



Accelerating Madgraph5_aMC@NLO via data parallelism (CPU vectorization and GPUs): status and lessons learnt

Andrea Valassi (CERN)

on behalf of the MG5AMC CUDACPP development team

Event Generators Acceleration Workshop – CERN, 13th November 2023

<https://indico.cern.ch/event/1312061>



Outline

- Motivation and overview
- Some results and future challenges for MG5AMC
 - Performance: throughout speedups on CPU SIMD and on GPUs for LO processes
 - Functionality: development status, usability for the experiments
 - Future prospects: NLO and beyond; collaborations with other MC teams
- Some lessons learnt for other MC generators
 - Applicability to other (existing and future) Monte Carlo generators
 - Do's and don't's
- Conclusions

Motivation and overview

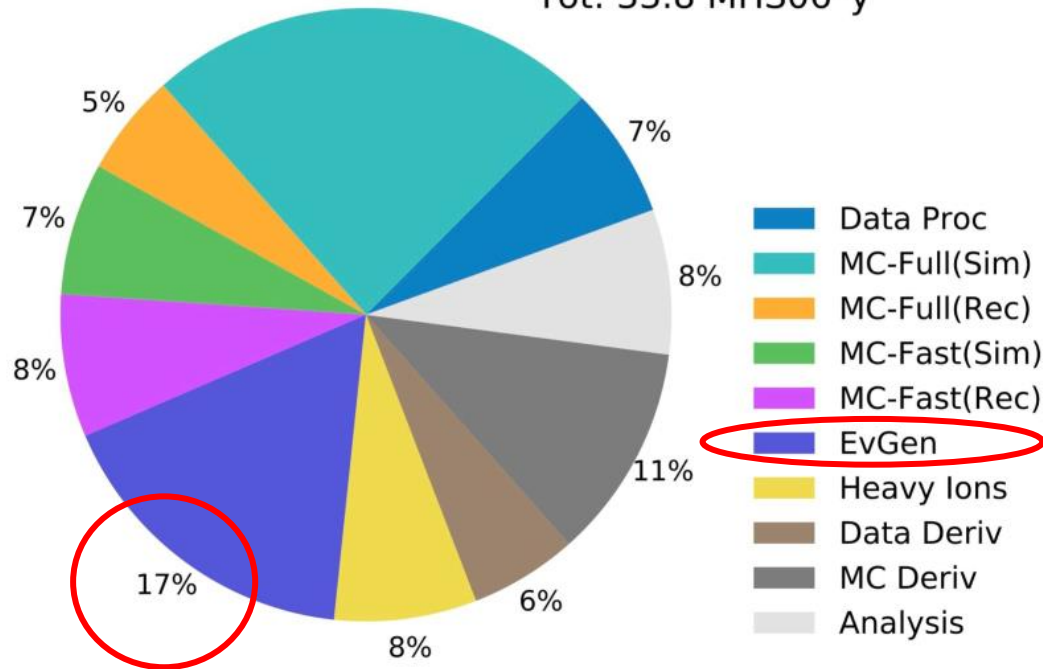
Event generators (1): why accelerate them?

ATLAS Software and Computing HL-LHC Roadmap, version 2.1

ATLAS Preliminary

2022 Computing Model - CPU: 2031, Conservative R&D

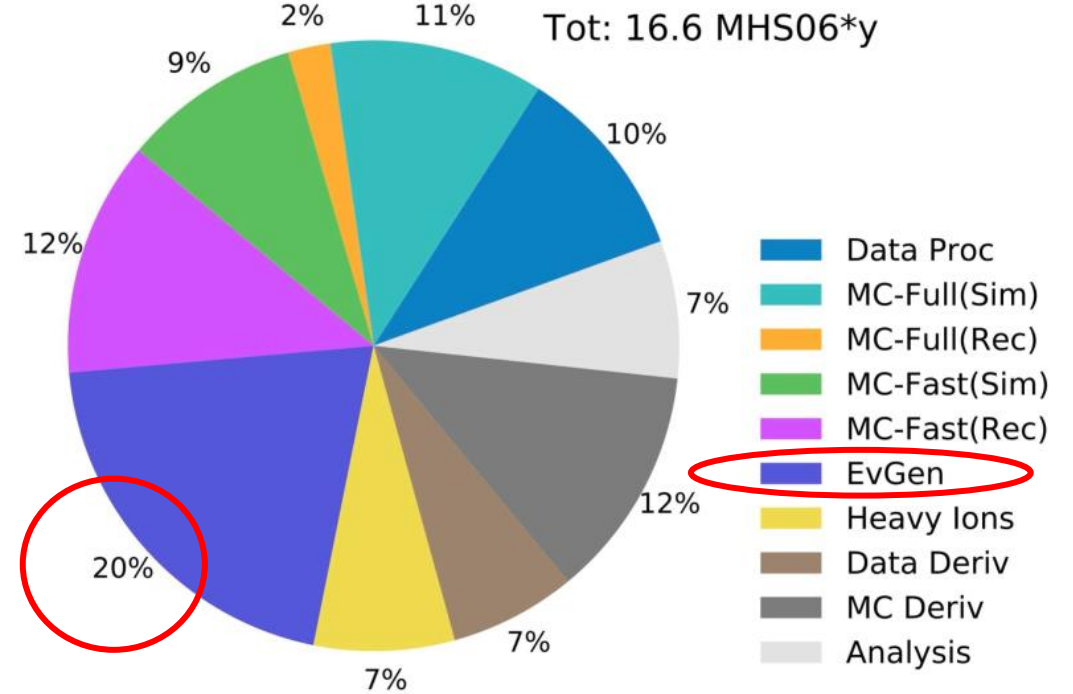
Tot: 33.8 MHS06*y



ATLAS Preliminary

2022 Computing Model - CPU: 2031, Aggressive R&D

Tot: 16.6 MHS06*y

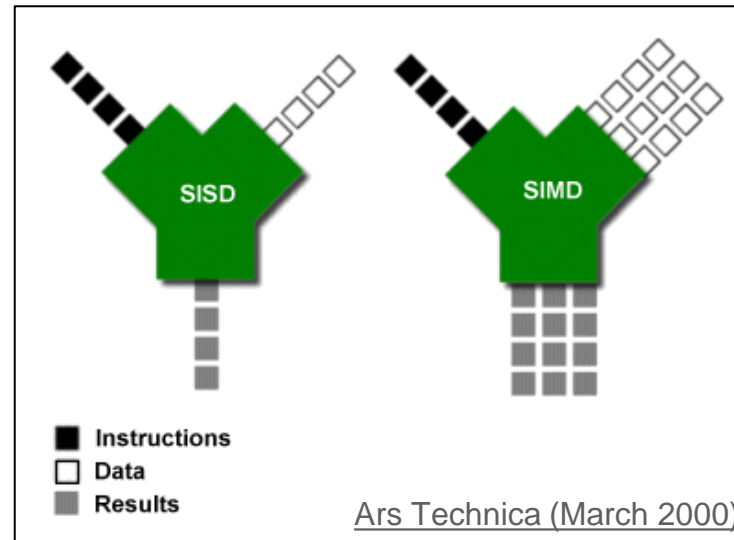


CERN-LHCC-2022-005

Sequential processing vs. Data-parallel processing

Sequential processing

Single Instruction Single Data:
1 input and 1 output per cycle
for a given instruction



Data-parallel processing (lockstep processing)

Single Instruction Multiple Data:
N inputs and N outputs per cycle
for the same instruction

Two hardware implementations
of essentially the same concept:

Vector CPUs – SIMD

More difficult to code
SOAs strictly needed
Need strict 100% lockstep

GPUs – “SIMT”

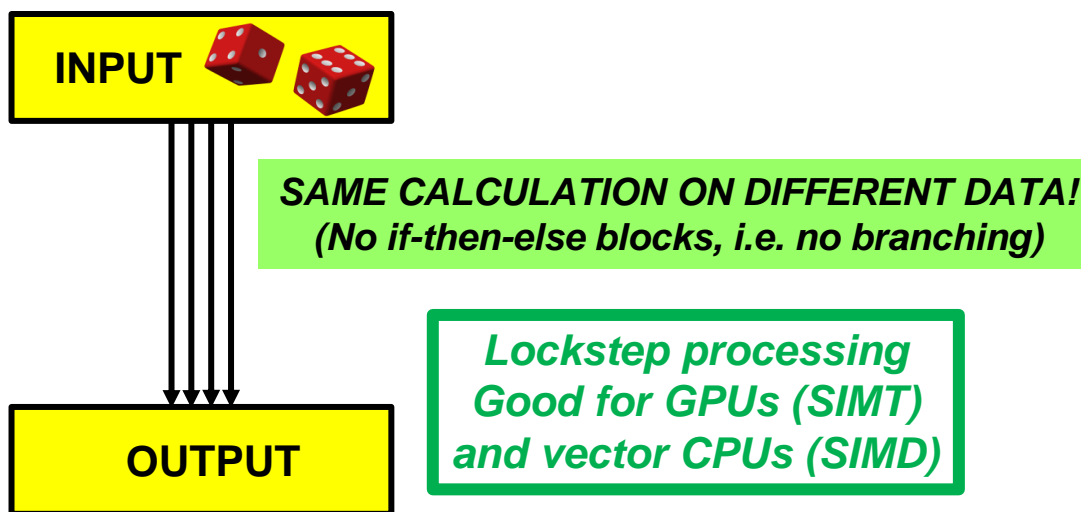
~Easier to code
SOAs not strictly needed
Tolerate lockstep <100%

In our work on MG5AMC “CUDACPP” we have targeted
data parallelism on both vector CPUs and GPUs
from the very beginning!

Note: task parallelism (multi-threading, multi-processing)
differs from data parallelism: it exploits a different dimension
of hardware parallelism (many CPU cores, many nodes...)

Event generators (2): why CPU vectorization and GPUs?

- **Vector CPUs and GPUs are widely available to HEP now for LHC processing (and later for FCC!?)**
 - Most of the CPUs in our computing Grid have at least AVX2 SIMD
 - GPUs are becoming more and more available to us especially at HPC centers
- ... **but they are generally very difficult to exploit in HEP software** ☹
 - Example: Monte Carlo detector simulation has a lot of stochastic branching (makes lockstep processing difficult)
- **Matrix element event generators, conversely, are ideal software workflows for SIMD and GPUs!**
 - Monte Carlo sampling of many data points → *Data parallelism with near-perfect lockstep processing!*



MG5AMC on GPUs and vector CPUs (the “CUDACPP” plugin)

C++ vectorization and CUDA in MG5AMC: the CUDACPP project

- MG5AMC CUDACPP development team →
(* Filip and Joergen left after the summer)

Stephan Hageboeck
Filip Optolowicz*
Stefan Roiser
Joergen Teig*
Andrea Valassi
Zenny Wettersten

Olivier Mattelaer



- *A collaboration* of theoretical physicists, experimental physicists and software engineers*
 - The project started in Q1 2020 (OM, SR, AV) in the context of the HSF event generator WG
 - Effort invested at that time in Louvain and in CERN IT's Understanding Performance team (thanks Markus Schulz!)
 - *See Danilo's slides for more comments on why this is necessary but also challenging



<https://doi.org/10.1007/s41781-021-00055-1>

For more details...

- Our work on MG5AMC CUDACPP is described in the [vCHEP2021](#), [ICHEP2022](#) and [ACAT2022](#) proceedings
 - And in the upcoming CHEP2023 proceedings (Stephan's [talk1](#), Zenny's [talk2](#))
 - See also the [Computing Accelerator Forum](#) (Feb 2023) talk for much more extensive details


EPJ Web of Conferences **251**, 03045 (2021) <https://doi.org/10.1051/epjconf/202125103045>
CHEP 2021

Design and engineering of a simplified workflow execution for the MG5aMC event generator on GPUs and vector CPUs

Andrea Valassi^{1,*}, *Stefan Roiser*¹, *Olivier Mattelaer*², and *Stephan Hageboeck*¹

¹CERN, IT-SC group, Geneva, Switzerland
²Université Catholique de Louvain, Belgium

<https://doi.org/10.1051/epjconf/202125103045>

 PROCEEDINGS OF SCIENCE

POs (ICHEP2022) 212

Developments in Performance and Portability for
MadGraph5_aMC@NLO

Andrea Valassi,^{a,*} Taylor Childers,^b Laurence Field,^a Stefan Hageböck,^a Walter Hopkins,^b Olivier Mattelaer,^c Nathan Nichols,^b Stefan Roiser^a and David Smith^a

<https://doi.org/10.22323/1.414.0212>

Speeding up Madgraph5_aMC@NLO through CPU vectorization and GPU offloading: towards a first alpha release

A Valassi¹, T Childers², L Field¹, S Hageböck¹, W Hopkins², O Mattelaer³, N Nichols², S Roiser¹, D Smith¹, J Teig¹, C Vuosalo⁴, Z Wettersten¹

<https://arxiv.org/abs/2303.18244>

- These also describe the work of our US and CERN collaborators on SYCL, Kokkos and Alpaka abstraction layers
 - Largely based on the developments and progress in the CUDACPP project, which will be the focus of this talk

Madgraph5_aMC@NLO (MG5aMC)

- One of the workhorses for event generation in ATLAS and CMS!

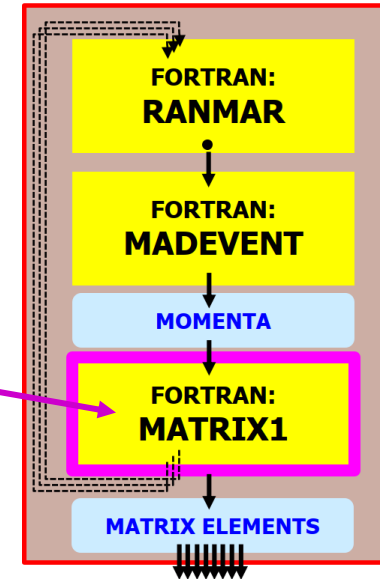
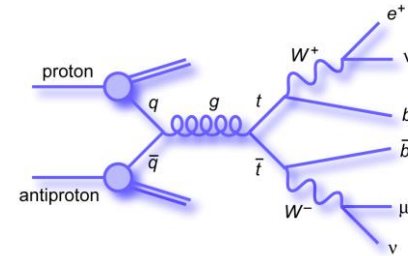
PUBLISHED FOR SISSA BY SPRINGER

RECEIVED: May 20, 2014
ACCEPTED: June 25, 2014
PUBLISHED: July 17, 2014

The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations

J. Alwall,^a R. Frederix,^b S. Frixione,^b V. Hirschi,^c F. Maltoni,^d O. Mattelaer,^d
H.-S. Shao,^e T. Stelzer,^f P. Torrielli^g and M. Zaro^{h,i}

[https://doi.org/10.1007/JHEP07\(2014\)079](https://doi.org/10.1007/JHEP07(2014)079)



- MG5aMC production version is in Fortran
 - Software outer shell: Madevent (random sampling, integration and event generation + I/O, multi-jet merging...)
 - Software inner core: Matrix Element (ME) calculation code, automatically generated for each physics process
 - **Matrix Element calculations take 95%+ of the CPU time for complex processes** (e.g. $gg \rightarrow t\bar{t}ggg$)
 - **And ME calculations are precisely one component that can be “easily” accelerated on GPUs and vector CPUs...**

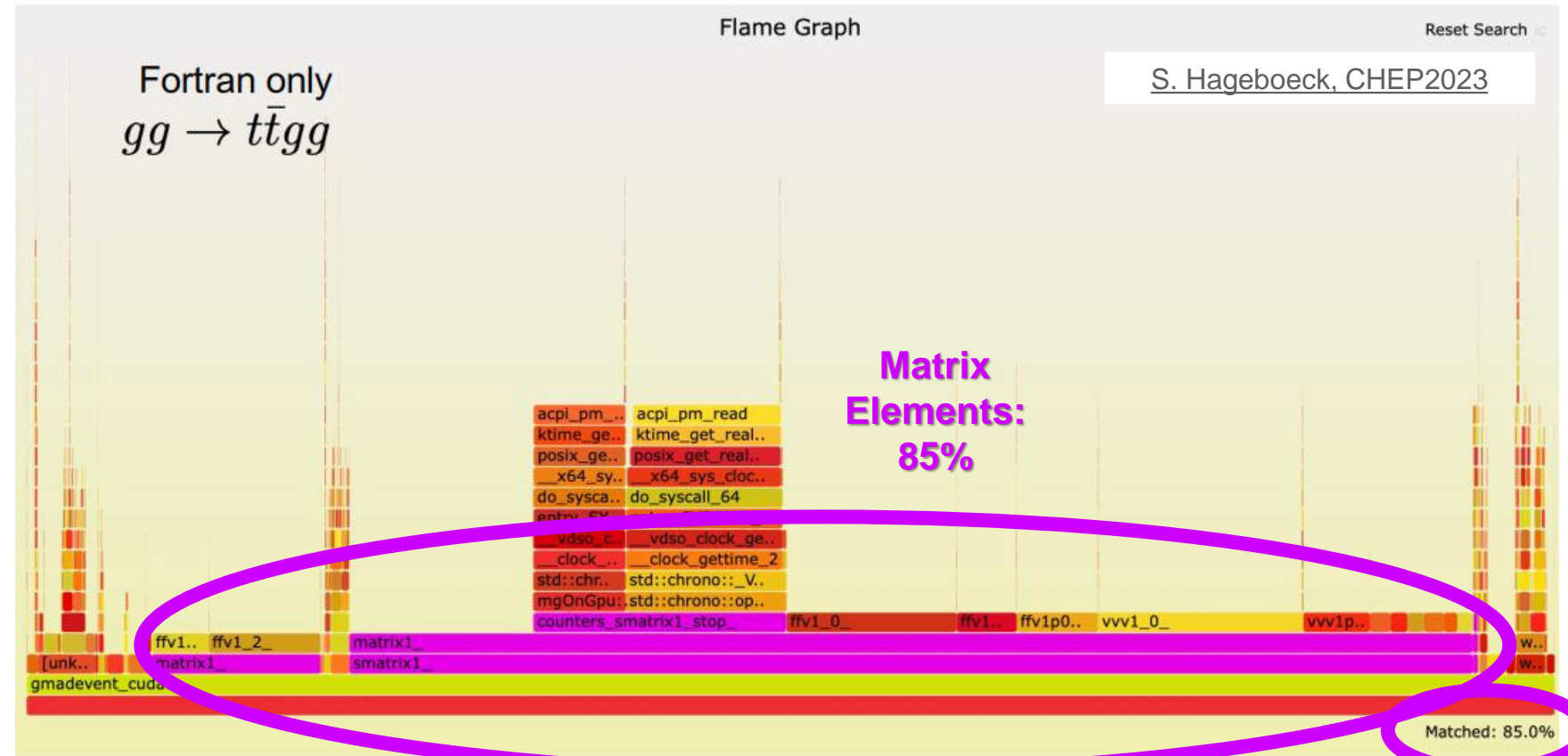
MG5AMC before acceleration (Fortran madevent + Fortran ME)

- In the current production MG5AMC in Fortran, the matrix element calculation is the bottleneck

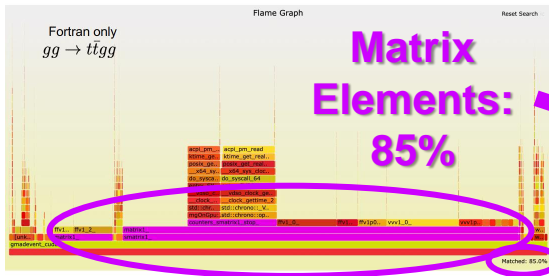
- Feynman diagrams
- Color sum (color matrix)

- We are lucky!

