Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples

LIMING LOU[1,2], PAUL NGUYEN[2], JASON LAWRENCE[2,3], CONNELLY BARNES[2]
[1]Shandong University, [2]University of Virginia

Image pipelines arise frequently in modern computational photography systems and consist of multiple processing stages where each stage produces an intermediate image that serves as input to a future stage. Inspired by recent work on loop perforation [Sidiroglou-Douskos et al. 2011], this paper introduces *image perforation*, a new optimization technique that allows automatically exploring the space of performance-accuracy trade-offs within an image pipeline. Image perforation works by transforming loops over the image at each pipeline stage into coarser loops that effectively "skip" certain samples. These missing samples are reconstructed for later stages using a number of different interpolation strategies that are relatively inexpensive to perform compared to the original cost of computing the sample. We describe a genetic algorithm for automatically exploring the resulting combinatoric search space of which loops to perforate, in what manner, by how much, and using what reconstruction method. We also present a prototype language that implements image perforation along with several other domain-specific optimizations and show results for a number of different image pipelines and inputs. For these cases, image perforation achieves speedups of 2x-10x with acceptable loss in visual quality and significantly outperforms loop perforation.

Categories and Subject Descriptors: I.4.3 [**Image Processing and Computer Vision**]: Enhancement—*Filtering*; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Languages*

General Terms: Image processing

Additional Key Words and Phrases: Image filters, compilers, optimizations

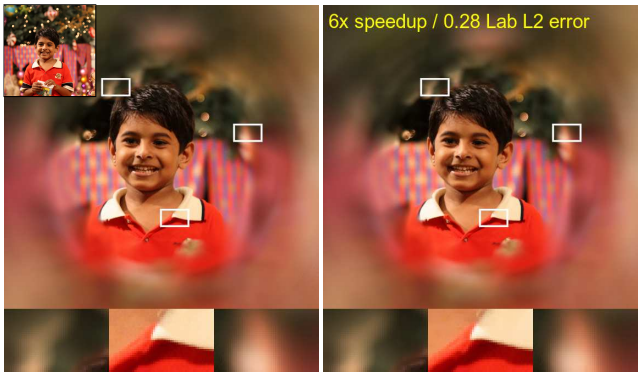
ACM Reference Format:

Lou, L., Nguyen, P, Lawrence, J., and Barnes, C. 2015. Image Perforation: Automatically Accelerating Image Pipelines by Intelligently Skipping Samples. ACM Trans. Graph. X, Y, Article Z (Month 2016), 13 pages.
DOI = ZZZ
http://doi.acm.org/ZZZ

[3] Author Jason Lawrence is now at Google Research.

Thanks to ShanShan He for a comparison. For supporting Liming Lou, we thank Meng Xiangxu and Chinese Scholarship Council. Funded by NSF grants CCF 0811493, CCF 0747220, HCC 1011444. Thanks to Creative Commons photographers: their photographs have been altered by our filters.

DOI 10.1145/XXXXXXX.YYYYYYY
http://doi.acm.org/10.1145/XXXXXXX.YYYYYYY



(a) Reference Output (b) Image Perforation

Fig. 1. Optimizing an image pipeline using image perforation. An anisotropic radial artistic blur is applied to (a, inset at top left) an input image to produce (a) the reference output. (b) An optimized version of this pipeline is automatically found by image perforation (the bottom row shows zoom regions). Image perforation works by transforming loops over image arrays to skip certain samples that are subsequently reconstructed from the available samples. This achieves faster running times at the cost of some loss in image fidelity. In this case, a speedup of 6x is achieved for a negligible amount of error. Photo credit:© Cheon Fong Liew.

1. INTRODUCTION

Image processing pipelines are common is modern graphics and computational photography systems. They consist of multiple processing stages where each stage produces an intermediate image that serves as input for a future stage. Examples of common image pipelines include HDR tone-mapping [Paris et al. 2011], edge-aware smoothing [Tomasi and Manduchi 1998; Chen et al. 2007], and many types of non-photorealistic image effects [Kang et al. 2009]. Figure 1 illustrates an image pipeline designed to produce a dramatic artistic effect that emphasizes the center of the image.

It is often the case that a particular implementation of an image pipeline can be altered, or *tuned*, to make it execute faster with some acceptable loss in fidelity. However, this type of optimization is still a difficult and time-consuming manual process, but one that is becoming more important alongside the proliferation of hardware platforms with widely varying resources and architectures from high-end laptops to smartwatches.

To help meet this challenge, we introduce *image perforation*, an automated approach for exploring an important space of performance-accuracy trade-offs inherent to an image pipeline. The output of image perforation is an ordered list of modified image pipelines (variants) that achieve faster and faster running times for greater sacrifices in accuracy. Our approach is inspired by recent work in software engineering on *loop perforation* [Sidiroglou-Douskos et al. 2011], a compiler technique that accelerates general-purpose programs by skipping certain loop iterations. Image perforation is a similar compiler optimization technique specifically designed to accelerate image pipelines. It works by skipping certain image samples that are expensive to compute and using relatively

inexpensive methods to reconstruct these missing samples when they are needed in future processing stages.

We introduce several code transformations designed specifically for loops over multidimensional data since images are typically interpreted as multidimensional arrays of color intensity values. These include loop transformations that skip a regular spaced set of samples and others that skip an irregular set of samples adapted to the local frequency content of the image (Section 3.1). Our approach also considers different methods for reconstructing missing samples such as nearest neighbor, Gaussian, spatially-varying Gaussian, and multilinear interpolation (Section 3.2). Finally, we explore several domain-specific code transformations including a technique we call *congruence simplification* (Section 3.3.3), which simplifies sample reconstruction code using congruence relations, and special handling of color channels, which can often be effectively perforated differently (Section 3.4).

The space of possible code transformations is combinatoric: type of perforation (e.g., uniform grid or nonuniform), reconstruction method, presence of domain-specific transformations, and associated parameters. Therefore, using an exhaustive or greedy search as was described by Sigiroglou-Douskos et al. [2011] makes it difficult to identify useful approximations. To overcome this limitation, we present a genetic search that efficiently explores the space of transformations by mutating a population of variants found to provide good performance-accuracy trade-offs over multiple generations (Section 5). The optimized programs have sampling rates automatically selected throughout their computations by the autotuner: in most cases grid sampling is used, but in some cases, nonuniform sampling is used.

We evaluate our approach using a prototype language and compiler that implements image perforation along with a benchmark of input images that spans different subjects, texture amounts, and resolutions (Section 6). We present results for seven common image pipelines: artistic blur (Figure 1), bilateral filter, bilateral grid, demosaicing, median filtering, two-pass blur, and unsharp masking (Section 7). We show that image perforation is able to achieve substantial speed-ups of 2x-10x with little loss in visual quality. We also demonstrate that image perforation significantly outperforms loop perforation in this domain. In Section 7.8, we port our optimized programs to the Halide [Ragan-Kelley et al. 2013] language for computational photography, and find that our ported programs are about $2\times$ faster than the original Halide programs.

2. RELATED WORK

This paper builds on developments in a number of different areas that have traditionally seen little overlap.

Filter approximation. Several techniques have been proposed that use various optimization strategies to approximate discrete linear filters [Uesaka 2003; Uesaka and Kawamata 2000; Sharman et al. 1998; Yu and Xinjie 2007; Alcázar et al. 1996]. Similar to our approach, these methods reduce running times at the cost of some approximation error. Farbman et al. [2011] demonstrated a multi-scale technique for accelerating convolutions with filters having large support. Their technique is able to approximate a certain class of image operations in linear time at error rates that are acceptable for many applications. Although our approach could also be used to tune individual linear filters, it is not restricted to this domain and can instead approximate arbitrary image pipelines that involve complex non-linear operations.

Optimizing image pipeline compilers. Image pipelines may be described in a number of domain-specific languages as graphs of operations that each process data locally with kernels [Shantzis 1994; Guenter and Nehab 2010; PixelBender 2010]. OpenCL and CUDA each expose a data-parallel programming model that can efficiently target both GPUs and CPUs [Buck 2007; OpenCL 2013]. The Halide programming language [Ragan-Kelley et al. 2013] optimizes image pipelines by allowing for loop transformations that trade-off parallelism, locality, and recomputation. In Halide, the programmer specifies a high-level algorithm along with a schedule that provides details about storage and execution order. Similar to our approach, Halide includes an auto-tuner that searches for schedules that deliver the best performance.

Image perforation differs from these previous methods in that it reduces runtime not by finding an optimal execution order or use of memory, but by replacing computationally expensive samples with interpolated values that are much faster to compute. Though our method does not explicitly search for optimizations that take advantage of execution scheduling, locality, parallelism, etc., we present some preliminary experiments that indicate the speed-ups due to our approximations persist through common vectorization transformations (Section 7.6), and offer performance advantages when combined with Halide (Section 7.8). Therefore, image perforation can be seen as a complementary approach that can be effectively combined with these alternative optimization strategies.

