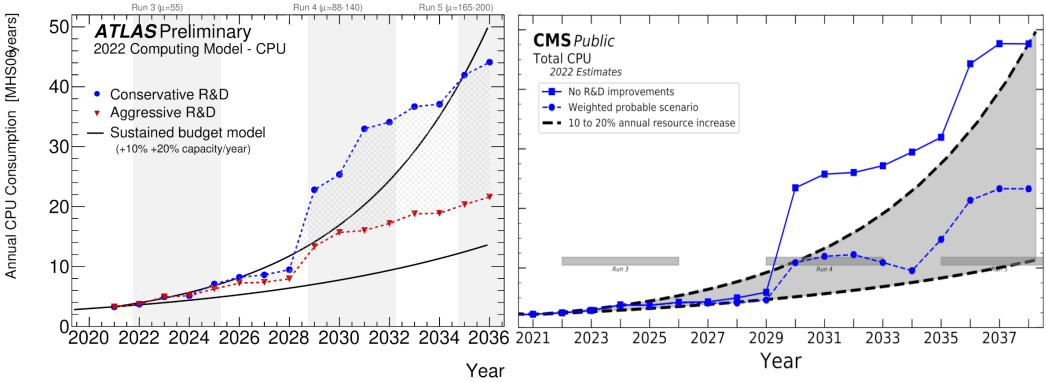
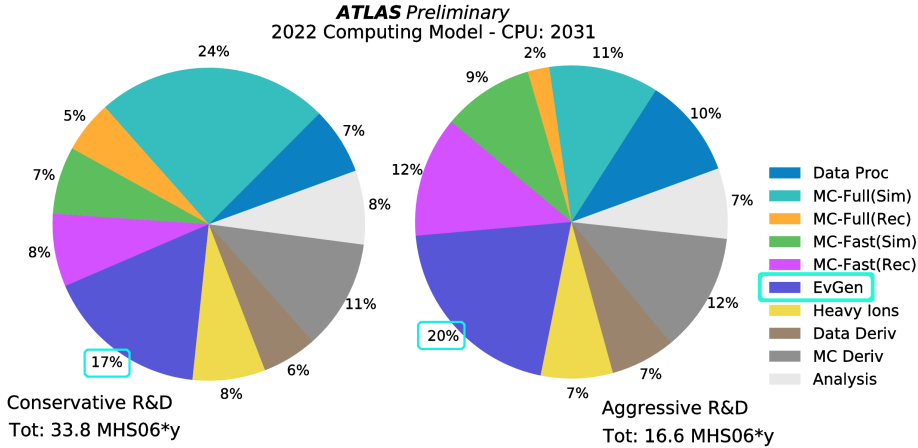

CERN



Projected CPU usage from present to 2036 (ATLAS) and 2038 (CMS).
(CERN-LHCC-2022-005, CMS-NOTE-2022-008)
ATLAS and CMS are the two biggest LHC experiments.

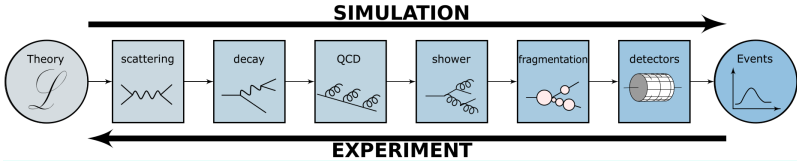


ATLAS projected CPU usage 2031. (CERN-LHCC-2022-005)
Highlighted is Monte Carlo event generation, the subject of this talk.

Hardware acceleration for hard event generation

Argonne: Taylor Childers, Walter Hopkins, Nathan Nichols
CERN: Laurence Field, Stephan Hageboeck, Filip Optołowicz, Stefan Roiser,
David Smith, Jorgen Teig, Andrea Valassi, Zenny Wettersten
UCLouvain: Olivier Mattelaer
UW-Madison: Carl Vuosalo

From theory to experiment...

