Madgraph5_aMC@NLO on GPUs and vector CPUs: towards production

The 5-year journey to the
first LO release CUDACPP v1.00.00

Andrea Valassi (CERN)
on behalf of the MG5AMC CUDACPP development team

CHEP2024, Krakow, 23rd October 2024
https://indico.cern.ch/event/1338689/contributions/6015964

The collaborating teams (summer 2024)

SYCL plugin (also Intel GPUs)
WIP: integration into CUDACPP

Taylor Childers
Nathan Nichols
(ANL)

Argonne NATIONAL LABORATORY

CUDACPP plugin core development (NVidia and AMD GPUs, vectorized C++ on CPUs)

Olivier Mattelaer*
(UCL Louvain)

UCL Université catholique de Louvain

Stephan Hageboeck*
Daniele Massaro*
Stefan Roiser*
Zenny Wettersten*
Jorgen Teig* (2023)
Filip Optolowicz (2023)
(CERN IT-GOV-INN)

CERN

Andrea Valassi*
(CERN IT-GOV-ENG)

CERN

**CUDACPP plugin AUTHORS*

CMS integration tests
See Jin's poster for details!

Jin Choi
(Seoul National University)
Saptaparna Bhattacharya
(Wayne State University)
Robert Schoefbeck
(HEPHY Vienna)

CMS

- **Focus of this CHEP2024 talk: first release of the CUDACPP plugin**
  - will give only minimal details about the parallel work on the SYCL plugin
  - will show work done for CMS – but see Jin's poster for details on the work in CMS!

Motivation and overview

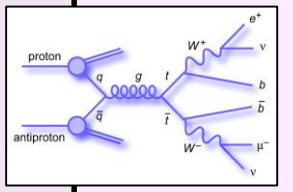
Event generators: the first step in the HEP simulation chain

Theoretical physics (Feynman diagrams)

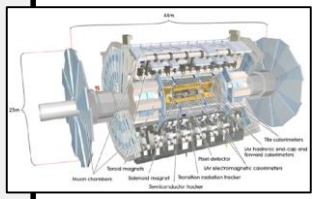
Monte Carlo methods (random numbers)

SIMULATED DATA PROCESSING ("MONTE CARLO")

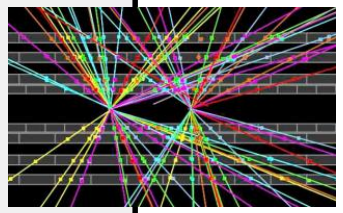
MC EVENT GENERATION (MC DATA)
Simulate physics process in beam collisions
Output: particles produced in beam collision



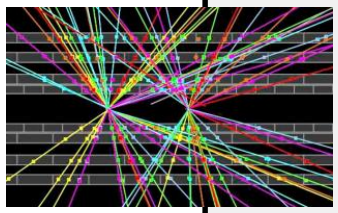
MC SIMULATION + DIGITIZATION (MC DATA)
Simulate interaction of collision products with detector
Output: simulated electronic signals



RECONSTRUCTION (MC DATA)
Translate electronic signals to particles passing through the detector

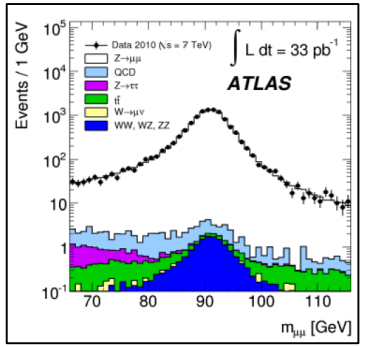


REAL DATA PROCESSING

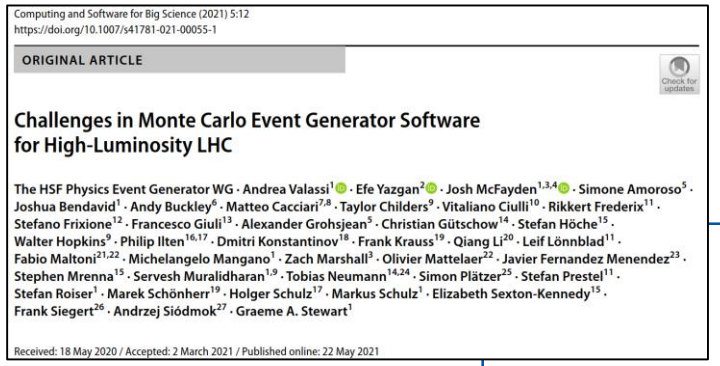


RECONSTRUCTION (REAL DATA)
Translate electronic signals to particles passing through the detector

ANALYSIS
Compare **real data** and **MC data** with statistical methods – measure parameters, search for new processes



Event generators (1): why accelerate them?



Computing and Software for Big Science (2021) 5:12
https://doi.org/10.1007/s41781-021-00055-1

ORIGINAL ARTICLE

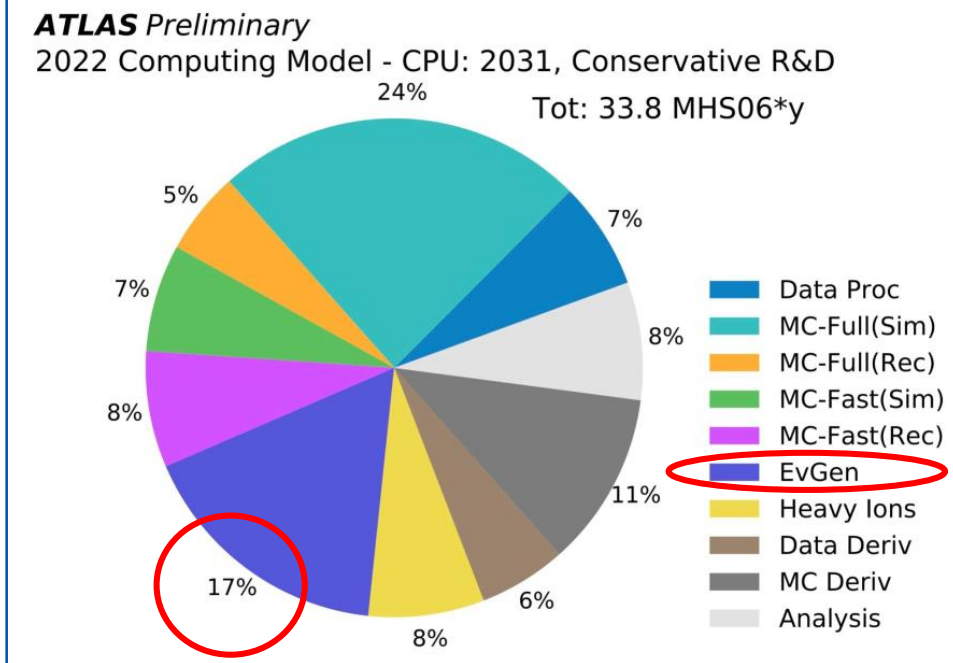
Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC

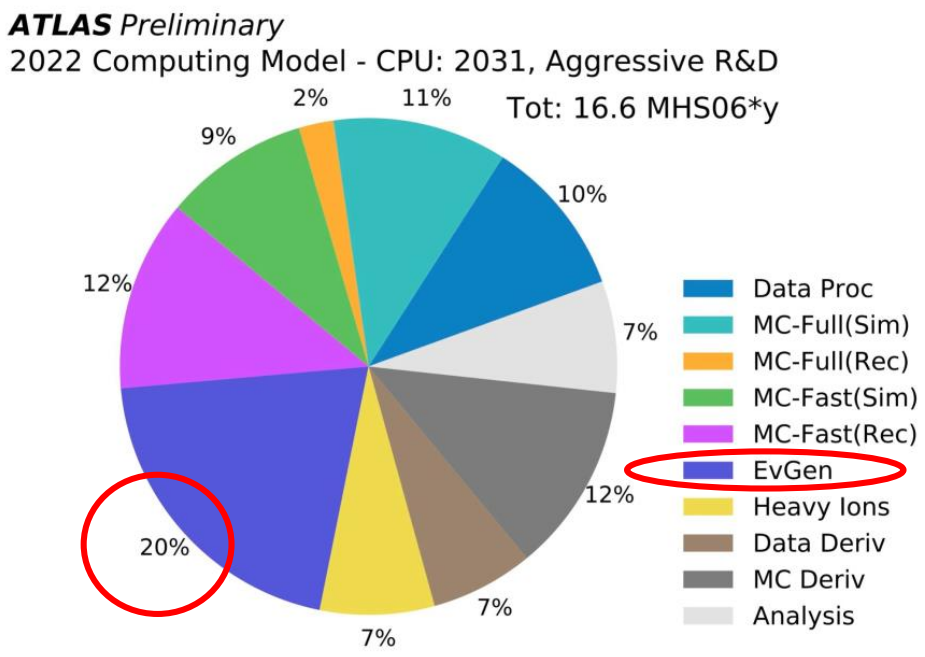
The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · Christian Gütschow[14] · Stefan Höche[15] · Walter Hopkins[9] · Philip Ilten[16,17] · Dmitri Konstantinov[18] · Frank Krauss[19] · Qiang Li[20] · Leif Lönnblad[11] · Fabio Maltoni[21,22] · Michelangelo Mangano[1] · Zach Marshall[3] · Olivier Mattelaer[22] · Javier Fernandez Menendez[23] · Stephen Mrenna[15] · Servesh Muralidharan[1,9] · Tobias Neumann[14,24] · Simon Plätzer[25] · Stefan Prestel[11] · Stefan Roiser[1] · Marek Schönherr[19] · Holger Schulz[17] · Markus Schulz[1] · Elizabeth Sexton-Kennedy[15] · Frank Siegert[26] · Andrzej Siódmok[27] · Graeme A. Stewart[1]

Received: 18 May 2020 / Accepted: 2 March 2021 / Published online: 22 May 2021

Around 10-20 % of LHC computing CPU costs (hence: important to speed them up!)

ATLAS Software and Computing HL-LHC Roadmap, version 2.1





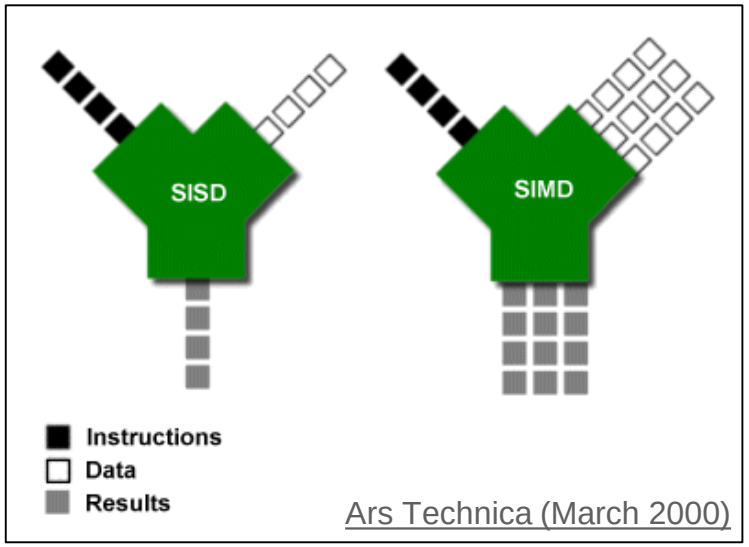
CERN-LHCC-2022-005

See also
Olivier Mattelaer's
plenary talk
on Thursday

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:

GPUs – "SIMT"
~Easier to code **(CUDA)**
SOAs not strictly needed
Tolerate lockstep <100%

Vector CPUs – SIMD
More difficult to code **(C++)**
SOAs strictly needed
Need strict 100% lockstep

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...)

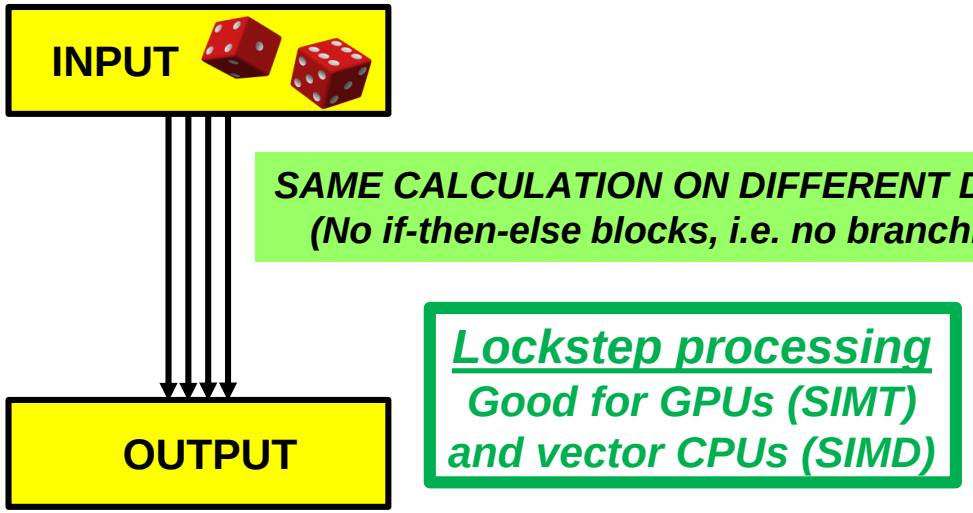
Event generators (2): why CPU vectorization and GPUs?

- **Vector CPUs and GPUs are widely available to HEP now...**
  - All of the CPUs in our computing Grid have SIMD (most have at least AVX2)
  - GPUs are becoming more and more available to us especially at HPC centers

See also Andrea Sciabà's plenary talk on Thursday

- **... but they are generally very difficult to exploit in most HEP software ☹**
  - Example: Monte Carlo detector simulation has a lot of stochastic branching (makes lockstep processing difficult)

- **However: matrix element event generators are ideal software workflows for SIMD and GPUs!**
  - Monte Carlo sampling of many data points → *Data parallelism with near-perfect lockstep processing!*



SAME CALCULATION ON DIFFERENT DATA!
(No if-then-else blocks, i.e. no branching)

Lockstep processing
Good for GPUs (SIMT)
and vector CPUs (SIMD)

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

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

For each event:

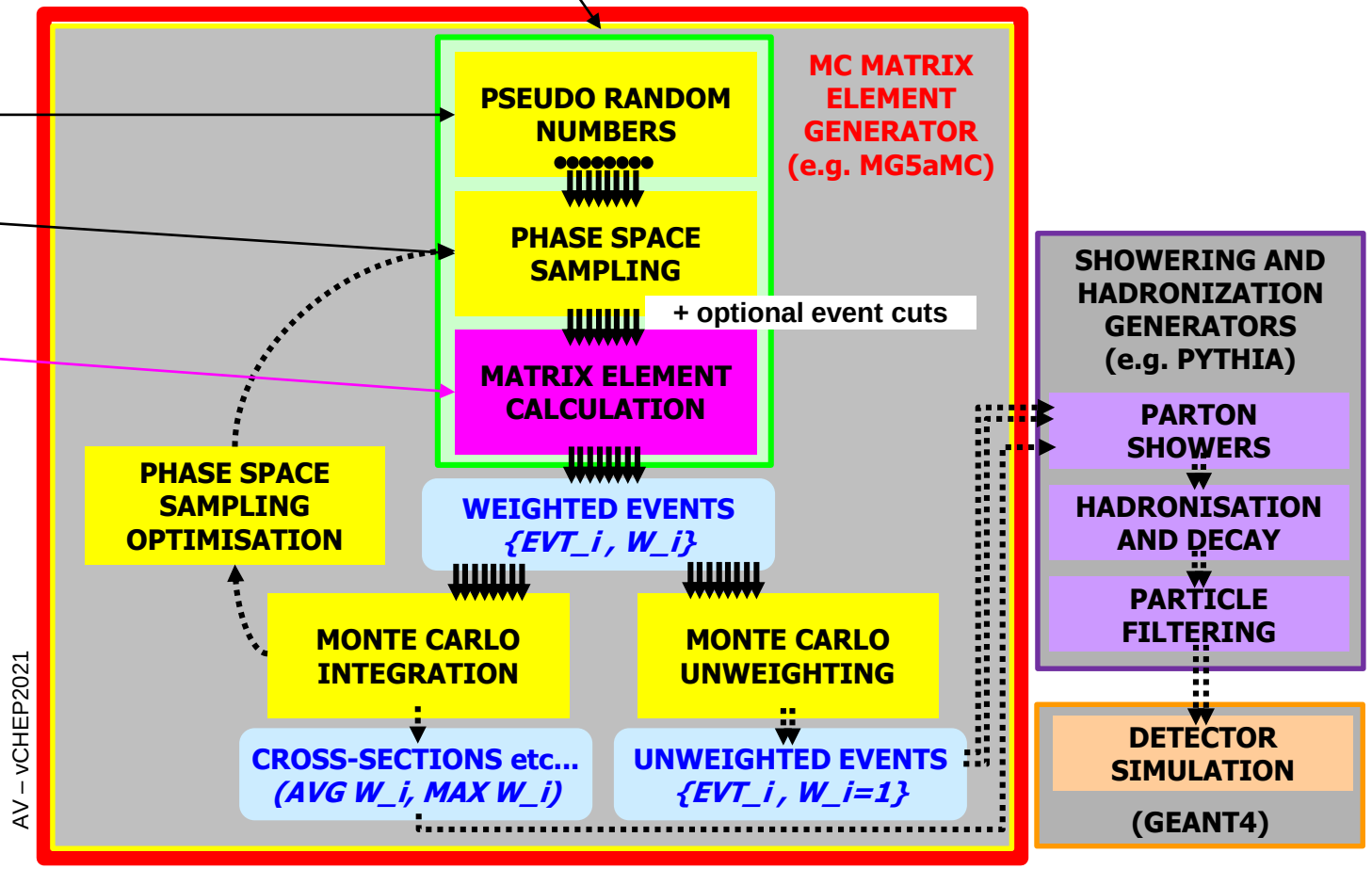
1. RANDOM NUMBERS
Output: random numbers

2. PHASE SPACE SAMPLING
Input: random numbers
Output: particle 4-momenta

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

CUDACPP: speed up the ME calculation using GPUs and SIMD

*(NB: "Matrix Element" is an element of the **scattering matrix**... not a linear algebra concept!)*

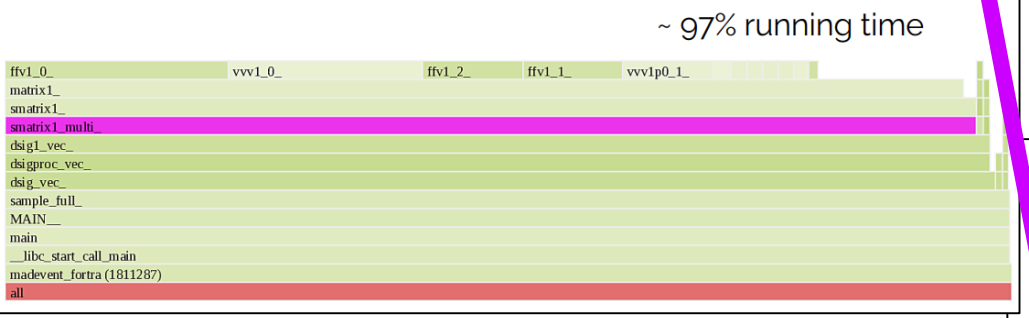


(FOR LATER!) Physics output: cross-section and LHE event file

In a nutshell: we speed up the “matrix element” (ME) calculation

$gg \to t\bar{t}gg$: FORTRAN

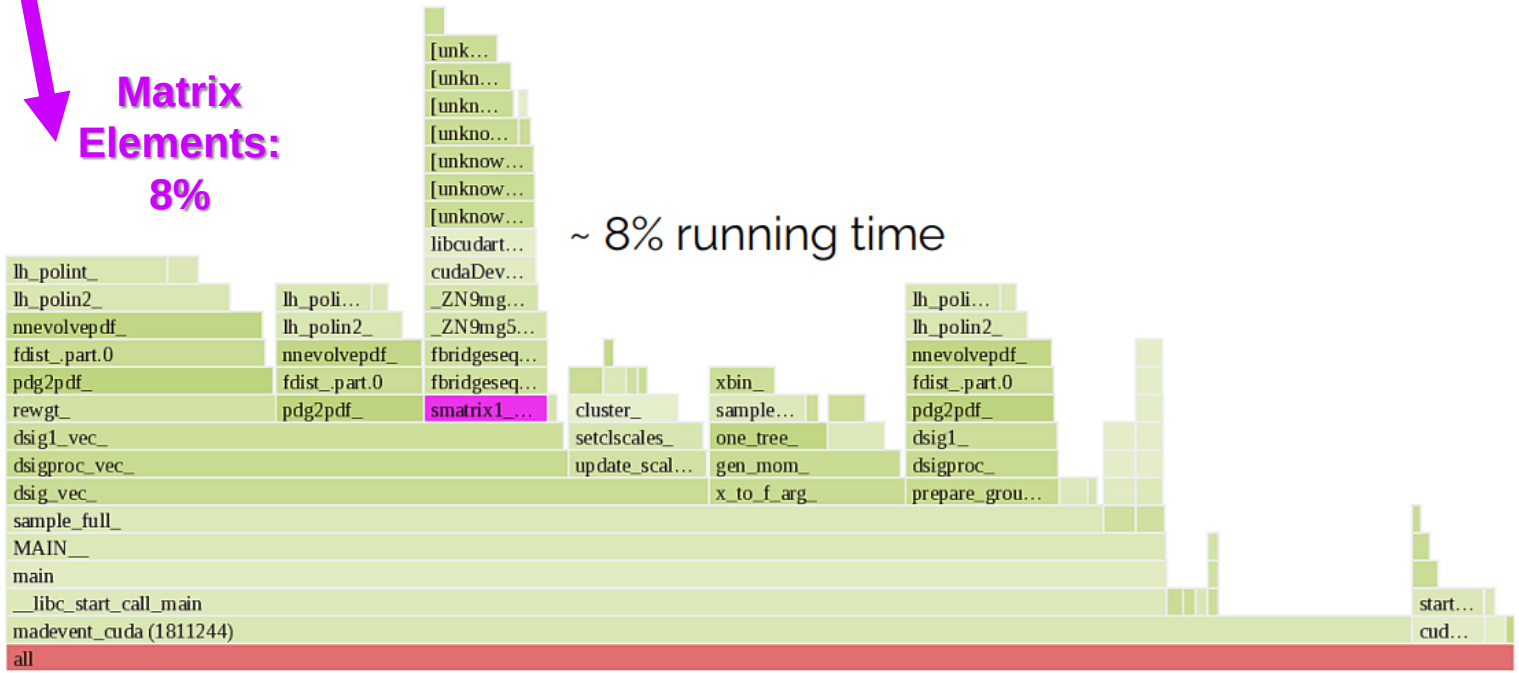
Matrix Elements: 97%



- In the old Fortran implementation, the ME calculation was the bottleneck
- *Using GPUs/SIMD we speed up MEs so much that previously unimportant components become the bottleneck!*
  - Phase space sampling, pdf’s, ...
  - As predicted by *Amdahl’s law (later...)*

