Developments in software performance and portability for Madgraph5_aMC@NLO

Taylor Childers
Walter Hopkins
Nathan Nichols

Argonne NATIONAL LABORATORY

Laurence Field
Stephan Hageboeck
Stefan Roiser
David Smith
Andrea Valassi

CERN

Olivier Mattelaer

UCL Université catholique de Louvain

ICHEP, Bologna, 8th July 2022

https://agenda.infn.it/event/28874/contributions/169193

Outline

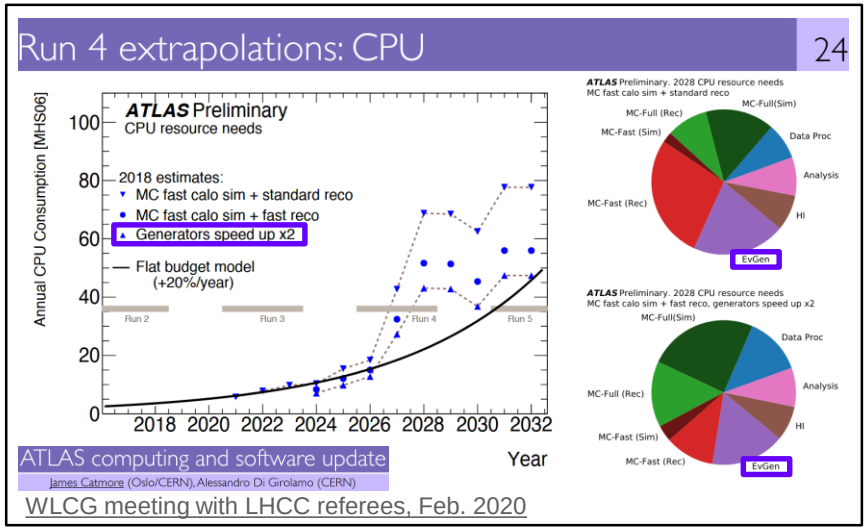
- Introduction
  - Monte Carlo generators in WLCG computing
  - Madgraph5_aMC@NLO (MG5aMC) and the madgraph4gpu project
  - Monte Carlo matrix element generators and data parallelism

- Results and outlook in three main areas of development
  (1) ME calculation in the 'cudacpp' implementation (C++ with vectorization on CPU, CUDA on Nvidia GPUs)
  (2) ME calculation in C++ portability frameworks (Alpaka, Kokkos, Sycl on CPUs and on Nvidia/AMD/Intel GPUs)
  (3) Integration of C++ based ME calculations into the Madevent Fortran framework

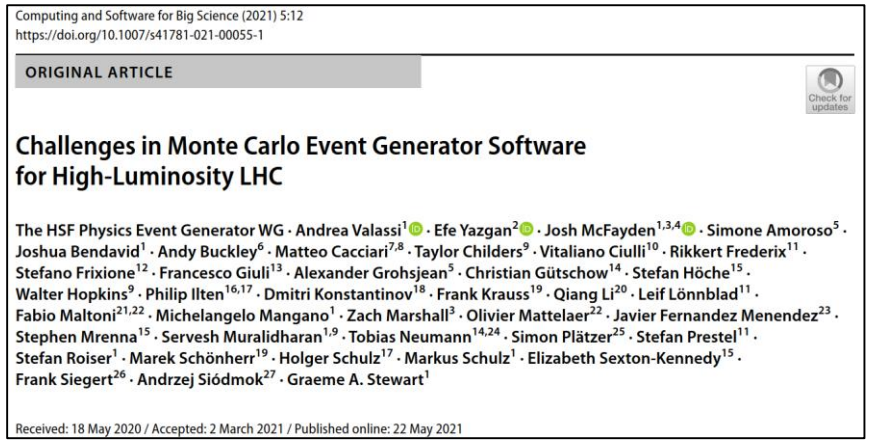
- Conclusions

Motivation: Monte Carlo Event Generators in WLCG computing

- LHC computing needs are predicted to outpace resource growth on HL-LHC timescales
  - Need aggressive R&D to improve software efficiency and port it to new architectures and resources
  - *GPUs increasingly important, in site clusters but also HPC centres* (already used opportunistically in WLCG)
  - Performance portability frameworks enable use of new systems without writing multiple software versions



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

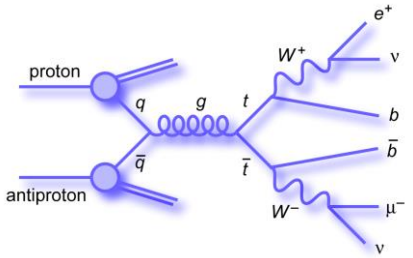
The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · 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, the essential 1st step in simulation, use 10-20% of ATLAS/CMS WLCG CPU budget
  - Many ways to speed up their performance – see the HEP Software Foundation (HSF) Generator WG review
  - *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!
  - SM and BSM, LO and NLO, integration with PDF and loop libraries...
  - Matrix Element (ME) calculations, merging of multi-jet final states, NLO matching of MEs and Parton Showers (PS)...

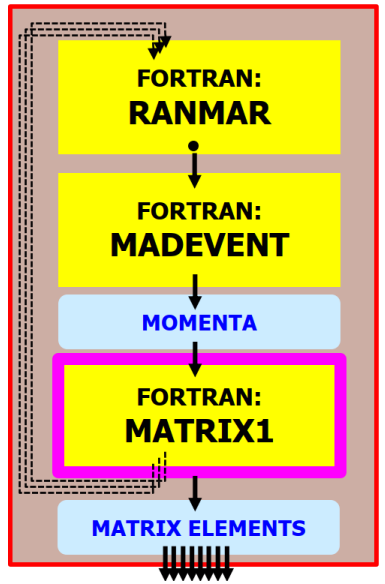


JHEP
Published for SISSA by Springer
Received: May 20, 2014
Accepted: June 25, 2014
Published: July 17, 2014

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

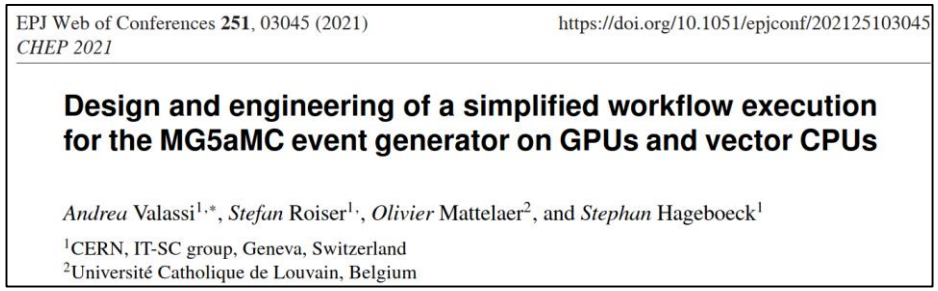


- MG5aMC production version in Fortran
  - *Software outer shell: Madevent*
    - A Fortran/Python/bash framework for phase space random sampling, integration and unweighted event generation
  - *Software inner core: ME calculation code*, automatically generated for each physics process
    - Production version in Fortran (but simpler, non optimized versions exist also in Python and C++)
    - *Matrix Element calculations take 95%+ of the CPU time* for complex processes (e.g. $gg \to t\bar{t}ggg$)

MG5aMC and the madgraph4gpu project

- *madgraph4gpu: speed up ME calculation in MG5aMC* on modern hardware (GPUs and vector CPUs)
  - Collaboration of theoretical/experimental physicists with software engineers – born in the HSF generator WG
  - It would not be possible without Olivier Mattelaer (MG5AMC co-author and current main maintainer) !

- Previous results were presented at vCHEP2021 (May 2021):
  (1) Only a simple $e^+e^-\to\mu^+\mu^-$ process, hardcoded one-off CUDA/C++
  (2) In C++ with vectorization for CPUs, in CUDA only for Nvidia GPUs
  (3) Only a standalone application (not usable by the experiments)


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

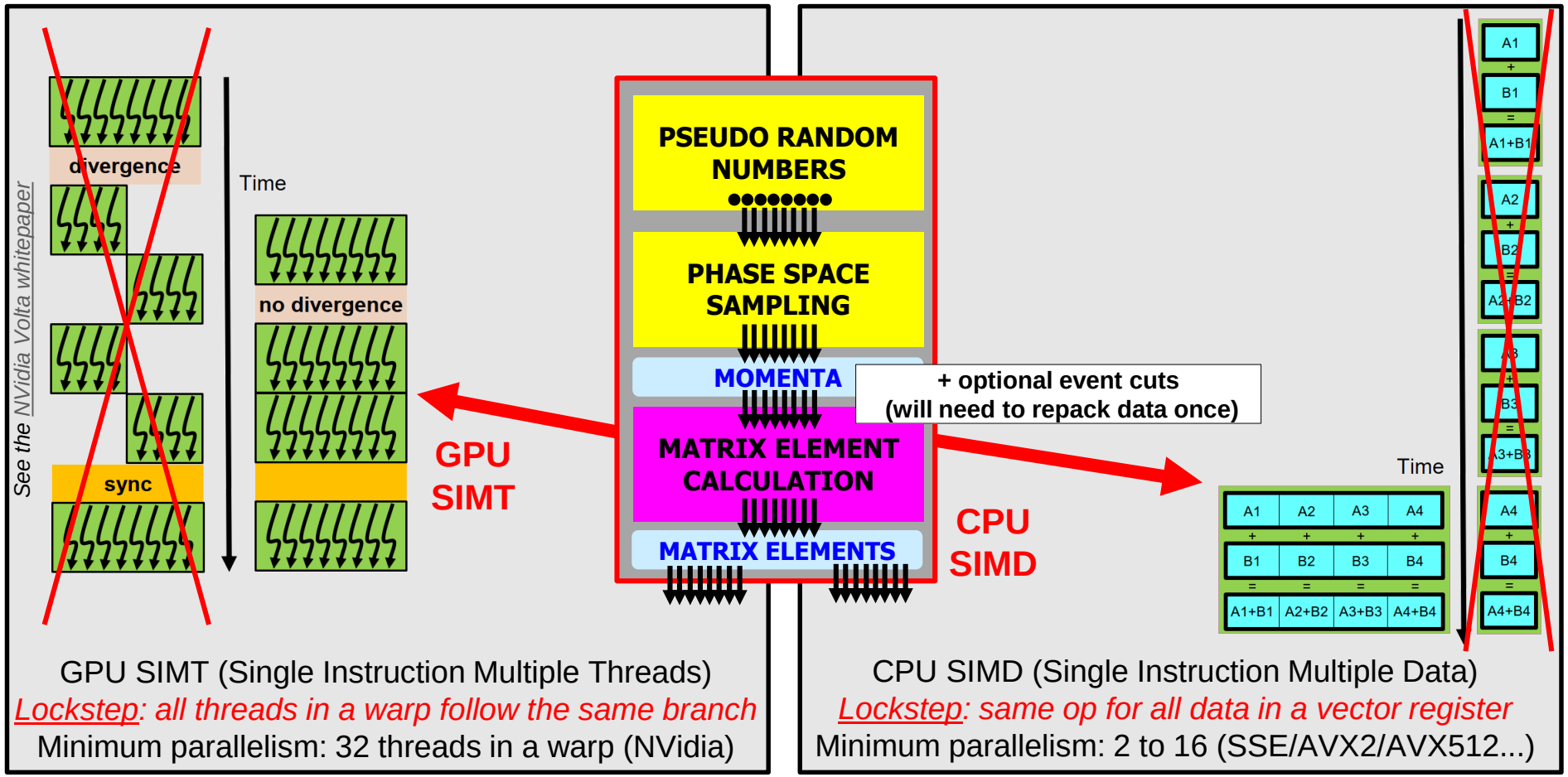
https://doi.org/10.1051/epjconf/202125103045

