Speeding up Madgraph5_aMC@NLO through CPU vectorization and GPUs: status and lessons learnt

Andrea Valassi (CERN IT)

With many thanks to the whole madgraph4gpu development team!

Les Houches, 16th June 2023

Overview – a few slides to have an open discussion

- Numerical results (not yet the full workflow) to show the potential of GPUs and CPU vectorization

- For users in the experiments: status of developments
  - As in the past, "we are almost there"... but this time we are much closer
  - Many useful discussions already in the last few days (thanks!), and we are here also next week...

- For developers of other MC tools: some lessons learnt
  - Move from a single-event API to a data-parallel multi-event API
  - Develop re-entrant functions with well defined inputs/outputs and avoid common-block status
  - The above points are a must inside new MCs and for inter-MC integration (eg MG5AMC + POWHEG)
    - "Et plus si affinités"... any interest in a modular software framework?

- Outlook and challenges
  - We are only at (QCD) LO – we are starting to address NLO this year

FEBRUARY 2023
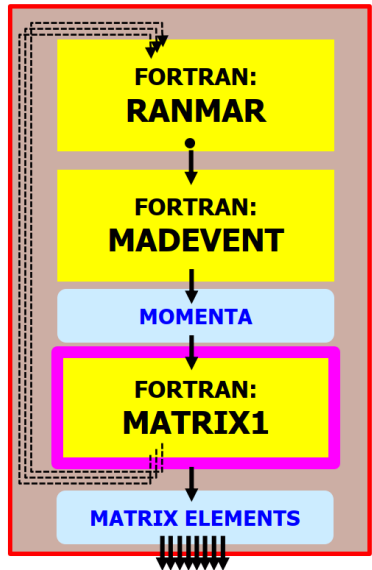# COMPUTE ACCELERATOR FORUM TALK,
"THE 4-SLIDE VERSION"

The 4-slide version! – Executive summary for the impatient

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

- Our reengineering of MG5aMC is close to a first fully functional alpha release for LO QCD processes
  - *The new ME calculation is integrated in MadEvent* – we get the same cross section *and LHE files* as in Fortran!

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

- On GPUs, using CUDA we *achieve O(100-1000) speedups for MEs alone over one no-SIMD CPU core*
  - The speedups may be much lower due to *Amdahl's law*, but we are improving on that
  - Example: our *current overall speedup is x60/x100 (double/float)* for $gg \to t\bar{t}ggg$ on an NVidia V100

- Floats are x2 faster than doubles in SIMD and NVidia GPUs – we also added 'mixed' precision modes

- In SYCL we get ~similar performances to CUDA on NVidia and we may run also on AMD or Intel GPUs

- Future challenges include optimizing heterogenous processing on one GPU and multiple CPU cores
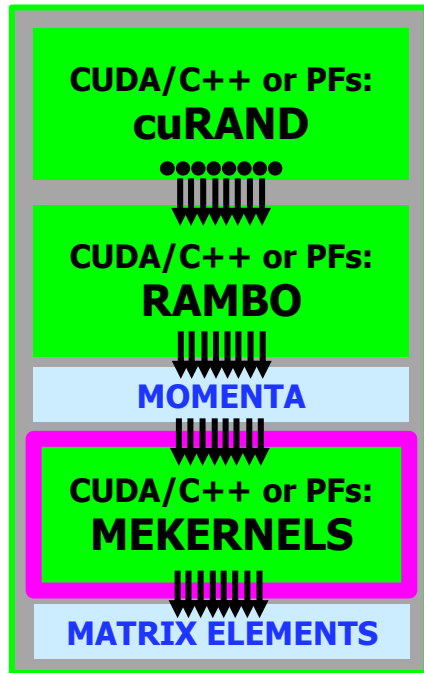
MG5aMC: old and new architecture designs

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

| FORTRAN: RANMAR |
| FORTRAN: MADEVENT |
| MOMENTA |
| FORTRAN: MATRIX1 |
| MATRIX ELEMENTS |

MATRIX ELEMENT:
CPU BOTTLENECK
IN OLD MADEVENT

First we developed
the new ME engines
in standalone applications

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

1. STANDALONE
(TOY APPLICATIONS)
MULTI-EVENT API

| CUDA/C++ or PFs: cuRAND |
| CUDA/C++ or PFs: RAMBO |
| MOMENTA |
| CUDA/C++ or PFs: MEKERNELS |
| MATRIX ELEMENTS |

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

| FORTRAN: RANMAR |
| FORTRAN: MADEVENT |
| MOMENTA |
| FORTRAN: MATRIX1 |
| MATRIX ELEMENTS |

| FORTRAN: RANMAR |
| FORTRAN: MADEVENT |
| MOMENTA |
| CUDA/C++ or PFs: MEKERNELS |
| MATRIX ELEMENTS |

(Amdahl...)

SCALAR:
NEW
BOTTLENECK?

PARALLEL:
MUCH FASTER!

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

$gg \to t\bar{t}gg$	MEs precision	ACAT2022 $t_{\text{TOT}} = t_{\text{Mad}} + t_{\text{MEs}}$ [sec]	madevent $N_{\text{events}}/t_{\text{TOT}}$ [events/sec]	$N_{\text{events}}/t_{\text{MEs}}$ [MEs/sec]	standalone
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)	—
C++/none(scalar)	double	37.8 = 1.7 + 36.0	2.17E3 (x1.0)	2.28E3 (x1.0)	2.37E3
C++/sse4(128-bit)	double	19.4 = 1.7 + 17.8	4.22E3 (x1.9)	4.62E3 (x2.0)	4.75E3
C++/avx2(256-bit)	double	9.5 = 1.7 + 7.8	8.63E3 (x3.9)	1.05E4 (x4.6)	1.09E4
C++/512y(256-bit)	double	8.9 = 1.8 + 7.1	9.29E3 (x4.2)	1.16E4 (x5.0)	1.20E4
C++/512z(512-bit)	double	6.1 = 1.8 + 4.3	1.35E4 (x6.1)	1.91E4 (x8.3)	2.06E4
C++/none(scalar)	float	36.6 = 1.8 + 34.9	2.24E3 (x1.0)	2.35E3 (x1.0)	2.45E3
C++/sse4(128-bit)	float	10.6 = 1.7 + 8.9	7.76E3 (x3.6)	9.28E3 (x4.1)	9.21E3
C++/avx2(256-bit)	float	5.7 = 1.8 + 3.9	1.44E4 (x6.6)	2.09E4 (x9.1)	2.13E4
C++/512y(256-bit)	float	5.3 = 1.8 + 3.6	1.54E4 (x7.0)	2.30E4 (x10.0)	2.43E4
C++/512z(512-bit)	float	3.9 = 1.8 + 2.1	2.10E4 (x9.6)	3.92E4 (x17.1)	3.77E4

512y = AVX512, ymm registers
512z = AVX512, zmm registers

The latter is only better on nodes with 2 FMA units (here an Intel Gold 6148)

Scalar
SSE4
AVX2
AVX512

ME speedup ~ x8 (double) and x16 (float) over scalar Fortran
Our ME engine reaches the maximum theoretical SIMD speedup!
Overall speedup so far~ x6 (double) and x10 (float) over scalar Fortran (Amdahl's law)
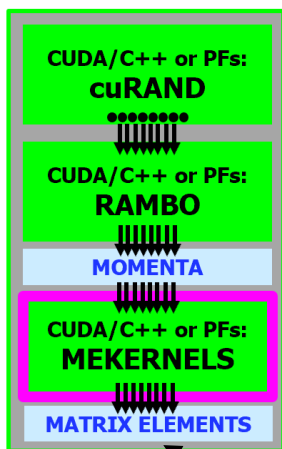
MadEvent/CUDA for gg→t̄tggg

		madevent			standalone	
CUDA grid size	**ACAT2022**		8192		16384	
$gg \to t\bar{t}ggg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]		$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]	
Fortran	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)	—	—
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)

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

Amdahl's law limits the overall speedup to x200 (parallelizable p=0.5%), and we achieve x60 (double) or x100 (float) in the overall speedup!
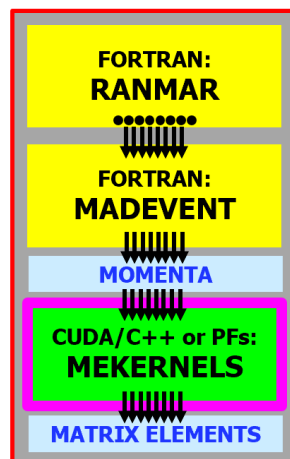
FOR USERS:
NEWS SINCE FEBRUARY 2023
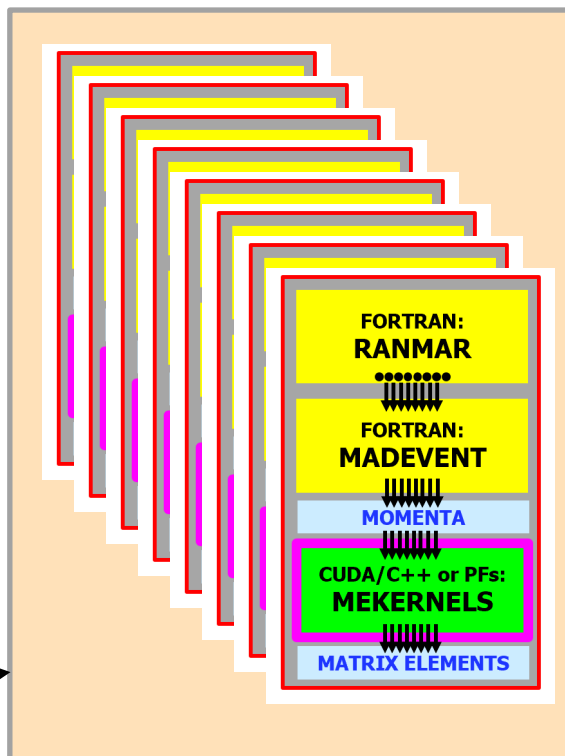
MG5AMC+cudacpp: CUDA/C++, Fortran, bash, python...

**4. COMPLETE WORKFLOW
INCLUDING CODE GENERATION
generate.. output.. launch
SOON! (2023)**

**3. MADEVENT
(N x APPLICATIONS)
./bin/generate_events
BEING TESTED (2023)**

**1. STANDALONE
TOY APPLICATION
OK! (2020-2021)**

**2. MADEVENT
(ONE APPLICATION)
OK! (2022)**

**CUDA/C++ or PFs:
cuRAND**

**CUDA/C++ or PFs:
RAMBO**

MOMENTA

**CUDA/C++ or PFs:
MEKERNELS**

MATRIX ELEMENTS

**FORTRAN:
RANMAR**

**FORTRAN:
MADEVENT**

MOMENTA

**CUDA/C++ or PFs:
MEKERNELS**

MATRIX ELEMENTS

**FORTRAN:
RANMAR**

**FORTRAN:
MADEVENT**

MOMENTA

**CUDA/C++ or PFs:
MEKERNELS**

MATRIX ELEMENTS

**FORTRAN:
RANMAR**

**FORTRAN:
MADEVENT**

MOMENTA

**CUDA/C++ or PFs:
MEKERNELS**

MATRIX ELEMENTS

**MG5AMCNLO GITHUB
+
MADGRAPH4GPU GITHUB**

MG5AMCNLO GITHUB

What exists now and is being tested (and what you could test if you are brave!)

- Code generation needs both the MG5AMC and our madgraph4gpu (cudacpp) github repositories
  - Code generation is driven by a madgraph4gpu script (...anyone can use it, at their own risk...)

- I have launched the first systematic tests of N x madevent application orchestration this week!...
  - I am still waiting for some results: maybe an update next week here?

- There are clearly some errors and crashes – some debugging and tuning is needed
  - The poor madevent workflow has just received a x 16k increase in the number of events to handle...
  - Tuning the SIMD C++ version (x 32 is enough) will be easier than tuning the CUDA version

What else still needs to be fixed, amongst other things

- There are very many issues specific to some physics processes (some NEW from this week!)
  - NEW issues for gg to uu (numerical instabilities? we routinely test gg to tt, massive final state)
  - NEW issues for pp to Wtt (floating point exceptions – process proposed by Francesco here in LH)
  - Parameter handling for BSM like EFT or SUSY (propagating alphas dependency needs new code...)

- Code generation needs to be fully integrated into the MG5AMCNLO repo

- Things evolve every day (some bugs appear, but many bugs are fixed) : STAY TUNED

FOR MC DEVELOPERS:
A FEW MORE INTERNAL DETAILS

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

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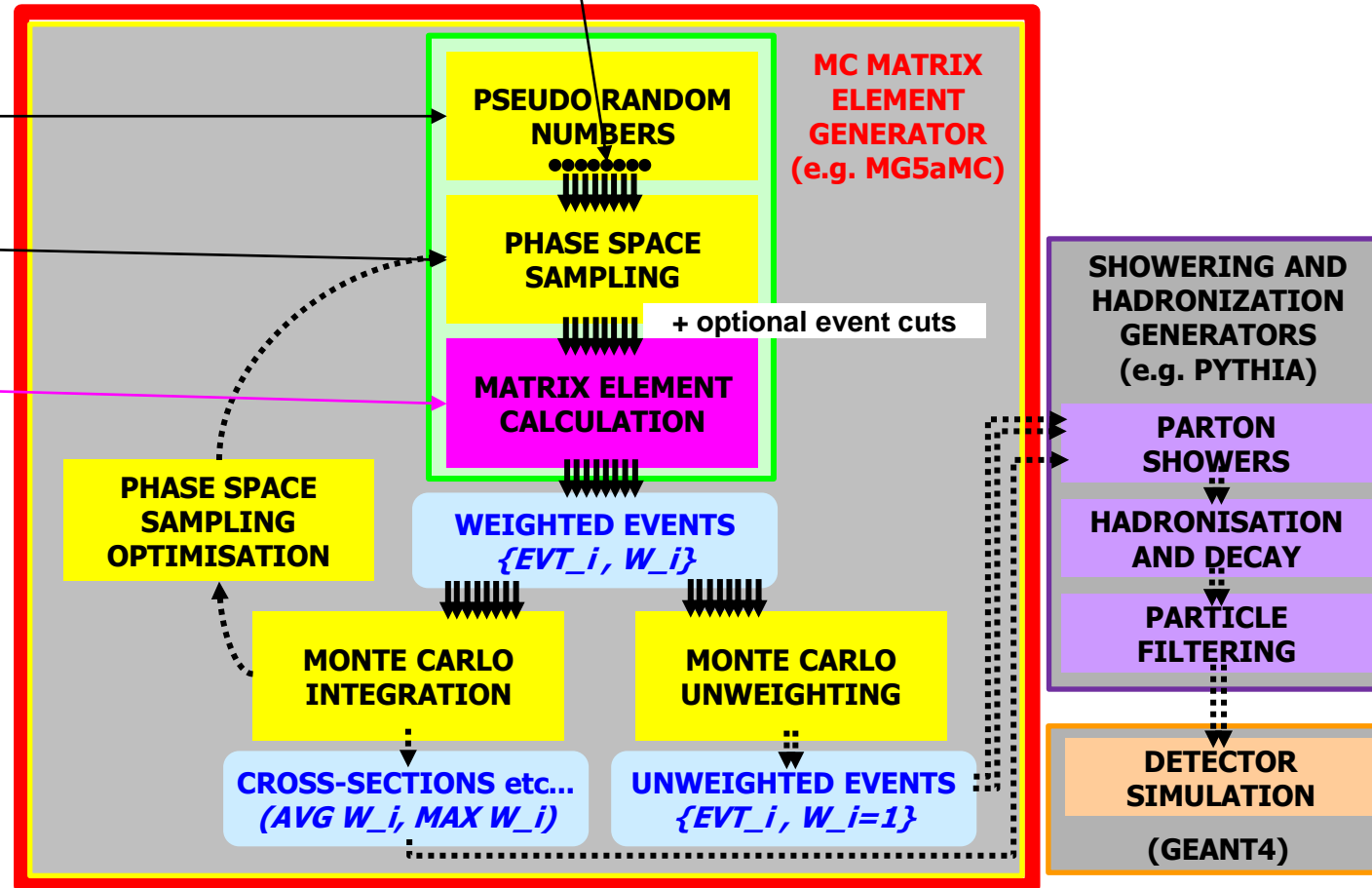MANY different events ("phase space points")*

This can be parallelized (SIMT/SIMD and multithreading)

