8

Implementing a Fast DDOF Solver

Holger Grin
Advanced Micro Devices, Inc.

This gem presents a fast and memory-efficient solver for the diffusion depth-offield (DDOF) technique introduced by Michael Kass et al. [2006] from Pixar. DDOF is a very high-quality depth-of-field effect that is used in motion pictures. It essentially works by simulating a heat diffusion equation to compute depth-offield blurs. This leads to a situation that requires solving systems of linear equations.

Kass et al. present an algorithm for solving these systems of equations, but the memory footprint of their solver can be prohibitively large at high display resolutions. The solver described in this gem greatly reduces the memory footprint and running time of the original DDOF solver by skipping the construction of intermediate matrices and the execution of associated rendering passes. In contrast to the solver presented by Shishkovtsov and Rege [2010], which uses DirectX 11, compute shaders, and a technique called *parallel cyclic reduction* [Zhang et al. 2010], this gem utilizes a solver that runs only in pixel shaders and can be implemented on DirectX 9 and DirectX 10 class hardware as well.

8.1 Introduction

DDOF essentially relies on solving systems of linear equations described by tridiagonal systems like the following:

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & a_3 & b_3 & c_3 & \\ & & \ddots & \ddots & \ddots \\ 0 & & & a_n & b_n \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \qquad (8.1)$$

It needs to solve such a system of equations for each row and each column of the input image. Because there are always only three coefficients per row of the matrix, the system for all rows and columns of an input image can be packed into a four-channel texture that has the same resolution as the input image.

The x_i represent the pixels of one row or column of the input image, and the y_i represent the pixels of one row or column of the yet unknown image that is diffused by the heat equation. The values a_i, b_i, and c_i, are derived from the circle of confusion (CoC) in a neighborhood of pixels in the input image [Riguer et al. 2004]. Kass et al. take a single time step from the initial condition $y_i = x_i$ and arrive at the equation

$$(y_i + x_i) = \beta_i(y_{i+1} - y_i) - \beta_{i-1}(y_i - y_{i-1}) \qquad (8.2)$$

Here, each β_i represents the heat conductivity of the medium at the position *i*. Further on, β_0 and β_n are set to zero for an image row of resolution n so that the row is surrounded by insulators. If one now solves for x_i then the resulting equation reveals a tridiagonal structure:

$$x_i = -\beta_{i-1}y_{i-1} + (\beta_{i-1} + \beta_i + 1)y_i - \beta_i y_{i+1} \qquad (8.3)$$

It is further shown that β_i is the square of the diameter of the CoC at a certain pixel position i. Setting up a_i ,b_i and c_i, now is a trivial task.

As in Kass et al., the algorithm presented in this gem first calculates the diffusion for all rows of the input image and then uses the resulting horizontallydiffused image as an input for a vertical diffusion pass.

Kass et al. use a technique called *cyclic reduction* (CR) to solve for the y_i Consider the set of equations

$$\begin{array}{ccccccccccc} a_{i-1}y_{i-2} & + & b_{i-1}y_{i-1} & + & c_{i-1}y_i & & & & & = & x_{i-1} \\ & & a_i y_{i-1} & + & b_i y_i & + & c_i y_{i+1} & & & = & x_i \\ & & & & a_{i+1}y_i & + & b_{i+1}y_{i+1} & + & c_{i+1}y_{i+2} & = & x_{i+1} \end{array} \qquad (8.4)$$

CR relies on the simple fact that a set of three equations containing five unknowns y_{i-2}, y_{i-1}, y_i, y_{i+1} and y_{i+2}, like those shown in Equation (8.4), can be reduced to one equation by getting rid of y_{i-1} and y_{i+1}. This is done by multiplying the first equation by - a_i/b_{i-1} and multiplying the third equation by - c_i/b_{i+1} to produce the equations

$$\begin{aligned} -a_i\frac{a_{i-1}}{b_{i-1}}y_{i-2} - a_i y_{i-1} - a_i\frac{c_{i-1}}{b_{i-1}}y_i &= -\frac{a_i}{b_{i-1}}x_{i-1} \\ -c_i\frac{a_{i+1}}{b_{i+1}}y_i - c_i y_{i+1} - c_i\frac{c_{i+1}}{b_{i+1}}y_{i+2} &= -\frac{c_i}{b_{i+1}}x_{i+1} \end{aligned} \qquad (8.5)$$

