



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!

CERN Openlab Workshop, 16th March 2023

<https://indico.cern.ch/event/1225408/contributions/5243830/>



Outline

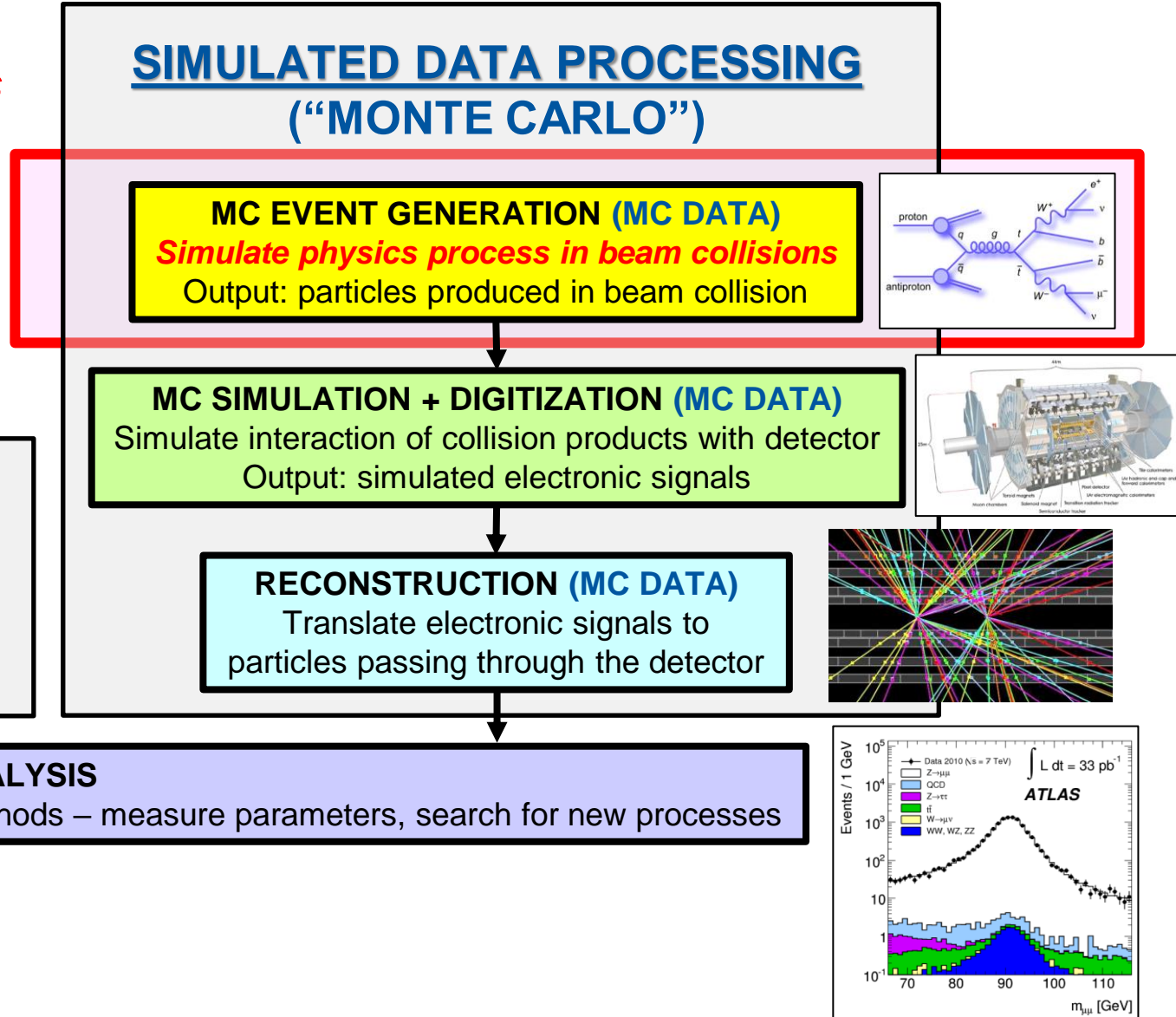
- Intro and motivation
 - What are event generators in High Energy Physics
 - Why is it interesting (and possible!) to speed them up using CPU vectorization or GPUs
- The madgraph4gpu project: a selection of interesting results on Intel CPUs, GPUs, compilers and APIs
- Summary

Event generators: the first step in the HEP simulation chain

Around 10-20 % of LHC computing CPU costs
(hence: important to speed them up!)

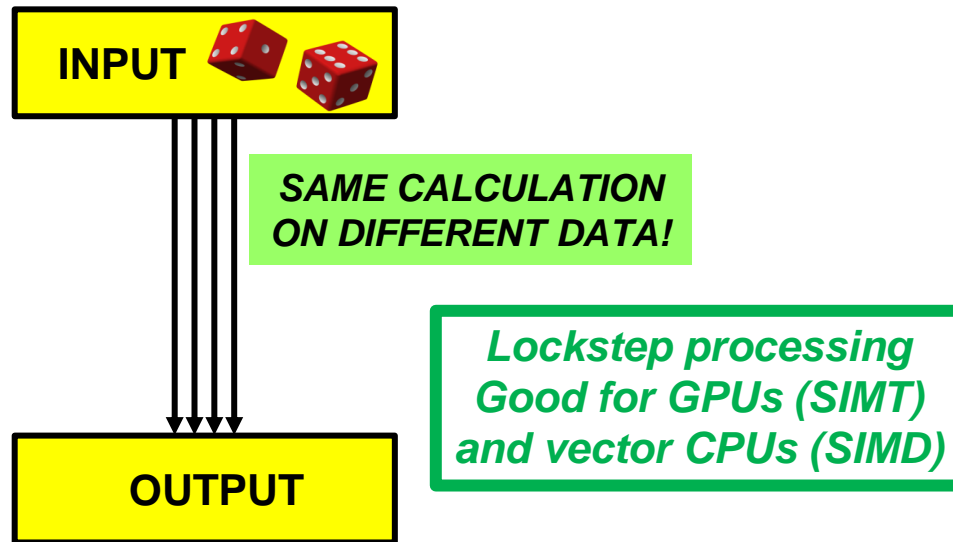
Theoretical physics (Feynman diagrams)

Monte Carlo methods (random numbers)



Event generators: why CPU vectorization and GPUs?

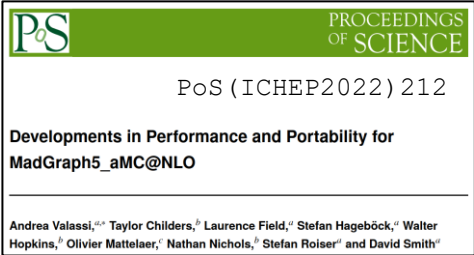
- **CPU vectorization and GPUs are widely available to HEP for processing...**
 - Most of the CPUs in our computing Grid have at least AVX2 SIMD
 - GPUs are becoming more and more available to us especially at HPC centers
- **... but our software, so far, generally underexploits them ☹**
 - Example: Monte Carlo detector simulation has a lot of stochastic branching (makes lockstep processing difficult)
- **Event generators, conversely, are ideal software workflows for SIMD and GPUs!**
 - Monte Carlo sampling of many data points → Data parallelism with near-perfect lockstep processing!



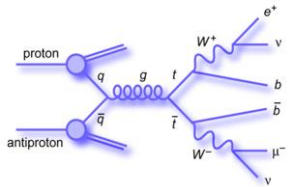
Porting Madgraph to CPU vectorization and GPUs



- Madgraph5_aMC@NLO: one of the workhorses for event generation in ATLAS and CMS!
- The madgraph4gpu project (started Q1 2020): speed up Madgraph5 using GPUs and vector CPUs
 - Code repository, CI tests, issue tracker: <https://github.com/madgraph5/madgraph4gpu>
 - More details: [vCHEP2021](#) paper, [ICHEP2022](#) paper, [CAF2023](#) talk
- **Port to GPUs and SIMD the parallelizable part (“Matrix Element” calculation)**
 - This is the main CPU bottleneck (95% or more) in the current Fortran implementation
 - There is also a non-negligible (up to ~5%) scalar part (new bottleneck due to Amdahl)

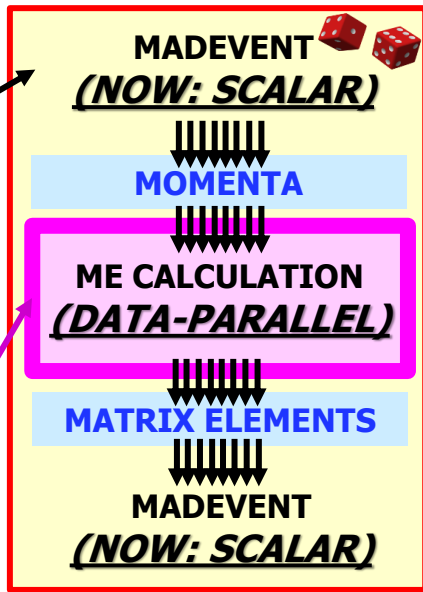


<https://doi.org/10.22323/1.414.0212>



“Madevent” framework (scalar overhead)

“Matrix Element” calculation (parallelizable)
Auto-generated code for each physics process!



CUDACPP vs SYCL implementations (for CPUs and GPUs)

CUDACPP: main implementation



- 95% common code + a few #ifdef's for CUDA vs C++
- GPUs: designed for NVidia (will also add HIP for AMD)
 - Full feature support, e.g. tensor cores, streams, graphs



Intel® AVX512

- CPU SIMD: designed upfront for C++ vectorization
 - gcc and clang compiler vector extensions
- CPU multithreading and heterogeneous modes: WIP
- Compilers: nvcc + gcc, clang, icpx

SYCL (+Kokkos, Alpaka): alternative implementation



- Write code once for many CPU/GPU vendors
- GPUs: support NVidia, AMD and Intel out-of-the-box
 - Limited support for vendor-specific features



- CPU SIMD: vectorization added Jan 2023, WIP
 - `sycl::vec` (also based on clang compiler extensions)
- CPU multithreading: out of the box
- Compiler: dpcpp (aka "icpx -fsycl"?)

