2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00063

DEFCON: Deformable Convolutions Leveraging Interval Search and GPU Texture Hardware

Malith Jayaweera*, Yanyu Li†, Yanzhi Wang‡, Bin Ren§, David Kaeli¶
*Department of Electrical and Computer Engineering**†‡¶, *Northeastern University*, Boston, MA, USA
Department of Computer Science§, *William & Mary*, Williamsburg, VA, USA
{malithjayaweera.d, li.yanyu, yanz.wang}@northeastern.edu*†‡, bren@cs.wm.edu§, kaeli@ece.neu.edu¶

***Abstract*—Deformable convolutions can improve detection accuracy in Convolution Neural Networks (CNNs) by leveraging flexible spatial sampling in augmenting kernels with learnable offsets. However, the resulting irregular memory access patterns and additional pixel lookup overhead introduced by deformable layers pose inherent challenges when executed on high-throughput devices such as GPUs. To address these challenges, we introduce DEFCON, a systematic approach to optimizing deformable convolutions. DEFCON is designed to provide: (1) better placement of operators in the neural architecture using interval search, (2) reduced computational demands by leveraging lightweight operators, and (3) optimized inference by using GPU texture hardware. By performing an interval search, we reduce the number of deformable layers in our architecture. By leveraging the GPU's texture hardware, we are able to use lightweight operators to improve the execution performance of layers, without sacrificing prediction accuracy. By combining these approaches, DEFCON increases the inference performance by 2.8× over YOLACT++ implementation, when run on an NVIDIA Jetson AGX Xavier GPU. Our work enables faster and more accurate predictions when performing deformable convolutions.**

***Index Terms*—GPUs, Deformable Convolution, Interval Search**

I. Introduction

During the past decade, research has significantly advanced the state-of-the-art in object detection and image segmentation [1]–[9]. Convolution Neural Networks (CNNs) have paved the way for groundbreaking approaches toward object detection. Earlier CNNs were unable to effectively accommodate geometric or spatial variations in terms of object scale, pose, viewpoint, and partial deformation [10]. Therefore, two approaches were followed: i) data augmentation, which includes spatial variations in the training dataset [11], [12], and ii) handcrafting of feature layers, such as pooling [13], [14]. However, such highly specialized approaches could not be generalized for new datasets or handle complicated deformations that require a different receptive field.

Earlier CNNs used rigid geometric structures as the receptive field to perform spatial sampling in fixed locations [1], [9]. However, some regions in an image are commonly more important than others, thus it could be advantageous to have a more flexible spatial sampling pattern applied across the image [10], [15]. Dai et al. [10] introduced flexible spatial sampling that could be applied to deformable convolutions. They have been heavily leveraged to improve the accuracy of contemporary models, such as YOLACT [16] and YOLACT++ [17].

Deformable convolutions (DCNs) allow flexible kernel shapes through learnable offsets versus relying on neural network engineers to handcraft feature layers to extract specific features. These offsets are dynamically computed through an additional convolution layer for each input feature map. The computed offsets are then used to augment the spatial sampling available through a rigid kernel (used in regular convolutions). Thus, the adaptability of a neural network is significantly improved by allowing the neural network to accurately categorize variations that are not available in the training dataset.

Compared to standard convolution, the increase in adaptability comes with extra computational costs because of the additional convolution operations required to compute learnable offsets. However, the associated accuracy improvements can justify an increase in the compute time. As these models are now increasingly being deployed on edge devices, it has become important to provide real-time inference performance.

In this context, this work characterizes the underlying computational patterns associated with deformable convolution and explores how to properly optimize the execution of these operations on GPUs. In particular, this work first identifies unique challenges (and optimization opportunities) encountered when searching for the best neural network architecture with the presence of deformable convolutions during training and then exploits optimization opportunities specific to model inference.

There are two optimization opportunities during the training stage when performing the neural architecture search. First, the placement of deformable layers has usually been performed by hand [10], so the overuse of deformable layers to improve accuracy can lead to an inefficient inference. To address this problem, this work employs an interval-based search method to automate the placement of deformable layers, achieving better accuracy than the state-of-the-art YOLACT++ [17] framework. Second, using regular 2D convolutions to compute offsets results in performing two convolution operations instead of one (one for offset computation and the other for the actual convolution of the input/activations with the weights). Inspired by MobileNetV2 [9], our work here replaces a regular 2D convolution with depth-wise convolution operators to compute offsets. Both techniques are geared toward optimizing neural architecture and improving performance during the

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00063

training pipeline.

In addition to training-based optimizations, there are opportunities to optimize deformable convolutions during the inference stage. The first tries to optimize adaptive receptive field computations, which suffer from irregular access to the input pixels on the GPU. Since the spatial locality of memory accesses is low, the resulting computation is not GPU-friendly. Since the computed offsets are fractional, a direct lookup cannot be performed per pixel just by augmenting the original kernel coordinates with the offsets. A bilinear interpolation using four neighboring pixels is required. In the case of boundary pixels, additional branch statements are required to ensure program correctness. Both of the aforementioned challenges can be addressed effectively by treating computation as a graphics application instead of purely computational, particularly by leveraging GPU's built-in graphics processing hardware capabilities. We can leverage the GPU's texture memory to store pixel values with high spatial locality and support linear, bilinear, and trilinear interpolation. This idea has not been explored in previous state-of-the-art implementations of deformable convolutions (e.g., the one in PyTorch [18]). In addition, we perform a search space exploration for the best-suited tile size for deformable offset computation when using texture hardware. Our evaluation shows that the choice of tile size significantly impacts the performance of deformable offset computation.

In summary, this work proposes a multi-step optimization approach for deformable convolutions, namely `DEFCON`, considering both algorithmic optimizations and hardware characteristics. The resulting computation is better suited to the characteristics of the underlying GPU. Our contributions are summarized as follows:

- We automate the deformable layer placement by proposing a new interval search technique applied during the training stage to achieve the lowest inference latency and satisfy model accuracy. Through interval search, we find that applying deformable convolutions is particularly beneficial in the downsampling layers (i.e., regular 2-D convolution layers, with a stride=2).
- We propose a novel technique to load channel-wise coordinate offsets to the GPU texture units and perform bilinear interpolation using GPU hardware, instead of existing software implementations of bilinear interpolations.
- We also study the impact of tile size selection on execution performance when using texture units, and propose the use of a search space exploration to find the best tile size for a given model and GPU.

With our proposed interval search methodology and the efficient use of GPU texture hardware, we increase the inference performance by 2.8× over the state-of-the-art YOLACT++ on an NVIDIA Jetson AGX Xavier GPU.

II. Deformable Convolutions

This section reviews the characteristics of deformable convolutions and discusses some of the challenges associated with performance optimization when working with this workload targeting GPUs.

A. Mechanics of Deformable Convolution

A regular 2-D convolution samples the input feature map using a regular grid $\mathcal{R}$. Then it sums the sampled values, weighted by the values of w. The grid $\mathcal{R}$ defines the size of the receptive field along with the dilation [19]. As an example, for a 3×3 kernel with dilation 1, the grid: $\mathcal{R} = \{(i,j);\ i,j \in \{-1,0,1\}\ and\ (i,j) \neq (0,0)\}$.

Therefore, for each output feature map location in a regular convolution:

$$y(p_o) = \Sigma_{p_i \in \mathcal{R}} w(p_i).x(p_o + p_i) \tag{1}$$

In contrast, when using deformable convolution, the regular grid $\mathcal{R}$ is augmented with offsets $\Delta p_i (\ 0 \leq i \leq |\mathcal{R}|, i \in Z)$.

$$y(p_o) = \Sigma_{p_i \in \mathcal{R}} w(p_i).x(p_o + p_i + \Delta p_i) \tag{2}$$

Since these offsets Δp_i are fractional, a bilinear interpolation is required to determine the pixel value.

$$x(p) = \Sigma_q G(p,q) \cdot x(q) \tag{3}$$

where p denotes an arbitrary location $p_o + p_i + \Delta p_i$ and q enumerates all integral spatial locations in the activations. $G(x,y)$ is the bilinear interpolation kernel.