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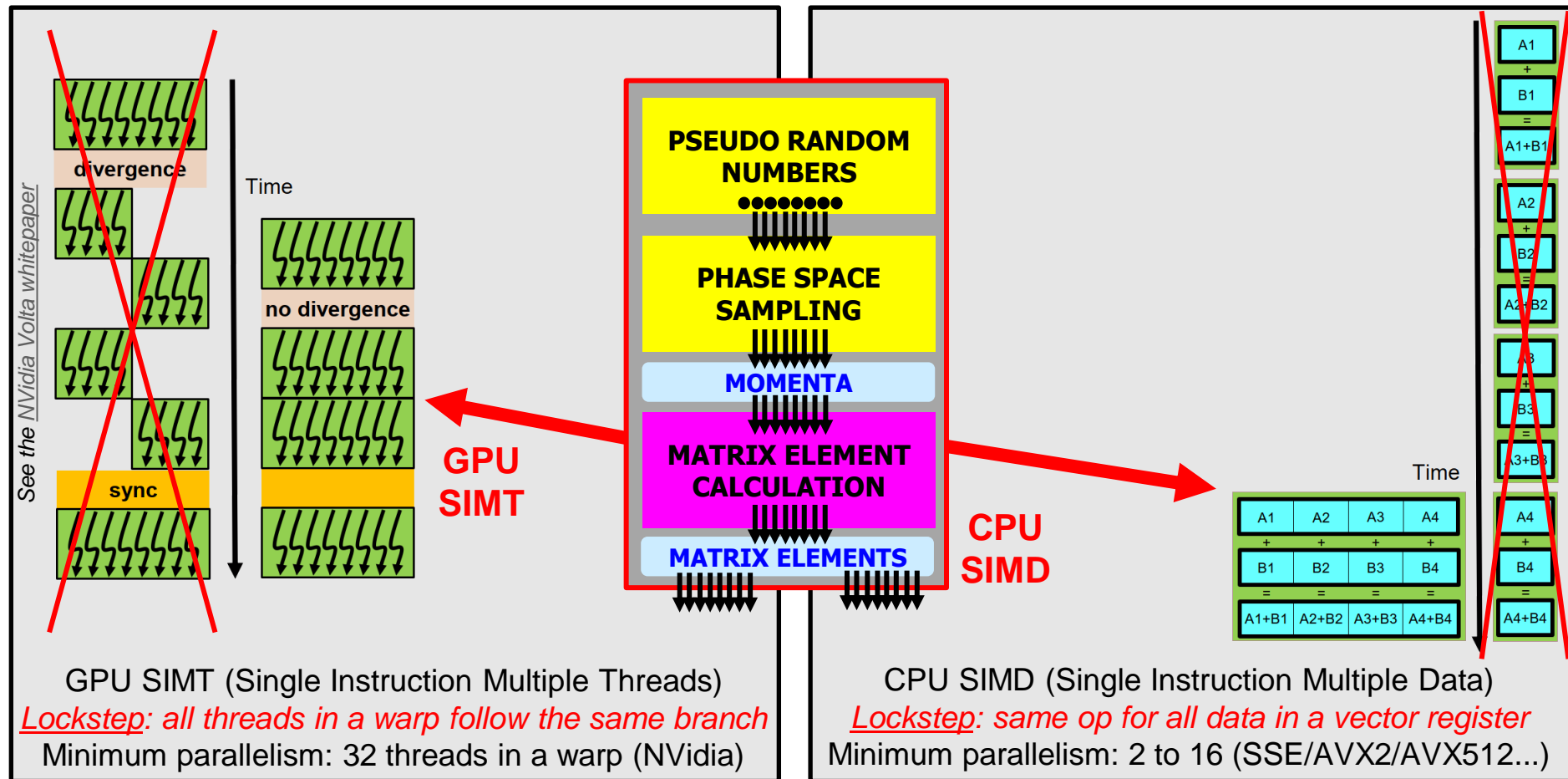
*(NB: "Matrix Element" is an
element of the **scattering matrix**...
not a linear algebra concept!)*

MC MATRIX ELEMENT GENERATOR (e.g. MG5aMC)

PSEUDO RANDOM NUMBERS

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}

SHOWERING AND HADRONIZATION GENERATORS (e.g. PYTHIA)

PARTON SHOWERS

HADRONISATION AND DECAY

PARTICLE FILTERING

DETECTOR SIMULATION

(GEANT4)

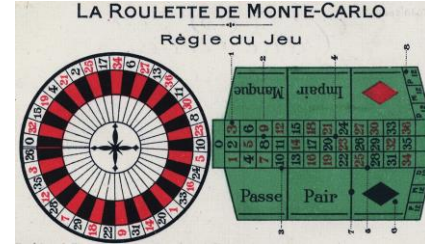**Physics output: cross-section and LHE event file**

MG5aMC data parallelism: design for lockstep processing!

- In MC generators, *the same function is used to compute the Matrix Element for many different events*
  - *ANY matrix element generator is a good fit for lockstep processing on GPUs (SIMT) and vector CPUs (SIMD)*
  - Data parallelism strategy in madgraph4gpu is event-level parallelism (many events = many phase space points)

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

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

Lockstep? MC generators (*lucky!*) vs MC detector simulation (unlucky)

- *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*:

Data parallelism (NB: MULTI-EVENT API !)

MC SAMPLING

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

INPUT

SAME CALCULATION ON DIFFERENT DATA!

OUTPUT

👍 *Lockstep processing Good for SIMT/SIMD*

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

MC DECISIONS 👎

INPUT

DECISION

OUTPUT

Stochastic branching Bad for SIMT/SIMD

<u>**Detector simulation (Geant4)**</u>
- Particle/matter interaction (when? how?)
- Particle decays (when?)

DIFFERENT CALCULATIONS ON DIFFERENT DATA!

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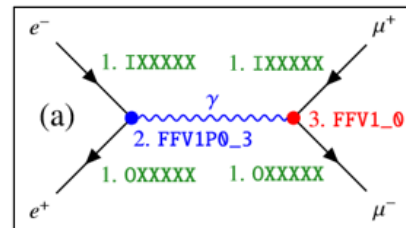 Parton showers (PS)
- Fragmentation and decays

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

Formally the same code for three back-ends *(cxtype_sv represents three types)*
- *CUDA:*   scalar complex → `typedef thrust::complex<fptype> cxtype; // two doubles: RI`
- *C++, no SIMD:*   scalar complex → `typedef std::complex<fptype> cxtype; // two doubles: RI`
- *C++, with SIMD:*   vector complex → `class cxtype_v { fptype_v m_real, m_imag; // RRRRIIII (SOA)`

```
__device__
void FFV1_0( const cxtype_sv F1[],   // input: wavefunction1[6]
             const cxtype_sv F2[],   // input: wavefunction2[6]
             const cxtype_sv V3[],   // input: wavefunction3[6]
             const cxtype COUP,
             cxtype_sv* vertex )      // output: amplitude
{
  mgDebug( 0, __FUNCTION__ );
  const cxtype cI( 0., 1. );
  const cxtype_sv TMP0 = (F1[2] * (F2[4] * (V3[2] + V3[5]) + F2[5] * (V3[3] + cI * (V3[4]))) +
                          (F1[3] * (F2[4] * (V3[3] - cI * (V3[4])) + F2[5] * (V3[2] - V3[5])) +
                          (F1[4] * (F2[2] * (V3[2] - V3[5]) - F2[3] * (V3[3] + cI * (V3[4]))) +
                            F1[5] * (F2[2] * (-V3[3] + cI * (V3[4])) + F2[3] * (V3[2] + V3[5]))))));
  (*vertex) = COUP * - cI * TMP0;
  mgDebug( 1, __FUNCTION__ );
  return;
}
```

FFV1_0:
helicity amplitude
for the γμ⁺μ⁻ vertex

Automatically generated!

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

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

C++ SIMD: gcc / clang
compiler vector extensions

```
#ifdef __clang__
  typedef fptype fptype_v __attribute__ ((ext_vector_type(neppV)));   // RRRR
#else
  typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype))));   // RRRR
#endif
```

- Old slide! The new code is different, the idea is the same!

- *Formally the same code for CUDA and scalar/vector C++*
  - hide type behind a typedef
  - add a few missing operators

SIMD in CUDA/C++ uses compiler vector extensions!

Flexible design: being reused also for vectorized SYCL!

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

Our internal Fortran-to-C++ interface: multi-event and stateless!

```fortran
C
C Execute the matrix-element calculation "sequence" via a Bridge on GPU/CUDA or CUDA/C++.
C - PBRIDGE: the memory address of the C++ Bridge
C - MOMENTA: the input 4-momenta Fortran array
C - GS:      the input Gs (running QCD coupling constant alphas) Fortran array
C - RNDHEL:  the input random number Fortran array for helicity selection
C - RNDCOL:  the input random number Fortran array for color selection
C - CHANID:  the input Feynman diagram to enhance in multi-channel mode if 1 to n (disable multi-channel if 0)
C - MES:     the output matrix element Fortran array
C - SELHEL:  the output selected helicity Fortran array
C - SELCOL:  the output selected color Fortran array
C
      INTERFACE
         SUBROUTINE FBRIDGESEQUENCE(PBRIDGE, MOMENTA, GS,
     &      RNDHEL, RNDCOL, CHANID, MES, SELHEL, SELCOL)
         INTEGER*8 PBRIDGE
         DOUBLE PRECISION MOMENTA(*)
         DOUBLE PRECISION GS(*)
         DOUBLE PRECISION RNDHEL(*)
         DOUBLE PRECISION RNDCOL(*)
         INTEGER*4 CHANID
         DOUBLE PRECISION MES(*)
         INTEGER*4 SELHEL(*)
         INTEGER*4 SELCOL(*)
         END SUBROUTINE FBRIDGESEQUENCE
      END INTERFACE
```
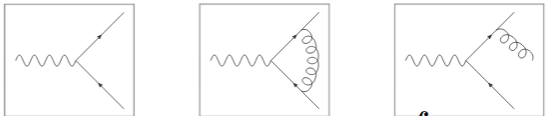
NLO, loops

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

- So far we have only worked on LO QCD processes!

- NLO QCD processes are more computationally intensive
  - They have more Feynman diagrams
  - But especially they have loop diagrams!
  - And, a matching procedure (MC@NLO) must be applied

- *We should be able to compute Born and Real emission contributions in our vectorized C++ and CUDA*
  - We should also be able to handle NLO matching using the current MadEvent based infrastructure
  - The main challenge will be understanding the computational impact of loops (Amdahl)?

MC@NLO: https://doi.org/10.1088/1126-6708/2002/06/029
Matching NLO QCD and parton showers (avoid double counting)

Marco Zaro – https://cp3.irmp.ucl.ac.be/projects/madgraph/wiki/Pavia2015

B, V, R: matrix elements
MC: parton shower

$$d\sigma^n_{NLO} = d\sigma^n_{LO} + d\sigma^n_V + \int d\Phi_1 \, d\sigma^{n+1}_R$$

$$\frac{d\sigma_{``MC@NLO"}}{dO} = \left[\int d\Phi_n (B + V + \int d\Phi_1 MC)\right] I^n_{MC}(O) + \left[\int d\Phi_{n+1}(R - MC)\right] I^{n+1}_{MC}(O)$$

S-events   H-events

S and **H** events: two separate sets of events (different matrix elements)
Integral = S+H is positive – but individual events can have negative weights

... and finally...

- Upcoming: **Workshop on software acceleration of MC event generators**
  - Where? at CERN
  - When? in ~October-November 2023 (any time constraints with important MC events?)
  - Organised together with LPCC, MCnet, HSF...

- Contact me if you are interested and/or if you have any suggestions

BACKUP SLIDES

Speeding up Madgraph5_aMC@NLO through data parallelism: CPU vectorization and GPUs

Andrea Valassi (CERN IT)

With contributions from and many thanks to the whole madgraph4gpu development team!

(TRAILER – SPOILER ALERT!)

THE 4-SLIDE VERSION

The 4-slide version! – Executive summary for the impatient

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

- Our reengineering of MG5aMC is close to a first fully functional alpha release for LO QCD processes
  - *The new ME calculation is integrated in MadEvent* – we get the same cross section *and LHE files* as in Fortran!

- On CPUs, in vectorized C++ we *reach the maximum x8/x16 (double/float) SIMD speedup for MEs alone*
  - The speedups achieved for the overall workflow are slightly lower due to *Amdahl's law*, but not much
  - Example: our *current overall speedup is x6/x10 (double/float)* for gg→t̄tgg (on one CPU core)

- On GPUs, using CUDA we *achieve O(100-1000) speedups for MEs alone over one no-SIMD CPU core*
  - The speedups may be much lower due to *Amdahl's law*, but we are improving on that
  - Example: our *current overall speedup is x60/x100 (double/float)* for gg→t̄tggg on an NVidia V100

- Floats are x2 faster than doubles in SIMD and NVidia GPUs – we also added 'mixed' precision modes

- In SYCL we get ~similar performances to CUDA on NVidia and we may run also on AMD or Intel GPUs

- Future challenges include optimizing heterogenous processing on one GPU and multiple CPU cores

MG5aMC: old and new architecture designs
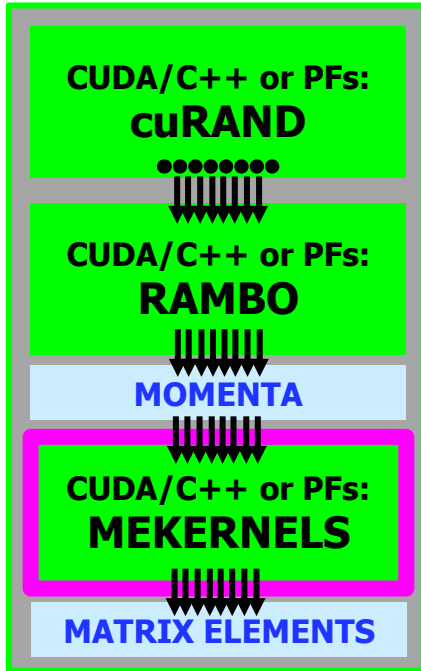
OLD MADEVENT
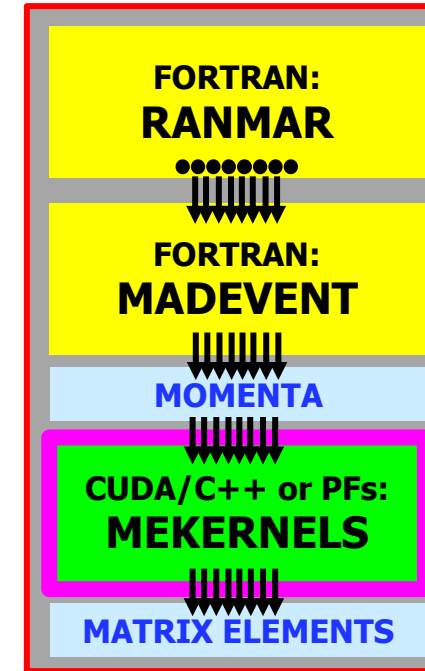*(NOW: LHC PROD)*
SINGLE-EVENT API

First we developed
the new ME engines
in standalone applications

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

FORTRAN:
RANMAR

FORTRAN:
MADEVENT

MOMENTA

FORTRAN:
MATRIX1

MATRIX ELEMENTS

MATRIX ELEMENT:
CPU BOTTLENECK
IN OLD MADEVENT

1. STANDALONE
(TOY APPLICATIONS)
MULTI-EVENT API

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

CUDA/C++ or PFs:
cuRAND

CUDA/C++ or PFs:
RAMBO

MOMENTA

CUDA/C++ or PFs:
MEKERNELS

MATRIX ELEMENTS

FORTRAN:
RANMAR

FORTRAN:
MADEVENT

MOMENTA

FORTRAN:
MATRIX1

MATRIX ELEMENTS

FORTRAN:
RANMAR

FORTRAN:
MADEVENT

MOMENTA

CUDA/C++ or PFs:
MEKERNELS

MATRIX ELEMENTS

(Amdahl...)

SCALAR:
NEW
BOTTLENECK?

PARALLEL:
MUCH FASTER!

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

gg→tt̄gg	MEs precision	ACAT2022 $t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	madevent N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]	standalone
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)	—
C++/none(scalar)	double	37.8 = 1.7 + 36.0	2.17E3 (x1.0)	2.28E3 (x1.0)	2.37E3
C++/sse4(128-bit)	double	19.4 = 1.7 + 17.8	4.22E3 (x1.9)	4.62E3 (x2.0)	4.75E3
C++/avx2(256-bit)	double	9.5 = 1.7 + 7.8	8.63E3 (x3.9)	1.05E4 (x4.6)	1.09E4
C++/512y(256-bit)	double	8.9 = 1.8 + 7.1	9.29E3 (x4.2)	1.16E4 (x5.0)	1.20E4
C++/512z(512-bit)	double	6.1 = 1.8 + 4.3	1.35E4 (x6.1)	1.91E4 (x8.3)	2.06E4
C++/none(scalar)	float	36.6 = 1.8 + 34.9	2.24E3 (x1.0)	2.35E3 (x1.0)	2.45E3
C++/sse4(128-bit)	float	10.6 = 1.7 + 8.9	7.76E3 (x3.6)	9.28E3 (x4.1)	9.21E3
C++/avx2(256-bit)	float	5.7 = 1.8 + 3.9	1.44E4 (x6.6)	2.09E4 (x9.1)	2.13E4
C++/512y(256-bit)	float	5.3 = 1.8 + 3.6	1.54E4 (x7.0)	2.30E4 (x10.0)	2.43E4
C++/512z(512-bit)	float	3.9 = 1.8 + 2.1	2.10E4 (x9.6)	3.92E4 (x17.1)	3.77E4

512y = AVX512, ymm registers
512z = AVX512, zmm registers

The latter is only better on nodes with 2 FMA units (here an Intel Gold 6148)

ME speedup ~ x8 (double) and x16 (float) over scalar Fortran
Our ME engine reaches the maximum theoretical SIMD speedup!
Overall speedup so far~ x6 (double) and x10 (float) over scalar Fortran (Amdahl's law)

MadEvent/CUDA for gg→t̄tggg

		madevent			standalone	
CUDA grid size	ACAT2022	8192			16384	
$gg \to t\bar{t}ggg$	MEs precision	$t_{\mathrm{TOT}} = t_{\mathrm{Mad}} + t_{\mathrm{MEs}}$ [sec]	$N_{\mathrm{events}}/t_{\mathrm{TOT}}$ [events/sec]	$N_{\mathrm{events}}/t_{\mathrm{MEs}}$ [MEs/sec]		
Fortran	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)	—	—
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)

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

Amdahl's law limits the overall speedup to x200 (parallelizable p=0.5%), and we achieve x60 (double) or x100 (float) in the overall speedup!