CUDACPP is where we add new features first for the integration with existing user applications

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

(CUDACPP implementation on CPUs)

Vectorized C++ on Intel CPUs

C++ compilers: gcc, clang, Intel icpx

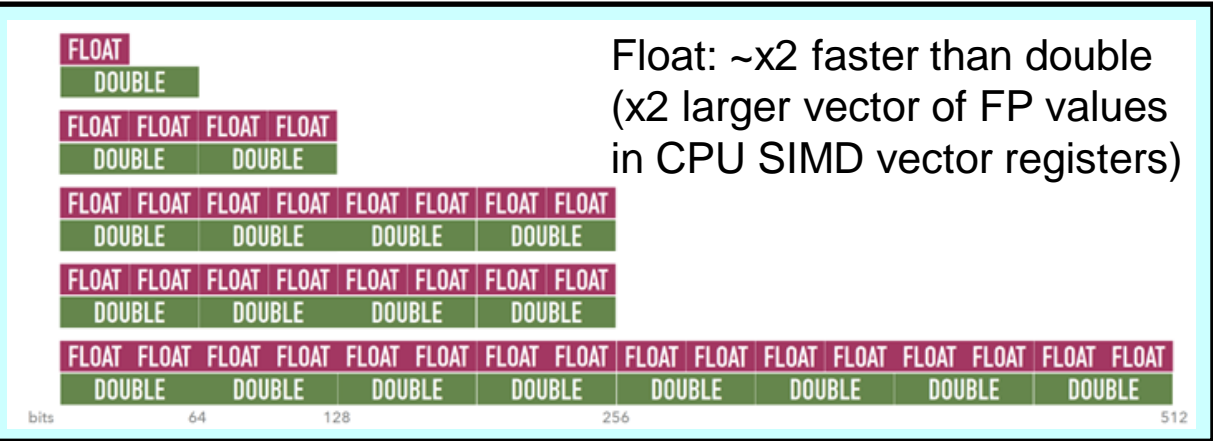
C++ vectorization in CUDACPP: overview

- Implementation is based on **compiler vector extensions (CVEs)**: explicit vectors of floating point types
 - Supported by all of the gcc, clang and (through clang) Intel icpx compilers
 - Powerful but easy to use (no debugging auto-vectorization!), intuitive (they force you to design code for vector types!)

```
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
```

- Routinely build and compare **five vectorization levels** on Intel CPUs (and similar features on AMD or ARM CPUs)

- none** 1xD, 1xF (scalar)
- sse4** 2xD, 4xF (128-bit xmm registers, “nehalem” **SSE4.2** instruction set)
- avx2** 4xD, 8xF (256-bit ymm registers, “haswell” **AVX2** instruction set)
- 512y** 4xD, 8xF (256-bit ymm registers, “skylake-avx512” **AVX512** instruction set)
- 512z** 8xD, 16xF (512-bit zmm registers, “skylake-avx512” **AVX512** instruction set)



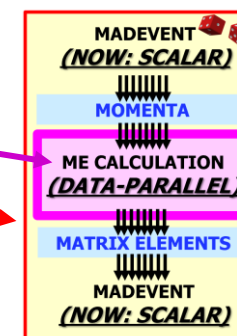
C++ vectorization in CUDACPP: results for $gg \rightarrow t\bar{t}gg$ (one core)

$gg \rightarrow t\bar{t}gg$	MEs precision	ACAT2022		madevent		standalone
		$t_{TOT} = t_{Mad} + t_{MEs}$ [sec] p=95%	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{MEs} [MEs/sec]		
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	

Intel Gold6148 (Juelich)
One single CPU core
gcc11.2 compiler (no inlining)

Float: ~x2 faster than double
(x2 larger vector of FP values
in CPU SIMD vector registers)

Data-parallel component alone (ME calculation):
speedup ~ x8 (double) and x17 (float) over scalar Fortran
We reach the maximum theoretical SIMD speedup for AVX512



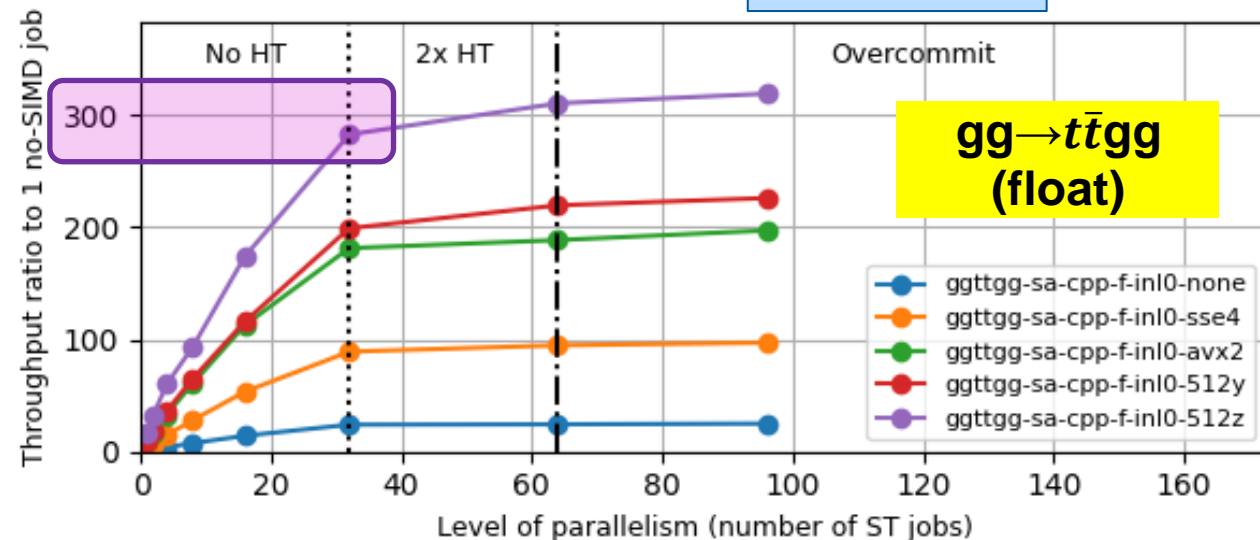
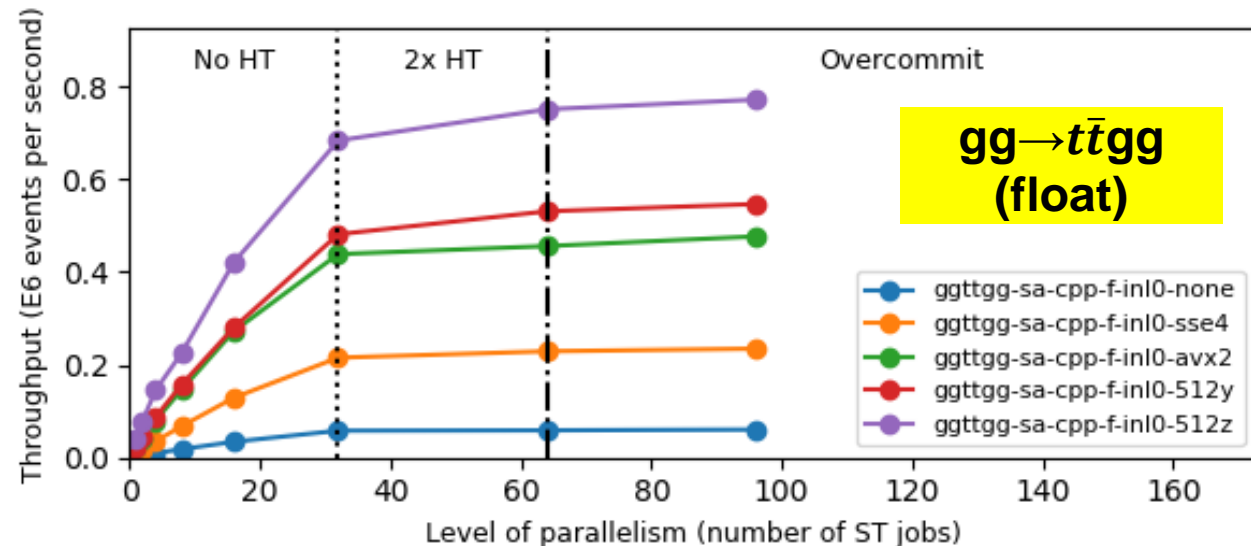
$$\lim_{s \rightarrow \infty} S_{\text{latency}}(s) = \frac{1}{1 - p}$$

Overall: speedup so far ~ x6 (double) and x10 (float) over scalar Fortran
Amdahl's law limit is for $gg \rightarrow t\bar{t}gg$ (2→4 process) is x20 (Fortran MEs are p=95% of the total)

C++ vectorization in CUDACPP: results for $gg \rightarrow t\bar{t}gg$ (many cores)