Subsampling in rendering. There is a rich history of rendering techniques that employ a similar *subsample-and-reconstruct* approach to achieve faster running times [Damez et al. 2003]. These include methods that subsample environment maps [Agarwal et al. 2003], indirect lighting calculations [Keller 1997], shadow calculations [Ramamoorthi et al. 2007], and even the framebuffer itself [Bishop et al. 1994; Yang et al. 2008; He et al. 2014; Vaidyanathan et al. 2014]. However, much less work has been done on subsampling image pipelines and, to our knowledge, no prior method attempts to automatically learn a profitable subsampling schedule "on the fly" for a particular scene or rendering technique.

Loop perforation. This paper builds on the work of Sigiroglou-Douskos and colleagues [2011] who described a general method for exploring the space of performance-accuracy trade-offs in general purpose programs by subsampling (perforating) `for` loops. Loop perforation can be applied to any loop that can be put in the form `for (int i = 0; i < n; i++)`. They considered transformations that multiply the step of a loop by a constant factor, skip a loop iteration with some probability, or alter the bounds of the loop. They described both an exhaustive search and a greedy search for identifying combinations of transformations that reduce running time for the least amount of error.

Perhaps the most significant difference between loop perforation and image perforation is the fact that, in many cases, image pipelines require reconstructing skipped samples in order to produce useful optimizations. For example, consider the final stage in which the output image is assembled. Simply skipping iterations would result in missing (black) pixels. To overcome these limitations, we introduce a broader class of loop transformations and new reconstruction strategies that provide a much richer set of optimizations that are more appropriate for image pipelines. Furthermore, we also introduce an efficient search strategy based on a genetic algorithm that is able to identify good optimizations more quickly than exhaustive search, which is very slow, or a greedy search, which is prone to becoming trapped in poor

local minima. We directly compare these two methods in Section 7.

Genetic algorithms and auto-tuning. Genetic algorithms and genetic programming (GP) are general machine learning strategies that use an evolutionary methodology to search for a set of programs that optimize some fitness criterion [Koza 1992]. From an initial population of candidates, a new evolved generation of candidates is produced through a series of mutation and combination operations (i.e., cross-over). After each generation, a new population is chosen that favors the fittest candidates and this process repeats.

Genetic algorithms have received renewed attention in the field of computer graphics lately. Recent work by Sitthi-Amorn et al. [2011] describes a GP approach to the problem of automatic procedural shader simplification. Brady and colleagues [2014] showed how to use GP to discover new analytic reflectance functions. We use a similar genetic algorithm as Sitthi-Amorn et al. [2011] to automatically search over the space of image perforations.

More generally, searching for optimal program implementations falls under the rubric of program auto-tuning. Auto-tuning has been successfully applied in many domains such as parallelizing stencil computations [Christen et al. 2011] and multigrid PDE solvers [Chan et al. 2009]. OpenTuner [Ansel et al. 2013] simultaneously employs many different search methods such as the simplex method, genetic search, and others. Although our method is inspired by this prior work, we employ a straightforward genetic algorithm and focus on the problem domain of image processing.

3. IMAGE PERFORATION

In this section, we introduce image perforation using a motivating example of optimizing a two-pass blur. We describe the different loop perforation strategies our approach considers for skipping and reconstructing samples (Sections 3.1-3.2) along with specific optimizations involving the choice of precomputed vs on-demand sample reconstruction (Section 3.3), congruence simplifications (Section 3.3.3), and handling color channels (Section 3.4). We present our prototype language and compiler in Section 4.

At its core, image perforation exploits the observation that the frequency content of the arrays produced at each stage in an image pipeline are often well below the theoretical Nyquïst limit. Sometimes this may be true only within local regions. This is due to a number of factors including characteristics of the input image itself (e.g., it may show a smooth blue sky alongside high-frequency textured regions) or the nature of the computation (e.g., some image regions may be more aggressively blurred than others). For example, if an intermediate stage aggressively blurs the image it receives, it should be possible to represent this stage's output using far fewer samples than the input. Following a similar argument, it should also be possible to accurately reconstruct these "skipped" samples when they are needed in future processing stages. The goal of image perforation is to automatically modify the source code of an image pipeline to exploit these opportunities while still accommodating a wide range of inputs.

We will assume that an image pipeline can be described as a directed acyclic graph where each vertex corresponds to a processing stage that computes an output array by executing a loop. A stage may depend on arrays computed in earlier stages. Note that the arrays can be of any dimensionality, although we use the term "image" because we focus on image processing. In our current prototype, the image processing stages are indicated by the developer using a semantic annotation (in our prototype language, we use `for each`). We leave for future work the automatic identification of these stages: this would require locating in the abstract syntax tree nested loops over spatial coordinates, which result in assignment to an output array. We also assume that the computation for each pixel is independent from other pixels.

To help describe image perforation, consider the following two-stage image pipeline that performs a Gaussian blur:

Listing 1. Two stage blur program.

```
function twoPassBlur(in, out)
  for each (x,y) of A as stage1
    A[x,y] = sum(K(i,σ)*in[x-i,y], i=-h...h)
  for each (x,y) of out as stage2
    out[x,y] = sum(K(i,σ)*A[x,y-i], i=-h...h)
```

The first stage computes a horizontal blur of the input within each scanline and stores this in a temporary array `A`. The second stage performs a vertical blur. In Section 7, we analyze the specific case of a wide Gaussian blur with $h = 8$ and $\sigma = 4$.

By the Nyquïst-Shannon's theorem [Pavlidis 2012], the uniform sampling theorem for band-limited signals, we know that the first stage is band-limited so that it can be subsampled by 4 in the x dimension without losing much information. Similarly, the second stage is band-limited such that it can be subsampled by 4 in both the x and y dimensions. By the same theorem, one can show that the samples that are skipped can be accurately reconstructed from those that are not skipped using some form of interpolation. Image perforation seeks to identify and exploit these types of optimizations automatically.

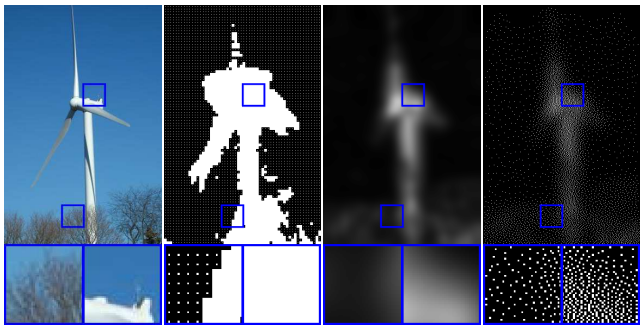
3.1 Perforation strategies

We consider three approaches for perforating loops over image arrays: grid sampling, adaptive sampling, and importance sampling.

Grid sampling. This technique transforms a loop so that it is executed only at a regularly distributed subset of locations in the array, or *samples*. This is an appropriate choice if the array is globally band-limited, such as the blur example. Specifically, grid sampling uses a spacing vector $\mathbf{s} \in \mathbb{N}^k$, where k is the number of variables in the `for each` loop. The samples are then distributed with equal spacing s_i along every dimension i. In the blur example above, `stage1` could be safely perforated using grid sampling with a spacing of $(4, 1)$ and `stage2` with a spacing of $(4, 4)$.

Adaptive sampling. This technique transforms a loop so that it is executed only at an irregularly distributed subset of samples, typically arranged so that high-frequency regions are sampled more densely (Figure 2). Adaptive sampling first evaluates a coarse grid of samples, and then adaptively refines areas by adding additional samples wherever the output is found to vary rapidly (Figure 2(d)). This approach was inspired by the shader sampling scheme of He et al. [2014]. Unlike He et al., however, we automatically determine where refinement occurs without requiring the user to write a separate procedure.

We introduce a parameter k that is the grid spacing. We construct the coarse grid with one sample every k pixels, and evaluate the output of the current pipeline stage at these coarse grid locations. Next, we assign a refinement priority for each coarse sample, which is the sum of absolute differences between the pixel value at the coarse sample and each of its 8 neighbors. We refine coarse samples with the highest priorities by surrounding them with closer samples. Specifically, we use a $\hat{k} \times \hat{k}$ fine grid, where $\hat{k}$ is k rounded up to the nearest odd integer. We continue to place fine samples until the target sample count is achieved.



(a) (b) (c) (d)

Fig. 2. Illustration of adaptive sampling and importance sampling. Adaptive sampling computes samples (b) by refining an initial coarse grid to have denser samples wherever the output image varies rapidly. Importance sampling computes from (a) the input image (c) an importance map using the texturedness measure of Bae et al. [2006] and takes a parameter f that determines the number of samples as a fraction of n the image resolution. We use a modified version of the dithering method of Ostromoukhov [2001] to (d) distribute these fn samples according to the importance map. Photo credit: © Frank Swift.

We also introduce a parameter f that is equal to the fraction of samples that will be computed relative to the size n of the input image. For example, $f = 0.5$ indicates that half of the pixels should be computed. The target sample count is then simply fn.

For efficiency, we use a histogram operation to decide the cutoff priority below which fine samples should not be generated. We place the refinement priorities in a histogram and work downward from the largest values of the histogram, accounting for the number of fine samples created, until we meet the target sample count. This histogram approach takes only $O(n)$ time, whereas a comparison sort would require $O(n \log n)$ time.

Importance sampling. Like adaptive sampling, importance sampling transforms a loop so that it is executed only at irregularly distributed samples (Figure 2). However, unlike adaptive sampling, importance sampling computes fixed sample locations based on the texture properties of the input image. This is an appropriate choice if the image array is locally band-limited. For example, in Figure 1, portions of the image near the edges are more aggressively blurred whereas regions near the center retain high-frequency details.

