The VDT Mathematical Library: A Modern Reimplementation of Cephes

V. Innocente CERN PH-SFT & CMS

T. Hauth CERN CMS

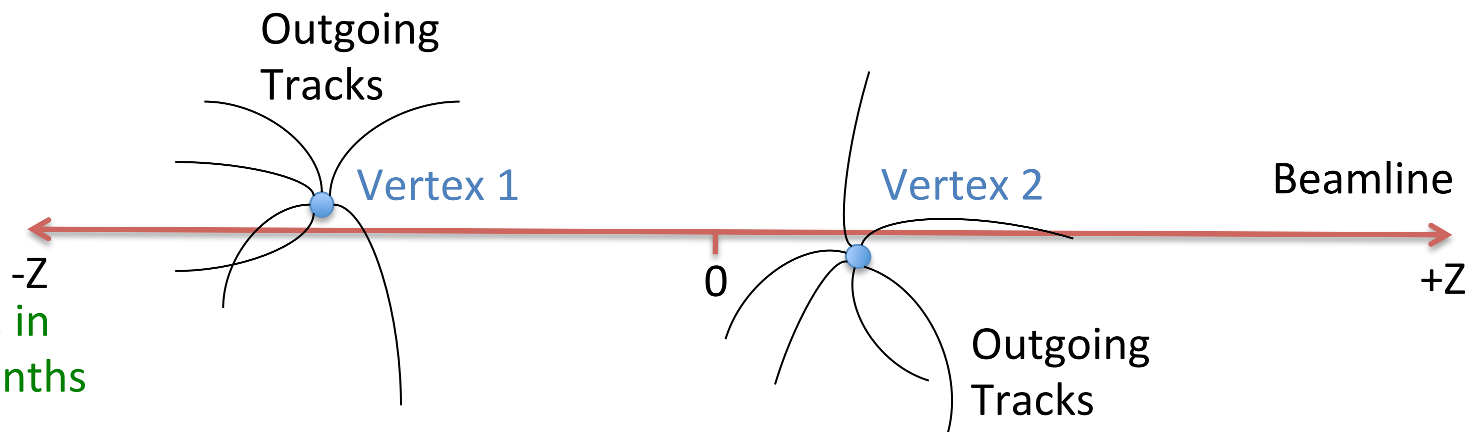
D. Piparo CERN PH-SFT & CMS

Outline

- Motivation
- The Building Blocks of VDT
- Timing and accuracy measurements

CMS Vertex Clustering

Reconstruct z coordinate of collisions vertices with deterministic annealing algorithm



2 shown, in reality tenths of them!

This algorithm accounted for 2.5% of overall reconstruction time and 60% of the time budget of the algorithm was represented by calls to *exp*

Motivation

- Floating point calculations: for an LHC experiment represent a substantial portion of the overall time budget of a typical data processing workflow
- A considerable part of these calculations is due to transcendental mathematical functions
  - i.e. exp, log, atan

Requirements and compromises for the CMS experiment:

- A fast implementation of such functions
- An open source product
- Possible to pay a price in terms of accuracy

That's why we developed the VDT mathematical library

The Building Blocks of VDT

Starting point: the well known Cephes library

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

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!

THE VDT MATHEMATICAL LIBRARY

A collection of transcendental mathematical functions

- Which are *fast* and *approximate*
- Licenced under *LGPL3*
- Which can be used in loops autovectorised by the compiler

The functions implemented at the moment are (double and single precision):

- ✓ Exp, Log
- ✓ (A)Sin, (A)Cos, (A)Tan
- ✓ 1/√ (different precision levels)

Single and Double precision implementations are different.

Signatures (identical for single precision):

1. double (double) – referred to as scalar signature
2. void(unsigned int, double*, double*) – referred to as array signature (just a simple for loop calling the scalar version)

The VDT Mathematical Library

Existing code can be converted easily ("sed")

- VDT functions symbols: *fast_<function_name>[f][v]*
  - *f* for floats, *v* for arrays

Functions are *inline*:

- The compiler does a lot of work by itself
- Portability for present *and future* targets
  - ARM, KNC, GPUs..

