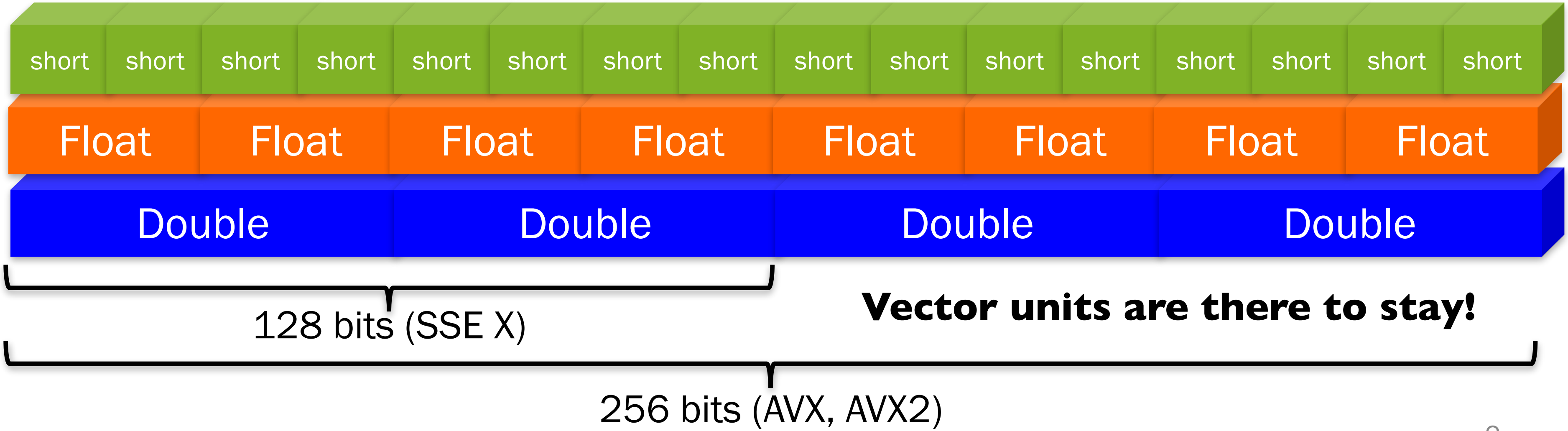
Modernising ROOT: Building Blocks for Vectorised Calculations

L. Moneta, D. Piparo, S. Wenzel – CERN

ACAT 2014
– bridging disciplines
1-5 September 2014
Prague, Czech Republic
Europe/Prague timezone

The Hardware's Landscape

- Hardware vendors raise computational power of today's CPUs with increasing support for parallelism:
  - More cores (beyond the scope of this talk)
  - Larger vector units, richer vector instruction sets
- Vector units: perform same operation on multiple data
  - Data parallelism at instruction level
- Peak performance achievable only if vector units are properly used
  - Especially for "extreme" architectures like the Xeon Phi



Vector units are there to stay!

There are different techniques to achieve vectorised code

Autovectorisation

The compiler generates vector instructions automatically for loops fullfilling some conditions, e.g. no external calls, no dependency between iterations. Maximally portable, might become fragile.

Explicit vectorisation

Implement algorithms with special types implying vectorised operations (e.g. 8 packed floats). Usage of instruction set specific intrinsics or, preferably, an abstraction above them.

Libraries

Utilise 3rd party libraries which encapsulate the aforementioned vectorisation strategies, hiding the technical details from the user.

Outline

- ROOT as a toolkit for algebra, numerical computing and statistics
- Fast and vectorisable mathematical functions
- Support for explicit vectorisation
- Geometry/Physics Vector and vector-matrix algebra
- Vectorization in fitting and statistical calculations
- Plans for the future