- *Two main goals for our current efforts in 2022*
  - *Release MG5AMC for LO (no NLO yet!) event generation in ATLAS/CMS (CPU SIMD speedups and GPU port)*
  - *Gain experience for the HEP software community on the usefulness of portability frameworks (PFs)*

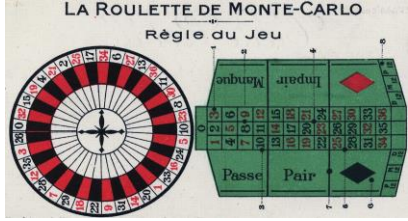
- Main new progress since May 2021:
  (1) Code generation plugins instead of one-off code: performance results for complex $gg\to t\bar{t}ggg$ processes
  (2) Additional implementations with PFs (Alpaka, Kokkos, Sycl), e.g. also for AMD and Intel GPUs
  (3) Integration of CUDA/C++ ME calculation into Madevent: cross sections done, event generation almost done

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)



Aside – Monte Carlo's: what about branching?

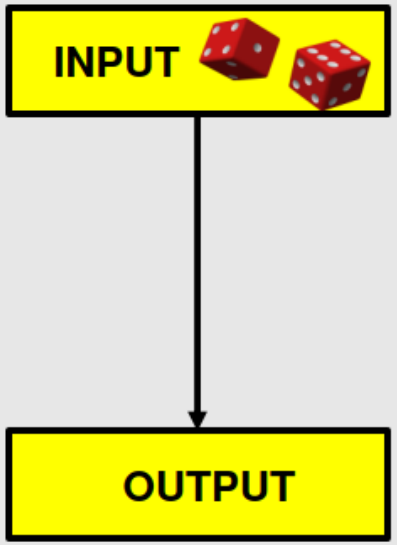


- *Monte Carlo methods are based on drawing (pseudo-)random numbers*: a dice throw

- From a software workflow point of view, these are used in *two rather different cases*:

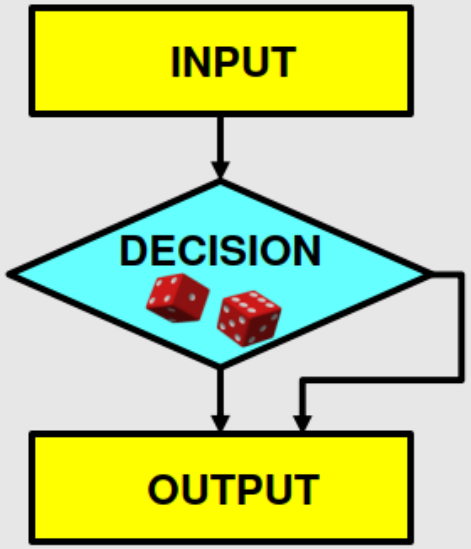
MC SAMPLING (within one channel)

Physics generators:
- 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



Stochastic branching
Bad for SIMT/SIMD

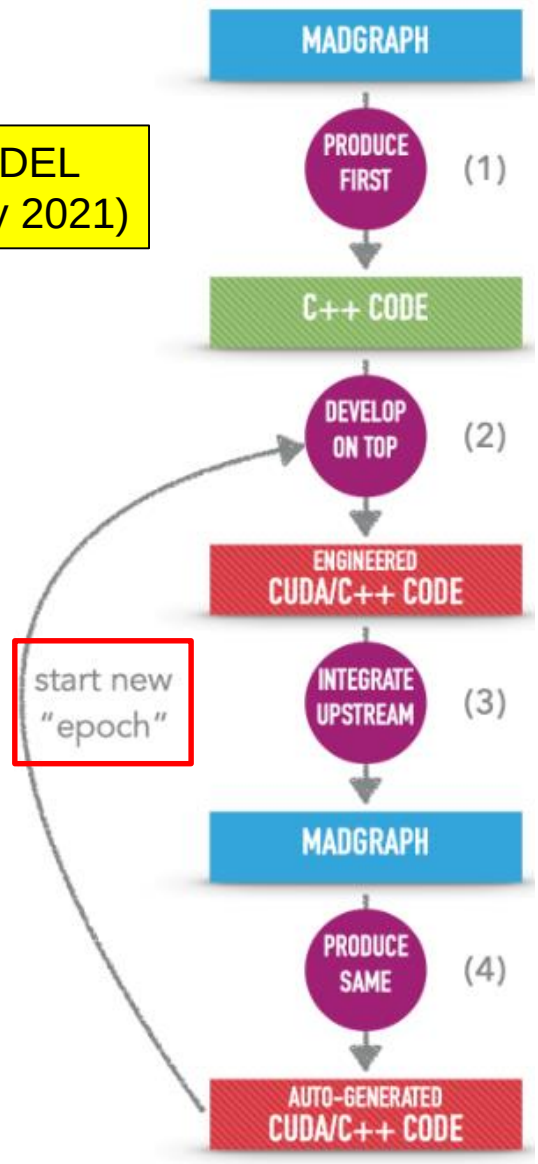
MC DECISIONS

Physics generators:
- MC sampling channel
- MC unweighting (accept/reject)
- Parton showers (PS)
- Fragmentation
- Particle decays (to what?)

MC detector simulation
- Particle/matter interaction (when? how?)
- Particle decays (when?)

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

Matrix Element (ME) calculation in cudacpp: results

(1) First line of development: the “cudacpp” plugin to calculate MEs in C++ (CPUs) or CUDA (GPUs)

Single code base for C++ and CUDA (with #ifdef’s): original development, currently the most advanced

Exploit SIMD vectorization through explicit Compiler Vector Extensions (gcc, clang, icpx)

Implementation ($gg \to t\bar{t}gg$)	MEs/second Double	MEs/second Float
1-core MadEvent Fortran scalar	3.96E3 (x2.2)	---
1-core Standalone C++ scalar	1.84E3 **(x1.00)**	1.80E3 (x0.98)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	3.36E3 (x1.8)	6.60E3 (x3.6)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	6.86E3 (x3.7)	1.31E4 (x7.1)
1-core Standalone C++ “256-bit” AVX512 (x4 doubles, x8 floats)	7.68E3 **(x4.2)**	1.41E4 **(x7.7)**
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	6.52E3 (x3.5)	1.32E4 (x7.2)
Standalone CUDA NVidia V100S-PCIE-32GB (TFlops*: 7.1 FP64, 14.1 FP32)	4.89E5 **(x270)**	9.27E5 **(x500)**

Intel Silver 4216 CPU (CERN)
Poor AVX512/zmm results ⇔ One FMA unit?

Helicity recycling (different/faster algorithm)

Implementation ($gg \to t\bar{t}gg$)	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	2.39E3 **(x1.00)**	2.50E3 (x1.05)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	4.59E3 (x1.9)	9.42E3 (x3.6)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	1.06E4 (x4.4)	2.15E4 (x9.0)
1-core Standalone C++ “256-bit” AVX512 (x4 doubles, x8 floats)	1.15E4 (x4.8)	2.28E4 (x9.5)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	1.96E4 **(x8.2)**	4.03E4 **(x16.9)**

Intel Gold 6148 CPU (Juwels Cluster HPC)
Better AVX512/zmm results ⇔ Two FMA units?

