A Hardware-Friendly Bilateral Solver for Real-Time Virtual Reality Video

Amrita Mazumdar
Armin Alaghi
University of Washington

Jonathan T. Barron
David Gallup
Google

Luis Ceze
Mark Oskin
Steven M. Seitz
University of Washington

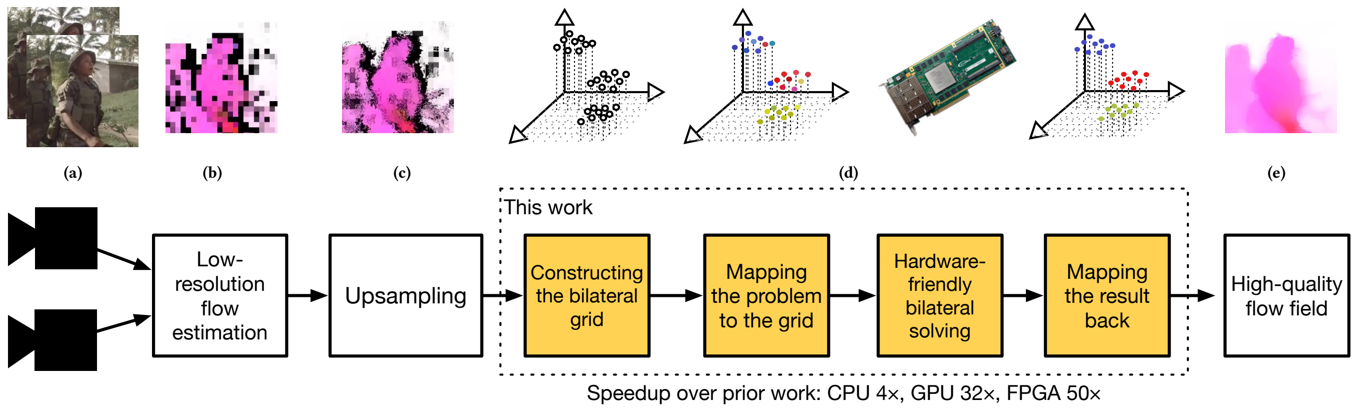

(a)
(b)
(c)
(d)
(e)
This work
Low-resolution flow estimation
Upsampling
Constructing the bilateral grid
Mapping the problem to the grid
Hardware-friendly bilateral solving
Mapping the result back
High-quality flow field
Speedup over prior work: CPU 4×, GPU 32×, FPGA 50×

Figure 1: Our bilateral solver produces smooth, edge-aware flow fields. Given an input pair of images (a), a low-resolution flow is estimated (b), upsampled to a noisy high-resolution flow (c), and processed with the bilateral solver (d) to produce an edge-aware smoothed flow (e). Our algorithm for bilateral solving is better-suited for hardware acceleration and results in speedups of up to 50× over prior work [2, 3].

ABSTRACT

Rendering 3D-360° VR video from a camera rig is computation-intensive and typically performed offline. In this paper, we target the most time-consuming step of the VR video creation process, high-quality flow estimation with the bilateral solver. We propose a new algorithm, the hardware-friendly bilateral solver, that enables faster runtimes than existing algorithms of similar quality. Our algorithm is easily parallelized, achieving a 4× speedup on CPU and 32× speedup on GPU over a baseline CPU implementation. We also design an FPGA-based hardware accelerator that utilizes reduced-precision computation and the parallelism inherent in our algorithm to achieve further speedups over our CPU and GPU implementations while consuming an order of magnitude less power. The FPGA design's power efficiency enables practical real-time VR video processing at the camera rig or in the cloud.

CCS CONCEPTS

•**Hardware** →**Hardware accelerators;** •**Computing methodologies** →*Graphics processors; Virtual reality;*

KEYWORDS

Hardware accelerators, parallelism, FPGA design, GPU algorithm, real-time image processing, virtual reality.

ACM Reference format:
Amrita Mazumdar, Armin Alaghi, Jonathan T. Barron, David Gallup, Luis Ceze, Mark Oskin, and Steven M. Seitz. 2017. A Hardware-Friendly Bilateral Solver for Real-Time Virtual Reality Video. In *Proceedings of HPG '17, Los Angeles, CA, USA, July 28-30, 2017,* 10 pages.
DOI: 10.1145/3105762.3105772

1 INTRODUCTION

Virtual reality (VR) devices are becoming widely available, from camera rigs for video capture [Anderson et al. 2016; Facebook 2017], to headsets for immersive viewing [Google 2017; Oculus 2017; Samsung 2017]. Real-time rendering of 3D-360° video can enable a wide range of VR applications, from live sports and concert broadcasting to telepresence. While the domains of VR video content capture and viewing are growing more popular, no system is capable of producing 3D-360° VR videos in real time as of yet.

The Google Jump camera rig [Anderson et al. 2016] is one example of a commodity VR video capture device, using 16 cameras to capture high-resolution (4K-1080p) overlapping video streams of a 360° scene. The collected video is used to estimate edge-aware flow fields, which are composited into a stereoscopic 3D-360° video. A key portion of this processing pipeline, flow estimation, is constructed around the *bilateral solver*, a fast and edge-aware algorithm that combines simple bilateral filters with domain-specific optimization problems [Barron and Poole 2016]. This flow estimation algorithm, highlighted in Figure 1, consumes the majority of the processing time, despite using one of the fastest existing algorithms across thousands of cores [Anderson et al. 2016].

In this paper, we introduce a new algorithm for this flow estimation problem, the hardware-friendly bilateral solver (HFBS). HFBS achieves significant speedups over the bilateral solver with little accuracy loss. While Barron and Poole's bilateral solver [2016] is challenging to parallelize on modern hardware, our "hardware-friendly" algorithm can be easily parallelized on GPUs and field-programmable gate arrays (FPGAs). To demonstrate, we design a scalable FPGA-based hardware accelerator for HFBS, employing specialized memory layout and reduced-precision fixed-point computation to achieve real-time results. Compared to the original bilateral solver, HFBS is 4× faster on a CPU, 32× faster on a GPU, and 50× faster on an FPGA. We evaluate the accuracy of HFBS on the depth superresolution task and show that our algorithm is faster than every more accurate algorithm, and more accurate than any faster algorithm.

This paper makes two contributions: an algorithm for hardware-friendly bilateral solving, and a fixed-function FPGA accelerator implementing HFBS. To achieve fast performance while maintaining accuracy, we take a hardware-software codesign approach where both the algorithm and hardware substrate are developed in tandem. Our algorithm modifies the original bilateral solver to ensure memory access is predictable and therefore fast, and performs optimization using preconditioned gradient descent with momentum to reduce global communication and enable parallel execution. Our hardware accelerator explores fixed-point arithmetic and bilateral-grid-specialized memory layout to process large-resolution bilateral grids in a scalable way in real time. Many of these performance optimizations are codependent, and we evaluate performance of the algorithm and hardware together to illustrate our results. Our algorithm and accelerator design make it more practical to generate real-time VR video from camera rigs, either locally at the capture device, or in the cloud to accelerate large-scale video processing.

2 BACKGROUND

Before formalizing our hardware-friendly bilateral solver, we provide an overview on bilateral solving and its role in VR video. We also describe related work in software and hardware acceleration for bilateral solving.

