A Performance Optimization Study for the DreamWorks Animation Fluid Solver

John J. O’Neill, Charles Congdon, Ram Ramanujam
Intel Corporation

Silviu Borac, Ron Henderson
DreamWorks Animation

July 2010

Abstract

We present a performance optimization study for an incompressible fluid simulation code designed for animated special effects. The solver uses a marker-and-cell (MAC) method to simulate the motion of liquids and gases. The main challenges in the optimization were identifying efficient data structures for managing the particle data, and rewriting the iterative pressure projection step so as to introduce threading without compromising the convergence rate. On average we achieved an improvement of over 300% in the performance of the serial implementation and parallel efficiency of 65% on an 8-core desktop by threading portions of the application.

Keywords: fluid simulation, threading, performance optimization, parallelism

1 Introduction

Fluid simulation is a widely used technique in special effects to create smoke, fire, splashes and similar visual elements. Figure 1 shows a typical example from DreamWorks’ Animation’s Megamind where a liquid surface is animated using a fluid simulation. While there are a variety of methods used to simulate fluid motion, most of them involve algorithms that are computationally intensive and run sequentially over a number of frames to produce an animated result. For this reason the performance of a given algorithm, in addition to its visual characteristics, is a critical factor in how useful the algorithm may be as an artistic tool.

In this report we discuss a performance optimization study for a fluid simulation package developed at DreamWorks’ Animation and based on techniques now commonly used in computer graphics [1, 2, 3]. Our report includes a description of the algorithm, an analysis of
Figure 1: In DreamWorks’ Animation’s *Megamind* we see the character Roxanne Ritchie suspended over a tank of alligators. The dynamic fluid surface was animated using the solver discussed in this report.

the serial performance, and a discussion of modifications required to achieve good scalability for threading.

## 2 Numerical Method

In order to simulate fluid motion we start with the incompressible Navier-Stokes equations for the conservation of mass and momentum:

\[
\nabla \cdot \vec{u} = 0 \tag{1}
\]

\[
\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \vec{u} + \vec{f} \tag{2}
\]

where \(\vec{u}(x, t)\) is the velocity, \(\rho\) is the fluid density, \(p\) is the pressure, \(\vec{f}(x, t)\) is an applied force, and \(\nu\) is the kinematic viscosity. Without loss of generality we take \(\rho \equiv 1\) to simplify the notation for the remainder of this description.

The above system of equations is integrated forward in time using a splitting scheme [4]. We first integrate the momentum equation followed by a separate step to enforce the conservation of mass. The momentum equation is handled explicitly using forward Euler time integration for the advection and diffusion terms:

\[
\tilde{u} = \vec{u}^n + \Delta t \left[-(\vec{u}^n \cdot \nabla)u^n + \nu \nabla^2 \vec{u}^n + \vec{f}^n\right]
\]  

where \(\vec{u}^n\) is the approximation to the velocity at time level \(t^n = n\Delta t\). This gives an intermediate velocity field \(\tilde{u}\) that no longer satisfies (1). We enforce continuity by then solving
the following equation for the pressure:

\[ \nabla^2 p = \frac{1}{\Delta t} \nabla \cdot \tilde{u} \]  (4)

Subtracting the pressure gradient from the intermediate velocity \( \tilde{u} \) produces a final velocity \( \tilde{u}^{n+1} \) that is an approximate solution to the full Navier-Stokes equations at the next time level:

\[ \tilde{u}^{n+1} = \tilde{u} - \Delta t \nabla p \]  (5)

Eqns. (3-5) constitute the time-integration method used in the solver.

We also need to describe how the fluid is discretized in space. Here we create a uniform grid of size \( h \) that covers the simulation domain of interest. Using the standard arrangement for a marker-and-cell (MAC) grid [5], velocities are stored on the cell faces while pressure and external forces are stored at the cell centers. This is sufficient to simulate the motion of a continuous fluid such as smoke, but for liquids (which might occupy only a fraction of the simulation domain) we need some additional information. For tracking we use a set of particles to identify the volume occupied by the liquid. Particles are advected with the flow using:

\[ \tilde{x}_p^{n+1} = \tilde{x}_p^n + \Delta t \tilde{u}^{n}(\tilde{x}_p^n) \]  (6)

which allows us to follow the motion of the fluid over time. Note that particle positions, more than the fluid velocity or pressure, are the primary output of a liquid simulation.

