Vectorization of ROOT Mathematical Libraries

G. Amadio, L. Moneta, X. Valls (CERN EP-SFT)
Outline

- Introduction
- VecCore library for vectorization
- Integration of VecCore in ROOT
- Vectorization in function evaluation (fitting), matrix and vector classes
- Future plans
- Conclusions
Introduction

- HEP software needs to fully exploit SIMD vectorisation and parallelisation to achieve the desired performances in simulation, reconstruction and data analysis.
VecCore Library

- Provide simple API to express SIMD algorithms
  - write directly SIMD code is challenging
- Can support different back-end implementation
  - \texttt{Vc} and \texttt{UME::SIMD}
  - users can choose the optimal one depending on the running architecture
  - New \texttt{Vc} version will be part of the C++ standard. With VecCore easy migration
- API covering essential parts of SIMD instructions
  - it allows to implement majority of numerical algorithms
  - e.g. masking operations for dealing with branches
**VecCore Performances**

- Study vectorisation performances in a mathematical algorithm
- Generation of Julia sets \( z_{n+1} = z_n^2 + 0.7885e^{i\alpha}, \quad -2 \leq z \leq 2, \quad \alpha \in [0, 2\pi), \quad n \leq 100. \)
- Speed-up is less than ideal due to branching
- different number of computations for each data points and varying as function of angle parameter

Generation time as function of angle when using scalar float types
VecCore Performances

- **Vc** seems to outperform the **UME::SIMD** implementation.
- **gcc** outperforms **clang** and **icc** (especially when using **Vc**).
- **Vc** does not provide an implementation working for AVX-512.
VecCore and ROOT

- **VecCore** is now integrated in ROOT together with the **Vc** back-end:
  - e.g. configure ROOT with
    ```
    cmake -Dbuiltin_veccore=On -Dbuiltin_vc=On
    ```
  - When VecCore is enabled (**R__HAS_VECCORE** is defined), ROOT provides these new VecCore SIMD vector types:
    - `ROOT::Float_v`
    - `ROOT::Double_v`
  - The SIMD vector sizes (`ROOT::Double_v::size()` will depend on the compiled instruction set:
    - `ROOT::Double_v::size()`=2 when code is compiled with SSE
    - `ROOT::Double_v::size()`=4 for AVX (e.g. on Haswell)
    - `ROOT::Double_v::size()`=8 for AVX-512 (e.g. on KNL)
VecCore and ROOT Math

- Vectorization of ROOT Math interfaces for function evaluations (used for fitting in ROOT)
- vectorize on the data $x$ which can be multi-dimensional

Add generic interfaces for evaluation:

```cpp
template<class T>
class IParametricFunctionMultiDimTempl: virtual public IBaseFunctionMultiDimTempl<T>,
                     virtual public IBaseParam {

  public:
    typedef T BackendType;
    ....
    // Evaluate the function at a point $x[]$ and parameters $p$
    T operator()(const T *x, const double   *p) const { return DoEvalPar(x,p); }

  private:
    virtual T DoEvalPar(const T *x, const double *p) const = 0;
    virtual T DoEval(const T *x) const;
};
```

- where $T$ can be instantiated as a `ROOT::Double_v` or just `double`
- Backward compatibility is preserved!
TF1 Extensions

- TF1 class has been extended to support vectorised user functions

- TF1("fs",[](double *x, double *p){ return p[0]*sin(p[1]*x[0]); }, 0., 10., 2);
- TF1("fv",[](ROOT::Double_v *x, double *p){ return p[0]*sin(p[1]*x[0]); }, 0., 10., 2);

- Template evaluation accepting VecCore SIMD vector types

- template <class T> TF1::EvalPar(const T * x, double * p) -> T;

- Vectorized TF1 function can then be used for fitting (e.g. in TH1::Fit)

very small overhead when evaluating using a TF1 instead of a direct free function
Vectorization of TFormula

- ROOT TFormula class is used to build parametric functions which can be used for fitting and modeling directly from string expression.

- e.g. `TFormula("f1","[a]*sin( [b]*x")`;

- expression is compiled using JIT provided by CLING.
  - compiled signature is based on
    - `f(double *x, double *p)->double`

- Added capability to JIT compile with a vectorised signature:
  - `f(ROOT::Double_v *x, double *p)->ROOT::Double_v`

- One can then easily have vectorised functions for fitting automatically.
Vectorized TFormula Performances

- Performance results evaluating a math expression using a free C++ function with TF1 and TF1 based on TFormula
- Study the speed-up by using vectorisation on AVX

CASE 1

CASE 2

Vectorization Performance on AVX2

- C++ Function
- TF1 Compiled
- TF1 Formula

1. 2nd degree polynomial
2. Exponential + gaussian
Fitting with Vectorized Functions

- Multi-dimensional input data $x$ (coming from histograms or TTree’s) is vectorized using `ROOT::Double_v`
- Organize data from AOS to SOA
  
  $$(x_0, y_0, z_0, \ldots, x_n, y_n, z_n) \rightarrow (x_0, \ldots, x_n, y_0, \ldots, y_n, z_0, \ldots, z_n)$$

- Model function is evaluated in vectorized mode when computing the chi-square or likelihood function (objective function) for fitting

- Computation of objective function is also parallelized with multi-threads by chunking the data
Fitting Performances

- Measure CPU performances in a typical HEP fitting
- Fit invariant mass spectrum to determine significance and location of the signal (e.g. $H \rightarrow \gamma \gamma$)

![Invariant mass distribution (\(\gamma \gamma\) events)](image)

