The GeantV prototype on KNL

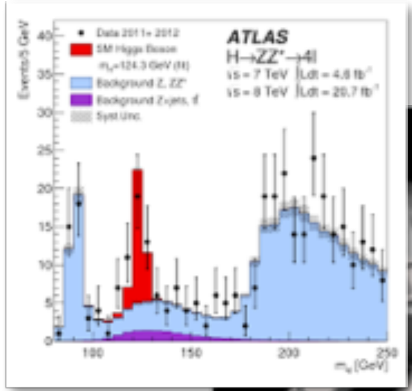
Federico Carminati, Andrei Gheata and Sofia Vallecorsa

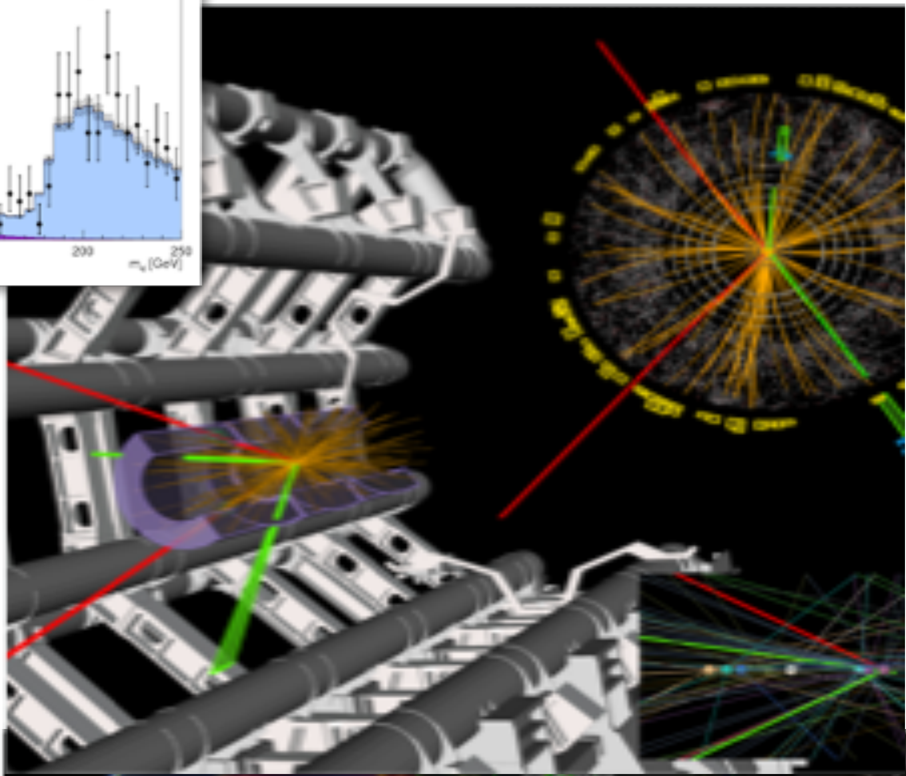
for the GeantV team

Outline

- Introduction
- (Digression on vectorization approach)
- Geometry benchmarks: vectorization and scalability
- Profiling + issues
- Particle transport improvement
  - Sub-node clustering + NUMA
  - Task based approach
  - Fast simulation using ML
- Improved GeantV scheduling
  - Preliminary features

The problem





- Detailed simulation of subatomic particles in detectors, essential for data analysis, detector design..
- Complex physics and geometry modeling
- Heavy computation requirements, massively CPU-bound

WLCG
Worldwide LHC Computing Grid

200 Computing centers in 20 countries: > 600k cores

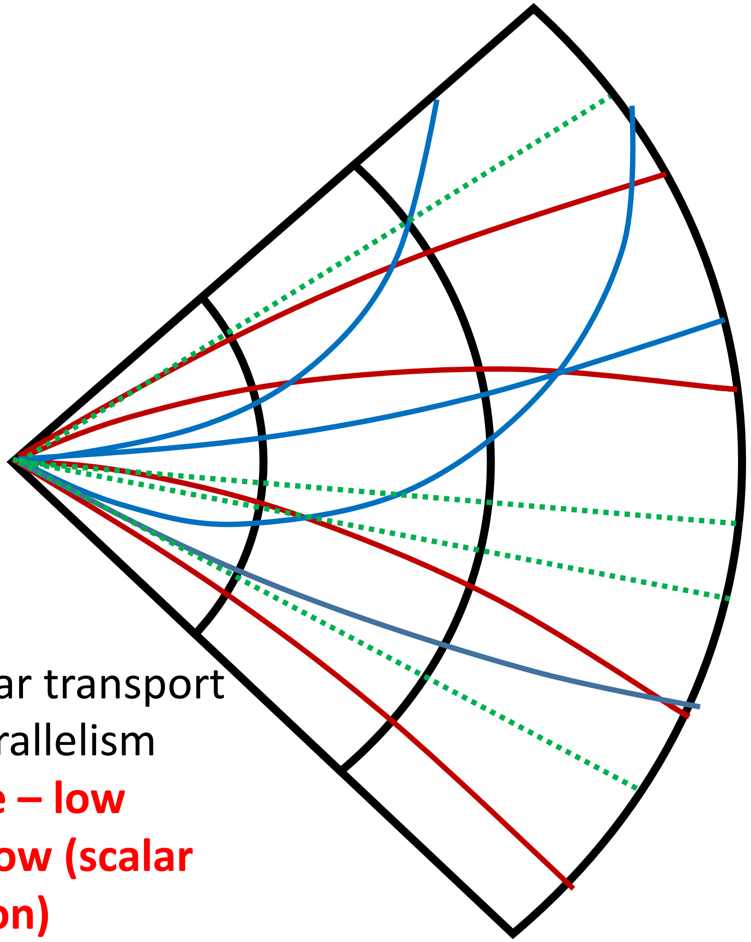
@CERN (20% WLCG): 65k processor cores ; 30PB disk + >35PB tape storage

More than 50% of WLCG power for simulations

GeantV – Adapting simulation to modern hardware

Classical simulation

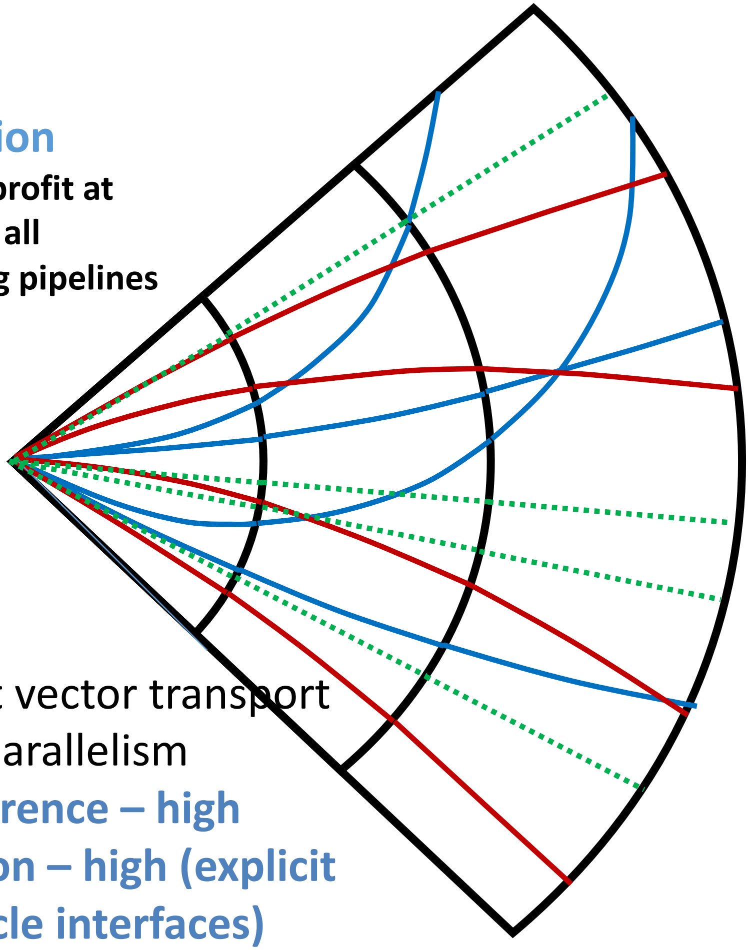
hard to approach the full machine potential



- Single event scalar transport
- Embarrassing parallelism
- **Cache coherence – low**
- **Vectorization – low (scalar auto-vectorization)**

GeantV simulation

needs to profit at best from all processing pipelines



- Multi-event vector transport
- Fine grain parallelism
- **Cache coherence – high**
- **Vectorization – high (explicit multi-particle interfaces)**

Tested hardware

Processor	Code name	#cores	Instruction set
Xeon E5-2695 v2 @ 2.40GHz	Ivy Bridge	2 x 12	AVX
Xeon E5-2630 v3 @ 2.4 GHz	Haswell	2 x 8	AVX2
Intel®Core i7 6700 @ 3.4 GHz	Skylake	4	AVX2
Xeon Phi™ 7210 @ 1.3GHz	KNL	64	AVX 512

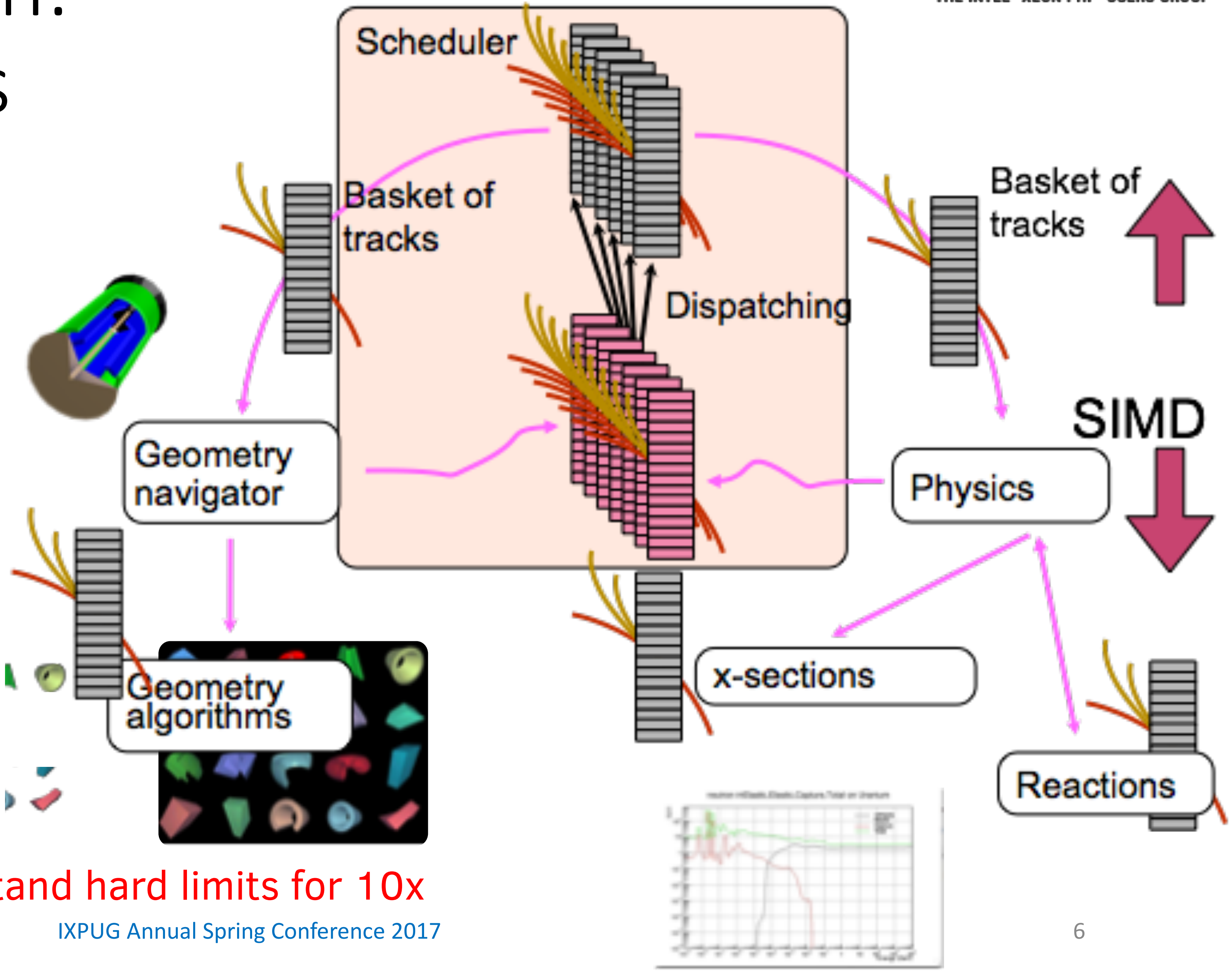
Compilers used: different versions of **icc**, **gcc** and **clang**

