CPU, SMP and GPU Implementations of Nohalo Level 1, a Fast Co-Convex Antialiasing Image Resampler

Nicolas Robidoux
Département de mathématiques et d'informatique
Université Laurentienne
Sudbury ON
P3E 2C6 Canada
nrobidoux@cs.laurentian.ca

Minglun Gong
Computer Science Department
Memorial University of Newfoundland
St. John's NL
A1B 3X5 Canada
gong@cs.mun.ca

John Cupitt
Experimental Medicine and Toxicology
Imperial College, London
Hammersmith Campus
London W12 0NN, UK
j.cupitt@imperial.ca.uk

Adam Turcotte
Department of Mathematics and Computer Science
Laurentian University
Sudbury ON
P3E 2C6 Canada
adam.turcotte@gmail.com

Kirk Martinez
Electronics and Computer Science Department
University of Southampton
Southampton SO17 1BJ, UK
km@ecs.soton.ac.uk

ABSTRACT

This article introduces Nohalo level 1 ("Nohalo"), the simplest member of a family of image resamplers which straighten diagonal interfaces without adding noticeable nonlinear artifacts. Nohalo is interpolatory, co-monotone, co-convex, antialiasing, local average preserving, continuous, and exact on linears.

Like many edge-enhancing methods, Nohalo has two main stages: first, nonlinear interpolation is used to create a double-density version of the original image; this double-density image is then resampled with bilinear interpolation. Nohalo is especially suited for GPU computing because the nonlinear slopes can be computed once and stored in a low bit-depth texture without rounding error, because the final bilinear stage can be performed in hardware, and because monotonicity allows full use of the texture's dynamic range. Demand-driven implementations for CPUs and SMPs are more complex, and require extra work to fix bottlenecks. Efficient implementations of the minmod function are key to performance.

Three implementations of Nohalo are presented and benchmarked: a CPU version in C for the graphics library GEGL, an SMP version in C++ for the graphics library VIPS and a GPU version in HLSL for DirectX. The GPU implementation is branch-free thanks to the discovery of a simple formula for the pixel values of the double density image. Branches are eliminated in the demand-driven C/C++ implementations by reflecting, if needed, Nohalo's 12-point stencil with pointer shifts. Overall, Nohalo is not much slower than standard bicubic resamplers.

Compared to twenty-three alternatives in tests involving the re-enlargement of images downsampled with nearest neighbour, Nohalo gets the best PSNRs.

C3S2E-09 2009, May 19-21 Montreal [QC Canada]
Editor: B. C. Desai

Categories and Subject Descriptors

I.4.3 [**Image processing and computer vision**]: Enhancement—*Filtering*; I.4.0 [**Image Processing and Computer Vision**]: General—*Image processing software*; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Antialiasing*

General Terms

Algorithms, performance.

Keywords

Fast image resampling, nonlinear image filtering, antialiasing, edge enhancement, video upsampling, minmod limiter, zooming, quantitative comparison of image enlargement methods, co-convex interpolation, abyss policy, GPU, SMP, arithmetic branching, benchmark, HLSL, VIPS, GEGL, natural and not-a-knot boundary conditions

1. INTRODUCTION

Image resampling [10] and upsizing (super-resolution) [17] have been extensively studied. Yet, the definitive all-purpose method for image warping, let alone image enlargement, does not appear to have been found.

Many approaches have been proposed to better interpolate edges. Often, but not always [19, 22], edge-enhancing methods have two main stages: First, a nonlinear interpolation scheme is used to create a double-density or dual version of the original image (this stage is often split into additional steps [14] or performed multiple times [13]). Then, this dou-

ble (or higher) density image is further resampled with a simpler "finishing" scheme [5, 6, 8, 9].

1.1 Original Contributions of This Article

We introduce a new resampling method which uses such a two-stage approach. The Nohalo method, so named because it does not add "halo" artifacts, is local, interpolatory, co-monotone, co-convex, local average preserving, continuous, and exact on linears (with suitable boundary conditions/abyss policy). Nohalo is weakly antialiasing, being the simplest member of a family of resamplers which double the input image more than once, smooth the input image prior to the initial subdivision, use a more sophisticated terminal resampling method, and/or have more sophisticated handling of high frequency modes. Nohalo's simplicity, however, allows it to run almost as fast as a standard bicubic resampler.

Nohalo makes heavy use of the minmod function. Formulas and code which allow its efficient computation are given. In addition, arithmetic branching tricks relevant to the computation of Nohalo and like subdividing resampling methods are described.

Demand-driven CPU/SMP and GPU implementations of Nohalo are described and benchmarked, and related programming issues are discussed.

The accuracy of Nohalo as an image reconstructor is quantitatively compared to one nonlinear and twenty-two linear alternatives with a new version of a test suite which involves the re-enlargement of images downsampled with nearest neighbour (box filtering was used in [18]). The results suggest that Nohalo level 1 is a high quality method. They also suggest that natural boundary conditions are generally preferable to not-a-knot boundary conditions.

1.2 Outline

Nohalo (level 1) is defined in §2. In addition, efficient implementations of the minmod function are given, arithmetic branching tricks used to speed up the code are explained, and appropriate boundary conditions/abyss policies are discussed. In §3–4, the main properties of Nohalo are stated, with particular attention paid to the built-in antialiasing. §5 shows an enlargement performed with Nohalo and two other high quality methods. §6 contains the results of an extensive comparative quantitative test suite and a discussion of its shortcomings. §7–9 detail the CPU, SMP and GPU implementations of Nohalo, implementations which are benchmarked in §10. The impact of image content on runtime is also discussed in §10. General conclusions are drawn in §11.

2. DESCRIPTION OF THE NOHALO RESAMPLING METHOD

Nohalo is very simple.

2.1 "Corner" Image Size Convention

Although Nohalo can be used with other conventions—for example, it can be made to function like an exact area method—it best fits the "corner" image size and geometry convention. In the corner image size convention, an $n \times m$ pixel image has surface $n-1 \times m-1$ (assuming unit interpixel distance) and, when resizing images, the (centers of the) resampled image's corner pixels are best understood as being aligned with those of the input image.

2.2 Resampling by Interpolation

Assuming that the input pixels are centered at integer coordinates, the interpolation problem can be stated as follows: Given an $m \times n$ image with pixel values $p_{j,i}$, construct a surface $f(x,y)$ such that

$$f(j,i) = p_{j,i} \quad (i \in \{0,1,\ldots,n-1\}, j \in \{0,1,\ldots,m-1\}). \tag{1}$$

Given a point transformation $\phi : \mathbb{R}^2 \to \mathbb{R}^2$, the pixel value at a point $(X,Y) = \phi(x,y)$ of the transformed image is given by $f(x,y)$. Thus, the resampled image is completely defined by the reconstructed intensity surface $f(x,y)$. For example, when resizing the entire input image to width $M-1$ and height $N-1$, the pixel value corresponding to the $J+1$st pixel of the $I+1$st row (indexing starts at 0) of the output image is

$$P_{J,I} = f\left(\frac{m-1}{M-1}J, \frac{n-1}{N-1}I\right).$$

2.3 Three-Stage Description of Nohalo

The construction of the interpolating surface $f(x,y)$ is best described as a three stage process:

1. Nonlinear gradient computation;
2. Construction of a double-density version of the original image;
3. Further interpolation of the double-density image with bilinear.

