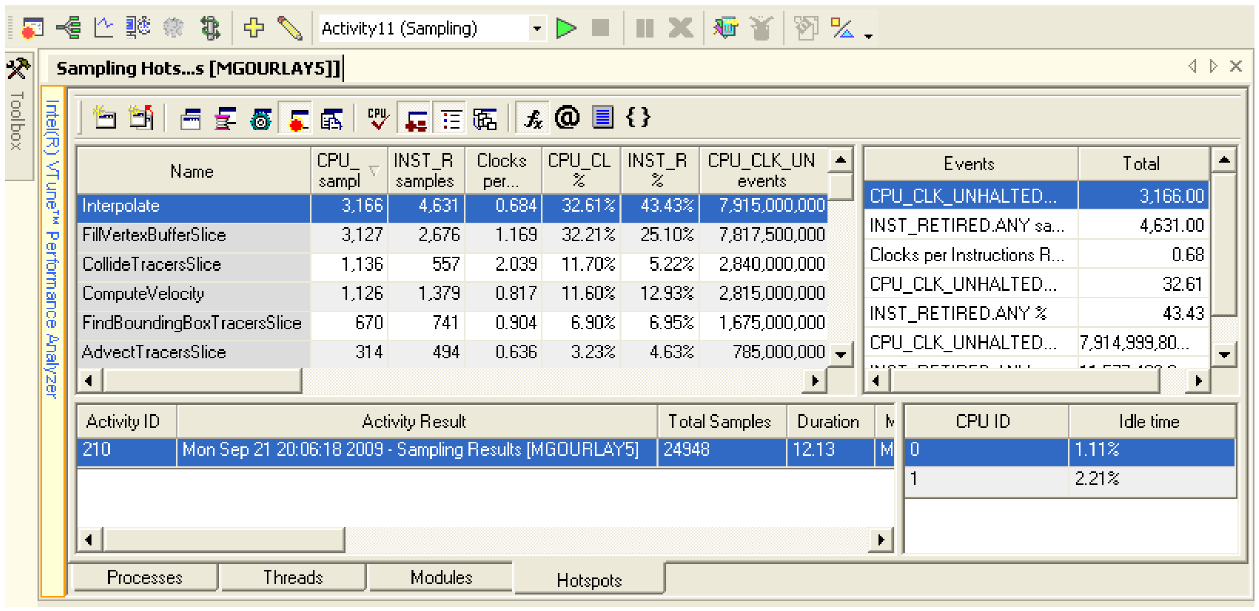
# Fluid Simulation for Video Games (part 5)

By Dr. Michael J. Gourlay



Sampling Hots...s [MGOURLAY5]

| Name | CPU_ sampl | INST_R samples | Clocks per... | CPU_CL % | INST_R % | CPU_CLK_UN events |
|---|---|---|---|---|---|---|
| Interpolate | 3,166 | 4,631 | 0.684 | 32.61% | 43.43% | 7,915,000,000 |
| FillVertexBufferSlice | 3,127 | 2,676 | 1.169 | 32.21% | 25.10% | 7,817,500,000 |
| CollideTracersSlice | 1,136 | 557 | 2.039 | 11.70% | 5.22% | 2,840,000,000 |
| ComputeVelocity | 1,126 | 1,379 | 0.817 | 11.60% | 12.93% | 2,815,000,000 |
| FindBoundingBoxTracersSlice | 670 | 741 | 0.904 | 6.90% | 6.95% | 1,675,000,000 |
| AdvectTracersSlice | 314 | 494 | 0.636 | 3.23% | 4.63% | 785,000,000 |

| Events | Total |
|---|---|
| CPU_CLK_UNHALTED... | 3,166.00 |
| INST_RETIRED.ANY sa... | 4,631.00 |
| Clocks per Instructions R... | 0.68 |
| CPU_CLK_UNHALTED... | 32.61 |
| INST_RETIRED.ANY % | 43.43 |
| CPU_CLK_UNHALTED... | 7,914,999,80... |

| Activity ID | Activity Result | Total Samples | Duration | M |
|---|---|---|---|---|
| 210 | Mon Sep 21 20:06:18 2009 - Sampling Results [MGOURLAY5] | 24948 | 12.13 | M |

| CPU ID | Idle time |
|---|---|
| 0 | 1.11% |
| 1 | 2.21% |

## Profiling and Optimization

This article, the fifth in a series, describes the profiling and optimization of a fluid simulation, presented in the third and fourth articles. The first article summarized fluid dynamics; the second surveyed fluid simulation techniques; and the third and fourth presented a vortex-particle fluid simulation with two-way fluid-body interactions, which runs in real time. This article exploits yet another feature of Intel® Threading Building Blocks (Intel® TBB) to spread more work across multiple threads. This article describes a process for profiling CPU usage and uses that information to optimize and further parallelize the code so that it runs faster.