$gg \to t\bar{t}gg$: CUDA

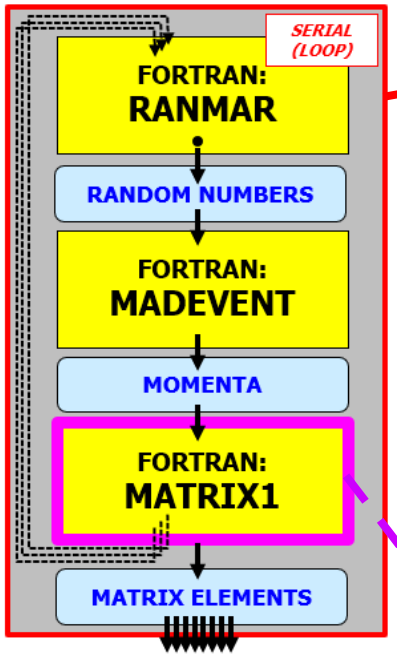
Matrix Elements: 8%



Architecture overview

MG5AMC: from single-event to multi-event APIs

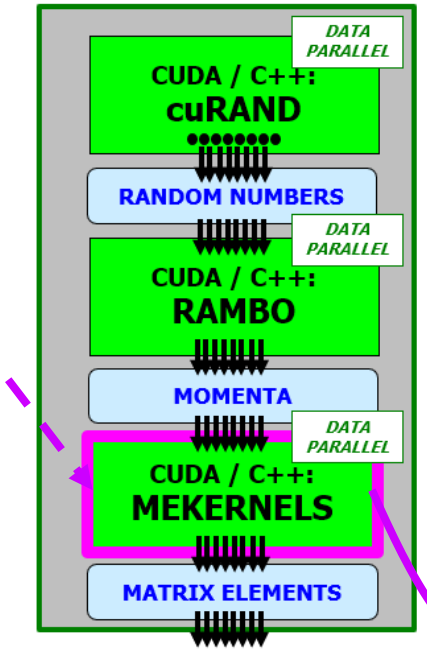
OLD madevent APPLICATION SINGLE-EVENT MEs (< 2020 or < MG 3.6.0)



MATRIX ELEMENT: CPU BOTTLENECK IN OLD madevent

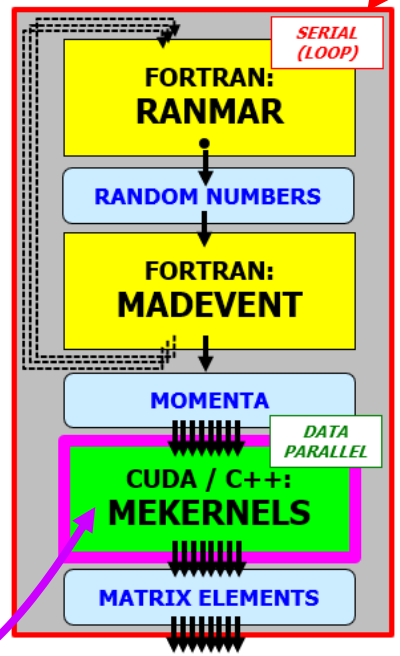
First we developed the new ME engines in **standalone applications**

1. STANDALONE (TOY) APPLICATION MULTI-EVENT MEs (2020-2021)



Then we modified the existing all-Fortran madevent application **to use *multi-event* APIs**, and **we injected CUDACPP MEs** into it (to replace Fortran MEs)

2. NEW madevent APPLICATION MULTI-EVENT MEs (2022)

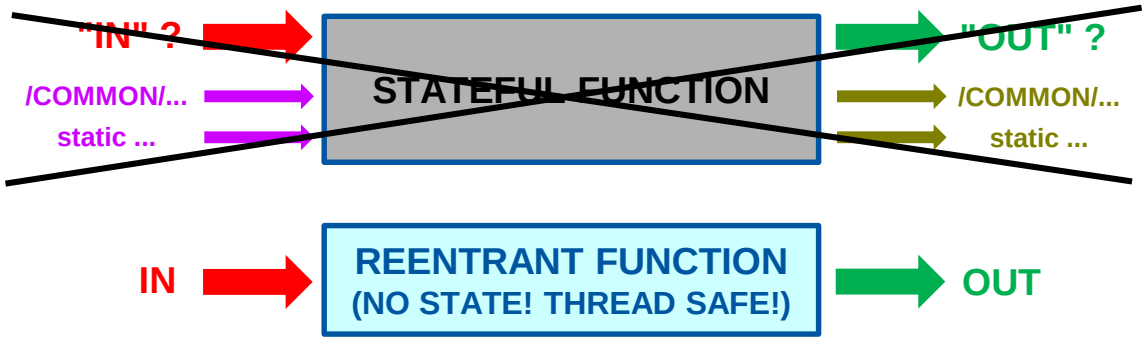


SCALAR: NEW BOTTLENECK (Amdahl's law)

PARALLEL: MUCH FASTER!

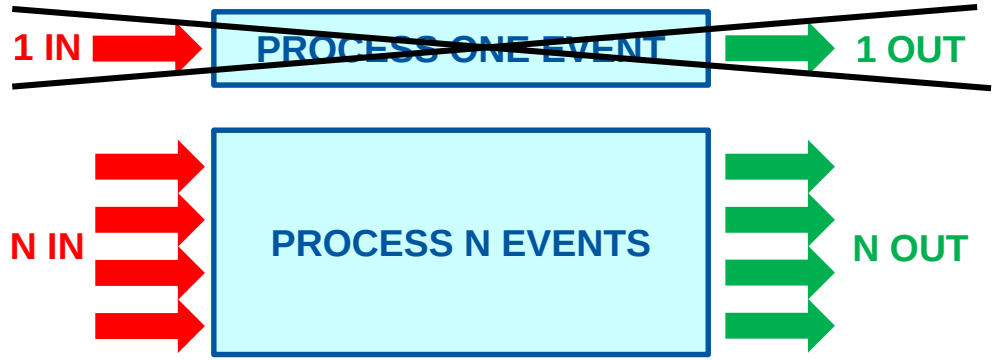
Do's and dont's - two simple lessons learnt for any MC generator

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



IMO, within MG5AMC this remains an important issue that complicates the porting to GPU/SIMD of non-ME components like phase space sampling...

- (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

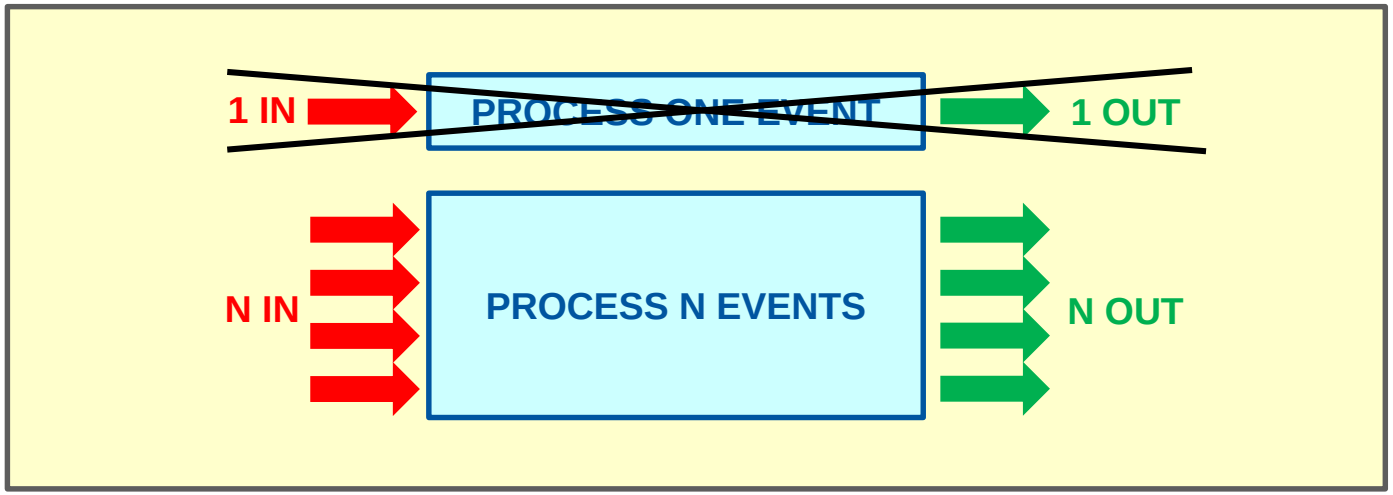


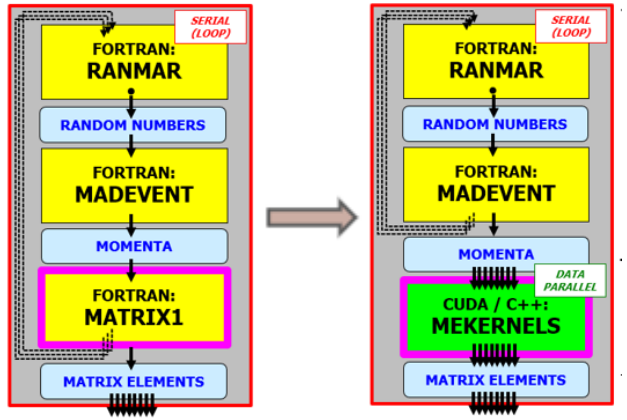
An additional technicality: prefer Structure-of-Array (SOA) memory layouts for the inputs and outputs!
[Strictly needed only internally for SIMD and useful for GPUs, but good to have also in the API of the function]

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 the work necessary on the madevent Fortran framework (to interface to the CUDACPP "bridge")***
- 2. **MG5AMC at NLO**: the ongoing work described in the next talk by Zenny!
  - The general idea (and possibly the interface of the CUDACPP "bridge") remains the same at NLO as at LO
- 3. **POWHEG + MG5AMC**: the work we plan to collaborate with!
  - This is the work the POWHEG team would need to do on their framework (to interface to the MG5AMC CUDACPP "bridge")



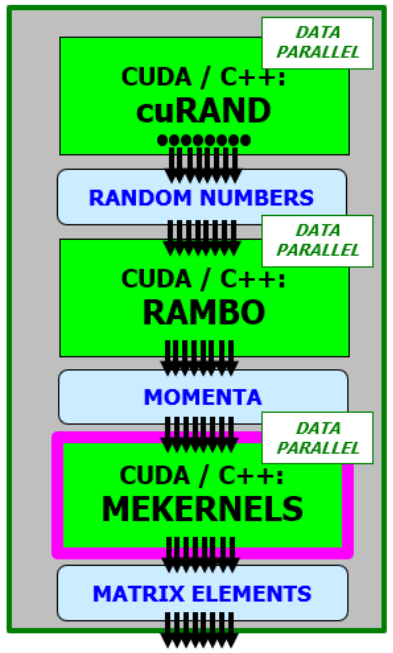


MG5AMC: CUDA/C++, Fortran, bash, python...

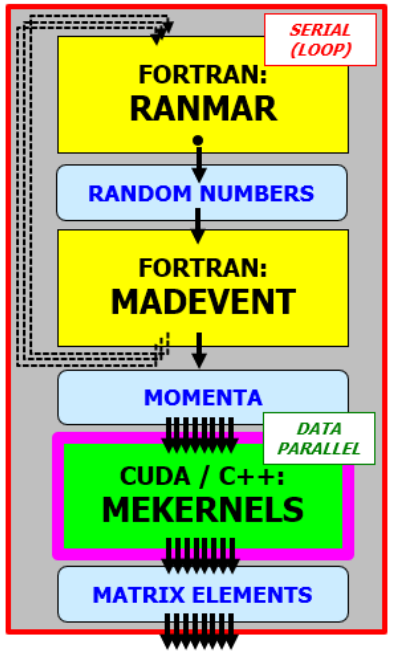
Initially (2020-2022) we focused on individual applications

In the last two years (2023-2024) we focused more and more on the full workflow orchestrating many applications
- testing/optimizing the sharing of work in many processes
- integrating the full user workflow including installation

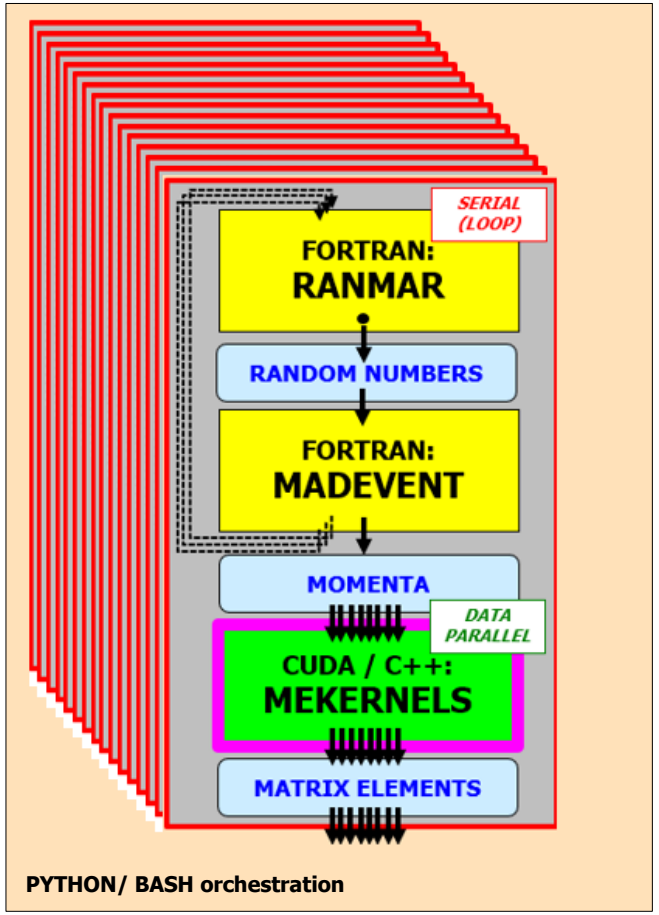
1. ONE STANDALONE TOY APPLICATION
(2020-2021)



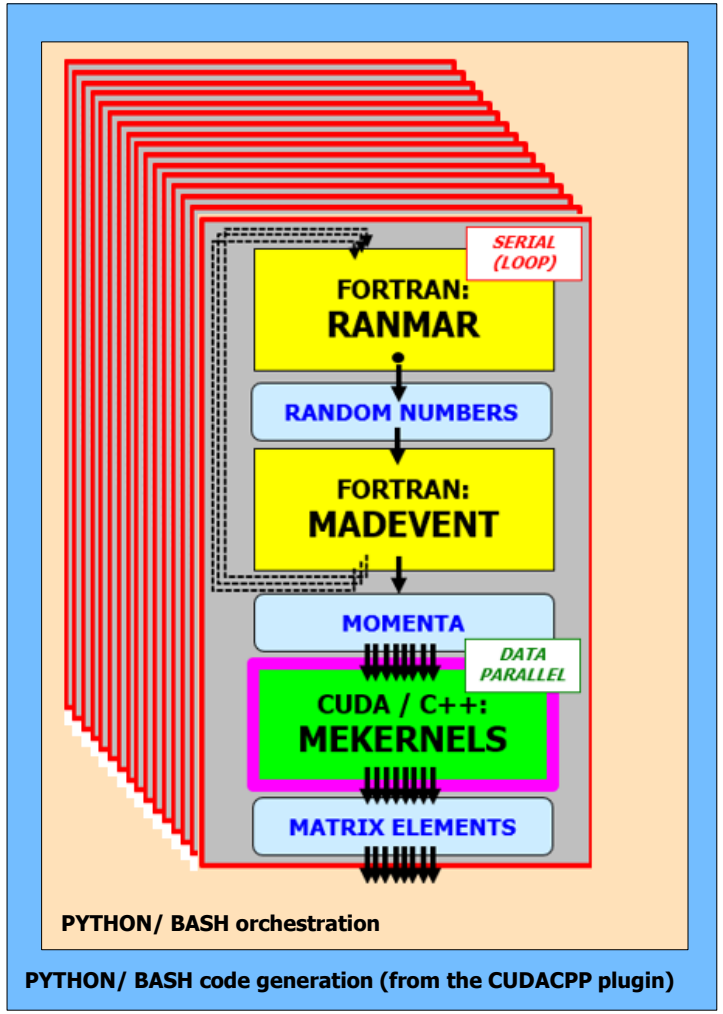
2. ONE madevent APPLICATION
(2022)



3. MANY madevent APPLICATIONs
./bin/generate_events
(2023)



4. FULL WORKFLOW
./bin/mg5_aMC
install cudacpp;
generate...; output...; launch
(2024)

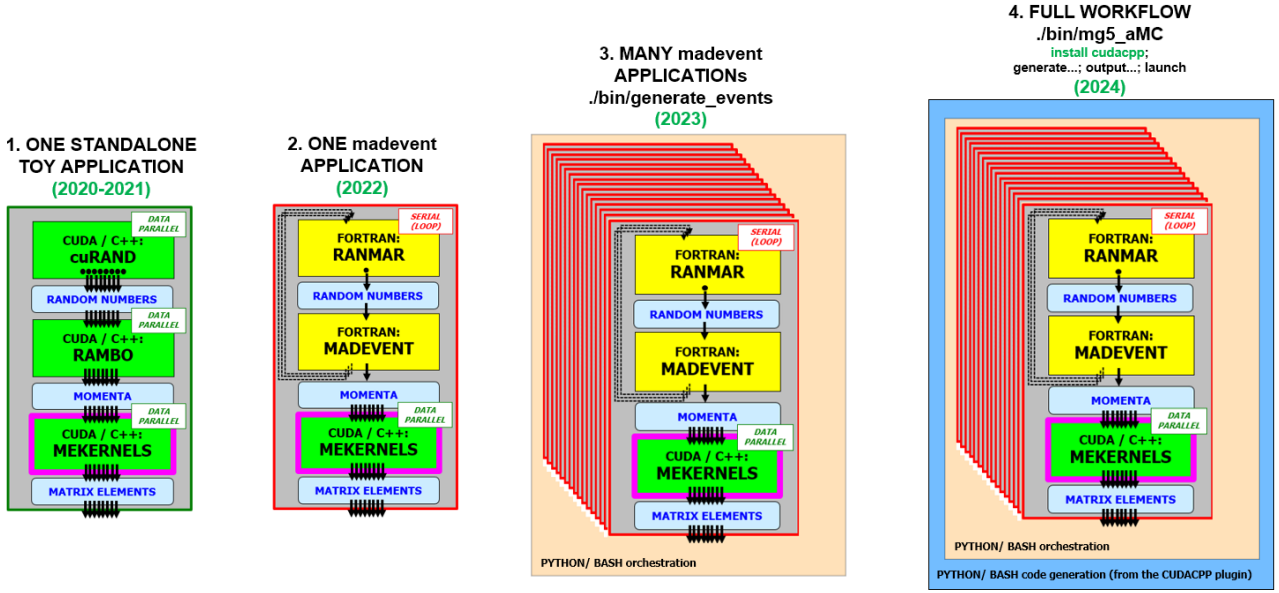


Test driven development

```
*** (2-512z) Compare MADEVENT_CPP x1 xsec to MADEVENT_FORTRAN xsec ***

OK! xsec from fortran (47.138611968034162) and cpp (47.138611968034169) differ by less than 3E-14 (2.220446049250313e-16)

*** (2-512z) Compare MADEVENT_CPP x1 events.lhe to MADEVENT_FORTRAN events.lhe reference (including colors and helicities) ***

OK! events.lhe.cpp.1 and events.lhe.ref.1 are identical
```

- 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)
  - One standalone application: use hardcoded random seeds, compare momenta and MEs to reference files (googletest)
  - One madevent application: *use* ***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!*
    - This test has been essential for identifying and later fixing a large number of important bugs
  - *New (2024): the two tests above are now in the CI for many physics processes, including automatic code generation in the CI*
  - Under development: full workflow with many madevent applications, compare overall cross sections and LHE files as above



Test a large phase space of development environments!
- Different physics processes
- Different vectorization scenarios
- Different floating point precisions
- Different compilers and O/S
- ...

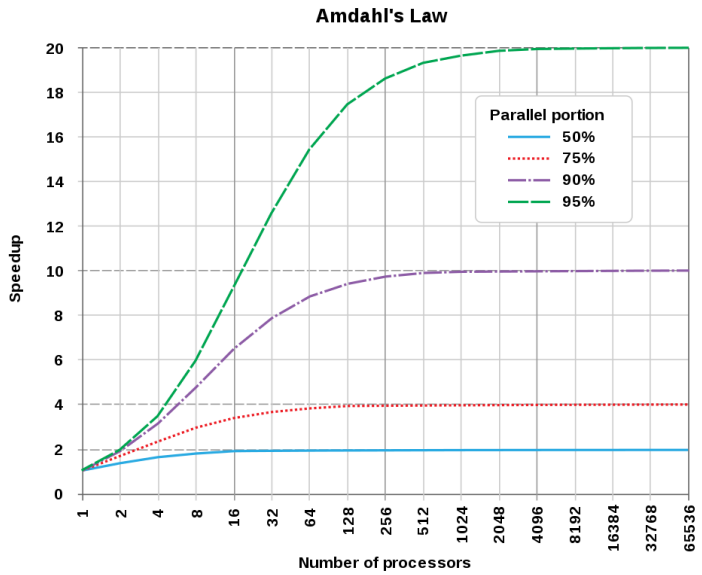
Ideally, we will try to port all these ad-hoc manual tests to the CI

Some results and new developments

Amdahl's law – not a theoretical possibility, we see it all the time!

- The matrix element (ME) calculation was the bottleneck >95% for many processes in Fortran Madgraph
  - But non-ME part <5% HAS become the bottleneck after we managed to accelerate MEs by factors O(10-1000)!

- ***Amdahl's law: if the parallelizable part takes a fraction of time p, the maximum speedup is 1/(1-p)***
  - If the non-ME part takes 5%, the maximum speedup is limited to x20 even when the ME speedup is infinite!



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

A few examples on the following slides:

- $gg \to t\bar{t}ggg$: Fortran MEs ~ 99.5% => max speedup is x200
- $gg \to t\bar{t}gg$: Fortran MEs ~ 95% => max speedup is x20
- Drell-Yan+3j: Fortran MEs ~ 66% => max speedup is x3

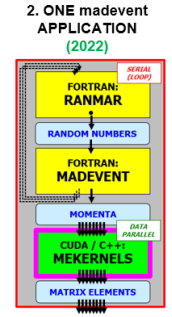
"Complex" physics process
MEs remain the bottleneck
Useful to further speedup the MEs

"Simple" physics process
MEs are no longer the bottleneck
Need to speed up the non-ME part

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

Results for $gg \to t\bar{t}ggg$ from CUDA on NVidia V100 GPUs

		madevent			standalone
CUDA grid size		8192			16384
$gg \to t\bar{t}ggg$	MEs precision	$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]	
Fortran	double	998.1 = 4.4 + 993.7	8.21E1 (=1.0)	8.24E1 (=1.0)	—
CUDA/GPU	double	16.8 = 5.9 + 10.9	4.88E3 (x60)	7.54E3 (x92)	9.54E3 (x115)
CUDA/GPU	mixed	14.3 = 5.7 + 8.6	5.72E3 (x70)	9.49E3 (x115)	1.16E4 (x141)
CUDA/GPU	float	10.7 = 5.4 + 5.3	7.65E3 (x94)	1.53E4 (x187)	2.16E4 (x264)



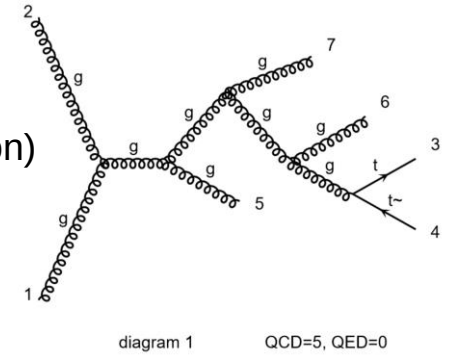
Results refer to a single CPU core

Amdahl's law: Overall speedup < ME speedup

Overall speedup x60 (double) and x90 (float) with GPU MEs over scalar Fortran

ME speedup x90 (double) and x180 (float) with GPU MEs over scalar Fortran

This is a "complex" physics process
Even after GPU acceleration, MEs remain the bottleneck (11s out of 17s in double precision)
Trying to further optimise the ME calculation is still useful to obtain further overall speedups
Example: increase the GPU grid size (requires work on Fortran too), smaller kernels, etc...