When these are added to the second line of Equation (8.4), we arrive at the equation

$$-a_i y_{i-2} + b_i' y_i + c_i' y_{i+2} = x_i' \ , \qquad (8.6)$$

Which no longer references y_{i-1}, or y_{i+1}, This means that we can halve the number of unknowns for each row or column by running a shader that writes its output to a new texture that is half as wide or half as high as the input texture. Repeating this process reduces the number of unknowns to a number that is low enough to be directly solved by the shader. The values of a'_i, b'_i, and c'_i, are functions of the original coefficients in Equation (8.4) and need to be stored in a texture. We also need a texture containing the value of each x'_i, (which are also functions of the original coefficients and the x_i), and since it is a color image, all operations are, in fact, vector operations.

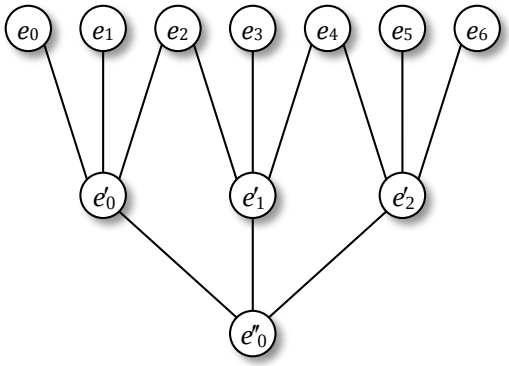

Figure 8.1. Reduction of a set of seven unknowns.

Figure 8.1 illustrates the process of reducing the set of unknowns for an input row that is seven pixels wide. Each circle, or ei, stands for an equation with a set of ai, bi, ci, xi values and their respective input/output textures. Now, at e″ 0 we have reached the case where the

tridiagonal system for one row has been reduced to a row with just one element. We can assume that this row of size one is still surrounded by insulators, and thus, the equation

$$a_i'' y_{i-2} + b_i'' y_i + c_i'' y_{i+2} = x_i'' \ , \qquad (8.7)$$

Simplifies to the equation

$$b_i'' y_i = x_i'' \ , \qquad (8.8)$$

since the a''_i and the c''_i need to be set to zero to implement the necessary insulation. This now allows us to trivially solve for y_i.

For modern GPUs, reducing the system of all image rows to size one does not make use of all the available hardware threads, even for high display resolutions. In order to generate better hardware utilization, the shaders presented in this chapter actually stop at two or three systems of equations and directly solve for the remaining unknown y_i. Each of the two or three results gets computed, and the proper one is chosen based on the output position (`SV_POSITION`) of the pixel in order to avoid dynamic flow control. The shaders need to support solving for the three remaining unknown values as well because input resolutions that are not a power of two are never reduced to a point where two unknown values are left.

If we reduce the rows of an input image and a texture containing the a_i, b_i, and c_i, then the resulting y_i now represent one column of the resulting horizontally diffused image. The y_i values are written to a texture that is used to create a full-resolution result. In order to achieve this, the y_i values need to be substituted back up the resolution chain to solve for all the other unknown y_i. This means running a number of solving passes that blow the results texture up until a fullresolution image with all y_i values is reached. Each such pass doubles the number of known y_i.

If we assume that a particular value y_i kept during reduction is on an odd index i, then we just have to use the value of y_i that is available one level higher. What needs to be computed are the values of y_i for even indices. Figure 8.2 illustrates this back-substitution process and shows which values get computed and which just get copied.

