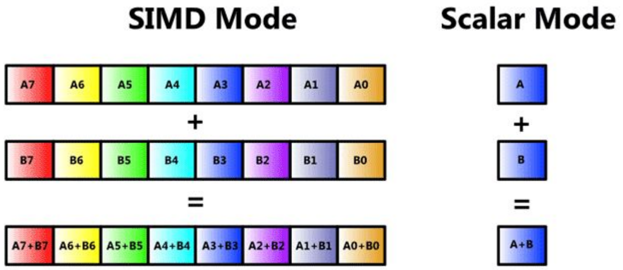
Auto-vectorization: recent progress

Guilherme Amadio, Sofia Vallecorsa

Geant4 Collaboration Meeting

SIMD Vectorization

- SIMD = Single Instruction, Multiple Data
- Auto-vectorization: automatic optimization of scalar code to use SIMD instructions done by the compiler
- Results vary greatly between compilers, but Intel C/C++ compiler has shown very good results recently
- Scalar code must be written carefully to avoid issues like misalignment, aliasing, data dependencies, etc, which prevent vectorization



SIMD Programming Models

- Auto-vectorization
- OpenMP SIMD
- Compiler Pragmas
- SIMD Library
- 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];
```

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

a = b * c;
```

```
__m256 a, b, c;

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