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

ACAT2022

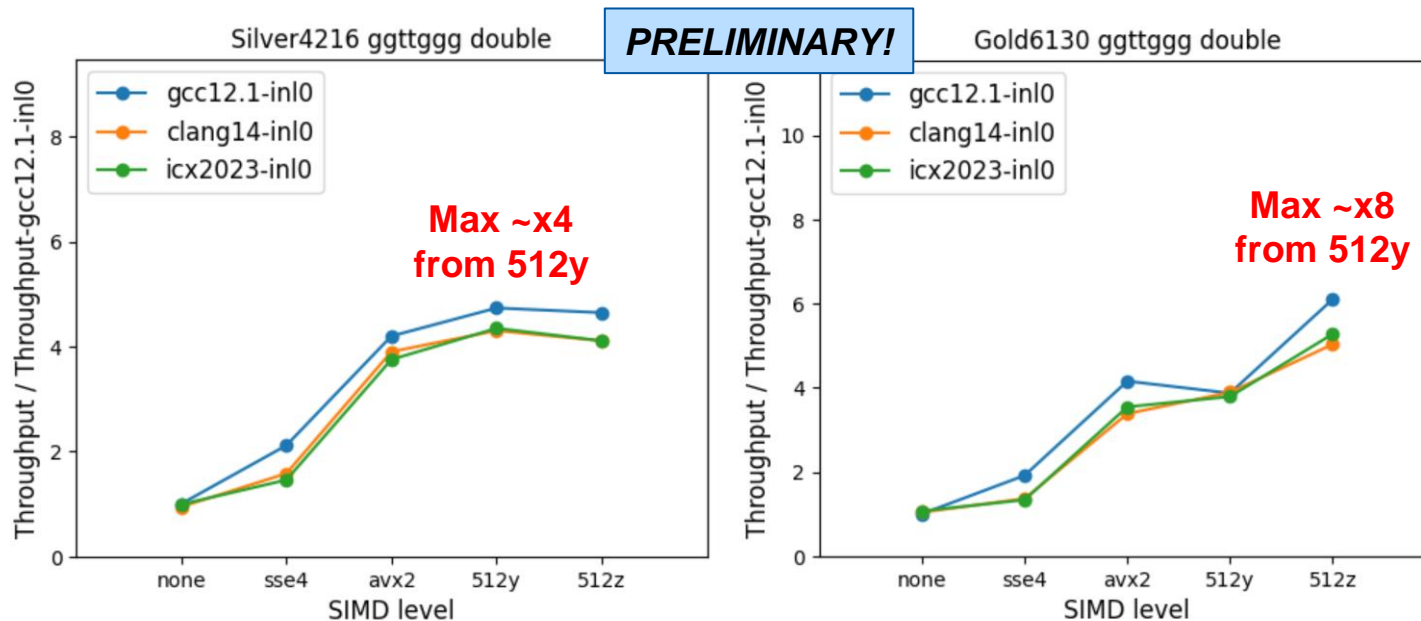


- **Large SIMD speedups are also confirmed when all CPU cores are used**
 - AVX512/zmm speedup of x16 over no-SIMD for a single core slightly decreases to ~x12 on a full node
 - Possibly due to clock slowdown from fully loaded AVX512 processor (to-do for us: further investigate this)
 - *Overall speedup on 32 physical cores (over no-SIMD on 1 core) is around 280 (maximum would be 16x32=512)*
- NB: this is a multi-process approach (many identical processes running the same benchmarking application)
 - These plots were produced using the infrastructure of the *HEP-SCORE benchmarking project (next talk by D. Giordano)*
 - We also have initial results from multi-threading in CUDACPP (using OpenMP), but this is work-in-progress

C++ vectorization in CUDACPP: Intel Silver vs Intel Gold CPUs

- Previous slide was for Intel Gold6148, but results with Silver CPUs not as good: compare Silver4216 and Gold6130
- Intel Gold 6130: max achieved double performance is ~x8 from "512z" (512-bit zmm registers, AVX512 instructions)
 - **There is an advantage using zmm registers on Gold6130 CPU (which has 2 FMA units - like Gold6148)** [1,2]
- Intel Silver 4216: max achieved double performance is ~x4 from "512y" (256-bit ymm registers, AVX512 instructions)
 - **There is no advantage using zmm registers on Silver4216 CPU (which has only 1 FMA unit)** [3]
 - Note that 512y is still ~10% better than avx2 (uses a few additional instructions in the AVX512 instruction set)

Question: would it be possible to specify the number of FMA units in /cpu/procinfo or other O/S properties?...



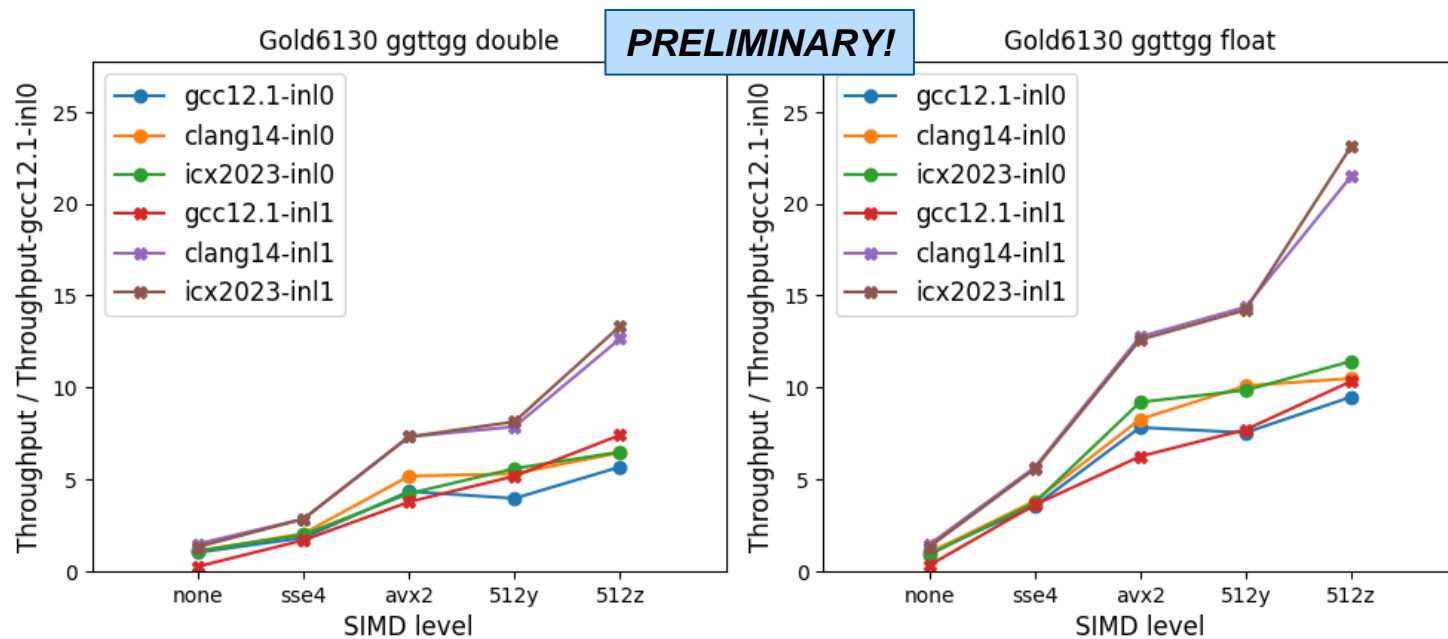
[1] <https://ark.intel.com/content/www/us/en/ark/products/120492/intel-xeon-gold-6130-processor-22m-cache-2-10-ghz.html>

[2] <https://ark.intel.com/content/www/us/en/ark/products/120489/intel-xeon-gold-6148-processor-27-5m-cache-2-40-ghz.html>

[3] <https://ark.intel.com/content/www/us/en/ark/products/193394/intel-xeon-silver-4216-processor-22m-cache-2-10-ghz.html>

C++ vectorization in CUDACPP: gcc vs clang (or Intel icpx)

- In our default implementation *inl0* ("no aggressive inlining"), gcc gives better throughput results than clang or icpx
 - The results in the previous slides were based on this gcc+inl0 baseline
- In CUDACPP we also have another implementation *inl1* ("with aggressive inlining")
 - With gcc, this is worse - but with clang (or icpx), this may give up to a factor 2 or more additional speedups!
 - Disadvantage: build times explode (similarly to Link Time Optimization, from which this implementation was inspired)
 - (On the to-do list for us, understand this better: profile data pipelining? further analysis at assembly level?)
- Whether for inl0 or inl1, **the additional benefits of icpx over clang seem to be very small, if there are any at all**
 - The benefits of icpx2023 over gcc come mainly from its internal use of clang16? (No benefit over standalone clang14)



Note: preliminary results from the SYCL implementation are compatible with those from CUDACPP using the icpx compiler

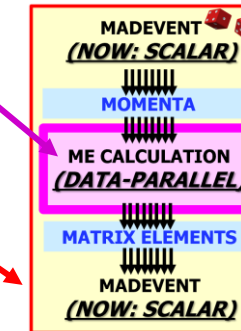
(CUDACPP implementations on Nvidia GPUs)

GPU offload in CUDACPP: example results for $gg \rightarrow t\bar{t}ggg$

CUDA grid size		ACAT2022	madevent		standalone	
			8192	16384		
$gg \rightarrow t\bar{t}ggg$	MES precision	$t_{TOT} = t_{Mad} + t_{MEs}$ [sec] p=99.5%	N_{events}/t_{TOT} [events/sec]	N_{events}/t_{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)

NVidia V100 GPU
Intel Silver 4216 CPU
cuda11.7 + gcc11.2

Float: ~x2 faster than double
(x2 FP FLOPS in V100 GPU)



$$\lim_{s \rightarrow \infty} S_{latency}(s) = \frac{1}{1-p}$$

Data-parallel component alone (ME calculation):
speedup ~ x100 (double) and x220 (float) over one-CPU-core scalar Fortran

Overall: speedup so far ~ x60 (double) and x100 (float) over one-CPU-core scalar Fortran
Amdahl's law limit for $gg \rightarrow t\bar{t}ggg$ (2→5 process) is x200 (Fortran MEs are p=99.5% of the total)