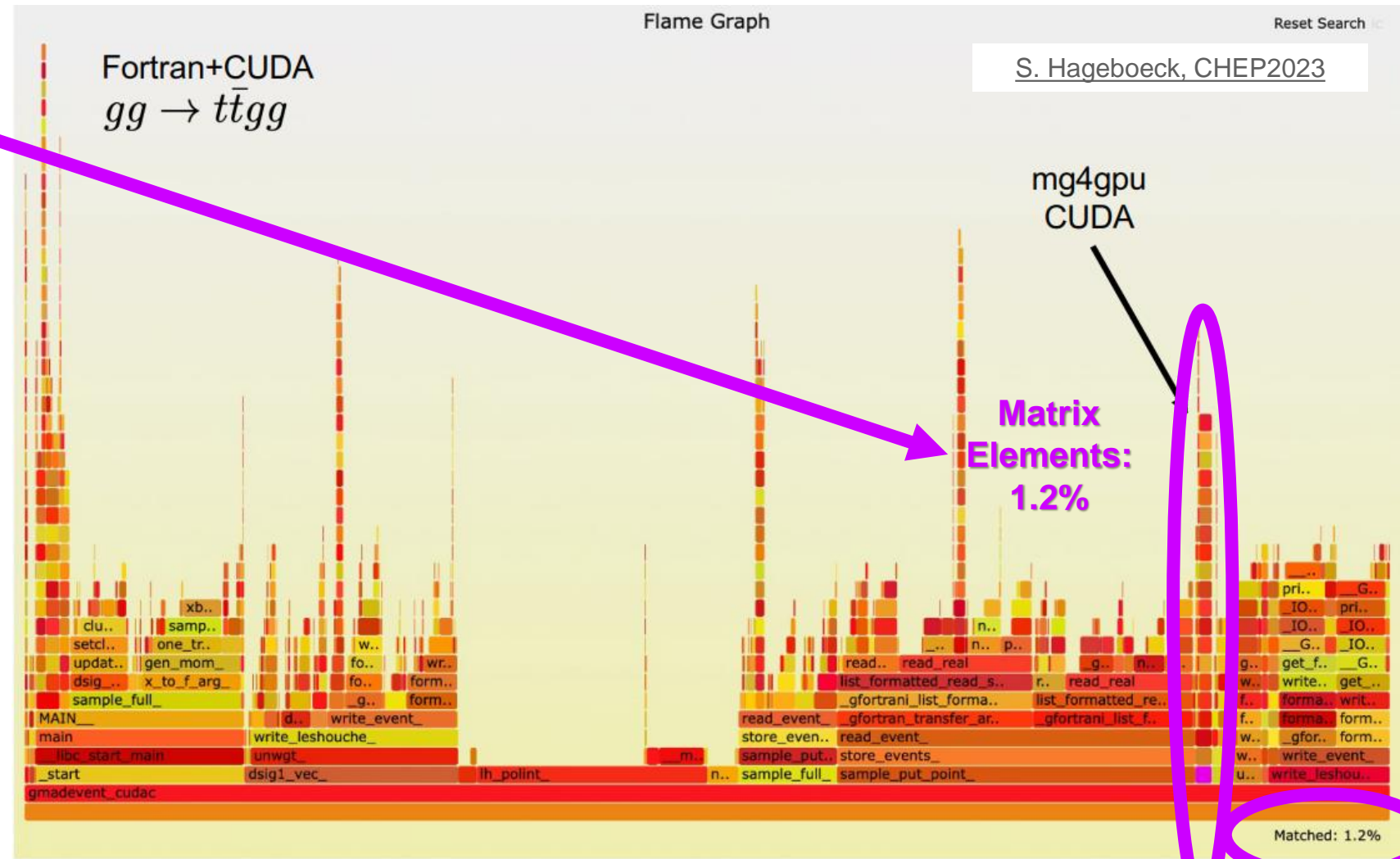
- *The fraction of time in the ME calculation increases with process complexity!*
- The MS calculation is the easiest to parallelize!



MG5AMC after acceleration (Fortran madevent + CUDA ME)

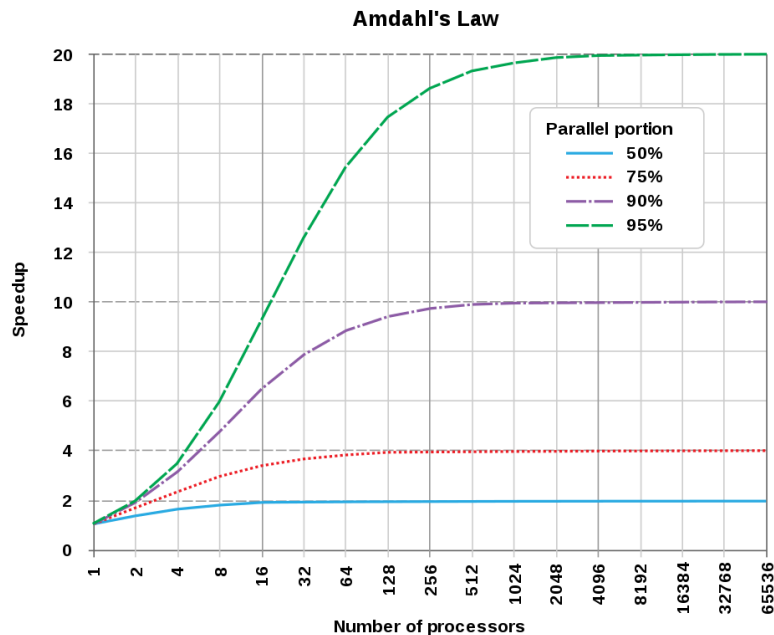


- On GPUs we speed up the ME so much that previously unimportant components become the bottleneck!
 - sampling (random numbers to momenta), unweighting, pdf's...
 - we are also working on speeding these up now!
- As predicted by *Amdahl's law*



Amdahl's law

- The matrix element calculation is now the bottleneck (e.g. >95% for $gg \rightarrow t\bar{t}gg$) in Fortran Madgraph
 - But the remaining <5% may fast become the bottleneck if you accelerate the matrix element by many factors!
- *Amdahl's law: if the parallelizable part takes a fraction of time p , the maximum speedup is $1/(1-p)$*
 - If the MadEvent overhead takes 5%, the maximum speedup is only 20 even if your GPU speedup s is 1000!

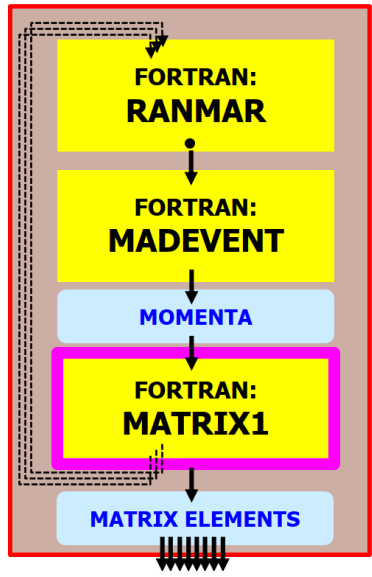


$$\lim_{s \rightarrow \infty} S_{\text{latency}}(s) = \frac{1}{1-p}$$

https://en.wikipedia.org/wiki/Amdahl%27s_law

MG5aMC: old and new architecture designs

OLD MADEVENT
(NOW: LHC PROD)
 SINGLE-EVENT API

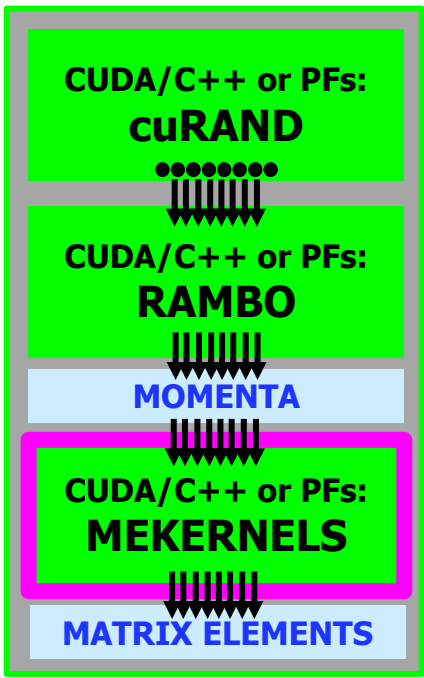


*MATRIX ELEMENT:
 CPU BOTTLENECK
 IN OLD MADEVENT*

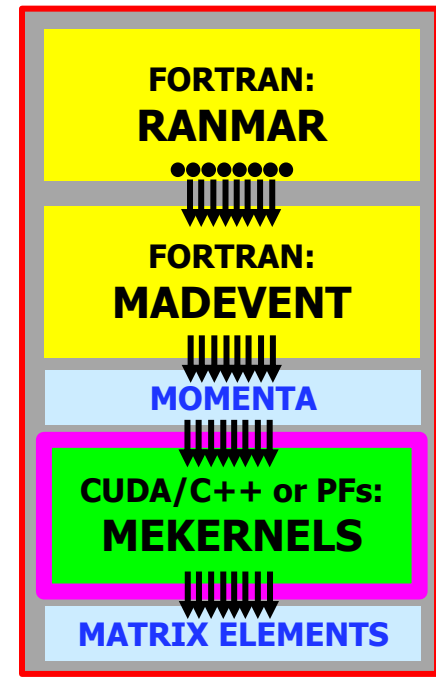
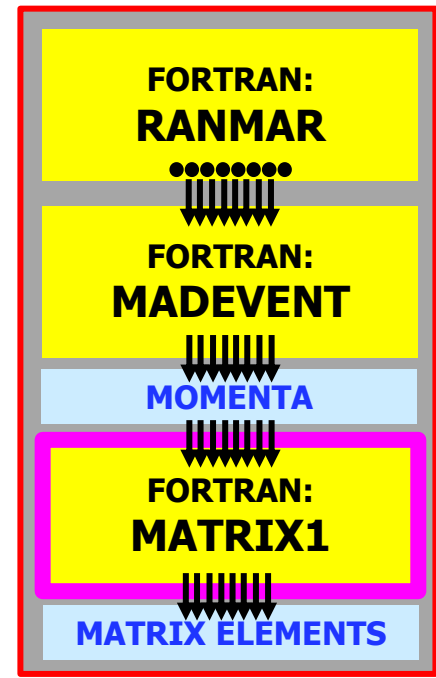
First we developed the new ME engines in standalone applications

Then we modified the existing all-Fortran MadEvent into a multi-event framework and we injected the new MEs into it

1. STANDALONE (TOY APPLICATIONS) MULTI-EVENT API



2. NEW MADEVENT (GOAL: LHC PROD) MULTI-EVENT API



(Amdahl...)
*SCALAR:
 NEW BOTTLENECK?*
*PARALLEL:
 MUCH FASTER!*

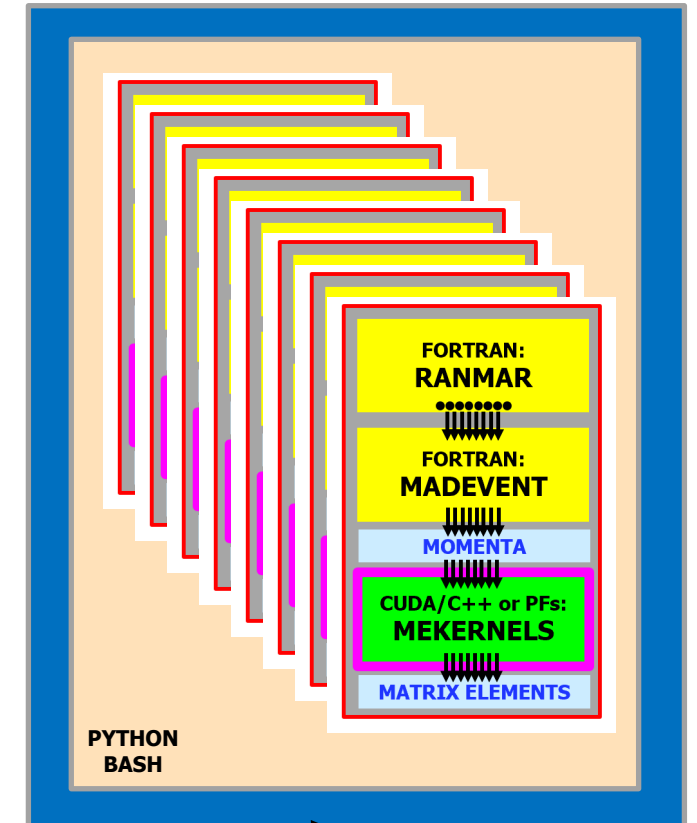
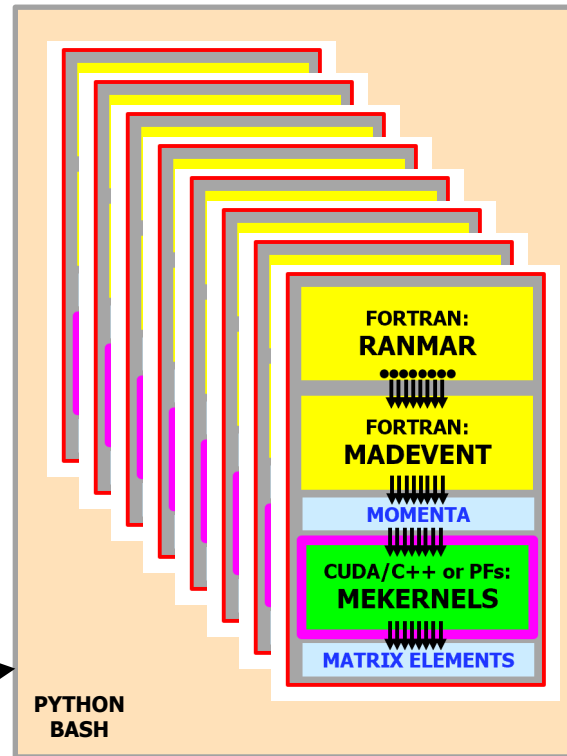
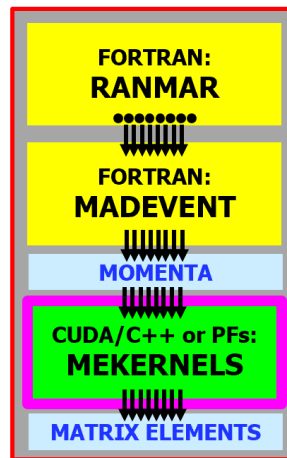
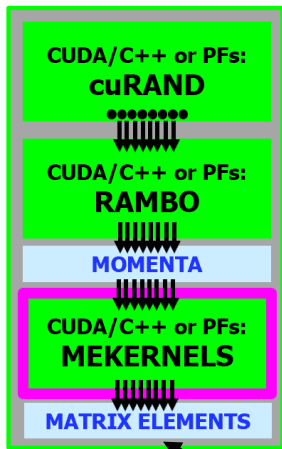
MG5AMC+cuDacpp: CUDA/C++, Fortran, bash, python...

