Computational developments
in Madgraph5_aMC@NLO
on GPUs and vector CPUs

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

Oliver Mattelaer

QCD@LHC2022, Orsay, 28th November 2022
https://indico.cern.ch/event/1150707/contributions/5147183
Motivation: Monte Carlo Event Generators in WLCG computing

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

MC generators, the essential 1\textsuperscript{st} step in simulation, use \textasciitilde5\%-20\% of ATLAS/CMS WLCG CPU budget
  - Many ways to speed them up – see the HEP Software Foundation (HSF) Generator WG review paper
  - MC generators are ideal candidates to exploit data parallelism in GPUs (SIMT) and in vector CPUs (SIMD)

This plot is probably obsolete by now! See Max Knobbe’s talk on Sherpa’s speedups!
Madgraph5_aMC@NLO (MG5aMC)

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

Madgraph5_aMC@NLO (MG5aMC)

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

J. Alwall,* R. Frederix,* S. Frixione,* V. Hirschi,* F. Maltoni,* O. Mattelaer,* H. S. Shao,* T. Stelzer,1 P. Torrielli1 and M. Zaro,1

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. $gg\rightarrow t\bar{t}ggg$)
    • And ME calculations are precisely one component that can be “easily” accelerated on GPUs and on vector CPUs...
MG5aMC and the madgraph4gpu project

- **madgraph4gpu**: speed up Matrix Element calculation in MG5aMC on GPUs and vector CPUs
  - Collaboration of theoretical/experimental physicists with software engineers – born in the HSF generator WG
  - Extensive details may be found in the vCHEP2021 and ICHEP2022 conference proceedings

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

- Two types of executables over time (in each of cudacpp and PFs)
  - (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!
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.
ANY MC event generator is a great fit for GPUs and vector CPUs!

- Monte Carlo methods are based on drawing (pseudo-)random numbers: a dice throw
- From a software workflow point of view, these are used in two rather different cases:

**MC SAMPLING**

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

SAME CALCULATION ON DIFFERENT DATA!

**INPUT**

![Dice](https://via.placeholder.com/150)

![Dice](https://via.placeholder.com/150)

**OUTPUT**

Lockstep processing Good for SIMT/SIMD

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

**MC DECISIONS**

**INPUT**

![Dice](https://via.placeholder.com/150)

![Dice](https://via.placeholder.com/150)

**OUTPUT**

Detector simulation (Geant4)
- 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

*NB: MULTI-EVENT API!*

Data parallelism

Stochastic branching Bad for SIMT/SIMD

Stochastic branching Bad for SIMT/SIMD
In practice in MG5aMC: use helicity amplitudes and QCD color decomposition

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

$$
(J^\lambda(p))^f = \sum_{d \in \{\text{diag}\}} (M^d_{\lambda}(p))^f
$$

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

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

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

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

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

Each step computes many events $\bar{p}$ in parallel! CPU: one SIMD vector at a time. GPU: one event per thread.
MadEvent throughputs (MEs per second) with CUDA for $gg \rightarrow t\bar{t}gg$

<table>
<thead>
<tr>
<th>CUDA grid size</th>
<th>MEs precision</th>
<th>$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]</th>
<th>$N_{events}/t_{TOT}$ [events/sec]</th>
<th>$N_{events}/t_{MEs}$ [MEs/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gg \rightarrow t\bar{t}gg$</td>
<td>double</td>
<td>55.4 = 2.4 + 53.0</td>
<td>1.63E3 (=1.0)</td>
<td>1.70E3 (=1.0)</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>2.9 = 2.6 + 0.35</td>
<td>3.06E4 (x18.8)</td>
<td>2.60E5 (x152)</td>
</tr>
<tr>
<td></td>
<td>float</td>
<td>2.8 = 2.6 + 0.24</td>
<td>3.24E4 (x19.9)</td>
<td>3.83E5 (x225)</td>
</tr>
</tbody>
</table>

- ME calculation alone: accelerated by x150/x225 (double/float) on GPU with respect to Fortran on one CPU core
  - (With a GPU grid size of 8k, limited by MadEvent RAM - and could reach x250/x500 with a larger grid size of 524k)
  - NB: a typical data-center GPU (here a V100) has twice as many FLOPs in single precision than in double precision...