THE LONG VERSION

(project evolution, physics, software details...)

The (very) long version! – Outline

(1) Inception
Motivation, overview, how it all started
Why speed up MC generators?
Why Madgraph5_aMC@NLO (MG5aMC)?
Why GPUs and vectorization?

(3) Outlook
Towards the upcoming first LO alpha release
WIP, plans, ideas for more speed and features
Multithreading and heterogeneous strategies
More compilers, more CPU and GPU architectures
Beyond LO: loops and NLO
Event-by-event reweighting
Faster smaller kernels, faster builds
Tensor cores, cuBLAS
Other ME generators? Parton showers?

(2) Implementation
Architecture design, technical choices, progress so far
Beyond $e^+e^- \rightarrow \mu^+\mu^-$: from discrete "epochs" to fast-turnaround code generation
Standalone CUDA/C++, memory layout, SIMD, GPU kernels
Beyond standalone toys: full functional integration with Fortran MadEvent
Recent performance improvements: AVX512, Amdahl, mixed precision...
Alternative to CUDA/C++: abstraction layers (Alpaka, Kokkos, Sycl)

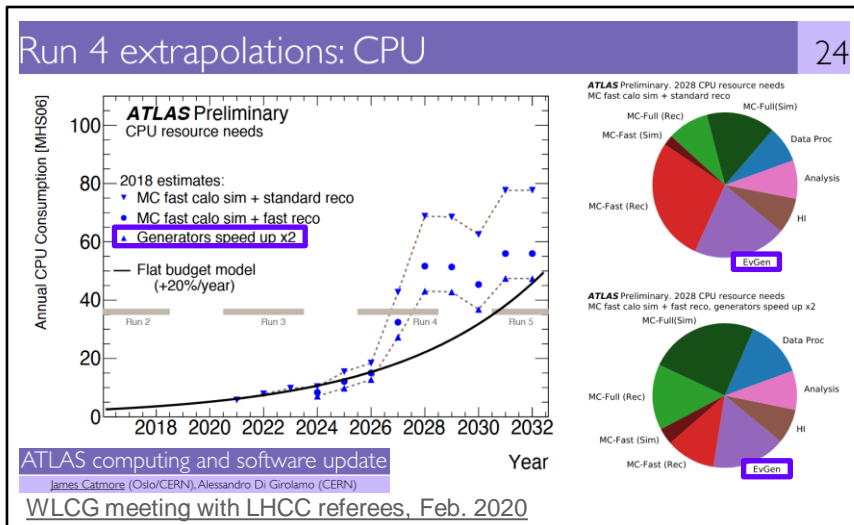# (1) Inception: motivation, overview, how it all started

Apr 2020: HSF Generator WG review paper v1
May 2020: start of LHCC review of HL-LHC Software and Computing

Mar 2019: HSF/OSG/WLCG "HOW" workshop JLAB
review of KEK GPU work on MG5AMC

Sep 2020: HSF generator WG presentation to LHCC

Nov 2018: HSF Generator Workshop

2018 Q4 Q1 Q2 Q3 Q4 **2019** **2020** Q1 Q2 Q3 **2021** **2022** **2023** **2024**
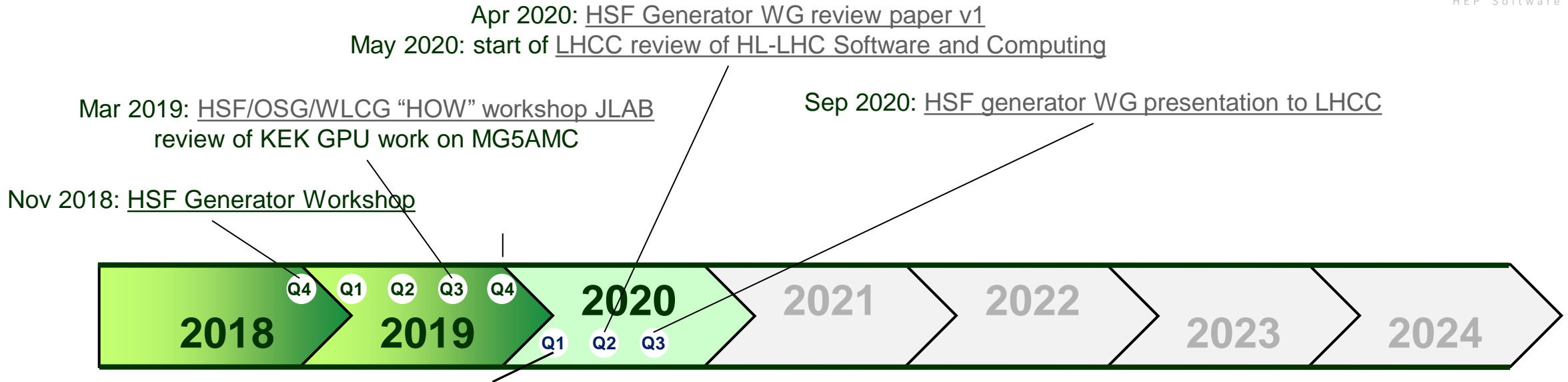
Q1 2020: start of madgraph4gpu project (Stefan Roiser, Olivier Mattelaer, AV)
Feb 2020: Stefan's first commits on gitlab, first standalone cuda/C++ executable

- The work on generators triggered by the HSF and the LHCC review created many opportunities
  - A heightened sensitivity to the need to *modernize and speed up HEP software (including generators)*
  - A breakdown of the *computational anatomy of ME event generators* and of the ways they can be improved
  - A review of previous work on porting Madgraph5_aMC@NLO to GPUs (2008-2013, mainly at KEK)
  - *An opportunity for theorists, experimentalists and software engineers to meet and start collaborating...*

Motivation 1: Monte Carlo Event Generators in WLCG computing

- HL-LHC computing needs are predicted to outpace resource growth: need R&D to improve software

- MC generators are essential for HL-LHC physics and *use ~5-20% of ATLAS/CMS WLCG CPU budgets*
  - Speeding them up helps address the resource gap and may allow more complex (N)NLO multi-jet simulations

This plot is probably obsolete by now!
Sherpa speedups >>2 were reported at ACAT

- Challenges and opportunities to improve MC software have been *discussed in the HSF generator WG*
  - See the WG review paper prepared for the LHCC review in 2020: https://doi.org/10.1007/s41781-021-00055-1

Motivation 2: GPUs and vector CPUs are underexploited in HEP

- *GPUs provide most of the compute power in recent HPCs (e.g. <u>Summit: 95%</u>)*
  - Supercomputers at HPC centers are already heavily used by the LHC experiments on an "opportunistic" basis
  - But *only a small share of HEP software workloads can run on GPUs today*

Summit: NVidia V100 GPUs   Aurora: Intel Xe GPUs   Leonardo: NVidia A100 GPUs   Juwels: NVidia A100 GPUs   LUMI: AMD MI250X GPUs

- *Most WLCG CPUs support wide vector registers (SSE4.2, AVX2 or above)*
  - But *only a small share of HEP software workloads efficiently exploit CPU vectorization today*

- These architectures are best suited to *<u>lockstep processing with limited branching</u>*... as in MC generators!
  - *MC generators are ideal candidates to exploit data parallelism in GPUs (SIMT) and vector CPUs (SIMD)!*
  - In this talk I will describe how we achieved this for Madgraph5_aMC@NLO (via event-level data parallelism)

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

Monte Carlo sampling: randomly generate and process MANY different events ("phase space points")
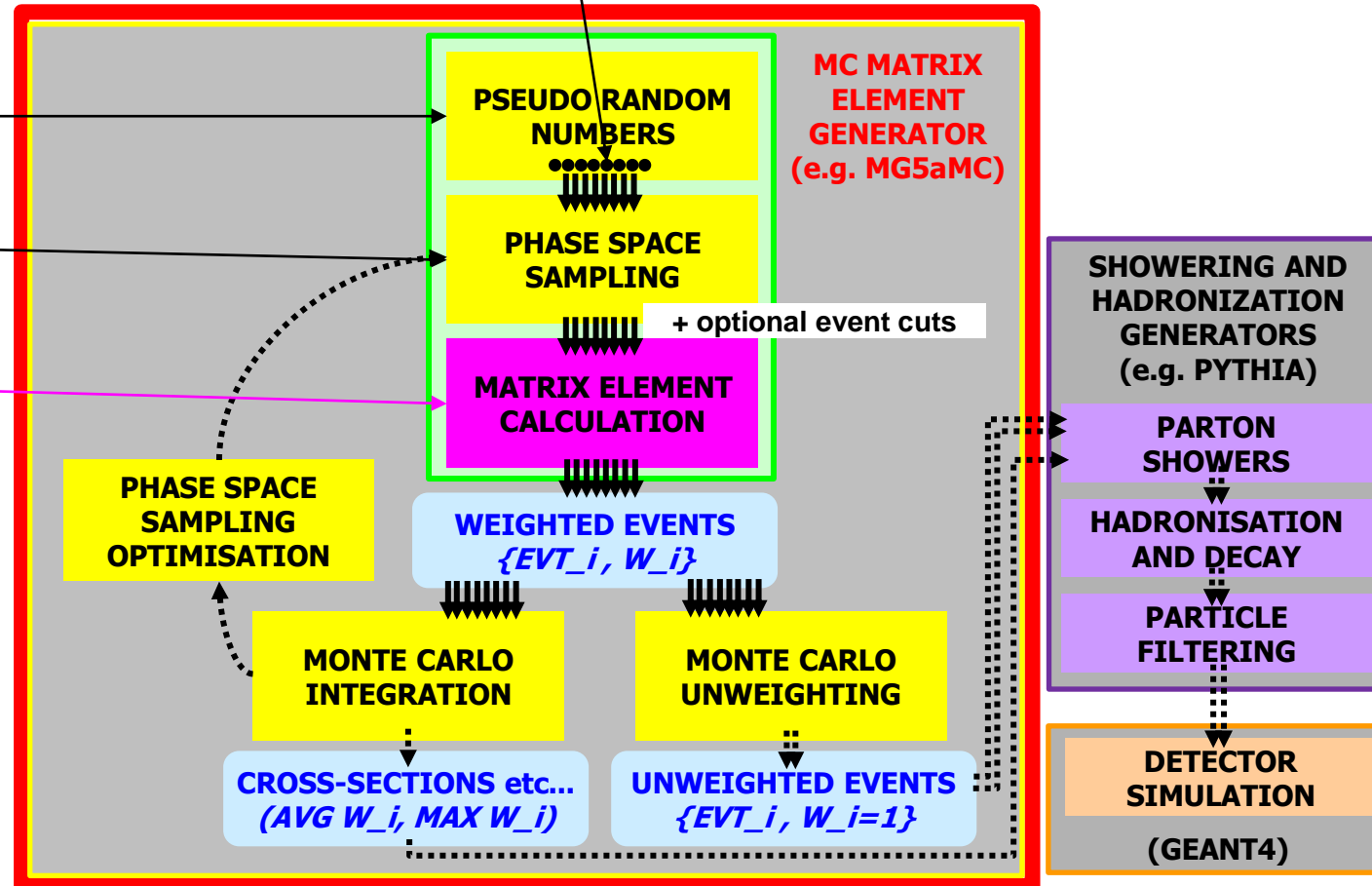
This can be parallelized (SIMT/SIMD and multithreading)

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

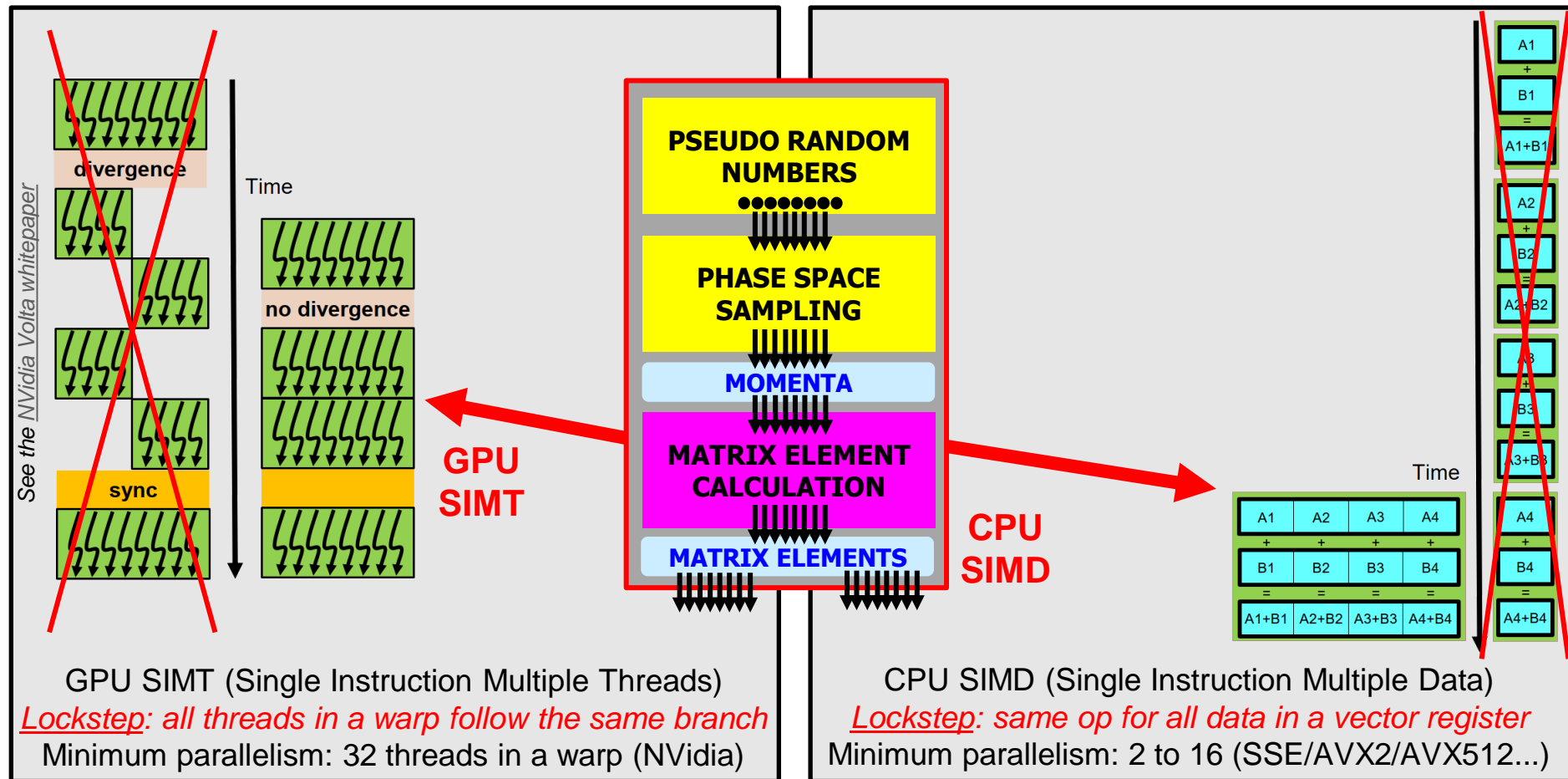(NB: "Matrix Element" is an element of the **scattering matrix**... not a linear algebra concept!)

MC MATRIX ELEMENT GENERATOR (e.g. MG5aMC)

PSEUDO RANDOM NUMBERS

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}

SHOWERING AND HADRONIZATION GENERATORS (e.g. PYTHIA)

PARTON SHOWERS

HADRONISATION AND DECAY

PARTICLE FILTERING

DETECTOR SIMULATION

(GEANT4)

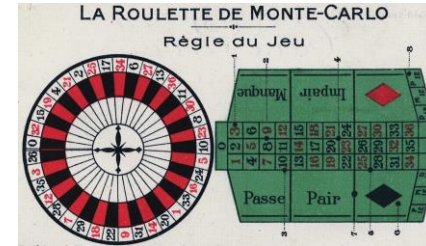**(FOR LATER!) Physics output: cross-section and LHE event file**

MG5aMC data parallelism: design for lockstep processing!

