IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 2 December 2023, accepted 2 January 2024, date of publication 15 January 2024, date of current version 1 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3354720

RESEARCH ARTICLE

SIMD-Constrained Lookup Table for Accelerating Variable-Weighted Convolution on x86/64 CPUs

YUKI NAGANAWA[1], (Graduate Student Member, IEEE), HIROKAZU KAMEI[1], YAMATO KANETAKA[1], (Graduate Student Member, IEEE), HARUKI NOGAMI[1], (Graduate Student Member, IEEE), YOSHIHIRO MAEDA[2], (Member, IEEE), AND NORISHIGE FUKUSHIMA[1], (Member, IEEE)

[1]Department of Engineering, Faculty of Engineering, Nagoya Institute of Technology, Showa-ku, Nagoya, Aichi 466-8555, Japan
[2]Department of Electrical Engineering, Faculty of Engineering, Tokyo University of Science, Niijuku, Katsushika-ku, Tokyo 125-8585, Japan

Corresponding author: Norishige Fukushima (fukushima@nitech.ac.jp)

This work was supported in part by the Japan Society for the Promotion of Science (JSPS) KAKENHI under Grant 21H03465 and Grant 21K17768, and in part by the Environment Research and Technology Development Fund of the Environmental Restoration and Conservation Agency of Japan under Grant JPMEERF20222M01.

ABSTRACT Convolution is the inner product of the neighborhood signal and weights and plays a fundamental role in image processing; thus, acceleration of convolution is essential. Among convolutions, variable-weighted convolution is used in adaptive filters and edge-preserving smoothing to realize various applications. Some weights are replaced with lookup tables (LUTs) to accelerate these filters. LUT reference is a classical acceleration method. However, the difference between the growth rate in computing speed and memory I/O speed has limited the scope of utilization of LUT references. Speedup would be possible if registers could be used as LUTs, but their small size makes them difficult to utilize. Therefore, this study proposes a downsampling method to fit LUTs into SIMD registers, which are relatively large and an efficient reference method for register-LUTs. Experimental results show that the proposed method can reproduce an accuracy in PSNR of 65.52 (+25.11) dB, while a simple full-size LUT in the register size can only reproduce 40.41 dB. Using a wider register width, the PSNR was 78.63 (+38.22) dB with AVX-512 and 84.5 (+44.09) dB with bfloat16. The fastest proposed method was on average 4.82/3.72 times faster than direct vector computing, 2.99/3.10 times faster than vector addressing, and 3.79/7.80 times faster than scalar addressing on the AVX2/AVX-512 computers while exceeding the display limit of 60 dB for 8-bit displays. Taking into account these speed/accuracy trade-offs, the performance of the proposed method was superior. This paper shows that LUT references can be realized with small SIMD registers in convolution. The proposed method is expected to be extended to adaptive filters, convolutional neural networks, and other image processing applications by accelerating the approximation with this register-LUT. Our code is available at https://fukushimalab.github.io/registerLUT4conv/.

INDEX TERMS Approximate computing, bilateral filtering, high-dimensional kernel filtering, high-performance computing, image filtering, nonlinear filters, parallel processing, SIMD, table lookup.

I. INTRODUCTION

Convolution is at the core of image processing and is used in various ways. For example, convolution is used in spatial invariant convolution (e.g., Gaussian and Laplacian filters), block spatial variation convolution (e.g., convolutional neural networks (CNN)), and variable-weighted convolution (e.g., adaptive filters and edge-preserving smoothing). These convolutions are essential tools for image processing applications.

The precomputed weights of spatial invariant and block variation convolutions are often prepared as lookup tables

The associate editor coordinating the review of this manuscript and approving it for publication was Tomas F. Pena.

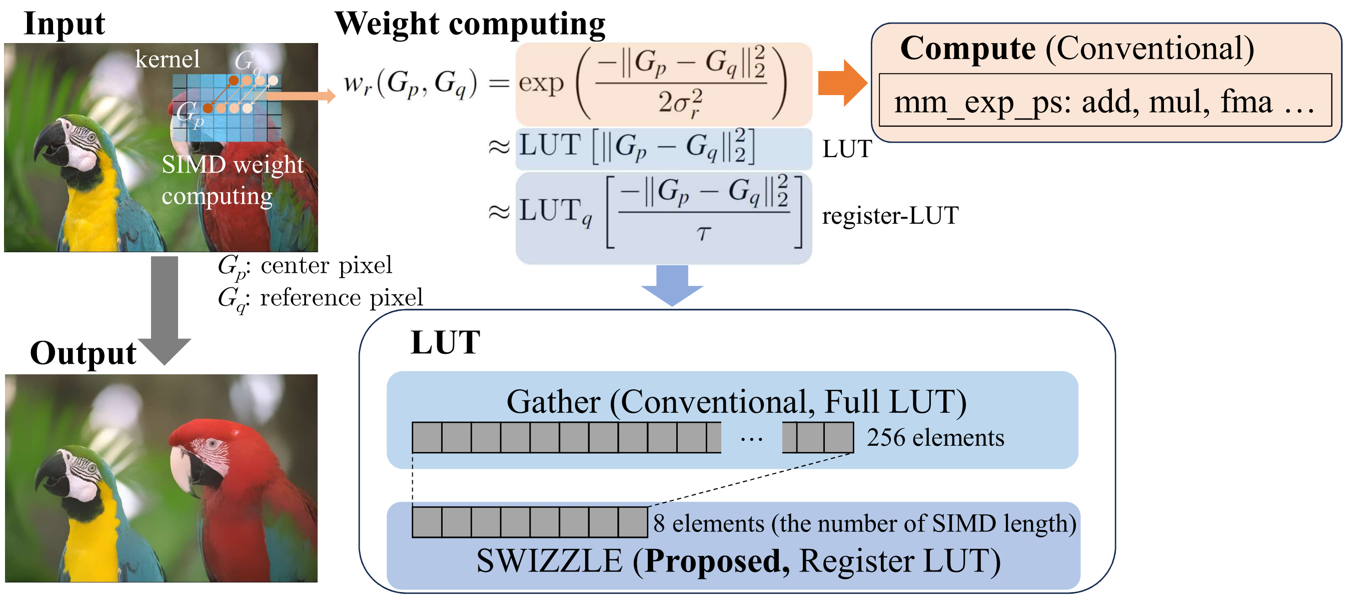

Input
kernel
G_q
G_p
SIMD weight computing
G_p: center pixel
G_q: reference pixel
Output
Weight computing
$w_r(G_p, G_q) = \exp\left(\frac{-\|G_p - G_q\|_2^2}{2\sigma_r^2}\right)$
$\approx \mathrm{LUT}\left[\|G_p - G_q\|_2^2\right]$ LUT
$\approx \mathrm{LUT}_q\left[\frac{-\|G_p - G_q\|_2^2}{\tau}\right]$ register-LUT
Compute (Conventional)
mm_exp_ps: add, mul, fma …
LUT
Gather (Conventional, Full LUT)
256 elements
8 elements (the number of SIMD length)
SWIZZLE (Proposed, Register LUT)

FIGURE 1. Overview of vector computing of weights between pixels: the conventional method by computation and by LUT with vector addressing (gather), the proposed method realizes a table that fits into a SIMD register size with SWIZZLE instructions.

(LUTs). Whereas variable-weight convolution is determined on a pixel-by-pixel basis and thus incurs an overhead if kept as weights; this overhead cannot be ignored like in CNN's im2col [1] due to the large convolution radius [2]. Therefore, weights are often computed numerically online, or only a portion is converted to LUTs offline to reduce overhead [3]. However, table lookups are becoming obsolete with the evolution of computer architecture by Moore's law because of the growth gap between computing performance and the memory I/O speed, and recalculation is becoming more efficient than LUTs. Although there are specific circuits that improve LUT efficiency, such as vector addressing with single instruction/multiple data (SIMD), the gap still needs to be fully closed. Cycles per instruction (CPI) of vector addressing is less than that of arithmetic computing, since vector addressing places the LUTs in memory (L1,2,3 cache or DRAM). The difference is especially noticeable when using instructions with long vector lengths, such as AVX-512, and the speedup is suppressed. This makes the choice between direct numerical and LUT calculations difficult [4].

While the gap widens, the memory devices that maintain speed are the registers equipped on the CPU. The registers are small, but the SIMD registers are dozens of times larger than the regular registers. In addition, they have high-speed SIMD instructions for register replacement. Therefore, it can also hold a certain amount of values as LUTs. However, its size is not large enough.

In this paper, we propose an approximate acceleration method to realize a partial LUT for variable-weighted convolution using SIMD registers. This paper focuses mainly on high-dimensional kernel filters (HDKF) [5], [6], [7], [8], which is a variable-weighted convolution for edge-preserving filtering [9]. HDKF are generalized forms for bilateral filters [10], joint bilateral filters [11], [12], multilateral filters [13], [14], joint bilateral upsampling [15], and non-local mean filters [16]. HDKF has various image processing applications. HDKFs are used in various image processing applications, such as denoising [17], deblurring [18], detail enhancement [19] and manipulation [20], high-dynamic-range imaging [21], [22], haze removal [23], low-light image manipulation [24], alpha matting [25], stereo matching [26], and optical flow estimation [27].

Typically, LUT for HDKF requires several KB in size. That is much smaller than the several GB required by im2col but still far less than the register size (e.g., 32B for AVX and 64B for AVX-512). Therefore, the paper proposes a LUT quantization method to keep it within the register size, maintaining high accuracy. In addition, we propose a fast LUT manipulation method using swizzle instructions.

The contributions of this paper are the following. We propose three lookup methods for the register-LUT and six techniques for their generation. The proposed lookup register-LUTs includes permute (Sec. IV-A), shuffle (Sec. IV-B), and permute with bfloat16 (Sec. IV-C). The proposed generating LUT techniques are LUT quantization (Sec. IV-D1), truncation (Sec. IV-D2), pre-division (Sec. IV-D3), LUT downsampling with Gauss integral (Sec. IV-E1), LUT tail specialization (Sec. IV-E2), and optimization method for quantization step (Sec. IV-F). Fig. 1 shows the flow of vector computing of weights, including the conventional computing and LUT approaches and the proposed register-LUT approach.

II. HIGH-DIMENSIONAL KERNEL FILTER

This section introduces HDKF as a variable-weighted convolution, which includes bilateral filtering and non-local mean filtering and is effectively implemented in this paper. Let the input and output images be $\boldsymbol{I}$ and $\boldsymbol{O}$: $\Omega \to [0, R]^c$, where $\Omega \subset \mathbb{N}^2$ is the spatial domain and $[0, R]^c$ is the range domain. Usually, $R = 255$ for unsigned char data, $c = 1$ for grayscale images, and $c = 3$ for color images. In addition, let the guidance image be $\boldsymbol{G} : \Omega \to [0, R]^d$, which is used to calculate the convolutional weight. For gray and color processing, $d = 1, 3$; for non-local mean filtering, d is a patch size. The values of pixels in a position vector $\boldsymbol{p}, \boldsymbol{q} \in \Omega$ (that is, $\boldsymbol{p} = (x, y)$) are represented by $\boldsymbol{I_p}$, $\boldsymbol{O_p}$ and $\boldsymbol{G_p}$. The HDKF output at $\boldsymbol{p}$ is defined as follows:

$$\boldsymbol{O_p} = 1/\eta_{\boldsymbol{p}} \sum_{\boldsymbol{q} \in \mathcal{S}_{\boldsymbol{p}}} w_s(\boldsymbol{p}, \boldsymbol{q}) w_r(\boldsymbol{G_p}, \boldsymbol{G_q}) \boldsymbol{I_q}, \tag{1}$$

$$\eta_{\boldsymbol{p}} = \sum_{\boldsymbol{q} \in \mathcal{S}_{\boldsymbol{p}}} w_s(\boldsymbol{p}, \boldsymbol{q}) w_r(\boldsymbol{G_p}, \boldsymbol{G_q}), \tag{2}$$

where $\mathcal{S}_{\boldsymbol{p}} \in \Omega$ is neighboring pixels around $\boldsymbol{p}$. Weight functions $w_s : \mathbb{N}^2 \times \mathbb{N}^2 \to \mathbb{R}$ and $w_r : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ are spatial and range domain weights, and these are typically defined as the Gaussian distribution:

$$w_s(\boldsymbol{p}, \boldsymbol{q}) = \exp\left(\frac{-\|\boldsymbol{p} - \boldsymbol{q}\|_2^2}{2\sigma_s^2}\right), \tag{3}$$

$$w_r(\boldsymbol{G_p}, \boldsymbol{G_q}) = \exp\left(\frac{-\|\boldsymbol{G_p} - \boldsymbol{G_q}\|_2^2}{2\sigma_r^2}\right), \tag{4}$$

where σ_s and σ_r are the distribution parameters. The set of space weights of (3) in $\mathcal{S}_{\boldsymbol{p}}$ are constant for each pixel $\boldsymbol{p}$, while the set of range weights of (4) are variable depending on the state of the image $\boldsymbol{G}$. The kernel weights are not limited to the Gaussian, but decay weights are usually used (e.g., Laplacian distribution and Hamming window). The following papers introduce the various forms for HDKF [5], [6], [7], [8].

HDKFs have various acceleration algorithms for gray [21], color [28], and more high-dimensional cases [5]. However, this paper focuses on the naïve algorithm, which is suitable for small and medium kernel sizes with parallel computers.

III. IMPLEMENTATION PATTERNS OF HDKF

We present design patterns for high-performance convolution codes using parallelization and vectorization. Sec. III-A introduces a plain code for parallelized and vectorized computing. Sec. III-B shows its vectorized code. Sec. III-C introduce the LUT approach instead of using computation. Sec. III-D shows the vector addressing for the SIMD LUT processing.

