A High-Performance Accelerator for Super-Resolution Processing on Embedded GPU

Wenqian Zhao[1], Qi Sun[1], Yang Bai[1], Wenbo Li[1], Haisheng Zheng[2], Bei Yu[1], Martin D.F. Wong[1]
[1]The Chinese University of Hong Kong [2]SmartMore
{wqzhao,qsun,ybai,wbli,byu,mdfwong}@cse.cuhk.edu.hk, leo.zheng@smartmore.com

***Abstract*—Recent years have witnessed impressive progress in super-resolution (SR) processing. However, its real-time inference requirement sets a challenge not only for the model design but also for the on-chip implementation. In this paper, we implement a full-stack SR acceleration framework on embedded GPU devices. The special dictionary learning algorithm used in SR models was analyzed in detail and accelerated via a novel dictionary selective strategy. Besides, the hardware programming architecture together with the model structure is analyzed to guide the optimal design of computation kernels to minimize the inference latency under the resource constraints. With these novel techniques, the communication and computation bottlenecks in the deep dictionary learning-based SR models are tackled perfectly. The experiments on the edge embedded NVIDIA NX and 2080Ti show that our method outperforms the state-of-the-art NVIDIA TensorRT significantly and can achieve real-time performance.**

I. Introduction

Super-resolution (SR), which refers to the process of recovering or generating high-resolution (HR) video frames from low-resolution (LR) frames, is an important class of graphical processing techniques in computer vision. Among the existing methods, the simplest one is to adopt basic spatially invariant nearest-neighbor, bilinear, and bicubic interpolation. In the past decade, with the fast developments of deep learning algorithms, a large variety of deep learning models has also been adopted to tackle the SR tasks, ranging from general convolution neural networks [1] to generative adversarial networks [2], [3]. Recently, by introducing dictionary learning methods with pixel-level local feature fusion operations [4], [5], the qualities of the generated high-resolution images or videos are further improved, which recovered richer details. Meanwhile, efficient deployments of these deep learning-based SR models have attracted more and more attention gradually.

Lots of previous arts have been proposed to deploy different deep learning algorithms on a variety of hardware platforms [6]–[9]. The deployed models are mostly for object classification [10], detection [11], [12], neural language processing [13], and *etc*. Although a wide range of application scenarios are covered, the deep learning operators implemented by them are so similar to each other that no explicit technique gaps distinguish them. Typical operators include direct convolution, fully connected operation, pooling, softmax, and *etc*. Due to the regularity of these operators, the commercial tools achieve state-of-the-art performance on these operators by using dedicatedly-optimized hardware codes. For example,

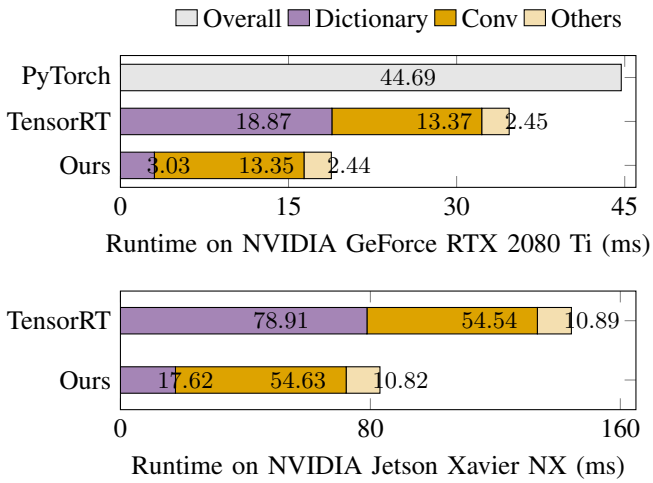

Fig. 1 Runtime profiling results of deploying a state-of-the-art super-resolution model with PyTorch, TensorRT, and our accelerator. PyTorch is installed and evaluated only on 2080Ti. Time cost breaks into 3 separate components: (1) dictionary query and filtering step (2) convolution operation; (3) data reformatting, concatenation or other operations.

TensorRT [14] outperforms other tools on NVIDIA GPUs and Intel MKL-DNN [15] has the dominating performance on Intel CPUs.

Despite the achievements of these traditional model deployments, the complexities and particularities of SR algorithms hinder the models from following the optimization strategies of the traditional DNN models to realize real-time performance (*i.e.*, more than 25 frames per second), under demanding edge devices. Firstly, the algorithmic processing logics of the traditional models and SR models are completely opposite. The mainstream DNN models down-sample the inputs to learn the embedded features, *e.g.*, VGG, GoogleNet, MobileNet, ResNet, Faster R-CNN, and *etc*. The down-sampling characteristic eases the communication and computation pressures on features and weights. In contrast, SR models enlarge (up-sample) the inputs continuously to recover more details. Much fewer weights are shared by much more features. Therefore, the features instead of weights become a crucial influence factor thus making the existing memory optimization techniques powerless. Similar phenomenons have been discovered in [16]. Secondly, the newly used SR operations, *e.g.*, local pixel-shuffle and dictionary learning,

exacerbate these challenges. The traditionally widely-used operations, *e.g.*, direct convolutions and pooling, have been solved through various techniques. Typical techniques include loop unrolling, tiling, systolic array, and so forth [7], [17], [18]. In comparison, as shown in Fig. 1, the novel operations in SR are time-consuming and require special computation re-organizations and parallelisms. Due to these challenges, the existing solutions are unsatisfactory, even the state-of-the-art commercial tool, *e.g.*, TensorRT.

In this paper, several novel techniques are proposed to handle these challenges. Firstly, we propose a fast model sliming strategy for model sliming to handle the large models and dense parameters of SR models. Structured pruning was utilized to select and compress the SR dictionaries. Only the most important dictionaries are reserved so that the serial computation iterations can be accelerated remarkably without degradations of result qualities. Secondly, to obtain the optimal hardware implementation given the SR models, a novel constrained-based design searching algorithm is proposed. The GPU architecture is analyzed in detail and the resources and computational workloads are considered as the constraints to restrict the candidate of feasible hardware implementations. The illegal and non-optimal designs are discarded and a Bayesian optimization-based searching algorithm is proposed to find the optimal design efficiently. As a result, the communication latencies are hidden and the bandwidth usage is improved. Last but not least, the original large task is re-organized to be smaller sub-tasks and then these sub-tasks are run in parallel. Based on these efforts, the overall system parallelism and resource utilization are maximized aggressively to ameliorate the computation- and communication-bounded issues. The main contributions of this paper are listed as follows:

- Dictionary learning algorithms on extremely large data frames are accelerated by a lot for the first time via our specifically designed acceleration engine.
- Model Slimming for SR dictionaries and parallel execution techniques are proposed which can greatly relieve the stress resulting from the large data frames. Both the computation and communication workloads are reduced.
- Resources- and workloads-aware constraints dedicated for GPUs are proposed for the first time to guide the searching of optimal hardware implementations. The optimal design can be achieved in a short time.
- Compared with the state-of-the-art tool TensorRT, on edge embedded GPU NVIDIA Jetson Xavier NX and server-level 2080Ti, our method achieves faster and real-time SR processing. Runtime profiling results are shown in Fig. 1.

