PROCEEDINGS OF SCIENCE

# Extending MadFlow: device-specific optimization

**Stefano Carrazza,**[a,b,c] **Juan M. Cruz-Martinez**[a,b,*] **and Gabriele Palazzo**[a,d]

[a] *TIF Lab, Dipartimento di Fisica, Università degli Studi di Milano and INFN Sezione di Milano*
*Via Celoria 16, 20133, Milano, Italy*

[b] *CERN, Theoretical Physics Department, CH-1211 Geneva 23, Switzerland*

[c] *Quantum Research Centre, Technology Innovation Institute, Abu Dhabi, UAE.*

[d] *Medical Physics, San Raffaele Scientific Institute, Milano, Italy*

*E-mail:* stefano.carrazza@mi.infn.it, juan.cruz.martinez@cern.ch, palazzo.gabriele@hsr.it

In this proceedings we demonstrate some advantages of a top-bottom approach in the development of hardware-accelerated code. We start with an autogenerated hardware-agnostic Monte Carlo generator, which is parallelized in the event axis. This allow us to take advantage of the parallelizable nature of Monte Carlo integrals even if we don't have control of the hardware in which the computation will run (i.e., an external cluster). The generic nature of such an implementation can introduce spurious bottlenecks or overheads. Fortunately, said bottlenecks are usually restricted to a subset of operations and not to the whole vectorized program. By identifying the more critical parts of the calculation one can get very efficient code and at the same time minimize the amount of hardware-specific code that needs to be written. We show benchmarks demonstrating how simply reducing the memory footprint of the calculation can increase the performance of a 2 → 4 process.

*41st International Conference on High Energy physics - ICHEP2022*
*6-13 July, 2022*
*Bologna, Italy*

[1] *Preprint numbers:*

[*] Speaker

## 1. Introduction and motivation

Research in high energy physics has historically been tied very closely to the development of high performance computing. This is still true today; obtaining physical predictions for some of the most accurate and precise calculations can take many CPU-years [1] due to the very complicate nature of the most interesting quantities. One example of a computationally expensive application in particle physics is event generators [2] which often make use of Monte Carlo methods.

In order to reduce the computing cost of a Monte Carlo integration several strategies can be followed, the most obvious of which is a simple (but effective) optimization of the existing code and algorithms, making the computation faster but otherwise exactly the same. Perhaps the most interesting and promising venue instead is to develop new algorithms, although this is also the most challenging path. Despite many attempts in recent years [3–6], the Vegas algorithm [7, 8] remains unbeaten in terms of use, specially when taking into account the many-dimensional space over which these integrals are calculated.

Another possibility, which can be seen as an intermediate path between the previous approaches, is to modify current algorithms to target different types of hardware, potentially enabling enormous speed-ups. This is something that has become possible only recently thanks to General Purpose GPU computing and the development of new hardware devices such as Tensor Processing Units (TPUs).

Based on familiar tools (tensors) and on an intuitive programming language (python), the techniques described in Ref. [9] allow physicists to write programs that can run on both CPUs and hardware accelerators, obtaining enormous speed-ups without having to deal with the peculiarities of any particular device or architecture. Furthermore, the code can be easily interfaced to other programs and libraries often used in HEP [10], allowing for great flexibility.

In Ref. [9] we introduced VegasFlow, a framework to facilitate the transition from CPU-based computations to GPU for developers familiarized with Monte Carlo integration. VegasFlow is a reimplementation of the original Vegas algorithm in a vectorized (or rather, *tensorized*) manner such that the events necessary for the calculation are computed in parallel. Since all events are parallel from the onset (instead of the parallelization of an existing sequential code) the usage of hardware accelerators, such as GPUs, becomes natural. In addition, the code is written using TensorFlow [11], a machine learning framework which already includes many primitives for everyday mathematical operations with kernels able to run in CPUs and GPUs of different manufacturers.

In Ref. [12] the aforementioned techniques were used to write PDFFlow, an adaptation of the well-known LHAPDF library [13] which exploits the massive capacity of modern GPUs for PDF interpolation. This is a crucial part of a simulation of proton collisions. As an example, in the same publication, a prototype of a Next-to-Leading Order calculation running entirely in a GPU was presented.

