Speeding up software with VecCore, a portable SIMD library

G. Amadio[1], P. Canal[2], D. Piparo[1], S. Wenzel[1], for the GeantV and ROOT Teams
[1]CERN, [2]FNAL

Introduction

Experiments at CERN have an ever increasing demand for computational resources, be it for simulated collisions or data analysis. In the last few decades, this demand has basically been met by hardware upgrades of the Worldwide LHC Computing Grid. However, with the next runs of the LHC approaching, the expected increases in beam luminosity will push this demand far beyond the limits of what further hardware upgrades can reach. Therefore, in order to bridge the widening gap between the needs of the HEP community and the existing computing resources, HEP software will need to be optimized to be able to fully exploit SIMD and multithreading parallelism available in modern hardware.

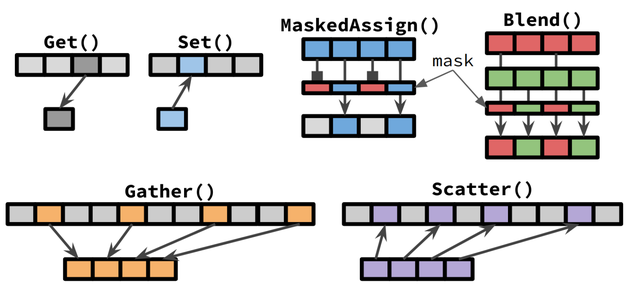
CERN Tier 0 Computing Center (Image: CERN)

One of the key areas where performance can be substantially improved in HEP software is SIMD vectorization. Even so, writing efficient SIMD vectorized code is a significant challenge in many large software projects such as Geant, ROOT, and experiment frameworks. While there are libraries that wrap SIMD intrinsics into a convenient interface, these libraries do not always support every architecture, or simply may not perform well depending on the platform.

The VecCore Library

The VecCore library was created in order to solve the lack of portability and unreliable performance problems usually related to SIMD code by providing a simple API to express SIMD-enabled algorithms that can be dispatched to different backend implementations, such as SIMD libraries like Vc (`https://github.com/VcDevel/Vc`) and UME::SIMD (`https://github.com/edanor/umesimd`) or even CUDA, if the code has the proper annotations (also supported by VecCore).

Using VecCore, developers can write generic computational kernels using abstract types that map to the different concrete types in each backend. The API for vectorization provided by VecCore is architecture-agnostic, making it easy to fallback to scalar types on hardware that does not have SIMD instructions. The API covers the essential parts of the SIMD instruction set that allows one to write many of the numerical algorithms needed by HEP software. The main functions of this API are shown below.

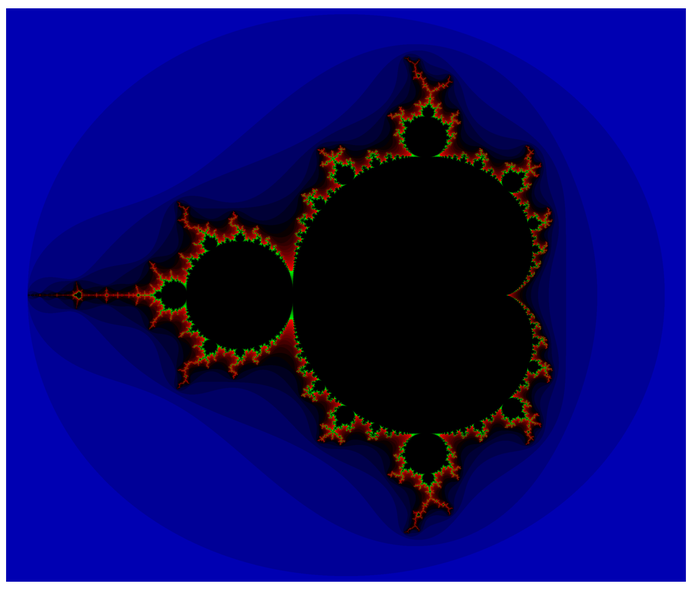


VecCore API

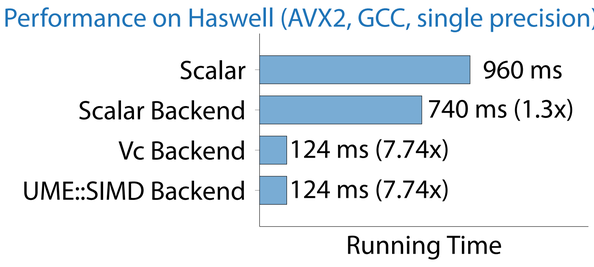
```
namespace vecCore {

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);

} // namespace vecCore
```

Example: Mandelbrot Set

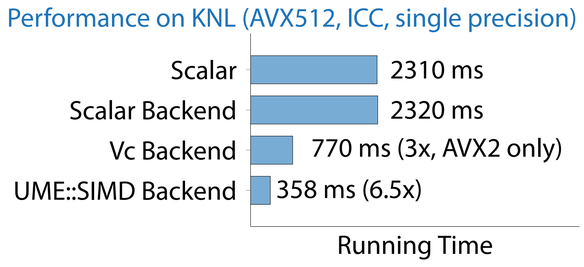
In order to demonstrate how an algorithm changes when using the VecCore API, we created scalar and vectorized implementations of the calculation of the Mandelbrot set. The vectorized version works on several pixels at a time, and both algorithms are single-threaded for simplicity. The final image was generated by computing a color map depending on the number of iterations before the point diverges (leaves the circle $|z| < 2$).



