Towards a HEP-workload GEN GPU benchmark: MadGraph5_aMC@NLO

Andrea Valassi (CERN IT-SC-RD)

HEPiX Benchmarking WG meeting, 5th November 2020 - https://indico.cern.ch/event/946409

Foreword: GEN workloads and GPUs in WLCG

- MC event generators are essential for HEP physics analyses
  - First step in the GEN-SIM-DIGI-RECO chain for simulated data
  - MadGraph5_aMC@NLO (MGaMC) is one of the main generators at the LHC

- Growing interest (and concerns) about the software aspects of generators
  - *GEN workloads are a sizeable (and growing) fraction of WLCG CPU budgets*
    - Currently ~12% for ATLAS (mainly Sherpa) and ~5% for CMS (mainly MGaMC)
  - Paper for the HL-LHC LHCC review: https://arxiv.org/abs/2004.13687
    - See also the recent LHCC talk: https://doi.org/10.5281/zenodo.4028834
  - Code modernization and speedup, faster algorithms, port to new architectures…

- WLCG does not provide/account GPUs yet, but it will one day
  - Example: HPCs (and the experiments already have access to these resources)
  - Only few workloads could make sizable use of GPUs today (e.g. CMS patatrack)
  - Porting generators to GPUs would open up new ways of using GPUs in WLCG
    - *And generators are natural candidates for exploiting parallelism on GPUs (see later…)*

$\Rightarrow$ *Generators are a natural fit for preparing a HEP-workload GPU benchmark*
  - Bonus: in principle one can run exactly the same algorithm on CPUs and GPUs

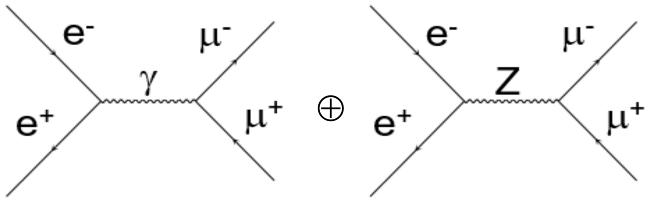
MGaMC on GPU – overview

- Why choose MadGraph and not another MC generator for a GPU port?
  - Earlier efforts at KEK in 2010-2013, which were never released for production
    - Touched both matrix element (ME) and MC integration – see summary at HOW2019
    - We are ~not leveraging on this work (based on old version of MadGraph's ME library)
  - Main reason: active involvement of core MadGraph developer (Olivier Mattelaer)

- Project is maintained on https://github.com/madgraph5/madgraph4gpu
  - Overall coordination: Stefan Roiser
  - Composite effort: *physicists & engineers* (CERN, Louvain, Argonne, Bangalore)
    - Core CUDA development: AV, OM, SR, Taran Singhania
    - Abstraction layers, profiling, MC integration: David Smith, Laurence Field, Smita Darmora, Taylor Childers, Tyler Burch, Walter Hopkins
    - Just joined: Josh McFayden, Stephan Hageboek
  - Meetings every two weeks
    - Plus more frequent discussions of core CUDA developers
  - The collaboration is largely a spin-off of the activities in the HSF generator WG
    - https://hepsoftwarefoundation.org/workinggroups/generators.html

The core of MGaMC is a code-generating code

- The heart of a MC generator is the software that calculates the *"matrix element" (ME)* for a given process – an element of the "scattering matrix"
  - This is the sum of many contributions, one for each relevant Feynman diagram
    - Simple example: LEP process, e+e- to μ+μ-, only two diagrams



    - LHC processes (e.g. gg to $t\bar{t}gg$) are much more complex, many more diagrams

- For every different physics process, MadGraph does two things
  - It identifies which are the relevant Feynman diagrams
  - It generates the code to compute the ME from those Feynman diagrams

$\Rightarrow$The core of MadGraph is code-generating code (ALOHA, written in Python)
  - The generated code can be *Fortran (default!)*, C++, Python*… or CUDA (new!)*

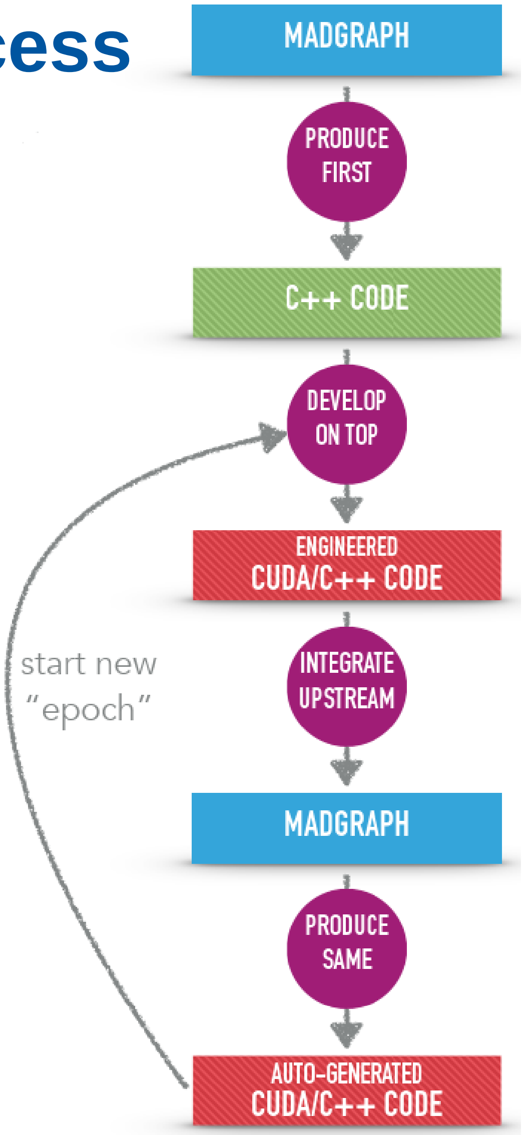
- There are also other (hardcoded) components, the same for all processes

MGaMC on GPU – development process

- MadGraph being a code-generating code complicates the development process
  - We eventually must deliver CUDA-generating code
  - We start from the C++-generating code

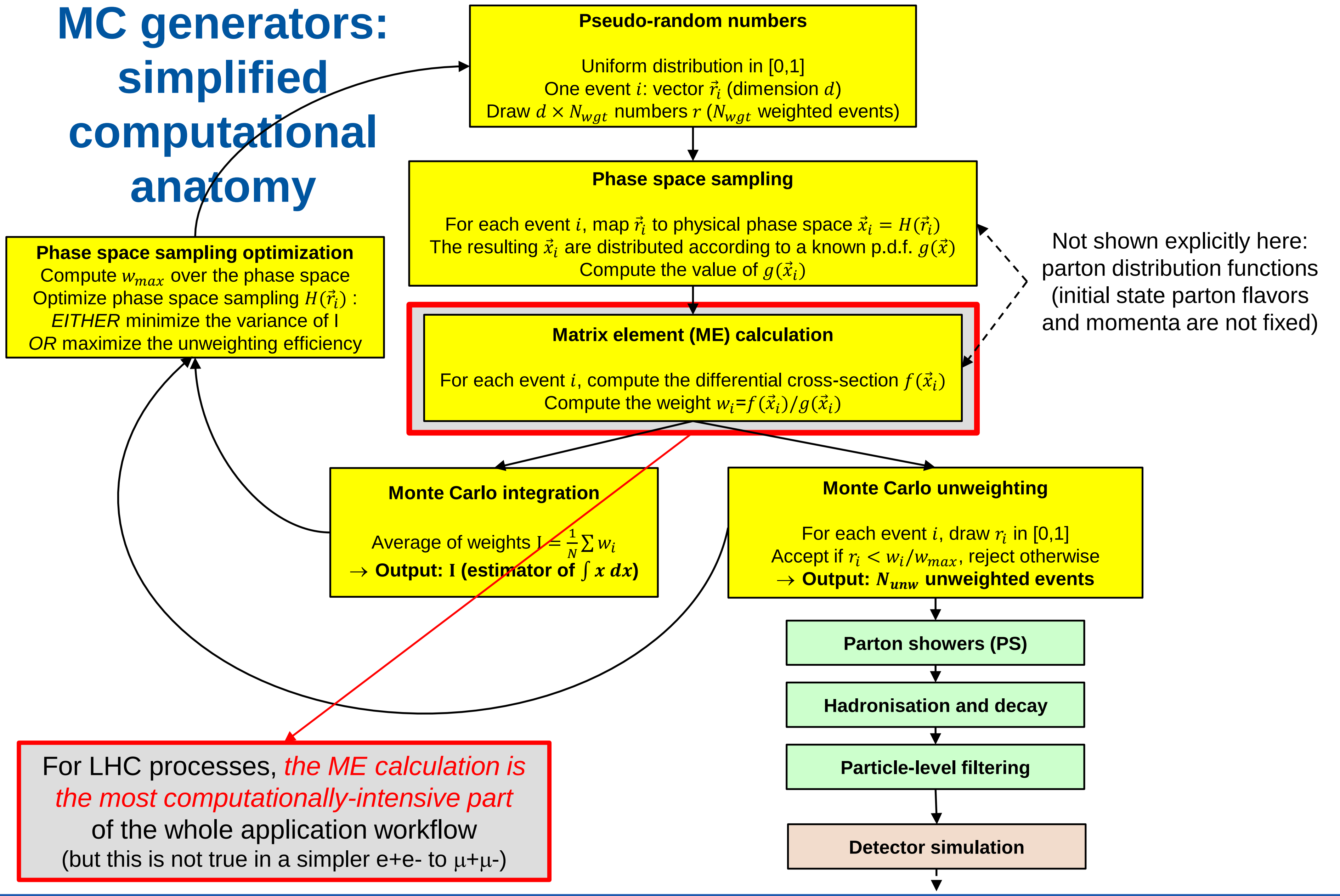
- We start from a simple process, e+e- to μ+μ-
  - Few diagrams = Few lines of code to manually port to CUDA, and manually optimize…
  - Then the code-generating code must be adapted
  - And we start another iteration
    - Potentially on a complex process with more diagrams