A. PARALLELIZED PATTERNS

SIMD and multi-thread parallelization are essential for high-performance image processing on CPUs. Convolution in image processing has various parallelization patterns. The most efficient pattern is to split the height loop of the image for parallelization and to unroll the width loop of the image for vectorization [2].

```
//range weight
float wr(Image G, int p, int q, float s)
{
    float sub = G(p)−G(q);
    float dist = sub∗sub;
    return exp(sub/(−2.0∗s∗s))
}

//filtering
void bilateralVector(Image I, Image G, Image O,
 int r/*radius*/,
 float s/*sigma_space*/,
 float sr /*sigma_range*/)
{
 //spatial table setup
 float sLUT[(2∗r+1)∗(2∗r+1)];
 int kc = width∗r/2+r/2;
 for(int l=−r;l<=r;l++){
  for(int k=−r;k<=r;k++){
   int q = width∗l+k;
   sLUT[q+kc] = exp((k∗k+l∗l))/(−2.0∗ss);
  }
 }
 //LUT-based filtering
 int simdwidth = 8; //for AVX2
 #pragma omp parallel for //multi-thread
 for(int j=0;j<height;j++){
 //OpenMP directive vectorization is possible , e.g., #pragma omp simd simdlen(8)
  for(int i=0;i<width;i+=simdwidth){
   int p = width∗j+i;
   float v[simdwidth];//numerator
   float w[simdwidth];//denominator
     for(int l=−r;l<=r;l++){
      for(int k=−r;k<=r;k++){
       int q = width∗l+k;
       //spatial weight
       float ws = sLUT[q+kc];
       //inner kernel loop
       //loop unrolling instead of intrinsics
       //numerator
       v[0]+=ws∗wr(G,q,p+0,sr)∗I(p+q+0);
       v[1]+=ws∗wr(G,q,p+1,sr)∗I(p+q+1);
       v[2]+=ws∗wr(G,q,p+2,sr)∗I(p+q+2);
       v[3]+=ws∗wr(G,q,p+3,sr)∗I(p+q+3);
       v[4]+=ws∗wr(G,q,p+4,sr)∗I(p+q+4);
       v[5]+=ws∗wr(G,q,p+5,sr)∗I(p+q+5);
       v[6]+=ws∗wr(G,q,p+6,sr)∗I(p+q+6);
       v[7]+=ws∗wr(G,q,p+7,sr)∗I(p+q+7);
       //denominator
       w[0]+=ws∗wr(G,q,p+0,sr);
       w[1]+=ws∗wr(G,q,p+1,sr);
       w[2]+=ws∗wr(G,q,p+2,sr);
       w[3]+=ws∗wr(G,q,p+3,sr);
       w[4]+=ws∗wr(G,q,p+4,sr);
       w[5]+=ws∗wr(G,q,p+5,sr);
       w[6]+=ws∗wr(G,q,p+6,sr);
       w[7]+=ws∗wr(G,q,p+7,sr);
      }
     }
    O[p+0] = v[0]/w[0];
    O[p+1] = v[1]/w[1];
    O[p+2] = v[2]/w[2];
    O[p+3] = v[3]/w[3];
    O[p+4] = v[0]/w[4];
    O[p+5] = v[1]/w[5];
    O[p+6] = v[2]/w[6];
    O[p+7] = v[3]/w[7];
   }
  }
 }
}
```

PROGRAM 1. **Bilateral filtering code for vectorization.**

```
...
//inner kernel loop
__m256 gp = _mm256_loadu_ps(G+p);
__m256 gq = _mm256_loadu_ps(G+q);
//distance computation
__m256 sub = _mm256_sub_ps(gp, gq);
__m256 dist = _mm256_mul_ps(sub, sub);

//weight computation
__m256 c = _mm256_set1_ps(−1.f/(2.f∗sigma∗sigma))
__m256 arg = _mm256_mul_ps(dist, c);
__m256 wr = _mm256_exp_ps(arg); //wr[0]...wr[7]
//set spatial weight and convolution
__m256 ws = _mm256_set1_ps(sLUT[q+kc]);
__m256 wrs = _mm256_mul_ps(ws, wr);
__m256 iq = _mm256_loadu_ps(I+q);
w = _mm256_fmadd_ps(wrs, iq, w);
```

PROGRAM 2. SIMD convolution with exp computing.

Program 1 is an example of this optimization for bilateral filtering (i.e., grayscale HDKF). Sometimes, the code is auto-vectorized as is code by compilers, but this paper manually tunes the code by SIMD intrinsics. If we use OpenMP to explicitly vectorize the code, then we can use the comment out line in Program 1's line 28, *#pragma omp simd simdlen(8)*. Note that the OpenMP directive parallelizes the j-loop in the code in Program 1's line 25. In Program 1's lines 15–22, the spatial weights are constant for each kernel position; thus, the weights can be precomputed without any loss. The variable factor is limited for range weights. In this paper, we focus on the variable part.

B. NUMERICAL COMPUTING FOR CONVOLUTION

The exponential function is required for the range weight computation (Program 1, lines 39–55). Intel Short Vector Mathematical Library (SVML) provides the SIMD exponential function, but no dedicated circuit exists. It is now supported by gcc, icc, and cl (Visual Studio). For vectorizing lines 39-55 in Program 1, we use *_mm256_exp_ps* function for range weight. The SIMD code for these parts is shown in Program 2. We can replace the *_mm256_exp_ps* intrinsics by the other library such as *fmath* library[1] and Agner Fog's Vector Class Library.[2] This paper used fmath for additional usage.

C. LUT-BASED CONVOLUTION

We introduce LUT-based convolution instead of numerical computing. In this section, we omit the spatial weight for readability.

Here, we define the operator that refers to LUT. The set of distance candidates (i.e., index) is $\mathcal{I} = \{0, 1, \ldots\} \subset \mathbb{N}$ and the set of elements of the LUT is $\mathcal{L} \subset \mathbb{R}$. Each element in these set, $i \in \mathcal{I}$ and $l \in \mathcal{L}$, are mapped as follows:

$$i \mapsto l = \mathrm{LUT}[i], \tag{5}$$

[1]https://github.com/herumi/fmath

[2]https://github.com/vectorclass/version2

where the operator LUT : $\mathcal{I} \to \mathcal{L}$, the size of set $|\mathcal{I}|$ and $|\mathcal{L}|$ are the same, and the max value of $\mathcal{I}$ is $|\mathcal{L}| - 1$ denoted as υ. Let an input vector be $\boldsymbol{x}$. The distance operator $d(\cdot) \in \mathcal{L}$, such as the ℓ_2 norm, computed the index value.

$$i = \mathrm{round}(d(\boldsymbol{x})) \tag{6}$$

where round$(\cdot) : \mathbb{R} \to \mathbb{N}$ is rounding operator.

Using (1), (5) and (6), the LUT-based convolution is defined by replacing w_r function with LUT operator:

$$\boldsymbol{O_p} = \sum_{\boldsymbol{q} \in \mathcal{S}_{\boldsymbol{p}}} \mathrm{LUT}[\mathrm{round}(d(\boldsymbol{G_p} - \boldsymbol{G_q}))]\boldsymbol{I_q}. \tag{7}$$

Here, we omit the spatial weight w_s and the normalization factor $1/\eta_{\boldsymbol{p}}$ in (1) to focus on the range weight. The LUT approaches can include pre-computation in arguments of the exponential function. We introduce two types of distance function d: linear mapping and root mapping.

The first is linear mapping, which includes the normalization in the argument and is defined as follows:

$$\mathrm{LUT}[i] := \exp\left(-\frac{i}{2\sigma^2}\right), \quad d(\boldsymbol{x}) := \|\boldsymbol{x}\|_2^2 \tag{8}$$

The argument in linear mapping is square distance; thus, the value tends to be significant. For the gray case, the LUT size is $255^2 = 65535$. For the color case, $r^2+g^2+b^2 = (255^2)*3 = 195,075$.

The second is root mapping, which additionally includes the square operator in the argument.

$$\mathrm{LUT}[i] := \exp\left(-\frac{i^2}{2\sigma^2}\right), \quad d(\boldsymbol{x}) := \|\boldsymbol{x}\|_2. \tag{9}$$

The linear mapping must compute the ℓ_2 norm, computed by the distance computation in the root operator. If an input for the distance function is scalar (i.e., grayscale), the distance function becomes absolute difference;

$$d(\boldsymbol{x}) := |x|. \tag{10}$$

Since the argument of a Gaussian function grows as the square of its size, linear mapping has a finer grading of the larger portions. Therefore, we first take the root and linearize the arguments to reduce the LUT size without quantization. The LUT size becomes 256 for the grayscale case and 442 ($\lceil\sqrt{195,075} = 441.67\ldots\rceil$) for the color case.

Color processing involves this linearization process, which adds a root operation to the distance calculation and increases the cost. On the other hand, direct calculation, such as calling *_mm256_exp_ps*, does not require a root operation because the square can be directly entered as an argument.

Note that subnormal numbers should be avoided for LUT values such that the small value does not contribute to accuracy, but significantly reduces speed [3].

D. VECTOR ADDRESSING IMPLEMENTATION

Vector addressing is the SIMD operation for referring to LUTs. Program 3 is part of a convolution that uses vector addressing for range weights. The intrinsic vgatherdps (*_mm256_i32gather_ps*) is for the vector addressing. Instead of computing range weights by *_mm256_exp_ps*, the table reference reduces its computational cost. Scalar addressing is also possible using the *mm256_set_ps* intrinsic, a macro for insert and extract instructions. Scalar addressing resolves table lookup for SIMD register element by element In Program 3 line 25, the function of scalar addressing is commented out. The performance of vector addressing depends on computer architectures; thus, scalar addressing is sometimes effective.

```
 1  //LUT update (root mapping)
 2  float coeff = −1.0/(2.f*s*s);
 3  for(int i=0;i<256;i++)
 4  {
 5      table[i]=exp(i*i*coeff)
 6  }
 7  ...
 8  //inner kernel loop
 9  __m256 gp = _mm256_loadu_ps(G+p);
10  __m256 gq = _mm256_loadu_ps(G+q);
11  //root mapping for gray
12  __m256 sub = _mm256_sub_ps(gp, gq);
13  __m256 dist = _mm256_abs_ps(sub);//abs
14
15  //vector addressing for weights wr[0]...wr[7]
16  __m256i index = _mm256_cvtps_epi32(dist);
17  //wr[0]=table[index[0]];
18  //wr[1]=table[index[1]];
19  //...
20  //wr[7]=table[index[7]];
21  __m256 wr = _mm256_i32gather_ps(table,index,4);
22  //4: sizeof(float)
23
24  //scalor access case: table->t, index->i
25  //__m256 wr = _mm256_setr_ps(t[i[0]],t[i[1]],
        t[i[2]],t[i[3]],t[i[4]],t[i[5]],t[i[6]],t
        [i[7]]);
```

PROGRAM 3. Vector addressing.

Vector addressing is used for various applications, such as the SIMD lookup table library for the global navigation satellite system (GNSS) correlator [29], a biological signal processing of heart simulation [30], a simplification of a real-world system of hydrologic model [31] (e.g., surface water, soil water, wetland, groundwater, estuary), and image and tensor processing [2], [32], [33].

IV. PROPOSED QUANTIZED LUT

The SIMD swizzle instructions rearrange the elements in registers according to specified rules. The category of the swizzle instructions include blend, broadcast, compress, expand, extract, insert, permute, shuffle, and unpack in Intel Intrinsics Guide.[3] This reordering can be used to refer to LUTs in SIMD registers. The problems of the register-LUT

```
__m256 lut8_permute(__m256 dist, __m256 rtbl, __m256i m7)
{
    __m256i a = _mm256_cvtps_epi32(dist);
    __m256i b =_mm256_min_epi32(m7, a);
    __m256i c =_mm256_permutevar8x32_ps(rtbl, b);
    return c;
}

__m256 lut16_shuffle(__m256 dist, __m256i rtbl, __m256i m15,
    __m256i mask)
{
    __m256i a = _mm256_cvtps_epi32(dist);
    __m256i b = _mm256_min_epi32(m15, a);
    __m256i c = _mm256_shuffle_epi8(rtbl, b);
    __m256i d = _mm256_andnot_si256(mask, c);
    __m256i e = _mm256_cvtepi32_ps(d);
    return e;
}
```

PROGRAM 4. Register-LUT functions (AVX2).

are how to refer to the register-LUT, and the number of elements in the LUT is limited to the number of elements in the SIMD register. First, we introduce two types of register-LUT reference: permute and shuffle. Second, we explain how to fit a larger LUT into the register-LUT size.

Before a detailed explanation, the actual permute and shuffle codes are shown in Program 4 (AVX2) and Program 5 (AVX-512). These functions can be used in place of the gather intrinsic in Program 3 for the approximated acceleration. Table 1 is a datasheet of Intel Skylake microarchitecture for the intrinsics used. The permute and shuffle intrinsics are faster than the gather vector addressing. The other microarchitecture information can be obtained at the uops site [34].[4]

TABLE 1. Datasheet of intrinsics. L: latency and T: throughput of intel cascade lake microarchitecture.

asm	intrinsic	L	T	AVX-512
vcvtps2dq	cvtps_epi32	4	0.5	
vcvtdq2ps	cvtepi32_ps	4	0.5	
vpminsd	min_epi32	1	0.5	
vpandn	andnot_si256	1	0.33	
vslld	slli_epi32	1	1	
vpermps	permutexvar8x32_ps	3	1	
vpermps	permutexvar_ps	3	1	✓
vpermi2ps	permutex2var_ps	3	1	✓
vpermi2w	permutex2var_epi16lph	7	2	✓
vpshufb	shuffle_epi8	1	0.5	
vgatherdps	i32gather_ps	22	5	
vblendps	blend_ps	1	0.33	
vblendmps	mask_blend_ps	1	0.5	✓
vpcmpgtd	cmpgt_epi32	1	0.5	
vpcmpd	cmpgt_epi32_mask	3	1	✓

A. INSTRUCTION: PERMUTE

Using AVX2, the vpermps (*permutevar*8×32*ps*) instruction can reorder 8 float-type elements by 8 int-type elements, where the int-type indices are within 0-7, and each reorder element can be specified arbitrarily, including duplications. Using AVX-512, 16 elements can be reordered by the vpermps (*permutexvar_ps*) instruction. In addition,

[3]https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[4]https://uops.info/

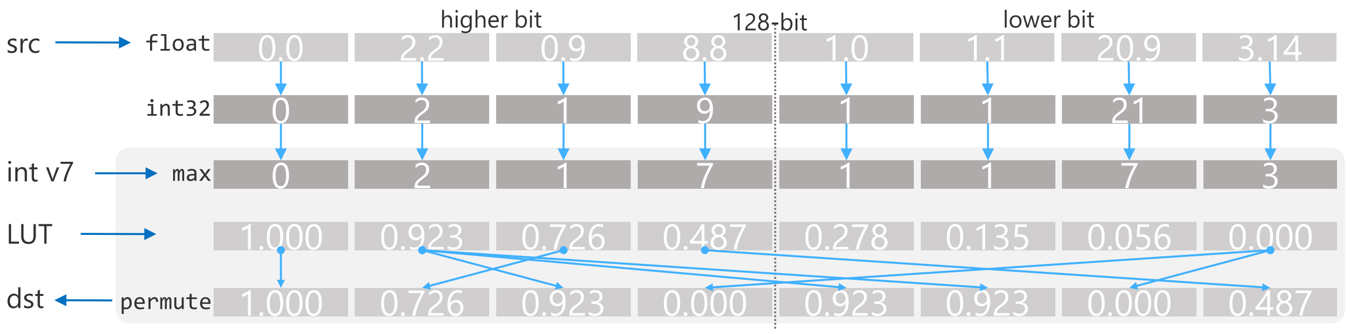


FIGURE 2. Register-LUT approach using permute (vpermps) intrinsic for 32-bit LUT (AVX2).

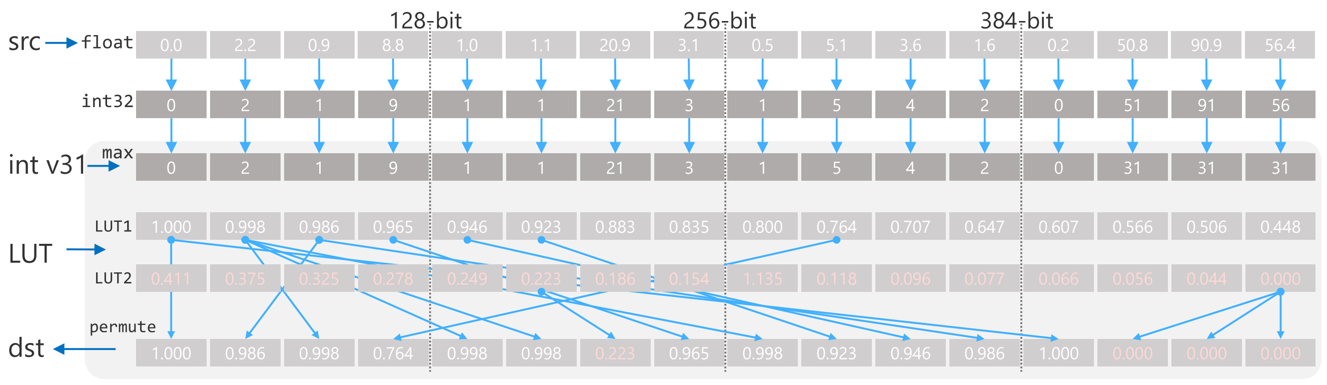


FIGURE 3. Register-LUT approach using permute (vpermi2ps) intrinsic for two 32-bit LUTs (AVX-512).

```
__m512 lut32_permute(__m512 dist, __m512 rtbl1, __m512 rtbl2,
        __m512i m31)
{
    __m512i a = _mm512_cvtps_epi32(dist);
    __m512i b = _mm512_min_epi32(m31, a)
    __m512i c = _mm512_permutex2var_ps(rtbl1, b, rtbl2)
    return c;
}

__m512 lut64_permute_bf(__m512 dist, __m512i rtbl1, __m512i
        rtbl2, __m512i m63)
{
    __m512i a = _mm512_cvtps_epi32(dist);
    __m512i b = _mm512_min_epi32(m63, a)
    __m512i c = _mm512_permutex2var_epi16(rtbl1, b, rtbl2);
    __m512i d = _mm512_slli_epi32(c, 16);
    return _mm512_castsi512_ps(d);// no cost
}

__m512 lut16_shuffle(__m512 dist, __m512i rtbl, __m512i m15,
        __m512i mask)
{
    __m512i a = _mm512_cvtps_epi32(dist);
    __m512i b = _mm512_min_epi32(m15, a);
    __m512i c = _mm512_shuffle_epi8(rtbl, b);
    __m512i d = _mm512_andnot_si256(mask, c);
    __m512i e = _mm512_cvtepi32_ps(d);
    return e;
}
```

PROGRAM 5. Register-LUT functions (AVX-512).

vpermi2ps (*permutex2var_ps*) instruction can refer to two register tables by a single register index with 0-31 values. In other words, looking up 32-element tables can be achieved by a single instruction. These instructions have a latency of 3 and a throughput of 1; thus, changing AVX2 to AVX-512 quadruples the table size at the same CPI. Usually, performance scalability from AVX2 to AVX-512 is double [35], but the size scalability is quadruple.