2.1 Bilateral Filtering and the Bilateral Grid

We base our design for fast and accurate VR video on a state-of-the-art bilateral-space optimization algorithm, the bilateral solver [Barron and Poole 2016]. The bilateral solver is general-purpose and scalable, and can be applied to the many vision applications: optical flow, stereo, depth superresolution, image colorization, and semantic segmentation. The bilateral solver can be used as part of an edge-aware optical flow algorithm for VR video, and scales to high resolutions efficiently [Anderson et al. 2016]. This optical flow algorithm generates a correspondence map from a pair of images by computing a rough flow vector for every pixel (Figure 1b-c), and then refining that flow field until a cost function has been minimized. To compute this edge-aware per-pixel flow field, the bilateral solver resamples a coarse flow field into *bilateral-space* (Figure 1d), and then solves an optimization problem in bilateral-space to infer the smoothest possible flow-field that is similar to the input coarse flow field. In bilateral-space, simple local filters are equivalent to costly, global, edge-aware filters in pixel-space—consequently, flow refinement in bilateral-space is much faster than its pixel-space equivalent. We perform optimization in a three-dimensional *bilateral grid* data structure [Chen et al. 2007].

Figure 2 illustrates a simplified version of bilateral space and its use in our problem. We begin with the noisy flow field of Figure 2a-i, where color corresponds to some flow value. If we attempt to denoise this noisy flow field by applying a simple smoothing kernel, the result will present undesirable blurring at color edges. In Figure 2a-ii, for instance, the green region is successfully denoised, but the blue and red regions (which likely belong to different objects) blend around the edges, producing incorrect flow values there.

To smooth this flow field while maintaining sharp edges, we map the problem to bilateral space. First, we construct a *bilateral grid* for the original image, where a pixel in the image at location (x, y) with luminance l corresponds to a grid block at location (x, y, l) (Figure 2b-i, b-ii). In the 3D bilateral space, the lighter pixels are separated from neighboring darker pixels. We then map the flow value of each pixel (Figure 2b-iii) to its corresponding grid location (Figure 2b-iv). When we smooth this noisy flow in bilateral space, the blue and red areas are no longer neighbors and do not affect each other's value. Finally, we map the smoothed 3D flow (Figure 2b-v) back to the 2D representation (Figure 2b-vi). The resulting bilateral-smooth output in Figure 2a-vi retains sharp edges.

Virtual reality video with the bilateral solver. We tailored our algorithm for a VR pipeline (similar to that of Anderson et al. [2016]), which takes 16 camera streams as input and processes them with the bilateral solver to construct 3D-360° video. There are many other ways to capture and render VR video, but each method presents unique challenges. Light fields [Levoy and Hanrahan 1996], a type of image that conveys information about the flow of light in a scene, are the most general and immersive solution to VR imagery. Proposed light field-based systems, however, require an enormous amount of input and output data, and rendering on the client side is compute-intensive. While impressive results for light field images have been demonstrated in VR [Huang et al. 2015], video is a greater challenge and still impractical. Other solutions for immersive video viewing, such as free-viewpoint video [Carranza et al. 2003] and concentric mosaics [Shum et al. 2005], are also challenging to process and display using standard video formats, resulting in systems that are not yet stable enough to motivate hardware support. In contrast, the Jump VR video system is designed to be practical to compute, edit, and stream. Processing the 16 camera video array requires computing optical flow between images from each adjacent pair of cameras and then interpolating the images to produce the *omnidirectional stereo* projection [Peleg et al. 2001]. The output is a pair of equirectangular spherical video streams, one stream for the

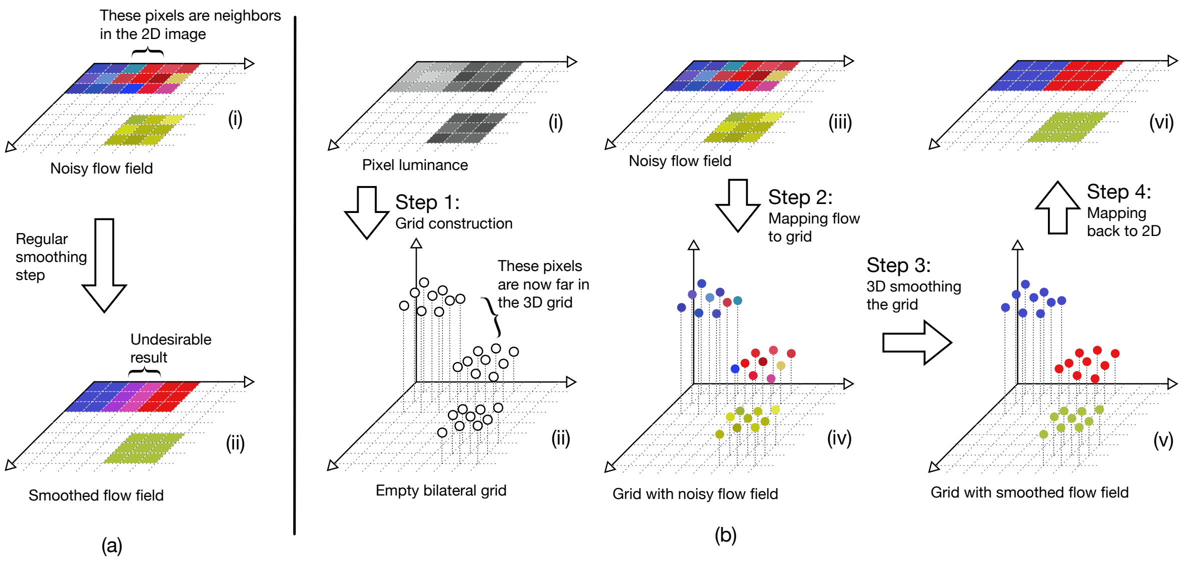

These pixels are neighbors in the 2D image
(i)
Noisy flow field
Regular smoothing step
Undesirable result
(ii)
Smoothed flow field
(a)
(i)
Pixel luminance
Step 1: Grid construction
These pixels are now far in the 3D grid
(ii)
Empty bilateral grid
(iii)
Noisy flow field
Step 2: Mapping flow to grid
(iv)
Grid with noisy flow field
Step 3: 3D smoothing the grid
Step 4: Mapping back to 2D
(vi)
(v)
Grid with smoothed flow field
(b)

Figure 2: Smoothing a noisy flow field in (a) regular 2D space and (b) bilateral space. Regular smoothing produces undesirable artifacts at edges as the flow values blur together. The bilateral grid allows edge-aware smoothing and produces a correct denoised output.

left eye and one stream for the right. Anderson et al. [2016] employ a bilateral-space solver for optical flow to efficiently produce high quality edge-aware flow results that are well-suited to omnidirectional stereo image interpolation. They observe that the majority of rendering time is spent running the bilateral solver.

2.2 Hardware Acceleration for Bilateral Grids

This work is the first, to our knowledge, to accelerate the bilateral solver on GPUs or with custom hardware, and builds on related work in hardware-efficient algorithms and accelerators for bilateral filtering and the bilateral grid. The bilateral grid itself was originally proposed as a solution for fast, parallelizable bilateral filtering on GPUs [Chen et al. 2007]. Towards more hardware-efficient execution on GPUs, Yang [2014] proposed a hierarchical bilateral filter technique, but their approach has much higher error than our algorithm. Most similar to our accelerator is that of Rithe et al. [2013], who designed a low-power reconfigurable processor for bilateral filtering. Their design differs from ours by implementing splatting and slicing in hardware, but it can only perform streaming bilateral filters and does not support the repeated filtering iterations necessary for bilateral-space optimization.

3 HARDWARE-FRIENDLY BILATERAL SOLVING

In this section, we formulate a bilateral solver that maintains speed, scalability, and accuracy, while also being parallelizable. We first describe the original bilateral solver of [Barron and Poole 2016], and motivate the requirements for a hardware-friendly bilateral solver. We then provide a detailed formulation of our algorithm and its advantages.