Vectorization Performance on AVX2

<table>
<thead>
<tr>
<th></th>
<th>Vectorization</th>
<th>MultiThreads</th>
<th>Vect and MT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Least-Square</td>
<td>3.1</td>
<td>21</td>
<td>53</td>
</tr>
<tr>
<td>Binned</td>
<td>2.8</td>
<td>23</td>
<td>66</td>
</tr>
<tr>
<td>Unbinned</td>
<td>2.4</td>
<td>30</td>
<td>89</td>
</tr>
</tbody>
</table>

Intel Xeon CPU E5-2683 with 28 physical cores
ROOT provides a template vector and matrix classes (optimized for small sizes) which can be used in single and double precision
- `SVector< double,N>`
- `SMatrix< double,N,M>`

Template classes for geometry and physics vectors with their transformations
- `DisplacementVector3D< Cartesian3D<double>`
- `LorentzVector<PxPyPzE4D<double>`

`VecCore` types (`ROOT::Double_v`) can be used as template parameters for vector and matrices classes and for the geometry vectors
- vectorisation for operations on a list of vectors/matrices (vertical vectorisation)
Transparent Data Parallelism: VecOps

Introduced `ROOT: :RVec<T>`: vectorised operations made easy

- std::vector like interface, ergonomic support of analysis operations
- Can adopt memory or own it
- Vectorised arithmetic operations, math functions

```cpp
RVec<double> mus_pt {15., 12., 10.6, 2.3, 4., 3.};
RVec<double> mus_eta {1.2, -0.2, 4.2, -5.3, 0.4, -2.};
RVec<double> good_mus_pt = mus_pt[mus_pt > 10 && abs(mus_eta) < 2.1];

RVec<float> vals = {2.f, 5.5f, -2.f};
RVec<float> sin_vals = sin(vals);
```
Future Plans: Math Functions

- Re-implement mathematical functions in TMath and ROOT::Math (e.g. statistics functions) using VecCore.
- Plans is to have a single template implementation, which can work for scalar and vector types.
- Example:
  
  ```cpp
template <class T>
  TMath::Gaus(const T & x, double mu, double sigma) -> T
  ```

- Basic math functions (e.g. exp, log, sin, cos) are already provided by VecCore.
Future Plans: Random Number

- Can use vectorization to speed-up pseudo-random number generations
- Horizontal (internal) or vertical vectorization can be used
  - By creating a vector state (based on VecCore) we can speed-up generators like Ranlux or MixMax
- on-going effort in collaboration with GeantV project and MIXMAX network (https://indico.cern.ch/event/731433/)

![CPU performance for PRNGs available in ROOT](image-url)
Future Plans: Machine Learning

- Use **VecCore** for matrix operations in Neural Network
- Interested in optimise the single event evaluation.
- Vectorisation can be used for:
  - applying weight to input layer data (matrix multiplication)
  - compute activation function using vectorised implementations (e.g. tanh)

Performances for evaluating a deep neural network architecture
Conclusions

- Advantages by using VecCore library which provides a simpler programming model for SIMD
- Benchmark of VecCore and its backend shows that Vc outperforms UME::SIMD and gcc performs better than icc or clang
- ROOT uses internally VecCore by defining new vector types: ROOT::Float_v and ROOT::Double_v
- Extension of ROOT function classes and interfaces to support these new types
- Integrate vectorisation also in TFormula class, thanks to ROOT JIT’ing
- Significant performances improvement in ROOT thanks to vectorization
- Plan to deploy vectorization even more:
  - math functions, random numbers and deep learning
Thank you!

References:

- **ROOT**: https://root.cern.ch
- **VecCore**: https://github.com/root-project/veccore
- **Vc**: https://github.com/VcDevel/Vc
- **UME::SIMD**: https://github.com/edanor/umesimd
- **SIMD C++ standardization**: http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3759.html
SIMD Vectorization

- Writing efficiently SIMD code is challenging
- Libraries exist that wrap SIMD intrinsic in a convenient interface
  - Vc
  - UME::SIMD
- They do not support all architectures or performances very dependent on specific platforms
History of Intel SIMD

- Intel® Pentium Processor (1993)
  - 32bit

- Multimedia Extensions (MMX in 1997)
  - 64bit integer support only

- Streaming SIMD Extensions (SSE in 1999 to SSE4.2 in 2008)
  - 32bit/64bit integer and floating point, no masking

- Advanced Vector Extensions (AVX in 2011 and AVX2 in 2013)
  - Fused multiply-add (FMA), HW gather support (AVX2)

- Many Integrated Core Architecture (Xeon Phi™ Knights Corner in 2013)
  - HW gather/scatter, exponential

- AVX512 on Knights Landing, Skylake Xeon, and Core X-series (2016/2017)
  - Conflict detection instructions

□ = 32 bit word
Why Use SIMD?

SIMD vectorization is already essential for high performance on modern Intel® processors, and its relative importance is expected to increase, especially on hardware geared towards HPC, such as Xeon Phi™ and Skylake Xeon™ processors.
namespace vecCore {

    template <typename T> struct TypeTraits;
    template <typename T> using Mask = typename TypeTraits<T>::MaskType;
    template <typename T> using Index = typename TypeTraits<T>::IndexType;
    template <typename T> using Scalar = typename TypeTraits<T>::ScalarType;

    // Vector Size
    template <typename T> constexpr size_t VectorSize();

    // Get/Set
    template <typename T> Scalar<T> Get(const T &v, size_t i);
    template <typename T> void Set(T &v, size_t i, Scalar<T> const val);

    // Load/Store
    template <typename T> void Load(T &v, Scalar<T> const *ptr);
    template <typename T> void Store(T const &v, Scalar<T> *ptr);

    // Gather/Scatter
    template <typename T, typename S = Scalar<T>> T Gather(S const *ptr, Index<T> const &idx);
    template <typename T, typename S = Scalar<T>> void Scatter(T const &v, S *ptr, Index<T> const &idx);

    // 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);

}