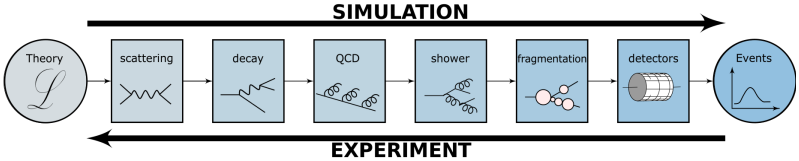


(figure taken from arXiv:2203.07460[1], modified)

- Theory given by a *Lagrangian*
  - Yields particle interactions
  - $\rightarrow$ Probability distributions
- Hard scattering probability
  ("fully analytic", subject of this talk)
- $\rightarrow$ Approximate (soft) scattering
- Jet fragmentation/hadronisation (remnants form e.g. nuclei)
- Finally, detector simulation, "what would we observe?"

[1] Anja Butter et al. "Machine learning and LHC event generation". In: *SciPost Phys.* 14 (2023), p. 079. DOI: 10.21468/SciPostPhys.14.4.079

...and back again



(figure taken from arXiv:2203.07460, modified)

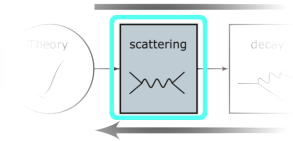
- Real detectors observe e.g. energy, momentum, charge
- Determine paths of high energy ("hard") particles
- ...and what particles they are

- Intersection $\rightarrow$ interaction point
- Unimaginably many events seen
  - Quantum mechanics probabilistic, no single event is representative
- Statistical/probabilistic analysis, *does this fit the theory?*

Hard events and cross sections

Hard event generation

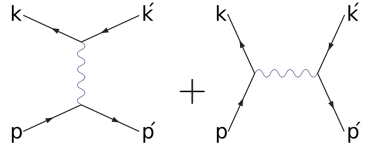
- First step in simulation chain: hard scattering probability
- Should give two outputs:
  1. Integrated process probability (*cross section* of the process)
  2. (Unweighted) singular events (input for further simulation)
- Integrals generally unsolvable
- $\rightarrow$ Monte Carlo integration!
  - Numerically integrate by sampling the distribution



- For a given phase space point, probability just linear algebra
- Simple *(not easy!)* to calculate
- Can use MC samples as events for simulation

Illustrative example: (hard) Bhabha scattering

- Illustration of calculation complexity
- Electron-antielectron scattering against/annihilating each other

$\mathcal{M} =$ 

Feynman diagrams by ChapEpstn, CC SA-BY 4.0

$$= -e^2 \frac{(\bar{v}_k \gamma^\mu v_{k'})\,(\bar{u}_{p'} \gamma_\mu u_p)}{(k-k')^2} + e^2 \frac{(\bar{v}_k \gamma^\nu u_p)\,(\bar{u}_{p'} \gamma_\nu v_{k'})}{(k+p)^2}$$

- $\bar{v}$, v, $\bar{u}$, u are 4-vectors, γ are 4×4 matrices
- Gammas and u, v act on different spaces — they commute
- Amplitude: Complex number, probability is its absolute square

Illustrative example: (hard) Bhabha scattering

- Tedious calculations, *but it is just linear algebra*
- Bhabha scattering: $\sim$ simplest HEP processes
  - Can be solved analytically — generally not the case
- Taylor expansion formula — this is *leading order* term (LO)
  - Most compute spent on *next-to-leading order* (NLO) contributions
- Numerical evaluations developed since the 90s[2,3]
- Experimental precision $\sim$ Monte Carlo error
- At High Luminosity LHC, expect $\sim 10\times$ more measurements
- $\rightarrow$ Need to generate $\sim 10\times$ more events

[2] H. Murayama, I. Watanabe, and K. Hagiwara. *HELAS: HELicity amplitude subroutines for Feynman diagram evaluations*. Tech. rep. KEK-91-11. National Lab. for High Energy Physics, Tsukuba, Ibaraki, 1998

