Halide and GENESIS for Generating Domain-Specific Architecture of Guided Image Filtering

Akari Ishikawa, Norishige Fukushima*
Nagoya Institute of Technology, Japan
* https://fukushima.web.nitech.ac.jp/en/

Akira Maruoka†
Fixstars Corporation, Japan
† https://www.fixstars.com/en/

Takuro Iizuka‡
Fixstars Solutions Inc., USA
‡ http://us.fixstars.com/

***Abstract*—Halide is a domain-specific language for image processing on CPUs and GPUs. The language is powerful for image processing with deep pipeline, e.g., guided image filtering. The guided image filtering is utilized for various applications. Some papers implement the filter on hardware; however, the implementing hardware and the purposes are different. Hardware implementation is hard; therefore, compiler supports are necessary. We utilize Halide with extended FPGA backend, called GENESIS. In our experiment, Halide code with CPU/FPGA backend is faster than the optimized C++. Also, the code length of the C++ and Halide for CPU/FPGA is 575, 141, 139 lines, respectively.**

***Index Terms*—Halide, GENESIS, domain-specific language, guided image filter, FPGA**

I. Introduction

Approaching the end of Moore's law, domain-specific hardwares/compilers are now focused. Halide [1]–[4] is a domain-specific language (DSL) for image processing. In the Halide, we can separate code into an algorithm part and a scheduling part. We can write how to work image processing in the algorithm part, and how to compute it in the scheduling part. Changing only the scheduling, we can optimize the code for the specific hardware, such as CPU (x86, ARM, MIPS, Hexagon, PowerPC, Xeon Phi) and GPU (CUDA, OpenCL, OpenGL).

Traditionally, FPGA design is usually written in hardware description languages (HDL). The HDL code tends to be much longer than the code of software programming languages, e.g., C/C++. Also, programmers are required in-depth hardware knowledge for writing the HDL, which is not friendly for non-experts. Therefore, we extend Halide to have FPGA backend. We call the compiler GENESIS.

In this paper, we reveal the effectiveness of the programming with Halide with GENESIS for image processing. We make an application specific integrated circuit for guided image filtering [5], [6]. The filter is an edge-preserving filtering, and the filter provides much applications, such as denoising [7], [8], detail enhancement [5], [9], high dynamic range imaging, haze removing [9], [10], under-water image processing [11], image matting [5], [12], saliency map estimation [13], upsampling [14], stereo matching and optical flow estimation [15], [16]. Also, the guided image filter has several extensions [9], [17]–[21].

Bilateral filtering [22] is typical edge-preserving filtering. The filter is a finite impulse response (FIR) filter, and there is efficient implementation for CPU [23], [24] and FPGA [25], [26]. The bilateral filtering depends on the kernel radius of the filter. When the algorithm is tuned for some specific parameters, the hard coded program has limitation in flexibility for various parameter. The guided image filtering does not depend on the kernel radius of the filter; thus, the filter adaptively works for various application with adjustable parameter without changing of circuits. However, the algorithm has a long image processing pipeline; hence, tuning of the code is hard without compiler supports.

Some papers implement the guided image filtering on hardware. The work of [27] implements the filter on ASIC by Verilog with the double integral image [28]. The work of [29] constructs the filter on FPGA using the separable summed area table [30] with approximated computation, and [31] extends this work. [32] implements fast guided filtering [14] on FPGA. Most approaches are aimed for grayscale images, and color image processing requires some algorithm modifications. Also, the optimal implementation is different for each hardware. Implementation on hardware is hard even if the difference from the previous work is small; therefore, compiler supports are necessary for easiness of development.

In this paper, we describe the tuning for CPU/FPGA to optimize the implementation with the compiler supports of Halide and GENESIS. The contributions are as follows:

- Halide and GENESIS code is shorter than the native code.
- We reveal that the optimal implementation of the guided image filtering CPU and FPGA.

II. Background

A. Guided Image Filtering

The guided image filtering converts local patches in an input image by a linear transformation of a guide image. Let the guide signal be G, and it is possible to be $G = I$, where I is an input image. The output J is assumed as follows:

$$J_{\boldsymbol{p}} = a_{\boldsymbol{k}} G_{\boldsymbol{p}} + b_{\boldsymbol{k}}, \forall \boldsymbol{p} \in \omega_{\boldsymbol{k}}, \tag{1}$$

where $\boldsymbol{k}$ indicates a center position of a rectangular patch $\omega_{\boldsymbol{k}}$, and $\boldsymbol{p}$ is a pixel-position in the patch. $a_{\boldsymbol{k}}$ and $b_{\boldsymbol{k}}$ are coefficients

for the linear transformation. The equation represents the coefficients linearly convert that guide signals in a patch.