Importance sampling relies on an *importance map* to determine where to place samples. In general, it is infeasible to compute an optimal importance map for each stage as this would require knowledge of the stage's output. In our experiments, we found that the image texturedness measure of Bae et al. [2006] worked well for this purpose. For efficiency, we modify their method to use a simple Gaussian blur instead of a bilateral filter. Specifically, the importance map is computed by applying this slightly modified texturedness measure to the input image (Figure 2(b)). Alternatively, the programmer could supply a custom importance map.

To achieve speedups, our sampling strategies need to have a low time overhead relative to the existing parts of the image pipeline. Therefore, we introduce an efficient method for placing samples that are proportional to the importance map, and also relatively well-distributed spatially. This is related to the literature on Poisson disc sampling [Wei 2008].

Our efficient importance sampling method works by simply remapping the intensities of the importance map followed by dithering. We remap the intensities of the importance map so that it sums to the target number of samples, fn, after being clamped to the range $[0, 1]$. This remapping can be done efficiently using histogram operations as described in Appendix A. We then use dithering to determine the final sample locations, as shown in Figure 2(c). We explored both Floyd-Steinberg [1976] and Ostromoukhov dithering [2001], and found that Ostromoukhov's technique performed better due to its blue noise properties.

3.2 Reconstruction

The perforation strategies described previously result in some entries in the output image array being skipped. In most cases, simply ignoring these missing samples is unacceptable as they would result in black pixels in the image. We investigated several methods for reconstructing missing values from the set of computed samples. These include nearest neighbor interpolation, spatially invariant and spatially-varying Gaussian interpolation, and multilinear interpolation. In each case, we seek to balance the fundamental trade-off between reconstruction accuracy and running time.

For nearest neighbor reconstruction we simply copy into the output array the nearest sampled location, as determined by the Manhattan distance metric [Rosenfeld and Pfaltz 1968]. For spatially invariant Gaussian reconstruction we use an infinite impulse response (IIR) approximation to a Gaussian kernel [Young and Van Vliet 1995], although faster parallel algorithms could be used instead [Nehab et al. 2011].

For spatially-varying Gaussian reconstruction we use the method of repeated integration [Heckbert 1986]. In particular, we perform two integrations which gives a tent filter approximation to the Gaussian. This method can be used with both adaptive and importance sampling. For importance sampling, the parameter for our spatially varying Gaussian in d dimensions is given by $\sigma(\mathbf{x}) = \sigma_0 / M(\mathbf{x})^{1/d}$, where M is the importance map after rescaling, $\mathbf{x}$ is the spatial location, and σ_0 is a parameter either provided by the developer or, more commonly, determined during the genetic search described in Section 5. This is derived by requiring a spherical filter of radius $\sigma(\mathbf{x})$ to cover an approximately equal number of samples for every location $\mathbf{x}$, i.e., M is the "volume" of the sphere, and σ is the "radius" of the sphere. For adaptive sampling, where there is a fine sample we use $\sigma(\mathbf{x}) = 0.01$ and where there is a coarse sample we use $\sigma(\mathbf{x}) = \sigma_0 k$. Again, σ_0 is a parameter typically determined by the genetic algorithm.

We reconstruct the final image I' from the samples S and sampled image I, using a reconstruction filter G, according to:

$$I' = \frac{G * I}{G * S} \tag{1}$$

Here we assume the samples S are either zero or one and the sampled image has been initialized to zero in non-sampled locations.

The final reconstruction method we consider is multilinear interpolation. Clearly, we can support this in the general case by using repeated integration, which gives a tent filter reconstruction. However, when multilinear reconstruction is used in conjunction with grid sampling further accelerations and storage reductions are possible, as discussed in the next section.

3.3 On-demand reconstruction

We previously described reconstructing the missing values at each skipped location and storing these in the image array. However, in many cases it is advantageous to reconstruct skipped samples *on-demand*, as they are needed in subsequent processing stages. This trade-off of precomputed vs on-demand reconstruction is especially helpful for grid sampling, because the fixed grid sample locations permit fast reconstructions. We do not use on-demand reconstruc-

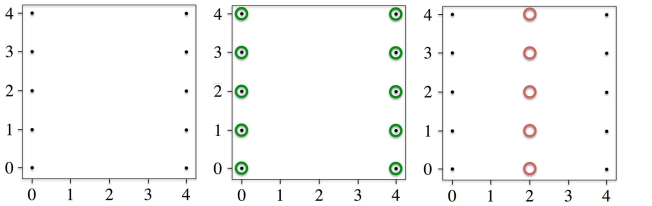


Fig. 3. Congruence simplification. Consider the two stage blur program of Listing 1, which applies a horizontal blur followed by a vertical blur. In (a), we show the locations that are sampled for the intermediate array `A` of the blur program. This array is horizontally blurred, and has grid sampling with spacing $(4, 1)$. In (b), stage 2 of the blur program reads from the array `A` and writes to the output array `out`. This stage reads from the array `A` locations shown in the green circles. Specifically, the array `A` is read at location $(x', y') = (x, y - i)$, where x is the variable we are looping over, with $x \equiv 0 \pmod 4$. We can prove that the read locations (shown in green) coincide with sample locations, so bilinear interpolation simplifies to nearest neighbor lookup. In (c) suppose that we instead read from a different array location $(x', y') = (x + 2, y - i)$. We can prove that bilinear interpolation for this expression simplifies to linear interpolation with constant weights $(0.5, 0.5)$ between horizontally neighboring samples (depicted by read locations in red being midway between adjacent samples).

tion for irregular samples, because it is inefficient in that case. In particular, we apply three optimizations: rewriting pixel reads, storage optimizations, and congruence simplification.

3.3.1 *Rewriting pixel reads.* This optimization replaces pixel reads from the reconstructed image I' with on-demand reconstructions from nearby grid samples. For example, with nearest neighbor reconstruction, we can simply read from the nearest grid point. Or, for linear interpolation, we can count the number of dimensions p with nontrivial grid spacing ($s_i \neq 1$), and then use multilinear interpolation of order p (i.e., linear for $p = 1$, bilinear for $p = 2$, trilinear for $p = 3$, etc). All pixel reads in the output image should be rewritten this way, when the read is made in a stage after the current stage.

3.3.2 *Storage optimization.* Storage optimization reduces storage requirements and improves cache performance by storing the grid spacing along with the type information for the image I that is being looped over. This allows rewriting accesses within the current stage so that $I(x, y, \ldots)$ becomes $I(x/s_1, y/s_2, \ldots)$. Pixel reads in later stages simply use the same reconstruction as before, but with coordinates appropriately divided by the spacing. Because the maximum index is divided by the spacing, the storage requirements for the image are likewise reduced. Note that the sampling array S can be omitted because reconstruction is done analytically instead of by equation (1). This storage optimization gives a result similar to downsampling, but note that computation is also being downsampled alongside the image values.

3.3.3 *Congruence simplification.* The final optimization is congruence simplification. This allows us to rewrite pixel read operations to have constant reconstruction weights whenever they have certain congruence properties. As a motivating example, recall the two stage blur program shown in Listing 1, which first horizontally blurs the input into a temporary array `A`, and then vertically blurs the temporary array into the output array `out`. Previously, we grid sampled the first computation stage (`stage1`) of the blur program by $(4, 1)$, and the second computation stage (`stage2`) by $(4, 4)$, and used multilinear reconstruction for both stages. In this case, the loop in `stage2` takes the following form:

Listing 2. Stage 2 of the blur program from Listing 1.

```
for y=0 to out.height() step 4
  for x=0 to out.width() step 4
     out[x, y] = sum(K(i,σ)*A[x, y-i], ...)
```

The stored samples for the intermediate (horizontally blurred) array `A` are depicted in Figure 3(a). Because `A` is reconstructed on-demand, we would ordinarily have to use multilinear reconstruction every time we read from the `A` array. Specifically, in `stage2` of the blur program, we read from the array `A` by using the code `A[x, y-i]`. Array `A` is grid sampled by $(4, 1)$, and therefore has some missing samples, so ordinarily we would have to do interpolation across the x coordinate. However, as shown in Figure 3(b), the green locations that are read from coincide exactly with the black locations that have been previously sampled and stored. Therefore, no interpolation is necessary for this read operation. Mathematically, we can define the read location (x', y') for the array `A` to be $(x', y') = (x, y - i)$. We can note that the loop variable x in Listing 2 has the property $x \equiv 0 \pmod 4$, so we can prove that the read index $x' \equiv 0 \pmod 4$, and thus linear interpolation can be replaced with a single nearest neighbor lookup. If we had instead read from location $x' = x + 2$, then we could prove $x' \equiv 2 \pmod 4$, and the linear interpolation weights would be the constants $(0.5, 0.5)$. This second scenario is shown in Figure 3(c).

In general, we perform congruence simplification by passing all the known loop congruences to a theorem prover [De Moura and Bjørner 2008] and attempting to prove that a given read index variable (e.g., x') is congruent to each possible constant, modulo the grid spacing. If the index is congruent to a constant then the interpolation has known, constant weights.