2.4 Nonlinear Gradient Computation

To every input pixel location (j,i), we associate a plane which interpolates the corresponding pixel value:

$$f_{j,i}(x,y) = p_{j,i} + s^x_{j,i}(x-j) + s^y_{j,i}(y-i).$$

Because $p_{j,i}$ is a given input pixel value, each such plane is fully determined by its gradient $(s^x_{j,i}, s^y_{j,i})$.

Ignoring boundary issues for now, consider the left and right differences

$$s^{x-}_{j,i} = p_{j,i} - p_{j-1,i} \text{ and } s^{x+}_{j,i} = p_{j+1,i} - p_{j,i}. \tag{2}$$

In terms of these differences, the horizontal slope $s^x_{j,i}$ is defined as follows: If $s^{x-}_{j,i}$ and $s^{x+}_{j,i}$ have the same sign, $s^x_{j,i}$ is the smaller of the two (in absolute value); otherwise, $s^x_{j,i} = 0$. This choice is motivated as follows: If the left and right slopes have different signs, the pixel value under consideration is a local minimum or maximum along this row of pixels and it makes sense to set the corresponding slope to zero. If, on the other hand, the two slopes have the same sign, taking the smallest one recovers the slopes of affine functions without error while minimizing oscillations. Using an analogous defininion for the top and bottom differences,

$$s^{y-}_{j,i} = p_{j,i} - p_{j,i-1} \text{ and } s^{y+}_{j,i} = p_{j,i_1} - p_{j,i}, \tag{3}$$

we thus set

$$s^x_{j,i} = \text{minmod}(s^{x-}_{j,i}, s^{x+}_{j,i}) \text{ and } s^y_{j,i} = \text{minmod}(s^{y-}_{j,i}, s^{y+}_{j,i}). \tag{4}$$

2.5 Programming the Minmod Function

Because minmod does not appear in Hacker's Delight [21] and like compendia, we provide formulas for it.

$$\text{minmod}(a,b) = \frac{1}{2}\{\text{sign}(a) + \text{sign}(b)\} \min(a\,\text{sign}(a), b\,\text{sign}(b))$$

where $\text{sign}(x) = 1$ if $x > 0$ and $\text{sign}(x) = -1$ if $x < 0$. The value of $\text{sign}(0)$ is irrelevant because the minimum vanishes when it occurs. In our C/C++ code, one conditional move is used to compute each sign as well as each minimum. Because signs can be recycled, only ten conditional moves are needed to compute the four minmods involved in the computation of a (single channel) pixel value.

The HLSL compiler implements the sign function with two conditional moves (so as to ensure that $\text{sign}(0) = 0$). For this reason, we use absolute values and the Heaviside step function $\text{H}(x)$ ((x≥0) in C, step(0,x) in HLSL and OpenGL) in our GPU implementation (Listing 1), for a total of three conditional moves per minmod computation:

$$\text{minmod}(a, b) = \{\text{H}(a) + \text{H}(b) - 1\} \min(|a|, |b|).$$

(The authors have recently discovered better formulas.)

2.6 Computation of the Double-Density Version of the Input Image

The interpolation condition (1) sets the value of $f(j, i)$ for i in $[0, n-1]$ and j in $[0, m-1]$. The pixel values which are missing in order to double the pixel density of the input image are consequently $f(x, y)$ for

$$(x,y) = \begin{cases} \left(j+\frac{1}{2}, i\right) & \text{for } i \in \{0,\dots,n-1\} \text{ and } j \in \{0,1,\dots,m-2\} \\ & \text{(horizontal halfway points),} \\ \left(j, i+\frac{1}{2}\right) & \text{for } i \in \{0,\dots,n-2\} \text{ and } j \in \{0,1,\dots,m-1\} \\ & \text{(vertical halfway points), and} \\ \left(j+\frac{1}{2}, i+\frac{1}{2}\right) & \text{for } i \in \{0,\dots,n-2\} \text{ and } j \in \{0,1,\dots,m-2\} \\ & \text{(diagonal halfway points).} \end{cases}$$

Such points are halfway between two consecutive input pixel locations in the horizontal or vertical direction, or they are located at the average of four nearby input pixel locations. This suggests averaging the values given by the approximating planes $f_{j,i}(x, y)$:

$$f(j+\frac{1}{2}, i) = \frac{1}{2}\left\{f_{j,i}(j+\frac{1}{2}, i) + f_{j+1,i}(j+\frac{1}{2}, i)\right\} \quad (5)$$
$$= \frac{1}{2}\{p_{j,i} + p_{j+1,i}\} + \frac{1}{4}\left\{s^x_{j,i} - s^x_{j+1,i}\right\};$$
$$f(j, i+\frac{1}{2}) = \frac{1}{2}\{p_{j,i} + p_{j,i+1}\} + \frac{1}{4}\left\{s^y_{j,i} - s^y_{j,i+1}\right\}; \quad (6)$$
$$f(j+\frac{1}{2}, i+\frac{1}{2}) = \frac{1}{4}\left\{\begin{array}{l} f_{j,i}(j+1/2, i+1/2) \\ +f_{j+1,i}(j+1/2, i+1/2) \\ +f_{j,i+1}(j+1/2, i+1/2) \\ +f_{j+1,i+1}(j+1/2, i+1/2) \end{array}\right\} \quad (7)$$
$$= \frac{1}{4}\{p_{j,i} + p_{j+1,i} + p_{j,i+1} + p_{j+1,i+1}\}$$
$$+ \frac{1}{8}\left\{s^x_{j,i} - s^x_{j+1,i} + s^x_{j,i+1} - s^x_{j+1,i+1}\right\}$$
$$+ \frac{1}{8}\left\{s^y_{j,i} + s^y_{j+1,i} - s^y_{j,i+1} - s^y_{j+1,i+1}\right\}$$
$$= \frac{1}{2}\left\{f(j+\frac{1}{2}, i) + f(j, i+\frac{1}{2})\right\} + \frac{1}{4}\{p_{j+1,i+1} - p_{j,i}\}$$
$$+ \frac{1}{8}\left\{s^x_{j,i+1} - s^x_{j+1,i+1} + s^y_{j+1,i} - s^y_{j+1,i+1}\right\}.$$

This defines the double-density version of the image.

Listing 1: GPU pixel shader for gradient calculation.

```
uniform sampler sImage;
uniform float2  vStepX, vStepY;
void main(    float2 vTex : TEXCOORD0,
          out float4 sx   : COLOR0,
          out float4 sy   : COLOR1)          {
float4 mid = tex2D(sImage, vTex);
float4 lef = mid - tex2D(sImage, vTex-vStepX);
float4 rit = tex2D(sImage, vTex+vStepX) - mid;
float4 top = mid - tex2D(sImage, vTex-vStepY);
float4 bot = tex2D(sImage, vTex+vStepY) - mid;
float4 hsize = min( abs(lef), abs(rit) );
float4 vsize = min( abs(top), abs(bot) );
float4 hswitch = step(0,lef) + step(0,rit);
float4 vswitch = step(0,top) + step(0,bot);
sx = hsize * hswitch - hsize + 128/255.;
sy = vsize * vswitch - vsize + 128/255.;      }
```

2.7 Arithmetic Branching on the GPU

In our GPU implementation, the double-density image is explicitly computed, and stored in a texture.

