Parallelization Potential in ROOT Math Libraries

Lorenzo Moneta (CERN)

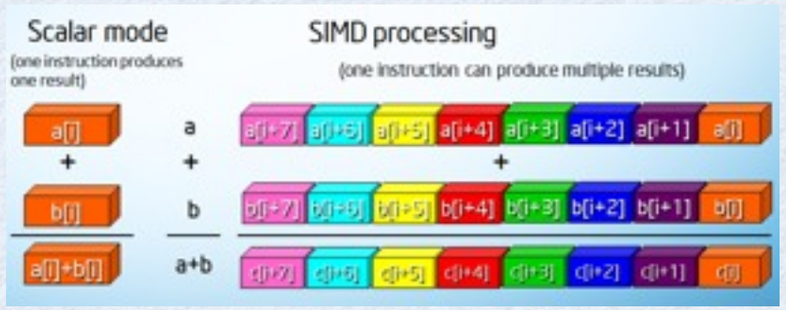
Annual Concurrency Forum Meeting, 4-6 February 2013, Fermilab

Outline

- Vectorization
  - Vc library
  - Potential for vectorization of core libraries using Vc
- Parallelization in data analysis (fitting)
  - low level with vectorization (new data structures)
  - multi-threads
  - experience gain from prototype studies
- Future plans for concurrent and parallel libraries
- Conclusions

Vectorization

- SIMD processing for performing many operation in parallel
  - size of registers depending on architectures
    - SSE : 128 bits : 2 double's or 4 float's
    - AVX: 256bit : 4 double's or 8 float's)



- Input data must be organized in vectors to perform operations simultaneously
- Automatically compiler can auto-vectorize loops
  - require no iteration dependency and no branching
- Alternatively use of intrinsic (but code will look like assembly)
- Can be exploited in low level libraries
  - no racing problems as in multi-thread
  - independent parallelization dimension
- Size of registers expected to increase in future hardware

Auto-Vectorization

- Compiler can vectorize the loops automatically, but
  - require data to be organized in vectors
  - no iteration dependence
  - no branching (if statement).
  - Much better in latest compiler versions
  - enabled with -O3 or with -O2 -ftree-vectorize
- Disadvantages:
  - need to change way passing data to functions
  - `double f(double x) ⇒ void f(const double *x, double *r )`
  - need to manage lists of results
  - can result in substantial code changes

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
- Provides vector classes (**Vc::float_v**, **Vc::double_v**) with semantics as built_in types
  - one can use **double_v** as a **double**
  - all basic operations between doubles are supported (**+,-,/,***)
  - provides also replacement for math functions (**sqrt, pow, exp, log, sin**,...)
- Possible to exploit vectorization without using intrinsic and with minimal code changes
  - e.g. replace double -> double_v in functions
  - easy to do in classes or functions templated on the value type
    - e.g ROOT classes in GenVector (3D or Lorentz vectors) or in SMatrix

Evaluation of Vc

- Use Vc to vectorize operation on a list of object (physics vectors, matrices, ...) and not within the object
- a `LorenzVector<PxPyPzE4D< double> >` becomes a `LorenzVector<PxPyPzE4D< Vc::double_v> >`,
- perform loop on list of objects (vectors, matrices), which is reduced by size of double_v (**`NITER = NITER / double_v::Size`**)
  - do not attempt parallelization within objects
- Tested on some basic operation between LorenzVectors:
  - Addition of vectors, scaling, invariant mass
  - Test using different compilation flags and Vc implementations (**`VC_IMPL = Scalar, SSE, AVX`**)
  - Compare results with auto-vectorization
    - compiling (using double) with **`-mavx -O3 -fast-math`**
    - reference is code compiled with **`-O2`**
    - use **`gcc 4.7.2`**

Vector Operations

- Speed-up versus scalar version (-O2)

Speed-up	auto-vectorization -Ofast -mavx	Vc scalar (auto-vec.)	Vc SSE	Vc AVX
Addition v3 = v1+v2	**5.0**	5	2.0	**3.3**
Scaling V2 = v1 * a	**6.6**	7.0	2.0	**3.6**
Inv. Mass M(v1,v2)	**1.3**	1.3	1.8	**2.0**
Boost	**1.02**	1.02	2.0	**2.1**

- Auto-vectorization works well simple operations, but not for more complex ones (e.g. when one needs to call Math functions)
- Vector lists must be not too large to avoid cache effects (used N=100)

Mathematical Functions

- Compare performances in evaluating Math Functions
- Use also VDT Mathematical library (*by D. Piparo*)
  - transcendental mathematical functions which can be auto-vectorized
  - but require a different interface: `std::sin(double x)`
    ⇒ `void vdt::fast_sinv(int n,const double *x, double *r)`

Speed-up	auto-vect. -Ofast -mavx	Vc scalar (auto-vec.)	Vc SSE	Vc AVX	VDT AVX
sqrt (x)	**2.4**	2.4	2.3	**2.4**	**2.4**
exp(x)	**1.0**	1.0	2.1	**4.9**	**4.1**
log(x)	**1.0**	1.0	3.8	**4.9**	**5.4**
sin(x)	**1.0**	1.0	0.4	**1.2**	**1.6**
atan(x)	**1.0**	1.0	1.5	**1.3**	**1.6**

SMatrix Operations

- Perform operations in SMatrix using `Vc::double_v` instead of `double`
  - speed-up obtained for processing operations on a list of 100 `SMatrix<double,5,5>` and `SVector<double,5>`

Speed-up	auto-vect. -Ofast -mavx	Vc scalar (auto-vec.)	Vc SSE	Vc AVX
v * v	**1.05**	0.8	1.2	**1.6**
v * M	**1.2**	0.7	1.5	**1.6**
M * M	**1.1**	0.6	1.1	**1.5**
vt * M *v	**1.0**	0.8	1.5	**2.1**
At * M * A	**1.1**	0.9	2.0	**2.3**
Inversion	**1.0**	0.9	1.7	**2.8**

