The HEP Software Foundation Generator WG & Reengineering MadGraph5_aMC@NLO for vector CPUs and GPUs

Andrea Valassi
(CERN IT Department, Scientific Computing Collaborations Group)

Many thanks to:
Josh McFayden, Efe Yazgan and the HSF generator WG
Stefan Roiser, Olivier Mattelaer and the madgraph4gpu development team
Stefano Frixione for useful comments

ECFA Higgs Factories 1st Topical Meeting on Generators
Tuesday 9th Nov 2021 - https://indico.cern.ch/event/1078675

Outline

- The HEP Software Foundation (HSF) and the HSF Generator WG [1,2]
  - LHCC Review of HL-LHC Software and Computing
- Reengineering MG5aMC for vector CPUs and GPUs [3,4]
  - Motivation: more efficient use of limited CPU resources, exploit new architectures
    - Both are general themes also throughout HSF activities and in the HL-LHC Review!
  - Matrix Element generators: ideal compute-intensive kernels for SIMT and SIMD
  - Our implementation in MG5aMC: design, results, status and plans
- Conclusions

[1] Computing and Software in Big Science, May 2021, https://doi.org/10.1007/s41781-021-00055-1
[2] HL-LHC Review, Oct 2021, https://arxiv.org/abs/2109.14938
[3] vCHEP2021 presentation, May 2021, https://indico.cern.ch/event/948465/contributions/4323568
[4] vCHEP2021 proceedings, Aug 2021, https://doi.org/10.1051/epjconf/202125103045

The HEP Software Foundation (HSF)

- A community organization to facilitate coordination and common efforts in High Energy Physics software and computing internationally since 2014

- HEP software is the result of 20+ years of development and must evolve
  - To meet the challenges of *new experimental programmes* (HL-LHC and more!)
  - To meet the technical challenges posed by an *evolving computing landscape*

- Objectives: share expertise; raise awareness of existing software solutions; catalyse common projects; promote commonality and collaboration to make the most of *limited resources*; support training and career development; provide a structure for the community to attract effort and support and help prioritising our work, while promoting collaboration with other sciences...

- 2017: WLCG charge for producing a Community White Paper
  - This resulted in the publication of a Roadmap for HEP Computing
  - Then: engagement in European Strategy Update, Snowmass, HL-LHC review...

- Today: biweekly coordination meetings, many Working Groups – get involved!
  - Browse https://hepsoftwarefoundation.org, register in our discussion forums

Computing and software are gaining the visibility they need

European Strategy Update

2020 Strategy Statements

4. Other essential scientific activities for particle physics

Computing and software infrastructure

- There is a need for strong community-wide coordination for computing and software R&D activities, and for the development of common coordinating structures that will promote coherence in these activities, long-term planning and effective means of exploiting synergies with other disciplines and industry
- A significant role for artificial intelligence is emerging in detector design, detector operation, online data processing and data analysis
- Computing and software are profound R&D topics in their own right and are essential to sustain and enhance particle physics research capabilities
- More experts need to be trained to address the essential needs, especially with the increased data volume and complexity in the upcoming HL-LHC era, and will also help in experiments in adjacent fields.

d) Large-scale data-intensive software and computing infrastructures are an essential ingredient to particle physics research programmes. The community faces major challenges in this area, notably with a view to the HL-LHC. As a result, the software and computing models used in particle physics research must evolve to meet the future needs of the field. *The community must vigorously pursue common, coordinated R&D efforts in collaboration with other fields of science and industry to develop software and computing infrastructures that exploit recent advances in information technology and data science. Further development of internal policies on open data and data preservation should be encouraged, and an adequate level of resources invested in their implementation.*

H. Abramowicz, https://indico.cern.ch/event/924500/

19/06/2020 CERN Council Open Session 24

The HSF Physics Event Generator WG

- WG formed in 2018 after the Physics Event Generator Computing Workshop

- A common forum for discussion and technical work on MC generators in HEP
  - Complementary approach: focus on computational issues, rather than on physics
  - A diverse community of 80+ theorists, experimentalists, software engineers
  - Meetings (16 so far, all minuted) on https://indico.cern.ch/category/8460/
  - Contact hsf-generator-wg-convenors@googlegroups.com to get involved!

- Main activity in 2020-2021: LHCC review of HL-LHC software and computing
  - First stage of the review last year – report (July 2020) available here
  - Second stage of the review completed last week (Nov 2021)!
    - Generators one of 7 areas – with ATLAS, CMS, simulation, DOMA, ROOT, analysis

- Two recent publications of the WG in the context of the LHCC review
  - CSBS paper, May 2021 – doi:10.1007/s41781-021-00055-1
    - A detailed review of (technical and human) computational challenges in MC generators
  - LHCC document, Oct 2021 – arxiv:2109.14938
    - Summarizes inputs received by many generator teams during dedicated WG meetings

Computing and Software for Big Science (2021) 5:12
https://doi.org/10.1007/s41781-021-00055-1

ORIGINAL ARTICLE

Check for updates

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

The HSF Physics Event Generator WG · Andrea Valassi[1] · Efe Yazgan[2] · Josh McFayden[1,3,4] · Simone Amoroso[5] · Joshua Bendavid[1] · Andy Buckley[6] · Matteo Cacciari[7,8] · Taylor Childers[9] · Vitaliano Ciulli[10] · Rikkert Frederix[11] · Stefano Frixione[12] · Francesco Giuli[13] · Alexander Grohsjean[5] · Christian Gütschow[14] · Stefan Höche[15] · Walter Hopkins[9] · Philip Ilten[16,17] · Dmitri Konstantinov[18] · Frank Krauss[19] · Qiang Li[20] · Leif Lönnblad[11] · Fabio Maltoni[21,22] · Michelangelo Mangano[1] · Zach Marshall[3] · Olivier Mattelaer[22] · Javier Fernandez Menendez[23] · Stephen Mrenna[15] · Servesh Muralidharan[1,9] · Tobias Neumann[14,24] · Simon Plätzer[25] · Stefan Prestel[11] · Stefan Roiser[1] · Marek Schönherr[19] · Holger Schulz[17] · Markus Schulz[1] · Elizabeth Sexton-Kennedy[15] · Frank Siegert[26] · Andrzej Siódmok[27] · Graeme A. Stewart[1]

HL-LHC Computing Review Stage-2, Common Software Projects: Event Generators

arxiv:2109.14938

The HSF Physics Event Generator WG
Efe Yazgan (editor)[1] Josh McFayden (editor)[2] Andrea Valassi (editor)[3]
Simone Amoroso[4] Enrico Bothmann[5] Andy Buckley[6] John Campbell[7]
Gurpreet Singh Chahal[8,9] Taylor Childers[10] Gloria Corti[3] Rikkert Frederix[11]
Stefano Frixione[12] Francesco Giuli[13] Alexander Grohsjean[4] Stefan Hoeche[7]
Phil Ilten[14,15] Frank Krauss[9] Michal Kreps[16] David Lange[17] Leif Lonnblad[11]
Zach Marshall[18] Olivier Mattelaer[19] Stephen Mrenna[7] Paolo Nason[20]
Simon Plaetzer[21] Christian Preuss[22] Emanuele Re[23] Stefan Roiser[3]
Marek Schoenherr[9] Steffen Schumann[5] Markus Seidel[24]
Elizabeth Sexton-Kennedy[7] Frank Siegert[25] Andrzej Siodmok[26]
Graeme A. Stewart[3] Aravind Thachayath Sugunan[27] Zbigniew Was[26]

