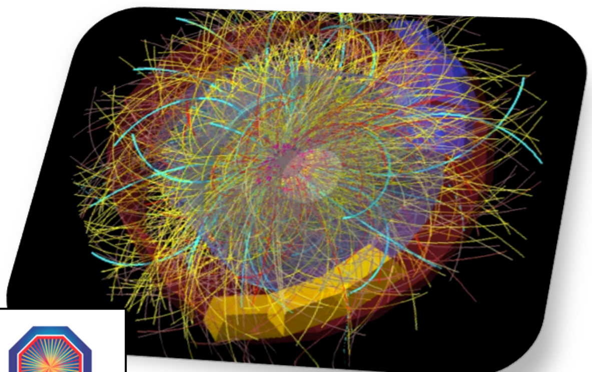
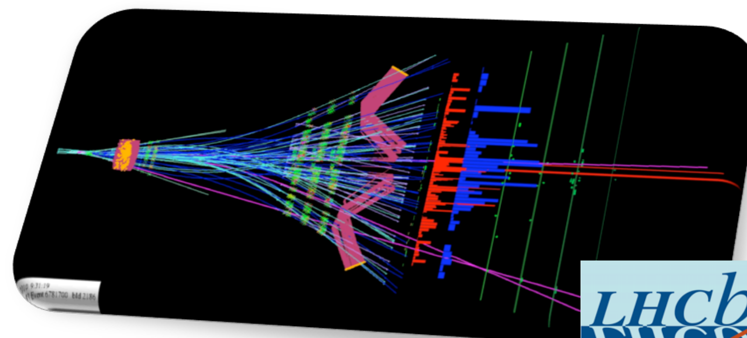
Opportunities of Target Accuracy in HEP Software: Focus on Mathematical Libraries





D. Piparo – PH-SFT

3rd CERN Openlab – Intel
Workshop on Numerical Computing

Introduction

- Floating point calculations represent a significant portion of the runtime budget of HEP applications
- Sofisticated FORmulas TRANSlated from literature into code (see Vincenzo's talk)
  - Mathematical functions appear often
  - Their execution is expensive!

Can the concept of target accuracy help us to improve the impact of mathematical functions' evaluation?

Pricetags of Mathematical Functions

Prices reported in percentage of the runtime of a full job

- LHCb:
  - Reconstruction: ~8.7%
- CMS:
  - Reconstruction: ~19.9%
  - Simulation: ~13.5%
  - Simulation Initialisation: ~30%
- Alice (Pb-Pb, Geant4):
  - Simulation: >25% (in event loop only)

Pricetags assume the Libm implementation of these functions (the one used in production since years)

Software stacks versions: beginning 2013

Absolute Costs: Sim @ LHC in 2012

An example of typical HEP workflow

Simulated Monte Carlo events are as important as "real" collision events

- Large statistics necessary to understand measured data
  - Data analyses aim to isolate small corners of the overall phase-space
  - Ultimately necessary for discoveries

Amount of events simulated in 2012 (in Billions):

- ATLAS: 2.1*
- CMS: 4
- LHCb: 1.2
- ALICE: 1

One event: ~seconds up to more than a minute CPU time
(For Alice, heavy ions collisions' simulation, ~30 minutes)

Ages in terms of total CPU time!

Simulation alone every year accounts for **lots** of computing resources

** +1.8 Billion Fast simulated*

Libm: the default library

With some exceptions, the default mathematical library used for HEP calculations is Libm:

- A rock-solid reference!
- Always focussed on accuracy rather than performance

Are these two features going to change in the future?

The Last Episode of the Saga

27-8-2012: http://rhn.redhat.com/errata/RHSA-2012-1207.html

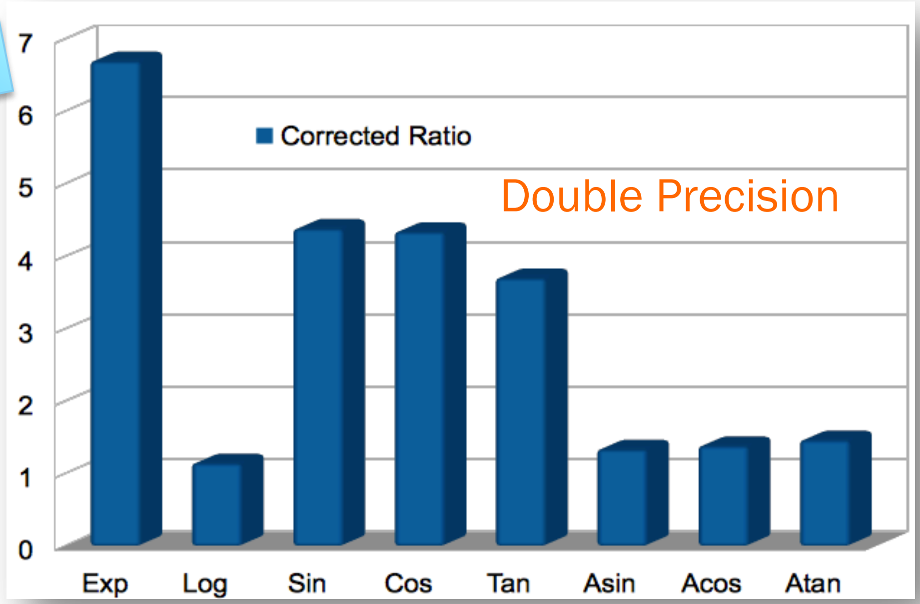
* Previously, logic errors in various mathematical functions, including exp, exp2, expf, exp2f, pow, sin, tan, and rint, caused inconsistent results when the functions were used with the non-default rounding mode. This could also cause applications to crash in some cases. With this update, the functions now give correct results across the four different rounding modes. (BZ#839411)

Excellent:

Now results are consistent!

(***feraiseexcept*** function used to raise fp exception when result "unprecise")

Is Libm Going to be Faster?



Wait: how much tax payers' money does this cost?

The modified routines cause a slow-down of a factor >6 for Exp and important ones for Sin, Cos and Tan.

Nice to have such a solid reference, but can we afford that in our production software?

Probably not…

What are the alternatives?

A Selection of Alternatives

A plethora of different products are available, for example:

- Intel's SVML, IMF, MKL (commercial)
- AMD Libm (free)
- VDT (VectoriseD maTh: free and open source)

Differences in the implementations but common underlying principle:

Trade off between accuracy and speed of execution

VDT

What is VDT?

