Design, implementation and performance results of the GeantV prototype

Andrei Gheata for the GeantV R&D team
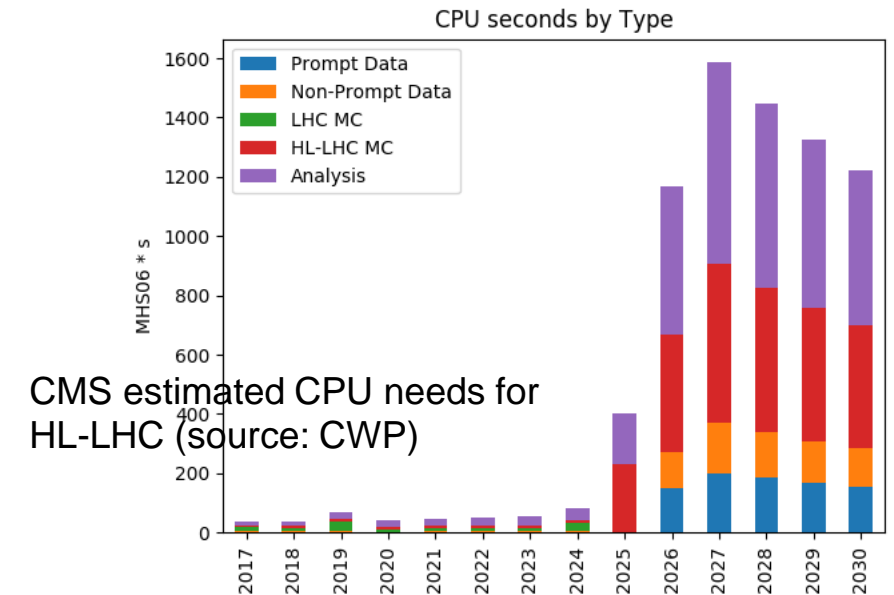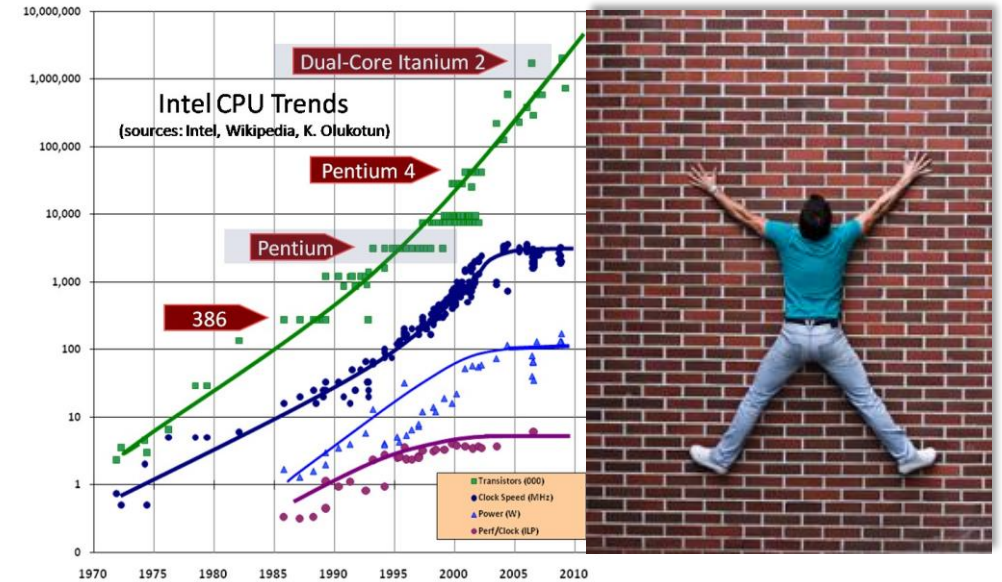
Draft_v2

1. Introduction, context, motivation
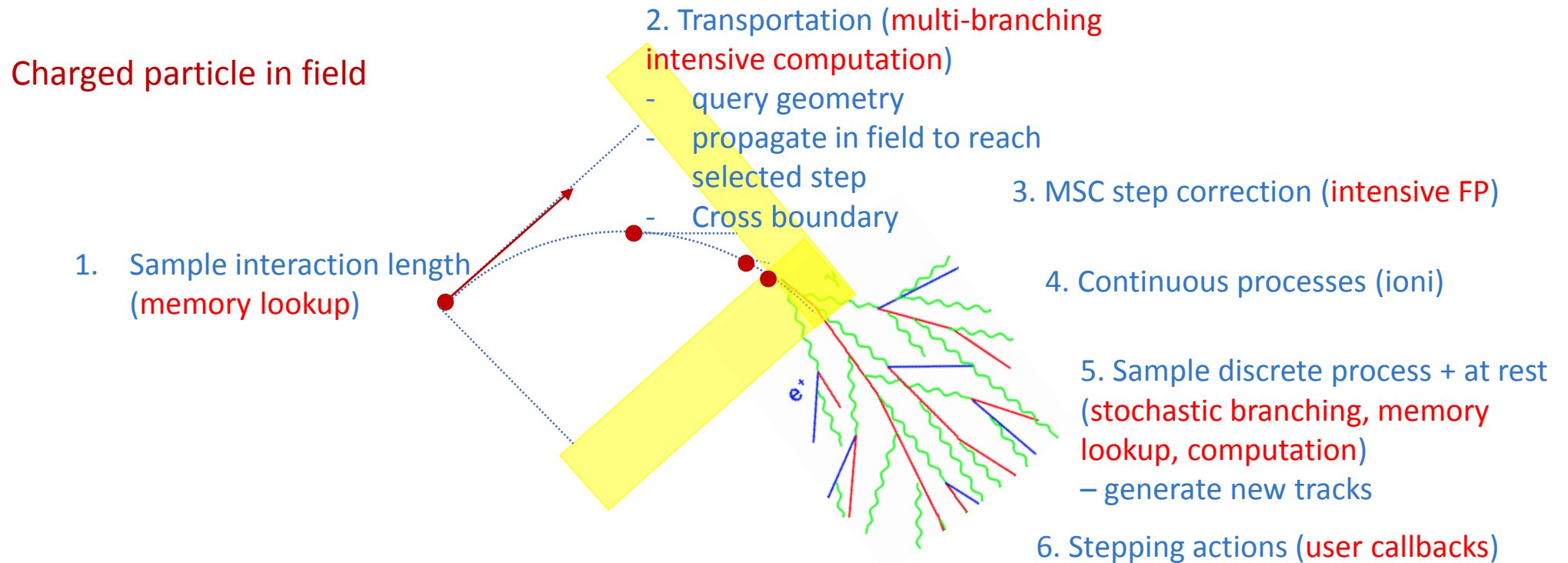
Why?

Context

- ## Hardware: hitting walls
  - power, ILP, memory access – limiting scaling up scalar code performance
    - -> multi/many cores with SIMD pipelines, accelerators

- ## LHC requirements++: Run3->HL
  - Simulation still a bottleneck in many workflows
  - demand for simulated samples ~luminosity

- ## Application requirements: simulation is hard to optimize
  - Very complex stepping per track
  - Large code, sequential OO design from early C++ adoption era (deep stacks, virtual calls)
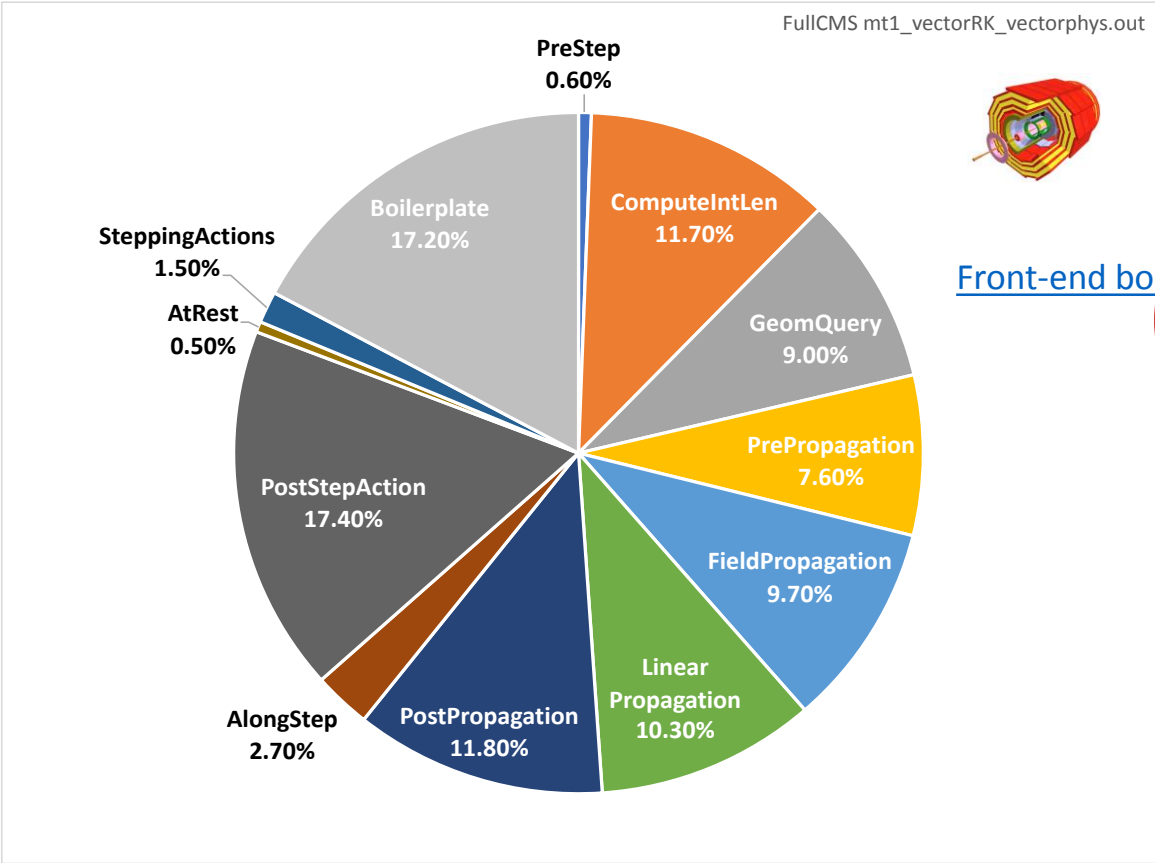 - Small number of instruction per cycle

CMS estimated CPU needs for HL-LHC (source: CWP)

Application morphology: stepping

Charged particle in field

2. Transportation (multi-branching intensive computation)
- query geometry
- propagate in field to reach selected step
- Cross boundary

3. MSC step correction (intensive FP)

1. Sample interaction length (memory lookup)

4. Continuous processes (ioni)

5. Sample discrete process + at rest (stochastic branching, memory lookup, computation)
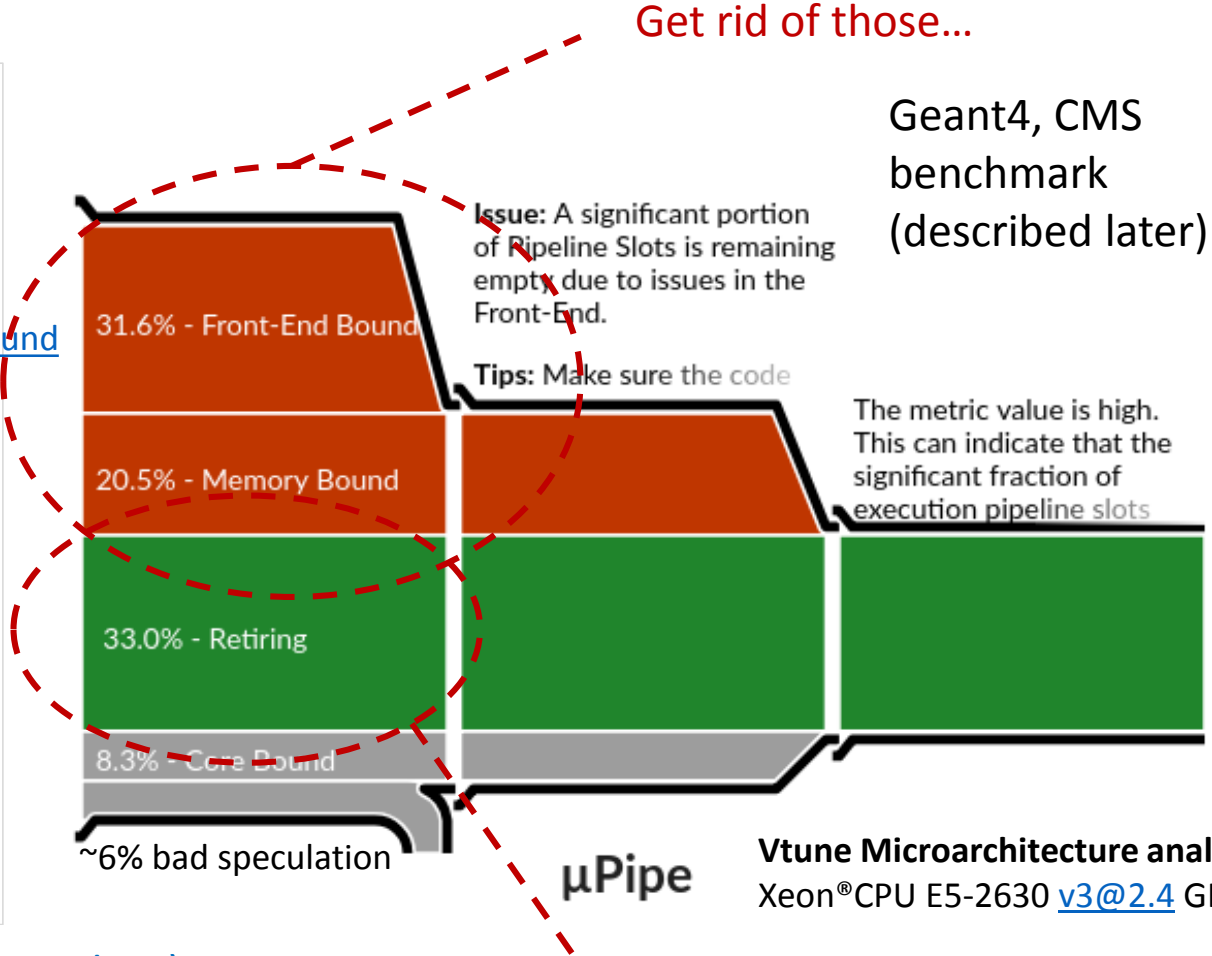– generate new tracks

6. Stepping actions (user callbacks)

- The stages can make a pipeline, but each stage is large and complex (10^2-10^{4+} LOC, deep stacks, stochastic decision tree)
- Track state gets changed after every stage -> strong data binding between functional parts

Motivations: locality & vectorization

FullCMS mt1_vectorRK_vectorphys.out

PreStep 0.60%
ComputeIntLen 11.70%
Boilerplate 17.20%
SteppingActions 1.50%
AtRest 0.50%
GeomQuery 9.00%
PostStepAction 17.40%
PrePropagation 7.60%
FieldPropagation 9.70%
AlongStep 2.70%
PostPropagation 11.80%
Linear Propagation 10.30%

Front-end bound

Get rid of those…

Geant4, CMS benchmark (described later)

31.6% - Front-End Bound

Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.

Tips: Make sure the code

20.5% - Memory Bound

The metric value is high. This can indicate that the significant fraction of execution pipeline slots

33.0% - Retiring

8.3% - Core Bound

~6% bad speculation

μPipe

Vtune Microarchitecture analysis
Xeon®CPU E5-2630 v3@2.4 GHz

The **profiling** numbers are not absolute (application dependent)
- Useful to get **insight** on what happens in a complex setup

Shrink some cycles used here…

5

GeantV

- Initial observations
  - There is large potential for **locality** in simulation
  - Locality opens up parallelism
- Initial ideas
  - Grouping tracks doing same work…
  - … and make the work **vectorizable** on tracks
  - Gather tracks from more events to increase populations
- Initial goals
  - Factor 2-5 per single core for **full sim**
    - increased locality and vectorization, usage of larger % of the hardware
  - New opportunities for accelerators
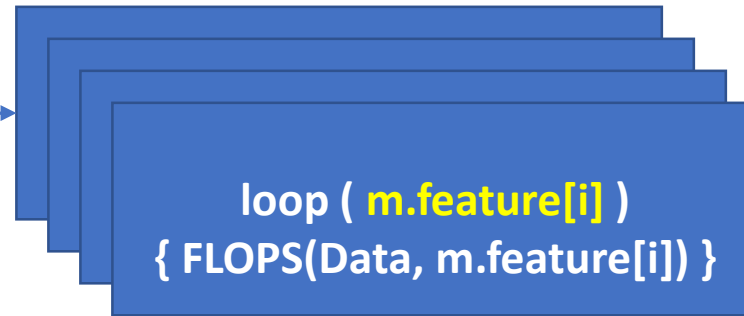 - "Prototype with 2000 lines" (2013)
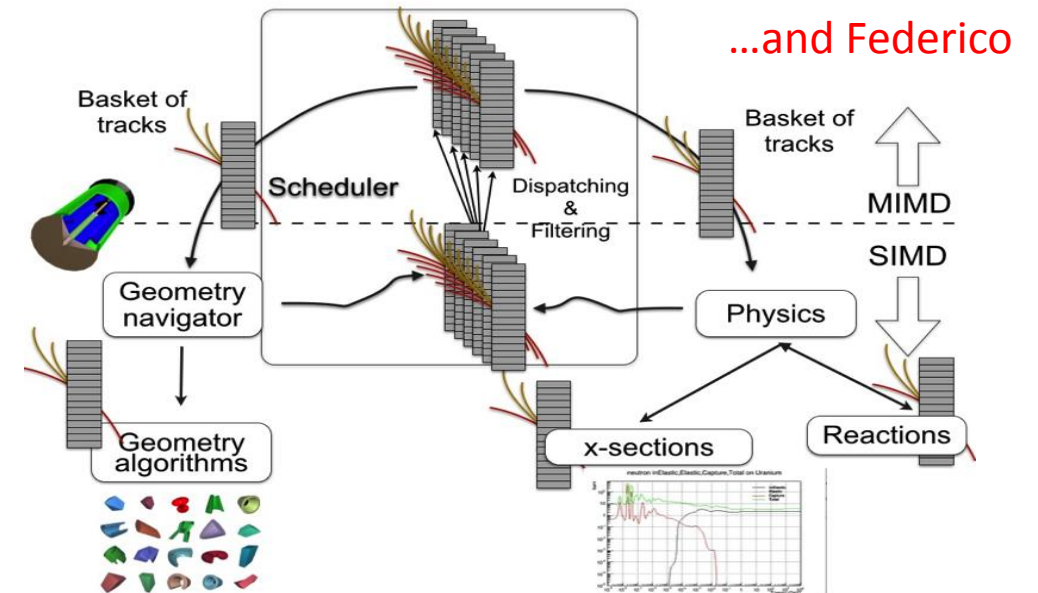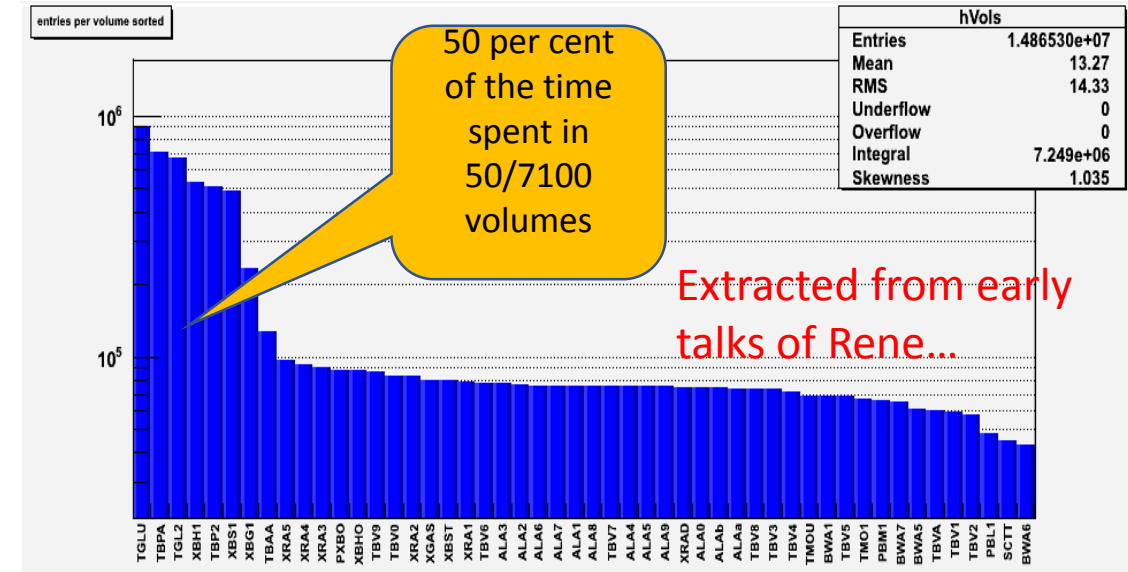
Initial challenge : vectorizing the outer loop?

Model feature parallelism
(e.g. surfaces of a polyhedron)

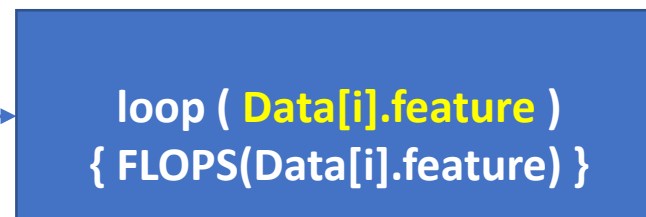Algorithm(Data &, ModelState& m)

Data

loop (**m.feature[i]**)
{ FLOPS(Data, m.feature[i]) }

Very few algorithms with natural inner loops

Data feature parallelism
(e.g. multiple tracks)

Algorithm(vector<Data*> &)

Data

loop (**Data[i].feature**)
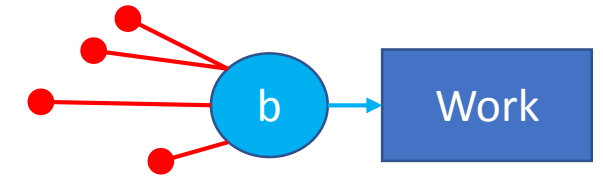{ FLOPS(Data[i].feature) }

A new programming model?

Needs track-parallel environment

Modifying the workflow involves more copy overhead, since m.feature may be const data vector while data[i].feature needs to be gathered -> vector FLOPS need to worth it

2. Concepts, design considerations

Can it be done?

Data processing oriented design

- Bundle data for same **work** type -> make it look more like pipeline
  - Need more tracks doing the same work -> **"basket"**
  - Tracks will need to be regrouped -> state fully contained in track
    - Reentrant methods with tracks/baskets as arguments -> API change
- Enable data locality
  - Tracks in baskets need to be nearby in physical memory -> **basket = SOA**
- May need to reinforce basket populations
  - Allow several events in flight -> **event slots**
  - Data indexed by slot number [0...nslots]
- ... concurrently
  - **Shared** basket data structure with atomic synchronization -> extra complexity

Code redesign: vector interfaces

- Reusing code -> templated kernels for scalar/vector data types
- Geometry locality and basket-aware navigation -> VecGeom
- Enforcing short vectorization -> support different vectorization types
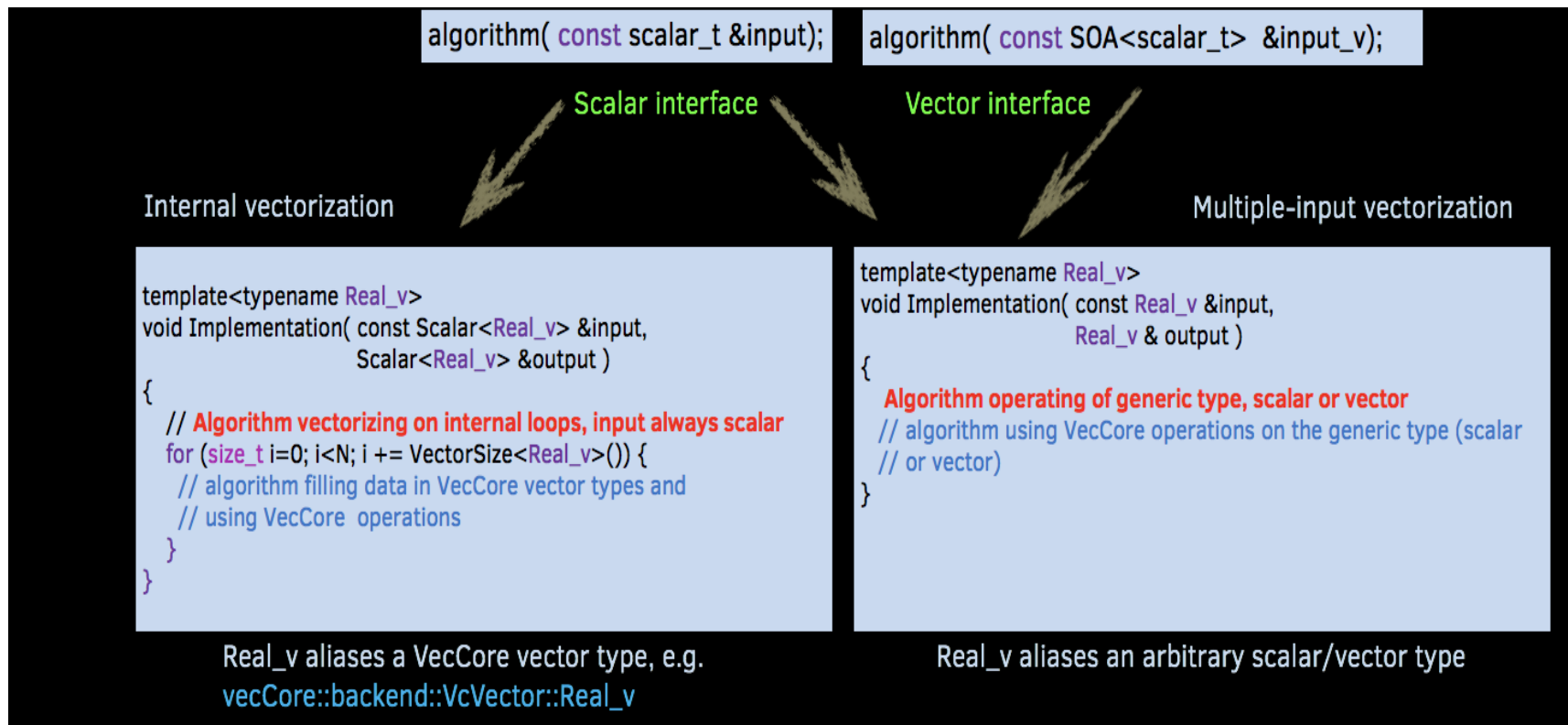
Transforming naturally vectorizable scalar algorithms to use explicit vectorization

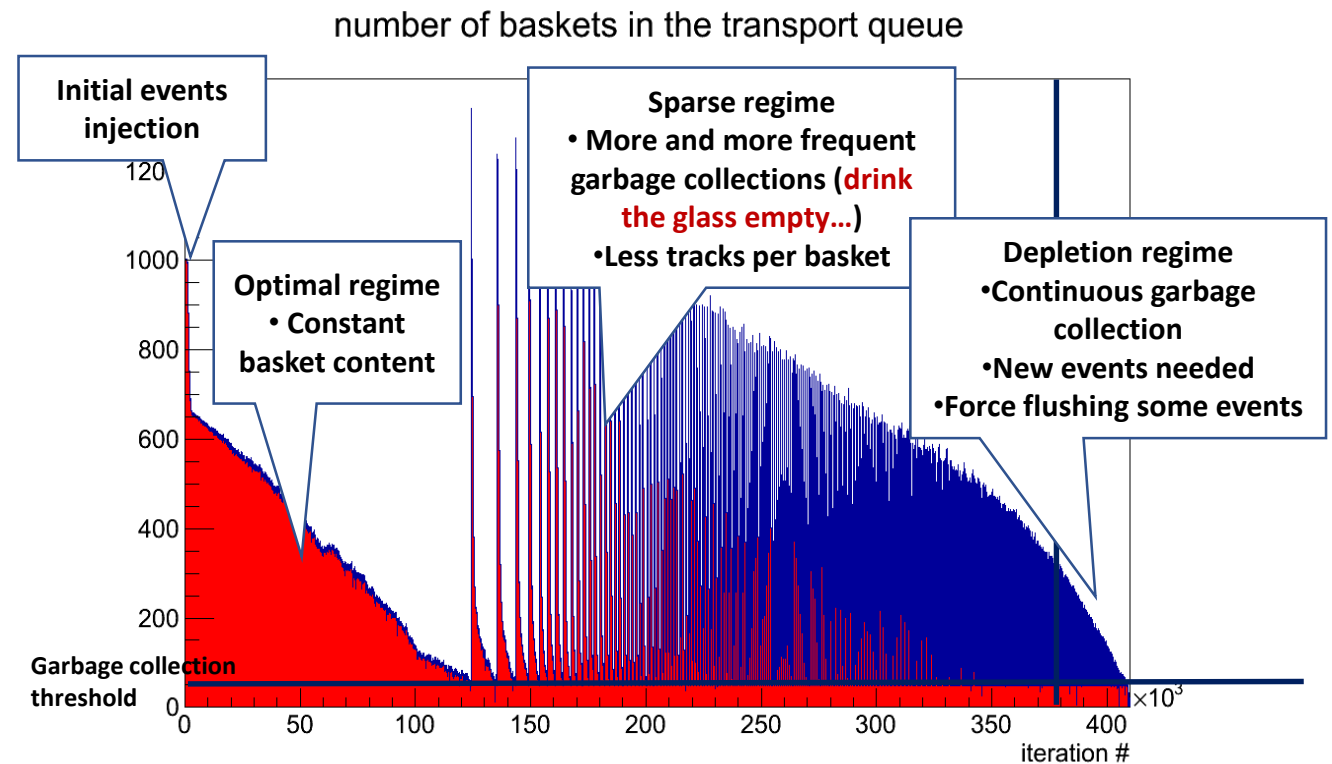**Single kernel for scalar/vector code**

Prototyping revealed unforeseen problems

- Track populations executing different kernels are not uniform
  - Putting one basket per logical volume requires lots of tracks to trigger basket processing on a threshold

- Exhausting input events implies at some point executing partially full baskets: **flushing**
  - Scalar, inefficient, but necessary (due to memory) to finish events

- Concurrent track gathering in general track SOA creates bottlenecks
  - Scaling problems, but also useless data copying

- Boosting both instructions and data

locality in simulation has a **price:**
  - Copy the state from sparse locations

Track block (contiguous)

Basket (SOA)
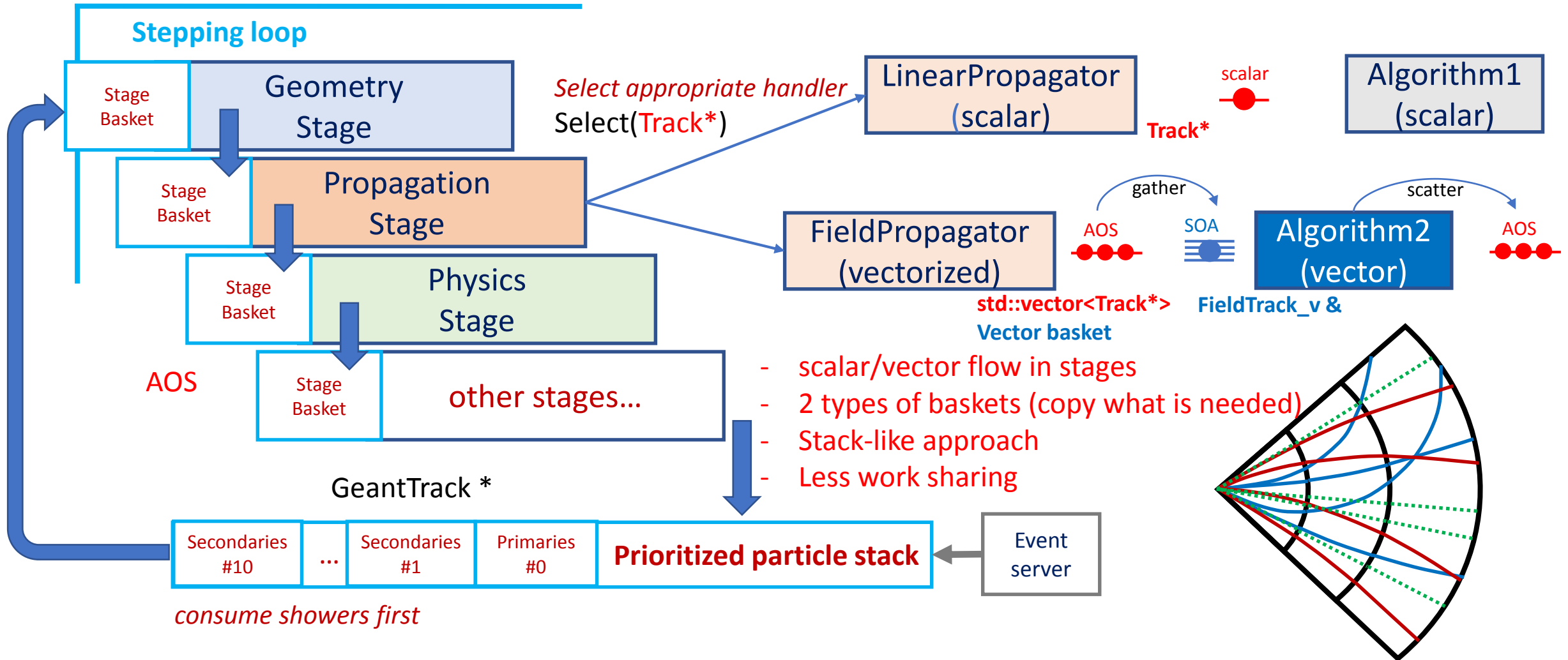
Different colors = different threads
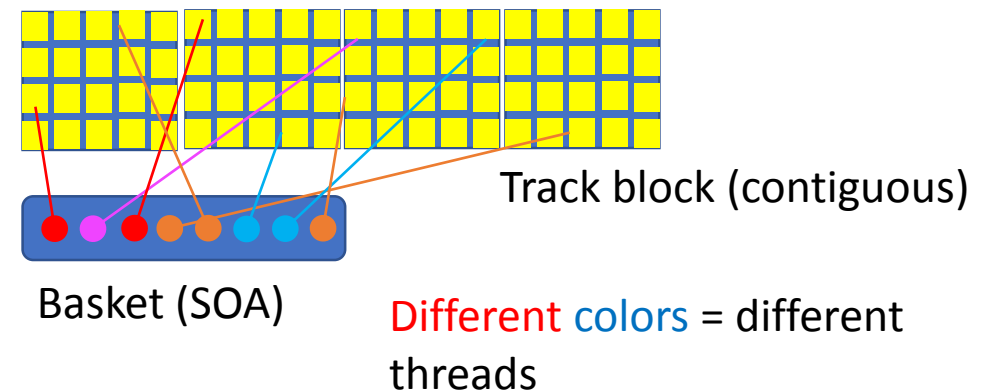
About 2000 lines, one year later...