Finally, note that congruence simplification combined with common subexpression elimination can reduce the number of taps in a filter. For example, suppose we have a two stage blur: a 5-tap convolution in x followed by 100-tap convolution in x. Suppose we grid sample both stages with a spacing of 2 horizontally. Then congruence simplification will convert the original 100 array reads in the second stage into 50 unique reads, giving a 50-tap filter.

3.4 Special handling of color channels

Our final approximation concerns color channels. In video processing, color spaces such as YUV [Black 2009] are common, where chromatic information is represented with fewer bits or subsampled. This is done because the human visual system is less sensitive to chrominance than luminance [Párraga et al. 1998].

We allow for similar channel subsampling. We assume the last array dimension stores color channels. If color approximation is invoked, then all loops over color do not sample channels above a maximum number of color channels c (e.g. $c = 1$ samples only the first color channel). The remaining channels are simply copied from the input image after reconstruction. Color subsampling is often more effective if the image processing is performed in a color space such as YUV, so we also expose several convenience functions for color conversion.

4. PROTOTYPE LANGUAGE AND COMPILER

We built a prototype programming language and compiler that implements image perforation. Our compiler expects programs with

C-like semantics as input, along with an approximation schedule, which specifies the manner in which each loop is perforated and missing samples are reconstructed, and any other specific optimizations like special handling of color channels. From these two input files our compiler produces backend code, which is currently C++.

As we mentioned earlier, our language exposes a `for` loop, which allows for standard loop perforations, as well as a `for each` loop which allows for image perforations. The `for each` loop serves to identify a particular image stage for the analysis described in the previous section. It has the following syntax:

```
for each vars of image as stagename
```

This creates a loop over variables such as (x, y) in the domain of `image`, starting with the first specified image dimension. The `stagename` is a unique identifier for the stage.

Our compiler first converts the input program into an abstract syntax tree (AST) [Aho et al. 1986]. Approximation schedules can be specified manually using annotations associated with each loop. Alternatively, the genetic search presented in the next section (Section 5) can be used to automatically explore this space. The compiler creates the output program by directly modifying the AST according to the approximation schedule, such as injecting sampling code above `for each` loops, and reconstruction code below. Array reads are modified as necessary due to any of the on-demand reconstruction optimizations discussed in Section 3.3, and color loops are modified as required by Section 3.4. The code generator produces output code in the back-end language.

We also explored simple parallelization strategies such as thread parallelism of loops and SIMD parallelization across color channels. Although we leave exploring the full transformation space of Halide [Ragan-Kelley et al. 2013] to future work, our preliminary results indicate that our approximations remain beneficial across common parallel loop scheduling optimizations. This is discussed further in Sections 7.6 and 7.8.

5. GENETIC SEARCH FOR GOOD SCHEDULES

Although image and loop perforation schedules can be specified by hand in our language, we found this to be a time-consuming process that can easily overlook beneficial optimizations. To address this challenge, we developed a genetic algorithm that automatically searches over the space of possible code transformations consisting of different perforation strategies, reconstruction techniques, and associated parameters.

Our genetic search closely follows the method of Sitthi-Amorn et al. [2011]. We adopt their fitness function and tournament selection rules and we use the same method to compute the Pareto frontier of program variants that optimally trade-off running time and image fidelity.

We employ standard mutation and cross-over operations to explore the space of code transformations. Specifically, the mutation step chooses a new perforation method (e.g., grid sampling or importance sampling) and its parameters (e.g., grid spacing or fraction f), as well as the reconstruction method, the choice of precomputed vs on-demand reconstruction (this only applies to grid sampling), and the choice of whether to sub-sample color channels. We use two-point crossover to generate a single child from two randomly chosen parents. For example, suppose that the loops in a pipeline are numbered $1, 2, 3, 4$ and parent A has genome $A_1A_2A_3A_4$ (genes may be empty for no approximation), and parent B has genome $B_1B_2B_3B_4$. Two crossover points are selected at random, which could result, for example, in a child $A_1B_2B_3A_4$.

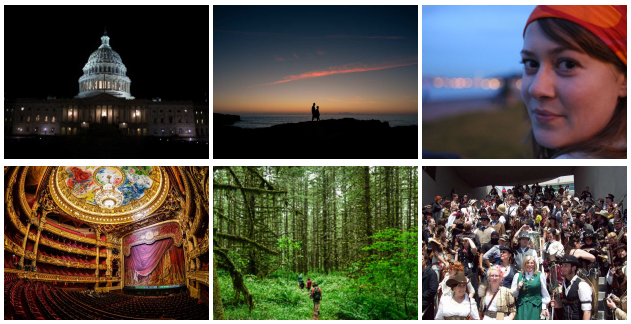


Fig. 4. These six training images were selected out of the 120 image benchmark dataset. The training images are used by our genetic algorithm to identify good perforation schedules. Photo credits clockwise from top left: © Gage Skidmore; Paulo Valdivieso; jhenryrose; Trey Ratcliff; Loren Kerns; ChrononautClub.

6. IMAGE BENCHMARK

We assembled a benchmark of images to serve as training data for our genetic search and to test the accuracy of our optimized image pipelines. This benchmark consists of 120 images downloaded from Flickr that span a range of subjects, frequency characteristics, and resolutions. A portion of this dataset is shown in Figure 4.

Our benchmark is divided into three categories: man-made (i.e. cities, bridges, streets, etc.); natural (i.e. bodies of water, caves, landscapes, etc.); and people (i.e. portraits, crowds, different skin color and ethnicities, etc.). We chose these specific categories because we believe they offer good coverage of the type of content typically used in modern image pipelines.

One goal of this benchmark was to allow studying how image perforation varies across images having different frequency content. To this end, for each of our categories we selected 10 "low texture" images and 10 "high texture" images. We determined the degree of texture within an image using the method of Bae et al. [2006]. Low texture images are those with a mean texture measure less than 12 (assuming 8 bit pixel values in the range [0,255]) and high texture images are those with a mean texture measure greater than 15. We calculated these texture measures on 820x614 images (approximately 0.5MP).

Another goal was to verify that optimized pipelines perform well on images with different resolutions. To this end, our benchmark includes a mixture of images with both low (256x192) and medium (820x614) resolutions. We chose these specific resolutions to make training and testing times tractable.

In total, our benchmark contains 120 images (3 categories x 2 texture levels x 10 images per category x 2 resolutions). We chose 6 images (one image from each category and texture level including both low and medium resolutions) to guide the genetic search (Figure 4) and used the remaining 114 images for testing.

7. RESULTS

We evaluated image perforation by using our prototype compiler (Section 4) and genetic search (Section 5) to automatically optimize seven image pipelines. For comparison, we used this same genetic algorithm to search over only the space of code transformations described in the original loop perforation paper [Sidiroglou-Douskos et al. 2011]. To accomplish this we simply replaced `for each` loops with `for` loops and used our same codebase. For two of these pipelines, we investigated combining image perforation with Halide (Section 7.8).

For the genetic search we used a cluster consisting of 64 AMD Opteron 6276 machines with 2MB cache, 2.3GHz processors with 8 cores and 4GB of memory. In every case, we evolved a population of size 100 over 200 generations using the process described in Section 5. The six images shown in Figure 4 were used to guide the search. Specifically, we computed the average running times required to compute the output along with the average L^2 difference in the CIE Lab color space against the reference output for each program variant in each population. (Recall that "program variants" in this context are schedules that describe the specific perforation and reconstruction strategies applied to each loop along with associated parameters.) At the end of each search, we compute the Pareto frontier, which is the set of optimal program variants in the sense that no other variant was found that performed better in one dimension (either accuracy or running time) while being no worse in the other. Figure 5 shows the Pareto frontiers for all seven image pipelines. These are discussed in detail below. Frontiers that hug the origin are preferable as these represent optimizations that offer the greatest performance gains with the smallest visual errors. The faint dots in these graphs plot the measured running time and mean Lab L^2 error of each program variant and each image in our testing set to give some indication of the statistical variance in these results.

Table I. Statistics for how often each perforation strategy was chosen in the image pipelines we studied. These numbers reflect only those program variants with mean Lab errors less than 10.

Application	Grid	Importance	Adaptive	None
Artistic blur	24%	15%	60%	2%
Bilateral filter	68%	21%	8%	3%
Bilateral grid	88%	0%	0%	12%
Demosaic	59%	0%	0%	41%
Median	40%	8%	52%	1%
Two-pass blur	96%	0%	0%	4%
Unsharp mask	86%	0%	0%	14%

Table I summarizes how often each perforation strategy was chosen in the optimized programs along the Pareto frontier for each image pipeline. Note that grid sampling tends to be the most popular choice. We attribute this to the fact that grid sampling has the lowest performance overhead since it permits fast reconstruction and avoids the more expensive computations associated with importance and adaptive sampling. Nonetheless, in many cases adaptive and importance sampling strategies were chosen, usually with spatially varying Gaussian reconstruction, which indicates the presence of particularly expensive inner loops whose execution outweighs these overheads. We highlight a few of these cases below.