- An open source math library library, LGPL3 licence
- Single and Double precision of (a)sin, (a)cos, sincos, (a)tan, atan(2), log, exp and 1/sqrt
- Fast, approximate, inline (see following slide for the details)
- Symbols names are different from traditional ones: vdt::fast_<name>
  - Do not force drop-in replacement!
- Autovectorisable since gcc 4.7
  - Array signatures available: calculate on multiple elements conveniently
  - Can be inserted in autovectorised loops (inline!)
- Inspired by the good old Cephes (and Quake III videogame)
- Standard C code only is used (no intrinsics): portability guaranteed
  - ARM, x86, GPGPUs, Xeon Phi, <future microarchitecture>

https://svnweb.cern.ch/trac/vdt

Pade' Approximants

Underlying principle behind VDT (and Cephes): Pade' Approximants

The "best" approximation of a function by a rational function of a given order
→ Often better approximation than a truncated Taylor series

Padé approximant of *f(x)* of order [m/n] is the function

$$R(x) = \frac{\sum_{j=0}^{m} a_j x^j}{1 + \sum_{k=1}^{n} b_k x^k} = \frac{a_0 + a_1 x + a_2 x^2 + \cdots + a_m x^m}{1 + b_1 x + b_2 x^2 + \cdots + b_n x^n}$$

which agrees to the highest possible order to *f(x)*

$$\begin{aligned} f(0) &= R(0) \\ f'(0) &= R'(0) \\ f''(0) &= R''(0) \\ &\vdots \\ f^{(m+n)}(0) &= R^{(m+n)}(0) \end{aligned}$$

Quake III Fast isqrt

Light effects (e.g. reflections): needed the calculation of several normalizations.

Important piece of the implementation: "magic constant" which yields to a first rough value of the sqrt, then improved with Newton's method iterations.

```
/// Sqrt implmentation from Quake3
inline float fast_isqrtf_general(float x, const uint32_t ISQRT_ITERATIONS) {

      const float threehalfs = 1.5f;
      const float x2 = x * 0.5f;
      float y  = x;
      uint32_t i  = details::sp2uint32(y);
      i  = 0x5f3759df - ( i >> 1 );
      y  = details::uint322sp(i);
      for (uint32_t j=0;j<ISQRT_ITERATIONS;++j)
            y  *= ( threehalfs - ( x2 * y * y ) );

      return y;
}
```

!!

Speed: VDT Vs Libm

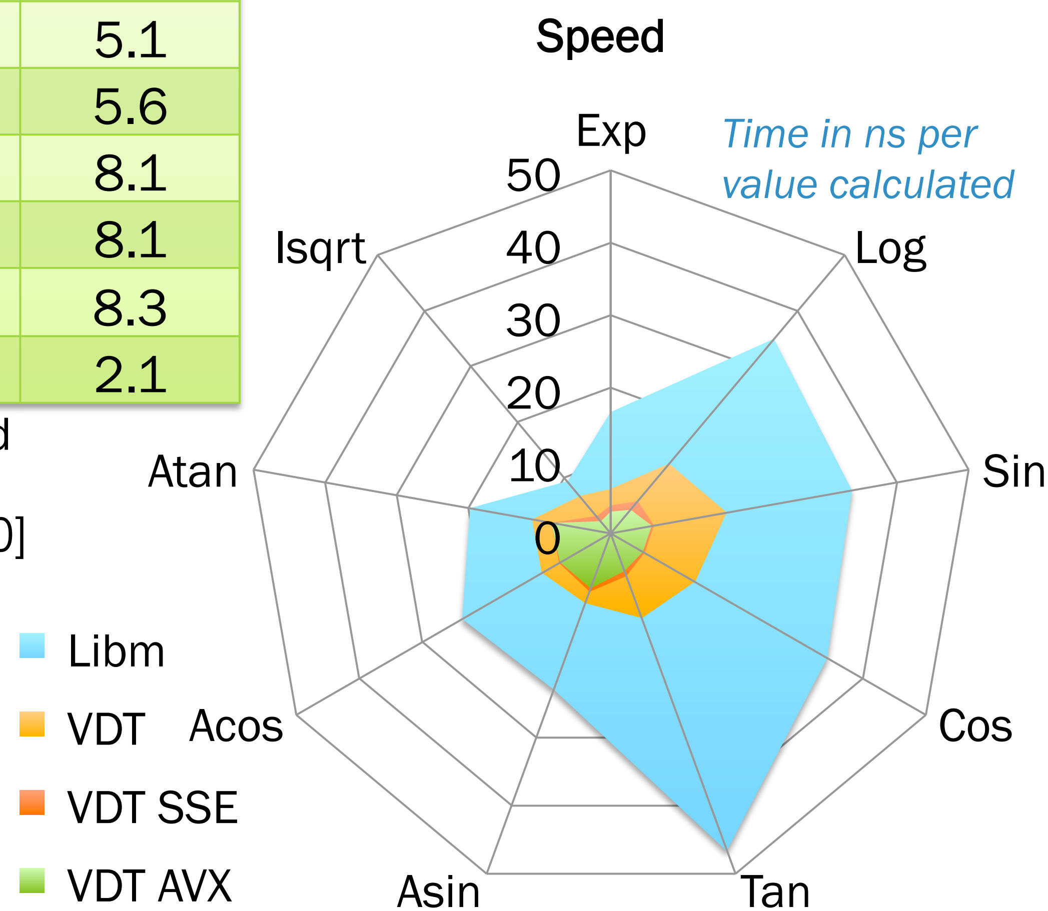
Function	Libm	VDT	VDT SSE	VDT AVX
Exp	16.7	6.1	3.8	2.9
Log	34.9	12.5	5.7	4.2
Sin	33.7	16.2	6.0	5.7
Cos	34.4	13.4	5.4	5.1
Tan	46.6	12.5	6.3	5.6
Asin	23.0	10.3	8.6	8.1
Acos	23.7	11.0	8.2	8.1
Atan	19.7	11.0	8.3	8.3
Isqrt	9.3	6.7	3.0	2.1

Time in ns per value calculated

Double Precision

Testbed:

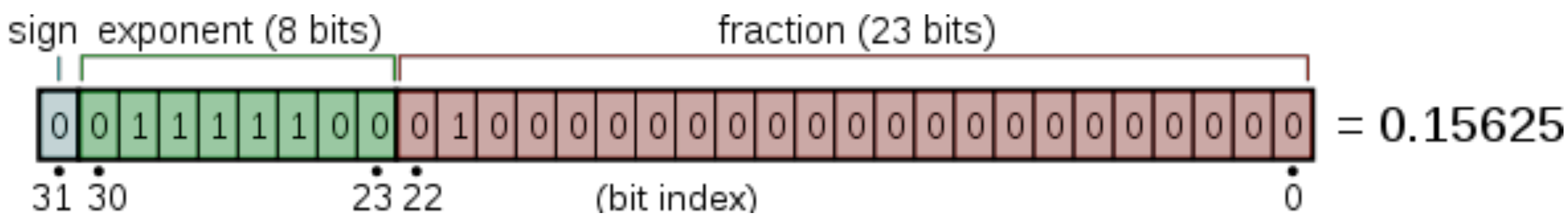
SLC6-GCC47, Core i7-3930K CPU @ 3.20GHz



- Operative input range: [-5000, 5000]
- VDT scalar functions:
  - Speedups of ~4x achievable
- Speedup scalar→SSE more significant than SSE→AVX
  - Some overhead is present

Some Words about Accuracy

- Accuracy was measured comparing the results of Libm and VDT bit by bit with the same input
- Differences quoted in terms of most significant different bit
- In the end they are just 32 (64) bits which are properly interpreted!



