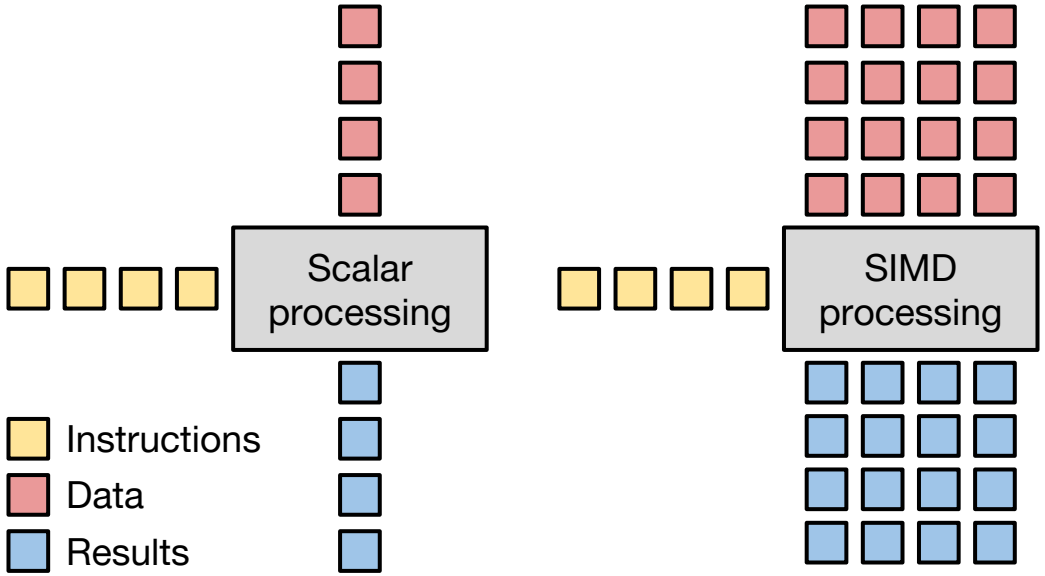
Portable SIMD and the VecCore Library

Guilherme Amadio, Philippe Canal, Sandro Wenzel

for the GeantV development team

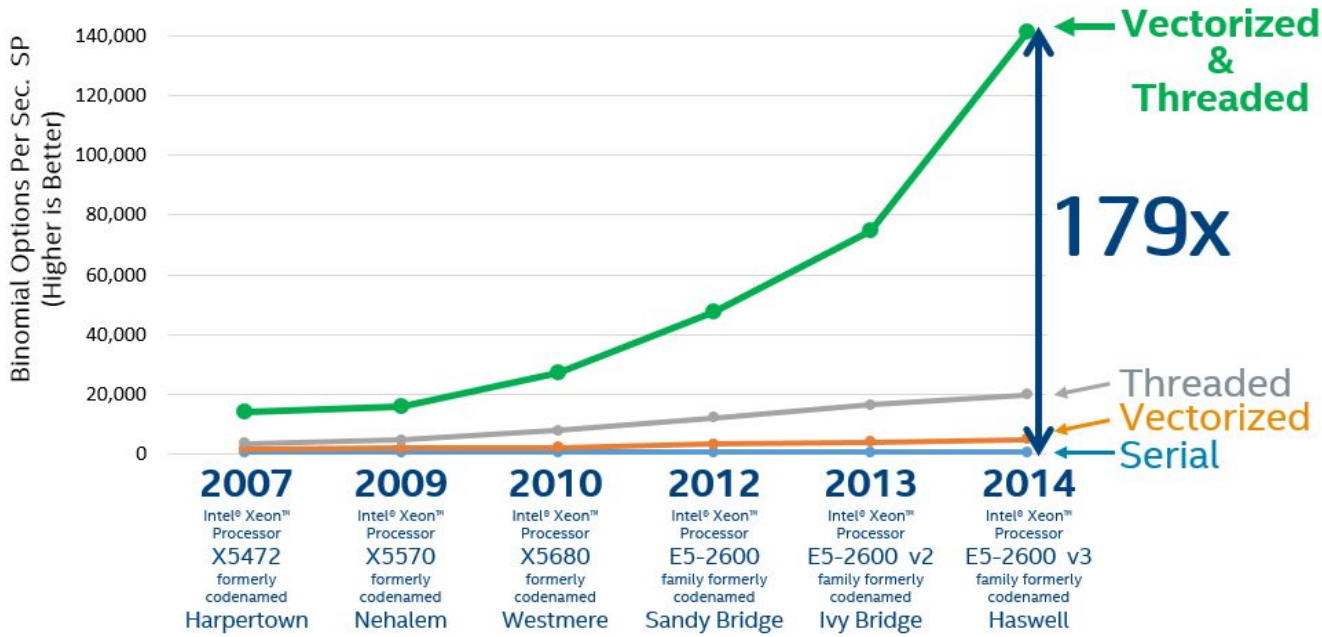
What is SIMD vectorization?