Since we also need a way to represent discretized objects such as characters and environments in the simulation, we use the general notion of a cell type to identify the problem configuration at any given time. Figure 2 shows a schematic of how cells are classified within the computational domain. Cells are empty (filled with nothing), occupied (filled with a solid object), or filled with fluid. We add one further classification for fluid cells to keep track of those on the surface, or liquid-air interface, in order to apply appropriate boundary conditions. This classification is dynamic and updated prior to each time step in the simulation.

This approach for fluid simulation is flexible enough to model a variety of behaviors. Although really designed as way to simulate liquids, the solver can be configured to operate like a particle simulation with fluid forces, and it can handle simulations of smoke by treating all cells not occupied by solid objects as type fluid. Algorithm 1 summarizes the overall simulation algorithm.

### 2.1 Performance Challenges

What might not be obvious from the listing in Algorithm 1 is which steps are most critical for performance. While the exact answer is dependent on the problem being simulated, we can state some general characteristics. Steps 1 and 3 involve updates to only a subset of the full problem and have little impact on performance. Steps 2 and 5–7 involve updates to every value stored in full three-dimensional arrays and therefore require efficient memory layout and array access. Also, Step 2 requires an efficient mapping between particle positions and
Figure 2: Steps in updating cell types: (a) initial configuration of particles and cells occupied by objects (green); (b) empty cells containing particles are tentatively marked as fluid (blue); (c) fluid cells adjacent to empty cells are marked as surface cells; and (d) complete cell type configuration.

grid cells. Step 4 represents a significant amount of work because the number of particles is typically large compared to the number of grid cells, typically ten or more particles per grid cell. Finally, Step 6 requires the solution of a linear system of equations that couples the pressure and velocity fields and is the one step that does not easily scale linearly with the problem size. So in summary, the important issues for performance are: an efficient three-dimensional array implementation, an efficient data structure for storing particle positions, and an efficient method for implementing the pressure projection step.

The technique used in the original implementation of the pressure projection step poses a particular challenge for threading. Note that Eq. (4) is formally a Poisson equation for the pressure and there are several classical approaches for solving it numerically. The technique used in the current solver is similar to Gauss-Sidel with Successive Over-Relaxation, or simply SOR [7]. An outline of the method is given in Algorithm 2. Here \( p(i, j, k) \) is the current approximation for the pressure, \( r(i, j, k) \) is the residual in solving Eq. (4), and \( \beta \in [1, 2] \) is
Algorithm 1 Fluid simulation algorithm

for $n = 1$ to $N$ do
  1. Inject new particles, $\vec{x}_p^n$
  2. Update cell types
  3. Update boundary conditions for surface cells and obstacles
  4. Advect particles, $\vec{x}_p^{n+1}$
  5. Compute intermediate velocity, $\vec{u}$
  6. Compute pressure for projection step, $p$
  7. Compute final velocity, $\vec{u}^{n+1}$
end for

the over-relaxation parameter. The algorithm chooses a local correction to $p(i, j, k)$ based on the local value of the residual, and then distributes the effect of that change to the neighboring cells. If the update step in classic SOR is viewed as a gather operation that takes the latest approximation from neighboring grid cells and updates the local solution, then Algorithm 2 is a scatter operation that updates the local solution and propagates the resulting change in the residual. In practice we have found this to be an effective algorithm for solving the pressure projection step that performs considerably better than classic SOR or related methods such as conjugate gradient iteration for the grid resolutions typical of simulations used in production [1, 7]. The difficulty of threading Algorithm 2 comes from distributing changes to $r(i, j, k)$ in overlapping cells.

In the remainder of this paper we discuss the specific techniques used to analyze the performance of the original implementation, make better choices for the data structures used to store three-dimensional arrays and particle positions, and introduce concurrency into the pressure projection step while retaining the advantages of the original algorithm.

3 Optimization

The optimization process is simple in principle and complex in practice. Before any optimization can start, we need representative and reproducible workloads demonstrating real use of the application in a wide variety of production cases, not just corner cases or isolated performance bugs.

