Hardware-Aware Analysis and Optimization of *Stable Fluids*

Theodore Kim
IBM TJ Watson Research Center

Abstract

We perform a detailed flop and bandwidth analysis of Jos Stam's *Stable Fluids* algorithm on the CPU, GPU, and Cell. In all three cases, we find that the algorithm is bandwidth bound, with the cores sitting idle up to 96% of the time. Knowing this, we propose two modifications to accelerate the algorithm. First, a Mehrstellen discretization for the pressure solver which reduces the running time of the solver by a third. Second, a static caching scheme that eliminates roughly 99% of the random lookups in the advection stage. We observe a 2x speedup in the advection stage using this scheme. Both modifications apply equally well to all three architectures.

1 Introduction

Jos Stam's *Stable Fluids* fluid simulation algorithm [Stam 1999] has enjoyed widespread popularity in the computer graphics community since its introduction. Many extensions have been proposed, most of which utilize some form of the core computational kernels of the original Stam algorithm. One of the main advantages of the algorithm is its speed, so interactive, real-time implementations are available for both the CPU and GPU ([Stam 2003][Harris 2004][Crane et al. 2007]). All three of these articles are primarily intended as educational guides, and forgo an analysis of the computational workload and bandwidth requirements of the algorithm.

However, if *Stable Fluids* is to be used in a larger interactive application, the computational and bandwidth requirements should be known to as granular a resolution as possible, so that resources can be budgeted accordingly. In this paper, we perform such an analysis and find that, despite its numerical nature, the algorithm has a fairly low arithmetic intensity (Eqns. 2 and 3) and is therefore bandwidth bound (Section 3). Knowing this, a natural question to ask is what computations can be performed while waiting for additional data to arrive from memory. We examine higher order *Mehrstellen* methods [Collatz 1960] as a possible use of this otherwise idle processor time (Section 4) and find that it can be used to reduce the running time of the linear solver by 17% in 2D and 33% in 3D.

Most of the stages of *Stable Fluids* exhibit highly regular memory access patterns. The exception is the semi-Lagrangian advection step, which in the worst case can reduce to a large number of small random accesses that are not ideal for SIMD hardware. However, intuition derived from the underlying fluid physics suggests that in the average case, these accesses should still be highly structured. We design a simple static caching scheme that exploits this intuition, and observe a 99% cache hit rate (Section 5).

2 Related Works

Many extensions have been proposed for *Stable Fluids* since its introduction, perhaps the most significant of which is the addition of free surfaces [Foster and Fedkiw 2001; Enright et al. 2002]. Since the purpose of this paper is optimization and not extension, we refer the interested reader to the previous works section of other recent works for a more complete overview of these techniques [Kim et al. 2007; Klingner et al. 2006].

The viability of various solvers has been demonstrated on parallel hardware, particularly the GPU. These solvers include Multigrid [Goodnight et al. 2003], conjugate gradient [Bolz et al. 2003] and FFT [Moreland and Angel]. Several GPU-based fluid solvers have also been proposed, including Lattice Boltzmann computation [Li et al. 2003], and Smoothed Particle Hydrodynamics [Harada et al. 2007]. These were all "proof-of-concept" implementations however, so they forgo the type of detailed flop and bandwidth analysis we undertake in favor of discussing higher-level parallelization strategies.

For brevity, we will forgo summarizing the *Stable Fluids* algorithm here, and instead refer the uninitiated reader to Stam's excellent entry-level overview [Stam 2003].

3 Flop and Bandwidth Analysis

In this section, we will perform a flop and bandwidth analysis of *Stable Fluids* so that we can estimate an upper bound on the performance we can expect from various hardware. We will track bandwidth utilization as optimistically as possible by assuming an ideal cache where the maximum data reuse is achieved. In practice, this means that three rows of the computational grid fit in cache, so this is not an unrealistically optimistic assumption.

In order to simplify the analysis, we will only track quantities to the leading term. The computational grid is of size N^2, so operations that are $O(N)$ such as boundary clamping will be ignored. Since the Geforce 8, Cell, and Intel SSE4 all support a combined multiply-add instruction, we will count multiply-adds as a single flop. This will simplify peak performance calculation later since we can then divide our totals against chip clock speeds, and do not have to rely on the somewhat more nebulous peak flop rates reported by vendors. We will initially track bandwidth on a per-float basis, so unless otherwise noted, N^2 corresponds to N^2 floats, not bytes. We will specifically be analyzing Stam's open source implementation [Stam 2003], and paraphrasing that code where appropriate. Both the GPU code [Harris 2004] and our Cell implementation are based directly on this code.