Scalar functions usable in loops to exploit autovectorisation

Distributed with tools for accuracy and speed diagnostics

Comparisons with Existing Libraries

Three references considered for comparisons of speed and accuracy of VDT: Libm, VC and SVML

VC (version 0.6.1):

- Open source
- Uses intrinsics
- Inspired as well by Cephes
- Exp, Tan, Acos not provided yet

SVML:

- Intel product
- Closed source
- Used with GCC
  - *... -mveclibabi=svml –L<SVML LIB DIR> -lsvml …*

TIMING MEASUREMENTS

The Setup of the Measurements

Machine:

Intel Core i7-3930K CPU @ 3.20GHz (AVX SIMD instructions supported)

Compiler:

GCC 4.7.1, *-Ofast* flag

Input:

- 1 Million random numbers repeated 150 times
- Subtraction of call overhead using "Identity" function: f(x)=x

Double Precision: VDT & 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 nanoseconds per value calculated

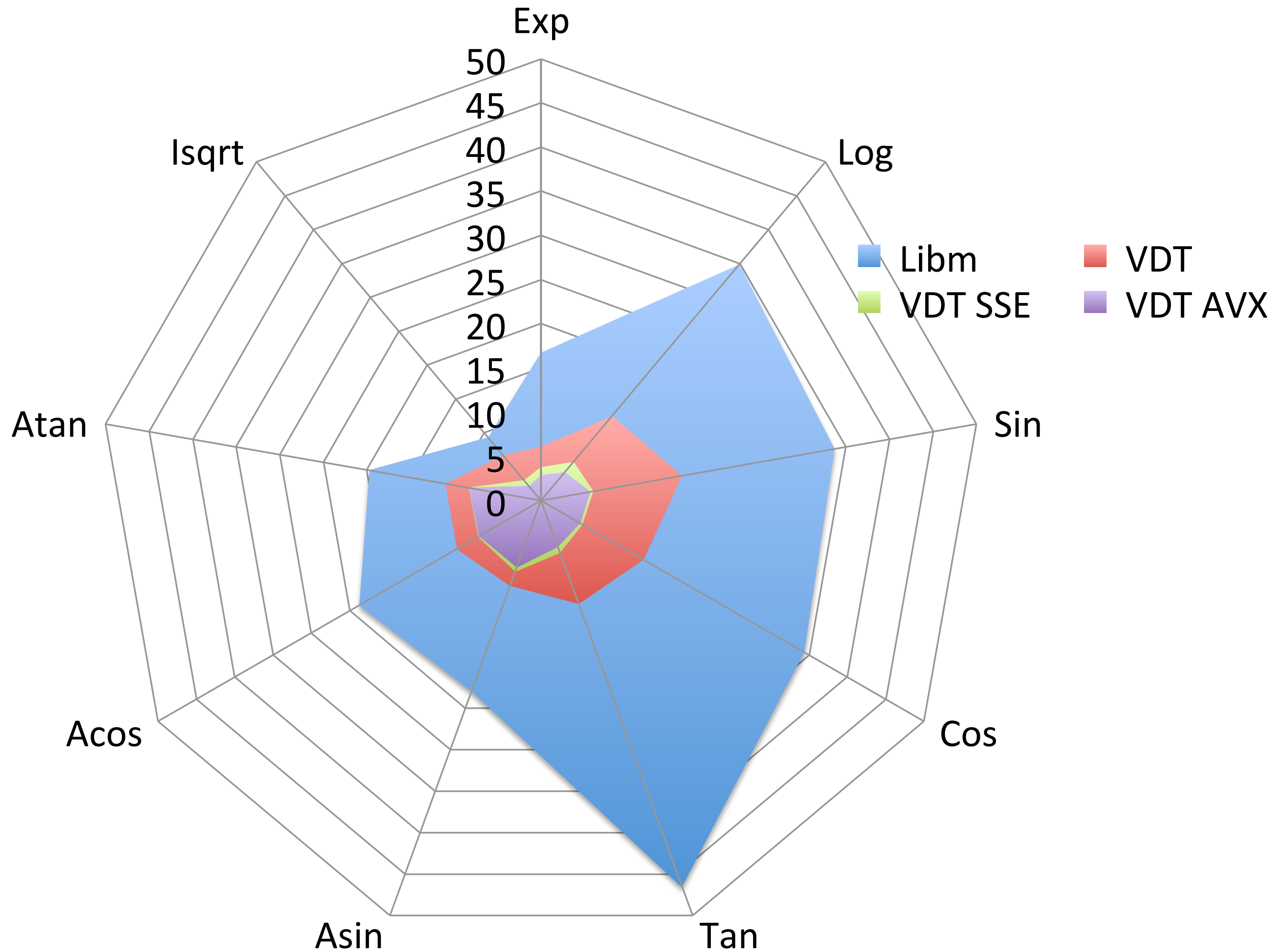
VDT scalar functions: already much faster than Libm

