Accelerated Monte Carlo Particle Generators for the LHC

(MC@GPU)

Gergely Gábor Barnaföldi[1]& Máté Ferenc Nagy-Egri[1,2]

[1]Wigner RCP of the HAS, Budapest, Hungary

[2]Eötvös Loránd University, Budapest, Hungary

OUTLINE

- MC generators in high-energy heavy-ion physics
- The biggest data challenge: LHC & WLCG with GPUs?
- GPU based PRNG for MC generators
- Performance tests by GPU based MC
- What can we learn from pp MC simulations?
- Outlook

MC generators in high-energy collisions

Why do we need Monte Carlo generators?

There are problems with no analytical expression, no closed form, or no deterministic description, like:

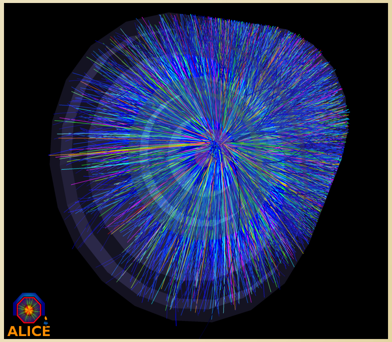
- stohastic processes (independent events)
- numerical (multi-D) integration
- optimalization
- ...and many more during the next days :-)

Solution & errors

Random sampling of numerical results

Error estimation by standard devitaion

Fast random numbers → Computing & IT

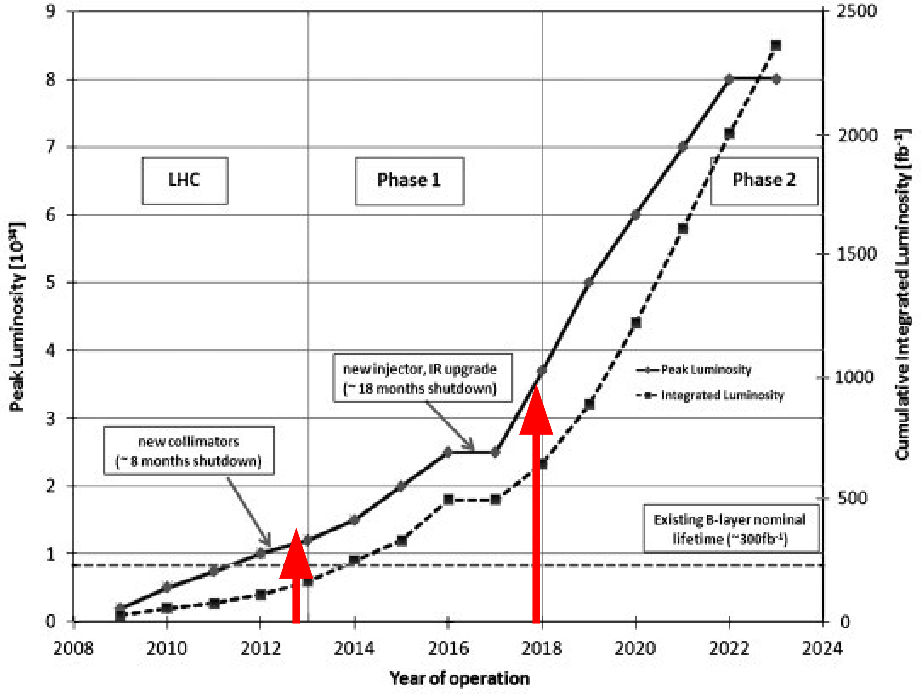


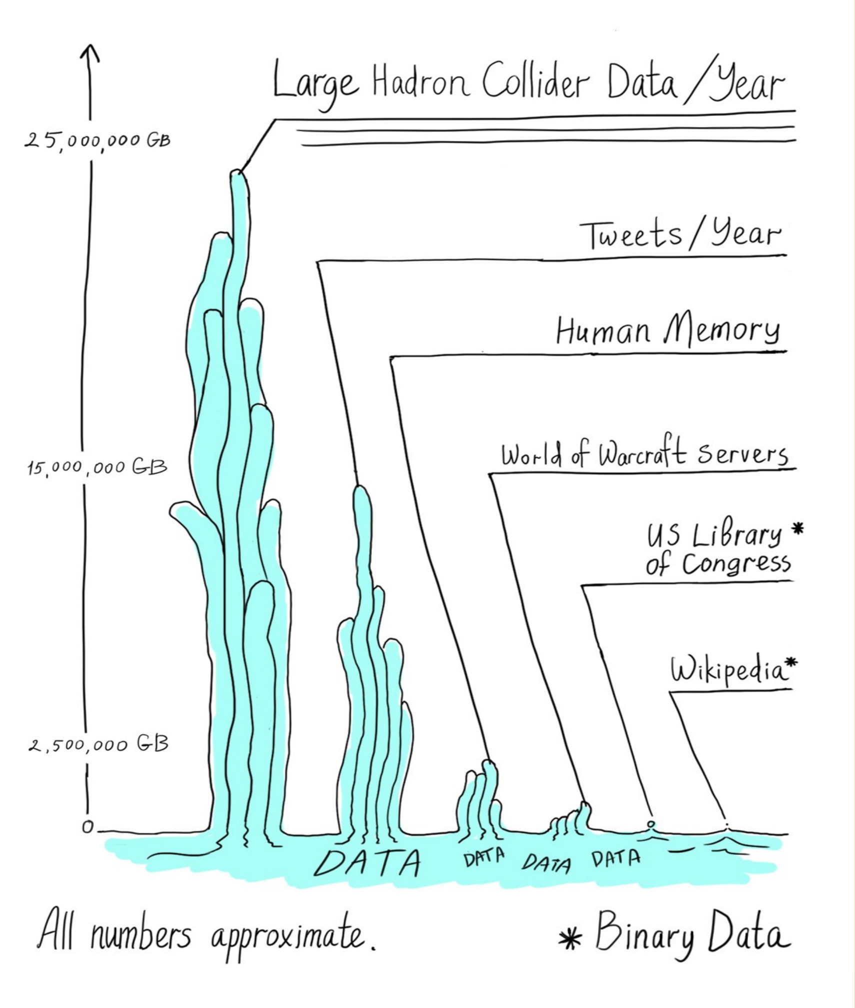
The biggest data challenge: LHC

WLCG – Worldwide LHC Computing GRID:

15-20 Petabytes data per year

...and more after LHC upgrades

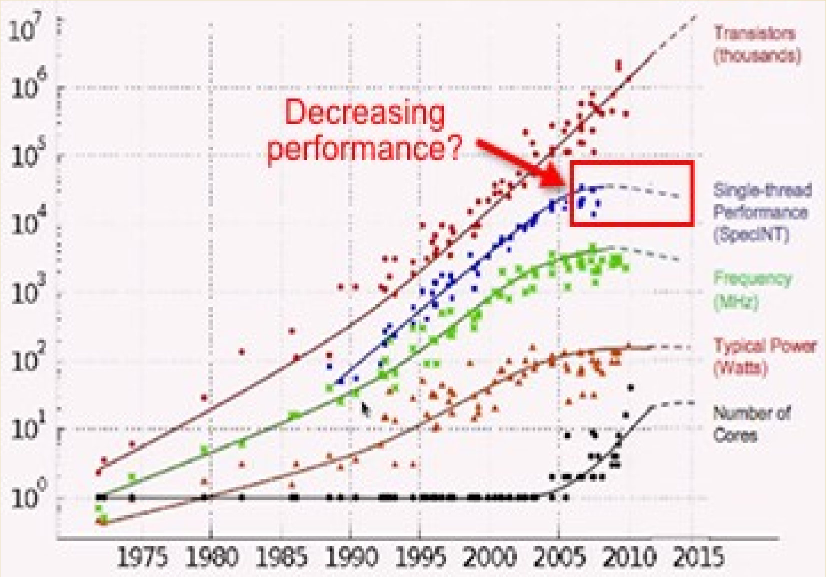




Fast computing=parallel computing

- Moore's law:

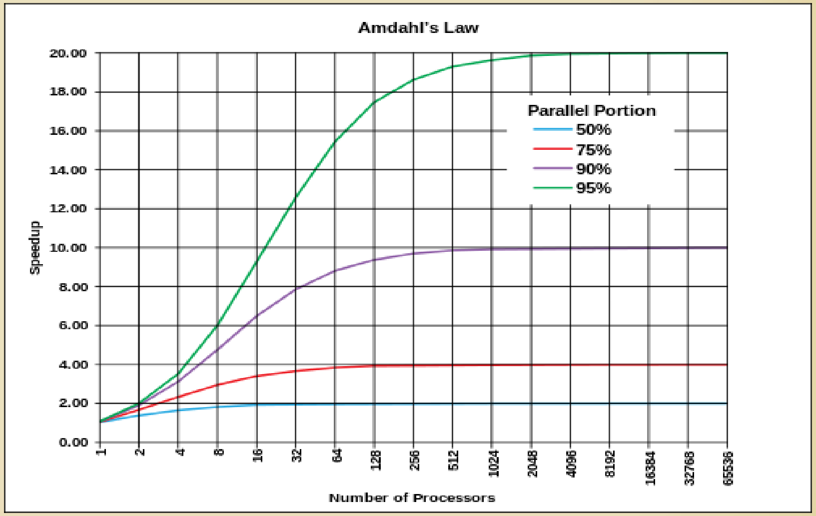
Every 2^{nd} year the number of transistors (integrated circuits) are doubled in computing hardwares.



- Amdalh's law:

The theoretical speedup is given by the portion of parallelizable program, p, & number of processors, N, is:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}.$$



How to improve the WLCG resources

WLCG:

- Critical points are the number and performance of the WNs
- There are multicore machines with single thread.
- If there are free multicores or GPU resources, improvement can be made at the sofrware and middleware level (cheap).
- Certainly, there is a budget issue as well.

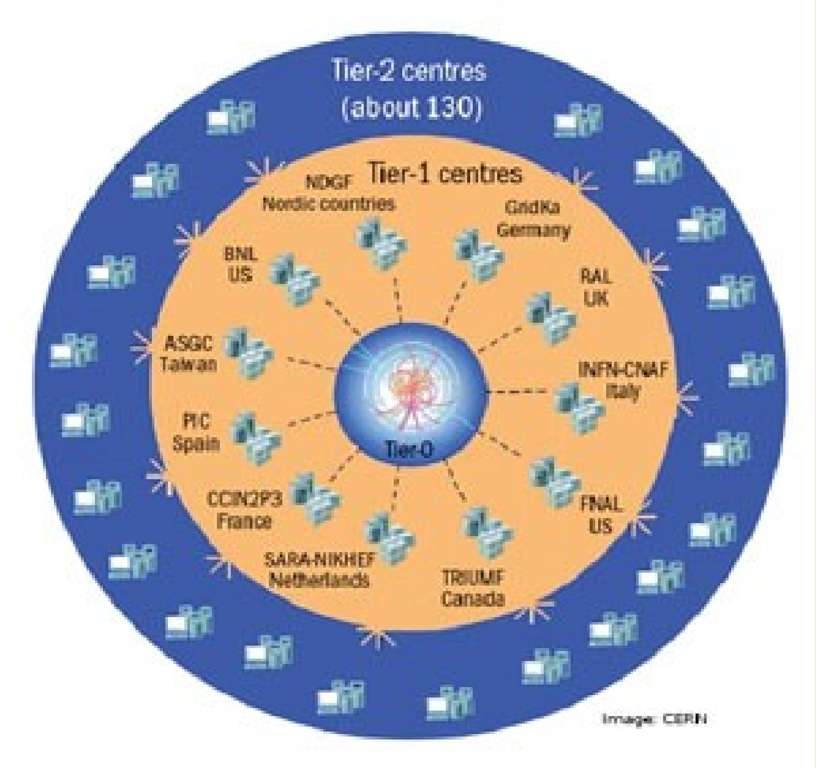
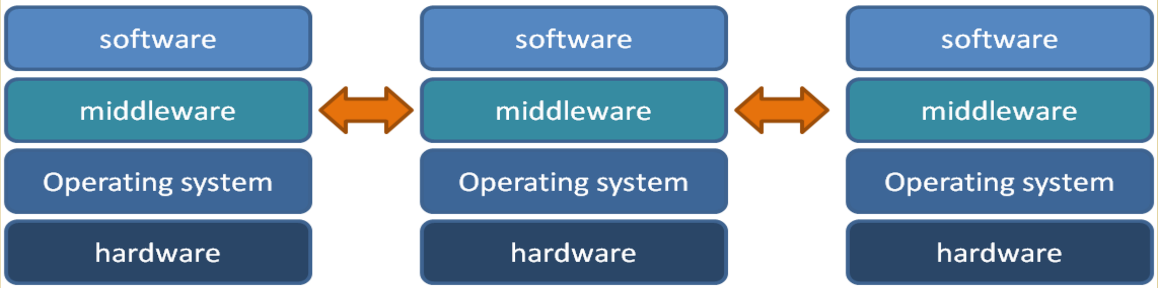


Image: CERN



software | software | software

middleware ↔ middleware ↔ middleware

Operating system | Operating system | Operating system

hardware | hardware | hardware

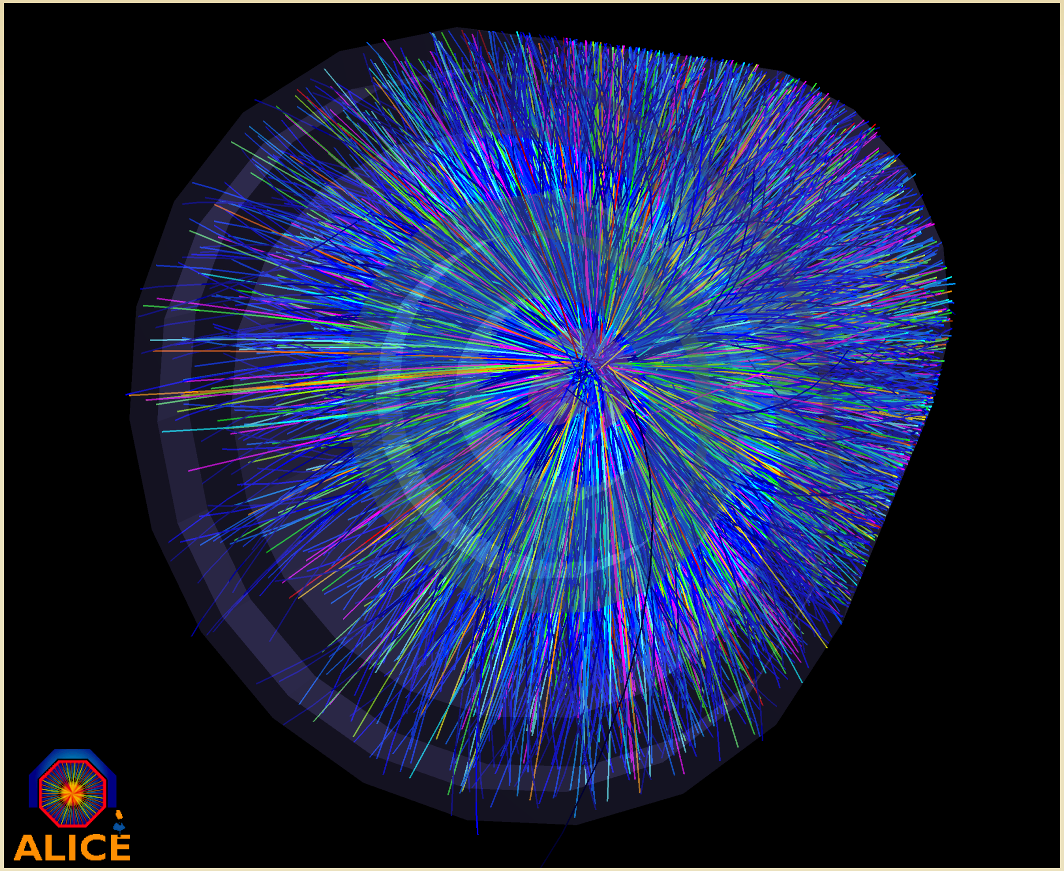
When is the moment to use GPUs?