- Focus on software and computing aspects rather than on physics precision “per se”
- Technical challenges – mainly, the costs associated to large scale event productions
  - Many (sometimes large) inefficiencies ⇒ many opportunities for (large) speedups!
    - Inefficiency in phase space sampling algorithms (many *events rejected during unweighting*)
    - *Inefficient software implementation (CPU SIMD and GPUs not yet used)* – *the rest of this talk!*
    - *Negative weights due to QCD NLO matching* mean more unweighted events to generate
  - Can we predict the cost (in CPU time) of increased physics precision (e.g. NLO to NNLO)?
  - Need reproducible benchmarks and detailed profiling of (per-process, per-generator) costs
- Human and collaborative challenges – another essential part of the same problem
  - *Training* on new software development paradigms – the HSF is very active in this area
  - Promote collaboration of theorists, experimentalists, software engineers – *the rest of this talk!*
    - Also promote easier *plug-and-play exchange of software components* via agreed APIs? Mentioned in WG meetings
  - Career paths for S&C work with limited physics content – *keep it in mind for future colliders!*

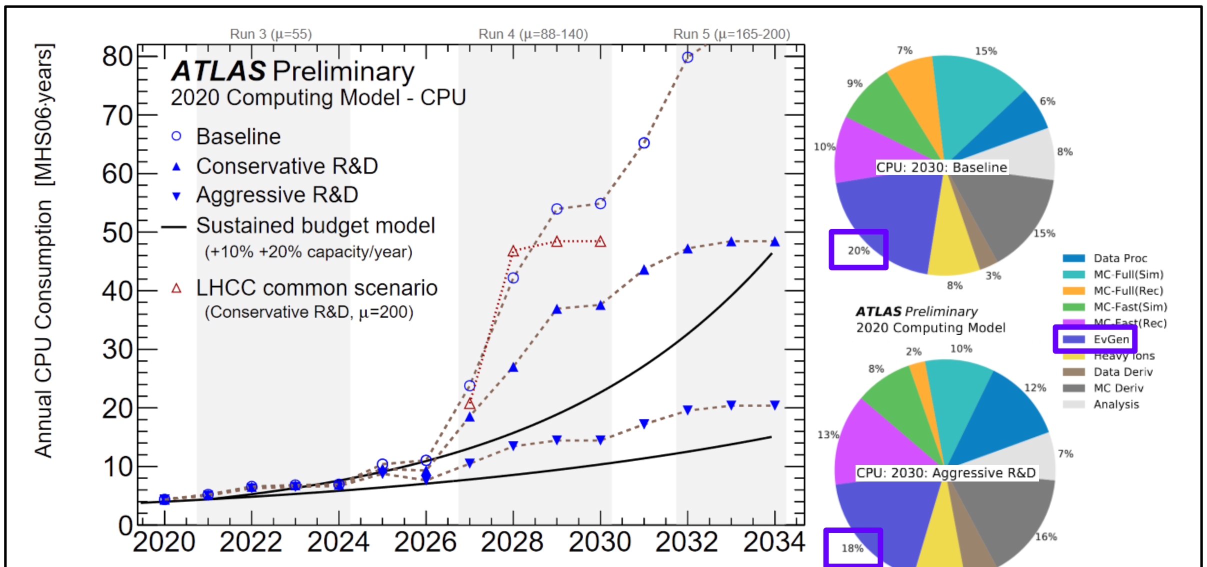
MG5aMC on GPU – project overview

- A joint effort started in Q1 2020 in the context of the HSF Generator WG
  - *Nice collaboration of theoretical / experimental physicists and software engineers!*
  - Initial team: *Oliver Mattelaer (Louvain)*, AV, Stefan Roiser (CERN)
    - *Main focus: port to Nvidia GPUs using CUDA; in parallel, optimisation of C++ backend*
  - Many contributors from different institutes joined (and left) over time:
    - Alphabetically: Smita Darmora, Taylor Childers, Tyler Burch, Walter Hopkins (Argonne), Taran Singhania (Bangalore), Vince Pascuzzi (Berkeley) Andreas Reepschlaeger, David Smith, Laurence Field, Stephan Hageboek (CERN), Carl Vuosalo (Madison), Josh McFayden (Sussex), Nathan Nichols (Vermont)
    - Tests/CI, performance plots/profiling, *abstraction layers (Kokkos, Alpaka, Sycl)*
  - Project is maintained on https://github.com/madgraph5/madgraph4gpu
    - Upstream MG5aMC is on https://launchpad.net/mg5amcnlo
    - Regular meetings every two weeks (overall activity coordination: SR)

- Why did we choose to focus on MG5aMC for a GPU port? Two reasons:
  - Earlier efforts at KEK in 2010-2013, not released for production (see HOW2019)
    - We are not leveraging on this work (based on an old version of MG5aMC's ME library)
  - Main reason: active involvement of core MG5aMC developer (OM)
  - *NB1: many of the design ideas we describe are applicable to other generators...*
  - *NB2: focusing on LO only for the moment (no MC@NLO merging yet...)*

Motivation #1 : speeding up code (including MC generators)

- Projected experiment needs at HL-LHC exceed projected available resources
  - For both CPU and storage – *need R&D to overcome this computing resource gap*
- Speeding up MC event generators is an important R&D goal (e.g. for ATLAS)
  - MC generators are essential for (HL-)LHC physics and large CPU time consumers
  - *Many inefficiencies*, both in algorithms (sampling,...) and implementation (SIMD,...)



ATLAS HL-LHC Computing CDR (2020)

ATLAS: GEN is projected to be ~20% of total CPU budget
(Aggressive R&D: *#events/second multiplied by x2*, generate fewer events)

Will MC generator workloads be large consumers of CPU time at future e^+e^- Higgs factories?

Maybe not (relatively simpler processes than at LHC?), but you'd better think of this upfront!
(And inefficiencies are always a waste!)

Motivation #2 : GPUs, vector CPUs – underexploited in HEP

- *GPUs provide most of the compute power in recent HPCs (e.g. Summit: 95%)*
  - Supercomputers at HPC centers: large opportunistic use by LHC experiments
  - But *only a small share of HEP software workloads can run on GPUs today*
- *Most WLCG CPUs support vectorization (SSE4.2, AVX2 or above)*
  - But *only a small share of HEP software workloads exploit CPU vectorization today*

⇒ **Can we exploit GPUs (and CPU vectorization) in MC event generators?**

The work described in the rest of this talk addresses this question for MG5aMC

https://www.flickr.com/photos/olcf

The computing hardware landscape is in continuous evolution!
Vector CPUs, GPUs, HPCs, FPGAs (and more yet-unknown platforms!) will most certainly be relevant at future e^+e^- Higgs factories...

You'd better plan upfront for a very heterogeneous computing!