Mathematical Functions

Mathematical Functions

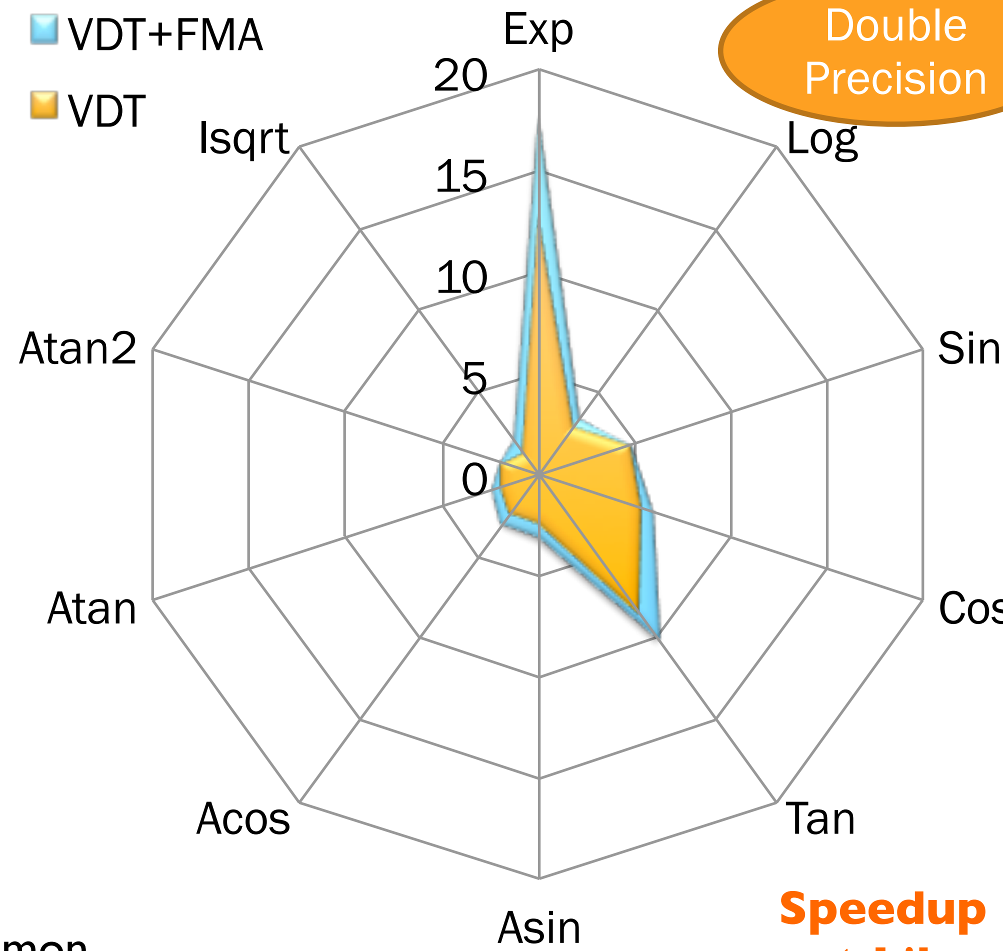
- ROOT provides single/double precision of (a)sin, (a)cos, sincos, (a)tan, atan(2), log, exp and 1/sqrt
- **Fast*, approximate*, inline**
- Symbols names are different from traditional ones:
  - **In the vdt namespace: vdt::fast_<name>**
  - Do not force drop-in replacement, allow full control
- **Functions usable in autovectorised loops**
  - **Array signatures** available: calculate on multiple elements conveniently
- **C++ code only,** no intrinsics: **portability guaranteed**
  - ARM, x86, GPGPUs, Xeon Phi, <future microarchitecture>

*wrt libm implementations

Speedup: ROOT Vs Libm

Fnc.	Libm	VDT	VDT-FMA
Exp	*102*	*8*	*5.8*
Log	*33.3*	*11.5*	*9.8*
Sin	*77.8*	*16.5*	*16.5*
Cos	*77.6*	*14.4*	*13.2*
Tan	*89.7*	*10.6*	*8.9*
Asin	*21.3*	*8.9*	*6.9*
Acos	*21.6*	*9.1*	*7.3*
Atan	*15.6*	*8.4*	*6.7*
Atan	*36.4*	*19.9*	*18.9*
Isqrt	*5.7*	*4.3*	*2.8*

Time in **ns** per value calculated



Double Precision

Speedup wrt Libm

FMA: Fused Multiply Add $d = a + b \times c$

- Operative input range: [-5k, 5k]
- **Speedup factors of >5** not uncommon
- **Effect of FMA clearly visible**
  - **A waste not to profit from it!**

Testbed:

SLC6-GCC48, i7-4770K at 3.50GHz Haswell

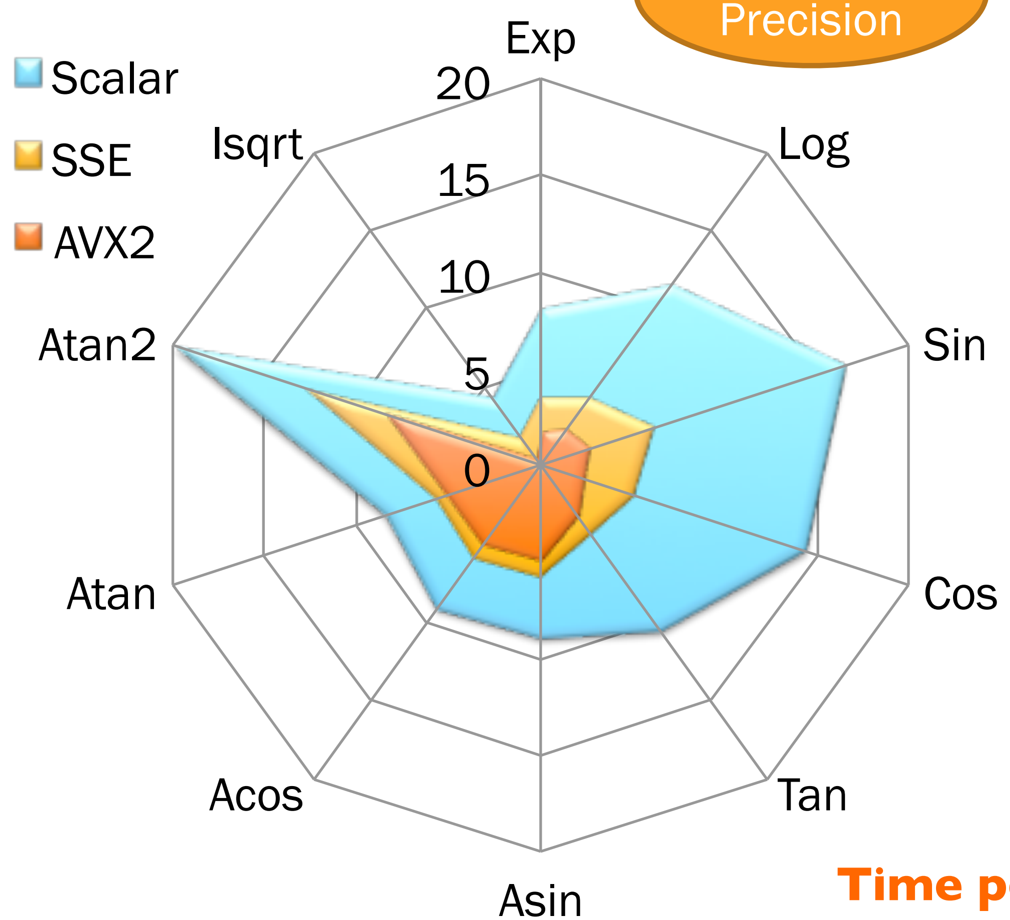
glibc 2.12-1.107.el6_4.4 and ROOT 5.34.20

Effect of vectorisation

Fnc.	Scalar	SSE	AVX2
Exp	*8*	*3.5*	*1.7*
Log	*11.5*	*4.3*	*2.2*
Sin	*16.5*	*6.2*	*2.6*
Cos	*14.4*	*5.1*	*2.3*
Tan	*10.6*	*4.4*	*3.2*
Asin	*8.9*	*5.8*	*5*
Acos	*9.1*	*5.9*	*5.1*
Atan	*8.4*	*5.6*	*5.1*
Atan	*19.9*	*12.7*	*8.4*
Isqrt	*4.3*	*1.8*	*0.4*

Time in **ns** per value calculated

Double Precision



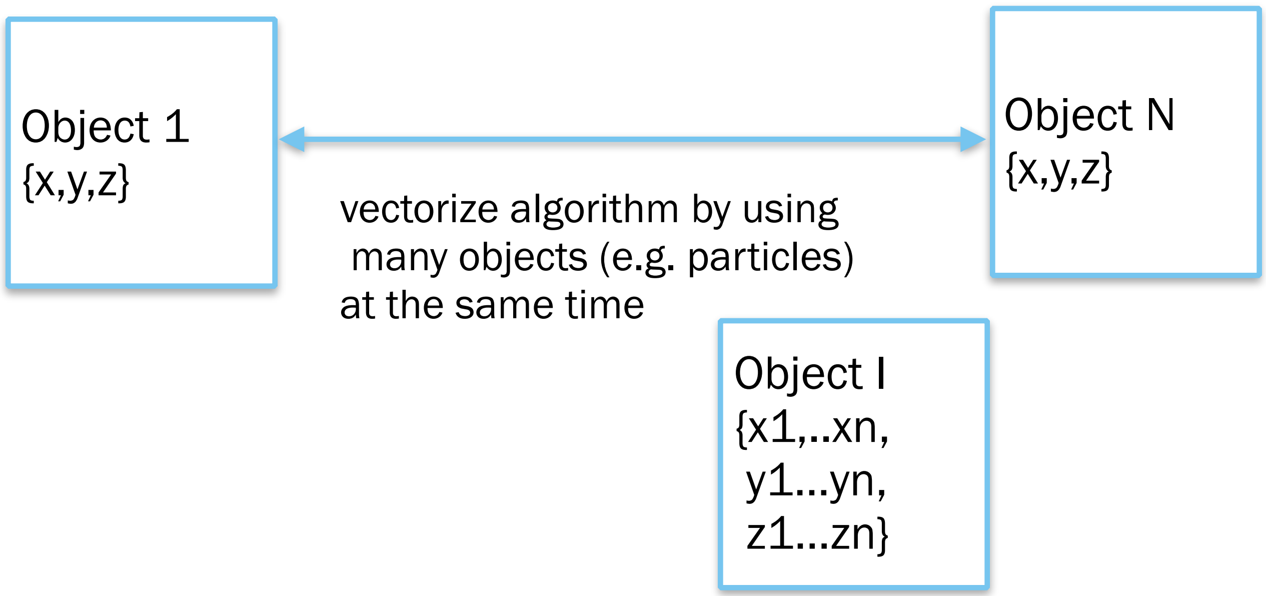
- **Effect of vectorisation clearly visible**