- Overall workflow speedup is x20 (double/float) - maximum achievable as scalar part is 5% (Amdahl's law)

- Must reduce the scalar MadEvent Fortran overhead (random numbers, sampling algo, I/O, MLM merging...)
  - (This was already reduced by a factor 2 between July/ICHEP and October/ACAT... and more work is in progress!)
  - (Or maybe run MadEvent on several CPU cores that share a single GPU...)
Even more interesting: MadEvent/CUDA for $gg \rightarrow t\bar{t}ggg$

<table>
<thead>
<tr>
<th>CUDA grid size</th>
<th>madevent</th>
<th>standalone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$gg \rightarrow t\bar{t}ggg$</td>
<td>$t_{TOT} = t_{Mad} + t_{MES}$ [sec]</td>
<td>$N_{events}/t_{TOT}$ [events/sec]</td>
</tr>
<tr>
<td>Fortran double</td>
<td>1228.2 = 5.0 + 1223.2</td>
<td>7.34E1 (=1.0)</td>
</tr>
<tr>
<td>CUDA double</td>
<td>19.6 = 7.4 + 12.1</td>
<td>4.61E3 (x63)</td>
</tr>
<tr>
<td>CUDA float</td>
<td>11.7 = 6.2 + 5.4</td>
<td>7.73E3 (x105)</td>
</tr>
<tr>
<td>CUDA mixed</td>
<td>16.5 = 7.0 + 9.6</td>
<td>5.45E3 (x74)</td>
</tr>
</tbody>
</table>

We are lucky! The more complex the physics process, the lower the relative overhead from the scalar Fortran MadEvent - here only 0.5% Amdahl’s law limits the overall speedup to x200, and we achieve 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)
MadEvent with vectorized C++ for $gg \rightarrow t\bar{t}gg$ (on a single CPU core)

<table>
<thead>
<tr>
<th>MEs precision</th>
<th>$t_{TOT} = t_{Mad} + t_{MEs}$ [sec]</th>
<th>$N_{events}/t_{TOT}$ [events/sec]</th>
<th>$N_{events}/t_{MEs}$ [MEs/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran (scalar)</td>
<td>double $37.3 = 1.7 + 35.6$</td>
<td>$2.20E3$ (1.0)</td>
<td>$2.30E3$ (1.0)</td>
</tr>
<tr>
<td>C++/none (scalar)</td>
<td>double $37.8 = 1.7 + 36.0$</td>
<td>$2.17E3$ (1.0)</td>
<td>$2.28E3$ (1.0)</td>
</tr>
<tr>
<td>C++/sse4 (128-bit)</td>
<td>double $19.4 = 1.7 + 17.8$</td>
<td>$4.22E3$ (1.9)</td>
<td>$4.62E3$ (2.0)</td>
</tr>
<tr>
<td>C++/avx2 (256-bit)</td>
<td>double $9.5 = 1.7 + 7.8$</td>
<td>$8.63E3$ (3.9)</td>
<td>$1.05E4$ (4.6)</td>
</tr>
<tr>
<td>C++/512y (256-bit)</td>
<td>double $8.9 = 1.8 + 7.1$</td>
<td>$9.29E3$ (4.2)</td>
<td>$1.16E4$ (5.0)</td>
</tr>
<tr>
<td>C++/512z (512-bit)</td>
<td>double $6.1 = 1.8 + 4.3$</td>
<td>$1.35E4$ (6.1)</td>
<td>$1.91E4$ (8.3)</td>
</tr>
<tr>
<td>C++/none (scalar)</td>
<td>float $36.6 = 1.8 + 34.9$</td>
<td>$2.24E3$ (1.0)</td>
<td>$2.35E3$ (1.0)</td>
</tr>
<tr>
<td>C++/sse4 (128-bit)</td>
<td>float $10.6 = 1.7 + 8.9$</td>
<td>$7.76E3$ (3.6)</td>
<td>$9.28E3$ (4.1)</td>
</tr>
<tr>
<td>C++/avx2 (256-bit)</td>
<td>float $5.7 = 1.8 + 3.9$</td>
<td>$1.44E4$ (6.6)</td>
<td>$2.09E4$ (9.1)</td>
</tr>
<tr>
<td>C++/512y (256-bit)</td>
<td>float $5.3 = 1.8 + 3.6$</td>
<td>$1.54E4$ (7.0)</td>
<td>$2.30E4$ (10.0)</td>
</tr>
<tr>
<td>C++/512z (512-bit)</td>
<td>float $3.9 = 1.8 + 2.1$</td>
<td>$2.10E4$ (9.6)</td>
<td>$3.92E4$ (17.1)</td>
</tr>
</tbody>
</table>