KNL memory configurations:

- quadrant + flat mode - tested
- MCDRAM in cache mode – ongoing tests
- “hot” data on MCDRAM (memkind/hbwmalloc) – ongoing tests

GeantV approach: boosting vectors

- Transport particles in vectors ("baskets")
  - Filter by geometry volume or physics process
- Redesign library and workflow to target fine grain parallelism
- Use an abstraction for vector types and their operations to achieve portable vectorization



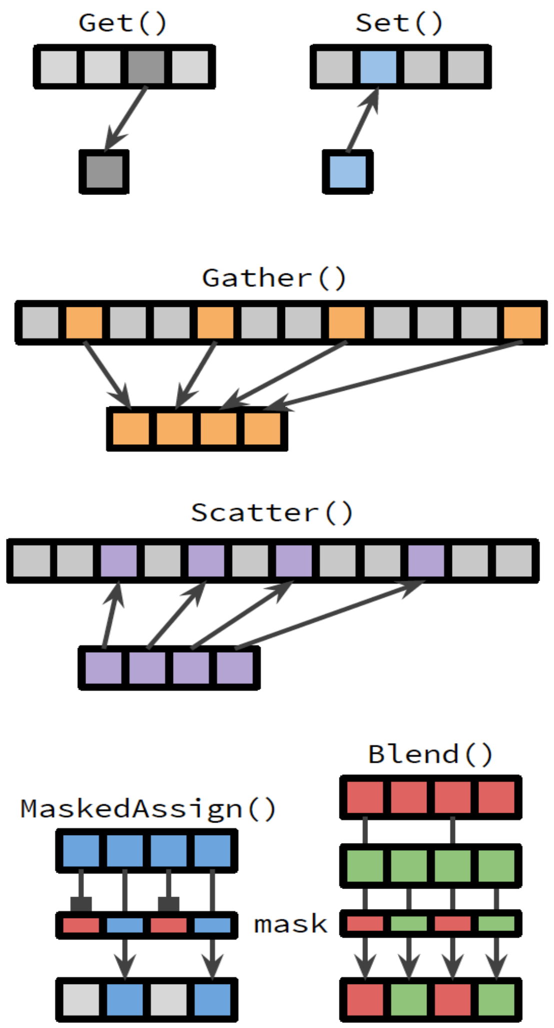
Aim for a 3x-5x faster code, understand hard limits for 10x

Why a vectorization abstraction?

- 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
  - Portability is an issue, exposure to users as well…
- Vectorization libraries do not always work well across all architectures
  - e.g. UME::SIMD uses scalar emulation for AVX2 (not as good as KNL)
- Still need portable solution when no library is available
- Brief digression on the subject following next

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

```cpp
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

```cpp
#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

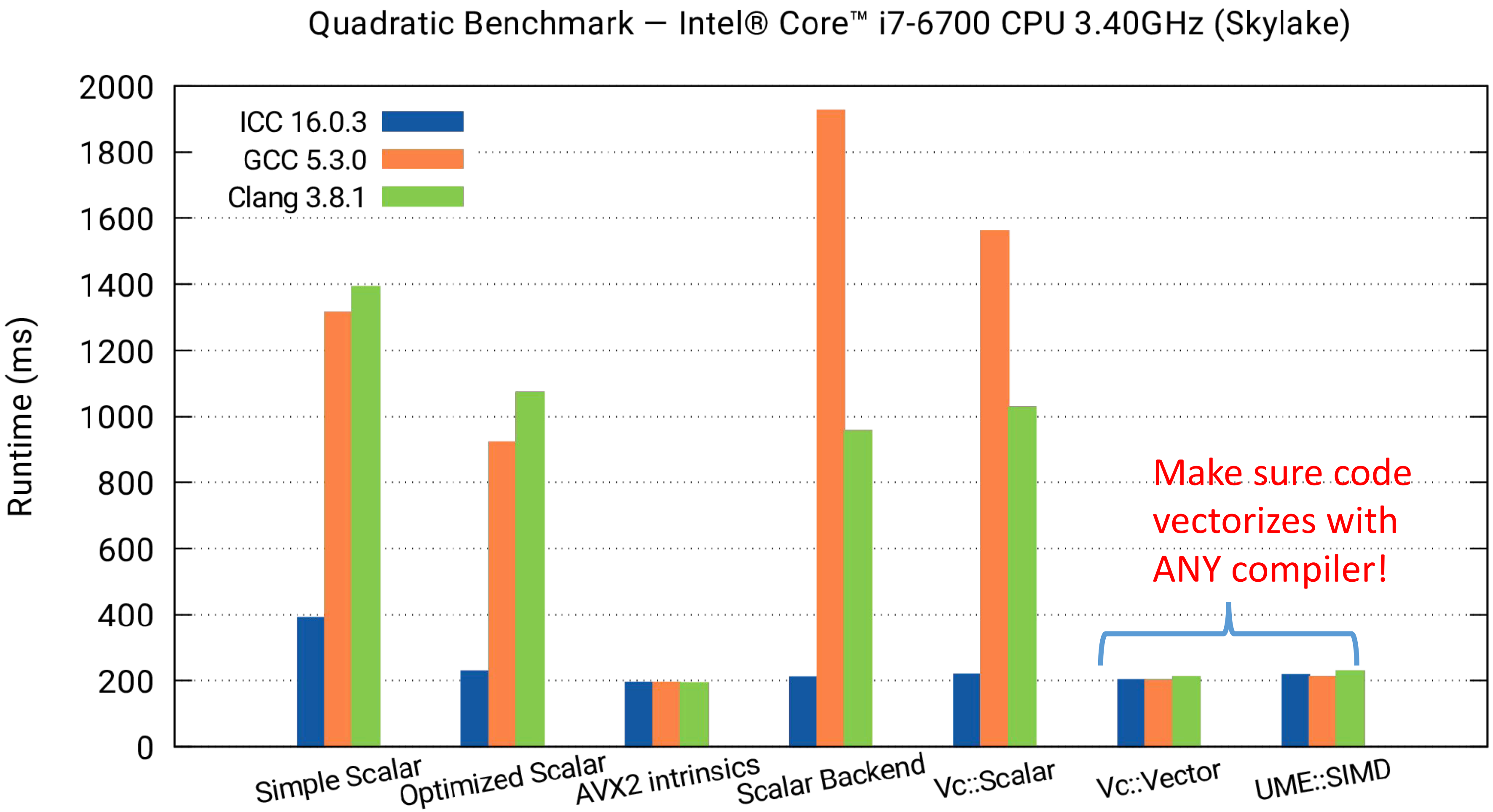
```
void QuadSolveAVX(const float *__restrict__ a, const float *__restrict__ b, const float *__restrict__ c,
                  float *__restrict__ x1, float *__restrict__ x2, int *__restrict__ roots)
{
  __m256 one    = _mm256_set1_ps(1.0f);
  __m256 va     = _mm256_load_ps(a);
  __m256 vb     = _mm256_load_ps(b);
  __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

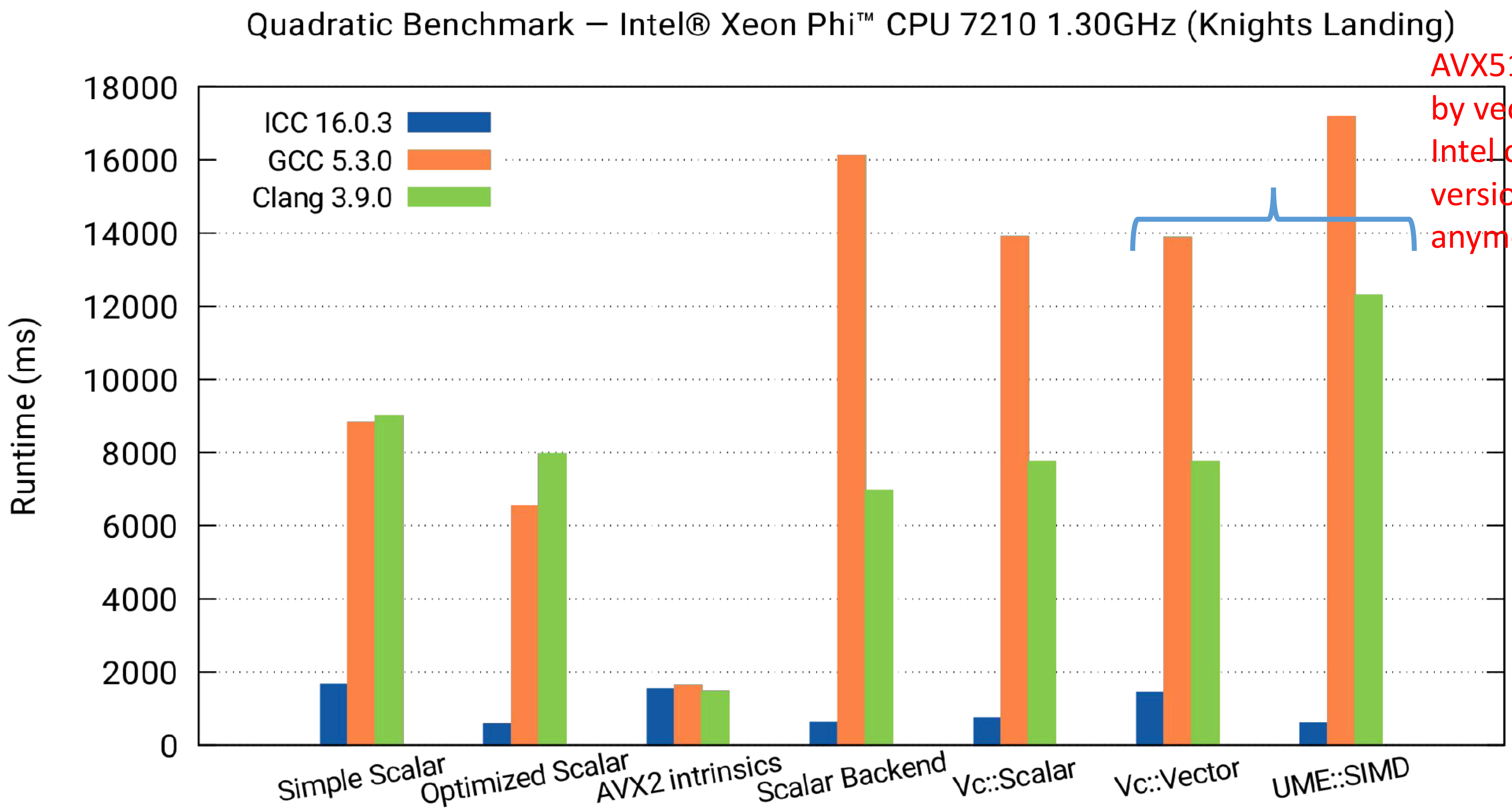
VecCore API Implementation

