Developments in software performance and portability for Madgraph5_aMC@NLO

Taylor Childers
Walter Hopkins
Nathan Nichols

Laurence Field
Stephan Hageboeck
Stefan Roiser
David Smith
Andrea Valassi

Olivier Mattelaer

ICHEP, Bologna, 8th July 2022

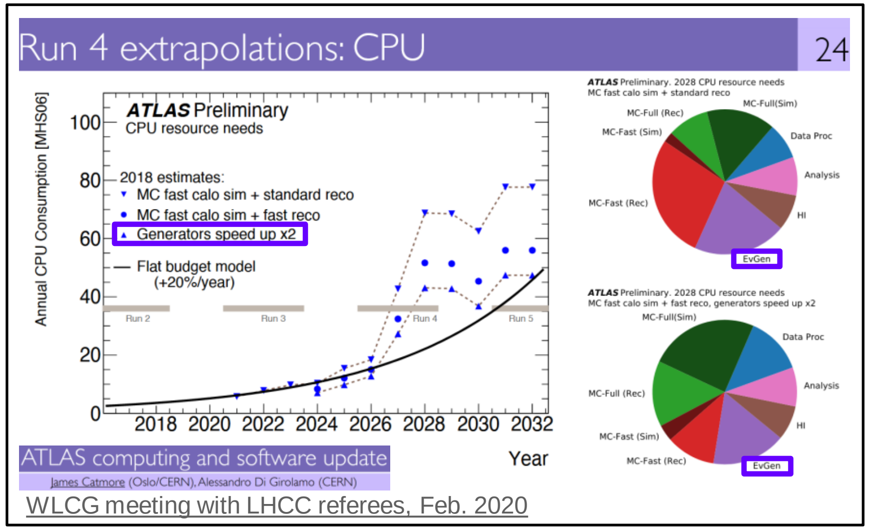
https://agenda.infn.it/event/28874/contributions/169193

Outline

- Introduction
- Results and outlook in three main areas of development (cudacpp, portability frameworks, Madevent)
- Conclusions

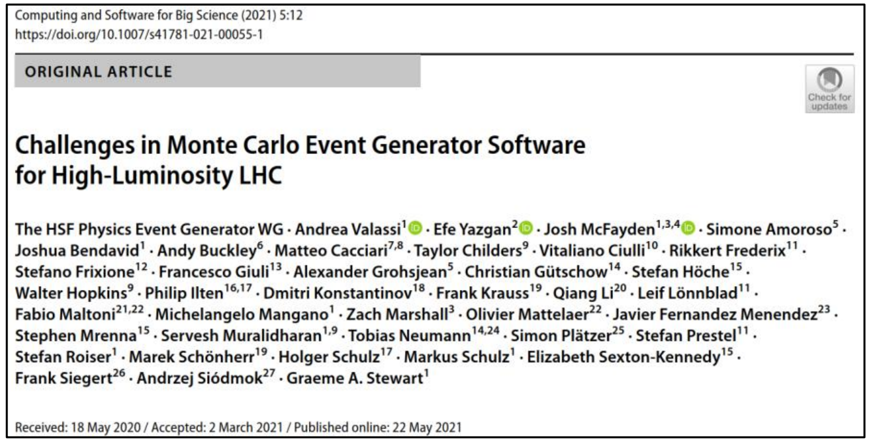
Motivation: Monte Carlo Event Generators in WLCG computing

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



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

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

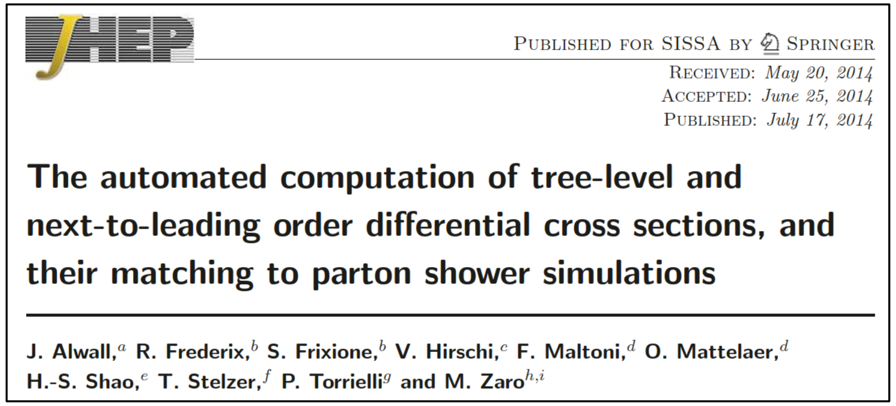
The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5]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

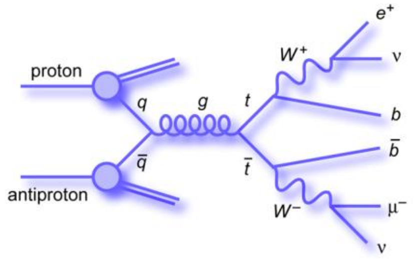
- One of the workhorses for event generation in ATLAS and CMS!

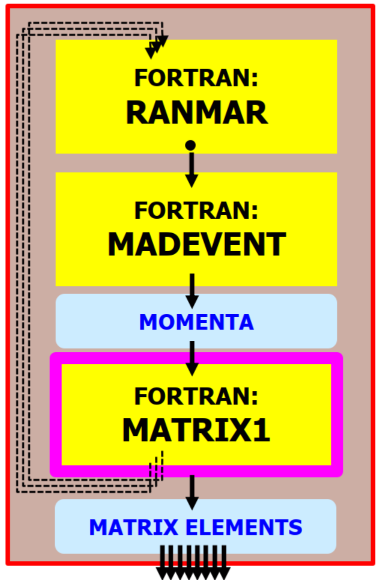

PUBLISHED FOR SISSA BY SPRINGER
RECEIVED: *May 20, 2014*
ACCEPTED: *June 25, 2014*
PUBLISHED: *July 17, 2014*

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

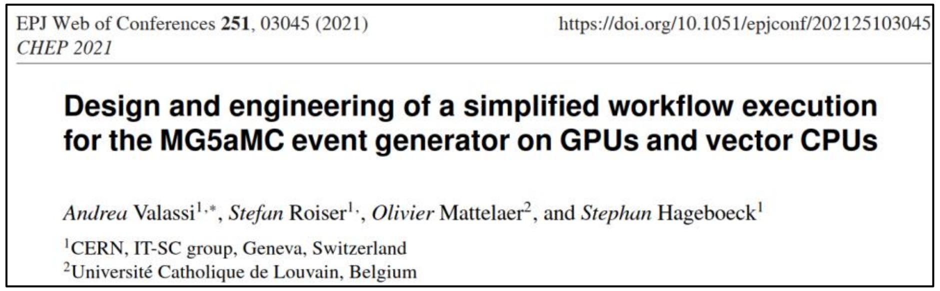




- MG5aMC production version is in Fortran
  - Software outer shell: Madevent (random sampling, integration and event generation)
  - Software inner core: Matrix Element (ME) calculation code, automatically generated for each physics process
    - *Matrix Element calculations take 95%+ of the CPU time for complex processes* (e.g. $gg\to t\bar{t}ggg$)

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
  - Previous results for ‘cudacpp’ (C++ vectorization on CPUs, CUDA on Nvidia GPUs) were shown at vCHEP2021


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

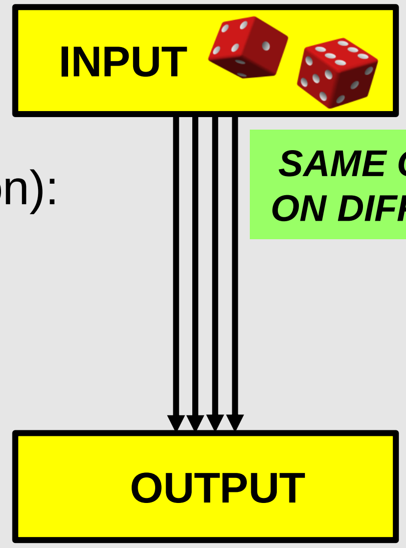
https://doi.org/10.1051/epjconf/202125103045

- New progress since May 2021 has been mainly in three areas of development:
  - (1) Not only hardcoded $e^+e^-\to\mu^+\mu^-$: full code generation in cudacpp, performance for more complex processes
  - (2) Not only cudacpp: add implementations in Alpaka, Kokkos, Sycl also for AMD/Intel GPUs
    - *Goal: gain experience for the HEP software community on the usefulness of portability frameworks (PFs)*
  - (3) Not only a standalone application: integrate CUDA/C++ MEs into Madevent (cross sections, event generation)
    - *Goal: first release of MG5aMC for LO event generation in ATLAS/CMS (CPU SIMD speedups and GPU port)*

MC event generators are 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*:

MC SAMPLING



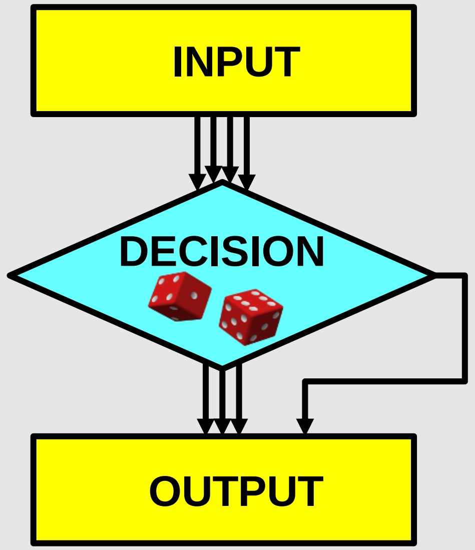
SAME CALCULATION ON DIFFERENT DATA!

<u>Event generators</u>*
(*before* ME calculation):
- MC integration (cross sections)
- MC generation (event samples)

Lockstep processing
Good for SIMT/SIMD

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

MC DECISIONS



<u>Detector simulation (Geant4)</u>
- 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

Matrix Element (ME) calculation in cudacpp: results

(all numbers for one single CPU thread)

Implementation ($gg \to t\bar{t}gg$)	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	2.39E3 **(=1.00)**	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 than on Intel Silver 4216 at CERN
(Gold 6148 has two FMA units, Silver 4216 has one FMA unit)

Implementation ($gg \to t\bar{t}gg$)	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	1.84E3 **(=1.00)**	1.80E3 (x0.98)
Standalone CUDA NVidia V100S-PCIE-32GB (TFlops*: 7.1 FP64, 14.1 FP32)	4.89E5 **(x270)**	9.27E5 **(x500)**

NVidia V100 GPU + Intel Silver 4216 CPU (CERN)

(1) First area of development: MEs in "cudacpp"
Single code base (#ifdef's) for C++ on CPUs and CUDA on Nvidia GPUs
SIMD vectorization on CPUs through Compiler Vector Extensions in C++

Main new results since vCHEP2021:

- Backport to code generation (test more complex processes)
  - speedups seen for ee_mumu now also ~confirmed for gg_ttgg
  - but GPU speedups decrease a bit (higher "register pressure")
- Achieve full theoretically possible SIMD speedup on CPUs
  - x8 double, x16 float from AVX512 on high-end Intel CPUs
- New features added for MadEvent integration

  (this slide shows numbers from the standalone test application; see the final slides for performance numbers within madevent)

Portability Frameworks (PFs)

(2) Second area of development: MEs on PFs

- PFs allow writing algorithms once and running on many architectures with some hardware-specific optimizations

- CUDA only runs on NVidia GPUs, while Kokkos, Alpaka and Sycl[Intel] also run on AMD and Intel GPUs

ME calculation in PFs: GPU results (Nvidia A100)