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

Speeding up Madgraph5_aMC@NLO on GPUs and vector CPUs

A. Valassi – QCD@LHC2022, Orsay, 28 November 2022
ME throughput in C++ for $gg \rightarrow t\bar{t}gg$ (on all the cores of a CPU)

- 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)
CUDACPP MEs

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

PF MEs

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

For the moment: we plan to continue development in parallel using both approaches

Two goals: not only production releases, but also aim to provide useful feedback to HEP about usability of PFs
• The performances of the SYCL and Kokkos implementations of MG5aMC seem comparable to direct CUDA
  – Further comparisons are in progress for ggtt(g(gg))) – performance scales differently with more jets for different backends

• 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

Speeding up Madgraph5_aMC@NLO on GPUs and vector CPUs

A. Valassi – QCD@LHC2022, Orsay, 28 November 2022
Status, WIP and outlook (1) - functionalities

• Cross-section calculation: done
  – Replacing Fortran MEs by CUDA/C++ MEs yields the same cross section to E-14 (same random numbers)

• Unweighted event generation: almost done
  – Replacing Fortran MEs by CUDA/C++ MEs yields almost the same LHE files...
  – ... still missing: event-by-event random choices of one helicity and one leading colour
  – Our main priority before an alpha release supporting LO processes for the LHC experiments!

• Work in progress: use the new ME engines for event-by-event reweighting of existing LHE files
  – Event weights (and weight derivatives?) for different choice of parameters

• Longer term (next year): add support for NLO QCD processes

• In parallel: further comparisons of different implementations (cudacpp, PFs...) on different platforms
Status, WIP and outlook (2) - performance

• Speed up the Matrix Element calculation in CUDA
  – Smaller kernels (and fewer events per grid): from one-event/all-helicities to one-event/one-helicity per thread
  – Smaller kernels: split Feynman diagrams and colour matrix calculations
  – Move colour matrix calculation to GPU tensor cores (e.g. using cublas)

• Speed up the Fortran MadEvent scalar component
  – Parallelize it on the many cores of the CPU (heterogeneous workflow)?
  – Further profiling...

• Further analysis of the effect of numerical precision (double vs float)
Conclusions

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

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

• On CPUs, using vectorized C++ we achieve the maximum x8/x16 (double/float) SIMD speedups for MEs alone
  – The speedups for the overall workflow are slightly lower due to Amdahl’s law, but not much
  – Example: our overall speedup is currently x6/x10 for gg→t̅tgg (on one CPU core)

• On GPUs, using CUDA we achieve O(100-1000) speedups for MEs alone
  – The speedups may be much lower due to Amdahl’s law, but we are improving on that
  – Example: our overall speedup is currently x60/x100 (double/float) for gg→t̅tggg

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

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

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

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

• We gratefully acknowledge the use (under ISCRA-C project MG5A100) of computing resources provided and operated by CINECA.

• We thank the organizers and our mentors at the GPU Hackathon in CSCS Lugano in September 2022.
Some ideas for heterogeneous processing

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 $gg \rightarrow t\bar{t}gg$, you need at least $\sim 16k$ 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 $n_{CPU}=1$ plateau is not understood yet!...)

#### Throughput variation as a function of GPU grid size ($\text{nblocks} \times \text{#threads}$)

- This is the number of events processed in parallel in one cycle
MadEvent/C++ for \( gg \rightarrow t\bar{t}ggg \) (on a single core)