3.1 Bilateral-Space Optimization

The original bilateral solver (OBS) consists of an objective and optimization technique [Barron and Poole 2016]. The input to the solver is a reference RGB image, a target image that contains noisy observed quantities we wish to improve, and a confidence image. The goal is to recover an "output" vector $\mathbf{x}$, which will resemble the input target where the confidence is large while being smooth and tightly aligned to edges in the reference image. To achieve this, Barron and Poole construct an optimization problem of the following form:

$$\underset{\mathbf{x}}{\text{minimize}} \ \frac{\lambda}{2} \sum_{i,j} \hat{W}_{i,j} \left(x_i - x_j\right)^2 + \sum_i c_i (x_i - t_i)^2 \tag{1}$$

The first term of the loss encourages that for all pixel pairs i and j, the overall difference between their flow values x_i and x_j is minimized if they are neighboring pixels in the bilateral space. The second term of Eq. 1 encourages each pixel x_i to be close to the target input t_i if that pixel's confidence c_i is high.

The affinity matrix $\hat{\mathbf{W}}$ is a *bistochastized* (all rows and columns sum to 1) version of a bilateral affinity matrix $\mathbf{W}$. Each element of the bilateral affinity matrix $W_{i,j}$ describes the affinity between pixels i and j in the reference image in the YUV colorspace:

$$W_{i,j} = \exp\left(- \frac{\left\| \left[p_i^x, p_i^y\right] - \left[p_j^x, p_j^y\right] \right\|^2}{2\sigma_{xy}^2} - \frac{\left(p_i^l - p_j^l\right)^2}{2\sigma_l^2} - \frac{\left\| \left[p_i^u, p_i^v\right] - \left[p_j^u, p_j^v\right] \right\|^2}{2\sigma_{uv}^2} \right) \tag{2}$$

where p_i is a pixel in the reference image with location (p_i^x, p_i^y) and color (p_i^l, p_i^u, p_i^v). The σ_{xy}, σ_l, and σ_{uv} parameters control the support of the spatial, luminance (luma), and chrominance (chroma) components of the filter. Bistochastization normalizes this affinity matrix while maintaining symmetry [Barron et al. 2015].

Bilateral operations (e.g., filtering) can be sped up by treating the filter as a "splat/blur/slice" procedure in the bilateral grid. The splat/blur/slice filtering approach corresponds to a compact factorization of $\mathbf{W}$:

$$\mathbf{W} = \mathbf{S}^{\mathrm{T}}\mathbf{B}\mathbf{S} \tag{3}$$

where $\mathbf{S}$ and $\mathbf{S}^{\mathrm{T}}$ are splatting and slicing, and $\mathbf{B}$ is a [1 2 1] blur kernel. As in Barron and Poole [2016], $\mathbf{S}$ defines a per-pixel mapping from a pixel to a coarse bin in the bilateral grid, where that mapping is a function of the x and y coordinates, l luma, u and v chroma of that pixel. Multiplying by $\mathbf{S}$ is a data-dependent histogramming operation, and multiplying by $\mathbf{S}^{\mathrm{T}}$ is a data-dependent interpolation. The bilateral space optimization formulation of Barron et al. [2015] performs bistochastization by calculating two matrices $\mathbf{m}$ and $\mathbf{n}$ that satisfy the following:

$$\hat{\mathbf{W}} = \mathbf{S}^{\mathrm{T}}\,\mathrm{diag}\left(\frac{\mathbf{n}}{\mathbf{m}}\right)\mathbf{B}\,\mathrm{diag}\left(\frac{\mathbf{n}}{\mathbf{m}}\right)\mathbf{S} \tag{4}$$

where $\hat{\mathbf{W}}$ is a bistochastic version of matrix $\mathbf{W}$. The vectors $\mathbf{m}$ and $\mathbf{n}$ describe a normalizing transformation required by the solver.

Barron and Poole also perform a variable substitution [2016], transforming the high-dimensional pixel-space optimization problem into one with lower-dimensional bilateral-space vertices:

$$\mathbf{x} = \mathbf{S}^{\mathrm{T}}\mathbf{y} \tag{5}$$

where $\mathbf{y}$ is a small vector of values for each bilateral grid vertex, and $\mathbf{x}$ is the large vector of values for each pixel.

Equations 3 and 5 allow us to reformulate the pixel-space loss function of Eq. 1 into bilateral-space in a quadratic form:

$$\begin{aligned}
&\underset{\mathbf{y}}{\text{minimize}} \quad \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{A}\mathbf{y} - \mathbf{b}^{\mathrm{T}}\mathbf{y} + c \qquad (6)\\
&\mathbf{A} = \lambda(\mathrm{diag}(\mathbf{m}) - \mathrm{diag}(\mathbf{n})\mathbf{B}\,\mathrm{diag}(\mathbf{n})) + \mathrm{diag}(\mathbf{S}\mathbf{c})\\
&\mathbf{b} = \mathbf{S}(\mathbf{c} \circ \mathbf{t}) \qquad c = \frac{1}{2}(\mathbf{c} \circ \mathbf{t})^{\mathrm{T}}\mathbf{t}
\end{aligned}$$

where $\mathbf{y}$ is the solution to the problem in bilateral-space, $\mathbf{m}$ and $\mathbf{n}$ are defined by Eq. 4, and $\mathbf{t}$ and $\mathbf{c}$ are per pixel initial solutions and confidences (Eq. 1). The Hadamard (element-wise) product is denoted by $\circ$.

The optimization problem of Eq. 1 is intractably slow to solve naively. However, the bilateral-space formulation allows feasible and fast execution. Minimizing Eq. 6 is equivalent to solving a sparse linear system:

$$\mathbf{A}\mathbf{y} = \mathbf{b}$$

and we can produce a pixel-space solution $\hat{\mathbf{x}}$ by slicing out the solution from the linear system:

$$\hat{\mathbf{x}} = \mathbf{S}^{\mathrm{T}}(\mathbf{A}^{-1}\mathbf{b}) \tag{7}$$

In summary, OBS takes an input image vector and a confidence image to construct a simplified bilateral grid from the reference image. With that, it produces the $\mathbf{A}$ matrix and $\mathbf{b}$ vector of Eq. 6 to solve the linear system in Eq. 7 and obtain an output image.

3.2 Algorithmic Modifications

Though computationally efficient, OBS as presented has a number of properties that make it difficult to implement in hardware, or even to achieve real-time operation on modern CPU or GPU systems. Vectorizing and parallelizing CPU or GPU processing on the sparse 5D bilateral grid $\hat{\mathbf{W}}$ demonstrates too-irregular memory access patterns to achieve large performance benefits from parallelization. Moreover, the use of second order global optimization limits the level of parallelism we can extract from the algorithm. We modify OBS to construct a hardware-friendly bilateral solver and address these specific challenges: color and sparse memory indexing, and second order global optimization. Our modifications also allow for an alternative, more efficient initialization and reduced quantization artifacts, which we will discuss after formulating our algorithm.

Color and sparse memory indexing. The bilateral solver of Barron and Poole [2016] was designed around a hard bilateral grid or a permutohedral lattice [Adams et al. 2010], meaning that optimization takes place in a "sparse" five-dimensional bilateral space (where the five dimensions are position in x and y, pixel luma, and two pixel chroma values). The resulting 5D grid has an image-dependent "sparsity" that is challenging to exploit in parallel algorithms. Moreover, the connectivity structure of the graph used in the bilateral solver varies as a function of the input, leading to expensive and unpredictable memory access patterns. Attempting to resolve this by solely converting the sparse grid into a "dense" representation of the 5D space requires a prohibitive amount of memory. Instead, HFBS ignores the color of the input image and uses a "dense" 3D bilateral grid [Chen et al. 2007], which makes memory indexing predictable and enables further optimizations. Ignoring color this way induces a small decrease in accuracy, as we will demonstrate.

Second order global optimization. The numerical optimization in OBS was performed using the preconditioned conjugate gradient method with a Jacobi or Jacobi-like hierarchical preconditioner. Conjugate gradient methods use a global optimization step: at each iteration, updating each variable of the optimization vector requires reasoning about the gradient at all other variables. Such global communication requirements make parallel hardware implementation difficult, as we want to be able to individually update and optimize any variable in our state space via local communication with the "neighboring" variables in our bilateral grid. To avoid global communication, HFBS performs optimization using gradient descent with momentum (i.e., the "Heavy Ball" algorithm), which can be shown to have similar asymptotic performance as conjugate gradient [Polyak 1964]. This converts an irregular number of global matrix operations into a regular, but larger, number of local updates that are much easier to execute in parallel.

