



# MadGraph5\_aMC@NLO (MG5aMC) as a benchmark for GPUs and vector CPUs

Andrea Valassi (CERN IT)

*HEPscore Workshop, Tuesday 20<sup>th</sup> September 2022*

*<https://indico.cern.ch/event/1170924/contributions/4954511>*

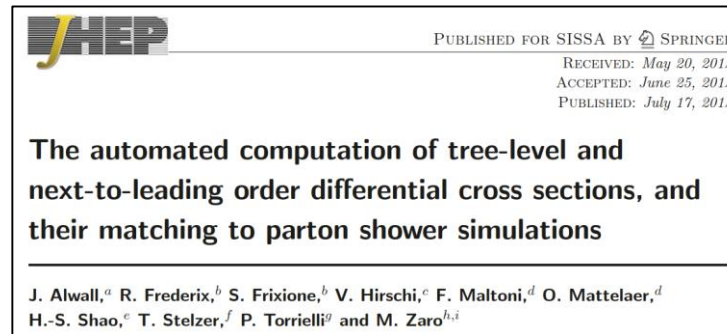
*Many thanks to S. Roiser, O. Mattelaer and the whole madgraph4gpu team!*

*And many thanks to S. Ponce, H. Grasland, S. Lantz, L. Atzori, D. Giordano for useful discussions on SIMD!*



# What is Madgraph5\_aMC@NLO (MG5aMC)?

- A MC physics event generator routinely used by ATLAS, CMS and others
  - One of the many GEN steps in the LHC experiment production workflows



- A highly flexible software application supporting many physics scenarios
  - The relevant code for a given collision process is auto-generated

# What is madgraph4gpu?


- An ongoing project (since 2020) to speed up MG5aMC and port it to GPUs
  - Specifically: *speed up "matrix elements" (ME) calculations: scattering amplitudes*
  - No production release for the experiments yet, but hopefully soon...

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<sup>1,\*</sup>, Stefan Roiser<sup>1</sup>, Olivier Mattelaer<sup>2</sup>, and Stephan Hageboeck<sup>1</sup>

<sup>1</sup>CERN, IT-SC group, Geneva, Switzerland  
<sup>2</sup>Université Catholique de Louvain, Belgium

 5

Developments in software performance and portability for Madgraph5\_aMC@NLO

Taylor Childers  
Walter Hopkins  
Nathan Nichols  
Argonne

Laurence Field  
Stephan Hageboeck  
Stefan Roiser  
David Smith  
Andrea Valassi  
CERN

Olivier Mattelaer  
UCL  
Université catholique de Louvain

ICHEP, Bologna, 8<sup>th</sup> July 2022  
<https://agenda.infn.it/event/28874/contributions/169193>

- The software behind the HEP-workloads container described in this talk!
  - I will only describe here the implementation in CUDA/C++ (but others exist)
- Last status update was at ICHEP (July), next one will be at ACAT (October)
  - New results in this talk with respect to the ICHEP talk include:
    - C++/CPU performance with several CPU processes in parallel
    - C++/CPU performance on Intel Gold 6130 using gcc11 (and with many CPU processes)
    - CUDA/GPU performance with several CPU processes in parallel

# Why a HEP-workloads container based on madgraph4gpu?

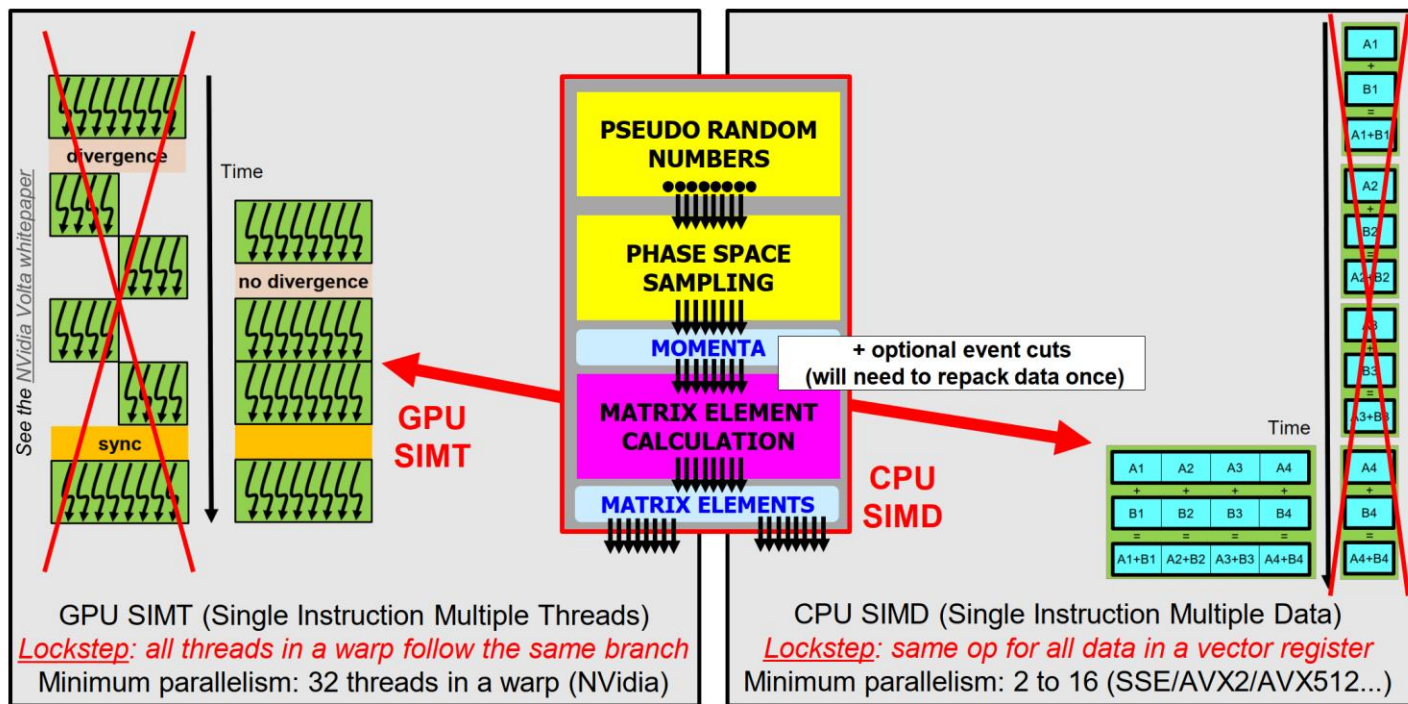
- Interesting for future *HEPscore extensions to GPUs*
  - Potentially a future GPU workload consuming significant Grid resources?
- Interesting for future *HEPscore extensions for vectorisation on CPUs*
  - Huge (up to x16) overall performance differences between no-SIMD and SIMD
    - The current HEPscore workloads are largely insensitive to SIMD (i.e. exploit it poorly)
  - Benchmarking is a multi-dimensional problem: AVX512 support, FMA units etc...
- Interesting for future *HEPscore extensions to heterogeneous processing*
  - Work in progress on understanding how to keep both the CPU and the GPU busy
- With one essential perk: *cross-platform reproducibility of results*
  - The exact same results can be obtained on GPUs, CPUs or CPUs+GPUs

# Why MG5aMC on GPUs and vector CPUs?

## An ideal fit for (event-level) *data parallelism!*


### MG5aMC computational anatomy and data parallelism strategy

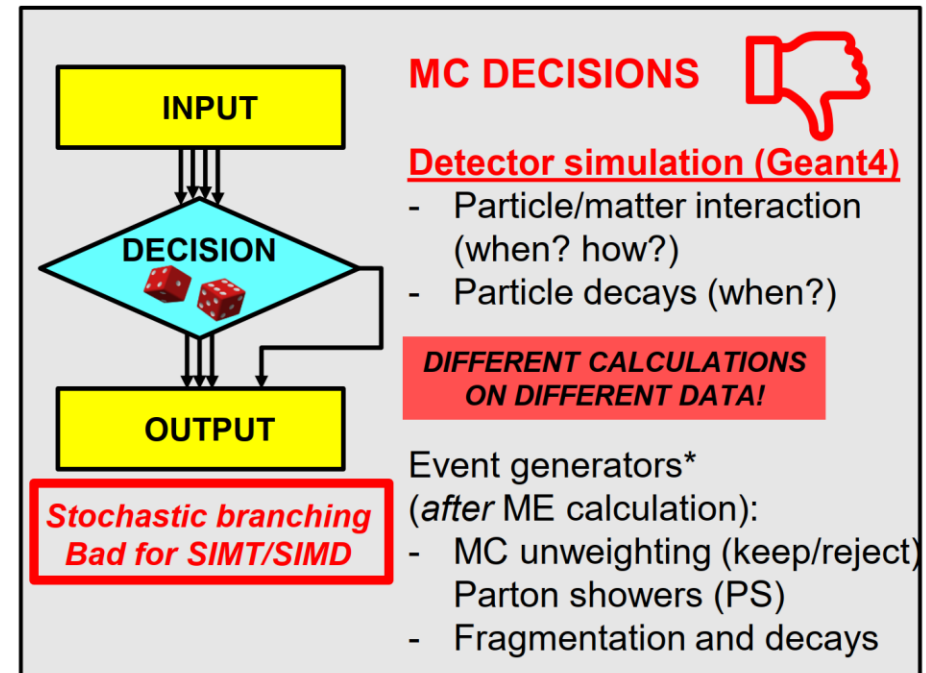
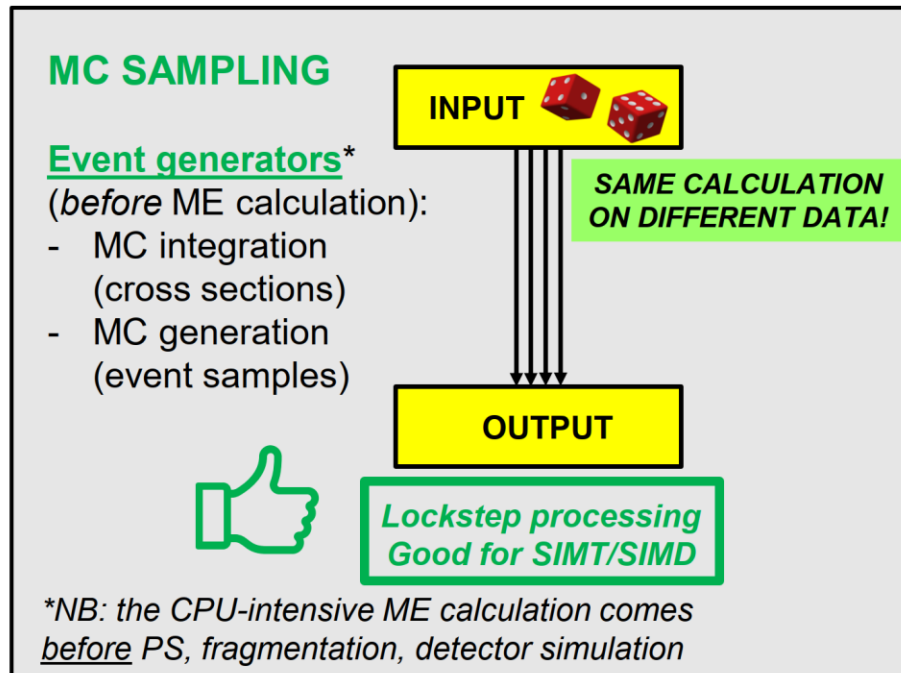
- In MC generators, the same function is used to compute the Matrix Element for many different events
  - *ANY matrix element generator is a good fit for lockstep processing on GPUs (SIMT) and vector CPUs (SIMD)*
  - Data parallelism strategy in madgraph4gpu is event-level parallelism (many events =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



