

EM Vectorization in GV.

17/04/2018

Vitalii Drohan

When vectorization can be useful.

Functions with a lot of math computations. Such as + * / sqrt sin cos exp log
(ordered according to approximate computation complexity)

Function with minimal branching. Branching force us to evaluate both branches for vectorized code.

Functions not bounded by memory access. Load 4 doubles into simd register is one instruction but it is not faster than loading values one by one.

Maximum speedup.

Generally not equal to vector register width because some operations are slower for vector registers.

Example: Inverse throughput for scalar division of doubles for SandyBridge CPU = 10-22 cycles. For vector division(AVX - 4 doubles) = 22-44. Hence maximum speedup for division will be ~ 2 for this CPU even though SIMD register width is 4.

Other important factor is number of execution units for particular instructions = number of instructions that can be executed simultaneously.

Frequency can go down by 10-50% if core uses avx512 instructions.

Math function benchmarks.

To get an idea about what speedup one can run.

`./bin/benchmarks/vecphys/MathBench` (uses google benchmarks)

Used to compare time to run simple math function on array of random doubles. To prevent auto-vectorization we compute sum while running loop.

Math benchmarks for SandyBridge(AVX)

Benchmark	Time	CPU Iterations		
ScalarExp/256	2126 ns	2124 ns	329966	
VectorExp/256	1026 ns	1025 ns	679434	2 times faster
ScalarLog/256	2669 ns	2661 ns	268482	
VectorLog/256	1020 ns	1017 ns	678943	2.6 times faster
ScalarSqrt/256	948 ns	946 ns	739281	
VectorSqrt/256	484 ns	482 ns	1460892	2 times faster
ScalarDiv/256	3722 ns	3718 ns	186312	
VectorDiv/256	1927 ns	1920 ns	363572	
ScalarSin/256	3233 ns	3229 ns	186522	
VectorSin/256	3074 ns	3064 ns	233650	Almost the same
ScalarCos/256	3667 ns	3656 ns	189328	
VectorCos/256	3096 ns	3088 ns	222168	
ScalarSinCos/256	7232 ns	7211 ns	101049	
VectorSinCos/256	3222 ns	3218 ns	212480	

Math benchmarks for Intel Xeon Platinum (AVX)

Benchmark	Time	CPU Iterations	
ScalarExp/256	2199 ns	2194 ns	322911
VectorExp/256	2363 ns	2357 ns	297650
ScalarLog/256	2458 ns	2452 ns	284679
VectorLog/256	1709 ns	1704 ns	414046
ScalarSqrt/256	606 ns	604 ns	1163093
VectorSqrt/256	213 ns	213 ns	3293941
ScalarDiv/256	1101 ns	1098 ns	637509
VectorDiv/256	544 ns	543 ns	1293428
ScalarSin/256	2292 ns	2287 ns	248492
VectorSin/256	5488 ns	5475 ns	129699
ScalarCos/256	2560 ns	2554 ns	258115
VectorCos/256	5612 ns	5599 ns	121026
ScalarSinCos/256	5105 ns	5093 ns	135239
VectorSinCos/256	6133 ns	6119 ns	112900

Negative speedup.

Only 40% faster

3 times faster

2 times faster

2 times slower

Long vs short vectorization

Typical code looks like this.

- 1) $a = a(x)$
- 2) $b = b(a)$

There are two general ways on how to vectorize it.

- 1) Have x, a, b as an arrays and do each step in loop over all values in array
Uses temporary arrays, more flexible(you can sort values in arrays) good for combining scalar code with SIMD one.
- 2) Have x, a, b as SIMD variables and write code as it is written without SIMD
Simple, works well in most cases

How to handle branching.