The Heavy Ball algorithm does not naturally accommodate a preconditioner, so we reformulate our optimization problem with a transformation that indirectly applies a Jacobi preconditioner during optimization. We find that HFBS slightly underperforms the preconditioned conjugate gradient solver of Barron and Poole [2016] and therefore requires roughly twice as many steps for convergence. However, since each step is significantly faster to compute (roughly 4× faster on CPU and much faster on GPU/FPGA), we see an overall increase in performance.

3.3 Algorithm Formulation

We now formalize the details of HFBS and how it relates to the original bilateral solver. Both OBS and HFBS minimize an optimization problem of the form of Eq. 1. In this case, the t_i is the low resolution flow shown in Figure 1b. We derive the confidence image for these low resolution flow fields by computing normalized sum-of-squared differences. The resulting confidence is larger for areas that are near each other and match well.

We obtain the weight $\hat{W}_{i,j}$, which determines the bilateral-space distance between two pixels i and j, from a bistochastized version of the matrix $\mathbf{W}$ whose elements are calculated via the following:

$$W_{i,j} = \exp\left(-\frac{\left\| [[p_i^x, p_i^y] - [p_j^x, p_j^y]] \right\|^2}{2\sigma_{xy}^2} - \frac{(p_i^l - p_j^l)^2}{2\sigma_l^2} \right) \tag{8}$$

where each pixel p_i has a spatial position (p_i^x, p_i^y) and luminance p_i^l. While OBS includes color information in $\hat{W}_{i,j}$ (Eq. 2), HFBS only considers luminance.

The bistochastization step in Barron and Poole [2016] requires 10-20 iterations to achieve low error. To reduce the fixed cost of this step, we use a faster, approximate bistochastization step for initializing the bilateral solver. Unlike OBS, which fully-bistochastizes $\mathbf{W}$ into $\hat{\mathbf{W}}$, we construct an approximately bistochastized $\hat{\mathbf{W}}$ (equivalent to one iteration of bistochastization) that still satisfies the requirements of the bilateral solver:

$$\mathbf{m}_0 = \mathbf{S1} \qquad \mathbf{n} = \sqrt{\frac{\epsilon + \mathbf{m}_0}{\epsilon + \mathbf{B1}}} \qquad \mathbf{m}_1 = \mathbf{n} \circ (\mathbf{Bn}) \tag{9}$$

In OBS, bistochastization is done to convergence, which produces a $\mathbf{n}$ which satisfies $\mathbf{m}_0 = \mathbf{n} \circ (\mathbf{Bn})$. Partial bistochastization requires that we treat this equality as an assignment, thereby constructing $\mathbf{m}_1$ to explicitly obey this constraint (Eq. 9). This produces nearly-indistinguishable output while being faster and easier to compute.

Our normalization also differs from OBS by the use of $\epsilon \sim 0.00001$ in the construction of $\mathbf{n}$. Adding ϵ to the numerator prevents divide-by-zero later and ensures that empty grid cells do not propagate information during optimization. Adding it to the denominator prevents the addition of ϵ in the numerator from biasing the solution towards 0. Note that the partial bistochastization step of HFBS is not iterative and does not require any convergence, and thus is significantly faster than the bistochastization step of OBS.

As described earlier, the expensive per-pixel optimization in Eq. 1 can be reformulated to a much more tractable optimization problem inside a bilateral grid. For convenience we will define $\mathbf{By}$ (the product of some grid $\mathbf{y}$ with a blur $\mathbf{B}$) as a scaling of and "diffusion" of $\mathbf{y}$:

$$\begin{aligned}\mathbf{By} &= 2\mathbf{y} + \mathbf{Dy} \\ \mathbf{Dy} = D(\mathbf{y}) &= \mathbf{y}(x+1, y, z) + \mathbf{y}(x-1, y, z) \\ &+ \mathbf{y}(x, y+1, z) + \mathbf{y}(x, y-1, z) \\ &+ \mathbf{y}(x, y, z+1) + \mathbf{y}(x, y, z-1)\end{aligned}$$

where $\mathbf{D}$ is a diffusion operator (which we can interchangeably refer to as a matrix and a function) that replaces each element in $\mathbf{y}$ with the sum of its neighbors. Because our 3D bilateral grid is dense in memory, this diffusion process is a simple stencil operation.

We now perform a variable substitution, as in Eq. 5. For us, this simply requires dividing by the square root of the diagonal of the $\mathbf{A}$ matrix:

$$\mathbf{y} = \mathbf{p} \circ \hat{\mathbf{z}} \qquad \mathbf{p} = \frac{1}{\sqrt{\mathbf{Sc} + \lambda\left(\mathbf{m}_1 - 2(\mathbf{n} \circ \mathbf{n})\right)}}$$

where $\hat{\mathbf{z}}$ is the solution to the substituted problem.

With our variable substitution in place, we can reformulate Eq. 6:

$$\begin{aligned}&\underset{\mathbf{z}}{\text{minimize}} \quad \frac{1}{2}\mathbf{z}^{\mathrm{T}}\tilde{\mathbf{A}}\mathbf{z} - \tilde{\mathbf{b}}^{\mathrm{T}}\mathbf{z} + c \\ &\tilde{\mathbf{A}} = \mathbf{I} - \operatorname{diag}(\mathbf{q})\,\mathbf{D}\operatorname{diag}(\mathbf{q}) \\ &\tilde{\mathbf{b}} = \mathbf{p} \circ (\mathbf{S}(\mathbf{c} \circ \mathbf{t})) \\ &\mathbf{q} = \sqrt{\lambda}\,(\mathbf{n} \circ \mathbf{p})\end{aligned}$$

Here c is the same as in Eq. 6. Note that the diagonal of $\tilde{\mathbf{A}}$ is $\mathbf{1}$, so optimizing this problem without a preconditioner is the same as optimizing Eq. 6 with a Jacobi preconditioner. Minimizing this problem requires solving a linear system, undoing our preconditioning variable substitution, and then slicing out a solution:

$$\hat{\mathbf{z}} = \tilde{\mathbf{A}}^{-1}\tilde{\mathbf{b}} \qquad \hat{\mathbf{x}} = \mathbf{S}^{\mathrm{T}}(\mathbf{p} \circ \hat{\mathbf{z}})$$

We will solve this problem using the "Heavy Ball" method (gradient descent with momentum). This problem is fully-described by the diffusion operator $D(\cdot)$ and the bilateral grids $\tilde{\mathbf{b}}$ and $\mathbf{q}$.

Algorithm 1 shows pseudocode describing how optimization is performed. It can be shown that if the momentum and step size hyperparameters are set correctly, this heavy ball method has the same asymptotic performance as conjugate gradient [Polyak 1964]. Because preconditioning has been absorbed into the problem, performance approaches preconditioned conjugate gradient. Since the diffusion operator $D(\cdot)$ is a local stencil, the gradient update to $\mathbf{g}$ and the optimization update to $\mathbf{h}$ and $\mathbf{z}$ can be performed efficiently (i.e., vectorized, parallelized, etc).