```
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 Skylake



Performance Comparison on Intel Xeon Phi



AVX512 not well supported by vector libraries with non-Intel compilers in the initial version – not the case anymore

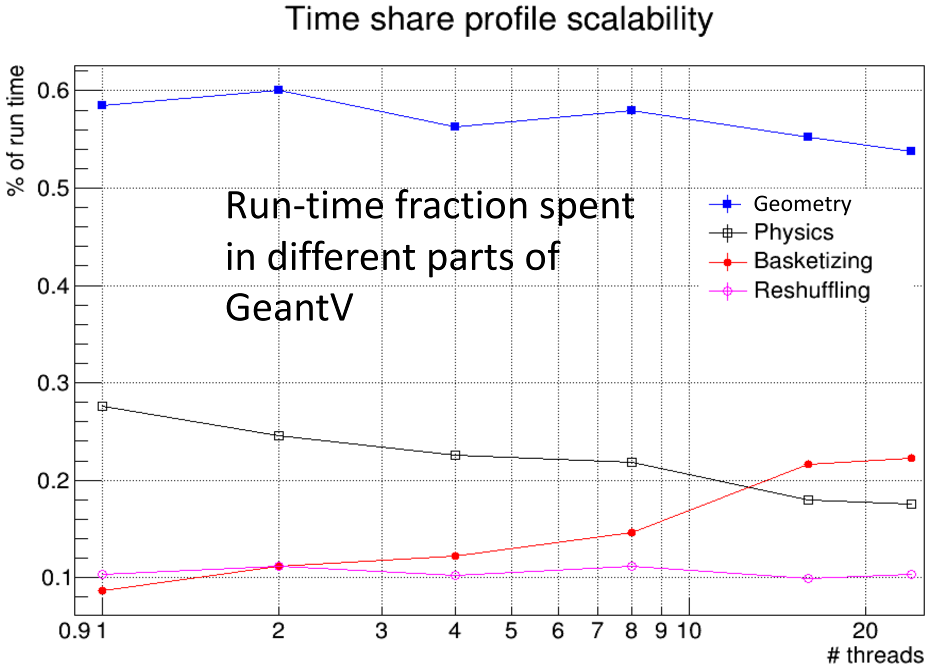
A real example: Electromagnetic Physics Models

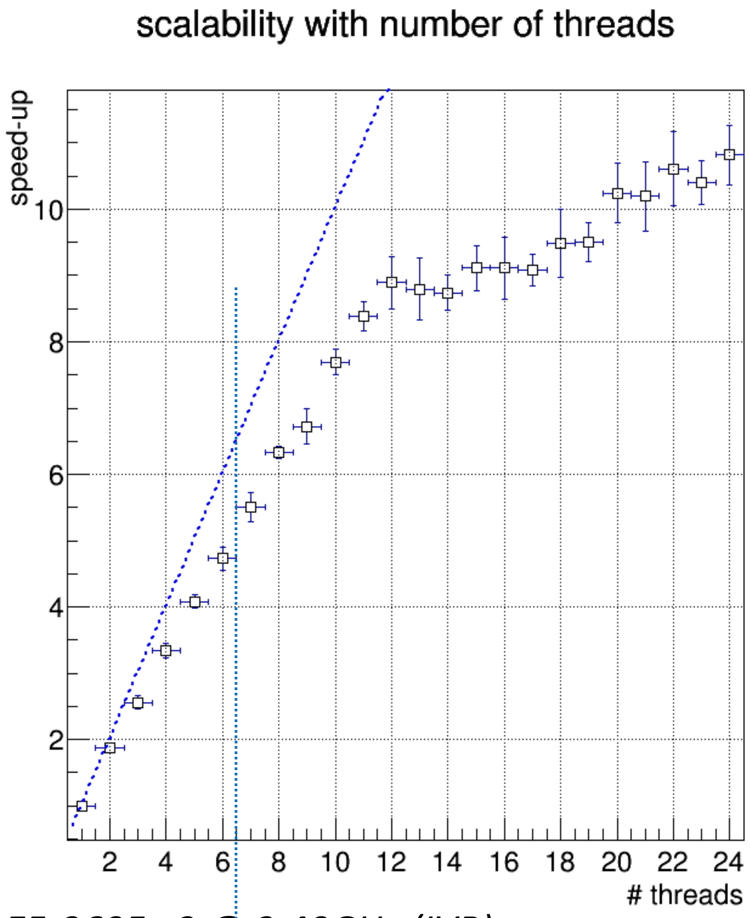
Ionization Model Final State

```
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));
}
```

Vectors and the challenges

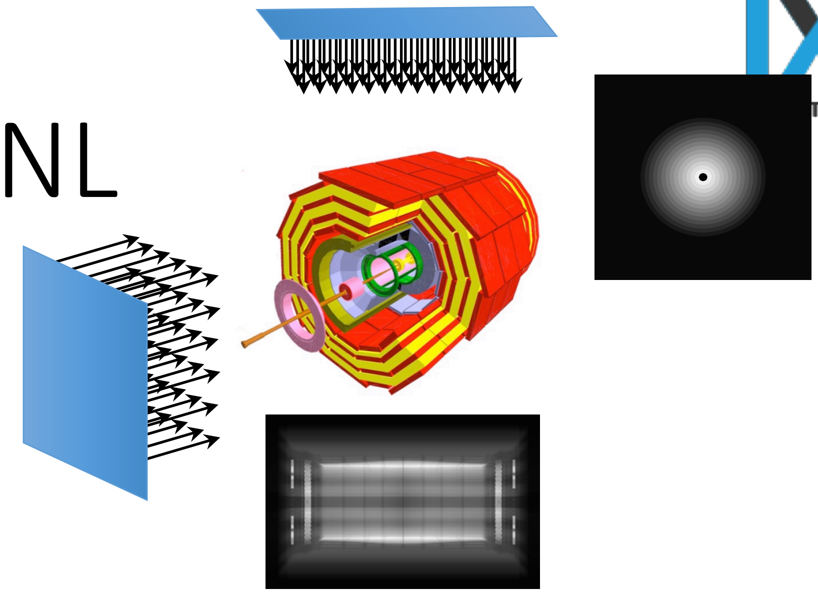
- Gather/reshuffle data into SOA, then into SIMD registers
- No free lunch: need to keep data gathering overheads < vector gains





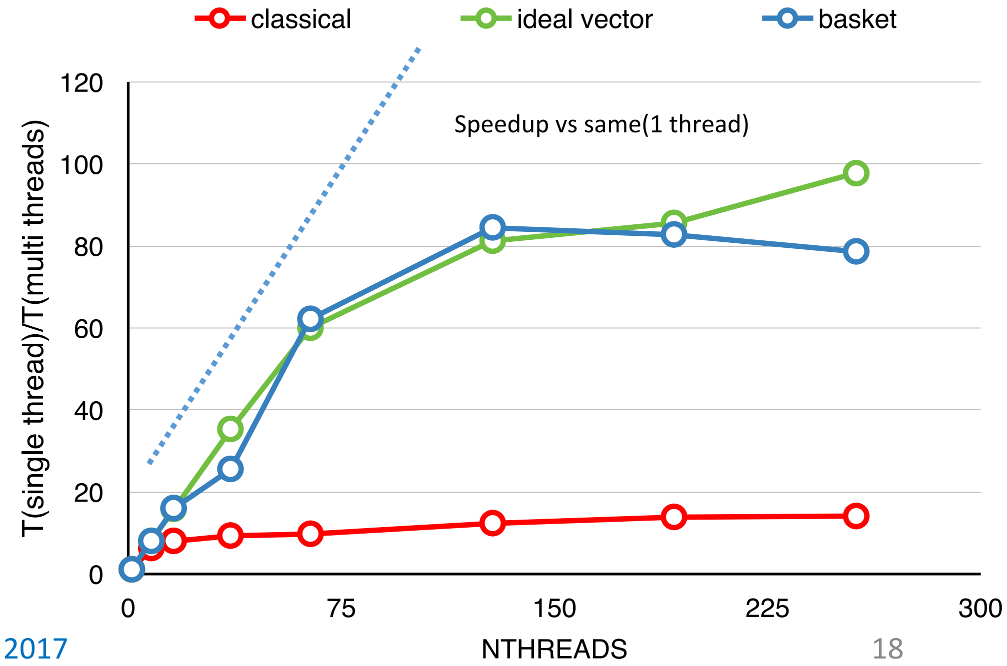
24-core dual socket E5-2695 v2 @ 2.40GHz (IVB).

