Vectorization of CMSSW offline software

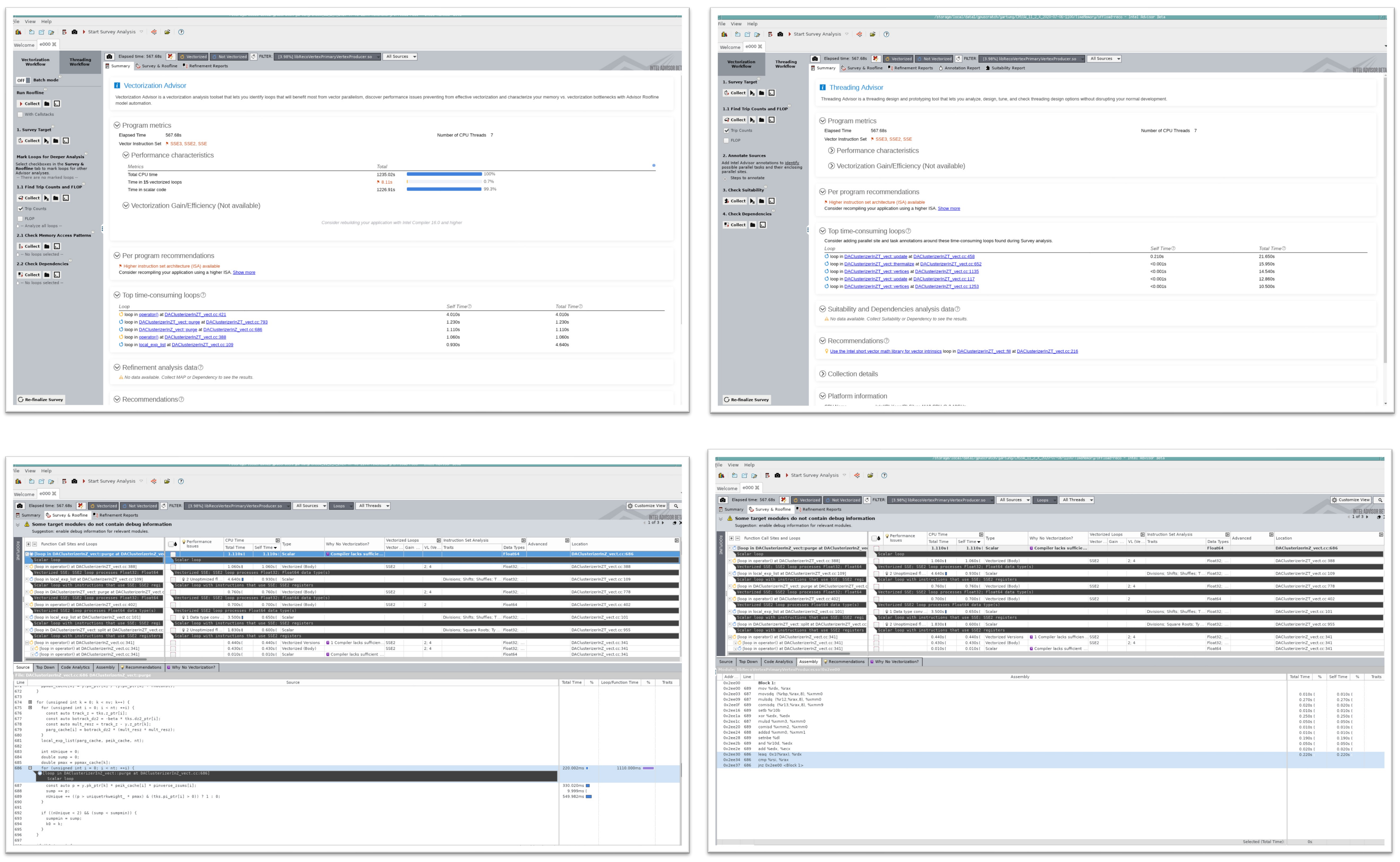
Dr Patrick Gartung[1] on behalf of the CMS Collaboration