The value of a double-density image pixel depends on whether it is an input pixel, horizontal halfway pixel, vertical halfway pixel, or "diagonal" average of four input pixel locations. In order to avoid conditional execution, we use arithmetic branching.

Let $\hat{x} = x - j$ and $\hat{y} = y - i$. Because $\hat{x}$ and $\hat{y}$ are equal to 0 or 1/2 at the relevant pixel locations,

$$\tilde{f}(x,y) = (1-\hat{y})\{(1-\hat{x})f_{j,i}(x,y) + \hat{x}f_{j+1,i}(x,y)\} + \hat{y}\{(1-\hat{x})f_{j,i+1}(x,y) + \hat{x}f_{j+1,i+1}(x,y)\} \quad (8)$$

satisfies

$$\tilde{f}(j,i) = f(j,i),\ \tilde{f}(j+1/2,i) = f(j+1/2,i),$$
$$\tilde{f}(j,i+1/2) = f(j,i+1/2), \text{ and}$$
$$\tilde{f}(j+1/2,i+1/2) = f(j+1/2,i+1/2),$$

provided Eq. (1) and Eq. (5)–(7) are satisfied. One single branch-free formula thus covers all four cases.

$\tilde{f}$ has two important properties besides emulating f. The first is that $\tilde{f}$ is continuous, which implies that small errors in the computation of $\hat{x}$ and $\hat{y}$ lead to small errors in the double-density pixel values. The second is that $\tilde{f}$ is invariant under uniform shifts of the slope values. This matters because the gradients are stored in 8-bit textures in order to save memory, which is harmless because minmod maps 8-bit unsigned integers to integers in the range $[-127, 127]$. In order to store the slopes in standard unsigned 8-bit textures, the slopes are shifted by 128 (hence the "`128/255.`" terms in Listing 1). Without shift invariance, one would have to undo the gradient shifts before use; with it, no "`128/255.`" appears in Listing 2.

2.8 Final Interpolation with Bilinear

If the task at hand is resizing the $m \times n$ input image to $2m-1 \times 2n-1$, we are done. However, the point transformation ϕ is generally not a simple doubling. Put another way: Although we now "know" about four times as many f-values, we still have not fully specified the surface $f(x, y)$.

The reconstructed intensity surface $f(x, y)$ is simply obtained from the double-density image by bilinear interpolation.

Listing 2: GPU pixel shader for 2X upsizing.

```
uniform sampler sImage, sGradX, sGradY;
uniform float2  vStepX, vStepY, vSize;
float4 main(float2 vTex : TEXCOORD0) : COLOR {
float4 p_c  = tex2D(sImage,vTex);
float4 dx_c = tex2D(sGradX,vTex);
float4 dy_c = tex2D(sGradY,vTex);
float4 p_r  = tex2D(sImage,vTex+vStepX);
... /* Likewise with dx_r and dy_r */
float4 p_b  = tex2D(sImage,vTex+vStepY);
... /* Likewise with dx_b and dy_b */
float4 p_br  = tex2D(sImage,vTex+vStepX+vStepY);
... /* Likewise with dx_br and dy_br */
float2 coord = frac(vTex * vSize) - .25;
float4 f_c = p_c + dx_c*coord.x + dy_c*coord.y;
float4 f_r = p_r + dx_r*(coord.x-1)
           + dy_r*coord.y;
float4 f_b = p_b + dx_b*coord.x
           + dy_b*(coord.y-1);
float4 f_br = p_br + dx_br*(coord.x-1)
            + dy_br*(coord.y-1);
float4 top = lerp(f_c, f_r, coord.x);
float4 bot = lerp(f_b, f_br, coord.x);
/* The compiler adds one extra instruction if
   lerp is used below in the obvious way. */
return top*(1-coord.y) + bot*coord.y;        }
```

2.9 Nohalo Has a 12-Point Stencil

The value of the reconstructed intensity surface at any point depends on the values of (at most) 12 nearby input values. The reason for this is that a resampling point is in the convex hull of four contiguous double-density pixels: one input pixel, one horizontal halfway pixel, one vertical halfway pixel, and one "diagonal" pixel. Thus, there are four cases to consider, depending on whether the input pixel is top-left, top-right, bottom-left or bottom-right. Tracking down the pixels needed to compute slopes in each of these cases leads to the following: The stencil of Nohalo is the 4×4 stencil of standard bicubic methods, minus the four corners. That is, the stencil of Nohalo is a two pixel thick "fat +." (Such a stencil is also used in the ICBI method [12].)

2.10 Arithmetic Branching with Pointer Shifts

The weights involved in the computation of a pixel value depend on the position of the closest input pixel relative to the sampling point. In our demand-driven ("pull") implementations, the four cases (top-left, top-right, bottom-left and bottom-right) are treated as one by reflecting the 12-point stencil by adding suitable shifts to the pointer used to pull values from the relevant input image tile.

2.11 Nearest Neighbour Abyss Policy

If the resampling location is within one inter-pixel distance of the boundary, the differences used to compute slopes involve undefined pixel values. Consider, for example, the plane

$$f_{0,0}(x,y) = p_{0,0} + s^x_{0,0}x + s^y_{0,0}y,$$

which enters the computation of resampled pixel values associated with locations (x,y) in $[0,1) \times [0,1)$. Computing $s^x_{0,0}$ and $s^y_{0,0}$ with Eq. (2)–(3) involves two "out of picture" pixel values, namely $p_{-1,0}$ and $p_{0,-1}$. More generally, the computation of resampled values at locations which are within one pixel width of the boundary may involve $p_{j,-1}$ or $p_{j,m}$ for $j \in \{0,1,\ldots,n-1\}$, or $p_{-1,i}$ or $p_{n,i}$ for $i \in \{0,1,\ldots,m-1\}$. The nearest neighbour abyss policy sets the value of an "out of picture" pixel to the value of the closest "in picture" (and consequently boundary) pixel.

In VIPS, this is accomplished by virtually extending the input image by two rows and two columns all around. That is, the top row of the input image is triplicated; so is the bottom row and the leftmost and rightmost columns.

In DirectX, this is implemented by setting the texture pixel lookup behavior to "clamp." Clamping the coordinates of requested pixel coordinates to the valid range automatically makes a request for the pixel value associated with an "out of picture" location return the value of the closest boundary pixel.

2.12 Linear Extrapolation Abyss Policy

With the nearest neighbour abyss policy, constant input data is "seen" as constant by the resampler. Therefore, Nohalo is globally exact on constants. Affine data is unfortunately not "seen" as such by the resampler near the boundary when the nearest neighbour abyss policy is used. As a result, Nohalo is not exact on linears near the boundary.

Linear extrapolation can be used to preserve exactness on linears up to and through the boundary. Although the pixel values of the extended input image may overflow or underflow with linear extrapolation, monotonicity is still maintained. This implies that the surface $f(x,y)$ is bounded by the minimum and maximum input values within the extent of the input image. (Note: Because the "corner" pixel values of the enlarged image are not used by Nohalo in the "corner" image size convention, the ambiguity which arises as to whether rows or columns should be used to compute the values of extrapolated "corner" pixels is of no consequence.) This abyss policy has not been implemented.