Much of the process of profiling and optimization described in this article mirrors the procedures described in *The Software Optimization Cookbook.* That process starts with creating a benchmark—a chunk of code used to quantify the performance of the algorithm being optimized. The rest of the process entails iterating on three steps:

1. Profile to identify so-called "hotspots," where the application spends most of its time.
2. Investigate the details of why the hotspot consumes so much time.
3. Modify code to try to make it faster.

This article chronicles applying these steps to the fluid simulation presented in the previous two articles.

## Benchmark

The benchmark should provide repeatable results so that a profiling tool obtains the same results each time. The test cases provided with the fluid simulation use a random number generator to initialize the fluid properties; formulating a proper benchmark required seeding that random number generator so that the simulation produced exactly the same results for each run. Calling `srand` with a constant value, from within `InteSiVis::InitialConditions` accomplished this.

I used Intel® VTune to perform many of the profiles. Although not necessary, it helps simplify the process if the benchmark automatically terminates after a fixed, constant duration. I therefore added code in `GlutDisplayCallback` that automatically terminates the process after a fixed number of updates. This code is only compiled when the pre-compiler macro `PROFILE` is defined with a value exceeding 1.

When the simulation code is built in benchmark mode, profiling the application is fast and easy. The application automatically runs the benchmark test case, executes the predefined number of iterations, and then terminates. The process is automated and easy to run, and the execution is identical for each run, all of which are important features to have in a benchmark.

## Profile

I used two tools to perform CPU profiling: performance counters and Intel® VTune. The kernel provides a pair of system calls to facilitate precise measurement of elapsed time. The `QueryPerformanceCounter` function obtains the current value of a high-resolution timer in units of cycles, and the `QueryPerformanceFrequency` function obtains the frequency (in cycles per second) of that timer. The simulation code includes macros that simplify the use of these timers. To time a section of code, place `QUERY_PERFORMANCE_ENTER` just before the start of the region and `QUERY_PERFORMANCE_EXIT` just after the region, where the `EXIT` macro also takes an argument used to label the region. These macros measure the duration of each iteration that the region executes, along with average durations of that region. This provides a quick and easy scheme for measuring aggregate code performance and is embedded in the code, so that it can be used even when fancier tools are not available.

Table 1 shows timing data for the simulation as it ran on a dual 2.33 GHz quad-core Intel® Xeon® processors, varying the number of cores used by the simulation. The shaded routines are parallelized using Intel® TBB, as described in articles three and four. As expected, those routines run faster as the number of cores increases. In contrast, the execution times of other routines, which have not been written to exploit multiple threads, do not vary significantly with the number of cores. In fact, because this process ran on a multiprocessing operating system, variations in runtime have less to do with this process in isolation and more to do with miscellaneous other processes running concurrently on the same machine during these timings. Also note that some of the entries in the table represent aggregates of other routines; so naturally, when those aggregates contain parallelized routines, the aggregate timings also vary with the number of threads.

**Table 1.** Benchmark Timings Before Optimization

| Code region | | | | Duration (ms) when executed on # CPUs | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 | 2 | 4 | 8 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | Vortons | 0.01 | 0.01 | 0.01 | 0.01 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | Tracers | 3.63 | 3.45 | 3.35 | 3.35 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | | 3.65 | 3.47 | 3.36 | 3.37 |
| VortonSim | CreateInfluenceTree | MakeBaseVortonGrid | | 0.05 | 0.05 | 0.05 | 0.05 |
| VortonSim | CreateInfluenceTree | AggregateClusters | | 0.04 | 0.04 | 0.04 | 0.04 |
| VortonSim | CreateInfluenceTree | | | 3.83 | 3.66 | 3.56 | 3.57 |
| VortonSim | ComputeVelocityGrid | | | 5.30 | 2.27 | 1.58 | 1.11 |
| VortonSim | StretchAndTiltVortons | | | 0.23 | 0.24 | 0.24 | 0.25 |
| VortonSim | DiffuseVorticityPSE | | | 0.24 | 0.24 | 0.24 | 0.25 |
| VortonSim | AdvectVortons | | | 0.05 | 0.05 | 0.05 | 0.06 |
| VortonSim | AdvectTracers | | | 18.51 | 12.71 | 7.12 | 3.94 |
| FluidBodySim | VortonSim_Update | | | 26.73 | 18.58 | 12.38 | 8.89 |
| FluidBodySim | SolveBoundaryConditions | | | 4.04 | 3.21 | 3.13 | 3.14 |
| FluidBodySim | RbSphere_UpdateSystem | | | 0.00 | 0.00 | 0.00 | 0.00 |
| InteSiVis | FluidBodySim_Update | | | 30.78 | 21.80 | 15.53 | 12.05 |
| InteSiVis | Render_SetMaterial | | | 0.01 | 0.01 | 0.01 | 0.01 |
| ParticlesRender | FillVertexBuffer | | | 12.48 | 8.06 | 5.83 | 4.73 |
| ParticlesRender | Draw | | | 11.82 | 11.76 | 11.77 | 11.97 |
| ParticlesRender | | | | 24.31 | 19.83 | 17.61 | 16.71 |
| InteSiVis | Render_SwapBuffers | | | 0.24 | 0.29 | 0.26 | 0.27 |
| InteSiVis | Render | | | 24.73 | 20.30 | 18.05 | 17.17 |
| InteSiVis | | | | 55.31 | 41.95 | 33.50 | 29.15 |