The remainder of the paper is organized as the following. Section II recaps the deep super-resolution models, dictionary learning, and the background of GPU programming. Section III illustrates our acceleration methods in detail. Section IV demonstrates the experiments and results. Finally, we conclude this paper in Section V.

II. Preliminaries

A. Super-Resolution Algorithms and Dictionary Learning

Super-resolution algorithms aim at reconstructing a high-resolution (HR) image from a low-resolution (LR) one. Due to its wide applications, lots of efforts have been made in the past few decades. Denote the height and width of the image as H and W respectively, and the channel number of the image as a squared value s^2. For a given high-resolution vectorized image $\boldsymbol{y} \in \mathbb{R}^{HWs^2}$, its low-resolution counterpart $\boldsymbol{x} \in \mathbb{R}^{HW}$ can be obtained via down-sampling and blurring, as shown in Equation (1).

$$\boldsymbol{x} = \boldsymbol{SHy}, \tag{1}$$

where $\boldsymbol{H} \in \mathbb{R}^{HWs^2}$ represents the blurring operation and $\boldsymbol{S} \in \mathbb{R}^{HW \times HWs^2}$ is the down-sampling operation. Correspondingly, the SR processing can be regarded as the reverse process of Equation (1), *i.e.*, recovering $\boldsymbol{y}$ from $\boldsymbol{x}$ by up-sampling and deblurring. However, solving Equation (1) is a notoriously challenging ill-posed problem because a specific $\boldsymbol{x}$ corresponds to a crop of possible $\boldsymbol{y}$. Besides, in most instances, the HR space that we intend to map the LR input to is usually intractable.

To tackle these challenges, some basic linear interpolation methods are adopted, *e.g.*, bilinear, and bicubic interpolations. In these methods, the strategies of mapping from the LR space to the HR space are quite straightforward and simple while neglecting some content varieties and local structures. Further, to constrain the mapping, some dictionary learning algorithms are proposed, which explicitly specify the mapping relationships between the LR space and HR space. Some pairs of dictionaries which map low-resolution (LR) patches to high-resolution (HR) patches are learned and used in inference. HR patches can be regarded as the spatial combination of the LR patches and now the problem is to learn the combination coefficients. Recently, with the fast developments of deep learning algorithms, some advanced methods have been proposed to learn better dictionaries and combination coefficients and achieved optimal performance [5], [19], [20].

Focusing on optimizing the deep dictionary learning-based SR algorithms, the basic processing flow is explained as follows. Firstly, the vectorized LR input $\boldsymbol{x} \in \mathbb{R}^{HW}$ is up-sampled to a matrix $\boldsymbol{B} \in \mathbb{R}^{HWs^2 \times k^2}$ containing HWs^2 upsampled LR patches with size k^2. Secondly, some transformation operations are conducted to transform the LR batches to HR batches. The i-th pixel $\boldsymbol{y}_i$ in the HR image vector $\boldsymbol{y} \in \mathbb{R}^{HWs^2}$ is obtained via integrating the neighboring pixels of batch $\boldsymbol{B}_i$ (*i.e.*, the i-th row of $\boldsymbol{B}$) centered at the coordinate of $\boldsymbol{y}_i$. This pixel-level operation can be formulated as Equation (2).

$$\boldsymbol{y}_i = \boldsymbol{F}_i \boldsymbol{B}_i^{\top}, \text{ with } \boldsymbol{F}_i = \boldsymbol{\Phi}_i \boldsymbol{D}, \tag{2}$$

where $\boldsymbol{F}_i \in \mathbb{R}^{1 \times k^2}$ is the integration coefficient vector (*a.k.a.* a filter). $\boldsymbol{F}_i$ can be further represented as the linear combination of a dictionary $\boldsymbol{D} \in \mathbb{R}^{L \times k^2}$ with combination coefficient

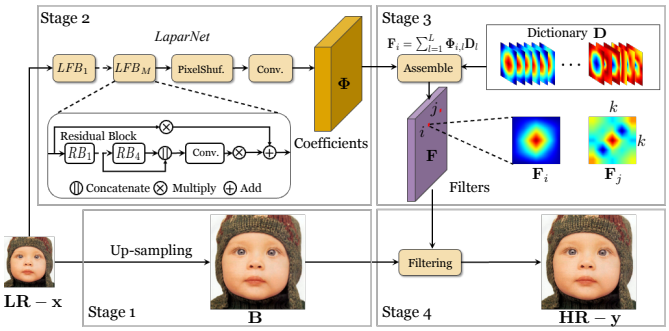

Fig. 2 The architecture of linearly-assembled pixel-adaptive regression network (LAPAR) [5] with four basic stages, *i.e.*, stage 1: up-sampling; stage 2: *LaparNet*; stage 3: dictionary assembling; stage 4: filtering.

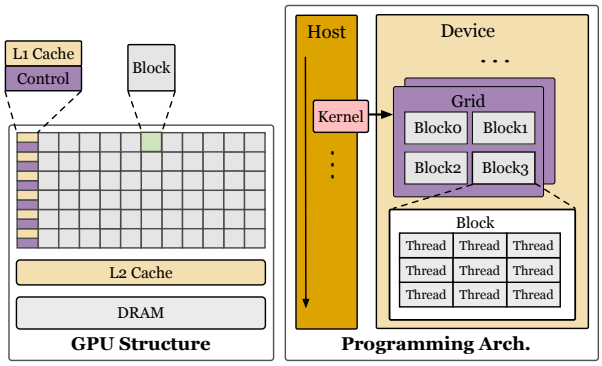

Fig. 3 GPU memory hierarchy and communication mode

vector $\boldsymbol{\Phi}_i \in \mathbb{R}^{1\times L}$. During inference, the dictionary $\boldsymbol{D}$ is pre-defined and can be directly used, while the coefficients $\boldsymbol{\Phi}_i$ need to be calculated in real-time. According to the pixel-level operation in Equation (2), the image-level transformation can be represented as Equation (3).

$$\boldsymbol{y} = \boldsymbol{F}\boldsymbol{B}^\top, \text{ with } \boldsymbol{F} = \boldsymbol{\Phi}\boldsymbol{D}, \tag{3}$$

with $\boldsymbol{F} \in \mathbb{R}^{HWs^2\times k^2}$ and $\boldsymbol{\Phi} \in \mathbb{R}^{HWs^2\times L}$. Some techniques have been proposed to learn the dictionary $\boldsymbol{D}$ and coefficient matrix $\boldsymbol{\Phi}$ [21]–[23]. In LAPAR [5], $\boldsymbol{D}$ is a group of Gaussian (G) and difference of Gaussians (DoG) filters which are pre-defined to accelerate the computations. The coefficient matrix $\boldsymbol{\Phi}$ is predicted via a residual network (which will be explained in detail in Section II-B).

