of the integral and of the error while imposing almost no assumptions on the integrand.

$$\int \mathrm{d}^d x f(\vec{x}) \simeq \frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i) = I \qquad \text{var} = \frac{1}{N} \left(\frac{1}{N} \sum_{i=1}^{N} f(\vec{x}_i)^2 - I^2 \right)$$

However, the error decreases with the number of shots only as $\frac{1}{\sqrt{N}}$.

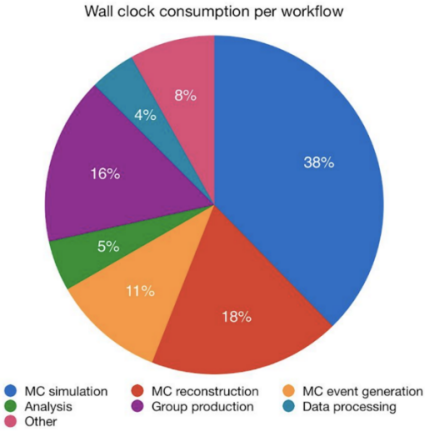
It can take forever

$\pi \simeq 3.16$

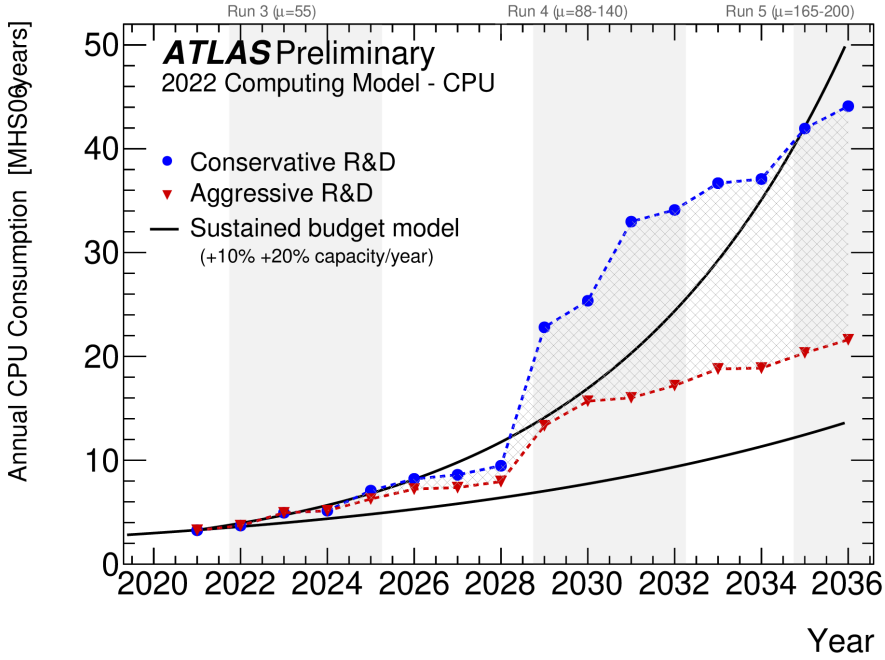


ATLAS current (2018) CPU usage

- *ATLAS CPU hours used by various activities in 2018*



ATLAS projected CPU usage



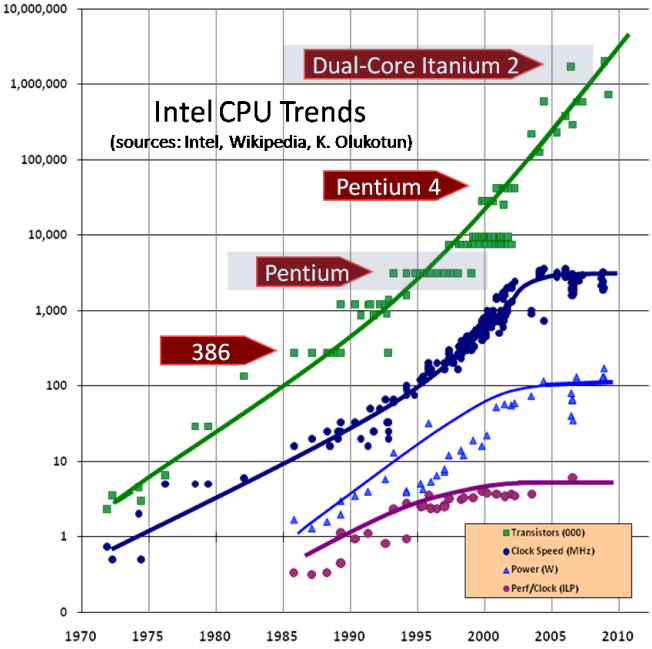
CPU parallelization

For years adding more power/transistors was enough

Then adding more cores...

... but even that is not enough anymore.