What is a MC generator? A simplified computational anatomy

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

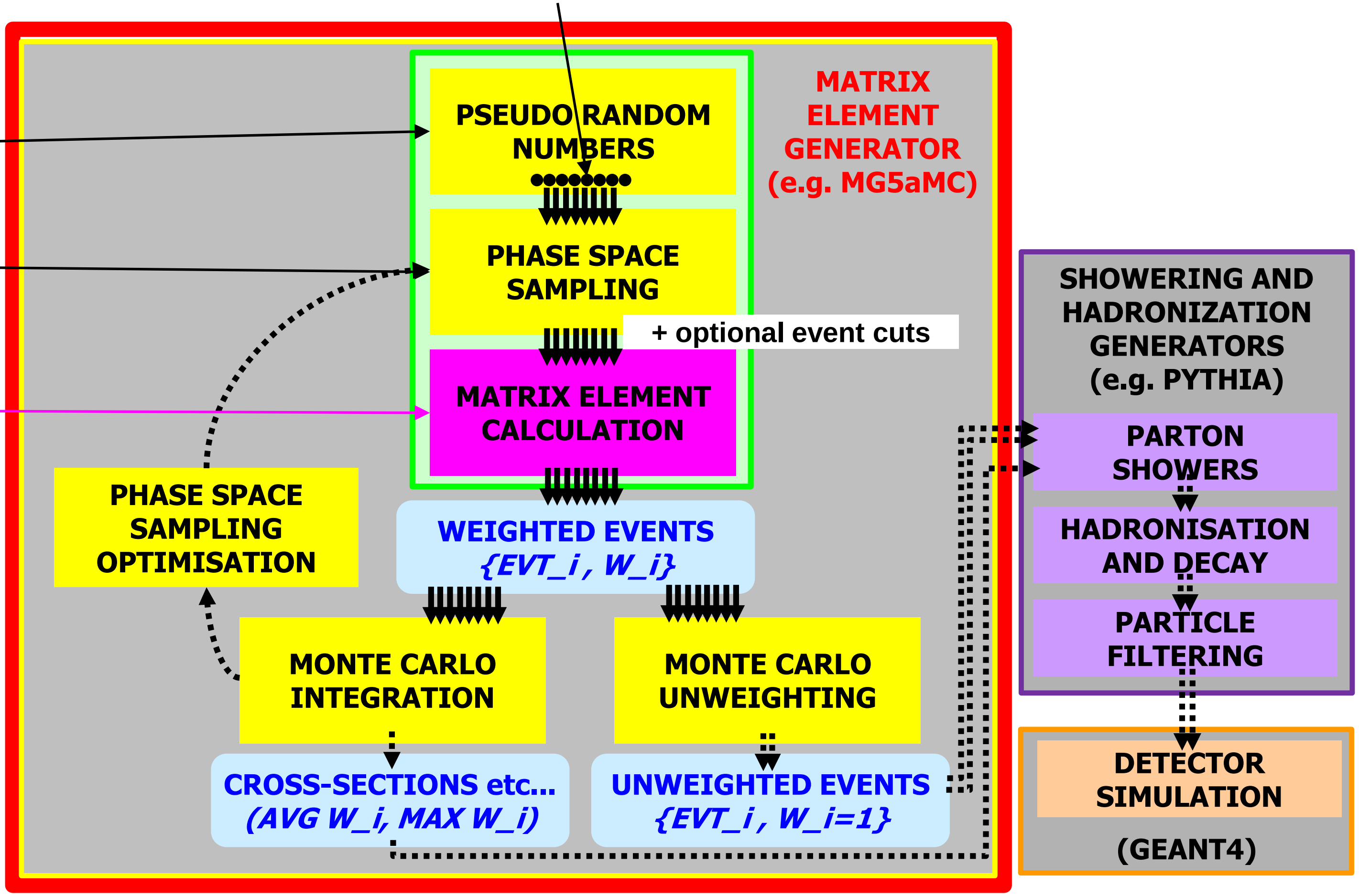
This can be parallelized (SIMT/SIMD and multithreading)

For each event:

1\.
Output: random numbers

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

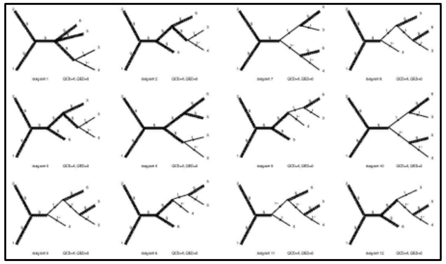
3\.
Input: particle 4-momenta
Output: Matrix Element (ME)
CPU BOTTLENECK



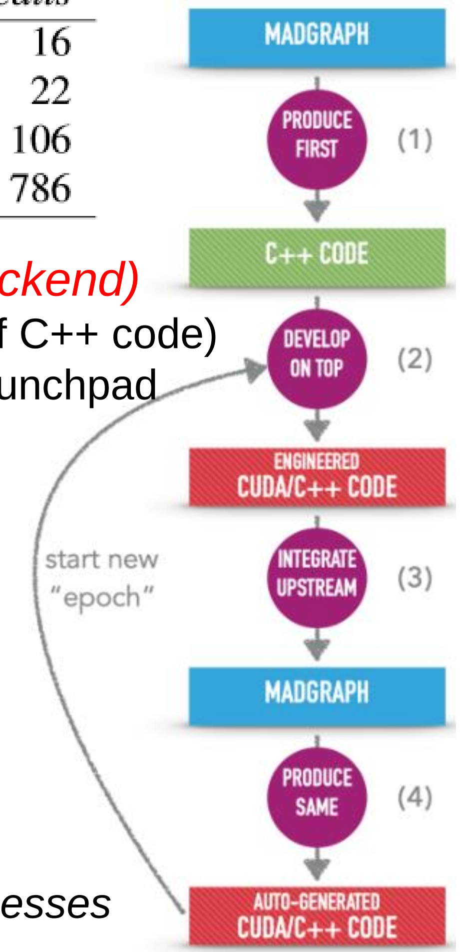
(NB: Matrix Element is an element of the scattering matrix... almost no linear algebra here!)

Code is auto-generated ⇒ Iterative development process

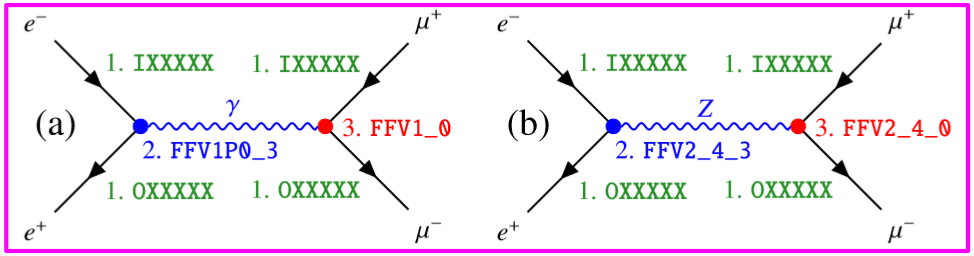
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Fortran (default), C++, or Python – all generated by Python code-generating meta-code
  - The more particles in the collision, the more Feynman diagrams and the more lines of code



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 meta-code in launchpad
  - (4) *Generate more complex LHC processes like* $gg \to t\bar{t}gg$
  - Add missing functionality, fix issues, improve performance, *iterate*