No direct answer!

- Pilot study to define parameters to be optimized
- Need for large scale and large-large scale computing
- Have time (5-10 times more code development)
- Manpower high-level (close to hardware) programming
- $$$$$$

What has been done so far to help us? – without CUDA, etc...

- Several libs & toolkits (BLAS, FFTW, CUBLAS, CUFFT)
- Wrappers (C, FORTRAN → CUDA)
- OpenCL standards (Ati, NVidia)
- Mathematica, MatLab (with GPU support)



GPU based PRNG
for
MC event generators

GPU based PRNG for MC event generators

- Software frameworks

CERN

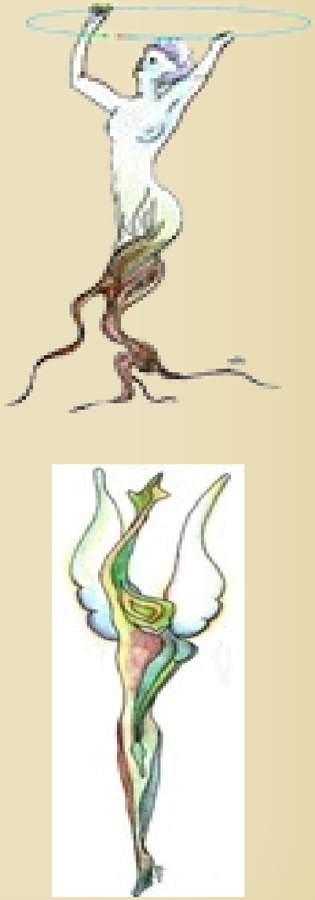
- OS: SLC 2.6.32-279.1.1.el6.x86_64
- Graphics: fglrx 9.002 (Catalyst 12.10)
- GCC: 4.4.6 20120305 (Red Hat 4.4.6-4)
- OpenCL: 1.2 AMD APP SDK 2.8

ALICE

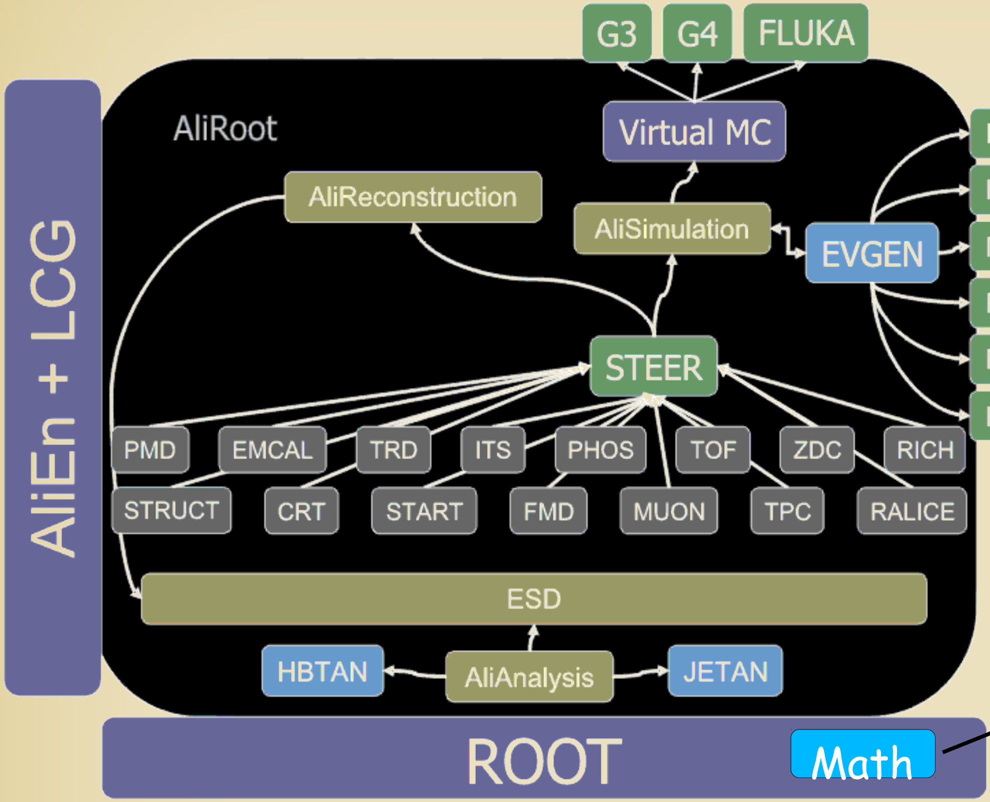
- Aliroot: v5-03-73-AN
- Root: v5-34-02
- Geant3: v1-14

PRNG tester

- Dieharder: 3.31.1



GPU based PRNG for MC event generators



AliEn + LCG

AliRoot

G3
G4
FLUKA

Virtual MC

AliReconstruction

AliSimulation

EVGEN

ISAJET
HIJING
MEVSIM
PYTHIA6
PDF
HBTP

STEER

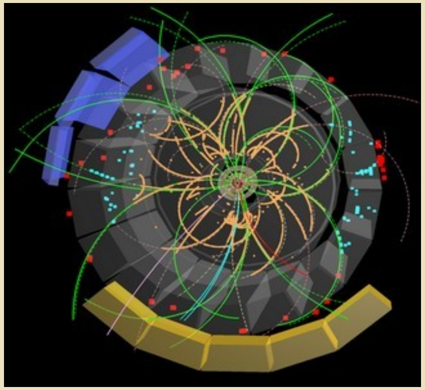
PMD
EMCAL
TRD
ITS
PHOS
TOF
ZDC
RICH

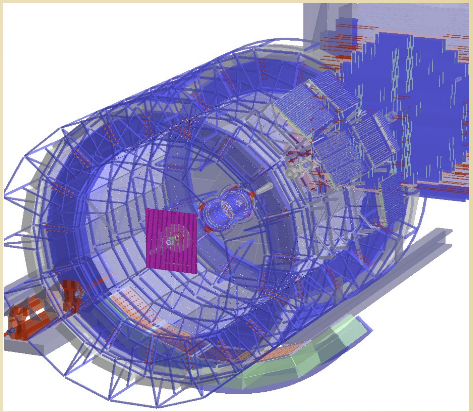
STRUCT
CRT
START
FMD
MUON
TPC
RALICE

ESD

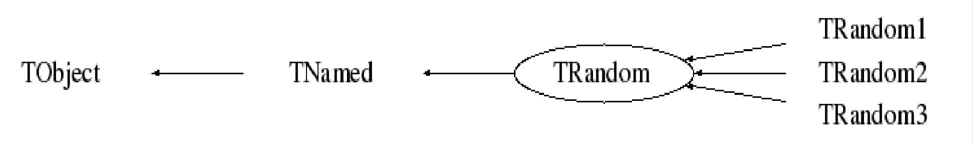
HBTAN
AliAnalysis
JETAN

ROOT
Math

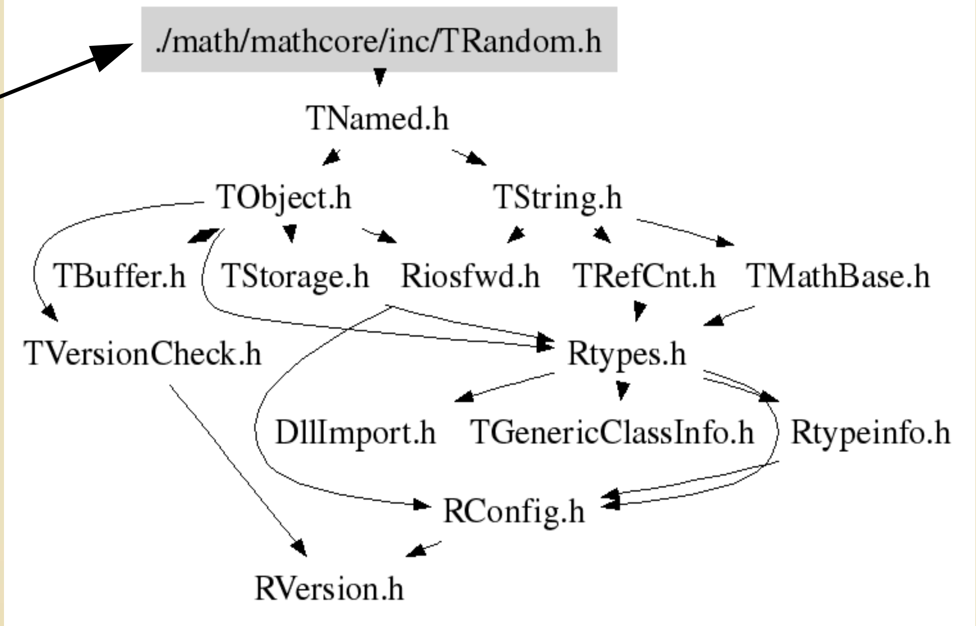




AliRoot framework for ALICE data simualtion, reconstruction, analysis



TObject
TNamed
TRandom
TRandom1
TRandom2
TRandom3



./math/mathcore/inc/TRandom.h
TNamed.h
TObject.h
TString.h
TBuffer.h
TStorage.h
Riosfwd.h
TRefCnt.h
TMathBase.h
TVersionCheck.h
Rtypes.h
DllImport.h
TGenericClassInfo.h
Rtypeinfo.h
RConfig.h
RVersion.h

GPU based PRNG for MC event generators

- The tested PRNG codes

  Trandom1 (RANLUX)

  TRandom2 (Tausworthe)

  TRandom3

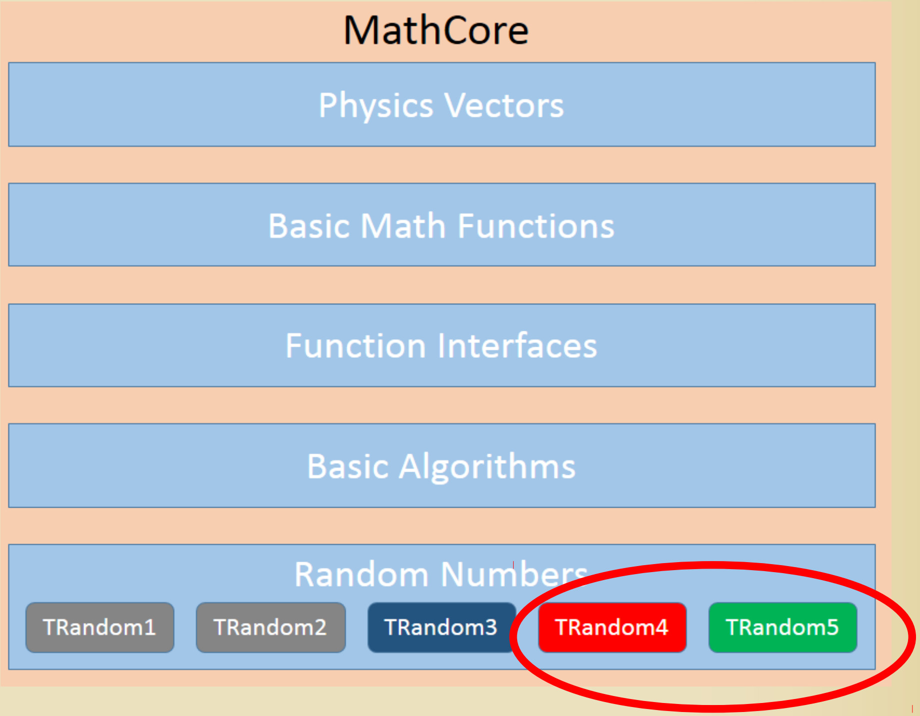
  - Original CPU based Mersenne Twister) algorithm

  TRandom4

  - CPU/GPU based SFMT (SIMD-oriented Fast Mersenne Twister) algorithm

  TRandom5

  - CPU/GPU based MWC64X algorithm



GPU based PRNG for MC event generators

- From the *user* side
  - Installation:

    Driver + OpenCL (SDK)

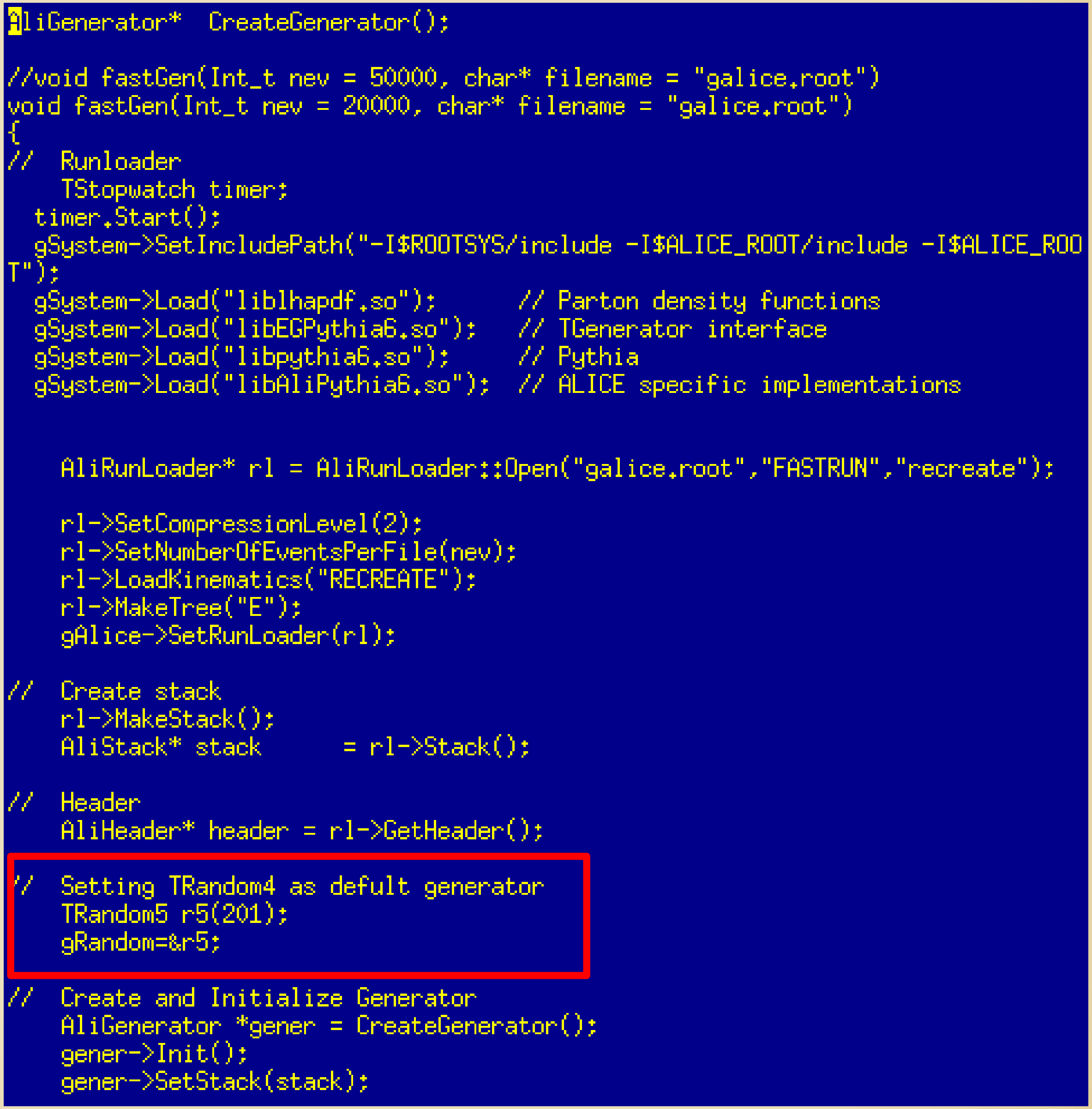
    Pre-complied modules
  - Usage:

    TRandomX, can be take as a regular PRNG.