From H. Sutter's "The Free Lunch Is Over"

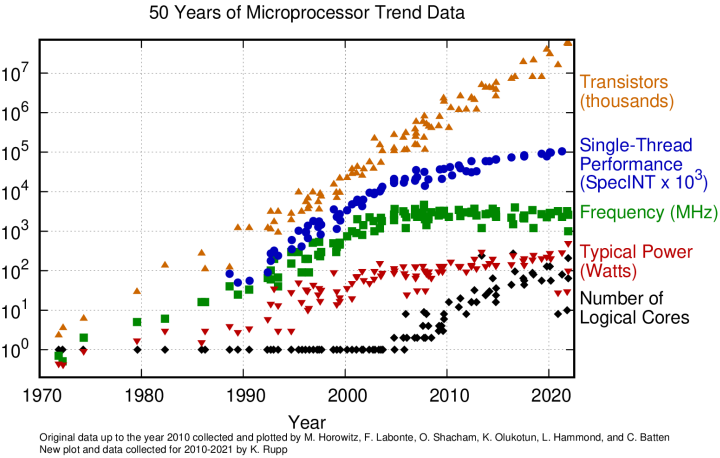


CPU parallelization

For years adding more power/transistors was enough

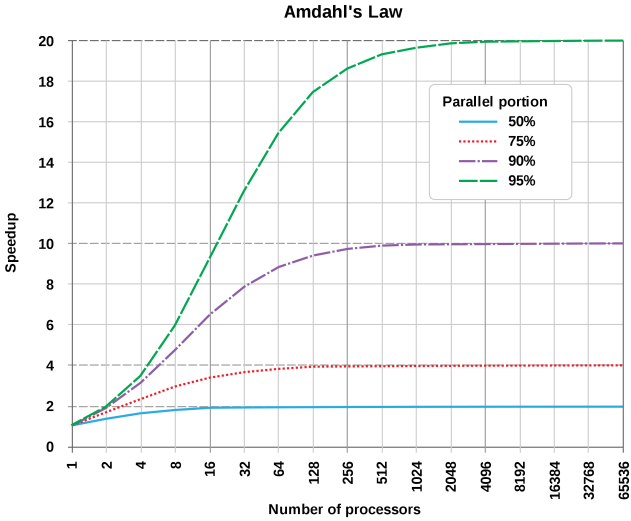
Then adding more cores...

... but even that is not enough anymore.



Just add more cores

Wait, not so fast...

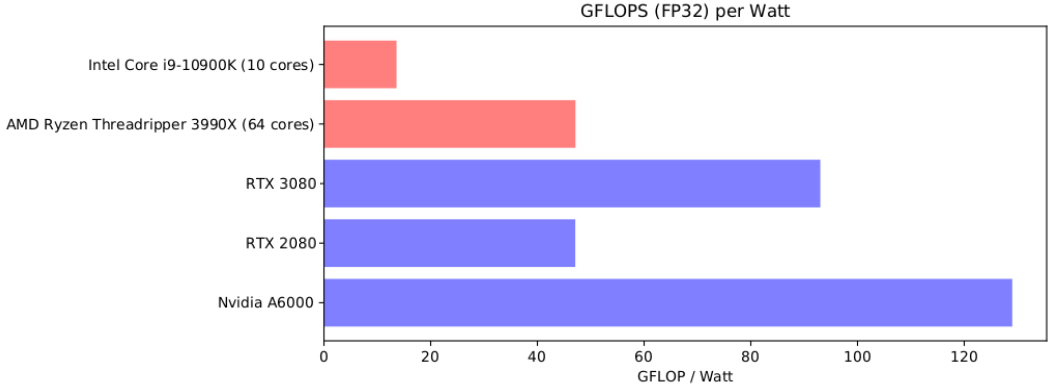


Plus...

- ✗ Power consumption
- ✗ Race conditions
- ✗ The memory wall
- ✗ Moore's law still applies

Why move to hardware accelerators

- ✓ Better performance
- ✓ Better efficiency
- ✓ GPUs are now as capable (and competitive!) as CPUs for many operations



Hardware accelerators

Or how I learned to stop worrying and love the ~~Central~~ Graphical Processing Unit

GPUs are designed to perform many operations at once in parallel:

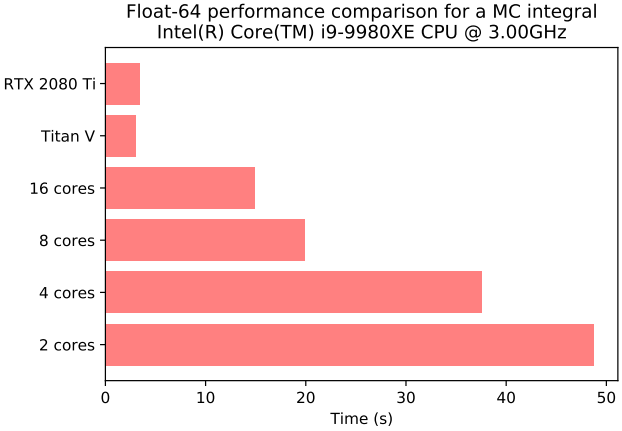
- ✗ Each "worker" in the GPU must be doing the same as all its siblings
- ✗ Cannot share data during the calculation[1]
- ✗ in summary: only useful for calculations where each event is independent of all other events and...