A look at Table 1 suggests that the `FindBoundingBox` routine could benefit from parallelization, because it takes a considerable fraction of the simulation time—38% when the number of cores is 8—and a look at its code reveals that it is a straightforward loop, ripe for data parallelism.

Intel® VTune is a sophisticated profiling tool that uses *sampling*—that is, it periodically interrupts the system to record performance information, such as the current instruction pointer and various event counters. This technique allows the profiler to obtain timing and other information for virtually every machine instruction in the program, so you can tell not only which functions occupy a lot of time but which line of code within the function and indeed which instruction within the line of code. Meanwhile, because the sampling occurs relatively infrequently, it does not unduly interfere with the execution of the algorithm.

**Note** Intel® VTune can also instrument code to provide call graphs. This functionality can be valuable in more complicated code, but this simulation has a simple software architecture. As a result, this article omits use of instrumenting and call graph features. Intel® VTune also supports programming the performance monitoring unit (PMU) to obtain far more detailed information about low-level details such as memory bandwidth and cache usage.

One of the nice features of a sampling profile is that it does not require any setup within the code. For example, you can just build the application, and then tell Intel® VTune to run your application. Then, you "surf" through Intel® VTune to discover where your program runs slowly.

Intel® VTune can provide information about many events, and the wealth of data could become overwhelming. David Levinthal's article "Cycle Accounting Analysis on Intel® Core™2 Processors" provides an in-depth analysis of how to interpret sampling data. Intel® VTune provides a "program" of sampling experiments that it runs on your code automatically and that includes calibration and data collection for dozens of events, such as cache misses, branch mispredictions, and many other events. To get started, let's keep things simple and look at just two quantities: CPU cycles and instructions retired.

Table 2 shows a tiny subset of the sampling data that Intel® VTune provides for a few hotspots. The **CPU_CLK** column indicates counts of the core CPU cycles in the unhalted state, which is approximately the number of CPU cycles spent in the routine. The **INST_RETIRED** column indicates the number of instruction retired, and **CPI** indicates the cycles per instruction. A general rule of thumb is that when CPI > 1, the routine is operating significantly less efficiently than the processor is capable of.

**Table 2.** Sampling Profiles Before and After Optimization

| | Before optimization | | | After optimization | | |
|---|---|---|---|---|---|---|
| **Name** | **CPU_CLK** | **INST_RETIRED** | **CPI** | **CPU_CLK** | **INST_RETIRED** | **CPI** |
| FillVertexBufferSlice | 3,291 | 2,971 | 1.108 | 3,166 | 2,656 | 1.192 |
| Interpolate† | 2,679 | 3,013 | 0.889 | 3,104 | 4,613 | 0.673 |
| IndicesOfPosition† | 2,130 | 2,198 | 0.969 | † | † | † |
| ComputeVelocity | 1,617 | 1,686 | 0.959 | 1,158 | 593 | 1.953 |
| SolveBoundaryConditions | 1,134 | 587 | 1.932 | 1,014 | 1,264 | 0.802 |
| FindBoundingBox* | 692 | 908 | 0.762 | 677 | 737 | 0.919 |
| AdvectTracersSlice | 289 | 409 | 0.707 | 377 | 498 | 0.757 |

*After optimization, the original `FindBoundingBox` was split into three routines, and the vast majority of computation occurred in `FindBoundingBoxTracersSlice`, whose data appears in the "after optimization" columns.
†After optimization, `IndicesOfPosition` became inlined, primarily contributing to `Interpolate`.

Table 2 suggests that several routines warrant investigation, including the `FillVertexBuffer`, `Interpolate`, and `IndicesOfPosition` routines.

It is interesting to note that although `ComputeVelocityGrid` consumes a lot of time, it appears only sixth on this list, and it has a relatively good CPI, implying that it is not terribly inefficient. When you look at its code, that fact comes as somewhat of a surprise, because it is one of the most mathematically and logically complicated parts of the code and runs recursively. You might have expected that the function call overhead and branching logic would come at a huge price. But it turns out that it's not the most immediate source of concern.

