Xenos: Accelerating Model Inference on Edge Devices with Hardware-Adaptive Dataflow Optimization

Runhua Zhang[1], Hongxu Jiang[1], Fangzheng Tian[1], Jinkun Geng[2], Xiaobin Li[1],
Yuhang Ma[1], Dong dong[1], Xin Li[1] and Haojie Wang[3]
[1]Beihang University, [2]Stanford University, [3]Tsinghua University
{rhzhang20, jianghx, amazingtian}@buaa.edu.cn, gjk1994@stanford.edu,
{lixiaobin, zy2106129, BY1906006}@buaa.edu.cn, alm_lx@163.com, wanghaojie@tsinghua.edu.cn

***Abstract*—Edge computing has been emerging as a popular scenario for model inference. However, the inference performance on edge devices (e.g., Multi-Core DSP, FGPA, etc.) suffers from inefficiency due to the lack of highly optimized inference frameworks. In this paper, we propose Xenos, an end-to-end framework targeting at high-performance inference for edge devices by fully exploiting the hardware specifications. Xenos incorporates two main strategies. First, Xenos leverages hardware-adaptive partition (HAP strategy), which conducts in-depth optimization on feature maps and operator parameters, to better fit memory hierarchy and improve execution parallelism. Second, Xenos restructures the inter-operator dataflow (IDR strategy), which preserves the data locality and reduces the overhead of data access. We evaluate Xenos with 6 benchmark models on two edge platforms, i.e. Multi-Core DSP (TMS320C6678) and FPGA (ZCU102). Our evaluation result proves the effectiveness of HAP and IDR, which can reduce the inference time by 19.4%–96.2% and 21.2%–84.9%, respectively. Besides, Xenos also significantly outperforms the widely-used TVM by 3.22×–51.36×. Moreover, Xenos can also be extended to a distributed solution, which we call d-Xenos. d-Xenos employs multiple edge devices to jointly conduct the inference task and achieves a speedup of 3.68×–3.78× compared with the single device.**

***Index Terms*—edge devices inference, platform heterogeneity, hardware-adaptive model partition, dataflow restructuring**

I. Introduction

Traditional neural network models (especially DNNs) are much favored in academy and industry because of their powerful capability in classification and recognition [20] [27] [13]. However, the recent years have witnessed the explosive growth of model sizes [7]. Researchers and engineers tend to add more and more heavy layers to the model, in order to strengthen the fitness capability. Such a practice may not be a problem during the model training, which can be conducted offline with multiple beefy servers. However, the huge model size really becomes a headache issue when we deploy the model to a variety of edge devices for model inference.

Edge devices are widely applied nowadays and becoming a prevalent scenario for deep learning applications [15], [17], [29]. These edge devices have heterogeneous configurations of hardware resources (e.g. memory, computation, etc) and usually require real-time responsiveness, i.e. the duration of the inference cannot take too long. Existing solutions (e.g. TVM [9]) execute model inference very inefficiently on these platforms, and fail to satisfy the responsiveness requirement.

With a deep dive into numerous typical model inference workflows, we have identified two main reasons for the inference inefficiency.

(1) **Hardware-Oblivious parallelism.** Different edge devices are usually equipped with heterogeneous computation resources and memory hierarchies. A fixed model partition scheme simply ignores the resource conditions and fails to fit the memory hierarchy and/or cannot fully utilize the computation resource. During the model inference process, parameters are frequently swapped in and out. Only a few digital signal processing (DSP) computing units (DSP cores/slices [1]) are active and undertaking the computation tasks, whereas the majority remains idle, waiting for the dependent data. Such partition schemes can waste much computation power and yield no satisfying performance.

(2) **Inefficient dataflow scheduling.** The dataflow scheduling of model inference can seriously spoil memory locality. Take the typical CNN inference as an example, after completing the inference of each layer, the computation operators output the feature maps to the shared memory region. These feature map elements will serve as the input for the next layer, and be fed to multiple DSP units for the following inference computation. However, there is a mismatch between the data layout (output from the prior layer) and the data access sequence (required by the next layer). In other words, DSP units are not reading in the sequential order as what was written previously. Therefore, while reading the feature maps, DSP units suffer from bad data locality and require *unnecessarily* much read operation, which leads to non-trivial overheads and prolongs the overall inference time.

In this paper, we develop a novel solution, Xenos, to address the two issues and boost model inference performance.

(1) **Hardware-Adaptive model partition (HAP).** Unlike the hardware-oblivious model partition adopted by existing works, Xenos fully considers the hardware information while partitioning the model, so that the partitioned scheme can be well fit into the memory hierarchy and be executed in high parallelism. More specifically, Xenos employs two main components, i.e. DSP-aware operator partition (abbr. DOP, detailed in §III-C2) and memory-aware operator split (abbr.

[1]Edge devices describe their computing units in different terminologies. Mutli-Core DSP uses the term "DSP core" whereas FPGA uses "DSP slice". We use "DSP unit" as the general term in the following description.

MOS, detailed in §III-C3), which jointly optimize the feature maps and operator parameters.

(2) **Inter-Operator dataflow restructuring (IDR).** Xenos also addresses the data locality issue during inference and restructures the dataflow between adjacent operators. While the operator(s) outputs the feature map, Xenos foresees the subsequent operator(s) which will use the output as its input, then Xenos manipulates the layout of the output feature map so that the subsequent operator(s) can read it sequentially. Take the CNN inference again as an example, after the inference of each layer, Xenos partitions the output feature map into smaller blocks, and reorganizes the layout of these data blocks, with reference to the data access pattern of the next layer. Therefore, during the inference of the next layer, the feature map can be fetched with good spatial locality. The cost of data access is much saved and the inference can be effectively accelerated.

We list our contributions as follows:

- **Framework**. Xenos is a complete end-to-end framework, including a rich operator library, an efficient memory management module, a hardware-adaptive model optimizer, etc.
- **HAP**. Xenos is integrated with HAP strategy. HAP conducts two-stage partition/split of input tensors (i.e. feature maps and operator parameters). In the first stage, the input tensors are partitioned to distribute the inference workload across multiple DSP units. In the second stage, the input tensors are further split to fit into the memory hierarchy of single DSP unit.
- **IDR**. Xenos also incorporates IDR strategy, which restructures the inter-operator dataflow and enables good data locality throughout the inference process.
- **Distributed inference**. We also extend Xenos from single-node inference to distributed inference, which we call d-Xenos. d-Xenos targets at large-volume inference workload which cannot be handled by single edge device efficiently. d-Xenos incorporates the bandwidth-optimal ring all-reduce algorithm for parameter synchronization and HAP for model parallelism, which effectively accelerates the inference computation.
- **Evaluation**. We conduct comparative experiments on different platforms showing that Xenos can reduce the inference time by 19.4%–96.2% and 21.2%–84.9%, respectively. Xenos also outperforms TVM by 3.22×–51.36×. Regarding distributed inference, d-Xenos achieves the a speedup by 3.68×–3.78× compared with the single-device baselines.

II. Background and Motivation

A. Model Inference Workflow

Despite the variety of edge computing systems, in general, they conduct the model inference according to the similar workflow, as illustrated in Figure 1. The general workflow of model inference consists of three main parts, including the image acquisition module, image preprocessing module, and inference module. The image acquisition module employs various types of image capture devices to collect original images in real-time, or requires the users to prepare the offline

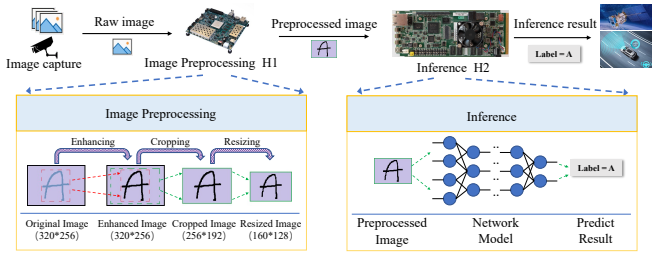


Fig. 1: The full stack of an inference workflow. During the real-time inference, the image capture device first generates images and sends them to the H1 for preprocessing, and then H1 sends preprocessed images to H2 for inference. The inference engine in H2 then makes prediction and sends the inference result to applications.

image in advance. After the image data is generated, it will be sent to the image preprocessing module. Two typical tasks will be conducted at the image preprocessing module. One is size adjustment, which aims to generate the image size required by the inference module. The other is image enhancement, which includes a series of enhancement operations such as zooming, cropping, and rotation, in order to enhance the performance for the following inference. After the size adjustment and image enhancement, the preprocessed image will be sent to the inference module, which will output the final inference result for applications.

