CoDL: Efficient CPU-GPU Co-execution for Deep Learning Inference on Mobile Devices

Fucheng Jia
Central South University
Microsoft Research
fuchengjia@csu.edu.cn

Deyu Zhang*
Central South University
zdy876@csu.edu.cn

Ting Cao
Microsoft Research
Ting.Cao@microsoft.com

Shiqi Jiang
Microsoft Research
shijiang@microsoft.com

Yunxin Liu
Institute for AI Industry Research (AIR), Tsinghua University
liuyunxin@air.tsinghua.edu.cn

Ju Ren*
Tsinghua University
renju@tsinghua.edu.cn

Yaoxue Zhang
Tsinghua University
Central South University
zhangyx@tsinghua.edu.cn

ABSTRACT

Concurrent inference execution on heterogeneous processors is critical to improve the performance of increasingly heavy deep learning (DL) models. However, available inference frameworks can only use one processor at a time, or hardly achieve speedup by concurrent execution compared to using one processor. This is due to the challenges to 1) reduce data sharing overhead, and 2) properly partition each operator between processors.

By solving the challenges, we propose CoDL, a concurrent DL inference framework for the CPU and GPU on mobile devices. It can fully utilize the heterogeneous processors to accelerate each operator of a model. It integrates two novel techniques: 1) *hybrid-type-friendly data sharing*, which allows each processor to use its efficient data type for inference. To reduce data sharing overhead, we also propose *hybrid-dimension partitioning* and *operator chain* methods; 2) *non-linearity- and concurrency-aware latency prediction*, which can direct proper operator partitioning by building an extremely light-weight but accurate latency predictor for different processors.

Based on the two techniques, we build the end-to-end CoDL inference framework, and evaluate it on different DL models. The results show up to 4.93× speedup and 62.3% energy saving compared with the state-of-the-art concurrent execution system.

*Corresponding author.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing**; • **Computing methodologies → Concurrent algorithms**.

KEYWORDS

CPU-GPU Co-execution, Deep Learning Inference, Mobile Devices

ACM Reference Format:
Fucheng Jia, Deyu Zhang, Ting Cao, Shiqi Jiang, Yunxin Liu, Ju Ren, and Yaoxue Zhang. 2022. CoDL: Efficient CPU-GPU Co-execution for Deep Learning Inference on Mobile Devices. In *The 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22), June 25-July 1, 2022, Portland, OR, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3498361.3538932

1 INTRODUCTION

Deep Learning (DL) is now the pillar for diverse mobile applications. On-device inference is gaining momentum compared to the on-cloud counterpart, due to the advantages in privacy protection, internet resilience, and low cloud-operation overhead. However, current on-device inference can only achieve acceptable responsiveness for some simple models, but not for others. For example, YOLO [23] for object detection takes over two hundred milliseconds to run on major mobile processors, *i.e.,* mobile CPUs or GPUs. To improve responsiveness, a nature thought is whether it is beneficial to concurrently utilize heterogeneous processors on a mobile device.

Fortunately, we identify that the specific design of mobile system-on-chips (SoCs) provides this opportunity for two causes: 1) comparable CPU and GPU performance. Different from server GPUs which run orders-of-magnitude faster than the CPUs, mobile CPUs and GPUs have similar performance for DL inference [15, 29]. They can therefore run side by side; 2) a unified memory. Different from server machines which usually have separate memories for the CPU and GPU, the mobile CPU and GPU use a unified memory [12]. It can avoid data copying between different memories.

However, current inference frameworks can only use one processor at a time, hindered by two major challenges of concurrent execution (co-execution). The first is *how to reduce data sharing overhead*. Albeit with a unified memory, considerable overhead is still needed to ensure the coherence of shared data. For example, to run an operator of a model concurrently on the CPU and GPU, the output of the last operator needs to be shared between the two processors. This leads to processor synchronization, data mapping, as well as potential data transformation if different data types are used by different processors. Without proper strategy, the data sharing overhead can easily over weigh the gain from concurrency. The second is *how to fairly partition each operator* of a model between processors. Online measurements for different partitioning candidates are infeasible. A latency predictor which is light-weight and accurate, and more importantly, aware of all possible overhead introduced by concurrency is required.