(one single thread)

Main new results since vCHEP2021:

- Backport to code generation
  Speedups previously reported for ee_mumu now ~confirmed for gg_ttgg
  - CPUs/SIMD: x4 double, x8 float
  - GPUs/V100: x270 double, x500 float
    (could do better, high register pressure)
- *Extra x2 from AVX512 on Intel Gold:*
  Achieve theoretical limit of speedup:
  x8 double, x16 float on AVX512 CPU
- New features for MadEvent integration

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){
```

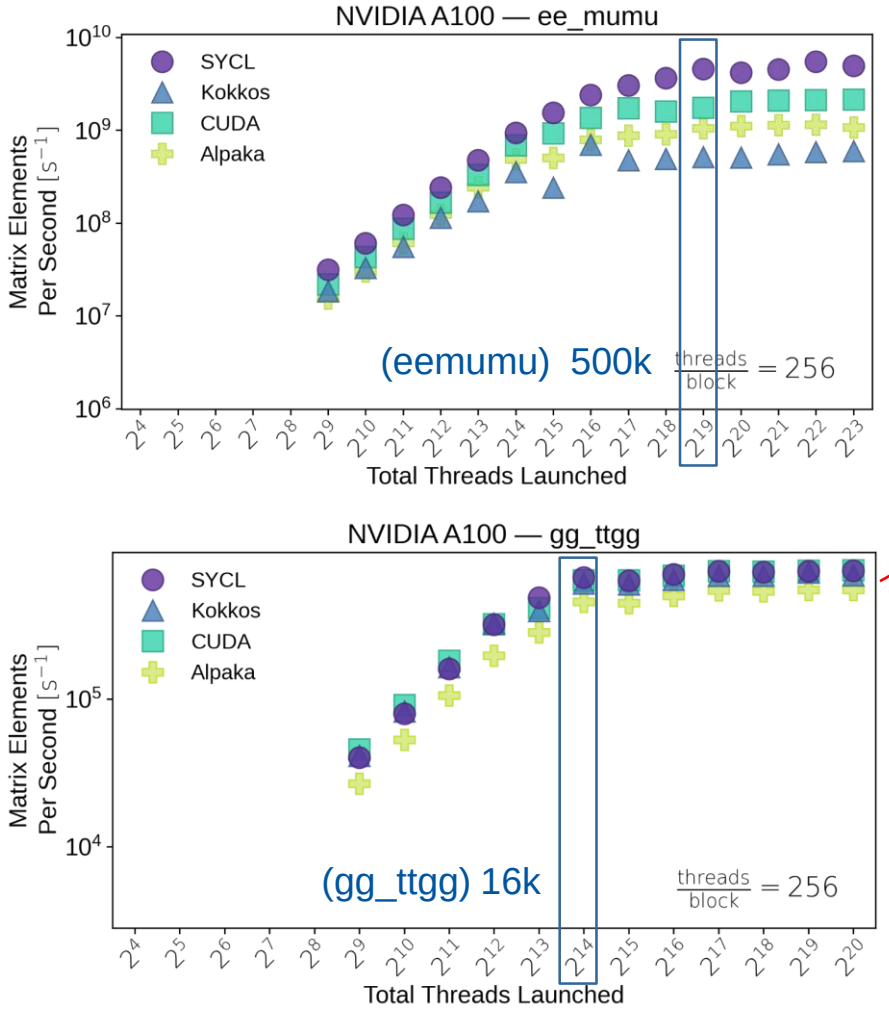
Kokkos example of Memory Management

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

ME calculation in PFs: GPU results (Nvidia A100)

Throughput scaling (threads, blocks) for a simple $e^+e^-\to\mu^+\mu^-$ process and a complex $gg\to t\bar{t}gg$ process

(note: this is an older version of the code with respect to the results shown earlier for cudacpp alone)

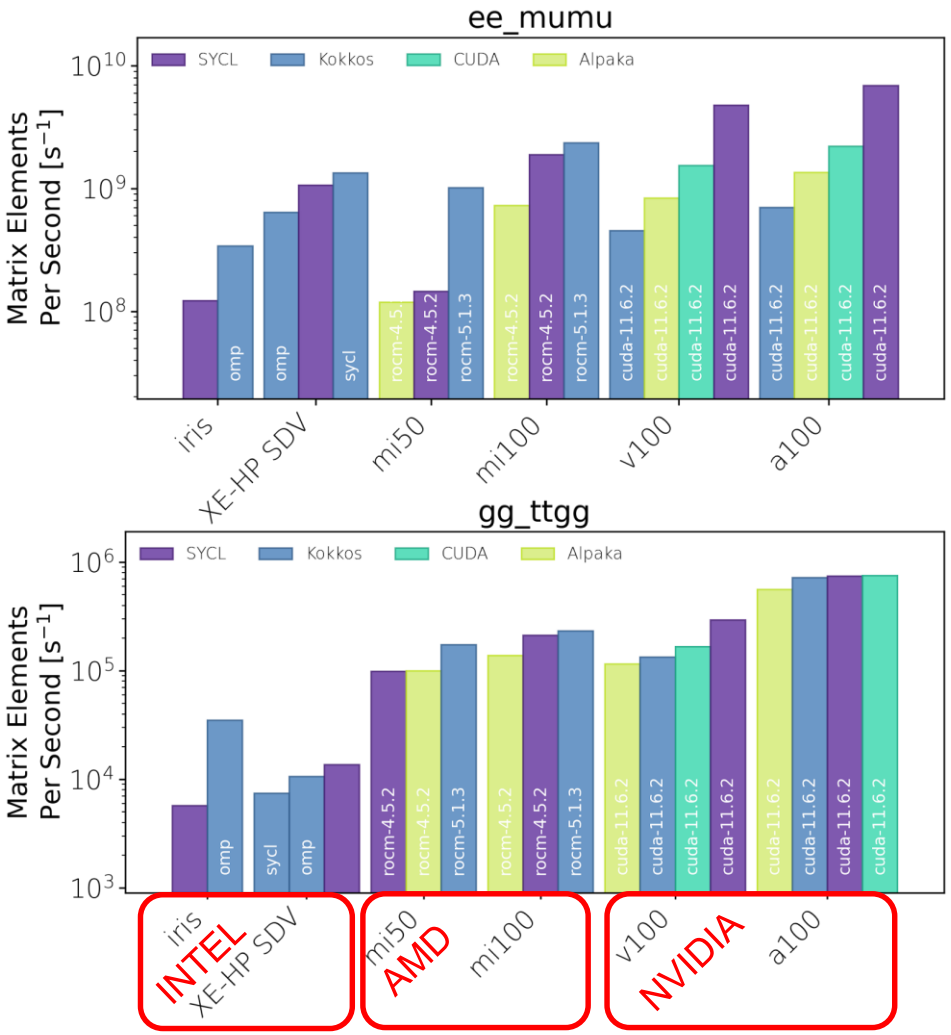


- This and the next slide show both ee_mumu and gg_ttgg for comparison, but *please focus only on the gg_ttgg results!*
  - The ME calculations in ee_mumu are extremely simple: the overhead of CPU-GPU memory copies on total MEs/s is huge (and maybe was handled differently in the 4 implementations?)
- Good news 1: for gg_ttgg, all four implementations look similar!
  - The benefit of direct CUDA over a PF is limited, if any at all
- En passant, keep in mind this for later: you need at least 16k "events per GPU grid" to fill up a V100 or A100 with gg_ttgg+
  - Simpler processes need even more, e.g. 500k for ee_mumu

ME calculation in PFs: GPU results (Nvidia, Intel, AMD)

Maximum throughput for a simple $e^+e^-\to\mu^+\mu^-$ process and a complex $gg\to t\bar{t}gg$ process

(note: this is an older version of the code with respect to the results shown earlier for cudacpp alone)



- Again, *please focus only on the gg_ttgg results!*
- Good news 1: for gg_ttgg, all four look similar on Nvidia!
  - The benefit of direct CUDA over a PF is limited, if any at all
- Good news 2: PFs also work on AMD and Intel GPUs!
  - Out of the box, with a single implementation

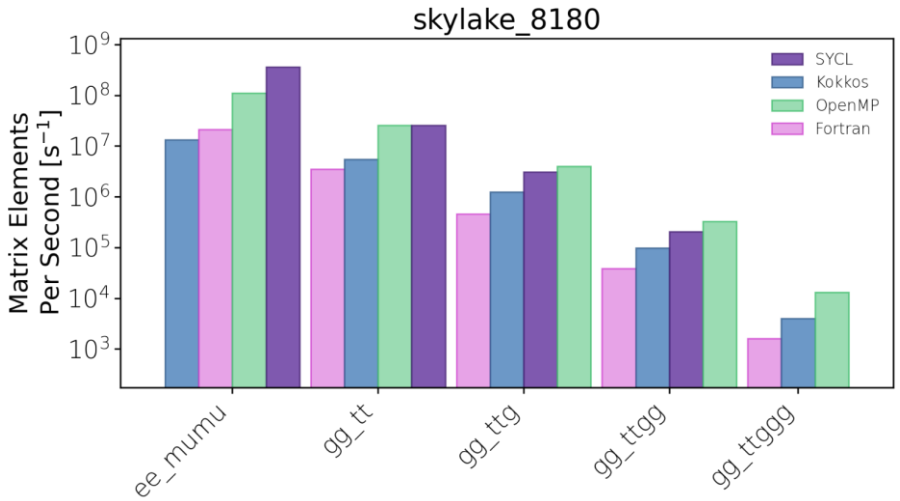
(There is no Alpaka on Intel in the plots because we use Cupla: we should move to using native Alpaka)

Xe-HP is a software development vehicle for functional testing only. It is currently used at Argonne and at other customer sites to prepare their code for future Intel data centre GPUs

ME calculation in PFs: CPU results *(preliminary! need systematic study)*

Maximum throughput for five processes, from simple ($e^+e^-\to\mu^+\mu^-$) to more complex ($gg\to t\bar{t}ggg$)

(note: this is an older version of the code with respect to the results shown earlier for cudacpp alone)



- CPUs have two very different parallelisms we can exploit:
  - Many floats/doubles per vector register: vectorization (SIMD)
  - Many physical/virtual cores: multi-threading (or many processes!)
- *NB: this plot is comparing apples to oranges and to peaches!*
  - Fortran: one single thread, no vectorization
  - Kokkos: internal multithreading? limited auto-vectorization?
  - SYCL: internal multithreading? limited auto-vectorization?
  - cudacpp: OpenMP multithreading, *explicit vectorization (CVE)*
    - The OMP multithreading in the cudacpp plugin is known to be suboptimal and will be reengineered (probably with std::thread instead)

On CPUs, for the moment, it seems better to use ad-hoc developments as in cudacpp, than rely on PFs

(NB: you may replace OMP by many applications in parallel, but you must do low-level coding to get a factor x4 or more from SIMD)

Matrix Element (ME) calculation in cudacpp and PFs: outlook

Short term (end 2022?)

- (Nvidia GPUs) Further improve CUDA performance with smaller kernels
  - Exploit tensor cores for color algebra in cudacpp? Would tensor cores be supported by PFs?
  - Finer grained strategy for distributing work on the GPU(s)? Multi-GPU support?
- (AMD/Intel GPUs) Add direct HIP to cudacpp implementation, in parallel advance in PF implementations
- (CPUs, multithreading) Replace OpenMP by std::thread; systematic thread scaling studies in cudacpp and PFs
  - Containerize the standalone application and collaborate on scaling studies with the HEPiX benchmarking WG
- (CPUs, vectorization) Systematic vectorization studies in PF implementations
- (CPUs, GPUs) Numerical precision studies: stress tests of -O3 and fast math (our default assumption...)

Medium term (2023+)

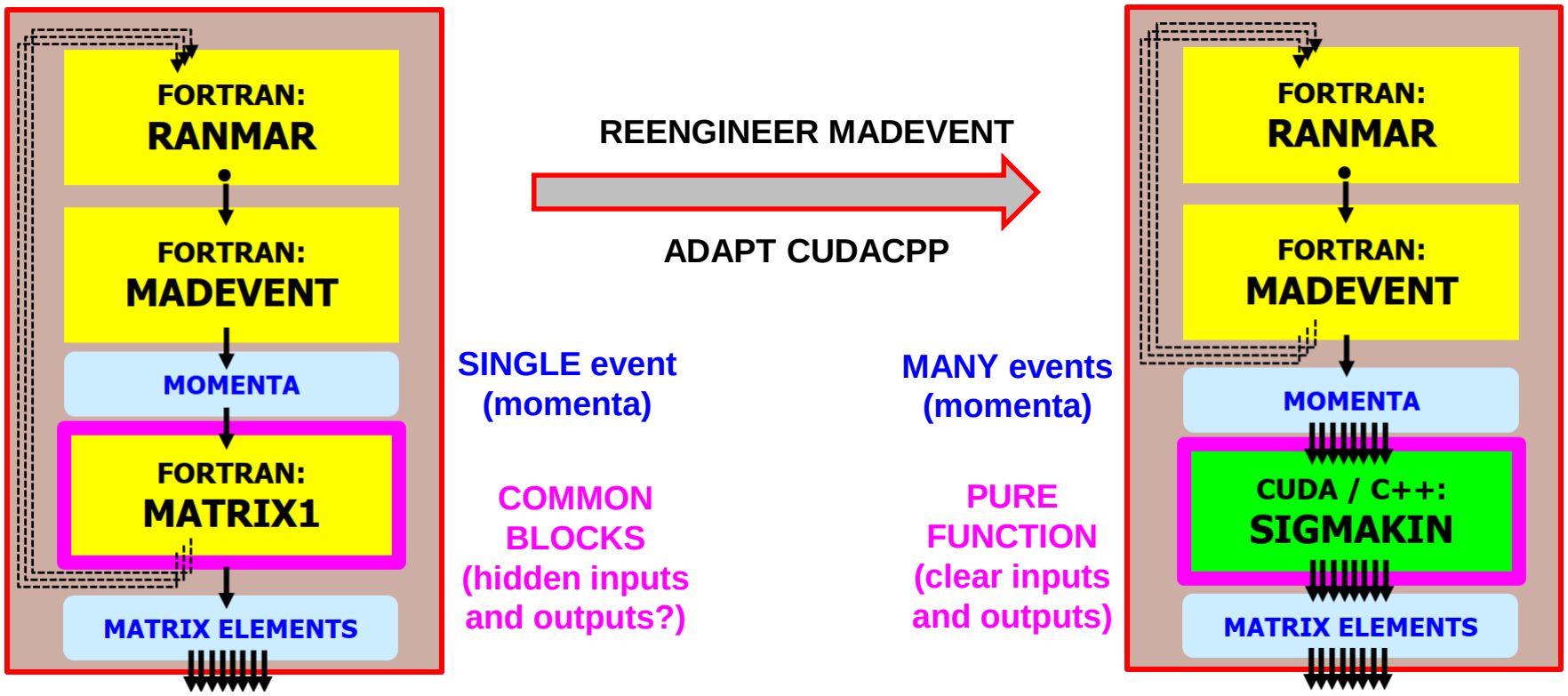
- *(CPUs, GPUs) Implement helicity recycling in cudacpp (additional x2-3 algorithmic speedup, now only in Fortran)*
- *(CPUs, GPUs) Handle NLO: loops and matching to PS*
- *(CPUs, GPUs) Numerical precision studies: would float be enough? (additional x2 speedup over double)*

Matrix element integration in MadEvent: overview