Geometry navigation on KNL



- "X-ray" scan of detector volumes
    - Trace a grid of virtual rays through geometry
- Simplified geometry emulating a tracker detector
- Compare GeantV basket approach to
    - Classical scalar navigation (ROOT)
    - An ideal "vector" case (no basketizing overheads)
- AVX512 vectorization enforced by API (UME:SIMD backend)
- ~100x speedup for the ideal and basket versions

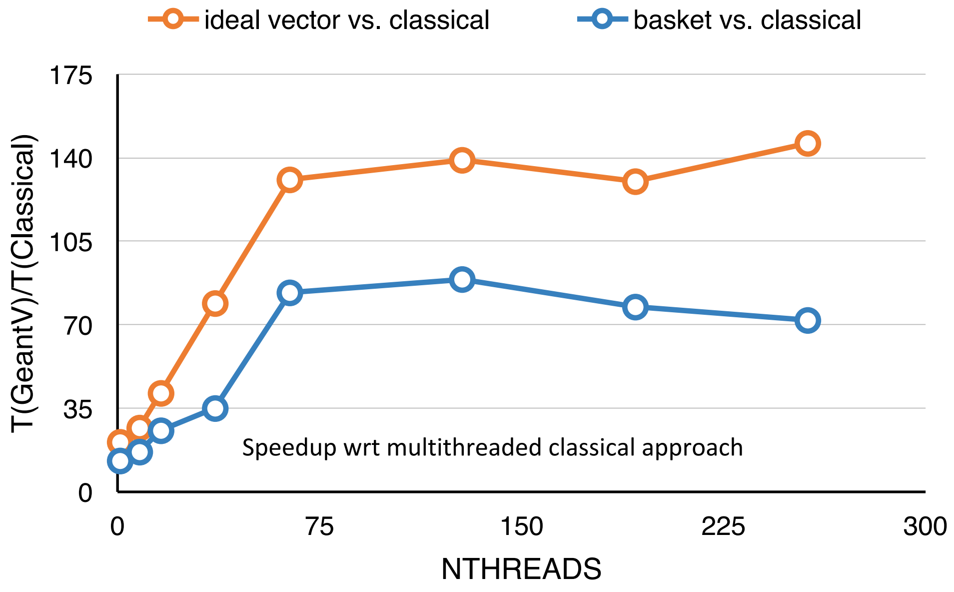
Intel® Xeon Phi™ CPU 7210 @ 1.30GHz

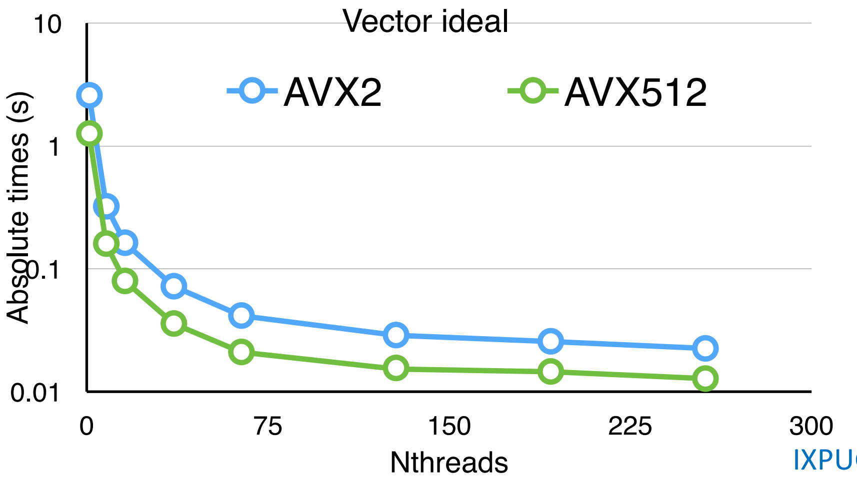


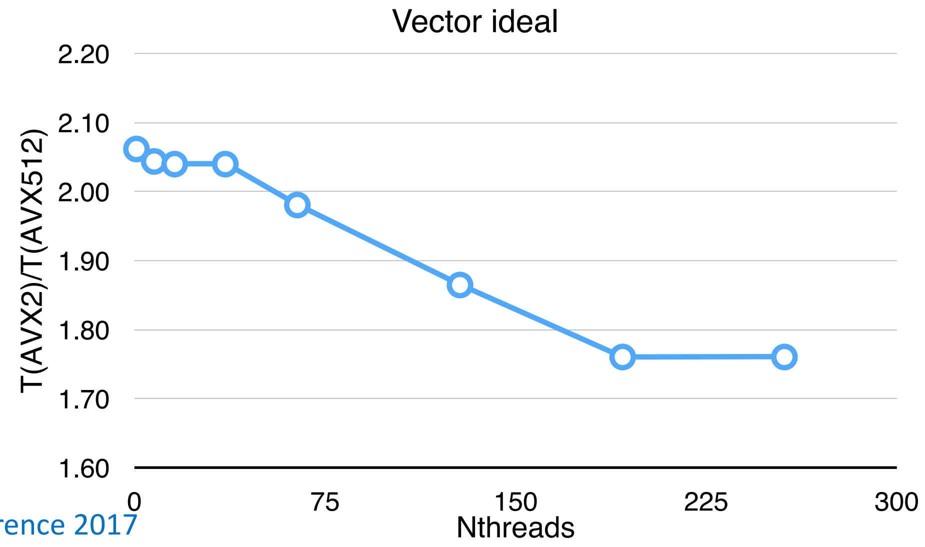
Performance

GeantV gives excellent benefits with respect to ROOT in terms of speedup

- High vectorization intensity achieved for both ideal and basketized cases
  - AVX-512 brings an extra factor of ~2 to our benchmark





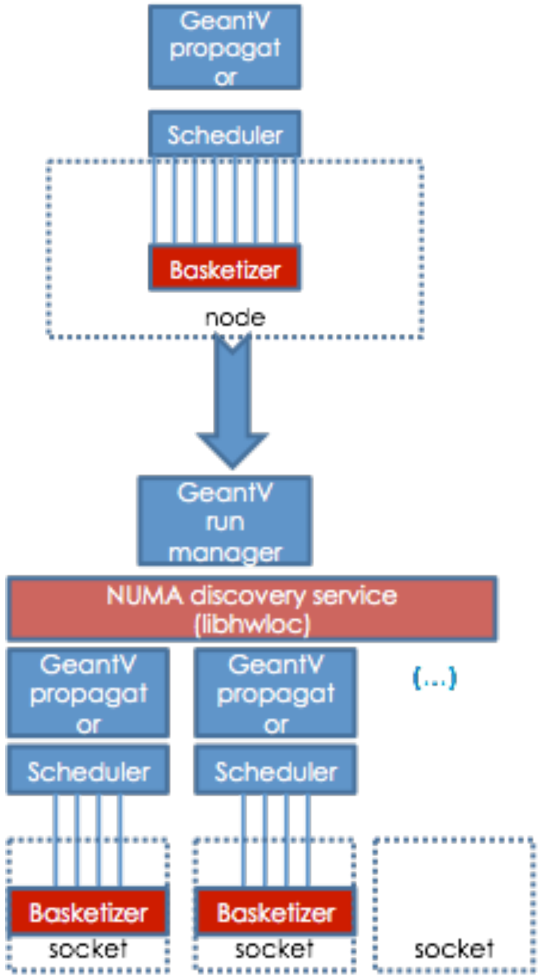


Improving the performance

- Sub-node clustering with multiple propagators
  - Improve data/processing locality and reduce contention
  - Improved memory management in basketizing procedure (NUMA awareness)
- TBB-based task based version
- Fast simulation using ML/DL

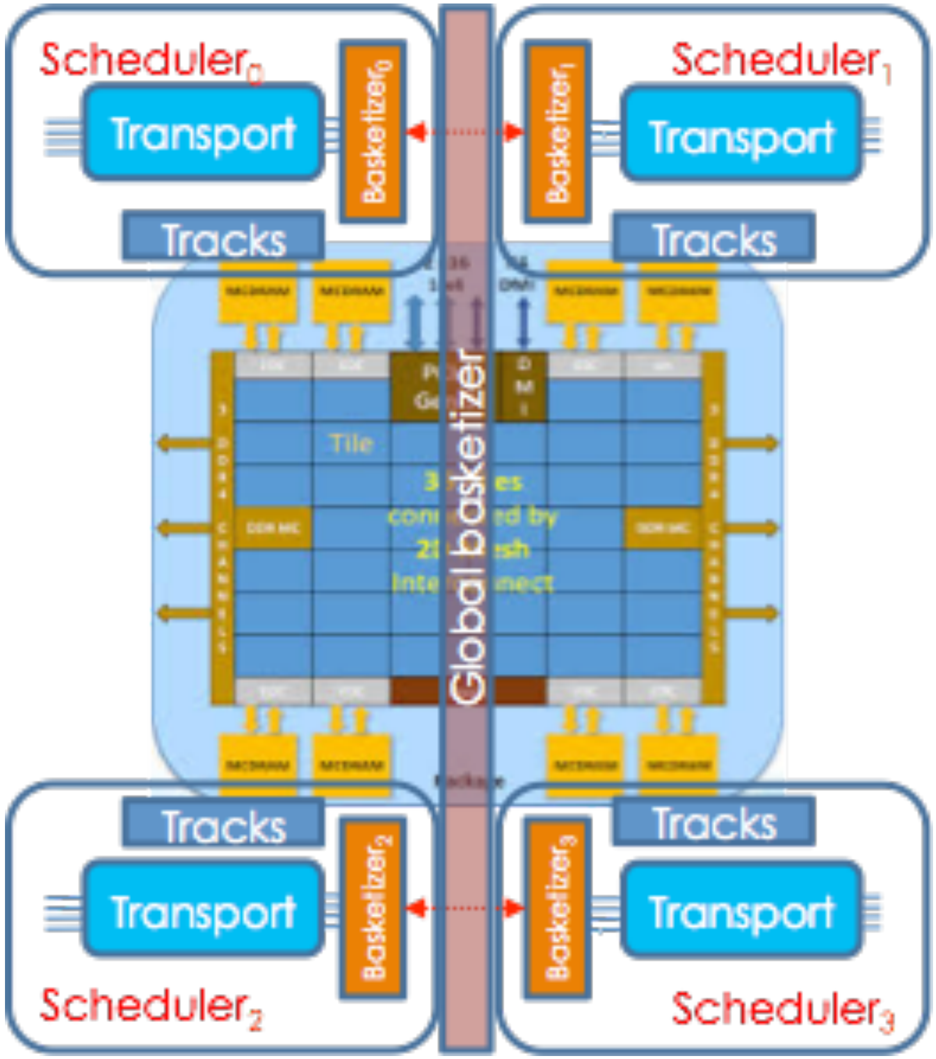
Sub-node clustering

- Known scalability issues of full GeantV due to synchronization in re-basketizing
- New approach deploying several propagators clustering resources at sub-node level
- Objectives: improved scalability at the scale of KNL and beyond, address both many-node and multi-socket (HPC) modes + non-homogenous resources
- Implemented recently - being tested on KNL