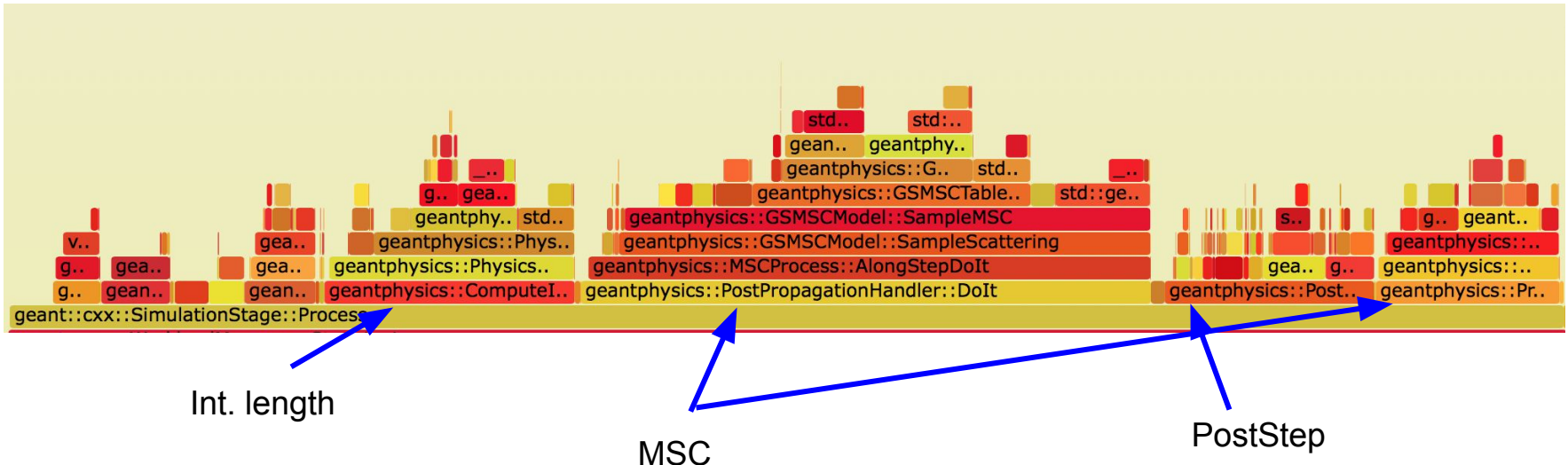
- 1) Prefilter particles to different arrays if branch is heavy.
- 2) Evaluate both branches if branch is light

```
double electronTotE;  
double positronTotE;  
if (td->fRndm->uniform() > 0.5) {  
    electronTotE = (1. - eps) * ekin;  
    positronTotE = eps * ekin;  
} else {  
    electronTotE = eps * ekin;  
    positronTotE = (1. - eps) * ekin;  
}
```

```
PhysDV electronTotE, positronTotE;  
PhysDM tmpM = td->fRndm->uniformV() > 0.5;  
vecCore::MaskedAssign(electronTotE, tmpM, (1. - eps) * ekin);  
vecCore::MaskedAssign(positronTotE, tmpM, eps * ekin);  
  
vecCore::MaskedAssign(positronTotE, !tmpM, (1. - eps) * ekin);  
vecCore::MaskedAssign(electronTotE, !tmpM, eps * ekin);
```


EM Physics

- 1) Multiple scattering (30%+10% of TestEM3)
- 2) Interaction length computation (14%) if TestEM3. Do not contain a lot of math computation, most of the time it is looking up value in a table.
- 3) Final state sampling. (11% of TestEM3) This part will be discussed.



Final state sampling.

- 1) After particle made step we should decide if it will undergo final state sampling.
- 2) Sample final particles energy and direction, create tracks for them.

There are two parts of this inside GeantV:

- 1) GeantV framework part: when we filter tracks, create light tracks from them, call physics models
- 2) Model part: specific code for particular process model.

Final state sampling for scalar case.

How it was done before:

- In PostStepAction stage PostStepAction handler will select appropriate process based on particle type and region(or return early if delta interaction happened)
- Process then will select model based on energy range and sample final state using that model.
- Handler will pass light track to models and models will return model to handler(through TaskData)

If we will basketize that handler we would not be able to basketize it, because we will end with basket full of different particles/models/processes.

Final state sampling for vectorized EM models

How it is done now. Functionality is the same but we move things around.