In order to satisfy the requirement of real-time responsiveness, the workflow should be executed very fast. Among the three modules, the inference module tend to be the latency bottleneck. For instance, based on our measurement of one commercial edge computing product, the inference time contributes to more than 60% of the overall latency. The inference module thus becomes the key to improve the system responsiveness, where Xenos is applied for acceleration.

B. Hardware Resource Heterogeneity

Typical edge devices are usually equipped with multiple DSP units and limited memory. Ideally, the whole inference model should be placed into the high-level memory (e.g. L2 memory) to run efficiently. However, such a requirement is non-trivial in practice. Although much pruning effort [14], [16], [23] has been conducted to facilitate model inference on edge devices, the model size is still too large to fit in the memory hierarchy of the hardware, such a mismatch can seriously degrade the inference performance. For example, we choose MobileNet [16], a very lightweight inference model specially designed for edge hardware, and run it on our 8-core DSP platform. We run MobileNet in data parallelism without model partition, i.e. each DSP core takes charge of one MobileNet instance. Although the 8 cores participate into the inference, the inference speed of the 8 tasks is generally slow (i.e. 213.3±0.001 seconds). The reason is because the unpartitioned model cannot be held by the high-level storage (i.e. L2 memory and/or share memory). Therefore, the model parameters have to be stored in lower-level storage (i.e.DDR). During the inference, there are frequent data swaps between

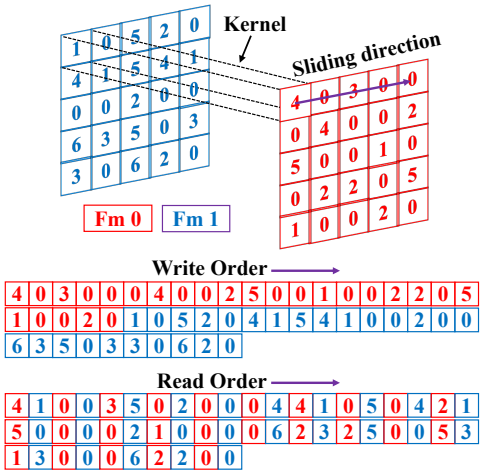


Fig. 2: Inefficient dataflow scheduling with bad locality

low-level storage and high-level storage, which causes significant overheads.

More importantly, edge devices can be various and possess heterogeneous configurations of resources, and they may run models of different shapes and sizes. As a result, there is no "one-size-fits-all" partition scheme to earn satisfying performance on every platform. Existing solutions, such as TASO [18] and PET [28], enumerate every possible partition scheme in a large search space, which is impractical in real cases. Manually-tuned scheme highly relies on the engineer's expertise, which can be very inefficient and causes a long deployment period. Therefore, we are motivated to design hardware-adaptive model partition (HAP) strategy, which takes the hardware resource information into account, and automatically generates a desirable partition scheme to both fully utilize the computation power and well fit into the memory hierarchy.

C. Data Locality in Inference Computation

While executing the inference computation, the DSP units usually read the data (i.e. feature maps) in a different order from the order in which these data are written. We use a simple example to illustrate this.

As shown in Figure 2, the DSP unit is doing the convolution computation with a depthwise separable `Conv1x1` kernel and two 5×5 matrices. The two matrices (a.k.a. two feature maps) are generated after the inference computation related to the previous operator, and they have been written as two separate data blocks into the shared memory. In other words, the data locality is maintained in each matrix block but there is no data locality across the matrix blocks.

When it comes to the current inference, the depthwise separable `Conv1x1` kernel needs to access one element from each matrix for every convolution computation (i.e. one red element and one blue element in Figure 2). Therefore, the intra-block locality does not benefit the read operation during the inference. Instead, the inter-block locality becomes the key factor affecting the performance but is completely ignored while the DSP unit outputs the two matrices. As a result, the inference process suffers from unnecessarily much cost of the read operation. Even worse, since the model is partitioned and there can be multiple depthwise separable `Conv1x1` kernels running on different DSP units, and all of them need to read the shared matrix parameters in a similar way, with spoiled data locality. In that case, all these affected DSP units have to spend much time in the data access and their computation resource cannot be efficiently utilized. To address the inefficiency, we are motivated to restructure the dataflow between adjacent operators and preserve data locality throughout the inference process.

D. Existing Drawbacks and Our Motivation

After a comprehensive investigation of the existing works, as well as a thorough review of our own practice, we can categorize the existing drawbacks into three main aspects.

First and foremost, the operator library provided by the existing frameworks (e.g. TensorRT [1] and TVM [9]) still lacks compatibility and usability. It requires time-consuming effort from the engineers. Besides, each framework has its ecosystem and is exclusive to the others. For instance, TensorRT is designed for the NVIDIA-series hardware and they are not compatible with TiC66x-series boards, or most Xilinx-series boards. A general, compatible, and easy-to-use operator library is very demanded by both the research and industry community.

Second, existing frameworks work with coarse-grained model partition and miss the useful knowledge of hardware resource conditions. Therefore, they cannot generate a highly efficient model partition scheme for the targeted edge devices. By contrast, a desirable model partition should be aware of the resource constraints (e.g. memory and computation units) on the edge hardware, achieving high parallelism while fitting the memory hierarchy.

Third, existing frameworks fail to notice the spoiled data locality during the inference dataflow, which proves to be a non-trivial factor to slow down the inference. Following the unoptimized dataflow, the inference engine experiences much unnecessary cache misses while reading the parameters for inference computation. Much performance cost can be saved with proper dataflow restructuring.

Such drawbacks motivate the development of `Xenos`, which mainly targets at the following three aspects.

(1) `Xenos` possesses a rich operator library including various fine-grained operators. Engineers can directly import these operators, or flexibly customize their operators in `Xenos`, which lowers the barrier for engineers to develop and deploy inference models on edge hardware.

(2) `Xenos` fully considers the hardware heterogeneity and uses *hardware-adaptive partition* (HAP) strategy to achieve desirable performance in various platforms. HAP partitions the input tensors across multiple DSP units and further splits them to fit into the private memory of every single DSP unit. In this way, HAP improves the inference parallelism and reduces the overheads of data fetch.

(3) `Xenos` preserves the data locality with *inter-operator dataflow restructuring* (IDR) strategy. During the inference

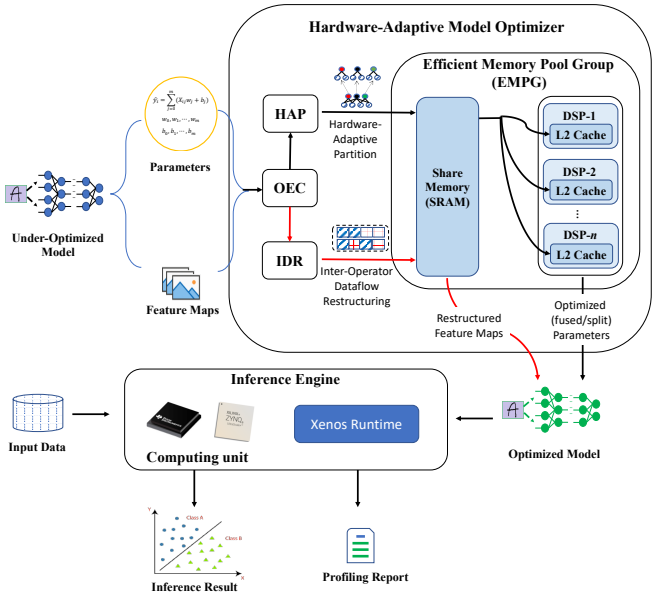


Fig. 3: The architecture of Xenos

computation, Xenos restructures the writing order of intermediate parameters output from the operators, so as to match the reading order of the subsequent operators. In this way, Xenos maintains good data locality throughout the whole inference process, thus avoiding costly cache misses and accelerating the workflow.