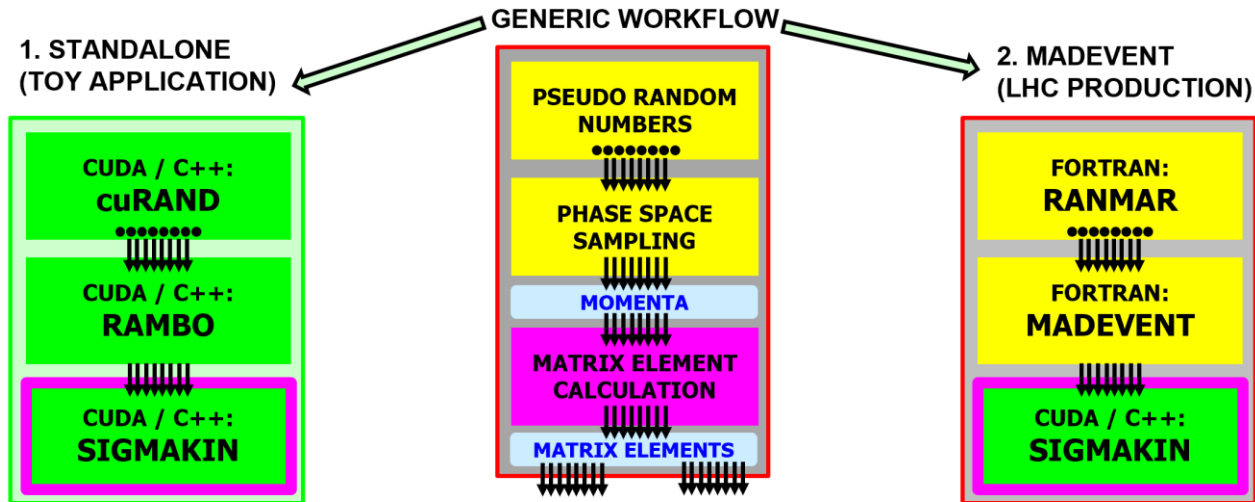
# Most other HEP workloads are not so lucky

MC event generators are a great fit for GPUs and vector CPUs!

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



# "Standalone" and "MadEvent" applications



- We are working on two applications in parallel
  - 1. Standalone: used since 2020 to optimize the ME calculation (all CUDA/C++)
  - 2. MadEvent: current main focus, future production version for the experiments
    - Inject the new/faster CUDA/C++ ME calculation into the existing Fortran framework
- *The current HEP-workloads container (v0.6) is based on the standalone app*
  - Eventually, the MadEvent app will also be integrated into HEP-workloads

# A few details on the v0.6 container

- It includes different software builds covering the combinations of many options:
  - Four physics processes:  $e^+e^- \rightarrow \mu^+\mu^-$ ,  $gg \rightarrow t\bar{t}$ ,  $gg \rightarrow t\bar{t}g$ ,  $gg \rightarrow t\bar{t}gg$
  - Two floating point precisions: double and single (float)
  - One CUDA build for GPU and five C++ builds for CPU with different SIMD scenarios
    - “none”, “sse4”, “avx2”, “512y” (AVX512, 256-bit ymm registers), “512z” (AVX512, 512-bit zmm)
    - For the same process and precision, all six builds give the same physics results
  - (Experimental) Two builds with and without “aggressive inlining” of C++ functions
- It is highly configurable: it is possible to run only a subset of the tests above
  - And it is possible to change other parameters (e.g. numbers of GPU blocks and threads)
- *Recommendation: run the CPU and GPU benchmarks separately*
  - CPU benchmarking: use \$(nproc) copies of the single-threaded benchmarks (all SIMDs)
  - GPU benchmarking: use a single copy (1 CPU process)
    - A (rough!) different tuning of GPU blocks and threads exists for each physics process
- Many different scores may be produced in each benchmark run
  - *Suggestion: use ggttgg-d-inl0 as the most relevant scores (“cuda” for GPU, “cpp-best” for CPU)*
  - But computing, storing and comparing the different benchmarks is very interesting!



# Are we efficiently exploiting the hardware?

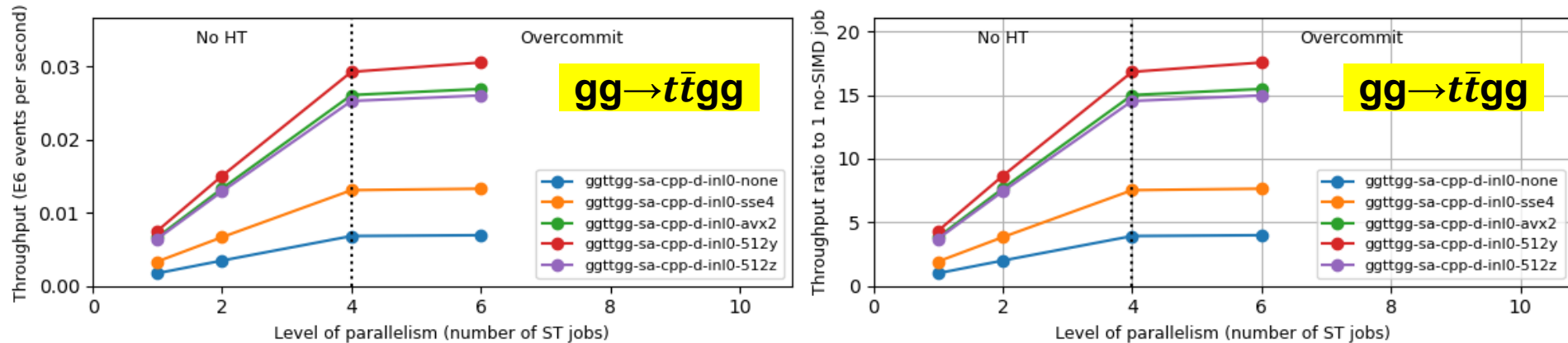
(a.k.a. estimating “realized potential” [Markus Schulz])

- In the following I present the results of a few tests using the v0.6 container
  - Out of the many *independent* dimensions of efficiency I will try to probe only two
  - (...After Vincenzo’s talk yesterday I realize there’s so many other ways to study this!...)
- **Data parallelism (SIMD and SIMT): lockstep processing on multiple data**
  - CPU, AVX512 example: 512-bit “zmm” registers fit 8 (8-byte) doubles or 16 floats (4-byte)
    - Compare no-SIMD to “512z” (AVX512/zmm) SIMD builds, maximum speedup is x8 and x16
  - GPU, NVidia V100 example: every “warp” includes 32 threads
    - The number of GPU “threads per block” is hardcoded in madgraph4gpu to multiples of 32
    - Not shown in next slides: NVidia profiling tools show no thread divergence (branch efficiency 100%)
- **Task parallelism: filling the system with as many hardware threads as possible**
  - CPU: increase #copies (single-threaded CPU processes) in parallel in the bmw-driver
    - throughput plateau around  $\$(nproc)$  i.e. number of physical cores time hyper-threading
  - GPU: increase the GPU “grid size” (#blocks \* #threads-per-block) in low-level options
    - throughput plateau depends (in ways I do not yet understand) on #SM, #threads-per-SM etc...
    - in addition: increase #copies (CPU processes, each with a GPU grid) in the bmw-driver
      - this is useful with a view to future heterogenous processing scenarios...

