Acceleration of Edge-Preserving Filtering on CPU Microarchitecture

CPUマイクロアーキテクチャに応じたエッジ保存平滑化フィルタの高速化

2019

Yoshihiro Maeda

Contents

1 Introduction **1**
1.1 Research Background . . . 1
1.2 Edge-Preserving Filtering . . . 2
1.2.1 FIR Filtering . . . 2
1.2.2 Bilateral Filtering . . . 3
1.2.3 Non-Local Means Filtering . . . 3
1.2.4 Guided Filtering . . . 4
1.3 Overview of CPU Microarchitectures . . . 5
1.4 Organization of This Dissertation . . . 6

2 Taxonomy of Vectorization Patterns of Programming for FIR Image Filters Using Kernel Subsampling and New One **9**
2.1 Introduction . . . 9
2.2 2D FIR Image Filtering and Its Acceleration . . . 12
2.2.1 Definition of 2D FIR Image Filtering . . . 12
2.2.2 General Acceleration of FIR Image Filtering . . . 12
2.3 Design Patterns of Vectorized Programming for FIR Image Filtering . . . 13
2.3.1 Data Loading and Storing in Vectorized Programming 13
2.3.2 Image Data Structure . . . 14
2.3.3 Vectorization of FIR Filtering . . . 15
2.3.4 Color Loop Unrolling . . . 16
2.3.5 Kernel Loop Unrolling . . . 17
2.3.6 Pixel Loop Unrolling . . . 18
2.4 Proposed Design Pattern of Vectorization . . . 19
2.5 Material and Methods . . . 22
2.5.1 Gaussian Range Filter . . . 22
2.5.2 Bilateral Filter . . . 23
2.5.3 Adaptive Gaussian Filter . . . 24
2.5.4 Randomly-Kernel-Subsampled Filter . . . 24
2.6 Experimental Results . . . 25

2.7 Conclusions 27

3 Effective Implementation of Edge-Preserving Filtering on CPU Microarchitectures 37

3.1 Introduction 37
3.2 Edge-Preserving Filters 40
3.3 Floating Point Numbers and Denormalized Numbers in IEEE Standard 754 41
3.4 CPU Microarchitectures and SIMD Instruction Sets 43
3.5 Proposed Methods for the Prevention of Denormalized Numbers 44
3.6 Effective Implementation of Edge-Preserving Filtering 47
3.7 Experimental Results 51
3.7.1 Influence of Denormalized Numbers 51
3.7.2 Effective Implementation on CPU Microarchitectures . 53
3.8 Conclusions 55

4 Directional Cubic Convolution Interpolation with Edge Preserving Detail Enhancement 71

4.1 Introduction 71
4.2 Proposed Framework 72
4.3 Experimental Results and Discussion 73
4.4 Conclusions 74

5 Conclusion 79

A Pixel Subsampling vs. Kernel Subsampling 99

B Implementation of Bilateral Filter in OpenCV 101

Chapter 1

Introduction

1.1 Research Background

Image processing is essential to realize various applications. Edge-preserving filters [1–5] are the basic tools used for image processing. Representative filtering include bilateral filtering [1,2], non-local means filtering [3], and guided image filtering [4–6]. These filters are used in various applications, such as image denoising [3, 7], high dynamic range imaging [8], detail enhancement [9–11], free viewpoint image rendering [12], flash/no-flash photography [13,14], up-sampling/super resolution [15,16], alpha matting [17,18], haze removal [19], optical flow and stereo matching [20], refinement processing in optical flow and stereo matching [21, 22], and coding noise removal [23, 24].

These filters have high computational cost because they compute adaptively for each pixel. Several acceleration algorithms have been proposed for the bilateral filtering [8, 25–33] and non-local means filtering [25, 31, 34, 35]. These algorithms reduce the computational order of these filtering. The order of a naïve algorithm is $O(r^2)$, where r is the kernel radius. The order of separable approximation algorithms [25, 26] is $O(r)$ and that of constant-time algorithms [28–37] is $O(1)$. In multi-channel image filtering with intermediate-sized kernels, the constant-time algorithms tend to be slower than the naïve algorithms owing to the curse of dimensionality [28], which indicates that the computational cost increases exponentially with increasing dimensions. Furthermore, when the kernel radius is small, a naïve algorithm can be faster than algorithms of the order $O(r)$ or $O(1)$ owing to the offset times of the latter algorithms. Therefore, we should also consider another acceleration approach by hardware, such as central processing units.

Moore's law [38] indicates that the number of transistors on an integrated circuit will double every two years. Early in the development of integrated

circuits, the increased numbers of transistors were largely devoted to increase clock speeds of CPUs. However, the CPU frequency is limited by power and thermal constraints; therefore, the utilization of the increased numbers of transistors has become complex [39]. Nowadays, most CPUs have multi-core architectures, complicated cache memories, and short vector processing units. Hence, In the early stage, due to growing CPU frequency, the performance of software improved without software optimization. However, nowadays, the performance cannot be improved without the optimization, such as cache-efficient parallelized and vectorized programming. This fact is called "The Free Lunch Is Over." [40]

Flynn's taxonomy [41] categorizes multi-core parallel programming as multiple-instruction, multiple data (MIMD) type and vectorized programming as single-instruction, multiple data (SIMD) type. Single-instruction, multiple threads (SIMT) is the same concept as SIMD in GPUs. Vectorization and parallelization can be simultaneously used in image processing applications. Vectorized programming, however, requires harder constraints than parallel programming in data structures. Vendor's short SIMD architectures, such as MMX, Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)/AVX2, AVX-512, AltiVec, and NEON, are expected to develop rapidly, and vector lengths will become longer [42]. SIMD instruction sets are changed by the microarchitecture of the CPU. This implies that vectorization is critical for effective programming.

To accelerate the edge-preserving filtering, we should be considering implementation of them for CPU microarchitectures. In this dissertation, we aim acceleration of the edge-preserving filtering and organize cyclopaedically effective implementation on CPU microarchitectures. Also, we focus on acceleration of upsampling and detail enhancement, which are one of application in the edge-preserving filtering.

1.2 Edge-Preserving Filtering

1.2.1 FIR Filtering

General edge-preserving filtering in finite impulse response (FIR) filtering is represented as follows:

$$\bar{\boldsymbol{I}}(\boldsymbol{p}) = \frac{1}{\eta} \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p}, \boldsymbol{q}) \boldsymbol{I}(\boldsymbol{q}), \tag{1.1}$$

where $\boldsymbol{I}$ and $\bar{\boldsymbol{I}}$ are the input and output images, respectively. $\boldsymbol{p}$ and $\boldsymbol{q}$ are the present and reference positions of pixels, respectively. A kernel-shaped

function $\mathcal{N}(\boldsymbol{p})$ comprises a set of reference pixel positions, and it varies for every pixel $\boldsymbol{p}$. The function $f(\boldsymbol{p},\boldsymbol{q})$ denotes the weight of position $\boldsymbol{p}$ with respect to the position $\boldsymbol{q}$ of the reference pixel. η is a normalizing function. If the gain of the FIR filter is one, we set the normalizing function as follows:

$$\eta = \sum_{\boldsymbol{q}\in\mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p},\boldsymbol{q}). \tag{1.2}$$

The edge-preserving filtering smooths an image while edges are kept. Various types of weight functions are employed in edge-preserving filtering. The edge-preserving filter can typically be decomposed into range and/or spatial kernels, which depend on the value and position of each pixel, respectively. The spatial kernel is invariant across all pixels. By contrast, the range kernel is variant. Thus, the range kernel is computed adaptively for each pixel, which means the process of edge-preserving filtering is computationally expensive.

1.2.2 Bilateral Filtering

The weight of the bilateral filter is expressed as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) \;:=\; \exp(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s^2})\exp(\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}), \tag{1.3}$$

where $\|\cdot\|_2$ is the L2 norm and σ_s and σ_r are the standard deviations of the spatial and the range kernels, respectively. The weight of the bilateral filter is determined by considering the similarity between the color and spatial distance between a target pixel and that of the reference pixel.

1.2.3 Non-Local Means Filtering

The weight of the non-local means filter is as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) \;:=\; \exp(\frac{\|\boldsymbol{v}(\boldsymbol{p})-\boldsymbol{v}(\boldsymbol{q})\|_2^2}{-h^2}), \tag{1.4}$$

where $\boldsymbol{v}(\boldsymbol{p})$ represents a vector, which includes a square neighborhood of the center pixel $\boldsymbol{p}$. h is a smoothing parameter. The weight of the non-local means filter is defined by computing the similarity between the patch on the target pixel and that on the reference pixel. It is similar to the range weight of the bilateral filter for a multi-channel image.

1.2.4 Guided Filtering

The guided filtering converts local patches in an input image by a linear transformation of a guide image. Let the guide signal be I. The output q is assumed as follows;

$$q_{\boldsymbol{i}} = a_{\boldsymbol{k}} I_{\boldsymbol{i}} + b_{\boldsymbol{k}}, \forall i \in \omega_{\boldsymbol{k}} \tag{1.5}$$

where $\boldsymbol{k}$ indicates a center position of a rectangular patch $\omega_{\boldsymbol{k}}$, and $\boldsymbol{i}$ indicates a position of a pixel in the patch. $a_{\boldsymbol{k}}$ and $b_{\boldsymbol{k}}$ are coefficients for the linear transformation. The equation represents that guide signals in a patch are linearly converted by the coefficients.

The coefficients are calculated by a linear regression of the input signal p and (1.5).

$$\arg\min_{a_{\boldsymbol{k}}, b_{\boldsymbol{k}}} = \sum_{\boldsymbol{i} \in \omega_{\boldsymbol{k}}} ((a_{\boldsymbol{k}} I_{\boldsymbol{i}} + b_{\boldsymbol{k}} - p_{\boldsymbol{i}})^2 + \epsilon a_{\boldsymbol{k}}^2) \tag{1.6}$$

The coefficients are estimated as follows;

$$a_{\boldsymbol{k}} = \frac{cov_{\boldsymbol{k}}(I, p)}{var_{\boldsymbol{k}}(I) + \epsilon}, \quad b_{\boldsymbol{k}} \ = \bar{p}_{\boldsymbol{k}} - a_{\boldsymbol{k}} \bar{I}_{\boldsymbol{k}}, \tag{1.7}$$

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 over overlapping in the output signals; thus, these coefficient are averaged.

$$\bar{a}_{\boldsymbol{i}} = \frac{1}{|\omega|} \sum_{\boldsymbol{k} \in \omega_{\boldsymbol{i}}} a_{\boldsymbol{k}}, \quad \bar{b}_{\boldsymbol{i}} = \frac{1}{|\omega|} \sum_{\boldsymbol{k} \in \omega_{\boldsymbol{i}}} b_{\boldsymbol{k}}, \tag{1.8}$$

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

$$q_{\boldsymbol{i}} = \bar{a}_{\boldsymbol{i}} I_{\boldsymbol{i}} + \bar{b}_{\boldsymbol{i}}, \tag{1.9}$$

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 $\boldsymbol{I}$, respectively. The per channel filtering output is defined as follows;

$$q_{\boldsymbol{i}}^n = \bar{\boldsymbol{a}}_{\boldsymbol{i}}^{n\mathrm{T}} \boldsymbol{I}_{\boldsymbol{i}} + \bar{b}_{\boldsymbol{i}}^n, \tag{1.10}$$

$$\bar{\boldsymbol{a}}_{\boldsymbol{i}}^n = \frac{1}{|\omega|} \sum_{\boldsymbol{k} \in \omega_{\boldsymbol{i}}} \boldsymbol{a}_{\boldsymbol{k}}^n, \quad \bar{b}_{\boldsymbol{i}}^n = \frac{1}{|\omega|} \sum_{\boldsymbol{k} \in \omega_{\boldsymbol{i}}} b_{\boldsymbol{k}}^n, \tag{1.11}$$

Table 1.1: CPU microarchitectures of Intel Core series Extreme Editions. In each CPU generation, the Extreme Edition versions offer the highest performance on the consumer level.

Generation	**1st**	**2nd**	**3rd**	**4th**	**5th**	**6th**
codename	Gulftown	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake
model	990X	3970X	4960X	5960X	6950X	7980XE
launch date	Q1'11	Q4'12	Q3'13	Q3'14	Q2'16	Q3'17
lithography	32 nm	32 nm	22 nm	22 nm	14 nm	14 nm
base frequency [GHz]	3.46	3.50	3.60	3.00	3.00	2.60
max turbo frequency [GHz]	3.73	4.00	4.00	3.50	3.50	4.20
number of cores	6	6	6	8	10	18
L1 cache (×number of cores)	64 KB (data cache 32 KB, instruction cache 32 KB)					
L2 cache (×number of cores)	256 KB	256 KB	25 6KB	256 KB	256 KB	1 MB
L3 cache	12 MB	15 MB	15 MB	20 MB	25 MB	24.75 MB
memory types	DDR3-1066	DDR3-1600	DDR3-1866	DDR4-2133	DDR4-2133	DDR4-2666
max number of memory channels	3	4	4	4	4	4
SIMD instruction sets	SSE4.2	SSE4.2 AVX	SSE4.2 AVX	SSE4.2 AVX/AVX2 FMA3	SSE4.2 AVX/AVX2 FMA3	SSE4.2 AVX/AVX2 AVX512 FMA3

The coefficients $\boldsymbol{a}_{\boldsymbol{k}}^n$, $b_{\boldsymbol{k}}^n$ for the linear transformation is obtained as follows;

$$\boldsymbol{a}_{\boldsymbol{k}}^n = \frac{cov_{\boldsymbol{k}}(\boldsymbol{I}, p^n)}{var_{\boldsymbol{k}}(\boldsymbol{I}) + \epsilon \boldsymbol{E}}, \quad b_{\boldsymbol{k}}^n = \bar{p}_{\boldsymbol{k}}^n - \boldsymbol{a}_{\boldsymbol{k}}^{n\mathrm{T}} \bar{\boldsymbol{I}}_{\boldsymbol{k}}, \tag{1.12}$$

where $\boldsymbol{E}$ is an identity matrix. When the output signal is color image, $cov_{\boldsymbol{k}}$ is the covariance matrix of the patch in p and $\boldsymbol{I}$. Also, $var_{\boldsymbol{k}}$ is the variance of the R, G, and B components, which will be covariance matrix, in the patch of $\boldsymbol{I}$. The division of the matrix is calculated by multiplying the inverse matrix of the denominator from the left. The calculation results of per pixel mean, variance, and covariance are obtained from the box filter. The filter can be implemented with a recursive filter [43], which can work in a constant time per pixel.

1.3 Overview of CPU Microarchitectures

The latest microarchitectures used in Intel CPUs are presented in Table 1.1. The table indicates that CPU frequencies are hardly growing. However, the number of cores is increasing, cache memory size is expanding and the SIMD instruction sets are growing. Therefore, we need to use parallelization and SIMD instruction sets for effective implementation.

SIMD instructions simultaneously calculate multiple data. Hence, high-performance computing utilizes SIMD. The SIMD instructions are classified data-level parallelization. Typical SIMD instructions include streaming SIMD extensions (SSE), advanced vector extensions (AVX)/AVX2 and

AVX512 in order of the oldest to newest [44]. Moreover, fused multiply-add 3 (FMA3) [44] is a special instruction. FMA3 computes $A \times B + C$ by one instruction. There are three notable changes in SIMD. First, the vector length is growing. For example, the lengths of SSE, AVX/AVX2 and AVX512 are 128 bits (4 float elements), 256 bits (8 float elements) and 512 bits (16 float elements), respectively. Second, several instructions have been added, notably, *gather* and *scatter* instructions [42]. These instructions load/store data of discontinuous positions in memory. *gather* has been implemented in the AVX2, and *scatter* has been implemented in the AVX512.

1.4 Organization of This Dissertation

This dissertation mainly discusses the acceleration of the edge-preserving filtering and organize effective implementation on CPU microarchitectures. Furthermore, we focus on acceleration and high accuracy of upsampling and detail enhancement, which are one of application in the edge-preserving filtering.

Chapter 2 examines vectorized programming for finite impulse response image filtering. Finite impulse response image filtering occupies a fundamental place in image processing, and has several approximated acceleration algorithms. However, no sophisticated method of acceleration exists for parameter adaptive filters or any other complex filter. For this case, simple subsampling with code optimization is a unique solution. Under the current Moore's law, increases in central processing unit frequency have stopped. Moreover, the usage of more and more transistors is becoming insuperably complex due to power and thermal constraints. Most central processing units have multi-core architectures, complicated cache memories, and short vector processing units. This change has complicated vectorized programming. Therefore, we first organize vectorization patterns of vectorized programming to highlight the computing performance of central processing units by revisiting the general finite impulse response filtering. Furthermore, we propose a new vectorization pattern of vectorized programming and term it as loop vectorization. Moreover, these vectorization patterns mesh well with the acceleration method of subsampling of kernels for general finite impulse response filters. Experimental results reveal that the vectorization patterns are appropriate for general finite impulse response filtering. A new vectorization pattern with kernel subsampling is found to be effective for various filters. These include Gaussian range filtering, bilateral filtering, adaptive Gaussian filtering, randomly-kernel-subsampled Gaussian range fil-

tering, randomly-kernel-subsampled bilateral filtering, and randomly-kernel-subsampled adaptive Gaussian filtering.

In Chapter 3, we propose acceleration methods for edge-preserving filtering. The filters natively include denormal numbers, which are defined in IEEE standard 754. The processing of denormal numbers has a higher computational; thus, the computational performance of edge-preserving filtering is diminished severely. We propose approaches to prevent the occurrence of denormal numbers. Moreover, we verify an effective vectorized implementation of the edge-preserving filtering based on changes in central processing unit microarchitectures by carefully treating kernel weights. The experimental results show that the proposed methods are up to five times faster than the straightforward implementation of bilateral filtering and non-local means filtering while maintaining high accuracy. In addition, we achieved effective vectorized implementation for each central processing unit microarchitecture. The effective vectorized implementation of the bilateral filter is up to 14 times faster than OpenCV implementation. The proposed methods and the effective vectorized implementation are practical for real-time tasks such as image editing.

In Chapter 4, we propose a framework can handle simultaneous processing of directional cubic convolution interpolation and detail enhancement. As a result of the experiment, the proposed method achieves upsampling with higher precision than the conventional method and at the same time achieves its approximate speedup as a detailed enhancement.

Finally, we conclude our work, in Chapter 5.

Chapter 2

Taxonomy of Vectorization Patterns of Programming for FIR Image Filters Using Kernel Subsampling and New One

2.1 Introduction

Image processing is known as high-load processing. Accordingly, vendors of central processing units (CPUs) and graphics processing units (GPUs) provide tuned libraries, such as the Intel Integrated Performance Primitives (Intel IPP) and NVIDIA Performance Primitives (NPP). Open-source communities also provide optimized image processing libraries, such as OpenCV, OpenVX, boost Generic Image Library (GIL), and scikit-image.

Moore's law [38] indicates that the number of transistors on an integrated circuit will double every two years. Early in the development of integrated circuits, the increased numbers of transistors were largely devoted to increase clock speeds of CPUs. Power and thermal constraints limit increases in CPU frequency, and the utilization of the increased numbers of transistors has become complex [39]. Nowadays, most CPUs have multi-core architectures, complicated cache memories, and short vector processing units. To maximize code performance, cache-efficient parallelized and vectorized programming is essential.

Flynn's taxonomy [41] categorizes multi-core parallel programming as multiple-instruction, multiple data (MIMD) type and vectorized programming as single-instruction, multiple data (SIMD) type. Single-instruction, multiple threads (SIMT) is the same concept as SIMD in GPUs. Vector-

ization and parallelization can be simultaneously used in image processing applications. Vectorized programming, however, requires harder constraints than parallel programming in data structures. Vendor's short SIMD architectures, such as MMX, Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX)/AVX2, AVX-512, AltiVec, and NEON, are expected to develop rapidly, and vector lengths will become longer [42]. SIMD instruction sets are changed by the microarchitecture of the CPU. This implies that vectorization is critical for effective programming.

Effective vectorization requires the consideration of three critical issues: memory alignment, valid vectorization ratio, and cache efficiency. Memory alignment is critical for data loading because SIMD operations load excessive data from non-aligned data in memory. This loading involves significant penalties. In valid vectorization ratio issues, padding operations remain a major topic of discussion. Padding data are inevitable in exception handling of extra data for vectorized loading. Moreover, rearranging data with padding resolves alignment issues [45]. However, padding decreases ratios of valid vectorized computing. For cache efficiency, there is a tremendous penalty for cache-missing because the cost of loading data from main memory is approximately 100 times higher than the cost of adding data. These issues can be moderated in various ways. Among such means are data padding for memory alignment, loop fusion/jamming, loop fission, tiling, selecting a loop number in multiple loops for parallelization and vectorization, data transformation [46], and so on.

We should completely utilize the functionality of the CPU/GPU for accelerating image processing using hardware [47, 48]. Vectorized programming matches image processing; thus, typical simple algorithms can be accelerated [49]. Even in cases where algorithms have more efficient computing orders, the parallelized and vectorized implementation of another higher-order algorithm would still be faster than the optimal algorithm in many cases. In parallel computers, for example, a bitonic sort [50] is faster than a quick sort [51]. In image processing, the brute-force implementation of box filtering proceeds more rapidly than the integral image [52] for small kernel-size cases. In both cases, optimal algorithms, i.e., the quick sort and integral image, do not have the appropriate data structure for parallelized and vectorized programming.

In image processing, various algorithmic acceleration have been proposed other than hardware acceleration. In particular, these include general and specific acceleration algorithms in finite impulse response (FIR) filtering. General FIR filtering allows the acceleration of filters by separable filtering [53, 54], image subsampling [55], and kernel subsampling [56]. Separable filters reduce the computational order from $O(r^2)$ to $O(r)$, where r denotes

the kernel radius. Separable filter requires the filtering kernel to be separable. Image subsampling and kernel subsampling are approximated acceleration. Image subsampling is faster than kernel subsampling; however, the accuracy of image subsampling is lower than that of kernel subsampling. For specific filters, such as Gaussian filters [57–64], bilateral filters [8, 25–31], box filters [43, 52], and non-local means filters [31], various acceleration algorithms exist.

In parameter-adaptive filters and other complex filters, however, there is no sophisticated way for acceleration. In such filters, there is no choice but to apply separable filtering, image subsampling, and kernel subsampling, with / without SIMD vectorization and MIMD parallelization. Separable filtering requires the filtering kernel to be separable, and such filters are not usually separable. Furthermore, image subsampling has low accuracy. Therefore, kernel subsampling with code optimization is the only solution. However, discontinuous access occurs in kernel subsampling; hence, the efficiency of vectorized programming greatly decreases.

Therefore, we summarize the vectorized patterns of programming for subsampled filtering to verify the effective programming. Moreover, we propose an effective preprocessing of data structure transformation and vectorized filtering with the data structure for this case. Note that the transformation becomes overhead; thus, we focus on the situation that can ignore the pre-processing time. The situation is interactive filtering, such as photo editing. Once the data structure is transformed, then we can filter an image to seek optimal parameters without preprocessing, because the filtering image already has been converted.

In this chapter, we contribute the following: We summarize a taxonomy of vectorized programming of FIR image filtering as vectorization patterns. We propose a new vectorizing pattern. Moreover, the proposed pattern is oriented to kernel subsampling. These patterns with kernel subsampling accelerate FIR filters, which do not have sophisticated algorithms. Moreover, the proposed pattern is practical for interactive filters.

The remainder of this chapter is organized as follows. Section 2.2 reviews general FIR filters. Section 2.3 systematizes vectorized programming for FIR image filters as vectorization patterns. Section 2.4 proposes a new vectorization pattern for FIR filtering. Section 2.5 introduces target algorithms of filtering for vectorization. Section 2.6 shows experimental results. Finally, Section 2.7 concludes this chapter.

2.2 2D FIR Image Filtering and Its Acceleration

2.2.1 Definition of 2D FIR Image Filtering

2D FIR filtering is typical image processing. It is defined as follows:

$$\bar{\boldsymbol{I}}(\boldsymbol{p}) = \frac{1}{\eta} \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p}, \boldsymbol{q}) \boldsymbol{I}(\boldsymbol{q}), \tag{2.1}$$

where $\boldsymbol{I}$ and $\bar{\boldsymbol{I}}$ are the input and output images, respectively. $\boldsymbol{p}$ and $\boldsymbol{q}$ are the current and reference positions, respectively. A kernel-shape function $\mathcal{N}(\boldsymbol{p})$ comprises a set of reference pixel positions, and varies at every pixel $\boldsymbol{p}$. The weight function $f(\boldsymbol{p}, \boldsymbol{q})$ is the weight of the position $\boldsymbol{p}$ with regard to the position $\boldsymbol{q}$ of the reference pixel. The function f could change at pixel position $\boldsymbol{p}$. η is a normalizing function. If the FIR filter's gain is 1, we set the normalizing function to be the following:

$$\eta = \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p}, \boldsymbol{q}). \tag{2.2}$$

2.2.2 General Acceleration of FIR Image Filtering

Several approaches have been taken with regard to the acceleration of general FIR filters. These include separable filtering [53], image subsampling [55], and kernel subsampling [56]. In the separable filtering, the filtering kernel is separated into vertical and horizontal kernels as a 1D filter chain using the separability of the filtering kernel. The general 2D FIR filter has the computational order of $O(r^2)$ for each pixel, where r denotes the kernel radius of the filter. The computational order of a separable filter is $O(r)$. If the filtering kernel is not separable, either singular value decomposition (SVD) or truncated SVD can be used to create separable kernels. When truncated SVD is used, the image is forcefully smoothed with a few sets of separable kernels for acceleration. However, when kernel weight changes for each pixel, we need SVD computation for every pixel. Here, the separable approach is inefficient.

Image subsampling resizes an input image and then filters it. Finally, the filtered image is upsampled. This subsampling greatly accelerates filtering, but the accuracy of approximation is not high. Further, the method has the significant drawback of losing high-frequency signals.

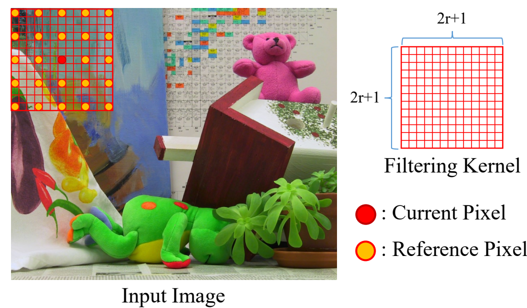


Figure 2.1: Example of kernel subsampling. Only samples of current (red) and reference (yellow) pixels are computed.

Kernel subsampling reduces the number of reference pixels in the filtering kernel as a similar approach to image subsampling. Figure 2.1 represents kernel subsampling. The reduction of computational time in kernel subsampling is not as extensive as that of image subsampling, while kernel subsampling could keep a higher approximation accuracy than image subsampling. Thus, we focus on kernel subsampling. In the Appendix, we examine the processing time and accuracy of image subsampling and kernel subsampling more closely. The approximation accuracy of these types of subsampling depends on the ratio and pattern of subsampling. Image and kernel subsampling generate aliasing, but a randomized algorithm [65, 66] moderates this negative effect. Random sampling reduces defective results from aliasing for human vision [67]. Random-sampling algorithms were first introduced in accelerating ray tracing and were utilized for FIR filtering in [55, 56].

The main subject of this chapter is the general acceleration of FIR filtering by using SIMD vectorization. We adopt kernel subsampling for acceleration because kernel subsampling has a high accuracy of approximation and is not limited by the type of kernel.