<table>
<thead>
<tr>
<th>( gg \rightarrow t\bar{t}gg )</th>
<th>MES precision</th>
<th>( \tau_{TOT} = t_{Mad} + t_{MES} ) [sec]</th>
<th>( \frac{N_{events}}{\tau_{TOT}} ) [events/sec]</th>
<th>( \frac{N_{events}}{t_{MES}} ) [MES/sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran (scalar)</td>
<td>double</td>
<td>813.2 = 3.7 + 809.6</td>
<td>1.01E2 (= 1.0)</td>
<td>1.01E2 (= 1.0)</td>
</tr>
<tr>
<td>C++/none (scalar)</td>
<td>double</td>
<td>986.0 = 4.3 + 981.7</td>
<td>8.31E1 (x0.8)</td>
<td>8.35E1 (x0.8)</td>
</tr>
<tr>
<td>C++/sse4 (128-bit)</td>
<td>double</td>
<td>514.7 = 4.2 + 510.5</td>
<td>1.59E2 (x1.6)</td>
<td>1.61E2 (x1.6)</td>
</tr>
<tr>
<td>C++/avx2 (256-bit)</td>
<td>double</td>
<td>231.6 = 4.0 + 227.6</td>
<td>3.54E2 (x3.5)</td>
<td>3.60E2 (x3.6)</td>
</tr>
<tr>
<td>C++/512y (256-bit)</td>
<td>double</td>
<td>208.6 = 3.9 + 204.8</td>
<td>3.93E2 (x3.9)</td>
<td>4.00E2 (x4.0)</td>
</tr>
<tr>
<td>C++/512z (512-bit)</td>
<td>double</td>
<td>124.6 = 4.0 + 120.6</td>
<td>6.58E2 (x6.5)</td>
<td>6.79E2 (x6.7)</td>
</tr>
<tr>
<td>C++/none (scalar)</td>
<td>float</td>
<td>936.1 = 4.3 + 931.8</td>
<td>8.75E1 (x0.9)</td>
<td>8.79E1 (x0.9)</td>
</tr>
<tr>
<td>C++/sse4 (128-bit)</td>
<td>float</td>
<td>228.9 = 3.9 + 225.0</td>
<td>3.58E2 (x3.6)</td>
<td>3.64E2 (x3.6)</td>
</tr>
<tr>
<td>C++/avx2 (256-bit)</td>
<td>float</td>
<td>114.1 = 3.8 + 110.4</td>
<td>7.18E2 (x7.2)</td>
<td>7.43E2 (x7.4)</td>
</tr>
<tr>
<td>C++/512y (256-bit)</td>
<td>float</td>
<td>104.5 = 3.8 + 100.7</td>
<td>7.84E2 (x7.9)</td>
<td>8.14E2 (x8.1)</td>
</tr>
<tr>
<td>C++/512z (512-bit)</td>
<td>float</td>
<td>61.8 = 3.8 + 58.0</td>
<td>1.33E3 (x13.3)</td>
<td>1.41E3 (x14.1)</td>
</tr>
<tr>
<td>C++/none (scalar)</td>
<td>mixed</td>
<td>986.0 = 4.3 + 981.6</td>
<td>8.31E1 (x0.8)</td>
<td>8.35E1 (x0.8)</td>
</tr>
<tr>
<td>C++/sse4 (128-bit)</td>
<td>mixed</td>
<td>500.4 = 3.9 + 496.5</td>
<td>1.64E2 (x1.6)</td>
<td>1.65E2 (x1.6)</td>
</tr>
<tr>
<td>C++/avx2 (256-bit)</td>
<td>mixed</td>
<td>220.5 = 3.8 + 216.7</td>
<td>3.72E2 (x3.7)</td>
<td>3.78E2 (x3.8)</td>
</tr>
<tr>
<td>C++/512y (256-bit)</td>
<td>mixed</td>
<td>195.6 = 3.7 + 191.8</td>
<td>4.19E2 (x4.2)</td>
<td>4.27E2 (x4.3)</td>
</tr>
<tr>
<td>C++/512z (512-bit)</td>
<td>mixed</td>
<td>118.5 = 3.8 + 114.7</td>
<td>6.92E2 (x6.9)</td>
<td>7.15E2 (x7.2)</td>
</tr>
</tbody>
</table>

- Lower overhead of scalar MadEvent in \( gg \rightarrow t\bar{t}ggg \) than in \( gg \rightarrow t\bar{t}gg \): higher overall throughput speedup \( \times 13 \)!
- Mixed floating-point precision (single precision color algebra) is 5-10% better than double
CUDACPP vs. Portability Frameworks – recap

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

• The SYCL and KOKKOS implementations of MG5aMC are now almost at the same level

• The ALPAKA implementation of MG5aMC is no longer maintained

<table>
<thead>
<tr>
<th>Backend</th>
<th>ME code generation</th>
<th>Standalone application</th>
<th>Actively maintained</th>
<th>MadEvent application</th>
<th>Latest dev code base</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUDACPP</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>SYCL</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>~ ✓</td>
</tr>
<tr>
<td>KOKKOS</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>WIP</td>
<td>~ ✓</td>
</tr>
<tr>
<td>ALPAKA (CUPLA)</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>
Fortran vs C++/CUDA/PFs: *(not yet)* an apples-to-apples comparison!