Performance on Haswell (AVX2, GCC, single precision)



Performance on KNL (AVX512, ICC, single precision)



The vectorization speedup on Haswell with AVX2 is almost ideal (7.74x when maximum is 8x). On Knights Landing (KNL), the SIMD vectors are too long, so the probability of getting out of the inner loop prevents better speedups. However, once vectorization gains are combined with multithreading, KNL can far exceed the speedup achievable on Haswell.

Scalar Implementation

```
template<typename T>
void mandelbrot(T xmin, T xmax, size_t nx,
                T ymin, T ymax, size_t ny,
                size_t max_iter, unsigned char *image)
{
    T dx = (xmax - xmin) / T(nx);
    T dy = (ymax - ymin) / T(ny);

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; ++j) {
            size_t k = 0;
            T x = xmin + T(i) * dx, cr = x, zr = x;
            T y = ymin + T(j) * dy, ci = y, zi = y;

            do {
                x  = zr*zr - zi*zi + cr;
                y  = 2.0 * zr*zi + ci;
                zr = x;
                zi = y;
            } while (++k < max_iter && (zr*zr+zi*zi < 4.0));

            image[ny*i+j] = k;
        }
    }
}
```

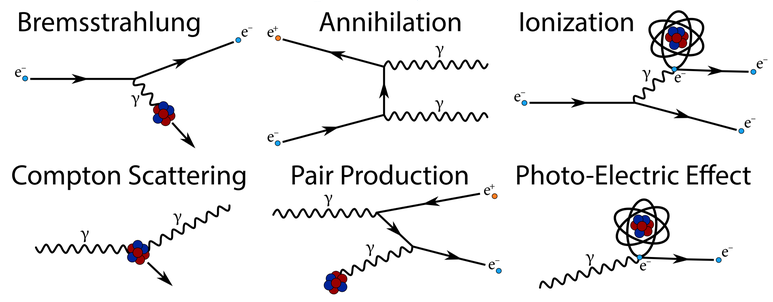
VecCore Implementation

```
#include <VecCore/VecCore>

using namespace vecCore;

template<typename T>
void mandelbrot_v(Scalar<T> xmin, Scalar<T> xmax, size_t nx,
                  Scalar<T> ymin, Scalar<T> ymax, size_t ny,
                  Scalar<Index<T>> max_iter,
                  unsigned char *image)
{
    T iota;
    for (size_t i = 0; i < VectorSize<T>(); ++i)
        Set<T>(iota, i, i);

    T dx = T(xmax - xmin) / T(nx);
    T dy = T(ymax - ymin) / T(ny), dyv = iota * dy;

    for (size_t i = 0; i < nx; ++i) {
        for (size_t j = 0; j < ny; j += VectorSize<T>()) {
            Scalar<Index<T>> k{0};
            T x = xmin + T(i) * dx,       cr = x, zr = x;
            T y = ymin + T(j) * dy + dyv, ci = y, zi = y;

            Index<T> kv{0};
            Mask<T> m{true};

            do {
                x = zr*zr - zi*zi + cr;
                y = T(2.0) * zr*zi + ci;
                MaskedAssign<T>(zr, m, x);
                MaskedAssign<T>(zi, m, y);
                MaskedAssign<Index<T>>(kv, m, ++k);
                m = zr*zr + zi*zi < T(4.0);
            } while (k < max_iter && !MaskEmpty(m));

            for (size_t k = 0; k < VectorSize<T>(); ++k)
                image[ny*i+j+k] = (unsigned char) Get(kv, k);
        }
    }
}
```