2.3 Design Patterns of Vectorized Programming for FIR Image Filtering

2.3.1 Data Loading and Storing in Vectorized Programming

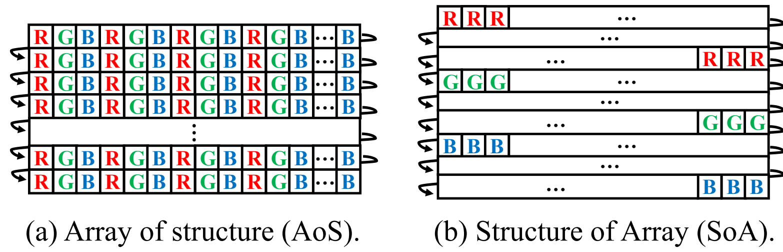
The SIMD operations calculate multiple data at once; hence, the SIMD operations are high performance. Such operations constrain all vector elements to follow the same control flow. That is, only one element in a vector cannot be processed with a different operation as a conditional branch. Therefore,

we require data structures, wherein processing data are continuously in memory, for the load and store instructions. Such instructions move continuous data from the memory/register to the register/memory. For this case, spatial locality in memory is high. On the other hand, we can relieve the restriction by performing discontinuous loading and storing. The operations are realized with set instruction or scalar operations. These methods use scalar registers; thus, these methods are slower than the load and store instructions. Recent SIMD instruction sets have the gather and scatter instructions, which load and store for discontinuous positions in memory. However, such instructions also have higher latency than the sequential load and store instructions. Discontinuous load and store operations also decrease spatial locality in memory access; hence, cache-missing occurs. Cache-missing decreases performance. Moreover, memory alignment is important for the load and store instructions. Most CPUs are word-oriented. Data are aligned in word-length chunks, with 32 bits and 64 bits. In aligned data loading and storing, CPUs access memory only once. In non-aligned data loading and storing, CPUs access memory twice. Therefore, the performance of the load and store instructions decreases if non-aligned data loading or storing occurs. Furthermore, vectorized programming requires padding if the data size is smaller than the SIMD register size. This is because SIMD operations require the number of register-size elements to be at least even if the data length is shorter than the SIMD register size. In this case, the data are padded with a value such as zero, which reduces the valid vectorization ratio.

2.3.2 Image Data Structure

An image data structure is a 1D array of pixels in memory. Such pixels have color channel information, such as R, G, B, or the transparent channel A. The usual image structure is interleaved with color channel information. In the image data structure, data are not continuously arranged in spatial sampling because the other channel pixels intercept sequential access. Therefore, the image data structure should be transformed into SIMD-friendly structures. Some frequently used transformations are *split*, which converts a multi-channel image into a plurality of images for each channel, and *merge*, which converts a few images of each channel into one multi-channel image.

The transformed data structures correspond to structure of array (SoA) and array of structures (AoS) [68]. Figure 2.2 shows data arrangements in memory for each data structure. AoS is a default data structure for images. SoA is a data structure used in the split transformation for the AoS structure. When pixels are accessed at different positions in the same channel, the accessing cost in AoS is higher than that in SoA. However, for the access of



(a) Array of structure (AoS). (b) Structure of Array (SoA).

Figure 2.2: Image data structure.

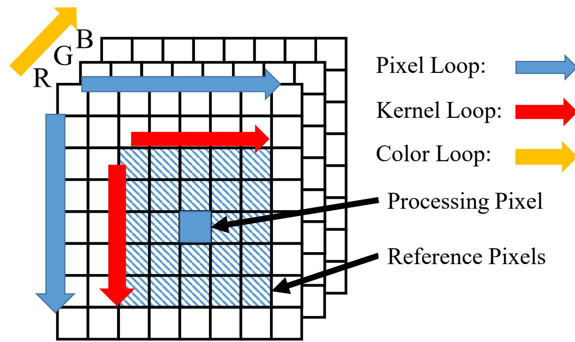


Figure 2.3: Loops in 2D finite impulse response filtering.

pixels at the same positions with different channels, the cost in AoS is lower than that in SoA. SoA is primarily used in vectorization programming. The size of a pixel including RGB is smaller than the size of the SIMD register. Thus, in vectorized programming, pixels are vectorized horizontally so that the size of the vectorized pixels is the same as the size of the SIMD register.

2.3.3 Vectorization of FIR Filtering

FIR image filtering contains five nested loops. There are three types of loops: loops for scanning image pixels, a kernel, and color channels. The loops for the image pixels and the kernel have four nested loops, which comprise loops for both pixel and kernel loops in the vertical and horizontal directions. Furthermore, when the filtering image has color channels, the processing for each channel is also regarded as a loop. Note that the length of the color loop is obviously shorter than the other loops. Figure 2.3 depicts the loops of the FIR filter, and Figure 2.4a indicates the code for general FIR filtering. To vectorize the code, loop unrolling to group pixels is necessary. Three types of loop unrolling are possible: pixel loop unrolling, kernel loop unrolling, and color loop unrolling. Here, we summarize each pattern as vectorization patterns of basic vectorized programming for 2D FIR filtering.

```
1   for(int y=0; y<img_height; y++){                          //pixel loop
2       for(int x=0; x<img_width; x++){
3           sum[channels] = {0};
4           weight_sum = 0;
5           for(int j=0; j<kernel_height; j++){               //kernel loop
6               for(int i=0; i<kernel_width; i++){
7                   temp_weight = calcWeight(j, i, y, x);
8                   for(int c=0; i<channels; c++){            //color loop
9                       sum[c] += temp_weight * I[y+j][x+i][c];
10                  }
11                  weight_sum += temp_weight;
12              }
13          }
14          for(int c=0;i<channels;c++){
15              D[y][x][c] = sum[c]/weight_sum;
16          }
17      }
18  }
```

(a) Brute-force implementation.

```
1   zeroPadding();
2   for(int y=0; y<img_height; y++){
3       for(int x=0; x<img_width; x++){
4           sum[4] = {0};
5           weight_sum = 0;
6           for(int j=0; j<kernel_height; j++){
7               for(int i=0; i<kernel_width; i++){
8                   temp_weight = calcWeight(j, i, y, x);
9                   sum[0] += temp_weight * I[y+j][x+i][0];
10                  sum[1] += temp_weight * I[y+j][x+i][1];
11                  sum[2] += temp_weight * I[y+j][x+i][2];
12                  sum[3] += temp_weight * I[y+i][x+i][3];// always 0
13                  weight_sum += temp_weight;
14              }
15          }
16          for(int c=0;i<channels;c++){
17              D[y][x][c] = sum[c]/weight_sum;
18          }
19      }
20  }
```

(b) Color loop unrolling.

```
1   convertSoA();
2   for(int y=0; y<img_height; y++){
3       for(int x=0; x<img_width; x++){
4           sum[channels] = {0};
5           weight_sum = 0;
6           temp_weight_sum[4] = {0;}
7           for(int j=0; j<kernel_height; j++){
8               for(int i=0; i<kernel_width; i+=4){
9                   temp_weight[4] = {0};
10                  temp_weight[0] = calcWeight(j, i+0, y, x);
11                  temp_weight[1] = calcWeight(j, i+1, y, x);
12                  temp_weight[2] = calcWeight(j, i+2, y, x);
13                  temp_weight[3] = calcWeight(j, i+3, y, x);
14                  for(int c=0; i<channels; c++){
15                      sum[c] += temp_weight[0] * I[c][y+j][x+i+0];
16                      sum[c] += temp_weight[1] * I[c][y+j][x+i+1];
17                      sum[c] += temp_weight[2] * I[c][y+j][x+i+2];
18                      sum[c] += temp_weight[3] * I[c][y+j][x+i+3];
19                  }
20                  temp_weight_sum[0] += temp_weight[0];
21                  temp_weight_sum[1] += temp_weight[1];
22                  temp_weight_sum[2] += temp_weight[2];
23                  temp_weight_sum[3] += temp_weight[3];
24              }
25              residual_processing();
26          }
27          weight_sum += temp_weight_sum[0];
28          weight_sum += temp_weight_sum[1];
29          weight_sum += temp_weight_sum[2];
30          weight_sum += temp_weight_sum[3];
31          for(int c=0;i<channels;c++){
32              D[y][x][c] = sum[c]/weight_sum;
33          }
34      }
35  }
```

(c) Kernel loop unrolling.

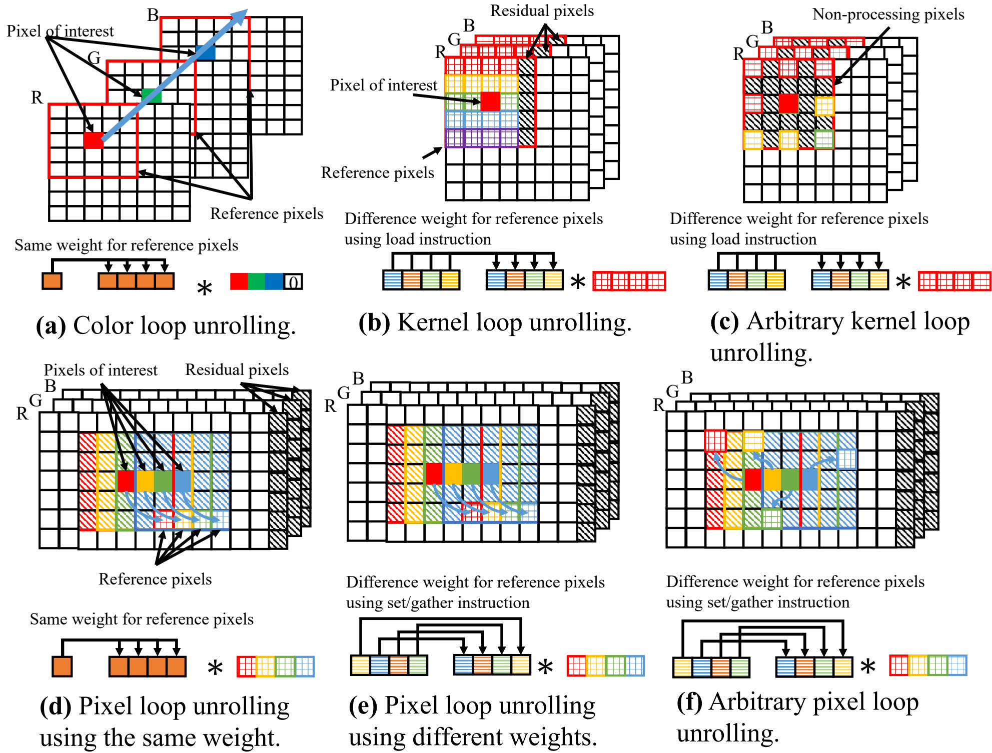
```
1   convertSoA();
2   for(int y=0; y<img_height; y++){
3       for(int x=0; x<img_width; x+=4){
4           sum[channels][4] = {0};
5           weight_sum[4] = {0};
6           for(int j=0; j<kernel_height; j++){
7               for(int i=0; i<kernel_width; i++){
8                   temp_weight[4] = {0};
9                   temp_weight[0] = calcWeight(j, i, y, x+0);
10                  temp_weight[1] = calcWeight(j, i, y, x+1);
11                  temp_weight[2] = calcWeight(j, i, y, x+2);
12                  temp_weight[3] = calcWeight(j, i, y, x+3);
13                  for(int c=0; i<channels; c++){
14                      sum[c][0] += temp_weight * I[c][y+j][x+i+0];
15                      sum[c][1] += temp_weight * I[c][y+j][x+i+1];
16                      sum[c][2] += temp_weight * I[c][y+j][x+i+2];
17                      sum[c][3] += temp_weight * I[c][y+j][x+i+3];
18                  }
19                  weight_sum[0] += temp_weight[0];
20                  weight_sum[1] += temp_weight[1];
21                  weight_sum[2] += temp_weight[2];
22                  weight_sum[3] += temp_weight[3];
23              }
24          }
25          residual_processing();
26          for(int c=0;i<channels;c++){
27              D[c][y][x+0] = sum[c][0]/weight_sum[0];
28              D[c][y][x+1] = sum[c][1]/weight_sum[1];
29              D[c][y][x+2] = sum[c][2]/weight_sum[2];
30              D[c][y][x+3] = sum[c][3]/weight_sum[3];
31          }
32      }
33  }
34  convertAoS();
```

(d) Pixel loop unrolling.

Figure 2.4: Code of vectorization patterns. The size of the SIMD register is 4. Usually, the data structure $I[y][x][c]$ represents RGB interleaving, where x and y are the horizontal and vertical positions, respectively, and c is the color channel. Splitting and merging the data by each channel are defined as follows: I[y][x][c] $\Leftrightarrow$ I[c][y][x]. For these data structures, the data in the final operator $[\cdot]$ can be sequential access.

2.3.4 Color Loop Unrolling

In color loop unrolling, color channels in a pixel are vectorized to compute each color channel in parallel. Figures 2.4b and 2.5a depict the code and vectorization approach to color loop unrolling. In this pattern, a pixel that includes all color channels requires a length of SIMD register size. Typi-



(a) Color loop unrolling. **(b)** Kernel loop unrolling. **(c)** Arbitrary kernel loop unrolling.

(d) Pixel loop unrolling using the same weight. **(e)** Pixel loop unrolling using different weights. **(f)** Arbitrary pixel loop unrolling.

Figure 2.5: Vectorization pattern of vectorized programming.

cally, the color represents three color channels, namely, R, B, and G. As it is known today, the SIMD register has 4 elements in SSE, 8 elements in AVX/AVX2, and 16 elements in AVX512 for the case of single-precision floating point numbers. Therefore, the size of a pixel, including all color channels, remains smaller than the size of SIMD register, and we require zero padding. Using zero padding, aligned data loading is always possible because every loading data address is aligned. However, this pattern decreases valid vectorization efficiency by the amount of zero padding. Kernel weight is scalar; thus, this pattern has no constraint in vector operations for weight handling. Since vectorization is performed for color channels in all pixels, the pattern has no constraints in image size and kernel shape.

2.3.5 Kernel Loop Unrolling

In kernel loop unrolling, reference pixels in a kernel are vectorized to calculate kernel convolution processing for a pixel of interest in parallel. Figures 2.4c and 2.5b indicate the code and vectorization approach to kernel loop unrolling. If the kernel width, which depends on parameters, is a multiple

of the size of the SIMD register, reference pixels are able to be efficiently loaded into the SIMD register. However, in most cases, kernel width is not a multiple of the SIMD register size. In such a case, residual processing is necessary for residual reference pixels, which generally occur at the lateral edge of the kernel. The set/gather instruction or scalar operations, which do not assume sequential access, are used for residual processing. The kernel loop steps incrementally; thus, the loading memory address must cross unaligned addresses. In this pattern, weights are vectorially calculated by reference pixels. If the kernel weight depends only on the position relative to the pixel of interest, it is possible to efficiently load the weight into the SIMD register. Because only reference pixels are vectorized, no restriction exists on image size. In this pattern, reference pixels are required to have continuous loading; thus, kernel shape is constrained. Kernel subsampling, where reference pixels are discontinuous, cannot use the pattern (see Figure 2.5c). The set/gather instruction is used for kernel subsampling. We call kernel loop unrolling with the set/gather instruction arbitrary kernel loop unrolling. Arbitrary kernel loop unrolling is slower than kernel loop unrolling because the set/gather instruction is inefficient.

This pattern can be achieved on SoA, but the image data structure is usually AoS. Therefore, color channel splitting must be performed before processing. Output data structure should be AoS, and this pattern outputs scalar data; thus, the data are stored with scalar instructions. The pattern has more constraints than color loop unrolling, but the vectorization efficiency of the pattern is significantly better than that of color loop unrolling.

2.3.6 Pixel Loop Unrolling

In pixel loop unrolling, pixels of interest and reference pixels are vectorized to calculate in parallel the multiplicity of kernel convolutions for multiple pixels of interest. We realize the processing by extending the kernel convolution processing as vector operations between pixels of interest and reference pixels. Figures 2.4d and 2.5d depict the code and vectorization approach to pixel loop unrolling. If image width is a multiple of the SIMD register size, the pixels of interest and reference pixels are efficiency loaded. If image width is not a multiple of SIMD register size, residual processing is required at the lateral edge of the image. For residual processing, the image is padded so that its width is a multiple of the SIMD register size, or the set/gather instruction or the scalar operation is executed similarly to kernel loop unrolling. Access to the reference pixels is incrementally stepping, as is kernel loop unrolling; thus, loading memory address must cross unaligned addresses. Kernel weight must be calculated for each reference pixel among vectorized elements. How-

ever, if the calculated weight depends only on relative position, it must be the same for each reference pixel in a vector. This is because the relative positions of the pixel of interest and the reference pixel are the same in the vector (see Figure 2.5d). There are no restrictions for the pattern in kernel width because the calculation for the pixel of interest and the reference pixel is vectorized. If the kernel shapes remain identical in all pixels of interest, the pattern can be used. However, if the kernel shapes are variant for each pixel of interest in a vector, the pattern cannot be used. Such condition is filtering with random kernel subsampling and adaptive spatial kernel filtering (see Figure 2.5f). In these filters, the relative positions of the pixel of interest and the reference pixel are not the same in a vector; thus, the access for the reference pixels is not continuous. The set/gather instruction resolves this discontinuous issues. We call this pattern arbitrary pixel loop unrolling. This pattern can accommodate different kernel shapes. The kernel size of this pattern must be adjusted to the largest number of reference pixels. Arbitrary pixel loop unrolling is slower than pixel loop unrolling as with the case of arbitrary kernel loop unrolling. If the weights are not the same for all reference pixels in a vector, the result is more expensive than using the same weight for all (see Figure 2.5e).

Before filtering in this pattern, we perform color channel splitting to adjust the data layout to acquire horizontally sequential data. The output of this pattern is SoA, and a usual image should be AoS; thus, postprocessing is needed for AoS conversion. In this pattern, parallel computing is used in the outermost loop; the vectorization of that loop is highly efficient. However, this pattern has greater constraints than kernel loop unrolling.

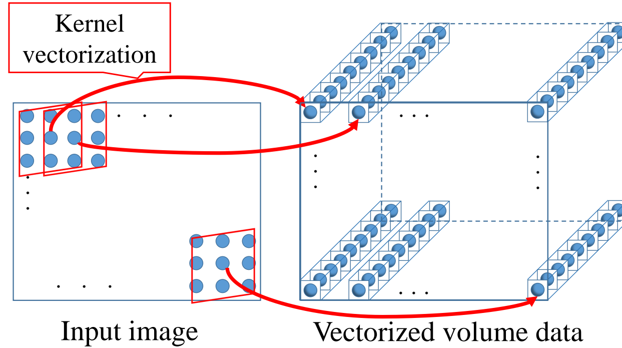
2.4 Proposed Design Pattern of Vectorization

In this section, we propose a new vectorization pattern in FIR image filtering with kernel subsampling, which we call *loop vectorization*. We summarize characteristics of the previous vectorization patterns and our new one in Table 2.1. The proposed pattern encounters none of the constraints that exist in the previous patterns.

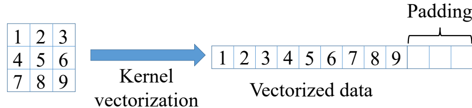
In this proposed pattern, reference pixels are extracted in the kernel and the pixels are grouped as a 1D vector. This vector must be multiple times as long as the SIMD register size. To adjust the length of the vector, extra data are padded with zero. We collect the vector for all pixels and construct volume data using the vectors. This rearrangement scheme is called loop vectorization. Figure 2.6 indicates an example of loop vectorization for kernel loop, which is called kernel loop vectorization. As a prepossessing for filtering,

Table 2.1: Characteristics of the vectorization patterns of vectorization in finite impulse response image filtering.

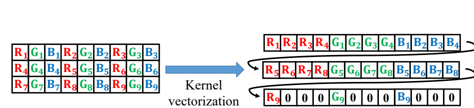
Vectorization Pattern	Arbitrary Parameter/Non-Limitation	Restriction Parameter/Limitation
loop vectorization	image width, kernel width, kernel shape, aligned load	long preprocessing time, huge memory usage
color loop unrolling	image width, kernel width, kernel shape, aligned load	requiring color image with padding
kernel loop unrolling	image width	kernel width, kernel shape, non-aligned load
arbitrary kernel loop unrolling	image width, kernel shape	kernel width, inefficient load, non-aligned load
pixel loop unrolling	kernel width	image width, kernel shape, non-aligned load
arbitrary pixel loop unrolling	kernel width, kernel shape	image width, inefficient load, non-aligned load



(a) Rearrange approach.



(b) Data structure of a pixel in gray image.



(c) Data structure of a pixel in color image.

Figure 2.6: Kernel vectorization. The size of the SIMD register is 4.

we transform an input image into volume data by loop vectorization. Let the number of elements in the kernel be K and the size of the image be S. The volume data size is KS. Figure 2.7 depicts the code of proposed pattern. The pattern on color images is shown in Figure 2.6c. This pattern interleaves individual R, G, and B vectors whose length is the size of an SIMD register. Zero paddings in kernel loops are required for each color channel. The proposed pattern has a data structure that is the array of structure of array (AoSoA) [68]. AoSoA is preferable for contiguous memory access, and its data structure has a high spatial locality in memory. Therefore, AoSoA has the greater efficiency than SoA and AoS in memory prefetching.

The FIR filtering is related convolutional neural network (CNN) [69] based deep learning. The proposed pattern is similar approach to convolution lowering (*im2col*) [70–72], which is CNN acceleration method. In the proposed pattern, we convert it to a data structure specialized for vector

```
1   loop_vectorization_for_kernel_loop();
2   for(int y=0; y<img_height; y++){
3       for(int x=0; x<img_width; x++){
4           sum[channels] = {0};
5           weight_sum = 0;
6           temp_weight_sum[4] = {0;}
7           for(int j=0; j<kernel_height; j++){
8               for(int i=0; i<kernel_width; i+=4){
9                   temp_weight[4] = {0};
10                  temp_weight[0] = calcWeight(j, i+0, y, x);
11                  temp_weight[1] = calcWeight(j, i+1, y, x);
12                  temp_weight[2] = calcWeight(j, i+2, y, x);
13                  temp_weight[3] = calcWeight(j, i+3, y, x);
14                  for(int c=0; i<channels; c++){
15                      sum[c] += temp_weight[0] * LV[y][x][j][c][i+0];
16                      sum[c] += temp_weight[1] * LV[y][x][j][c][i+1];
17                      sum[c] += temp_weight[2] * LV[y][x][j][c][i+2];
18                      sum[c] += temp_weight[3] * LV[y][x][j][c][i+3];
19                  }
20                  temp_weight_sum[0] += temp_weight[0];
21                  temp_weight_sum[1] += temp_weight[1];
22                  temp_weight_sum[2] += temp_weight[2];
23                  temp_weight_sum[3] += temp_weight[3];
24              }
25          }
26          weight_sum += temp_weight_sum[0];
27          weight_sum += temp_weight_sum[1];
28          weight_sum += temp_weight_sum[2];
29          weight_sum += temp_weight_sum[3];
30          for(int c=0;i<channels;c++){
31              D[y][x][c] = sum[c]/weight_sum;
32          }
33      }
34  }
```

(a) Loop vectorization for kernel loop.

```
1   loop_vectorization_for_pixel_loop();
2   for(int y=0; y<img_height; y++){
3       for(int x=0; x<img_width; x+=4){
4           sum[channels][4] = {0};
5           weight_sum[4] = {0};
6           for(int j=0; j<kernel_height; j++){
7               for(int i=0; i<kernel_width; i++){
8                   temp_weight[4] = {0};
9                   temp_weight[0] = calcWeight(j, i, y, x+0);
10                  temp_weight[1] = calcWeight(j, i, y, x+1);
11                  temp_weight[2] = calcWeight(j, i, y, x+2);
12                  temp_weight[3] = calcWeight(j, i, y, x+3);
13                  for(int c=0; i<channels; c++){
14                      sum[c][0] += temp_weight * LV[y][j][i][c][x+0];
15                      sum[c][1] += temp_weight * LV[y][j][i][c][x+1];
16                      sum[c][2] += temp_weight * LV[y][j][i][c][x+2];
17                      sum[c][3] += temp_weight * LV[y][j][i][c][x+3];
18                  }
19                  weight_sum[0] += temp_weight[0];
20                  weight_sum[1] += temp_weight[1];
21                  weight_sum[2] += temp_weight[2];
22                  weight_sum[3] += temp_weight[3];
23              }
24          }
25          for(int c=0;i<channels;c++){
26              D[c][y][x+0] = sum[c][0]/weight_sum[0];
27              D[c][y][x+1] = sum[c][1]/weight_sum[1];
28              D[c][y][x+2] = sum[c][2]/weight_sum[2];
29              D[c][y][x+3] = sum[c][3]/weight_sum[3];
30          }
31      }
32  }
```

(b) Loop vectorization for pixel loop.

Figure 2.7: Code of loop vectorization. The size of the SIMD register is 4. LV represents the data structure transformed by loop vectorization. For the data structure, the data in the final operator [·] can be sequential access. The data structure is always accessed sequentially.

operation in CPU by considering data alignment and data arrangement of the color channel. In addition, parallelization efficiency is improved by the proposed pattern for the pixel loop. Therefore, the proposed pattern can also be effective for CNN-based deep learning in CPU.

The proposed pattern can also vectorize pixel loop. In the proposed pattern of pixel loop vectorization, a vector is created with the accessed pixels through pixel loop unrolling; thus, a vector is created in the units of the pixels of interest to be unrolled. Pixel loop vectorization is highly parallelization efficient because it parallelizes the outermost loop as well as the case of pixel loop unrolling. However, if the filtering parameters are different per each kernel, pixel loop vectorization requires the set instruction for the different parameters for each pixel of interest. The limitations of this pattern are the same as those of pixel loop unrolling.

The advantages of the proposed pattern include the fact that, unlike other patterns, these patterns are not restricted by the image width, kernel width, and kernel shape. In addition, data alignment will clearly be consistent in any conditions. The disadvantage is that the proposed pattern requires huge memory capacity. Kernel subsampling, however, moderates the memory

footprint of loop vectorization. Furthermore, the proposed pattern is particularly effective in its use of kernel subsampling, because memory accesses of the other patterns are not sequential in filtering with kernel subsampling but those of the proposed pattern are sequential. In random subsampling, performance will be more outstanding. The proposed pattern is also effective for cases where kernel radius or image size is large. In such conditions, cache-missing frequently occurs in the other patterns.

A limitation of the proposed pattern is the rearrangement processing is overhead. However, the proposed pattern is practical for certain applications, such as image editing, where the same image is processed multiple times. In image editing, rearrangement is only performed when the process begins. In this application, a user interactively changes parameters and repeats filtering several times to seek more desirable results. In interactive filtering, the overhead caused by the rearrangement may be simply a waiting time for interactive photo editing to begin. The characteristics of interactive filtering can also be utilized in feature extraction of scale-space filtering, e.g., SIFT [73].

2.5 Material and Methods

We here vectorize six filtering algorithms, namely, the Gaussian range filter (GRF), the bilateral filter (BF) [1], the adaptive Gaussian filter (AGF) [74], the randomly-kernel-subsampled Gaussian range filter (RKS-GRF), the randomly-kernel-subsampled bilateral filter (RKS-BF) [56], and the randomly-kernel-subsampled adaptive Gaussian filter (RKS-AGF). The main characteristics of these filters are summarized in Table 2.2. Note that the BF has various acceleration algorithms [8, 27–30], but we select a naïve BF to cover types of the general FIR filter. In this chapter, we deal with two types of implementation of these filters: the calculating weights with SIMD instructions and calculated weights with lookup tables (LUTs). The kinds of implementation differ in their characteristics. In calculating weights, it is possible to focus on data loading, and in using LUTs, it focuses on the case of optimal implementation.

2.5.1 Gaussian Range Filter

The weight of the GRF is defined as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) := \exp\left(\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}\right), \tag{2.3}$$

Table 2.2: Characteristics of the Gaussian range filter (GRF), the bilateral filter (BF), the adaptive Gaussian filter (AGF), the randomly-kernel-subsampled Gaussian range filter (RKS-GRF), the randomly-kernel-subsampled bilateral filter (RKS-BF), and the randomly-kernel-subsampled adaptive Gaussian filter (RKS-AGF).

Filter	Weight Depending	LUT	Kernel Shape
GRF	pixel value	range	invariant
BF	pixel value, pixel position	space, range	invariant
AGF	parameter map, pixel position	space	variant
RKS-GRF	pixel value	range	variant
RKS-BF	pixel value, pixel position	space, range	variant
RKS-AGF	parameter map, pixel position	space	variant

where $\| \cdot \|_2$ is the L2 norm, and σ_r is a standard deviation.