The coefficients are calculated by a linear regression of the input signal I and (1);

$$\arg\min_{a_{\boldsymbol{k}},b_{\boldsymbol{k}}} = \sum_{\boldsymbol{p}\in\omega_{\boldsymbol{k}}} ((a_{\boldsymbol{k}}G_{\boldsymbol{p}} + b_{\boldsymbol{k}} - I_{\boldsymbol{p}})^2 + \epsilon a_{\boldsymbol{k}}^2). \quad (2)$$

The estimated coefficients are as follows:

$$a_{\boldsymbol{k}} = \frac{cov_{\boldsymbol{k}}(G,I)}{var_{\boldsymbol{k}}(G)+\epsilon}, \quad b_{\boldsymbol{k}} = \bar{I}_{\boldsymbol{k}} - a_{\boldsymbol{k}}\bar{G}_{\boldsymbol{k}}, \quad (3)$$

where ϵ indicates a parameter of smoothing degree. $\bar{\cdot}_{\boldsymbol{k}}$, $cov_{\boldsymbol{k}}$ and $var_{\boldsymbol{k}}$ indicate mean, variance, and covariance values of the patch $\boldsymbol{k}$. The coefficients are overlapping in the output signals; thus, these coefficients are averaged;

$$\bar{a}_{\boldsymbol{i}} = \frac{1}{|\omega|}\sum_{\boldsymbol{k}\in\omega_{\boldsymbol{p}}} a_{\boldsymbol{k}}, \quad \bar{b}_{\boldsymbol{i}} = \frac{1}{|\omega|}\sum_{\boldsymbol{k}\in\omega_{\boldsymbol{p}}} b_{\boldsymbol{k}}, \quad (4)$$

where $|\cdot|$ indicates the number of elements in the set. Finally, the output is calculated as follows:

$$J_{\boldsymbol{i}} = \bar{a}_{\boldsymbol{i}}G_{\boldsymbol{i}} + \bar{b}_{\boldsymbol{i}}. \quad (5)$$

For color filtering, let input, output and guidance signals be $\boldsymbol{p} = \{p^1, p^2, p^3\}$, q^n $(n = 1, 2, 3)$, and $\boldsymbol{G}$, respectively. The per-channel output is defined as:

$$J_{\boldsymbol{p}}^n = \bar{\boldsymbol{a}_{\boldsymbol{p}}^n}^{\mathrm{T}}\boldsymbol{G}_{\boldsymbol{p}} + \bar{b_{\boldsymbol{p}}^n}, \quad (6)$$

$$\bar{\boldsymbol{a}_{\boldsymbol{p}}^n} = \frac{1}{|\omega|}\sum_{\boldsymbol{k}\in\omega_{\boldsymbol{p}}} \boldsymbol{a}_{\boldsymbol{k}}^n, \quad \bar{b_{\boldsymbol{p}}^n} = \frac{1}{|\omega|}\sum_{\boldsymbol{k}\in\omega_{\boldsymbol{p}}} b_{\boldsymbol{k}}^n. \quad (7)$$

The coefficients $\boldsymbol{a}_{\boldsymbol{k}}^n$, $b_{\boldsymbol{k}}^n$ are obtained as follows:

$$\boldsymbol{a}_{\boldsymbol{k}}^n = \frac{cov_{\boldsymbol{k}}(\boldsymbol{G}, I^n)}{var_{\boldsymbol{k}}(\boldsymbol{G}) + \epsilon\boldsymbol{E}}, \quad b_{\boldsymbol{k}}^n = \bar{I_{\boldsymbol{k}}^n} - \boldsymbol{a}_{\boldsymbol{k}}^{n\,\mathrm{T}}\bar{\boldsymbol{G}}_{\boldsymbol{k}}, \quad (8)$$

where $\boldsymbol{E}$ is an identity matrix. When the output signal is a color image, $cov_{\boldsymbol{k}}$ is a vector, which elements are covariance of the patch in I and $\boldsymbol{G}$. Also, $var_{\boldsymbol{k}}$ is the variance of the R, G, and B components, which will be a covariance matrix, in the patch of $\boldsymbol{G}$. The matrix division is calculated by multiplying the inverse matrix of the denominator from the left. We use box filtering for the calculation results of per pixel mean, variance, and covariance. The filter with the summed area table [30], [33] or integral image [34] has $O(1)$ for kernel radii.