- A lot of good signs and hopes
  - Geometry components showing good vectorization
- Revealed how **complex** the problem really was...
  - ...and how reality can be different from blackboard drawings
- Dealing with extra complexity and seeking solutions since then
  - Moving gradually from a toy example with geometry only to the full complexity of an LHC experiment simulation

number of baskets in the transport queue

Initial events injection

Optimal regime
• Constant basket content

Sparse regime
• More and more frequent garbage collections (drink the glass empty...)
• Less tracks per basket

Depletion regime
• Continuous garbage collection
• New events needed
• Force flushing some events

Garbage collection threshold

iteration #

Going complex: algorithm-oriented design

Stepping loop

Stage Basket | Geometry Stage

Stage Basket | Propagation Stage

Stage Basket | Physics Stage

AOS

Stage Basket | other stages…

GeantTrack *

Secondaries #10 | … | Secondaries #1 | Primaries #0 | **Prioritized particle stack** ← Event server

consume showers first

Select appropriate handler
Select(Track*)

LinearPropagator (scalar)

scalar ●
Track*

Algorithm1 (scalar)

FieldPropagator (vectorized)

gather   scatter

AOS ●●●   SOA   Algorithm2 (vector)   AOS ●●●

std::vector<Track*>
Vector basket   **FieldTrack_v &**

- scalar/vector flow in stages
- 2 types of baskets (copy what is needed)
- Stack-like approach
- Less work sharing

13

Concurrency design

- Tested different concurrency models: single kernel thread model (worker) vs. TBB-base attempts

- Event server + event slots: thought as a necessity…
  - More data for same work, data indexed by slot number

- Track parallelism: try to find the optimum
  - Be able to share state data, but exchange the minimum

- Towards functional programming style: percolating all state (task data & tracks) through interfaces, making functions fully re-entrant
  - Helpful for cutting down the Amdahl

- Support for externally-driven parallelism
  - What are HEP concurrent frameworks happy with?

3. Implementation: the components

How?

VecCore – SIMD Abstraction Library

- Simple API abstracting common SIMD operations in a generic way
- Evolution of "backends" from VecGeom
  - Abstraction across scalar and vector types
  - Used by VecGeom and ROOT besides GeantV
- Relies on Vc + other backends for the implementation
- Supports ARM, PPC64 with scalar backend
- Supports Windows, Mac, and Linux
- Foundation layer for GPU

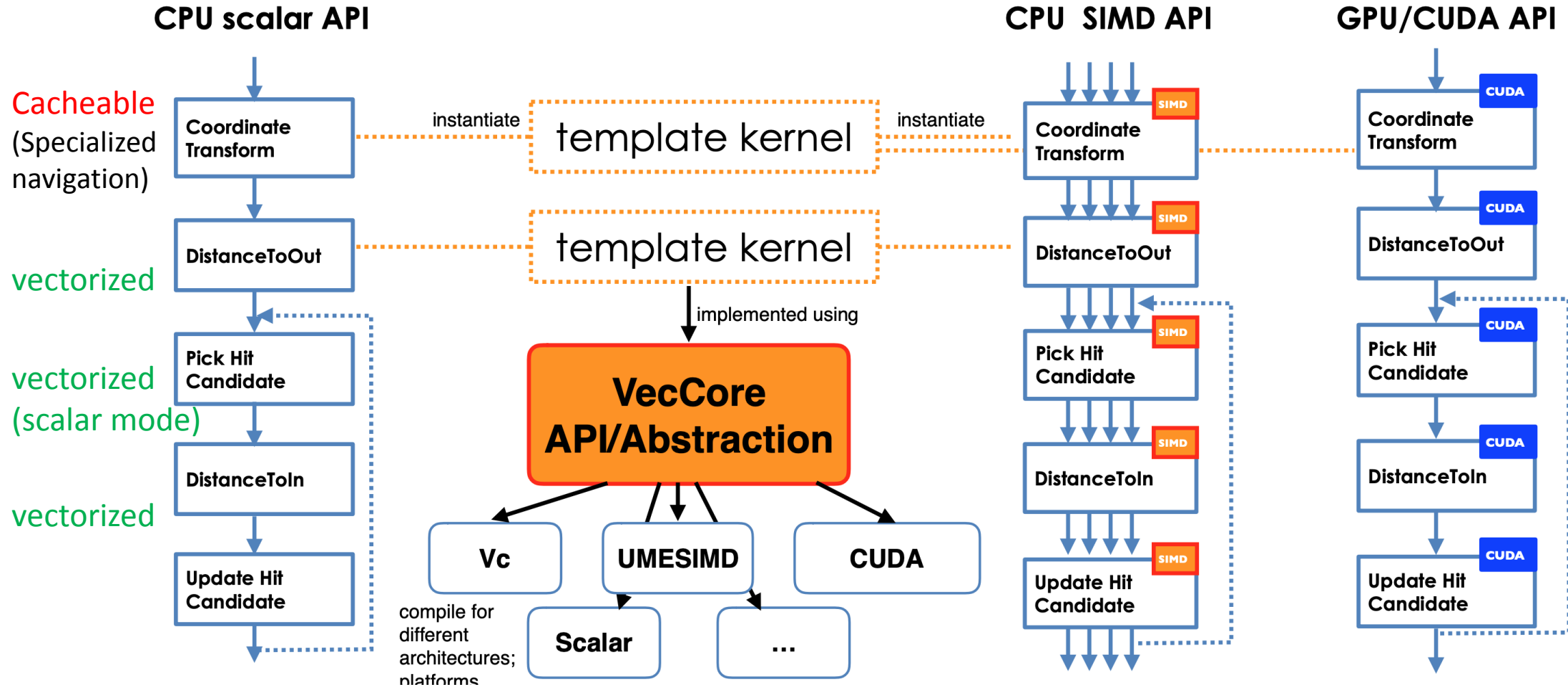https://github.com/root-project/veccore

16
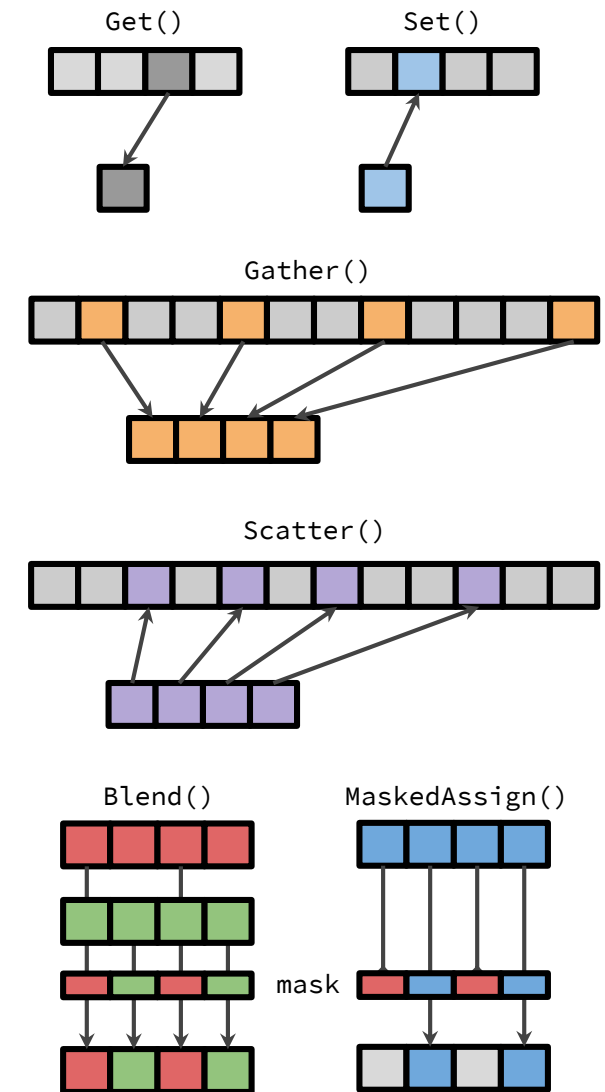
VecGeom – vectorized geometry library

- Workhorse for implementing the main project ideas:
  - Multi-architecture support, scalar/vector workflows
  - Multi-level vectorization (optimization structures + shape algorithms)

- **Performance-driven** development
  - best algorithms inspired from GeantV/ROOT/USolids
  - Production-level quality
  - Unit-tests showing excellent performance figures

- Remaining problem: dispatching efficiently to lower level algorithms
  - Cannot group by boxes/tubes/..., but only by logical volumes
  - Volume navigation has to talk to several daughters -> work divergence

VecGeom Shape Performance Example

Fig. 1: Performance comparison of the tube-segment shape
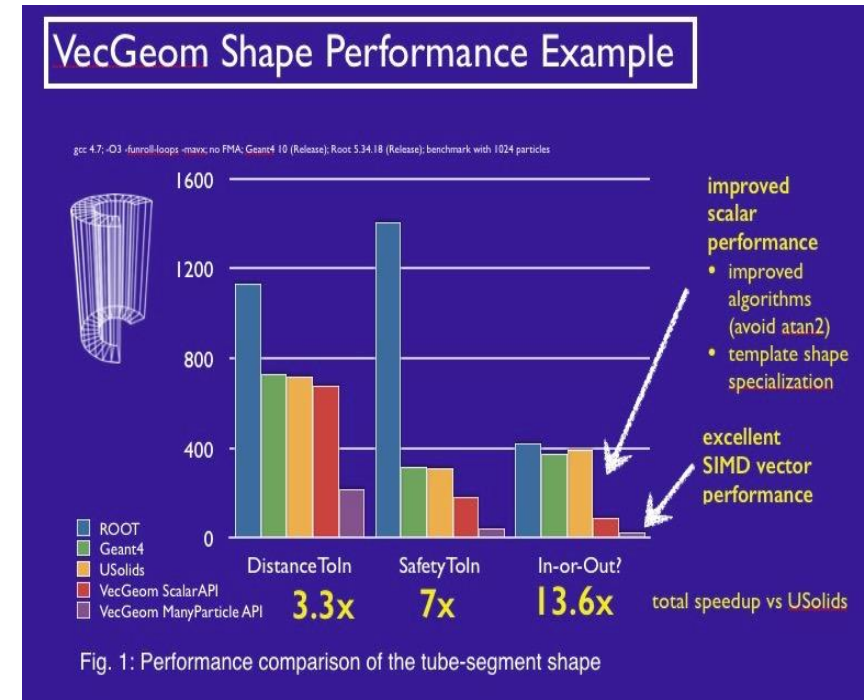
VecGeom code organization

VecMath: vectorization support for math utilities

- Library needed for common vectorized algorithms, math, vector-aware types (Vector3D, SOA3D, AOS3D, ...)
  - So far only: PRNG implementations, fast math functions (vectorized)
- Idea: extending the library to provide common vectorization support
  - Migrate existing common components from VecGeom & GeantV, add extra general-interest utilities
- VectorFlow: A way to express outer loop vectorization in a general way in a *scalar workflow*
  - Templated abstraction based on the concepts of *'work'* and *'flow'* inspired by GeantV
  - Extracted as independent library, not part of VecMath

GeantV EM physics models

| particle	processes	models(s)	
		GeantV	Geant4 defaults
e⁻	ionisation	Møller[100eV-100TeV]	Møller[100eV-100TeV]
	bremsstrahlung	Seltzer-Berger [1keV-1GeV]	Seltzer-Berger [1keV-1GeV]
		Tsai (Bethe-Heitler) w. LPM. [1GeV-100TeV]	Tsai (Bethe-Heitler) w. LPM. [1GeV-100TeV]
	Coulomb sc.	GS MSC model [100eV-100TeV]	Urban MSC model [100eV-100TeV]
			Mixed model [100MeV-100TeV]
e⁺	ionisation	Bhabha [100eV-100TeV]	Bhabha [100eV-100TeV]
	bremsstrahlung	Seltzer-Berger [1keV-1GeV]	Seltzer-Berger [1keV-1GeV]
		Tsai (Bethe-Heitler) w. LPM. [1GeV-100TeV]	Tsai (Bethe-Heitler) w. LPM. [1GeV-100TeV]
	Coulomb sc.	GS MSC model [100eV-100TeV]	Urban MSC model [100eV-100TeV]
			Mixed model [100MeV-100TeV]
	annihilation	-Heitler (2γ) [0-100TeV]	Heitler (2γ) [0-100TeV]
γ	photoelectric	Sauter-Gavrila + EPICS2014 [1eV-100TeV]	Sauter-Gavrila + EPICS2014 [1eV-100TeV]
	incoherent sc.	Klein-Nishina⁺ [100eV-100TeV]	Klein-Nishina⁺ [100eV-100TeV]
	e⁺e⁻ pair production	Bethe-Heitler⁺ [100eV-100TeV]	Bethe-Heitler⁺ [100eV-100TeV]
		Bethe-Heitler⁺ w. LPM [80GeV-100TeV]	Bethe-Heitler⁺ w. LPM [80GeV-100TeV]
	coherent sc	-	Livermore
+	energy loss fluct.	-	Urban

GeantV physics list used also in Geant4 for comparisons

Vectorized EM physics models

- Rewritten models describing nearly complete EM physics (except energy loss fluctuations)
  - More compact implementations, simplified interfaces
  - Support for multiple physics list
  - Several features went back also in Geant4, so the physics Geant4/GeantV can be numerically compatible
- All the models are multi-particle vectorized
  - Most important work was done to vectorize the common services: sampling algorithms (alias, table), track rotation/boost
  - Many challenges: unpredictable recursions, memory access, code complexity
  - **Final state EM speedup**: between **1.5-3** on Haswell, **2-4** on Skylake with **AVX2**
- Most efficient implementation for a model depends on many factors
  - Energy, material composition
  - Precalculated tables or accept/reject
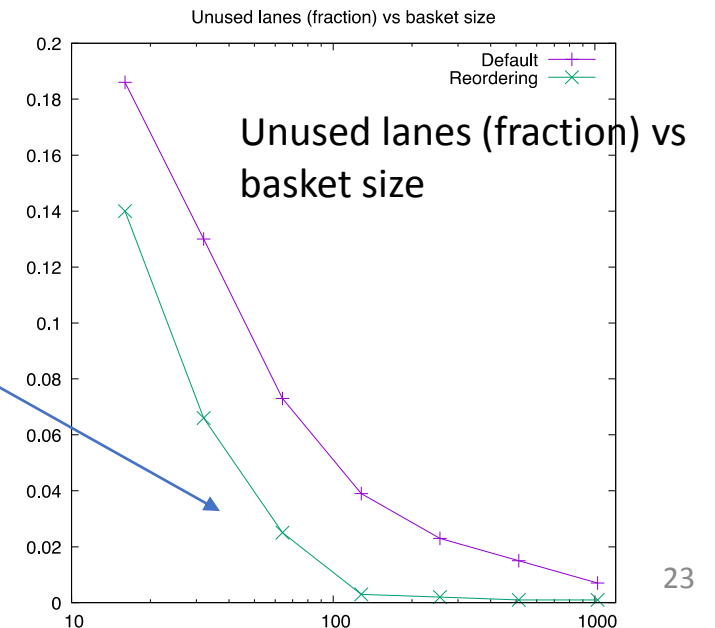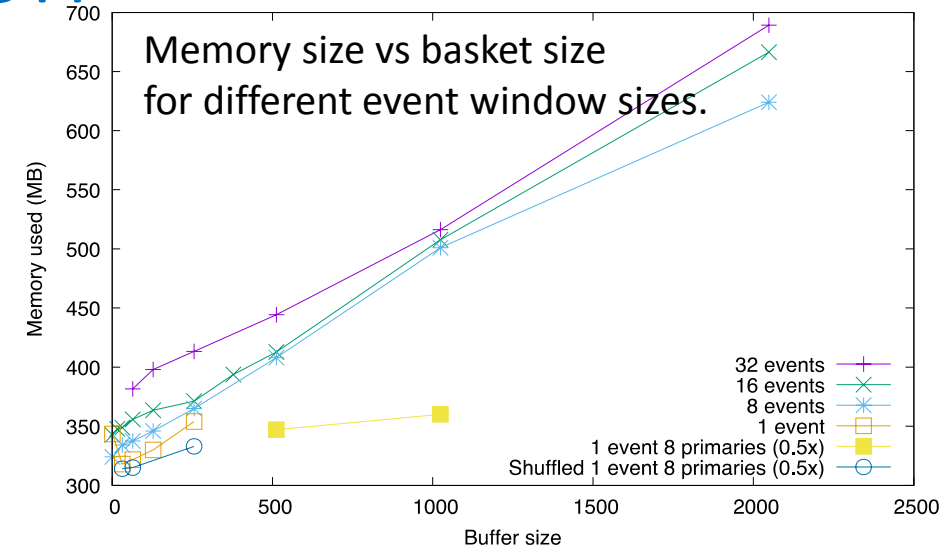 - The full performance study is not complete

Integration of motion in Field

- Integration takes about 18% CPU time in 'scalar' GeantV

- Vectorized over tracks
  - Driver, RK stepper, equation, field interpolation

- Lower level classes 'simply' vectorizable
  - Implementation templated on Field/Equation types

- Top level 'Driver' fully rewritten
  - checks good step and end of integration, reloads vector lanes with new work
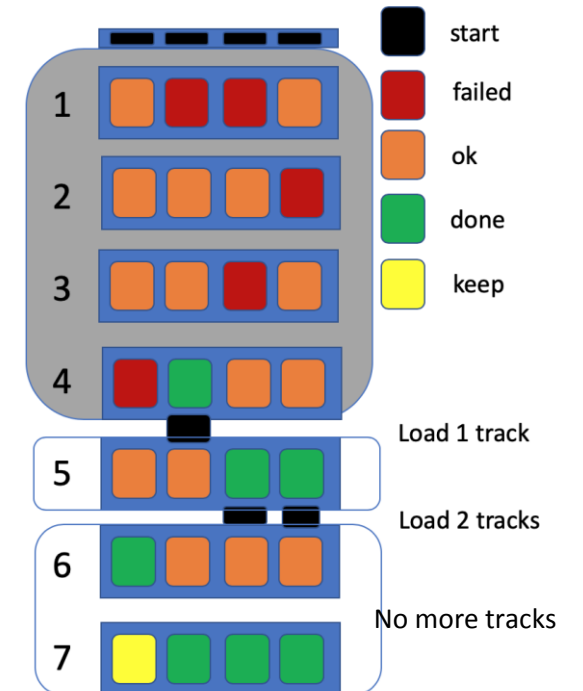
Field Propagation - optimization

- Efficiency and memory use depend strongly on basket size b
  - Lanes doing useful work increases from 82% (b=16) to 99.3% (b=1024)
  - Memory size increases – by 160MB for b=1024 (16 event window) (why?)

- Further refinements possible
  - 'Reordering' tracks - so long integration moves to basket front (tested – 97.5% util. @ bsz=64)
  - Using 'single' track code if only 1 track is left.
  - Improved load / store.

Memory size vs basket size for different event window sizes.

Legend:
- 32 events
- 16 events
- 8 events
- 1 event
- 1 event 8 primaries (0.5x)
- Shuffled 1 event 8 primaries (0.5x)

Unused lanes (fraction) vs basket size

Unused lanes (fraction) vs basket size

Legend:
- Default
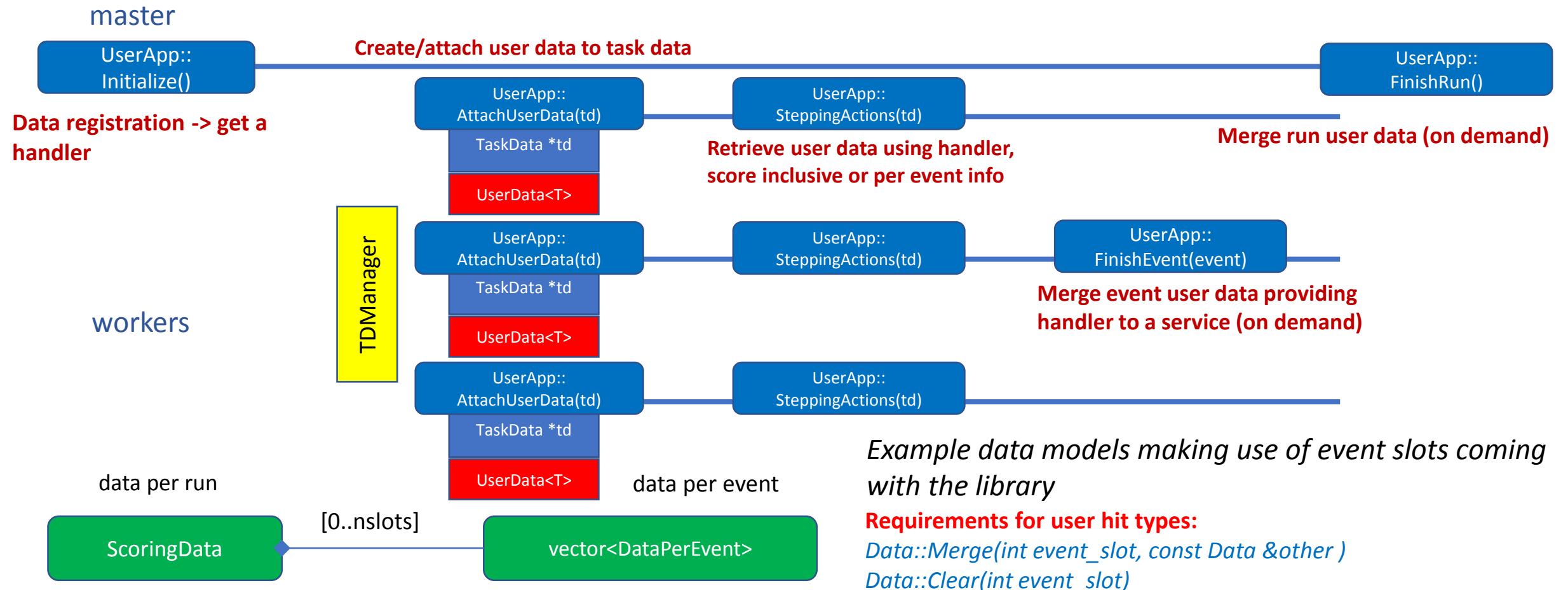- Reordering

MC truth: keeping track of kinematics

- Problem: we need to store the particle history necessary to understand the given event (process)
  - there is no single solution that would cover all use cases
  - functionality is provided as a user-hook allowing concrete user implementations
- interface (MCTruthMgr) implemented in the prototype
  - receives (concurrent) notifications from transport threads about: adding/ending particles, events finishing
  - delegates processing of particles history to concrete MC truth implementation
- Light coupling to transport
  - minimal 'disturbance' to transport threads
  - maximal flexibility of implementing custom particle history handlers
- concrete example implementation provided based on HepMC3
  - See backup slides for more details
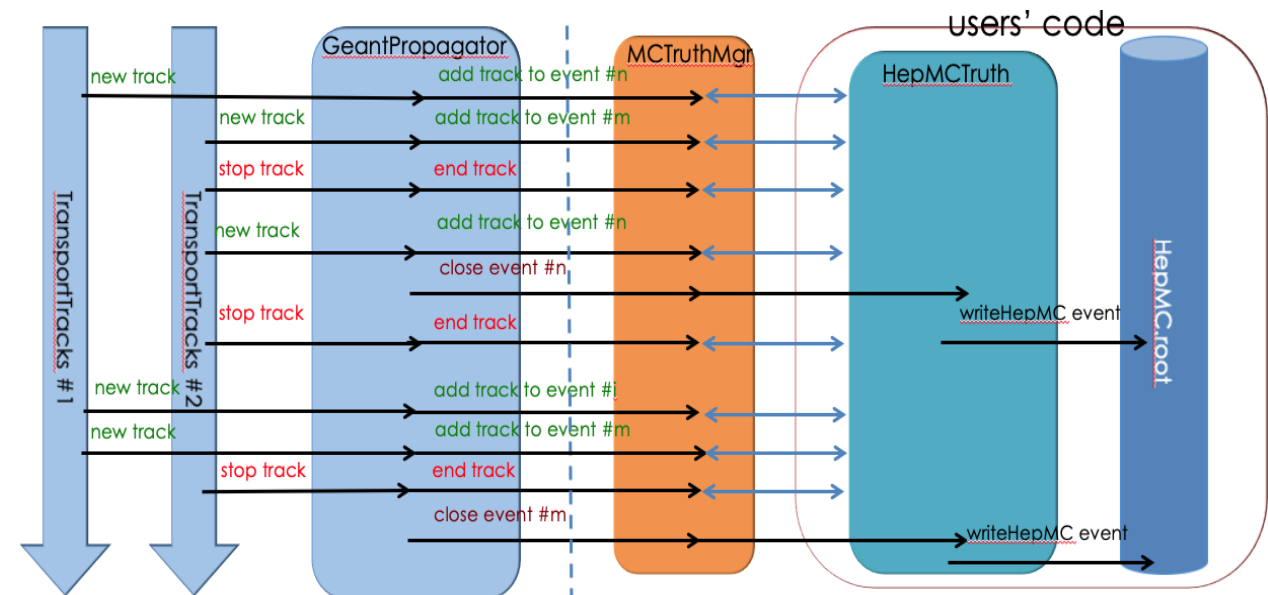
User interfaces: a compromise

- Same callbacks as in Geant4, but dealing with the extra complexity of multiple events and multiple threads
  - Data structures: templated approach (users provide their own types)
  - Data indexed only by event slot, not thread id
- Approach changed from:
  - *"give me your hit model, I give you factories and tools to handle and store them efficiently concurrently"*
    - Nice concurrent merging service ending up in ROOT (TBufferMerger)
- To:
  - *"Here are the hooks allowing to allocate your own data and providing per-thread handles"*
  - *"Here is the workflow allowing to score concurrently and merge hit information"*
- Storing the hits or passing them to digitization is the user business

User data integration in GeantV callbacks

master

UserApp::Initialize()

Create/attach user data to task data

UserApp::FinishRun()

Data registration -> get a handler

UserApp::AttachUserData(td)

TaskData *td

UserData<T>

UserApp::SteppingActions(td)

Retrieve user data using handler, score inclusive or per event info

Merge run user data (on demand)

TDManager

UserApp::AttachUserData(td)

TaskData *td

UserData<T>

UserApp::SteppingActions(td)

UserApp::FinishEvent(event)

Merge event user data providing handler to a service (on demand)

workers

UserApp::AttachUserData(td)

TaskData *td

UserData<T>

UserApp::SteppingActions(td)

Example data models making use of event slots coming with the library

data per run

data per event

Requirements for user hit types:

Data::Merge(int event_slot, const Data &other)

Data::Clear(int event_slot)

ScoringData

[0..nslots]

vector<DataPerEvent>

4. Integration with experiment frameworks

Easy to use?