In fact, the first line from Equation (8.4) can be used again because y_i and y_{i-2} are already available from the last solving pass. We can now just compute y_{i-1} as follows:

$$y_{i-1} = \frac{x_{i-1} - a_i y_{i-2} - c_{i-1} y_i}{b_{i-1}} \ , \qquad (8.9)$$

Running all solving passes results in the diffused image y.

For an in-depth discussion of DDOF, please consult Kass et al. [2006], as such a discussion is beyond the scope of this gem.

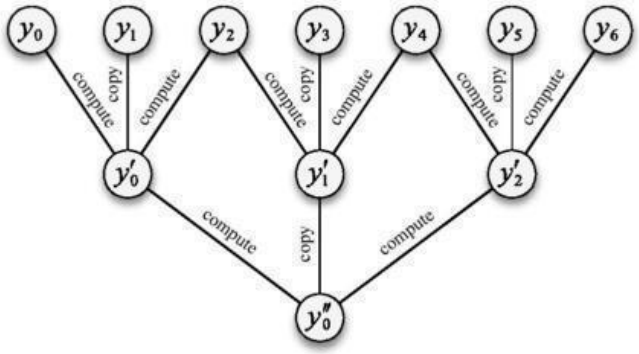

Figure 8.2. Back-substituting the known y.

8.2 Modifying the Basic CR Solver

As mentioned at the beginning of this chapter, one of the big problems with the solvers proposed so far is memory consumption. So the main goal of this gem is to reduce the memory footprint of the solver by as much as possible. Experiments have shown that these changes also increase the speed of the solver significantly as a side effect. The following changes to the original CR solver algorithm achieve this goal:

1. Never construct the highest-resolution or second highest-resolution (see the second change below) textures containing the a_i, b_i, and c_i. Instead, compute the a_i, b_i, and c_i, on the fly when doing the first reduction and the last solving pass. This saves the memory of one fullscreen buffer and two half-sized buffers that need to have a 32-bit floating-point format. It also makes a construction pass for the highest-resolution a_i, b_i, and c_i redundant.

2. Perform an initial four-to-one reduction instead of a two-to-one reduction to reduce the memory footprint even more. Instead of getting rid of the two unknown variables y_{i-1} and y_{i+1}, the solver gets rid of y_{i-2}, y_{i-1}, y_{i+1}, and y_{i+2}, leaving only y_{i-3}, y_i and y_{i+3}. It is possible to derive formulas that describe the four-to-one reduction and to write a shader computing these. Instead, a more hands-on approach is used. Listing 8.1 shows a shader model 4 code

fragment implementing a four-to-one horizontal reduction. This shader performs the four-to-one reduction in three phases (see comments in the shader). Phase 1 gathers and computes all the data that is necessary to compute the four-to-one reduction. Phase 2 performs a series of two-to-one reductions on the data from Phase 1. Phase 3 performs a final two-to-one reduction, completing the four-to-one reduction. A shader performing vertical reduction can be easily derived from this code fragment.