Wait...

[1] Not in the CPU sense anyway

GPU computing

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



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.

If it is so good, why are we not using GPUs everywhere?

At least in the field of theoretical calculations there are a few points holding progress back

✗ 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.

If it is so good, why are we not using GPUs everywhere?

At least in the field of theoretical calculations there are a few points holding progress back

✗ 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.

If it is so good, why are we not using GPUs everywhere?

At least in the field of theoretical calculations there are a few points holding progress back

✗ 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.

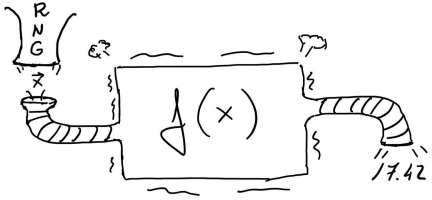
✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

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

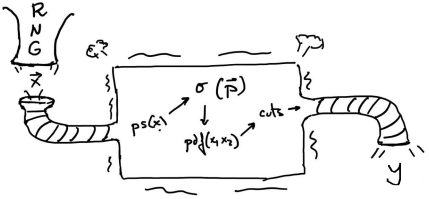
Act in parallel: CPU

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

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



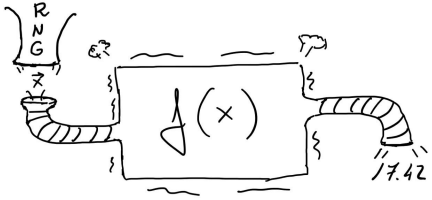
Where the form of the function $f(\vec{x})$ might be arbitrarily complicated



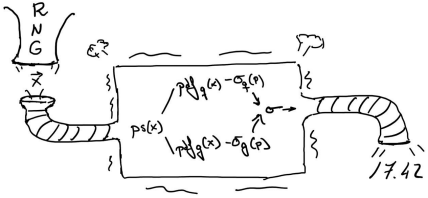
Act in parallel: CPU

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

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

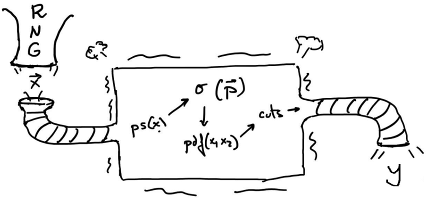


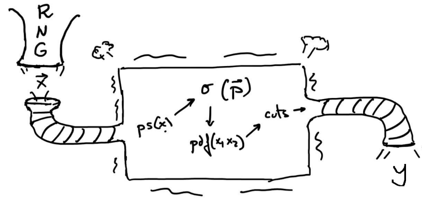
Where the form of the function $f(\vec{x})$ might be arbitrarily complicated

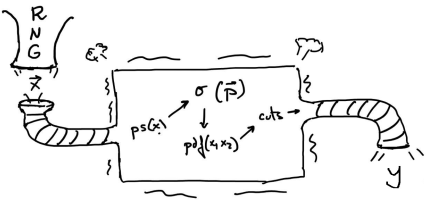


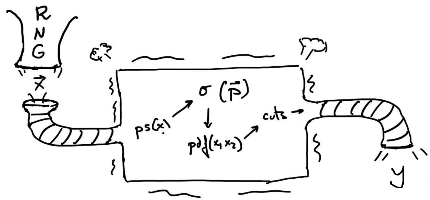
Act in parallel: CPU

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



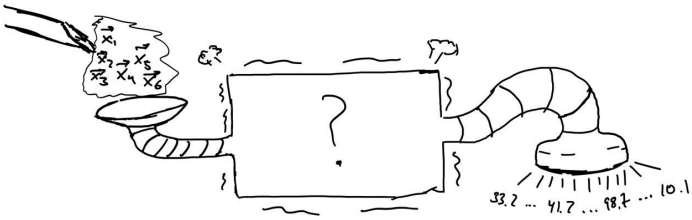






Act in parallel: GPU

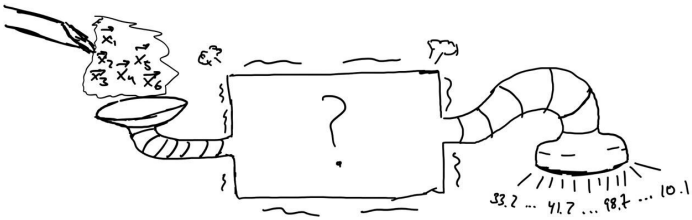
What can we do then in these machines?