Figures 2 and 3 present visual examples for AVX2 and AVX-512 cases, respectively. The processing of AVX2 has three processing chains:

1) Convert float type to int type for index
2) Truncate index with an integer value of 7
3) Permute a LUT with an index

In addition, the AVX-512 case is as follows:

1) Convert float type to int type for index
2) Truncate index with an integer value of 31
3) Permute two LUTs with an index

If we use more register tables to refer to a larger LUT, we can combine the results of multiple register table lookups. The practical implementation is as follows. First, the permute instruction references the register tables 1 and 2, ignoring that the index value is greater than 8. The swizzle instruction's index value is the remainder of the number of register elements. For example, index=0 and index=8 in the vpermps instruction have the same semantics. Next, a mask is created to indicate whether the index value exceeds 8 or 16, respectively. Finally, the results are blended according

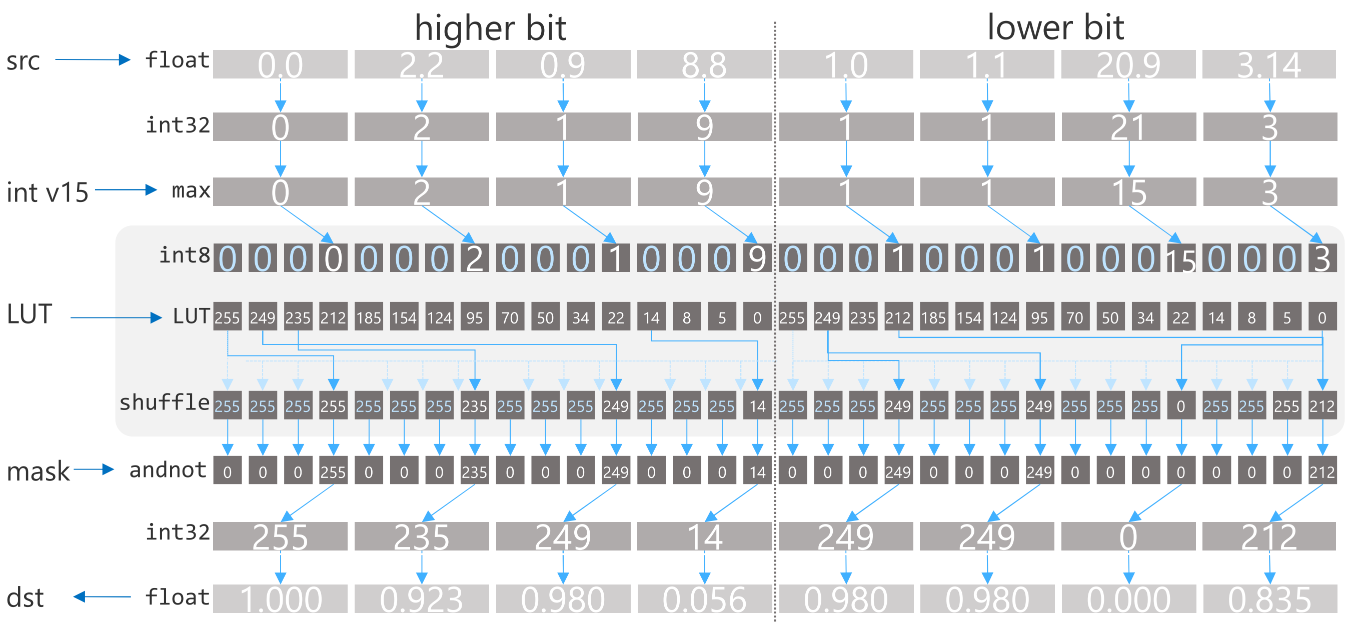


FIGURE 4. Register-LUT approach using shuffle intrinsic for 8-bit LUTs (AVX2).

to the masks to enable multi-register table lookups. This implementation requires a permute instruction for the number of tables. In addition, compare and blend instructions for the number of tables minus one. This paper combines up to three register tables (i.e., 8, 16, and 24 elements for AVX2 and 32, 64, and 96 for AVX-512).

```
__m256 lut24_permute(__m256 dist,
                  __m256 rtbl1, __m256 rtbl2, __m256 rtbl3,
                  __m256i m7, __m256i m15, __m256i m23)
{
   __m256i a = _mm256_cvtps_epi32(dist);
   __m256i b = _mm256_min_epi32(m23, a);
   __m256i c = _mm256_permutevar8x32_ps(rtbl1, b);
   __m256i d = _mm256_permutevar8x32_ps(rtbl2, b);
   __m256i e = _mm256_permutevar8x32_ps(rtbl3, b);
_m256i m=_mm256_castsi256_ps(_mm256_cmpgt_epi32(b,m7));
_m256i n=_mm256_castsi256_ps(_mm256_cmpgt_epi32(b,m15));
   __m256i f = _mm256_blendv_ps(c, d, m);
   __m256i g = _mm256_blendv_ps(f, e, n);
   return g;
}
```

PROGRAM 6. Merging three register-LUTs (AVX2 permute).

Program 6 shows the case of merging three register-LUTs by the permute intrinsics on AVX2 CPUs. The program performs three permutes and two compares and blends. The registers m7, m15, and m23 contain constant values for each register.

B. INSTRUCTION: SHUFFLE

The shuffle instruction can also refer to register-LUTs. The shuffle instruction can move elements within a 128-bit register lane. In other words, it can move only the same number of elements as the SSE instructions. Even if AVX2 or AVX-512 is used, the range of movement is not increased, but the throughput is 2 or 4 times higher. In addition, only the 8-bit element moving instruction, vpshufb (shuffle_epi8), can be used, and it is impossible to move a float array dynamically. Therefore, the 32-bit float table should be converted to an 8-bit integer table to make it a 16-element LUT reference. The type conversion is defined as follows:

$$\mathrm{LUT}_{8\mathrm{u}}[i] = \mathrm{round}(255 \cdot \mathrm{LUT}[i]), \tag{11}$$

where $\cdot$ is the scalar multiply operator.

More elements (16 elements) can be referenced in the 8-bit table (8 elements) than permuting the 32-bit table in the AVX2 case. However, there is an overhead of type conversion from integer to floating point after the LUT reference. In addition, the quantization reduces accuracy. Therefore, it depends on the computer architecture whether it is better to perform a two-element LUT or this approach.

Figure 4 shows the process steps. As a preprocessing, the float-type LUT is converted to 8-bit integers. In the AVX2 case, we replicate the register-LUTs to the first and second half lanes; in the AVX-512 case, we replicate them to four lanes. In the case of SSE, no replication is required (i.e., only one lane, 128-bit, is required). The processing flow has five processing chains:

1) Convert float type to int type for index
2) Truncate index with a 32-bit integer value of 15 and regard the index as 8-bit char type (not cast)
3) Perform shuffle_epi8 to a LUT with index
4) Bitmask to clear unnecessary values to 0
5) Return to float type

The shuffle instruction can apply to SSE-only computers (e.g., Intel Atom), but SSE CPUs do not have pemute

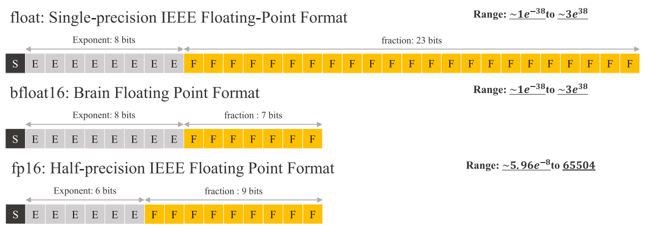

float: Single-precision IEEE Floating-Point Format
Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$
Exponent: 8 bits
fraction: 23 bits
S E E E E E E E E F F F F F F F F F F F F F F F F F F F F F F F
bfloat16: Brain Floating Point Format
Range: $\sim 1e^{-38}$ to $\sim 3e^{38}$
Exponent: 8 bits
fraction : 7 bits
S E E E E E E E E F F F F F F F
fp16: Half-precision IEEE Floating Point Format
Range: $\sim 5.96e^{-8}$ to 65504
Exponent: 6 bits
fraction : 9 bits
S E E E E E E F F F F F F F F F

FIGURE 5. **Floating-point value representations for 32-bit float, bfloat16 and fp16.**

instructions, which were introduced from AVX. In addition, SSE CPUs do not have the gather intrinsic.

Similarly, the shuffle can superimpose multiple register-LUTs with the compare and blend instructions, such as the permute.

C. INSTRUCTION: PERMUTE WITH BFLOAT16

In AVX-512, we can use the 16-bit permute intrinsic, permi2w (*permutex2var_epi16*), but the intrinsic is used for 16-bit integer registers. However, the bitwise movement of elements is not affected by type, as long as the number of bits is consistent.

There are two well-known 16-bit floating-point types: bfloat16 and fp16. Fig. 5 shows each type. The difference between a float and a bfloat16 is the number of fraction bits, whereas the difference between a float and a bfloat16 is both the exponent and fraction bits. The bfloat16-type can be used in AVX512BF16-supported CPUs: Cooper Lake, Alder Lake, Sapphire Rapid, and Zen 4. The AVX512BF16 category only contains type conversion and dot-product arithmetics for bfloat16. The fp16-type can be used in AVX512FP16-supported CPUs: Alder Lake and Sapphire Rapid. The AVX512FP16 category contains almost all 16-bit instructions with the same functions as those supported by 32-bit floats. However, only a limited number of CPUs support these 16-bit floating-point instructions.

Therefore, in this paper, we propose an effective software implementation of bfloat16 for LUT reference. This implementation can be used for all AVX-512 CPUs. The conversion of the type from 32-bit float to bfloat16 is just truncating the lower-bit part, whereas fp16 requires complex operations. The fraction bits range from 23 bits to 7 bits, with 16 bits in the fraction bits. By this conversion, we can store 32 elements of 16-bit floating-point value in 512-bit AVX-512 registers. Then, we refer to register-LUTs using permutex2var_epi16. A 16-bit shift to the left is needed to return to a 32-bit float by padding the lower bits to 0 while clearing the dirty flags in the upper bits.

Figure 6 shows the process steps. The processing flow has four processing chains:

1) Convert float type to int type for index
2) Truncate index with a 32-bit integer value of 63 and regard the index as 16-bit short type (not cast)
3) Perform permutex2var_epi16 to two bfloat16 register-LUTs with the index
4) Left bit-shift to move sign and exponent bits to higher bit with zero-padding for lower-bit (resulting data can be regarded as 32-bit float).

D. QUANTIZATION, TRUNCATION, AND PRE-DIVISION

When we refer to a table, the index does not always fit within the upper limit of the number of LUT elements. For example, the absolute value of the difference of luminance values is within 0-255, whereas the size of a register table is 8 in the permute on AVX2. Therefore, we need additional processes if the difference is larger than the table size.

1) QUANTIZATION

A simple way to keep it within the number of LUT elements is to divide and round the index by the number of LUT elements for quantization. The index quantization function q is defined as follows:

$$q(i,\tau) = \mathrm{round}(i/\tau), \tag{12}$$

where τ is a division step. The number of LUT elements can be small enough to fit in the register size by adjusting *tau*.

The most straightforward approach for the LUT quantization is the nearest neighbor quantization. Note that advanced quantization will be presented in the following subsection. The nearest neighbor approach quantizes the input LUT by τ:

$$\mathrm{LUT_q}[i] = \mathrm{LUT}[\tau * i] = \exp\left(\frac{(i\tau)^2}{-2\sigma^2}\right). \tag{13}$$

For example, we convert an LUT with 256 elements to 8 elements by setting $\tau = 64$. The index is then divided by τ, and the value always falls within 0-7. Using (12) and (13), the LUT-based convolution is as follows:

$$\boldsymbol{O_p} = \sum_{\boldsymbol{q}\in\mathcal{S}_p} \mathrm{LUT_q}[q(d(\boldsymbol{G_p}-\boldsymbol{G_q}),\tau)]\boldsymbol{I_q}. \tag{14}$$

However, this method quantizes LUTs by a large τ, reducing precision.

2) TRUNCATED QUANTIZATION

To overcome the problem of large quantization, we quantize LUTs with truncation, named *truncated quantization*. Gaussian distribution has long tails, and the responses in the tail are almost zero. Therefore, we ignore the tail regions by truncating the index argument. Here, let the upper limit of the index value be $\upsilon = |L|-1$. The index value for the truncated quantization is defined as follows:

$$tq(i,\tau,\upsilon) = \min(\mathrm{round}(i/\tau),\upsilon), \tag{15}$$

Using (15), the LUT-based convolution is as follows:

$$\boldsymbol{O_p} = \sum_{\boldsymbol{q}\in\mathcal{S}_p} \mathrm{LUT_q}[tq(d(\boldsymbol{G_p}-\boldsymbol{G_q}),\tau,\upsilon)]\boldsymbol{I_q}. \tag{16}$$

This approach can suppress the large *tau* value by ignoring the distribution tails. Both approaches of (14) and (16) require

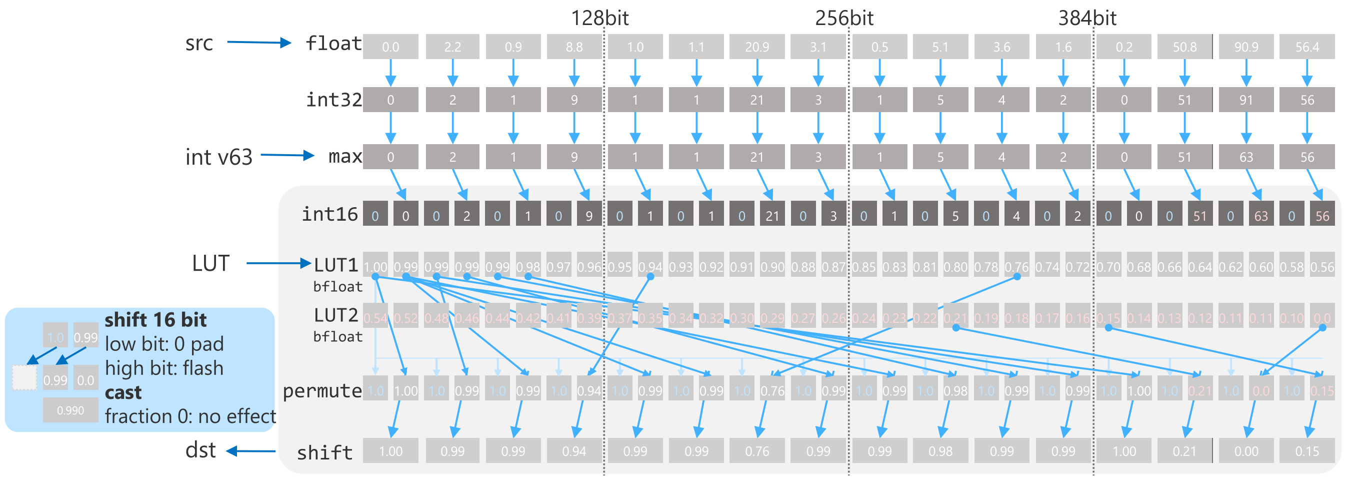


FIGURE 6. Register-LUT approach using permute (vpermi2w) intrinsic with bfloat16 for 16-bit LUTs (AVX-512).