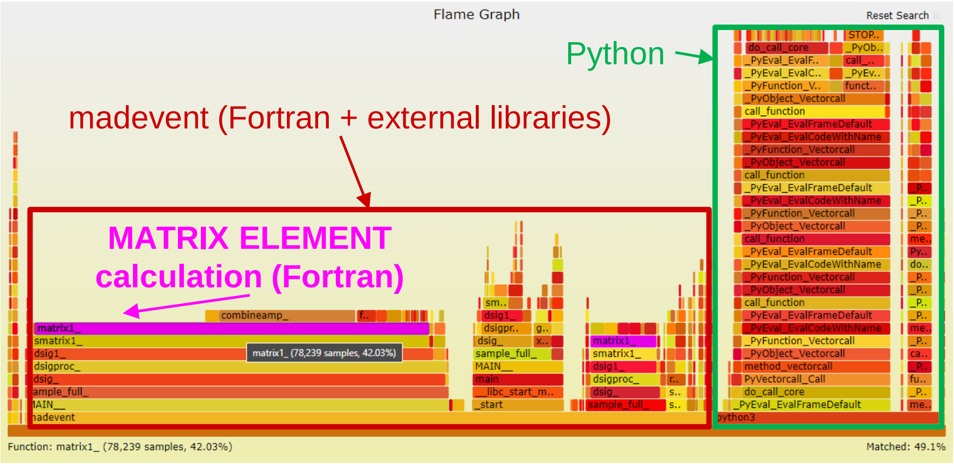


- ***NEW (Oct 2021): Python code-gen plugin is also in github***
  - *Much faster iterations to port features (e.g. vectorization) to ~all processes*

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)

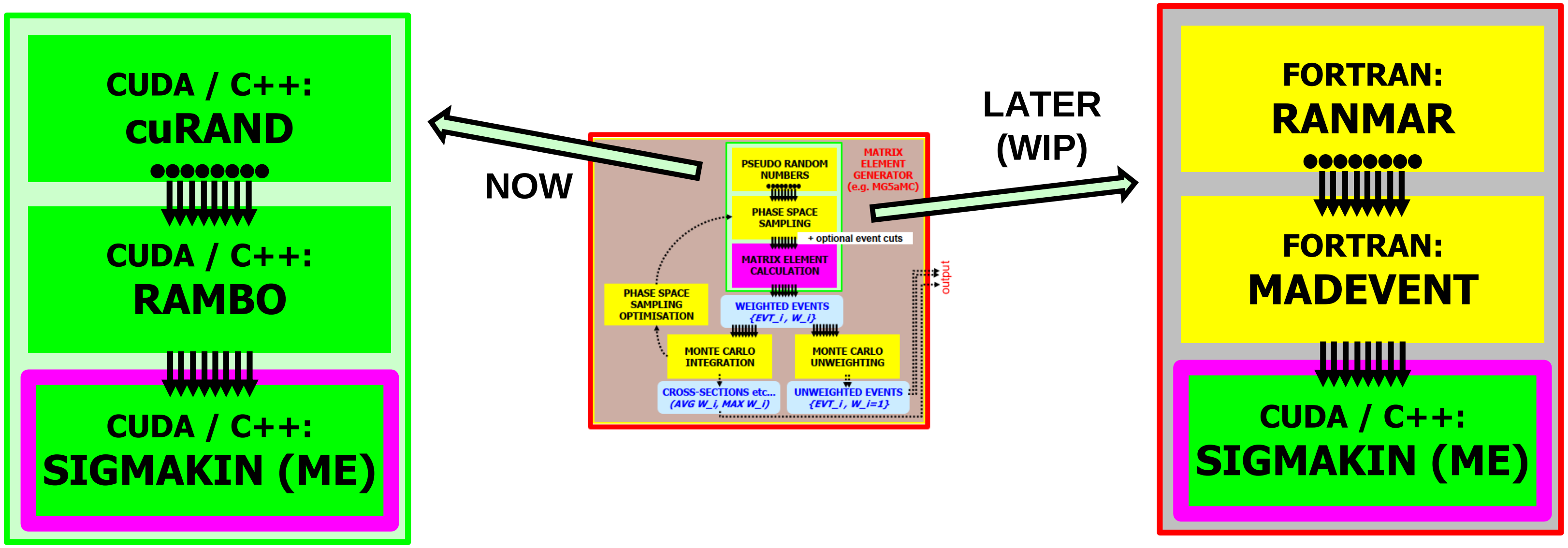
Our main focus is the ME calculation: develop new CUDA implementation (and speed up existing C++)

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 presented in this talk come from this framework*

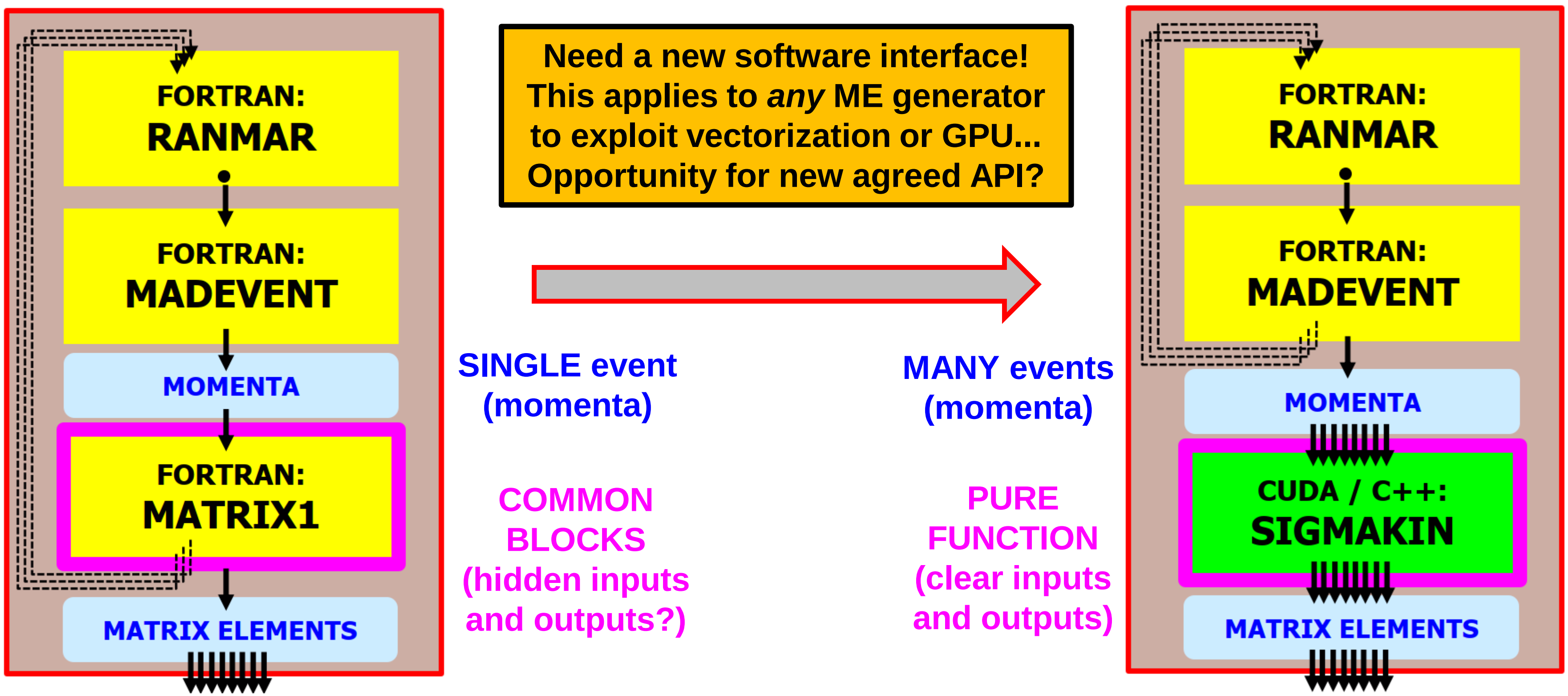
cuRAND: identical random number sequences on host (CPU) and device (GPU), allowing CUDA/C++ bitwise comparisons