NUMA awareness

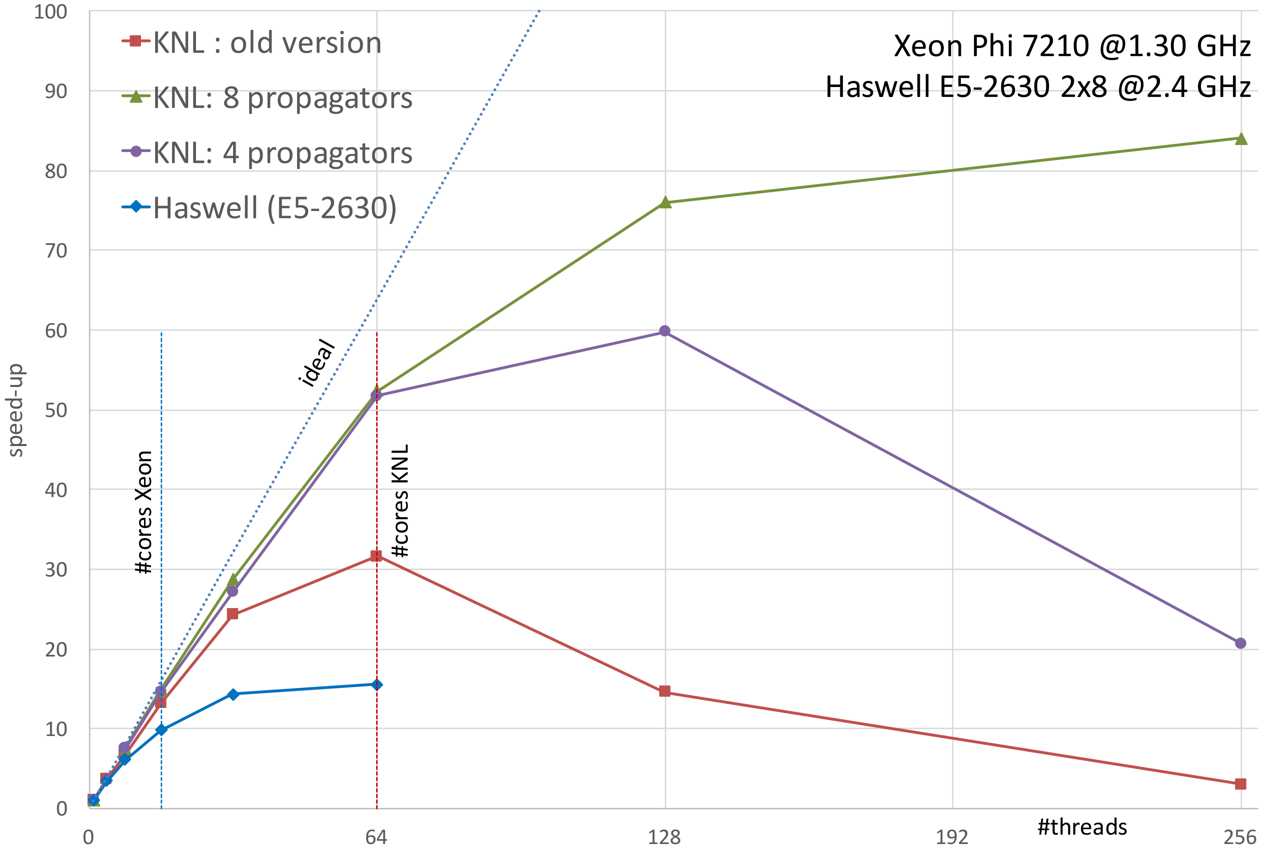
- Replicate schedulers on NUMA clusters
  - One basketizer per NUMA node
  - libhwloc to detect topology
  - Use pinning/NUMA allocators to increase locality
- Multi-propagator mode running one/more clusters per quadrant
  - Loose communication between NUMA nodes at basketizing step
  - Implemented, currently being tested



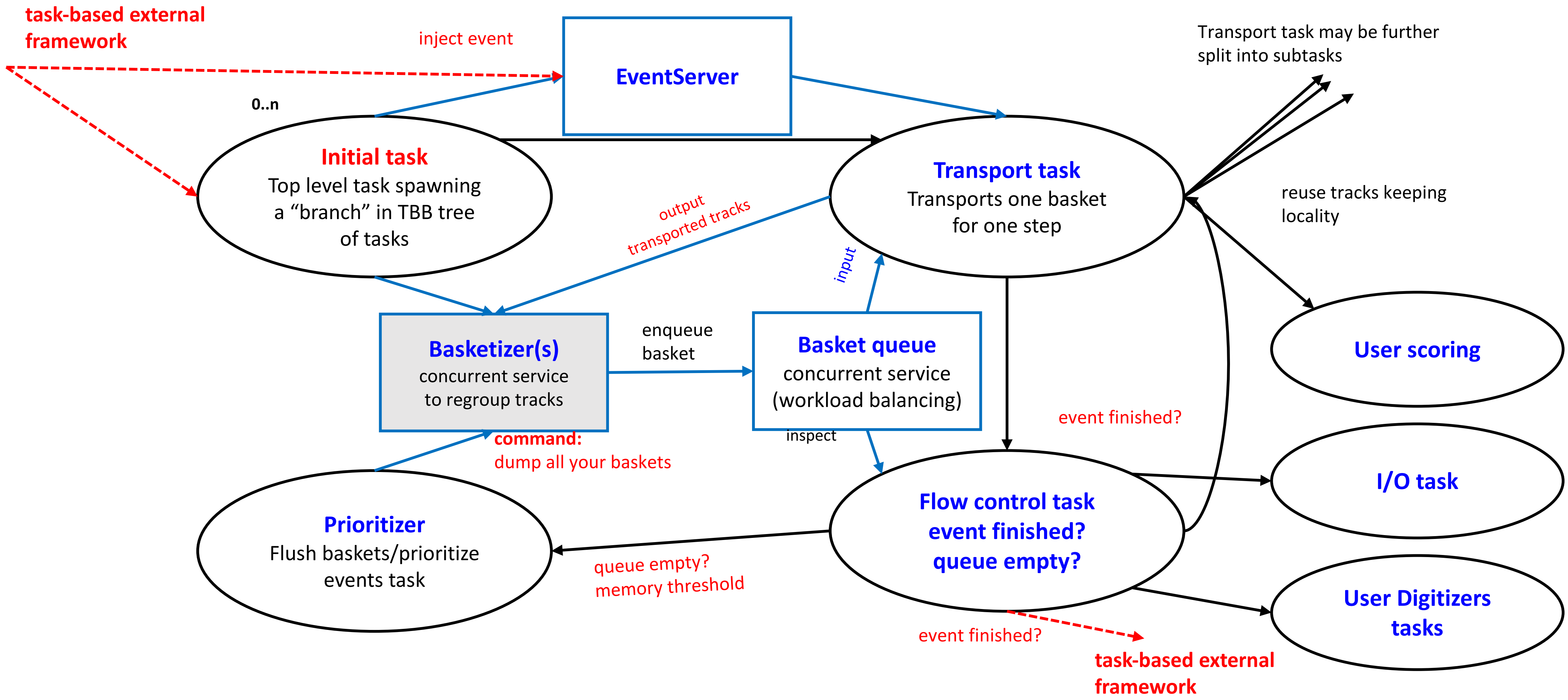
Multi-propagators prototype

- Full track transport and basketization procedure
- Simplified calorimeter
- Tabulated physics (EM processes + various materials)
- Scalability gets better by increasing number of propagators
- Not final results, still fixing/optimizing

Good scalability up to the number of physical cores

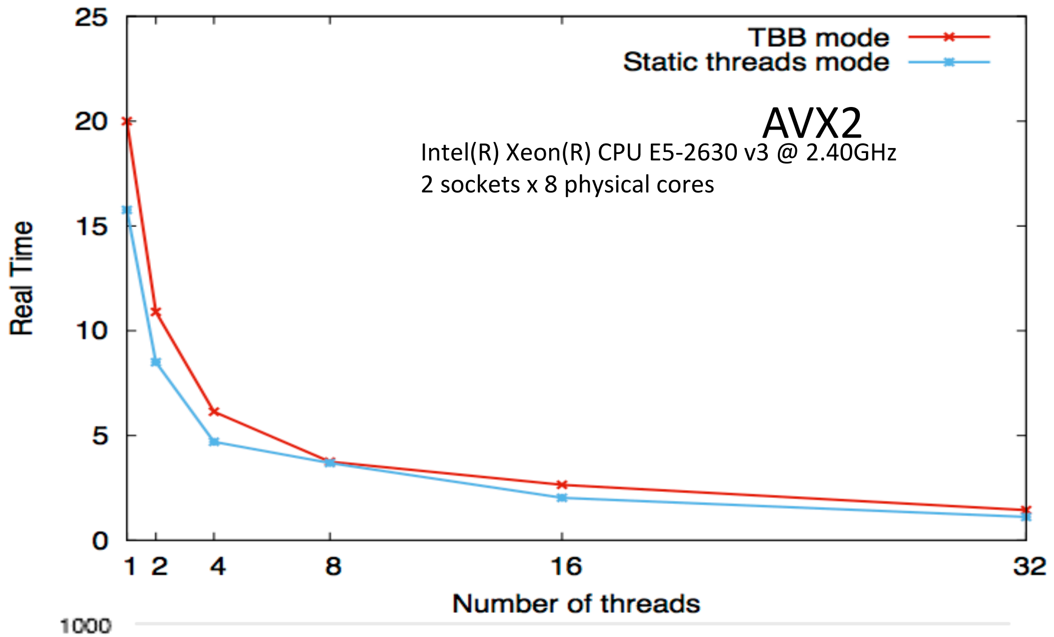


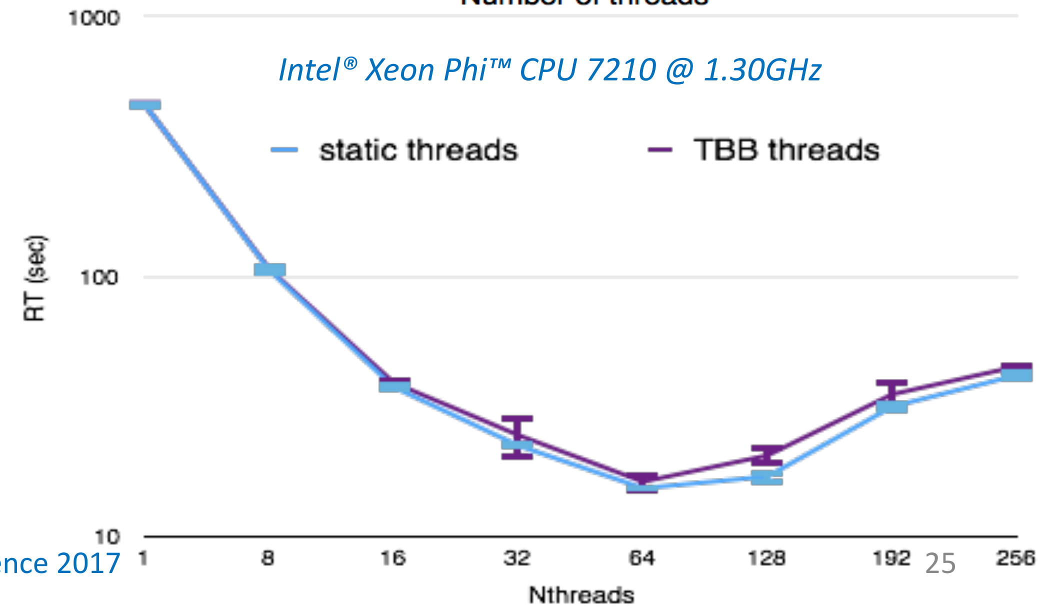
Task based GeantV



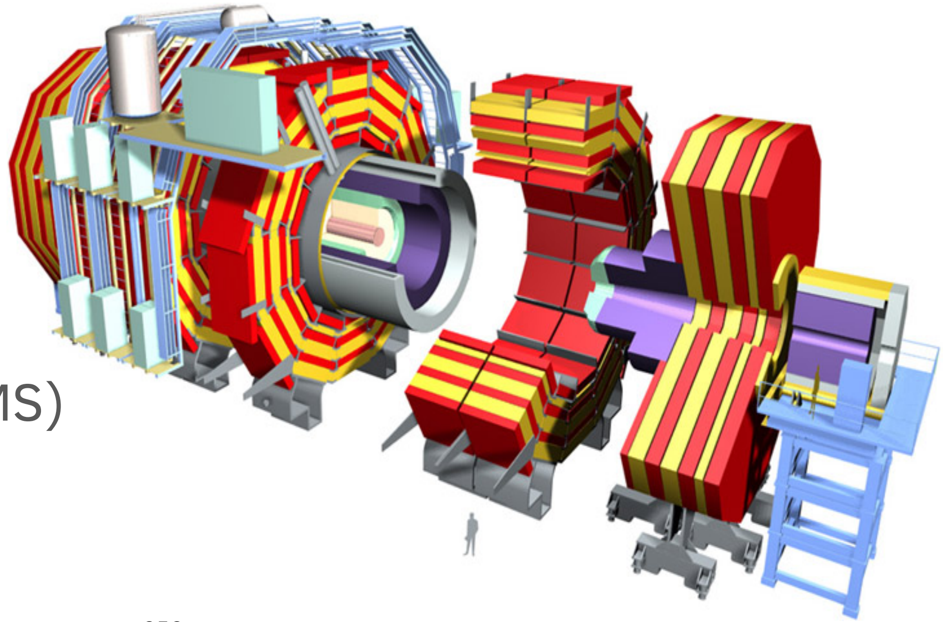
TBB tasks: preliminary results

- A first implementation of TBB task-based approach on the full track transport prototype
  - TBB preferred over OpenMP tasks due to requirements for integration with user code and other frameworks
- Some overheads on Haswell/AVX2, not so obvious on KNL/AVX512
  - Re-entrance of tasks compared to the static approach

