A Study on Machine Learning for Image Processing using GPUs

(GPU を用いた画像処理に対する機械学習の研究)

Naoki Matsumura

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Engineering in Information Engineering

Under Supervision of
Professor Koji Nakano

Department of Information Engineering
Graduate School of Engineering
Hiroshima University

September, 2021

Abstract

Machine learning is an application of artificial intelligence that makes systems learn the ability and improve from experience automatically without being explicitly programmed. In recent years, machine learning has been developing rapidly with the developing of *Convolutional Neural Networks (CNNs)*. To reduce the computation time of them, *Graphics Processing Units (GPU)* is mainly used for accelerator. In this dissertation, we research in machine learning for image processing using the GPUs.

First, we present two tile art image generation algorithms using greedy algorithm and machine learning as a greedy approach and a machine learning approach, respectively. The greedy approach takes a large amount of computation time since it pastes tiles one by one. Hence, we introduce a parallel greedy approach and implement it on the GPU. The parallel greedy approach on NVIDIA TITAN V GPU can run up to 318 times faster than sequential one. On the other hand, the machine learning approach generate tile art images by image transformation networks. It can generate a tile art image of size 4096×3072 within 1.04 seconds while the parallel greedy approach takes 571 seconds. In addition, we propose an improvement technique of the machine learning approach to generate a high quality tile art image using iterative inference. As a result, in the tile art image with iterative inference technique, the characteristics of tiles can be enhanced.

Second, we present a novel structured sparse Fully-Connected Layer (FCL) in the CNNs for image classification problem. The proposed approach eliminates the connections between the input and the output nodes except for those in the same position of the feature maps. Since the proposed architecture is defined initially, it is suitable to parallel computation. Therefore, we introduce an efficient implementation for the proposed sparse FCLs on the GPU. As a result for the large scale image recognition dataset, the proposed approach achieves a 21.3 times compression with 0.68% top-1 accuracy and 0.31% top-5 accuracy decrease for VGG-16 network. Also, in the experiment on NVIDIA RTX 2080 Ti GPU, the GPU implementation for the proposed FCLs achieves speed-up factor 14.97 and 16.67 for forward and backward propagation compared to that for the non-compressed FCLs, respectively. Furthermore, to confirm that our proposed approach is applicable to practical image classification problems, we have trained the proposed models using transfer learning on various datasets. The experimental results show the proposed approach can achieve high test accuracy with high compression ratio on each dataset.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Machine learning is an application of Artificial Intelligence (AI) that makes systems learn the ability and improve from experience automatically without being explicitly programmed. In recent years, machine learning has been developing rapidly with the developing of Deep Neural Network (DNNs). In particular, *Convolutional Neural Networks (CNNs)* have become one of common architectures of DNNs. The CNNs are composed of convolutional layers and fully-connected layers. In general, the convolutional layers are used as a feature extractor while the FCLs are used as a feature identifier. The CNNs are applicable to various applications such as image classification, scene classification and visual instance retrieval [1]. The ability of the CNNs is improved significantly in recent years. However, this improvement causes the enlargement of the network size. Hence, several studies related to the reduction of the network size have been devoted. To support programming these technique, various machine learning frameworks such as TensorFlow [2], PyTorch [3] and MXNet [4] have been developed. In these frameworks, *Graphics Processing Units (GPU)* is mainly used for accelerating the computation.

The GPU is a specialized circuit designed to accelerate computation for building and manipulating images. Latest GPUs are designed for general purpose computing that include the computation of machine learning, and can perform computation in applications traditionally handled by the CPU. NVIDIA corporation provides a parallel computing architecture called *Compute Unified Device Architecture (CUDA)* [5]. Also, NVIDIA corporation provides various libraries such as cuBLAS [6] and cuSPARSE [7]. cuBLAS supports basic linear algebra subprograms (BLAS) on the CUDA runtime. On the other hand, cuSPARSE is specialized for handling sparse matrices on the GPU.

1.2 Contributions

In this dissertation, we present tile art image generation methods and a novel structured sparse fully-connected layer in the CNNs.

1.2.1 Proposals for tile art image generation algorithms

In this work, we focus on *tile art image generation* [8] which is *non-photorealistic rendering* technique [9]. The first contribution of this work is to propose a tile art image generation method using the greedy algorithm as *greedy approach*. The greedy approach is based on *the human visual system* which is used for digital halftoning that simulates continuous-tone by varying size or density of tiny black dots. The goodness of a generated tile art image is evaluated by the error between its projected image onto human eyes [10] and the original image. The projected image is simulated on the computer using a two-dimensional Gaussian filter. From the goodness, the approach generates tile art images by pasting tiles one by one to the white canvas image. However, this approach takes a large amount of computation time since it pastes tiles sequentially. Therefore, we propose a parallel greedy approach which can pastes tiles in parallel. To accelerate the computation of the parallel greedy approach, we implement it on the GPU. The experiment results show the GPU implementation of the parallel greedy approach on NVIDIA TITAN V GPU can run up to 318 times faster than the sequential CPU implementation of the greedy approach and 16.19 times faster than the multi-core CPU implementation of the parallel greedy approach with 160 threads.

Nevertheless, the computation time of the greedy approach is still long. Therefore, we propose a tile art image generation method using machine learning as *machine learning approach*. In machine learning approach, tile art images are generated by inferencing the input image on the trained network. The training dataset is composed of tile art images generated by the greedy approach. The network architecture we used is *pix2pix* [11] which is based on the conditional GANs [12]. By training the networks using the training dataset, the trained network can generate the tile art image that have feature of the original image as well as the structure of tiles. Besides, with regard to computation time, the machine learning approach only takes 1.04 seconds to generate tile art image of size 4096×3072, while the greedy approach on the GPUs takes 571 seconds. However, in the tile art images generated by the machine learning approach, some tiles have lack of edge and noises that are not included in the greedy approach. Therefore, we propose a quality improvement technique of the machine learning approach to generate a high quality tile art image using iterative inferences. As a result, in the tile art image with iterative inference technique, the characteristics of tiles can be enhanced.

1.2.2 A proposal for a novel sparse fully-connected layer

In recent, the ability of the CNNs increases significantly. However, it causes enlargement of the network size. Thus far, several works related to reduction of the network size have been tackled [13, 14, 15, 16, 17, 18, 19, 20]. In many cases, these approaches result in producing unstructured networks that is not suitable to parallel computation. To avoid this problem, we propose a novel structured sparse fully-connected layer in the CNNs. This proposal focuses on *Fully-Connected Layers (FCLs)* which occupy a large amount of network parameters. In the popular CNNs such as AlexNet [21] and VGG-16 [22], each input element of the general FCLs is fully-connected to all output elements with any consideration. By contrast, each input element of the proposed FCLs is fully-connected to the output elements in the same position of the feature maps. Hence, the connections between the input and output nodes are decreased, and it result in reduction of the network size. We have evaluated the proposed approach for AlexNet and VGG-16 on ILSVRC-2012 dataset [23] and various small datasets. As a result for ILSVRC-2012 dataset, the proposed approach achieves a 14.7 times compression with 0.68% top-1 accuracy and 0.19% top-5 accuracy decrease for AlexNet, and a 21.3 times compression with 0.68% top-1 accuracy and 0.31% top-5 accuracy decrease for VGG-16. Moreover, since our proposed architecture is structured, the parallel efficiency is higher than that of the unstructured architecture. Also, to accelerate the computation of the proposed approach, we propose an efficient implementation on the GPUs. In the experiment on NVIDIA RTX 2080 Ti GPU, the GPU implementation for the proposed FCLs achieves speed-up factor 14.97 and 16.67 for forward and backward propagation compared to that for the non-compressed FCLs, respectively. Furthermore, to confirm that our proposed approach is applicable to practical image classification problems, we have trained the proposed models using transfer learning on various datasets. The experimental results show the proposed approach can achieve high test accuracy with high compression ratio on each dataset.

1.3 Dissertation organization

This dissertation is organized as follows. In Chapter 2, we describe CNNs and its training flow. In Chapter 3, we describe GPU and CUDA. In Chapter 4, we present the tile art image generation methods using the greedy approach and the machine learning approach. In Chapter 5, we present the structured sparse fully-connected layer in the CNNs. Finally, we conclude this dissertation in Chapter 6.

Chapter 2

Convolutional neural networks and its training flow

In this chapter, we describe the convolutional neural networks and its training flow. In recent years, machine learning has been developing rapidly with the developing of Deep Neural Network (DNNs). In particular, *Convolutional Neural Networks (CNNs)* have become one of common architectures of DNNs. In 2012, Krizhevsky et al. [21] designed a CNN architecture, called AlexNet, which is one of the triggers of the development for CNNs. After that, several powerful CNN architectures such as VGG-16 [22], GoogLeNet [24] and ResNet [25] have been proposed. These CNNs are often utilized as benchmarks to evaluate the performance of novel machine learning algorithms. The general CNNs are composed of *convolutional layers* and *Fully-Connected Layers (FCLs)*, and applicable to various applications such as image classification, scene classification and visual instance retrieval [1]. These applications are implemented on a wide range of hardware platforms from embedded devices [26, 27] to supercomputers [28, 29].

Since the CNNs have appropriateness for image processing, they are used for subroutines in image transformation networks such as *Generative Adversarial Networks (GANs)* [30]

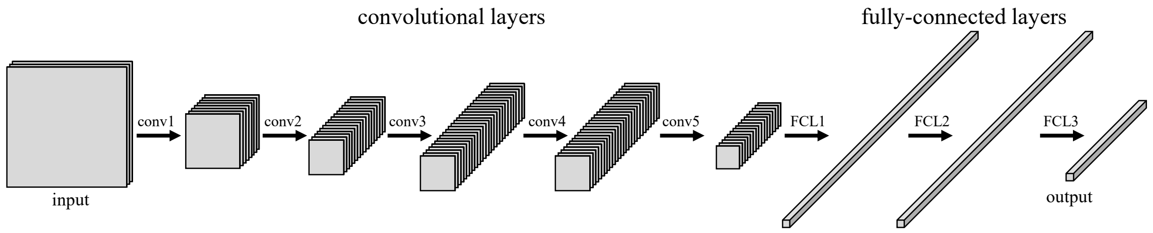


Figure 2.1: The overall architecture of the general CNNs. The general CNNs consist of the convolutional layers and fully-connected layers.

and *pix2pix* [11]. These network architectures are described in Chapter 4.

2.1 Convolutional neural networks

In this section, some techniques about the machine learning used in this dissertation are described. Figure 2.1 shows the overall architecture of the general CNNs. In general, the convolutional layers are used as a feature extractor while the FCLs are used as a feature identifier. The convolutional layers extract the features of the input image as a feature map, and then the FCLs identify what things are included in the images. The convolutional layers process the image by computing two-dimensional convolution shown in Figure 2.2. It has two trainable parameters; *a kernel* and *a bias*. First, an element-wise multiplication of the input image and the kernel is performed. Second, the results of the multiplication are reduced into one element by adding them. Finally, the result of the reduction and the bias are added and outputted as one element of the elements in the output image. This procedure is repeated by striding the kernel on the input image. In general, the stride is set to 1 shonw in Figure 2.2 or 2. The trainable parameters are initialized by random values. Through the training phase described in Section 2.2, the trainable parameters are updated and then the convolutional layers can extract the features of the input image.

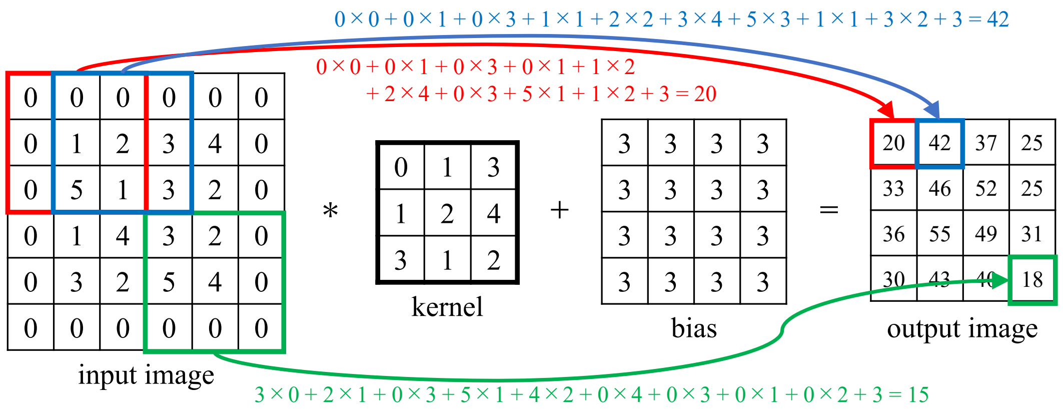


Figure 2.2: The computation of the convolutional layer. This example shows the case when the kernel size is 3 and the stride is 1. In each patch, a two-dimensional convolution and an element-wise addition are computed.

In the CNNs, it is common to insert *a pooling layer* between the convolutional layers. The role of this layer is to reduce the size of feature maps. This results in reducing the number of network parameters and computation in the CNNs. Hence, the overfitting problem can be avoided. There are some kind of pooling layers such as max pooling and average pooling [31]. The max pooling is common in image classification problems, and have achieved better performance than the other pooling layers. In this dissertation, we have used the max pooling. Figure 2.3 shows the computation of the max pooling. It extracts a maximum element from a patch of the image as shown in Figure 2.3. In general, the stride is equal to the patch size, that is, these is no overlap of the patch. Note that the patch size that is larger than the stride (called overlapping pooling) is used in this dissertation. This procedure is repeated until the overall of the image is processed, and the size of feature maps are reduced into one over the size of the patch.

On the other hand, the FCLs estimate what features in the input image indicated. The FCLs process the features by computing a product-sum operation shown in Figure 2.4. It

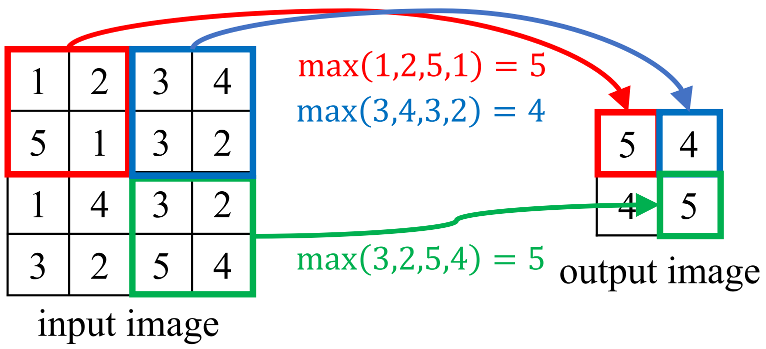


Figure 2.3: The computation of the max pooling layer. This example shows the case when the patch size is 2 and the stride is 2. The maximum element is computed in each patch.

has two trainable parameters; the weight and the bias. First, the input features which are two-dimensional is reshaped to one-dimensional elements to address in the first FCL. After that, a product-sum operation of the reshaped elements and the weights are computed. The product is repeated (the number of the input nodes) $\times$ (the number of the output nodes) times. Therefore, the numbers of the input and the output nodes affect computational complexity. Also, we need to allocate the above number of the weights, thus, the number of the weights becomes the bottleneck in respect to the memory usage. Finally, the results of the product and the biases are added, and the result is outputted. These procedures are repeated for the number of FCLs, and the output of the last FCL will be the output of the CNNs. In general, the number of output nodes of the CNNs is corresponded with the number of classes in the image classification problem. If the number of the classes is 1000, the number of the output nodes of the CNNs is adjusted to 1000. Therefore, each output node shows a probability what class the input is belonging via the softmax function. As same as the training parameters of the convolutional layers, those of the FCLs are initialized by random values. Through the training phase, the trainable parameters are updated and then

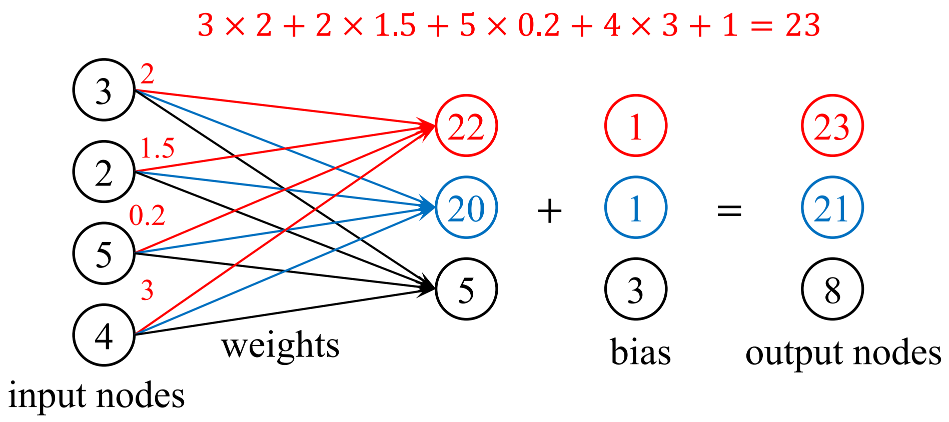

$3 \times 2 + 2 \times 1.5 + 5 \times 0.2 + 4 \times 3 + 1 = 23$
3
2
5
4
2
1.5
0.2
3
22
20
5
+
1
1
3
=
23
21
8
weights
bias
output nodes
input nodes

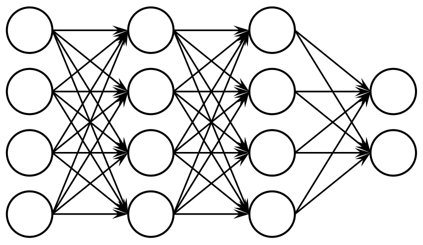
Figure 2.4: The computation of the fully-connected layer. The output node can be computed by product-sum operation. We note that these product-sum can be regard as matrix-vector multiplication, therefore, it is suitable to parallel computation.

the FCSs can identify what the input image indicates.

After each computation of the convolutional layers and the FCLs, *an activation function* is applied to improve their representation ability. As the activation functions, several functions such as sigmoid and rectified linear unit (ReLU) [32] have been used. In this dissertation, we mainly use ReLU activation function that can be computed by the following equation;

$$f(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0). \end{cases}$$

If the network ability is strong, it causes overfitting of the network for training datasets. To avoid this problem, *a dropout technique* [33] has been developed. Figure 2.5 shows what the dropout technique processes when the dropout rate is 50%. In this technique, some nodes in the network are dropped and regarded as which are not exist. For example, if the dropout rate is 50%, the training of the network is processing in the state of a half of nodes is dropped. In other words, the weights which are connected to the dropped nodes


(a) Standard Network

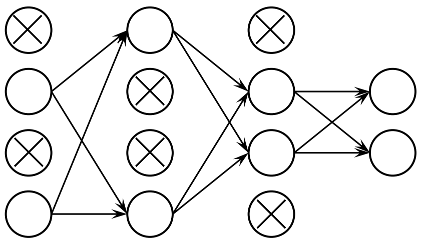

(a) After applying dropout

Figure 2.5: The process of the dropout technique (dropout rate 50%). A half of nodes is dropped and the weights which are connected to the dropped nodes are regarded as zero.

will be regarded as zero and not updated. The dropped nodes are selected randomly per step; therefore, the network can be regarded as the multiple small network. Hence, this technique will improve the versatility of the network.

2.2 Training flow

In this section, the way to train the CNNs is illustrated. The training flow of the CNN is composed of two steps; *a forward propagation* and *a backward propagation*. Figure 2.6 shows the training processes of the CNNs. The forward propagation as illustrated in Figure 2.6(a) is a process that the computation of the CNNs follows from the input to the output. First, we input images and then the convolutional layers and the pooling layers extract feature maps. From the feature maps, the FCLs estimate what the image indicates as the probabilities. After the forward propagation, the loss which is the difference between correct data and the inference result is computed.

The backward propagation as illustrated in Figure 2.6(b) is computed for leading *weight gradients*. The weight gradients are used for updating the weights. To compute the weight gradients, *activation gradients* are needed. The activation gradients are gradient for prop-

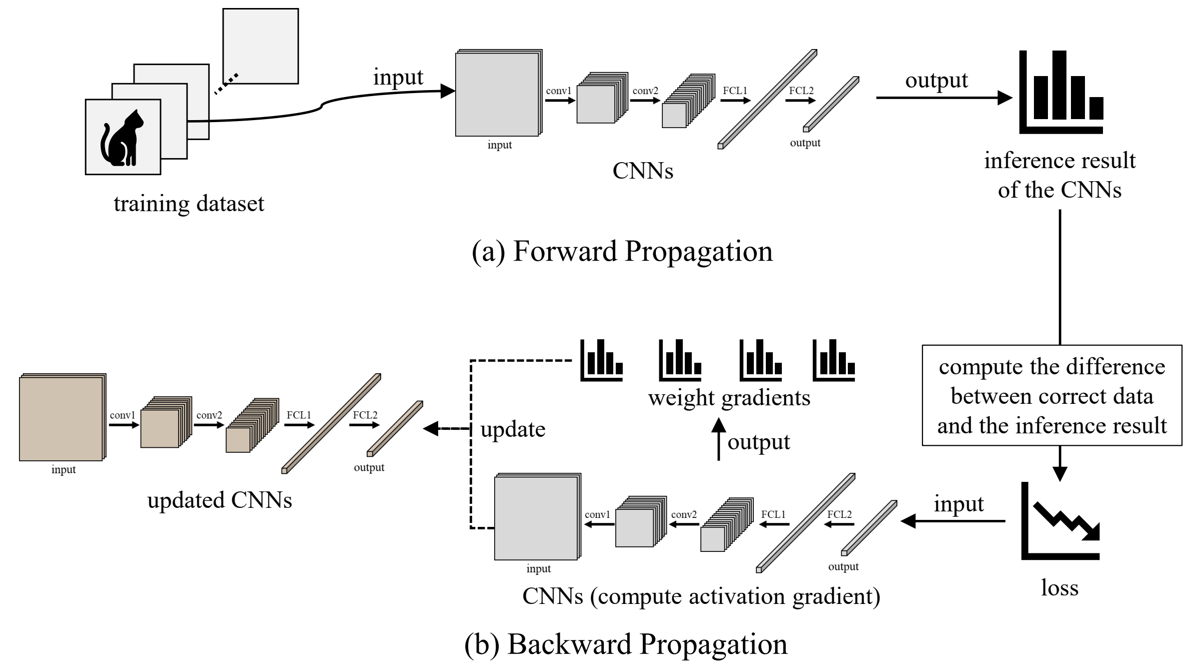


Figure 2.6: Training processes of the CNNs. They mainly consist of two steps; (a) a forward propagation and (b) a backward propagation. The forward propagation computes the outputs which show the probability of each class. The backward propagation computes the amount of updating values which lead the network improvement.

agating the loss up to the first convolutional layer. Therefore, the backward propagation is composed of computations for the activation gradients and the weight gradients. The computation for activation gradients proceeds from the output to the input of the network. Here, we explain the computations of them by focusing to the i-th layer. The activation gradient of i-th layer can be computed by convolution and multiplication of the weights in i-th layer and the activation gradients in $(i+1)$-th layer for the convotlutional layers and the FCLs, respectively. The weight gradient of i-th layer can be computed by convolution and multiplication of the feature maps in $(i-1)$-th layer and the activation gradients in i-th layer for the convolutional layers and the FCLs, respectively. After the backward propagation, the trainable parameters can be updated using the weight gradients. Since the computational complexities of the forward and the backward propagations are high, the training is processed by accelerators such as the GPUs. In the next chapter, we explain the GPU in detail.

As the optimization for the trainable parameters, various techniques have been developed. In this dissertation, we mainly use *mini-batch stochastic gradient descent* [34]. This optimizer adjusts the trainable parameters by subtracting the output of multiplication for the weight gradients and *a learning rate*. The learning rate adjusts the amount of update in the training and is scheduled over the training. In general, the learning rate is starting small value such as 0.01 and decreased when the loss (or accuracy) is not to improve. To stabilise the training in the early phase, the learning rate is set to smaller value than 0.01 [35]. The mini-batch denotes that the number of images used in a update of the trainable parameters. A set of the forward and backward propagations is called *a step*. Therefore, the network consumes mini-batch images per a step. Consuming all images in the training dataset is called *a epoch*. The training dataset is repeatedly loaded many times until defined number of epochs. Repeating these training process until multiple epochs, the training of the network is finished.

Chapter 3

GPU and CUDA

In this chapter, we describe a NVIDIA GPU architecture and a CUDA. *A GPU* (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images. Latest GPUs are designed for general purpose computing that include the computation of machine learning. They can also perform computation in applications traditionally handled by the CPU.

NVIDIA corporation provides a parallel computing architecture called a *Compute Unified Device Architecture (CUDA)* [5]. The CUDA allows developers access to the virtual instruction set and memory in NVIDIA GPU. In many cases, the GPU has thousands of processor cores and very high memory bandwidth. Therefore, the performance of the GPU is more efficient than that of multi-core processors [36]. Figure 3.1 shows an architecture of CUDA-enabled GPU. It is composed of a global memory and multiple Streaming Multiprocessors (SMs) each of which has execution cores and a shared memory. The global memory is composed of an off-chip DRAM or an on-chip HBM2. It has large capacity, but its access latency is very long. On the other hand, the shared memory is an extremely fast on-chip memory with lower capacity. Therefore, the shared memory is often used as a

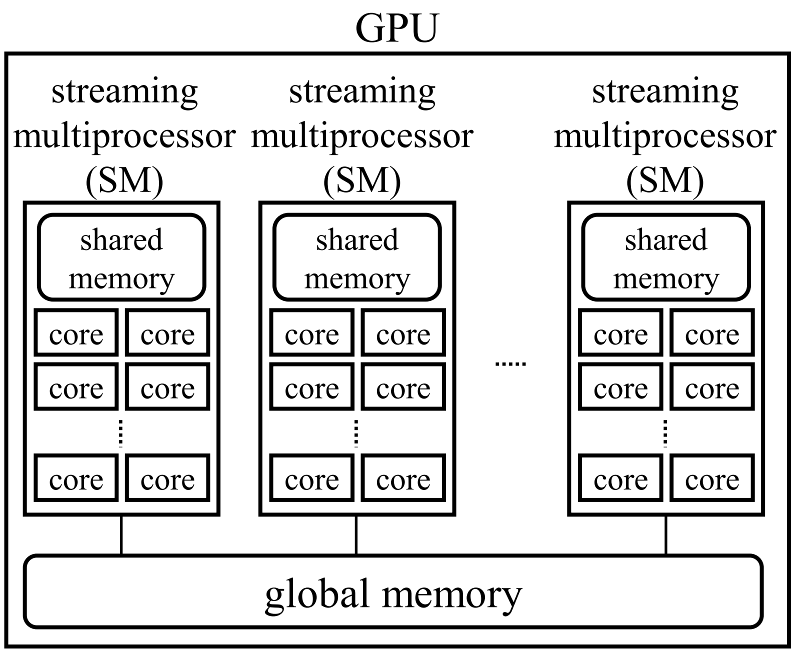

GPU
streaming multiprocessor (SM)
streaming multiprocessor (SM)
streaming multiprocessor (SM)
shared memory
shared memory
shared memory
core
.....
global memory

Figure 3.1: NVIDIA GPU architecture. It is composed of multiple streaming multiprocessors and a global memory. Each streaming multiprocessor consists of a shared memory and multiple cores.

scratchpad memory to cache data accessed frequently.