Considering the communication patterns of Equation (3), $\boldsymbol{\Phi}$ and $\boldsymbol{B}$ usually occupy much more bandwidth compare with $\boldsymbol{D}$, *i.e.*,

$$HWs^2 \times L + HWs^2 \times k^2 \gg L \times k^2. \tag{4}$$

While considering the computation patterns, the dictionary $\boldsymbol{D}$ plays the key role. Whether the data in $\boldsymbol{\Phi}$ and $\boldsymbol{B}$ are ought to be computed is determined by $\boldsymbol{D}$, since $\boldsymbol{D}$ is the bridge connecting $\boldsymbol{\Phi}$ and $\boldsymbol{B}$. According to $\boldsymbol{D}$, if some computations can be skipped with no harm to the performance, we shall not load the data to on-chip memories, to save the precious bandwidth. The role of $\boldsymbol{D}$ makes the dictionary learning algorithm distinct from the traditional deep learning algorithms which only rely on weights and features. By optimizing the dictionary, it is believed that the communication and computation bottlenecks can be eased simultaneously.

B. SR Model Architecture

Typically, the deep dictionary learning-based models are composed of some residual units, convolutional layers, pixel-shuffle layers, dictionary assembling, and *etc*.

The state-of-the-art SR model LAPAR [5] is taken as an example to explain the model inference and the model structure. As shown in Fig. 2, during inference, there are four stages. Firstly, bilinear up-sampling is adopted to upscale the input image $\boldsymbol{x}$ to get the patch matrix $\boldsymbol{B}$. Secondly, the coefficient matrix $\boldsymbol{\Phi}$ is predicted by a *LaparNet* with the original $\boldsymbol{x}$ as the input. *LaparNet* is a stack of some local fusion blocks (LFBs) [24], pixel-shuffle layers, and several convolutional layers, while an LFB consists of some residual blocks, concatenations, multiplications, and addition operations. The third stage is dictionary assembling, in which the transformation matrix $\boldsymbol{F}$ is computed according to $\boldsymbol{\Phi}$ and the pre-defined dictionary $\boldsymbol{D}$. The final stage is filtering, in which the output HR image $\boldsymbol{y}$ is obtained by applying $\boldsymbol{F}$ to $\boldsymbol{B}$, *i.e.*, $\boldsymbol{y} = \boldsymbol{F}\boldsymbol{B}^\top$. To deploy the SR models on GPU efficiently, the dictionary learning is ought to be analyzed in detail which has not been considered in previous arts.

C. GPU Programming Architecture

The NVIDIA GPU architecture together with the CUDA programming model provides a well-designed abstraction that bridges the software applications and low-level hardware implementations, as illustrated in Fig. 3. The hardware architecture of a GPU is composed of some streaming multiprocessors (SMs). Each streaming multiprocessor consists of several processing blocks, some shared memory units, control logics, and *etc*. Each processing block contains a group of computation cores (CUDA cores, Tensor Cores, and *etc.*), register files, load/store units, and *etc*.

CUDA programming model is designed [25] to implement the computation tasks on the NVIDIA GPU. The programming model is composed of a host device (CPU) that controls the executions, and a device (GPU) that runs the kernel code to finish the computations, as shown in Fig. 3. Each kernel contains a computation grid that can be further divided into multiple blocks. Following the single instruction multiple threads (SIMT) mechanism, each block is partitioned into a group of threads that can run the same code on different data, synchronously. Usually, a thread is assigned to a hardware streaming processor. Once the kernel code is compiled, all of the threads will execute the same program in parallel, and thread blocks may execute in any order. These mechanisms will be carefully considered in this paper to obtain the optimal model deployments.

III. Optimization of Deployments on GPU

A. Dictionary Compression

The state-of-the-art SR model LAPAR [5] shows outstanding performance with a limited size of parameters ($<$ 1M). However, even using the state-of-the-art NVIDIA TensorRT [14], the inference performance cannot meet the demanding real-time requirement. The reason is that the high-resolution feature maps play the key role instead of the parameters in the SR models, as mentioned above in Section II-A. The running time profiling is shown in Fig. 1. The time distribution demonstrates that dictionary learning consumes the most time and is obviously the bottleneck. Such analysis result comes from the fact that the existing tools and methods, *e.g.*, TensorRT, as effective in acceleration for normal DNN kernels such as convolution, ReLU. For certain layers in the computation graph with less customized efficient implementation from TensorRT, they will cost inevitably a large amount of time.

Compressing the dictionary can ease both the computation and communication workloads, as has been analyzed in Section II-A. Consequently, a structural dictionary selection strategy is proposed in this paper to compress the dictionary. On the one hand, an ideal dictionary $\boldsymbol{D}$ can provide sufficient information for the restoration of image details. On the other hand, the dictionary $\boldsymbol{D}$ is expected to be less bulky to perform an inference within the required time limit, without degradation to the quality of results. A threshold value $\alpha \in (0, 1)$ is set to specify the sparsity of the dictionary $\boldsymbol{D} \in \mathbb{R}^{L\times k^2}$, *i.e.*, after compression, only the most representative $\alpha \cdot L$ items from L will be reserved. To avoid the greedy compression which would fall into the local optimum, the dictionary items are compressed iteratively until the sparsity threshold α is reached. In step t, the compression ratio is set to be α_t, with $\alpha_t < \alpha_{t-1}$. The most representative $\alpha_t L$ items are reserved while others are discarded. In the next step, we set $\alpha_{t+1} = \alpha_t - \Delta_\alpha$, to further prune more items. Meanwhile, parameters $\boldsymbol{W}$ of the *LaparNet* is fine-tuned accordingly to minimize the reconstruction error [26]–[29]. The problem can be formulated as:

$$\begin{aligned} \boldsymbol{\beta}, \boldsymbol{W} = \quad & \arg\min_{\boldsymbol{\beta},\boldsymbol{W}} \tfrac{1}{N} \left\| \boldsymbol{Y} - \textstyle\sum_{i=0}^{L} \beta_i \boldsymbol{\Phi} \boldsymbol{D} \right\|_2^2, \\ \text{s.t.} \quad & \boldsymbol{\Phi} = \mathit{LaparNet}(\boldsymbol{X}, \boldsymbol{W}), \\ & \|\boldsymbol{\beta}\|_0 \le \alpha L, \end{aligned} \tag{5}$$

where N is the size of input batch of images, $\boldsymbol{\Phi}$ is the coefficient vector extracted from *LaparNet* with parameters $\boldsymbol{W}$ and $\boldsymbol{Y}$ is the output matrix of this layer after dictionary query. $\boldsymbol{\beta}$ is the selecting vector on filters of $\boldsymbol{D}$ where $\beta_i = 0$ means the i-th item in the dictionary will be ignored during the compression process.

Further, we improve Equation (5) by considering the loss of the final results of the SR flow. In other words, the filtering stage after the dictionary query stage is considered, to guarantee the quality of results after compression. The reconstruction error $\mathscr{L}$ of the dictionary compression is defined as the difference between the ground truth high-resolution image

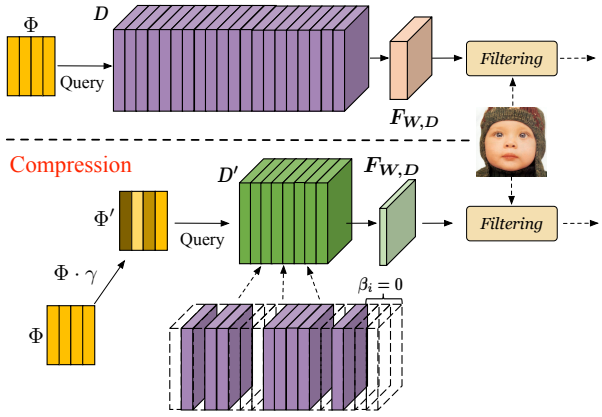

Fig. 4 Visual illustration of dictionary compression, the upper flow represents original dictionary query and filtering, namely stage 3 + stage 4 in Fig. 2, The flow below demonstrates the compression process of the dictionary query.

