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

Taylor Childers
Walter Hopkins
Nathan Nichols
Argonne NATIONAL LABORATORY

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

Olivier Mattelaer
UCL Université catholique de Louvain

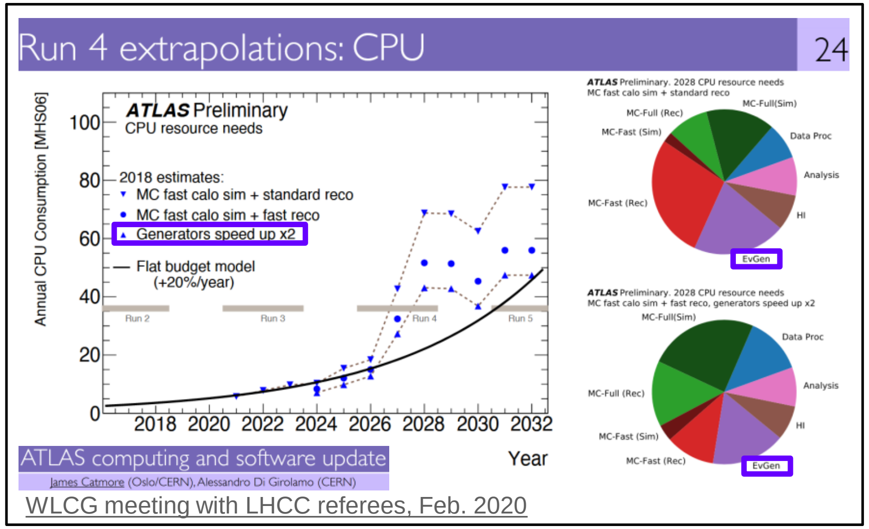
Carl Vuosalo
WISCONSIN UNIVERSITY OF WISCONSIN–MADISON

ACAT, Bari, 24th October 2022

https://indico.cern.ch/event/1106990/contributions/4997226

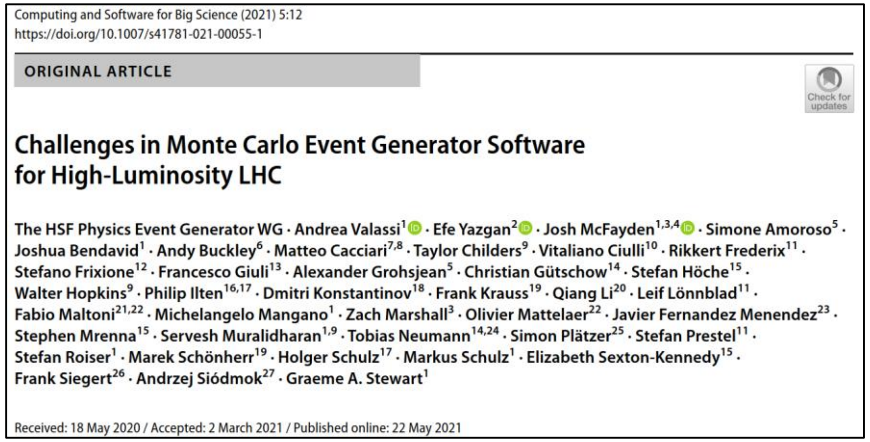
Motivation: Monte Carlo Event Generators in WLCG computing

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



WLCG meeting with LHCC referees, Feb. 2020

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

Check for updates

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

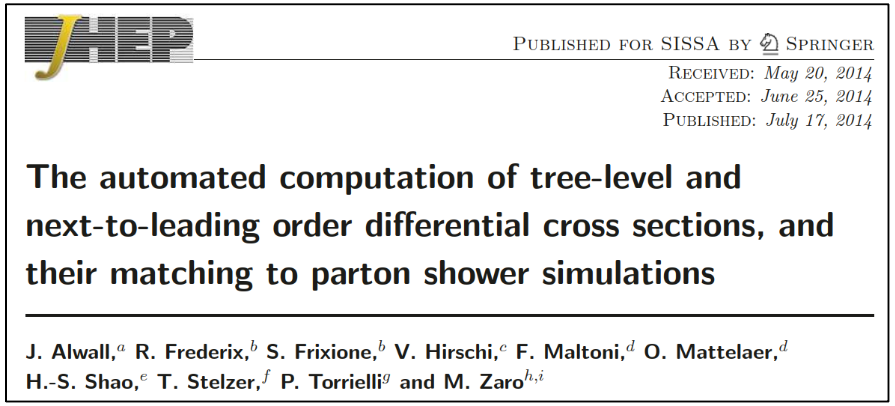
The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · 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, the essential 1st step in simulation, use ~5%-20% of ATLAS/CMS WLCG CPU budget
  - Many ways to speed them up – see the HEP Software Foundation (HSF) Generator WG review paper
  - *MC generators are ideal candidates to exploit data parallelism in GPUs (SIMT) and in vector CPUs (SIMD)*

Madgraph5_aMC@NLO (MG5aMC)

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



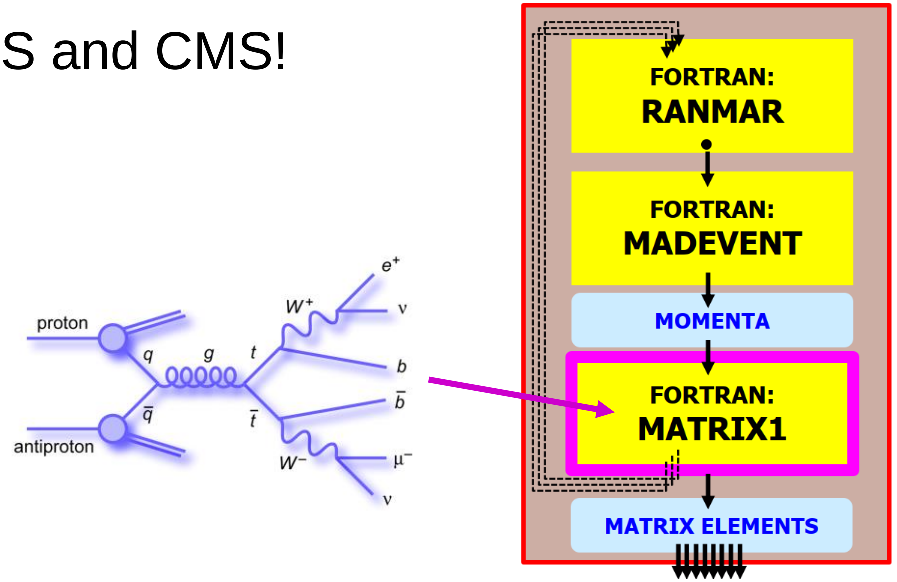
PUBLISHED FOR SISSA BY SPRINGER

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

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

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

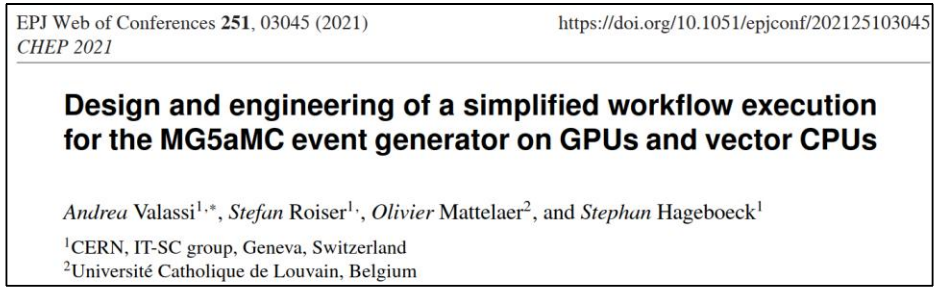
https://doi.org/10.1007/JHEP07(2014)079



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

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
  - Extensive details may be found in the vCHEP2021 and ICHEP2022 conference proceedings



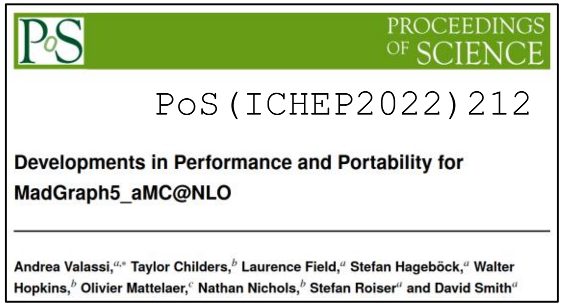
EPJ Web of Conferences **251**, 03045 (2021) https://doi.org/10.1051/epjconf/202125103045
CHEP 2021

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]

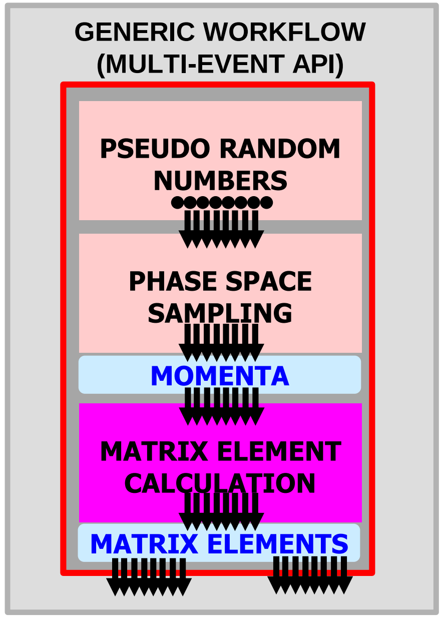
https://doi.org/10.22323/1.414.0212

- 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), later addition supporting many GPUs (and CPUs too)

- Two types of executables: initially standalone applications, then MadEvent-integrated applications

- *In this ACAT2022 presentation I give an overview, and a few new results since ICHEP in July 2022*

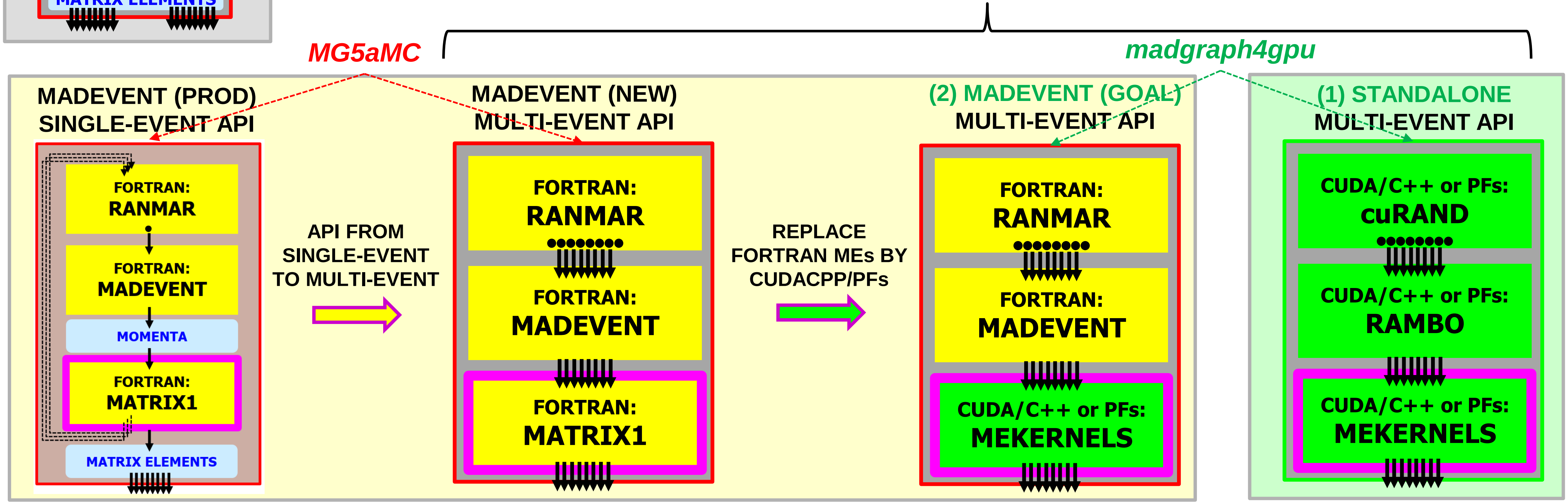
MG5aMC: old and new architecture designs



(1) First we developed the new (CUDACPP/PF) ME engines within standalone applications

(2) Then we modified the existing all-Fortran MadEvent into a multi-event framework and we injected the new (CUDACPP/PF) MEs into it

In the following I will give performance results from these three applications

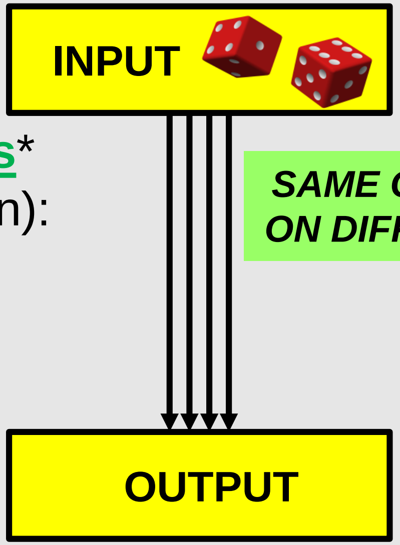


ANY MC event generator is a great fit for GPUs and vector CPUs!

- *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*:

NB: MULTI-EVENT API

MC SAMPLING



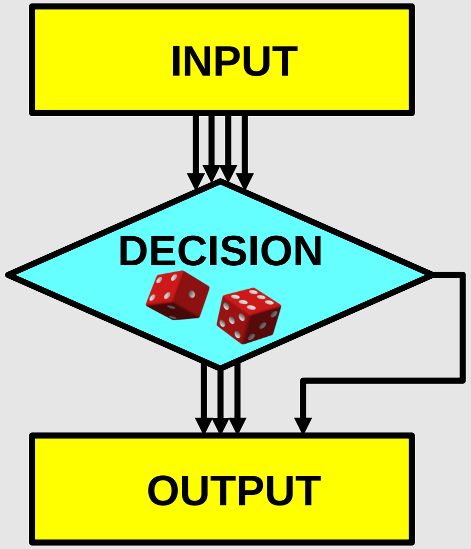
SAME CALCULATION ON DIFFERENT DATA!

ME event generators*
(*before* ME calculation):
- MC integration (cross sections)
- MC generation (event samples)

Lockstep processing
Good for SIMT/SIMD

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

MC DECISIONS



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

DIFFERENT CALCULATIONS ON DIFFERENT DATA!