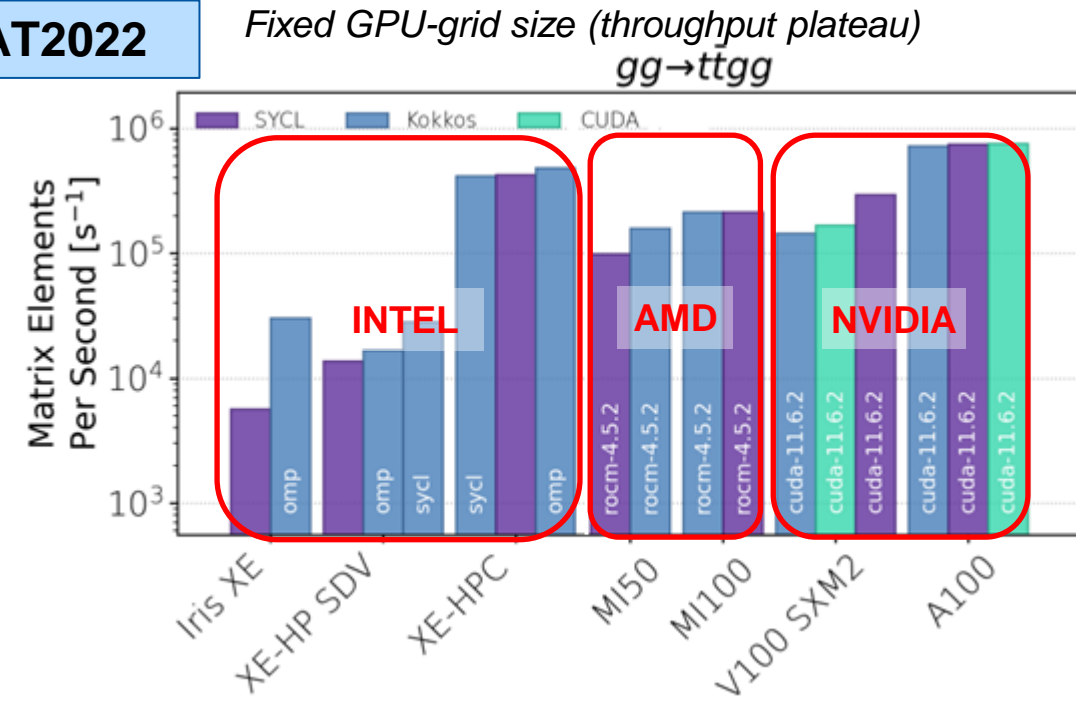
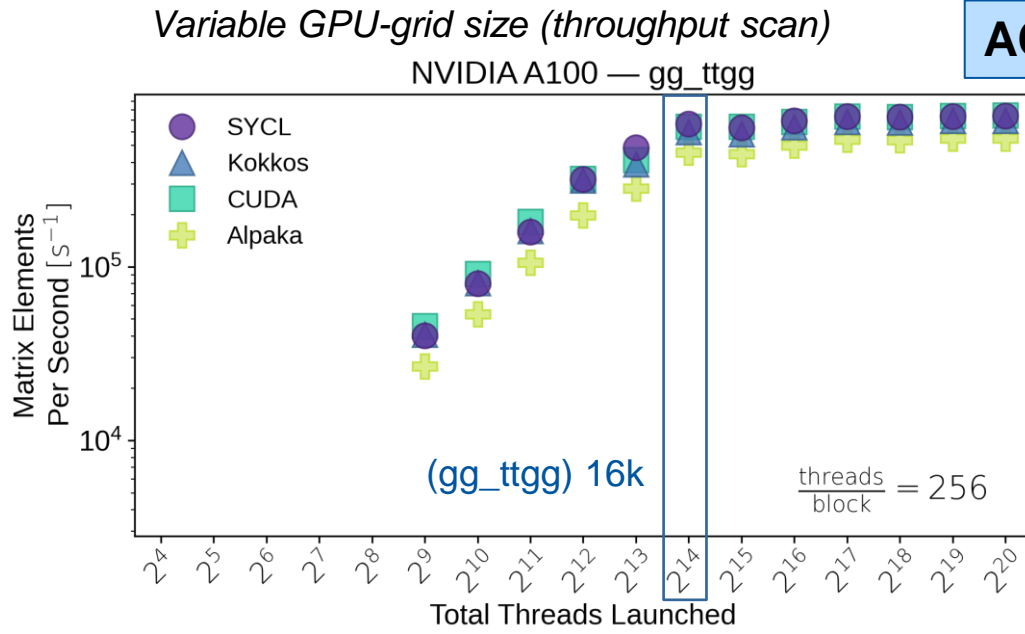
(SYCL implementation on GPUs)

CUDA vs SYCL on Nvidia GPUs

SYCL on Intel GPUs (compiler: Intel dpcpp)

CUDACPP vs SYCL on NVidia/AMD/Intel GPUs

ACAT2022

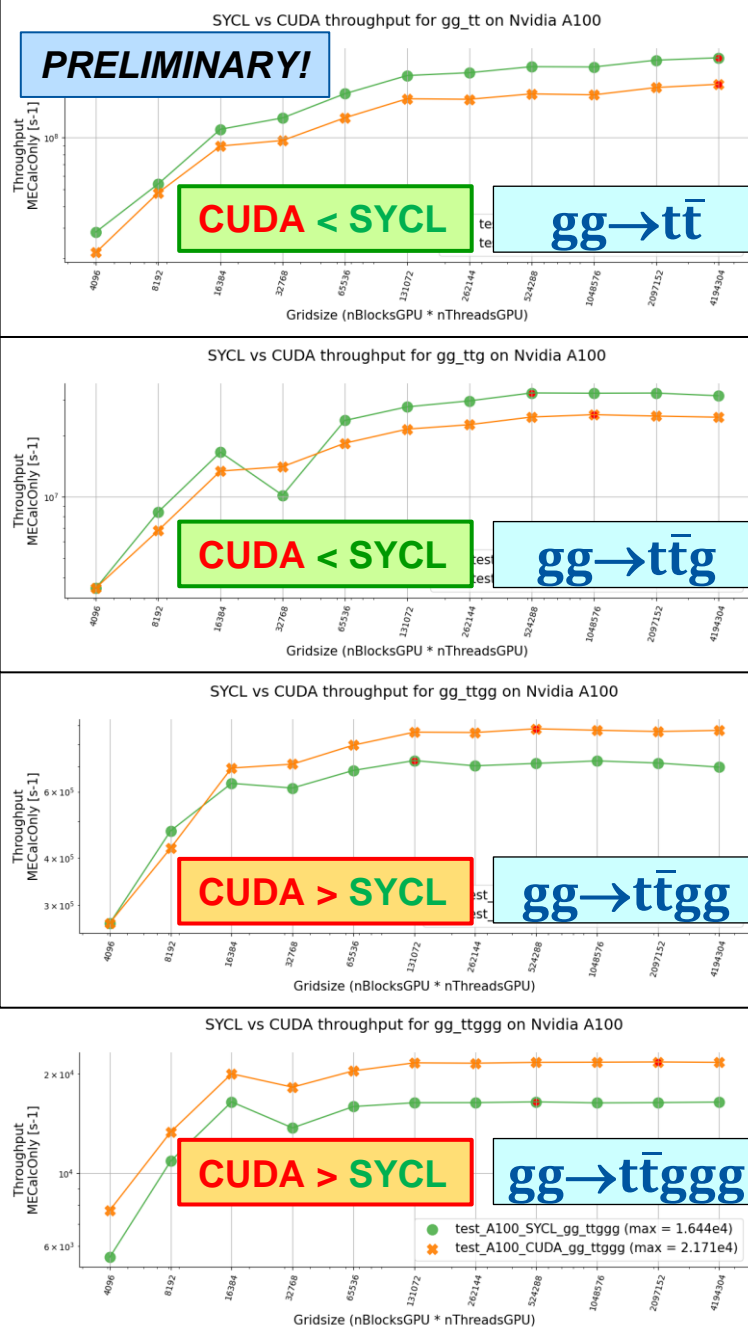


- Nvidia GPUs: the performances of the SYCL implementation seems ~comparable to direct CUDA for gg→tg
– More fine-grained analysis on the next slide, for different physics processes
- Intel and AMD GPUs: the SYCL implementation runs out of the box

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



CUDACPP vs SYCL on NVidia A100 GPUs



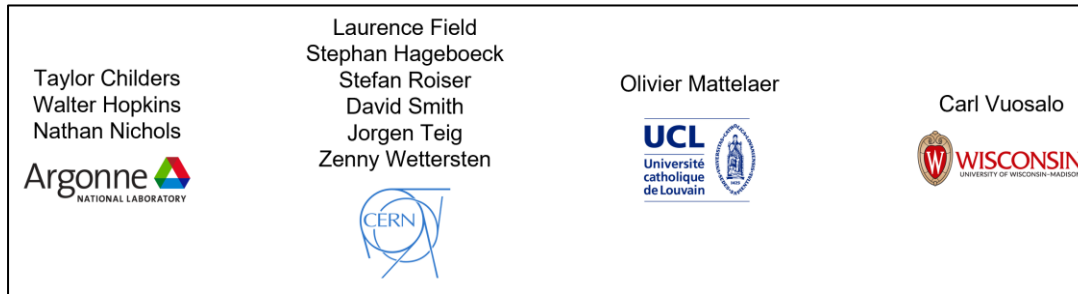
- SYCL and CUDA implementations have ~similar performances but
 - SYCL seems better for less complex processes
 - *CUDA seems better for more complex processes*
- These are very recent results, which are still being digested (WIP!)

Summary

- We have implemented efficient data-parallelism for Madgraph5 using CPU SIMD and GPUs
 - Our main implementation is based on CUDA and C++, we also have an alternative SYCL implementation
- On Intel Gold6xxx CPUs (2 FMA units), we achieve the full x8/x16 speedup of AVX512 for double/float
 - Lower-end Intel CPUs with 1 FMA unit are a factor 2 slower than those with 2 FMA units for AVX512
 - Is it possible to read directly the number of FMS units from /proc/cpuinfo or other O/S properties?
- We use compiler vector extensions for C++ vectorization in gcc and clang (and Intel icpx via clang)
 - The clang CVEs are also what sycl::vec uses under the hood in our alternative SYCL implementation
- The Intel icpx compiler gives very similar results to clang for C++ vectorization
 - Slightly worse than gcc for our baseline, but around a factor 2 better with aggressive inlining
- On NVidia GPUs, our direct CUDA implementation is better than the SYCL one for complex processes
 - On Intel and AMD GPUs, the SYCL implementation runs out of the box, unlike the CUDA implementation
 - Installing and using SYCL with all relevant plugins (e.g. NVidia) is complex: is it foreseen to 'yum install' this?