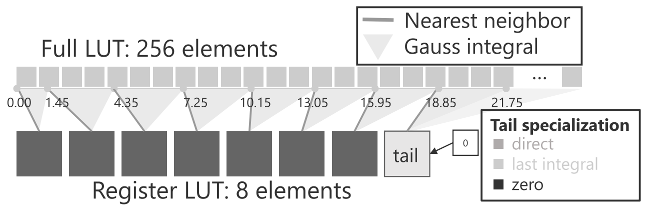


FIGURE 7. LUT quantization ($\tau = 2.9$).

division (reciprocal multiplication) for each computing index in (12) and (15).

3) PRE-DIVISION

For acceleration, we can remove the division factor in q and tq by dividing the guidance image $\boldsymbol{G}$ by τ to shrink the range domain, $\boldsymbol{G}' = \boldsymbol{G}/\tau$: $[0 : 255] \rightarrow [0 : 255/\tau]$. This pre-division removes division operators in the kernel processing for indexing;

$$t(i, \upsilon) = \min(\mathrm{round}(i), \upsilon). \tag{17}$$

Finally, the LUT-based convolution is defined as follows:

$$\boldsymbol{O}_p = \sum_{q \in \mathcal{S}_p} \mathrm{LUT_q}[t(d(\boldsymbol{G}'_p - \boldsymbol{G}'_q), \upsilon)]\boldsymbol{I}_q, \tag{18}$$

E. GAUSS INTEGRAL AND TAIL SPECIALIZATION

Quantizing full LUT with the nearest neighbor method ignores many skipped elements. Additionally, most tail elements are ignored. Fig. 7 visualizes the access. We consider the skipped elements by Gauss integral and tail specialization.

1) GAUSS INTEGRAL

We can use the Gauss integral to improve the accuracy of the quantized LUT instead of the nearest neighbor method. The definition of the Gauss integral is as follows:

$$\begin{aligned}\mathrm{LUT_{qg}}[i] &= \frac{1}{t_{i+1} - t_i} \int_{t_i}^{t_{i+1}} \exp\left(\frac{-x^2}{2\sigma^2}\right) dx \\ &= \sqrt{\frac{\pi\sigma^2}{2}} \left\{ \mathrm{erf}\left(\frac{t_{i+1}}{\sqrt{2\sigma^2}}\right) - \mathrm{erf}\left(\frac{t_i}{\sqrt{2\sigma^2}}\right) \right\}, \end{aligned} \tag{19}$$

where t_i and t_{i+1} are lower and upper limits for subsampled index i that covers full sample LUT. The function erf($\cdot$) is the error function. When we use round off, t_i is defined as follows:

$$t_i = \begin{cases} 0 & (i = 0) \\ 0.5\tau & (i = 1) \\ t_1 + (i-1)\tau & else. \end{cases} \tag{20}$$

The number t_i is denoted in Fig. 7 under the full LUT boxes. The method covers skipped elements.

2) TAIL SPECIALIZATION

We specialize the operation for the last element in the register table because the tail values are almost zero and the element supports the broader range of non-subsampled LUT. For the last index υ, we consider three cases. The first is direct setting using (19), named *last-direct*. The second is the last mean setting, named *last-mean*, which is defined as follows:

$$\mathrm{LUT_{qg}}[\upsilon] := \frac{1}{t_{i+1} - t_i} \int_{t_\upsilon}^{t_\infty} \exp\left(\frac{-x^2}{2\sigma^2}\right) dx \tag{21}$$

$$= \sqrt{\frac{\pi\sigma^2}{2}} \left\{ \mathrm{erf}\left(\frac{t_\infty}{\sqrt{2\sigma^2}}\right) - \mathrm{erf}\left(\frac{t_\upsilon}{\sqrt{2\sigma^2}}\right) \right\}, \tag{22}$$

The third is the zero setting, named *last-zero*.

$$\mathrm{LUT_{qg}}[\upsilon] := 0 \tag{23}$$

F. OPTIMIZING QUANTIZATION STEP

We propose an optimization method for the quantization parameter τ. For small values of τ, the LUT resolution

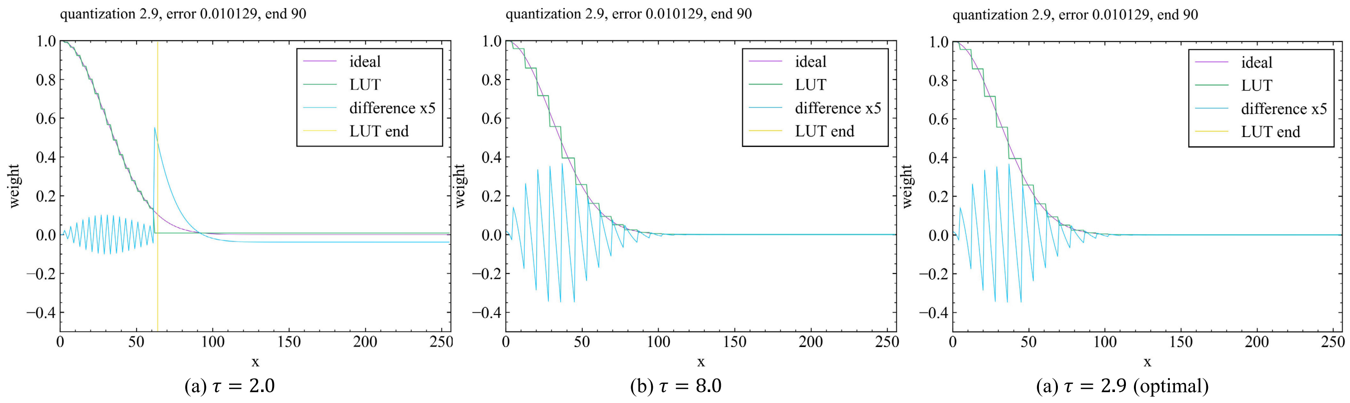


FIGURE 8. Kernel shapes and their $\times 5$ boosted errors with permute-32 when τ is varied (32 elements on AVX-512 for $\sigma = 30$).

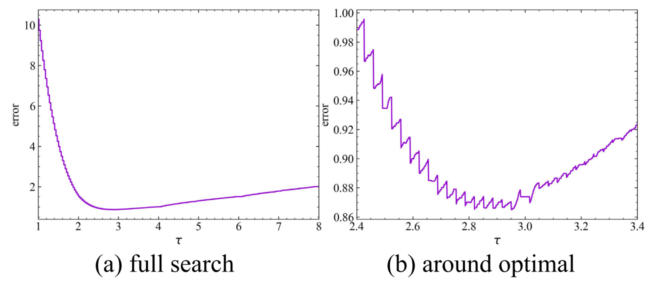


FIGURE 9. Error in linear search for the parameter τ (32 elements on AVX-512, $\sigma = 30$).

TABLE 2. Architecture used. *E-cores are disabled for using AVX-512.

Architecture	CPU	Vendor	SIMD
Sapphire Rapids	Xeon w9-3495X	Intel	AVX-512
Cascade Lake	Core i9-10980XE	Intel	AVX-512
Alder Lake	Core i9-12900K	Intel	AVX-512*
Rocket Lake	Core i9-11900K	Intel	AVX-512
Zen4	Ryzen 9 7950x	AMD	AVX-512
Alder Lake	Core i9-12900K	Intel	AVX2
Coffee Lake Refresh	Core i9-9900K	Intel	AVX2
Zen3	Ryzen 9 5950x	AMD	AVX2
Zen2	Ryzen 9 3950x	AMD	AVX2
Zen+	Ryzen 7 2700x	AMD	AVX2

near zero is higher, while the values near the tail of the LUT drop sharply to 0. For large values of τ, the resolution near 0 is lower, but the tail approaches zero smoothly. The proposed method minimizes the difference between the full sample LUT and the quantized and truncated LUT. We define the error function as follows and find the minimization argument τ:

$$\tau = \arg\min_{t} \sum_{i \in L} \left(\mathrm{LUT}[i] - \mathrm{LUT_X}[tq(i, t, \upsilon)]\right)^2, \tag{24}$$

where $\mathrm{LUT_X}$ represents arbitrary quantized LUTs (e.g., $\mathrm{LUT_q}$ and $\mathrm{LUT_{qg}}$). Figure 8 shows the kernel shapes and errors for each parameter τ. A smaller τ results in a larger tail error, and a larger τ results in a larger overall error. The optimum τ is the one that balances these two.

The parameter τ was optimized with a golden search. Actual errors oscillate due to quantization errors; hence, they can only be found if a full search is performed. However, τ is a real number, and a continuous linear search is too costly. Therefore, we used the golden search method, which has been experimentally successful. Figure 9(a) shows the linear search result, and the error is almost differential. Figure 9(b) shows the close of the linear search results around optimal, and the error oscillates, but the difference is small.

V. EXPERIMENTAL RESULTS

We conducted three experiments. The first evaluated the performance of quantization methods for LUTs. Second, we investigated the effect of filtering parameters on accuracy. The third evaluated the accuracy-speed trade-off among different computer architectures. Tab. 2 shows the computers used for each experiment. We used Visual Studio 2022 on Windows and OpenMP (/openmp:llvm option) for parallelization for each experiment. We used $\sigma = 3.0$ and $\sigma_r = 30$ as default parameters. We used peak-signal-noise-ratio (PSNR) as a metric and direct computing of exponential functions with $r = 6\sigma_s$ convolution as a ground truth for accuracy evaluation.

A. QUANTIZE LUT METHOD

Fig. 10 shows the effect of the proposed features enabled one by one. The label "442 clip" is the $\tau = 442/8 = 55.25$ with the nearest neighbor sampling with the last-direct case that is the simplest method just considering quantization (Sec. IV-D1). The label "180 or 90 clip" is the $\tau = 180/8 = 22.5$ or $\tau = 90/8 = 11.25$ with the nearest neighbor sampling with the last-direct case considering truncation (Sec. IV-D2). The label "opt. NN-direct" is the optimal τ case using the optimization step (Sec. IV-F). The label "opt. Gauss-direct" changes downsampling from the nearest neighbor to the Gauss integral (Sec. IV-E1) and "the opt. Gauss-mean" changes tail handling from direct to last-mean

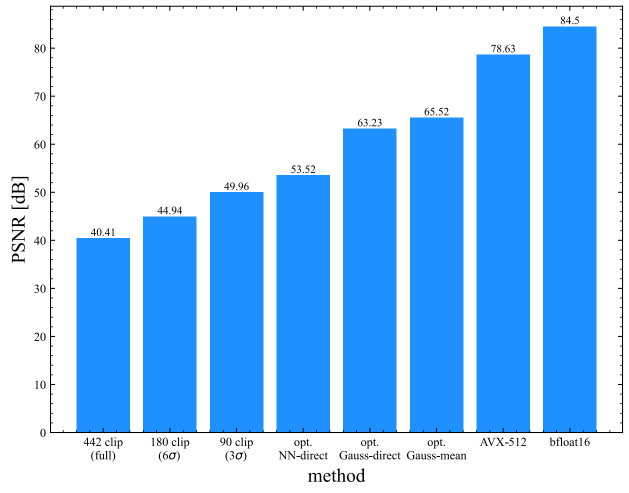


FIGURE 10. **Performance improvement when each feature is enabled. Color filtering ($\sigma_r = 30$, $\sigma_s = 3.0$) with permute.**

(Sec. IV-E2). The label "AVX-512" switches AVX2 to AVX-512 so that the register size is from 8 to 32 (Sec. IV-A), and the label "bfloat-16" is 64 (Sec. IV-C). From the simplest to the entire proposed method case, PSNR is improving from 40.41 dB to 65.52 (+25.11) dB in AVX2. Using AVX-512, PSNR is 78.63 (+38.22) dB and 84.5 (+44.09) dB with bloat-16.

Figs. 11 to 14 evaluate the downsampling methods for the register-LUTs using various lookup methods. The Gauss integral is superior to the nearest neighbor method in all cases. When the number of LUT elements is large, there is little difference between the methods (see the permute-bf 192 case). The last-integral is the best when the number of elements is small.

B. FILTERING PARAMETER DEPENDENCY

Fig. 15 shows the parameter dependency of σ_r and σ_s for each register-LUT method in color images. The LUT quantization method is Gauss integral with last-mean. We omit the AVX-512 shuffle because the response is the same as its AVX2 case.

The higher the number of register elements, the higher the approximation accuracy. In addition, PSNR tends to be higher when σ_r is large, but no parameter causes an extreme drop in PSNR. Even in the case of permute-8, which has the lowest number of LUT elements, the PSNR never falls below 60 dB.

Figure 16 shows PSNR for σ_r with various register-LUT methods that are profile plots in Fig. 15 on $\sigma_s = 3.0$. The performance of each method improves with the number of LUTs processed. The number of LUTs for shuffle-16 and permute-16 are the same, so they are almost on the same plot, but shuffle-16 is slightly worse due to quantization. In addition, the pairs (permute-32, shuffle-32) and (permute-64, permute-bf-64) have the same tendency. The PSNR is higher for the wide σ than for the narrow σ, but the difference is insignificant.

Figure 16 shows PSNR for σ_s with various register-LUT methods that are profile plots in Fig. 15 on $\sigma_r = 30.0$. The trend is almost the same as in the σ_r case, but the lack of LUT quantization makes it less jagged.

C. ARCHITECTURE DEPENDENCY

This section shows the trade-off between PSNR and time for various methods for each CPU architecture. Figs 17, 18, 19, 20, and 21 show AVX2 cases and Figs 22, 23, 24, 25, and 26 show AVX-512 cases. We evaluate four competitive methods with the proposed methods: direct weight computing with SVML(exp) and with fmath library (fmath), LUT vector addressing (gather), and scalar addressing (set).

In AVX2 implementation on x86/64 CPUs, register-LUT implementations of the permute and shuffle are faster than all competitive methods. Among the register-LUT methods, the shuffle approach has better trade-off performance than the permute approach in all cases. However, permute is faster and exceeds the display limit of 60 dB for 8-bit displays; thus, it depends on the case in which one is used. In Intel CPU, the gather is faster than the set, and computing (exp and fmath) is relatively slower than the gather. Blending three permutes or shuffles is slower than the gather in Intel CPUs. In AMD CPU, the gather is slower than the set in Zen+, Zen2, and Zen3, but Zen3 improves the gather performance. The fmath implementation also uses gather intrinsic; thus, the performance is slow. In AMD CPUs register-LUT methods are relatively faster than the competitive methods because the shuffle and permute are higher CPI and computing and vector addressing is slower CPI than Intel CPU; thus, three merges are effective.