Finally, in Ref. [14] we presented MadFlow. MadFlow includes a Madgraph plugin which takes advantage of MG5_aMC's [15] great flexibility and extensibility to autogenerate matrix-elements completely *tensorized* in the event dimension, and thus ripe to be offloaded to a hardware accelerator.

In each of the previous steps the flexibility and user-friendliness of the code have been primed with respect to performance. Nevertheless, as it has been demonstrated in several benchmarks in the cited publications, the hardware-accelerated calculation has often been proven to be more efficient

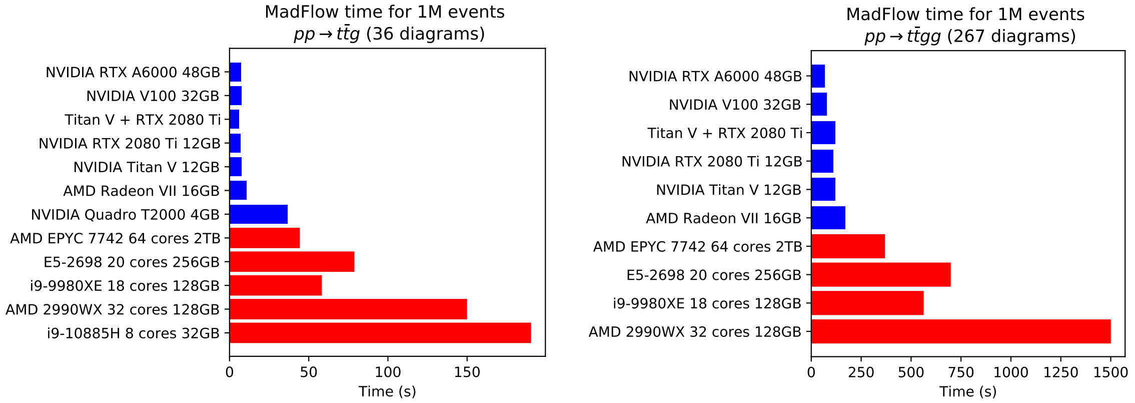

**Figure 1:** Benchmark of `MadFlow` running on different devices. CPUs at the bottom (red) and GPUs at the top (blue). All the GPUs perform better than any of the CPUs for the processes being considered.

than the CPU-based one (at equal cost and even without taking into consideration more specific optimizations). However, as it is usually the case, the price to pay for convenience and comfort is a high-level code that cannot truly take full advantage of the hardware as much as a low-level targeted implementation could. High-level code often introduces spurious bottlenecks which are not fundamental to the underlying algorithms.

In this proceedings we demonstrate how surgically eliminating some of these bottlenecks can further unlock the potential of hardware accelerators while keeping most of the convenience of having a generic codebase. This is akin to a top-bottom approach in which we optimize at low-level only the parts of the computation where it is actually needed. By reducing the scope of what needs to be implemented at low-level one can more easily leverage the features of the available hardware. In exchange, we lose the flexibility as different hardware architectures allow for different kinds of optimizations or programming languages.

## 2. Technical details

The technical details of `MadFlow` have been discussed elsewhere [14], the code is open source [16] and accessible at https://github.com/N3PDF/madflow. In Fig. 1 we recall some of the benchmarks performed in [14]. As it can be appreciated, the improvements on performance are less pronounced as the number of diagrams grows (see Table 1).

| Process | Diagrams | `MadFlow` CPU | `MadFlow` GPU | `MG5_aMC` |
|---|---|---|---|---|
| $pp \to t\bar{t}g$ | 36 | 57.84 $\mu$s | 7.54 $\mu$s | 93.23 $\mu$s |
| $pp \to t\bar{t}gg$ | 267 | 559.67 $\mu$s | 121.05 $\mu$s | 793.92 $\mu$s |

**Table 1:** Comparison of event computation time for `MadFlow` and `MG5_aMC`, using an Intel i9-9980XE with 18 cores and 128 GB of RAM for CPU simulation and the NVIDIA Titan V 12 GB for GPU simulation. We observe a factor of $\sim 8$ for the simpler calculation and only a factor of $\sim 4$ for the more complicated one.