$G(p,q)$ is computed using component-wise computations: $g(p_x,q_x)$, $g(p_y,q_y)$ and where $g(p_x,q_x) = max(0, 1-|p_x - q_x|)$. Thus, only four neighboring pixels are required for the bilinear kernel.

We can outline the deformable convolution computation as follows:

❶ a regular convolution (conv2D), with the input activations and filter to derive the offsets.

❷ a convolution with the input activations and a filter augmented with offsets computed in the previous step used to derive the final convolution output.

The goal of the ❶ convolution step is to derive offsets in the x and y directions. The offsets are derived on a per-kernel value basis. For example, if a 3×3 kernel is used, 9 offset values are computed. Since the offsets are calculated in both x and y directions, the number of offsets for a 3×3 kernel is 18 ($18 = 9 \times 2$). The offset values can be shared per input channel through a variable called a deformable group. For example, when using deformable groups, input channels can be grouped such that offsets are shared between groups of input channels. This will keep the computational costs and memory requirements in check, which would not have happened if the input channels had not shared any offsets. As shown in Fig. 1, the output height and output width are computed similarly to a regular convolution. The number of output channels is equal to *deformable groups* × *kernel height* × *kernel width* × 2.

As the kernel moves through the image, the offsets also vary. In the deformable convolution operation (step ❷), these

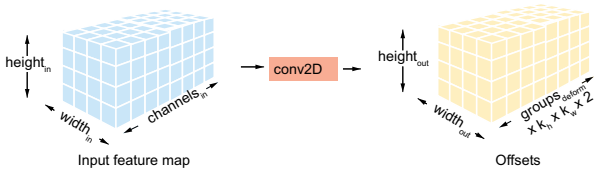

Fig. 1. Deformable offset computation.

offsets are augmented on the kernel coordinates to allow for arbitrary shapes. As mentioned earlier, since the offset computed (Δp_i) is fractional, a regular pixel lookup cannot be performed on x. Thus, we apply bilinear interpolation to each pixel value along the x and y directions, choosing the four nearest-neighbor pixels. In the case where the pixel of interest lies in a boundary region, the value of out-of-bounds neighbors is taken as zero. We will see that this computation is supported in texture hardware.

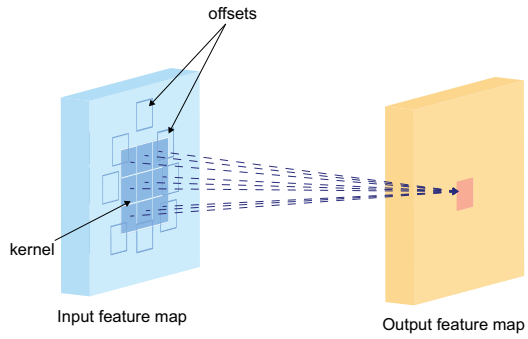

Fig. 2. Deformable convolution.

B. Hardware Deployment Challenges

Although deformable convolution provides us with flexibility, it also poses some inherent challenges. Compared to a regular conv2D operation, the use of offsets to allow for arbitrary kernel shapes means that the input pixels are often accessed in a more random manner. Thus, the efficiency of the GPU's coalescing unit will be impacted. Recent work on accelerator designs has attempted to address this problem through input and output tiling [20]. However, such tiling strategies on GPUs tend to be computationally expensive, as each inference would require a tile-matching mechanism.

The other main challenge faced with using deformable convolution is the additional overhead of bilinear interpolation. For each output feature pixel computation, a bilinear interpolation must be performed with four adjacent neighbors. Hence, for each output pixel, four multiplications must be performed, along with three additions. Current implementations often resort to bilinear interpolation at the software level. Thus, it would require additional clock cycles compared to a regular conv2D operation for a simple operation, such as the lookup of a pixel. However, as we will discuss in the next few sections, by treating this computation as a graphics problem, we are able to exploit much of the GPU's built-in hardware that supports bilinear interpolation and, in fact, reduce floating point operations while improving instruction level parallelism.

Algorithm 1 Gradient-based architecture search with Gumbel Softmax sampling, to search deformable convolutions.

Require: $\sum_n i_n > 0$
Ensure: $\sum_n \lceil \boldsymbol{\alpha}_n^1 > \boldsymbol{\alpha}_n^0 \rfloor \cdot \boldsymbol{\alpha}_n^1 \cdot t(\boldsymbol{w}_n) \approx \mathcal{T}$
Interval Search:
for each search epoch **do**
  for each iter **do**
    for each layer $\boldsymbol{n}$ **do**
      $\boldsymbol{d}_{n+1} = \sum_i \frac{e^{(\boldsymbol{\alpha}_n^i + \epsilon_n^i)/\tau}}{\sum_i e^{(\boldsymbol{\alpha}_n^i + \epsilon_n^i)/\tau}} \cdot \boldsymbol{w}_n^i(\boldsymbol{d_n})$
    end for ▷ get *output*
    $\mathcal{L} \leftarrow criterion(output, label)$
    $\mathcal{L}_s \leftarrow \left| \sum_n \lceil \boldsymbol{\alpha}_n^1 > \boldsymbol{\alpha}_n^0 \rfloor \cdot \boldsymbol{\alpha}_n^1 \cdot t(\boldsymbol{w}_n) - \mathcal{T} \right|^2$
    backpropagate $(\mathcal{L} + \mathcal{L}_s)$, update parameters
  end for
end for
Select Layer Type by the Magnitude of $\boldsymbol{\alpha}$.
Fine-tune the result architecture:
for each fine-tuning epoch **do**
  for each iter **do**
    for each layer $\boldsymbol{n}$ **do**
      $\boldsymbol{d}_{n+1} = \hat{\boldsymbol{w}}_n^i(\boldsymbol{d_n})$
    end for ▷ get *output*
    $\mathcal{L} \leftarrow criterion(output, label)$
    backpropagate $(\mathcal{L})$, update parameters
  end for
end for

III. Deformable Optimizations

We take a holistic approach to deformable optimizations and focus on both the training and inference stages. Fig. 3 summarizes our optimizations. We begin by deciding the placement of deformable operators in the neural network by using our interval search technique. Later, we reduce the computational cost of these layers by substituting regular convolutions with depthwise convolution operators (referred to as lightweight operators). To improve inference speed, we also include bounded deformations. In the final step, we perform GPU texture-based optimizations which boost the inference speed. Next, we will discuss each technique in detail.

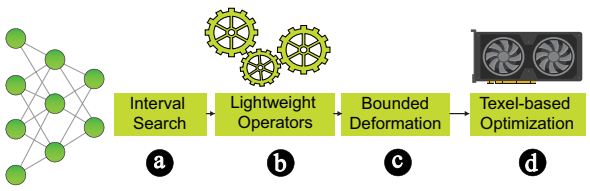

Fig. 3. Sequence of deformable optimizations.

A. Algorithm Improvements

ⓐ *Deformation Interval Search:* In state-of-the-art applications using deformable convolution [17], [21], deformable operations are often manually applied in the last several stages to extract high-level spatial information. Information is obtained by applying deformable convolution layers as a sequence or by interleaving regular 2-D convolution layers with deformable convolution layers to limit the high inference cost. For example, YOLACT++ [17] applies DCN in the last three stages of the ResNet backbone, using an interval of 3. It is obvious that these handcrafted design choices are far from optimal and lack adaptability when trying to classify untrained inputs, which is one of the main goals of using deformable convolutions. Instead of using hand-crafted layer placement, in this work, we propose a systematic method of deformable convolution layer placement guided by a gradient-based interval search method which automatically decides the best positions in the neural network to apply deformable convolutions. With such a system in place, we can automatically reduce the total number of deformable layers without compromising accuracy.

We formulate the differentiable interval search as a bi-level optimization problem:

$$\min_{\boldsymbol{W},\boldsymbol{A}} \mathcal{L}(\boldsymbol{D},\boldsymbol{W},\boldsymbol{A}) + \beta \cdot \mathcal{L}_s(\boldsymbol{A}) \tag{4}$$

where $\mathcal{L}$ refers to the task loss, $\boldsymbol{D}$ represents input data, $\boldsymbol{W}$ is the set of trainable parameters (i.e., weights) in the network, and $\boldsymbol{A}$ is the set of architectural parameters that determine whether to use regular 2-D convolution or deformable convolution. To manage the inference speed of the final model in a user-specified manner, a speed penalty $\mathcal{L}_s$ is applied to the architectural parameters if a deformable convolution operator is used, and β is a hyperparameter to balance loss and stabilize training.

