Towards a framework for GPU event generation

Juan M. Cruz-Martinez - Cern TH Department
Event generators' and N(n)LO codes' acceleration, November 2023
Some points from yesterday…

These were well summarised in Danilo’s talk from yesterday:

- 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

however…

- many very important codes cannot even deal with the maintenance burden
- good code practise is a luxury
- since the important metric (papers?) is not always aligned with improving existing tools, people become experts in something else
- many of the efforts towards acceleration are done as a “side job”
- incremental improvements are a hard-sell, while more ambitious efforts don’t guarantee a career afterwards
Other points

• Cultivating (and keeping) expertise is important

• Keeping parallelism in mind from the very beginning (a bit late now?)

• Incremental improvements can precisely help us there

• Some of this knowledge can be highly transferable

• We should consider continuous testing, good code practices, documentation, etc not as a luxury but rather something as important and writing a paper that’s easy to read.

• We should aim for low entrance barriers: it is very difficult for a person to dedicate large amounts of time… maybe we should aim for many people dedicating some time
The ingredients of a Monte Carlo generator

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

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

integrator

histogramming / analysis

Parton Distribution Functions

phase space

matrix element

cuts
Monte Carlo integrals are highly parallelizable.

\[
\pi \approx 3.2
\]

\[
\pi \approx 3.16
\]
If they are so good, why are we not using them?

1. Lack of expertise? motivation? and (mostly) time
   - 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!

2. Lack of tools
   - There are many tools to help you with your MC calculation… as long as it is running on a CPU…
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
The phenomenologist (GPU-based) lack thereof

integrator

phase space

matrix element

Parton Distribution Functions

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


github.com/N3PDF/pdfflow

More about PDFFlow in Stefano Carraza’s talk later today

PDFflow - LHAPDF performances

NNPDF31_nlo_as_0118/0, flav = 1
GPU-aware integration wrapper: 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.

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

MadFlow time for 1M events

$pp \rightarrow t\bar{t}g$ (36 diagrams)
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

MadFlow time for 100k events

\[ pp \rightarrow t\bar{t}ggg \text{ (2604 diagrams)} \]
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.
Conclusions

- Full-fledged GPU Monte Carlo event generators required a great dedicated efforts
- Incremental improvements are not always appreciated…
- Having a small framework as a testing ground can be used to
  - Quickly test and benchmark possible ideas
  - Train students without paying a big “opportunity cost”
  - A computational version of a Toy Model

Thanks!