Time per value calculated

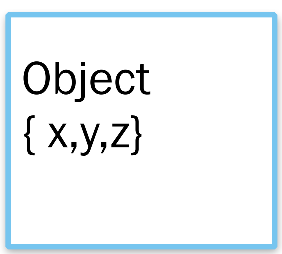
Explicit Vectorization using VC

Horizontal vs Vertical Vectorization

- **Horizontal (external) vectorisation:**



- **Vertical (internal) vectorisation:**



vectorize internally the algorithm operating on a single object
Object data member (e.g. x,y,z) must be stored in a vector

- **Horizontal vectorization**
  - does not require to change algorithmic part of code
  - requires changing input/output data structures (flow of data)
    - need to collect inputs in vectors (i.e. in structure of arrays)
  - use case is limited to the same algorithm applied to several objects
- **Internal vectorization**
  - require changing internal algorithm code to vectorise
  - more difficult to achieve performance gain
    - e.g data sizes might be too small to fit in a vector
  - but use case is more general
- In ROOT we provide both solutions

Vc Library

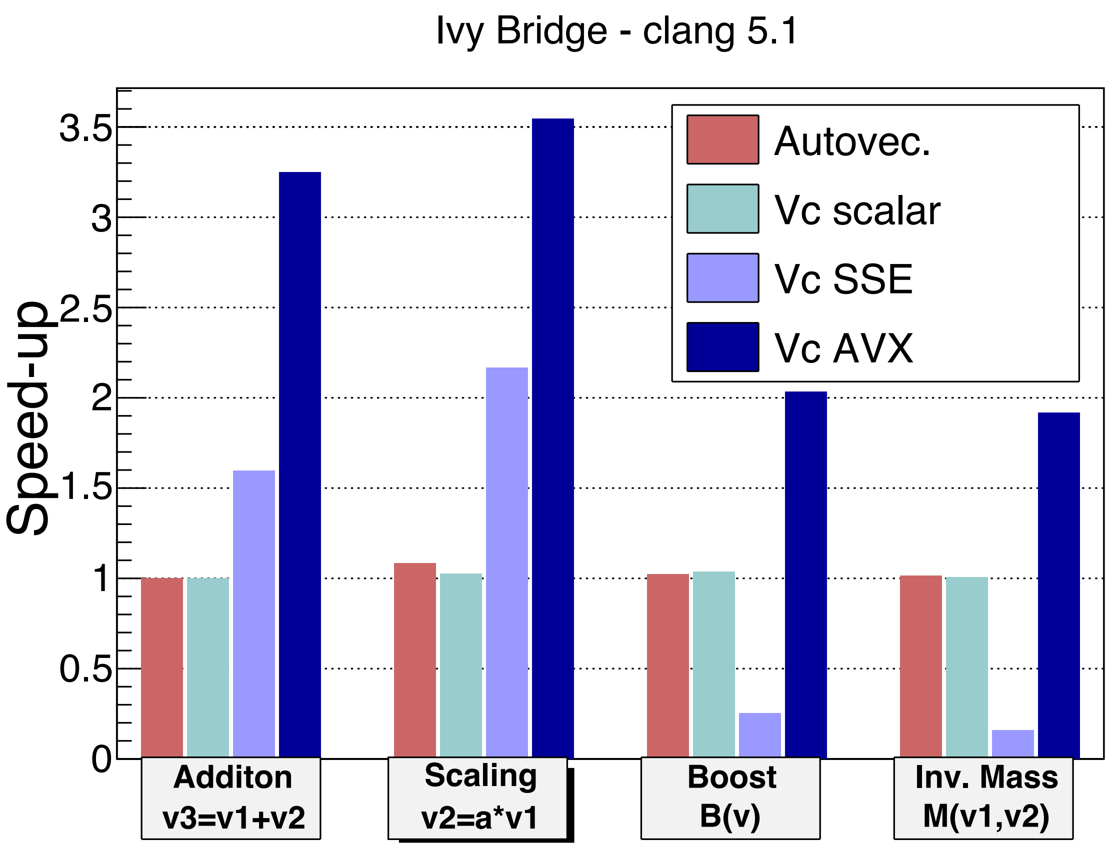
- C++ wrapper library around intrinsic for using SIMD
  - *developed by M. Kretz (Goethe University Frankfurt)*
  - minimal overhead by using template classes and inline functions
- Included in ROOT 6.00 and 5.34 versions
- Provides vector classes (**`Vc::float_v, Vc::double_v`**) with semantics as built_in types
  - one can use **`float_v/double_v`** as **`float/double`**
  - all basic operations between the built_in types are supported (**+,-,/,***)
  - provides also replacement for math functions (**`sqrt, pow, exp, log, sin,`**…)
    - planned to use in the future **vdt**.
- Possible to exploit vectorization without using intrinsic and with minimal code changes
  - e.g. replace `double` → `double_v` in functions