**MadEvent + Fortran (double precision)**

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

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

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

**TODAY**

**SOON!** *(THE GOAL)*

*NB: THE SAME APPLES!*
MORE BACKUP SLIDES
Matrix element integration in MadEvent: detailed results (CPU)

<table>
<thead>
<tr>
<th></th>
<th>mad</th>
<th>(81952 MEs)</th>
<th>mad</th>
<th>mad</th>
<th>sa/brdg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ggttgg</td>
<td>[sec] tot = mad + MEs</td>
<td>[TOT/sec]</td>
<td>[MEs/sec]</td>
<td>[MEs/sec]</td>
<td></td>
</tr>
<tr>
<td>Fortran</td>
<td>41.82 = 3.23 + 38.60</td>
<td>1.96e+03 (= 1.0)</td>
<td>2.12e+03 (= 1.0)</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>CPP/none</td>
<td>47.78 = 3.56 + 44.22</td>
<td>1.72e+03 (× 0.9)</td>
<td>1.85e+03 (× 0.9)</td>
<td>1.90e+03</td>
<td></td>
</tr>
<tr>
<td>CPP/sse4</td>
<td>23.04 = 2.97 + 20.07</td>
<td>3.56e+03 (× 1.8)</td>
<td>4.08e+03 (× 1.9)</td>
<td>4.05e+03</td>
<td></td>
</tr>
<tr>
<td>CPP/avx2</td>
<td>12.19 = 2.88 + 9.32</td>
<td>6.72e+03 (× 3.4)</td>
<td>8.80e+03 (× 4.2)</td>
<td>9.24e+03</td>
<td></td>
</tr>
<tr>
<td>CPP/512y</td>
<td>11.57 = 2.86 + 8.71</td>
<td>7.08e+03 (× 3.6)</td>
<td>9.41e+03 (× 4.4)</td>
<td>1.01e+04</td>
<td></td>
</tr>
<tr>
<td>CPP/512z</td>
<td>8.26 = 2.88 + 5.38</td>
<td>9.92e+03 (× 5.1)</td>
<td>1.52e+04 (× 7.2)</td>
<td>1.60e+04</td>
<td></td>
</tr>
</tbody>
</table>

**TIME**
- Total = MadEvent (scalar) + MEs (parallel)
- MadEvent (scalar)
- MEs (parallel)

**THROUGHPUT**
- MEs (within madevent)
- MEs (within standalone test application)
Matrix element integration in MadEvent: detailed results (GPU)

<table>
<thead>
<tr>
<th></th>
<th>mad</th>
<th>mad</th>
<th>mad</th>
<th>sa/brdg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ggttggg</td>
<td>[sec] tot = mad + MEs</td>
<td>[TOT/sec]</td>
<td>[MEs/sec]</td>
<td>[MEs/sec]</td>
</tr>
<tr>
<td>nevt/grid</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
<td>8192</td>
</tr>
<tr>
<td>nevt total</td>
<td>90112</td>
<td>90112</td>
<td>90112</td>
<td>256<em>32</em>1</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>1286.09 = 62.74 + 1223.35</td>
<td>7.01e+01 (= 1.0)</td>
<td>7.37e+01 (= 1.0)</td>
<td>---</td>
</tr>
<tr>
<td>CUDA/8192</td>
<td>77.06 = 64.87 + 12.19</td>
<td>1.17e+03 (x16.7)</td>
<td>7.39e+03 (x100.)</td>
<td>7.48e+03</td>
</tr>
<tr>
<td>nevt/grid</td>
<td>16384</td>
<td>512<em>32</em>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nevt total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA/max</td>
<td>9.33e+03</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Nvidia V100 GPU + Intel Silver 4216 CPU (CERN)

1. REDUCE THIS TO INCREASE SPEEDUP
2. INCREASE GPU GRIDS (REDUCE CPU MEMORY) TO INCREASE SPEEDUP
3. SMALLER GPU KERNELS TO INCREASE SPEEDUP

MadEvent (scalar)