Listing 8.1. Horizontal four-to-one reduction.

```
struct Reduce0
{
  float4 abc : SV_TARGET0;
  float4 x   : SV_TARGET1;
};

Reduce0 InitialReduceHorz4(PS_INPUT input)
{
  Reduce0 output;

  float fPosX = floor(input.Pos.x) * 4.0 + 3.0;
  int3 i3LP = int3(fPosX, input.Pos.y, 0);

  //Phase 1: Gather and compute all data necessary
  //for the four-to-one reduction.

  //Compute the CoC values in the support needed for
  //the four-to-one reduction.
  float fCoc_4 = computeCoC(i3LP, int2(-4, 0));
  float fCoc_3 = computeCoC(i3LP, int2(-3, 0));
  float fCoc_2 = computeCoC(i3LP, int2(-2, 0));
  float fCoc_1 = computeCoC(i3LP, int2(-1, 0));
  float fCoc0 = computeCoC(i3LP, int2(0, 0));
  float fCoc1 = computeCoC(i3LP, int2(1, 0));
  float fCoc2 = computeCoC(i3LP, int2(2, 0));
```

```
float fCoc3 = computeCoC(i3LP, int2(3, 0));
float fCoc4 = computeCoC(i3LP, int2(4, 0));

//Ensure insulation at the image borders by setting
//the CoC to 0 outside the image.
float fCoc_4 = (fPosX - 4.0 >= 0.0) ? fCoc_4 : 0.0;
float fCoc_3 = (fPosX - 3.0 >= 0.0) ? fCoc_3 : 0.0;
float fCoc4 = (fPosX + 4.0) < g_vImageSize.x ? fCoc4 : 0.0;
float fCoc3 = (fPosX + 3.0) < g_vImageSize.x ? fCoc3 : 0.0;
float fCoc2 = (fPosX + 2.0) < g_vImageSize.x ? fCoc2 : 0.0;
float fCoc1 = (fPosX + 1.0) < g_vImageSize.x ? fCoc1 : 0.0;

//Use the minimum CoC as the real CoC as described in Kass et al.
float fRealCoC_4 = min(fCoc_4, fCoc_3);
float fRealCoC_3 = min(fCoc_3, fCoc_2);
float fRealCoC_2 = min(fCoc_2, fCoc_1);
float fRealCoC_1 = min(fCoc_1, fCoc_0);
float fRealCoC0 = min(fCoc0, fCoc1);
float fRealCoC1 = min(fCoc1, fCoc2);
float fRealCoC2 = min(fCoc2, fCoc3);
float fRealCoC3 = min(fCoc3, fCoc4);

//Compute beta values interpreting the CoCs as the diameter.
float bt_4 = fRealCoC_4 * fRealCoC_4;
float bt_3 = fRealCoC_3 * fRealCoC_3;
float bt_2 = fRealCoC_2 * fRealCoC_2;
float bt_1 = fRealCoC_1 * fRealCoC_1;
float bt0 = fRealCoC0 * fRealCoC0;
float bt1 = fRealCoC1 * fRealCoC1;
float bt2 = fRealCoC2 * fRealCoC2;
float bt3 = fRealCoC3 * fRealCoC3;

//Now compute the a, b, c and load the x in the support
//region of the four-to-one reduction.
```

```
float3 abc_3 = float3(-bt_4, 1.0 + bt_3 + bt_4, -bt_3);
float3 x_3 = txX.Load(i3LP, int2(-3, 0)).xyz;

float3 abc_2 = float3(-bt_3, 1.0 + bt_2 + bt_3, -bt_2);
float3 x_2 = txX.Load(i3LP, int2(-2, 0)).xyz;

float3 abc_1 = float3(-bt_2, 1.0 + bt_1 + bt_2, -bt_1);
float3 x_1 = txX.Load(i3LP, int2(-1, 0)).xyz;

float3 abc_0 = float3(-bt_1, 1.0 + bt_0 + bt_1, -bt_0);
float3 x_0 = txX.Load(i3LP, int2(0, 0)).xyz;

float3 abc1 = float3(-bt0, 1.0 + bt1 + bt0, -bt1);
float3 x_1 = txX.Load(i3LP, int2(1, 0)).xyz;

float3 abc2 = float3(-bt1, 1.0 + bt2 + bt1, -bt2);
float3 x_2 = txX.Load(i3LP, int2(2, 0)).xyz;

float3 abc3 = float3(-bt2, 1.0 + bt3 + bt2, -bt3);
float3 x_3 = txX.Load(i3LP, int2(3, 0)).xyz;

//Phase 2: Reduce all the data by doing all two-to-one
//reductions to get to the next reduction level.
float a_1 = -abc_2.x / abc_3.y;
float g_1 = -abc_2.z / abc_1.y;
float a0 = -abc0.x / abc_1.y;
float g0 = -abc0.z / abc1.y;
float a1 = -abc2.x / abc1.y;
float g1 = -abc2.z / abc3.y;

float3 abc_p = float3(a_1 * abc_3.x, abc_2.y + a_1 * abc_3.z
  + g_1 * abc_1.x, g_1 * abc_1.z);
float3 x_p = float3(x_2 + a_1 * x_3 + g_1 * x_1);
```

```
  float3 abc_c = float3(a0 * abc_1.x, abc0.y + a0 * abc_1.z
    + g0 * abc1.x, g0 * abc_1.z);
  float3 x_c = float3(x0 + a0 * x_1 + g0 * x1);

  float3 abc_n = float3(a1 * abc1.x, abc2.y + a1 * abc1.z
    + g1 * abc3.x, g1 * abc3.z);
  float3 x_n = float3(x2 + a1 * x1 + g1 * x3);

  //Phase 3: Do the final two-to-one reduction to complete
  //the four-to-one reduction.
  float a = -abc_c.x / abc_p.y;
  float g = -abc_c.z / abc_n.y;

  float3 res0 = float3(a * abc_p.x, g * abc_c.y, a * abc_p.z +
    g * abc_n.x, g * abc_n.z );
  float3 res1 = float3(x_c + a * x_p + g * x_n);

  output.abc = float4(res0, 0.0);
  output.x = float4(res1, 0.0);
  return output;
}
```