[3] T. Stelzer and W.F. Long. "Automatic generation of tree level helicity amplitudes". In: *CPC* 81.3 (July 1994), pp. 357–371

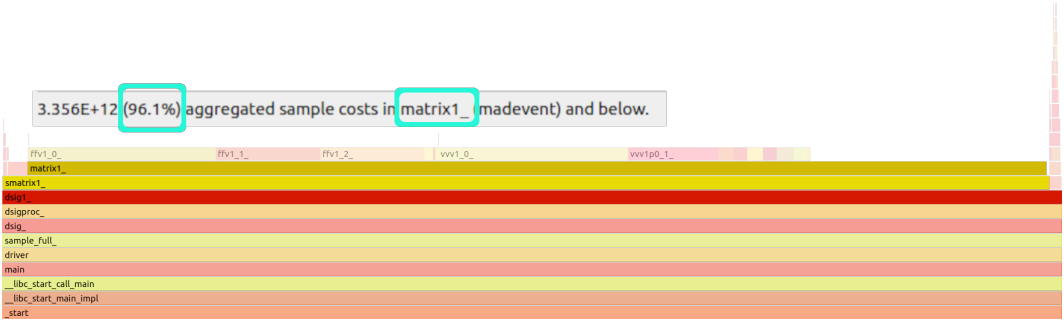
Numerical tools

MadGraph5_aMC@NLO[4] (MGaMC)

- MGaMC: Software for particle physics theory calculations
- At LHC experiments, one of the main event generators
  - Code generator — outputs (`Fortran`) code for input process
- (Also popular among theorists)

- Calculates scatterings using *helicity amplitudes*
- $\rightarrow$ Spin-summed probabilities, same expressions with different input vectors
- Same calculation for each input for each MC-sampled point $\rightarrow$ perfect vectorisation

[4] J. Alwall et al. "The automated computation of tree-level and next-to-leading order differential cross sections, and their matching to parton shower simulations". In: *JHEP* 2014.7 (July 2014)



Performance profile, MGaMC `Fortran` output ($gg \to t\bar{t}gg$, LO)

x-axis represents fraction of runtime spent in given function,

y-axis represents stack depth for given function call.

For in-depth coverage on flamegraphs, see e.g. Brendan Gregg.

Examples of MGaMC `Fortran` code

```
429           ...
430           CALL IXXXXX(P(0,1),ZERO,NHEL(1),+1*IC(1),W(1,1))
431           CALL OXXXXX(P(0,2),ZERO,NHEL(2),-1*IC(2),W(1,2))
432           CALL VXXXXX(P(0,3),ZERO,NHEL(3),+1*IC(3),W(1,3))
433           CALL VXXXXX(P(0,4),ZERO,NHEL(4),+1*IC(4),W(1,4))
434           CALL VXXXXX(P(0,5),ZERO,NHEL(5),+1*IC(5),W(1,5))
435           CALL VVV1P0_1(W(1,3),W(1,4),GC_10,ZERO, FK_ZERO,W(1,6))
436           CALL FFV1_1(W(1,2),W(1,5),GC_11,ZERO, FK_ZERO,W(1,7))
437     C     Amplitude(s) for diagram number 1
438           CALL FFV1_0(W(1,1),W(1,7),W(1,6),GC_11,AMP(1))
439           CALL FFV1_2(W(1,1),W(1,5),GC_11,ZERO, FK_ZERO,W(1,8))
440     C     Amplitude(s) for diagram number 2
441           CALL FFV1_0(W(1,8),W(1,2),W(1,6),GC_11,AMP(2))
442           CALL FFV1P0_3(W(1,1),W(1,2),GC_11,ZERO, FK_ZERO,W(1,9))
443     C     Amplitude(s) for diagram number 3
444           CALL VVV1_0(W(1,6),W(1,5),W(1,9),GC_10,AMP(3))
445           CALL VVV1P0_1(W(1,3),W(1,5),GC_10,ZERO, FK_ZERO,W(1,6))
446           CALL FFV1_1(W(1,2),W(1,4),GC_11,ZERO, FK_ZERO,W(1,10))
447     C     Amplitude(s) for diagram number 4
448           CALL FFV1_0(W(1,1),W(1,10),W(1,6),GC_11,AMP(4))
449           CALL FFV1_2(W(1,1),W(1,4),GC_11,ZERO, FK_ZERO,W(1,11))
450           ...
```