# SIMD on CPU (C++)

# 4-core Silver 4216, double precision (double)

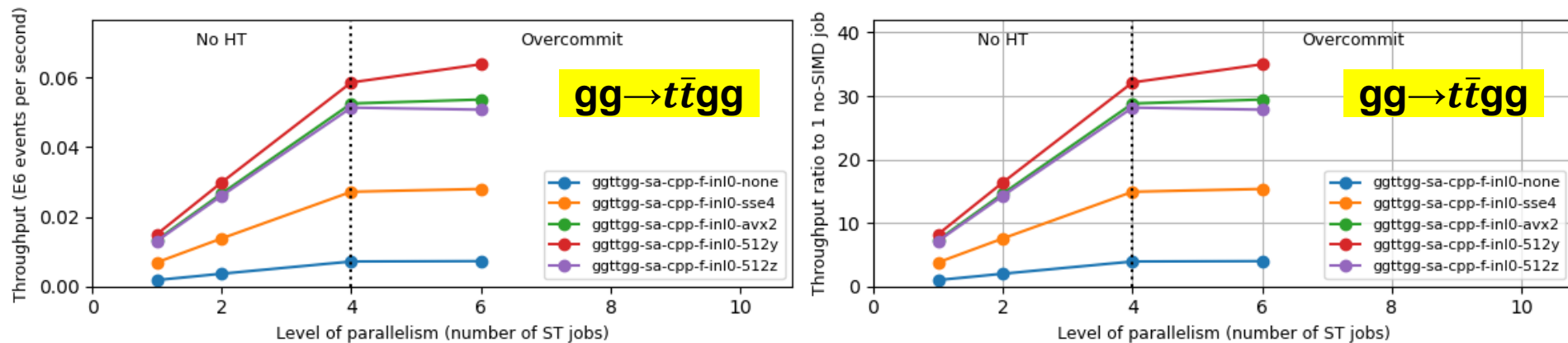
ggttgg check.exe scalability on itscrd70 (1x 4-core 2.1GHz Xeon Silver 4216 without HT) for 10 cycles



- Both plots - with respect to throughput of no-SIMD (blue)
  - SSE4 (yellow) ~ x2 (two 8-byte doubles in a 128-bit register)
  - AVX2 (green) ~ x4 (four 8-byte doubles in a 256-bit register)
  - **512y (red) ~x4 is 10% higher than AVX2 (same register width + AVX512 set): “BEST”**
  - 512z (purple) is worse than AVX2 or 512y (?! will come back to this...)
- Right plot, absolute scale is ratio to one-core no-SIMD: total speedup ~ x16 because
  - ~ x4 from SIMD (512y)
  - ~ x4 from 4 cores

# 4-core Silver 4216, single precision (float)

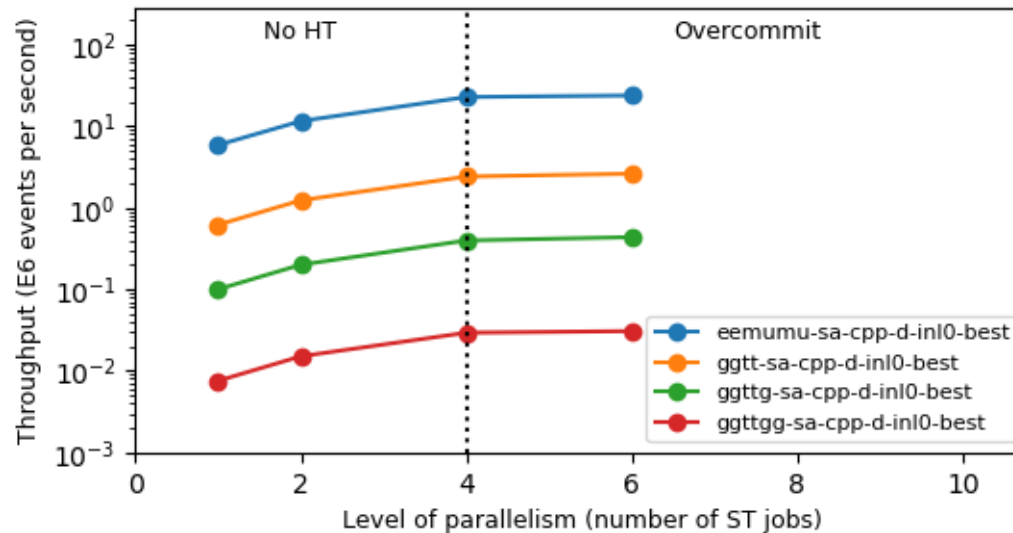
ggttgg check.exe scalability on itsrd70 (1x 4-core 2.1GHz Xeon Silver 4216 without HT) for 10 cycles



- Both plots - with respect to throughput of no-SIMD (blue)
  - SSE4 (yellow) ~ x4 (four 8-byte floats in a 128-bit register)
  - AVX2 (green) ~ x8 (eight 8-byte floats in a 256-bit register)
  - **512y (red) ~ x8 is 10% higher than AVX2 (same register width + AVX512 set): “BEST”**
  - 512z (purple) is worse than AVX2 or 512y (?! will come back to this...)
- Right plot, absolute scale is ratio to one-core no-SIMD: total speedup ~ x32 because
  - ~ x8 from SIMD (512y)
  - ~ x4 from 4 cores

# 4-core Silver 4216, double precision (double)

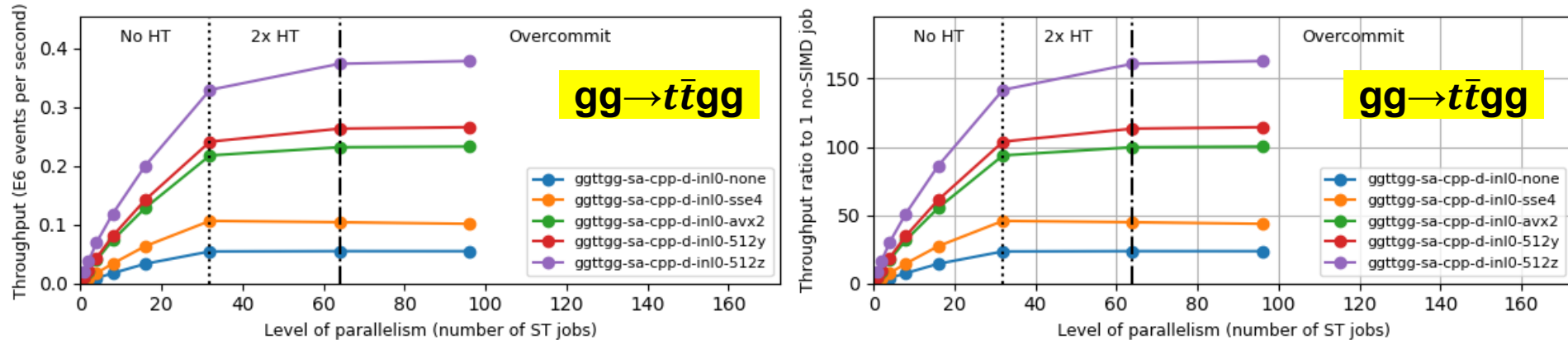
check.exe scalability on itscrd70 (1x 4-core 2.1GHz Xeon Silver 4216 without HT) for 10 cycles



- Compare the four physics processes
  - *one order of magnitude slower throughput on each extra gluon – more Feynman diagrams*
  - and  $gg \rightarrow t\bar{t}$  is even one order of magnitude slower than  $e^+e^- \rightarrow \mu^+\mu^-$
- *Focus on  $gg \rightarrow t\bar{t}gg$  for any realistic (and LHC-relevant) benchmarking*
  - $e^+e^- \rightarrow \mu^+\mu^-$  is limited by memory access
  - $gg \rightarrow t\bar{t}gg$  is limited by computations
  - both on the CPU and later on the GPU

# 2x16-core 2xHT Gold 6130, double precision

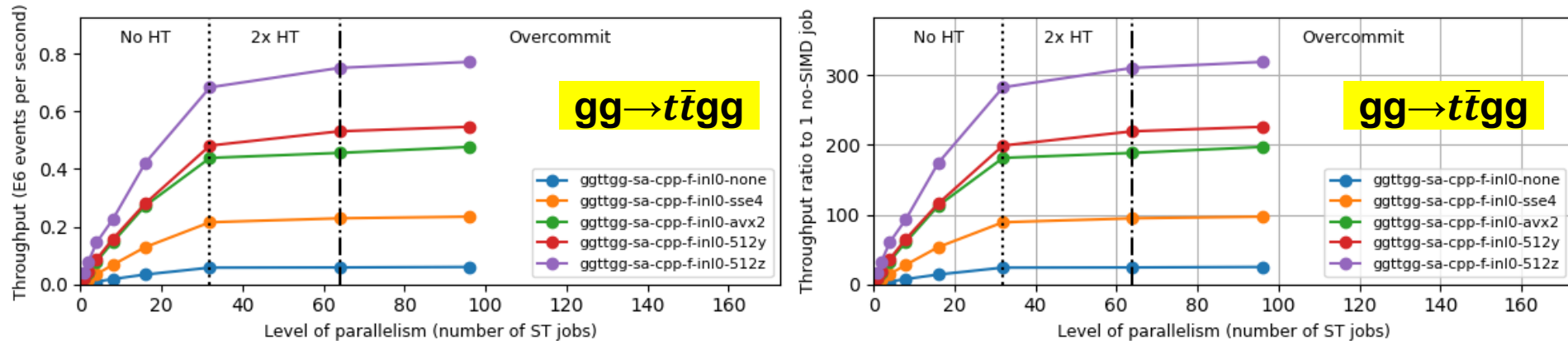
ggttgg check.exe scalability on "bmk6130" (2x 16-core 2.1GHz Xeon Gold 6130 with 2x HT) for 10 cycles



- Both plots - with respect to throughput of no-SIMD (blue)
  - SSE4 (yellow) ~ x2 (two 8-byte doubles in a 128-bit register)
  - AVX2 (green) ~ x4 (four 8-byte doubles in a 256-bit register)
  - 512y (red) ~x4 is 10% higher than AVX2 (same register width + AVX512 set): “BEST”
  - **512z (purple) ~ x6-x8 (eight 8-byte doubles in a 512-bit register): “BEST”**
    - *AVX512/zmm much better on Gold 6130 (two FMA units) than on Silver 4216 (one FMA unit)!*
    - The speedup of 512z is lower than x8 (and decreases with #cores) – clock slowdown?
- Right plot, ratio to one-core no-SIMD: total no-HT speedup ~ x150 because
  - ~ x6 from SIMD (512z)... lower than x8 but not bad!
  - ~ x25 from 32 cores... lower than x32 but not bad!
  - HT also gains a tiny bit more...

