VegasFlow and PDFFlow: offloading Monte Carlo simulations to hardware accelerators

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

PDFFlow: hep-ph/2009.06635 VegasFlow: 10.1016/j.cpc.2020.107376

Machine Learning • PDFs • QCD

Milan Joint Phenomenology Seminar
February 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.

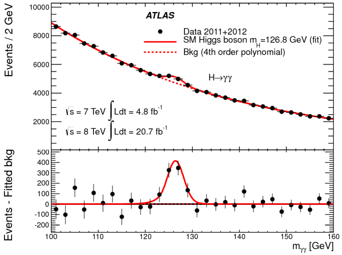
Outline

Parton-level Monte Carlo generators

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

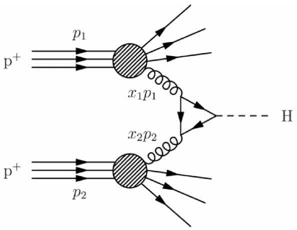
$$\int \mathrm{d}x_1\, \mathrm{d}x_2\, f_1(x_1, q^2) f_2(x_2, q^2) |M(\{p_n\})|^2 \mathcal{J}_m^n(\{p_n\})$$

→ $f(x, q)$: 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:

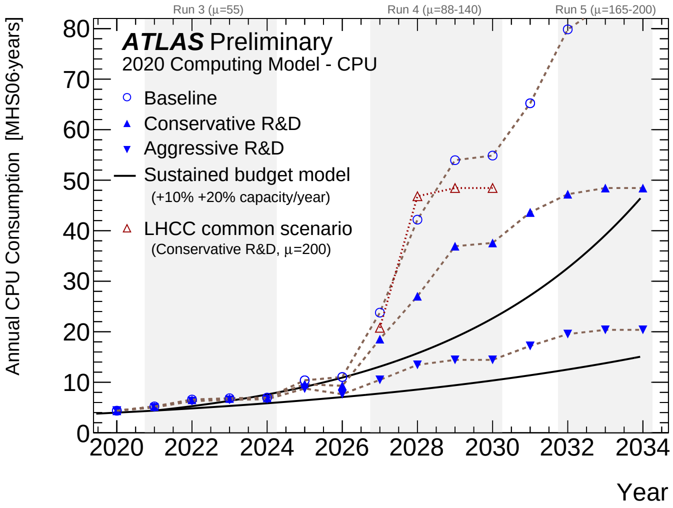
$$\int \mathrm{d}x_1\,\mathrm{d}x_2\, f_1(x_1,q^2) f_2(x_2,q^2) |M(\{p_n\})|^2 \mathcal{J}_m^n(\{p_n\})$$



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

$$\begin{tikzcd}
\text{integrator} \arrow[d] \\
\text{phase space} \arrow[d] \\
\text{cuts} \arrow[d] \\
\text{matrix element} \arrow[d] \\
\text{parton distribution function} \arrow[d] \\
\text{histogramming/analysis} \arrow[d] \\
\text{result!}
\end{tikzcd}$$

ATLAS projected CPU usage



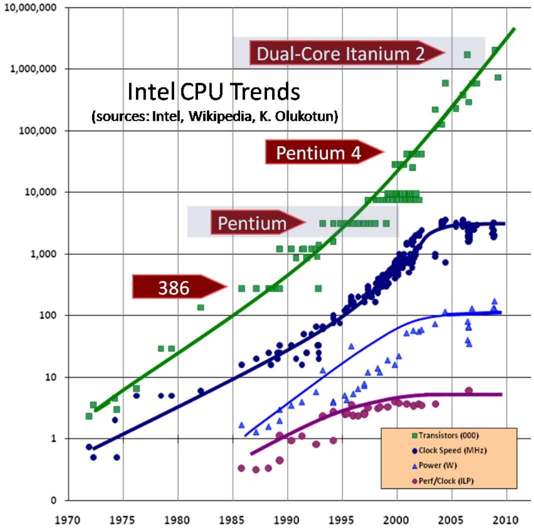
CPU parallelization

For years adding more power/transistors was enough

Then adding more cores...