$gg \to t\bar{t}ggg$
subprocess of $pp \to t\bar{t}+3jets$)
1240 Feynman diagrams
120x120 color matrix

Results for gg→tt̄ggg from vectorized C++ on Intel Gold CPUs

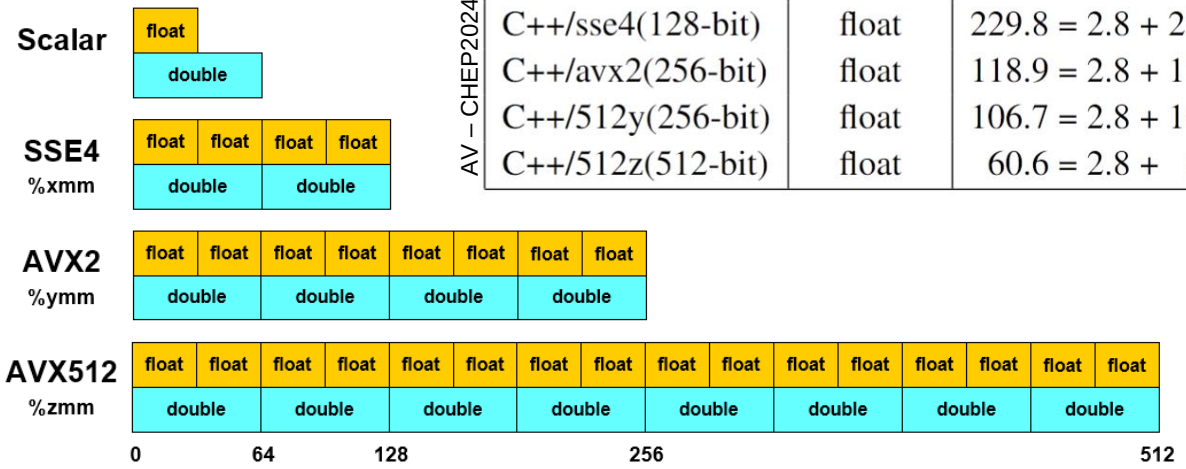
Intel Xeon Gold 6326
(2 FMA units for AVX512)

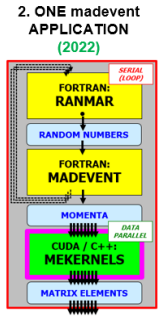
512y = AVX512, ymm registers
512z = AVX512, zmm registers

The latter is only better on nodes with 2 FMA units (here an Intel Gold 6326)

$gg \to t\bar{t}ggg$	MEs precision	madevent $t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	madevent N_{events}/t_{TOT} [events/sec]	madevent N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	854.1 = 2.8 + 851.3	9.59E1 (=1.0)	9.62E2 (=1.0)
C++/none(scalar)	double	970.8 = 3.0 + 967.9	8.44E1 (x0.9)	8.46E1 (x0.9)
C++/sse4(128-bit)	double	506.8 = 2.9 + 503.9	1.62E2 (x1.7)	1.63E2 (x1.7)
C++/avx2(256-bit)	double	235.3 = 2.8 + 232.5	3.48E2 (x3.6)	3.52E2 (x3.7)
C++/512y(256-bit)	double	209.6 = 2.8 + 206.8	3.91E2 (x4.1)	3.96E2 (x4.1)
C++/512z(512-bit)	double	116.8 = 2.8 + 114.0	7.01E2 (x7.3)	7.19E2 (x7.5)
C++/none(scalar)	mixed	983.6 = 3.0 + 980.6	8.33E1 (x0.9)	8.35E1 (x0.9)
C++/sse4(128-bit)	mixed	491.5 = 2.9 + 488.7	1.67E2 (x1.7)	1.68E2 (x1.7)
C++/avx2(256-bit)	mixed	227.8 = 2.8 + 199.2	3.60E2 (x3.8)	3.64E2 (x3.8)
C++/512y(256-bit)	mixed	202.0 = 2.8 + 199.2	4.05E2 (x4.2)	4.11E2 (x4.3)
C++/512z(512-bit)	mixed	116.6 = 2.8 + 113.8	7.03E2 (x7.3)	7.20E2 (x7.5)
C++/none(scalar)	float	943.5 = 3.0 + 940.6	8.68E1 (x0.9)	8.71E1 (x0.9)
C++/sse4(128-bit)	float	229.8 = 2.8 + 227.0	3.56E2 (x3.7)	3.61E2 (x3.7)
C++/avx2(256-bit)	float	118.9 = 2.8 + 116.0	6.89E2 (x7.2)	7.06E2 (x7.3)
C++/512y(256-bit)	float	106.7 = 2.8 + 103.9	7.68E2 (x8.0)	7.89E2 (x8.2)
C++/512z(512-bit)	float	60.6 = 2.8 + 57.8	1.35E3 (x14.1)	1.42E3 (x14.7)

AV – CHEP2024





Results refer to a single CPU core

ME speedup x8 (double) and x15 (float) with AVX512 MEs over scalar Fortran

Our ME engine reaches the maximum theoretical SIMD speedup!
This is because we have perfect lockstep during most of the ME calculation!

Amdahl's law: Overall speedup < ME speedup

Overall speedup x7 (double) and x14 (float) with GPU MEs over scalar Fortran

Floating point precision → constraints on GPU hardware

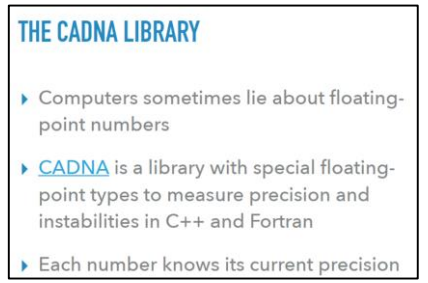
- Previous slides: if we could use floats instead of doubles, our MEs would be a factor 2x faster!
  - Our vectorized C++ is 2x faster on CPU (e.g. AVX512: a 512-bit register holds 16 floats but only 8 doubles)
  - Our CUDA is 2x faster on V100's (on NVidia data-centre GPUs, the FP64 FLOPs are x1/2 the FP64 FLOPs)

- But ***we need double precision for Feynman diagrams*** (single precision gives numerical instabilities)
  - This means that *we cannot use consumer-grade GPUs* (on T4's, the FP64 FLOPs are x1/32 the FP32 FLOPs)
  - Also: *GPUs for AI like Blackwell GB200 do have (a lot of!) FP64, but what you pay are FP4 tensor core FLOPs!*
    - (En passant: in CUDACPP we do NOT use tensor cores at all – these require a different software API than CUDA cores)

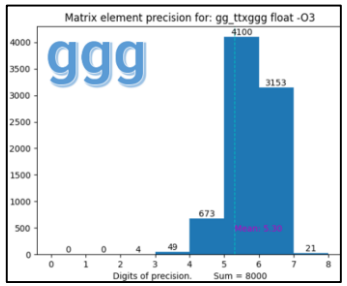
Architecture	Blackwell	Hopper	Ampere	Volta
Year	2024	2022	2020	2017
GPU Name	NVIDIA GB200	NVIDIA H100	NVIDIA A100	NVIDIA V100
FP64	90 teraFLOPS	34 teraFLOPS	9.7 teraFLOPS	7.8 teraFLOPs
FP32	180 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS	15.7 teraFLOPs
BF16	N/A	134 teraFLOPS	39 teraFLOPS	N/A
FP16	N/A	134 teraFLOPS	78 teraFLOPS	N/A
FP64 Tensor Core	90 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS	N/A
TF32 Tensor Core	5 petaFLOPS	989 teraFLOPS	312 teraFLOPS	N/A
FP16 Tensor Core	10 petaFLOPS	1979 teraFLOPS	624 teraFLOPS	125 teraFLOPs
BF16 Tensor Core	10 petaFLOPS	1979 teraFLOPS	624 teraFLOPS	N/A
FP8 Tensor Core	20 petaFLOPS	3958 teraFLOPS	N/A	N/A
FP4 Tensor Core	40 petaFLOPS	N/A	N/A	N/A

https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://resources.nvidia.com/en-us-blackwell-architecture

Stephan Hageboeck



Filip Optolowicz



- *We did switch to floats where possible – "mixed-precision": double for Feynman, float for color matrix*
  - This is the default in CUDAPP v1.00.00 (even if the speedup over double is limited and still to be improved)
  - We also had a closer look at the source of numerical instabilities with the CADNA tool

Beyond NVidia GPUs

NEW in 2024!

- 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
- **CUDACPP v1.00.00 includes support for AMD GPUs through HIP, using the same #ifdef approach**
  - This was inspired by the LHCb approach to GPUs
  - *NVidia and AMD provide ~80% of the GPU power in top500 HPCs*
- Our Argonne colleagues are working on extending this to Intel GPUs via SYCL (based on their earlier work on a SYCL plugin)

Accelerated Supers, June 2024 Top500			Peak		Total	
By Accelerator Type	Systems	Share	Teraflops	Share	Cores	Share
AMD MI200	11	5.7%	2,499,680	26.5%	12,757,568	25.3%
AMD MI300	3	1.6%	96,294	1.0%	387,072	0.8%
Intel GPU Max	4	2.1%	2,067,806	22.0%	9,613,760	19.0%
Nvidia A100	83	43.0%	1,462,714	15.5%	9,754,044	19.3%
Nvidia H100	29	15.0%	2,379,348	25.3%	6,599,000	13.1%
Nvidia Other	60	31.1%	899,135	9.5%	10,994,436	21.8%
Other	3	1.6%	12,870	0.1%	360,120	0.7%
Total	**193**		**9,417,847**		**50,466,000**	
All Supers	**500**		**12,499,181**		**114,650,780**	
Accelerator Share	38.6%		75.3%		44.0%	
CPU-Only Share	61.4%		24.7%		56.0%	

https://www.nextplatform.com (13 May 2024)

GPU Teraflops by GPU family (HPCs in June 2024 Top500)

- NVidia GPUs 50.3%

- AMD GPUs 27.5%

- Intel GPUs 22.0%

Joergen Teig, AV

CUDA

HIP


```
// 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.

#ifndef 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
```

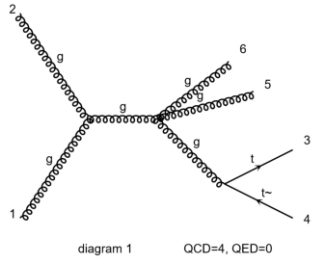
Beyond NVidia GPUs: results with AMD GPUs at LUMI

NEW in 2024!

$gg \to t\bar{t}gg$	MEs precision	madevent $t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	madevent N_{events}/t_{TOT} [events/sec]	madevent N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	26.6 = 1.4 + 25.2	3.09E3 (=1.0)	3.25E3 (=1.0)
C++/none(scalar)	double	33.2 = 1.4 + 31.2	2.47E3 (x0.8)	2.58E3 (x0.8)
C++/sse4(128-bit)	double	16.9 = 1.4 + 15.5	4.85E3 (x1.6)	5.28E3 (x1.6)
C++/avx2(256-bit)	double	8.1 = 1.4 + 6.7	1.01E4 (x3.3)	1.22E4 (x3.8)
HIP/GPU	double	2.9 = 1.8 + 1.1	2.88E4 (x9.3)	7.69E4 (x24)
C++/none(scalar)	mixed	33.2 = 1.4 + 31.8	2.47E3 (x0.8)	2.57E3 (x0.8)
C++/sse4(128-bit)	mixed	16.7 = 1.4 + 15.3	4.91E3 (x1.6)	5.36E3 (x1.6)
C++/avx2(256-bit)	mixed	8.3 = 1.4 + 6.9	9.93E3 (x3.2)	1.20E4 (x3.7)
HIP/GPU	mixed	2.9 = 1.8 + 1.1	2.88E4 (x9.3)	7.69E4 (x24)
C++/none(scalar)	float	32.1 = 1.4 + 30.8	2.55E3 (x0.8)	2.66E3 (x0.8)
C++/sse4(128-bit)	float	9.2 = 1.4 + 7.8	8.92E3 (x2.9)	1.05E4 (x3.2)
C++/avx2(256-bit)	float	4.9 = 1.4 + 3.5	1.69E4 (x5.5)	2.37E4 (x7.3)
HIP/GPU	float	2.4 = 1.8 + 0.7	3.36E4 (x11)	1.20E5 (x37)

AV – CHEP2024

$gg \to t\bar{t}gg$
(subprocess of $pp \to t\bar{t}$+2jets)
123 Feynman diagrams
24x24 color matrix



Overall speedups for $gg \to t\bar{t}gg$
- x9.3 for MEs on an AMD Instinct MI200 GPU
- x3.2 for MEs on an AMD 7A53 CPU with AVX2
(Amdahl: maximum overall speedup is x20)

One limitation: was unable to build HIP code for the more complex $gg \to t\bar{t}ggg$ process...

We kindly acknowledge the use of LUMI HPC resources under project 465001114 ("CERN / HEPiX Benchmarking GPU WP" EHPC-BEN-2024B04-053) to produce these results

Beyond Standard Model physics in MG5aMC CUDACPP

NEW in 2024!

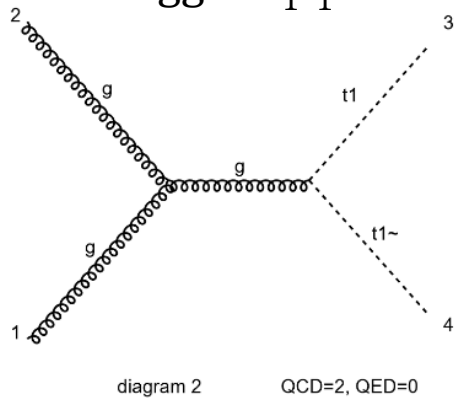
- ***CUDACPP v1.00.00 includes support for several BSM processes (at LO): SUSY, HEFT, SMEFT***

- Motivation: speed up large productions of BSM processes at LO (for many SM processes NLO is required)
  - Need event samples exploring a large parameter space (by event generation or by event reweighting)

- *Technical challenge (with respect to SM): non-standard parameters and couplings*
  - debugged/fixed non-standard code to propagate the running of the QCD coupling α_s to these BSM parameters and couplings
  - (reminder: for each event, α_s scale is computed in Fortran and passed to cuda/c++ that computes α_s-dependent parameters)

SUSY (MSSM_SLHA2)

$gg \to t_1 \bar{t}_1$

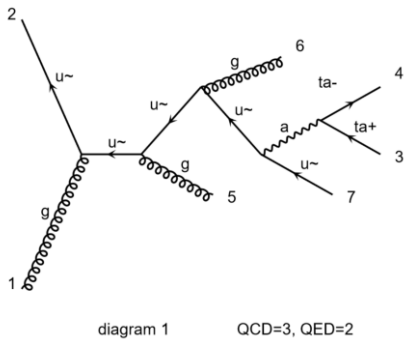


Studies with CMS: understanding Drell-Yan+3jets speedups (1)

- CMS have been the first early adopters of CUDACPP – an extremely useful, mutually beneficial, collaboration!
  - See the details of all the studies performed by/within CMS in Jin Choi's poster

- One of many issues we discussed with CMS: **what is the speedup we can achieve from cudacpp in DY+jets?**

$g\bar{u} \to \tau^+\tau^- g g \bar{u}$
(subprocess of DY+3j)
100 Feynman diagrams
6x6 color matrix



AV – CHEP2024

		madevent		
$g\bar{u} \to \tau^+\tau^- g g \bar{u}$ (81920 weighted events)	MEs precision	$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	52.2 = 17.0 + 35.2	1.57E3 (=1.0)	2.32E3 (=1.0)
C++/none(scalar)	double	50.9 = 17.0 + 33.9	1.61E3 (x1.0)	2.42E3 (x1.0)
C++/sse4(128-bit)	double	35.5 = 17.0 + 18.5	2.31E3 (x1.5)	4.44E3 (x1.9)
C++/avx2(256-bit)	double	24.5 = 16.9 + 7.6	3.34E3 (x2.1)	1.08E4 (x4.7)
C++/512y(256-bit)	double	23.9 = 16.9 + 7.0	3.43E3 (x2.2)	1.17E4 (x5.0)
C++/512z(512-bit)	double	26.7 = 17.0 + 9.6	3.09E3 (x2.0)	8.57E3 (x3.7)
CUDA/GPU	double	17.6 = 17.4 + 0.3	4.65E3 (x3.0)	3.26E5 (x140)
C++/none(scalar)	mixed	50.9 = 16.9 + 33.9	1.61E3 (x1.0)	2.41E3 (x1.0)
C++/sse4(128-bit)	mixed	33.9 = 16.9 + 17.0	2.41E3 (x1.5)	4.82E3 (x2.1)
C++/avx2(256-bit)	mixed	24.8 = 17.2 + 7.6	3.31E3 (x2.1)	1.08E4 (x4.7)
C++/512y(256-bit)	mixed	24.1 = 17.1 + 7.0	3.40E3 (x2.2)	1.18E4 (x5.0)
C++/512z(512-bit)	mixed	26.5 = 17.0 + 9.6	3.09E3 (x2.0)	8.57E3 (x3.7)
CUDA/GPU	mixed	17.7 = 17.4 + 0.3	4.64E3 (x3.0)	3.23E5 (x138)
C++/none(scalar)	float	50.1 = 16.9 + 33.1	1.64E3 (x1.0)	2.47E3 (x1.1)
C++/sse4(128-bit)	float	26.3 = 16.9 + 9.4	3.11E3 (x2.0)	8.70E3 (x3.7)
C++/avx2(256-bit)	float	20.8 = 16.9 + 3.9	3.94E3 (x2.5)	2.12E4 (x9.1)
C++/512y(256-bit)	float	20.6 = 16.9 + 3.6	3.99E3 (x2.5)	2.27E4 (x9.7)
C++/512z(512-bit)	float	21.7 = 16.9 + 4.8	3.78E3 (x2.4)	1.71E4 (x7.3)
CUDA/GPU	float	17.6 = 17.4 + 0.2	4.66E3 (x3.0)	4.46E5 (x191)

For one typical subprocess of DY+3jets:
Fortran MEs ~ 67% of the total time
=> **Max overall speedup is x3 (Amdahl)**

Achieved speedups (mixed FP precision):
- **x3.0 on GPU** (NVidia V100)
- **x2.2 on SIMD** ("512y" on Intel Silver)

NVidia V100 GPU
Intel Xeon Silver 4216
(1 FMA unit for AVX512)

Studies with CMS: understanding Drell-Yan+3jets speedups (2)

- Non-ME is 33% of overall Fortran time (max speedup x3): what exactly takes time? => Profiling!
  - The answer is: **Fortran phase space sampling is now the bottleneck for DY+3j** (93s out of 177s overall with C++ MEs)

```
pp_dy3j.mad//fortran/output.txt
[GridPackCmd.launch] GRIDPCK TOTAL    447.7169 seconds
[madevent COUNTERS]  PROGRAM TOTAL 443.48
[madevent COUNTERS]  Fortran Other 6.5439
[madevent COUNTERS]  Fortran Initialise(I/O) 4.4648
[madevent COUNTERS]  Fortran Random2Momenta 93.2692
[madevent COUNTERS]  Fortran PDFs 8.2697
[madevent COUNTERS]  Fortran UpdateScaleCouplings 7.3142
[madevent COUNTERS]  Fortran Reweight 3.6975
[madevent COUNTERS]  Fortran Unweight(LHE-I/O) 4.8636
[madevent COUNTERS]  Fortran SamplePutPoint 8.3255
[madevent COUNTERS]  Fortran MEs 306.731
[madevent COUNTERS]  OVERALL NON-MEs 136.748
[madevent COUNTERS]  OVERALL MEs 306.731
```

FORTRAN

AV – preliminary!

Overall:
- Fortran 447s
- C++ 177s *(speedup x2.5)*

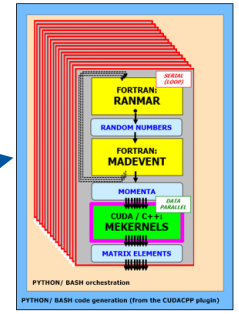
Matrix elements:
- Fortran 307s
- C++ 36s *(speedup x8)*

Non-ME (overall - ME):
- (Same for Fortran/C++)
- Total non-ME 140s
- Phase space sampling 93s

```
pp_dy3j.mad//cpp512z/output.txt
[GridPackCmd.launch] GRIDPCK TOTAL    176.8891 seconds
[madevent COUNTERS]  PROGRAM TOTAL 172.637
[madevent COUNTERS]  Fortran Other 6.5768
[madevent COUNTERS]  Fortran Initialise(I/O) 4.486
[madevent COUNTERS]  Fortran Random2Momenta 93.2907
[madevent COUNTERS]  Fortran PDFs 8.2998
[madevent COUNTERS]  Fortran UpdateScaleCouplings 7.2827
[madevent COUNTERS]  Fortran Reweight 3.7045
[madevent COUNTERS]  Fortran Unweight(LHE-I/O) 4.8719
[madevent COUNTERS]  Fortran SamplePutPoint 8.2892
[madevent COUNTERS]  CudaCpp Initialise 0.3619
[madevent COUNTERS]  CudaCpp Finalise 0.0221
[madevent COUNTERS]  CudaCpp MEs 35.4557
[madevent COUNTERS]  OVERALL NON-MEs 137.181
[madevent COUNTERS]  OVERALL MEs 35.4557
```

C++ AVX512

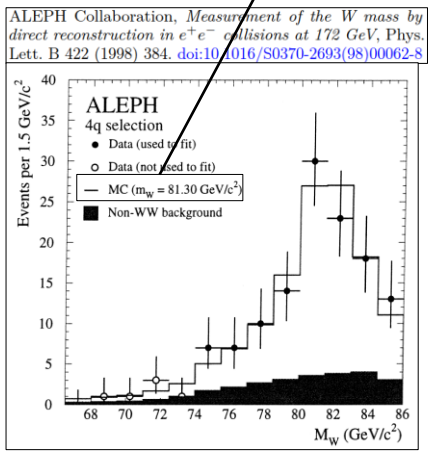
- *Profiling above is via code instrumentation* (complementary to perf/flamegraph sampling profiling)
  - Inserted low-overhead rdtsc counters in madevent internal fortran calls (still being tuned!)
  - *Keep, parse, aggregate all madevent logs from many processes* in the python/bash orchestrator
  - Not yet merged in mg5amcnlo or cudacpp but might be added in upcoming versions



Reweighting

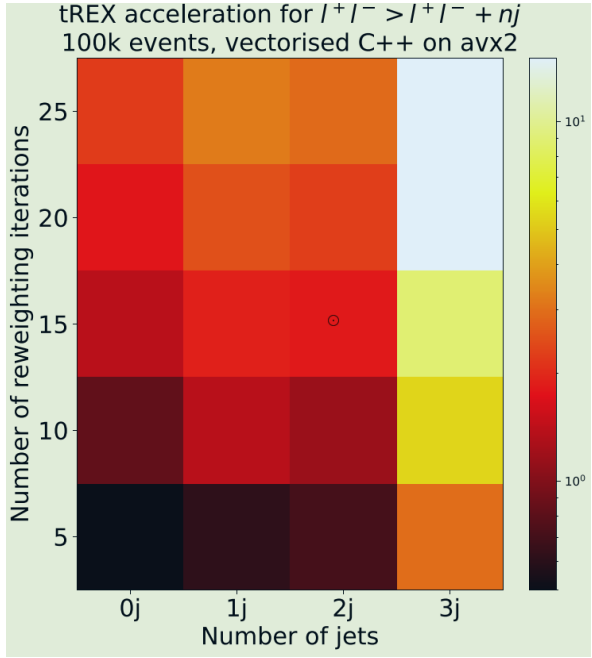
Old technique, renewed interest!