B. Halide

There are several image processing DSL, such as Halide [1]–[4], and Darkroom [35]. The Halide is a major DSL for image processing. The language is a pure functional language and is embedded in C++. The Halide code is modularized as algorithm and scheduling parts This modularization makes the Halide code flexible. The algorithm parts show the image processing algorithm, and the scheduling parts reveal the computational order and computational method, e.g., vectorization and parallelization.

Fig. 1 shows the Halide code of 3×3 box filtering for CPU backend. *Func* indicates equations and *Var* shows variables. "*Func* f" represents an input image, and "*Var* x, y" show x and y coordinates of images. In the algorithm parts, we horizontally average the input image f, and then vertically mean the averaged image. In the scheduling parts, computational scheduling is defined for each equation of *Func* by calling various class methods, e.g., *tile, vectorize, parallel*, and *compute_at*. *tile* points image tiling, and the scheduling splits the image into 256×32 tiles. *vectorize* orders vectorized computing with SIMD units, e.g., SSE, AVX, and NEON, and this vectorizes pixels along the x loop. *parallel* shows multi-thread computing with multi-core/thread CPU, and the scheduling parallelize along the y loop. *compute_at* indicates how to memorize computed results, and we compute and memorize "*Func* blur_x" on x, y in ranged computation of "*Func* blur_y" under the schedule. In the default schedule, no computation is memorized, i.e., all functions are re-computed.

We also show the algorithm of guided image filtering and box filtering in Alg. 1. The algorithm is written in Halide,

```
Func blur_3x3(Func f)
{
    Func blur_x,blur_y;
    Var x,y,xi,yi;
    // algorithm part
    blur_x(x,y)=(f(x-1,y)+f(x,y)+f(x+1,y))/3;
    blur_y(x,y)=(blur_x(x,y-1)
                 +blur_x(x,y)+blur_x(x,y+1))/3;
    // scheduling part
    blur_y.tile(x,y,xi,yi,256,32)
          .vectorize(xi,8).parallel(y);
    blur_x.compute_at(blur_y,x).vectorize(x,8);

    return blur_y;
}
```

Fig. 1. Halide code of 3×3 box filtering for CPU backend.

Algorithm 1 Halide-like code for guided image filtering.

1: $\mathrm{II}(c,x,y) = \mathrm{I}(c,x,y) * \mathrm{I}(c,x,y)$
   $\mathrm{Ip}(cx,cy,x,y) = \mathrm{p}(cx,x,y) * \mathrm{I}(cy,x,y)$
   $\mathrm{mean}_I = \mathrm{f}_{mean}(I,r)$
   $\mathrm{mean}_p = \mathrm{f}_{mean}(p,r)$
   $\mathrm{mean}_{II} = \mathrm{f}_{mean}(II,r)$
   $\mathrm{mean}_{Ip} = \mathrm{f}_{mean}(Ip,r)$
2: $\mathrm{var}_I(cx,cy,x,y)$
   $=$select$(cx == cy,$
   $\mathrm{mean}_{II}(cx,cy,x,y) - \mathrm{mean}_I(cx,x,y) * \mathrm{mean}_I(cy,x,y) + \epsilon,$
   $\mathrm{mean}_{II}(cx,cy,x,y) - \mathrm{mean}_I(cx,x,y) * \mathrm{mean}_I(cy,x,y))$
   $\mathrm{cov}_{Ip}(cx,cy,x,y) = \mathrm{mean}_{Ip}(cx,cy,x,y)$
   $-\mathrm{mean}_I(cx,x,y) * \mathrm{mean}_p(cy,x,y)$
   $\mathrm{inv}_I(cx,cy,x,y) = \mathrm{covMat}(\mathrm{var}_I)(cx,cy,x,y)$
3: $\mathrm{covDet}(x,y) = \sum_{i=0}^{2}(\mathrm{inv}_I(0,i,x,y,0,i) * \mathrm{var}_I(0,i,x,y))$
4: $\mathrm{a}(cx,cy,x,y)$
   $= \sum_{i=0}^{2}(\mathrm{inv}_I(i,cy,x,y) * \mathrm{cov}_{Ip}(cx,i,x,y))/\mathrm{covDet}(x,y)$
   $\mathrm{b}(c,x,y)$
   $= \mathrm{mean}_p(c,x,y) - \sum_{i=0}^{2}(\mathrm{a}(c,i,x,y) * \mathrm{mean}_I(i,x,y))$
5: $\mathrm{mean}_a = \mathrm{f}_{mean}(a,r)$
   $\mathrm{mean}_b = \mathrm{f}_{mean}(b,r)$
6: $q(c,x,y) = \sum_{i=0}^{2}(\mathrm{mean}_a(c,i,x,y) * \mathrm{I}(i,x,y)) + \mathrm{mean}_b(c,x,y)$

