Parallelization and Vectorization of ROOT Fitting classes

X. Valls, L. Moneta for the ROOT Team

Introduction

In order to take full advantage of new computer architectures and to satisfy the requirement of maximizing the CPU usage with increasing amount of data to analysis, parallelization and vectorization have been introduced in the ROOT mathematical and statistical libraries.

CERN Computing Needs Keep Growing (Images: CERN)

As part of this effort, new generic classes supporting a task based parallelization mode have been introduced in ROOT, which can be used for a wide range of computational tasks in the field of High Energy Physics. The support for different SIMD's libraries has also been included.

All these different tools for parallelism come together when parallelizing the fitting. As a result of this work, vectorization and parallelization have been introduced in ROOT requiring minimal changes in user code.

The Higgs fit is a good case example to report on the improvements obtained in the function evaluation, used for data modelling, by adding the support for SIMD vectorization and multithreaded parallelization. We will display for this use case how the different evaluations of the likelihood and the least square functions used for fitting ROOT histograms, graphs and trees perform.

Tools for parallelism

ROOT provides several generic classes for the expression of parallelism at different levels. Some of them play a central role when parallelizing ROOT fitting classes.

Task level parallelism: TThreadExecutor

TThreadExecutor is a task-oriented, multithreaded MapReduce for ROOT, provides a simple programming model for parallel MapReduce operations on decoupled-data based loops:

```
auto mapFunc = [](const UInt_t &i){
    return i+1;
};

auto reduceFunc = [](const std::vector<UInt_t> &mapV){
    return std::accumulate(mapV.begin(), mapV.end());
};

ROOT::TThreadExecutor pool;
pool.MapReduce( mapFunction, ROOT::TSeq<int>(100), reductionFunction);
```

Its interface includes operations like Map, Reduce, Foreach and even chunked mapping with partial reduction.

Right now is used among others in math fitting, TMVA(Boosted Decision Tree evaluation, Deep Neural Networks processing), Implicit multithreading operations in I/O (reading, deserialization and decompression of tree branches in parallel, parallel writing) or for parallel execution of functional chains in TDataFrame.

Instruction level parallelism: VecCore

VecCore is a library providing efficient vectorization on all platforms by offering an abstraction layer on top of the libraries Vc and UME::SIMD, which will provide as optional backends along a scalar one for the case when SIMD operations are not available.

VecCore allows writing abstract, architecture-generic code that will map to each of the backends concrete types, methods or instructions.

For more information on VecCore please see the poster ”Speeding up software with VecCore, a portable SIMD library” by Guilherme Amadio.

Masking for control flow in VecCore

```
namespace vecCore {

template <typename T> struct TypeTraits;
template <typename T> using Mask = typename TypeTraits<T>::MaskType;

// Masking/Blending
template <typename M> bool MaskFull(M const &mask);
template <typename M> bool MaskEmpty(M const &mask);

template <typename T> void MaskedAssign(T &dst, const Mask<T> &mask, const T &src);
template <typename T> T Blend(const Mask<T> &mask, const T &src1, const T &src2);

} // namespace vecCore
```