3.1 Add Source

The first stage of the algorithm is an `add_source` function that adds user-input forces to the velocity and density fields. The loop takes the form:

```
for (int j = 0; i < rows; j++)
  for (int i = 0; i < columns; i++)
    x[i][j] += s * x0[i][j];
```

The code performs a multiply-add over the entire N^2 grid of velocity and the density fields. There is no opportunity for data reuse here, so for every multiply-add, there are two loads (`x[i][j]` and `x0[i][j]`) and a store (the new value of `x[i][j]`). Taking into account the 2 scalar components of the velocity and the single density field, the we obtain that following sums:

- Flops: $3N^2$
- Bandwidth: $9N^2$ floats

3.2 Diffusion

Diffusion also occurs over the entire grid for the velocity and density fields. Stam codes up the inner loop of the diffusion stage as:

```
x[i][j] = x0[i][j] +
          d * (x[i-1][j] + x[i+1][j]+
               x[i][j-1] + x[i][j+1]));
```

This operation is performed on every grid cell for I number of iterations, and occurs for both of the velocity fields and the density field. Therefore, after we calculate the flop and bandwidth usage for the inner loop, we should multiply it by $3IN^2$. There is one store to `x[i][j]`, and at least one load, `x0[i][j]`. There are seemingly four other loads from `x` on the right hand side, but these are the immediate grid neighbors of `x[i][j]`. In the ideal cache case, most of these elements would already have been loaded into cache from when the `x[i][j-1]` row was computed, except for the `x[i][j+1]` element. So in the ideal cache case, only this single element needs to be loaded.

The three adds on the right hand side of the four `x` elements are unavoidable, so they add three flops. The multiplication by `d` and the addition to `x0[i][j]` can be folded into a multiply-add, giving a total of 4 flops. After multiplying everything by $3IN^2$, we obtain the new totals:

- Flops: $(3 + 12I)N^2$
- Bandwidth: $(9 + 9I)N^2$ floats

3.3 Projection

The projection stage proceeds in three stages: divergence computation, pressure computation, and the final projection. First the divergence is computed over each grid cell thus:

```
div[i][j] = -c * (u[i+1][j] - u[i-1][j] +
                  v[i][j+1] - v[i][j-1]);
```

The store to `div[i][j]` on the left hand side is unavoidable. However, the loads from `u` and `v` can be amortized in a manner similar to loads from `x` in the diffusion stage. If we assume that the entire current row for `u` is loaded all at once into cache, this translates to one load per element, not two. If we assume that the `j-1` and `j` rows of the `v` array are already in cache from computation over previous rows, we only need to load the `j+1`th row, which again translates to one load per element. Thus, divergence computation takes $3N^2$ floats of bandwidth. The flop count is a straightforward $4N^2$, since nothing can be folded into a multiply-add.

Next, pressure is computed by a linear solver. The function called is exactly the same as the one from diffusion, which we know from the previous subsection consumes $4IN^2$ flops, and $3IN^2$ floats of bandwidth.

Last, the divergence-free component is projected out of the velocity field. The inner loop of this projection is two lines:

```
u[i][j] += -f * (p[i+1][j])] - p[i-1][j])]);
v[i][j] += -f * (p[i][j+1])] - p[i][j-1])]);
```

The loads and stores to `u` and `v` are unavoidable, and the loads from `p` are neighbor accesses that can again be amortized to a single load. Each line consumes 2 flops: one subtraction between `p` elements, and a multiply-add. So, the projection stage consumes $4N^2$ flops and $5N^2$ floats of bandwidth. Pulling all three substages together, we get a total for the projection stage:

- Flops: $(4 + 4I + 4)N^2 = (8 + 4I)N^2$
- Bandwidth: $(3 + 3I + 5)N^2 = (8 + 3I)N^2$ floats

The new total for the entire algorithm so far is now:

- Flops: $(11 + 16I)N^2$
- Bandwidth: $(17 + 12I)N^2$ floats

3.4 Advection

The final stage of the solver is advection. Advection occurs for both the velocity and density fields, so we should again multiply our final count by three. First, backtraces are computed for each grid cell:

```
x = i - dt0 * u[i][j];
y = j - dt0 * v[i][j];
```

While the final values here are stored to `x` and `y`, these are local variables that are not written out to main memory, so they do not require a store. Aside from this there are loads from `u` and `v` with a multiply-add each. So the backtrace consumes N^2 flops and $2N^2$ floats of bandwidth.

Next the grid indices of the backtrace are computed:

```
if (x < 0.5f) x = 0.5f;
if (x > columns + 0.5f) x = columns + 0.5f;
i0 = (int)x; i1 = i0 + 1;

if (y < 0.5f) y = 0.5f;
if (y > rows + 0.5f) y = rows + 0.5f;
j0 = (int)y; j1 = j0 + 1;
```

The `if` statements clamp the indices to the boundaries of the grid, and for simplicity we will ignore these terms. These lines can be stated as ternaries (`x = (x < 0.5f) ? 0.5f : x;`) at any rate, so they would not truly incur branch penalties.

The lines `i0 = (int)x;` and `j0 = (int)y;` use an `(int)` cast to emulate a floor function. For simplicity we will count these as a flop each, though the actual number of cycles burned will be library-dependant. Two additional flops are expended computing `i1` and `j1`. Since these are all occurring on local variables, the total count for this section is $4N^2$ flops and zero bandwidth.