(3) Third line of development: replacing Fortran by cudacpp MEs in Madevent *(keep the 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 (need 16k+ per GPU grid ⇒ *huge arrays in CPU memory!*)
- 2. Debugging the issues caused by hidden inputs and outputs, largely coming from Fortran common blocks



Matrix element integration in MadEvent: results

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

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

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

CERN: Intel Silver 4216 + Nvidia V100

gg → ttggg		[seconds] Overall = MadEvent + MEs	[MEs/second]
6k events	FORTRAN	93.65 = 4.16 + 89.49	7.19e+01
	CPP/none	111.50 = 4.89 + 106.62	6.03e+01
	CPP/sse4	62.16 = 4.50 + 57.66	1.12e+02
	CPP/avx2	33.78 = 4.26 + 29.52	2.10e+02
	CPP/512y	30.66 = 4.22 + 26.44	2.43e+02
	CPP/512z	28.36 = 4.34 + 24.02	2.68e+02
	CUDA/32	63.72 = 5.34 + 58.38	1.10e+02
800k events	CUDA/8192	639.20 = 527.37 + 111.83	7.40e+03

Juwels: Intel Gold 6148

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

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

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

Matrix element integration in MadEvent: outlook

Very short term (Q3 2022 – alpha release for the experiments)

- Implement event-by-event random choice of colors and helicities in cudacpp (goal: same LHE files!)
- Cross-check the few last details (pdfs, user parameters...)

Short to medium term (end 2022 – 2023)

- Reduce overhead from scalar Madevent framework (goal: overall speedups closer to ME speedups)
  - This is currently the bottleneck preventing higher throughputs for the overall workflow using GPUs
  - One possible option: heterogeneous workflow (multithreaded Madevent on CPU, parallel ME on GPU)?
- Reduce number of Fortran arrays in Madevent (goal: lower CPU memory, allow larger GPU grids beyond 8k)

Conclusions

- *ALL* Matrix Element Generators are perfect fits to exploit CPU vectorization/SIMD and GPUs
  - Lockstep parallelism in MEGs much easier to exploit than in detector simulation (Geant4, stochastic branching)
- **An alpha release of MG5aMC *for LO* with GPU ports and CPU speedups from SIMD is imminent**
  - Cross section calculation is ready; a few details to fix for unweighted event generation (random color/helicity...)
- On Intel Gold CPUs, AVX512 C++ is x8 faster than scalar C++ for ME calculations (in double precision)
  - A slightly lower speedup ~x6 holds for the full Madevent + ME workflow (Amdahl's law, as Madevent is scalar)
  - Overall speedup ~x5 compared to Fortran (comparing to the old Fortran release without helicity recycling)
  - An additional x2-3 algorithmic speedup will come through helicity recycling (not yet in cudacpp)
- On GPUs, much larger O(300+) speedups may be achieved for the ME calculation
  - But we must reduce the scalar component in Fortran MadEvent to see those in the full workflow (Amdahl's law)
- Additional x2 speedups may be achieved on CPUs and GPUs by moving from double to single precision
- Portability Frameworks work well for us! Simplify development with a single code for many GPU flavors
  - Similar performance to direct CUDA on Nvidia GPUs; we may also run out of the box on AMD and Intel GPUs

BACKUP SLIDES

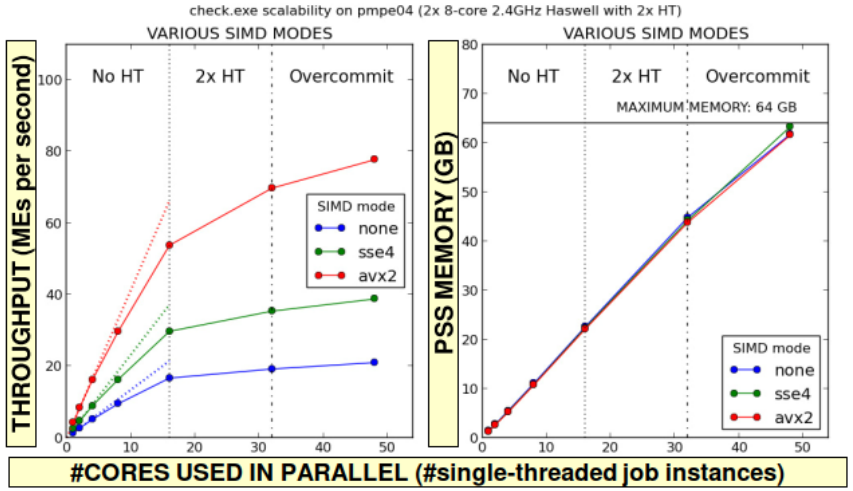
Acknowledgements

- 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.

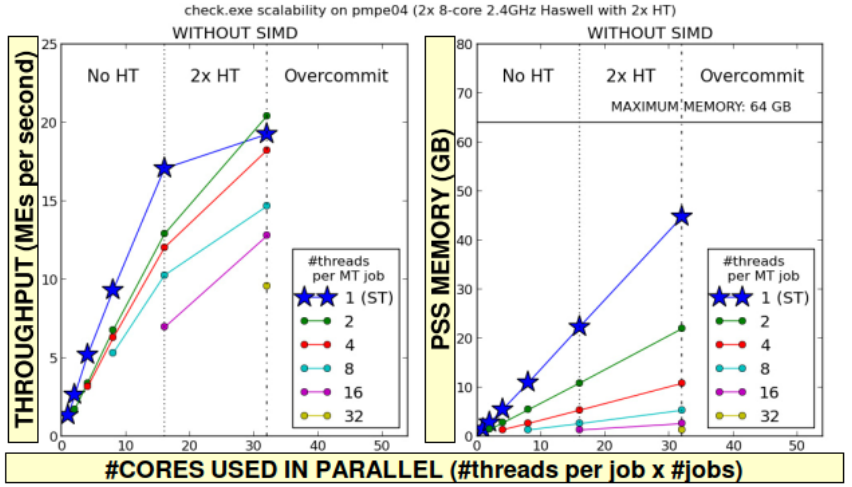
- We gratefully acknowledge the use (under PRACE proposal PRACE-DEV-2022D01-022) of the JUWELS supercomputer and other computing resources provided and operated by the Jülich Supercomputing Centre at Forschungszentrum Jülich.

CPU throughput plots – SIMD + multi-core

- *Two different throughput speedup factors multiply each other: SIMD and multi-core*
  - SIMD: fewer instructions per processor (e.g. in AVX2 each instruction applies to 4 doubles)
  - Multi-core: many cores used in parallel (e.g. multiple jobs, multi-threading, multi-processing)



Multiple instances of single-threaded MG5aMC
Combine SIMD and multi-core speedup
Memory proportional to number of cores used



Prototype of OpenMP multi-threaded MG5aMC
Trivial coding (one pragma!), but suboptimal/unstable
Much lower memory (~proportional to number of jobs)
Will probably reimplement this using std::thread

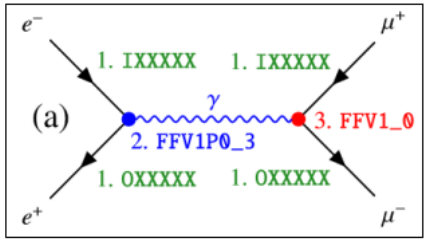
CUDA/C++: ME code example (complex number scalar/vector)

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

- *CUDA*: scalar complex →
- *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
Soon to be automatically generated

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

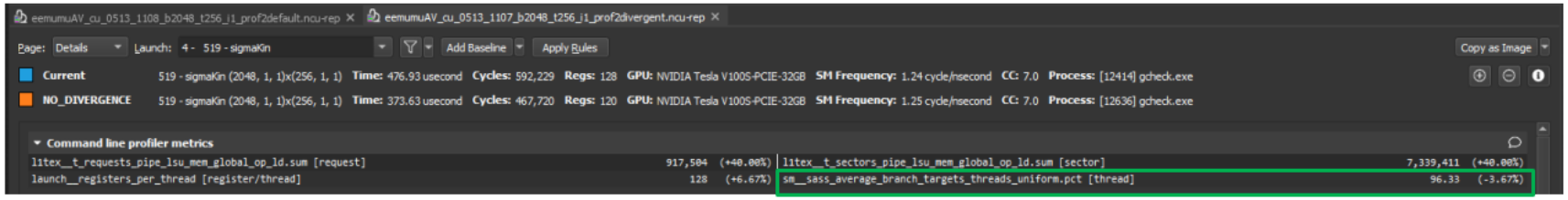
```
inline
cxtype_v operator+( const cxtype_v& a, const cxtype_v& b )
{
  return cxmake( a.real() + b.real(), a.imag() + b.imag() );
}
```

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
```