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

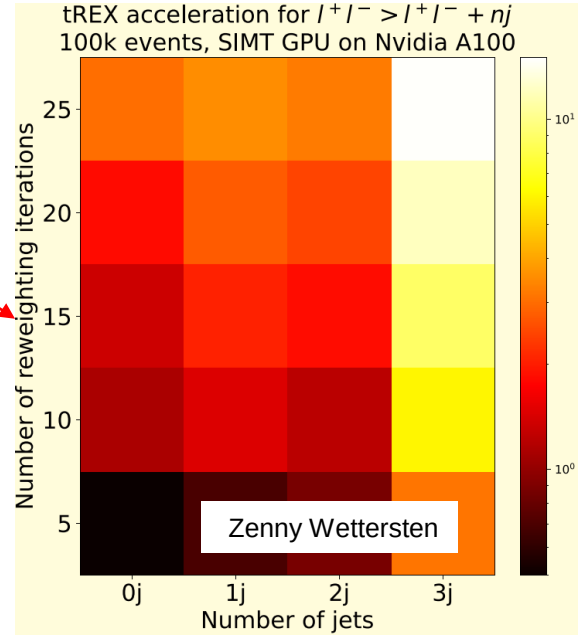


1. *Generate signal sample at θ_{ref}, with $w_i(\theta_{ref})=1$*
   (By definition, background does not depend on θ)
2. *Full detector simulation*
   (MC truth event properties $\mathbf{x}_i^{(true)} \rightarrow$ observed event properties $\mathbf{x}_i$)
3. *Reweight each event by matrix element ratio*

$$w_i(\theta) = \frac{\mathrm{Prob}_{(\theta)}(\mathbf{x}_i^{(\mathrm{true})})}{\mathrm{Prob}_{(\theta_{\mathrm{ref}})}(\mathbf{x}_i^{(\mathrm{true})})} = \frac{|\mathcal{M}(\theta, \mathbf{x}_i^{(\mathrm{true})})|^2}{|\mathcal{M}(\theta_{\mathrm{ref}}, \mathbf{x}_i^{(\mathrm{true})})|^2}$$



- *Two advantages: lower costs (no detector simulation), fewer statistical fluctuations*
  - Interest in CMS for EFT studies (exploration of large space of model parameters)
- In practice *for MG5AMC: read in LHE file, add weights, write back modified LHE file*
  - ***Reweighting using the new ME engine in CUDA/C++ provides O(x10) speedups!***
  - Zenny's "tREX" is essentially ~ready to be included in an upcoming CUDACPP v1.01



One further possible application: weight derivatives in parameter measurements
- My suggestion: save weight derivatives (w.r.t measured parameter) in LHE files
- Use them to compute ideal measurement error or for weight derivative regression
- See https://doi.org/10.1051/epjconf/202024506038 and https://zenodo.org/records/11120823

A tale of two repositories

Our work of the last 5 years is mainly here!

mg5amcnlo

https://github.com/mg5amcnlo/mg5amcnlo

- ***the* MG5AMC repo** (previously launchpad)
- python framework, fortran codegen
- permissive NCSA-style license

Users download this repository

Users install cudacpp as a tarball ←

(A specific commit is in madgraph4gpu) →

madgraph4gpu (will be moved and renamed!)

https://github.com/madgraph5/madgraph4gpu

- **the CUDACPP plugin** (cuda/c++ codegen)
- (legacy stuff – code, logs... – to be removed)
- *more restrictive LGPL license*

(Developers download this repository)

Release tags are packaged as tarballs →

(Developers find mg5amcnlo as a git submodule)

- For more details:
  - Our wiki: https://github.com/madgraph5/madgraph4gpu/wiki/Working-with-cudacpp-v1.00.00-(October-2024)
  - Our bi-weekly development meetings: https://indico.cern.ch/category/12586

The long journey to the v1.00.00 CUDACPP release – some steps

First commit! (Feb 2020) SR

initial commit eemumu 9f2d7a3

roiser committed on Feb 17, 2020

AOSOA, SIMD vectorization (Dec 2020) AV

C++ SIMD: gcc / clang compiler vector extensions

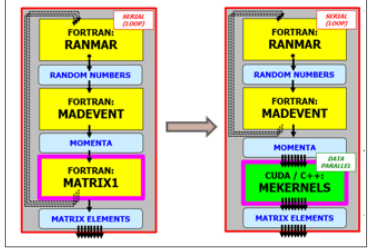
```
#ifdef __clang__
  typedef fptype fptype_v __attribute__ ((ext_vector_type(neppV))); // RRRR
#else
  typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype)))); // RRRR
#endif
```

Set up a github CI (Jan 2021) SH

[CI] Add GPU build to CI configuration.

hageboeck committed on Jan 28, 2021

Inject CUDA/C++ into Fortran (2021-2022) AV/OM/SR



NVidia / AMD GPU abstraction (2023-2024) JT/AV

Tarball install. Release tag! (Oct 2024) AV/OM

2 weeks ago

Oct 3, 2024, 12:10 PM GMT+2

cudacpp_for3.6.0_v1.00.00

Version tag cudacpp_for3.6.0_v1.00.00 (changelog)

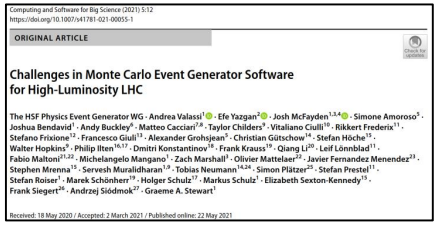
Validated for mg5amcnlo version 3.6.0 (commit 55a291d)

This is a release tag: you may install the *latest* release tag cudacpp_for3.6.0_latest as follows

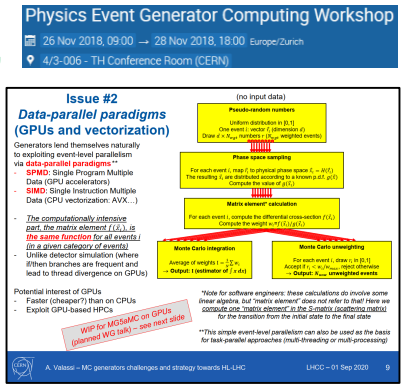
MG5_aMC>install cudacpp

2018 2019 2020 2021 2022 2023 2024 2025?

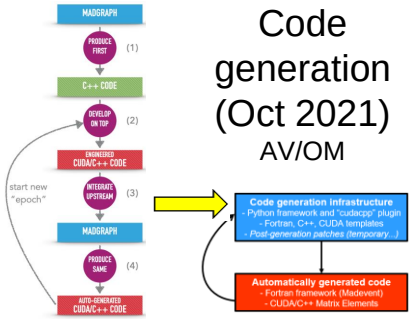
Inception: HSF Workshop (Nov 2018), HSF paper (2019-2020), LHCC review (2020-2021)



Challenges in Monte Carlo Event Generator Software for High-Luminosity LHC



Code generation (Oct 2021) AV/OM



Test x-sections and LHE files Fortran vs Cudacpp (2022-2024) AV

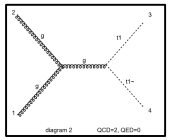
```
*** (2-512z) Compare MADEVENT_CPP x1 xsec to MADEVENT_FORTRAN xsec ***

OK! xsec from fortran (47.138611968034162) and cpp (47.138611968034169) differ by less than 3E-14 (2.220446049250313e-16)

*** (2-512z) Compare MADEVENT_CPP x1 events.lhe to MADEVENT_FORTRAN events.lhe reference (including colors and helicities) ***

OK! events.lhe.cpp.1 and events.lhe.ref.1 are identical
```

Memory buffers
Memory access
Kernel launchers
(Jan 2022) AV

Running couplings (Apr 2022) OM/SR/AV

```
DOUBLE PRECISION G, ALL_G(nb_page)
COMMON/STRONG/ G, ALL_G
```

BSM (2024) AV



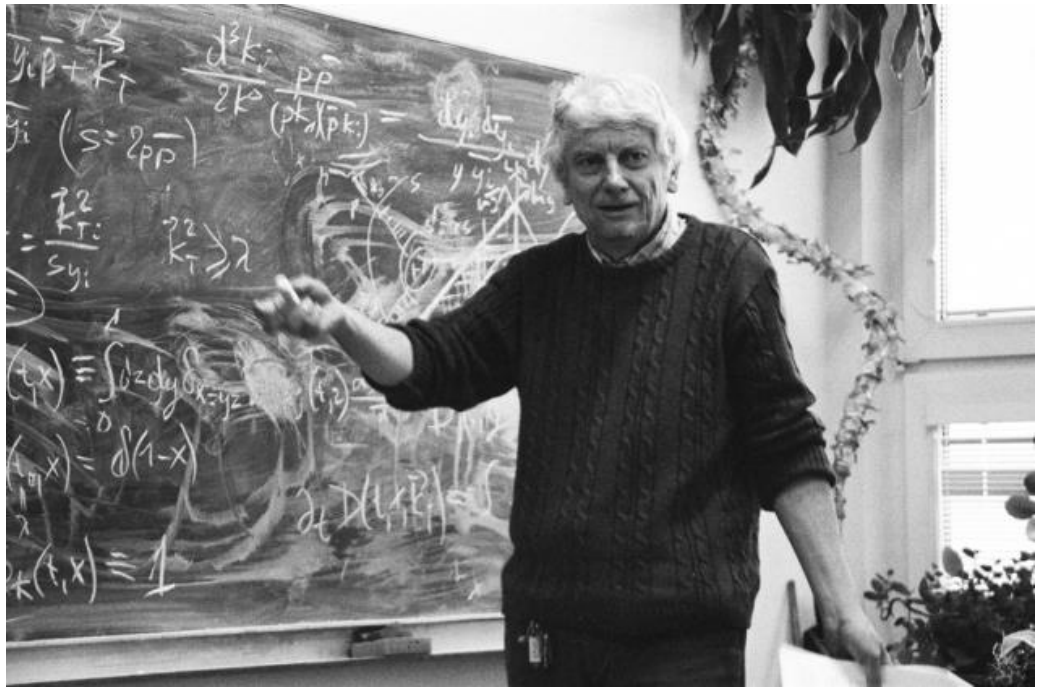
Conclusions

- ***The first release of the MG5AMC CUDACPP plugin for LO processes has been delivered!***
- We speed up the matrix element (ME) calculation via data parallelism with excellent lockstep processing
  - On NVidia and AMD GPUs in the order of x100 to x1000
  - On vector CPUs we achieve the theoretical speedup limit of x8 for AVX512 SIMD in double precision
- We achieved overall speedups between x3 and x70 (for DY+3j and $gg \to t\bar{t}ggg$) depending on the physics process
  - For many processes (like DY+3j) we speed up the ME so much that the bottleneck is the non-ME component
- Many opportunities for further speedups, especially in the non-ME components (phase space sampling, PDFs...)
  - WIP to profile the largest residual bottlenecks and to identify speedup strategies (with/without data parallelism)
- We need double precision (FP64) in the majority of floating-point operations
  - Using single precision (FP32) would allow a further speedup of MEs by a factor ~x2 but leads to numerical instabilities
  - *WARNING: in the latest GPUs targeting AI like NVidia Blackwell, what you pay for is mainly FP4/FP8/FP16 instead!*
- Mutually beneficial collaboration with CMS to test our software in production-like environments
  - See the details in the poster by Jin Choi on Thursday
- The techniques and software used in the LO release were designed with NLO in mind and most can be reused!
  - See the details in the next talk by Zenny Wettersten
  - More generally, two lessons learnt for any MC: use reentrant functions with well-defined inputs/outputs, use multi-event APIs

Thanks Krakow!

Thanks to the organizers of this conference

Thanks Staszek
and the whole KORALW/YFSWW team
for everything I learnt from you
about Monte Carlo's



BACKUP SLIDES

Benchmarking GPUs (CUDA cores or tensor cores? Which FP precision?)

On data-centre NVidia GPUs, in CUDA cores
the FP32 FLOPs are 2x the FP64 FLOPs

Architecture	Blackwell	Hopper	Ampere	Volta
Year	2024	2022	2020	2017
GPU Name	NVIDIA GB200	NVIDIA H100	NVIDIA A100	NVIDIA V100
FP64	90 teraFLOPS	34 teraFLOPS	9.7 teraFLOPS	7.8 teraFLOPs
FP32	180 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS	15.7 teraFLOPs
BF16	N/A	134 teraFLOPS	39 teraFLOPS	N/A
FP16	N/A	134 teraFLOPS	78 teraFLOPS	N/A
FP64 Tensor Core	90 teraFLOPS	67 teraFLOPS	19.5 teraFLOPS	N/A
TF32 Tensor Core	5 petaFLOPS	989 teraFLOPS	312 teraFLOPS	N/A
FP16 Tensor Core	10 petaFLOPS	1979 teraFLOPS	624 teraFLOPS	125 teraFLOPs
BF16 Tensor Core	10 petaFLOPS	1979 teraFLOPS	624 teraFLOPS	N/A
FP8 Tensor Core	20 petaFLOPS	3958 teraFLOPS	N/A	N/A
FP4 Tensor Core	40 petaFLOPS	N/A	N/A	N/A

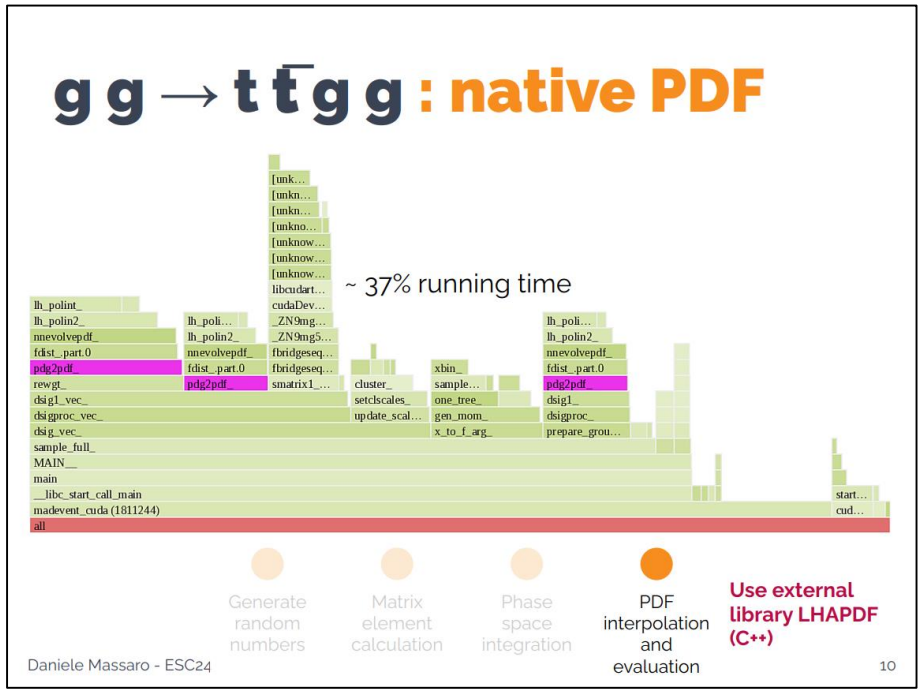
Blackwell GPUs do have many more FP64 FLOPs in CUDA cores than Hopper or Ampere or Volta...

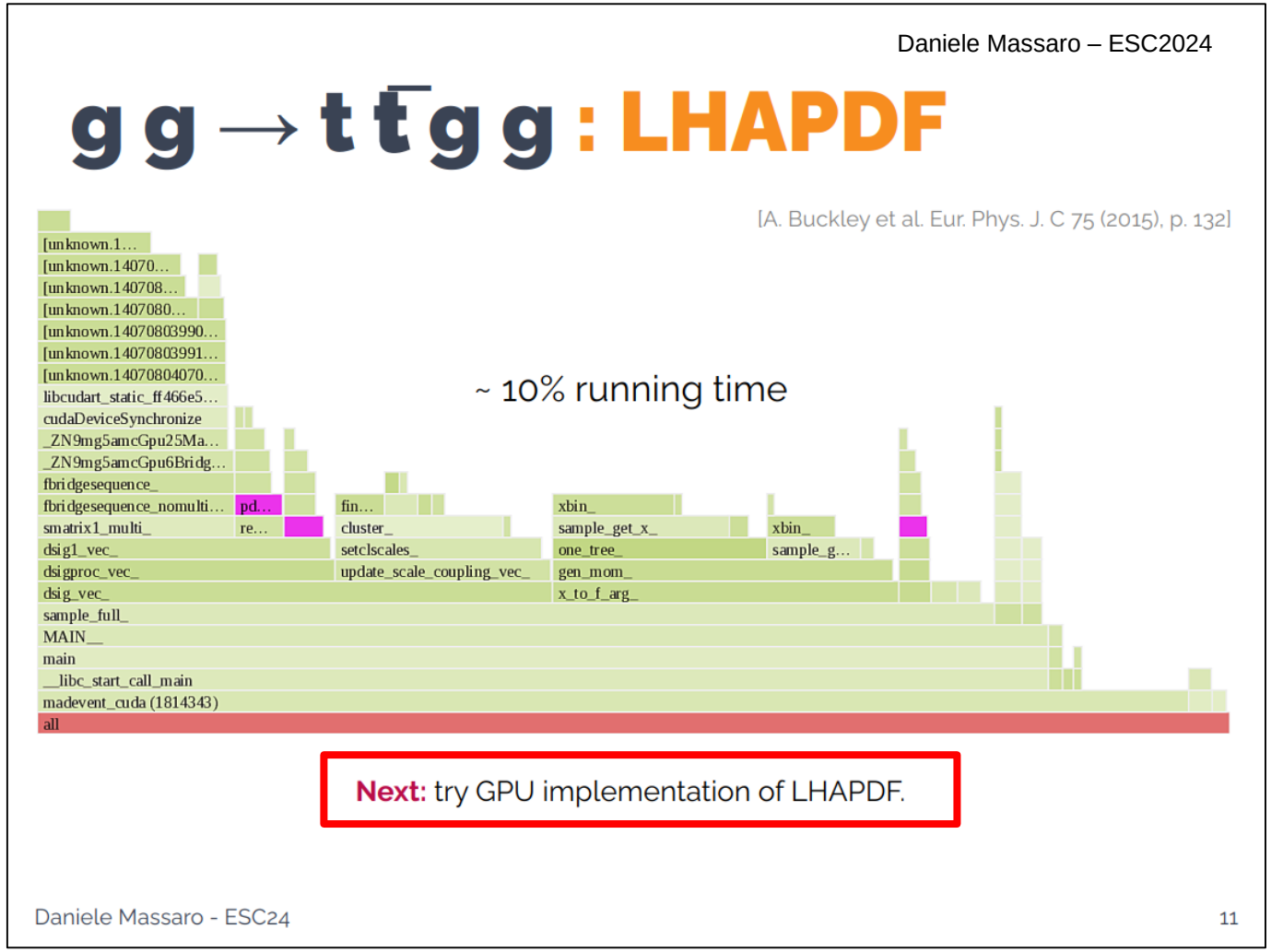
...but what you really pay for in Blackwell GPUs are the FP4/FP8/FP16 tensor core FLOPs for AI!

NB: CUDA cores and tensor cores are two different types of processors on the same chip! You must develop two different types of software!
(In MG5AMC we do not use tensor cores yet...)

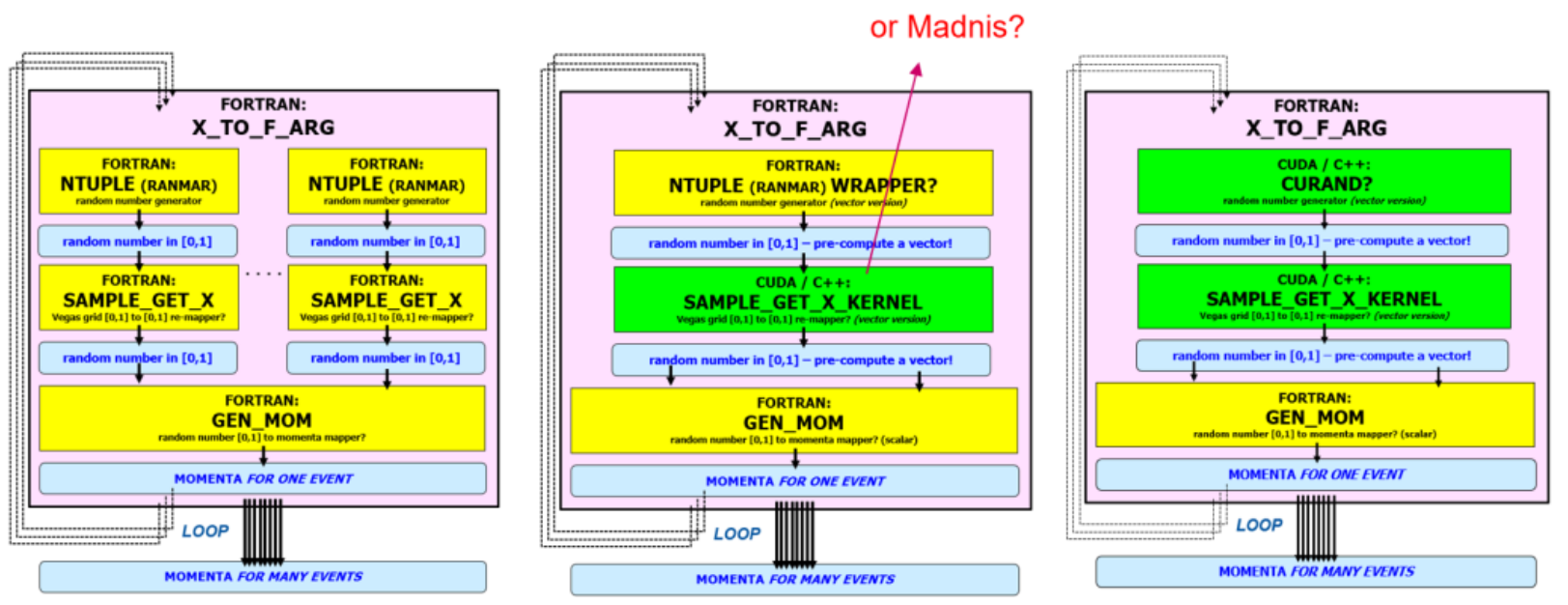
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf
https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf
https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper
https://resources.nvidia.com/en-us-blackwell-architecture

Speeding up PDF's





Vectorizing phase space sampling?



- I had a first quick look at possibly vectorizing sample_get_x
  - these are relatively short functions with simple operations, it is not rocket science
    - API: could start by preparing baskets and then looping internally as Olivier did for MEs
  - *the main problem I see is that there are many COMMON's making this stateful*
    - can the hidden inputs/outputs requiring these COMMON's be avoided?
    - or can these hidden inputs/outputs be moved outside the event/particle loop?

Low hanging fruits in sample_get_x

- Identified and fixed a couple of simple possible changes
  - the xbin() function called by sample_get_x is one of the bottlenecks: avoid it!
  - non-controversial(?) changes
    - (1) xbin is very often (not always) called with the same arguments, e.g. 0 or 1: cache it!
    - (2) xbin is sometimes called in dead or repeated code, avoid those calls
  - more controversial(?) changes
    - (3) expensive xbin calls take place in some internal checks to issue warnings: are these needed?
      - I have the impression this code is not completely functional anyway... (e.g. warning counters look strange)
  - some nice gains from (1) and especially (3)... will give details another time
- Are other improvements possible in the xbin function?
  - I had no time to look at this in more detail than caching it or avoiding it...
    - internals look reasonable, there is a binary tree search... but maybe can be improved?

