Fermilab

Managed by Fermi Research Alliance, LLC for the U.S. Department of Energy Office of Science

Vectorization of Bertini cascade

**J.G.Lima (FNAL), S.Y. Jun (FNAL)
and T. Koi* (SLAC)**

GeantV bi-weekly meeting
August 21, 2018

* T.Koi is not active
in this project anymore

Outline

- Introduction
  - goals and scope
- Progress
  - process flow and vectorization
  - features request
  - preliminary results
- Current status and prospects

SLAC-FNAL pilot project on Geant R&D

Explore new computing avenues for hadronic physics simulation in HEP

Hadronic simulation is an important missing component of the GeantV vector prototype, which explores fine-grain parallelism using a top to bottom vectorization approach for particle transport simulation for next generation detector simulation.

Bertini cascade was chosen for this project, since it is the preferred model for low energy hadron-nucleus interactions and it handles a large number of particle types.

Goals

- Provide standalone, vectorized Bertini algorithms (a specific hadronic cascade model)
- Modularized components (Geant4 and GeantV)
- Modern hardware technologies and parallel architectures

Project scope

- Modularize Geant4 Bertini cascade model and optimization – T.Koi (SLAC)
- SIMD vectorization of some computing-intensive algorithms – G. Lima (FNAL)
- Integration and computing performance evaluation – S.Y. Jun (FNAL)
- Identify requirements for future extension/development

Co-PIs: D. Elvira (FNAL) and A. Dotti (SLAC)

Implementation details and choices

- Use detailed profiling to identify CPU-heavy algorithms to demonstrate performance gains from vectorization
- Redesign data structures to promote vectorization with minimal overhead (*SoA structures*)
- Use templated types to write generic kernels to be instantiated using scalar or vector types as needed
- VecCore package to isolate the complexities of vectorization implementation from algorithms
- Benchmark every vectorized class, for close performance monitoring
- Validate physics simulation results with respect to Geant4

Profiler/OpenISpeedshop

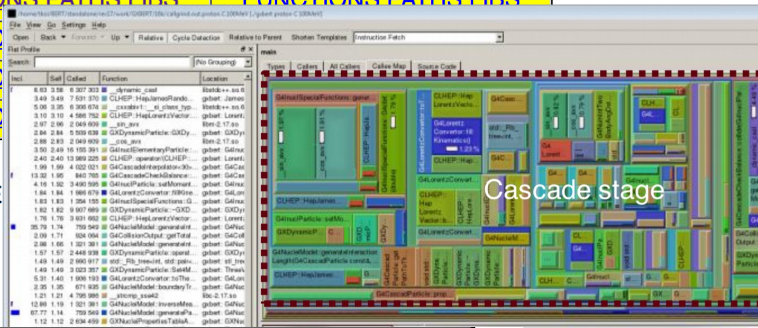
GXBERT (shared libs)