3. PROPERTIES OF NOHALO

Nohalo is local, interpolatory, co-monotone, co-convex, local average preserving, continuous and exact on linears (except possibly at the boundary). These properties, with the exception of continuity which is not applicable, hold for the density doubling scheme. They also hold for bilinear resampling. This is why they hold for their combination.

3.1 Built-In Antialiasing

Ideally, if an image is constant along diagonals, the reconstructed intensity surface $f(x,y)$ should have the same property. Bilinear interpolation does not preserve diagonality; neither does Nohalo. The best one can hope for is for the double-density version of the image to be constant on diagonals if the input image is. This is the case to some extent.

Suppose that the input image is given by

$$p_{j,i} = \begin{cases} 0 & \text{if } j < i, \\ 1/2 & \text{if } j = i, \\ 1 & \text{if } j > i. \end{cases}$$

Then, the resulting double-density image is constant on diagonals as well, taking the values 0, 1/4, 1/2, 3/4 and 1 as the main diagonal is approached. The double-density image is also constant on diagonals if

$$p_{j,i} = \begin{cases} 0 & \text{if } |j-i| > 1, \\ 1/2 & \text{if } |j-i| = 1, \\ 1 & \text{if } j = i. \end{cases}$$

That is: Nohalo preserves "soft" diagonal interfaces and lines.

4. MAIN WEAKNESS OF NOHALO

In general, Nohalo is more suited for natural images than man-made ones. The reason for this is that, in some cases, Nohalo boils down to plain vanilla bilinear, so that the computed slopes are wasted and the visual quality is low.

If every pixel is either no less or no more than both of its immediate neighbours in the horizontal direction, and if, likewise, it is a local minimum or maximum when compared to its immediate neighbours in the vertical direction, then the slopes specified by Eq. (4) are all equal to zero. Therefore, the values computed by Eq. (5)–(7) are identical to those obtained with bilinear interpolation, and Nohalo is reduced to bilinear interpolation. This is the case when the input image is bichromatic, for example, for black on white text images.

5. SAMPLE 200% ENLARGEMENT

An archival high resolution scan of Ansel Adams' photoportrait of Tōyō Miyatake, part of the copyfree Manzanar series held at the Library of Congress, was downsampled with box filtering and cropped, yielding a low noise 64x64 pixel image which was then re-enlarged to 127x127 with Nohalo and two other high quality methods: monotone cubic splines (a Scilab [2] implementation of the method of Fritsch and Carlson [11] interfaced by the authors to SIVP [3]); and ICBI, used with all parameters set to default values, in particular, so that it iterates until convergence [12]. Crops of the resulting enlargements are shown in Figure 1.

6. ACCURACY: QUANTITATIVE IMAGE QUALITY COMPARATIVE TEST SUITE

A quantitative comparison of resampling methods based on the re-enlargement of downsampled images was performed.

6.1 Tested Resamplers

Twenty-two linear and two nonlinear resampling methods, listed in Table 1, were compared. All methods were used with default parameter values. Although tested, the ImageMagick filters based on quadratic and cubic approximations of the Gaussian curve will not be discussed further because they performed more poorly than plain vanilla Gaussian blur.

6.2 Consistent Image Alignment

The test suite is set up so that errors do not originate from image size convention mismatch. For example, ImageMagick 6.3.6_10 resize filters use the "center" image size convention (this is undocumented); for this reason, the authors modified the relevant ImageMagick source code (resize.c) to make it consistent with the "corner" convention.

6.3 Test Images

Nine copyfree colour and two greyscale images were used: one CG image of a living room (M. Gong), and ten digital photographs/scans of small objects (J.-F. Avon), astronauts and spacecraft (NASA), a woodcut print of a wave and boat (K. Hokusai), a chapel (M. Ryckaert), a katydid (wikipedia

Figure 1: 103x89 crops of enlargements of a low-noise 64x64 pixel image to 127x127: monotone bicubic splines (top), Nohalo (middle) and ICBI.

Table 1: Compared resampling methods

method	description (implementation)
Bessel	Bessel windowed Jinc (ImageMagick)
Bicubic	bicubic Lagrange interpolant (ImageMagick)
Bilinear	bilinear (VIPS)
Blackman	Blackman windowed Sinc (ImageMagick)
BoxFilter	Exact area box filtering (by the authors)
CatRom	Catmull-Rom (ImageMagick)
Gaussian	Gaussian blur (ImageMagick)
Hamming	Hamming windowed Sinc (ImageMagick)
Hann	Hann windowed Sinc (ImageMagick)
Hermite	Hermite with $\nabla f(j,i) = 0$ (ImageMagick)
Kaiser	Kaiser windowed Sinc (ImageMagick)
Lanczos3	Lanczos (3-lobes) (ImageMagick)
Mitchell	Mitchell-Netravali bicubic (ImageMagick)
Monotone	monotone bicubic spline (Scilab/SIVP)
NaKSplin	not-a-knot bicubic spline (Scilab/SIVP)
NatSplin	natural bicubic spline (Scilab/SIVP)
Nearest	nearest neighbour (ImageMagick)
Nohalo	Nohalo (VIPS)
Parzen	Parzen windowed Sinc (ImageMagick)
Welsh	Welsh windowed Sinc (ImageMagick)

user wadems), a seated man in full regalia (S. Prokudin-Gorskii), a vervet in a tree (W. Welles), as well as close ups of a baby (M. Gong) and a man (A. Adams).

First, each image was cropped to 1681×1681. The cropped images were then downsampled with nearest neighbour—that is, decimated—to 841×841, 561×561, 421×421, 337×337, 281×281, 241×241, 211×211, 169×169, 141×141, 121×121, 113×113 and 106×106. For example, the 841×841 downsamples were created by keeping every other pixel of every other row (keeping the top left pixel). Although one could argue that decimation indirectly amplifies the noise and artifacts present in the full size images, decimation does not, by itself, introduce error. For this reason, the downsampled versions of the cropped originals were treated as if error free in the rational re-enlargement tests.

6.4 Description of the Upsampling Tasks

Eighteen resampling tasks were performed with each resampling method on each of the eleven test images. For the twelve integer magnification tests, the images were enlarged back to the original 1681×1681. For the six rational magnification tests, images were enlarged to the next larger size. For example, 3/2 magnification was tested by enlarging 561×561 images to 841×841. The re-enlargements were then compared to the cropped originals (integer magnifications) or their downsampled versions (rational magnifications), and statistics reported in groups of six enlargement ratios: small fractional (Table 2), small integer (Table 3) and large integer magnifications (Table 4).

6.5 Error Metrics

Four error metrics were used: Root Mean Squared Error (RMSE), Average Absolute Error (AAE), Maximum Absolute Error (MAE), and Mean Structural SIMililarity index (MSSIM). MSSIM is analogous to a correlation in that larger MSSIMs correspond to smaller errors [20]. Statistics are amalgamated as follows: the RMSEs by taking the square root of the mean of their squares, the AAEs, MAEs and MSSIMs by plain averaging.