- Instruction-level parallelism
- SIMD → Single Instruction, Multiple Data



Why SIMD vectorization?

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™ processors.



source: Intel

History of Intel® SIMD ISA Extensions

- 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 Skylake Xeon™ and Xeon Phi™ Knights Landing (2016)

  Conflict detection instructions

 = 32 bit word

SIMD Programming Models

- Auto-vectorization
- OpenMP 4.1
- Compiler Pragmas
- SIMD Libraries
- Compiler Intrinsics
- Inline Assembly

```
float a[N], b[N], c[N];

for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
float a[N], b[N], c[N];

#pragma omp simd
#pragma ivdep
for (int i = 0; i < N; i++)
  a[i] = b[i] * c[i];
```

```
#include <Vc/Vc>
Vc::SimdArray<float, N> a, b, c;

a = b * c;
```

```
#include <x86intrin.h>
__m256 a, b, c;

a = _mm256_mul_ps(b, c);
```

```
asm volatile("vmulps %ymm1, %ymm0");
```

Why is VecCore necessary?

- Performance with auto-vectorization varies wildly for different compilers and versions
  - Intel® C/C++ compiler is significantly ahead of GCC and Clang
- Compiler intrinsics are not an ideal interface
  - Limited to C name mangling, so portability is an issue
- Libraries do not always work well across all architectures
  - No KNL support in Vc
  - UME::SIMD uses scalar emulation for AVX2 (not as good as KNL)
- Different programming models for Intel® Arch vs CUDA
- Still need portable solution for when no library is available

History and General Project Info

- VecCore is an evolution of the backend system that existed in VecGeom prior to VecCore
  - Unification and formalization of the backend API
  - New features (i.e. gather/scatter interface)
- VecCore is hosted in VecGeom's Git repository: https://gitlab.cern.ch/VecGeom/VecGeom
- Hardware architectures
  - Intel® Architectures (SSE4.2, AVX/AVX2, IMCI, AVX512)
  - Nvidia GPUs via CUDA
  - ARM neon and PowerPC Altivec *not yet supported*

Scope & Objectives

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

Vc Library

"Vc: A C++ library for explicit vectorization",
Software: Practice and Experience (2011),
M. Kretz and V. Lindenstruth, http://dx.doi.org/10.1002/spe.1149

Vc supports SSE2, SSE3, SSE4.2, AVX, AVX2, and IMCI (MIC).
Vc works very well with both GCC and Clang compilers, and performs best on hardware supporting up to the AVX2 instruction set.

Support for ARM NEON, Nvidia CUDA, and AVX512 is under development, but not yet available. Matthias Kretz is working with C++ standard committee to add SIMD support into the language.