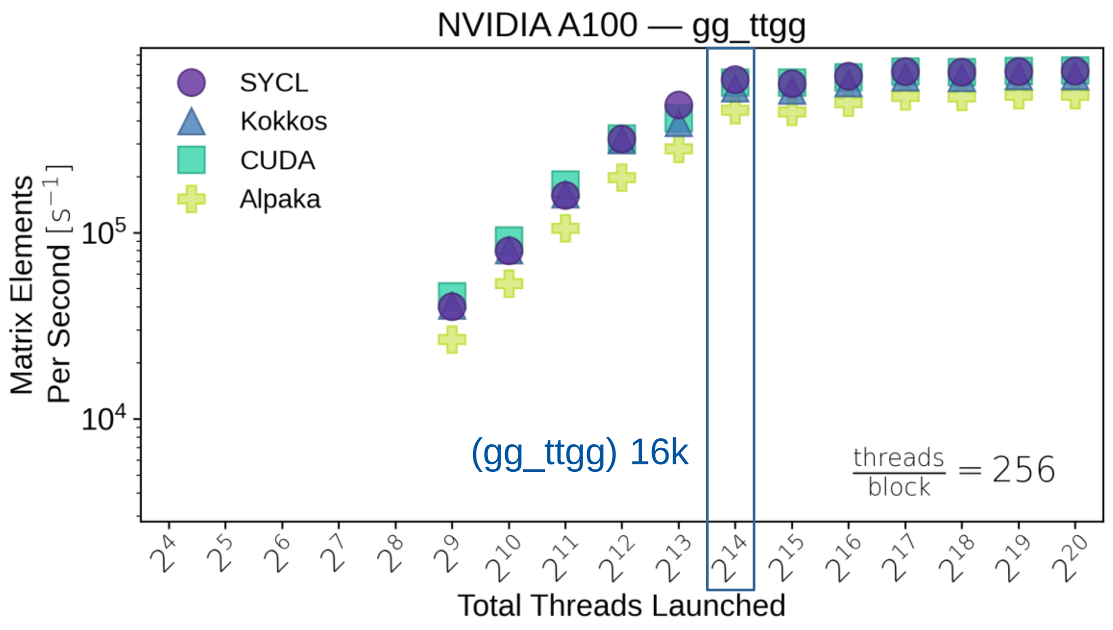
Throughput scaling (threads, blocks) for a complex $gg \to t\bar{t}gg$ process

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

- Good news 1: all four implementations look similar for Nvidia in gg_ttgg!
  - The benefit of direct CUDA over a PF is limited, if any at all

NB: focus on gg_ttgg which is computationally intensive!

In simpler processes like ee_mumu, performance is more affected by data copies, memory access or kernel launching overheads (and the observed SYCL implementation is faster than the CUDA one - to be understood)



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+

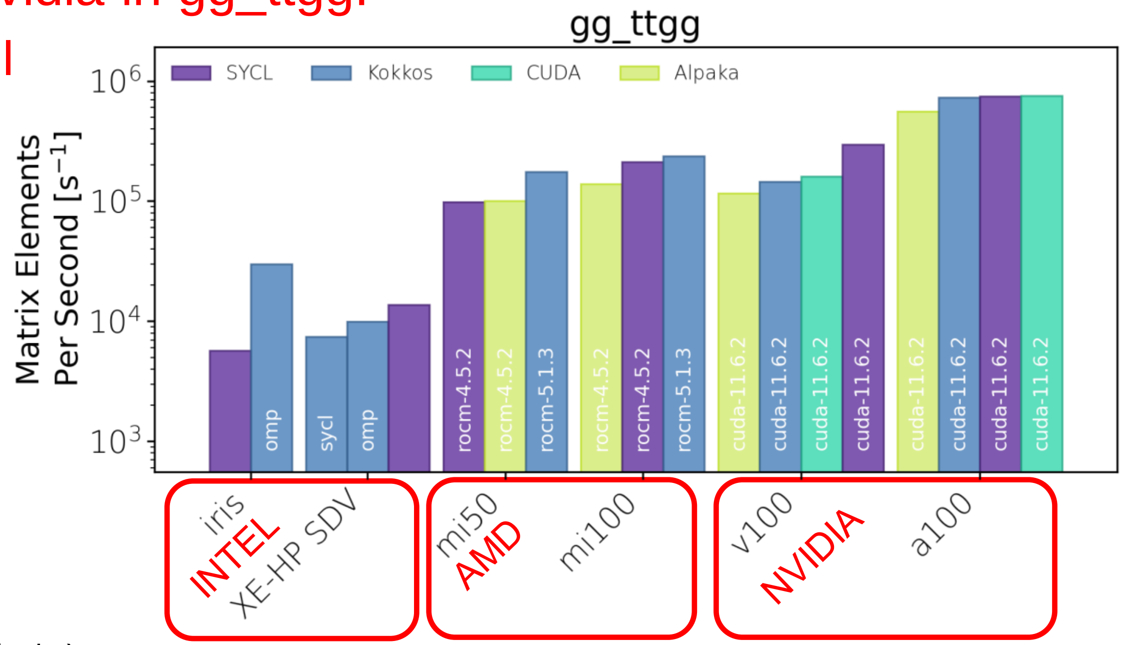
– Simpler processes need even more, e.g. 500k for ee_mumu

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

Maximum throughput (plateau of scaling plot on the previous slide) for 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)

- Good news 1: all four implementations look similar on Nvidia in gg_ttgg!
  - 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)
  - PFs also run out of the box on CPUs (performance under investigation)

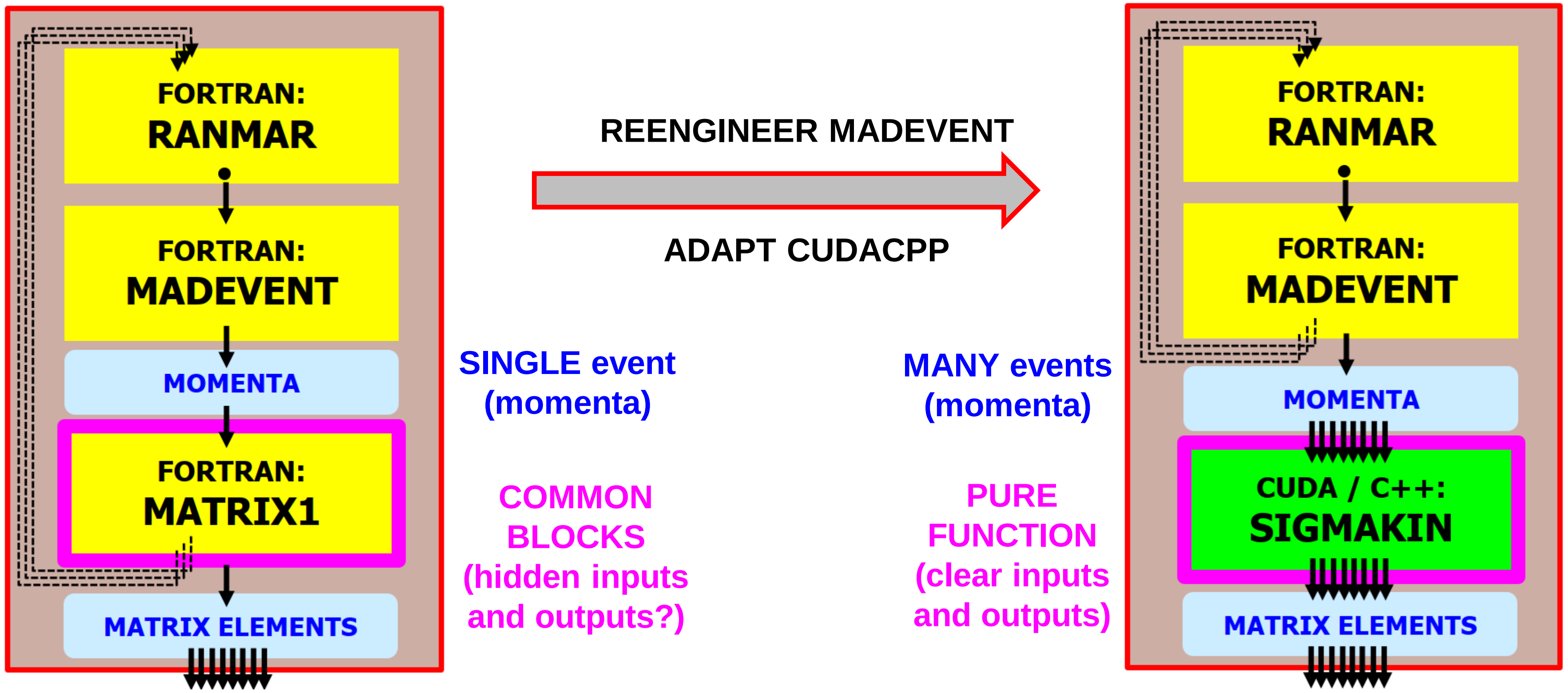
Xe-HP is a software development vehicle for functional testing only - currently used at Argonne and other customer sites to prepare their code for future Intel data centre GPUs

Matrix element integration in MadEvent: overview

(3) Third area of development: replace 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)

Intel Gold 6148 CPU (Juwels Cluster HPC)

Implementation ($gg \to t\bar{t}gg$)	Evts/second full workflow	MEs/second MEs only
1-core MadEvent Fortran scalar	1.96E3 **(=1.00)**	2.12E3 **(=1.00)**
1-core Standalone C++ scalar	1.72E3 (x0.9)	1.85E3 (x0.9)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	3.56E3 (x1.8)	4.08E3 (x1.9)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	6.72E3 (x3.4)	8.80E3 (x4.2)
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	7.08E3 (x3.6)	9.41E3 (x4.4)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	9.92E3 **(x5.1)**	1.52E4 **(x7.2)**

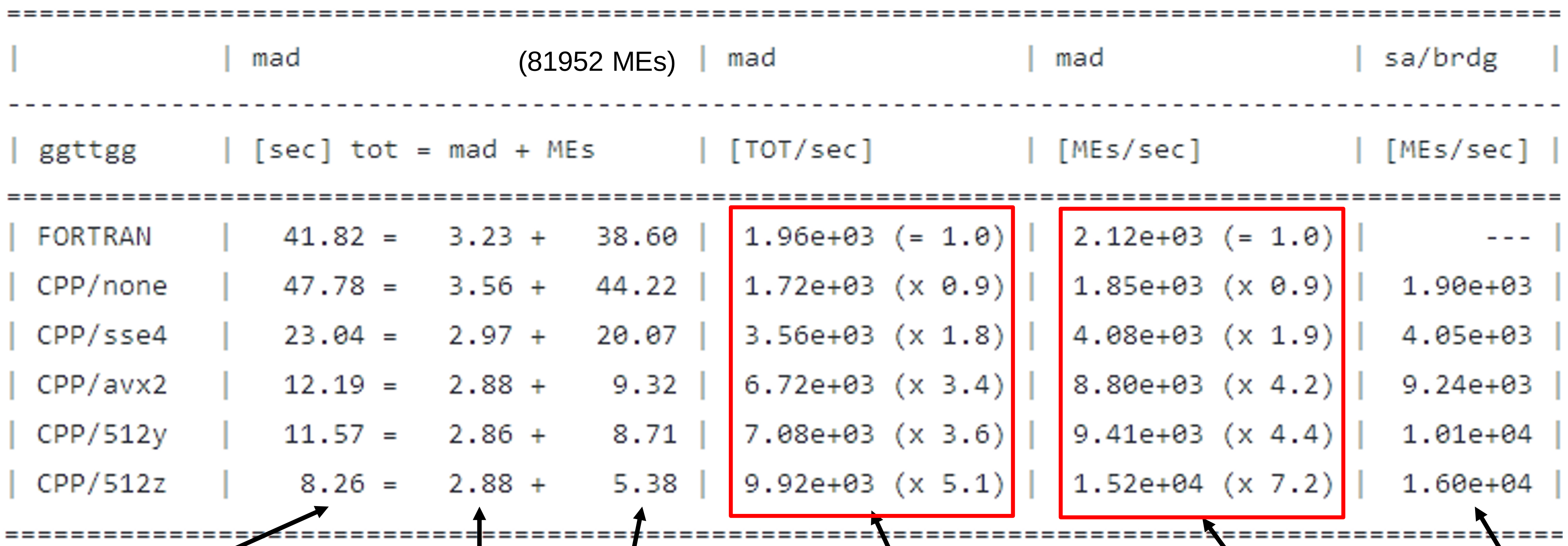
NVidia V100 GPU + Intel Silver 4216 CPU (CERN)