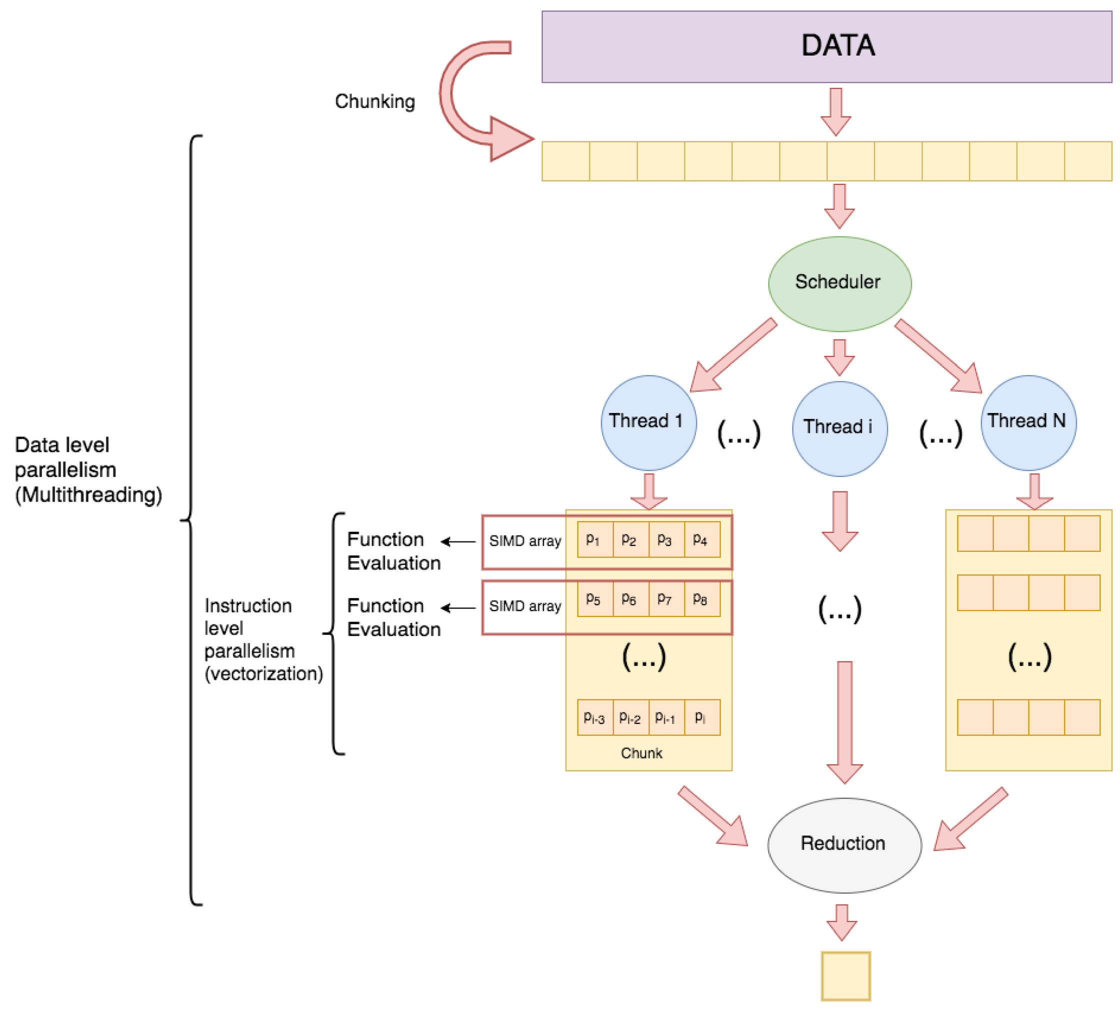
The parallelization tools described define the building blocks for parallelizing the fit. TThreadExecutor provides the MapReduce framework to chunk the evaluation and share the computational workload between the several threads of the system and VecCore is used for abstracting SIMD operations and types from the code making it portable between architectures with different sets of SIMD instructions.

Fitting

The figure below describes in detail the parallelization process of the fitting functions in both task level and instruction level.

Vectorization will be applied for evaluating the function several points at a time in the multithreaded Map stage, which will generate several partial results in each thread to combine in a final reduction step.