Vc in ROOT - Examples and Performances

- Use Vc for horizontal (external) vectorisation
- Support for replacing data members in ROOT classes:
    - **LorenzVector<PxPyPzE4D<double> >** → **LorenzVector<PxPyPzE4D<Vc::double_v> >**
    - **SMatrix<double, N1, N2 >** → **SMatrix<double_v, N1,N2>**
- Loop on list of objects (vectors, matrices) will be reduced by size of double_v (**NITER = NITER / double_v::Size**)
- Performances results on some basic vector and matrix operation (using double types)
    - Addition of physics vectors, scaling, invariant mass, boost
    - vector product, vector-matrix operations, matrix inversions
- Test using different compilation flags and Vc implementations
    - **VC_IMPL = Scalar, SSE, AVX**
- Compare results with also auto-vectorization
    - compiling using **-mavx -O3 -ftree-vectorize**

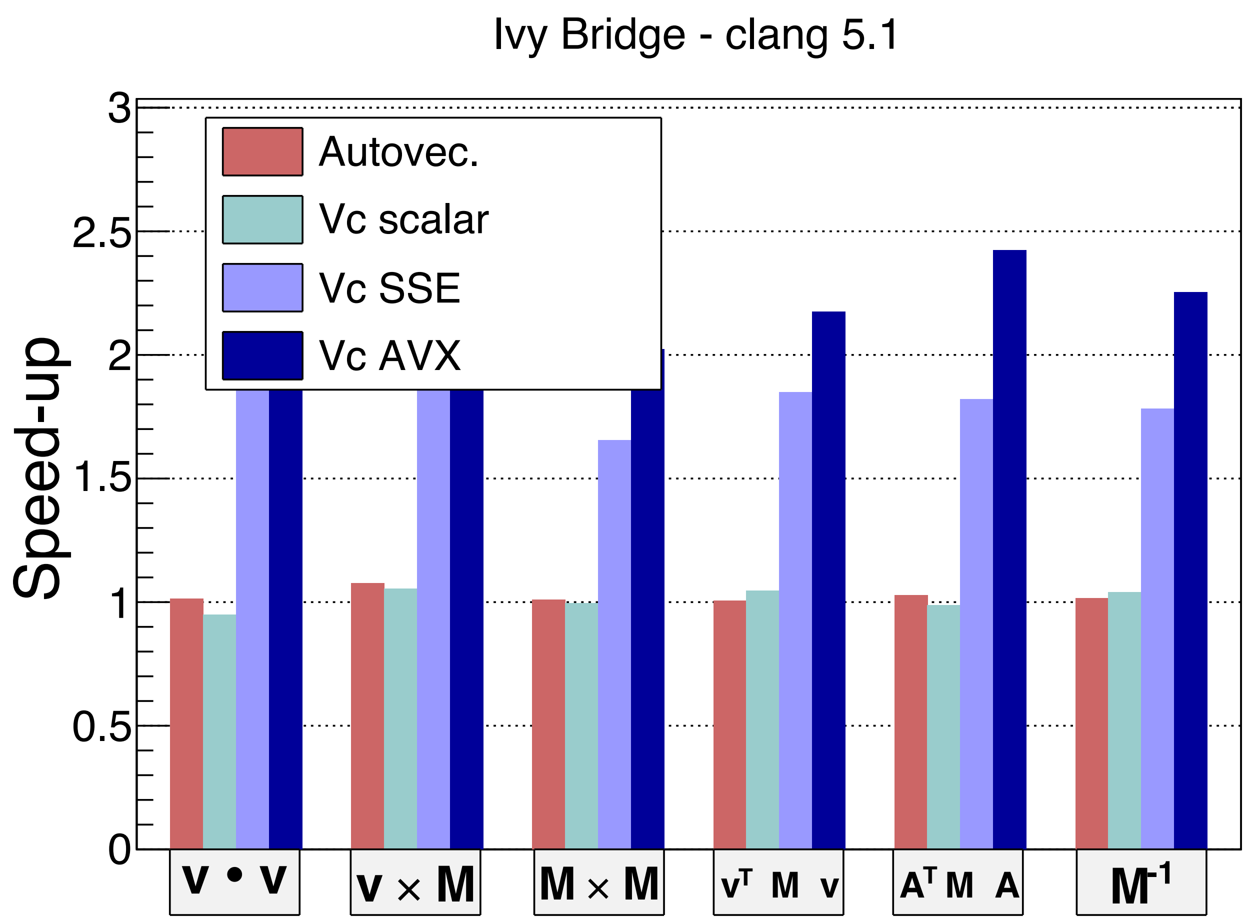
– reference is code compiled with -O2

- Test list of 128: `LorentzVector<double>` vs `LorentzVector<Vc::double_v>`
- Speed-up measured versus a scalar version compiled with -O2



Some compiler optimisation bugs when using SSE implementation ?
Effect not seen when using other compiler (e.g. gcc)