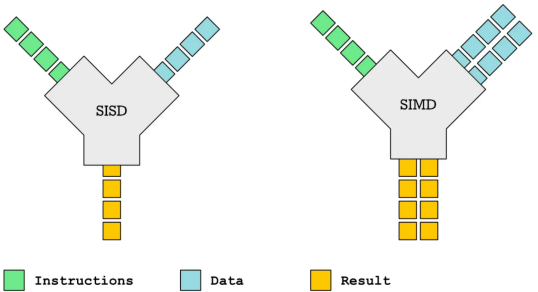
Excerpt from `matrix1.f`.

```
38            ...
39             TMP0 = (V3(3)*P1(0)-V3(4)*P1(1)-V3(5)*P1(2)-V3(6)*P1(3))
40             TMP2 = (V3(3)*P2(0)-V3(4)*P2(1)-V3(5)*P2(2)-V3(6)*P2(3))
41             TMP4 = (P1(0)*V2(3)-P1(1)*V2(4)-P1(2)*V2(5)-P1(3)*V2(6))
42             TMP5 = (V2(3)*P3(0)-V2(4)*P3(1)-V2(5)*P3(2)-V2(6)*P3(3))
43             TMP6 = (V3(3)*V2(3)-V3(4)*V2(4)-V3(5)*V2(5)-V3(6)*V2(6))
44             DENOM = COUP/(P1(0)**2-P1(1)**2-P1(2)**2-P1(3)**2 -
45            $ M1 * (M1 -CI * W1))
46             V1(3)= DENOM*(TMP6*(-CI*(P2(0))+CI*(P3(0)))+(V2(3)*
47            $ (-CI*(TMP0)+CI*(TMP2))+V3(3)*(+CI*(TMP4)-CI*(TMP5))))
48             V1(4)= DENOM*(TMP6*(-CI*(P2(1))+CI*(P3(1)))+(V2(4)*
49            $ (-CI*(TMP0)+CI*(TMP2))+V3(4)*(+CI*(TMP4)-CI*(TMP5))))
50             V1(5)= DENOM*(TMP6*(-CI*(P2(2))+CI*(P3(2)))+(V2(5)*
51            $ (-CI*(TMP0)+CI*(TMP2))+V3(5)*(+CI*(TMP4)-CI*(TMP5))))
52             V1(6)= DENOM*(TMP6*(-CI*(P2(3))+CI*(P3(3)))+(V2(6)*
53            $ (-CI*(TMP0)+CI*(TMP2))+V3(6)*(+CI*(TMP4)-CI*(TMP5))))
54             END
```

Excerpt from `VVV1P0_1.f`.

How do we speed this up?

Data parallelism for CPUs (SIMD)

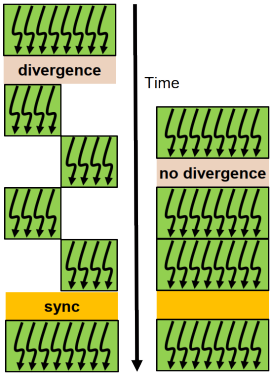


(E. Engheim, ITNEXT)

- Single instruction, single data
  - Run operations sequentially
  - One instruction on one datum
  - $\rightarrow$ Repetitive tasks take long
- Single instruction, multiple data
  - Vector of data in register
  - Run instruction on entire register
  - $\rightarrow$ Accelerate repetitive tasks
  - *(No performance cost)*
  - E.g. AVX architectures on Intel, AMD HPC-CPUs

Data parallelism for GPUs (SIMT)