- Eventually: not only CUDA/Nvidia, also Intel, AMD?
  - Either natively or using abstraction layers: oneAPI, SYCL, HIP/ROCm, Alpaka, Kokkos, OpenCL…
  - Code-generating code would be needed for these too



S. Roiser, madgraph4gpu meeting, Oct 2020

MC generators: simplified computational anatomy



Pseudo-random numbers

Uniform distribution in [0,1]
One event i: vector $\vec{r}_i$ (dimension d)
Draw $d \times N_{wgt}$ numbers r (N_{wgt} weighted events)

Phase space sampling

For each event i, map $\vec{r}_i$ to physical phase space $\vec{x}_i = H(\vec{r}_i)$
The resulting $\vec{x}_i$ are distributed according to a known p.d.f. $g(\vec{x})$
Compute the value of $g(\vec{x}_i)$

Matrix element (ME) calculation

For each event i, compute the differential cross-section $f(\vec{x}_i)$
Compute the weight w_i=$f(\vec{x}_i)/g(\vec{x}_i)$

Phase space sampling optimization
Compute w_{max} over the phase space
Optimize phase space sampling $H(\vec{r}_i)$:
EITHER minimize the variance of I
OR maximize the unweighting efficiency

Not shown explicitly here: parton distribution functions (initial state parton flavors and momenta are not fixed)

Monte Carlo integration

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

Monte Carlo unweighting

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

Parton showers (PS)

Hadronisation and decay

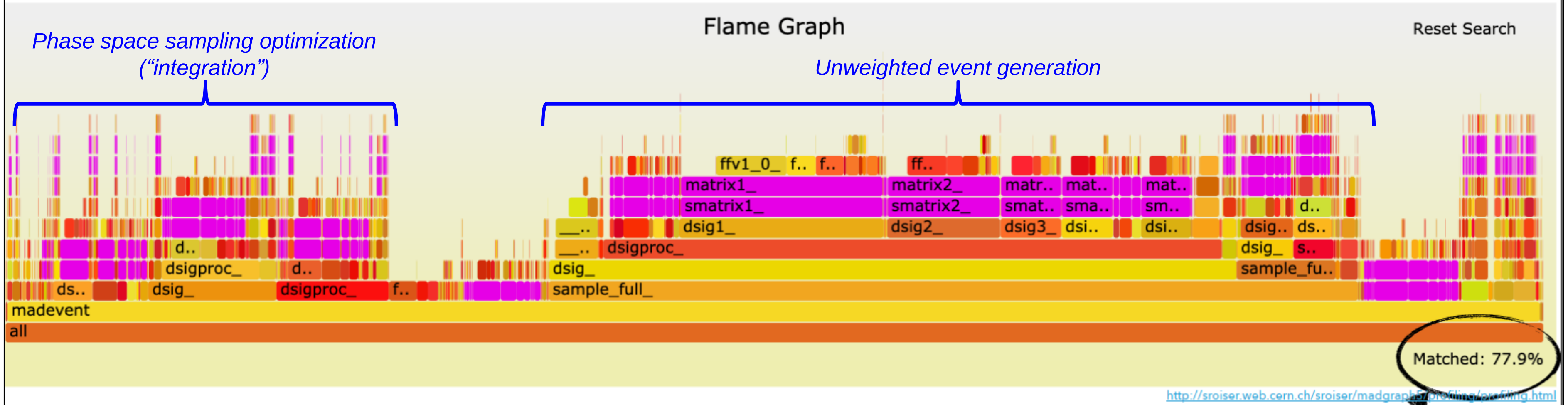
Particle-level filtering

Detector simulation

For LHC processes, *the ME calculation is the most computationally-intensive part* of the whole application workflow
(but this is not true in a simpler e+e- to μ+μ-)

The ME calculation is where CPU is spent at LHC

- E.g. real world CMS example: p p –> l+ l- j j j j / h @0
  - Madgraph/MadEvent (Fortran), 10^5 events



MATRIX ELEMENT CALCULATION

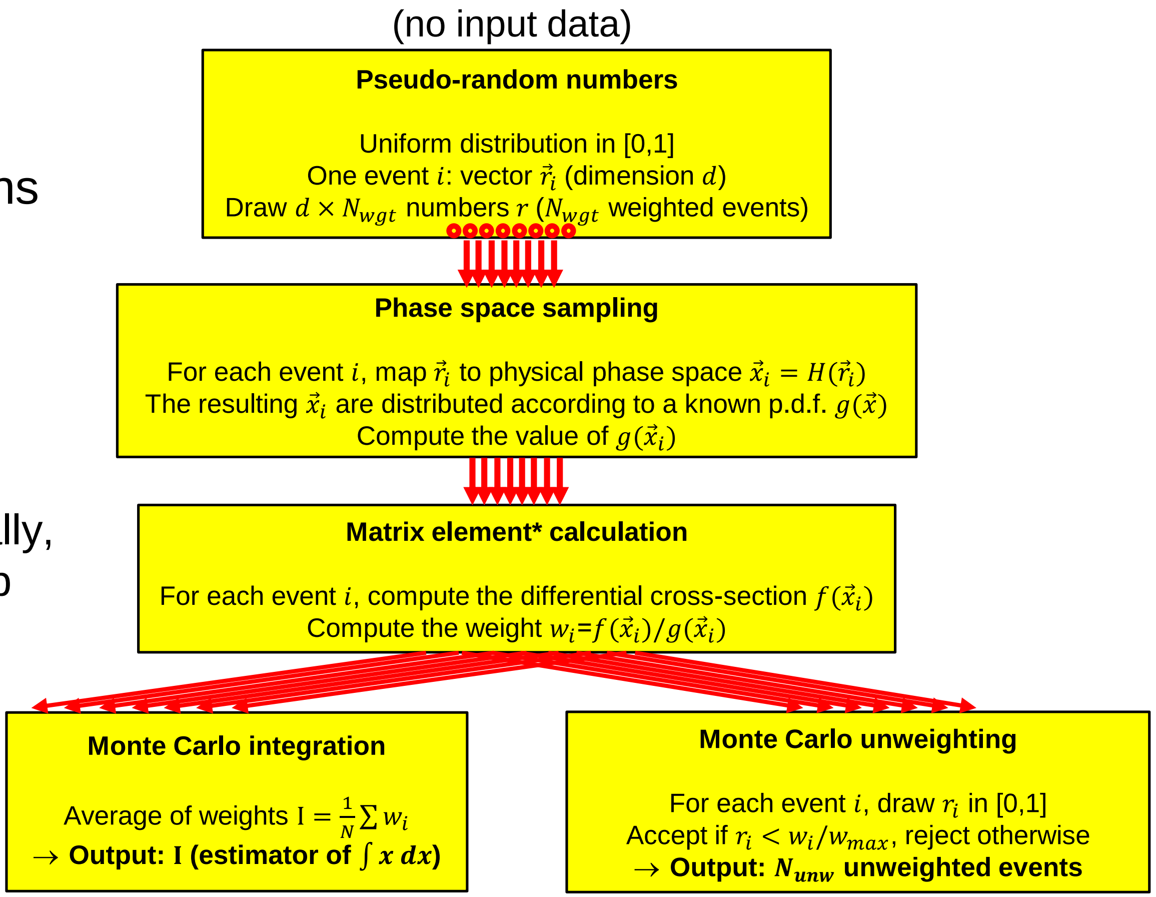
- Matrix element calculations use majority of CPU time

S. Roiser, madgraph4gpu meeting, Oct 2020

MGaMC on GPU: event-level data-parallel approach

- One main reason why generators are excellent candidates for a GPU port:
  - *The compute-intensive ME calculation is exactly the same function for all events*
  - This reduces the risk of “thread divergence” on GPUs (unlike detector simulation)

- We use an event-level data-parallel approach
  - Execute the same operations on multiple data in parallel
  - *On GPUs: SPMD (Single Program Multiple Data)*
  - On CPUs: SIMD (Single Instruction Multiple Data)?
    - Aka vectorization: eventually, we will also try to speed up the C++ code this way



(no input data)

Pseudo-random numbers

Uniform distribution in [0,1]
One event i: vector $\vec{r}_i$ (dimension d)
Draw $d \times N_{wgt}$ numbers r (N_{wgt} weighted events)

Phase space sampling

For each event i, map $\vec{r}_i$ to physical phase space $\vec{x}_i = H(\vec{r}_i)$
The resulting $\vec{x}_i$ are distributed according to a known p.d.f. $g(\vec{x})$
Compute the value of $g(\vec{x}_i)$