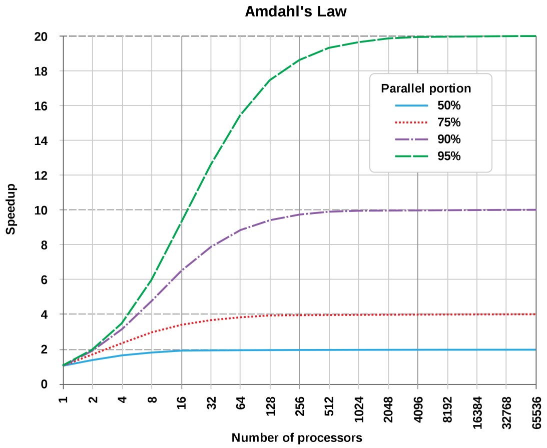
... but even that is not enough anymore.

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



Just add more cores

Wait, not so fast...

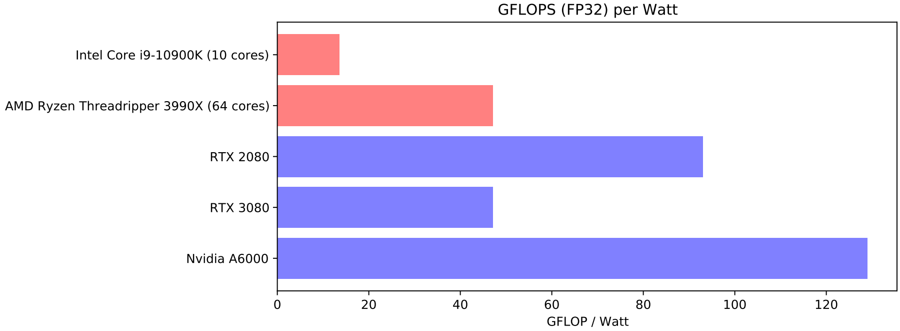


Plus...

- ✗ 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:

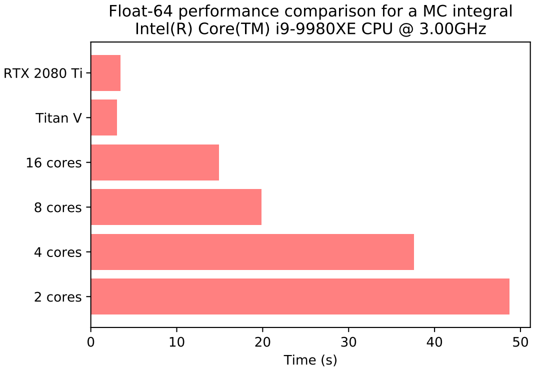
- ✗ Each “worker” in the GPU must be doing the same as all its siblings
- ✗ Cannot share data during the calculation[1]
- ✗ in summary: only useful for calculations where each event is independent of all other events and...

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

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

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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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?

Most of the more advance theoretical calculations still rely exclusively on CPU. With only a few libraries providing GPU interfaces such as pySecDec.

✗ 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.

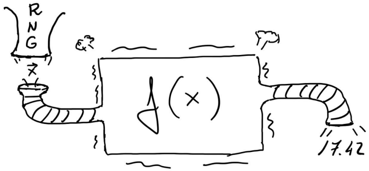
✗ Lack of expertise
- CPU expertise is not necessarily applicable to GPU programming.
- New programming languages: Cuda? OpenCL?
- Low-reward situation when trying to achieve previous performance.

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

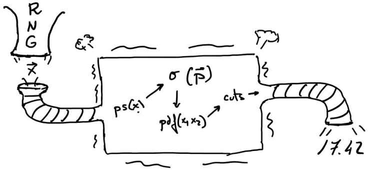
Act in parallel: CPU

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

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



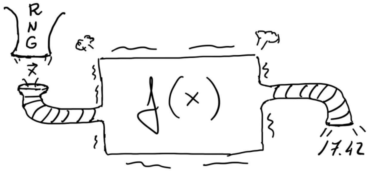
Where the form of the function $f(\vec{x})$ might be arbitrarily complicated