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

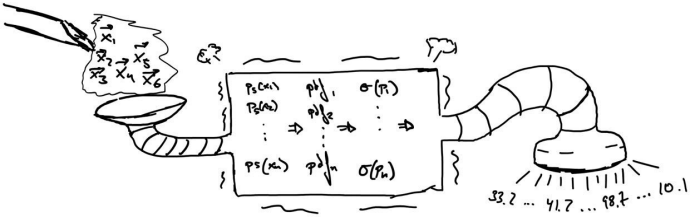
Act in parallel: GPU

What can we do then in these machines?



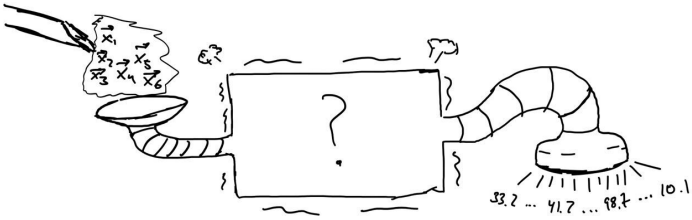
We need a completely different machine, which takes a different input and a different output

All operations must act on all inputs at once!



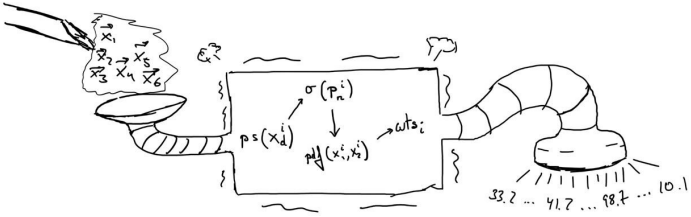
Act in parallel: GPU

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?

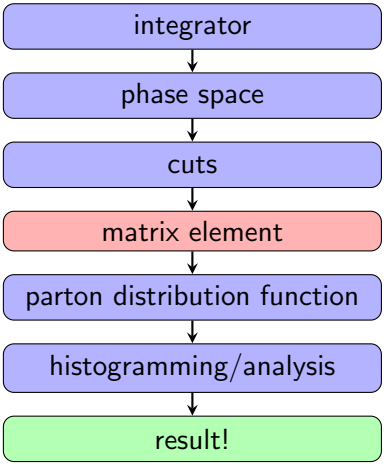
Lack of Tools

Running on a CPU:

Indeed, you can usually only worry about the part of the calculation that you are interested in (say, a new NNLO matrix element).

While you can find tools that solve everything else (if you didn't already had that tools yourself!)

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



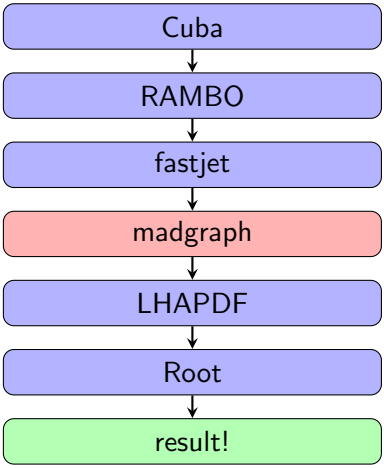
Lack of Tools

Running on a CPU:

Indeed, you can usually only worry about the part of the calculation that you are interested in (say, a new NNLO matrix element).

While you can find tools that solve everything else (if you didn't already had that tools yourself!)

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

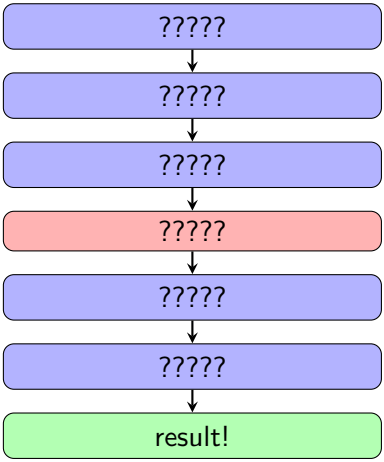


Lack of Tools

Running on a GPU:

There is no such tool set yet

so it needs to be written from scratch



Wish list

Ideally we want a framework such that:

- Able to run on GPU
- ...of any brand!
- Can be interfaced with other languages.
- With primitives for often-used operations.
- Same program running in CPU and GPUs.
- Actively mantained.
- No need to learn a GPU-specific language.

There are many frameworks available that can be used but they are often quite complex or fail one of the given points.

- CUDA
- OpenCL
- OpenACC
- numba
- Alpaka
- Kokkos
- etc...

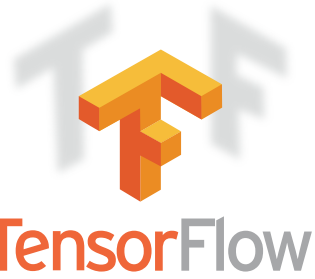
However we can look at another set of tools: Machine Learning frameworks.