3. Perform a final one-to-four solving pass to deal with the initial four-to-one reduction pass. Again, a very hands-on approach for solving the problem at hand is used, and it also has three phases. Since an initial four-to-one reduction shader was used, we don't have all the data available to perform the needed one-to-four solving pass. Phase 1 of the shader therefore starts to reconstruct the missing data from the unchanged and full-resolution input data in the same fashion that was used in Listing 8.1. Phase 2 uses this data to perform several one-to-two solving steps to produce the missing y, values of the intermediate pass that we skip. Phase 3 finally uses all that data to produce the final result. Listing 8.2 shows a shader model 4 code fragment implementing the corresponding algorithm for that final solver stage. Again, only the code for the horizontal version of the algorithm is shown.

Listing 8.2. Final stage of the solver.

```
float4 FinalSolveHorz4(PS_INPUT input): SV_TARGET
{
  // First reconstruct the level 1 x, abc.
  float fPosX = floor(input.Pos.x * 0.25) * 4.0 + 3.0;
  int3 i3LoadPos = int3(fPosX, input.Pos.y, 0);

  // Phase 1: Gather data to reconstruct intermediate data
  // lost when skipping the first two-to-one reduction step
  // of the original solver.
  float fCoC_5 = computeCoC(i3LoadPos, int2(-5, 0));
  float fCoC_4 = computeCoC(i3LoadPos, int2(-4, 0));
  float fCoC_3 = computeCoC(i3LoadPos, int2(-3, 0));
  float fCoC_2 = computeCoC(i3LoadPos, int2(-2, 0));
  float fCoC_1 = computeCoC(i3LoadPos, int2(-1, 0));
  float fCoC0 = computeCoC(i3LoadPos, int2(0, 0));
  float fCoC1 = computeCoC(i3LoadPos, int2(1, 0));
  float fCoC2 = computeCoC(i3LoadPos, int2(2, 0));
  float fCoC3 = computeCoC(i3LoadPos, int2(3, 0));
  float fCoC4 = computeCoC(i3LoadPos, int2(4, 0));

  fCoC_5 = (fPosX - 5.0 >= 0.0) ? fCoC_5 : 0.0;
  fCoC_4 = (fPosX - 4.0 >= 0.0) ? fCoC_4 : 0.0;
  fCoC_3 = (fPosX - 3.0 >= 0.0) ? fCoC_3 : 0.0;
  fCoC4 = (fPosX + 4.0 < g_vIMageSize.x) ? fCoC4 : 0.0;
  fCoC3 = (fPosX + 3.0 < g_vIMageSize.x) ? fCoC3 : 0.0;
  fCoC2 = (fPosX + 2.0 < g_vIMageSize.x) ? fCoC2 : 0.0;
  fCoC1 = (fPosX + 1.0 < g_vIMageSize.x) ? fCoC1 : 0.0;

  float fRealCoC_5 = min(fCoC_5, fCoC_4);
  float fRealCoC_4 = min(fCoC_4, fCoC_3);
  float fRealCoC_3 = min(fCoC_3, fCoC_2);
  float fRealCoC_2 = min(fCoC_2, fCoC_1);
```

```
float fRealCoC_1 = min(fCoC_1, fCoC0);

float fRealCoC0 = min(fCoC0, fCoC1);
float fRealCoC1 = min(fCoC1, fCoC2);
float fRealCoC2 = min(fCoC2, fCoC3);
float fRealCoC3 = min(fCoC3, fCoC4);

float b_5 = fRealCoC_5 * fRealCoC_5;
float b_4 = fRealCoC_4 * fRealCoC_4;
float b_3 = fRealCoC_3 * fRealCoC_3;
float b_2 = fRealCoC_2 * fRealCoC_2;
float b_1 = fRealCoC_1 * fRealCoC_1;

float b0 = fRealCoC0 * fRealCoC0;
float b1 = fRealCoC1 * fRealCoC1;