Acknowledgements

- Many thanks to the whole madgraph4gpu team for the great collaboration! 😊



- 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:
 - Igor Vorobtsov and Klaus-Dieter Oertel for our useful "Intel oneAPI coffee" discussions
 - Our Openlab colleagues for their help with CPU testbeds and the OneAPI development environment
 - Our CERN IT colleagues (especially Ricardo Rocha and Ulrich Schwickerath) for their help with GPU nodes!
 - Our EP-SFT colleagues for their help with new compilers and software development environments
 - Sebastien Ponce, Hadrien Grasland, Steve Lantz, Marco Clemencic for their suggestions on vectorization
 - The organizers and our mentors at the Sheffield 2020 and Lugano 2022 GPU hackathons
 - Domenico Giordano and the HEPiX benchmarking WG
 - The original authors of Madgraph5_aMC@NLO

BACKUP SLIDES

(mainly from the February 2023 CAF talk: <https://indico.cern.ch/event/1207838/>)

C++ vectorization – why choose Compiler Vector Extensions?

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

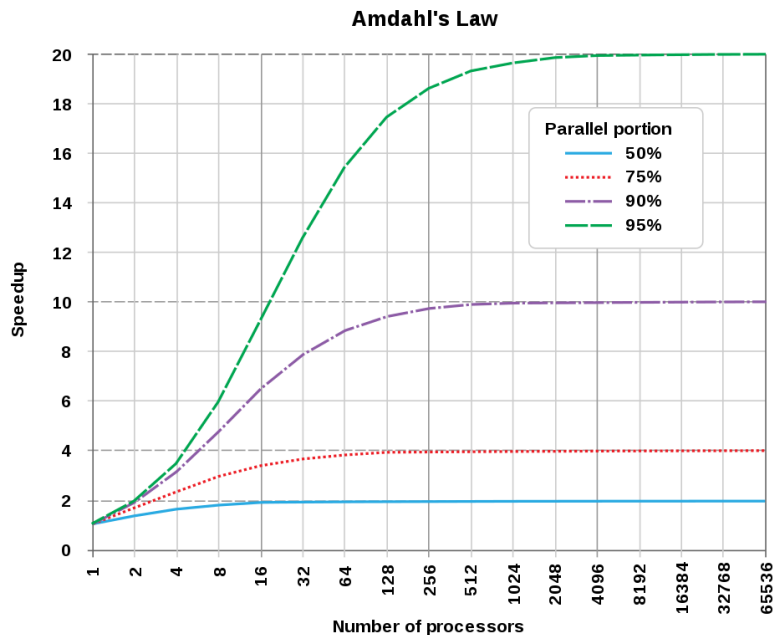
- Portable – available in gcc, clang, icpx (from clang) with minimal differences
 - *Do not require any external libraries* or tools (VC, VCL, VecCore, xSIMD, UME::SIMD, or SYCL...)
- Powerful, but easy to use
 - *No need to debug auto-vectorization* when it does not vectorize
 - *As powerful as intrinsics, 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 (RRRRIII 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*...

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!

Amdahl's law

- The matrix element calculation is now the bottleneck (e.g. >95% for gg→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!

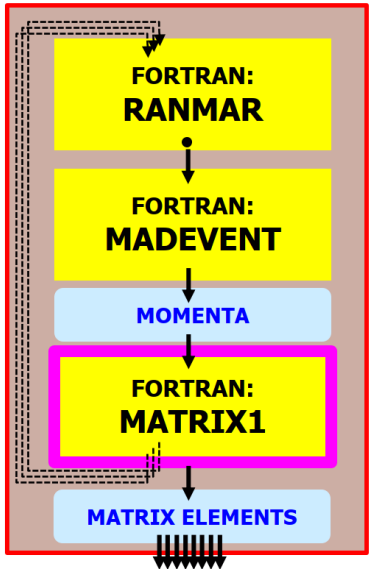


$$\lim_{s \rightarrow \infty} S_{\text{latency}}(s) = \frac{1}{1-p}$$

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

MG5aMC: old and new architecture designs

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

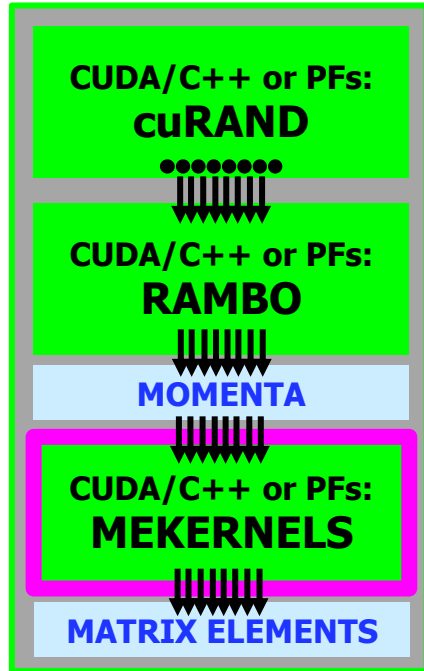


*MATRIX ELEMENT:
 CPU BOTTLENECK
 IN OLD MADEVENT*

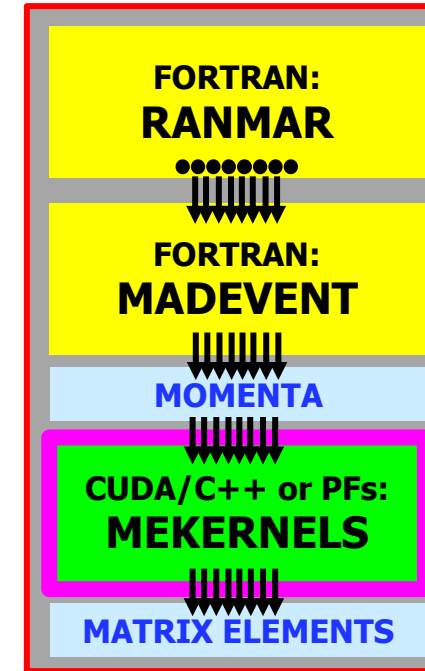
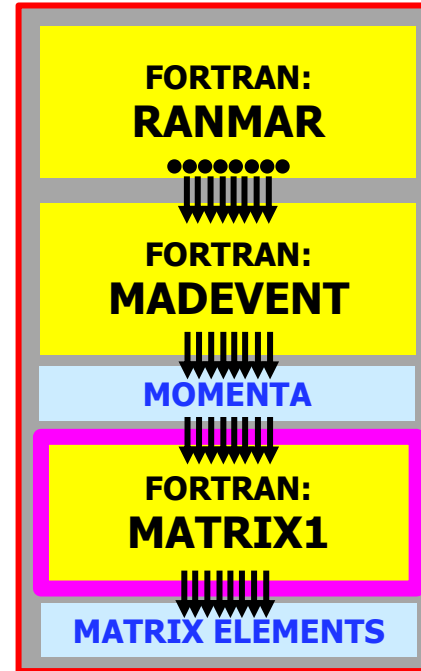
First we developed
 the new ME engines
 in standalone applications

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**



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



(Amdahl...)
**SCALAR:
 NEW
 BOTTLENECK?**

**PARALLEL:
 MUCH FASTER!**

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

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



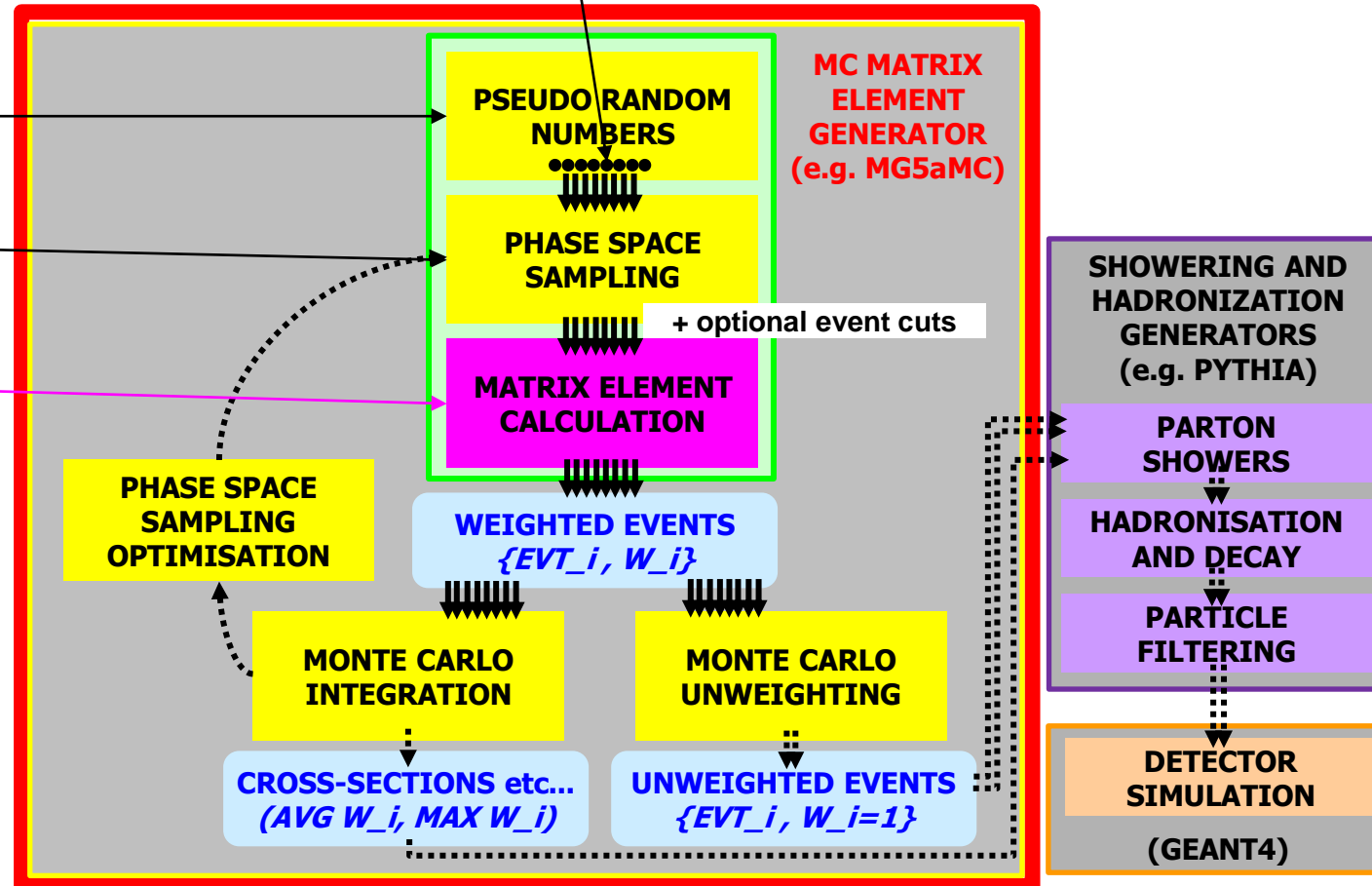
This can be parallelized (SIMT/SIMD and multithreading)