Kalman Filter Test

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

Speed-up	auto-vect. -Ofast -mavx	Vc scalar (auto-vec.)	Vc SSE	Vc AVX
2 x 5 matrix	**1.3**	1.3	2.2	**3.3**
5 x 5 matrix	**1.1**	1.03	2.0	**3.2**

- Clear advantage in using Vc
  - SMatrix code works fine using `double_v` as value_type
  - good boost in performance in an already performant code (5-10 times faster than CLHEP)

Vectorization in Fitting

- Vectorize chi-square calculation in fitting ROOT histograms
  - *work performed by M. Borinsky (summer student 2012)*
- Required change in data set layout and in functions
  - from array of structure to structure of arrays for input data
  - vectorized function interface (TF1)

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

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

Listing 1: Old callback function for TF1

```
void func ( double* x, double* p, double* val )
2 {
      for ( i in range )
4         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 and from vectorization using VDT library

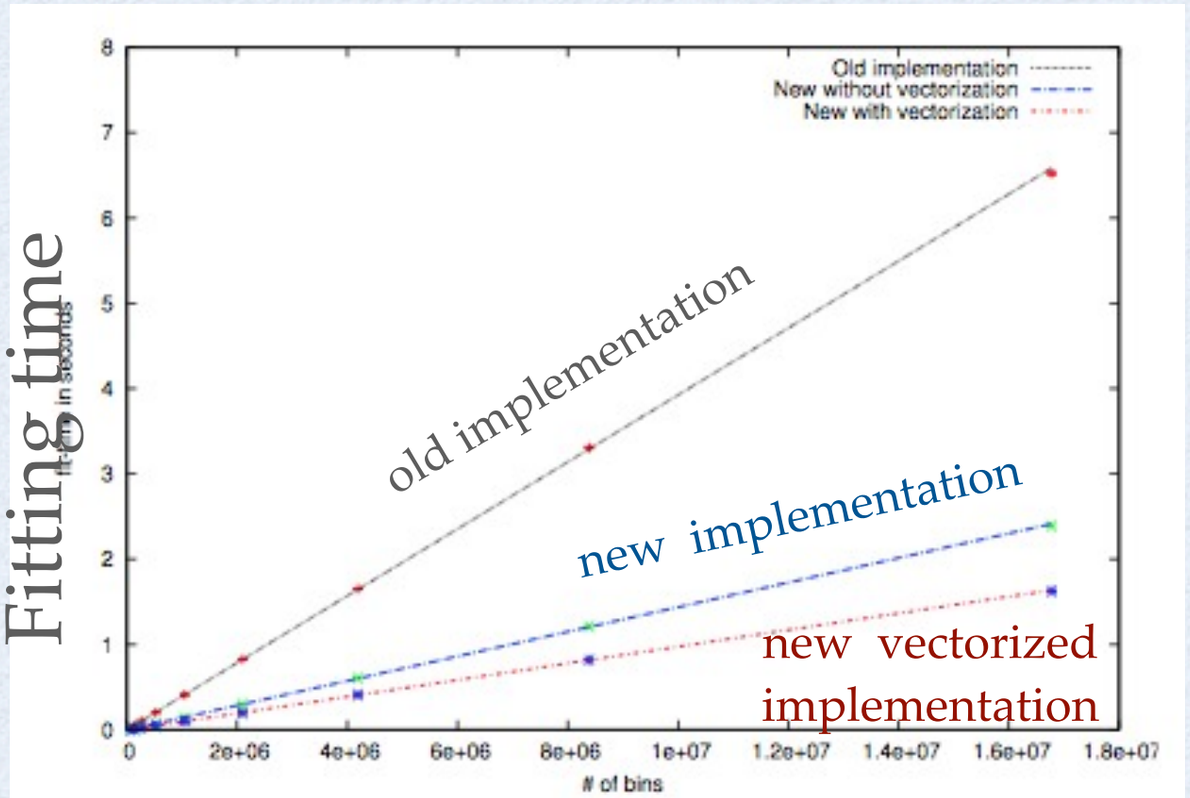


Figure: Performance with and without vectorization

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

old ⇒ new : 2.7x

new ⇒ vec: 1.5x

Total speed-up: 4.0x

- Test also using Vc: similar speed-up results (~ 3.5x) but with less code changes (would be easy if fit functions are templated)

Parallelization of Fitting

- Use also multi-threads (OpenMP) for parallelizing the chi2 sum