UME::SIMD Library

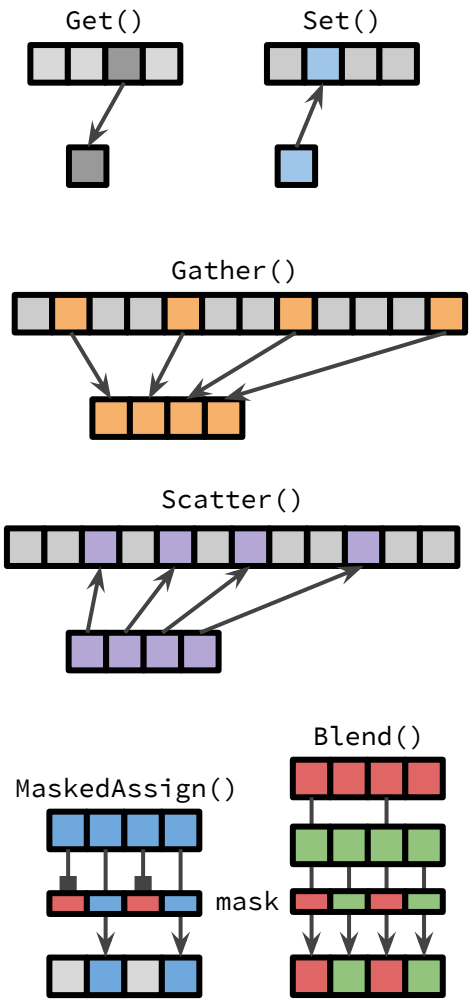
UME::SIMD is a library for explicit vectorization developed by Przemyslaw Karpinsky (CERN OpenLab).

UME::SIMD has a homogeneous interface for accessing functionality of SIMD registers of AVX, AVX2, AVX512 and IMCI (KNCNI, k1om) instruction sets, and offers the best performance on the Intel® Xeon Phi™ Knights Landing (KNL).
UME::SIMD also supports ARM and support for PowerPC's Altivec is nearly complete.

UME::SIMD was developed as part of an ICE-DIP (European Industrial Doctorate) project at CERN.

VecCore Library 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
```



VecCore Backends

Scalar Backend

```
namespace vecCore {

template <typename T> struct TypeTraits {
  using ScalarType = T;
  using MaskType   = bool;
  using IndexType  = size_t;
};

namespace backend {

template <typename T = Real_s> class ScalarT {
public:
  using Real_v   = T;
  using Float_v  = Float_s;
  using Double_v = Double_s;

  using Int_v   = Int_s;
  using Int32_v = Int32_s;
  using Int64_v = Int64_s;

  using UInt_v   = UInt_s;
  using UInt32_v = UInt32_s;
  using UInt64_v = UInt64_s;
};

using Scalar = ScalarT<>;

} // namespace backend
} // namespace vecCore
```

Vc::Vector Backend

```
#include <Vc/Vc>

namespace vecCore {

template <typename T> struct TypeTraits<Vc::Vector<T>> {
  using ScalarType = T;
  using MaskType   = typename Vc::Vector<T>::MaskType;
  using IndexType  = typename Vc::Vector<T>::IndexType;
};

namespace backend {

template <typename T = Real_s> class VcVectorT {
public:
  using Real_v   = Vc::Vector<T>;
  using Float_v  = Vc::Vector<Float_s>;
  using Double_v = Vc::Vector<Double_s>;

  using Int_v   = Vc::Vector<Int_s>;
  using Int32_v = Vc::Vector<Int32_s>;
  using Int64_v = Vc::Vector<Int64_s>;

  using UInt_v   = Vc::Vector<UInt_s>;
  using UInt32_v = Vc::Vector<UInt32_s>;
  using UInt64_v = Vc::Vector<UInt64_s>;
};

using VcVector = VcVectorT<>;

} // namespace backend
} // namespace vecCore
```

Code Example: Quadratic Solver

Simple Implementation