- In MC generators, *the same function is used to compute the Matrix Element for many different events*
  - *__ANY__ matrix element generator is a good fit for lockstep processing on GPUs (SIMT) and vector CPUs (SIMD)*
  - Data parallelism strategy in madgraph4gpu is event-level parallelism (many events = many phase space points)

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

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

Lockstep? MC generators (*lucky!*) vs MC detector simulation (unlucky)

- *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*:

Data parallelism (NB: MULTI-EVENT API !)

MC SAMPLING

INPUT

SAME CALCULATION ON DIFFERENT DATA!

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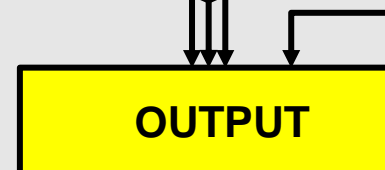 (cross sections)
- MC generation
  (event samples)

OUTPUT

Lockstep processing Good for SIMT/SIMD

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

INPUT

DECISION

OUTPUT

MC DECISIONS

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

DIFFERENT CALCULATIONS ON DIFFERENT DATA!

Stochastic branching Bad for SIMT/SIMD
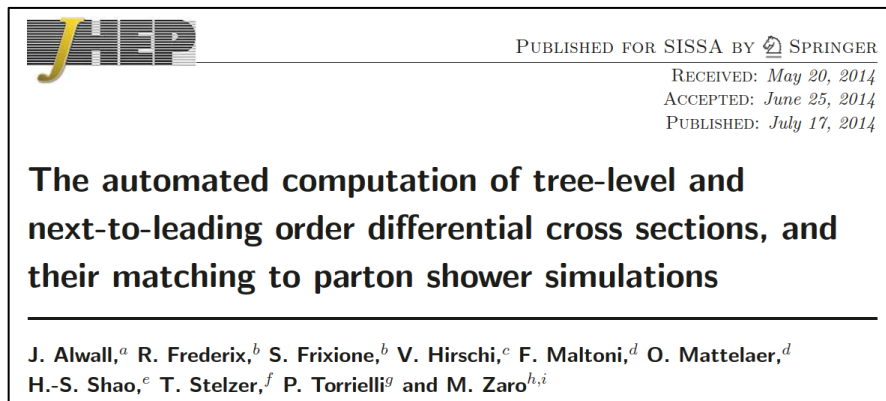
Event generators*
(*after* ME calculation):
- MC unweighting (keep/reject)
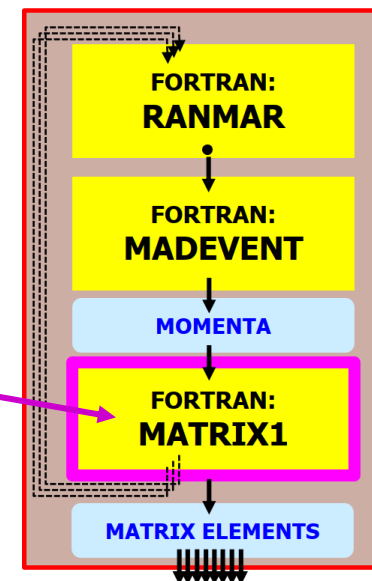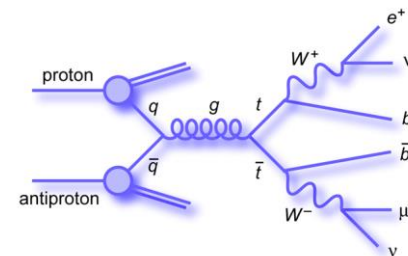 Parton showers (PS)
- Fragmentation and decays

Madgraph5_aMC@NLO (MG5aMC)

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

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

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

- Code repository recently moved from bazaar launchpad to https://github.com/mg5amcnlo/mg5amcnlo

MG5aMC and the madgraph4gpu project

- *madgraph4gpu: speed up Matrix Element calculation in MG5aMC* on GPUs and vector CPUs
  - Code repository, CI tests, issue tracker: https://github.com/madgraph5/madgraph4gpu
  - Development meetings every two weeks:

- Collaboration of theoretical/experimental physicists with software engineers, born in HSF generator WG
  - Initial core team since Q1 2020: Stefan Roiser – project leader, AV (CERN), Olivier Mattelaer (Louvain)
  - Many other collaborators have joined over time and contributed to coding, testing and/or meeting discussions
    - Currently active (alphabetical order): Taylor Childers, Walter Hopkins, Nathan Nichols (Argonne), Stephan Hageboeck, Jorgen Teig, Zenny Wettersten (CERN), Josh McFayden (Sussex), Carl Vuosalo (Wisconsin)
    - Past collaborators: Tyler Burch, Smita Darmora (Argonne), Taran Singhania (Bangalore), Vince Pascuzzi (Berkeley), Laurence Field, Andreas Reepschlaeger, David Smith (CERN)

- Why MG5aMC and not another MC generator for our GPU port?
  - The reason is not that earlier ports of MG5aMC to GPUs exist (HOW talk): we are not leveraging on this work!
    - KEK developments in 2008-2013, based on old version of MG5aMC's ME library were never released for production use
  - The main reason we focused on MG5aMC is the active involvement of Olivier (MG5aMC code developer)!
    - This is also one of the main reason our collaboration is making good progress!
    - *Need a good mix of technical expertise and domain-specific knowledge, either alone would not be enough...*
  - *NB: many of the design ideas we describe are applicable to other generators...*

Guide to the next slides – a few steps/alternatives in development

- Over time, our approach to code generation for different physics processes has evolved
  - Initially, we started with a simple $e^+e^- \to \mu^+\mu^-$ process, with infrequent back-ports to code generation
  - By now, we back-port to code generation immediately and handle ~arbitrarily complex processes like $gg \to t\bar{t}ggg$

- Two parallel approaches to reimplement the ME calculation
  - (1) "CUDACPP", our initial single-code CUDA/C++ back-end targeting <u>NVidia GPUs</u> and <u>SIMD on vector CPUs</u>
  - (2) Portability Frameworks (PFs: Alpaka, Kokkos, SYCL), later addition supporting <u>many GPUs</u> (and CPUs too)

- Two types of executables over time (in each of cudacpp and PFs) – both are still maintained in parallel
  - (a) standalone applications, our initial prototype – we still use this to optimize the ME calculation alone
  - (b) MadEvent-integrated applications, our final goal – usable by LHC experiments, same interface with faster ME!

- *Disclaimer! I will ~only describe the CUDA/C++ implementation as it's the one I work on and know best*
  - New features (e.g. MadEvent integration) have been added to CUDA/C++ first and then ported to SYCL/Kokkos

(2) Implementation: design, technical details, progress so far

PROCEEDINGS OF SCIENCE

PoS(ICHEP2022)212

Developments in Performance and Portability for MadGraph5_aMC@NLO

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

https://doi.org/10.22323/1.414.0212

Jul 2022 ICHEP:
CUDA/C++ **madevent** gg→t$\bar{\text{t}}$gg
Kokkos/SYCL standalone gg→t$\bar{\text{t}}$gg

Oct 2022 ACAT: CUDA/C++ madevent gg→t$\bar{\text{t}}$gg
performance studies (Amdahl, MT, heterogeneous)

Oct 2021: **full code generation**
CUDA/C++ standalone gg→t$\bar{\text{t}}$gg

NEW!
Jan 2023: CUDA/C++ madevent gg→t$\bar{\text{t}}$gg
functionally complete LHE files

2018 2019 **2020** Q1 Q2 Q3 Q4 **2021** Q1 Q2 Q3 Q4 **2022** Q1 Q2 Q3 Q4 Q1 **2023** 2024
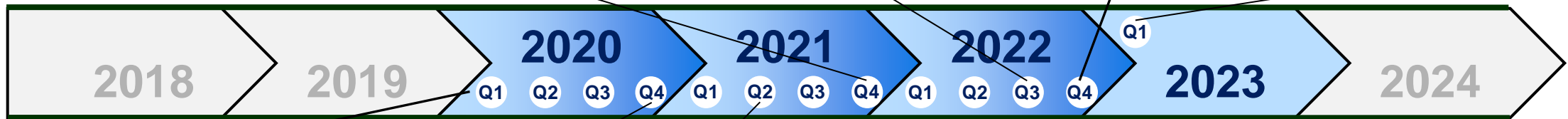
Feb 2020: project starts
CUDA and (scalar) C++
standalone e⁺e⁻→μ⁺μ⁻

Jun 2021 vCHEP:
CUDA/C++ standalone e⁺e⁻→μ⁺μ⁻

Dec 2020: **C++ vectorization**

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

- *Disclaimer - again! Above and in the rest of this talk I focus on the CUDA/C++ implementation*

CUDA/C++: a single source code approach (so far...)

- The main difference between our CPU (C++) and GPU (CUDA) implementations is the following
  - *on the CPU*, all computations and all memory access takes place *on the host*
  - *on the GPU*, it is necessary to distinguish computations and memory accesses *on the host and on the device*

- Within the GPU code, the amount of code that is specific to NVidia/CUDA is minimal
  - Memory allocations (cudaMalloc), encapsulated within host/device buffer classes
  - Kernel executions (<<<...>>>), encapsulated within very few specific classes
  - A few specific types or features (thrust::complex, curand, cuBLAS), also encapsulated in specific classes

- *The rest of the code is (at least formally – see example later) ~identical for C++ and CUDA!*
  - There are almost more differences between scalar and vector C++ code...

- Therefore, *we presently use a single source code approach for CUDA/C++ (with #ifdef __CUDA__)*
  - We might review this later on – as it sometimes imposes slightly unnatural choices, and may hinder readability
  - But so far it has allowed us to make rapid progress for both CUDA and C++ in parallel!

Memory layouts – AOS, SOA, AOSOA

MOMENTA

CUDA/C++:
MEKERNELS
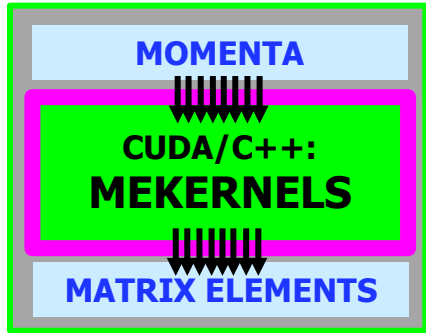
MATRIX ELEMENTS

Matrix element calculation (simplified example)
– *inputs[4*Npar*Nevt]* = (x,y,z,E)-momentum of Npar particles for Nevt events (n-dim array, substructure)
– *outputs[Nevt]* = matrix element for Nevt events (1-dim array, no substructure)

Example: Npar=6 particles for the 2→4 process gg→t̄tgg

We have experimented with three possible memory layouts for momenta
(1) Array-of-Structures **AOS**: momenta[Nevt][Npar][4]
(2) Structure-of-Arrays **SOA**: momenta[Npar][4][Nevt]
(3) **AOSOA: momenta[Npag][Npar][4][Nepp]** with Nevt = Npag ("pages") * Nepp ("events per page")
We are using AOSOA's as the current default – but this is still largely configurable

- *For CPU vectorization, AOSOAs (or SOAs) are absolutely mandatory!*
  - We use an AOSOA with Nepp equal to the SIMD vector size NeppV – and an aligned *malloc* is needed too!
  - For performance comparison we also build a no-SIMD mode with Nepp=1, which is effectively an AOS

- *For GPUs (1 event per thread), AOSOAs are faster (fewer memory accesses) but not strictly necessary*
  - We use Nepp=4(8) for doubles(floats) so that each page is 32 bytes (the "sector" size, or L2 cache line size)
  - For a given number of "requests", *AOS uses 4 times more "sectors" (transactions) than AOSOA* with Nepp=4

- Coding for SIMD is more complex than coding for GPUs...

Monitoring GPU memory access – NSight Compute

- Explicitly collect two relevant profiler metrics in NSight Compute
  - "requests" : *l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum*
  - "sectors" (i.e. transactions, network roundtrips): *l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum*
  - this is from old tests in August 2020 (issue #16), the profiler metrics names may have changed since then
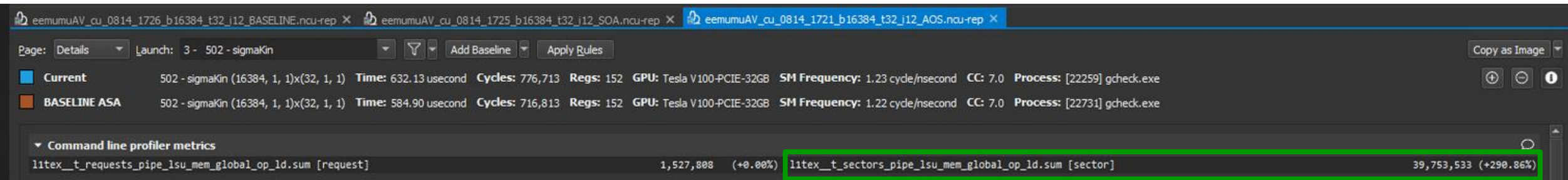
- Profile AOS against the AOSOA baseline
  - same number of "requests" in AOS and AOSOA
  - AOS needs 4 times as many "sectors" as AOSOA (which fits 4 doubles in a 32-byte cache line)
  - in other words: *AOSOA provides coalesced memory access, AOS does not*
  - for what it is worth (not much!), the actual slowdown in this $e^+e^- \rightarrow \mu^+\mu^-$ example was only 7% however

Inside the ME calculation: Feynman diagrams, colors, helicities

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{\text{hel}\}} \left[\sum_{c \in \{\text{col}\}} \left| \sum_{d \in \{\text{diag}\}} (\mathcal{M}_\lambda^d(\vec{p}))^{(c)} \right|^2 \right]$$

Given the momenta $\vec{p}$ of initial+final partons ==in one specific event==
==Sum over all helicity combinations λ== of initial+final partons
==Sum over all color combinations c== of initial+final partons
==Include all Feynman diagrams d== allowed for the given λ and c

In practice in MG5aMC: use **helicity amplitudes and QCD color decomposition**

1. (for each helicity λ) compute partial amplitudes J^f for each color ordering permutation f (sum diagrams relevant to f)

$$(J_\lambda(\vec{p}))^f = \sum_{d \in \{\text{diag}\}} (\mathcal{M}_\lambda^d(\vec{p}))^f$$

Example for $gg \to t\bar{t}ggg$: 1240 Feynman diagrams (using helicity amplitudes)
This takes **~40% of the CPU time** for this process

2. (for each helicity λ) compute the sum over colors as the quadratic form JCJ* using the constant color matrix C

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{\text{hel}\}} \left[\sum_{f,g} (J_\lambda(\vec{p}))^f (C)^{fg} (J_\lambda^*(\vec{p}))^g \right]$$

Example for $gg \to t\bar{t}ggg$: 120 color ordering permutations, 120x120 matrix
This takes **~60% of the CPU time** for this process

3. sum over helicities [Example for $gg \to t\bar{t}ggg$: 128 helicities (before and after filtering)]

Each step computes many events $\vec{p}$ in parallel! CPU: 1 SIMD event-vector at a time. GPU: 1 event per thread.
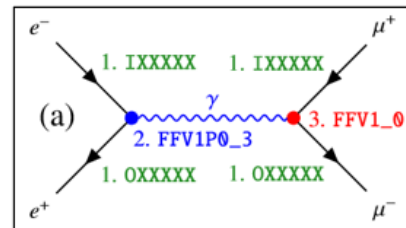
Helicity amplitudes – same code in CUDA and in vectorized C++

Formally the same code for three back-ends (cxtype_sv represents three types)
- *CUDA*:   scalar complex → `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

Automatically generated!

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

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

C++ SIMD: gcc / clang
compiler vector extensions

```
#ifdef __clang__
  typedef fptype fptype_v __attribute__ ((ext_vector_type(neppV))); // RRRR
#else
  typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype)))); // RRRR
#endif
```

- Old slide! The new code is different, the idea is the same!

- *Formally the same code for CUDA and scalar/vector C++*
  - hide type behind a typedef
  - add a few missing operators

SIMD in CUDA/C++ uses compiler vector extensions!

Flexible design: being reused also for vectorized SYCL!

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

# C++ vectorization – why choose Compiler Vector Extensions?

```
typedef fptype fptype_v __attribute__ ((vector_size (neppV*sizeof(fptype))));
```

- Portable – available in gcc, clang, icpx (from clang) with minimal differences
  - *Do not require any external libraries* or tools (VC, VCL, VecCore, xSIMD, UME::SIMD, or SYCL...)

- Powerful, but easy to use
  - *No need to debug auto-vectorization* when it does not vectorize
  - *As powerful as intrinsics, but much easier to write* (higher-level abstractions)

- Intuitive – *CVEs force you to think in terms of vector types!*

- Minor disadvantage – no vector complex type out of the box
  - But it was easy to write it in our case (RRRRIIII memory layout) as we only need + - × ÷
  - A few extensions for Boolean vector masks were needed, too

- One technical detail: we malloc a standard (aligned!) fptype* and reinterpret_cast as fptype_v*...