- Our plan (in progress): *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

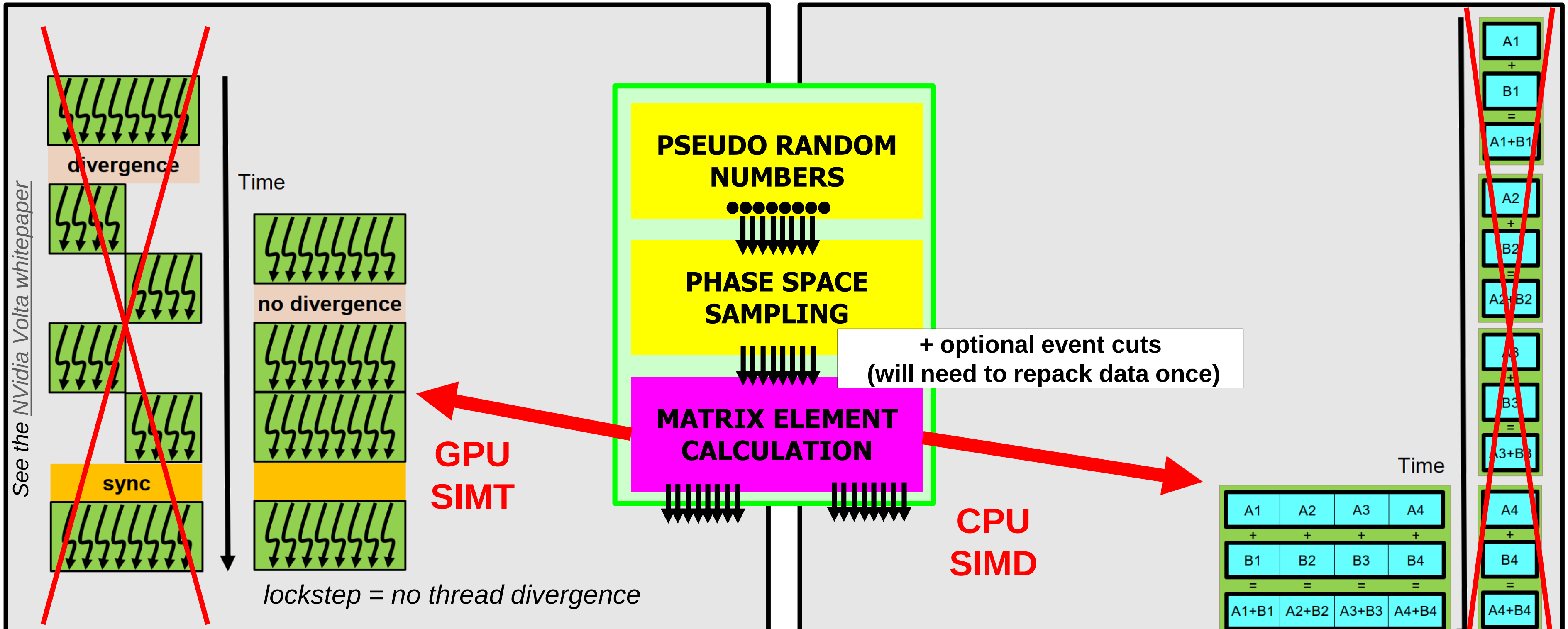
WIP: interfacing with Fortran MadEvent – potential challenges

- Linking Fortran and C++ should be easy (just transpose multidimensional array indexe
- From a first look at MadEvent: two potential challenges (legacy code reengineering)
  - *(1) must <u>create event baskets a posteriori</u>* (current code loops on individual events)
  - *(2) <u>Fortran common blocks</u> complicate separation of inputs and outputs?* (not pure functions)



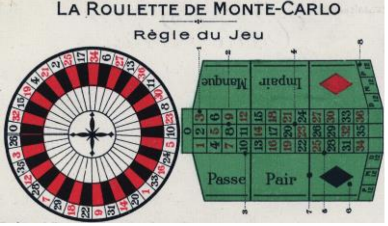
Main design idea: event-level data parallelism (lockstep)

- In MC generators, all events *in one channel* initially go through the same calculations
  - *Computing MEs involves the calculation of the exact same function on different data points*
  - This is what *makes event generators a good fit for GPUs (SIMT) and vector CPUs (SIMD)*



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...)

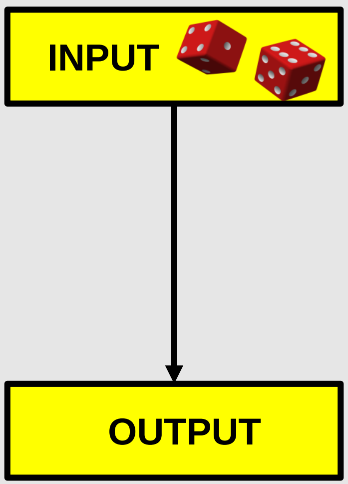


Aside – Monte Carlo’s: what about branching?

- *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*:

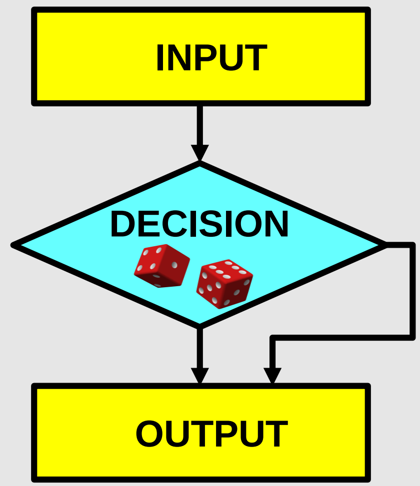
MC SAMPLING (within one channel)

Physics generators:
- MC integration (cross sections)
- MC generation (event samples)



Lockstep processing
Good for SIMT/SIMD

NB: the CPU-intensive ME calculation comes <u>before</u> 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?)

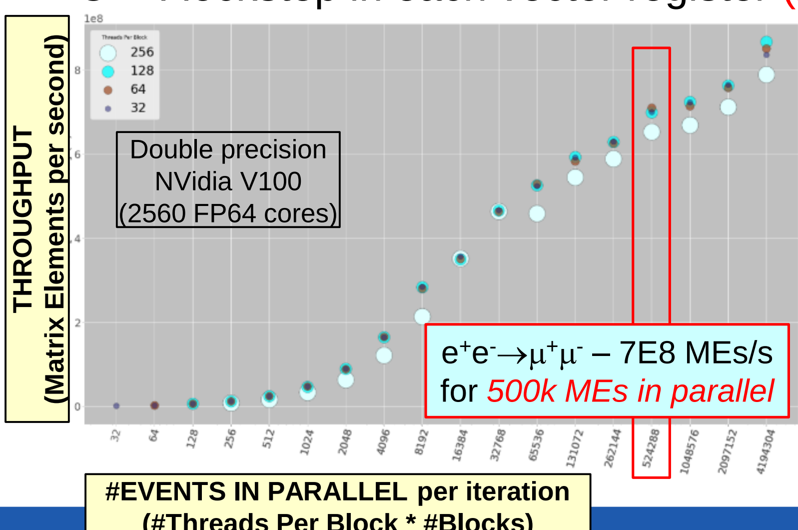
MC detector simulation
- Particle/matter interaction (when? how?)
- Particle decays (when?)