    CPU/GPU run can be choosen via parameters:

    GPU: parameter > 200

    CPU: parameter < 200



```
AliGenerator*  CreateGenerator();

//void fastGen(Int_t nev = 50000, char* filename = "galice.root")
void fastGen(Int_t nev = 20000, char* filename = "galice.root")
{
//  Runloader
    TStopwatch timer;
  timer.Start();
  gSystem->SetIncludePath("-I$ROOTSYS/include -I$ALICE_ROOT/include -I$ALICE_ROO
T");
  gSystem->Load("liblhapdf.so");      // Parton density functions
  gSystem->Load("libEGPythia6.so");   // TGenerator interface
  gSystem->Load("libpythia6.so");     // Pythia
  gSystem->Load("libAliPythia6.so");  // ALICE specific implementations


    AliRunLoader* rl = AliRunLoader::Open("galice.root","FASTRUN","recreate");

    rl->SetCompressionLevel(2);
    rl->SetNumberOfEventsPerFile(nev);
    rl->LoadKinematics("RECREATE");
    rl->MakeTree("E");
    gAlice->SetRunLoader(rl);

//  Create stack
    rl->MakeStack();
    AliStack* stack      = rl->Stack();

//  Header
    AliHeader* header = rl->GetHeader();

//  Setting TRandom4 as defult generator
    TRandom5 r5(201);
    gRandom=&r5;

//  Create and Initialize Generator
    AliGenerator *gener = CreateGenerator();
    gener->Init();
    gener->SetStack(stack);
```

GPU based PRNG for MC event generators

- Behind the scene

  – TRandom4 & TRandom5

  – No single random number generation only in 500k blocks

  – RAM buffer is for random numbers.

  – Only speeddown is the 'stack depth check'.

  – Copy work from buffer is by the CPU.

  – Due to OpenCL platform this works on both CPU/GPU

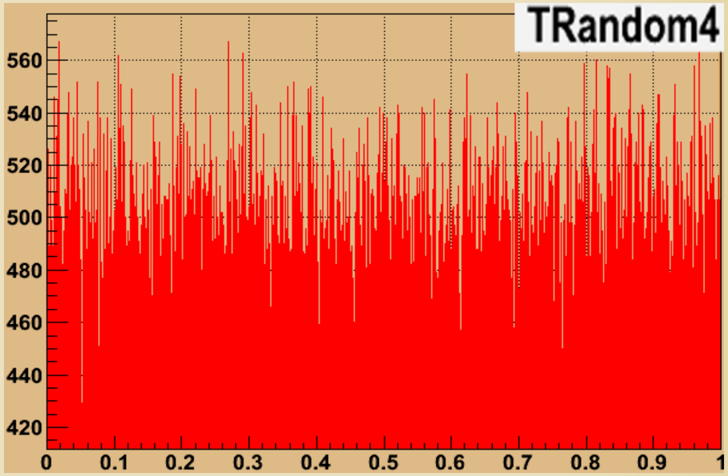
- Constructor

  – It contains all tasks

    - Platform check
    - Context creation
    - Device info
    - Kernel compilation
    - Command queue
    - Buffer allocation
    - Sending random seeds to devices
    - Tread ID settings

The PRNG quality test

How good is a PRNG?



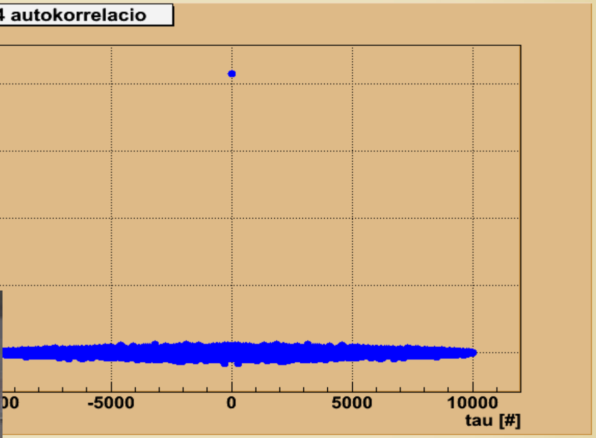
- 1st simply randomness tests can be carried out via taking the numbers and calculation mometns, etc.

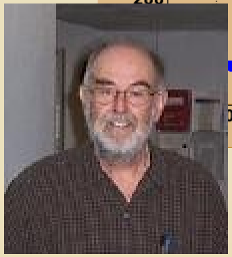
$$C(\tau) = \sum_n f(n) f(n+\tau)$$

- 2nd test is the autocorrelation

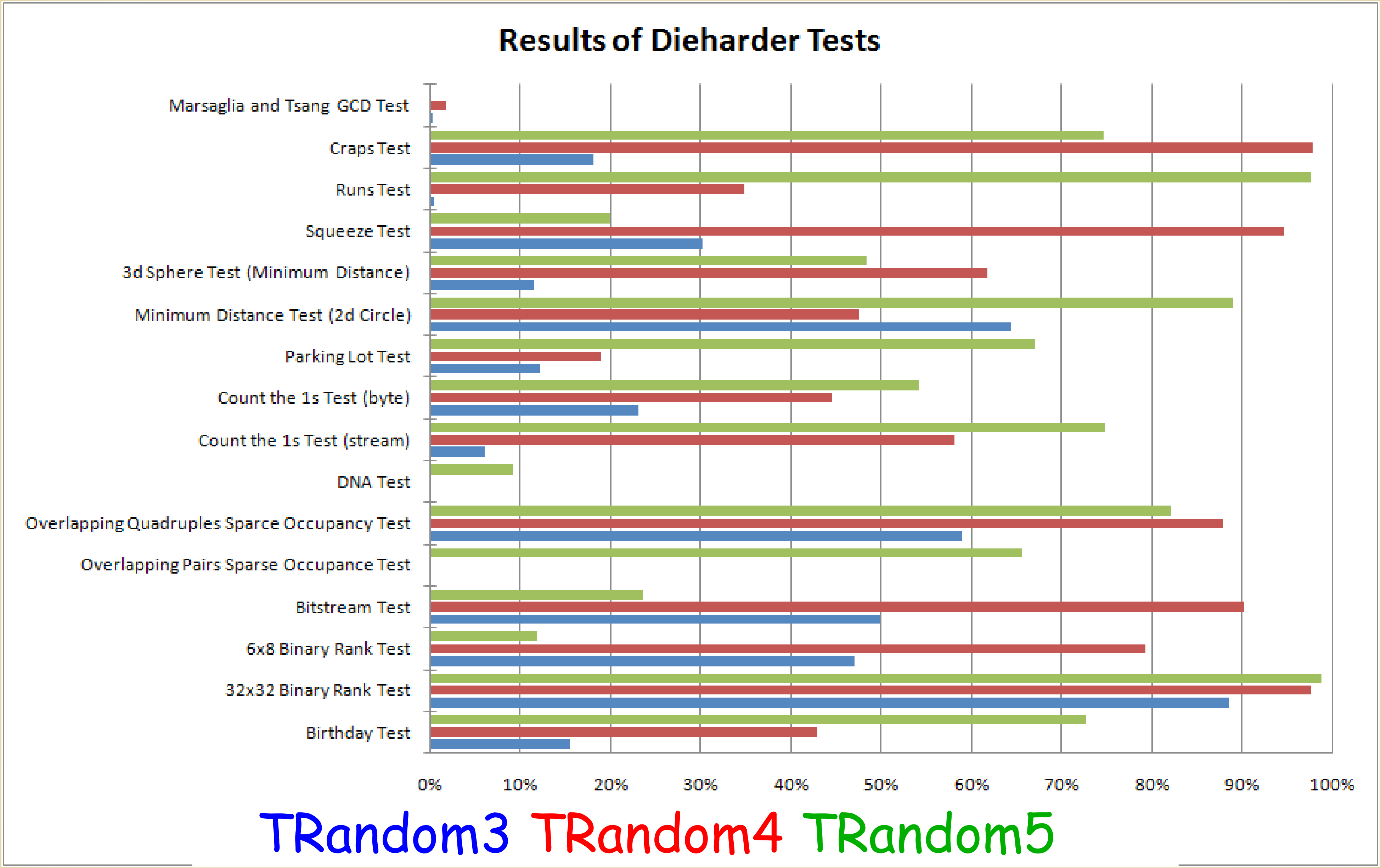
- 3rd Complex test where PRNGs can die hared is the „Diehard test“:

  R.G. Brown, D. Eddelbüttel, D. Bauer: Diehard 3.31.1 a Kolmogorov-Smirov test based open source random number statistical test suite package, based on G. Marsaglia „Diehard battery of test of randomness“.





The PRNG quality test



TRandom3 TRandom4 TRandom5

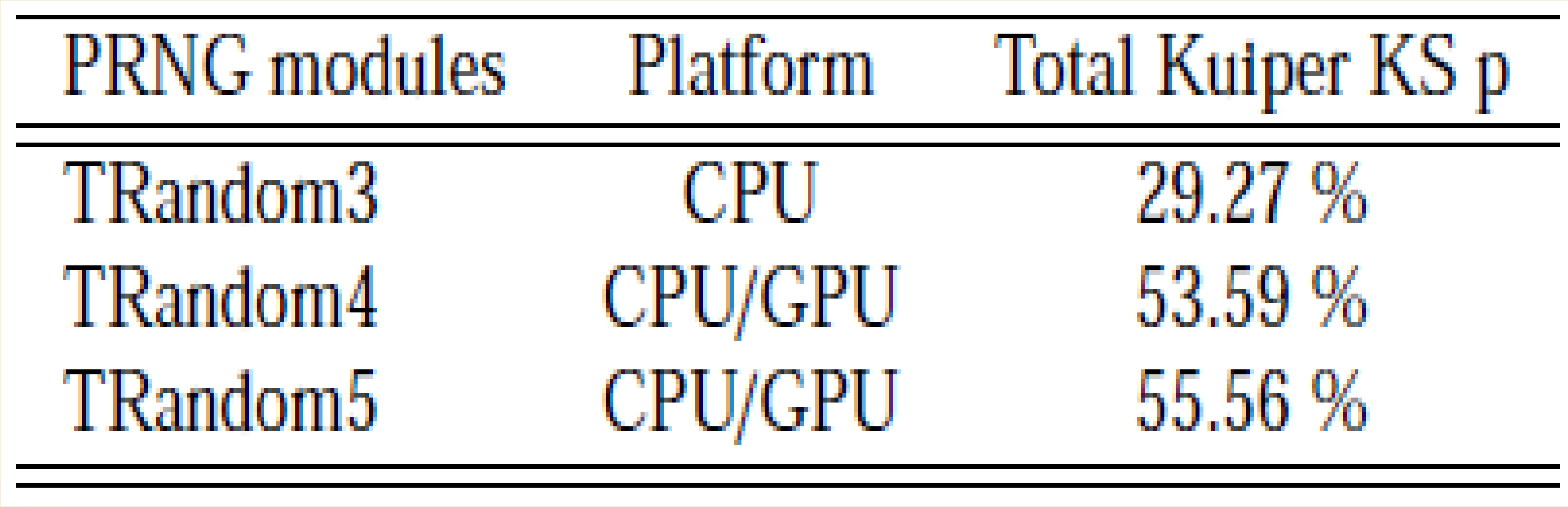
The PRNG quality test

- Summary of the DieHard quality tests of PRNGs

TRandom3 – Original CPU based Mersenne Twister

TRandom4 – CPU/GPU based SFMT (SIMD-oriented Fast MT)

TRandom5 – CPU/GPU based MWC64X algorithm



PRNG modules	Platform	Total Kuiper KS p
TRandom3	CPU	29.27 %
TRandom4	CPU/GPU	53.59 %
TRandom5	CPU/GPU	55.56 %

Performance

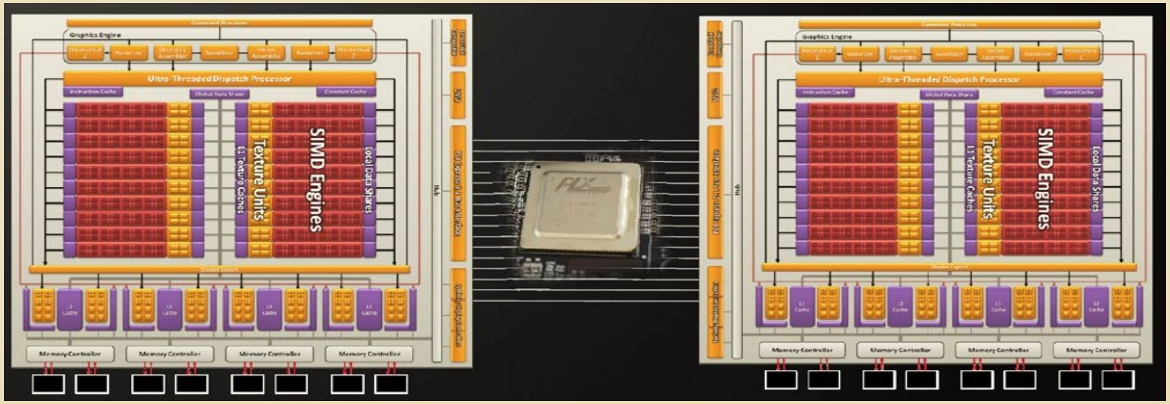
Performance tests by GPU based MC

- Hardware framework

gpu001 at GPU Laboratory of the Wigner RCP

  - MB: ASUS P6T6 PCIExpress 2.0x16
  - CPU: Core i7 920 (2.76 Ghz, 8 KB cache)
  - Memory: 12GB DDR3 (1333 MHz)
  - HDD: 1 TB
  - GPU: 3 pcs. ATi Radeon HD5970