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

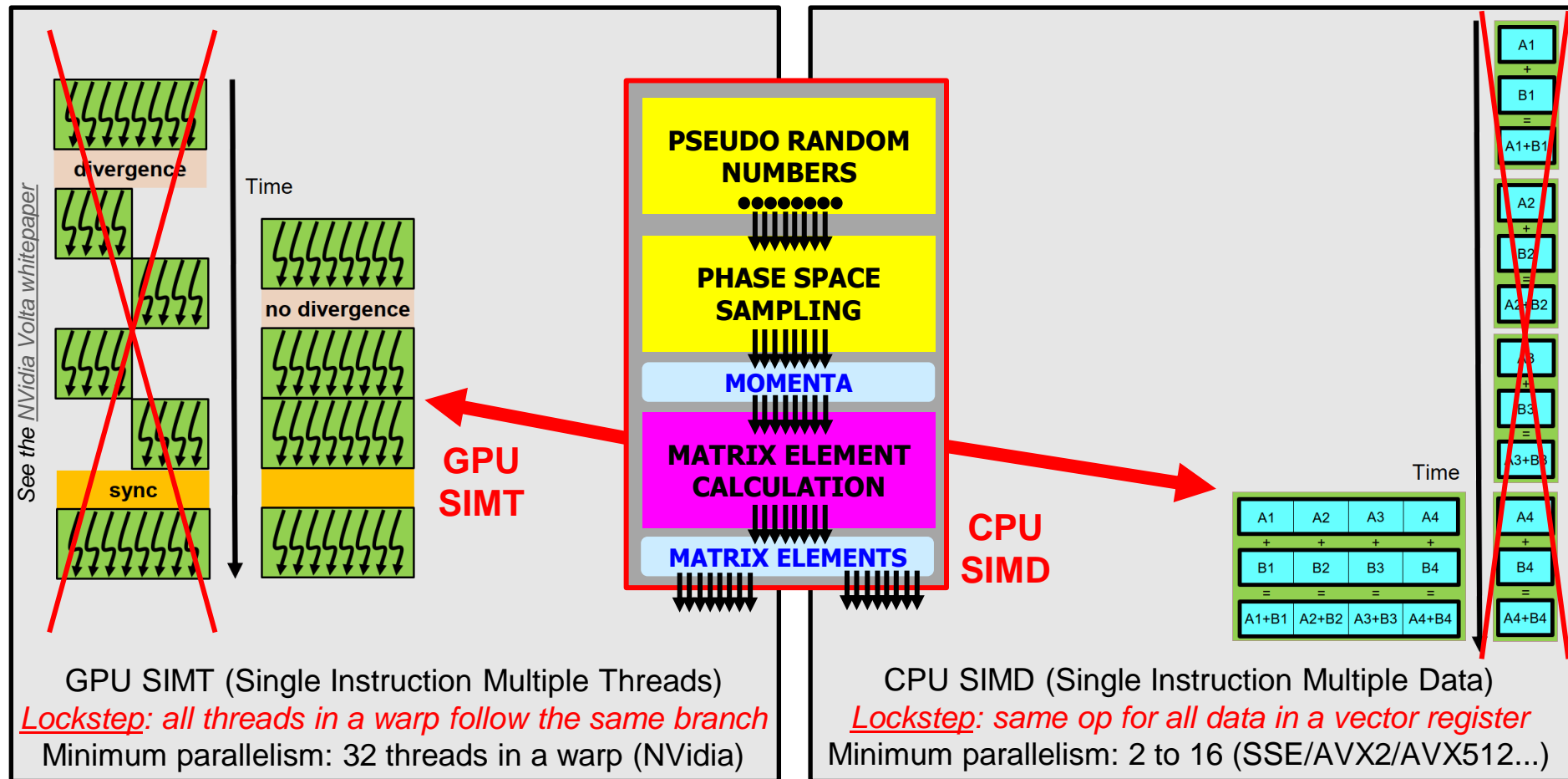


(NB: "Matrix Element" is an element of the **scattering matrix**... not a linear algebra concept!)

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

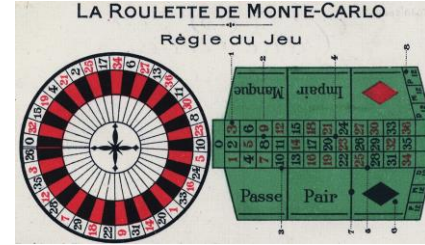
MG5aMC data parallelism: design for lockstep processing!

- 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)



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

ME event generators*

(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 before PS, fragmentation, detector simulation

INPUT



MC DECISIONS



Detector simulation (Geant4)

- Particle/matter interaction (when? how?)
- Particle decays (when?)

DIFFERENT CALCULATIONS
ON DIFFERENT DATA!

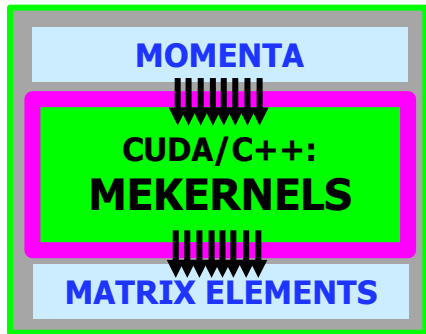
Stochastic branching
Bad for SIMT/SIMD

Event generators*

(after ME calculation):

- MC unweighting (keep/reject)
- Parton showers (PS)
- Fragmentation and decays

Memory layouts – AOS, SOA, AOSOA



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

We have experimented with three possible memory layouts for momenta

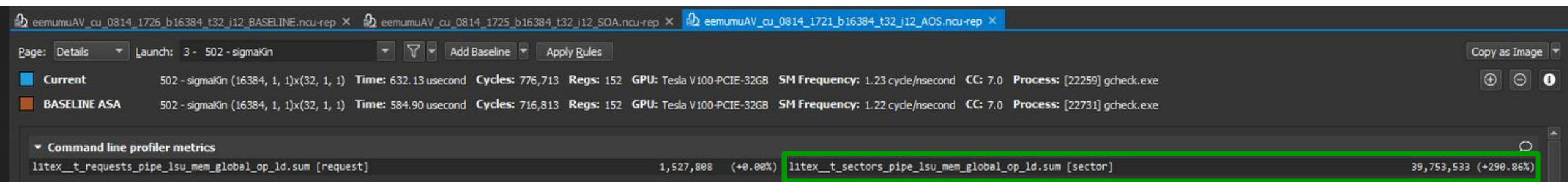
- (1) Array-of-Structures **AOS**: $momenta[Nevt][Npar][4]$
- (2) Structure-of-Arrays **SOA**: $momenta[Npar][4][Nevt]$
- (3) **AOSOA**: $momenta[Npag][Npar][4][Nepp]$ with $Nevt = Npag$ (“pages”) * $Nepp$ (“events per page”)

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

3. sum over helicities [Example for $gg \rightarrow t\bar{t}ggg$: 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.

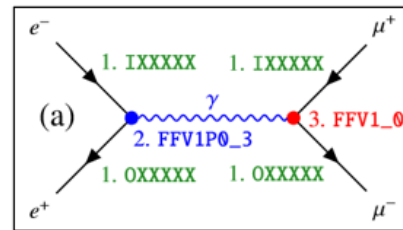
Helicity amplitudes – same code in CUDA and in vectorized C++

Formally the same code for three back-ends (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; // RRRRIIII (SOA)`

```

__device__
void FFV1_0( const cxttype_sv F1[], // input: wavefunction1[6]
             const cxttype_sv F2[], // input: wavefunction2[6]
             const cxttype_sv V3[], // input: wavefunction3[6]
             const cxttype COUP,
             cxttype_sv* vertex ) // output: amplitude
{
    mgDebug( 0, __FUNCTION__ );
    const cxttype cI( 0., 1. );
    const cxttype_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
cxttype_v operator+( const cxttype_v& a, const cxttype_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!

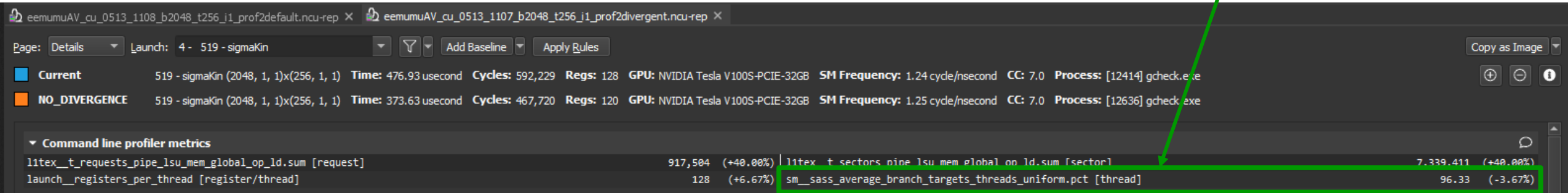
```

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



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)

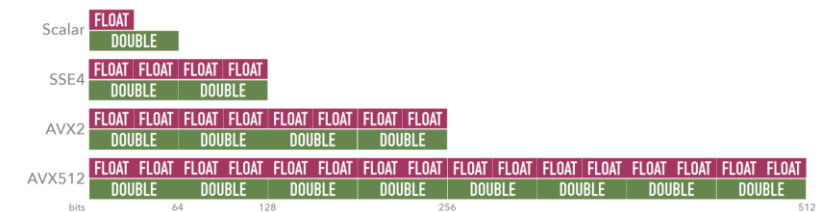


- CPU: the best lockstep metric IMO is the speedup over a no-SIMD case (reach theoretical maximum!)
 - but is also useful to disassemble the object using objdump and categorize SIMD intrinsics symbols...