CUDA parallel programming model consists of three hierarchical thread groups: *grid*, *CUDA block*, and *thread*. Figure 3.2 shows these hierarchical of CUDA. A grid has multiple CUDA blocks, each of which has equal number of threads. The CUDA blocks are allocated to streaming multiprocessors such that all threads in a CUDA block are concurrently executed on the allocated streaming multiprocessor. With regard to the memory access of parallel threads, all threads can access to the global memory, whereas threads in a CUDA block can access to the shared memory of the streaming multiprocessor within the same CUDA block. Since CUDA blocks are arranged to multiple streaming multiprocessors, threads in different CUDA blocks cannot share data in the shared memories. To maximize the memory throughput, *coalescing access* is always desired. The coalescing ac-

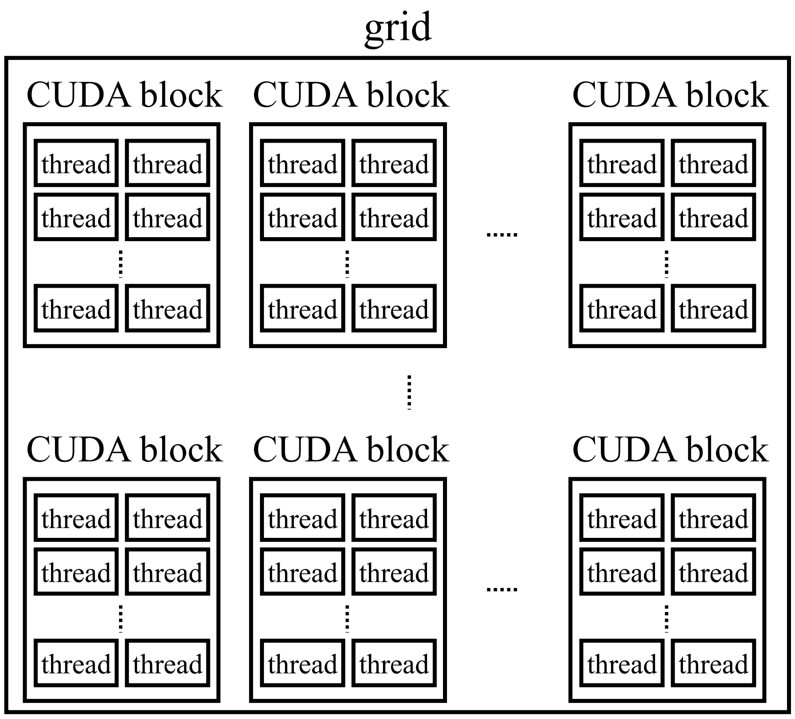


Figure 3.2: CUDA architecture. NVIDIA adopts three hierarchical group. A grid consists of multiple CUDA blocks, and each CUDA block has equal number of threads to other CUDA block.

cess illustrated in Figure 3.3(a) is an efficient memory access pattern to the global memory by multiple threads [37]. By contrast, a stride access illustrated in Figure 3.3(b) is not desired since its access throughput is quite slow. If the bus size of the global memory is four elements, the coalesced access needs two accesses to the global memory while the stride access needs four accesses. CUDA C is an extension to the C language by allowing the programmer to define C functions, called *kernels*. By invoking a kernel, all CUDA blocks in the grid are allocated in streaming multiprocessors, and threads in each CUDA block are executed by processor cores in a single streaming processor in parallel.

Also, NVIDIA corporation provides various libraries such as *cuBLAS* [6] and *cuSPARSE* [7]. cuBLAS supports basic linear algebra subprograms (BLAS) on the CUDA runtime. It is

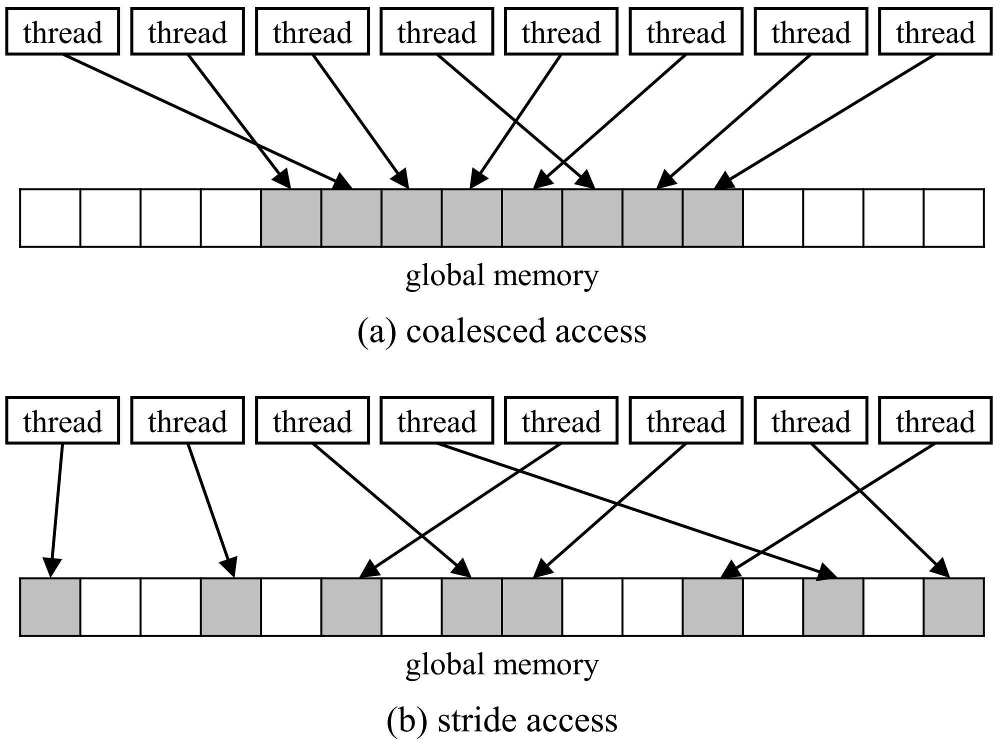


Figure 3.3: The two types of global memory access. The coalesced access (a) is faster than the stride access (b). This is due to the number of access to the global memory from threads. If each thread access to distant elements, multiple accesses to the global memory are needed.

highly tuned for NVIDIA GPUs and is used as an accelerator for deep learning frameworks such as TensorFlow [2], PyTorch [3] and MXNet [4]. In this dissertation, we have used two cuBLAS functions: level 2 functions and level 3 functions. The level 2 functions perform matrix-vector multiplication and vector-vector addition. On the other hand, the level 3 functions perform matrix-matrix multiplication and matrix-matrix addition.

In addition, we have used a cuSPARSE which is a library for handling sparse matrices on the GPU. This library is also highly tuned for NVIDIA GPUs as well as cuBLAS. cuSPARSE can address multiple sparse matrix data formats such as *Compressed Sparse Row Format* (CSR), *Compressed Sparse Column Format* (CSC) and *Block Compressed*

Sparse Row Format (BSR). In this dissertation, a level 3 function which computes a matrix-matrix multiplication between a sparse matrix and a dense matrix is used. Also, as the sparse matrix data format, we have adopted the CSR data format which is often used in machine learning.

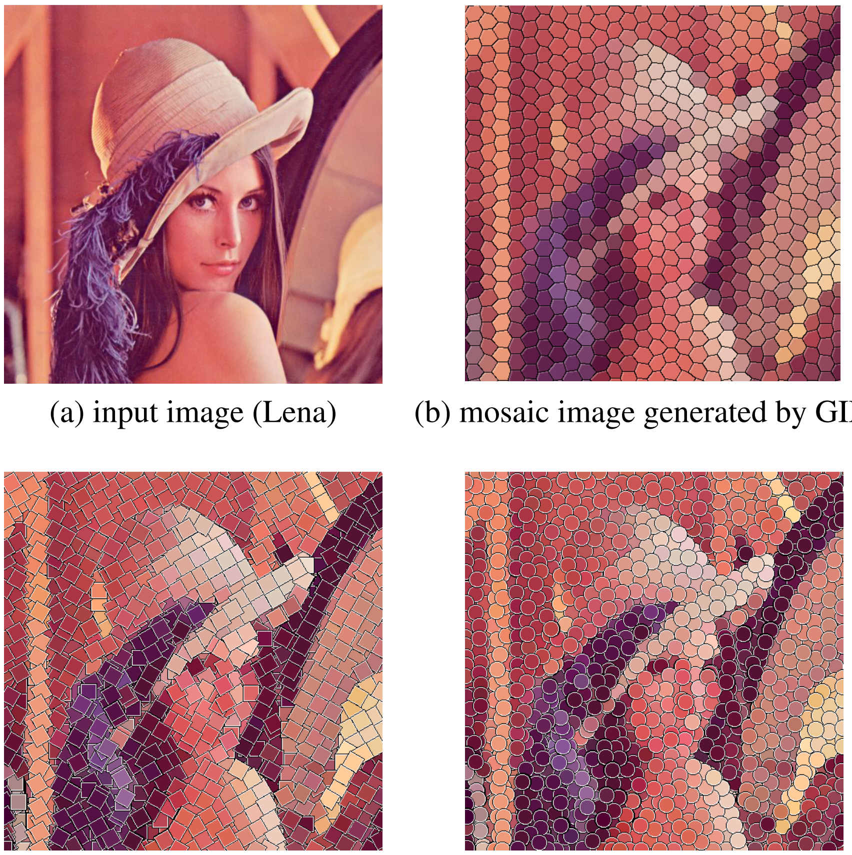
Chapter 4

Tile art image generation on the GPUs and its approximation with machine learning

4.1 Introduction

Tile art made by assembling small pieces of tiles is one of the artistic techniques and has a long history [8]. While, in the field of computer graphics, tile art images are generated on the computer. Generating tile art images by computers is known as one of the *non-photorealistic rendering* techniques which produce an image resembling artistic representation. So far, several researches and studies of non-photorealistic rendering for tile art have been devoted [9] and several photo retouch tools support such artistic image transformation. Figure 4.1(b) shows mosaic image generated by GIMP [38] from Figure 4.1(a) as an input image. This mosaic image is covered with non-overlapped hexagons. Each hexagon has a black border and it is filled by a uniform color.

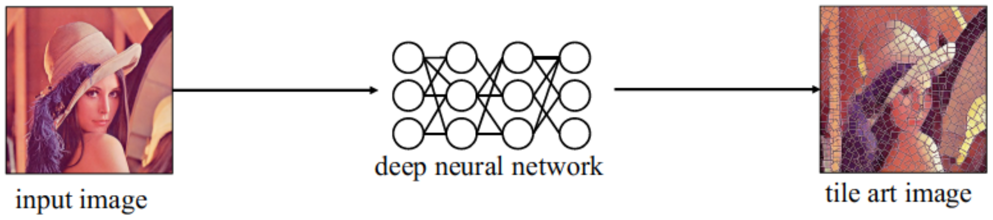
In this work, we propose two approaches for tile art image generation; *a greedy approach* and *a machine learning approach.* As the first contribution, we propose the greedy approach for the tile art image generation. Figures 4.1(c) and (d) show generated tile art


(a) input image (Lena)　(b) mosaic image generated by GIMP

(c) square tile art image　(d) circle tile art image

Figure 4.1: An example of our tile art image generation. We have used two tile patterns; (c) square tiles and (d) circle tiles.

images with square and circle tiles, respectively. Unlike general tile art images as shown in Figure 4.1(b), in this work, tiles can be overlapped. To generate tile art images, the idea of *the human visual system* is adopted. This idea is used for digital halftoning that is a process that simulates continuous-tone by varying size or density of tiny black dots. Analoui et al. introduce the definition of the goodness of a generated binary image as the error between its projected image onto human eyes and the original image [10]. The projected image is simulated on the computer using a two-dimensional Gaussian filter that approximates the characteristic of the human visual system. As a result, a high quality binary image that



(b) Machine learning approach

Figure 4.2: An outline of the two approaches for tile art image generation. (a) The greedy approach generates the tile art image by pasting tiles one by one. (b) The machine learning approach generates the tile art image by using deep neural network.

represents a given continuous-tone image well is obtained. This idea has also been utilized in the digital halftoning that varies density of black dots [39, 40] and ASCII art generation [41]. In this work, we consider overlapped tile patterns as tiny black dots in digital halftoning and apply this idea to simulate a continuous-tone image.

To obtain high quality tile art images, we generate a tile art image such that the total error is minimized. While, several studies, that tackle tile art image generation, consider features in the original image such as edges and directions of gradation [42, 43, 44]. Unlike these studies, our proposed approach can represent such features without any image feature extraction techniques. In the greedy approach, we adopt the greedy search technique. Given an original input image, the approach generates tile art images by pasting tiles one by one to the canvas image as illustrated in Figure 4.2(a). In Figure 4.1, 2621 square tiles and 2657 circle tiles of size 23×23 are pasted to obtain the tile art images each. We can see that the tile images consist of overlapped tile patterns, and also edges and gradations larger than the square patterns are represented well. However, considering the computation time, it is not realistic for practical applications. Therefore, in this work, we show a parallel method of the greedy approach and its GPUs (Graphics Processing Units) implementation to accelerate the computation. We efficiently use very high memory bandwidth and thousands of parallel threads of the GPUs. The experimental results show that the GPU implementation of the greedy approach can run up to 318 times faster than the sequential CPU implementation and 16.19 times faster than the parallel multi-core CPU implementation with 160 threads.

However, the problem of the generation time still exists because it takes several minutes to generate a large tile art image using the GPU implementation. Therefore, in this work, we propose an approximation for tile art image generation using a machine learning approach. The approximation is to generate tile art images with a deep neural network for image-to-image translation problems. The problems have been rapidly improved using machine learning for super-resolution images [45, 46, 47, 48], image colorization [49, 50, 51], image completion [52, 53, 54, 55], image style translation [56, 57], among others. To solve these problems, *Convolutional Neural Networks* (CNNs) and *Generative Adversarial Networks* (GANs) [30] have been used. Since the connections between neurons resembles the organization of the receptive field, the feature of images, that is difficult for traditional fully connected neural networks, can be caught by introducing CNNs. While, GANs have been used for image generation problems. GANs are composed of two network models, *the generator* and *the discriminator*. The generator is a network model that tries to generate outputs that mimic training data, whereas the discriminator is a model that tries to distinguish training data from outputs obtained by the generator. In network learning, these two models are improved with each other by training these two models adversarially. However, since the size of image is reduced once to extract the feature once, CNN-based models often generate blurred images which have lack of details. To tackle this problem, Isola *et al.* showed a general-purpose solution, called *pix2pix* [11]. The network of pix2pix is composed of the conditional GANs [12] and various contrivances are devoted. Using pix2pix technique, in this work, for a given input image, we generate a tile art image by inferring the network shown in Figure 4.2(b). To construct the proposed network, we adapted the idea of the pix2pix method. In addition, we use tile art images generated by the above greedy approach as the training data set. Namely, the aim of the proposed network is to generate tile art images close to those by the greedy approach. As a result, the proposed network can generate tile art images that have feature of the original image as well as the structure of tiles. With regard to computation time, the greedy approach on the GPUs takes 571 seconds to generate tile art image of size 4096×3072, while the machine learning approach takes 1.04 seconds.

However, in the tile art images generated by the machine learning approach, some tiles have lack of edge and noises that are not included in the greedy approach. Therefore, we propose an improvement technique of the machine learning approach to generate a tile art image using iterative inference.

This chapter is organized as follows. Section 4.2 explains the proposed tile art image generation using greedy approach and shows the GPU implementation with the parallel algorithm. In Section 4.3, we propose a tile art image generation method using machine learning. Section 4.4 shows the resulting tile art images and the computation time. In Section 4.5, we propose an improvement technique for the machine learning approach to enhance characteristic of tiles of output images. Section 4.6 concludes this work.

4.2 The greedy approach for tile art image generation

The main purpose of this section is to propose a tile art image generation method using the human visual system. We first introduce the error between an original input image and the output tile art image based on the human visual system. After that, we show a greedy approach for generating tile art image with the error. In addition, to shorten the computation time, we propose the parallelization of the greedy approach and its GPU implementation. In this work, the target shape of tiles is either square or circle. For brevity of explanation, the tile art image generation using square tiles is mainly considered below.

First, we introduce the error between an original input image and the output tile art image based on the human visual system. In the following, we explain it for a gray scale image for the sake of ease. After that we extend it to a color image. Consider an input original image $A = (a_{i,j})$ of size $N \times N$, where $a_{i,j}$ denotes the intensity level at position (i, j) $(1 \leq i, j \leq N)$ taking a real number in the range $[0, 1]$. We note that for simplicity we consider that an input image is square in what follows. The tile art image generation is to find an image $B = (b_{i,j})$ obtained by pasting a large number of tile patterns such that the original input image A is reproduced. The error of the output image B from an input image A can be computed using the two-dimensional Gaussian filter that simulates the feature of the human visual system. Let $G = g_{p,q}$ denote a two-dimensional Gaussian blurred filter.

The filter is composed of a two-dimensional symmetric matrix and the size of the filter is $(2w+1)\times(2w+1)$, where each non-negative real number $g_{p,q}$ $(-w \le p, q \le w)$ is determined by a two-dimensional Gaussian distribution;

$$g_{p,q} = s \cdot e^{-\frac{p^2+q^2}{2\sigma^2}}, \tag{4.1}$$

where σ is a parameter of the Gaussian distribution and s is a constant real number to satisfy $\sum_{-w \le p,q \le w} g_{p,q} = 1$. Let $R = (r_{i,j})$ denote the projected gray scale image of an image $B = (b_{i,j})$ obtained by applying the two-dimensional Gaussian filter as follows:

$$r_{i,j} = \sum_{-w \le p,q \le w} g_{p,q} b_{i+p,j+q} \quad (1 \le i, j \le N). \tag{4.2}$$

As $\sum_{-w \le p,q \le w} g_{p,q} = 1$ and $g_{p,q}$ is non-negative, each $r_{i,j}$ takes a real number in the range $[0, 1]$. Hence, the projected image R is a gray scale image. An image B is a good approximation of the original input image A if the difference between A and R is small. The error $e_{i,j}$ at each pixel location (i, j) is defined by

$$e_{i,j} = a_{i,j} - r_{i,j}, . \tag{4.3}$$

Using this error, the total error is defined by the sum of them:

$$\text{Error}(A, B) = \sum_{1 \le i,j \le N} |e_{i,j}|. \tag{4.4}$$

When $Error(A, B)$ is small enough, we can consider that the image B reproduces original input image A.

Using the above similarity based on the human visual system, we explain how we generate a tile art image. A tile art image is obtained by pasting fixed-size tiles of the same

(a) square tiles (b) circle tiles

Figure 4.3: Two sets of tile patterns P. The square tiles (a) need to consider the rotation of tiles, while the circle tiles (b) do not.

size $(2t + 1) \times (2t + 1)$. Every tile has white and black borders of width is 1 pixel each and inside pixels of tiles are filled with a uniform color. Tiles are pasted one by one to a white canvas of the same size as the input image, in which tiles are allowed to be rotated and overlapped other tiles. Let P denote a set of square tile patterns and each element $p_{u,v}$ $(1 \leq u \leq N_L, 1 \leq v \leq N_R)$ in P denotes a tile pattern, where N_L and N_R be the numbers of colors and rotation variation of patterns, respectively. Figure 4.3 depicts examples of patterns of square tiles and circle tiles. For example, when $N_R = 4$, square patterns are rotated by 0, 22.5, 45, and 67.5 degrees. Thus, by considering each rotated pattern is distinct, the total number of square tile patterns is $N_L N_R$. We note that since the shape of circles does not change by rotation, $N_R = 1$ for circle tiles and the total number of circle tile patterns is N_L.

We introduce an improvement value I of the total error in Eq. (4.4) by pasting a tile pattern to explain where and which tile pattern is pasted:

$$I(A, B, p, i, j) = \text{Error}(A, B) - \text{Error}(A, B'), \tag{4.5}$$

where B' is a canvas image when a tile pattern p is pasted at (i, j) to B. We paste a tile pattern with the maximum value of the improvement I for all possible patterns to B. In other words, we select a pattern $q_{i,j}$ whose value I is the maximum for each position (i, j) $(1 \leq i, j \leq N)$ such that

$$q_{i,j} = arg \max_{p \in P} I(A, B, p, i, j).$$

From $q_{i,j}$'s, we find the most improved pattern q_{best} whose improvement value is the maximum. After that, the pattern q_{best} is pasted to B. Whenever a tile pattern is pasted, improvement values have to be recomputed and a new tile is pasted with them. This procedure is repeated until no more improvement is possible, that is, the improvement value is negative wherever any tile is pasted.

In general tile art, tiles are put on a blank canvas whose color is white or black and the canvas might be seen from the gap of tiles. While, in this work, such background is entirely covered by pasting tile patterns. In other words, the original color of canvas cannot be seen in generated tile art images. This is because in our experiment, if the background is visible through the gap, such part looks conspicuous noises. Therefore, in the following algorithm, we first cover background pixels with tile patterns. We introduce the contribution ratio of covering background pixels when a pattern p is pasted to canvas B at (i, j), expressed as:

$$C(B, p, i, j) = \frac{\text{the number of covered background pixels}}{\text{the number of pixels of } p}.$$

When a pattern is pasted on background pixels without any overlap, $C(B, p, i, j) = 1$, whereas when a pattern is pasted on non-background pixels, $C(B, p, i, j) = 0$. Also, when the ratio $C(B, p, i, j)$ is larger, more background pixels are covered by a pattern p. To cover background pixels, we choose and paste a pattern with the value of $C(B, p, i, j)$ as large as possible. By extending Eq. (4.5), we define an improvement value I_{cover} to consider the background pixels covering:

$$I_{\text{cover}}(A, B, p, i, j) = (C(B, p, i, j), I(A, B, p, i, j)). \tag{4.6}$$

The improvement value is defined by a pair of 'the contribution ratio of covering background pixels' and 'the improvement of the total error' when a pattern p is pasted to B at (i, j). We assume that the comparison of any two values of $I_{\text{cover}}(A, B, p, i, j)$ are based on the lexicographical order, that is, $I_{\text{cover}}(A, B, p, i, j) > I_{\text{cover}}(A, B, p', i, j)$ if and only if

- $C(B, p, i, j) > C(B, p', i, j)$ or,
- $C(B, p, i, j) = C(B, p', i, j)$ and $I(A, B, p, i, j) > I(A, B, p', i, j)$.

To cover background pixels, we use I_{cover} instead of I. We note that if background pixels in the two-dimensional Gaussian filter application are included, the values of background pixels that will be covered affect the total error. Therefore, until all background pixels are covered, to exclude background pixels in the Gaussian filter application when I_{cover} is computed, instead of Eq. (4.3), we use the following error between an original image and the canvas image;

$$e_{i,j} = a_{i,j} - b_{i,j}.$$

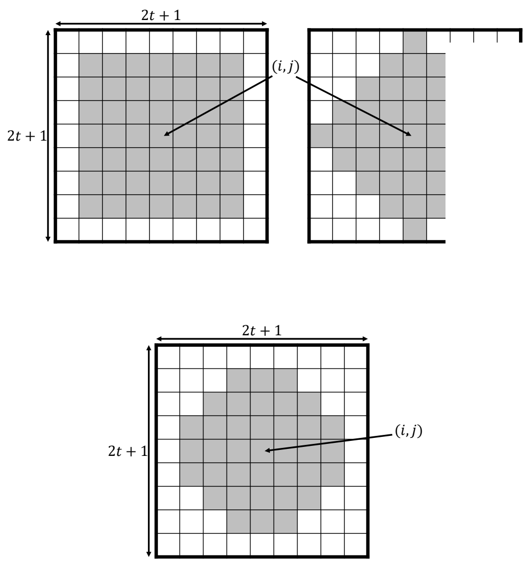
Now, we extend the error computation for gray scale images to color images. We consider RGB color whose value is specified with three real numbers in the range $[0, 1]$ that represent red, green, and blue, respectively. For color images, projected image R and the error in Eq. (4.3) are computed for each color. Namely, for each color, two-dimensional

Gaussian filter is applied, and the following error is computed. Let $e^R_{i,j}$, $e^G_{i,j}$, and $e^B_{i,j}$ denote the errors of red, green, and blue at each pixel location (i, j), respectively. Eq. (4.4) is extended to the sum of each color value as follows;

$$\text{Error}(A, B) = \sum_{1 \leq i,j \leq N} (|e^R_{i,j}| + |e^G_{i,j}| + |e^B_{i,j}|). \tag{4.7}$$

In the following, we show a tile art image generation method, called *the greedy approach* using the above error between an original image and the tile art image based on the human visual system. The idea of the greedy approach is to paste tile patterns one by one such that the total error is minimized.