[1] Fermi National Accelerator Laboratory

FERMILAB-POSTER-23-029-CMS-CSAID

Profiling with Intel's OneAPI Toolset

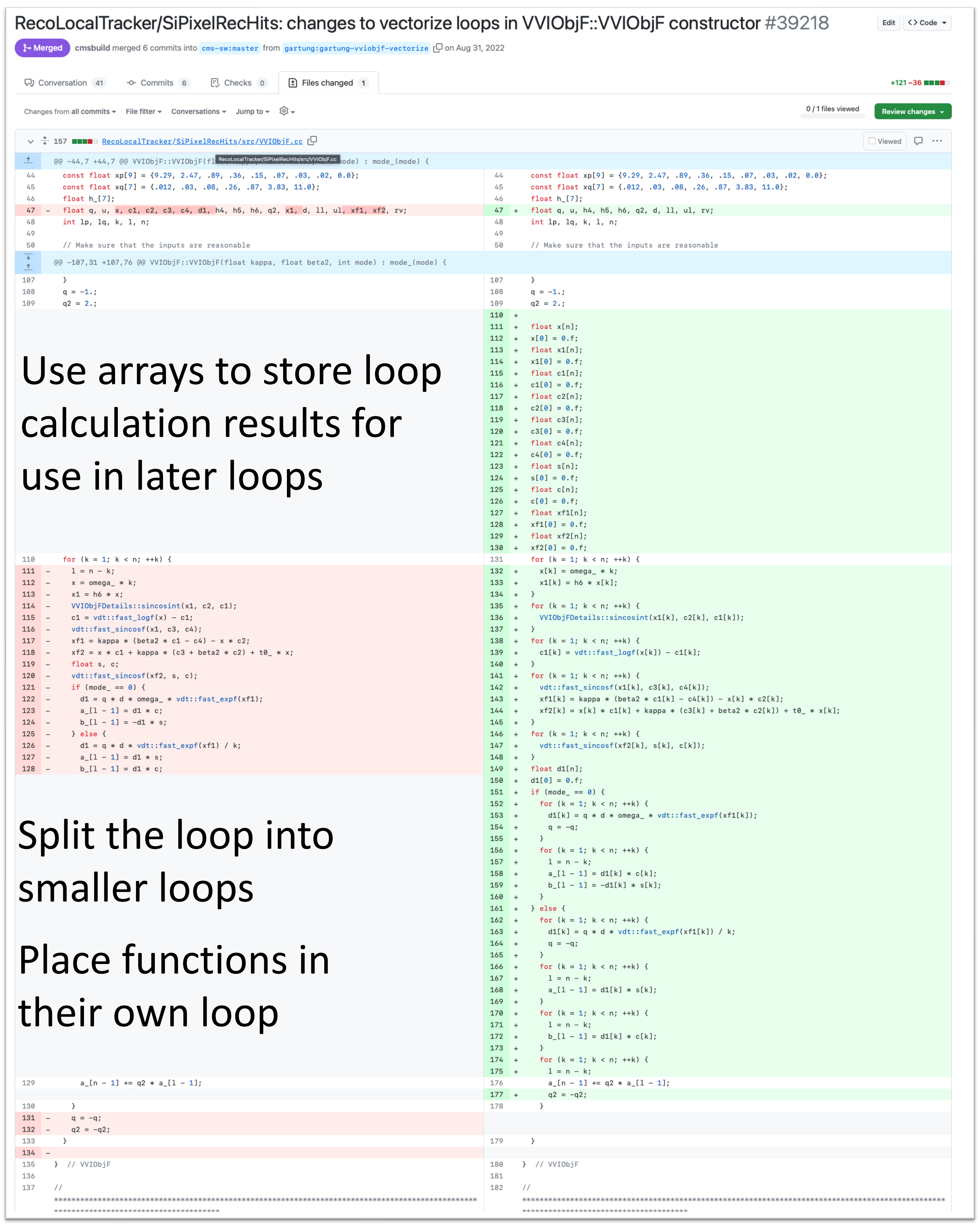
CMSSW uses Intel's Thread Building Blocks (TBB) library. Intel Vtune is used to profile the CMSSW application as it is the only profiler that can stitch together TBB thread stack frames. Intel Advisor is used to identify scalar loops with floating point operations that can potentially be replaced by SIMD instructions – auto-vectorization. Intel's legacy compiler ICC and small vector math library (SVML) can produce more vectorization with the SSE3 instruction set but are not validated for use in CMSSW.