HUGE THANKS TO SEBASTIEN PONCE *for his Practical Vectorization lectures mentioning CVEs!*

Monitoring lockstep – GPU NSight compute, CPU disassemble

- GPU: explicitly collect one profiler metric in NSight Compute
  - "branch efficiency" : *sm__sass_average_branch_targets_threads_uniform.pct*
  - old test (May 2021 [issue #25](#)) comparing two code bases: *no-divergence baseline has 100% efficiency,* alternative with minor forced divergence has 96% efficiency (and is 20% slower)
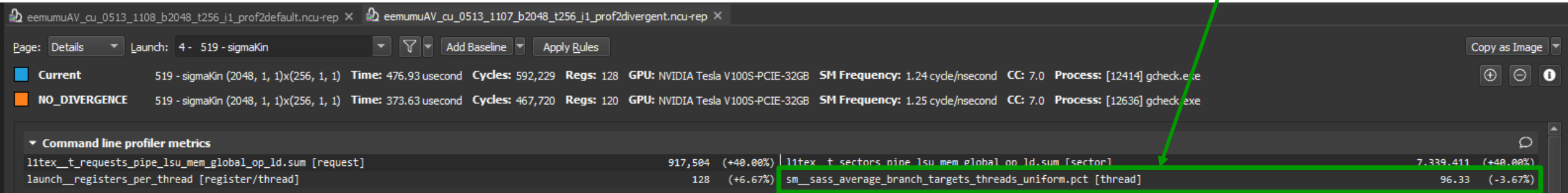

- CPU: the best lockstep metric IMO is the speedup over a no-SIMD case (reach theoretical maximum!)
  - but is also useful to disassemble the object using objdump and categorize SIMD intrinsics symbols...

| # Symbols in .o	SSE4.2 (xmm)	AVX2 (ymm)	AVX512 (ymm)	AVX512 (zmm)
Build type				
Scalar	4534	0	0	0
SSE4.2	12916	0	0	0
AVX2	0	10630	0	0
256-bit AVX512	0	10366	12	0
512-bit AVX512	0	1267	60	9910

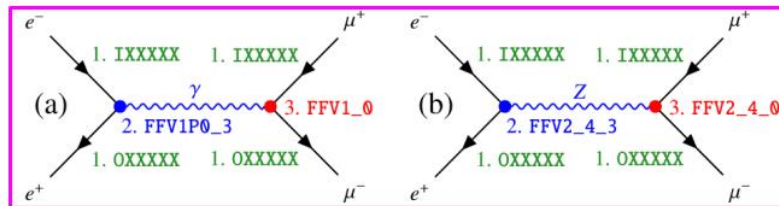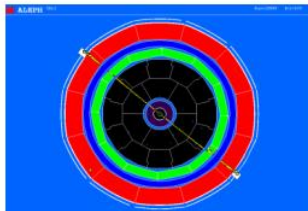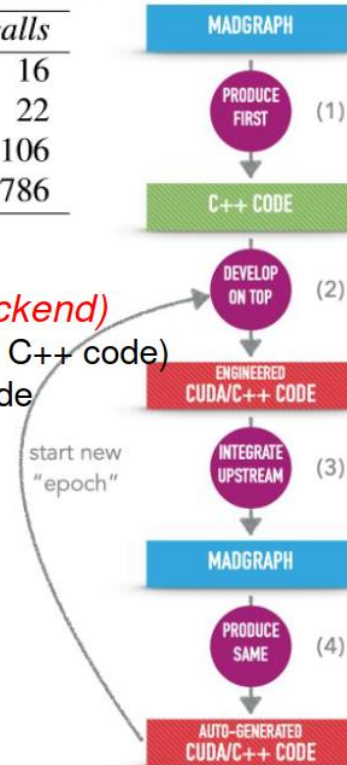| ACAT2022		madevent
$gg \rightarrow t\bar{t}gg$	MEs precision	N_{events}/t_{MEs} [MEs/sec]
Fortran(scalar)	double	2.30E3 (=1.0)
C++/none(scalar)	double	2.28E3 (x1.0)
C++/sse4(128-bit)	double	4.62E3 (x2.0)
C++/avx2(256-bit)	double	1.05E4 (x4.6)
C++/512y(256-bit)	double	1.16E4 (x5.0)
C++/512z(512-bit)	double	1.91E4 (x8.3)
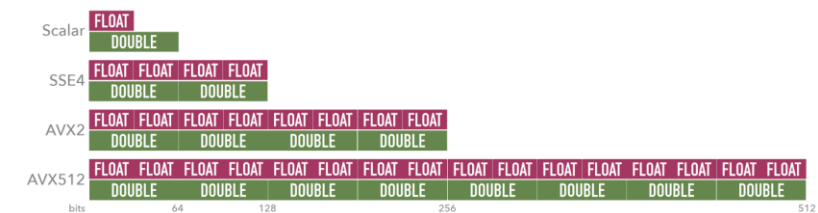
Code generation: how did we bootstrap the project?

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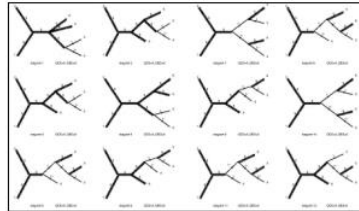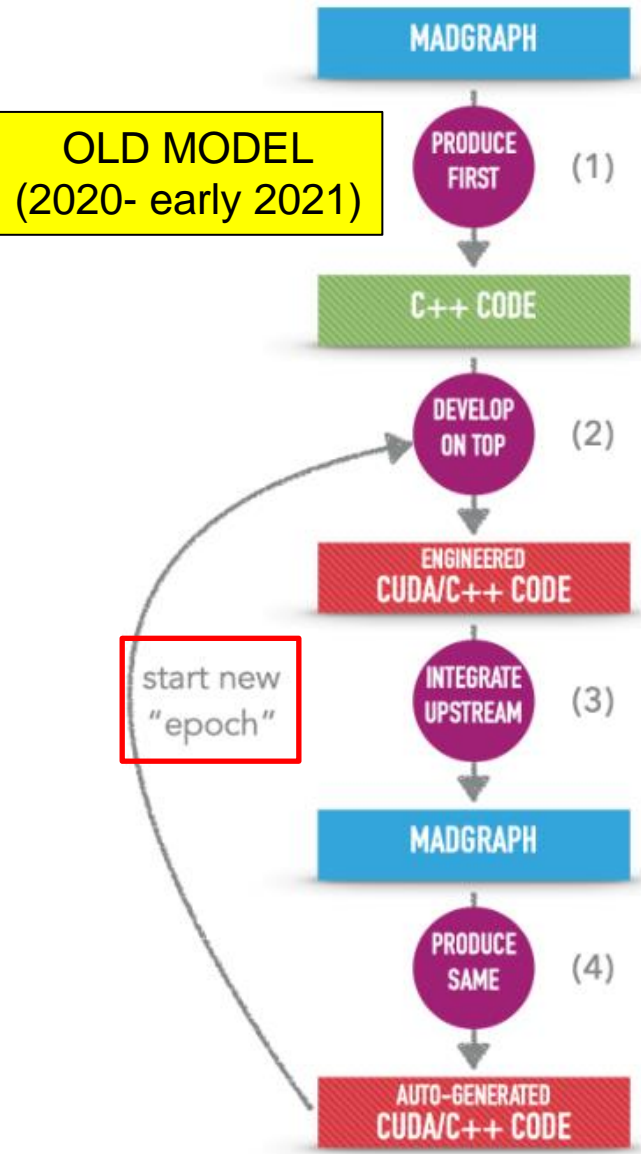
Process	LOC	functions	function calls
$e^+e^- \to \mu^+\mu^-$	776	8	16
$gg \to t\bar{t}$	839	10	22
$gg \to t\bar{t}g$	1082	36	106
$gg \to t\bar{t}gg$	1985	222	786

- *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*

MADGRAPH

PRODUCE FIRST (1)

C++ CODE

DEVELOP ON TOP (2)

ENGINEERED CUDA/C++ CODE

start new "epoch"

INTEGRATE UPSTREAM (3)

MADGRAPH

PRODUCE SAME (4)

AUTO-GENERATED CUDA/C++ CODE

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

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

OLD MODEL
(2020- early 2021)

(1) MG5AMC Python framework, Fortran templates:
"upstream" https://github.com/mg5amcnlo/mg5amcnlo

(2) CUDACPP plugin, post-generation patches,
generated CUDA/C++ physics processes:
our https://github.com/madgraph5/madgraph4gpu

Code generation infrastructure
- Python framework and "cudacpp" plugin
- Fortran, C++, CUDA templates
- *Post-generation patches (temporary...)*
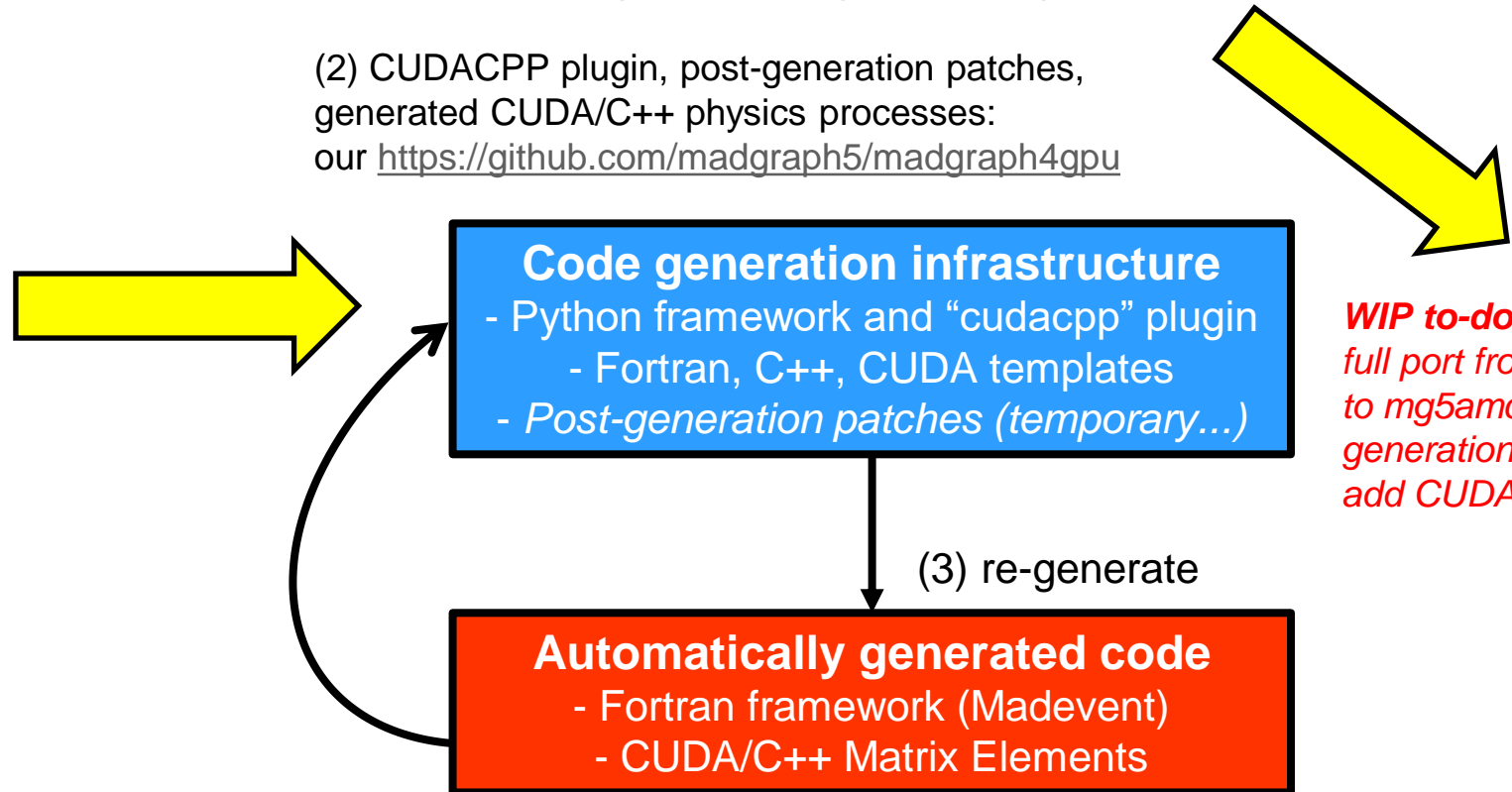
WIP to-do before a release:
full port from madgraph4gpu
to mg5amcnlo (remove post-
generation Fortran patches,
add CUDACPP upstream)

(3) re-generate

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

NEW MODEL
(since end 2021)

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

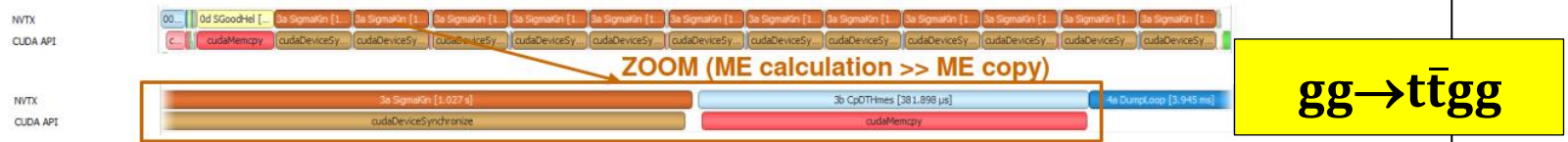# Why focus on complex processes? Compute >> memory!

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. $e^+e^- \rightarrow \mu^+\mu^-$: 0.4ms ME calculation ~ 0.4ms ME copy)
    - Note: our ME throughput numbers are (number of MEs) / (time for ME calculation + ME copy)

ZOOM (ME calculation ~ ME copy)

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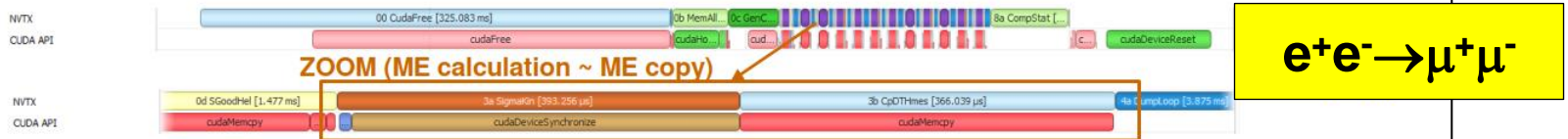
- But the time *cost of data transfers is negligible in complex processes*
  - ME calculation on GPU is slow (e.g. $gg \rightarrow t\bar{t}gg$: 1000ms ME calculation >> 0.4ms ME copy)
  - We expect that *this will not be an issue for typical LHC collision processes*

ZOOM (ME calculation >> ME copy)

$gg \rightarrow t\bar{t}gg$

A. Valassi – Reengineering Madgraph5_aMC@NLO for GPUs and vector CPUs   vCHEP – 19 May 2021   15

- We are lucky: the more complex the physics process, the less relevant is the cost of GPU-CPU data copies!
  - Similarly (later): the more complex the process, the less relevant is the overhead from scalar Fortran in madevent!
  - And the fewer events in flight needed to fill the GPU...

- In this talk I mainly give performance numbers for complex processes like $gg \rightarrow t\bar{t}gg$ or $gg \rightarrow t\bar{t}ggg$

Amdahl's law

- The matrix element calculation is now the bottleneck (e.g. >95% for $gg \to t\bar{t}gg$) in Fortran Madgraph
  - But the remaining <5% may fast become the bottleneck if you accelerate the matrix element too much!

- *Amdahl's law: if the parallelizable part takes a fraction of time p, the maximum speedup is 1/(1-p)*
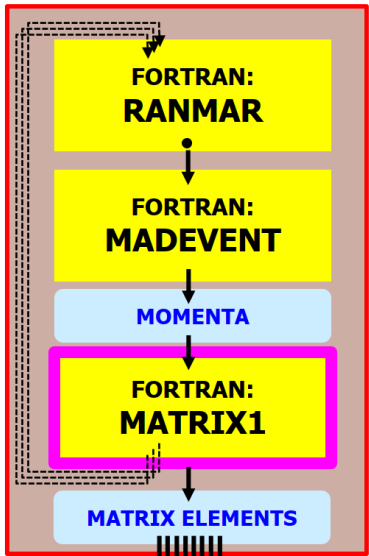 - If the MadEvent overhead takes 5%, the maximum speedup is only 20 even if your GPU speedup s is 1000!