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

Test using Gaussian model function
(1 dimensional data set)

- Simple model: $T = t_{\text{serial}} + \frac{t_{\text{parallel}}}{\#\text{threads}}$

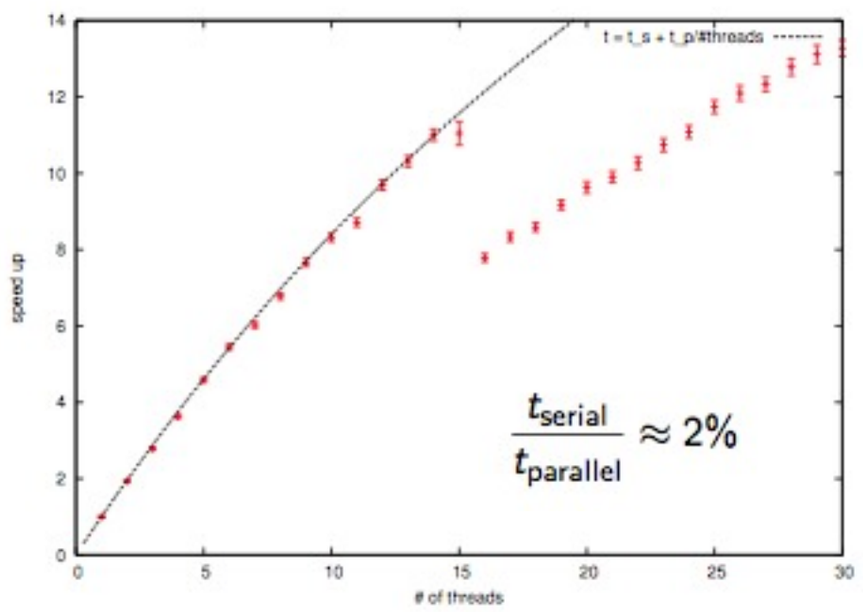
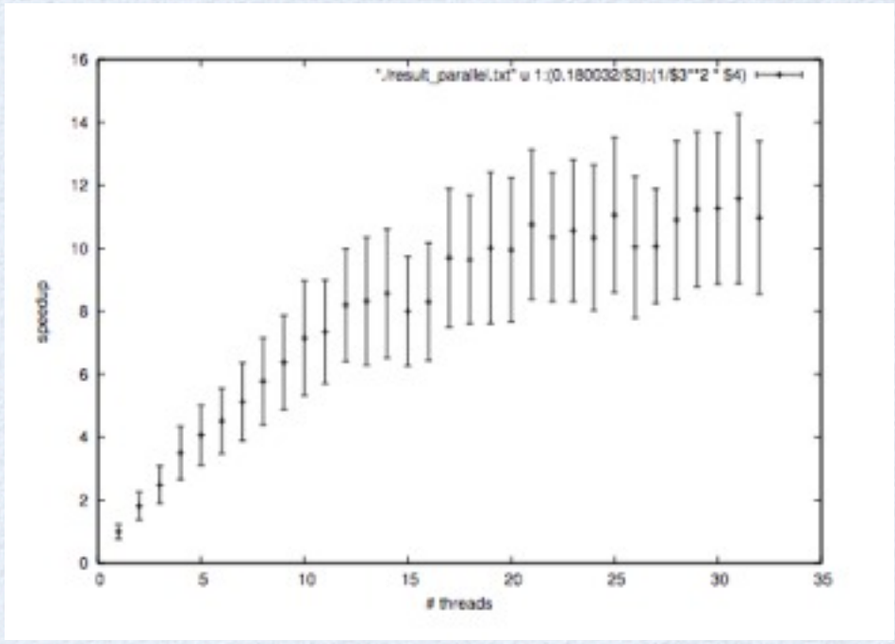


Figure: Performance with different thread numbers and $6 \cdot 10^4$ bins

M. N. Borinsky Parallelization of Fitting in ROOT

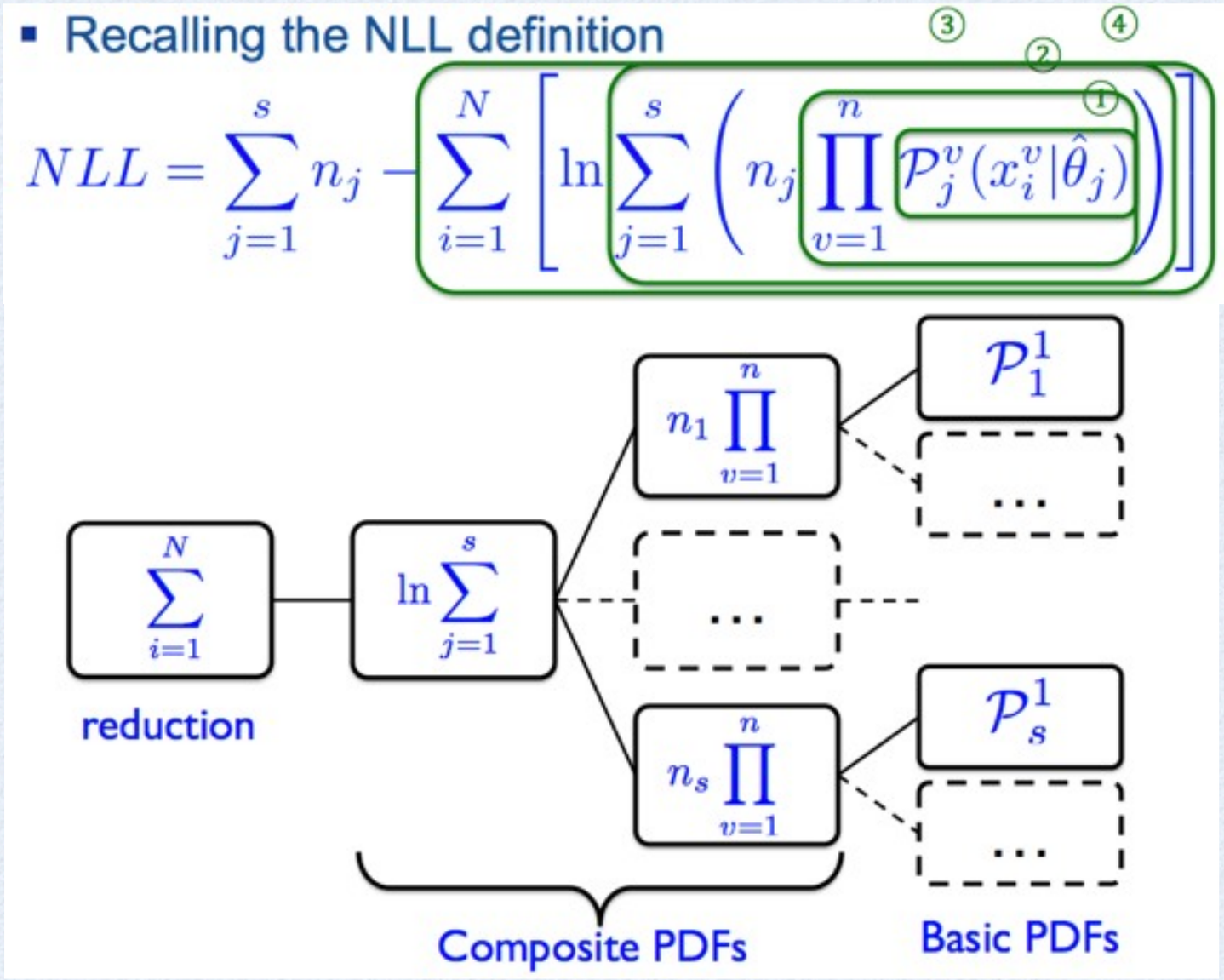
Using dynamic scheduling in OpenMP



Only some minimal code changes to be thread-safe

OpenLab Fitting Prototype

- Parallelize likelihood evaluation using RooFit *(by A. Lazzaro)*



Current RooFit code:

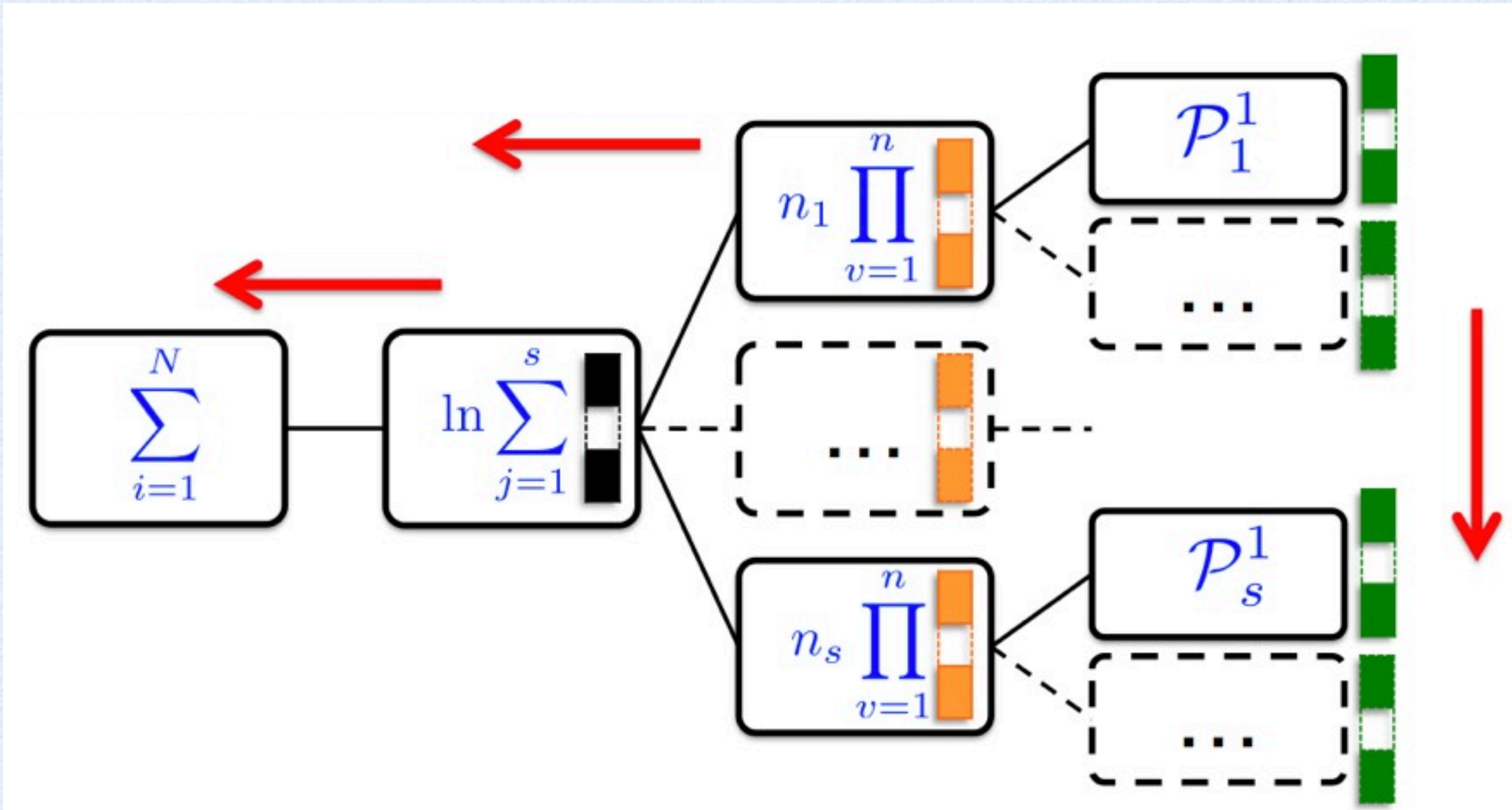
loop over N events and evaluate the full model for every event and perform sum at the end.

Parallelize reduction requires thread-safety of code: need to copy everything except the data.

- use a realistic data analysis model from B physics
- evaluate different architectures in collaboration with Intel

OpenLab Fitting Prototype

- Evaluate low-level PDF for an array of events
- Produce array of results:
  - vectorization + parallelization with OpenMP

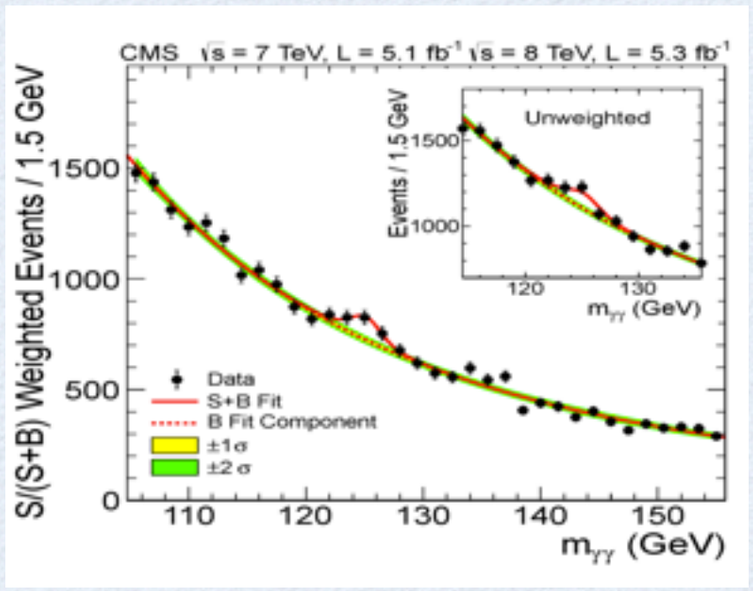


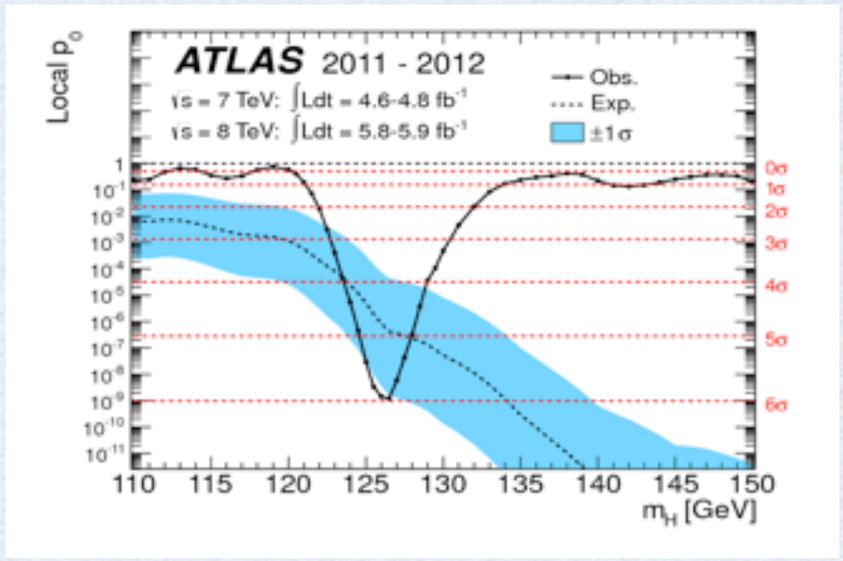
Results from OpenLab Prototype

- Low Level Parallelization
  - speed-up from vectorization (~ 1.7 on SSE)
  - speed-up from multi-threads:
    - 7.6x for 12 threads and 8.9x with 24 threads
    - overhead from having several OpenMP regions