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



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

EVEN MORE BACKUP SLIDES

Argonne's Joint Laboratory for System Evaluation (JLSE)

We used JLSE systems to run all performance tests described for Alpaka/Kokkos/Sycl vs Cuda/OpenMP

NVidia A100 Nodes
AMD 7532 32c 2.4Ghz
DDR4-3200 256GB (8x32G DIMMs) RAM
1x Nvidia A100 40GB PCIe 4.0
Mellanox ConnectX-6 EDR

NVidia V100 Nodes
4x NVIDIA Tesla V100 SXM2 w/32GB HBM2
2x Intel Xeon Gold 6152 CPU 22c 2.10GHz
192GB RAM DDR4-2666
Mellanox ConnectX-5 EDR

Iris Nodes
Intel Xeon E3-1585 v5 CPU w/ Intel Iris Pro Graphics P580
4x 16GB DDR4-2666 SODIMMs (operating at DDR4-2133)
1GbE Onboard

Arcticus Nodes
2x Intel development GPU card (Codename XeHP_SDV)
2x Intel(R) Xeon Gold 6336Y CPU (48 physical cores total) 2.4Ghz
256GB: 16x 16GB DDR4 @ 3200
Mellanox ConnectX-6: EDR InfiniBand (100 Gbps)

Skylake Nodes
Intel S2600WF,
2x Intel Xeon Platinum 8180M CPU @ 2.50GHz
768GB RAM

AMD MI100 Nodes
2x AMD EPYC 7543 32c (Milan)
4x AMD MI100 32GB GPUs
Infinity Fabric
512GB DDR4-3200

AMD MI50 Nodes
Gigabyte G482-Z51
2x 7742 64c Rome
4x AMD MI50 32GB GPUs
Infinity Fabric
256GB DDR-3200 RAM

Build environment on JLSE (Sycl)

- We used JLSE systems to run all performance tests described here for Alpaka/Kokkos/Sycl

NVidia A100 Nodes
Intel oneAPI DPC++ (commit b9cb1d1247e2)
CUDA 11.6.2

Iris Nodes
Intel oneAPI DPC++ (NDA)

AMD MI100 Nodes
Intel oneAPI DPC++ (commit b9cb1d1247e2)
ROCM 4.5.2

NVidia V100 Nodes
Intel oneAPI DPC++ (commit b9cb1d1247e2)
CUDA 11.6.2

Arcticus Nodes
Intel oneAPI DPC++ (NDA)

AMD MI50 Nodes
Intel oneAPI DPC++ (commit b9cb1d1247e2)
ROCM 4.5.2

Skylake Nodes
Intel oneAPI DPC++ (2021.4.0)

Build environment on JLSE (Kokkos)

We used JLSE systems to run all performance tests described for Alpaka/Kokkos/Sycl vs Cuda/OpenMP

NVidia A100 Nodes
Kokkos 3.5.00
CUDA 11.6.2
g++ 9.4.0

NVidia V100 Nodes
Kokkos 3.5.00
CUDA 11.6.2
g++ 9.4.0

Iris Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

Arcticus Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

Skylake Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

AMD MI100 Nodes
Kokkos 3.5.00
ROCM 4.5.2

AMD MI50 Nodes
Kokkos 3.5.00
ROCM 4.5.2

Build environment on JLSE (Alpaka)

We used JLSE systems to run all performance tests described for Alpaka/Kokkos/Sycl vs Cuda/OpenMP

NVidia A100 Nodes
Kokkos 3.5.00
CUDA 11.6.2
g++ 9.4.0

NVidia V100 Nodes
Kokkos 3.5.00
CUDA 11.6.2
g++ 9.4.0

Iris Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

Arcticus Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

Skylake Nodes
Intel oneAPI DPC++ (NDA)
Kokkos (NDA)

AMD MI100 Nodes
Kokkos 3.5.00
ROCM 4.5.2

AMD MI50 Nodes
Kokkos 3.5.00
ROCM 4.5.2

Build environment on JLSE (Cuda and OpenMP)

We used JLSE systems to run all performance tests described for Alpaka/Kokkos/Sycl vs Cuda/OpenMP