Algorithm 2 Halide-like code for mean (box) filter.

1: $\text{val}(c, x, y) = \text{input}(c, x, y)$
2: $\text{val}(rc, rx, ry) \mathrel{+}= \text{val}(rc, rx - 1, ry)$
   $\text{val}(rc, rx, ry) \mathrel{+}= \text{val}(rc, rx, ry - 1)$
3: $\text{output} = \text{val}(c,x+r,y+r) - \text{val}(c,x-r-1,y+r)$
   $-\text{val}(c,x+r,y-r-1)+\text{val}(c,x-r-1,y-r-1)$

but some lines are omitted for readability. The code contains the map computation, i.e., matrix addition, subtraction, Hadamard product, and small matrix inversion, and reduction computation. Each operation has low computational intensity, and also most of the computation in the naïve algorithm has low computational intensity. Therefore, it is essential for generating codes with high computational intensity.

III. GENESIS

GENESIS is a DSL compiler. It converts Halide codes into the Vivado C/C++ code, which can be performed by high-level synthesis (HLS) for Xilinx's FPGA. The output of GENESIS is highly optimized for the HLS compiler by analyzing and transforming the input code; thus, it is not naive converting. The transformed code generates a domain specific architecture to compute a specific algorithm effectively. An optimization purpose depends on the developer because FPGA is highly flexible. As a result, searching "best" architecture in manual consumes an enormous amount of time. The GENESIS compiler minimizes the amount of coding and controls the various factors in performance through scheduling function in Halide extensions.

A. Strategy of Generating Architecture in GENESIS

It is essential for balancing data I/O performance and computing performance of arithmetic units to maximize hardware performance. The followings denote the strategy of composing of the arithmetic and data I/O units.

We can naturally convert the description into arithmetic units for HLS languages, since the Halide is a pure functional DSL, and the language can describe operations for multiple data set without side effect. In GENESIS, we generate fully pipelined arithmetic units to operate the order for each cycle. The throughput of arithmetic units can be determined at compile time since throughput itself depends only on the number of arithmetic units. The latency of the arithmetic units is determined at the successive design flow because that depends on the hardware speed grade, frequency, and wiring length after technology mapping. *unroll* scheduling in Halide can control the number of arithmetic units.

It is complex and important how to implement architectures for data I/O. The straightforward approach is the typical memory I/O architecture, i.e., we allocate and fetch data on the static/dynamic random access memory (SRAM/DRAM) through the addressable memory bus. There are several issues in this approach. Firstly, the size of the memory is proportional to the amount of data size. SRAM is rare resources, and the current FPGA even has dozens MB. DRAM utilization mitigates the size issue; however, the memory bus becomes a bottleneck, since the interface of the DRAM exists in the outside of the FPGA. Secondly, data reusability is low. We should access the memory bus every time, even if data have high spatial locality. We can moderate the issue by adding memory cache architecture, but it consumes hardware resources. Finally, we cannot generate pipeline across multiple processes of memory I/O. When and where the address in memory is used is determined at run-time; thus, we should wait for reading from a data buffer until writing to the buffer is finished.