    - cache memory effects in dealing with arrays
- High-Level parallelization with block-splitting (evaluation only on a sub-group of events) for vectorization
  - OpenMP only at root of NLL
  - Speed-up: 10.9x with 12 threads
- Race condition problems solved by several changes in original code, which are difficult to port in production code

Parallelization in Fitting

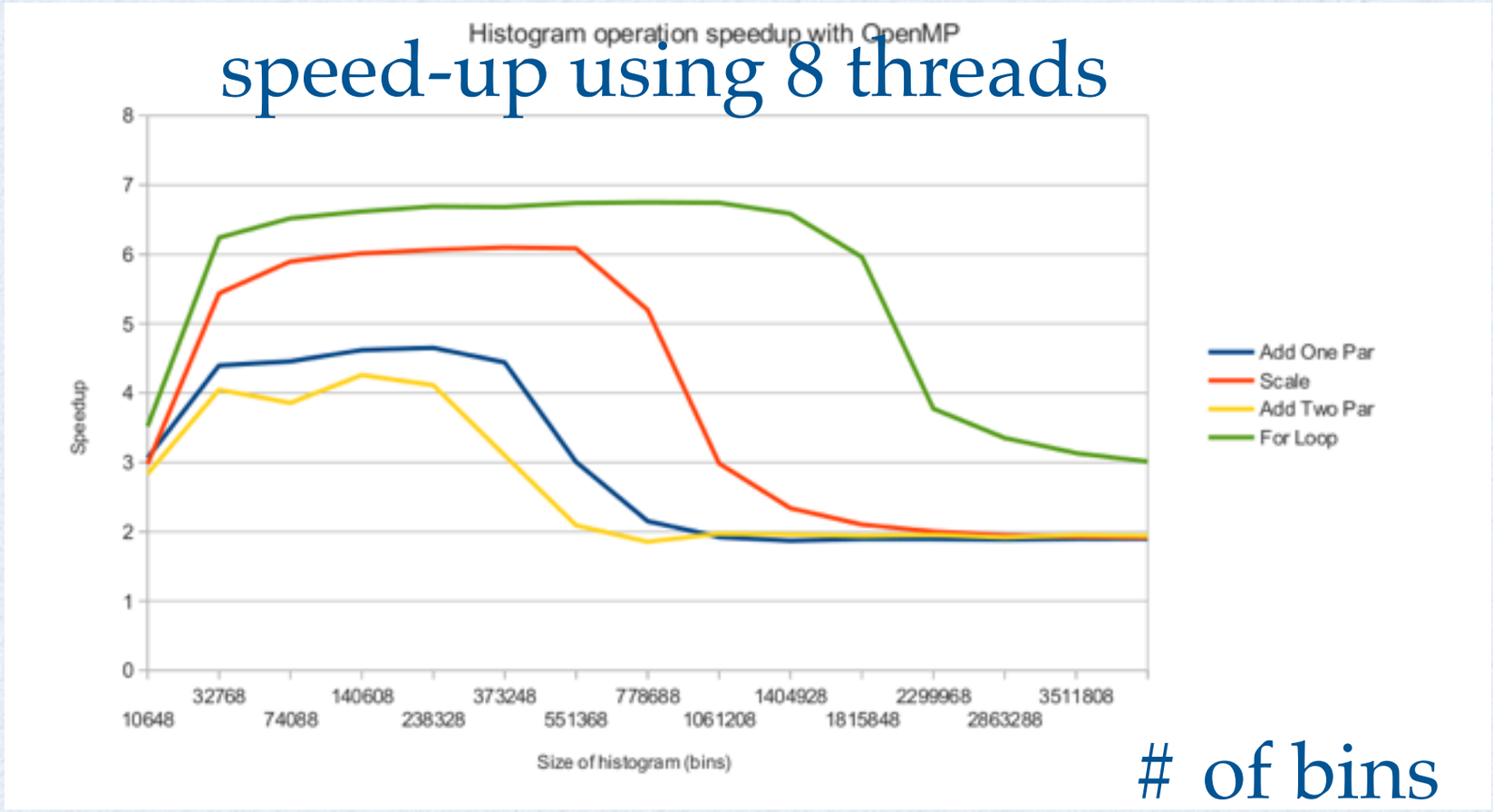
- Fitting/Minimization is one of the most CPU consuming tasks in data analysis
  - to produce Higgs combination results, many minimizations of complex likelihood functions (> 200 parameters) need to be done
  - currently run many jobs in parallel on the grid
  - memory usage could become problematic with increase complexity of the models (more parameters, more data, etc..)
  - need more efficient evaluation of and use low-level parallelization





Parallelization in Histogram

- Speed-up operation on ROOT histograms like scaling or merging (add) using multi-threads (openMP)



- Improve also performances by using more efficient serial code in several other histogram functions *(work by I.G. Bucur)*

Future Plans

- Distributed Vc library as part of ROOT
  - already existing in a svn branch
  - support using Vc in SMatrix and GenVector packages
  - investigate if could be used in other libraries (e.g. TMatrix)
- Support also auto-vectorization in low-level libraries
  - include also VDT for vectorized Math functions
- Change data structure and function interface for vectorization in fitting