Stochastic branching
Bad for SIMT/SIMD

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

CUDACPP MEs

- 95% common code + a few #ifdef's for CUDA vs C++
- Designed for NVidia GPUs (so far) and vector CPUs
- Designed for vector CPUs (large SIMD speedups)
- Full feature support, e.g. tensor cores, streams, graphs

PF MEs

- Write code once for many CPU/GPU vendors
- Support NVidia, AMD and Intel GPUs out-of-the-box
- No explicit design for SIMD on CPUs (speedups?)
- Limited feature support

CUDACPP vs. Portability Frameworks – recap

- CUDAPP (our initial implementation) is still where we add new features first
- SYCL and KOKKOS are now almost at the same level
- ALPAKA is no longer maintained

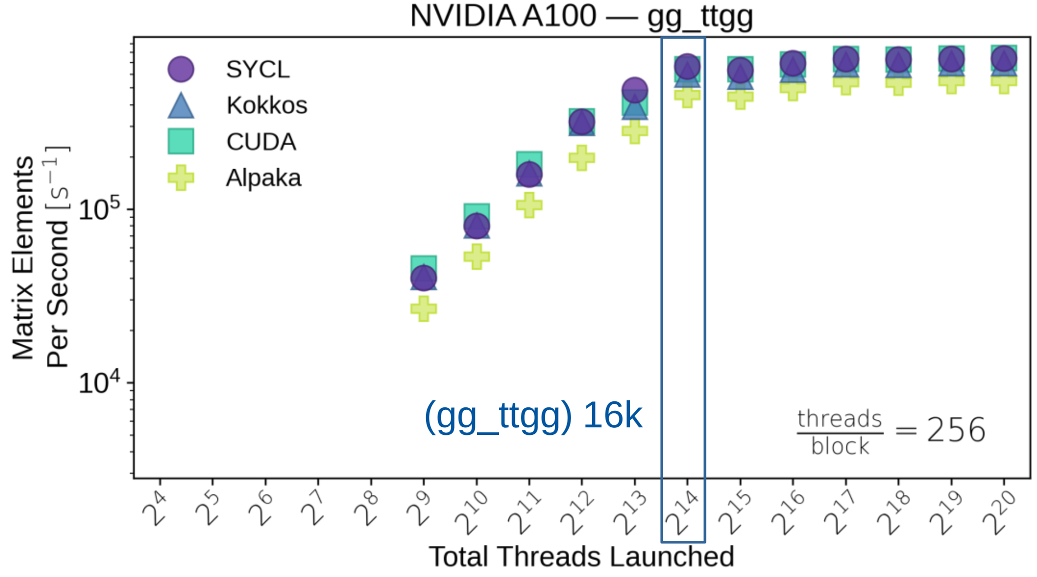
Backend	ME code generation	Standalone application	Actively maintained	MadEvent application	Latest dev code base
CUDACPP	✓	✓	✓	✓	✓
SYCL	✓	✓	✓	✓	~ ✓
KOKKOS	✓	✓	✓	WIP	~ ✓
ALPAKA (CUPLA)	✓	✓	✗	✗	✗

For the moment: we plan to continue development in parallel using both approaches, CUDACPP and PFs

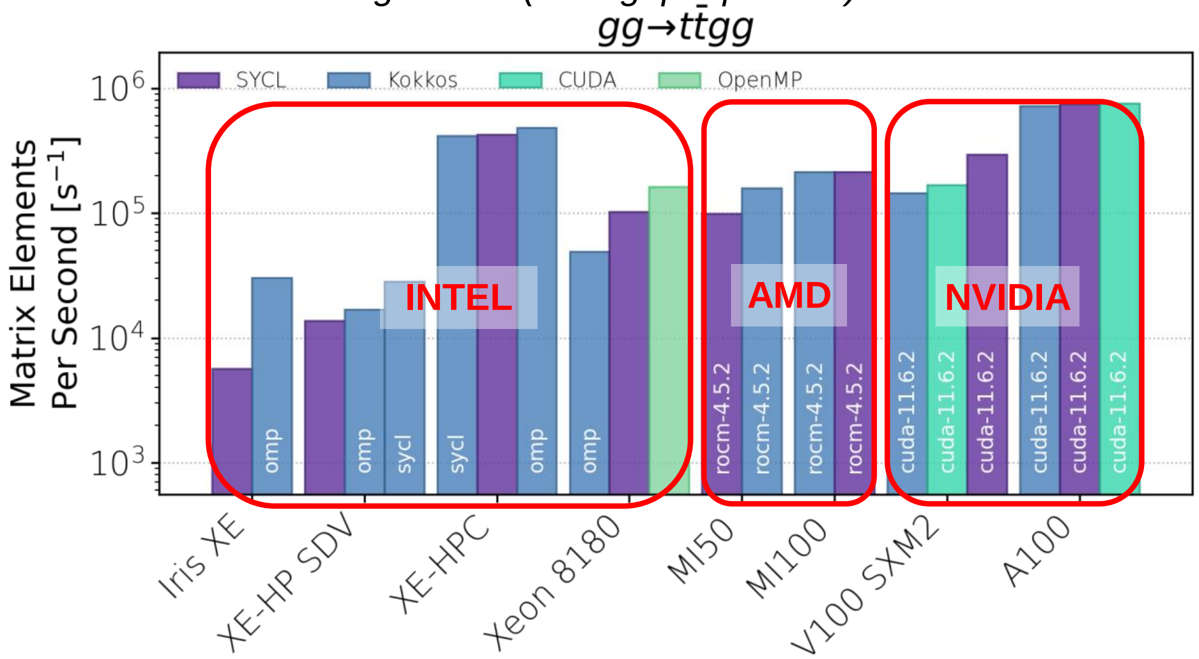
Two goals: not only production releases, but also *aim to provide useful feedback to HEP about usability of PFs*

CUDACPP vs PFs - GPU ME throughputs (standalone application)

Variable GPU-grid size (throughput scan)



Fixed GPU-grid size (throughput plateau)



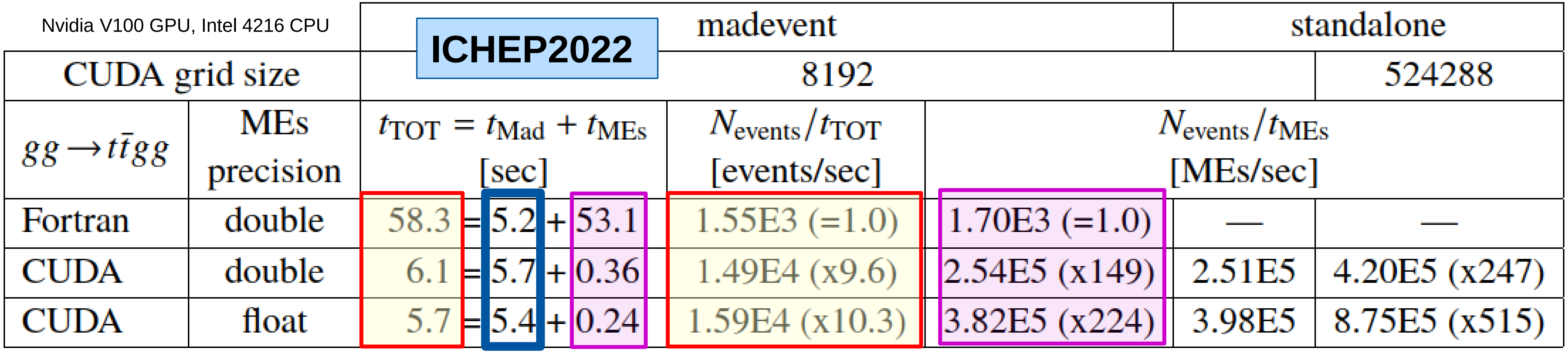
- The performances of SYCL and Kokkos are comparable to direct CUDA

- 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

MadEvent throughputs with CUDA for $gg\to t\bar{t}gg$ (ICHEP2022)



ICHEP2022

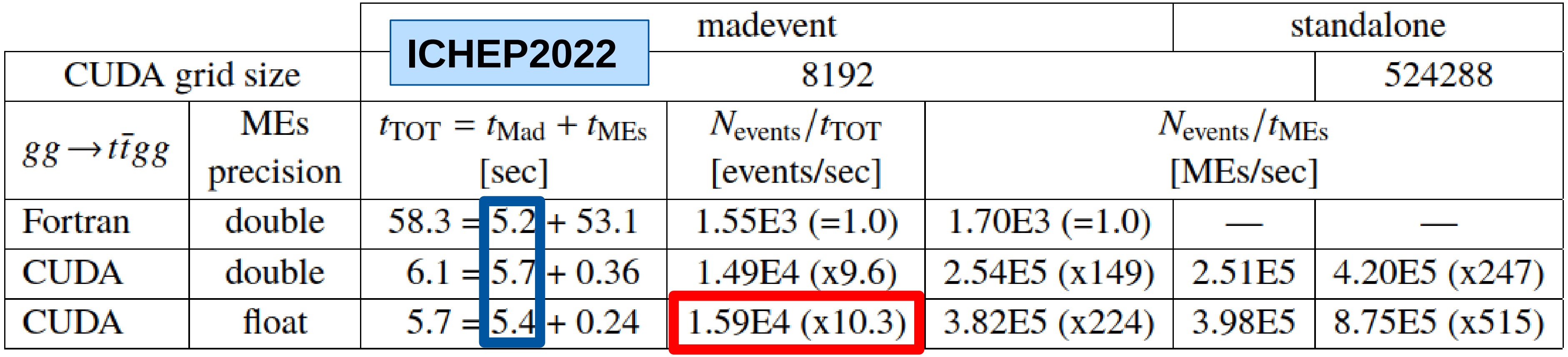
Nvidia V100 GPU, Intel 4216 CPU		madevent			standalone	
CUDA grid size		8192				524288
$gg\to t\bar{t}gg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]		
Fortran	double	58.3 = 5.2 + 53.1	1.55E3 (=1.0)	1.70E3 (=1.0)	—	—
CUDA	double	6.1 = 5.7 + 0.36	1.49E4 (x9.6)	2.54E5 (x149)	2.51E5	4.20E5 (x247)
CUDA	float	5.7 = 5.4 + 0.24	1.59E4 (x10.3)	3.82E5 (x224)	3.98E5	8.75E5 (x515)

- ME calculation alone: accelerated by x150/x225 (double/float) on GPU with respect to Fortran on CPU
  - (With a GPU grid size of 8k, limited by MadEvent RAM - and could reach x250/x500 with a larger grid size of 524k)

- Overall workflow speedup is only x10 (double/float) - maximum achievable as scalar part was 10% **(Amdahl's law)**

- **Must reduce the scalar MadEvent Fortran overhead (random numbers, sampling algo, I/O, MLM merging...)**

MadEvent/CUDA for gg→tt̄gg (improved at ACAT2022)

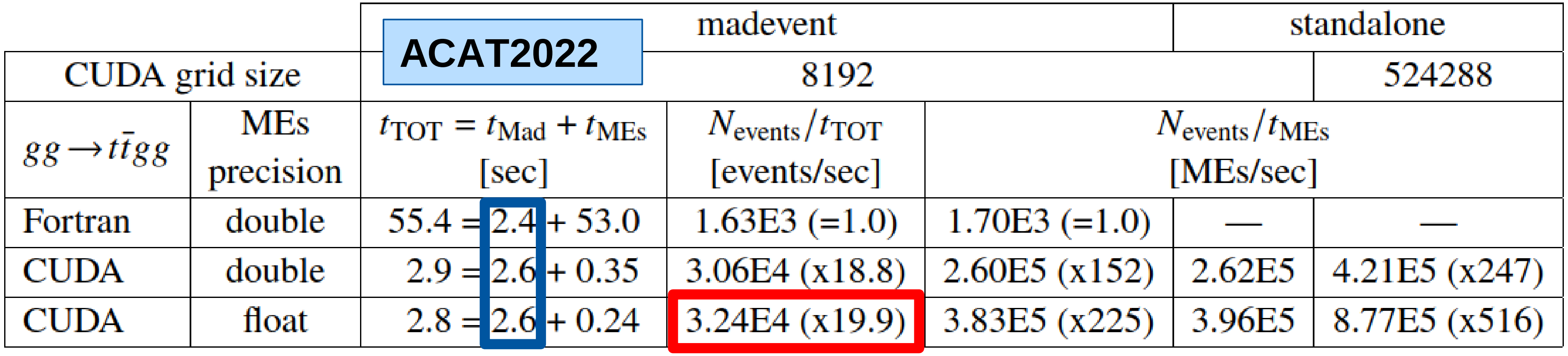


ICHEP2022

		madevent			standalone	
CUDA grid size		8192				524288
$gg \to t\bar{t}gg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]		
Fortran	double	58.3 = 5.2 + 53.1	1.55E3 (=1.0)	1.70E3 (=1.0)	—	—
CUDA	double	6.1 = 5.7 + 0.36	1.49E4 (x9.6)	2.54E5 (x149)	2.51E5	4.20E5 (x247)
CUDA	float	5.7 = 5.4 + 0.24	1.59E4 (x10.3)	3.82E5 (x224)	3.98E5	8.75E5 (x515)

Reduced the overhead from scalar Fortran MadEvent overhead from 10% to 5% of initial Fortran (improved handling of MLM merging)

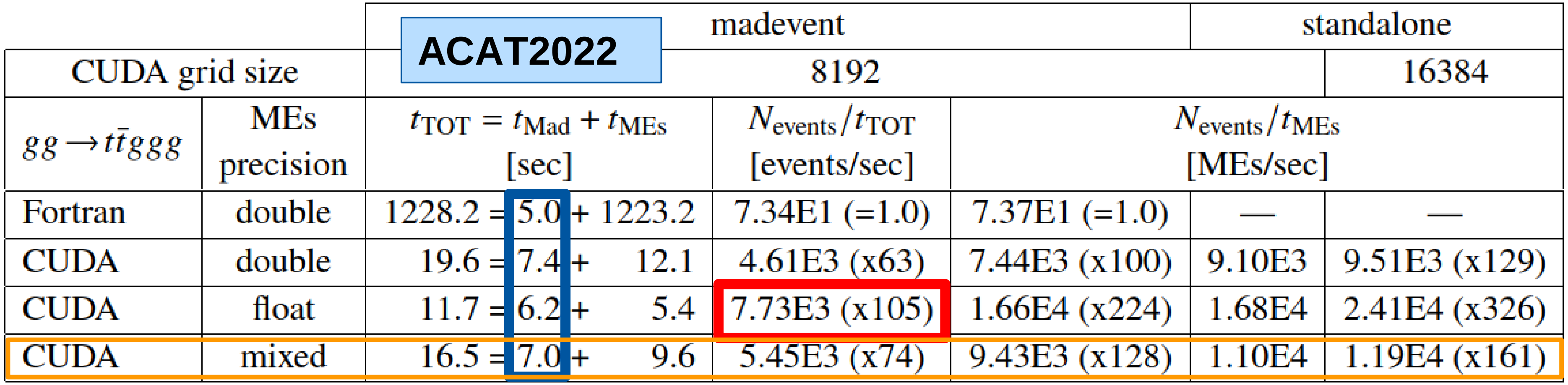
Maximum allowed overall speedup from Amdahl's law is now increased from x10 to x20 - which we do achieve



ACAT2022

		madevent			standalone	
CUDA grid size		8192				524288
$gg \to t\bar{t}gg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]		
Fortran	double	55.4 = 2.4 + 53.0	1.63E3 (=1.0)	1.70E3 (=1.0)	—	—
CUDA	double	2.9 = 2.6 + 0.35	3.06E4 (x18.8)	2.60E5 (x152)	2.62E5	4.21E5 (x247)
CUDA	float	2.8 = 2.6 + 0.24	3.24E4 (x19.9)	3.83E5 (x225)	3.96E5	8.77E5 (x516)

More interesting: MadEvent/CUDA for gg→t$\bar{t}$ggg



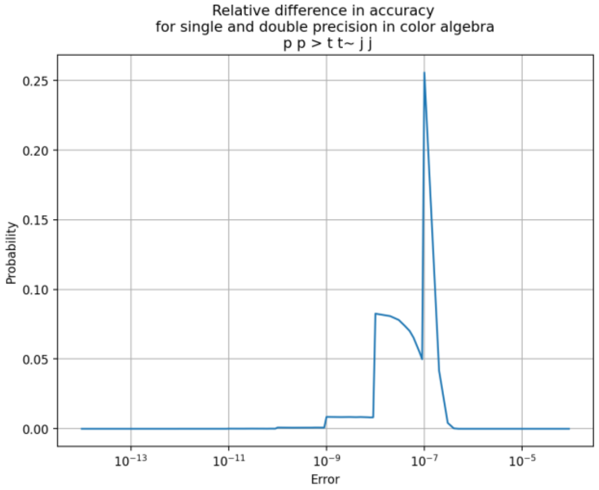
ACAT2022

		madevent			standalone	
CUDA grid size		8192				16384
$gg \to t\bar{t}ggg$	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	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)	—	—
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)

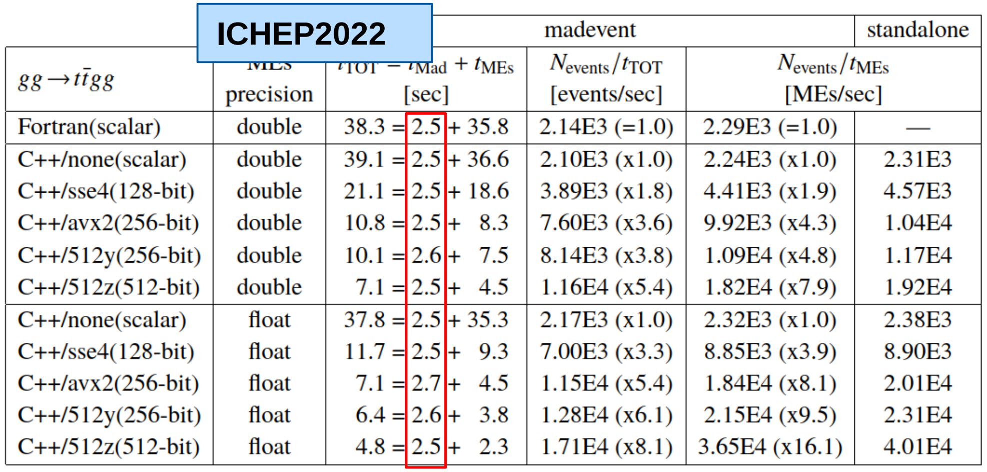
We are lucky! The more complex the physics process, the lower the relative overhead from the scalar Fortran MadEvent - here only 0.5%

Amdahl's law limits the overall speedup to x200, and we achieve x100 in the overall speedup!