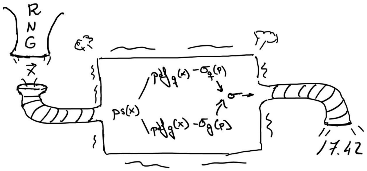
Act in parallel: CPU

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

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

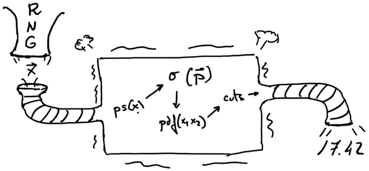


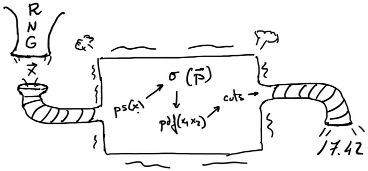
Where the form of the function $f(\vec{x})$ might be arbitrarily complicated

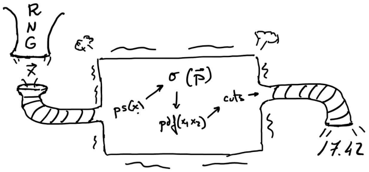


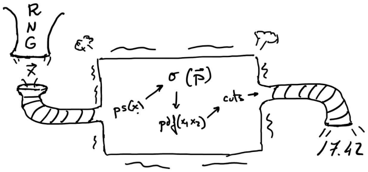
Act in parallel: CPU

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



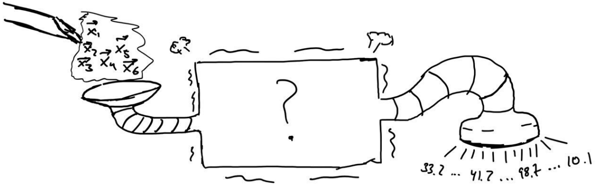






Act in parallel: GPU

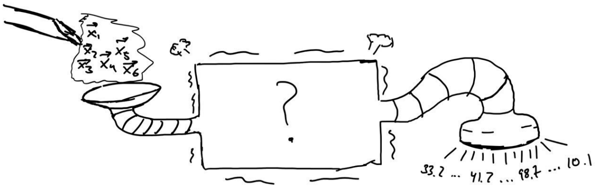
What can we do then in these machines?



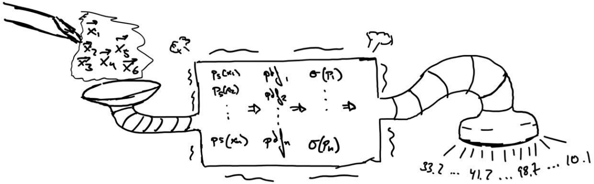
We need a completely different machine, which takes a different input and a different output

Act in parallel: GPU

What can we do then in these machines?



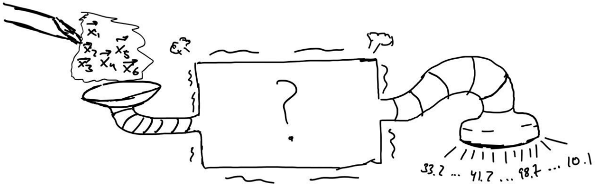
We need a completely different machine, which takes a different input and a different output



All operations must act on all inputs at once!

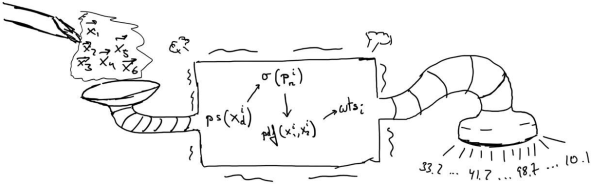
Act in parallel: GPU

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!



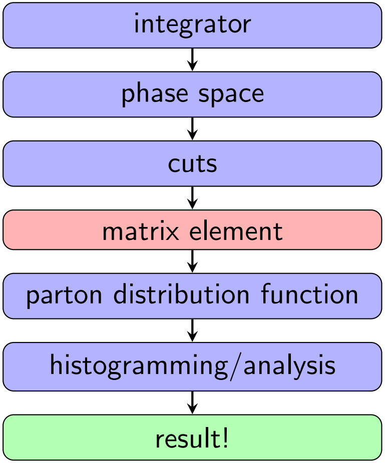
So far so good, but how can we do it?

Lack of Tools

Running on a CPU:

Worry only about what you are instered in (i.e., the phyisical process)

There exists a complete toolset for producing results.