# 2x16-core 2xHT Gold 6130, single precision

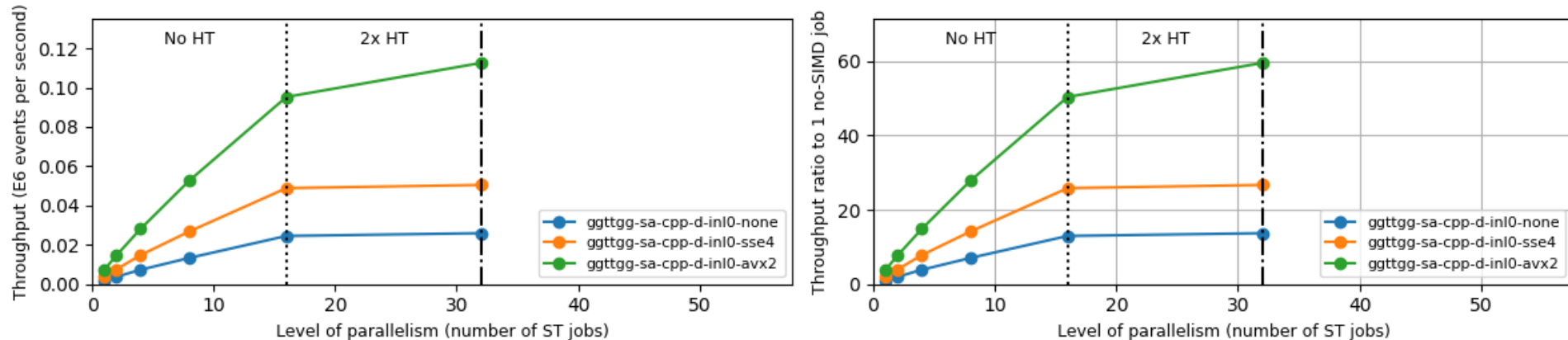
ggttgg check.exe scalability on "bmk6130" (2x 16-core 2.1GHz Xeon Gold 6130 with 2x HT) for 10 cycles



- Both plots - with respect to throughput of no-SIMD (blue)
  - SSE4 (yellow) ~ x4 (four 8-byte floats in a 128-bit register)
  - AVX2 (green) ~ x8 (eight 8-byte floats in a 256-bit register)
  - 512y (red) ~x8 is 10% higher than AVX2 (same register width + AVX512 set): “BEST”
  - **512z (purple) ~ x12-x16 (sixteen 8-byte floats in a 512-bit register): “BEST”**
    - *AVX512/zmm much better on Gold 6130 (two FMA units) than on Silver 4216 (one FMA unit)!*
    - The speedup of 512z is lower than x16 (and decreases with #cores) – clock slowdown?
- Right plot, ratio to one-core no-SIMD: total no-HT speedup ~ x300 because
  - ~ x12 from SIMD (512z)... lower than x16 but not bad!
  - ~ x25 from 32 cores... lower than x32 but not bad!
  - HT also gains a tiny bit more...

# 2x8-core 2xHT Haswell E5-2630, double

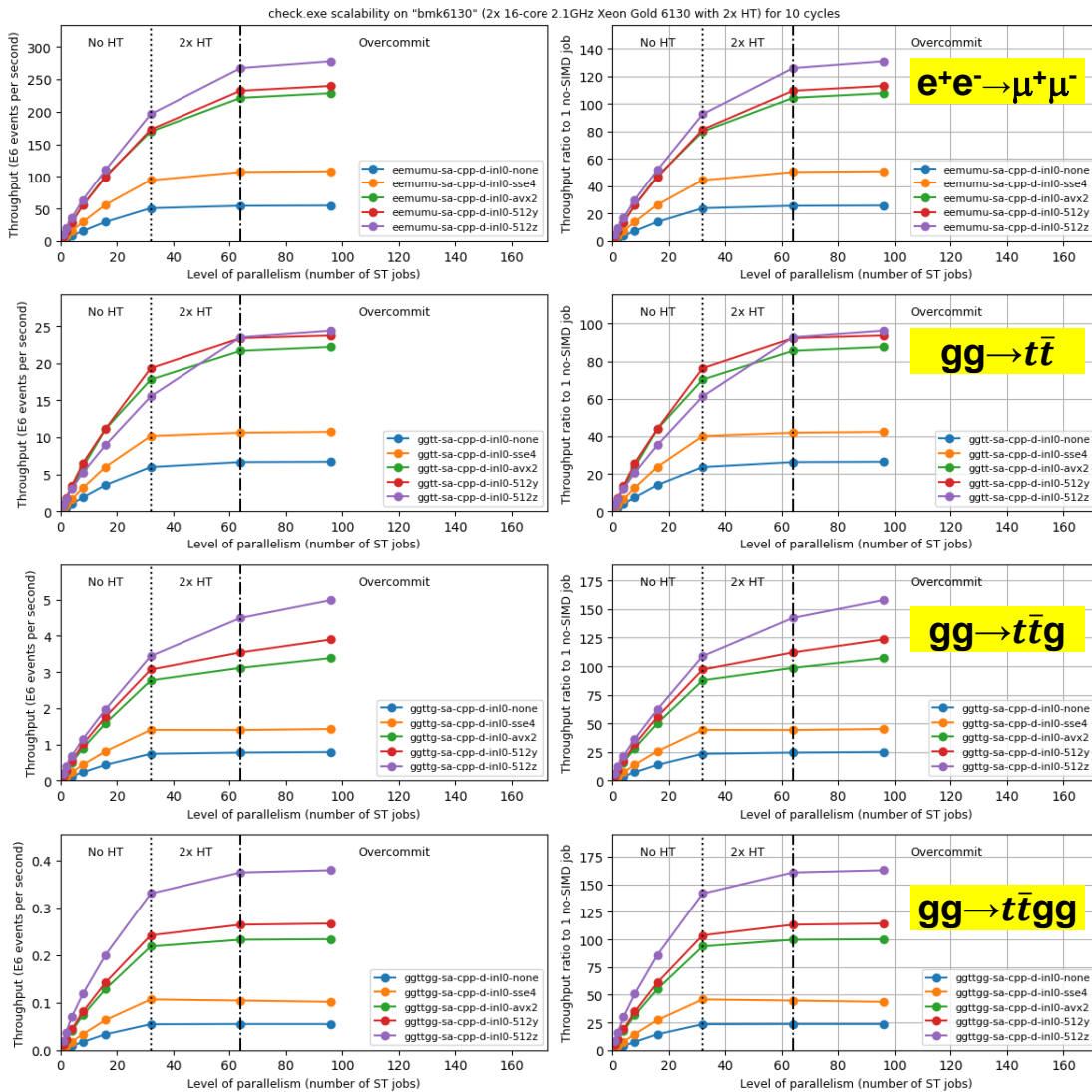
ggttgg check.exe scalability on pmpe04 (2x 8-core 2.4GHz Xeon E5-2630 v3 with 2x HT) for 10 cycles



- *Best is AVX2 – Haswell does not support AVX512*



# Gold 6130, double – compare four processes

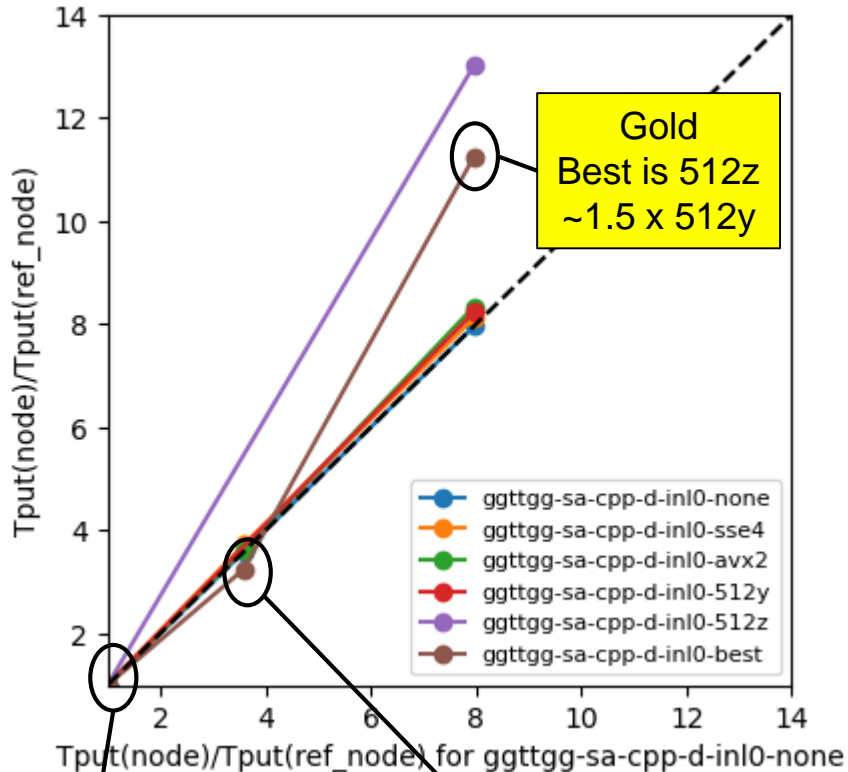


- The relative benefit of 512y and 512z is different in different processes
  - 512y is best for  $gg \rightarrow t\bar{t}$
  - Complex interplay of data access and computation?

• *Even for the same type of GEN workload using the same software, different processes stress different parts of the hardware...*

Note: throughput increase for overcommit in ggttgg is a measurement bug (processes run too few events)

# Correlation plot (à la HEPscore vs HS06)



- Blue curve: no SIMD
  - essentially a straight line
  - three nodes have similar frequencies
  - power per core very similar
- *Focus on brown curve: best SIMD*
  - Silver (reference) has best=512y
  - Haswell has best=avx2
    - No support for AVX512
    - *Haswell ~10% below the diagonal*
  - Gold has best best=512z
    - Two FMA units instead of one
    - *Gold ~50% above the diagonal*

Silver  
(reference node)  
Best is 512y

Haswell  
Best is avx2  
~0.9 x 512y

“A computer system’s performance cannot be characterized by a single number or a single benchmark. [...] Many users (decision makers), however, are looking for a single-number performance characterization. [...] There are no simple answers. *Both the press and the customer, however, must be informed about the danger and the folly of relying on either a single performance number or a single benchmark.*”

Kaivalya M. Dixit, Overview of the SPEC Benchmarks, in J. Gray (Ed.), The Benchmark Handbook for Database and Transaction Systems, 1993.