8k events per GPU grid

16k events per GPU grid

ggttgg GPU MEs speedup is lower than eemumu (higher register pressure)
Matrix element integration in MadEvent

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

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

- 1. **Moving Madevent from single-event to many-event** (functional reengineering of the algorithm)
  - Now also an active area of performance optimizations (next slides: GPU grid and CPU RAM; CPU time and Amdahl...)
- 2. Debugging functional issues caused by hidden inputs and outputs, e.g. coming from Fortran common blocks

---

**Diagram:**

- **FORTRAN:**
  - **RANMAR**
  - **MADEVENT**
  - **MATRIX1**

- **Matrix Elements**
  - **MOMENTA**
  - **COMMON BLOCKS** (hidden inputs and outputs?)

- **Pure Functions** (clear inputs and outputs)
  - **CUDA/C++ or PFs: MEKERNELS**

---

**Flow:**

- REENGINEER MADEVENT
- ADAPT CUDACPP (and PFs)
- SINGLE event (momenta)
- MANY events (momenta)
Code generation: from many “epochs” to a single evolving “epoch”

OLD MODEL
(2020- early 2021)

NEW MODEL
(since end 2021)

Start new
“epoch”

Code generation infrastructure
- Python framework and “cudacpp” plugin
- Fortran, C++, CUDA templates
- Post-generation patches (temporary...)

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

Now using upstream MG5AMC from https://github.com/mg5amcnlo!
MG5aMC computational anatomy and data parallelism strategy

- In MC generators, the same function is used to compute the Matrix Element for many different events
  - ANY matrix element generator is a good fit for lockstep processing on GPUs (SIMT) and vector CPUs (SIMD)
  - Data parallelism strategy in madgraph4gpu is event-level parallelism (many events = many phase space points)
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 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”
CUDA/C++: ME code example (complex number scalar/vector)

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

- CUDA: scalar complex → `typedef thrust::complex<fptype> cxttype;` // two doubles: RI
- C++, no SIMD: scalar complex → `typedef std::complex<fptype> cxttype;` // two doubles: RI
- C++, with SIMD: vector complex → `class cxttype_v { fptype_v m_real, m_imag;` // RRRRIII (SOA)

```
void FFW1_0( const cxttype_sv F1[], const cxttype_sv F2[], const cxttype_sv V3[], cxttype_sv* vertex )
{
  mgDebug( 0, __FUNCTION__;
  const cxttype cI( 0., 1. );

  (*vertex) = cI * cI * THF0;
  mgDebug( 1, __FUNCTION__;
  return;
}
```

C++ SIMD: gcc / clang compiler vector extensions

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

FFV1_0: helicity amplitude for the $\gamma\mu^+\mu^-$ vertex

Soon to be automatically generated

“+” is the usual sum of two (thrust/std) scalar complex, or the user defined sum of two vector complex
CUDA: Profiling with NVidia NSight Compute – ncu

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

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

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

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

Example: compare baseline implementation (100% branch efficiency) to a test with artificial divergence
EVEN MORE BACKUP SLIDES
What is a MC generator? A simplified computational anatomy

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

This can be parallelized (SIMT/SIMD and multithreading)

For each event:
1. Output: random numbers
2. Input: random numbers
   Output: particle 4-momenta
3. Input: particle 4-momenta
   Output: Matrix Element (ME) - CPU BOTTLENECK

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

MATRIX ELEMENT GENERATOR
(e.g. MG5aMC)

PHASE SPACE SAMPLING

WEIGHTED EVENTS
\{EVT_{i}, W_{i}\}

MONTE CARLO INTEGRATION

MONTE CARLO UNWEIGHTING

UNWEIGHTED EVENTS
\{EVT_{i}, W_{i}=1\}

CROSS-SECTIONS etc...
\{AVG W_{i}, MAX W_{i}\}

SHOWERING AND
HADRONIZATION
GENERATORS
(e.g. PYTHIA)

PARTON
SHOWER

HADRONTISATION
AND DECAY

PARTICLE
FILTERING

DETECTOR
SIMULATION
(GEANT4)
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