The weight depends on the intensities of the nearest pixels. The values of the nearest pixels are different; thus, the LUT of the Gaussian range weight is discontinuously accessed for each pixel differential. In direct weight computation, the vectorized exponential operation is not including in the SIMD instruction set, although the Intel compiler extendedly provides the vectorized exponential operation. Hence, we use the Intel compiler.

2.5.2 Bilateral Filter

The BF is a representative filter of edge-preserving filtering. The weight of the BF is denoted in the following way:

$$f(\boldsymbol{p}, \boldsymbol{q}) \quad := \quad \exp(\frac{\|\boldsymbol{p} - \boldsymbol{q}\|_2^2}{-2\sigma_s^2}) \exp(\frac{\|\boldsymbol{I}(\boldsymbol{p}) - \boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}), \tag{2.4}$$

where σ_s and σ_r are the standard deviations for the space and range kernels, respectively.

The weight can be decomposed into spatial and range weight. The spatial weight, which is the first exponential function, matches the weight in Gaussian filtering. The range weight, which is the second exponential function, matches the weight in the GRF. The LUT of the space weights is continuously accessed because relative positions of the reference pixels are continuous. On the other hand, the LUT of the range weight is not continuous as with the GRF.

2.5.3 Adaptive Gaussian Filter

The AGF operates in a slightly different manner from Gaussian filtering. The standard deviation dynamically changes, pixel by pixel. The weight of the AGF is defined as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) \ := \ \exp(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s(\boldsymbol{p})^2}), \tag{2.5}$$

where $\sigma_s(\boldsymbol{p})$ is a pixel-dependent parameter found in a parameter map.

Here, we use this filter for refocusing. In this application, we change the parameter of the Gaussian distribution using a depth map [75, 76] as the parameter map. The detail of the AGF based on the depth map is defined as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) \ := \ \exp(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2(\sigma_s+\alpha|d-\boldsymbol{D}(\boldsymbol{p})|)^2}), \tag{2.6}$$

where $\boldsymbol{D}$ is the depth map, d is the focusing depth value, and α is a parameter of the range of the depth of field. The function of the kernel shape $\mathcal{N}(\boldsymbol{p})$ in Equations (2.1) and (2.2) is different for each pixel of interest $\boldsymbol{p}$.

Blurring is minimal at the focused pixel, and most of the kernel weights may become zero. In this case, the region whose kernel weights are not zero can be regarded as an arbitrary kernel depending on r', which is less than the actual r, due to the property of the Gaussian distribution. This means that, if the processing pixel of interest is in focus, we can only process a small kernel depending on r'. Pixel loop vectorization, pixel loop unrolling, and kernel loop unrolling are restricted in terms of the kernel shape function. Therefore, the largest kernels in a vector of the pixel of interest should be used to maintain the restriction. Further, within kernel loop vectorization, arbitrary kernel loop unrolling, and color loop unrolling, the amount of processing can be reduced using small kernels.

In the AGF, multiple LUTs are prepared to compute kernel weights whose size is $\mathcal{D}\times(2r+1)\times(2r+1)$. $\mathcal{D}$ is the number of elements in the depth range. The utilized LUT is switched by the depth value. In kernel loop vectorization, kernel loop, and color loop unrolling, the LUT is sequentially accessed within a single LUT. In pixel loop vectorization and pixel loop unrolling, the LUT is instead discontinuously accessed across multiple LUTs.

2.5.4 Randomly-Kernel-Subsampled Filter

The randomly-kernel-subsampled filter is an approximation of FIR filtering. This filter uses a different kernel shape function $\mathcal{N}(\boldsymbol{p})$ to randomly subsample

pixels in a kernel. The kernel shape functions $\mathcal{N}(\boldsymbol{p})$ of the GRF, BF, and AGF return permanent positions. The kernel functions $\mathcal{N}(\boldsymbol{p})$ of the RKS-GRF, RKS-BF, and RKS-AGF return variable positions for a pixel-by-pixel $\boldsymbol{p}$. The RKS-GRF, RKS-BF, and RKS-AGF are represented as follows:

$$\bar{\boldsymbol{I}}(\boldsymbol{p}) \simeq \bar{\boldsymbol{I}}'(\boldsymbol{p}) = \frac{\sum_{j=1}^{n} f(\boldsymbol{p}, \boldsymbol{R}_j(\boldsymbol{p})) I(\boldsymbol{R}_j(\boldsymbol{p}))}{\sum_{j=1}^{n} f(\boldsymbol{p}, \boldsymbol{R}_j(\boldsymbol{p}))}, \tag{2.7}$$

where $n = |\mathcal{N}(\boldsymbol{p})|$ denotes the number of samples, and $\boldsymbol{R}_j(\boldsymbol{p})$ randomly returns the positions of support pixels around $\boldsymbol{p}$. $\boldsymbol{R}$ similarly works for kernel subsampling. Note that $\mathcal{N}(\boldsymbol{p})$ can be decomposed into $\boldsymbol{R}_j(\boldsymbol{p})$ and the partial summation operation $\sum_{j=1}^{n}$ in the randomly-kernel-subsampled filter.

2.6 Experimental Results

We verified all the vectorization patterns and proposed vectorization pattern using kernel subsampling for the GRF, BF, AGF, RKS-GRF, RKS-BF, and RKS-AGF. Further, we compared the two types of proposed loop vectorization, which were kernel loop vectorization and pixel loop vectorization, with pixel loop, kernel loop, and color loop unrolling. Importantly, arbitrary kernel loop unrolling was used instead of kernel loop unrolling in kernel subsampling and randomly-kernel-subsampling conditions. Further, arbitrary pixel loop unrolling was used in the place of pixel loop unrolling in randomly-kernel-subsampled filters. This step was taken because kernel loop unrolling cannot be used in (randomly) kernel subsampling, and pixel loop unrolling cannot be used in randomly-kernel-subsampled filters. These filters were implemented in C++ using AVX2 and FMA instructions as SIMD instruction sets. Additionally, multi-core parallelization was used with concurrency. The CPU used was an Intel Core-i7 6850X 3.0 GHz, and the memory used was DDR4 16 GBytes. Windows 10 64 bits was used for the OS, and Intel Compiler 18.0 was employed as the compiler. The experimental code reached around 100,000 lines.

Figures 2.8–2.20 indicate the processing time and speedup ratio for each filter. The time for computation is judged to be the median value of 100 trials. In addition, the time for computation does not include rearrangement time in all patterns because we focus on interactive filters. The speedup ratio relates the kernel loop vectorization vs. another pattern. If the speedup ratio exceeds 1, the other pattern is faster than kernel loop vectorization. Figures 2.8 and 2.9 show the results of the GRF. The computational times for the two types of the proposed pattern are almost the same as those for pixel loop unrolling, and such patterns are faster than other patterns. In the

two types of proposed pattern, the non-aligned load does not occur, in contrast with other patterns; hence, the two types of the proposed pattern are fast. Pixel loop unrolling has the most cache-efficiency because the locality of the input image is high. Cache-missing errors, however, occur in the large kernel radius and/or large images cases because of the gaps in the discontinuous access due to memory increase. Kernel loop unrolling is less rapid in conditions of kernel subsampling because arbitrary kernel loop unrolling is present in kernel-subsampling conditions. Color loop unrolling is the slowest pattern.

Figures 2.10 and 2.11 indicate the results for the BF. The BF has a Gaussian spatial kernel added to the GRF's kernel. The accessing pattern to the spatial kernel is sequential for all vectorization patterns; for this reason, the spatial kernel does not dramatically change the efficiency of all patterns in this filter. Therefore, the BF results follow almost the same trend as the GRF results.

Figures 2.12 and 2.13 indicate the results of the AGF. In the case of weight computation, the figures indicate that kernel loop vectorization is the fastest pattern. If LUTs are used, kernel loop vectorization is the fastest when r is large. When r is small, pixel loop unrolling is the fastest. Where LUTs are used, the implementation of pixel loop unrolling is efficient. However, where r is large, cache-missing occurs and the speed decreases.

Figures 2.14 and 2.15 present the results of the RKS-GRF. The two types of the proposed pattern have the greatest speed of all the patterns. The two types of the proposed pattern continuously access reference pixels. However, other vectorization patterns cannot continuously access reference pixels. In particular, pixel loop unrolling is the most affected by this.

Figures 2.16 and 2.17 present the results of the RKS-BF. Kernel loop vectorization is the fastest pattern. Pixel loop vectorization is slower than kernel loop vectorization. Pixel loop vectorization and pixel loop unrolling discontinuously access the spatial LUTs. Kernel loop vectorization and kernel loop unrolling continuously access spatial LUT.

Figures 2.18 and 2.19 present the results of the RKS-AGF. These figures indicate that the fastest pattern is kernel loop vectorization. The kernel loop vectorization continuously accesses reference pixels and LUTs. Together, these results indicate a special efficiency for proposed kernel loop vectorization in the kernel-adaptive sampling technique.

Figure 2.20 depicts the accuracies of GRF, BF, AGF, RKS-GRF, RKS-BF, and RKS-AGF. Here, we compare the results of a full sampling with the results of a subsampling using peak signal noise ratio (PSNR). In this figure, PSNR is found to be around 40 dB for all cases; thus, the filtered image has sufficient accuracy of approximation for all filters.

Offset computing time for loop vectorization in data structure transformation is discussed. Figure 2.21 presents the processing time required for loop vectorization. Processing time increases with increases in kernel radius and image size. Computing time is linearly proportional to the number of elements in the loop-vectorized data, namely, $(2r+1)^2 \times s$, where the kernel radius is r and the image size is s. However, for interactive filtering, this drawback can be neglected. Rearranged processing time must occur before filtering can be done at first in the proposed pattern, but it is not required for subsequent filtering. Therefore, rearrangement of processing time does not lead to significant problems in interactive filtering. If images show continuous change, such as in video, the second-fastest pattern should be used instead of the proposed one. The pattern that is proposed requires rearrangement for every image.

2.7 Conclusions

In this chapter, we summarize a taxonomy of vectorized programming for FIR image filtering. We also propose a new vectorization pattern of vectorized programming, which we call loop vectorization. These vectorization patterns are combined with an acceleration method of kernel subsampling for general FIR filters. The experimental results indicate that the patterns are appropriate for FIR filtering, and a new pattern with kernel subsampling can be profitably used for Gaussian range filtering (GRF), bilateral filtering (BF), adaptive Gaussian filtering (AGF), randomly-kernel-subsampled Gaussian range filtering (RKS-GRF), randomly-kernel-subsampled bilateral filtering (RKS-BF), and randomly-kernel-subsampled adaptive Gaussian filtering (RKS-AGF).

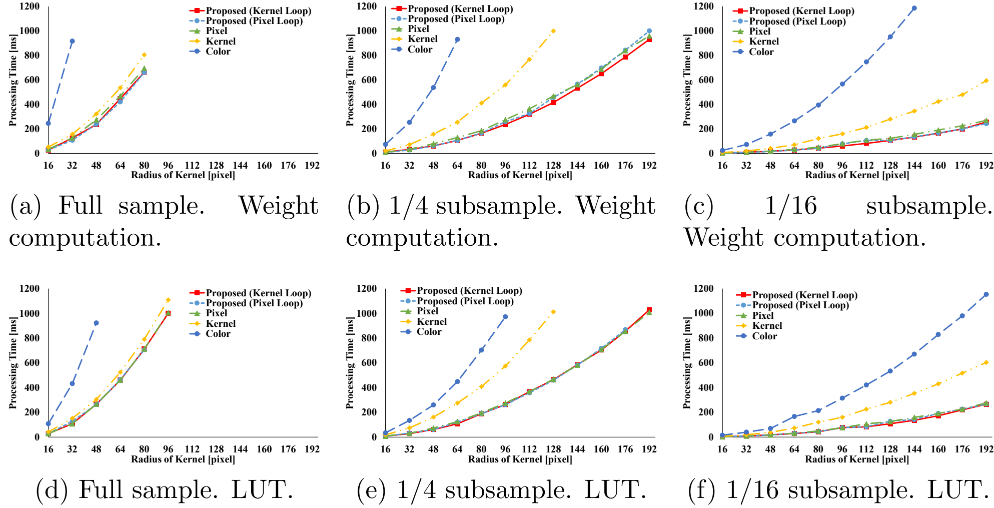
The results are summarized as follows:

1. The two types of the proposed pattern, which are kernel loop vectorization and pixel loop vectorization, are both effective for adaptive kernel shapes, that is, randomized filters and the AGF.

2. There remains, however, a trade-off in weight and data loading for changing spatial LUTs in each filtering pixel. Kernel loop unrolling is more suitable for weight loading, and loop vectorization is more suitable for data loading. Kernel loop vectorization is effective for weight and data loading; thus, the kernel loop vectorization is suitable for AGF, RKS-AGF, and RKS-BF.

3. For the large-radius condition, the two types of the proposed pattern

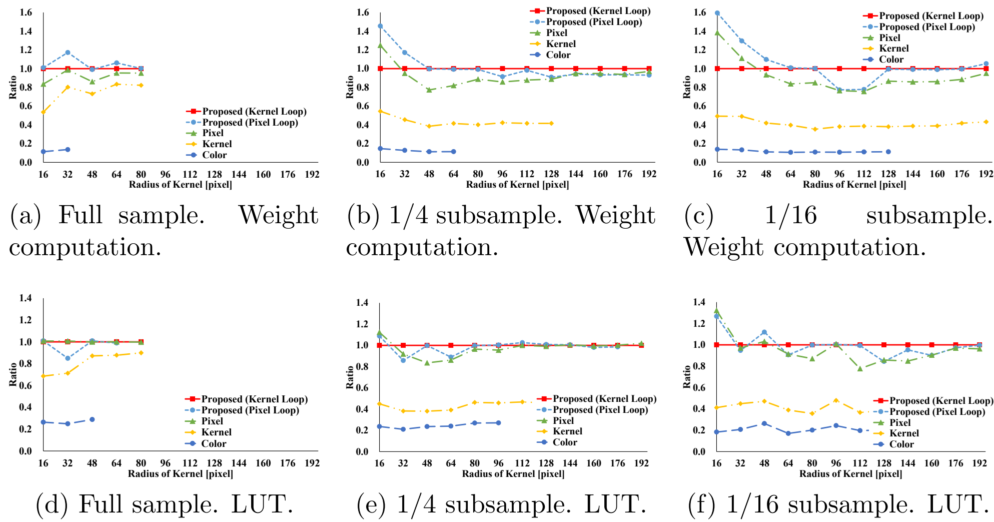
have moderate effectivity for other filters in the above effective cases, that is, the GRF and BF.



(a) Full sample. Weight computation. (b) 1/4 subsample. Weight computation. (c) 1/16 subsample. Weight computation.

(d) Full sample. LUT. (e) 1/4 subsample. LUT. (f) 1/16 subsample. LUT.

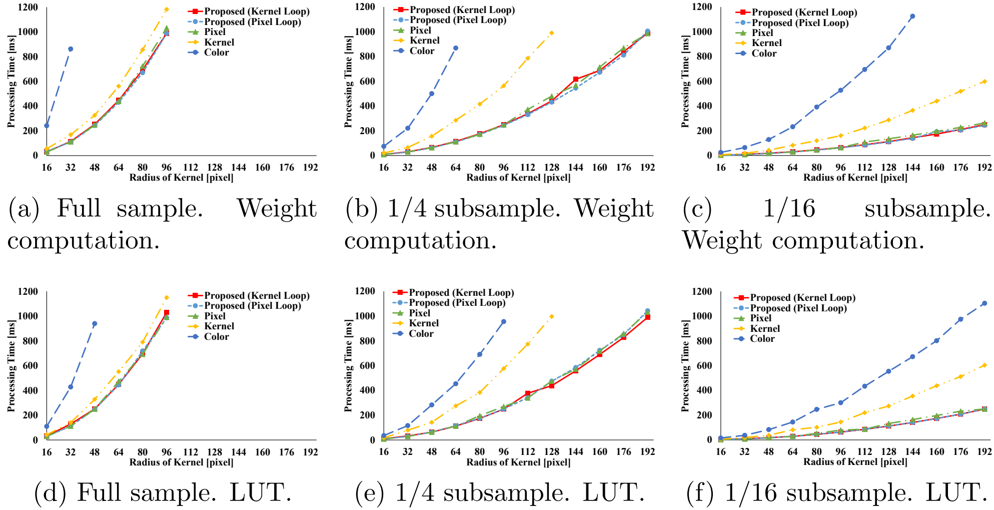
Figure 2.8: Processing time for Gaussian range filtering (GRF) with respect to the kernel radius of FIR filtering. Note that arbitrary kernel loop unrolling is used instead of kernel loop unrolling under kernel-subsampling conditions.



(a) Full sample. Weight computation. (b) 1/4 subsample. Weight computation. (c) 1/16 subsample. Weight computation.

(d) Full sample. LUT. (e) 1/4 subsample. LUT. (f) 1/16 subsample. LUT.

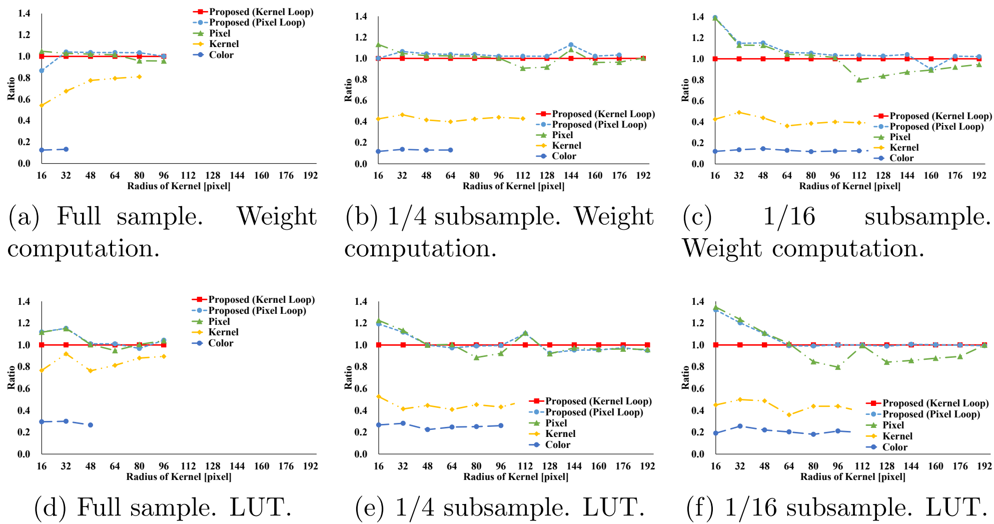
Figure 2.9: The speedup ratio for Gaussian range filtering (GRF) with respect to the kernel radius of FIR filtering. If the ratio exceeds 1, the given pattern is faster than the kernel loop vectorization.



(a) Full sample. Weight computation. (b) 1/4 subsample. Weight computation. (c) 1/16 subsample. Weight computation.

(d) Full sample. LUT. (e) 1/4 subsample. LUT. (f) 1/16 subsample. LUT.

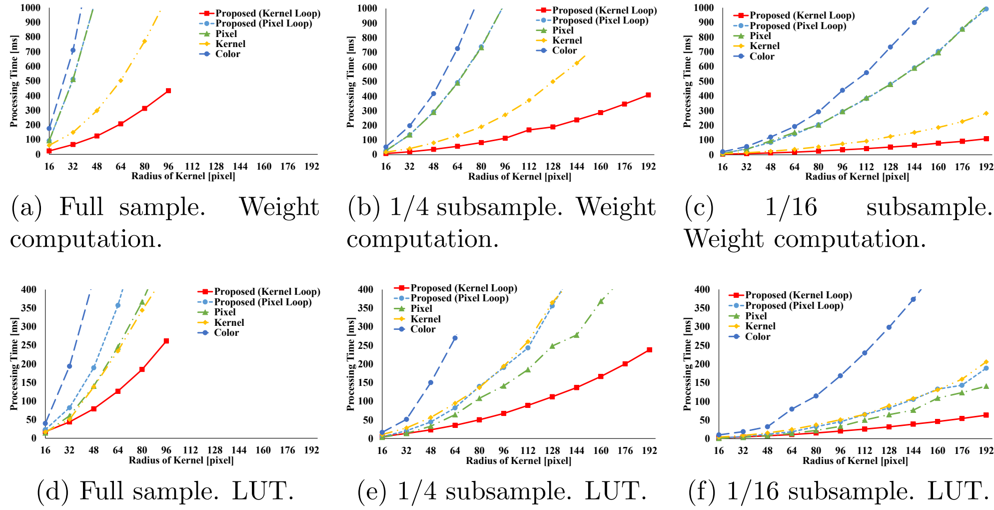
Figure 2.10: Processing time for bilateral filtering (BF) with respect to the kernel radius of FIR filtering. Note that arbitrary kernel loop unrolling is used instead of kernel loop unrolling in kernel-subsampling conditions.



(a) Full sample. Weight computation. (b) 1/4 subsample. Weight computation. (c) 1/16 subsample. Weight computation.

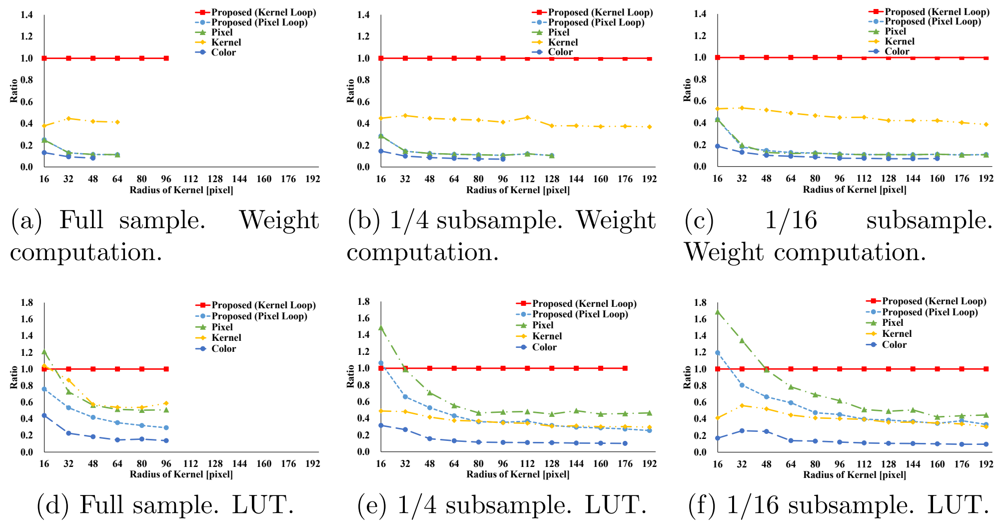
(d) Full sample. LUT. (e) 1/4 subsample. LUT. (f) 1/16 subsample. LUT.

Figure 2.11: The speedup ratio of bilateral filtering (BF) with respect to the kernel radius of FIR filtering. If the ratio exceeds 1, the given pattern is faster than the kernel loop vectorization.



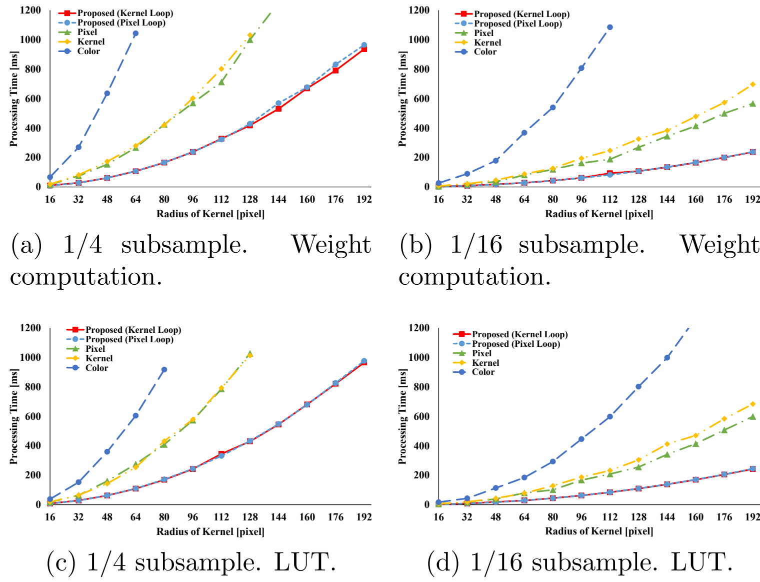
(a) Full sample. Weight computation.
(b) 1/4 subsample. Weight computation.
(c) 1/16 subsample. Weight computation.
(d) Full sample. LUT.
(e) 1/4 subsample. LUT.
(f) 1/16 subsample. LUT.

Figure 2.12: Processing time for adaptive Gaussian filtering (AGF) with respect to the kernel radius of FIR filtering. Note that arbitrary kernel loop unrolling is used instead of kernel loop unrolling in the kernel-subsampling conditions.



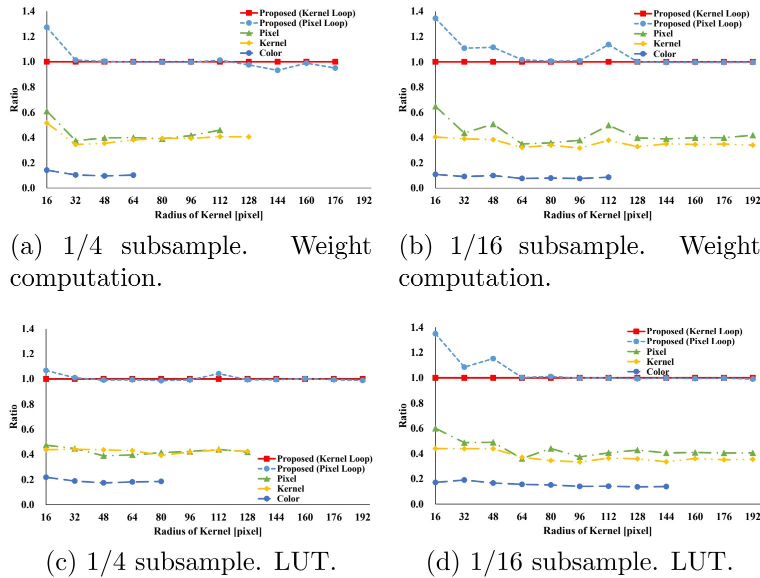
(a) Full sample. Weight computation.
(b) 1/4 subsample. Weight computation.
(c) 1/16 subsample. Weight computation.
(d) Full sample. LUT.
(e) 1/4 subsample. LUT.
(f) 1/16 subsample. LUT.

Figure 2.13: The speedup ratio for adaptive Gaussian filtering (AGF) with respect to kernel radius of FIR filtering. If the ratio exceeds 1, this pattern is faster than the kernel loop vectorization.



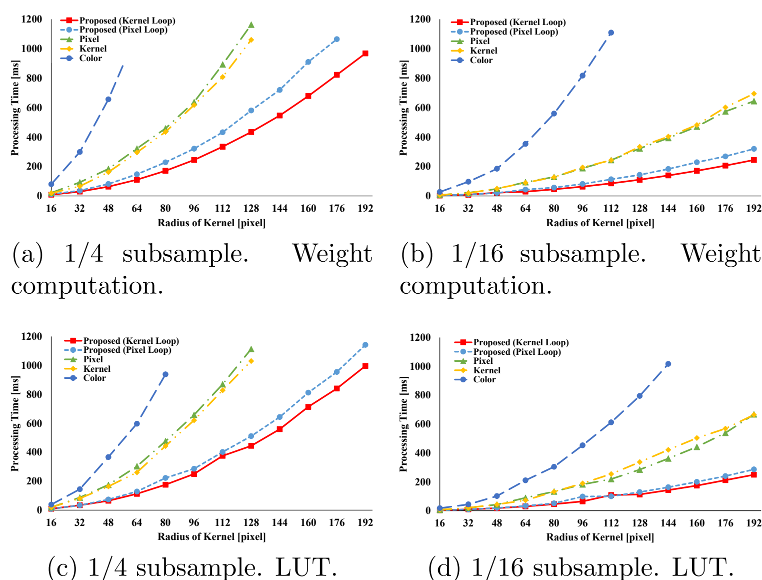
(a) 1/4 subsample. Weight computation.
(b) 1/16 subsample. Weight computation.
(c) 1/4 subsample. LUT.
(d) 1/16 subsample. LUT.