- Support concurrency in RooFit will be not easy
  - investigate alternatives way of implementing complex models for fitting by keeping same user interface
- Investigate parallelization also in other high level libraries (e.g. TMVA) and algorithms (e.g. numerical integration)
- Provide parallel random number generators

Conclusions

- Big potential of using vectorization in speeding-up Math libraries
  - performance gains expected for simulation, reconstruction and data analysis
- Plan to provide thread-safe concurrent implementations of high level tools and algorithms
  - occasion also to improve code and make it more efficient
  - less virtual functions and using more templates and meta-programming
- Would be nice to start using C++11 for implementing the new code
  - need experiments to move to it as well

References

- Vc
  - http://code.compeng.uni-frankfurt.de/projects/vc/
  - M. Kretz, V. Lindenstruth, Vc, a C++ Library for explicit vectorization
  - M. Kretz, Efficient Use of Multi- and Many-Core Systems with Vectorization and Multithreading, Diplomarbeit (2009)
- VDT
  - see https://svnweb.cern.ch/trac/vdt
- ROOT fitting studies (M. Borinsky)
  - See summer student report: http://seal.web.cern.ch/seal/documents/mathlib/MichaelBorinsky_report.pdf
- OpenLab parallelization studies:
  - See Forum presentation by A. Lazzaro and various reports ,latest ones:
    - S. Jarp et al., Parallel Likelihood Function Evaluation on Heterogeneous Many-core Systems, proceeding of International Conference on Parallel Computing, Ghent, Belgium, 2011. EPRINT: CERN-IT-2011-012