NVidia A100 Nodes
CUDA 11.6.2
g++ 9.4.0

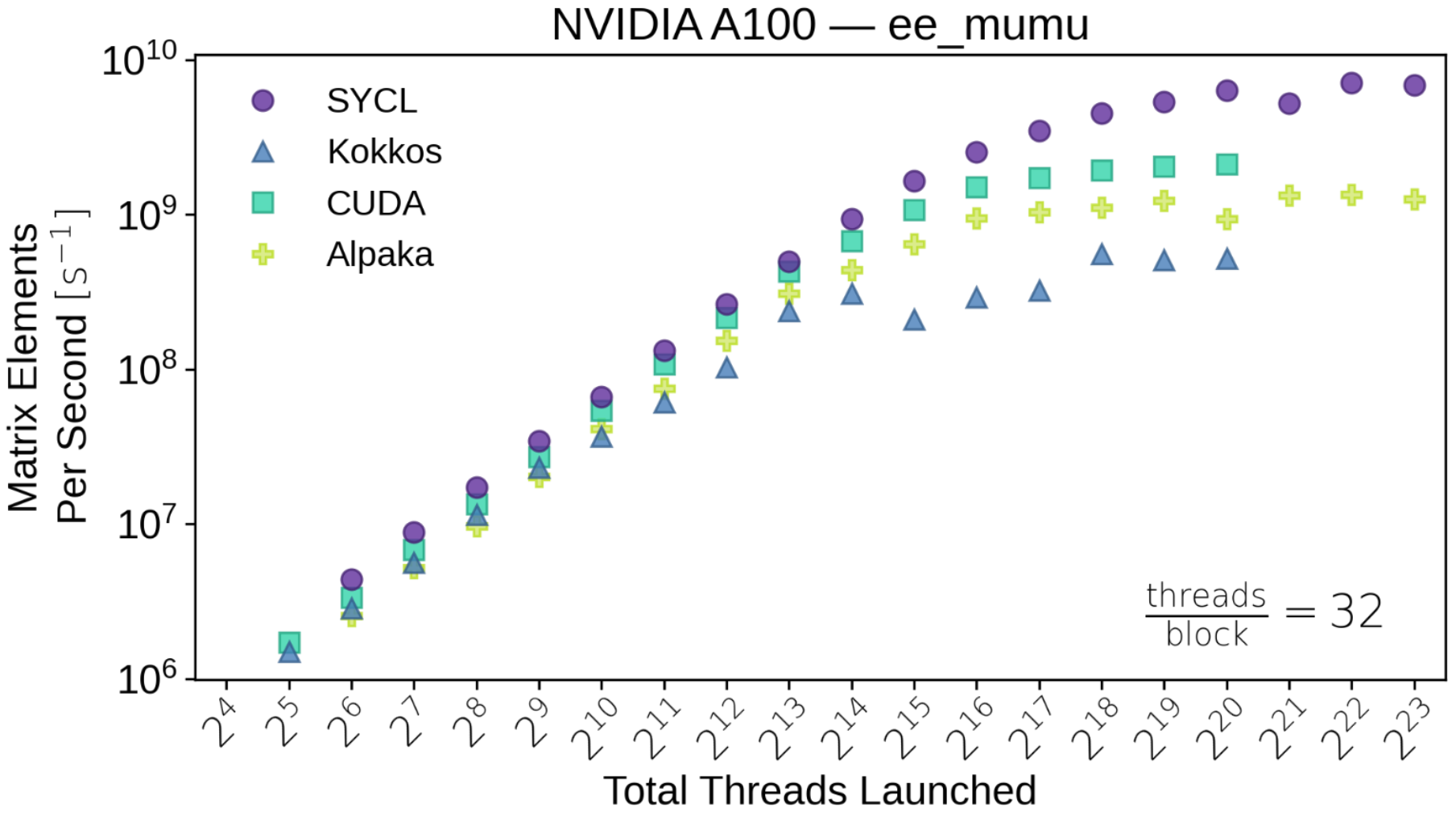
NVidia V100 Nodes
CUDA 11.6.2
g++ 9.4.0

Skylake Nodes
g++ 11.3.0
OMP_NUM_THREADS=56

Matrix Element (ME) calculation in PFs: GPU results

Thread and block scaling for a simple $e^+e^-\to\mu^+\mu^-$ process

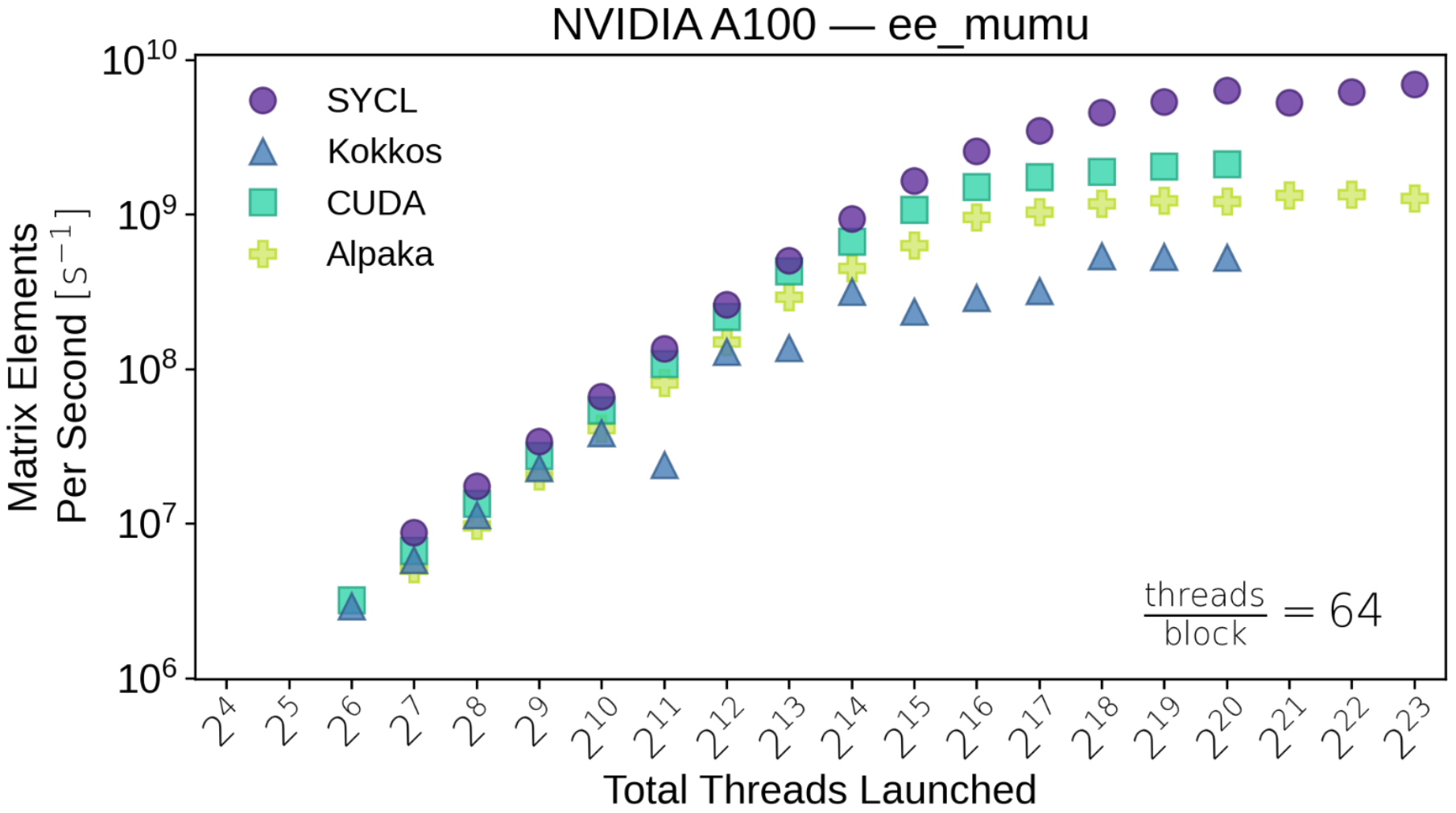
(note: this is an older version of the code with respect to the results shown earlier for cudacpp and complex $gg\to t\bar{t}ggg$ processes)



Matrix Element (ME) calculation in PFs: GPU results

Thread and block scaling for a simple $e^+e^-\to\mu^+\mu^-$ process

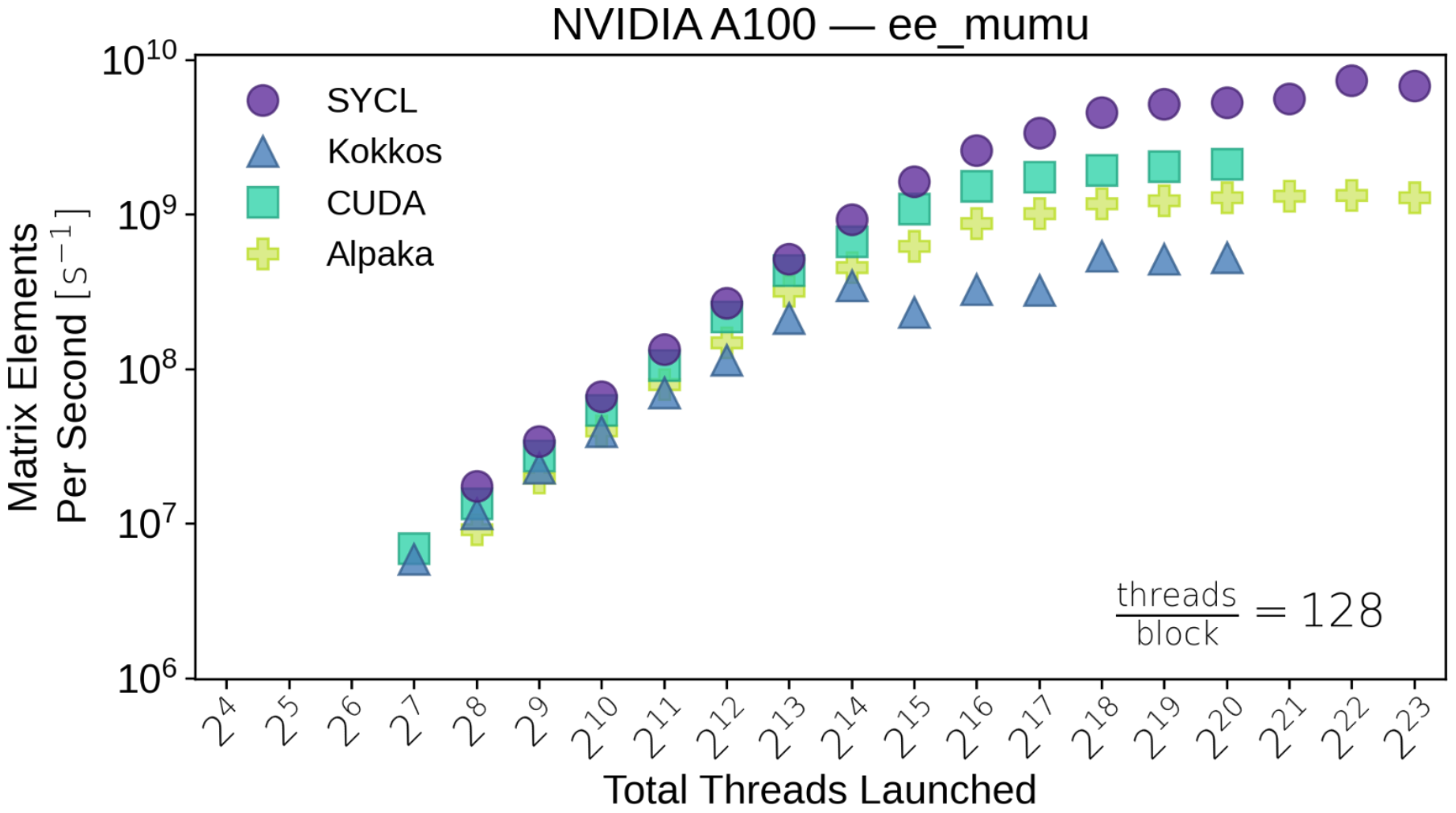
(note: this is an older version of the code with respect to the results shown earlier for cudacpp and complex $gg\to t\bar{t}ggg$ processes)