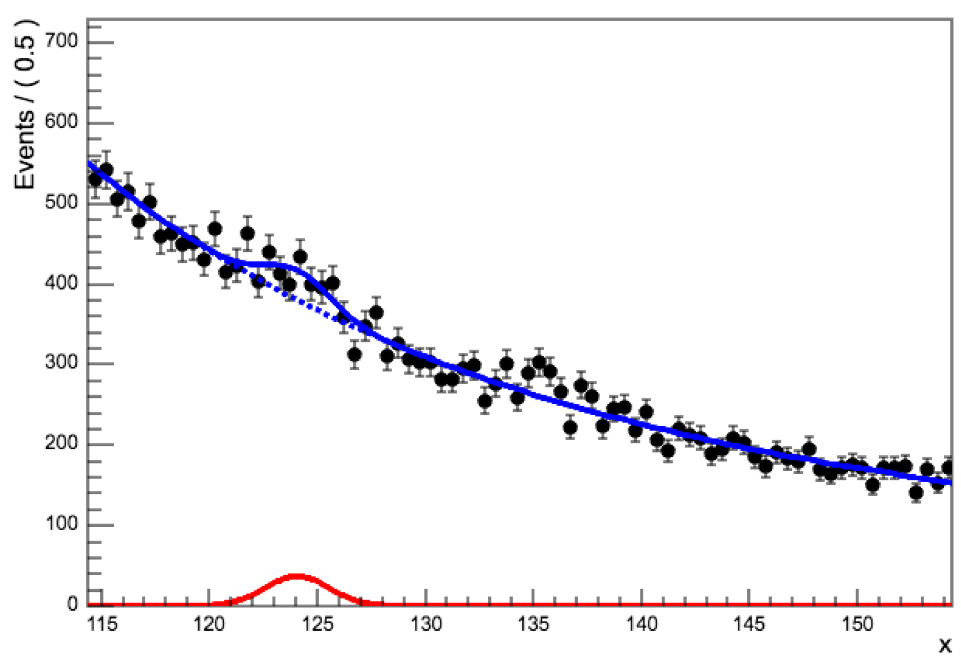
Fitting Parallelization



Case example: Higgs Fit

In order to report on the performance of the parallelization described we decided to apply it to the well known Higgs Fit.

Higgs Fit



The pieces of code below compare the current code used for fitting the case example with the code needed for a fully parallelized implementation of the same fit.

Current implementation

```
//Higgs Fit: Implementation of the scalar function
double func(const double *data, const double *params)
{
   return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
          params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
          ((*data) * (0.01)) * ((*data) * (0.01))));
}

TF1 *f = new TF1(”fScalar”, func, 100, 200, 4);
f->SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f(”h1f”, ”Test random numbers”, 12800, 100, 200);
h1f.FillRandom(”fvScalar”, 1000000);
h1f.Fit(f);
```

Vectorized plus parallelized implementation

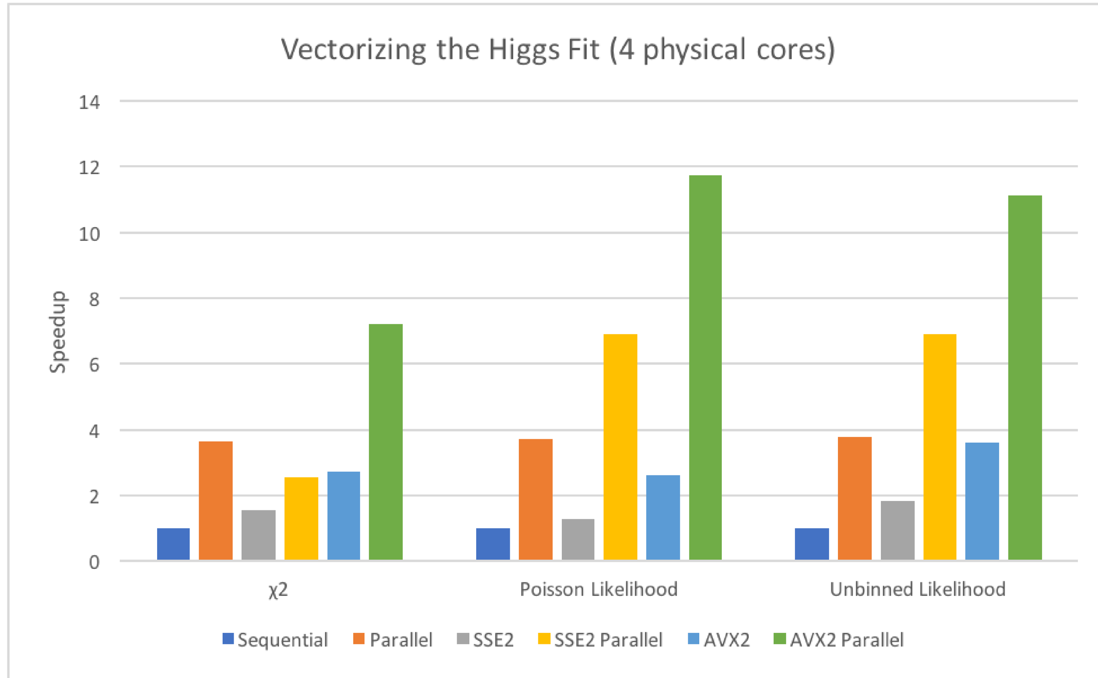
```
//Higgs Fit: Implementation of the vectorized function
ROOT::Double_v func(const ROOT::Double_v *data, const double *params)
{
   return params[0] * exp(-(*data + (-130.)) * (*data + (-130.)) / 2) +
          params[1] * exp(-(params[2] * (*data * (0.01)) - params[3] *
          ((*data) * (0.01)) * ((*data) * (0.01))));
}

//This code is totally backwards compatible
TF1 *f = new TF1(”fvCore”, func, 100, 200, 4);
f->SetParameters(1, 1000, 7.5, 1.5);
TH1D h1f(”h1f”, ”Test random numbers”, 12800, 100, 200);
h1f.FillRandom(”fvCore”, 1000000);

//Added multithreaded fit option
h1f.Fit(f, ”MULTITHREAD”);
```

The only changes required are make the fitting function vectorized (change the data parameter type and the return type in this case) and to specify the ”MULTITHREAD” option to the fitting. This makes most of the existing user code elegible to vectorize with very little effort.