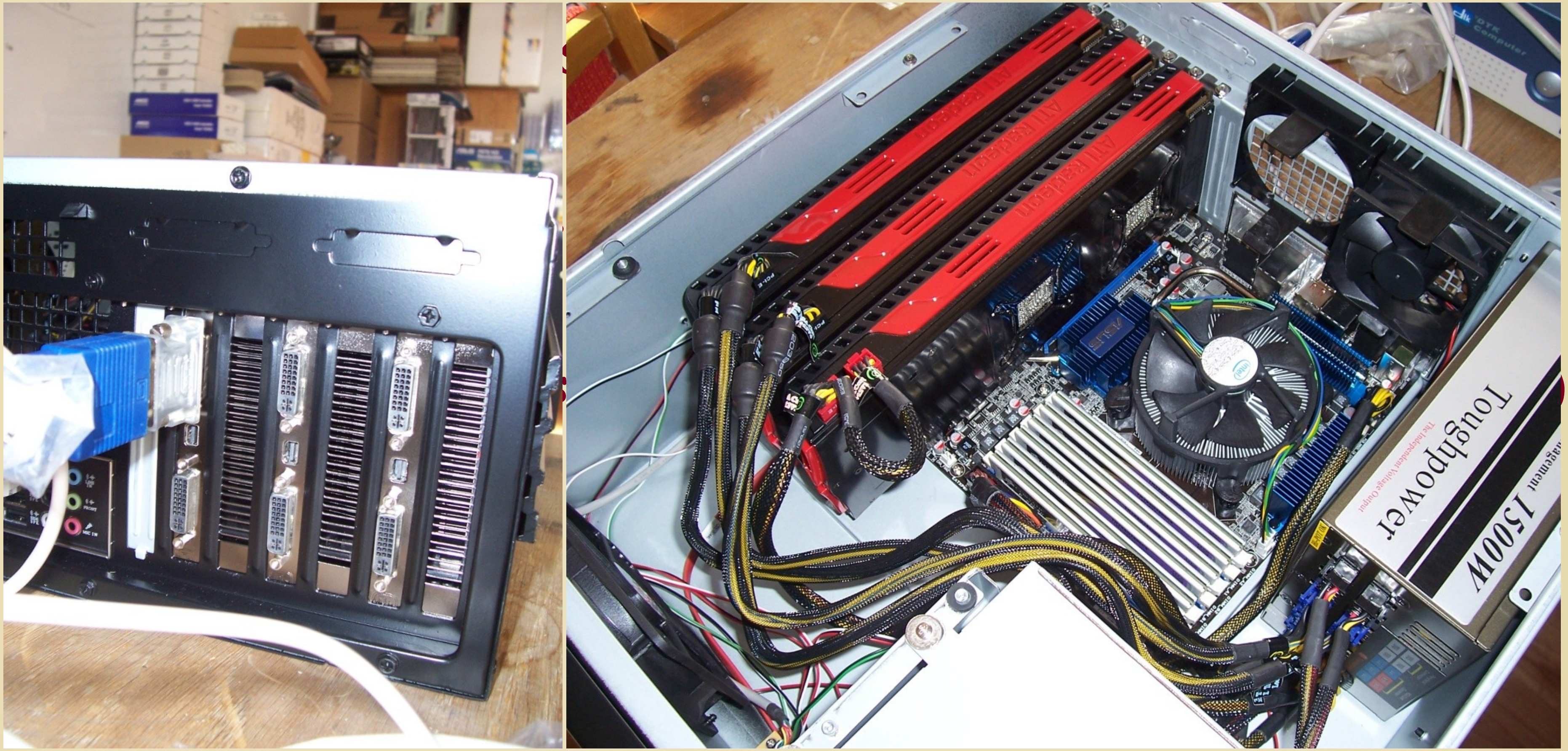
    (2 GPUs, 735 MHz, 1+1 GB GDDR, 4.64 TFlop)



Performance tests by GPU based MC

- Hardware framework

gpu001 at GPU Laboratory of the Wigner RCP



The main question is: How about SPEED?

- Levels of speedtest

  Kernel speed

  - Real geneation time of a PRNG in CPU or in GPU.

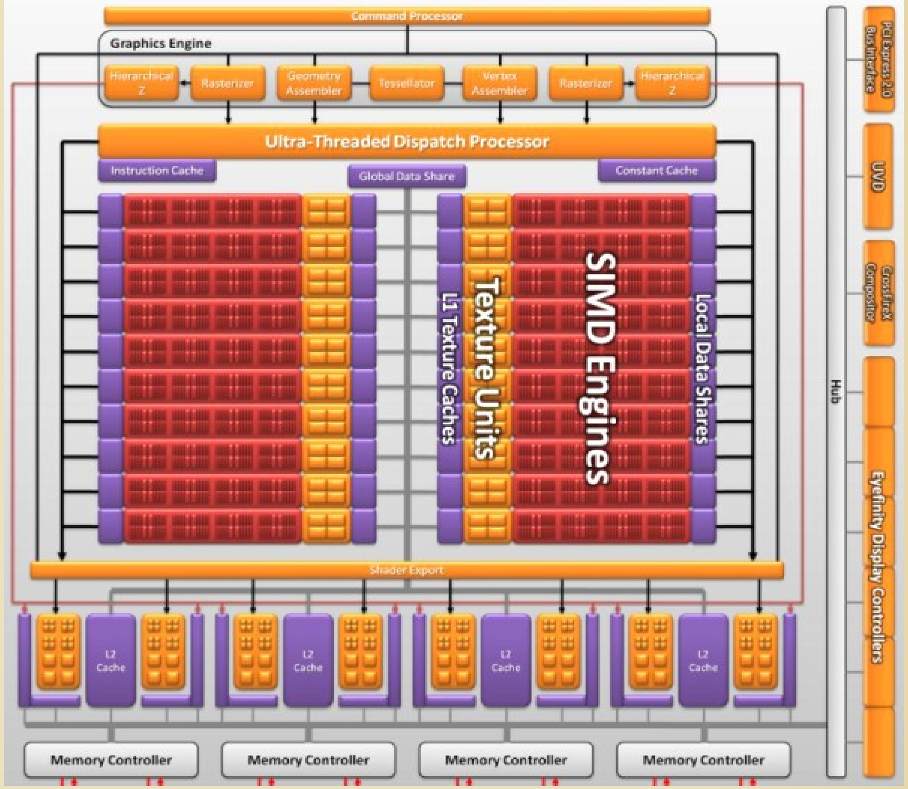
  Total speed

  - Generation time of the PRNGs within the proper program framework

  Real speed

  - The above two, but with real (V)RAM usage.

  Here we used a 200 million event sample!