- Speedups of about 4x can be achieved.

Speedup from scalar to SSE more significant than from SSE to AVX

- Some overhead is present

Double Precision: VDT & Libm



Netplot showing time/execution in ns for Libm and VDT scalar and array signatures. The smaller the area, the faster the function.

Double Precision: VDT, SVML & VC

Function	VDT SSE	svml SSE	VDT AVX	svml AVX	VC AVX
Exp	3.84	3.02	2.93	3.08	X
Log	5.72	5.27	4.25	4.72	5.46
Sin	6.03	4.57	5.66	4.31	3.6
Cos	5.36	5.03	5.06	4.79	3.06
Tan	6.27	4.86	5.61	4.67	X
Asin	8.65	6.65	8.16	6.34	5.44
Acos	8.19	6.18	8.15	5.94	X
Atan	8.26	8.36	8.3	8.34	6.22
Isqrt	2.95	5.24*	2.1	5.22*	5.07

Time in nanoseconds per value calculated

Svml and VDT: speed is comparable

VC slightly faster *but:*

- Sin/Cos: power series: see accuracy
- Pade' approximant for single precision used for doubles: see accuracy

** Obtained with 1/sqrt*

Single Precision: VDT & Libm

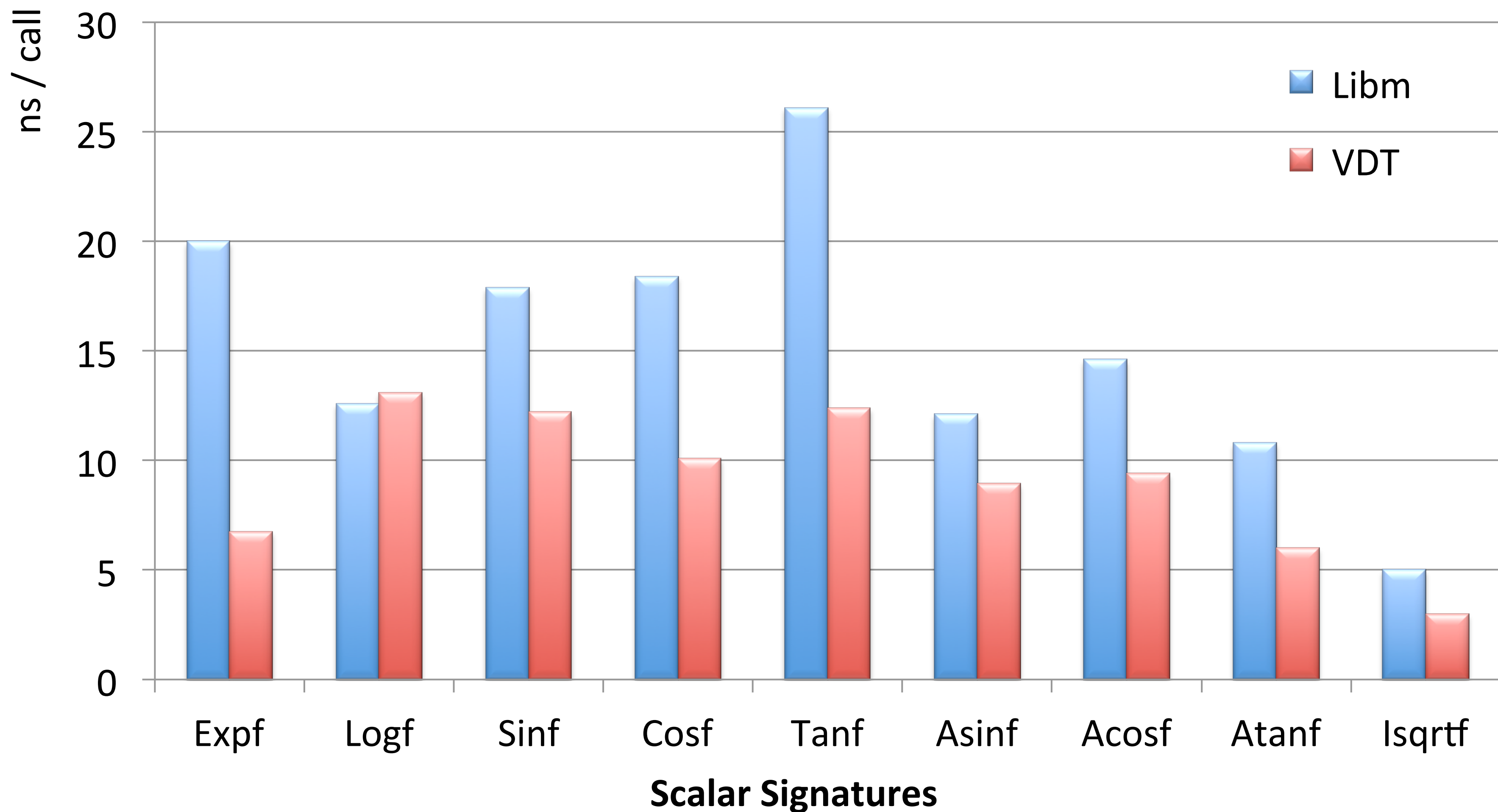
Function	Libm	VDT	VDT SSE	VDT AVX
Expf	180	6.76	2.50	2.07
Logf	12.6	13.1	2.48	1.90
Sinf	180*	12.2	2.69	2.00
Cosf	180*	10.1	2.45	1.71
Tanf	183*	12.4	3.31	2.58
Asinf	12.1	8.93	2.00	0.71
Acosf	14.6	9.42	2.16	0.72
Atanf	10.8	6.01	1.92	0.70
Isqrtf	5.02	2.99	0.58	0.42

Time in nanoseconds per value calculated

*Reducing range to [-10,10]

Func.	libm
Sinf	17.9
Cosf	18.4
Tanf	26.1

Single Precision: VDT & Libm



ACCURACY/SPEED TRADE-OFF

Inverse square root is implemented as an iterative process

- Two levels of accuracy implemented (2 and 3 iterations)