- In addition: *prototype a "mixed" floating point precision*
  - Double precision for Feynman diagrams, *single precision for the "color algebra"*
  - Overall performance is in between single and double precision
    - NB: relative importance of color algebra is higher for more complex processes (lucky again!)
  - Physics precision ~ E-6 should be OK for production (float everywhere faster but less precise)



MadEvent/C++ for $gg \to t\bar{t}gg$ (on a single CPU core)



ICHEP2022

		madevent			standalone
$gg \to t\bar{t}gg$	MEs precision	$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]	
Fortran(scalar)	double	38.3 = 2.5 + 35.8	2.14E3 (=1.0)	2.29E3 (=1.0)	—
C++/none(scalar)	double	39.1 = 2.5 + 36.6	2.10E3 (x1.0)	2.24E3 (x1.0)	2.31E3
C++/sse4(128-bit)	double	21.1 = 2.5 + 18.6	3.89E3 (x1.8)	4.41E3 (x1.9)	4.57E3
C++/avx2(256-bit)	double	10.8 = 2.5 + 8.3	7.60E3 (x3.6)	9.92E3 (x4.3)	1.04E4
C++/512y(256-bit)	double	10.1 = 2.6 + 7.5	8.14E3 (x3.8)	1.09E4 (x4.8)	1.17E4
C++/512z(512-bit)	double	7.1 = 2.5 + 4.5	1.16E4 (x5.4)	1.82E4 (x7.9)	1.92E4
C++/none(scalar)	float	37.8 = 2.5 + 35.3	2.17E3 (x1.0)	2.32E3 (x1.0)	2.38E3
C++/sse4(128-bit)	float	11.7 = 2.5 + 9.3	7.00E3 (x3.3)	8.85E3 (x3.9)	8.90E3
C++/avx2(256-bit)	float	7.1 = 2.7 + 4.5	1.15E4 (x5.4)	1.84E4 (x8.1)	2.01E4
C++/512y(256-bit)	float	6.4 = 2.6 + 3.8	1.28E4 (x6.1)	2.15E4 (x9.5)	2.31E4
C++/512z(512-bit)	float	4.8 = 2.5 + 2.3	1.71E4 (x8.1)	3.65E4 (x16.1)	4.01E4

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

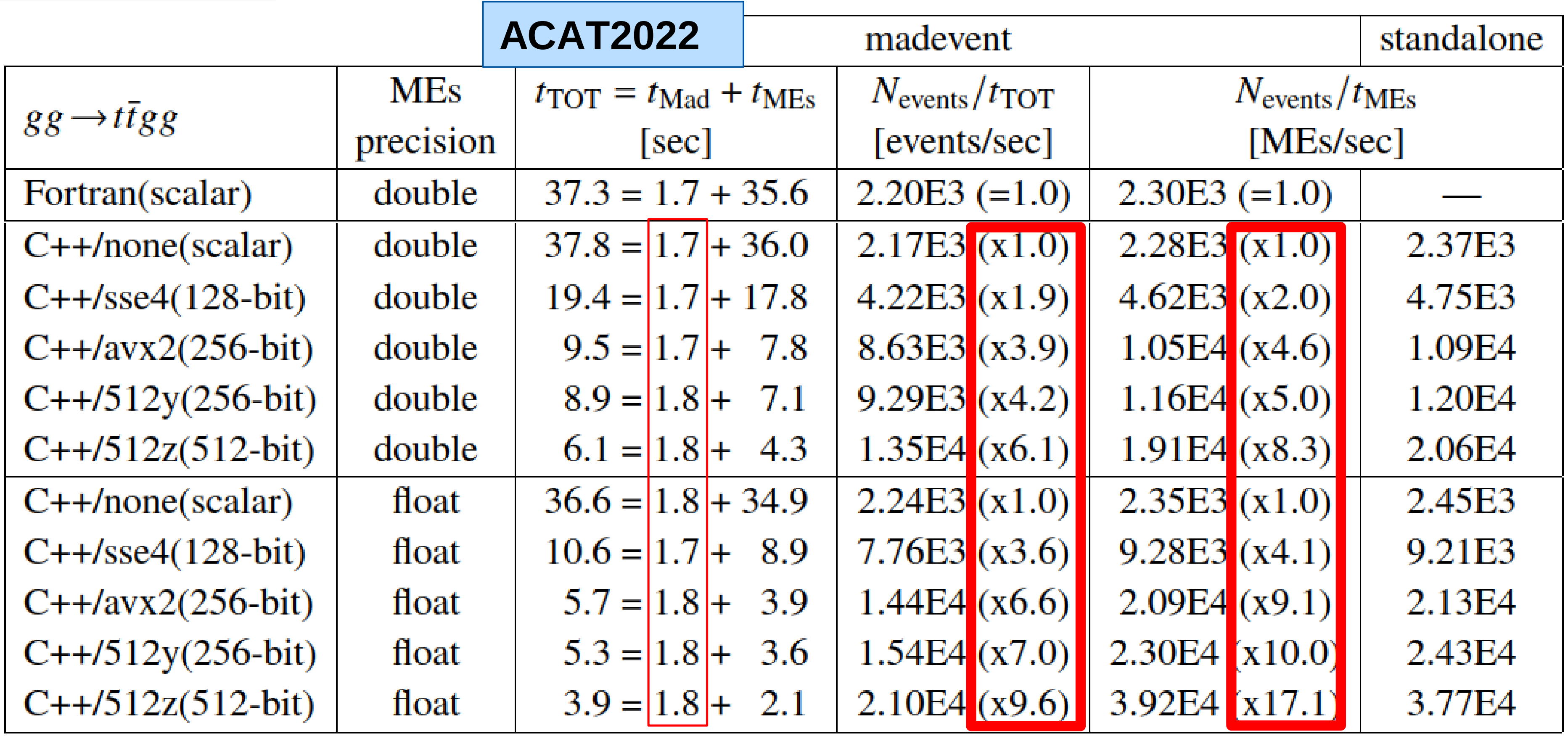
Our ME engine reaches the maximum theoretical SIMD speedup! *

Overall speedup ~ x6 (double) and x10 (float) over scalar Fortran

Improved since ICHEP:

- Lower overhead from scalar Madevent, hence higher overall throughput
- 10% faster MEs via better color algebra algorithm
- (Prototype mixed floating point precision as in CUDA, speedup only in $gg \to t\bar{t}ggg$)

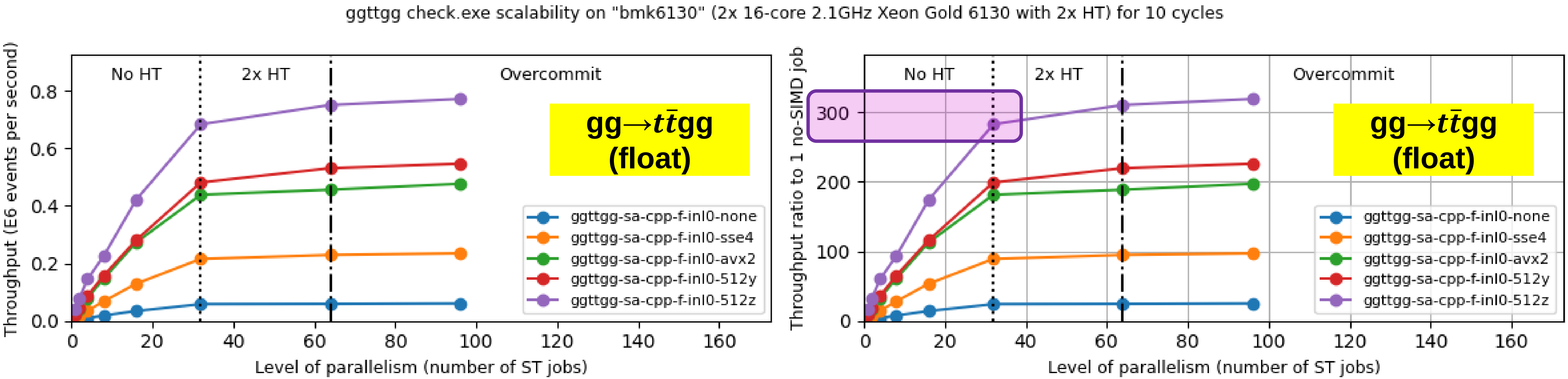
** This is promising in view of the upcoming VPUs with 256 doubles per vector register! (Estela Suarez's plenary today)*



ACAT2022

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

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


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

- Previous tables for SIMD speedups on C++ were for a single CPU core

- **New at ACAT 2022: 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)*

- Plots prepared using HEP-workloads containers developed in the HEP-score project (see D. Giordano's talk)
  - Aggregate MEs throughput from many identical processes using the standalone application

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

- Performance improvements (speed up the Matrix Element calculation in CUDA)
  - Smaller kernels (and fewer events per grid): from one-event/all-helicities to one-event/one-helicity per thread
  - Smaller kernels: split Feynman diagrams and color algebra
  - Move color algebra to tensor cores (e.g. using cublas)

- Performance improvements (speed up the Fortran MadEvent scalar component)
  - Parallelize it on the many cores of the CPU (heterogeneous workflow)?
  - Further profiling...

- Functional improvements and longer term plans
  - Support for NLO QCD processes
  - Event-by-event ME reweighting (and derivatives?)

Fortran vs C++/CUDA/PFs: *(not yet)* an apples-to-apples comparison!

MadEvent + Fortran
(double precision)

SOON!
(THE GOAL)

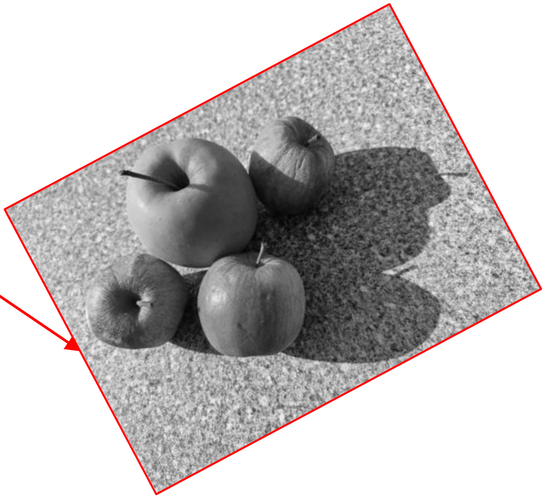
MadEvent + CUDA/C++/PFs
(double precision)

Cross-sections: same as Fortran
with 2E-14 relative precision* – OK

LHE files: same events as Fortran, each with
- same weight to 7 significant digits*
- same leading color flow *(needed for parton showers)*
- same helicities *(needed for particle decays)*

** WIP – how much lower precision for cross-sections and event weights with single or with 'mixed' precision?*

TODAY



Cross-sections: same as Fortran
with 2E-14 relative precision* – OK

LHE files: same events as Fortran, each with
- same weight to 7 significant digits* – OK
- same leading color flow – not yet
- same helicities – not yet

NB: THE ***SAME*** APPLES!

Implementing the per-event choice of color and helicity is our last main TO-DO before an alpha release: SOON!

Conclusions

- The Matrix Element calculation in ANY ME event generator can be efficiently parallelized using SIMD or GPUs

- Our reengineering of MG5aMC is close to a first fully functional alpha release for LO QCD processes
  - The new ME calculation is integrated in MadEvent, we are mainly missing the per-event choice of colour and helicity

- On CPUs, using vectorized C++ we achieve the maximum x8/x16 (double/float) SIMD speedups for MEs alone
  - The speedups for the overall workflow are slightly lower due to Amdahl's law, but not much
  - Example: our overall speedup is currently x6/x10 for $gg \to t\bar{t}gg$ (on one CPU core)

- On GPUs, using CUDA we achieve O(100-1000) speedups for MEs alone
  - The speedups may be much lower due to Amdahl's law, but we are improving on that
  - Example: our overall speedup is currently x60/x100 (double/float) for $gg \to t\bar{t}ggg$

- Floats are x2 faster than doubles in SIMD and data centre GPUs - we are testing their use e.g. in colour algebra

- Using SYCL and Kokkos we get similar performances to CUDA and we may also run on AMD or Intel GPUs

Acknowledgements

- We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

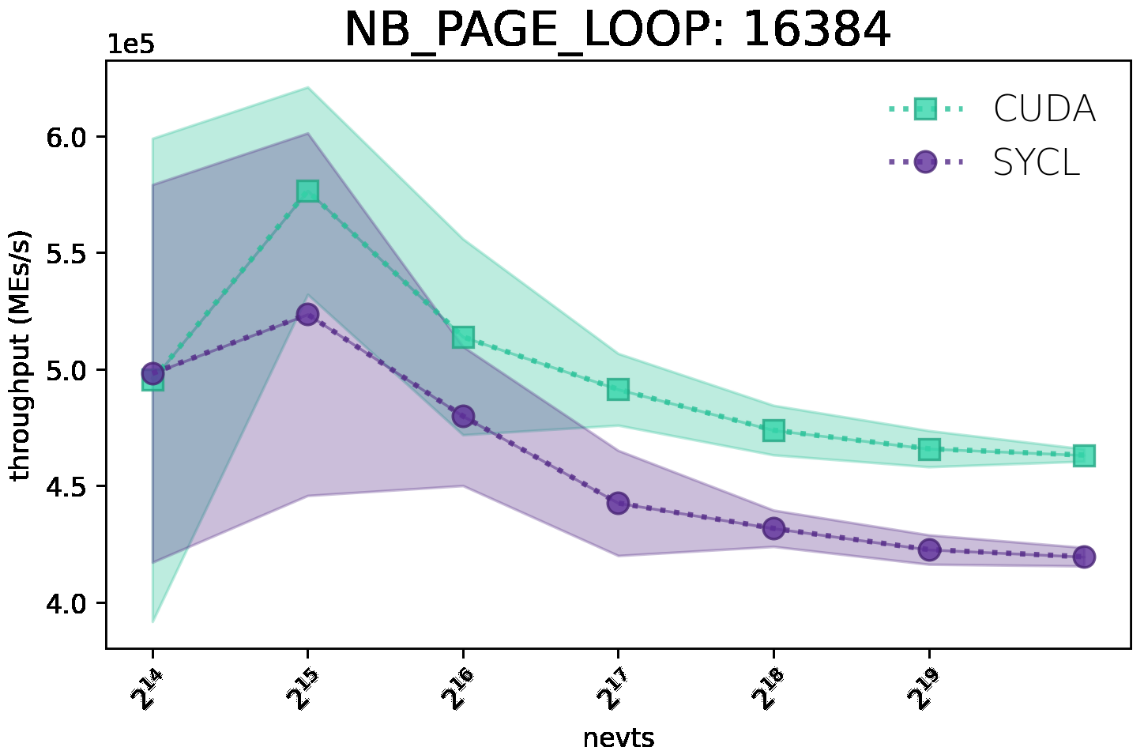
- We gratefully acknowledge the use (under PRACE proposal PRACE-DEV-2022D01-022) of the JUWELS supercomputer and other computing resources provided and operated by the Jülich Supercomputing Centre at Forschungszentrum Jülich.