Figure 2.14: Processing time for randomly-kernel-subsampled Gaussian range filtering (RKS-GRF) with respect to the kernel radius of FIR filtering. Arbitrary pixel loop unrolling and arbitrary kernel loop unrolling are used in the place of pixel loop unrolling and kernel loop unrolling, respectively.



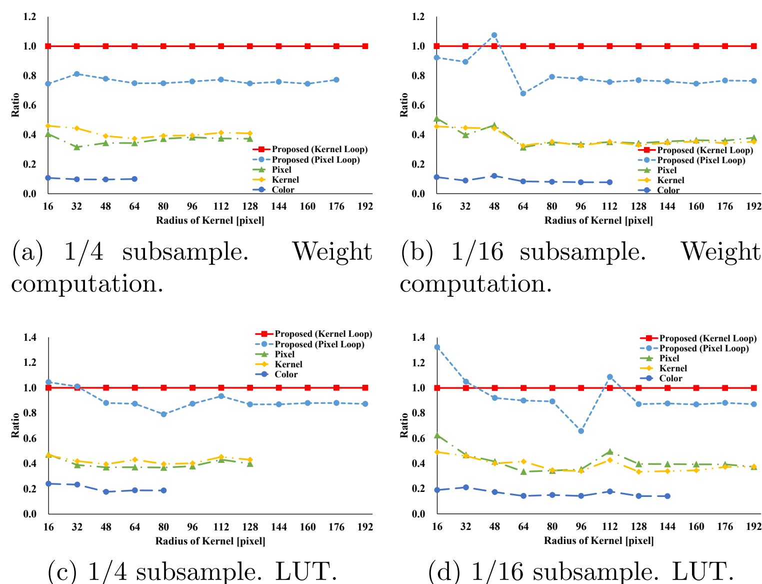
(a) 1/4 subsample. Weight computation.
(b) 1/16 subsample. Weight computation.
(c) 1/4 subsample. LUT.
(d) 1/16 subsample. LUT.

Figure 2.15: The speedup ratio of randomly-kernel-subsampled Gaussian range filtering (RKS-GRF) with respect to the kernel radius of FIR filtering. If the ratio exceeds 1, the pattern is faster than kernel loop vectorization.



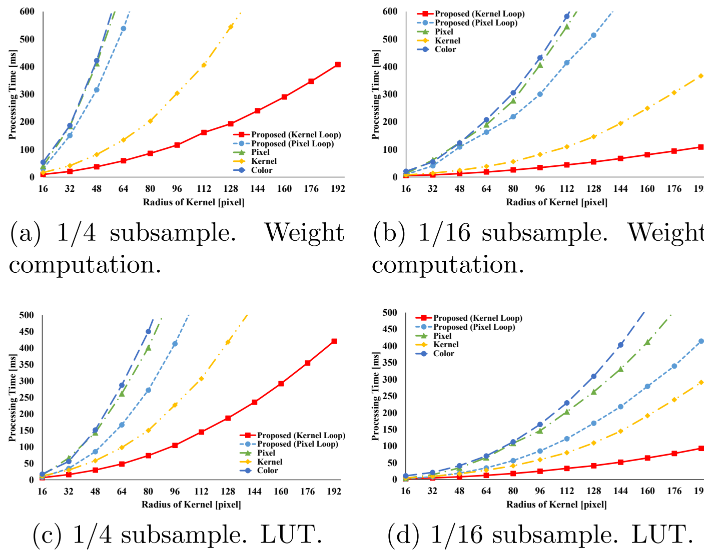
(a) 1/4 subsample. Weight computation.
(b) 1/16 subsample. Weight computation.
(c) 1/4 subsample. LUT.
(d) 1/16 subsample. LUT.

Figure 2.16: Processing time for randomly-kernel-subsampled bilateral filtering (RKS-BF) with respect to the kernel radius of FIR filtering. Arbitrary pixel loop unrolling and arbitrary kernel loop unrolling are used in place of pixel loop unrolling and kernel loop unrolling, respectively.



(a) 1/4 subsample. Weight computation.
(b) 1/16 subsample. Weight computation.
(c) 1/4 subsample. LUT.
(d) 1/16 subsample. LUT.

Figure 2.17: The speedup ratio of randomly-kernel-subsampled bilateral filtering (RKS-BF) with respect to the kernel radius of FIR filtering. If the ratio exceeds 1, the given pattern is faster than kernel loop vectorization.



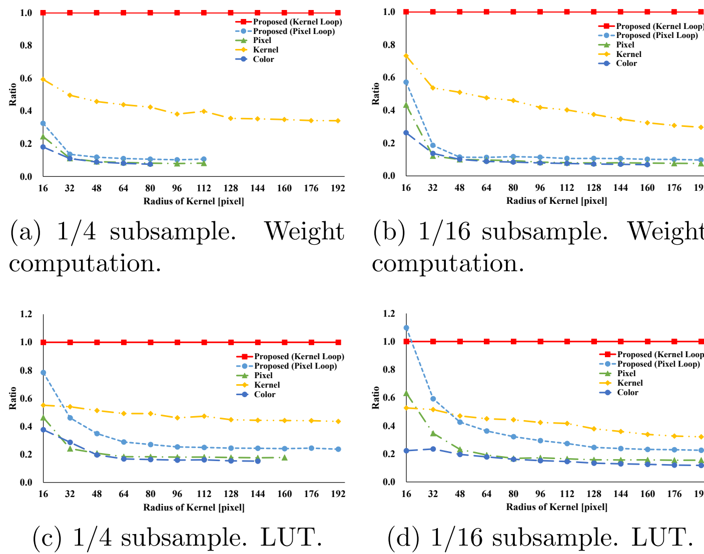
(a) 1/4 subsample. Weight computation.

(b) 1/16 subsample. Weight computation.

(c) 1/4 subsample. LUT.

(d) 1/16 subsample. LUT.

Figure 2.18: Processing time for randomly-kernel-subsampled adaptive Gaussian filtering (RKS-AGF) with respect to the kernel radius of FIR filtering. Arbitrary pixel loop unrolling and arbitrary kernel loop unrolling are used in place of pixel loop unrolling and kernel loop unrolling, respectively.



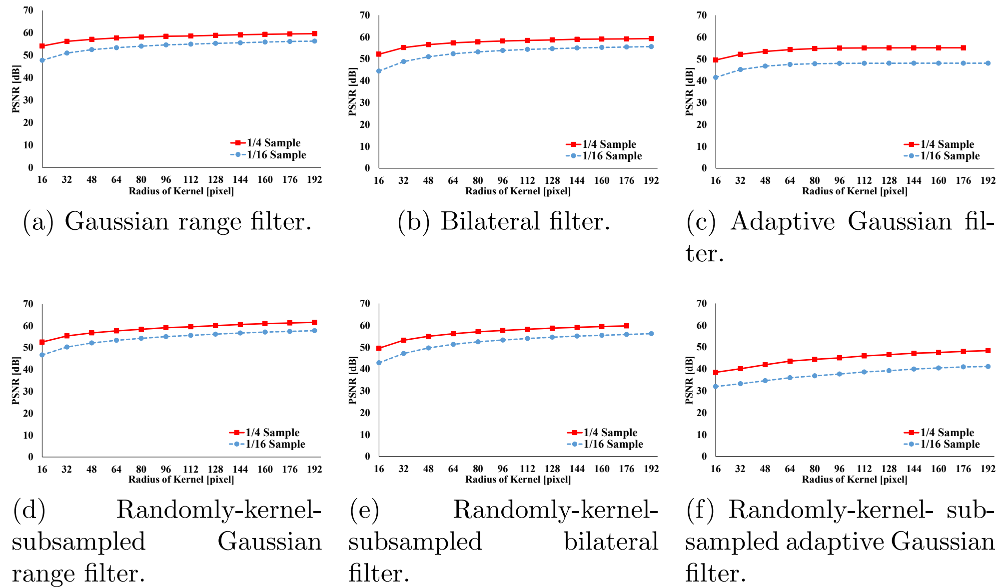
(a) 1/4 subsample. Weight computation.

(b) 1/16 subsample. Weight computation.

(c) 1/4 subsample. LUT.

(d) 1/16 subsample. LUT.

Figure 2.19: The speedup ratio for randomly-kernel-subsample adaptive Gaussian filtering (RKS-AGF) with respect to the kernel radius of FIR filtering. If the ratio exceeds 1, the given pattern is faster than the kernel loop vectorization.



(a) Gaussian range filter.

(b) Bilateral filter.

(c) Adaptive Gaussian filter.

(d) Randomly-kernel-subsampled Gaussian range filter.

(e) Randomly-kernel-subsampled bilateral filter.

(f) Randomly-kernel- subsampled adaptive Gaussian filter.

Figure 2.20: PSNR with respect to kernel radius of FIR filtering. Image size is 512 × 512.

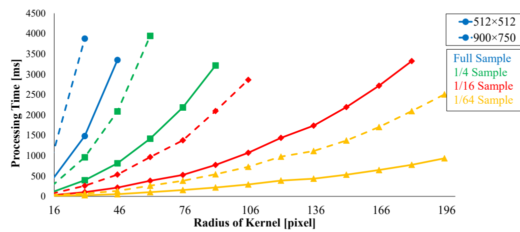


Figure 2.21: Processing time for loop vectorization with respect to the kernel radius of FIR filtering. There are 2 × 4 lines, and their combinations represent image resolution (512 × 512 and 900 × 750) and kernel subsampling ratio (full, 1/4, and 1/16).

Chapter 3

Effective Implementation of Edge-Preserving Filtering on CPU Microarchitectures

3.1 Introduction

Edge-preserving filters [1–5] are the basic tools for image processing. The representatives of the filters include bilateral filtering [1, 2], non-local means filtering [3] and guided image filtering [4–6]. These filters are used in various applications, such as image denoising [3, 7], high dynamic range imaging [8], detail enhancement [9–11], free viewpoint image rendering [12], flash/no-flash photography [13, 14], up-sampling/super resolution [15, 16], alpha matting [5, 18], haze removal [19], optical flow and stereo matching [20], refinement processing in optical flow and stereo matching [21, 22] and coding noise removal [23, 24].

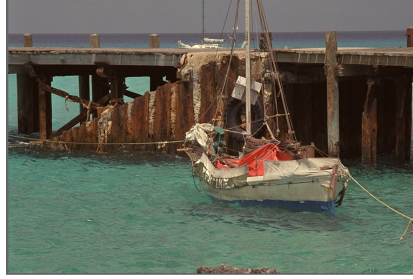
The kernel of the edge-preserving filter can typically be decomposed into range and/or spatial kernels, which depend on the difference between the value and position of reference pixels. Bilateral filtering has a range kernel and a spatial kernel. Non-local means filtering has only a range kernel. The shape of the spatial kernel is invariant across all pixels. By contrast, that of the range kernel is variant; thus, the range kernel is computed adaptively for each pixel. The adaptive computation is expensive.

Several acceleration algorithms have been proposed for the bilateral filtering [8, 25–33] and non-local means filtering [25, 31, 34, 35]. These algorithms reduce the computational order of these filters. The order of the naïve algorithm is $O(r^2)$, where r is the kernel radius. The order of the separable approximation algorithms [25, 26] is $O(r)$, and that of constant-

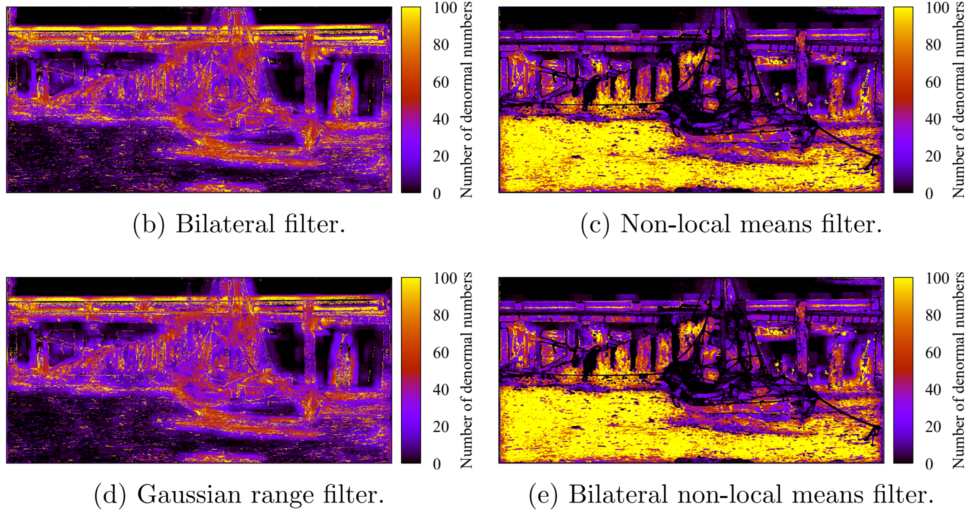
time algorithms [28–37] is $O(1)$. The separable approach is faster than the naïve; however, the approximation accuracy is low. The constant-time algorithms are faster than the $O(r^2)$ and $O(r)$ approaches in large kernel cases. In the case of multi-channel image filtering with intermediate-sized kernels, the method tends to be slower than the naïve algorithms owing to the curse of dimensionality [28], which indicates that the computational cost increases exponentially with increasing dimensions. Furthermore, when the kernel radius is small, the naïve algorithm can be faster than the algorithms of the order $O(r)$ or $O(1)$ owing to the offset times, which refers to pre-processing and post-processing such as creating intermediate images. In the present study, we focus on accelerating the naïve algorithm of the edge-preserving filtering based on the characteristics of computing hardware.

The edge-preserving filter usually involves denormalized numbers, which are special floating-point numbers defined in IEEE Standard 754 [77]. The definition of the denormalized numbers is discussed in Section 3.3. The formats are supported by various computing devices, such as most central processing units (CPUs) and graphics processing units (GPUs). The denormalized numbers represent rather small values that cannot be expressed by normal numbers. Although the denormalized numbers can improve arithmetic precision, their format is different from the normal numbers. Therefore, the processing of the denormalized numbers incurs a high computational cost [78–80]. The edge-preserving filters have small weight values, where a pixel is across an edge. The values tend to be the denormalized numbers. Figure 3.1 shows the occurrence of the denormalized numbers in various edge-preserving filtering. The denormalized numbers do not influence the eventual results, because these values are almost zero in the calculations. Hence, we can compute edge-preserving filtering with high-precision even by omitting the denormalized numbers. Moreover, the omission would be critical for accelerating the edge-preserving filtering.

A fast implementation requires effective utilization of the functionalities in CPUs and GPUs. In the present study, we focus on a CPU-centric implementation. Existing CPU microarchitectures are becoming complex. The architectures are based on multi-core architectures, complicated cache memories and short vector processing units. Single-instruction, multiple-data (SIMD) [41] instruction sets in vector processing units are especially changed. The evolution of the SIMD instructions has taken the form of the increased vector length [42], increased number of types of instructions and decreased latency of instructions. Therefore, it is essential to use SIMD instructions effectively for extracting CPU performance. In the edge-preserving filtering, execution of the weight calculation is the main bottleneck. Thus, the vectorization for weight calculation has a significant effect.



(a) Original image.



(b) Bilateral filter. (c) Non-local means filter.

(d) Gaussian range filter. (e) Bilateral non-local means filter.

Figure 3.1: Occurrence status of denormalized numbers. (**b**–**e**) present heat maps of the occurrence frequency of denormalized numbers in each kernel. The filtering parameters are as follows: $\sigma_r = 4$, $\sigma_s = 6$, $r = 3\sigma_s$ and $h = \sqrt{2}\sigma_r$. The template window size is $(3, 3)$, and the search window size is $(2r + 1, 2r + 1)$. The image size is 768×512. In (**b**–**e**), the ratios of denormalized numbers in all weight calculations are 2.11%, 3.26%, 1.97% and 3.32%, respectively.

In the present study, we focus on two topics: the influence of denormalized numbers and effective vectorized implementation on CPU microarchitectures in the edge-preserving filtering. For the first, we verify the influence of the denormalized numbers on the edge-preserving filtering, and then, we propose methods to accelerate the filter by removing the influence of the denormalized numbers. For the second, we compare several types of vectorization of bilateral filtering and non-local means filtering. We develop various implementations to clarify suitable representations of the latest CPU microarchitectures for these filters.

The remainder of this chapter is organized as follows. In Section 3.2, we review bilateral filtering, non-local means filtering and their variants. Section 3.3 describes IEEE standard 754 for floating point numbers and denormalized numbers. In Section 3.4, we present CPU microarchitectures and SIMD instruction sets. We propose novel methods for preventing the occurrence of the denormalized numbers in Section 3.5. In Section 3.6, we introduce several types of vectorization. Section 3.7 presents our experimental results. Finally, in Section 3.8, we show a few concluding remarks.

3.2 Edge-Preserving Filters

General edge-preserving filtering in finite impulse response (FIR) filtering is represented as follows:

$$\bar{\boldsymbol{I}}(\boldsymbol{p}) = \frac{1}{\eta} \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p}, \boldsymbol{q}) \boldsymbol{I}(\boldsymbol{q}), \tag{3.1}$$

where $\boldsymbol{I}$ and $\bar{\boldsymbol{I}}$ are the input and output images, respectively. $\boldsymbol{p}$ and $\boldsymbol{q}$ are the present and reference positions of pixels, respectively. A kernel-shaped function $\mathcal{N}(\boldsymbol{p})$ comprises a set of reference pixel positions, and it varies for every pixel $\boldsymbol{p}$. The function $f(\boldsymbol{p}, \boldsymbol{q})$ denotes the weight of position $\boldsymbol{p}$ with respect to the position $\boldsymbol{q}$ of the reference pixel. η is a normalizing function. If the gain of the FIR filter is one, we set the normalizing function as follows:

$$\eta = \sum_{\boldsymbol{q} \in \mathcal{N}(\boldsymbol{p})} f(\boldsymbol{p}, \boldsymbol{q}). \tag{3.2}$$

Various types of weight functions are employed in edge-preserving filtering. These weights are composed of spatial and range kernels or only a range kernel. The weight of the bilateral filter is expressed as follows:

$$f(\boldsymbol{p}, \boldsymbol{q}) := \exp\left(\frac{\|\boldsymbol{p} - \boldsymbol{q}\|_2^2}{-2\sigma_s^2}\right) \exp\left(\frac{\|\boldsymbol{I}(\boldsymbol{p}) - \boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}\right), \tag{3.3}$$

where $\| \cdot \|_2$ is the L2 norm and σ_s and σ_r are the standard deviations of the spatial and the range kernels, respectively. The weight of the non-local means filter is as follows:

$$f(\boldsymbol{p}, \boldsymbol{q}) := \exp\left(\frac{\|\boldsymbol{v}(\boldsymbol{p}) - \boldsymbol{v}(\boldsymbol{q})\|_2^2}{-h^2}\right), \tag{3.4}$$

where $\boldsymbol{v}(\boldsymbol{p})$ represents a vector, which includes a square neighborhood of the center pixel $\boldsymbol{p}$. h is a smoothing parameter. The weight of the bilateral filter

is determined by considering the similarity between the color and spatial distance between a target pixel and that of the reference pixel. The weight of the non-local means filter is defined by computing the similarity between the patch on the target pixel and that on the reference pixel. The weight of the non-local means filter is similar to the range weight of the bilateral filter for a multi-channel image.

To discuss the influence of the denormalized numbers, we introduce two variants of the bilateral and non-local means filters, namely the Gaussian range filter and the bilateral non-local means filter. The weight of the Gaussian range filter is expressed as follows:

$$f(\boldsymbol{p}, \boldsymbol{q}) \quad := \quad \exp(\frac{\|\boldsymbol{I}(\boldsymbol{p}) - \boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}). \tag{3.5}$$

The weight of the bilateral non-local means filter [81] is as follows:

$$f(\boldsymbol{p}, \boldsymbol{q}) \quad := \quad \exp(\frac{\|\boldsymbol{p} - \boldsymbol{q}\|_2^2}{-2\sigma_s^2}) \exp(\frac{\|\boldsymbol{v}(\boldsymbol{p}) - \boldsymbol{v}(\boldsymbol{q})\|_2^2}{-h^2}). \tag{3.6}$$

The Gaussian range filter is composed of the range kernel alone in the bilateral filtering. The bilateral non-local means filter is composed of the spatial kernel and the range kernel in the non-local means filtering.

3.3 Floating Point Numbers and Denormalized Numbers in IEEE Standard 754

The formats of floating point numbers are defined in IEEE Standard 754 [77]. The floating point number is composed of a set of normal numbers and four special numbers, which are *not a number (NaN)*, *infinities*, *zeroes* and *denormalized (or subnormal) numbers*. The normal numbers are represented as follows:

$$(-1)^{sign} \times 2^{exponent - bias} \times 1.fraction. \tag{3.7}$$

In a single-precision floating point number (float), parameters are as follows: bit length of *exponent* is 8 bit; that of $fraction$ is 23 bit; $bias = 127$. In a single-precision floating point number (double), parameters are as follow: bit length of *exponent* is 11 bit; that of $fraction$ is 52 bit; $bias = 1023$. In the normal number, *exponent* is neither zero nor the maximum value of *exponent*. In the special numbers, *exponent* is zero or it has the maximum value of *exponent*. When *exponent* and $fraction$ are zero, the format represents zero. When *exponent* of a given number has the maximum value, the

format represents infinity or NaN. In the case of the denormalized numbers, *exponent* is zero, but $fraction$ is not zero. The denormalized numbers are represented as follows:

$$(-1)^{sign} \times 2^{1-bias} \times 0.fraction. \tag{3.8}$$

Note that *exponent* is set to zero for the special number flags, but *exponent* can be forcefully regarded as one even if the settled value is zero. The range of magnitudes of the denormalized numbers is smaller than that of the normal numbers. In terms of float, the range of magnitudes of the normal numbers is $1.17549435 \times 10^{-38} \leq |x| \leq 3.402823466 \times 10^{38}$, while that of the denormalized numbers is $1.40129846 \times 10^{-45} \leq |x| \leq 1.17549421 \times 10^{-38}$. Typical processing units are optimized for the normal numbers. Thus, the normal numbers are processed using specialized hardware. By contrast, the denormalized numbers are processed using general hardware. Therefore, the computational cost of handling the denormalized numbers is higher than that of the normal numbers.

There are three built-in methods for suppressing the speed reduction caused by the denormalized numbers. The first approach is computation with high-precision numbers. A high-precision number format has a large range of magnitudes in a normal number. In float, most denormalized numbers are represented by normal numbers in double. However, the bit length of double is longer than that of float. Thus, computational performance degrades owing to the increased cost of memory write/read operations. The second approach is computation with the flush to zero (FTZ) and denormals are zero (DAZ) flags. These flags are implemented in most CPUs and GPUs. If the FTZ flag is enabled, the invalid result of an operation is set to zero. The invalid result is an underflow flag or a denormalized number. If the DAZ flag is enabled, an operand in assembly language is set to zero when the operand is already a denormalized number. When the computing results are denormalized numbers or operands are already denormalized numbers, the DAZ flag ensures the denormalized numbers are set to zero. These flags suppress the occurrence of denormalized numbers; thereby, computing is accelerated. However, computation with these flags has events that convert denormalized numbers to normal numbers. Hence, the calculation time with these flags is not the same as that without denormalized numbers. In the third approach, a denormalized number is converted into a normal number by a min or max operation. This approach forcibly clips a calculated value to a normal number in the calculation, whether the calculated value is a denormalized number or not. The approach suppresses the denormalized numbers after their occurrence. Thus, it is not optimal for accelerating computation. Therefore, in this study, we propose a novel approach to prevent the occurrence of the

denormalized numbers themselves to eliminate the computational time for handling the denormalized numbers.

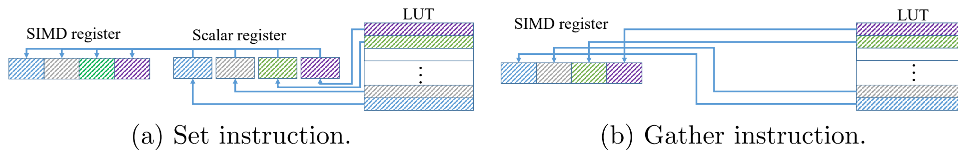
3.4 CPU Microarchitectures and SIMD Instruction Sets

Moore's law [38] states that the number of transistors on an integrated circuit will double every two years. In the early stages, CPU frequencies were increased by increasing the number of transistors. In recent years, owing to heat and power constraints, the use of a larger number of transistors has become difficult [39], such as chips with multiple cores, complicate cache memory and short vector units. The latest microarchitectures used in Intel CPUs are presented in Table. 1.1. The table indicates that the number of cores is increasing, cache memory size is expanding and the SIMD instruction sets are growing.

SIMD instructions simultaneously calculate multiple data. Hence, high-performance computing utilizes SIMD. Typical SIMD instructions include streaming SIMD extensions (SSE), advanced vector extensions (AVX)/AVX2 and AVX512 in order of the oldest to newest [44]. Moreover, fused multiply-add 3 (FMA3) [44] is a special instruction. FMA3 computes $A \times B + C$ by one instruction. There are three notable changes in SIMD. First, the vector length is growing. For example, the lengths of SSE, AVX/AVX2 and AVX512 are 128 bits (4 float elements), 256 bits (8 float elements) and 512 bits (16 float elements), respectively. Second, several instructions have been added, notably, *gather* and *scatter* instructions [42]. These instructions load/store data of discontinuous positions in memory. *gather* has been implemented in the AVX2, and *scatter* has been implemented in the AVX512. Before *gather* was implemented, the *set* instruction was used. The *set* instruction stores data in the SIMD register from scalar registers (see Figure 3.2). Thus, the instruction incurs a high computational cost. Third, even with the same instruction, instruction latency depends on CPU microarchitecture [1]. Therefore, the effective vectorization is different for each CPU microarchitecture.

The expansion of SIMD instructions influences vectorization for the edge-preserving filtering. We can accelerate the filters by the increased number of vectorizing elements. Furthermore, the *gather* instruction is useful for referencing lookup tables (LUTs). Using LUTs is a typical acceleration technique

[1]for example, the latency of the add instruction indicated at `https://software.intel.com/sites/landingpage/IntrinsicsGuide/\#text=_mm256_add_ps`



(a) Set instruction. (b) Gather instruction.

Figure 3.2: Set and gather instructions.

for arithmetic computation [82]. Weights are stored in the LUTs, and then, the weights are used by loading them from the LUTs. The loading process is accelerated by *gather*. Moreover, FMA3 is beneficial for FIR filtering. The summation term of Equation 3.1 can be realized by using FMA3. Therefore, we can accelerate the edge-preserving filtering with proper usage of SIMD.

3.5 Proposed Methods for the Prevention of Denormalized Numbers

An edge-preserving filter has a range kernel and a spatial kernel or only a range kernel. When the similarity of color intensity is low, and the spatial distance is long, the weight of the kernel is exceedingly small. For example, in the bilateral filter for a color image, when the parameters are $\sigma_r = 32$, $\boldsymbol{I}(\boldsymbol{p}) = (255, 255, 255)$ and $\boldsymbol{I}(\boldsymbol{q}) = (0, 0, 0)$, the range weight is 4.29×10^{-42}, which is the minimum value of the range kernel and is a denormalized number in float. Note that the remaining spatial kernel does not multiply the weight. Thus, the total value becomes smaller than the range weight. Moreover, the non-local means filtering is more likely to involve denormalized numbers from Equation 3.4. Notably, the occurrence frequency of denormalized numbers is low when the smoothing parameters are large. This parameter overly smooths edge-parts; thus, the smoothing parameters should be small in most cases.

We propose new methods to prevent the occurrence of denormalized numbers for the edge-preserving filtering. The proposed methods deal with the two cases: a weight function contains only one term or multiple terms. For the former cases, we consider two implementations: computing directly and referring to only one LUT. For the latter cases, we also consider two implementations: computing directly and referring to each of multiple LUTs.