4a90ec2 gg→tggg

# 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

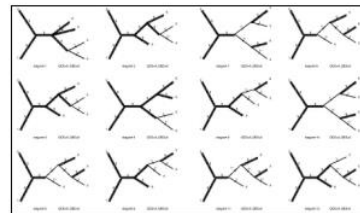
	ACAT2022	madevent
$gg \rightarrow t\bar{t}gg$	MEs precision	$N_{events}/tMEs$ [MEs/sec]
Fortran(scalar)	double	2.30E3 (=1.0)
C++/none(scalar)	double	2.28E3 (x1.0)
C++/sse4(128-bit)	double	4.62E3 (x2.0)
C++/avx2(256-bit)	double	1.05E4 (x4.6)
C++/512y(256-bit)	double	1.16E4 (x5.0)
C++/512z(512-bit)	double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



Code generation: how did we bootstrap the project?

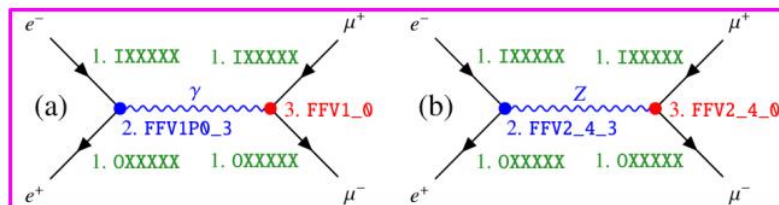
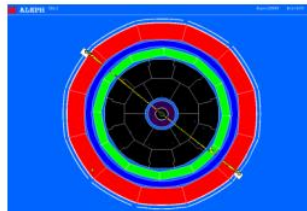
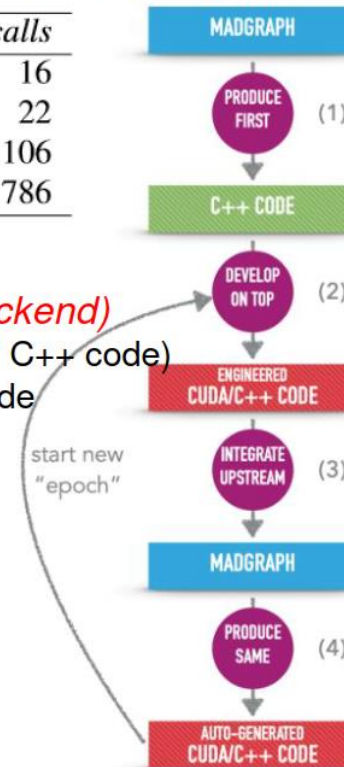
Code is auto-generated \Rightarrow Iterative development process

- 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

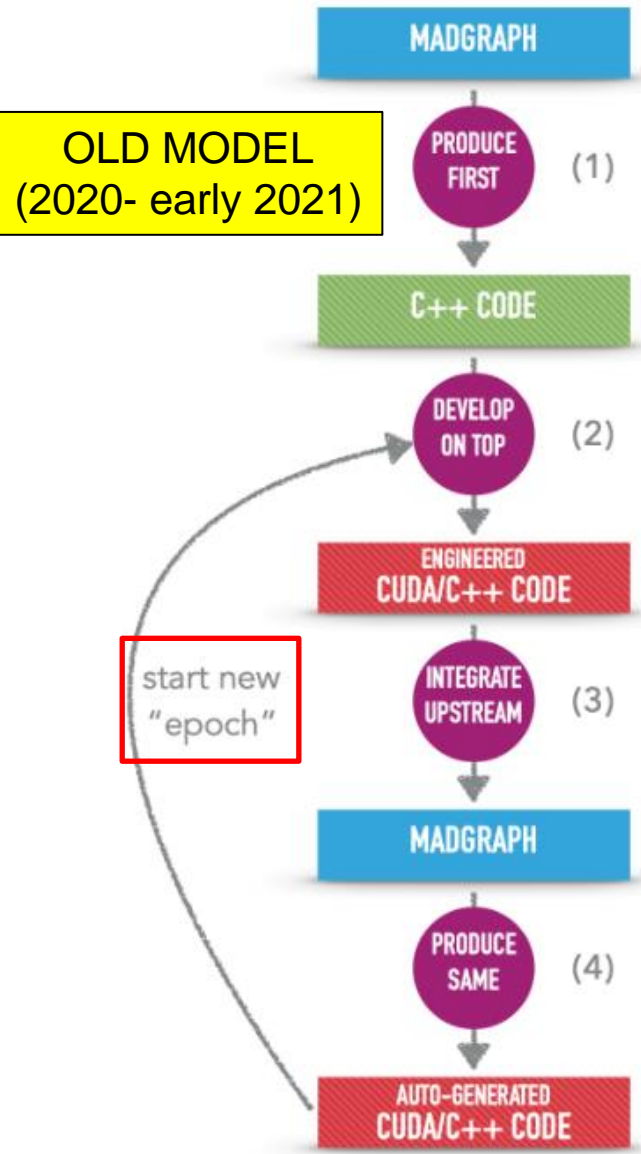


Process	LOC	functions	function calls
$e^+e^- \rightarrow \mu^+\mu^-$	776	8	16
$gg \rightarrow t\bar{t}$	839	10	22
$gg \rightarrow t\bar{t}g$	1082	36	106
$gg \rightarrow t\bar{t}gg$	1985	222	786

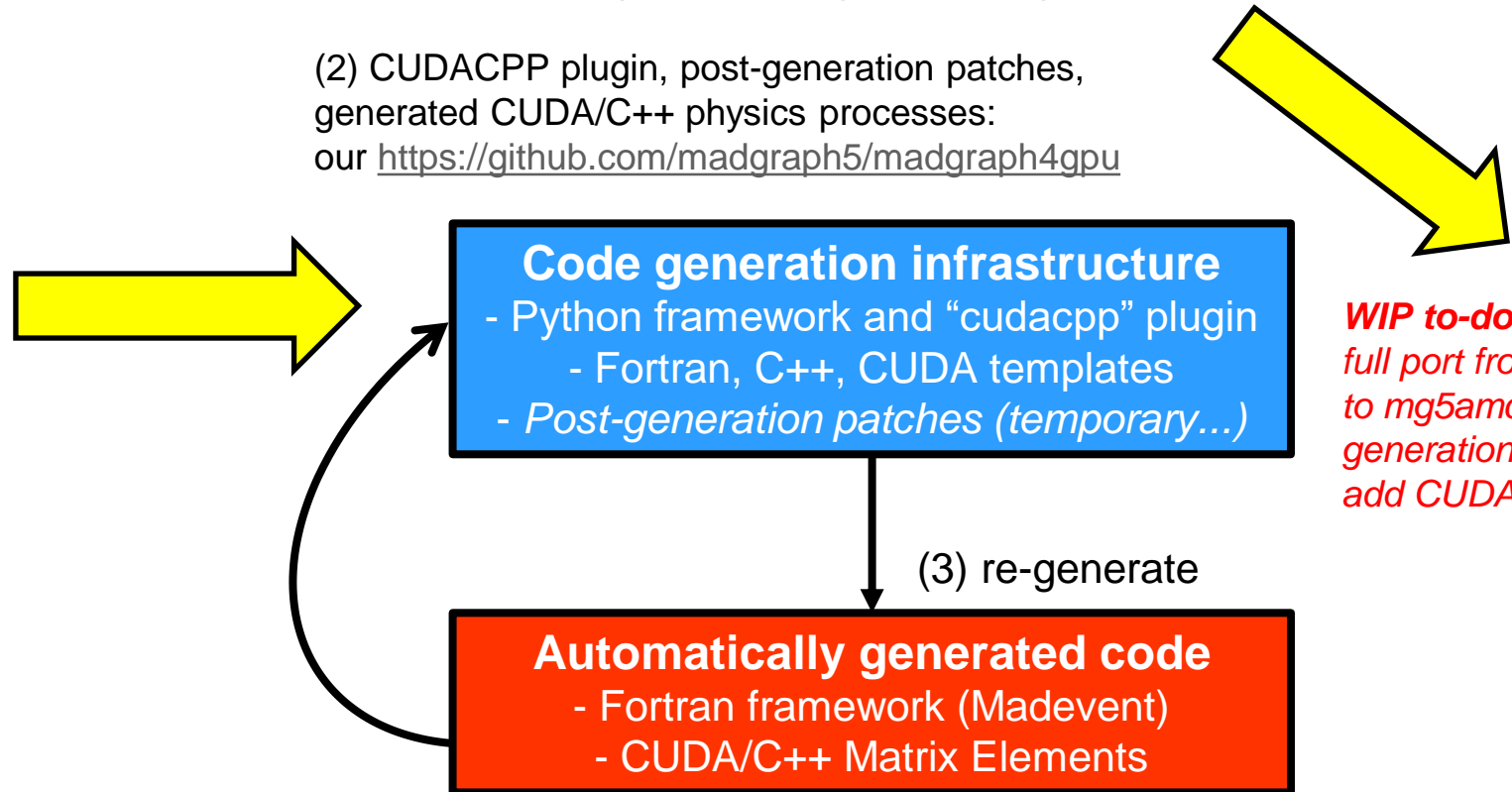
- Goal: modify code-generating code (add CUDA, improve C++ backend)*
 - (1) Start simple: *bootstrap with $e^+e^- \rightarrow \mu^+\mu^-$* (two diagrams, few lines of C++ code)
 - (2,3) Add CUDA and improve C++, port upstream to Python meta-code
 - (4) *Generate more complex LHC processes $gg \rightarrow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



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



- (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>



NEW MODEL
(since end 2021)

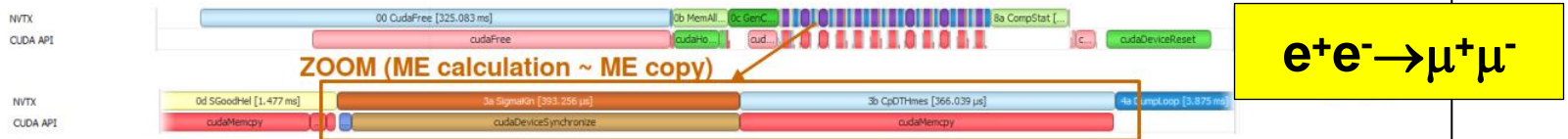
- (1) develop on top of auto-generated code
- (2) backport immediately to code generation infrastructure

WIP to-do before a release: full port from madgraph4gpu to mg5amcnlo (remove post-generation Fortran patches, add CUDACPP upstream)

Why focus on complex processes? Compute >> memory!