    - S. Jarp et al., Parallel Likelihood fits with OpenMP and CUDA, Journal of Physics: Conference Series EPRINT: CERN-IT-2011-009

Extra Slides

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
- ✓ $1/\sqrt{}$ (different precision levels)

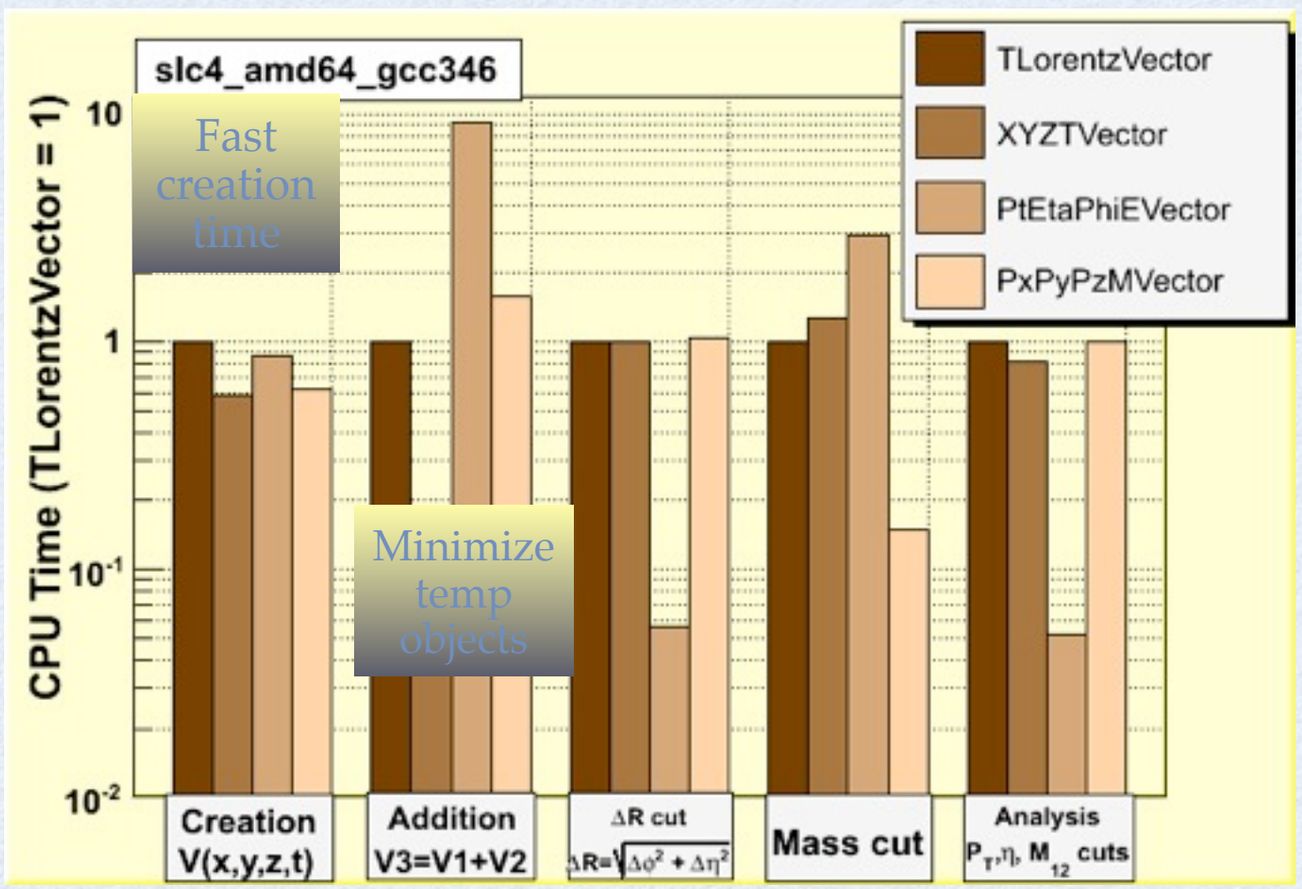
Single and Double precision implementations are different.

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)

ROOT Vector Classes

- New ROOT physics vector classes
  - GenVector package (see http://project-mathlibs.web.cern.ch/project-mathlibs/sw/html/Vector.html)
  - generic coordinate system concept
  - vector template class on coordinate system type in 2,3 and 4D

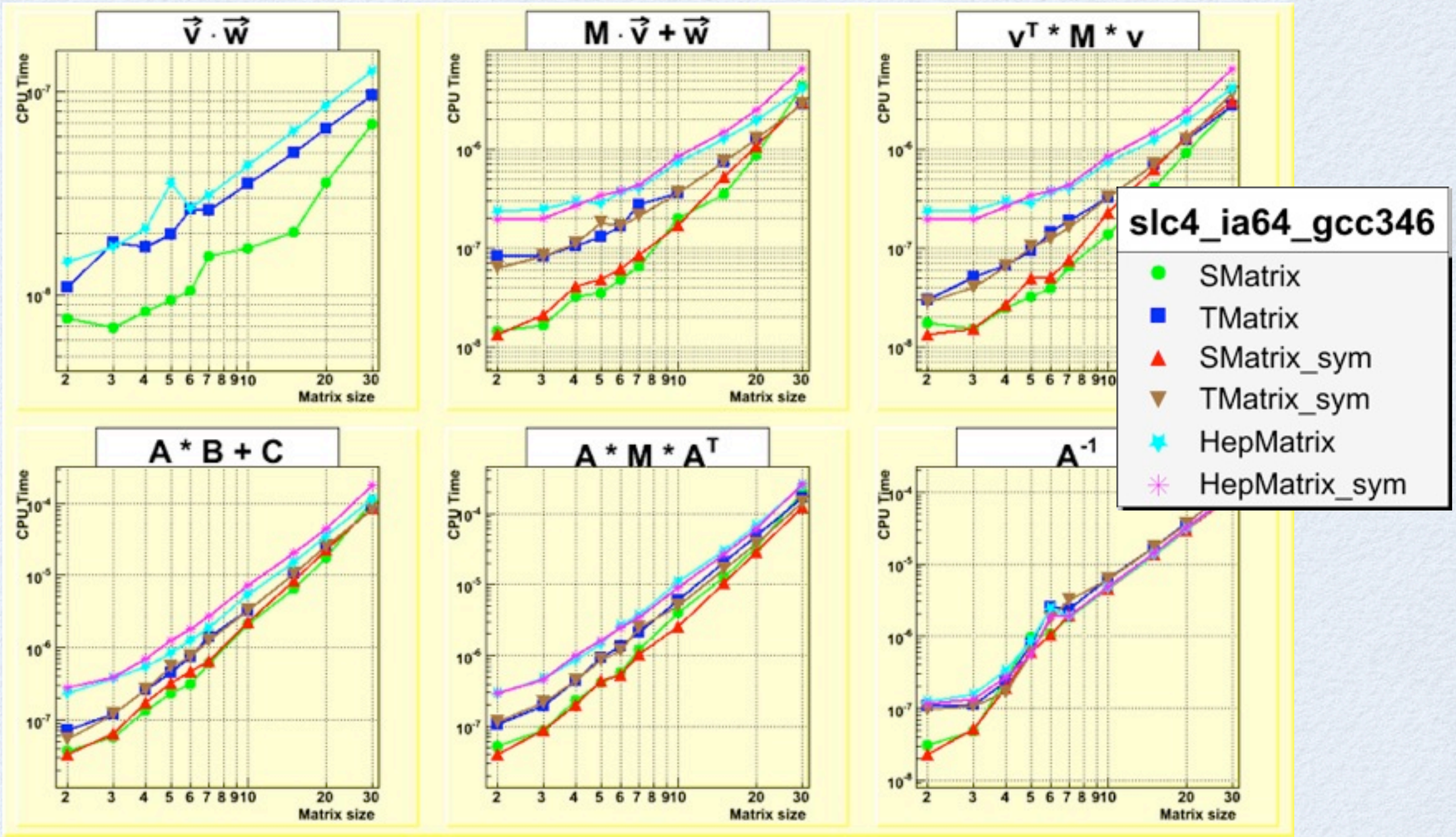


- `LorentzVector<PxPyPzE<double> >  v1;`
- `LorentzVector<PtEtaPhiE<double> >  v2;`
- `LorentzVector<PxPyPzM<double> >  v3;`
- `LorentzVector<PtEtaPhiM<double> >  v4;`

Advantage in performances using generic physics vector compared to TLorentzVector

Matrix Operations

- Old Comparison tests ROOT (*TMatrix*/*SMatrix*) and CLHEP (*HepMatrix*)
  - lxplus (Intel dual-core 64 bits) running slc4 with gcc 3.4.6



Fitting Tests Using Vc

- Performed a similar fit as before:
  - using Vc we do no really need big changes in data-structure
    - gain ~ 3.5 on AVX (1.8 on SSE)
      - probably due to faster exp and log functions
  - with VDT and vectorized data structure
    - gain ~ 4 on AVX