The results of the profiling stage often come as a surprise. Before I ran these profiles, I had expected that the `ComputeVelocity` routine would consume most of the time, but it is fifth on the list. In practically every situation where I have profiled and optimized code, the profiles revealed a different picture of the situation than the one I predicted. This fact suggests that programmers should avoid premature optimization. Profiles, not guesses, should drive your optimization efforts.

## *Investigate*

The sampling summary indicates which routines to investigate, and Intel® VTune provides additional detail about precisely which sections of those routines run slowly. For example, the seemingly simple routine `IndicesOfPosition` consists of only three lines of code, which are simple assignment statements. Each of these lines of C code generates around 10 assembly instructions. Upon closer investigation, Intel® VTune reveals that, of these instructions, one in particular consumes a disproportionate number of cycles.

Understanding why code runs slowly requires a wide range of information about the CPU, which is beyond the scope of this article. But the books and articles listed in the "Further Reading" section provide a wealth of pertinent information.

Ultimately, the investigation phase should result in some hypotheses of what causes the hotspots. Intel® VTune can provide more detail by running a suite of tests. Upon clicking **Sampling**, the Sampling Configuration Wizard appears. Under "Tuning Assistance," I selected **Automatically generate tuning advice** and selected the option to generate over 30 different "experiments." For each experiment, Intel® VTune reprograms the PMU to collect data about a different "event." The validity of these experiments depends on having a repeatable, automated benchmark, as mentioned above. This article presents some of the details of this experiment in the "Hotspot" sections below.

## *Modify*

Armed with details about where the code spends most of its time and hypotheses on why, there are a wide variety of solutions to try. Here, optimization transitions from science to art, because finding the optimal solution entails a lot of trial and error. Do not expect your first attempt at optimization to succeed, and never assume that your code change improved execution time.

Modifications to try include these:

- ❑ Changing the algorithm to one with less complexity.
- ❑ Changing branching logic to avoid mispredicted branches.
- ❑ Making the code access data in a more cache-friendly manner.
- ❑ Avoiding inherently slow operations.
- ❑ Avoiding or replacing certain floating-point operations.
- ❑ Using intrinsic vector operations.
- ❑ Distributing algorithms across multiple threads.

You might naïvely assume that slow code results from using slow operations, but very often, code runs slowly because some portion of the code spends a lot of time idly waiting for memory to arrive. Simply knowing exactly what line of C code is the hotspot can imply that the operation performed is slow, but often the operation is not inherently slow: memory or instruction fetches are. The take-home point is that you should expect to try more than one idea when modifying your code.

### *Repeat*

Finally, each time you modify the code, repeat the entire profile–investigate–modify process. You will run through that loop dozens or hundreds of times, which emphasizes the point that the benchmark needs to be very quick and easy to run.

## Hotspots

The profile data indicates that the code spends most of its time on a few operations. This article focuses on just a few of them: `Rendering`, `Interpolate`, `IndicesOfPosition`, and `FindBoundingBox`.

### *Hotspot 1: Rendering*

Intel® VTune reports the following:

```
Execution may be bound by load operations: 0.54 Number of load
instructions per clockticks
     Advice:
•  Reduce the number of load operations.
```

The slowest part of this code comes from rendering. This series of articles focuses on simulation and side-steps rendering, because rendering fluids is its own large and complicated topic. The code provided with this series of articles does not even use programmable shaders or fancy features like “instancing,” both of which would result in monumental gains in speed.

Still, you can drastically improve the rendering speed simply by eliminating a useless operation that is simply an artifact of the OpenGL rendering application programming interface (API). Rendering vertices entails two steps: filling a buffer with vertices, then handing that buffer to the renderer to draw it. Intel® VTune reported that `DrvCopyContext` within the video driver consumed a large percentage of the total cycles for this process. You can’t change the video driver, but you can change how you use it. The original code used “vertex arrays” in which OpenGL copies that buffer to yet another buffer before rendering. Microsoft* DirectX* technology and more recent versions of OpenGL allow you to skip that step and write vertex data directly into the final destination. I changed the rendering routine to use OpenGL's vertex buffer objects (VBOs), which sped up the draw call from 11.83 ms to 0.027 ms—an approximately 450× improvement.

## Hotspot 2: Interpolate

The `Interpolate` routine is not necessarily inefficient (because its CPI is relatively low), but for simulations with a lot of particles—tracers in particular—the simulation spends a lot of time interpolating to obtain velocity values for each tracer. Intel® VTune reports the following:

```
Floating-point activity is significant: 6.7 % floating-point
retired operations.
    Advice:
•   Replace non-Single Instruction, Multiple Data (SIMD) code with
    SIMD code where possible.
•   Use the Intel® Math Kernel Libraries, if possible.
```

Two optimizations seem most likely to bear fruit: reduce the number of instructions, and replace the scalar operations with native SSE vector operations.