$\boldsymbol{H}_{gt}$ and the images generated by the compressed model, as shown in Equation (6).

$$\begin{aligned} \boldsymbol{\beta}, \boldsymbol{W} = \quad & \arg\min_{\boldsymbol{\beta},\boldsymbol{W}} \tfrac{1}{N} \left\| \boldsymbol{H}_{gt} - \boldsymbol{F}_{W,\beta} \boldsymbol{B}^\top \right\|_2^2, \\ \text{s.t.} \quad & \boldsymbol{F}_{W,\beta} = \textstyle\sum_{i=0}^{L} \beta_i \boldsymbol{\Phi} \boldsymbol{D}, \\ & \boldsymbol{\Phi} = \mathit{LaparNet}(\boldsymbol{X}, \boldsymbol{W}), \\ & \|\boldsymbol{\beta}\|_0 \le \alpha L. \end{aligned} \tag{6}$$

With two objectives $\boldsymbol{W}$ and $\boldsymbol{\beta}$, to solve this optimization problem efficiently, an alternating method including two steps is adopted. The first step is to search the suitable selecting vector β corresponding to the required α_t. The second step is to tune the parameters $\boldsymbol{W}$ corresponding to the reserved dictionary items with the minimization objective in Equation (6). The first step is actually an NP-hard problem. [27] suggested to relax the problem to ℓ_1 regulation. Therefore the objective $\boldsymbol{\beta}$ can be solved by utilizing the LASSO regression with parameter λ, as shown in Equation (7).

$$\begin{aligned} \boldsymbol{\beta} = \quad & \arg\min_{\boldsymbol{\beta}} \tfrac{1}{N} \left\| \boldsymbol{H}_{gt} - \boldsymbol{F}_{W,\beta} \boldsymbol{B}^\top \right\|_2^2 + \lambda \|\boldsymbol{\beta}\|_1, \\ & \text{s.t.} \|\boldsymbol{\beta}\|_0 \le \alpha L. \end{aligned} \tag{7}$$

The complete selection strategy is illustrated in Algorithm 1.

For channel selecting, we accumulated a batch of input feature map from the *LaparNet* prior to the dictionary query step as well as the ground-truth high-resolution images. The LASSO regression feature will be randomly sampled from the input feature map within width each plane. The λ in Equation (7) is carefully set to adjust the pruning ratio for filters. We begin with a tiny λ and increase the value exponentially until the reduced filter fits the required number. In lines 12–20 in Algorithm 1, a binary search is applied on λ_{t+1} within the range of last step size to adjust the compression ratio close to α_{t+1}.

The second step is to update the parameters in line 21 of Algorithm 1. The problem is formulated into:

$$\boldsymbol{W} = \arg\min_{\boldsymbol{W}} \frac{1}{N} \left\| \boldsymbol{H}_{gt} - \boldsymbol{F}_{W,D'} \boldsymbol{B}^\top \right\|_2^2. \tag{8}$$

Algorithm 1 Dictionary Selection Strategy

Input: $\boldsymbol{D} \in \mathbb{R}^{L \times k^2}$, small λ_0, target α, tolerance ϵ;
Input: pre-trained $\boldsymbol{W}_0$, coefficient matrix $\boldsymbol{\Phi}$;
$t \leftarrow 0$, $\alpha_0 \leftarrow 1.0$, $\beta_0 \leftarrow \mathbf{1} \in \mathbb{R}^L$, $\gamma_0 \leftarrow \mathbf{1} \in \mathbb{R}^L$;
$\mathscr{L} \leftarrow$ reconstruction error ▷ Equation (6)
repeat
  $\alpha_{t+1} \leftarrow \alpha_t - \Delta\alpha$;
  $\lambda_{t+1} \leftarrow \lambda_t$;
  while $|\beta_{t+1}|_0 > \alpha_{t+1} \cdot L$ **do**
    Fix $\boldsymbol{W}_t$, update $\beta_{t+1} \leftarrow \arg\min_{\boldsymbol{\beta}} \mathscr{L}(\boldsymbol{W}_t, \boldsymbol{\beta D}) + \lambda_{t+1} |\boldsymbol{\beta}|$; ▷ Equation (7)
    $\lambda_{t+1} \leftarrow 2 \cdot \lambda_{t+1}$
  end while
  $\lambda_{left} \leftarrow 0.5\lambda_{t+1}$, $\lambda_{right} \leftarrow \lambda_{t+1}$;
  while $\left|\alpha_{t+1} \cdot L - |\beta_{t+1}|_0\right| > \epsilon \cdot L$ **do**
    $\lambda_{t+1} = 1/2(\lambda_{left} + \lambda_{right})$;
    Fix $\boldsymbol{W}_t$, update $\beta_{t+1} \leftarrow \arg\min_{\boldsymbol{\beta}} \mathscr{L}(\boldsymbol{W}_t, \boldsymbol{\beta D}) + \lambda_{t+1} |\boldsymbol{\beta}|$;
    if $|\beta_{t+1}|_0 < \alpha_{t+1} \cdot L$ **then**
      $\lambda_{left} \leftarrow \lambda_{t+1}$;
    else if $|\beta_{t+1}|_0 > \alpha_{t+1} \cdot L$ **then**
      $\lambda_{right} \leftarrow \lambda_{t+1}$;
    end if
  end while
  Fix β_{t+1}, update $\boldsymbol{W}_{t+1} \leftarrow \arg\min_{\boldsymbol{W}} \mathscr{L}(\boldsymbol{W}, \beta_{t+1}\boldsymbol{D})$; ▷ Equation (8)
  $t = t + 1$;
until $\alpha_t \leq \alpha$

To fine-tune the parameters of the whole model to fit the new dictionary through training at each step is time-consuming. As shown in Equation (8), The $\boldsymbol{D}'$ is the Dictionary which is the compressed dictionary with layers neglected in the previous LASSO step. We can use linear regression in the iterative steps to reconstruct the parameters of the dictionary query layer which generates coefficient vector $\boldsymbol{\Phi}$ for fast tuning requirement. We note parameters of this layer as $\boldsymbol{W}_{D'}$ and parameters at i-th channel will be adjusted by the regression coefficient $\boldsymbol{\gamma}$. The iterative tuning step of Equation (8) will be re-written into Equation (9) where $\boldsymbol{W}_{D'}^{new}$ is the updated parameters. Note that $\boldsymbol{\gamma}$ is actually a weight coefficient on $\boldsymbol{W}_{D'}$ along channel dimension for update. The weighted coefficient matrix $\boldsymbol{\Phi}'$ is the new query to the selected dictionary $\boldsymbol{D}'$. The whole modified dictionary query and filtering flow are illustrated in Fig. 4.

$$\begin{aligned} \boldsymbol{\gamma} = \arg\min_{\boldsymbol{\gamma}} \quad & \frac{1}{N} \left\| \boldsymbol{H}_{gt} - \sum_{i=0}^{L} \gamma_i \boldsymbol{F}_{W,D'} \boldsymbol{B}^\top \right\|_2^2, \\ & \boldsymbol{W}_{D'}^{new} = \boldsymbol{\gamma} \boldsymbol{W}_{D'}. \end{aligned} \tag{9}$$

As shown in Fig. 5, the pruning process sustains the SR performance with barely no accuracy degradation. Essentially the well trained backbone network is capable of extracting sparse information for dictionary so the zero-out layers of the dictionary will not incur information loss. We can shrink to dictionary to size of 10% without noticeable accuracy loss. Compared with other widely-used SR models, *e.g.*, [30], [31] , our performance is the optimal.

