Madgraph5_aMC@NLO on GPUs and vector CPUs: experience with the first alpha release

Taylor Childers
Walter Hopkins
Nathan Nichols

Argonne NATIONAL LABORATORY

Laurence Field
Stephan Hageboeck
Stefan Roiser
David Smith
Jorgen Teig
Andrea Valassi
Zenny Wettersten

CERN

Olivier Mattelaer

UCL Université catholique de Louvain

Carl Vuosalo

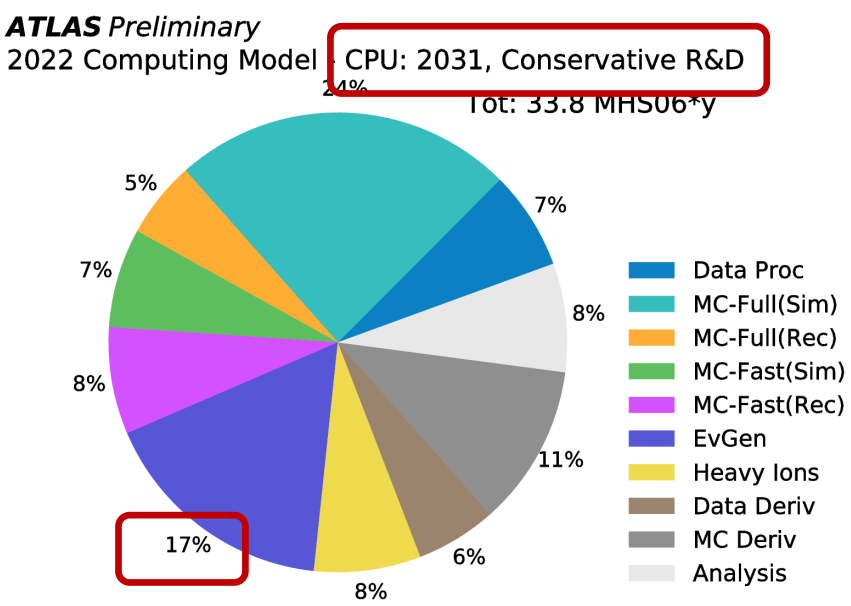
WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

CHEP, Norfolk, VA, 08 May 2023

https://indico.jlab.org/event/459/contributions/11829/

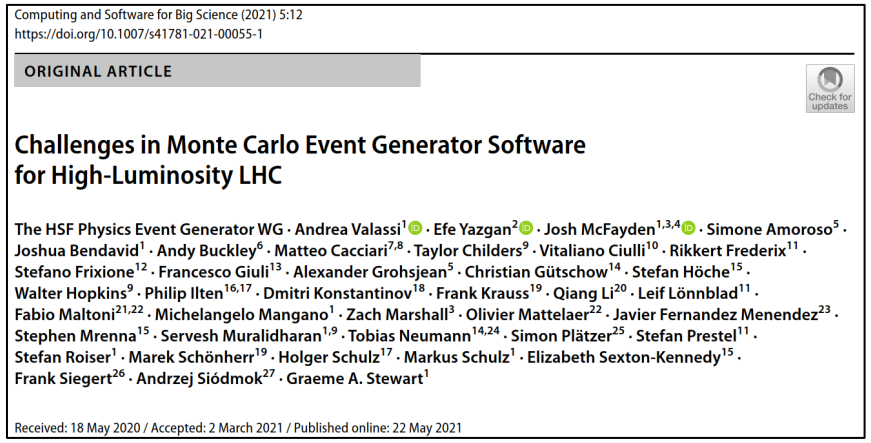
Motivation: Monte Carlo Event Generators in WLCG computing

- HL-LHC computing needs expected to outgrow resource growth
  - Need R&D on software to improve efficiency and port it to new resources, such as GPUs at HPC centres



CERN-LHCC-2022-005

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



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

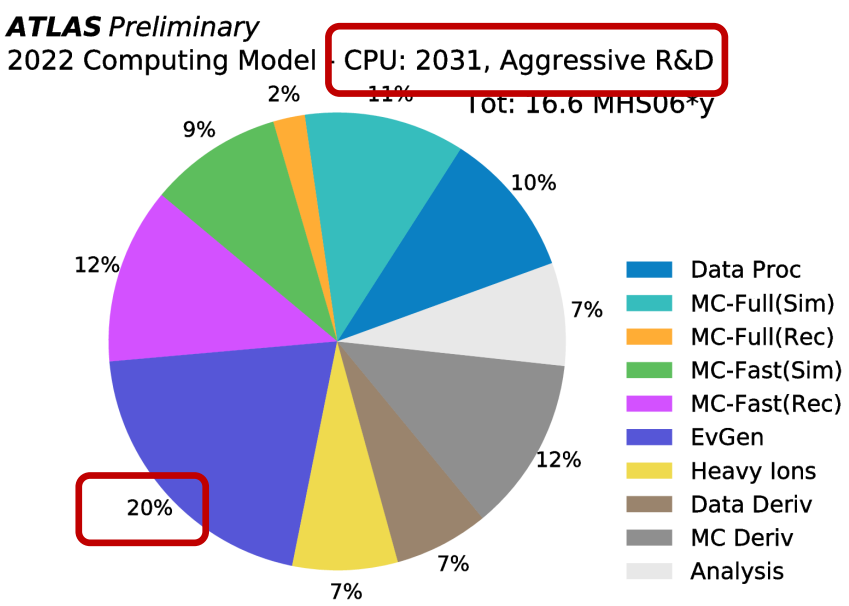
The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · 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

- MC generators projected to use 10% - 20% of ATLAS/CMS WLCG CPU budget
  - Many ways to speed them up – see the HEP Software Foundation (HSF) Generator WG review paper
  - *Ideal candidates to exploit data parallelism in GPUs (SIMT) and CPUs (SIMD)?*

Motivation: Monte Carlo Event Generators in WLCG computing

- HL-LHC computing needs expected to outgrow resource growth
  - Need R&D on software to improve efficiency and port it to new resources, such as GPUs at HPC centres



CERN-LHCC-2022-005

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

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

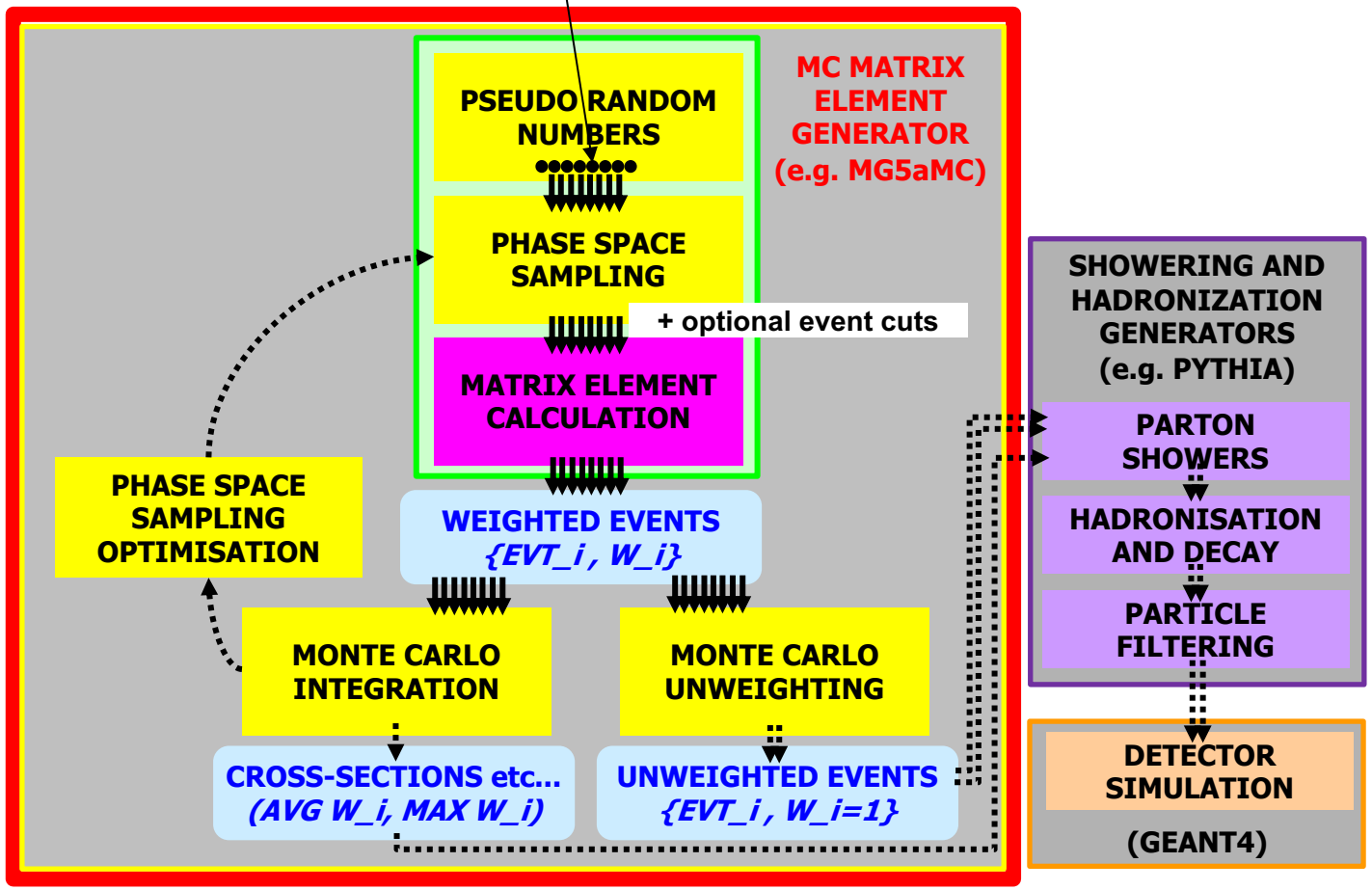
- MC generators projected to use 10% - 20% of ATLAS/CMS WLCG CPU budget
  - Many ways to speed them up – see the HEP Software Foundation (HSF) Generator WG review paper
  - *Ideal candidates to exploit data parallelism in GPUs (SIMT) and CPUs (SIMD)?*

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

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

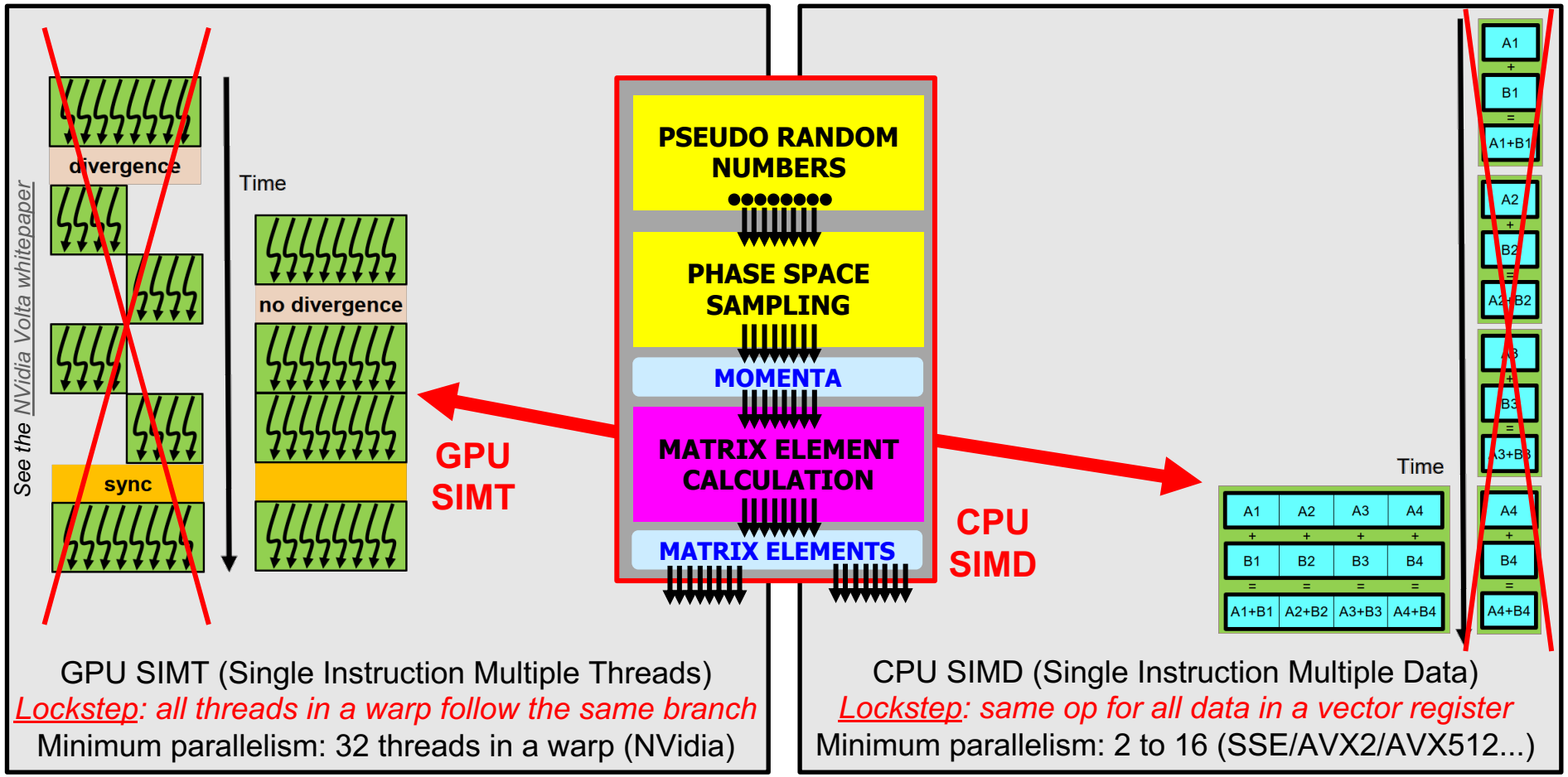
This can be parallelized (SIMT/SIMD and multithreading)