7.1 Artistic blur

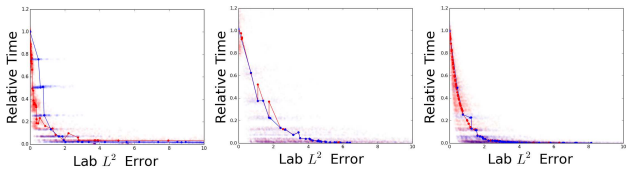
The image effect in Figure 1 is designed to dramatically emphasize the center of a photo. This is achieved with a spatially varying anisotropic blur that becomes stronger near the periphery of the image. The source location for the blur kernel is also randomly jittered to add an interesting artistic effect. In pseudocode, this effect is implemented as follows:

```
function artistic(in, out)
for each (x, y) of out as stageOutput
  r = distance of (x,y) from image center
  if (r < cutoff)
    out[x,y] = in[x,y]
  else
    out[x,y] = 0
    for (i) of -h..h as stageKernelX
      for (j) of -h..h as stageKernelY
        out[x,y] += kern(r,i,j)*in[x-i,y-j]
```

The function `kern(r,i,j)` returns pre-normalized weights of a kernel that produces a wider blur with increasing values of `r`, the distance to the image center.

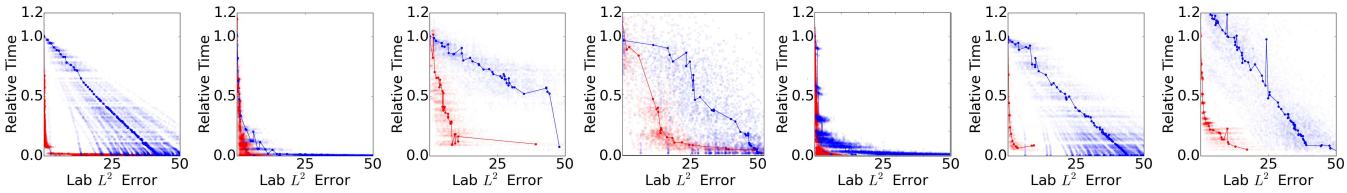
Figure 5(a) shows the results of the genetic search for image perforation. Image perforation overall far outperforms loop perforation for this effect. To better understand why this is the case, consider the specific result in Figure 1, which shows an optimized pipeline found with image perforation that gives a speedup of roughly 6x. In this case, image perforation optimizes `stageOutput` using adaptive sampling and computes only 25% of the samples with a grid spacing of 3. Spatially varying Gaussian interpolation was used to reconstruct skipped samples and no perforations were applied to the inner loops `stageKernelX` and `stageKernelY`. In contrast, the best strategy for loop perforation to achieve a similar speedup is to optimize `stageOutput` by computing only the first 75% of samples and similarly reduces `stageKernelX` and `stageKernelY` by computing only the first 41% and last 96% of samples, respectively. However, this results in large missing (black) areas in the output image (see page 24 of the supplemental for the loop perforation result). Additionally, skipping portions of the nested sum over the pre-normalized filter kernel causes the blurred regions to become noticeably darker. Due to the far greater flexibility in how loops over images may be reduced and the fact that missing values can be accurately reconstructed, especially in low-frequency regions like the blurred areas near the edges of the image, image perforation finds a program that is six times faster than the original with very few visible artifacts. Please see the supplemental document that includes more extensive comparisons for each of these pipelines.

We also performed an experiment that compares our full method with our method restricted to use only grid sampling. The results are shown in Figures 6(a) and 7(top). As illustrated in Figure 7(top), our full method more faithfully reproduces high-frequency details in the center of the output image. Specifically, for a target speedup of 2x, our full method uses adaptive sampling and computes only 45% of the samples with a grid spacing of 2. The comparable grid sampling strategy uses spacing $(1, 2)$.



(a) Artistic blur (b) Bilateral filter (c) Median

Fig. 6. A comparison of Pareto frontiers between (red) our full method and (blue) our method restricted to use only grid sampling. (See also Figure 7.) Note that in most cases having the ability to perform importance and adaptive sampling gives overall better performance. However, in some cases such as bilateral filter, grid sampling is preferable and one is better off spending more time searching over this restricted set of perforation strategies. These results are discussed in more detail in Sections 7.1, 7.2, and 7.5.



(a) Artistic blur (b) Bilateral filter (c) Bilateral grid (d) Demosaic (e) Median (f) Two pass blur (g) Unsharp mask

Fig. 5. Pareto frontiers for (red) image perforation and (blue) loop perforation for the image pipelines considered in our evaluation. These are the result of taking the Pareto frontier from the training set and evaluating on the test set, thus, the curves are not monotonic. Opaque points joined with line segments correspond to mean time and error for a single program variant over the entire test dataset, whereas translucent points correspond to individual test image/program variant combinations.

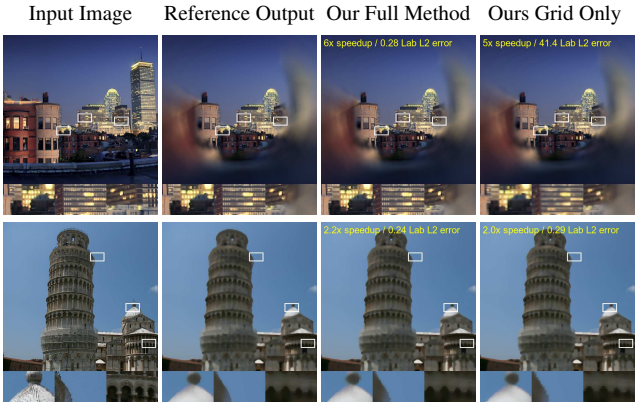


Fig. 7. Individual image results for a comparison between our full method and our method restricted to use only grid sampling (see also Figure 6). These are shown for the two applications where we observed visual improvements: **artistic blur** (top) and **median** (bottom). Each row compares optimized pipelines computed using each method for similar speedup factors. The target speedup for both applications is 2x. In both cases, our full method chooses adaptive sampling, which allows more accurate reproduction of high-frequency texture details. These results are discussed in more detail in Sections 7.1 and 7.5. Photo credits: © Werner Kunz and Robert Izumi.

7.2 Bilateral filter

A bilateral filter [Tomasi and Manduchi 1998] is a nonlinear smoothing filter that avoids smoothing across edges. Several methods have been designed specifically to accelerate bilateral filters [Chen et al. 2007; Banterle et al. 2012]. In particular, the method of Banterle et al. [2012] generates samples using a Poisson-disk distribution. This is similar to our importance sampling, only we use a simpler dithering approach to generate samples.

We evaluated image and loop perforation using a standard bilateral filter [Tomasi and Manduchi 1998], implemented as follows:

```
function bilateral(in, out)
for each (x,y) of out as stageOutput
  out[x,y] = 0
  norm = 0
  for each (i,j) as stageKernel
    rw = range(in[x,y],in[x-i,y-j])
    sw = spatial(i,j)
    out[x,y] += rw*sw*in[x-i,y-j]
    norm += rw*sw
  out[x,y] /= norm
```

Here the functions `range()` and `spatial()` return typical values for the range and spatial filter components, respectively, computed using a Gaussian function with standard deviations of 4 and 0.8 pixels, respectively. The spatial filter has a 9x9 square shape.

Pareto frontiers for image perforation and loop perforation are plotted in Figure 5(b). Note that unlike the artistic blur, the convolution computed in `stageKernel` is normalized outside of the loop. This allows loop perforation to skip iterations without introducing such egregious errors. Figure 8(top row) shows two optimized pipelines with comparable speedups. Interestingly, the image perforation result optimizes the outer loop `stageOutput` using importance sampling with a value of f equal to 75%. The inner loop `stageKernel` is perforated using grid sampling with a spacing vector of $(2, 2)$. In contrast, loop perforation computes only the last 96% iterations of `stageOutput` (notice the black scanline of missing pixels along the top of the image) and executes only the first half of the iterations in `stageKernel` in each of the two spatial dimensions. Although the benefits of image perforation are less dramatic than in other cases, the results obtained with our approach are still consistently better.

When we compared optimizing the bilateral filter using our full method to using our method restricted to only grid sampling, we found the visual differences were fairly minor (see Figures 6 and 7). This appears to be because of a combination of two factors: (1) our bilateral filter implementation is already quite fast and so the overhead of using non-uniform sampling is only slightly preferred by the search process, and (2) our error metric is simplistic and does not take into account subtleties of human vision such as a preference for greater fidelity along high-frequency edges. It would be worth exploring the use of perceptual error measures in future work.

7.3 Bilateral grid

The bilateral grid [Chen et al. 2007] accelerates the bilateral filter by embedding the problem in a higher-dimensional domain that enables the use of fast linear filters. It computes the output in three steps. First, the input image is *splatted* into a three-dimensional grid with two spatial dimensions and one range dimension. Second, a *linear blur* is applied to this grid in order to smooth values that are nearby in both space and pixel value. Finally, the output image is *sliced* from the grid by evaluating the result of the blur at the grid location associated with each pixel in the input.