4. COMPLETE USER WORKFLOW
(CODEGEN + N x APPLICATIONS)
generate.. output.. launch
BEING DEVELOPED/TESTED (since Aug 2023)

3. MADEVENT
(N x APPLICATIONS)
./bin/generate_events
BEING TESTED (since Jun 2023)

1. STANDALONE
TOY APPLICATION
OK! (2020-2021)

2. MADEVENT
(ONE APPLICATION)
OK! (2022)



MG5AMCNLO GITHUB
+ MADGRAPH4GPU GITHUB

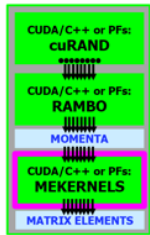
MG5AMCNLO GITHUB
+ CUDACPP GITHUB (PLUGIN)

We are now somewhere in between 3 and 4
(and this is what CMS is testing)

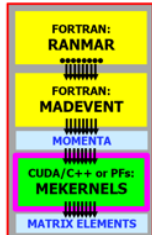
Test driven development

- I personally think that **writing tests is as important as (more important than?) writing implementation code!**
- At each stage of development we have been adding new tests – and we still run them (manually and/or in the CI)
 - Standalone applications: use hardcoded random seeds, compare momenta and MEs to reference files (googletest)
 - One madevent application: *use the same random seeds, compare cross sections and LHE files for Fortran/C++/CUDA MEs*
 - Require ~bit-by-bit equal results (within numerical precision), **this is much more than statistical comparisons!**
 - Full workflow with many madevent applications (under development): compare overall cross sections and LHE files as above
 - Full workflow including code generation: similar tests as above, regenerating physics process from the cudacpp plugin

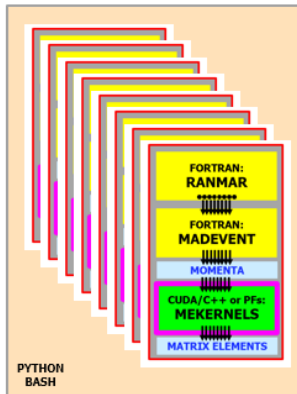
1. STANDALONE TOY APPLICATION
OK! (2020-2021)



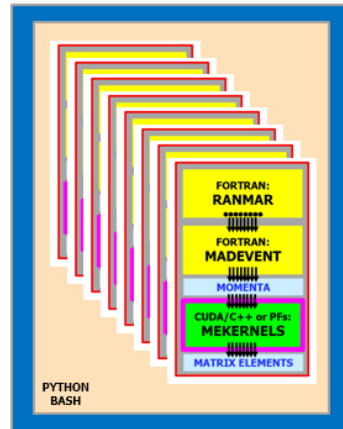
2. MADEVENT (ONE APPLICATION)
OK! (2022)



3. MADEVENT (N x APPLICATIONS)
/bin/generate_events
BEING TESTED (since Jun 2023)



4. COMPLETE USER WORKFLOW (CODEGEN + N x APPLICATIONS)
generate.. output.. launch
BEING DEVELOPED/TESTED (since Aug 2023)



Test a large phase space of development environments!

- Different physics processes
- Different vectorization scenarios
- Different floating point precisions
- Different compilers and O/S
- ...

MadEvent with vectorized C++ for $gg \rightarrow t\bar{t}gg$ (on a single CPU core)

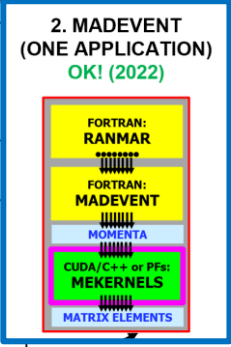
Compute Accelerator Forum, February 2023
<https://indico.cern.ch/event/1207838>

ACAT2022

madevent

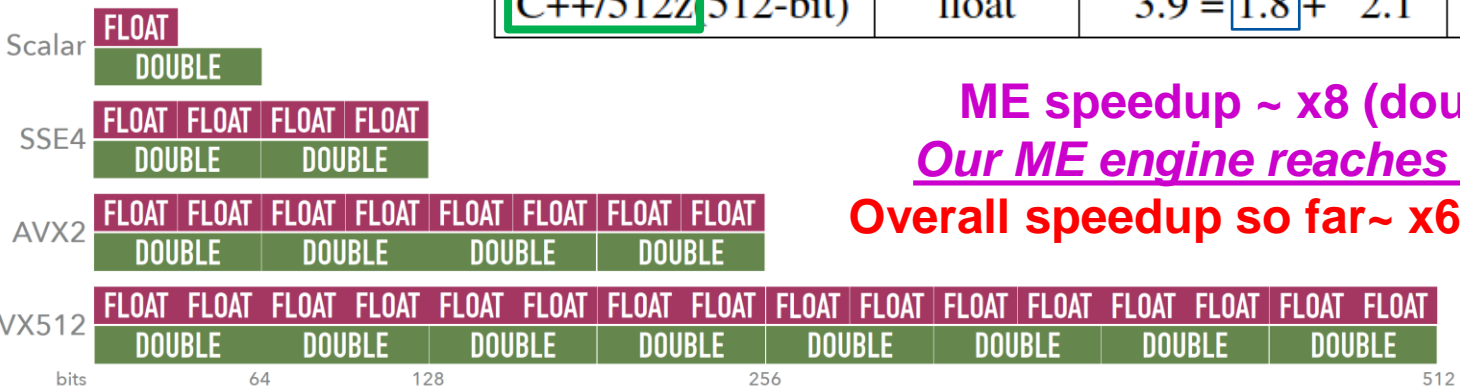
standalone

$gg \rightarrow t\bar{t}gg$	MEs precision	$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]	
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)	—
C++/none(scalar)	double	37.8 = 1.7 + 36.0	2.17E3 (x1.0)	2.28E3 (x1.0)	2.37E3
C++/sse4(128-bit)	double	19.4 = 1.7 + 17.8	4.22E3 (x1.9)	4.62E3 (x2.0)	4.75E3
C++/avx2(256-bit)	double	9.5 = 1.7 + 7.8	8.63E3 (x3.9)	1.05E4 (x4.6)	1.09E4
C++/512y(256-bit)	double	8.9 = 1.8 + 7.1	9.29E3 (x4.2)	1.16E4 (x5.0)	1.20E4
C++/512z(512-bit)	double	6.1 = 1.8 + 4.3	1.35E4 (x6.1)	1.91E4 (x8.3)	2.06E4
C++/none(scalar)	float	36.6 = 1.8 + 34.9	2.24E3 (x1.0)	2.35E3 (x1.0)	2.45E3
C++/sse4(128-bit)	float	10.6 = 1.7 + 8.9	7.76E3 (x3.6)	9.28E3 (x4.1)	9.21E3
C++/avx2(256-bit)	float	5.7 = 1.8 + 3.9	1.44E4 (x6.6)	2.09E4 (x9.1)	2.13E4
C++/512y(256-bit)	float	5.3 = 1.8 + 3.6	1.54E4 (x7.0)	2.30E4 (x10.0)	2.43E4
C++/512z(512-bit)	float	3.9 = 1.8 + 2.1	2.10E4 (x9.6)	3.92E4 (x17.1)	3.77E4



512y = AVX512, ymm registers
 512z = AVX512, zmm registers

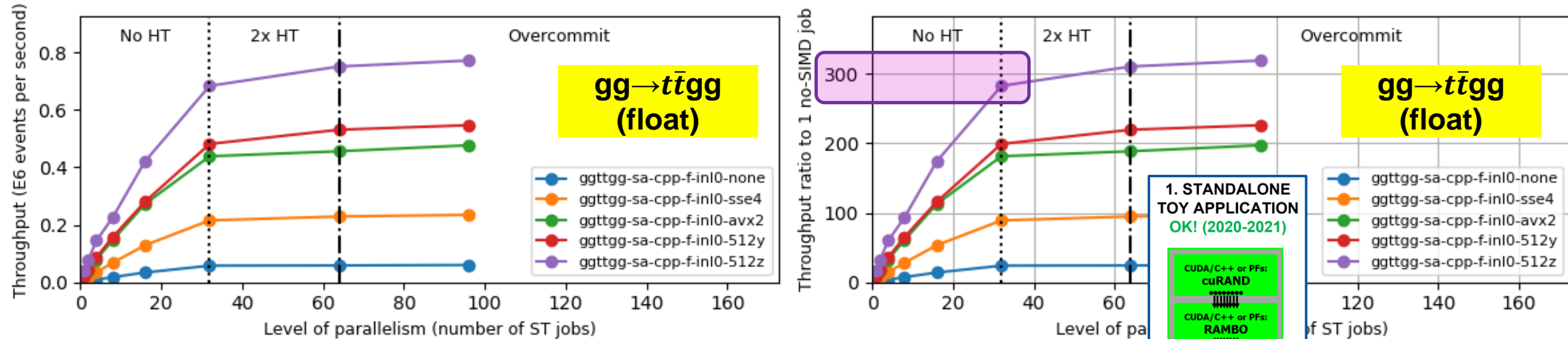
The latter is only better on nodes with 2 FMA units (here an Intel Gold 6148)



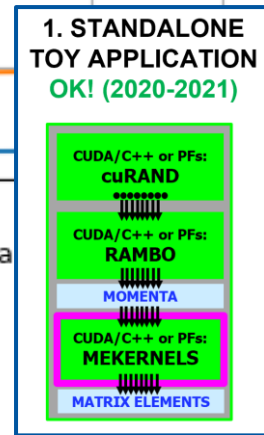
ME speedup ~ x8 (double) and x16 (float) over scalar Fortran
Our ME engine reaches the maximum theoretical SIMD speedup!
Overall speedup so far ~ x6 (double) and x10 (float) over scalar Fortran (Amdahl's law)

ME throughput in C++ for $gg \rightarrow t\bar{t}gg$ (on all the cores of a CPU)

ggttgg check.exe scalability on "bmk6130" (2x 16-core 2.1GHz Xeon Gold 6130 with 2x HT) for 10 cycles



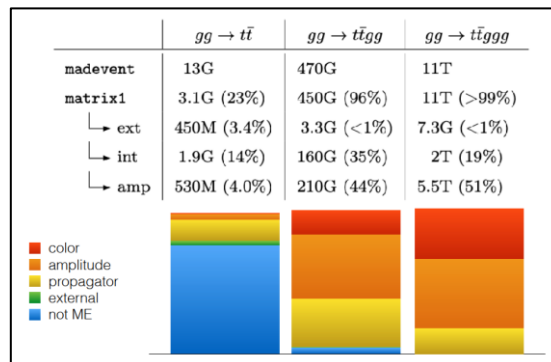
- Previous tables for SIMD speedups on C++ were for a single CPU core
- **Large SIMD speedups are also confirmed when all CPU cores are used**
 - AVX512/zmm speedup of x16 over no-SIMD for a single core slightly decreases to ~x12 on a full node (clock slowdown?)
 - *Overall speedup on 32 physical cores (over no-SIMD on 1 core) is around 280 (maximum would be 16x32=512)*
 - Aggregate MEs throughput from many identical processes using the standalone application
 - (HEP-workload Docker container from the HEPIX Benchmarking WG)



(This addresses the question by Liz earlier this afternoon)

Floating point precision

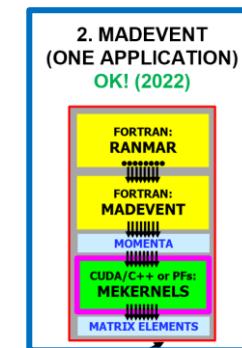
- Previous slides: *our vectorized C++ on CPUs is 2x faster in single-precision than in double-precision*
 - In a 512-bit register you fit 16 (4-byte) floats but only 8 (8-byte) doubles
- Next slide: *our CUDA implementation on V100 GPUs is also 2x faster for floats than for doubles*
 - On data-center NVidia GPUs (e.g. V100 or A100), you have twice as many FLOPs in float as in double
 - Note that lower-end GPUs (e.g. T4) have very limited double-precision FLOPs...
- But *single-point precision is not enough for physics: numerical instabilities* (e.g. in Feynman diagrams)
 - It would be useful to study if these instabilities can be worked around – anyone interested? 😊
 - We had a closer look at the source of these instabilities with the CADNA tool (see later)
 - **Alternative: we prototyped a “mixed-precision” calculation (double for Feynman, float for color matrix)**
 - Color sum is the largest CPU time consumer for complex processes, but can be done with floats!



O. Mattelaer,
this workshop

MadEvent/CUDA for $gg \rightarrow t\bar{t}ggg$

CUDA grid size		ACAT2022		madevent		standalone	
		8192		16384			
$gg \rightarrow t\bar{t}ggg$	MES precision	$t_{TOT} = t_{Mad} + t_{MES}$ [sec]		N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MES} [MEs/sec]		
Fortran	double	1228.2 = 5.0 + 1223.2		7.34E1 (=1.0)	7.37E1 (=1.0)	—	—
CUDA	double	19.6 = 7.4 + 12.1		4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)
CUDA	float	11.7 = 6.2 + 5.4		7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)
CUDA	mixed	16.5 = 7.0 + 9.6		5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)



We are lucky! The more complex the physics process, the lower the relative overhead from the scalar Fortran MadEvent - here only 0.5%
Amdahl's law limits the overall speedup to x200 (parallelizable p=0.5%), and we achieve x60 (double) or x100 (float) in the overall speedup!

One technicality about mixed precision (to answer Peter's question this afternoon):
 mixing double and float vectors implies some transition moments
 where you merge two 4-double vectors into one 8-float vector, and/or viceversa

- ▶ Computers sometimes lie about floating-point numbers
- ▶ [CADNA](#) is a library with special floating-point types to measure precision and instabilities in C++ and Fortran
- ▶ Each number knows its current precision
- ▶ CADNA counts unstable operations
- ▶ See [seminar at CERN](#)

<https://indico.cern.ch/event/1264290/>

▶ $P(x,y) = 9x^4 - y^4 + 2y^2$

Without CADNA:
 $P(10864,18817) = 2.0000000000000000$ (exact value: 1)
 $P(1/3,2/3) = 0.8024691358024691$

With CADNA:
 $P(10864,18817) = @.0$ (exact value: 1)
 $P(1/3,2/3) = 0.802469135802469E+000$

```

0 UNSTABLE DIVISION(S)
0 UNSTABLE POWER FUNCTION(S)
0 UNSTABLE MULTIPLICATION(S)
0 UNSTABLE BRANCHING(S)
0 UNSTABLE MATHEMATICAL FUNCTION(S)
0 UNSTABLE INTRINSIC FUNCTION(S)
2 UNSTABLE CANCELLATION(S)
    
```



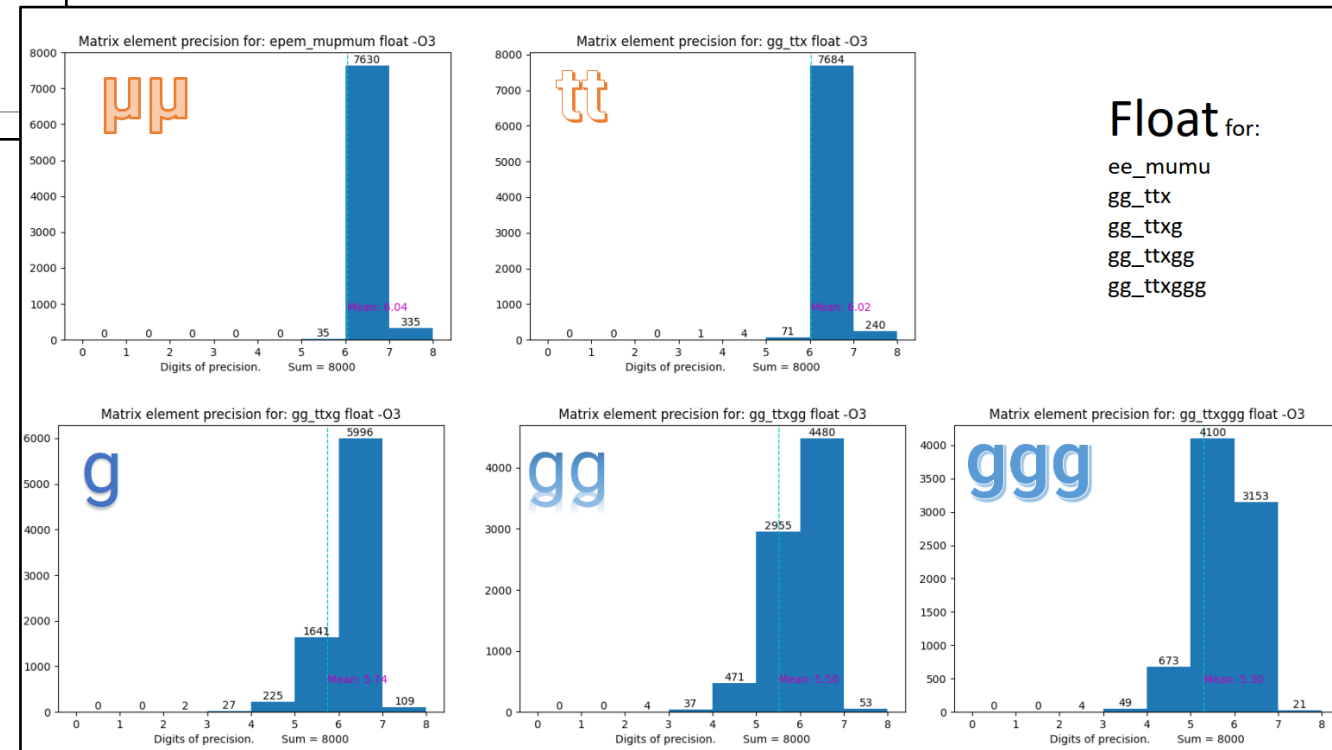
Numerical precision: CADNA (can we use floats instead of doubles?)