Matrix element* calculation

For each event i, compute the differential cross-section $f(\vec{x}_i)$
Compute the weight $w_i = f(\vec{x}_i)/g(\vec{x}_i)$

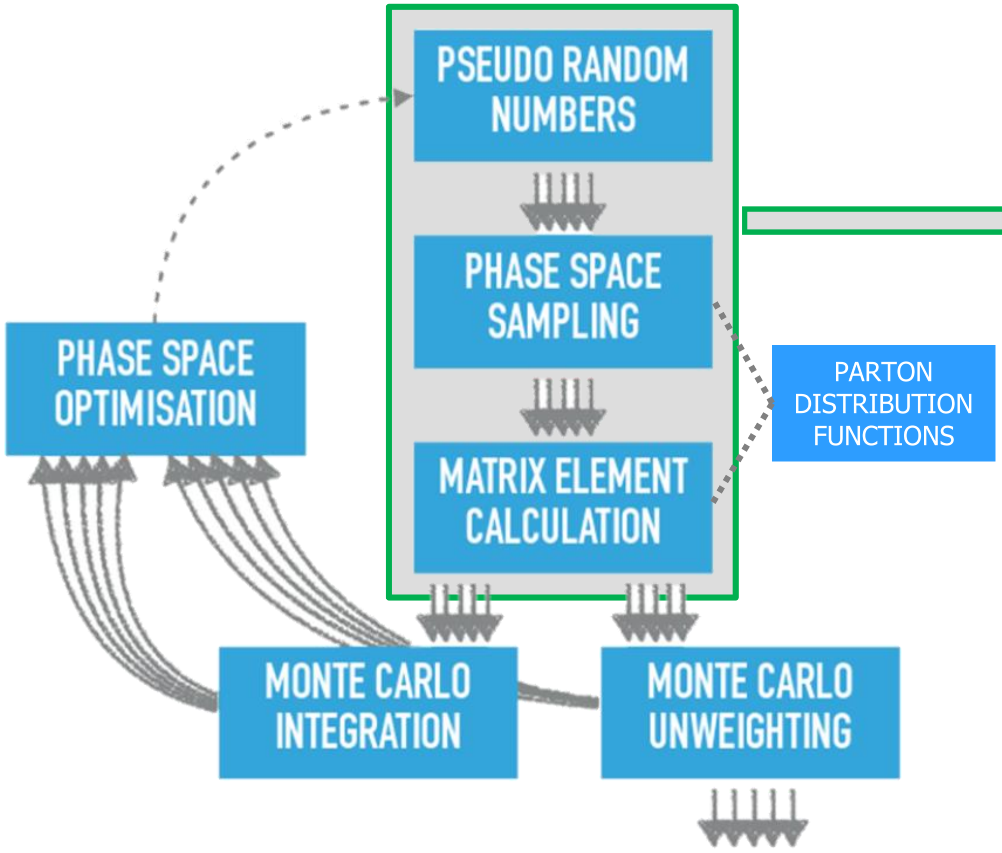
Monte Carlo integration

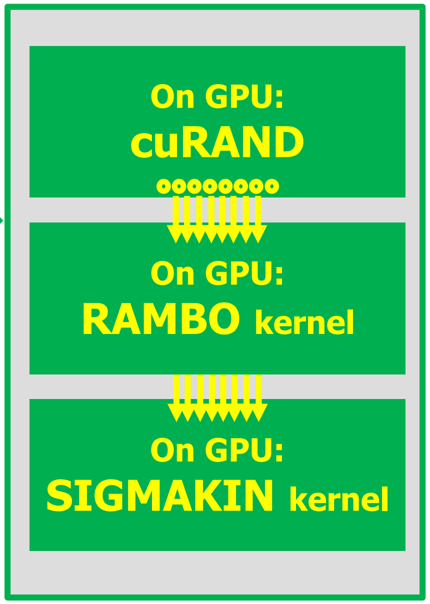
Average of weights $\mathrm{I} = \frac{1}{N}\sum w_i$
$\rightarrow$ **Output: I (estimator of $\int x\,dx$)**

Monte Carlo unweighting

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

Status: where are we now?





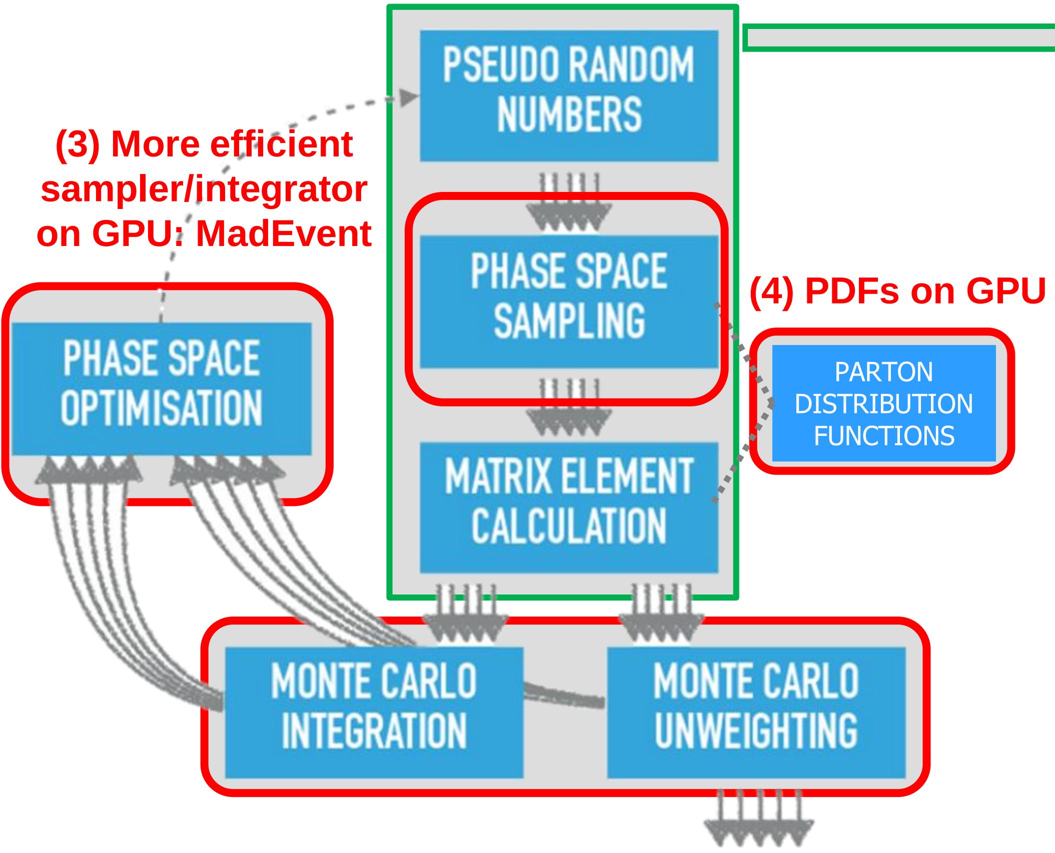
Basic (uniform) “democratic” sampler, no optimization

ME for simple (LEP) e+e- to μ+μ- process

CUDA implementation
Not yet fully ported upstream to code-generating code

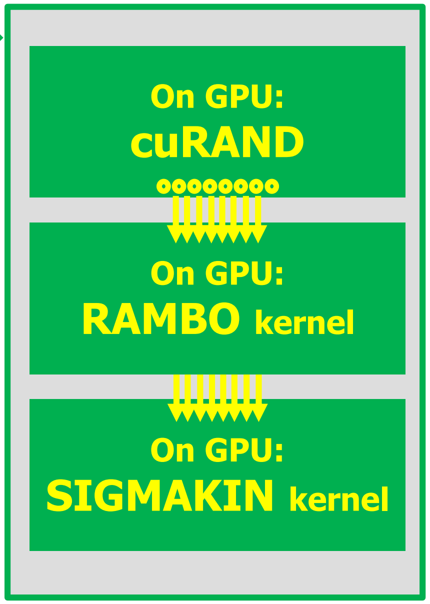
Single source code for C++ and CUDA (using #ifdef’s): same algorithms

Status: which are we the (main) ingredients that we are missing for LHC processes?