A single precision floating point number

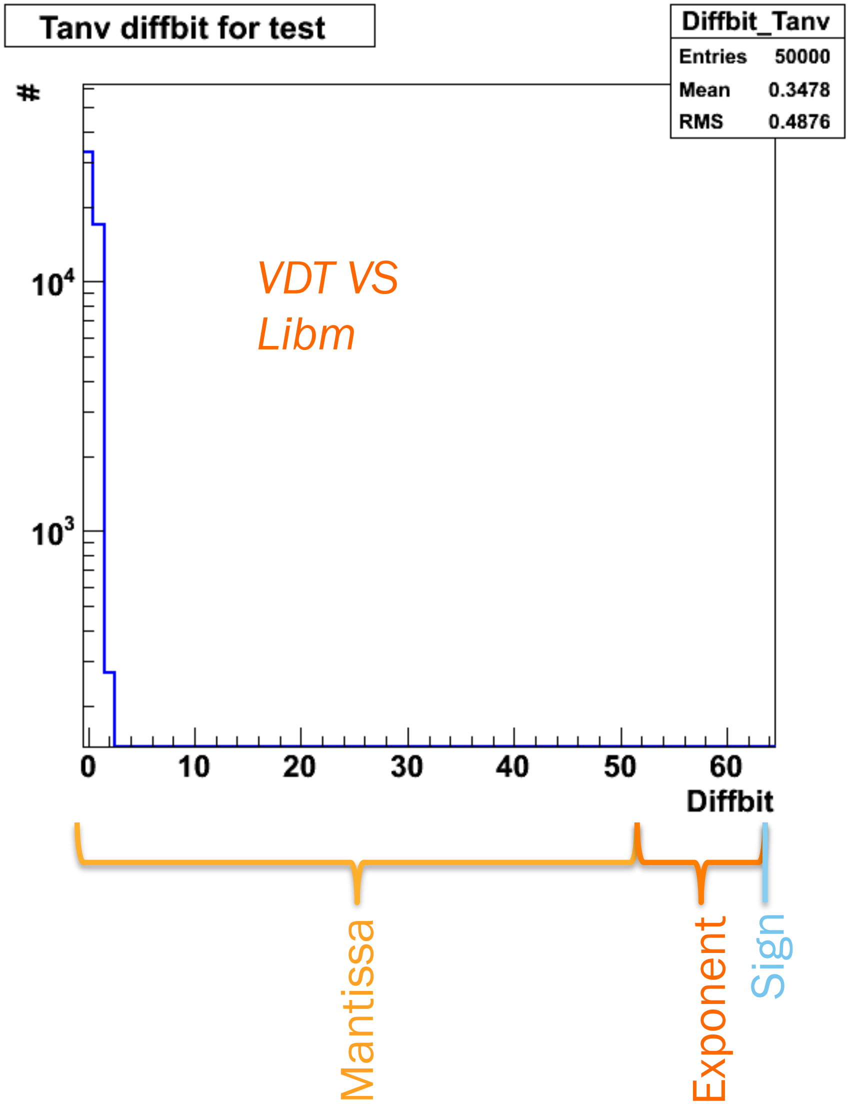
Accuracy: VDT Vs Libm

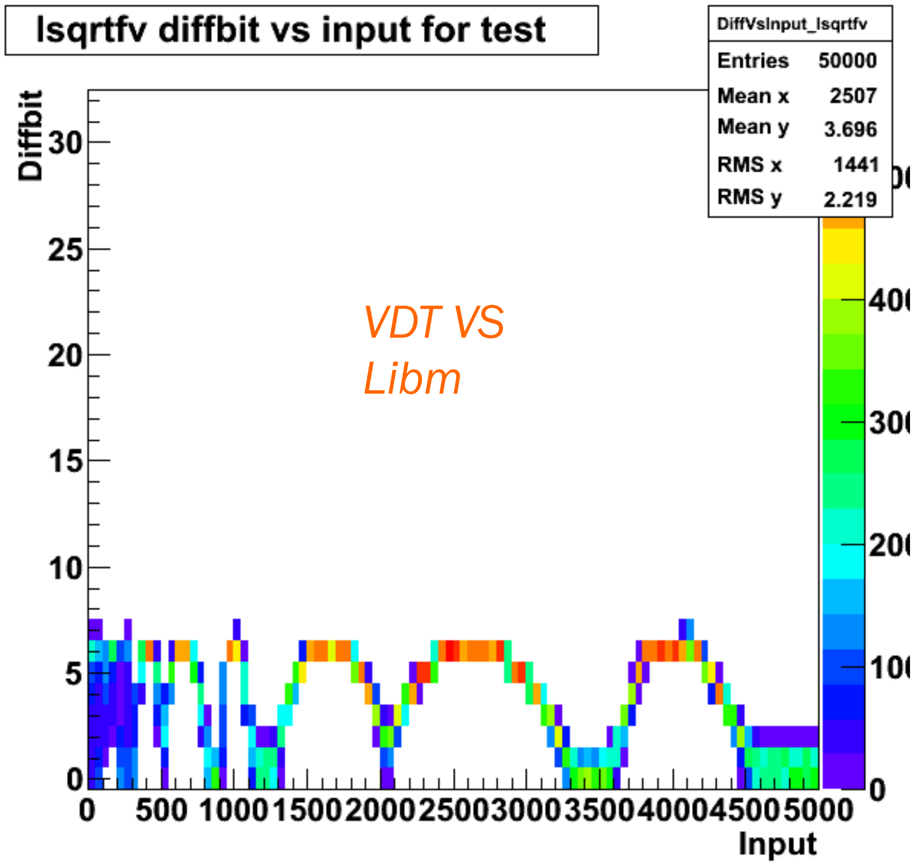
Double Precision

	MAX VDT	AVG VDT
Acos	8	0.39
Asin	2	0.32
Atan	1	0.33
Cos	2	0.25
Exp	2	0.14
Isqrt	2	0.45
Log	2	0.42
Sin	2	0.25
Tan	2	0.35