For the one-term case, the argument of the term is clipped using appropriate values so that the resulting value is not a denormalized number. If the weight function is a Gaussian distribution, the argument value x satisfies the

following equations:

$$\begin{aligned} \exp(x) &> \delta_{max}, \\ x &> \ln(\delta_{max}), \end{aligned} \tag{3.9}$$

where δ_{max} is the maximum value of the denormalized number. In other words, Equation 3.9 can be written as follows:

$$x \geq \ln(\nu_{min}), \tag{3.10}$$

where ν_{min} denotes the minimum value of the normal number. δ_{max} and ν_{min} are set based on the precision of floating point numbers. In the proposed method, an argument value is clipped by $-87.3365478515625 = \ln(\nu_{min})$ in float.

For the multiple terms, the clipping method is inadequate because denormalized numbers could occur owing to the multiplication of multiple terms. Therefore, the weights are multiplied by an offset value in the proposed method. The offset value satisfies the following equations:

$$o \times \prod_{n}^{N} \min_{x \in \Lambda_n} w_n(x) > \delta_{max}, \tag{3.11}$$

$$\frac{\nu_{max}}{255|\mathcal{N}(\boldsymbol{p})|} \geq o \times \prod_{n}^{N} \max_{x \in \Lambda_n} w_n(x), \tag{3.12}$$

$$\nu_{max} \geq o > \delta_{max}, \tag{3.13}$$

where o is an offset value and w_k is the k-th weight function, which is a part of the decomposed weight function. N is the number of terms in the weight function. Λ_k is a set of possible arguments in the k-th weight function, and ν_{max} is the maximum value of normal numbers. Equation 3.12 limits the summation in Equation 3.1 such that it does not exceed the normal number when the image range is 0–255. Notably, $\min_x w_n(x)$ and its product are occasionally zero owing to underflow, even if the mathematical results of the weight function are non-zero. When the number of terms is large or $\min_x w_n(x)$ is very small, o is very large. Therefore, Equations (3.12) and (3.13) cannot be satisfied. In this condition, we must reduce the number of significant figures of $w_n(\cdot)$ to eliminate the occurrence of denormalized numbers. Note that o should be large to ensure that the number of significant figures of $w_n(\cdot)$ is large. In the edge-preserving filtering, $\max_x w_n(x)$ is one. Therefore, Equation 3.12 is transformed as follows:

$$\frac{\nu_{max}}{255|\mathcal{N}(\boldsymbol{p})|} \geq o. \tag{3.14}$$

Accordingly, o should be $\frac{\nu_{max}}{255|\mathcal{N}(\boldsymbol{p})|}$, if we achieve higher accuracy. Even when the number of significant figures cannot be decreased sufficiently, the method can decrease the rate of occurrence of denormalized numbers.

The proposed methods are implemented by using max and/or multiplication operations. The weight function of the bilateral filter is considered to be composed of only one term or multiple terms. The one-term case of the bilateral filter is implemented as follows:

$$f(\boldsymbol{p},\boldsymbol{q}) := \exp(\max(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s^2}+\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2},\ln(\nu_{min}))). \quad (3.15)$$

The multiple terms case is implemented as follows:

$$\begin{aligned} f(\boldsymbol{p},\boldsymbol{q}) &:= o\times\exp(\max(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s^2}),\ln(\nu_{min})) \\ &\quad \exp(\max(\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2})),\ln(\nu_{min})), \quad (3.16) \\ o &= \frac{\nu_{max}}{255|\mathcal{N}(\boldsymbol{p})|}, \quad (3.17) \end{aligned}$$

where the following equation must be satisfied:

$$o\times\exp(\frac{2r^2}{-2\sigma_s^2})\exp(\frac{3\times 255^2}{-2\sigma_r^2}) > \delta_{max}. \quad (3.18)$$

Note that o has no effect unless it is firstly multiplied by the term of the decomposed weight function. If the equation is not satisfied because the minimal values of the range and spatial kernels are very small, Equation 3.16 is transformed as follows:

$$\begin{aligned} f(\boldsymbol{p},\boldsymbol{q}) &:= o\times\exp(\max(\frac{\|\boldsymbol{p}-\boldsymbol{q}\|_2^2}{-2\sigma_s^2},s)) \\ &\quad \exp(\max(\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{-2\sigma_r^2}),s)), \quad (3.19) \end{aligned}$$

where s controls the number of significant figures and is obtained using the same method as Equation 3.10. The other filters can be realized in the same way. The computational costs of the proposed methods are significantly lower than the cost of computing denormalized numbers. Moreover, when using LUTs, the costs of the proposed methods can be neglected. In this case, the proposed methods are applied in preprocessing for creating LUTs. Therefore, the benefits of the proposed methods can be significant.

3.6 Effective Implementation of Edge-Preserving Filtering

In the edge-preserving filtering, weight calculation accounts for the largest share of processing time. Thus, we consider the implementation of the weight calculation. There are three approaches for the arithmetic computation of the weight function [82]: direct computation, by using LUTs and a combination of both [82]. Computing of usual arithmetic functions has lower cost than the transcendental functions, i.e., exp, log, sin and cos, or heavy algebraic functions, e.g., sqrt. For the high-cost function, the LUT is effective when arithmetic computing is a bottleneck. By contrast, computing is valid when memory I/O is a bottleneck. We can control the trade-off by using the LUT and computation.

In the bilateral filter, the possible types of implementation are as follows:

- RCSC: range computing spatial computing; range and spatial kernels are directly and separately computed.
- MC: merged computing; range and spatial kernels are merged and directly computed.
- RCSL: range computing spatial LUT; the range kernel is directly computed, and LUTs are used for the spatial kernel.
- RLSC: range LUT spatial computing; LUTs are used for the range kernel, and the spatial kernel is directly computed.
- RLSL: range LUT spatial LUT; LUTs are used for both range and spatial kernels.
- ML: merged LUT; LUTs are used for the merged range and spatial kernels;
- RqLSL, RLSqL: range (quantized) LUT spatial (quantized) LUT; LUTs are quantized for each range and spatial LUT in RLSL
- MqL: merged quantized LUT; range and spatial kernels are merged, and then, the LUTs are quantized.

In the non-local means filtering process, the possible types of implementation are reduced, because the filter does not contain the spatial kernel. The possible types of implementation are as follows:

- RC: range computing; the range kernel is directly computed.

- RL: range LUT; LUTs are used for the range kernel.
- RqL: range quantized LUT; quantized LUTs are used for the range kernel.

We consider five types of implementation for bilateral filtering, namely, MC, RCSL, RLSL, RqLSL and MqL. Notably, we did not implement the RCSC, RLSC, RLSqL and ML, because the cost of computing the spatial kernel is lower than that of computing the range kernel, and the size of the range/merged LUT is larger than that of the spatial LUT. We also implement three types for non-local means filtering, such as RC, RL and RqL. Note that the pairs of MC/RC, RLSL/RL and RqLSL/RqL are similar without spatial computation.

In the MC/RC implementation, weights are directly computed for each iteration. In the bilateral and bilateral non-local means filtering, two exponential terms are computed as one exponential term considering the nature of the exponential function. The implementation is computationally expensive because it involves weight calculation every time. However, this point is not always a drawback. The calculation increases arithmetic intensity, which is the ratio of the number of float-number operations per the amount of the accessed memory data. When the arithmetic intensity is low, the computational time is limited by memory reading/writing [83]. In image processing, arithmetic intensity tends to be low, but the MC/RC implementation improves the arithmetic intensity. Therefore, the MC/RC implementation may be practical in a few cases.

RCSL can be applied to filters, which contain a spatial kernel. These filters include the bilateral and bilateral non-local means filters. The exponential term of the range kernel is computed every time, and LUTs are used as the weights of the spatial kernel. The size of the spatial LUT is the kernel size. In the bilateral filters, the weight function in the proposed implementation can be expressed as follows:

$$
\begin{aligned}
f(\boldsymbol{p},\boldsymbol{q}) &:= EXP_s[\boldsymbol{p}-\boldsymbol{q}]\exp\left(-\frac{\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2}{2\sigma_r^2}\right), && (3.20)\\
EXP_s[\boldsymbol{x}] &:= \exp\left(\frac{\|\boldsymbol{x}\|_2^2}{-2\sigma_s^2}\right), && (3.21)
\end{aligned}
$$

where $EXP_s[\cdot]$ is the spatial LUT. The first term is calculated for all possible arguments; subsequently, the weight values are stored in a LUT before filtering. Because the combinations of the relative distances of $\boldsymbol{p}$ and $\boldsymbol{q}$ are identical in all kernels, it is not required to calculate the relative distances for each kernel.

In RLSL/RL, LUTs are used as the weights of the range and the spatial kernels. The LUTs are created for the range or spatial kernel. For Gaussian range and non-local means filtering, the spatial kernel is omitted or always considered to be one. Only one LUT is used for the kernel, but the LUT is referenced for each channel using the separate representation of an exponential function. Notably, we can save the LUT size, which is 256. If we use an LUT for merged representation of an exponential function, its size becomes $255^2 \times 3 + 1 = 195{,}075$. In the bilateral filter for a color image, the weight function of the implementation is expressed as follows:

$$\begin{aligned} f(\boldsymbol{p},\boldsymbol{q}) &:= EXP_s[\boldsymbol{p}-\boldsymbol{q}] \\ &\quad EXP_r[\lfloor\|\boldsymbol{I}(\boldsymbol{p})_r-\boldsymbol{I}(\boldsymbol{q})_r\|_1\rfloor] \\ &\quad EXP_r[\lfloor\|\boldsymbol{I}(\boldsymbol{p})_g-\boldsymbol{I}(\boldsymbol{q})_g\|_1\rfloor] \\ &\quad EXP_r[\lfloor\|\boldsymbol{I}(\boldsymbol{p})_b-\boldsymbol{I}(\boldsymbol{q})_b\|_1\rfloor] \qquad (3.22) \\ EXP_r[x] &:= \exp(\frac{x^2}{-2\sigma_r^2}), \qquad (3.23) \end{aligned}$$

where $\lfloor\cdot\rfloor$ is the floor function, $\|\cdot\|_1$ is the L1 norm and $\boldsymbol{I}(\cdot)_r$, $\boldsymbol{I}(\cdot)_g$ and $\boldsymbol{I}(\cdot)_b$ are the red, green and blue channels in $\boldsymbol{I}(\cdot)$, respectively. $EXP_r[\cdot]$ is the range LUT, and $EXP_s[\cdot]$ is identical to Equation 3.21. These LUTs are accessed frequently. Hence, the arithmetic intensity is low.

In RqLSL/RqL, the range LUT for the merged representation of an exponential function is quantized to reduce the LUT size. Therefore, the LUT is approximated. This implementation is faster than using a large LUT and accessing the LUT multiple times, such as RLSL/RL. In the bilateral filter, the weight function of the implementation is expressed as follows:

$$\begin{aligned} f(\boldsymbol{p},\boldsymbol{q}) &:= EXP_s[\boldsymbol{p}-\boldsymbol{q}]EXP_{rq}[\lfloor\phi(\|\boldsymbol{I}(\boldsymbol{p})-\boldsymbol{I}(\boldsymbol{q})\|_2^2)\rfloor], \qquad (3.24) \\ EXP_{rq}[x] &:= \exp(\frac{\psi(x)}{-2\sigma_r^2}), \qquad (3.25) \end{aligned}$$

where $\phi(\cdot)$ and $\psi(\cdot)$ denote a quantization function and an inverse quantization function, respectively. By converting the range of the argument through the quantization function, the size of the LUT can be reduced. The LUT size is $\lfloor\phi(3\times 255^2)\rfloor+1$. We use the square root function (sqrt) and division for the quantization function. In sqrt, the quantization function and the inverse quantization function are expressed as follows:

$$\begin{aligned} \phi(x) &:= n\sqrt{x}, \qquad (3.26) \\ \psi(x) &:= \frac{x^2}{n^2}, \qquad (3.27) \end{aligned}$$

where n controls the LUT size. In div, they are expressed as follows:

$$\phi(x) := \frac{x}{n}, \tag{3.28}$$

$$\psi(x) := x \times n. \tag{3.29}$$

In sqrt, the size of the quantization range LUT is $442 = \lfloor\sqrt{3 \times 255^2}\rfloor + 1$. In div, it is $195{,}076 = 3 \times 255^2 + 1$, where $n = 1$.

In MqL, the range and spatial LUTs are merged, and then, the LUT is quantized. This implementation uses only one LUT. Thus, we do not require to multiply the range and spatial kernels. In the MqL, the weight function of the bilateral filter is expressed as follows:

$$f(\boldsymbol{p}, \boldsymbol{q}) := EXP_{rq}[\lfloor\phi(\frac{\sigma_r^2}{\sigma_s^2}\|\boldsymbol{p}-\boldsymbol{q}\|_2^2 + \|\boldsymbol{I}(\boldsymbol{p}) - \boldsymbol{I}(\boldsymbol{q})\|_2^2)\rfloor], \tag{3.30}$$

where $EXP_{rq}[\cdot]$ is identical to Equation 3.25. The other filters are implemented in the same way. The size of the quantization merged LUT is larger than that of the quantization range LUT. The quantization merged LUT can be accessed only once in the weight calculation. The accuracy decreases, as does the quantization range LUT.

Furthermore, we consider the data type of an input image. The typical data type of input images is unsigned char, although floating point numbers are used in filter processing. Therefore, unsigned char values of the input image are converted to float/double before the filtering or during the filtering. Note that float is typically used. The bit length of the double is longer than that of float. Hence, the computational time when using double is slower than that when using float. When the input type is unsigned char, we must convert pixel values redundantly to floating point numbers for every loaded pixel. The converting time is SK, where S and K are the image and kernel size, respectively. By contrast, if the input type is a floating point number, which is pre-converted before filtering, the conversion process can be omitted in filtering. The converting times is S. However, in the case of inputting unsigned char, the arithmetic intensity is higher than that of float. This is because the bit length of unsigned char is shorter than that of float. We should consider the tradeoff between the number of converting times and arithmetic intensity owing to the size of the image and kernel.

In the use of LUTs, these implementation approaches can be applied to arbitrary weight functions, which are not limited to the weighting functions consisting of exponential functions in the present study. Especially, if a weight function is computationally expensive, the use of a LUT is more practical.

Table 3.1: Computers used in the experiments.

CPU	Intel Core i7 3970X	Intel Core i7 4960X	Intel Core i7 5960X	Intel Core i7 6950X	Intel Core i9 7980XE	AMD Ryzen Threadripper 1920X
memory	DDR3-1600 16 GBytes	DDR3-1866 16 GBytes	DDR4-2133 32 GBytes	DDR4-2400 32 GBytes	DDR4-2400 16 GBytes	DDR4-2400 16 GBytes
SIMD instruction sets	SSE4.2 AVX	SSE4.2 AVX	SSE4.2 AVX/AVX2 FMA3	SSE4.2 AVX/AVX2 FMA3	SSE4.2 AVX/AVX2 AVX512F FMA3	SSE4.2 AVX/AVX2 FMA3

Notably, in these types of implementations, the weight of the target pixels, which is at the center of the kernel, need not be calculated. The weight of the target pixels is always one. When r is small, this approach accelerates the filtering process somewhat.

3.7 Experimental Results

We verified the occurrence of denormalized numbers in the bilateral filtering, non-local means filtering, Gaussian range filtering and bilateral non-local means filtering processes. Moreover, we discussed the effective vectorization of bilateral filtering and non-local means filtering on the latest CPU microarchitectures. These filters were implemented in C++ by using OpenCV [84]. Additionally, multi-core parallelization was executed using Concurrency, which is a parallelization library provided by Microsoft, also called the parallel patterns library (PPL). Table 3.1 shows the CPUs, SIMD instruction sets and memory employed in our experiments. Windows 10 64-bit was used as the OS, and Intel Compiler 18.0 was employed. For referring LUTs, the *set* or *gather* SIMD instructions were employed. The outermost loop was parallelized by multi-core threading, and we had pixel-loop vectorization [85]. This implementation was found to be the most effective [85]. Notably, a vectorized exponential operation is not implemented in these CPUs. Hence, we employed a software implementation, which is available in Intel Compiler. The experimental code[2] spanned around 95,000 lines.

3.7.1 Influence of Denormalized Numbers

We compared the proposed methods with the straightforward approach (none) and four counter-approaches for denormalized numbers. These approaches involve converting denormalized numbers to normal numbers (convert) and using FTZ and/or DAZ flags (FTZ, DAZ, and FTZ and DAZ). The convert implementation clips a calculated value to a normal number value in

[2] `https://github.com/yoshihiromaed/FastImplementation-BilateralFilter`

the weight calculation by means of a min-operation with the minimal value of the normal numbers, such as $\min(exp(a) \times exp(b), v)$, where a and b are variables and v is the minimal value of the normal numbers. Figures 3.3–3.10 show the results of computational time and speedup ratio of various types of implementation on Intel Core i9 7980XE. The computational time was taken as the median value of 100 trials. The parameters were identical for all filters. Notably, in the RqLSL/RqL and MqL implementation, sqrt was used as the quantization function, and $n = 1$. These figures indicate that the proposed methods for handing denormalized number were the fastest among the other approaches for each implementation. The proposed methods were up to four-times faster than the straightforward approach. In many cases, the none implementation using double was faster than that using float. The range of magnitudes of double was larger than that of float. Hence, the occurrence frequency of denormalized numbers was lower. In the case of double, however, there was no significant speedup in all approaches for managing denormalized numbers. Because double had twice the byte length compared to float, the corresponding computational time was approximately twice as long, as well. Therefore, the implementation using double was slower than that using float when the influence of denormalized numbers was eliminated. In the RqL of the Gaussian range and the non-local means filters and the MqL of the bilateral and bilateral non-local means filters, the speedup ratio of the proposed methods was almost the same as that of the convert, FTZ and FTZ and DAZ implementation. In these approaches, denormalized numbers occurred only when LUTs were created, and the denormalized numbers were eliminated during LUT creation. Thus, during the filtering process, denormalized numbers did not occur. Therefore, the RqL and MqL implementation could achieve the same effect as the proposed methods did. In addition, DAZ had almost no effect because DAZ was executed only if an operand was a denormalized number. As shown in Figure 3.1, denormalized numbers occurred in edges. Therefore, if the weight of the range kernel was small or the multiplication of the range kernel with the spatial kernel was possible, denormalized numbers were likely to occur.

Tables 3.2–3.5 show the speedup ratio of the MC/RC implementation between the proposed methods and the none implementation for each set of smoothing parameters. Note that when σ_s, r, and the search window size are larger, the amount of processing increases. The tables indicate that the proposed methods are 2–5-times faster than the none implementation. When the smoothing parameters are small and the amount of processing is large, the speedup ratio is high. Therefore, the influence of denormalized numbers is strong when the degree of smoothing is small and the amount of processing is large.

To verify the accuracy of the proposed methods, we compared the scalar implementation in double-precision with the proposed methods and other approaches regarding peak signal-to-noise ratio (PSNR). The results are shown in Figure 3.11. The proposed methods hardly affect accuracy. Note that the accuracies of the RqL and MqL implementation are slightly lower than those of the other types of implementation because the LUTs are approximated. In these types, the accuracy deterioration is not significant because human vision does not sense differences higher than 50 dB [86, 87].

3.7.2 Effective Implementation on CPU Microarchitectures

In this subsection, we verify the effective vectorization of the bilateral filter and the non-local means filter on the latest CPU microarchitectures. The proposed methods for denormalized numbers have already been applied to these filters. Figures 3.12 and 3.13 show the computational times of the bilateral filter and the non-local means filter for each CPU microarchitecture. These filters were implemented using float. Notably, in the RqLSL/RqL and MqL implementation, sqrt was used as the quantization function and $n = 1$. The RqLSL/RqL implementation is the fastest in these CPU microarchitectures. This implementation has a lower computational cost than the MC/RC implementation. Moreover, the number of LUT accesses is lower than that in the case of the RLSL/RL implementation. The computational time of the MC/RC implementation is almost the same as that of the RLSL/RL and RqLSL/RqL implementation or faster than that of the RLSL/RL implementation. This tendency can be stronger in the latest CPU microarchitectures because computational time is limited by the memory reading/writing latency. Besides, the computation time of the RqLSL implementation is almost the same as that of the MqL implementation, but that of the RqLSL implementation is slightly faster than that of the MqL implementation. The size of the merged quantization LUT is larger than that of the range quantization LUT. The effects of the size of the quantization LUT and the quantization function are discussed in the following paragraph. The RLSL/RL, RqLSL/RqL and MqL implementations in which the *gather* instruction is employed are faster than the implementations in which the *set* instruction is employed. However, on the Intel Core i7 5960X and AMD Ryzen Threadripper 1920X CPUs, the implementations in which the *gather* instruction is used are slower than the implementations in which the *set* instruction is used. In the bilateral filter and the non-local means filter, when the SIMD's vector length increases, all types of implementations with longer SIMD instructions

are faster than that with shorter SIMD instructions. Furthermore, in a comparison of the implementations with/without FMA3, the FMA3 instruction improved computational performance slightly. These results indicate that effective vectorization of the bilateral filter and the non-local means filter are different for each CPU microarchitecture. Figures 3.14 and 3.15 show the speedup ratio of the bilateral filter and the non-local means filter for each CPU microarchitecture. If the ratio exceeds one, the corresponding implementation is faster than the scalar implementation for each CPU microarchitecture. Multi-core threading parallelized both the scalar and the vectorized implementations for focusing on comparing vectorized performance. In the case of the bilateral filter, the fastest implementation is 170-times faster than the scalar one. Moreover, in the case of the non-local means filter, the fastest implementation is 200-times faster than the scalar one. The speedup was determined by the management of denormalized numbers and effective vectorization. Thus, the effect of using a multi-core CPU is not evaluated in the verification.

We discuss the relationship between accuracy and computational time for various types of implementation involving the use of the quantization LUT. Figure 3.16 shows the relationship between calculation time and accuracy of the RqL and the MqL implementation. In this figure, we changed the quantization functions and the size of the quantization range/merged LUTs. The quantization functions were square root function (sqrt) and division (div). Note that the maximal value in the case of the RqLSL/RqL implementation is commensurate with that of the non-quantized cases. In the RqLSL/RqL and MqL implementation, the accuracy and computational time have a tradeoff. These implementations accelerate these filters while maintaining high accuracy, when the LUT size is practical. The characteristics of the performance depend on the quantization functions. Therefore, we must choose the functions and the LUT size by considering the required accuracy and computational time.

Figure 3.17 shows the computational time of using unsigned char (8U) and float (32F) in the MC/RC implementation. Note that the computational time is plotted on a logarithmic scale. The 8U implementation of the bilateral filter is faster than the 32F implementation, when r is small. By contrast, when r is large, the 8U implementation is slower than the 32F implementation. If the cost of the conversion process in 8U is low, the arithmetic intensity of the 8U implementation would be larger than that of the 32F implementation. However, in the 8U implementation, the conversion cost increases owing to the redundant conversion of the pixels as the amount of processing increases. In the non-local means filter, when the template window size is $(3, 3)$, the 8U implementation is always faster than the 32F

implementation. When the template window size is $(5, 5)$, the 8U implementation is slower than the 32F implementation in the case of the large search window. The arithmetic intensity of the non-local means filter is low because many pixels are accessed. Therefore, we can improve the arithmetic intensity by using the 8U implementation. However, if the amount of processing is large, we should use 32F. As a result, the speeds of the 8U and 32F implementation depend on the amount of processing, which are changed by the filtering kernel size and arithmetic intensity.

Figure 3.18 shows the speedup ratio of each implementation of the proposed methods for prevention of denormalized numbers. If the speedup ratio exceeds one, the implementation is faster than the scalar implementation. All types of implementation were parallelized by multi-core threading. The figure shows that the fastest implementation of the bilateral filter and the non-local means filter is 152- and 216-times faster than the scalar implementation, respectively. Therefore, we can achieve significant acceleration by applying the proposed methods for preventing the occurrence of denormalized numbers and selecting the implementation approaches of weight calculation and the appropriate data type.

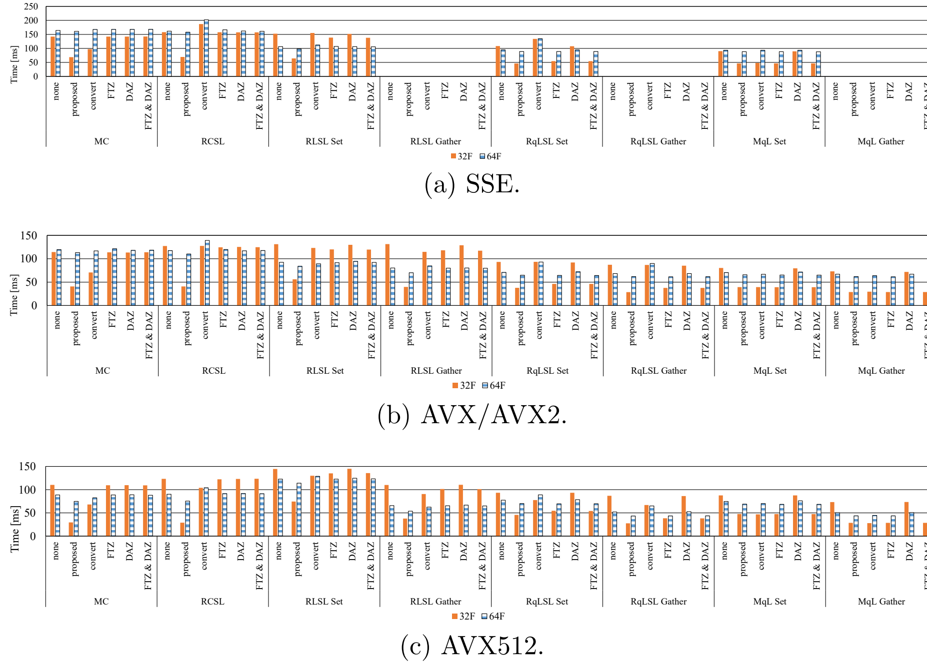
Finally, we compared the fastest implementation with OpenCV, which is the de facto standard image processing library [84]. Figure 3.19 shows the computational times of our implementation and the OpenCV's implementation with its speedup ratio. Notably, the computational time is plotted on the logarithmic scale. Our method is 2–14-times faster than OpenCV. In the OpenCV implementation, the distance function of the range kernel is not the L2 norm, and the kernel shape is circular for acceleration (see Appendix B); therefore, PSNR is low. By contrast, our implementation slightly approximates the kernel LUT, and the kernel shape is rectangular. Therefore, the proposed methods are more practical.

3.8 Conclusions

In this chapter, we propose methods to accelerate bilateral filtering and non-local means filtering. The proposed methods prevent the occurrence of denormalized numbers, which has a large computational cost for processing. Moreover, we verify various types of vectorized implementations on the latest CPU microarchitectures. The results are summarized as follows:

1. The proposed methods are up to five-times faster than the implementation without preventing the occurrence of denormalized numbers.

2. In modern CPU microarchitectures, the *gather* instruction in the SIMD



(a) SSE.

(b) AVX/AVX2.

(c) AVX512.

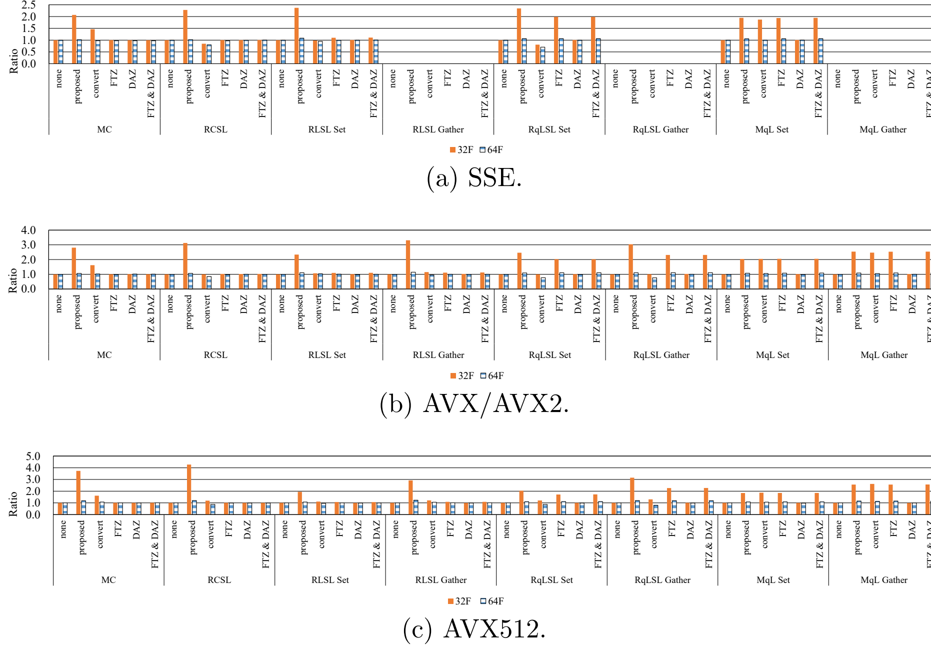
Figure 3.3: Computational time of the bilateral filter on Intel Core i9 7980XE. The computational times are shown in terms of single precision (32F) and double precision (64F) floating point numbers. $\sigma_r = 4$, $\sigma_s = 6$, $r = 3\sigma_r$. Image size is 768×512.

instruction set is effective for loading weights from the LUTs.