Implementation ($gg \to t\bar{t}ggg$)	Evts/second full workflow	MEs/second MEs only
1-core MadEvent Fortran scalar	7.01E1 **(=1.00)**	7.37E1 **(=1.00)**
Standalone CUDA NVidia V100S-PCIE-32GB	1.17E3 **(x16.7)**	7.39E3 **(x100)**

Summary of performance within madevent so far:
- on CPU: **~x8** for MEs alone, **~x5** for madevent+MEs
- on GPU: **~x100-300** for MEs alone, **~x20** for madevent+MEs

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)

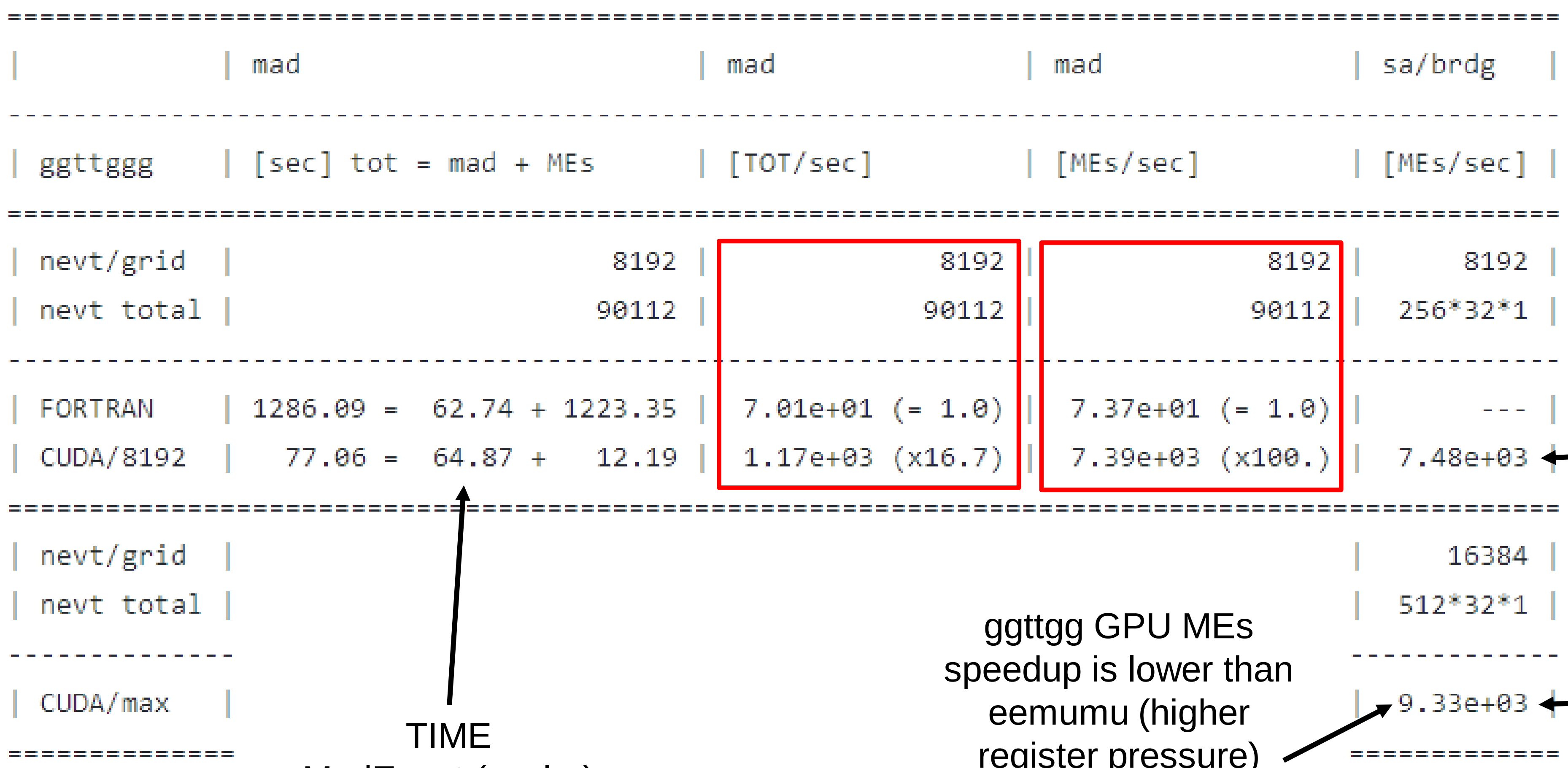
THROUGHPUT MadEvent + MEs (within madevent)

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

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

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

Outlook and main plans

- [madevent/functionality, Q3 2022] ***Alpha release for the experiments***
  - Add event-by-event random choice of color and helicity (same LHE files), check pdf, user parameters…

- [madevent/performance, end 2022?] *Speed up GPU workflow (reduce madevent scalar overhead)*
  - One possible option: heterogenous workflow (madevent on many CPU cores, MEs on GPU)?

- [madevent/performance, end 2022?] *Speed up GPU MEs in madevent (allow larger GPU grids)*
  - This requires reducing the number of Fortran arrays in madevent (reduce CPU memory usage)

- [MEs/functionality, 2023] *Add support for NLO (loops and matching to parton showers)*
- [MEs/performance, 2023] *Add "helicity recycling" (x2-3 extra speedup in CPU/GPU, now only in Fortran)*
- [MEs/performance, 2023] *Numerical precision studies: float (x2 extra speedup in CPU/GPU), fast math*
- [MEs/performance, end 2022?] *Try smaller GPU kernels, try Nvidia tensor cores for QCD color algebra*

- [portability/performance/GPU, 2023?] *HIP in cudacpp, detailed comparisons with PFs with newer code*
- [portability/performance/CPU, 2023?] *std::thread in cudacpp, detailed SIMD/MT comparisons with PFs*

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...)
- For the ME calculation alone, speedups ~x8 for CPUs and ~x300 for V100 GPUs have been observed
  - Our performance on CPUs achieves the theoretical limit for SIMD vectorization: ~perfect lockstep processing
- For the full MadEvent+MEs, performance is limited by scalar processing in MadEvent (Amdahl's law)
  - On CPUs, an overall speedup ~x5 compared to Fortran has been achieved so far
  - On GPUs, the overall speedup is limited to ~x20 so far for processes where MEs "only" take up 95% of the time
  - Speeding up the scalar processing in MadEvent is one of our next challenges (with large potential gains!)
- There is potential for many additional speedups (single precision, helicity recycling, smaller kernels...)
- Portability Frameworks work well for us - easier development with a single code for many GPU flavors
  - Similar performance to direct CUDA on Nvidia GPUs - and also run out of the box on AMD and Intel GPUs

Acknowledgements

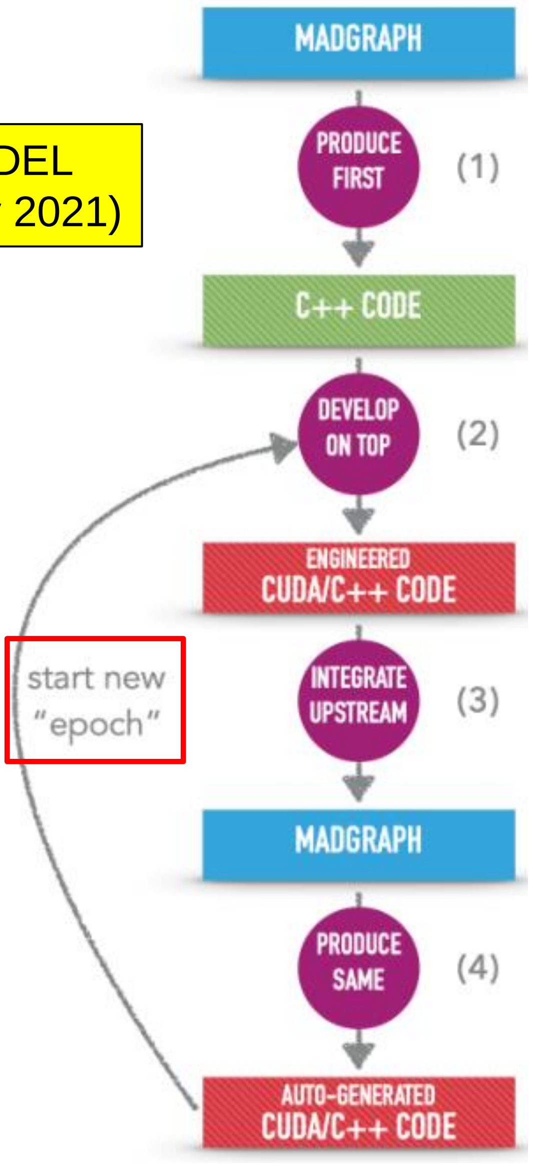
- We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract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

BACKUP SLIDES

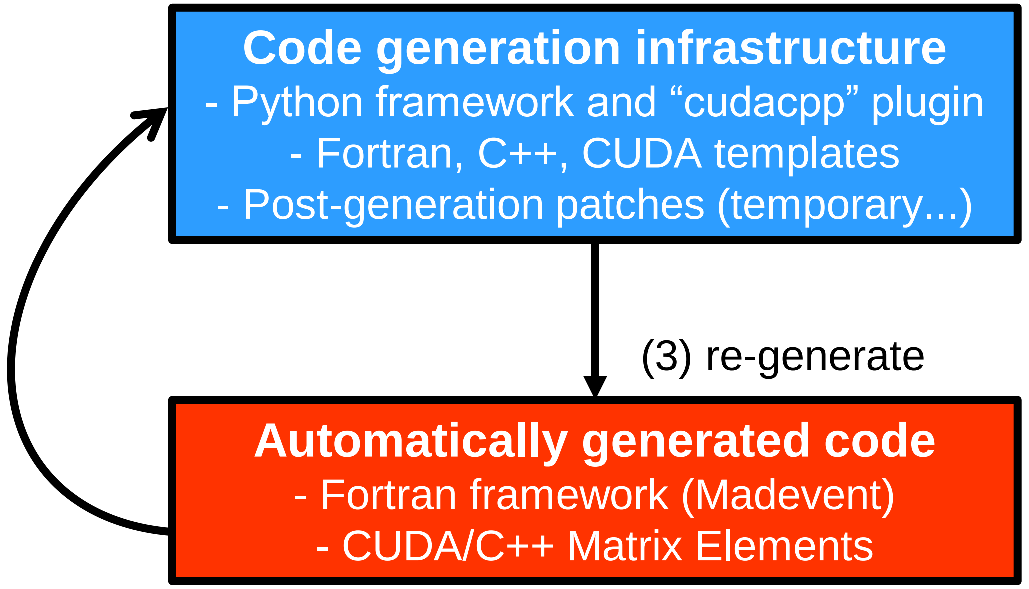
Code generation: from many "epochs" to a single evolving "epoch"

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...)