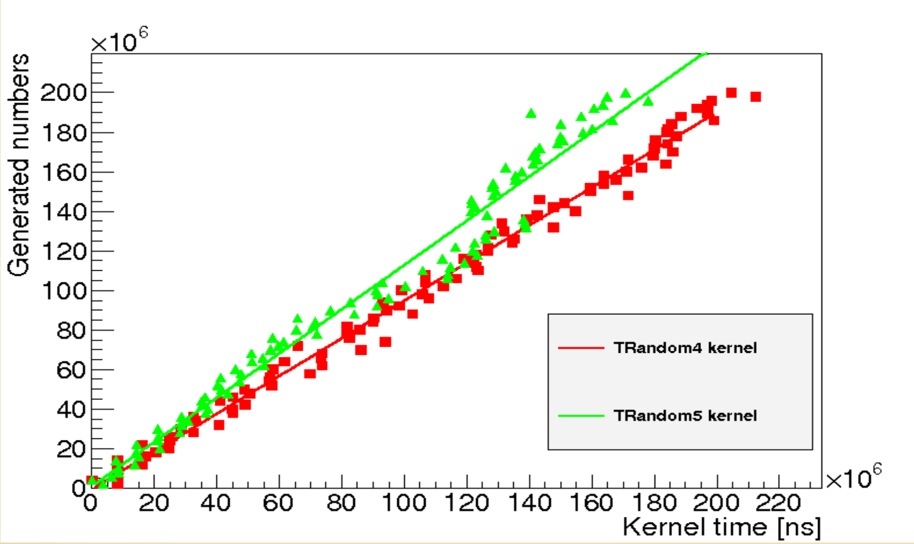


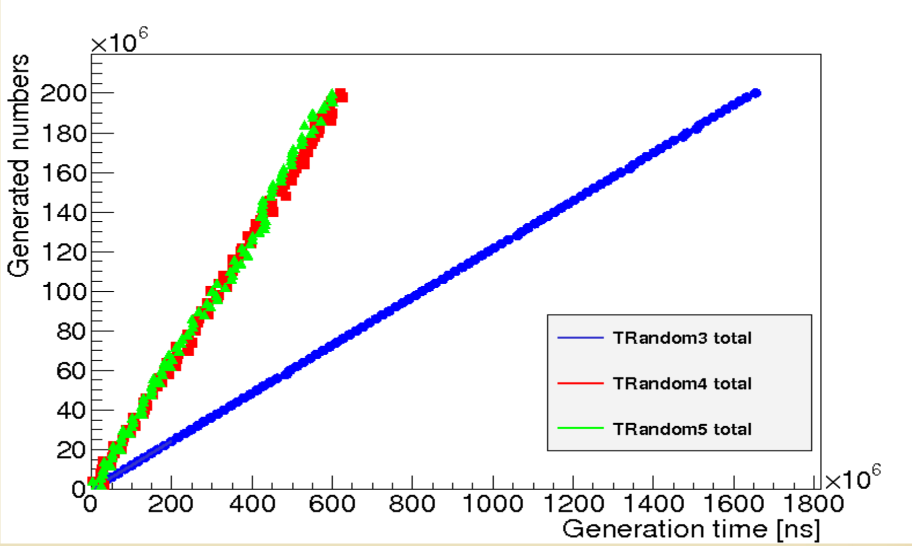
SPEED without writing (V)RAM

Kernel time

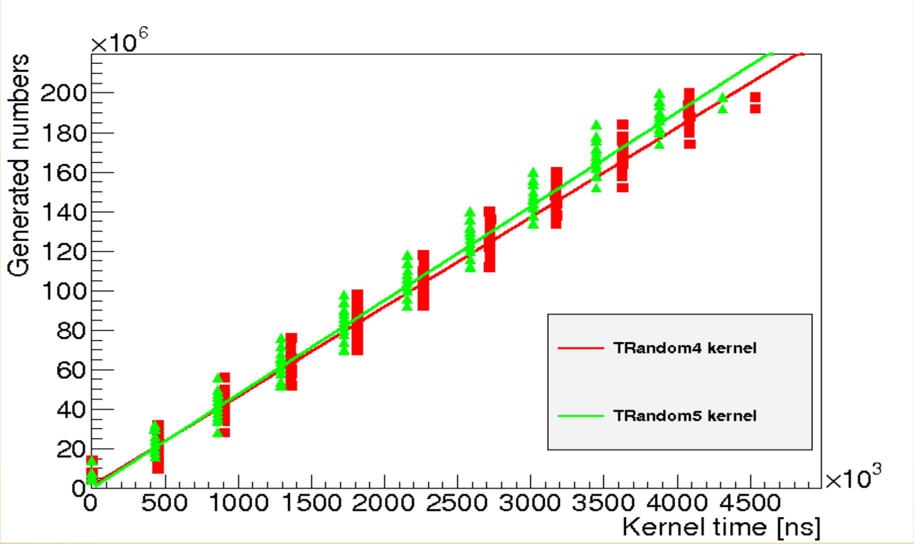
Full calculation

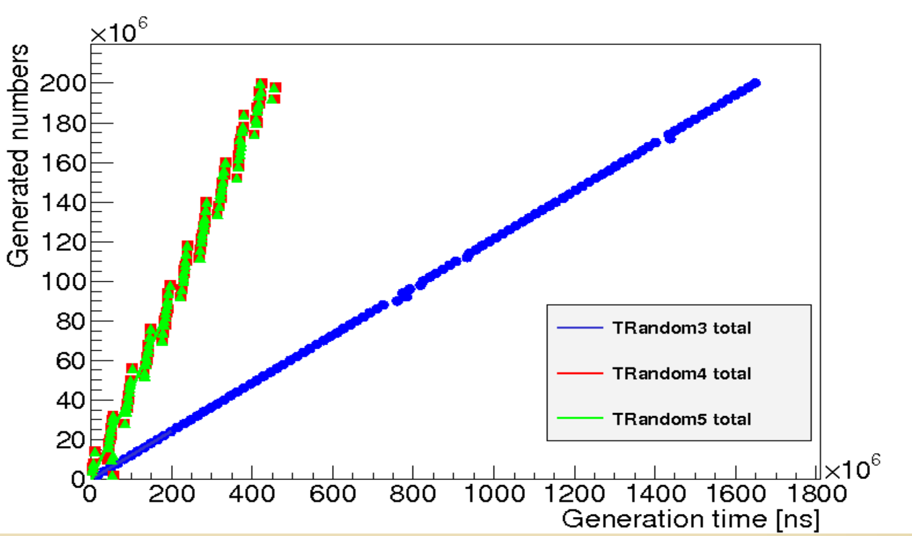
CPU





GPU



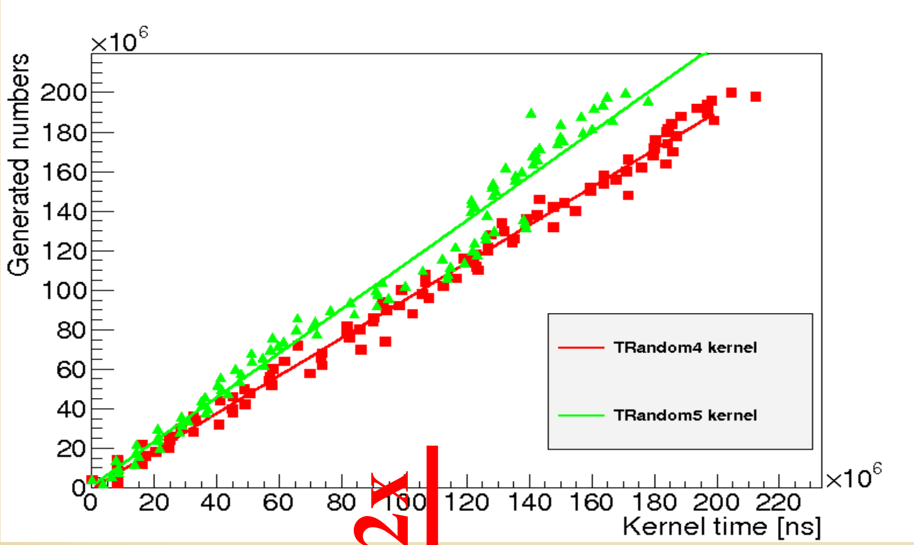


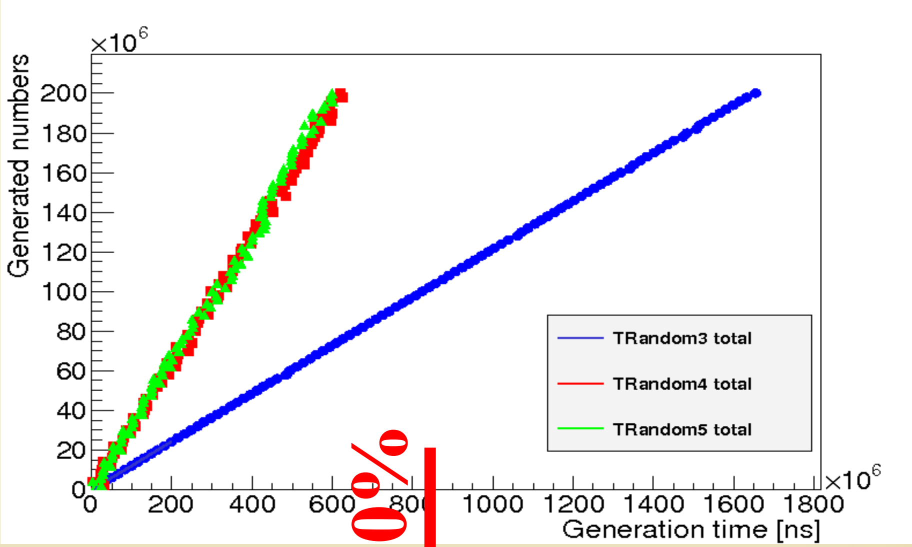
SPEED without writing (V)RAM

Kernel time

Full calculation

CPU

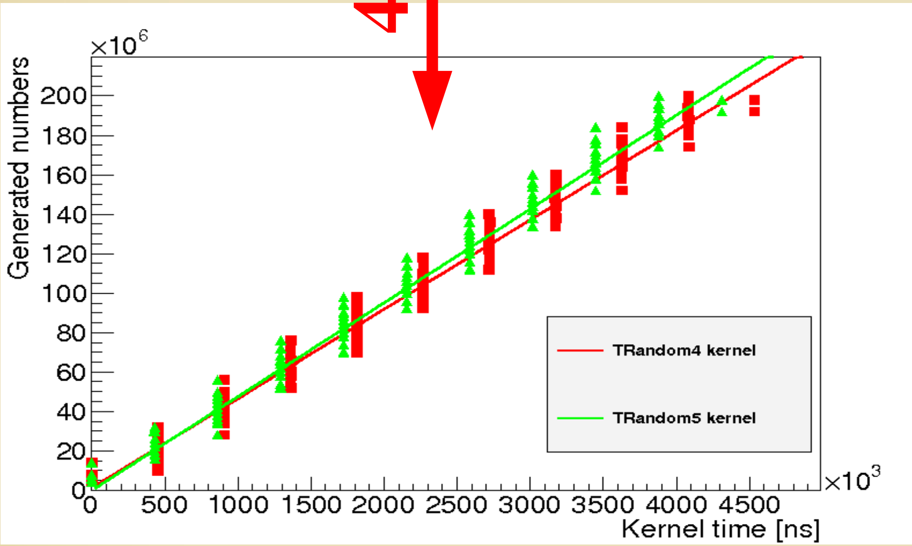


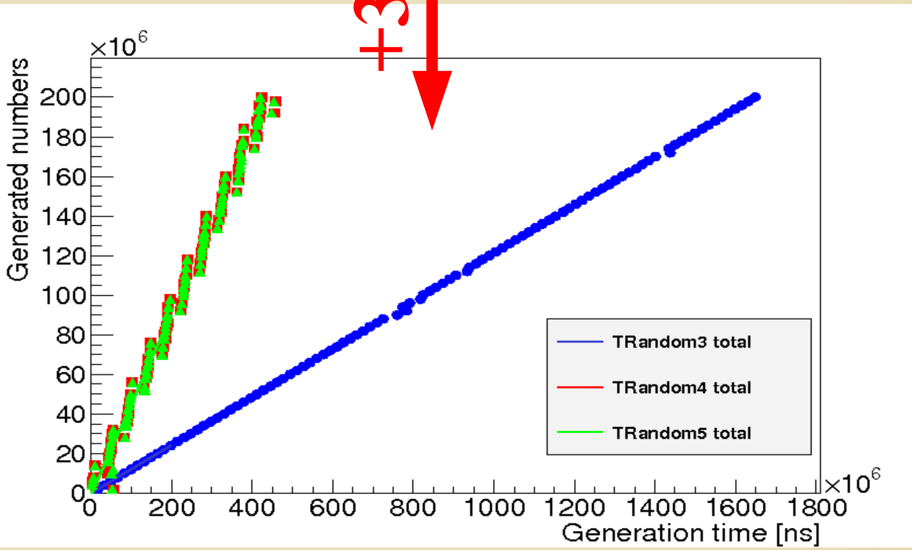


42x

+30%

GPU

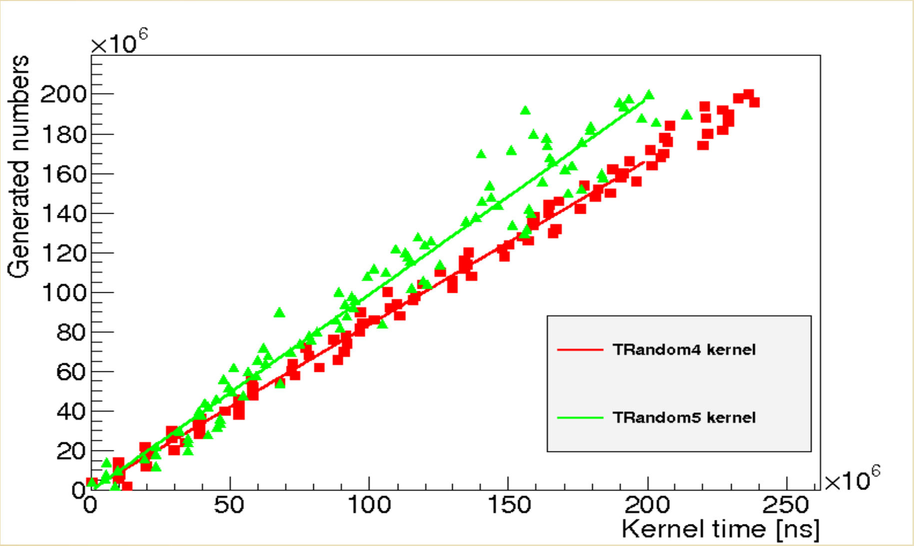


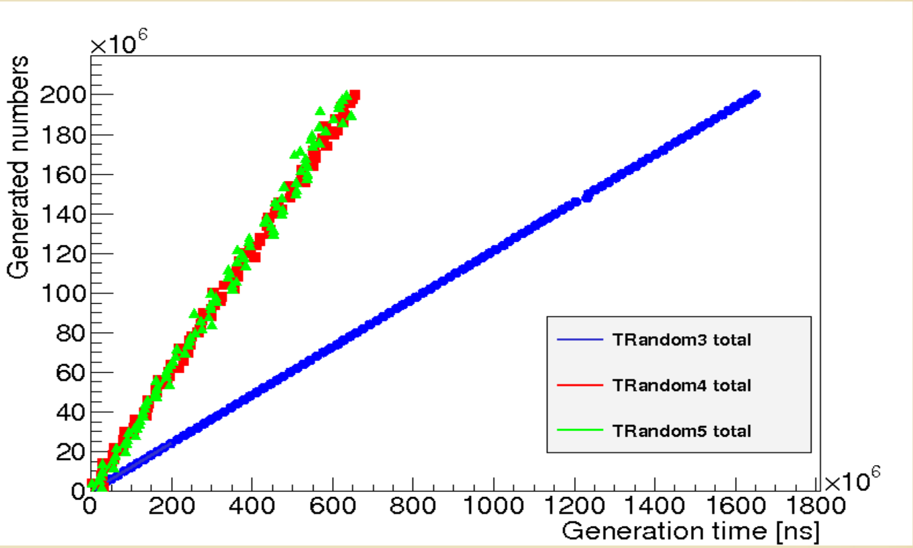


SPEED with writing (V)RAM

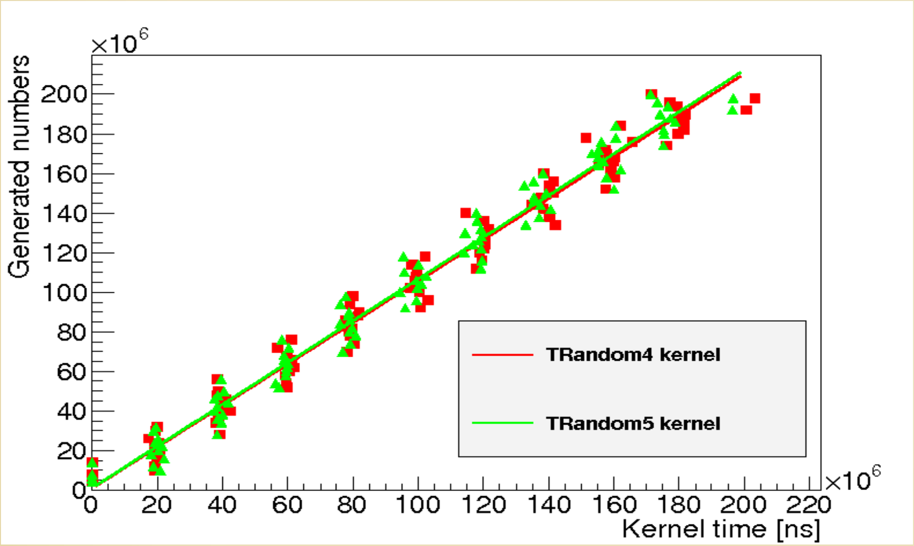
Kernel time

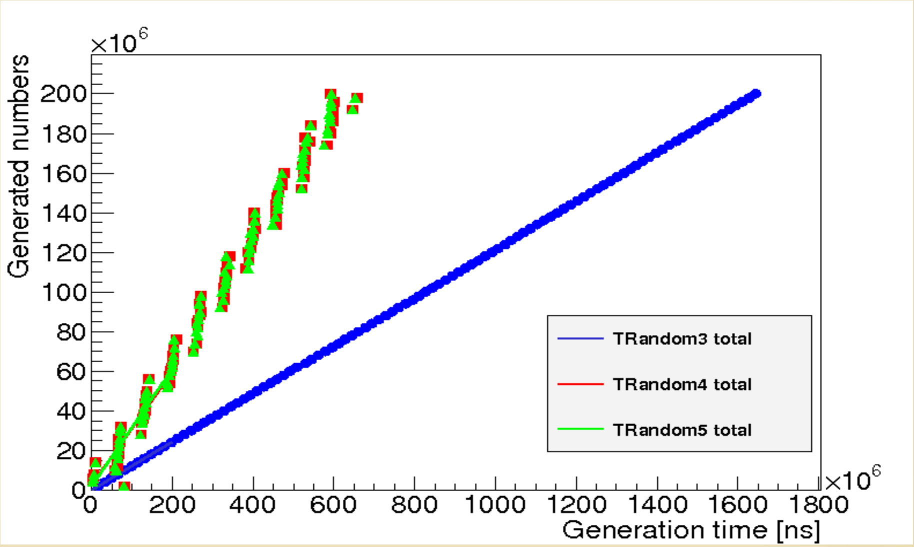
Full calculation





CPU

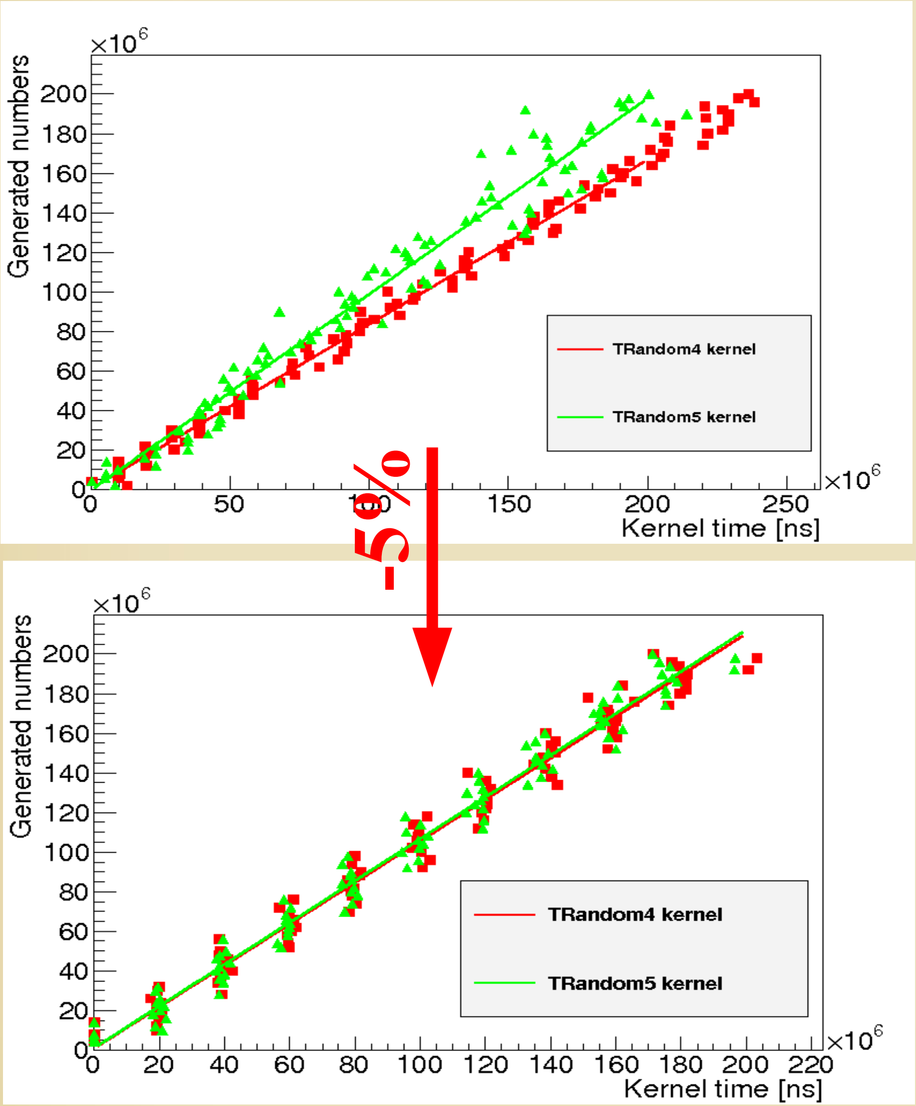




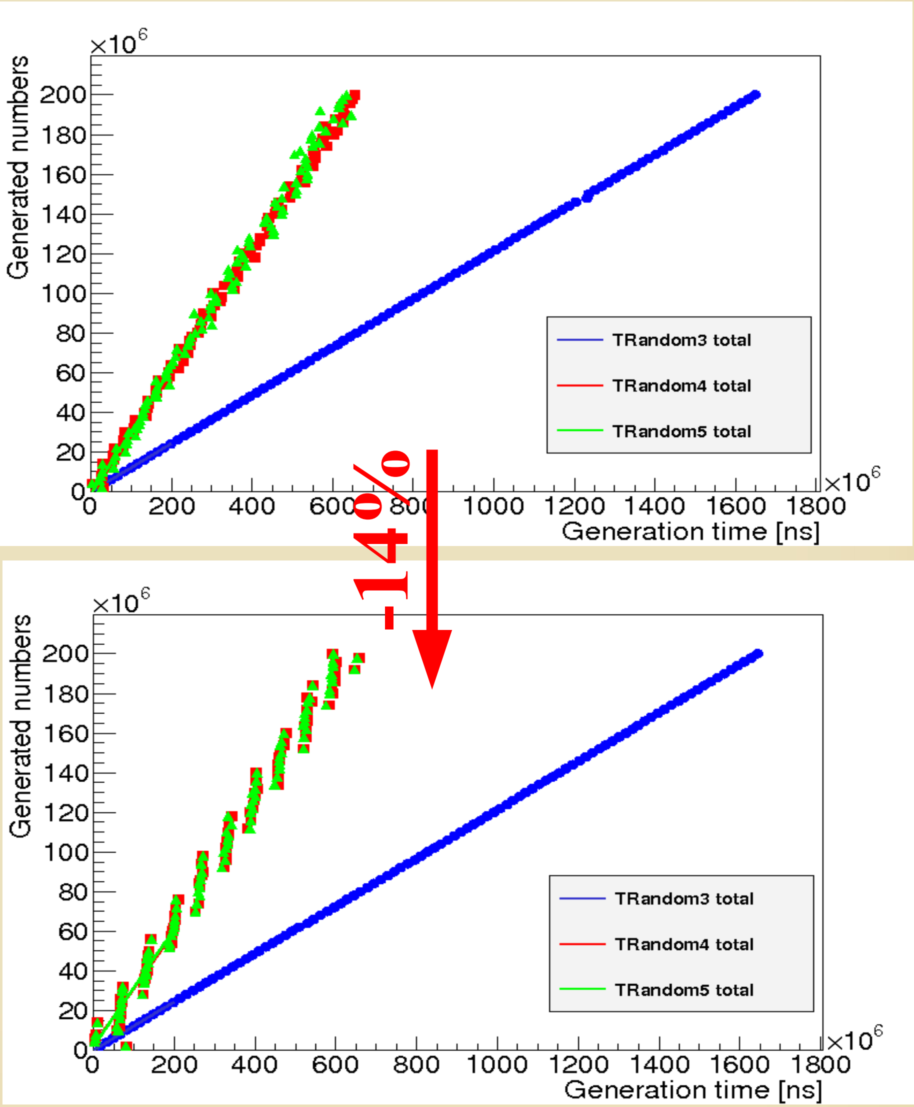
GPU

SPEED with writing (V)RAM

Kernel time



Full calculation



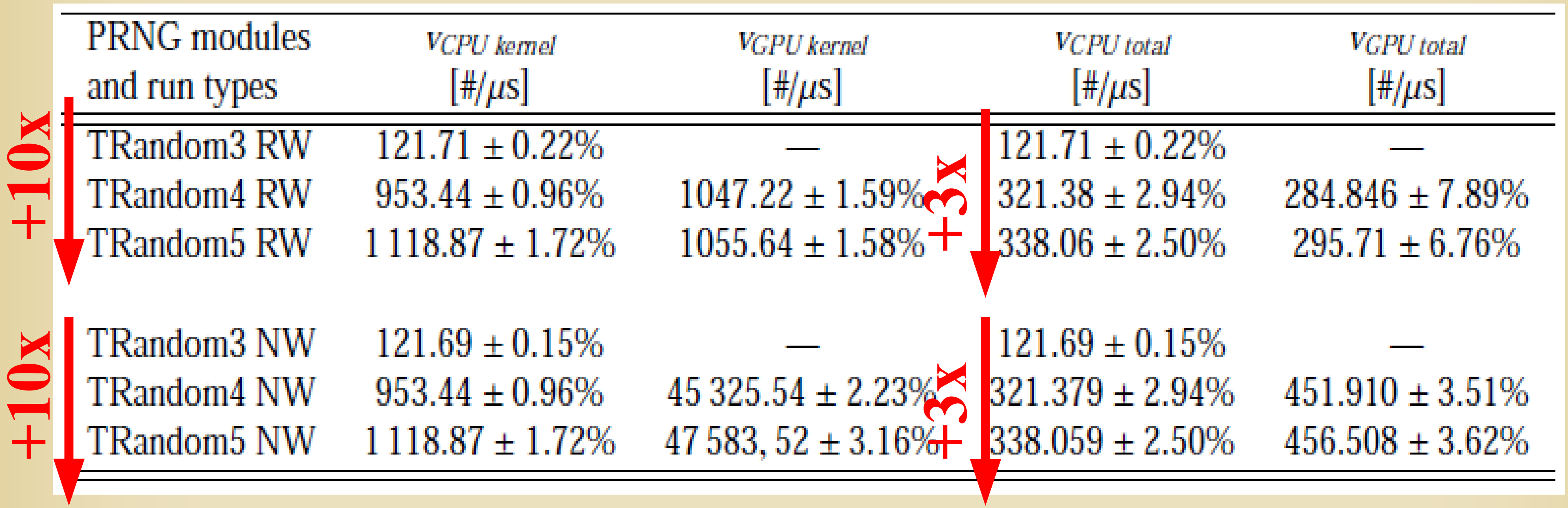
CPU

GPU

So, how about SPEED test?

- For this setup (Core i7 vs. ATi Radeon HD5970)

TRandom3 < TRandom4 < Trandom5



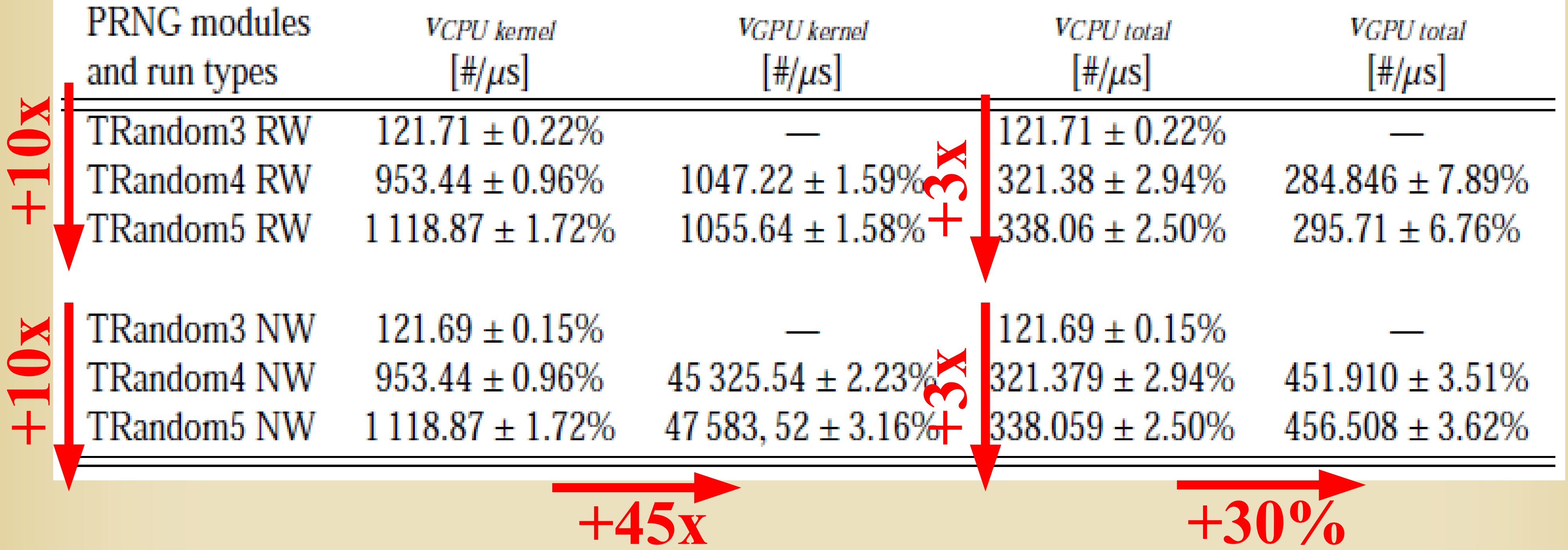
PRNG modules and run types	$v_{CPU\ kernel}$ [#/μs]	$v_{GPU\ kernel}$ [#/μs]	$v_{CPU\ total}$ [#/μs]	$v_{GPU\ total}$ [#/μs]