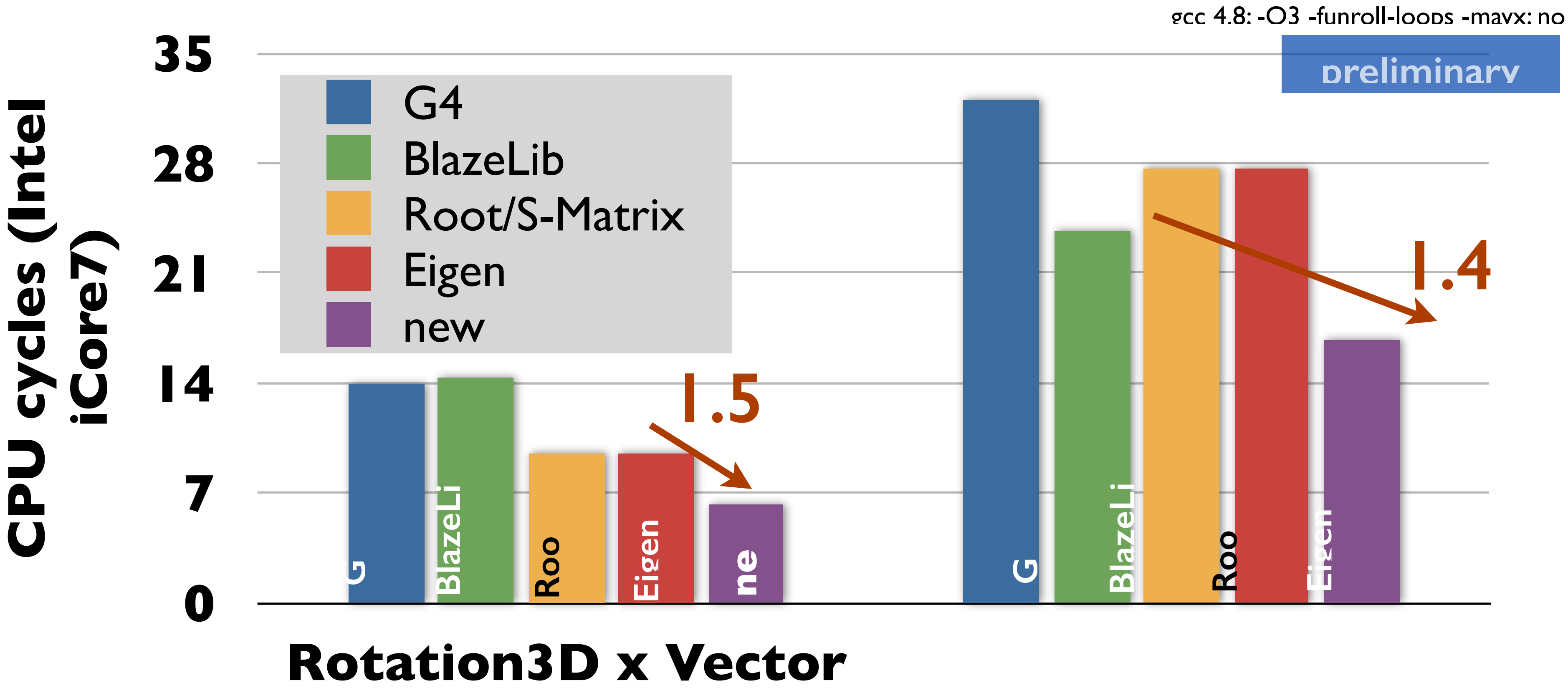
- Operations in SMatrix using `Vc::double_v` instead of `double`
  - speed-up obtained for processing operations on a list of 128 `SMatrix<double,5,5>` and `SVector<double,5>`



Specialized Vector classes

- **New vector classes for internal vectorization**
  - 3D Vector classes and their transformations developed as part of Geant4 Vector prototype
  - support for internal vectorisation in
    - vector-vector operations (additions)
    - vector-matrix transformation (rotations)
    - matrix-matrix transformation (rotation combinations)
  - use Vc for representing internal data
    - use `Vc::memory<double_v, 3>`
    - padding the unused 4-th element of the vector

- test performances on AVX



with Georgios Bitzes (CERN Openlab), Raman Sehgal

Vectorization in statistical calculations

Vectorization in Fitting

- **Vectorize chi-square calculation in fitting ROOT histograms**
  - *work performed by M. Borinsky (CERN summer student)*

$$\chi^2 = \sum_i \frac{(y_i - f_{a,b,\ldots}(x_i))^2}{\sigma_i^2}$$

- Required change in data set layout and in functions
  - from array of structure to structure of arrays for input data
  - vectorized function interface (TF1)

```
double func( double x, double* p )
{
    return exp( - p[0] * x );
}
```

Listing 1: Old callback function for TF1

```
void func ( double* x, double* p, double* val )
{
    for ( i in range )
        val[i] = exp( - p[0] * x[i] );
}
```

Listing 2: New vectorizable callback function for TF1

ROOT Fitting Tests

- Observed performance gain from
  - new data structure (organising fit data in a structure of arrays)
    - array for x values, array for y, array for z…
  - from auto-vectorization and using VDT library (for *log* and *exp*)