Amdahl's Law
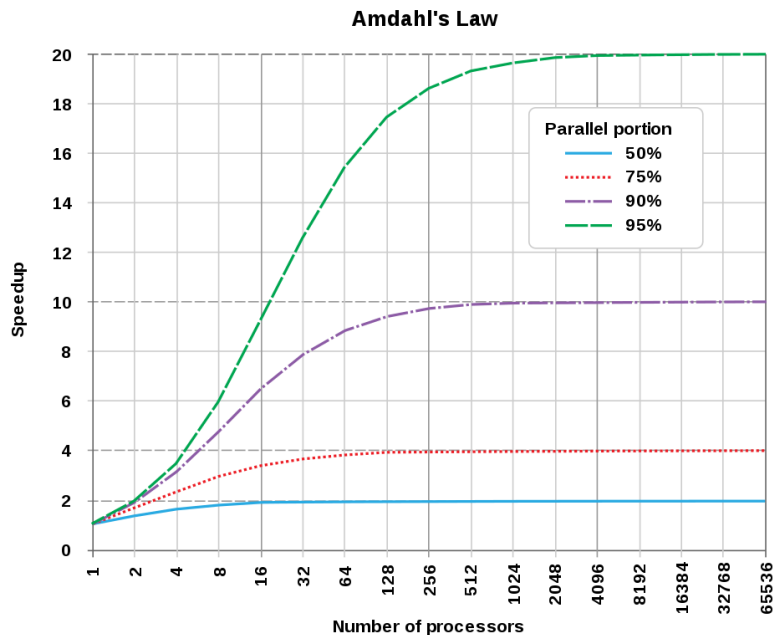
$$\lim_{s \to \infty} S_{\text{latency}}(s) = \frac{1}{1-p}$$

https://en.wikipedia.org/wiki/Amdahl%27s_law

MG5aMC: old and new architecture designs

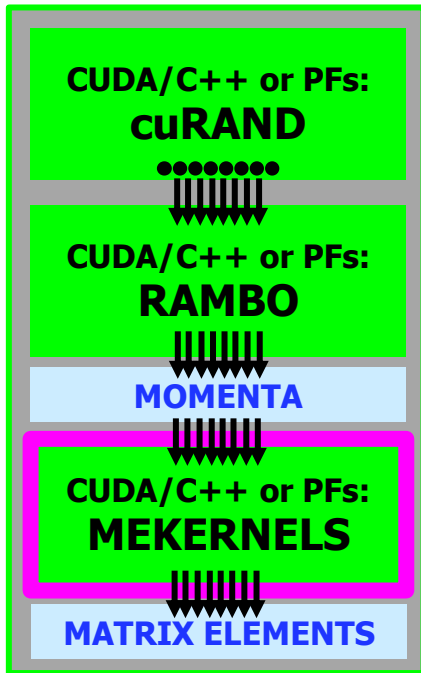**OLD MADEVENT**
(NOW: LHC PROD)
SINGLE-EVENT API

FORTRAN: **RANMAR**

FORTRAN: **MADEVENT**

MOMENTA

FORTRAN: **MATRIX1**

MATRIX ELEMENTS

MATRIX ELEMENT: CPU BOTTLENECK IN OLD MADEVENT

First we developed
the new ME engines
in standalone applications

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

**1. STANDALONE
(TOY APPLICATIONS)
MULTI-EVENT API**

CUDA/C++ or PFs: **cuRAND**

CUDA/C++ or PFs: **RAMBO**

MOMENTA

CUDA/C++ or PFs: **MEKERNELS**

MATRIX ELEMENTS

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

FORTRAN: **RANMAR**

FORTRAN: **MADEVENT**

MOMENTA

FORTRAN: **MATRIX1**

MATRIX ELEMENTS

FORTRAN: **RANMAR**

FORTRAN: **MADEVENT**

MOMENTA

CUDA/C++ or PFs: **MEKERNELS**

MATRIX ELEMENTS

(Amdahl...)

SCALAR: NEW BOTTLENECK?

PARALLEL: MUCH FASTER!

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

$gg \to t\bar{t}gg$	MEs precision	ACAT2022 $t_{TOT} = t_{Mad} + t_{MEs}$ [sec]	madevent N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]	standalone
Fortran(scalar)	double	37.3 = 1.7 + 35.6	2.20E3 (=1.0)	2.30E3 (=1.0)	—
C++/none(scalar)	double	37.8 = 1.7 + 36.0	2.17E3 (x1.0)	2.28E3 (x1.0)	2.37E3
C++/sse4(128-bit)	double	19.4 = 1.7 + 17.8	4.22E3 (x1.9)	4.62E3 (x2.0)	4.75E3
C++/avx2(256-bit)	double	9.5 = 1.7 + 7.8	8.63E3 (x3.9)	1.05E4 (x4.6)	1.09E4
C++/512y(256-bit)	double	8.9 = 1.8 + 7.1	9.29E3 (x4.2)	1.16E4 (x5.0)	1.20E4
C++/512z(512-bit)	double	6.1 = 1.8 + 4.3	1.35E4 (x6.1)	1.91E4 (x8.3)	2.06E4
C++/none(scalar)	float	36.6 = 1.8 + 34.9	2.24E3 (x1.0)	2.35E3 (x1.0)	2.45E3
C++/sse4(128-bit)	float	10.6 = 1.7 + 8.9	7.76E3 (x3.6)	9.28E3 (x4.1)	9.21E3
C++/avx2(256-bit)	float	5.7 = 1.8 + 3.9	1.44E4 (x6.6)	2.09E4 (x9.1)	2.13E4
C++/512y(256-bit)	float	5.3 = 1.8 + 3.6	1.54E4 (x7.0)	2.30E4 (x10.0)	2.43E4
C++/512z(512-bit)	float	3.9 = 1.8 + 2.1	2.10E4 (x9.6)	3.92E4 (x17.1)	3.77E4

512y = AVX512, ymm registers
512z = AVX512, zmm registers

The latter is only better on nodes with 2 FMA units (here an Intel Gold 6148)

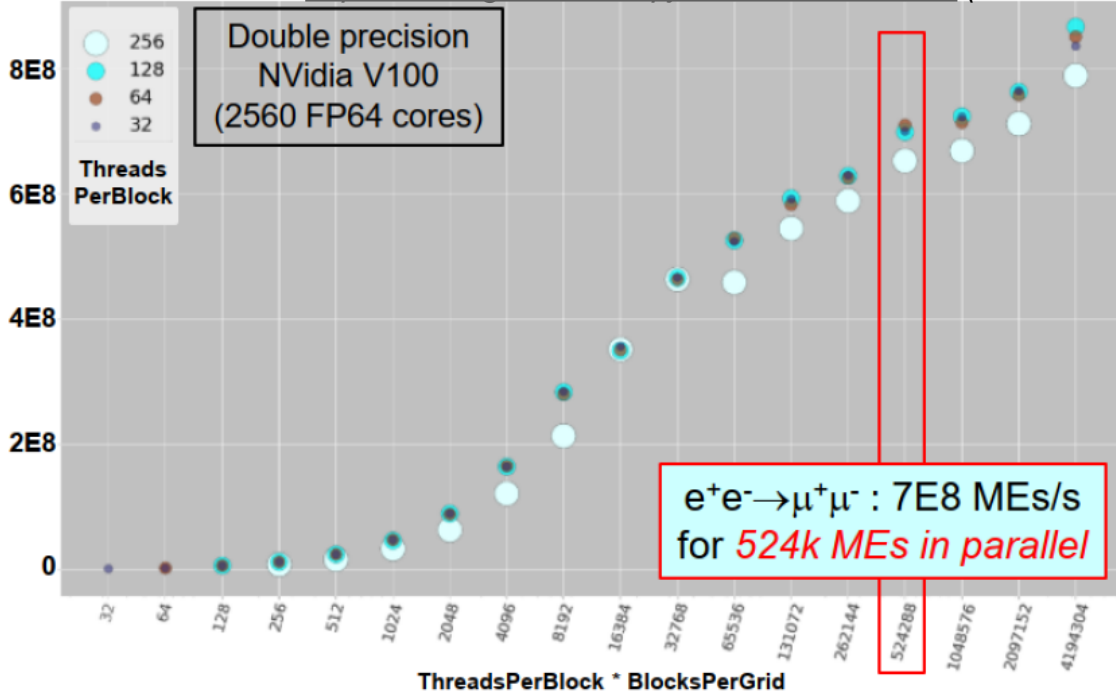**ME speedup ~ x8 (double) and x16 (float) over scalar Fortran**
Our ME engine reaches the maximum theoretical SIMD speedup!
Overall speedup so far~ x6 (double) and x10 (float) over scalar Fortran (Amdahl's law)
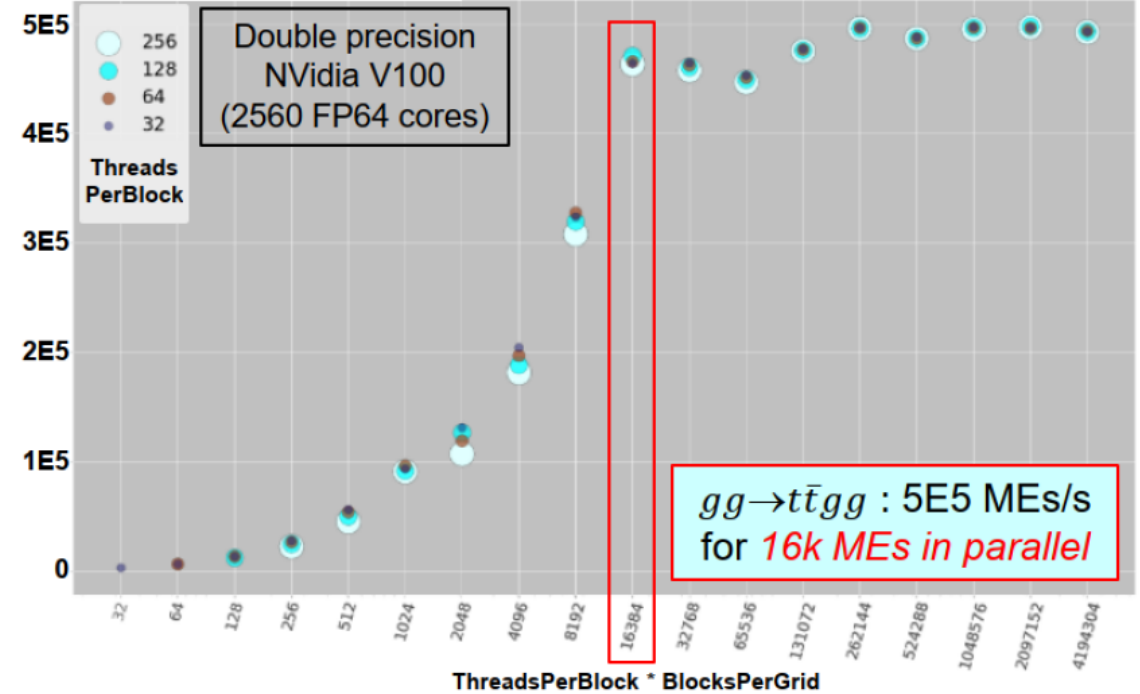
Floating point precision

- Previous slide: *our vectorized C++ on CPUs is 2x faster in single-precision than in double-precision*
  - In a 512-bit register you fit 16 (4-byte) floats but only 8 (8-byte) doubles

- Next slides: *our CUDA implementation on V100 GPUs is also 2x faster for floats than for doubles*
  - On data-center NVidia GPUs (e.g. V100 or A100), you have twice as many FLOPs in float as in double
  - Note that lower-end GPUs (e.g. T4) have almost no double-precision FLOPs...

- But *single-point precision is not enough for physics: numerical instabilities* (e.g. in Feynman diagrams)
  - It would be useful to study if these instabilities can be worked around – anyone interested? ☺
  - Alternative: we prototyped a "mixed-precision" calculation (double for Feynman, float for color matrix...)

Filling the GPU – minimum number of threads (events in flight)

Matrix Elements / second *https://doi.org/10.1051/epjconf/202125103045 (vCHEP 2021)*

$e^+e^- \to \mu^+\mu^-$: 7E8 MEs/s for *524k MEs in parallel*

Matrix Elements / second

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

- We are lucky, again: *the more complex the process, the fewer the events in flight needed to fill the GPU*

- But *even 16k events is a lot*: it results in *imbalanced phase space sampling*, and *high RAM in Fortran*
  - Eventually, maybe: one helicity per kernel (fewer events in flight, spread each event across many kernels)?
  - Eventually, maybe: many CPU cores/processes in parallel (fewer events in flight per CPU core/process)?
  - Eventually, maybe: different channels in parallel (fewer events in flight in a single channel)?

MadEvent/CUDA for gg→t̄tggg

		ACAT2022	madevent			standalone	
CUDA grid size			8192			16384	
$gg \to t\bar{t}ggg$	MEs precision	$t_{\text{TOT}} = t_{\text{Mad}} + t_{\text{MEs}}$ [sec]	$N_{\text{events}}/t_{\text{TOT}}$ [events/sec]		$N_{\text{events}}/t_{\text{MEs}}$ [MEs/sec]		
Fortran	double	1228.2 = 5.0 + 1223.2	7.34E1 (=1.0)	7.37E1 (=1.0)	—	—	
CUDA	double	19.6 = 7.4 + 12.1	4.61E3 (x63)	7.44E3 (x100)	9.10E3	9.51E3 (x129)	
CUDA	float	11.7 = 6.2 + 5.4	7.73E3 (x105)	1.66E4 (x224)	1.68E4	2.41E4 (x326)	
CUDA	mixed	16.5 = 7.0 + 9.6	5.45E3 (x74)	9.43E3 (x128)	1.10E4	1.19E4 (x161)	

We are lucky! The more complex the physics process, the lower the relative overhead from the scalar Fortran MadEvent - here only 0.5%
Amdahl's law limits the overall speedup to x200 (parallelizable p=0.5%), and we achieve x60 (double) or x100 (float) in the overall speedup!

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

Relative difference in accuracy for single and double precision in color algebra
p p > t t~ j j
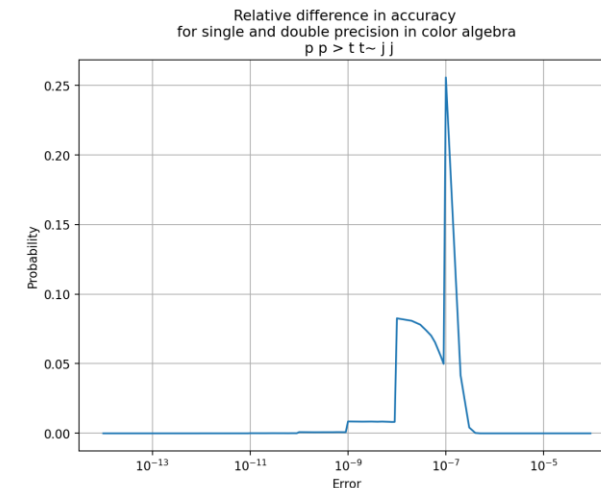
All MadEvent functionalities have been integrated over time

Most of these required some changes to the input/output API of our **Fortran-to-CUDA/C++ "Bridge"**

- *Helicity filtering* – at initialization time, compute the allowed combinations of particle helicities
  - This is computed in CUDA/C++ using the same criteria as in Fortran

- *"Multi-channel"* – single-diagram enhancement of ME output
  - This is the specificity of the MadEvent sampling algorithm (Maltoni Stelzer 2003)

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

- Event-by-event *running QCD coupling constants* $\alpha_s(Q^2)$
  - The scale is currently computed in Fortran from momenta and passed to the CUDA/C++ for each event

- Event-by-event *choice of helicity and color* in LHE files
  - Pass two additional random numbers per event from Fortran to CUDA/C++, retrieve helicity and color
  - **NEW (January 2023)!** This was the last big missing physics functionality (showstopper to a release)
    - We now get the same cross section AND the same LHE files (within numerical precision) in Fortran and CUDA/C++

The road to an alpha release (Q1-Q2 2023)

- Complete the back-port of code generation from madgraph4gpu to mg5amcnlo upstream
  - Including extra cross-checks that the LHE color IDs are those required for parton showers

- Make the CUDA/C++ madevent executable consistent with the Madgraph "launch" infrastructure
  - Modify the names and input parameters of the madevent executable
  - Ensure consistency in the handling of physics parameter card files
  - Ensure consistency of packaging and builds with the "launch" infrastructure
  - Provide and document user hooks for new configurable options (SIMD mode, GPU vector size...)
  - Tune some reasonable defaults for out-of-the-box SIMD modes and GPU grid sizes
  - Test that a Madgraph "launch" works out-of-the-box for one of our current processes like ggttgg

- Test and fix any bugs in pp collisions, including pdf integration
  - Iterate on a few other physics processes, e.g. including Susy

- Stay tuned! ☺
  - We will be happy to help your experiment in the integration!
  - NB This will use a new Fortran version too (multi-event API), need statistical validation...

CUDACPP MEs

- 95% common code + a few #ifdef's for CUDA vs C++

- Designed for NVidia GPUs (so far: will add HIP/AMD)
  – Full feature support, e.g. tensor cores, streams, graphs

- Designed upfront for SIMD speedups on vector CPUs

  Intel®AVX512

- WIP on CPU multithreading and heterogeneous modes

PF MEs

- Write code once for many CPU/GPU vendors

- Support NVidia, AMD and Intel GPUs out-of-the-box
  – Limited support for vendor-specific features

- SIMD added via SYCL in Jan 2023, analysing results

- CPU multithreading out of the box

For the moment: we plan to continue development in parallel using both approaches – comparisons are very useful!