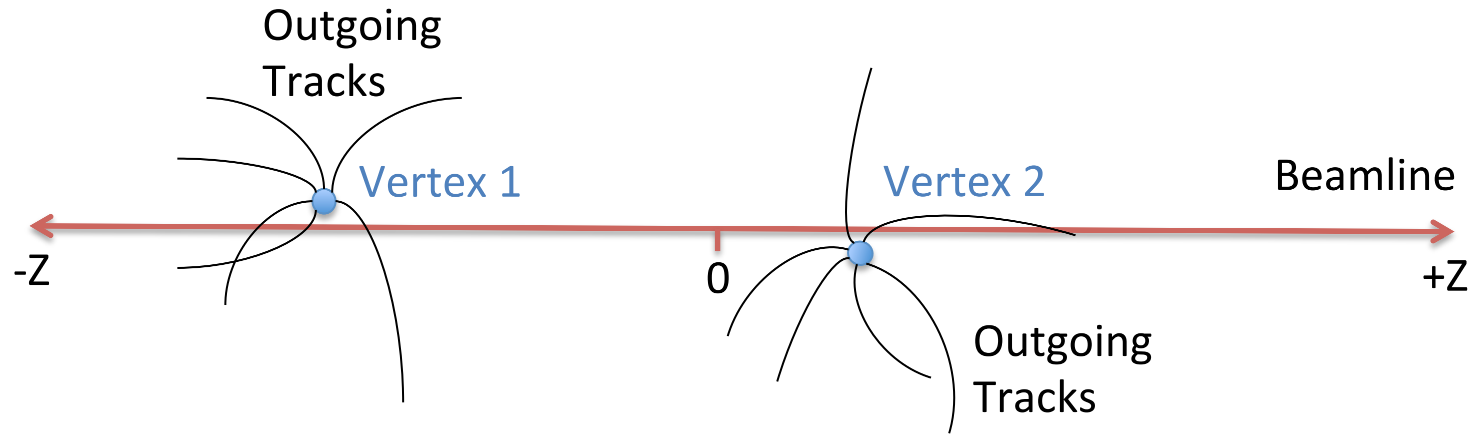
Function	Libm*	VDT	VDT SSE	VDT AVX
Fast_isqrt	9.3	6.7	2.9	2.1
Appr_isqrt	9.3	4.4	2	1.5

Double precision, time in nanoseconds per value

Range / speed Trade-Off possible and probably beneficial in some cases, but not discussed today.

** To be precise, GCC intrinsics.*

An Example: CMS Vertex Clustering



Exp represents 60% of the time consumed by this algorithm

→Perfect case for a fast mathematical library