One-way streams are efficient design of the data I/O on FPGA. However, an implementing algorithm does not always assure the one-way access for data. We combine address analysis and stream conversion to solve this problem in GENESIS. At first, we obtain the range of accessing data in the compiling code by the address analysis. Then, we generate local buffers consisted of registers and shift memories also based on the address analysis. This local buffer is optimized for static addressing, i.e., specific registers, which is statically analyzed, are connected to arithmetic units by partially connected crossbar switches. We can provide data to the arithmetic units by combining the local buffer and stream I/O. Fig. 2 shows an example architecture of the I/O stream, which has one input and one output. GENESIS try to convert data I/O to stream as much as possible in default. Also, we can control explicitly the type of I/O by extended scheduling function "*hls_interface*".

B. Scheduling Functions in GENESIS

compute_root function modularizes hardware blocks. Arithmetic units and I/O streams are constructed per this module as one unit The latency of the system becomes long by fine-grained modularization, which is realized by issuing many *compute_root* schedules. This is caused by the latency of the bus between modules and local buffer for each module. By contrast, each function defined by *Func* with *compute_root* scheduling dramatically reduces the size of the local buffer in the case of multiple pipelined functions, e.g., deep learning.

unroll scheduling affects to how many of arithmetic units are constructed. The scheduling has an argument. It is the number of unrolling units. This scheduling improves the throughput of arithmetic units, but keep that of I/O. *unroll* is effective when arithmetic operations are a bottleneck, not data I/O.

hls_burst scheduling adjusts throughput of data I/O. The scheduling function has arguments for the coefficients in burst size, and then our compiler determines the width of data bus based on the coefficient.

IV. Domain Specific Implementation

A. CPU Backend

Fig. 3 depicts the optimal scheduling for CPU backend. The schedule computes linear coefficients a, b and then stores them on memory before output computation by *compute_root* scheduling. For computation of the determinant of covariance *covDet*, where the computing of the coefficient a, the *covDet* value is computed at once and stored the result on memory

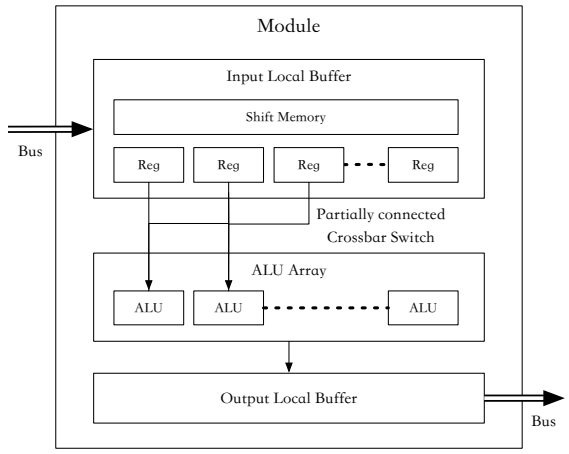

Fig. 2. Microarchitecture for I/O stream.

by *compute_at* and *store_at* scheduling. $output, a, b$ are also scheduled with parallelization and vectorization by *parallel* and *vectorize* methods. For parallelization, we split these images by *split* method and then parallelly perform for each split slice. Besides, we unroll the loop of the color channel by the unrolling method of *unroll* with the domain specification method of *bound*.

B. FPGA Backend with GENESIS

Fig. 4 shows the optimal scheduling for FPGA backend. We do not need schedules of splitting the image and then parallelizing and vectorizing computation for FPGA since GENESIS fully pipelines arithmetic units as described in Sec. III-A. The GENESIS extension of the *schedule* method simultaneously performs *compute_root* and *bound* schedules for the first argument of *Func*. Herein, we adjust the computational timing of *Func* and also allocate the buffer, which is required for the computation of *Func*. The most inner loop is the color channel, and the dimension is always 3 in the color image processing. In this case, we unroll arithmetic units for parallelly processing 24 bit data. Notice that required I/O becomes larger as the length of the unrolling is longer; thus, we should adjust the length of the burst of data I/O. The GENESIS extension of *hls_burst* expands the width of the I/O bus as its argument.

In Alg. 1, there is some difference in $\sum$ operations for CPU and FPGA implementation. For CPU backend, we usually use *sum* function with *RDom* ranged variables for the summation. For FPGA backend, we use *sum_unroll*, which is extended in GENESIS, for the same purpose. This function sums up and then unrolling each operation.

V. Experimental Results

We compared each scheduling for guided image filtering in CPU and FPGA backend. The input image was 512×512 color images. The parameter of the filter is $r = 3$ and $\epsilon = 0.04$ CPU was Intel Core i7-7800 3.50 GHz compiled with Visual Studio 2017. FPGA was simulated Xilinx's ZedBoard.