```
template <typename T> int QuadSolve(T a, T b, T c, T &x1, T &x2)
{
  T delta = b * b - 4.0 * a * c;

  if (delta < 0.0) return 0;

  if (delta < std::numeric_limits<T>::epsilon()) {
    x1 = x2 = -0.5 * b / a;
    return 1;
  }

  if (b >= 0.0) {
    x1 = -0.5 * (b + std::sqrt(delta)) / a;
    x2 = c / (a * x1);
  } else {
    x2 = -0.5 * (b - std::sqrt(delta)) / a;
    x1 = c / (a * x2);
  }

  return 2;
}
```

Code Example: Quadratic Solver

Optimized Implementation

```
template <typename T> void QuadSolve(const T &a, const T &b, const T &c, T &x1, T &x2, int &roots)
{
  T a_inv = T(1.0) / a;
  T delta = b * b - T(4.0) * a * c;
  T s     = (b >= 0) ? T(1.0) : T(-1.0);

  roots = delta > numeric_limits<T>::epsilon() ? 2 : delta < T(0.0) ? 0 : 1;

  switch (roots) {
  case 2:
    x1 = T(-0.5) * (b + s * std::sqrt(delta));
    x2 = c / x1;
    x1 *= a_inv;
    return;

  case 0:
    return;

  case 1:
    x1 = x2 = T(-0.5) * b * a_inv;
    return;

  default:
    return;
  }
}
```

Code Example: Quadratic Solver

AVX2 Intrinsics Implementation