Screen caps of intel Advisor showing Vectorization Advisor pane, Threading Advisor pane, Survey pane with Source view of loop with timing, and Survey pane with Assembly view of loop with timing.

Example of changes to vectorize

An example of changes made to auto-vectorize a loop with GCC and the SSE3 instruction set



RecoLocalTracker/SiPixelRecHits: changes to vectorize loops in VVIObjF::VVIObjF constructor #39218

Merged cmsbuild merged 6 commits into `cms-sw:master` from `gartung:gartung-vviobjf-vectorize` on Aug 31, 2022

Conversation 41 · Commits 6 · Checks 0 · Files changed 1 · +121 −36

157 `RecoLocalTracker/SiPixelRecHits/src/VVIObjF.cc`

```
@@ -44,7 +44,7 @@ VVIObjF::VVIObjF(fl RecoLocalTracker/SiPixelRecHits/src/VVIObjF.cc ode) : mode_(mode) {
     const float xp[9] = {9.29, 2.47, .89, .36, .15, .07, .03, .02, 0.0};
     const float xq[7] = {.012, .03, .08, .26, .87, 3.83, 11.0};
     float h_[7];
-    float q, u, x, c1, c2, c3, c4, d1, h4, h5, h6, q2, x1, d, ll, ul, xf1, xf2, rv;
+    float q, u, h4, h5, h6, q2, d, ll, ul, rv;
     int lp, lq, k, l, n;

     // Make sure that the inputs are reasonable
@@ -107,31 +107,76 @@ VVIObjF::VVIObjF(float kappa, float beta2, int mode) : mode_(mode) {
     }
     q = -1.;
     q2 = 2.;
```

Use arrays to store loop calculation results for use in later loops

```
+
+    float x[n];
+    x[0] = 0.f;
+    float x1[n];
+    x1[0] = 0.f;
+    float c1[n];
+    c1[0] = 0.f;
+    float c2[n];
+    c2[0] = 0.f;
+    float c3[n];
+    c3[0] = 0.f;
+    float c4[n];
+    c4[0] = 0.f;
+    float s[n];
+    s[0] = 0.f;
+    float c[n];
+    c[0] = 0.f;
+    float xf1[n];
+    xf1[0] = 0.f;
+    float xf2[n];
+    xf2[0] = 0.f;
```

Removed:

```
     for (k = 1; k < n; ++k) {
-      l = n - k;
-      x = omega_ * k;
-      x1 = h6 * x;
-      VVIObjFDetails::sincosint(x1, c2, c1);
-      c1 = vdt::fast_logf(x) - c1;
-      vdt::fast_sincosf(x1, c3, c4);
-      xf1 = kappa * (beta2 * c1 - c4) - x * c2;
-      xf2 = x * c1 + kappa * (c3 + beta2 * c2) + t0_ * x;
-      float s, c;
-      vdt::fast_sincosf(xf2, s, c);
-      if (mode_ == 0) {
-        d1 = q * d * omega_ * vdt::fast_expf(xf1);
-        a_[l - 1] = d1 * c;
-        b_[l - 1] = -d1 * s;
-      } else {
-        d1 = q * d * vdt::fast_expf(xf1) / k;
-        a_[l - 1] = d1 * s;
-        b_[l - 1] = d1 * c;
```

Split the loop into smaller loops

Place functions in their own loop

```
         a_[n - 1] += q2 * a_[l - 1];

       }
-      q = -q;
-      q2 = -q2;
     }
-
   }  // VVIObjF

   //
   ***********************************************************************************************
   ****************************************
```