With a good set of workloads in hand, we must first analyze the application performance in the context of overall system use: conflicts with other processes, how often the application performs I/O, its memory footprint, etc. Once all issues are understood at the system level, and any performance issues that turn up are fixed, the next step is to analyze the performance of the application at the function and algorithm level. Many times this turns up surprises, such as functions that were considered of minor importance consuming a high percentage of application runtime. Finally, once the algorithms and calling patterns have been optimized, for best performance it makes sense to look at the application from the architectural level. This can help frame questions such as how effectively the application
Algorithm 2 Pressure solver

\[
p := 0 \\
r := \frac{h^2}{\Delta t} \ast \text{divergence}(\tilde{u})
\]
while max(r) > tol do
  for i,j,k in grid do
    if cell(i,j,k) is fluid and r(i,j,k) > tol then
      q = \beta \ast \frac{r(i,j,k)}{6}.
      p(i,j,k) += q
      r(i,j,k) -= q \ast 6.
      r(i+1,j,k) += q
      r(i-1,j,k) += q
      r(i,j+1,k) += q
      r(i,j-1,k) += q
      r(i,j,k+1) += q
      r(i,j,k-1) += q
    end if
  end for
end while

makes use of the processor caches, whether parts of the application can make use of SIMD-
style data parallelism, whether threading results in excessive data movement between caches
or memory subsystems, and so forth.

At each level the optimization process is iterative, starting with measurement, then iden-
tifying possible performance issues, trying to fix any issues, testing the correctness of the
fixed code, measuring performance and looking for new issues in the updated application,
and so on until you are content with application behavior. More detailed information on the
optimization process can be found in white papers available online [9, 10].

3.1 Representative and Reproducible Workloads

In keeping with the above methodology, four production workloads were isolated from the
production environment and simplified into individual standalone programs. We were able
to repeatedly run these simulations with consistent timing results. The four test cases were:

- slosh - fluid moving around in a sealed box
- splash - fluid splashing when an object was dropped into it
- insert - fluid particles rapidly added to the simulation for advection
- explosion - a case where the solver ran in gas mode (no liquid tracking)
These test cases are representative of how the simulator is used in production and cover the spectrum of performance issues from particle-dominated simulations to grid-dominated simulations. We used the simulation time per frame (including all I/O required for generating output files) as the high-level metric for performance. Table 1 shows the grid resolution, particle count and simulation times for our various workloads, and Figure 3 show a representative render of the simulation output.
<table>
<thead>
<tr>
<th>Simulation</th>
<th>Grid Resolution</th>
<th>Particles</th>
<th>Original (s)</th>
<th>Optimized (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>slosh</td>
<td>20x20x20</td>
<td>20,412</td>
<td>0.19</td>
<td>0.06</td>
</tr>
<tr>
<td>splash</td>
<td>124x94x124</td>
<td>5,600</td>
<td>7.15</td>
<td>1.15</td>
</tr>
<tr>
<td>insert</td>
<td>50x75x50</td>
<td>14,778</td>
<td>3.61</td>
<td>0.73</td>
</tr>
<tr>
<td>explosion</td>
<td>75x75x75</td>
<td>4,000</td>
<td>12.13</td>
<td>2.14</td>
</tr>
</tbody>
</table>

Table 1: Grid resolution and particle count for the simulations used as performance benchmarks. Simulation time is provided as geometric mean time-per-frame since the number of time steps in a given frame is adjusted dynamically. Times do not include I/O for writing output files. Run times were collected on a workstation with dual 3.2 GHz Intel® Xeon® Processors X5482 and 8GB of memory.

Figure 3: One frame from each of the test cases.
Table 2: Intel® VTune™ Performance Analyzer data for the slosh simulation, showing how much of the runtime is spent in each module linked into the fluid simulator.

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU Samples</th>
<th>CPU samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>flu_slosh</td>
<td>27,389</td>
<td>97.4%</td>
</tr>
<tr>
<td>libc-2.3.4.so</td>
<td>418</td>
<td>1.5%</td>
</tr>
<tr>
<td>vmlinux-2.6.9.0.1.ELsmp</td>
<td>154</td>
<td>0.6%</td>
</tr>
</tbody>
</table>

Figure 4: Timeline of the slosh simulation showing which modules (shared libraries) are used during each time interval, and how much CPU time was accumulated in each module during a given time interval: red means a lot of time in that module, green means only a little time in the module, and white periods indicate no time was spent in that module. The height of the lines also gives a rough idea of the relative intensity of the work done in a given module during each time interval.

3.2 System-level analysis

Before doing low-level optimization, it is always useful to understand how the application executes in the context of the overall operating system, and how much time is spent in application code versus utility code. Tools for doing this analysis on Linux* include top, mpstat, iostat, and Intel® VTune™ Performance Analyzer.

As an example, Table 2 shows the amount of runtime spent in the slosh simulation versus the amount of time spent in code like the C-runtime library. As you can see, this application spends over 97% of its runtime in the simulation code. This suggests that the application is already making good use of system resources, so that we should spend most of our time investigating optimization at the application level. Had we seen large amounts of time in library functions like file I/O, memory allocation, or math library calls, then our approach would have been to optimize the system utilization of the solver before worrying about particular algorithms.