```
void QuadSolveAVX(const float *__restrict__ a, const float *__restrict__ b, const float *__restrict__ c,
                  float *__restrict__ x1, float *__restrict__ x2, int *__restrict__ roots)
{
  __m256 one    = _mm256_set1_ps(1.0f);
  __m256 va     = _mm256_load_ps(a);
  __m256 vb     = _mm256_load_ps(b);
  __m256 zero   = _mm256_set1_ps(0.0f);
  __m256 a_inv  = _mm256_div_ps(one, va);
  __m256 b2     = _mm256_mul_ps(vb, vb);
  __m256 eps    = _mm256_set1_ps(std::numeric_limits<float>::epsilon());
  __m256 vc     = _mm256_load_ps(c);
  __m256 negone = _mm256_set1_ps(-1.0f);
  __m256 ac     = _mm256_mul_ps(va, vc);
  __m256 sign   = _mm256_blendv_ps(negone, one, _mm256_cmp_ps(vb, zero, _CMP_GE_OS));
#if defined(__FMA__)
  __m256 delta = _mm256_fmadd_ps(_mm256_set1_ps(-4.0f), ac, b2);
  __m256 r1    = _mm256_fmadd_ps(sign, _mm256_sqrt_ps(delta), vb);
#else
  __m256 delta = _mm256_sub_ps(b2, __256_mul_ps(_mm256_set1_ps(-4.0f), ac));
  __m256 r1    = _mm256_add_ps(vb, _mm256_mul_ps(sign, _mm256_sqrt_ps(delta)));
#endif
  __m256 mask0 = _mm256_cmp_ps(delta, zero, _CMP_LT_OS);
  __m256 mask2 = _mm256_cmp_ps(delta, eps, _CMP_GE_OS);
  r1           = _mm256_mul_ps(_mm256_set1_ps(-0.5f), r1);
  __m256 r2    = _mm256_div_ps(vc, r1);
  r1           = _mm256_mul_ps(a_inv, r1);
  __m256 r3    = _mm256_mul_ps(_mm256_set1_ps(-0.5f), _mm256_mul_ps(vb, a_inv));
  __m256 nr    = _mm256_blendv_ps(one, _mm256_set1_ps(2), mask2);
  nr           = _mm256_blendv_ps(nr, _mm256_set1_ps(0), mask0);
  r3           = _mm256_blendv_ps(r3, zero, mask0);
  r1           = _mm256_blendv_ps(r3, r1, mask2);
  r2           = _mm256_blendv_ps(r3, r2, mask2);
  _mm256_store_si256((__m256i *)roots, _mm256_cvtps_epi32(nr));
  _mm256_store_ps(x1, r1);
  _mm256_store_ps(x2, r2);
}
```

Code Example: Quadratic Solver

AVX2 Intrinsics Implementation

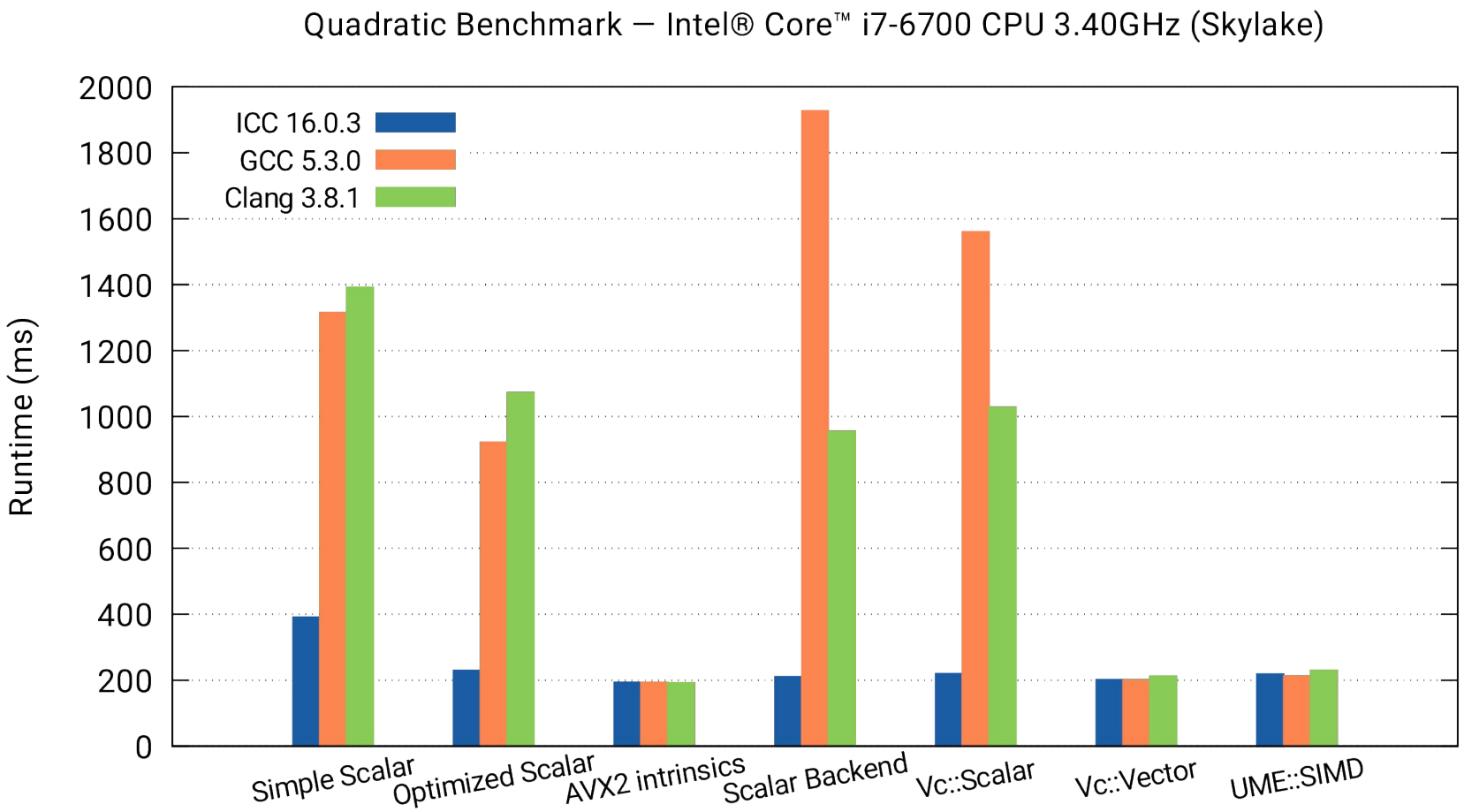