TRandom3 RW	121.71 ± 0.22%	—	121.71 ± 0.22%	—
TRandom4 RW	953.44 ± 0.96%	1047.22 ± 1.59%	321.38 ± 2.94%	284.846 ± 7.89%
TRandom5 RW	1 118.87 ± 1.72%	1055.64 ± 1.58%	338.06 ± 2.50%	295.71 ± 6.76%
TRandom3 NW	121.69 ± 0.15%	—	121.69 ± 0.15%	—
TRandom4 NW	953.44 ± 0.96%	45 325.54 ± 2.23%	321.379 ± 2.94%	451.910 ± 3.51%
TRandom5 NW	1 118.87 ± 1.72%	47 583, 52 ± 3.16%	338.059 ± 2.50%	456.508 ± 3.62%

So, how about SPEED test?

- For this setup (Core i7 vs. ATi Radeon HD5970)

TRandom3 < TRandom4 < Trandom5

Kernel calculation is faster (NW)



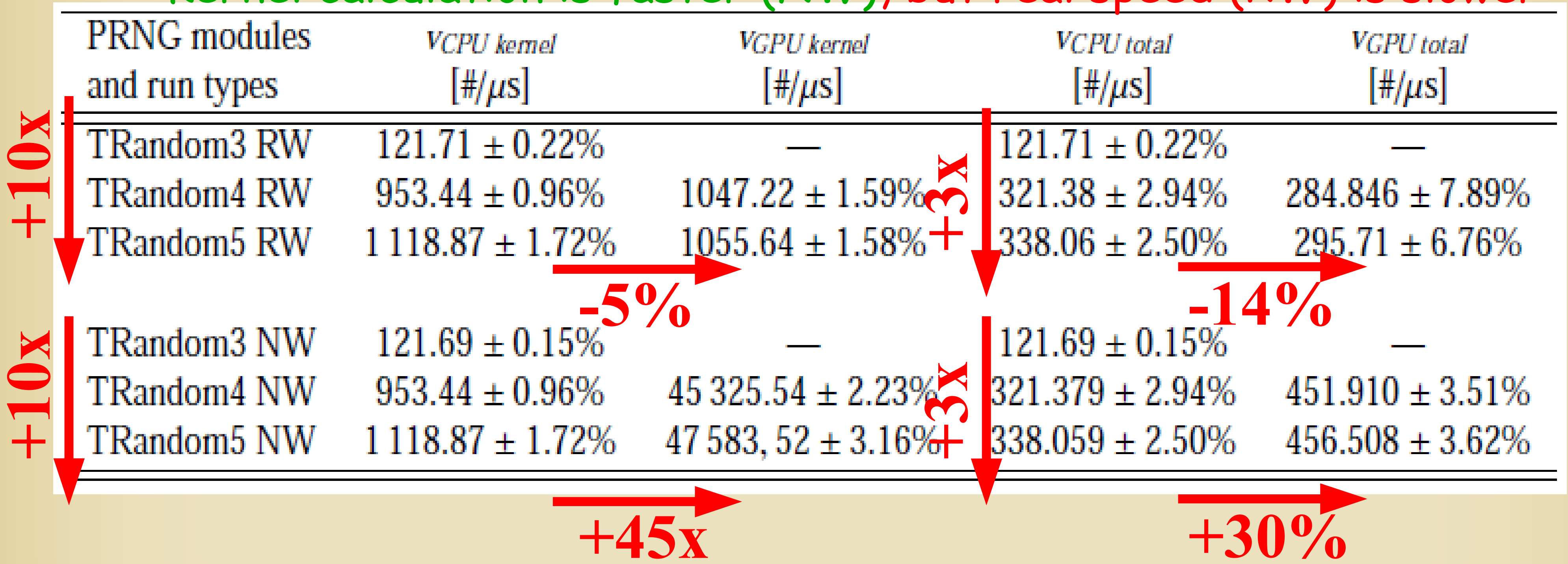
PRNG modules and run types	$V_{CPU\ kernel}$ [#/μs]	$V_{GPU\ kernel}$ [#/μs]	$V_{CPU\ total}$ [#/μs]	$V_{GPU\ total}$ [#/μs]
TRandom3 RW	121.71 ± 0.22%	—	121.71 ± 0.22%	—
TRandom4 RW	953.44 ± 0.96%	1047.22 ± 1.59%	321.38 ± 2.94%	284.846 ± 7.89%
TRandom5 RW	1 118.87 ± 1.72%	1055.64 ± 1.58%	338.06 ± 2.50%	295.71 ± 6.76%
TRandom3 NW	121.69 ± 0.15%	—	121.69 ± 0.15%	—
TRandom4 NW	953.44 ± 0.96%	45 325.54 ± 2.23%	321.379 ± 2.94%	451.910 ± 3.51%
TRandom5 NW	1 118.87 ± 1.72%	47 583, 52 ± 3.16%	338.059 ± 2.50%	456.508 ± 3.62%

So, how about SPEED test?