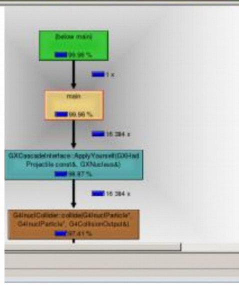
CPU profiling reports

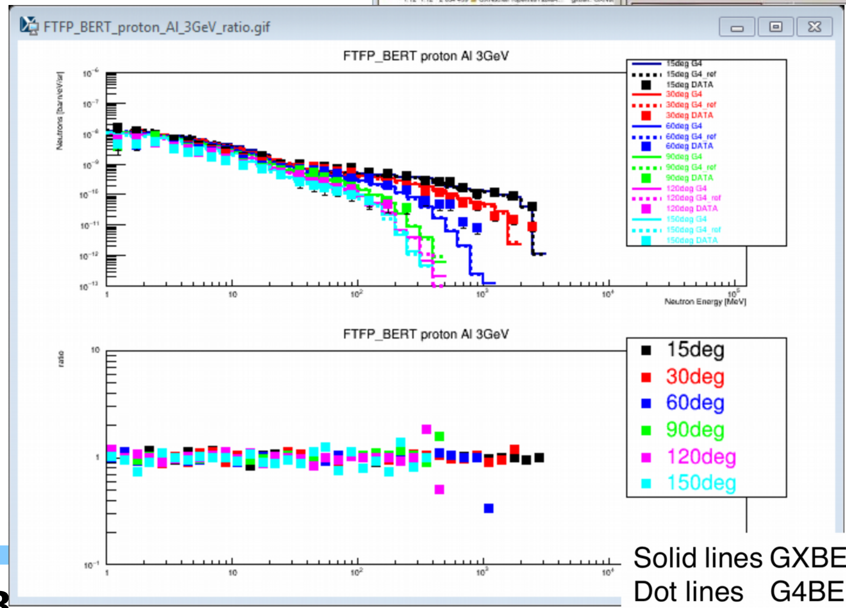
- FUNS: program counter @100Hz: EXCLUSIVE time for functions
- PATH: call path counter @35Hz: INCLUSIVE time for functions
- LIBS: libraries counter (LIBS)

Energy=1.5GeV	Carbon (C)	Lead (Pb)
gamma	FUNCTIONS PATHS LIBS	FUNCTIONS PATHS LIBS
pi-	FUNCTIONS PATHS LIBS	FUNCTIONS PATHS LIBS
proton	FUNCTIC	
neutron	FUNCTIC	
Lambda	FUNCTIC	

Processor: Intel(R) Xeon(R) CPU E







Solid lines GXBERT
Dot lines G4BERT
Boxes Data

Fermilab

Progress on Bertini vectorization

- Combining a top-down approach...
  - Vectorizing function interfaces (passing SIMD-vectors down into algorithms)
  - Vectorized utilities (e.g. rotations, Lorentz boosts, ...) and data structures (InuclParticle and InuclElementaryParticle classes)
  - Processing flow: lots of sanity checks and triage based on particle types

    → assume homogeneous SIMD-vector inputs – e.g. [p][C] becomes [pp...p] onto [CC...C]

    → hadron-hadron, hadron-nucleus, nucleus-nucleus collisions (algorithm functions)

- ... and a bottom-up approach
  - Generic kernels for generating multiplicity, particle types, kinematics (momenta, angles)
    - hadron-hadron collisions: class G4ElementaryParticleCollider (lots of non-trivial functions)
  - Math functions – Log, Exp, Pow, Factorial, LogFactorial
    - Not obvious how to vectorize, e.g. Pow([x1, x2, x3, x4], [n1, n2, n3, n4])
    - Eventually moved into VecMath, if useful outside of Bertini
  - Currently vectorizing the functions to generate multiplicities and FS types, and their validation tests and benchmarks.
    (I hope to have new benchmark numbers available by the time of Collab. Meeting)

- Next pages, pseudo-code is used to illustrate vectorization progress, and rationale behind suggestions for algorithmical changes

Class G4ElementaryParticleCollider

Functions: generateSCMfinalState( ), generateMultiplicity( ), generateOutgoingPartTypes( )

```
** class G4ElementaryParticleCollider : public G4CascadeColliderBase : public G4VCascadeCollider
*** Constructor: trivial
*** G4ElementaryParticleCollider::collide(bullet, target, output)...
*** G4ElementaryParticleCollider::generateSCMfinalState(ekin, etot_scm, particle1, particle2)
+ loop to generate valid final state (itry_max = 10)
  + clear output vectors
  + multipl = generateMultiplicity(initState, ekin)          <=== [vectorized]
  + generateOutgoingPartTypes(initState, multipl, ekin)     <=== [vectorizable (imultipl?)]
  + fillOutgoingMasses()   // cache each particle mass, mass2        [vectorized (imultipl?)]
    // Attempt to produce final state kinematics
  + fsGenerator.Configure(particle1, particle2, particle_kinds);       <=== [being vectorized (imultipl?)
  + generate = !fsGenerator.Generate(etot_scm, masses, scm_momentums);  <=== [being vectorized]

+ if any problems, return (no valid final state)
+ store final state particles (and their SCM kinematics)
+ return

*** G4ElementaryParticleCollider::generateMultiplicity(initState, ekin)
+ xsecTable = G4CascadeChannelTables::GetTable(initState);
+ if (xsecTable) multipl = xsecTable->getMultiplicity(ekin);
+ return multipl;

*** G4ElementaryParticleCollider::generateOutgoingPartTypes(initState, multipl, ekin)
+ particle_kinds.clear()        // Initialize buffer for generation
+ xsecTable = G4CascadeChannelTables::GetTable(initState)
+ xsecTable->getOutgoingParticleTypes(particle_kinds, mult, ekin)
+ return particle_kinds
```