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

CUDACPP vs. Portability Frameworks – recap

- CUDAPP (our initial implementation) is where we add new features first

- The SYCL implementation of MG5aMC is now almost at the same level, the KOKKOS one somewhat behind

- The ALPAKA implementation of MG5aMC is no longer maintained

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

CUDACPP vs PFs - GPU ME throughputs (standalone application)

Variable GPU-grid size (throughput scan)

Fixed GPU-grid size (throughput plateau)
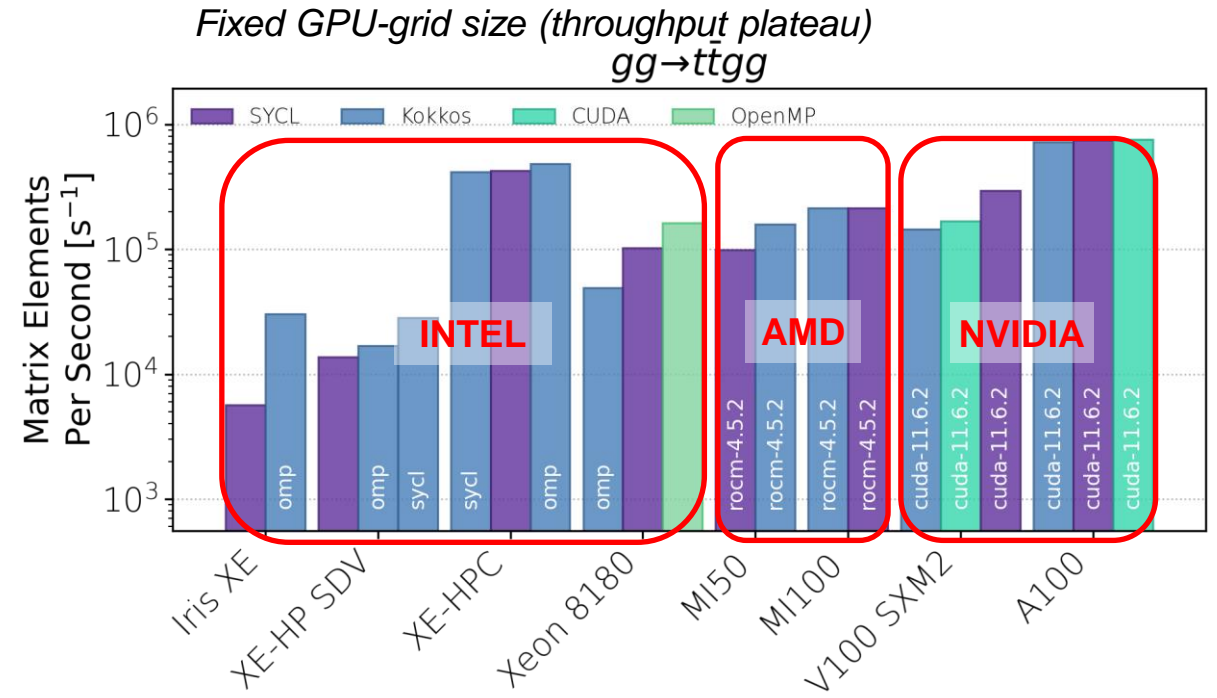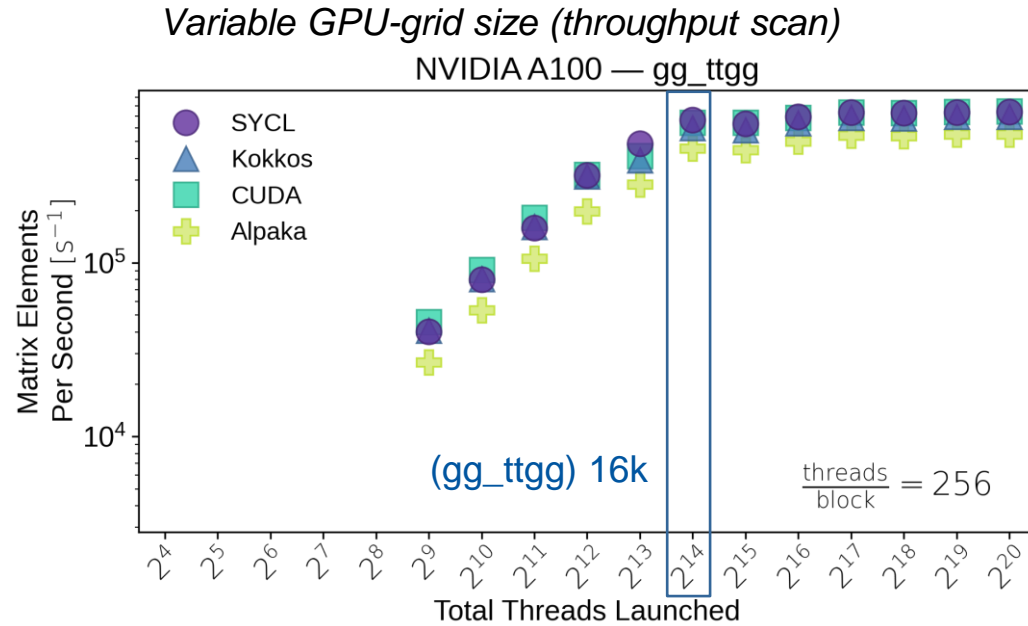
- The performances of the SYCL and Kokkos implementations of MG5aMC seem comparable to direct CUDA
  - Further comparisons are in progress, performance scales differently with more jets for different backends (next slide)

- SYCL and Kokkos run out of the box also on AMD and Intel GPUs
  - They also run out of the box on CPUs (performance under investigation)

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

XE-HPC is an early implementation of the Aurora GPU
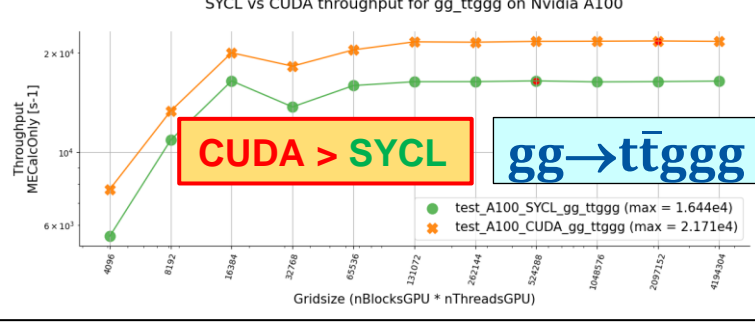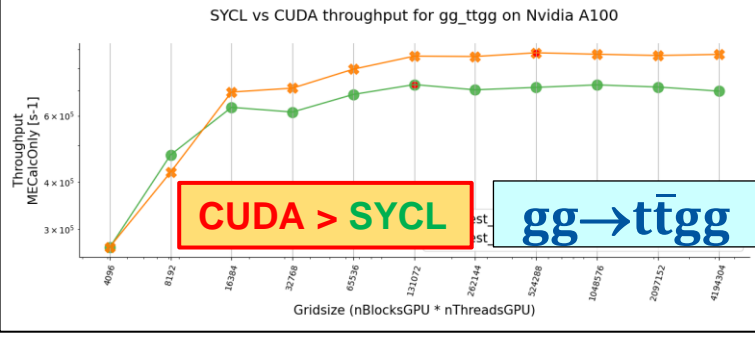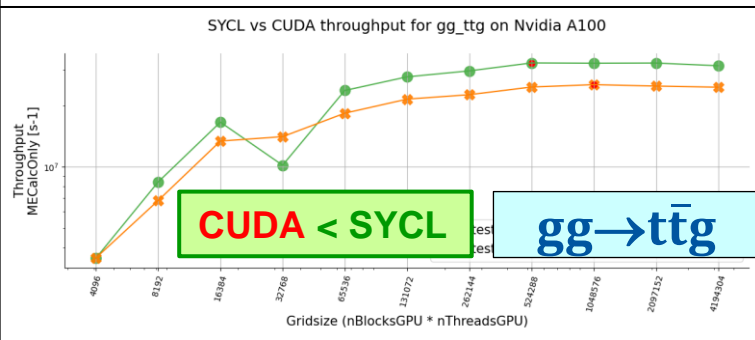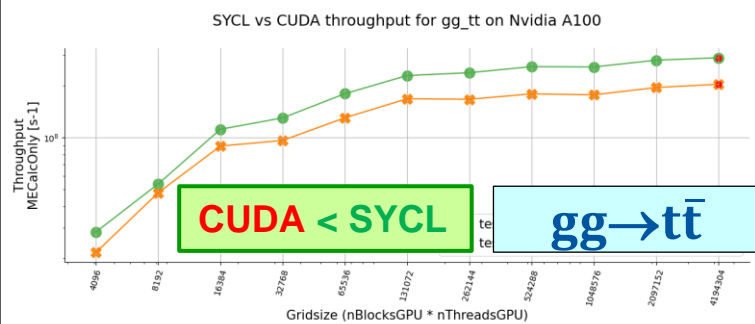
CUDA vs SYCL on NVidia A100

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

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

- These are very recent results, which are still being digested (WIP!)
  - It will be very interesting to understand more in detail what goes on

We plan to also compare more systematically the CUDACPP and SYCL performances on CPUs (vectorization, multi-core), but it will take time and optimization tweaks... WIP!

(3) Outlook: WIP, plans, ideas for more speed and features

Q1-Q2 alpha release?...

2018 2019 2020 2021 **2022** Q3 Q4 | Q1 Q2 Q3 Q4 **2023** | Q1 Q2 Q3 Q4 **2024**

Sep 2022 HEPscore
benchmarking workshop

Sep 2022 Lugano hackathon
(sub-event-level parallelism...)

From Q3 2022: Zenny's PhD work
(reweighting, NLO...)
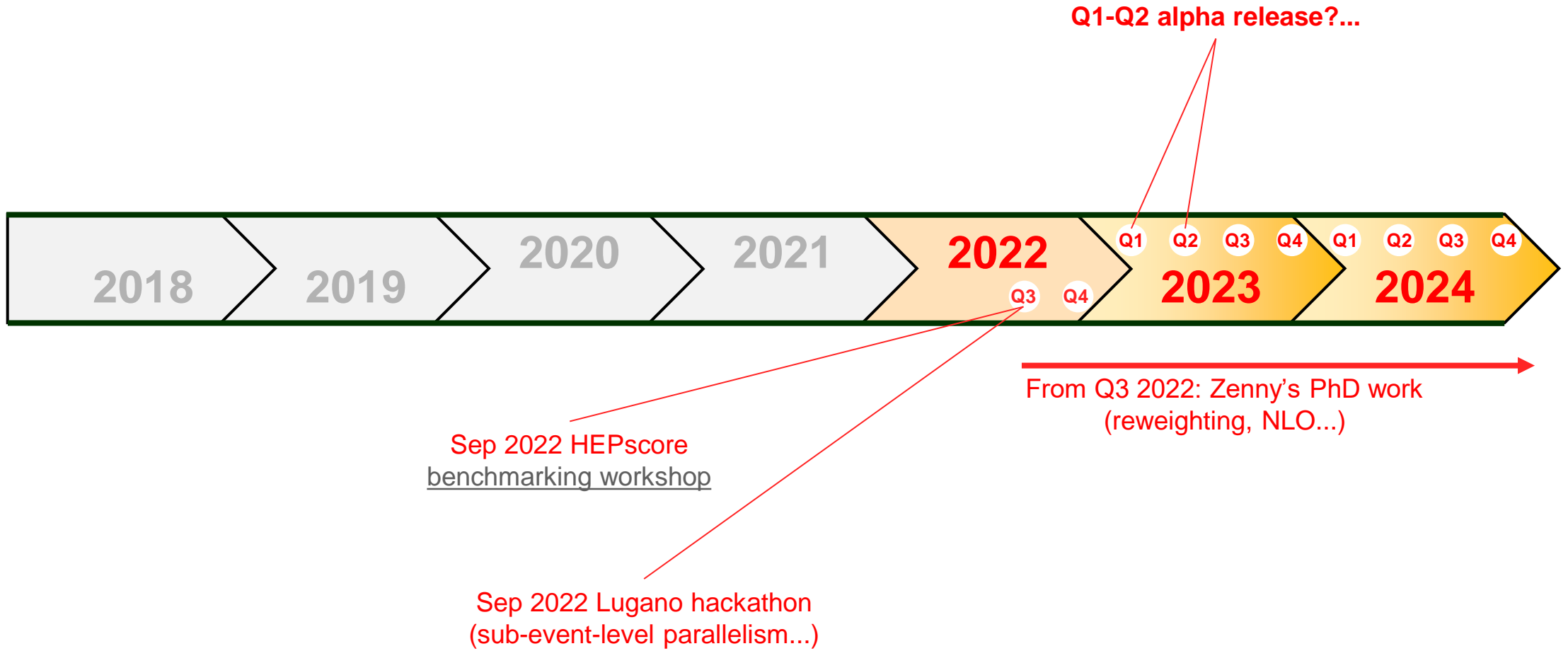
Benchmarking – Madgraph and the HEP-SCORE project

- HEPscore: the new HEP benchmark for compute resources, replacing HepSpec06
  - Based on *reproducible HEP workloads* (GEN, SIM, DIGI, REC...) within docker containers
  - The first version HEPscore23 should become production in April 2023 for (x86 and ARM) CPUs

- The aim is to *benchmark a fully loaded server:* all CPU cores, and eventually all associated GPUs
  - (and ideally measure how well an application is doing compared to the theoretical power of the server...)
  - fill all CPU cores by a combination of application multi-threading and/or several identical copies/processes

- A first container based on our Madgraph-on-GPU has been prepared
  - Very useful because it gives the same physics results on CPU and GPU: may compare them to each other!
  - And eventually may be used to evaluate heterogeneous processing on CPU+GPU...

- *The plots on the next slides are based on this HEPscore container: several identical copies/processes*
  - (A multi-threaded CUDACPP version exists but not optimized yet – SYCL and Kokkos also provide MT)

ME throughput in C++ for gg→t̄tgg (on all the cores of a CPU)

ggttgg check.exe scalability on "bmk6130" (2x 16-core 2.1GHz Xeon Gold 6130 with 2x HT) for 10 cycles
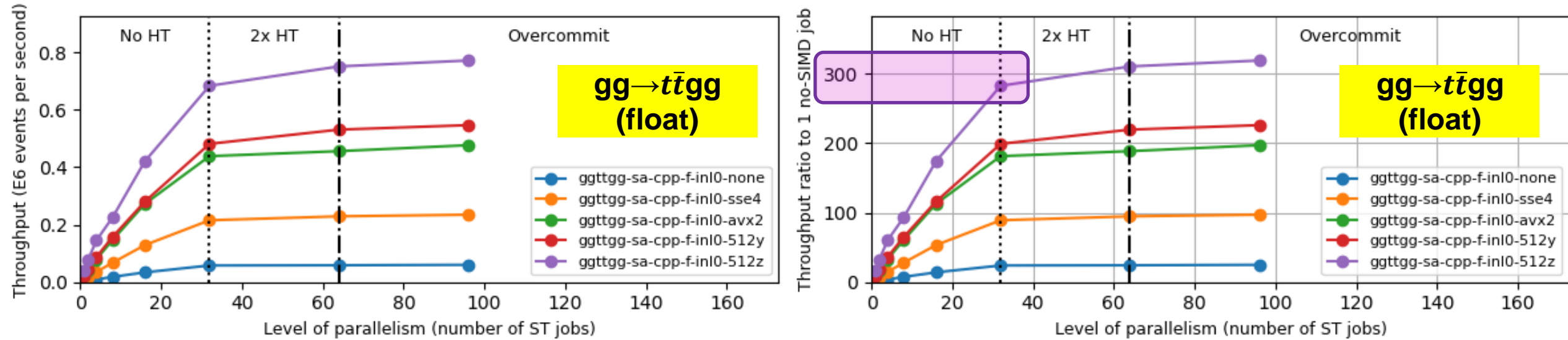
- Previous tables for SIMD speedups on C++ were for a single CPU core

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

Some ideas for heterogeneous processing
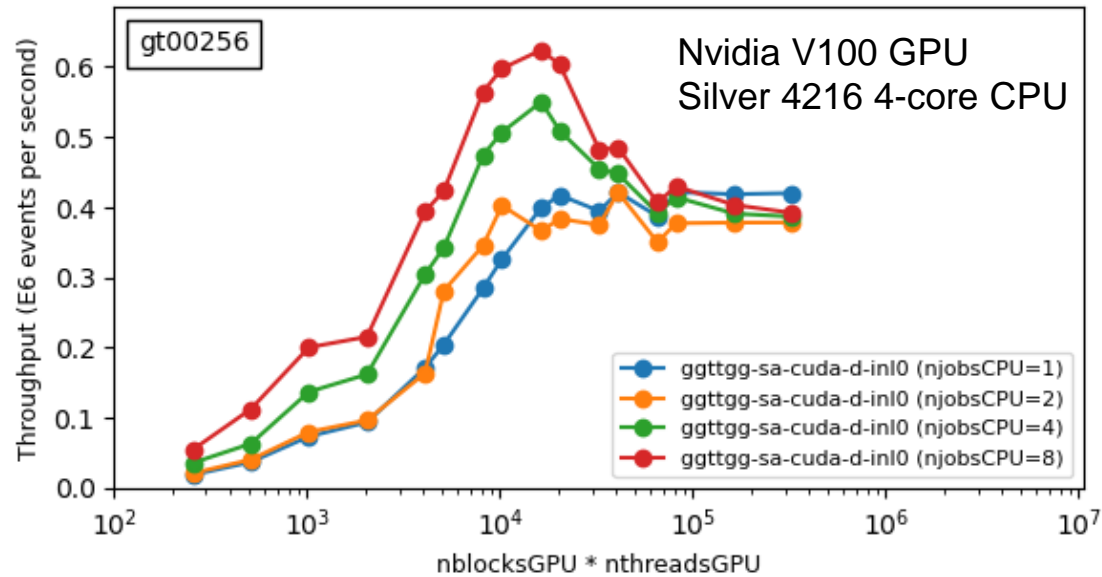
Throughput variation as a function of
*GPU grid size (#blocks * #threads)*

*This is the number of events
processed in parallel in one cycle*

To further reduce the relative overhead of the scalar Fortran MadEvent - parallelize it on many CPU cores?

