Vectorization of simulation code

Andrei Gheata for GeantV R&D team

HOW2019

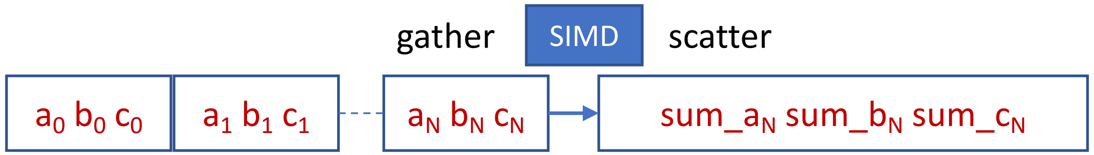
18-22 March 2019, Jefferson Lab

Preamble: “loop was not vectorized: vectorization possible but seems inefficient”

```
#include<iostream>
#define N 100

struct s1 {int a, b, c;};

int main()
{
  s1 arr[N], sum;
  for(int i = 0; i < N; i++) {
    sum.a += arr[i].a;
    sum.b += arr[i].b;
    sum.c += arr[i].c;
  }
  std::cout << sum.a << "t” << sum.b << "t” << sum.c << "n";
  return 0;
}
```



- Many possible reasons
  - Non unit stride access, loop reminders, branching generating inefficient masking, …
  - In general too much memory copy/move
- Compiler heuristics evaluates vectorization cost
  - Sometimes generating the code to check efficiency

Budget: $T_{SIMD} + T_{OVERHEAD} < T_{SCALAR}$?

Vector Simulation R&D

- GeantV: performance study for a vector simulation workflow
  - An attempt to improve computation performance of Geant4
- Steering framework revisited
  - Track-level parallelism, “basket” workflow
  - Improving instruction and data locality, leverage vectorization
  - Adaptability to new hardware and accelerators
- Making simulation components more portable and vector friendly
  - VecGeom: modern geometry modeler handling single/multi particle queries
  - New physics framework, more simple and efficient
  - VecCore, VecMath: new SIMD API, SIMD-aware RNG and math algorithms

Leveraging vectorization

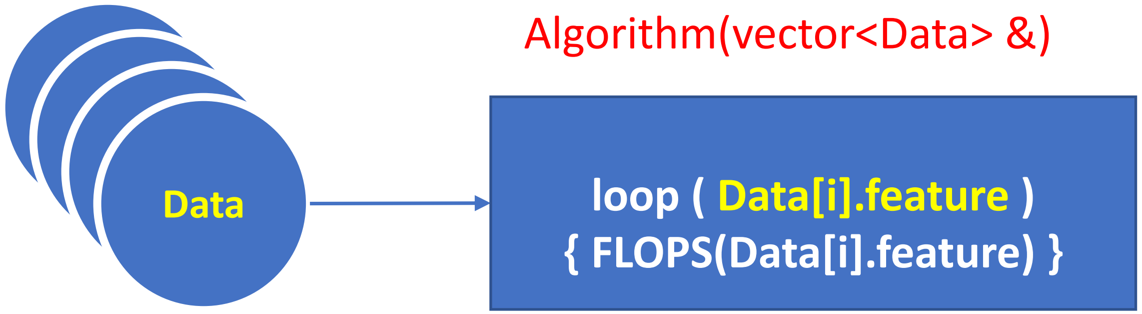
Model feature parallelism
(e.g. surfaces of a polyhedron)

Algorithm(Data &, ModelState& m)

Not so many models with natural inner loops

Data feature parallelism
(e.g. multiple tracks)

Algorithm(vector<Data> &)