Below we present some of the performance measurements made while woking on this case example. All the fitting times have been normalized to the number of function calls made by the fitter, as the nature of minimization problems will make the number of calls fluctuate between examples and influence the times.

Note that the compiler will try to autovectorize operations in the scalar case, and our current implementation for the function evaluation will provide help in that direction. This may make the vectorization times not look so good.

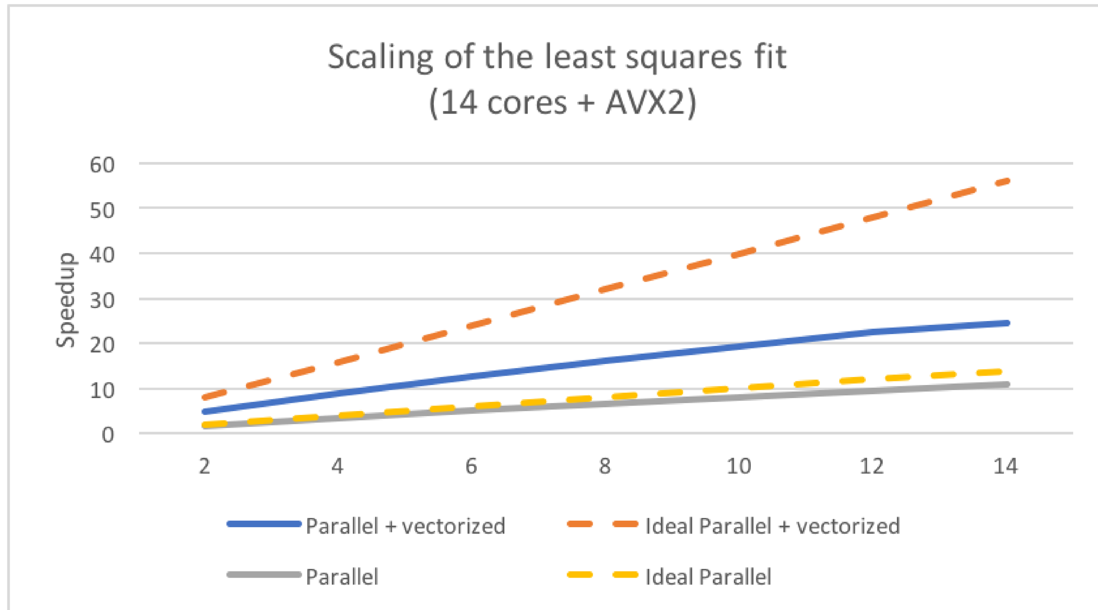
Performance of the Higgs Fit on your laptop



The figure above describes the speed up obtained fitting o a typical mass-consumption general purpose computer, a 4-core machine with different sets of vectorization instructions and different evaluation functions.