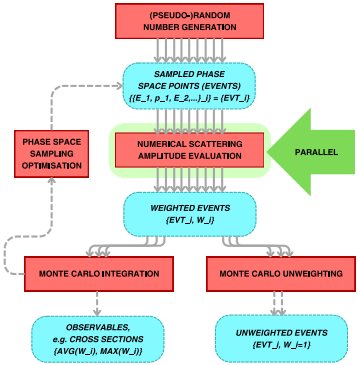
- Single instruction, multiple threads
  - Many processing units in parallel (threads)
  - Operations on distinct data sets (registers)
  - Simultaneous operations $\implies$ **lockstep**
- Lockstep processing
  - Each thread runs same operation at same time
  - Threads "pause" for branching code
  - $\Rightarrow$ Need to minimise thread divergence when programming SIMT hardware, e.g. GPUs



(NVidia Volta whitepaper, here modified)

Parallel helicity amplitudes

"[E]vent-level parallelism [seems] an appropriate approach" (arXiv:2004.13687)



- Parallelism in event generation
  - Same probability distribution, many points
  - Same calculations for each one
  - $\rightarrow$ No divergence between events
- Scattering amplitudes in MGaMC
  - MGaMC generates phase space points
  - Sends events to amplitude evaluation
  - $\rightarrow$ Evaluate amplitudes in parallel

Parallelised MGaMC implementations[5]

Two separate codebases

- CUDACPP
  - Vectorised C++ routines with CUDA GPU API integration
    - Replaces LO scattering probability Fortran code with C++/CUDA

  ⇒ vector CPU output

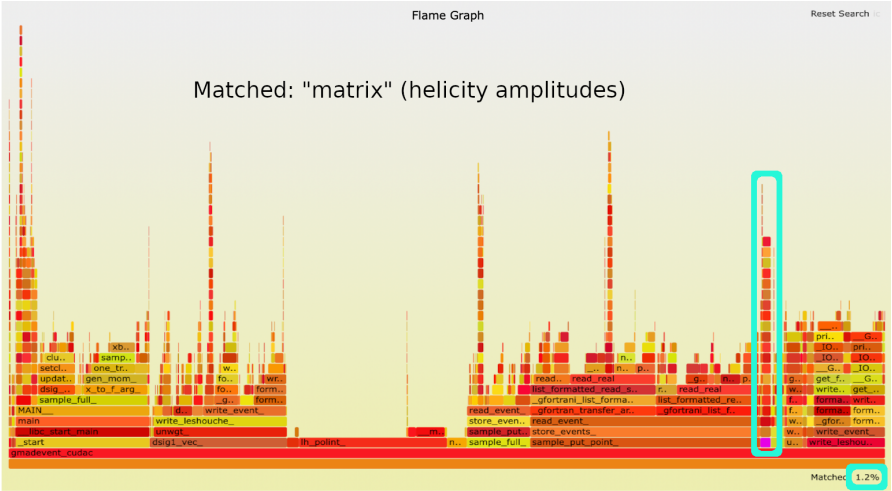
  ⇒ NVidia GPU output

  ⇒ HIP API backend (compile time toggle), AMD GPU output
- SYCL
  - Tracks CUDACPP development, ports it to SYCL portability framework
    - Work on integrating with CUDACPP codebase just started

  ⇒ Intel hardware output (CPU, GPU)

  ⇒ NVidia, AMD GPU output

[5] Stephan Hageböck et al. "Madgraph5_aMC@NLO on GPUs and vector CPUs: Experience with the first alpha release". In: *EPJ Web of Conf.* 295 (2024), p. 11013. DOI: 10.1051/epjconf/202429511013. URL: https://doi.org/10.1051/epjconf/202429511013



Performance profile, MGaMC `CUDACPP` ($gg \to t\bar{t}gg$), NVidia A100

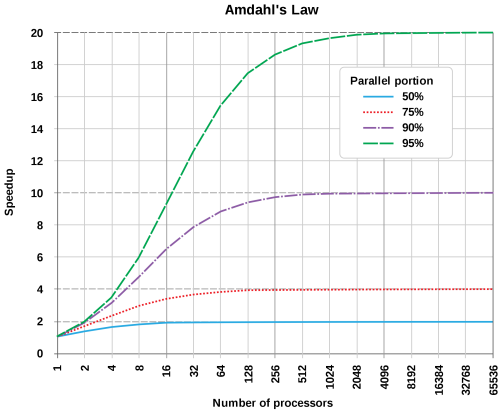
Note: Only amplitudes in `CUDA`, rest in `Fortran` or `Python` driver.