    - but reduced to 2 in case of large data sizes
      - probably due to cache effects given by managing large arrays of results
      - similar effect seen in OpenLab implementation

Parallelization: limitations

- Accessing the arrays of results: overlap computation and memory accesses
  - The amount of arrays to manage becomes consistent in case of complex models and large data samples
    - Crucial to have an optimal treatment of the data inside the cache memories
  - Effect particularly important for PDFs with simple function, like polynomials, and for the normalization loop (i.e. a product) and composite PDFs
    - Composite PDFs have to combine several arrays of results with just a simple operation (i.e. products and sums)
    - Fast computation, not enough time to fetch the data from memory

/Function /Call Stack	CPU Time
__svml_exp2.N	39.1%
PdfPolynomial::evaluateOpenMP	11.5%
PdfArgusBG::evaluateOpenMP	8.2%
PdfGaussian::evaluateOpenMP	6.8%
PdfAdd::evaluateOpenMP	6.5%
[libiomp5.so]	6.1%
PdfProd::evaluateOpenMP	5.4%
AbsPdf::GetVal	4.3%
NLL::GetVal	3.6%
PdfBifurGaussian::evaluateOpenMP	2.9%

(a) $N = 100\ 000$

/Function /Call Stack	CPU Time
PdfProd::evaluateOpenMP	22.8%
__svml_exp2.N	18.5%
PdfAdd::evaluateOpenMP	17.8%
PdfPolynomial::evaluateOpenMP	11.8%
PdfGaussian::evaluateOpenMP	6.6%
AbsPdf::GetVal	6.6%
PdfBifurGaussian::evaluateOpenMP	4.7%
PdfArgusBG::evaluateOpenMP	4.4%
[libiomp5.so]	2.1%
__svml_log2.L	1.7%

(b) $N = 1\ 000\ 000$

24 SMT threads