```
void QuadSolveAVX(const float *__restrict__ a, const float *__restrict__ b, const float *__restrict__ c,
                  float *__restrict__ x1, float *__restrict__ x2, int *__restrict__ roots)
{
  __m256 one    = _mm256_set1_ps(1.0f);
  __m256 va     = _mm256_load_ps(a);
  __m256 vb     = _mm256_load_ps(b);
  __m256 zero   = _mm256_set1_ps(0.0f);
  __m256 a_inv  = _mm256_div_ps(one, va);
  __m256 b2     = _mm256_mul_ps(vb, vb);
  __m256 eps    = _mm256_set1_ps(std::numeric_limits<float>::epsilon());
  __m256 vc     = _mm256_load_ps(c);
  __m256 negone = _mm256_set1_ps(-1.0f);
  __m256 ac     = _mm256_mul_ps(va, vc);
  __m256 sign   = _mm256_blendv_ps(negone, one, _mm256_cmp_ps(vb, zero, _CMP_GE_OS));
#if defined(__FMA__)
  __m256 delta = _mm256_fmadd_ps(_mm256_set1_ps(-4.0f), ac, b2);
  __m256 r1    = _mm256_fmadd_ps(sign, _mm256_sqrt_ps(delta), vb);
#else
  __m256 delta = _mm256_sub_ps(b2, __256_mul_ps(_mm256_set1_ps(-4.0f), ac));
  __m256 r1    = _mm256_add_ps(vb, _mm256_mul_ps(sign, _mm256_sqrt_ps(delta)));
#endif
  __m256 mask0 = _mm256_cmp_ps(delta, zero, _CMP_LT_OS);
  __m256 mask2 = _mm256_cmp_ps(delta, eps, _CMP_GE_OS);
  r1           = _mm256_mul_ps(_mm256_set1_ps(-0.5f), r1);
  __m256 r2    = _mm256_div_ps(vc, r1);
  r1           = _mm256_mul_ps(a_inv, r1);
  __m256 r3    = _mm256_mul_ps(_mm256_set1_ps(-0.5f), _mm256_mul_ps(vb, a_inv));
  __m256 nr    = _mm256_blendv_ps(one, _mm256_set1_ps(2), mask2);
  nr           = _mm256_blendv_ps(nr, _mm256_set1_ps(0), mask0);
  r3           = _mm256_blendv_ps(r3, zero, mask0);
  r1           = _mm256_blendv_ps(r3, r1, mask2);
  r2           = _mm256_blendv_ps(r3, r2, mask2);
  _mm256_store_si256((__m256i *)roots, _mm256_cvtps_epi32(nr));
  _mm256_store_ps(x1, r1);
  _mm256_store_ps(x2, r2);
}
```

Code Example: Quadratic Solver

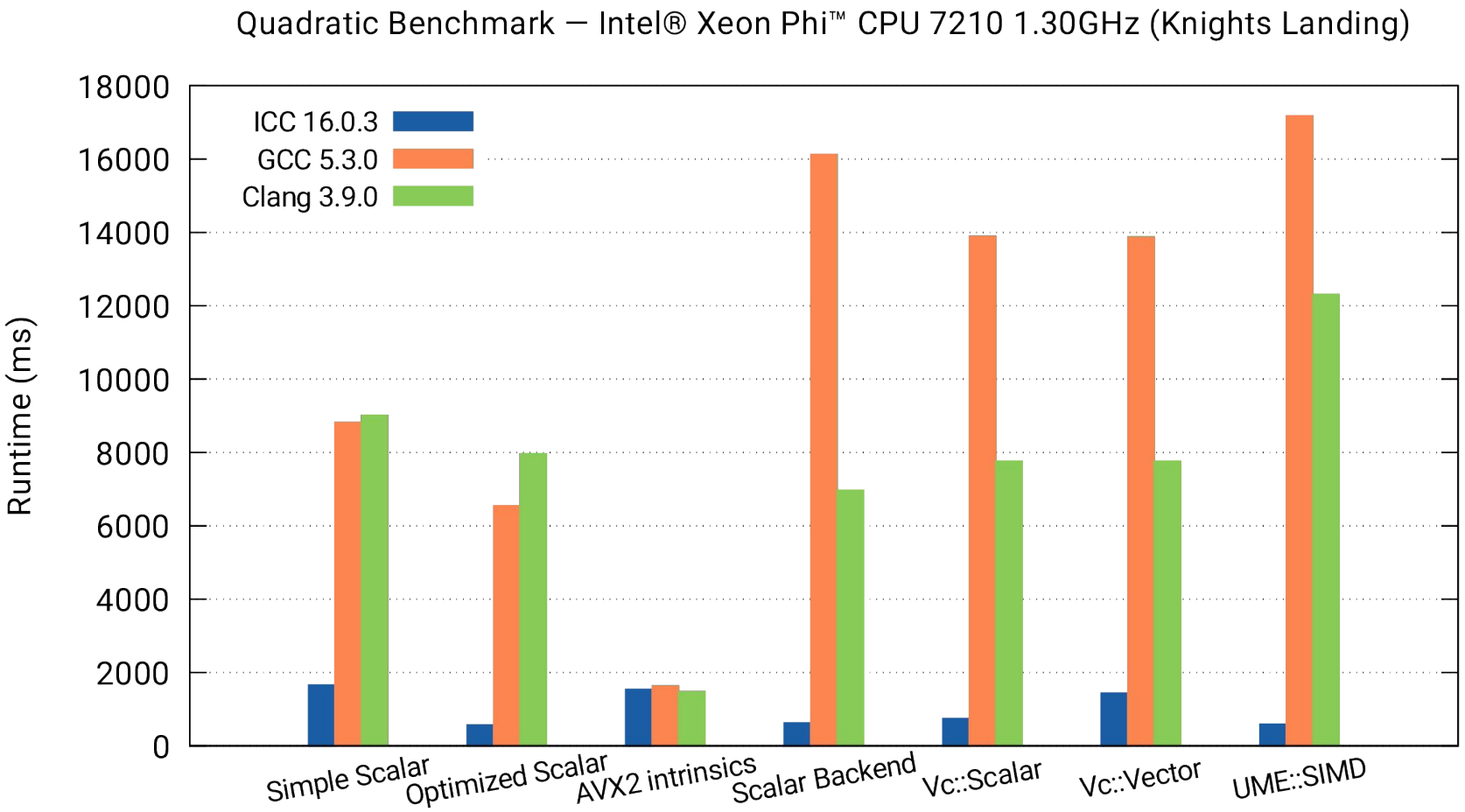
VecCore API Implementation