The results of Table 1 do not come as a surprise, a more complex computation leads to a more frequent context-switching and memory transference, both of which are very expensive operations in hardware accelerators. This is in part driven by the growth of the memory footprint of the calculation. The resulting effect is that fewer events can be computed in parallel leading to a loss of efficiency. Since the performance of the calculation is connected to the number of diagrams, we will write for this exercise a low-level implementation of the matrix element using Cuda, due to the availability of GPUs manufactured by NVIDIA.

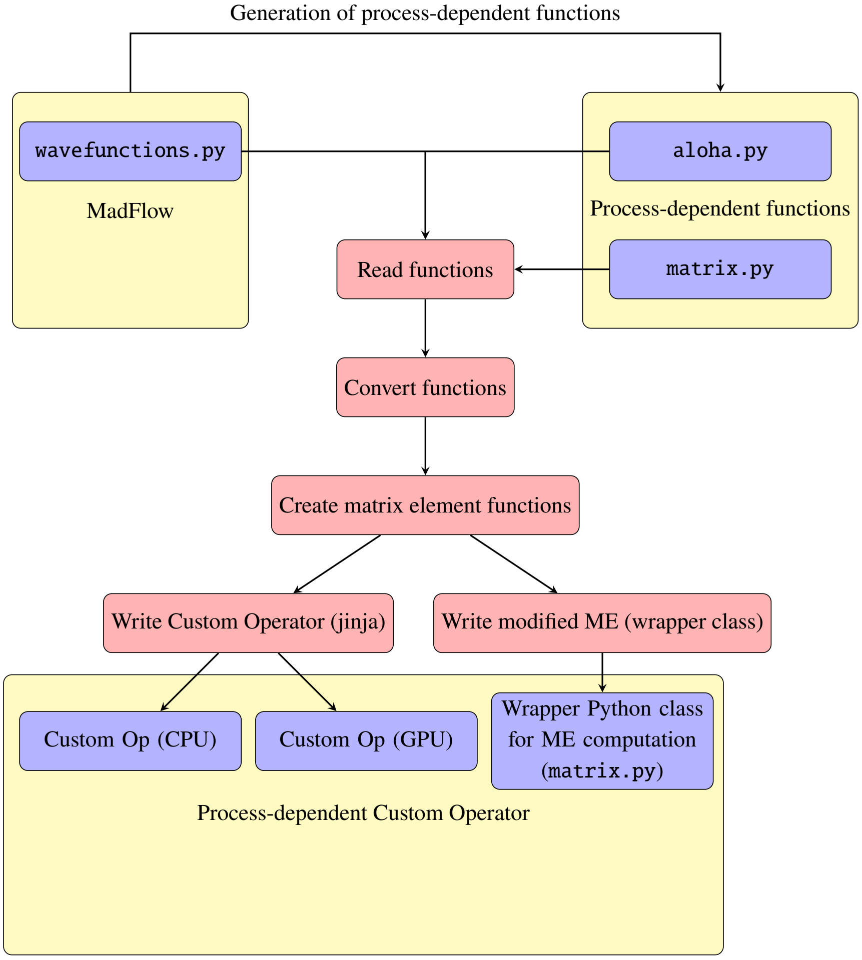
Generation of process-dependent functions
wavefunctions.py
MadFlow
aloha.py
Process-dependent functions
matrix.py
Read functions
Convert functions
Create matrix element functions
Write Custom Operator (jinja)
Write modified ME (wrapper class)
Custom Op (CPU)
Custom Op (GPU)
Wrapper Python class for ME computation (matrix.py)
Process-dependent Custom Operator

**Figure 2:** Architecture of the `MadFlow` framework, extended with the automatic generation of custom code written using the `jinja` template language with the `MG5_aMC` matrix element as input.

In order to maintain the number of changes to a minimum, the Cuda code is automatically transpiled from the `MadFlow` code (which is itself already parallelized on events). A schematic diagram of the structure of the framework is shown in Fig. 2. Only the necessary quantities (vertices, spinors or propagators) are generated as separated Cuda kernels. The communication between the CPU and GPU kernels is done utilizing the C++ interface provided by TensorFlow from which one can launch Cuda kernels normally.