Approximate results, but ok for a wide range of applications

Example of an Accuracy Test





Well known behaviour of the "Quake III" inverse square root

Examples from the LHC Experiments

Examples from the experiments

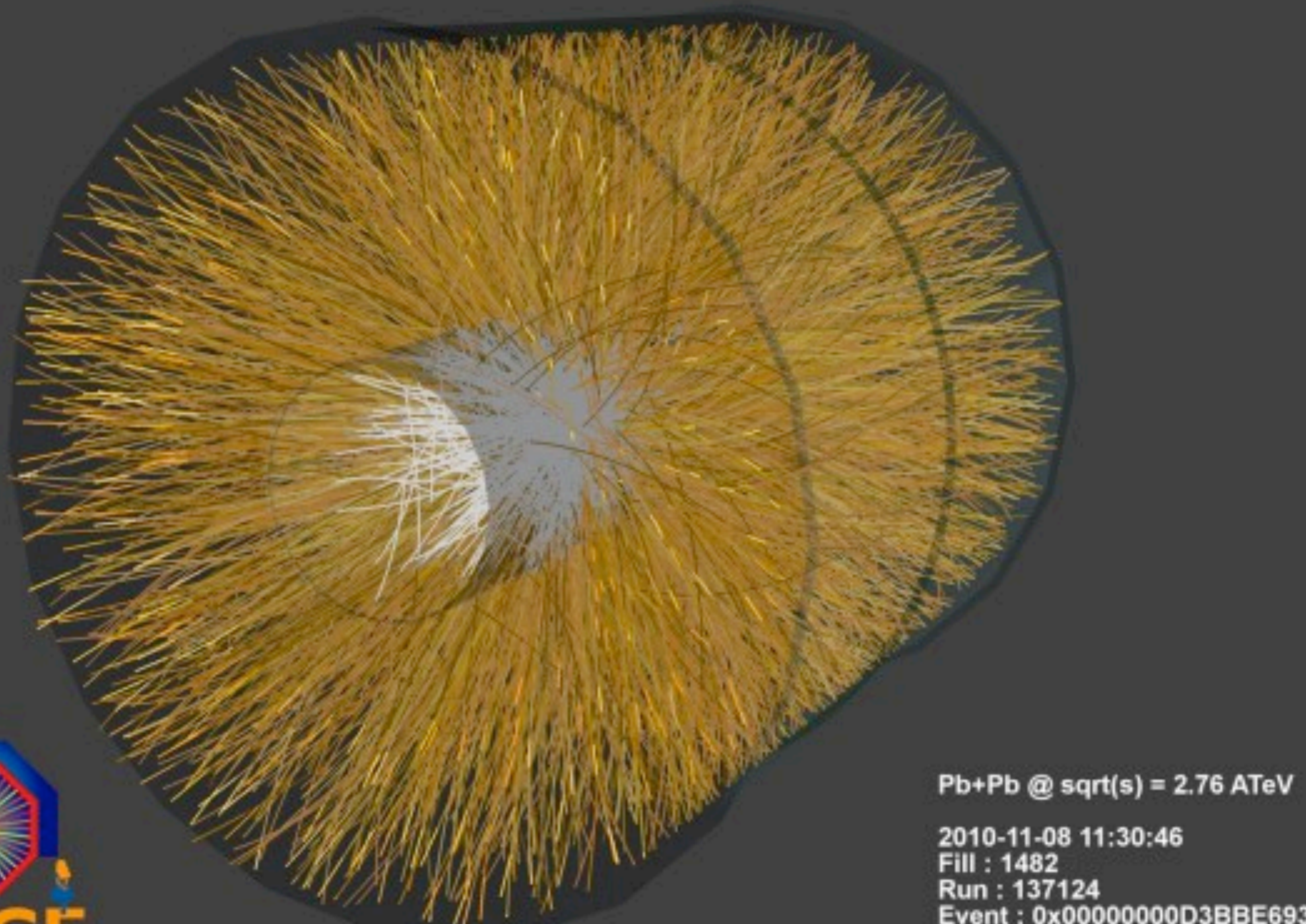
Methodology:

- Replace calls to Libm functions with the VDT ones: *LD_PRELOAD*
- No hotspots but an overall replacement

Caveats:

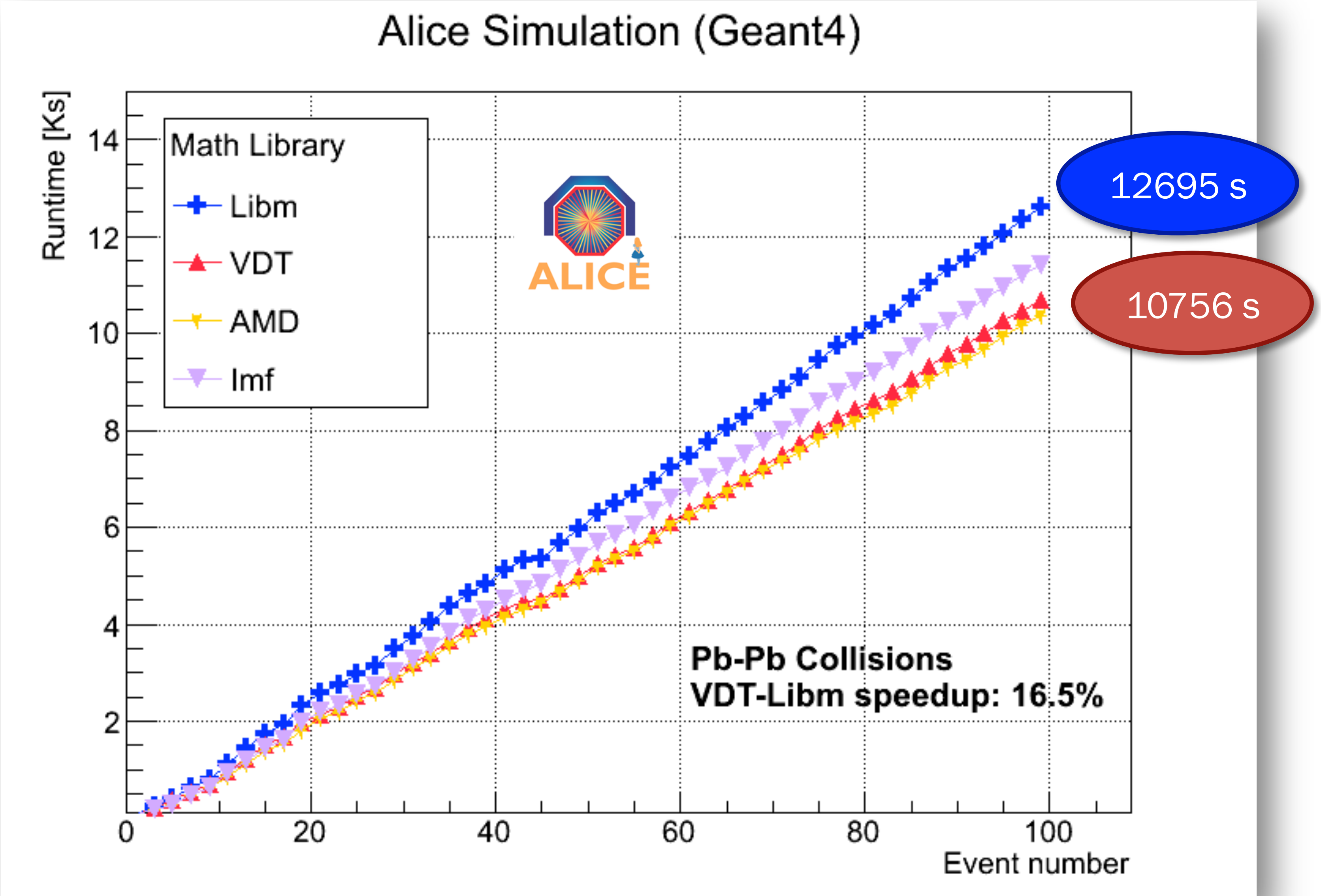
- Not the best way to proceed: no case by case control of accuracy… But the less intrusive!
- Code performance improvements cited: conservative – no inlining with preload!
- Physics performance: maximum variation obtainable (all calls replaced!)