3. By reducing the LUT size through quantization, the filtering can be accelerated while maintaining high accuracy. Moreover, the optimum quantization function and the quantization LUT size depends on the required accuracy and computational time.

4. When the kernel size is small, the 8U implementation is faster than the 32F implementation. By contrast, in the case of the large kernel, the 32F implementation is faster than the 8U implementation.

In the future, we will verify the influence of denormalized numbers on GPUs. In particular, we are planning to implement edge-preserving filters on GPUs and to verify the influence of the denormalized numbers on computation time in GPUs. Furthermore, we will design effective implementations of the filters on GPUs.



(a) SSE.

(b) AVX/AVX2.

(c) AVX512.

Figure 3.4: Speedup ratio of bilateral filter on Intel Core i9 7980XE. The speedup ratio is shown regarding single precision (32F) and double precision (64F) floating point numbers. If the ratio exceeds one, all implementation of the method are faster than the straightforward implementation (none). $\sigma_r = 4$, $\sigma_s = 6$, $r = 3\sigma_r$. Image size is 768 × 512.

Table 3.2: Computational time and speedup ratio of bilateral filtering in the merged computing (MC) implementation using AVX512 for various parameters. These results were obtained on an Intel Core i9 7980XE. If the ratio exceeds 1, the proposed methods are faster than the straightforward implementation (none) for each parameter. $r = 3\sigma_s$, and image size is 768 × 512.

(a) Computational time (proposed) [ms].

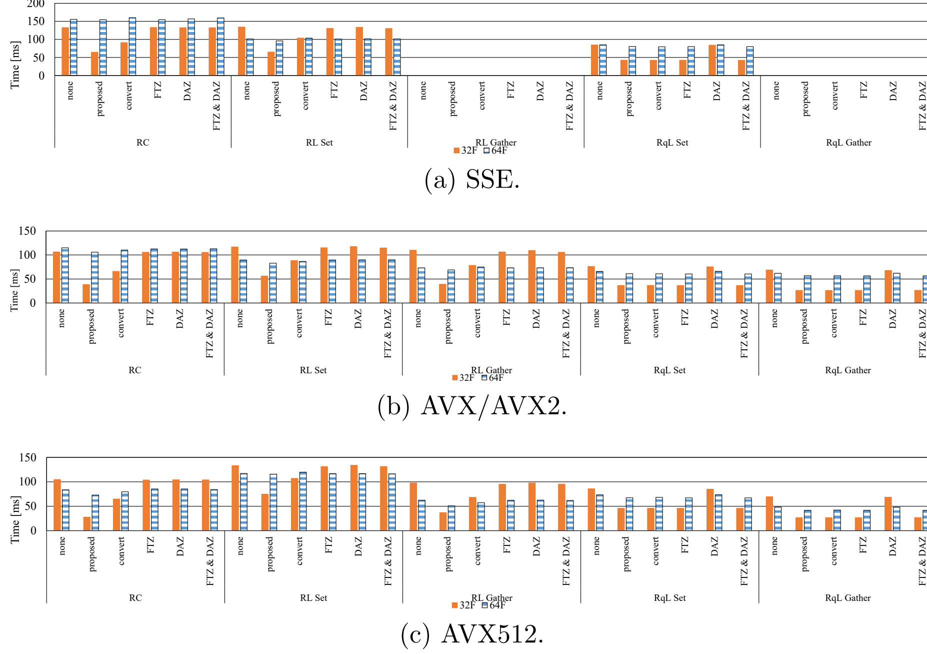
σ_s \ σ_r	**4**	**8**	**16**
4	17.48	17.57	17.63
8	43.96	43.86	43.73
16	147.76	147.58	147.50

(b) Computational time (none) [ms].

σ_s \ σ_r	**4**	**8**	**16**
4	66.48	51.36	50.24
8	217.55	194.94	192.96
16	763.87	755.56	719.00

(c) Speedup ratio.

σ_s \ σ_r	**4**	**8**	**16**
4	3.80	2.92	2.85
8	4.95	4.45	4.41
16	5.17	5.12	4.87



(a) SSE.

(b) AVX/AVX2.

(c) AVX512.

Figure 3.5: Computational time of Gaussian range filter on Intel Core i9 7980XE. The computational times are shown in terms of single precision (32F) and double precision (64F) floating point numbers. $\sigma_r = 4$, and $r = 18$. Image size is 768×512.

Table 3.3: Computational time and speedup ratio of Gaussian range filter in the range computing (RC) implementation using AVX512 for various parameters. These results were obtained on an Intel Core i9 7980XE. If the ratio exceeds 1, the proposed methods are faster than the straightforward implementation (none) for each parameter. Image size is 768×512.

(a) Computational time (proposed) [ms].

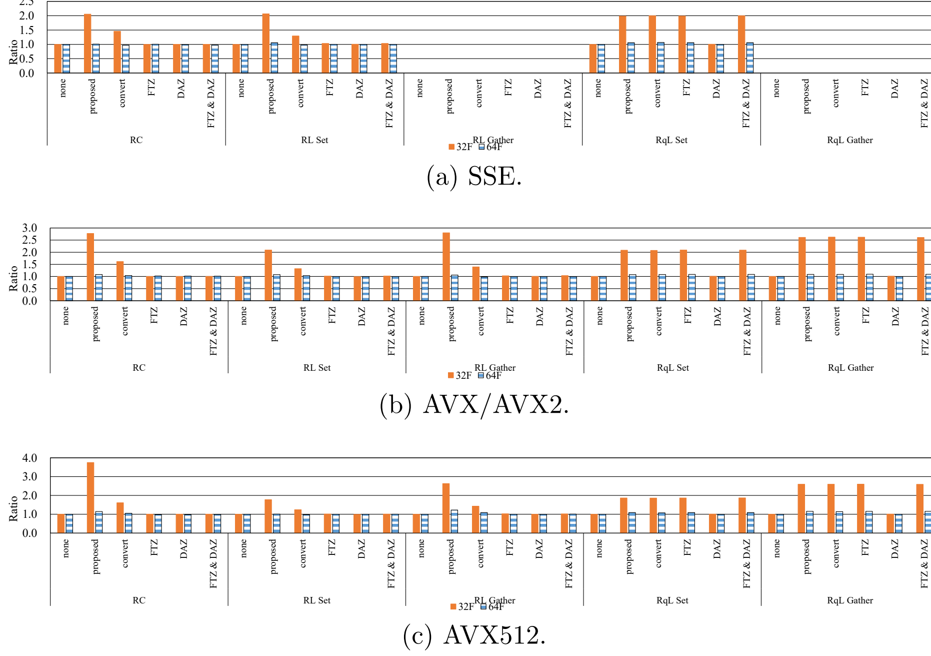
r \ σ_r	4	8	16
12	16.76	16.83	16.85
24	42.53	42.40	42.39
48	143.24	143.01	142.58

(b) Computational time (none) [ms].

r \ σ_r	4	8	16
12	63.95	48.69	48.09
24	207.54	185.16	181.40
48	741.42	717.41	685.18

(c) Speedup ratio.

r \ σ_r	4	8	16
12	3.82	2.89	2.85
24	4.88	4.37	4.28
48	5.18	5.02	4.81



(a) SSE.

(b) AVX/AVX2.

(c) AVX512.

Figure 3.6: Speedup ratio of Gaussian range filter on Intel Core i9 7980XE. The speedup ratio is shown in single precision (32F) and double precision (64F) floating point numbers. If the ratio exceeds one, all implementation of the method are faster than the straightforward implementation (none). $\sigma_r = 4$, and $r = 18$. Image size is 768×512.

Table 3.4: Computational time and speedup ratio of non-local means filter in the RC implementation using AVX512 for various parameters. These results were obtained on an Intel Core i9 7980XE. If the ratio exceeds 1, the proposed methods are faster than the straightforward implementation (none) for each parameter. Template window size is $(3, 3)$, and image size is 768×512.

(a) Computational time (proposed) [ms].

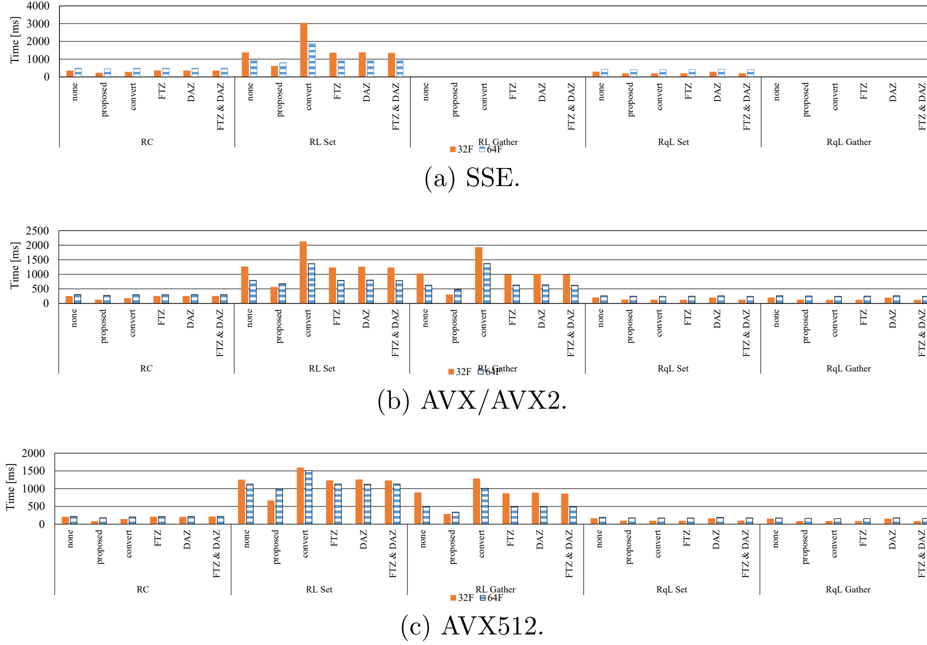
Search Window \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
(25, 25)	40.31	39.80	39.64
(49, 49)	128.90	128.90	128.90
(97, 97)	485.88	485.88	485.36

(b) Computational time (none) [ms].

Search Window \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
(25, 25)	98.91	82.84	80.66
(49, 49)	332.88	307.58	300.00
(97, 97)	1148.48	1158.53	1134.93

(c) Speedup ratio.

Search Window \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
(25, 25)	2.45	2.08	2.03
(49, 49)	2.58	2.39	2.33
(97, 97)	2.36	2.38	2.34



(a) SSE.

(b) AVX/AVX2.

(c) AVX512.

Figure 3.7: Computational time of non-local means filter on Intel Core i9 7980XE. The computational times are shown in terms of single precision (32F) and double precision (64F) floating point numbers. $h = 4\sqrt{2}$, template window size is $(3, 3)$, and search window size is $(37, 37)$. Image size is 768×512.

Table 3.5: Computational time and speedup ratio of the bilateral non-local means filter in the MC implementation using AVX512 for various parameters. These results were calculated using an Intel Core i9 7980XE. If the ratio exceeds 1, the proposed methods are faster than the straightforward implementation (none) for each parameter. Template window size is $(3, 3)$; search window size is $(2 \times 3\sigma_s + 1, 2 \times 3\sigma_s + 1)$; and image size is 768×512.

(a) Computational time (proposed) [ms].

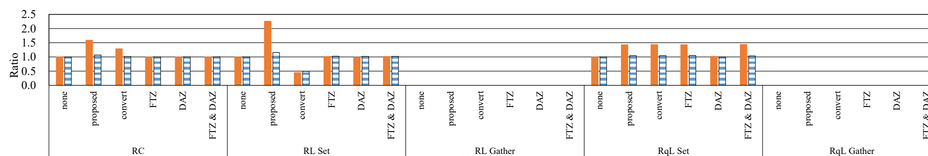
σ_s \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
4	40.20	40.05	39.92
8	133.61	133.02	132.85
16	496.86	496.57	495.89

(b) Computational time (none) [ms].

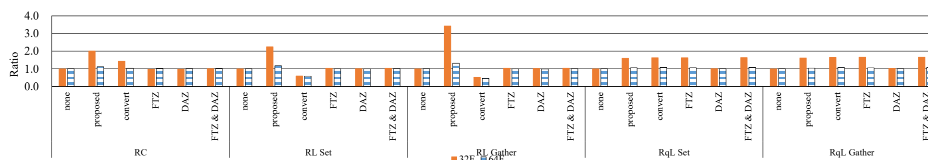
σ_s \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
4	99.14	84.85	82.85
8	340.29	311.72	301.38
16	1166.17	1191.36	1163.34

(c) Speedup ratio.

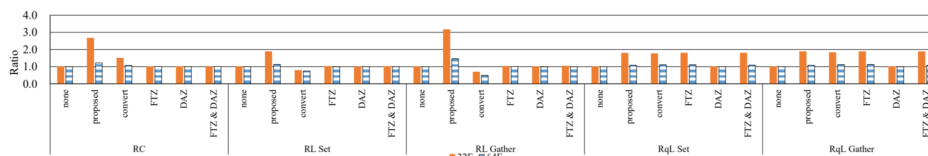
σ_s \ h	$4\sqrt{2}$	$8\sqrt{2}$	$16\sqrt{2}$
4	2.47	2.12	2.08
8	2.55	2.34	2.27
16	2.35	2.40	2.35



(a) SSE.

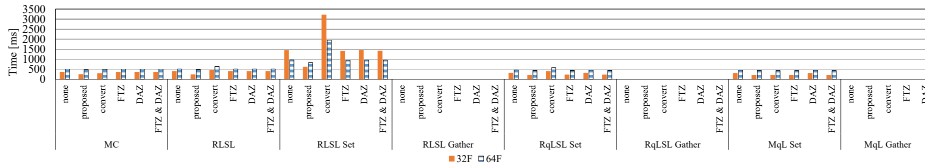


(b) AVX/AVX2.

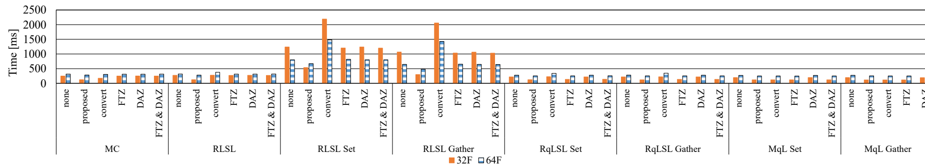


(c) AVX512.

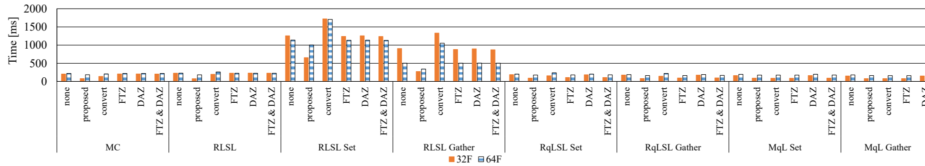
Figure 3.8: Speedup ratio of non-local means filter on Intel Core i9 7980XE. The speedup ratio is shown regarding single precision (32F) and double precision (64F) floating point numbers. If the ratio exceeds one, all implementation of the method are faster than the straightforward implementation (none). $h = 4\sqrt{2}$; template window size is $(3, 3)$, and search window size is $(37, 37)$. Image size is 768×512.



(a) SSE.

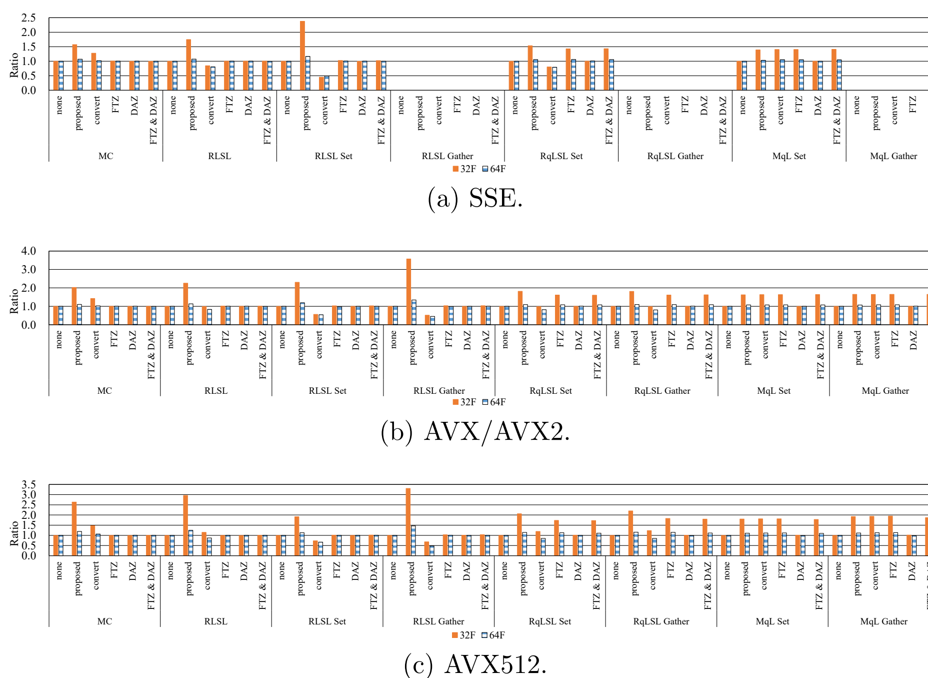


(b) AVX/AVX2.



(c) AVX512.

Figure 3.9: Computational time of bilateral non-local means filter on Intel Core i9 7980XE. The computational times are shown in terms of single precision (32F) and double precision (64F) floating point numbers. $\sigma_s = 6$, $h = 4\sqrt{2}$; template window size is $(3, 3)$, and search window size is $(2 \times 3\sigma_s + 1, 2 \times 3\sigma_s + 1)$. Image size is 768×512.

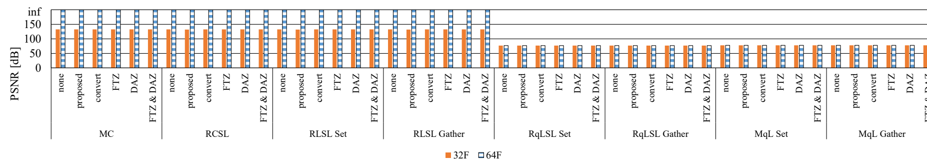


(a) SSE.

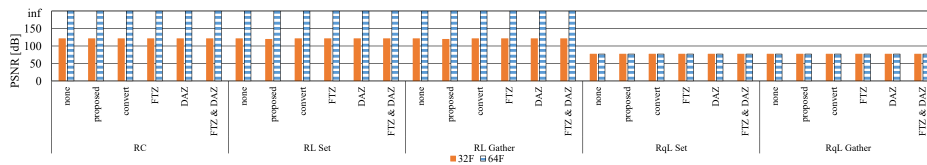
(b) AVX/AVX2.

(c) AVX512.

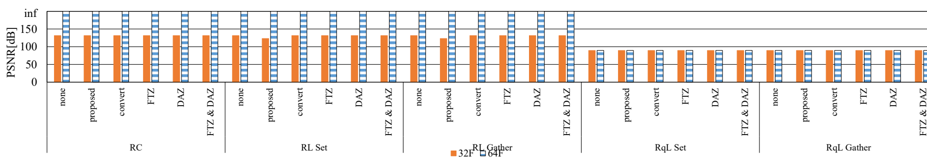
Figure 3.10: Speedup ratio of bilateral non-local means filter on Intel Core i9 7980XE. The speedup ratio is shown regarding single precision (32F) and double precision (64F) floating point numbers. If the ratio exceeds one, all implementation of the method are faster than the straightforward implementation (none). $\sigma_s = 6$, $h = 4\sqrt{2}$, template window size is $(3, 3)$, and search window size is $(2 \times 3\sigma_s + 1, 2 \times 3\sigma_s + 1)$. Image size is 768×512.



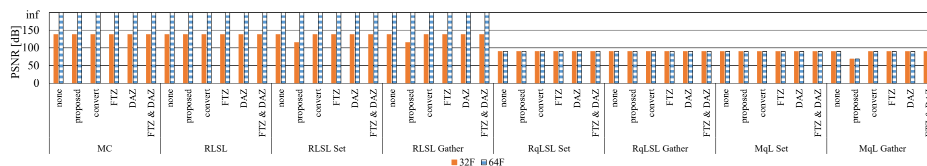
(a) Bilateral filter.



(b) Gaussian range filter.



(c) Non-local means filter.



(d) Bilateral non-local means filter.

Figure 3.11: PSNRs of the bilateral filter, Gaussian range filter, non-local means filter and bilateral non-local means filter. Note that the maximal value in (**a**–**d**) is infinity.

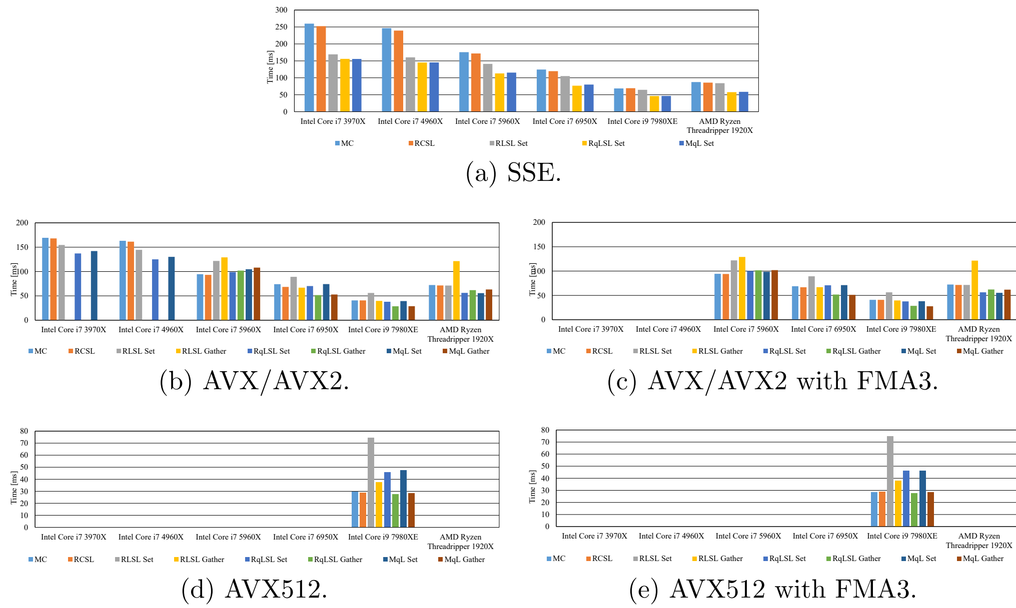


Figure 3.12: Computational time of the bilateral filter in various CPU microarchitectures. $\sigma_r = 4$, $\sigma_s = 6$ and $r = 3\sigma_s$. Image size is 768×512.

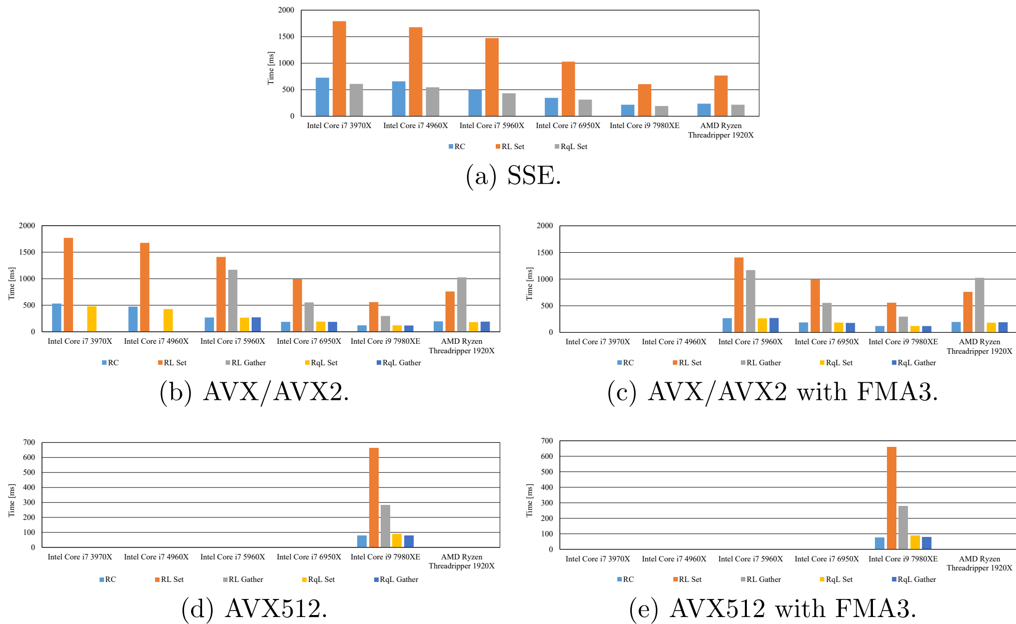
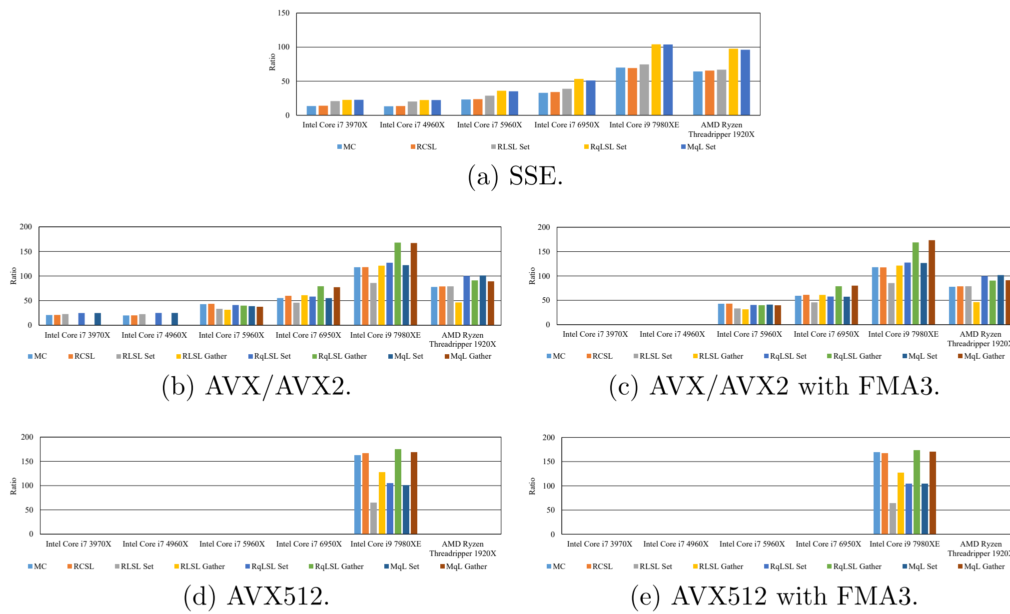


Figure 3.13: Computational time of non-local means filter in various CPU microarchitectures. $h = 4\sqrt{2}$; template window size is $(3, 3)$; and search window size is $(37, 37)$. Image size is 768×512.



(a) SSE.