- We have the same kind of stage `PostStepActionPhysModelStage` but now there is one handler per EM model. `PostStepActionPhysModelHandler`
- In stage select method we chose which process and model will be used for particle so that in handler when we will call `DoIt` we will be sure that this particle will be sampled with corresponding model.
- Handler will pass SOA of `LightTracks` to model and model will pass SOA of `LightTracks` back to handler.

Sampling inside model.

Now we can implement function with similar signature.

- Scalar:

```
EMModel::SampleSecondaries(LightTrack& primary, TaskData* td)
```

```
returns -> std::vector<LightTracks> secondaries (inside task data)
```

- Vector:

```
EMModel::SampleSecondariesVector(LightTrack_v& primaries, TaskData* td)
```

```
returns -> LightTrack_v secondaries (inside task data)
```

Typical model.

- 1) Select appropriate alias table based on current particle energy/material and sample some reduced quantity from it(i.e. fraction of transferred energy to electron in pair creation) OR sample this quantity with accept/reject method.
- 2) With this sampled quantity compute kinematics of primary/secondary after interaction(kin. energy and direction)
- 3) Update primary LT
- 4) Add secondary LTs to TaskData storage.

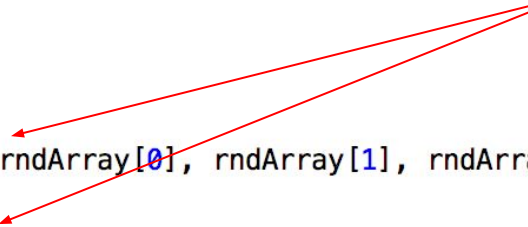
Positron to 2 Gamma example

1) sample some reduced quantity

```
int PositronTo2GammaModel::SampleSecondaries(LightTrack &track, geant::TaskData *td)
{
    int numSecondaries = 0;
    // sample gamma energy
    const double pekin = track.GetKinE();

    const double tau = pekin / geant::units::kElectronMassC2; // E_kin of the e+ in rest mass units
    const double tau2 = tau + 2.;
    const double gamma = tau + 1.;
    double eps = 0.;
    if (GetUseSamplingTables()) {
        double *rndArray = td->fDblArray;
        td->fRndm->uniform_array(3, rndArray);
        eps = SampleEnergyTransfer(pekin, gamma, rndArray[0], rndArray[1], rndArray[2]);
    } else {
        eps = SampleEnergyTransfer(gamma, td);
    }
}
```

Sampling done here



Positron to 2 Gamma example

```
void VecPositronTo2GammaModel::SampleSecondariesVector(LightTrack_v &tracks, geant::TaskData *td)
```

```
{  
    int N = tracks.GetNtracks();  
    PhysicsModelScratchpad &data = td->fPhysicsData->fPhysicsScratchpad;  
    double *epsArr = data.fEps;  
    double *gamma = data.fDoubleArr; // Used by rejection
```

← Temporary storage arrays

```
    for (int i = 0; i < N; i += kPhysDVWidth) {  
        PhysDV pekin = tracks.GetKinEVec(i);  
        PhysDV gammaV = pekin * geant::units::kInvElectronMassC2 + 1.0;  
        if (GetUseSamplingTables()) {  
            PhysDV rnd1 = td->fRndm->uniformV();  
            PhysDV rnd2 = td->fRndm->uniformV();  
            PhysDV rnd3 = td->fRndm->uniformV();  
            PhysDV epsV = SampleEnergyTransferAlias(pekin, rnd1, rnd2, rnd3, gammaV);  
            vecCore::Store(epsV, &epsArr[i]);  
        } else {  
            vecCore::Store(gammaV, &gamma[i]);  
        }  
    }  
    if (!GetUseSamplingTables()) {  
        gamma[N] = gamma[N - 1];  
        SampleEnergyTransferRej(gamma, epsArr, N, td);  
    }  
}
```

← Sampling and storing result in tmp. array.