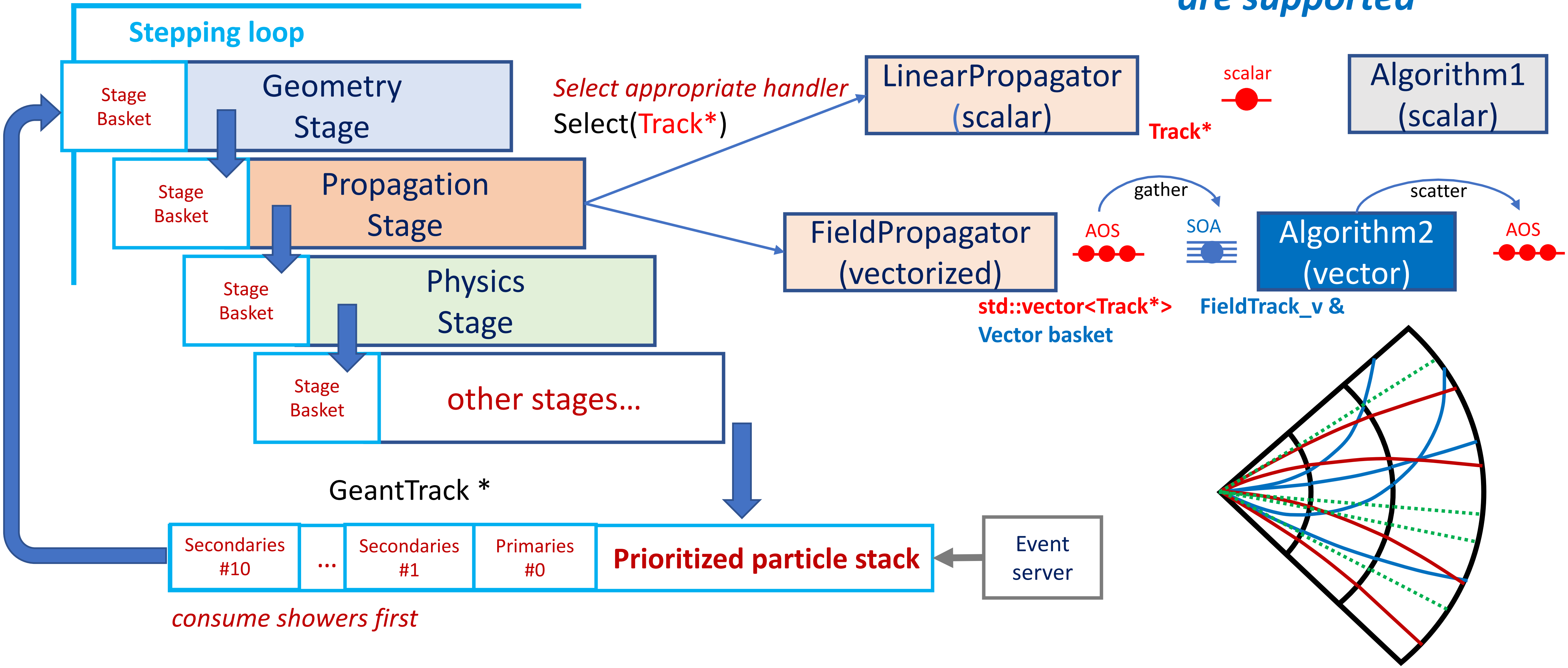


Needs track-parallel environment

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

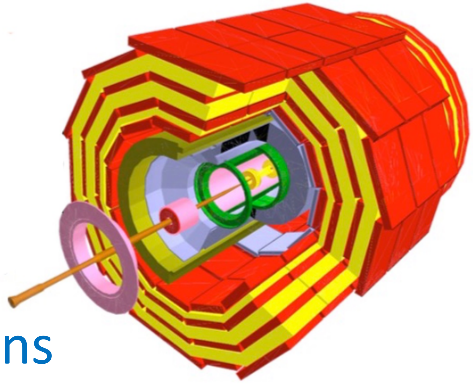
GeantV multi-particle stepping

Both scalar/vector flow are supported



Ongoing CPU performance study

- Examples: simplified sampling calorimeter and a CMS simulation using 2018 geometry and 4T uniform field
  - Complete set of models for e^{+}, e^{-}, γ
  - Geant4 running equivalent physics list, field, geometry setup and cuts
  - Identical physics results, and equivalent #steps, energy deposits, particle yields
- Several configurations testing different performance aspects
  - Field ON/OFF (uniform field, field map version not yet efficient)
  - Single thread / MT performance -> scaling
  - Single track mode (emulating Geant4 tracking) -> locality
  - "Basketization" ON/OFF for different components -> vectorization gains
  - Vector baskets dispatched to scalar code -> overheads