Are these possible to be vectorized? *Maybe*, if and only if multiplicity is homogeneous

Experiment with intra-algorithm re-basketization

G4CascadeFinalStateAlgorithm class

```
** class GXCascadeFinalStateAlgorithm : public GXVHadDecayAlgorithm
*** GXCascadeFinalStateAlgorithm::Configure(bullet, target, particle_kinds)
{
  // Identify initial and final state (if two-body) for algorithm selection
  multiplicity = particle_kinds.size();        <=== must be same multiplicity
  G4int is = bullet->type() * target->type();
  G4int fs = (multiplicity==2) ? particle_kinds[0]*particle_kinds[1] : 0;

  ChooseGenerators(is, fs);    <=== probably vectorizable IFF particle_kinds[2] is homogeneous

  // Save kinematics for use with distributions
  SaveKinematics(bullet, target);      <=== [vectorized]

  // Save particle types for use with distributions
  kinds = particle_kinds;
}

*** GXCascadeFinalStateAlgorithm::ChooseGenerators(bullet, target, particle_kinds)
{
  // Choose generator for momentum
  if (G4CascadeParameters::usePhaseSpace()) momDist = 0;
  else momDist = G4MultiBodyMomentumDist::GetDist(is, multiplicity);

  // Choose generator for angle
  if (fs > 0 && multiplicity == 2) {
    G4int kw = (fs==is) ? 1 : 2;
    angDist = G4TwoBodyAngularDist::GetDist(is, fs, kw);
  } else if (multiplicity == 3) {
    angDist = G4TwoBodyAngularDist::GetDist(is);
  } else {
    angDist = 0;
  }
}
```

Several different objects returned depending on
is (initial state),
fs (final state)
and multiplicity

Redesigning for vectorization

- Requesting new supporting tools
  - Vectorization efficiency requires homogeneous lanes (for maximum lane synchronization)

- Hadronic processes tend to diverge quickly
  - GeantV baskets: homogeneous input arrays for simulation
    - e.g. [pp...p] on [Scint, Scint, ... Scint]
    - Bertini: protons will collide with either C-atoms or H-atoms

    → rebasketizing here will promote higher levels of lane synchronization

  - from previous slide: multiplicity-based basketization is particularly important for Bertini algos
    - both final state and kinematics sampling algorithms are based on multiplicity

    → rebasketizing by multiplicity promotes the development of more efficient Bertini kernels

  → proposed functionality: intra-algorithm (or algorithm-level) re-basketization

- Another challenge: dealing with Vector<int> and Vector<double> in the same algorithms
  - VcVector<long int> is not supported by Vc library
  - Work-around: using Int_v = VcSimdArray<VectorSize<Real_v>> to create SIMDVectors of *int*s corresponding to *doubles*

  → best long-term solution: native suport from VecCore
    - to be discussed with VecCore developers

Vectorizing math functions

- Modularized gxbert code contains some “fast math functions”
  - tabulated exp(x) for integer or half-integer x, truncated $O(x^3)$ Taylor series for |x|<84, otherwise use VDT implementation (internal vectorization)
  - tabulated log(x) for integers up to 512, otherwise use VDT implementation
  - specialized Pow(x,y) for integer x or y, etc…