To the best of our knowledge, existing works cannot well address the above challenges. μlayer [15] and Optic [29] enable the co-execution of CPU and GPU on mobile devices. However, for the first challenge, they use the same data type (*i.e.,* buffer type) for both the CPU and GPU to simplify data sharing. As we will show in Sec. 2, this design makes CPU+GPU co-execution even slower than the GPU alone, due to the use of inefficient data type for the GPU. For the second challenge, to direct operator partitioning, they model the operator latency by linear regression on the number of computations (*i.e.,* FLOPs). Though this model is light-weight, the prediction accuracy is very poor ($<$ 10%). The reason is that the FLOPs-based predictor cannot capture the real latency behavior. There are also latency predictors [6, 31, 33] that use complex black-box machine learning models to capture latency behavior and achieve high accuracy. However, these models suffer from big running overhead. For example, the model size of nn-Meter [33] for a convolution operator is >800 MB, too heavy to run on mobile devices for real-time prediction. Besides, none of these predictors considers the concurrency-related overhead.

To address the challenges, we propose CoDL, a CPU+GPU concurrent DL inference framework that can fully utilize the heterogeneous processors to accelerate a model. The design of CoDL stems from two key findings. 1) Different processors prefer different data type for optimal performance. For example, we observe using the image type on Adreno GPU can achieve 3.5× speedup compared to the buffer type for convolution. It is necessary to use the efficient type for each processor for co-execution; 2) To make the latency predictor both accurate and light-weight, it is imperative to incorporate platform features into the model, rather than a pure black-box learning.

Based on the two findings, CoDL integrates two new techniques. 1) *Hybrid-type-friendly data sharing*. It allows the heterogeneous processors to use different data types for inference. Then, to reduce data sharing overhead, we propose *hybrid-dimension partitioning* and *operator chain* methods. Hybrid-dimension partitioning can select the optimal partitioning dimension for each operator shape to achieve the tradeoff between data sharing overhead and processor utilization. Operator chain makes sure the operators on a chain only require local data to execute rather than the shared data from the other processor, to avoid data sharing overhead. 2) *Non-linearity- and concurrency-aware latency prediction*. CoDL can conduct online

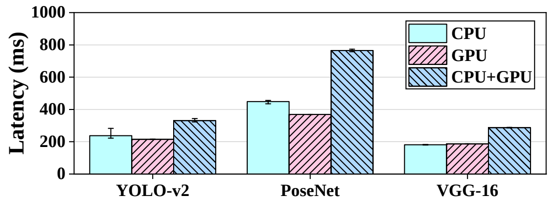


Figure 1: The latency of three DL models executed on CPU, GPU and CPU+GPU, in MACE.

and fair operator partitioning by building a light-weight but accurate latency predictor. Our insight is that the high complexity of other learned latency predictors is to capture the non-linear latency response caused by different algorithms and execution blocks. We therefore analytically formulate the calculation of blocks for each algorithm to extract the non-linearity. Only the linear component is learned by an extremely light-model (~500 B v.s. 800 MB of nn-Meter) linear-regression model. Besides, our predictor is the first to consider all the concurrency-related overhead.

Based on the two techniques, we build the end-to-end CoDL framework. For a given model, the best data partitioning and sharing plan for each operator can be worked out based on the predictor. CoDL then coordinates the processors to execute the plan. We implement CoDL based on the state-of-the-art (SOTA) mobile inference framework MACE [19]. The experiments on commercial off-the-shelf (COTS) mobile devices, including Snapdragon 855, 865 and 888, and Kirin 990, demonstrate that CoDL can achieve on-average 3.43× speedup and 62.3% energy saving in comparison with the SOTA co-execution system. By taking the non-linear features into account, our predictor achieves 86.21% and 82.69% accuracy on predicting the runtime latency of operators on CPU and GPU, respectively, with low inference overhead ($<$ 1 ms for an operator). Furthermore, with one-time collected data samples (6000 samples with running time less than 1.5 hours), the predictor can be trained in an on-device manner with latency ranging from 1 to 2 seconds.

The main contributions are as follows:

- In-depth analysis on the performance bottleneck of concurrent CPU+GPU execution;
- Propose hybrid-type-friendly data sharing between CPU and GPU, which utilizes hybrid-dimension partitioning and operator chain to reduce sharing overhead.
- Propose the extremely light-weight but accurate non-linearity- and concurrency-aware latency prediction.
- Implement the end-to-end CoDL framework and demonstrate that it outperforms the state-of-the-art solutions.