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



ME generator 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)

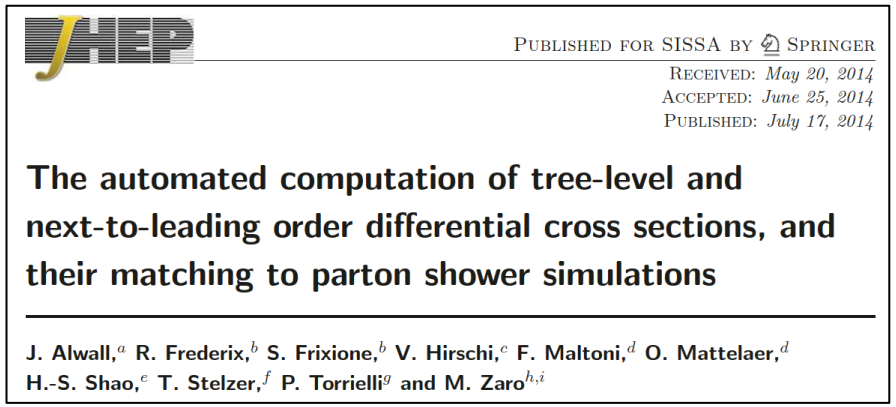


GPU SIMT (Single Instruction Multiple Threads)
Lockstep: all threads in a warp follow the same branch
Minimum parallelism: 32 threads in a warp (NVidia)

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

Madgraph5_aMC@NLO (MG5aMC)

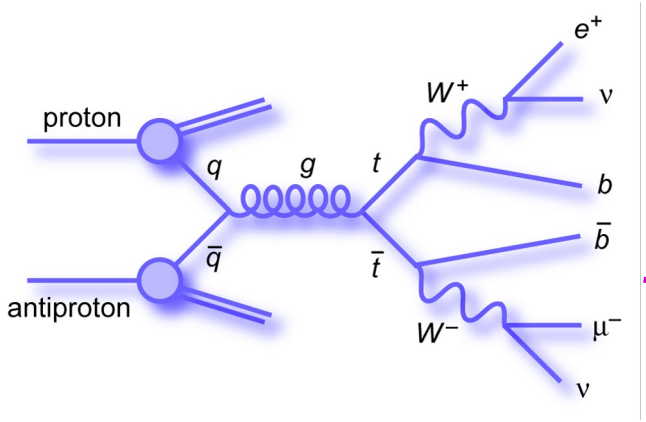
- One of the workhorses for event generation in ATLAS and CMS!

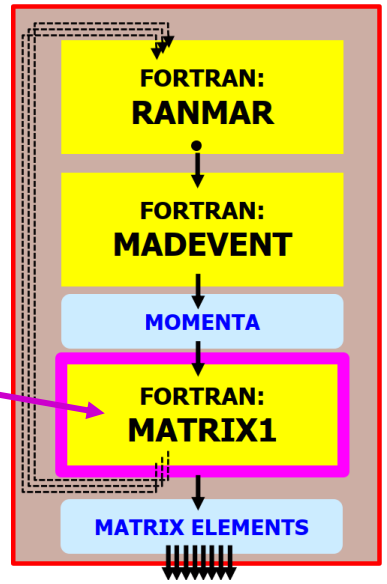

JHEP
Published for SISSA by Springer
Received: May 20, 2014
Accepted: June 25, 2014
Published: July 17, 2014

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

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

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

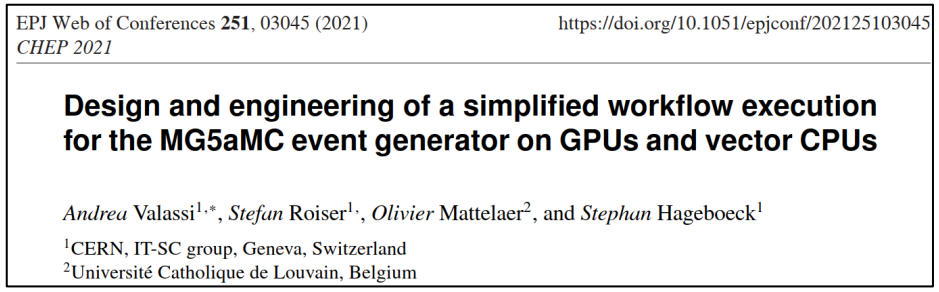




- MG5aMC production version is in Fortran
  - Outer shell: Madevent (random sampling, integration and event generation + I/O, multi-jet merging...)
  - Matrix Element (ME) core
    - MG5aMC **generates ME code** for each physics process
    - MEs may take >95% of the CPU time for complex processes (e.g. $gg \to t\bar{t}ggg$)
    - High potential for acceleration on CPUs / GPUs

MG5aMC and the madgraph4gpu project

- *madgraph4gpu: speed up Matrix Element calculation in MG5aMC* on GPUs and vector CPUs
  - Collaboration of theoretical/experimental physicists with software engineers – born in the HSF generator WG
  - More history and details in vCHEP2021 and ICHEP2022

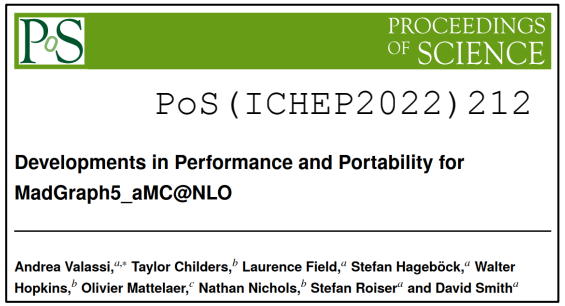

EPJ Web of Conferences **251**, 03045 (2021) https://doi.org/10.1051/epjconf/202125103045
CHEP 2021

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

Andrea Valassi[1,*], *Stefan* Roiser[1], *Olivier* Mattelaer[2], and *Stephan* Hageboeck[1]

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

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


PROCEEDINGS OF SCIENCE

PoS(ICHEP2022)212

Developments in Performance and Portability for MadGraph5_aMC@NLO

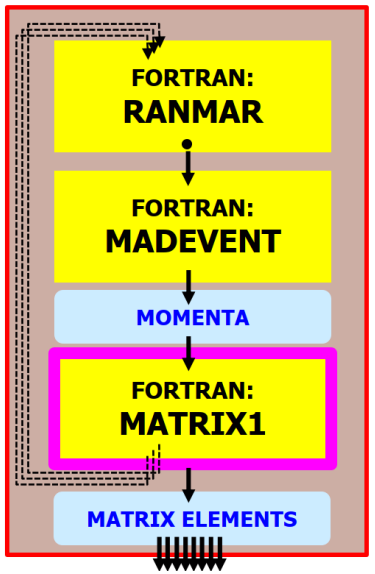
Andrea Valassi,[a,*] Taylor Childers,[b] Laurence Field,[a] Stefan Hageböck,[a] Walter Hopkins,[b] Olivier Mattelaer,[c] Nathan Nichols,[b] Stefan Roiser[a] and David Smith[a]

https://doi.org/10.22323/1.414.0212

- Two parallel approaches to reimplement the ME calculation
  - (1) "CUDACPP", our initial single-code CUDA/C++ back-end targeting NVidia GPUs and SIMD on vector CPUs
  - (2) Portability Frameworks (PFs: Alpaka, Kokkos, SYCL); added to gain experience with portability

MG5aMC: old and new architecture designs

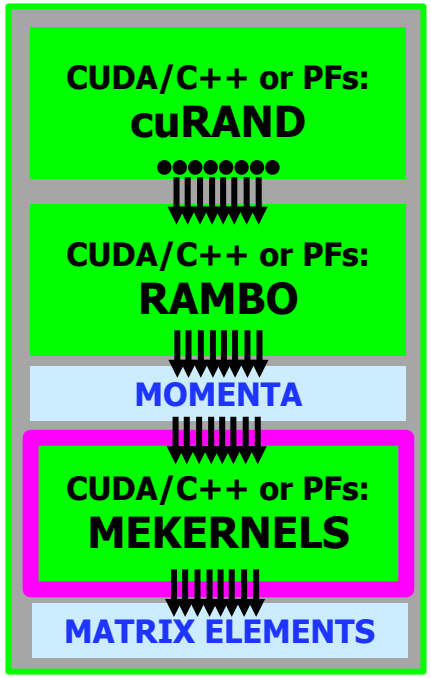
OLD MADEVENT
(CURRENT: LHC PROD)
SINGLE-EVENT API



MATRIX ELEMENT:
CPU BOTTLENECK
IN OLD MADEVENT

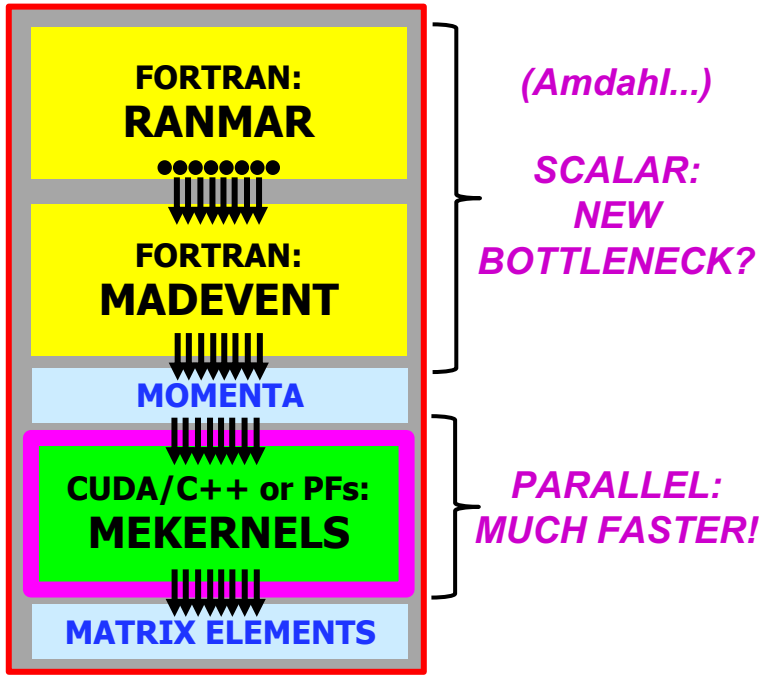
First we developed
the new ME engines
in standalone applications