Next, interpolation weights for the lookup at computed:

```
s1 = x - i0; s0 = 1 - s1;
t1 = y - j0; t0 = 1 - t1;
```

Since these are again operations on local variables, we count as $4N^2$ flops and zero bandwidth.

Finally, we arrive at the interpolation:

```
d[i][j] = s0 * (t0 * d0[i0][j0] + t1 * d0[i0][j1]) +
          s1 * (t0 * d0[i1][j0] + t1 * d0[i1][j1]);
```

The store to `d` is unavoidable. The loads from the array `d0` are arranged in a 2x2 neighborhood around the result of the backtrace. Unfortunately, there is no predicting where the backtrace points to, so we cannot amortize this memory access, and incur a cost of 4 loads. This issue will be revisited in section 5.

In its current nested form, there are 9 flops per interpolation. With judicious use of multiply-adds, this can be reduced to 6 flops, though the details of this are left as an exercise to the reader. The total for the advection step is then:

- Flops: $(1 + 4 + 4 + 6)N^2 = 15N^2$
- Bandwidth: $(2 + 5)N^2 = 7N^2$ floats

Advection is applied three times for the velocity and density fields, so when added to the total counts, we obtain the final count for the whole algorithm:

- Flops: $(56 + 16I)N^2$

- Bandwidth: $(38 + 12I)N^2$ floats

This same counting method can be applied to 3D to obtain the following totals:

- Flops: $(106 + 30I)N^3$
- Bandwidth: $(71 + 15I)N^3$ floats

In the 3D case, we make the much more generous assumption that three z slices of the 3D grid fit into cache. However, even with this overly optimistic assumption, the algorithm is still firmly bandwidth-bound.

3.5 Peak Performance Estimates

With final flop and bandwidth counts in hand, we can now obtain rough estimates for the peak performance of *Stable Fluids* on various architectures. Note that these are *peak* estimates, so we are assuming peak memory bandwidth is achieved, and flops are fully pipelined and dispatched at each clock cycle. For the CPU, we use the specifications for a Xeon 5100 ("Woodcrest"). For the GPU, we will use the specifications for a Geforce 8800 Ultra. For the Cell we will use the specifications for an IBM QS20 blade. In both 3rd party CPU and GPU codes, the variable I is fixed to 20, so will set $I = 20$ in all of our calculations as well.

A Intel Xeon 5100 runs two cores at 3 Ghz, and is capable of dispatching a 4-float SIMD instruction each clock cycle. Thus, we characterize the peak Xeon performance is 24 GFlops/s. The published peak memory bandwidth is 10.66 GB/s [Intel 2007].

The Nvidia Geforce 8800 Ultra runs 128 scalar cores at 1.5 Ghz. Though some sources cite the peak flop performance as greater than 500 GFlops/s, this only applies to the special case where a multiply-add and a multiply instruction are dual issued, and the multiply-add is counted as two flops. Most of the multiplies in *Stable Fluids* have been folded into multiply-adds, so such dual issues will be fairly rare for our purposes. Therefore, we characterize the peak performance of the Geforce 8800 Ultra as 192 GFlop/s. The published peak memory bandwidth to graphics memory is 103.7 GB/s [Nvidia 2007a].

The IBM QS20 Cell blade runs two Cell chips at 3.2 Ghz. Each Cell has 8 Synergistic Processing Elements (SPEs) which are each capable of dispatching 4-float SIMD instructions every clock cycle. While the Cell also includes a Power Processing Element (PPE), it is usually not used for heavy computation, so its computational capability is not included in the total. Thus, we characterize the peak performance of the QS20 blade at 204.8 GFlops/s. The published peak memory bandwidth is 25.6 GB/s [IBM 2007].

The peak performance estimates in 2D and 3D are given in Table 1. For each architecture, we computed the peak compute-bound performance by dividing the peak flops per second by the number of flops necessary per timestep of *Stable Fluids* (ie the Flops total from the previous subsection).

Equivalently, we computed the peak bandwidth-bound performance by dividing the peak gigabytes per second by the number of bytes per timestep (ie the Bandwidth total from the previous section times four, to account for four bytes per float). In all cases, the bandwidth-bound number was smaller, leading us to conclude that *Stable Fluids* is bandwidth-bound on all three architectures.

According to our 2D estimates, the CPU computation runs 6.65x faster than data arrives, projecting that the processor is idle more than 85% of the time. On the GPU, computation is 5.47x faster, and the cores are idle 82% of the time. On the Cell, computation is 23.66x faster, and the cores are idle 96% of the time.

According to our 3D estimates, the CPU computation runs 4.74x faster than data arrives, is 79% idle, the GPU runs 3.89x faster than data arrives, is 74% idle, and the Cell runs 16.8x faster than data arrives, and is 94% idle.