(3) More efficient sampler/integrator on GPU: MadEvent

(4) PDFs on GPU



Basic (uniform) “democratic” sampler, no optimization

ME for simple (LEP) e+e- to μ+μ- process

(2) gg to $t\bar{t}gg$ ME Complete the code-generating code

CUDA implementation

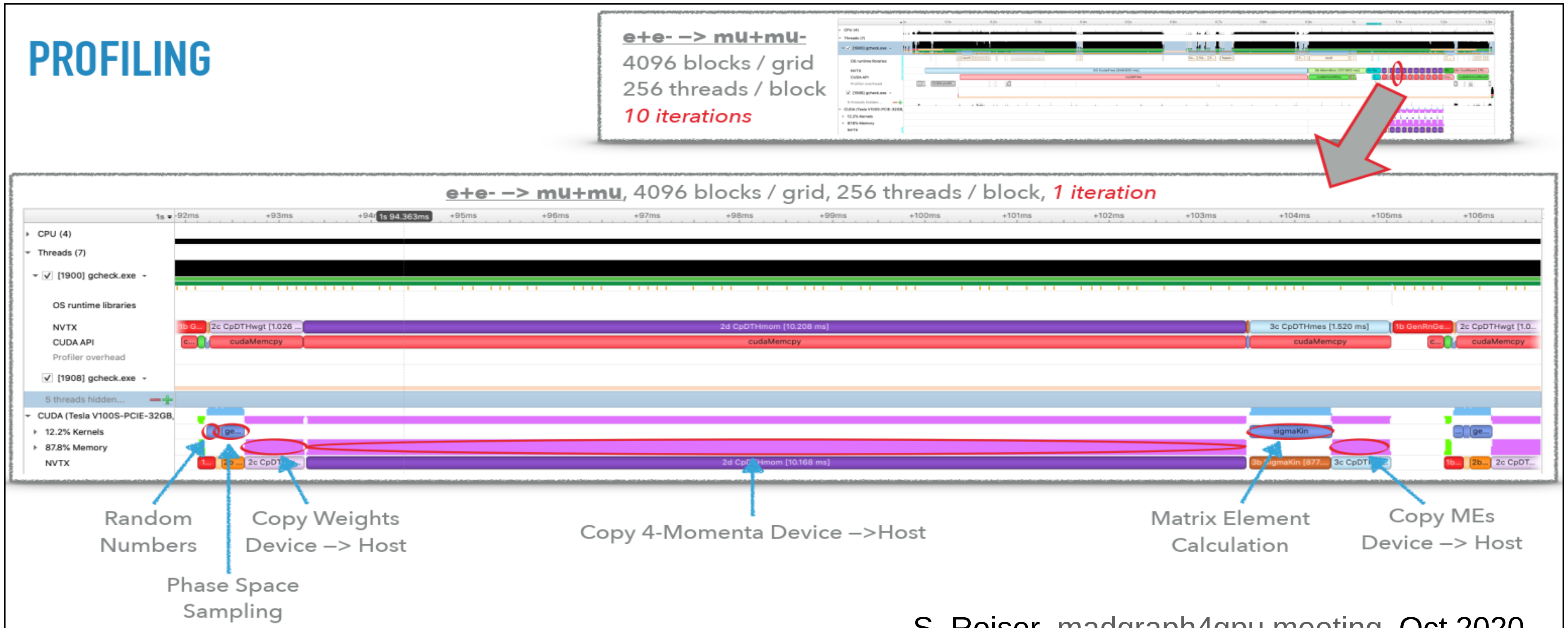
Not yet fully ported upstream to code-generating code

Single source code for C++ and CUDA (using #ifdef’s): same algorithms

(1) More complete workflow

NB: this is needed also to extract more realistic benchmark results for a possible BMK container based on e+e- to μ+μ-

Benchmarking based on (our current) e+e- to μ+μ-?


PROFILING
e+e- –> mu+mu-
4096 blocks / grid
256 threads / block
10 iterations
e+e- –> mu+mu, 4096 blocks / grid, 256 threads / block, 1 iteration
Random Numbers
Phase Space Sampling
Copy Weights Device –> Host
Copy 4-Momenta Device –>Host
Matrix Element Calculation
Copy MEs Device –> Host

S. Roiser, madgraph4gpu meeting, Oct 2020

Most time is currently spent in data copy. Conclusion: e+e- –> mu+mu- calculations are too simple

- *Benchmarking GPUs today based on MG is possible, but VERY preliminary!*
  - 1. Even for eeμμ, we need at least a realistic integration/unweighting workflow
  - 2. In any case, it would not be representative of more complex LHC processes

Hardware benchmarking, Software benchmarking

- In the HEPiX BMK WG we mainly deal with *hardware benchmarking*
  - Run the same frozen software on different CPUs/GPUs and compare them
    - Goals: accounting/pledging and procurement of compute resources (à la HS06)
  - Reproducible applications, pre-built and containerized as docker images
- In the madgraph4gpu effort we do a whole lot of *software benchmarking*
  - *Change the software (or change build options) and compare old/new versions*
  - We do more, but we *also* compare software performance on different GPUs
    - Software optimizations depend on hardware (professional/FP64 vs consumer/FP32)
    - Software languages and abstraction layers depend on hardware (Nvidia, AMD, Intel)
- There is a lot in common, but also some important differences
  - Main similarity: we both need reproducible application workloads
  - Main difference: prebuilt library/executables vs source code to rebuild
  - It makes sense to keep these two efforts well synchronized

How do we benchmark our own code today? WIP!!!

CUDA/GPU
Full V100

```
./gcheck.exe -p 16384 32 12
***************************************
NumIterations           = 12
NumThreadsPerBlock      = 32
NumBlocksPerGrid        = 16384
---------------------------------------
TotalEventsComputed     = 6291456
RamboEventsPerSec       = 8.130406e+07 sec^-1
MatrixElemEventsPerSec  = 6.703134e+08 sec^-1
***************************************
MeanMatrixElemValue     = 1.372152e-02 GeV^0
***************************************
00 CudaFree : 0.928572 sec
0a ProcInit : 0.000625 sec
0b MemAlloc : 0.062333 sec
0c GenCreat : 0.010851 sec
1a GenSeed  : 0.000015 sec
1b GenRnGen : 0.007457 sec
2a RamboIni : 0.000108 sec
2b RamboFin : 0.000049 sec
2c CpDTHwgt : 0.006603 sec
2d CpDTHmom : 0.070621 sec
3a SGoodHel : 0.001733 sec
3b SigmaKin : 0.000081 sec
3c CpDTHmes : 0.009305 sec
4a DumpLoop : 0.022506 sec
8a CompStat : 0.031386 sec
9a GenDestr : 0.000064 sec
9b MemFree  : 0.013876 sec
9c CudReset : 0.027250 sec
9d DumpScrn : 0.000217 sec
9e DumpJson : 0.000003 sec
TOTAL       : 1.193654 sec
TOTAL(123)  : 0.095971 sec
TOTAL(23)   : 0.088500 sec
TOTAL(3)    : 0.011118 sec
***************************************
```

RamboPerSec
No longer sure this
makes sense… bug?

Exactly the same calculation,
exactly the same result

Throughput adding Rambo – i.e. ME, plus:
- add random-to-momenta mapping
- (CUDA only) add copy momenta to host
- (CUDA only) add copy weight to host

GPU: 6M in 0.09 sec ~ 7E7/sec
CPU: 6M in 18 sec ~ 3E5/sec
GPU is ~x200 a single CPU thread

MEPerSec throughput:
- ME calculation
- (CUDA only) copy MEs to host
GPU: 6M in 0.011 sec ~ 7E8/sec
CPU: 6M in 16 sec ~ 4E5/sec
GPU is ~x1500 a single CPU thread

C++/CPU
Single thread

```
./check.exe -p 16384 32 12
***************************************
NumIterations           = 12
NumThreadsPerBlock      = 32
NumBlocksPerGrid        = 16384
---------------------------------------
TotalEventsComputed     = 6291456
RamboEventsPerSec       = 3.243099e+06 sec^-1
MatrixElemEventsPerSec  = 3.962151e+05 sec^-1
***************************************
MeanMatrixElemValue     = 1.372152e-02 GeV^0
***************************************
0a ProcInit : 0.000467 sec
0b MemAlloc : 0.055191 sec
0c GenCreat : 0.000923 sec
1a GenSeed  : 0.000032 sec
1b GenRnGen : 0.321630 sec
2a RamboIni : 0.082014 sec
2b RamboFin : 1.857938 sec
3b SigmaKin : 15.878891 sec
4a DumpLoop : 0.019507 sec
8a CompStat : 0.028952 sec
9a GenDestr : 0.000104 sec
9b MemFree  : 0.001440 sec
9d DumpScrn : 0.000248 sec
9e DumpJson : 0.000002 sec
TOTAL       : 18.247335 sec
TOTAL(123)  : 18.140505 sec
TOTAL(23)   : 17.818844 sec
TOTAL(3)    : 15.878891 sec
***************************************
```

This is WIP!!! https://github.com/madgraph5/madgraph4gpu/issues/22
Device-to-host copies dominate because e+e- to μ+μ- MEs are too simple!