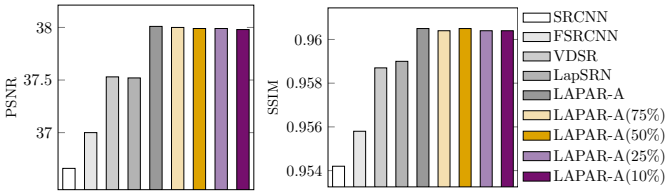

Fig. 5 Single image super-resolution (SISR) performance of our model with different dictionary compression ratios, in comparison with other SR methods. LAPAR-A (Per.%) represents our model with dictionary size shrunk to Per.%. PSNR means peak signal-to-noise ratio. SSIM means structural similarity index measure. PSNR and SSIM are two common metrics to measure the quality of images. The higher the better.

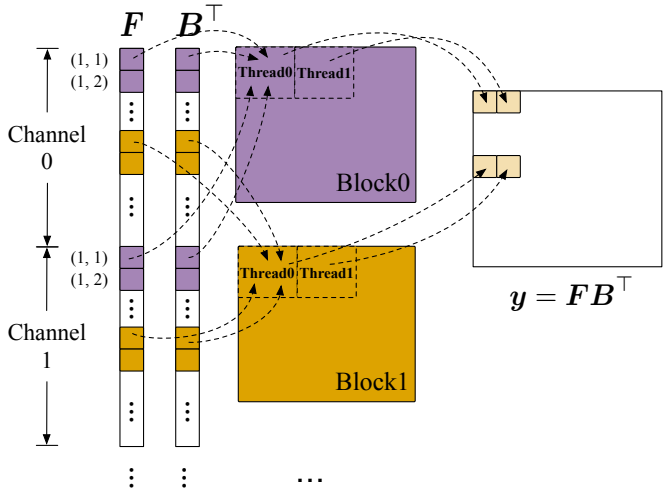

Fig. 6 An example of the proposed computation engine for image filtering operation.

B. Constraint-Based Optimization of Deployments On GPU

After shrinking the volume of the dictionary, we successfully slim the volume of the model. Another bottleneck for accelerating the computations is the filtering operation following the dictionary query, which can be regarded as a Hadamard product of matrices followed by a reducesum operation along channel dimension. Such kind of computations are common in SR tasks but ignored by the existing deployment tools. In this section, we take advantage of the parallelizing mechanism of GPU to improve the computation throughput from a low-level perspective. An example of the proposed computation engine is shown in Fig. 6.

The data (either images or filters) are stored in the NCHW format in linear memory addresses continuously, as shown in Fig. 6. The data with the same color are assigned to the same block to conduct the computations, *e.g.*, the data in purple are assigned to block 0, and the data in orange are assigned to block 1. The data with the same index but from various channels are assigned to the same thread. For example, the data at location $(1, 1)$ from these channels are all assigned to thread 0 in block 0, while the data at $(1, 2)$ are

assigned to thread 1 in block 0. Two values with the same location index from $\boldsymbol{F}$ and $\boldsymbol{B}$ are firstly multiplied. Then, the products of the data with the same location index but from various channels are summed up as the final result. In other words, each thread in Fig. 6 computes the product of two vectors, where each vector contains the data from all of the channels. Note that in our implementation, each thread directly adds each intermediate product to the final result. With this implementation, all of the threads can execute the same code segment and achieve parallel reduction perfectly without falling into the paradox of thread divergence [25]. Moreover, frequent cache interaction with main memory is avoided with our design. Blocks within an SM share same shared memory/L1 cache as shown in Fig. 3. Thus cache miss rate is reduced since data assigned to consecutive blocks in each channel is carefully stored consequently on memory as illustrated in Fig. 6.

The resources in GPU are limited, which concurrently restricts the computation patterns with respect to the threads, blocks, and *etc*. Different GPUs have distinct compute capabilities. For clarity, the edge embedded GPU NVIDIA Jetson Xavier NX which uses the Volta microarchitecture is taken as an example to illustrate this. From the perspective of hardware architectures, there are 6 streaming multiprocessors (SMs) in it. Each SM occupies a 96 KB shared memory/L1 cache. Each SM is further partitioned into 4 processing blocks. Every processing block has 16 FP32 cores, 8 FP64 cores, 16 INT32 cores, 2 Tensor cores, and a 64 KB shared register file. Besides, each processing block has a warp scheduler, to schedule the threads assigned to this processing block. From the perspective of the programming model, the computation kernel is executed as a grid of thread blocks. Each thread block (different from the processing block mentioned above) is assigned to a single streaming multiprocessor. Once the block is scheduled to an SM, threads in this block are further partitioned into warps. Every warp consists of 32 consecutive threads and all threads in a warp are executed in SIMT fashion. While the warps within a thread block may be scheduled in any order, the number of active warps is limited by SM resources. Four processing blocks in every SM of NX means there are at most four active warps in executing at the same moment. Besides, the number of warps in a thread block is constrained by the programming model to fit the sizes of warp schedulers, instruction registers, and *etc*. Sharing data in the shared register files among the parallel threads in the same processing block, or sharing data among the processing blocks in the same SM may cause a race condition: multiple threads accessing the same data in the memory simultaneously. Once a warp idles for the race conditions, the SM is free to schedule other available warps. The number of warps for a thread block can be determined as follows:

$$\text{Warps Per Block} = \left\lceil \frac{\text{Threads Per Block}}{\text{Warp Size}} \right\rceil, \tag{10}$$

where Warp Size $= 32$ for mainstream NVIDIA GPUs.