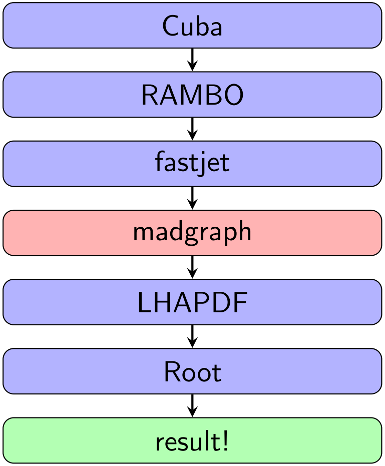
Lack of Tools

Running on a CPU:

Worry only about what you are instered in (i.e., the phyisical process)

There exists a complete toolset for producing results.

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

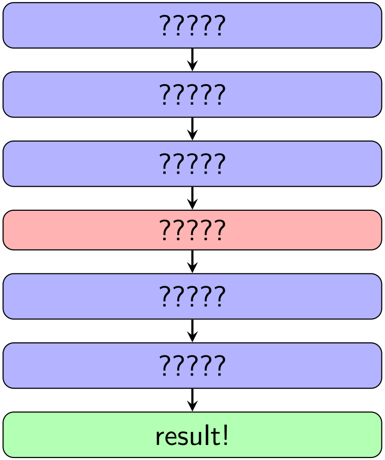


Lack of Tools

Running on a GPU:

There is no such toolset 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

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

2. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase 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
2. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase space points.
3. 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

2. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase space points.

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

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

Filling up the box

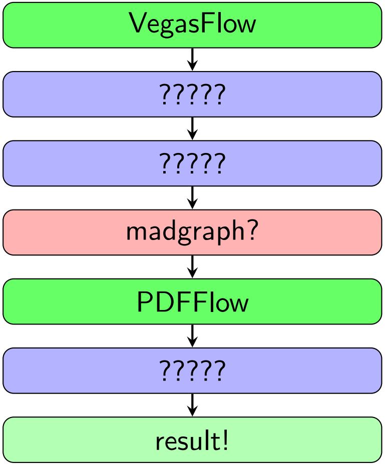
1. A phase space generator which takes an array of (`n_events`) random numbers and returns (`n_events`) an array of phase space points
2. A tool for the evaluation of Matrix Elements (at tree or loop level) parallelized on the received phase space points.
3. A PDF interpolation tool to generate luminosities in parallel for many events ✓
4. An integrator framework able to send and receive batches to and from the GPU ✓
5. Analysis tools, experiment simulation, jet algorithms...

Filling up the box: VegasFlow and PDFflow

The pdf and vegas-flow libraries 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



Easy to use

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

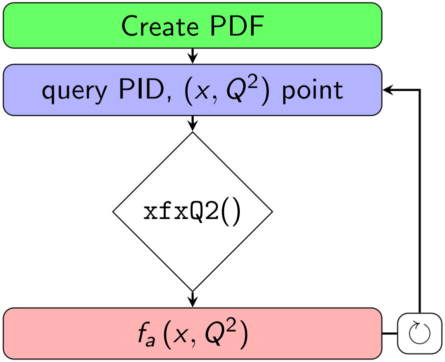


Figure: `LHAPDF6`

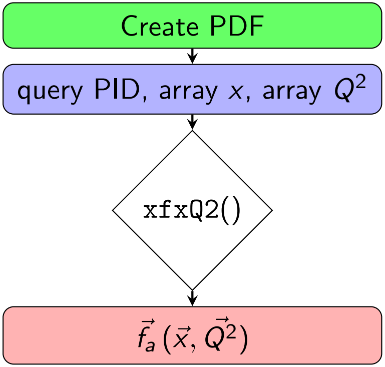

Create PDF
query PID, array x, array Q^2
xfxQ2()
$\vec{f}_a\left(\vec{x}, \vec{Q}^2\right)$

Figure: `PDFFlow`

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

Usability status

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.

While perfectly possible, it will take some times for the tools to be widely used. The ecosystem today available to the particle physics comunity is far from being GPU-ready and it will take years to be truly plug & play.