2 MOTIVATION AND ANALYSIS

To direct CoDL design, we first explore the performance bottlenecks of processor co-execution by analyzing the SOTA inference systems. We evaluate μLayer [15] and MACE [19] as the SOTA co-execution and single-processor execution system, respectively. This section shows the results on Snapdragon 855 for example. Fig. 1 compares their inference latency for different models. Surprisingly, the CPU+GPU co-execution of μLayer is slower than the

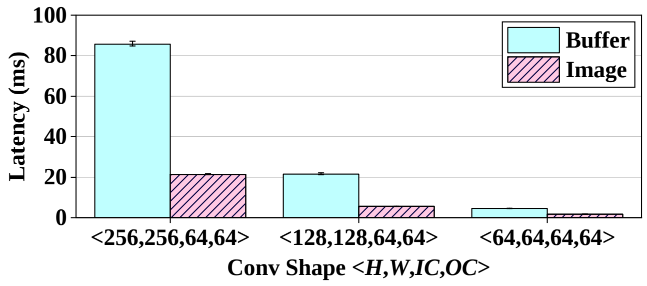


Figure 2: Latency comparison of using buffer and image data type for 3×3 convolution. The height (*H*) and width (*W*) of the input feature maps range from 64 to 256; the input channels (*IC*) and output channels (*OC*) are both 64.

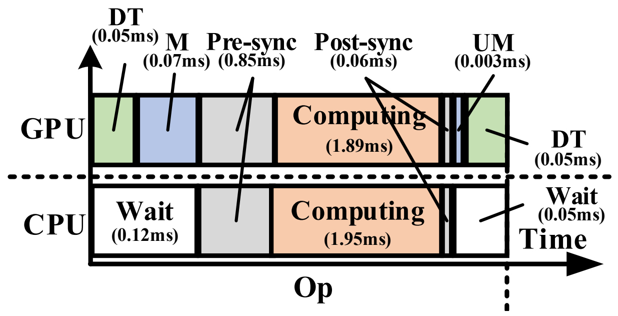


Figure 3: Latency components of co-execution of the CPU and GPU. The op is a 3×3 convolution with shape <112,112,32,64>. The partition ratio is 0.5. DT: Data Transformation, M: Mapping, UM: unmapping.

CPU or GPU alone of MACE. For PoseNet [35], it even leads to ~ 2× slowdown.

We analyze the systems in depth and expose the performance issues of the current co-execution system: 1) the use of unified data type for different processors; 2) the neglect of data sharing overhead; and 3) the unbalanced workload partitioning. We will next discuss these issues in detail, as well as the implications for CoDL design.

A unified data type is not efficient for heterogeneous processor co-execution. To simplify data sharing, current co-execution systems use a common data type for different processors. For example, the *buffer* type is supported by both the CPU and GPU, and thus used by μLayer. Buffer type organizes data into contiguous and pointer-accessible chunks. However, we identify that the *image* type can be much more efficient than buffer on Adreno GPUs. Image type organizes data into multi-dimensional chunks to facilitate rendering tasks. It leverages the fast L1 texture cache on GPU to accelerate the data access. Fig. 2 illustrates the performance difference of using image and buffer type for 3×3 convolution with different input shapes. The latency is reduced by 3.5× using the image type, compared to the buffer type.

Therefore, to fully utilize each heterogeneous processor, the corresponding efficient data type should be used.

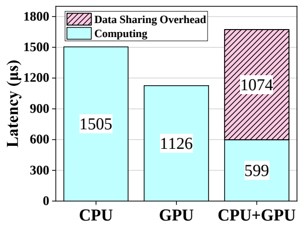


Figure 4: The latency of computing and data sharing for a 1×1 convolution with shape <52, 52, 256, 128>.

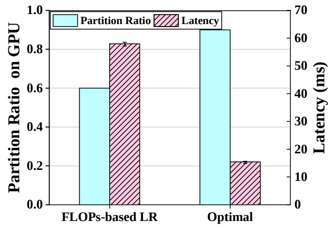


Figure 5: The partitioning ratio and latency of a 3×3 convolution with shape <240,320,64,128>.