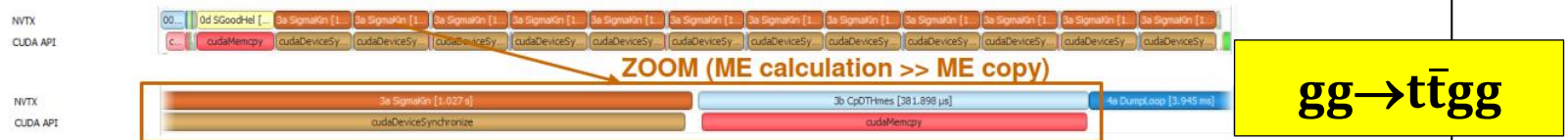
CUDA: Host(CPU)-to/from-Device(GPU) data copy has a cost

- 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)



- 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*

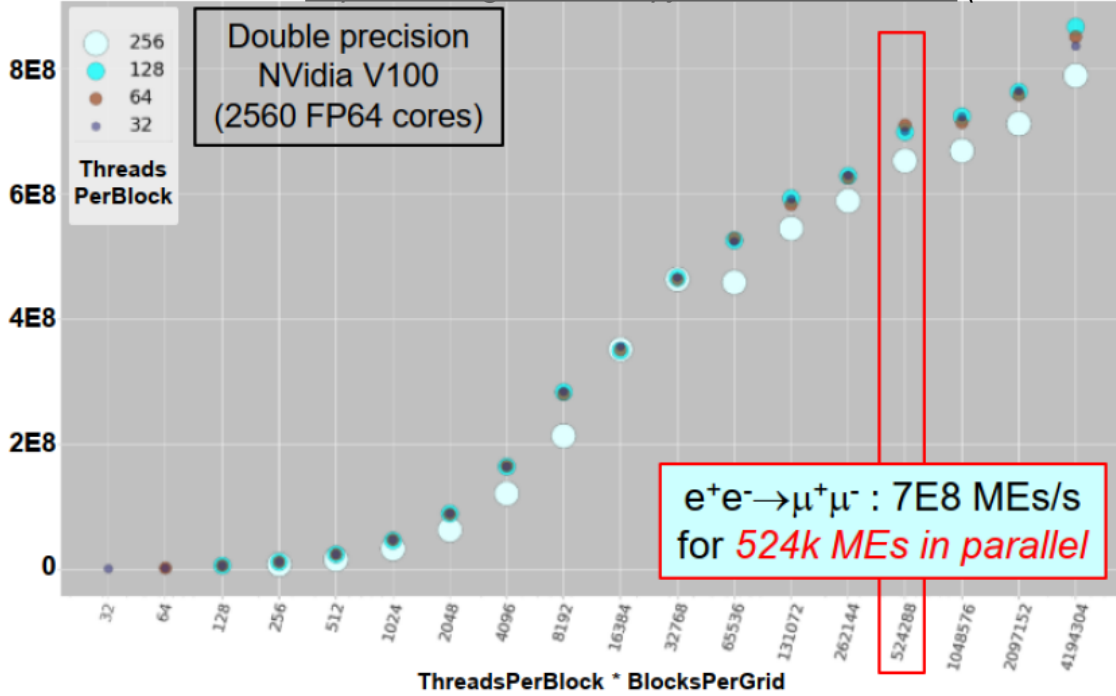


- We are lucky: the more complex the physics process, the less relevant is the cost of GPU-CPU data copies!
 - Similarly (later): the more complex the process, the less relevant is the overhead from scalar Fortran in madevent!
 - And the fewer events in flight needed to fill the GPU...

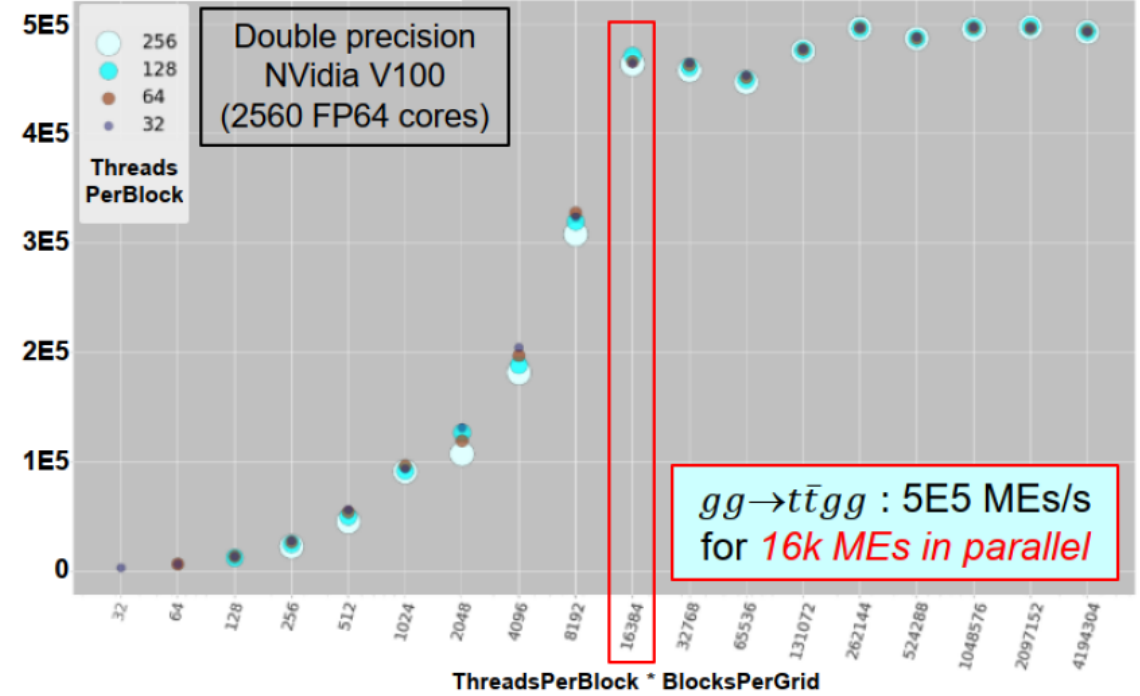
- In this talk I mainly give performance numbers for complex processes like $gg \rightarrow t\bar{t}gg$ or $gg \rightarrow t\bar{t}ggg$

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

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



Matrix Elements / second



- 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)?

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)

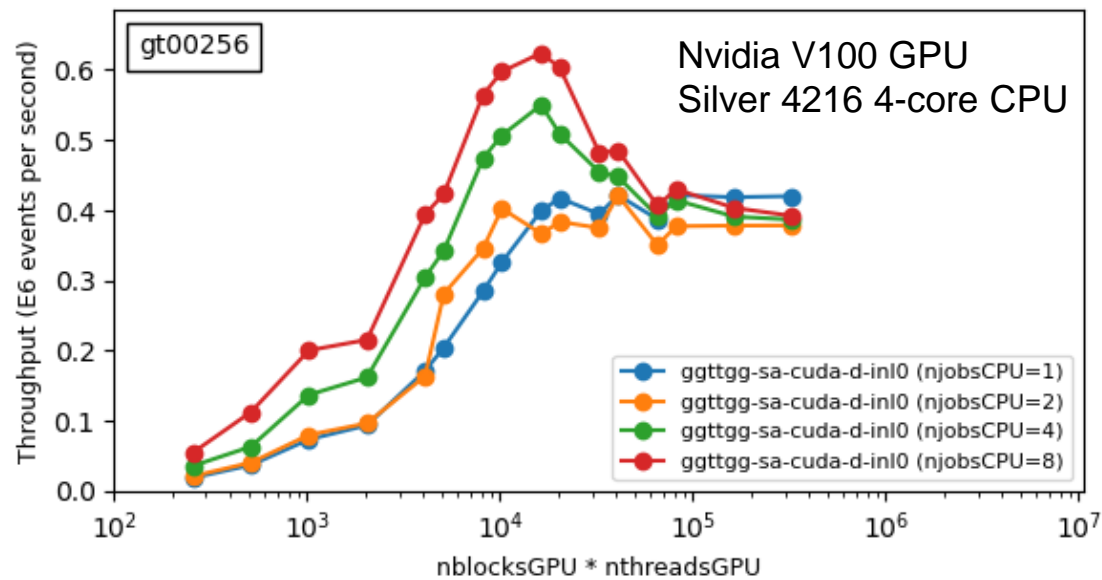
- Complete the **back-port of code generation** from madgraph4gpu to mg5amcnlo upstream
 - Including extra cross-checks that the LHE color IDs are those required for parton showers
- Make the CUDA/C++ madevent executable consistent with the Madgraph “launch” infrastructure
 - Modify the names and input 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 gggtgg
- 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 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)	✓	✓	✗	✗	✗

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 $gg \rightarrow t\bar{t}gg$, 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 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?*

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

- 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

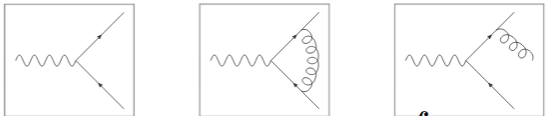
Z. Wettersten (+ OM, SR, AV, R. Schoefbeck)

- 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

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



$$d\sigma_{NLO}^n = d\sigma_{LO}^n + d\sigma_V^n + \int d\Phi_1 d\sigma_R^{n+1}$$

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

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

- *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)?