- 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.

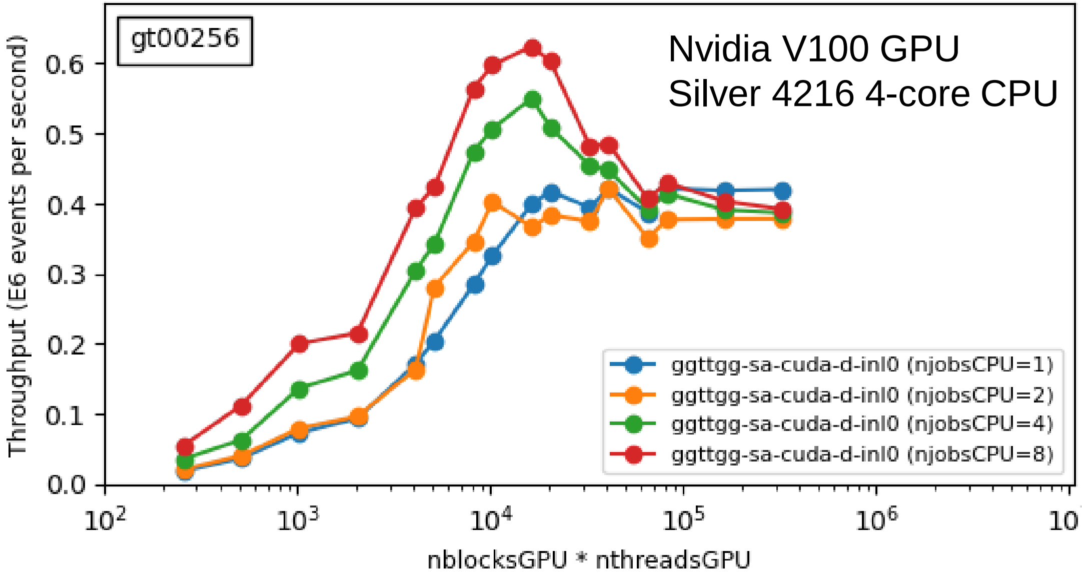
BACKUP SLIDES

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



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

Some ideas for heterogeneous processing



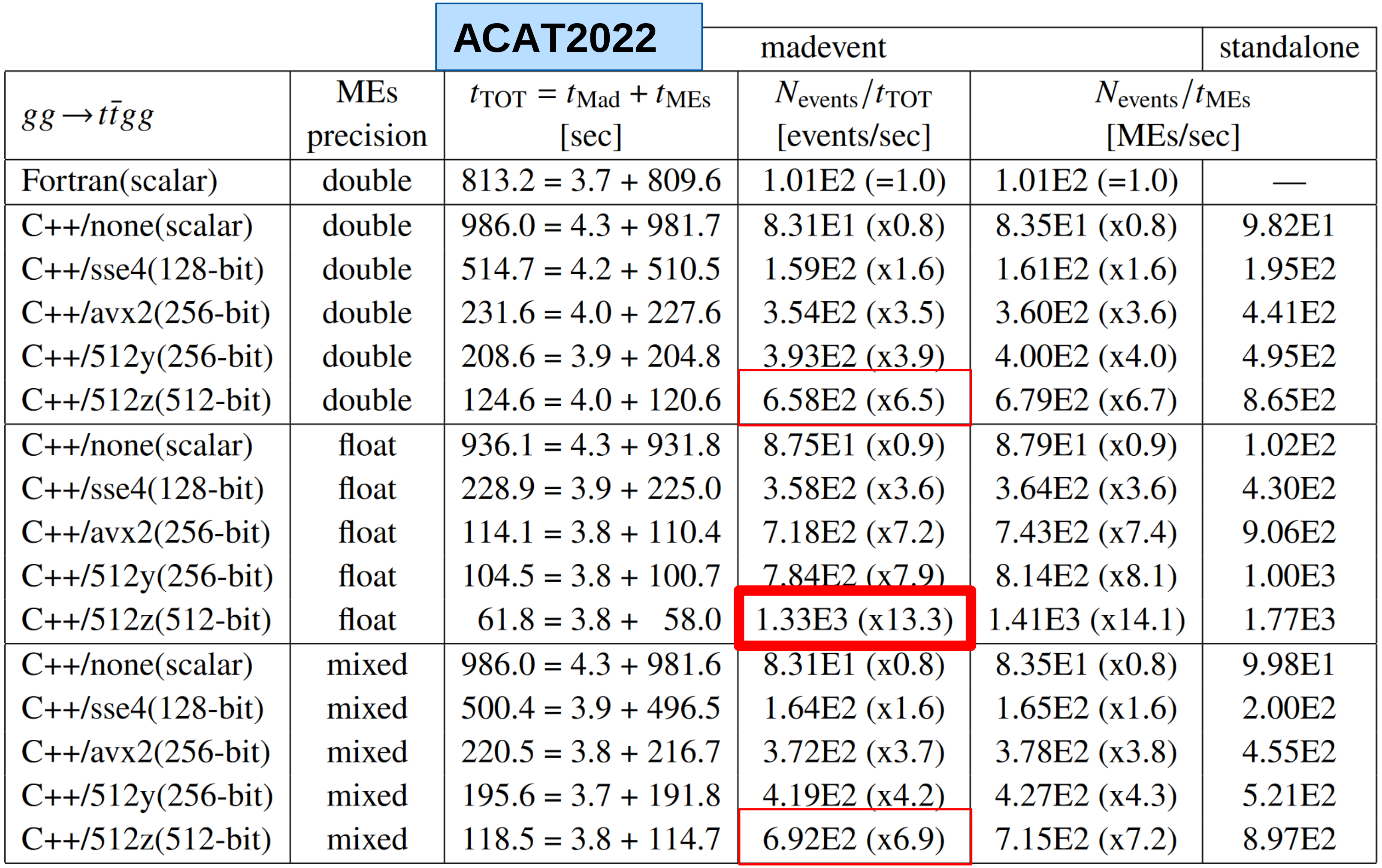
Throughput variation as a function of *GPU grid size (#blocks * #threads)*

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 $gg\to t\bar{t}ggg$ (on a single core)



ACAT2022

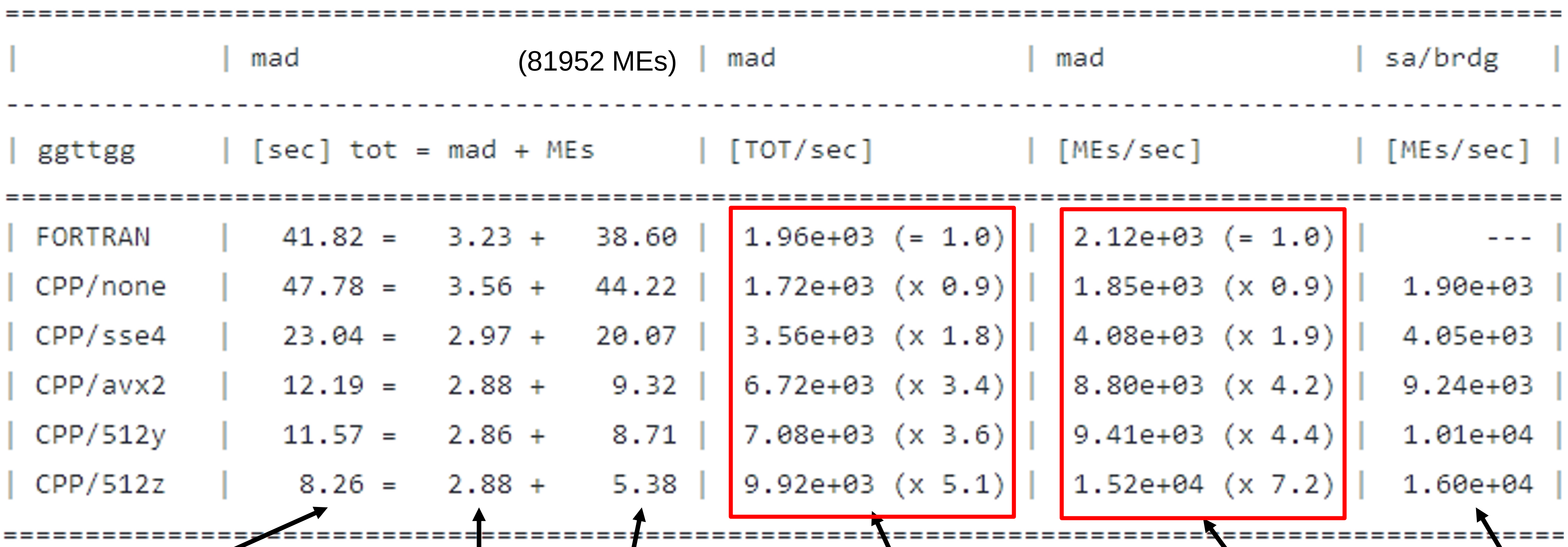
		madevent			standalone
$gg \to t\bar{t}gg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]	
Fortran(scalar)	double	813.2 = 3.7 + 809.6	1.01E2 (=1.0)	1.01E2 (=1.0)	—
C++/none(scalar)	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 $gg\to t\bar{t}ggg$ than in $gg\to t\bar{t}gg$: higher **overall throughput speedup x13!**
- Mixed floating-point precision (single precision color algebra) is 5-10% better than double

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)

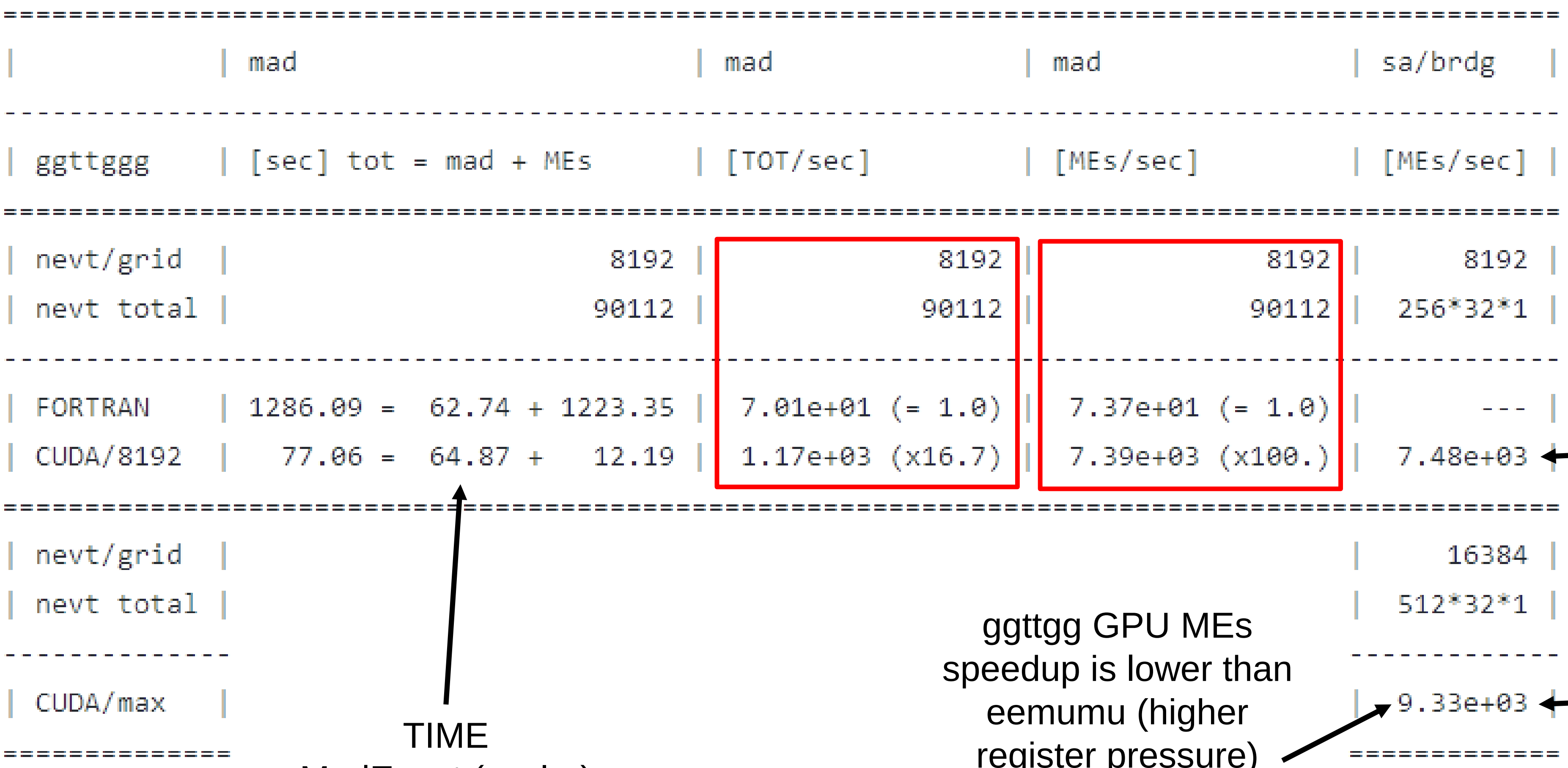
THROUGHPUT
MadEvent + MEs
(within madevent)

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

16k events per GPU grid

2. INCREASE GPU GRIDS (REDUCE CPU MEMORY) TO INCREASE SPEEDUP

TIME MadEvent (scalar)

1. REDUCE THIS TO INCREASE SPEEDUP

ggttgg GPU MEs speedup is lower than eemumu (higher register pressure)

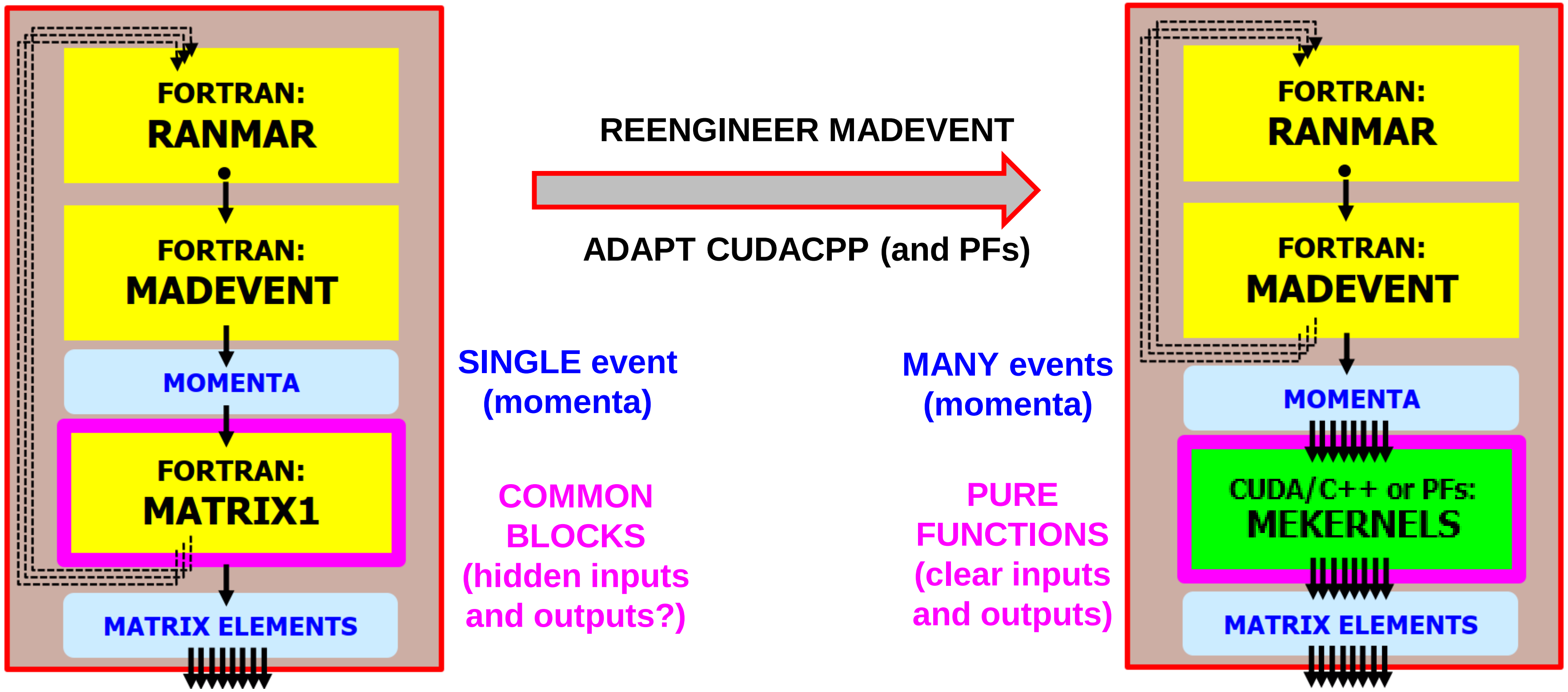
3. SMALLER GPU KERNELS TO INCREASE SPEEDUP