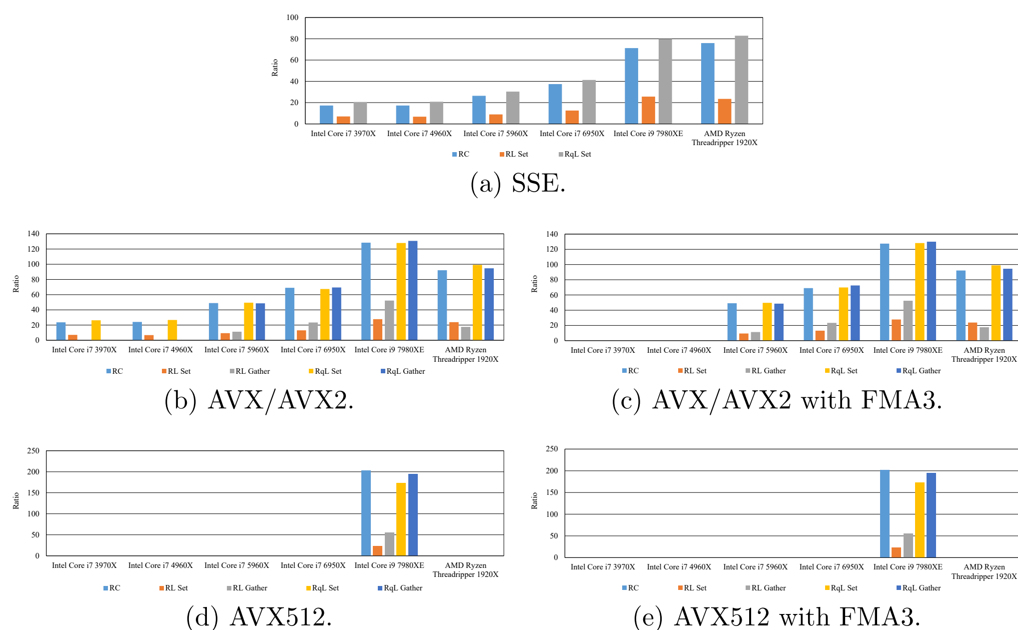
(b) AVX/AVX2.

(c) AVX/AVX2 with FMA3.

(d) AVX512.

(e) AVX512 with FMA3.

Figure 3.14: Speedup ratio of bilateral filter in various CPU microarchitectures. If the ratio exceeds one, the implementation is faster than a scalar implementation for all CPU microarchitectures. Note that the scalar implementation is parallelized using multi-core. $\sigma_r = 4$, $\sigma_s = 6$ and $r = 3\sigma_s$. Image size is 768×512.



(a) SSE.

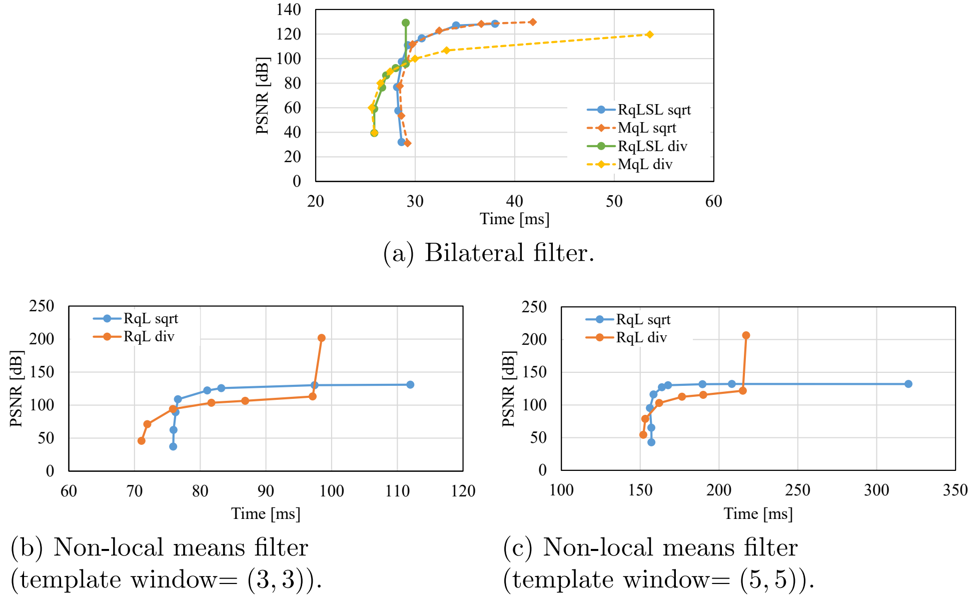
(b) AVX/AVX2.

(c) AVX/AVX2 with FMA3.

(d) AVX512.

(e) AVX512 with FMA3.

Figure 3.15: Speedup ratio of non-local means filter in various CPU microarchitectures. If the ratio exceeds one, the implementation is faster than a scalar implementation for each CPU microarchitecture. Note that the scalar implementation is parallelized using multi-core. $h = 4\sqrt{2}$; template window size is $(3, 3)$; and search window size is $(37, 37)$. Image size is 768×512.

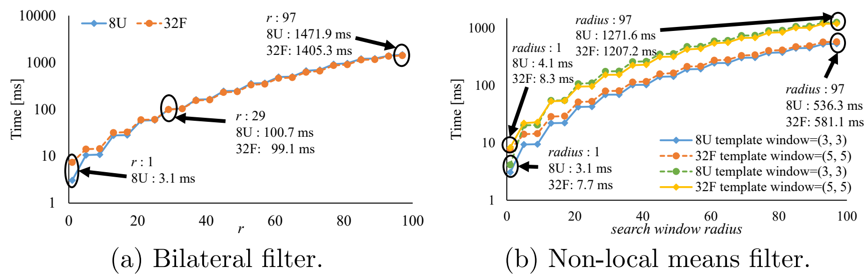


(a) Bilateral filter.

(b) Non-local means filter (template window= $(3,3)$).

(c) Non-local means filter (template window= $(5,5)$).

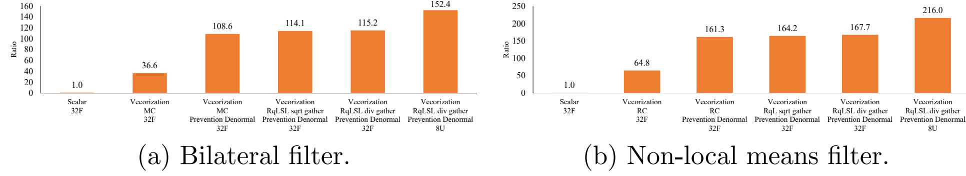
Figure 3.16: PSNR vs. computational time in quantization range/merged LUT. In sqrt and div, the quantization function is square root and division, respectively. These results were obtained on an Intel Core i9 7980XE. $\sigma_r = 4$, $\sigma_s = 6$, $h = 4\sqrt{2}$; and search window size is $(37, 37)$. Image size is 768×512.



(a) Bilateral filter.

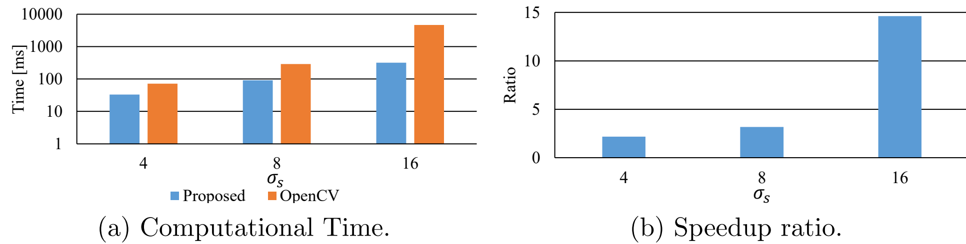
(b) Non-local means filter.

Figure 3.17: Computational time when using unsigned char (8U) and single precision floating point number (32F) with respect to kernel radius. Note that the computational time is plotted on the logarithmic scale. These results were obtained on an Intel Core i9 7980XE. $\sigma_r = 4$, $\sigma_s = 4$; and image size is 768×512.



(a) Bilateral filter. (b) Non-local means filter.

Figure 3.18: Speedup ratios of various proposed implementation approaches and that of scalar implementation. The types of implementation considered herein are parallelized using multi-cores. These results were obtained on an Intel Core i9 7980XE. $\sigma_r = 4$, $\sigma_s = 4$, $r = 3\sigma_s$; and image size is 768×512.



(a) Computational Time. (b) Speedup ratio.

Figure 3.19: Computational time and speedup ratio of fastest implementation and OpenCV implementation in bilateral filter. These results were calculated using an Intel Core i9 7980XE. Note that the computational time is plotted on the logarithmic scale. If the speedup ratio exceeds one, the fastest implementation is faster than the OpenCV implementation. $\sigma_r = 16$, $r = 3\sigma_s$; and image size is 768×512. For $\sigma_s = 4$, the PSNRs of the proposed method and OpenCV are 84.63 dB and 44.08 dB, respectively. For $\sigma_s = 8$, they are 85.45 dB and 43.55 dB, respectively. For $\sigma_s = 16$, they are 84.41 dB and 43.19 dB, respectively.

Chapter 4

Directional Cubic Convolution Interpolation with Edge Preserving Detail Enhancement

4.1 Introduction

Image upsampling is one of fundamental tools in image processing. Typical algorithms of the image upsampling are linear interpolation and bicubic interpolation. In addition, edge-directed algorithms are proposed [88–90]. The edge-directed algorithms interpolate pixels while considering the edge direction. The representatives of the algorithms include new edge directed interpolation (NEDI) [88] and directional cubic convolution interpolation (DCCI) [90]. Especially, DCCI achieves both high accuracy and high speed among these algorithms. These algorithms upsample an image in a stepwise manner. A part of pixels are interpolated based on a low-resolution image at first, and then other pixels are interpolated based on interpolated pixel and low-resolution image. The algorithms have a high accuracy of upsampling and suppress ringing of diagonal edges. However, the algorithms limit upsampling rate to exponent of 2.

In image upsampling, an input image often lack high-frequency signal; thus, detail enhancement is used for supplementing high-frequency signal. The detail enhancement emphasizes edges of the image for amplifying high-frequency signals. In its processing, detail and base signals are separated by using smoothing filtering and the detail signals are amplified. The output image is obtained by adding amplified detail signals and base signals. If 4- or 8-neighbor average filtering is employed as the smoothing filtering, it is classical unsharp mask filtering. The drawback of the unsharp mask

filtering is that halo and gradient reversal artifacts occurs on the contour in the enhanced image when the kernel radius of smoothing filtering is large. By using edge-preserving filtering as the smoothing filtering, it avoids the halo and gradient reversal artifacts [17, 91].

The edge-directed upsampling has high quality for images, in which there are large edges and not textures, e. g., cartoon images. Therefore, the upsampling can accurately upsample the base signals of the detail enhancement. It is suggested that we can achieve high accurate upsampling by separately upsampling these signals after the input image is separated the base and detail signals by using the edge preserving filtering. Moreover, the approach can be obtained the detail signals; thus, the detail enhancement can be performed. In this section, we propose a framework that can handle simultaneous processing of upsampling and detail enhancement.

4.2 Proposed Framework

Figure 4.1 shows the flowchart of the proposed framework. Firstly, the base signal is filtered input image by using the guided image filtering [17] and then the detail signal is a difference between an input image and the base signal. Secondly, the signals are upsampling with the DCCI. Finally, the detail signal is amplified and added it to the base signal. The proposed framework can simultaneously achieve the detail enhancement and upsampling.

The advantages of the proposed framework are that we can achieve high accuracy upsampling. The base signal is separated by the detail signal and then only includes large edges by reducing the textures; hence, the DCCI can high accurate upsampling. Therefore, the proposed framework can achieve higher accuracy upsampling than classical and conventional upsampling when the degree of amplification is 1. In addition, the detail enhancement can be performed by controlling the degree of amplification.

In the detail enhancement, the edge-preserving filtering filters the original size input image filters. In the proposed framework, the low-resolution image is filtered; thus, the computational cost of filtering is reduced. Meanwhile, the proposed framework twice employs DCCI on the base and detail signals. The computational cost of the DCCI is larger than that of classical upsampling algorithms, but that of DCCI is smaller than that of the edge-preserving filtering. In other words, the proposed framework dare downsample the input image so that the framework can accelerate the detail enhancement.

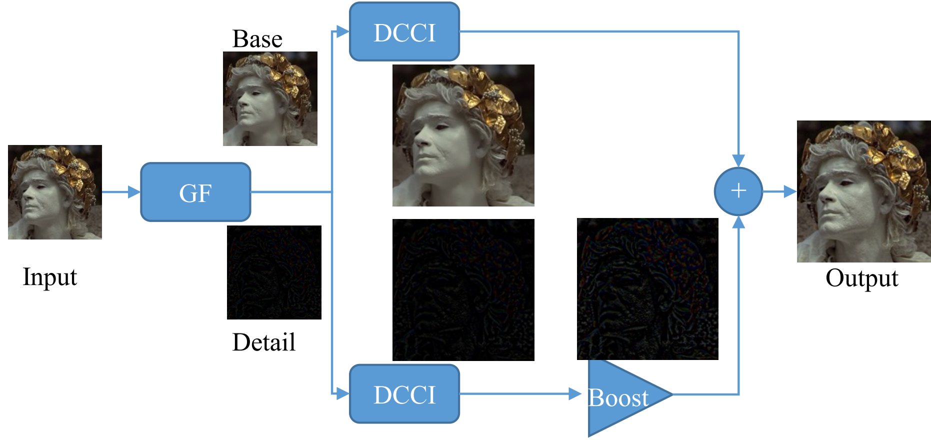


Figure 4.1: Flowchart. GF in this figure is guided filtering.

4.3 Experimental Results and Discussion

In the experiments, the input image is the downsampled original image. The upsampling rate was 2, 4, and 8-power. In the upsampling experiment, we measured PSNR between the upsampled image and the original image as correct image. The upsampling paramters are follows: the kernel radius of the guided filtering is 1 when the upsampling rate is 2 and 4. When the upsampling rate is 8, the radius is 2. The smoothing parameter ϵ is 0.0099, 0.0115, and 0.0135 when upsampling rate is 2, 4, and 8, respectively. The threshold T, which is edge detection parameter, was set to 1 when the rate was 2, and set to 1.1 when it was 4 and 8. These parameters were used in all images. $T = 1.15$ is appropriate for upsampling [90] but it was better to lower than 1.15 when separating the base and detail signals in the experiments.

In the experiment of the detail enhancement, the correct image is detail enhanced image, which is enhanced the original image. The parameters were assumed known. The kernel radius of the guided filtering is 2 when the upsampling rate is 2, and the radius is 1 when the rate is 4 and 8. The smoothing parameter ϵ is 0.0025. T was set to 1.15, and the degree of amplification, which is the parameter of amplifying the detail signal, was set to 2.

Table 4.1-4.3 show results of the upsampling and detail enhancement. The accuracy of the proposed framework is the highest of the cubic convolution interpolation (CCI) and DCCI in all images and all upsampling rate. The difference of PSNR between the proposed framework and conventional methods are 0.006dB to 0.08dB. Also, in the results of detail enhancement, the proposed framework can almost approximate the upsampling results as

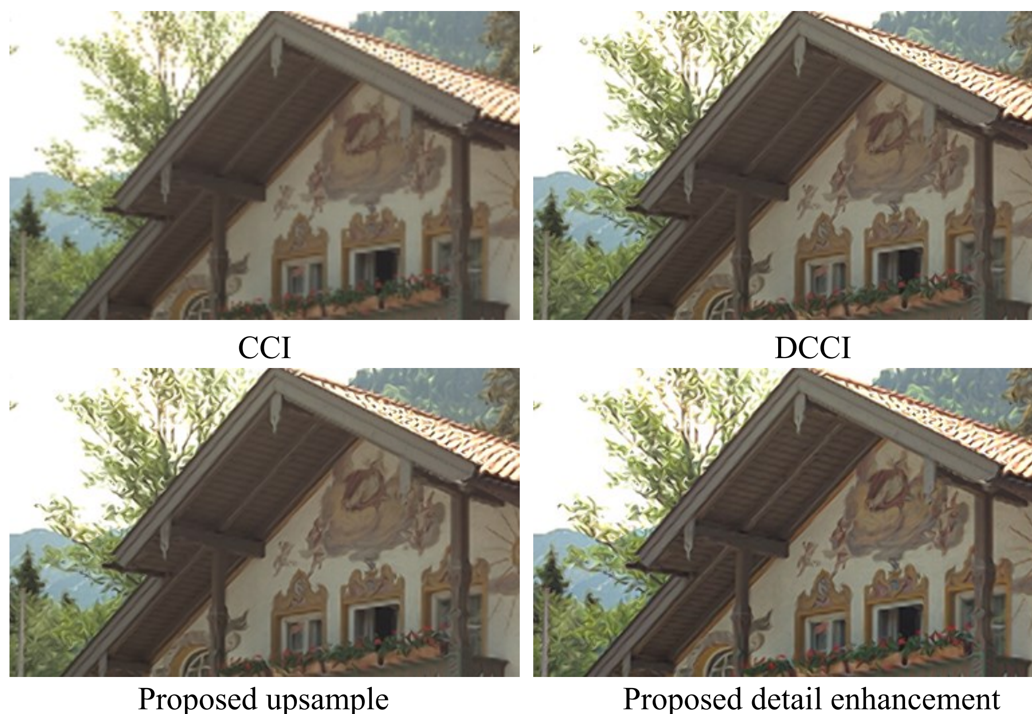


Figure 4.2: Resulting images.

the same accuracy. The result image is shown in the Figure 4.2 and 4.3. The red circle in Figure 4.3 is the part where the difference of each method appears. The proposed methods can be restored high-frequency signal than the DCCI. For example, we can see the effect at the roof part. The proposed framework correctly reproduces diagonal edges than DCCI slightly.

We also verified the case of using bilateral filtering as the smoothing filtering, but the guided filtering was more suitable for this purpose and the results were also better than the bilateral filtering. Because the bilateral filtering does not consider the gradient in images, it converges to one color and stair effect like postalization tends to occur. On the other hand, the guided filtering can be smoothed considering the gradient. Therefore, the suitable separated signal can be obtained by using the guided filtering.

4.4 Conclusions

In this section, we propose a framework that can handle simultaneous processing of directional cubic convolution interpolation and detail enhancement. The proposed framework employs the guided filtering on to separate the base and detail signals and these signals are upsampled by DCCI individ-

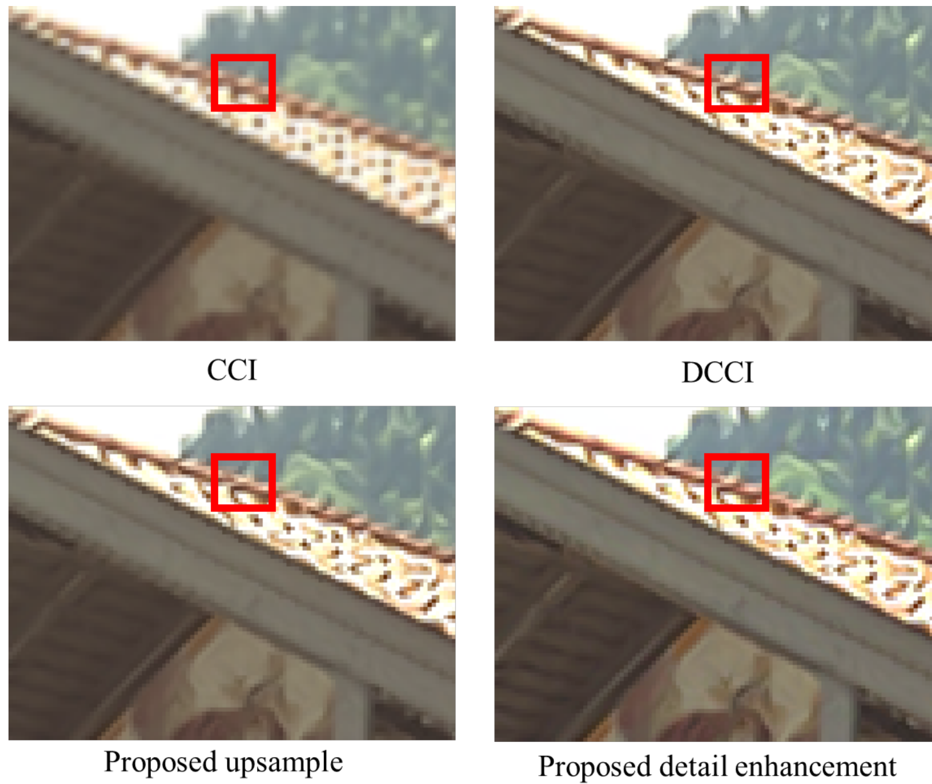


Figure 4.3: Resulting images (roof).

ually. The DCCI is appropriate for upsampling the separated signals. The proposed framework can perform the detail enhancement simultaneously by amplifying the detail signal. The results are summarized as follows:

1. In image upsampling, the proposed framework is the highest accuracy of cubic convolution interpolation and directional cubic convolution interpolation. The PSNR of the proposed framework is 0.006 dB to 0.008 dB higher than that of conventional algorithms.

2. In detail enhancement, the proposed framework obtains the same accuracy of the image upsampling.

3. The proposed framework achieve high accuracy upsampling and detail enhancement simultaneously.

Table 4.1: PSNR of upsampling and detail enhancement results. The upsample ratio is 2.

No	Upsample			Enhance
	CCI	DCCI	Proposed	Proposed
1	23.75	24.57	**24.65**	24.43
2	30.79	31.99	**32.09**	30.22
3	32.01	33.41	**33.48**	32.21
4	30.92	32.74	**32.85**	31.19
5	23.78	26.24	**26.33**	26.00
6	25.19	26.05	**26.12**	25.65
7	29.54	33.49	**33.57**	32.71
8	20.68	22.22	**22.26**	22.00
9	29.10	31.73	**31.81**	30.53
10	29.30	31.36	**31.45**	30.56
11	26.54	27.93	**28.03**	27.09
12	30.62	31.52	**31.57**	30.72
13	21.64	22.39	**22.46**	22.27
14	26.30	28.24	**28.35**	27.53
15	30.02	30.22	**30.28**	30.92
16	28.81	29.68	**29.76**	28.53
17	29.37	31.29	**31.36**	30.50
18	25.60	26.65	**26.74**	26.40
19	25.31	27.06	**27.11**	26.35
20	28.63	31.07	**31.15**	30.17
21	25.75	27.21	**27.29**	26.76
22	27.75	29.23	**29.34**	28.18
23	31.61	35.37	**35.46**	33.73
24	24.39	25.31	**25.38**	25.15
Average	27.39	29.04	**29.12**	28.33

Table 4.2: PSNR of upsampling and detail enhancement results. The upsample ratio is 4.

No	Upsample			Enhance
	CCI	DCCI	Proposed	Proposed
1	19.04	20.43	**20.49**	20.03
2	26.15	27.89	**27.95**	26.24
3	26.98	29.12	**29.20**	27.90
4	25.84	28.03	**28.13**	26.69
5	18.78	20.90	**20.97**	20.44
6	21.02	22.32	**22.41**	21.70
7	23.31	26.94	**26.96**	26.19
8	16.21	17.97	**18.01**	17.62
9	23.57	26.16	**26.19**	25.22
10	24.29	26.64	**26.71**	25.66
11	22.21	23.85	**23.91**	22.94
12	25.27	27.82	**27.87**	26.73
13	17.76	18.97	**19.05**	18.57
14	21.44	23.48	**23.57**	22.65
15	24.13	26.21	**26.27**	26.72
16	24.90	26.25	**26.37**	24.98
17	24.38	26.73	**26.76**	25.95
18	21.00	22.63	**22.72**	22.14
19	20.40	21.98	**22.02**	21.39
20	22.78	25.99	**26.05**	25.50
21	21.08	22.96	**23.04**	22.37
22	23.26	24.99	**25.08**	23.86
23	25.97	28.59	**28.64**	27.54
24	20.13	21.55	**21.62**	21.18
Average	22.50	24.52	**24.58**	23.76

Table 4.3: PSNR of upsampling and detail enhancement results. The upsample ratio is 8.

	Upsample			Enhance
No	CCI	DCCI	Proposed	Proposed
1	17.40	18.58	**18.63**	18.18
2	23.96	25.95	**26.02**	24.51
3	24.29	26.88	**26.93**	25.77
4	22.65	25.20	**25.29**	24.09
5	16.39	18.09	**18.17**	17.71
6	19.29	20.77	**20.86**	20.23
7	19.92	22.15	**22.22**	21.54
8	14.03	15.63	**15.67**	15.39
9	20.83	22.85	**22.91**	22.20
10	21.84	23.93	**23.99**	23.12
11	20.00	21.78	**21.85**	21.01
12	22.15	25.36	**25.40**	24.46
13	16.35	17.45	**17.53**	17.13
14	18.92	20.85	**20.95**	20.16
15	20.63	24.72	**24.79**	24.23
16	23.25	24.80	**24.91**	23.60
17	21.35	24.10	**24.19**	23.38
18	19.19	20.53	**20.62**	20.04
19	18.46	19.99	**20.05**	19.51
20	19.89	23.33	**23.38**	22.96
21	18.97	20.73	**20.80**	20.24
22	21.26	22.75	**22.84**	21.78
23	22.87	25.38	**25.43**	24.59
24	18.53	19.78	**19.86**	19.41
Average	20.10	22.15	**22.22**	21.47

Chapter 5

Conclusion

In this dissertation, we aim acceleration of the edge-preserving filtering and organize cyclopaedically effective implementation on CPU microarchitectures. Also, we focus on acceleration and high accuracy of upsampling and detail enhancement, which are one of application in the edge-preserving filtering.

In Chapter 2, we summarize a taxonomy of vectorized programming for FIR image filtering. We also propose a new vectorization pattern of vectorized programming, which we call loop vectorization. These vectorization patterns are combined with an acceleration method of kernel subsampling for general FIR filters. The experimental results indicate that the patterns are appropriate for FIR filtering, and a new pattern with kernel subsampling can be profitably used for Gaussian range filtering (GRF), bilateral filtering (BF), adaptive Gaussian filtering (AGF), randomly-kernel-subsampled Gaussian range filtering (RKS-GRF), randomly-kernel-subsampled bilateral filtering (RKS-BF), and randomly-kernel-subsampled adaptive Gaussian filtering (RKS-AGF).

In Chapter 3, we propose methods to accelerate bilateral filtering and non-local means filtering. The proposed methods prevent the occurrence of denormal numbers. Denormal numbers are special floating point numbers defined in IEEE standard 754, and they are considerably smaller than normal numbers. The processing of denormal numbers has a large computational cost. Moreover, we verify various types of vectorized implementation on the latest CPU microarchitectures. Experimental results show that the proposed methods are faster than implementation that does not prevent the occurrence of denormal numbers. Also, the proposed implementation of the edge-preserving filtering is efficient.

In Chapter 4, we proposed a framework that can process simultaneous processing of directional cubic convolution interpolation and detail enhancement. Experimental results show that the proposed framework achieves high

accuracy upsampling and detail enhancement simultaneously.

In the future work, the acceleration approaches should be extended to other architectures, such as GPU and FPGA. Future work also includes acceleration of various applications, which employ the edge-preserving filtering.

References

[1] C. Tomasi and R. Manduchi, “Bilateral filtering for gray and color images,” in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 1998, pp. 839–846.

[2] P. Kornprobst and J. Tumblin, *Bilateral filtering: Theory and applications*. Now Publishers Inc., 2009.

[3] A. Buades, B. Coll, and J. M. Morel, “A non-local algorithm for image denoising,” in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005, pp. 60–65.

[4] 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.

[5] ——, “Guided image filtering,” in *Proc. European Conference on Computer Vision (ECCV)*, 2010.

[6] 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.

[7] W. Kang, S. Yu, D. Seo, J. Jeong, and J. Paik, “Push-broom-type very high-resolution satellite sensor data correction using combined wavelet-fourier and multiscale non-local means filtering,” *Sensors*, vol. 15, no. 9, pp. 22 826–22 853, 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/9/22826

[8] F. Durand and J. Dorsey, “Fast bilateral filtering for the display of high-dynamic-range images,” *ACM Trans. on Graphics*, vol. 21, no. 3, pp. 257–266, 2002.

[9] S. Bae, S. Paris, and F. Durand, “Two-scale tone management for photographic look,” *ACM Trans. on Graphics*, vol. 25, no. 3, pp. 637–645, 2006.