- Blue curve: one single CPU process using the GPU
    - *For gg→t$\bar{t}$gg, you need at least ~16k events to reach the throughput plateau*

- Yellow, Green, Red curves: 2, 4, 8 CPU processes using the GPU at the same time
    - *Fewer events in each GPU grid are needed to reach the plateau if several CPU processes use the GPU*
    - The total Fortran RAM would remain the same, but the CPU time in the Fortran overhead would be reduced
    - (Why total throughput increases beyond the nCPU=1 plateau is not understood yet!...)

Lockstep beyond event-level parallelism

- Efficient data parallelism (lockstep processing) requires the *same function computed for different data*
  - This is true in MG5AMC at the *event level* (different events i.e. different phase space points)
  - But it is also true at the *sub-event level* (different helicities within the same event)

- We are evaluating the move to a different data parallelism strategy on GPUs
  - Currently: *one event (sum over all helicities) per GPU thread*
  - In the future: *one helicity of one event per GPU thread?*

$$|\mathcal{M}|^2(\vec{p}) = \sum_{\lambda \in \{hel\}} \left[\sum_{f,g} (J_\lambda(\vec{p}))^f (\mathcal{C})^{fg} (J_\lambda^*(\vec{p}))^g \right] \qquad (J_\lambda(\vec{p}))^f = \sum_{d \in \{diag\}} (\mathcal{M}_\lambda^d(\vec{p}))^f$$

- Advantages:
  - You can fill the GPU with much fewer "events in flight" – more balanced sampling/integration in MadEvent
  - This is a prerequisite for moving the color matrix to externally-launched cuBLAS and tensor cores
  - This is also a prerequisite if we want to evaluate much smaller kernels
    - *From all Feynman diagrams in one kernel to one Feynman diagram per kernel?*
    - Which might decrease register pressure and increase kernel occupancy, but would require more global memory access
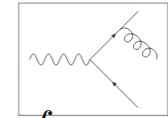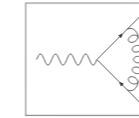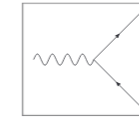
NLO, loops

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

Matching NLO QCD and parton showers (avoid double counting)

Marco Zaro – https://cp3.irmp.ucl.ac.be/projects/madgraph/wiki/Pavia2015

B, V, R: matrix elements
MC: parton shower

$$d\sigma^n_{NLO} \quad = \quad d\sigma^n_{LO} \quad + \quad d\sigma^n_V \quad + \int d\Phi_1 \, d\sigma^{n+1}_R$$

$$\frac{d\sigma_{\text{``MC@NLO''}}}{dO} = \left[\int d\Phi_n (B + V + \int d\Phi_1 MC)\right] I^n_{MC}(O) + \left[\int d\Phi_{n+1}(R - MC)\right] I^{n+1}_{MC}(O)$$

S-events   H-events

S and **H** events: two separate sets of events (different matrix elements)
Integral = S+H is positive – but individual events can have negative weights

- So far we have only worked on LO QCD processes!

- NLO QCD processes are more computationally intensive
  - They have more Feynman diagrams
  - But especially they have loop diagrams!
  - And, a matching procedure (MC@NLO) must be applied

- *We should be able to compute Born and Real emission contributions in our vectorized C++ and CUDA*
  - We should also be able to handle NLO matching using the current MadEvent based infrastructure
  - The main challenge will be understanding the computational impact of loops (Amdahl)?

Reweighting

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

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

2. Full detector simulation
(MC truth event properties $\mathbf{x}_i^{(true)} \rightarrow$ observed event properties $\mathbf{x}_i$)

*3. **Reweight each event by matrix element ratio***

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

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

ALEPH
4q selection
● Data (used to fit)
○ Data (not used to fit)
— MC (m_W = 81.30 GeV/c²)
■ Non-WW background
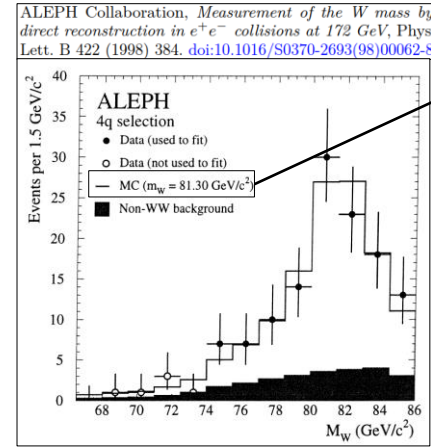
$$w_i(m_W, \Gamma_W) = \frac{|\mathcal{M}(m_W, \Gamma_W, p_i^1, p_i^2, p_i^3, p_i^4)|^2}{|\mathcal{M}(m_W^{MC}, \Gamma_W^{MC}, p_i^1, p_i^2, p_i^3, p_i^4)|^2}$$

Old technique, renewed interest!

- Advantages of reweighting: savings in computing costs (no detector simulation), fewer statistical fluctuations

- In practice *for MG5AMC: read in an LHE file, add weights, write back the modified LHE file*
    - *Will use the new matrix element engine in CUDA/C++*
    - For further details and a status report: Zenny's upcoming poster at CHEP 2023!

- Theoretical and technical challenges
    - NLO reweighting (see O. Mattelaer, https://arxiv.org/abs/1607.00763)
    - Coverage of phase space in the new parameter set
    - Reweighting for a given event-by-event helicity and color

Reweighting and weight derivatives in parameter estimation

- Weight derivative: event-by-event sensitivity to the measured parameter $\gamma_i|_\theta = \left(\dfrac{1}{w_i}\dfrac{\partial w_i}{\partial \theta}\right)_\theta$

- First: makes it possible to determine the limit error with an ideal detector, and how much (0 to 1) we do worse
  - *with a given luminosity of FCC, what is the best theoretically achievable measurement on this parameter?*

Knowing one's limits: maximum achievable information with an ideal detector
- Ideal acceptance, select all signal events $S_{sel}=S_{tot}$
- Ideal resolution, measured γ_i is that from MC truth
(implies ideal rejection of background events, $\gamma_i=0$)

$$\mathcal{I}_\theta^{(\text{ideal})} = \sum_{i=1}^{N_{\text{tot}}} \gamma_i^2 = \sum_{i=1}^{S_{\text{tot}}} \gamma_i^2$$

$$\text{FIP} = \frac{\mathcal{I}_\theta}{\mathcal{I}_\theta^{(\text{ideal})}} = \frac{(\Delta\theta^{(\text{ideal})})^2}{(\Delta\theta)^2} \leq 100\%$$

- Second: can be used as a basis for an "improved optimal observable" ML method

Weight Derivative Regression

$\gamma_i^{(\text{MC truth})} \sim q(\, x_i^{(\text{MC})}\,)$

Data observable
event properties $x_i^{(\text{DATA})}$

Fit WDR regressor
$q_i^{(\text{DATA})} = q(\, x_i^{(\text{DATA})}\,)$
$q_i^{(\text{MC})} = q(\, x_i^{(\text{MC})}\,)$

https://doi.org/10.1051/epjconf/202024506038
https://zenodo.org/record/3715951

Beyond Madgraph

- *ANY matrix element event generator is a perfect fit for vectorization and GPUs!*
  - Opportunities for 100% lockstep processing at the event-level and event the sub-event level
  - Had some discussions with the Sherpa team about vectorization...

- Beyond matrix element event generators: *are parton showers a good fit for vectorization and GPUs?*
  - More stochastic branching than in ME event generators
  - But the calculations before/after each PS splitting are the same? Would just need some basketization...
  - Some discussions a few months ago with the Pythia team on possible collaborations

Acknowledgements

- We gratefully acknowledge the computing resources provided by the Joint Laboratory for System Evaluation at Argonne National Laboratory, the Jülich Supercomputing Centre at Forschungszentrum Jülich, the Super Computing Applications and Innovation department at CINECA, and the EuroHPC Joint Undertaking at CSC's Kajaani data centre

- Many thanks from me and the whole team to:
    - Our CERN IT colleagues (especially Ricardo Rocha and Ulrich Schwickerath) for their help with GPU nodes!
    - Sebastien Ponce, Hadrien Grasland, Steve Lantz, Marco Clemencic for their suggestions on vectorization
    - Igor Vorobtsov and Klaus-Dieter Oertel for useful discussions on vectorization on Intel CPUs
    - The organizers and our mentors at the Sheffield 2020 and Lugano 2022 GPU hackathons
    - Domenico Giordano and the HEPiX benchmarking WG
    - The organizers of this Compute Accelerator Forum
    - The HSF event generator WG
    - The original authors of Madgraph5_aMC@NLO

- Many thanks from me personally to the whole madgraph4gpu team for the great collaboration! ☺

~~Executive summary for the impatient~~ Conclusions!

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

- Our reengineering of MG5aMC is close to a first fully functional alpha release for LO QCD processes
  - *The new ME calculation is integrated in MadEvent* – we get the same cross section *and LHE files* as in Fortran!

- On CPUs, in vectorized C++ we *reach the maximum x8/x16 (double/float) SIMD speedup for MEs alone*
  - The speedups achieved for the overall workflow are slightly lower due to *Amdahl's law*, but not much
  - Example: our *current overall speedup is x6/x10 (double/float)* for gg→t̄tgg (on one CPU core)

- On GPUs, using CUDA we *achieve O(100-1000) speedups for MEs alone over one no-SIMD CPU core*
  - The speedups may be much lower due to *Amdahl's law*, but we are improving on that
  - Example: our *current overall speedup is x60/x100 (double/float)* for gg→t̄tggg on an NVidia V100

- Floats are x2 faster than doubles in SIMD and NVidia GPUs – we also added 'mixed' precision modes

- In SYCL we get ~similar performances to CUDA on NVidia and we may run also on AMD or Intel GPUs

- Future challenges include optimizing heterogenous processing on one GPU and multiple CPU cores

BACKUP SLIDES

Functional tests – CI, standalone, madevent

- Functional tests in the CI
  - Starting from some hardcoded momenta, reproduce some hardcoded matrix elements (within precision)
  - A few different processes, architectures, all in double-float-mixed precision

- Functional tests for standalone tests
  - Always use the same random seeds, visually check that results do not change in logs

- Functional tests for madevent
  - Compute cross sections and generate LHE event files in Fortran
  - Then do the same in CUDA/C++ and compare results (within precision)
  - Note: for floats, the tolerance for allowed differences is now very generous...

Build challenges and issues

- The build times increase enormously as you add more final state gluons
  - Many more diagrams...
    - ...but also the result of some whole-program link time optimization? (we disable RDC in CUDA)
  - This may improve if/when we go to smaller kernels
  - We use ccache to ease the pain
    - Conversely, ninja would probably not help as we have few very large compilation units, not many small ones

- We currently build separately for our five SIMD modes
  - Eventually we might move to a "fat binary" approach with dynamic choice of the best implementation

A FEW OLD SLIDES

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

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

Table header label: **ACAT2022**

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

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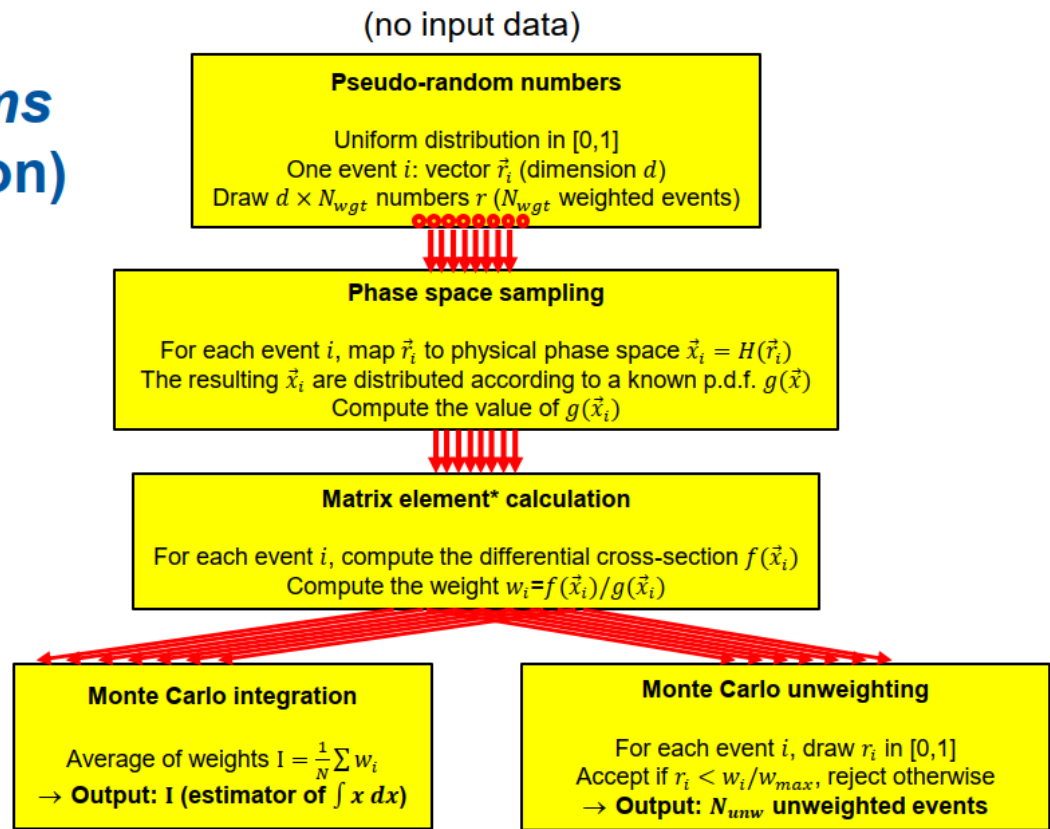## *Data-parallel paradigms (GPUs and vectorization)*

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

Monte Carlo unweighting

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

*Note for software engineers: these calculations do involve some linear algebra, but "matrix element" does not refer to that! Here we __compute one "matrix element" in the S-matrix (scattering matrix)__ for the transition from the initial state to the final state
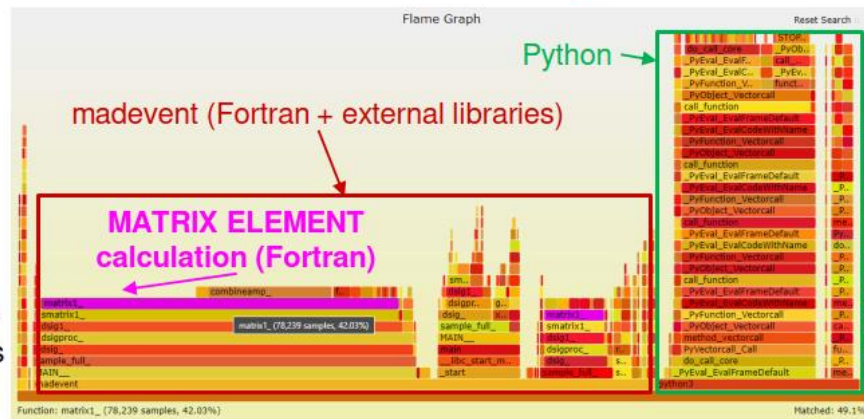
**This simple event-level parallelism can also be used as the basis for task-parallel approaches (multi-threading or multi-processing)

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

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

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

Flame Graph   Reset Search

Python →

madevent (Fortran + external libraries)

MATRIX ELEMENT calculation (Fortran)

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

matrix1_ (78,239 samples, 42.03%)

Function: matrix1_ (78,239 samples, 42.03%)   Matched: 49.1%

- 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++)

Portability Frameworks (PFs)

(2) Second line of development: MEs on PFs

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

- CUDA code can only run on NVidia GPUs, while Kokkos, Alpaka, and Sycl[Intel] codes can run on most hardware

- In "cudacpp", #ifdef directives separate code branches for GPU and CPU code during compilation (but these are very few: only kernel launching and memory access, not MEs)

- With PFs, the algorithm is typically the same, but the compilation occurs once per architecture type

- PFs often use templating to handle data types and hardware configuration and function lambdas or pointers for passing kernels (the cudacpp plugin has many of these, too)

- PFs still require user to think about "host" vs "device"

"cudacpp" example of compiler directives

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

Kokkos example of Templating & lambda

```
324  {
325    using member_type = typename Kokkos::TeamPolicy<Kokkos::DefaultExecut
326    Kokkos::TeamPolicy<Kokkos::DefaultExecutionSpace> policy( league_size
327    Kokkos::parallel_for(__func__,policy,
328    KOKKOS_LAMBDA(member_type team_member){
329
```

Kokkos example of Memory Management

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