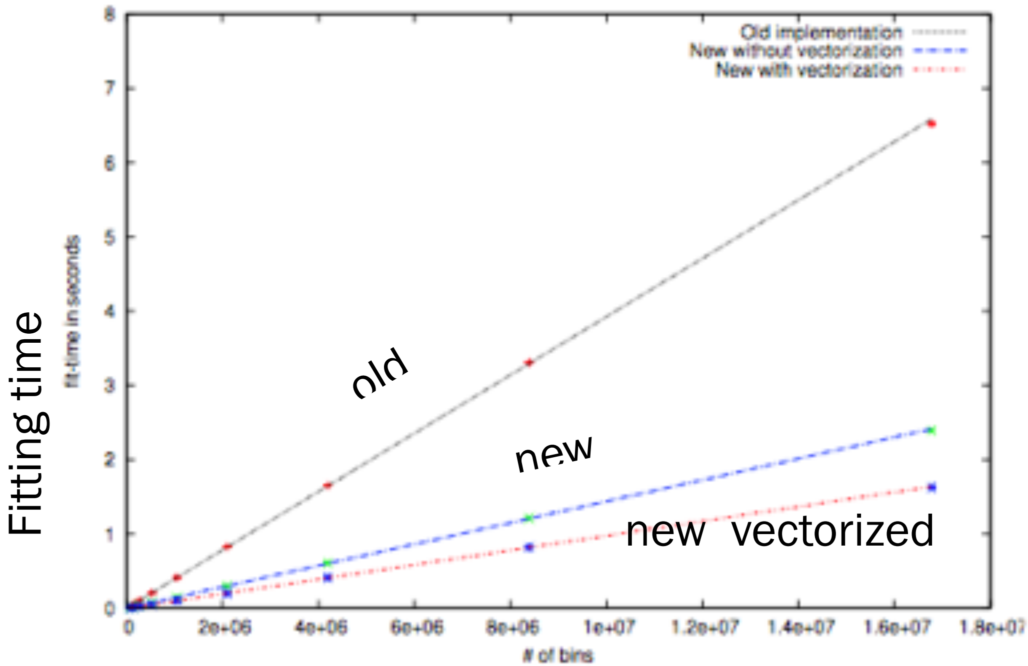


Figure: Performance with and without vectorization

Performance gains on AVX (E5-2690), gcc 4.7

old ⇒ new : 2.7x

new ⇒ vect: 1.5x

Total speed-up: 4.0x

- **Vectorisation in the ROOT fitting classes**
  - change internally interface for function evaluation
  - change fit data structure
    - have a template interface able to switch between scalar and vector data
    - use Vc for the vectors and Vdt for function evaluations
- Integrate vectorized vector and rotation classes based on Vc in the ROOT GenVector package
  - develop also classes for 4D (Physics vectors)
  - have a new type 3D Vector type using internally the new fast vector:
    - `DisplacementVector<double, Cartesian3DFast>`
    - `Rotation3DFast` class

Summary

- **ROOT provides several building block for vectored calculations**
  - vdt for mathematical functions
  - Vc library
  - physics (GenVector) and linear algebra (Smatrix) classes based on Vc
    - support for both external (already available in latest versions) and internal vectorisation (will be soon available)
  - vectorized function evaluations for fitting and statistical calculations

Backup Slides

Speed: VDT On ARM

Fnc.	Libm	VDT
Exp	*155*	71.4
Log	153	64.6
Sin	202	57.9
Cos	199	54.9
Tan	290	96.4
Asin	99.2	77.9
Acos	95.4	78.9
Atan	127	75.4
Atan	187	89.7
Isqrt	24.7	52.0

Double Precision

Time in **ns** per value calculated

- ARM Cortex A9, arm-v7 Odroid
- **VDT: Portable and very convenient**
- **Simple implementation pays on a simple architecture!**