The original routine expressed the trilinear interpolation with this formula:

```
vResult = oneMinusTween.x * oneMinusTween.y * oneMinusTween.z * (*this)[ offsetX0Y0Z0 ]
        +         tween.x * oneMinusTween.y * oneMinusTween.z * (*this)[ offsetX1Y0Z0 ]
        + oneMinusTween.x *         tween.y * oneMinusTween.z * (*this)[ offsetX0Y1Z0 ]
        +         tween.x *         tween.y * oneMinusTween.z * (*this)[ offsetX1Y1Z0 ]
        + oneMinusTween.x * oneMinusTween.y *         tween.z * (*this)[ offsetX0Y0Z1 ]
        +         tween.x * oneMinusTween.y *         tween.z * (*this)[ offsetX1Y0Z1 ]
        + oneMinusTween.x *         tween.y *         tween.z * (*this)[ offsetX0Y1Z1 ]
        +         tween.x *         tween.y *         tween.z * (*this)[ offsetX1Y1Z1 ] ;
```

The replacement routine uses this instead:

```
vResult =      ( ( oneMinusTween.x * (*this)[ offsetX0Y0Z0 ]
                 +         tween.x * (*this)[ offsetX1Y0Z0 ] ) * oneMinusTween.y
               + ( oneMinusTween.x * (*this)[ offsetX0Y1Z0 ]
                 +         tween.x * (*this)[ offsetX1Y1Z0 ] ) * tween.y )
               * oneMinusTween.z
             + ( ( oneMinusTween.x * (*this)[ offsetX0Y0Z1 ]
                 +         tween.x * (*this)[ offsetX1Y0Z1 ] ) * oneMinusTween.y
               + ( oneMinusTween.x * (*this)[ offsetX0Y1Z1 ]
                 +         tween.x * (*this)[ offsetX1Y1Z1 ] ) * tween.y )
               * tween.z ;
```

I also replaced the `Vec3` routines with routines from Devir and Zohar's "Matrix Library"—but it resulted in no speed improvement, possibly because of the overhead associated with increased memory-alignment restrictions. The optimized code distributed with this article includes the vectorized version of `Vec3`, but it is disabled, because its use did not result in speed improvements.

Note that using the SSE instructions effectively requires that the memory address each `Vec3` is aligned be an integer multiple 16 bytes. This causes each `Vec3` to include padding for a phantom fourth component (which not used) and also requires changing all memory allocations to use 16-byte alignment. (Note that the STL `vector` container `resize` method infamously takes its “default value” argument by value, not by reference, which means that it is impossible to use an STL vector with objects that require peculiar alignments. To dodge this, I wrote my own.) The additional memory bandwidth and allocation overhead offset gains provided by using the SSE instructions.

At first glance, Table 2 *seems* to imply that `Interpolate` actually runs slower after optimization. But Table 1 and Table 3 clearly indicate otherwise. After optimization, `AdvectTracers` (which heavily relies on `Interpolate`) runs significantly faster. The next section explains that mystery.

## Hotspot 3: IndicesOfPosition

The number 3 hotspot, `IndicesOfPosition`, seems trivial:

```
void IndicesOfPosition( unsigned indices[3] , const Vec3 & vPosition ) const
{
    Vec3 vPosRel( vPosition - GetMinCorner() ) ;
    Vec3 vIdx( vPosRel.x * GetCellsPerExtent().x ,
               vPosRel.y * GetCellsPerExtent().y ,
               vPosRel.z * GetCellsPerExtent().z ) ;
    indices[0] = unsigned( vIdx.x ) ;
    indices[1] = unsigned( vIdx.y ) ;
    indices[2] = unsigned( vIdx.z ) ;
}
```

Intel® VTune tells you that a huge fraction of the time spent in this routine goes toward the float-to-int conversions, and indeed, this operation is well known to be slow. The Tuning Assistant reported the following:

```
Frequent Floating-Point Control Word Modifications.: 10.55 %
cycles when FP Control Word access stalled execution.
    Advice:
•  If your code requires float-to-int conversion, make sure you
   use fast float-to-int routines.
•  Avoid multiple Floating-point Control Word (FPCW) writes.
```

The problem comes from a conflict between the way the processor wants to do the conversion and how C prescribes how to do the conversion. Look at the assembly code for *one* of these lines:

```
   279:              indices[1] = unsigned( vIdx.y ) ;
004066F2  fld          dword ptr [esp+0Ch]    // Push vIdx.y onto FPU stack
004066F6  fnstcw       word ptr [esp+2]       // Store FPU control word onto runtime stack
004066FA  movzx        eax,word ptr [esp+2]   // eax = FPU control word
004066FF  or           ah,0Ch                 // Set FPU control word to truncation mode
00406702  mov          dword ptr [esp+4],eax  // Save new control word onto runtime stack
00406706  fldcw        word ptr [esp+4]       // Load new control word
0040670A  fistp        dword ptr [esp+4]      // Convert float to int and pop FPU stack
0040670E  mov          ecx,dword ptr [esp+4]  // ecx = integer
00406712  mov          dword ptr [edx+4],ecx  // Store integer
00406715  fldcw        word ptr [esp+2]       // Restore original FPU control word
```

