Fermilab

Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

First steps on vectorization of LArSoft simulation readout

Guilherme Lima, Gianluca Petrillo, Erica Snider
Fermi National Accelerator Laboratory (USA)

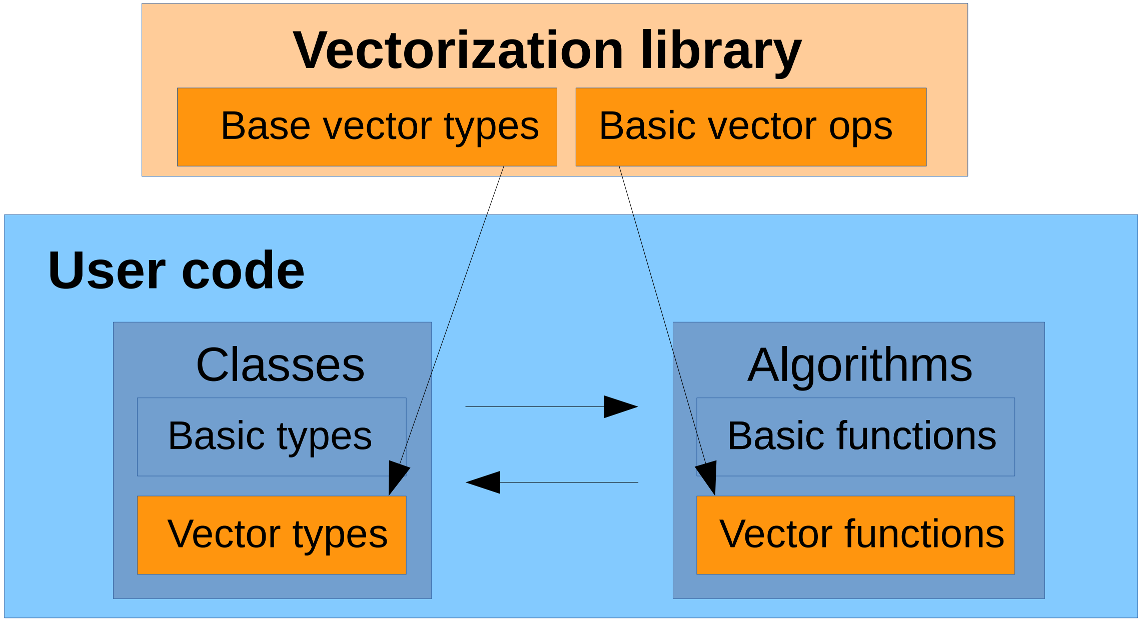
HSF Workshop – Simulation Section
Napoli, Italy, March 26-29, 2018

Outline

- SIMD vectorization
  - why: faster sim infrastructure into LArSoft
  - how: veccore types + vectorized algorithms
- Plans
  - use profiling to search for good candidates
  - simple LArSoft algorithms as basis for vectorization tests and benchmarks
  - evaluate gains and needed changes
- Perspectives

Vectorization libraries

- Vectorization libraries provide high level types to explicitly leverage SIMD vectorization without sacrificing portability, readability or maintainability
- User code is written in terms of vectorized types and preprocessor macros provided by vectorization library
- Undesired issue: strong dependence on a third-party vectorization library
  - mitigated using VecCore (see next slides)
- Examples of libraries:
  - M.Kretzman's Vc library
  - P.Karpinski's Ume::SIMD library
  - Agner Fog's Vector Class library
  - several others