F. Optolowicz, CERN EP-SFT meeting 21 Aug 2023

STEPHAN HAGEBOECK - MG5_AMC MEETING 2023

S. Hageboeck, Gargnano meeting 18 Sep 2023

- Application to MG5AMC CUDACPP:
 - assess precision of the ME calculation (when using floats: down to 3 significant digits in gg to ttgg)
 - understand where in the code the precision is lost (typically, cancellations subtracting large terms, one example being heavily suppressed helicities)



Preliminary results for the full workflow (June 2023)

```
grep ELAPSED `ls -tr tlau/logs_ggtt*/*.txt`
tlau/logs_ggtt_CUDA/output.txt:ELAPSED: 24 seconds
tlau/logs_ggtt_FORTRAN/output.txt:ELAPSED: 23 seconds
tlau/logs_ggtt_CPP/output.txt:ELAPSED: 22 seconds
tlau/logs_ggttg_CUDA/output.txt:ELAPSED: 35 seconds
tlau/logs_ggttg_FORTRAN/output.txt:ELAPSED: 49 seconds
tlau/logs_ggttg_CPP/output.txt:ELAPSED: 36 seconds
tlau/logs_ggttgg_CUDA/output.txt:ELAPSED: 116 seconds
tlau/logs_ggttgg_FORTRAN/output.txt:ELAPSED: 857 seconds
tlau/logs_ggttgg_CPP/output.txt:ELAPSED: 280 seconds
tlau/logs_ggttggg_CUDA/output.txt:ELAPSED: 2705 seconds
tlau/logs_ggttggg_FORTRAN/output.txt:ELAPSED: 57322 seconds
tlau/logs_ggttggg_CPP/output.txt:ELAPSED: 17034 seconds
```

- On the most complex gg to ttgg
- **CPP with “512y” SIMD**
 - around **x 3.4 faster than FORTRAN**
- **CUDA (V100 GPU vs 4-core CPU)**
 - around **x 21 faster than FORTRAN**
 - (was ~ x 60 over a single CPU core)

~ Same physics results in FORTRAN, CUDA, CPP from the same random number (some final tests underway...)

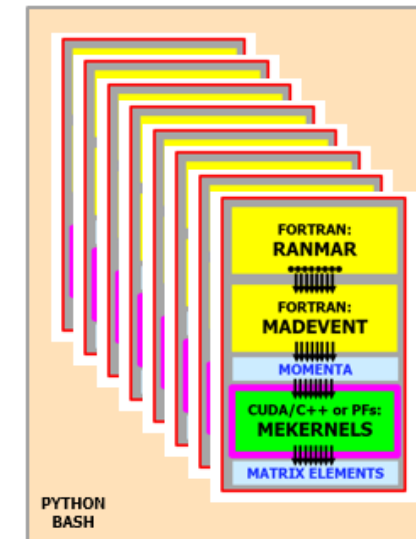
Note: Fortran here is NOT what the LHC experiments are using

- It has a multi-event API
- It has -O3 -ffast-math

TO BE TUNED!
We think we could do even better...

For more recent and more realistic results, see the CMS talk! (thanks Sapta, Jin, Robert for your collaboration!)

3. MADEVENT
(N x APPLICATIONS)
./bin/generate_events
BEING TESTED (since Jun 2023)



WANT TO GIVE IT A TRY?

PLEASE CONTACT US IN CASE OF
PROBLEMS / REQUESTS / SUGGESTIONS

```
$ git clone --branch garda23 --recurse-submodules https://github.com/madgraph5/madgraph4gpu.git  
$ cd madgraph4gpu/MG5aMC/mg5amcnlo  
$ ./bin/mg5_aMC proc_gg_ttx.mg5
```

generate g g > t t~
output madevent_simd
launch

S. Roiser, Gargnano meeting 18 Sep 2023

The code has many additional fixes since then,
many of which already picked by CMS.
A “proper” release will come eventually!

NB: gcc >= 11.2.1 recommended

STEFAN ROISER, 18 SEP 2023, MADGRAPH5_AMC@NLO

18

Beyond NVidia GPUs

Joergen Teig

- The CUDACPP plugin uses a single source-code approach for CPUs (C++) and NVidia GPUs (CUDA), based on #ifdef's
 - The *few CUDA calls* are encapsulated by design in GPU classes
 - We do not use any vendor-specific features (e.g. Streams) yet
- We recently added support for AMD GPUs (HIP), using the same #ifdef approach (status: pull request merge is pending)
 - NVidia and AMD GPUs represent almost all of HPC top500 GPUs
 - It might be possible to extend this further to Intel GPUs
- Another implementation using SYCL (for CPUs and NVidia, AMD, Intel GPUs) was developed by our Argonne colleagues
 - Detailed performance comparisons are planned

```
// Copyright (C) 2020-2023 CERN and UCLouvain.
// Licensed under the GNU Lesser General Public License (version 3 or later).
// Created by: J. Teig (Jul 2023) for the MG5aMC CUDACPP plugin.
// Further modified by: J. Teig, A. Valassi (2020-2023) for the MG5aMC CUDACPP plugin.

#ifdef MG5aMC_GPUABSTRACTION_H
#define MG5aMC_GPUABSTRACTION_H 1

#include <cassert>

//-----

#ifdef __CUDACC__

#define gpuError_t cudaError_t
#define gpuPeekAtLastError cudaPeekAtLastError
#define gpuGetErrorString cudaGetErrorString
#define gpuSuccess cudaSuccess

#define gpuMallocHost( ptr, size ) checkGpu( cudaMallocHost( ptr, size ) )
#define gpuMalloc( ptr, size ) checkGpu( cudaMalloc( ptr, size ) )

#define gpuMemcpy( dstData, srcData, srcBytes, func ) checkGpu( cudaMemcpy( dstData, srcData, srcBytes, func ) )
#define gpuMemcpyHostToDevice cudaMemcpyHostToDevice
#define gpuMemcpyDeviceToHost cudaMemcpyDeviceToHost
#define gpuMemcpyToSymbol( type1, type2, size ) checkGpu( cudaMemcpyToSymbol( type1, type2, size ) )

#define gpuFree( ptr ) checkGpu( cudaFree( ptr ) )
#define gpuFreeHost( ptr ) checkGpu( cudaFreeHost( ptr ) )

#define gpuSetDevice cudaSetDevice
#define gpuDeviceSynchronize cudaDeviceSynchronize
#define gpuDeviceReset cudaDeviceReset

#define gpuLaunchKernel( kernel, blocks, threads, ... ) kernel<<<blocks, threads>>>( __VA_ARGS__ )
#define gpuLaunchKernelSharedMem( kernel, blocks, threads, sharedMem, ... ) kernel<<<blocks, threads, sharedMem>>>( __VA_ARGS__ )

//-----

#elif defined __HIPCC__

#define gpuError_t hipError_t
#define gpuPeekAtLastError hipPeekAtLastError
#define gpuGetErrorString hipGetErrorString
#define gpuSuccess hipSuccess

#define gpuMallocHost( ptr, size ) checkGpu( hipHostMalloc( ptr, size ) ) // HostMalloc better
#define gpuMalloc( ptr, size ) checkGpu( hipMalloc( ptr, size ) )

#define gpuMemcpy( dstData, srcData, srcBytes, func ) checkGpu( hipMemcpy( dstData, srcData, srcBytes, func ) )
#define gpuMemcpyHostToDevice hipMemcpyHostToDevice
#define gpuMemcpyDeviceToHost hipMemcpyDeviceToHost
#define gpuMemcpyToSymbol( type1, type2, size ) checkGpu( hipMemcpyToSymbol( type1, type2, size ) )

#define gpuFree( ptr ) checkGpu( hipFree( ptr ) )
#define gpuFreeHost( ptr ) checkGpu( hipHostFree( ptr ) )

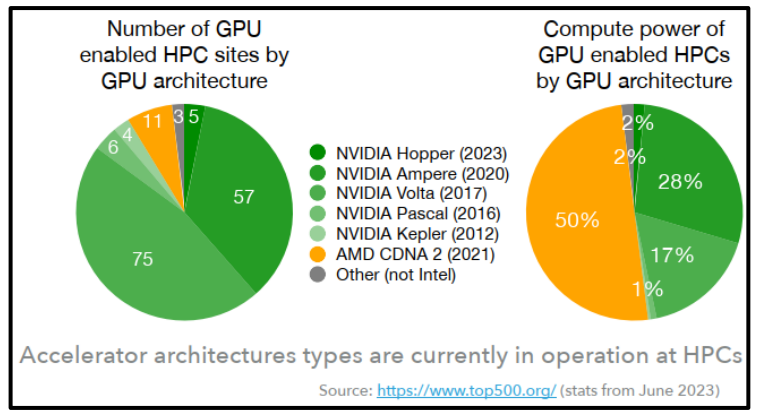
#define gpuSetDevice hipSetDevice
#define gpuDeviceSynchronize hipDeviceSynchronize
#define gpuDeviceReset hipDeviceReset

#define gpuLaunchKernel( kernel, blocks, threads, ... ) kernel<<<blocks, threads>>>( __VA_ARGS__ )
#define gpuLaunchKernelSharedMem( kernel, blocks, threads, sharedMem, ... ) kernel<<<blocks, threads, sharedMem>>>( __VA_ARGS__ )

//-----

#endif

#endif // MG5aMC_GPUABSTRACTION_H
```



S. Roiser,
Gargnano meeting 18 Sep 2023



Status: recent progress and LO prospects

- An as-is but usable version of the code has been provided to and tested by CMS
 - Including recent bug fixes for specific physics processes (Drell-Yan plus jets) or computing environments (HPCs)
- There are pending bugs, affecting some processes relevant to the experiments
 - Example: floating point exceptions in pp to ttW (suggested by ATLAS) and other processes
 - Example: code generation and/or builds fail for EFT and SUSY processes, mainly related to the scaling of alphas
- A few additional technical issues to iron out before a proper LO release
 - Proper choice of user configuration via the runcard (floating point precision, SIMD vectorization level etc)
 - Possibly: integration of AMD support, cleanup of Makefiles
- Longer-term improvements and performance optimizations will still be needed for LO after the release
 - Integration of helicity recycling (see Olivier's talk): combine the speedups of both SIMD and helicity recycling on CPU
 - Smaller GPU kernels (leading to shorter builds for 2→6 and allowing even more gluons in the final state?)
 - Combine event-level and helicity-level parallelism (reduce the minimum number of events needed on the GPU – see later)
 - Multi-GPU support, tuning and optimization of heterogeneous environments (with the HEPIX benchmarking WG)
 - Further extend infrastructure for matrix element reweighting
 - Port to GPU/SIMD more than just the MEs (e.g. PDF's, momenta computation from random numbers in sampling step...)

NLO, loops

Z. Wettersten (+ OM, SR, AV, R. Schoefbeck)

- So far we have only worked on LO QCD processes!
- NLO QCD processes are more computationally intensive
 - More Feynman diagrams
 - And especially, loop diagrams! (quad precision needed?)
 - A matching procedure (MC@NLO) must also be applied
- *We should be able to compute Born and Real emission contributions in our vectorized C++ and CUDA*
 - We should also be able to handle NLO matching using the current MadEvent based infrastructure
 - The main challenge will be understanding the computational impact of loops (Amdahl bottleneck?)

- News (for me!) from some discussions last week at Les Houches
 - **Branching should not be an issue at NLO, but will be at NNLO? Local subtraction schemes...**
 - What the code does depends on where you are in phase space...
 - NLO and NNLO needs “complicated” functions like polylogarithms (are these supported in SIMD and CUDA?)
 - Libraries exist to emulate quad precision (even for SIMD and CUDA), we can look at these (strip them down?)

- What about EWK beyond-LO corrections?
 - If I understand correctly, our approach would be portable, and the same types of challenges would apply?

More work since June (next slide)

Profiled the impact of loops

Discussed with Stefano Frixione (thanks!) about branching and lockstep processing for NLO



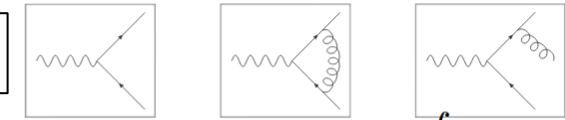
CPU time profiles for NLO: a first look

MC@NLO: <https://doi.org/10.1088/1126-6708/2002/06/029>

Matching NLO QCD and parton showers (avoid double counting)

Marco Zaro – <https://cp3.irmp.ucl.ac.be/projects/madgraph/wiki/Pavia2015>

B, V, R: matrix elements
MC: counterterms (B, PS)

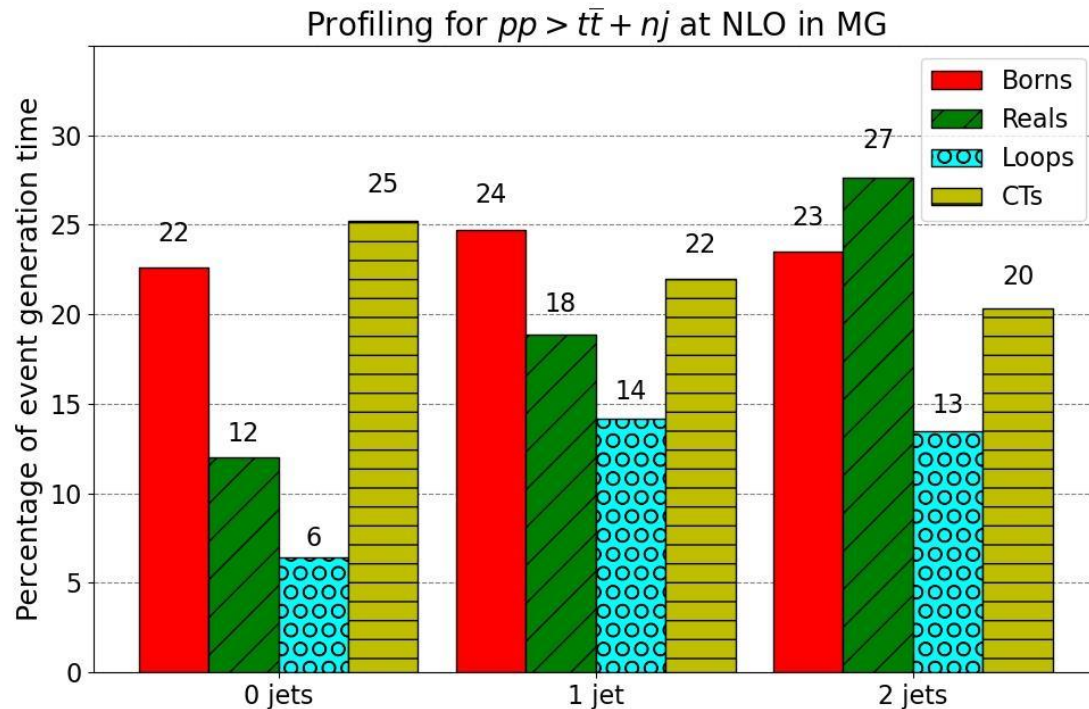


$$d\sigma_{NLO}^n = d\sigma_{LO}^n + d\sigma_V^n + \int d\Phi_1 d\sigma_R^{n+1}$$

$$\frac{d\sigma_{MC@NLO}^n}{dO} = \left[\int d\Phi_n (B + V + \int d\Phi_1 MC) \right] I_{MC}^n(O) + \left[\int d\Phi_{n+1} (R - MC) \right] I_{MC}^{n+1}(O)$$

S-events H-events

S and H events: two separate sets of events (different matrix elements)
Integral = S+H is positive – but individual events can have negative weights



Z. Wettersten

Borns, Reals, Counterterms, and ~half of virtual Loops contributions involve LO matrix elements that we already ported to GPUs!