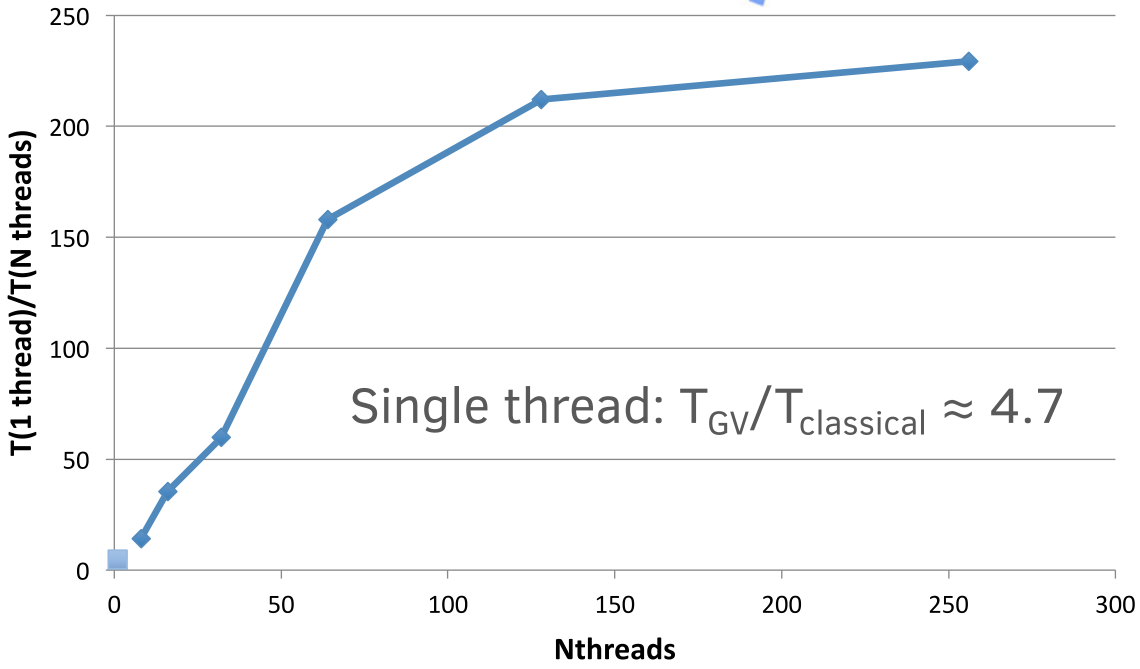




The full prototype



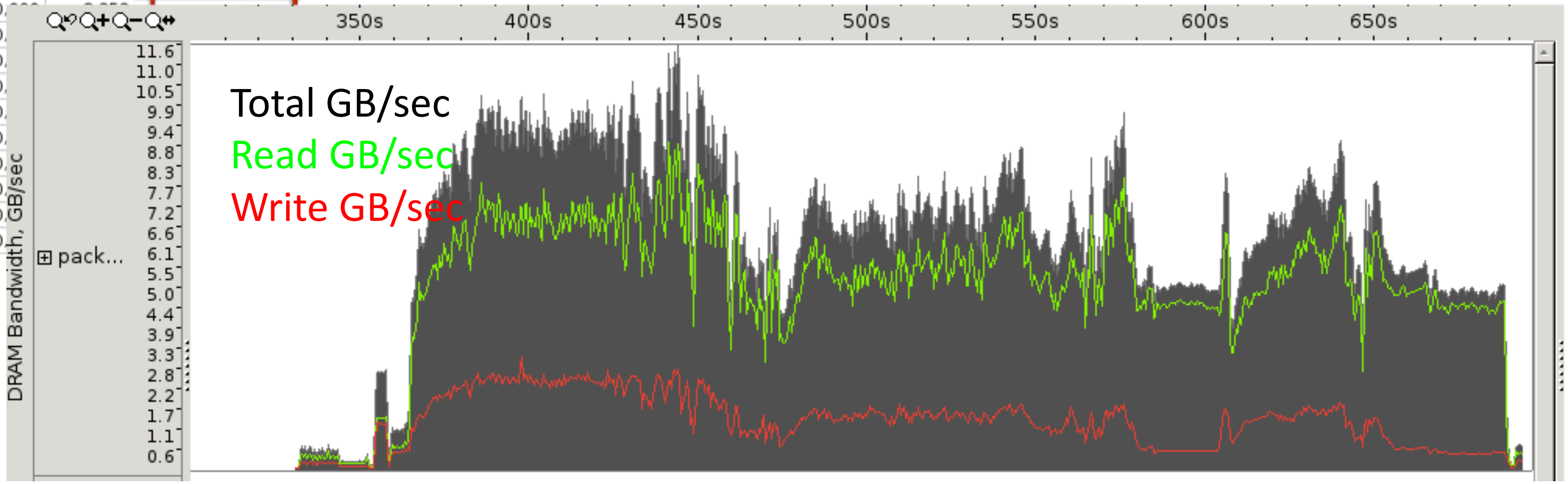
- Exercise at the scale of LHC experiments (CMS)
  - Full geometry + uniform magnetic field
  - Tabulated physics, fixed 1MeV energy threshold
- Full track transport and basketization procedure
- First results on speed-up (comparison to classical approach single-thread)



Full prototype performance on KNL

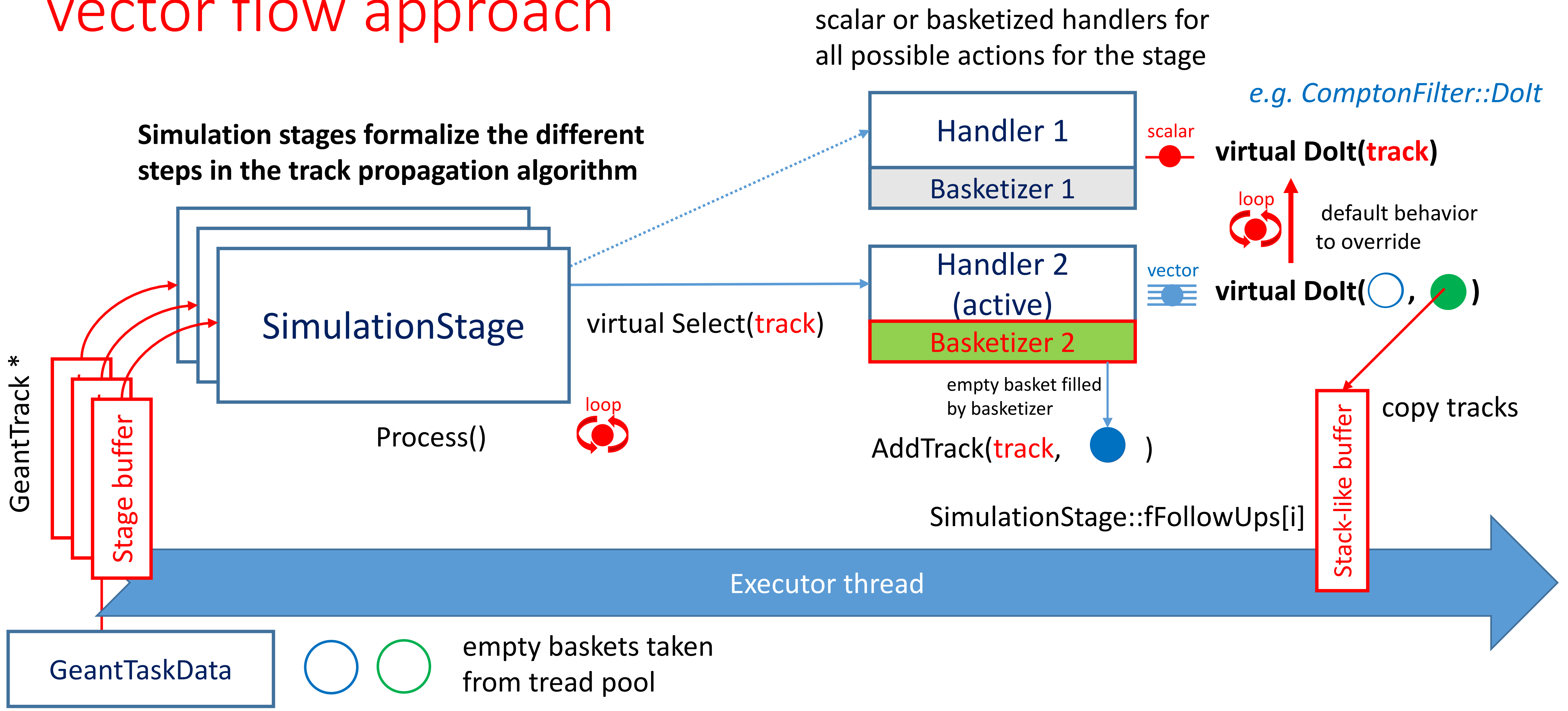
Function / Call Stack	Clockticks ▼	Instructions Retired	CPI Rate	VPU Utilization
Geant::cxx::GeantBasketMgr::GarbageCollect	1,007,992,700...	177,079,500,000	5.692	0.0%
vecgeom::cxx::BoxImplementation::IntersectCached	765,502,400,0...	304,720,000,000	2.512	75.1%
Geant::cxx::GeantBasketMgr::IsActive	572,096,200,0...	75,033,400,000	7.625	0.0%
vecgeom::cxx::ABBoxImplementation::ABBoxContai	548,169,700,0...	270,182,900,000	2.029	67.1%
vecgeom::cxx::Transformation3D::MultiplyFromRight	510,380,000,0...	284,544,000,000	1.794	0.0%
__do_softirq	465,290,800,0...	38,340,900,000	12.136	49.1%
vecgeom::cxx::Transformation3D::DoRotation<(int)-1	375,128,000,0...	244,116,600,000	1.537	0.1%
UME::SIMD::SIMDVec_f<float, (unsigned int)8>::~SI	308,042,800,0...	201,848,400,000	1.526	99.8%
UME::SIMD::SIMDVecFloatInterface<UME::SIMD::SIM	281,847,800,0...	198,386,500,000	1.421	99.6%
vecgeom::cxx::Vector3D<double>::operator[]	273,231,400,0...	131,582,100,000	2.077	0.8%
vecgeom::cxx::HybridNavigator<(bool)0>::GetHitCa	256,391,200,0...	109,603,000,000	2.339	47.1%
__memcpy_ssse3_back	244,886,200,0...	79,907,100,000	3.065	100.0%
UME::SIMD::SIMDVecFloatInterface<UME::SIMD::SIM	241,406,100,0...	162,740,500,000	1.483	98.8%
Geant::cxx::ScalarNavInterfaceVGM::NavFindNextBo	226,302,700,0...	54,250,300,000	4.171	6.1%
vecgeom::cxx::TSimpleABBoxLevelLocator<(bool)0>	220,662,000,0...	115,125,400,000	1.917	36.0%
UME::SIMD::SIMDVecBaseInterface<UME::SIMD::SIM	216,894,600,0...	166,778,300,000	1.300	99.2%
vecgeom::cxx::Vector3D<double>::operator[]	190,830,900,0...	100,120,800,000	1.906	0.1%
vecgeom::cxx::ABBoxImplementation::ABBoxSafety	187,236,400,0...	83,105,100,		
Geant::cxx::GeantTrack_v::AddTrackSync	182,943,800,0...	86,737,300,		
vecgeom::cxx::NavigationState::CopyTo	181,840,100,0...	55,740,100,		
Geant::cxx::WorkloadManager::TransportTracks	176,066,800,0...	51,192,700,		
vecgeom::cxx::NavigationState::Top	160,837,300,0...	63,872,900,		
UME::SIMD::SIMDVecMask<(unsigned int)8>::~SIMD	155,048,400,0...	95,629,300,		
UME::SIMD::SIMDMaskBaseInterface<UME::SIMD::SI	148,993,000,0...	72,455,500,		
Geant::cxx::GeantScheduler::AddTracks	141,200,800,0...	33,819,500,		
Geant::cxx::GeantTrack_v::PropagateTracks	137,473,700,0...	49,199,800,		
vecCore::MaskingImplementation<UME::SIMD::SIMD	134,274,400,0...	94,216,200,		