We implemented the bilateral grid using five `for each` loops without any nesting:

```
function bilateral_grid(in, out)
for each (x, y, z) of in as stageSplat
```

```
  grid[x/σs,y/σs,in[x,y]/σr] += (in[x,y], 1)
for each (x, y, z) of blurX as stageBlurX
  blurX[x,y,z] = sum(K(i)*grid[x-i,y,z],
                                  i=-h..h)
for each (x, y, z) of blurXY as stageBlurY
  blurXY[x,y,z] = sum(K(i)*blurX[x,y-i,z],
                                  i=-h..h)
for each (x, y, z) of blur as stageBlurZ
  blur[x,y,z] = sum(K(i)*blurXY[x,y,z-i],
                                  i=-h..h)
for each (x, y, z) of out as stageSlice
  out[x,y] = blur[x/σs,y/σs,in[x,y]/σr]
```

Note that the bilateral grid is designed to already chose reasonably good sub-sampling rates along each dimension. Therefore, image perforation can carefully explore decimating every loop by varying amounts, and does find some advantageous strategies, but the overall performance gain is modest in comparison to other pipelines we evaluated. The sampling rates used by the bilateral grid were a spatial σ_s of 1/64 the image width, a range σ_r of 0.1, and a grid Gaussian with width of 5. Note that in general there are collisions in the grid, so the splatting stage increments an additional counter "channel" by 1 in the grid (this stage is not parallelized), and the slicing stage divides by the sum of these counters when converting back to color space. See Chen et al. [2007] for details.

As illustrated by the Pareto frontiers in Figure 5(c), image perforation provides dramatically better results than loop perforation for this pipeline. We attribute this to the ability of our method to subsample with reconstruction the three blur stages and the slice stage. The results in Figure 8(second row) offer a representative example of the different optimizations found with these methods. The image perforation result subsamples `stageBlurX` and `stageBlurY` with spacing $(8, 4, 1)$, `stageBlurZ` with spacing $(4, 4, 1)$, and `stageSlice` with spacing $(2, 2)$. Based on the spatial σ_s parameter setting, the subsampled grid arrays therefore work out to have size $(8, 16, 64)$ for `blurX` and `blurXY`, and size $(16, 16, 64)$ for `blur`. In each case, on-demand reconstruction is used with multilinear interpolation. In contrast, the loop perforation result chooses to sample contiguously the last 91% of `stageBlurX`, the last 93% of `stageBlurY`, the first 70% of `stageBlurZ`, and the last 74% of `stageSlice`. Note that loop perforation cannot reconstruct missing data, so it introduces black regions in the image and fails to handle certain color ranges.

7.4 Demosaic

A color digital camera usually acquires color information by using a Bayer filter mosaic, which alternates between sampling the red, green, and blue portions of the incoming spectrum in a repeating pattern across the sensor. To obtain a final RGB image it is necessary to *demosaic* the raw Bayer image by interpolating a complete RGB vector at each pixel. We implemented a simple image demosaic algorithm which takes an input pattern that is 50% green, 25% red and 25% blue, known as "RGGB." The algorithm first interpolates missing colors along the x-dimension and then along the y-dimension using simple linear blending of adjacent samples:

```
function demosaic(in, out)
repeat for each color channel c,
for each (x,y) of A[c] as stageInterpX[c]
    A[c][x,y]=sum(W(x,y,c,i)*in[y,x+i],
                  i=-1...1)
for each (x,y) of out[c] as stageInterpY[c]
    out[c][x,y]=sum(W(x,y,c,i)*A[c](y+i,x),
                  i=-1...1)
for each (x,y) of out, combine channels as
    OutputCombination
```

The filter weight function `W(x,y,c,i)` gives kernel weights for a neighborhood around the center pixel in either the x or y direction. The kernel weights are a function of (x, y), because some pixels have known colors whereas others must be interpolated (in practice, we use weights that are either 0, $1/2$, or 1). This simple algorithm is not intended to suppress aliasing, and serves as a "stress test" for our algorithm, because its per-pixel efficiency is already quite high, thus limiting potential speed improvements.

Figure 5(d) compares the Pareto frontiers for image perforation and loop perforation on this pipeline. By applying different grid sampling strategies for each color channel, image perforation typically produces better results. The images in Figure 8(fourth row) show a representative comparison of the two approaches. Image perforation applies grid sampling with a spacing of $(4, 8)$ for `stageInterpX[G]`, a spacing of $(8, 2)$ for `stageInterpY[G]`, and $(8, 4)$ for `stageInterpY[B]`. Finally, it grid samples `OutputCombination` with a spacing of $(8, 1)$ and the missing samples are reconstructed on-demand. In contrast, loop perforation executes only the first 66% of `stageInterpX[G]`, the first 71% of `stageInterpY[G]`, the first 75% of `stageInterpX[R]`, the first 69% of `stageInterpY[R]`, the first 67% of `stageInterpX[B]`, the first 73% of `stageInterpY[B]`, and the first 80% of `OutputCombination`. This strategy produces unusable results.

Although image perforation outperforms loop perforation in this case, it attenuates some high-frequency details and introduces chromatic aberrations due to using different sampling rates for different color channels. We believe this type of result could be improved by replacing the simple error metric that guides our search by a more perceptually meaningful one.

7.5 Median

The median filter is a popular non-linear filter that removes noise in an image. It works by replacing each pixel value with the median of the colors in its surrounding neighborhood according to the following pseudocode:

```
function median_filter(in, out)
for each(x,y) of input as stageLoop
    for each i as stageKernelX
        for each j as stageKernelY
            list_r.add(in_r[x+i,y+j])
            list_g.add(in_g[x+i,y+j])
            list_b.add(int_b[x+i,y+j])
    out_r[x,y]=median(list_r)
    out_g[x,y]=median(list_g)
    out_b[x,y]=median(list_b)
```

Figure 5(e) compares the Pareto frontiers of image perforation and loop perforation for this filter. In general, image perforation outperforms loop perforation across the range of acceptable errors. A comparison of the two methods is shown in Figure 8(fifth row) at a target speedup of 3x. The loop perforation result contains strong aliasing artifacts because it samples `stageKernelX` and `stageKernelY` with a spacing of 2. These artifacts are due to the estimated median oscillating between modes when the gathered colors have a multi-modal distribution. Image perforation samples `stageLoop` using adaptive sampling and computes only 28% of the samples with a grid spacing of 2, and then uses spatially varying Gaussian interpolation to reconstruct skipped samples with $\sigma = 0.95$.

The second row in Figure 7 compares the result of applying our full method to using image perforation restricted to only grid sampling. At the target speedup of 2x, the full method selects an adaptive sampling strategy that allows computing only 36% of the samples. This program corresponds to a single point in the graph of Figure 6(c). The comparable grid sampling strategy uses a spacing of $(1, 2)$. In the full method, samples skipped in `stageLoop` are reconstructed using Gaussian interpolation with $\sigma = 0.82$, while `stageKernelX` and `stageKernelY` are modified to evaluate only 6% and 22% of the samples, respectively. The grid sampling method, in contrast, reduces `stageKernelX` and `stageKernelY` to consider only 29% and 77% of the samples in those loops, respectively.

7.6 Two-pass blur

Figure 5(f) compares the Pareto frontiers for the two-pass Gaussian blur presented in Section 3 for image perforation and loop perforation. We implemented the `sum` within the convolution as a manually unrolled expression, therefore neither our method nor loop perforation are able to subsample this step. Two specific optimizations that give roughly 5-6x speedups are highlighted in Figure 8(third row). Image perforation achieves a 5.6x speedup by perforating both `stage1` and `stage2` using grid sampling with spacing vectors of $(4, 1)$ and $(4, 4)$, respectively. Note that this corresponds to the maximum amount of perforation possible without exceeding the Nyquïst limit with the chosen standard deviation of $\sigma = 4$. To achieve a similar speedup, loop perforation executes only the first 12% of iterations in `stage1` and the first 58% of `stage2`. This leaves the majority of the output pixels unassigned.

We also used this pipeline to study how image perforation works in conjunction with common parallel scheduling and vectorization optimizations, where vectorization is applied across color channels. For the image perforation result shown in the third row of Figure 8, we found that for high-resolution inputs we were able to gain an additional speedup of 4x for parallel scheduling, 2.7x for vectorized scheduling, and a combined 6x speedup for both parallel and vectorized scheduling. We performed these experiments using an x86 machine with the AVX instruction set and 4 processing cores. We observed similar performance gains for the program variants found elsewhere along the Pareto frontier, which is shown in Figure 5(f). For lower resolution inputs we found that the benefit of parallelism is less pronounced due to the overhead of inter-thread communications. In all, these experiments confirm that pipelines optimized with image perforation enjoy benefits of parallelism and vectorization, similar to other image pipelines (see also Section 7.8 for additional results).

7.7 Unsharp mask

Finally, we used image perforation to accelerate a standard unsharp mask filter that is designed to enhance high-frequency image details, implemented as follows:

```
function unsharp(in,out)
  twoPassBlur(in,blur)
  for each (x, y) of out as stageOutput
    out[x,y] = s*in[x,y] + (1-s)blur[x,y]
```

Here the sharpness coefficient `s` controls the amount of detail enhancement and was set to 3. Figure 5(g) shows the Pareto frontiers for this effect. Again, image perforation delivers a significantly better performance-error trade-off than loop perforation across the range of acceptable errors. Figure 8(bottom row) highlights two optimized results with comparable performance. The image perforation result uses grid sampling with spacings $(8, 1)$ for both stages of the blur, and leaves the output stage alone. In contrast, the loop perforation result samples the last 42% of the first blur stage and last 34% of the second blur stage. Note that the loop perforation introduces black regions in the blur, which after unsharp masking become overly bright regions of the output image.