Table 2: Aggregate test results for the magnification factors 8/7, 7/6, 6/5, 5/4, 4/3 and 3/2

	RMSE	AAE	MAE	MSSIM
Nohalo	12.4133	5.3978	162.1	.862019
Mitchell	12.5117	5.6129	162.3	.859667
Bilinear	12.5276	5.5294	161.7	.860294
Monotone	12.5529	5.3625	163.7	.862043
Bessel	12.5818	5.7338	162.2	.856801
Bicubic	12.6162	5.5849	164.9	.860653
Gaussian	12.6903	5.8472	160.8	.853527
CatRom	12.7013	5.5857	165.7	.860482
BoxFilter	12.7120	5.5284	164.2	.858633
Parzen	12.9260	5.7497	168.2	.857633
Blackman	13.0069	5.8182	169.1	.856252
NatSplin	13.0310	5.8130	169.0	.855727
NaKSplin	13.0353	5.8155	169.0	.855669
Kaiser	13.1126	5.9045	170.3	.854365
Lanczos3	13.1548	5.9305	170.8	.853992
Hann	13.1587	5.9442	170.7	.853449
Hamming	13.1950	5.9747	171.0	.852728
Welsh	13.3994	6.1450	172.8	.848545
Hermite	13.8794	5.8289	180.1	.847375
Nearest	21.1768	9.1685	212.5	.725917

6.6 Discussion of the Results

Nohalo consistently obtained the best overall RMSE, the second best AAE, the second best MSSIM, and one of the best MAEs. Other methods performed well, most notably the only other tested nonlinear resampler, Fritsch and Carlson's monotone cubic spline method [11], which scored the lowest AAEs and highest MSSIMs. The mildly smoothing bicubic method of Mitchell-Netravali [16] also performed well. So did bilinear.

6.7 Side Note About Boundary Conditions for Global Cubic Splines

Not-a-knot boundary conditions for cubic splines—which make interpolation exact on linears, and consequently second order accurate—are generally prefered to natural boundary conditions—which satisfy a stronger variational principle but are only first order accurate near the boundary. In both this article's comparative test suite and the variant found in [18], natural boundary conditions, against conventional wisdom, achieve better results. In our opinion, this is because images are generally not smooth enough for the usual error bounds to be fully applicable.

6.8 Shortcomings of the Test Suite

The main shortcoming of the test suite is that downsampling/reconstruction test suites measure the accuracy of a resampler as a reconstructor, with only indirect bearing on resampling accuracy. That is: What such tests really address is how close Nohalo and the other resamplers come to being left inverses of the projection operator defined by the chosen downsampling method. For example, the authors would expect Nohalo to lose its top ranking if downsampling was performed with box filtering instead of decimation. The authors hope to address this shortcoming in the future with carefully designed image rotation tests.

Table 3: Aggregate test results for the magnification factors 2, 3, 4, 5, 6 and 7

	RMSE	AAE	MAE	MSSIM
Nohalo	11.0642	4.6603	169.7	.841853
Mitchell	11.1710	4.8979	167.9	.837422
Monotone	11.1744	4.6348	170.8	.842780
Bilinear	11.1883	4.7889	169.3	.839327
Bicubic	11.2436	4.8161	172.3	.838083
Bessel	11.2499	5.0196	166.9	.833130
CatRom	11.3090	4.8158	173.1	.838494
Gaussian	11.3767	5.1412	165.9	.831173
Parzen	11.5040	4.9490	175.6	.834943
Blackman	11.5754	5.0052	176.3	.832974
NatSplin	11.5973	4.9985	176.5	.831978
NaKSplin	11.6044	5.0034	176.7	.831850
Kaiser	11.6695	5.0768	177.2	.830289
Lanczos3	11.7090	5.1006	177.5	.829520
Hann	11.7111	5.1103	177.5	.828996
Hamming	11.7428	5.1354	177.7	.827914
Welsh	11.9273	5.2800	179.0	.821748
Hermite	12.4025	5.0585	187.1	.822265
BoxFilter	14.3752	5.8072	199.3	.792552
Nearest	19.2097	8.2889	216.7	.725886

Table 4: Aggregate test results for the magnification factors 8, 10, 12, 13, 14, 15 and 16

	RMSE	AAE	MAE	MSSIM
Nohalo	18.9989	9.0238	209.8	.734328
Bilinear	19.0126	9.1451	208.8	.732630
Mitchell	19.0242	9.2099	209.9	.731337
Bessel	19.0833	9.3927	209.3	.729001
Gaussian	19.0955	9.4933	207.0	.729239
Monotone	19.2584	8.9184	211.4	.735694
Bicubic	19.2622	9.2554	213.5	.728511
CatRom	19.4490	9.2483	214.6	.728565
Parzen	19.8338	9.5489	217.3	.724724
Blackman	19.9557	9.6677	218.0	.722551
NatSplin	20.0005	9.7106	218.3	.721542
NaKSplin	20.0349	9.7374	219.5	.721190
Kaiser	20.1110	9.8143	219.0	.719744
Hann	20.1785	9.8817	219.3	.718445
Lanczos3	20.1805	9.8499	219.3	.718578
Hamming	20.2280	9.9296	219.5	.717322
Welsh	20.5106	10.2085	220.9	.711307
Hermite	21.0558	9.5137	222.3	.707548
BoxFilter	25.1472	11.2354	229.9	.668163
Nearest	31.0015	14.5384	232.5	.639862

Downsampling by decimation generally increases the local variance of pixel values. Put another way, images downsampled with nearest neighbour not only are less detailed but also considerably less smooth. Consequently, the present rankings may be more relevant for noisy or destructively compressed images than high quality digital photographs.

The rankings obtained in an earlier version of the test suite, in which downsampling was performed with (exact area) box filtering instead of nearest neighbour, were very different [18]. For example, Bessel (windowed Jinc), which ranks about fifth in the present test suite, performed so poorly in the box filtered version that it was kept off the charts. The reason for this discrepancy is that box filtering, unlike decimation, is a smoothing operator. Consequently, the downsampled images used in [18] were smoother than typical digital photographs and scans, which allowed windowed sinc methods to shine. When images are noisy, monotone methods (Nohalo, monotone cubic splines, bilinear, exact area box filtering and nearest neighbour) and smoothing methods (Bessell and Mitchell-Netravali) are generally at an advantage, because they do not amplify noise.

Another shortcoming of the test suite is that upsampling was performed without gamma correction. That is, interpolation was performed using pixel values without regard to colour profiles. (When downsampling with decimation, colour profiles are irrelevant. However, in the case of box filtering test suites as found in [18], colour profiles should also be taken into account in the averaging performed in the course of box filtering.) This favours monotone and smoothing methods, because they do not amplify overshoots and undershoots caused by an invalid linear interpretation of pixel values.

Perceptually, not all errors and resampling artifacts are created equal. A failing of this test suite is that error measures are only loosely correlated with "perceptual accuracy." Although MSSIM attempts to bridge the gap between the quantitative and the subjective, the only way to measure subjective quality is to involve people.

A wider variety of nonlinear resampling methods, edge-enhancing or not, should be included. Finally, the correctness of the ImageMagick code should be verified.