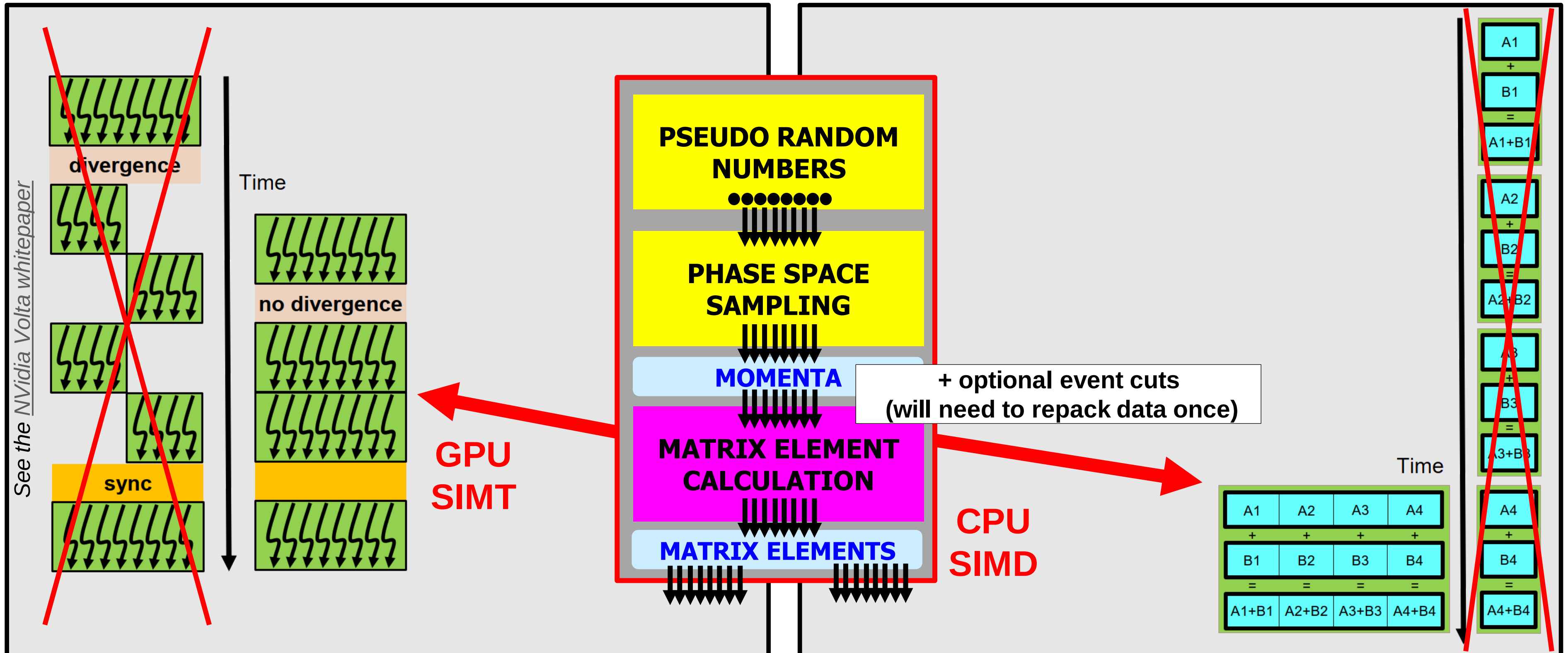
(3) re-generate

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)



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

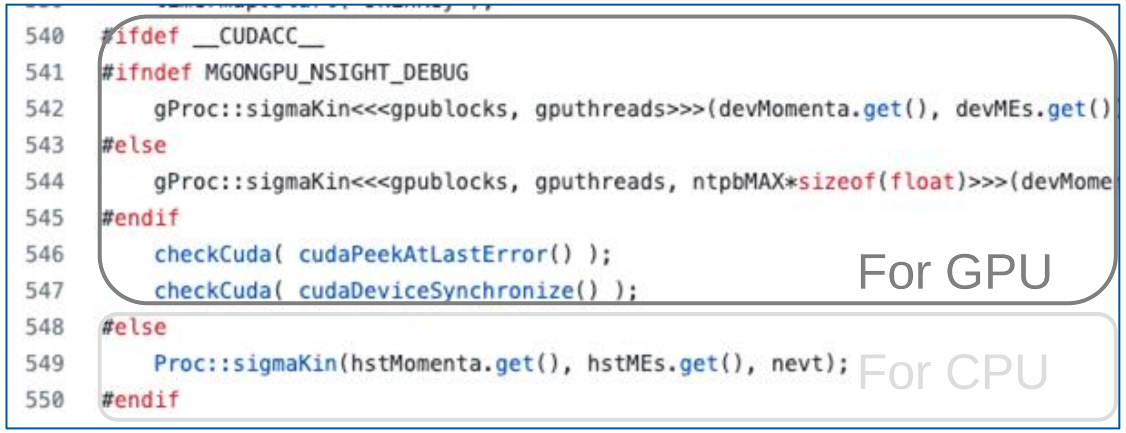
CPU SIMD (Single Instruction Multiple Data)
Lockstep: same op for all data in a vector register
Minimum parallelism: 2 to 16 (SSE/AVX2/AVX512...)

Portability Frameworks (PFs)

(2) Second line of development: MEs on PFs

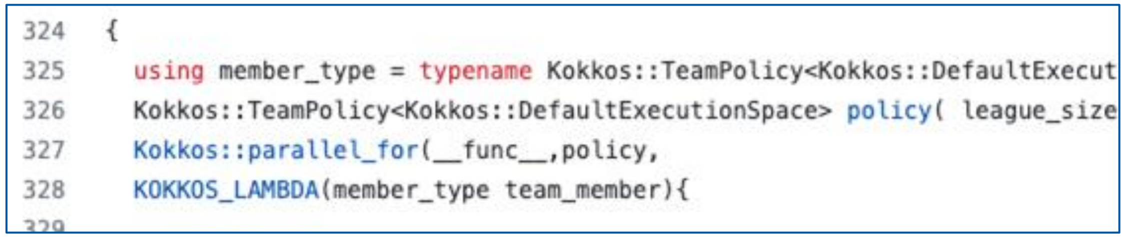
- PFs allow writing algorithms once and running on many architectures with some hardware-specific optimizations
- CUDA code can only run on NVidia GPUs, while Kokkos, Alpaka, and Sycl[Intel] codes can run on most hardware
- In "cudacpp", #ifdef directives separate code branches for GPU and CPU code during 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