GeantV Integration in CMSSW

- Integration testing of GeantV w/ CMSSW has several goals:
  - Demonstrate benefits of co-development between R&D team & experiments
  - Exercise capabilities of CMSSW framework to interface with external processing (ExternalWork mechanism) and handle track-level parallelization in detector simulation
  - Measure any potential CPU penalties or gains when running GeantV in CMSSW
  - Estimate cost of adapting to new interfaces and eventually migrating to new (and potentially backward-incompatible) tools such as GeantV
    - Thinking forward to HPC/GPU solutions

- *Not* planning to migrate CMS simulation to GeantV
  - This is an R&D exercise

Overview of the integration exercise

- Exercise and debug features of GeantV and CMSSW
  - Run GeantV using CMSSW ExternalWork feature:
    - Asynchronous, non-blocking, task-based processing
  - Resolved impedance mismatch between original GV scheduler and CMSSW
- Template wrappers for Sensitive detectors (SD) and scoring
  - Ensure exact same SD code used for Geant4 & GeantV
  - Minimize overhead (no branching or virtual table)
  - Handle that each event processed in multiple threads, mixed in with other events (i.e. merge at end of each event processing)
- Performance results and conclusions discussed in separate sections

5. Performance results
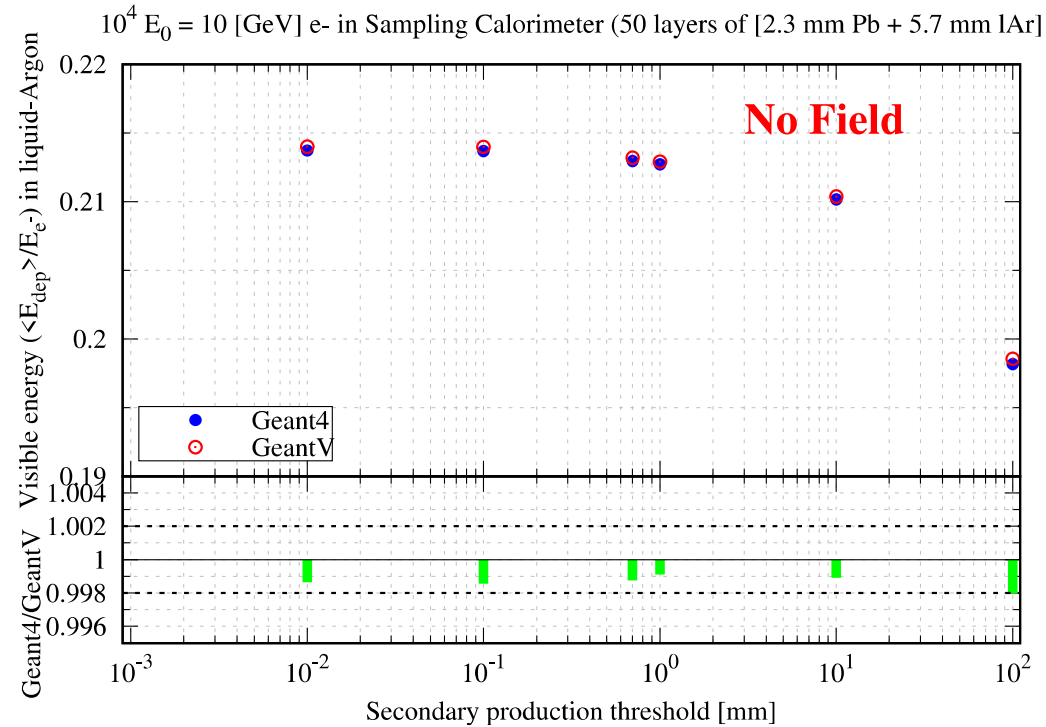
Is it efficient?

Benchmarks

- Set of application examples to demonstrate functionality and/or measure performance
  - Simple setups: simplified layered calorimeter
  - Complex setup with general non-experiment specific stepping actions: CMS
    - Full geometry and production cuts
    - Shooting electrons to fire EM physics
    - Allowing to tune internal GeantV parameters
    - GeantV and Geant4 applications mapped 1 to 1 (geometry, physics lists, gun, cuts)
- Set of CPU platforms
  - Different architectures, CPU, cache configurations
  - Performance results given mostly for the CMS benchmark

Checking physics performance (1)

10^5 e$^-$ with E_0 = 1 [GeV] in 100 layers of 0.8 [mm] liquid-Argon

3×10^5 E_0 = 10 [GeV] Primary Energy e$^-$ in Simplified Sampling Calorimeter
(50 layers of [2.3 mm Pb + 5.7 mm lAr] with 0.7 [mm] Secondary-Production Threshold

4 [T] Transverse Field   No Field

No Field

4 [T] Transverse Field
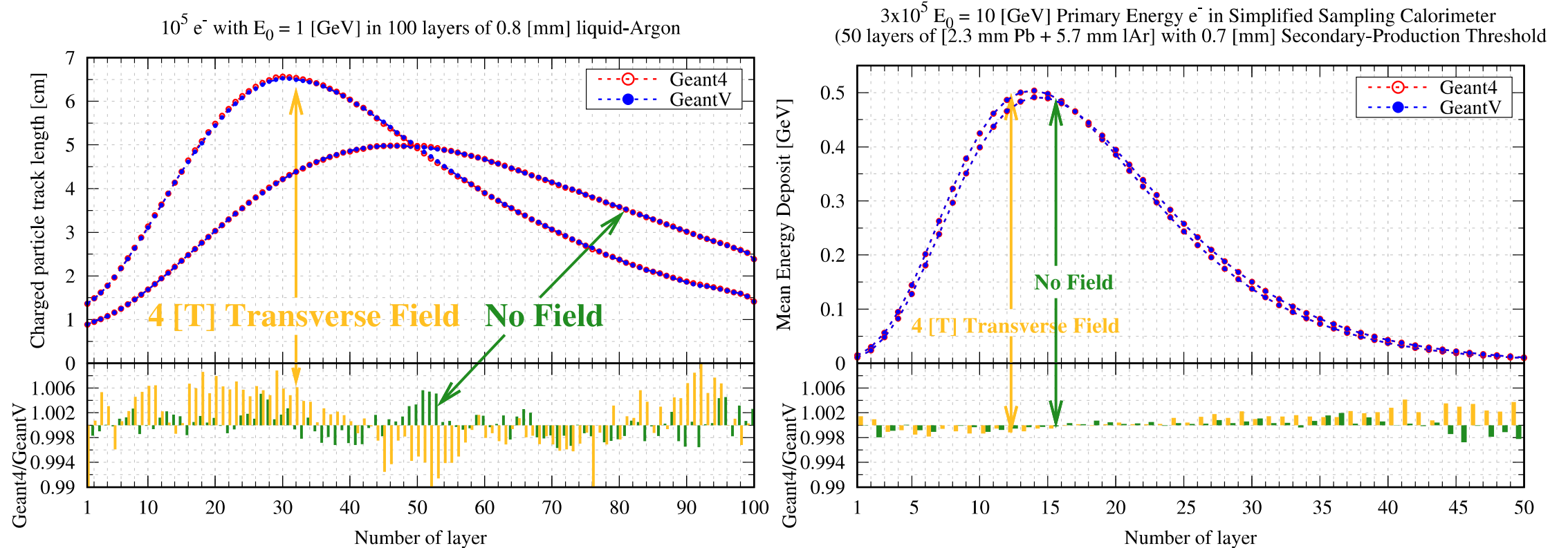
- Per-mil agreement for all observables in most cases

Checking physics performance (2)

10^4 E_0 = 10 [GeV] e- in Sampling Calorimeter (50 layers of [2.3 mm Pb + 5.7 mm lAr]

No Field

10^4 E_0 = 10 [GeV] e- in Sampling Calorimeter (50 layers of [2.3 mm Pb + 5.7 mm lAr]

4 [T] Transverse Field
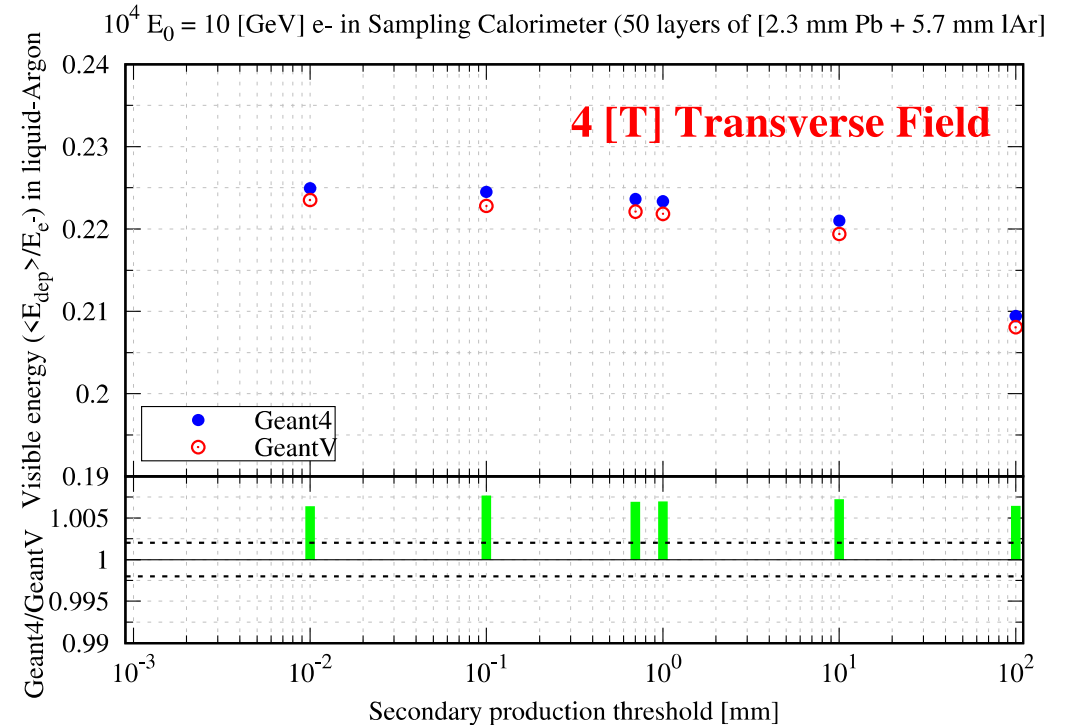
- Some ~1% systematics visible in magnetic field
  - Known issue due to difference in tracking/boundary crossing between GeantV and Geant4

CMS example comparisons

- Configuration details: GeantV vs. Geant4 10.04.p03
  - FullCMS geometry (cms.gdml)
  - No field/Map-based magnetic field
  - GeantV-defined physics lists
  - Input: 10 jobs x 1000 $e+$ each
- Jobs run on single-thread, scalar mode
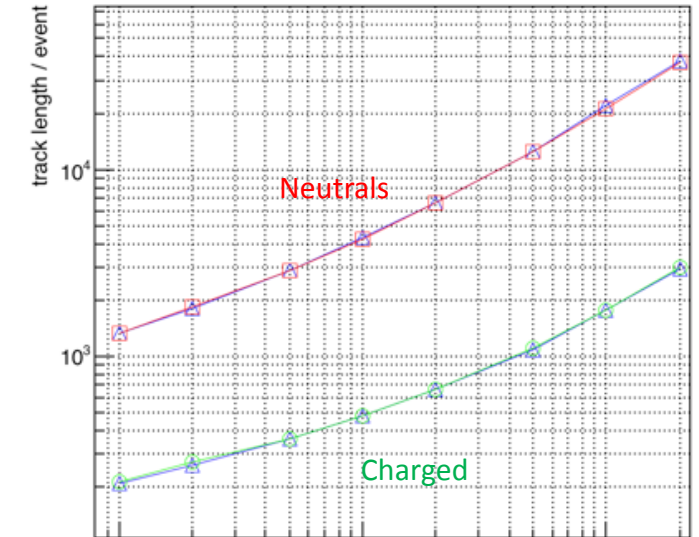- Observables:
  - number of secondaries
  - track length
- Good matching within few percent level
  - statistics to be improved
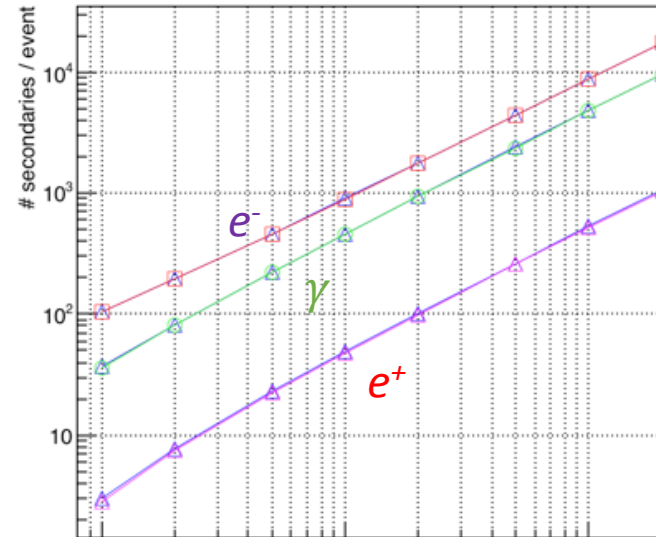
Geant4: blue
GeantV: non-blue
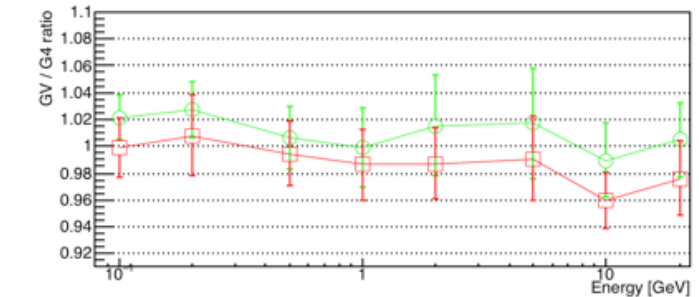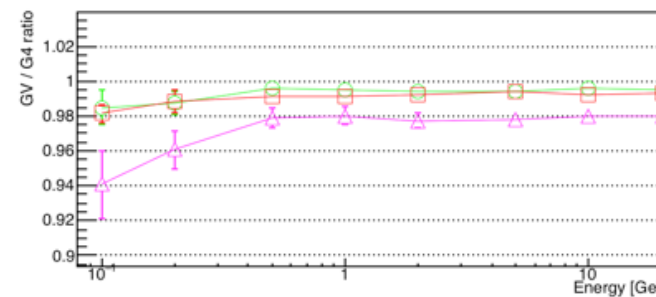
CMS example: stepping observables

10^4, 200 [GeV] e- in CMS detector into dir=[0.109764, 0.987878, 0.109764] with exact CMS regions(cuts)

B = 0

Good matching for all observables

| Mean values per primary	e$^-$, e$^+$ and γ interactions; **Magnetic Field: NO**	
	Geant4	**GeantV**
total energy deposit:	200 [GeV]	200 [GeV]
total (charged) track length:	198.09$\pm$ 2.72 [m]	197.92$\pm$ 2.98 [m]
total (neutral) track length:	2285$\pm$ 100 [m]	2222$\pm$ 108 [m]
number of (charged) steps:	4.102$\times$10$^5\pm$ 4.914$\times$10^4	4.024$\times$10$^5\pm$ 4.999 $\times$10^4
number of (neutral) steps:	7.242$\times$10$^5\pm$ 1.18$\times$10^4	6.994$\times$10$^5\pm$ 1.208$\times$10^4
number of secondary γ:	9.78$\times$10$^4\pm$ 311	9.73$\times$10$^4\pm$ 321
number of secondary e$^-$:	1.739$\times$10$^5\pm$ 1481	1.727$\times$10$^5\pm$ 1574
number of secondary e$^+$:	1.083$\times$10$^4\pm$ 79.32	1.061$\times$10$^4\pm$ 79.69

10^4, 200 [GeV] e- in CMS detector into dir=[0.109764, 0.987878, 0.109764] with exact CMS regions(cuts)

B = 3.8 T

Mean values per primary	e$^-$, e$^+$ and γ interactions; **Magnetic Field: 3.8 [T]**	
	Geant4	**GeantV**
total energy deposit:	200 [GeV]	200 [GeV]
total (charged) track length:	199.68$\pm$ 6.11 [m]	198.94$\pm$ 6.25 [m]
total (neutral) track length:	2328.25$\pm$ 108 [m]	2260.64$\pm$ 101 [m]
number of (charged) steps:	4.253$\times$10$^5\pm$ 8.128$\times$10^4	4.126$\times$10$^5\pm$ 7.644 $\times$10^4
number of (neutral) steps:	7.729$\times$10$^5\pm$ 1.605$\times$10^4	7.471$\times$10$^5\pm$ 1.5488$\times$10^4
number of secondary γ:	9.78$\times$10$^4\pm$ 311	9.73$\times$10$^4\pm$ 293
number of secondary e$^-$:	1.739$\times$10$^5\pm$ 1475	1.726$\times$10$^5\pm$ 1397
number of secondary e$^+$:	1.083$\times$10$^4\pm$ 80.6	1.061$\times$10$^4\pm$ 75.3

Physics validation in CMSSW simulation test

- Geant4 10.4p2 w/ VecGeom v0.5 (scalar) vs GeantV pre-beta-7 w/ VecGeom v1.1
  - All CMS-specific G4 optimizations disabled
  - Same production cuts (default 1mm)
  - Single thread (reproducible pRNG sequences)

- Roughly the same distributions with no magnetic field

- Small difference in the physics results in the presence of constant B-field
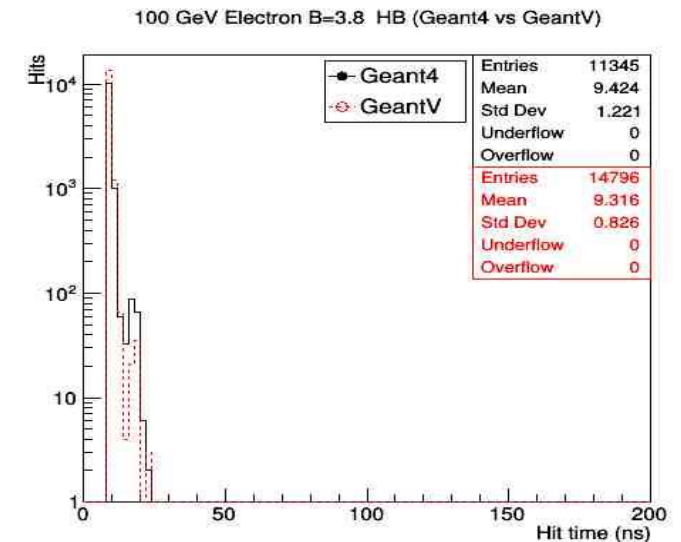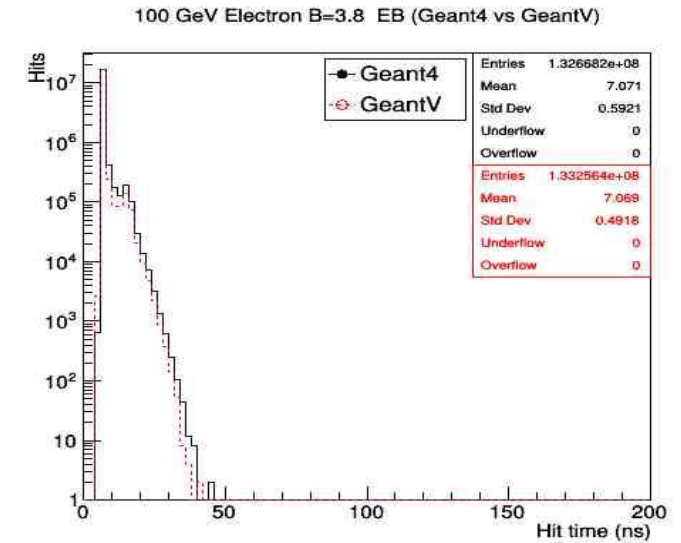
Hit Time for 100 GeV e- (B=3.8)

Basketizing: efficiency, vectorization, overhead per stage

- Several execution modes to measure stage performance
  - Scalar mode (no baskets): T_{scalar}
  - Vector mode (fill baskets and call vector algorithm): T_{vector}
  - Basket "emulation" mode* (fill baskets and call scalar algorithm in loop): T_{BE}
  - Scalar dispatch mode+ (execute full stepping loop with single particle): T_{SD} -> measure impact of improved GeantV code

- Measure efficiency & overhead for basketization relative to total run time
  - Overhead: $B_o = (T_{BE} - T_{scalar}) / T_{scalar}$
  - Observed efficiency: $B_e = (T_{scalar} - T_{vector}) / T_{scalar}$
  - Vectorization efficiency: $B_v = B_e + B_o$

*BE mode hard to measure for some stages (e.g. physics) missing emulation of scalar scatter of internal SOA basket
+ emulating Geant4 stepping but with GeantV data model

Results: basketizing efficiency

CMS application benchmark

- 100 GeV isotropic e[-]
- 100 primaries, 16 event slots
- Field type: CMS map
- 1 thread, performance mode

- ### Bad vectorization efficiency overall
  - Up to 10-12% depending on platform
  - Large overhead, sometimes bigger than vectorization
    - Dispatching to many small basketizers (geometry) not effective

- ### Good use cases: field and MSC
  - larger % FLOP

Fractions of total scalar execution time

Xeon®CPU E5-2630 v3@2.4 GHz ,16 core

2x32KB L1/core, 256KB L2, 20MB L3

Stage	% total	B_e	B_o	B_v
Field	14.4%	5.0%	2.0%	7.0%
Phys[+]	9.4%	0.3%	1.4%	1.7%
Geom[*]	12.3%	-3.3%	3.7%	0.4%
MSC[x]	8.6%	1.8%	0.2%	2.0%
FPM[o]	32.4%	5.8%	1.5%	7.3%

+ Only post step sampling of physics models
* Only querying distance to boundary and safety
x Only MSC position/direction correction calculation
o Best configuration for vectorization (Field /Physics/MSC)

Measurement errors < 0.5%

Basketizing overheads dependence on architecture

Geant4 (10.4.p03) vs. GeantV (beta)
10 GeV electron x 1000 events (1-thread, 10 measurements), 16 event slots

CPU	OS	gcc	SIMD	Cache L1	Cache L2	Cache L3	B_o (field)	B_o (physics)	B_o (geometry)	B_o (MSC)	B_o (FPM)
Intel i7 2.5GHz	Ubuntu 16.04	5.4.0	AVX2	4x32 KB I 4x32 KB D	4x256 KB	8 MB	2% ± 1%	2% ± 1%	6% ± 1%	0% ± 1%	3% ± 1%
Intel Core i7-4510U 2GHz	Ubuntu 16.04	5.4.0	AVX	2x32 KB I 2x32 KB D	2x256 KB	4 MB	-1% ± 7%	-3% ± 7%	12% ± 9%	-4% ± 8%	2% ± 8%
AMD A10-7700k	Fedora Workstation 29	8.2.1	AVX	2x96 KB I 4x16 KB D	2 x 2 MB	-	15% ± 1%	4% ± 1%	15% ± 1%	1% ± 1%	13% ± 1%
Intel Celeron 1000M 1.8GHz	Fedora Workstation 29	8.3.1	SSE4	2x32 KB I 2x32 KB D	2x256 KB	2 MB	9% ± 1%	5% ± 1%	9% ± 1%	-1% ± 1%	9% ± 1%
Intel Centrino2	Fedora Workstation 29	8.2.1	AVX	-	4 MB	-	6% ± 1%	3 ± 1%	13% ± 1%	-1% ± 1%	7% ± 1%
11AMD e-300	Ubuntu 18.10	8.2.0	SSE2	2x32 KB I 2x32 KB D	2x512 KB	-	1% ± 1%	3% ± 1%	-3% ± 1%	-2% ± 1%	In process

Overhead seems to largely increase for smaller (data) cache size

"Basketizing": benefits vs. costs

- Costs (coming from initial scalar approach):
  - Workflow redesign, interface redesign, data structure re-engineering
  - Copy overheads: data regrouping, gather/scatter
  - Filling baskets concurrently -> additional overheads due to contention
  - Algorithm vectorization effort
- Benefits:
  - Improved instruction locality
  - Data locality would improve if re-basketizing could be done only with collocated tracks
  - SIMD instructions: making use of important % of the silicon for more algorithms
  - Code more compact/efficient and accelerator-ready
- Efficient basketization needs reasonable FLOPS workload
  - Algorithm vectorization can be inefficient for the same reasons as loop vectorization...
    - Branching, early returns, complexity

Performance summary table: Geant4 vs. GeantV

Geant4 (10.4.p03) vs. GeantV (beta)

10 GeV electron x 1000 events (1-thread, 10 measurements)

strk (single track mode): emulation of Geant4 style tracking
,

Summary of speed-ups for different architectures

CPU	OS	gcc	SIMD	Cache L1	Cache L2	Cache L3	GV [sec]	G4/GV	strk/GV0	Vector Gain
Intel i7 2.5GHz	Ubuntu 16.04	5.4.0	AVX2	126KB	1MB	8 MB	941 ± 5	1.41 ± 0.04	1.02 ± 0.02	1.09 ± 0.01
Intel Core i7-4510U 2GHz	Ubuntu 16.04	5.4.0	AVX	128KB	512KB	4 MB	1,303 ± 3	1.09 ± 0.01	0.95 ± 0.07	1.09 ± 0.08
AMD A10-7700k	Fedora Workstation 29	8.2.1	AVX	2x96 KB I 4x16 KB D	2x2M	-	1,828 ± 5	1.80 ± 0.04	1.25 ± 0.03	1.01 ± 0.01
Intel Celeron 1000M 1.8GHz	Fedora Workstation 29	8.3.1	SSE4	64KB	512KB	2 MB	2,769 ± 10	1.03 ± 0.01	1.11 ± 0.01	0.84 ± 0.01
Intel Centrino2	Fedora Workstation 29	8.2.1	AVX	-	2x2 MB	-	2,592 ± 2	1.92 ± 0.01	1.24 ± 0.01	1.01 ± 0.01
11AMD e-300	Ubuntu 18.10	8.2.0	SSE2	64KB	1 MB	-	Not Vc compatible	Not Vc compatible	in process	Not Vc compatible

CPU performance of G4/GV varies significantly over different platforms

Some observations and open questions

- Vectorization efficiency largely impacted by the basketizing overheads
  - Removing the overheads would give about same efficiency everywhere
- Single track mode emulating Geant4-like stepping shows loss of locality only on systems with poor caching. Possible reason:
  - More compact code fitting cache on modern CPUs
- Wildly varying performance ratio GV/G4 depending on architecture, cache configuration
  - Coming from Geant4 being frontend-bound?
  - Cache size/architecture, but also memory latency/throughput?

CPU Benchmark on the Fermilab Wilson Cluster

- Benchmark
  - GeantV (pre-beta-7) vs. Geant4 (10.5)
  - The standalone Geant4/GeantV application using a CMS gdml with a CMS field map
  - 10 × 10 GeV e−/event, 1000 events
  - measurements on quiet batch nodes (error < 1%)
- CPU Time in [sec] and performance comparisons between Geant4 and Geant4
  - CPU performance widely varies on different processors
  - marginal gain by SIMD vectorization (maximum ~ 10%)

Processor	GeantV	GeantV-vec	Geant4	G4/GV	G4/GV-vec
SSE4-2.3-15	4457	4333	6627	1.49	1.53
AVX-2.0-15	2621	2331	4938	1.88	2.12
AVX2-2.4-35	1628	1530	2182	1.34	1.43
AVX2-2.5-28	1186	1275	1875	1.58	1.47

  - Processor: SIMD-CPU[GHz]-Cache[MB]
- What is the source of gain (~1.4-2.1) in Geant4/GeantV?

Vector Instruction and Gain (AVX)

- % of vectorization = (PAPI_DP_VEC)/(PAPI_DP_OPS)
  - PAPI DP VEC = Double precision vector/SIMD instructions
  - PAPI DP OPS = Floating point (double precision) operations
- PAPI (performance API) hardware counters in [1 Billion]

Mode	PAPI_DP_OPS	PAPI_DP_VEC	% vectorization	CPU gain
scalar	1770	277	15.67	-
vec-geo	1771	333	18.82	0.96
vec-mag	1858	814	43.83	1.08
vec-msc	1789	397	22.24	1.02
vec-phys	1785	343	19.25	1.00
vec-all	1868	1051	56.26	1.00
vec-opt	1868	996	53.35	1.12

- % of vectorization is high, but gain is small
  - Vectorization comes with the price of too many data moves

Performance Comparison: Geant4 vs. GeantV libraries

- Exclusive time (%) of big libraries

GeantV Library (%)	AVX	AVX2	SSE4	Geant4 Library (%)	AVX	AVX2	SSE4
libGeant_v.so	42.1	46.3	43.2	libG4geometry.so	41.8	43.6	42.3
libRealPhysics.so	36.0	34.2	37.3	libG4processes.so	22.0	20.8	21.0
libGeantExamplesRP.so	14.1	14.1	14.5	libG4global.so	7.3	8.0	7.5
libc-2.12.so	3.8	1.8	1.1	libG4tracking.so	7.3	6.5	7.2
libVmagfield.so	3.1	2.8	3.1	libG4track.so	6.0	4.7	5.8
libm-2.12.so	0.6	0.6	0.6	full_cms	5.2	6.1	6.6
libCore.so	0.1	0.1	0.1	libG4clhep.so	3.3	3.0	3.0
libGeom.so	0.1	0.1	0.1	libm-2.12.so	2.7	3.5	2.9

- There are not much variations in the percent of time over different processors (CPUs/Cache Size)
- The performance difference between Geant4 and GeantV is a global effect (i.e., not driven by a single module or a set of functions)

Performance Comparison: L1 Cache and TLB Misses

- **L1 cache miss: in [Billion] counters**

Processor	GV (ICM)	G4(ICM)	GV (DCM)	G4(DCM)
AVX-2.0-15	48	398	190	250
AVX2-2.4-35	48	462	194	248
SSE4-2.3-15	100	285	282	134

- ICM (DCM) = Instruction (data) cache miss
- GeantV shows much significantly less ICM

- **TLB (translation lookaside buffer) miss: in [1M] counters**
  - cache for page tables which map addresses between virtual and physical memory
  - GeantV show much less TLB misses

Processor	GV (IM)	G4(IM)	GV (DM)	G4(DM)
AVX-2.0-15	53	4256	3168	4626
AVX2-2.4-35	N/A	N/A	24	82
SSE4-2.3-15	55	149	88	1628

Performance Comparison: IPC and FMO

- ## IPC = Instruction(INS)/Cycle(CYC) : Good Balance with Minimal Stall

Processor	GV INS/CYC	GV IPC	G4 INS/CYC	G4 IPC
AVX-2.0-15	7209/6846	1.05	8388/10788	0.78
AVX2-2.4-35	6733/5544	1.21	8458/6178	1.37
SSE4-2.3-15	7847/8869	0.88	8459/11228	0.75

- IPC varies relatively smaller in GeantV compared to Geant4
- For Geant4, INS is nearly constant, but CYC widely varies

- ## FMO = FL/(LD+SR) : CPU Utilization

- FL (Floating point instruction), LD (load), SR (store) in [1B] counters
- GeantV shows the better FMO in all tested platforms

Processor	GV FL/(LD+SR)	FMO	G4 FL/(LD+SR)	FMO
AVX-2.0-15	1718/(2422+980)	0.50	2181/(3812+1697)	0.40
AVX2-2.4-35	2347/(882+876)	1.34	3824/(1704+1396)	1.23
SSE4-2.3-15	3191/(1756+1948)	0.86	1620/(1397+4118)	0.29

Application profiles from VTune microarchitecture analysis – CMS benchmark

Geant4

Can this be improved?

GeantV

Issue: A significant portion of Pipeline Slots is remaining empty due to issues in the Front-End.