III. Design of Xenos

A. Architecture and Workflow

Figure 3 illustrates the architecture of Xenos. Xenos consists of two modules, a *hardware-adaptive model optimizer* and an *inference engine*. Given an NN model to be optimized, the model optimizer will first conduct an offline optimization with reference to the edge hardware constraints, and then output an optimized model, which is equivalent to the original model. The inference engine is fed with the input data and the optimized model, then it employs multiple DSP units to execute the inference task. The inference engine outputs the inference result to the applications, as well as a profiling report for engineers to conduct further performance analysis and optimization.

As the key module in Xenos, the model optimizer includes four main components, namely, Efficient Memory Pool Group (EMPG), Operator Elastic Combiner (OEC), Hardware-Adaptive Partitioner (HAP), and Inter-operator Dataflow Restructurer (IDR).

The optimizer optimizes the data organization in two aspects. HAP reorganizes the input tensors, which include the tensors of feature maps and operator parameters, to improve inference parallelism and better fit into the memory hierarchy organized by EMPG. IDR restructures the dataflow to achieve better data locality, thus the feature maps and operator parameters can be efficiently accessed by the inference engine. Both HAP and IDR are supported by OEC (detailed in §III-C).

EMPG is the main storage for Xenos, which organizes L2 memory and SRAM-based share memory in a hierarchical way. Since the under-optimized model may generate too large parameters and feature maps to well fit in the memory hierarchy, Xenos relies on the other three components (OEC, HAP and IDR) to optimize the data storage and dataflow to achieve memory efficiency.

OEC provides multiple techniques to support HAP and IDR. In general, OEC reorganize the parameter storage by performing operator linking , operator partition, operator split, as well as operator fusion. Operator linking further reduces the redundant memory access by considering the complex dataflow which cannot be optimized by operator fusion (detailed in §III-C1). DSP-aware operator partition (DOP) distributes input tensors across multiple DSP units for high parallelism (detailed in §III-C2). Memory-aware operator split (MOS) is used to split input tensors when the parameters (after operator partition) cannot be fit into the private L2 memory of each DSP unit (detailed in §III-C3).

HAP generates the partition scheme of input tensors according to the DSP units. More specifically, the partition scheme enables each DSP unit to be assigned with a partitioned feature map (and the corresponding operator parameters) in load balance, and these DSP units can execute in high parallelism with the partitioned feature maps (detailed in §III-C2) and operator parameters (detailed in §III-C3).

IDR defines the data layout of the feature maps when they are output to the share memory. As mentioned in §II-C, the inference engine can suffer from lots of compulsory cache misses without considering the data layout of the feature maps. Targeting at this, IDR restructures the dataflow between adjacent operators linked by OEC, so that the feature maps output from the previous operator can be accessed by the subsequent operator in good locality (detailed in §III-D).

The following subsections give the design detail of each component in Xenos.

B. Efficient Memory Pool Group

During the inference process, the frequent swap of feature map data (with variable sizes) can lead to memory fragmentation problems. To mitigate the fragmentation and better explore the cache benefit, we develop an efficient *memory reuse mechanism* in EMPG. The reuse mechanism follows the typical producer-consumer model. Before the launch of the inference process, EMPG prepares in advance a pool of memory chunks, with typical model layer sizes. While allocating memory for a fresh model layer, EMPG first checks the available memory chunks with the same size (or even larger size), then allocates a suitable chunk to hold the model layer. Meanwhile, the chunk is marked as *in-use*. When the lifetime of the model layer is terminated, the chunk is marked again as *available* and recycled in the memory pool, so that EMPG can use it for the other incoming model layers. Besides, for very small data, EMPG will also do batching with them and fit them into one memory chunk. In this way, the fragmentation is further reduced and data locality is much preserved.

C. Operator Elastic Combination

A neural network (NN) model is typically represented as a computation graph in existing NN frameworks like PyTorch [24] or TensorFlow [8].With reference to the computation graph, it is straightforward to generate a complete but unoptimized inference code using existing operator library. However, the unoptimized code involves redundant or inefficient memory access and fail to achieve balanced inference computation among DSP units, leading to poor performance. Operator fusion, which is a widely used optimization technique, can significantly reduce the redundant memory access, but still misses many optimization opportunities. Targeting at this, `Xenos` proposes OEC to further improve memory efficiency and reduce inference time. OEC consists of three main components, including operator linking (§III-C1), operator partition (§III-C2) and operator split (§III-C3). In general, `Xenos` implements IDR strategy with operator linking and implements HAP with operator partition and operator split.

1) Operator linking: Operator fusion are commonly used optimizations in existing NN frameworks. Typical operator fusion mainly fuses element-wise operators with their adjacent operators, so that the extra memory read and write between operators can be eliminated thus improving the memory efficiency. Besides operator fusion, IDR gives `Xenos` the opportunities to further reduce memory overhead, which we call *operator linking*.

As shown in Figure 5 (a), fusing operators `Conv1x1`, `Bn`, `Bias` and `Relu` into one operator `CBR` is a typical optimization. However, simply fusing operator `CBR` and `AvgPooling` will usually not get better memory performance due to the complex dataflow on the input feature map of `AvgPooling` (a 2-D sliding window). As shown in Figure 4, the dataflow of `conv`'s output mismatches with the dataflow required by `AvgPooling`'s input, leading to extra data shuffle overhead even when these two operators are fused.

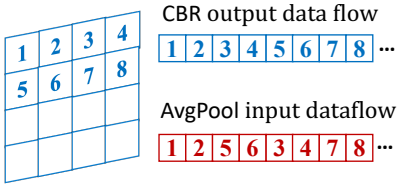


Fig. 4: Dataflow mismatch between `conv`'s output and `AvgPooling`'s input.

Based on IDR, `Xenos` is able to reorganize the dataflow. As shown in Figure 5 (b), operator `CBR` and operator `AvgPooling` will be marked as linkable operators in OEC. In IDR, the feature map dataflow of linked operator will be reorganized to eliminate the data shuffle overhead, and the detail will be illustrated in §III-D.

Algorithm 1 shows the algorithm of performing fusion and linking on a given computation graph in `Xenos`. `Xenos` will first try to fuse all the fusable operators, which is similar with existing NN frameworks. Then for the unfusable operators, `Xenos` will link them if the size of their parameters are

Algorithm 1 Operator fusion and linking

```
   Input:
   inGraph:The input computation graph
1: for Op ∈ inGraph's operators do
2:     if Op.predecessors.size == 1 then
3:         PrevOp = Op's predecessor
4:         if PrevOp and Op are fusable then
5:             Fuse PrevOp and Op in inGraph
6:         if LINKABLE(PrevOp, Op) then
7:             Link PrevOp and Op in inGraph
8: function LINKABLE(PrevOp, Op)
9:     ParamSize = PrevOp.parameters.size + Op.parameter.size
10:    return ParamSize ≤ threshold
```

less than a certain size $threshold$, which is set as the L2 memory size in `Xenos`. The dataflow of linked operators will be optimized in IDR.

2) DSP-aware Operator Partition (DOP): `Xenos` circumvents the memory constraints and places all parameters into L2 memory. Still, it is insufficient for high parallelism, so `Xenos` partitions the operators across multiple DSP units to share the inference workload.