[10] R. Fattal, M. Agrawala, and S. Rusinkiewicz, “Multiscale shape and detail enhancement from multi-light image collections,” *ACM Trans. on Graphics*, vol. 26, no. 3, 2007.

[11] L. Li, Y. Si, and Z. Jia, “Remote sensing image enhancement based on non-local means filter in nsct domain,” *Algorithms*, vol. 10, no. 4, 2017. [Online]. Available: http://www.mdpi.com/1999-4893/10/4/116

[12] Y. Mori, N. Fukushima, T. Yendo, T. Fujii, and M. Tanimoto, “View generation with 3d warping using depth information for ftv,” *Signal Processing: Image Communication*, vol. 24, no. 1-2, pp. 65 – 72, 2009.

[13] G. Petschnigg, M. Agrawala, H. Hoppe, R. Szeliski, M. Cohen, and K. Toyama, “Digital photography with flash and no-flash image pairs.” *ACM Trans. on Graphics*, vol. 23, no. 3, pp. 664–672, 2004.

[14] E. Eisemann and F. Durand, “Flash photography enhancement via intrinsic relighting.” *ACM Trans. on Graphics*, vol. 23, no. 3, pp. 673–678, 2004.

[15] J. Kopf, M. F. Cohen, D. Lischinski, and M. Uyttendaele, “Joint bilateral upsampling,” *ACM Transactions on Graphics*, vol. 26, no. 3, 2007.

[16] D. Zhou, R. Wang, J. Lu, and Q. Zhang, “Depth image super resolution based on edge-guided method,” *Applied Sciences*, vol. 8, no. 2, 2018. [Online]. Available: http://www.mdpi.com/2076-3417/8/2/298

[17] K. He, J. Shun, and X. Tang, “Guided image filtering,” in *Proc. European Conference on Computer Vision (ECCV)*, 2010.

[18] N. Kodera, N. Fukushima, and Y. Ishibashi, “Filter based alpha matting for depth image based rendering,” in *Proc. Visual Communications and Image Processing (VCIP)*, Nov. 2013.

[19] K. He, J. Sun, and X. Tang, “Single image haze removal using dark channel prior.” in *Proc. Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 2341–2353.

[20] 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.

[21] T. Matsuo, N. Fukushima, and Y. Ishibashi, "Weighted joint bilateral filter with slope depth compensation filter for depth map refinement," in *Proc. International Conference on Computer Vision Theory and Applications (VISAPP)*, 2013.

[22] A. V. Le, S.-W. Jung, and C. S. Won, "Directional joint bilateral filter for depth images," *Sensors*, vol. 14, no. 7, pp. 11 362–11 378, 2014. [Online]. Available: http://www.mdpi.com/1424-8220/14/7/11362

[23] S. Liu, P. Lai, D. Tian, and C. W. Chen, "New depth coding techniques with utilization of corresponding video," *IEEE Trans. on Broadcasting*, vol. 57, no. 2, pp. 551–561, June 2011.

[24] N. Fukushima, T. Inoue, and Y. Ishibashi, "Removing depth map coding distortion by using post filter set," in *Proc. IEEE International Conference on Multimedia and Expo (ICME)*, 2013.

[25] N. Fukushima, S. Fujita, and Y. Ishibashi, "Switching dual kernels for separable edge-preserving filtering," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[26] T. Q. Pham and L. J. V. Vliet, "Separable bilateral filtering for fast video preprocessing." in *Proc. International Conference on Multimedia and Expo (ICME)*, 2005.

[27] J. Chen, S. Paris, and F. Durand, "Real-time edge-aware image processing with the bilateral grid," *ACM Trans. on Graphics*, vol. 26, no. 3, 2007.

[28] K. Sugimoto, N. Fukushima, and S. Kamata, "Fast bilateral filter for multichannel images via soft-assignment coding," in *Proc. APSIPA ASC*, 2016.

[29] K. Sugimoto and S. Kamata, "Compressive bilateral filtering," *IEEE Trans. on Image Processing*, vol. 24, no. 11, pp. 3357–3369, 2015.

[30] S. Paris and F. Durand, "A fast approximation of the bilateral filter using a signal processing approach," *International Journal of Computer Vision*, vol. 81, no. 1, pp. 24–52, 2009.

[31] K. N. Chaudhury, "Acceleration of the shiftable $o(1)$ algorithm for bilateral filtering and nonlocal means," *IEEE Transactions on Image Processing*, vol. 22, no. 4, pp. 1291–1300, April 2013.

[32] K. Chaudhury, "Constant-time filtering using shiftable kernels," *IEEE Signal Processing Letters*, vol. 18, no. 11, pp. 651–654, 2011.

[33] K. Chaudhury, D. Sage, and M. Unser, "Fast o(1) bilateral filtering using trigonometric range kernels," *IEEE Trans. on Image Processing*, vol. 20, no. 12, pp. 3376–3382, 2011.

[34] A. Adams, N. Gelfand, J. Dolson, and M. Levoy, "Gaussian kd-trees for fast high-dimensional filtering," *ACM Trans. on Graphics*, vol. 28, no. 3, 2009.

[35] A. Adams, J. Baek, and M. A. Davis, "Fast high-dimensional filtering using the permutohedral lattice," *Computer Graphics Forum*, vol. 29, no. 2, pp. 753–762, 2010.

[36] F. Porikli, "Constant time o(1) bilateral filtering." in *Proc. Computer Vision and Pattern Recognition (CVPR)*, 2008.

[37] Q. Yang, K. H. Tan, and N. Ahuja, "Real-time o(1) bilateral filtering." in *Proc. Computer Vision and Pattern Recognition (CVPR)*, 2009, pp. 557–564.

[38] G. E. Moore, "Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff." *IEEE Solid-State Circuits Society Newsletter*, vol. 11, no. 3, pp. 33–35, Sept 2006.

[39] E. Rotem, R. Ginosar, A. Mendelson, and U. C. Weiser, "Power and thermal constraints of modern system-on-a-chip computer," in *19th International Workshop on Thermal Investigations of ICs and Systems (THERMINIC)*, Sept 2013, pp. 141–146.

[40] H. Sutter, "The concurrency revolution," in *C/C++ Users Journal*, 2 2005.

[41] M. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. on Computers*, vol. C-21, no. 9, pp. 948–960, 1972.

[42] C. J. Hughes, "Single-instruction multiple-data execution," *Synthesis Lectures on Computer Architecture*, vol. 10, no. 1, pp. 1–121, 2015.

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

[44] "Intel architecture instruction set extensions and future features programming reference," https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf, accessed: 2018-10-01.

[45] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," *SIGPLAN Not.*, vol. 33, no. 5, pp. 38–49, May 1998.

[46] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proc. International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software (CC'11/ETAPS'11)*, 2011, pp. 225–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987255

[47] T. Saegusa, T. Maruyama, and Y. Yamaguchi, "How fast is an fpga in image processing?" in *Proc. International Conference on Field Programmable Logic and Applications*, Sept 2008, pp. 77–82.

[48] S. Asano, T. Maruyama, and Y. Yamaguchi, "Performance comparison of fpga, gpu and cpu in image processing," in *Proc. International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 126–131.

[49] T. Kurafuji, M. Haraguchi, M. Nakajima, T. Nishijima, T. Tanizaki, H. Yamasaki, T. Sugimura, Y. Imai, M. Ishizaki, T. Kumaki, K. Murata, K. Yoshida, E. Shimomura, H. Noda, Y. Okuno, S. Kamijo, T. Koide, H. J. Mattausch, and K. Arimoto, "A scalable massively parallel processor for real-time image processing," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 10, pp. 2363–2373, Oct 2011.

[50] K. E. Batcher, "Sorting networks and their applications," in *Proc. spring joint computer conference (SJCC)*. ACM, 1968, pp. 307–314.

[51] C. A. R. Hoare, "Quicksort," *The Computer Journal*, vol. 5, no. 1, pp. 10–16, 1962.

[52] 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, pp. 511–518.

[53] R. C. Gonzalez and R. E. Woods, *Digital Image Processing.* Prentice Hall, 2008.

[54] S. Treitel and J. L. Shanks, "The design of multistage separable planar filters," *IEEE Transactions on Geoscience Electronics*, vol. 9, no. 1, pp. 10–27, Jan 1971.

[55] L. Lou, P. Nguyen, J. Lawrence, and C. Barnes, "Image perforation: Automatically accelerating image pipelines by intelligently skipping samples," *ACM Trans. Graph.*, vol. 35, no. 5, pp. 153:1–153:14, sep 2016.

[56] F. Banterle, M. Corsini, P. Cignoni, and R. Scopigno, "A low-memory, straightforward and fast bilateral filter through subsampling in spatial domain," in *Computer Graphics Forum*, vol. 31, no. 1. Wiley Online Library, 2012, pp. 19–32.

[57] R. Deriche, "Recursively implementating the gaussian and its derivatives," in *Proc. IEEE International Conference on Image Processing (ICIP)*, 1992, pp. 263–267.

[58] I. T. Young and L. J. Van Vliet, "Recursive implementation of the gaussian filter," *Signal processing*, vol. 44, no. 2, pp. 139–151, 1995.

[59] L. J. Van Vliet, I. T. Young, and P. W. Verbeek, "Recursive gaussian derivative filters," in *Proc. IEEE International Conference on Pattern Recognition*, vol. 1, 1998, pp. 509–514.

[60] W. M. Wells, "Efficient synthesis of gaussian filters by cascaded uniform filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 2, pp. 234–239, 1986.

[61] E. Elboher and M. Werman, "Cosine integral images for fast spatial and range filtering," in *Proc. IEEE International Conference on Image Processing*, 2011, pp. 89–92.

[62] K. Sugimoto and S. Kamata, "Fast gaussian filter with second-order shift property of dct-5," in *Proc. International Conference on Image Processing (ICIP)*, 2013, pp. 514–518.

[63] ——, "Efficient constant-time gaussian filtering with sliding dct/dst-5 and dual-domain error minimization," *ITE Transactions on Media Technology and Applications*, vol. 3, no. 1, pp. 12–21, 2015.

[64] P. Getreuer, "A survey of gaussian convolution algorithms," *Image Processing On Line*, vol. 2013, pp. 286–310, 2013.

[65] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis.* New York, NY, USA: Cambridge University Press, 2005.

[66] R. Motwani and P. Raghavan, *Randomized Algorithms.* New York, NY, USA: Cambridge University Press, 1995.

[67] R. L. Cook, "Stochastic sampling in computer graphics," *ACM Transactions on Graphics (TOG)*, vol. 5, no. 1, pp. 51–72, 1986.

[68] Y. Asahi, G. Latu, T. Ina, Y. Idomura, V. Grandgirard, and X. Garbet, "Optimization of fusion kernels on accelerators with indirect or strided memory access patterns," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 7, pp. 1974–1988, July 2017.

[69] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[70] K. Chellapilla, S. Puri, and P. Simard, "High performance convolutional neural networks for document processing," in *Prco. International Workshop on Frontiers in Handwriting Recognition*, Oct 2006.

[71] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, vol. abs/1410.0759, 2014.

[72] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *Proc. IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2017, pp. 19–24.

[73] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, nov 2004.

[74] G. Deng and L. Cahill, "An adaptive gaussian filter for noise reduction and edge detection," in *Proc. IEEE Nuclear Science Symposium and Medical Imaging Conference*, 1993, pp. 1615–1619.

[75] S. Bae and F. Durand, "Defocus magnification," in *Computer Graphics Forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 571–579.

[76] W. Zhang and W.-K. Cham, "Single image focus editing," in *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 2009, pp. 1947–1954.

[77] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[78] E. M. Schwarz, M. Schmookler, and S. D. Trong, "Fpu implementations with denormalized numbers," *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 825–836, July 2005.

[79] ——, "Hardware implementations of denormalized numbers," in *Proceedings 2003 16th IEEE Symposium on Computer Arithmetic*, June 2003, pp. 70–78.

[80] L. Zheng, H. Hu, and S. Yihe, "Floating-point unit processing denormalized numbers," in *2005 6th International Conference on ASIC*, vol. 1, Oct 2005, pp. 6–9.

[81] Y. S. Kim, H. Lim, O. Choi, K. Lee, J. D. K. Kim, and J. Kim, "Separable bilateral nonlocal means," in *Proc. International Conference on Image Processing (ICIP)*, Sep. 2011, pp. 1513–1516.

[82] M. D. Ercegovac and T. Lang, *Digital Arithmetic*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[83] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009. [Online]. Available: http://doi.acm.org/10.1145/1498765.1498785

[84] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[85] 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. [Online]. Available: http://www.mdpi.com/2076-3417/8/8/1235

[86] I. Telegraph and T. C. Committee, *CCITT Recommendation T.81: Terminal Equipment and Protocols for Telematic Services : Information Technology - Digital Compression and Coding of Continuous-tone Still Images - Requirements and Guidelines.* International Telecommunication Union, 1993.

[87] M. Domanski and K. Rakowski, "Near-lossless color image compression with no error accumulation in multiple coding cycles," in *CAIP*, ser. Lecture Notes in Computer Science, vol. 2124. Springer, 2001, pp. 85–91.

[88] X. Li and M. T. Orchard, "New edge-directed interpolation," *IEEE Transactions on Image Processing*, vol. 10, no. 10, pp. 1521–1527, Oct 2001.

[89] A. Giachetti and N. Asuni, "Real-time artifact-free image upscaling," *IEEE Transactions on Image Processing*, vol. 20, no. 10, pp. 2760–2768, Oct 2011.

[90] D. Zhou, X. Shen, and W. Dong, "Image zooming using directional cubic convolution interpolation," *IET Image Processing*, vol. 6, no. 6, pp. 627–634, August 2012.

[91] Z. Farbman, R. Fattal, D. Lischinski, and R. Szeliski, "Edge-preserving decompositions for multi-scale tone and detail manipulation," *ACM Trans. on Graphics*, vol. 27, no. 3, 2008.

Acknowledgement

I am deeply grateful to Prof. Hiroshi Matsuo of Nagoya Institute of Technology for his support and advice. I would like to express my sincere gratitude to Prof. Norishige Fukushima of Nagoya Institute of Technology for his valuable guidance, support, and advice. Without his encouragement and guidance, this dissertation would not have materialized. I also appreciate feedbacks of this dissertation offered by Prof. Shoichi Saito and Prof. Hidekata Hontani of Nagoya Institute of Technology.

I am deeply grateful to Prof. Isao Yamada at National Institute of Technology, Gifu College. He provided me overall encouragement. I have greatly benefited from Prof. Takeshi Nakaya of Shinshu University.

I would like to express my sincere appreciation to President Yuuki Umemoto of Arc co. Ltd. for his support and encouragement. I also have greatly benefited from the member of Arc co. Ltd.

Special thanks to the member of Fukushima laboratory.

Finally, I would sincerely like to thank my family and my friends for their encouragement.

List of Publications

Journals

1. Y. Maeda, N. Fukushima, H. Matsuo, "Effective Implementation of Edge-Preserving Filtering on CPU Microarchitectures," Applied Sciences, vol. 8, no. 10, Oct. 2018.

2. Y. Maeda, N. Fukushima, H. Matsuo, "Taxonomy of Vectorization Patterns of Programming for FIR Image Filters Using Kernel Subsampling and New One," Applied Sciences, vol. 8, no. 8, July 2018.

3. Y. Maeda, T. Sasaki, M. Nakamura, N. Fukushima, H. Matsuo, "Directional Cubic Convolution Interpolation with Edge Preserving Detail Enhancement," (in Japanese), IEICE Trans. Information and Systems (Japanese Edition), vol. J100-D, no. 9, pp. 846–849, Sep. 2017.

4. T. Miyoshi, Y. Maeda, Y. Morita, Y. Ishibashi, K. Terashima, "Development of Haptic Network Game Based on Multi-lateral Tele-control Theory And Influence of Network Delay on Hptic Feeling," (in Japanese), Transactions of the Virtual Reality Society of Japan, vol. 19, no. 4, pp. 559-569, Dec. 2014.

5. I. Yamada, Y. Maeda, M. Onogi, T. Yoshida, Y. Ido, "Evaluation of signal detectability of a medical high resolution monitor and a multipurpose monitor using ROC analysis," (in Japanese), Memoirs of Gifu National College of Technology, vol. 47, pp. 11-15, Mar. 2012.

International Conference Proceedings

1. N. Fukushima, Y. Maeda, Y. Kawasaki, M. Nakamura, T. Tsumura, K. Sugimoto, S. Kamata, "Efficient Computational Scheduling of Box and Gaussian FIR Filtering for CPU Microarchitecture," Asia-Pacific

Signal and Information Processing Association Annual Summit and Conference (APSIPA), Oct. 2018.

2. Y. Murooka, Y. Maeda, M. Nakamura, T. Sasaki, N. Fukushima, "Principal Component Analysis for Acceleration of Color Guided Image Filtering," International Workshop on Frontiers of Computer Vision (IW-FCV), Feb. 2018.

3. K. Watanabe, Y. Maeda, N. Fukushima, "Stability of Recursive Gaussian Filtering for Piecewise Linear Bilateral Filtering," International Workshop on Frontiers of Computer Vision (IW-FCV), Feb. 2018.

4. Y. Kawasaki, Y. Maeda, N. Fukushima, "Parallelized and Vectorized Implementation of DCT denoising with FMA instructions," International Workshop on Advanced Image Technology (IWAIT), Jan. 2018.

5. Y. Maeda, N. Fukushima, H. Matsuo, "Basic Study on Recognition of Seven-Segment LED Digits by Using Binary Template Matching," International Workshop on Advanced Image Technology (IWAIT), Jan. 2017.

6. K. Suzuki, Y. Maeda, Y. Ishibashi, N. Fukushima, "Improvement of operability in remote robot control with force feedback," IEEE Global Conference on Consumer Electronics (GCCE), Oct. 2015.

7. Y. Maeda, Y. Ishibashi, N. Fukushima, "QoE assessment of sense of presence in networked virtual environment with haptic and auditory senses: Influence of network delay," IEEE Global Conference on Consumer Electronics (GCCE), pp. 679-683, Oct. 2014.

8. Y. Maeda, N. Fukushima, N. Fukushima, S. Sugawara, "Contribution of olfactory, haptic, and auditory senses to sense of presence in virtual environments," IEEE International Communications Quality and Reliability (CQR) Workshop, May 2013.

Domestic Conferences Proceedings

1. Y. Murooka, Y. Maeda, N. Fukushima, "PCA for Acceleration of Guided Filtering," (in Japanese), IEICE Society Conference, Sep. 2018.

2. Y. Kawasaki, Y. Maeda, N. Fukushima, "Accelerate Frequency Filter by Post Scaling Type DCT," (in Japanese), IEICE Society Conference, Sep. 2018.

3. Y. Murooka, Y. Maeda, N. Fukushima, "Principal Component Analysis for Approximation of Guided Filtering," (in Japanese), Tokai-Section Joint Conference on Electrical, Electronics, Information, and Related Engineering, Sep. 2018.

4. Y. Kawasaki, Y. Maeda, N. Fukushima, "Redundant Frequency Filter by Post Scaling Type DCT," (in Japanese), Tokai-Section Joint Conference on Electrical, Electronics, Information, and Related Engineering, Sep. 2018.

5. A. Ishikawa, Y. Maeda, N. Fukushima, "Optimization of computational scheduling of Guided Filtering in Halide," (in Japanese), Technical report of IEICE. IE, June 2018.

6. T. Sasaki, Y. Maeda, M. Nakamura, N. Fukushima, "Optimization of Computational Scheduling for Acceleration of Directional Cubic Convolution Interpolation," (in Japanese), Technical report of IEICE. SIP, Mar. 2018.

7. T. Sasaki, Y. Maeda, M. Nakamura, N. Fukushima, "Simultaneous Processing of Image Upsampling and Detail Enhancement with Directional Filter," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2017.

8. Y. Maeda, N. Fukushima, "Acceleration of FIR Filtering According to CPU Microarchitecture," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2017.

9. Y. Murooka, T. Sasaki, S. Yamashita, Y. Maeda, N. Fukushima, "Accelerated Local Linear Filtering with Chroma Subsampling," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2017.

10. K. Watanabe, Y. Maeda, N. Fukushima, "Improvement in Accuracy of Constant Time Bilateral Filtering by Switching IIR Gaussian Filtering Considering Stability," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2017.

11. Y. Kawasaki, Y. Maeda, N. Fukushima, "Fast DCT Denoising Using Post-scaling DCT of FMA Instrucitons," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2017.

12. Y. Murooka, T. Sasaki, M. Nakamura, Y. Maeda, N. Fukushima, "TECHNICAL REPORT OF GUIDED FILTER WITH CHROMA SUBSAMPLING AND COVARIANCE," (in Japanese), IEICE Society Conference, Sep. 2017.

13. K. Watanabe, Y. Maeda, N. Fukushima, "Boundary Processing in Double Precision for Stabilizing IIR Filter in Single Precision," (in Japanese), Forum on information technology (FIT), Sep. 2017.

14. Y. Murooka, T. Sasaki, M. Nakamura, Y. Maeda, N. Fukushima, "Technical Report of Downsampling for Guided Filter by use of Color Conversion," (in Japanese), Tokai-Section Joint Conference on Electrical, Electronics, Information, and Related Engineering, Sep. 2017.

15. K. Watanabe, Y. Maeda, N. Fukushima, "Double Precision-Based Boundary Processing for Stable IIR Filtering in Single Precision for Real-Time O(1) Bilateral Filtering," (in Japanese), Tokai-Section Joint Conference on Electrical, Electronics, Information, and Related Engineering, Sep. 2017.

16. Y. Maeda, S. Yamashita, M. Nakamura, N. Fukushima, H. Matsuo, "Image Rearrangement for Vectorized Operations in FIR Filtering," (in Japanese), Technical report of IEICE. IE, May 2017.

17. M. Nakamura, Y. Maeda, N. Fukushima, "Acceleration of Guided Filtering for Cache Efficiency," (in Japanese), National Conventions of IPSJ, Mar. 2017.

18. Y. Maeda, N. Fukushima, H. Matsuo, "Basic Study on Seven-Segment LED Digits Recognition by Using Binary Template Matching," (in Japanese), Picture Coding Symposium of Japan and Image Media Processing Symposium, Nov. 2016.

19. Y. Maeda, Y. Ishibashi, N. Fukushima, K. Suzuki, "Work Efficiency Comparison of Haptic Control Schemes in Remote Robot Control," (in Japanese), Tokai-Section Joint Conference on Electrical, Electronics, Information, and Related Engineering, Sep. 2015.

20. K. Suzuki, Y. Maeda, Y. Ishibashi, N. Fukushima, "Influence of network delay on pen strokes in bilateral remote robot control with haptic sence ," (in Japanese), Technical report of IEICE. CQ, Aug. 2015.

21. K. Suzuki, Y. Maeda, Y. Ishibashi, N. Fukushima, "QoE Assessment of Operability in Remote Robot Control with Force Feedback," (in Japanese), Technical report of IEICE. IN, May 2015.

22. Y. Maeda, K. Suzuki, Y. Ishibashi, N. Fukushima, "Influence of Network Delay on Work Efficiency in Remote Robot Control with Force Feedback," (in Japanese), Technical report of IEICE. CQ, Jan. 2015.

23. S. Nakano, Y. Maeda, Y. Ishibashi, N. Fukushima, P. Huang, K. E. Psannis, "Influence of Network Delay on Fairness Between Players in Networked Game with Olfactory and Haptic Senses," Technical report of IEICE. CQ, Jan. 2015.

24. Y. Maeda, Y. Okada, Y. Ishibashi, N. Fukushima, "Subjective Assessment of Sense of Presence in Virtual Environment with Stereoscopic Vision, Haptic Sense, and Auditory Sense," (in Japanese), ITE Winter Annual Convention, Dec. 2014.

25. Y. Maeda, Y. Ishibashi, N. Fukushima, "Influence of Network Delay on Sense of Presence in Networked Virtual Enviroment with Haptic and Auditory Senses," (in Japanese), Technical report of IEICE. CQ, July 2014.

26. Y. Maeda, Y. Ishibashi, N. Fukushima, "Effects of Stereoscopic Vision, Olfaction, and Haptic Sense on Sense of Presence in Virtual Environment," (in Japanese), ITE Winter Annual Convention, Dec. 2013.

27. Y. Maeda, Y. Ishibashi, N. Fukushima, "QoE Assessment of Sense of Presence in Virtual Environment with Haptic and Auditory Senses," (in Japanese), Tokai-Section Joint Conference on Electrical and Related Engineering, Sep. 2013.

28. Y. Maeda, P. Huang, Y. Ishibashi, N. Fukushima, S. Sugawara, "Assessment of Sense of Presence in Virtual Environments Using Olfactory, Haptic, and Auditory Senses by Pair Comparison Method," (in Japanese), IEICE General Conference, Mar. 2013.

29. M. Sithu, Y. Maeda, Y. Ishibashi, "Influence of local lag on interactivity in networked haptic drum performance," IEICE General Conference, Mar. 2013.

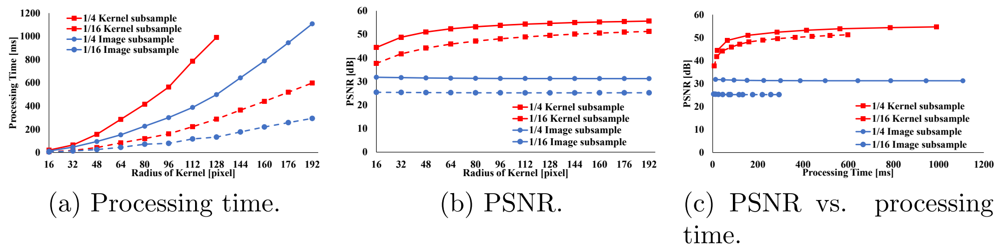
30. Y. Maeda, P. Huang, Y. Ishibashi, N. Fukushima, S. Sugawara, "Effects of Olfactory, Haptic and Auditory Senses on Sense of Presence

in Virtual Environments," (in Japanese), Technical report of IEICE. Multimedia and virtual environment, Jan. 2013.

Appendix A

Pixel Subsampling vs. Kernel Subsampling

In this section, we compare image subsampling to kernel subsampling in bilateral filtering. Figure A.1 indicates the processing time, as well as the accuracy of image and kernel subsampling. It is indicated in the figure that processing time for image subsampling is reduced relative to that for kernel subsampling. However, the PSNR for image subsampling remains lower than that for kernel subsampling. In Figure A.1c, kernel subsampling is greater than image subsampling. This indicates that kernel subsampling has a greater accuracy for the same processing time.



(a) Processing time. (b) PSNR. (c) PSNR vs. processing time.

Figure A.1: Processing time for and accuracy of image subsampling and kernel subsampling with respect to kernel radius of FIR filtering. (**a**) Processing time; (**b**) PSNR; (**c**) PSNR vs. processing time. Image size is 512×512.

Appendix B

Implementation of Bilateral Filter in OpenCV

The weight function of the bilateral filter in OpenCV implementation is defined as follows:

$$
\begin{aligned}
f(\boldsymbol{p},\boldsymbol{q}) &:= EXP_s[\boldsymbol{p}-\boldsymbol{q}] \\
&\quad\; EXP_r[|\boldsymbol{I}(\boldsymbol{p})_r-\boldsymbol{I}(\boldsymbol{q})_r|+|\boldsymbol{I}(\boldsymbol{p})_g-\boldsymbol{I}(\boldsymbol{q})_g|+|\boldsymbol{I}(\boldsymbol{p})_b-\boldsymbol{I}(\boldsymbol{q})_b|] \\
EXP_s[\boldsymbol{x}] &:= \exp(\frac{\|x\|_2^2}{-2\sigma_s^2}), \\
EXP_r[x] &:= \exp(\frac{x^2}{-2\sigma_r^2}).
\end{aligned}
$$

The distance function of the range kernel is not the L2 norm; thus, the range kernel is approximated. Moreover, the shape of the spatial kernel is circular in this implementation. When the kernel shape is circular, the number of referred pixels is smaller than when the kernel shape is rectangular; therefore, this implementation deviates majorly from the naïve bilateral filter. Moreover, we can incorporate this type of approximation in our approach. After using the OpenCV' s approximation, our result is accelerated even more, but the resulting images are overly approximated. Therefore, we do not use the approximation.