Suitable choices of blocks usually strike a good balance between the parallelism and resource race conditions and therefore stimulating the computations. In summary, the sizes of the blocks are constrained by the sizes of input data and the available on-board resources. Meanwhile, to accelerate the computations as much as possible, once the resources are available, the tasks will be assigned to occupy the resources. In other words, the parallelism is maximized so as to reach the upper-bound value of the resource utilization. Assume that the size of the three-dimensional input data is $D = H \times W \times C$, corresponding to the three dimensions of the thread blocks. Denote the number of SMs in GPU as S, the number of processing blocks in each SM as P, the size of register file in each processing block as R, the maximum number of threads in each warp as WS. The GPU compute capability constrains the number of warps in each block as smaller than T_{sm}. Denote the three dimensions of the thread block as (nx, ny, nz). Therefore, we have the following equations and constraints:

$$\begin{aligned} &T_r = (H \times W \times C)/(S \times P \times R), \\ &T \leq \min(T_r, T_{sm}), \\ &nx \times ny \times nz \leq WS \times P \times T, \end{aligned} \tag{11}$$

where T_r is computed by distributing data evenly to different SMs. The computational resources are implicitly organized and scheduled by the processing blocks. Therefore constraints on the computational resources are reflected in T_{sm}, which is a constant value determined by the compute capability. Constrained by T_r and T_{sm}, T represents the upper bound of the number of warps assigned to each processing block. Besides, the sizes are also constrained by the size of input data as follows:

$$\begin{aligned} &1 \leq nx \leq H, \\ &1 \leq ny \leq W, \\ &1 \leq nz \leq C. \end{aligned} \tag{12}$$

These constraints are usually ignored by designers, which wastes lots of optimization workloads. For examples, for $T \in [T_r, T_{sm}]$, these T values are legal while on-board resources are not fully utilized and the system parallelism can be further improved. With the above constraints, the feasible domain of the block sizes is shrunken significantly. The visualization view of the constraints are shown in Fig. 7. To the best of our knowledge, this is the first to consider these constraints for deployments of DNN models on GPUs, compared with previous arts, *e.g.*, [32].

With the target of minimizing the inference latency, we tend to build a regression model with respect to candidate values of nx, ny, and nz. The key challenge is that the clear form is the objective function is unknown because of the invisible execution process of GPU and CUDA programming model. Bayesian optimization is adopted in this paper as the searching algorithm to search the optimal configuration of the blocks with Gaussian process (GP) model utilized as the surrogate model [33]. Firstly, several configurations are randomly sampled from the design space to initialize the GP

TABLE I Inference Time (ms) and Acceleration ratios

Input size	Scale	NVIDIA GeForce RTX 2080 Ti					NVIDIA Jetson Xavier NX		
		PyTorch	TensorRT	Ours	Acc. (PyTorch)	Acc. (TensorRT)	TensorRT	Ours	Acc. (TensorRT)
64 × 64	×2	6.94	1.30	1.02	×680.39%	×127.45%	12.37	9.04	×136.84%
	×3	8.26	1.94	1.40	×590.00%	×138.57%	22.62	14.28	×158.40%
	×4	9.86	2.79	1.88	×524.46%	×148.40%	35.83	20.54	×174.44%
128 × 128	×2	8.74	3.59	2.66	×328.57%	×134.96%	52.12	37.25	×139.92%
	×3	13.04	6.19	4.16	×313.46%	×148.80%	90.33	54.26	×166.48%
	×4	18.07	9.71	6.13	×294.78%	×158.40%	144.34	81.29	×177.56%
180 × 320	×2	17.12	12.40	9.25	×185.08%	×134.05%	177.57	124.12	×143.06%
	×3	30.83	21.66	14.63	×210.73%	×148.05%	325.07	200.02	×162.52%
	×4	44.69	34.69	22.12	×202.03%	×156.82%	534.99	318.60	×167.92%
360 × 640	×2	67.36	50.26	37.47	×179.77%	×134.13%	748.72	530.23	×141.21%
	×3	105.32	88.45	59.20	×177.90%	×149.41%	1466.91	973.25	×150.72%
	×4	406.93	141.08	91.09	×540.02%	×154.88%	-	-	-
Average	-	61.43	31.17	**20.91**	**×352.27%**	**×144.49%**	328.26	**214.81**	**×156.28%**

Inference time on NVIDIA Jetson Xavier NX with input size 360 × 640 and scale 4 is not available due to the memory limit of the edge device.

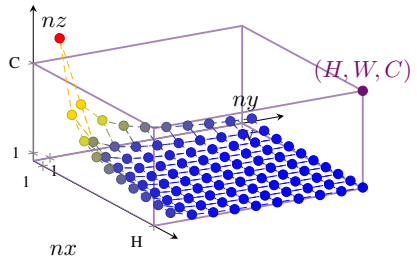

Fig. 7 The visualized solution space. The solution points below the dotted points are legal configurations.

model. And then the Bayesian optimization is used to iteratively select new configurations which have higher predictive performance reported by the GP model. The GP model is further optimized with newly sampled configurations and their on-chip inference latencies. Finally, the best configuration selected by the Bayesian algorithm in this exploration process is our optimal design.

IV. Experimental Results

A. Experimental Setup

Hardware Implementation: To validate the performance of our accelerator with respect to the acceleration ratio and quality of results, we deploy our proposed high-performance accelerator on edge embedded GPU NVIDIA Jetson Xavier NX, in comparison with the state-of-the-art tool NVIDIA TensorRT. NX integrates an ARM v8.2 64-bit CPU processor and a 384-core NVIDIA Volta GPU with 48 Tensor Cores. To take full advantage of the AI workloads, we use 15W of power to make it deliver up to 21 TOPS to compute. The clock frequency of the ARM processor is 2-core 1900MHz, and 4/6 core 1400MHz. The clock frequency of the GPU processor is 1100MHz. We also test our accelerator on NVIDIA GeForce RTX 2080 Ti with 4352 FP32 FPUs (CUDA cores) and 544 Tensor cores for accuracy evaluation. The results are also compared with PyTorch.

Software Implementation: All the designs are implemented by CUDA 11.0 and TensorRT 7.1.3. We use 32-bit floating point precision data types for evaluation. The training and evaluation of both original and our modified model are implementation through PyTorch based on official LAPAR code repository [42].

Dataset: The proposed accelerator is evaluated on common single image super-resolution (SISR) Set5 [43], Set14 [44], B100 [45], Urban100 [46], Manga109 [47] dataset.

B. Performance Evaluation

To evaluate the acceleration performance of our proposed accelerator, we compare it to the baseline designs on NVIDIA Jetson Xavier NX and RTX 2080 Ti. The inference time is measured with multiple input frame size and scale ratio. The results are in 32-bit floating point precisions. The running times are shown in TABLE I. We successfully realize SR with output of 540P quality to real-time inference. Compared with the widely-used PyTorch on 2080 Ti, our accelerator outperforms it by 352.27%. On average, our accelerator is faster than TensorRT by 144.49% on 2080 Ti and by 156.28% on Jetson Xavier NX, respectively. On various sizes of inputs and scales, our accelerator achieves +27.45%∼77.56% remarkable acceleration to TensorRT. An exciting result is that the acceleration ratios on Jetson Xavier NX are higher than on 2080 Ti. This shows the outstanding performance of our accelerator while handling the complex and difficult dictionary learning algorithms on limited computation and communication resources of edge embedded GPUs.