The workarounds

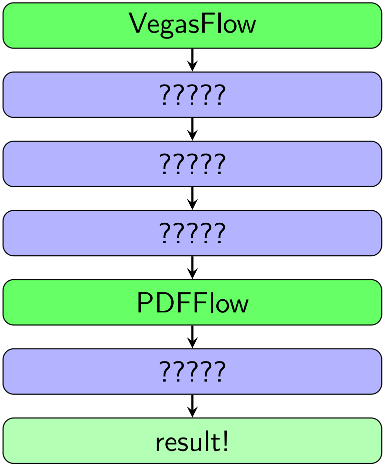
We have thus focused on compatibilities with existing code and tools. PDFFlow python and C interfaces follow a structure very similar to LHAPDF while VegasFlow is compatible with integrands written in Cuda, C++ or regular python.

Interface with Madgraph's matrix generation

work in collaboration with M. Zaro

A very powerful tool in the phenomenology's toolbox is the Madgraph. Among its many features it can automatically generate partonic cross sections.

- ✓ Can we generate them automatically in a tensorized form?



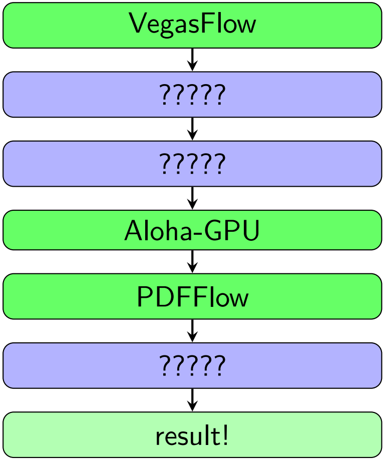
Interface with Madgraph's matrix generation

work in collaboration with M. Zaro

A very powerful tool in the phenomenology's toolbox is the Madgraph. Among its many features it can automatically generate partonic cross sections.

- ✓ Can we generate them automatically in a tensorized form?

- ✓ Yes, we can!



~~Madflow~~ ~~Alohaflow~~ ~~mg5MC@NLOwVegasFlow~~ NIP

1. Use Madgraph to generate all necessary channels

~~Madflow~~ ~~Alohaflow~~ ~~mg5MC@NLOwVegasFlow~~ NIP

1. Use Madgraph to generate all necessary channels
2. Output the calculation in a way that can be run in parallel

~~Madflow~~ ~~Alohaflow~~ ~~mg5MC@NLOwVegasFlow~~ NIP

1. Use Madgraph to generate all necessary channels
2. Output the calculation in a way that can be run in parallel
3. Write a wrapper to join all pieces: phase space generator (RamboFlow?), VegasFlow, PDFFlow, ...

~~Madflow~~ ~~Alohaflow~~ ~~mg5MC@NLOwVegasFlow~~ NIP

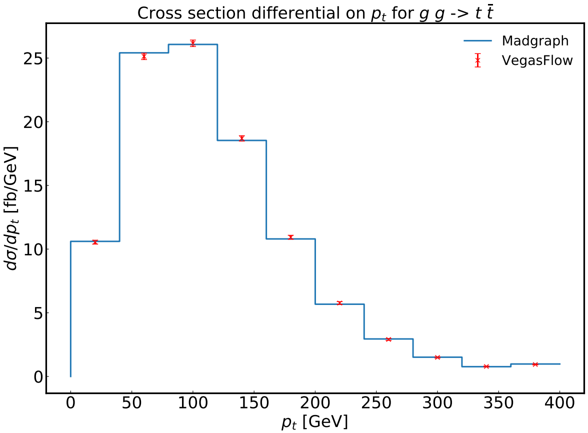
1. Use Madgraph to generate all necessary channels
2. Output the calculation in a way that can be run in parallel
3. Write a wrapper to join all pieces: phase space generator (RamboFlow?), VegasFlow, PDFFlow, ...
4. Run the process in a GPU!