7.8 Comparison with Halide

For two of our image pipelines, we compared optimized results found with image perforation to optimized implementations available in Halide [Ragan-Kelley et al. 2013] in order to study the relationship between vectorization and parallel scheduling optimizations and perforation optimization strategies that can result in irregular patterns in the processed image arrays. However, note that C++ compilers do not apply the same optimizations as Halide. Therefore, to achieve a fair comparison we ported our optimized programs from C++ to Halide to allow directly comparing approximated and non-approximated Halide programs. We ported the same optimized programs that were previously discussed and highlighted in Figure 8.

The results are shown in Table II. Modest speedups of just under 2x are obtained when using vectorization. When vectorization is disabled for the blur program, a more substantial 3x speedup is obtained. The reason our programs are not faster when vectorized is that Halide's current vectorization does not always succeed when presented with the memory access patterns used in our grid samples. We believe these results could be improved by carefully loading non-contiguous samples into SIMD registers. However, we defer this important topic to future research.

Table II. Performance comparison between optimized Halide implementations and programs first optimized using image perforation and then using Halide.

Application	Vectorized	Original Halide [ms]	Perforation + Halide [ms]	Speedup
Bilateral grid	Yes	7.8	4.1	1.9x
Two stage blur	No	170	52.2	3.3x
Two stage blur	Yes	72.7	41.3	1.8x

8. CONCLUSION AND FUTURE WORK

We have presented a new general purpose technique for optimizing image pipelines called image perforation. Image perforation works by transforming loops so that they skip certain samples. Our method then reconstructs these skipped samples from the samples that are computed, in case the skipped samples are needed in future processing stages. Our approach considers a range of perforation strategies and reconstruction methods that include regular grid sample placement and two strategies for nonuniform sample placement that attempt to locate samples more densely in areas with high-frequency content. We also described a genetic search that explores the combinatoric space of possible optimizations and outputs a sorted list of program variants that effectively trade visual fidelity for performance gains. We evaluated image perforation by implementing seven different image pipelines using a prototype compiler and search tool. We also compared our approach to loop perforation, a related method described by Sidiroglou-Douskos et al. [2011] for optimizing general purpose programs that

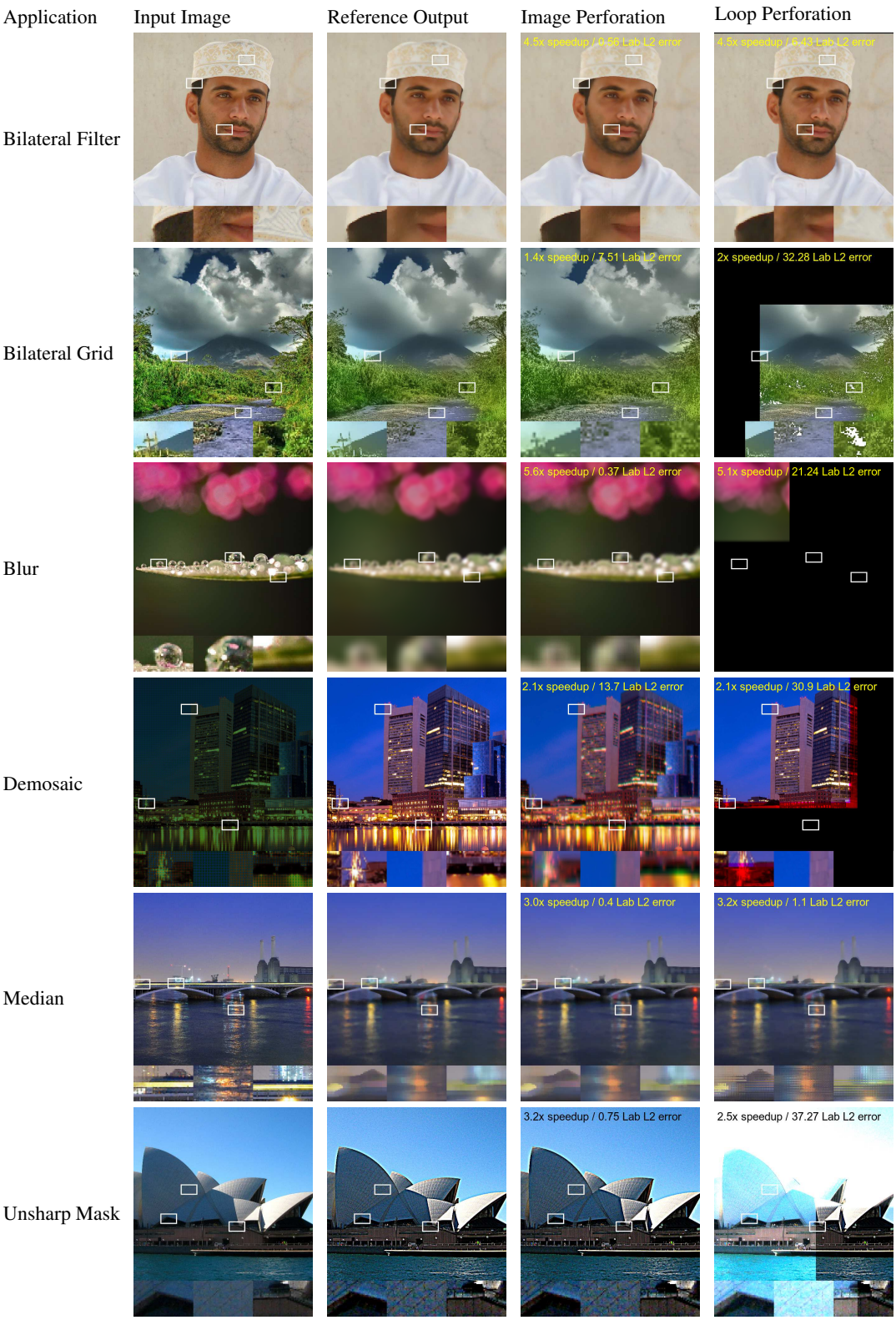


Fig. 8. Image perforation and loop perforation results for four image pipelines from top to bottom: **bilateral filter**, **bilateral grid**, **blur**, **demosaic**, **median** and **unsharp mask**. Each row compares optimized pipelines computed using each method for similar speedup factors. Please consult the supplemental document for extensive comparisons for each of these pipelines. Note that one can zoom in to see the Bayer mosaic pattern for the demosaic input. From top to bottom row, credits: © Charles Roffey; Trey Ratcliff; Neal Fowler; Eric Wehmeyer; Duncan Harris; Sandy Glass.

considers only a subset of the optimizations used in image perforation. In all of the cases we studied, image perforation outperforms loop perforation, often by orders of magnitude.

Several complementary optimizations could be combined with our approach. As discussed in Section 7.6, we performed preliminary experiments on loop transformations by exploring parallelization and vectorization over color channels, and found our speedups persisted under these optimizations. It would be interesting to combine our system with sophisticated loop scheduling optimizers such as Halide [Ragan-Kelley et al. 2013] since their optimizations are complementary to ours. We presented preliminary experiments showing that our method can be used to gain additional accelerations in Halide. However: challenges are also present, such as how one might best vectorize the regular sampling patterns produced by grid sampling, or the irregular access patterns produced by adaptive or importance sampling. Local program modifications [Sitthi-amorn et al. 2011] are another source of complementary optimizations. Our method performs more global modifications which change image sampling and reconstruction, but both kinds of mutations could potentially be useful in accelerating image pipelines. Finally, it may be beneficial to use reconstruction that is tailored to be efficient for our adaptive sampling pattern, so that adaptive sampling can be chosen even for pipelines that are already highly efficient.

The original loop perforation method could potentially cause errors to grow arbitrarily large through an arbitrary computation. In practice, our errors appear to be smaller when applied to image pipelines, due to the reconstruction step. However, an interesting area for future work would be to incorporate probabilistic accuracy bounds similar to Misailovic et al. [2011] or proofs that show programs robust to small perturbations [Chaudhuri et al. 2011].

Another avenue for future work is adding support for streaming or GPU implementations. Streaming facilities have been implemented in GPU languages such as Brook [Buck et al. 2004] and are particularly useful for video processing or processing very large images that cannot fit in memory. Extending image perforation to work with a GPU would require careful consideration of the sampling and reconstruction steps to achieve high performance. For example, our spatially varying Gaussian blur currently uses the method of repeated integration [Heckbert 1986], but this requires additional precision which might be a bad assumption on the GPU. In these cases, other approaches such as Gaussian pyramids [Adelson et al. 1984] might prove more efficient.

APPENDIX

A. INTENSITY REMAPPING OF IMPORTANCE MAPS

This appendix develops an intensity remapping technique that is used as part of importance sampling in Section 3.1. As input, we are given an importance map image v with n pixels indexed $0, \ldots, n-1$. What we would like is an output image v' such that simply quantizing v' using existing error-diffusion methods will result in the desired sample locations.