Consider an original input image A and a white canvas image B and let $W(i, j)$ be a square window of size $(2t + 1) \times (2t + 1)$ whose center is at position (i, j) as illustrated in Figure 4.4. The window is the minimum upright square can include all tile patterns including its rotation in Figure 4.3. Because we use a two-dimensional Gaussian filter of size $(2w + 1) \times (2w + 1)$, pasting a tile pattern affects the errors in a square region of size $(2t + 2w + 1) \times (2t + 2w + 1)$. In following, we call such region *affected region*. Figure 4.5 illustrates a window and the affected region. We note that the best pasting of a tile pattern can be chosen by computing the total errors of the affected region of size $(2t + 2w + 1) \times (2t + 2w + 1)$ because pasting a tile pattern does not affect errors at pixels outside the affected region. The error of a fixed pixel in an affected region can be computed in $O((2w + 1)^2) = O(w^2)$ time with Eqs. (4.2) and (4.3). Hence, all the errors in the affected region can be computed in $O(w^2(2t + 2w + 1)^2) = O(w^2(t^2 + w^2))$ time. After that, their sum can be computed in $O((2t + 2w + 1)^2) = O(t^2 + w^2)$ time. Thus, the total error in the affected region can be computed in $O(w^2(t^2 + w^2))$ time. Since it is necessary to check all



(b) circle tile patterns

Figure 4.4: A window $W(i, j)$ of a tile.

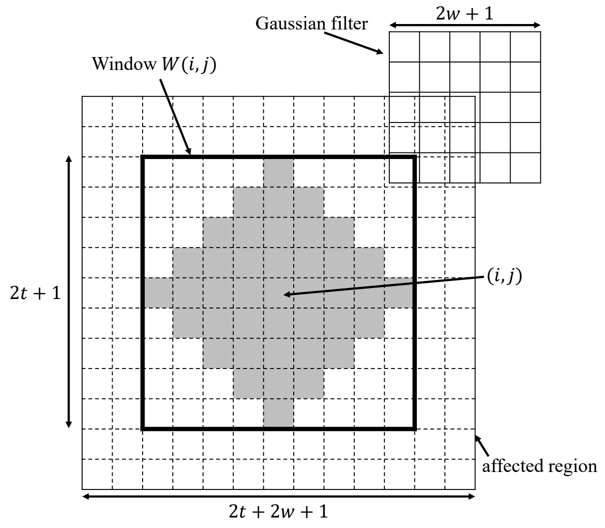


Figure 4.5: An affected region of the window $W(i, j)$.

the possible $N_R N_L$ tile patterns in P, the best pasting can be obtained in $O(N_R N_L w^2(t^2 + w^2))$ time.

4.2.1 Sequential algorithm for the greedy approach

In the following, we present a sequential algorithm for the greedy approach. Algorithm 1 shows the sequential algorithm for the greedy approach for a given input image A. A canvas image B of the same size as A is initialized to a blank white image. After that, we find the best tile pattern $q_{i,j}$ for every position (i, j) by computing the improvement value I_{cover} in Eq. (4.6). After that, we find the most improved pattern q_{best} that has the maximum value of I_{cover} from the tile patterns $q_{i,j}$'s and then we paste q_{best} to the canvas. By repeating this procedure, tile art image can be generated. However, in order to perform the finding q_{best}, it is not necessary to find $q_{i,j}$'s for all the points once all $q_{i,j}$'s are obtained. If the projected image in the affected region does not change, we can omit the search for the current window using the previous best pattern $q_{i,j}$ as the current best tile pattern. Let $\mathcal{A}_{i,j}$ denote a set of positions in the affected region of the image in Figure 4.5 such that

$$\mathcal{A}_{i,j} = \{(i', j') | i - t - w \leq i' \leq i + t + w, j - t - w \leq j' \leq j + t + w\}.$$

Therefore, we compute the total error at (i, j) in Eq. (4.4) by evaluating the following formula for gray scale images:

$$\sum_{(i',j') \in \mathcal{A}_{i,j}} |e_{i',j'}|. \tag{4.8}$$

Similarly, for color images, the total error in Eq. (4.7) can be computed by

$$\sum_{(i',j') \in \mathcal{A}_{i,j}} (|e^R_{i',j'}| + |e^G_{i',j'}| + |e^B_{i',j'}|). \tag{4.9}$$

Thus, in Algorithm 1, the tile pattern search that finds $q_{i,j}$ is performed for all positions. After that, the search is performed only for positions $\mathcal{A}_{i,j}$ in the affected region and the best tile pattern q_{best} is pasted. This procedure is repeated until no more improvement is possible. We note that pasting tile patterns and the total error computation require values from pixels outside of the image boundaries when patterns are pasted around borders. Therefore, in this work, the nearest border pixels in an original image A are extended as far as necessary to perform the search for $q_{i,j}$'s shown in the above around the boundaries [58]. More specifically, an original image A of size $N \times N$ is extended to the $(N+2t+2w) \times (N+2t+2w)$ image by copying the boundary pixel values.

To accelerate the above search finding $q_{i,j}$'s, we introduce *a partial search technique* that limits the search space for color of a pattern. More specifically, the color space of the search space is reduced. In this technique, instead of the exhaustive search that finds

Algorithm 1 Sequential algorithm for the greedy approach

```
Input: Original image $A$
Output: Tile art image $B$
1: $B_0$ is initialized to a blank white image
2: for $i = 1$ to $N$ do
3:     for $j = 1$ to $N$ do
4:         Find $q_{i,j}$.
5:     end for
6: end for
7: $k \leftarrow 1$
8: loop
9:     $B_k \leftarrow B_{k-1}$
10:    for $i = 1$ to $N$ do
11:        for $j = 1$ to $N$ do
12:            Update $q_{i,j}$ if the projected image in the affected region of $W(i, j)$ for $B_k$ and $B_{k-1}$ are not identical.
13:        end for
14:    end for
15:    Find the most improved pattern $q_{best}$ from all $q_{i,j}$'s
16:    if the total error decreases when $q_{best}$ is pasted to $B_k$ then
17:        Paste $q_{best}$ to $B_k$
18:    else
19:        return $B_k$
20:    end if
21:    $k \leftarrow k + 1$
22: end loop
```

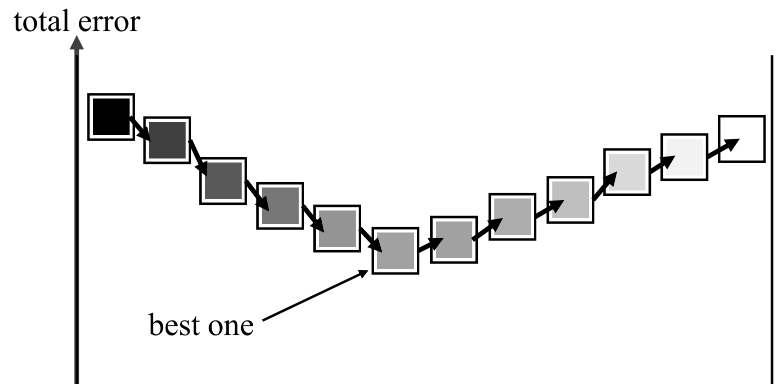
an optimal color of a tile pattern for all possible N_L colors as illustrated in Figure 4.6(a), the following partial search is performed. Beforehand, we make a list of N_L colors of tile patterns by sorting them in order of average brightness. Let α be a pattern that has the closest color to the average brightness in window $W(i, j)$ of the original input image A. We start the search in the list from α. By checking the neighboring tile patterns of the list in an obvious way we can find the bottom of the concave sequence as illustrated in Figure 4.6(b). Using this partial search technique, the search space can be reduced. One may argue that the quality of the resulting tile art images that obtained by this partial search is inferior to that produced by the exhaustive search. In our preliminary experiment, however, there is almost no difference of the total error and the quality of generated images between them.

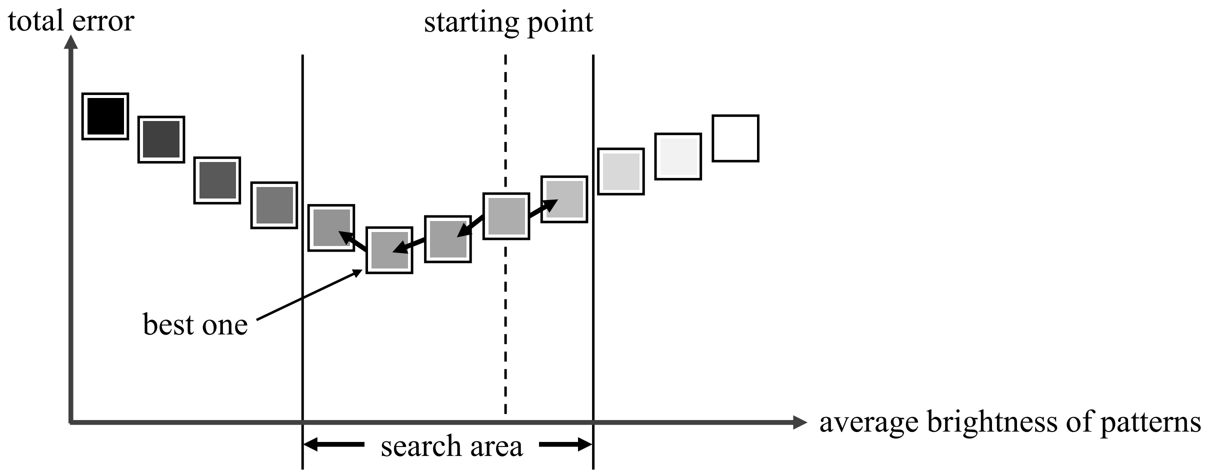
4.2.2 Parallel algorithm for the greedy approach

In the above sequential algorithm of the greedy approach, tile patterns are pasted one by one. However, two tile patterns of which affected regions overlap cannot be pasted since such pastes affect each other. Therefore, it is difficult to parallelize the algorithm as it is. Here, we show the parallel algorithm to paste multiple tile patterns at the same time. In this parallel algorithm, we paste multiple tile patterns so that they do not share the affected region in parallel. To do this, we split an original input image A and the canvas image B of size $N \times N$ into subimages of size $h \times h$. We classify the subimages into four groups such that

- Group 1: odd columns and odd rows;
- Group 2: even columns and odd rows;
- Group 3: odd columns and even rows; and
- Group 4: even columns and even rows.

Figure 4.7 illustrates the four groups of the subimages. We note that, if $h \geq 2t+2w+1$, then the two-dimensional Gaussian filter of two subimages in a group never affect each other, where the subimage is $h \times h$ and the affected region is $(2t+2w+1) \times (2t+2w+1)$. In other words, affected regions of a certain group do not overlap with each other. In the parallel algorithm, we perform the sequential algorithm in every subimage for Group 1, Group 2,





(b) Partial search

Figure 4.6: The partial search technique to find color of a pattern. (a) The exhaustive search needs to search for all color patterns. (b) The partial search technique can reduce the search area since it explores the color patterns from the nearest neighbor color patterns.

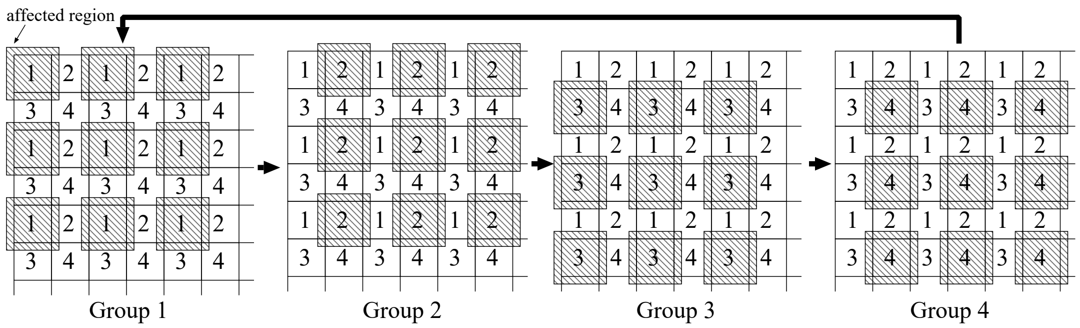


Figure 4.7: The groups of subimages and parallel execution without race condition. By separating the image to four groups, each affected region within the same group will not affect each other.

Group 3, and Group 4, in turn. Since there are $\frac{N^2}{4h^2}$ subimages in each group and at least one tile pattern is pasted in each subimage, at most $\frac{N^2}{4h^2}$ tile patterns can be pasted for each group execution. This parallel execution of the four group is repeated until no more improvement is possible. In other words, the execution is finished when the total error cannot be reduced by pasting a tile anywhere. Readers may think the generated tile art images using this parallel algorithm is different from those by the sequential algorithm since multiple patterns are pasted at the same time. However, in our preliminary experiment, there is a little practical difference between them for the total error, the number of pasted tiles, and the quality. We will show a GPU implementation using the above parallel execution technique in the next.

4.2.3 GPU acceleration

We propose an efficient GPU implementation of the parallel algorithm of the greedy approach to accelerate the computation. We first explain how we implement the parallel algorithm of the greedy approach on the GPUs. We assume that an original input image of size $N \times N$ is stored in the global memory in advance, the implementation writes the resulting tile art image to the global memory. For these images, we repeat to invoke a kernel for Group 1, Group 2, Group 3, and Group 4 illustrated in Figure 4.7, in turn. Each kernel consists of CUDA blocks, each of which is assigned to each subimage in one of the four groups. In each subimage, multiple threads in an allocated CUDA block are used to find $q_{i,j}$'s for each position (i, j) in parallel. In other words, each CUDA block with multiple threads is responsible for finding $q_{i,j}$'s in a subimage. First of all, to reduce global memory access, each CUDA block caches the elements of A and B that are necessary to perform the computation, to the shared memory. To cache data to the shared memory, multiple threads in the CUDA block cooperate to store the data in the global memory with coalescing access in parallel. Multiple threads in a CUDA block are used to compute the improvement values I_{cover} in Eq. (4.6) for each tile pattern and then find the most improved pattern at (i, j) $q_{i,j}$ whose value I is the maximum. After that, the most improved tile pattern q_{best} is selected from $q_{i,j}$'s and the pattern is pasted to the canvas. This procedure consisting of kernel calls for the four groups is repeated until no more improvement is possible. We note that in the first round consisting of kernels for the four groups, $q_{i,j}$'s for all positions need to be computed. However, once $q_{i,j}$'s for all positions are computed, the following kernels update $q_{i,j}$'s only for the affected region where a tile pattern is pasted in the previous kernels. Using this idea, the computation cost can be reduced greatly.

In addition to the above parallel computation, we use the following ideas efficient to perform finding $q_{i,j}$'s. As shown in Section 4.2.1, whenever a tile is pasted, the projected image has to be updated. To update it, we compute the application of the two-dimensional Gaussian filter in Eq. (4.2) for each pixel. In the GPU implementation, we replace the two-dimensional convolution to addition of blurred tile patterns. More specifically, the sum of products in the convolution is replaced by the sum of additions. To do this, we compute the projected tile patterns blurred by the two-dimensional Gaussian filter and store them to the global memory in advance. Instead of the convolution, the following operations are performed for each tile pattern using them. First, a tile pattern whose value is 0, that is, black tile pattern, is pasted at which the tile is put in order to clear the pasted tiles so far. After that, the blurred tile pattern is added to B instead of the application of the Gaussian filter. By replacing the sum of product computation in the convolution to the addition, we reduce the computation cost. In addition, to obtain the total error in Eq. (4.8) and Eq. (4.9), the sum of the error values is computed. In the GPU implementation, we use the parallel sum reduction technique using *the warp shuffle instructions*. A warp is a set of threads that all share the same instructions and it consists of 32 threads in a CUDA block. The warp shuffle instructions allow threads in a warp to perform the data communication between them without the shared memory [37]. In this technique, the parallel sum computation is efficiently performed using the warp shuffle instructions. We use this parallel sum reduction technique in applying the two-dimensional Gaussian filter with addition in the above.

4.3 Tile art image generation using machine learning

The main purpose of this section is to propose a tile art image generation using machine learning. In the greedy approach, a lot of tile patterns are repeatedly pasted on the canvas. On the other hand, in this machine learning approach, we directly generate a tile art image for a given input image using a deep neural network. In other words, we obtain a tile art image only by inference computation of the trained network without iteration of pasting. The architecture of the network in the proposed approach is based on *the pix2pix technique* [11]. In the following, we explain the network model, network architecture, and optimization method of the proposed machine learning approach.

4.3.1 Network models

In the proposed machine learning approach, we use *Generative Adversarial Networks* (GANs) as a network model. GANs are a deep neural network model, that is based a game theory [30], consisting two networks, a *generator* G and a *discriminator* D. The generator learns to output $G(z)$ in a target domain, where z is a latent variable that is usually random noise. Here, we consider that an image in the training data set is a *real* image, and an output image generated by the generator is a *fake* image. The discriminator learns to classify images as real or fake. More specifically, the discriminator tries to output a probability assigned near to 1 for real images and near to 0 for fake images. The generator and the discriminator are trained each other adversarially and simultaneously. During the training, the generator tries to generate images that are not able to be identified as generated images, whereas the discriminator aims to distinguish generated images by the generator from training images. After training, the generator produces images indiscernible from training images to trick the discriminator. To achieve the above training, we introduce a loss function $\mathcal{L}_{GAN}$:

$$\mathcal{L}_{GAN}(G, D) = E_x[\log D(x)] + E_z[\log(1 - D(G(z)))], \tag{4.10}$$

where $E_x[\cdot]$ and $E_z[\cdot]$ are expected values on the probability distributions of training images x and latent variable z, respectively.

In the above, the generator produces images only from latent variable z. However, in image-to-image translation, the produced image must be related to an input source image. Therefore, to associate a generated image with a source image, we employ *conditional GANs* that are an extension of GANs [12]. In conditional GANs, a source image y is given as an additional input for generator network G and discriminator network D. Figure 4.8 depicts the outline of conditional GANs. The generator has two inputs; latent variable z and source image y and outputs image $G(z, y)$. On the other hand, to the discriminator, either a real pair or a fake pair is given. A real pair consists of training image x and source image y, where a fake pair is composed of output image $G(z, y)$ generated by the generator and source image y. The discriminator learns to distinguish a fake pair and a real pair. To train the networks of the conditional GANs, the loss function of conditional GANs $\mathcal{L}_{cGAN}(G, D)$ that is an extension of Eq. (4.10) is used:

$$\mathcal{L}_{cGAN}(G, D) = E_{x,y}[\log D(x, y)] + E_{y,z}[\log(1 - D(G(z, y), y))]. \tag{4.11}$$

Additionally, in pix2pix technique, the following loss function is added to the above loss function in Eq. (4.11) to generate images similar to the corresponding training images [11];

$$\mathcal{L}_{L_1}(G) = E_{x,y,z}[||x - G(z, y)||_1],$$

where $\mathcal{L}_{L_1}(G)$ is the average value of absolute values of the difference at each pixel between a training image x and the generated image $G(z, y)$.

In addition, we introduce *the feature matching* [59] that is a loss function to stabilize the learning of the generator. The loss function of feature matching $\mathcal{L}_{FM}$ is defined by

$$\mathcal{L}_{FM}(G, D) = E_{x,y,z} \sum_{t=1}^{T} \frac{1}{N_t}[||D^{(t)}(x, y) - D^{(t)}(G(z, y), y)||_1],$$

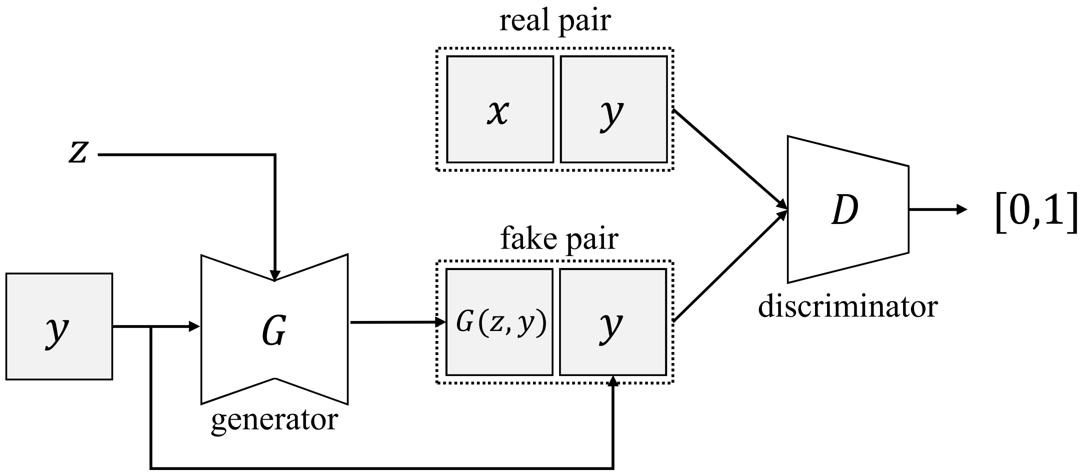


Figure 4.8: An outline of conditional GANs. The conditional GANs consist of the generator and the discriminator. The generator generates fake image $G(z, y)$ from the source image y and noise z. The discriminator discriminates whether the inputted pair is a real pair by outputting the value in range 0 to 1. (1 means the inputted pair is discriminated as a real pair.)

where T is the total number of intermediate layers in the discriminator, $D^{(t)}$ denotes the t-th layer in the discriminator, and N_t is the number of nodes in $D^{(t)}$. Using this loss function, the generator is trained such that it can produce diverse images from the target distribution in the training data set. We note that the loss function of the original feature mapping proposed in [59], mean squared error is utilized. While, in this work, we adopted average error from our preliminary experiments.

Finally, the loss function in this work $\mathcal{L}(G, D)$ by combining the above loss functions is used as

$$\mathcal{L}(G, D) = \mathcal{L}_{cGAN}(G, D) + \lambda_1 \mathcal{L}_{L_1}(G) + \lambda_2 \mathcal{L}_{FM}(G, D), \tag{4.12}$$

where λ_1 and λ_2 denote hyper-parameters that control the weights of the terms for $\mathcal{L}_{L_1}(G)$ and $\mathcal{L}_{FM}(G, D)$, respectively. During training, the generator and the discriminator attempts to minimize and maximize $\mathcal{L}(G, D)$, respectively. In other words, the purpose of training is to find the generator G^* obtained by solving the optimization problem;

$$G^* = \arg\min_G \max_D \mathcal{L}(G, D).$$

4.3.2 Network architectures

Using the proposed network model shown in the above, we show the details of architecture of the generator network and the discriminator network as follows. In the proposed architecture of the networks, we mainly use the idea of the pix2pix technique [11].

We use the idea of *U-Net* [60] in the generator network. Figure 4.9 depicts the proposed architecture of the generator network. The architecture is composed of two parts, *an encode network* and *a decode network*. In pix2pix technique [11], there are 8 layers for the encode network and 8 layers for the decode network. The number of layers is a little large since it is able to deal with various applications for image-to-image translation. In our preliminary experiments, we have evaluated 3 to 8 layers of the generator network and the discriminator network each. As a result, the quality of generated tile art images using 5 to 8 layers is almost the same and the quality becomes worse for the networks of less than 5 layers. Therefore, we adopt a 5-layer network from image quality and computation time. Also, in every downsampling and upsampling computation in the networks, the number of convolutions is reduced compared with the original U-Net. We explain the detail of the encode network and the decode network, as follows. The structure of the encode network is a convolutional neural network. The network is composed of the repeated application of a 4×4 convolution with stride 2 while spatial downsampling. After each convolution layer, as an activate function is applied, we use a leaky rectified linear unit (LeakyReLU) [32]:

$$f(x) = \begin{cases} x & (x \geq 0) \\ \gamma x & (x < 0), \end{cases} \tag{4.13}$$

where γ denotes a hyper-parameter in range $(0, 1)$ that makes the slope small when the input is negative. While, the decode network uses a deconvolutional, or up-convolutional, network. Every step in the decode network consists of an upsampling of the feature map by a 4×4 deconvolution with stride 2, and each followed by a rectified linear unit (ReLU) [32]:

$$f(x) = \begin{cases} x & (x \geq 0) \\ 0 & (x < 0). \end{cases}$$

On the other hand, in the original U-Net [60], the encode network consists of the repeated application of two convolutions and a max pooling operation with stride 2 to downsample the image. Also, the decode network is composed of the repeated application of two convolutions and an up-convolution to upsample the image. Therefore, in the encoder and decoder networks, the numbers of convolutions and downsamples of pooling are reduced compared with the original U-Net. Each layer of the decoder network has a concatenation with the feature map from the encode network via skip connection [60]. This concatenation via skip connection is employed to avoid losing pixel-level localization by downsampling in the encode network. The output of three channels corresponds to the RGB color planes of the output color image. We note that in U-Net, instead of inputting a latent variable z, *the dropout* is applied to the generator network. The dropout is a training technique such that a certain set of nodes which is selected randomly is ignored during the training phase.

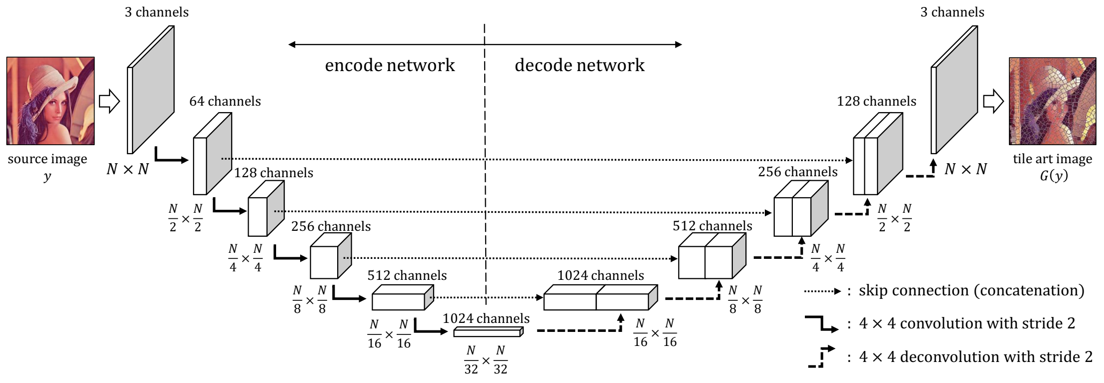

3 channels
encode network
decode network
64 channels
128 channels
256 channels
512 channels
1024 channels
source image y
$N \times N$
$\frac{N}{2} \times \frac{N}{2}$
$\frac{N}{4} \times \frac{N}{4}$
$\frac{N}{8} \times \frac{N}{8}$
$\frac{N}{16} \times \frac{N}{16}$
$\frac{N}{32} \times \frac{N}{32}$
tile art image $G(y)$
: skip connection (concatenation)
: 4 × 4 convolution with stride 2
: 4 × 4 deconvolution with stride 2

Figure 4.9: The architecture of the generator network. It consists of a encode network which has 5 convolutional layers and a decode network which has 5 deconvolutional layers. The dotted lines mean the skip connections to promote the backward propagation of the loss.