Positron to 2 Gamma example

2) With this sampled quantity compute kinematics

```
// direction of the first gamma
double ct      = (eps * tau2 - 1.) / (eps * std::sqrt(tau * tau2));
const double cost = std::max(std::min(ct, 1.), -1.);
const double sint = std::sqrt((1. + cost) * (1. - cost));
const double phi  = geant::units::kTwoPi * td->fRndm->uniform();
double gamDirX    = sint * std::cos(phi);
double gamDirY    = sint * std::sin(phi);
```

Positron to 2 Gamma example

```
for (int i = 0; i < N; i += kPhysDVWidth) {  
    PhysDV pekin = tracks.GetKinEVec(i);  
    PhysDV tau   = pekin * geant::units::kInvElectronMassC2;  
    PhysDV tau2  = tau + 2.0;  
    PhysDV eps; vecCore::Load(eps, &epsArr[i]);  
    // direction of the first gamma  
    PhysDV ct      = (eps * tau2 - 1.) / (eps * Sqrt(tau * tau2));  
    const PhysDV cost = Max(Min(ct, (PhysDV)1.), (PhysDV)-1.);  
    const PhysDV sint = Sqrt((1. + cost) * (1. - cost));  
    const PhysDV phi  = geant::units::kTwoPi * td->fRndm->uniformV();  
    PhysDV sinPhi, cosPhi;  
    SinCos(phi, &sinPhi, &cosPhi);  
    PhysDV gamDirX = sint * cosPhi;  
    PhysDV gamDirY = sint * sinPhi;
```

Loop over array of sampled eps.

Positron to 2 Gamma example

3) Add secondary LTs to TaskData storage

```
LightTrack &gamma1Track = td->fPhysicsData->InsertSecondary();
gamma1Track.SetDirX(gamDirX);
gamma1Track.SetDirY(gamDirY);
gamma1Track.SetDirZ(gamDirZ);
gamma1Track.SetKinE(gamEner);
gamma1Track.SetGVcode(fSecondaryInternalCode); // gamma GV code
gamma1Track.SetMass(0.0);
gamma1Track.SetTrackIndex(track.GetTrackIndex()); // parent Track index
//
```

Positron to 2 Gamma example

```
for (int l = 0; l < kPhysDVWidth; ++l) {  
    int idx = secondaries.InsertTrack();  
    secondaries.SetKinE(gamEner[l], idx);  
    secondaries.SetDirX(gamDirX[l], idx);  
    secondaries.SetDirY(gamDirY[l], idx);  
    secondaries.SetDirZ(gamDirZ[l], idx);  
    secondaries.SetGVcode(fSecondaryInternalCode, idx);  
    secondaries.SetMass(0.0, idx);  
    secondaries.SetTrackIndex(tracks.GetTrackIndex(i + l), idx);  
}
```

← Looping over
lanes of SIMD
to insert
secondary

Alias sampling

- 1) From energy compute index of alias table(vectorized)
- 2) Each particle will need to go to different table to get some quantity. (non vectorized, loop over SIMD lanes)
- 3) Compute resulting value based on value from the table. Usually log/exp(vectorized)

Overall win for performance is significant for this part of calculation.

Rejection sampling.

Loop while condition is satisfied. Some particles will finish earlier. This issue can be fixed with lane refilling.

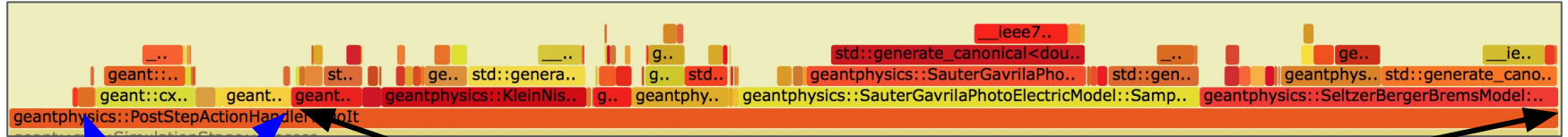
- 1) Prepare values that are needed for sampling in form of arrays.
- 2) Keep SIMD of indexes in array that have to be sampled in this loop iteration.
- 3) Gather values from array using this indexes.
- 4) Sample. Scatter resulting value to array with corresponding indexes.
- 5) If values are accepted, replace finished indexes with next if there is no next - fill index with $\text{MAXINDEX}+1$. (for that in input arrays add fake element after all data)
- 6) Goto 3 until everything is done.