For in-depth coverage on flamegraphs, see e.g. Brendan Gregg.

Amdahl's law

- Acceleration limited by time spent in *non-parallel* code
- Parallelising code which originally takes fraction p of runtime can speed up program by

$$\text{Acceleration} \leq \frac{1}{1-p}.$$



(Daniels220, CC BY-SA 3.0)

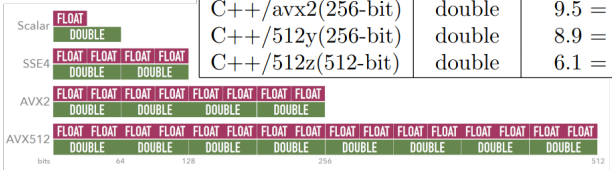
CUDACPP performance, NVidia GPU

Process	Madevent 262 144 events		
	Total	Momenta+unweight	Scattering
$e^+e^- \to \mu^+\mu^-$	17.9 s	10.2 s	7.8 s
+CUDA Tesla A100	10.0 s	10.0 s	0.02s
	1.8x	1.0 x	**390 x**
$gg \to t\bar{t}gg$	209.3 s	7.8 s	201.5 s
+CUDA Tesla A100	8.4 s	7.8 s	0.6 s
	24.9x	1.0 x	**336 x**
$gg \to t\bar{t}ggg$	2507.6 s	12.2 s	2495.3 s
+CUDA Tesla A100	30.6 s	14.1 s	16.5 s
	82.0x	0.9 x	**151 x**

Event generation time comparison, offloading scattering amplitudes on GPU.
Measurements are run single-threaded on an AMD EPYC 7313 CPU.
Comparisons are between single-threaded CPU with and without GPU offloading.
Note: Maximum speedup in e.g. $gg \to t\bar{t}gg \sim 25.6\times$ by Amdahl.

CUDACPP performance, vector CPU

$gg \to t\bar{t}gg$	MEs precision	madevent $t_{\text{TOT}} = t_{\text{Mad}} + t_{\text{MEs}}$ [sec]	madevent $N_{\text{events}}/t_{\text{TOT}}$ [events/sec]	madevent $N_{\text{events}}/t_{\text{MEs}}$ [MEs/sec]
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)
C++/none(scalar)	double	37.8 = 1.7 + 36.0	2.17E3 (x1.0)	2.28E3 (x1.0)
C++/sse4(128-bit)	double	19.4 = 1.7 + 17.8	4.22E3 (x1.9)	4.62E3 (x2.0)
C++/avx2(256-bit)	double	9.5 = 1.7 + 7.8	8.63E3 (x3.9)	1.05E4 (x4.6)
C++/512y(256-bit)	double	8.9 = 1.8 + 7.1	9.29E3 (x4.2)	1.16E4 (x5.0)
C++/512z(512-bit)	double	6.1 = 1.8 + 4.3	1.35E4 (x6.1)	1.91E4 (x8.3)



Event generation time comparison, levels of vectorisation.
Measurements taken on an Intel Gold 6148 CPU.
Comparisons are of compilation with different levels of vectorisation, no further modifications.