Matrix element integration in MadEvent

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

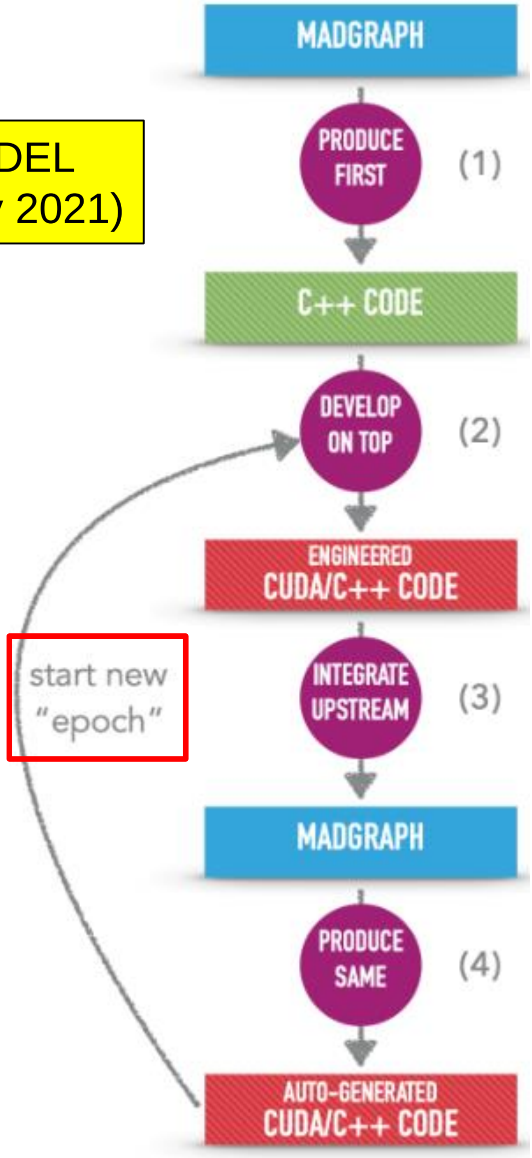
Linking Fortran and C++ has been easy. As expected, the two main issues have been, instead:

- 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



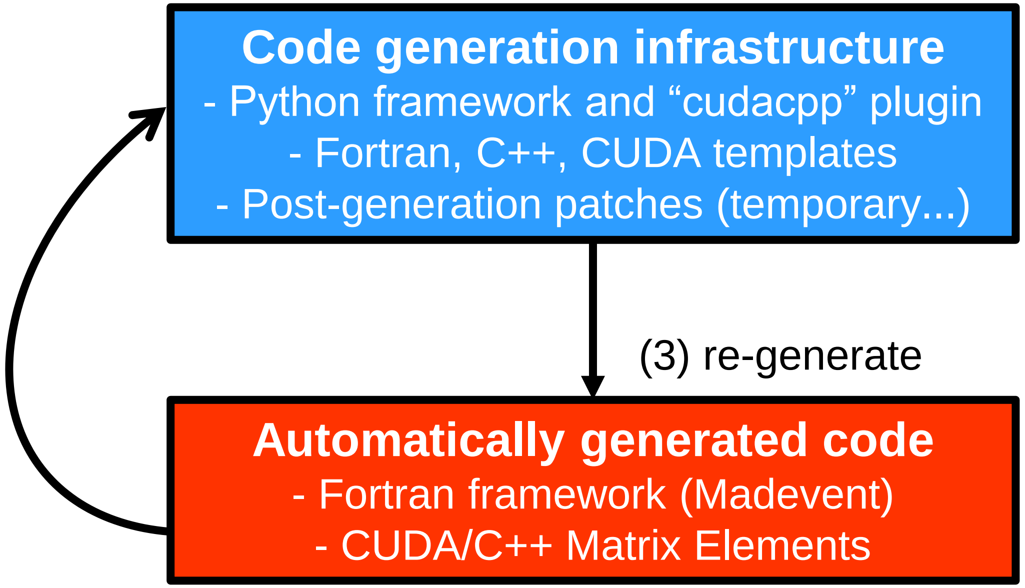
Code generation: from many “epochs” to a single evolving “epoch”

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...)

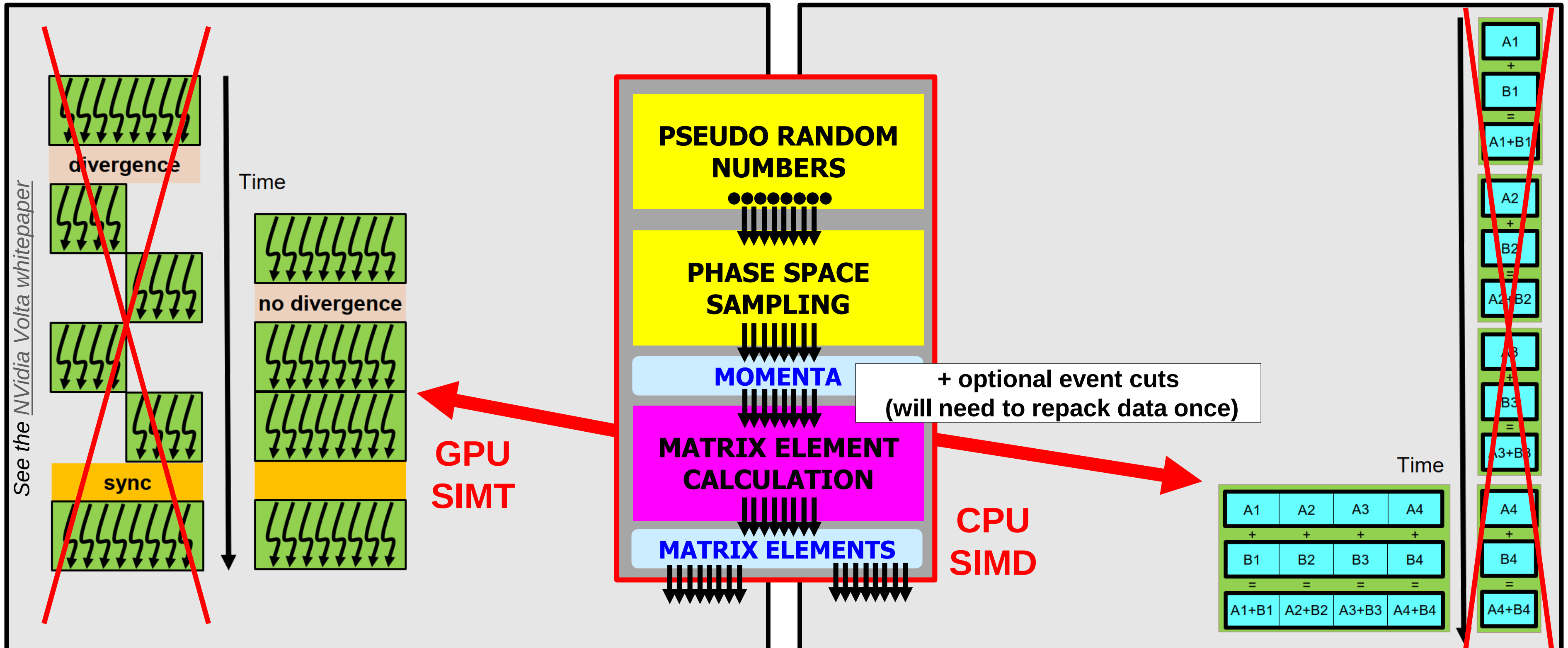
(3) re-generate

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

MG5aMC computational anatomy and data parallelism strategy

- 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)



+ optional event cuts (will need to repack data once)

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

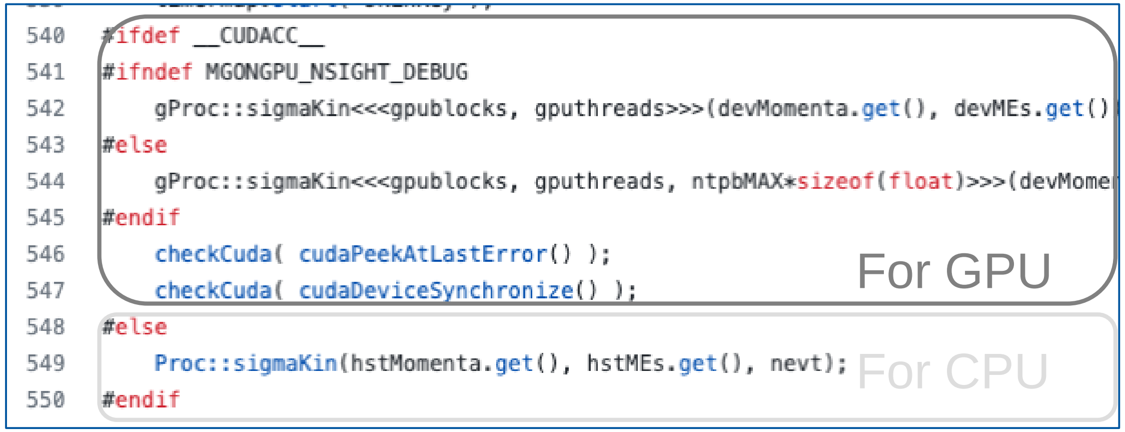
CPU SIMD (Single Instruction Multiple Data)
Lockstep: same op for all data in a vector register
Minimum parallelism: 2 to 16 (SSE/AVX2/AVX512...)

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

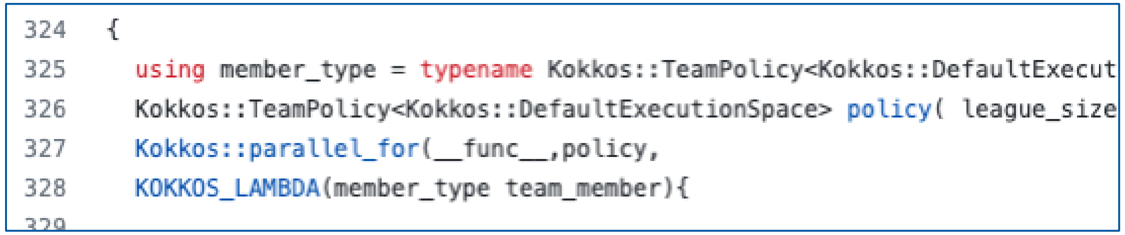


```
#ifdef __CUDACC__
#ifndef MGONGPU_NSIGHT_DEBUG
    gProc::sigmaKin<<<gpublocks, gputhreads>>>(devMomenta.get(), devMEs.get()
#else
    gProc::sigmaKin<<<gpublocks, gputhreads, ntpbMAX*sizeof(float)>>>(devMome
#endif
    checkCuda( cudaPeekAtLastError() );
    checkCuda( cudaDeviceSynchronize() );
#else
    Proc::sigmaKin(hstMomenta.get(), hstMEs.get(), nevt);
#endif
```

For GPU

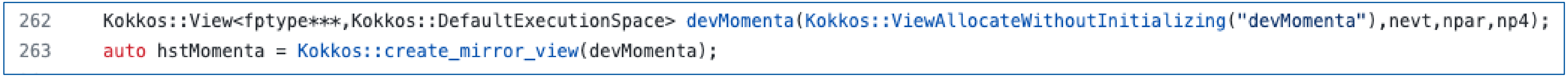
For CPU

Kokkos example of Templating & lambda

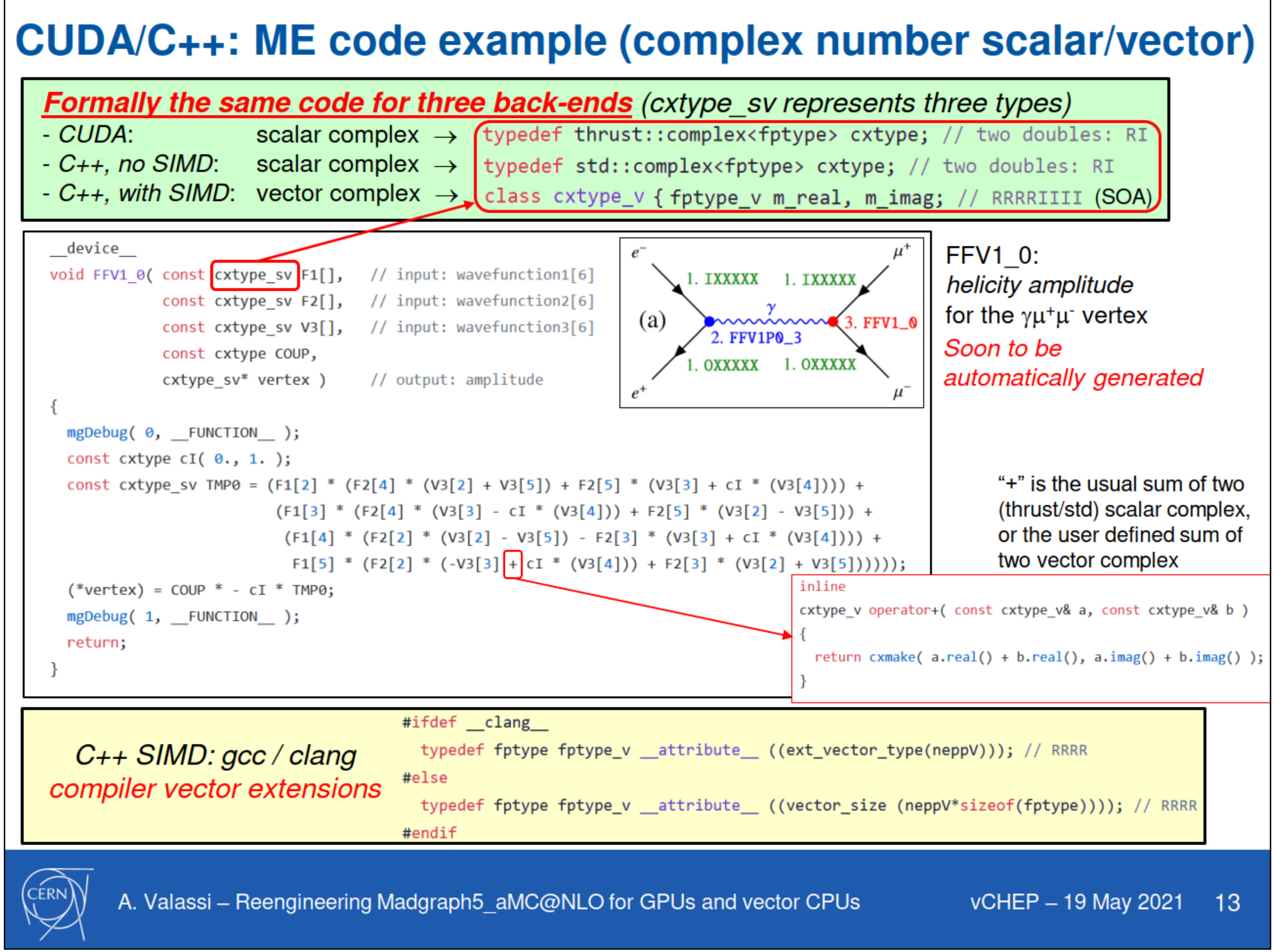


```
{
  using member_type = typename Kokkos::TeamPolicy<Kokkos::DefaultExecut
  Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace> policy( league_size
  Kokkos::parallel_for(__func__,policy,
  KOKKOS_LAMBDA(member_type team_member){
```