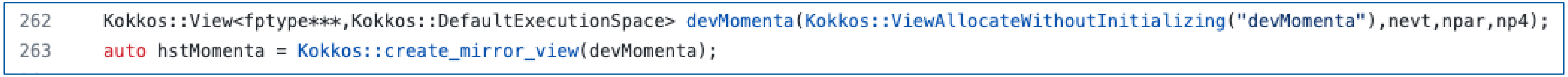
```
#ifdef __CUDACC__
#ifndef MGONGPU_NSIGHT_DEBUG
    gProc::sigmaKin<<<gpublocks, gputhreads>>>(devMomenta.get(), devMEs.get()
#else
    gProc::sigmaKin<<<gpublocks, gputhreads, ntpbMAX*sizeof(float)>>>(devMome
#endif
    checkCuda( cudaPeekAtLastError() );
    checkCuda( cudaDeviceSynchronize() );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

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)

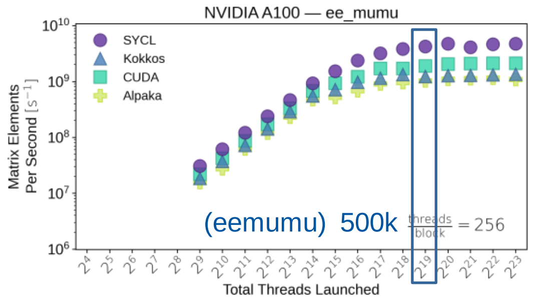
Throughput scaling (threads, blocks) for a complex $gg \to t\bar{t}gg$ process

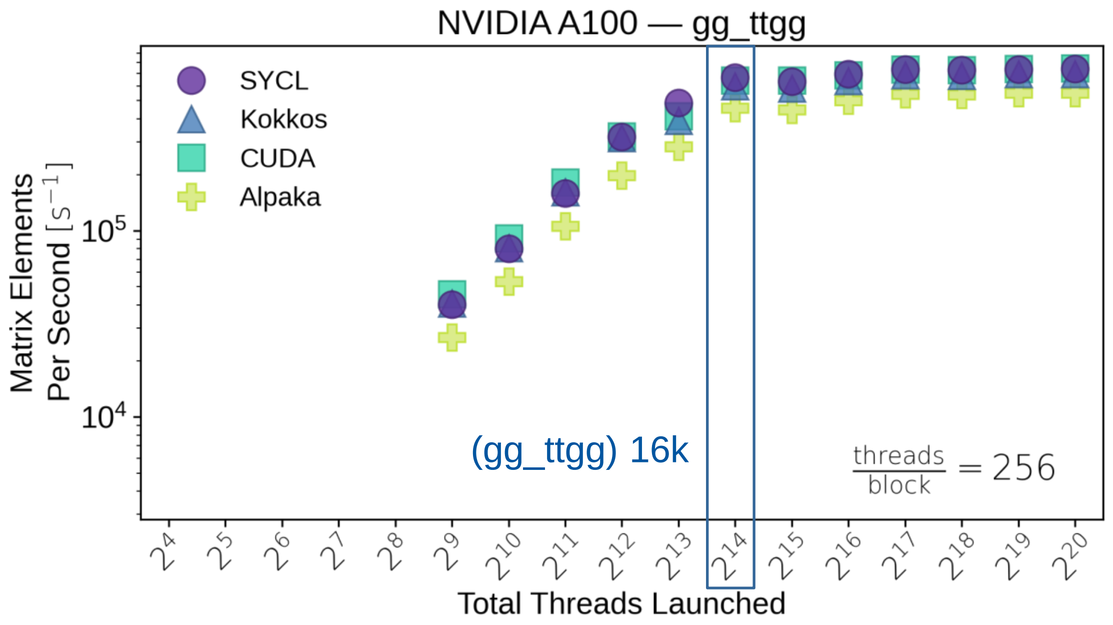
(note: this is an older version of the code with respect to the results shown earlier for cudacpp alone)

- Good news 1: all four implementations look similar for Nvidia in gg_ttgg!
  - The benefit of direct CUDA over a PF is limited, if any at all

NB: focus on gg_ttgg which is computationally intensive!

In simpler processes like ee_mumu, performance is more affected by data copies, memory access or kernel launching overheads (and the observed SYCL implementation is faster than the CUDA one - to be understood)





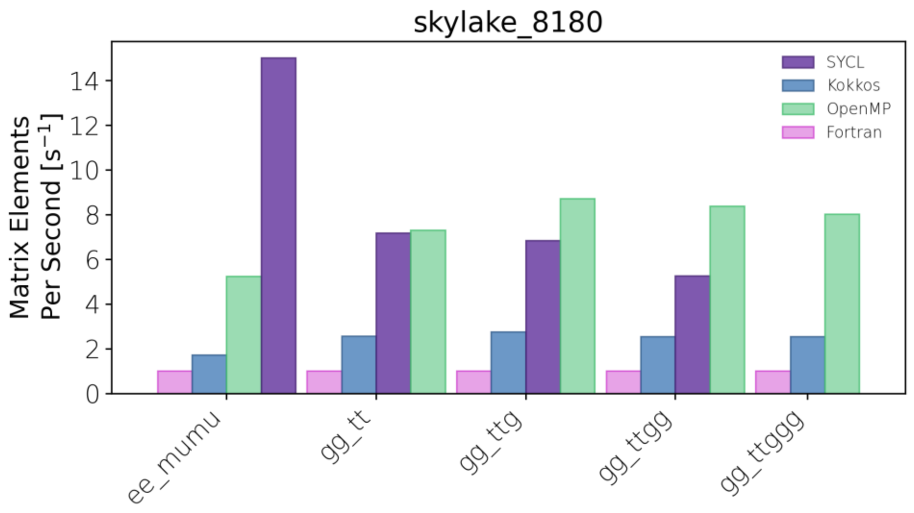
En passant, keep in mind this for later: you need at least 16k "events per GPU grid" to fill up a V100 or A100 with gg_ttgg+

– Simpler processes need even more, e.g. 500k for ee_mumu

ME calculation in PFs: 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)



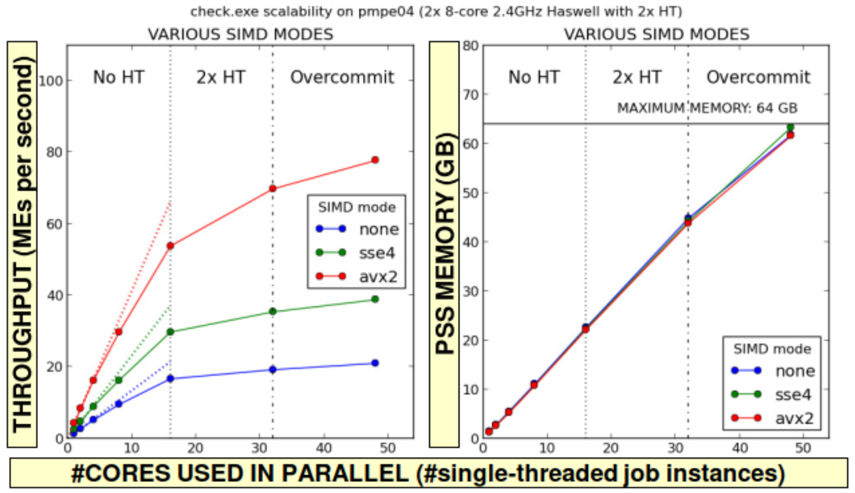
- CPUs have two very different parallelisms we can exploit:
  - Many floats/doubles per vector register: vectorization (SIMD)
  - Many physical/virtual cores: multi-threading (or many processes)
- *NB: this plot is not comparing exactly apples to apples!*
  - Fortran: many copies of one process (MPI), no vectorization
  - Kokkos: internal multithreading? limited auto-vectorization?
  - SYCL: internal multithreading? limited auto-vectorization?
  - cudacpp: OpenMP multithreading, *explicit vectorization (CVE)*
    - NB: the OMP multithreading in the cudacpp plugin is known to be suboptimal and will be reengineered (probably with std::thread instead)

PFs code runs out of the box also on CPUs

The cudacpp implementation handles both vectorization and threading at a much lower level

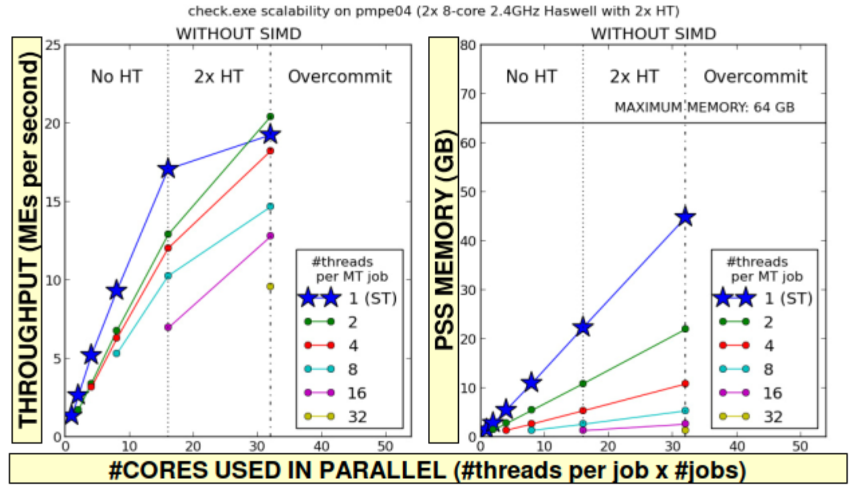
CPU throughput plots – SIMD + multi-core

- *Two <u>different</u> 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)

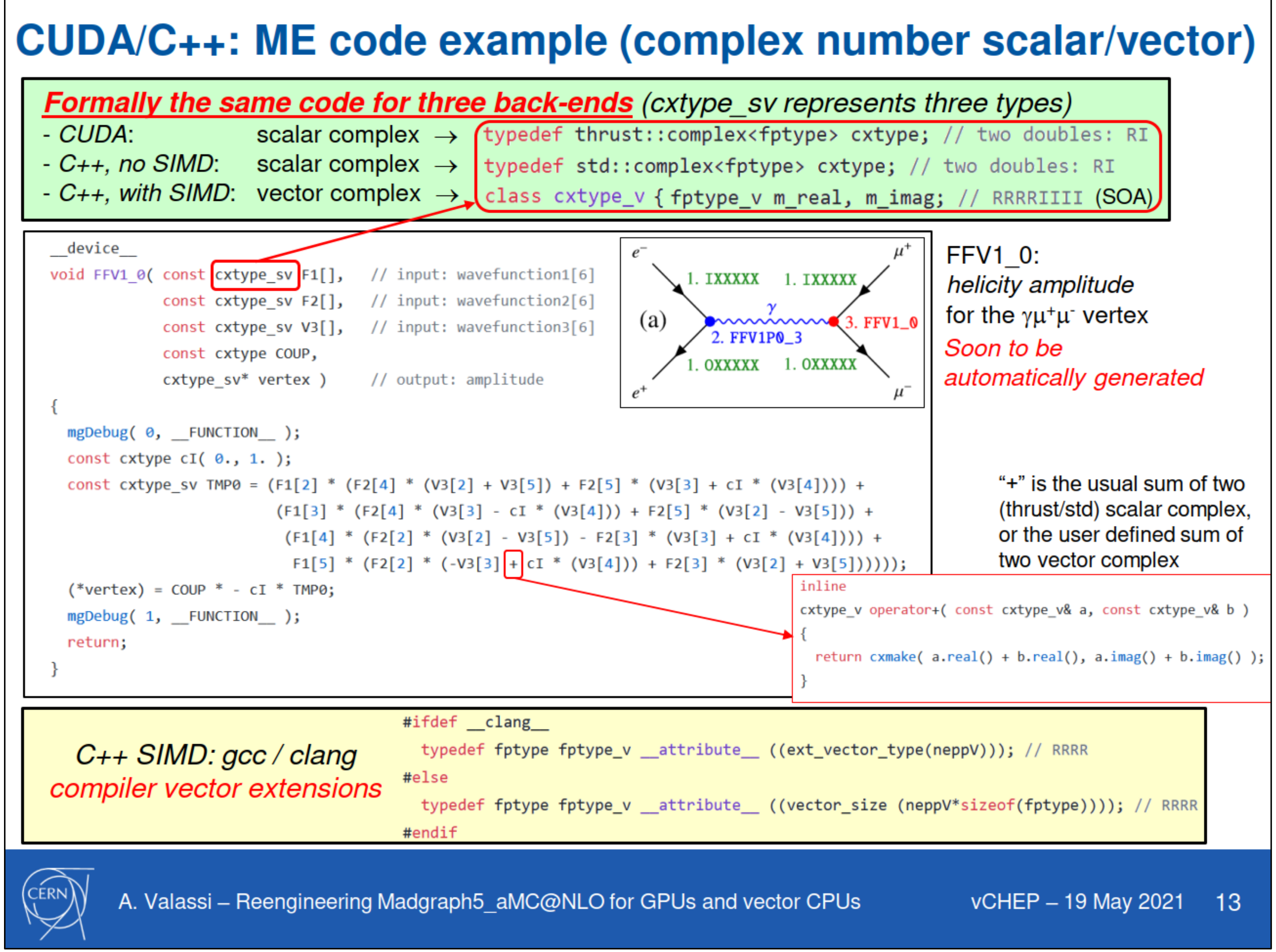


Multiple instances of single-threaded MG5aMC
Combine SIMD and multi-core speedup
Memory proportional to number of cores used

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



A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   21

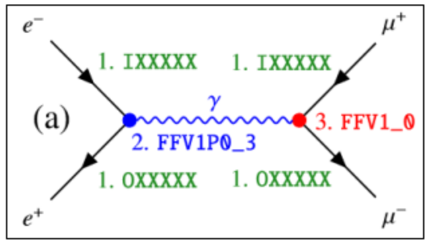


CUDA/C++: ME code example (complex number scalar/vector)

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 $\gamma\mu^+\mu^-$ 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() );
}
```