Only ~half of Loops need a GPU port (here: ninja) or would remain on the CPU (less than 10% overall in this example)

- **Our first priority for NLO: move the NLO madevent framework from a single-event to a multi-event API**
 - And replace the Fortran ME calculations (in Born, Reals, CTs and part of Loops) using the CUDACPP ME bridge
 - Technicality: keep strict lockstep processing in cleanly separate FKS sectors *[thanks to Stefano Frixione for a useful chat!]*
- Porting loop packages like ninja to NLO will become important (Amdahl's law...) but is not the first priority

Wishes for the future – interaction with other generator teams

- Some discussions with other generator teams have already started – non-exhaustive list of examples below
- POWHEG (E. Re, S. Alioli, C. Oleari)
 - **Integrate MG5AMC GPU/SIMD MEs into POWHEG** (status: interest confirmed – technically feasible, need time/effort)
- SHERPA (S. Hoeche, M. Knobbe, E. Bothmann) + ATLAS (J. McFayden)
 - **Detailed comparison of MG5AMC and SHERPA** (status: interest ~confirmed? – need time/effort)
 - Compare generation speeds (with and without GPU/SIMD) for a few benchmark processes relevant to ATLAS and CMS
 - Understand the impact on speed and precision of technical choices (Berends-Giele recursion relations, helicity sampling/summing...)
- PYTHIA (P. Ilten)
 - Porting of parton showers and/or minimum bias to GPU/SIMD? (status: wish – interesting technical challenge, branching)
- HERWIG (S. Platzer)
 - Integration of MG5AMC GPU/SIMD [complex] amplitudes, rather than [real] MEs (status: wish – heavier technical work)

My personal thanks to the organizers of Les Houches 2023 for a memorable experience! 😊



Lessons learnt for other MC generators?

What is a MC *ME* generator? A simplified computational anatomy

(at least at LO!)

Monte Carlo sampling: randomly generate and process
MANY different events ("phase space points")

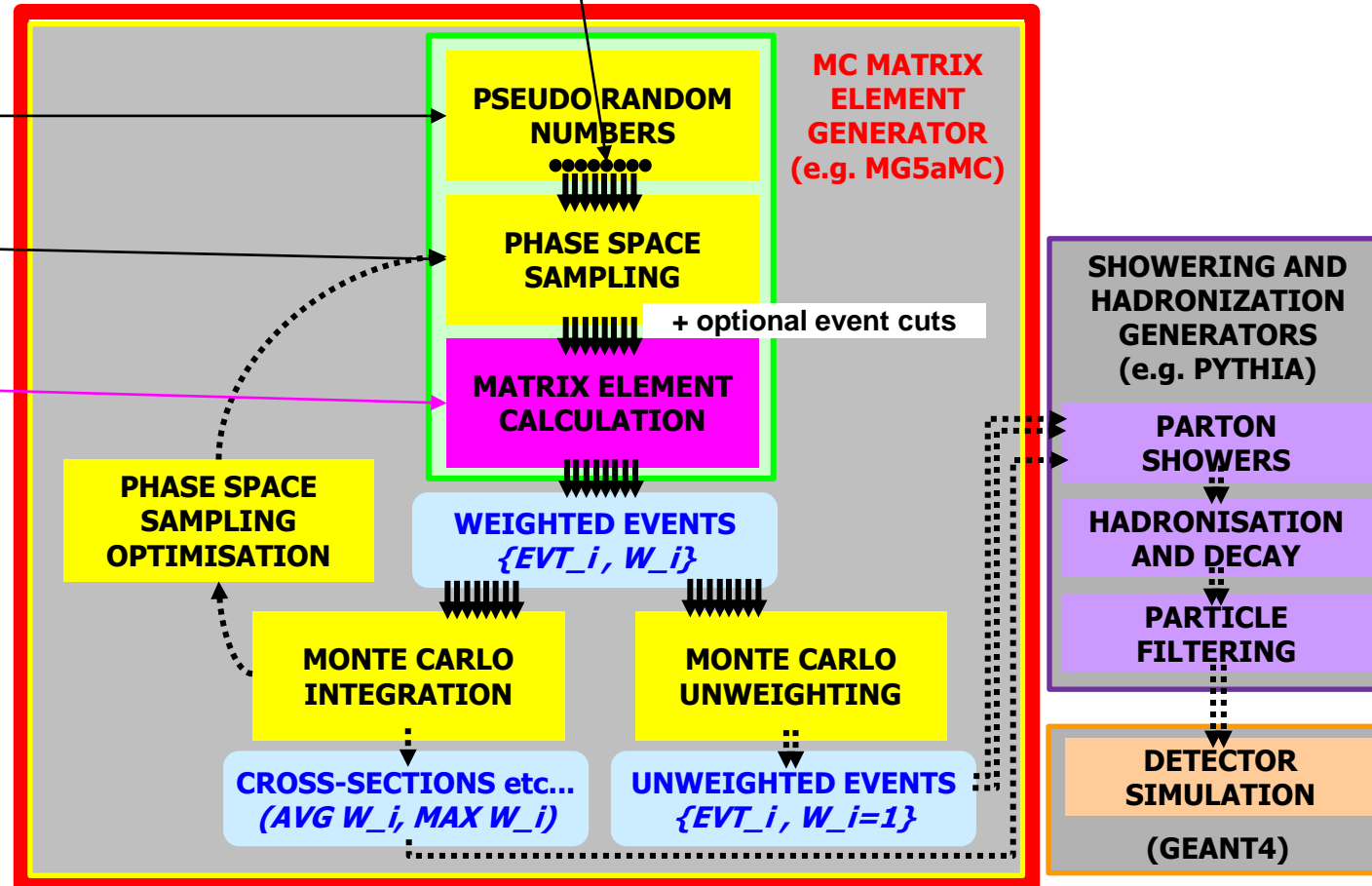


This can be parallelized (SIMT/SIMD and multithreading)

For each event:

1. _____
Output: random numbers
2. _____
Input: random numbers
Output: particle 4-momenta

3. _____
Input: particle 4-momenta
Output: Matrix Element (ME)
CPU BOTTLENECK

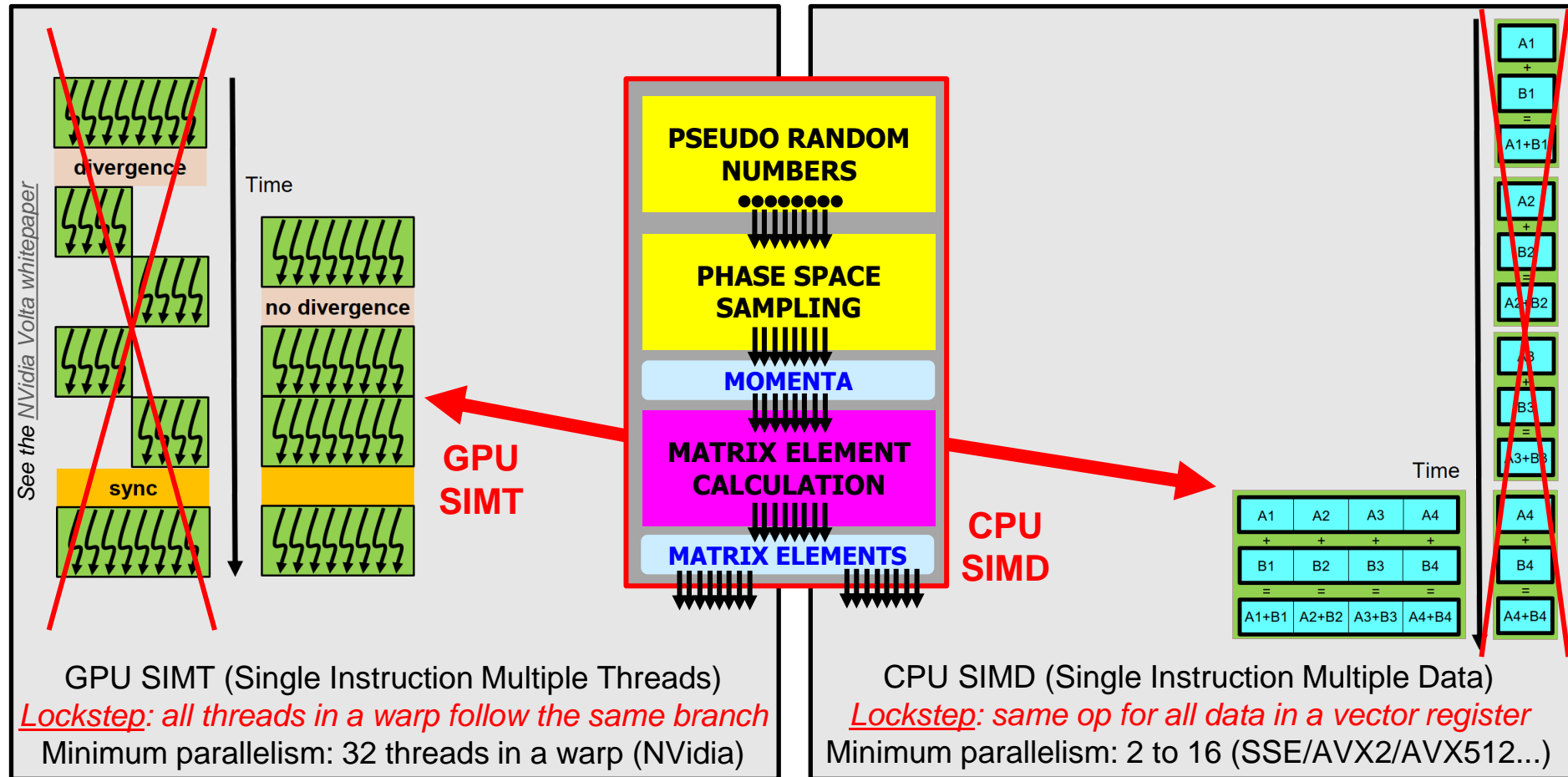


(NB: for non-physicists in the room, the "Matrix Element" is an element of the **scattering matrix**... not a linear algebra calculation!)

Physics output: cross-section and LHE event file

MG5aMC data parallelism: design for lockstep processing!

- In MC ME generators, the same function is used to compute the Matrix Element for many different events
 - ANY** matrix element generator is a good fit for lockstep processing on GPUs (SIMT) and vector CPUs (SIMD)
 - Data parallelism strategy in madgraph4gpu is event-level parallelism (many events = many phase space points)



ANY ME event generator is a great fit for GPUs and vector CPUs!

(at least at LO!)

- Monte Carlo methods are based on drawing (pseudo-)random numbers: a dice throw
- From a software workflow point of view, these are used in *two rather different cases*:



Data parallelism (NB: MULTI-EVENT API !)

MC SAMPLING

ME event generators*

(before ME calculation):

- MC integration (cross sections)
- MC generation (event samples)



Lockstep processing
Good for SIMT/SIMD

INPUT



SAME CALCULATION
ON DIFFERENT DATA!

OUTPUT

*NB: the CPU-intensive ME calculation comes before PS, fragmentation, detector simulation

INPUT



OUTPUT

Stochastic branching
Bad for SIMT/SIMD

MC DECISIONS



Detector simulation (Geant4)

- Particle/matter interaction (when? how?)
- Particle decays (when?)

DIFFERENT CALCULATIONS
ON DIFFERENT DATA!

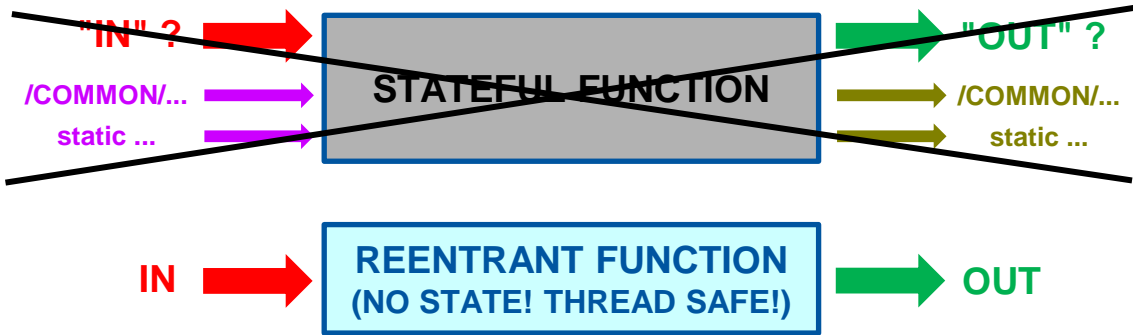
Event generators*

(after ME calculation):

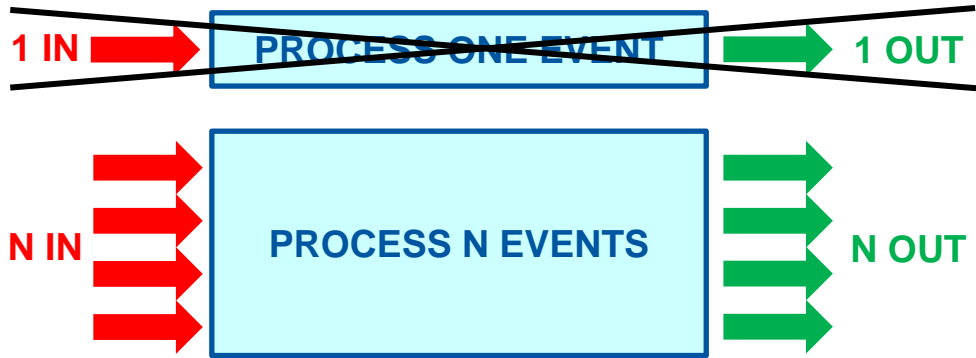
- MC unweighting (keep/reject)
- Parton showers (PS)
- Fragmentation and decays

Do's and dont's - two simple recommendations

- (1) Design computational units with **well-defined inputs and outputs!**
 - Beware of hidden inputs and outputs from common blocks and static data...



- (2) Keep data parallelism in mind from the start: move **from single-event APIs to multi-event APIs!**
 - Well-defined input array of many events, well-defined output array of many events

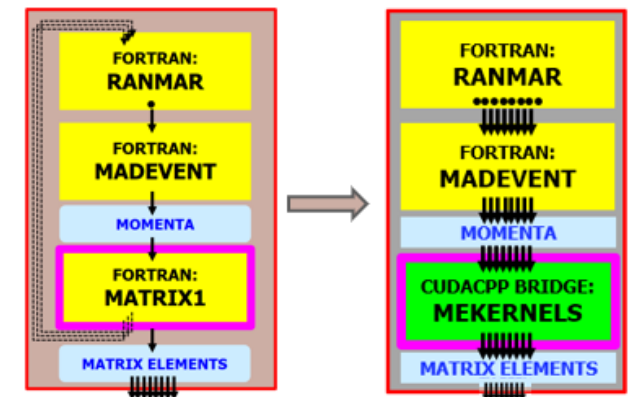
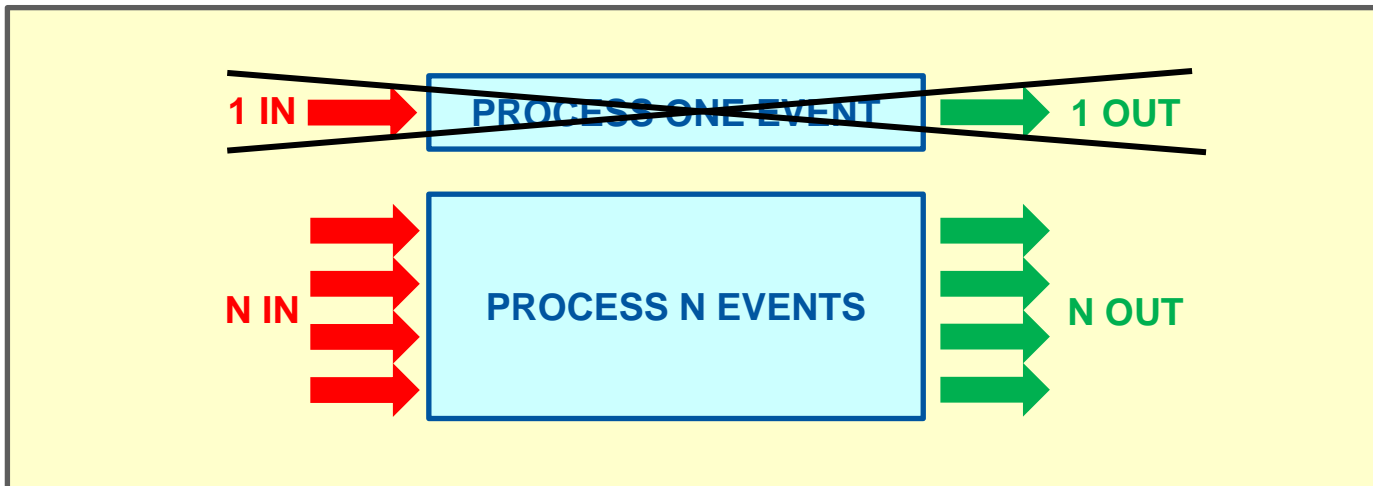


An additional technicality: prefer Structure-of-Array (SOA) memory layouts for the inputs and outputs! [Strictly needed only internally for SIMD and useful for GPUs, but good to have also in the API of the function]

If you design a new Monte Carlo from scratch, these are MUST's, not SHOULD's!