By default, the floating-point unit (FPU) converts floating-point values to integers using rounding; but C specifies truncation. So each float-to-int conversion requires changing the FPU mode to "truncation," but to do that safely requires first saving the old control word, then restoring it after the conversion. You can see (but the compiler cannot) that you only need to change the control word once for all three conversions. The new code for *all three* conversions looks like this:

```
0040782F  fnstcw      word ptr [esp+10h]
00407833  mov         ax,word ptr [esp+10h]
00407838  or          ax,0C00h
0040783C  mov         word ptr [esp+0Ch],ax
00407841  fldcw       word ptr [esp+0Ch]
00407845  mov         eax,dword ptr [esp+10h]
00407849  mov         dword ptr [esp+18h],eax
0040784D  fld         dword ptr [esp+20h]
00407851  fistp       dword ptr [esp+14h]
00407855  fld         dword ptr [esp+24h]
00407859  fistp       dword ptr [esp+10h]
0040785D  fld         dword ptr [esp+28h]
00407861  fistp       dword ptr [esp+1Ch]
00407865  fnstcw      word ptr [esp+0Ch]
00407869  fldcw       word ptr [esp+18h]
```

In addition to reducing expensive FPU control word changes, the resulting routine is simple enough that `IndicesOfPosition` becomes inlined. The `Interpolate` routine calls `IndicesOfPosition`, so the cost subsequently gets tallied with `Interpolate`. That explains why `IndicesOfPosition` disappeared from the postoptimization profiles. That also explains why, despite the fact that interpolation, overall, runs faster, Intel® VTune reports that the optimized `Interpolate` routine consumes more CPU cycles than it did before. You can measure the aggregate improvement from changing `Interpolate` and `IndicesOfPosition` from the `AdvectTracers` routine, which went from 12.9 ms to 18.5 ms—an approximately 1.4× improvement.

## *Hotspot 4: ComputeVelocity*

Most of the time spent in the `ComputeVelocity` routine happens in `VORTON_ACCUMULATE_VELOCITY`:

```
{
    const Vec3          vNeighborToSelf     = vPosQuery - mPosition ;
    const float         radius2             = mRadius * mRadius ;
    const float         dist2               = vNeighborToSelf.Mag2() + sAvoidSingularity ;
    const float         oneOverDist         = 1.0f / sqrt( dist2 ) ;
    const Vec3          vNeighborToSelfDir  = vNeighborToSelf * oneOverDist ;
    const float         distLaw             = ( dist2 < radius2 )
                                                ?   /* Inside vortex core */
                                                ( oneOverDist / radius2 )
                                                :   /* Outside vortex core */
                                                ( oneOverDist / dist2 ) ;
    vVelocity +=  OneOverFourPi * ( 8.0f * radius2 * mRadius )
                  * mVorticity ^ vNeighborToSelf * distLaw ;
}
```

The Intel® VTune Tuning Assistant reports the following:

- Look at the assembly code to understand what optimizations the compiler is making.
- Consider replacing long-latency floating-point arithmetic operations with faster operations.

The `1/sqrt(dist2)` stands out as particularly expensive, because it includes both division and a square-root, both of which are expensive. Replacing that with an approximation (see the code for details) sped up this routine nearly 1.6×, from 5.30 ms to 3.36 ms.

Intel® VTune also reports that this routine has a lot of mispredicted branches. This is because of the unpredictable nature of recursively descending the octree. The problem requires a more sophisticated algorithm, which will be the topic of a future article.

## *Hotspot 5: SolveBoundaryConditions*

Intel® VTune reports that the following loop occupies the vast majority of time spent in the `SolveBoundaryCondition` routine:

```
Particle * pTracers = & mVortonSim.GetTracers()[ 0 ] ;
// Collide tracers with rigid body.
for( unsigned uTracer = 0 ; uTracer < numTracers ; ++ uTracer )
{   // For each tracer in the simulation...
    Particle &  rTracer          = pTracers[ uTracer ] ;
    const Vec3  vSphereToTracer = rTracer.mPosition - rSphere.mPosition ;
    const float fSphereToTracer = vSphereToTracer.MagnitudeFast() ;
    const float fCombinedRadii  = rTracer.mSize + rSphere.mRadius ;
    if( fSphereToTracer < fCombinedRadii )
    {   // Tracer is colliding with body. … }
}
```