# GPU (CUDA)

and heterogenous (GPU/CUDA + CPU/Fortran)

# GPU vs CPU throughput with the same output

(all numbers for one single CPU thread)

Implementation ( <i>gg</i> → <i>ttgg</i> )	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	2.39E3 (=1.00)	2.50E3 (x1.05)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	4.59E3 (x1.9)	9.42E3 (x3.6)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	1.06E4 (x4.4)	2.15E4 (x9.0)
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	1.15E4 (x4.8)	2.28E4 (x9.5)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	1.96E4 (x8.2)	4.03E4 (x16.9)

**Intel Gold 6148 CPU (Juwels Cluster HPC)**  
Better AVX512 performance results than on Intel Silver 4216 at CERN  
(Gold 6148 has two FMA units, Silver 4216 has one FMA unit)

Implementation ( <i>gg</i> → <i>ttgg</i> )	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	1.84E3 (=1.00)	1.80E3 (x0.98)
Standalone CUDA Nvidia V100S-PCIe-32GB TFlops*: 7.1 FP64, 14.1 FP32	4.89E5 (x270)	9.27E5 (x500)

**Nvidia V100 GPU + Intel Silver 4216 CPU (CERN)**

Software performance and portability in Madgraph5: [mg@NL0](mailto:mg@NL0)

## Matrix Element (ME) calculation in cudacpp: results

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

Main new results since vCHEP2021:

- **Backport to code generation** (test more complex processes)
  - speedups seen for ee\_mumu now also ~confirmed for gg\_ttgg
  - but GPU speedups decrease a bit (higher "register pressure")
- **Achieve full theoretically possible SIMD speedup on CPUs**
  - x8 double, x16 float from AVX512 on high-end Intel CPUs
- **New features added for MadEvent integration**
  - (this slide shows numbers from the standalone test application; see the final slides for performance numbers within madevent)

- **One full Nvidia V100 GPU vs 1 typical CPU core gives a O(100)-O(1000) speedup**
  - Internally, maximizing the throughput depends on a lot of fine tuning...
  - Note also that float performance is x2 double performance (twice the number of FLOPs)

ICHEP, Bologna, 8 July 2022



7



# O(100) speedup? Don't forget Amdahl!

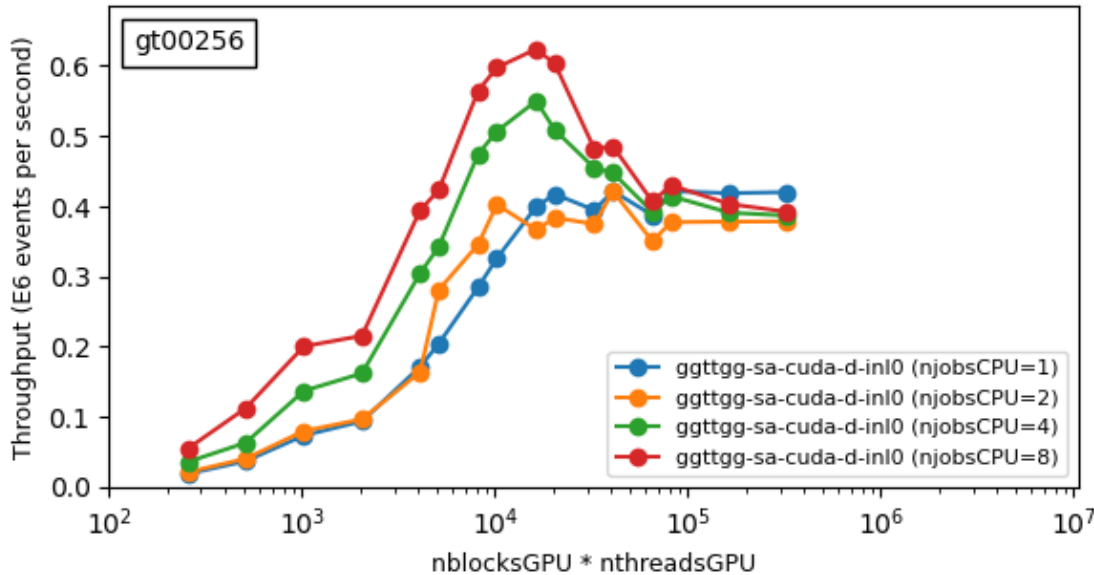
```

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

```

- Current production MG5aMC (MadEvent/Fortran + MEs/Fortran)
  - Matrix Element calculation is 95% of the overall time
- Prototype new MG5aMC (MadEvent/Fortran + MEs/CUDA)
  - Using a GPU speeds up the ME calculation by a factor 100 here (can do even better)
  - *The overall speedup is only a factor 20 - Amdahl's law:  $1 / (1.00 - 0.95)$*
  - *Currently, the bottleneck in our full prototype is still on the CPU (MadEvent)*

# One NVidia V100 on a 4-core Silver CPU



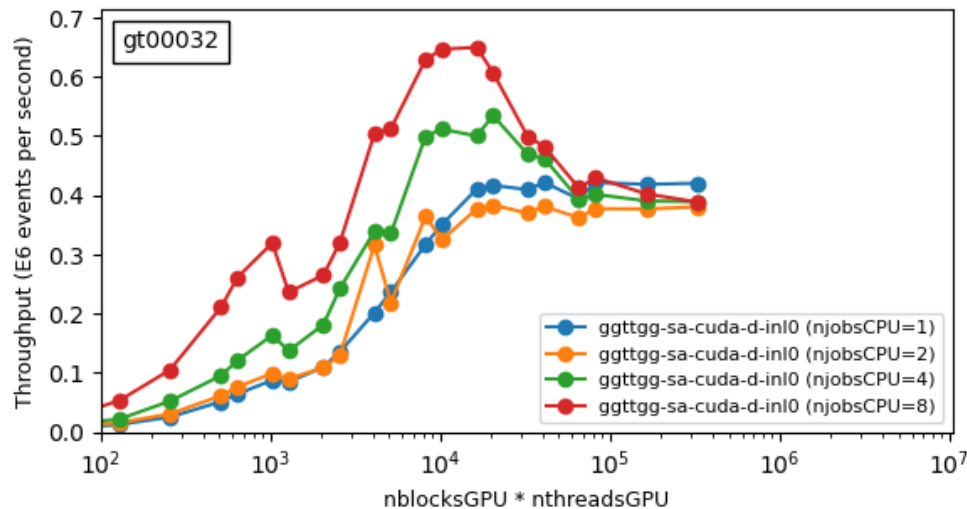
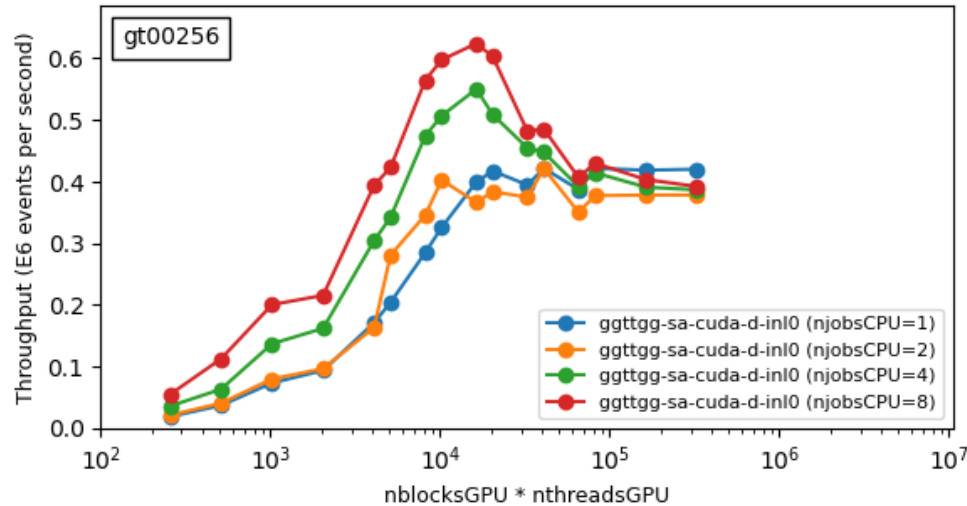
Throughput variation as a function of GPU grid size (#blocks \* #threads)