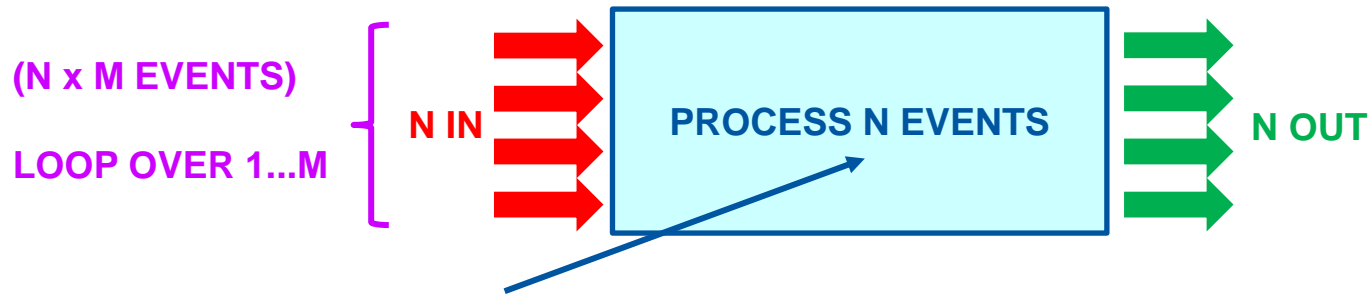
From single-event to multi-event APIs: some specific examples

- 1. **MG5AMC at LO**: the work described in this talk!
 - *This was all the work Olivier had to do on the madevent Fortran framework (to interface to the CUDACPP “bridge”)*
- 2. **MG5AMC at NLO**: the work we are planning!
 - This is the work Zenny and Olivier will do on the madevent Fortran framework (to interface to the CUDACPP “bridge”)
- 3. **POWHEG + MG5AMC**: the work we plan to collaborate with!
 - This is the work the POWHEG team will do on their framework (to interface to the MG5AMC CUDACPP “bridge”)



What about loops? And how many are N events?

- You will still need to loop over multiple sets of N events
 - And the internal implementation of N-event processing may still involve some loops!

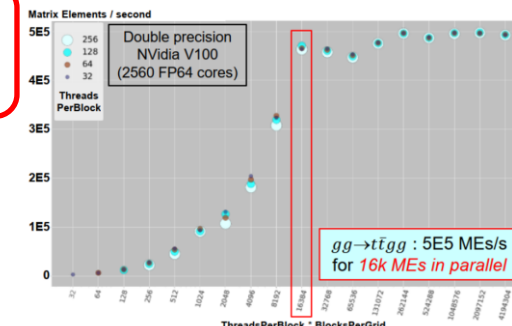


"Process N events": three implementation examples (there can be more!)

1. **CPU scalar**: internally loop over N events, process each one individually
2. **CPU vector**: hold the events in a SIMD vector of size N,
3. **GPU kernel**: each of the N events is processed by one of N GPU threads

...one more item on our to-do list for next year...
(also because so many events may lead to biases)

- N should be *at least* as big as the minimum number of events for which strict lockstep is required
 - On a CPU: number of variables in a vector register (*most complex case: 16* floats in a 512-bit AVX512 register)
 - On a GPU: strictly speaking, *number of threads (typically: 32) in a GPU "warp"*
 - Our present implementation: *number of threads to "fill" the GPU (typically: 16k, up to 500k)*
- NB: I focus on event-level parallelism, but other options exist
 - In MG5AMC we will investigate using 1 GPU thread per helicity per event...



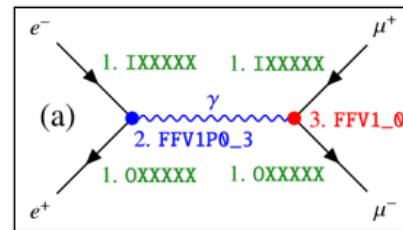
Helicity amplitudes – same code in CUDA and in vectorized C++

Formally the same code for three back-ends (cxttype_sv represents three types)

- CUDA: scalar complex → `typedef thrust::complex<fptype> cxttype; // two doubles: RI`
- C++, no SIMD: scalar complex → `typedef std::complex<fptype> cxttype; // two doubles: RI`
- C++, with SIMD: vector complex → `class cxttype_v { fptype_v m_real, m_imag; // RRRRIIII (SOA)`

```

__device__
void FFV1_0( const cxttype_sv F1[], // input: wavefunction1[6]
             const cxttype_sv F2[], // input: wavefunction2[6]
             const cxttype_sv V3[], // input: wavefunction3[6]
             const cxttype COUP,
             cxttype_sv* vertex ) // output: amplitude
{
    mgDebug( 0, __FUNCTION__ );
    const cxttype cI( 0., 1. );
    const cxttype_sv TMP0 = (F1[2] * (F2[4] * (V3[2] + V3[5]) + F2[5] * (V3[3] + cI * (V3[4]))) +
                             (F1[3] * (F2[4] * (V3[3] - cI * (V3[4])) + F2[5] * (V3[2] - V3[5])) +
                             (F1[4] * (F2[2] * (V3[2] - V3[5]) - F2[3] * (V3[3] + cI * (V3[4]))) +
                             F1[5] * (F2[2] * (-V3[3] + cI * (V3[4])) + F2[3] * (V3[2] + V3[5]))));
    (*vertex) = COUP * - cI * TMP0;
    mgDebug( 1, __FUNCTION__ );
    return;
}
    
```



FFV1_0:
helicity amplitude
for the $\gamma\mu^+\mu^-$ vertex

Automatically
generated!

“+” is the usual sum of two
(thrust/std) scalar complex,
or the user defined sum of
two vector complex

```

inline
cxttype_v operator+( const cxttype_v& a, const cxttype_v& b )
{
    return cxmake( a.real() + b.real(), a.imag() + b.imag() );
}
    
```

C++ SIMD: gcc / clang
compiler vector extensions

```

#ifdef __clang__
    typedef fptype fptype_v __attribute__((ext_vector_type(neppV))); // RRRR
#else
    typedef fptype fptype_v __attribute__((vector_size (neppV*sizeof(fptype)))); // RRRR
#endif
    
```

- Old slide! The new code is different, the idea is the same!
- **Formally the same code for CUDA and scalar/vector C++**
 - hide type behind a typedef
 - add a few missing operators

SIMD in CUDA/C++ uses compiler vector extensions!

Flexible design: being reused also in the vectorized SYCL implementation

```

typedef sycl::vec<fptype, MGONGPU_MARRAY_DIM> fptype_sv;
    
```



Conclusions

Conclusions

- Our journey **to port MG5AMC LO calculations to GPUs and SIMD** is starting to bring fruits (see the CMS talk!)
 - It has been long (3+ years) and challenging, but it has been fun!
 - We still have a few things to iron out before a “proper” release – and many optimizations are still possible after that!
- We have demonstrated that **near-perfect lockstep processing in LO ME event generators is possible**
 - A typical factor 4 speedup on CPUs from AVX2 SIMD in double precision, much more than that on GPUs
 - We accelerated the ME calculation so much, that previously unimportant components become the bottleneck (Amdahl)
- Our journey **to port MG5AMC NLO calculations to GPUs and SIMD** is just starting and will also be long (and fun!)
 - We have indications that porting the bulk of NLO calculations will be feasible using our LO tools
 - Porting the packages providing virtual loop calculations will be harder, but is not our first bottleneck and priority
- Any Monte Carlo ME event generator (at least at LO) is a great fit for data parallelism using GPUs and SIMD
 - We would be **eager to collaborate with other MC teams** (to help them interface with MG5AMC or reengineer their code)

We believe that this work is essential to fully exploit the HL-LHC physics program, and that of future colliders

BACKUP SLIDES

Our internal Fortran-to-C++ interface: multi-event and stateless!

```
C
C Execute the matrix-element calculation "sequence" via a Bridge on GPU/CUDA or CUDA/C++.
C - PBRIDGE: the memory address of the C++ Bridge
C - MOMENTA: the input 4-momenta Fortran array
C - GS:      the input Gs (running QCD coupling constant alphas) Fortran array
C - RNDHEL:  the input random number Fortran array for helicity selection
C - RNDCOL:  the input random number Fortran array for color selection
C - CHANID:  the input Feynman diagram to enhance in multi-channel mode if 1 to n (disable multi-channel if 0)
C - MES:     the output matrix element Fortran array
C - SELHEL:  the output selected helicity Fortran array
C - SELCOL:  the output selected color Fortran array
C
  INTERFACE
    SUBROUTINE FBRIDGESEQUENCE(PBRIDGE, MOMENTA, GS,
&    RNDHEL, RNDCOL, CHANID, MES, SELHEL, SELCOL)
    INTEGER*8 PBRIDGE
    DOUBLE PRECISION MOMENTA(*)
    DOUBLE PRECISION GS(*)
    DOUBLE PRECISION RNDHEL(*)
    DOUBLE PRECISION RNDCOL(*)
    INTEGER*4 CHANID
    DOUBLE PRECISION MES(*)
    INTEGER*4 SELHEL(*)
    INTEGER*4 SELCOL(*)
    END SUBROUTINE FBRIDGESEQUENCE
  END INTERFACE
```

This outputs the squared sum of amplitudes (real number)

As discussed with Simon, for HERWIG and other generators it may be useful to also expose an API that gives the partial amplitude (complex number) for a given colour structure

MG5AMC is not alone – SHERPA on GPU (BlockGen)

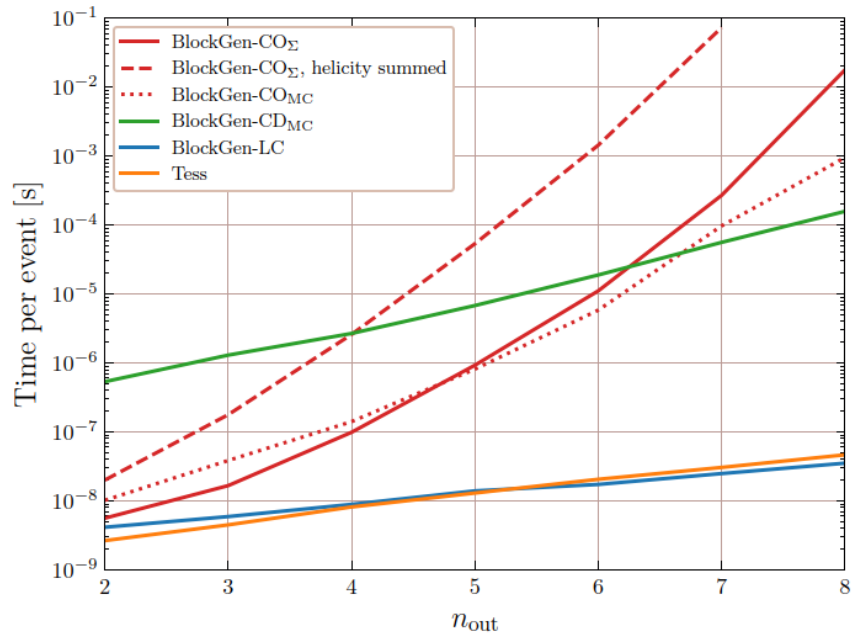


Figure 7: The timings for various GPU-based algorithms are compared as a function of gluon multiplicity. All algorithms were run on an NVIDIA V100 (16 GB global memory, 5,120 CUDA cores, 6144 KB L2 cache).

From <http://dx.doi.org/10.21468/SciPostPhysCodeb.3>

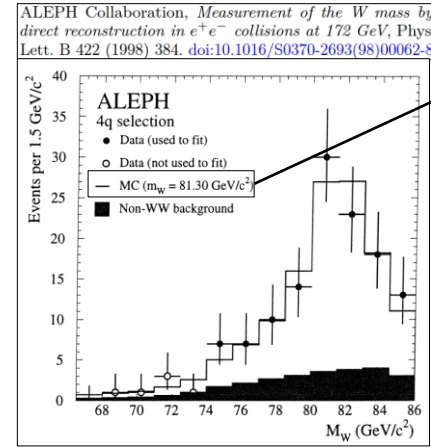
More recent results were presented in June 2023
in Les Houches by Max Knobbe

- Note: unlike MG5aMC, based on Feynman diagrams, SHERPA uses ~Berends-Giele recursion relations
 - Allows computations with more final-state jets
- No ongoing effort on CPU vectorization (yet)
- Planned Les Houches project: a detailed comparison of *software performances* of MG5AMC and SHERPA
 - Tentative process list: pp to tt(0-3jets) or Z(0-3jets)
 - Previously, an old wish of the HSF generator WG
 - (NB: not a comparison of physics results or distributions)



Reweighting

1. *Generate signal sample at θ_{ref} with $w_i(\theta_{ref})=1$*
(By definition, background does not depend on θ)
 2. *Full detector simulation*
(MC truth event properties $\mathbf{x}_i^{(true)} \rightarrow$ observed event properties \mathbf{x}_i)
 3. ***Reweight each event by matrix element ratio***
- $$w_i(\theta) = \frac{\text{Prob}_{(\theta)}(\mathbf{x}_i^{(true)})}{\text{Prob}_{(\theta_{ref})}(\mathbf{x}_i^{(true)})} = \frac{|\mathcal{M}(\theta, \mathbf{x}_i^{(true)})|^2}{|\mathcal{M}(\theta_{ref}, \mathbf{x}_i^{(true)})|^2}$$



$$w_i(m_W, \Gamma_W) = \frac{|\mathcal{M}(m_W, \Gamma_W, p_i^1, p_i^2, p_i^3, p_i^4)|^2}{|\mathcal{M}(m_W^{MC}, \Gamma_W^{MC}, p_i^1, p_i^2, p_i^3, p_i^4)|^2}$$

Old technique, renewed interest!

- Advantages of reweighting: savings in computing costs (no detector simulation), fewer statistical fluctuations
- In practice ***for MG5AMC: read in an LHE file, add weights, write back the modified LHE file***
 - ***Will use the new matrix element engine in CUDA/C++***
 - For further details and a status report: Zenny's [talk](#) (and upcoming paper) at CHEP 2023
- Theoretical and technical challenges
 - NLO reweighting (see O. Mattelaer, <https://arxiv.org/abs/1607.00763>)
 - Coverage of phase space in the new parameter set
 - Reweighting for a given event-by-event helicity and color

Reweighting and weight derivatives in parameter estimation

- Weight derivative: event-by-event sensitivity to the measured parameter $\gamma_i|\theta = \left(\frac{1}{w_i} \frac{\partial w_i}{\partial \theta} \right)_\theta$
- First: makes it possible to determine the limit error with an ideal detector, and how much (0 to 1) we do worse
 - with a given luminosity at a FCC-ee, what is the best theoretically achievable measurement on Higgs couplings?

Knowing one's limits: maximum achievable information with an ideal detector

- Ideal acceptance, select all signal events $S_{\text{sel}}=S_{\text{tot}}$
- Ideal resolution, measured γ_i is that from MC truth (implies ideal rejection of background events, $\gamma_i=0$)

$$\mathcal{I}_\theta^{(\text{ideal})} = \sum_{i=1}^{N_{\text{tot}}} \gamma_i^2 = \sum_{i=1}^{S_{\text{tot}}} \gamma_i^2$$

$$\text{FIP} = \frac{\mathcal{I}_\theta}{\mathcal{I}_\theta^{(\text{ideal})}} = \frac{(\Delta\theta^{(\text{ideal})})^2}{(\Delta\theta)^2} \leq 100\%$$

- Second: can be used as a basis for an “improved optimal observable” ML method

Weight Derivative Regression

$$\gamma_i^{(\text{MC truth})} \sim q(x_i^{(\text{MC})})$$

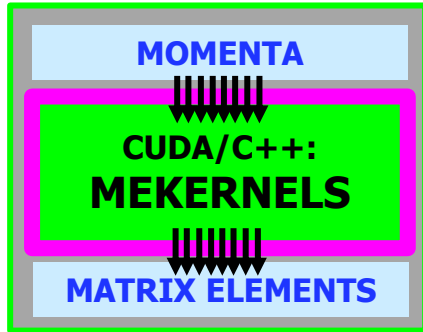
Data observable event properties $x_i^{(\text{DATA})}$

Fit WDR regressor

$$\begin{aligned} \mathbf{q}_i^{(\text{DATA})} &= \mathbf{q}(x_i^{(\text{DATA})}) \\ \mathbf{q}_i^{(\text{MC})} &= \mathbf{q}(x_i^{(\text{MC})}) \end{aligned}$$

<https://doi.org/10.1051/epjconf/202024506038>
<https://zenodo.org/record/3715951>

Memory layouts – AOS, SOA, AOSOA



Matrix element calculation (simplified example)

- $inputs[4*Npar*Nevt]$ = (x,y,z,E)-momentum of Npar particles for Nevt events (n-dim array, substructure)
- $outputs[Nevt]$ = matrix element for Nevt events (1-dim array, no substructure)

Example: Npar=6 particles for the 2→4 process $gg \rightarrow t\bar{t}gg$

We have experimented with three possible memory layouts for momenta

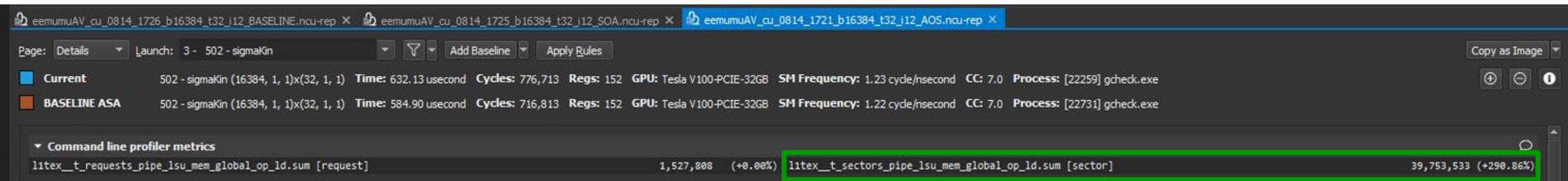
- (1) Array-of-Structures **AOS**: $momenta[Nevt][Npar][4]$
- (2) Structure-of-Arrays **SOA**: $momenta[Npar][4][Nevt]$
- (3) **AOSOA**: $momenta[Npag][Npar][4][Nepp]$ with $Nevt = Npag$ (“pages”) * $Nepp$ (“events per page”)

We are using AOSOA’s as the current default – but this is still largely configurable

- **For CPU vectorization, AOSOAs (or SOAs) are absolutely mandatory!**
 - We use an AOSOA with $Nepp$ equal to the SIMD vector size $NeppV$ – and an *aligned malloc* is needed too!
 - For performance comparison we also build a no-SIMD mode with $Nepp=1$, which is effectively an AOS
- **For GPUs (1 event per thread), AOSOAs are faster (fewer memory accesses) but not strictly necessary**
 - We use $Nepp=4(8)$ for doubles(floats) so that each page is 32 bytes (the “sector” size, or L2 cache line size)
 - For a given number of “requests”, *AOS uses 4 times more “sectors” (transactions) than AOSOA* with $Nepp=4$
- Coding for SIMD is more complex than coding for GPUs...