Wish list

Ideally we want a framework such that:

- Able to run on GPU
- ...of any brand!
- Can be interfaced with other languages.
- With primitives for often-used operations.
- Same program running in CPU and GPUs.
- Actively mantained.
- No need to learn a GPU-specific language.

Most machine learning frameworks usually comply with our wishlist, the two most used frameworks at the moment are `TensorFlow` and `pyTorch`.



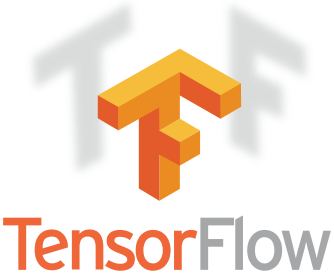
Filling up the box: tools for modern computation

The goal is to provide tools that can facilitate the transition:

VegasFlow: Monte Carlo library with different algorithms that can be used in any device: single-threaded and multi-threaded CPUs or AMD/nvidia GPUs.

PDFFlow: Bulk PDF interpolation, specially well suited for parallel calculation where sequential steps can harm performance.

- Python and TF-based engine.
- Compatible with other languages: Cuda, C++, Rust, Fortran.
- Seamless CPU and GPU computation out of the box (develop in a laptop, deploy in a cluster).
- A language with a mathematically oriented "standard library".



Other efforts

Beyond the tools presented on this talk, other groups are also working on the same direction:

- TorchQuad by the ESA, similar to VegasFlow but using pytorch instead: https://github.com/esa/torchquad

Most of the HEP focus is in the generation of Matrix Elements for Sherpa and Madgraph:

- Madgraph for GPU: make fortran generate cuda code instead of fortran.
- BlockGen algorithms using Berends-Giele recursion.

A LHAPDF vectorized/parallel implementation is in the roadmap for 6.6 using Kokkos.

Easy to use and develop

Minimal changes to the external interface can achieve enormous speed ups.

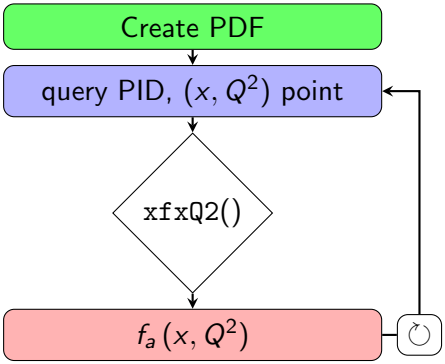


Figure: `LHAPDF6`

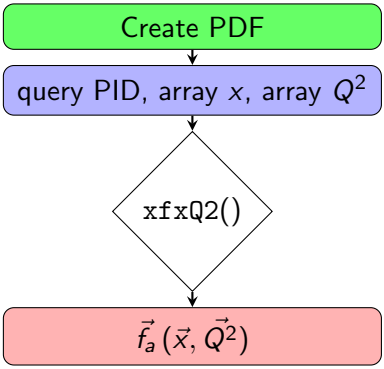


Figure: `PDFFlow`

The heavy lifting is done "automagically" internally in `PDFFlow`

VegasFlow in detail

The VegasFlow library focuses on speed and efficiency for both the computer and the developer

- Python and `TensorFlow` based engine
- Compatible with other Cuda, C++, Fortran (and of course, python)
- Seamless CPU and GPU computation out of the box
- Easily interfaceable with NN-based integrators (if you don't like MC anymore).
- Tensors are a natural way of thinking for many physicists!

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

$$w_i = M_i^j v_j$$

In a classic C++ program we would write:

```
#pragma omp parallel for
for(int i = 0; i < 3; i++) {
    w[i] = 0;
    for(int j = 0; j < 3; j++) {
        w[i] += M[i][j] * v[j]
    }
}
```

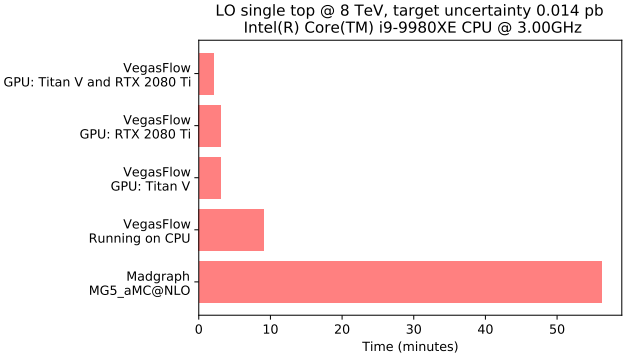
while instead we can just do:

```
w = tf.einsum("ij, j -> i", M, v)
```

TensorFlow will automatically select the right code depending on the target hardware, `pragmas` included!