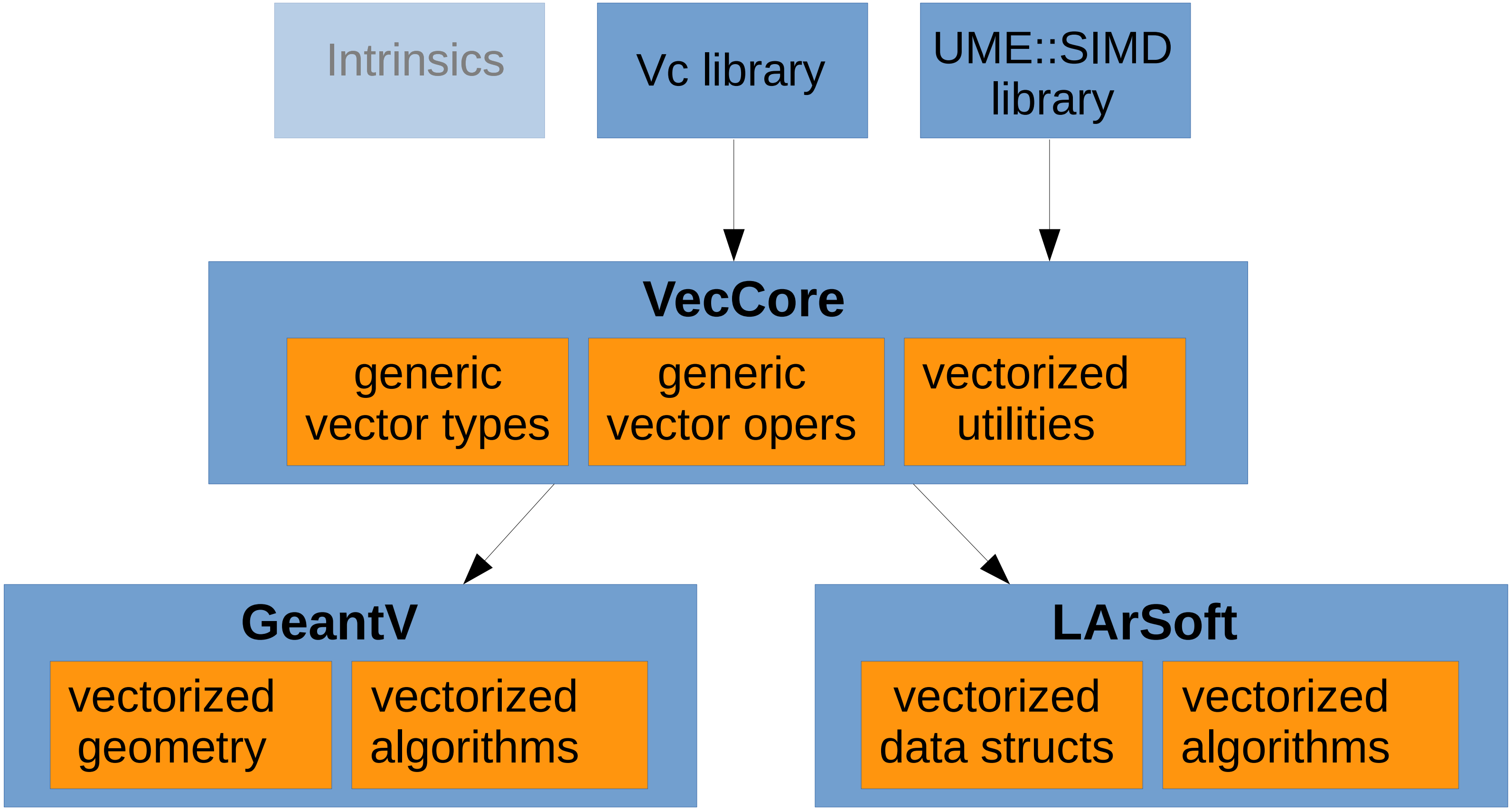
Introducing VecCore

- Developed within GeantV project, then integrated into ROOT
- Provides a uniform interface for SIMD vectorization
  - Backends form a coherent set of types to be used together
  - Arithmetics, comparisons, logical operators
  - Vectorized math functions
  - Masking/blending operations
  - Gather/Scatter operations
  - Support for multiple architectures without code duplication

- Supports multiple backend implementations
  - Scalar/CUDA
  - Vc Library — https://github.com/VcDevel/Vc
  - UME::SIMD — https://github.com/edanor/umesimd

- See these slides for more information about VecCore

Introducing VecCore



VecCore details

- Source: http://github.com/root-project/veccore
- Generic vectorized types
  - Real_v, Float_v, Double_v, Int_v, Int16_v, Int32_v, Int64_v, UInt_v, …, UInt64_v
    → relevant algorithms re-written in terms of these generic vectorized types
- Vectorized operations
  - Arithmetics, MaskedAssign(), Blend(), IsFull(), isEmpty(), EarlyReturnsAllowed()
- Implementation backends
  - Scalar, ScalarWrapper
  - VcScalar, VcVector, VcSimdArray<N>
  - UMESimd, UMESimdArray<N>

- Implementation is selected at compilation time via CMake switches (if supported by the system)
  - -DVC=[ON|off] -DUMESIMD=[on|OFF] -DCUDA=[on|OFF]
  - Note that carefully designed programs can use multiple backends at the same time (e.g. quadratic solver benchmark under veccore/bench/)
  - Also supports GPUs (through CUDA)