These sizable idle times make sense if we look at the arithmetic intensity [Harris 2005] of *Stable Fluids*. The arithmetic intensity is defined as:

$$\text{arithmetic intensity} = \frac{\text{Total ops}}{\text{Total words transferred}}. \quad (1)$$

Using the results of our previous analysis, if we assume the number of iterations I is large, we obtain the intensities:

$$\lim_{I\to\infty} \frac{(56 + 16I)N^2}{(38 + 12I)N^2} = \frac{4}{3} \quad \text{(in 2D)} \quad (2)$$

$$\lim_{I\to\infty} \frac{(106 + 30I)N^2}{(71 + 15I)N^2} = 2 \quad \text{(in 3D).} \quad (3)$$

Algorithms runs well on the Cell and GPU when their arithmetic intensities are much greater than one. As both the 2D and 3D cases are close to one, the available flops will be underutilized.

3.6 Performance Measurements

We measured the frame rate of CPU [Stam 2003], GPU [Harris 2004], and Cell implementations in order to validate our analysis. The results can be seen in Table 2. Note that this is not intended as a benchmark, since the codes are not necessarily optimized. Instead it is intended as an experimental validation of our analysis. For example, if the frame rates obtained exceeded our bandwidth-bound estimates, it would suggest a flaw in our reasoning. However, as expected, our predicted theoretical peaks were never exceeded, providing additional evidence that the algorithm is bandwidth-bound.

There are several points of note in the data. First, the original GPU code works on 16 bit textures, whereas the CPU and Cell implementations use 32 bit floating point. We modified the code to use 32 bit textures, and then collected both 16 and 32 bit timings. Perhaps due to the immaturity of the series 8 drivers, the 16 bit timings on a Geforce 7900 were superior, so those timings are listed here instead.

A trend to note on both the GPU and Cell is that as the resolution is increased, the theoretical peak is more closely approached. This is probably due to the larger coherent loads that can be performed, which makes more effective use of the available bandwidth. This issue will be revisited when designing a caching scheme in Section 5. Finally, we note the 740 Hz plateau observed on the Geforce 7900. This is probably because 16-bit texture copies smaller than 256^2 are automatically rounded up to 256^2, so the bandwidth usage remains the same even at lower resolutions. This limitation appears to have been removed on the Geforce 8.

4 Mehrstellen Schemes

The analysis from the previous section, while highly idealized, provides significant evidence that *Stable Fluids* is a bandwidth-bound algorithm. Knowing this, a natural question to ask is if there are any useful computations that can be added to the algorithm to further occupy the cores while they are waiting for data to arrive from main memory.

In this section, we investigate Mehrstellen schemes [Collatz 1960], also known as compact schemes, as a candidate computation to further occupy the cores. We choose Mehrstellen schemes because there is some evidence [Gupta et al. 1997] that they can be used to

Resolution	Intel Xeon 5100 (Woodcrest)		Nvidia Geforce 8800 Ultra		IBM Cell QS20	
	Compute-bound	Bandwidth-bound	Compute-bound	Bandwidth-bound	Compute-bound	Bandwidth-bound
64^2	15583 Hz	2340 Hz	124670 Hz	22767 Hz	132980 Hz	5620 Hz
128^2	3896 Hz	585 Hz	31167 Hz	5691 Hz	33245 Hz	1405 Hz
256^2	974 Hz	146 Hz	7791 Hz	1423 Hz	8311 Hz	351 Hz
512^2	243 Hz	36 Hz	1948 Hz	356 Hz	2078 Hz	87 Hz
1024^2	61 Hz	9 Hz	487 Hz	89 Hz	519 Hz	21 Hz
2048^2	15 Hz	2 Hz	122 Hz	22 Hz	130 Hz	5 Hz
64^3	130 Hz	27 Hz	1037 Hz	267 Hz	1106 Hz	66 Hz
128^3	16 Hz	3 Hz	130 Hz	33 Hz	138 Hz	8 Hz
256^3	2 Hz	0.4 Hz	16 Hz	4 Hz	17 Hz	1 Hz

Table 1: *Estimated peak frames per second of* Stable Fluids *over different resolutions for several architectures. Peak performance is estimated for each architecture assuming the computation is compute-bound (ie infinite bandwidth is available) and bandwidth-bound (ie infinite flops are available). The lesser of these two quantities is the more realistic estimate. In all cases, the algorithm is bandwidth-bound.*

Resolution	Intel Xeon 5100 (Woodcrest)		Geforce 8800 Ultra (32 bit)		Geforce 7900 (16 bit)		IBM Cell QS20	
	Peak	Measured	Peak	Measured	Peak	Measured	Peak	Measured
64^2	2340 Hz	74 Hz	22767 Hz	484 Hz	45534 Hz	740 Hz	5620 Hz	472 Hz
128^2	585 Hz	26.7 Hz	5691 Hz	212 Hz	11382 Hz	745 Hz	1405 Hz	347 Hz
256^2	146 Hz	6.13 Hz	1423 Hz	65 Hz	2846 Hz	744 Hz	351 Hz	164 Hz
512^2	36 Hz	0.296 Hz	356 Hz	18 Hz	712 Hz	355 Hz	87 Hz	49 Hz
1024^2	9 Hz	0.038 Hz	89 Hz	5 Hz	178 Hz	99 Hz	21 Hz	18 Hz