Algorithm 1 Bilateral-Space Heavy Ball Method
Input: problem description $\{D(\cdot), \tilde{\mathbf{b}}, \mathbf{q}\}$, initial state $\mathbf{z}_{\text{init}}$, step size $\alpha = 1$, momentum $\beta = 0.9$, number of iterations $n = 256$.
Output: state after n iterations $\mathbf{z}$

```
1: $\mathbf{z} \leftarrow \mathbf{z}_{\text{init}}$
2: $\mathbf{h} \leftarrow \mathbf{0}$
3: for $i = 1 : n$ do
4:     $\mathbf{g} \leftarrow \mathbf{z} - \mathbf{q} \circ D(\mathbf{q} \circ \mathbf{z}) - \tilde{\mathbf{b}}$
5:     $\mathbf{h} \leftarrow \beta\mathbf{h} + \mathbf{g}$
6:     $\mathbf{z} \leftarrow \mathbf{z} - \alpha\mathbf{h}$
7: end for
```

Better Initialization to Reduce Optimization Iterations. Our objective function is convex and thus invariant to the initialization $\mathbf{z}_{\text{init}}$, but a better initialization may allow us to converge in fewer iterations. We can achieve this with a simple weighted blur in our bilateral grid.

$$\mathbf{z}_{\text{init}} = \frac{\text{blur}(\mathrm{S}(\mathbf{c} \circ \mathbf{t}), \sigma_b)}{\mathbf{p} \circ \text{blur}(\mathrm{S}(\mathbf{c}), \sigma_b)}$$

where $\text{blur}(\mathbf{a}, \sigma_b)$ is a large-support 3D Laplacian blur of $\mathbf{a}$ with a scale of σ_b:

$$\text{blur}(\mathbf{a}, \sigma_b)(t_x, t_y, t_z) = \iiint_{-\infty}^{\infty} e^{\left(\frac{-|\tau_x|-|\tau_y|-|\tau_z|}{\sigma_b}\right)} \mathbf{a}(t_x - \tau_x, t_y - \tau_y, t_z - \tau_z)\, d\tau_x d\tau_y d\tau_z$$

and t_x, t_y, and t_z are 3D coordinates. This can be efficiently implemented as three separable infinite impulse response filters (i.e., exponential smoothing, forward and backward) in the three dimensions of the grid. The intuition behind this initialization is that the solution should be close to **b** where the confidence is large and smooth where confidence is small. We found that this initialization can be implemented efficiently on a CPU and roughly halves the number of required iterations.

Reduced Quantization Artifacts. In OBS, slicing can introduce "blocky" quantization artifacts in the output [Barron and Poole 2016]. This quantization requires post-processing, adding to the computational cost of running the bilateral solver. However, HFBS uses a dense and low-dimensional grayscale bilateral grid which allows us to efficiently slice out of our bilateral grid using trilinear interpolation. As shown in Chen et al. [2007], this produces smooth results without post-processing. The trilinear interpolation can be done through a weighted slice, where $\mathbf{S}_{\text{tri}}$ is analogous to $\mathbf{S}$ but with trilinear weights instead of hard "one-hot" assignment:

$$\hat{\mathbf{x}} = \frac{\mathbf{S}_{\text{tri}}^{\mathrm{T}}\left(\mathbf{m}_0 \circ \mathbf{p} \circ \hat{\mathbf{z}}\right)}{\mathbf{S}_{\text{tri}}^{\mathrm{T}}\left(\mathbf{m}_0\right)}$$

By performing a weighted slice according to the per-vertex grid occupancy $\mathbf{m}_0$ this process produces artifact-free results in comparison to results from OBS, even if trilinear interpolation is not used in the splatting step. This "soft" slicing is only slightly more expensive than its "hard" equivalent, though both forms can be implemented very efficiently by virtue of being simple gather operations in a dense bilateral grid.

4 HARDWARE ARCHITECTURE

The formulation of HFBS allows for fast bilateral solving on high-performance CPUs or GPUs, but the resulting power consumption may prove prohibitively costly for a full system. FPGA platforms, on the other hand, can demonstrate fast performance with better power efficiency. This makes them a more suitable target for a system requiring multiple high-performance processors in a single chassis that can support processing 16-camera outputs simultaneously. To demonstrate power and performance efficiency on FPGAs, we co-designed our hardware implementation with the HFBS algorithm. In addition to the algorithmic optimizations, we apply hardware-specific techniques such as customized variable bitwidths and bilateral-space memory partitioning to enable better performance. We first discuss the hardware system at a high level, and then our specific design exploration for bitwidth precision and bilateral grid memory layout. Finally, we describe the hardware-software interface of our design and how we integrate the accelerator into an application.

4.1 Microarchitectural Design

We focus on executing the inner loop of Algorithm 1 with custom hardware, and maintaining the higher-level control flow in software. In this scheme, a software application splats the optimization problem defined in Section 3 onto a bilateral grid, and transfers it to the accelerator for iterative solving. Figure 3 shows a high-level overview of how application functionality is distributed across the system. The figure also illustrates details of our design's microarchitecture.

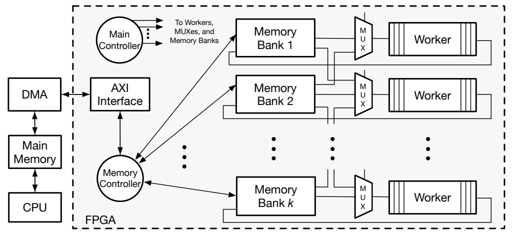


Figure 3: High-level system overview of our accelerator. Parallel workers process bilateral grid vertices stored in partitioned memory banks.

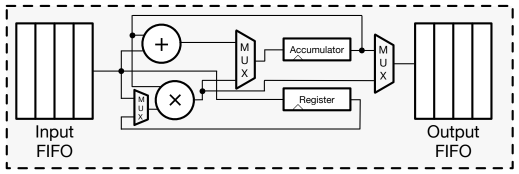


Figure 4: Block diagram of a single worker in our design.

Table 1: Worker resource utilization and maximum workers at varied bitwidths. Reported MSE is relative to 32-bit floating-point.

Bitwidth	32	47	64
DSPs per Worker	1	4	16
Maximum Workers	6840	1710	427
Min. MSE	8.30×10^{-4}	6.69×10^{-7}	7.16×10^{-13}

The CPU constructs the bilateral grid based on the input reference image and the initial low-resolution solution provided from prior steps. The transferred data includes $\hat{\mathbf{b}}$, $\mathbf{q}$, and the initial solution $\mathbf{z}_{\text{init}}$ shown in Algorithm 1. During transfer, the memory controller of Figure 3 interleaves the data corresponding to each bilateral grid vertex, and partitions the data into memory banks. After the data transfer is complete, a pool of parallel workers iteratively solve the optimization problem by running the loop of Algorithm 1. After some number of iterations (we chose 256 iterations for our experiments to ensure convergence), the CPU reads back the final solution and slices it into a 2D result.

Each worker (shown in in Figure 4) performs the inner loop of Algorithm 1 on one grid cell. It computes the result by streaming in the data from the neighboring cells (required for the "blur" operation), as well as the normalization factors required for the optimization process. The workers compute their local stencil operations synchronously, interfacing primarily with an assigned memory bank and occasionally the neighboring memory banks to access grid blocks that may be stored across banks. Because each worker executes in lockstep, there are no memory collisions when accessing data in other banks. Figure 3 demonstrates how multiplexers, managed by the main controller, shepherd access to neighboring banks. As we scale the number of workers, we find that parallelism introduces a 1% reduction in speedup against perfect linear scaling. This near-linear scaling can entirely be attributed to our inclusion of the "Heavy Ball" algorithm in HFBS, which allowed our design to use only local-neighbor communication rather than global synchronization after each iteration.

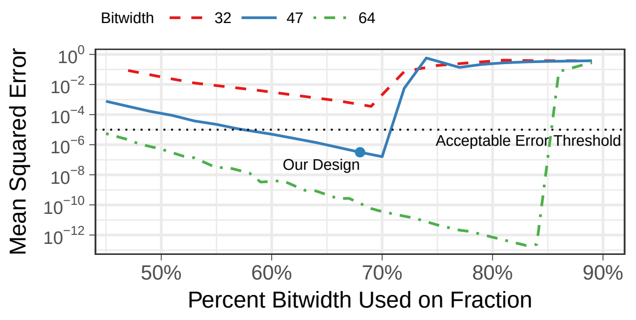


Figure 5: MSE of fixed-point implementations at varied fractional widths, for different bitwidths. MSE is reported relative to 32-bit floating-point. We chose a configuration with 31 bits of fractional precision to reduce chance of overflow in the integer portion.