You might guess that the problem comes from `Magnitude`, which involves a `sqrt`, but a closer look with Intel® VTune reveals the real trouble:

```
   394: for( unsigned uTracer = 0 ; uTracer < numTracers ; ++ uTracer )
00407329  test        edi,edi
0040732B  mov         eax,dword ptr [ecx+0BCh]
00407331  jbe         00407556
00407337  add         eax,14h
0040733A  mov         edx,edi
0040733C  lea         esp,[esp]
   395: {   // For each tracer in the simulation...
   396:     Particle &  rTracer         = pTracers[ uTracer ] ;
   397:     const Vec3  vSphereToTracer = rTracer.mPosition - rSphere.mPosition ;
00407340  fld         dword ptr [eax-14h]
00407343  movss       xmm0,dword ptr [eax+20h]
00407348  fsub        dword ptr [esi]    // SUPER SLOW
```

Intel® VTune reports that the `fsub` instruction that computes (`rTracer.mPosition.x-rSphere.mPosition.x`) runs slowly. The subtraction itself is very fast, so what is the problem? Intel® VTune reports that this instruction is waiting on memory to arrive. The hardware prefetch should be able to predict that it needs this memory, but the memory is not available when needed. Adding a software prefetch five iterations ahead improves that wait only modestly. What's going on? Because this loop is so computationally simple and accesses such a large amount of memory (thousands of tracers), the process is bound by memory bandwidth. You could speed it up by reducing the amount of memory it accesses or incorporating the same instructions into another loop that reads the same memory. Reducing the memory footprint of a particle would not be helpful, because the data accessed is set of tracer particles, whose structure is quite common in video games. So shrinking it here would make the results less relevant to real-world situations. And incorporating this routine into another would reduce modularity.

As a last resort, it is tempting to parallelize this process. But alas, because it is memory-bound, parallelization would not speed it up. Because Intel® TBB makes parallelization so easy, however, I went ahead and parallelized this process anyway, just to confirm my suspicions. Each iteration of this process potentially affects the same data (by applying impulses to the rigid body), so you must again use the map-reduce paradigm—that is, `parallel_reduce`. The results indicate that the process speeds up going from one to two CPUs (probably because that effectively doubles the L1 cache, as each core has its own), but going from two to four or four to eight CPUs does not improve throughput whatsoever, as expected.

## Hotspot 6: FindBoundingBox

The `FindBoundingBox` routine was neither terribly inefficient (its CPI was appreciably below 1) nor very "hot" (it consumed under 7% of the total cycles). But it was not threaded, so as the number of CPUs increased, `FindBoundingBox` occupied a greater fraction of the time. As earlier articles mentioned, parallelizing this routine is not trivial, because each iteration affects data "shared" across all iterations. Fortunately, however, combining that shared information is trivial. This routine follows the classic map-reduce pattern, which resembles "divide-and-conquer": the problem can be split into pieces, each piece can be solved separately, and the pieces can be joined efficiently. The "merge sort" algorithm also follows this pattern.

Parallelizing this routine with Intel® TBB `parallel_reduce` requires two parts: the usual “map” part, which earlier articles applied to other routines, and a new “reduce” part, which joins the results of the individual threads.

The `VortonSim_FindBoundingBoxTracers_TBB` class below shows a function for use with Intel® TBB `parallel_reduce`, which performs the same operations that `FindBoundingBox` did for tracers. The `VortonSim_FindBoundingBoxVortons_TBB` class performs the analogous operations for vortons, and you can find it in the source code that accompanies this article.

```
class VortonSim_FindBoundingBoxTracers_TBB
{
    public:
        VortonSim_FindBoundingBoxTracers_TBB( VortonSim * pVortonSim )
            : mVortonSim( pVortonSim )
            , mMin( pVortonSim->mMinCorner )
            , mMax( pVortonSim->mMaxCorner )
        {}
        VortonSim_FindBoundingBoxTracers_TBB( VortonSim_FindBoundingBoxTracers_TBB & that
            , tbb::split )
            : mVortonSim( that.mVortonSim )
            , mMin( that.mMin )
            , mMax( that.mMax )
        {}
        void operator() ( const tbb::blocked_range<size_t> & r )
        {   // Advect subset of tracers.
            const Particle * pTracers = & mVortonSim->GetTracers()[0] ;
            for( unsigned iTracer = r.begin() ; iTracer < r.end() ; ++ iTracer )
            {   // For each passive tracer particle in this simulation...
                const Particle & rTracer = pTracers[ iTracer ] ;
                // Find corners of axis-aligned bounding box.
                UpdateBoundingBox( rTracer.mPosition ) ;
            }
        }
        void join( const VortonSim_FindBoundingBoxTracers_TBB & other )
        {
            UpdateBoundingBox( other.mMin ) ;
            UpdateBoundingBox( other.mMax ) ;
        }
        Vec3 mMin ; ///< Bounding box minimum corner for vortons visited by this thread
        Vec3 mMax ; ///< Bounding box maximum corner for vortons visited by this thread
    private:
        void UpdateBoundingBox( const Vec3 & vPoint )
        {
            mMin.x = MIN2( mMin.x , vPoint.x ) ;
            mMin.y = MIN2( mMin.y , vPoint.y ) ;
            mMin.z = MIN2( mMin.z , vPoint.z ) ;
            mMax.x = MAX2( mMax.x , vPoint.x ) ;
            mMax.y = MAX2( mMax.y , vPoint.y ) ;
            mMax.z = MAX2( mMax.z , vPoint.z ) ;
        }
        VortonSim * mVortonSim ;    ///< Address of VortonSim object
} ;
```

## Results

Table 3 shows the final timing results from the optimized version of this simulation code.