- Overall we fill VPUs reasonably well (for function calls that are supposed to vectorize)
- Memory access analysis shows we are not bandwidth bound: most of the code runs as “low utilisation”(<12 GB/sec)

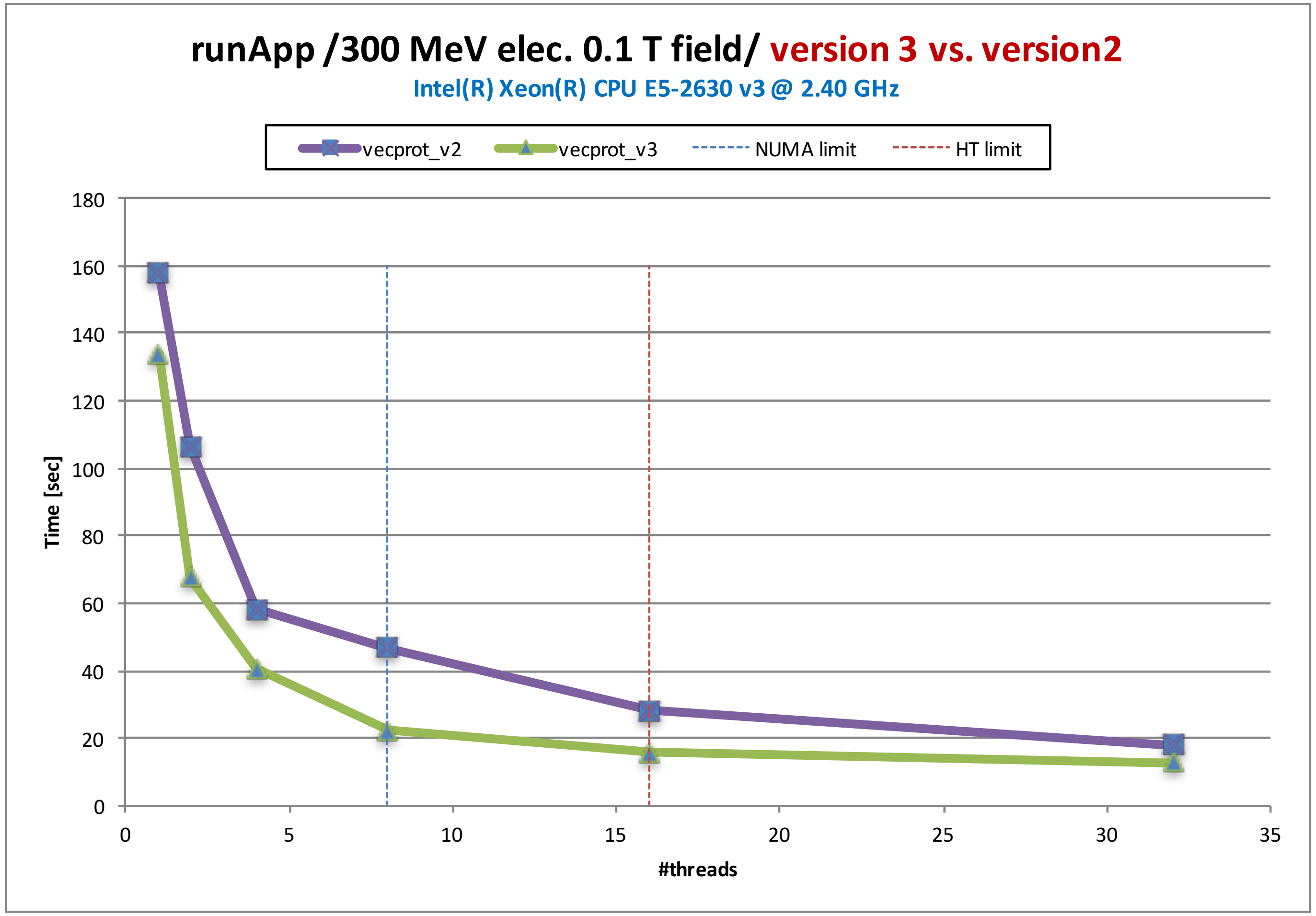


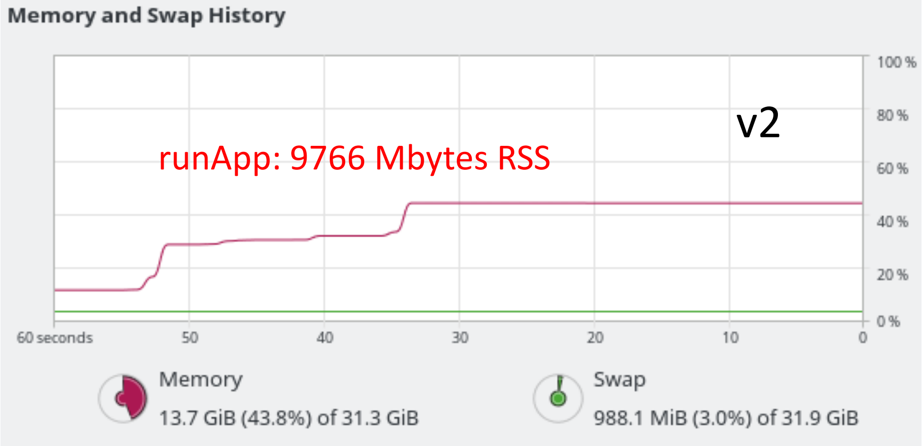
- A lot of copying to regroup SIMD vectors -> cache misses + contention, high memory usage

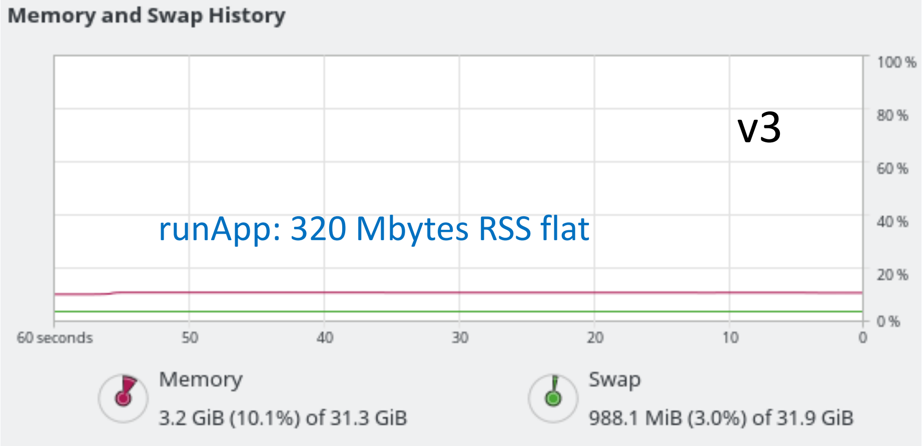
GeantV version 3: A generic vector flow approach



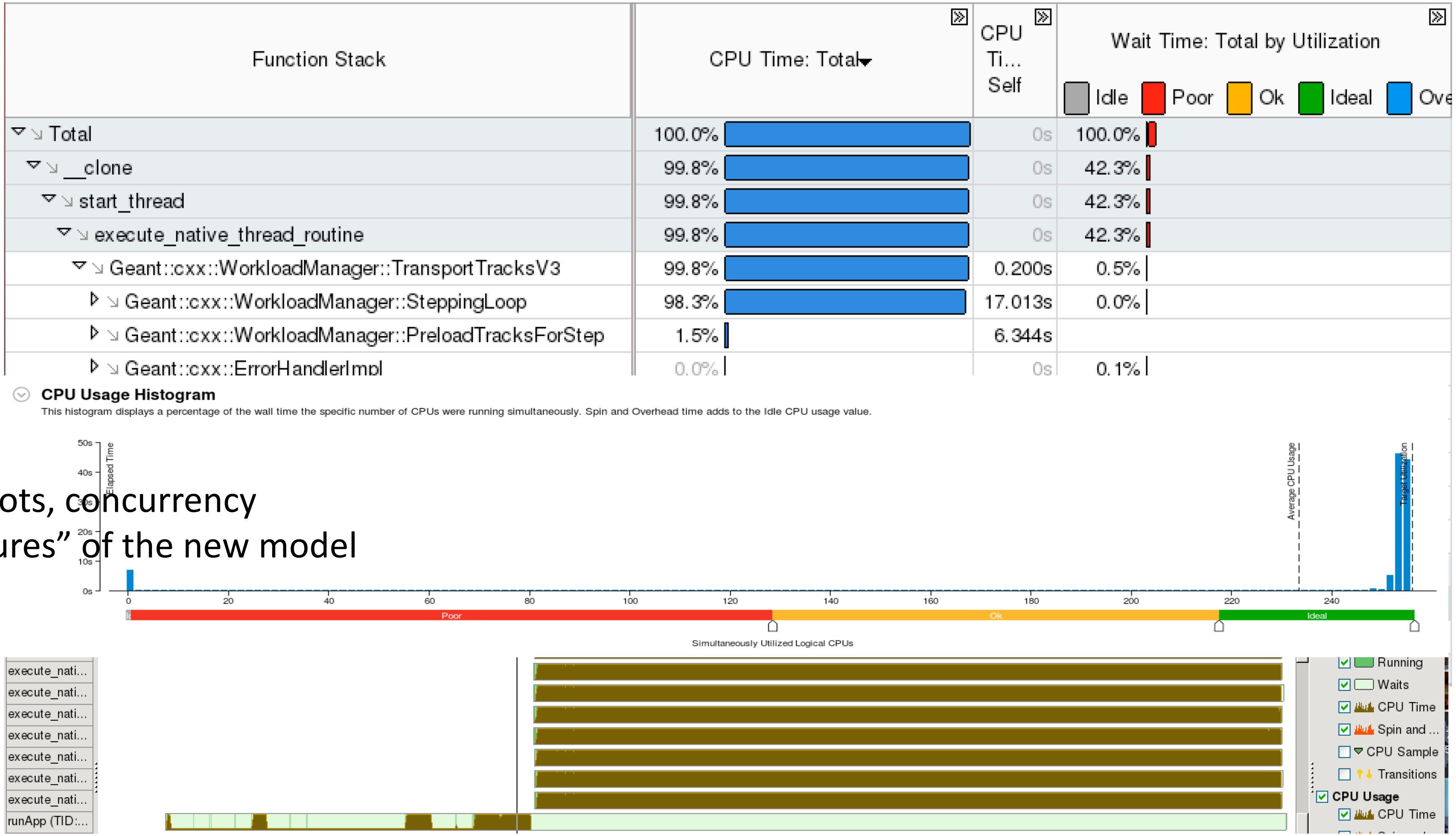
Version 3 preliminary (runApp benchmark)