The Alice Case



ALICE

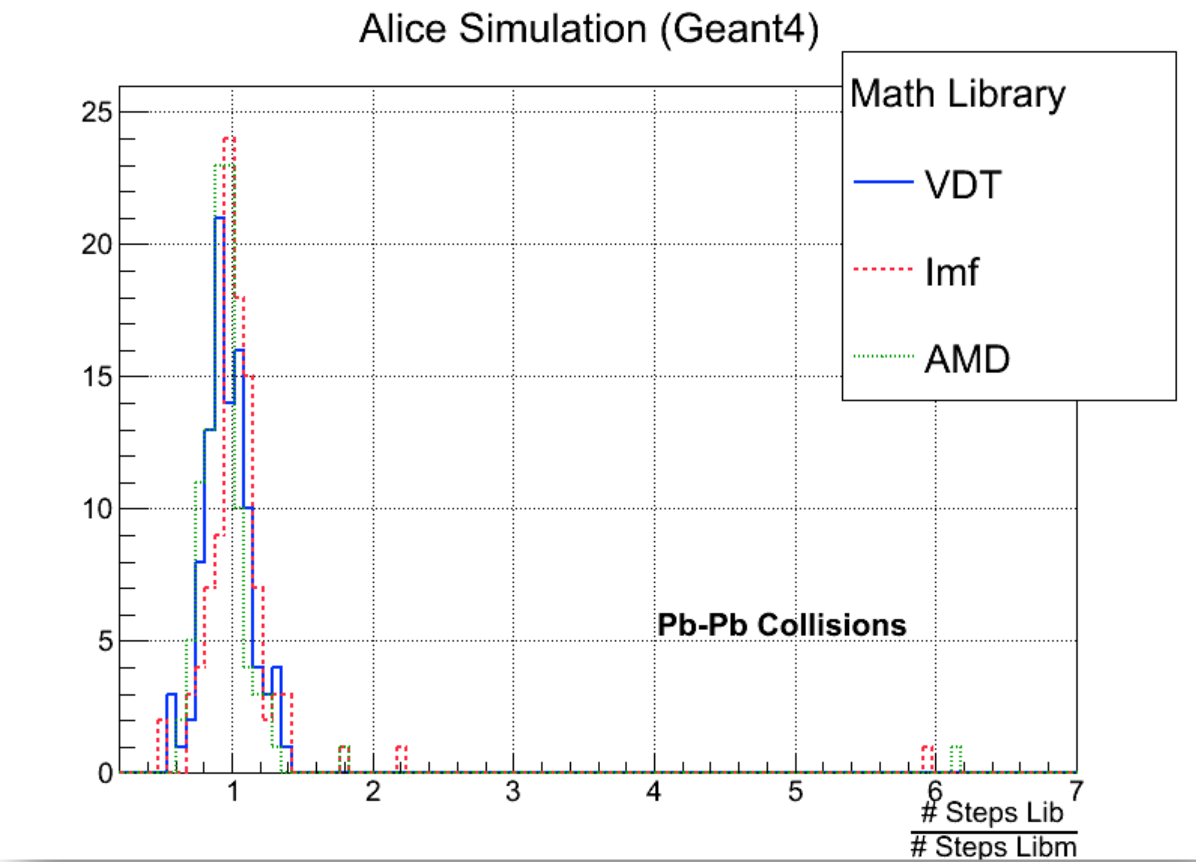
Alice Simulation: Switching to VDT



Alice Sim: Preliminary Validation

Output of simulation validated in terms of number "simulation steps"

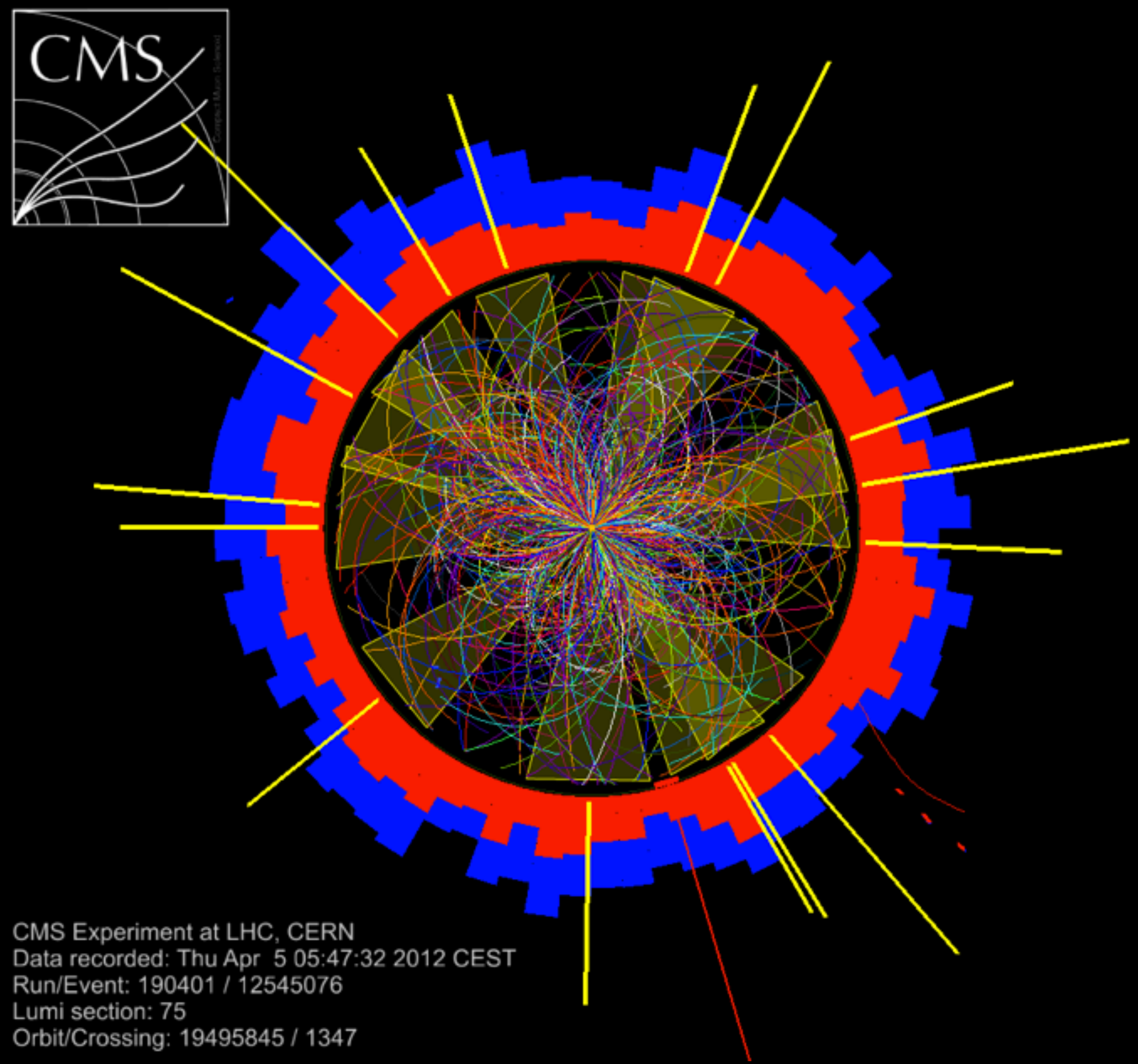
- Necessary to have a similar number of simulation steps in order to have compatible results!
- Number of steps for the AMD Libm, VDT and IMF cases compared to Libm:
  - Nicely distributed around 1