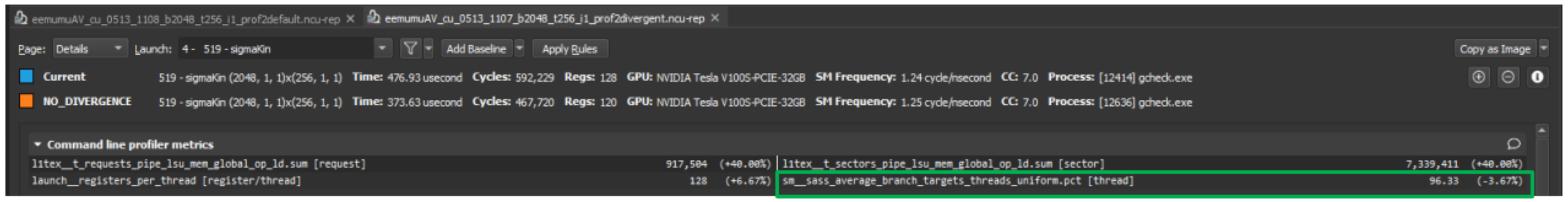
C++ SIMD: gcc / clang
compiler vector extensions

```
#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_j1_prof2default.ncu-rep   eemumuAV_cu_0513_1107_b2048_t256_j1_prof2divergent.ncu-rep

Page: Details   Launch: 4 - 519 - sigmaKin   Add Baseline   Apply Rules   Copy as Image

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

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)

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

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

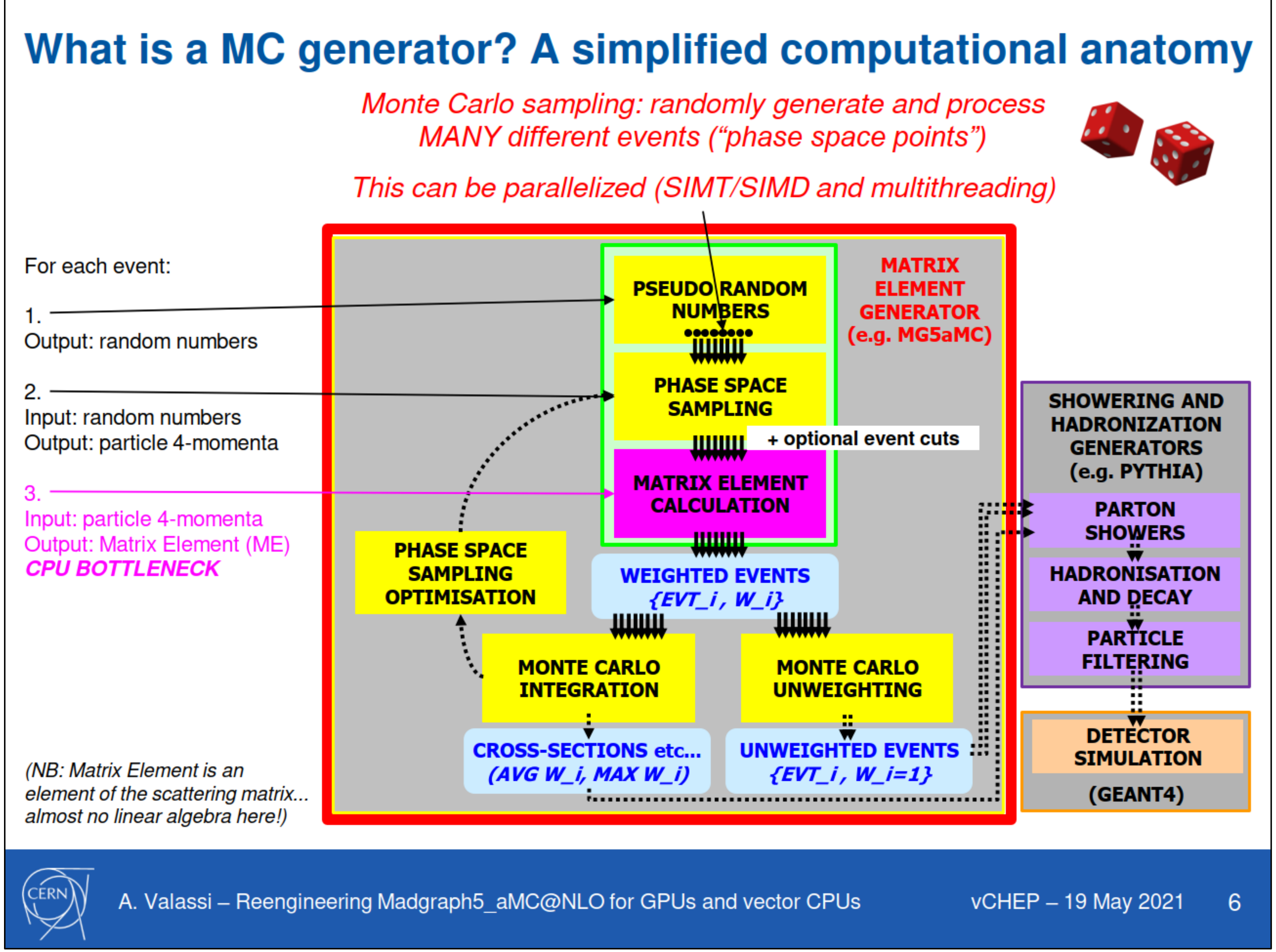
Build environment on JLSE (Cuda and OpenMP)

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



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

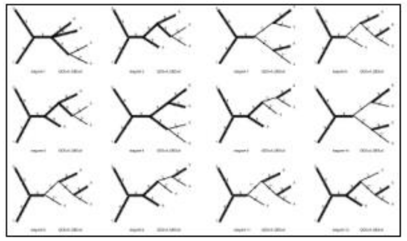
(NB: Matrix Element is an element of the scattering matrix… almost no linear algebra here!)

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}
SHOWERING AND HADRONIZATION GENERATORS (e.g. PYTHIA)
PARTON SHOWERS
HADRONISATION AND DECAY
PARTICLE FILTERING
DETECTOR SIMULATION (GEANT4)

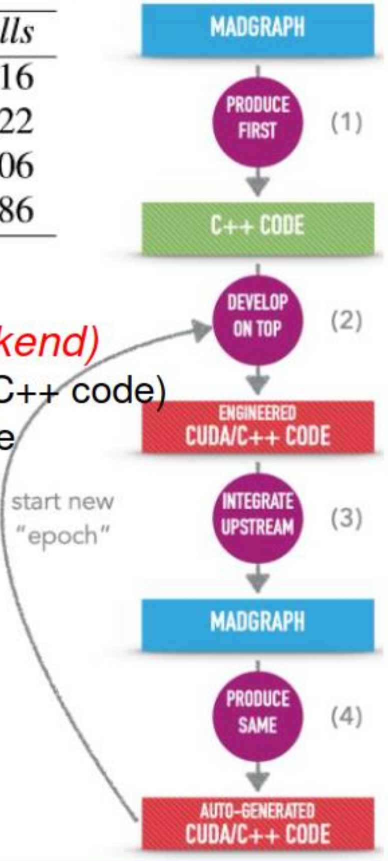
A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   6

Code is auto-generated ⇒ Iterative development process

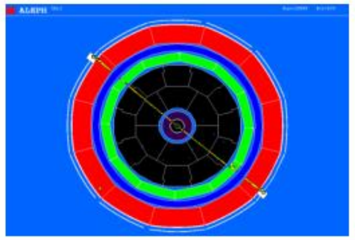
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Currently Fortran (default), C++, or Python
  - The more particles in the collision, the more Feynman diagrams and the more lines of code

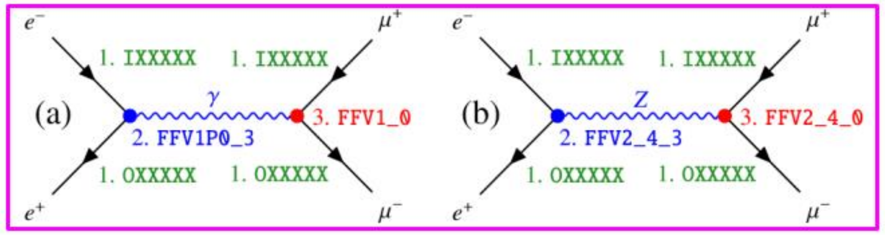


Process	*LOC*	*functions*	*function calls*
$e^+e^- \to \mu^+\mu^-$	776	8	16
$gg \to t\bar{t}$	839	10	22
$gg \to t\bar{t}g$	1082	36	106