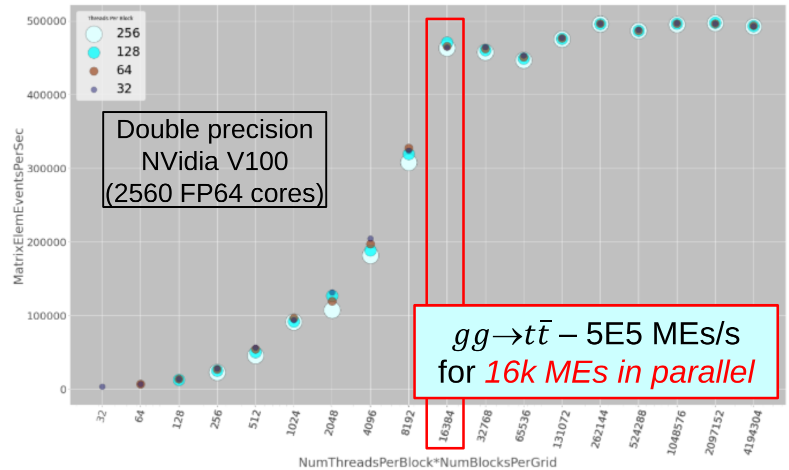
Event-level parallelism in practice – coding and #events

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





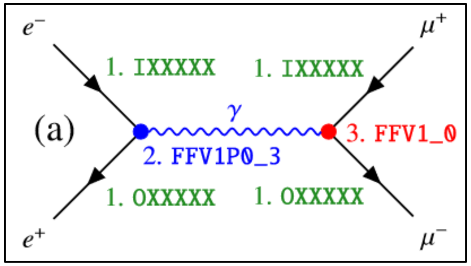
(plots – Andy Reepschlaeger)