## **Table 3.** Benchmark Timings After Optimization

| Code region | | | | Duration (ms) when executed on # CPU's | | | |
|---|---|---|---|---|---|---|---|
| | | | | 1 | 2 | 4 | 8 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | Vortons | 0.012 | 0.027 | 0.028 | 0.030 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | Tracers | 3.593 | 1.626 | 1.181 | 0.924 |
| VortonSim | CreateInfluenceTree | FindBoundingBox | | 3.613 | 1.660 | 1.217 | 0.961 |
| VortonSim | CreateInfluenceTree | MakeBaseVortonGrid | | 0.036 | 0.039 | 0.040 | 0.039 |
| VortonSim | CreateInfluenceTree | AggregateClusters | | 0.037 | 0.038 | 0.038 | 0.039 |
| VortonSim | CreateInfluenceTree | | | 3.787 | 1.838 | 1.405 | 1.153 |
| VortonSim | ComputeVelocityGrid | | | 3.362 | 1.594 | 1.028 | 0.783 |
| VortonSim | StretchAndTiltVortons | | | 0.256 | 0.273 | 0.283 | 0.282 |
| VortonSim | DiffuseVorticityPSE | | | 0.223 | 0.227 | 0.227 | 0.230 |
| VortonSim | AdvectVortons | | | 0.038 | 0.038 | 0.038 | 0.038 |
| VortonSim | AdvectTracers | | | 12.739 | 12.186 | 6.625 | 3.821 |
| FluidBodySim | VortonSim_Update | | | 20.853 | 16.174 | 9.627 | 6.326 |
| FluidBodySim | SolveBoundaryConditions | | | 3.827 | 2.876 | 2.928 | 2.914 |
| FluidBodySim | RbSphere_UpdateSystem | | | 0.002 | 0.002 | 0.002 | 0.002 |
| InteSiVis | FluidBodySim_Update | | | 24.693 | 19.063 | 12.569 | 9.253 |
| InteSiVis | Render_SetMaterial | | | 0.011 | 0.012 | 0.011 | 0.011 |
| ParticlesRender | FillVertexBuffer | | | 11.441 | 6.149 | 3.008 | 2.087 |
| ParticlesRender | Draw | | | 0.033 | 0.028 | 0.024 | 0.021 |
| ParticlesRender | | | | 11.506 | 6.209 | 3.065 | 2.141 |
| InteSiVis | Render_SwapBuffers | | | 0.282 | 0.318 | 0.289 | 0.312 |
| InteSiVis | Render | | | 11.889 | 6.633 | 3.458 | 2.558 |
| InteSiVis | | | | 36.597 | 25.705 | 16.037 | 11.823 |

The overall speedup varies from 1.5× on a single CPU to 2.5× on eight CPUs.

## Summary

Tuning the performance of any program involves timing measurements that must be precise in both time and space. You started by composing a repeatable, easy-to-run benchmark. You manually instrumented this benchmark with timing code to measure the duration of various chunks of code. That provided a simple value with a simple meaning, and although it is precise in *time,* it was not very precise in space—that is, it did not tell you precisely which instructions ran slowly. Intel® VTune, a sampling profiler, narrowed down the search for hotspots by reporting precisely which instructions of which routines occupied most of the time. Intel® VTune also provided a wealth of other information, such as cache misses and branch mispredictions, to alert you of where the code runs less efficiently than it theoretically could. This led to many inspirations about how to make the code more efficient. In addition, you used the `parallel_reduce` feature of Intel® TBB to parallelize a routine that requires some amount of synchronization between threads. These optimizations led to an overall speedup of 1.5× to 2.5×, depending on the number of CPUs.

Future articles will cover more drastic changes to the algorithm to compute velocity along with other enhancements to make this fluid simulation more practical for use in video games.

## Further Reading

- Devir and Zohar (2008): Optimized Matrix Library for use with the Intel® Pentium® 4 Processor's Streaming SIMD Extensions (SSE2)
- Reinders (2007): Intel Threading Building Blocks. O'Reilly.
- Gerber, Bik, Smith & Tian (2006): *The Software Optimization Cookbook, 2nd edition*. Intel Press.
- David Levinthal: Cycle Accounting Analysis on Intel® Core™2 Processors. Intel Corp.