```
template <typename Float_v, typename Int32_v>
void QuadSolveVecCore(Float_v const &a, Float_v const &b, Float_v const &c,
                      Float_v &x1, Float_v &x2, Int32_v &roots)
{
  Float_v a_inv = Float_v(1.0f) / a;
  Float_v delta = b * b - Float_v(4.0f) * a * c;

  Mask<Float_v> mask0(delta < Float_v(0.0f));
  Mask<Float_v> mask2(delta >= NumericLimits<Float_v>::Epsilon());

  Float_v root1 = Float_v(-0.5f) * (b + math::Sign(b) * math::Sqrt(delta));
  Float_v root2 = c / root1;
  root1         = root1 * a_inv;

  MaskedAssign(x1, mask2, root1);
  MaskedAssign(x2, mask2, root2);
  roots = Blend(Mask<Int32_v>(mask2), Int32_v(2), Int32_v(0));

  Mask<Float_v> mask1 = !(mask2 || mask0);

  if (MaskEmpty(mask1)) return;

  root1 = Float_v(-0.5f) * b * a_inv;
  MaskedAssign(roots, Mask<Int32_v>(mask1), Int32_v(1));
  MaskedAssign(x1, mask1, root1);
  MaskedAssign(x2, mask1, root1);
}
```

Performance Comparison on Intel Core i7



Performance Comparison on Intel Xeon Phi



Real Code Sample: VecGeom Box

Box Implementation of `DistanceToIn()`