float b2 = fRealCoC2 * fRealCoC2;
float b3 = fRealCoC3 * fRealCoC3;

float3 abc_4 = float3(-b_5, 1.0 + b_4 + b_5, -b_4);
float3 x_4 = txX.Load(i3LoadPos, int2(-4, 0)).xyz;

float3 abc_3 = float3(-b_4, 1.0 + b_3 + b_4, -b_3);
float3 x_3 = txX.Load(i3LoadPos, int2(-3, 0)).xyz;

float3 abc_2 = float3(-b_3, 1.0 + b_2 + b_3, -b_2);
float3 x_2 = txX.Load(i3LoadPos, int2(-2, 0)).xyz;

float3 abc_1 = float3(-b_2, 1.0 + b_1 + b_2, -b_1);
float3 x_1 = txX.Load(i3LoadPos, int2(-1, 0)).xyz;

float3 abc0 = float3(-b_1, 1.0 + b0 + b_1, -b0);
float3 x0 = txX.Load(i3LoadPos, int2(0, 0)).xyz;

float3 abc1 = float3(-b0, 1.0 + b1 + b0, -b1);
```

```
float3 x1 = txX.Load(i3LoadPos, int2(1, 0)).xyz;

float3 abc2 = float3(-b1, 1.0 + b2 + b1, -b2);
float3 x2 = txX.Load(i3LoadPos, int2(2, 0)).xyz;

float3 abc3 = float3(-b2, 1.0 + b3 + b2, -b3);
float3 x3 = txX.Load(i3LoadPos, int2(3, 0)).xyz;

float a_2 = -abc_3.x / abc_4.y;
float g_2 = -abc_3.z / abc_2.y;

float a_1 = -abc_2.x / abc_3.y;
float g_1 = -abc_2.z / abc_1.y;

float a0 = -abc0.x / abc_1.y;
float g0 = -abc0.z / abc1.y;

float a1 = -abc2.x / abc1.y;
float g1 = -abc2.z / abc3.y;

float3 l1_abc_pp = float3(a_2 * abc_4.x,
  abc_3. + a_2 * abc_4.z + g_2 * abc_2.x, g_2 * abc_2.z);
float3 l1_x_pp = float3(x_3 + a_2 * x_4 + g_2 * x_2);

float3 l1_abc_p = float3(a_1 * abc_3.x,
  abc_2.y + a_1 * abc_3.z + g_1 * abc_1.x, g_1 * abc_1.z);
float3 l1_x_p = float3(x_2 + a_1 * x_3 + g_1 * x_1);

float3 l1_abc_c = float3(a0 * abc_1.x,
  abc0.y + a0 * abc_1.z + g0 * abc1.x, g0 * abc1.z);
float3 l1_x_c  = float3(x0 + a0 * x_1 + g0 * x1);

float3 l1_abc_n = float3(a1 * abc1.x,
  abc2.y + a1 * abc1.z + g1 * abc3.x g1 * abc3.z);
```

```
  float3 l1_x_n = float3(x2 + a1 * x1 + g1 * x3);

  // Phase 2: Now solve for the intermediate-level
  // data we need to compute to go up to full resolution.
  int3 i3LoadPosC = int3(input.Pos.x * 0.25, input.Pos.y, 0);
  float3 l2_y0 = txYn.Load(i3l2_LoadPosC).xyz;
  float3 l2_y1 = txYn.Load(i3l2_LoadPosC, int2(1, 0)).xyz;
  float3 l2_y_1 = txYn.Load(i3l2_LoadPosC, int2(-1, 0)).xyz;
  float3 l2_y_2 = txYn.Load(i3l2_LoadPosC, int2(-2, 0)).xyz;
  float3 l1_y_c = l2_y0;
  float3 l1_y_p = (l1_x_p - l1_abc_p.x * l2_y_1
    - l1_abc_p.z * l2_y0) / l1_abc_p.y;
  float3 l1_y_pp = l2_y_1;
  float3 l1_y_n = (l1_x_n - l1_abc_n.x * l2_y0
    - l1_abc_n.z * l2_y1) / l1_abc_n.y;

  // Phase 3: Now use the intermediate solutions to solve
  // for the full result.
  float3 fRes3 = l2_y0;
  float3 fRes2 = (x_1 - abc_1.x * l1_y_p
    - abc_1.z * l1_y_c) / abc_1.y;       //y_1
  float3 fRes1 = l1_y_p;                //y_2
  float3 fRes0 = (x_3 - abc_3.x * l1_y_pp
    - abc_3.z * l1_y_p) / abc_3.y;       //y_3

  float3 f3Res[4] = {fRes0, fRes1, fRes2, fRes3};
  return (float4(f3Res[uint(input.Pos.x) & 3], 0.0));
}
```