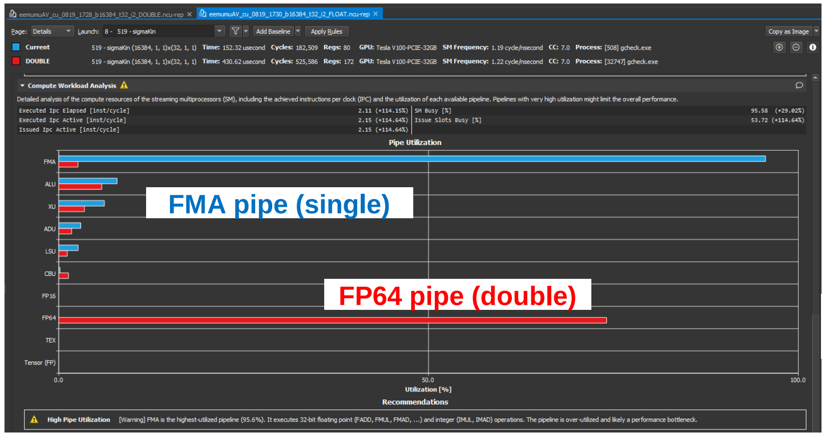
Nvidia GPUs – hackathons and profiling tools

- Some of us attended the http://gpuhackathons.org Sheffield event in July
  - Extremely beneficial for us, I highly recommend it for any CUDA developer!
  - Many thanks to our mentors at this specific event! https://gpuhack.shef.ac.uk
- This was useful also to understand performance in a benchmarking context
  - Which features of the GPU hardware are relevant (e.g. registers, FP32/64…)
  - Application profiling using two Nvidia tools, Nsight Systems and Nsight Compute
- Most plots in these slides are from tools/concepts learnt at the hackathon
  - Some detailed studies in https://github.com/madgraph5/madgraph4gpu/issues

One example relevant to GPU benchmarking: float vs. double (consumer vs. professional cards)

- By default MGaMC uses double-precision complex number arithmetic
  - This is required for physics precision… but it is worth checking this again!
- Moving from double to single precision (V100): gain a factor 2.4! (issue #5)
  - Intuitively, being able to use FP32 cores and not only FP64 cores gains a factor 2
  - In addition, single-precision reduces register pressure and increases occupancy

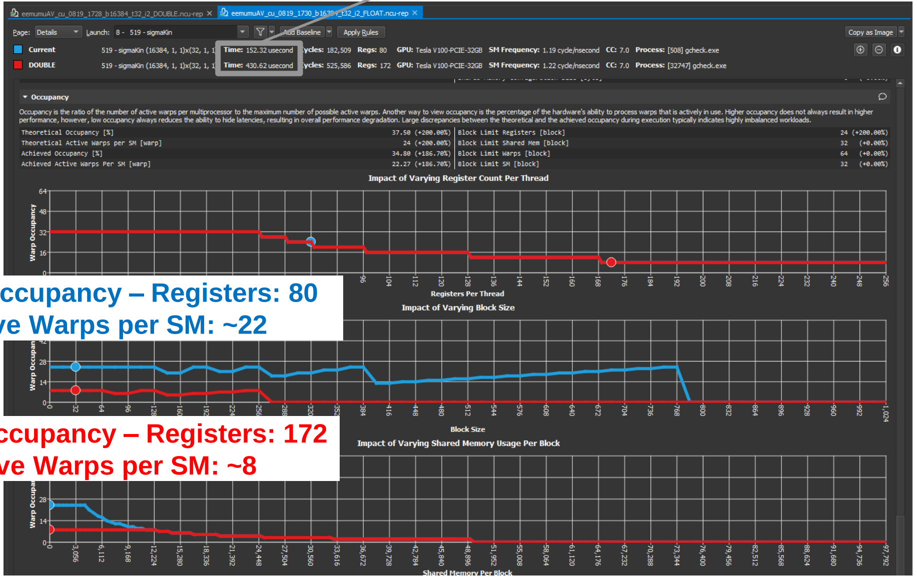
single is 2.4 faster than double (sigmakin = ME kernel)



```
eemumuAV_cu_0819_1728_b16384_t32_i2_DOUBLE.ncu-rep   eemumuAV_cu_0819_1730_b16384_t32_i2_FLOAT.ncu-rep
Page: Details   Launch: 8 - 519 - sigmaKin   Add Baseline   Apply Rules   Copy as Image
Current  519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 152.32 usecond  Cycles: 182,509  Regs: 80   GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [508] gcheck.exe
DOUBLE   519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 430.62 usecond  Cycles: 525,586  Regs: 172  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.22 cycle/nsecond  CC: 7.0  Process: [32747] gcheck.exe

Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle]      2.11 (+114.15%) | SM Busy [%]           95.58  (+29.02%)
Executed Ipc Active [inst/cycle]       2.15 (+114.64%) | Issue Slots Busy [%]  53.72 (+114.64%)
Issued Ipc Active [inst/cycle]         2.15 (+114.64%) |

Recommendations
High Pipe Utilization  [Warning] FMA is the highest-utilized pipeline (95.6%). It executes 32-bit floating point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.
```

FMA pipe (single)

FP64 pipe (double)



```
eemumuAV_cu_0819_1728_b16384_t32_i2_DOUBLE.ncu-rep   eemumuAV_cu_0819_1730_b16384_t32_i2_FLOAT.ncu-rep
Page: Details   Launch: 8 - 519 - sigmaKin   Add Baseline   Apply Rules   Copy as Image
Current  519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 152.32 usecond  Cycles: 182,509  Regs: 80   GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [508] gcheck.exe
DOUBLE   519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 430.62 usecond  Cycles: 525,586  Regs: 172  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.22 cycle/nsecond  CC: 7.0  Process: [32747] gcheck.exe

Occupancy
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.
Theoretical Occupancy [%]              37.50 (+200.00%) | Block Limit Registers [block]   24 (+200.00%)
Theoretical Active Warps per SM [warp]    24 (+200.00%) | Block Limit Shared Mem [block]  32   (+0.00%)
Achieved Occupancy [%]                 34.80 (+186.70%) | Block Limit Warps [block]       64   (+0.00%)
Achieved Active Warps Per SM [warp]    22.27 (+186.70%) | Block Limit SM [block]          32   (+0.00%)
```

Impact of Varying Register Count Per Thread
Warp Occupancy
Registers Per Thread
Impact of Varying Block Size
Block Size
Impact of Varying Shared Memory Usage Per Block
Shared Memory Per Block

Single Occupancy – Registers: 80
Active Warps per SM: ~22

Double Occupancy – Registers: 172
Active Warps per SM: ~8

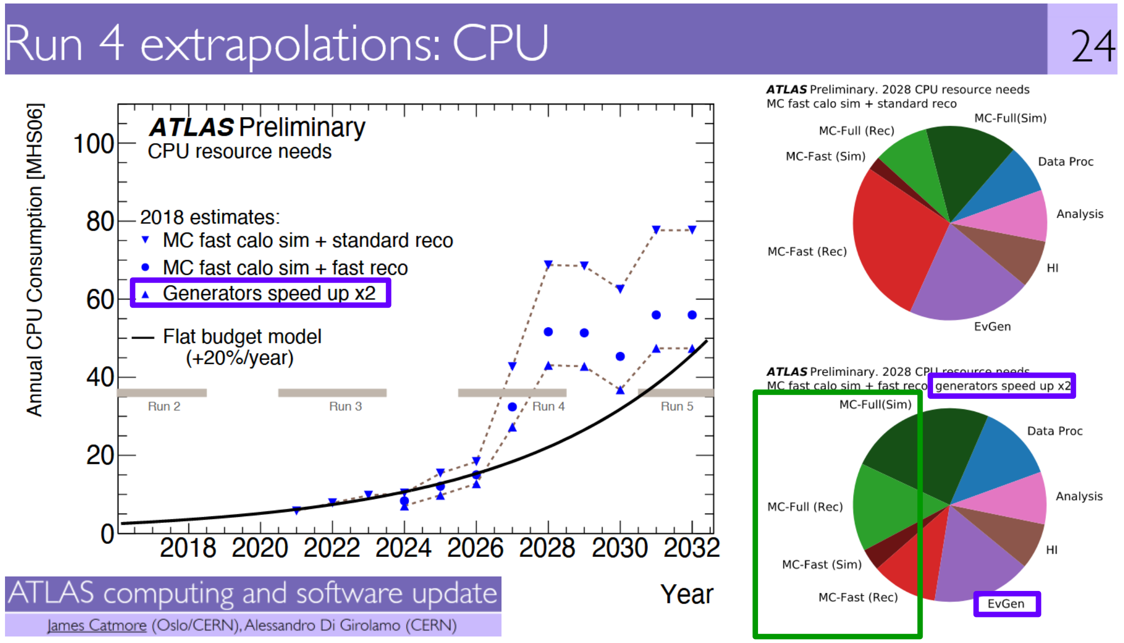
Outlook: a BMK container for MGaMC on GPU?

- *Creating a container for e+e- to μ+μ- should be relatively easy and fast*
  - *But first, we need a realistic unweighting workflow and its throughput metric!*
  - This can be interesting to compare GPUs, but is of little relevance to WLCG…
- Creating an LHC-type container needs more development progress first
  - At least a more complete backport of ME calculations to code-generating code
  - A more realistic sampler is also needed for complex processes like gg to $t\bar{t}gg$
  - LHC proton collisions also need a port of PDFs to CUDA
    - But one could also prepare a simpler benchmark before for a gg to $t\bar{t}gg$ process
- *We do not use /cvmfs at all for the moment* (no released/installed version)
  - We would need to build CUDA executables in the CI and embed those in docker
  - There is no input data, this is essentially a pure "GEN" type of workload
- *There is a synergy between the MG and BMK projects, let's stay in touch!*
  - Example: we both need to get hold and test AMD and Intel GPUs at some point!