- Hard to implement fully vectorized versions
  - e.g. Pow([x1, x2, x3, x4], [n1, n2, n3, n4])
  - vectorize interface only, [Pow(x1,n1), Pow(x2,n2), Pow(x3,n3), Pow(x4,n4)]
    - scalar functions are called once per lane, to build the SIMD vector
    - this is actually how it is done in VecCore, e.g. Sin(), Cos(), Abs(), …
    - slower than original implementation due to SIMD storing overhead
  - the vectorized interface is useful to simplify vectorization of mathematical expressions involving such functions (maybe worthy the overhead)
  - some vectorization is possible on the “fast” versions

- See next pages for some performance comparisons

Two illustrative preliminary results

- Unit test for InuclElementaryParticle

```
lima@mac: build 🍺 ./TestInuclElementaryParticle
=== GXInuclElemParticles:  Particles=[proton; neutron; gamma; deuteron] masses=[938.272; 939.565; 0; 1875.61] types=<1 2 9 41> ekin=[1073.52, 925.289, 827.232
, 1155.82]
4 tracks: <1 2 9 41>
 kinE=[1073.52, 925.289, 827.232, 1155.82]
 totE=[1073.52, 925.289, 827.232, 1155.82]
 nucleon:m[1100]
 pion:m[0000]
 photon:m[0010]
 baryon:<1 1 0 2>
 strange:<0 0 0 0>
 quasi_deutron(): m[0000]
=== GXInuclElemParticles:  Particles=[pi+; pi-; diproton; dineutron] masses=[139.57; 139.57; 1876.54; 1879.13] types=<3 5 111 122> ekin=[1073.52, 925.289, 827
.232, 1155.82]
4 tracks: <3 5 111 122>
 kinE=[1073.52, 925.289, 827.232, 1155.82]
 totE=[1213.09, 1064.86, 2703.78, 3034.95]
 nucleon:m[0000]
 pion:m[1100]
 photon:m[0000]
 baryon:<0 0 2 2>
 strange:<0 0 0 0>
 quasi_deutron(): m[0011]
>>> GXInuclElementaryParticle tests passed.
```

New SoA data structures can handle particles of different types

- Benchmark for GXLorentzConvertor (~4x faster)

```
lima@mac: build 🍺 ./LorentzConvertorBenchmark 3.0 1048576 10
GXBert results:   sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 100.117
Scalar results:   sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 63.2348
Vector size: 4
Vector results:   sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 14.7479
VectorL result:   sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 14.6649


GXBert results:   sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 100.376
double results: sumEscm = 1.96957e+09   sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 63.1925
Double_v results: sumEscm = 1.96957e+09 sumEkin = 3.75451e+06   sumP2 = 9.23118e+11   CPUtime = 14.451
```

Benchmarking math functions

- Preliminary measurements of *relative performance* (AVX)
  - Original: Geant4 “fast” implementations (global/management)
  - Scalar, Vector: my “vectorized interface” versions, templated on scalar or vectorized types, calling Geant4 “fast” implementations
  - ScalarStd, VectorStd: same as above, but calling std::functions instead of the Geant4 “fast” implementations

```
* ./ExpABenchmark: exp(x)

ExpA() Benchmark: nReps=100 and nvals=2097152

 Original ExpA(): 3.53498e+227 1334.87 msec
   Scalar ExpA(): 3.53498e+227 1339.55 msec
   Vector ExpA(): 3.53498e+227 1365.73 msec
Scalar ExpAVec(): 3.53498e+227 1455.18 msec
Vector ExpAVec(): 3.53498e+227 626.344 msec
ScalarStd ExpA(): 3.53498e+227 1241.94 msec
VectorStd ExpA(): 3.53498e+227 1184.67 msec
```

~2x

```
* ./LogZBenchmark: log(n)

LogZ() Benchmark: nReps=100 and nvals=2097152

 Original LogZ(): 8.73641e+06 239.454 msec
   Scalar LogZ(): 8.73641e+06 530.022 msec
   Vector LogZ(): 8.73641e+06 238.823 msec
ScalarStd LogZ(): 8.73641e+06 1096.34 msec
VectorStd LogZ(): 8.73641e+06 1183.11 msec
```