1. STANDALONE
(TOY APPLICATIONS)
MULTI-EVENT API



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

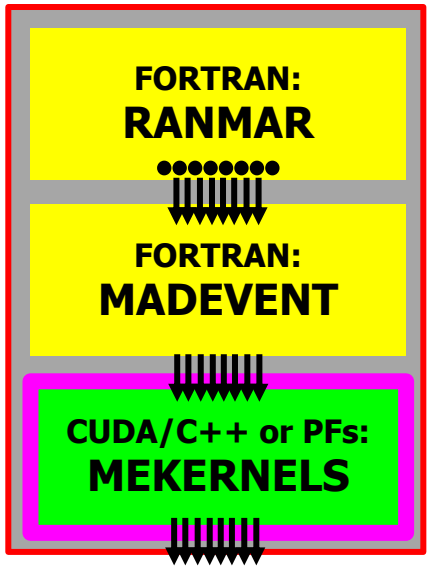
2. NEW MADEVENT
(GOAL: LHC PROD)
MULTI-EVENT API



MadEvent + CUDA Speed up

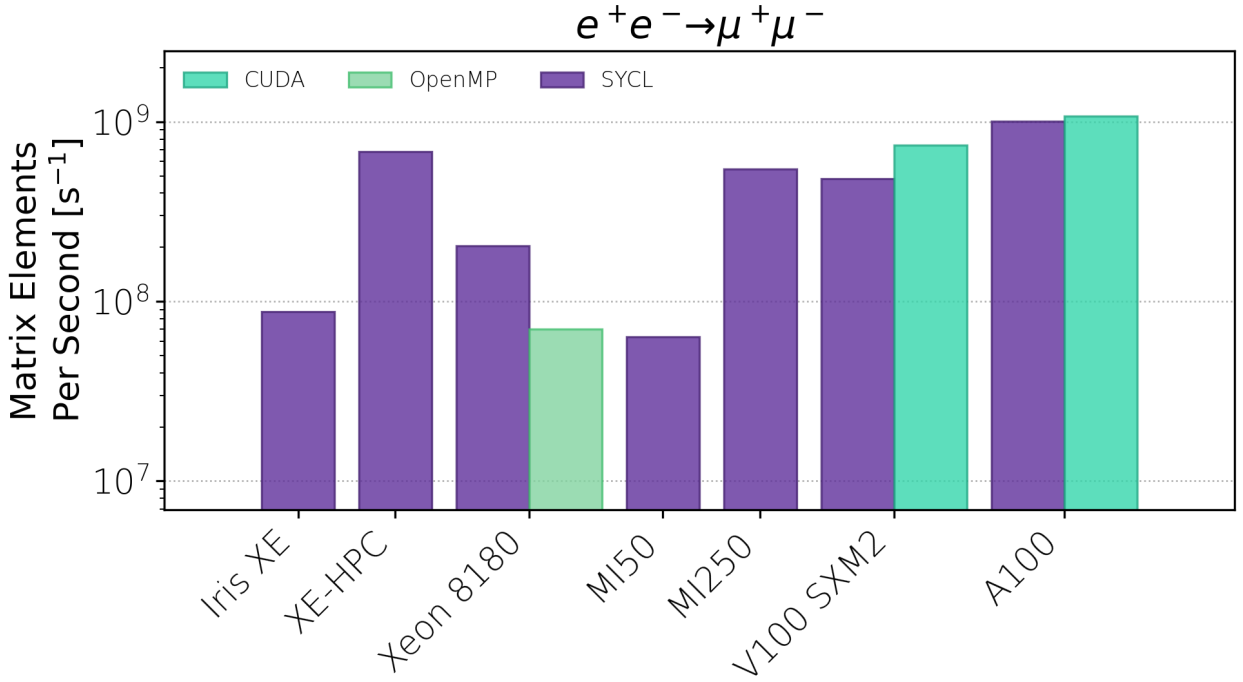
- Data-centre GPUs such as the Tesla A100 speed up the matrix-element calculations significantly
  - Simpler GPUs lack double-precision performance
  - So far, Madgraph only works reliably in double-precision mode

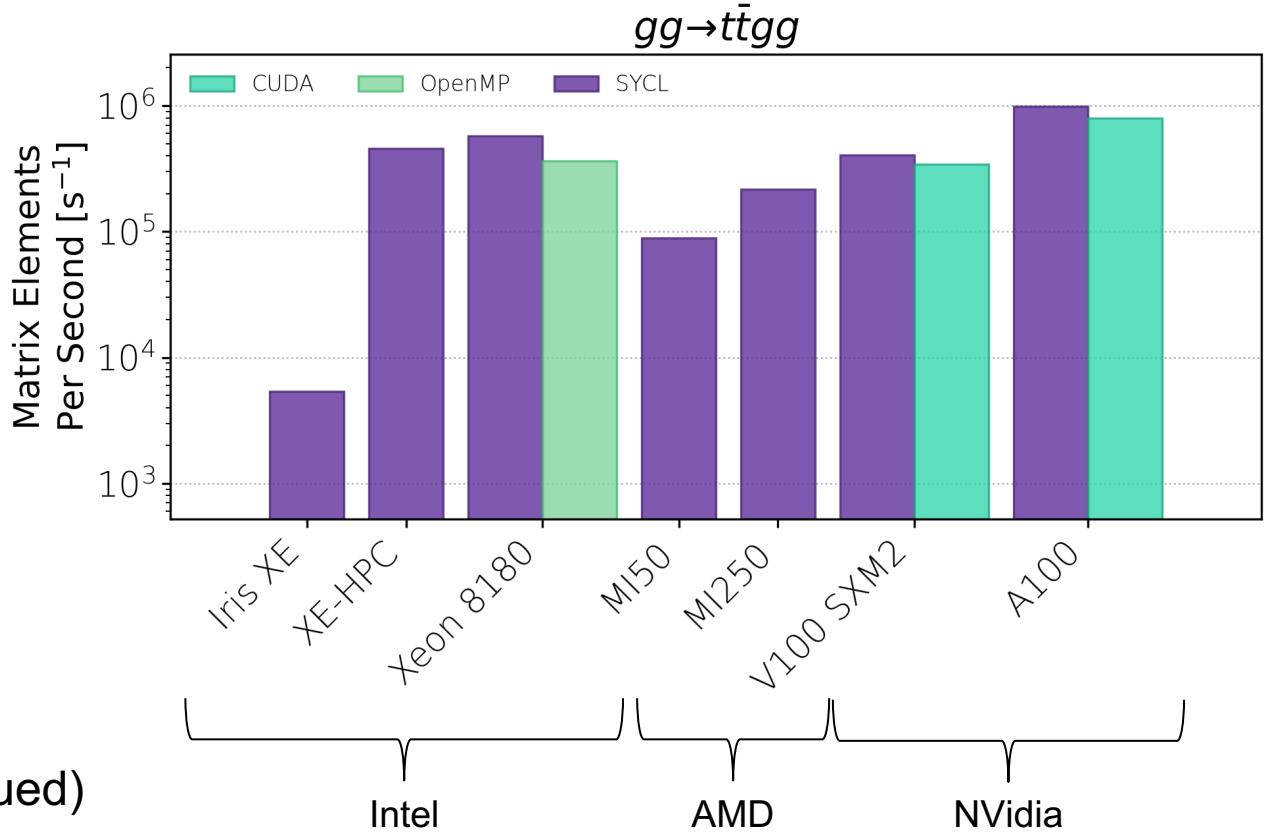
Process	Madevent 262 144 events			Standalone CUDA
	Total	Momenta+unweight	Matrix elm	ME Throughput
$e^+e^- \to \mu^+\mu^-$	17.9 s	10.2 s	7.8 s	$1.9 \times 10^6 s^{-1}$
+CUDA Tesla A100	10.0 s	10.0 s	0.02s	$633.8 \times 10^6 s^{-1}$
	1.8 x	1.0 x	390 x	334 x
$gg \to t\bar{t}gg$	209.3 s	7.8 s	201.5 s	$2.8 \times 10^3 s^{-1}$
+CUDA Tesla A100	8.4 s	7.8 s	0.6 s	$758.9 \times 10^3 s^{-1}$
	24.9 x	1.0 x	336 x	271 x
$gg \to t\bar{t}ggg$	2507.6 s	12.2 s	2495.3 s	$1.1 \times 10^2 s^{-1}$
+CUDA Tesla A100	30.6 s	14.1 s	16.5 s	$170.7 \times 10^2 s^{-1}$
	82.0 x	0.9 x	151 x	155 x



- FORTRAN parts limit total achievable speed up (Amdahl's law)
  - Especially for simple SM processes
  - E.g. phase-space sampling, generation of momenta, unweighting

What about non-NVidia architectures?