We have three goals in this remapping process. First, v might have very low or high importance values, so it is necessary to remap them using a global scale factor s. Additionally, we would like to produce output intensities that do not exceed 1, because our quantization should result in either the absence or presence of a sample (0 or 1, respectively). Finally, we would like to reach the target sample count fn, where f is the fraction of pixels to sample. We express these mathematically by setting $v'_i = \min(sv_i, 1)$. We want the target image to sum to the sample count, so we have the objective:

$$\sum_{i=0}^{n-1} \min(sv_i, 1) = fn \tag{2}$$

In theory, we could directly solve for the unknown scale factor s by using Newton's method on equation (2), and then use this to recover the image v'. However, that is inefficient and can take a large number of iterations. Our efficient approximation algorithm proceeds by binning the intensities v_i into a histogram with m bins, where the bin centers are x_i and the counts are h_i. We approximate equation (2) using the histogram:

$$\sum_{i=0}^{m-1} h_i \min(sx_i, 1) = fn \tag{3}$$

For any scale factor s we can choose a split point k such that the clamp operation is not applied before k and is applied after k. We have the constraints: $sx_k < 1$ and either $sx_{k+1} \geq 1$ or $k = m-1$. Thus equation (3) becomes:

$$s\sum_{i=0}^{k} h_i x_i + \sum_{k+1}^{m-1} h_i = fn \tag{4}$$

Now we define the cumulative sums $H_k = \sum_{i=0}^{k} h_i$ and $X_k = \sum_{i=0}^{k} h_i x_i$. These can be precomputed along with the histogram. We solve equation (4) to obtain s_k, which is a potential solution for the scale factor s, indexed by the split point k that has been chosen:

$$s_k = \frac{H_k - n(1-f)}{X_k} \tag{5}$$

Our algorithm determines the final scale factor s by enumerating $k = 0, \ldots, m-1$ and returning any s_k such that the constraints hold. In practice, to handle round-off error, we add a small machine epsilon and check whether $sx_k < 1 + \epsilon$ and $sx_{k+1} \geq 1 - \epsilon$. Once the scale factor is known we then evaluate the remapped image v' and proceed to dither it.

The time complexity of this algorithm is $O(n+m)$. We empirically determined by simulations that the error in s is inversely proportional to the number of histogram bins. In practice, we choose $m = 1,000$ bins which gave a maximum relative error of 0.5% for our simulated data.

REFERENCES

ADELSON, E. H., ANDERSON, C. H., BERGEN, J. R., BURT, P. J., AND OGDEN, J. M. 1984. Pyramid methods in image processing. *RCA engineer 29,* 6, 33–41.

AGARWAL, S., RAMAMOORTHI, R., BELONGIE, S., AND JENSEN, H. W. 2003. Structured importance sampling of environment maps. *ACM Transactions on Graphics 22,* 3, 605–612.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA.

ALCÁZAR, A. I. E., ALCZAR, A. I. E., AND SHARMAN, K. C. 1996. Some applications of genetic programming in digital signal processing.

ANSEL, J., KAMIL, S., VEERAMACHANENI, K., O'REILLY, U., AND AMARASINGHE, S. 2013. Opentuner: An extensible framework for program autotuning.

BAE, S., PARIS, S., AND DURAND, F. 2006. Two-scale tone management for photographic look. In *ACM Transactions on Graphics (TOG).* Vol. 25. ACM, 637–645.

BANTERLE, F., CORSINI, M., CIGNONI, P., AND SCOPIGNO, R. 2012. A low-memory, straightforward and fast bilateral filter through subsampling in spatial domain. *Computer Graphics Forum 31,* 1 (February), 19–32.

BISHOP, G., FUCHS, H., MCMILLAN, L., AND ZAGIER, E. J. S. 1994. Frameless rendering: Double buffering considered harmful. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '94. ACM, New York, NY, USA, 175–176.

BLACK, G. M. R. B. 2009. Yuv color space. *Communications Engineering Desk Reference*, 469.

BRADY, A., LAWRENCE, J., PEERS, P., AND WEIMER, W. 2014. genbrdf: Discovering new analytic brdfs with genetic programming. *ACM Transactions on Graphics (Proc. SIGGRAPH)*.

BUCK, I. 2007. Gpu computing: Programming a massively parallel processor. *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society*.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*. Vol. 23. ACM, 777–786.

CHAN, C., ANSEL, J., WONG, Y. L., AMARASINGHE, S., AND EDELMAN, A. 2009. Autotuning multigrid with petabricks. 5:1–5:12.

CHAUDHURI, S., GULWANI, S., LUBLINERMAN, R., AND NAVIDPOUR, S. 2011. Proving programs robust. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 102–112.

CHEN, J., PARIS, S., AND DURAND, F. 2007. Real-time edge-aware image processing with the bilateral grid. *ACM Trans. Graph. 26,* 3 (July).

CHRISTEN, M., SCHENK, O., AND BURKHART, H. 2011. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. 676–687.

DAMEZ, C., DMITRIEV, K., AND MYSZKOWSKI, K. 2003. State of the art in global illumination for interactive applications and high-quality animations. *Computer Graphics Forum 22,* 1, 55–77.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

FARBMAN, Z., FATTAL, R., AND LISCHINSKI, D. 2011. Convolution pyramids. *ACM Trans. Graph. 30,* 6, 175:1–175:8.

FLOYD, R. AND STEINBERG, L. 1976. An adaptive algorithm for spatial grey scale. *Proceedings of the Society of Information Display 17*, 75–77.

GUENTER, B. AND NEHAB, D. 2010. The neon image processing language. *Tech. Rep. MSR-TR-2010-175, Microsoft Research*.

HE, Y., GU, Y., AND FATAHALIAN, K. 2014. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Transactions on Graphics (TOG) 33,* 4, 142.

HECKBERT, P. S. 1986. Filtering by repeated integration. *ACM SIGGRAPH Computer Graphics 20,* 4, 315–321.

KANG, H., LEE, S., AND CHUI, C. K. 2009. Flow-based image abstraction. *Visualization and Computer Graphics, IEEE Transactions on 15,* 1, 62–76.

KELLER, A. 1997. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

KOZA, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

MISAILOVIC, S., ROY, D. M., AND RINARD, M. C. 2011. Probabilistically accurate program transformations. In *Static Analysis*. Springer, 316–333.

NEHAB, D., MAXIMO, A., LIMA, R. S., AND HOPPE, H. 2011. Gpu-efficient recursive filtering and summed-area tables. In *ACM Transactions on Graphics (TOG)*. Vol. 30. ACM, 176.

OPENCL. 2013. The OpenCL specification, version 2.0. *Khronos Group*.

OSTROMOUKHOV, V. 2001. A simple and efficient error-diffusion algorithm. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 567–572.

PARIS, S., HASINOFF, S. W., AND KAUTZ, J. 2011. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *ACM Trans. Graph. 30,* 4 (July), 68:1–68:12.

PÁRRAGA, C. A., BRELSTAFF, G., TROSCIANKO, T., AND MOOREHEAD, I. R. 1998. Color and luminance information in natural scenes. *Journal of the Optical Society of America 15,* 3, 563–569.

PAVLIDIS, T. 2012. *Algorithms for graphics and image processing*. Springer Science & Business Media.

PIXELBENDER. 2010. Adobe PixelBender reference. *Adobe*.

RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. 2013. Halide: A language and compiler for optimizing parallelism, locality and recomputation in image processing pipelines. *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

RAMAMOORTHI, R., MAHAJAN, D., AND BELHUMEUR, P. 2007. A first-order analysis of lighting, shading, and shadows. *ACM Trans. Graph. 26,* 1 (Jan.).

ROSENFELD, A. AND PFALTZ, J. L. 1968. Distance functions on digital pictures. *Pattern recognition 1,* 1, 33–61.

SHANTZIS, M. 1994. A model for efficient and flexible image computing. *ACM Transactions on Graphics (Proc. SIGGRAPH)*.

SHARMAN, K., ALCAZAR, A., AND LI, Y. 1998. Evolving signal processing algorithms by genetic programming. In *Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414)*. 473–480.

SIDIROGLOU-DOUSKOS, S., MISAILOVIC, S., HOFFMANN, H., AND RINARD, M. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium*. ACM, 124–134.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic programming for shader simplification. *ACM Transactions on Graphics 24,* 111, 647.

TOMASI, C. AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In *Proceedings of the International Conference on Computer Vision (ICCV)*. 839–846.

UESAKA, K. 2003. Evolutionary synthesis of digital filter structures using genetic programming. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing 50,* 12.

UESAKA, K. AND KAWAMATA, M. 2000. Synthesis of low-sensitivity second-order digital filters using genetic programming with automatically defined functions. *Signal Processing Letters, IEEE 7,* 4, 83–85.

VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., ET AL. 2014. Coarse pixel shading. In *High Performance Graphics*.

WEI, L.-Y. 2008. Parallel poisson disk sampling. In *ACM Transactions on Graphics (TOG)*. Vol. 27. ACM, 20.

YANG, L., SANDER, P. V., AND LAWRENCE, J. 2008. Geometry-aware framebuffer level of detail. In *Eurographics Symposium on Rendering (EGSR)*.

YOUNG, I. T. AND VAN VLIET, L. J. 1995. Recursive implementation of the gaussian filter. *Signal processing 44,* 2, 139–151.

YU, Y. AND XINJIE, Y. 2007. Cooperative coevolutionary genetic algorithm for digital iir filter design. *Industrial Electronics, IEEE Transactions on 54,* 3, 1311–1318.

Received September 2015; accepted Month ZZZ