- For this setup (Core i7 vs. ATi Radeon HD5970)

TRandom3 < TRandom4 < Trandom5

Kernel calculation is faster (NW), but real speed (RW) is slower



PRNG modules and run types	$V_{CPU\ kernel}$ [#/μs]	$V_{GPU\ kernel}$ [#/μs]	$V_{CPU\ total}$ [#/μs]	$V_{GPU\ total}$ [#/μs]
TRandom3 RW	121.71 ± 0.22%	—	121.71 ± 0.22%	—
TRandom4 RW	953.44 ± 0.96%	1047.22 ± 1.59%	321.38 ± 2.94%	284.846 ± 7.89%
TRandom5 RW	1 118.87 ± 1.72%	1055.64 ± 1.58%	338.06 ± 2.50%	295.71 ± 6.76%
TRandom3 NW	121.69 ± 0.15%	—	121.69 ± 0.15%	—
TRandom4 NW	953.44 ± 0.96%	45 325.54 ± 2.23%	321.379 ± 2.94%	451.910 ± 3.51%
TRandom5 NW	1 118.87 ± 1.72%	47 583, 52 ± 3.16%	338.059 ± 2.50%	456.508 ± 3.62%

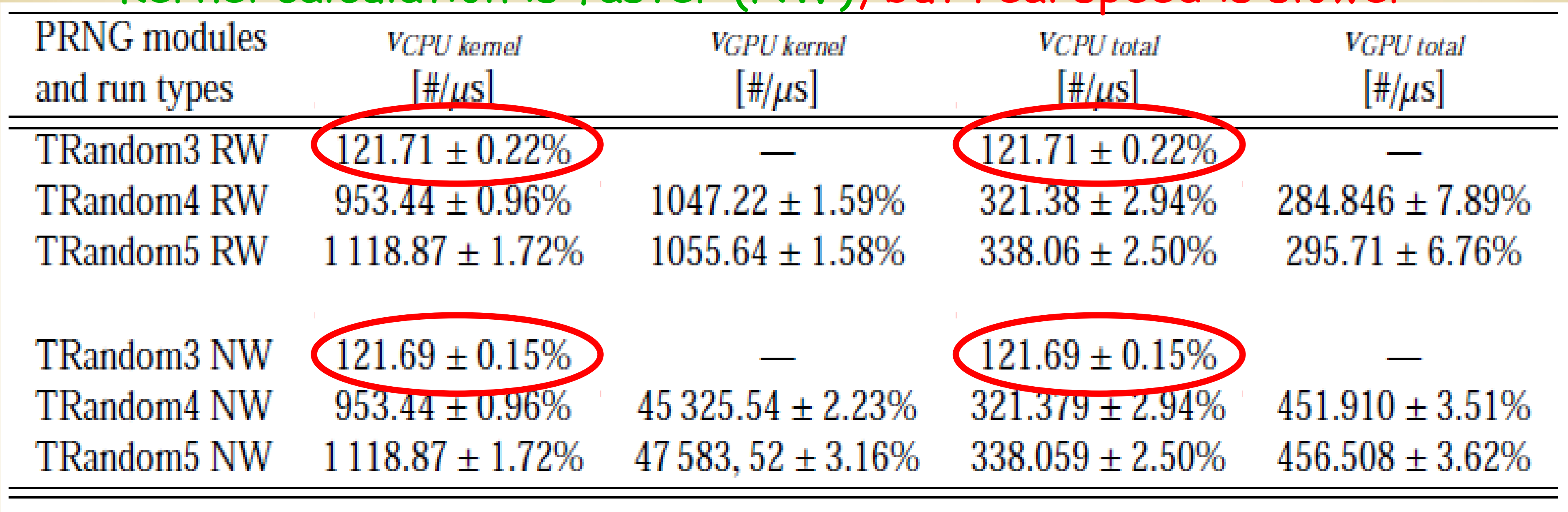
Note$_1$: New GPU cards are 2-5 times faster

So, how about SPEED test?

- For this setup (Core i7 vs. ATi Radeon HD5970)

TRandom3 < TRandom4 < Trandom5

Kernel calculation is faster (NW), but real speed is slower



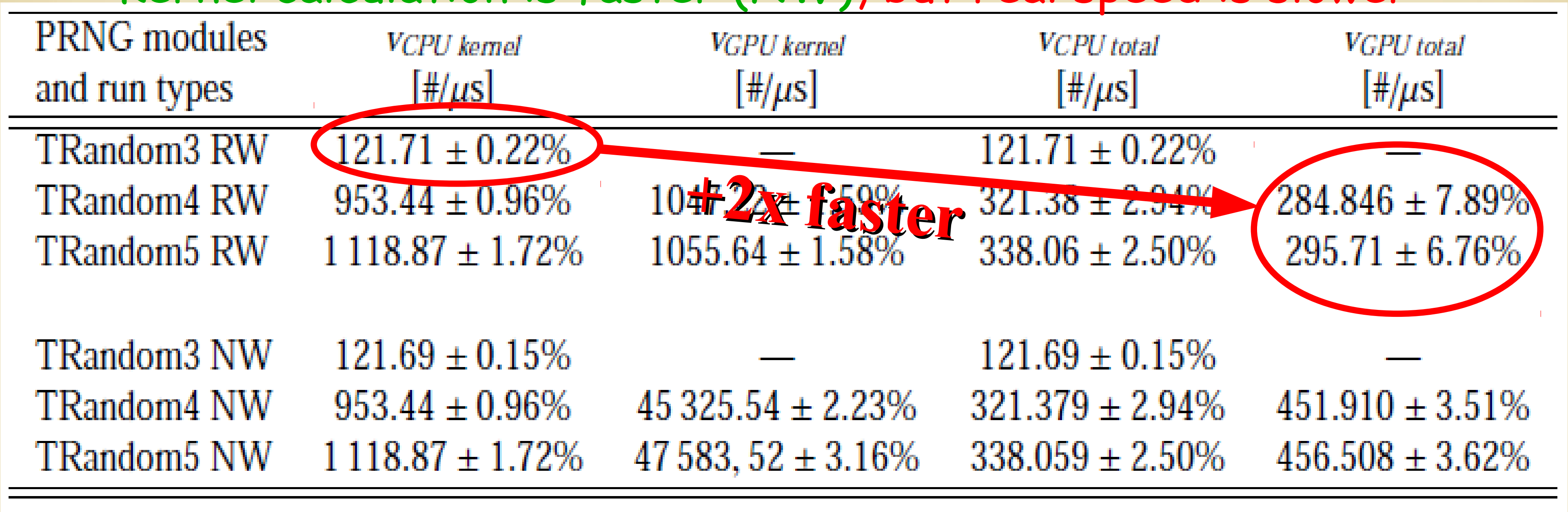
PRNG modules and run types	$V_{CPU\ kernel}$ [#/μs]	$V_{GPU\ kernel}$ [#/μs]	$V_{CPU\ total}$ [#/μs]	$V_{GPU\ total}$ [#/μs]
TRandom3 RW	121.71 ± 0.22%	—	121.71 ± 0.22%	—
TRandom4 RW	953.44 ± 0.96%	1047.22 ± 1.59%	321.38 ± 2.94%	284.846 ± 7.89%
TRandom5 RW	1 118.87 ± 1.72%	1055.64 ± 1.58%	338.06 ± 2.50%	295.71 ± 6.76%
TRandom3 NW	121.69 ± 0.15%	—	121.69 ± 0.15%	—
TRandom4 NW	953.44 ± 0.96%	45 325.54 ± 2.23%	321.379 ± 2.94%	451.910 ± 3.51%
TRandom5 NW	1 118.87 ± 1.72%	47 583, 52 ± 3.16%	338.059 ± 2.50%	456.508 ± 3.62%

So, how about SPEED test?

- For this setup (Core i7 vs. ATi Radeon HD5970)

TRandom3 < TRandom4 < Trandom5

Kernel calculation is faster (NW), but real speed is slower



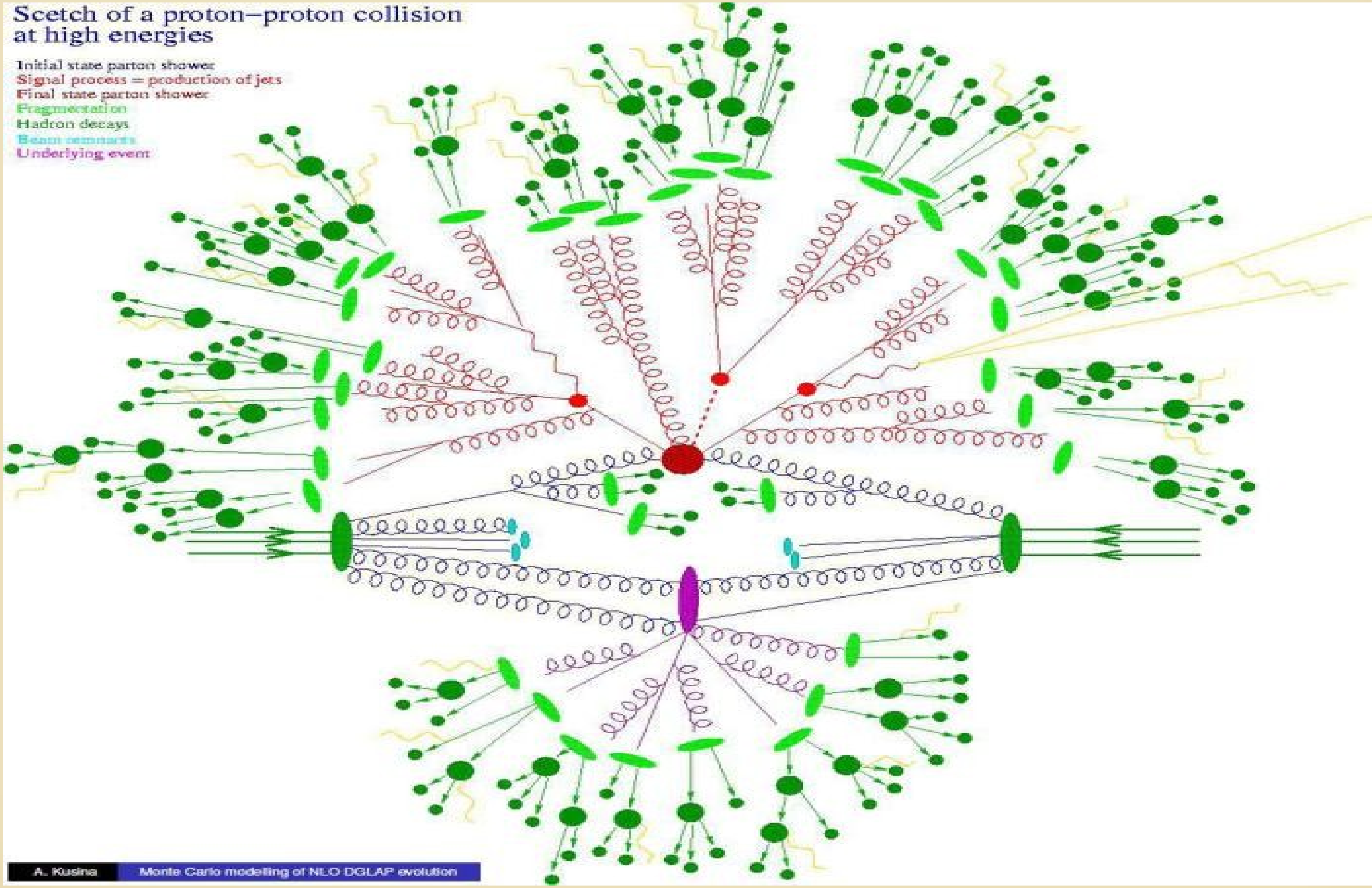
PRNG modules and run types	$v_{CPU\ kernel}$ [#/μs]	$v_{GPU\ kernel}$ [#/μs]	$v_{CPU\ total}$ [#/μs]	$v_{GPU\ total}$ [#/μs]
TRandom3 RW	121.71 ± 0.22%	—	121.71 ± 0.22%	—
TRandom4 RW	953.44 ± 0.96%	1047.22 ± 1.59%	321.38 ± 2.94%	284.846 ± 7.89%
TRandom5 RW	1 118.87 ± 1.72%	1055.64 ± 1.58%	338.06 ± 2.50%	295.71 ± 6.76%
TRandom3 NW	121.69 ± 0.15%	—	121.69 ± 0.15%	—
TRandom4 NW	953.44 ± 0.96%	45 325.54 ± 2.23%	321.379 ± 2.94%	451.910 ± 3.51%
TRandom5 NW	1 118.87 ± 1.72%	47 583, 52 ± 3.16%	338.059 ± 2.50%	456.508 ± 3.62%

+2x faster

Note$_2$: Parallel computing (OpenCL) improves speed!