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

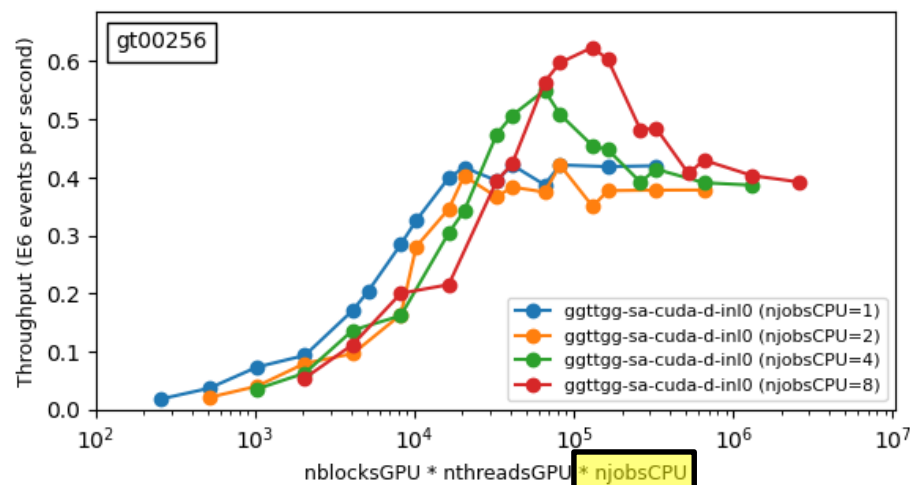
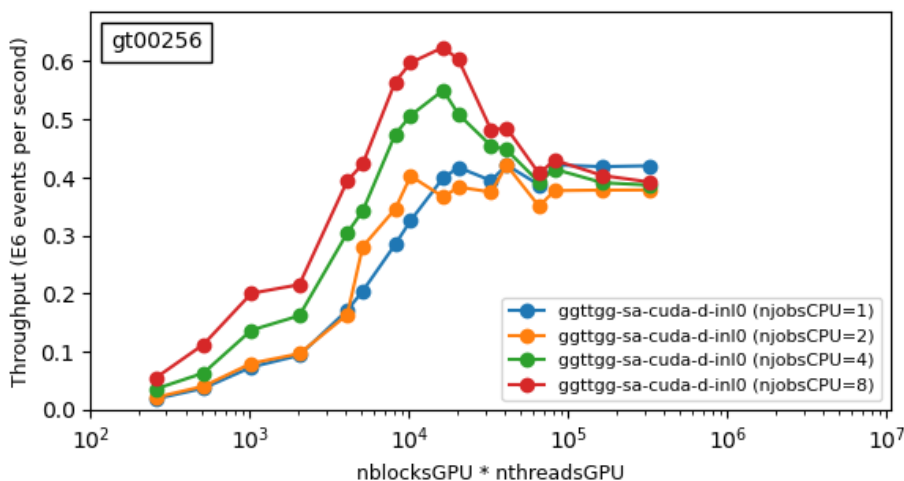
- Blue curve: one single CPU process using the GPU
  - *For  $gg \rightarrow t\bar{t}gg$ , you need at least  $\sim 16k$  events to reach the throughput plateau*
  - The numbers in the table on the previous slides are the throughput  $\sim$  at this plateau
- Yellow, Green, Red curves: 2, 4, 8 CPU processes using the GPU at the same time
  - *Fewer events in each GPU grid are needed if several CPU processes use the GPU*
  - (Why total throughput increases beyond the nCPU=1 plateau is not understood yet!...)

# 256 vs 32 threads per GPU block

- Very similar results, ~no change
- Some fine tuning possible of course



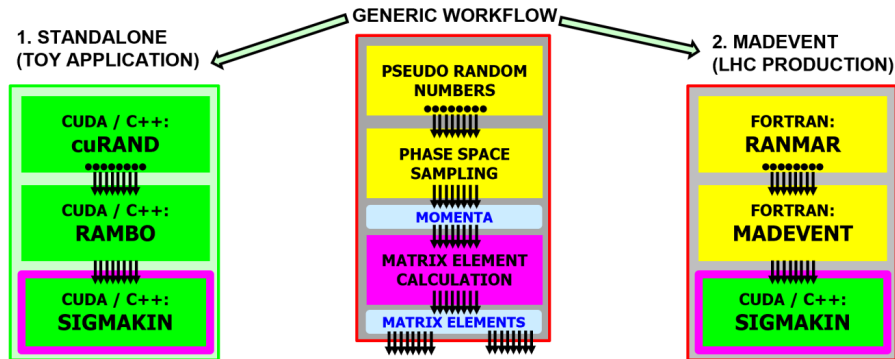
# One NVidia V100 on a 4-core Silver CPU



- The same four curves as before – with the x-axis redefined
  - Total throughput as a function of GPU grid size per CPU process times #processes
- *Using several processes reaches the same throughput faster, with a small overhead*
  - (Does it even allow you to reach higher throughputs? To be understood...)



# Why is this relevant? Heterogeneous apps



Production MadEvent app is heterogeneous: MEs on the GPU, MadEvent on the CPU

- Remember: in our current CPU+GPU offload prototype, the bottleneck is the CPU!
- One likely development in the future:
  - *spread out the MadEvent Fortran processing to several CPU cores in parallel*
  - use smaller GPU grids in each CPU process
  - as per the previous slide, the overall GPU throughput should be the same (or higher?)
- *The message: tuning a heterogenous CPU+GPU system depends on the application!*

# A few thoughts on GPU benchmarks

- *Benchmarking GPUs for a realistic workload is in itself a complex task*
  - “Filling” a GPU depends on non trivial details of the hardware and the application
  - Number of blocks, threads per block, register pressure, occupancy...
- Heterogeneous performance depends even more heavily on the application
- *We are probably better off benchmarking CPUs and GPUs separately?*

# Summary and outlook

- MC matrix element generators are a perfect fit for data parallelism (GPUs, SIMD)
- A HEP-workloads container based on the standalone madgraph4gpu exists
  - And makes it possible to easily collect a lot of useful information
- CPU benchmarking is a complex multi-dimensional problem!
  - Heavily-vectorized workloads stress non obvious CPU features (e.g. how many FMA units?)
- For heterogenous applications, better benchmark GPUs and CPUs separately?
  - Fine tuning the performance of these applications is heavily application-dependent
- Our priority now is to complete the functionality of the full MadEvent-based app
  - This will be the basis of a software release usable by the LHC experiments
- Once that is done, a new HEP-workloads container will be built on top of that
  - Analysis performance using that (possibly with Vincenzo's tool) will be very interesting

# Backup slides

# A few useful links for reference

- Previous talks about the MG5aMC benchmark container
  - HEPiX benchmarking WG, 30 Aug 2022, <https://indico.cern.ch/event/1164106>
  - HEPiX benchmarking WG, 23 Aug 2022, <https://indico.cern.ch/event/1164125>
  - HEPiX benchmarking WG, 05 Nov 2020, <https://indico.cern.ch/event/946409>
- Conference talks and papers about MG5aMC on GPUs and vector CPUs
  - ICHEP, 08 July 2022, <https://agenda.infn.it/event/28874/contributions/169193>
  - vCHEP paper, 23 Aug 2021, <https://doi.org/10.1051/epjconf/202125103045>
  - vCHEP, 15 May 2021, <https://indico.cern.ch/event/948465/contributions/4323568>
  - HSF Workshop, 20 Nov 2020, <https://indico.cern.ch/event/941278/contributions/4101793>
- Software repository of madgraph4gpu
  - Project repository: <https://github.com/madgraph5/madgraph4gpu>

# Matrix element integration in MadEvent: detailed results (GPU)

**Nvidia V100 GPU + Intel Silver 4216 CPU (CERN)**

	mad	mad	mad	sa/brdg
ggttggg	[sec] tot = mad + MEs	[TOT/sec]	[MEs/sec]	[MEs/sec]
nevt/grid	8192	8192	8192	8192
nevt total	90112	90112	90112	256*32*1
FORTTRAN	1286.09 = 62.74 + 1223.35	7.01e+01 (= 1.0)	7.37e+01 (= 1.0)	---
CUDA/8192	77.06 = 64.87 + 12.19	1.17e+03 (x16.7)	7.39e+03 (x100.)	7.48e+03 ← 8k events per GPU grid
nevt/grid				16384
nevt total				512*32*1
CUDA/max				9.33e+03 ← 16k events per GPU grid

**Annotations:**

- TIME MadEvent (scalar)** (points to the 'tot' column)
- 1. REDUCE THIS TO INCREASE SPEEDUP** (points to the 'tot' column)
- ggttgg GPU MEs speedup is lower than eemumu (higher register pressure)** (points to the 'MEs/sec' column)
- 3. SMALLER GPU KERNELS TO INCREASE SPEEDUP** (points to the 'MEs/sec' column)
- 2. INCREASE GPU GRIDS (REDUCE CPU MEMORY) TO INCREASE SPEEDUP** (points to the 'sa/brdg' column)

# 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

The screenshot shows the NVidia NSight Compute interface. At the top, there are two tabs: 'eemumuAV\_cu\_0513\_1108\_b2048\_t256\_j1\_prof2default.ncu-rep' and 'eemumuAV\_cu\_0513\_1107\_b2048\_t256\_j1\_prof2divergent.ncu-rep'. The 'Current' tab is selected, showing details for a launch of '519 - sigmaKin'. The metrics are: Time: 476.93 usecond, Cycles: 592,229, Regs: 128, GPU: NVIDIA Tesla V100S-PCIE-32GB, SM Frequency: 1.24 cycle/nsecond, CC: 7.0, Process: [12414] gched.exe. Below this, a table compares two configurations: 'Current' and 'NO\_DIVERGENCE'. The 'Command line profiler metrics' section is expanded, showing a table of metrics for both configurations. The 'sm\_sass\_average\_branch\_targets\_threads\_uniform.pct [thread]' metric is highlighted with a green box, showing a decrease from 100% in the current configuration to 96.33% in the 'NO\_DIVERGENCE' configuration.