4.2 Fixed-point Conversion

To improve resource utilization, we converted the algorithm from floating-point to fixed-point number representation. We first implemented our workers using single-precision floating-point, like our CPU and GPU implementations. We found that the large number of digital signal processing units (DSPs) required for a single floating-point multiplier prohibitively limited the number of workers we could employ, and consequently, the amount of parallelism. Converting FPGA designs from floating-point to fixed-point number representation resolves this by reducing the resource requirements of hardware multipliers. Using the cheaper fixed-point multiplier, however, required us to evaluate three competing tradeoffs: (1) the bitwidth of our fixed-point numbers, (2) the precision at a given bitwidth for the integer and fractional portions of the number, and (3) convergence of the solver. If less than 12 bits were used for the integer portion, the bilateral grid data would quickly populate with overflow values. If less than 24 bits were used for the fractional component of the number, the bilateral solver would not converge, because grid vertices would not have enough precision to capture the change in a value after blurring. These constraints prevented us from using 32-bit fixed-point numbers, as highlighted by the high mean squared error (MSE) shown in Figure 5 across integer-fraction-ratio configurations. We delineate a maximum error threshold of ~ 0.00001, because any errors exceeding that precision eliminate the positive benefits of using an ϵ-value to reduce zero-propagation. Using 64-bit fixed-point numbers resulted in very low MSE, but, as seen in Table 1, required 16 DSPs per worker, limiting the number of parallel workers we could deploy with these configurations.

As a compromise, we evaluated a 47-bit number representation that was more accurate than 32-bit fixed-point, with 75% less DSPs than 64-bit fixed-point. To maintain some precision of 64-bit numbers during non-multiplier arithmetic, we chose a 64-bit fixed-point representation with 15 bits of integer precision and 48 bits of fractional precision, and cast it to and from 47-bit for multiply operations only. Before multiplying two 64-bit numbers, we round off the bottom 16-bits of each number, resulting in the 1-bit sign, 15-bit integer, 31-bit fraction number highlighted in Figure 5. We zero-extend the resulting 47-bit output back to a 64-bit number for the rest of the computation. This fixed-point configuration has a MSE of 3.17×10^{-7} compared to the floating-point implementation, resulting in negligible accuracy loss at the solver output, and the solver converges at the same number of iterations.

4.3 On-Chip Bilateral Grid Memory Layout

To take advantage of block RAM distribution on the FPGA, we partitioned the bilateral grid into chunks along different dimensions, and dedicated grid workers for each partition. For large, finely divided grids with many vertices (the largest grids we consider have up to 5 million vertices), we could achieve full resource utilization simply by partitioning the grid along one dimension and allocating a single worker to process each memory bank. For more coarse grids, we partitioned the memory in multiple dimensions.

Our method for laying out data in memory consists of storing all the data needed for a grid vertex in a single packet, and writing the packets sequentially in memory. Rather than storing multiple bilateral grid data structures separately and repeatedly indexing into each of them to process a single vertex, we interleave the data structures together to access all the information for processing a grid vertex as a single packet. When a worker is assigned a grid vertex to process, it can fetch most of the data required for its computation from a single partition, including neighboring vertex data for some dimensions. For large grids, where we only partition on one dimension, the data for two of the three dimensions is stored in the same memory bank, and the worker only has to communicate across banks for the two neighbors in other partitions. For smaller grids, where we partition along multiple dimensions to improve parallelism, workers may need to fetch more of their neighbors from neighboring partitions. All inter-bank communication is handled via the main controller of Figure 3.

To aid in fetching grid vertex data for a worker's vertex or neighboring ones, we abstracted this memory layout into a simple addressing method: we dedicate $\lceil \log_2(k) \rceil$ bits of address space for each grid dimension with size k, and use the last three bits to index into the packet for a grid vertex. For instance, with a bilateral grid of shape $\left[247, 166, 16\right]$ partitioned on the first dimension only, a worker assigned the address `0b 00001010 10100001 0100 001` would map the first dimension's value to memory bank 10, and use the second and third dimensions to fetch the second item in the packet for grid vertex $\left[10, 161, 4\right]$. Indexing into a neighboring vertex in any dimension means incrementing or decrementing a dimension's tag; the main controller detects when a worker is requesting an address in a neighboring bank and multiplexes the request appropriately. This discrete mapping of grid dimensions to address spaces results in simple logic for memory addressing, but at the cost of wasted memory space. Each grid vertex packet contains five items but requires the memory space for eight. The same is true at the grid partition level, since the number of grid vertices along a dimension is a function of the image resolution and the σ_{xy} or σ_l, and does not often fit nicely in power-of-two partitions.

4.4 FPGA Implementation

We implemented a maximal design in Verilog and wrapped the accelerator in an AXI4-Stream compliant interface for portable

Table 2: Resource requirements for FPGA implementations with maximum parallelism.

Model	Logic	RAM	DSP	Clock (MHz)
Virtex Ultrascale+	44%	99%	100%	250

deployment across Xilinx FPGAs. We can fit a maximum of 1,710 workers on a Xilinx Virtex UltraScale+ device. We detail resource utilization of our maximal design in Table 2.

To invoke the bilateral solver accelerator in an application, the application sends a bytestream of bilateral grid data over the FPGA's PCIe-to-AXI DMA interface. The FPGA's driver can be invoked with standard Unix I/O system calls like *read*() and *write*(), and can thus be integrated with software applications written in any high-level language.

At configuration time, we fix the number of grid workers, memory size, and partitioning based on a chosen set of parameters for image resolution and bandwidths in the luma and spatial dimensions. The parameters essentially define the maximum memory size and number of partitions, which can be interpreted as the upper bound of grid sizes that can be run under a certain configuration. The bilateral grid dimensions and number of iterations are software-defined at program runtime. This level of flexibility in our design allows applications to process images of varied resolutions at varied grid bandwidths, but can result in wasted resources if the grid size being processed is much smaller than the accelerator's configured grid size.

5 EXPERIMENTAL RESULTS

We designed HFBS with the goal of improving bilateral solver performance by parallelizing on modern hardware, while maintaining comparable visual accuracy. In this section we evaluate the performance of our algorithm and hardware, including runtime comparison, power consumption measurements, and accuracy evaluation.

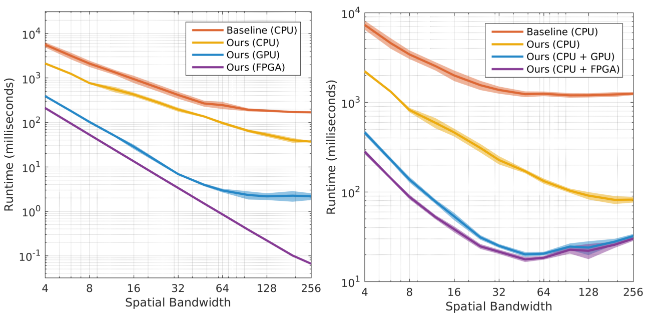
5.1 Methodology

We compare our algorithm across CPU, GPU, and FPGA implementations. The CPU is a Xeon E5-2620 with six cores, and the GPU is an NVIDIA GTX 1080 Ti. Both platforms execute optimized implementations of the kernels written and tuned with Halide [Ragan-Kelley et al. 2013]. We prototyped our FPGA design on a Xilinx Kintex-7 connected to a host CPU over PCIe to evaluate host-device memory traffic, and synthesized and simulated a target evaluation design for the Xilinx Virtex UltraScale+ VU9P to evaluate full-resolution frame processing. In this section we only report results from the Virtex UltraScale+ design.