4. Stop at two or three unknowns instead of reducing it all down to just one unknown. Given that the number of hardware threads in a modern GPU is in the thousands, this actually makes sense because it keeps a lot more threads of a modern GPU busy compared to going down to just one unknown. Cramer's rule is used to solve the resulting 2 × 2 or 3 × 3 equation systems.

5. Optionally pack the evolving y_i, and the a_i, b_i and c_i, into just one fourchannel `uint32` texture to further save memory and to gain speed since the number of texture operations is cut down by a factor of two. This packing uses Shader Model 5 instructions (see Listing 8.3) and relies on the assumption that the x_i values can be represented as 16-bit floating-point values. It further assumes that one doesn't need the full mantissa of the 32-bit floating-point values for storing a_i, b_i and c_i, and it steals the six lowest mantissa bits of each one to store a 16-bit x_i, channel.

Listing 8.3. Packing/unpacking all solver variables into/from one rgab32 uint value.

```
/ Pack six floats into a uint4 variable. This steals six mantissa bits
// from the three. 32-bit FP values that hold abc to store x.
uint4 pack (float3 abc, float3 x)
{
  uint z = f3tof16(x.z);
  return (uint4(((asuint(abc.x) & 0xFFFFFFC0) | (z & 0x3F)),
    ((asuint(abc.y) & 0xFFFFFFC0) | ((z >> 6) & 0x3F)),
    ((asuint(abc.z) & 0xFFFFFFC0) | ((z >> 12) & 0x3F)),
    (f32tof16(x.x) + (f32tof16(x.y) << 16))));
}

struct ABC_X
{
  float3 abc;
  float3 x;
};

ABC_X unpack(uint4 d)
{
  ABC_X res;
  res.abc = asfloat(d.xyz & 0xFFFFFFC0);
  res.x.xy = float2(f16tof32(d.w), f16tof32(d.w >> 16));
  res.x.z = f16tof32(((d.x & 0x3F) + ((d.y & 0x3F) << 6) +
    ((d.z & 0x3F) << 12)));
```

```
  return (res);
}
```

8.3 Results

Table 8.1 shows how various implementations of the DDOF solver perform at various resolutions and how much memory each solver consumes. These performance numbers (run on a system with an AMD HD 5870 GPU with 1 GB of video memory) show that the improved solver presented in this gem outperforms the traditional solvers in terms of running time and also in terms of memory requirements.