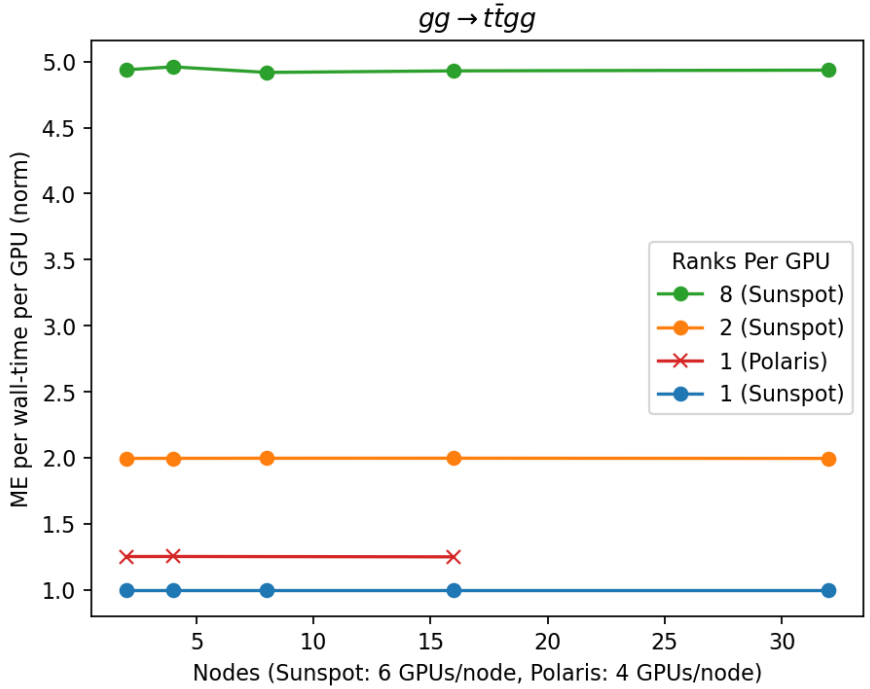


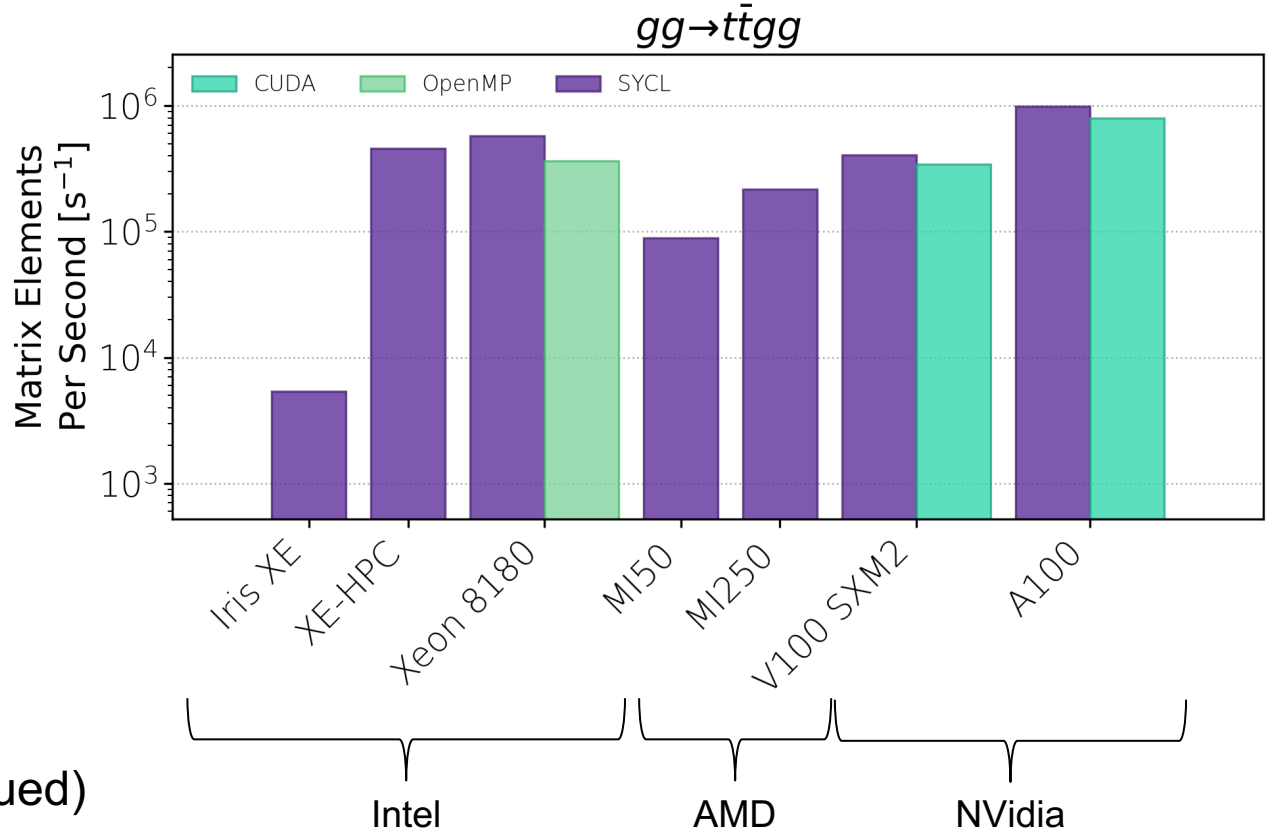


- Initially, ported madgraph C++ matrix elements to CUDA
- Also tested SYCL, as well as Kokkos / Alpaka (discontinued)
- SYCL:
  - Similar performance as for direct CUDA
  - Portability to AMD / Intel GPUs
- Good scaling observed on Aurora supercomputer testbed “Sunspot”

The XE-HPC results are from early silicon devices and early oneAPI DPC++ software stack. The GPUs are currently used at the Argonne Leadership Computing Facility and other customer sites to prepare code for future Intel data center GPUs, including those to be used in the Aurora exascale supercomputer.

What about non-NVidia architectures?



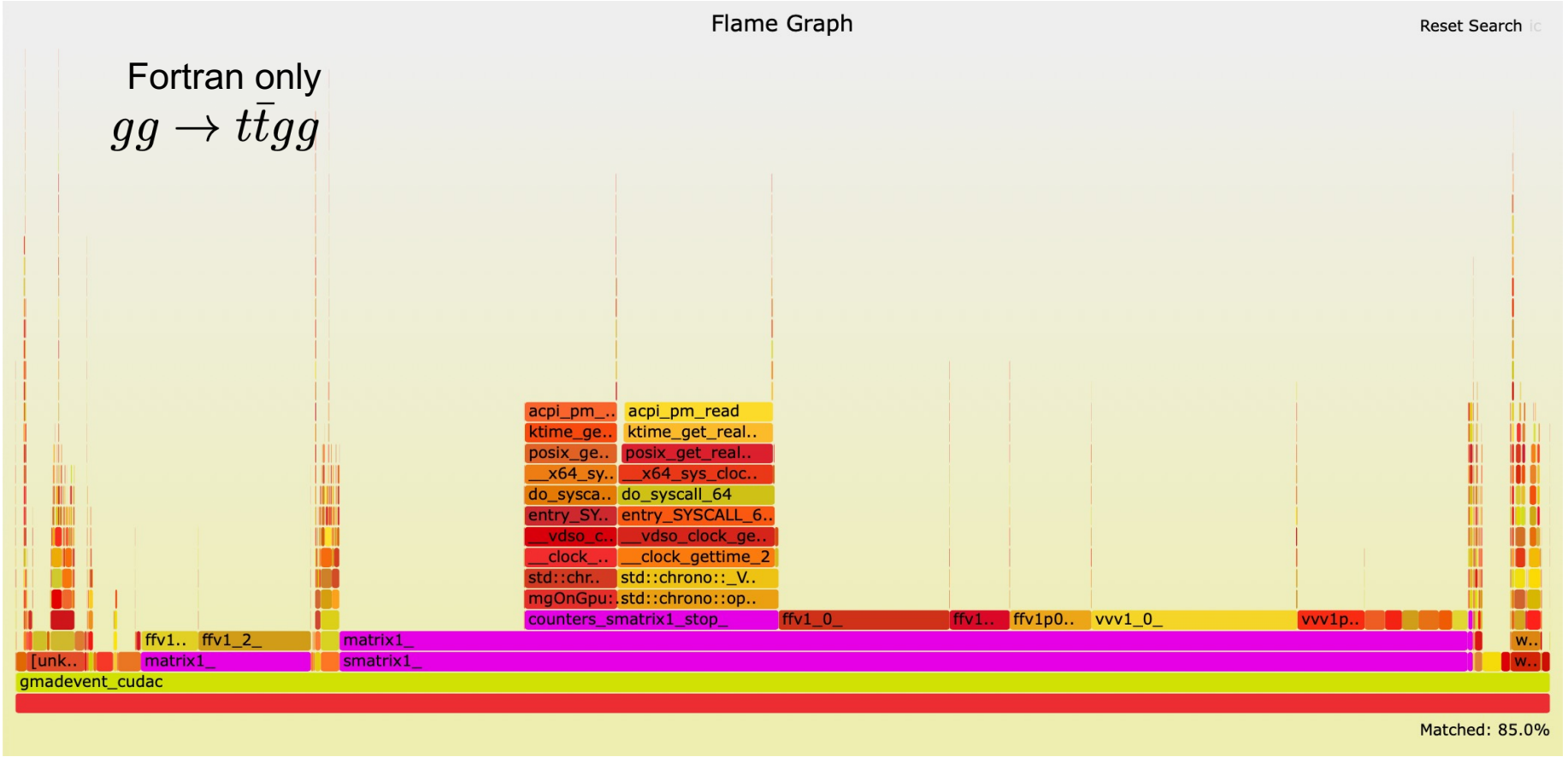


- Initially, ported madgraph C++ matrix elements to CUDA
- Also tested SYCL, as well as Kokkos / Alpaka (discontinued)
- SYCL:
  - Similar performance as for direct CUDA
  - Portability to AMD / Intel GPUs
- Good scaling observed on Aurora supercomputer testbed “Sunspot”

The XE-HPC results are from early silicon devices and early oneAPI DPC++ software stack. The GPUs are currently used at the Argonne Leadership Computing Facility and other customer sites to prepare code for future Intel data center GPUs, including those to be used in the Aurora exascale supercomputer.