Low hanging fruits?

#946 Status: WIP PR exists,
to be rediscussed with Olivier

```
For the cuda backend is now, skipping xbin checks #968
Phase space sampling in dy+3j has decreased from 78s to 53s (down by 30%)
> [GridPackCmd.launch] GRIDPCK TOTAL                  135.1144
> [madevent COUNTERS]  PROGRAM TOTAL                  130.8140s
> [madevent COUNTERS]  Fortran PhaseSpaceSampling      53.0338s for   44652395 events
> ...
> [madevent COUNTERS]  CudaCpp MEs                     35.4908s for    1769472 events
> [madevent COUNTERS]  OVERALL NON-MEs                 95.3232s
> [madevent COUNTERS]  OVERALL MEs                     35.4908s for    1769472 events

For the cuda backend was, including xbin checks but including trivial improvements #969
Phase space sampling in dy+3j has decreased from 93s to 78s (down by 15%)
< [GridPackCmd.launch] GRIDPCK TOTAL                  160.1718
< [madevent COUNTERS]  PROGRAM TOTAL                  155.8605s
< [madevent COUNTERS]  Fortran PhaseSpaceSampling      78.1023s for   44652395 events
< ...
< [madevent COUNTERS]  CudaCpp MEs                     35.4320s for    1769472 events
< [madevent COUNTERS]  OVERALL NON-MEs                120.4290s
< [madevent COUNTERS]  OVERALL MEs                     35.4320s for    1769472 events

For the cuda backend was in 2e59eca00, without trivial improvements
< [GridPackCmd.launch] GRIDPCK TOTAL                  176.8891
< [madevent COUNTERS]  PROGRAM TOTAL                  172.6370s
< [madevent COUNTERS]  Fortran Random2Momenta         93.2907s for   44651014 events
< ...
< [madevent COUNTERS]  CudaCpp MEs                     35.4557s for    1769472 events
< [madevent COUNTERS]  OVERALL NON-MEs                137.1806s
< [madevent COUNTERS]  OVERALL MEs                     35.4557s for    1769472 events
```

946b. (issue 968)
“More controversial” changes?
Save an additional 30%

946a. (issue 969)
“Non-controversial” changes?
Save 15%

https://github.com/valassi/madgraph4gpu/commit/348664c66d90f47d1d9e6fd72d7dd7f4b0fa7cff

Issue #2
Data-parallel paradigms
(GPUs and vectorization)

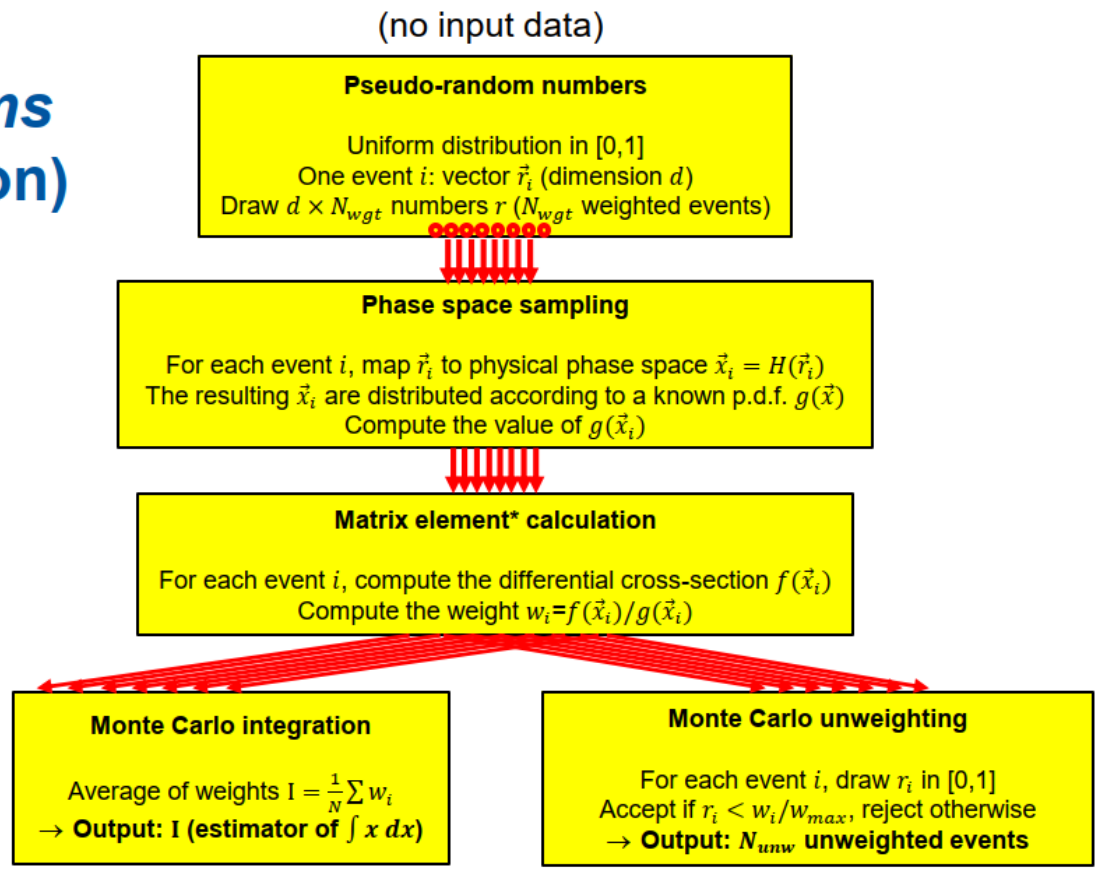
Generators lend themselves naturally to exploiting event-level parallelism via **data-parallel paradigms****

- **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)*

Scientific Computing and Software Collaborations

(or: working ~~on the bridge~~ between different units and communities)

in the cracks

- **A big lesson learnt from porting MG5AMC to GPUs: you need collaborations with a mix of skills!**

- Developing Monte Carlo generator software: which kind of job is this? *In which box should it be?*
  - A scientist's job? A theorist's job? An experimentalist's job? (A computing engineer's job?)
  - Do we need dedicated Scientific Computing units in our labs and universities?
  - Do we need to have dedicated career paths similar to Research Software Engineers?

- **The challenge: attracting, training, retaining people with the right competencies and interests**
  - Can we attract and motivate *young theorists* to work on software and computing optimizations?
    - A theorist colleague I was recently talking to: "We had an opening for working on software optimizations for our Monte Carlo generator. The only suitable candidates were two theorists. But they were concerned that *working on software optimizations would harm their future careers as theorists* and refused the job. In the end, we did not hire anyone."
  - Can we attract and motivate *young software engineers* to work with us instead of tech or finance companies?

I am only reporting a problem here... I do not have a magic-wand solution ☹

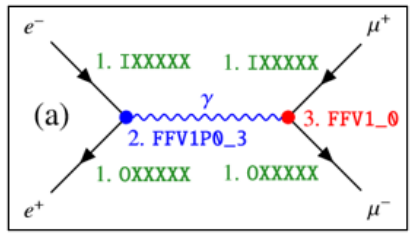
Helicity amplitudes – same code in CUDA and in vectorized C++

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

- *CUDA*: scalar complex →
- *C++, no SIMD*: scalar complex →
- *C++, with SIMD*: vector complex →

```
typedef thrust::complex<fptype> cxtype; // two doubles: RI
typedef std::complex<fptype> cxtype; // two doubles: RI
class cxtype_v { fptype_v m_real, m_imag; // RRRRIIII (SOA)
```

```
__device__
void FFV1_0( const cxtype_sv F1[],   // input: wavefunction1[6]
             const cxtype_sv F2[],   // input: wavefunction2[6]
             const cxtype_sv V3[],   // input: wavefunction3[6]
             const cxtype COUP,
             cxtype_sv* vertex )     // output: amplitude
{
  mgDebug( 0, __FUNCTION__ );
  const cxtype cI( 0., 1. );
  const cxtype_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
cxtype_v operator+( const cxtype_v& a, const cxtype_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;
```

C++ vectorization in CUDACPP: overview

- Implementation is based on **compiler vector extensions (CVEs):** explicit vectors of floating point types
  - Supported by all of the gcc, clang and (through clang) Intel icpx compilers
  - Powerful but easy to use (no debugging auto-vectorization!), intuitive (they force you to design code for vector types!)