```
__asm {
 vmulps ymm0, ymm1;
};
```

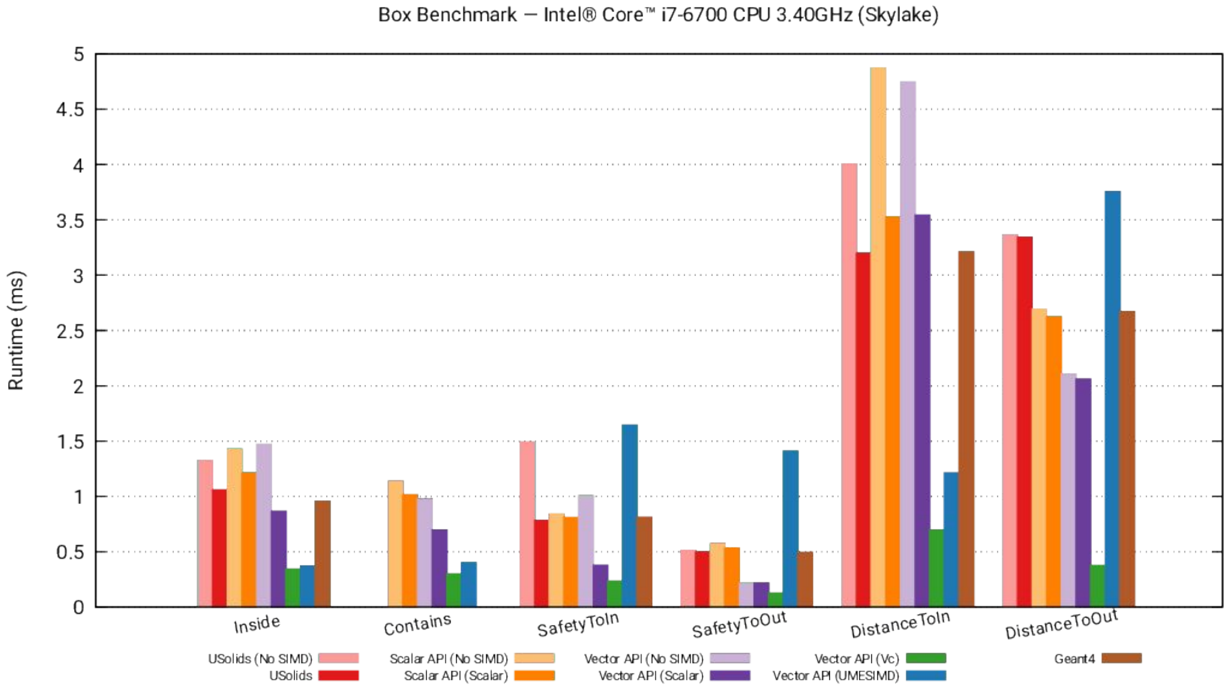
VecCore Backend Interface

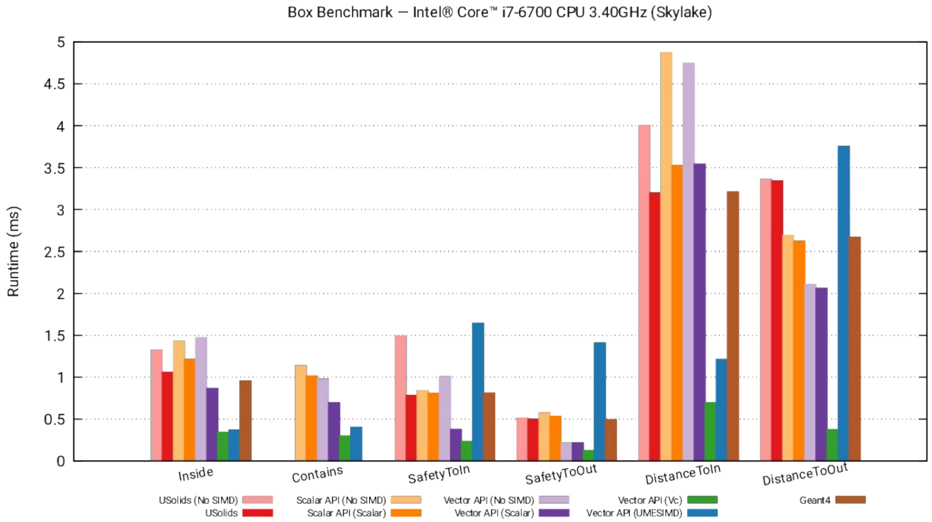
- SIMD Vector Size
- Regular arithmetics operators
- Get/Set individual values in SIMD vector
- Load/Store SIMD vector to memory
- Gather/Scatter SIMD vector from/to non-contiguous memory
- Masking/Blending Operations
- SIMD-enabled math functions
- Implementation varies for each backend
- Main backends: Scalar, Vc, UME::SIMD

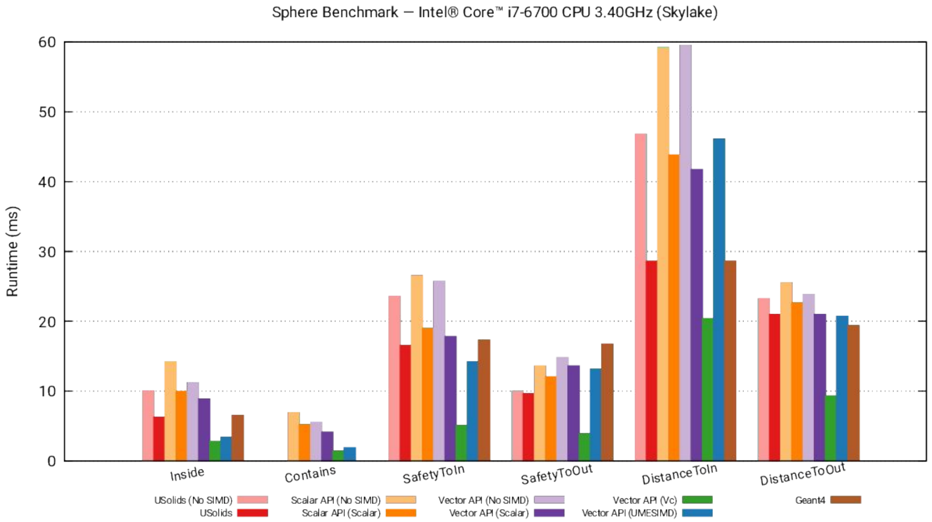
```
namespace vecCore {

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
```

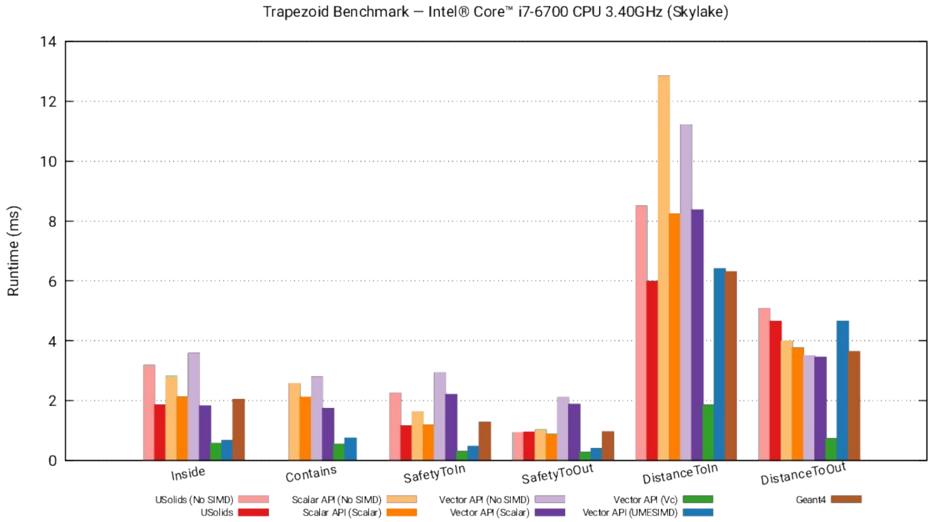
VecGeom Benchmarks on Intel® Skylake (AVX2)

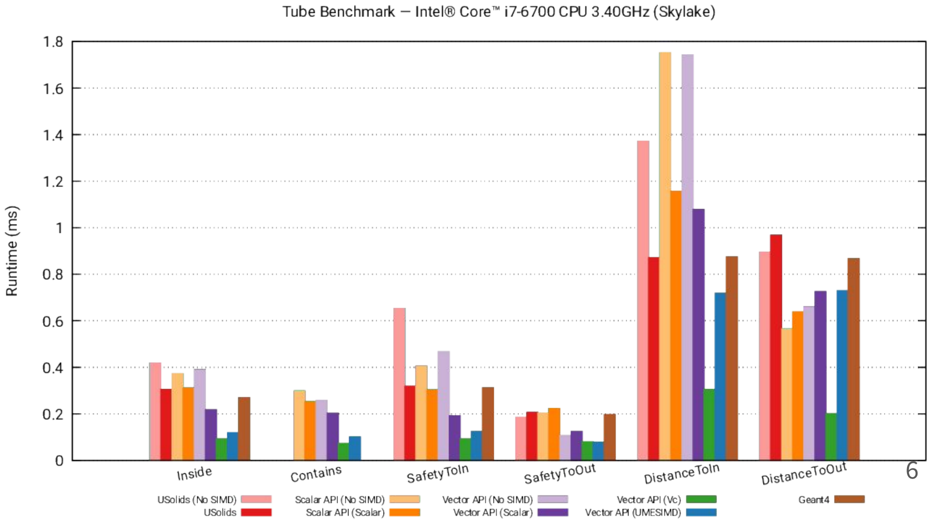
- Everything was compiled with Intel C/C++ compiler 16.0.2