The modules that will launch the GPU-kernels are wrapped as TensorFlow custom operators. Since we have not implemented the derivative of these operators, we have lost the possibility of taking the derivatives of the matrix element, however it would be trivial to extend the procedure to include that possibility.

## 3. Benchmarks

We now compare the performance of the default `MadFlow`, without device-specific optimization, with the new version. This option can be enabled by using the `--custom_op` flag.

An example on how a low-level interface gives the user a finer control of the calculation can be seen in Fig. 3. The computing architecture of NVIDIA GPUs divides the computation in blocks, each of these blocks is then further divided into threads. How the computation is spread on these computing units has a huge effect on the efficiency of the calculation.

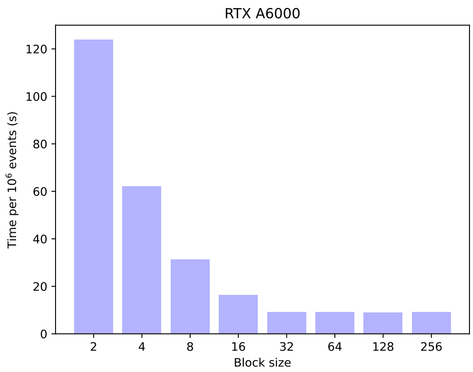

**Figure 3:** Time to run $10^6$ for a Leading Order $gg \longrightarrow t\bar{t}gg$ calculation using `MadFlow` as a function of the number of threads in each block. We observe how a bad choice of the block size can have a huge impact on the performance of the calculation.

By having direct access to the underlying hardware one might be able to outperform the more generic settings deduced automatically by a higher-level language. For instance, two different graphic cards of the same manufacturer might have very different characteristics and so a one-size fits-all approach might not take full advantage of the hardware. As seen in Fig. 3, there is a range of block-sizes that achieve similar performance, but this might depend on the ability of the GPU (or the underlying framework) to hide latency by for instance transferring memory while the previous block computation is still ongoing.

In Fig. 4 we compare the time it takes to compute two different processes on different hardware with and without custom code. We observe that in the simplest of the two calculations ($pp \longrightarrow t\bar{t}$) there is only a performance gain when using a lower end GPU, while for the high-end GPUs there is no appreciable loss of effi-

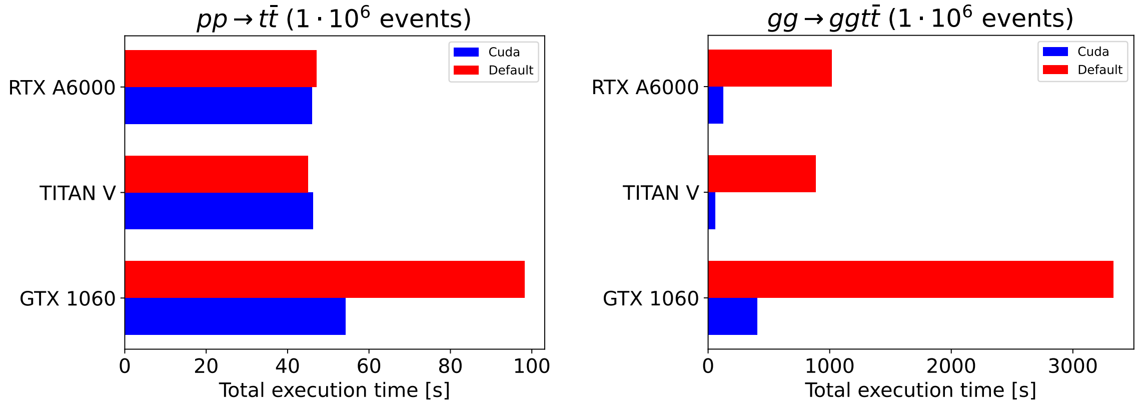

**Figure 4:** Benchmark of the default `MadFlow` (red) against the Cuda-extended version (blue) for the matrix element. In the left plot we observe no difference between the default autogenerated code and the custom operator written in Cuda for the better GPUs, while we find some difference for a low-end GPU. In the right plot instead we add two extra gluons to the process, increasing the number of diagrams being computed. In this case we find a much bigger difference for all the three GPUs tested.

ciency by using a more generic approach. Being able to have a finer control of the computation allow us to overcome some limitations of a lesser hardware. When the computation becomes more complicated we find that the generic approach becomes worse even for the high-end GPUs. It is interesting to note that they still perform much better than a low-end GPU, and, as a consequence, the gains of the custom code are also more important in the low-end case.