Another useful way to understand application performance at a system level is to visualize how it runs over time with Intel® VTune™ Performance Analyzer as show in Figure 4. There are a couple of observations to make from the timeline graph of the slosh simulation. First, it can be seen that execution runtime is by no means constant. There are periods where overall runtime in this process dips by half during the time interval, and there are some
periods where the application is essentially stalled for significant periods of time (at around $t = 15.5$ seconds and $t = 29$ seconds in the timeline). In this example, the stalled periods account for almost 10% of total runtime, which can be significant when accumulated over many simulation runs. This may require further investigation, but waiting on I/O operations (writing the completed simulation for a given animation frame) is a likely candidate.

This graph also shows that we are using the C-runtime library (libc) rather consistently, as opposed to in bursts. The implication is unclear at this level of analysis, but it can be a useful piece of the puzzle when trying to understand memory allocation or I/O patterns in an application.

Based on a similar examination of the other workloads, we were reasonably convinced that the serial application was running efficiently, and that runtime characteristics were not being overly effected by system-level activity. It was now time to look at the behavior of the application itself at the method/function level.

### 3.3 Module-level analysis

When we first looked at the module-level performance data returned by Intel® VTune™ Performance Analyzer, we turned up a surprising result: much of the total runtime was being consumed by three-dimensional array access. For example, Table 3(a) shows data from the slosh simulation indicating more than 40% of the runtime is associated with array access operations, implemented as `operator[]` on a class that wraps a contiguous block of memory. Digging further, we discovered that this implementation (from an external library) was not inlining properly even in optimized builds, so an operation that should have been a simple offset memory lookup turned into an expensive function call.

Fortunately, this problem was easily solved by writing a new three-dimensional array class with a proper inline implementation of `operator[]`, and this was easily swapped in for the previous implementation. The result is that we saw runtimes decrease by as much as 300% on some cases. Table 3(b) shows the resulting profile, which is more like what one would intuitively expect given the description in Algorithm 1.

Another problem revealed by module-level analysis was the large number of cache misses originating from the code that moves particles between cells, as shown in Table 4. This was particularly obvious for cases dominated by particle processing, and tied to the data structure used for storing particle positions. To explain why, we need to provide further details on how particle data is processed in the application.

A particle is represented as a data structure with several attributes: position, velocity, age, grid coordinates, etc., for a total of 136 bytes per particle. Particles are divided into sets, one set for each grid cell. These sets were implemented using `std::list` to store a linked list of particle positions, which adds another 16 bytes per particle.

Using `std::list` allows particles to be inserted or deleted efficiently as they move from cell to cell during the simulation. However, iterating over the elements of a list can mean moving a large distance in the application address space, and as the per-cell particle lists are updated many times during a simulation, the memory used to store particle data can
Table 3: Intel® VTune™ Performance Analyzer data for the slosh simulation showing a large amount of runtime spent in a number of different array access function calls. Adopting a more efficient three-dimensional array implementation eliminated this unexpected runtime overhead.

become extremely fragmented. Using std::list makes the updates efficient but destroys the locality of the particle data.

Our solution was to implement a new container that still has constant time insertion and deletion, but stores data in contiguous chunks of memory. It was implemented as a std::list of chunks, and each chunk was a plain array of data values. This leads to improved cache performance since more list elements are contiguous in memory, and reduced memory overhead because fewer pointers need to be stored for each element of the list. In fact the pointer overhead is reduced from 11.8% to only 0.3% per particle.

There are at least two other standard container types that could also be considered, but that we found problematic: std::vector and std::deque. Unlike std::vector our chunked list does not require contiguous memory for the entire data set, and avoids the potentially large memory overhead of growing a std::vector dynamically. In fact we measured an approximately 4-fold increase in memory usage when testing std::vector to store particle data because of unused memory allocated during list updates. Compared to std::deque one has the freedom to choose the block size, whereas the implementation of std::deque that we tested had a hard-coded block size of 512 bytes [6]. For our application this size translated to only 3 particles and is too small to significantly improve the cache performance over the original std::list. In practice we found that a block size of 32 particles (a bit more than 4K of data per block) worked well and provided as much as a 2x improvement in
Table 4: Intel® VTune™ Performance Analyzer data for the insert simulation: (top) using std::list to store a list of particles for each cell; (bottom) using our chunked list. High values for clocks-per-instruction (CPI) indicate that the processor is stalled waiting for data, as does the number of times a performance sample is taken while the processor is waiting because of an L2 cache line miss.