Tips: Make sure the code

31.6% - Front-End Bound

20.5% - Memory Bound

33.0% - Retiring

8.3% - Core Bound

The metric value is high. This can indicate that the significant fraction of execution pipeline slots

μPipe

Frontend & memory bound

34.8% - Memory Bound

30.1% - Retiring

19.9% - Core Bound

10.8% - Bad Speculation

The metric value is high. This can indicate that the significant fraction of execution pipeline slots could be stalled due to demand memory load and

This metric represents how much Core non-memory issues were of a bottleneck. Shortage in hardware

A significant proportion of pipeline slots containing

μPipe

Core & memory bound

Large, not understood

Xeon®CPU E5-2630 v3@2.4 GHz

48

Performance tests for CMSSW integration exercise

- Settings:
  - GeantV pre-beta-7+ (63468c9b)
    - Enabled: vectorized multiple scattering, field (not physics)
  - Disable output
- Machines:
  - FermiCloud VM w/
    - Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz, **4096 KB cache**, sse4.2 (Sandy Bridge EP)
  - CERN OpenLab w/
    - Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz, **35840 KB cache**, 28 cores, sse4.2 (Haswell)

- Standalone GV/G4 test: 2.14x (Sandy Bridge), 1.6x (Haswell) speedup
- CMSSW GV/G4 test: 2.6× (Sandy Bridge), 1.7x (Haswell) speedup with single thread
  - G4 has better scaling w/ # threads than GV (see backup)

Concurrency: Strong scaling

Not just Amdahl, but also basket efficiency loss
due to more flushes (scalar mode)

- Acceptable efficiency, but far from perfect (~80% for 16 threads)
  - Price to pay for concurrent services, track stealing
  - Hard to improve w/o fully binding events to threads
- Do we need to exchange tracks between threads?
  - Buffer of events bound to threads, using more memory…
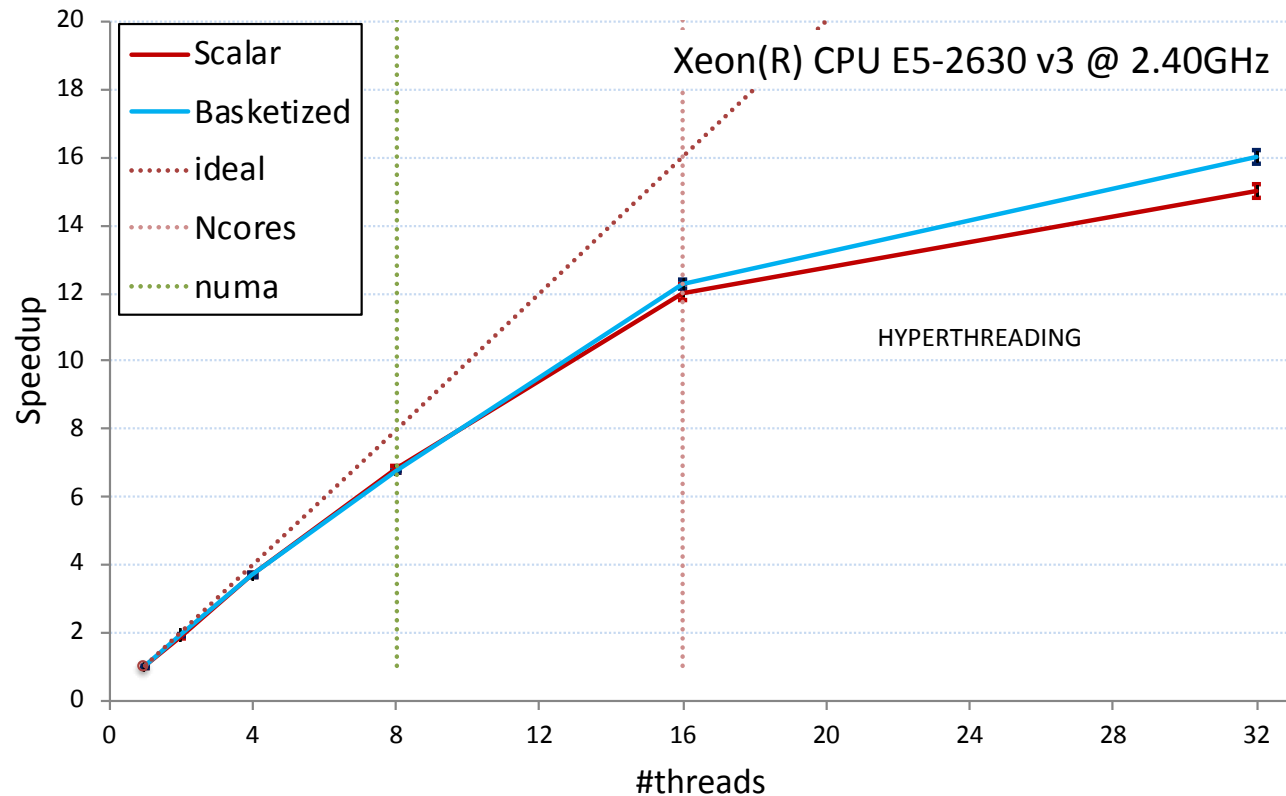
Memory efficiency

- Larger memory footprint than Geant4 (~3x)
  - Expected to improve for large #nthreads (no study yet)
  - Depending on number of buffered events
- Memory efficiency
  - Scaling with number of tracks in flight (see also backup)
  - Price to pay: decrease of memory access coherency -> data cache misses increasing with #nthreads

CMS benchmark, 10 GeV electrons

Track block (contiguous)

Dead track (hole)

New track

Basket (pointers to tracks)
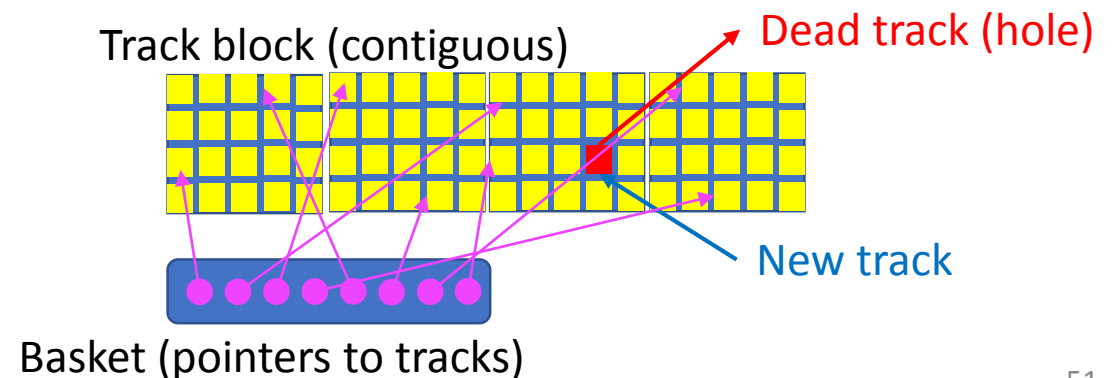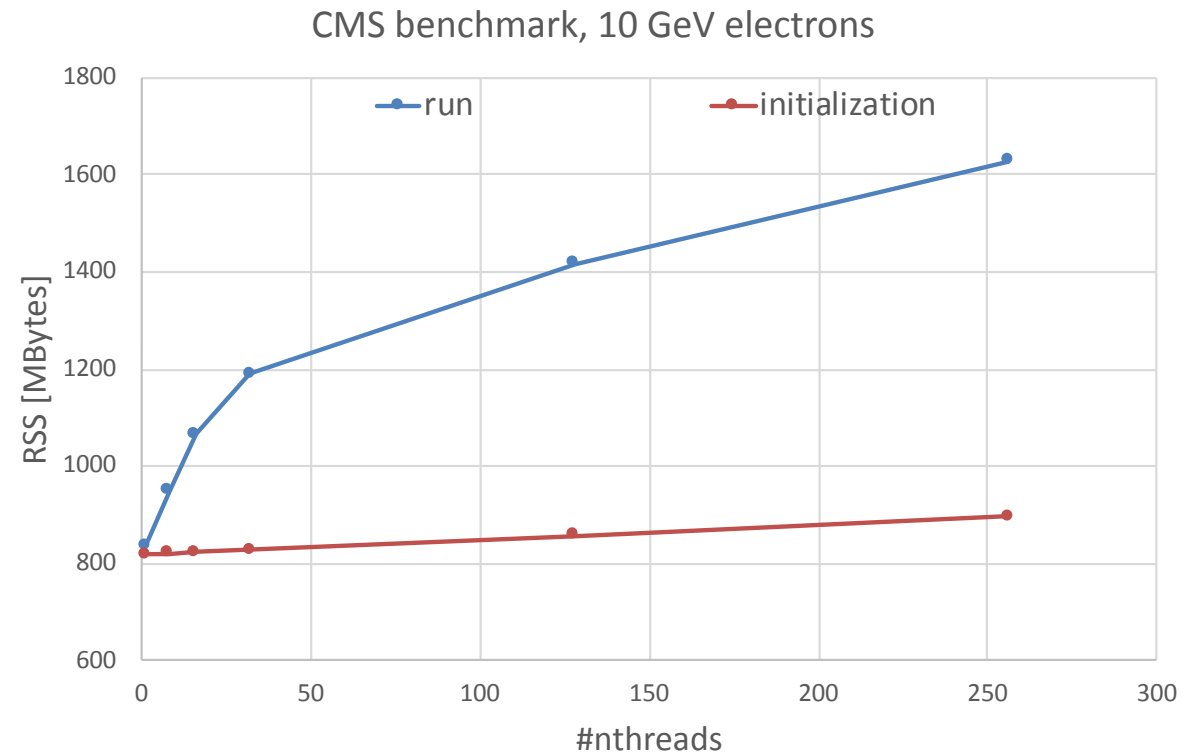
Topology awareness
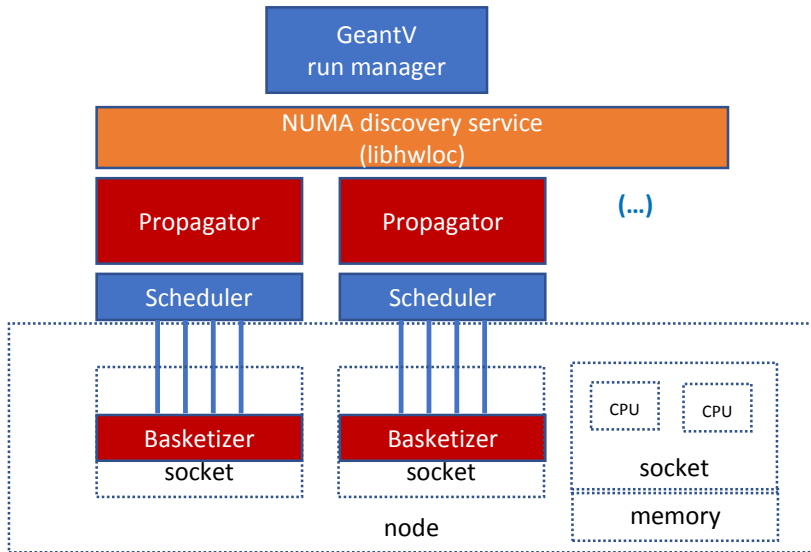
- Thread binding to cores by policy
  - Compact/scatter over NUMA domains
  - Multiple propagators/schedulers
- Track block allocator NUMA aware
  - Blocks owned by threads
- Measurable NUMA effect
  - pointing to data cache misses

Diagram (left):
GeantV run manager
→ NUMA discovery service (libhwloc)
→ Propagator | Propagator (...)
→ Scheduler | Scheduler
→ Basketizer (socket) | Basketizer (socket) | CPU CPU (socket / memory) — node

Chart (right): CMS example, 8 threads on XEON®E5-2630, 2x8 cores

- 2 propagators — NUMA: 12.8
  - CPU #0 / CPU #1
  - P #0 / P #1
- 2 propagators — SYS: 13.9
  - CPU #0 + CPU #1
  - P #0 / P #1

Axis: 12.2 12.4 12.6 12.8 13 13.2 13.4 13.6 13.8 14

time[s]
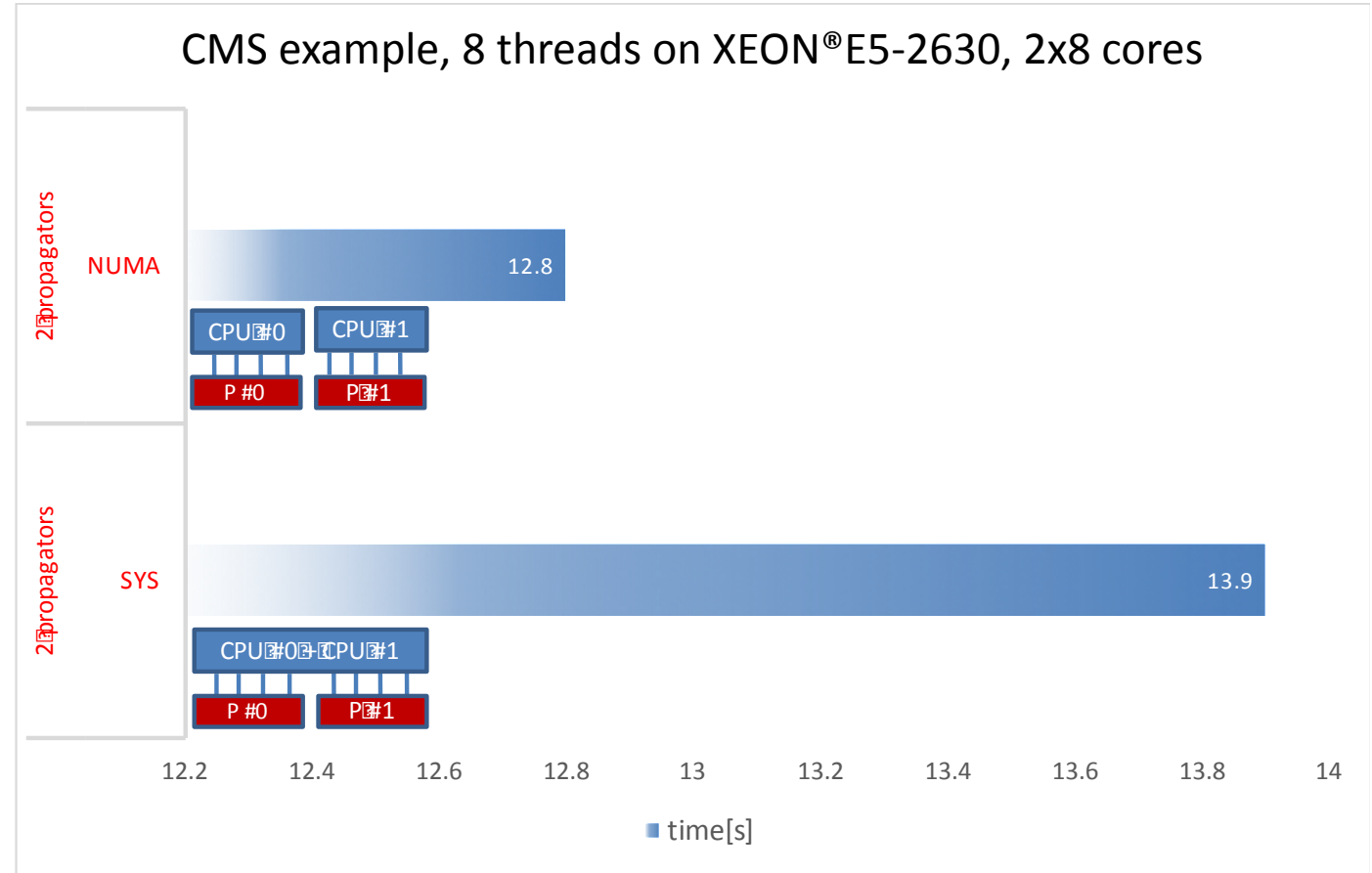
Accelerators: integration with the "basket" model

- Portability: CUDA as a backend

- GPU baskets and performance
  - CPU Intel Xeon E5 (1 core @2.6GHz)
  - NVidia K20 GPU (2096 cores @0.7GHz)
  - EM models (sampling final states)
  - **Performance (including transfer host->device) vs. number of tracks**
  - Potential of *x30* on GPU, but requires 10^4 tracks per process

- Lessons:
  - Portability is feasible, but does not come for free
  - efficiency comes with very large baskets, which are difficult to maintain
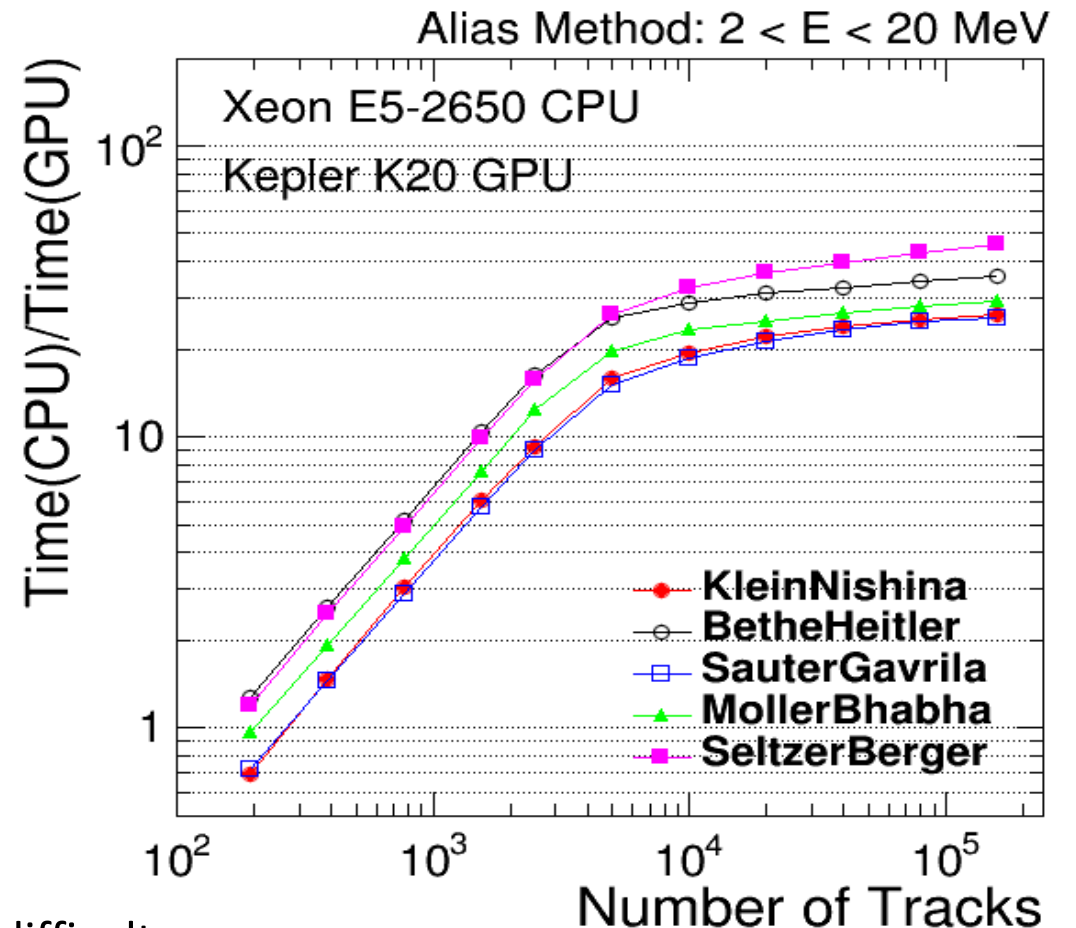
Host-device transfer included

6. Lessons learnt

How could it be done better?

Concurrency "lessons"

- The more thread-local, then the data flow is better
  - Keep tracks in the same thread, with minimum stealing
  - Avoid high contention on common data containers
    - Concurrent basketizing has high price, baskets w/ high populations must be thread local
- A multi-threaded, multi-basketized flow becomes inefficient on event tails
  - Partial baskets have to be flushed in scalar mode to sustain the data flow
- Track-level parallelism is the path to instruction-level parallelism
  - cuts event tails, but has large price
  - Multiple events in flight, but owned by a thread - is it a good compromise?

Conclusions from CMSSW integration

- CMS studies met ~all goals laid out
  - Co-development led to improvements and bug fixes in GeantV to facilitate experiments' use
  - One of the first projects to exercise CMSSW ExternalWork feature
  - Physics validation & CPU measurements show very positive results
  - Path to adapt interfaces efficiently is laid out
- Demonstrator to test major elements of GeantV-CMSSW integration is ready
  - Up to 2.6× speedup in CMSSW application
  - More efficient use of CPU caches in GV seems to translate in improved performance within CMSSW
  - The CMS simulation group thanks the GeantV R&D team for providing support to this integration exercise and making it a successful co-development endeavor.

Main lessons from physics vectorization

- There is **no generic solution** to achieve speedup
  - The best approach is often a compromise
    - E.g. choosing sampling method to be used, **depending** on scalar/vector, but also on energy or material composition

- Complex code can be also vectorized, but it has to have a sufficient hotspot to be worth it
  - There are "important" and "less important" models depending on the simulation, ranging from < 1% to 4-5% of the total time

- Compactness and more efficient data access actually brings much more benefits than vectorization for "small" hotspots

More open questions...

- Benchmarked CMS only, what about other setups?
  - ATLAS, LHCb, ALICE, upgrades, ...
- Still not fully understanding all sources of performance increase of GeantV
  - Would need extra time/resources/expertise
- Sharing tracks opens up fine grain parallelism, but extra communication hinders on performance: what is the best trade-off?
- How to improve both instruction and data locality, is it even possible?
  - Needs rethinking the data model and access patterns
  - Can data load/store be substantially reduced?

Main lessons (1)

- Main factors in the speedup seem to include
  - Better cache use (single track mode shows performance decrease for small caches)
  - Tighter code (e.g., less indirections and branching)
- Vectorization's impact (much) smaller than hoped for
  - Basketization can bring benefits for FP hotspots (e.g. magnetic field, multiple scattering)
  - Small fraction of the code has been vectorized or is run in vector mode effectively
  - Overhead of basketization cost similar to vector gain for "small" modules
- Basketization cost in
  - Either extra memory copy (using collection of tracks)
  - Or lower memory access coherency (using collection of pointers)

Main lessons (2)

- Geometry navigation not (yet?) vectorized and introduces a bottleneck (Amdahl)
  - Mainly due to the end-of-event track collection/gathering from the 'rarely' used volumes
  - Should decrease for larger track multiplicities
  - But no guarantee that these volumes internally vectorize
- Code/Algorithm needs to be designed from the ground up for vectorization for best results
  - Compact code, compact data fitting caches and being reused
- Actual upper limit on potential vectorization gains are still to be fully understood
  - including whether different approaches and **trade-offs** in the physics code implementation could bring extra computing performance
  - Including AVX512 that was not tested due to not working backends

Conclusions (1)

- Innovative and disruptive R&D allowing to investigate novel technologies and approaches to simulation
  - Allowed to improve performance-critical code in the simulation chain
  - Still more room for improvement
- Showed the significant impact of good CPU cache behavior (and the challenge of measuring this effect)
  - Further research warranted to see if this can be exploited even further
- Basketization gains overshadowed by associated costs mainly due to data copy/management overheads.
  - Nonetheless might still benefit other workflows (e.g. pipelines)
  - Balance might be different under different conditions (e.g. larger multiplicities)

Conclusions (2)

- Amdahl's law applies to vectorization too :(
- Lessons learned will be useful for GPU architecture investigations
- GeantV allowed to venture into interesting and ambitious R&D paths and set new expectations for the future, both in terms of potential gains and the cost of achieving them

Thank you!

Thanks to all the GeantV collaborators contributing to different parts of this R&D

Backup slides

More detailed info about the different topics

Why use 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™ and Skylake Xeon™ processors.

Vectorization of ROOT Mathematical Libraries — CHEP 2018, Sofia, Bulgaria
Reference: https://indico.cern.ch/event/587955/contributions/2938041/attachments/1683418/2706263/Vectorization_chep2018.pdf
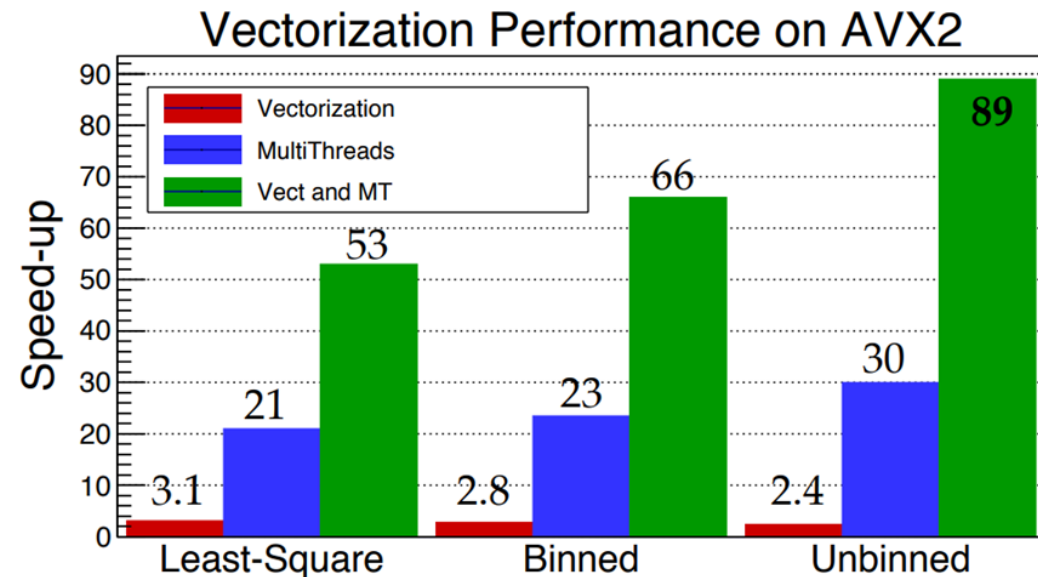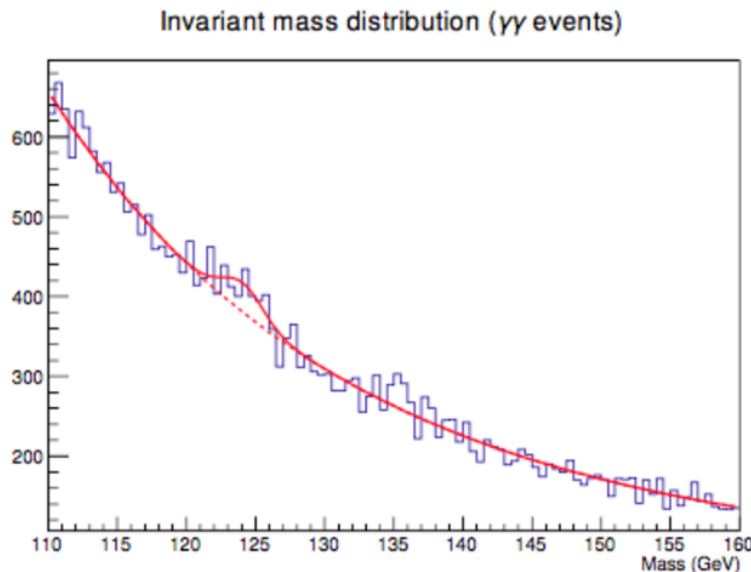
SIMD Programming Models

- Auto-vectorization

- OpenMP 4.1

- Compiler Pragmas

- SIMD Library

- Compiler Intrinsics

- Assembly

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

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

```cpp
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];
```

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

a = b * c;
```

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

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

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

Why did we need VecCore?

▶ Unreliable performance with auto-vectorization

- `https://godbolt.org/g/bjQzbA` (change int to bool)

- `https://godbolt.org/g/R6fXAw` (change -O1 to -O3)

▶ Compiler intrinsics are not an ideal interface

- Limited to C name mangling, so portability is an issue

▶ Libraries do not work well across all architectures

- UME::SIMD is best on KNL, but Vc is better for Skylake

- ARM support only in UME::SIMD, but poor performance

▶ Portable solution for when no library is available

- For example, on PowerPC

```
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
```
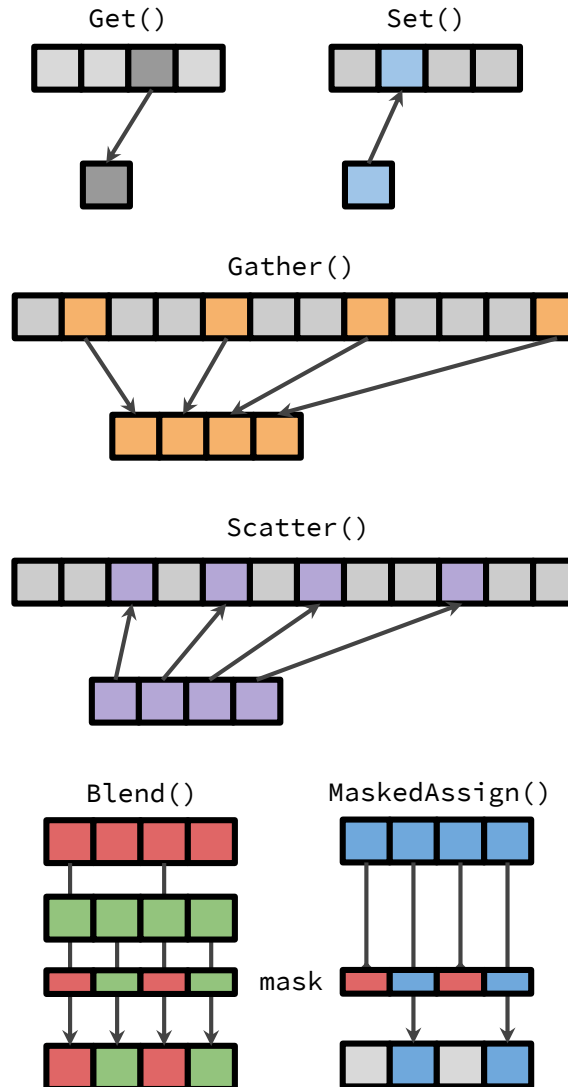

68

VecCore Example: Mandelbrot Set

Iterate

$f(z) = z^2 + c$

N times and check if *z* diverges

Example included in VecCore
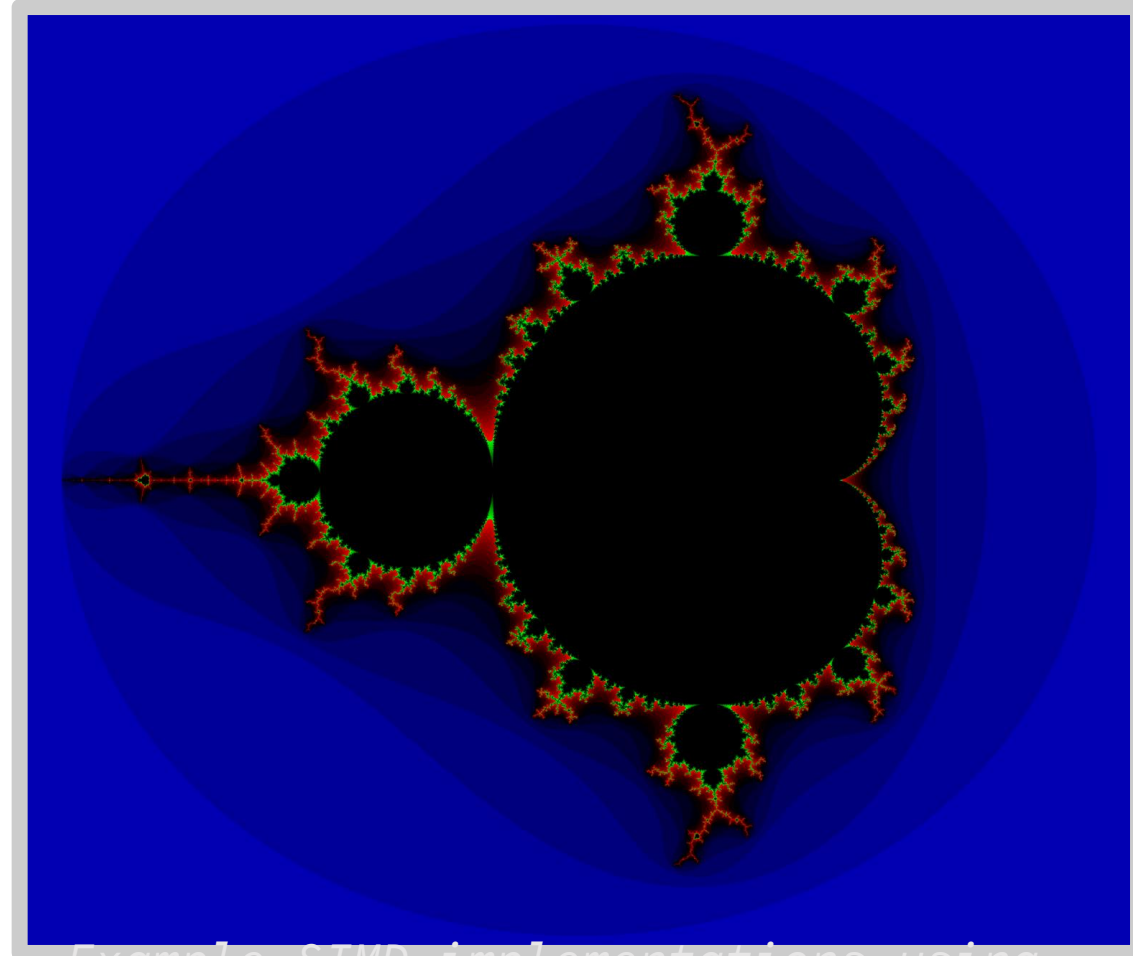
Example SIMD implementations using intrinsics:

https://github.com/skeeto/mandel-simd

Shows speedup of 5.8x with AVX

VecCore Example: Mandelbrot Set

Iterate

$$f(z) = z^2 + c$$

N times and check if z diverges

Example included in VecCore

Scalar Implementation

```cpp
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 Example: Mandelbrot Set

Iterate

$$f(z) = z^2 + c$$

N times and check if z diverges

Example included in VecCore

VecCore Implementation

```cpp
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);
        }
    }
}
```

Performance of Mandelbrot Set

Runtime (ms)		Intel Core i7 6700			Intel Xeon Phi 7210	
		GCC-7.2	Clang-5.0	ICC-18.0	GCC-7.2	ICC-18.0
Single Precision	Scalar	550	549	677	3415	3609
	Scalar Backend	570	569	677	3353	3510
	Vc 1.3.3	110	110	126	1064	1160
	UME::SIMD 0.8.1	117	117	131	–	543
Double Precision	Scalar Algorithm	548	548	672	3409	3602
	Scalar Backend	571	571	674	3348	3502
	Vc 1.3.3	267	267	257	2101	2087
	UME::SIMD 0.8.1	421	421	422	–	846

Speedup vs Scalar						
Single Precision	Scalar Backend	0.96	0.96	1.00	1.02	1.03
	Vc 1.3.3	5.00	4.99	5.37	3.21	3.11
	UME::SIMD 0.8.1	4.70	4.69	5.17	–	6.65
Double Precision	Scalar Backend	0.96	0.96	0.99	1.02	1.03
	Vc 1.3.3	2.05	2.05	2.61	1.72	1.73
	UME::SIMD 0.8.1	1.30	1.30	1.59	–	4.25

Note: "Scalar" above has SSE2 enabled, single precision time with SSE2 disabled with GCC-7.2 is 764ms.