The technique is used for reducing overfitting on the network. Since this dropout can be considered as inputting a latent variable z, in the proposed architecture, we omit a latent variable, that is, the input of the generator network is only a source image y.

Figure 4.10 depicts the architecture of the proposed discriminator network. We utilized the architecture of patchGAN [61] as the discriminator network. In the discriminator described in the above, one discriminating result is output for an input. On the other hand, in the patchGAN, the input image is divided into small subimages and the result of the discrimination for each subimage is output. The network is given by either a real pair or a fake pair that is a concatenation of two images. A real pair consists of a source image y and a training tile art image x generated by the greedy approach from y. In contrast, a fake pair is a concatenation of a source image y and a tile art image obtained by the generator $G(z)$. In the discriminator network, there are three 4×4 convolution layers with stride 2 for spatial downsampling and two 4×4 convolution layers with no strides. Also, each convolution layer is followed by LeakyReLU [32]. Finally, for a given pair of images of size $N \times N$

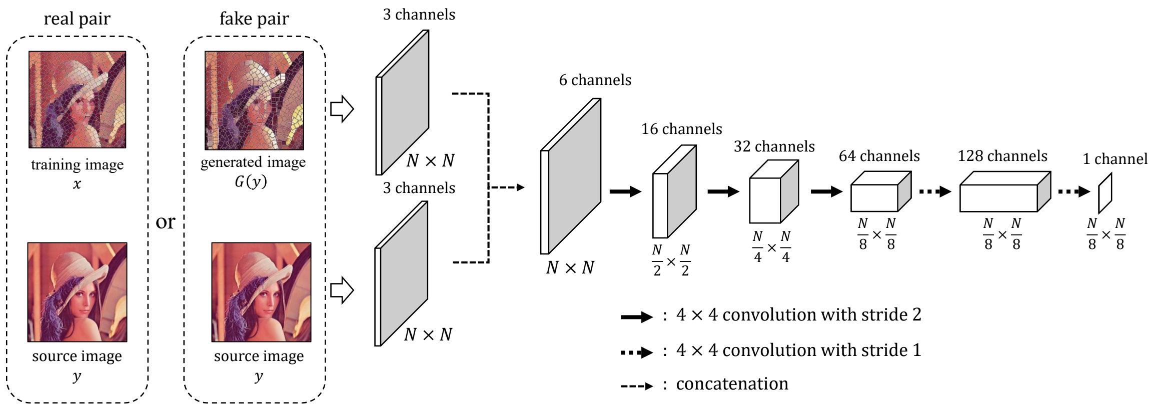

real pair
fake pair
training image
x
generated image
$G(y)$
or
source image
y
source image
y
3 channels
$N \times N$
3 channels
$N \times N$
6 channels
$N \times N$
16 channels
$\frac{N}{2} \times \frac{N}{2}$
32 channels
$\frac{N}{4} \times \frac{N}{4}$
64 channels
$\frac{N}{8} \times \frac{N}{8}$
128 channels
$\frac{N}{8} \times \frac{N}{8}$
1 channel
$\frac{N}{8} \times \frac{N}{8}$
: 4 × 4 convolution with stride 2
: 4 × 4 convolution with stride 1
: concatenation

Figure 4.10: The architecture of the discriminator network. It consists of 5 convolutional layers. The input images are concatenated and inputted to the discriminator.

each, the output of the discriminator network is reduced to $\frac{N}{8} \times \frac{N}{8}$. Each element of the output corresponds to a resulting probability of the discrimination for a divided subimage.

We note that in this work, a trained network can only generate a tile art image in which tiles are similar size and shape to those in the training tile art images. If different shape of tiles is required, the network has to be trained again. Also, if the size of tile is changed, the network structure may be changed since the number of layers in the generator and discriminator networks depends on the size of tiles. The network structure shown in this section is optimized for 23×23 square or circle tiles. We note that the trained network can be used for any size of input images because the networks consist of convolutional layers.

4.3.3 Network optimization

We are now in position to train the proposed networks. We adopted a technique of mini-batch stochastic gradient descent [34] that is one of the stochastic optimization methods in training deep neural networks. The size of input images for the generator and discriminator networks is set to 256×256, i.e., $N = 256$. If an input image is larger than 256×256, the convolutional operation is repeatedly applied for horizontal and vertical directions because every layer of the generator network is either convolutional or deconvolutional layer. In other words, for a given image, the generator network reduces the size while encoding and then enlarges the size while decoding. For smaller images less than 256×256, an original image is extended to the 256×256 by copying the boundary pixel values and the resulting image is cropped to the original size. Therefore, the proposed networks can be used for any size of images. With regard to the activate functions, in the LeakyReLU (Eq. (4.13)), the slope of the leak is set to $\gamma = 0.2$ both in the generator and discriminator networks. This is because in the original pix2pix technique, the slope is 0.2, and also from our preliminary experiment, when the slope is 0.2, the quality of the resulting tile art image is higher compared with the other value of the slope. We utilized the dropout technique with a rate of 50% for the first and second layers in the decode network of the generator. Also, in the discriminator, the technique is applied with the same rate to the first, second and third layers. In addition, to accelerate training, we use the Adam optimizer [62]. We set values of parameters in the Adam optimizer such that $\alpha = 0.0001, \beta_1 = 0.5, \beta_2 = 0.99$, and $\epsilon = 10^{-12}$. For the details of these parameters of the Adam optimizer, the interested reader may refer to the reference [62]. All weight values in the generator and discriminator networks were initialized from a zero-centered Gauss distribution with standard deviation 0.02. The learning rates of the generator network and the discriminator network were set to 0.0002 and 0.00001, respectively. The control weights λ_1 and λ_2 in Eq. (4.12) were set to 1.

Algorithm 2 shows a training algorithm for our proposed networks, where the size of mini-batch m is set to 4. The loop in the algorithm is repeated until the output image almost does not change. Each loop corresponds to one *epoch* that is when a whole training dataset is passed forward and backward through the networks only once. In our experiment shown in the next section, we repeated the loop 200 times, that is, 200 epochs. The loop in the algorithm consists of two parts. The former part is to train the generator network for the fixed discriminator network. The latter part is to train the discriminator network for the fixed generator network. We note that to balance the ability of the two networks from the result of our preliminary experiments, the discriminator is trained once every two loops.

Algorithm 2 Training algorithm for the proposed networks

```
1: Normalize all pixel values of images in the training data set to $[-1, 1]$.
2: $k \leftarrow 1$
3: loop
4:     Randomly select a mini-batch of $m$ examples of source images $y_1, \dots, y_m$ with the
       corresponding tile art images $x_1, \dots, x_m$.
5:     Update the generator by ascending its stochastic gradient for the fixed discriminator:
       $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} (\log(1 - D(G(y_i), y_i)) + \lambda_1(\|x_i - G(y_i)\|_1) + \lambda_2(\sum_{t=1}^{T} \frac{1}{N_t}(\|D^{(t)}(x_i, y_i) - D^{(t)}(G(y_i), y_i)\|_1))).$$
6:     if $k$ is even then
7:         Update the discriminator by ascending its stochastic gradient for the fixed generator:
           $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} (\log D(x_i, y_i) + \log(1 - D(G(y_i), y_i))).$$
8:     end if
9:     $k \leftarrow k + 1$
10: end loop
```

4.4 Experimental results

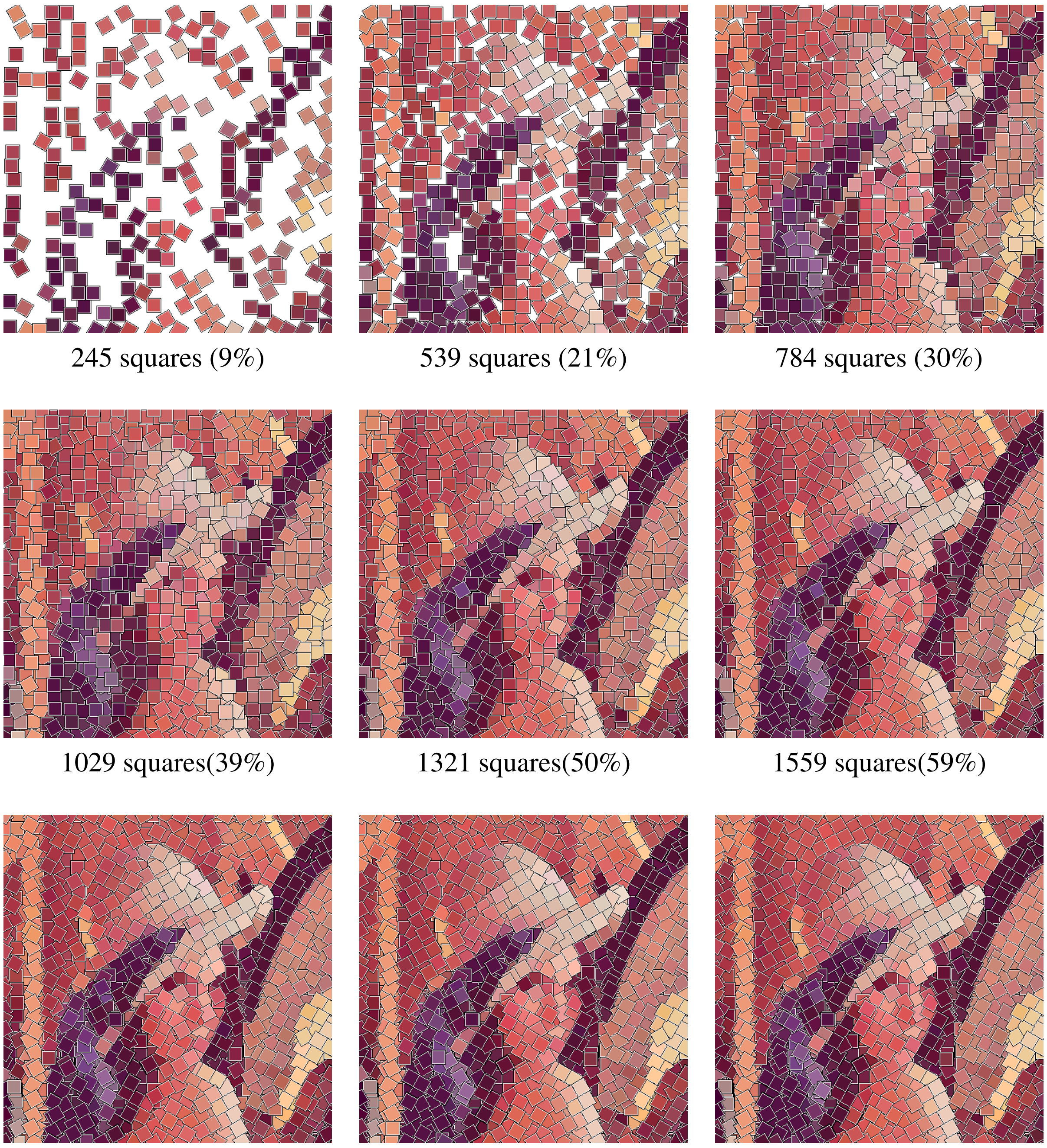
In this section, we show the resulting tile art images using the greedy approach and the machine learning approach. We also show training process in the machine learning approach. After that, the generated tile art images are compared between two approaches. At the end of this section, we evaluate the computation time of them.

4.4.1 Resulting tile art images using the greedy approach

First, we show the resulting tile art images using the greedy approach. We have used Lena [63] of size 512×512 in Figure 4.1(a). We have generated square tile art images and circle tile art images, where each tile pattern is 23×23 and 4096 colors. Square tile patterns are also rotated with angles of 0, 30, and 60 degrees. In total, the number of patterns in P is $4096 \times 3 = 12288$ for square tile patterns and 4096 for circle tile patterns. The two-dimensional Gaussian filter in Eq. (4.1) has been set with parameters $\sigma = 1.3$ and $w = 3$. Also, the size of subimage used in the parallel algorithm of the greedy approach is 39×39, that is, $h = 39$.

Figure 4.11 shows the snapshots of process of generating a tile art image by the greedy approach. To generate the resulting tile art image in Figure 4.12(a), 2621 square patterns have been pasted in total. From the resulting images, in first, tiles have been put to hide the canvas due to Eq. (4.6). Therefore, until the canvas is almost filled by tiles, the detailed part cannot be represented well. After that, we can see that tiles are put to represent the original image.

Figure 4.12 shows the images generated by the sequential and parallel algorithms of



245 squares (9%) 539 squares (21%) 784 squares (30%)

1029 squares(39%) 1321 squares(50%) 1559 squares(59%)

1835 squares(70%) 2089 squares(80%) 2361 squares(90%)

Figure 4.11: Snapshots of pasting tiles in the parallel greedy approach with square tile patterns of size 23×23.

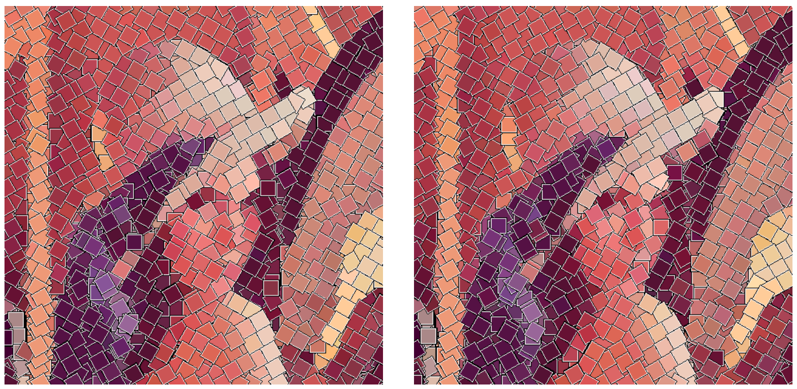

(a) sequential algorithm (b) parallel algorithm

Figure 4.12: Square tile art images generated by the greedy approach. There is no difference between these images at a glance.

the greedy approach with square patterns. Compared with the images, all of them are a bit different by looking at the detail, whereas the quality of these images seems to be almost the same at a glance. In both images, the intricate parts such as her hair cannot be represented well since the tile patterns are too large to represent such parts in detail. On the other hand, principal edges and large gradations of colors are well-reproduced by overlapped tile patterns even though the border of tiles is prominent.

4.4.2 Network training and generated tile art images for the machine learning approach

In this work, we have trained two networks for machine learning approach, one is for square tile image generator, and the other is for circle tile image generator. To train these two networks, we use Caltech-256 [64] image database consisting of 30607 images about 256 object categories as source images. From the database, we have randomly chosen 20000 images and scaled them to 256×256. We have obtained tile art images generated by the greedy approach from the source images with the same condition as the above. We

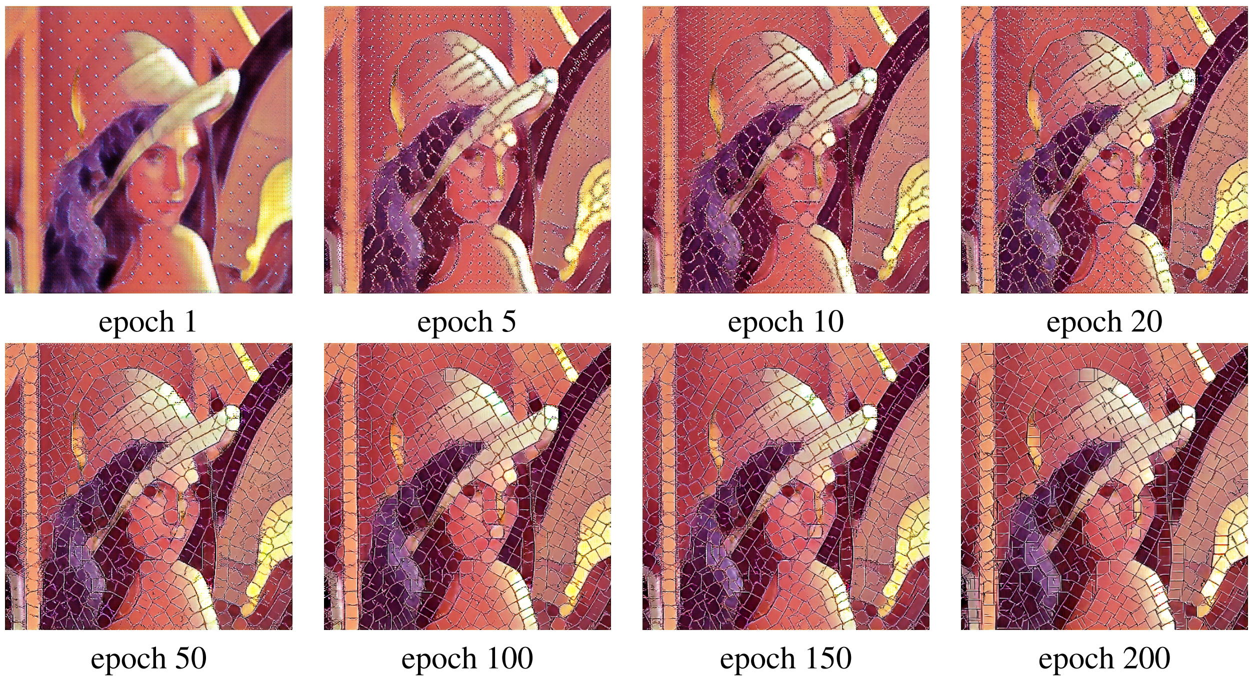


Figure 4.13: The output images of the generator during the training. At epoch 1, the original image still appears as it is. At epoch 50, tiles become visible, however, the edges of tiles are thin. At epoch 200, tiles become clearly.

use the resulting tile art images as training dataset of images.

We have trained the networks by the optimization method shown in Section 4.3.3. We have implemented the machine learning approach using TensorFlow version 1.12.0 [2]. The networks have been trained until the output tile art images are not changed. In our experiment, the change of the output images almost disappeared at 200 epochs. The training takes approximately 25 hours using NVIDIA Tesla V100 GPU. Figure 4.13 shows the snapshots of the resulting tile art images for Lena during the training. We note that the image Lena is not included in the training data. From the images, in earlier epochs, the original image still appears as it is. As the training progressed, the characteristic of tiles become visible. After 150 epochs, the characteristic of tiles is more visible.

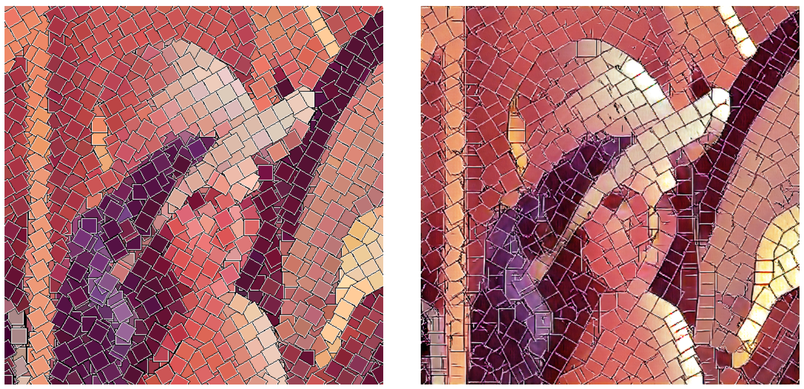

(a) greedy approach (b) machine learning approach

Figure 4.14: The generated tile art images for square tile patterns. Comparing with the two approaches, the quality of the greedy approach is higher.

4.4.3 Comparison between the greedy approach and the machine learning approach

Figures 4.14 and 4.15 show the generated tile art images for Lena of size 512 × 512 obtained by the greedy approach both for square and circle patterns. The resulting images can represent the original image well. Especially, we can see that the overlapped patterns can form the outline consisting of long edges for the both patterns.

On the other hand, compared to the tile art images generated by the greedy approach, those by the machine learning approach have the larger tendency to arrange the tiles regularly. Although the machine learning approach does not put tile patterns one by one, the structure of each tile and the overlap between tiles can be represented in the generated images. However, there are chips in the edge of tiles and non-uniform color in some tiles. The machine learning approach can generate higher quality circle tile art image compared with the square tile image. This is because the shape of circle patterns is simpler than that of square patterns since circle patters have no rotation. Also, Figures 4.16 and 4.17 show the generated tile art images for a Full HD photograph of size 1920 × 1080 (Figure 4.16(a)).

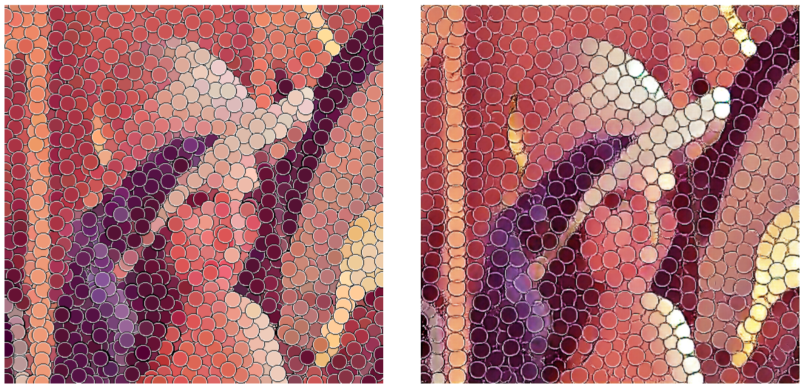
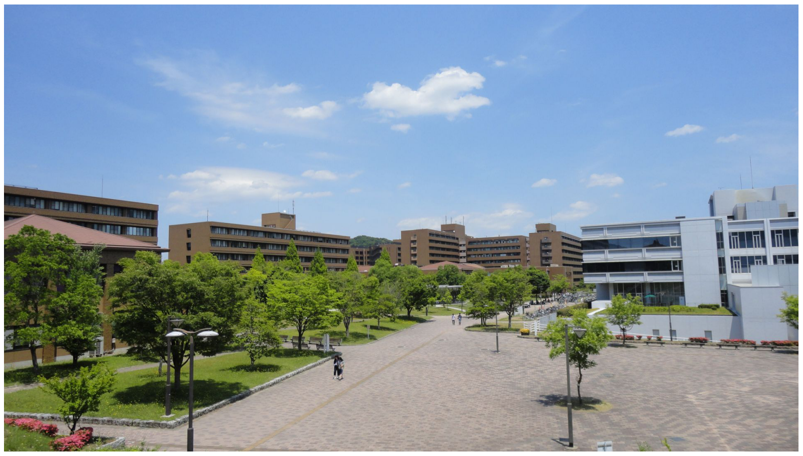

(a) greedy approach (b) machine learning approach

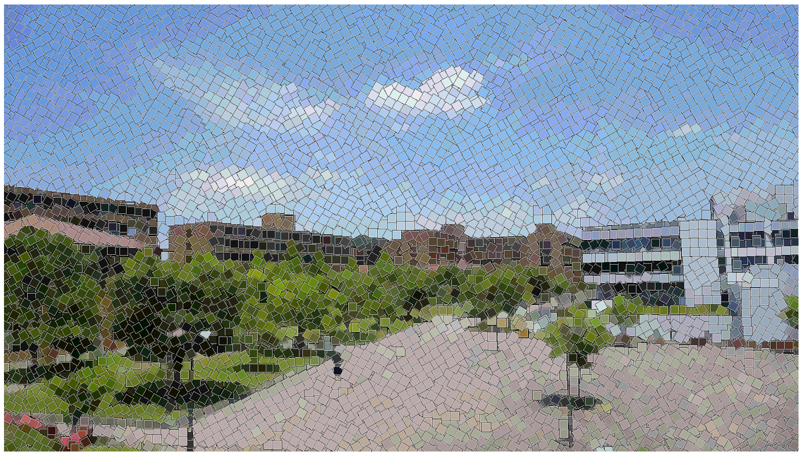
Figure 4.15: The generated tile art images for circle patterns. Unlike the case of square tiles patterns, there is no difference between the generated images of the greedy approach and the machine learning approach at a glance. In other words, the machine learning approach can generate high quality tile art image since the generator does not need to consider the rotation of tiles.

For these images, the resulting images by the greedy approach have the same tendency. While, the generated images with the machine learning approach are also represented well though the regular arrangement of tiles in the sky is more conspicuous both for square and circle patterns.

Figure 4.18 shows the mosaic image generated by GIMP [38] for a Full HD image in Figure 4.16(a). Since the mosaic image consists of non-overlapped hexagons that can be distorted, it is difficult to compare them with the proposed method directly. At a glance the quality of this image is higher than that of the tile art images generated by the proposed approaches. However, the detailed parts such as buildings and trees cannot be represented well. On the other hand, in the tile art images generated by the proposed approaches, such detailed parts are clearer.

Table 4.1 shows the number of pasted patterns in the greedy and the total error in Eq. (4.9) in the greedy and machine learning approaches for various size of images.