Preliminary Results

Exact same ME and feynman diagrams ✓

"Hand-made" phase space ✗

Perfect compatibility ✓

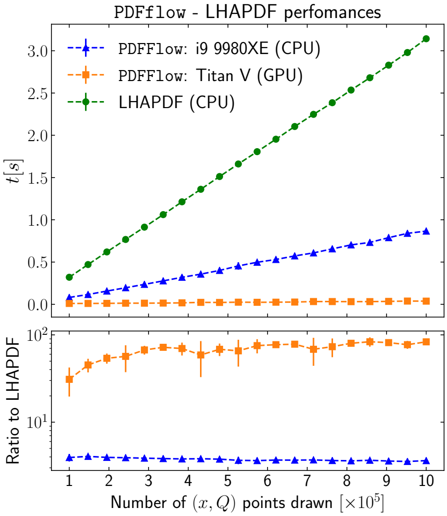


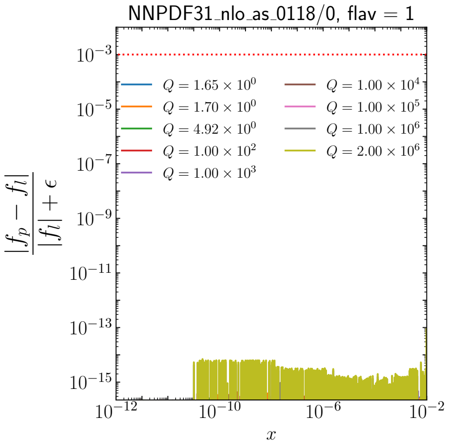
Benchmarks and examples

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
- ✓ LO calculations, CPU vs GPU
- ✓ NLO caluculations, CPU vs GPU
- ✓ Pineapple and Dask integration
- ✓ + some example code

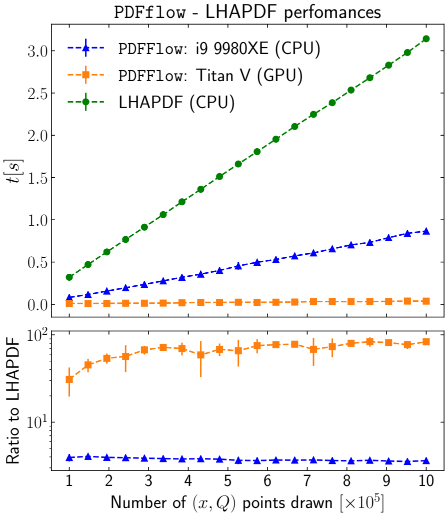
LHAPDF vs PDFFlow

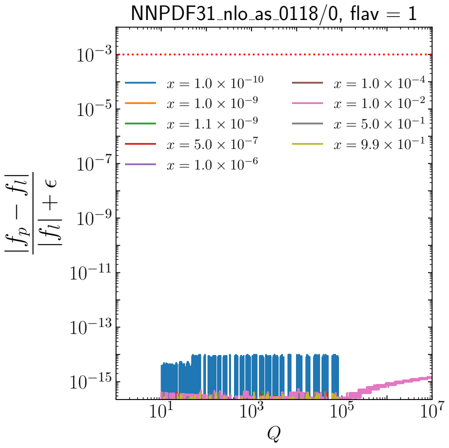




Interpolation in x for fixed q.

LHAPDF vs PDFFlow





Interpolation in q for fixed x.

VegasFlow Vs plain Madgraph LO

For Leading Order calculations the advantages are immediately visible

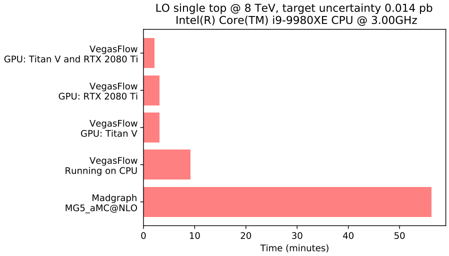


Figure: Plain Madgraph Vs VegasFlow implementation

- Port of CPU (C++ based) code, no GPU-specific optimization.
- Phase Space, spinors, cuts... all done 'the old way"

✓ There's even room for improvement with GPU-specific code!

VegasFlow Vs plain Madgraph LO

For Leading Order calculations the advantages are immediately visible

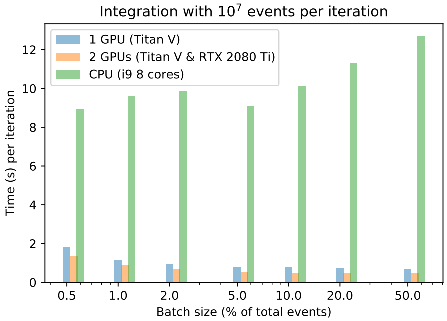


Figure: VegasFlow implementation in different devices

- Port of CPU (C++ based) code, no GPU-specific optimization.
- Phase Space, spinors, cuts... all done ‘the old way”
- ✓ There’s even room for improvement with GPU-specific code!