Our gradient-based interval search approach is illustrated in Fig. 4(c). We first construct a dual-path layer [22], with a deformable convolution and a regular one. The convolution outputs are linearly combined using Gumbel Softmax sampling [23], which can be formulated as:

$$\boldsymbol{d}_{n+1} = \sum_i \frac{e^{(\boldsymbol{\alpha}_n^i + \epsilon_n^i)/\tau}}{\sum_i e^{(\boldsymbol{\alpha}_n^i + \epsilon_n^i)/\tau}} \cdot \boldsymbol{w}_n^i(\boldsymbol{d_n}) \tag{5}$$

where $\boldsymbol{d}$ represents data (features) of two consecutive layers: n and $n+1$, $\boldsymbol{\alpha}$ is the architecture parameter and $\boldsymbol{w}$ denotes trainable weights of the corresponding layer. $\epsilon \sim U(0,1)$ ensures exploration, and τ is the temperature. $i \in \{0,1\}$ represents the dual operator of regular convolution and deformable convolution in our case. With Gumbel Softmax sampling, the derivatives with respect to $\boldsymbol{w}$ and $\boldsymbol{\alpha}$ can both be easily computed, so that we can perform interval search in a gradient-based fashion.

As the target of our network architecture (interval) search, we aim to constrain the inference latency on edge GPUs (e.g., NVIDIA Jetson AGX Xavier). Prior work collected on-device latency data to build a lookup table [24]–[26], which was used to estimate candidate neural network layer latency during the search, or deploy each candidate layer on-chip to gather real latency data [27]. In this work, we build our search algorithm based on collecting on-device latency and building a lookup table, since deformable convolution is only applied in certain 3×3 conv2D layers, *e.g.*, the 3×3 convolution in the Bottleneck block in ResNet-101, and it is trivial to collect their latency with all possible configurations. If we denote $\boldsymbol{\alpha}^0$ as the architecture-specific parameter of a regular convolution, while $\boldsymbol{\alpha}^1$ is the corresponding parameter of the deformable convolution, the latency penalty of the interval search can be expressed as:

$$\mathcal{L}_s = \left| \sum_n \lceil \boldsymbol{\alpha}_n^1 > \boldsymbol{\alpha}_n^0 \rfloor \cdot \boldsymbol{\alpha}_n^1 \cdot t(\boldsymbol{w}_n) - \mathcal{T} \right|^2, \tag{6}$$

where t denotes the latency mapping that stores latency, based on DCN characteristics (feature size, input and output channel, etc.). $\lceil \cdot \rfloor$ is an element-wise transform from $\{True, False\}$ to $\{1, 0\}$, and does not require a gradient. $\mathcal{T}$ is the target latency, and $\sum_n \cdot$ denotes the latency accumulated by blocks. In summary, the gradients with respect to the architecture parameters are computed as follows:

$$\begin{aligned} G_{\alpha^0} &= \frac{\partial \mathcal{L}(D,W,A)}{\partial \alpha^0} \\ G_{\alpha^1} &= \frac{\partial \mathcal{L}(D,W,A)}{\partial \alpha^1} + \beta \cdot \frac{\partial \mathcal{L}_s}{\partial \alpha^1} \end{aligned} \tag{7}$$

where the second derivative in G_{α^1} (*e.g.,* $\frac{\partial \mathcal{L}_s}{\partial \alpha^1}$) can be calculated from Equation (6).

$$\frac{\partial \mathcal{L}_s}{\partial \alpha_n^1} = 2 \times \Big(\lceil \boldsymbol{\alpha}_n^1 > \boldsymbol{\alpha}_n^0 \rfloor \cdot \boldsymbol{\alpha}_n^1 \cdot t(\boldsymbol{w}_n) - \mathcal{T} \Big) \cdot \lceil \boldsymbol{\alpha}_n^1 > \boldsymbol{\alpha}_n^0 \rfloor \cdot t(\boldsymbol{w}_n) \tag{8}$$

The entire workflow of our proposed interval search is shown in Algorithm 1.

ⓑ *Lightweight Offset Convolution:* In the context of deformable convolution, input-adaptive spatial sampling is achieved by introducing an additional convolution layer that learns and predicts the offset for each input feature map during inference. Our observations have shown that using regular 2-D convolution is excessive for this purpose. Hence, inspired by MobileNet [9], we replace standard 2-D convolutions with depth-wise convolution operations (also referred to as a lightweight convolution in this work). The computational cost of a regular 2-D convolution is $W^{2k^2 \times C \times 3 \times 3}$, where 3 is the kernel size, C is the input channel number, and k is the kernel size of the following deformable convolution. However, our lightweight implementation replaces this convolution with two consecutive layers, $W_1^{C \times 1 \times 3 \times 3}$ and $W_2^{2k^2 \times C \times 1 \times 1}$, where W_1 is a depth-wise 3×3 convolution and W_2 is a 1×1 convolution to linearly combine the output and transform it to the required dimension ($2k^2$). Assuming that we perform a deformable convolution with $k = 3$, this lightweight replacement can reduce multiply-accumulate (MAC) operations by:

$$1 - \frac{H \times W \times 3 \times 3 \times C + C \times H \times W \times 1 \times 1 \times 2 \times k^2}{C \times H \times W \times 3 \times 3 \times 2 \times k^2} = 83.3\% \tag{9}$$

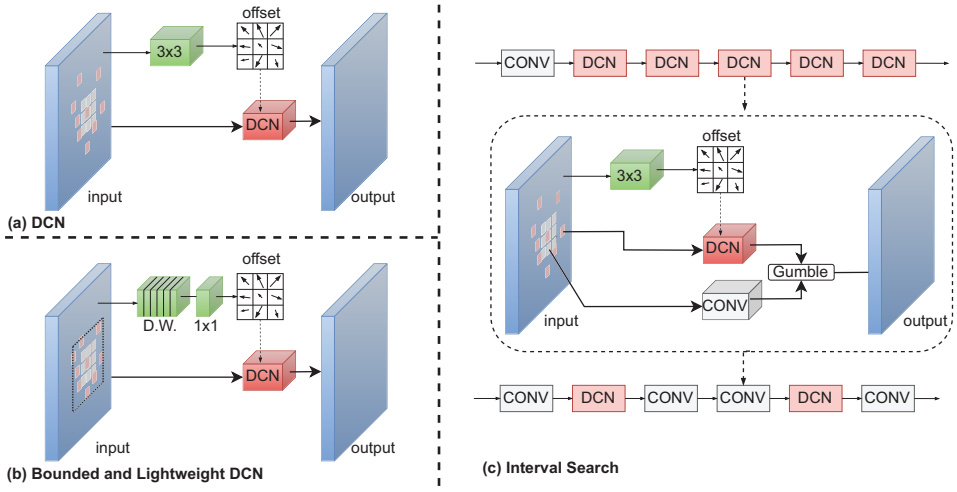

Fig. 4. The DCN optimization paradigm. (a) is the original DCN; (b) shows optimizations applied using the bounded offset and light-weight convolution; (c) illustrates the gradient-based interval search.

Batch normalization and ReLU activation are applied after the first depth-wise convolution, but not for the following 1×1 convolution, as it outputs the floating-point offsets used to augment the kernel as explained in section II-A. The architecture of our lightweight DCN is shown in Fig. 4(b).

ⓒ *Bounded Deformation:* The flexible receptive fields of DCNs tend to reduce the available spatial locality of input pixel access during kernel execution. Thus, this computation pattern is not the most hardware-friendly form of convolution. One intuitive and more reliable solution is to set an upper limit on the deformation so that memory accesses to outliers are avoided, improving the spatial locality. Furthermore, enabling an arbitrarily large receptive field within a single layer is unnecessary from a spatial perspective since using a stack of conv2D layers can naturally achieve the same effect in detecting image features. In practice, if we assumed the upper boundary of offsets was defined by the hardware accelerator was P, we could restrict the learned offset to $[0, P]$ before the application of the deformable kernel. The application of the bounded deformation is illustrated in Fig. 4(b). We determine the appropriate boundary value (P) by exploring the boundary limits, as shown in Fig. 5. ∞ denotes unrestricted deformation, and the lowest boundary is chosen to be the kernel size. We observe that choosing an upper bound of greater than 7 for deformable convolutions leads to negligible accuracy gains compared to setting the upper bound to 7. Hence, we determine that using an upper bound of 7 is the best choice for deformable convolutions in terms of accuracy.