7. CPU IMPLEMENTATION (GEGL)

GEGL, the GEneric Graphics Library, primarily designed for interactive use, is the future engine of GIMP, the GNU Image Manipulation Program, a widely used Photoshop alternative developed by a large distributed group of programmers. Free and open source, GEGL runs on most operating systems. Written in C, GEGL will bring non-destructive editing, high dynamic range, and improved handling of very large images to the next major release of GIMP (GNU Image Manipulation Program). Nohalo, under the name "sharp," is an integral part of GEGL, and its source code (gegl-sampler-sharp.c) can be downloaded from gegl.org [1].

GEGL is "image type agnostic," relying on an external library to convert, on demand, image pixel data to and from linear RGBA float buffers. This greatly simplifies the code base, since a uniform data type is seen by methods. (This is expected to change in the future.)

Intended for real time, interactive use by graphic artists and others, GEGL is structured so that the task-defining DAGs are dynamic structures. For example, one can go back to any of the operations, change its parameters, and the changes are propagated automatically. This would be fairly simple to implement if the results of earlier stages were "pushed" onto later stages of the computation. Howver, like many recent image processing systems, GEGL is demand-driven, meaning that output pixel values are computed somewhat independently, each pixel "pulling" the needed input data through the DAG, the needed operations being performed along the way. With this data processing model, implementing a separate gradient stage for Nohalo is not really an option, because it would add an extra graph node

without the benefit of allowing for the recycling of slopes common to several output pixels.

Nohalo is implemented as a "single stage" method in GEGL. Basically, the method is passed the double precision coordinates of the sampling location within the input image and a pointer which is used to return computed values. A key aspect of the implementation is that GEGL fulfils requests for a pointer to a specific input pixel by embedding it within a buffer large enough to contain the stencil and with missing values set in accordance to the abyss policy. This buffer is constructed so that it can be reused, without change, in the computation of other pixel values.

8. EMBARRASSINGLY PARALLEL CPU/SMP IMPLEMENTATION (VIPS)

VIPS, the Virtual Image Processing System, was developed by a handful of programmers and is tuned for high performance batch processing. VIPS is a free and open source image processing library developed over several EU-funded research projects [15]. Written in C and C++, VIPS and its GUI Nip2 run on Unix, Windows and OS X. Nohalo is an integral part of VIPS, and its source code (nohalo.cpp) can be downloaded from [4] and SourceForge.

VIPS is demand-driven. A DAG of processing elements is built as operations are invoked. When the final operation connects to a data sink, which can be a disc file, a memory area, or a screen display, the sink pulls pixels through the pipeline one tile at a time. Because no image (initial, intermediate or final) is fully present in memory, VIPS usually needs little RAM and easily handles very large images. Once the pipeline has been built, there is little synchronisation between threads and almost no memory allocation. As the computation proceeds, work is distributed among the available processing elements, tile sizes are adjusted, data source resources are shared, common regions of pixels are reused, and so on.

8.1 Improving Scalability

VIPS was originally designed in the mid-1990s on machines with two processors and occasionally tested on a six-processor machine. In 2005, a 64-processor supercomputer became available for testing as part of the development of the PARSEC benchmark [7]. To our surprise, VIPS topped out at about a 10x speed up, even with all 64 processors enabled.

Following a lot of profiling, several mechanisms were added to VIPS to improve scalability. First, we added a system for recycling and sharing pixel buffers, almost eliminating memory allocation during computation. We also almost eliminated thread synchronisation in intermediate steps, a minimal amount of thread synchronisation remaining in the handling of sources and sinks. Finally, we decoupled the file sink from the output imaging library so that worker threads keep processing tiles when groups of scanlines are written to disc. As a result of these changes, the VIPS component of the benchmark now scales linearly to more than 32 CPUs and reaches about a 40x speed up with 64 processors. (Access to the supercomputer unfortunately ended before we could find out what was preventing VIPS from achieving perfect scalability across the board.)

The PARSEC benchmark predates the addition of sophisticated resamplers to VIPS.

8.2 Implementing Resamplers

Adding a new resampler to VIPS requires subclassing VipsInterpolator, setting the size of a rectangular input pixel window so that it is large enough to contain the stencil of the interpolator, and implementing the interpolate method.

Although GEGL and VIPS are fairly similar, they are also quite different, owing to VIPS' batch processing bias. One distinguishing feature is that although VIPS' task-defining DAGs are fully configurable, they are essentially static, in the sense that changing any part of the overall task requires destroying and rebuilding subsequent parts of the DAG. Another is that VIPS implements boundary conditions by extending the input image instead of implementing an abyss policy at the tile level. Yet another difference is that different image data types (8 to 32 bit signed/unsigned integers, single or double precision real or complex floating point numbers) are handled differently by the resamplers. Roughly speaking, the DAGs which define processing tasks are implemented in a type polymorphic way. Yet, every type gets custom treatment under the hood. For example, VIPS bilinear and bicubic resamplers use fixed point arithmetic and table lookups to compute the values of the cardinal basis functions for small integer data types, which allows the computation to bypass the FPU and leads to substantial speed gains on older hardware.

Falling short of this level of detail, the VIPS implementation of Nohalo has three stages: the first stage sets pointer shifts according to the input pixel type; the second stage implicitly performs the image doubling in double precision, the same source code handling this task for all data types through the use of parameterized macros; and the third stage performs the final bilinear interpolation and downcasts using rounding methods matched to individual data types.

9. GPU IMPLEMENTATION (DIRECTX)

The GPUs built into modern graphics cards allow computational kernels to run on multiple data in parallel. GPUs are designed for 3D graphics applications in which the computational kernels are used to calculate the transformation and lighting of each vertex (vertex shaders) or to compute the shading of each rasterized pixel (pixel shaders). Other tasks are often cast as processes with one or more rendering passes, each involving the following sequence of operations:

1. Represent the input data as a collection of 2D or 3D arrays to be loaded into the video memory as textures;
2. Load the algorithm into the GPU as a pixel shader;
3. Set either the screen or a pixel buffer in video memory as the rendering target; and
4. Execute the shader by rendering an image-sized rectangle.

GPU implementations straddle the boundary between demand-driven ("pull") and data-driven ("push"). The reason for this is that, although the pixel or vertex texture values are computed by essentially independent "pull" threads, global results can be stored in textures which are "pushed" toward later stages of the computation. Two groups of intermediate textures are used in our three-stage GPU implementation: two textures which store gradient values (the slopes computed with Eq. (2)–(4)), and a texture which holds the double-density image (given by Eq. (8)).

The first stage loads the input image as a colour texture, calculates the horizontal and vertical slopes using a pixel shader, and stores them into two colour textures. The source code for this stage is shown in Listing 1; the inputs are sImage, the input colour image, and vStepX and vStepY, the distance between adjacent pixels in texture coordinates. The resulting gradient textures are illustrated at the top of Figure 2: the horizontal slopes are on the left, the vertical slopes are on the right.

Figure 2: 64x64 pixel crops of the horizontal (top left) and vertical (top right) gradient textures (grey means 0) produced in the first stage (Listing 1), and corresponding 128x128 crop of the double-density image produced in the second stage (Listing 2).