In AVX-512 implementation on x86/64 CPUs in the register-LUT methods, conversely, the permute has better trade-off performance than the shuffle in all cases because AVX-512 permute can handle substantially longer vector lengths than shuffle, and IPC is superior. In the case of AVX-512, a register-LUT technique with bfloat16 is added, and a single-stage bfloat16 instruction has the same number of registers as a stacked two-stage permute instruction. In all cases except for the Rocket Lake architecture, the bfloat16 implementation is slightly better than the float implementation indicating that the choice of implementation should be based on the architecture. The computation and vector addressing implementations vary by architecture, but the register-LUT method is superior in all cases.

Tab. 3 and 4 show normalized time by the fastest permute method for each architecture on AVX2 and AVX-512. Even the fastest permute exceeds 60 dB, so the difference is not noticeable on an 8-bit display. In AVX2, gather is the second fastest on Intel CPUs and set is the second fastest on AMD CPUs. In AVX-512, gather is the second fastest on Intel CPUs and exp is the second fastest on AMD CPUs. The fastest proposed method was on average 4.82/3.72 times faster than direct vector computing, 2.99/3.10 times faster than vector

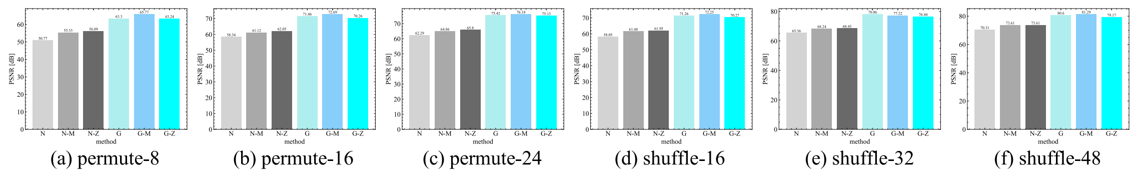


FIGURE 11. PSNR for each LUT quantization method in AVX2 (gray).

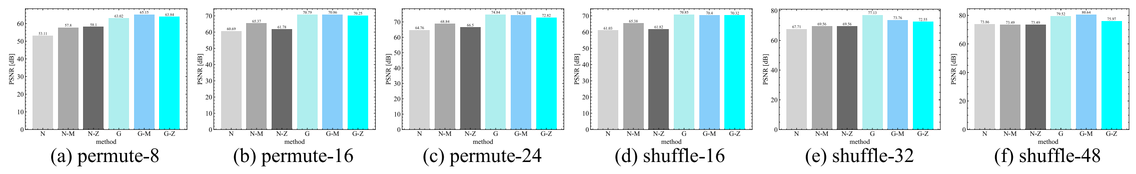


FIGURE 12. PSNR for each LUT quantization method in AVX2 (color).

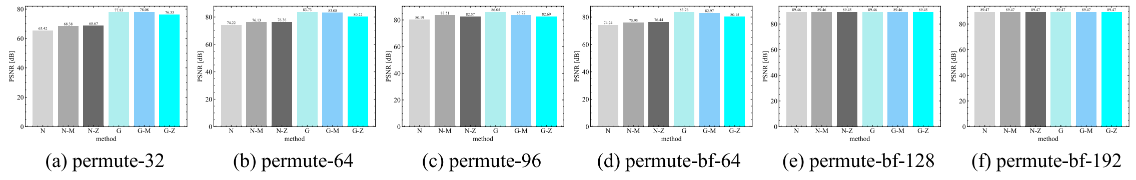


FIGURE 13. PSNR for each LUT quantization method in AVX-512 (gray).

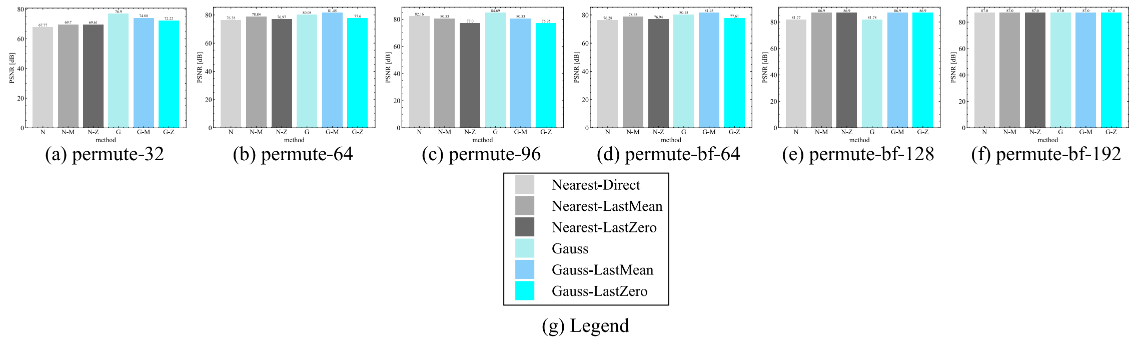


FIGURE 14. PSNR for each LUT quantization method in AVX-512 (color).

addressing, and 3.79/7.80 times faster than scalar addressing on the AVX2/AVX-512 computers.

VI. RELATED WORK

This paper focuses on accelerating HDKF convolution on x86/64 CPUs using SIMD and LUT. In this section, we review mathematical functions for alternative LUT computing, acceleration of convolutions, and acceleration of HDKFs.

A. MATHEMATICAL FUNCTIONS

Knowing how to speed up numerical computing is important, not refering to LUTs. This paper uses an exponential function for numerical computing.

The libm library is the standard mathematical library used in C, which includes an exponential function. The algorithms described by Gal [36] and Crlibm [37] achieved very accurate results by reducing rounding errors. The GNU C Library (glibc)[5] is the most widely used implementation of libm and includes Freely Distributable LIBM (FDLIBM).[6] OpenLibm[7] is an effort to have a high quality, portable, standalone libm derived from FDLIBM. Intel oneAPI Math Kernel Library (oneMKL)[8] is a math library for numerical computing on Intel's CPUs and GPUs, and Intel Short Vector Math Library (SVML) is one of them. AOCL-LibM[9] is a set of numerical libraries optimized for AMD processors and is part of AMD Optimizing CPU Libraries (AOCL), the

[5]https://www.gnu.org/software/libc/manual/
[6]https://www.netlib.org/fdlibm/
[7]https://openlibm.org/
[8]https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[9]https://github.com/amd/aocl-libm-ose

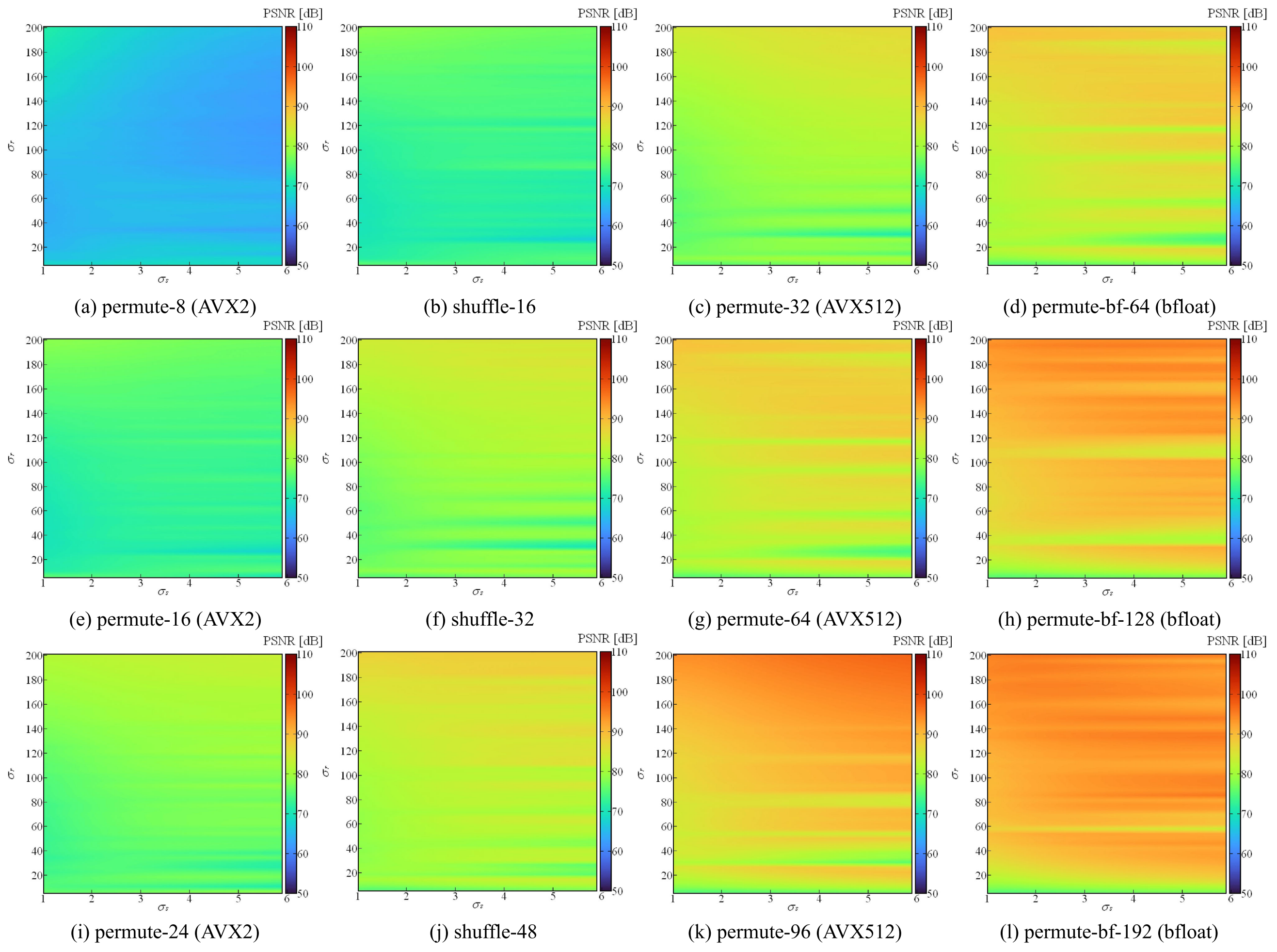


FIGURE 15. Approximation accuracy for each parameter σ_s and σ_r.

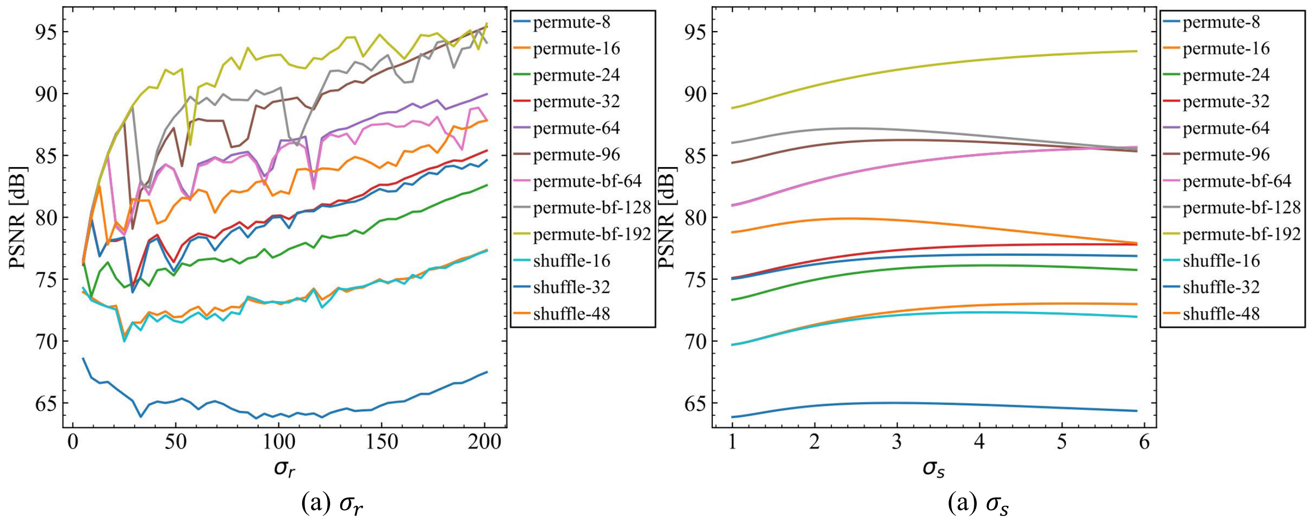
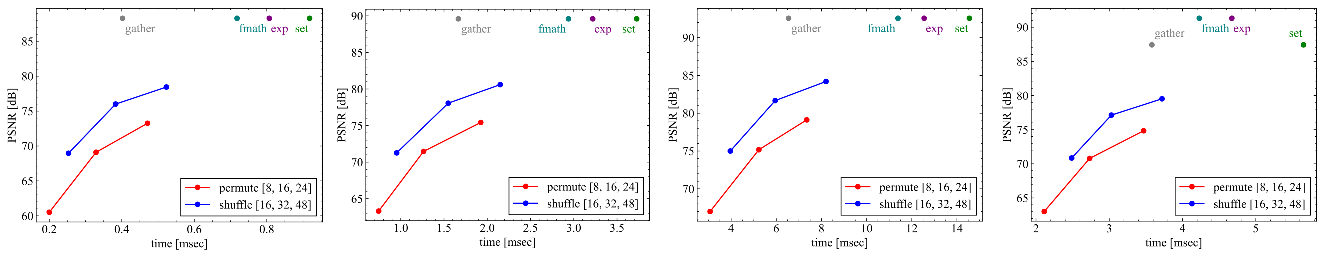


FIGURE 16. σ_r and σ_s to PSNR for various register-LUT methods. One of the parameters is fixed at the following value: $\sigma_s = 3.0$ and $\sigma_r = 30.0$.

successor to AMD Core Math Library (ACML). Libmvec,[10] Yeppp! [38] and Vector-libm [39] are the other vectorized implementations of libm. SLEEF [40][11] is a vectorized libm that achieved excellent performance and portability. Agner

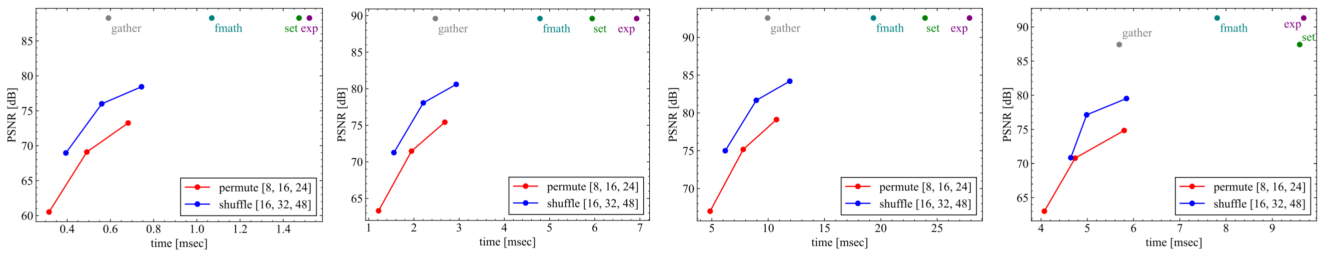
[10]https://sourceware.org/glibc/wiki/libmvec