- Implementations marked with “No SIMD” were compiled with “-no-vec”
- Other implementations were compiled with “-O3 -march=native”
- Vc gives best performance on Skylake, some scalar code gets auto-vectorized




Box Benchmark — Intel® Core™ i7-6700 CPU 3.40GHz (Skylake)
Runtime (ms)
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
Inside
Contains
SafetyToIn
SafetyToOut
DistanceToIn
DistanceToOut
USolids (No SIMD)
USolids
Scalar API (No SIMD)
Scalar API (Scalar)
Vector API (No SIMD)
Vector API (Scalar)
Vector API (Vc)
Vector API (UMESIMD)
Geant4

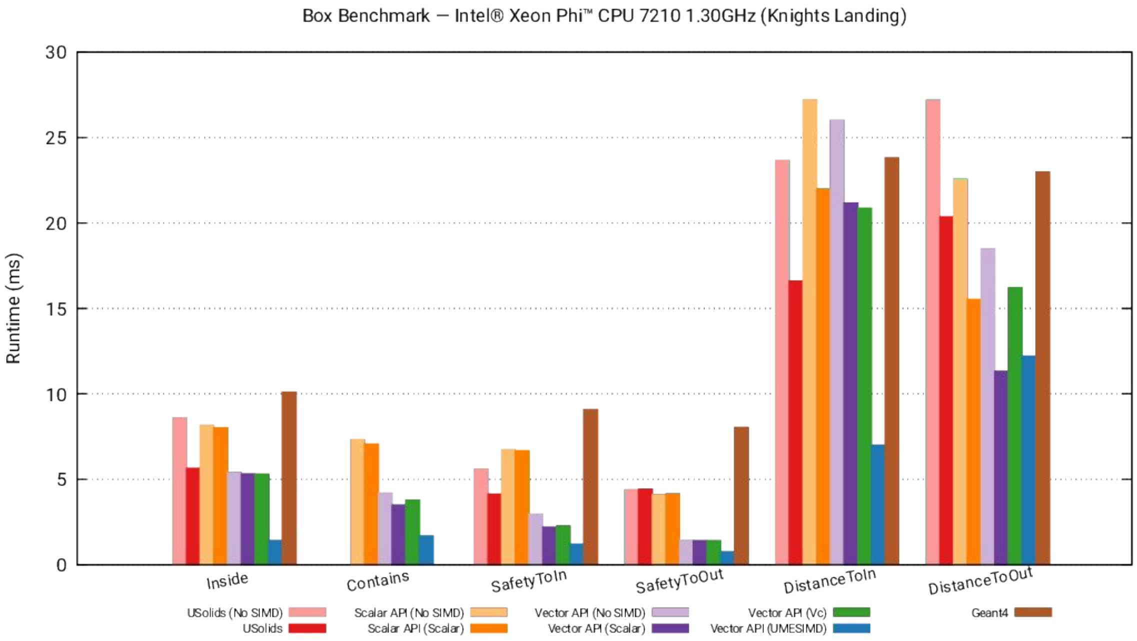

Sphere Benchmark — Intel® Core™ i7-6700 CPU 3.40GHz (Skylake)
Runtime (ms)
0
10
20
30
40
50
60
Inside
Contains
SafetyToIn
SafetyToOut
DistanceToIn
DistanceToOut
USolids (No SIMD)
USolids
Scalar API (No SIMD)
Scalar API (Scalar)
Vector API (No SIMD)
Vector API (Scalar)