Some work would be needed for final sign-off, Results already very positive!

S. Wenzel

The CMS Case


CMS
CMS Experiment at LHC, CERN
Data recorded: Thu Apr 5 05:47:32 2012 CEST
Run/Event: 190401 / 12545076
Lumi section: 75
Orbit/Crossing: 19495845 / 1347

Full Sim: Costs of the Functions 1/2

Function	Runtime % 50 ev.	Runtime % 1 ev.
_ieee_754_log	3.77	13.30
_ieee_754_exp	1.80	5.85
_ieee_754_atan2	1.74	0.75
sincos	0.60	0.37
_ieee_754_pow	0.51	0.45
__exp1	0.29	0.26
_ieee_754_log10	0.16	0.08
_ieee_754_atan2f	0.15	0.03
TOTAL	**9.02**	**21.9**

Performance profile of 2 jobs obtained:

1) 50 events (~5k seconds)
2) 1 event (310 seconds): estimator of the initialisation overhead

Numbers for 1) are in black in the table, the ones for 2) red in the table

Self costs shown, callees are not considered!

Full Sim: Costs of the Functions 2/2

Libm: not only calculate function value, but also performs other operations:

- Is argument Nan?
- Raise floating point exceptions: e.g. fp_inexact

Isnan callers	Runtime % 50 ev.	1 ev.
log	0.11	0.38
pow	0.04	0.05
log10	0.02	0.02
TOTAL	**0.17**	**0.45**

Part of the callees of the math functions

Feraiseexcept Callers	Runtime % 50 ev.	1 ev.
__ieee_754_exp	3.67	10.5
__ieee_754_pow	0.48	0.63
cos	0.13	0.18
sin	0.07	0.03
TOTAL	**4.36**	**11.33**

These checks do have a cost!

- *Do we need them in production?*

Total cumulative cost (self + callees)
50 evts: ~13.5%
1 evt: ~33.7%

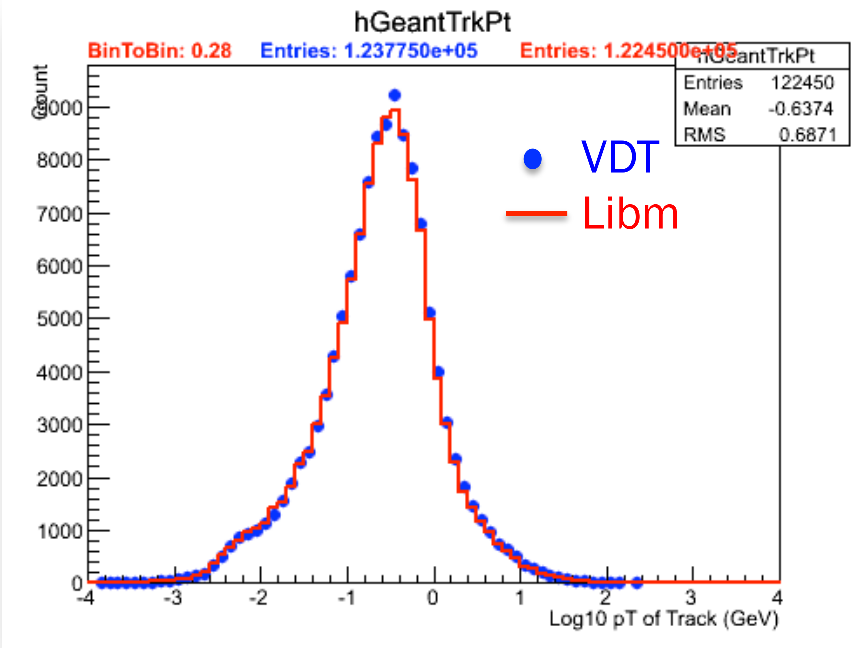
CMS Simulation: Switching to VDT

Result:

- 9% speedup achieved (FullSim 50 Events)
- 25% speedup achieved (FullSim1 Event – Initialisation cost)

Validation:

- Good compatibility of results assessed
  – Use standard CMS histograms
- Changes expected and found
  – Different accuracy of the functions!
  – Experts' validation must sign-off!