Matrix Element (ME) calculation in PFs: GPU results

Thread and block scaling for a simple $e^+e^-\to\mu^+\mu^-$ process

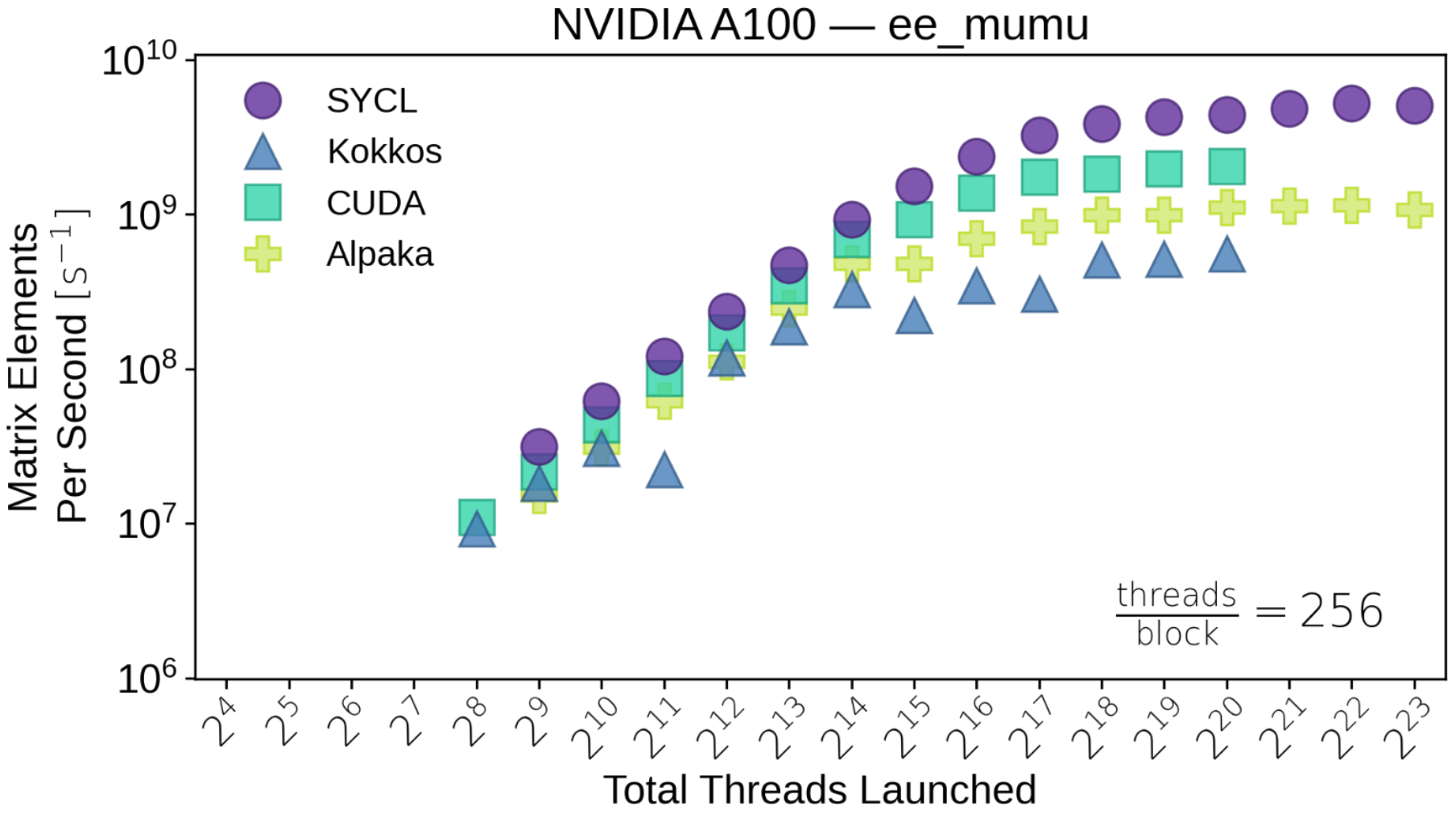
(note: this is an older version of the code with respect to the results shown earlier for cudacpp and complex $gg\to t\bar{t}ggg$ processes)



Matrix Element (ME) calculation in PFs: GPU results

Thread and block scaling for a simple $e^+e^-\to\mu^+\mu^-$ process

(note: this is an older version of the code with respect to the results shown earlier for cudacpp and complex $gg\to t\bar{t}ggg$ processes)



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)

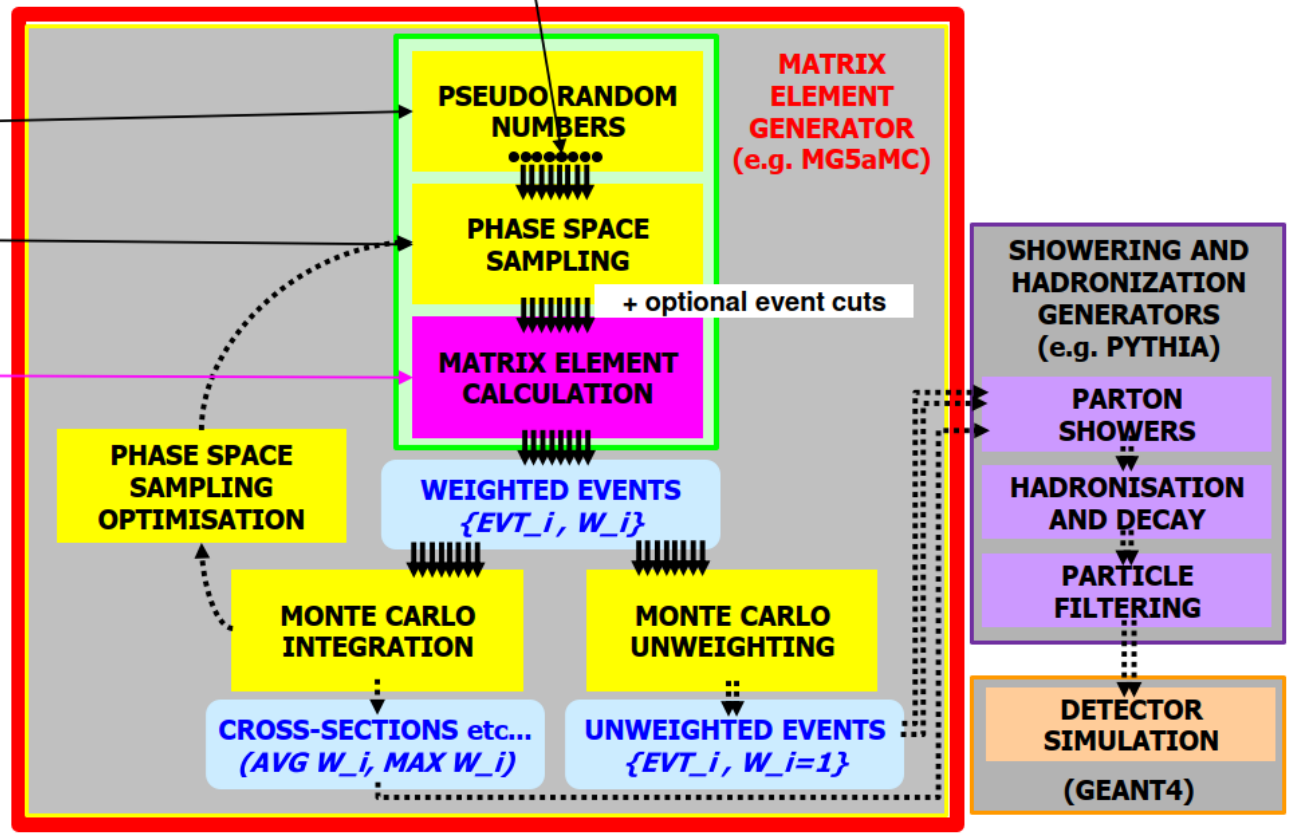
For each event:

1. Output: random numbers

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

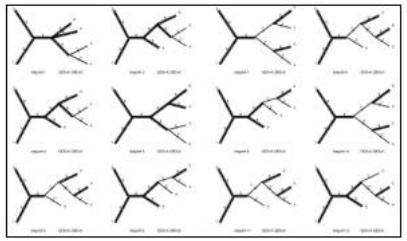
3. Input: particle 4-momenta
Output: Matrix Element (ME)
CPU BOTTLENECK

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

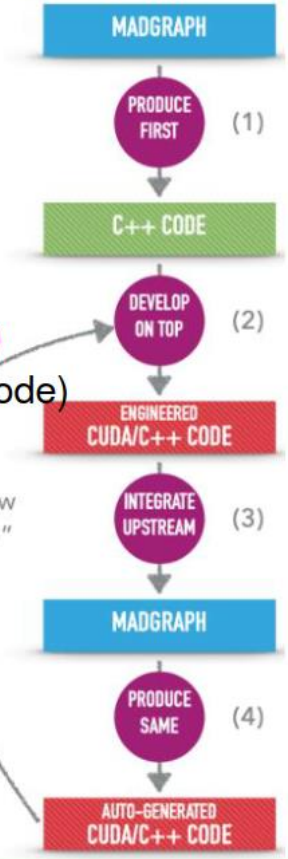


Code is auto-generated ⇒ Iterative development process

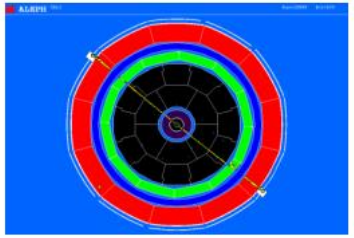
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Currently Fortran (default), C++, or Python
  - The more particles in the collision, the more Feynman diagrams and the more lines of code

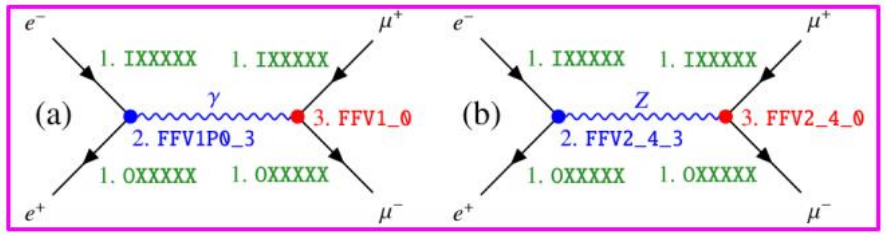


Process	*LOC*	*functions*	*function calls*
$e^+e^- \to \mu^+\mu^-$	776	8	16
$gg \to t\bar{t}$	839	10	22
$gg \to t\bar{t}g$	1082	36	106
$gg \to t\bar{t}gg$	1985	222	786