Table 2: *Theoretical peak frames per second (The bandwidth-bound values from Table 1) and actual measured frames per second. None of the measured times exceed the predicted theoretical peaks, validating the finding that the algorithm is bandwidth bound. A GeForce 7900 was used for the 16 bit timings because the frame rates were uniformly superior to the 8800.*

reduce the number of Jacobi iterations necessary to solve a system of of equations. This would be a quite useful property, as it would mean we could use a smaller value for I and reduce the overall work of the entire algorithm.

4.1 A Fourth Order Discretization

Both the diffusion and projection stages of *Stable Fluids* solve a Poisson equation of the form:

$$\nabla^2 x = b. \quad (4)$$

The Laplace operator ∇^2 is usually discretized to second order accuracy, yielding a system of equations of the following form,

$$x_{i-1,j} + x_{i,j-1} - 4x_{i,j} + x_{i+1,j} + x_{i,j+1} = b_{i,j}, \quad (5)$$

where the (i, j) indices denote coordinates on the computational grid. This same equation is perhaps more easily visualized spatially in its stencil form:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} x = b \quad (6)$$

The discretization could be extended from second to fourth order accuracy by adding more terms from the Taylor expansion in both the i and j directions:

$$\begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 16 & 0 & 0 \\ -1 & 16 & -60 & 16 & -1 \\ 0 & 0 & 16 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix} x = b \quad (7)$$

An additional amount of computation has been introduced, but the spatial extent of the stencil has expanded as well, adding complexity to the memory access pattern. In order to preserve the ideal caching assumption from our analysis, we would have to assume that five lines of the computational grid fit in cache instead of three, ie that the cache is 66% larger. We would like to avoid making this broader assumption.

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix} x = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 4 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix} b$$

Figure 1: *The 2D Mehrstellen discretization.*

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & -24 & 2 \\ 1 & 2 & 1 \end{bmatrix} x = \begin{bmatrix} 0 & \frac{1}{2} & 0 \\ \frac{1}{2} & 3 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 \end{bmatrix} b$$
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Figure 2: *The 3D Mehrstellen discretization.*

The Mehrstellen scheme (roughly translated, "many points") [Collatz 1960] is an alternate discretization that allows us to increase the accuracy from second to fourth order without significantly increasing the complexity of the memory access pattern. Instead of adding more terms to the 1D Taylor expansions, the Mehrstellen scheme subtracts off an additional viscosity term in a manner similar to the Lax-Wendroff scheme [Lax and Wendroff 1960]. The Mehrstellen discretizations for 2D and 3D are shown in Figures 1 and 2. The spatial support of the stencil on the left hand side is the same as that of the second order stencil. While the diagonal neighbors to $x_{i,j}$ have been newly added to the computation, we have already been assuming that rows $j-1$, j, and $j+1$ are in cache, so we should be able to access all of these diagonal values without adding any additional bandwidth complexity.

The right hand side of the equation is now more complex, as it requires us to compute a stencil over the values of b as well. However, note that we only have to do this once during the divergence computation portion of the projection stage. The inner loop of the linear solver, where the bulk of the computation and bandwidth consumption takes place, is unaffected.

4.2 Spectral Radius of the Discretization

In a Jacobi solver, the error of the current solution is multiplied by the spectral radius of the Jacobi matrix every iteration. We can gauge the error reduction rate of the Mehrstellen discretization by measuring the spectral radius of its resultant matrix. If the radius is significantly smaller than that of the second order discretization, then we should need less Jacobi iterations overall.

The spectral radius ρ_M of Jacobi iteration using the Mehrstellen discretization can be obtained with the formula [Demmel 1997],

$$\rho_M = \max\left(1 - \frac{\lambda_M}{20}\right) \quad \text{(in 2D)} \tag{8}$$

$$\rho_M = \max\left(1 - \frac{\lambda_M}{24}\right) \quad \text{(in 3D).} \tag{9}$$

where λ_M is the eigenvalues of the matrices formed by the Mehrstellen stencils in Figures 1 and 2. The equivalent radius for the standard Jacobi matrix (ρ_S) is available analytically

$$\rho_S = \left|\cos\frac{N\pi}{N+1}\right|, \tag{10}$$

where N is the size of one of the grid dimensions. Note that ρ_S is the same in both 2D and 3D.

We computed the spectral radius of several small grid resolutions in 2D and 3D using both Matlab and SLEPc [SLEPc 2007]. While we would have liked to compute the radii of larger grids, the memory requirements of solving the eigenvalue problem (even using a sparse, iterative method) quickly became prohibitive.