Reference: https://indico.cern.ch/event/567550/papers/2700128/files/6152-

Effect of branching on SIMD performance

Iterate $f(z) = z^2 + c$, where
$c = 0.7885\ e^{i\alpha}$ and $\alpha \in [0, 2\pi]$
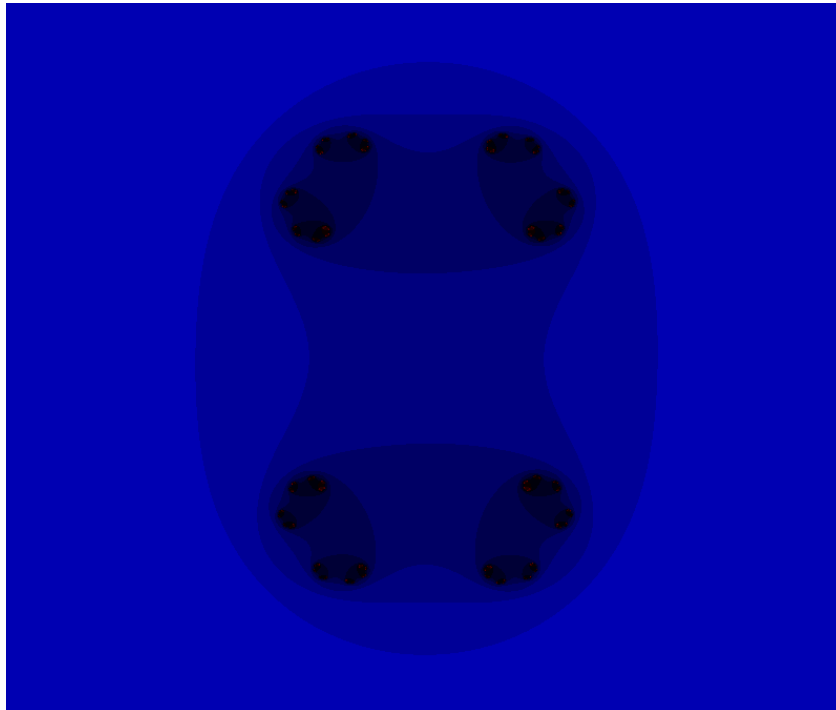
Julia Set
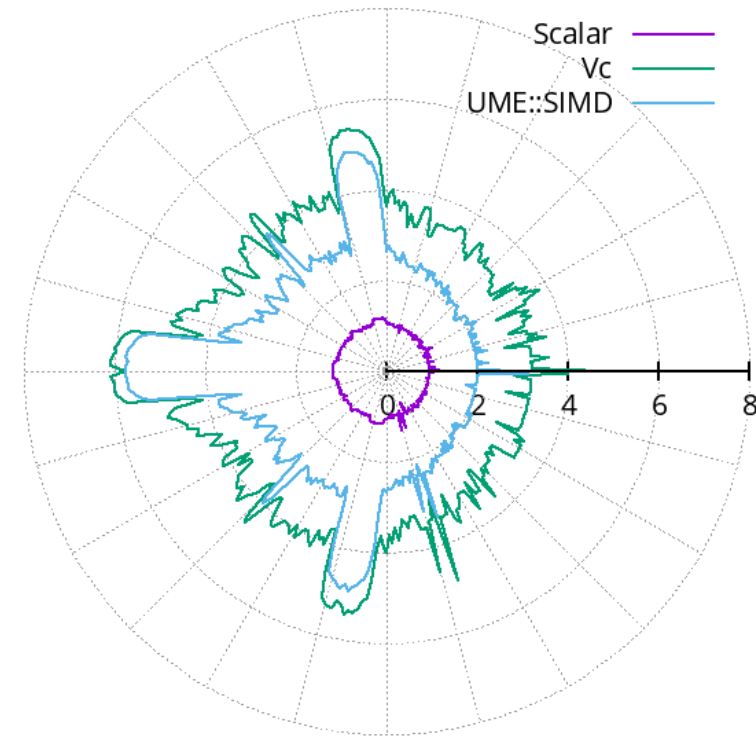
Speedup

Code Sample: VecGeom Box

Box Implementation of `DistanceToIn()`

```cpp
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);
 }
```

Performance of VecGeom Box Algorithms

Box Benchmark — Intel® Core™ i7-6700 CPU 3.40GHz (Skylake)
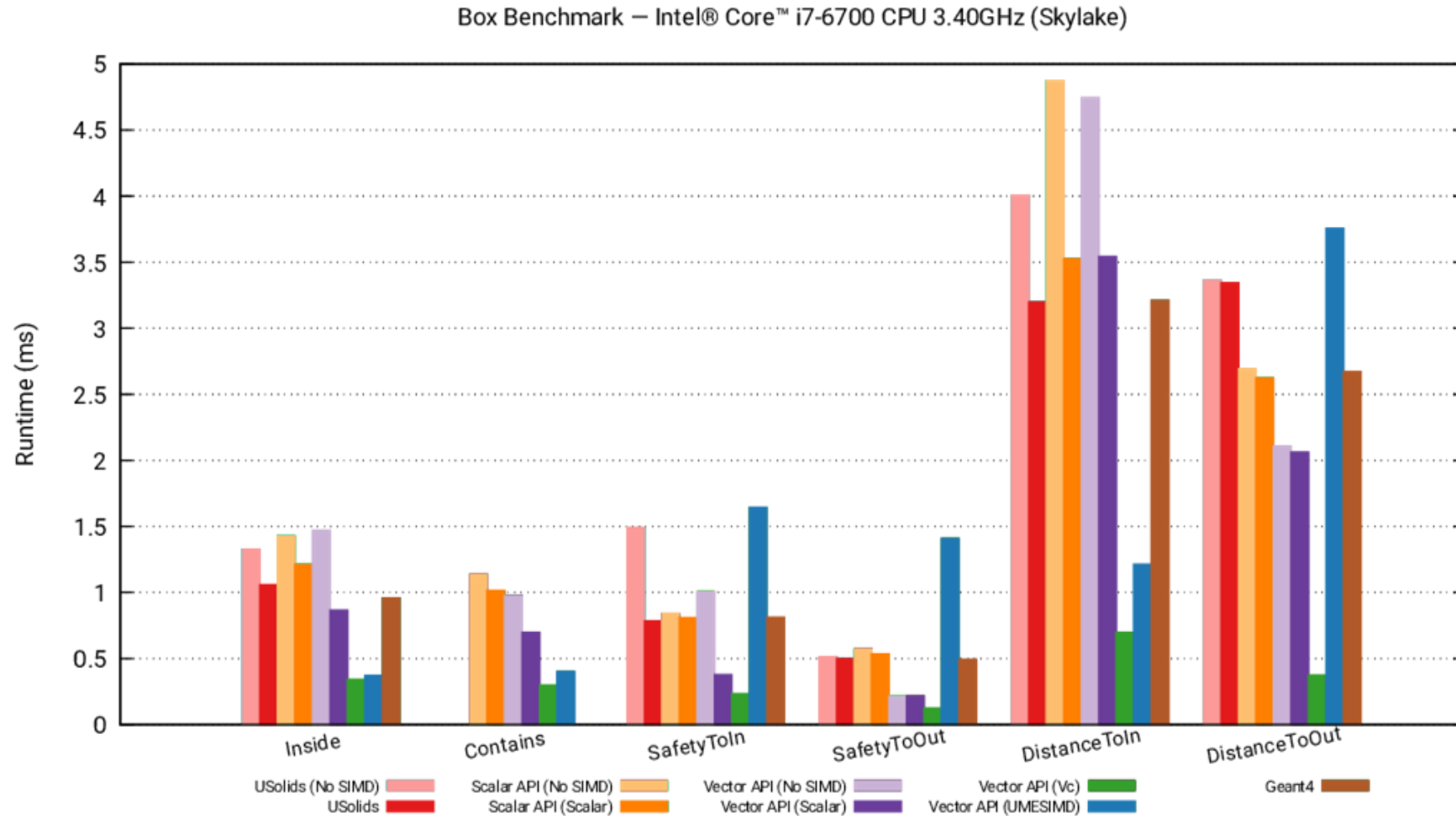
VecGeom Speedups on Knights Landing
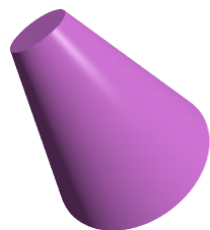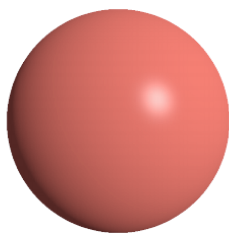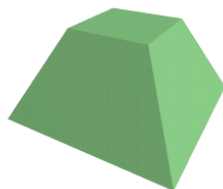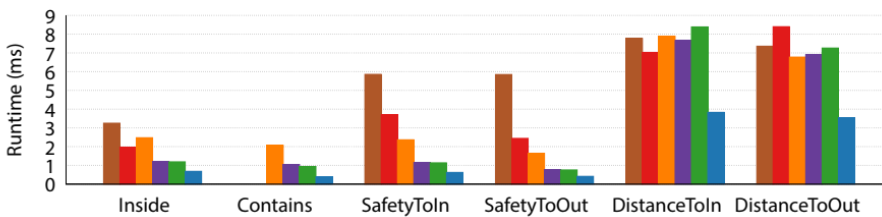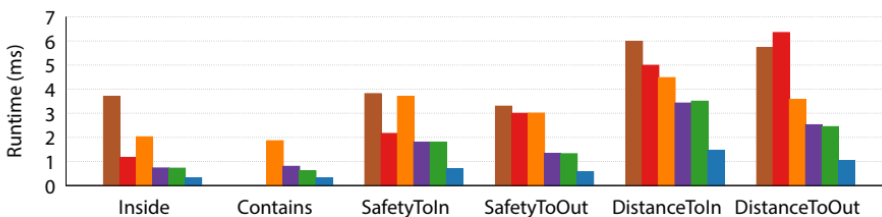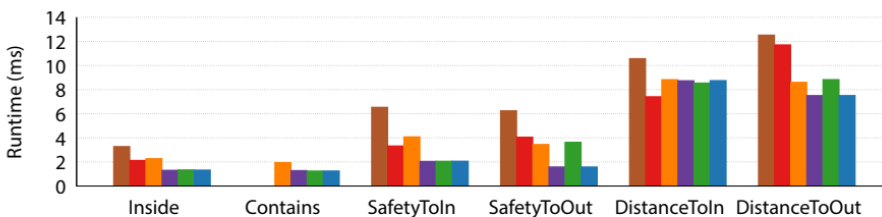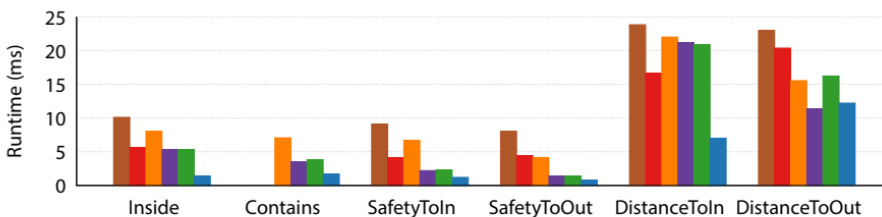
Field propagation overview

x_0, p_0
x_1, p_1, Δx, Δp
x_2, p_2, Δx, Δp

- Field propagation involves solution of Ordinary Differential Equation
  - Typically Runge-Kutta methods are used (as in Geant4)
- In GeantV created **vectorised Runge-Kutta** propagation
  - Charged tracks in a basket are sent to the FieldPropagation classes
  - Vectorised over tracks
- Challenges are
  - To use mostly vector operations
  - To ensure that all vector lanes are doing useful work

- Motion in field requires solving ODE for endpoint **x**, **p** after length s
- Runge-Kutta step: evaluate B-field, estimate **x**, **p**, **Δx**, **Δp**
- Successful if $|\Delta x| < \varepsilon$ s & $|\Delta p| < \varepsilon |p|$
- Each step of a Runge-Kutta algorithm is easy to vectorise
  - But different tracks (vector lanes) can take different number of iterations to finish integration
  - The 'driver' class which calls the RK 'stepper' must play coordinate the work

Vector propagation in Field–v2

- A step can either
  - Fail,
  - Succeed but not get to the end ("ok")
  - Finish the integration ("done")

- Driver rewritten to use
  - Tight loop with all lanes integrating until at n>=threshold reach the end of interval
  - Reload lanes with new work.

- Profiled with (semi-)realistic RZ field
  - Interpolated from sampled CMS field

Call # 1

Call # 2

Call # 3

1
2
3
4
5
6
7

start
failed
ok
done
keep

Load 1 track

Load 2 tracks

78
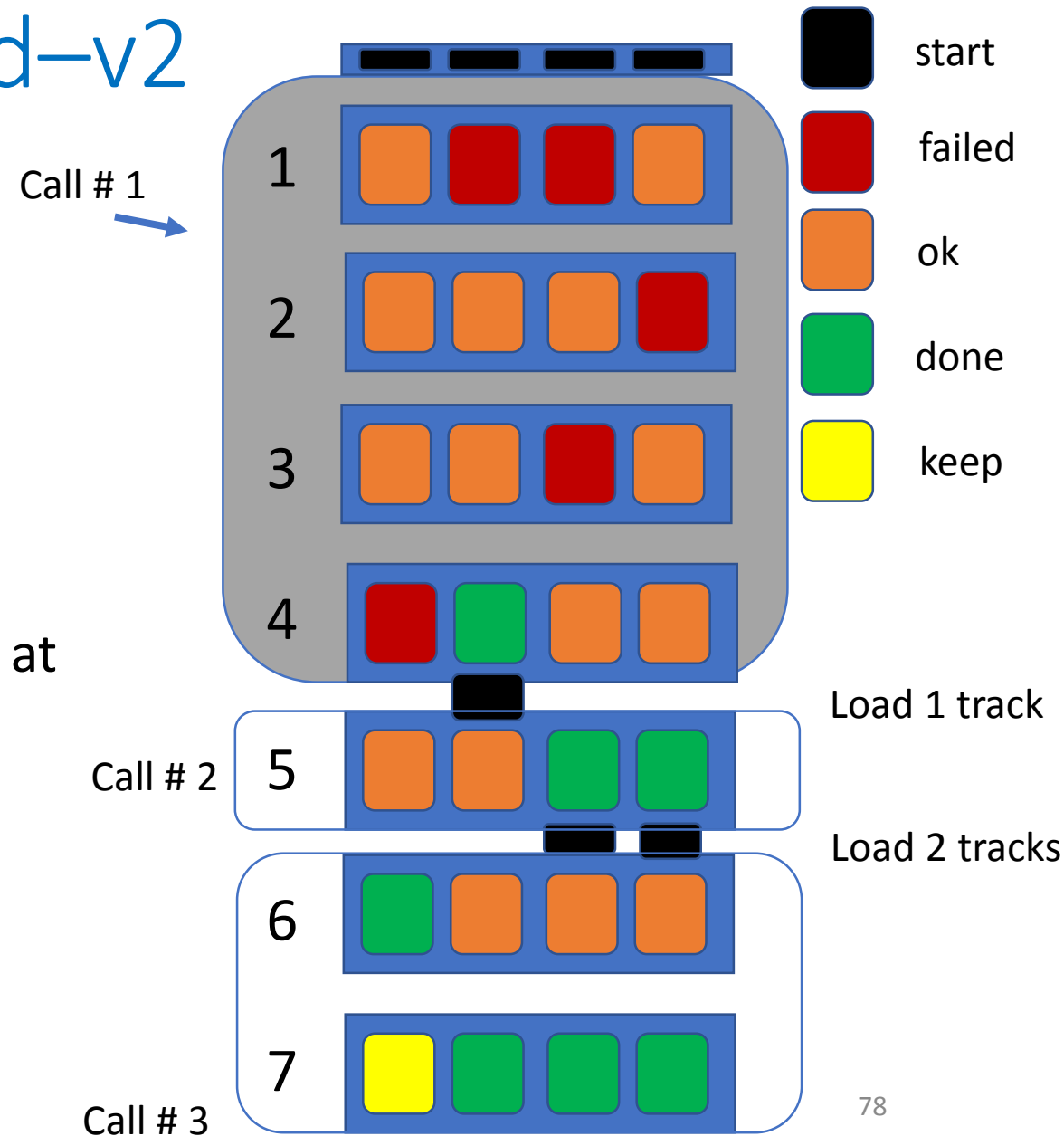
Performance with baskets for field only

Basket size	16	32	64	128	256	512	1024
Unused lanes	0.186	0.130	0.073	0.039	0.023	0.015	0.007
Unused lanes (reordering)	0.140	0.066	0.025	0.003	0.002	0.001	0.001

Fraction of lanes which have finished integration.

Event window	Tracks/event	Basket sz= 16	32	64	128	256	512	1024
16	16	3.4 (2) %	**5.6 (2) %**	4.0 (2) %	4.1 (2) %	5.2(2) %	3.6 (2) %	1.6 (2) %
1	16		4.7 (2) %	5.0 (2) %	5.5 (2) %	6.8(2) %	**7.0 (3) %**	
1	8 reordering	6.6 (3) %	5.3 (3) %	6.9 (3) %	7.2 (4) %	7.9(4) %	**8.2 (3) %**	7.3(3) %

Benchmarks on 1 thread of *MacBook Pro 2016, 2.6 GHz Core i7 6700HQ (Skylake),* 16 GB LPDDR3 2133MHz RAM, with clang from Xcode 10.1.
Baseline is "Basket off" configuration with 16 event window with 16 tracks/event (10 GeV e-).
'Reordering' means bringing forward tracks with length / radius(curvature) over threshold (=1.5)

EM Physics

Backup slides

Vectorized EM physics models – Intro

- A **simulation step**, limited by a discrete physics interaction, can be divided into **two** distinct parts
  - **Select the physics interaction** with the corresponding interaction point:
    - Driven by the integrated cross section values of the physics
    - Cross section table lookups and interpolations (Memory bounded operations with very little mathematical computations)
  - **Invoke the interaction** (**final state sampling**):
    - Computation of the post-interaction kinematical state of the primary particle
    - Generation of possible secondary particles
    - Contains significantly more mathematical operations (CPU bounded)
- The **final state computation** includes generation of **stochastic variables** from their probability distributions determined by the corresponding differential (in energy, angle) cross sections (DCS) of the underlying physics interaction
  - **Composition-rejection** method is typically used in Geant4 to sample from these PDFs

Sampling techniques for the Final State generation

- **Rejection Sampling:**
  - Unpredictable n. of loop executions: Exit condition depends on the outcome of the stochastic variable
  - Non deterministic behavior for the different tracks filled into the vector register, resulting in undesired divergence and eventually loss of potential computational gain
  - Solution: lane refilling

- **Table Sampling:**
  - **Alias** method efficiently generates samples of discrete stochastic variables
  - An intermediate **discrete random variable** is introduced by partitioning the range of the original continuous PDFs into distinct intervals
    - new discrete variable is the probability of having the original continuous variable lying in a given interval
  - **Appropriate partitions** of the original variable range and/or **variable transformations** are used to transform the original PDF to a smooth function, in order to have a linear behavior of the PDF over each discrete interval

82

Vectorized EM physics models

[1]

- Architecture: Intel Haswell
Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

Speedup from Vectorization of the **Table Sampling**

Speedup from vectorization of the **Rejection Sampling**

Speedup of **vectorized Table Sampling** w.r.t **scalar Rejection**
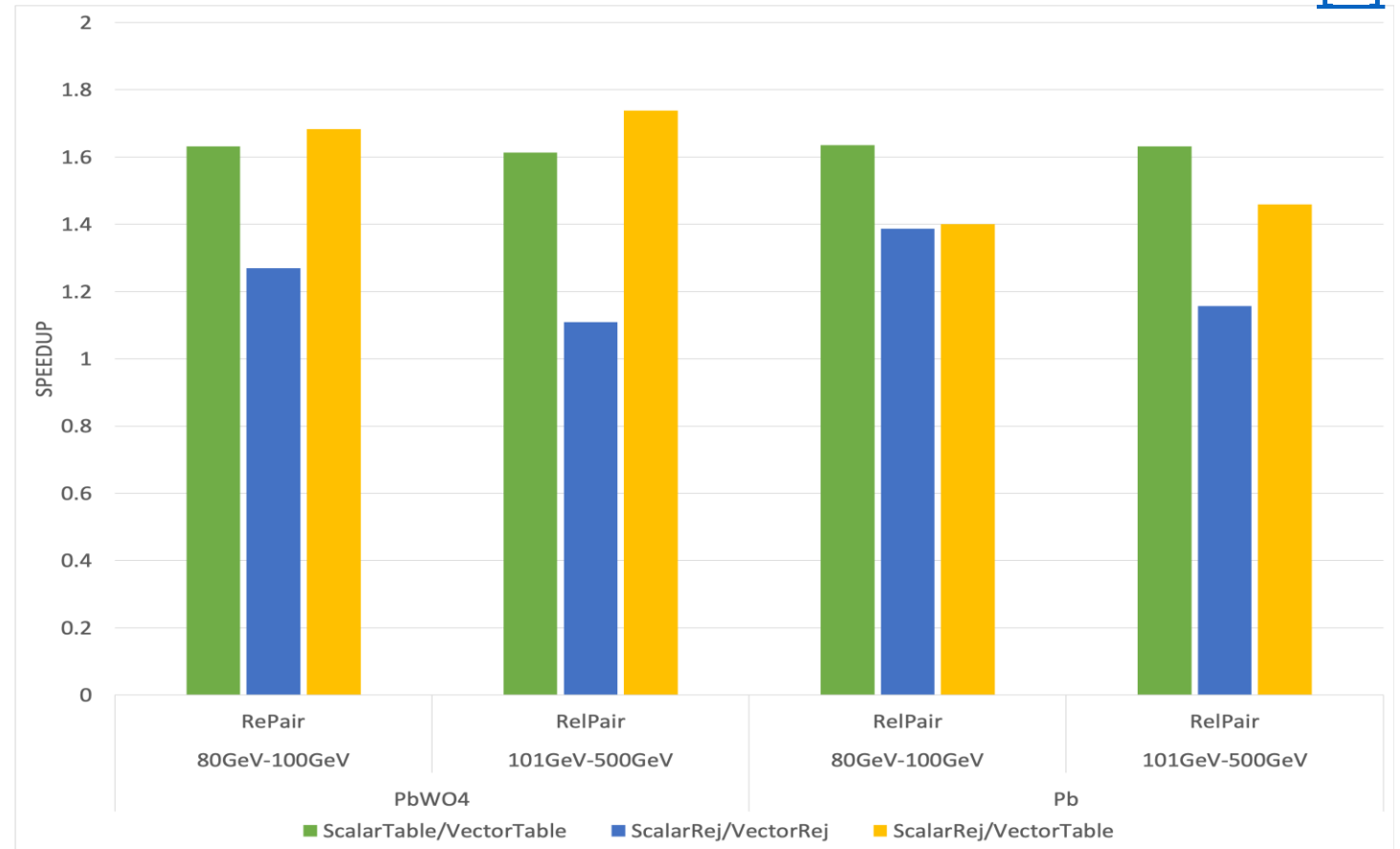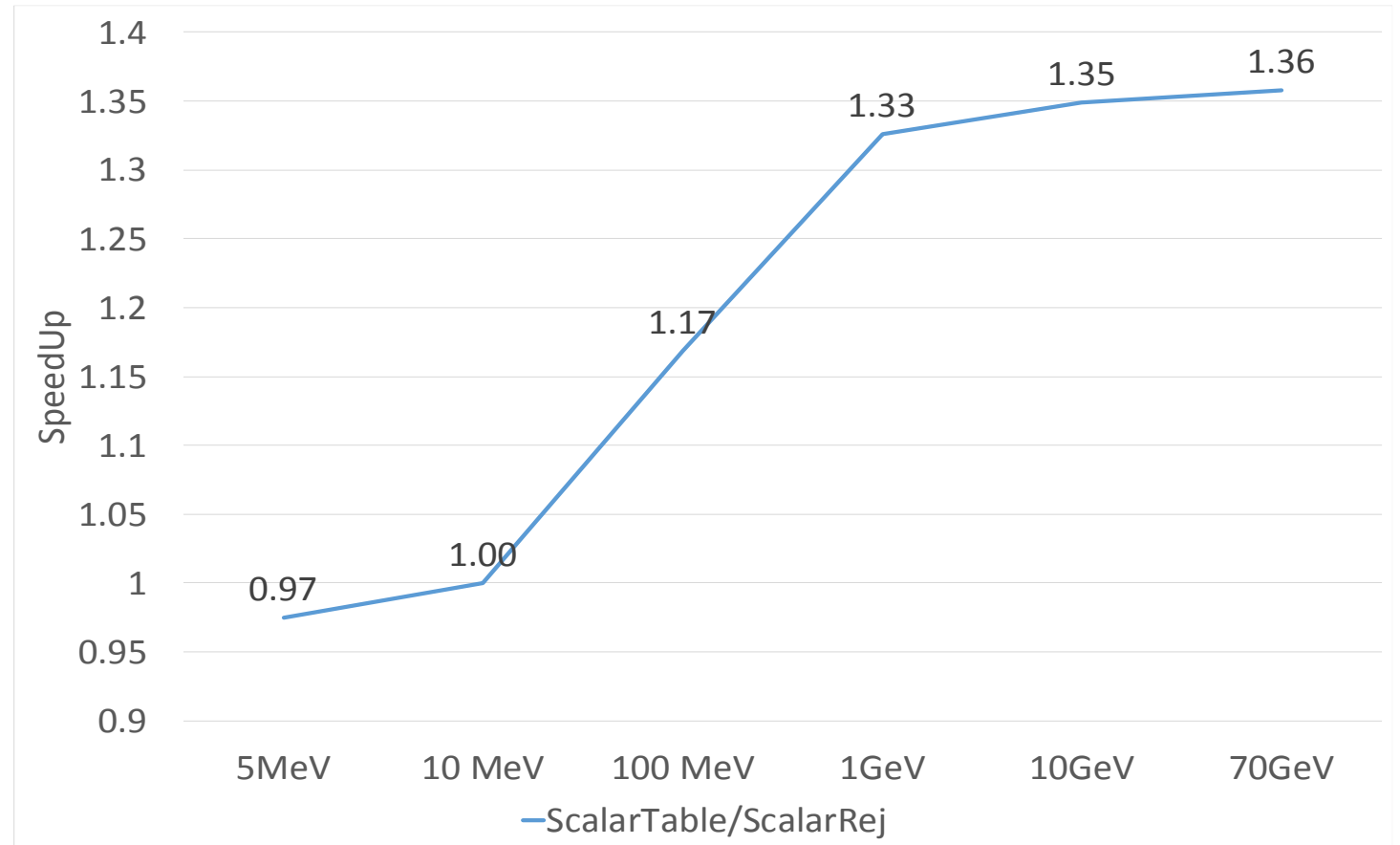

M. Bandieramonte, M. Novak

Speedup of the final state generation of different electromagnetic physics models obtained with SIMD vectorization in case of different sampling algorithms. The results were obtained by using Google Benchmarks on an Intel[R] Haswell Core[TM] i7-6700HQ, 2.6 GHz, with Vc backend and AVX2 instruction set processing 256 tracks.

83

Vectorized EM physics models

- Architecture: Intel Haswell
Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

Speedup from Vectorization of the **Table Sampling**

Speedup from vectorization of the **Rejection Sampling**

Speedup of **vectorized Table Sampling** w.r.t **scalar Rejection**

M. Bandieramonte, M. Novak

Microbenchmark results for final state generation in case of the high energy e_-/e_+ pair production model under different primary energy (80-100 [GeV] and 101-500 [GeV]) and target material conditions (left: PbWO₄, right: Pb).

Vectorized EM physics models

- Architecture: Intel Haswell
Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

Speedup of the *rejection* based final state sampling compared to the sampling *table* based one in case of the Bethe-Heitler e_-/e_+ pair production model, as a function of the primary γ particle energy.