Moving to VDT:

- Algorithm 2x faster
- From 2.5% to 1.2% of the total reconstruction time!

Improvement used in production by CMS since January

ACCURACY MEASUREMENTS

About the Measurement

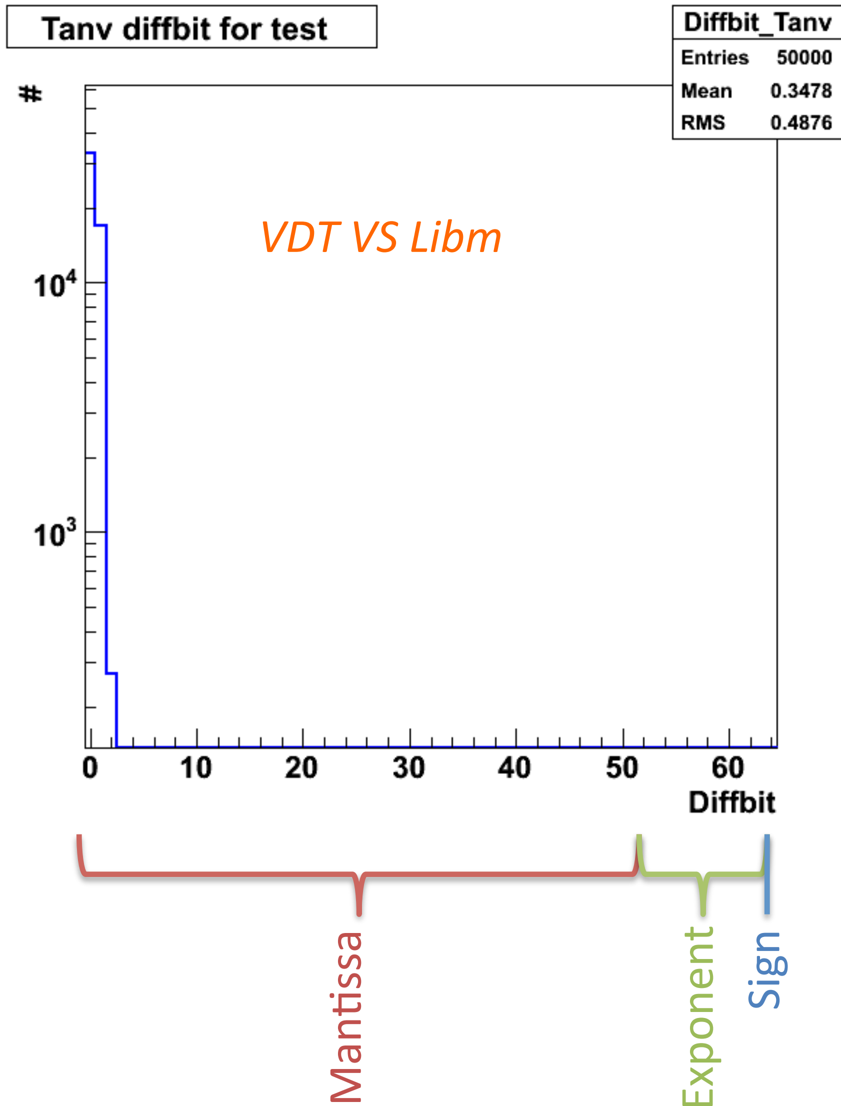
- The results yielded by the VDT functions validated against Libm.
- Same procedure applied to Svml and VC.
- The results are presented in terms of most significant different bit.