VegasFlow Vs collission simulators

For Leading Order calculations the advantages are immediately visible

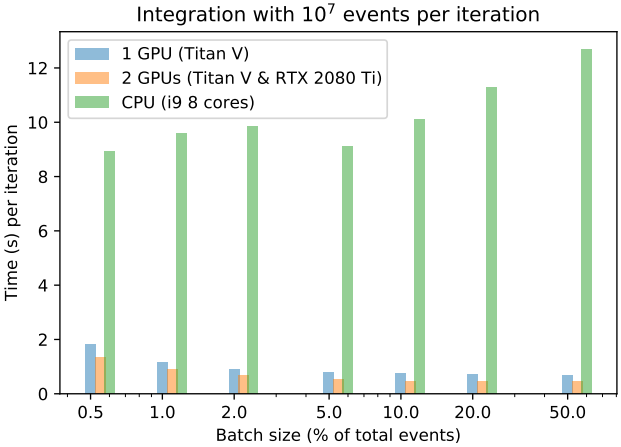


Madgraph Vs VegasFlow implementation

- Port of CPU (C++ based) code, no GPU-specific algorithms (beyond "tensorization")
- Meaning: many points being thrown away, which is ok for CPU but harmful for GPU performance!

VegasFlow Vs collission simulators

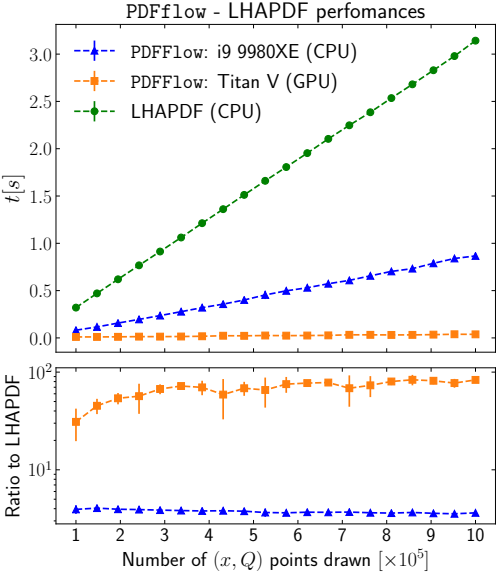
For Leading Order calculations the advantages are immediately visible

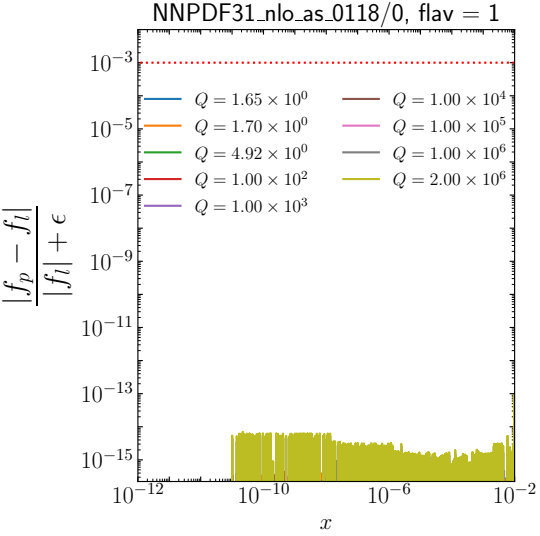


VegasFlow implementation in different devices

- Port of CPU (C++ based) code, no GPU-specific algorithms (beyond "tensorization")
- Meaning: many points being thrown away, which is ok for CPU but harmful for GPU performance!

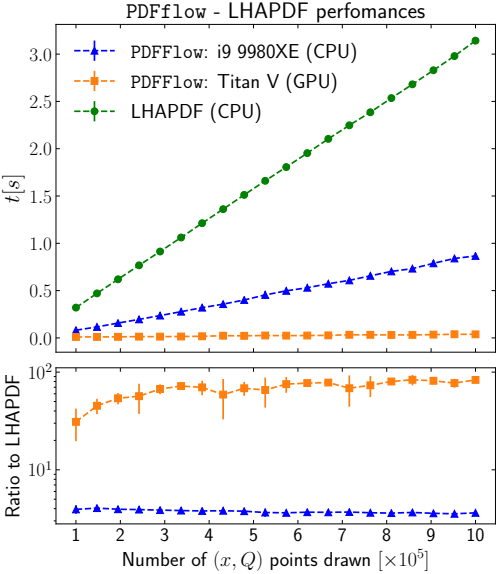
LHAPDF vs PDFFlow

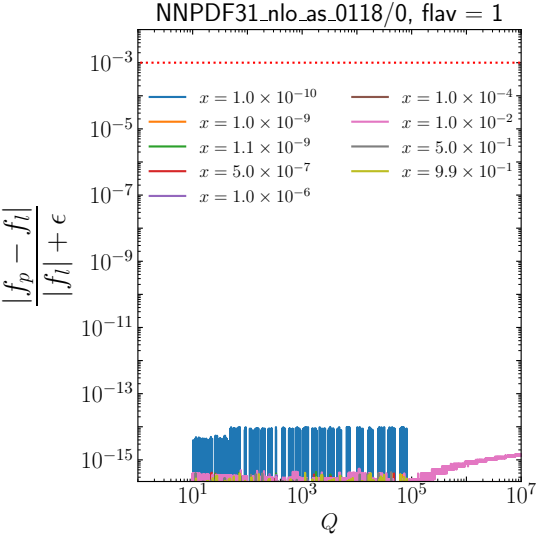




Interpolation in x for fixed q.