A closely related approach is to use *rounded offsets* [28], [29], which round the sampling coordinates to their closest integer values so that bilinear interpolation can be avoided. However, rounding the sampling coordinates which are in floating point format to integers ultimately results in a significant loss of accuracy [28], [29]. Therefore, we do not employ such techniques in our model training and rely on GPU hardware to perform bilinear interpolation.

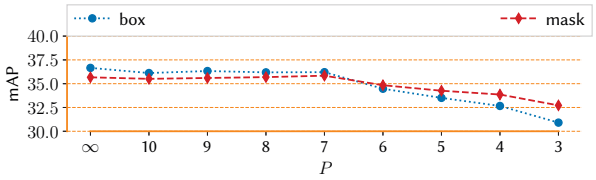

Fig. 5. Determining the P value for bounded deformation.

Another closely related and popular technique is to employ *square-shaped deformation*, which constrains the shape of the deformable kernel [28], [29] to promote better spatial locality of memory accesses. However, the resulting convolution kernel is effectively a simple dilated convolution using a rigid kernel size, forgoing the superior advantages of using flexible receptive fields available with the original version of the deformable convolution [10].

B. Hardware Optimization

ⓓ *Texel-based Optimization:* GPUs provide a number of hardware features specifically for graphics processing. In this section, we will look at how to leverage the texture units and use reduced-precision interpolation to improve performance.

GPU texture memory is a read-only memory that is designed to fetch data elements for applications that possess high spatial locality. Texture memory is also cached in a dedicated, read-only, texture cache. However, it is designed to stream fetches with a constant latency [30].

Textures have built-in functionality, such as read mode, addressing mode, and filtering mode; Read mode allows access to pixels using normalized coordinates. Addressing mode specifies how out-of-bounds pixels are handled. The default mode is used to set values to zero for out-of-bounds values. With normalized coordinates, wrap mode, and mirror mode are also

available for addressing. With the wrap mode, each coordinate x is converted to $frac(x) = x - floor(x)$, where $floor(x)$ is the largest integer not greater than x. With mirror mode, each coordinate x is converted to $frac(x)$, if $floor(x)$ is even, and $1 - frac(x)$ if floor(x) is odd. Filtering mode specifies how the fetch value of the texture is computed based on the input texture coordinates. Linear texture filtering performs low-precision interpolation between neighboring texels. When enabled, the texels surrounding a texture fetch location are read. The returned values are interpolated based on where the texture coordinates fall between the texels. Linear interpolation is performed for one-dimensional textures, bilinear interpolation for two-dimensional textures, and trilinear interpolation for three-dimensional textures. Using these features, software-level interpolation of pixels can be converted to a hardware-level interpolation of texels.

One of the major challenges of using texture memory for input feature map storage is the requirement to have a layered view (i.e., each channel of the input feature map should be able to perform interpolations without interfering with neighboring channels). NVIDIA GPUs that have texture memory provide two types of layered texture memory types: layered texture and mipmapped arrays. As mipmapped arrays are pre-computed pyramidal structures of optimized sequences of images that are not suited for layered computations, we limit our discussion to layered textures [31].

A *layered texture* (whether 1-dimensional or 2-dimensional) is a texture made up of a sequence of layers, all of which are regular textures with the same dimensions, size, and data type. This construct can be easily adapted to an input feature map, as each layer can be used to store separate channels. Also, since the CUDA API provides one-step data loading for layered textures, the programming complexity and the memory transfer cost are comparatively low. However, the input feature maps are 4-D tensors and consist of batch, channel, height, and width dimensions. This problem can be addressed by merging the 1^{st} and 2^{nd} dimensions, and using an index to navigate between mini-batches (i.e., $batch_{idx}\times$ number of channels).

Mipmapped arrays are widely used in gaming engines to simulate the illusion of near and far objects by lowering the resolution of distant objects in the game, until the user is in close proximity to the object [32]. Thus, mipmaps are pre-computed pyramidal structures of optimized sequences of images, each of which has a progressively lower resolution representation than the previous image. However, due to the pyramidal structure of mipmaps, each layer must be loaded and computed using the previous layer. Since this functionality is inconsistent with our desired behavior, we use a layered texture for storage. Furthermore, there is also surface memory, which provides read and write support. However, as we are not interested in writes to input features, we do not consider leveraging surface memory for storage.

One thing to note is that 2-D layered textures on an NVIDIA Jetson Xavier AGX GPU are limited to 32,768 by 32,768 by 2048 (height, width, and number of layers, respectively). Since layered dimensions are used to store both batch & channel dimensions, the number of batches $\times$ the number of channels should be less than or equal to 2048. Since we are targeting inference, this limit is more than enough to support almost all convolutional networks targeting vision. However, during training, with the use of multiple GPUs and multiple mini-batches (as high as 32 or 64), the developers must be aware of the texture memory limitations. To handle memory limitations, a partitioning scheme can be used so that only a subset of mini-batches are loaded into texture memory. However, such a scheme results in the overhead associated with multiple invocations of the GPU kernel. Therefore, during large mini-batch training, it could be more beneficial to use global memory and to strategically avoid the use of texture units for larger layers. We leave this analysis for future work.

IV. Evaluation

This section evaluates the accuracy of DEFCON[1] by comparing it with three state-of-the-art YOLACT++ neural networks: YOLACT with ResNet50 backbone, YOLACT++ with ResNet50, and YOLACT++ with ResNet101. We then evaluate the execution performance of DEFCON by comparing it with YOLACT++ (ResNet101 backbone).

A. Evaluation Methodology

Evaluation Objectives. Our overall evaluation demonstrates speedups produced by DEFCON through the use of GPU texture cache and the benefits of our interval search method for neural architecture search. Specifically, our evaluation has three objectives: (1) demonstrating that DEFCON optimizations achieve similar or improved accuracy compared to the state of the art, (2) showing that DEFCON outperforms the PyTorch framework when performing deformable operations, without compromising accuracy, and (3) exploring performance effects and explaining why DEFCON brings performance gains.

Benchmarks and Datasets. Our experiments use YOLACT++ [17], a state-of-the-art real-time instance segmentation model, which applies deformable convolutions. We use YOLACT++ with ResNet50 and ResNet101 as the backbone for accuracy evaluations and with ResNet101 as the backbone for performance evaluations as more DCNs are used with the latter backbone. We train the model on MS COCO [33], which is a large-scale object detection, segmentation, key-point detection, and captioning dataset available from Microsoft. We employ the 2017 release, which has a split of 118K/5K images for training/validation. To evaluate performance, we follow the standard COCO metric, computing the mean Average Precision (mAP) measured over multiple IoU thresholds, and we report both box and mask mAP for this instance segmentation task.

Hardware and Software Configurations. We evaluate the DEFCON performance on an Nvidia Jetson AGX Xavier GPU running Jetpack 4.5.1 and PyTorch 1.9.0. In addition, we evaluate the performance of DEFCON on an Nvidia 2080Ti

[1]Artifact available at https://github.com/malithj/dcn-defcon.