Backup slides

MC generators and *HL-LHC software and computing*

- One of the main issues (not the only one!): *HL-LHC computing resource gap*
  - Generator performance must also keep up with higher physics precision



Run 4 extrapolations: CPU 24

ATLAS computing and software update

James Catmore (Oslo/CERN), Alessandro Di Girolamo (CERN)

WLCG meeting with LHCC referees, Feb. 2020

CPU cost of generators as a fraction of WLCG CPU resources: for ATLAS, ballpark of 10%-20% (for CMS, this is lower)

ATLAS considers an overall generator speedup by a factor x2 as an R&D goal for HL-LHC

Side note: the higher the fraction of **negative-weight events from MC generators**, the higher the CPU cost of MC simulation, MC reconstruction and analysis (need more MC events)

- Many other challenges, including:
  - WLCG software workloads on non-traditional resources (HPCs, GPUs…)
  - Funding and careers (especially at the theory/experiment/computing interface)

A few performance comparisons

- A few studies from August 2020 (essentially the same code as today)
  - Measurements on lxbatch: sensitive to external load, reproducible within ~10%
  - Extensive discussion of numbers and Nsight Compute profiler plots on https://github.com/madgraph5/madgraph4gpu/issues
  - General approach: change a #define switch to a non-default value in the code
    - Now kept cuComplex and float switches, but hardcoded AOSOA and “local” memory

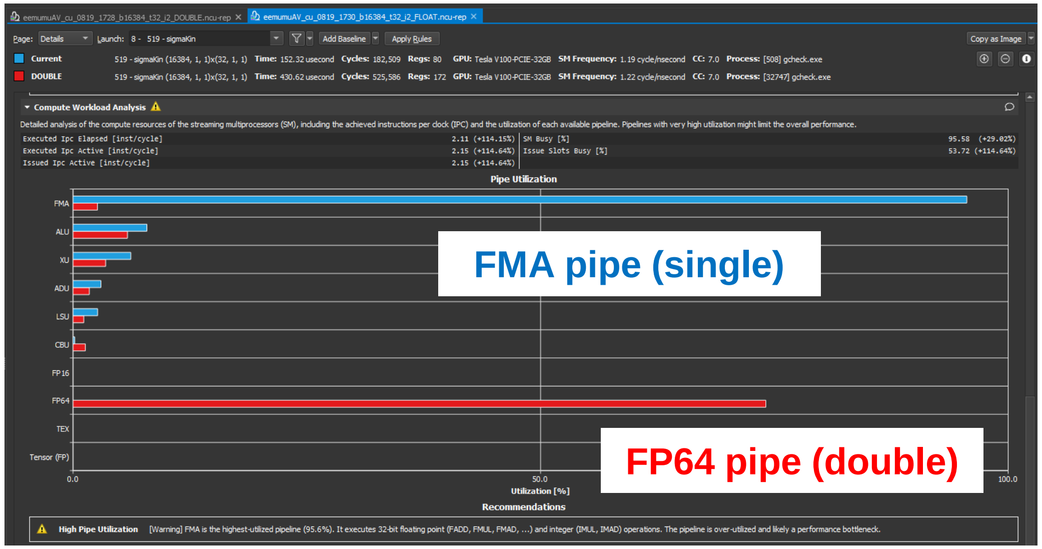
- From double precision to single precision: gain a factor 2.4! (issue #5)
  - FMA (FP32) used instead of FP64 pipe; fewer registers hence higher occupancy

- Memory layout for 4-momenta (issue #16): study “requests” vs “sectors”
  - Default: AOSOA with 4 events per array (four 8-byte doubles: 32-byte cache line)
    - AOS ~7% slower: memory not coalesced, #sectors (transactions) factor 4 higher
    - SOA ~2% slower: slightly higher number of registers
  - Side result of this study: improve helicity filtering to reduce #requests (issue #24)

- Memory for wavefunctions in ixxx/oxxx/FV_xx (issue #7): default is “local”
  - Lose 70% with “global” (become memory-bound!), also lose 35% with “shared”

- From thrust::complex to cuComplex: lose ~5% (issue #6)
  - Require execution of higher number of instructions (something to fix?)

FP: double vs single precision (issue #5)



```
Current   519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 152.32 usecond  Cycles: 182,509  Regs: 80   GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [508] gcheck.exe
DOUBLE    519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 430.62 usecond  Cycles: 525,586  Regs: 172  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.22 cycle/nsecond  CC: 7.0  Process: [32747] gcheck.exe
```

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

```
Executed Ipc Elapsed [inst/cycle]      2.11 (+114.15%) | SM Busy [%]            95.58  (+29.02%)
Executed Ipc Active [inst/cycle]       2.15 (+114.64%) | Issue Slots Busy [%]   53.72 (+114.64%)
Issued Ipc Active [inst/cycle]         2.15 (+114.64%) |
```

Recommendations

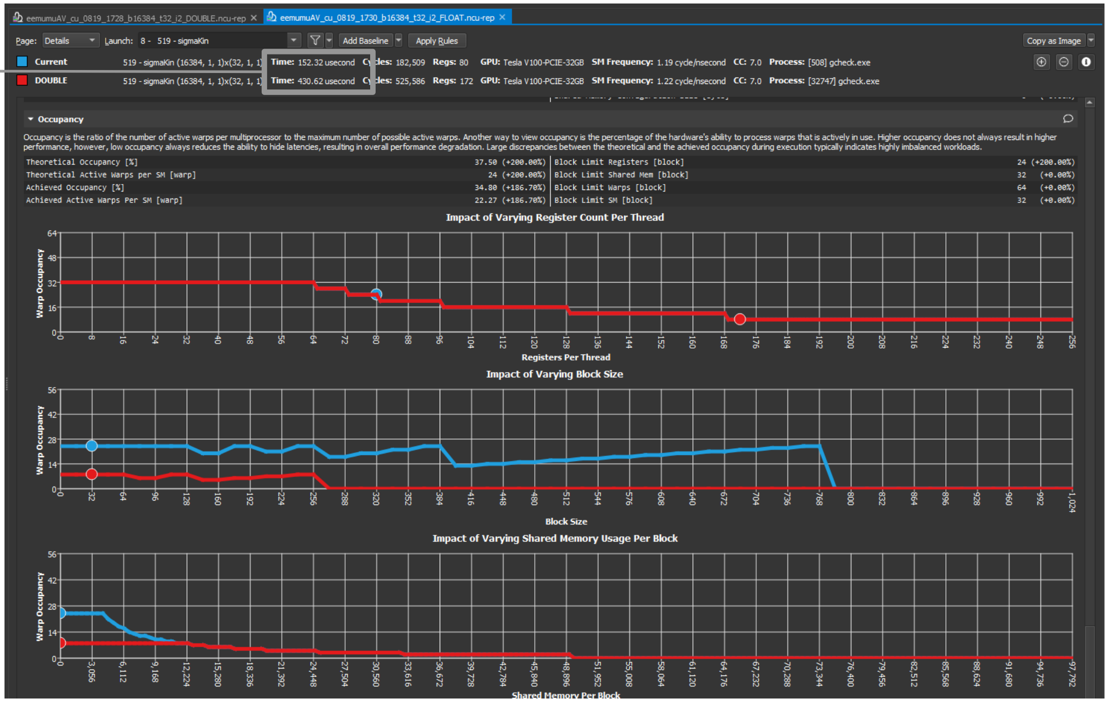
High Pipe Utilization [Warning] FMA is the highest-utilized pipeline (95.6%). It executes 32-bit floating point (FADD, FMUL, FMAD, ...) and integer (IMUL, IMAD) operations. The pipeline is over-utilized and likely a performance bottleneck.

- Two main effects
  - Exploit the FP32 units (unused otherwise!)
  - Fewer registers hence higher warp occupancy

single is 2.4 faster than double (sigmakin = ME kernel)

Single Occupancy
Registers: 80
Active Warps per SM: ~22

Double Occupancy
Registers: 172
Active Warps per SM: ~8


eemumuAV_cu_0819_1728_b16384_t32_i2_DOUBLE.ncu-rep
eemumuAV_cu_0819_1730_b16384_t32_i2_FLOAT.ncu-rep
Page: Details
Launch: 8 - 519 - sigmaKin
Add Baseline
Apply Rules
Copy as Image
Occupancy
Impact of Varying Register Count Per Thread
Warp Occupancy
Registers Per Thread
Impact of Varying Block Size
Block Size
Impact of Varying Shared Memory Usage Per Block
Shared Memory Per Block

```
Current   519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 152.32 usecond  Cycles: 182,509  Regs: 80   GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.19 cycle/nsecond  CC: 7.0  Process: [508] gcheck.exe
DOUBLE    519 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 430.62 usecond  Cycles: 525,586  Regs: 172  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.22 cycle/nsecond  CC: 7.0  Process: [32747] gcheck.exe
```