M. Bandieramonte, M. Novak

Vectorized EM physics models – lesson learned

- Main message: there is **no generic solution** to achieve speedup
  - **Final state EM speedup**: between **1.5-3** on Haswell, **2-4** on Skylake with **AVX2**
  - **We never tested vectorized EM physics with AVX512 (speedup expected to be doubled): lack of person power**
- The **computational diversity** of the physics models directly implies a variation of the optimal algorithmic solution as a function of the models.
- In addition, the **dependence** of the underlying physics on some **external conditions** such as target material composition or primary particle energy introduces further variations
- In order to **maximize the achievable gain** from vectorization it's necessary to **profile** all the available final state generation algorithms as a function of:
  - The complexity of the underlying DCS
  - Target material composition
  - Primary particle energy
- **Due to lack of person power the profiling activity was not completed**

MC truth

Backup slides

GeantV kinematics output (MC truth)

- handling of MC truth is problematic per se
  - which particles to store, how to keep connections, where to connect hits

- multithreading adds the complexity
  - order of processing of particles is 'random'
    - processing of 'daughter' particle may be completed before 'mother' particle 'end of life'
  - events need to be 'put together' after parallel processing

MC truth

- we can't (and we don't need) to store all particles
  - typically no delta-e, no low-E gamma showers, etc needed
- we need to store particles necessary to understand the given event (process)
- we need to store particles to associate hits
- in all cases, we need to (re)connect particles to have consistent event trees

MC truth handling requirements

- no MC truth-handling strategy is perfect, nor complete, but:
  - we need to give user a way to decide
- transport need to provide/allow
  - links between mother and daughter particles
  - the possibility to flag particles as 'to be stored'
  - possibility to introduce 'rules' what to store
  - a way to 'reconnect' tracks and hits if some are skipped
    - if we don't store a particle, we need to update the daughter particles to point back to the last stored one in the chain
- for the final output we need to have some event record
  - for our proof of principle, we can start with HepMC

MC truth handling architecture

- light coupling to transport
  - minimal 'disturbance' to transport threads
  - maximal flexibility of implementing custom particle history handlers
- interface provided by MCTruthMgr
  - receives (concurrent) notifications from transport threads about
    - adding (primary or secondary) new particles
    - ending particles
    - finishing events
  - delegates processing of particles history to concrete MC truth implementation

MC truth infrastructure and users code

- MCTruthMgr provides interface and underlying infrastructure for particles history
  - light-weight transient, intermediate event record
- users code:
  - decision making (filtering) algorithm
  - conversion to users' event format
- concrete example implementation provided based on HepMC3

MC truth call sequence
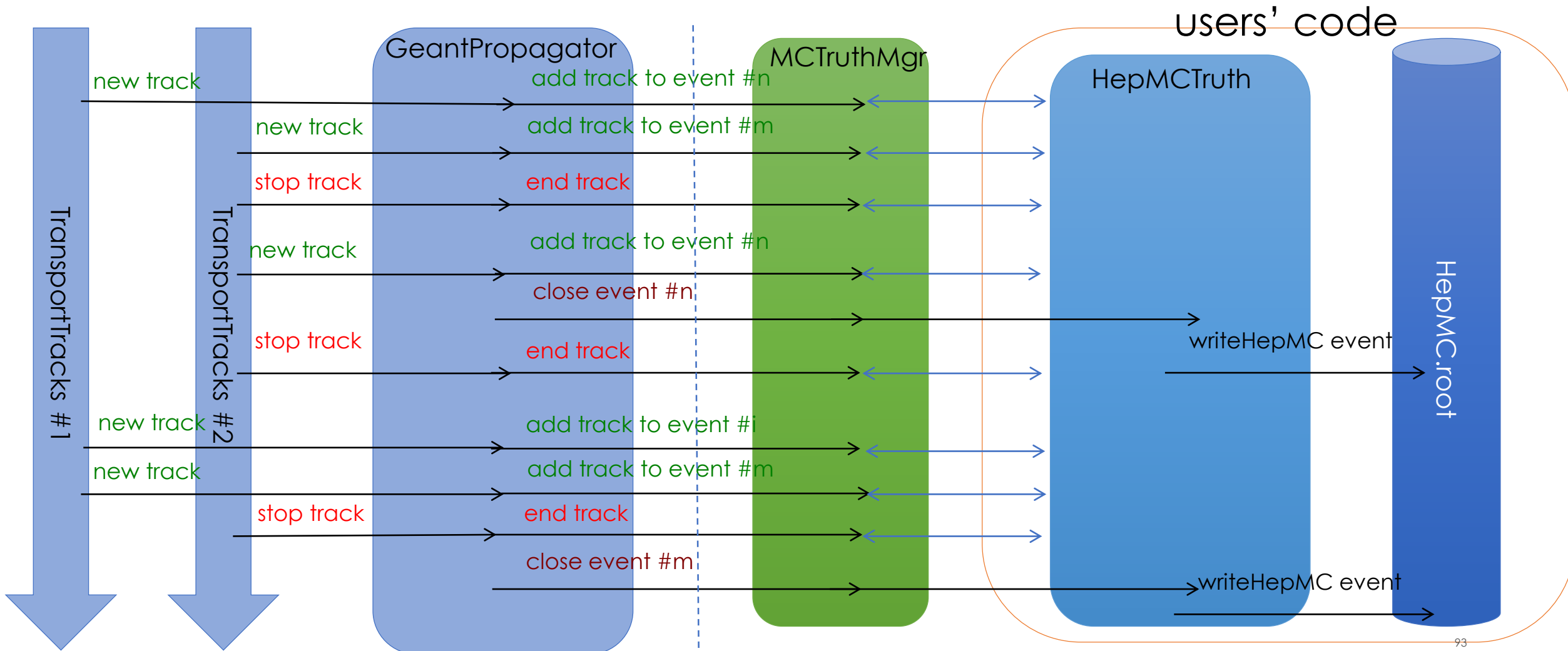
MC truth output status

- GeantV MC truth manager provides handles to deal with particles history
  - allows 'physics' studies
  - first implementation, further iterations possible to look in detail at performance
- example implementation based on HepMC3 provided
- further performance testing/improvements in highly concurrent environment to be studied

Vectorized EM physics

Backup slides

Vectorized EM physics models – Intro

- A **simulation step**, limited by a discrete physics interaction, can be divided into **two** distinct parts
  - **Select the physics interaction** with the corresponding interaction point:
    - Driven by the integrated cross section values of the physics
    - Cross section table lookups and interpolations (Memory bounded operations with very little mathematical computations)
  - **Invoke the interaction** (**final state sampling**):
    - Computation of the post-interaction kinematical state of the primary particle
    - Generation of possible secondary particles
    - Contains significantly more mathematical operations (CPU bounded)
- The **final state computation** includes generation of **stochastic variables** from their probability distributions determined by the corresponding differential (in energy, angle) cross sections (DCS) of the underlying physics interaction
  - **Composition-rejection** method is typically used in Geant4 to sample from these PDFs

Sampling techniques for the Final State generation

- **Rejection Sampling:**
  - Unpredictable n. of loop executions: Exit condition depends on the outcome of the stochastic variable
  - Non deterministic behavior for the different tracks filled into the vector register, resulting in undesired divergence and eventually loss of potential computational gain
  - Solution: lane refilling

- **Table Sampling:**
  - **Alias** method efficiently generates samples of discrete stochastic variables
  - An intermediate **discrete random variable** is introduced by partitioning the range of the original continuous PDFs into distinct intervals
    - new discrete variable is the probability of having the original continuous variable lying in a given interval
  - **Appropriate partitions** of the original variable range and/or **variable transformations** are used to transform the original PDF to a smooth function, in order to have a linear behavior of the PDF over each discrete interval

9
7

Vectorized EM physics models

- Architecture: Intel Haswell Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

■ Speedup from Vectorization of the **Table Sampling**

■ Speedup from vectorization of the **Rejection Sampling**

□ Speedup of **vectorized Table Sampling** w.r.t **scalar Rejection**

In KleinNishina model, Rejection Sampling is faster both in scalar and vector mode

In Relativistic Pair model, Table Sampling is faster both in scalar and vector mode

M. Bandieramonte, M. Novak

Speedup of the final state generation of different electromagnetic physics models obtained with SIMD vectorization in case of different sampling algorithms. The results were obtained by using Google Benchmarks on an Intel_R Haswell Core_TM i7-6700HQ, 2.6 GHz, with Vc backend and AVX2 instruction set processing 256 tracks.

Vectorized EM physics models

- Architecture: Intel Haswell
Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

Speedup from Vectorization of the **Table Sampling**

Speedup from vectorization of the **Rejection Sampling**

Speedup of **vectorized Table Sampling** w.r.t **scalar Rejection**
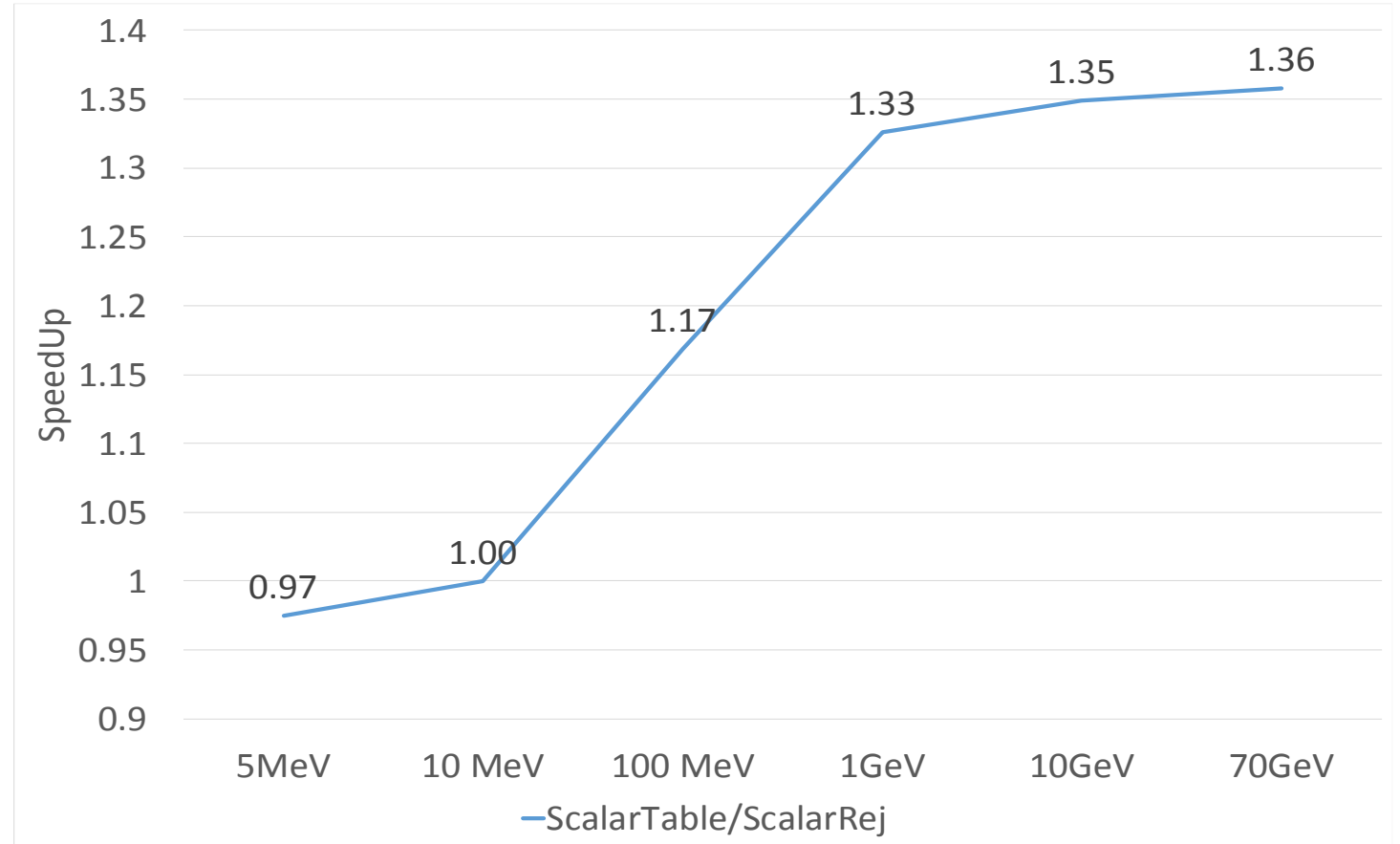

M. Bandieramonte, M. Novak

Microbenchmark results for final state generation in case of the high energy e_-/e_+ pair production model under different primary energy (80-100 [GeV] and 101-500 [GeV]) and target material conditions (left: PbWO$_4$, right: Pb).

Vectorized EM physics models

- Architecture: Intel Haswell
Core i7-6700HQ, 2.6 GHz
- Instruction Set: AVX2
- Backend: Vc
- Detector: Lead
- #baskets: 256
- Run with GoogleBenchmarks

Speedup of the *rejection* based final state sampling compared to the sampling *table* based one in case of the Bethe-Heitler e_-/e_+ pair production model, as a function of the primary γ particle energy.

M. Bandieramonte, M. Novak

Vectorized EM physics models – lesson learned

- Main message: there is **no generic solution** to achieve speedup
  - **Final state EM speedup**: between **1.5-3** on Haswell, **2-4** on Skylake with **AVX2**
  - **We never tested vectorized EM physics with AVX512 (speedup expected to be doubled): lack of person power**
- The **computational diversity** of the physics models directly implies a variation of the optimal algorithmic solution as a function of the models.
- In addition, the **dependence** of the underlying physics on some **external conditions** such as target material composition or primary particle energy introduces further variations
- In order to **maximize the achievable gain** from vectorization it's necessary to **profile** all the available final state generation algorithms as a function of:
  - The complexity of the underlying DCS
  - Target material composition
  - Primary particle energy
- **Due to lack of person power the profiling activity was not completed**

5. Performance results

Backup slides

Performance Comparisons: Tested Platforms

- Process-Cores-CPU[GHz]-Memory[GB]-Cache[MB]-SIMD

Processor	Core	CPU	Mem	Cache	SIMD
Intel E2620 (Sandy Bridge)	2x6	2.0	32	15	AVX
Intel E2680 (Broadwell)	2x14	2.4	128	35	AVX2
AMD 6128 (Opteron)	4x8	2.3	64	15	SSE4

- Cache size

Processor(*)	L1 set	L2 set	L3 set
AVX-2.0-15	6x32 KB 8-way	6x256 KB 8-way	15 MB 20-way
AVX2-2.4-35	4x32 KB 8-way	14x256 KB 8-way	35 MB 20-way
SSE4-2.3-15	8x64 KB 2-way	8x 512 KB 16-way	2x6 MB

  - * Processor Convention: SIMD-CPU-Cache

Locality

- Single track mode (strk)
  - Emulate Geant4 style tracking
  - A meaure of locality: GeantV (strk)/GeantV(default)
- CPU Time in [sec] and their ratios

Processor	GeantV	GeantV-strk	strk/default
AVX-2.0-15	2621	2960	1.13
AVX2-2.4-35	1628	1533	0.94
SSE4-2.3-15	4457	4817	1.08

- The data locality does not explain the performance difference between Geant4 and GeantV (scalar)

Hardware Counters: L2 Cache and L3 Cache

- L2 cache miss (~12 cycles): in [Billion] counters

Processor	GV (ICM)	G4(ICM)	GV (DCM)	G4(DCM)
AVX-2.0-15	20	36	89	46
AVX2-2.4-35	24	37	100	51
SSE4-2.3-15	16	3.3	55	8.9

- ICM (DCM) = Instruction (data) cache miss
- GeantV has less ICM and Geant4 has less DCM (AVX/AVX2)

- L3 cache miss (~38 cycles): in [1B] counters

Processor	GV (TCM)	G4(TCM)	GV (TCA)	G4(TCA)
AVX-2.0-15	1.9	0.19	109	80
AVX2-2.4-35	1.3	0.012	126	82
SSE4-2.3-15	N/A	N/A	N/A	N/A

- TCM (TCA): Total Cache Miss (Access)

Performance summary table (go to backup)

Normalized performance factor with respect to the Intel i7 2.5GHz taking into account the clock speed

CPU	OS	gcc	SIMD	Cache	GV	G4 [sec]
Intel i7 2.5GHz	Ubuntu 16.04	5.4.0	AVX2	8 MB	1 ± 0.01	1 ± 0.03
Intel Core i7-4510U 2GHz	Ubuntu 16.04	5.4.0	AVX	4MB	1.39 ± 0.01	0.98 ± 0.01
AMD A10-7700k	Fedora Workstation 29	8.2.1	AVX	4 MB	1.94 ± 0.01	2.48 ± 0.02
Intel R 1.8GHz	Fedora Workstation 29	8.3.1	SSE4	2 MB	2.95 ± 0.01	2.15 ± 0.01
Intel Centrino 2	Fedora Workstation 29	8.2.1	AVX±	4 MB	2,76 ± 0.01	3.75 ± 0.02
11AMD e-300	Ubuntu 18.10	8.2.0	SSE2	1 MB	Not Vc compatible	13.32 ± 0.01

Geant4 fluctuates more than GeantV over different tested platforms

Integration with experimental frameworks

Backup slides

Status of GeantV Integration in CMSSW

Kevin Pedro, Sunanda Banerjee

(FNAL)

October 4, 2019

Introduction

- Integration testing of GeantV w/ CMSSW has several goals:

  o Demonstrate benefits of co-development between R&D team & experiments

  o Exercise capabilities of CMSSW framework to interface with external processing (ExternalWork mechanism) and handle track-level parallelization in detector simulation

  o Measure any potential CPU penalties when running GeantV in CMSSW

  o Estimate cost of adapting to new interfaces and eventually migrating to new (and potentially backward-incompatible) tools such as GeantV

    ▪ Thinking forward to HPC/GPU solutions

- *Not* planning to migrate CMS simulation to GeantV

  o This is an R&D exercise

GeantV Integration Tests in CMSSW

- Repositories: install-geant, SimGVCore
- ✓ Generate events in CMSSW framework, convert HepMC to GeantV format
- ✓ Build CMSSW geometry natively and pass to GeantV engine (using TGeo)
- Using constant magnetic field, limited EM-only physics list
- ✓ Calorimeter scoring adapted
- ✓ Run GeantV using CMSSW ExternalWork feature:
  - o Asynchronous, non-blocking, task-based processing
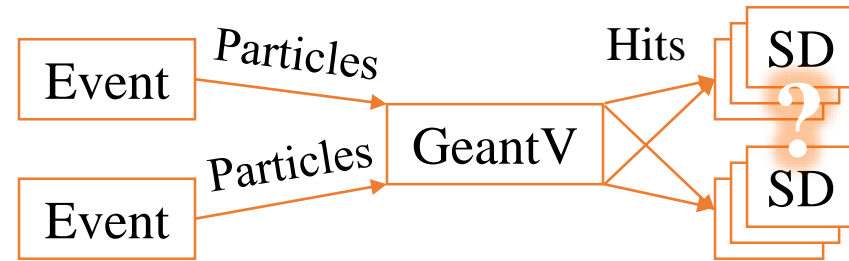- ✓ Output in CMS format, immediately suitable for digitization etc.
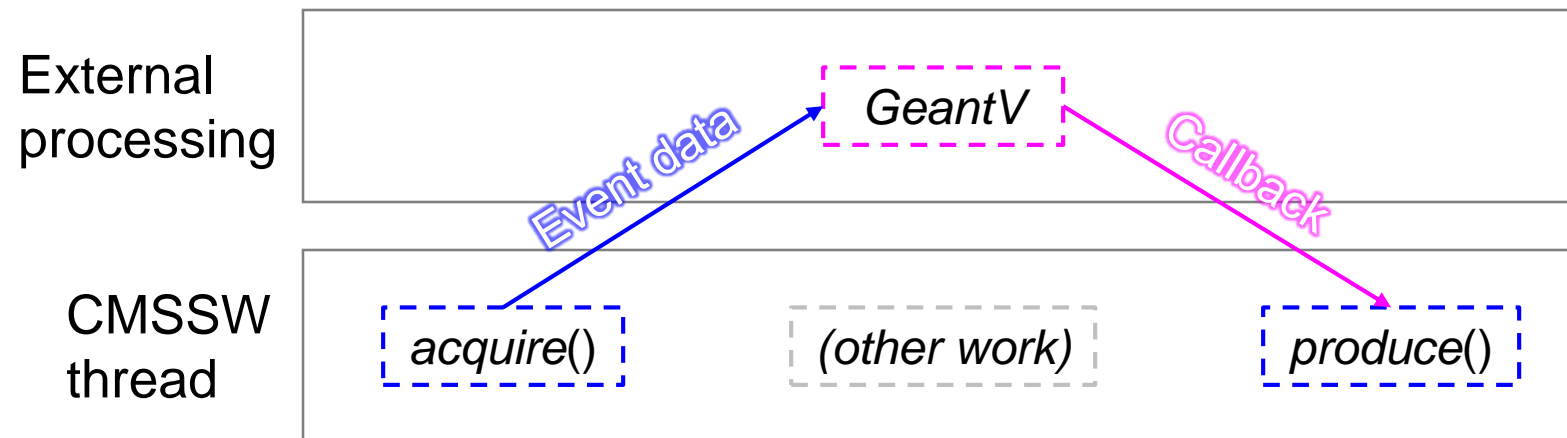
Geant4 vs. GeantV Scoring

- Sensitive detectors (SD) and scoring trickiest to adapt
  - Necessary to test "full chain" (simulation → digitization → reconstruction)
 - Significantly more complicated than Geant4 MT
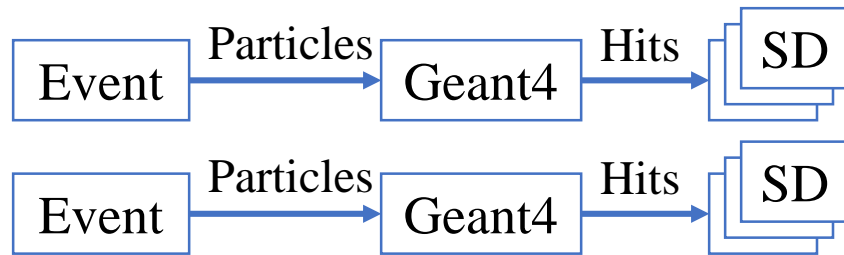
Geant4 shares memory, but each event processed in separate thread

Each event processed in multiple threads, mixed in with other events

- Duplicate SD objects per event per thread, then aggregate
  → 4 streams, 4 threads = 16 SD objects
  - GeantV TaskData supports this approach
- Use template wrappers to unify interfaces and operations
  - Ensure exact same SD code used for Geant4 & GeantV
  - Minimize overhead (no branching or virtual table)

GeantV Data Aggregation
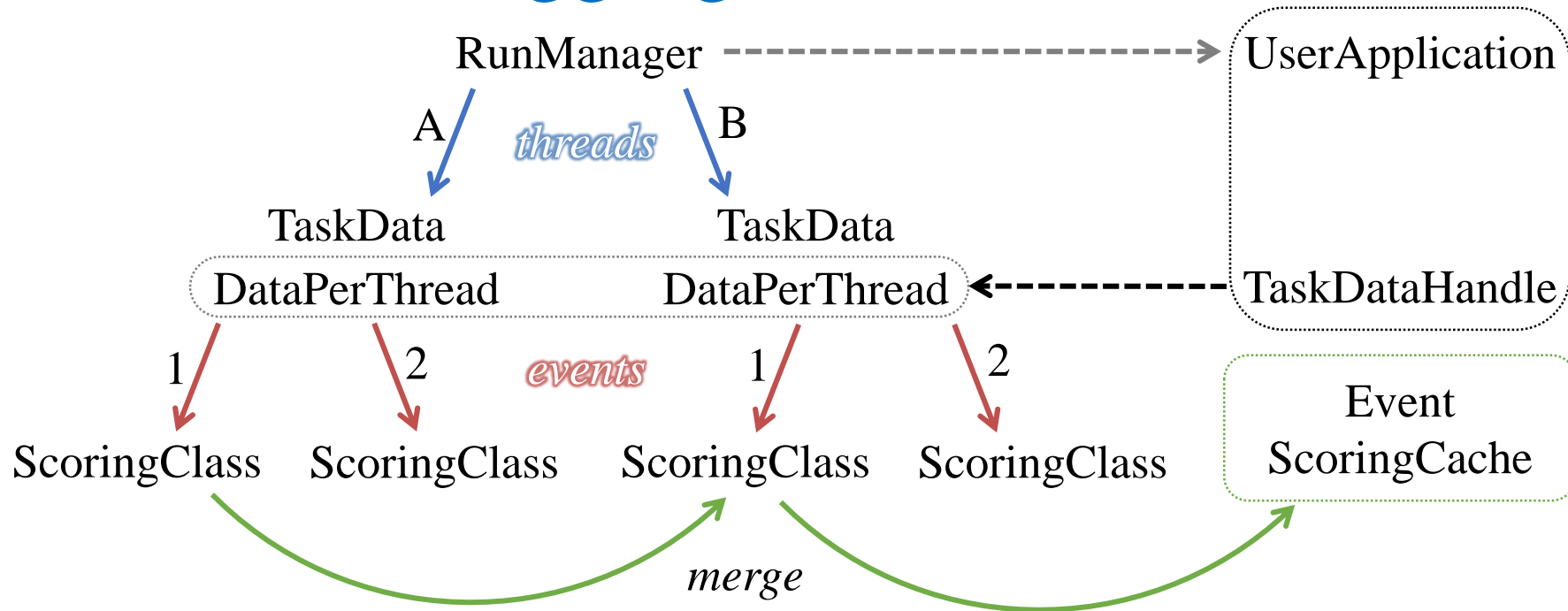
- Each ScoringClass object has instance of CaloSteppingAction
  - Some additional memory overhead from duplicated class members
    - Attempt to minimize this by storing volume maps in magic static struct
- Merged ScoringClass output copied to cache attached to Event object
  - GeantV may consider event finished before CMSSW has written output
    → copy to cache, then immediately clear ScoringClass objects
    (avoid possible race conditions)

Physics Validation

- Settings:
  - Geant4 10.4p2 w/ VecGeom v0.5 (scalar)
  - GeantV pre-beta-7 w/ VecGeom v1.1
  - All CMS-specific G4 optimizations disabled
  - Same production cuts (default 1mm)
  - Single thread (reproducible pRNG sequences)
  - Generate 1000 events w/ single electron, E = 100 GeV, $\eta = 1.0$, $\varphi = 1.1$
- Tests: (same generated events used for G4 and GV)
1. No field (B = 0)
2. Constant field (B = 3.8 T)

(more in backup)

1. Energy Deposits for 100 GeV e- (B=0)

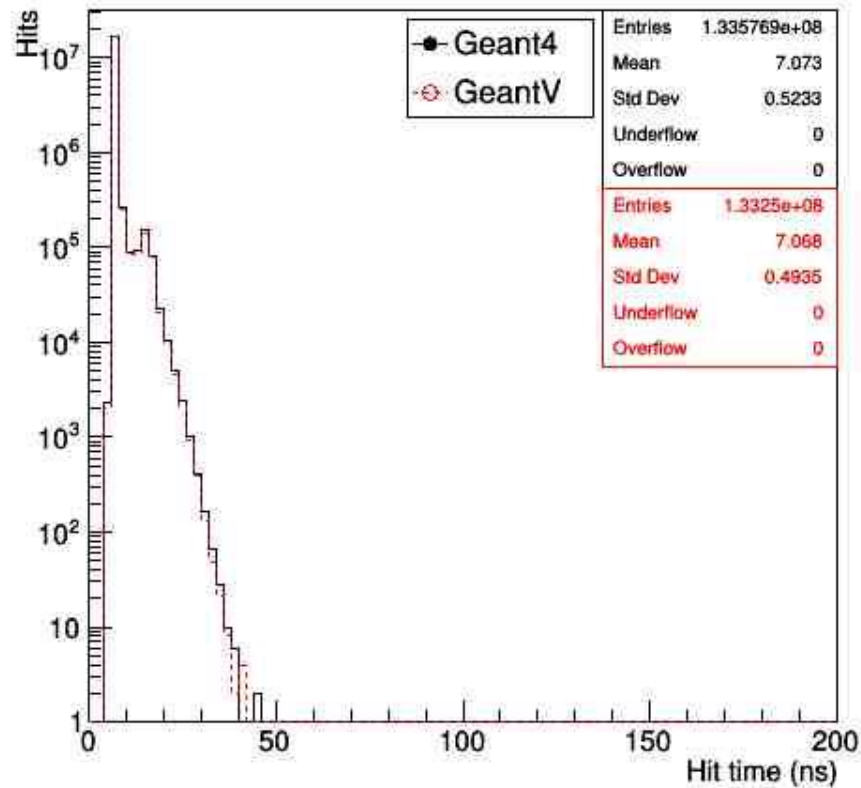

100 GeV Electron B=0 EB (Geant4 vs GeantV)
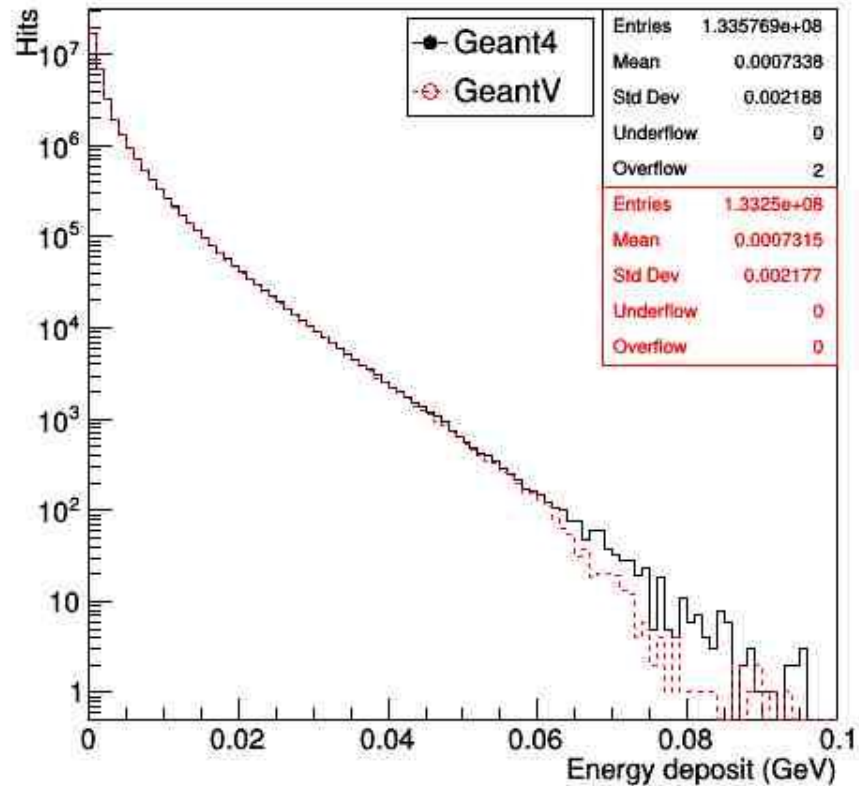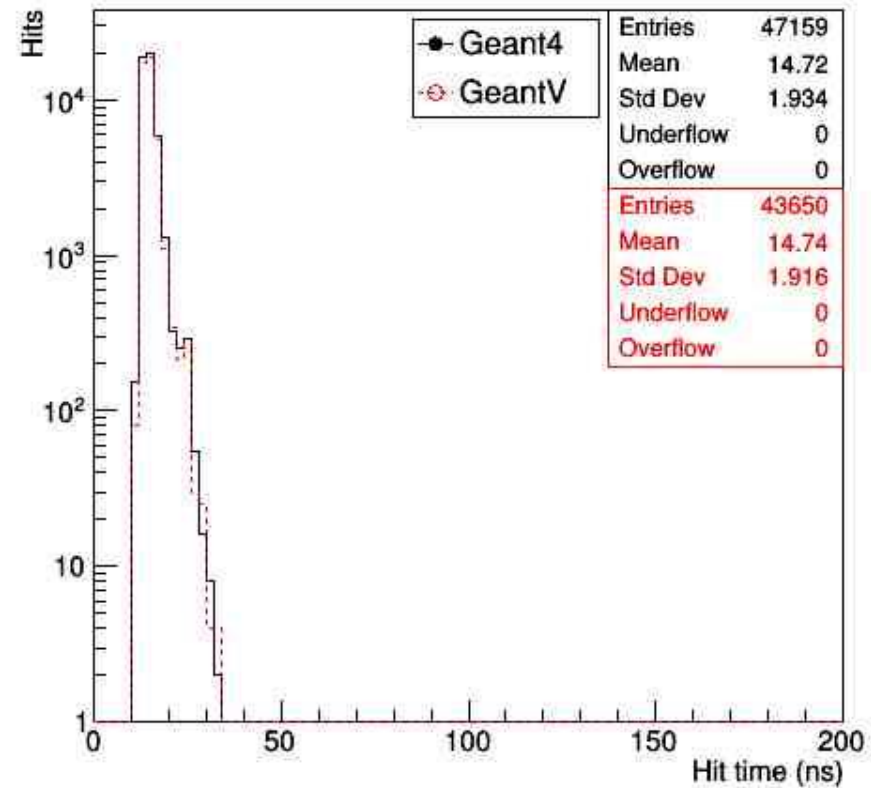
100 GeV Electron B=0 EE (Geant4 vs GeantV)
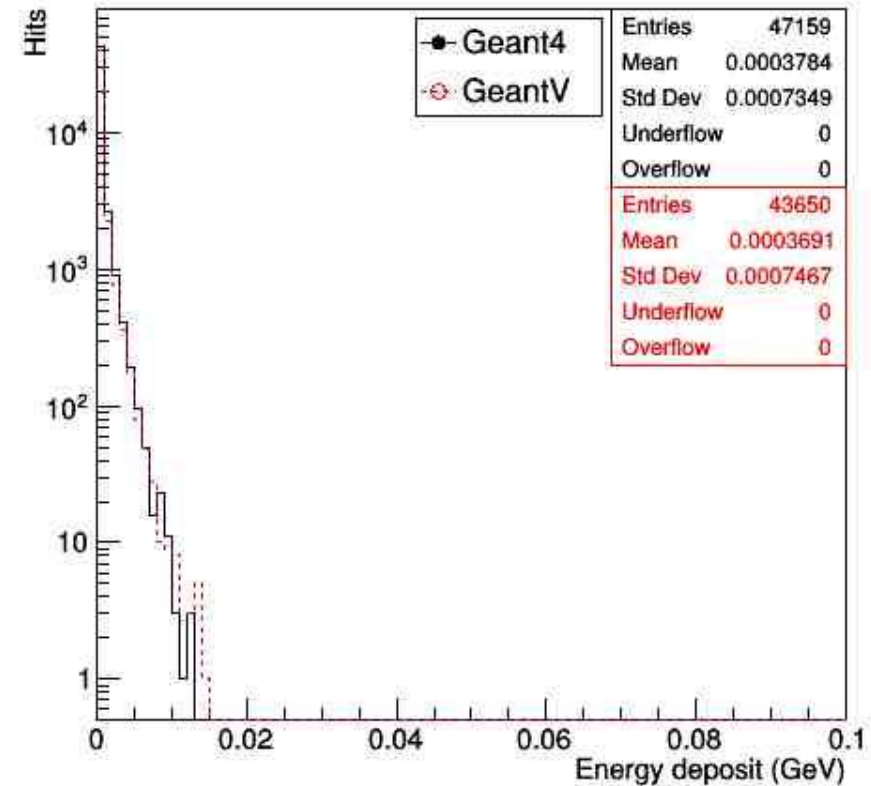
- The number of entries differs by 0.3% (7.4%) in EB (EE)
- The means differ by 0.2% for EB and 2.5% for EE