Added:

```
     for (k = 1; k < n; ++k) {
+      x[k] = omega_ * k;
+      x1[k] = h6 * x[k];
+    }
+    for (k = 1; k < n; ++k) {
+      VVIObjFDetails::sincosint(x1[k], c2[k], c1[k]);
+    }
+    for (k = 1; k < n; ++k) {
+      c1[k] = vdt::fast_logf(x[k]) - c1[k];
+    }
+    for (k = 1; k < n; ++k) {
+      vdt::fast_sincosf(x1[k], c3[k], c4[k]);
+      xf1[k] = kappa * (beta2 * c1[k] - c4[k]) - x[k] * c2[k];
+      xf2[k] = x[k] * c1[k] + kappa * (c3[k] + beta2 * c2[k]) + t0_ * x[k];
+    }
+    for (k = 1; k < n; ++k) {
+      vdt::fast_sincosf(xf2[k], s[k], c[k]);
+    }
+    float d1[n];
+    d1[0] = 0.f;
+    if (mode_ == 0) {
+      for (k = 1; k < n; ++k) {
+        d1[k] = q * d * omega_ * vdt::fast_expf(xf1[k]);
+        q = -q;
+      }
+      for (k = 1; k < n; ++k) {
+        l = n - k;
+        a_[l - 1] = d1[k] * c[k];
+        b_[l - 1] = -d1[k] * s[k];
+      }
+    } else {
+      for (k = 1; k < n; ++k) {
+        d1[k] = q * d * vdt::fast_expf(xf1[k]) / k;
+        q = -q;
+      }
+      for (k = 1; k < n; ++k) {
+        l = n - k;
+        a_[l - 1] = d1[k] * s[k];
+      }
+      for (k = 1; k < n; ++k) {
+        l = n - k;
+        b_[l - 1] = d1[k] * c[k];
+      }
+      for (k = 1; k < n; ++k) {
+        l = n - k;
         a_[n - 1] += q2 * a_[l - 1];
+        q2 = -q2;
       }

     }

   }  // VVIObjF

   //
   ***********************************************************************************************
   ****************************************
```

Screencap of partial diff from Github pull request for CMSSW https://github.com/cms-sw/cmssw/pull/39218/files

CMSSW multi microarchitecture releases and performance

For production, CMSSW is optimized for the SSE3 instruction set, the lowest common instruction set. Optimizing for higher instruction sets and vector widths enables more loops to be vectorized potentially increasing performance. The CMSSW build system was enhanced to produce three sets of libraries. These are compiled with the GCC march flags sse3, haswell and skylake-avx512. The library set is chosen automatically unless overridden by an environment variable.

Micro-architecture	GEN-SIM	DIGI-HLT	RECO
default	0.079 ev/sec per thread	0.087 ev/sec per thread	0.082 ev/sec per thread
haswell	0.090 ev/sec per thread	0.099 ev/sec per thread	0.077 ev/sec per thread
skylake-avx512	0.080 ev/sec per thread	0.087 ev/sec per thread	0.078 ev/sec per thread

Throughput results on a 50% loaded node for a Run 3 workflow for micro-architectures 11th Gen Intel Core i7 i11700 @ 2.50Ghz

Micro-architecture	GEN-SIM	DIGI-HLT	RECO
default	0.056 ev/sec per thread	0.053 ev/sec per thread	0.045 ev/sec per thread
haswell	0.051 ev/sec per thread	0.053 ev/sec per thread	0.053 ev/sec per thread
skylake-avx512	0.052 ev/sec per thread	0.052 ev/sec per thread	0.052 ev/sec per thread

Throughput results on a 100% loaded node for a Run 3 workflow for micro-architectures 11th Gen Intel Core i7 i11700 @ 2.50Ghz

The use of higher vector widths can cause the boost clock to be lowered for thermal management[1]. For the 50% loaded RECO process, the throughput is lower for higher vector widths. This indicates that the boost clock was used but lowered. For the 100% loaded RECO process, the throughput is higher for higher vector widths. This indicates that the boost clock was not used and the improvement from higher vector widths can be seen.

[1] https://cvw.cac.cornell.edu/vector/performance_turbo

This document was prepared by the CMS Collaboration using the resources of the Fermi National Accelerator Laboratory (Fermilab), a U.S. Department of Energy, Office of Science, HEP User Facility. Fermilab is managed by Fermi Research Alliance, LLC (FRA), acting under Contract No. DE-AC02-07CH11359.