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

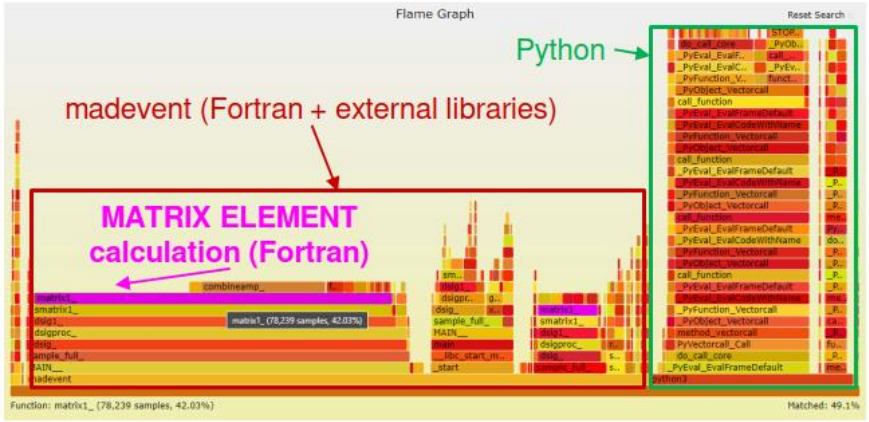




A complex outer shell – with a CPU-intensive core: the ME

- 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*

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-

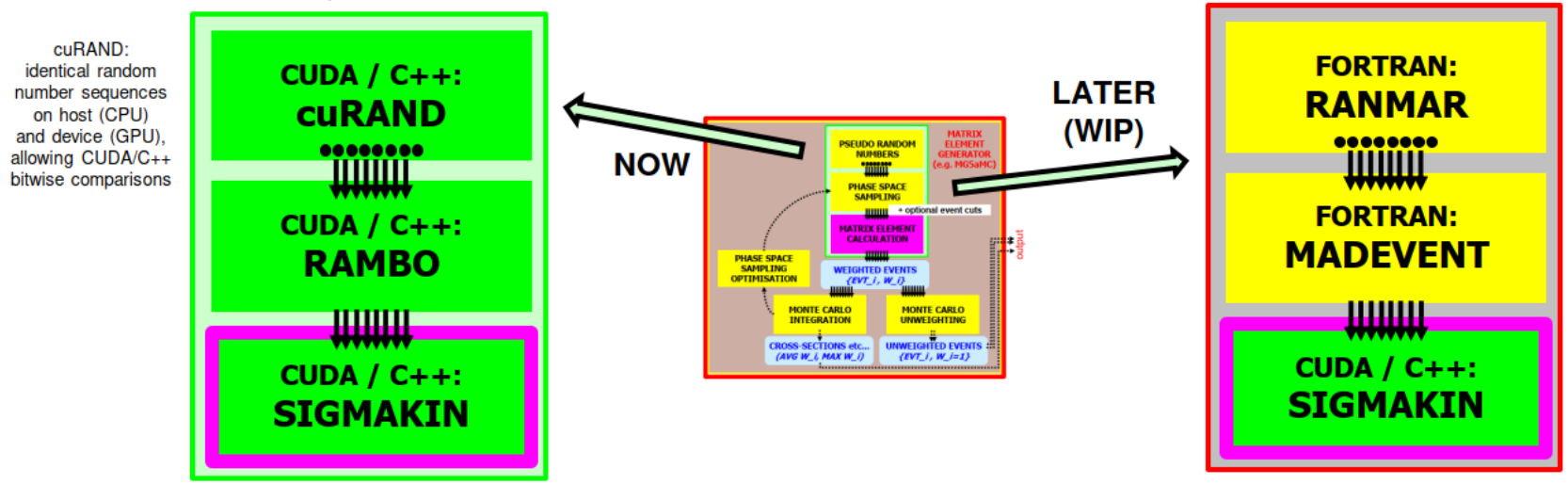
	$gg \to t\bar{t}$	$gg \to t\bar{t}gg$	$gg \to t\bar{t}ggg$
madevent	13G	470G	11T
matrix1	3.1G (23%)	450G (96%)	11T (>99%)

(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*

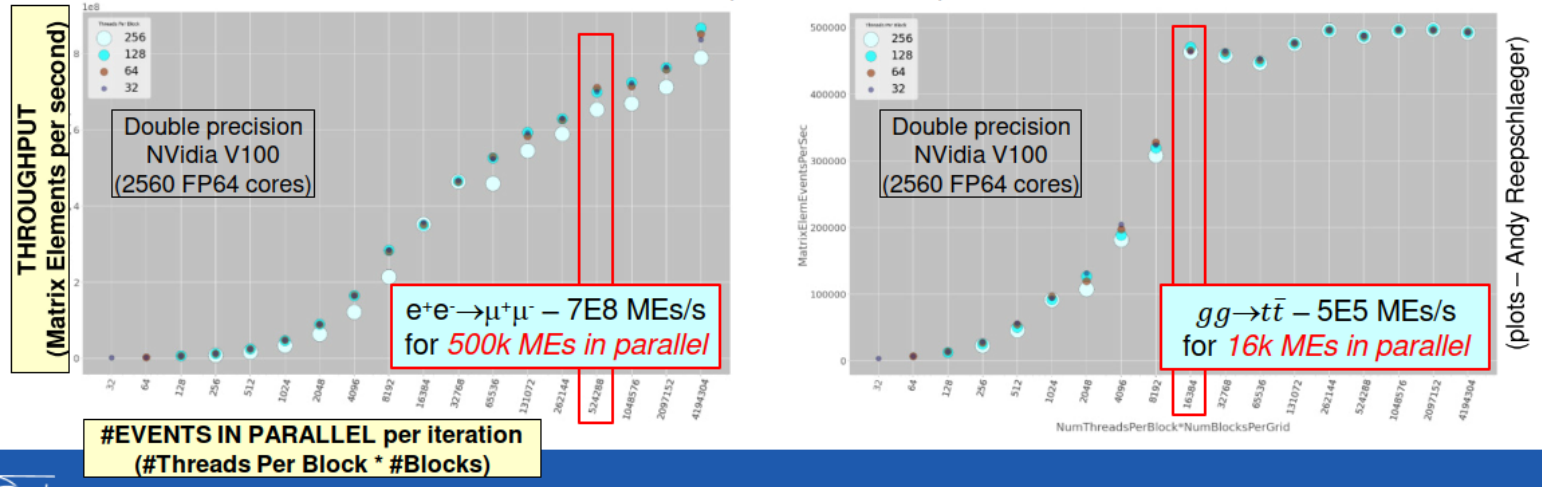


- Our WIP: *we plan to inject CUDA/C++ ME kernel into MadEvent/gridpack framework*
  - Fastest way to production – easier than rewriting MadEvent in CUDA/C++
  - Validated code/infrastructure, same user interface – discussed with experiments at HSF WG

Event-level parallelism in practice – coding and #events

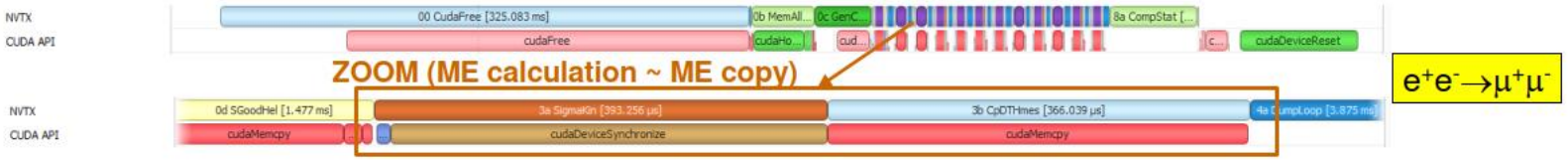
- Easier to code for GPU SIMT than for CPU SIMD: *CUDA code was faster to prototype*
- CUDA (GPU) implementation
  - For SIMT, event loop is "orthogonal": one thread = one event *(GPU thread ID ↔ event ID)*
  - For SIMT, SOA memory layouts are beneficial (coalesced access), but not strictly essential
- 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



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)



- 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*

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 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!

- 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"?

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…

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

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

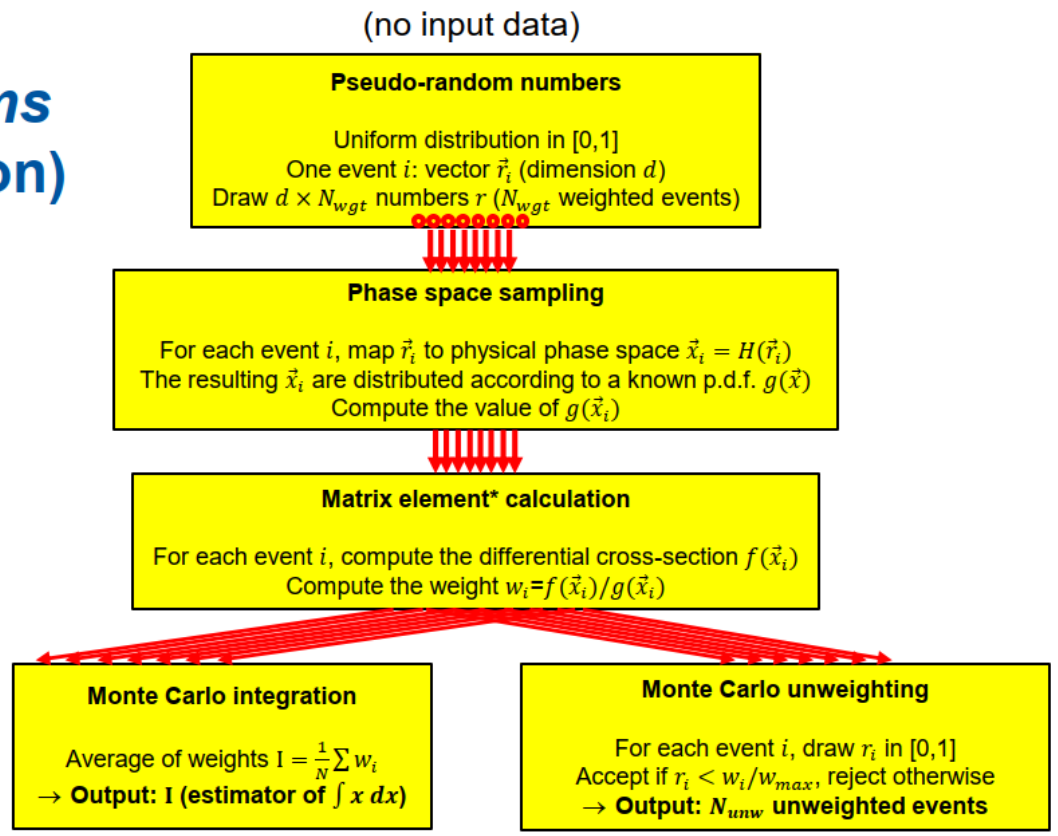
Generators lend themselves naturally to exploiting event-level parallelism via **data-parallel paradigms****

- **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



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

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

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