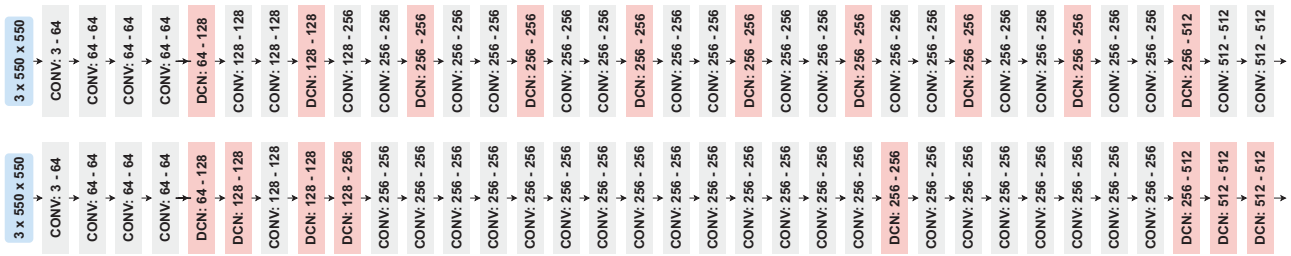

Fig. 6. Interval search results on YOLACT++ [17] with ResNet101 backbone network. Each box represents the 3×3 convolution of the residual block. Top: applying DCN layers at an interleaved interval of 3 as proposed in YOLACT++. Bottom: applying DCN using our proposed interval search method. Our interval search reduces the number of DCN layers by 2 while achieving higher performance ($+1.05$ Mask mAP).

TABLE I
ACCURACY OF OUR OPTIMIZED DCN ON YOLACT [16], [17] INSTANCE SEGMENTATION TASK.

Method	Backbone	Resolution	# of DCNs	Box mAP	Mask mAP	Mask AP50
YOLACT	ResNet50	550	0	29.79	27.97	45.92
YOLACT++	ResNet50	550	13	34.72	34.51	54.22
YOLACT++	ResNet50	550	5	34.69	34.11	53.27
Ours	ResNet50	550	5	34.81	34.44	53.85
YOLACT	ResNet101	550	0	32.07	29.73	48.01
YOLACT++	ResNet101	550	30	36.68	35.75	55.05
YOLACT++	ResNet101	550	10	35.07	34.62	53.79
Ours	ResNet101	550	8	35.38	35.35	55.51

running PyTorch 2.1. Following the configurations presented in prior work [17], we set the input resolution to 550×550. We employ an initial learning rate of 10^{-2} and decay by 10^{-1} at selected iterations, which saturates to 10^{-6}. For training purposes, we use a total mini-batch size of 128 images. As a result, the number of training iterations is decreased from 800k to 50k, as we increase the total mini-batch size. A Stochastic Gradient Descent (SGD) optimizer is used for training with a momentum of 0.9.

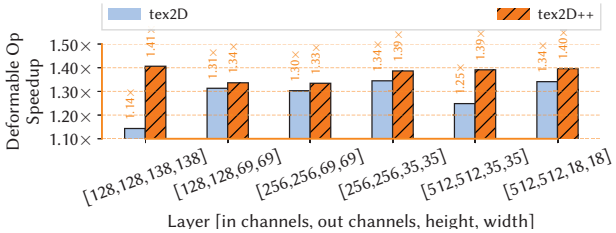

Fig. 7. Deformable operation speedup.

TABLE II
DEFORMABLE OPERATION SPEEDUP ON XAVIER.

In ch	Out ch	H	W	PyTorch (ms)	tex2D (ms)	tex2D++ (ms)	Speedup w.r. Torch
128	128	138	138	6.87	6.01	4.89	1.41×
128	128	69	69	23.03	17.54	17.23	1.34×
256	256	69	69	23.02	17.67	17.25	1.33×
256	256	35	35	47.87	35.60	34.53	1.39×
512	512	35	35	25.25	20.22	18.15	1.39×
512	512	18	18	97.00	72.33	69.48	1.40×

B. Accuracy Evaluation on Instance Segmentation

We report accuracy results in Table I, corresponding to applying the optimizations described in Section III-A. To test a backbone image detection network, we use ResNet50 and ResNet101 [3]. Utilizing the same or even fewer DCN layers, our method consistently outperforms baseline YOLACT++ models by a large margin of accuracy. When using the ResNet50 backbone, our optimized network outperforms the YOLACT++ baseline by 0.33 mask mAP. Further, with the ResNet101 backbone, we can achieve a $+0.73$ mask mAP with bounded offsets, using lightweight modifications and with even two fewer DCN layers than YOLACT++.

We demonstrate the benefits of using the DCN interval search method using a ResNet101 backbone network in Fig. 6. Our interval search reduces the number of DCN layers by 2, while achieving a $+1.05$ Mask mAP, by determining the best placement positions of DCN layers in the network. We observe that deformable convolutions are especially advantageous in downsampling layers (i.e., applied to regular 2-D convolution layers, with a stride=2) and in the final layers. Downsampling operations enlarge the receptive field, but there is a higher degree of filtering of information. Consequently, downsampling layers are critical to performance in most CNNs. It is beneficial to enhance downsampling convolution with deformation, as it is able to capture some of the information that might otherwise have been lost. Towards the latter part of the network, flexible receptive fields of DCNs are much more capable than rigid receptive fields in extracting feature dependencies and spatial information, as found using our interval search technique.

C. Execution Performance Results

We evaluate the execution performance of our proposed implementation of deformable convolutions leveraging texture units. To facilitate understanding of the maximum speedups, Fig. 7 shows the layer-wise performance gains of the deformable operator that can be achieved using texture units.

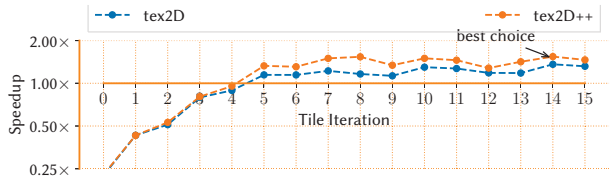

Fig. 8. Tile size selection for *tex2D* and *tex2D++* x(the y-axis is in log scale).

Fig. 7 compares the speedup obtained over PyTorch on Xavier GPU when using layered textures (*tex2D*) and reduced bit bilinear interpolation (*tex2D++*) where we only use 16 bits to compute the offset. Note that the *tex2D++* technique is not the same as applying quantization, which results in an information loss from input feature maps. The bit-reduced computation in *tex2D++* is only used to perform bilinear interpolation using the offsets derived at training time. In contrast, quantization reduces the precision of the input feature map and/or the filter by mapping values from a larger scale to a smaller scale. Thus, *tex2D++* does not result in any negative impact on accuracy. As we can see in Fig. 7, we accelerate deformable operation by 1.27× with *tex2D*, and by 1.39× with *tex2D++*. Due to the reduction in memory bandwidth, the performance of *tex2D++* is superior to *tex2D*. In addition, we show the speedup obtained over PyTorch 2.1 when executing *tex2D* and *tex2D++* on the 2080Ti GPU in Table IV.

Next, Table III shows the end-to-end execution speedup of DEFCON over our baseline neural network (YOLACT++) with varied optimizations we designed. Specifically, with all optimizations, DEFCON outperforms YOLACT++ by up to 2.80×. To prove that DEFCON results in good accuracy and to help readers better understand the performance gains, Table III also shows multiple accuracy results and the performance gains only coming from *tex2D* and *tex2D++*, respectively. More specifically, the interval search method greatly improves mask mAP by 1.05 over baseline YOLACT++, and brings 1.25× speedup over YOLACT++ because it uses 2 fewer DCN layers than YOLACT++. Bounding the offsets improves mask mAP, but slightly impacts box mAP, while maintaining reasonable performance in terms of accuracy. Lightweight layer modifications result in a slight drop in accuracy, but DEFCON still outperforms state-of-the-art YOLACT++. Moreover, with varied optimizations, all *tex2D* layers achieve a speedup up to 2.20×, and *tex2D++* layers achieve a speedup up to 2.24×, respectively. By substituting regular 2d convolutions with lightweight operators to compute offsets, we are able to achieve more than a 2× performance improvement, with an acceptable level of impact on accuracy as shown in Table III.

D. Optimization Evaluation

Next, to better understand our performance gains, we investigate the performance effects of each proposed algorithmic optimization and the usage of texture units more carefully. We use the YOLACT++ model (with ResNet101 backbone) as our baseline (PyTorch) which we discover using our interval search technique explained in section III-A. The baseline (PyTorch) deformable convolution shown as interval search in Fig. 9 performs a regular 2-D convolution to first compute the offsets, and then uses the offsets to perform the actual deformation using the flexible receptive field. We can see a few trends in our results. First, better texture unit utilization (*tex2D*) produces a speedup of the deformable layers. However, we only observe moderate speedups for layers $[128, 128, 138, 138]$. This is due to the larger input feature map height and width, which increases the number of texel computations for bilinear interpolation. Second, with *tex2D++*, we achieve a slight speedup of layers over *tex2D* for the interval search and bounded configurations. Finally, another interesting observation is that contrary to the recent work [28], [29] on accelerators where bounded offset techniques seem to deliver superior performance, we did not observe speedup improvements while using bounded offsets on the GPU.