Algorithm 2 exemplifies the DSP-aware operator partition (DOP) algorithm. DOP considers partitioning the feature map in three dimensions, namely, the input feature map height (inH), the input feature map width (inW), and the output feature map channel ($outC$). Here we dismiss the input-channel-based partition for simplicity since we need to perform extra reduction if we partition along input channel which will introduce computation overhead. Considering that the feature maps are stored in the share memory with size of 4 MB, which is far beyond the input channel size in typical CNN model, we usually do not have to partition along input channel.

DOP prioritizes $outC$-based partition (line 7-13) due to its less complexity: DOP simply distributes the kernel parameters to different DSP units, and these kernel parameters will be placed into the L2 memory of DSP units. All DSP units can access the feature map located in the share memory. On the other hand, inH-based scheme (line 14-21) and inW-based scheme (line 22-28) partition the remaining feature map after $outC$-based partition, and they usually require special handling of the boundary rows/columns. Only if the kernels cannot be evenly distributed across DSP units, DOP will seek further partition by inH/inW. If imbalance still exists after the triple partition, DOP will randomly assign the remaining feature map (workload) to different DSP units (line 29-34).

While the operator partition procedure distributes the inference workload across multiple DSP units, the single DSP unit may still fail to conduct the inference work efficiently, because the operator parameters assigned to it is too large to fit into the L2 memory (e.g. the CNN model has very large-sized kernels). To address that, OEC needs to do further split of the operator parameters according to the memory resource of the DSP unit. We explain in §III-C3.

3) Memory-aware operator split (MOS): OEC uses dimension-based memory-aware operator split to cut the large-

Algorithm 2 DSP-aware Operator Partition (DOP)

Input:
inH: The input height of the feature map.
inW: The input width of the feature map.
outC: The number of output channels (operator parameters)
K$[1 \ldots outC]$**:** The list of kernels
F$[1 \ldots inH]$ $[1 \ldots inW]$ $[1 \ldots inC]$**:** The input feature map, size of $inH \times inW \times inC$.
P: The expected parallelism degree, i.e. the number of DSP units available on the edge device.

upon RECEIVING (K, F) **do**
▷ While receiving a feature map F and the target operator (kernels) K, we output a partitioned $scheme$. We assume P is global constant
  $scheme$ = FMP-ALGO(K, F, P)
  Forward $scheme$ to Inference Engine
function DOP-ALGO(K, F, P)
▷ $scheme$ is encoded as a dictionary, with key as the
▷ core-id and value as the partitioned operators and
▷ feature map (fmp) assigned to this core
  for $i \leftarrow 1$ to P **do** ▷ Initialization of $scheme$
    $scheme[i]$={op : {}, fmp : {}}
  ▷ Partition by output channel $outC$
  $sliceByC = \lfloor outC/P \rfloor$
  for $i \leftarrow 1$ to P **do**
  ▷ $K[k_1 : k_2]$ represents a slice in K, ranging from k_1 to k_2
    $k_1 = sliceByC \times (i-1) + 1$
    $k_2 = k_1 + sliceByC$
    $scheme[i][op]$.append($K[k_1{:}k_2]$)
    $scheme[i][fmp]$.append(F)
  $restC = sliceByC \times P + 1$
  ▷ Partition by input height inH
  $K' = K[restC{:}outC]$
  $sliceByH = \lfloor inH/Pn \rfloor$
  for $i \leftarrow 1$ to P **do**
    $h_1 = sliceByH \times (i-1) + 1$
    $h_2 = h_1 + sliceByH$
    $scheme[i][op]$.append(K')
    $scheme[i][fmp]$.append($F[h_1{:}h_2][:][:]$)
  $restH = sliceByH \times P + 1$
  ▷ Partition by input width inW
  $sliceByW = \lfloor inW/P \rfloor$
  $F' = F[restH : outH][:][:]$
  for $i \leftarrow 1$ to P **do**
    $w_1 = sliceByW \times (i-1) + 1$
    $w_2 = w_1 + sliceByW$
    $scheme[i][op]$.append(K')
    $scheme[i][fmp]$.append($F'[:][w_1{:}w_2][:]$)
  ▷ Randomly assign the remaining feature map
  $restW = sliceByW \times P + 1$
  for $i \leftarrow restW$ to inW **do**
    $F'' = F[:][i][:]$ ▷ Pick the ith column of F
    r=RAND()%W ▷ Assign to a random DSP unit
    $scheme[r][op]$.append(K')
    $scheme[r][fmp]$.append(F'')
  return $scheme$

sized operator parameters into smaller chunks so that they can be placed into the private L2 memory of the DSP unit. In this way, the parameter fetch can become highly effecient and the inference time can be reduced.

During the operator splitting process, OEC follows a certain priority for each dimension when performing parameter splitting, to guarantee that minimum computation overhead is introduced after splitting. Taking the popular CNN model as an example, which have four dimensions for parameters, i.e., output channel (K), input channel (C), kernel height (R), and kernel width (S). Splitting at K dimension will not introduce any extra computation, while splitting at other three dimensions requires additional reduction operation to aggregate the results on these dimensions, which introduces extra computation overhead. Thus, OEC will first try to split the parameters at K dimension, and then C, R and S if only splitting K dimension is not enough to fit the parameter in L2 memory.

Equation 1 gives an example of output-channel-based splitting. Since the large-sized parameters (W and B) can not be put into the L2 memory, OEC performs fine-grained split of the operator: W is split into W_1 and W_2, and B is split into B_1 and B_2. After that, parameters can be distributed into the L2 memory of two different DSP units. Equation y_0 and y_1 can be jointly executed on two DSP units in parallel, or on one DSP unit one by one. The output $y_1(x_i)$ and $y_2(x_i)$ are automatically joined together using IDR, without performing any data layout transformation operators. Other types of splitting work in a similar way.

$$\begin{aligned} y(x_i) &= Wx_i + B \\ &\Downarrow \\ y_1(x_i) &= W_1 x_i + B_1 \\ y_2(x_i) &= W_2 x_i + B_2 \end{aligned} \tag{1}$$

4) Examplar optimization process: Equipped with the fusion, link and split capality of OEC, a concrete user case of operator optimization is illustrated in Figure 5.

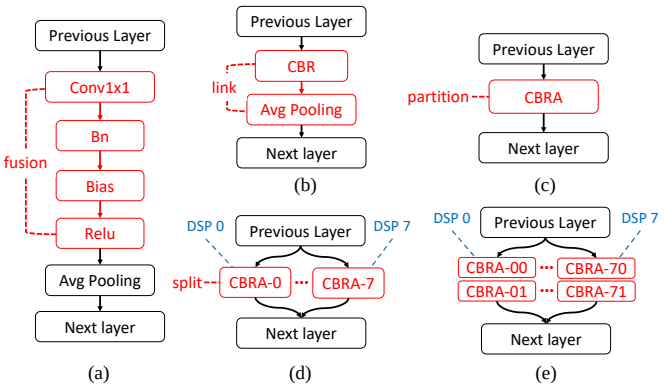


Fig. 5: Illustration of Operator Elastic Combination.

Figure 5 gives an example of how OEC optimizes a part of the computation graph of MobileNet [16]. OEC first performs operator fusion on operator `Conv1x1` (convolution), `Bn` (batch normalization), `Bias`, and `Relu`, and generates a fused operator `CBR`, then links `CBR` and `AvgPooling` together to form the operator `CBRA`. Operator `CBRA` will be first split into eight partition to fit each DSP unit's L2 memory. Considering that the parameter size of each partition ($1024 \times 1024 \times 1 \times 1 \times 4\text{B}/8 = 512\text{KB}$ for convolution kernel,

and $1024 \times 7 \times 7 \times 4\text{B}/8 = 24.5\text{KB}$ for bias) is larger than L2 memory size (512KB), thus it will be further split into two partitions and executed one by one on each DSP unit.