~4x

```
* ./PowZBenchmark: Z^x

PowZ() Benchmark: nReps=100 and nvals=2097152

 Original PowZ(): 2.01829e+17 1335.01 msec
   Scalar PowZ(): 2.01829e+17 1692.68 msec
   Vector PowZ(): 2.01829e+17 1446.95 msec
ScalarStd PowZ(): 2.01835e+17 3893.61 msec
VectorStd PowZ(): 2.01835e+17 3844.93 msec
```

~2x

- Preliminary conclusion: overhead of vectorized interface is significant, but it is probably worth the convenience.
- In some cases, the *fast* Geant4 implementation is not better than the standard version, so we can use it for those cases.

Current status

- What has been accomplished
  - Git repository available: https://github.com/gxbert/gxbert.git
  - Basic infrastructure for development, unit testing and performance evaluation (v01 done)
  - New SoA data structure for tracks and kinematics (v01 done, extensions needed for nuclei)
  - Vectorized ThreeVectors (a la CLHEP) and LorentzVectors (done)
  - Basic algorithms for Lorentz boosts (Lab frame ↔ projectile ↔ center of mass frame) as needed (done)
    - measured speedups of up to 4x in avx mode (theo.max = 4) w.r.t. scalar mode
    - additional 25% gain (scalar vs. G4), due to less branching and better memory locality
  - Integration of our vectorized pRNG (pseudo-Random Number Generator) (done, not yet VecMath)

Prospects

- What can be done in the short- to medium-term (h-h interactions)?
  - Currently vectorizing algorithms that handle hadron-hadron interactions
  - Finalize vectorized interfaces of all processing flow (under way)
  - Vectorization of all algorithms which can deal with homogeneous input (under way)
  - Unit tests and benchmarks for vectorized functions (partly done)
  - I am more optimistic now than at the beginning of this project.

- What requires more time
  - Full cascade algorithms – it is a long process, because of the large number of non-trivial functions involved.
    [see backup slides for more details on the Bertini processing flow]
  - Supporting tools will be very helpful
    - Intra-algorithm re-basketization in GeantV
    - Native support to Vector<double> ← → Vector<int> in VecCore
  - Full vectorized prototype corresponding to Tatsumi's tests for hadron-nucleus toy experiments, showing some speedup due to vectorization (not started)
  - Vectorization of hadron-nucleus and nucleus-nucleus processes (is Bertini used for those?)
  - profiling-based optimization of vectorized algorithms

- Assessment of performance gains

Backup slides

Bertini processing flow

- Start from Tatsumi's example, gxbertTest, which:
  - Sets up a large number of homogeneous collisions (e.g. projectiles(=protons) on targets(=Lead)
  - calls GXCascadeInterface::ApplyYourself(bullet,target) for each pair

```
** class GXCascadeInterface
*** GXCascadeInterface::construtor()
*** GXCascadeInterface::ApplyYourself(GXHadProjectile, GXNucleus)
+ sanity checks
+ fill bullet params
  either hadronBullet:  G4InuclElementaryParticle    <=== [vectorized]
      or hadronNucleus: G4InuclNuclei
+ fill target params
  either targetBullet:  G4InuclElementaryParticle    <=== [vectorized]
      or targetNucleus: G4InuclNuclei

+ loop {
    collider->collide(bullet, target, output)     <=== [being vectorized] (more details below)
    balance->collide(bullet, target, output)
  } until( 20 iterations or final state valid )

+ rotate final state back to lab frame            <=== [vectorized]
+ convert result (final state) from
  Bertini format         to      GX format
  ---------------------          ----------------------------
   vector<G4InuclEP>              vector<GXDynamicParticle*>
   vector<G4InuclNuclei>
   vector<GXFragment>
```