We compare the radii of Mehrstellen Jacobi (ρ_M) with the radii of standard Jacobi (ρ_S) in Table 3. The differences in radii may seem negligible, but recall that the error is multiplied by this factor each iteration, so the difference between the two numbers is amplified exponentially with each application. We can compute the number of iterations it would take Mehrstellen Jacobi to achieve an error reduction equivalent to 20 iterations of standard Jacobi by computing

$$\text{Equivalent Mehrstellen iterations} = \frac{\log\left(\rho_S^{20}\right)}{\log\rho_M}. \tag{11}$$

In fact, the number of Mehrstellen iterations necessary is always a constant fraction of the standard iterations:

$$\frac{\text{Equivalent Mehrstellen iterations}}{\text{Standard iterations}} = \frac{\log\rho_S}{\log\rho_M}. \tag{12}$$

As shown in Table 3, 2D Mehrstellen Jacobi consistently only needs 16 iterations to match the error reduction from 20 iterations of 2D standard Jacobi. In general, 2D Mehrstellen Jacobi achieves results equivalent to standard Jacobi using 83% of the iterations. In 3D, only about 13 iterations are needed, and in general only 66% of the iterations are needed.

The additional flops necessary to compute the Mehrstellen discretization should be entirely hidden by memory latency. Since the Jacobi iterations comprise the bulk of the computation of *Stable Fluids*, these reduced iterations translate almost directly to a 17% speedup in 2D and a 33% speedup in 3D.

We prototyped a Mehrstellen version of 3D *Stable Fluids* on the CPU. Note that adding this modification to an unoptimized code will not result in a speedup, since the additional computation will not be hidden by the memory latency. However, after a reasonable amount of tuning (loop unrolling, temporary variables to reduce register dependencies), we observed the expected 33% speedup in the pressure solve. Figure 5 compares the results of standard Jacobi with Mehrstellen Jacobi. The results are virtually identical, showing that Mehrstellen Jacobi does not compromise the visual fidelity of the final result.

Resolution	ρ_M	ρ_S	$\frac{\log(\rho_S)^{20}}{\log\rho_M}$	$\frac{\log\rho_S}{\log\rho_M}$
10^2	0.9517	0.9595	16.70	0.8351
20^2	0.9866	0.9888	16.69	0.8349
30^2	0.9938	0.9949	16.44	0.8221
40^2	0.9965	0.9971	16.56	0.8283
50^2	0.9977	0.9981	16.51	0.8259
60^2	0.9984	0.9987	16.24	0.8124
70^2	0.9988	0.9990	16.66	0.8332
80^2	0.9991	0.9992	16.67	0.8334
10^3	0.9401	0.9595	13.38	0.6690
20^3	0.9833	0.9595	13.38	0.6673

Table 3: *Spectral radii of the fourth order accurate Mehrstellen Jacobi matrix (ρ_M) and the standard second order accurate Jacobi matrix (ρ_S). The third column computes the number of Mehrstellen iterations necessary to match the error reduction of 20 standard iterations. The last column is the fraction of Mehrstellen iterations necessary to match the error reduction of one standard iteration.*

5 Advection Caching

In this section we design a caching scheme that eliminates most of the small, random accesses to main memory exhibited by the advection stage of *Stable Fluids*. Small, incoherent accesses are best avoided on the GPU and Cell because they exhibit much larger memory latencies than large, contiguous accesses [Kistler et al. 2006] [Nvidia 2007b]. On the Cell for example, 1 KB accesses achieve a bandwidth of less than 10 GB/s, which is less than half of the peak bandwidth of 25.6 GB/s [Kistler et al. 2006]. Our own numerical experiments on the Nvidia Geforce 8800 Ultra have shown that similar bandwidth penalties are incurred for small transfer sizes.

5.1 Physical Characteristics

In the general case, designing a caching scheme that works well for an arbitrary vector field is difficult, since a vector field can always be constructed that produces the worst case access pattern. The vector fields produced *Stable Fluids* are far from arbitrary however, because they have the characteristics of incompressible flow. There are additional qualitative reasons to expect that the majority of the vector field exhibits high spatial locality. While *Stable Fluids* supports arbitrarily large timesteps, in practice the timestep size will never exceed $\frac{1}{24}$ seconds, since the speed of film is 24 Hz (theater projectors run at 48 Hz, but show each frame twice). Games and TV run at higher rates, usually at least 30 Hz, which results in even smaller step sizes. Additionally, the projection and diffusion operators smear out the velocity field, so even if large velocities are introduced into the simulation, they quickly dissipate into smaller velocities in both space and time. So, the majority of the vectors in the velocity field should have small magnitudes which are then further reduced by a factor of at least 24 during the advection step.