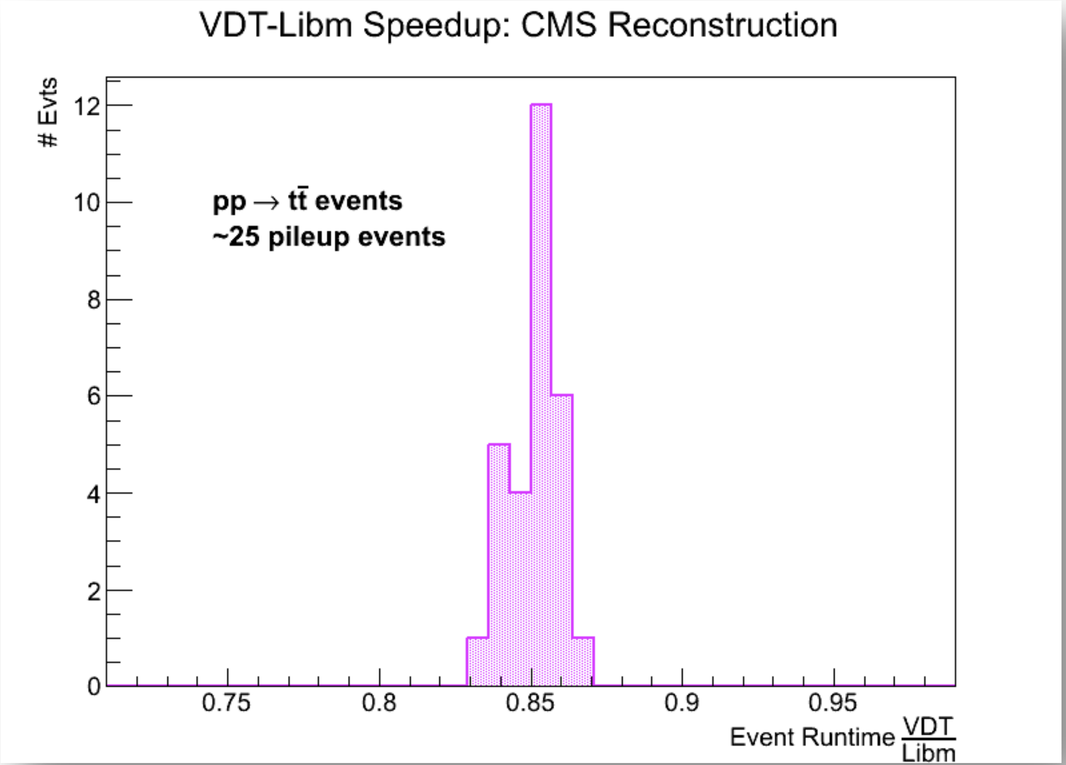


T. Hauth

CMS Reco: Switching to VDT

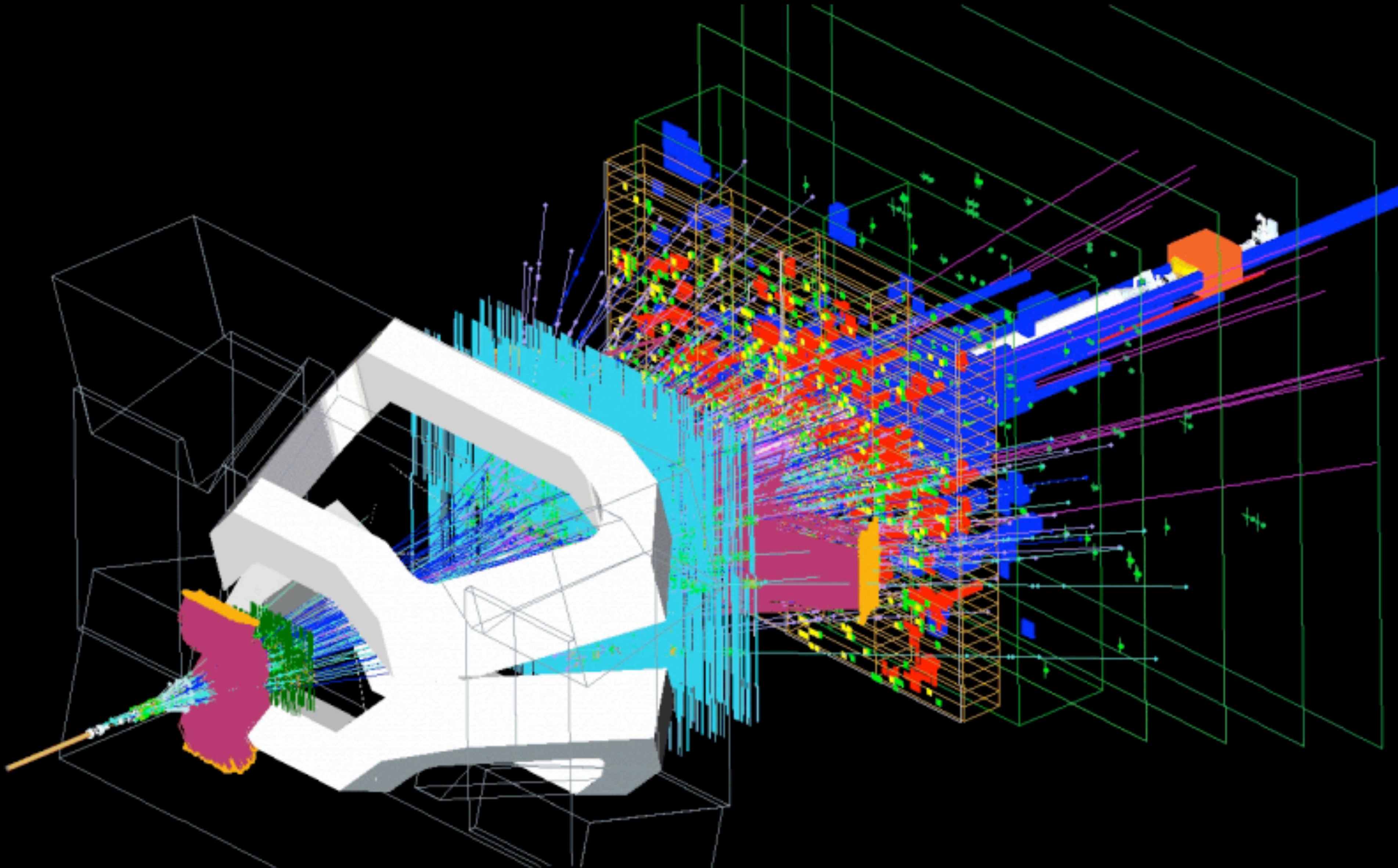
- Reconstruct simulated top-antitop events + ~25 pileup collisions
- 15 % speedup in event loop (w/o initialisation)
  - Symbol *magfieldparam::TkBfield::Bcycl: 77.3s* → *18.18s* (extensive usage of *exp*)
- Very good agreement of Physics performance!
  - 120.000 plots compared, marginal differences

Interesting opportunities for fast mathematical functions even with reduced accuracy!



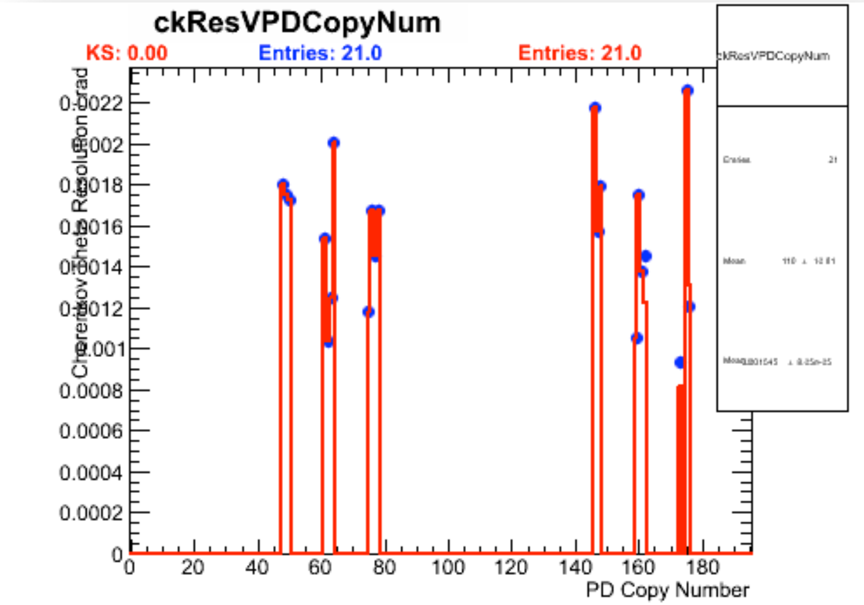
T. Hauth

The LHCb Case



LHCb Reco: switching to VDT

- Portion of total runtime due to math functions is relatively small: 8.7% of the total budget
- Interesting optimisations (like ad hoc polynomial expansions) already introduced in LHCb software
- Event loop: 3.5% speedup measured (in event loop, no initialisation)



Example of different Physics performance

LHCb reconstruction is robust against small changes in the math functions accuracies

Is This the End?