[11]https://sleef.org/



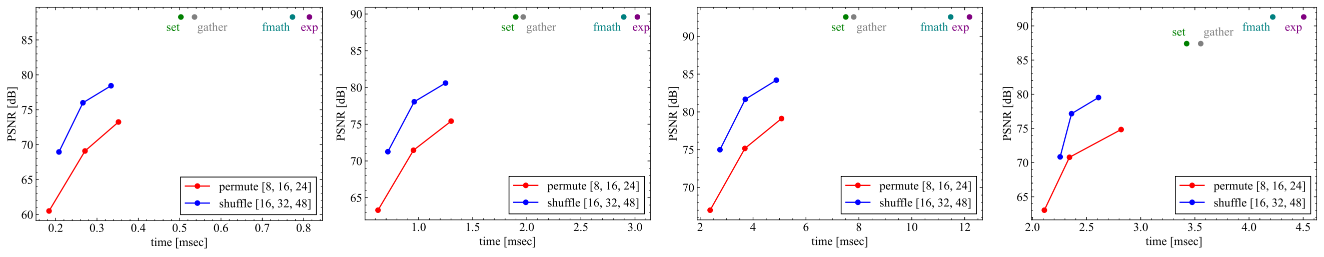
(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 17. **Intel alder lake (AVX2).**



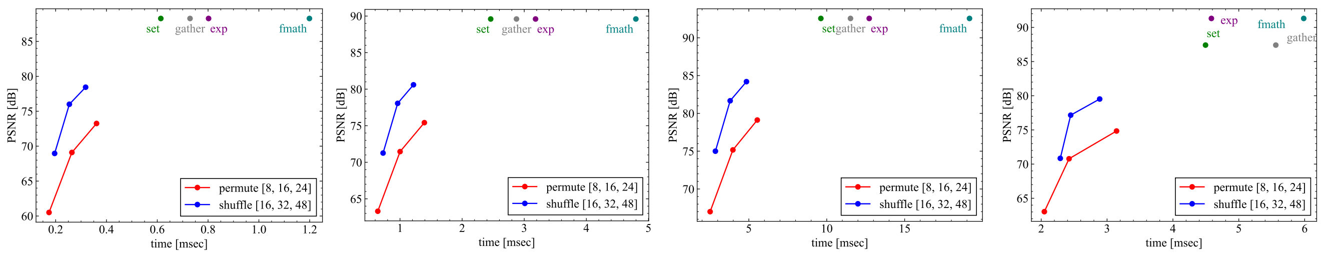
(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 18. **Intel coffee lake refresh (AVX2).**



(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 19. **AMD Zen3 (AVX2).**



(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

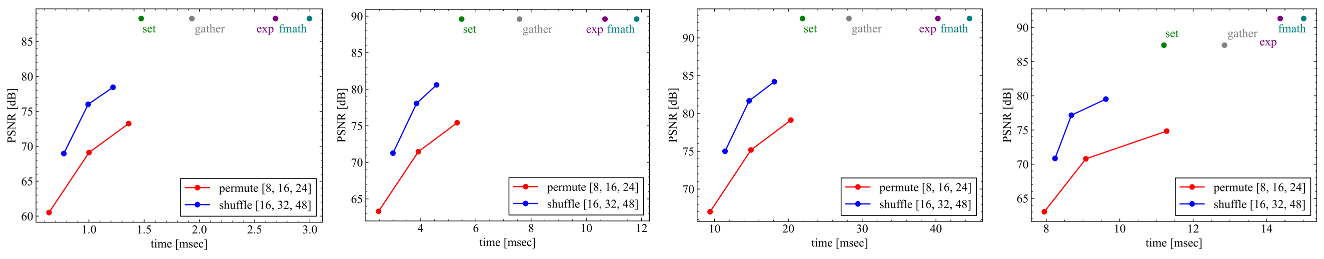
FIGURE 20. **AMD Zen2 (AVX2).**

Fog's Vector Class Library[12] is a C++ class library for using SIMD instructions to improve performance on modern microprocessors with the x86 or x86/64 instruction set.

There are various optimizations for specific functions. Yamamoto et al. tested various approximations of the exponential function from the viewpoint of computational efficiency [41]. The fmath library[13] utilizes the computation of exponential functions with vector addressing to achieve higher speed. de Lassus Saint-Geniés et al. reported accurate LUTs for trigonometric and hyperbolic functions [42]. Shen et al. proposed effective vectorizations of trigonometric functions [43].

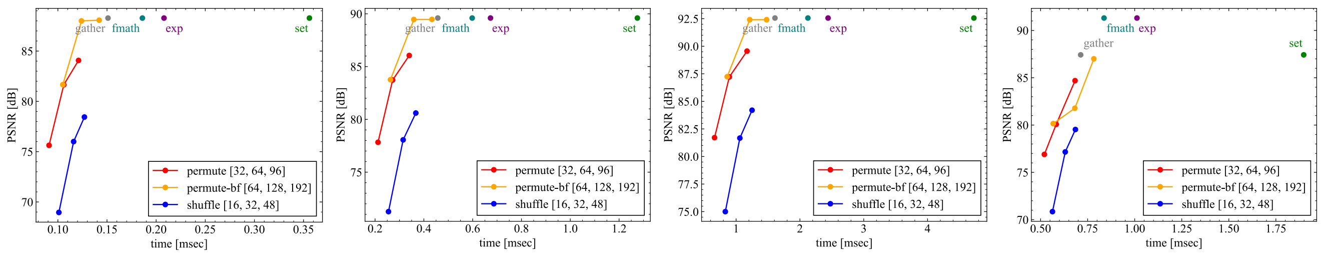
[12]https://github.com/vectorclass/version2

[13]https://github.com/herumi/fmath



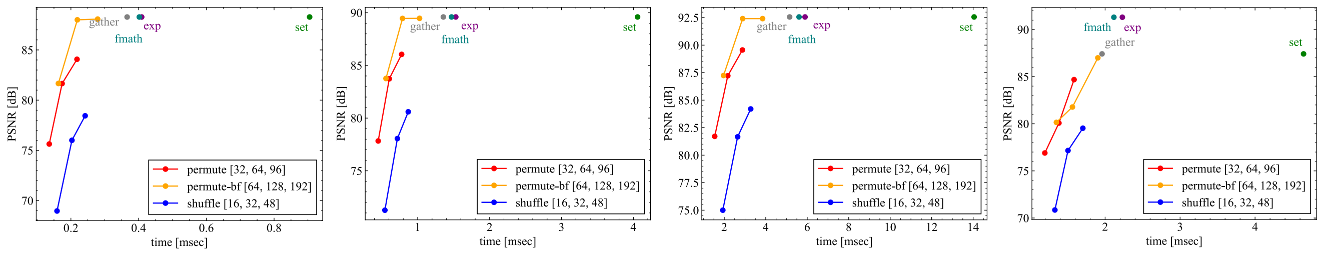
(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 21. AMD Zen+ (AVX2).



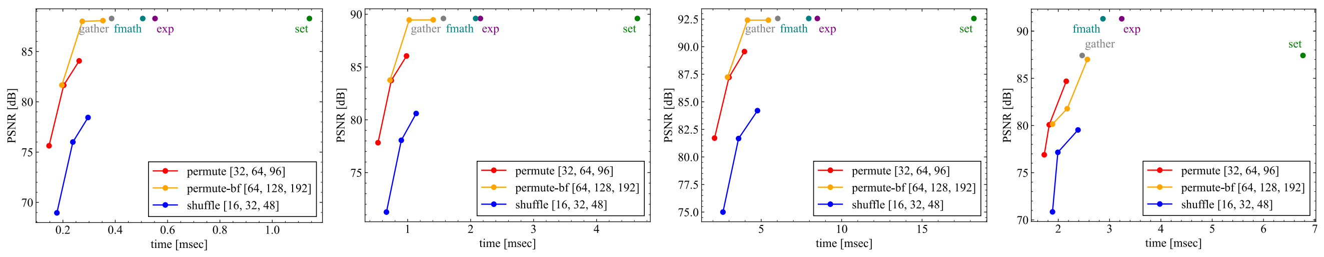
(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 22. Intel sapphire rapids (AVX-512).



(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 23. Intel cascade lake (AVX-512).

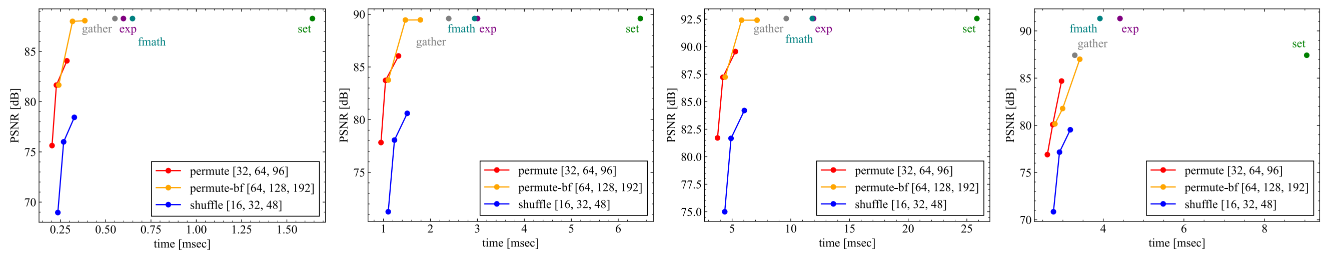


(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 24. Intel alder lake (AVX-512).

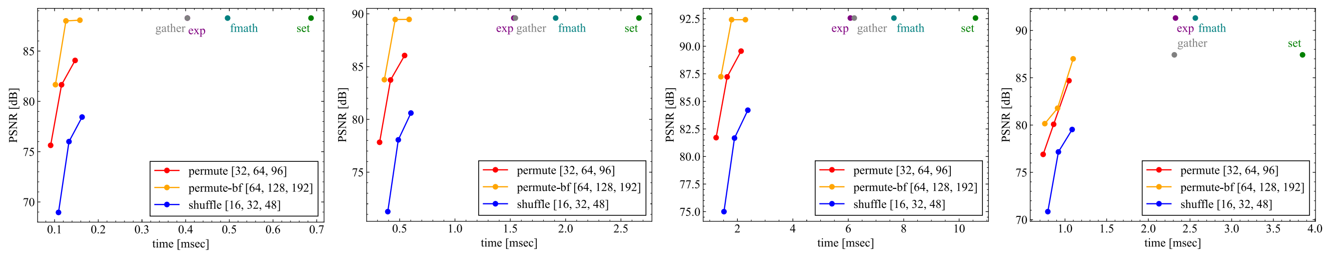
Currently, compiler-based math functions are utilized. Zhang et al. proposed a special-purpose compiler that is capable of generating LUTs that use Taylor series interpolants based on accuracy and memory constraints [44]. Anand and Kahl [45] proposed a domain-specific language (DSL) for libm functions tuned for Cell/B.E. SPU compute engine. VDT Mathematical Library [46] leverages the capability of compilers to emit machine instructions optimized for the target architecture. FunC [4][14](for Function Comparetor) evaluates the performance of direct evaluation relative to various implementation LUTs. MegaLibm [47] is a DSL for implementing, testing, and tuning math library implementations.

[14]https://github.com/uofs-simlab/func



(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 25. Intel rocket lake (AVX-512).



(a) gray 256 × 256 (b) gray 512 × 512 (c) gray 1024 × 1024 (d) color 512 × 512

FIGURE 26. AMD Zen4 (AVX-512).

TABLE 3. Time ratio of each method to permute-8 (63.6 dB) on AVX2 with 512 × 512 grayscale.

	Ald.	Cof.	Zen3	Zen2	Zen+	ave.
exp	4.32	5.69	4.84	4.96	4.31	4.82
fmath	3.94	3.93	4.63	7.47	4.77	4.95
gather	**2.24**	**2.03**	3.15	4.48	3.06	2.99
set	5.00	4.88	**3.04**	**3.83**	**2.22**	3.79
permute-8	1.00	1.00	1.00	1.00	1.00	1.00
permute-16	1.70	1.60	1.52	1.56	1.58	1.59
permute-24	2.58	2.20	2.08	2.17	2.15	2.24
shuffle-16	1.28	1.28	1.15	1.13	1.21	1.21
shuffle-32	2.08	1.81	1.54	1.50	1.56	1.70
shuffle-48	2.89	2.41	2.00	1.90	1.85	2.21

TABLE 4. Time ratio of each method to permute-32 (77.83 dB) on AVX-512 with 512 × 512 grayscale.

	Sap.	Cas.	Ald.	Roc.	Zen4	ave.
exp	3.17	3.39	4.07	3.17	**4.80**	3.72
fmath	2.82	3.26	3.93	3.11	5.99	3.82
gather	**2.16**	**3.01**	**2.96**	**2.52**	4.85	3.10
set	6.01	9.00	8.79	6.83	8.35	7.80
permute-32	1.00	1.00	1.00	1.00	1.00	1.00
permute-64	1.28	1.34	1.40	1.10	1.31	1.29
permute-96	1.60	1.72	1.86	1.39	1.71	1.66
permute-bf-64	1.24	1.24	1.36	1.16	1.13	1.23
permute-bf-128	1.69	1.75	1.94	1.55	1.44	1.67
permute-bf-192	2.04	2.27	2.66	1.89	1.84	2.14
shuffle-16	1.20	1.21	1.25	1.16	1.23	1.21
shuffle-32	1.49	1.59	1.70	1.30	1.53	1.52
shuffle-48	1.73	1.93	2.15	1.59	1.89	1.86

B. OPTIMIZATION FOR CONVOLUTION

FIR filtering with spatial invariant convolution was deployed in various architectures, such as MMX [48], SSE [49], PowerPC and Cell/B.E. [50], AVX [2], AVX-512 with loop unrolling [51], ARM [52]. WebAssembly [53], GPU [54], [55], [56], and integrated CPU-GPU, and FPGA [57]. In addition, integer convolution was proposed on x86/64 CPUs [58], [59], [60], [61], [62]. Vectorization of image processing pipelines that include spatial invariant convolution had also been proposed, such as Harris corner detection [63], which uses Gaussian filtering for structure tensor images, [64], [65], edge detection with Sobel filtering [66], morphological filter on ARM CPU [67], and wavelet transforms [68], [69].

Variable-weighted convolution of edge-preserving filtering and adaptive weighted filtering using LUT was vectorized on x86/64 CPUs [2], [3], [70] and GPUs [71]. This paper is one of this type.

Halide [72] is a DSL for image processing, and the language can easily vectorize codes with a simple description. There are various effective implementations in Halide, such as interpolation [73], FIR [74], recursive [75], median [76], and variable-weighted [77] convolutions.

For CNN accelerations, we can use four algorithms: direct, lowering, FFT, and Winograd. Direct algorithms are implemented as six nested loops with a multiply-add instruction, and various CPU optimization approaches are proposed [78], [79], [80], [81], [82]. The lowering (im2col) approach [1] transforms image structure followed by general matrix-matrix multiplications [83], [84], [85], [86]. FFT accelerates convolution [87], [88] and Winograd-based convolution [89] to decrease arithmetic operations [87], [90], [91], [92].

C. APPROXIMATED HDKF FOR ACCELERATION

Durand and Dorsey [21] proposed early work to accelerate grayscale bilateral filtering (BF). This approach decomposed

BF into multiple Gaussian filters (GFs) with FFT accelerations. Paris and Durand [93] extended Durand's work by representing BF as HDKF with downsampling acceleration. Subsequent studies have proposed higher-performance approximations by refining the range kernel representation [94], [95], [96], [97], [98], [99], [100], [101]. Recently, constant-time Gaussian filter approximations have been used to speed up the process [102], [103].

For the color case, Paris and Durand [28] and Yang et al. [104] proposed an extension of the gray BF, and the method was $O(K^3)$. Recent approaches reduced the number of range kernel convolutions by random subsampling [105], [106], [107]. Adams et al. proposed an HDKF data structure for efficient processing, the Gaussian KD-tree [5] and permutohedral lattice [6]. Additionally, clustering [108] can acclerate HDKFs [7], [8], [109], [110], [111].