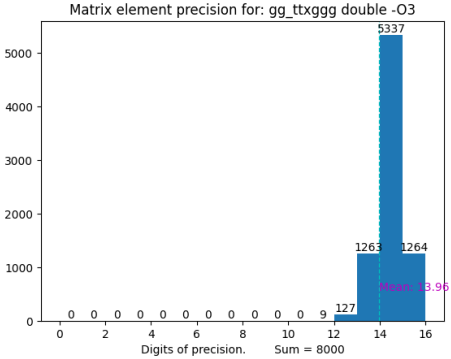
CUDACPP performance, float precision

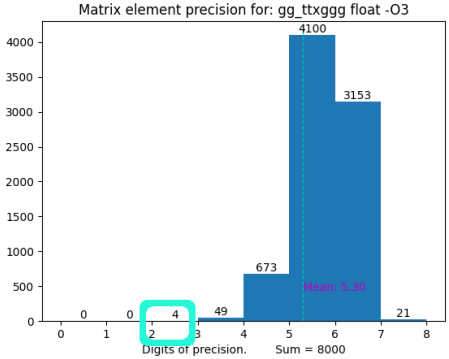
$gg \to t\bar{t}gg$	MEs precision	madevent $t_{\rm TOT} = t_{\rm Mad} + t_{\rm MEs}$ [sec]	madevent $N_{\rm events}/t_{\rm TOT}$ [events/sec]	madevent $N_{\rm events}/t_{\rm MEs}$ [MEs/sec]
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)
C++/none(scalar)	double	37.8 = 1.7 + 36.0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

Event generation time comparison, levels of vectorisation and double vs float (single) precision.
Measurements taken on an Intel Gold 6148 CPU.
Comparisons are of compilation with different levels of vectorisation, no further modifications.

What about numerical stability?

CADNA measurements for MGaMC



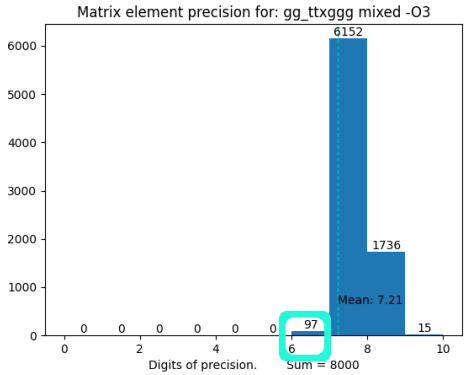


Measurements taken with CADNA over 8 000 phase space points for process $gg \to t\bar{t}\,ggg$.

Mixed precision

- Helicity amplitudes vary orders of magnitude
- So-called "colour algebra" consistent in magnitude
  - Part of calculation distinct from helicity amplitudes, easily factored

⇒ Check stability with only colour algebra in float (*mixed precision*)



Measurements taken with CADNA over 8 000 phase space points for $gg \to t\bar{t}ggg$, mixed precision.

Acceleration from mixed precision

		madevent		
CUDA grid size		8192		
$gg \to t\bar{t}ggg$	MEs precision	$t_{\text{TOT}} = t_{\text{Mad}} + t_{\text{MEs}}$ [sec]	$N_{\text{events}}/t_{\text{TOT}}$ [events/sec]	$N_{\text{events}}/t_{\text{MEs}}$ [MEs/sec]
Fortran	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)

Event generation time comparison, offloading scattering amplitudes on GPU, levels of precision. Measurements taken single-threaded on an Intel Silver 4216 CPU and an NVidia V100 GPU. Comparisons are between single-threaded CPU with and without GPU offloading.

Summary

Summary

- Helicity amplitudes are an optimal case for data parallelism
  - Hard event generation: Evaluating same process at many points
  - With vector CPUs we see theoretical maximum gain
  - With GPUs, for relevant processes, achieve speedup of $10 - 100$
- Our vectorised MGaMC plugin implements this accessibly
  - MGaMC plugin has same command line interface and arguments
  - Working with LHC experiments to use this in their workflows
- Conclusions
  - Without vectorisation, HiLumi-LHC targets are unreachable
    - Helicity amplitudes are perfect for hardware acceleration
  - Numerical stability *generally* requires double precision
  - This has all been done for *leading order* calculations:
    - Most of compute time in event generation is on *next-to-leading order*
    - $\rightarrow$ Acceleration seen here motivates further NLO development


CERN