(a) input image


(b) greedy approach

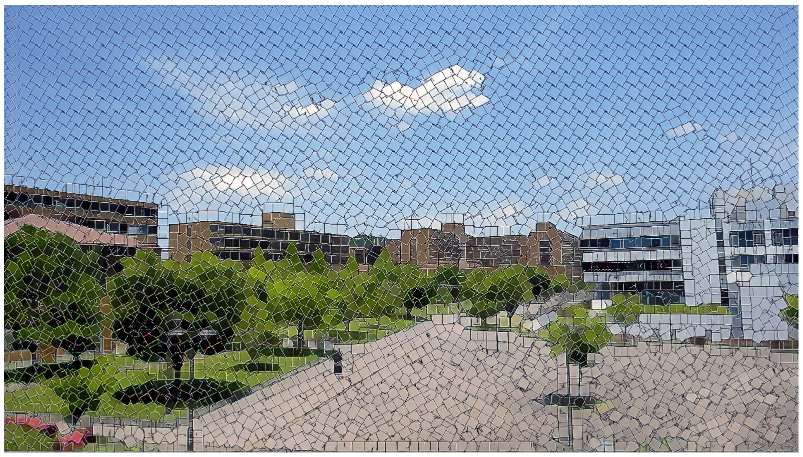
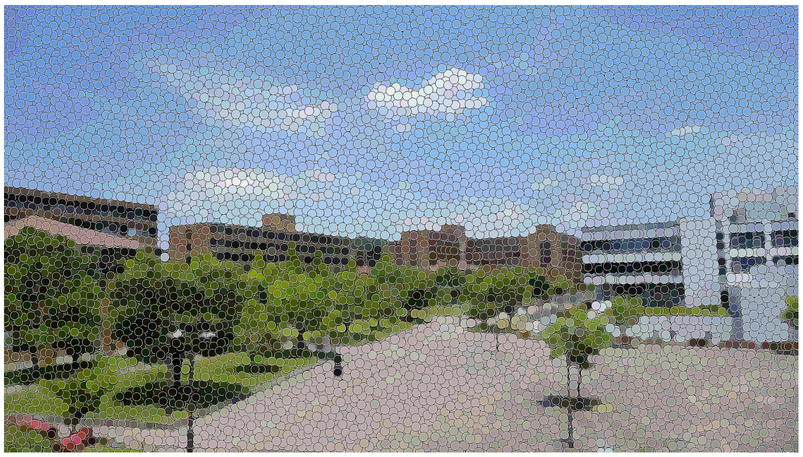

(c) machine learning approach

Figure 4.16: The generated square tile art image for a Full HD image (1920 × 1080).


(a) greedy approach

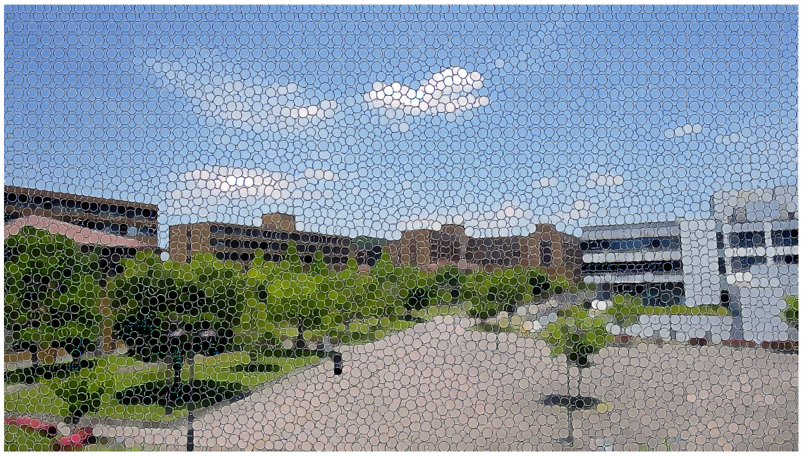

(b) machine learning approach

Figure 4.17: The generated circle tile art image for a Full HD image (1920 × 1080).

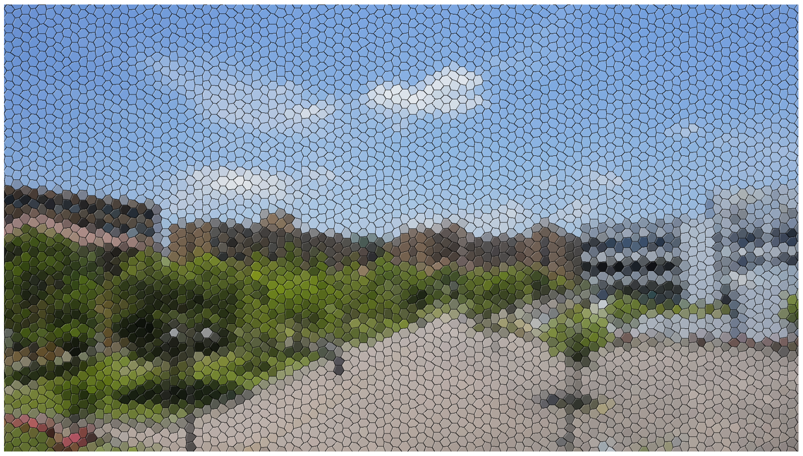


Figure 4.18: Mosaic image generated by GIMP for a Full HD image(1920 × 1080).

Table 4.1: The number of pasted patterns and the total error of generated tile art images for square patterns

tile	image size	greedy approach		machine learning approach
		# pasted patterns	total error	total error
square	256×256	960	13,335.4	13,464.7
	512×512	2,587	49,407.7	57,982.5
	1024×1024	9,936	231,624.1	229,231.0
	1920×1080	16,037	358,615.5	409,735.2
	2048×1536	28,346	538,732.2	539,549.4
	4096×3072	128,725	2,640,159.2	2,682,169.8
circle	256×256	971	13,893.8	13,961.4
	512×512	2,642	51,942.8	57,544.4
	1024×1024	11,824	245,532.5	244,774.5
	1920×1080	19,458	393,821.8	410,594.8
	2048×1536	30,894	591,093.6	551,336.0
	4096×3072	160,055	2,821,585.3	2,781,416.7

Regarding the number of pasted patterns in the greedy approach, circle tile patterns are pasted more than square tile patterns since the size of circle patterns is less than that of square patterns. We note that the total error of the both approaches strongly depends on structural elements in an input image. For example, when an input image mainly consists of straight borders, the total error of the square tile image tends to be smaller than that of the circle tiles since the straight borders can be easily represented by the square tiles. Also, the networks in the machine learning approach are trained so as to generate tile art images that are close to tile art images generated by the greedy approach, but so as not to minimize the total error. Therefore, the total error of the machine learning approach does not show the quality of tile art images directly. However, since the total error shows the difference from an input image, there are not large difference between tile art images generated by the greedy approach and the machine learning approach because the difference of the total errors is small.

4.4.4 Evaluation of computation time

Finally, we have evaluated the computation time of the greedy approach and the machine learning approach. We have used 4 Intel Xeon E7-8870V4 CPUs running in 3.0GHz with 1TB memory for CPU implementations. Each CPU has 20 physical cores each of which acts 2 logical cores by hyper-threading technology, that is we use 80 physical cores or 160 logical cores in total. We have implemented two CPU implementations of the greedy algorithm: *the sequential CPU implementation* and *the parallel CPU implementation*. In the sequential CPU implementation, a single thread is used, whereas in the parallel CPU implementation, 160 threads are used. In the parallel CPU implementation, each thread concurrently finds $q_{i,j}$'s at lines 4 and 12 in Algorithm 1. To implement parallel execution of threads, we have used OpenMP 3.1 [65]. The source code programs of the CPU implementations are compiled by gcc version 4.8.5 with -O2 and -fopenmp options. In addition, for the GPU implementation of the greedy approach, we have used NVIDIA TITAN V GPU with 5120 processing cores running in 1.37GHz with 16GB memory The source code program of the GPU implementation is compiled by nvcc version 9.0.176 with -O2 and -arch=sm_70 options. On the other hand, for the machine learning approach, we have used the same computing platform as the GPU implementation of the greedy approach. We have implemented and evaluated the machine learning approach using Python 3.5.2 with TensorFlow version 1.12.0 [2].

Table 4.2 shows the computation time of generating tile art images. In the computation time of the GPU implementations, data communication time between the host PC and the GPU is included. With regard to the greedy approach, the GPU implementation is much faster than the CPU implementations. In both of the parallel CPU implementation and the GPU implementation, we perform the computation of $q_{i,j}$'s in parallel. Furthermore, in the GPU implementation, each computation of $q_{i,j}$'s is concurrently executed by threads in a CUDA block. On the other hand, compared with the parallel CPU implementation, the GPU implementation runs at most 13.9 times faster. Also, in circle tile art image generation, the computation time of the GPU implementation reduced by a factor up to 283 over the sequential implementation. The GPU implementation runs at most 16.2 times faster than the parallel CPU implementation.

In the machine learning approach, we can see that it is much faster than the greedy approach. Quite surprisingly, the machine learning approach takes approximately 1 second even for the 4K image (4096×3072). The direct comparison of the computation time may be pointless since the resulting images are not equivalent between them. However, the machine learning approach can generate a tile art image 205 to 580 times faster than the GPU implementation of the greedy approach.

4.5 Further improvement for the machine learning approach using iterative inference

We have proposed the machine learning approach of generating a tile art image using the conditional generative adversarial networks in Section 4.3. However, in the generated tile art images, some tiles have lack of edge and noises that are not included in the greedy approach. In this section, we propose an improvement technique of the machine learning approach to generate a tile art image. The technique is very simple; a generated tile art

Table 4.2: The computation time in seconds of generating tile art images for square and circle patterns

tile	image size	greedy approach			machine learning approach
		CPU 1 thread	CPU 160 threads	GPU	GPU
square	256 × 256	1404.75	69.02	4.98	0.011
	512 × 512	2729.82	105.34	9.78	0.022
	1024 × 1024	12376.38	365.97	40.64	0.070
	1920 × 1080	18355.11	517.92	59.41	0.138
	2048 × 1536	32598.81	881.20	110.82	0.222
	4096 × 3072	181632.76	4561.27	571.18	1.040
circle	256 × 256	490.93	23.92	2.26	0.011
	512 × 512	1206.47	73.99	4.57	0.021
	1024 × 1024	6608.46	208.30	23.73	0.070
	1920 × 1080	9528.28	249.75	33.56	0.139
	2048 × 1536	15261.10	415.11	60.74	0.220
	4096 × 3072	108354.55	2445.75	395.30	1.037

image obtained by the inference of the generator is given to the generator again as an input image, and this iteration of inference is repeated several times as illustrated in Figure 4.19. The aim of this technique is that the generate network further enhances the shapes of tiles in the generated image.

Figures 4.20 and 4.21 show the generated tile art images using the iterative inference technique for square and circle tile patterns, respectively. From the figures, we can see that characteristics of tiles are enhanced using the this technique compared with non-iterative images shown in Figures 4.14(b) and 4.15(b). However, the images become dark gradually in each iteration due to the effect of edges of tiles that excludes in the original image. In our experiment, the iteration is repeated two or three times to obtain tile art images with well-balance between characteristic of tiles and whole brightness. As shown in Section 4.4.2, the discriminator learns to classify images as real or fake. More specifically,

Table 4.3: Output value of the discriminator with varying the number of inferences

inference	1	2	3	4
square tiles	0.5004	0.5265	0.5400	0.5483
circle tiles	0.4867	0.5145	0.5409	0.5392

the discriminator tries to output a probability assigned near to 1 for real images and near to 0 for fake images. In this work, real images correspond to tile art images generated by the greedy approach. On the other hand, fake images are output images by the generator in the machine learning approach. Therefore, as a criterion to evaluate generated tile art images, we use the output value of discriminator. Table 4.3 shows average output values of the discriminator to the generated tile art images with varying the number of inferences. Each value is an average of the output for $\frac{N}{8} \times \frac{N}{8}$ subimages of generated images in Figures 4.14(b), 4.15(b), 4.20 and 4.21. According to the table, the value without iterative inference is smaller than those with iterative inference. This fact means that the discriminator judges that the output images with iterative inference are more realistic tile art images. On the other hand, the computation time simply increases in proportion to the number of iterations. However, as shown in Table 4.2, since the computation time of the machine learning approach is extremely short, the computation time of this improvement technique is short enough for practical cases.

4.6 Conclusion

In this chapter, we have proposed two tile art image generation methods using greedy approach and machine learning approach. The greedy approach is inspired by characteristic of the human visual system and the resulting tile art images well-reproduce the original

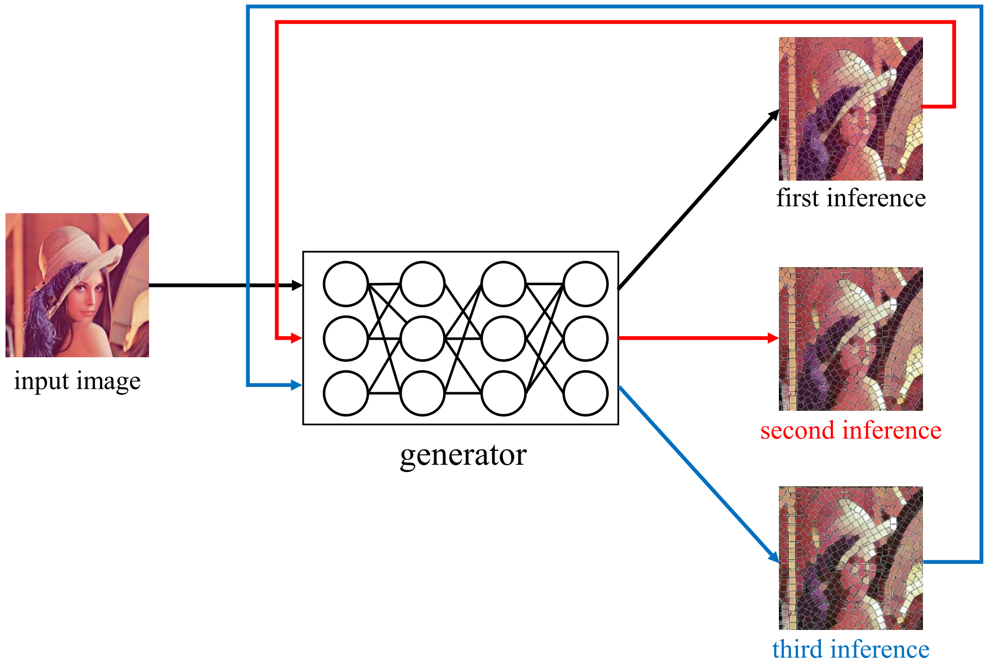


Figure 4.19: The iterative inference for the machine learning approach. Note that the generator is already trained and can generate tile art images. The generated tile art image is inputted to the generator again as an input image.

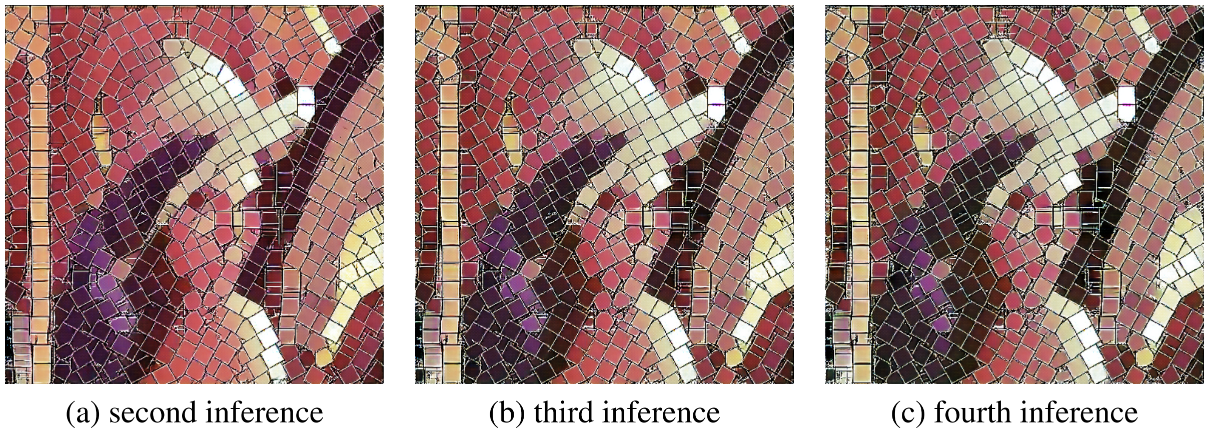
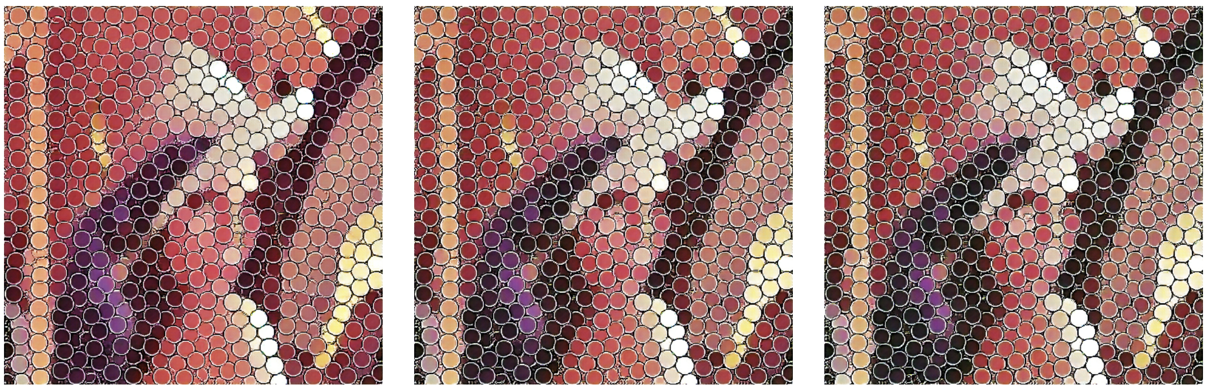


Figure 4.20: The generated tile art images for square patterns using iterative inferences. We can see that the characteristics of tiles are enhanced. However, the images become dark gradually in each iteration due to the effect of edges of tiles that excludes in the original image.



(a) second inference (b) third inference (c) fourth inference

Figure 4.21: The generated tile art images for circle patterns using iterative inferences. We can see that the noise in the images are disappeared by this technique.

images without any image feature extraction techniques such as edges, corners, among others. Also, to accelerate tile art image generation by this approach, we have proposed its parallel algorithm and implemented it on the GPU. The experimental results show that the GPU implementation can achieve a speed-up factor up to 318 over the sequential CPU implementation. On the other hand, the machine learning approach approximates the tile art image generation of the greedy approach using the deep neural networks. The proposed network can generate tile art images that have the structure of tiles and reproduce the original images well. As regards generating time, the GPU implementation of the greedy approach takes 4561 seconds to generate a tile art image of size 4096×3072, whereas the machine learning approach takes approximately 1 second. Furthermore, we have proposed an improvement technique of the machine learning approach to generate a tile art image. In the technique, a generated tile art image obtained by the inference of the generator is given to the generator again as an input image, and this iteration of inference is repeated several times. As a result, this technique can generate high quality tile art images with less noise and pronounced tiles.

Chapter 5

A novel structured sparse fully-connected layer in CNNs

5.1 Introduction

In recent, the network ability has become stronger. However, these CNNs need a large amount of memory to store their network parameters. We note that almost of CNNs memory usage is occupied by the FCLs. For example, in the case of VGG-16, there are 138.36 million parameters consisting of 14.71 million parameters in the convolutional layers and 123.64 million parameters in the FCLs. When each parameter is a 32-bit floating point number, approximately 553 MByte memory is required to store them. To run these applications, we need to store the network parameters into the device memory. Therefore, devices with a small amount of memory such as embedded devices may not be able to perform CNN applications. Therefore, the compact models with a small network size are often desired for the practical applications.

To solve this problem, several researches to reduce the amount of memory consumption of CNNs such as pruning [13, 14, 15, 16, 17, 18, 66] and structured sparsity [19, 67, 68, 20] have been devoted. Structured sparsity approaches aim to reduce the number of network

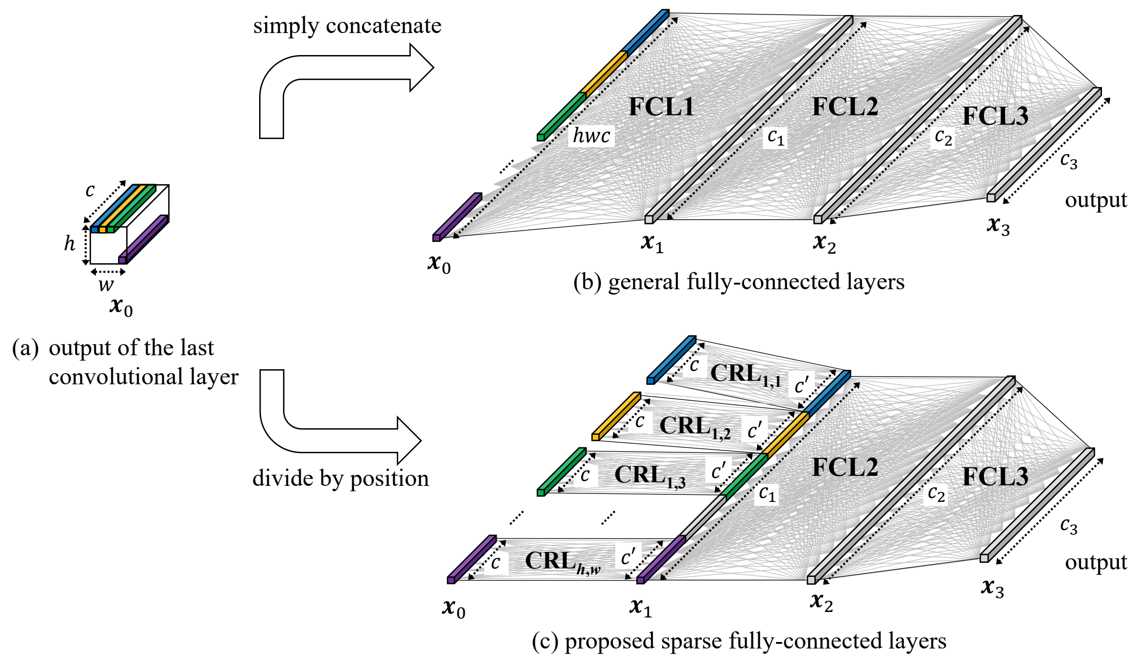


Figure 5.1: Architectures of FCLs in convolutional neural networks. The output of the last convolutional layer (a) in the general FCLs (b) is simply concatenated. On the other hand, in the proposed sparse FCLs (c), the output of the last convolutional layer (a) is divided by each position of the feature maps.

parameters by replacing the network architecture to smaller and more efficient one. On the other hand, pruning approaches aim to reduce the number of network parameters by removing unimportant network weights. However, in many cases, pruning approaches produce irregular network structures. Such irregularity of weights prevents efficient parallel computation.

In this work, we propose a structured sparse FCL in CNNs that can be efficiently computed in parallel. Especially, we focus on the first FCL and make it sparse. Figure 5.1 is outlines of the general structure of FCLs and the proposed sparse one. In the general FCLs, the output of the last convolutional layer illustrated in Figure 5.1(a) is simply concatenated as the input of the first FCL. Each element in the output of the first FCL is connected to all input nodes of the first FCL. Note that the number of the output nodes of the last convolutional layer is larger than that of the FCLs generally. Hence, the number of parameters in the first FCL is larger than that in the second and third FCLs. For example, in the case of VGG-16, the numbers of parameters in the first, second, and third FCLs are 103 million, 17 million and 4 million, respectively.

In this work, therefore, we aim to reduce the number of parameters in the first FCL. In the general first FCL, each output element is fully-connected to all input elements regardless of the position and the channel in the feature maps of the output convolutional layer. However, the connections may not have a strong relation between distant elements in the feature maps, for instance, two distant elements in the upper-left and lower-right. Therefore, we eliminate the connections between elements in different position from the first FCL. In other words, the output elements are connected to the input elements only in the same position of the feature maps. Figure 5.2 illustrates the general first FCL and the proposed one. In this figure, we can see that the connections between the input and the output of the first FCL are removed except for those in the same positions. Here, if we look at the proposed architecture from another perspective, it can regard that the proposed first FCL is composed of hw networks each of which reduces the channel of the feature maps from c to c'. Hence, we call these networks in the proposed first FCL *Channel Reduction Layers* (*CRLs*). Each CRL is a small fully-connected network whose numbers of input and output nodes are c and c', respectively. Note that the numbers of input and output nodes in the general first FCL are hwc and hwc', respectively. The proposed first FCL has hw CRLs, therefore, the total number of parameters in the proposed first FCL is $hw \times (c \times c')$, while

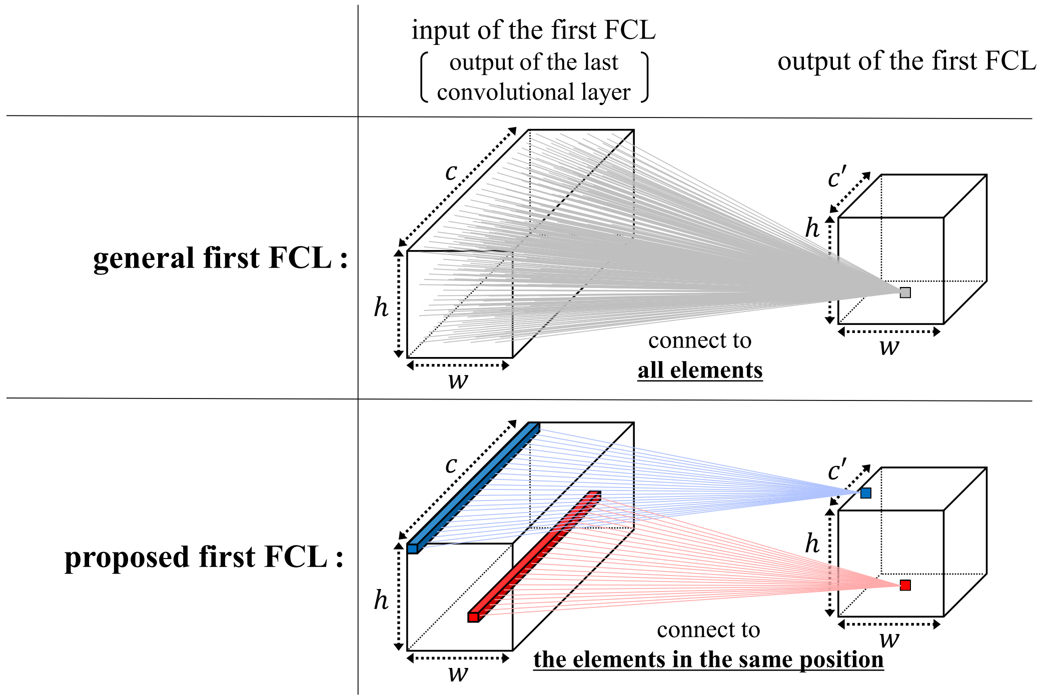


Figure 5.2: Connections of elements from the input of the first FCL (the output of the last convolutional layer) to the output of the first FCL. The connections between distant elements in the feature maps may not have a strong relation, therefore, we eliminate them from the general first FCL.

that in the general first FCL is $hwc \times hwc'$. Thus, we can reduce the number of parameters to $\frac{1}{hw}$ by replacing the general first FCL to the proposed one. Our proposed approach can be applied to the CNNs which have three FCLs such as AlexNet and VGG-16. Note that it cannot be applied to the CNNs which have a single FCL such as GoogLeNet and ResNet.