1. Hit Time for 100 GeV e- (B=0)

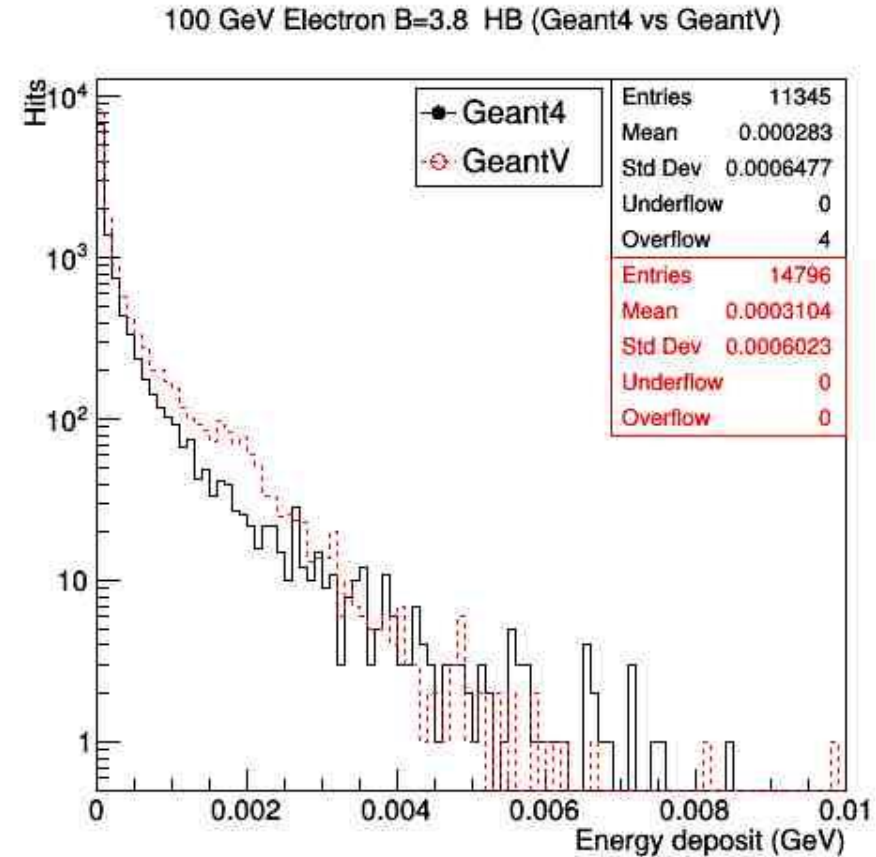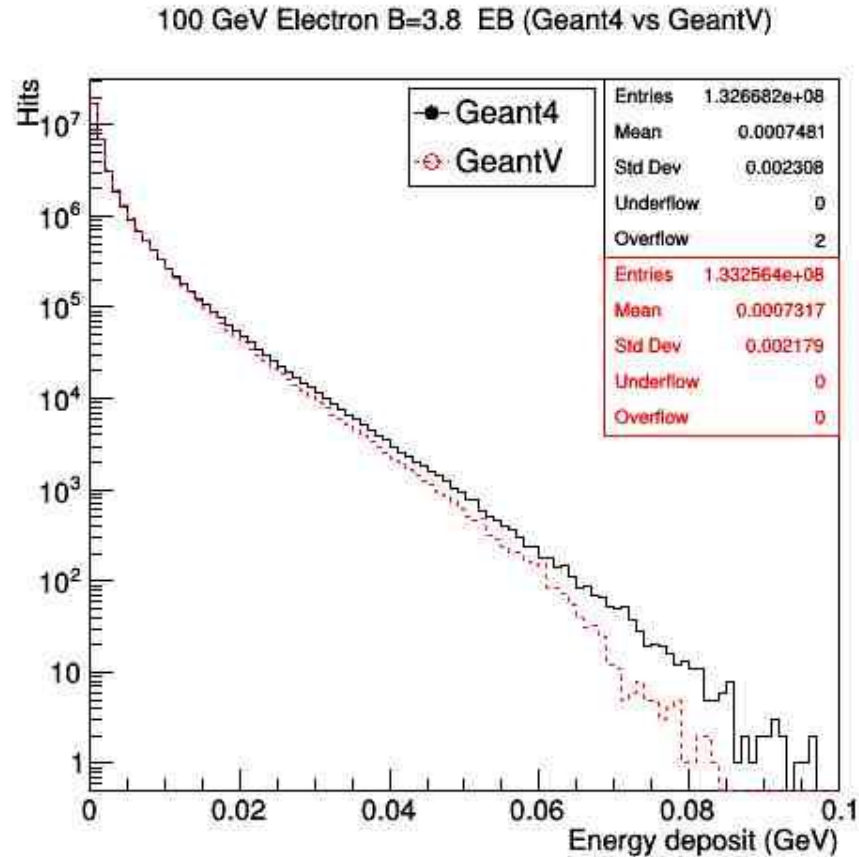100 GeV Electron B=0 EB (Geant4 vs GeantV)

100 GeV Electron B=0 EE (Geant4 vs GeantV)

- Means differ by 0.07% for EB and 0.13% for EE
- GeantV and Geant4 applications provide roughly the same distributions
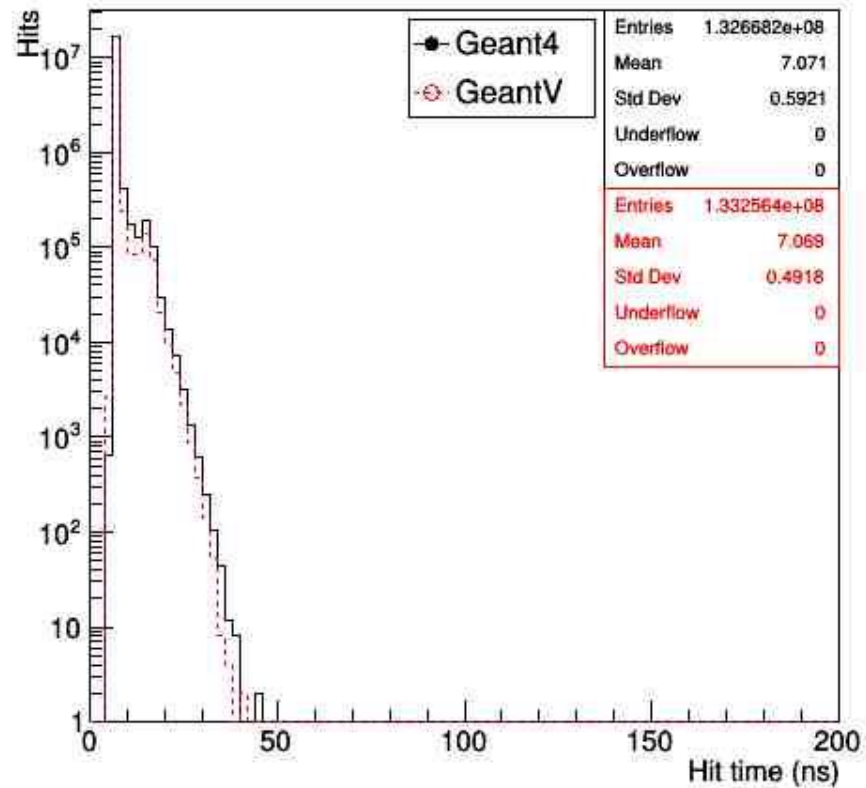
2. Energy Deposits for 100 GeV e- (B=3.8)

100 GeV Electron B=3.8 EB (Geant4 vs GeantV)

100 GeV Electron B=3.8 HB (Geant4 vs GeantV)

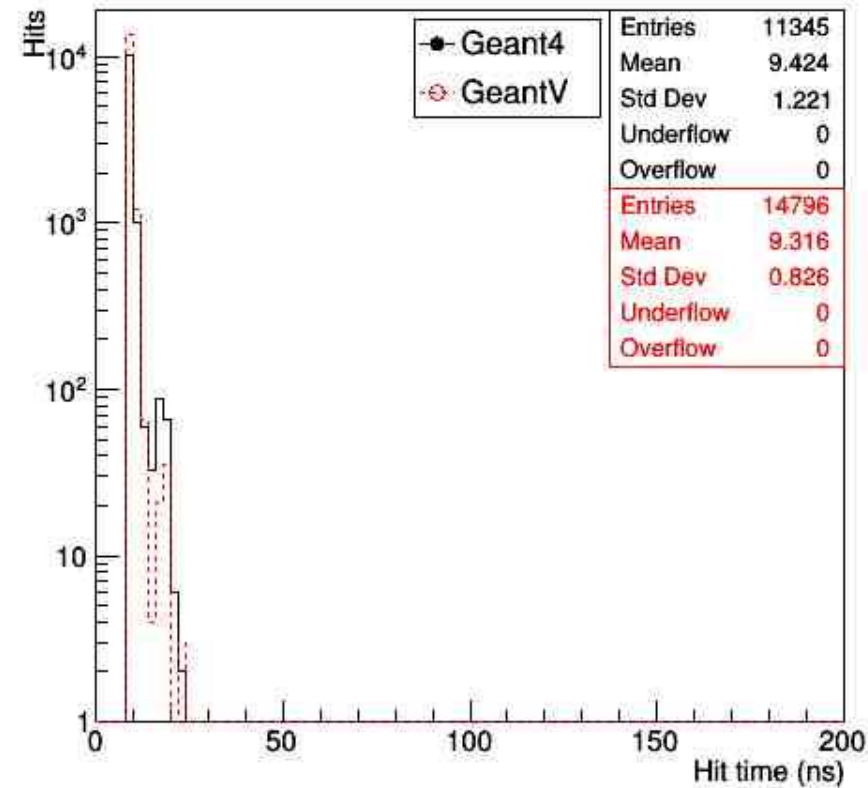- The number of entries differ by 0.4% (23.3%) in EB (HB)
- The means differ by 2.2% for EB and 8.8% for HB

2. Hit Time for 100 GeV e- (B=3.8)

100 GeV Electron B=3.8 EB (Geant4 vs GeantV)

100 GeV Electron B=3.8 HB (Geant4 vs GeantV)

- The means differ by 0.03% for EB and 1.15% for HB
- There is a small difference in the physics results of GeantV and Geant4 applications in the presence of B-field

Performance Tests

- Settings:
  - GeantV pre-beta-7+ (63468c9b)
    - Enabled: vectorized multiple scattering, field (not physics)
  - Generate 500 events, 2 electrons w/ E = 50 GeV, random directions
  - Keep # events / thread constant (copy & concat 500 generated events)
  - Use same generated events in G4 and GV
  - Keep unused threads busy
  - Disable output
- Machine: FermiCloud VM w/
  - Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz, 4096 KB cache
  - Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz, 35840 KB cache, 28 cores
  - sse4.2 instructions
- Track wall clock time & memory with CMSSW TimeMemoryInfo tool
  - Measures VSIZE, RSS per event
  - Calculate speedup from wall time
   (divided by # threads used, since # events / thread is constant)

Characterization

- VM CPU has relatively small cache

  - Known that major component of GeantV speedup arises from smaller library → fewer cache misses

- To characterize CMSSW performance results, first run built-in GeantV FullCMS standalone test

  - Single thread, settings as close to previous slide as possible (see test script: testStandalone.sh)

  - NB: different physics list used in standalone vs. CMSSW

- Results E5-2660 v2 @ 2.20GHz, 4096 KB cache:

  - GeantV: RealTime=756.002s CpuTime=753.09s

  - Geant4: User=1617.36s Real=1618.52s

→**2.14× speedup (standalone**)

- Results E5-2683 v3 @ 2.00GHz, 35840 KB cache:

  - GeantV: RealTime=526.796s CpuTime=526.76s

  - Geant4: User=842.67s Real=842.76s

→**1.6× speedup (standalone)**

Time Performance E5-2660 v2 @ 2.20GHz

speedup = time(threads=1)/time(threads=N)
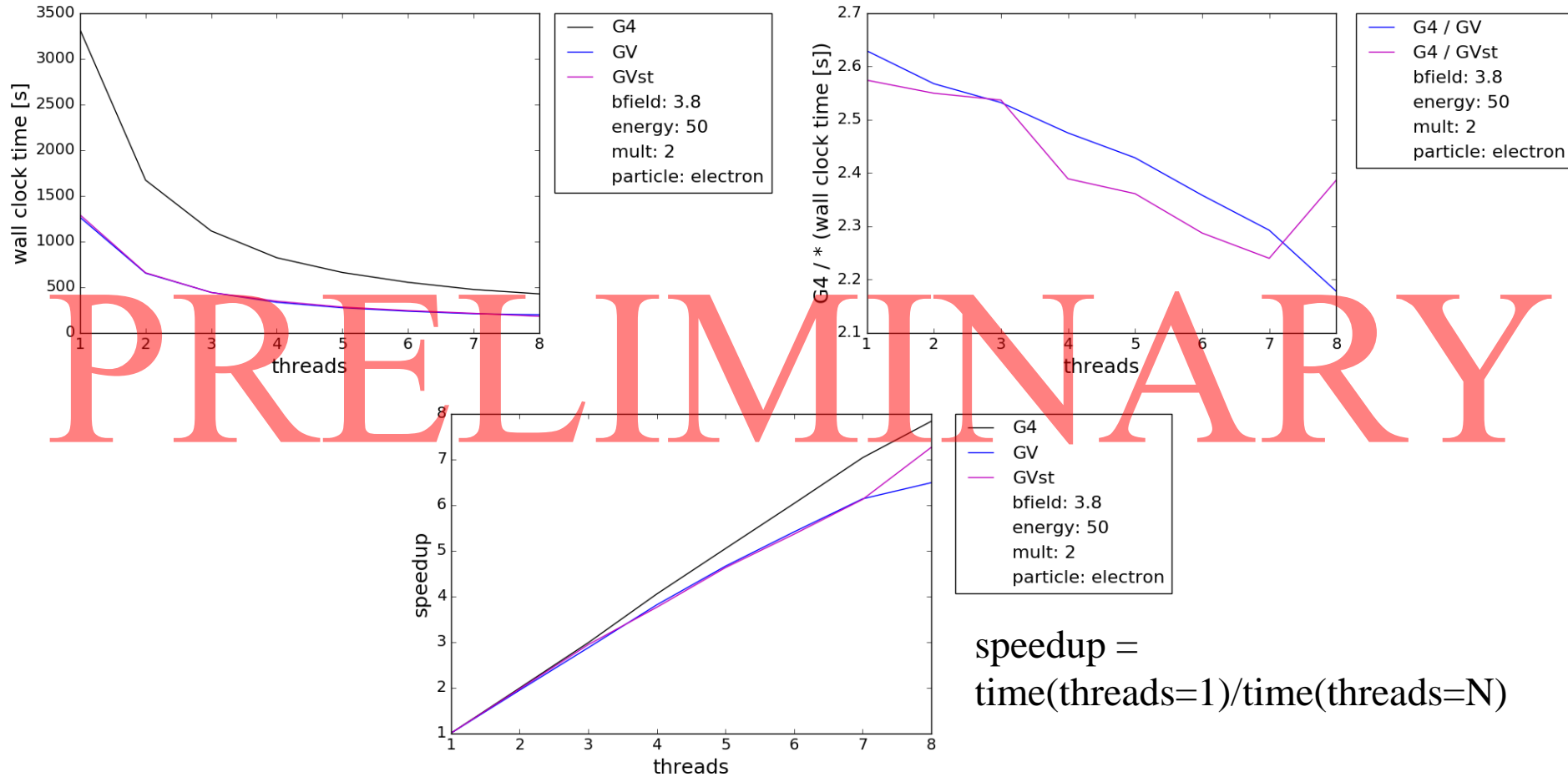
- GV 2.6× faster than G4 single thread, still ~2.2× faster in MT
- GV single track mode similar to basketized
- G4 has better scaling w/ # threads than GV

Memory Performance E5-2660 v2 @ 2.20GHz
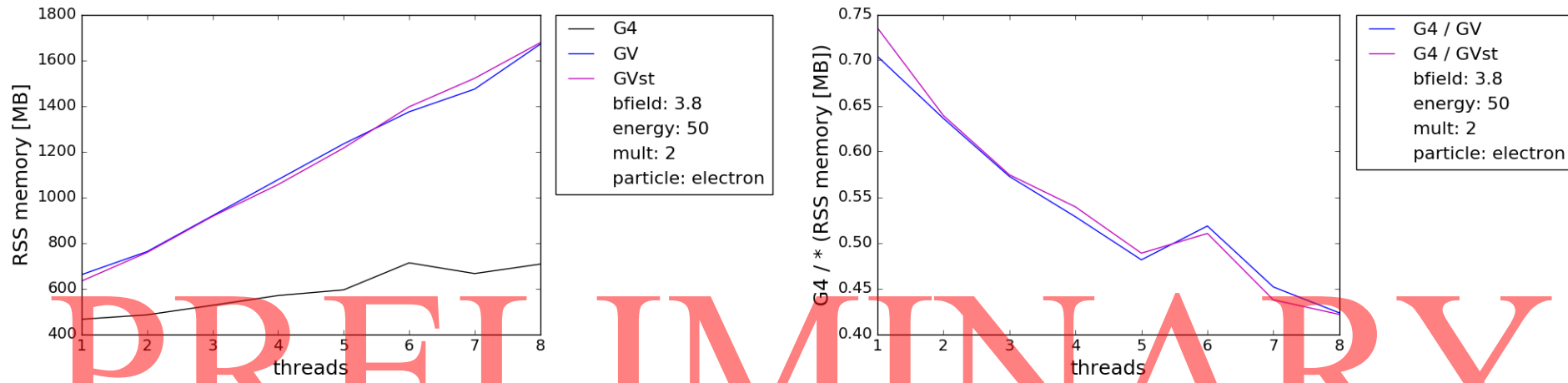
- Memory grows ~linearly w/ # threads (expected)

- GV uses more memory than G4 (expected)

- Single track mode uses similar memory to basketized

Time Performance E5-2683 v3 @ 2.00GHz
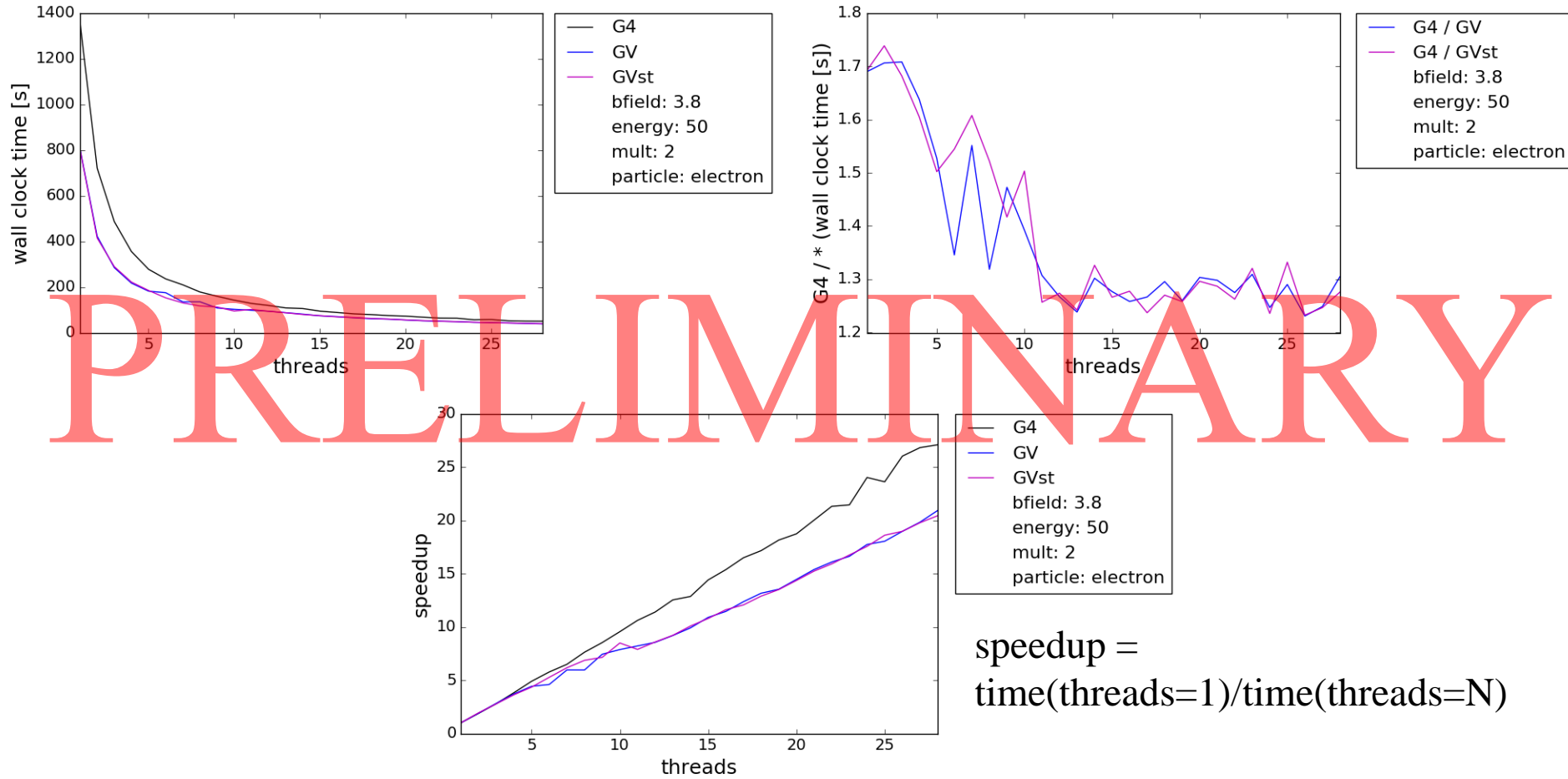

$$speedup = time(threads=1)/time(threads=N)$$

- GV 1.7× faster than G4 single thread, still ~1.3× faster in MT
- GV single track mode similar to basketized
- G4 has better scaling w/ # threads than GV
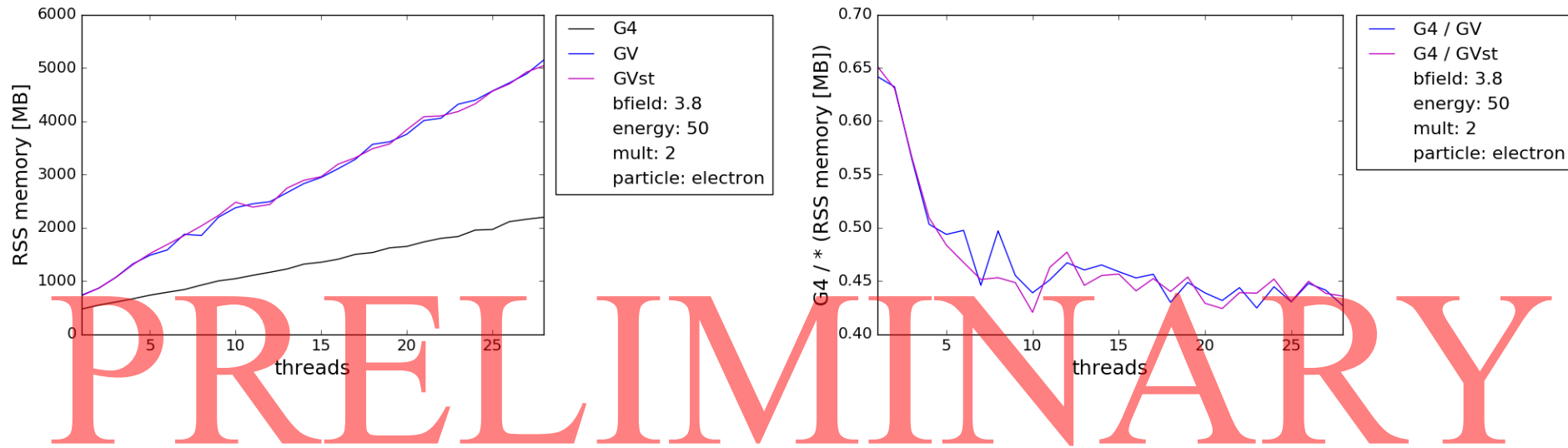
Memory Performance E5-2683 v3 @ 2.00GHz

- Memory grows ~linearly w/ # threads (expected)

- GV uses more memory than G4 (expected)

- Single track mode uses similar memory to basketized

What Would Be Next?