```
template <typename Real_v>
void DistanceToIn(UnplacedStruct_t const &box, Vector3D<Real_v> const &point,
                  Vector3D<Real_v> const &direction, Real_v const &stepMax, Real_v &dist)
 {
   const Vector3D<Real_v> invDir(Real_v(1.0) / NonZero(direction[0]),
                                 Real_v(1.0) / NonZero(direction[1]),
                                 Real_v(1.0) / NonZero(direction[2]));

   const Real_v distIn = Max((-Sign(invDir[0]) * box.fDimensions[0] - point[0]) * invDir[0],
                             (-Sign(invDir[1]) * box.fDimensions[1] - point[1]) * invDir[1],
                             (-Sign(invDir[2]) * box.fDimensions[2] - point[2]) * invDir[2]);

  const Real_v distOut = Min((Sign(invDir[0]) * box.fDimensions[0] - point[0]) * invDir[0],
                             (Sign(invDir[1]) * box.fDimensions[1] - point[1]) * invDir[1],
                             (Sign(invDir[2]) * box.fDimensions[2] - point[2]) * invDir[2]);

   dist = Blend(distIn >= distOut || distOut <= Real_v(kTolerance), Infinity<Real_v>(), distIn);
 }
```

Code Sample: EM Physics Models

Interaction Kernel for Ionization Model

```
template <class Backend>
void IonisationMoller::InteractKernel(typename Backend::Double_v energyIn,
                                      Index_v<typename Backend::Double_v> /*zElement*/,
                                      typename Backend::Double_v &energyOut,
                                      typename Backend::Double_v &sinTheta)
{
  using Double_v = typename Backend::Double_v;

  Double_v probNA, fraction, minimumE, deltaE;
  Index_v<Double_v> irow, icol, ncol, index, aliasInd;

  ncol(fAliasSampler->GetSamplesPerEntry());
  fAliasSampler->SampleLogBin<Backend>(energyIn, irow, icol, fraction);

  index = ncol * irow + icol;
  fAliasSampler->GatherAlias<Backend>(index, probNA, aliasInd);

  mininumE = fDeltaRayThreshold;
  deltaE   = energyIn / 2.0 - mininumE;

  energyOut = mininumE + fAliasSampler->SampleX<Backend>(deltaE, probNA, aliasInd, icol, fraction);
  sinTheta  = SampleSinTheta<Backend>(energyIn, energyOut);
}
```

Code Sample: EM Physics Models

Ionization Model Final State

```
template <class Backend>
typename Backend::Double_v
IonisationMoller::SampleSinTheta(typename Backend::Double_v energyIn,
                                 typename Backend::Double_v energyOut)
{
  using Double_v = typename Backend::Double_v;

  // angle of the scatterred electron

  Double_v energy        = energyIn + electron_mass_c2;
  Double_v totalMomentum = math::Sqrt(energyIn * (energyIn + 2.0 * electron_mass_c2));
  Double_v deltaMomentum = math::Sqrt(energyOut * (energyOut + 2.0 * electron_mass_c2));
  Double_v cost          = energyOut * (energy + electron_mass_c2) /
                           (deltaMomentum * totalMomentum);
  Double_v sint2         = 1.0 - cost * cost;

  return Blend(sint2 < 0.0, Double_v(0.0), math::Sqrt(sint2));
}
```

Future Work

- Extend VecCore API (streaming loads/stores, etc)
- Add vectorized pseudo-random number generators
- Add support for managing high-bandwidth memory (HBM)
- Add support for querying CUDA hardware capabilities
- Extend benchmarks to analyze compiler performance
- Optimize Scalar backend with auto-vectorization
- Add more backend implementations
  - Explicit support for more hardware architectures
  - Add more alternatives for Intel® Architectures (Agner Fog, etc)

Thank you!

Questions?