At first, we optimized C++ code for CPU parallelized by OpenMP and vectorized by AVX intrinsics. Parallelization

```
Guided Image Filtering
covDet.store_at(a,yo).compute_at(a,y);
a.compute_root().bound(cx,0,3).unroll(cx)
 .split(y,yo,y,16).vectorize(x,4).parallel(yo);
b.compute_root().bound(c,0,3).unroll(c)
 .split(y,yo,y,16).vectorize(x,4).parallel(yo);
output.bound(c,0,3).unroll(c)
 .split(y,yo,y,16).vectorize(x,4).parallel(yo);

Box Filter
val.comute_root().bound(c,0,3).unroll(c)
   .split(y,yo,y,16).vectorize(x,4).parallel(yo);
val.update(0).unroll(c).parallel(ry);
val.update(1).unroll(c).vectorize(rx);
.split(y,yo,y,16).vectorize(x,4).parallel(yo);
```

Fig. 3. Optimal scheduling for CPU backend.

```
Guided Image Filtering
input.hls_burst(3);
schedule(a,{3,3,width,height})
 .unroll(cx).unroll(cy).hls_burst(9);
schedule(b,{3,width,height})
 .unroll(c).hls_burst(3);
schedule(output,{3,width,height})
 .unroll(c).hls_burst(3);

Box Filter
schedule(val,{3,width,height})
 .unroll(cy).hls_burst(3);
val.update(0).unroll(rc);
val.update(1).unroll(rc);
```

Fig. 4. Optimal scheduling for FPGA backend with GENESIS extension.

and vectorization were applied for each matrix operation, e.g., matrix multiplication, addition and subtraction and box filtering, and matrix inversion. The computational time of the code was 49.81 ms. The code length was 575 lines.

The computational time of Halide's CPU backend with the scheduling in Fig. 3 was 21.04 ms. The scheduling parallelizes processing with redundant processing; however, that performs coarse-grained parallelization. On the contrary, the native C++ code was parallelized in fine-grained, since each matrix processing is forked and then joined for parallel processing. The code length of Halide for the CPU backend was 141 lines.

Next, the latency of FPGA is 2100619 cycles. If we assume that the FPGA's clock is 510 MHz, the computational time is 14.00 ms on the simulator. Therefore, the FPGA implementation is 1.5 times faster than the CPU backend. The code length of Halide for FPGA backend was 139 lines.

VI. Conclusion

In this paper, we proposed effective scheduling for guided image filtering with extending Halide to have FPGA backend. GENESIS extension for Halide supports FPGA backend well, and the code for FPGA implementation becomes short. Based on the tuning flexibility from Halide, the filter is easily optimized for CPU and FPGA backend. Experimental results support that the Halide and GENESIS code is shorter than the native code and the computational performance is also higher.

Acknowledgment

This work was supported by KAKENHI JP17H01764, JP18K19813.

REFERENCES

[1] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Transactions on Graphics*, vol. 31, no. 4, 2012.

[2] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM Transactions on Graphics*, vol. 48, no. 6, pp. 519–530, 2013.

[3] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Transactions on Graphics*, vol. 35, no. 4, 2016.

[4] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in Halide," *ACM Transactions on Graphics*, vol. 37, no. 4, 2018.

[5] K. He, J. Sun, and X. Tang, "Guided image filtering," in *Proc. European Conference on Computer Vision (ECCV)*, 2010.

[6] K. He, J. Sun, and X. Tang, "Guided image filtering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 6, pp. 1397–1409, 2013.

[7] J. Lu, K. Shi, D. Min, L. Lin, and M. N. Do, "Cross-based local multipoint filtering," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[8] C. C. Pham and J. W. Jeon, "Efficient image sharpening and denoising using adaptive guided image filtering," *IET Image Processing*, vol. 9, no. 1, pp. 71–79, 2014.

[9] N. Fukushima, K. Sugimoto, and S. Kamata, "Guided image filtering with arbitrary window function," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2018.

[10] K. He, J. Sun, and X. Tang, "Single image haze removal using dark channel prior," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 12, pp. 2341–2353, 2011.