LHAPDF vs PDFFlow





Interpolation in q for fixed x.

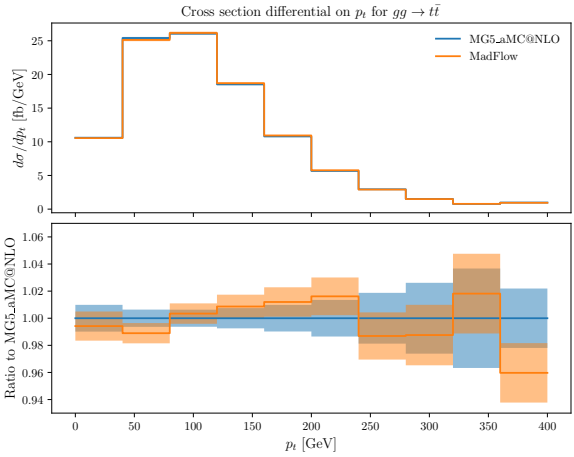
MadFlow: a madgraph interface

An example of what can be obtained is **MadFlow**: Taking advantage of Madgraph's ALOHA we produce `tensorflow`-versions of the matrix elements.

The `TensorFlow` library contains all necessary kernels to run the matrix elements in parallel.

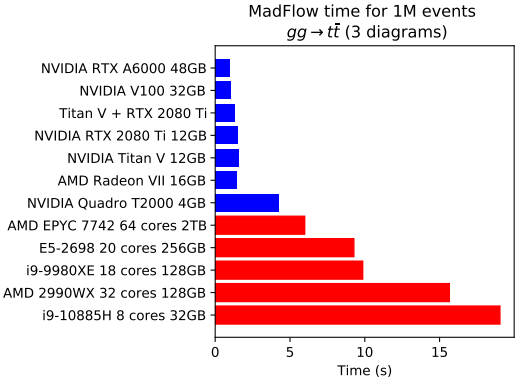
Everything can run in both a CPU or a GPU

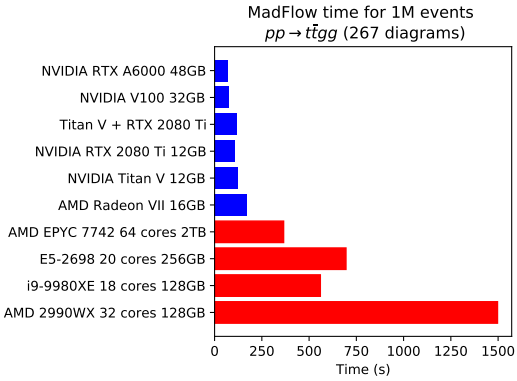
Perfect compatibility ✓



MadFlow in different devices

We see speed-ups for both complex and simple processes



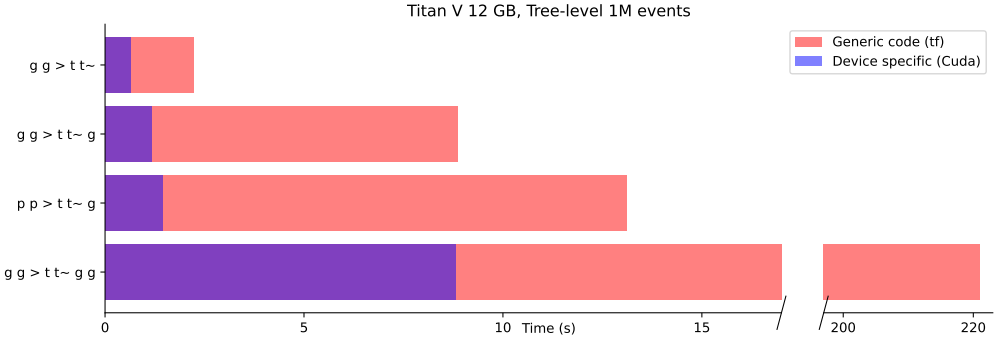


While having the whole thing in `TensorFlow` gives us great flexibility (the exact same code is running in all those systems) we might want to forfeit some flexibility in exchange for device-based optimization.

Interfacing with CUDA

Main goal: flexibility and ease-of-use, that also means one can also make life harder for oneself... for a benefit.