- See these slides for more information about veccore

Choosing good vectorization candidates

- Some profiling performed on LArSoft code by S.Y.Jun (FNAL)
- Started search in signal processing and hit finding algorithms
  - profiling results from Soon Y. Jun
    - look at *detsim* numbers on the proton_6GeV job, listed by functions (FUNS):
      - 9.9% IdealAdcSimulator::count(…)
      - 9.8% Legacy35tZeroSuppressService::filter(…)
      - 6.7% ExponentialChannelNoiseService::addNoise(…)
      (…)

dupetpc_06_57_00 LArSoft/protoDune (proton 6GeV detsim)

CPUTIME(Inclusive)	CPUTIME(Exclusive)	Function
1.21e+09 100 %	1.21e+09 100 %	Experiment Aggregate Metrics
1.61e+08 13.4%	1.61e+08 13.4%	__GI_memcpy
1.20e+08 9.9%	1.20e+08 9.9%	IdealAdcSimulator::count(double, unsigned int, unsigned int) const
1.38e+08 11.5%	1.18e+08 9.8%	Legacy35tZeroSuppressService::filter(std::vector > const&, unsigned int, float, std::vector
8.30e+07 6.9%	8.12e+07 6.7%	ExponentialChannelNoiseService::addNoise(unsigned int, std::vector >&) const
7.24e+08 60.0%	6.94e+07 5.8%	SimWireDUNE::produce(art::Event&)
4.91e+07 4.1%	4.91e+07 4.1%	AdcCodeHelper::subtract(short, float) const

- All good candidates, but they are DUNE experiment code, not LArSoft

How is GetDist2(...) CPU time?

dupetpc_06_57_00 LArSoft/Dune-FD (prodgenie_nue-dune10kt_1x2x6 reco)

5.60e+10 66.5%	5.06e+08 0.6%	operator()
4.99e+08 0.6%	4.99e+08 0.6%	tbb::internal::generic_scheduler::lock_task_pool(tbb::internal::arena_slot*) const
4.67e+08 0.6%	4.67e+08 0.6%	tbb::internal::cpu_ctl_env::operator!=(tbb::internal::cpu_ctl_env const&) const
4.46e+08 0.5%	4.46e+08 0.5%	pma::Segment3D::GetDist2(TVector2 const&, TVector2 const&, TVector2 const&)
3.81e+08 0.5%	3.81e+08 0.5%	tbb::internal::generic_scheduler::steal_task(tbb::internal::arena_slot&)
8.94e+08 1.1%	2.90e+08 0.3%	pma::Segment3D::SumDist2Hits() const
2.79e+08 0.3%	2.79e+08 0.3%	tbb::internal::prolonged_pause()

dupetpc_06_57_00 LArSoft/protoDune (proton 6GeV reco)

CPI metric (not CPU time!)

CPI	PAPI_TOT_CYC:(I)	PAPI_TOT_CYC:(E)	PAPI_TOT_INS:(I)	PAPI_TOT_INS:(E)	Function
5.50e-01	1.66e+13 100 %	1.66e+13 100 %	3.02e+13 100 %	3.02e+13 100 %	Experiment Aggregate Metrics
4.01e-01	3.85e+12 23.2%	3.85e+12 23.2%	9.60e+12 31.8%	9.60e+12 31.8%	Eigen_tf::internal::gebp_kernel, 8, 4, fals float, long, long, long, long) [clone .const
4.59e-01	1.32e+12 7.9%	1.32e+12 7.9%	2.87e+12 9.5%	2.87e+12 9.5%	pma::Segment3D::GetDist2(TVector2 co
7.68e-01	8.60e+11 5.2%	8.56e+11 5.2%	1.12e+12 3.7%	1.12e+12 3.7%	__ieee754_exp
3.61e-01	7.67e+11 4.6%	7.67e+11 4.6%	2.12e+12 7.0%	2.12e+12 7.0%	Eigen_tf::internal::gebp_kernel, 8, 4, fals float, long, long, long, long) [clone .const
4.16e-01	6.52e+11 3.9%	6.52e+11 3.9%	1.57e+12 5.2%	1.57e+12 5.2%	Eigen_tf::internal::TensorContractionInpu const> const> const, Eigen_tf::ThreadPo const
6.95e-01	1.93e+12 11.6%	6.26e+11 3.8%	3.45e+12 11.4%	9.01e+11 3.0%	pma::Segment3D::SumDist2Hits() const
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Eigen_tf::internal::gemm_pack_rhs cons const, Eigen_tf::ThreadPoolDevice>, std