<table>
<thead>
<tr>
<th>Process</th>
<th>LOC</th>
<th>functions</th>
<th>function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^+e^- \rightarrow \mu^+\mu^-$</td>
<td>776</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>$gg \rightarrow t\bar{t}$</td>
<td>839</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>$gg \rightarrow t\bar{t}g$</td>
<td>1082</td>
<td>36</td>
<td>106</td>
</tr>
<tr>
<td>$gg \rightarrow t\bar{t}gg$</td>
<td>1985</td>
<td>222</td>
<td>786</td>
</tr>
</tbody>
</table>

- **Goal**: modify code-generating code (add CUDA, improve C++ backend)
  - (1) Start simple: bootstrap with $e^+e^- \rightarrow \mu^+\mu^-$ (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 $gg \rightarrow t\bar{t}$, $t\bar{t}g$, $t\bar{t}gg$
  - Add missing functionality, fix issues, improve performance, **Iterate**
A complex outer shell – with a CPU-intensive core: the ME

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

Gridpack to generate 100k $gg \rightarrow t\bar{t}gg$ 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

<table>
<thead>
<tr>
<th></th>
<th>$gg \rightarrow t\bar{t}$</th>
<th>$gg \rightarrow t\bar{t}gg$</th>
<th>$gg \rightarrow t\bar{t}ggg$</th>
</tr>
</thead>
<tbody>
<tr>
<td>madevent</td>
<td>13G</td>
<td>470G</td>
<td>11T</td>
</tr>
<tr>
<td>matrix1</td>
<td>3.1G (23%)</td>
<td>450G (96%)</td>
<td>11T (~99%)</td>
</tr>
</tbody>
</table>


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/CPUs, but not good for production (RAMBO algorithm is inefficient)
  - *The results I present in this talk come from this framework*

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

- Easier to code for GPU SIMT than for CPU SIMD: *CUDA code was faster to prototype*

  - CUDA (GPU) implementation
    - For SIMT, event loop is “orthogonal”: one thread = one event *(GPU thread ID ↔ event ID)*
    - For SIMT, SOA memory layouts are beneficial (coalesced access), but not strictly essential

  - C++ (CPU) implementation
    - For SIMD, event loop must be the innermost loop (e.g. invert helicity and event loops)
    - For SIMD, SOA memory layouts in the computational kernel are essential

- To be efficient, *CUDA needs $O(10^k)-O(1M)$ events in parallel* – much more than C++!
  - CUDA: lockstep within each warp (32 threads) + many warps in parallel to fill the GPU
  - C++: lockstep within a vector register (2-8 doubles) + multi-threading or multi-processing

"#EVENTS IN PARALLEL per iteration
(#Threads Per Block * #Blocks)"
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 \( \text{number of MEs} / (\text{time for ME calculation + ME copy}) \)

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

<table>
<thead>
<tr>
<th>Implementation</th>
<th>MEs / second Double</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-core MadEvent Fortran scalar</td>
<td>1.50E6 (x1.15)</td>
</tr>
<tr>
<td>1-core Standalone C++ scalar</td>
<td>1.31E6 (x1.00)</td>
</tr>
<tr>
<td>1-core Standalone C++ 128-bit SSE4.2 (x2 doubles)</td>
<td>2.52E6 (x1.9)</td>
</tr>
<tr>
<td>1-core Standalone C++ 256-bit AVX2 (x4 doubles)</td>
<td>4.58E6 (x3.5)</td>
</tr>
<tr>
<td>1-core Standalone C++ “256-bit” AVX512 (x4 doubles)</td>
<td>4.91E6 (x3.7)</td>
</tr>
<tr>
<td>1-core Standalone C++ 512-bit AVX512 (x8 doubles)</td>
<td>3.74E6 (x2.9)</td>
</tr>
</tbody>
</table>

### Table: Memory symbols

<table>
<thead>
<tr>
<th># Symbols in template</th>
<th>SSE4.2 (x8mm)</th>
<th>AVX2 (y8mm)</th>
<th>AVX512 (z8mm)</th>
<th>AVX512 (x8mm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build type</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar</td>
<td>614</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SSE4.2</td>
<td>3274</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AVX2</td>
<td>0</td>
<td>2746</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256-bit AVX512</td>
<td>0</td>
<td>2572</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>512-bit AVX512</td>
<td>0</td>
<td>1127</td>
<td>205</td>
<td>2045</td>
</tr>
</tbody>
</table>

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

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

---

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

vCHEP – 19 May 2021
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...