$gg \to t\bar{t}gg$	1985	222	786

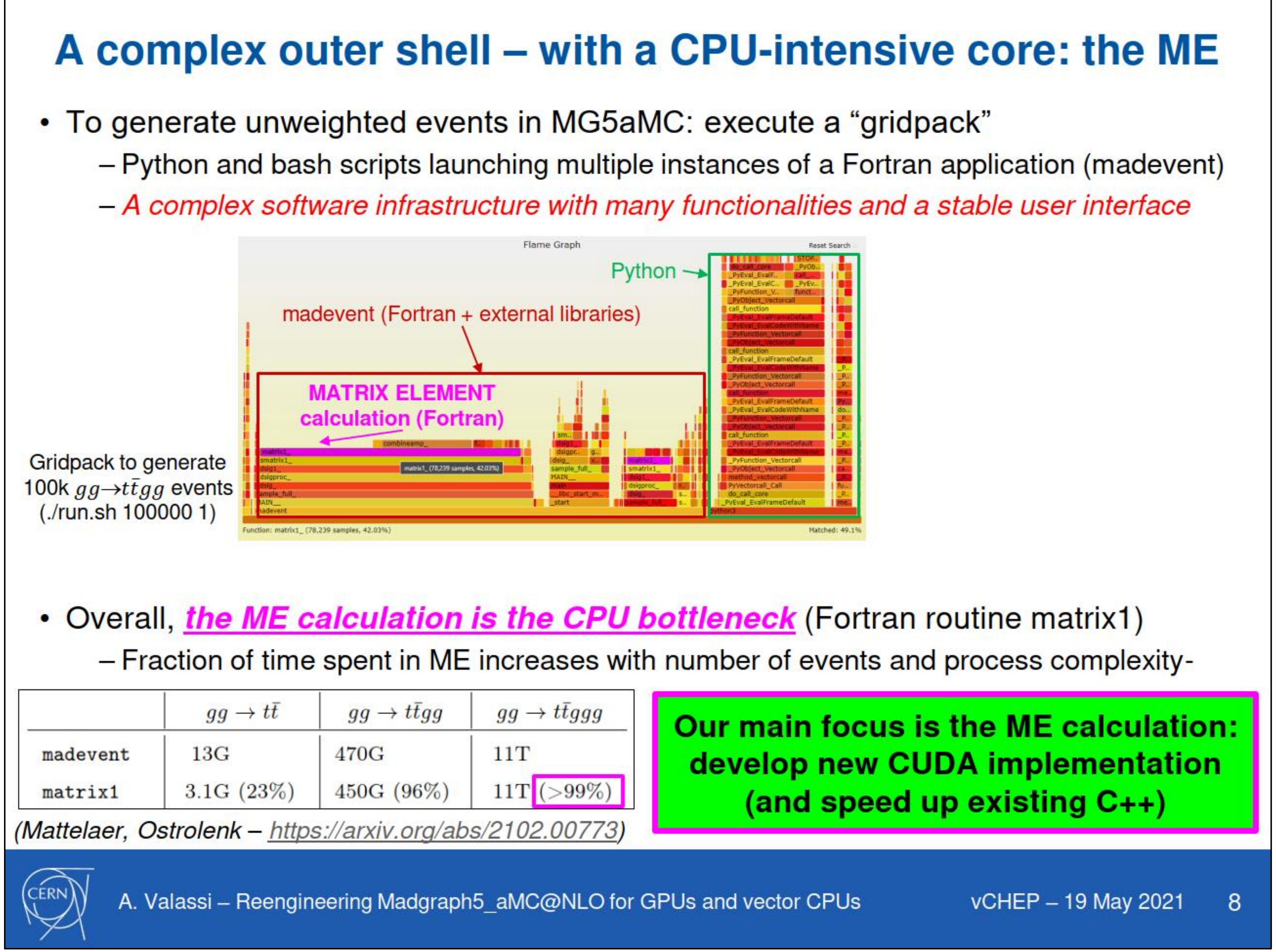


- *Goal: modify code-generating code (add CUDA, improve C++ backend)*
  - (1) Start simple: *bootstrap with $e^+e^- \to \mu^+\mu^-$* (two diagrams, few lines of C++ code)
  - (2,3) Add CUDA and improve C++, port upstream to Python meta-code
  - (4) *Generate more complex LHC processes $gg \to t\bar{t}$,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





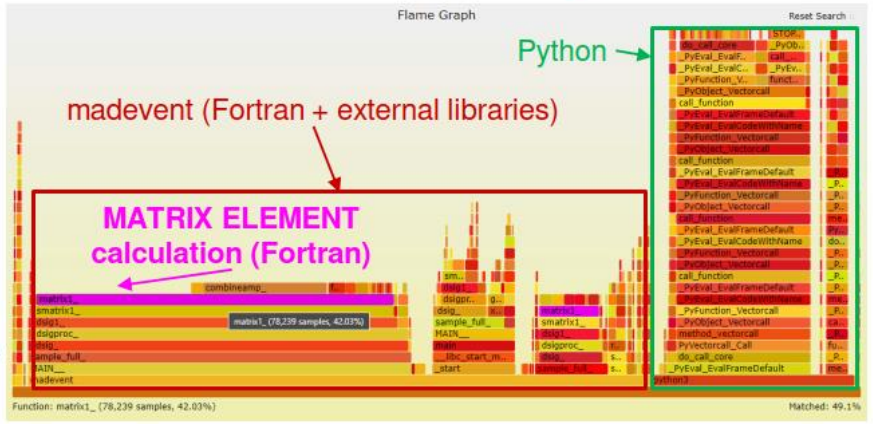
A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   7



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*

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



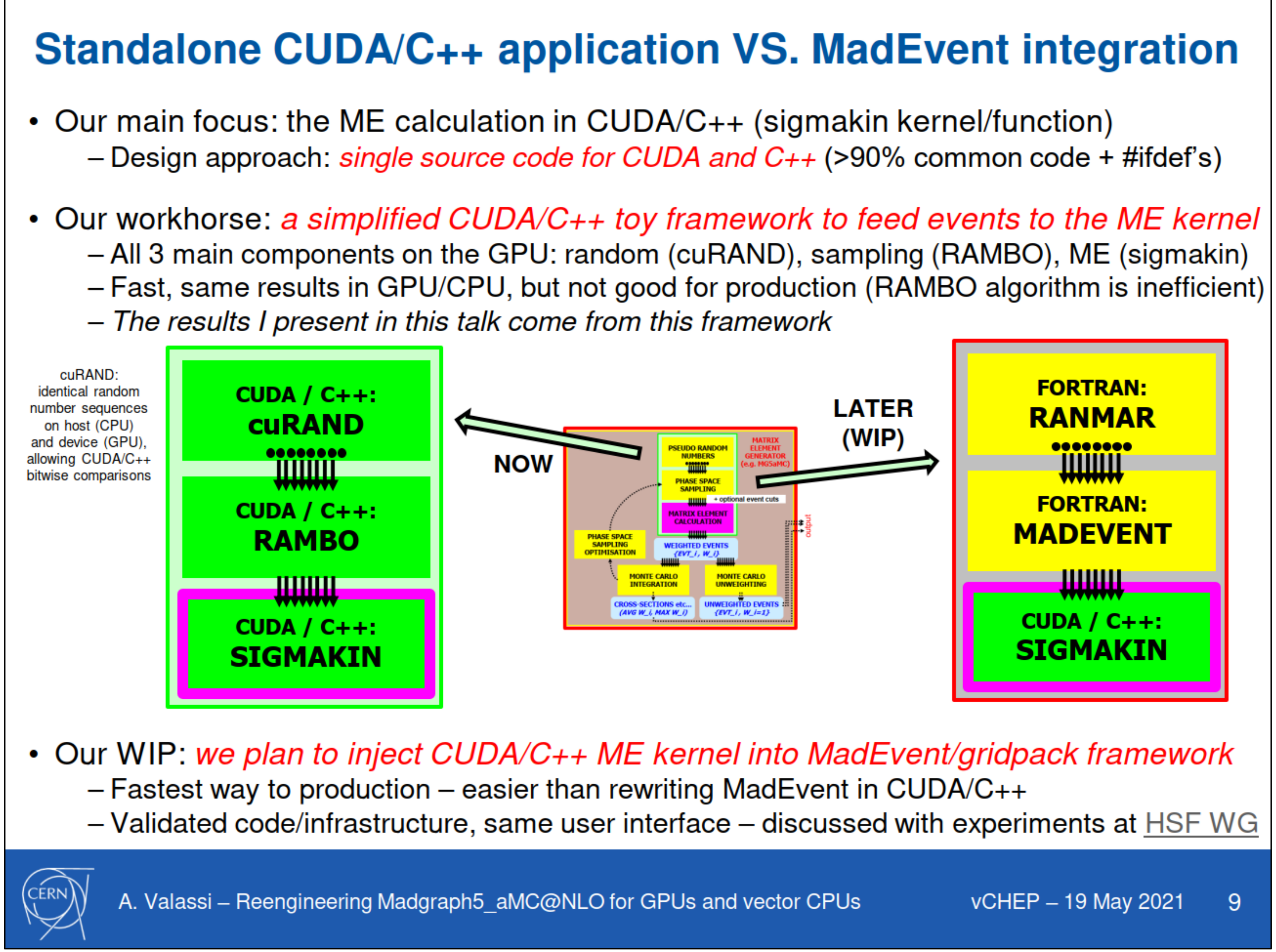
- Overall, ***the ME calculation is the CPU bottleneck*** (Fortran routine matrix1)
  - Fraction of time spent in ME increases with number of events and process complexity-

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

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

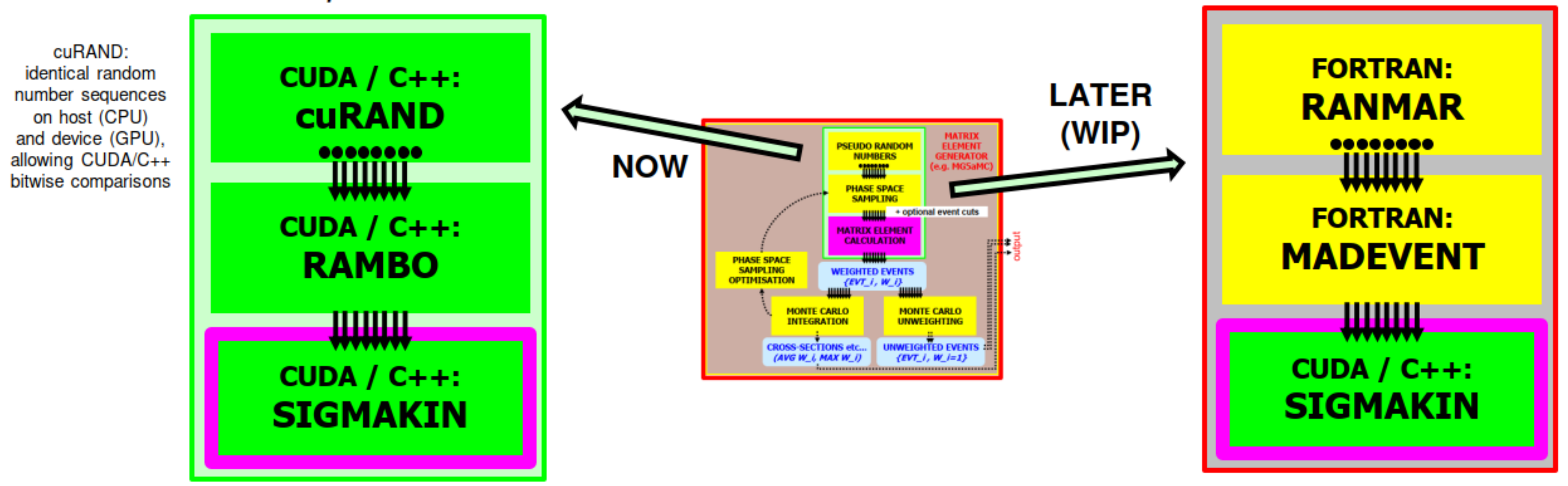
Our main focus is the ME calculation: develop new CUDA implementation (and speed up existing C++)

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   8



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*



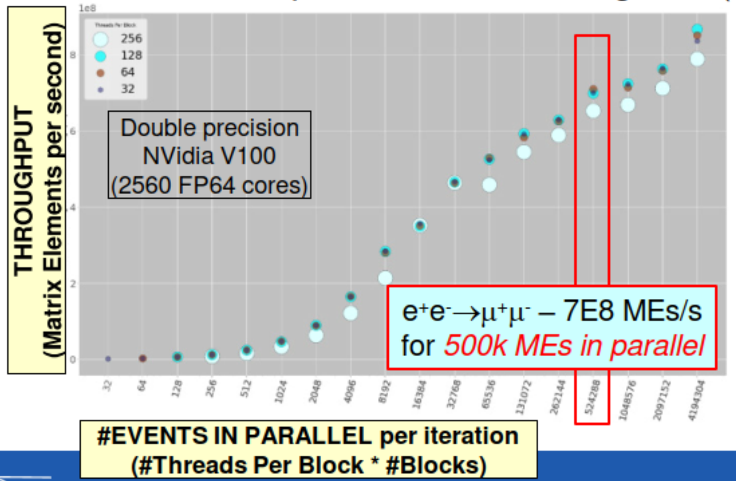
cuRAND: identical random number sequences on host (CPU) and device (GPU), allowing CUDA/C++ bitwise comparisons
CUDA / C++: cuRAND
CUDA / C++: RAMBO
CUDA / C++: SIGMAKIN
NOW
PSEUDO RANDOM NUMBERS
MATRIX ELEMENT GENERATOR (e.g. MG5aMC)
PHASE SPACE SAMPLING
+ optional event cuts
MATRIX ELEMENT CALCULATION
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