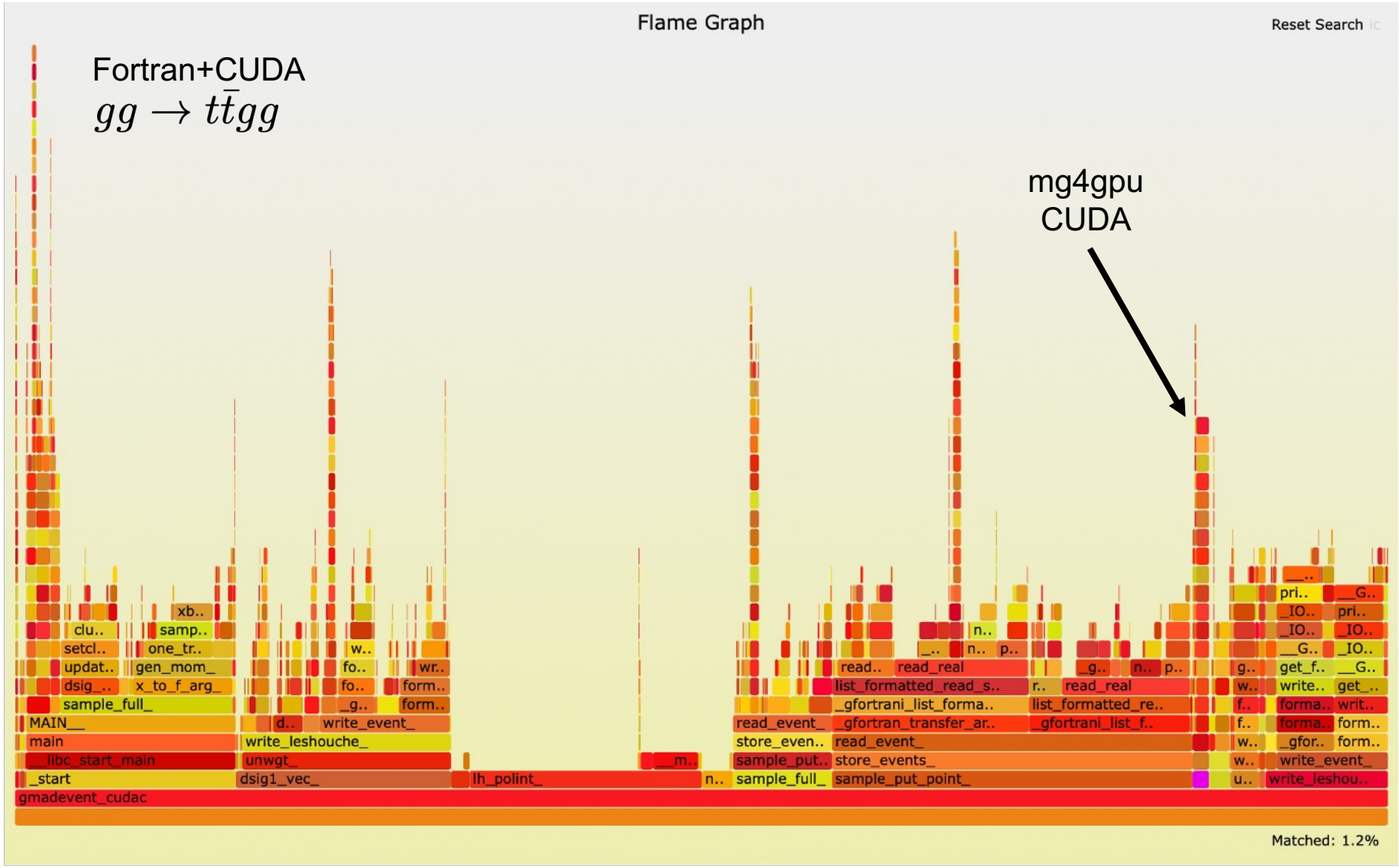
MadEvent: FORTRAN ←→ FORTRAN+CUDA

- Old MadEvent spent almost all time in matrix-element calculations



Madevent: FORTRAN ←→ FORTRAN+CUDA

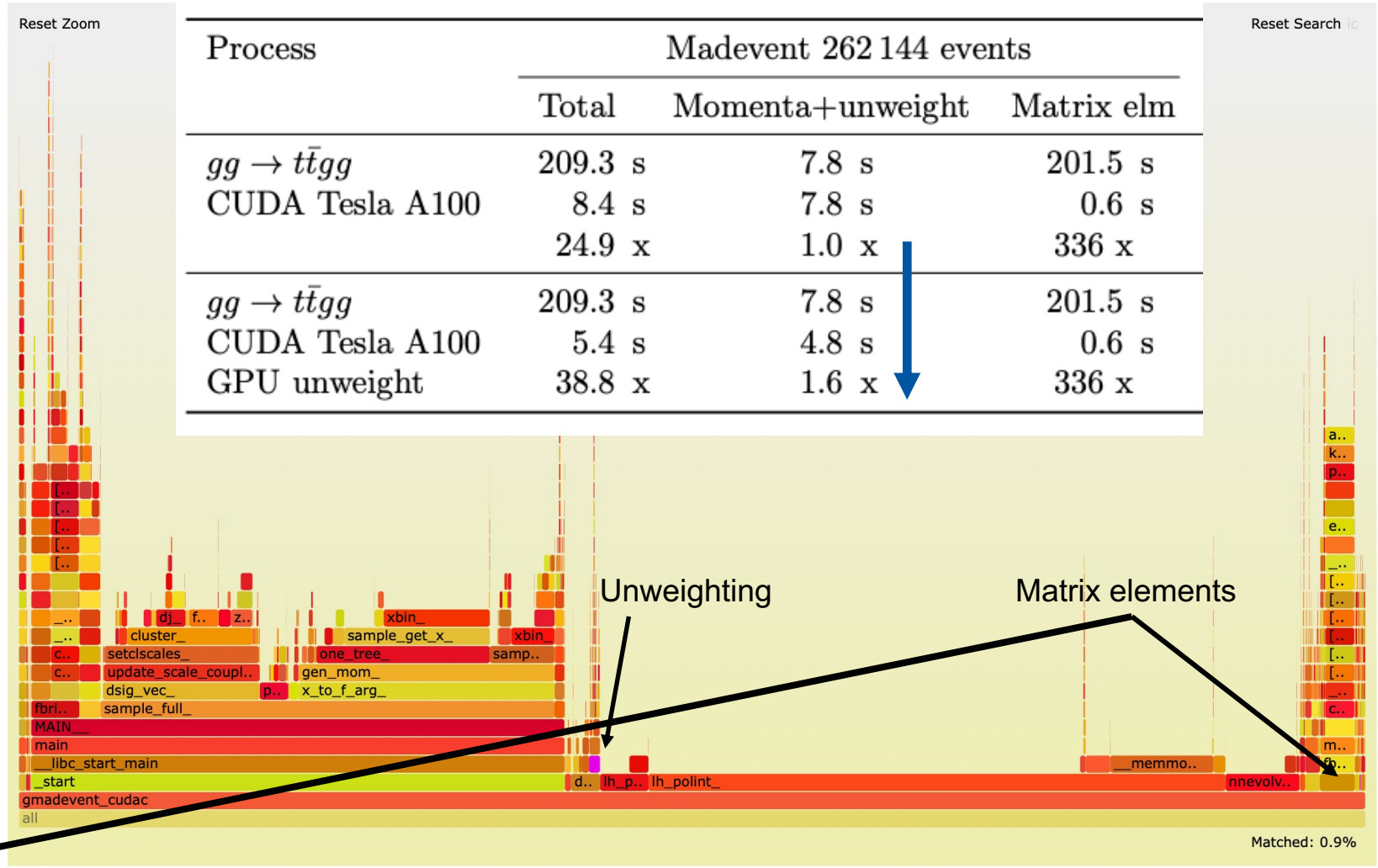
- Old MadEvent spent almost all time in matrix-element calculations
- With CUDA, the matrix elements are about 1% of the total run time
- Previously unimportant steps such as unweighting limit the speed up

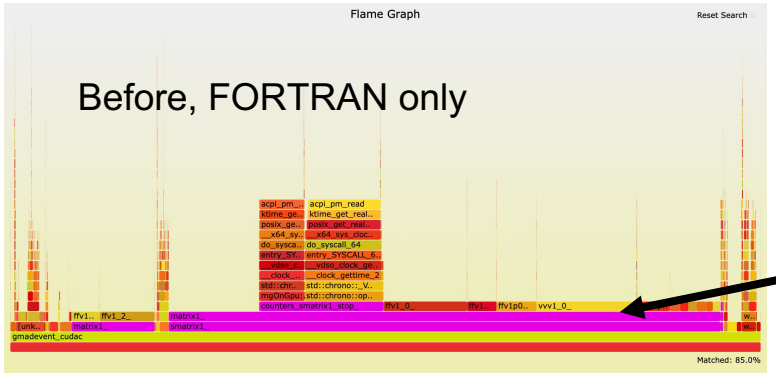


Improving the Madevent Side

- Improved handling of MLM
- GPU-assisted unweighting
  - Use GPU to for parallel weight computation
  - Helps FORTRAN unweighting routine to discard events faster
- More investigation on madevent side possible

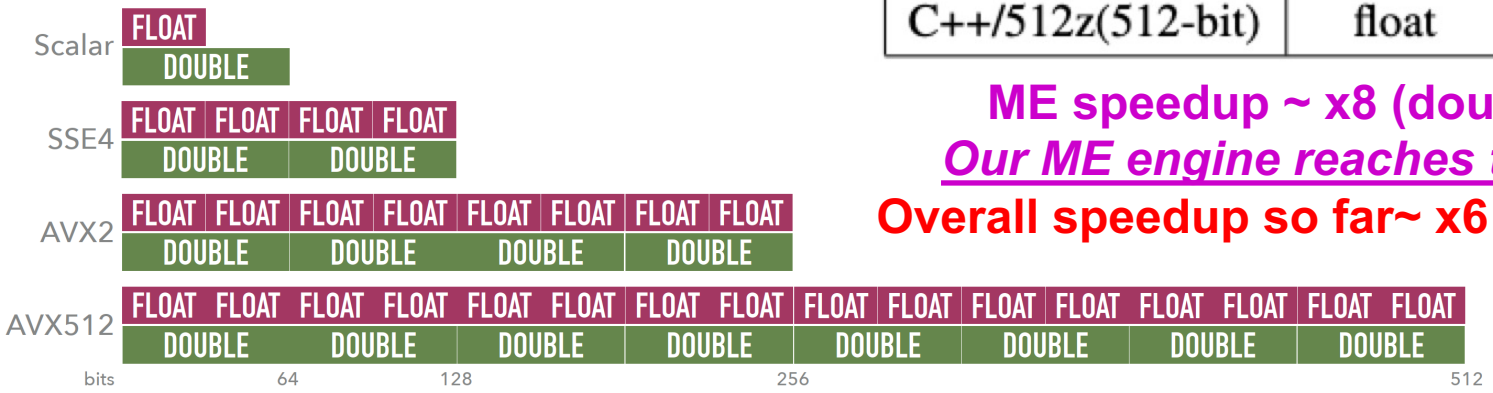
Process	Madevent 262 144 events		
	Total	Momenta+unweight	Matrix elm
$gg \to t\bar{t}gg$	209.3 s	7.8 s	201.5 s
CUDA Tesla A100	8.4 s	7.8 s	0.6 s
	24.9 x	1.0 x	336 x
$gg \to t\bar{t}gg$	209.3 s	7.8 s	201.5 s
CUDA Tesla A100	5.4 s	4.8 s	0.6 s
GPU unweight	38.8 x	1.6 x	336 x





MadEvent with vectorized C++ for gg→tt̄gg (on a single CPU core)

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



ME speedup ~ x8 (double) and x16 (float) over scalar Fortran

Our ME engine reaches the maximum theoretical SIMD speedup!

Overall speedup so far~ x6 (double) and x10 (float) over scalar Fortran (Amdahl's law)

Work-In-Progress, future plans, ideas...

- More performance for matrix elements in CUDA?
  - Smaller kernels: compiler optimizations, data placement, thread scheduling …
  - Split Feynman diagrams and color algebra
  - Tensore cores for color algebra computations (e.g. cublas)

- More performance for MadEvent
  - Parallel execution, heterogeneous workflow, vectorisation
  - Algorithmic optimizations (profiling)

- Functional improvements and longer-term plans
  - Support for NLO QCD processes
  - Event-by-event ME reweighting (and derivatives?) → See talk by Z. Wettersten
  - Test LHAPDF on GPU
  - Single-precision floating point?

The “alpha release”

- First ”gridpacks” generated for testing in LHC experiments
- Upstream madgraph4gpu GPU & SIMD code generation to mg5amcnlo
  - Including extra cross-checks that the LHE color IDs are those required for parton showers
- Test and fix any bugs in full pp collision simulation
  - Implement weak, SUSY, BSM processes
- Stay tuned! ☺
  - We will be happy to help your experiment in the integration!
  - NB: This will use a new Fortran version too (multi-event API). Needs statistical validation...

Conclusions

- The Matrix Element calculation in ANY ME event generator can be efficiently parallelized using SIMD or GPUs
- For Madgraph, we get:
  - SIMD: Speed ups of 6x for AVX512
  - GPU: 20x to 80x depending on the process using Tesla A100

- Our reengineering of MG5aMC is a functional alpha release for LO QCD / EM processes, but weak interactions need more work
- Please get in touch if you want to test the gridpacks

- Using SYCL, we get similar performances to CUDA and we may also run on AMD or Intel GPUs
- Tests on supercomputer testbeds show good scaling

Acknowledgements

- This research used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility operated under contract DE-AC02-06CH11357, and the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.

- We gratefully acknowledge the use (under PRACE proposal PRACE-DEV-2022D01-022) of the JUWELS supercomputer and other computing resources provided and operated by the Jülich Supercomputing Centre at Forschungszentrum Jülich.

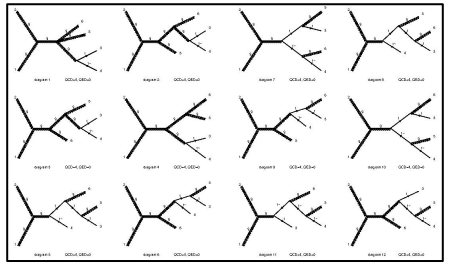
- We gratefully acknowledge the use (under ISCRA-C project MG5A100) of computing resources provided and operated by CINECA.

- We thank the organizers and our mentors at the GPU Hackathon in CSCS Lugano in September 2022.

BACKUP

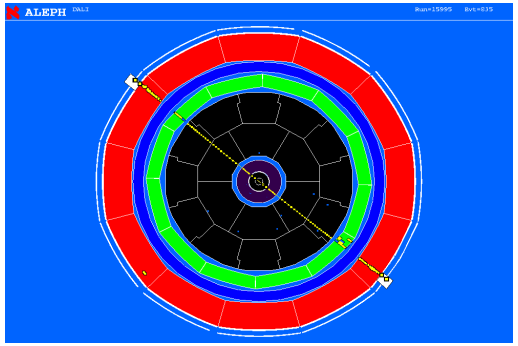
Code Generator and Transformation to CUDA

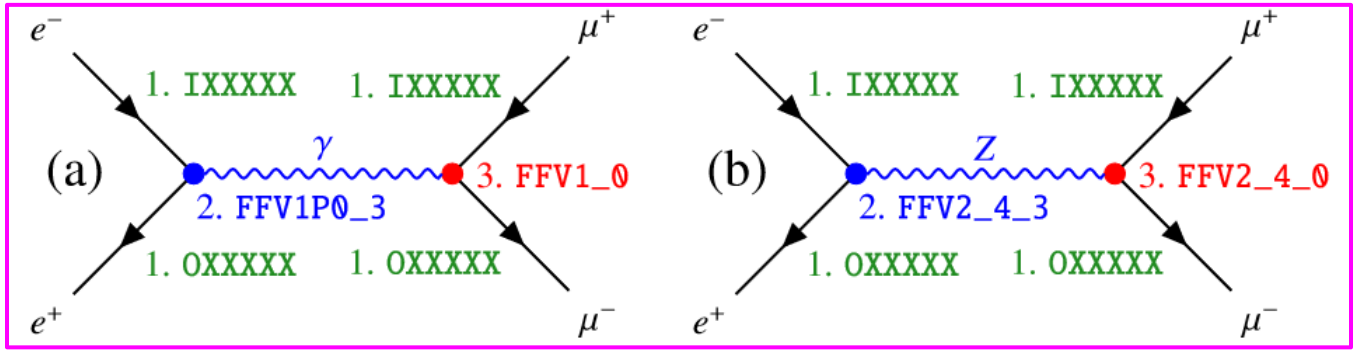
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Initially Fortran (default), C++, or Python
  - More complicated processes produce larger code