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU Samples</th>
<th>CPI</th>
<th>L2 line miss samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>update_particles</td>
<td>38390</td>
<td>3.68</td>
<td>6224</td>
</tr>
<tr>
<td>eval_viscosity</td>
<td>28517</td>
<td>45.34</td>
<td>4458</td>
</tr>
<tr>
<td>eval_velocity</td>
<td>6800</td>
<td>0.62</td>
<td>15</td>
</tr>
<tr>
<td>update_particles</td>
<td>13252</td>
<td>1.53</td>
<td>464</td>
</tr>
<tr>
<td>eval_viscosity</td>
<td>6451</td>
<td>5.62</td>
<td>1155</td>
</tr>
<tr>
<td>eval_velocity</td>
<td>5255</td>
<td>0.54</td>
<td>9</td>
</tr>
</tbody>
</table>

simulation time for cases where performance was dominated by particle updates.

The optimizations described above improved the performance of the serial implementation tremendously, and further performance analysis indicated that the application was now running efficiently on a single core. For further improvements we next turned to threading.

### 3.4 Adding Parallelism

We used OpenMP* to introduce parallelism in parts of the application that could benefit from parallel processing. Generally the decisions about what parts of an application to implement in parallel are straightforward. Functions that are called a few times and do a lot of work are good cases for task or thread-based parallelism, because the overhead of managing the threads or tasks is minimized by the comparatively large amount of useful work done. Functions called many times that do little work are not good candidates for threading. In these cases, it is better to try to introduce parallelism higher up in the call graph to allow each thread to do more work.

As indicated above, one obvious candidate for threading was the pressure projection step because the function is called exactly once per time step and consumes a significant portion of the overall runtime (as much as 40% or more based on our test cases). However, introducing threading in this algorithm is not completely straightforward because each iteration of the update loop in Algorithm 2 writes results into neighboring cells. While locking is a valid workaround to concurrency issues, it has the unfortunate side effect of limiting scaling on larger numbers of cores, where lock contention eventually dominates runtime.

Our solution was to use a variant of red-black decomposition [11]. In the iterative pressure solver each iteration of the update loop references at most one cell in each direction away from the cell currently being updated. By organizing the cells into distinct slices the entire grid can be updated in three passes without any interference from other threads. We refer to this as red-green-blue decomposition. The modified pressure iteration is shown in Algorithm 3.
Algorithm 3 Pressure solver (threaded)

\[
p := 0 \\
r := (h^2/\Delta t) \times \text{divergence}(\tilde{u})
\]

while max(r) > tol do
  for i in red cells do
    update_residual(i, p, r, cell)
  end for
  for i in green cells do
    update_residual(i, p, r, cell)
  end for
  for i in blue cells do
    update_residual(i, p, r, cell)
  end for
end while

Figure 5: Grid cells are partitioned into red-green-blue slices that can be updated concurrently. Each thread updates a contiguous slice, e.g. the blue box above. A complete pressure iteration requires three passes, but during a single pass all slices of a given color can be updated concurrently.
and the decomposition of grid cells is shown in Figure 5. This traversal pattern allows all of the red slices to be updated concurrently, followed by the green slices, followed by the blue slices. An update of the entire grid therefore requires three passes.

One unavoidable consequence of this type of iterative technique is that the final answer will depend on the order of the iterations. While the difference for a single iteration might be at the level of numerical roundoff error, those differences can accumulate over many iterations and cause the simulation to produce visibly different results after many time steps. Fortunately, we found that using a smaller convergence tolerance eliminated the dependency on iteration order and had minimal impact on overall runtime.

The actual speed-up depends on the amount of work that can be performed in parallel. In the test cases we looked at, the explosion benchmark saw the largest benefit because it was a larger problem size and the pressure was updated in every grid cell; overall we saw a speedup of 4x on 8 cores for this test. Not surprisingly, the other workloads show less benefit, but obviously in proportion to the amount of runtime consumed by the pressure solver.

Parallel scaling is limited by the amount of work done in parallel versus the overhead of thread management and synchronization. Despite the large number of cells in the explosion simulation, the work done by each thread on its portion of the grid is insufficient to completely hide threading overhead. As a result we end up with many short bursts of parallel work until the simulation converges. Viewing this over time, the thread activity looks something like
Table 5: Optimization results for the four workloads studied, showing the breakdown of performance improvements due to serial optimization and threading: $T$ is the original serial time, $T'$ is the optimized serial time, $T_8'$ is the optimized parallel time on 8 cores. The last two columns provide the overall speedup. All times are reported in seconds and reflect the total runtime of the application after I/O has been subtracted. Runtimes collected on workstation with dual 3.2 GHz Intel® Xeon® Processors X5482 and 8GB of memory.