D. Inter-Operator Dataflow Restructuring

While HAP-based OEC enables high-parallel execution by reorganizing the parameters and feature maps, the dataflow during the execution is under-optimized and spoils the data locality. To address that, `Xenos` employs IDR to conduct *inter-operator dataflow restructuring*.

IDR takes into account output/input sequence of adjacent operators. In other words, after one operator completes the inference computation, it should output the feature map to the share memory in the desirable layout, so that the subsequent operators can read them with good data locality.

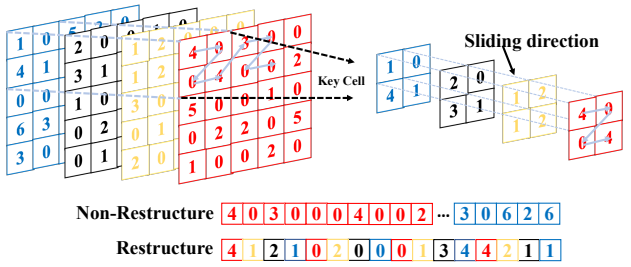


Fig. 6: Dataflow restructuring

We exemplify IDR with a linked operator (`Conv1x1` + `AvgPooling2x2`) in Figure 6. The input feature map is generated by the previous operator which has 4 output channels, so the input feature map consists of four matrices (marked with different colors in Figure 6).

Without IDR optimization, the four matrices are placed into the memory one by one in a row-based manner, with the red one placed first, and the blue one placed last. However, during the inference computation, the fused operator needs to read 1×1 feature map from each matrix every time, and then computes the average on every 2×2 square after the convolution. As magnified on the right side of Figure 6, the dataflow goes through the four matrices every time (`Conv1x1` computation). On each matrix, it follows the zigzag pattern (`AvgPooling` computation), leading to the restructured dataflow. Comparing the non-restructured and restructured pattern, the former one is oblivious to the data access, and each data access suffers from compulsory cache misses. By contrast, the latter one completely matches the data access pattern during the inference, so it maximizes the data locality and data can be fetched in much higher efficiency.

To preserve the data locality, the previous operator is required to be aware of the data access pattern of the subsequent operator when it outputs the feature map. With such awareness, the inference engine can restruture the data layout following the access pattern, instead of simply outputting the matrices one by one. To attain this, IDR analyzes the adjacent operators linked by OEC, and pre-encodes the data access pattern of each operator in the optimized model. After the optimized model is fed into the inference engine, the inference engine gains the knowledge of the desirable feature map layout for each operator. During runtime, once an operator completes the inference computation and is about to output the feature map, the inference engine foresees the data access pattern of the subsequent operator and writes the feature map according to the pattern. Such a restructuring method involves data redundancy in standard convolution, since it replicates some parameters of the feature map to avoid the subsequent operator from looking back. However, the memory sacrifice proves to be worthwhile, because the performance benefit brought by the restructuring outweighes the additional memory cost and the inference workflow is effectively accelerated (shown in §VI-B).

IV. Distributed Inference

As the model size grows increasingly large, we envision that the inference workload can soon go beyond the capacity of single edge devices. Recent models, such as ResNet-101 (60.2M) [13], Bert (340M∼481000M) [10] and GPT-3 [25](175000M), can hardly be used for single-device inference, making distributed inference become a necessity. Driven by these current and future scenarios, we extend `Xenos` to leverage multiple devices for joint inference computation, and we call the distributed version of `Xenos` as `d-Xenos`.

Algorithm 3 Enumerating Partition Schemes

Input:
dset:The dimension set to be partitioned

```
bestShm, bestTime = ∅, +∞
for shm ∈ PERMUTATION(dset) do
    execTime = PROFILING(shm)
    if execTime < bestTime then
        bestShm, bestTime = shm, execTime
return bestShm
```

The key idea of `d-Xenos` is to incorporate both model-parallel computation and parameter synchronization among multiple devices, to cut down the overall inference time. However, we note that, compared with `Xenos`, `d-Xenos` may not achieve optimal inference time if it still follows the priorities of DOP (§III-C2) to partition the feature map. Taking convolution operator as an example, recall that convolution's DOP prioritizes *outC*-based partition over *inH-/inW*-based partition. The reason is that *outC*-based partition can fully leverage the shared memory across different DSP units and avoid padding overheads due to *inH-/inW*-based partition. However, when it comes to `d-Xenos`, different edge devices do not share memory, and it can hardly tell whether or not *outC*-based partition can outperform *inH-/inW*-based partition. Therefore, `d-Xenos` needs to balance two types of overheads, i.e., the communication overheads due to parameter synchronization and the computation overheads due to imbalanced partition . To address this problem, `d-Xenos` uses a tree based traversal algorithm to determine the partition strategies.

The general enumeration algorithm is described in Algorithm 3. `d-Xenos` enumerates every possible combination of partition schemes towards dimensions appeared in feature

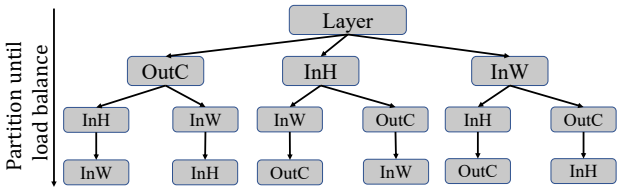


Fig. 7: Enumerating partition schemes in `d-Xenos`

maps or operator parameters involved in the inference, which can reach $d!$ different combinations at most if there are totally d dimensions. For example, there are three dimensions in matrix multiplication (m, n, and k, where k is shared by both feature map and parameter) and seven dimensions in convolution ($batch$, inC, inH, inW, $outC$, r and s, where inC is shared by both feature map and parameter). However, as we discussed in §III-C2 that `Xenos` only partitions along certain dimensions, `d-Xenos` also chooses these dimensions to partition because we empirically notice that the other dimension-based partition (e.g. inC-based partition for convolution operator) cannot achieve better performance in the distributed setting either. Therefore, `d-Xenos` still focuses on the partition schemes along *inH*, *inW* and *outC* when partitioning a convolution, as shown in Figure 7. Then `d-Xenos` chooses the best partition strategy among them according to the profiling result of their execution time. Due to such search process is one-off for each model when deploying on a specific platform, the search cost is acceptable.

V. Implementation

To use `Xenos`, users need to provide a computation graph for the inference model, along with the hardware-specific information for a edge device, e.g., memory hierarchy, number of DSP units, etc., so that `Xenos` can perform hardware-adaptive optimizations. The optimized model will be generated as an executable supported by `Xenos` runtime, including an efficient operator library, and the middlewares for memory management and scalable communication. Currently, `Xenos` can support different edge devices including Multi-Core DSP hardware [2], Xilinx U-series hardware [4], Xilinx Zynq-series hardware [5], etc.

A. Operator Library

Existing frameworks tend to offer API interfaces with high-level languages (e.g. Python) and mask the low-level implementation details to programmers. However, while working with edge hardware platforms, programmers are required to use the low-level operators and programming languages (e.g. C/C++ and Assembly languages) to implement their own functions.

Our goal is to provide ordinary staff with a rich operator library, which is efficient and can be easily used to implement their high-level functions on these edge hardware platforms. Therefore, we begin by implementing with low-level languages (C/C++ and Assembly) various types of operators (e.g. standard convolution with different KSize and different Stride, depthwise separable convolution, MaxPooling/Global Pooling, Concat, etc.). Meanwhile, we keeps iterating over the operator library for usability and compatibility.The basic operators provided by `Xenos` can be summarized as Table I. Nowadays different ML frameworks tend to provide hundreds of operators, nearly all typical operators can be finally categorized into one type of basic operator listed in Table I. For example, the convolution operator with different sizes (e.g. 1x1, 3x3, etc) are implemented with the same `x.conv` operator API, but with different parameters. We also implement additional split/fuse operators in `Xenos`, which have not been supported by other ML systems. Besides, programmers can customize their own operator and encapsulate their own libraries by combining these basic operators in `Xenos`.