Kokkos example of Memory Management



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



CUDA/C++: ME code example (complex number scalar/vector)

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

- *CUDA*: scalar complex → `typedef thrust::complex<fptype> cxtype; // two doubles: RI`
- *C++, no SIMD*: scalar complex → `typedef std::complex<fptype> cxtype; // two doubles: RI`
- *C++, with SIMD*: vector complex → `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
Soon to be 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
```

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   13

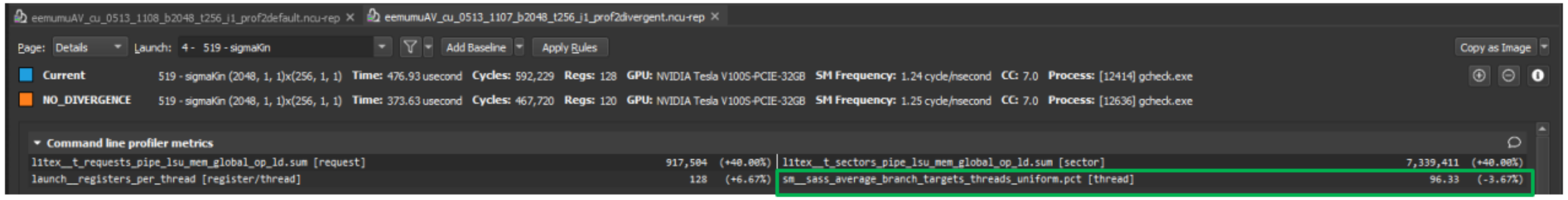
CUDA: Profiling with NVidia NSight Compute – ncu

- We regularly profile CUDA with ncu [both one-off studies and on-commit checks]
  - *Thanks to our mentors at the Sheffield GPU hackathon for getting us started!*

- We see *no evidence of thread divergence* *[branch efficiency is 100%]*

- Our *AOSOA layout* ensures *coalesced memory access* [requests vs transactions]

- We continuously *monitor register pressure* – decreasing it is one of our future goals
  - We plan to split the ME computation into many kernels coordinated by CUDA Graphs

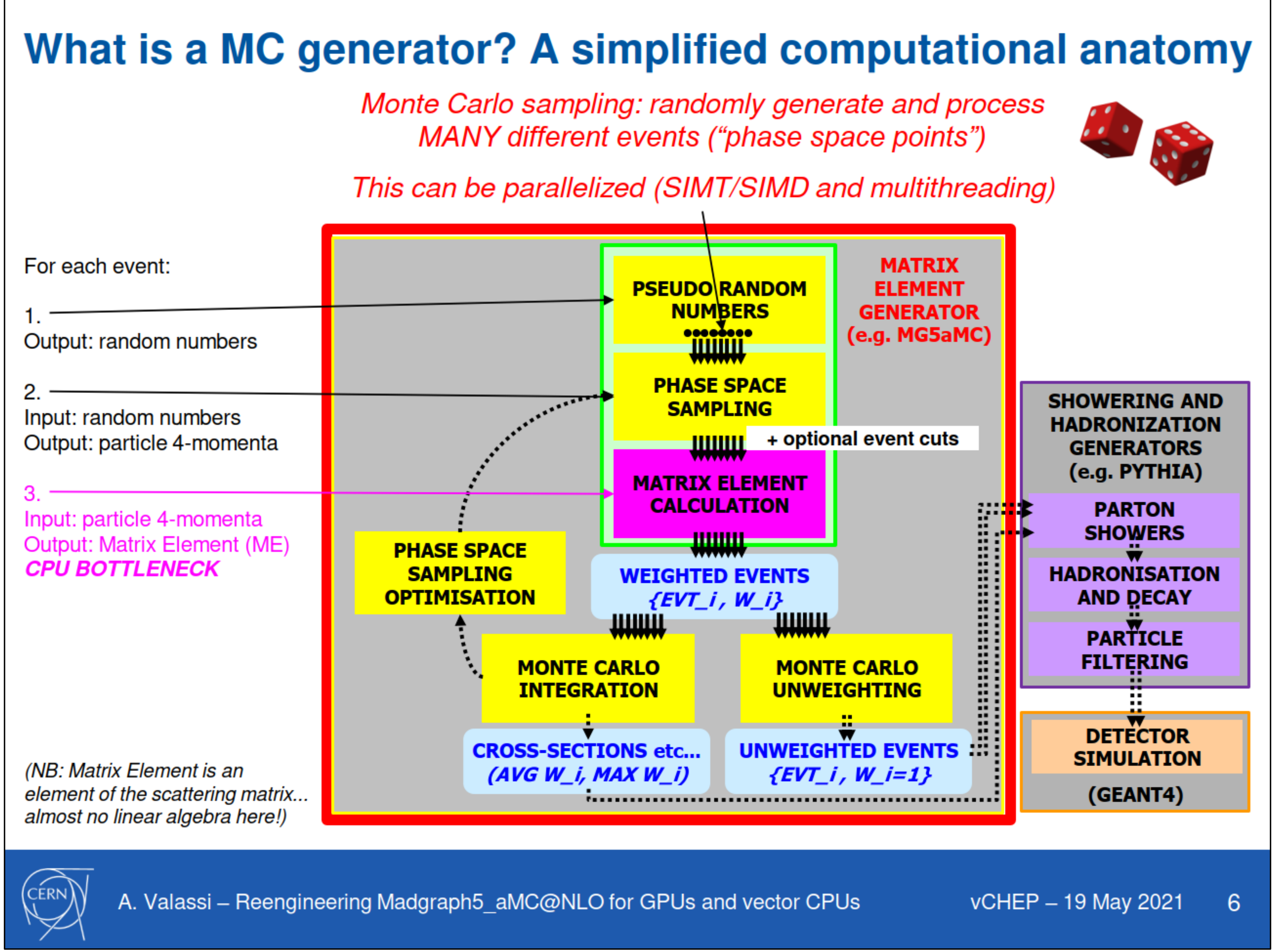

eemumuAV_cu_0513_1108_b2048_t256_i1_prof2default.ncu-rep ×
eemumuAV_cu_0513_1107_b2048_t256_i1_prof2divergent.ncu-rep ×
Add Baseline
Apply Rules
Copy as Image

```
Page: Details   Launch: 4 -  519 - sigmaKin

Current         519 - sigmaKin (2048, 1, 1)x(256, 1, 1)  Time: 476.93 usecond  Cycles: 592,229  Regs: 128  GPU: NVIDIA Tesla V100S-PCIE-32GB  SM Frequency: 1.24 cycle/nsecond  CC: 7.0  Process: [12414] gcheck.exe
NO_DIVERGENCE   519 - sigmaKin (2048, 1, 1)x(256, 1, 1)  Time: 373.63 usecond  Cycles: 467,720  Regs: 120  GPU: NVIDIA Tesla V100S-PCIE-32GB  SM Frequency: 1.25 cycle/nsecond  CC: 7.0  Process: [12636] gcheck.exe