```
// Inline method for the Gaussian PDF calculation,
// defined inside the class RooGaussian
inline double evaluateLocal(const double x,
                const double mu,
                const double sigma) const
{
  return std::exp(-0.5*std::pow((x-mu)/sigma,2));
}

// Virtual method for the calculation of the
// Gaussian PDF on a single event
// (this is the original RooFit algorithm)
virtual double evaluate() const
{
  return evaluateLocal(x,mu,sigma);
}


// Virtual method for the calculation of the
// Gaussian PDF on all events
// (new implemented algorithm)
virtual bool evaluate(const RooAbsData& data)
{
  // retrive the data array of values for the variable
  const double *dataArray = data.GetDataArray(x.arg());
  // check if there is an array for the variable
  if (dataArray==0)
    return false;

  // retrive the number of events
  int nEvents = data.GetEntries();
  // retrive the array for the partial results
  double *resultsArray = GetResultsArray();
  double m_mu = mu;
  double m_sigma = sigma;

  // loop over the events to calculate the Gaussian
#pragma omp parallel for
  for (int idx = 0; idx<nEvents; ++idx) {
    resultsArray[idx] = evaluateLocal(dataArray[idx],
                      m_mu,m_sigma);
  }

  return true;
}
```

Implementation

- Take benefit from the code optimizations
  - No virtual functions
  - Inlining of the functions
- Evaluation of functions over arrays of read-only data
  - Balanced independent iterations
- Input data are shared in memory
  - Memory footprint increases with the number of events and number of PDFs, but not with the number of threads
- Possible to exploit vectorization
  - Using Intel compiler for the auto-vectorization of the loops (using svml library by Intel)
- Very easy parallelization with OpenMP
  - Easy thread-safety, limiting the parallelization to the PDF loops

NOTE: error checking inside the loops with output warnings will destroy vectorization and parallelization

Data Analysis Parallelization

- Possible various level of parallelizations:
  - Evaluation of probability density functions for observed data points
  - Loop on events for computing log-likelihood
  - Algorithms (e.g Minuit) require multiple likelihood evaluations
  - Loop on toy data analysis (on various likelihood minimization)
  - Repetition of same analysis on different inputs (analysis points)
- What is possible now:
  - Using PROOF for toy data analysis (RooFit/RooStats)
  - Parallelization of log-likelihood using multi-processes in RooFit (using PROOF or fork)
  - Parallelization in Minuit with MPI or multi-thread (OpenMP)
    - multi-thread implementation requires thread-safety of lower level code.
    - difficult to achieve in case of complex models (e.g. when built using RooFit)
    - good scalability only for large number of parameters (only gradient evaluation is parallelized)