Vector API (Vc)
Vector API (UMESIMD)
Geant4

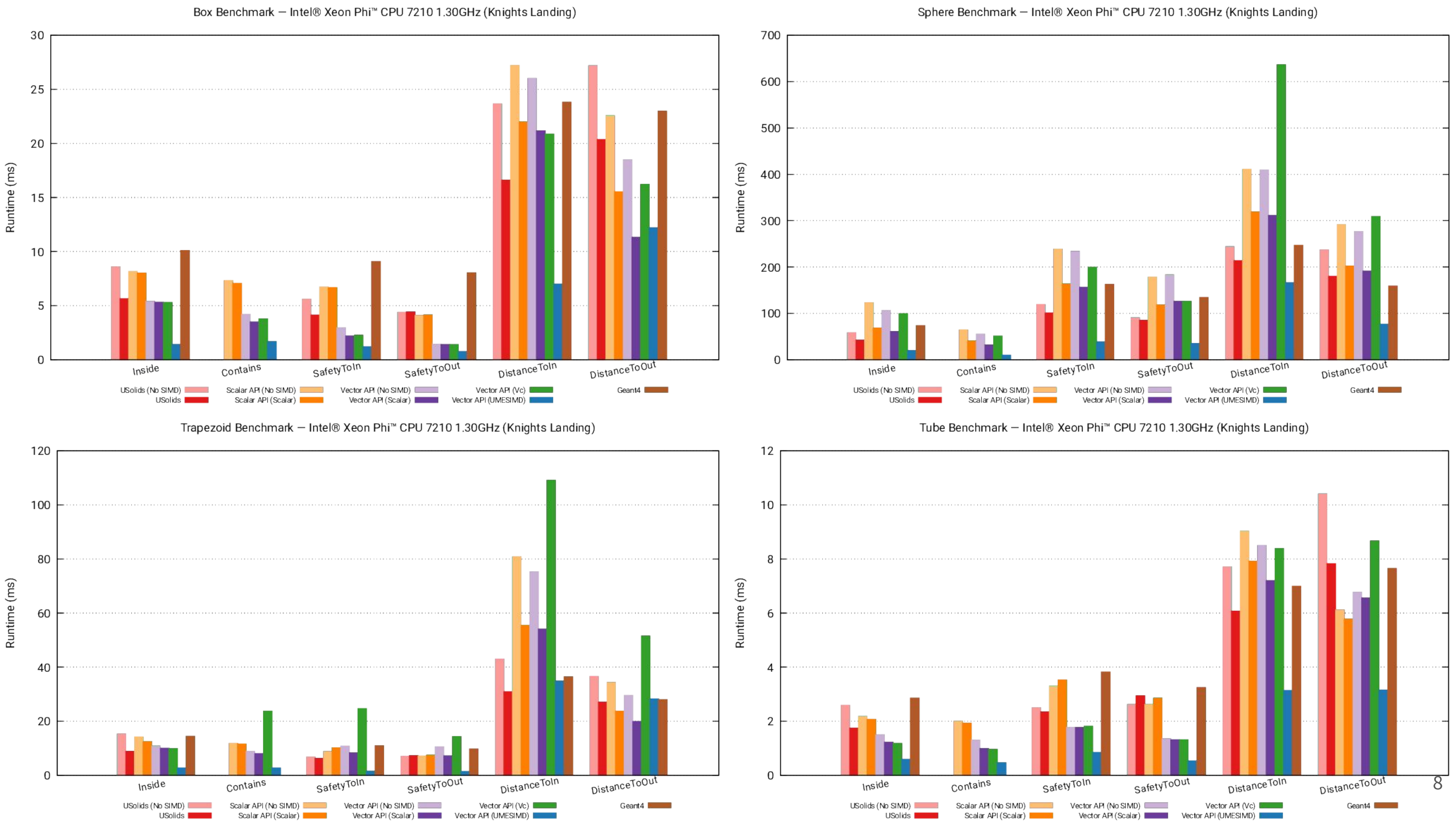

Trapezoid Benchmark — Intel® Core™ i7-6700 CPU 3.40GHz (Skylake)
Runtime (ms)
0
2
4
6
8
10
12
14
Inside
Contains
SafetyToIn
SafetyToOut
DistanceToIn
DistanceToOut
USolids (No SIMD)
USolids
Scalar API (No SIMD)
Scalar API (Scalar)
Vector API (No SIMD)
Vector API (Scalar)
Vector API (Vc)
Vector API (UMESIMD)
Geant4


Tube Benchmark — Intel® Core™ i7-6700 CPU 3.40GHz (Skylake)
Runtime (ms)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
Inside
Contains
SafetyToIn
SafetyToOut
DistanceToIn
DistanceToOut
USolids (No SIMD)
USolids
Scalar API (No SIMD)
Scalar API (Scalar)
Vector API (No SIMD)
Vector API (Scalar)
Vector API (Vc)
Vector API (UMESIMD)
Geant4

VecGeom Benchmarks on Intel® Xeon Phi™ (KNL)

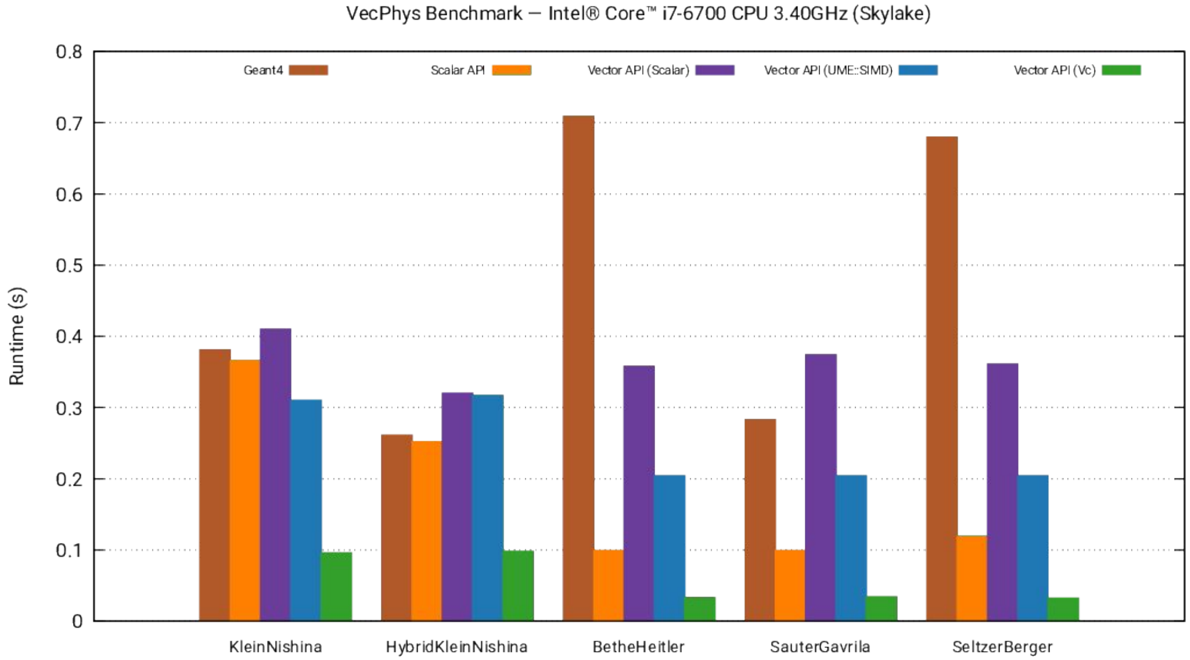