We have evaluated the proposed architecture for image classification by applying to AlexNet and VGG-16 on ILSVRC-2012 dataset [23]. For AlexNet, we can reduce the number of parameters in the FCLs from 58.6 million to 4.0 million with top-1 accuracy and top-5 accuracy decrease by 0.68% and 0.19% compared to the original architecture, respectively. On the other hand, for VGG-16, we can reduce the number of parameters in the FCLs from 123.6 million to 5.8 million with 0.68% top-1 accuracy and 0.31% top-5 accuracy decreases compared to the original architecture. Also, to confirm that our proposed approach is applicable to practical image classification problems, we have trained the proposed models using transfer learning on three fine-grained datasets. The experimental results show that the proposed approach can achieve high test accuracy with high compression ratio on each dataset.

In addition, we propose an implementation of the above proposed sparse architecture on Graphics Processing Units (GPUs) using cuBLAS [6]. We have evaluated the computation time of forward and backward propagation for the FCLs using an NVIDIA GeForce RTX 2080 Ti GPU. When the batch size is 128, the implementations of forward and backward propagation for the proposed architecture achieves speed-up factors 12.33 and 11.09 for AlexNet compared to those for the original architecture, respectively. For VGG-16, the implementations of forward and backward propagation for the proposed architecture can run up to 13.19 and 16.79 times faster than those for the original architecture, respectively.

The remainder of this paper is organized as follows. Section 5.2 shows several works related to the reduction of the network size for CNNs. Section 5.3 presents the proposed sparse architecture. Section 5.4 shows the implementations of the proposed sparse architecture using cuBLAS. Section 5.5 presents the experimental results for AlexNet and VGG-16. Finally, Section 5.6 concludes our work.

5.2 Related works

With rapid development of the machine learning, many researches related to the reduction of network size have been devoted. In the following, we introduce two types of network size reduction, *network pruning* and *structured sparsity*.

Network Pruning: Network pruning is one of the most popular methods in network model compression. The main idea of this approach is removing unimportant network weights from the trained network and then re-training the pruned network. The above procedure is repeated until sufficient number of weights are eliminated and/or the test accuracy drops. Han et al. proposed a network pruning technique [13, 14]. Their experimental result shows that it achieves the improvement of the test accuracy as well as the reduction of weights in the same time. However, their approach results in producing the irregular weight matrices which prevent the efficient parallel computations. Therefore, they proposed a special hardware accelerator to compute such irregular weight matrices [15]. Other network pruning approaches which produce regular weight matrices have been proposed [16, 17, 18]. The idea of their approach is removing whole convolutional filter with small sum of weights, not some weights in the convolutional filter. Since their approach leads to almost regular weight matrices, there is no need to prepare any special hardware accelerators. The disadvantage of the filter level pruning approach is that the accuracy is lower compared to the element level pruning approach. Mao et al. explored the relation between granularity of pruning and test accuracy for the pruning approach [66]. They changed the granularity of pruning from fine-grain to course-grain and measured the test accuracy for CNNs. More specifically, the four types of granularity are used: element level (one-dimensional), vector level (two-dimensional), kernel level (three-dimensional) and filter level (four-dimensional). Their experimental results for famous CNNs such as AlexNet, VGGNet, GoogLeNet and ResNet50 show that the fine-grain pruning can achieve higher test accuracy than that of the course-grain pruning.

Structured Sparsity: Structured sparsity approaches aim to reduce weights in network by changing the structure of the network. These approaches can produce complete regular weight matrices by performing regularly sparsity, and then the hardware which performing inference and training can benefit of the efficient parallel computing. However, it is difficult to compress the models with no accuracy decrease compared to the network pruning approaches. In our proposed approach, this type of network size reduction is applied. One of the methods related to the structured sparsity is proposed by Wen et al., that compresses the model by replacing the weight value to zero in the convolutional filters in regularly [19]. Also, they proposed a GPU implementation of the compressed networks, and it achieved speed-up factor 3 compared to that of the non-compressed networks for AlexNet with a 2% top-1 accuracy decrease. Also, Novikov et al. achieved both of network size reduction and improvement of test accuracy using Tensor-Train format (TT format) [67]. TT format is a format that converts a large matrix to small ones. They applied this format to the weight matrices in VGG-16. As a result, they achieved 7.9 times reduction with 1.3% top-1 accuracy and 1.1% top-5 accuracy decrease and a speed-up factor 2.3 on the GPU. In the paper [68], Cyclic Sparsely Connected (CSC) layer was proposed. The CSC layer consists of two or three sparse weight matrices. In the experiment on AlexNet, this approach can reduce the size of the FCLs to 5% with 1.5% top-5 accuracy decrease, and the compressed network can run in a half resources (energy cost and memory space) compared to the original on the FPGA. Deng et al. proposed a model compress method PermDNN using permuted diagonal matrices [20]. The idea of PermDNN is to delete the weights which are not in diagonal or sub-diagonal components in regularly. This method can mainly be applied to not only CNNs, but also some types of DNNs. Their experimental result shows PermDNN can accelerate computation without significant test accuracy drop. Howard et al. proposed MobileNet v1 [69], v2 [70] and v3 [71] which aim to implement on the mobile devices mainly. They utilize a depthwise separable convolution algorithm. They reduce the network by separating the two-dimensional convolution to the depthwise convolution and the pointwise convolution. In addition, the other techniques such as global average pooling [72], inverted residual block and Squeeze-and-Excite module [73] are used. Using these networks, they achieved significant results for both of network size reduction and test accuracy.

Another method related to the reduction of the network size is quantization. In the quantization, model compression is achieved by reducing effective number of bits. Courbariaux et al. proposed a binarization method for network weights for the forward propagation on CNNs [74, 75]. They achieved the saving of memory utilization and the improvement of test accuracy. The quantization may be applicable to our proposed sparse models. However, in this paper, we focus on the reduction of the number of weight parameters only by changing network architecture.

5.3 Proposed sparse architecture

First, we propose a sparse CNN architecture that is suitable to the parallel computation. The proposed method focuses on *Fully-Connected Layers* (*FCLs*) which occupy most of the memory utilization in CNNs such as AlexNet [21] and VGG-16 [22]. We focus on the first FCL consisting of input nodes which corresponds to the output nodes of the last convolutional layer and output nodes. Every output node connects to all input nodes. In the proposed FCLs, the first FCL is replaced to *Channel Reduction Layers* (*CRLs*) each of which is composed of a small fully-connected network by considering the structure of the output of the last convolutional layer.

Figure 5.1 illustrates the network architectures of the FCLs in general CNNs and the proposed CNNs. Let x_0 be the output of the last convolutional layer. Also, let h, w and c be the height, width and channel of x_0. The general first FCL is composed of a fully-connected network with hwc input nodes and c_1 output nodes. The input nodes can be obtained by simply concatenating the output nodes of the last convolutional layer. On the other hand, the structure of the proposed first FCL consists of the hw CRLs. Each CRL is composed of a small fully-connected network whose size of the input channel and the output channel are c and c', respectively. In other words, each CRL reduces the nodes from c to c' with a fully-connected network. The inputs of the proposed first FCL can be obtained by dividing the output of the last convolutional layer based on each position in the feature maps. By replacing the general first FCL to the proposed first FCL, the number of weights is reduced from $hwc \times c_1 = hwcc_1$ to $hw\,(c \times c') = hwcc'$. We note that $hw \times c'$ equals to c_1, that is, c' equals to $\frac{c_1}{hw}$. Hence, when comparing to the general first FCL, the number of weights in the proposed one is reduced to $\frac{1}{hw}$. For simplicity, we assume that c_1 is divisible by hw. In addition, to achieve higher reduction rate of weights, we can decrease the numbers of the input channels in the second FCL and the third FCL. In the following, the two types of FCLs, that are the general FCLs and the proposed sparse FCLs, are explained in detail.

5.3.1 General fully-connected layers

In this subsection, we explain the general dense FCLs illustrated in Figure 5.1(b). The general dense FCLs consist of three general FCLs. In the FCL1, the output nodes of the last convolutional layer are fully-connected to those of the FCL1. The input of the FCL1 corresponds to the output of the last convolutional layer consisting of a three-dimensional tensor $\boldsymbol{x}_0 \in \mathbb{R}^{h\times w\times c}$. However, in the general FCLs, it is given to the FCL1 without any consideration about the structure of the feature maps as illustrated in Figure 5.1(b). In other words, the input is provided as a one-dimensional tensor $\boldsymbol{x}_0 \in \mathbb{R}^{hwc}$. Thus, the output of the FCL1 $\boldsymbol{x}_1 \in \mathbb{R}^{c_1}$, is defined by

$$\boldsymbol{x}_1(i) = \varphi(\sum_{l=1}^{hwc} \boldsymbol{W}_{\mathrm{FCL1}}^T(i,l)\boldsymbol{x}_0(l) + \boldsymbol{b}_{\mathrm{FCL1}}(i)), \tag{5.1}$$

where c_1 is the number of the output channel, $\boldsymbol{W}_{\mathrm{FCL1}} \in \mathbb{R}^{hwc\times c_1}$ is the weights and $\boldsymbol{b}_{\mathrm{FCL1}} \in \mathbb{R}^{c_1}$ is the biases of the FCL1. Also, $\varphi(\cdot)$ denotes an activation function such as sigmoid and ReLU function [32]. Note that we use ReLU activation function in this work. Also, the FCL2 and the FCL3 have the same structure as the FCL1. Therefore, the output of the FCL2 $\boldsymbol{x}_2 \in \mathbb{R}^{c_2}$ is defined by

$$\boldsymbol{x}_2(i) = \varphi(\sum_{l=1}^{c_1} \boldsymbol{W}_{\mathrm{FCL2}}^T(i,l)\boldsymbol{x}_1(l) + \boldsymbol{b}_{\mathrm{FCL2}}(i)), \tag{5.2}$$

where c_2 is the number of the output channel, $\boldsymbol{W}_{\mathrm{FCL2}} \in \mathbb{R}^{c_1\times c_2}$ is the weights and $\boldsymbol{b}_{\mathrm{FCL2}} \in \mathbb{R}^{c_2}$ is the biases of the FCL2. Also, the output of the FCL3 $\boldsymbol{x}_3 \in \mathbb{R}^{c_3}$ is defined by

$$\boldsymbol{x}_3(i) = \varphi(\sum_{l=1}^{c_2} \boldsymbol{W}_{\mathrm{FCL3}}^T(i,l)\boldsymbol{x}_2(l) + \boldsymbol{b}_{\mathrm{FCL3}}(i)), \tag{5.3}$$

where c_3 is the number of the output channel, $\boldsymbol{W}_{\mathrm{FCL3}} \in \mathbb{R}^{c_2\times c_3}$ is the number of weights and $\boldsymbol{b}_{\mathrm{FCL3}} \in \mathbb{R}^{c_3}$ is the biases of the FCL3.

5.3.2 Proposed sparse fully-connected layers

Next, we explain the proposed sparse FCLs illustrated in Figure 5.1(c). The proposed FCLs are composed of the proposed FCL1, the general FCL2 and the general FCL3. The proposed FCL1 is composed of the hw CRLs each of which is a fully-connected network whose number of the input and the output nodes are c and c', respectively. In other words, each CRL performs the channel reduction of the feature maps from c to c'. Therefore, in total, the proposed FCL1 performs the dimension reduction of the output of the last convolutional layer $\boldsymbol{x}_0 \in \mathbb{R}^{h \times w \times c}$, and then output a three-dimensional tensor $\boldsymbol{x}_1 \in \mathbb{R}^{h \times w \times c'}$. The input of the FCL2 can be obtained by simply concatenating the output of the proposed FCL1. Note that the total numbers of the input nodes and the output nodes of the proposed FCL1 are the same as the general FCL1 described in Subsection 5.3.1. We define the output of the proposed FCL1 $\boldsymbol{x}_1 \in \mathbb{R}^{h \times w \times c'}$ as follow;

$$\boldsymbol{x}_1(i, j, k) = \varphi(\sum_{l=1}^{c'} \boldsymbol{W}^T_{\mathrm{CRL}_{i,j}}(k, l)\boldsymbol{x}_0(i, j, l) + \boldsymbol{b}_{\mathrm{CRL}_{i,j}}(k)), \tag{5.4}$$

where $\boldsymbol{W}_{\mathrm{CRL}_{i,j}} \in \mathbb{R}^{c \times c'}$ is the weights and $\boldsymbol{b}_{\mathrm{CRL}_{i,j}} \in \mathbb{R}^{c'}$ is the biases of $\mathrm{CRL}_{i,j}$ $(1 \leq i \leq h, 1 \leq j \leq w)$. As regards FCL2 and FCL3 in the proposed FCLs, the architectures are the same as those of the general FCL2 and FCL3. Therefore, they can be computed by the same way as Eqs. (5.2) and (5.3).

5.4 Implementation of proposed architecture using cuBLAS

In this section, we show implementations of the dense FCLs and CRLs described in Section 5.3. To focus on the proposed method, we discuss them of only the FCLs. Therefore, the implementations of forward and backward propagation for the FCLs are explained. Also, we assume that NHWC data format, which is one of data formats used in CNNs and supported in TensorFlow [2] and Tensor Core [76], is used in the last convolutional layer. In the data format, N represents the number of images in a batch, H and W represent image size of vertical and horizontal direction, respectively, and C represents the number of channels. This data format is one form of data representation that describes how four-dimensional arrays are stored in one-dimensional memory address space. The elements in right side of this format will store in one-dimensional memory address space firstly. Hence, in NHWC data format, the elements are stored in one-dimensional memory address space from those of channel-direction at first.

Our implementations use cuBLAS [6] which is a library supporting basic linear algebra subprograms (BLAS) on the CUDA runtime. cuBLAS is highly tuned for NVIDIA GPUs, and is used as an accelerator for deep learning frameworks such as TensorFlow [2] and MXNet [4]. In this work, we have used two cuBLAS functions: level 2 functions and level 3 functions. The level 2 functions perform matrix-vector multiplication and vector-vector addition. On the other hand, the level 3 functions perform matrix-matrix multiplication and matrix-matrix addition.

5.4.1 Forward propagation

In this subsection, we explain implementations of forward propagation. First, the existing GPU implementation for the general FCLs illustrated in Figure 5.1(b) is described. In general, the dense FCL can be computed by performing a matrix-vector multiplication or a matrix-matrix multiplication. As shown in Eq. (5.1), the general FCL1 needs to perform a matrix-vector multiplication for the weight matrix $\boldsymbol{W}_{\mathrm{FCL1}}$ and the input activation vector $\boldsymbol{x}_0$, and a vector-vector addition of its result and the bias vector $\boldsymbol{b}_{\mathrm{FCL1}}$. Thus, we use the level 2 function of cuBLAS. However, when the batch size is greater than 1, we need to iterate the above process for the batch size. To avoid this iteration, multiple matrix-vector multiplications are considered as one matrix-matrix multiplication. The input activation vectors are converted to a batched input activation matrix $\boldsymbol{X}_0$ by concatenating its elements, and the bias vector is extended to a batched bias matrix $\boldsymbol{B}_{\mathrm{FCL1}}$ by broadcasting its elements. Then, we can obtain the results of the general FCL1 for the batch size by applying the level 3 function to the matrix-matrix multiplication for the weight matrix $\boldsymbol{W}_{\mathrm{FCL1}}$ and the batched input activation matrix $\boldsymbol{X}_0$, and the addition of the batched bias matrix $\boldsymbol{B}_{\mathrm{FCL1}}$. This process of the FCL1 can be applied to the FCL2 and FCL3 shown in Eqs. (5.2) and (5.3), respectively.

Next, an implementation of forward propagation for the proposed CRLs illustrated in Figure 5.1(c) is described. Note that we need to compute hw full connections since there are hw small fully-connected networks in the proposed FCL1. When the batch size is 1, the CRLs can be computed by hw matrix-vector multiplications. On the other hand, when the batch size is greater than 1, we need to convert the hw input activation vectors to the hw batched input activation matrices for efficient computation. In this conversion, we have used the same idea described above. Thereby, the CRLs for the batch size can be computed by hw matrix-matrix multiplications. Figure 5.3 shows the efficient computation approach of the CRLs for a batch. In this figure, the forward computation for the hw full connec-

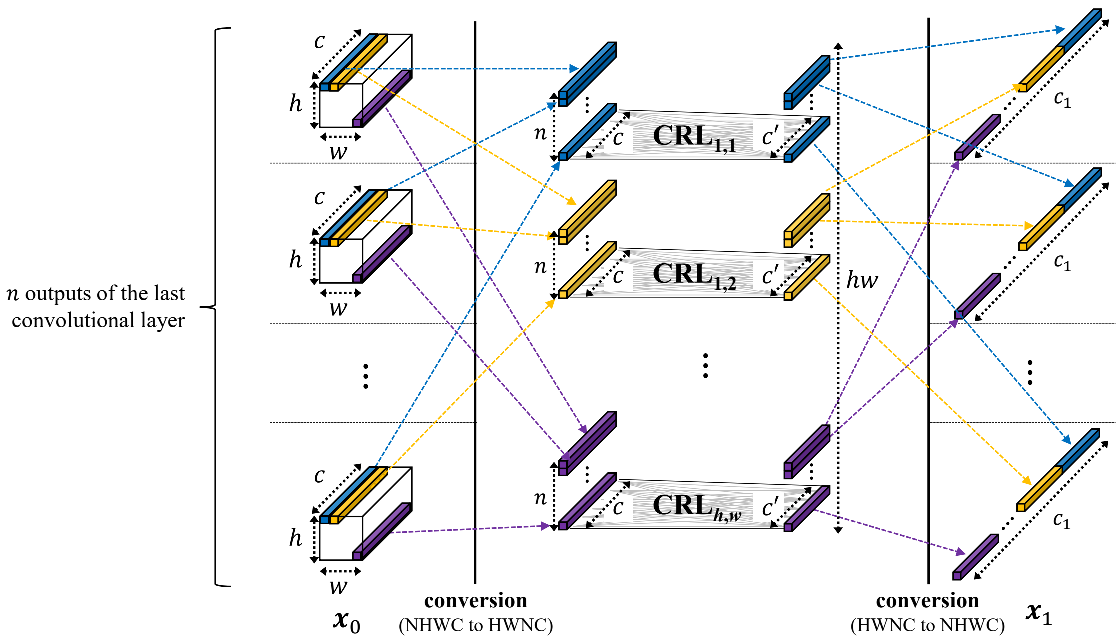


Figure 5.3: The idea of computation for the CRLs in the implementation using cuBLAS when the batch size is n. To compute the process of the n CRLs, the memory conversions (NHWC to HWNC, and HWNC to NHWC) are performed. Thereby, the computation of the n CRLs can be performed by a cuBLAS's function at once.

tions of the batch size n is performed as hw matrix-matrix multiplications. Here, we note that each hw matrix-vector or matrix-matrix multiplication can be computed independently. Thus, for any batch size, the CRLs can be computed by a stride batched function which is cuBLAS function computing the independent multiple matrix-vector or matrix-matrix multiplications in efficiently. However, when the batch size is greater than 1, this implementation for the CRLs requires to convert the data format from NHWC to HWNC and from HWNC to NHWC. Therefore, we have implemented two data format conversion kernels using CUDA [5] that is a parallel computing platform for NVIDIA GPUs. In addition, we have also implemented the ReLU activation function [32] using CUDA. Note that the order of NHWC is equal to that of HWNC when the batch size is 1. In other words, the GPU implementation of CRL with batch size 1 does not require the data format conversions.

5.4.2 Backward propagation

In this subsection, we explain implementations of backward propagation using cuBLAS. At first, we describe the implementations when the batch size is more than one. After that, we describe them when the batch size is one. Here, we assume that the activation gradients for the last FCL are obtained in advance.

In the following, the existing implementation of backward propagation for the general FCLs is described. The backward propagation consists of two steps, activation gradient computation and weight gradient computation. In general, these steps are performed by a matrix-matrix multiplication when the batch size is greater than 1. We define the batched activation gradient for the last convolutional layer as $\Delta \boldsymbol{X}_0$, and the batched activation gradient and the weight gradient for the l-th FCL as $\Delta \boldsymbol{X}_l$ and $\Delta \boldsymbol{W}_{\mathrm{FCL}_l}$ ($l = 1, 2, 3$), respectively. The l-th batched activation gradient $\Delta \boldsymbol{X}_l$ can be computed as follow;

$$\Delta \boldsymbol{X}_l = \boldsymbol{W}_{\mathrm{FCL}_l} \Delta \boldsymbol{X}_{l+1}.$$

On the other hand, the weight gradient $\Delta \boldsymbol{W}_{\mathrm{FCL}_l}$ can be computed as follow;

$$\Delta \boldsymbol{W}_{\mathrm{FCL}_l} = \boldsymbol{X}_l (\Delta \boldsymbol{X}_{l+1})^T.$$

Therefore, to process the backward propagation for the general FCLs, we apply the level 3 function of cuBLAS to these matrix-matrix multiplications.

Next, an implementation of backward propagation using cuBLAS for the proposed FCLs is described. We define the batched activation gradients for the last convolutional layer as $\Delta \boldsymbol{X}_{0_{i,j}}$ and the weight gradients for the CRLs as $\Delta \boldsymbol{W}_{\mathrm{CRL}_{i,j}}$ ($1 \leq i \leq h, 1 \leq j \leq w$). Since the CRLs consists of hw small fully-connected networks, hw matrix-matrix multiplications must be performed for each the activation gradient and the weight gradient computation. In the proposed architecture, the batched activation gradient $\Delta \boldsymbol{X}_{0_{i,j}}$ can be obtained by repeating the following equation hw times;

$$\Delta \boldsymbol{X}_{0_{i,j}} = \boldsymbol{W}_{\mathrm{CRL}_{i,j}} \Delta \boldsymbol{X}_{1_{i,j}},$$

where $\Delta \boldsymbol{X}_{1_{i,j}}$ can be obtained by simply dividing $\Delta \boldsymbol{X}_1$ per c_1 elements. On the other hand, the weight gradient matrices $\Delta \boldsymbol{W}_{\mathrm{CRL}_{i,j}}$ can be obtained by repeating the following equation hw times;

$$\Delta \boldsymbol{W}_{\mathrm{CRL}_{i,j}} = \boldsymbol{X}_{0_{i,j}} (\Delta \boldsymbol{X}_{1_{i,j}})^T.$$

In order to compute these hw matrix-matrix multiplications efficiently, we use the stride batched function of cuBLAS. Also, same as the forward propagation, the data format conversions (NHWC to HWNC and HWNC to NHWC) are required in the activation gradient computation step when the batch size is greater than 1. Therefore, we use the same kernels as for the forward propagation. We note that they are not required when the batch size is 1 since the order of NHWC format will be equal to that of NCHW format. The computations of backward propagation in the FCL2 and the FCL3 are the same as those for the general FCLs. In addition, we have implemented a kernel which compute the backward propagation of ReLU activation function using CUDA.

Here, we describe the implementation of backward propagation using cuBLAS when the batch size is one. In the existing implementation for the dense FCLs, the activation gradient computation can be performed by a matrix-vector multiplication for the weight matrix and the activation gradient vector. On the other hand, the weight gradient computation can be performed by a multiplication of the input activation vector and transposed activation gradient vector, and it results in a weight gradient matrix. Thus, we have used the level 2 function for the activation gradient computation and the level 3 function for the weight gradient computation. Next, an implementation for the proposed FCL1 is described. In the CRLs, hw matrix-vector multiplications of the weight matrices and the activation gradient vectors are performed in the activation gradient computation step. On the other hand, the weight gradient computation is performed by hw multiplications of the input activation vector and transposed activation gradient vector. These computations contain hw multiplications, hence, we have used the stride batched function.

5.5 Experimental results

In this section, we show the experimental results to evaluate the proposed sparse architecture using AlexNet[21] and VGG-16[22]. AlexNet is the winner of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) on 2012 that is competition for image classification accuracy. The network of AlexNet is composed of 5 convolutional layers and 3 FCLs. On the other hand, VGG-16 is one of the networks called VGGNet which is the first runner-up of ILSVRC on 2014 and has significantly improvement of accuracy over AlexNet. The structure of VGG-16 is composed of 13 convolutional layers and 3 FCLs. Since these networks have relatively simple architectures, they have been commonly used as benchmarks for machine learning algorithms. The original FCLs for both of AlexNet and VGG-16 are the general FCLs shown in Figure 5.1(b), where $c_1 = 4096$ and $c_2 = 4096$. In addition, to make the effectiveness of the proposed approach clear, the original architecture is simply compressed by reducing the channels c_1 and c_2. In this section, we refer these simply compressed models as *simple compression models*, while we call the proposed models, illustrated in Figure 5.1(c), *proposed compression models*. The last convolutional layer shown in Figure 5.1(a) is given by $h = 6$, $w = 6$, $c = 256$ for AlexNet and $h = 7$, $w = 7$, $c = 512$ for VGG-16. As the experimental environments, we have used CUDA 10.0, cuDNN 7.4.2, cuBLAS 10.0, python 3.6.7 and TensorFlow 1.13.1. Also, we have used Intel Core i9-9960X CPU and 4 NVIDIA RTX 2080 Ti GPUs.

We have trained AlexNet models using a single GPU because the memory usage of the parameters is much smaller than the size of the GPU memory. Also, we have used a single GPU to perform the inference computation. The training time of the original, the simple compression and the proposed compression models for AlexNet is approximately 65 hours each using a single GPU. Even if the network size is reduced, the training time was not changed. The reason of this is that the convolutional layers occupies most of the training time of CNNs. In other words, the ratio of the training time of the FCLs is much smaller than that of the convolutional layers. Therefore, the training time of CNNs was not changed in spite of the reduction of the parameters in the FCLs.