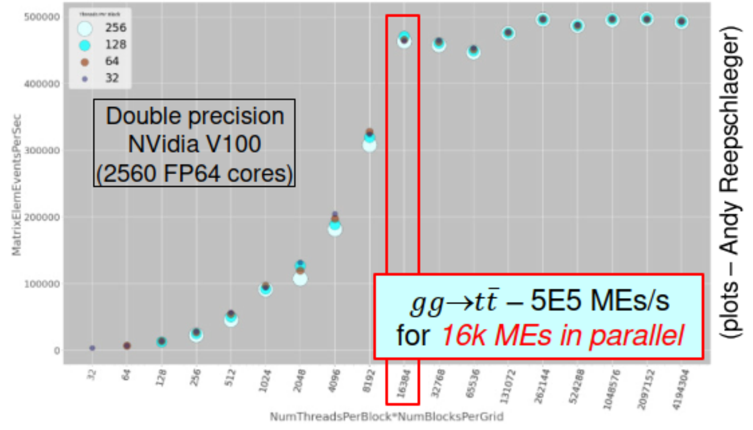
- Our WIP: *we plan to inject CUDA/C++ ME kernel into MadEvent/gridpack framework*
  - Fastest way to production – easier than rewriting MadEvent in CUDA/C++
  - Validated code/infrastructure, same user interface – discussed with experiments at HSF WG

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

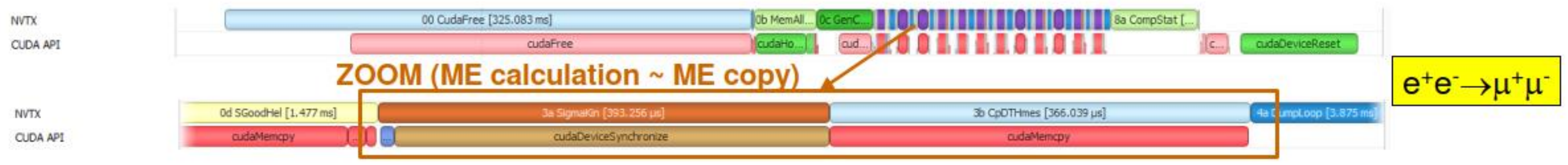




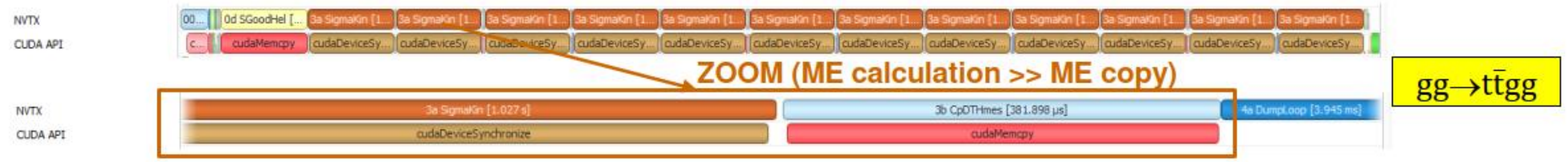
A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   12

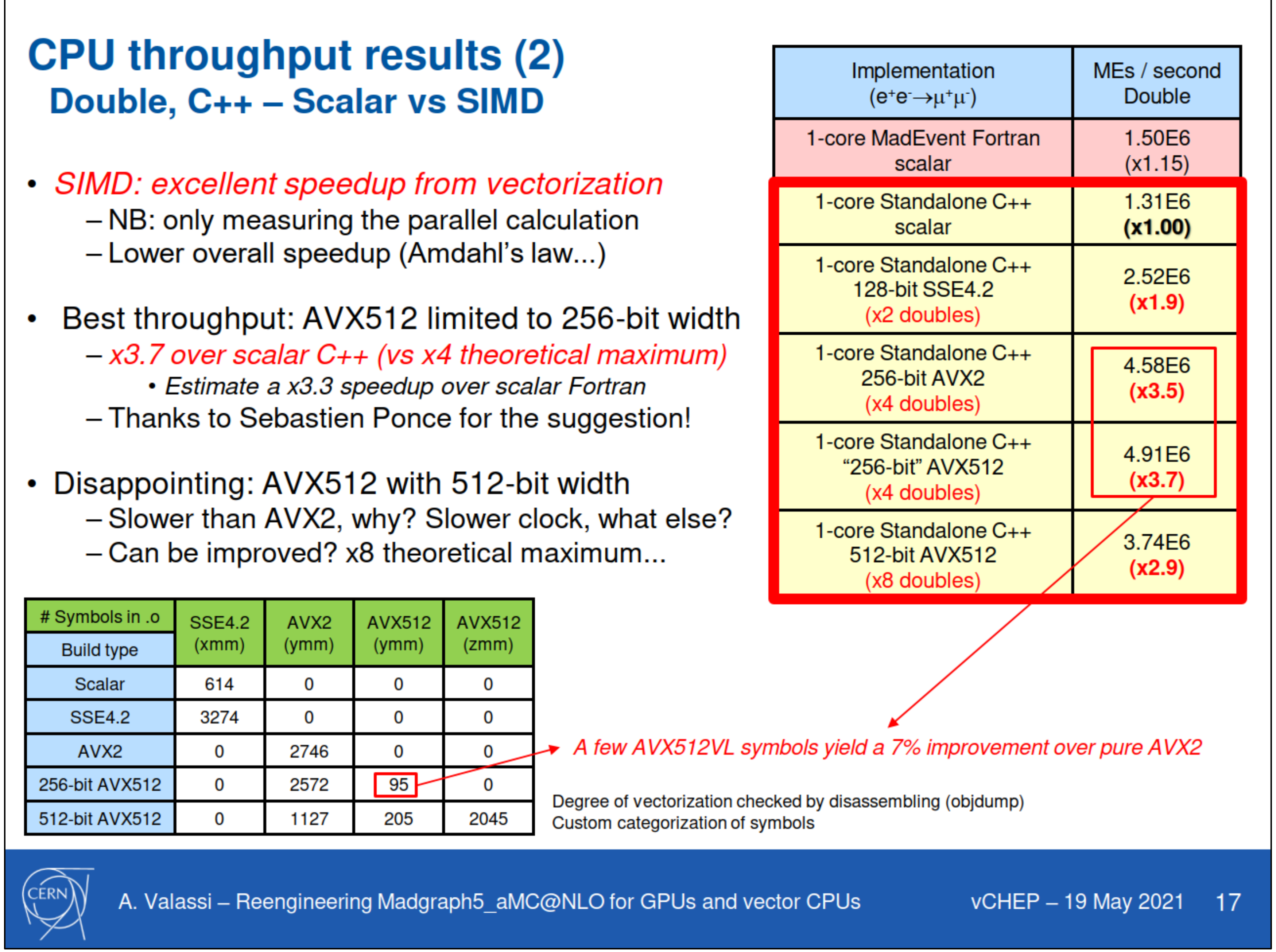
CUDA: Host(CPU)-to/from-Device(GPU) data copy has a cost

- 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)



- 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...

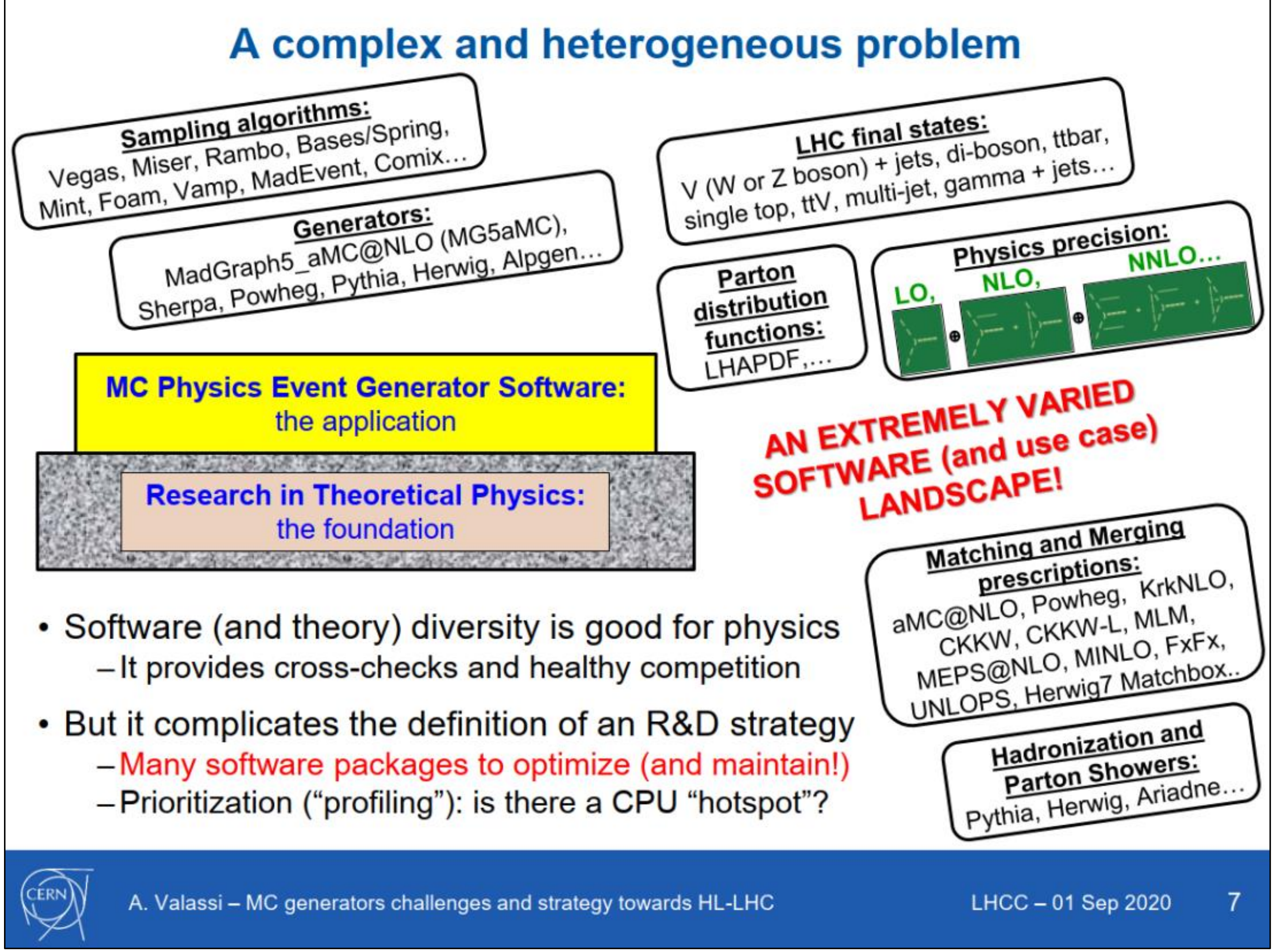
Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double
1-core MadEvent Fortran scalar	1.50E6 (x1.15)
1-core Standalone C++ scalar	1.31E6 **(x1.00)**
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles)	2.52E6 **(x1.9)**
1-core Standalone C++ 256-bit AVX2 (x4 doubles)	4.58E6 **(x3.5)**
1-core Standalone 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	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
Build type				
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!

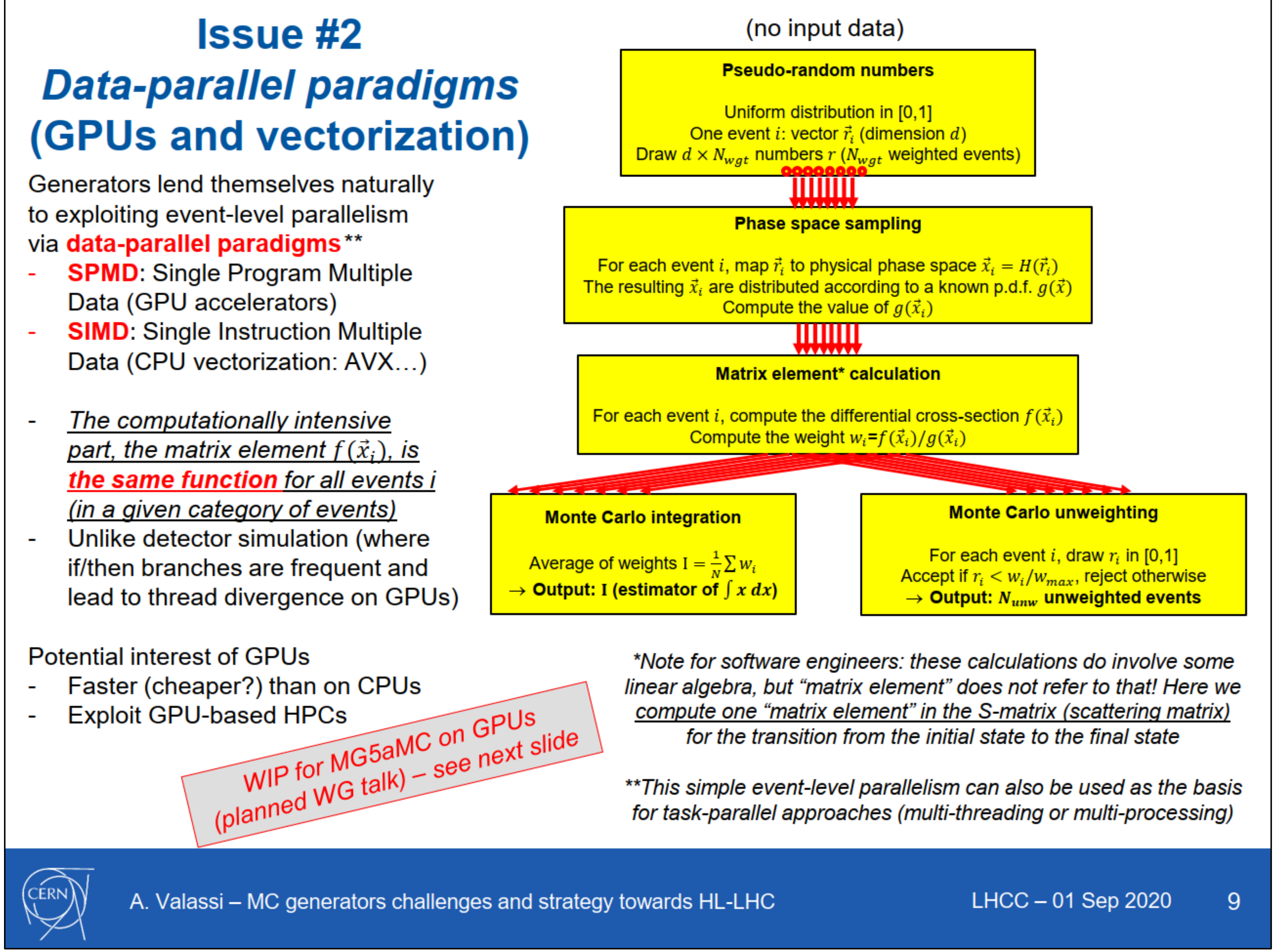
Matching and Merging prescriptions:
aMC@NLO, Powheg, KrkNLO, CKKW, CKKW-L, MLM, MEPS@NLO, MINLO, FxFx, UNLOPS, Herwig7 Matchbox..

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 $\mathrm{I} = \frac{1}{N}\sum w_i$
$\rightarrow$ **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
$\rightarrow$ **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