These methods are independent of kernel size and work better when the convolution radius is large. In contrast, this paper approximates HDFK of the range kernel in the naïve implementation.

VII. CONCLUSION

In this paper, we propose a method to perform LUT reference by vectorized computation using swizzle instructions for high-dimensional kernel filtering, a variable-weighted convolution. Experimental results show that the proposed method can reproduce a PSNR of 65.52 (+25.11) dB, while a simple full-size LUT in the register size can only reproduce a PSNR of 40.41 dB. Using a wider register width, the PSNR was 78.63 (+38.22) dB; using the bfloat16 type, it could be improved to 84.5 (+44.09) dB. Speed was also tested on various architectures. The fastest proposed method was on average 4.82/3.72 times faster than direct vector computing, 2.99/3.10 times faster than vector addressing, and 3.79/7.80 times faster than scalar addressing on the AVX2/AVX-512 computers while exceeding the display limit of 60 dB for 8-bit displays. Considering these speed/accuracy trade-offs, the performance of the proposed method was superior. Moreover, using various LUT generation methods, it was possible to operate with an approximation accuracy of 60 dB or better (the limit of an 8-bit display).

REFERENCES

[1] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Proc. Int. Workshop Frontiers Handwriting Recognit.*, 2006, pp. 1–7. [Online]. Available: https://inria.hal.science/inria-00112631

[2] Y. Maeda, N. Fukushima, and H. Matsuo, "Taxonomy of vectorization patterns of programming for FIR image filters using kernel subsampling and new one," *Appl. Sci.*, vol. 8, no. 8, p. 1235, Jul. 2018, doi: 10.3390/app8081235.

[3] Y. Maeda, N. Fukushima, and H. Matsuo, "Effective implementation of edge-preserving filtering on CPU microarchitectures," *Appl. Sci.*, vol. 8, no. 10, p. 1985, Oct. 2018, doi: 10.3390/app8101985.

[4] K. R. Green, T. A. Bohn, and R. J. Spiteri, "Direct function evaluation versus lookup tables: When to use which?" *SIAM J. Sci. Comput.*, vol. 41, no. 3, pp. C194–C218, Jan. 2019, doi: 10.1137/18m1201421.

[5] A. Adams, N. Gelfand, J. Dolson, and M. Levoy, "Gaussian KD-trees for fast high-dimensional filtering," *ACM Trans. Graph.*, vol. 28, no. 3, pp. 1–12, Jul. 2009, doi: 10.1145/1531326.1531327.

[6] A. Adams, J. Baek, and M. A. Davis, "Fast high-dimensional filtering using the permutohedral lattice," *Comput. Graph. Forum*, vol. 29, no. 2, pp. 753–762, May 2010, doi: 10.1111/j.1467-8659.2009.01645.x.

[7] P. Nair and K. N. Chaudhury, "Fast high-dimensional kernel filtering," *IEEE Signal Process. Lett.*, vol. 26, no. 2, pp. 377–381, Feb. 2019, doi: 10.1109/LSP.2019.2891879.

[8] S. Oishi and N. Fukushima, "Tiling and PCA strategy for clustering-based high-dimensional Gaussian filtering," *Social Netw. Comput. Sci.*, vol. 5, no. 1, p. 40, Nov. 2023, doi: 10.1007/s42979-023-02319-6.

[9] P. Milanfar, "A tour of modern image filtering: New insights and methods, both practical and theoretical," *IEEE Signal Process. Mag.*, vol. 30, no. 1, pp. 106–128, Jan. 2013, doi: 10.1109/MSP.2011.2179329.

[10] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Proc. 6th Int. Conf. Comput. Vis.*, Jan. 1998, pp. 839–846, doi: 10.1109/ICCV.1998.710815.

[11] G. Petschnigg, R. Szeliski, M. Agrawala, M. Cohen, H. Hoppe, and K. Toyama, "Digital photography with flash and no-flash image pairs," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 664–672, Aug. 2004, doi: 10.1145/1015706.1015777.

[12] E. Eisemann and F. Durand, "Flash photography enhancement via intrinsic relighting," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 673–678, Aug. 2004, doi: 10.1145/1015706.1015778.

[13] E. P. Bennett, J. L. Mason, and L. Mcmillan, "Multispectral bilateral video fusion," *IEEE Trans. Image Process.*, vol. 16, no. 5, pp. 1185–1194, May 2007, doi: 10.1109/TIP.2007.894236.

[14] T. Matsuo, N. Fukushima, and Y. Ishibashi, "Weighted joint bilateral filter with slope depth compensation filter for depth map refinement," in *Proc. Int. Conf. Comput. Vis. Theory Appl.*, vol. 2, 2013, pp. 300–309, doi: 10.5220/0004292203000309.

[15] J. Kopf, M. F. Cohen, D. Lischinski, and M. Uyttendaele, "Joint bilateral upsampling," *ACM Trans. Graph.*, vol. 26, no. 3, p. 96, Jul. 2007, doi: 10.1145/1276377.1276497.

[16] A. Buades, B. Coll, and J.-M. Morel, "A non-local algorithm for image denoising," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2005, pp. 60–65, doi: 10.1109/CVPR.2005.38.

[17] M. Zhang and B. K. Gunturk, "Multiresolution bilateral filtering for image denoising," *IEEE Trans. Image Process.*, vol. 17, no. 12, pp. 2324–2333, Dec. 2008, doi: 10.1109/TIP.2008.2006658.

[18] S. Dai, M. Han, Y. Wu, and Y. Gong, "Bilateral back-projection for single image super resolution," in *Proc. IEEE Multimedia Expo. Int. Conf.*, Jul. 2007, pp. 1039–1042, doi: 10.1109/ICME.2007.4284831.

[19] K. Hayashi, Y. Maeda, and N. Fukushima, "Local contrast enhancement with multiscale filtering," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Oct. 2023, pp. 765–770, doi: 10.1109/apsipaasc58517.2023.10317242.

[20] Y. Sumiya, T. Otsuka, Y. Maeda, and N. Fukushima, "Gaussian Fourier pyramid for local Laplacian filter," *IEEE Signal Process. Lett.*, vol. 29, pp. 11–15, 2022, doi: 10.1109/LSP.2021.3121198.

[21] F. Durand and J. Dorsey, "Fast bilateral filtering for the display of high-dynamic-range images," *ACM Trans. Graph.*, vol. 21, no. 3, pp. 257–266, Jul. 2002, doi: 10.1145/566654.566574.

[22] V. Ramakrishnan and D. J. Pete, "Savitzky–Golay filtering-based fusion of multiple exposure images for high dynamic range imaging," *Social Netw. Comput. Sci.*, vol. 2, no. 3, p. 191, May 2021, doi: 10.1007/s42979-021-00594-9.

[23] N. Fukushima, K. Sugimoto, and S.-I. Kamata, "Guided image filtering with arbitrary window function," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Apr. 2018, pp. 1523–1527, doi: 10.1109/ICASSP.2018.8462016.

[24] S. Oishi and N. Fukushima, "Retinex-based relighting for night photography," *Appl. Sci.*, vol. 13, no. 3, p. 1719, Jan. 2023, doi: 10.3390/app13031719.

[25] E. S. L. Gastal and M. M. Oliveira, "Shared sampling for real-time alpha matting," *Comput. Graph. Forum*, vol. 29, no. 2, pp. 575–584, May 2010, doi: 10.1111/j.1467-8659.2009.01627.x.

[26] T. Matsuo, S. Fujita, N. Fukushima, and Y. Ishibashi, "Efficient edge-awareness propagation via single-map filtering for edge-preserving stereo matching," *Proc. SPIE*, vol. 9393, Mar. 2015, Art. no. 93930S, doi: 10.1117/12.2083087.

[27] S. Fujita, T. Matsuo, N. Fukushima, and Y. Ishibashi, "Cost volume refinement filter for post filtering of visual corresponding," *Proc. SPIE*, vol. 9399, Mar. 2015, Art. no. 93990Q, doi: 10.1117/12.2083086.

[28] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *Int. J. Comput. Vis.*, vol. 81, no. 1, pp. 24–52, Jan. 2009, doi: 10.1007/s11263-007-0110-8.

[29] D. Miralles and D. M. Akos, "A SIMD intrinsic correlator library for GNSS software receivers," *GPS Solutions*, vol. 23, no. 3, p. 72, Jul. 2019, doi: 10.1007/s10291-019-0865-8.

[30] J. Cooper, S. McKeever, and A. Garny, "On the application of partial evaluation to the optimisation of cardiac electrophysiological simulations," in *Proc. ACM SIGPLAN Symp. Partial Eval. Semantics-Based Program Manipulation*, Jan. 2006, pp. 12–20, doi: 10.1145/1111542.1111546.

[31] C. B. Marsh, K. R. Green, B. Wang, and R. J. Spiteri, "Performance improvements to modern hydrological models via lookup table optimizations," *Environ. Model. Softw.*, vol. 139, May 2021, Art. no. 105018, doi: 10.1016/j.envsoft.2021.105018.

[32] R. Cypher and J. L. C. Sanz, "SIMD architectures and algorithms for image processing and computer vision," *IEEE Trans. Acoust., Speech, Signal Process.*, vol. 37, no. 12, pp. 2158–2174, 1989, doi: 10.1109/29.45558.

[33] T. Tsubokawa, H. Tajima, Y. Maeda, and N. Fukushima, "Local look-up table upsampling for accelerating image processing," *Multimedia Tools Appl.*, Aug. 2023, doi: 10.1007/s11042-023-16405-7.

[34] A. Abel and J. Reineke, "Uops.Info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2019, pp. 673–686, doi: 10.1145/3297858.3304062.

[35] J. M. Cebrian, L. Natvig, and M. Jahre, "Scalability analysis of AVX-512 extensions," *J. Supercomput.*, vol. 76, no. 3, pp. 2082–2097, Mar. 2020, doi: 10.1007/s11227-019-02840-7.

[36] S. Gal, "An accurate elementary mathematical library for the IEEE floating point standard," *ACM Trans. Math. Softw.*, vol. 17, no. 1, pp. 26–45, Mar. 1991, doi: 10.1145/103147.103151.

[37] C. Daramy, D. Defour, F. Dinechin, and J.-M. Müller, "CR-LIBM: A correctly rounded elementary function library," *Proc. SPIE*, vol. 5205, pp. 458–464, Dec. 2003, doi: 10.1117/12.505591.

[38] M. Dukhan and R. Vuduc, "Methods for high-throughput computation of elementary functions," in *Proc. Parallel Process. Appl. Math.*, 2014, pp. 86–95, doi: 10.1007/978-3-642-55224-3_9.

[39] C. Lauter, "A new open-source SIMD vector libm fully implemented with high-level scalar c," in *Proc. 50th Asilomar Conf. Signals, Syst. Comput.*, Nov. 2016, pp. 407–411, doi: 10.1109/ACSSC.2016.7869070.

[40] N. Shibata and F. Petrogalli, "SLEEF: A portable vectorized library of c standard mathematical functions," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 6, pp. 1316–1327, Jun. 2020, doi: 10.1109/TPDS.2019.2960333.

[41] A. Yamamoto, Y. Kitamura, and Y. Yamane, "Computational efficiencies of approximated exponential functions for transport calculations of the characteristics method," *Ann. Nucl. Energy*, vol. 31, no. 9, pp. 1027–1037, Jun. 2004, doi: 10.1016/j.anucene.2004.01.003.

[42] H. de Lassus Saint-Geniès, D. Defour, and G. Revy, "Exact lookup tables for the evaluation of trigonometric and hyperbolic functions," *IEEE Trans. Comput.*, vol. 66, no. 12, pp. 2058–2071, Dec. 2017, doi: 10.1109/TC.2017.2703870.

[43] J. Shen, B. Long, and C. Huang, "Optimizing fast trigonometric functions on modern CPUs," in *Proc. IEEE 24th Int. Conf. High Perform. Comput. Commun.; 8th Int. Conf. Data Sci. Syst.; 20th Int. Conf. Smart City; 8th Int. Conf. Dependability Sensor, Cloud Big Data Syst. Appl. (HPCC/DSS/SmartCity/DependSys)*, Dec. 2022, pp. 1022–1029, doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00162.

[44] Y. Zhang, L. Deng, P. Yedlapalli, S. P. Muralidhara, H. Zhao, M. Kandemir, C. Chakrabarti, N. Pitsianis, and X. Sun, "A special-purpose compiler for look-up table and code generation for function evaluation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2010, pp. 1130–1135, doi: 10.1109/DATE.2010.5456978.

[45] C. K. Anand and W. Kahl, "An optimized cell BE special function library generated by coconut," *IEEE Trans. Comput.*, vol. 58, no. 8, pp. 1126–1138, Aug. 2009, doi: 10.1109/TC.2008.223.

[46] D. Piparo, V. Innocente, and T. Hauth, "Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions," *J. Phys., Conf. Ser.*, vol. 513, no. 5, Jun. 2014, Art. no. 052027, doi: 10.1088/1742-6596/513/5/052027.

[47] I. Briggs, Y. Lad, and P. Panchekha, "Implementation and synthesis of math library functions," in *Proc. ACM Symp. Princ. Program. Lang. (POPL)*, 2024, pp. 942–969.

[48] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, "Efficient vectorization of the fir filter," in *Proc. Annu. Workshop Circuits, Syst. Signal Process. (ProRISC)*, 2005, pp. 432–437.

[49] J. G. A. Barbedo and A. Lopes, "On the vectorization of FIR filterbanks," *EURASIP J. Adv. Signal Process.*, vol. 2007, no. 1, pp. 1–10, Dec. 2006, doi: 10.1155/2007/91741.

[50] D. Nuzman and A. Zaks, "Outer-loop vectorization–revisited for short SIMD architectures," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, Oct. 2008, pp. 2–11, doi: 10.1145/1454115.1454119.

[51] I. Masamae and P. Chaikan, "High performance 2D convolution utilizing the AVX512 on a multi-core architecture," *Songklanakarin J. Sci. Technol.*, vol. 43, no. 4, pp. 1230–1236, 2021, doi: 10.14456/sjst-psu.2021.160.

[52] A. Shevchenko, P. Prystavka, and V. Tymchyshyn, "Research on possible convolution operation speed enhancement via AArch64 SIMD," in *Proc. Int. Conf. Comput. Sci., Eng. Educ. Appl.*, 2022, pp. 61–75, doi: 10.1007/978-3-031-04812-8_6.

[53] S. Oishi, K. Ishikawa, H. Nogami, and N. Fukushima, "Performance evaluation of image convolution with WebAssembly," in *Proc. Int. Workshop Adv. Imag. Technol. (IWAIT)*, Mar. 2023, Art. no. 125922, doi: 10.1117/12.2667004.

[54] B. van Werkhoven, J. Maassen, H. E. Bal, and F. J. Seinstra, "Optimizing convolution operations on GPUs using adaptive tiling," *Future Gener. Comput. Syst.*, vol. 30, pp. 14–26, Jan. 2014.

[55] K. Preethi and K. S. Vishvaksenan, "Gaussian filtering implementation and performance analysis on GPU," in *Proc. Int. Conf. Inventive Res. Comput. Appl. (ICIRCA)*, Jul. 2018, pp. 936–939, doi: 10.1109/ICIRCA.2018.8597299.

[56] Y. Zhou, F. He, and Y. Qiu, "Accelerating image convolution filtering algorithms on integrated CPU–GPU architectures," *J. Electron. Imag.*, vol. 27, no. 3, May 2018, Art. no. 033002, doi: 10.1117/1.jei.27.3.033002.

[57] L. Rao, B. Zhang, and J. Zhao, "Hardware implementation of reconfigurable 1D convolution," *J. Signal Process. Syst.*, vol. 82, no. 1, pp. 1–16, Jan. 2016, doi: 10.1007/s11265-015-0969-5.