- ✗ Limited to a single architecture
- ✗ Requires a transpiler to the "low"-level language of choice
- ✓ More efficient memory management and performance
- ✓ We can limit this more complicated step only to the bottlenecks



Usability

The goal

The developer writes the code once (for instance, the matrix element for the process they are interested in) and it can automatically be used for both GPU and CPU.

There is an obvious and common caveat: what if I already have some codebase (e.g., analysis tools) that have no need to be parallelized but that I need to call at certain stages of the calculation.

The workaround

We have focus on compatibility with existing code and have tested interfaces with Cuda, C++ and regular python at many stages of the development.

In what follows I will show some examples.

Run a simple integrand

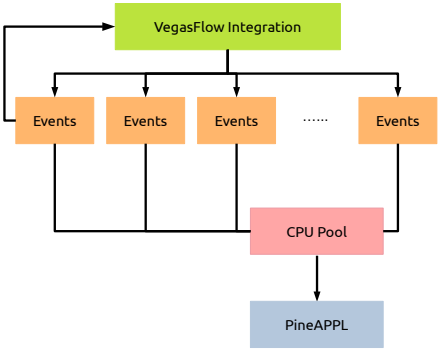
```
>>> @tf.function
>>> def complicated_integrand(xarr, **kwargs):
>>>     return tf.reduce_sum(xarr, axis=1)
>>> from VegasFlow.vflow import VegasFlow
# Instantiate the integrator
# limit the number of events to be computed at once
# (hardware dependent!)
>>> n_dim = 10
>>> n_events = int(1e6)
>>> integrator = VegasFlow(n_dim, n_events, events_limit = int(1e5))
# Register the integrand
>>> integrator.compile(complicated_integrand)
# Run a number of iterations
>>> res = integrator.run_integration(n_iter = 5, log_time = True)


Result for iteration 0:  5.0000 +/- 0.0009(took 0.47029 s)
Result for iteration 1:  5.0006 +/- 0.0003(took 0.32042 s)
                    ⋮
Final results:  4.99995 +/- 8.95579e-05
```

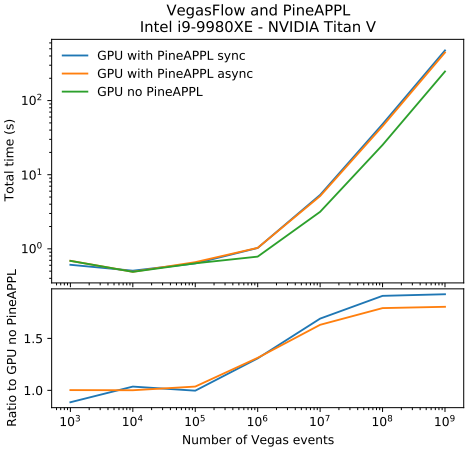
Analyze the results mid-way

```python
from vegasflow import vegas_wrapper
import tensorflow as tf

def my_cpu_only_analysis(results):
    # Very complicated analysis:
    return 2.0*results

def my_integrand(xarr, **kwargs):
    # Start the calculation in the GPU
    raw = tf.sin(xarr*4.0*3.1416)**2
    res = tf.py_function(my_cpu_only_analysis, [raw], Tout=tf.float64)
    # continue in the GPU
    return tf.reduce_prod(res, axis=1)*xarr.shape[-1]**2

n_dim = 4
n_events = int(1e4)
n_iter = 5
result = vegas_wrapper(my_integrand, n_dim, n_iter, n_events)
```

PineAPPL

Example: interfce with the grid filling tool PineAPPL (Carrazza, Nocera, Schwan, Zaro, hep-ph/2008.12789) addresses the problem of generating grids to produce predictions for generic set of PDFs.





The generation of such grids is a common use of Monte Carlo generators.

The importance of being open

Usual concerns:

- ✗ The code is too ugly to follow / be useful
- ✗ We would need to dedicate people to technical assistance
- ✗ Other fears about scooping or people finding bugs in the code.

But in reality:

- ✓ Perfect is the enemy of good
- ✓ Having many users is a good thing! (also, you can always decide not to respond)
- ✓ Opening the door to new collaborations and improvements is a good thing!

Where to find the code

Where to obtain the code

Vegasflow, **PDFFlow** and **MadFlow** are open source and can be found at the N3PDF organization repository `github.com/N3PDF` (alongside other projects by the group)

How to install

They can all easily be installed with `pip`:

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

Documentation

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

Summary

- Monte Carlo simulations (fixed-order and otherwise) are great targets for parallelization on hardware accelerators.

→ Despite being more than competitive with CPU not many groups are working on it!

✗ Maybe we still have a big entry barrier?

✓ 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.

✓ **Remove all entry barrires while still leaving space for further optimization**

Available open source

the code for madflow is available at `https://github.com/n3pdf/madflow`

Thanks!