▾ Command line profiler metrics
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum [request]   917,504  (+40.00%) | l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]        7,339,411  (+40.00%)
launch__registers_per_thread [register/thread]                  128   (+6.67%) | sm__sass_average_branch_targets_threads_uniform.pct [thread]      96.33   (-3.67%)
```

Example: compare baseline implementation (100% branch efficiency) to a test with artificial divergence

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   14

EVEN MORE BACKUP SLIDES



What is a MC 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)

For each event:

1.
Output: random numbers

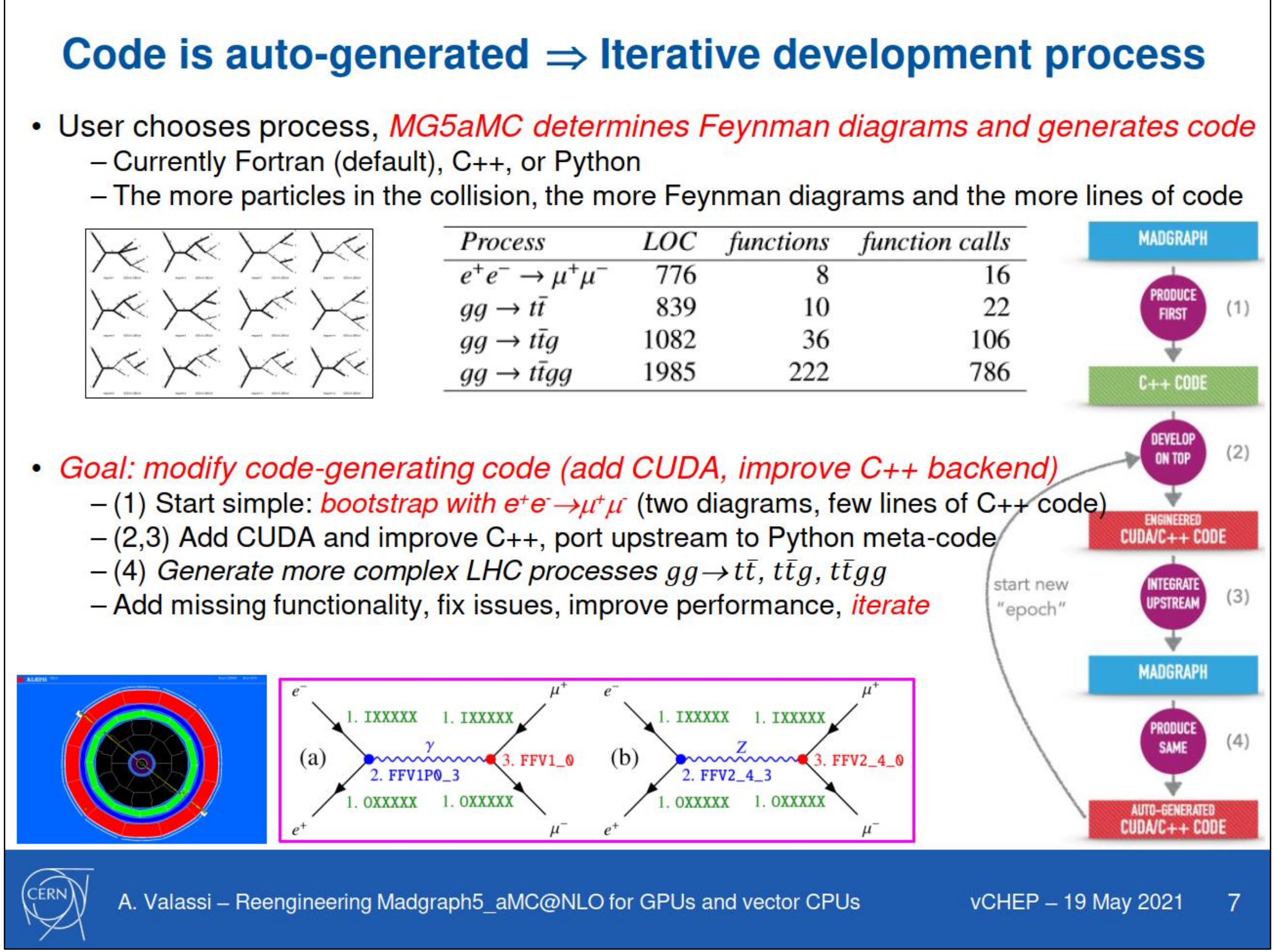
2.
Input: random numbers
Output: particle 4-momenta

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

(NB: Matrix Element is an element of the scattering matrix... almost no linear algebra here!)

MATRIX ELEMENT GENERATOR (e.g. MG5aMC)
PSEUDO RANDOM NUMBERS
PHASE SPACE SAMPLING
+ optional event cuts
MATRIX ELEMENT CALCULATION
WEIGHTED EVENTS {EVT_i, W_i}
PHASE SPACE SAMPLING OPTIMISATION
MONTE CARLO INTEGRATION
MONTE CARLO UNWEIGHTING
CROSS-SECTIONS etc... (AVG W_i, MAX W_i)
UNWEIGHTED EVENTS {EVT_i, W_i=1}
SHOWERING AND HADRONIZATION GENERATORS (e.g. PYTHIA)
PARTON SHOWERS
HADRONISATION AND DECAY
PARTICLE FILTERING
DETECTOR SIMULATION
(GEANT4)

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   6



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

MADGRAPH

PRODUCE FIRST (1)

C++ CODE

DEVELOP ON TOP (2)

ENGINEERED CUDA/C++ CODE

INTEGRATE UPSTREAM (3)

MADGRAPH

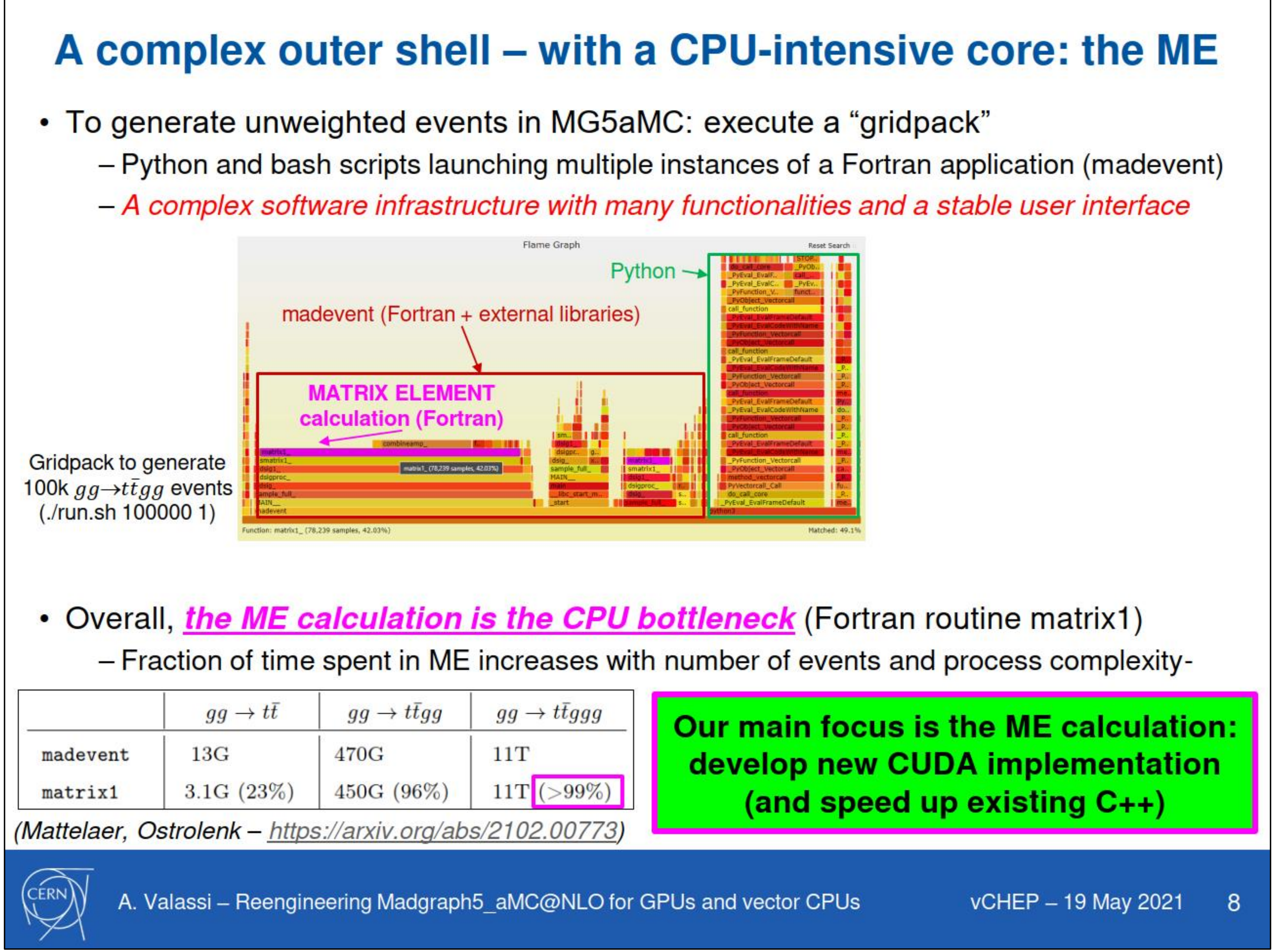
PRODUCE SAME (4)

AUTO-GENERATED CUDA/C++ CODE

start new "epoch"

- *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*

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   7



A complex outer shell – with a CPU-intensive core: the ME

- To generate unweighted events in MG5aMC: execute a "gridpack"
  - Python and bash scripts launching multiple instances of a Fortran application (madevent)
  - *A complex software infrastructure with many functionalities and a stable user interface*

Flame Graph
Reset Search
Python
madevent (Fortran + external libraries)
MATRIX ELEMENT calculation (Fortran)
matrix1_
smatrix1_
dsig1_
dsigproc_
dsig_
sample_full_
MAIN__
madevent
matrix1_ (78,239 samples, 42.03%)
python3
Function: matrix1_ (78,239 samples, 42.03%)
Matched: 49.1%

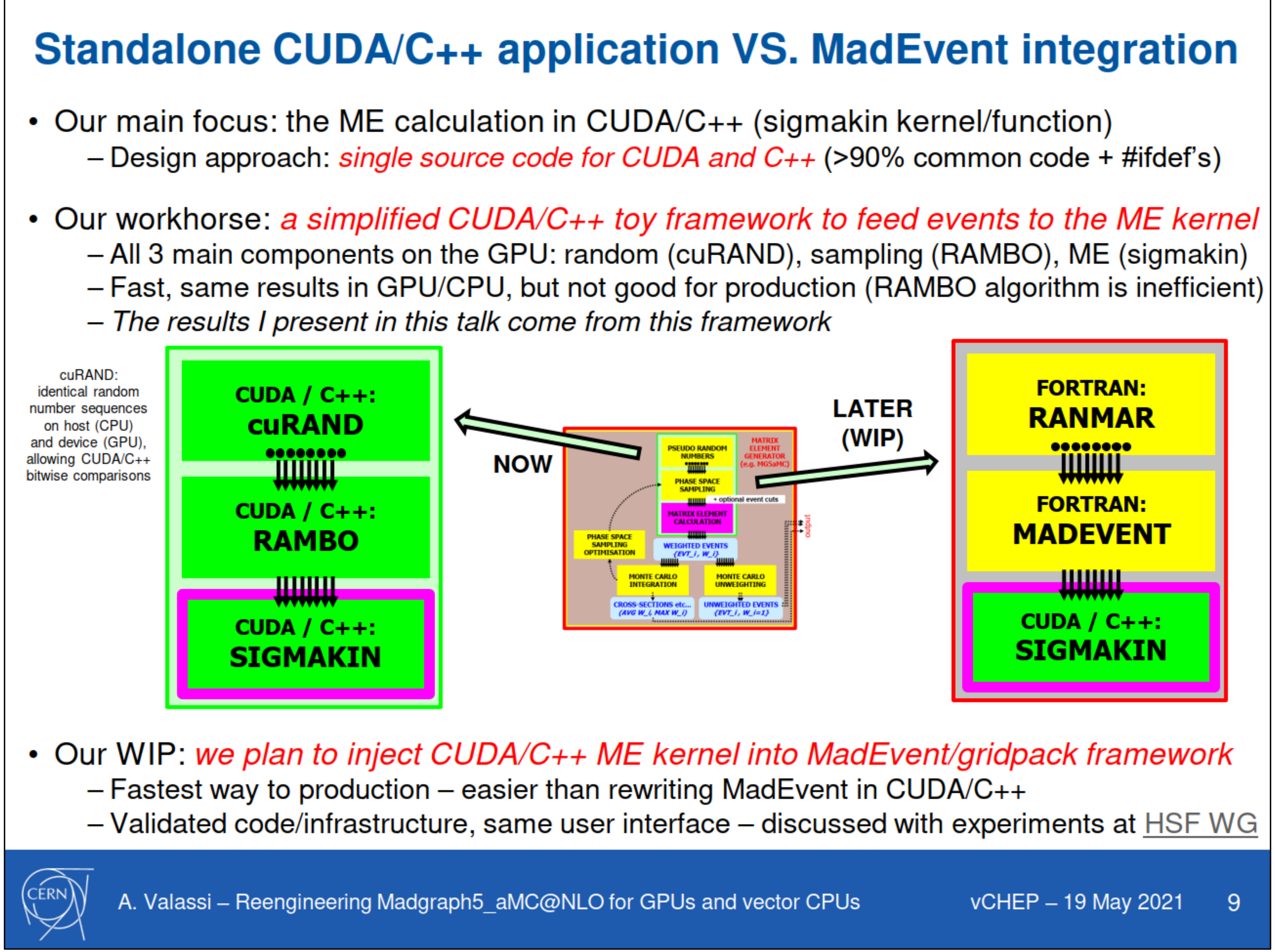
Gridpack to generate 100k $gg \to t\bar{t}gg$ events (./run.sh 100000 1)

- Overall, ***the ME calculation is the CPU bottleneck*** (Fortran routine matrix1)
  - Fraction of time spent in ME increases with number of events and process complexity-

	$gg \to t\bar{t}$	$gg \to t\bar{t}g$	$gg \to t\bar{t}gg$
madevent	13G	470G	11T
matrix1	3.1G (23%)	450G (96%)	11T (>99%)

(Mattelaer, Ostrolenk – https://arxiv.org/abs/2102.00773)

Our main focus is the ME calculation: develop new CUDA implementation (and speed up existing C++)



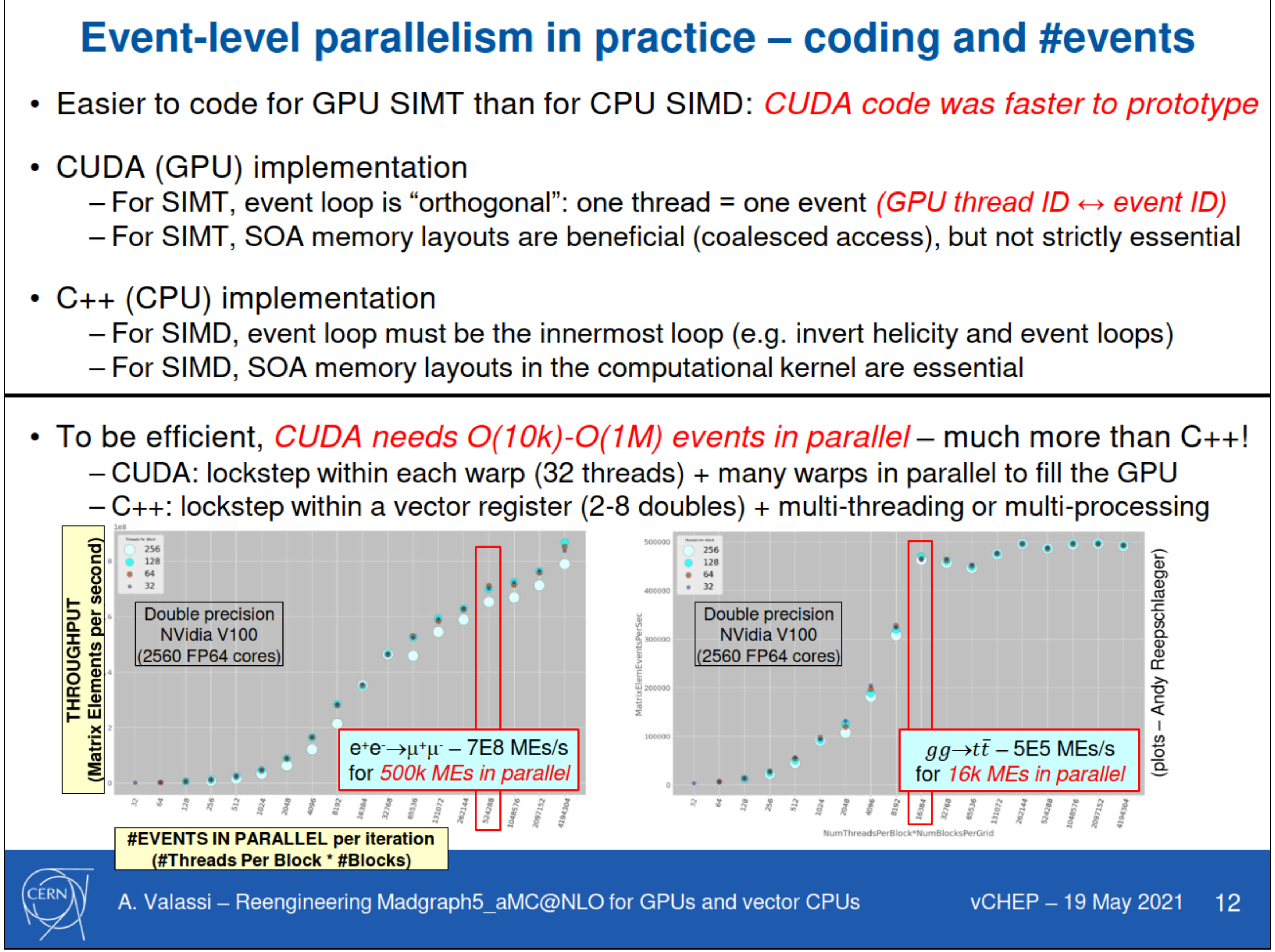
Standalone CUDA/C++ application VS. MadEvent integration

- 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
CUDA / C++: cuRAND
CUDA / C++: RAMBO
CUDA / C++: SIGMAKIN
NOW
PSEUDO RANDOM NUMBERS
PHASE SPACE SAMPLING
+ optional event cuts
MATRIX ELEMENT CALCULATION
MATRIX ELEMENT GENERATOR (e.g. MG5aMC)
PHASE SPACE SAMPLING OPTIMISATION
WEIGHTED EVENTS {EVT_i, W_i}
MONTE CARLO INTEGRATION
MONTE CARLO UNWEIGHTING
CROSS-SECTIONS etc... (AVG W_i, MAX W_i)
UNWEIGHTED EVENTS {EVT_i, W_i=1}
output
LATER (WIP)
FORTRAN: RANMAR
FORTRAN: MADEVENT
CUDA / C++: SIGMAKIN

- 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

CERN A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs vCHEP – 19 May 2021 9



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

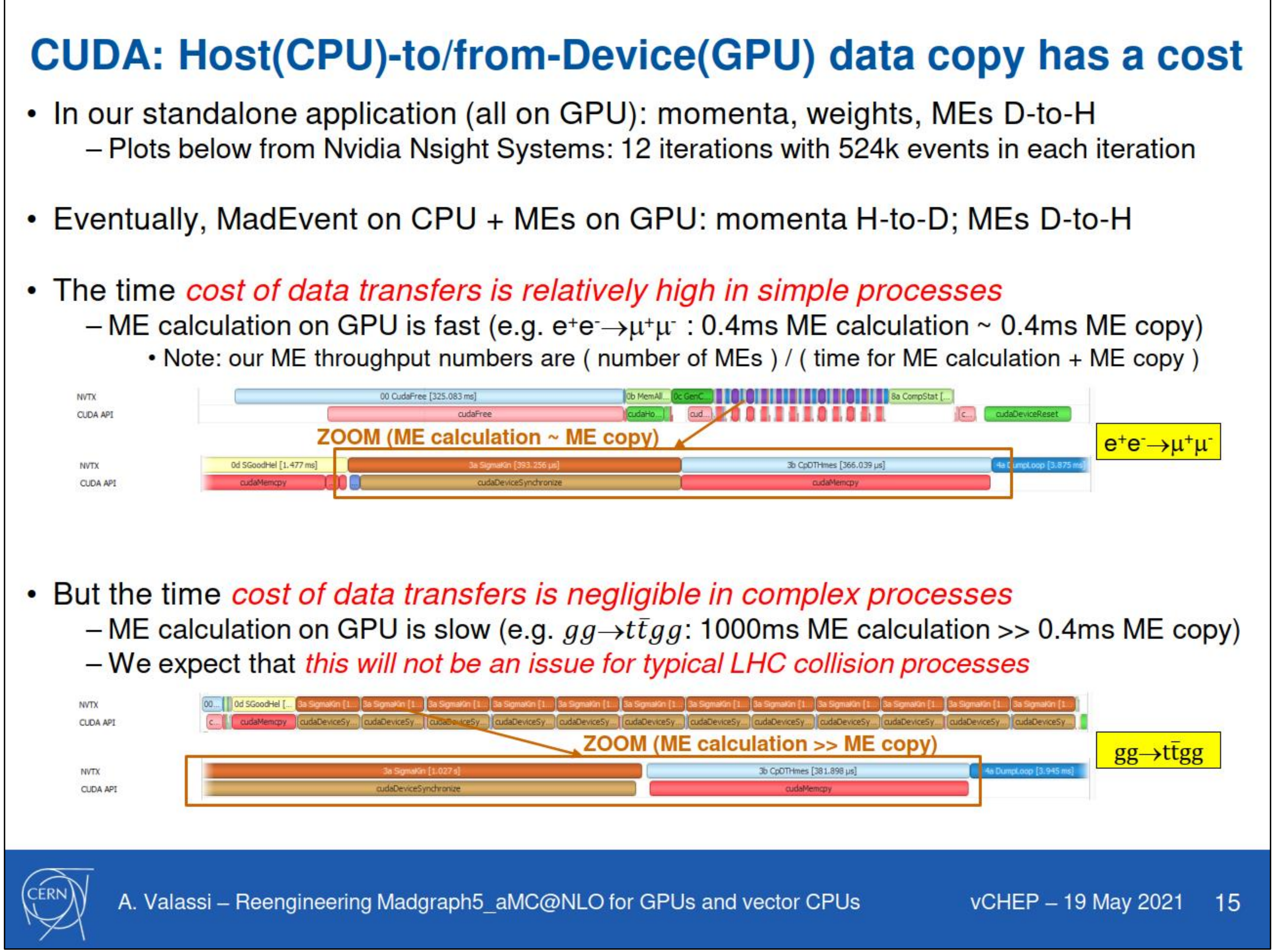
$e^+e^- \to \mu^+\mu^-$ – 7E8 MEs/s for *500k MEs in parallel*

$gg \to t\bar{t}$ – 5E5 MEs/s for *16k MEs in parallel*

(plots – Andy Reepschlaeger)

#EVENTS IN PARALLEL per iteration (#Threads Per Block * #Blocks)

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs vCHEP – 19 May 2021 12



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)

ZOOM (ME calculation ~ ME copy)