To demonstrate the high-quality results of our proposed accelerator without quality degradation, we compare the output images with other popular models, as shown in TABLE II. The performance metrics are PSNR (peak signal-to-noise ratio) and SSIM (structural similarity index measure), which are widely used to measure the qualities of images and videos. The higher values represent the better results. Note that our framework compresses the models with dictionary shrunk to 10% of the original size, while all of the baselines are not

TABLE II Comparisons on multiple benchmark datasets of our model and other popular SR networks. The dictionary in our model is compressed to 10% of original size for evaluation. Performance metrics are PSNR/SSIM. **Bold**: **best** results

Scale	Method	Set5	Set14	B100	Urban100	Manga109
×2	SRCNN [34]	36.66/0.9542	32.42/0.9063	31.36/0.8879	29.50/0.8946	35.74/0.9661
	FSRCNN [30]	37.00/0.9558	32.63/0.9088	31.53/0.8920	29.88/0.9020	36.67/0.9694
	VDSR [31]	37.53/0.9587	33.03/0.9124	31.90/0.8960	30.76/0.9140	37.22/0.9729
	DRRN [35]	37.74/0.9591	33.23/0.9136	32.05/0.8973	31.23/0.9188	37.92/0.9760
	LapSRN [36]	37.52/0.9590	33.08/0.9130	31.80/0.8950	30.41/0.9100	37.27/0.9740
	SRFBN-S [37]	37.78/0.9597	33.35/0.9156	32.00/0.8970	31.41/0.9207	38.06/0.9757
	FALSR-A [38]	37.82/0.9595	33.55/0.9168	32.12/0.8987	31.93/0.9256	-
	SRMDNF [39]	37.79/0.9600	33.32/0.9150	32.05/0.8980	31.33/0.9200	-
	Ours	**37.98/0.9604**	**33.59/0.9181**	**32.19/0.8999**	**32.09/0.9281**	**38.60/0.9771**
×3	SRCNN [34]	32.75/0.9090	29.28/0.8209	28.41/0.7863	26.24/0.7989	30.59/0.9107
	FSRCNN [30]	33.16/0.9140	29.43/0.8242	28.53/0.7910	26.43/0.8080	30.98/0.9212
	VDSR [31]	33.66/0.9213	29.77/0.8314	28.82/0.7976	27.14/0.8279	32.01/0.9310
	DRRN [35]	34.03/0.9244	29.96/0.8349	28.95/0.8004	27.53/0.8378	32.74/0.9390
	SelNet [40]	34.27/0.9257	30.30/0.8399	28.97/0.8025	-	-
	CARN [41]	34.29/0.9255	30.29/0.8407	29.06/0.8034	28.06/0.8493	-
	SRFBN-S [37]	34.20/0.9255	30.10/0.8372	28.96/0.8010	27.66/0.8415	33.02/0.9404
	Ours	**34.35/0.9267**	**30.33/0.8420**	**29.11/0.8054**	**28.12/0.8523**	**33.48/0.9439**
×4	SRCNN [34]	30.48/0.8628	27.49/0.7503	26.90/0.7101	24.52/0.7221	27.66/0.8505
	FSRCNN [30]	30.71/0.8657	27.59/0.7535	26.98/0.7150	24.62/0.7280	27.90/0.8517
	VDSR [31]	31.35/0.8838	28.01/0.7674	27.29/0.7251	25.18/0.7524	28.83/0.8809
	DRRN [35]	31.68/0.8888	28.21/0.7720	27.38/0.7284	25.44/0.7638	29.46/0.8960
	LapSRN [36]	31.54/0.8850	28.19/0.7720	27.32/0.7280	25.21/0.7560	29.09/0.8845
	CARN [41]	32.13/0.8937	28.60/0.7806	27.58/0.7349	26.07/0.7837	-
	SRFBN-S [37]	31.98/0.8923	28.45/0.7779	27.44/0.7313	25.71/0.7719	29.91/0.9008
	Ours	**32.15/0.8944**	**28.61/0.7817**	**27.59/0.7366**	**26.14/0.7873**	**30.39/0.9072**

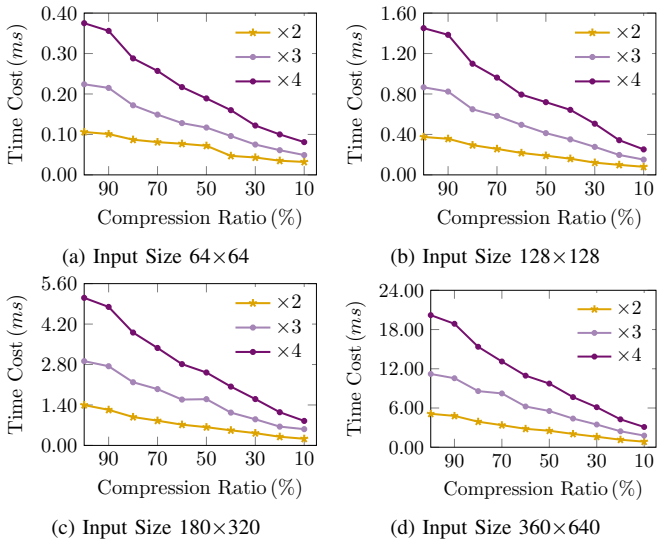

Fig. 8 Time consumptions of the dictionary query and filtering with different compression ratios. Different input image sizes and scaling factors (from 2 to 4) are evaluated.

compressed. The results show that our method is superior to all of the baselines on both of these two metrics.

Some ablation study results with respect to the compressions of the dictionary learning are shown in Fig. 8. The compression ratio of 100% represents the original dictionary without compression. As the compression ratios shrink, the time costs decrease continuously on all of these tests. When the compression ratio reaches 10%, the dictionary query and filtering flow is accelerated by up to nearly ×20.

C. *Discussions*

The results show the outstanding performance of our high-performance SR accelerator, especially on the resource-limited edge embedded GPU NVIDIA Jetson Xavier NX. The difficulties are resulting from the special memory and computation patterns of the dictionary learning algorithms, which cannot be handle by the existing tools. Another important reason is the great memory pressures because of the large scales of the super-resolution images. To the best of our knowledge, our proposed accelerator is the first to achieve superior performance on SR applications on edge embedded GPUs.

V. Conclusion

In this paper, a high-performance accelerator is proposed for the super-resolution model LAPAR. We introduce a lightweight compression method to learn representative dictionaries of the dictionary learning algorithm. A novel acceleration engine is designed to get the best efficiency and utilization of hardware resources. The evaluation results have shown our system outperforms the state-of-the-art tool TensorRT, and PyTorch on edge embedded GPU NVIDIA Jetson NX and 2080 Ti significantly, without quality degradation.

Acknowledgment

This work is partially supported by SmartMore and ITF Partnership Research Programme (No. PRP/65/20FX).

REFERENCES

[1] C. Dong, C. C. Loy, K. He, and X. Tang, "Image super-resolution using deep convolutional networks," *IEEE TPAMI*, vol. 38, no. 2, pp. 295–307, 2015.

[2] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proc. CVPR*, 2017, pp. 4681–4690.

[3] S. Y. Kim, J. Oh, and M. Kim, "JSI-GAN: GAN-based joint super-resolution and inverse tone-mapping with pixel-wise task-specific filters for UHD HDR video." in *Proc. AAAI*, 2020, pp. 11 287–11 295.

[4] W. Li, X. Tao, T. Guo, L. Qi, J. Lu, and J. Jia, "Mucan: Multi-correspondence aggregation network for video super-resolution," *Proc. ECCV*, 2020.

[5] L. Wenbo, Z. Kun, Q. Lu, J. Nianjuan, L. Jiangbo, and J. Jiaya, "LAPAR: Linearly-assembled pixel-adaptive regression network for single image super-resolution and beyond," in *Proc. NIPS*, 2020.

[6] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "FPGA/DNN Co-Design: An efficient design methodology for 1oT intelligence on the edge," in *Proc. DAC*. IEEE, 2019, pp. 1–6.