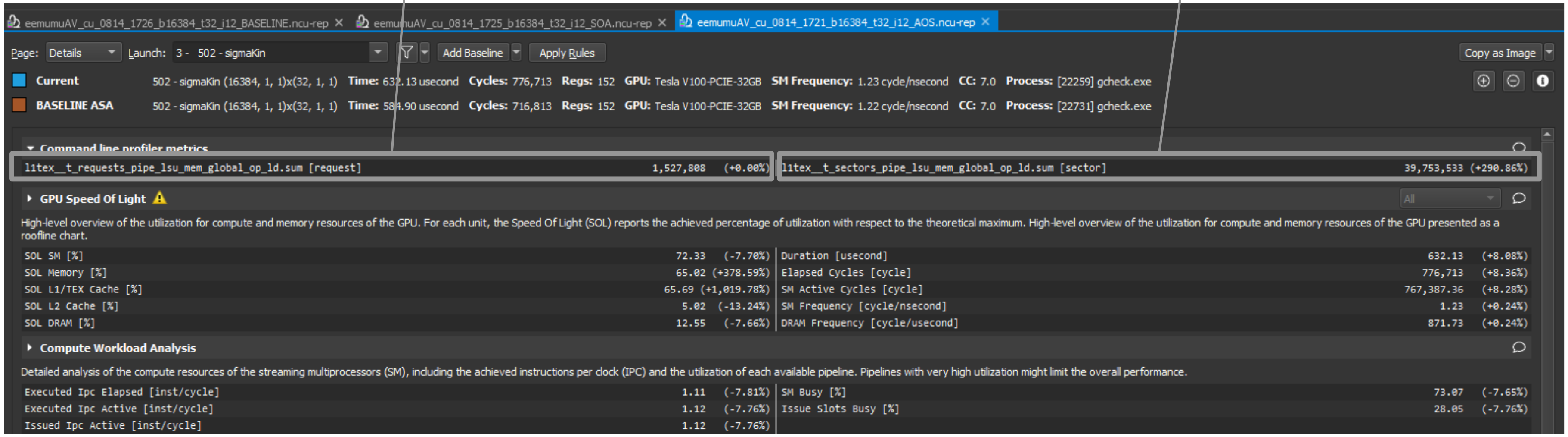
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

```
Theoretical Occupancy [%]                37.50 (+200.00%) | Block Limit Registers [block]    24 (+200.00%)
Theoretical Active Warps per SM [warp]      24 (+200.00%) | Block Limit Shared Mem [block]   32   (+0.00%)
Achieved Occupancy [%]                   34.80 (+186.70%) | Block Limit Warps [block]        64   (+0.00%)
Achieved Active Warps Per SM [warp]      22.27 (+186.70%) | Block Limit SM [block]           32   (+0.00%)
```

4-momenta memory: AOSOA vs AOS (issue #16)

Number of "requests":
- It is the same for AOSOA or AOS (the same information needs to be retrieved…)
- Later on, the number show here was dramatically decreased by improving helicity filtering (issue #24)

Number of "sectors" (transactions):
- It is a factor 4 higher for AOS than for AOSOA
- One roundtrip (AOSOA, coalesced) vs four roundtrips (AOS, not coalesced) to retrieve four 8-byte doubles from a 32-byte cache line

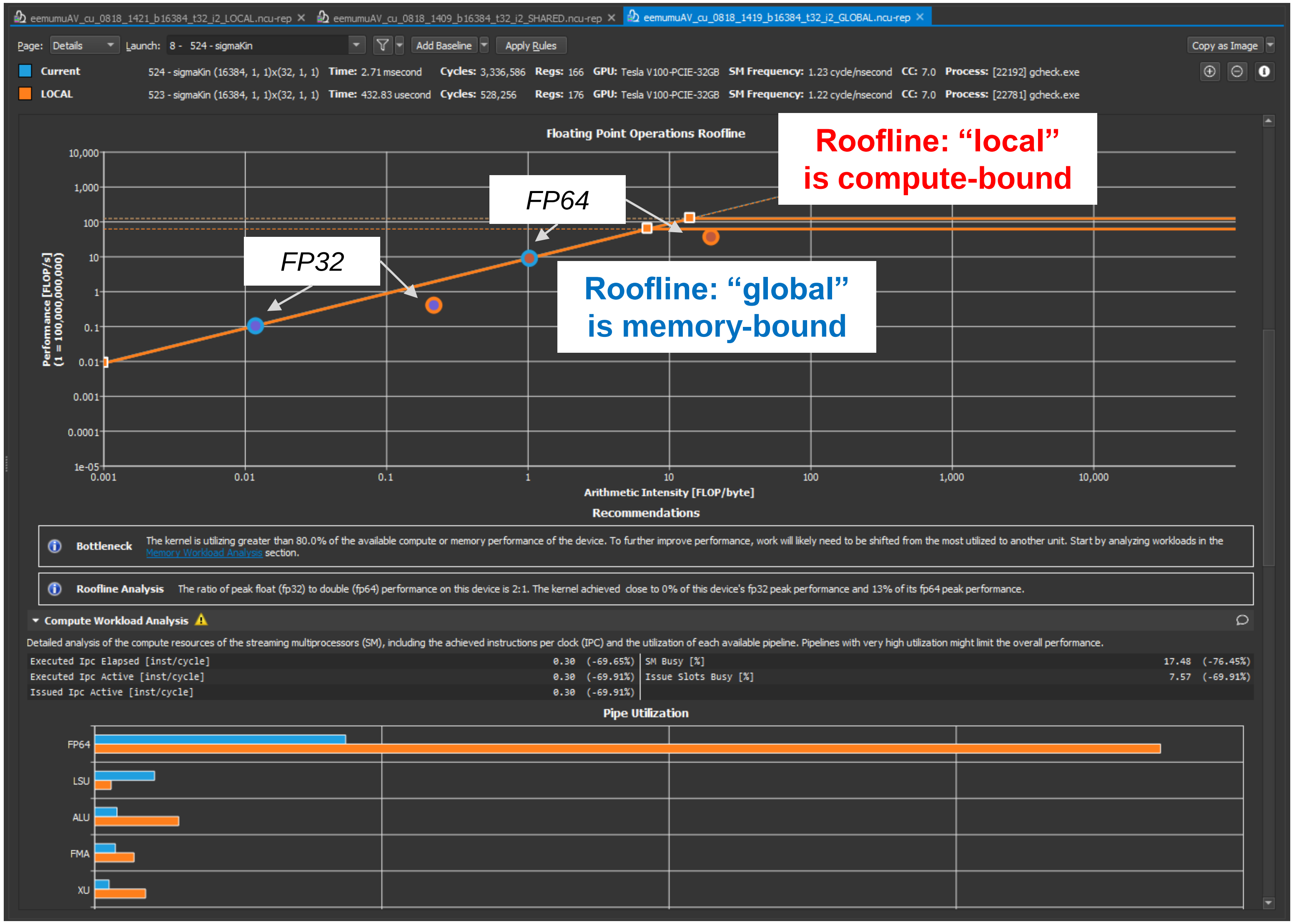


eemumuAV_cu_0814_1726_b16384_t32_i12_BASELINE.ncu-rep
eemumuAV_cu_0814_1725_b16384_t32_i12_SOA.ncu-rep
eemumuAV_cu_0814_1721_b16384_t32_i12_AOS.ncu-rep
Page: Details
Launch: 3 - 502 - sigmaKin
Add Baseline
Apply Rules
Copy as Image
All

```
🟦 Current        502 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 632.13 usecond  Cycles: 776,713  Regs: 152  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.23 cycle/nsecond  CC: 7.0  Process: [22259] gcheck.exe
🟫 BASELINE ASA   502 - sigmaKin (16384, 1, 1)x(32, 1, 1)  Time: 584.90 usecond  Cycles: 716,813  Regs: 152  GPU: Tesla V100-PCIE-32GB  SM Frequency: 1.22 cycle/nsecond  CC: 7.0  Process: [22731] gcheck.exe

▾ Command line profiler metrics
l1tex__t_requests_pipe_lsu_mem_global_op_ld.sum [request]     1,527,808   (+0.00%) | l1tex__t_sectors_pipe_lsu_mem_global_op_ld.sum [sector]     39,753,533 (+290.86%)

▸ GPU Speed Of Light ⚠
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]                 72.33    (-7.70%) | Duration [usecond]                  632.13   (+8.08%)
SOL Memory [%]             65.02  (+378.59%) | Elapsed Cycles [cycle]              776,713  (+8.36%)
SOL L1/TEX Cache [%]       65.69 (+1,019.78%)| SM Active Cycles [cycle]         767,387.36  (+8.28%)
SOL L2 Cache [%]            5.02   (-13.24%) | SM Frequency [cycle/nsecond]           1.23  (+0.24%)
SOL DRAM [%]               12.55    (-7.66%) | DRAM Frequency [cycle/usecond]       871.73  (+0.24%)

▸ Compute Workload Analysis
Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.
Executed Ipc Elapsed [inst/cycle]    1.11   (-7.81%) | SM Busy [%]              73.07   (-7.65%)
Executed Ipc Active [inst/cycle]     1.12   (-7.76%) | Issue Slots Busy [%]     28.05   (-7.76%)
Issued Ipc Active [inst/cycle]       1.12   (-7.76%) |
```

A few useful links:
- https://developer.nvidia.com/blog/using-nsight-compute-to-inspect-your-kernels/
- https://docs.nvidia.com/nsight-compute/2019.5/NsightComputeCli/index.html#nvprof-metric-comparison
- https://stackoverflow.com/questions/60535867/what-is-a-transaction-and-a-request-in-the-gld-transactions-per-request-metric

Wavefunction memory: “local” vs “global” (issue #7)



eemumuAV_cu_0818_1421_b16384_t32_i2_LOCAL.ncu-rep ✕ eemumuAV_cu_0818_1409_b16384_t32_i2_SHARED.ncu-rep ✕ eemumuAV_cu_0818_1419_b16384_t32_i2_GLOBAL.ncu-rep ✕