Resolution	With Cache	Without Cache
64^2	15.91 GB/s	8.01 GB/s
128^2	16.45 GB/s	7.98 GB/s
256^2	16.54 GB/s	7.98 GB/s
512^2	16.5 GB/s	8.18 GB/s
1024^2	15.54 GB/s	8.16 GB/s

Table 4: *Bandwidth achieved by the advection stage on the Cell with and without the static cache.*

Thus, it is reasonable to assume that most of the advection rays terminate in regions that are very close to their origins.

5.2 A Static Caching Scheme

In order to test this hypothesis, we employed a simple static caching scheme. Recall the advection interpolation step:

```
d[i][j] = s0 * (t0 * d0[i0][j0] + t1 * d0[i0][j1]) +
          s1 * (t0 * d0[i1][j0] + t1 * d0[i1][j1]);
```

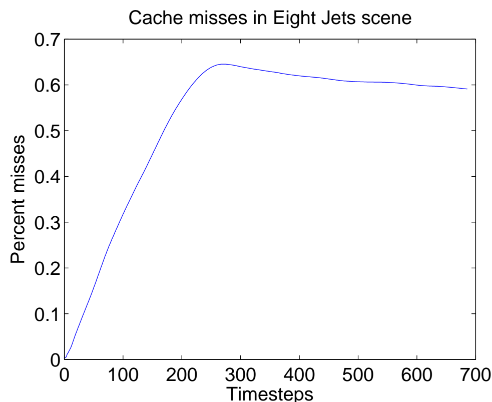
Prior to beginning computation on row j of array d, we prefetched the rows $j-1$, j, and $j+1$ from the d0 array. While iterating over the elements of row j, we first checked to see if the semi-Lagrangian ray terminated in a 3x3 neighborhood of the origin. If so, we made use of the prefetched d0 values for the interpolation. Else, we performed the more expensive fetch from main memory. We implemented this caching scheme on the Cell, where DMAs to main memory can be controlled in a highly granular manner. An equivalent scheme could be implemented in CUDA, with the cores performing a parallel prefetch of d0 to shared memory prior to beginning a new row.

Two test scenes were constructed to measure the cache hit rate of the static scheme. In the 2D scene, eight jets of velocity and density were injected into a 512^2 simulation at different points and in different directions in order induce a wide variety of directions into the velocity field (Table 4 top). In the 3D scene, a buoyant pocket of smoke is continually inserted into a 64^3 simulation, much like in [Fedkiw et al. 2001]. In both cases, the diffusion and viscosity constants were set to zero so that the velocities maintained their larger magnitudes for as long as possible.

The cache miss rates can be seen in Figure 3. For the 2D scene, the miss rate never exceeds 0.65%, and actually appears to be decreasing slightly as time progresses. In the 3D scene, the miss rate never exceeds 0.44%. In Table 4, we list the effective bandwidths achieved by the advection stage on the Cell for the 2D scene. We emphasize that we are not actually eliminating any memory accesses from the advection stage. We are instead hypothesizing that most of the memory accesses can be coalesced into a large, coherent access, and that a simple caching scheme can reap the bandwidth benefits of this kind of access. Table 4 validates this hypothesis, as we achieve roughly twice the bandwidth, which translates to a 2x speedup of the advection stage.

6 Conclusions and Future Work

Through a detailed flop and bandwidth analysis, we have come to the conclusion that *Stable Fluids* is a bandwidth-bound algorithm on current CPU, GPU and Cell architectures. Performance comparisons of 16 bit and 32 bit GPU codes lend further support to this conclusion. Using this knowledge, we proposed the use a Mehrstellen discretization, and found that not only will it further occupy the idle cores, but it will allow the linear solver to terminate 17% earlier in 2D, and 33% earlier in 3D. We also designed a static caching scheme for the advection stage that makes more effective use of the available memory bandwidth. We measured a 2x speedup in the advection stage using this scheme on the Cell.



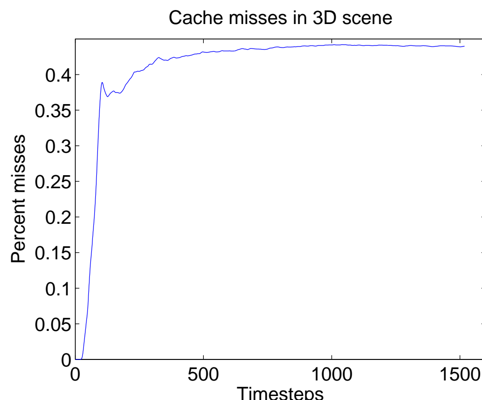


Figure 3: *Static advection cache misses in two test scenes.* **Top:** *Misses in the 2D eight jets scene over 686 timesteps. The maximum percentage of misses was 0.65% and the mean miss rate after the knee (300 timesteps) was 0.61%.* **Bottom:** *Misses in the 3D scene over 1518 timesteps. The maximum percentage of misses was 0.44%.*

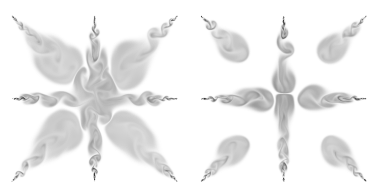


Figure 4: *Two frames from the 2D caching test scene.*

There have been numerous extensions proposed for the *Stable Fluids* algorithm, the most significant of which is perhaps the handling of free surfaces [Foster and Fedkiw 2001]. The most obvious future work is to carry out a similar hardware-aware analysis for each of these extensions to see if similar opportunities for optimizations can be identified. Free surfaces use fast marching and level set methods, which are more challenging to map to parallel hardware in general, so the analysis becomes correspondingly more challenging as well.