Class G4InuclCollider

We try to simplify complex inheritance structures

```
** class G4InuclCollider : public G4CascadeColliderBase : public G4VCascadeCollider
- theElementaryParticleCollider:  G4ElementaryParticleCollider*
- theIntraNucleiCascader:         G4IntraNucleiCascader*
- theDeexcitation:                G4VCascadeDeexcitation* theDeexcitation;      // User switchable!
- output:    G4CollisionOutput  // Secondaries from main cascade
- DEXoutput: G4CollisionOutput  // Secondaries from de-excitation

*** InuclCollider::constructor()
+ instantiate theEPCollider [being vectorized]  and theINuclCascader

*** InuclCollider::collide()
+ if(UseEPCollider(bullet,target)) {
    theEPCollider->collide();   <=====  [being vectorized] (more details later)
    return;
  }
+ else { #.. at least one nucleus is present
    (target must always be a nucleus)
    + classify bullet as a hadron or a nucleus (and fill zbullet parameters)
    + boost to TRS (Target Rest System)
    + call theInuclCascader->collide(zbullet, target, output)
    + deexcite then remove recoil fragment
    + boost to LAB frame
    + save final state into output (G4CollisionOutput) object
    + adjust final state kinematics to balance energy and momentum
  }
  + if anything is wrong, call G4CollisionOutput::trivialise()
  return;
```

hadron-hadron collisions

hadron-nucleus or
nucleus-nucleus collisions

Class G4ElementaryParticleCollider

This class has a large number of non-trivial functions!

Function: collide( )

```
** class G4ElementaryParticleCollider : public G4CascadeColliderBase : public G4VCascadeCollider
*** Constructor: trivial
*** G4ElementaryParticleCollider::collide(bullet, target, output)
+ if(UseEPCollider(bullet, target)) # unnecessary
+ instantiate interCase = InteractionCase(bullet, target)   <== [vectorized]
+ sanity checks
+ instantiate and fill G4LorentzConvertor     <=== [vectorized]
+ boost to center of mass frame               <=== [vectorized]
+ if a nucleon() is involved, then {          <=== [vectorized]
    if (pionNucleonAbsorption(ekin)) {        <=== [vectorized]
      generateSCMpionNAbsorption(etot_scm, particle1, particle2);   <=== [partly vectorized, not tested]
    } else {
      generateSCMfinalState(ekin, etot_scm, particle1, particle2);   <=== [main function, being vectorized]
    }
  }

+ if a quasi_deutron() is involved {                  <=== [vectorized]
    if (particle1->isMuon() || particle2->isMuon()) {  <=== [vectorized]
      generateSCMmuonAbsorption(etot_scm, particle1, particle2);
    } else {            // Currently, pion absoprtion also handles gammas
      generateSCMpionAbsorption(etot_scm, particle1, particle2);
    }
  }

+ if no valid final state produced so far, return!
+ loop over final state particles
    + boost to Lab frame
+ validate final state for energy and momentum conservation
+ sort FS particles by kinetic energy
+ returns final state particles (output)
```

Plans to re-write these steps with vectorization in mind, to profit from vectorized boosts. Originally, all secondaries are stored in an std::vector.

Class G4ElementaryParticleCollider

Functions: generateSCMfinalState( ), generateMultiplicity( ), generateOutgoingPartTypes( )

```
** class G4ElementaryParticleCollider : public G4CascadeColliderBase : public G4VCascadeCollider
*** Constructor: trivial
*** G4ElementaryParticleCollider::collide(bullet, target, output)...
*** G4ElementaryParticleCollider::generateSCMfinalState(ekin, etot_scm, particle1, particle2)
+ loop to generate valid final state (itry_max = 10)
  + clear output vectors
  + multipl = generateMultiplicity(initState, ekin)          <=== [vectorized]
  + generateOutgoingPartTypes(initState, multipl, ekin)    <=== [vectorizable (imultipl?)]
  + fillOutgoingMasses()   // cache each particle mass, mass2        [vectorized (imultipl?)]
    // Attempt to produce final state kinematics
  + fsGenerator.Configure(particle1, particle2, particle_kinds);        <=== [being vectorized (imultipl?)
  + generate = !fsGenerator.Generate(etot_scm, masses, scm_momentums);  <=== [being vectorized]

+ if any problems, return (no valid final state)
+ store final state particles (and their SCM kinematics)
+ return

*** G4ElementaryParticleCollider::generateMultiplicity(initState, ekin)
+ xsecTable = G4CascadeChannelTables::GetTable(initState);
+ if (xsecTable) multipl = xsecTable->getMultiplicity(ekin);
+ return multipl;

*** G4ElementaryParticleCollider::generateOutgoingPartTypes(initState, multipl, ekin)
+ particle_kinds.clear()        // Initialize buffer for generation
+ xsecTable = G4CascadeChannelTables::GetTable(initState)
+ xsecTable->getOutgoingParticleTypes(particle_kinds, mult, ekin)
+ return particle_kinds
```