And what about NLO?

VegasFlow for NLO calculations

Still can't achieve an order of magnitude for NLO. But it is already better!

- Same caveats as before → no GPU-specific optimization on the phase space, cuts or subtraction terms

- Proof-of-concept, not a full parton-level MC simulator.

✓ Great potential for accelerating fixed order calculations.

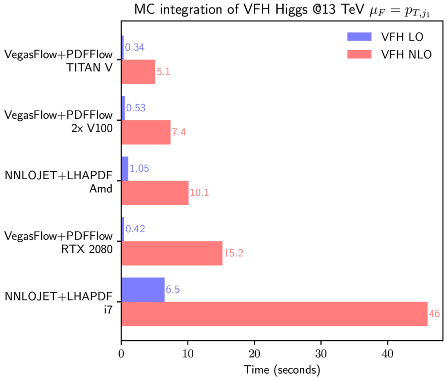
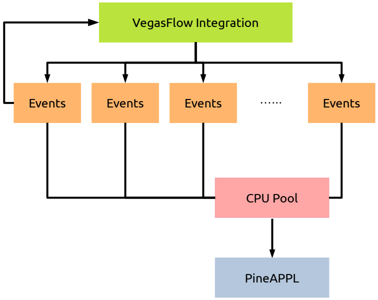
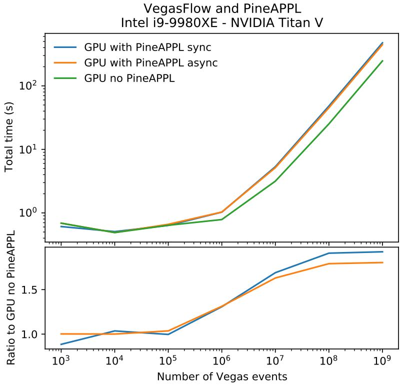


Figure: NNLOJET+LHAPDF vs VegasFlow+PDFFlow

PineAPPL

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.

Dask

Another very common situation for users of Monte Carlo tools is the possibility of running in a distributed system.
In VegasFlow this task is facilitated by the implementation of a dask interface.

```
>>> from dask_jobqueue import SLURMCluster
>>>
>>> cluster = SLURMCluster(queue="<q>",
>>>          project="<p>", cores=4, memory="2g")
>>>
>>> integrator.set_distribute(cluster)
>>> res = integrator.run_integration(n_iter)
```

Cluster systems not included in the dask library should be easy to implement following the same internal logic.

Open source for HEP

Where to obtain the code

Both VegasFlow and PDFFlow are open source and can be found at the N3PDF organization repository `github.com:N3PDF`

How to install

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`

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
```

Get PDF values in parallel