**MC Physics Event Generator Software:**
the application

**Research in Theoretical Physics:**
the foundation

**LHC final states:**
- V (W or Z boson) + jets, di-boson, t\(\bar{t}\)bar,
  - single top, tt\(\bar{t}\), multi-jet, gamma + jets...

**Parton distribution functions:**
- LHAPDF...

**Physics precision:**
- LO, NLO, NNLO...

**AN EXTREMELY VARIED SOFTWARE (and use case) LANDSCAPE!**

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

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

**Hadronization and Parton Showers:**
- Pythia, Herwig, Ariadne...
Data-parallel paradigms (GPUs and vectorization)

Generators lend themselves naturally to exploiting event-level parallelism via data-parallel paradigms**
- **SPMD**: Single Program Multiple Data (GPU accelerators)
- **SIMD**: Single Instruction Multiple Data (CPU vectorization: AVX...)

*The computationally intensive part, the matrix element \( f(\vec{x}_i) \), is the same function for all events \( i \) (in a given category of events)*

- Unlike detector simulation (where if/then branches are frequent and lead to thread divergence on GPUs)

Potential interest of GPUs
- Faster (cheaper?) than on CPUs
- Exploit GPU-based HPcs

WIP for MG5aMC on GPUs (planned WG talk) – see next slide

---

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

**This simple event-level parallelism can also be used as the basis for task-parallel approaches (multi-threading or multi-processing)**
1 – Binned fit of a parameter $\theta$

**Event-by-event sensitivities $\gamma_i$: MC weight derivatives**

Bin-by-bin model prediction $n_k(\theta)$

$$n_k(\theta) = \sum_{i=1}^{N_{\text{bin}}} w_i(\theta) = \sum_{i=1}^{N_{\text{bin}}} \sum_{j=1}^{N_{\text{evt}}} w_{ij}(\theta) \cdot n_{ij} = n_k(\theta) + h_k$$

Define the event-by-event sensitivity $\gamma_i$ to $\theta$ as the derivative with respect to $\theta$ of the MC weight $w_i$.

$$\gamma_i = \frac{1}{w_i} \frac{\partial w_i}{\partial \theta} = \frac{\partial n_k}{\partial \theta}$$

(normalized by $1/w_i$, but $w_i|_{\theta_{\text{ref}}} = 1$ at the reference $\theta_{\text{ref}}$)

The bin-by-bin sensitivity to $\theta$ in bin $k$ is the average in bin $k$ of the event-by-event sensitivity $\gamma_i$ to $\theta$

$$\left( \frac{1}{n_k} \frac{\partial n_k}{\partial \theta} \right)_{\theta_{\text{fit}}} = \frac{1}{n_k} \sum_{i=1}^{N_{\text{bin}}} \gamma_i = \frac{1}{w_i} \frac{\partial w_i}{\partial \theta}$$

https://zenodo.org/record/3715951

http://dx.doi.org/10.1051/epjconf/202024506038

---

1 – Binned fit of a parameter $\theta$

**Ideal case: partition by the evt-by-evt sensitivity $\gamma_i$**

Information $I_\theta$ in terms of average bin-by-bin sensitivities:

$$I_\theta = \sum_{k=1}^{K} n_k \left( \frac{1}{n_k} \frac{\partial n_k}{\partial \theta} \right)^2 = \sum_{k=1}^{K} n_k \langle \gamma_i \rangle_k^2$$

There is an information gain in partitioning two events $i_1$ and $i_2$ in two 1-event bins rather than one 2-event bin if their sensitivities $\gamma_{i_1}$ and $\gamma_{i_2}$ are different

$$\Delta I_\theta = \gamma_{i_1}^2 + \gamma_{i_2}^2 - 2 \frac{\gamma_{i_1} \gamma_{i_2}}{2} = \frac{1}{2} (\gamma_{i_1} - \gamma_{i_2})^2$$

**Goal of a distribution fit: partition events by their different MC-truth event-by-event sensitivities $\gamma_i$ to $\theta$**

How to achieve this in practice: next two slides (WDR)

Use $I_\theta^\text{max}$ to compute FIP: following two slides

**Knowing one’s limits: maximum achievable information with an ideal detector**

- Ideal acceptance, select all signal events $S_{\text{sel}} = S_{\text{tot}}$
- Ideal resolution, measured $\gamma_i$ is that from MC truth (implies ideal rejection of background events, $\gamma_i = 0$)

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