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
NEW (Oct 2021): now 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
```

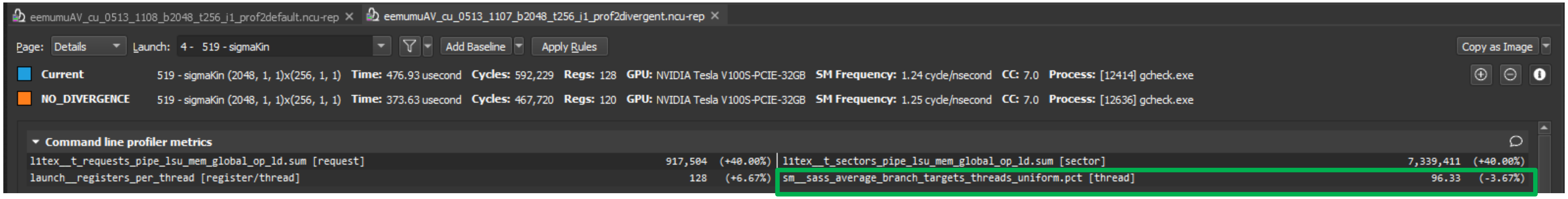
CUDA: Profiling with NVidia NSight Compute – ncu

- 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



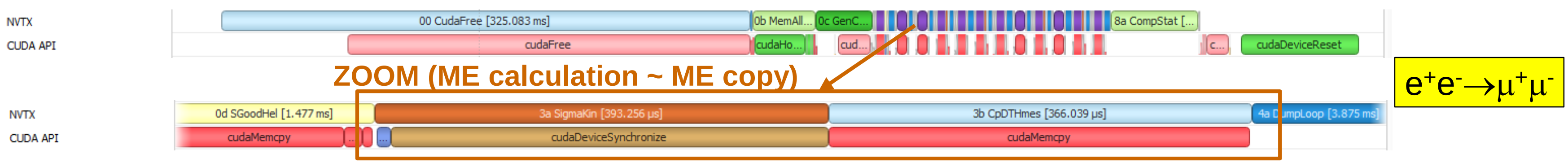
Example: compare baseline implementation (100% branch efficiency) to a test with artificial divergence

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

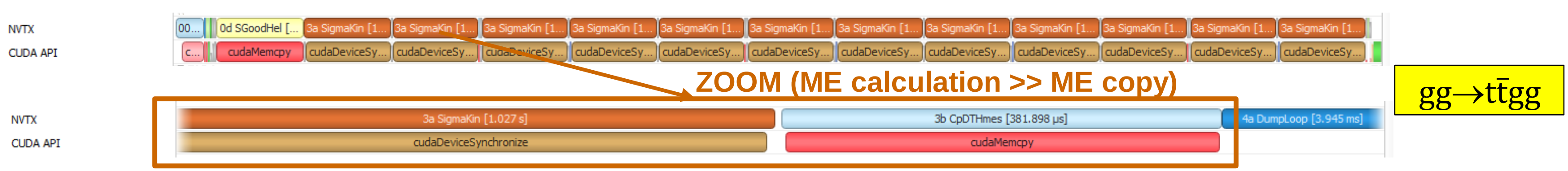
- In our standalone application (all on GPU): momenta, weights, MEs D-to-H
  - Plots below from Nvidia Nsight Systems: 12 iterations with 524k events in each iteration

- 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*



Summary of (preliminary) throughput results

Being re-checked: probably this Fortran implementation uses a different (2x faster) algorithm – helicity recycling?

C++ vectorization double speedup x4.2
- Achieves theoretical limit of x4 for 256-bit
- Further WIP on 512-bit with x8 theoretical

C++ vectorization float speedup x7.7
- Achieves theoretical limit of x8 for 256-bit
- *Twice as many floats as doubles in SIMD!*

CUDA V100: ~x300 over 1-core C++
- There is room for further improvements
- $e^+e^- \rightarrow \mu^+\mu^-$ was x2 better (fewer registers)
- Need to optimize QCD color algebra

CUDA V100: float x2 faster than double
- Similar to CPU SIMD, different reasons
- *V100 Flops (&cores): FP32 = 2x FP64*
  - *NB: much fewer FP64 on consumer cards!*
  - *e.g. FP32 ~ 32x FP64 on T4 cards*

Implementation ($gg \rightarrow t\bar{t}gg$)	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)**

(CUDA11.1 and gcc10.2)

* https://www.techpowerup.com/gpu-specs/tesla-t4.c3316
https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184

Overview of work in progress and plans

- *Backport ME abstraction layers to code-generating meta-code*
  - Kokkos (T. Childers) is ~done; Alpaka (D. Smith) is progressing well; Sycl is also WIP
  - Will allow a detailed performance comparison to native CUDA/C++
  - Extend native CUDA to native HIP on AMD GPUs and compare to abstraction layers
- *Integration of CUDA/C++ ME with Fortran MadEvent*
  - Improve CUDA/C++ encapsulation, split Fortran single-event loops, review common blocks
- Further performance optimizations of ME kernel
  - CUDA: split sigmakin, *reduce register pressure, CUDA graphs*, investigate tensor cores
  - C++: review AVX512 vectorization
- Improve task parallelization and orchestration
  - C++ multithreading, heterogeneous CPU/GPU workloads, optimize 'whole node' throughput
  - Also: collaborate with HEPIX benchmarking WG on compute benchmark (prototype exists)
- Not yet started: deal with even more complex (and relevant to LHC) physics
  - pp collisions: many subprocesses and interface with PDFs
  - NLO precision (including loop calculations), matching to parton showers

Conclusions

- HSF Generator WG: a pleasant collaboration of theorists, experimentalists, engineers
  - Focus on computational challenges rather than on physics – you are all welcome to join!
  - *May be relevant for Michelangelo's Future Collider Unit? (do not forget software/computing!)*
  - MG5aMC reengineering project was born in this context

- *We demonstrated the potential of GPUs and vector CPUs for any ME event generator*
  - ME calculation is the main CPU consumer and can largely be executed in lockstep
  - CUDA on NVidia V100 is ~ x300 faster than one CPU core and shows *no thread divergence*
  - We see *almost a factor 4 speedup over scalar C++ from SIMD* with 256-bit registers

- *We plan to interface this in MG5aMC for production use by the LHC experiments*
  - Keep the (mainly) Fortran outer shell and replace the ME calculation by our CUDA/C++
  - A few other ingredients still missing for the LHC experiments (PDFs, NLO...)

- *Floating-point numerical precision is an important issue (do experiments need double?)*
  - Moving from double to float would gain a factor >2 both on GPUs and on SIMD CPUs
  - Can we avoid NaN's in MEs with float? Is fast math ok?
    - Note – sometimes (for a small fraction of phase space points) need quadruple precision for NLO loops

- Work on abstraction layers is progressing well – comparison to Kokkos coming soon
  - An attractive option (not the only one) to enlarge our work from NVidia to AMD or Intel GPUs
  - Note: alternative approaches are also being worked on by the MadFlow team

Backup slides

A complex and heterogeneous problem

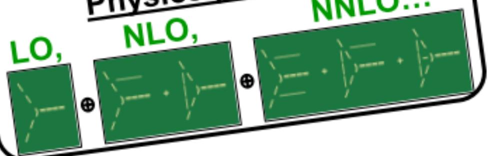
Sampling algorithms:
Vegas, Miser, Rambo, Bases/Spring, Mint, Foam, Vamp, MadEvent, Comix…

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..

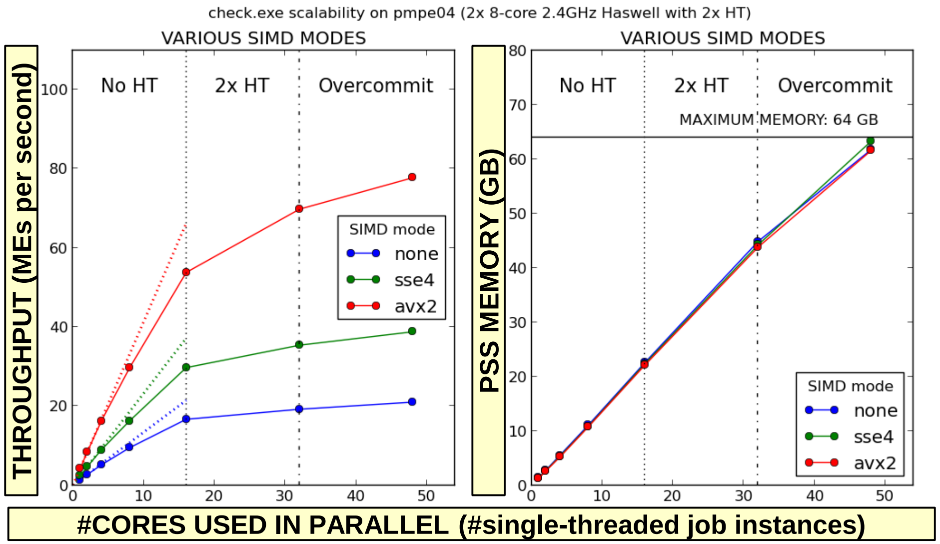
Hadronization and Parton Showers:
Pythia, Herwig, Ariadne…

- 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"?

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

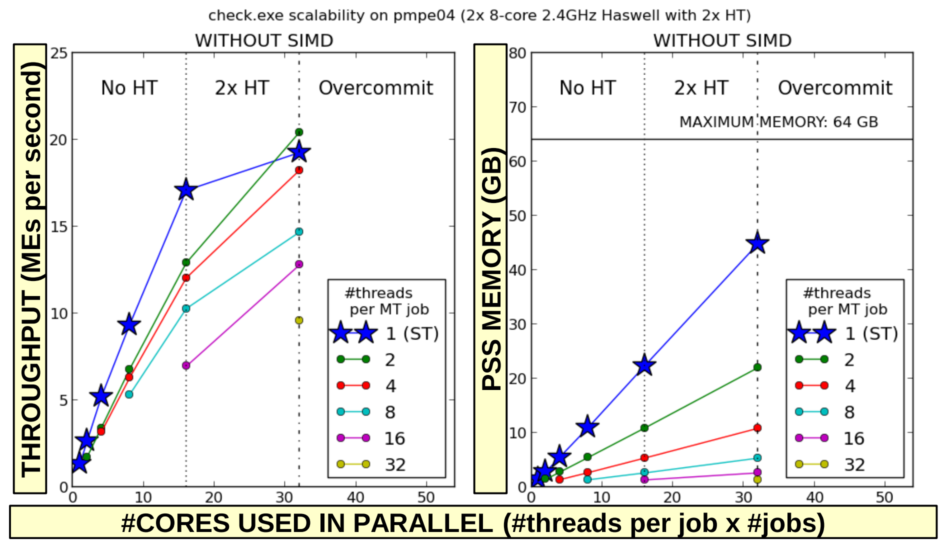
CPU throughput plots – SIMD + multi-core

- *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



CPU throughput results (1)
Double, Scalar – Fortran vs C++

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)**

- *C++ is only 15% slower than Fortran*
- Results on 1 core of a Skylake-AVX512 CPU (Intel Xeon Silver 4216)
- VM running CentOS8, same compiler (gcc9) and compiler flags (-O3 -ffast-math)
- Take this with a grain of salt: *not an apple-to-apple comparison!*
  - Fortran: MadEvent framework instrumented with timers
  - C++: standalone toy framework using Rambo
  - Slightly different versions of upstream MG5aMC (slightly different algorithms)

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

CPU throughput results (3)
C++, SIMD – Double vs Float

Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double	MEs / second Float
1-core MadEvent Fortran scalar	1.50E6 (x1.15)	---
1-core Standalone C++ scalar	1.31E6 **(x1.00)**	1.21E6 **(x0.92)** ***[x1.00]***
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	2.52E6 (x1.9)	4.50E6 **(x3.4)** ***[x3.7]***
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	4.58E6 (x3.5)	8.17E6 **(x6.2)** ***[x6.8]***
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	4.91E6 (x3.7)	8.84E6 **(x6.7)** ***[x7.3]***
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	3.74E6 (x2.9)	7.42E6 **(x5.7)** ***[x6.1]***

- Scalar: float slower than double
  - To be understood (8% effect)
- *SIMD: float ~ x2 better than double!*
  - Execute ½ as many vector instructions
  - Best throughput: 256-bit AVX512 *(x7.3 speedup against x8 theoretical maximum)*