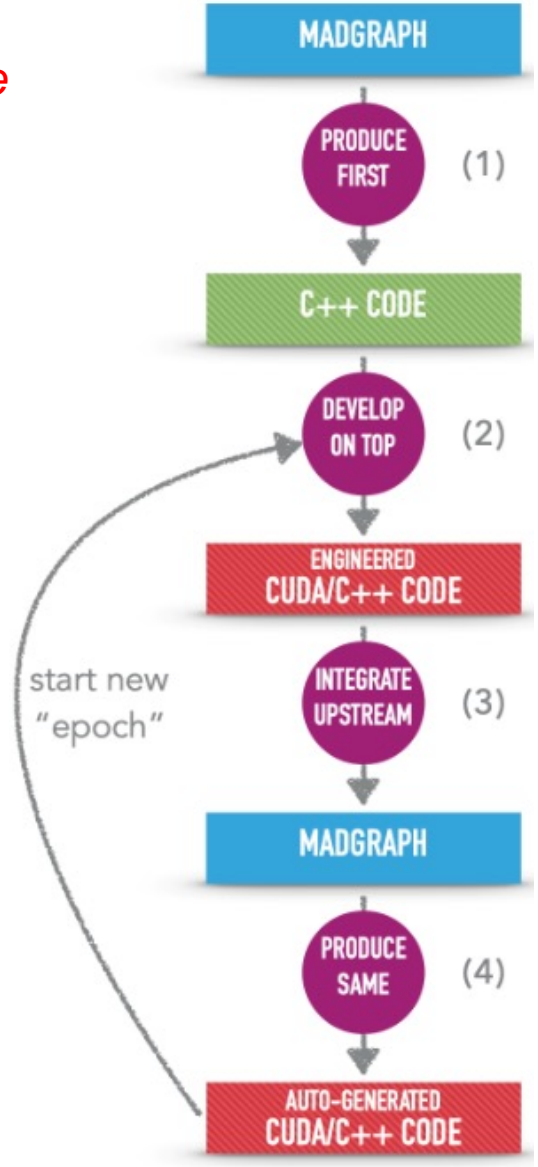


Process	*LOC*	*functions*	*function calls*
$e^+e^- \to \mu^+\mu^-$	776	8	16
$gg \to t\bar{t}$	839	10	22
$gg \to t\bar{t}g$	1082	36	106
$gg \to t\bar{t}gg$	1985	222	786

- Strategy: 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 code generator (Python)
  - (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*







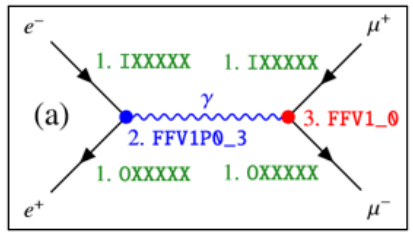
Helicity amplitudes – same code in CUDA and in vectorized C++

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 γμ⁺μ⁻ 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 for vectorized SYCL!

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

Reweighting

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

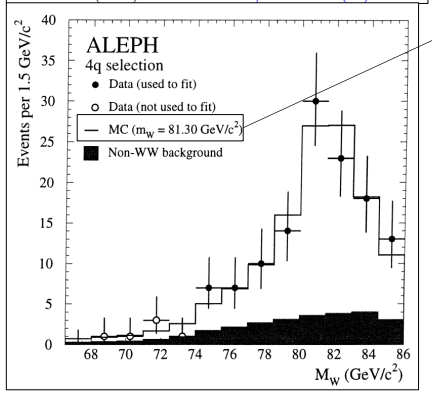
1. Generate signal sample at θ_{ref}, with $w_i(\theta_{ref})=1$
(By definition, background does not depend on θ)

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***

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

ALEPH Collaboration, *Measurement of the W mass by direct reconstruction in e^+e^- collisions at 172 GeV*, Phys. Lett. B 422 (1998) 384. doi:10.1016/S0370-2693(98)00062-8



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

Old technique, renewed interest!

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

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"**

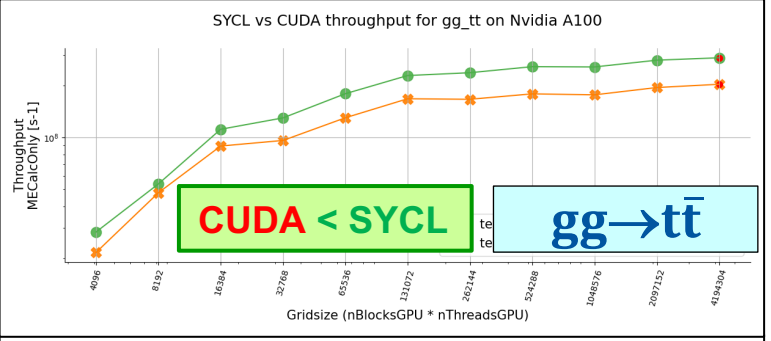
- *Helicity filtering* – at initialization time, compute the allowed combinations of particle helicities
  - This is computed in CUDA/C++ using the same criteria as in Fortran
- *"Multi-channel"* – single-diagram enhancement of ME output
  - This is the specificity of the MadEvent sampling algorithm (Maltoni Stelzer 2003)

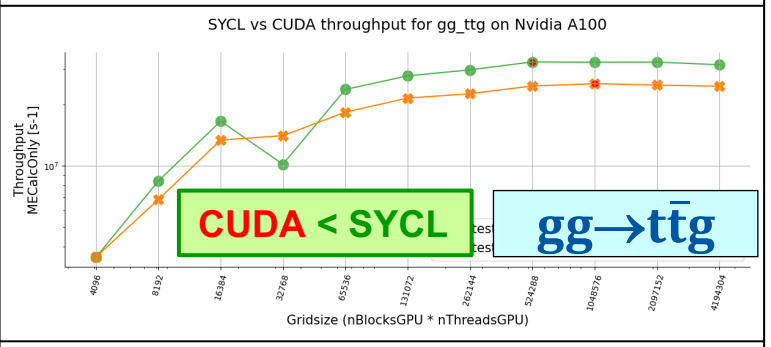
$$f_i = \frac{|A_i|^2}{\sum_i |A_i|^2} |A_{\text{tot}}|^2$$

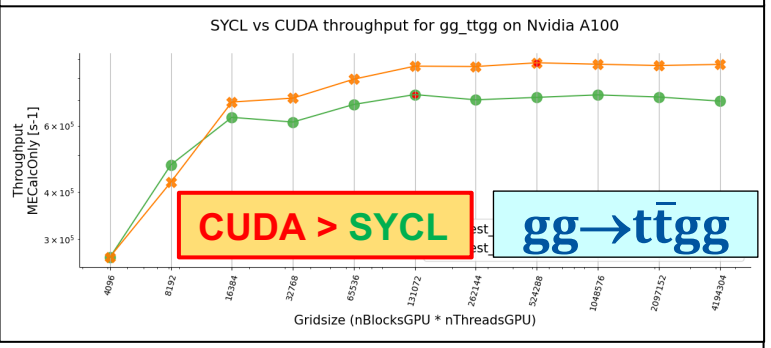
- Event-by-event *running QCD coupling constants* $\alpha_s(Q^2)$
  - The scale is currently computed in Fortran from momenta and passed to the CUDA/C++ for each event
  - No support yet for weak interaction / BSM / SUSY
- Event-by-event *choice of helicity and color* in LHE files
  - Pass two additional 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++

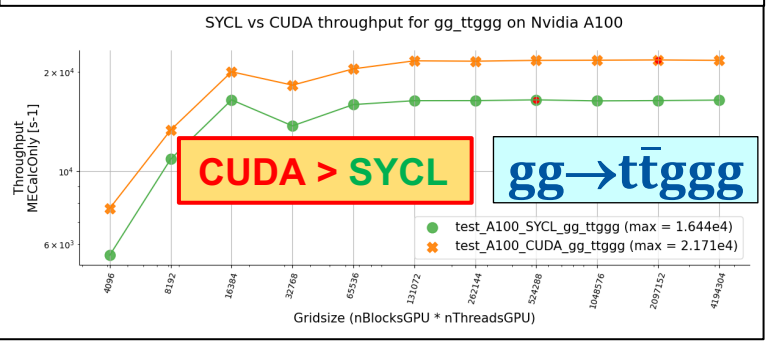
CUDA vs SYCL on NVidia A100

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







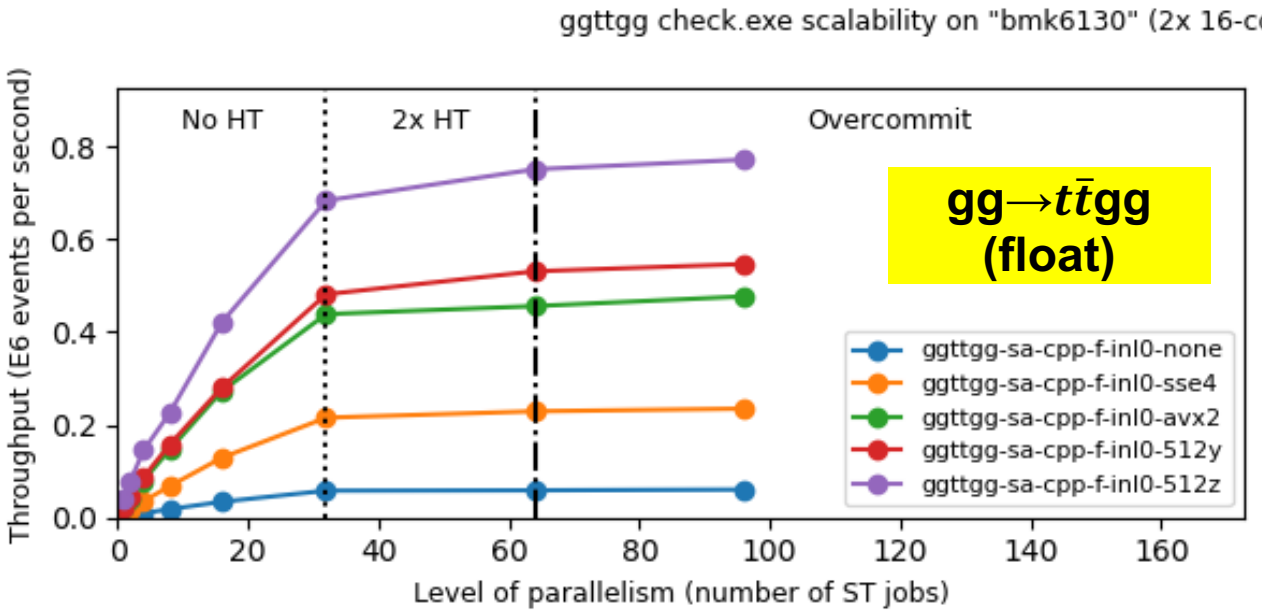


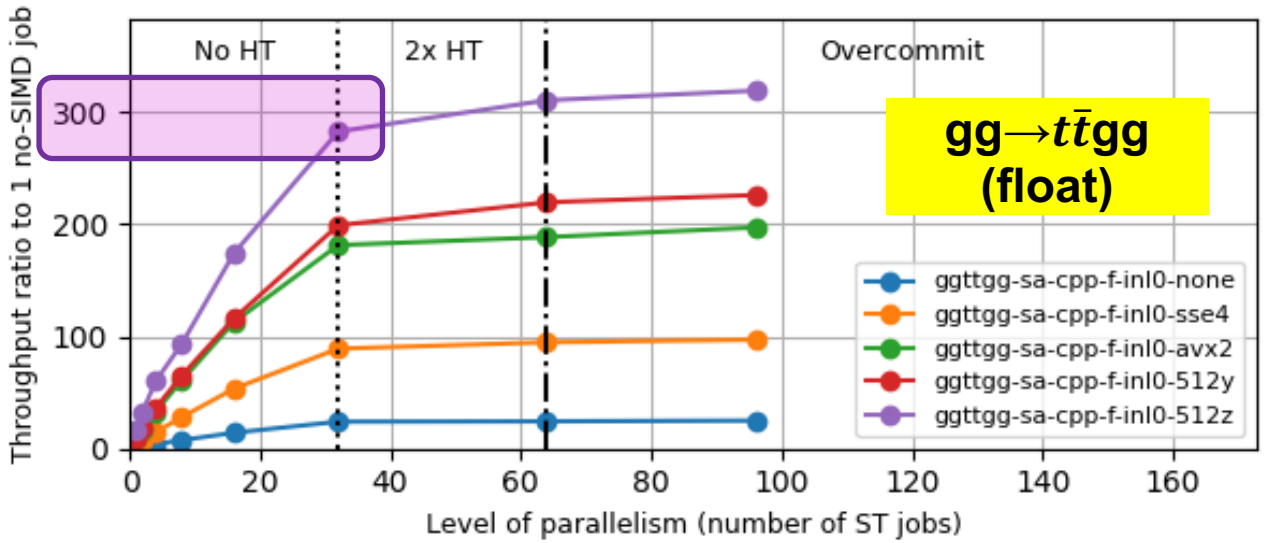
- SYCL and CUDA implementations have ~similar performances but
  - SYCL seems better for less complex processes
  - *CUDA seems better for more complex processes*

- 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!