[7] C. Guo, Y. Zhou, J. Leng, Y. Zhu, Z. Du, Q. Chen, C. Li, B. Yao, and M. Guo, "Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration," in *Proc. DAC*. IEEE, 2020, pp. 1–6.

[8] H. Li, M. Bhargav, P. N. Whatmough, and H.-S. P. Wong, "On-chip memory technology design space explorations for mobile deep neural network accelerators," in *Proc. DAC*. IEEE, 2019, pp. 1–6.

[9] Q. Sun, C. Bai, H. Geng, and B. Yu, "Deep neural network hardware deployment optimization via advanced active learning," in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1510–1515.

[10] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. DAC*, 2017, pp. 29:1–29:6.

[11] R. Pinkham, S. Zeng, and Z. Zhang, "QuickNN: Memory and performance optimization of k-d tree based nearest neighbor search for 3D point clouds," in *Proc. HPCA*. IEEE, 2020, pp. 180–192.

[12] Y. Bai and W. Wang, "ACPNet: Anchor-Center Based Person Network for Human Pose Estimation and Instance Segmentation," in *Proc. ICME*, 2019, pp. 1072–1077.

[13] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and effective sparse LSTM on FPGA with bank-balanced sparsity," in *Proc. FPGA*, 2019, pp. 63–72.

[14] "NVIDIA TensorRT," https://docs.nvidia.com/deeplearning/tensorrt/index.html.

[15] "Intel MKL-DNN," https://github.com/oneapi-src/oneDNN.

[16] Y. Jung, Y. Choi, J. Sim, and L.-S. Kim, "eSRCNN: A framework for optimizing super-resolution tasks on diverse embedded CNN accelerators," in *Proc. ICCAD*. IEEE/ACM, 2019.

[17] X. Wei, Y. Liang, and J. Cong, "Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management." in *Proc. DAC*, 2019, pp. 125–1.

[18] Q. Sun, T. Chen, J. Miao, and B. Yu, "Power-driven DNN dataflow optimization on FPGA," in *Proc. ICCAD*, 2019, pp. 1–7.

[19] T. Dai, J. Cai, Y. Zhang, S.-T. Xia, and L. Zhang, "Second-order attention network for single image super-resolution," in *Proc. CVPR*, 2019, pp. 11 065–11 074.

[20] Y. Huang, S. Li, L. Wang, T. Tan *et al.*, "Unfolding the alternating optimization for blind super resolution," *Proc. NIPS*, vol. 33, 2020.

[21] J. Yang, Z. Wang, Z. Lin, S. Cohen, and T. Huang, "Coupled dictionary training for image super-resolution," *IEEE Transactions on Image Processing*, vol. 21, no. 8, pp. 3467–3478, 2012.

[22] Y. Romano, J. Isidoro, and P. Milanfar, "RAISR: rapid and accurate image super resolution," *IEEE Transactions on Computational Imaging*, vol. 3, no. 1, pp. 110–125, 2016.

[23] P. Getreuer, I. Garcia-Dorado, J. Isidoro, S. Choi, F. Ong, and P. Milanfar, "BLADE: Filter learning for general purpose computational photography," in *Proc. ICCP*. IEEE, 2018, pp. 1–11.

[24] C. Wang, Z. Li, and J. Shi, "Lightweight image super-resolution with adaptive weighted learning network," *arXiv preprint arXiv:1904.02358*, 2019.

[25] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C programming*. John Wiley & Sons, 2014.

[26] Y. He, P. Liu, Z. Wang, Z. Hu, and Y. Yang, "Filter pruning via geometric median for deep convolutional neural networks acceleration," in *Proc. CVPR*, 2019, pp. 4340–4349.

[27] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *Proc. ICCV*, 2017.

[28] Z. Huang and N. Wang, "Data-driven sparse structure selection for deep neural networks," in *Proc. ECCV*, 2018, pp. 304–320.

[29] T. Chen, B. Duan, Q. Sun, M. Zhang, G. Li, H. Geng, Q. Zhang, and B. Yu, "An efficient sharing grouped convolution via bayesian learning," in *IEEE TNNLS*, 2021, pp. 1–13.

[30] C. Dong, C. C. Loy, and X. Tang, "Accelerating the super-resolution convolutional neural network," in *Proc. ECCV*. Springer, 2016, pp. 391–407.

[31] J. Kim, J. K. Lee, and K. M. Lee, "Accurate image super-resolution using very deep convolutional networks," in *Proc. CVPR*, 2016, pp. 1646–1654.

[32] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI*, 2018, pp. 578–594.

[33] J. R. Gardner, G. Pleiss, D. Bindel, K. Q. Weinberger, and A. G. Wilson, "Gpytorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration," in *Proc. NIPS*, 2018.

[34] C. Dong, C. C. Loy, K. He, and X. Tang, "Learning a deep convolutional network for image super-resolution," in *Proc. ECCV*. Springer, 2014, pp. 184–199.

[35] Y. Tai, J. Yang, and X. Liu, "Image super-resolution via deep recursive residual network," in *Proc. CVPR*, 2017, pp. 3147–3155.

[36] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, "Deep laplacian pyramid networks for fast and accurate super-resolution," in *Proc. CVPR*, 2017, pp. 624–632.

[37] Z. Li, J. Yang, Z. Liu, X. Yang, G. Jeon, and W. Wu, "Feedback network for image super-resolution," in *Proc. CVPR*, 2019, pp. 3867–3876.

[38] X. Chu, B. Zhang, H. Ma, R. Xu, and Q. Li, "Fast, accurate and lightweight super-resolution with neural architecture search," in *Proc. ICPR*. IEEE, 2021, pp. 59–64.

[39] K. Zhang, W. Zuo, and L. Zhang, "Learning a single convolutional super-resolution network for multiple degradations," in *Proc. CVPR*, 2018, pp. 3262–3271.

[40] J.-S. Choi and M. Kim, "A deep convolutional neural network with selection units for super-resolution," in *Proc. CVPR*, 2017, pp. 154–160.

[41] N. Ahn, B. Kang, and K.-A. Sohn, "Fast, accurate, and lightweight super-resolution with cascading residual network," in *Proc. ECCV*, 2018, pp. 252–268.

[42] "Simple-SR," https://github.com/dvlab-research/Simple-SR.

[43] M. Bevilacqua, A. Roumy, C. Guillemot, and M. L. Alberi-Morel, "Low-complexity single-image super-resolution based on nonnegative neighbor embedding," in *Proc. BMVC*, 2012.

[44] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proc. CVPR*, 2017, pp. 4681–4690.

[45] D. Martin, C. Fowlkes, D. Tal, and J. Malik, "A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics," in *Proc. ICCV*, vol. 2, 2001, pp. 416–423.

[46] C. Ledig, L. Theis, F. Huszár, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang *et al.*, "Photo-realistic single image super-resolution using a generative adversarial network," in *Proc. CVPR*, 2017, pp. 4681–4690.

[47] W.-S. Lai, J.-B. Huang, N. Ahuja, and M.-H. Yang, "Deep laplacian pyramid networks for fast and accurate super-resolution," in *Proc. CVPR*, 2017, pp. 624–632.