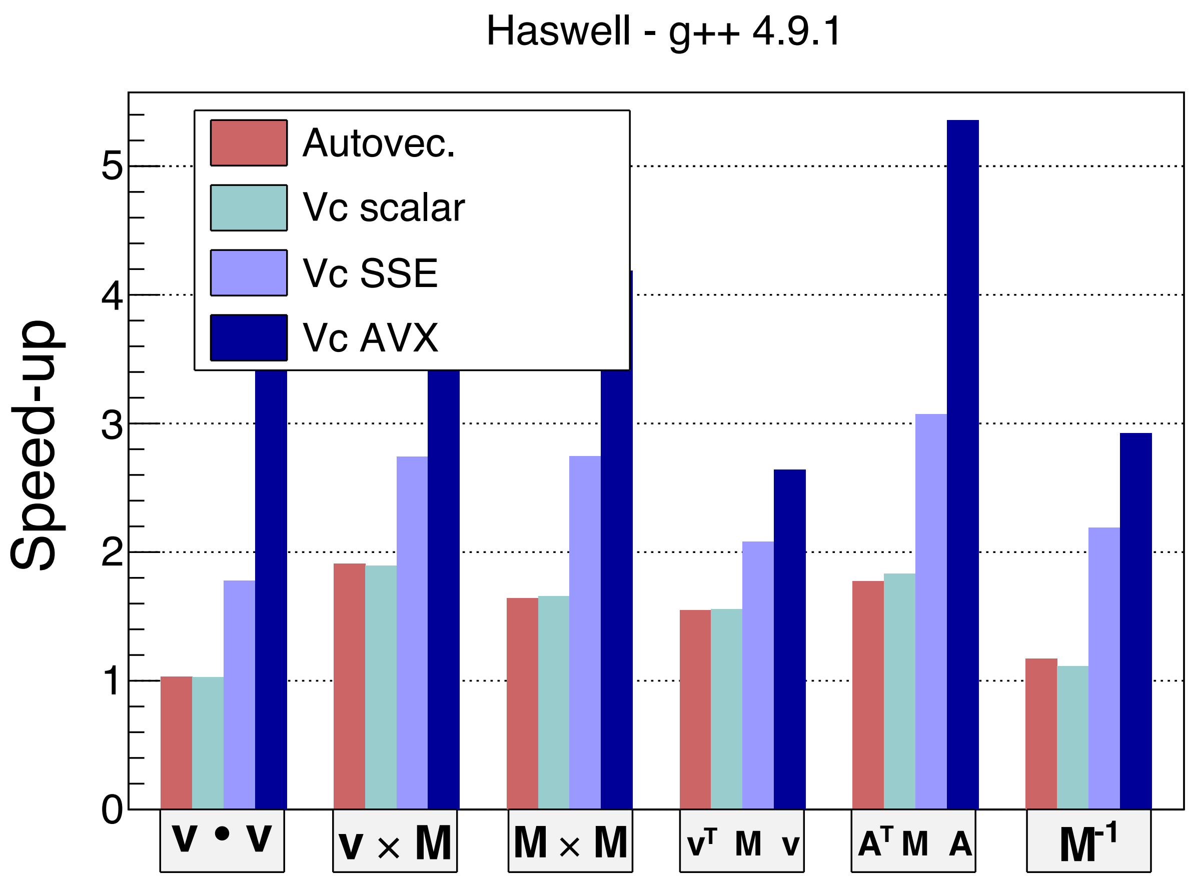
Accuracy: An Example

- 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 (sign, exponent, mantissa)!

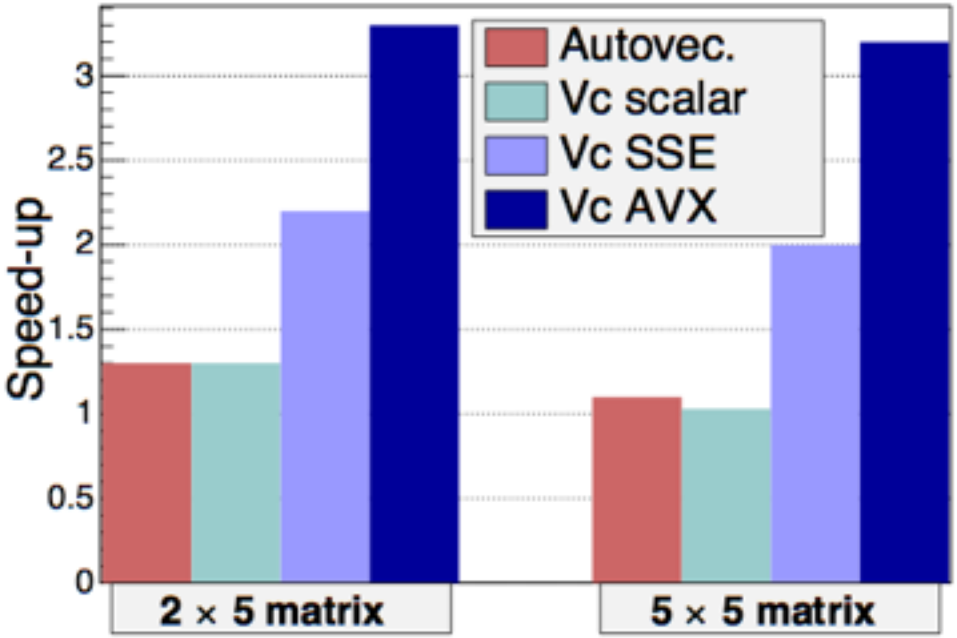
Double Precision	MAX VDT	AVG VDT
Exp	2	0.14
Log	2	0.42
Sin	2	0.25
Cos	2	0.25
Tan	2	0.35
Asin	2	0.32
Acos	8	0.39
Atan	1	0.33
Atan2	2	0.27
Isqrt	2	0.45

Only slight difference present: already enough for many applications

- Operations in SMatrix using `Vc::double_v` instead of `double`
  - speed-up obtained for processing operations on a list of 128 `SMatrix<double,5,5>` and `SVector<double,5>`



- Typical operation in track reconstruction
  - very time consuming
    - inversion + several matrix-vector multiplications



Clear advantage with Vc

SMatrix code can works using double_v as value_type

good boost in performance in an already performant code (5-10 times faster than CLHEP)