PMAlg::Segment3D::GetDist2(...)

```
double pma::Segment3D::GetDist2(const TVector2& psrc, const TVector2& p0, const TVector2& p1)
{
        pma::Vector2D v0(psrc.X() - p0.X(), psrc.Y() - p0.Y());
        pma::Vector2D v1(p1.X() - p0.X(), p1.Y() - p0.Y());
        pma::Vector2D v2(psrc.X() - p1.X(), psrc.Y() - p1.Y());

        double v1Norm2 = v1.Mag2();
        if (v1Norm2 >= 1.0E-6) // >= 0.01mm
        {
            double v0v1 = v0.Dot(v1);
            double v2v1 = v2.Dot(v1);
            double v0Norm2 = v0.Mag2();
            double v2Norm2 = v2.Mag2();

            double result = 0.0;
            if ((v0v1 > 0.0) && (v2v1 < 0.0))
            {
                    double cosine01_square = 0.0;
                    double mag01_square = v0Norm2 * v1Norm2;
                    if (mag01_square != 0.0) cosine01_square = v0v1 * v0v1 / mag01_square;

                    result = (1.0 - cosine01_square) * v0Norm2;
            }
            else // increase distance to prefer hit assigned to the vertex, not segment
            {
                    if (v0v1 <= 0.0) result = 1.0001 * v0Norm2;
                    else result = 1.0001 * v2Norm2;
            }
            if (result >= 0.0) return result;
            else return 0.0;
        }
        else // short segment or its projection
        {
                double dx = 0.5 * (p0.X() + p1.X()) - psrc.X();
                double dy = 0.5 * (p0.Y() + p1.Y()) - psrc.Y();
                return dx * dx + dy * dy;
        }
}
```

Vector arithmetics are usually easy to SIMD-vectorize.
Created a vectorized version of this function (see next slide)
and a benchmark for comparisons

Generic (vectorized) GetDist2(...) function

```
//=====  explicit SIMD code

template <typename Real_v>
VECGEOM_FORCE_INLINE
void GetDist2SIMD(const Vector3D<Real_v>& ps, const Vector3D<Real_v> &p0,
                  const Vector3D<Real_v> &p1, Real_v *__restrict__ result)
{
  using Bool_v  = vecCore::Mask_v<Real_v>;

  const Vector3D<Real_v> v0(ps - p0);
  const Vector3D<Real_v> v1(p1 - p0);
  const Vector3D<Real_v> v2(ps - p1);

  // most common case: if (v1Norm2 >= 1.0E-6) {  // >= 0.01mm
  const Real_v zero = Real_v(0.0);
  const Real_v v0v1 = v0.Dot(v1);
  const Real_v v2v1 = v2.Dot(v1);
  const Real_v v0Norm2 = v0.Mag2();
  const Real_v v1Norm2 = v1.Mag2();
  const Real_v v2Norm2 = v2.Mag2();

  const Bool_v opposite  = (v0v1 > zero) && (v2v1 < zero);
  const Real_v mag01_square = v0Norm2 * v1Norm2;
  const Real_v cosine01_square = v0v1 * v0v1 / NonZero(mag01_square);
  *result = (Real_v(1.0) - cosine01_square) * v0Norm2;

  const Bool_v nonnegv2v1 = (v2v1 >= zero) && (!opposite);
  const Bool_v nonposv0v1 = (v0v1 <= zero) && (!opposite);
  vecCore::MaskedAssign( *result, nonnegv2v1, Real_v(1.0001) * v2Norm2);
  vecCore::MaskedAssign( *result, nonposv0v1, Real_v(1.0001) * v0Norm2);
  vecCore::MaskedAssign( *result, *result <= zero, zero);

  Bool_v close = (v1Norm2 < Real_v(1.0e-6));
  if (!vecCore::MaskEmpty(close)) { // for a short p1-p0 segment or its projection
    Vector3D<Real_v> temp = 0.5 * (p0 + p1) - ps;
    vecCore::MaskedAssign( *result, close, temp.Mag2());
  }
  return;
}
```

templated on a FP type → scalar type (float, double), or vector type (Float_v, Double_v)

consts help compiler optimizations

avoid divisions by zero without adding if(cond)

masks used as conditions...

...in MaskedAssigns to replace if(cond)

This version with vector types processes large numbers of points 3x faster!

Current status

- A few Dune and LArSoft functions identified as candidates
  - Profiling shows that our current DUNE candidates are promising (~10% CPU time)
  - Started with a LArSoft Segment3D::GetDist2(...) function for a first evaluation → observed 3x faster using vector types!

- I am currently working on vectorized versions of a few specific functions, and on benchmarks to assess performance improvements
  - Using VecCore types and Vc library backend

- Also porting vectorized code back to LArSoft
  - adding dependences on VecCore headers and Vc library for builds

- Planning to use vectorize DUNE functions (larger relative contributions)
- First preliminary results look promising