Metric	Current	Change	NO_DIVERGENCE	Change
litex__t_requests_pipe_lsu_mem_global_op_ld.sum [request]	917,504	(+40.00%)	7,339,411	(+40.00%)
launch_registers_per_thread [register/thread]	128	(+6.67%)	96.33	(-3.67%)
sm_sass_average_branch_targets_threads_uniform.pct [thread]	100.00		96.33	(-3.67%)

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



(all numbers for one single CPU thread)

Implementation ( $gg \rightarrow t\bar{t}gg$ )	MEs/second Double	MEs/second Float
1-core Standalone C++ scalar	2.39E3 (=1.00)	2.50E3 (x1.05)
1-core Standalone C++ 128-bit SSE4.2 (x2 doubles, x4 floats)	4.59E3 (x1.9)	9.42E3 (x3.6)
1-core Standalone C++ 256-bit AVX2 (x4 doubles, x8 floats)	1.06E4 (x4.4)	2.15E4 (x9.0)
1-core Standalone C++ "256-bit" AVX512 (x4 doubles, x8 floats)	1.15E4 (x4.8)	2.28E4 (x9.5)
1-core Standalone C++ 512-bit AVX512 (x8 doubles, x16 floats)	1.96E4 (x8.2)	4.03E4 (x16.9)

# Matrix Element (ME) calculation in cudacpp: results

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

Main new results since vCHEP2021:

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

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

**Intel Gold 6148 CPU (Juwels Cluster HPC)**  
Better AVX512/zmm results than on Intel Silver 4216 at CERN  
(Gold 6148 has two FMA units, Silver 4216 has one FMA unit)

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

**NVidia V100 GPU + Intel Silver 4216 CPU (CERN)**

Software performance and portability in Madgraph5\_aMC@NLO

ICHEP, Bologna, 8 July 2022





# ME calculation in PFs: GPU results (Nvidia A100)

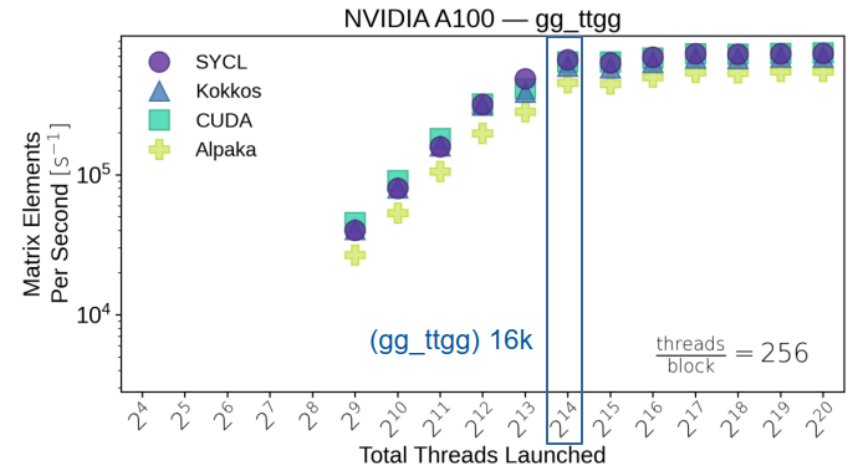
Throughput scaling (threads, blocks) for a complex  $gg \rightarrow t\bar{t}gg$  process

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

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

*NB: focus on `gg_ttgg` which is computationally intensive!*

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



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

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

# Matrix element integration in MadEvent: results

- Functional results (Madevent with Fortran MEs vs CUDA/C++ MEs, using the same random seeds)
  - Cross section calculation: done! (*Same cross section within  $\sim E-14$  relative accuracy*)
  - Unweighted event generation: almost done! (*Same LHE output files, except for missing color/helicity*)
- Performance results  $\Rightarrow$  Total time = Madevent time (scalar, sequential) + ME time (vector, parallel)
  - **The overall speedup is limited by the incompressible scalar component (we need to reduce that too!)**
  - Amdahl's law: if parallel fraction is initially  $p$ , maximum speedup is  $1/(1-p)$

Intel Gold 6148 CPU (Juwels Cluster HPC)

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

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

NVidia V100 GPU +  
Intel Silver 4216 CPU  
(CERN)

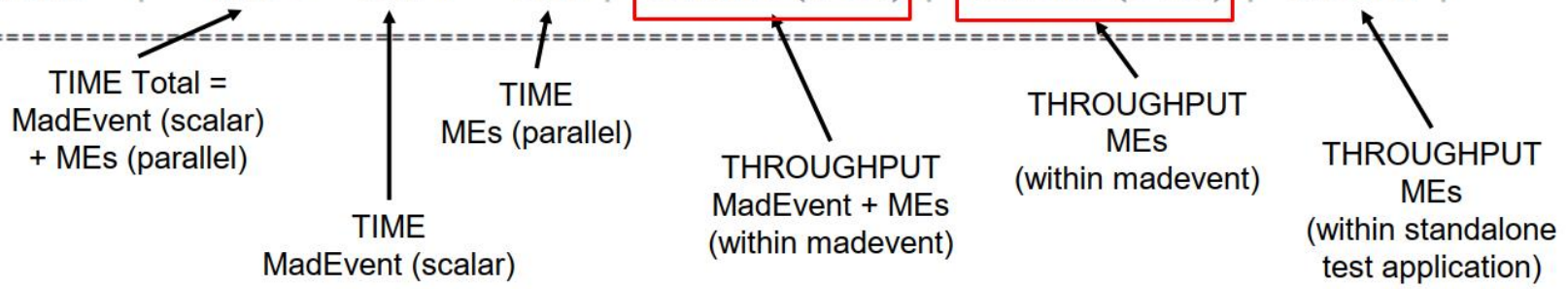
Summary of performance within madevent so far:

- on CPU:  **$\sim x8$**  for MEs alone,  **$\sim x5$**  for madevent+MEs
- on GPU:  **$\sim x100-300$**  for MEs alone,  **$\sim x20$**  for madevent+MEs

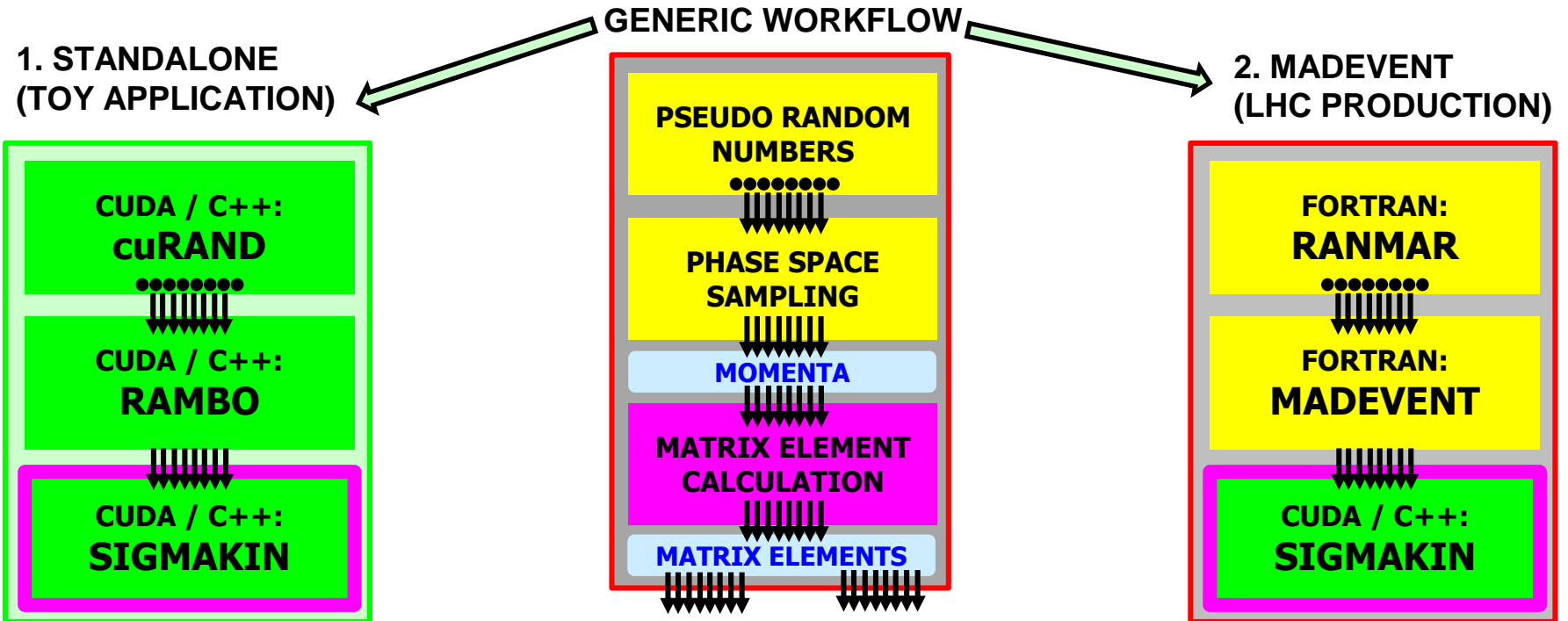
# Matrix element integration in MadEvent: detailed results (CPU)

Intel Gold 6148 CPU (Juwels Cluster HPC)

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



# "Standalone" and "MadEvent" applications



# What is a MC generator? A simplified computational anatomy

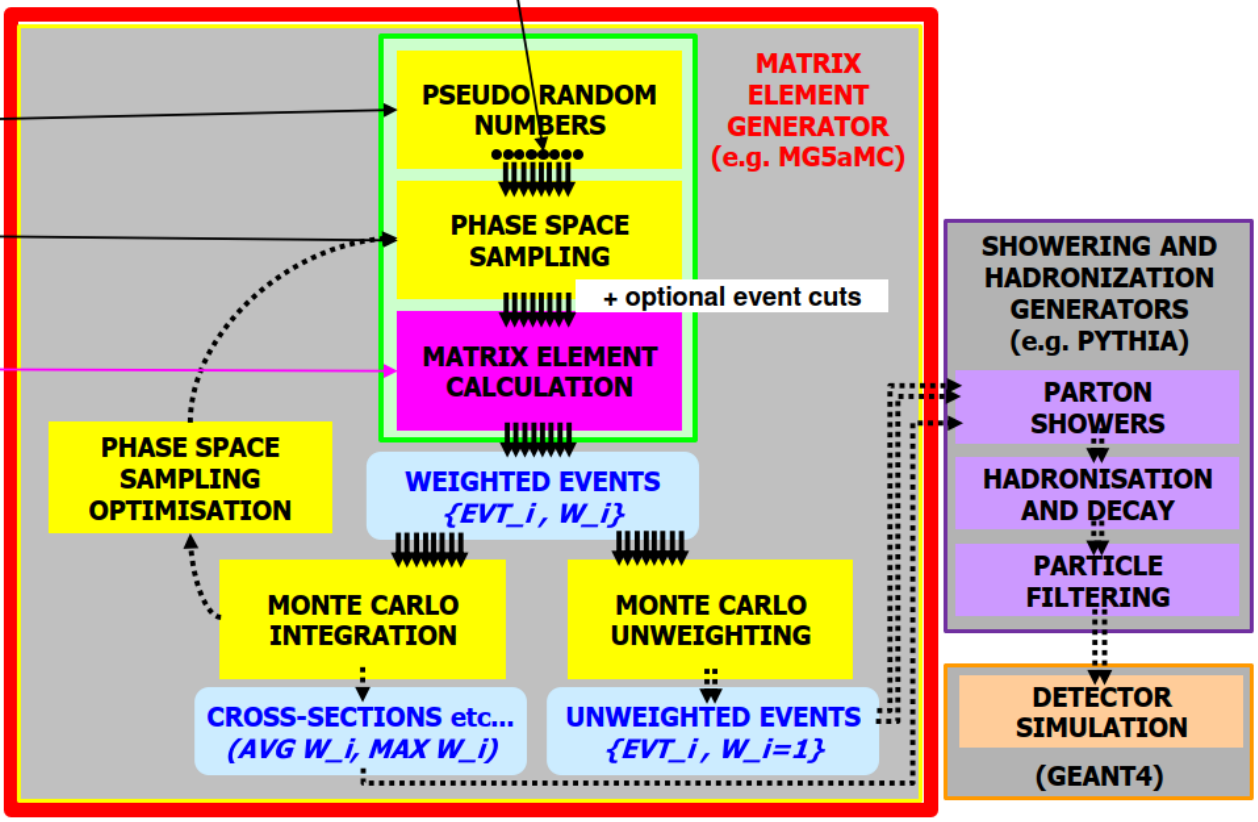
Monte Carlo sampling: randomly generate and process MANY different events (“phase space points”)