To benchmark our algorithm, we execute the bilateral solver on flow fields and confidences generated from the ten training images in the Middlebury stereo dataset [Scharstein and Szeliski 2002], and evaluate runtime and accuracy. We compare runtimes for our algorithm on CPU, GPU, and FPGA with the bilateral solver of Barron and Poole [2016] on CPU as the baseline. For CPU and GPU implementations, we report the average runtime from 8 trials; the FPGA runtime is deterministic and did not vary across trials. We

Table 3: Runtimes for different variants of the bilateral solver on different hardware for the VR video use-case. Runtimes for optimization by itself and for the entire algorithm (problem construction/splatting, optimization, and slicing) are shown independently.

Algorithm / Platform	Opt. (ms)	Total (ms)	Power (W)
Baseline (CPU)	1322±171	2529±271	16
Our Algorithm (CPU)	545 ±77	588 ±77	152
Our Algorithm (CPU + GPU)	49 ±3	78 ±5	245
Our Algorithm (CPU + FPGA)	23 ±1	52 ±3	25



(a) Runtime (Optimization Only) **(b) Runtime (Total)**

Figure 6: Runtimes of the baseline, CPU, GPU, and FPGA implementations of HFBS, as a function of the spatial bandwidth (σ_{xy} of Eq. 8).

characterize power consumption for the CPU and GPU by measuring utilization and scaling from the reported device power. For the FPGA, we report estimated power consumption from Xilinx Vivado's power report.

The size of the bilateral grid data ranges from 4KB-1.8GB, depending on the σ_{xy} used to construct the grid. All results use a $\sigma_l = 16$. We use 256 iterations of optimization in all cases, more than enough guarantee convergence for all algorithms and implementations. Note that our performance comparison is not at iso-quality, as our algorithm has slightly more error but qualitatively similar results, which we discuss more in Section 5.3. All computation is single-precision floating-point, except for bilateral solving on the FPGA which is conducted with 64-bit fixed-point numbers. We observe transfer throughput for the FPGA over a single PCIe channel to range between 9.6-11.3 Gbps, which is in keeping with reported estimates. Since both GPU and FPGA communicate with the host over PCIe and we assume frames can be pipelined, we omit the transfer time between the host processor and the device from reported runtimes.

5.2 Runtime Results

Figure 6 plots the runtime results of bilateral solver implementations on all our benchmarks, as a function of the spatial bandwidth. Figure 6a shows the runtime for the optimization portion of the solver, and Figure 6b shows the runtime for the complete bilateral solver including pre-processing. As we increase the spatial bandwidth (σ_{xy} of Eq. 8), we see that the overall grid size decreases,

Table 4: Depth Superresolution Task [Ferstl et al. 2013]

Algorithm	Error (MSE)	Runtime (sec)
Chan et al. [2008]	3.89	3.02
Min et al. [2014]	3.78	0.383
Domain Transform [Gastal and Oliveira 2011]	3.60	0.021
Ma et al. [2013]	3.53	18
Zhang et al. [2014]	3.51	1.346
Guided Filter (Matlab) He et al. [2010]	3.51	0.434
Fast Guided Filter He and Sun [2015]	3.45	0.225
Yang [2015]	3.44	0.304
Farbman et al. [2008]	3.24	6.11
JBU [Adams et al. 2010; Kopf et al. 2007]	3.19	1.98
Barron and Poole [2016]	2.75	0.234
Our Model	3.27	0.047 ± 0.002

and runtimes shorten for all implementations. We find that our algorithm outperforms the baseline on all platforms at all spatial bandwidths. For optimization alone, CPU and FPGA results scale with the grid size, while the GPU results scales until the size of the grid is too small to fully utilize resources. Because splatting and slicing is not accelerated on the FPGA, runtime for the entire bilateral solver does not scale as well at large grid sizes.

Table 3 highlights the runtime results specifically for the VR Video use-case, where $\sigma_{xy} = 12$ as in Anderson et al. [2016], as well as the power consumption of each hardware configuration. Our algorithm's speed outperforms the CPU baseline on all platforms evaluated, and our FPGA accelerator is significantly faster than the baseline while also reducing power consumption. Note that the CPU-only HFBS runtime reported in Table 3 is still far from the real-time requirement of 30 frames-per-second. The GPU and FPGA implementations get very close to real-time for $\sigma_{xy} = 12$, but still do not make it. By selecting $\sigma_{xy} = 32$ and losing some accuracy, both FPGA and GPU implementations meet the real-time requirements.

We also observe that HFBS significantly reduces pre-processing. This is mainly caused by the elimination of the Jacobi preconditioner. The switch to a dense 3D bilateral grid improves available parallelism in the splat-slice routines as well.

5.3 Depth Superresolution

Because our proposed model is an approximation to the bilateral solver, we should expect some drop in the quality of our output relative to that of Barron and Poole [2016]. To quantify this drop in accuracy, we evaluate on the depth superresolution task of Ferstl et al. [2013], which was the primary evaluation used in Barron and Poole [2016]. We evaluate using the same experimental setup and the same hyperparameters as Barron and Poole [2016] ($\sigma_{xy} = 8$, $\sigma_l = 4$), and report MSE with respect to ground truth from the Middlebury Stereo Dataset [Scharstein and Szeliski 2002].

As can be seen in Table 4, our model produces a slightly higher error than that of Barron and Poole [2016], but has a significantly lower runtime (here we report runtime on a Nvidia 1080 Ti). This increase in error is due to the fact that our model ignores color in the input image, and so has difficulty distinguishing between pixels with different chroma but similar luma. The images in this task are unusually colorful and "cartoonish", by virtue of being a constructed

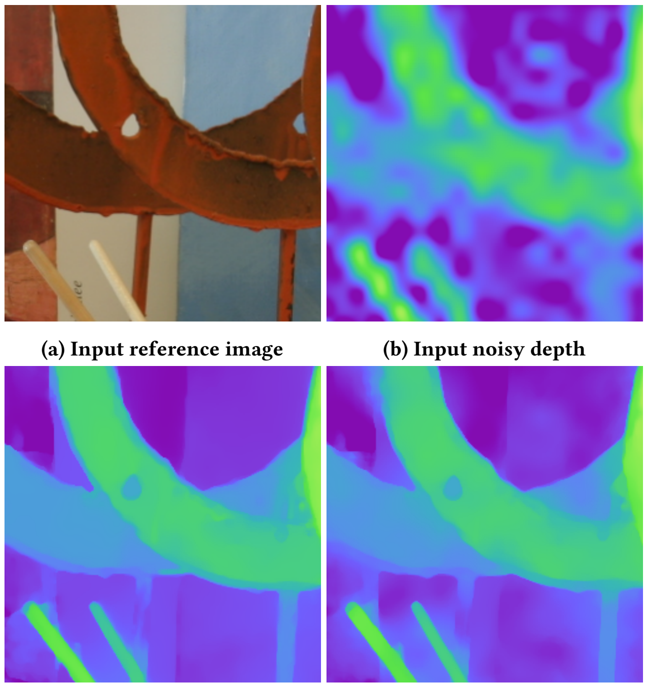

(a) Input reference image (b) Input noisy depth
(c) Improved depth [Barron and Poole 2016] (d) Our improved depth (with HFBS)

Figure 7: A qualitative comparison of HFBS's performance compared to the model of Barron and Poole [2016] on the depth superresolution task of Ferstl et al. [2013]. HFBS produces similar output to Barron and Poole [2016] and is significantly faster.

vision task, so this increase in error represents an upper-bound on the increased error we expect to see in natural scenes. Even with this reduction in error, we see that HFBS is significantly faster than all more-accurate techniques, and significantly more accurate than all faster techniques.