Results. Microbenchmarks.

./bin/benchmarks/vephys/*

Model	SandyBridge	
	Time	Speedup
KleinNishina alias table	90	2.4
KleinNishina reject/accept	70	2.9
BetheHeitler alias table	156	2.32
BetheHeitler reject/accept	155	1.8
BetheHeitler LPM alias table	150	2.12
BetheHeitler LPM reject/accept	230	1.65
Heitler alias table	90	2.5
Heitler reject/accept	68	2.56
MollerBhabba alias table	96	2.18
MollerBhabba reject/accept	100	1.96
Seltzer-Berger alias table	135	2.15
Seltzer-Berger reject/accept	240	1.66
LPM (brems) alias table	150	2
LPM (brems) reject/accept	571	2

Sampling inside framework before.



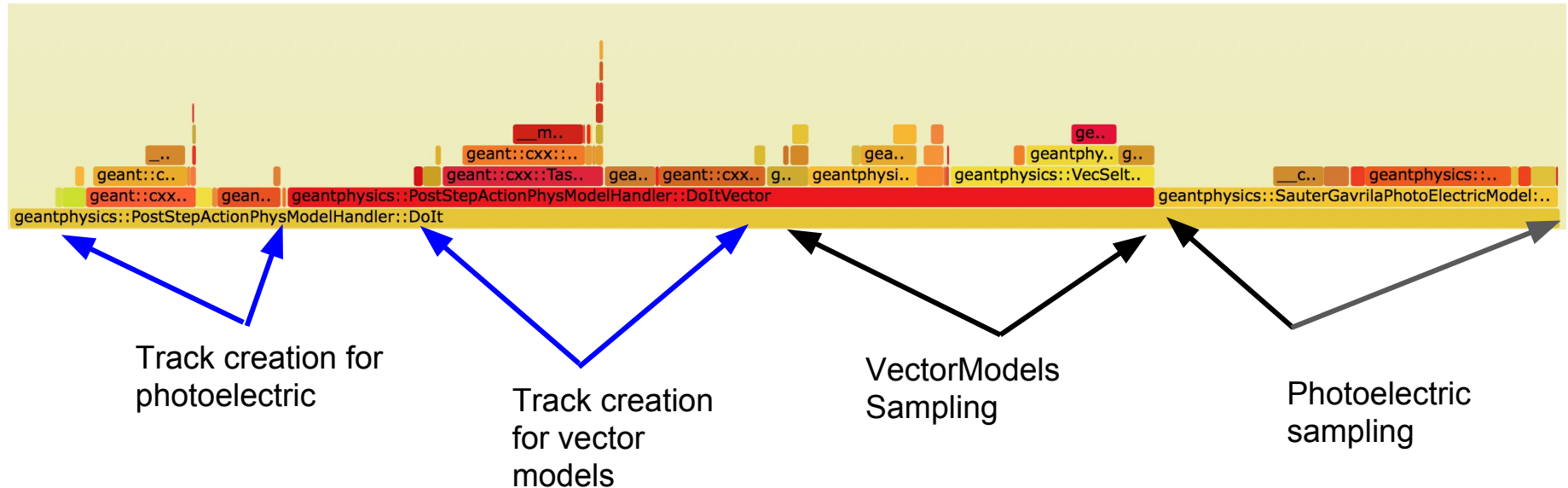
Track
creation

Models sampling

Results. Inside framework.

Speedup is comparable to what is expected.

I.e. SeltzerBergerBrems scalar = 1.88% vector = 0.96%



Conclusion

git checkout vidrohan/postacthandlers-physics-filter

- 1) `./bin/benchmarks/vecphys/*` (only if google benchmark is installed)
- 2) There are model level statistical tests for each vectorized model in `./bin/tests/vecphys/*`

Histograms for important values + energy , momentum conservations testing