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

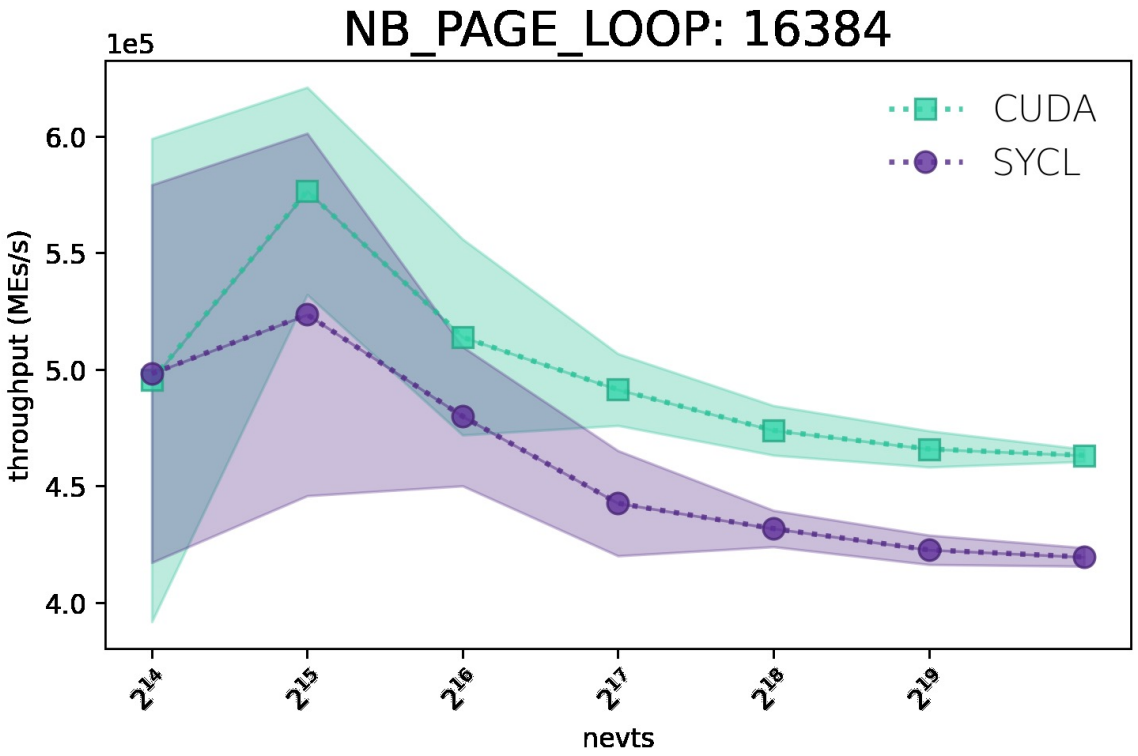




- Most previous results were single-thread peak throughputs

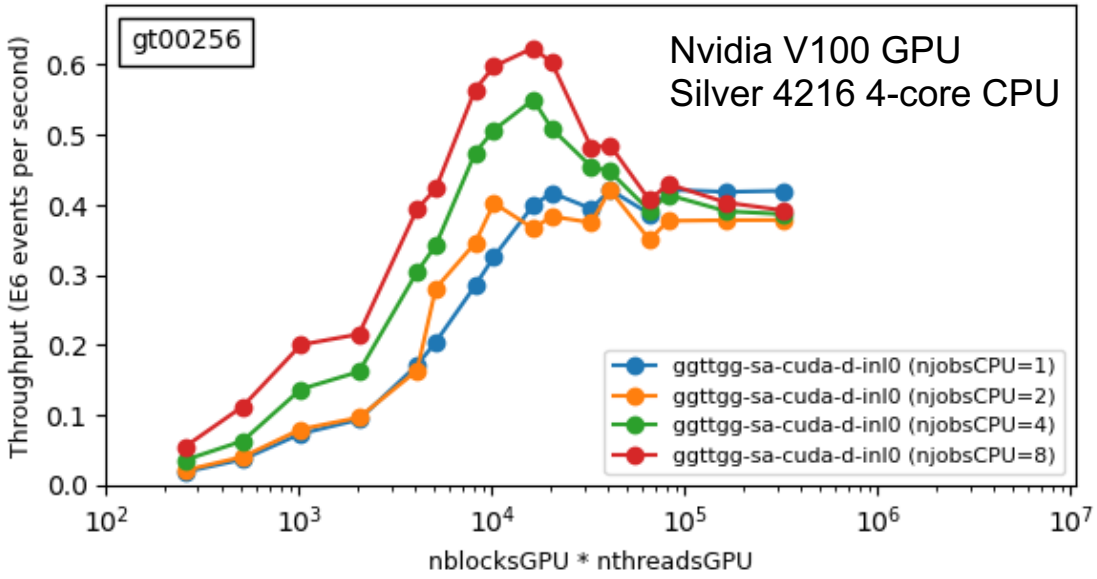
- **Large SIMD speedups are also confirmed when all CPU cores are used**
  - AVX512/zmm speedup of x16 over no-SIMD for a single core slightly decreases to ~x12 on a full node (clock slowdown?)
  - *Overall speedup on 32 physical cores (over no-SIMD on 1 core) is around 280 (maximum would be 16x32=512)*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

MEs in MadEvent: CUDA vs SYCL for $gg \to t\bar{t}gg$



- ME throughput only - SYCL comparable to CUDA but somewhat lower

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!...)

MadEvent/C++ for gg→tt̄ggg (on a single core)

		madevent			standalone
$gg \to t\bar{t}gg$	MEs precision	$t_{\rm TOT} = t_{\rm Mad} + t_{\rm MEs}$ [sec]	$N_{\rm events}/t_{\rm TOT}$ [events/sec]	$N_{\rm events}/t_{\rm MEs}$ [MEs/sec]	
Fortran(scalar)	double	813.2 = 3.7 + 809.6	1.01E2 (=1.0)	1.01E2 (=1.0)	—
C++/none(scalar)	double	986.0 = 4.3 + 981.7	8.31E1 (x0.8)	8.35E1 (x0.8)	9.82E1
C++/sse4(128-bit)	double	514.7 = 4.2 + 510.5	1.59E2 (x1.6)	1.61E2 (x1.6)	1.95E2
C++/avx2(256-bit)	double	231.6 = 4.0 + 227.6	3.54E2 (x3.5)	3.60E2 (x3.6)	4.41E2
C++/512y(256-bit)	double	208.6 = 3.9 + 204.8	3.93E2 (x3.9)	4.00E2 (x4.0)	4.95E2
C++/512z(512-bit)	double	124.6 = 4.0 + 120.6	6.58E2 (x6.5)	6.79E2 (x6.7)	8.65E2
C++/none(scalar)	float	936.1 = 4.3 + 931.8	8.75E1 (x0.9)	8.79E1 (x0.9)	1.02E2
C++/sse4(128-bit)	float	228.9 = 3.9 + 225.0	3.58E2 (x3.6)	3.64E2 (x3.6)	4.30E2
C++/avx2(256-bit)	float	114.1 = 3.8 + 110.4	7.18E2 (x7.2)	7.43E2 (x7.4)	9.06E2
C++/512y(256-bit)	float	104.5 = 3.8 + 100.7	7.84E2 (x7.9)	8.14E2 (x8.1)	1.00E3
C++/512z(512-bit)	float	61.8 = 3.8 + 58.0	1.33E3 (x13.3)	1.41E3 (x14.1)	1.77E3
C++/none(scalar)	mixed	986.0 = 4.3 + 981.6	8.31E1 (x0.8)	8.35E1 (x0.8)	9.98E1
C++/sse4(128-bit)	mixed	500.4 = 3.9 + 496.5	1.64E2 (x1.6)	1.65E2 (x1.6)	2.00E2
C++/avx2(256-bit)	mixed	220.5 = 3.8 + 216.7	3.72E2 (x3.7)	3.78E2 (x3.8)	4.55E2
C++/512y(256-bit)	mixed	195.6 = 3.7 + 191.8	4.19E2 (x4.2)	4.27E2 (x4.3)	5.21E2
C++/512z(512-bit)	mixed	118.5 = 3.8 + 114.7	6.92E2 (x6.9)	7.15E2 (x7.2)	8.97E2

- Lower overhead of scalar MadEvent in gg→tt̄ggg than in gg→tt̄gg : higher **overall throughput speedup x13!**
- Mixed floating-point precision (single precision color algebra) is 5-10% better than double

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.

MORE BACKUP SLIDES

Matrix element integration in MadEvent: detailed results (CPU)

Intel Gold 6148 CPU (Juwels Cluster HPC)

	mad (81952 MEs)	mad	mad	sa/brdg
ggttgg	[sec] tot = mad + MEs	[TOT/sec]	[MEs/sec]	[MEs/sec]
FORTRAN	41.82 = 3.23 + 38.60	1.96e+03 (= 1.0)	2.12e+03 (= 1.0)	---
CPP/none	47.78 = 3.56 + 44.22	1.72e+03 (x 0.9)	1.85e+03 (x 0.9)	1.90e+03
CPP/sse4	23.04 = 2.97 + 20.07	3.56e+03 (x 1.8)	4.08e+03 (x 1.9)	4.05e+03
CPP/avx2	12.19 = 2.88 + 9.32	6.72e+03 (x 3.4)	8.80e+03 (x 4.2)	9.24e+03
CPP/512y	11.57 = 2.86 + 8.71	7.08e+03 (x 3.6)	9.41e+03 (x 4.4)	1.01e+04
CPP/512z	8.26 = 2.88 + 5.38	9.92e+03 (x 5.1)	1.52e+04 (x 7.2)	1.60e+04

TIME Total = MadEvent (scalar) + MEs (parallel)

TIME MadEvent (scalar)

TIME MEs (parallel)

THROUGHPUT MadEvent + MEs (within madevent)

THROUGHPUT MEs (within madevent)

THROUGHPUT MEs (within standalone test application)

Matrix element integration in MadEvent: detailed results (GPU)

NVidia V100 GPU + Intel Silver 4216 CPU (CERN)

	mad	mad	mad	sa/brdg
ggttggg	[sec] tot = mad + MEs	[TOT/sec]	[MEs/sec]	[MEs/sec]
nevt/grid	8192	8192	8192	8192
nevt total	90112	90112	90112	256*32*1
FORTRAN	1286.09 = 62.74 + 1223.35	7.01e+01 (= 1.0)	7.37e+01 (= 1.0)	---
CUDA/8192	77.06 = 64.87 + 12.19	1.17e+03 (x16.7)	7.39e+03 (x100.)	7.48e+03
nevt/grid				16384
nevt total				512*32*1
CUDA/max				9.33e+03

8k events per GPU grid

TIME MadEvent (scalar)
1. REDUCE THIS TO INCREASE SPEEDUP

ggttgg GPU MEs speedup is lower than eemumu (higher register pressure)
3. SMALLER GPU KERNELS TO INCREASE SPEEDUP