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
```

- Routinely build and compare **five vectorization levels** on Intel CPUs (and similar features on AMD or ARM CPUs)

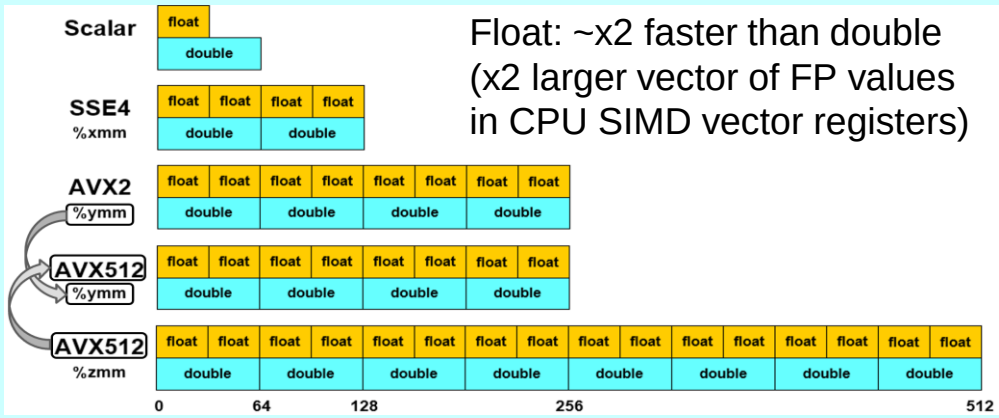
none **1xD, 1xF** (scalar)

sse4 **2xD, 4xF**(128-bit xmm registers, “nehalem” **SSE4.2** instruction set)

avx2 **4xD, 8xF** (256-bit ymm registers, “haswell” **AVX2** instruction set)

512y **4xD, 8xF** (256-bit ymm registers, “skylake-avx512**”** **AVX512** instruction set)

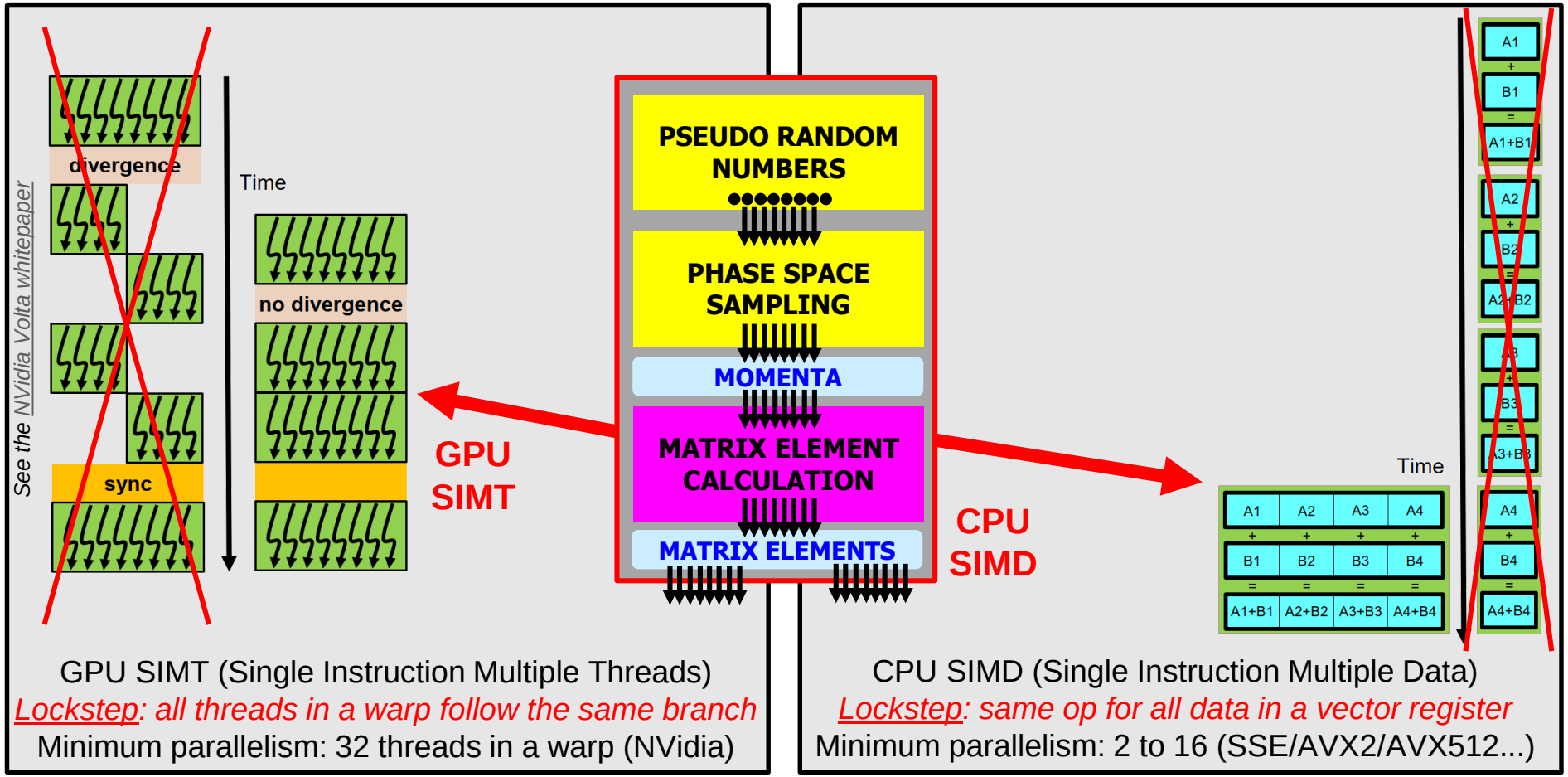
512z **8xD, 16xF** (512-bit zmm registers, “skylake-avx512” **AVX512** instruction set)



Float: ~x2 faster than double
(x2 larger vector of FP values
in CPU SIMD vector registers)

MG5aMC data parallelism: design for lockstep processing!

- In MC 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)

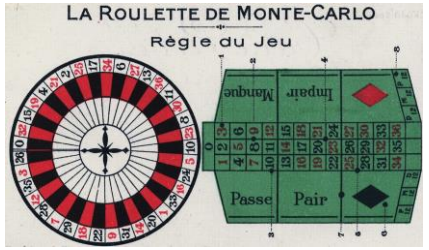


GPU SIMT (Single Instruction Multiple Threads)
Lockstep: all threads in a warp follow the same branch
Minimum parallelism: 32 threads in a warp (NVidia)

CPU SIMD (Single Instruction Multiple Data)
Lockstep: same op for all data in a vector register
Minimum parallelism: 2 to 16 (SSE/AVX2/AVX512...)

Lockstep? MC generators (*lucky!*) vs MC detector simulation (unlucky)

- *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)

SAME CALCULATION ON DIFFERENT DATA!

Lockstep processing
Good for SIMT/SIMD

MC DECISIONS* (*after* ME calculation):

Stochastic branching
Bad for SIMT/SIMD

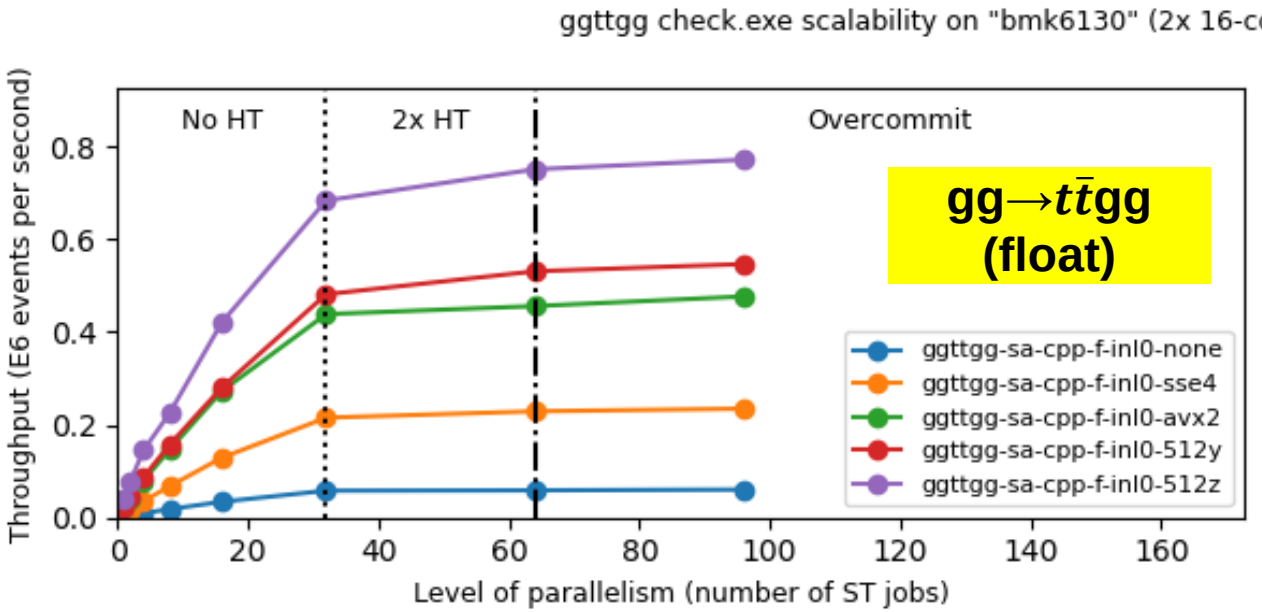
Detector simulation (Geant4)
- Particle/matter interaction (when? how?)
- Particle decays (when?)

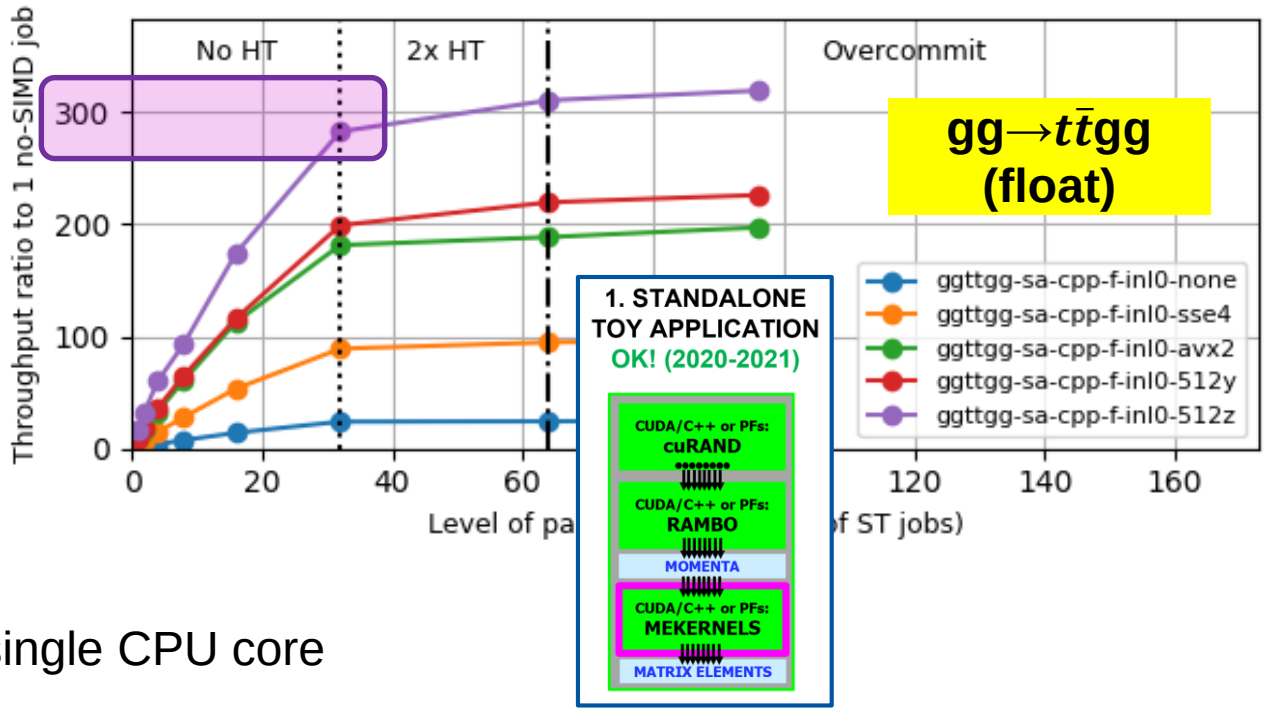
DIFFERENT CALCULATIONS ON DIFFERENT DATA!

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

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

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





- 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)

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

Reweighting and weight derivatives in parameter estimation

AV

- 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 $S_{sel}=S_{tot}$
- Ideal resolution, measured γ_i is that from MC truth

(implies ideal rejection of background events, $\gamma_i=0$)

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

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

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

⇨ **Weight Derivative Regression**

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

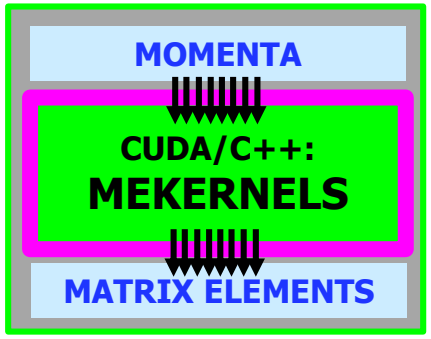
⇨ Data observable event properties $x_i^{(\mathrm{DATA})}$

Fit WDR regressor

$\mathbf{q_i^{(DATA)} = q(\, x_i^{(DATA)}\,)}$

$\mathbf{q_i^{(MC)} = q(\, x_i^{(MC)}\,)}$

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)

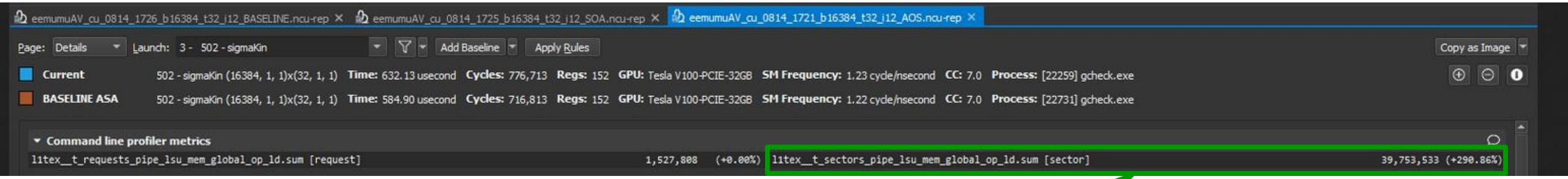
Example: Npar=6 particles for the 2→4 process $gg \to t\bar{t}gg$

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 $e^+e^- \to \mu^+\mu^-$ 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 $gg \to t\bar{t}ggg$: 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 JCJ* 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 $gg \to t\bar{t}ggg$: 120 color ordering permutations, 120x120 matrix
This takes **~60% of the CPU time** for this process

3. sum over helicities [Example for $gg \to t\bar{t}ggg$: 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?

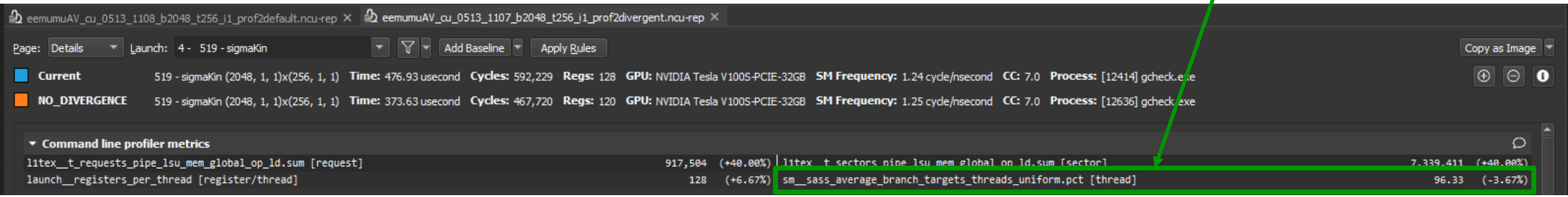
```
typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype))));
```

- 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 (RRRRIIII 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)



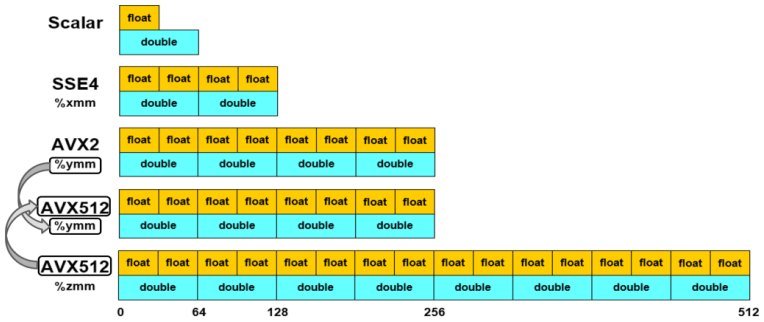
- CPU: the best lockstep metric IMO is the speedup over a no-SIMD case (reach theoretical maximum!)
  - but is also useful to disassemble the object using objdump and categorize SIMD intrinsics symbols...

4a90ec2 gg→ttgg

# Symbols in .o Build type	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
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

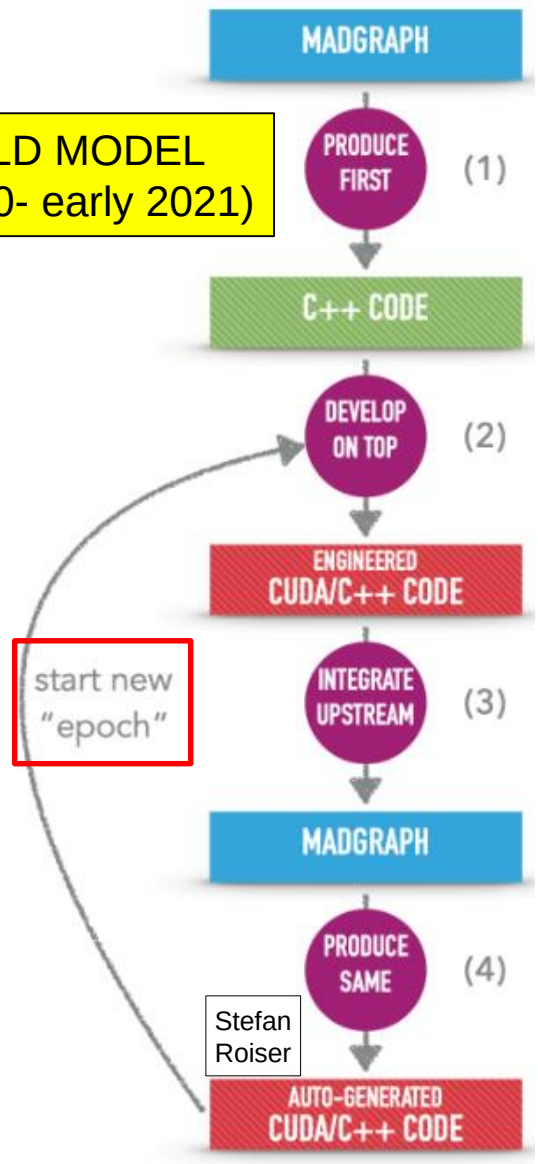
ACAT2022

$gg \to t\bar{t}gg$	MEs precision	madevent 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: from many “epochs” to a single evolving “epoch” ... and beyond

OLD MODEL (2020- early 2021)



(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

Code generation infrastructure
- Python framework and “cudacpp” plugin
- Fortran, C++, CUDA templates
- *Post-generation patches (temporary...)*

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

(3) re-generate

A.V.

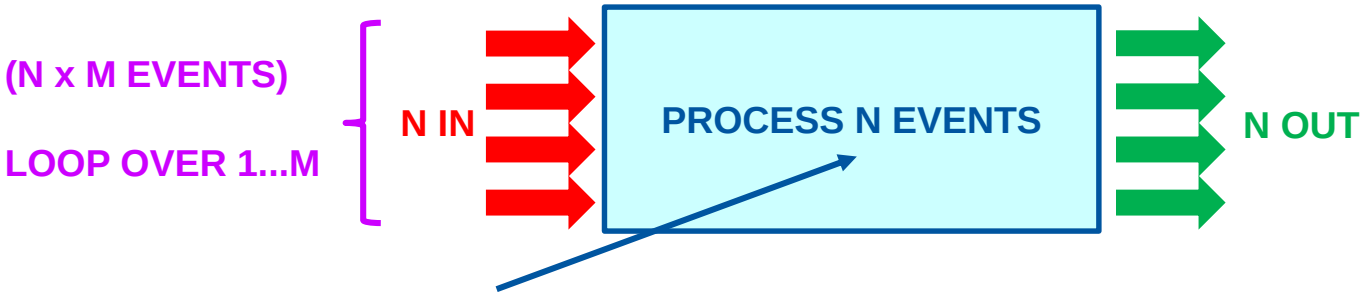
Automatically generated code
- Fortran framework (Madevent)
- CUDA/C++ Matrix Elements

NEW MODEL (since end 2021)

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

What about loops? And how many are N events?

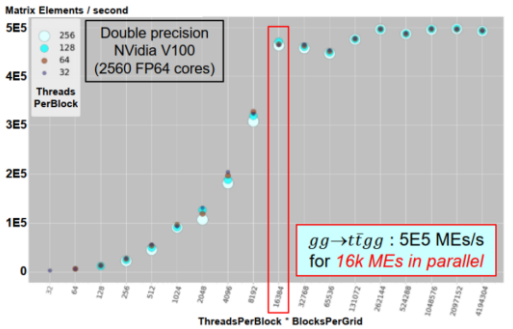
- You will still need to loop over multiple sets of N events
  - And the internal implementation of N-event processing may still involve some loops!



"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)

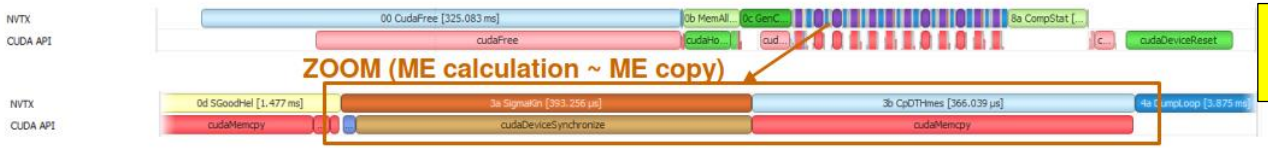


- NB: I focus on event-level parallelism, but other options exist
  - In MG5AMC we will investigate using 1 GPU thread per helicity per event...

Why focus on complex processes? Compute >> memory!

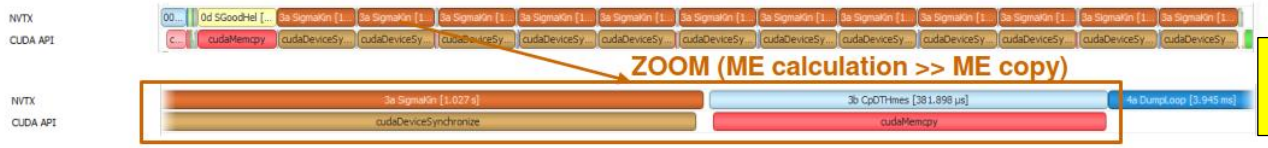
CUDA: Host(CPU)-to/from-Device(GPU) data copy has a cost

- 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. $e^+e^-\to\mu^+\mu^-$: 0.4ms ME calculation ~ 0.4ms ME copy)
    - Note: our ME throughput numbers are (number of MEs) / (time for ME calculation + ME copy)



$e^+e^-\to\mu^+\mu^-$

- But the time *cost of data transfers is negligible in complex processes*
  - ME calculation on GPU is slow (e.g. $gg\to t\bar{t}gg$: 1000ms ME calculation >> 0.4ms ME copy)
  - We expect that *this will not be an issue for typical LHC collision processes*

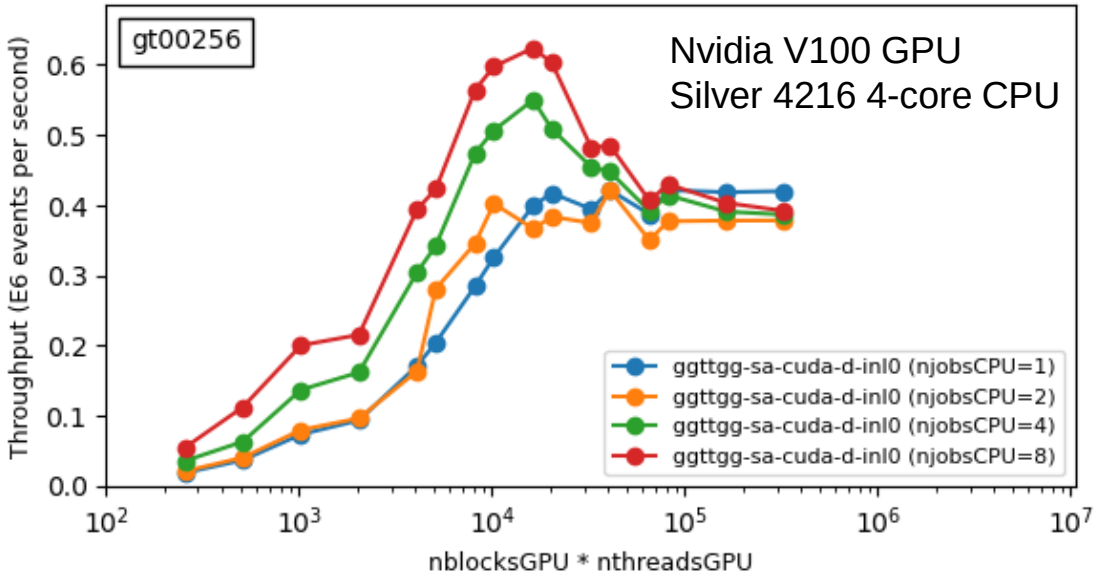


$gg\to t\bar{t}gg$

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs vCHEP – 19 May 2021 15

- 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 $gg\to t\bar{t}gg$ or $gg\to t\bar{t}ggg$

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 $gg \to t\bar{t}gg$, you need at least ~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?*

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

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

- 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

CUDACPP MEs

- 95% common code + a few #ifdef's for CUDA vs C++
- Designed for NVidia GPUs (so far: will add HIP/AMD)
  - Full feature support, e.g. tensor cores, streams, graphs
- Designed upfront for SIMD speedups on vector CPUs
- WIP on CPU multithreading and heterogeneous modes

PF MEs

- Write code once for many CPU/GPU vendors
- Support NVidia, AMD and Intel GPUs out-of-the-box
  - Limited support for vendor-specific features
- SIMD added via SYCL in Jan 2023, analysing results
- CPU multithreading out of the box

For the moment: we plan to continue development in parallel using both approaches – comparisons are very useful!
Two goals: not only production releases, but also *aim to provide useful feedback to HEP about usability of PFs*

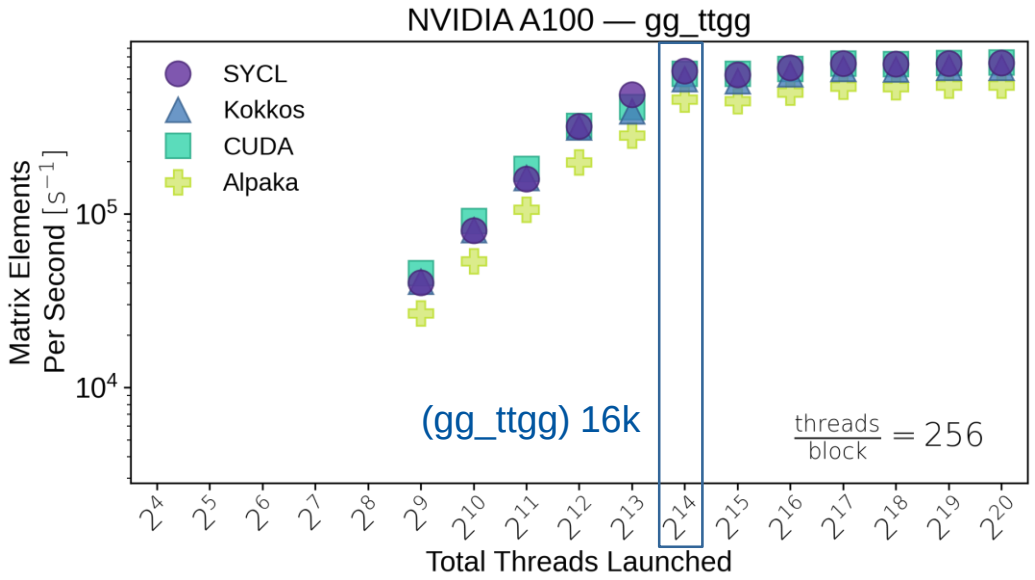
CUDACPP vs. Portability Frameworks – recap

- CUDAPP (our initial implementation) is where we add new features first
- The SYCL implementation of MG5aMC is now almost at the same level, the KOKKOS one somewhat behind
- The ALPAKA implementation of MG5aMC is no longer maintained

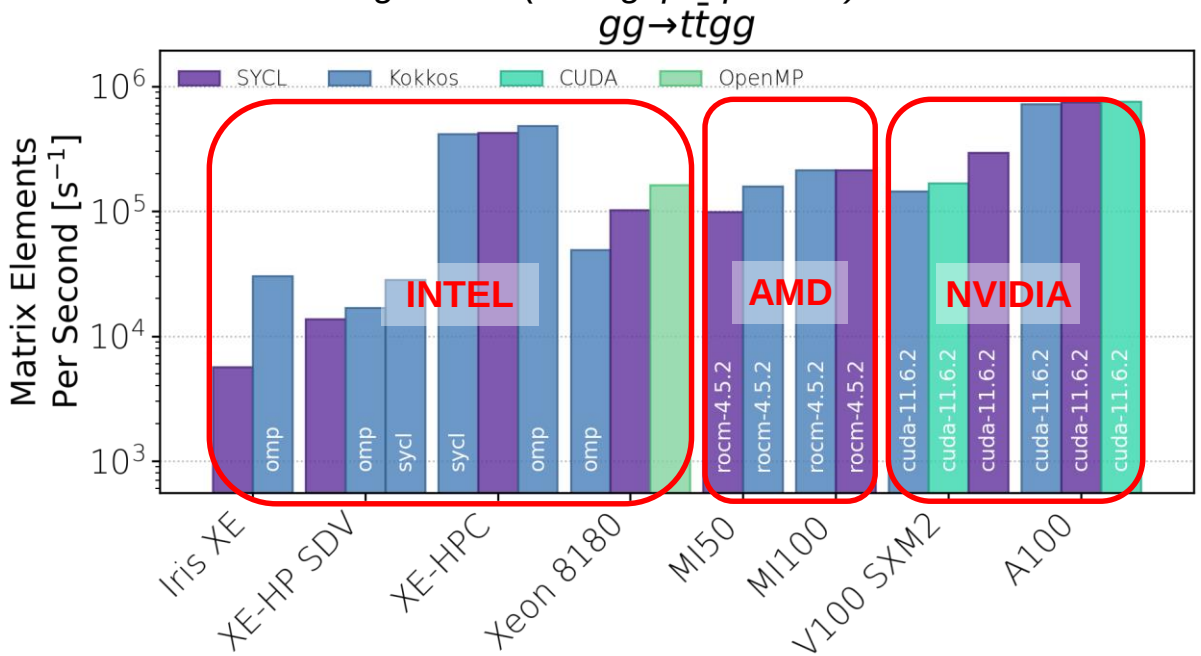
Backend	ME code generation	Standalone application	Actively maintained	MadEvent application	Latest dev code base
CUDACPP	✓	✓	✓	✓	✓
SYCL	✓	✓	✓	✓	~ ✓
KOKKOS	✓	✓	~ ✓	WIP	WIP
ALPAKA (CUPLA)	✓	✓	✘	✘	✘

CUDACPP vs PFs - GPU ME throughputs (standalone application)

Variable GPU-grid size (throughput scan)



Fixed GPU-grid size (throughput plateau)

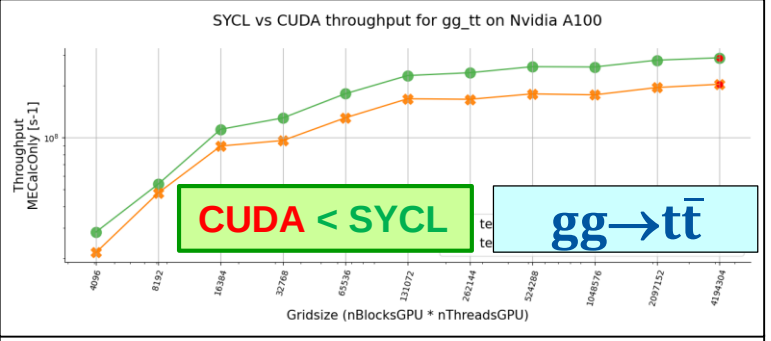


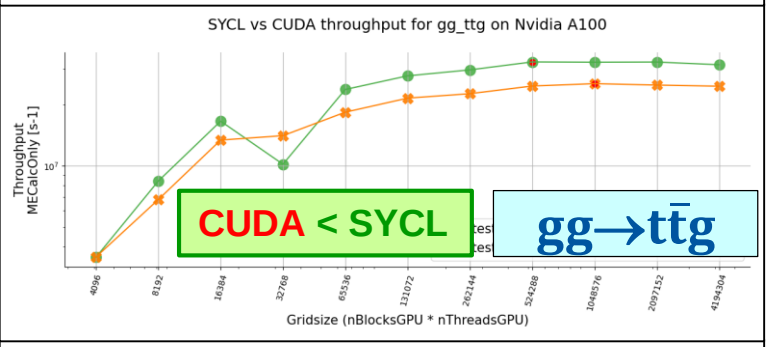
- The performances of the SYCL and Kokkos implementations of MG5aMC seem comparable to direct CUDA
  - Further comparisons are in progress, performance scales differently with more jets for different backends (next slide)
- SYCL and Kokkos run out of the box also on AMD and Intel GPUs
  - They also run out of the box on CPUs (performance under investigation)

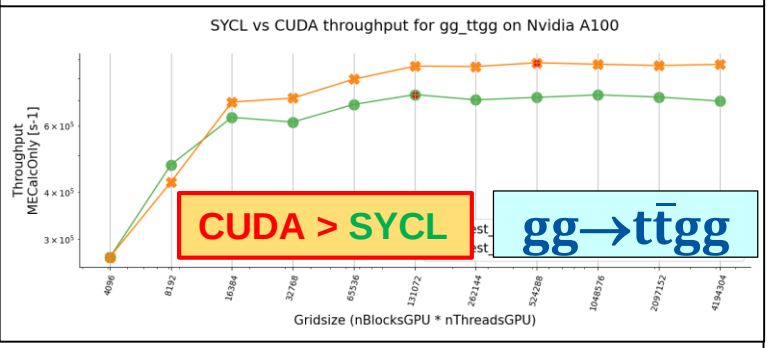
Xe-HP is a software development vehicle for functional testing only - currently used at Argonne and other customer sites to prepare their code for future Intel data centre GPUs
XE-HPC is an early implementation of the Aurora GPU

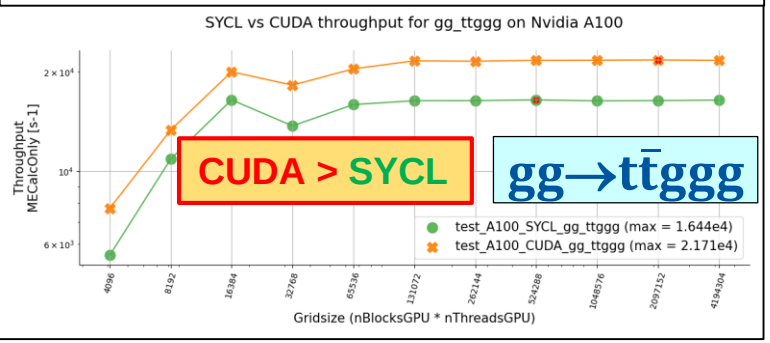
CUDA vs SYCL on NVidia A100

PRELIMINARY!
N. Nichols, T. Childers (SYCL)
J. Teig (tests/plots)









- SYCL and CUDA implementations have ~similar performances but
  - SYCL seems better for less complex processes
  - *CUDA seems better for more complex processes*

- These are very recent results, which are still being digested (WIP!)
  - It will be very interesting to understand more in detail what goes on

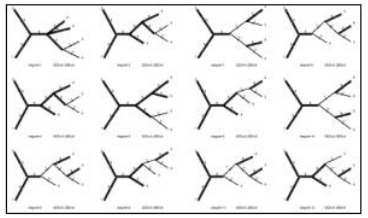
We plan to also compare more systematically the CUDACPP and SYCL performances on CPUs (vectorization, multi-core), but it will take time and optimization tweaks... WIP!

MORE BACKUP SLIDES

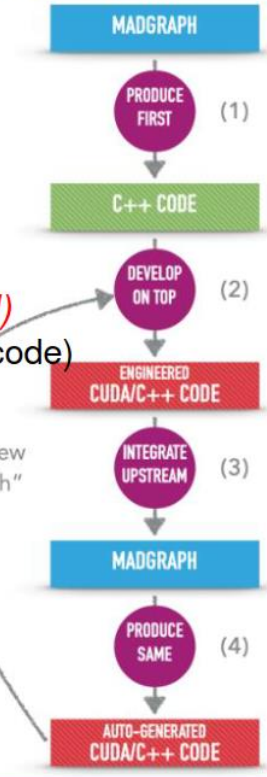
Code generation: how did we bootstrap the project?

Code is auto-generated ⇒ Iterative development process

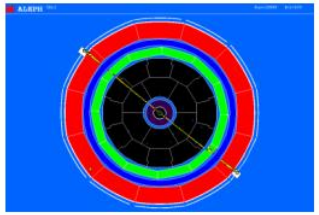
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Currently Fortran (default), C++, or Python
  - The more particles in the collision, the more Feynman diagrams and the more lines of code

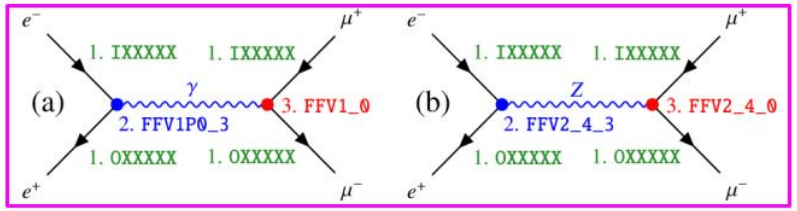


Process	*LOC*	*functions*	*function calls*
$e^+e^- \to \mu^+\mu^-$	776	8	16
$gg \to t\bar{t}$	839	10	22
$gg \to t\bar{t}g$	1082	36	106
$gg \to t\bar{t}gg$	1985	222	786

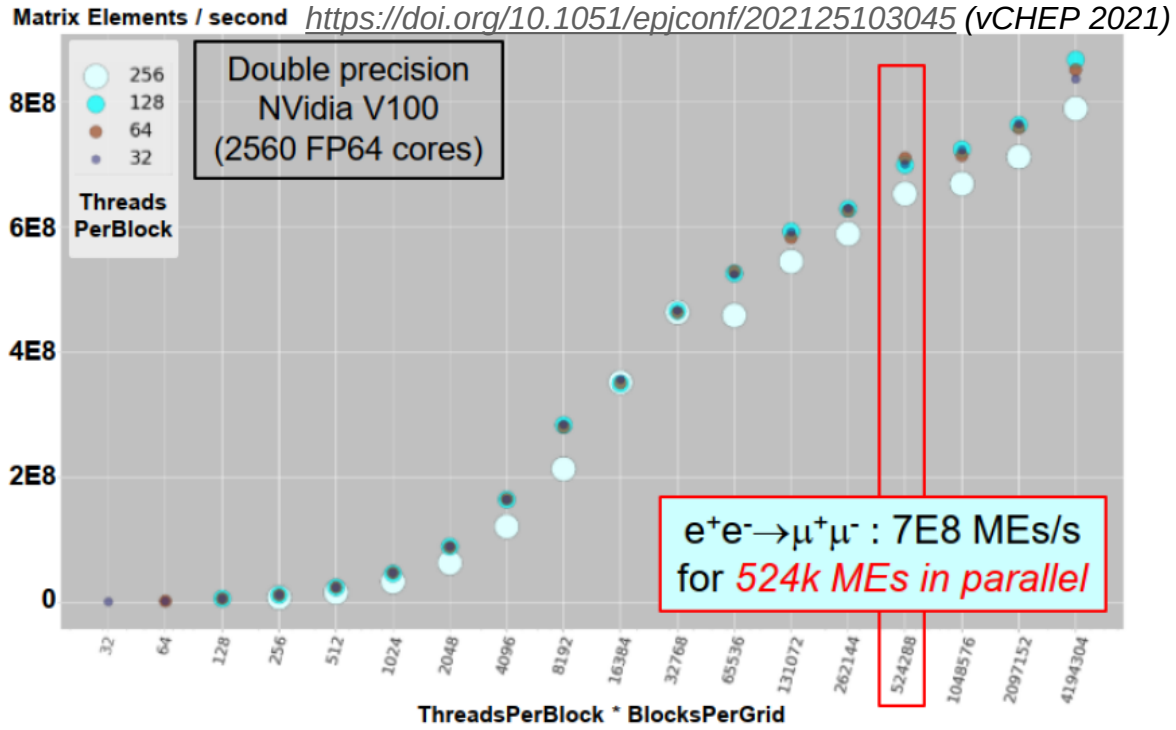


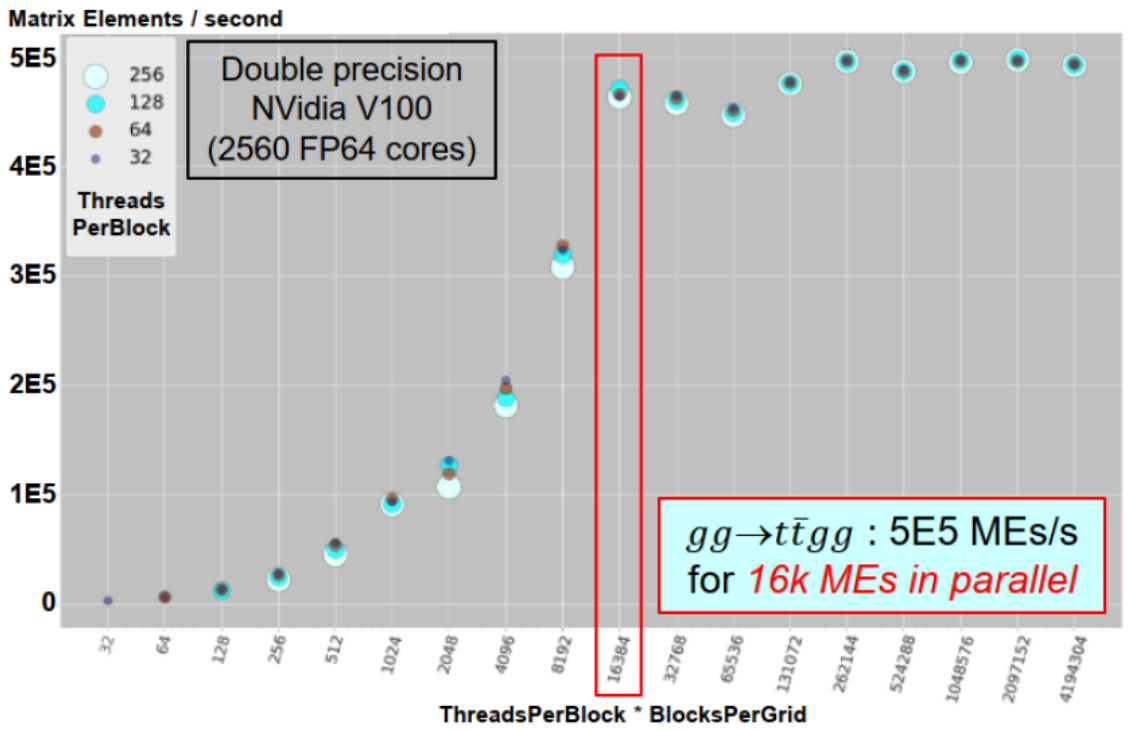
- *Goal: modify code-generating code (add CUDA, improve C++ backend)*
  - (1) Start simple: *bootstrap with* $e^+e^- \to \mu^+\mu^-$ (two diagrams, few lines of C++ code)
  - (2,3) Add CUDA and improve C++, port upstream to Python meta-code
  - (4) *Generate more complex LHC processes* $gg \to t\bar{t}$,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





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





- 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)?

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

THE CADNA LIBRARY

- 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/

- $P(x,y)= 9x^4 - y^4 + 2y^2$

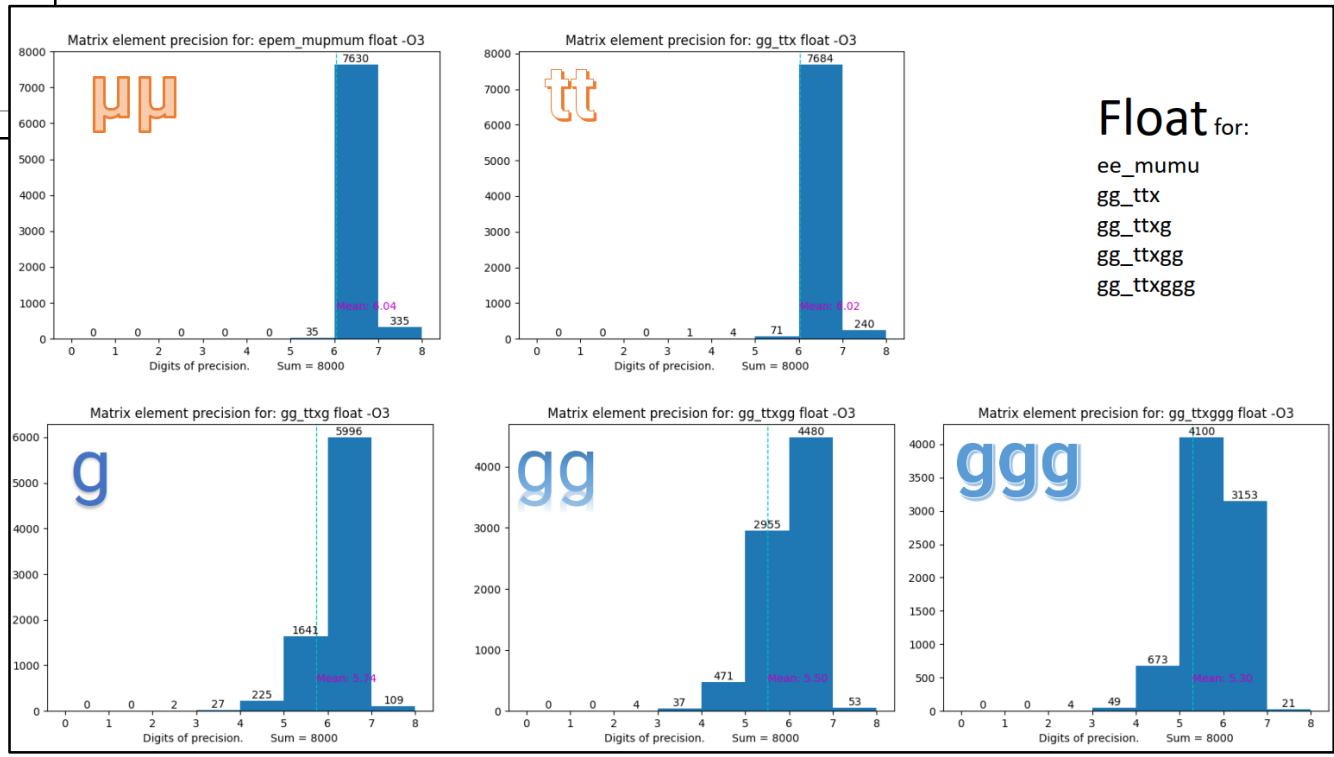
```
Without CADNA:

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)
```

Cadna

STEPHAN HAGEBOECK - MG5_AMC MEETING 2023

S. Hageboeck, Gargnano meeting 18 Sep 2023

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



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

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)

MG5AMC is not alone – SHERPA on GPU (BlockGen)

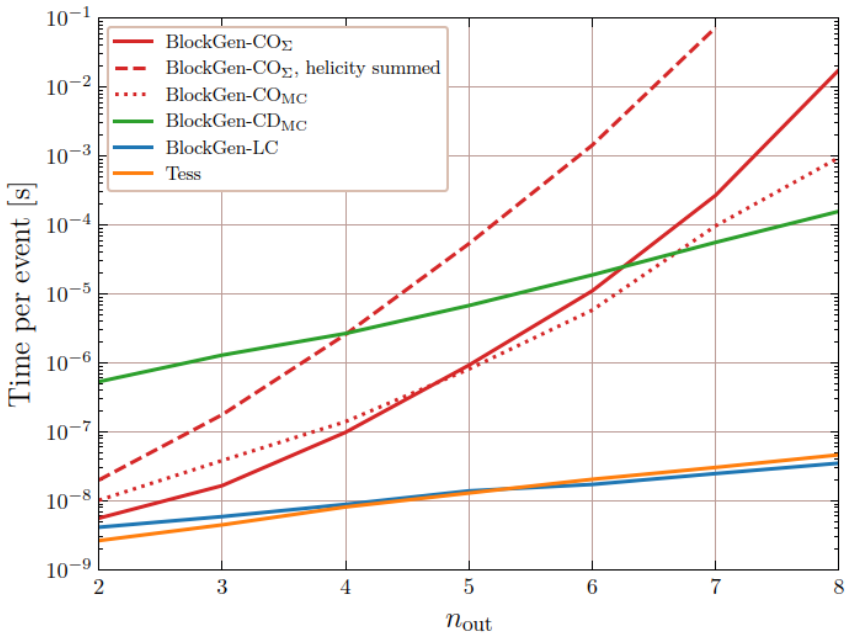


Figure 7: The timings for various GPU-based algorithms are compared as a function of gluon multiplicity. All algorithms were run on an NVIDIA V100 (16 GB global memory, 5,120 CUDA cores, 6144 KB L2 cache).

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

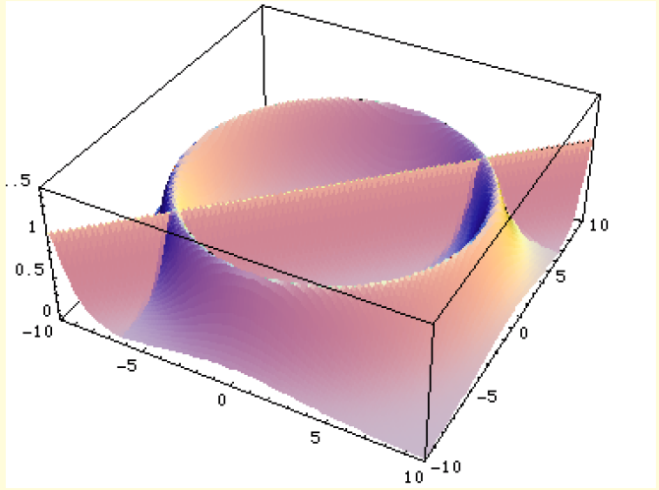
More recent results were presented in June 2023 in Les Houches by Max Knobbe

- 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)

ÉCOLE DE PHYSIQUE DES HOUCHES

EVEN MORE BACKUP SLIDES

Multi-channel



What do we do if there is no transformation that aligns all integrand peaks to the chosen axes?
Vegas is bound to fail!

Solution: use different transformations = channels

$$p(x) = \sum_{i=1}^{n} \alpha_i p_i(x) \qquad \text{with} \qquad \sum_{i=1}^{n} \alpha_i = 1$$

with each $p_i(x)$ taking care of one "peak" at the time

$$I = \int dx\, f(x) = \sum_{i=1}^{m} \alpha_i \int \frac{f(x)}{p(x)} dP_i(x),$$

OM, mgongpu dev meeting 22 Jun 2020

Additional information tracking

- Leading-Color information
  - ➡ Needed for starting point of the shower
  - ➡ Can be memory heavy
  - ➡ Really needed
  - ➡ Not easy to include as post-processing
- Helicity information
  - ➡ Not really needed (not provided at NLO)
    - CMS request it (use post-processing for NLO)
  - ➡ Easy for post-processing

OM, mgongpu dev meeting 22 Jun 2020

3. Github Actions

- Actions file in `.github/workflows/c-cpp.yml`
- If you have C++-only job, can hook yourself into that
- WIP: CUDA job on CERN premises
  - Already ran CUDA tests in container
- Development done, waiting for review: PR#52
- Will apply same strategy to epoch2/cuda

```yaml
name: C/C++ CI
on:
 pull_request:
   branches: [ master ]
jobs:
 epoch1_eemumu:
   runs-on: ubuntu-latest
   defaults:
     run:
       working-directory:
       epoch1/cuda/ee_mumu/SubProcesses/P1_Sigma_sm_epem_mupmum
   steps:
   - uses: actions/checkout@v2
   - name: make gtest
     working-directory: tools
     run: make
   - name: make
     run: make
   - name: make check
     run: make check
```

SH, mgongpu dev meeting 30 Nov 2020 8

Simultaneous development for CUDA and C++

OLD
Aug 2020
(master)

- **Design principle: same source code for CUDA and C++**
  - Why? Large overlap from the start (CUDA derived from standalone C++)
  - Why? Avoid divergence and the need to duplicate effort
  - Why? Because some optimizations (e.g. AOSOA) may be good for both!
  - How? Use #ifdef's – more than 90% of the code remains common
- **Design principle: decouple floating-point data types from calculations**
  - Why? Allow switching from double to float (in both CUDA and C++)
  - How? Typedef, default being *"typedef double fptype;"*
- **Design principle: decouple data access from calculations**
  - Why? Allow using formally the same code in CUDA and C++
    - Note: C++ is not vectorized yet here, data types are the same (scalar) in both
  - Why? In the past, also allowed SOA/AOS and global/shared/local comparison
  - How? Use references and constant references in compute engines
  - How? Move data access to separate routines, build references from them

    const fptype& pIparIp4Ievt(const fptype momenta1d, const int ipar, const int ip4, const int ievt)*

AOSOA memory layouts for CUDA and C++

OLD
Aug 2020
(master)

- **Design principle: move from variable-size to fixed-size arrays**
  - Why? The cardinalities of all relevant data sets (#particles, #momenta...) are fixed!
  - How: replace all *std::vector<std::vector<xxx>>* by C-style arrays *xxx[][]*
    - This alone gave a nice performance boost to both CUDA and C++ (IIRC...)

- **Vague design intuition: achieve SOA or AOSOA memory layouts**
  - Why? It may help for coalesced memory access in CUDA *(eventually it did)*
  - Why? It may help for SIMD vectorization in C++ *(eventually it did)*
  - How? C-style arrays: e.g. SOA, "double momenta[4=xyzE][#particles][#events]"
    - Note: this can also be cast from/to "double momenta[]" i.e. "double* momenta"
  - How? Many iterations, each with nevt events (=ndim=#threads*#blocks on GPU)
    - All arrays are dimensioned to handle the nevt events in one iteration
  - How? AOSOA eventually favored over SOA, split nevt into "pages"
    - Hence nevt=npag*nepp (#pages * #events per page), yielding AOSOA[npag][...][nepp]
    - More manageable than SOA[...][npag*nepp], better performance in CUDA
    - Note: can always recover AOS[nevt][...][1] by setting npag=nevt and nepp=1

SIMD vectorization (1) – event loops

NEW
Dec 2020
(klas branch)

- **Design principle: make the event loop the innermost loop**
  - **Why? Event-level data parallelism: the same calculations apply to all events**
  - How? Some refactoring, e.g. move **helicity loops** outside the **event loop**
    - Note: involves rethinking the interfaces of C++ methods ~analogous to CUDA kernels
      - Hence the name of the branch "klas": kernel launchers and SIMD vectorization
  - Note: our CUDA implementation is also based on event-level data parallelism
    - But in CUDA we ***already*** achieved event-level parallelism even without these changes
    - GPU SIMT/SPMD parallelism is simpler to achieve than CPU SIMD vectorization

GPU SIMT/SPMD: yes
CPU SIMD: no

(master branch Aug-Dec 2020)

```
#ifndef __CUDACC__
for (int ievt = 0; ievt < nevt; ++ievt) // CPU
#endif
{
#ifdef __CUDACC__
// CUDA - using precomputed good helicities
  for ( int ighel = 0; ighel < cNGoodHel[0]; ighel++ ) { ...
    const int ihel = cGoodHel[ighel];
    // GPU: const int ievt = blockDim.x * blockIdx.x + threadIdx.x;
    calculate_wavefunctions( ihel, allmomenta, meHelSum[0] ); ... } ...
#else
  // C++ - compute good helicities within this loop.
  for ( int ihel = 0; ihel < ncomb; ihel++ ) { ...
    calculate_wavefunctions( ihel, allmomenta, meHelSum[0], ievt ); ... } ...
#endif
}
```

GPU SIMT/SPMD: yes
CPU SIMD: yes (eventually!)

(klas branch Nov-Dec 2020)

```
// Both CUDA and C++, using precomputed good helicities
for ( int ighel = 0; ighel < cNGoodHel; ighel++ )
{
  const int ihel = cGoodHel[ighel];
#ifdef __CUDACC__
  // GPU: const int ievt = blockDim.x * blockIdx.x + threadIdx.x;
  calculate_wavefunctions( ihel, allmomenta, allMEs );
#else
  // CPU: loop on ievt=1..nevt using SIMD
  calculate_wavefunctions( ihel, allmomenta, allMEs, nevt );
#endif
}
```

En passant: more consistent interfaces for CUDA and C++
Both code snippets are within a CUDA kernel for the GPU

SIMD vectorization (2) – interfaces and types

NEW
Dec 2020
(klas branch)

- **Implementation choice: compiler vector extensions (CVE)**
  - Why? No external dependency (VC, VecCore...), simpler than intrinsics
    - More predictable than auto-vectorization alone, but still very lightweight
  - Why? Available for gcc, but also (with some limitations) for clang/OpenCL and icc
    - My initial tests: may need intrinsics if we want to support clang, operator[] is not a ref
  - How? Floating point vector fptype_v (RRRR) is a CVE typedef of size neppV

    *"typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype))));" // RRRR*
  - How? Complex vector cxtype_v (RRRRIIII) is a class with two fptype_v vectors

    "class cxtype_v{ ...; private: fptype_v m_real, m_imag; }; // RRRRIIII
    - Add boilerplate implementation for missing "cxtype[_v] operator cxtype_v|fptype[_v]"
    - No ambition to be complete: only add what is needed (as we do for cxtype's in CUDA)
  - How? Refactor interfaces of C++ compute engines using vector types
    - The nevt=npagV*neppV events are split in npagV pages with neppV events per page
    - Explicit loop on npagV in the calling function, implicit loop on neppV in CVE vectors
    - *This is where most of the actual coding work was necessary*

SIMD vectorization (3) – formal language

NEW
Dec 2020
(klas branch)

- **Design principle: decouple data access *and data types* from calculations**
  - Why? Allow using formally the same code in CUDA and scalar or vectorized C++
  - How? Add another typedef which can be scalar or vector depending on context

(klas branch Nov-Dec 2020)

```
#ifdef __CUDACC__
typedef fptype fptype_sv;
typedef cxtype cxtype_sv;
#else
typedef fptype_v fptype_sv;
typedef cxtype_v cxtype_sv;
#endif
```

Same code
- vector CPU
- scalar CPU or GPU

(klas branch Nov-Dec 2020)

```
__device__
void FFV1_0( const cxtype_sv F1S[],   // input wavefunction1[6]
             const cxtype_sv F2S[],   // input wavefunction2[6]
             const cxtype_sv V3S[],   // input wavefunction3[6]
             const cxtype COUP,
             cxtype_sv vertex[] )     // output
{ ...
  const cxtype_sv& F1_2 = F1S[2]; ...
  const cxtype_sv& F2_2 = F2S[2]; ...
  const cxtype_sv& V3_2 = V3S[2]; ...
  const cxtype_sv TMP4 =
    ( 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 ) * TMP4; ...
  return;
}
```

SIMD vectorization (4) – architectures

NEW
Dec 2020
(klas branch)

- Tested various combinations of compiler flags and architectures
  - **AVX2 gives almost a factor 4 speedup from SIMD vectorization!**
    - Close to maximum theoretical limit (vectors of 4 doubles)
  - AVX512 is not faster (or is even a bit slower) than AVX2
    - Not too surprising, there are known issues – may investigate more later if needed

- **Support choice: focus on AVX2, support scalars, AVX512 and SSE**
  - Reminder: scalar=1, SSE=2, AVX2=4, AVX512=8 doubles per vector
  - Why? AVX2 gets the highest gain (4x), keep AVX512 anyway for further tests
  - Why? Most modern CPUs have AVX2, but some Grid nodes only have SSE
    - I had initially discarded SSE, will add support for it as agreed
  - How? Hardcode vector sizes depending on #ifdef''s for AVX2 and AVX512F

Tests on Cori – First success on AVX512/zmm

- See https://github.com/madgraph5/madgraph4gpu/pull/236
  - Results from Friday evening – not yet merged
- CORI at NERSC – Intel Xeon Gold 6148 CPUs plus V100 GPUs
  - Temporary access during a training about CUDA Streams (thanks ATLAS!)
- **First demonstrated benefit of AVX512/zmm over AVX512/ymm**
  - For doubles: around 10%
  - For floats: around 30%
  - To be compared to a theoretical factor 2 maximum speedup
- Things I have not tested
  - perf was not available
  - I forgot to compare with/without LTO/inlining (results above are no LTO, no inlining)

Towards production: two development lines?

- Fastest: “standalone” C++ driver
- Allows fastest prototyping
  - minimal overhead, ~pure ME
- Complete minimal eemumu anyway
  - unweighting, cross section
- Freedom to redesign from scratch
  - new sampling e.g. VegasFlow
  - GPU specific e.g. PDFFlow

- For production: madevent Fortran driver
- Keep the framework used by users
  - and all complexities (NLO, pdf...)
- Two basic options
  - 1. Rewrite full madevent Fortran in C++, keep the Python machinery
    - MadEvent input/output?
    - Complex, slow, unnecessary?
  - 2. Inject only C++/CUDA ME calculation into Fortran madevent
    - Seems feasible and ~fast
    - Small apple-to-apple replacement (and timing comparison)

Fortran: integration proposal details

- Analysis (so far) in https://github.com/madgraph5/madgraph4gpu/issues/121
- Two main problems:
  - 1. Fortran matrix1 routine is not completely stateless: must clean up inputs/outputs
  - 2. Fortran matrix1 operates on a single event: must rewrite the flow for many events
    - Absolutely needed both for GPUs and for SIMD on CPUs
- Can proceed in two steps
  - 1. Make Fortran stateless, inject single-event C (C++/CUDA), test cross-linkage with data
  - 2. Split the Fortran event loop that calls matrix1 (via dsig)

Split here:
- for many events, call x_to_f_arg (random + sampling), store the corresponding momenta
- loop over the stored momenta, call dsig in parallel for many events (must remove state also from dsig, not only from matrix1...)

Allocate memory in advance, before that