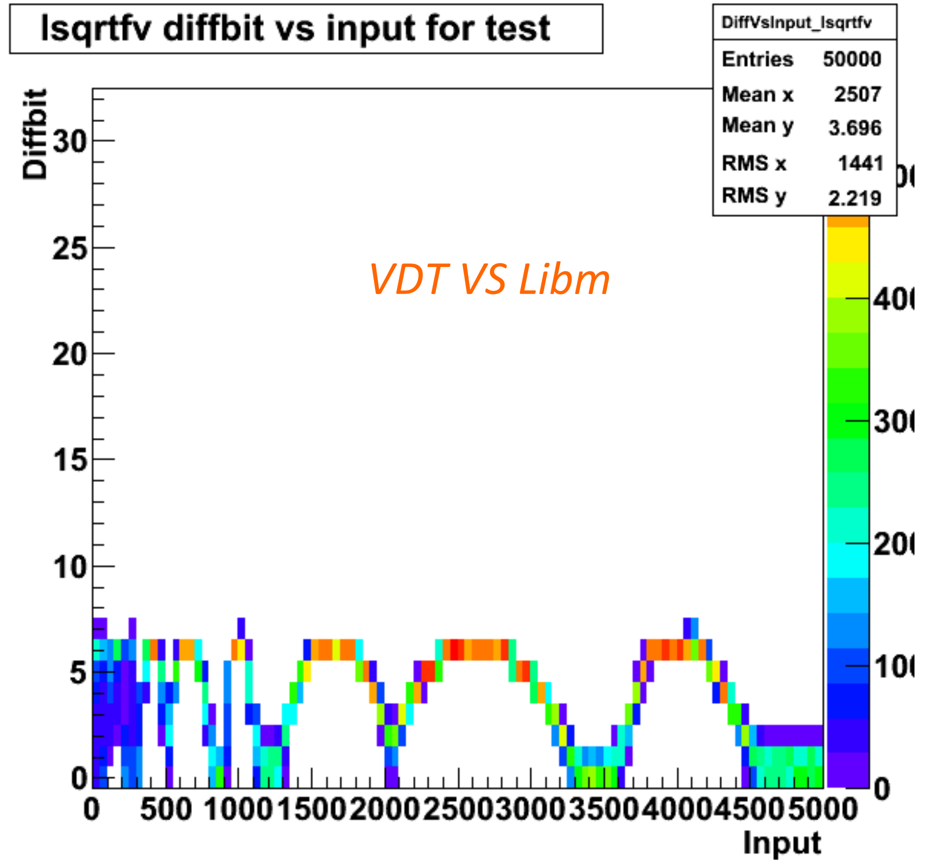
Double Precision

	MAX VDT	AVG VDT	MAX svml	AVG svml	MAX VC	AVG VC
Acos	8	0.39	2	0.79	x	x
Asin	2	0.32	2	0.24	26	24.3
Atan	1	0.33	2	0.28	28	2.97
Cos	2	0.25	2	0.35	35	23.3
Exp	2	0.14	2	0.43	x	x
Isqrt	2	0.45	0	x	x	x
Log	2	0.42	1	0	1	0.01
Sin	2	0.25	2	0.35	35	23.3
Tan	2	0.35	3	0.49	x	x
Apr_isqrt	19	12.65	x	x	x	x

- Svml accuracy comparable with the VDT one
  - Values approximate but still ok for a wide range of applications
- VC: usage of single precision pade‘ approximation for doubles, loss of precision in the range reduction