```
>>> from pdfflow import mkPDF
>>> pdf = mkPDF("NNPDF31_nlo_as_0118/0")
# Can be used within python code
>>> pdf.py_xfxQ2([21, 1, 2], [0.45], [91.**2]).numpy()
    array([0.01829599, 0.0357393 , 0.14706923])
>>> pdf.py_xfxQ2([21, 1, 2], [0.45], [91.**2])
<tf.Tensor: shape=(3,), dtype=float64, numpy=
    array([0.01829599, 0.0357393 , 0.14706923])>
# But it can also be used within a TensorFlow function
>>> x_arr = tf.constant([0.5, 0.2])
>>> q2_arr = tf.constant([91.**2, 173.**2])
>>> pdf.xfxQ2([1,21], x_arr, q2_arr)
<tf.Tensor: shape=(2, 2), dtype=float64, numpy=
    array([[0.01969848, 0.00930697],
    [0.21846276, 0.2335565 ]])>
>>> pdf.alphasQ2(float_me([91.**2, 173.**2]))
<tf.Tensor: shape=(2,), dtype=float64, numpy=
    array([0.11803883, 0.10762763])>
```

Summary

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

$\rightarrow$ Being competitive with CPU for MC simulations.

$\rightarrow$ Efforts in that direction

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

✓ Easy to use and interface.

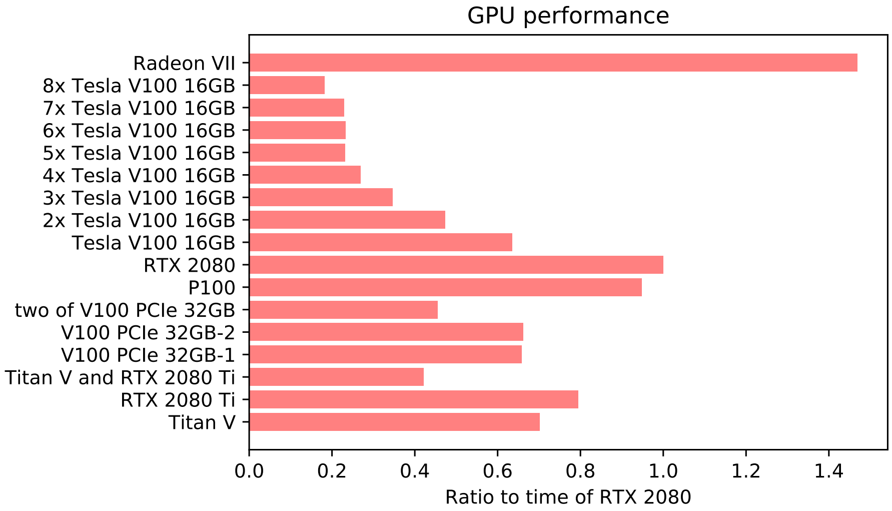
✓ Easy implementation of new-generation or NN-based integration algorithms.

Where to obtain the code

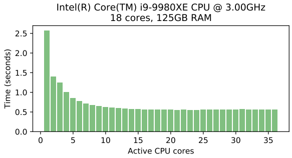
VegasFlow and PDFFlow are opensource and available at `github.com:N3PDF/pdfflow` and `github.com:N3PDF/VegasFlow`

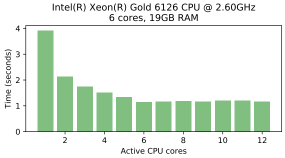
Thanks!

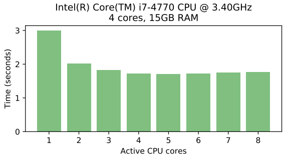
Benchmark on different GPUs

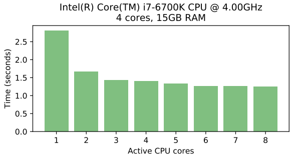


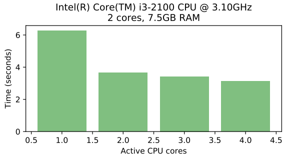
Benchmark on different CPUs

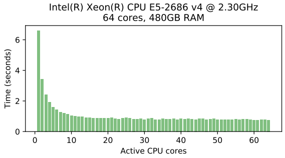

Intel(R) Core(TM) i9-9980XE CPU @ 3.00GHz
18 cores, 125GB RAM
Time (seconds)
Active CPU cores

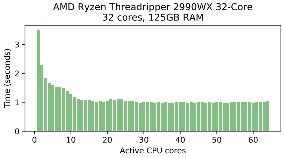

Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz
6 cores, 19GB RAM
Time (seconds)
Active CPU cores

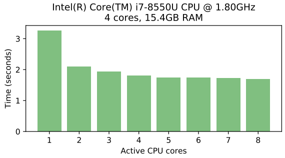

Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
4 cores, 15GB RAM
Time (seconds)
Active CPU cores

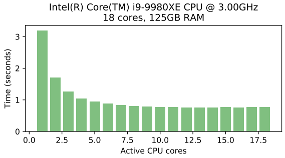

Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
4 cores, 15GB RAM
Time (seconds)
Active CPU cores


Intel(R) Core(TM) i3-2100 CPU @ 3.10GHz
2 cores, 7.5GB RAM
Time (seconds)
Active CPU cores


Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz
64 cores, 480GB RAM
Time (seconds)
Active CPU cores


AMD Ryzen Threadripper 2990WX 32-Core
32 cores, 125GB RAM
Time (seconds)
Active CPU cores


Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
4 cores, 15.4GB RAM
Time (seconds)
Active CPU cores


Intel(R) Core(TM) i9-9980XE CPU @ 3.00GHz
18 cores, 125GB RAM
Time (seconds)
Active CPU cores