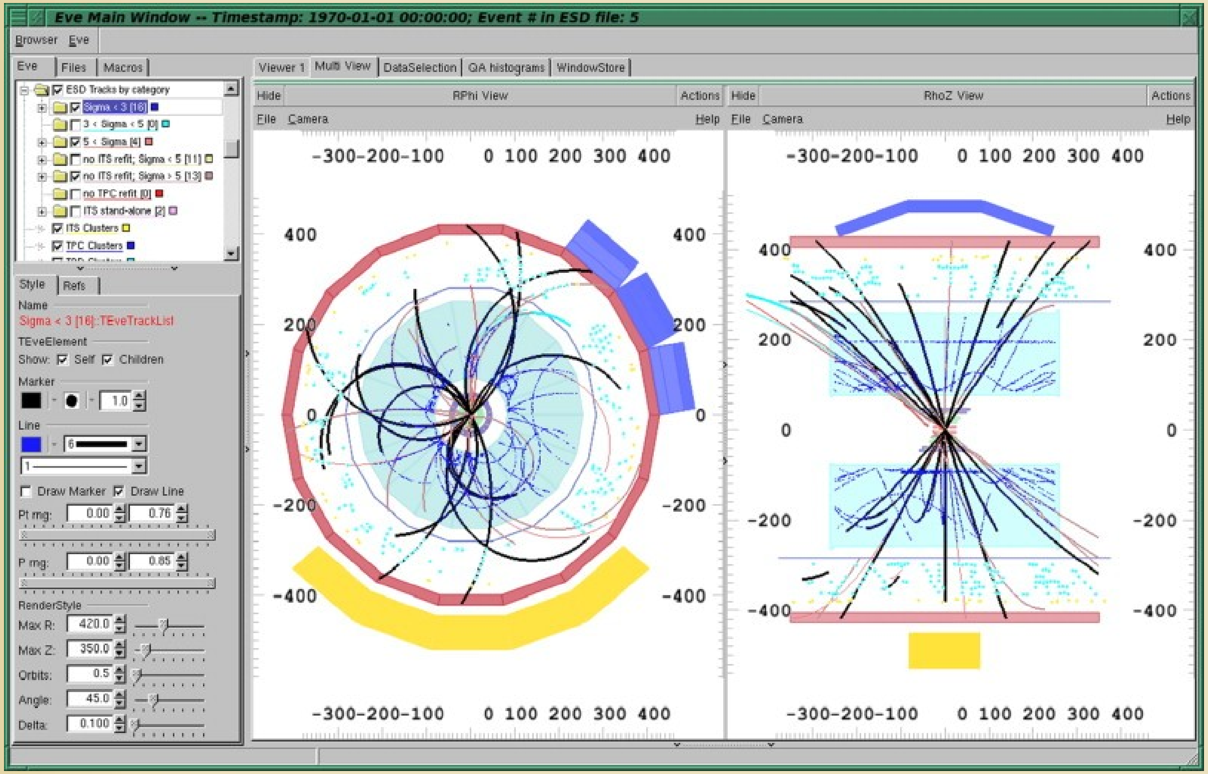
Some Physics: proton-proton collisions

- Theoretical model of a pp collisions



Some Physics: proton-proton collisions

- A reconstructed pp event in the ALICE experiment

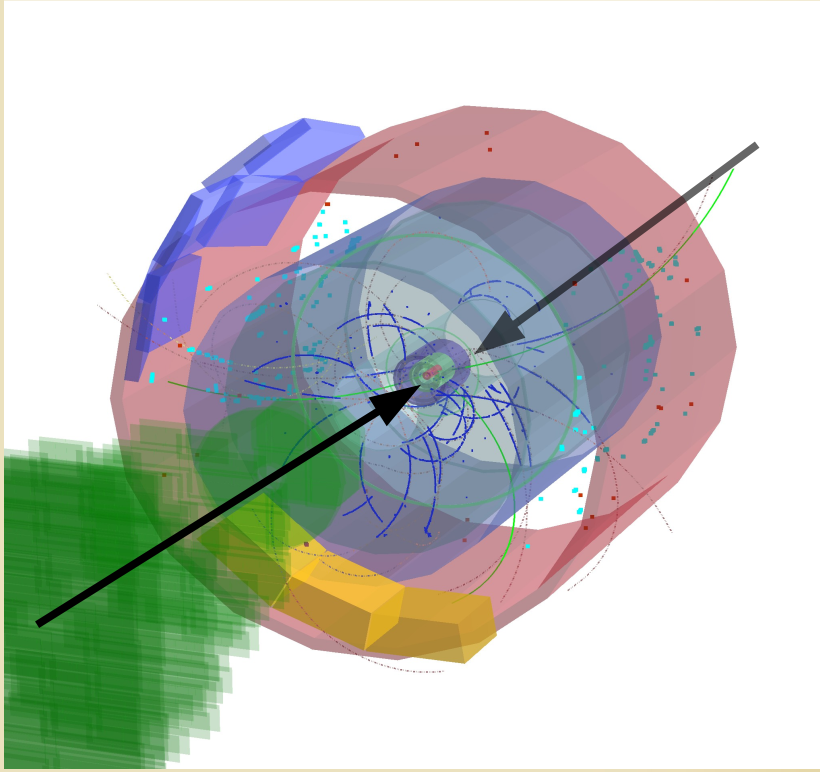


Eve Main Window -- Timestamp: 1970-01-01 00:00:00; Event # in ESD file: 5

☑ ESD Tracks by category
- ☑ Sigma < 3 [16]
- ☐ 3 < Sigma < 5 [0]
- ☑ 5 < Sigma [4]
- ☐ no ITS refit; Sigma < 5 [11]
- ☑ no ITS refit; Sigma > 5 [13]
- ☐ no TPC refit [0]
- ☐ ITS stand-alone [2]

☑ ITS Clusters
☑ TPC Clusters

Name: Sigma < 3 [16]:TEveTrackList
TEveElement
Show: ☑ Self ☑ Children
Marker: 1.0
Line: 6, 1
☐ Draw Marker ☑ Draw Line
Pt rng: 0.00 0.76
P rng: 0.00 0.85
RenderStyle
Max R: 420.0
Max Z: 350.0
Orbits: 0.5
Angle: 45.0
Delta: 0.100

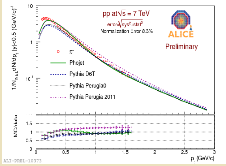


Some Physics: pp collisions at GPU

- 400k TRandom5 PRNG

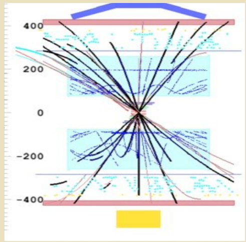
Transverse momentum spectrum

dN/dp_T (Tsallis distr.)



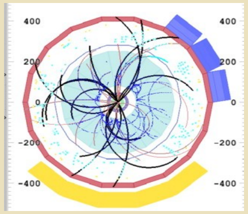
Rapidity distribution

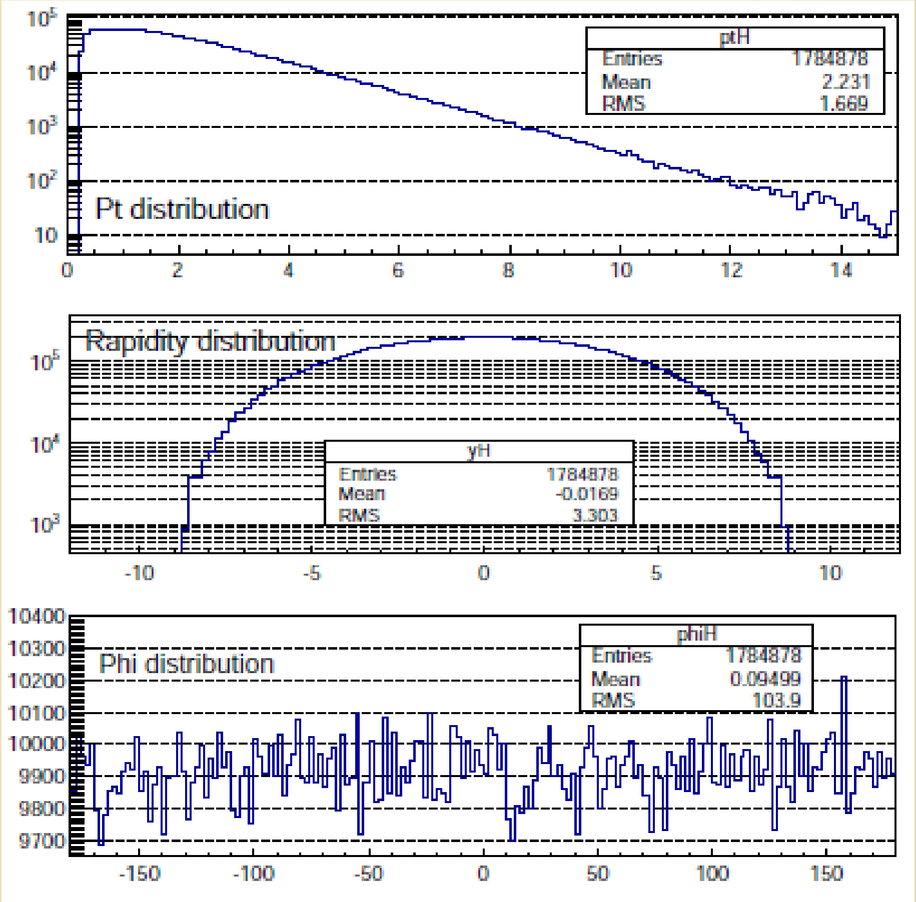
dN/dy (Gaussian distr.)



Angular distribution

$dN/d\varphi$ (Isotropy)



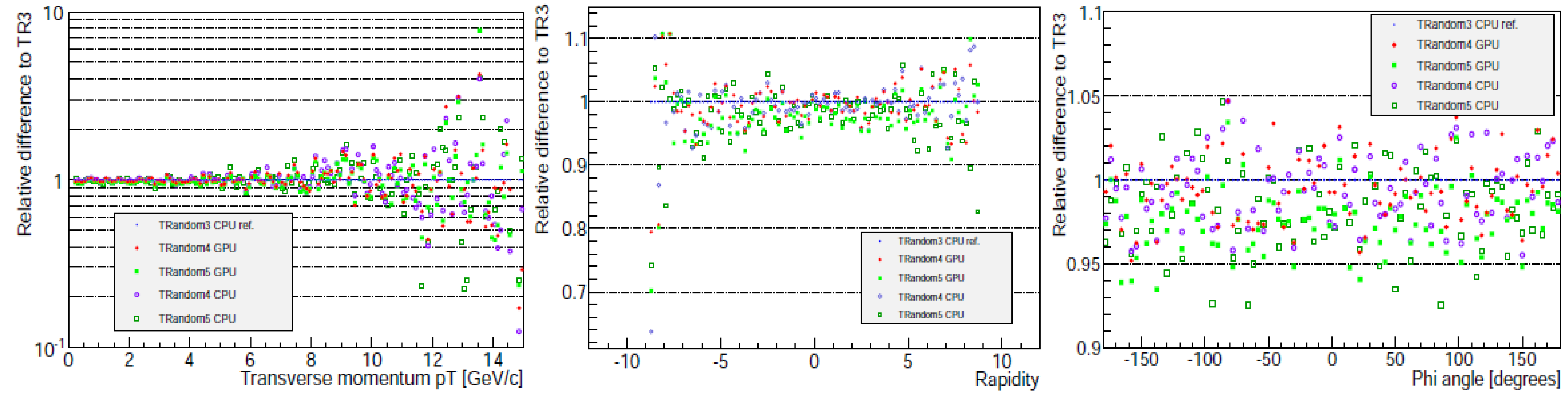


Some Physics: pp collisions at GPU

- To check the validity of the 'physics':

  Compare calulated distributions to the original Trandom3 CPU

  TRandomX/TRandom3 must be ~1 depending on statistics

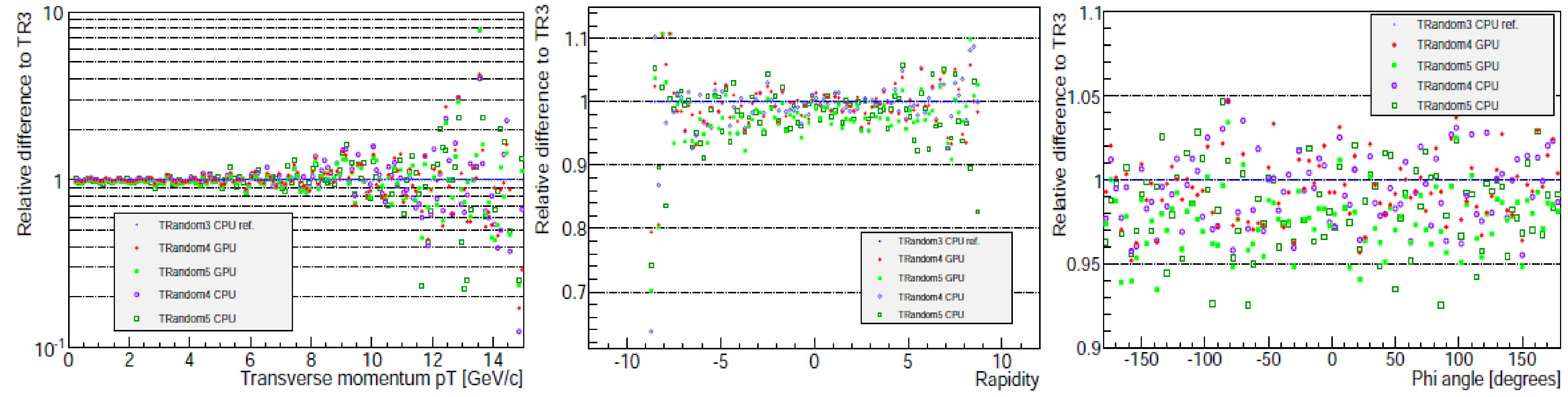


Some Physics: pp collisions at GPU

- To check the validity of the 'physics':

Compare calulated distributions to the original Trandom3 CPU

TRandomX/TRandom3 must be ~1 depending on statistics



10% agreement

up to p_T<6 GeV/c

5% agreement

in |y|<5

5% agreement

in the whole φ

SUMMARY

- Aim
  - Faster MC event generation for HIC
- Resuts for pp MC @ GPUs
  - Diehard test of open source PRNGs: (SFMT, MWC64X) on GPUs
  - Implementation of new GPU based modules (TRandom4, TRandom5) to Root/AliRoot framework
  - Tests: simulation of high-energy pp collisions
- Take away message
  - GPUs can be used for Monte Carlo generators in HIC
  - One needs more programming (CUDA/OpenCL/...)
  - Need to optimize (price/speed) since other technologies available (e.g. Intel Xeon Phi)

OUTLOOK

- The presented results are on
  - AliRoot, especially AliPYTHIA for proton-proton
  - CPU/GPU SIMD-oriented Fast MT & MWC64X
  - Standalone machine (with ATi Radeon HD5970)
- How to improve?
  - Ongoing: HIJING calculations (need for more PRNGs), so might be more efficient, faster
  - Trivial: Buy new fast cards and re-test – we are on it and we hope the funging agency on it as well.
  - The framework is almost ready to test in the GRID using JDL (required HW: GPUs, SW: OpenCL/CUDA/...)
  - More faster PRNGs on CPUs/GPUs (Tiny MT, MTGP), but note, faster PRNG less randomness quality.
  - Further modules can be moved to GPU

BACKUP

The PRNG quality test

Some DieHard tests by George Marsaglia

Birthday spacings: Choose random points on a large interval. The spacings between the points should be asymptotically exponentially distributed. The name is based on the birthday paradox.

Overlapping permutations: Analyze sequences of five consecutive random numbers. The 120 possible orderings should occur with statistically equal probability.

Ranks of matrices: Select some number of bits from some number of random numbers to form a matrix over {0,1}, then determine the rank of the matrix. Count the ranks.

Monkey tests: Treat sequences of some number of bits as "words". Count the overlapping words in a stream. The number of "words" that don't appear should follow a known distribution. The name is based on the infinite monkey theorem.

Count the 1s: Count the 1 bits in each of either successive or chosen bytes. Convert the counts to "letters", and count the occurrences of five-letter "words".

Parking lot test: Randomly place unit circles in a 100 x 100 square. If the circle overlaps an existing one, try again. After 12,000 tries, the number of successfully "parked" circles should follow a certain normal distribution.

Minimum distance test: Randomly place 8,000 points in a 10,000 x 10,000 square, then find the minimum distance between the pairs. The square of this distance should be exponentially distributed with a certain mean.

Random spheres test: Randomly choose 4,000 points in a cube of edge 1,000. Center a sphere on each point, whose radius is the minimum distance to another point. The smallest sphere's volume should be exponentially distributed with a certain mean.

The squeeze test: Multiply 231 by random floats on [0,1) until you reach 1. Repeat this 100,000 times. The number of floats needed to reach 1 should follow a certain distribution.

Overlapping sums test: Generate a long sequence of random floats on [0,1). Add sequences of 100 consecutive floats. The sums should be normally distributed with characteristic mean and sigma.

Runs test: Generate a long sequence of random floats on [0,1). Count ascending and descending runs. The counts should follow a certain distribution.

The craps test: Play 200,000 games of craps, counting the wins and the number of throws per game. Each count should follow a certain