Our analysis of Mehrstellen discretizations only applies to relaxation solvers such as Jacobi, Gauss-Seidel, SSOR, and some versions of Multigrid. It does not apply to preconditioned conjugate gradient (PCG), which is arguably the more popular solver for non-realtime applications of *Stable Fluids*. The convergence rates of PCG are less well understood however, so a similar analysis would be challenging. As relaxation solvers are used in other real-time graphics applications, such the LCP solvers in rigid body simulation [Smith 2007], it would be interesting to see if the results we obtained here could be applied to these domains as well.

References

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖDER, P. 2003. Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. In *Proceedings of SIGGRAPH*.

COLLATZ, L. 1960. *The Numerical Treatment of Differential Equations*. Springer-Verlag.

CRANE, K., TARIQ, S., AND LLAMAS, I. 2007. *GPU Gems 3*. ch. Real-time Simulation and Rendering of 3D Fluids.

DEMMEL, J. 1997. *Applied Numerical Linear Algebra*. SIAM.

ENRIGHT, D., FEDKIW, R., FERZIGER, J., AND MITCHELL, I. 2002. A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics 183*, 83–116.

FEDKIW, R., STAM, J., AND JENSEN, H. W. 2001. Visual simulation of smoke. *Proc. of SIGGRAPH*, 15–22.

FOSTER, N., AND FEDKIW, R. 2001. Practical animation of liquids. *Proc. of SIGGRAPH*, pp. 15–22.

GOODNIGHT, N., WOOLLEY, C., LEWIN, G., LUEBKE, D., AND HUMPHREYS, G. 2003. A multigrid solver for boundary value problems using programmable graphics hardware. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*.

GUPTA, M., KOUATCHOU, J., AND ZHANG, J. 1997. Comparison of second- and fourth-order discretizations for multigrid poisson solvers. *Journal of Computational Physics*, 226–232.

HARADA, T., KOSHIZUKA, S., AND KAWAGUCHI, Y. 2007. Smoothed particle hydrodynamics on gpus. In *Computer Graphics International*, 63–70.

HARRIS, M. 2004. *GPU Gems*. ch. Fast Fluid Dynamics Simulation on the GPU.

HARRIS, M. 2005. *GPU Gems 2*. ch. Mapping Computational Concepts to the GPU.

IBM, 2007. Cell broadband engine programming handbook. http://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine.

INTEL, 2007. Dual-core intel xeon processor 5100 series datasheet. http://www.intel.com/design/xeon/datashts/313355.htm.

KIM, B., LIU, Y., LLAMAS, I., JIAO, X., AND ROSSIGNAC, J. 2007. Simulation of bubbles in foam by volume control. In *Proceedings of ACM SIGGRAPH*.

KISTLER, M., PERRONE, M., AND PETRINI, F. 2006. Cell multiprocessor interconnection network: Built for speed. *IEEE Micro 26*, 3.

KLINGNER, B. M., FELDMAN, B. E., CHENTANEZ, N., AND O'BRIEN, J. F. 2006. Fluid animation with dynamic meshes. In *Proceedings of ACM SIGGRAPH 2006*.

LAX, P., AND WENDROFF, B. 1960. Systems of conservation laws. *Communications on Pure and Applied Mathematics*, 217–237.

LI, W., WEI, X., , AND KAUFMAN, A. 2003. Implementing lattice boltzmann computation on graphics hardware. *The Visual Computer*, 444–456.

MORELAND, K., AND ANGEL, E. The fft on a gpu. In *Eurographics Workshop on Graphics Hardware*, 112.

NVIDIA, 2007. Geforce 8 series. http://www.nvidia.com/page/geforce8.html.

NVIDIA. 2007. *Nvidia CUDA Programming Guide*.

SLEPC, 2007. Scalable library for eigenvalue problem computations. http://www.grycap.upv.es/slepc/.

SMITH, R., 2007. Ode: Open dynamics engine. http://www.ode.org/.

STAM, J. 1999. Stable fluids. *Proceedings of ACM SIGGRAPH*, 121–128.

STAM, J. 2003. Real-time fluid dynamics for games. In *Proceedings of the Game Developer Conference*.

Figure 5: *3D Smoke rising with standard Jacobi (left) and Mehrstellen Jacobi (right). The results are virtually identical, but the pressure solve runs 33%.*