The second stage uses the input image and the two gradient textures to interpolate the input image at the double-density pixel locations, yielding a new image with $2m-1 \times 2n-1$ pixels which is, again, stored as a texture. The corresponding pixel shader is shown in Listing 2; the input textures are sImage and sGradX (horizontal slopes) and sGradY (vertical slopes); vSize holds the dimensions of the input image. An example of double-density image produced by this shader is shown at the bottom of Figure 2.

The third and final stage takes the double-density image as input texture and scales it to the user specified resolution using bilinear interpolation. Because GPUs have bilinear interpolation built-in, this stage does not require a shader.

Nohalo was implemented on the GPU with Visual C++ 2005 and Direct3D 9.0. The code is lean: The Microsoft HLSL Shader Compiler fills 25 instruction slots for the gradient computation (Listing 1), and 29 for the 2X upsizing (Listing 2).

10. PERFORMANCE

10.1 Laptop Image Resizing and Rotation Benchmarks (GEGL and VIPS)

The GEGL implementation of Nohalo performs resampling tasks faster than GEGL bicubic and barely slower than GEGL bilinear. This unduly flattering picture of the performance of Nohalo arises from the fact that GEGL has a relatively high run-time overhead. (A Google Summer of Code 2009 project, *Performance tools and study of GEGL*, will be the first step in identifying bottlenecks.) In addition, GEGL bicubic, which implements the full two-parameter family of cubic splines, is not optimised for speed. (One of the authors helped tune GEGL bilinear, however.)

The following VIPS benchmarks level the playing field. Bilinear, Catmull-Rom and Nohalo were programmed for VIPS by two of the authors. The benchmarks were performed on 32-bit float images so that the integer optimisation tricks built into the VIPS versions of bilinear and Catmull-Rom and the careful rounding built into Nohalo not come into play. All tests were performed with two image formats: uncompressed tiled (128x128) TIFF, and VIPS, a scanline image format analogous to PPM. (Random access is not efficiently implemented in LibTIFF for scanline images, hence the decision to use the VIPS scanline format.)

Table 5: Resampling 2500x2500 RGB float natural images on a dual-core laptop: total run times (in seconds) using 1/2 cores for VIPS scanline (.v) and TIFF tiled (.tif) formats

method (format)	downsample	rotate	upsample
bilinear (.v)	0.22/0.17	0.76/0.57	3.87/3.68
bilinear (.tif)	0.36/0.31	0.90/0.70	3.84/3.91
Catmull-Rom (.v)	0.31/0.21	1.13/0.73	3.96/3.67
Catmull-Rom (.tif)	0.45/0.36	1.31/0.87	4.08/3.87
Nohalo (.v)	0.77/0.45	3.06/1.71	7.87/4.31
Nohalo (.tif)	0.90/0.59	3.11/1.81	8.12/4.61

In the first set of benchmarks, three tasks were performed on 2500x2500 RGB float (72MB) images with a dual-core Intel Core 2 2Ghz, 2GB RAM, laptop running 32-bit Ubuntu 8.10: resizing (downsampling) to 1148x1148, rotating by 5 degrees about the top-left corner (maintaining the image size), and resizing (upsampling) to 3924x3924 (177MB). As seen in Table 5, Nohalo is about three times slower than bilinear when few output pixels are computed per input pixel (downsampling and rotating), and nearly as fast when upsampling, in which case I/O is significant and on-chip branch prediction more successful. Most of the run time is actuallly spent outside of the resampling code when upsampling. For example, out of the 3.87s total run time reported in Table 5 for bilinear upsampling, only 1.92s is "user" time (according to the time command).

10.2 Speed of Execution Depends on Image Content

In the C and C++ Nohalo code, signs and minima are computed in order to compute the minmods of pairs of slopes as well as the pointer shifts needed to reflect the stencil. Because signs and minima are computed with flags and conditional moves in our GEGL and VIPS implementations, run times depend on whether branch prediction successfully guesses the signs of slopes as well as the minima of their absolute values. For this reason, Nohalo runs faster on smooth images. Another consequence is that Nohalo's performance relative to bilinear and bicubic is better for upsampling than downsampling because the same signs and minima are computed more than once when several output pixels are computed with the same stencil, hence they are easier to "predict."

In order to illustrate the dependence of run-time on image content, we re-ran the benchmarks on maximally smooth images, namely monochrome images. As seen in Table 6, the run times for Nohalo were significantly reduced. (The run times for bilinear and Catmull-Rom were basically unchanged.) Implementing minmod with bit twiddling would speed up Nohalo considerably.

Table 6: Resampling (constant) black 2500x2500 RGB float images on a dual-core laptop: total run times using 1/2 cores for VIPS and tiled TIFF formats

method (format)	downsample	rotate	upsample
Nohalo (.v)	0.51/0.31	2.08/1.20	5.24/3.93
Nohalo (.tif)	0.76/0.56	2.26/1.44	5.45/4.44

10.3 8-Core Desktop Image Resizing and Rotation Benchmarks (VIPS)

In this set of benchmarks, a desktop computer with two quad-core Intel Xeon 1.86GHz CPUs with 8GB RAM total performed analogous tasks on 5000x5000 RGB float images: downsampling to 2296x2296, rotating by 5 degrees about the top-left corner, and upsampling to 7848x7848.

The results are shown in Figure 3. Unsurprisingly, the best scalability is seen when going from one to two cores, and when using Nohalo, which performs many operations on a stencil smaller than Catmull-Rom's, hence has the best arithmetic to I/O ratio. For example, rotating the scanline (.v) image with Nohalo takes 16.57s with one core and 8.27s with two. Such perfect scalability is not seen across the board: The tiled TIFF multi-core results, in particular, suffer greatly from the fact that LibTIFF is not thread-friendly.

10.4 GPU Video Enlargement Benchmarks (DirectX)

The processing speed of the DirectX implementation of Nohalo was evaluated by upsizing DVD video to HDTV video.

The enlargement ratios involved in TV to HDTV conversion, being approximately equal to two, are ideally suited for Nohalo. The reason for this is that the final bilinear stage of Nohalo resamples the double density image to approximately the same density, so that the lack of derivative continuity of the reconstructed intensity surface is not as apparent as it would be with higher upsampling ratios. Indeed,