In the settings used in these tests, the packing shown in Listing 8.3 does not show any obvious differences (see Figure 8.3). Small differences are revealed in Figure 8.4, which shows the amplified absolute difference between the images in Figure 8.3. If these differences stay small enough, then packing should be used in DirectX 11 rendering paths in games that implement this gem.

Table 8.1. Comparison of solver efficiency.

Resolution	Solver	Running Time (ms)	Memory (~MB)
1280×1024	Standard solver	2.46	90
1280×1024	Standard solver + Packing	1.97	70
1280×1024	Four-to-one reduction	1.92	50
1280×1024	Four-to-one reduction + Packing	1.87	40
1600×1200	Standard solver	3.66	132
1600×1200	Standard solver + Packing	2.93	102
1600×1200	Four-to-one reduction	2.87	73
1600×1200	Four-to-one reduction + Packing	2.75	58
1920×1200	Standard solver	4.31	158
1920×1200	Standard solver + Packing	3.43	122
1920×1200	Four-to-one reduction	3.36	88
1920×1200	Four-to-one reduction + Packing	3.23	70
2560×1600	Standard solver	7.48	281
2560×1600	Standard solver + Packing	5.97	219

2560×1600	Four-to-one reduction	5.8	156
2560×1600	Four-to-one reduction + Packing	5.59	125

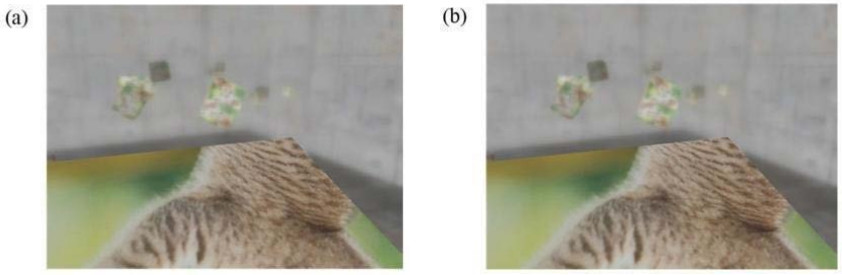

Figure 8.3. A comparison between images for which (a) packing was not used and (b) the packing shown in Listing 8.3 was used.

Figure 8.4. Absolute difference between the images in Figure 8.3, multiplied by 255 and inverted.

References

[Kass et al. 2006] Michael Kass, Aaron Lefohn, and John Owens. "Interactive Depth of Field Using Simulated Diffusion on a GPU." Technical report. Pixar Animation Studios, 2006. Available at http://www.idav.ucdavis.edu/publications/print_pub? pub_id=898.

[Shishkovtsov and Rege 2010] Oles Shishkovtsov and Ashu Rege. "DX11 Effects in Metro 2033: The Last Refuge." Game Developers Conference 2010. Available at

[Riguer et al. 2004] Guennadi Riguer, Natalya Tatarchuk, and John Isidoro. "Real-Time Depth of Field Simulation." *ShaderX2*, edited by Wolfgang Engel. Plano, TX: Wordware, 2004. Available at http://ati.amd.com/developer/shaderx/shaderx2_ real-timedepthoffieldsimulation.pdf.

[Zhang et al. 2010] Yao Zhang, Jonathan Cohen, and John D. Owens, "Fast Tridiagonal Solvers on the GPU." *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2010. Available at http://www. idav.ucdavis.edu/func/retum_pdf?pub_id=978.