TABLE I: The main tensor operators used in Xenos

Operator	Description
`x.add`	Element-wise Addition
`x.mul`	Element-wise Multiplication
`x.mac`	Multiply Accumulate
`x.conv`	Convolution (kernel size, stride, padding, etc)
`x.matmul`	Matrix Multiplication
`x.gampool`	Global / Average / Max Pooling
`x.transpose`	Matrix Transpose
`x.concat`	Concatenation of Multiple Tensors
`x.split`	Split a Tensor into Multiple Tensors
`x.cbr`	Fused `Conv-Bn-Relu` operator
`x.cbrm`	Linked `CBR-MaxPooling` operator
`x.cbra`	Linked `CBR-AvgPooling` operator

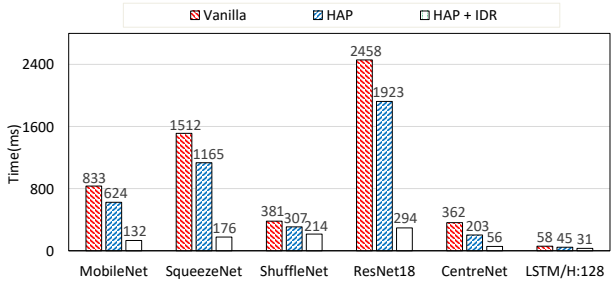
B. Scalable Communication Middleware

Image data usually requires a series of preprocessing operations before inference. Normally, the preprocessing module and the inference module are not executed on the same device. Therefore, we need to build a bridge between them. For the sake of modularity and scalability, we encapsulate the communication primitive into an independent middleware. The middleware is compatible with both the SRIO protocol (for embedded applications) technology and the conventional Ethernet protocol. Meanwhile, our communication middleware is integrated with pipeline and batch transmission mechanisms, to achieve high throughput performance. Additionally, we also customized the efficient packing/unpacking function for the sake of lower real-time latency.

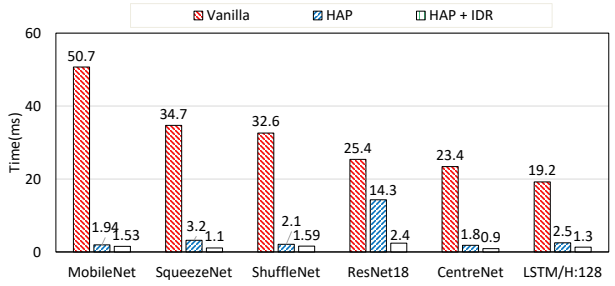
VI. Experimental Evaluation

A. Experiment Setting

Testbeds: We employ two testbeds for evaluation, namely, (1) a Multi-Core DSP device, which is equipped with 2 TMS320C6678-type nodes and a high-speed image collector, which are directly connected via SRIO; (2) the ZCU102-type FPGA device, with code generated by Intel High-Level Synthesis (HLS) Compiler. We use TMS320C6678 and ZCU102 to refer to them for simplicity.



(a) TMS320C6678



(b) ZCU102

Fig. 8: Inference time comparison

Benchmarks: We choose 6 typical models as the benchmarks, i.e. MobileNet, SqueezeNet, ShuffleNet, ResNet18, CentreNet and LSTM.

Baselines: We first conduct an ablation study with `Xenos`: we compare the complete `Xenos` solution (HAP + IDR) with the two baselines, from which we show the benefit of `Xenos`'s two strategies. As shown in Figure 8, one baseline only adopts the does not involve HAP and IDR, which we call it Vanilla baseline. The other baseline only adopts HAP optimization, which we call it HAP baseline. Then, we compare `Xenos` with TVM [9], running both of them with the same benchmarks on ZCU102. Besides, we also compare `Xenos` with a GPU baseline, where we run the same benchmarks with PyTorch on an NVIDIA RTX 3090 GPU.

Metrics: We mainly study the inference time cost and the hardware resource cost in our evaluation. As for the inference time cost, we run each inference workload for 1000 times and report the average value. As for the resource cost, we compare the cost of L2, SRAM, and DDR on TMS320C6678; and we compare the cost of DSP [2], FF [3] and LUT [4] on ZCU102.

B. Inference Time Comparison

TMS320C6678. The comparative results of inference time on TMS320C6678 are shown in Figure 8(a). From the figure we can see, compared with Vanilla, HAP reduce the inference time by 19.4%-43.9%. The advantage of HAP over Vanilla comes from two main aspects. First, the smart operator fusion in `Xenos` can effectively merge redundant computation in the inference process (described in §III-C4). Second, DOP partitions the inference workload in a balanced way across DSP units and MOS helps to better fit the memory hierarchy. These two components combine to bring the acceleration.

[2]DSP (refer to DSP slice) is composed of high-speed multiplier circuits to perform high-speed multiply accumulate operations in FPGA.

[3]FF (Flip Flop) is a storage unit that can only store 1 binary bit and can be used as a memory element for sequential logic circuits.

[4]LUT (Look-up Table) is a module that can realize the function of the combinational logic circuit through the truth table stored in the memory unit.

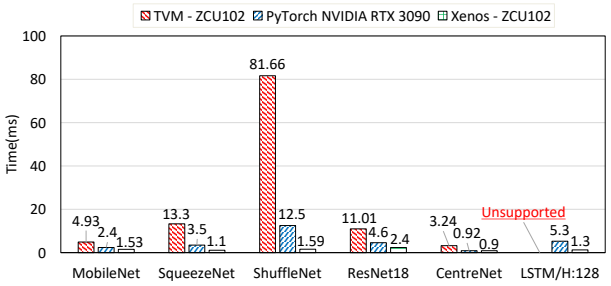


Fig. 9: Inference time comparison with TVM and PyTorch

We further evaluate the performance benefit of IDR by comparing the HAP baseline and the full `Xenos` solution. We can see that the IDR further reduces the inference time by 30.3%-84.9%. The experimental results prove the effectiveness of IDR, which improves the data locality for data access and thus reduce the inference time cost.

ZCU102. The comparative results on ZCU102 (Figure 8(b)) also demonstrate the performance benefit of HAP and IDR. Compared with the Vanilla baseline, HAP can reduce the inference time by 90.6%-96.2%. Compared to the HAP baseline, IDR further reduces the inference time by 21.1%-83.3%.

Comparing Figure 8(a) and Figure 8(b), we can easily notice that IDR contributes more inference time reduction on TMS320C6678, whereas HAP contributes more on ZCU102. The reasons are explained in two main aspects.

1) IDR is more effective on TMS320C6678 than ZCU102. This is because a large number of LUT resources are used on ZCU102 to implement data mapping, therefore, the memory access efficiency has already been improved a lot even without IDR. By contrast, TMS320C6678 is not equipped with such a utility, so the memory access efficiency can be seriously damaged when the inference workload breaks the data locality. IDR, however, helps to preserve the data locality with dataflow restructuring and becomes the main contributor.

2) HAP is more effective on ZCU102 than TMS320C6678. This is because ZCU102 has much more DSP units than TMS320C6678. While TMS320C6678 only has 8 DSP units, ZCU102 can allocate thousands of DSP units to participate in the computation. Therefore, the management of model partition and parallel execution becomes more essential to the efficiency of ZCU102. Since the Vanilla baseline is not equipped with a proper partition scheme, it fails to exploits the abundant computation resource. In contrast, HAP works with the optimized operators (by OEC) and achieves high utilization of DSP computation resource, which can significantly reduce the inference time.

Comparing with Other Baselines. To further demonstrate the outperformance of `Xenos`, we compare `Xenos` with two other baselines. First, we run TVM on ZCU102 with the same inference models [3]. Second, we also report the inference performance in a GPU environment: we use PyTorch and run

the same models with an NVIDIA RTX 3090 GPU. Figure 9 shows Xenos also significantly outperforms the other two baselines. To be more specific, Xenos achieves 1.02×–1.87× speedup compared with the GPU baseline, and it achieves 3.22×–51.36× speedup compared with the TVM baseline. Across all the benchmark models, except LSTM [5], TVM falls far behind Xenos running on the same hardware, because TVM fails to fully exploit the hardware information during the inference process.