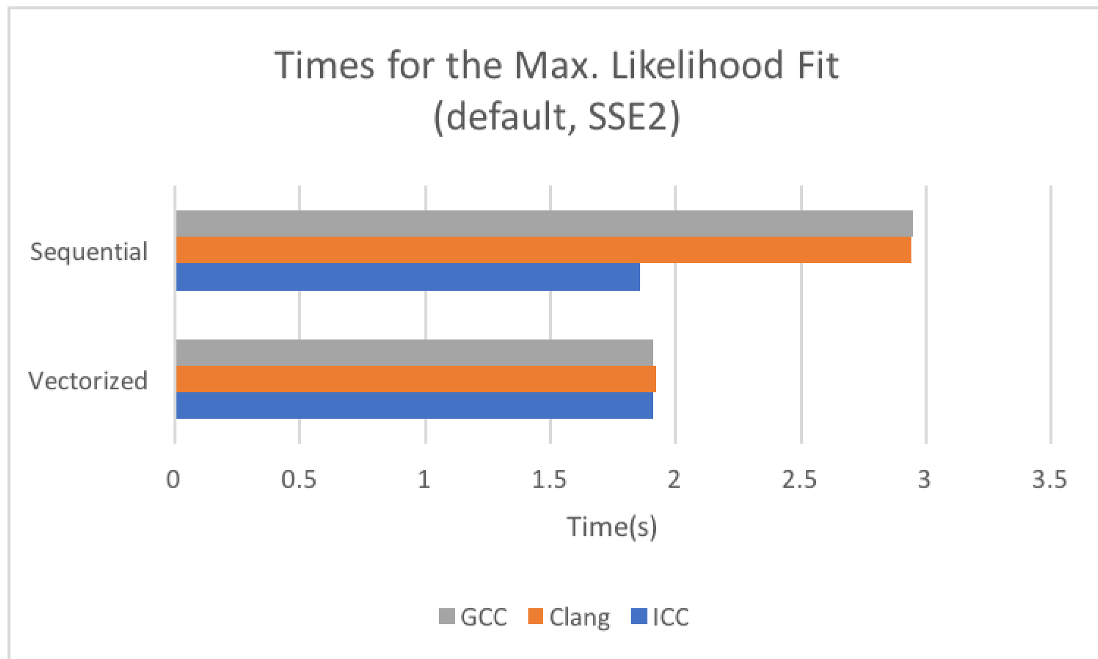
Below, we study the scaling of multithreaded operations on a 14 cores Haswell processor with SSE2 as SIMD instruction set. While it scales in the multithreaded scalar case, vectorization limits the speedup obtained on the multithreaded vectorized case.

Higgs Fit Scaling With Number of Cores



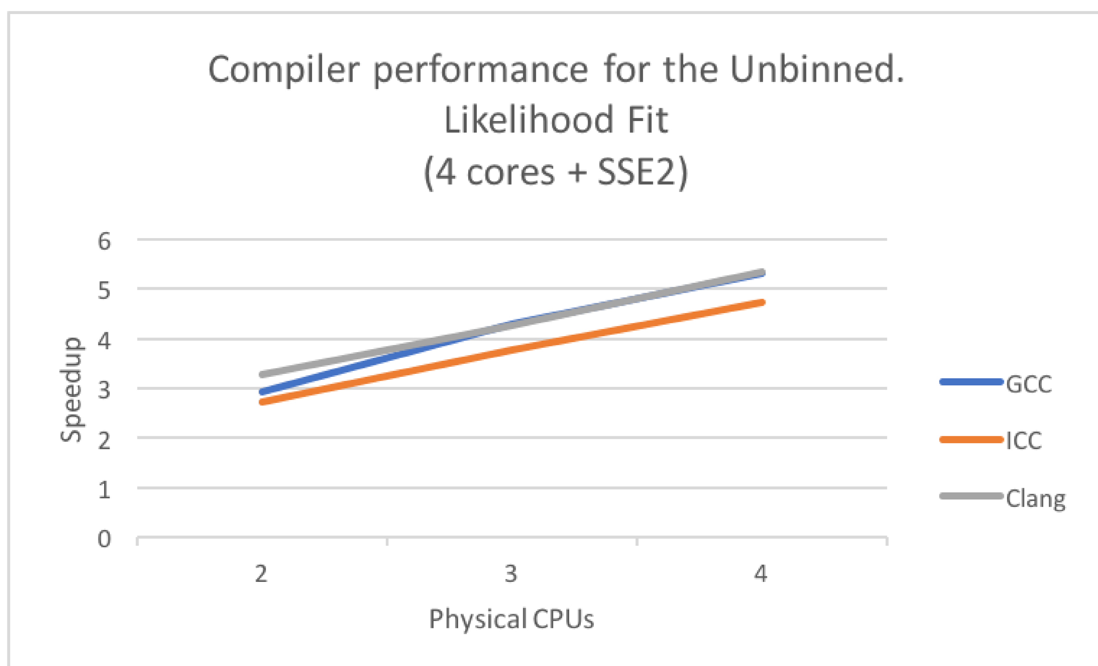
It is also interesting to show how different compilers behave when vectorizing:

Autovectorization in ICC



We find out that ICC autovectorizes more agressively than the other compilers, matching the explicitly vectorized times. For a more fair comparision, avoiding autovectorization when compiling, the results between compilers are close enough, with ICC slightly underperforming Clang and GCC:

Compiler performance



References

1. VecCore Library https://github.com/root-project/veccore
2. Vc https://github.com/VcDevel/Vc
3. UME::SIMD https://github.com/edanor/umesimd
4. ROOT Data Analysis Framework https://root.cern