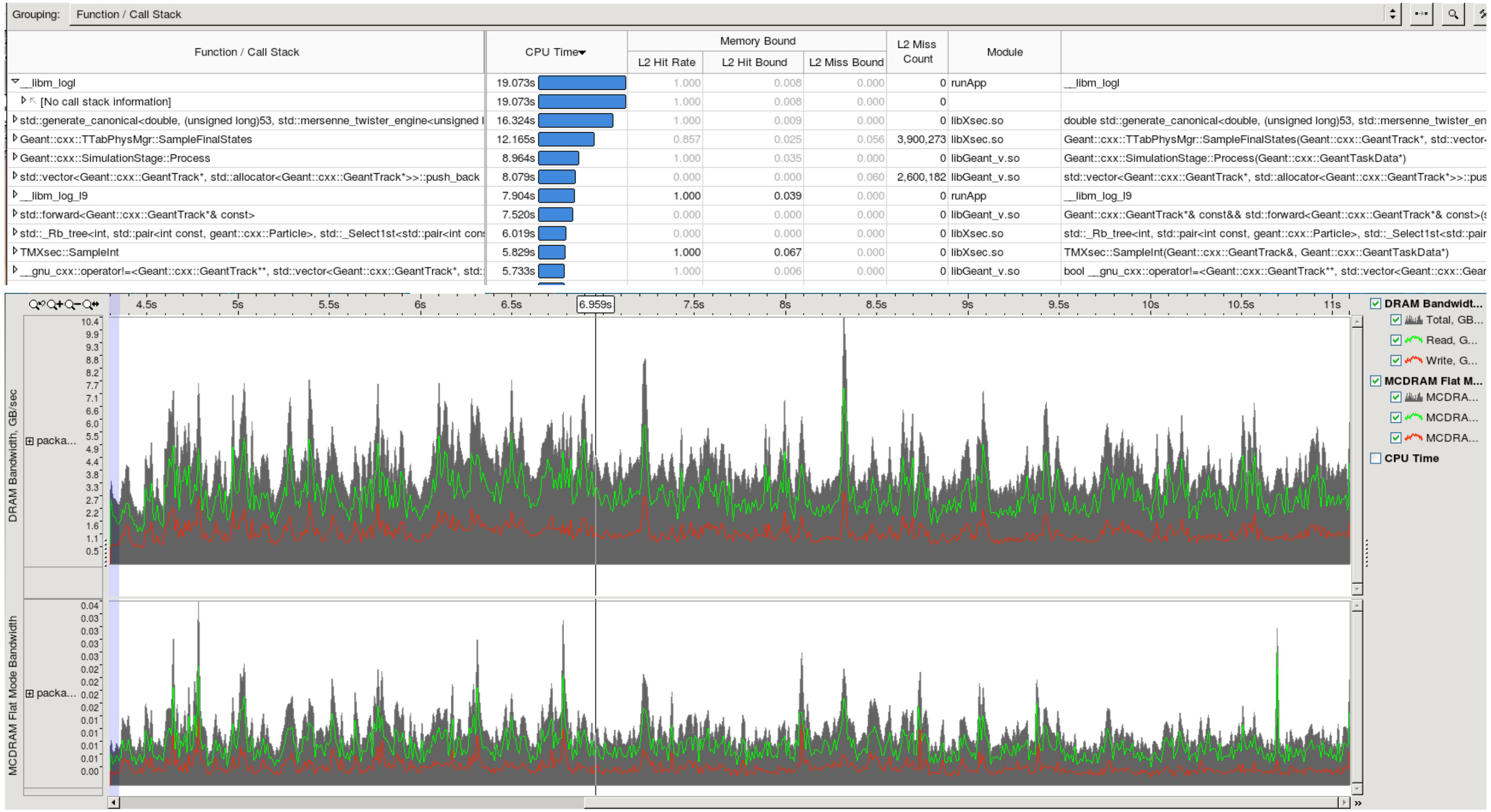
Version 3 preliminary: VTune analysis



- Hotspots, concurrency
- “Features” of the new model

Version 3 preliminary: VTune analysis

- Cache: L2 hit rate is high and L2 hit/miss bound is low for all hotspots
- Mem bandwidth seems OK, peaks at only 10GB/sec, MCDRAM not really used in this mode
- Analysis continues:
  - Vectorization
  - Investigating NUMA effects
  - Hot buffers allocation on MCDRAM
  - Cache vs. flat memory model

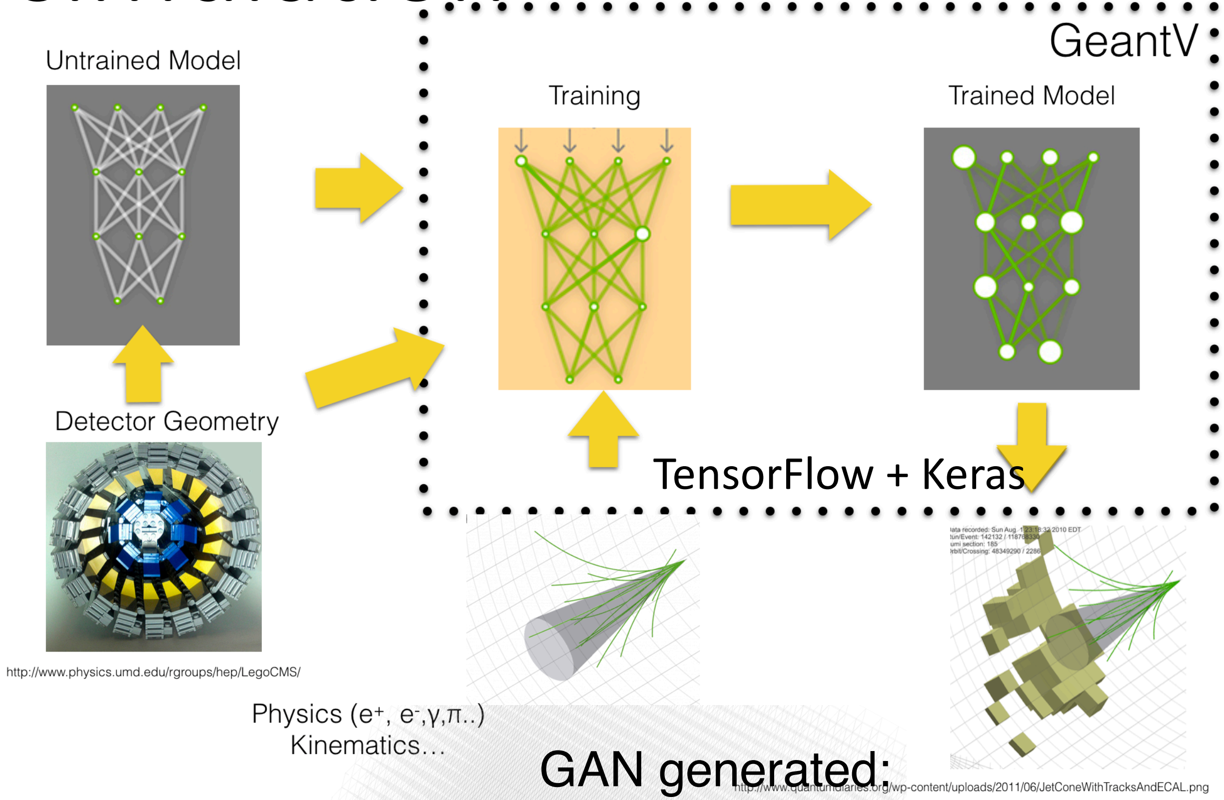


Going beyond x10: fast simulation

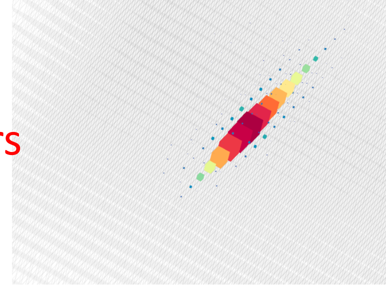
- In the best case scenario GeantV will give O(10) speedup
  - O(100) is rather needed to cope with HL-LHC expected needs
- **Improved, efficient and accurate fast simulation**
  - Currently available solutions are detector dependent
  - Looking for a generic approach + user API
- **A general fast simulation tool based on Machine Learning techniques**
  - ML techniques are more and more performant in different HEP fields
  - Optimizing training time becomes crucial

ML/DL engine for fast simulation

- Train on full simulation
  - Test training on real data
- Test different techniques/models
  - Multi Objective regression, Feature extraction
  - Predictive Clustering Trees & Standard Perceptron (TMVA)
  - Generative adversarial networks (GANs)
- Later: embedded algorithm for hyper-parameters tuning and meta-optimization



GAN generated:
100 GeV e



Fast training by parallelizing (many-core, clusters), lower communication overhead

First 3D images of single particle showers in LCD ECAL obtained training GAN

Conclusion

- GeantV delivers already a part of the expected performance
  - Many optimization requirements, now understanding how to handle most of them
  - GeantV dispatcher version 3 now ready – tests on KNL ongoing
- Additional levels of locality (NUMA) available: topology detection already in GeantV, currently being integrated
- Exploring task-based approach: TBB-enabled version is ready
- Next step: integration with physics and optimization

Thank you!

Backup slides

Vectorization examples using VecCore abstraction library

GeantV plans for HPC environments

future R&D

- Standard mode (1 independent process per node)
  - Always possible, no-brainer
  - Possible issues with work balancing (events take different time)
  - Possible issues with output granularity (merging may be required)
- Multi-tier mode (event servers)
  - Useful to work with events from file, to handle merging and workload balancing
  - Communication with event servers via MPI to get event id's in common files