We present qualitative results for this task in Figure 7. As discussed in Section 3.2, HFBS requires double the iterations to achieve the same accuracy level of OBS, but still performs significantly faster. We can see that our output depths are qualitatively very similar to those of Barron and Poole [2016], as expected.

6 DISCUSSION

There are a number of optimizations our hardware design can integrate for improved performance. Nevertheless, we observe that our design can be practically deployed at both the camera node or in the cloud to enable real-time VR video rendering.

Accelerator optimizations. There are many opportunites to further optimize our hardware design. For instance, our design only accelerates the iterative optimization portion of HFBS. Integrating splat and slice operations into our accelerator, as in Rithe et al. [2013], would reduce transfer costs from GB-large bilateral grid to smaller MB-sized images and further reduce runtimes. Also, many vertices of the dense bilateral grid begin as zeros and do not need

Table 5: Full-system specification for an end-to-end real-time VR pipeline

Item	Use	#	Unit $	Total $	Max. Power
GoPro	Camera	16	$360	$5, 760	N/A
Virtex Ultrascale+	HFBS	16	$2, 995	$47, 920	≈400 W
Intel i7-7700K	Host CPU	1	$350	$350	≈ 90 W
Full-System				$55,360	490 W

to be processed; intelligently ignoring these zero-valued grid vertices can reduce wasted computation and potentially improve the runtime. Similarly, the wasted memory space from our addressing scheme can be mitigated with increased control logic, which may allow us to maximize the bilateral grid size.

System specifications for real-time VR video processing platforms. While our design can execute bilateral solving under real-time constraints, the bilateral solver is just one step in the Jump VR video rendering pipeline. Moreover, the design we present processes the flow field from a camera pair while the VR video capture system we target processes 16 flow fields from a 16-camera rig. We outline the specifications and cost for a system that could process the full 16-camera input to produce virtual reality video in real-time in Table 5. The monetary cost of deploying such a many-FPGA system in both configurations is high, but the power consumption of our FPGA-based system, with 16 high-end fully-utilized FPGAs, is approximately that of two GPUs. Such power savings can be critical for mobile camera rigs. At the data center level, power constraints are less stringent, but deploying custom hardware for high-bandwidth tasks can still reduce power consumption and operating costs.

7 CONCLUSIONS

The hardware-friendly bilateral solver enables scalable, real-time processing of VR video on modern hardware. We explore a hardware-software codesign approach to construct an algorithm that is both faster *and* more accurate than prior work, optimizing algorithm details and hardware implementation together. In particular, HFBS uses a bilateral-space Heavy Ball algorithm and a 3D dense bilateral grid that allows fast and predictable memory accesses. We also design an FPGA accelerator for HFBS using reduced-precision fixed-point numbers and customized memory layout. Our CPU, GPU, and FPGA implementations of HFBS are 4×, 32×, and 50× faster than the original bilateral solver. We observe that our FPGA accelerator is more energy-efficient than comparable CPU and GPU implementations, and can be practically deployed at both the camera node or in the cloud to enable real-time VR video rendering.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their thoughtful feedback. This work was supported in part by the National Science Foundation under Grant CCF-1518703 and a generous gift from Google.

REFERENCES

Andrew Adams, Jongmin Baek, and Myers Abraham Davis. 2010. Fast high-dimensional filtering using the permutohedral lattice. *Eurographics* (2010).

Robert Anderson, David Gallup, Jonathan T Barron, Janne Kontkanen, Noah Snavely, Carlos Hernández, Sameer Agarwal, and Steven M Seitz. 2016. Jump: virtual reality video. *SIGGRAPH Asia* (2016).

Jonathan T Barron, Andrew Adams, YiChang Shih, and Carlos Hernández. 2015. Fast bilateral-space stereo for synthetic defocus. *CVPR* (2015).

Jonathan T Barron and Ben Poole. 2016. The fast bilateral solver. *ECCV* (2016).

Joel Carranza, Christian Theobalt, Marcus A Magnor, and Hans-Peter Seidel. 2003. Free-viewpoint video of human actors. (2003).

Derek Chan, Hylke Buisman, Christian Theobalt, and Sebastian Thrun. 2008. A noise-aware filter for real-time depth upsampling. *ECCV Workshops* (2008).

Jiawen Chen, Sylvain Paris, and Frédo Durand. 2007. Real-time edge-aware image processing with the bilateral grid. *SIGGRAPH* (2007).

Facebook. 2017. Facebook Surround 360. https://facebook360.fb.com/facebook-surround-360/. (2017). Accessed: 2017-06-15.

Zeev Farbman, Raanan Fattal, Dani Lischinski, and Richard Szeliski. 2008. Edge-preserving decompositions for multi-scale tone and detail manipulation. *SIGGRAPH* (2008).

David Ferstl, Christian Reinbacher, Rene Ranftl, Matthias Ruether, and Horst Bischof. 2013. Image guided depth upsampling using anisotropic total generalized variation. *ICCV* (2013).

Eduardo S. L. Gastal and Manuel M. Oliveira. 2011. Domain transform for edge-aware image and video processing. *SIGGRAPH* (2011).

Google. 2017. Daydream - Google VR. https://vr.google.com/daydream/. (2017). Accessed: 2017-06-15.

Kaiming He and Jian Sun. 2015. Fast guided filter. *CoRR* abs/1505.00996 (2015).

Kaiming He, Jian Sun, and Xiaoou Tang. 2010. Guided image filtering. *ECCV* (2010).

Fu-Chung Huang, Kevin Chen, and Gordon Wetzstein. 2015. The light field stereoscope: immersive computer graphics via factored near-eye light field displays with focus cues. *ACM Transactions on Graphics (TOG)* (2015).

Johannes Kopf, Michael F. Cohen, Dani Lischinski, and Matt Uyttendaele. 2007. Joint bilateral upsampling. *SIGGRAPH* (2007).

Marc Levoy and Pat Hanrahan. 1996. Light field rendering. *SIGGRAPH* (1996).

Ziyang Ma, Kaiming He, Yichen Wei, Jian Sun, and Enhua Wu. 2013. Constant time weighted median filtering for stereo matching and beyond. *ICCV* (2013).

Dongbo Min, Sunghwan Choi, Jiangbo Lu, Bumsub Ham, Kwanghoon Sohn, and Minh N. Do. 2014. Fast global image smoothing based on weighted least squares. *Transactions on Image Processing* (2014).

Oculus. 2017. Oculus Rift. https://www.oculus.com/rift/. (2017). Accessed: 2017-06-15.

S. Peleg, M. Ben-Ezra, and Y. Pritch. 2001. Omnistereo: panoramic stereo imaging. *PAMI* (2001).

Boris Teodorovich Polyak. 1964. Some methods of speeding up the convergence of iteration methods. *U. S. S. R. Comput. Math. and Math. Phys.* 4, 5 (1964).

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *PLDI* (2013).

Rahul Rithe, Priyanka Raina, Nathan Ickes, Tenneti Srikanth V, and Anantha P Chandrakasan. 2013. Reconfigurable processor for energy-scalable computational photography. *ISSCC* (2013).

Samsung. 2017. Gear VR. http://www.samsung.com/us/explore/gear-vr/. (2017). Accessed: 2017-06-15.

Daniel Scharstein and Richard Szeliski. 2002. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International journal of computer vision* (2002).

Heung-Yeung Shum, King-To Ng, and Shing-Chow Chan. 2005. A virtual reality system using the concentric mosaic: construction, rendering, and data compression. *IEEE Transactions on Multimedia* (2005).

Qingxiong Yang. 2014. Hardware-efficient bilateral filtering for stereo matching. *PAMI* (2014).

Qingxiong Yang. 2015. Stereo matching using tree filtering. *PAMI* (2015).

Qi Zhang, Li Xu, and Jiaya Jia. 2014. 100+ times faster weighted median filter (WMF). *CVPR* (2014).