- *Is single precision enough for physics?* Can we improve numerical stability?
  - Observed a few NaN every million MEs when using single precision
  - Using fast math (~x2 speedup) also requires excellent control of numerical stability

GPU throughput results (1)
Double – C++ vs CUDA (V100)

- Full *V100 GPU ~x600 faster than one CPU core*
- Just a preliminary ballpark indication!
  - CUDA: ½ of the time spent in data copy ($e^+e^- \to \mu^+\mu^-$)
  - CUDA: can optimize #threads*#blocks (here:524k)
  - CUDA: should optimize scheduling and registers
  - CPU: should use vectorization
  - CPU: should use all cores (e.g. multi-threading)

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, x4 floats)	2.52E6 (x1.9)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	4.58E6 (x3.5)
1-core Standalone C++ “256-bit” AVX512 (x4 doubles, x8 floats)	4.91E6 (x3.7)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	3.74E6 (x2.9)
Standalone CUDA NVidia V100S-PCIE-32GB (2560 FP64 cores*)	7.25E8 **(x550)**

* https://www.techpowerup.com/gpu-specs/tesla-t4.c3316

GPU throughput results (2)

CUDA – Double vs Float
(and NVidia V100 vs T4)

- *V100: float ~ x2.2 better than double!*
  - Similar to CPU SIMD, different reasons
  - V100 Flops (&cores): FP32 = *2x* FP64
  - Fewer registers: float=48, double=120
- *T4: very limited double performance*
  - T4 Flops: FP32 = *32x* FP64
  - May be even worse in consumer cards

Implementation ($e^+e^- \to \mu^+\mu^-$)	MEs / second Double	MEs / second Float
1-core MadEvent Fortran scalar	1.50E6 (x1.15)	---
1-core Standalone C++ scalar	1.31E6 **(x1.00)**	1.21E6 (x0.92)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	2.52E6 (x1.9)	4.50E6 (x3.4)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	4.58E6 (x3.5)	8.17E6 (x6.2)
1-core Standalone C++ “256-bit” AVX512 (x4 doubles, x8 floats)	4.91E6 (x3.7)	8.84E6 (x6.7)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	3.74E6 (x2.9)	7.42E6 (x5.7)
Standalone CUDA NVidia V100S-PCIE-32GB (TFlops*: 7.1 FP64, 14.1 FP32)	7.25E8 **(x550)**	1.59E9 **(x1200)**
Standalone CUDA NVidia T4 (TFlops*: 0.25 FP64, 8.1 FP32)	3.21E7 **(x25)**	6.52E8 **(x500)**

* https://www.techpowerup.com/gpu-specs/tesla-t4.c3316
https://www.techpowerup.com/gpu-specs/tesla-v100-pcie-32-gb.c3184

Throughput summary table

524k eemumu events per iteration

CPU single-thread scalar, double precision, “-O3 -ffast-math”:
C++ (reference value) is only 15% slower than Fortran
[WARNING: different contexts, Rambo vs MadEvent]

CPU with C++ SIMD vectorization:
AVX2 / 256-bit AVX512 (4 doubles/vector) are
a factor 3.5 / 3.8 faster than scalar code

CPU with C++ SIMD vectorization:
AVX2 / 256-bit AVX512 (8 floats/vector) are
a factor 6.2 / 6.8 faster than scalar double code
(a factor 6.8 / 7.3 faster than scalar float code)

Description	*Compiler flags*	*Register width*	*Throughput MEs/sec* *Double*	*Float*
MadEvent Fortran (scalar)	—	— (x1 double, x1 float)	1.50E6 (x1.15)	—
Standalone C++ “none” (scalar)	—	— (x1 double, x1 float)	1.31E6 **(x1.00)**	1.21E6 (x0.92)
Standalone C++ “sse4” (SSE4.2)	-march=nehalem	128 bits (x2 double, x4 float)	2.52E6 (x1.92)	4.50E6 (x3.43)
Standalone C++ “avx2” (AVX2)	-march=haswell	256 bits (x4 double, x8 float)	4.58E6 (x3.50)	8.17E6 (x6.24)
Standalone C++ “512y” (256bit AVX512VL)	-march=skylake-avx512 -mprefer-vector-width=256	256 bits (x4 double, x8 float)	4.91E6 (x3.75)	8.84E6 (x6.75)
Standalone C++ “512z” (AVX512VL)	-march=skylake-avx512 -DMGONGPU_PVW512	512 bits (x8 double, x16 float)	3.74E6 (x2.85)	7.42E6 (x5.66)
Standalone CUDA NVidia V100	—	—	7.25E8 (x550)	1.59E9 (x1210)
Standalone CUDA NVidia T4	—	—	3.21E7 (x25)	6.52E8 (x500)

Table 1: Throughputs (matrix elements per second) for eemumu. For Fortran: estimates from MATRIX1 in MadEvent. For C++ and CUDA: measurements from the epoch1 standalone executables, over 12 iterations with 524k events (2048 blocks, 256 threads per block in CUDA), as of commit 51d7f52bf3 on May 04. Compilers: gcc9.2 and CUDA11.0. All builds use “-O3” and “-ffast-math” or “-use_fast_math”. Virtual machine itscrd70 (Fortran, C++ and CUDA/V100 results): skylake-avx512 CPU (Intel Xeon Silver 4216) with 4 virtual cores, NVidia V100 GPU. Virtual machine lxplus770 (CUDA/T4 results): skylake-avx512 CPU (Intel Xeon Silver 4216) with 4 virtual cores, NVidia T4 GPU. Fortran and C++ throughputs use a single CPU core. CUDA throughputs include device-to-host copies of all matrix element values.

CPU with C++ SIMD vectorization:
512-bit AVX512 slower than 256-bit
(CPU clock slowdown... what else?)

GPU with CUDA, single precision:
V100: float is 2.2x faster than double
(T4: float is 20x faster than double!)

GPU with CUDA, double precision:
V100 a factor 550 faster than 1 CPU core
(T4 performs poorly – limited FP64)

The MadGraph5_aMC@NLO (MGaMC) event generator

- Software framework for phenomenological studies of HEP collision processes
  - Both within the Standard Model (SM) and beyond (BSM)
  - Computation of cross sections and generation of hard events
  - At tree level (LO) and at next-to-leading order (NLO)
  - NLO matching to parton shower (PS) simulations
  - Merging of matched samples with different numbers of jets
  - Uses some external libraries (parton distribution functions, Feynman loops...)

- Essential tool for the LHC experiments, well established in ATLAS and CMS

- *In our work, so far we used a subset of its features*
  - Individual processes, no merging
  - Only LO processes, no matching
  - No PDFs (and limited use of QCD so far)
  - *We start simple...*

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]
[a] *Department of Physics, National Taiwan University, Taipei 10617, Taiwan*
[b] *PH Department, TH Unit, CERN, CH-1211 Geneva 23, Switzerland*
[c] *SLAC National Accelerator Laboratory, 2575 Sand Hill Road, Menlo Park, CA 94025-7090 U.S.A.*
[d] *CP3, Université Catholique de Louvain, B-1348 Louvain-la-Neuve, Belgium*
[e] *Department of Physics and State Key Laboratory of Nuclear Physics and Technology, Peking University, Beijing 100871, China*
[f] *University of Illinois, Urbana, IL 61801-3080, U.S.A.*
[g] *Physik-Institut, Universität Zürich, Winterthurerstrasse 190, 8057 Zurich, Switzerland*
[h] *Sorbonne Universités, UPMC Univ. Paris 06, UMR 7589, LPTHE, F-75005, Paris, France*
[i] *CNRS, UMR 7589, LPTHE, F-75005, Paris, France*
E-mail: johan.alwall@gmail.com, rikkert.frederix@cern.ch, stefano.frixione@cern.ch, vahirsch@slac.stanford.edu, fabio.maltoni@uclouvain.be, olivier.mattelaer@uclouvain.be, huasheng.shao@cern.ch, tstelzer@illinois.edu, torriell@physik.uzh.ch, zaro@lpthe.jussieu.fr

JHEP07(2014)079