Page: Details Launch: 8 - 524 - sigmaKin

🟦 Current	524 - sigmaKin (16384, 1, 1)x(32, 1, 1)	Time: 2.71 msecond	Cycles: 3,336,586	Regs: 166	GPU: Tesla V100-PCIE-32GB	SM Frequency: 1.23 cycle/nsecond	CC: 7.0	Process: [22192] gcheck.exe
🟧 LOCAL	523 - sigmaKin (16384, 1, 1)x(32, 1, 1)	Time: 432.83 usecond	Cycles: 528,256	Regs: 176	GPU: Tesla V100-PCIE-32GB	SM Frequency: 1.22 cycle/nsecond	CC: 7.0	Process: [22781] gcheck.exe

Add Baseline
Apply Rules
Copy as Image
Floating Point Operations Roofline
Performance [FLOP/s] (1 = 100,000,000,000)
10,000
1,000
100
10
1
0.1
0.01
0.001
0.0001
1e-05
0.001
0.01
0.1
1
10
100
1,000
10,000
Arithmetic Intensity [FLOP/byte]
FP64
FP32
Roofline: “local” is compute-bound
Roofline: “global” is memory-bound

Recommendations

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

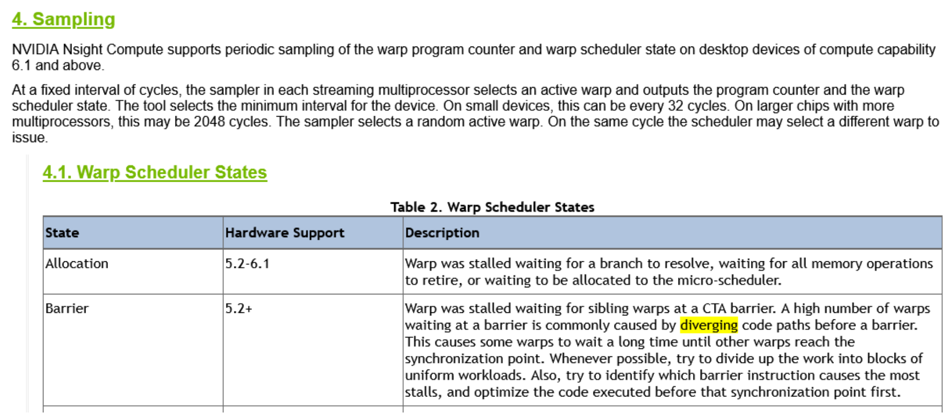
Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved close to 0% of this device's fp32 peak performance and 13% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

```
Executed Ipc Elapsed [inst/cycle]      0.30  (-69.65%) | SM Busy [%]             17.48  (-76.45%)
Executed Ipc Active [inst/cycle]       0.30  (-69.91%) | Issue Slots Busy [%]     7.57  (-69.91%)
Issued Ipc Active [inst/cycle]         0.30  (-69.91%) |
```

No evidence for “thread divergence” (issue #25)



4. Sampling

NVIDIA Nsight Compute supports periodic sampling of the warp program counter and warp scheduler state on desktop devices of compute capability 6.1 and above.

At a fixed interval of cycles, the sampler in each streaming multiprocessor selects an active warp and outputs the program counter and the warp scheduler state. The tool selects the minimum interval for the device. On small devices, this can be every 32 cycles. On larger chips with more multiprocessors, this may be 2048 cycles. The sampler selects a random active warp. On the same cycle the scheduler may select a different warp to issue.

4.1. Warp Scheduler States

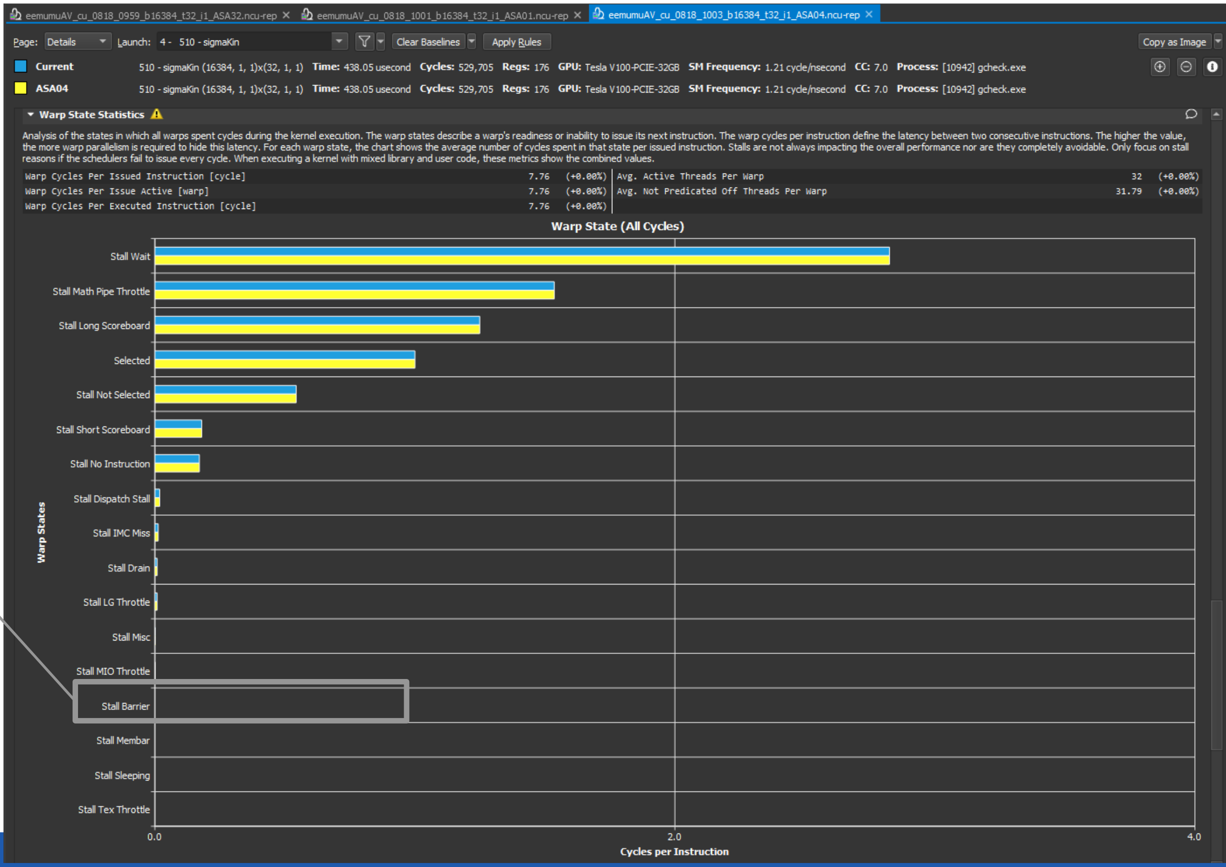
Table 2. Warp Scheduler States

State	Hardware Support	Description
Allocation	5.2-6.1	Warp was stalled waiting for a branch to resolve, waiting for all memory operations to retire, or waiting to be allocated to the micro-scheduler.
Barrier	5.2+	Warp was stalled waiting for sibling warps at a CTA barrier. A high number of warps waiting at a barrier is commonly caused by diverging code paths before a barrier. This causes some warps to wait a long time until other warps reach the synchronization point. Whenever possible, try to divide up the work into blocks of uniform workloads. Also, try to identify which barrier instruction causes the most stalls, and optimize the code executed before that synchronization point first.

https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#statistical-sampler

- Nsight Compute:
  - Stall Barrier = 0

- Similarly, no indication for thread divergence from Nsight System



eemumuAV_cu_0818_0959_b16384_t32_i1_ASA32.ncu-rep
eemumuAV_cu_0818_1001_b16384_t32_i1_ASA01.ncu-rep
eemumuAV_cu_0818_1003_b16384_t32_i1_ASA04.ncu-rep
Page: Details
Launch: 4 - 510 - sigmaKin
Clear Baselines
Apply Rules
Copy as Image

🟦 **Current** 510 - sigmaKin (16384, 1, 1)x(32, 1, 1) **Time:** 438.05 usecond **Cycles:** 529,705 **Regs:** 176 **GPU:** Tesla V100-PCIE-32GB **SM Frequency:** 1.21 cycle/nsecond **CC:** 7.0 **Process:** [10942] gcheck.exe

🟨 **ASA04** 510 - sigmaKin (16384, 1, 1)x(32, 1, 1) **Time:** 438.05 usecond **Cycles:** 529,705 **Regs:** 176 **GPU:** Tesla V100-PCIE-32GB **SM Frequency:** 1.21 cycle/nsecond **CC:** 7.0 **Process:** [10942] gcheck.exe

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

```
Warp Cycles Per Issued Instruction [cycle]     7.76   (+0.00%) | Avg. Active Threads Per Warp              32   (+0.00%)
Warp Cycles Per Issue Active [warp]            7.76   (+0.00%) | Avg. Not Predicated Off Threads Per Warp  31.79   (+0.00%)
Warp Cycles Per Executed Instruction [cycle]   7.76   (+0.00%) |
```