```
124   c
125   c     Main Integration Loop
126   c
127         iter = 1
128         do while(iter .le. itmax)
129   c
130   c     Get integration point
131   c
132            call sample_get_config(wgt,iter,ipole)
133            if (iter .le. itmax) then
134               ievent=ievent+1
135               call x_to_f_arg(ndim,ipole,mincfig,maxcfig,ninvar,wgt,x,p)
136               if (pass_point(p)) then
137                  fx = dsig(p,wgt,0) !Evaluate function
138                  wgt = wgt*fx
139                  if (wgt .ne. 0d0) call graph_point(p,wgt) !Update graphs
140               else
141                  fx =0d0
142                  wgt=0d0
143               endif
144               call sample_put_point(wgt,x(1),iter,ipole) !Store result
145            endif
146            if (wgt .ne. 0d0) kevent=kevent+1
147   c
148   c     Write out progress/histograms
149   c
150            if (kevent .ge. nwrite) then
151               nwrite = nwrite+ncall*itmax/nsteps
152               nwrite = min(nwrite,ncall*itmax)
153               call graph_store
154            endif
155     99    enddo
```

Interfacing with Fortran (MadEvent): potential challenges

- Work just starting: inject our CUDA/C++ ME calculation into MadEvent
  - Fastest way to bring this R&D work to production – discussed with experiments in HSF WG
  - More than just MadEvent and Fortran: bash, Python, C++ and pdf, PS, NLO merging...
  - Easier validation – leverage on established infrastructure, no change in user interface

- From a first look at MadEvent: two potential challenges (legacy code reengineering)
  - *(1) must create event baskets a posteriori* (current code loops on individual events)
  - *(2) Fortran common blocks complicate separation of inputs and outputs* (not pure functions)

FOR ievent in 1,...N:
- draw random numbers
- map to phase space point
- apply kinematic cuts
- **compute ME in Fortran**
  (extra input from common?)
  (extra output to common?)

FOR ievent in 1,...N:
- draw random numbers
- map to phase space point
- apply kinematic cuts
- *store to event basket*

FOR ievent in 1,...N (SIMT, SIMD):
- *read from event basket*
- **compute ME in CUDA/C++**
  (no extra input from common)
  (no extra output to common)

Old design

dsample.f

call x_to_f_arg(…)
-> x, p, wgt (local)
-> xbk, q2fact, cm_rap (common)

fx = dsig(p,wgt,0)
-> lot is hidden here!
-> cuts
-> matrix-element choice
-> symmetry handling
-> scale choice

call sample_put_point()
-> add grid running information
-> decide to pass to next iteration
-> update the grid

auto_dsig.f

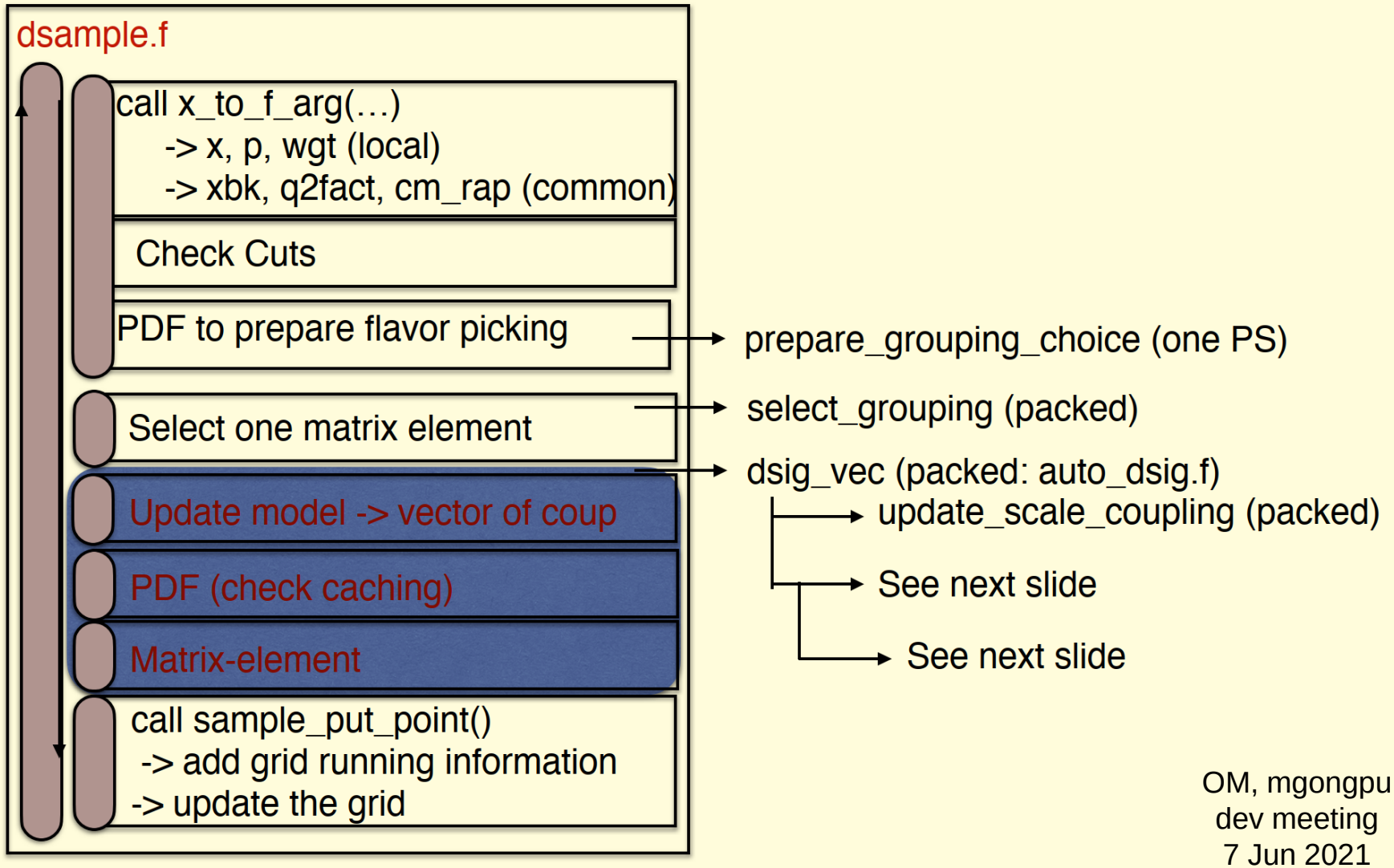
SELPROC(…)=DSIGPROC(…)
-> Compute PDF only for all flavor
- Pick a given matrix-element
- Pick a symmetry channel
-> run the cuts (inside dsigproc)

Reset clustering flag

DSIG=DSIGPROC()
-> re-run cuts
-> update model
-> compute matrix-element
-> compute the pdf
-> in principle cached

OM, mgongpu
dev meeting
7 Jun 2021

New design



OM, mgongpu
dev meeting
7 Jun 2021

Edit metadata __init__.py

```
File Edit Options Buffers Tools Python Help
## import the required files
# example: import maddm_interface as maddm_interface # local file
#          import madgraph.various.cluster as cluster #MG5 distribution file
# Three types of functionality are allowed in a plugin
#   1. new output mode
#   2. new cluster support
#   3. new interface

# 1. Define new output mode
#    example: new_output = {'myformat': MYCLASS}
#    madgraph will then allow the command "output myformat PATH"
#    MYCLASS should inherated of the class madgraph.iolibs.export_v4.VirtualExporter
import output
new_output = {'standalone_cuda':output.CUDAExporter}

# 2. Define new way to handle the cluster.
#    example new_cluster = {'mycluster': MYCLUSTERCLASS}
#    allow "set cluster_type mycluster" in madgraph
#    MYCLUSTERCLASS should inherated from madgraph.various.cluster.Cluster
new_cluster = {}


# 3. Define a new interface (allow to add/modify MG5 command)
#    This can be activated via ./bin/mg5_aMC --mode=PLUGINNAME
## Put None if no dedicated command are required
new_interface = None


########################## CONTROL VARIABLE ##################################
__author__ = ''
__email__ = ''
__version__ = (1,0,0)
minimal_mg5amcnlo_version = (2,3,4)
maximal_mg5amcnlo_version = (1000,1000,1000)
latest_validated_version = (2,4,0)
```

Key: name of output

value: class to use

OM, mgongpu
codegen workshop
16 Sep 2021

Stateful interface for Fortran bridge?

- See the discussion in https://github.com/madgraph5/madgraph4gpu/issues/329
- First point (easy): need the same Bridge instance across several ME computations
  - compute good helicities only once
  - more generally, provide a single constructor/initialization (e.g. also for Kokkos etc)
- Second point (we should agree): static Bridge singleton vs. allow several Bridges
  - OPTION 1 (a la SR): Bridge address invisible from Fortran (use static Bridge singleton)
    - No explicit Bridge constructor call from Fortran (construct C++ Bridge on first ME calculation)
    - Assumes Fortran will not use multi-threading (but C++ Bridge can internally be MT)
    - Fewer Fortran calls to C (but we must ensure Fortran/C interface match for other methods anyway)
    - No explicit destructor, relies on static going out of scope at the end of the program
  - OPTION 2 (a la AV): Bridge address visible from Fortran (several Bridges possible)
    - Explicit Bridge constructor and destructor from Fortran (memory address is Fortran INTEGER*8)
    - Two more Fortran calls to C (two more methods needing checks of Fortran/C interface match)
  - *What's your preference? (Olivier, especially what's your preference?)*
  - In both cases: I suggest adding a Fortran INTERFACE block to expose the C interface

Code generation backport – *completed* for cudacpp!

- Track overall progress on https://github.com/madgraph5/madgraph4gpu/issues/244
  - From epoch1, epoch2... to a single evolving epochX
  - Include code generation plugin, auto code, manual code, throughput logs
  - See https://github.com/madgraph5/madgraph4gpu/tree/master/epochX/cudacpp

- Various steps for Cuda/C++ plugin – NB: epochX is the current latest cuda/c++ code!
  - 1. Basic infrastructure **[DONE]** – PR #245
  - 2. Backport "epoch2 ggttgg" **[DONE]** – PR #247, PR #254, PR #256
    - Tagged as "golden_epochX2" – for Kokkos/Alpaka/Sycl comparison, Fortran/Cuda bridging...
  - 3. Backport "epoch1 eemumu" (vectorization) **[new: DONE]** – PR #253
  - 4. Generate vectorized ggttgg in epochX **[new: DONE]** – PR #267, PR #270
    - Tagged as "golden_epochX4" – for Kokkos/Alpaka/Sycl comparison, Fortran/Cuda bridging...
    - *Some preliminary results on the next slide*
  - 5+. Ongoing/planned: cleanup, develop, iterate... (but this is beyond the backport!)
    - Cleaned up several old tickets, tried to document important issues
    - Ongoing work on tests for ggtt/ggttgg with StephanH
    - Planned: documentation for epochX (code generation, performance/functional tests)
    - Available to help on Alpaka/Kokkos/... backport if needed
    - A few other pending developments, some in epoch1 (e.g. neppM vs neppR)

Class architecture – 3+ types of classes

- (1) Memory buffers
  - Allocate unstructured memory (e.g. double*) for a given number of events
  - Different classes for host/pinnedhost/device memory and for momenta, MEs etc.
- (2) Memory access
  - Define the internal substructure of memory buffers (e.g. AOSOA for momenta)
  - Define how CUDA/C++ kernels locate event records and fields in memory buffers
  - Different classes for host and device memory access and for momenta, MEs etc.
- (3) Kernel launchers
  - Constructors from references to existing input and output memory buffers
  - Computational functions with C++ interface, encapsulating C++ or CUDA kernels
    - Internally use low level Rambo/HelAmps functions for specific memory access template parameter
  - Different classes for host and device kernel processing and for Rambo, ME calculation etc.
- Applications (check_sa, runTest, Bridge) are built in terms of these 3 “LEGO bricks”
  - Eventually: sequence classes for some of these sequences?

Recap of missing pieces (in Bridge and implementation)

- My current Bridge: input momenta → output MEs, per event (arrays)
  - Overview (not updated) in https://github.com/madgraph5/madgraph4gpu/issues/404
- Multichannel: must add input diagram ID (scalar)
  - MadEvent single-diagram enhancement: output ME multiplied by amp(ID)/sum_amp
    - Done by Olivier in standalone_gpu, I must integrate it in madevent + cudacpp
  - NB can ignore "get_channel_cut' https://github.com/madgraph5/madgraph4gpu/issues/419
- Running of alphas: must add input alphas, per event (array)
  - https://github.com/madgraph5/madgraph4gpu/issues/373
  - The QCD running coupling constant alpha_strong is different in each event
  - What we chose (for now?): Fortran chooses the scale and passes it to cudacpp
    - Done by Stefan in standalone_cudacpp, I must integrate it in madevent + cudacpp
- Random color choice: must add input random and output color, per event (array)
  - https://github.com/madgraph5/madgraph4gpu/issues/402
  - To be implemented in cudacpp: random picking of color
- Random helicity choice: must add input random and output helicity, per event (array)
  - https://github.com/madgraph5/madgraph4gpu/issues/403
  - To be implemented in cudacpp: random picking of helicity
- For xsec: multichannel, alphas. Only for unweighted events: random color, helicity.

Matrix element integration in MadEvent: results

- Functional results (Madevent with Fortran MEs vs CUDA/C++ MEs, using the same random seeds)
  - Cross section calculation: done! *(Same cross section within ~E-14 relative accuracy)*
  - Unweighted event generation: almost done! *(Same LHE output files, except for missing color/helicity)*

- Performance results ⇒ Total time = Madevent time (scalar, sequential) + ME time (vector, parallel)
  - The overall speedup is limited by the incompressible scalar component (we need to reduce that too!)
  - Amdahl's law: if parallel fraction is initially p, maximum speedup is 1/(1-p)

AVX512 on Intel Silver: x4.4 speedup for MEs, x3.9 for full workflow
AVX512 on Intel Gold: x7.8 speedup for MEs, x6.4 for full workflow

CERN: Intel Silver 4216 + Nvidia V100

gg → ttggg		[seconds] Overall = MadEvent + MEs	[MEs/second]
6k events	FORTRAN	93.65 = 4.16 + 89.49	7.19e+01
	CPP/none	111.50 = 4.89 + 106.62	6.03e+01
	CPP/sse4	62.16 = 4.50 + 57.66	1.12e+02
	CPP/avx2	33.78 = 4.26 + 29.52	2.18e+02
	CPP/512y	30.66 = 4.22 + 26.44	2.43e+02
	CPP/512z	28.36 = 4.34 + 24.02	2.68e+02
	CUDA/32	63.72 = 5.34 + 58.38	1.10e+02
800k events	CUDA/8192	639.20 = 527.37 + 111.83	7.40e+03

Juwels: Intel Gold 6148

	[seconds] Overall = MadEvent + MEs	[MEs/second]
FORTRAN	68.93 = 2.84 + 66.09	9.73e+01
CPP/none	84.01 = 3.38 + 80.63	7.98e+01
CPP/sse4	46.29 = 3.04 + 43.25	1.49e+02
CPP/avx2	22.26 = 2.85 + 19.41	3.31e+02
CPP/512y	20.49 = 2.89 + 17.60	3.66e+02
CPP/512z	13.11 = 2.81 + 10.30	6.24e+02

GPU: ~x120 speedup for MEs, only ~x20 for full workflow *[Amdahl: p = 0.95 ⇒ max speedup = 20]*

(ME speedup would be ~x300 with 16k+ events per GPU grid, but Madevent CPU memory is limited to ~8k per grid)

See https://github.com/madgraph5/madgraph4gpu/raw/c2d22aebd45348e56a0477f0c85bab3d8eab18ab/epochX/cudacpp/CODEGEN/PLUGIN/CUDACPP_SA_OUTPUT/madgraph/iolibs/template_files/COPYRIGHT

```
Copyright (C) 2020-2023 CERN and UCLouvain.
Licensed under the GNU Lesser General Public License (version 3 or later).
All rights not expressly granted are reserved.

The copyright and license notice above cover the CUDACPP code-generating plugin
of the MadGraph5_aMC@NLO (in the following "MG5aMC") software, and all code
generated using that plugin. These are collectively referred to as "this work"
or "the MG5aMC CUDACPP plugin and the code that it generates", or more simply
as "the MG5aMC CUDACPP plugin", in the following and throughout this work.

The MG5aMC CUDACPP plugin and the code that it generates are based on the
initial work on porting MG5aMC to GPUs using CUDA and on speeding up MG5aMC on
CPUs using vectorized C++ by three original authors from CERN and UCLouvain.
The full development team currently includes the following authors :
  Stephan Hageboeck (CERN)
  Olivier Mattelaer (Universite Catholique de Louvain, original author)
  Stefan Roiser (CERN, original author)
  Andrea Valassi (CERN, original author)
  Zenny Wettersten (CERN)
See https://github.com/madgraph5/madgraph4gpu for more details. For the full
list of authors and collaborators of this work, see the file "AUTHORS" in the
same directory as this "COPYRIGHT" file in the source code of the plugin.

The MG5aMC CUDACPP plugin and the code that it generates are derived from, and
are intended to be used in combination with, the MG5aMC software and the code
that it generates. The MG5aMC software is developed by the MadGraph5_aMC@NLO
development team and contributors, also known as the "MadTeam", who are the
owners of its copyright and have licensed it as specified in
https://github.com/mg5amcnlo/mg5amcnlo/blob/main/madgraph/LICENSE.

The MG5aMC CUDACPP plugin and the code that it generates are free software;
you can redistribute them and/or modify them under the terms of the GNU Lesser
General Public License as published by the Free Software Foundation, either
version 3 or (at your option) any later version.

This work is distributed in the hope that it will be useful, but WITHOUT ANY
WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

The GNU Lesser General Public License (LGPL) version 3 is copied verbatim in
the file "COPYING.LESSER" in the same directory as this "COPYRIGHT" file. It is
also available at <https://www.gnu.org/licenses/lgpl-3.0.txt>.

This version of the GNU Lesser General Public License incorporates the terms
and conditions of version 3 of the GNU General Public License (GPL), which is
copied verbatim in the file "COPYING" in the same directory as this "COPYRIGHT"
file and is also available at <https://www.gnu.org/licenses/gpl-3.0.txt>.

In line with the license above, the authors emphasise the following points. For
the developers' and authors' protection, the GPL clearly explains that there is
no warranty for this free software. For both users' and authors' sake, the GPL
requires that modified versions be marked as changed, so that their problems
will not be attributed erroneously to authors of previous versions.
```

Proposed COPYRIGHT file

This would be distributed with the code-generating plugin, and also in generated code

A tale of two repositories (now)

mg5amcnlo

https://github.com/mg5amcnlo/mg5amcnlo

- *the* MG5AMC repo (previously launchpad)
- python framework, fortran codegen
- permissive NCSA-style license

A specific commit is in madgraph4gpu →

Important branches for GPU/SIMD work:

- **gpucpp** (the baseline: merge here!)
- gpucpp_june24 (channelid array) **MERGED**
- gpucpp_goodhel (new helicity filter) **WIP**
- gpucpp_for360 (complete 3.6.0 sync) **WIP**

madgraph4gpu

https://github.com/madgraph5/madgraph4gpu

- cudacpp plugin (cuda/c++ codegen)
- generated code, tests, results (+legacy stuff)
- more restrictive LGPL license

Includes mg5amcnlo as a git submodule

Important branches for GPU/SIMD work:

- **master** (the baseline: merge here!)
- master_june24 (...) **MERGED**
- master_goodhel (...) **WIP**
- master_for360 (...) **WIP**

Status: finally merged "june24" this week; now fixing the conflicts with "goodhel" and "for360"
Aim for a v3.6.0 release including the GPU/SIMD support... possibly by end September!?

Still a tale of two repositories (later)?

Option 1 – our assumption so far

mg5amcnlo
https://github.com/mg5amcnlo/mg5amcnlo
- *the* MG5AMC repo (NCSA-style)

Includes cudacpp as a git submodule ← in PLUGIN/CUDACPP_OUTPUT

mg5amcnlo_cudacpp *(OLD WIP AUG 2023)*
https://github.com/mg5amcnlo/mg5amcnlo_cudacpp
- cudacpp plugin (LGPL)

A specific commit is in mg5amcnlo

Option 2? – recent discussion AV/OM

mg5amcnlo https://github.com/mg5amcnlo/mg5amcnlo
Includes cudacpp as a subdirectory in PLUGIN/**CUDACPP_OUTPUT**

Advantages/Disadvantages?
- Option 1 gives cleaner separation; but merge conflicts with git submodules are hard
- Option 2 is easier to manage, but more monolithic; following up if licensing is ok

Packaging – ~~two weeks ago~~ *now*

Next priority: packaging (3)

- I am not entirely sure which option I prefer – I ask here before doing real work...
  - Option 1 gives cleaner separation; but merge conflicts with git submodules are hard
  - Option 2 easier to manage, but more monolithic; following up (OSPO) if licensing is ok
  - (Option 3 keep madgraph4gpu and restructure it? probably better not...)
- *I have a slight preference for Option 2 however (i.e. a single repo)*
  - a specific version of cudacpp needs a ~specific version of mg5amcnlo
  - a specific version of mg5amcnlo needs a ~specific version of cudacpp
  - ~~having them in a single repo simplifies this bi-directional dependency~~
    - ~~and, again, simplifies the handling of PRs, which may be a complete mess with git submodules~~
- *Concrete proposal for mg5amcnlo?*
  - ~~*mg5amcnlo has its own main branch for releases (currently branch "3.x" IIUC?)*~~
  - ~~*permanently maintain our "gpucpp" branch now including PLUGIN/CUDACPP_OUTPUT*~~
    - ~~*periodically merge gpucpp into 3.x (this is what other development lines do too, right?)*~~
- Things to do (AV), whether we go for Option 1 or Option 2
  - ~~Prepare the move out of madgraph4gpu, including history and preserving links~~
  - ~~Prepare some scripts for further resync from/to madgraph4gpu (there is still WIP there...)~~

Very useful discussion at the meeting 17 Sep (and later with OM)

And the answer is... ~ Option 3!

Keep mg5amcnlo as a git submodule in cudacpp

Download cudacpp as a tarball into mg5amcnlo

May keep madgraph4gpu ~as-is for the moment (and clean it later)

Issues and PRs remain

Commit history remains