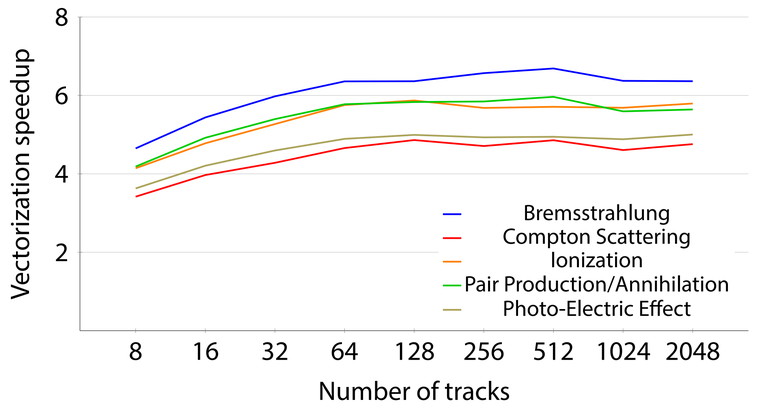
Electromagnetic Physics Models

In GeantV, VecCore has been used to create vectorized versions of electromagnetic physics models. Since energetic photons and electrons produce *particle showers*, they are created much more frequently compared to other particle types during simulation. Therefore, electromagnetic physics processes can take a large portion of computing time and are the natural first targets for vectorization. The main electromagnetic processes and vectorization speedups obtained on Intel® Xeon Phi™ are shown in the figures below.

Electromagnetic Physics Processes



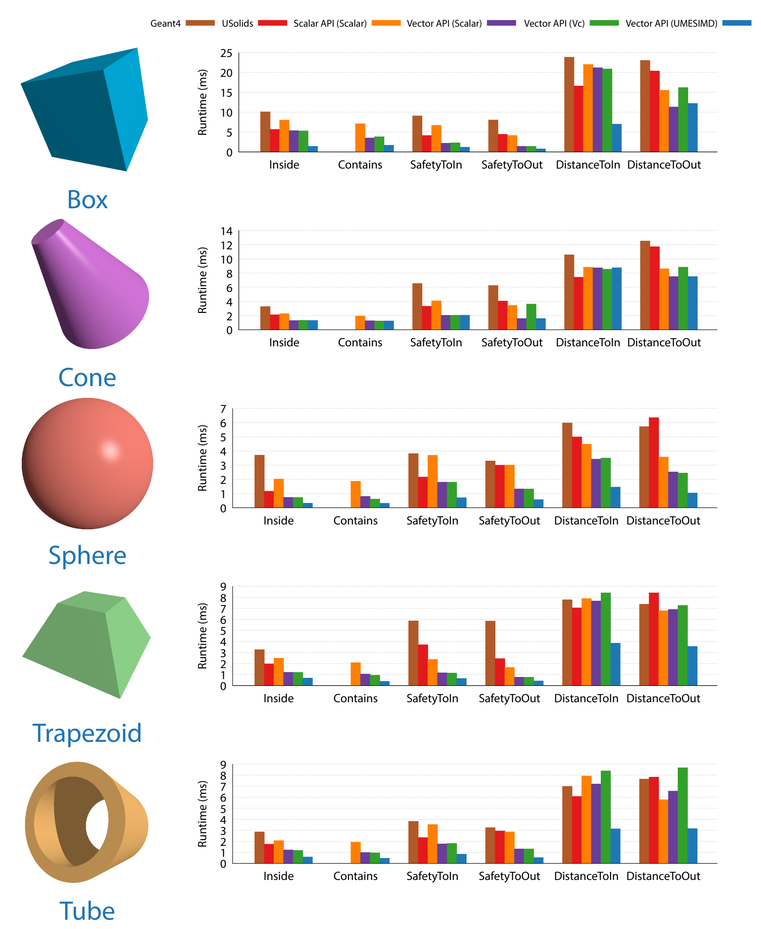
Vectorization Speedup of Physics Models on Intel® Xeon Phi™ (KNC)



Geometry Algorithms

The other compute-intensive component of simulations of large LHC-scale detectors is navigating their geometry. The VecGeom geometry library, used in GeantV and recent versions of Geant4, has introduced a vectorized multi-particle API based on VecCore to perform ray casting, distance calculations, and navigation in sets of particles within detector geometries. The figure below shows performance gains compared to previous implementations and the scalar API.

Vectorization Speedup of Selected Shapes on Intel® Xeon Phi™ (KNL)



VecCore is now integrated into ROOT, where it is being used to vectorize classes used in data fitting. For more information, please see the poster "Parallelization and Vectorization of ROOT Fitting Classes", by Xavier Valls Pla.

Acknowledgements

VecCore development has been funded by an Intel® Parallel Computing Center project at São Paulo State University (UNESP), Fermilab, and CERN, as part of the GeantV project. VecCore is now maintained by the ROOT Team.

References

1. VecCore Library `https://github.com/root-project/veccore`
2. "The GeantV project: preparing the future of simulation", J. Phys.: Conf. Ser. 664 072006 (2015) `http://dx.doi.org/10.1088/1742-6596/664/7/072006`
3. "A concurrent vector-based steering framework for particle transport", J. Phys.: Conf. Ser. 523 012004 (2014) `http://dx.doi.org/10.1088/1742-6596/523/1/012004`