On the other hand, in VGG-16, the models have been trained using four GPUs because the memory usage of the parameters is much larger than the size of the GPU memory. Also, we have used four GPUs to perform the inference computation. As same as the case of AlexNet, the convolutional layers occupy most of the computation time. Therefore, the network reduction of the FCLs does not shorten the computation time. However, every GPU needs to broadcast updated parameters to the other GPUs via the PCI Express bus for each training step. The memory bandwidth of the PCI Express bus is lower than that of the GPU. Therefore, the training time depends on the number of the updated parameters. In other words, the training time is shorter if the compression ratio is higher. As a result, when we trained the original, the simple compression and the proposed compression models, the training time of each model was in the range of 60 to 125 hours.

5.5.1 Test accuracy and compression ratio

In this subsection, we show the training results of the proposed compression models for AlexNet and VGG-16. After that, we evaluate the proposed compression models compared to the original, the simple compression models. We have trained the models on ILSVRC-2012 dataset [23]. ILSVRC-2012 dataset consists of one million or more training images and 50000 evaluation images that are classified into 1000 classes such as goldfish, balloon and microwave. Top-1 accuracy and Top-5 accuracy have been measured on the evaluation images.

AlexNet: In this part, we show the training results for AlexNet on ILSVRC-2012 dataset. We have trained both compression models on the hyper-parameters shown in [21] except for using gradual warm-up technique [35]. Also, in the proposed compression models and the simple compression models with $c_1 = 2304$ and $c_1 = 1152$, we set the dropout rate to 0.25 while that of the other models are set to 0.5. In the inference phase, we have used the standard 10-crop testing [21] to evaluate the test accuracy of the models. The weights of the CRLs have been initialized with a normal distribution with the zero mean and 0.01 variance. Also, the values of biases have been initialized to zero.

Table 5.1: Test accuracy and the number of parameters for AlexNet on ILSVRC-2012 dataset

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	top-5 accuracy (%)
original	4096	4096	58.6M	100.0	58.23	80.77
simple compression	4608	4096	65.4M	111.6	58.43	80.93
	4608	2048	54.0M	92.0	58.17	80.82
	4608	1024	48.2M	82.2	57.02	80.17
	2304	2048	28.0M	47.8	57.30	80.39
	2304	1024	24.6M	42.0	56.08	79.58
	1152	1024	12.8M	21.9	54.14	78.31
proposed compression	4608	4096	24.2M	41.7	57.82	80.60
	4608	2048	12.7M	21.7	57.79	80.47
	4608	1024	6.9M	11.8	57.45	80.24
	2304	2048	7.4M	12.5	57.96	79.84
	2304	1024	4.0M	6.8	56.77	79.68
	1152	1024	2.5M	4.3	55.17	78.90

Table 5.1 shows the training result of the original, the simple compression and the proposed compression models. The number of parameters and the compression ratio are only for the FCLs excluding the convolutional layers. We can see that the number of parameters on the simple compression models cannot be reduced drastically in any channels. This is because of its structure that the output nodes of the FCL1 must be connected to all nodes in the output of the last convolutional layer. Nevertheless, the loss of test accuracy is significant in the simple compression models which have high compression ratio.

By contrast, the proposed compression models have higher compression ratio than the simple compression models in the same channels since the proposed FCL1 has $\frac{1}{hw}$ weights compared to the general FCL1 in the simple compression models. Also, the test accuracy of the proposed compression models are higher than those of the simple compression models with a similar number of parameters. It seems that the reason for high test accuracy of the proposed compression models is that the proposed architecture can promote the training by finding an advantage in the channels of feature maps. According to the table, the proposed approach can reduce the network size up to 11.8% compression ratio compared to the original model with almost the same test accuracy. On the other hand, the test accuracy of the proposed models whose compression ratios are 6.8% and 4.3% is slightly decreased compared to that of the original model. Therefore, as a further approach, we adopted *fine-tuning* technique [77] to improve the accuracy for such cases.

Fine-tuning is a training technique that uses the weights and the biases of the pre-trained model as initial values of a distinct model. We have used the trained original model of AlexNet as the pre-trained model, and initialized the parameters of the convolutional layers. The CRLs and FCLs are initialized by the same manner as described above. The learning rate of the convolutional layers initialized with the pre-trained model were decreased from 0.01 to 0.001 to restrict major changes. Other training details are the same as described in Krizhevsky et al. [21], except for reducing dropout rate from 0.5 to 0.25.

Table 5.2 shows the training result of the proposed compression models with fine-tuning. According to the table, the proposed compression models achieve 1% test accuracy improvements in top-1 and top-5. Therefore, in the proposed compression model when c_1 is 2304 and c_2 is 1024, the test accuracy differences compared to the original are restricted to 0.68% for top-1 and 0.19% for top-5. On the other hand, those of with the channel $c_1 = 1152$ and $c_2 = 1024$ are 2.17% for top-1 and 1.25% for top-5. From the test accuracy aspect, we should choose the channel c_1 from 2304 or more for AlexNet on ILSVRC-2012 dataset. As a result, the proposed approach achieves high model compression ratio in almost the same accuracy compared to the original by using fine-tuning.

Table 5.2: Test accuracy improvement with fine-tuning

model	c_1	c_2	top-1 accuracy (%)	top-5 accuracy (%)
original	4096	4096	58.23	80.77
proposed compression (non-fine-tuning)	2304	1024	56.77	79.68
	1152	1024	55.17	78.90
proposed compression (fine-tuning)	2304	1024	57.55	80.58
	1152	1024	56.06	79.52

VGG-16: Next, we show the training result for VGG-16 on ILSVRC-2012 dataset. We have trained the original, the simple compression and the proposed compression models using the pre-trained VGG-11 model. The pre-trained VGG-11 model had been trained on the hyper-parameters shown in [22]. After training, some convolutional layers in VGG-16 were initialized with all convolutional layers in VGG-11. The VGG-16's convolutional layers which are not included in VGG-11, the CRLs and the FCLs have been initialized with a normal distribution with the zero mean and 0.01 variance. Also, the values of biases have been initialized to zero. In the simple compression and the proposed compression models, the hyper-parameters are same as shown in [22] except for using gradual warmup. Also, in the simple compression and the proposed compression models, we set the dropout rate to 0.25 while that of the original model are set to 0.5. In the inference phase, we have used the standard 10-crop testing[21].

Table 5.3 shows the test accuracy and the number of parameters for VGG-16 in the original, the simple compression and the proposed compression models. We have trained the simple compression model when c_1 and c_2 are 256 to compare with the proposed compression models which have high compression ratio. According to the table, the compression ratios of the simple compression models are lower except for the model when the channels

Table 5.3: Test accuracy and the number of parameters for VGG-16 on ILSVRC-2012 dataset

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	top-5 accuracy (%)
original	4096	4096	123.6M	100.0	69.31	88.85
simple compression	3136	2048	87.2M	70.5	69.23	88.91
	3136	1024	82.9M	67.1	68.85	88.55
	1568	1024	42.0M	34.0	68.54	88.30
	256	256	6.7M	5.5	67.13	87.39
proposed compression	6272	4096	33.0M	26.7	68.60	88.35
	6272	2048	18.1M	14.7	68.40	88.52
	6272	1024	10.7M	8.6	68.73	88.60
	3136	2048	10.1M	8.2	68.61	88.56
	3136	1024	5.8M	4.7	68.63	88.54
	1568	1024	3.4M	2.8	68.02	88.07

are reduced significantly. However, since we have trained these models using fine-tuning, the accuracy for the model with 34% or more compression ratio are almost the same as those for the original model. By contrast, in the model with the 5.5% compression ratio, the top-1 accuracy and the top-5 accuracy are decreased by 2.18% and 1.46%, respectively. This is because of restriction of the network representation ability caused by the significant reduction of its output nodes in the FCL1 and the FCL2.

On the other hand, the proposed compression models achieve high compression ratios without significant decrease of the number of the output nodes in the FCL1 and the FCL2. Therefore, the proposed model with 4.7% compression ratio can keep the test accuracy in less than 1% while the simple compression model with 5.5% compression ratio cannot. However, the top-1 accuracy of the proposed model when the channels $c_1 = 1568$ and $c_2 = 1024$ is decreased by 1.29%. Considering the test accuracy, the channel c_1 in the proposed architecture should be 3136 or more for VGG-16 on ILSVRC-2012 dataset. As a result, our proposed approach can achieve high model compression with a few test accuracy decrease even for VGG-16 network.

Now, let us consider the number of parameters in the whole network including the convolutional layers for VGG-16. The total number of parameters in the whole non-compressed network is 138.36 million. In the original model, the numbers of parameters in the convolutional layers and the FCLs are 14.71 million and 123.64 million, respectively. In other words, the parameters in the FCLs account for 89% of the total. On the other hand, the number of parameters in the proposed compression model's FCLs when c_1 is 3136 and c_2 is 1024 is reduced from 123.64 million to 5.84 million. Therefore, the number of parameters in the proposed network including the convolutional layers is only 20.55 million. Considering the actual memory usage to store the models when each parameter is a 32-bit floating point number, we can reduce it from 553 MBytes to 82 MBytes. Therefore, using the proposed sparse approach, it may be possible to execute even an embedded system with limited memory usage.

The comparison with the sparse fully-connected layers divided by channel for VGG-16: In the proposed compression models, we divided the output of the last convolutional layers by position of the feature maps as shown in Figure 5.1. However, by dividing the output of the last convolutional layers by channel as illustrated in Figure 5.4, we can consider this structure is the structured sparsity as well. Therefore, we compare the compression ratio and the accuracy of these structures. Table 5.4 shows the number of parameters in the three FCLs, the compression ratio compared to the original model and the test accuracy for VGG-16. From this table, we can see that the models divided by channel can compress well. However, if the compression ratio is higher, the drop of the test accuracy is signifi-

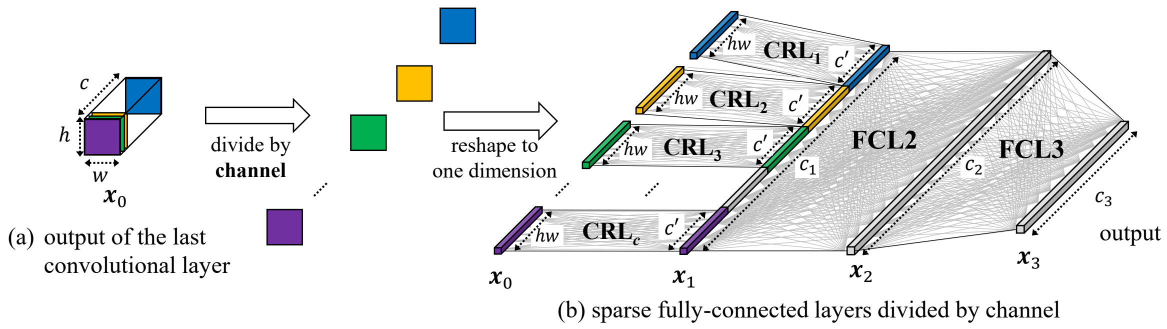


Figure 5.4: The sparse fully-connected layers divided by channel (b). The output of the last convolutional layer (a) is divided by channel, and each divided two-dimensional map is inputted to the CRL.

Table 5.4: The comparison with another compression structure for VGG-16

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	top-5 accuracy (%)
original	4096	4096	123.6M	100.0	69.31	88.85
divide by position (proposed)	6272	4096	33.0M	26.7	68.60	88.35
	6272	2048	18.1M	14.7	68.40	88.52
	6272	1024	10.7M	8.6	68.73	88.60
	3136	2048	10.1M	8.2	68.61	88.56
	3136	1024	5.8M	4.7	68.63	88.54
	1568	1024	3.4M	2.8	68.02	88.07
divide by channel	4096	4096	21.1M	17.5	68.64	88.26
	4096	2048	10.6M	8.6	68.18	88.52
	4096	1024	5.4M	4.4	68.21	88.33
	2048	2048	6.3M	5.1	67.94	88.03
	2048	1024	3.2M	2.6	67.51	87.63

cant. For example, when we compare the two structures in approximately 4.5 compression ratio, the test accuracy of the sparse model divided by channel is 0.41 and 0.23 lower for top-1 and top-5 compared to the proposed model, respectively. As a result, the performance of our proposed compression structure (divided by position) is higher than that of the other compression structure (divided by channel).

The comparison with other network reduction approaches: Here, we compare the test accuracy and compression ratio of the proposed sparse approach with those of the

Table 5.5: The comparison with other network reduction approaches for AlexNet

type of network reduction	approach	number of parameters	compression ratio (%)	top-1 accuracy (%)	top-5 accuracy (%)
—	original	62.4M	100.0	58.23	80.77
network pruning	network pruning [13]	6.7M	10.7	57.23	78.33
structured sparsity	PermDNN [68]	16.6M	26.7	—	79.90
	CSC Layers [20]	6.8M	11.0	—	77.60
	ours ($c_1 = 2304, c_2 = 1024$)	7.7M	12.4	57.55	80.58
	ours ($c_1 = 1152, c_2 = 1024$)	6.2M	10.0	56.06	79.52

Table 5.6: The comparison with other network reduction approaches for VGG-16

type of network reduction	approach	number of parameters	compression ratio (%)	top-1 accuracy (%)	top-5 accuracy (%)
—	original	138.3M	100.0	69.31	88.85
network pruning	network pruning [13]	10.0M	7.5	68.66	89.12
structured sparsity	TT-format [67]	35.6M	25.7	68.50	88.50
	ours ($c_1 = 3136, c_2 = 1024$)	20.5M	14.8	68.63	88.54
	ours ($c_1 = 1568, c_2 = 1024$)	18.1M	13.1	68.02	88.07

pruning and structured sparsity approaches shown in Section 5.2. Table 5.5 shows the comparison with other network reduction approaches [13, 68, 20] for AlexNet. Note that the number of parameters and the compression ratio are for the overall network including the convolutional layers and the FCLs. Therefore, in our proposed approach, the numbers of parameters are increased compared to those shown in Table 5.1. From this table, the proposed model with $c_1 = 2304, c_2 = 1024$ achieved the highest test accuracy with high compression ratio among them.

Table 5.6 shows the comparison with other network reduction approaches [13, 67] for VGG-16. In the comparison with the network pruning approach [13], the compression ratio and the test accuracy of the proposed approach are lower than those of the network pruning approach. However, we note that the network pruning approach produces the irregular weight matrices which prevent the efficient parallel computation. On the other hand, the proposed approach can achieve higher test accuracy with higher compression ratio than the existing structured sparsity approach [67].

As a result, our proposed approach can compress the networks in almost the same performance compared to the network pruning approaches for AlexNet. Also, our approach achieved higher compression ratio and higher accuracy compared to the other structured sparsity approaches for AlexNet and VGG-16.

5.5.2 Computation time

In this subsection, we evaluate the forward and the backward propagation time of the proposed compression models using a single GPU. We have used the network parameters of AlexNet and VGG-16 on ILSVRC-2012 dataset. Therefore, the output of the last convolutional layer is given by $h = 6$, $w = 6$, $c = 256$ for AlexNet and $h = 7$, $w = 7$, $c = 512$ for VGG-16, and the last FCL is given by $c_3 = 1000$ in the both architectures. Note that the forward and the backward propagation time are measured for only FCLs to focus on our proposed approach.

First, as a preliminary experiment, we evaluate the computation efficiency of the proposed compression approach by comparing to the network pruning approach. The network pruning approach has advantages of the compression ratio and the high test accuracy, while it is not suitable to the parallel computation. On the other hand, our proposed compression approach has these advantages and is also suitable to the parallel computation. To compare these approaches, we show the computation time of the network pruning approach. However, since Han et al. [13] did not evaluate the computation time and the trained network parameters are not available, we can not measure the computation time with the same

Table 5.7: The comparison of the forward propagation time between the network pruning and the proposed compression approaches for the three FCLs when the batch size is 1.

approach	c_1	c_2	compression ratio (%)	forward propagation time [ms]			
				FCL1/CRLs	FCL2	FCL3	total
original	4096	4096	100.0	0.7170	0.1250	0.0336	0.8757
network pruning	4096	4096	26.7	0.6858	0.1127	0.0360	0.8346
			14.7	0.3867	0.0601	0.0209	0.4677
			8.6	0.1979	0.0319	0.0128	0.2426
			8.2	0.1839	0.0347	0.0138	0.2324
			4.7	0.1263	0.0281	0.0142	0.1687
			2.8	0.0758	0.0172	0.0101	0.1031
proposed compression	6272	4096	26.7	0.0303	0.1865	0.0341	0.2509
	6272	2048	14.7	0.0301	0.0964	0.0186	0.1452
	6272	1024	8.6	0.0186	0.0513	0.0186	0.0884
	3136	2048	8.2	0.0186	0.0513	0.0186	0.0884
	3136	1024	4.7	0.0185	0.0288	0.0112	0.0585
	1568	1024	2.8	0.0126	0.0174	0.0114	0.0414

pruned parameters as the paper. Therefore, we generated three FCL networks with from 26.7% to 2.7% parameters pruned randomly. In general, the pruned network weights are stored as a sparse data format to reduce the memory usage. As the sparse data format, we adopted the Compressed Sparse Row (CSR) format which is a standard data format for the sparse matrix. For the computation with the CSR format, we have used cuSPARSE 10.0 which is a library for handling sparse matrices on the GPU. Table 5.7 shows the forward propagation time of the network pruning and the proposed compression approaches for VGG-16 when the batch size is 1. From this table, we can see that the proposed compression approach runs faster than the network pruning approach. More specifically, when we compare the network these approaches in the same compression ratio, the speed-up factors of the proposed compression approach are in range from 2.5 to 3.3. Thus, our proposed sparse network architecture is suitable to the parallel computation.

Table 5.8: Computation time of the FCLs for the original, the simple compression and the proposed compression model for AlexNet

batch size	model	c_1	c_2	forward propagation time [ms]				speed-up	backward propagation time [ms]				speed-up
				FCL1/CRLs	FCL2	FCL3	total		FCL1/CRLs	FCL2	FCL3	total	
1	original	4096	4096	0.2807	0.1248	0.0342	0.4397	1.00	0.5878	0.2544	0.0703	0.9125	1.00
	simple compression	4608	4096	0.3222	0.1572	0.0337	0.5130	0.86	0.6608	0.2866	0.0707	1.0181	0.90
		4608	2048	0.3226	0.0806	0.0187	0.4219	1.04	0.6600	0.1504	0.0417	0.8521	1.07
		4608	1024	0.3226	0.0428	0.0112	0.3766	1.17	0.6597	0.0828	0.0268	0.7692	1.19
		2304	2048	0.1615	0.0394	0.0188	0.2197	2.00	0.3342	0.0835	0.0410	0.4587	1.99
		2304	1024	0.1614	0.0230	0.0112	0.1955	2.25	0.3342	0.0485	0.0269	0.4095	2.23
		1152	1024	0.0851	0.0145	0.0111	0.1108	3.97	0.1689	0.0326	0.0266	0.2281	4.00
	proposed compression	4608	4096	0.0165	0.1576	0.0338	0.2079	2.11	0.0285	0.2847	0.0704	0.3836	2.38
		4608	2048	0.0164	0.0806	0.0187	0.1157	3.80	0.0288	0.1497	0.0408	0.2193	4.16
		4608	1024	0.0163	0.0435	0.0112	0.0710	6.19	0.0285	0.0810	0.0261	0.1356	6.73
		2304	2048	0.0108	0.0392	0.0187	0.0687	6.40	0.0195	0.0815	0.0404	0.1404	6.45
		2304	1024	0.0109	0.0229	0.0113	0.0450	9.76	0.0194	0.0478	0.0255	0.0927	9.84
		1152	1024	0.0085	0.0141	0.0112	0.0339	12.99	0.0158	0.0320	0.0262	0.0740	12.33
128	original	4096	4096	0.9288	0.3195	0.0925	1.3408	1.00	1.5848	0.7378	0.2007	2.5233	1.00
	simple compression	4608	4096	1.0799	0.4799	0.0921	1.6520	0.81	1.7711	0.8881	0.2004	2.8596	0.88
		4608	2048	1.0913	0.2770	0.0540	1.4222	0.94	1.7776	0.4375	0.1120	2.3271	1.08
		4608	1024	1.0940	0.1341	0.0362	1.2644	1.06	1.7733	0.2453	0.0830	2.1016	1.20
		2304	2048	0.9061	0.1362	0.0518	1.0941	1.23	0.9041	0.2361	0.1100	1.2505	2.25
		2304	1024	0.9052	0.0718	0.0349	1.0119	1.33	0.8989	0.1380	0.0833	1.1202	2.25
		1152	1024	0.3058	0.0448	0.0346	0.3852	3.48	0.4584	0.0912	0.0837	0.6332	3.98
	proposed compression	4608	4096	0.0813	0.4789	0.0883	0.6485	2.07	0.1111	0.8554	0.1947	1.1612	2.17
		4608	2048	0.0828	0.2768	0.0512	0.4108	3.26	0.1096	0.4125	0.1064	0.6286	4.01
		4608	1024	0.0820	0.1318	0.0346	0.2485	5.40	0.1062	0.2210	0.0805	0.4077	6.16
		2304	2048	0.0745	0.1386	0.0514	0.2646	5.07	0.0769	0.2186	0.1028	0.3983	6.33
		2304	1024	0.0744	0.0737	0.0347	0.1829	7.33	0.0753	0.1223	0.0782	0.2757	9.15
		1152	1024	0.0732	0.0478	0.0346	0.1556	8.62	0.0695	0.0819	0.0763	0.2276	11.09

AlexNet: Here, we evaluate the computation time of the proposed approach in some settings. Table 5.8 shows the computation time of the FCLs for the original, the simple compression and the proposed compression models for AlexNet. The computation time of ReLU activation function is included in for each the column FCL1/CRLs and FCL2 regardless the batch size. Also, the computation time of the data format conversions are included in only the column FCL1/CRLs for the proposed compression models when the batch size is 128. According to the table, the implementation for the proposed compression models can run faster than that for the original and the simple compression models. The main reason is that the computational complexity of the proposed compression models is highly reduced since the proposed compression model can compress the network with high compression ratio. Also, the computation time of forward propagation is shorter than that of backward propagation because the computational complexity of forward propagation is small. However, when the batch size is 128, the forward propagation time of the proposed compression models with $c_1 = 2304$ and $c_1 = 1152$ have little difference compared to the backward propagation time. This is caused by the algorithm of cuBLAS. cuBLAS mainly uses a 128×64 matrix-matrix multiplication algorithm for the matrix-matrix multiplication in smaller matrix size than 128×64 such as 64×64. Therefore, when 64×64 matrix-matrix multiplication is performed, cuBLAS regards it as 128×64 matrix-matrix multiplication and computes them by padding zeros. In other words, the computation time of 64×64 matrix-matrix multiplication is the same as 128×64 matrix-matrix multiplication. When the batch size is 128, it is necessary to perform 64×128 and 32×128 matrix-matrix multiplications about the forward propagation for the proposed compression models when the channel $c_1 = 2304$ and 1152, respectively. Thus, the computation time of them is close to that of the proposed compression models with $c_1 = 4608$ which performs 128×128 matrix-matrix multiplications. Note that the difference of the computation time between the models with $c_1 = 4608$ and $c_1 = 2304$ is caused by including the ReLU and the GPU functions of the data format conversions.

Nevertheless, the implementation of our proposed approach is much faster. More specifically, the forward propagation time of the proposed compression models achieved speed-up factor up to 12.99 and 8.62 for the batch size 1 and 128, respectively. Also, the backward propagation time can run 12.33 and 11.09 times faster for the batch size 1 and 128, respectively.

VGG-16: Table 5.9 shows the computation time of the FCLs for the original, the sim-

Table 5.9: Computation time of the FCLs for the original, the simple compression and the proposed compression model for VGG-16

batch size	model	c_1	c_2	forward propagation time [ms]				speed-up	backward propagation time [ms]				speed-up
				FCL1/CRLs	FCL2	FCL3	total		FCL1/CRLs	FCL2	FCL3	total	
1	original	4096	4096	0.7170	0.1250	0.0336	0.8757	1.00	1.7385	0.2536	0.0708	2.0630	1.00
	simple compression	3136	2048	0.5514	0.0517	0.0188	0.6219	1.41	1.3215	0.1112	0.0413	1.4740	1.40
		3136	1024	0.5512	0.0292	0.0112	0.5917	1.48	1.3218	0.0625	0.0268	1.4111	1.46
		1568	1024	0.2799	0.0177	0.0112	0.3088	2.84	0.6335	0.0370	0.0267	0.6972	2.96
	proposed compression	6272	4096	0.0303	0.1865	0.0341	0.2509	3.49	0.0579	0.3997	0.0702	0.5278	3.91
		6272	2048	0.0301	0.0964	0.0186	0.1452	6.03	0.0581	0.2054	0.0409	0.3044	6.78
		6272	1024	0.0300	0.0516	0.0114	0.0930	9.41	0.0579	0.1070	0.0263	0.1912	10.79
		3136	2048	0.0186	0.0513	0.0186	0.0884	9.90	0.0347	0.1088	0.0415	0.1850	11.15
		3136	1024	0.0185	0.0288	0.0112	0.0585	14.97	0.0348	0.0625	0.0265	0.1238	16.67
		1568	1024	0.0126	0.0174	0.0114	0.0414	21.16	0.0233	0.0370	0.0261	0.0864	23.88
128	original	4096	4096	2.6624	0.3325	0.0947	3.0895	1.00	4.2631	0.7532	0.2054	5.2217	1.00
	simple compression	3136	2048	2.1978	0.1826	0.0547	2.4352	1.27	3.2770	0.3058	0.1143	3.6971	1.41
		3136	1024	2.2031	0.0937	0.0366	2.3334	1.32	3.2736	0.1952	0.0848	3.5535	1.47
		1568	1024	1.4055	0.0556	0.0356	1.4966	2.06	1.6663	0.1142	0.0839	1.8644	2.80
	proposed compression	6272	4096	0.1704	0.6331	0.0890	0.8926	3.46	0.2283	1.1090	0.2006	1.5379	3.40
		6272	2048	0.1741	0.3788	0.0517	0.6047	5.11	0.2252	0.5557	0.1092	0.8901	5.87
		6272	1024	0.1738	0.1775	0.0346	0.3859	8.01	0.2202	0.2800	0.0832	0.5834	8.95
		3136	2048	0.1513	0.1751	0.0513	0.3777	8.18	0.1511	0.2811	0.1053	0.5374	9.72
		3136	1024	0.1509	0.0925	0.0345	0.2779	11.12	0.1499	0.1720	0.0807	0.4026	12.97
		1568	1024	0.1449	0.0552	0.0344	0.2342	13.19	0.1366	0.0973	0.0772	0.3110	16.79

ple compression and the proposed compression models for VGG-16. We can see that the tendency of the computation time for VGG-16 is similar to that for AlexNet. Also, when the batch size is 128, the forward propagation time of the proposed compression models with $c_1 = 3136$ and $c_1 = 1568$ is close to the backward propagation time. This is caused by the same reason of the case for AlexNet, that is, cuBLAS uses the 128×64 matrix-matrix multiplication for the matrix sizes 64×128 and 32×128 in the forward propagation with $c_1 = 3136$ and $c_1 = 1568$, respectively.