C. Micro-Benchmark on Typical Operators

TABLE II: Speedup for typical operators. CBR is the abbrevation for Conv-Bn-Relu.

Operators	OEC Optimization	Speedup
CBR $3 \times 224 \times 224$ $3 \times 3 \times 3 \times 32$	Operator Fusion	3.1×
CBR $128 \times 28 \times 28$ $1 \times 1 \times 128 \times 256$	Operator Fusion	2.4×
CBR-MaxPooling $24 \times 224 \times 224$ $3 \times 3 \times 3 \times 224$	Operator Linking	3.3×
CBR-AvgPooling $1024 \times 7 \times 7$ $1 \times 1 \times 1024 \times 1024$	Operator Linking	2.3×
FullyConnected $1 \times 1 \times 1536$ $1 \times 1 \times 1536 \times 1000$	Operator Split	2.25×

Whereas §VI-B demonstrates the effectiveness of Xenos towards the complete model inferene process, in this section, we solicit typical operators, and study Xenos's acceleration for them. To do that, we run a micro-bench test on TMS320C6678 with typical operators in CNN, and Table II shows significant speedup brought by Xenos.

D. Resource Cost Comparison

1) Comparison on TMS320C6678: Figure 10 presents the resource cost comparison when running MobileNet on TMS320C6678. The other 5 models show similar trends, so we omit them due to space limitation and include the full result in our technical report [30]. Since the difference in storage cost is mainly due to HAP rather than IDR, we omit the intermediate baseline and only compare Vanilla and the full Xenos solution (HAP +IDR).

From Figure 10 we can see that, Vanilla's DDR overheads are contributed by both the feature maps and the parameters (e.g. weights). During the inference process, both feature map size and parameter size can go beyond the storage capacity of SRAM. As shown in Figure 10(c), the burst of DDR during the initial ∼22ms is mainly due to the output feature map. The input feature map was occupying the SRAM at that time, and SRAM becomes insufficient to hold both the input feature map and output feature map. At the end of ∼163ms, the burst is due to the large-sized convolution layer of MobileNet, whose parameter size reaches more than 4MB and cannot be placed in either L2 memory or SRAM-based share memory.

2) Comparison on ZCU102: Figure 11(a) and Figure 11(b) illustrate the resource cost of MobileNet and SqueezeNet on ZCU102 respectively. The other 4 models shows similar trend as MobileNet and we include the full results in our technical report [30].

From Figure 11(a), We can see that both HAP and IDR help to reduce the resource cost on ZCU102.

HAP enables higher parallelism and improves resource utilization efficiency. The computation of some inference operations are completed faster, so that the corresponding resource (e.g. DSP units) can be freed and reused for the other inference operations, instead of allocating more resource. Compared with Vanillia, HAP can complete the same workload with shorter time but with less resource cost.

IDR also contributes to the reduction of the resources cost. Because of the restructured dataflow, the data access becomes faster, which in turn reduces the idle time of resources (e.g. DSP units do not need to wait for the input feature maps for long time). Therefore, the resource utilization is improved and the inference engine does not need to request unnecessarily more resource from the edge device.

However, we notice that SqueezeNet implies some inconsistent trend with the other models regarding the resource cost. More specifically, HAP does not help to reduce the cost of DSP units (even leads to a slight increase). We finally identify that SqueezeNet's network structure can be easily paralleled, which makes it benefit from the default optimization from HLS. In other words, HLS also integrates some optimization mechanism during the code generation, which already helps Vanilla to achieve a high utilization of DSP units, leaving little optimization room for HAP. As a result, HAP bring no evident performance benefit towards SqueezeNet.

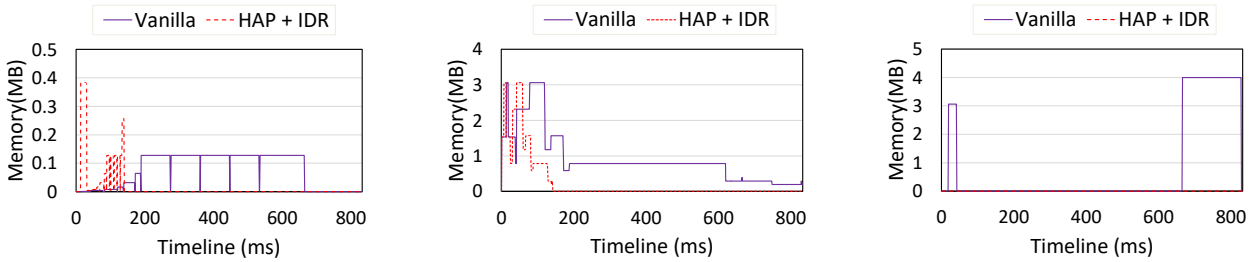
E. Distributed Xenos (d-Xenos)

We employ 4 TMS320C6678 devices to evaluate d-Xenos. We equip d-Xenos with the ring all-reduce primitive, which proves to be bandwidth optimal in parameter synchronization [22]. We run two models (MobileNet and ResNet18) in d-Xenos. The evaluation result is shown as Figure 12. We summarize two main takeaways:

(1) The ring all-reduce synchronization is more efficient than the traditional parameter-server (PS) synchronization. When equipped with PS-based synchronization, the inference time can become even worse compared with the single-device inference because parameter synchronization dominates the overheads. By contrasts, when equipped with ring all-reduce synchronization, d-Xenos achieves distinct speedup.

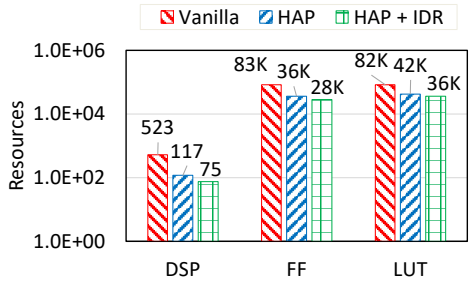
(2) There is no "one-size-fits-all" partition scheme to achieve optimal inference time. Therefore, none of the single-mode partition schemes (i.e. *inH*-based. *inW*-based and *outC*-based) achieves the optimal performance. By contrast, our profiling-driven method chooses the most appropriate partition

[5]TVM cannot run LSTM on edge hardware, because TVM is relies on the development kit provided by Xilinx to run the inference models, but the development kit does not support running LSTM on ZCU102

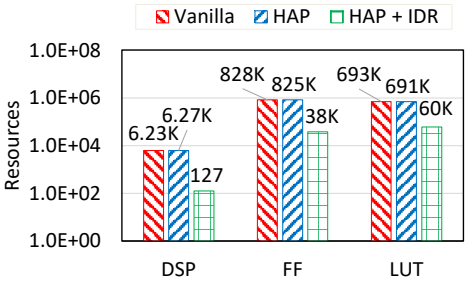


(a) MobileNet L2 Cost (b) MobileNet SRAM Cost (c) MobileNet DDR Cost

Fig. 10: Resources cost comparison on TMS320C6678 (MobileNet)



(a) MobileNet Resources Cost



(b) SqueezeNet Resources Cost

Fig. 11: Resources cost comparison on ZCU102

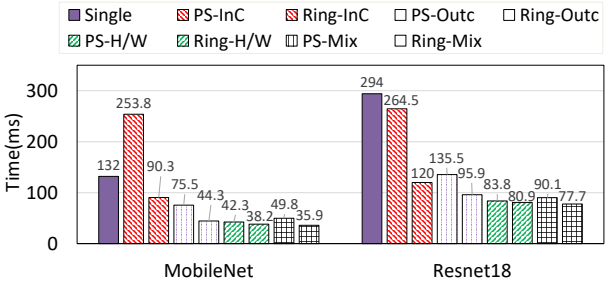


Fig. 12: `d-Xenos` Inference time

schemes for the related operators, thus making the hybrid partition scheme (Ring-Mix) yield the most performance advantage for `d-Xenos`.