Monitoring GPU memory access – NSight Compute

- Explicitly collect two relevant profiler metrics in NSight Compute
 - “requests” : `l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum`
 - “sectors” (i.e. transactions, network roundtrips): `l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum`
 - this is from old tests in August 2020 ([issue #16](#)), the profiler metrics names may have changed since then



- Profile AOS against the AOSOA baseline
 - same number of “requests” in AOS and AOSOA
 - AOS needs 4 times as many “sectors” as AOSOA (which fits 4 doubles in a 32-byte cache line)
 - in other words: *AOSOA provides coalesced memory access, AOS does not*
 - for what it is worth (not much!), the actual slowdown in this $e^+e^- \rightarrow \mu^+\mu^-$ example was only 7% however

Inside the ME calculation: Feynman diagrams, colors, helicities

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{\text{hel}\}} \left[\sum_{c \in \{\text{col}\}} \left| \sum_{d \in \{\text{diag}\}} (\mathcal{M}_\lambda^d(\vec{p}))^{(c)} \right|^2 \right]$$

Given the momenta \vec{p} of initial+final partons **in one specific event**
Sum over all helicity combinations λ of initial+final partons
Sum over all color combinations c of initial+final partons
Include all Feynman diagrams d allowed for the given λ and c

In practice in MG5aMC: use **helicity amplitudes** and **QCD color decomposition**

1. (for each helicity λ) compute partial amplitudes J^f for each color ordering permutation f (sum diagrams relevant to f)

$$(J_\lambda(\vec{p}))^f = \sum_{d \in \{\text{diag}\}} (\mathcal{M}_\lambda^d(\vec{p}))^f$$

Example for $gg \rightarrow t\bar{t}ggg$: 1240 Feynman diagrams (using helicity amplitudes)
 This takes **~40% of the CPU time** for this process

2. (for each helicity λ) compute the sum over colors as the quadratic form $J C J^*$ using the constant color matrix C

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{\text{hel}\}} \left[\sum_{f,g} (J_\lambda(\vec{p}))^f (C)^{fg} (J_\lambda^*(\vec{p}))^g \right]$$

Example for $gg \rightarrow t\bar{t}ggg$: 120 color ordering permutations, 120x120 matrix
 This takes **~60% of the CPU time** for this process

3. sum over helicities [Example for $gg \rightarrow t\bar{t}ggg$: 128 helicities (before and after filtering)]

Each step computes many events \vec{p} in parallel! CPU: 1 SIMD event-vector at a time. GPU: 1 event per thread.

C++ vectorization – why choose Compiler Vector Extensions?

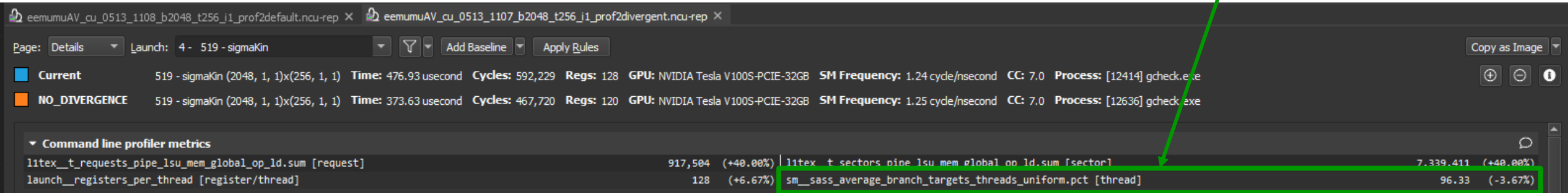
```
typedef fptype fptype_v __attribute__((vector_size (neppV*sizeof(fptype))));
```

- Portable – available in gcc, clang, icpx (from clang) with minimal differences
 - *Do not require any external libraries* or tools (VC, VCL, VecCore, xSIMD, UME::SIMD, or SYCL...)
- Powerful, but easy to use
 - *No need to debug auto-vectorization* when it does not vectorize
 - *As powerful as intrinsics, but much easier to write* (higher-level abstractions)
- Intuitive – *CVEs force you to think in terms of vector types!*
- Minor disadvantage – no vector complex type out of the box
 - But it was easy to write it in our case (RRRRIII memory layout) as we only need + - × ÷
 - A few extensions for Boolean vector masks were needed, too
- One technical detail: we malloc a standard (aligned!) fptype* and reinterpret_cast as fptype_v*...

HUGE THANKS TO SEBASTIEN PONCE for his Practical Vectorization lectures mentioning CVEs!

Monitoring lockstep – GPU NSight compute, CPU disassemble

- GPU: explicitly collect one profiler metric in NSight Compute
 - “branch efficiency” : `sm__sass_average_branch_targets_threads_uniform.pct`
 - old test (May 2021 [issue #25](#)) comparing two code bases: *no-divergence baseline has 100% efficiency*, alternative with minor forced divergence has 96% efficiency (and is 20% slower)

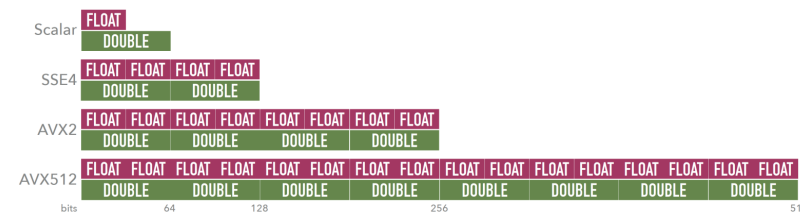


- CPU: the best lockstep metric IMO is the speedup over a no-SIMD case (reach theoretical maximum!)
 - but is also useful to disassemble the object using objdump and categorize SIMD intrinsics symbols...

4a90ec2 gg → ttgg

# Symbols in .o	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
Build type				
Scalar	4534	0	0	0
SSE4.2	12916	0	0	0
AVX2	0	10630	0	0
256-bit AVX512	0	10366	12	0
512-bit AVX512	0	1267	60	9910

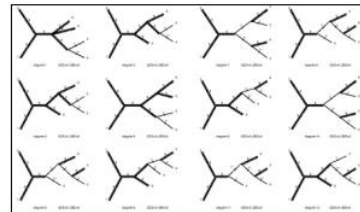
	ACAT2022	madevent
<code>gg → ttgg</code>	MEs precision	N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	2.30E3 (=1.0)
C++/none(scalar)	double	2.28E3 (x1.0)
C++/sse4(128-bit)	double	4.62E3 (x2.0)
C++/avx2(256-bit)	double	1.05E4 (x4.6)
C++/512y(256-bit)	double	1.16E4 (x5.0)
C++/512z(512-bit)	double	1.91E4 (x8.3)



Code generation: how did we bootstrap the project?

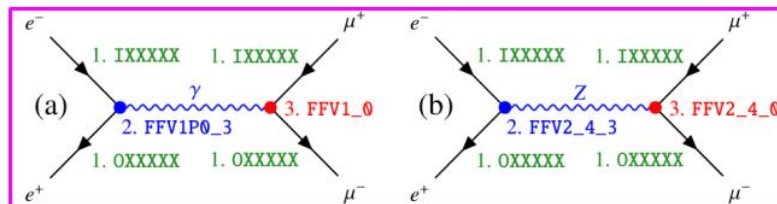
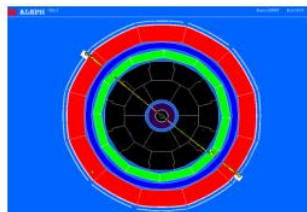
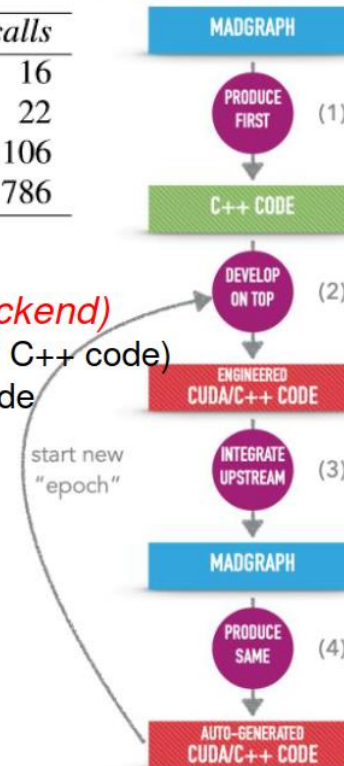
Code is auto-generated \Rightarrow Iterative development process

- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
 - Currently Fortran (default), C++, or Python
 - The more particles in the collision, the more Feynman diagrams and the more lines of code

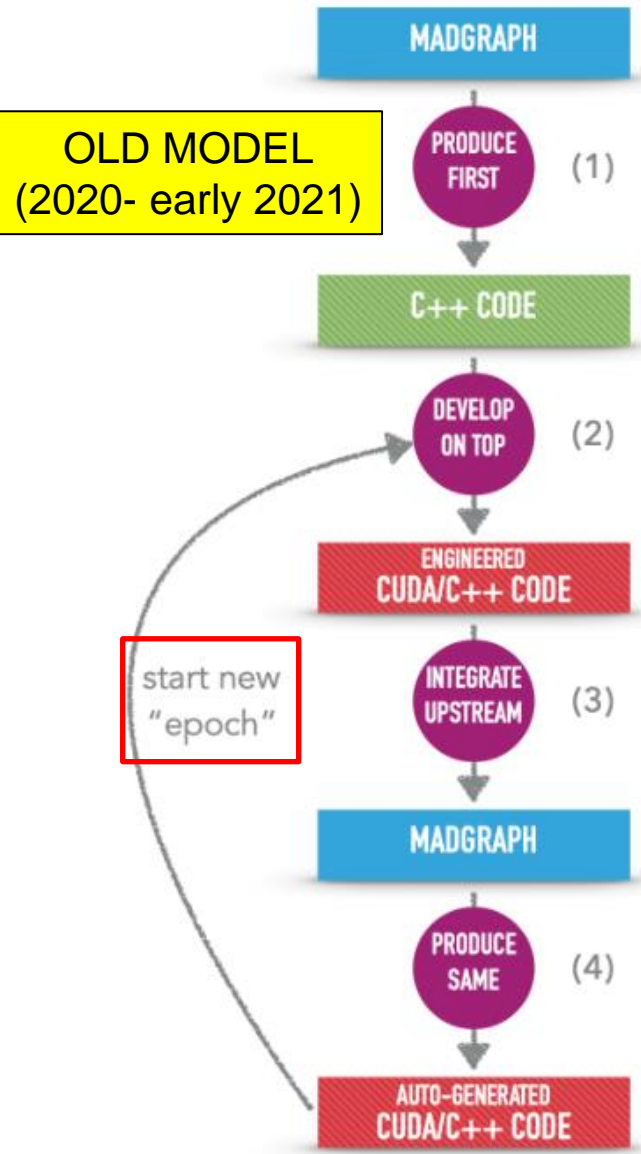


Process	LOC	functions	function calls
$e^+e^- \rightarrow \mu^+\mu^-$	776	8	16
$gg \rightarrow t\bar{t}$	839	10	22
$gg \rightarrow t\bar{t}g$	1082	36	106
$gg \rightarrow t\bar{t}gg$	1985	222	786

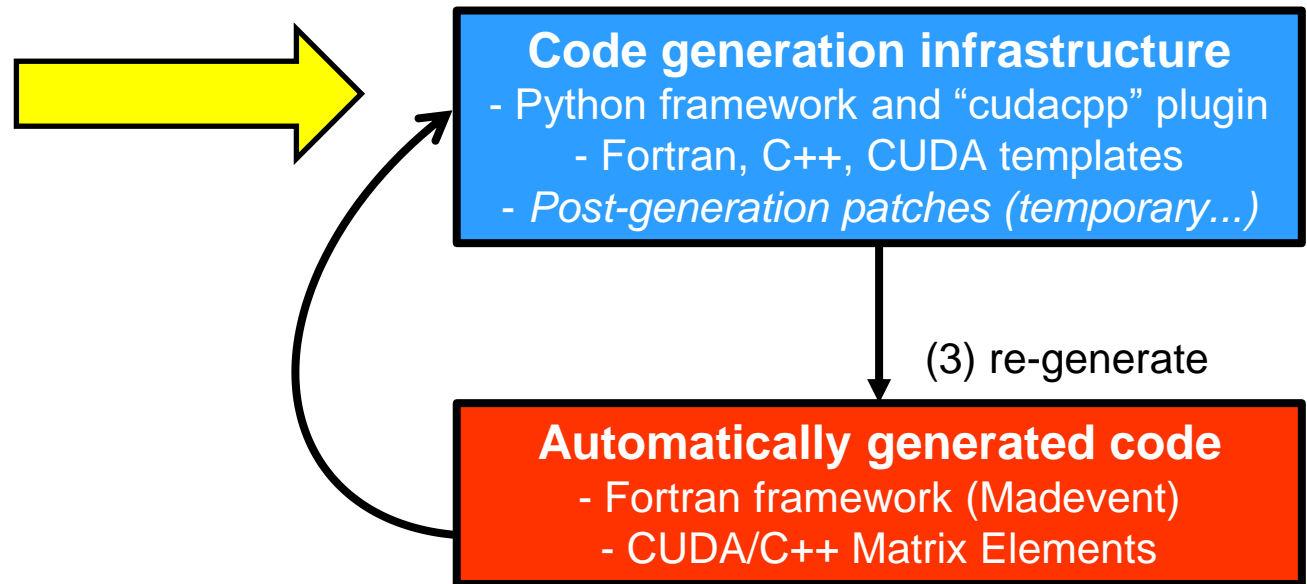
- Goal: modify code-generating code (add CUDA, improve C++ backend)*
 - (1) Start simple: *bootstrap with $e^+e^- \rightarrow \mu^+\mu^-$* (two diagrams, few lines of C++ code)
 - (2,3) Add CUDA and improve C++, port upstream to Python meta-code
 - (4) *Generate more complex LHC processes $gg \rightarrow t\bar{t}, t\bar{t}g, t\bar{t}gg$*
 - Add missing functionality, fix issues, improve performance, *iterate*



Code generation: from many “epochs” to a single evolving “epoch” ... and beyond



- (1) MG5AMC Python framework, Fortran templates: “upstream” <https://github.com/mg5amcnlo/mg5amcnlo>
- (2) CUDACPP plugin, post-generation patches, generated CUDA/C++ physics processes: our <https://github.com/madgraph5/madgraph4gpu>



WIP to-do before a release: full port from madgraph4gpu to mg5amcnlo (remove post-generation Fortran patches, add CUDACPP upstream)

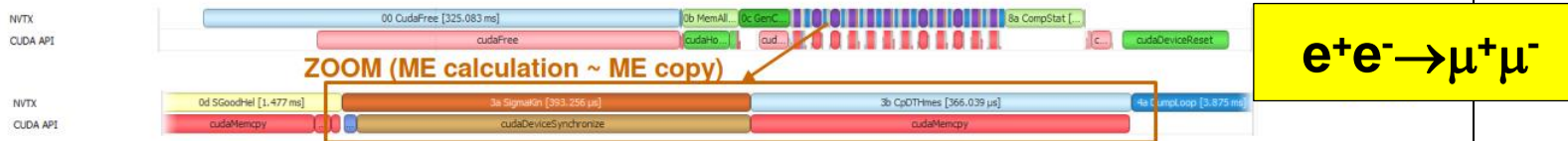
NEW MODEL (since end 2021)

- (1) develop on top of auto-generated code
- (2) backport immediately to code generation infrastructure

Why focus on complex processes? Compute >> memory!