This can be parallelized (SIMT/SIMD and multithreading)

For each event:

1. \_\_\_\_\_  
Output: random numbers
2. \_\_\_\_\_  
Input: random numbers  
Output: particle 4-momenta
3. \_\_\_\_\_  
Input: particle 4-momenta  
Output: Matrix Element (ME)  
**CPU BOTTLENECK**

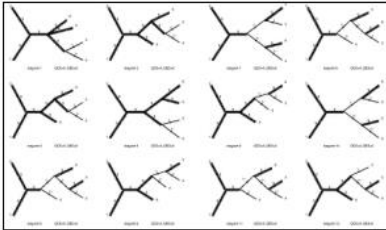


(NB: Matrix Element is an element of the scattering matrix... almost no linear algebra here!)

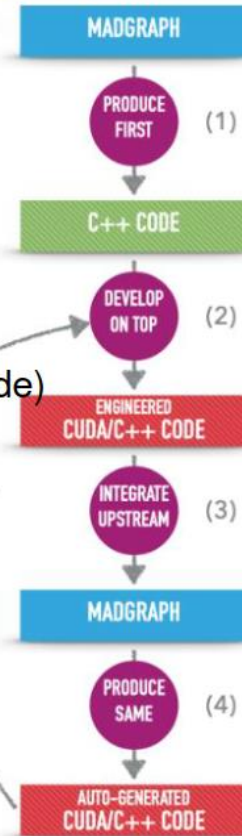


# Code is auto-generated $\Rightarrow$ Iterative development process

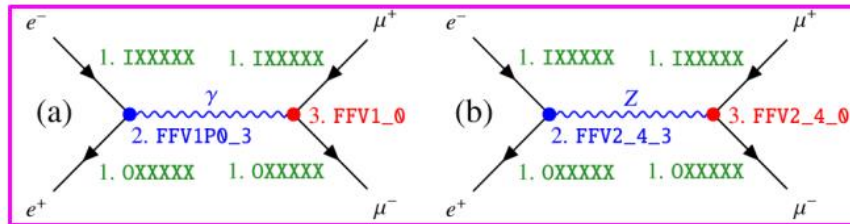
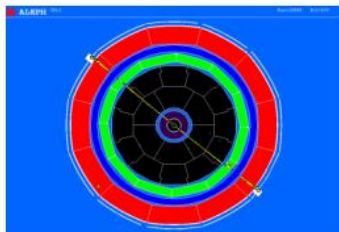
- User chooses process, *MG5aMC determines Feynman diagrams and generates code*
  - Currently Fortran (default), C++, or Python
  - The more particles in the collision, the more Feynman diagrams and the more lines of code



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

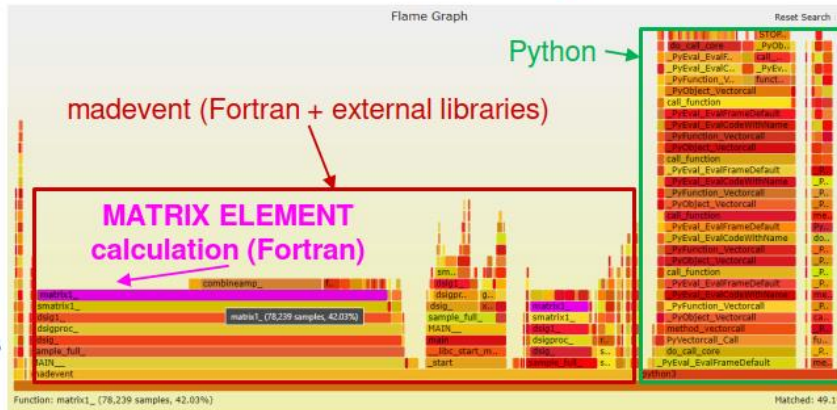


- Goal: modify code-generating code (add CUDA, improve C++ backend)*
  - (1) Start simple: *bootstrap with  $e^+e^- \rightarrow \mu^+\mu^-$*  (two diagrams, few lines of C++ code)
  - (2,3) Add CUDA and improve C++, port upstream to Python meta-code
  - (4) *Generate more complex LHC processes  $gg \rightarrow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

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

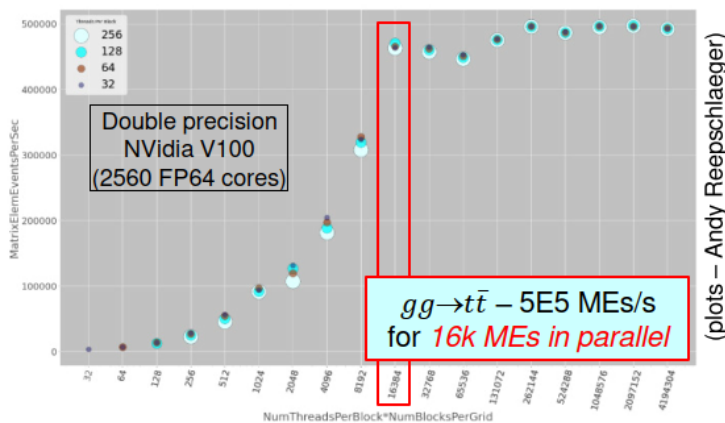
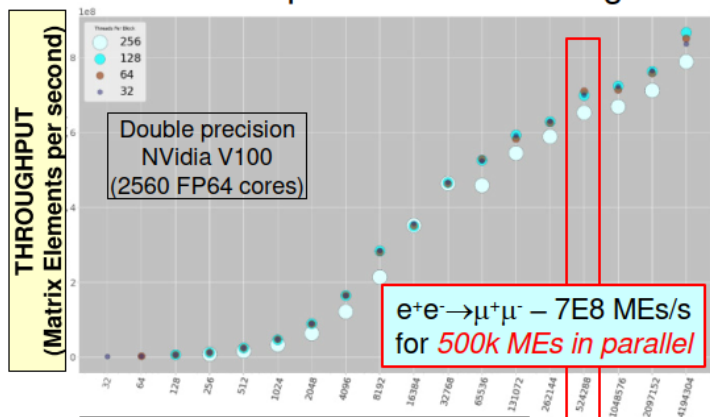
**Our main focus is the ME calculation: develop new CUDA implementation (and speed up existing C++)**

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



# Event-level parallelism in practice – coding and #events

- Easier to code for GPU SIMT than for CPU SIMD: *CUDA code was faster to prototype*
- CUDA (GPU) implementation
  - For SIMT, event loop is “orthogonal”: one thread = one event (*GPU thread ID ↔ event ID*)
  - For SIMT, SOA memory layouts are beneficial (coalesced access), but not strictly essential
- C++ (CPU) implementation
  - For SIMD, event loop must be the innermost loop (e.g. invert helicity and event loops)
  - For SIMD, SOA memory layouts in the computational kernel are essential
- To be efficient, *CUDA needs  $O(10k)$ - $O(1M)$  events in parallel* – much more than C++!
  - CUDA: lockstep within each warp (32 threads) + many warps in parallel to fill the GPU
  - C++: lockstep within a vector register (2-8 doubles) + multi-threading or multi-processing



#EVENTS IN PARALLEL per iteration  
(#Threads Per Block \* #Blocks)





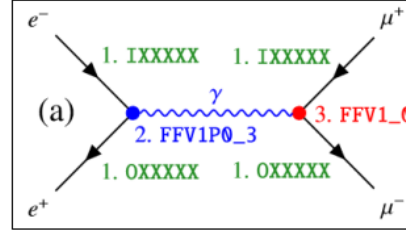
# CUDA/C++: ME code example (complex number scalar/vector)

**Formally the same code for three back-ends** (*cxtype\_sv* represents three types)

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

```

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



FFV1\_0:  
helicity amplitude  
for the  $\gamma\mu^+\mu^-$  vertex  
*Soon to be  
automatically generated*

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

```

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

```

C++ SIMD: gcc / clang
compiler vector extensions
#ifdef __clang__
    typedef fptype fptype_v __attribute__((ext_vector_type(neppV))); // RRRR
#else
    typedef fptype fptype_v __attribute__((vector_size (neppV*sizeof(fptype)))); // RRRR
#endif
    
```



# 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 )



- 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  $\gg$  0.4ms ME copy)
  - We expect that *this will not be an issue for typical LHC collision processes*