Next, we consider the importance of selecting GPU-specific parameters (e.g., tile size) to increase SM utilization and exploit spatial locality. We search for the best tile size for *tex2D* and *tex2D++* using the ytopt [34] autotuning framework that employs Bayesian optimization. The search is conducted offline, thus avoiding runtime overhead. Fig. 8 plots the speedup of *tex2D* and *tex2D++* over our baseline, clearly showing that tile size significantly affects the resulting speedup, and our autotuning-based tile size search results in the best performance.

To further analyze the benefit brought by texture memory usage, we investigate GPU metrics using nvprof in Fig. 10. *nvprof* provides MFLOP, Global Load Transactions per Request, Global Load Efficiency, Texture load requests, as shown in Fig. 10. We can see that PyTorch (baseline YOLACT++) does not use any texture load requests, where *tex2d* and *tex2d++* use texture load requests. By observing the MFLOP count, we can see that there is a reduction of floating point operations by about 4× (approximately) due to performing hardware bilinear interpolation using texture units instead of software bilinear interpolation used by PyTorch. The PyTorch native implementation performs bilinear interpolation using four neighboring pixels. The ratio is not exactly four, as boundary pixels are often not computed and are substituted as zero [10]. Global load efficiency (GLD Efficiency) measures how many of the DRAM memory accesses are coalesced. Due to the texture unit utilization, the global load efficiency reaches 100% for all layers. Finally, we also observe that the number of global memory transfers performed per each memory request (GLD Transactions/Request) also decreases. Thus, texture unit utilization improves the spatial locality of data accesses.

TABLE III

SUMMARY OF ACCURACY AND SPEEDUP RESULTS ON XAVIER. *Search* REFERS TO USING INTERVAL SEARCH, *Boundary* REFERS TO USING BOUNDED OFFSET, AND *Light* REFERS TO USING LIGHTWEIGHT DCN. B.L. SHOWS THE TOTAL ELAPSED TIME FOR THE PYTORCH BASELINE (WITHOUT TEXEL-BASED OPTIMIZATIONS). THE SPEEDUP OVER YOLACT++ (RESNET101 BACKBONE) IS SHOWN SEPARATELY - OUR INTERVAL SEARCH DELIVERS A 1.25× IMPROVEMENT DUE TO THE REDUCTION OF DCN LAYERS, WITHOUT COMPROMISING ACCURACY.

Method				Accuracy			B.L.	*tex2D++*	*tex2D*	*tex2D++*	Speedup over
Search	Boundary	Light	*tex2D*	Box mAP	Mask mAP	Mask AP50	(ms)	(ms)	Speedup	Speedup	YOLACT++
				35.07	34.62	53.79	478.12	-	-	-	1.00×
✓				36.66	35.67	55.84	382.49	-	-	-	1.25×
✓			✓	36.66	35.67	55.84	382.49	332.60	1.13×	1.15×	1.44×
✓	✓		✓	36.21	35.84	55.60	384.41	329.73	1.13×	1.16×	1.45×
✓		✓	✓	35.42	35.21	55.48	222.37	171.52	2.20×	2.23×	2.79×
✓	✓	✓	✓	35.38	35.35	55.51	224.99	171.01	2.20×	2.24×	2.80×

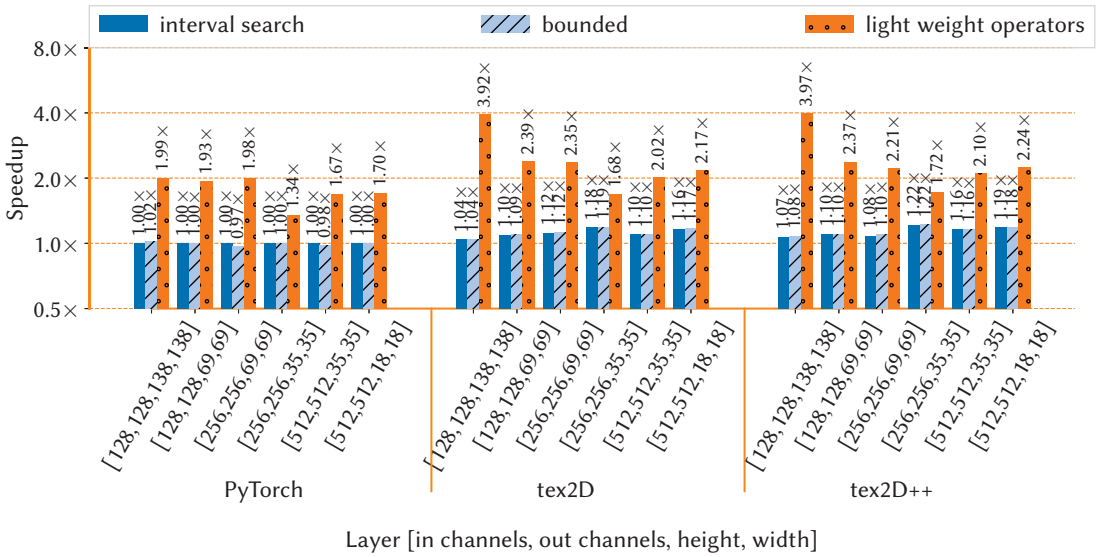

Fig. 9. Speedup of algorithmic optimizations on Xavier. The baseline is YOLACT++ model (with ResNet-101 backbone) we discover using our "interval search" technique (y-axis in log scale).

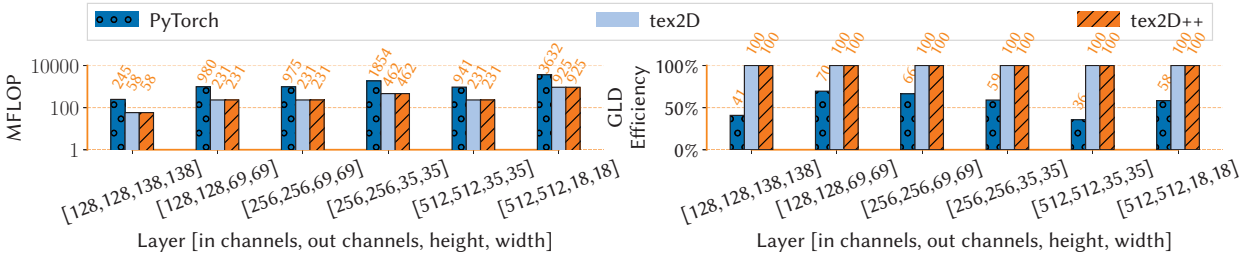

Fig. 10. *nvprof* statistics (MFLOP, Global Load Efficiency).

E. Qualitative Comparison

Finally, to conclude our evaluation, we perform a qualitative comparison with deformable convolutions [17]. YOLACT++ [17] boosts the accuracy by adding deformable convolution layers due to: (1) DCNs strengthening the network's capability to handle instances with different scales and rotations; (2) As a single shot method, YOLACT++ lacks flexible sampling. However, similar to DCNv2 [21], they place deformable layers by determining the positions manually, and adopt a policy of placing DCNs with an interval of 3 (i.e., skipping two ResNet blocks between, resulting in a total of 10 deformable layers). In contrast, with interval search, we reduce the number of DCN layers by 2 while achieving a +1.05 Mask mAP. Furthermore, to determine the placement of DCNs, prior work required formulating hand-crafted strategies

TABLE IV
DEFORMABLE OPERATION SPEEDUP WITH PYTORCH 2.1 ON 2080TI GPU.