[11] I-C. Huang, H.-H. Chen C.-H. Chang, M.-F. Lin, and S.-R. Kuang, "Hardware implementation of an efficient guided image filter for underwater image restoration," *International Journal of Signal Processing Systems*, , no. 5, pp. 94–999, 2017.

[12] N. Kodera, N. Fukushima, and Y. Ishibashi, "Filter based alpha matting for depth image based rendering," in *IEEE Visual Communications and Image Processing (VCIP)*, 2013.

[13] Y. Ding, J. Xiao, and J. Yu, "Importance filtering for image retargeting," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2011, pp. 89–96.

[14] K. He and J. Sun, "Fast guided filter," *CoRR*, vol. abs/1505.00996, 2015.

[15] A. Hosni, C. Rhemann, M. Bleyer, C. Rother, and M. Gelautz, "Fast cost-volume filtering for visual correspondence and beyond," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 35, no. 2, pp. 504–511, 2013.

[16] T. Matsuo, S. Fujita, N. Fukushima, and Y. Ishibashi, "Efficient edge-awareness propagation via single-map filtering for edge-preserving stereo matching," in *Proc. SPIE*, 2015, vol. 9393.

[17] X. Tan, C. Sun, and T. D. Pham, "Multipoint filtering with local polynomial approximation and range guidance," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[18] L. Dai, M. Yuan, F. Zhang, and X. Zhang, "Fully connected guided image filtering," in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 2015.

[19] S. Fujita and N. Fukushima, "High-dimensional guided image filtering," in *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, 2016, pp. 27–34.

[20] S. Fujita and N. Fukushima, *Extending Guided Image Filtering for High-Dimensional Signals*, vol. 693, pp. 439–453, Springer International Publishing, 2017.

[21] L. Dai, M. Yuan, Z. Li, X. Zhang, and J. Tang, "Hardware-efficient guided image filtering for multi-label problem," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[22] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proc. International Conference on Computer Vision (ICCV)*, 1998.

[23] Y. Maeda, N. Fukushima, and H. Matsuo, "Taxonomy of vectorization patterns of programming for fir image filters using kernel subsampling and new one," *Applied Sciences*, vol. 8, no. 8, 2018.

[24] Y. Maeda, N. Fukushima, and H. Matsuo, "Effective implementation of edge-preserving filtering on cpu microarchitectures," *Applied Sciences*, vol. 8, no. 10, 2018.

[25] A. Gabiger, M. Kube, and R. Weigel, "A synchronous fpga design of a bilateral filter for image processing," in *Annual Conference of IEEE Industrial Electronics (IECON)*, 2009.

[26] A. Gabiger-Rose, M. Kube, R. Weigel, and R. Rose, "An fpga-based fully synchronized design of a bilateral filter for real-time image denoising," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 8, pp. 4093–4104, 2014.

[27] C. Kao, J. Lai, and S. Chien, "Vlsi architecture design of guided filter for 30 frames/s full-hd video," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 24, no. 3, pp. 513–524, 2014.

[28] Y. Tseng, P. Hsu, and T. Chang, "A 124 mpixels/s vlsi design for histogram-based joint bilateral filtering," *IEEE Transactions on Image Processing*, vol. 20, no. 11, pp. 3231–3241, 2011.

[29] C. Ttofis, C. Kyrkou, and T. Theocharides, "A low-cost real-time embedded stereo vision system for accurate disparity estimation based on guided image filtering," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2678–2693, 2016.

[30] F. C. Crow, "Summed-area tables for texture mapping," in *Proc. ACM SIGGRAPH*, 1984, pp. 207–212.

[31] X. Zhang, H. Sun, S. Chen, and N. Zheng, "Vlsi architecture exploration of guided image filtering for 1080p@60hz video processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 1, pp. 230–241, 2018.

[32] D. EI Mezeni and L. Saranovac, "Fast guided filter for power-efficient real-time 1080p streaming video processing," *Journal of Real-Time Image Processing*, 2018.

[33] N. Fukushima, Y. Maeda, Y. Kawasaki, M. Nakamura, T. Tsumura, K. Sugimoto, and S. Kamata, "Efficient computational scheduling of box and gaussian fir filtering for cpu microarchitecture," in *Proc. Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, 2018.

[34] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2001.

[35] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines.," *ACM Transactions on Graphics*, vol. 33, no. 4, 2014.