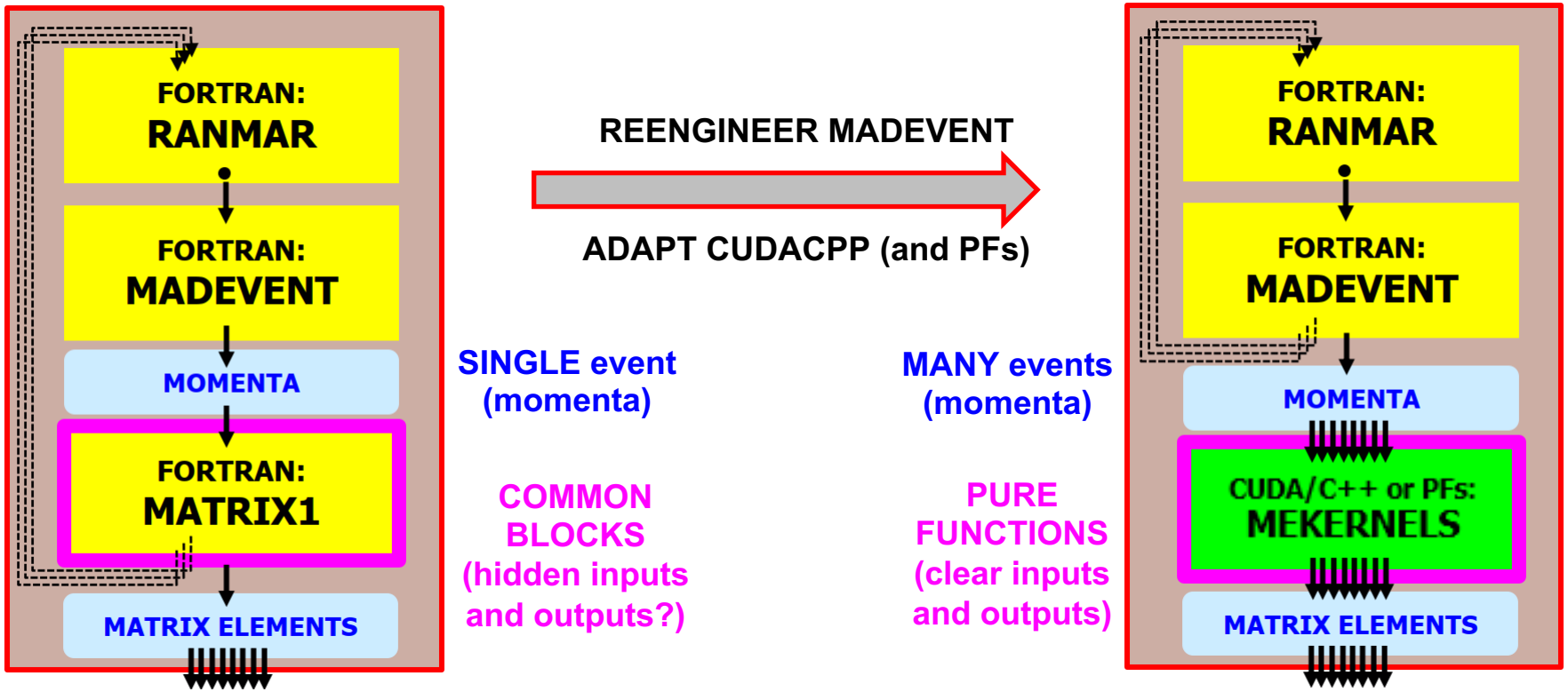
16k events per GPU grid
2. INCREASE GPU GRIDS (REDUCE CPU MEMORY) TO INCREASE SPEEDUP

Matrix element integration in MadEvent

Replace Fortran MEs by cudacpp (or PFs) MEs in Madevent *(keep the same user interface!)*

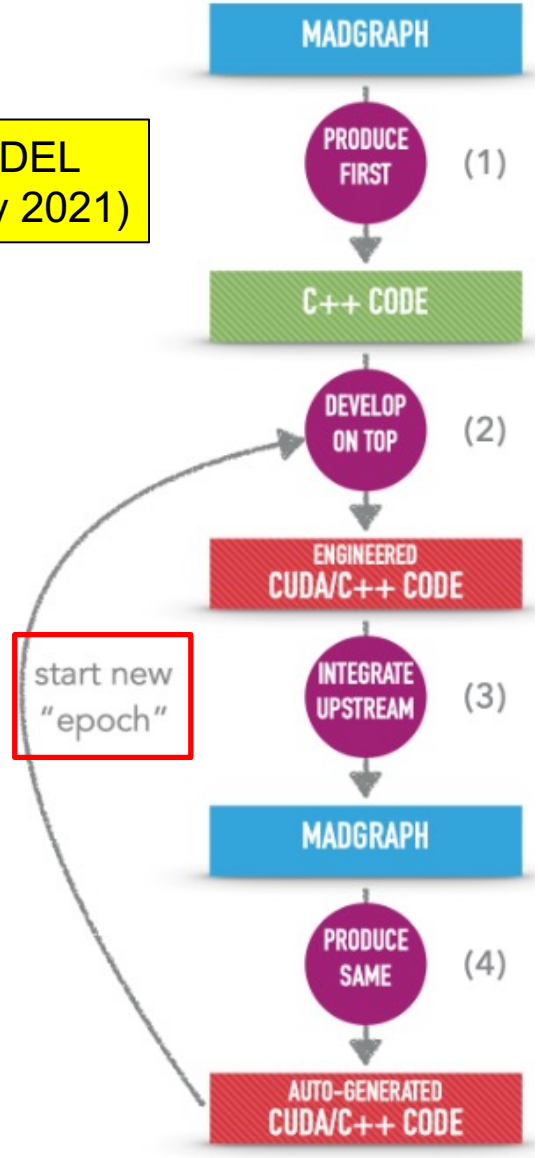
Linking Fortran and C++ has been easy. As expected, the two main issues have been, instead:

- 1. Moving Madevent from single-event to many-event (functional reengineering of the algorithm)
  - Now also an active area of performance optimizations (next slides: GPU grid and CPU RAM; CPU time and Amdahl...)
- 2. Debugging functional issues caused by hidden inputs and outputs, e.g. coming from Fortran common blocks



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

OLD MODEL
(2020- early 2021)



Now using upstream MG5AMC from
https://github.com/mg5amcnlo !

NEW MODEL
(since end 2021)

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

(3) re-generate

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

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

Portability Frameworks (PFs)

(2) Second line of development: MEs on PFs

- PFs allow writing algorithms once and running on many architectures with some hardware-specific optimizations
- CUDA code can only run on NVidia GPUs, while Kokkos, Alpaka, and Sycl[Intel] codes can run on most hardware
- In “cudacpp”, #ifdef directives separate code branches for GPU and CPU code during compilation (but these are very few: only kernel launching and memory access, not MEs)
- With PFs, the algorithm is typically the same, but the compilation occurs once per architecture type
- PFs often use templating to handle data types and hardware configuration and function lambdas or pointers for passing kernels (the cudacpp plugin has many of these, too)
- PFs still require user to think about “host” vs “device”

“cudacpp” example of compiler directives

```
540 #ifdef __CUDACC__
541 #ifndef MGONGPU_NSIGHT_DEBUG
542     gProc::sigmaKin<<<gpublocks, gputhreads>>>(devMomenta.get(), devMEs.get()
543 #else
544     gProc::sigmaKin<<<gpublocks, gputhreads, ntpbMAX*sizeof(float)>>>(devMome
545 #endif
546     checkCuda( cudaPeekAtLastError() );
547     checkCuda( cudaDeviceSynchronize() );
548 #else
549     Proc::sigmaKin(hstMomenta.get(), hstMEs.get(), nevt);
550 #endif
```

For GPU

For CPU

Kokkos example of Templating & lambda

```
324 {
325   using member_type = typename Kokkos::TeamPolicy<Kokkos::DefaultExecut
326   Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace> policy( league_size
327   Kokkos::parallel_for(__func__,policy,
328   KOKKOS_LAMBDA(member_type team_member){
```

Kokkos example of Memory Management

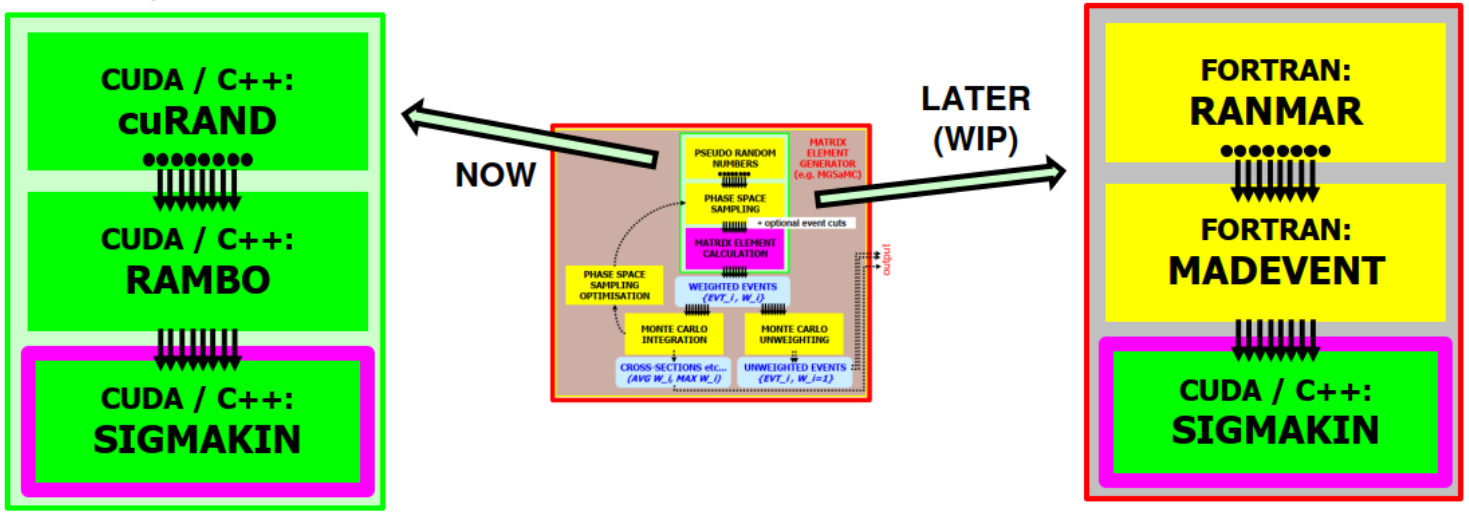
```
262     Kokkos::View<fptype***,Kokkos::DefaultExecutionSpace> devMomenta(Kokkos::ViewAllocateWithoutInitializing("devMomenta"),nevt,npar,np4);
263     auto hstMomenta = Kokkos::create_mirror_view(devMomenta);
```

EVEN MORE BACKUP SLIDES

Standalone CUDA/C++ application VS. MadEvent integration

- Our main focus: the ME calculation in CUDA/C++ (sigmakin kernel/function)
  - Design approach: *single source code for CUDA and C++* (>90% common code + #ifdef's)
- Our workhorse: *a simplified CUDA/C++ toy framework to feed events to the ME kernel*
  - All 3 main components on the GPU: random (cuRAND), sampling (RAMBO), ME (sigmakin)
  - Fast, same results in GPU/CPU, but not good for production (RAMBO algorithm is inefficient)
  - *The results I present in this talk come from this framework*

cuRAND: identical random number sequences on host (CPU) and device (GPU), allowing CUDA/C++ bitwise comparisons



- Our WIP: *we plan to inject CUDA/C++ ME kernel into MadEvent/gridpack framework*
  - Fastest way to production – easier than rewriting MadEvent in CUDA/C++
  - Validated code/infrastructure, same user interface – discussed with experiments at HSF WG

Event-level parallelism in practice – coding and #events

- Easier to code for GPU SIMT than for CPU SIMD: *CUDA code was faster to prototype*
- CUDA (GPU) implementation
  - For SIMT, event loop is "orthogonal": one thread = one event *(GPU thread ID ↔ event ID)*
  - For SIMT, SOA memory layouts are beneficial (coalesced access), but not strictly essential
- C++ (CPU) implementation
  - For SIMD, event loop must be the innermost loop (e.g. invert helicity and event loops)
  - For SIMD, SOA memory layouts in the computational kernel are essential

- To be efficient, *CUDA needs O(10k)-O(1M) events in parallel* – much more than C++!
  - CUDA: lockstep within each warp (32 threads) + many warps in parallel to fill the GPU
  - C++: lockstep within a vector register (2-8 doubles) + multi-threading or multi-processing