In ch	Out ch	H	W	PyTorch (ms)	tex2D (ms)	tex2D++ (ms)	Speedup w.r. Torch
128	128	138	138	5.64	5.18	5.13	1.10×
128	128	69	69	11.89	9.17	9.16	1.30×
256	256	69	69	11.89	9.16	9.14	1.30×
256	256	35	35	26.68	21.27	21.24	1.26×
512	512	35	35	27.87	25.82	25.41	1.10×
512	512	18	18	67.41	56.27	56.14	1.20×

TABLE V
ABLATION STUDY ON OFFSETS BASED ON USING INTERVAL SEARCH WITH YOLACT++ (RESNET-101 BACKBONE.) A BOUNDARY IS PREDEFINED TO LIMIT THE MAXIMUM VALUE FOR OFFSET COORDINATES. WE APPLY REGULARIZED TRAINING AND OFFSET ROUNDING TO INTEGERS. REGULARIZED TRAINING HAS NEGLIGIBLE ACCURACY LOSS, AND IT IS POSSIBLE TO APPLY ONLY A BOUNDARY-LIMITING TECHNIQUE.

Boundary	Regularization	Round	Box mAP	Mask mAP
✓			35.38	35.35
✓	✓		35.36	35.30
✓		✓	34.52	34.37

such as skipping layers, or choosing the first/last layers [17]. Applying our technique, we discover that a hybrid approach is the most suitable, where the last few layers are replaced with DCNs and selectively placed throughout the network.

Rounding the sampling coordinates from floating point format to integers ultimately results in a significant loss of accuracy [28], [29] as observed in Table V, and without significant performance benefits. Regularized training, which adds penalties to the offset, is an alternative solution to constrain sampling coordinates. In Table V, we can see that the accuracy of regularized training results is close to the accuracy of using only the boundary method.

V. RELATED WORK

Object Detection and Instance Segmentation. A significant amount of research effort has been devoted to improving the accuracy of instance segmentation. One such effort is Mask-RCNN [35] which is a two-stage instance segmentation method. Mask-RCNN first generates regions of interest (ROI) and then classifies and segments ROIs. Follow-on work improved the accuracy of Mask-RCNN by using FPN features [36]. Such two-stage methods require re-pooling features for each ROI and additional processing, making them a poor choice when real-time throughput (i.e., 30 frames/second, i.e., fps) is required. Although real-time object detection [37]–[39] and semantic segmentation [40], [41] methods exist, Mask R-CNN was one of the fastest instance segmentation methods available, especially when processing a semantically challenging dataset such as COCO [42] (13.5 fps on 5502 px images). More recently, YOLACT++ [17] was reported to provide real-time instance segmentation, achieving competitive results on MS COCO [42]. YOLACTEdge [43] is the first competitive instance segmentation approach that runs on modest edge devices at real-time speeds (the target hardware platform is the NVIDIA Jetson AGX Xavier).

Deformable Convolution. Computer vision research has explored the use of spatially invariant features. Before CNNs, vision applications relied on custom features that were constrained by geometric transformations [44], [45]. In the context of CNNs, partial transformer networks (STNs) [46] represent the first approach that proposed learning features invariant to translation. However, the global affine transformations used in STNs cannot model complex geometric variations commonly encountered in vision tasks. Deformable convolution was first proposed by Dai et al. [10] and was extended by Zhu et al. [21], to address these issues. These Deformable Convolutional Networks (DCN) have been able to provide significant accuracy gains in vision tasks, including segmentation [17], [47], object detection [48], video restoration, super resolution [49], and other tasks [50].

GPU Texture based Optimization: Pyramidal image processing has been used to implement depth-of-field effects by employing hardware-supported filtering of pinhole images (i.e., mip-mapping) [51]–[53]. In addition, texture memory has been used to improve the performance of tree boosting on GPUs, by mapping the tree data structure to texture memory [54]. The texture memory cache has been used to store data structures from some variants of programs when adaptive code tuning is performed [55] and also to reduce redundant data loads from global memory [56]. Recently, Ukarande et al. [57] proposed Cooperative Thread Array (CTA) mapping techniques to co-locate neighboring work tokens in the same streaming multiprocessor (SM) to improve the locality of texture access patterns. In summary, previous work on texture use focused on machine learning techniques such as tree boosting [54] and optimized graphics performance [57]. In contrast, we are the first to explore deformable convolution in the context of texture-based inference on GPUs.

Accelerating Deformable Convolutional Networks (DCN). Though DCNs have been shown to be effective, they have some implementation challenges. First, additional convolution layers are required to learn the offset in an input-adaptive manner. Second, though the deformable kernel executes the same computation as a regular convolution kernel, the arbitrary sampling positions require additional software-level instructions to perform the linear interpolation. To address these inefficiencies, multiple methods have been proposed to accelerate deformable convolutions. Some popular techniques to optimize DCN on FPGA include: (i) limiting adaptive offsets to a fixed range, thus increasing the temporal locality of the input [28], [29]; (ii) constraining arbitrary offset displacements, thus reducing irregular accesses and enabling parallel accesses to on-chip memory [58]; (iii) rounding the offset displacements to integers and removing fractional bilinear interpolations [28], [29]; and (iv) using depthwise convolution to reduce the total number of Multiply-Accumulate operations (MACs) [28]. In contrast to existing FPGA-based optimizations, our work utilizes available hardware units present on just about every GPU, allowing our technique to remain applicable

to similar operators in the future. We also note that although the substitution of the depthwise convolution operator has been proposed in earlier work [28], it has only been used with manual layer placement and has not been combined with a neural search engine. Orthogonally, accelerators based on the ReRAM architecture [59] and accelerators utilizing tile dependency tables [60] have been proposed.

In contrast to previous work, which focused primarily on the design of custom accelerators, our work (DEFCON) determines the best placement of deformable operations through interval search methods and utilizes the *existing* GPU texture hardware to improve execution speed.

VI. Conclusion

This paper presents DEFCON, the first work to optimize deformable convolutions on GPU hardware. DEFCON introduces a holistic approach towards deformable convolution optimization, including better placement of operators and utilization of GPU texture hardware. Automated placement of deformable convolution layers, versus hand-tuned placement, is a key contribution of our approach. In essence, our proposed technique can run twice as fast as the state-of-the-art frameworks, providing better accuracy and using fewer deformable layers. In future work, we expect to use our approach to improve other DNN operators by leveraging texture hardware. The adoption and integration of these features to widely available machine learning frameworks should vastly improve object detection and image segmentation tasks, helping to further accelerate the artificial intelligence revolution.

References

[1] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *CoRR*, vol. abs/1404.5997, 2014. [Online]. Available: http://arxiv.org/abs/1404.5997

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds. San Diego, CA, USA: DBLP, 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 770–778. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.90

[4] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, vol. abs/1602.07360, 2016.

[5] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2017, pp. 2261–2269. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.243

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 2818–2826. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.308

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2015, pp. 1–9. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2015.7298594

[8] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2018, pp. 6848–6856. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00716

[9] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2018, pp. 4510–4520. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00474

[10] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei, "Deformable convolutional networks," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*. Venice, Italy: IEEE, Oct 2017, pp. 764–773.

[11] G. Rizos, K. Hemker, and B. Schuller, "Augment to prevent: Short-text data augmentation in deep learning for hate-speech classification," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, ser. CIKM '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 991–1000. [Online]. Available: https://doi.org/10.1145/3357384.3358040

[12] J. Munoz-Bulnes, C. Fernandez, I. Parra, D. Fernández-Llorca, and M. A. Sotelo, "Deep fully convolutional networks with random data augmentation for enhanced generalization in road detection," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, IEEE. Los Alamitos, CA, USA: IEEE Computer Society, 2017, pp. 366–371.

[13] P. Dollár, R. Appel, S. Belongie, and P. Perona, "Fast feature pyramids for object detection," *IEEE transactions on pattern analysis and machine intelligence*, vol. 36, no. 8, pp. 1532–1545, 2014.

[14] D. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the Seventh IEEE International Conference on Computer Vision*, vol. 2, 1999, pp. 1150–1157 vol.2.

[15] W. Wang, J. Dai, Z. Chen, Z. Huang, Z. Li, X. Zhu, X. Hu, T. Lu, L. Lu, H. Li, X. Wang, and Y. Qiao, "Internimage: Exploring large-scale vision foundation models with deformable convolutions," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2023, pp. 14 408–14 419.

[16] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "Yolact: Real-time instance segmentation," in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Seoul, Korea: IEEE, 2019, pp. 9156–9165.