VII. Further Discussion: Blackbox vs. Whitebox Optimization

In the existing literature, TASO [18] and PET [28] also conduct operator split/fusion to accelerate machine learning training/inference. However, such optimizations (which we call them *blackbox optimization*) are completely different from `Xenos`, because they only manipulate the typical operators without reference to the external information of the hardware. By contrast, `Xenos` is *whitebox optimization* and it fully utilizes the context knowledge (i.e. hardware resource conditions, inter-operator dataflow, etc) for deeper optimization (HAP and IDR).

Knowledgeable readers may wonder, *would TASO and PET be able to search out a comparable (or even better) solution than `Xenos`, if they were equipped with the same operator library?* Theoretically, if TASO or PET were equipped with a proper cost function, and were given infinite time and computation power, their search algorithm (i.e. essentially depth-first-search, DFS) should output a scheme, which can yield an optimal memory layout and model partition. However, such cases are impractical due to two reasons. First of all, the cost function, though claimed to be customized, is hard to define for memory layout. TASO and PET simply use the execution time as the cost function in their cases, and there are no guidelines on defining a cost function targets at memory layout, so it cannot preserve the data locality as IDR does. More importantly, the search algorithms of TASO and PET are simply based on the enumeration of all candidates with DFS, without considering any prior knowledge of the hardware platform. Therefore, its search space can easily blow up. Even after pruning, the search-based optimization can only work with a very small number of operators. To be more specific, TASO can only execute its search-based optimization with at most 4 operators, whereas PET can work with at most 5 operators in practice. Such deficiencies seriously constrain the scope of the blackbox-like optimizations including TASO and PET.

`Xenos`, on the other hand, leverages the prior knowledge–including the resource information of the hardware and the dataflow information of the operators–as its guideline, so its strategies (HAP and IDR) avoid the explosive search space and can find a near-optimal scheme efficiently. However, the *blackbox* and *whitebox optimization* are not mutually exclusive, and the automatic search algorithm of TASO/PET can also be inherited by `Xenos` to discover more optimized schemes. We leave the integration of *blackbox optimization* as our future work.

VIII. RELATED WORK

Graph-level optimization Most existing machine learning frameworks represent neural network models as computation graph, and perform graph-level optimization to optimize machine learning tasks. TensorFlow [8] with XLA [6], TensorRT [1], MetaFlow [19] and DNNFusion [21] optimize the computation graph by transformation rules designed by domain experts. TASO [18] and PET [28] further adopts super-optimization technique for automatically graph-level optimization, which can significantly enlarge the optimization space while reducing human efforts. Lacking of efficient operator library on edge devices, simply applying the above optimization techniques cannot achieve promising performance. Thus, `Xenos` uses architecture-aware approaches, including OEC, HAP and IDR to perform in-depth optimizations to fully utilize the computation resource.

Code generation Halide [26] presents a domain specific programming language for tensor programs and proposes 'computation+schedule' model to decouple the optimization stages. Inspired by Halide, TVM [9] uses a similar optimization model and proposes a learning-based approach to automatically search through the hyper parameters for a given schedule to generate highly efficient code. Ansor [31] and FlexTensor [32] explore different schedule strategies to further enlarge the searching space for TVM.

Inference on edge hardware Compared with the cloud environment, edge-based inference usually yields much better latency performance, and has gained momentum in recent years. AOFL parallelization [33] improves edge-based inference efficiency by fusing convolutional layers and dynamically selecting the optimal parallelism according to the availability of computing resources and network conditions. `Xenos`, by contrast, considers more about the resource conditions on the edge device, including both memory and computation, and its IDR strategy brings deeper optimization to the operator by restructuring the dataflow. Mema [11] enhances the scheduling policy to run multiple inference jobs without additional edge resources. While `Xenos` currently focuses on accelerating single inference job, we believe its strategies (HAP and IDR) can become compatible to work in multiple-job scenario with some adaption. [12] proposes a collaborative solution, which employs multiple edge devices to jointly undertake one big inference task. `Xenos`, on the other hand, mainly targets at the inference work executed on single device. Nevertheless, it would be an interesting direction to integrate `Xenos` optimization into a distributed framework, and we leave it as our future work.

IX. CONCLUSION AND FUTURE WORK

We present `Xenos`, which incorporates hardware-adaptive partition (HAP) and inter-operator dataflow restructuring (IDR) strategies to accelerate edge-based inference. We conduct comprehensive experiments with 6 benchmarks on two typical platforms, Multi-Core DSP(TMS320C6678) and FPGA(ZCU102). Our evaluation results demonstrate the effectiveness of HAP and IDR, and also prove `Xenos`'s outperformance versus TVM while executing edge-based inference under the same setting. We also develop a primary distributed version (`d-Xenos`), which can achieve a speedup of 3.68×–3.78× compared with the single device.

The future development of `Xenos` will mainly include two aspects: (1) We will continue to optimize the distributed version of `Xenos` to improve the efficiency of joint inference across multiple edge devices. (2) We will incorporate the blackbox-like and whitebox-like approaches to enhance the optimization strategies of `Xenos`.

REFERENCES

[1] Nvidia tensorrt. https://developer.nvidia.com/tensorrt.

[2] TI TMS320C6678. https://www.ti.com/product/TMS320C6678.

[3] Tvm-deploy. http://tvm.apache.org/docs/how_to/deploy/vitis_ai.html?highlight=vitis.

[4] Xilinx U series. https://www.xilinx.com/products/boards-and-kits/alveo.html.

[5] Xilinx ZCU102. https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html.

[6] Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla.

[7] Aaron and et al. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv:1806.03377*, 2018.

[8] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[9] Tianqi Chen and et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI'18*.

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[11] Jeroen Galjaard, Bart Cox, Amirmasoud Ghiassi, Lydia Y Chen, and Robert Birke. Mema: Fast inference of multiple deep models. In *2021 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 281–286. IEEE, 2021.

[12] Ramyad Hadidi, Jiashen Cao, Micheal S Ryoo, and Hyesoon Kim. Collaborative execution of deep neural networks on internet of things devices. *arXiv preprint arXiv:1901.02537*, 2019.

[13] Kaiming He and et al. Deep residual learning for image recognition. In *CVPR'16*.

[14] Yihui He and et al. Amc: Automl for model compression and acceleration on mobile devices. In *ECCV'18*.

[15] Jacob Hochstetler and et al. Embedded deep learning for vehicular edge computing. In *SEC'18*.

[16] Andrew G Howard and et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[17] Yutao Huang and et al. When deep learning meets edge computing. In *ICNP'17*.

[18] Zhihao Jia and et al. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *the SOSP'19*.

[19] Zhihao Jia, James Thomas, Todd Warzawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing dnn computation with relaxed graph substitutions. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[20] Krizhevsky and et al. Imagenet classification with deep convolutional neural networks. *NeuIPS'12*.

[21] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. Dnnfusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 883–898, 2021.

[22] Pitch Patarasuk and Xin Yuan. Bandwidth optimal all-reduce algorithms for clusters of workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[23] Antonio Polino and et al. Model compression via distillation and quantization. *arXiv:1802.05668*, 2018.

[24] Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org, 2017.

[25] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

[26] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, 2013.

[27] Karen Simonyan and et al. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.

[28] Haojie Wang and et al. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *the OSDI'21*.

[29] Haoxin Wang and et al. User preference based energy-aware mobile ar system with edge computing. In *INFOCOMM'20*.

[30] Runhua Zhang and et al. Xenos: Accelerating Model Inference on Edge Devices with Hardware-Adaptive Dataflow Optimization. https://pacman.cs.tsinghua.edu.cn/~whj/pubs/xenos.pdf, 2022.

[31] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.

[32] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[33] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 195–208, 2019.