- To complete the goals of CMS R&D studies for the paper:
  - Full magnetic field map
  - Test on machines w/ different cache sizes
- Stretch goals/notes for future similar projects:
  - Random number generator
  - Adapt scoring classes for other detectors (beyond calorimeters)
  - Combine w/ other simulation improvements
    - Notably Russian Roulette & HF shower library, which give largest gains
- If GeantV project were to continue:
  - Better solution for geometry conversion than TGeo
  - Sensitive volume/detector functionality
  - Vectorized hadronic physics
  - Improve threading, memory management, and ownership models
  - Decouple event loading & task launching in ExternalLoop mode
  - Event-wise scoring rather than current thread-wise scoring w/ TaskData

Conclusions

➤ CMS studies met ~all goals laid out

   o Co-development led to improvements and bug fixes in GeantV to facilitate experiments' use

   o One of the first projects to exercise CMSSW ExternalWork feature

   o Physics validation & CPU measurements show very positive results

   o Path to adapt interfaces efficiently is laid out:
     "Rosetta stone" mostly contained in StepWrapper and VolumeWrapper

Geant4	GeantV
StepWrapper	StepWrapper
VolumeWrapper	VolumeWrapper

➤ Demonstrator to test major elements of GeantV-CMSSW integration is ready

   o Speedup of 1.7x - 2.6× in CMSSW application

   o Will finalize results for paper

   o The CMS simulation group thanks the GeantV R&D team for providing support to this integration exercise and making it a successful co-development endeavor

Extras from CMSSW integration exercise

Template Wrappers

➢ **Goal**: use *exact same* SD code for Geant4 and GeantV
- Problem: totally incompatible APIs
  - Example: `G4Step::GetTotalEnergyDeposit()` vs. `geant::Track::Edep()`
- Solution: template wrapper with unified interface
  e.g. `StepWrapper<T>::getEnergyDeposit()`
  - SD code *only calls* the wrapper
  - Wrapper stores pointer to T (minimize overhead)
- Current wrappers:
  - BeginRun
  - BeginEvent
  - Step
  - Volume
  - EndEvent
  - EndRun

Traits

- Collect Geant4/GeantV-specific types and wrappers into unified **Traits** class:

```
struct G4Traits {
  typedef G4Step Step;
  typedef sim::StepWrapper<Step> StepWrapper;
};
struct GVTraits {
  typedef geant::Track Step;
  typedef sim::StepWrapper<Step> StepWrapper;
};
```

- Provides standardized typenames to be used by SD class:

```
template <class Traits> class CaloSteppingActionT : …,
  public Observer<const typename Traits::Step *>
{
  public:
    void update(const Step * step) override {
update(StepWrapper(step)); }
  private:
    // subordinate functions with unified interfaces
    void update(const StepWrapper& step);
};
```

Organization

Old

CaloG4
CaloSteppingAction (.h, .cc)

New

Calo
CaloSteppingActionT (.h, .icc)
Wrappers (.h)

CaloG4
CaloSteppingAction (.h, .cc)
G4 Wrappers (.h), Traits (.h)

CaloGV
CaloSteppingAction (.h, .icc)
GV Wrappers (.h), Traits (.h)
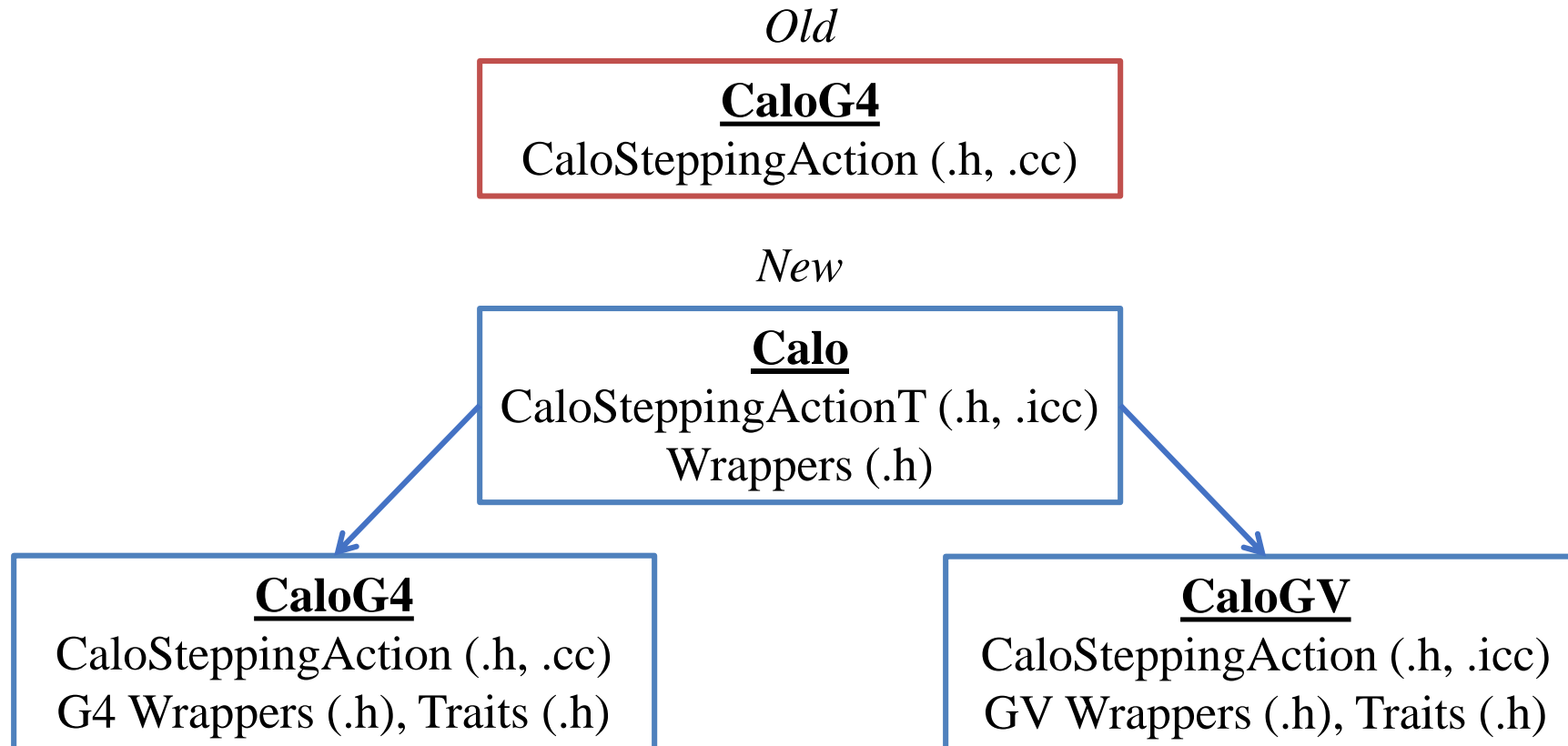
- SD interface & implementation in **Calo** (.icc file), w/ unimplemented wrapper interfaces
- G4/GV wrapper specializations in **CaloG4/GV**, w/ specific instances of templated SD class → isolate dependencies

Scoring Approaches

- Two approaches to scoring in CMSSW:

1. Inherit from **G4VSensitiveDetector** (Geant4 class)
   → automatically initialized for geometry volumes marked as sensitive

2. Inherit from **SimWatcher** (CMSSW standalone class)
   → need to specify names of watched geometry volumes

- CaloSteppingAction is a demonstrator class w/ approach 2

  o Simplified version of ECAL and HCAL scoring

  o Less dependent on Geant4 interfaces

- "Real" SD code uses approach 1

➢ More work to extract Geant4 dependencies will be necessary

  o Some SD class methods directly from Geant4 (via inheritance)

  o Need to mock up Geant4-esque interfaces w/ dummy classes for GeantV

More Physics Validation

3. Generate 1000 events of single electrons at 2, 10 and 50 GeV at a fixed direction and compare GeantV against Geant4 with magnetic field off and on at 3.8 Tesla
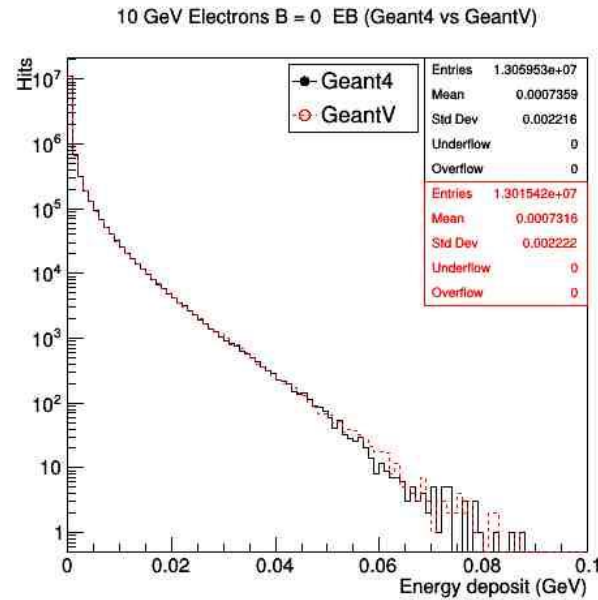
4. Generate 100 events of 50 GeV double electrons at 50 GeV with $-3 < \eta < 3$ and $0 < \varphi < 2\pi$, run in multi-threaded mode (4 threads), B = 0 Tesla

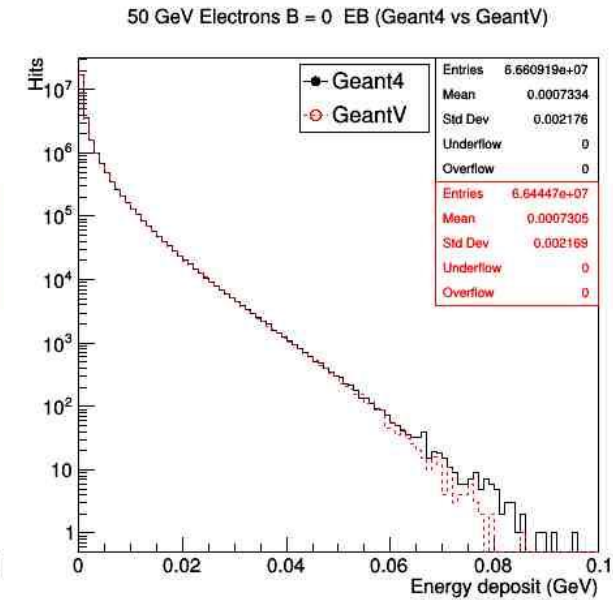5. Repeat multi-threaded test with B = 3.8 Tesla

3. Energy Deposit with B = 0

2 GeV Electrons

10 GeV Electrons

50 GeV Electrons

2 GeV Electrons B = 0 EB (Geant4 vs GeantV)

	Entries	2443613
Geant4	Mean	0.0007388
GeantV	Std Dev	0.002317
	Underflow	0
	Overflow	0
	Entries	2441583
	Mean	0.0007327
	Std Dev	0.002327
	Underflow	0
	Overflow	0

10 GeV Electrons B = 0 EB (Geant4 vs GeantV)

	Entries	1.305953e+07
Geant4	Mean	0.0007359
GeantV	Std Dev	0.002216
	Underflow	0
	Overflow	0
	Entries	1.301542e+07
	Mean	0.0007316
	Std Dev	0.002222
	Underflow	0
	Overflow	0

50 GeV Electrons B = 0 EB (Geant4 vs GeantV)

	Entries	6.660919e+07
Geant4	Mean	0.0007334
GeantV	Std Dev	0.002176
	Underflow	0
	Overflow	0
	Entries	6.64447e+07
	Mean	0.0007305
	Std Dev	0.002169
	Underflow	0
	Overflow	0

- Number of hits is the same for all 3 energies. The differences are at the level of 0.1/0.3/0.2% for 2, 10 and 50 GeV
- The means differ by 0.8/0.6/0.4% at the three energies

3. Energy Deposit with B = 3.8

2 GeV Electrons

2 GeV Electrons B = 3.8 Tesla EB (Geant4 vs GeantV)

10 GeV Electrons

10 GeV Electrons B = 3.8 Tesla EB (Geant4 vs GeantV)

50 GeV Electrons

50 GeV Electrons B = 3.8 Tesla EB (Geant4 vs GeantV)
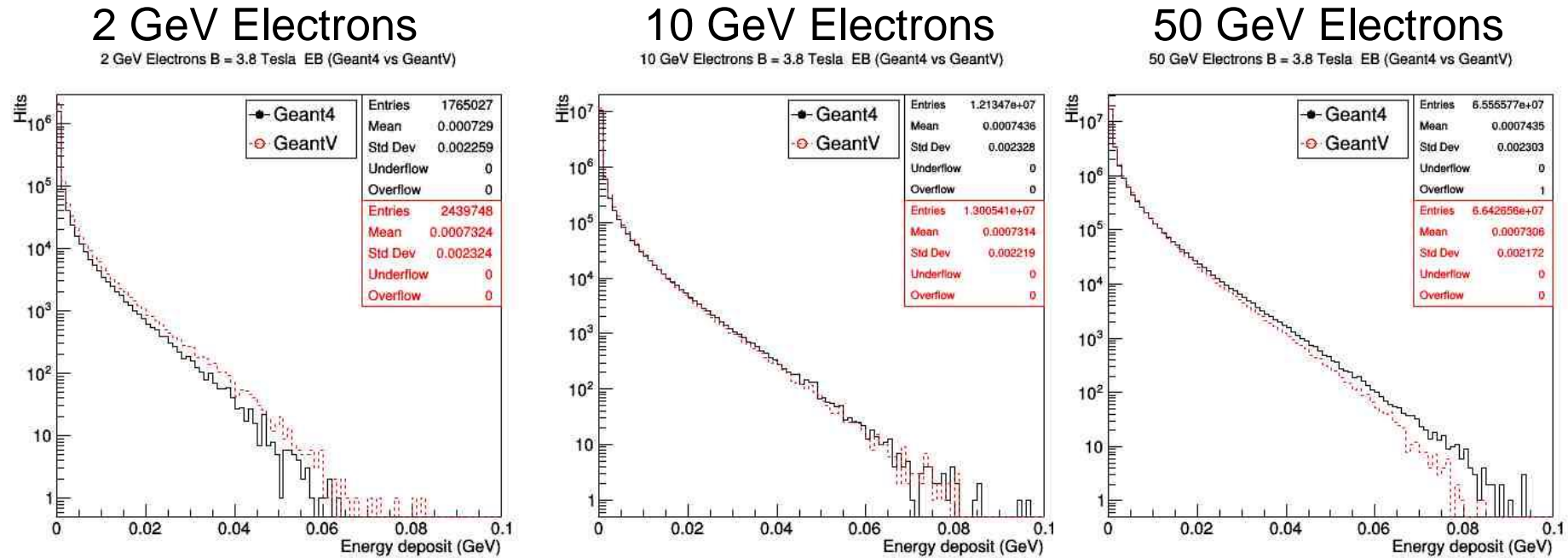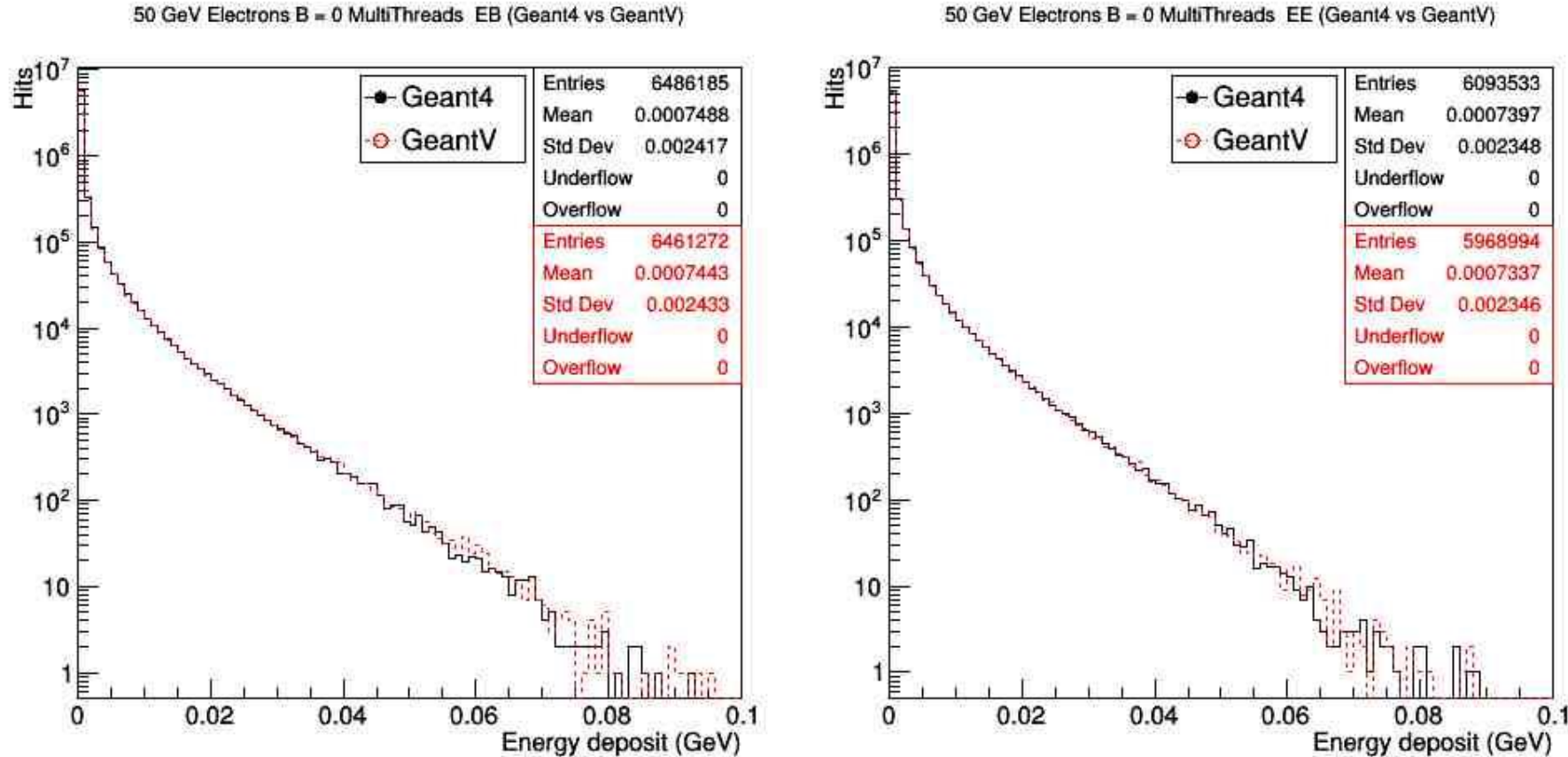

- Number of hits is the same for all 3 energies. The differences are at the level of 27.7/6.7/1.3% for 2, 10 and 50 GeV
- The means differ by 0.5/1.6/1.7% at the three energies
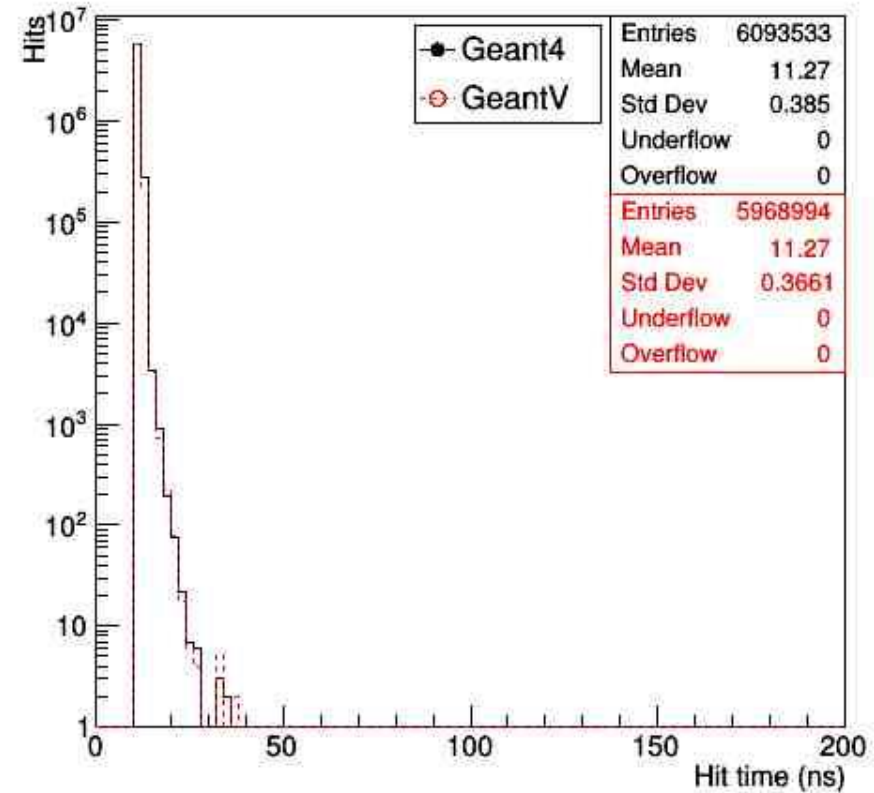
4. Energy Deposit with B = 0, MT

50 GeV Electrons B = 0 MultiThreads EB (Geant4 vs GeantV)

Geant4		
Entries	6486185
Mean	0.0007488
Std Dev	0.002417
Underflow	0
Overflow	0

GeantV		
Entries	6461272
Mean	0.0007443
Std Dev	0.002433
Underflow	0
Overflow	0

50 GeV Electrons B = 0 MultiThreads EE (Geant4 vs GeantV)

Geant4		
Entries	6093533
Mean	0.0007397
Std Dev	0.002348
Underflow	0
Overflow	0

GeantV		
Entries	5968994
Mean	0.0007337
Std Dev	0.002346
Underflow	0
Overflow	0

- Events are generated with 50 GeV electrons having random direction within a limited range of η and φ
- The agreement is pretty good in the B=0 option for both # of hits as well as in the shape of the distributions for EB and EE

4. Hit Times with B = 0, MT

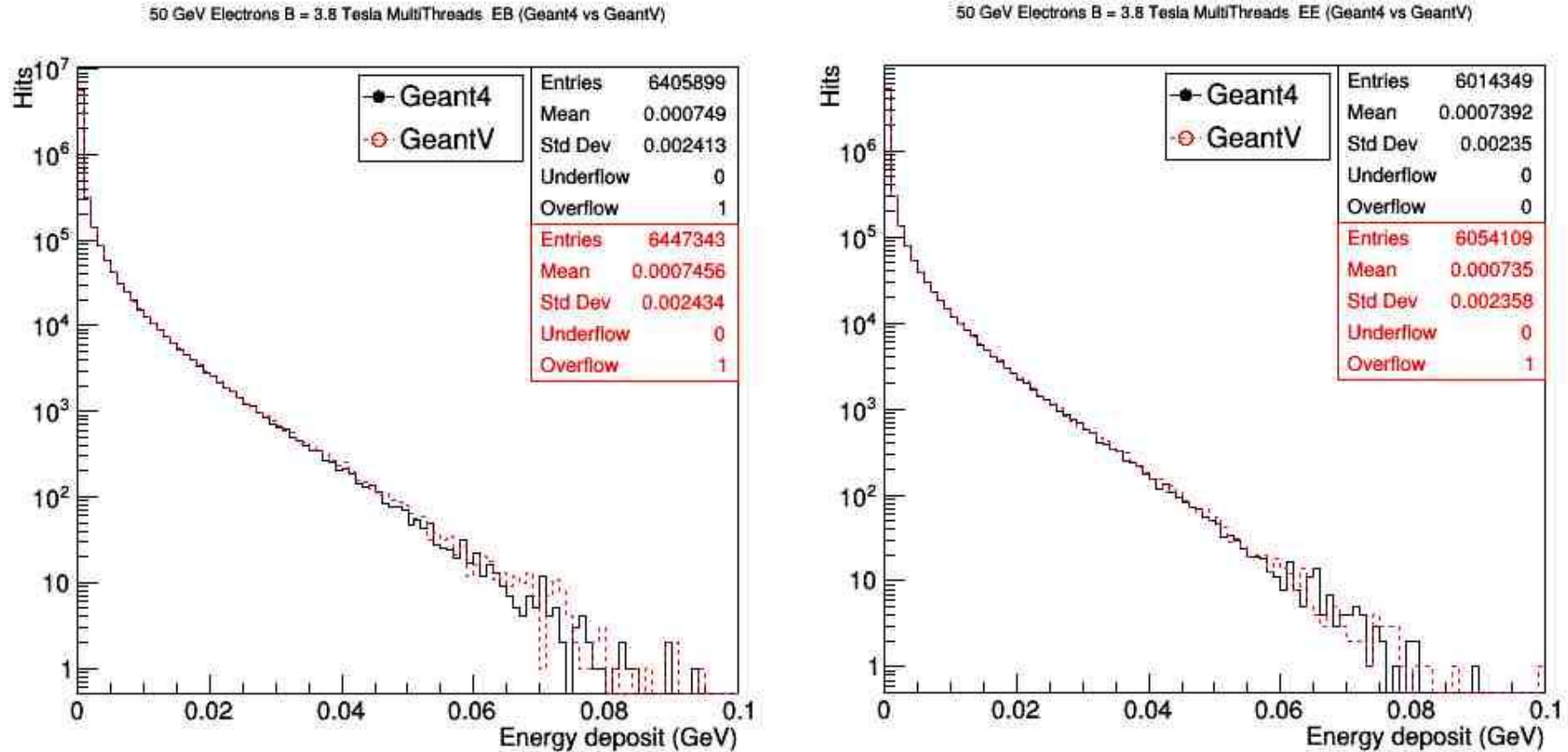

50 GeV Electrons B = 0 MultiThreads EB (Geant4 vs GeantV)
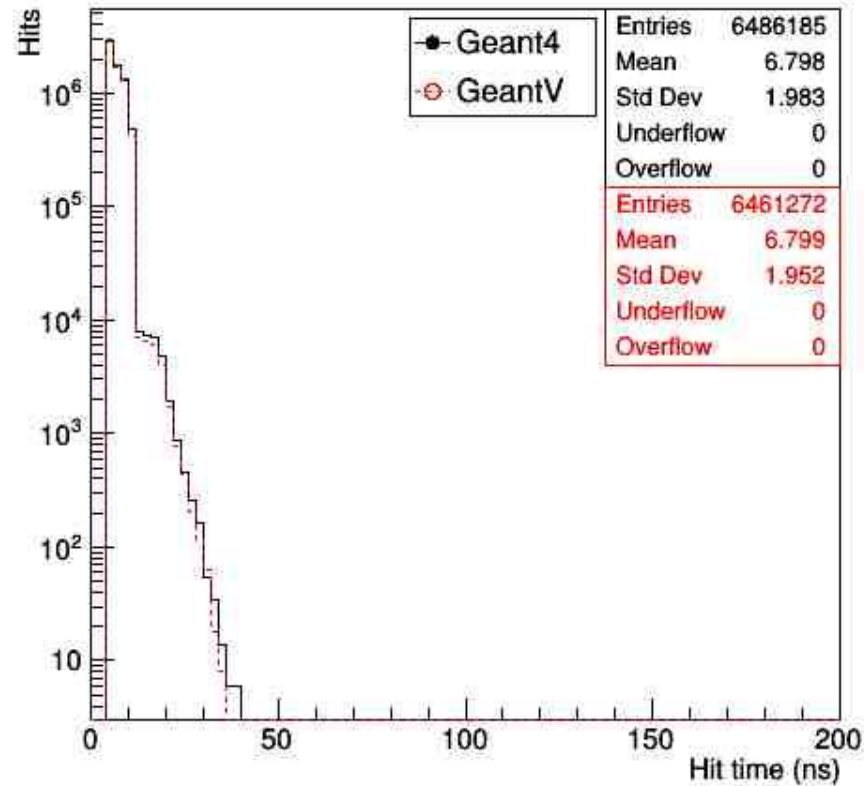
50 GeV Electrons B = 0 MultiThreads EE (Geant4 vs GeantV)
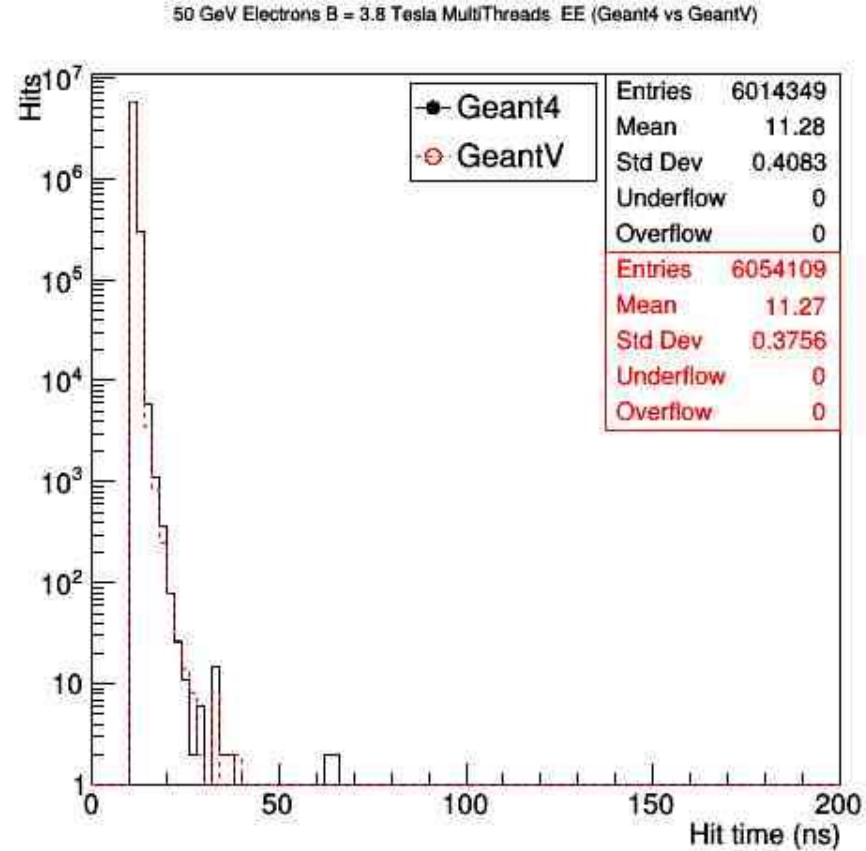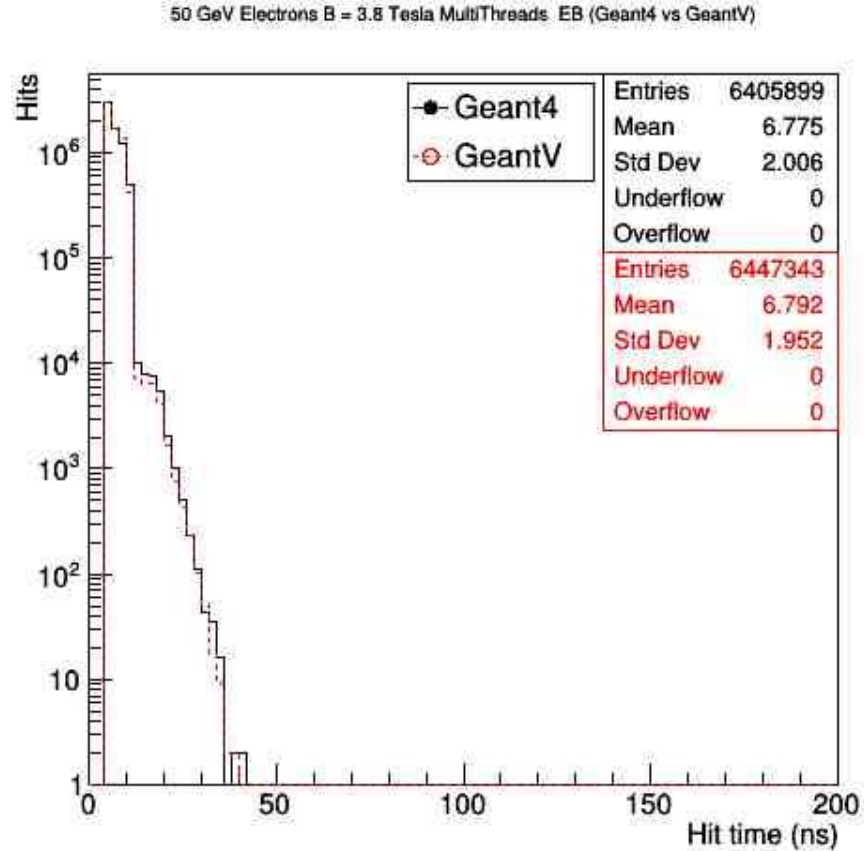
- Hit time distributions are also in good agreement for the B=0 option in EB as well as in EE

5. Energy Deposit with B = 3.8, MT

50 GeV Electrons B = 3.8 Tesla MultiThreads EB (Geant4 vs GeantV)

50 GeV Electrons B = 3.8 Tesla MultiThreads EE (Geant4 vs GeantV)

- Same events (50 GeV electrons, random direction within a limited range of η and φ) are simulated in a uniform B-field option of 3.8 Tesla
- The agreement is still good for both # of hits as well as in the shape of the distributions for EB and EE

5. Hit Times with B = 3.8, MT

50 GeV Electrons B = 3.8 Tesla MultiThreads EB (Geant4 vs GeantV)

50 GeV Electrons B = 3.8 Tesla MultiThreads EE (Geant4 vs GeantV)

- Hit time distributions are also in reasonable agreement for the B = 3.8 Tesla option in EB as well as in EE

CMS Simulation Optimizations

Configuration	Relative CPU usage	
	MinBias	ttbar
No optimizations	1.00	1.00
Static library	0.95	0.93
Production cuts	0.93	0.97
Tracking cut	0.69	0.88
Time cut	0.95	0.97
Shower library	0.60	0.74
Russian roulette	0.75	0.71
FTFP_BERT_EMM	0.87	0.83
VecGeom (scalar)	0.87	0.93
Mag. field step,track	0.92	0.90
All optimizations	0.16	0.24

- HF shower library, Russian Roulette have largest impacts

- VecGeom, mag. field improvements entered production in past ~year

  o Enabled by validating and using latest Geant4 versions

- Cumulative effects: overall, simulation is **6.2×** (**4.1×**) faster for MinBias (ttbar) vs. default Geant4 settings

➢ CMS full simulation is at least 8× faster than ATLAS

Follow-up R&D directions

Backup slides

Some follow-up directions

- Reusable components and ideas for improving existing Geant4
  - Extending VecCore, VecMath, VecGeom
  - Investigate basketization of few performance-critical components in Geant4
    - Started already

- Follow-up plans
  - Compact specialized libraries as alternative to general stepping approach
  - Extraction of basketization generic library, possible basketization of FP-intensive components
  - Disentangling state from managers and move towards more functional programing style
    - Increase flexibility of functional-based regrouping, enable parallelism opportunities below event level.
  - Review the data model and flow in Geant4 to pre-empt extra parallelism and acceleration opportunities