CUDA: Host(CPU)-to/from-Device(GPU) data copy has a cost

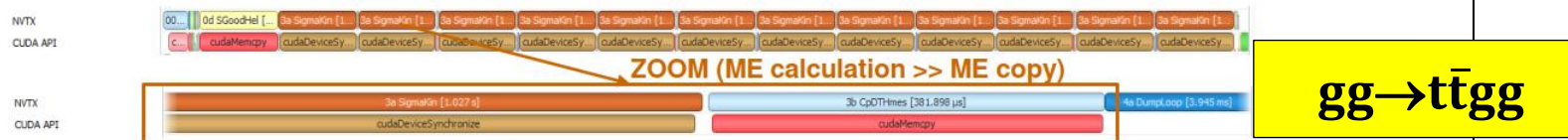
- In our standalone application (all on GPU): momenta, weights, MEs D-to-H
 - Plots below from Nvidia Nsight Systems: 12 iterations with 524k events in each iteration
- Eventually, MadEvent on CPU + MEs on GPU: momenta H-to-D; MEs D-to-H
- The time *cost of data transfers is relatively high in simple processes*
 - ME calculation on GPU is fast (e.g. $e^+e^- \rightarrow \mu^+\mu^-$: 0.4ms ME calculation ~ 0.4ms ME copy)
 - Note: our ME throughput numbers are (number of MEs) / (time for ME calculation + ME copy)



$e^+e^- \rightarrow \mu^+\mu^-$

- But the time *cost of data transfers is negligible in complex processes*

- ME calculation on GPU is slow (e.g. $gg \rightarrow t\bar{t}gg$: 1000ms ME calculation >> 0.4ms ME copy)
- We expect that *this will not be an issue for typical LHC collision processes*

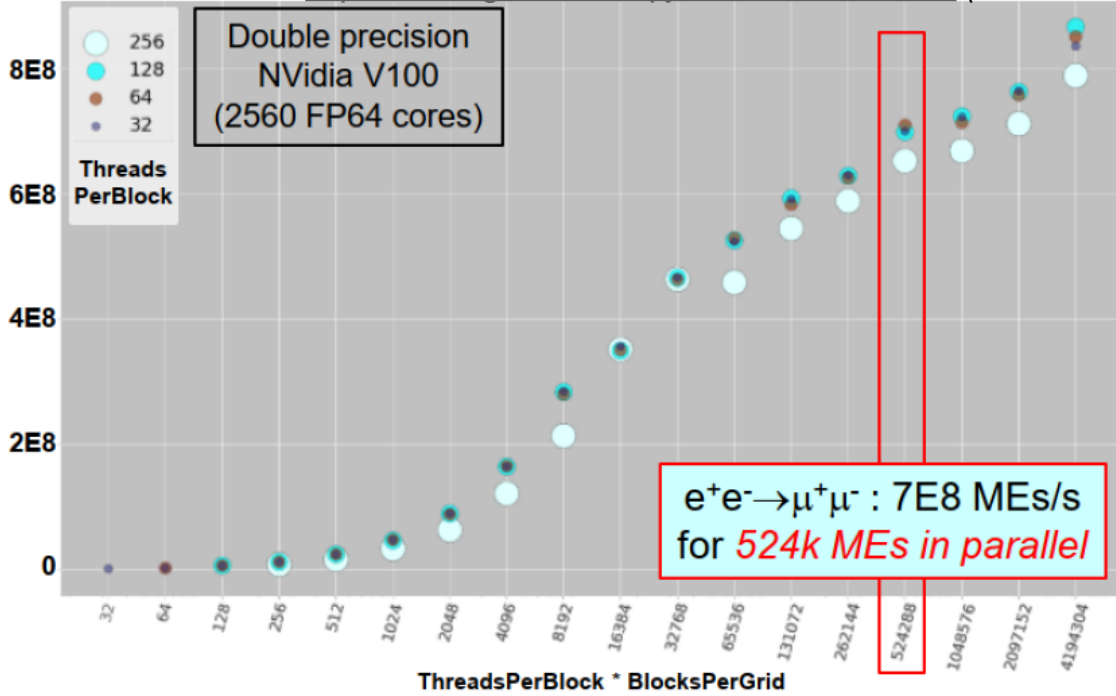


$gg \rightarrow t\bar{t}gg$

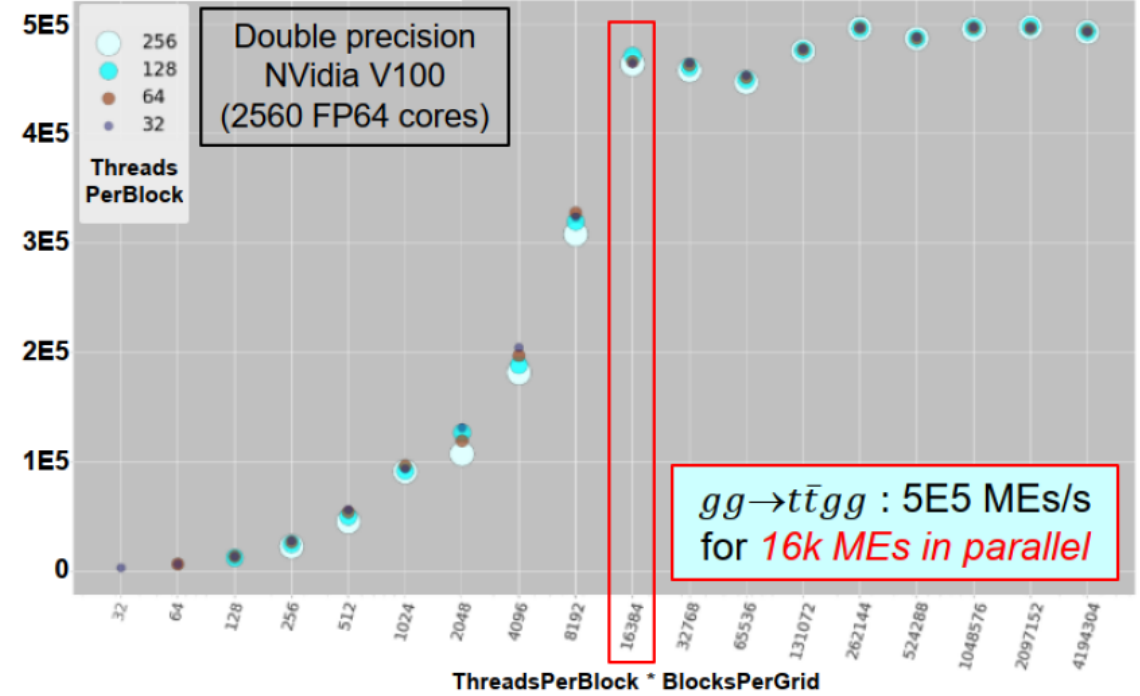
- We are lucky: the more complex the physics process, the less relevant is the cost of GPU-CPU data copies!
 - Similarly (later): the more complex the process, the less relevant is the overhead from scalar Fortran in madevent!
 - And the fewer events in flight needed to fill the GPU...
- In this talk I mainly give performance numbers for complex processes like $gg \rightarrow t\bar{t}gg$ or $gg \rightarrow t\bar{t}ggg$

Filling the GPU – minimum number of threads (events in flight)

Matrix Elements / second <https://doi.org/10.1051/epjconf/202125103045> (vCHEP 2021)



Matrix Elements / second



- We are lucky, again: *the more complex the process, the fewer the events in flight needed to fill the GPU*
- But *even 16k events is a lot*: it results in *imbalanced phase space sampling*, and *high RAM in Fortran*
 - Eventually, maybe: one helicity per kernel (fewer events in flight, spread each event across many kernels)?
 - Eventually, maybe: many CPU cores/processes in parallel (fewer events in flight per CPU core/process)?
 - Eventually, maybe: different channels in parallel (fewer events in flight in a single channel)?

All MadEvent functionalities have been integrated over time

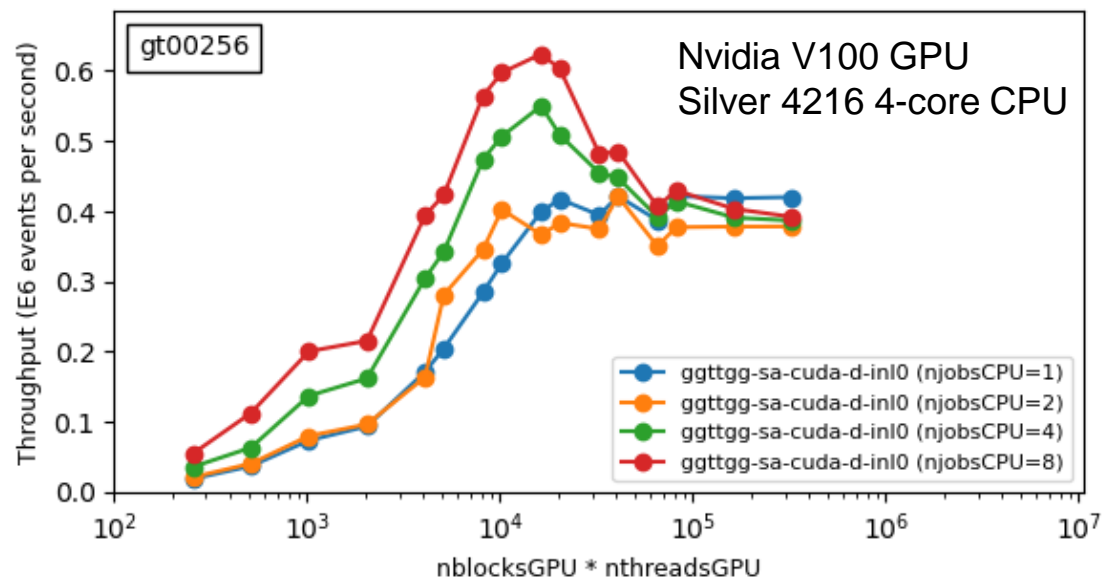
Most of these required some changes to the input/output API of our **Fortran-to-CUDA/C++ “Bridge”**

- *Helicity filtering* – at initialization time, compute the allowed combinations of particle helicities
 - This is computed in CUDA/C++ using the same criteria as in Fortran
- *“Multi-channel”* – single-diagram enhancement of ME output
 - This is the specificity of the MadEvent sampling algorithm (Maltoni Stelzer 2003)
$$f_i = \frac{|A_i|^2}{\sum_i |A_i|^2} |A_{\text{tot}}|^2$$
- Event-by-event *running QCD coupling constants* $\alpha_s(Q^2)$
 - The scale is currently computed in Fortran from momenta and passed to the CUDA/C++ for each event
- Event-by-event *choice of helicity and color* in LHE files
 - Pass two additional random numbers per event from Fortran to CUDA/C++, retrieve helicity and color
 - **NEW (January 2023)!** This was the last big missing physics functionality (showstopper to a release)
 - We now get the same cross section AND the same LHE files (within numerical precision) in Fortran and CUDA/C++

Benchmarking – Madgraph and the HEP-SCORE project

- HEPscore: the new HEP benchmark for compute resources, replacing HepSpec06
 - Based on *reproducible HEP workloads* (GEN, SIM, DIGI, REC...) within docker containers
 - The first version HEPscore23 should become production in April 2023 for (x86 and ARM) CPUs
- The aim is to *benchmark a fully loaded server*: all CPU cores, and eventually all associated GPUs
 - (and ideally measure how well an application is doing compared to the theoretical power of the server...)
 - fill all CPU cores by a combination of application multi-threading and/or several identical copies/processes
- A first container based on our Madgraph-on-GPU has been prepared
 - Very useful because it gives the same physics results on CPU and GPU: may compare them to each other!
 - And eventually may be used to evaluate heterogeneous processing on CPU+GPU...
- *The plots on the next slides are based on this HEPscore container: several identical copies/processes*
 - (A multi-threaded CUDACPP version exists but not optimized yet – SYCL and Kokkos also provide MT)

Some ideas for heterogeneous processing



Throughput variation as a function of GPU grid size (#blocks * #threads)

This is the number of events processed in parallel in one cycle

To further reduce the relative overhead of the scalar Fortran MadEvent - parallelize it on many CPU cores?

- Blue curve: one single CPU process using the GPU
 - For $gg \rightarrow t\bar{t}gg$, you need at least $\sim 16k$ events to reach the throughput plateau
- Yellow, Green, Red curves: 2, 4, 8 CPU processes using the GPU at the same time
 - *Fewer events in each GPU grid are needed to reach the plateau if several CPU processes use the GPU*
 - The total Fortran RAM would remain the same, but the CPU time in the Fortran overhead would be reduced
 - (Why total throughput increases beyond the nCPU=1 plateau is not understood yet!...)

Lockstep beyond event-level parallelism

- Efficient data parallelism (lockstep processing) requires the *same function computed for different data*
 - This is true in MG5AMC at the *event level* (different events i.e. different phase space points)
 - But it is also true at the *sub-event level* (different helicities within the same event)
- We are evaluating the move to a different data parallelism strategy on GPUs
 - Currently: *one event (sum over all helicities) per GPU thread*
 - In the future: *one helicity of one event per GPU thread?*

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{\text{hel}\}} \left[\sum_{f,g} (J_\lambda(\vec{p}))^f (C)^{fg} (J_\lambda^*(\vec{p}))^g \right] \quad (J_\lambda(\vec{p}))^f = \sum_{d \in \{\text{diag}\}} (\mathcal{M}_\lambda^d(\vec{p}))^f$$

- Advantages:
 - You can fill the GPU with much fewer “events in flight” – more balanced sampling/integration in MadEvent
 - This is a prerequisite for moving the color matrix to externally-launched cuBLAS and tensor cores
 - This is also a prerequisite if we want to evaluate much smaller kernels
 - *From all Feynman diagrams in one kernel to one Feynman diagram per kernel?*
 - Which might decrease register pressure and increase kernel occupancy, but would require more global memory access

Issue #2

Data-parallel paradigms (GPUs and vectorization)

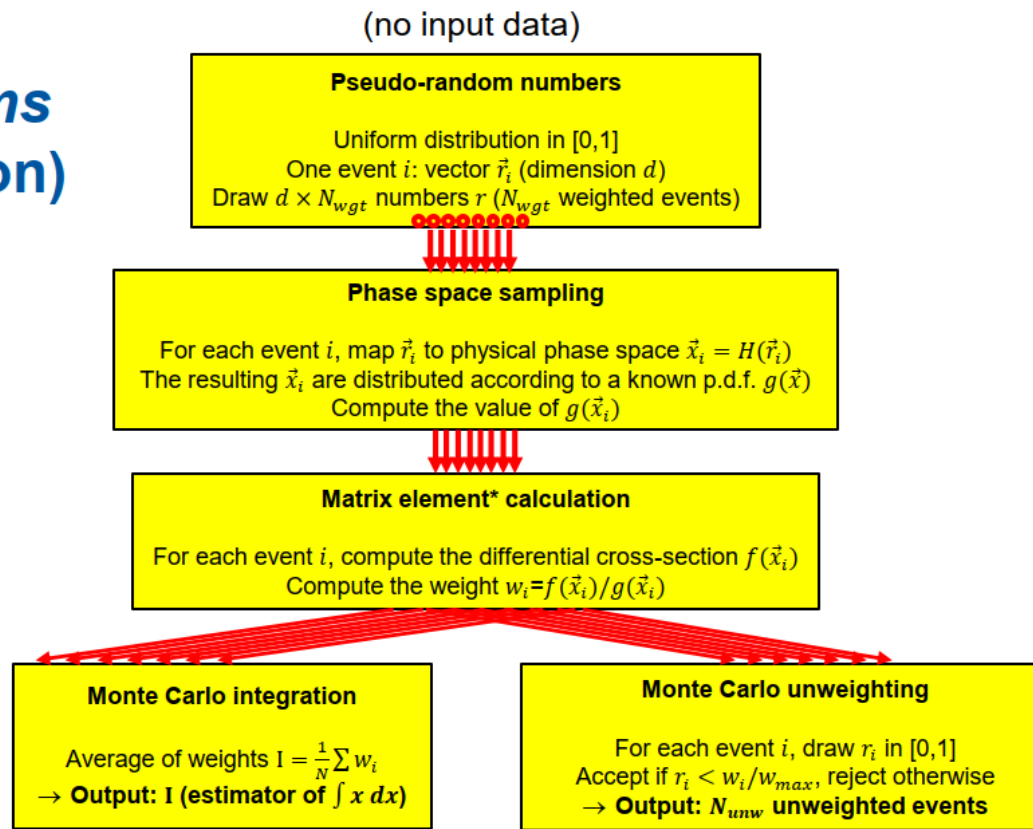
Generators lend themselves naturally to exploiting event-level parallelism via **data-parallel paradigms****

- **SPMD**: Single Program Multiple Data (GPU accelerators)
- **SIMD**: Single Instruction Multiple Data (CPU vectorization: AVX...)
- The computationally intensive part, the matrix element $f(\vec{x}_i)$, is the same function for all events i (in a given category of events)
- Unlike detector simulation (where if/then branches are frequent and lead to thread divergence on GPUs)

Potential interest of GPUs

- Faster (cheaper?) than on CPUs
- Exploit GPU-based HPCs

WIP for MG5aMC on GPUs (planned WG talk) – see next slide



*Note for software engineers: these calculations do involve some linear algebra, but “matrix element” does not refer to that! Here we compute one “matrix element” in the S-matrix (scattering matrix) for the transition from the initial state to the final state

**This simple event-level parallelism can also be used as the basis for task-parallel approaches (multi-threading or multi-processing)

https://doi.org/10.5281/zenodo.4028834