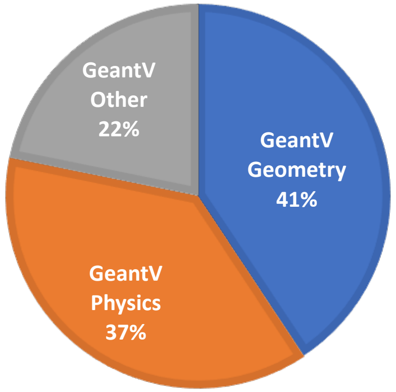


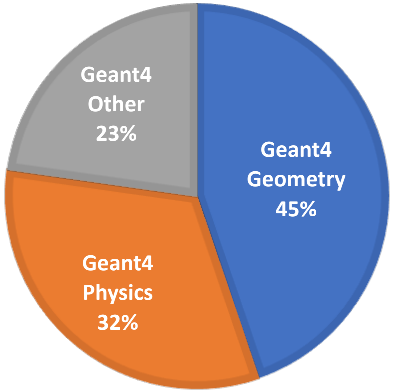
Preliminary performance: CMS example

- GeantV time performance improvement ranges from 1.9 to 2.1 depending on configurations (see latest benchmarks)
- Gains come from every component: geometry, physics, stepping management
  - Hard to disentangle component gains from a “background" of more efficient computation
  - The most efficient CMS GeantV configuration with a uniform field gives a factor of 1.92
  - The CMS experiment is working on realistic tests within the CMS simulation framework (see presentation of Kevin Pedro)
- Global gains from vectorization and workflow can now be evaluated
  - Vectorization benefits: up to 15% total time
  - Basket workflow gains averaging at ~15% total time, with a large variance (0-30%) dependent on CPU architecture
- The rest of performance gain coming mostly from instruction locality
  - Analysis still ongoing, but performance counters showing far fewer instruction cache misses compared to Geant4

Component and global performance figures

- Similar time fractions by category, and very close number of FLOPS (GV/G4)
  - Geometry: important time reduction due to VecGeom navigation
  - Physics: more compact physics code
- Performance indicators better for GeantV
  - Computation intensity, CPU utilization
  - Far fewer instruction cache misses





	GeantV	Geant4	GeantV/ Geant4
FLOPS (DP_OPS)	1.86E12	1.67E12	**1.11**
FLOPS Per Cycle	0.26	0.13	2.00
Instructions Per Cycle	1.06	0.80	1.32
FLOPS per Memory Op	0.56	0.33	1.70
L1 instruction cache misses			1/7.7
L2 instruction cache misses			1/2.2
TLB misses			1/11.2

Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz

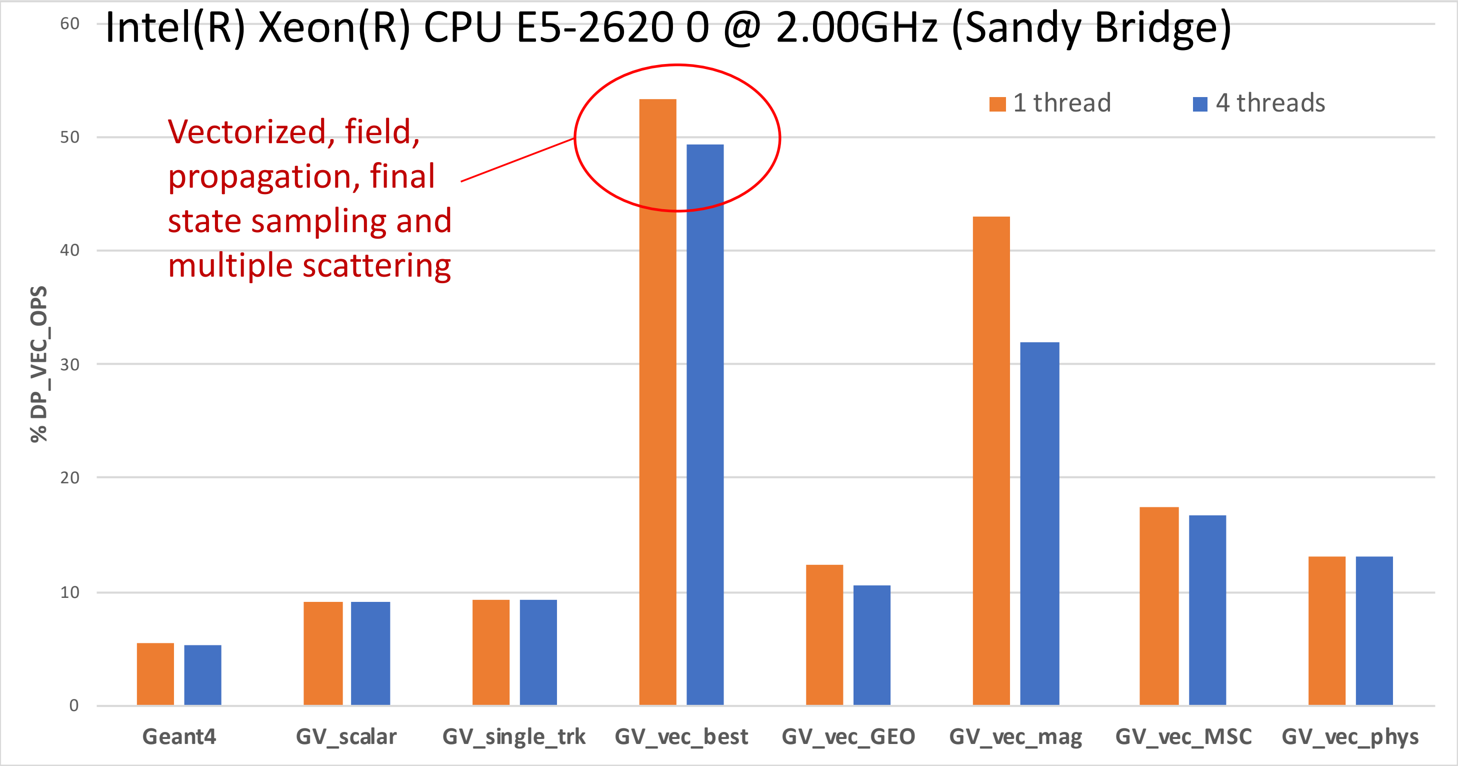
Vectorization performance: CMS example

- Fraction (% total CPU time) of code vectorized so far rather small
  - Physics: 7-11% final state sampling, 6-12% multiple scattering, 15-17% magnetic field propagation
  - Geometry: vectorized code in many branches (~4K volumes in CMS), not yet efficient to basketize
- Important intrinsic vectorization gain factors from unit tests
  - AVX2: Physics models: 1.3-2.5, geometry: 1.5-3.5, field propagation: ~2
- Visible vectorization gains in the total CPU time
  - Physics models: no gain (but MSC: 2-5%), geometry: performance loss, field propagation: 5-9%
  - Performance loss in case of "small" hotspots (e.g. geometry volumes)
- Basketizing is efficient only when applied to "dense FLOPS" algorithms
  - Best basketized configuration in most recent tests brings ~10% (total CPU time) on Haswell AVX2 for vectorized code weighting ~35% (~1.4x visible speedup)

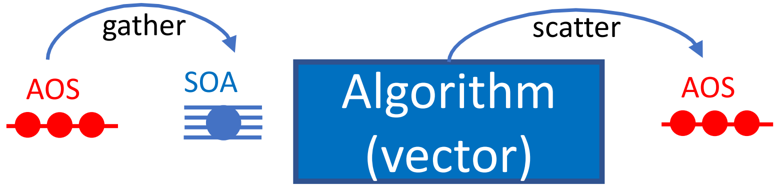
Vectorization potential in simulation

- What % of total DP operation we vectorize?
  - What is the potential to go further?
- Vectorizing more than 50% of the executed DP instructions
  - Coming mostly from field propagation and multiple scattering
  - Remaining potential in geometry/physics
- Vectorization decreases in MT mode
  - Event tail penalties visible

% of AVX vectorized DP instructions



"Basketizing": benefits vs. costs



- Costs (coming from initial scalar approach):
  - Workflow redesign, interface redesign, data structure re-engineering
  - Basketizing overheads: data regrouping, gather/scatter
  - Fine grain parallelism overheads, dealing with tails concurrently
  - Algorithm vectorization effort
- Benefits:
  - Improved instruction locality
    - Single track mode to emulate Geant4-like workflow gives ~15% effect in CPU time (large variance depending on CPU architecture)
  - SIMD instructions: making use of important % of the silicon
  - 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…
  - Working on quantifying this in the current study

Outlook and conclusions

- GeantV prototype demonstrates that vectorizing a large-scale complex HEP application is possible
  - Most of the available DP-ops vectorized, about 50% visible
  - Still some vectorization potential left, more difficult to harvest
- Efficient vectorization is not a piece of cake (for simulation)
  - The limits of the “basket” model now visible, ongoing performance study to outline them
  - Having more computation hotspots would have helped…
- Contributions from basket workflow and vectorization do not explain the full performance gain, the major part (60-70%) is coming from other sources
  - improved instruction cache use, more compact code, less virtual calls, …
  - Currently trying to disentangle these effects
- Finalizing this performance study will outline the directions to go
  - Technical document (facts, numbers and lessons learned) to be prepared
  - What are the directions for adopting some of these benefits in Geant4