As a result, the forward propagation time of the proposed compression models achieved speed-up factor up to 21.16 and 13.19 for the batch size 1 and 128, respectively. Also, the backward propagation time can run 23.88 and 16.79 times faster for the batch size 1 and 128, respectively. These results for AlexNet and VGG-16 imply that the proposed approach can be utilized to accelerate the computation as well as compress the network models.

Table 5.10: The performance of the proposed approach on the overall architecture including the convolutional layers and the FCLs when the batch size is 1.

CNN	model	c_1	c_2	number of parameters	compression ratio (%)	forward		backward	
						time (ms)	speed-up	time (ms)	speed-up
AlexNet	original	4096	4096	62.4M	100.0	0.9910	1.00	1.9375	1.00
	proposed compression	4608	4096	27.9M	44.8	0.7593	1.31	1.4086	1.38
		4608	2048	16.4M	26.4	0.6671	1.49	1.2443	1.56
		4608	1024	10.6M	17.1	0.6224	1.59	1.1606	1.67
		2304	2048	11.1M	17.9	0.6200	1.60	1.1664	1.66
		2304	1024	7.7M	12.4	0.5964	1.66	1.1177	1.73
		1152	1024	6.2M	10.0	0.5852	1.69	1.0990	1.76
VGG-16	original	4096	4096	138.3M	100.0	4.4919	1.00	11.2722	1.00
	proposed compression	6272	4096	47.7M	34.5	3.8671	1.16	9.7370	1.16
		6272	2048	32.8M	23.7	3.7614	1.19	9.5136	1.18
		6272	1024	25.4M	18.3	3.7093	1.21	9.4004	1.20
		3136	2048	24.8M	17.9	3.7046	1.21	9.3942	1.20
		3136	1024	20.5M	14.9	3.6747	1.22	9.3330	1.21
		1568	1024	18.1M	13.1	3.6576	1.23	9.2956	1.21

5.5.3 Performance of the proposed approach on the overall network

In the above subsections, we evaluated the performance of only the FCLs. Here, we evaluate the performance of the proposed approach on the overall network including the convolutional layers and the FCLs. Table 5.10 shows the number of parameters, the compression ratio, the computation time and the speed-up on the overall network when the batch size is 1. The compression ratio is still high even if the number of parameters in the convolutional layers is included. The speed-up is in range from 1.31 to 1.76 for AlexNet, and from 1.16 to 1.23 for VGG-16. The speed-up factors of the proposed compression models are lower than those in Tables 5.8 and 5.9. This is because the computational cost of the convolutional layer is more expensive than that of the FCLs. However, in this work, our proposed approach does not change the architecture of the convolutional layers. Therefore, by combining the proposed approach and the other sparse approaches [16, 17, 18] for the convolutional layers, we can shorten the computation time more.

5.5.4 Transfer learning using VGG-16 on various datasets

In this part, to confirm reliability of the proposed approach, we tackle another experiment using transfer learning for VGG-16. Considering the practical application, we have trained the models when all convolutional layers are initialized by pre-trained model of VGG-16 taken from the Caffe model zoo[1]. This pre-trained model had been trained on ILSVRC-2012 dataset [23]. Therefore, the source domain of transfer learning is various common domain such as bird, fish, machine and construction. As datasets for transfer learning, we have used three datasets which have different target domains, CUBS Birds [78], Stanford Cars [79] and Stanford Dogs [80]. CUBS Birds dataset is the image dataset of birds. This dataset contains 5994 training images and 5794 evaluation images classified into 200 classes of birds. Stanford Cars dataset is the image dataset of cars. It contains 8144 training images and 8041 evaluation images classified into 196 classes of cars. Stanford Dogs dataset is the image dataset of dogs. It contains 12000 training images and 8580 evaluation images classified into 120 classes of dogs. In the network training, we have mainly adopted the training method by Krizhervsky et al [21]. The different points are shown as follows. The evaluation images have been resized from 256×256 to 224×224 by cropping the outside. The weights of all FCLs and the CRLs have been initialized with a normal distribution with the zero mean and 0.01 variance. Also, the initial values of biases have been set zero. As the optimization settings, the batch size has been set to 32, and the learning rate has been set to 0.0005 and then decayed to $\frac{1}{10}$ when the test top-1 accuracy is not improved across 5 epochs. We have trained every model for 60 epochs. Top-1 test accuracy is

[1]https://github.com/BVLC/caffe/wiki/Model-Zoo

Table 5.11: The result of transfer learning for VGG-16 on CUBS Birds dataset and inference time of the FCLs

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	inference time total time [ms]	inference time speed-up
original	4096	4096	123.4M	100.0	65.49	0.8552	1.00
simple compression	784	512	20.2M	16.8	67.87	0.1644	5.20
	784	256	20.0M	16.6	68.37	0.1626	5.26
	392	256	10.0M	8.3	67.87	0.1003	8.53
	256	256	6.5M	5.4	67.47	0.0691	12.37
	128	128	3.3M	2.7	65.59	0.0499	17.12
	64	64	1.6M	1.3	53.28	0.0379	22.57
proposed compression	1568	1024	2.6M	2.2	66.97	0.0450	18.99
	1568	512	1.7M	1.4	67.90	0.0381	22.43
	1568	256	1.3M	1.0	66.75	0.0298	28.67
	784	512	0.9M	0.8	65.90	0.0298	28.72
	784	256	0.7M	0.5	65.35	0.0279	30.69
	392	256	0.4M	0.3	62.31	0.0274	31.26

measured for the evaluation images of each dataset. The training was repeated for 5 times and the top-1 test accuracy we show are the maximum values among these training results.

CUBS Birds: Table 5.11 shows the training result of the original, the simple compression and the proposed compression models on CUBS Birds dataset and the inference time when the batch size is 1. According to the table, the simple compression model when the channels $c_1 = 784$ and $c_2 = 256$ achieved the highest test accuracy with slightly high compression ratio. On the other hand, the proposed compression models achieved high compression ratio with high accuracy. We can see that the proposed compression model achieves high test accuracy compared to the simple compression model with almost the same compression ratio. For example, the test accuracy of the proposed model with 1.4% compression ratio is 14.62% higher than that of the simple compression model with 1.3% compression ratio. Also, the proposed compression models can perform inference faster owing to the significant reduction of the number of the network parameters.

Table 5.12: The result of transfer learning for VGG-16 on Stanford Cars dataset and inference time of the FCLs

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	inference time	
						total time [ms]	speed-up
original	4096	4096	123.4M	100.0	79.91	0.8541	1.00
simple compression	784	512	20.2M	16.8	83.36	0.1661	5.14
	784	256	20.0M	16.6	83.41	0.1624	5.26
	392	256	10.0M	8.3	83.08	0.0997	8.56
	256	256	6.5M	5.4	82.56	0.0639	12.32
	128	128	3.3M	2.7	79.75	0.0502	17.01
	64	64	1.6M	1.3	62.43	0.0376	22.70
proposed compression	1568	1024	2.6M	2.2	82.56	0.0433	19.74
	1568	512	1.7M	1.4	82.14	0.0385	22.21
	1568	256	1.3M	1.0	81.68	0.0295	28.94
	784	512	0.9M	0.8	80.87	0.0293	29.16
	784	256	0.7M	0.5	79.99	0.0279	30.65
	392	256	0.4M	0.3	76.75	0.0270	31.69

Stanford Cars: Table 5.12 shows the training result on Stanford Cars dataset and the inference time when the batch size is 1. We can see that the training result on Stanford Cars dataset has the same tendency to that on CUBS Birds dataset. More specifically, the difference of the top-1 accuracy between the simple compression model with 1.3% compression ratio and the proposed compression model with 1.4% compression ratio is 19.71%. Therefore, the proposed compression models can achieve high test accuracy with high compression ratios.

Stanford Dogs: Table 5.13 shows the training result on Stanford Dogs dataset and the inference time when the batch size is 1. According to the table, the simple compression model with $c_1 = 1568$ and $c_2 = 512$ achieved the highest top-1 accuracy. However, when comparing to that model, the proposed compression model with $c_1 = 3136$ and $c_2 = 512$ achieves approximately 12 times reduction of the number of parameters with only 0.2% top-1 accuracy decrease. In addition, when comparing two types of the compression models in almost the same compression ratio, the proposed compression models can get higher

Table 5.13: The result of transfer learning for VGG-16 on Stanford Dogs dataset and inference time of the FCLs

model	c_1	c_2	number of parameters	compression ratio (%)	top-1 accuracy (%)	inference time	
						total time [ms]	speed-up
original	4096	4096	120.0M	100.0	73.08	0.8522	1.00
simple compression	3136	1024	82.0M	68.3	73.72	0.5884	1.45
	3136	512	80.3M	66.9	73.74	0.5771	1.48
	1568	512	40.2M	33.1	73.95	0.2975	2.86
	1568	256	39.8M	33.0	73.36	0.2953	2.89
	198	128	5.0M	4.1	72.39	0.0646	13.18
	128	128	3.2M	2.7	71.84	0.0500	17.03
	64	64	1.6M	1.3	65.78	0.0378	22.52
proposed compression	3136	1024	4.9M	4.1	73.66	0.0615	13.86
	3136	512	3.3M	2.7	73.75	0.0488	17.46
	1568	512	1.7M	1.4	73.09	0.0377	22.61
	1568	256	1.2M	1.0	72.45	0.0294	28.94

accuracy than that for the simple compression models. Also, regarding the inference time, the proposed models can run faster than the simple compression models.

These results of transfer learning show our proposed sparse architecture can handle three target domains with high test accuracy and high compression ratio. As a result, the reliability of the proposed approach was proved in this subsection. However, it may be enough to use the simple compression approach for these classification problems since it can achieve the highest accuracy on each dataset. If high compression ratio with high accuracy is desired, the proposed compression method should be implemented to achieve them.

5.6 Conclusion

In this work, we have proposed a structured sparse fully-connected layer in CNNs. The proposed approach focuses on the first FCL and eliminates the connections between distant elements in the feature maps. Using the approach, we succeeded in reducing the number of parameters in the FCL with almost no loss of accuracy. In addition, we have shown implementations using cuBLAS for the proposed models. As a result, the proposed model achieves a 14.7 times compression with 0.68% top-1 accuracy and 0.19% top-5 accuracy decreases for AlexNet, and a 21.3 times compression with 0.68% top-1 accuracy and 0.31% top-5 accuracy decreases for VGG-16 on ILSVRC-2012 dataset. The implementation for the proposed FCLs achieved speed-up factors 12.33 and 11.09 compared to that for the general FCLs on AlexNet architecture when the batch size is 128. Also, the implementation for the proposed FCLs can run 11.12 and 12.97 times faster for forward and backward propagation than that for the general FCLs on VGG-16 architecture when the batch size is 128. We believe that the proposed approach can compress the network with no accuracy drop for CNNs which have three FCLs.

Chapter 6

Conclusion of the dissertation

In this dissertation, we have presented the two tile art image generation methods, and the structured sparse FCL in the CNNs and its efficient GPU implementation.

In Chapter 4, we have presented the two tile art image generation methods; the greedy approach and the machine learning approach. The experiment results show that the greedy approach on the GPU can run up to 318 times faster than that on CPU with single thread and the machine learning approach can generate a tile art image of size of size 4096×3072 within 1.04 seconds while the greedy approach on the GPU takes 571 seconds.

In Chapter 5, we have presented the structured sparse FCL in the CNNs that is suitable to parallel computation. As a result for the large scale image recognition dataset, the proposed approach achieves a 14.7 times compression with 0.68% top-1 accuracy and 0.19% top-5 accuracy decrease for AlexNet, and a 21.3 times compression with 0.68% top-1 accuracy and 0.31% top-5 accuracy decrease for VGG-16. Also, in the experiment on NVIDIA RTX 2080 Ti GPU, the GPU implementation for the proposed FCLs achieves speed-up factor 14.97 and 16.67 for forward and backward propagation compared to that for the non-compressed FCLs, respectively.

Bibliography

[1] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. CNN features off-the-shelf: an astounding baseline for recognition. pages 806–813. Proceedings of the IEEE conference on computer vision and pattern recognition workshops, 2014.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, et al. TensorFlow: Large-scale machine learning on heterogeneous systems. `https://www.tensorflow.org/`, 2015.

[3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[4] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv:1512.01274*, 2015.

[5] NVIDIA Corporation. CUDA C programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2020.

[6] NVIDIA Corporation. cuBLAS. `https://docs.nvidia.com/cuda/cublas/`, 2020.

[7] NVIDIA Corporation. cuSPARSE. `https://docs.nvidia.com/cuda/cusparse/`, 2020.

[8] Sarah Kelly. *The Complete Mosaic Handbook: Projects, Techniques, Designs*. Firefly Books, first edition, 2004.

[9] S. Battiato, G. Di Blasi, G. M. Farinella, and G. Gallo. Digital mosaic frameworks - an overview. *Computer Graphics Forum*, 26(4):794–812, 2007.

[10] M. Analoui and J.P. Allebach. Model-based halftoning by direct binary search. In *Proc. of SPIE/IS&T Symposium on Electronic Imaging Science and Technology*, pages 96–108, San Jose, CA, USA, 1992.

[11] Phillip Isola, Jun-Yan Zhu, and Tinghui Zhou. Image-to-image translation with conditional adversarial networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5967–5976, 2017.

[12] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[13] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. pages 1135–1143. Proc. of the 28th International Conference on Neural Information Processing Systems - Volume 1, 2015.

[14] Song Han, Jeff Pool, Sharan Narang, Huizi Mao, Enhao Gong, Shijian Tang, Erich Elsen, Peter Vajda, Manohar Paluri, John Tran, Bryan Catanzaro, and William J. Dally. DSD: Dense-Sparse-Dense training for deep neural networks. *arXiv:1607.04381*, 2016.

[15] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. EIE: Efficient inference engine on compressed deep neural network. pages 243–254. 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016.

[16] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. Pruning filters for efficient ConvNets. *arXiv:1608.08710*, 2016.

[17] V. Lebedev and V. Lempitsky. Fast ConvNets using group-wise brain damage. pages 2554–2564. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[18] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. ThiNet: A filter level pruning method for deep neural network compression. pages 5068–5076. Proc. of IEEE International Conference on Computer Vision (ICCV), 2017.

[19] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. pages 2074–2082. Advances in neural information processing systems, 2016.

[20] C. Deng, S. Liao, Y. Xie, K. K. Parhi, X. Qian, and B. Yuan. PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices. pages 189–202. 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2018.

[21] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. pages 1097–1105. Advances in Neural Information Processing Systems 25, 2012.

[22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[23] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. pages 248–255, Miami, FL, 2009. 2009 IEEE Conference on Computer Vision and Pattern Recognition.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–9. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[26] Baohua Sun, Lin Yang, Patrick Dong, Wenhan Zhang, Jason Dong, and Charles Young. Ultra power-efficient CNN domain specific accelerator with 9.3tops/watt for mobile and embedded applications. pages 1677–1685. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), 2018.

[27] Sangyoon Oh, Minsub Kim, Donghoon Kim, Minjoong Jeong, and Minsu Lee. Investigation on performance and energy efficiency of CNN-based object detection on embedded device. 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT), 2017.

[28] Kuldeep Kurte, Jibonananda Sanyal, Anne Berres, Dalton Lunga, Mark Coletti, Hsiuhan Lexie Yang, Daniel Graves, Benjamin Liebersohn, and Amy Rose. Performance analysis and optimization for scalable deployment of deep learning models for country-scale settlement mapping on titan supercomputer. *Concurrency and Computation: Practice and Experience*, 31(20), 2019.

[29] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A machine-learning supercomputer. pages 609–622. 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, 2014.

[30] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems 27*, pages 2672–2680, 2014.

[31] Dominik Scherer, Andreas M 端 ller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. pages 92–101, 01 2010.

[32] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. page 3. in ICML Workshop on Deep Learning for Audio, Speech and Language Processing, 2013.

[33] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[34] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT press, 2016.

[35] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv:1706.02677*, 2017.

[36] Duhu Man, Kenji Uda, Yasuaki Ito, and Koji Nakano. A GPU implementation of computing Euclidean distance map with efficient memory access. In *Proc. of International Conference on Networking and Computing*, pages 68–76, Dec. 2011.

[37] NVIDIA CUDA C programming guide version 10.1, 2019.

[38] GIMP - GNU image manipulation program.

[39] Hiroaki Kouge, Takumi Honda, Toru Fujita, Yasuaki Ito, Koji Nakano, and Jacir L. Bordim. Accelerating digital halftoning using the local exhaustive search on the GPU. *Concurrency and Computation: Practice and Experience*, 29(2):e3781, 2017.

[40] Yasuaki Ito and Koji Nakano. A new FM screening method to generate cluster-dot binary images using the local exhaustive search with FPGA acceleration. *International Journal on Foundations of Computer Science*, 19(6):1373–1386, 2008.

[41] Yuji Takeuchi, Koji Nakano, Daisuke Takafuji, and Yasuaki Ito. A character art generator using the local exhaustive search, with GPU acceleration. *International Journal of Parallel Emergent and Distributed Systems*, 31(1):47–63, 2016.

[42] Dongyeon Kim, Minjung Son, Yunjin Lee, Henry Kang, and Seungyong Lee. Feature-guided image stippling. *Computer Graphics Forum*, 27(4):1209–1216, 2008.

[43] Thomas Houit and Frank Nielsen. Video stippling. In *Advanced Concepts for Intelligent Vision Systems: 13th International Conference*, pages 384–395, 2011.

[44] Dongxiang Chi. A natural image pointillism with controlled ellipse dots. *Advances in Multimedia*, 2014:1–16, 2014. art. ID 567846.

[45] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *Proc. of European Conference on Computer Vision*, pages 694–711, 2016.

[46] Emily L Denton, Soumith Chintala, arthur szlam, and Rob Fergus. Deep generative image models using a Laplacian pyramid of adversarial networks. In *Advances*

in Neural Information Processing Systems 28, pages 1486–1494. Curran Associates, Inc., 2015.

[47] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. In *Proc. of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[48] Xintao Wang, Ke Yu, Shixiang Wu, Jinjin Gu, Yihao Liu, Chao Dong, Yu Qiao, and Chen Change Loy. ESRGAN: Enhanced super-resolution generative adversarial networks. In *Proc. of the European Conference on Computer Vision (ECCV)*, pages 63–79, 2018.

[49] Richard Zhang, Phillip Isola, and Alexei A. Efros. Colorful image colorization. *CoRR*, abs/1603.08511, 2016.

[50] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Learning representations for automatic colorization. In *Proc. of European Conference on Computer Vision*, pages 577–593, 2016.

[51] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Let there be color!: joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2016)*, 35(4):110, 2016.

[52] Jia-Bin Huang, Sing Bing Kang, Narendra Ahuja, and Johannes Kopf. Image completion using planar structure guidance. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2014)*, 33(4):129, 2014.

[53] Deepak Pathak, Philipp Krähenbühl, Jeff Donahue, Trevor Darrell, and Alexei Efros. Context encoders: Feature learning by inpainting. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2536–2544, 2016.

[54] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Globally and locally consistent image completion. *ACM Transactions on Graphics (Proc. of SIGGRAPH 2017)*, 36(4):107, 2017.

[55] Guilin Liu, Fitsum A Reda, Kevin J Shih, Ting-Chun Wang, Andrew Tao, and Bryan Catanzaro. Image inpainting for irregular holes using partial convolutions. In *Proc. of the European Conference on Computer Vision (ECCV)*, pages 85–100, 2018.

[56] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proc. of the IEEE international conference on computer vision*, pages 2223–2232, 2017.

[57] Matteo Tomei, Marcella Cornia, Lorenzo Baraldi, and Rita Cucchiara. Art2real: Unfolding the reality of artworks via semantically-aware image-to-image translation. In *Proc. of IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019.

[58] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice-Hall, Inc., 3rd edition, 2006.

[59] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[60] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, 2015.

[61] Chuan Li and Michael Wand. Precomputed real-time texture synthesis with Markovian generative adversarial networks. In *Computer Vision – ECCV 2016*, pages 702–716, 2016.

[62] Diederik P. Kingma and Jimmy Lei Ba. ADAM: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2017.

[63] Lai-Man Po. Lenna 97: A complete story of Lenna. http://www.ee.cityu.edu.hk/˜lmpo/lenna/Lenna97.html, 2001.

[64] Gregory Griffin, Alex Holub, and Pietro Perona. Caltech-256 object category dataset. Technical report, California Institute of Technology, 2007.

[65] OpenMP. `http://www.openmp.org/`.

[66] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. Exploring the granularity of sparsity in convolutional neural networks. pages 1927–1934. Computer Vision and Pattern Recognition Workshops, 2017.

[67] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. Tensorizing neural networks. pages 442–450. Proceedings of the 28th International Conference on Neural Information Processing Systems, 2015.

[68] M. Hosseini, M. Horton, H. Paneliya, U. Kallakuri, H. Homayoun, and T. Mohsenin. On the complexity reduction of dense layers from $o(n^2)$ to $o(n \log n)$ with cyclic sparsely connected layers. pages 1–6. Proceedings of the 56th Annual Design Automation Conference, 2019.

[69] Andrew Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[70] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted residuals and linear bottlenecks. pages 4510–4520. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[71] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc Le, and Hartwig Adam. Searching for MobileNetV3. pages 1314–1324. Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2019.

[72] Min Lin amd Qiang Chen amd Shuicheng Yan. Network in network. *arXiv:1312.4400*, 2014.

[73] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-Excitation networks. pages 7132–7141. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[74] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. BinaryConnect: Training deep neural networks with binary weights during propagations. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 3123–3131. Curran Associates, Inc., 2015.

[75] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to $+1$ or -1. *arXiv:1602.02830*, 2016.

[76] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, 2018.

[77] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc., 2014.

[78] P. Welinder, S. Branson, T. Mita, C. Wah, F. Schroff, S. Belongie, and P. Perona. Caltech-UCSD Birds 200. Technical Report CNS-TR-2010-001, California Institute of Technology, California, USA, 2010.

[79] Jonathan Krause, Michael Stark, Jia Deng, and Li Fei-Fei. 3D object representations for fine-grained categorization. pages 554–561. 4th International IEEE Workshop on 3D Representation and Recognition (3dRR-13), 2013.

[80] Aditya Khosla, Nityananda Jayadevaprakash, Bangpeng Yao, and Li Fei-Fei. Novel dataset for fine-grained image categorization. First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition, 2011.

Acknowledgement

First and foremost, I would like to show my deepest gratitude to my supervisor, Professor Koji Nakano for his continuous encouragement, advice and support. He is a respectable and resourceful scholar. His knowledge and research experience are in value through the whole period of my study. As a supervisor, he taught me skills and practices that will benefit my future research career.

I shall express sincere appreciation to my thesis committee members, Professor Hiroaki Mukaidani and Associate Professor Yasuaki Ito for reviewing my dissertation.

I would also like to express my thanks to Assistant Professor Daisuke Takafuji and Assistant Professor Ryota Yasudo for their support and continuous guidance in every stage of my study. My heartiest thanks go to all members of computer system laboratory. They were always kind and very keen to help.

I would express thanks to all the faculty members of the Department of Information Engineering of Hiroshima University.

Last but not least, I wish to express my thanks to my parents who have always supported me.

List of publications

Journals

J-1: Naoki Matsumura, Hiroki Tokura, Yuki Kuroda, Yasuaki Ito, Koji Nakano, Tile Art Image Generation using Parallel Greedy Algorithm on the GPU and its Approximation with Machine Learning, Concurrency and Computation: Practice and Experience, Vol.33, No.12, e5623, June 2021.

- Chapter 4: Tile art image generation on the GPU and its approximation with machine learning

J-2: Naoki Matsumura, Yasuaki Ito, Koji Nakano, Akihiko Kasagi, Tsuguchika Tabaru, A Novel Structured Sparse Fully-Connected Layer in Convolutional Neural Networks, Concurrency and Computation: Practice and Experience, to appear.

- Chapter 5: A novel structured sparse fully-connected layer in CNNs

International Conferences

C-1: Naoki Matsumura, Hiroki Tokura, Yuki Kuroda, Yasuaki Ito, Koji Nakano, Tile Art Image Generation Using Conditional Generative Adversarial Networks, in Proc. of International Symposium on Computing and Networking Workshops, pp. 209-215, Takayama, Japan, November 2018.

- Chapter 4: Tile art image generation on the GPU and its approximation with machine learning

C-2: Naoki Matsumura, Yasuaki Ito, Koji Nakano, Akihiko Kasagi and Tsuguchika Tabaru, Structured Sparse Fully-Connected Layers in the CNNs and its GPU Acceleration, in Proc. of International Symposium on Computing and Networking Workshops, pp. 148–154, Nagasaki, Japan, November 2019.

- Chapter 5: A novel structured sparse fully-connected layer in CNNs

Others

J-3: Takuma Wada, Naoki Matsumura, Ryota Yasudo, Koji Nakano, Yasuaki Ito, Efficient implementations of Bloom filter using block RAMs and DSP slices on the FPGA, Concurrency and Computation: Practice and Experience, Vol.33, No.12, e5475, June 2021.

C-3: Takuma Wada, Naoki Matsumura, Koji Nakano, Yasuaki Ito, Efficient Byte Stream Pattern Test using Bloom Filter with Rolling Hash Functions on the FPGA, in Proc. of International Symposium on Computing and Networking, pp. 66–75, Takayama, Japan, November 2018.