<table>
<thead>
<tr>
<th>Workload</th>
<th>$T$ (s)</th>
<th>$T'$ (s)</th>
<th>$T_8'$ (s)</th>
<th>$T/T'$</th>
<th>$T/T_8'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slosh</td>
<td>32</td>
<td>10</td>
<td>11</td>
<td>3.1</td>
<td>3.0</td>
</tr>
<tr>
<td>Splash</td>
<td>246</td>
<td>71</td>
<td>39</td>
<td>3.5</td>
<td>6.3</td>
</tr>
<tr>
<td>Insert</td>
<td>201</td>
<td>46</td>
<td>41</td>
<td>4.4</td>
<td>4.9</td>
</tr>
<tr>
<td>Explosion</td>
<td>1847</td>
<td>677</td>
<td>165</td>
<td>2.7</td>
<td>11.2</td>
</tr>
</tbody>
</table>

Figure 6 when the pressure solver is iterating to convergence. Zooming in further on the timeline, we can see that each burst of parallel activity shows a marked delay at the start of each component thread due to OpenMP* runtime overhead, and a further delay as threads complete and synchronize. Notice that while there is a degree of parallel work done, there are also long periods when fewer than four threads are running at once. The impact of this overhead will decrease with larger problem sizes.

3.5 Results

As can be seen in Table 5, combining all the optimizations considerably improved the performance of the overall simulation times.¹ Serial optimizations, in particular changing the three-dimensional array implementation and adopting a different data structure for storing particle data, improved the performance by over 300% on average. For the three largest workloads, adding parallelism to the pressure solver further improved performance, and tests on additional workloads showed similar overall improvements.

4 Conclusions

Performance optimization should be a data-driven activity. As we saw with several of the solutions outlined here, a developer’s intuition about how a particular implementation should perform can be vastly different from how it actually performs when measured on real hardware with real production workloads. Fortunately there are good tools for collecting and

---

¹Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, go to [http://www.intel.com/performance/resources/benchmark_limitations.htm](http://www.intel.com/performance/resources/benchmark_limitations.htm)
analyzing performance data that make the job of improving application performance straightforward.

In this study we focused on a particular algorithm, and a particular implementation of that algorithm. As computer architectures evolve it is likely that methods that can effectively exploit a high degree of parallelism will prove to be more efficient than the best serial algorithms. In the future we may look at alternative techniques for the pressure solver, or the entire fluid simulation algorithm, to find alternatives that provide the same visual qualities yet achieve better scaling on multi-core hardware.

References


Optimization Notice

Intel compilers, associated libraries and associated development tools may include or utilize options that optimize for instruction sets that are available in both Intel and non-Intel microprocessors (for example SIMD instruction sets), but do not optimize equally for non-Intel microprocessors. In addition, certain compiler options for Intel compilers, including some that are not specific to Intel micro-architecture, are reserved for Intel microprocessors. For a detailed description of Intel compiler options, including the instruction sets and specific microprocessors they implicate, please refer to the “Intel Compiler User and Reference Guides” under “Compiler Options.” Many library routines that are part of Intel compiler products are more highly optimized for Intel microprocessors than for other microprocessors. While the compilers and libraries in Intel compiler products offer optimizations for both Intel and Intel-compatible microprocessors, depending on the options you select, your code and other factors, you likely will get extra performance on Intel microprocessors.

Intel compilers, associated libraries and associated development tools may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include Intel Streaming SIMD Extensions 2 (Intel SSE2), Intel Streaming SIMD Extensions 3 (Intel SSE3), and Supplemental Streaming SIMD Extensions 3 (Intel SSSE3) instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors.

While Intel believes our compilers and libraries are excellent choices to assist in obtaining the best performance on Intel and non-Intel microprocessors, Intel recommends that you evaluate other compilers and libraries to determine which best meet your requirements. We hope to win your business by striving to offer the best performance of any compiler or library; please let us know if you find we do not.

Performance Disclaimer

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. Configurations: Tests conducted at DreamWorks by Intel using a workstation with dual 3.2 GHz Intel® Xeon® Processors X5482 and 8GB of memory.

For more information go to http://www.intel.com/performance