The ultimate mathematical library:

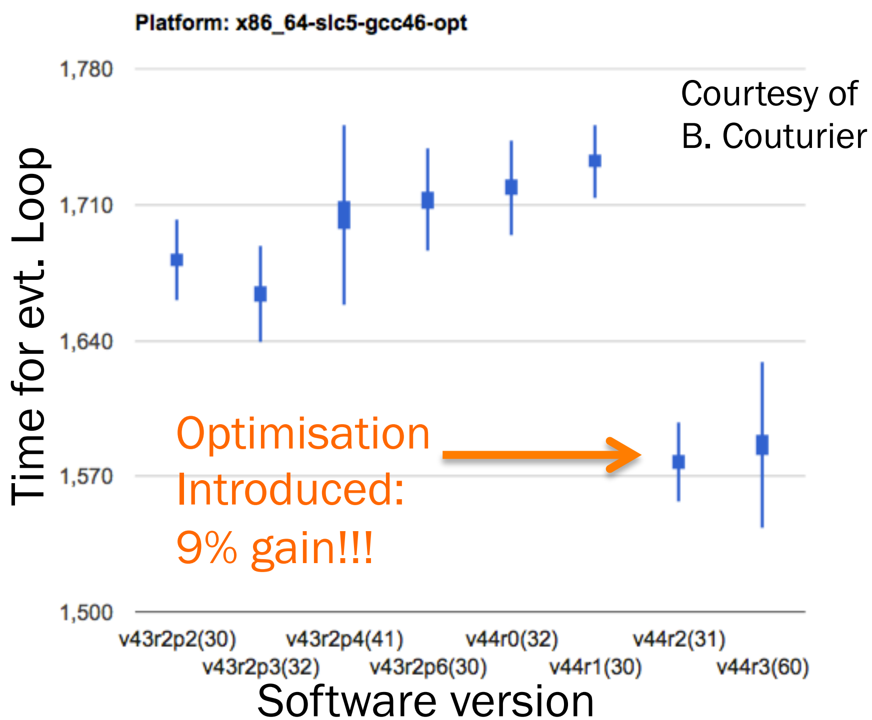
- A high performance "metalibm"
- Automatically obtain implementation of a function for a particular range and accuracy
- See Florent's presentation!

High quality polynomial approximations:

- Credible alternative to several of the FORmulas TRANslated from literature present in HEP code
- Replacing full formula: faster and gives better control than replacing just the math functions in it

Polynimials: an example from LHCb

- High precision measurements involving likelihood ratios
- log(exp()-1) function involved
- Maxima used to find Pade' approximants in 3 different ranges



Maxima is a useful tool to play with Pade approximants!
http://maxima.sourceforge.net

See:
http://lhcb-release-area.web.cern.ch/LHCb-release-area/DOC/davinci/releases/latest/doxygen/d9/d33/class_rich_1_1_rec_1_1_global_p_i_d_1_1_likelihood_tool.html#aa288748034a29a8caffcc2cabd01d2fb

Conclusions

- Mathematical functions account for a substantial portion of the overall runtime of HEP applications
- Traditional Libm may have become too expensive (although a perfect reference)
- Different alternatives to Libm available. Potential gains attractive: 10% – 20% speedups not unreasonable for typical reconstruction / simulation workflows of LHC experiments
- Replacing full formulas with high quality polynomials / Pade' approximations
  - Can be faster than replacing the single math functions
  - Accuracy is better controlled

Backup

VDT Array and Scalar Signatures

VDT provides "array" and "scalar" signatures

Scalar signature: `T(T)`

- Example: double `y = vdt::fast_exp(x);`

Array signature: `void(const unsigned int,T*,T*)`

- Example: `vdt::fast_expv(11, input_array, output_array);`

Array signatures trivially autogenerated: script steered by CMake

```
void fast_expv(const unsigned int n, float* in, float* out{
    for (unsigned int i=0;i<n;++i)
        out[i] = fast_exp(in[i]);}
```

All the difficulties dropped on the compiler

Similar generator script is in place to autogenerate signatures to allow library preload if requested.

Single Precision

Function	VDT	VDT SSE	svml SSE	VDTAVX	svml AVX
Expf	6.76	1.9	1.33	2.07	1.38
Logf	13.1	2.48	1.70	1.90	1.62
Sinf	12.2	2.69	1.60	2.00	1.44
Cosf	10.1	2.45	1.89	1.71	1.82
Tanf	12.4	3.31	1.99	2.58	1.86
Asinf	8.93	2.00	2.37	0.71	2.19
Acosf	9.42	2.16	2.74	0.72	2.55
Atanf	6.01	1.92	2.00	0.70	1.79
Isqrtf	2.99	0.58	0.1*	0.42	0.1*

Time in nanoseconds per value calculated

Single Precision

Function	Libm	VDT	VDT SSE	VDT AVX
Expf	180	6.76	2.50	2.07
Logf	12.6	13.1	2.48	1.90
Sinf	180*	12.2	2.69	2.00
Cosf	180*	10.1	2.45	1.71
Tanf	183*	12.4	3.31	2.58
Asinf	12.1	8.93	2.00	0.71
Acosf	14.6	9.42	2.16	0.72
Atanf	10.8	6.01	1.92	0.70
Isqrtf	5.02	2.99	0.58	0.42

Time in nanoseconds per value calculated

*Reducing range to [-10,10]

Func.	libm
Sinf	17.9
Cosf	18.4
Tanf	26.1

Single Precision

	MAX VDT	AVG VDT
Acosf	7	0.48
Asinf	3	0.6
Atanf	2	0.37
Cosf	6	0.24
Expf	6	3.36
Isqrtf	7	3.7
Logf	2	0.26
Sinf	6	0.24
Tanf	6	0.52

VDT Building Blocks

Starting point: the well known Cephes library

- Developed by Stephen Moshier in the eighties in C
- Pade' approximation
- Single, double and quad precisio

Tool: a modern compiler like GCC 4.7

- Autovectorisation capabilities of GCC are getting more and more mature
- Go the extra mile: make the functions not only fast, but autovectorisable!