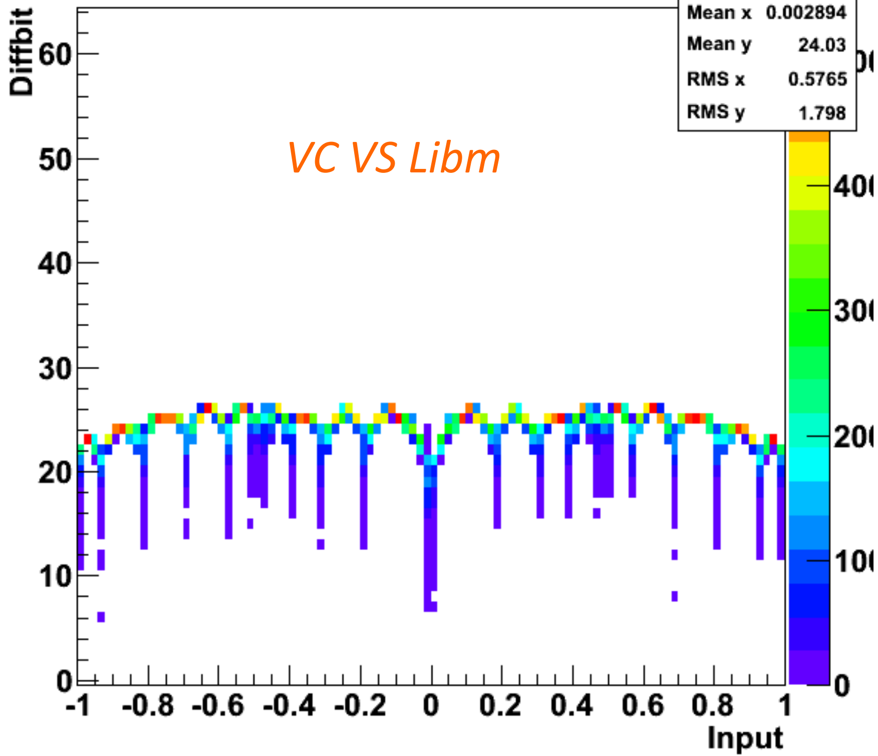
ACCURACY PLOTS

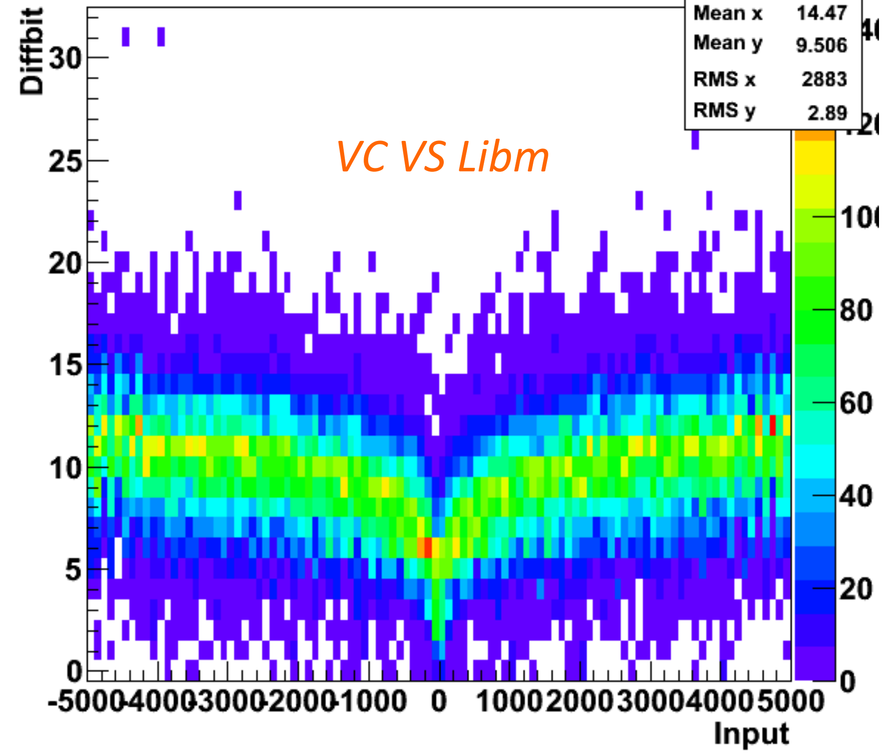




Well known behaviour of the "Quake III" inverse square root

Accuracy Plots





Conclusions

A modern version of the Cephes library was implemented

VDT provides both scalar and array signatures

- Scalar: speedups of 2x-3x wrt Libm not unreasonable
- Array: factors of 10x achieved

Automatic usage of vector units via the compiler

- GCC allows high quality vectorisation
- Scalar functions usable in loops to exploit autovectorisation: same speedup as for array signatures
- Simpler than intrinsics usage

Open Source (LGPL3)

Links:

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

http://svnweb.cern.ch/world/wsvn/vdt

BACKUP

Single Precision: VDT, SVML & VC

Function	VDT	VDT SSE	svml SSE	VDTAVX	svml AVX	VC AVX
Expf	6.76	1.9	1.33	2.07	1.38	X
Logf	13.1	2.48	1.70	1.90	1.62	2.31
Sinf	12.2	2.69	1.60	2.00	1.44	1.44
Cosf	10.1	2.45	1.89	1.71	1.82	1.09
Tanf	12.4	3.31	1.99	2.58	1.86	X
Asinf	8.93	2.00	2.37	0.71	2.19	2.05
Acosf	9.42	2.16	2.74	0.72	2.55	X
Atanf	6.01	1.92	2.00	0.70	1.79	2.09
Isqrtf	2.99	0.58	0.1*	0.42	0.1*	0.03

Time in nanoseconds per number

Nice scaling going from scalar to SSE and to AVX

** Obtained with 1/sqrt*

SINGLE PRECISION

	MAX VDT	AVG VDT	MAX svml	AVG svml	MAX VC	AVG VC
Acosf	7	0.48	2	0.4	x	x
Asinf	3	0.6	3	0.58	2	0.31
Atanf	2	0.37	2	0.37	2	0.37
Cosf	6	0.24	2	0.33	31	9.51
Expf	6	3.36	2	0.45	x	x
Isqrtf	7	3.7	x	x	x	x
Logf	2	0.26	1	0.03	1	0
Sinf	6	0.24	2	0.33	32	9.51
Tanf	6	0.52	3	0.52	x	x
Apr_isqrtf	15	13.81	x	x	x	x

Quake III Fast Isqrt

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

The important piece of the implementation is the "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) {

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

        return y;
}
```

PADE' APPROXIMATION

Operational Definition:

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)*

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