Are these possible to be vectorized? *Maybe*, iff multiplicity is homogeneous

Experiment with intra-algorithm re-basketization

G4CascadeFinalStateGenerator class

```
** class GXHadDecayGenerator
*** GXHadDecayGenerator::Generate(G4double initialMass,
                                   const std::vector<G4double>& masses,
                                   std::vector<G4LorentzVector>& finalState)
{
  if (masses.size() == 1U) {
    return GenerateOneBody(initialMass, masses, finalState);
  }
  else {
    theAlgorithm->Generate(initialMass, masses, finalState);
  }
  return !finalState.empty();          // Generator failure returns empty state
}

** class G4CascadeFinalStateGenerator : public GXHadDecayGenerator
*** GXCascadeFinalStateGenerator::Configure(particle1, particle2, particle_kinds)
+ cascadeFinalStateAlg->Configure(bullet, target, particle_kinds)

** class GXVHadDecayAlgorithm
*** GXVHadDecayAlgorithm::Generate(G4double initialMass,
                                    const std::vector<G4double>& masses,
                                    std::vector<G4LorentzVector>& finalState)
{
  // Initialization and sanity check
  finalState.clear();
  if (!IsDecayAllowed(initialMass, masses)) return;

  // Allow different procedures for two-body or N-body distributions
  if (masses.size() == 2U) {
    GenerateTwoBody(initialMass, masses, finalState);     <=== [vectorizable]
  }
  else {
    GenerateMultiBody(initialMass, masses, finalState);   <=== [hard, probably partially vectorizable]
  }
}
```

G4CascadeFinalStateAlgorithm class

```
** class GXCascadeFinalStateAlgorithm : public GXVHadDecayAlgorithm
*** GXCascadeFinalStateAlgorithm::Configure(bullet, target, particle_kinds)
{
  // Identify initial and final state (if two-body) for algorithm selection
  multiplicity = particle_kinds.size();        <=== must be same multiplicity
  G4int is = bullet->type() * target->type();
  G4int fs = (multiplicity==2) ? particle_kinds[0]*particle_kinds[1] : 0;

  ChooseGenerators(is, fs);    <=== probably vectorizable IFF particle_kinds[2] is homogeneous

  // Save kinematics for use with distributions
  SaveKinematics(bullet, target);      <=== [vectorized]

  // Save particle types for use with distributions
  kinds = particle_kinds;
}

*** GXCascadeFinalStateAlgorithm::ChooseGenerators(bullet, target, particle_kinds)
{
  // Choose generator for momentum
  if (G4CascadeParameters::usePhaseSpace()) momDist = 0;
  else momDist = G4MultiBodyMomentumDist::GetDist(is, multiplicity);

  // Choose generator for angle
  if (fs > 0 && multiplicity == 2) {
    G4int kw = (fs==is) ? 1 : 2;
    angDist = G4TwoBodyAngularDist::GetDist(is, fs, kw);
  } else if (multiplicity == 3) {
    angDist = G4TwoBodyAngularDist::GetDist(is);
  } else {
    angDist = 0;
  }
}
```

Several different objects returned depending on is (initial state), fs (final state) and multiplicity