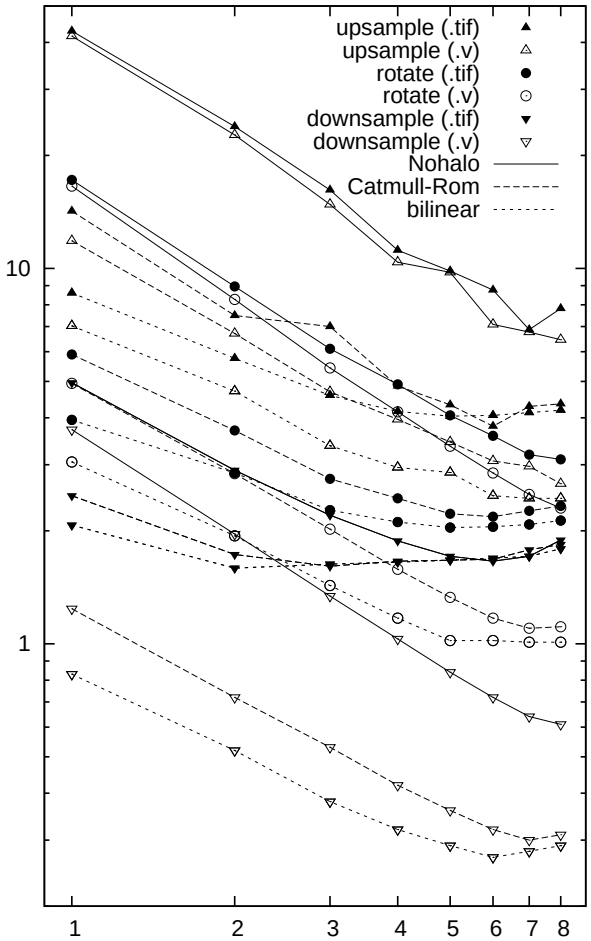

Figure 3: Resampling 5000x5000 float RGB natural images on a twin quad-core desktop: total run time in seconds (vertical axis) versus number of cores (horizontal axis).

videos resampled with Nohalo are noticeably smoother and more visually pleasing than those resampled with bilinear.

DVD video (720x480) was enlarged to HDTV (cropped to 1620x1080 to maintain the aspect ratio) directly with the vendor's hardware bilinear upsampler (with no Nohalo involvement whatsoever), and with Nohalo (which involves the vendor's hardware bilinear in its final stage). To accurately measure the enlargement processing speed, video decoding time is excluded; this is done by repeatedly enlarging a single DVD video frame.

Because the implementation is branch-free, the GPU's SIMD architecture yields identical frame rates for colour and grayscale video. The frame rates obtained with the two approaches are shown in Table 7. On all platforms, Nohalo achieves at least 30 fps, a frame rate suitable for streaming video. ([13] reports high frame rates for local power-of-two zooming with an alternative edge-enhancing resampling method.) The performance of Nohalo relative to hardware bilinear is even better when DVD video is enlarged to full HD without cropping (1920x1080); this is not surprising given that the fixed cost of computing the double-density image is spread over more pixels.

11. CONCLUSIONS

Table 7: GPU benchmark results: Resizing DVD video to cropped HDTV on consumer hardware

GPU	VRAM	Nohalo	hardware bilinear	ratio
NVIDIA GeForce 6800	256MB	30 fps	36 fps	.83
NVIDIA GeForce 8600 GT	256MB	33 fps	36 fps	.92
ATI Mobility Radeon X1400	512MB	42 fps	349 fps	.12
NVIDIA GeForce 9800 GT	512MB	575 fps	1985 fps	.29
NVIDIA GeForce 9800 GX2	1024MB	606 fps	1694 fps	.36

Nohalo is a general purpose resampling method which runs fast on a variety of platforms. Accurate, co-convex and mildly antialiasing, Nohalo produces pleasant enlargements of natural images without noticeable nonlinear artifacts.

Provided the computation is structured in a branch-free way, and the impact of low precision fixed point intermediate result storage is minimal, the architecture and instruction set of GPUs is ideal for local and monotone resamplers which rely on subdivision like Nohalo.

In contrast, high performance implementations of resamplers in demand-driven mode on the CPU require careful and intricate programming, and obtaining good scalability on SMPs requires identifying and fixing bottlenecks all along the powertrain, from I/O buffering to tile handling to using symmetry to minimise branching to scheduling flag computations ahead of conditional moves etc.

12. ACKNOWLEDGMENTS

We are especially grateful toward Geert Jordaens and Øyvind Kolås for their comments and code. We also thank Andrea Giachetti, David Gowers, Sven Neumann, Martin Nordholts, Xin Li, Alexey Lukin, Darren Janeczek and David Kelly. Research funding was provided by two Canada NSERC Discovery grants, one NSERC USRA grant, and one Ontario CFI New Opportunity grant.

13. REFERENCES

[1] GEGL. http://gegl.org.

[2] Scilab. http://www.scilab.org.

[3] SIVP–Scilab Image and Video Processing toolbox. http://sivp.sourceforge.net.

[4] VIPS. http://www.vips.ecs.soton.ac.uk.

[5] N. Asuni. iNEDI – tecnica adattativa per l'interpolazione di immagini. Master's thesis, Università degli Studi di Cagliari, 2007.

[6] S. Battiato, G. Gallo, and F. Stanco. A locally adaptive zooming algorithm for digital images. *Image and Vision Comput.*, 20:805–812, 2002.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[8] Y. Cha and S. Kim. The Error-Amended Sharp Edge (EASE) scheme for image zooming. *IEEE T. Image Process.*, 16(6):1496–1505, June 2007.

[9] M.-J. Chen, C.-H. Huang, and W.-L. Lee. A fast edge-oriented algorithm for image interpolation. *Image and Vision Comput.*, 23(9):791–798, 2005.

[10] N. A. Dodgson. Image resampling. Technical Report UCAM–CL–TR–261, University of Cambridge Computer Lab., 15 JJ Thomson Avenue, Cambridge CB3 0FD, UK, Aug. 1992.

[11] F. N. Fritsch and R. E. Carlson. Monotone piecewise cubic interpolation. *SIAM J. Numer. Anal.*, 17(2):238–246, 1980.

[12] A. Giachetti and N. Asuni. Fast artifacts-free image interpolation. In *Proc. of the British Machine Vision Conf. 2008 (Leeds, Sept. 2008)*, pages 123–132.

[13] M. Kraus, M. Eissele, and M. Strengert. GPU-based edge-directed image interpolation. In *Image Analysis*, volume 4522 of *LNCS*, pages 532–541. Springer, 2007.

[14] X. Li and M. T. Orchard. New edge-directed interpolation. *IEEE T. Image Process.*, 10(10):1521–1527, 2001.

[15] K. Martinez and J. Cupitt. VIPS—a highly tuned image processing software architecture. In *IEEE International Conf. on Image Process., 2005. ICIP 2005.*, volume 2, pages 574–577.

[16] D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer-graphics. *SIGGRAPH Comput. Graph.*, 22(4):221–228, 1988.

[17] S. C. Park, M. K. Park, and M. G. Kang. Super-resolution image reconstruction: A technical overview. *IEEE Signal Process. Mag.*, 20(3):21–36, 2003.

[18] N. Robidoux, A. Turcotte, M. Gong, and A. Tousignant. Fast exact area image upsampling with natural biquadratic histosplines. In A. C. Campilho and M. S. Kamel, editors, *Image Analysis and Recognition*, volume 5112 of *LNCS*, pages 85–96. Springer, 2008.

[19] J. Sun, J. Sun, Z. Xu, and H.-Y. Shum. Image super-resolution using gradient profile prior. In *IEEE Conf. on Computer Vision and Pattern Recognition*, 2008.

[20] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE T. Image Process.*, 13(4):600–612, Apr. 2004.

[21] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2003.

[22] M. Zhao and G. de Haan. Content adaptive video up-scaling. In S. Vassiliadis, L. Florack, J. Heijnsdijk, and A. van der Steen, editors, *Proc. of ASCI 2003, 9th Annual Conf. of the Advanced School for Computing and Imaging, 4-6 June 2003*, pages 151–156.