[17] ——, "Yolact++ better real-time instance segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 2, p. 1108–1121, feb 2022. [Online]. Available: https://doi.org/10.1109/TPAMI.2020.3014297

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Canada: Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[19] Y. Wei, H. Xiao, H. Shi, Z. Jie, J. Feng, and T. S. Huang, "Revisiting dilated convolution: A simple approach for weakly- and semi-supervised semantic segmentation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[20] C. Chu, F. Chen, D. Xu, and Y. Wang, *RECOIN: A Low-Power Processing-in-ReRAM Architecture for Deformable Convolution*. New York, NY, USA: Association for Computing Machinery, 2021, p. 235–240. [Online]. Available: https://doi.org/10.1145/3453688.3461480

[21] X. Zhu, H. Hu, S. Lin, and J. Dai, "Deformable convnets v2: More deformable, better results," 2018.

[22] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/f7e0b956540676a129760a3eae309294-Paper.pdf

[23] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," 2017.

[24] X. Dai, P. Zhang, B. Wu, H. Yin, F. Sun, Y. Wang, M. Dukhan, Y. Hu, Y. Wu, Y. Jia, P. Vajda, M. Uyttendaele, and N. K. Jha, "Chamnet: Towards efficient network design through platform-aware model adaptation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[25] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[26] S. Huai, L. Zhang, D. Liu, W. Liu, and R. Subramaniam, "Zerobn: Learning compact neural networks for latency-critical edge systems," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 151–156.

[27] S. Zhang, S. Zhang, Z. Qian, J. Wu, Y. Jin, and S. Lu, "Deepslicing: Collaborative and adaptive cnn inference with low latency," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 9, pp. 2175–2187, 2021.

[28] Q. Huang, D. Wang, Y. Gao, Y. Cai, Z. Dong, B. Wu, K. Keutzer, and J. Wawrzynek, "Algorithm-hardware co-design for deformable convolution," 2020.

[29] Q. Huang, D. Wang, Z. Dong, Y. Gao, Y. Cai, T. Li, B. Wu, K. Keutzer, and J. Wawrzynek, *CoDeNet: Efficient Deployment of Input-Adaptive Object Detection on Embedded FPGAs*. New York, NY, USA: Association for Computing Machinery, 2021, p. 206–216. [Online]. Available: https://doi.org/10.1145/3431920.3439295

[30] NVIDIA, "Memory statistics - texture," 2015. [Online]. Available: https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/memorystatisticstexture.htm

[31] N. Cornelis and L. Van Gool, "Fast scale invariant feature detection and matching on programmable graphics hardware," in *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2008, pp. 1–8.

[32] D. Maung, Y. Shi, and R. Crawfis, "Procedural textures using tilings with perlin noise," in *2012 17th International Conference on Computer Games (CGAMES)*, 2012, pp. 60–65.

[33] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft coco: Common objects in context," 2015.

[34] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. Hovland, V. Taylor, and M. Hall, "Autotuning polybench benchmarks with llvm clang/polly loop optimization pragmas using bayesian optimization," in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 61–70.

[35] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *2017 IEEE International Conference on Computer Vision (ICCV)*. Venice, Italy: IEEE, 2017, pp. 2980–2988.

[36] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2018, pp. 8759–8768. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00913

[37] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*, Springer. Amsterdam, Netherlands: IEEE, 2016, pp. 21–37, to appear. [Online]. Available: http://arxiv.org/abs/1512.02325

[38] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2017, pp. 6517–6525. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.690

[39] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 779–788. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.91

[40] M. Treml, J. Arjona-Medina, T. Unterthiner, R. Durgesh, F. Friedmann, P. Schuberth, A. Mayr, M. Heusel, M. Hofmarcher, M. Widrich, B. Nessler, and S. Hochreiter, "Speeding up semantic segmentation for autonomous driving," in *NIPS 2016 Proceedings*. USA: Openreview, 12 2016.

[41] A. Paszke, A. Chaurasia, S. Kim, and E. Culurciello, "Enet: A deep neural network architecture for real-time semantic segmentation," *arXiv preprint arXiv:1606.02147*, vol. abs/1606.02147, 2016.

[42] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 740–755.

[43] H. Liu, R. A. R. Soto, F. Xiao, and Y. J. Lee, "Yolactedge: Real-time instance segmentation on the edge," in *2021 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE. Xi'an, China: IEEE, 2021, pp. 9579–9585.

[44] D. G. Lowe, "Object recognition from local scale-invariant features," in *Computer Vision, IEEE International Conference on*, vol. 2. Los Alamitos, CA, USA: IEEE Computer Society, sep 1999, p. 1150. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV.1999.790410

[45] G. Bradski, K. Konolige, V. Rabaud, and E. Rublee, "Orb: An efficient alternative to sift or surf," in *2011 IEEE International Conference on Computer Vision (ICCV 2011)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2011, pp. 2564–2571. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV.2011.6126544

[46] M. Jaderberg, K. Simonyan, A. Zisserman, and k. kavukcuoglu, "Spatial transformer networks," in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Montreal, Canada: Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/33ceb07bf4eeb3da587e268d663aba1a-Paper.pdf

[47] X. Wang, T. Kong, C. Shen, Y. Jiang, and L. Li, "Solo: Segmenting objects by locations," in *European Conference on Computer Vision*, Springer. USA: Springer, 2020, pp. 649–665.

[48] X. Zhu, W. Su, L. Lu, B. Li, X. Wang, and J. Dai, "Deformable {detr}: Deformable transformers for end-to-end object detection," in *International Conference on Learning Representations*. Vienna: Openreview, 2021. [Online]. Available: https://openreview.net/forum?id=gZ9hCDWe6ke

[49] X. Wang, K. C. K. Chan, K. Yu, C. Dong, and C. C. Loy, "Edvr: Video restoration with enhanced deformable convolutional networks," 2019.

[50] A. Siarohin, E. Sangineto, S. Lathuiliere, and N. Sebe, "Deformable gans for pose-based human image generation," in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2018, pp. 3408–3416. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2018.00359

[51] M. Kraus and M. Strengert, "Depth-of-field rendering by pyramidal image processing," in *Computer graphics forum*, vol. 26, no. 3. Wiley Online Library, 2007, pp. 645–654.

[52] R. M. Bastos, S. D. Lew, C. A. Beeson, and J. E. Demers Jr, "Depth-of-field effects using texture lookup," Dec. 13 2005, uS Patent 6,975,329.

[53] B. A. Barsky, D. R. Horn, S. A. Klein, J. A. Pang, and M. Yu, "Camera models and optical systems used in computer graphics: Part i, object-based techniques," in *Computational Science and Its Applications — ICCSA 2003*, V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 246–255.

[54] N. Birkbeck, M. Sofka, and S. K. Zhou, "Fast boosting trees for classification, pose detection, and boundary detection on a gpu," in *CVPR 2011 WORKSHOPS*, 2011, pp. 36–41.

[55] S. Muralidharan, M. Shantharam, M. Hall, M. Garland, and B. Catanzaro, "Nitro: A framework for adaptive code variant tuning," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 2014, pp. 501–512.

[56] E. H. Phillips and M. Fatica, "Implementing the himeno benchmark with cuda on gpu clusters," in *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010, pp. 1–10.

[57] A. Ukarande, S. Patidar, and R. Rangan, "Locality-aware cta scheduling for gaming applications," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 1, dec 2021. [Online]. Available: https://doi.org/10.1145/3477497

[58] S. Ahn, J.-W. Chang, and S.-J. Kang, "An efficient accelerator design methodology for deformable convolutional networks," in *2020 IEEE International Conference on Image Processing (ICIP)*, 2020, pp. 3075–3079.

[59] C. Chu, F. Chen, D. Xu, and Y. Wang, *RECOIN: A Low-Power Processing-in-ReRAM Architecture for Deformable Convolution*. New York, NY, USA: Association for Computing Machinery, 2021, p. 235–240. [Online]. Available: https://doi.org/10.1145/3453688.3461480

[60] D. Xu, C. Chu, C. Liu, Y. Wang, H. Li, X. Li, and K.-T. Cheng, "Energy-efficient accelerator design for deformable convolution networks," 2021.