In conclusion, the choice of sacrificing the convenience of hardware-agnostic code depends on both the type of hardware available and the specific calculation at hand. The tools presented in this note aim to be flexible enough that both approaches are possible.

## Acknowledgments

SC and JCM are supported by the European Research Council under the European Union's Horizon 2020 research and innovation Programme (grant agreement number 740006).

## References

[1] D. Britzger et al., *NNLO interpolation grids for jet production at the LHC*, 2207.13735.

[2] J. M. Campbell et al., *Event Generators for High-Energy Physics Experiments*, in *2022 Snowmass Summer Study*, 3, 2022, 2203.11110.

[3] T. Müller, B. McWilliams, F. Rousselle, M. Gross and J. Novák, *Neural importance sampling*, *CoRR* **abs/1808.03856** (2018) [1808.03856].

[4] E. Bothmann, T. Janßen, M. Knobbe, T. Schmale and S. Schumann, *Exploring phase space with Neural Importance Sampling*, *SciPost Phys.* **8** (2020) 069 [2001.05478].

[5] C. Gao, S. Höche, J. Isaacson, C. Krause and H. Schulz, *Event Generation with Normalizing Flows*, *Phys. Rev. D* **101** (2020) 076002 [2001.10028].

[6] C. Gao, J. Isaacson and C. Krause, *i-flow: High-dimensional Integration and Sampling with Normalizing Flows*, *Mach. Learn. Sci. Tech.* **1** (2020) 045023 [2001.05486].

[7] G. P. Lepage, *A New Algorithm for Adaptive Multidimensional Integration*, *J. Comput. Phys.* **27** (1978) 192.

[8] G. P. Lepage, *VEGAS: AN ADAPTIVE MULTIDIMENSIONAL INTEGRATION PROGRAM*, 1980.

[9] S. Carrazza and J. M. Cruz-Martinez, *VegasFlow: accelerating Monte Carlo simulation across multiple hardware platforms*, *Comput. Phys. Commun.* **254** (2020) 107376 [2002.12921].

[10] S. Carrazza, J. M. Cruz-Martinez and C. Schwan, *Constructing PineAPPL grids on hardware accelerators*, *PoS* **LHCP2020** (2021) 057 [2009.11798].

[11] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro et al., *TensorFlow: Large-scale machine learning on heterogeneous systems*, 2015.

[12] S. Carrazza, J. M. Cruz-Martinez and M. Rossi, *PDFFlow: Parton distribution functions on GPU*, *Comput. Phys. Commun.* **264** (2021) 107995 [2009.06635].

[13] A. Buckley, J. Ferrando, S. Lloyd, K. Nordström, B. Page, M. Rüfenacht et al., *LHAPDF6: parton density access in the LHC precision era*, *Eur. Phys. J. C* **75** (2015) 132 [1412.7420].

[14] S. Carrazza, J. Cruz-Martinez, M. Rossi and M. Zaro, *MadFlow: automating Monte Carlo simulation on GPU for particle physics processes*, *Eur. Phys. J. C* **81** (2021) 656 [2106.10279].

[15] J. Alwall, R. Frederix, S. Frixione, V. Hirschi, F. Maltoni, O. Mattelaer et al., *The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations*, *JHEP* **07** (2014) 079 [1405.0301].

[16] J. M. Cruz-Martinez, M. Rossi, M. Zaro and S. Carrazza, *N3pdf/madflow: Automatic autogenerated accelerated mc generator*, June, 2021. 10.5281/zenodo.4954375.