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

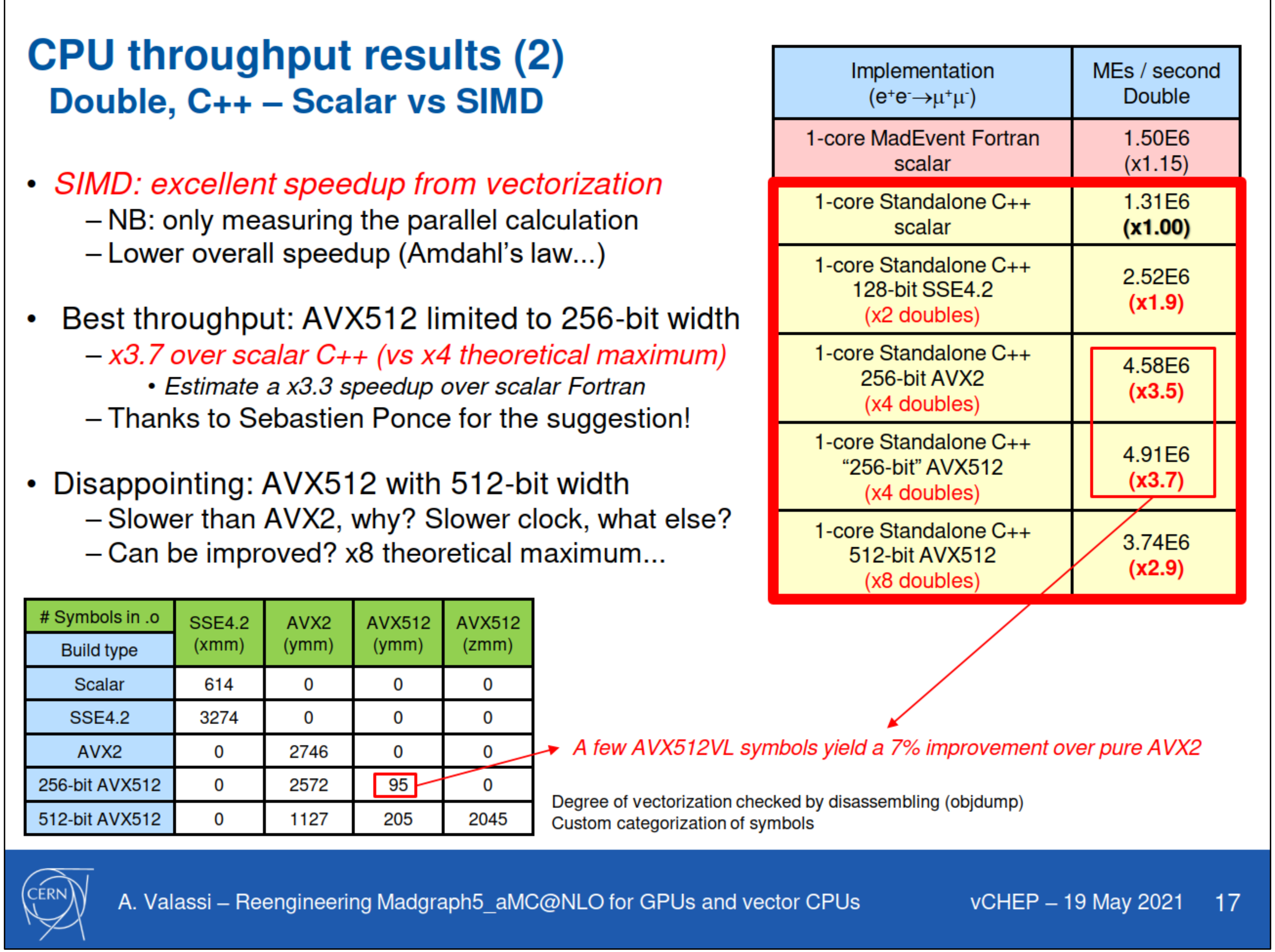
- But the time *cost of data transfers is negligible in complex processes*
  - ME calculation on GPU is slow (e.g. $gg \to t\bar{t}gg$: 1000ms ME calculation >> 0.4ms ME copy)
  - We expect that *this will not be an issue for typical LHC collision processes*

ZOOM (ME calculation >> ME copy)

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

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs

vCHEP – 19 May 2021 15



CPU throughput results (2)

Double, C++ – Scalar vs SIMD

- *SIMD: excellent speedup from vectorization*
  - NB: only measuring the parallel calculation
  - Lower overall speedup (Amdahl's law...)
- Best throughput: AVX512 limited to 256-bit width
  - *x3.7 over scalar C++ (vs x4 theoretical maximum)*
    - *Estimate a x3.3 speedup over scalar Fortran*
  - Thanks to Sebastien Ponce for the suggestion!
- Disappointing: AVX512 with 512-bit width
  - Slower than AVX2, why? Slower clock, what else?
  - Can be improved? x8 theoretical maximum...

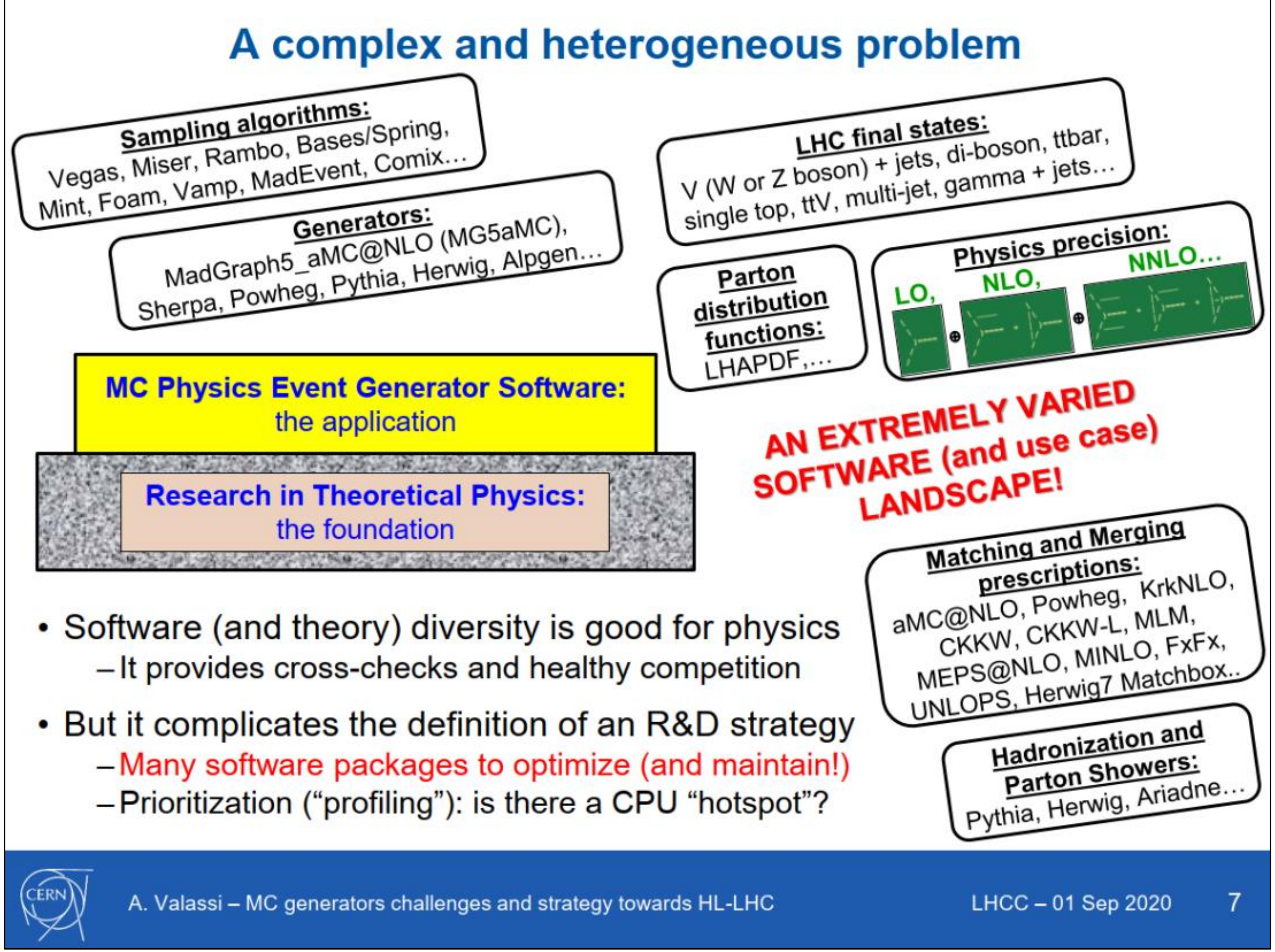
Implementation ($e^+e^-\to\mu^+\mu^-$)	MEs / second Double
1-core MadEvent Fortran scalar	1.50E6 (x1.15)
1-core Standalone C++ scalar	1.31E6 **(x1.00)**
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles)	2.52E6 **(x1.9)**
1-core Standalone C++ 256-bit AVX2 (x4 doubles)	4.58E6 **(x3.5)**
1-core Standalone C++ "256-bit" AVX512 (x4 doubles)	4.91E6 **(x3.7)**
1-core Standalone C++ 512-bit AVX512 (x8 doubles)	3.74E6 **(x2.9)**

# Symbols in .o Build type	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
Scalar	614	0	0	0
SSE4.2	3274	0	0	0
AVX2	0	2746	0	0
256-bit AVX512	0	2572	95	0
512-bit AVX512	0	1127	205	2045

A few AVX512VL symbols yield a 7% improvement over pure AVX2

Degree of vectorization checked by disassembling (objdump)
Custom categorization of symbols

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   17



A complex and heterogeneous problem

Sampling algorithms:
Vegas, Miser, Rambo, Bases/Spring,
Mint, Foam, Vamp, MadEvent, Comix…

Generators:
MadGraph5_aMC@NLO (MG5aMC),
Sherpa, Powheg, Pythia, Herwig, Alpgen…

LHC final states:
V (W or Z boson) + jets, di-boson, ttbar,
single top, ttV, multi-jet, gamma + jets…

Parton distribution functions:
LHAPDF,…

Physics precision:
LO, NLO, NNLO…

MC Physics Event Generator Software:
the application

Research in Theoretical Physics:
the foundation

AN EXTREMELY VARIED SOFTWARE (and use case) LANDSCAPE!

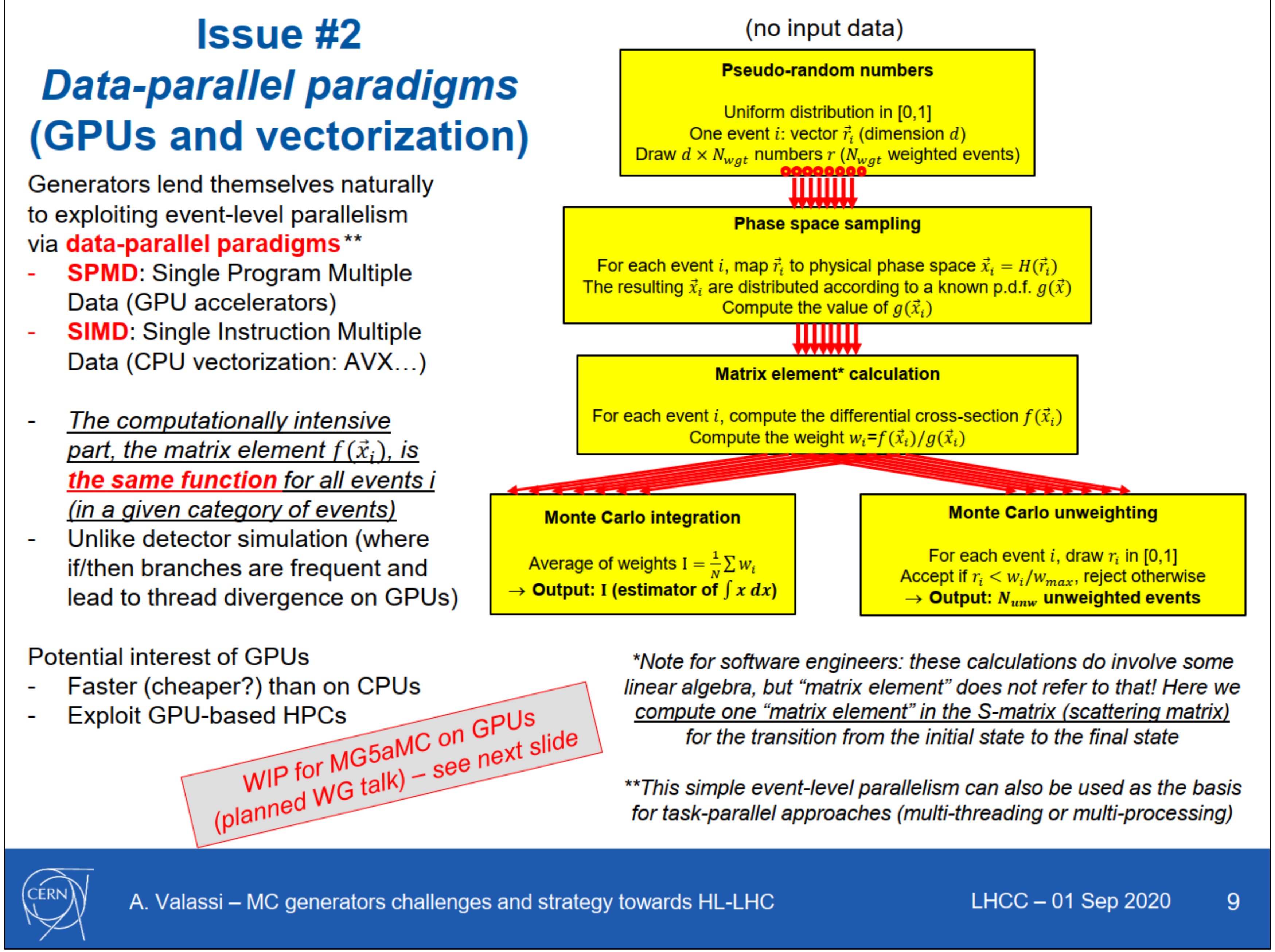
Matching and Merging prescriptions:
aMC@NLO, Powheg, KrkNLO,
CKKW, CKKW-L, MLM,
MEPS@NLO, MINLO, FxFx,
UNLOPS, Herwig7 Matchbox..

Hadronization and Parton Showers:
Pythia, Herwig, Ariadne…

- Software (and theory) diversity is good for physics
  - It provides cross-checks and healthy competition
- But it complicates the definition of an R&D strategy
  - Many software packages to optimize (and maintain!)
  - Prioritization (“profiling”): is there a CPU “hotspot”?

A. Valassi – MC generators challenges and strategy towards HL-LHC   LHCC – 01 Sep 2020   7

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



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

(no input data)

Pseudo-random numbers

Uniform distribution in [0,1]
One event i: vector $\vec{r}_i$ (dimension d)
Draw $d \times N_{wgt}$ numbers r (N_{wgt} weighted events)

Phase space sampling

For each event i, map $\vec{r}_i$ to physical phase space $\vec{x}_i = H(\vec{r}_i)$
The resulting $\vec{x}_i$ are distributed according to a known p.d.f. $g(\vec{x})$
Compute the value of $g(\vec{x}_i)$

Matrix element* calculation

For each event i, compute the differential cross-section $f(\vec{x}_i)$
Compute the weight w_i=$f(\vec{x}_i)/g(\vec{x}_i)$

Monte Carlo integration

Average of weights $\mathrm{I} = \frac{1}{N}\sum w_i$
→ **Output: I (estimator of $\int x\, dx$)**

Monte Carlo unweighting

For each event i, draw r_i in [0,1]
Accept if $r_i < w_i/w_{max}$, reject otherwise
→ **Output: N_{unw} unweighted events**

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)*

A. Valassi – MC generators challenges and strategy towards HL-LHC LHCC – 01 Sep 2020 9

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