[58] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, Jun. 2013, pp. 24–35, doi: 10.1145/2485922.2485925.

[59] H. Amiri and A. Shahbahrami, "High performance implementation of 2-D convolution using AVX2," in *Proc. 19th Int. Symp. Comput. Archit. Digit. Syst. (CADS)*, Dec. 2017, pp. 1–4, doi: 10.1109/CADS.2017.8310675.

[60] A. Frickenstein, M. R. Vemparala, C. Unger, F. Ayar, and W. Stechele, "DSC: Dense-sparse convolution for vectorized inference of convolutional neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops (CVPRW)*, Jun. 2019, pp. 1353–1360, doi: 10.1109/CVPRW.2019.00175.

[61] M. Moradifar and A. Shahbahrami, "Performance improvement of Gaussian filter using SIMD technology," in *Proc. Int. Conf. Mach. Vis. Image Process. (MVIP)*, Feb. 2020, pp. 1–6, doi: 10.1109/MVIP49855.2020.9116883.

[62] V. Kelefouras and G. Keramidas, "Design and implementation of 2D convolution on x86/x64 processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3800–3815, Dec. 2022, doi: 10.1109/TPDS.2022.3171471.

[63] C. Harris and M. Stephens, "A combined corner and edge detector," in *Proc. Alvey Vis. Conf.*, 1988, pp. 147–151.

[64] L. Lacassagne, D. Etiemble, A. H. Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for SIMD and low-level computer vision algorithms," in *Proc. Workshop Program. Models SIMD/Vector Process.*, Feb. 2014, pp. 49–56, doi: 10.1145/2568058.2568067.

[65] O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, and B. Ouni, "Harris corner detection on a numa manycore," *Future Gener. Comput. Syst.*, vol. 88, pp. 442–452, 2018, doi: 10.1016/j.future.2018.01.048.

[66] A. S. Zekri, "Optimizing image spatial filtering on single CPU core," *Multimedia Tools Appl.*, vol. 77, no. 1, pp. 251–281, Jan. 2018, doi: 10.1007/s11042-016-4266-5.

[67] E. Limonova, A. Terekhin, D. Nikolaev, and V. Arlazarov, "Fast implementation of morphological filtering using ARM NEON extension," 2020, *arXiv:2002.09474*.

[68] Y. Sumiya, H. Kamei, K. Ishikawa, and N. Fukushima, "Vectorized computing for edge-avoiding wavelet," in *Proc. Int. Workshop Adv. Imag. Technol. (IWAIT)*, May 2022, pp. 23–28, doi: 10.1117/12.2626109.

[69] A. Shahbahrami, B. Juurlink, and S. Vassiliadis, ''Implementing the 2-D wavelet transform on SIMD-enhanced general-purpose processors,'' *IEEE Trans. Multimedia*, vol. 10, no. 1, pp. 43–51, Jan. 2008, doi: 10.1109/TMM.2007.911195.

[70] N. Fukushima, T. Tsubokawa, and Y. Maeda, ''Vector addressing for non-sequential sampling in fir image filtering,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2019, pp. 4185–4189, doi: 10.1109/ICIP.2019.8803565.

[71] T. Kondo, Y. Maeda, and N. Fukushima, ''Accelerating finite impulse response filtering using tensor cores,'' in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Dec. 2021, pp. 74–79. [Online]. Available: https://ieeexplore.ieee.org/document/9689358

[72] 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,'' in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Design Implement.* New York, NY, USA: Association for Computing Machinery, Jun. 2013, pp. 519–530, doi: 10.1145/2491956.2462176.

[73] H. Nogami, S. Oishi, T. Sasaki, Y. Maeda, and N. Fukushima, ''Performance evaluation of halide auto-scheduler with directional cubic convolution interpolation,'' in *Proc. Int. Workshop Adv. Imag. Technol. (IWAIT)*, Mar. 2023, Art. no. 125922, doi: 10.1117/12.2666979.

[74] H. Takagi and N. Fukushima, ''Domain specific description in halide for randomized image convolution,'' in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, Dec. 2021, pp. 63–69. [Online]. Available: https://ieeexplore.ieee.org/document/9689317

[75] H. Takagi and N. Fukushima, ''An efficient description with halide for iir Gaussian filter,'' in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, 2020, pp. 28–35. [Online]. Available: https://ieeexplore.ieee.org/document/9306460

[76] A. Ishikawa, H. Tajima, and N. Fukushima, ''Halide implementation of weighted median filter,'' in *Proc. Int. Workshop Adv. Imag. Technol. (IWAIT)*, Jun. 2020, pp. 535–539, doi: 10.1117/12.2566536.

[77] A. Ishikawa, N. Fukushima, A. Maruoka, and T. Iizuka, ''Halide and GENESIS for generating domain-specific architecture of guided image filtering,'' in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2019, pp. 1–5, doi: 10.1109/ISCAS.2019.8702260.

[78] J. Zhang, F. Franchetti, and T. M. Low, ''High performance zero-memory overhead direct convolutions,'' in *Proc. Int. Conf. Mach. Learn. (ICML)*, 2018, pp. 5776–5785. [Online]. Available: https://proceedings.mlr.press/v80/zhang18d.html?ref=https://githubhelp.com

[79] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, ''Anatomy of high-performance deep learning convolutions on SIMD architectures,'' in *Proc. SC Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2018, pp. 830–841, doi: 10.1109/SC.2018.00069.

[80] H. Kataoka, K. Yamashita, Y. Ito, K. Nakano, A. Kasagi, and T. Tabaru, ''An efficient multicore CPU implementation for convolution-pooling computation in CNNs,'' in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, May 2020, pp. 548–556, doi: 10.1109/IPDPSW50202.2020.00097.

[81] V. Ferrari, R. Sousa, M. Pereira, J. P. L. De Carvalho, J. N. Amaral, J. Moreira, and G. Araujo, ''Advancing direct convolution using convolution slicing optimization and ISA extensions,'' *ACM Trans. Archit. Code Optim.*, vol. 20, no. 4, pp. 1–26, Dec. 2023, doi: 10.1145/3625004.

[82] V. Kelefouras and G. Keramidas, ''Design and implementation of deep learning 2D convolutions on modern CPUs,'' *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 12, pp. 3104–3116, Dec. 2023, doi: 10.1109/TPDS.2023.3322037.

[83] P. San Juan, A. Castelló, M. F. Dolz, P. Alonso-Jordá, and E. S. Quintana-Ortí, ''High performance and portable convolution operators for multicore processors,'' in *Proc. IEEE 32nd Int. Symp. Comput. Archit. High Perform. Comput. (SBAC-PAD)*, Sep. 2020, pp. 91–98, doi: 10.1109/SBAC-PAD49847.2020.00023.

[84] S. Barrachina, M. F. Dolz, P. San Juan, and E. S. Quintana-Ortí, ''Efficient and portable GEMM-based convolution operators for deep neural network training on multicore processors,'' *J. Parallel Distrib. Comput.*, vol. 167, pp. 240–254, Sep. 2022, doi: 10.1016/j.jpdc.2022.05.009.

[85] M. Seznec, N. Gac, F. Orieux, and A. Sashala Naik, ''Computing large 2D convolutions on GPU efficiently with the im2tensor algorithm,'' *J. Real-Time Image Process.*, vol. 19, no. 6, pp. 1035–1047, Dec. 2022, doi: 10.1007/s11554-022-01240-0.

[86] S. Lu, J. Chu, and X. T. Liu, ''Im2win: Memory efficient convolution on SIMD architectures,'' in *Proc. IEEE High Perform. Extreme Comput. Conf. (HPEC)*, Sep. 2022, pp. 1–7, doi: 10.1109/HPEC55821.2022.9926408.

[87] A. Zlateski, Z. Jia, K. Li, and F. Durand, ''The anatomy of efficient FFT and Winograd convolutions on modern CPUs,'' in *Proc. ACM Int. Conf. Supercomputing*, Jun. 2019, pp. 414–424, doi: 10.1145/3330345.3330382.

[88] Q. Wang, D. Li, X. Huang, S. Shen, S. Mei, and J. Liu, ''Optimizing FFT-based convolution on ARMV8 multi-core CPUs,'' in *Proc. Eur. Conf. Parallel Process.*, 2020, pp. 248–262, doi: 10.1007/978-3-030-57675-2_16.

[89] S. Winograd, *Arithmetic Complexity of Computations*, vol. 33. Philadelphia, PA, USA: SIAM, 1980, doi: 10.1137/1.9781611970364.

[90] A. Lavin and S. Gray, ''Fast algorithms for convolutional neural networks,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021, doi: 10.1109/CVPR.2016.435.

[91] M. F. Dolz, A. Castelló, and E. S. Quintana-Ortí, ''Towards portable realizations of Winograd-based convolution with vector intrinsics and OpenMP,'' in *Proc. 30th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Mar. 2022, pp. 39–46, doi: 10.1109/PDP55904.2022.00015.

[92] M. F. Dolz, H. Martínez, A. Castelló, P. Alonso-Jordá, and E. S. Quintana-Ortí, ''Efficient and portable Winograd convolutions for multi-core processors,'' *J. Supercomput.*, vol. 79, no. 10, pp. 10589–10610, Jul. 2023, doi: 10.1007/s11227-023-05088-4.

[93] S. Paris and F. Durand, ''A fast approximation of the bilateral filter using a signal processing approach,'' in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2006, pp. 568–580, doi: 10.1007/11744085_44.

[94] F. Porikli, ''Constant time O(1) bilateral filtering,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2008, pp. 1–2, doi: 10.1109/CVPR.2008.4587843.

[95] Q. Yang, K.-H. Tan, and N. Ahuja, ''Real-time O(1) bilateral filtering,'' in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 557–564, doi: 10.1109/CVPR.2009.5206542.

[96] K. N. Chaudhury, D. Sage, and M. Unser, ''Fast $O(1)$ bilateral filtering using trigonometric range kernels,'' *IEEE Trans. Image Process.*, vol. 20, no. 12, pp. 3376–3382, Dec. 2011, doi: 10.1109/TIP.2011.2159234.

[97] K. N. Chaudhury, ''Acceleration of the shiftable $O(1)$ algorithm for bilateral filtering and nonlocal means,'' *IEEE Trans. Image Process.*, vol. 22, no. 4, pp. 1291–1300, Apr. 2013, doi: 10.1109/TIP.2012.2222903.

[98] K. Sugimoto and S.-I. Kamata, ''Compressive bilateral filtering,'' *IEEE Trans. Image Process.*, vol. 24, no. 11, pp. 3357–3369, Nov. 2015, doi: 10.1109/TIP.2015.2442916.

[99] N. Fukushima, K. Sugimoto, and S.-I. Kamata, ''Complex coefficient representation for IIR bilateral filter,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2017, pp. 2458–2462, doi: 10.1109/ICIP.2017.8296724.

[100] K. Sugimotoy, N. Fukushimazy, and S.-I. Kamatay, ''200 FPS constant-time bilateral filter using SVD and tiling strategy,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2019, pp. 190–194, doi: 10.1109/ICIP.2019.8802927.

[101] Y. Sumiya, N. Fukushima, K. Sugimoto, and S.-I. Kamata, ''Extending compressive bilateral filtering for arbitrary range kernel,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2020, pp. 1018–1022, doi: 10.1109/ICIP40778.2020.9191123.

[102] K. Sugimoto and S.-I. Kamata, ''Fast image filtering by DCT-based kernel decomposition and sequential sum update,'' in *Proc. 19th IEEE Int. Conf. Image Process.*, Sep. 2012, pp. 125–128, doi: 10.1109/ICIP.2012.6466811.

[103] T. Otsuka, N. Fukushima, Y. Maeda, K. Sugimoto, and S.-I. Kamata, ''Optimization of sliding-DCT based Gaussian filtering for hardware accelerator,'' in *Proc. IEEE Int. Conf. Vis. Commun. Image Process. (VCIP)*, Dec. 2020, pp. 423–426, doi: 10.1109/VCIP49819.2020.9301775.

[104] Q. Yang, N. Ahuja, and K.-H. Tan, ''Constant time median and bilateral filtering,'' *Int. J. Comput. Vis.*, vol. 112, no. 3, pp. 307–318, May 2015, doi: 10.1007/s11263-014-0764-y.

[105] C. Karam, C. Chen, and K. Hirakawa, ''Stochastic bilateral filter for high-dimensional images,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2015, pp. 192–196, doi: 10.1109/ICIP.2015.7350786.

[106] S. Ghosh and K. N. Chaudhury, ''Fast bilateral filtering of vector-valued images,'' in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2016, pp. 1823–1827, doi: 10.1109/ICIP.2016.7532673.

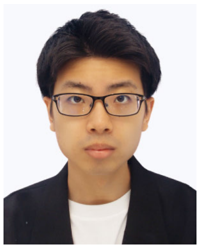
[107] W.-C. Tu, Y.-A. Lai, and S.-Y. Chien, "Constant time bilateral filtering for color images," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Sep. 2016, pp. 3309–3313, doi: 10.1109/ICIP.2016.7532972.

[108] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proc. Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2007, pp. 1027–1035, doi: 10.1145/1283383.1283494.

[109] M. G. Mozerov and J. van de Weijer, "Global color sparseness and a local statistics prior for fast bilateral filtering," *IEEE Trans. Image Process.*, vol. 24, no. 12, pp. 5842–5853, Dec. 2015, doi: 10.1109/TIP.2015.2492822.

[110] K. Sugimoto, N. Fukushima, and S.-I. Kamata, "Fast bilateral filter for multichannel images via soft-assignment coding," in *Proc. Asia–Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA)*, Dec. 2016, pp. 1–4, doi: 10.1109/APSIPA.2016.7820813.

[111] T. Miyamura, N. Fukushima, M. Waqas, K. Sugimoto, and S.-I. Kamata, "Image tiling for clustering to improve stability of constant-time color bilateral filtering," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Oct. 2020, pp. 1038–1042, doi: 10.1109/ICIP40778.2020.9191059.



YUKI NAGANAWA (Graduate Student Member, IEEE) received the B.E. degree from the Nagoya Institute of Technology, in 2022, where he is currently pursuing the master's degree in computer science. His research interests include computer vision and image processing.

HIROKAZU KAMEI received the B.E. degree from the Nagoya Institute of Technology, in 2022, where he is currently pursuing the master's degree in computer science. His research interests include image processing and programming languages.

YAMATO KANETAKA (Graduate Student Member, IEEE) received the B.E. degree from the Nagoya Institute of Technology, in 2022, where he is currently pursuing the master's degree in computer science. His research interests include image processing, programming languages, and iOS.

HARUKI NOGAMI (Graduate Student Member, IEEE) received the B.E. degree from the Nagoya Institute of Technology, in 2022, where he is currently pursuing the master's degree in computer science. His research interests include image processing and programming languages.

YOSHIHIRO MAEDA (Member, IEEE) received the B.E., M.E., and Ph.D. degrees in information engineering from the Nagoya Institute of Technology, Japan, in 2013, 2015, and 2019, respectively. He became an Assistant Professor with the Tokyo University of Science, Japan, in 2019. His research interests include image signal processing, parallel image processing, and multispectral sensing. He is a member of IEICE.

NORISHIGE FUKUSHIMA (Member, IEEE) received the B.E., M.E., and Ph.D. degrees from Nagoya University, Japan, in 2004, 2006, and 2009, respectively. He became an Assistant Professor and an Associate Professor with the Nagoya Institute of Technology, Japan, in 2009 and 2015, respectively. His research interests include image signal processing, parallel image processing, and compilers. He is a member of IEEE CAS, IEEE SPS, IEICE, and IPSJ.