- Everything was compiled with Intel C/C++ compiler 16.0.3
- Used "-O3 -xMIC-AVX512"
- Contrary to AVX2 benchmarks on Skylake, UME::SIMD gives best performance on Knights Landing
- Scalar code under Vector API shows auto-vectorization in many cases




Box Benchmark — Intel® Xeon Phi™ CPU 7210 1.30GHz (Knights Landing)
Sphere Benchmark — Intel® Xeon Phi™ CPU 7210 1.30GHz (Knights Landing)
Trapezoid Benchmark — Intel® Xeon Phi™ CPU 7210 1.30GHz (Knights Landing)
Tube Benchmark — Intel® Xeon Phi™ CPU 7210 1.30GHz (Knights Landing)
Runtime (ms)
Inside
Contains
SafetyToIn
SafetyToOut
DistanceToIn
DistanceToOut
USolids (No SIMD)
USolids
Scalar API (No SIMD)
Scalar API (Scalar)
Vector API (No SIMD)
Vector API (Scalar)
Vector API (Vc)
Vector API (UMESIMD)
Geant4

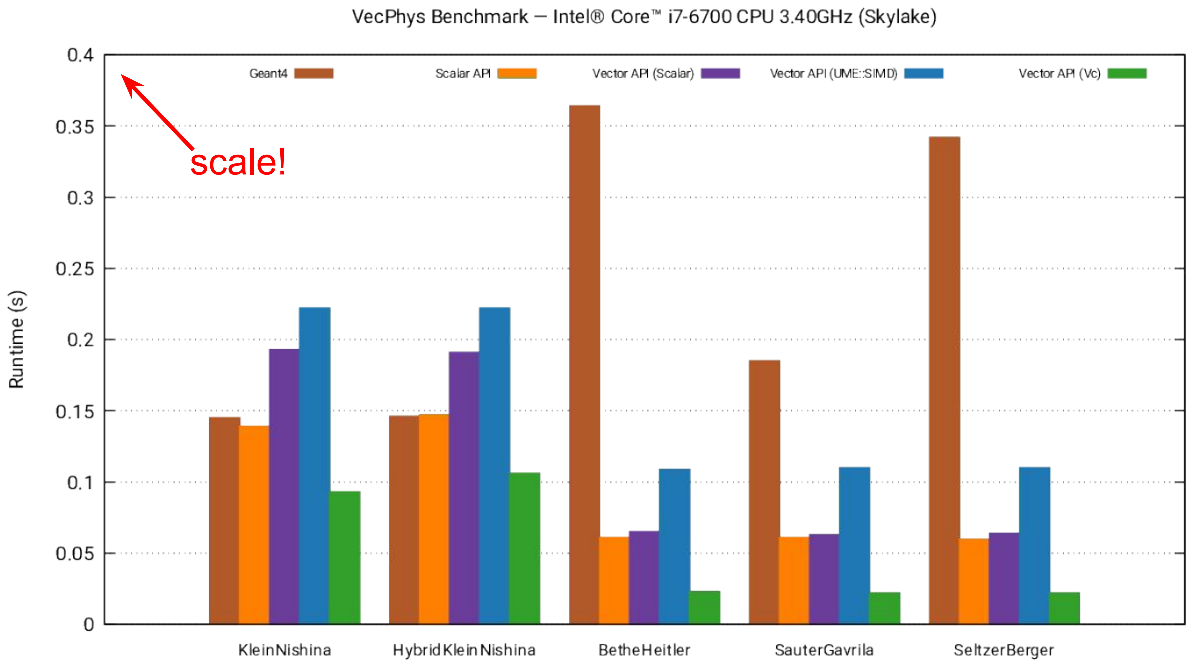
VecPhys Benchmarks (Electromagnetic Physics Models)

- Compiled with GCC Compiler
- Big speedup respective to Geant4 models, except for KleinNishina
- Vc backend offers best performance for physics models



VecPhys Benchmarks (Electromagnetic Physics Models)

- Compiled with Intel® C/C++ Compiler
- Big speedup respective to Geant4 models, except for KleinNishina
- Vc backend offers best performance for physics models
- ICC can auto-vectorize more code than GCC



Summary

- Auto-vectorization is a powerful tool and compilers are getting better at it.
  - PRO: Almost “free lunch” provided the code is free of “vectorization hazards”
  - CONS: There are still differences among compilers, operations, architectures.
- However explicit vectorization using specific libraries still gives significantly the best result (ex. Vc for AVX2 and UME::SIMD for AVX512)