The data sharing overhead for co-execution is not negligible, especially for small operators. Co-execution introduces data sharing overhead to make sure data coherence between processors. It is overlooked by current co-execution systems. Fig. 3 demonstrates the process and latency components of an operator co-execution on the CPU and GPU. Assuming the operator input is generated by the last operator on the GPU and now shared between the CPU and GPU for co-execution. On top of operator computation, the extra overhead is from 1) data transformation, if different data type is used; 2) data mapping, which maps the input to the CPU address space; 3) synchronization, which informs the other processor the completion of mapping (pre-sync) or computation (post-sync).; and 4) data unmapping, which unmaps the output from the CPU address space.

We identify that this overhead is not negligible. Particularly for small operators, it easily becomes the dominant overhead and offsets the gain brought by the CPU-GPU co-execution. Fig. 4 demonstrates an example. Given that the co-execution reduces the execution latency from 1126μs to 599μs, the data sharing overhead contributes 1074 μs, leading to a 1.5× slowdown.

Therefore, the co-execution system should aim to reduce the overhead and concurrently execute the operator only when the gain outperforms the overhead.

Balanced workload partitioning for co-execution requires a light-weight and accurate latency predictor. Current co-execution systems usually use predicted latency by a light-weight model to direct the workload partitioning between processors. For instance, μLayer uses a FLOPs-based linear model to predict the latency. The light-weight latency model is suitable for the online prediction. However, it is too inaccurate (< 10% according to our measurements, details in Sec. 7).

The inaccurate latency prediction in turn leads to the poor inference performance, due to the unbalanced workload. Fig. 5 demonstrates an operator of a popular model as an example. The FLOPs-based predictor leads to a 4× slowdown by allocating 60% of the operator on the GPU, given that the optimal partitioning ratio is 90%.

The reason for the inaccurate prediction is that the latency is not simply a linear relationship with FLOPs, but greatly impacted by the platform features such as the algorithm implementation and data block size [26, 33]. As shown in Fig. 6, the latency shows a non-linear response as the FLOPs increases for the GPU and CPU.

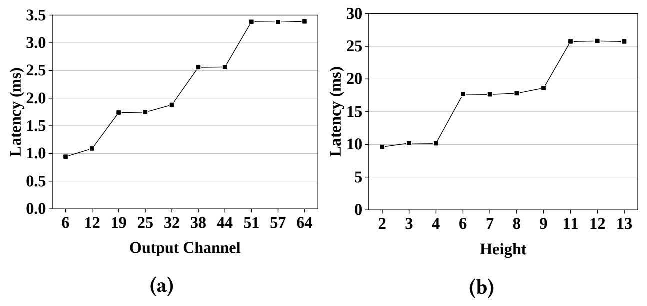


Figure 6: Non-linear latency response as FLOPs increases (by increasing channel and height) on the (a) GPU and (b) CPU.

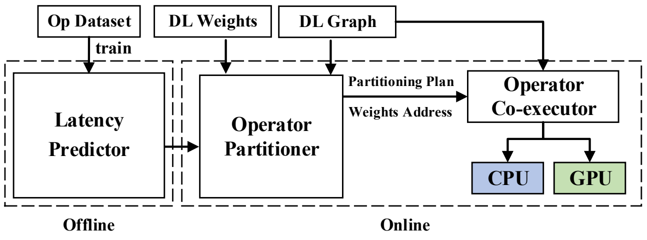


Figure 7: System architecture and workflow of CoDL.

There are also works aiming for the accurate latency prediction, *e.g.,* nn-Meter [33], which use black-box machine learning methods to learn the latency response based on a number of operator hyperparameters. However, suffering from the lack of knowledge on the underlying platform features, the black-box methods obtain satisfactory accuracy at the cost of large model size (*e.g.,* over 800 MB for convolution by nn-Meter) and infeasible execution time (*e.g.,* more than 80 ms on a PC by nn-Meter). It is unpractical to be deployed on mobile devices.

Therefore, a latency predictor that can incorporate platform features and thus be both light-weight and accurate is required for the co-execution system.

3 CODL OVERVIEW

To achieve the optimal performance of co-execution on heterogeneous processors, we raise three design principles: 1) fully utilizing the computing capability of each processor; 2) minimizing the extra overhead caused by the data sharing, *i.e.,* the data transformation, mapping and synchronization; 3) best partitioning and balancing the workload among heterogeneous processors.

Guided by the principles, we design CoDL. It runs in two phases, *i.e.,* the offline phase and online phase, as shown in Fig. 7.

In the offline phase, CoDL designs a light-weight but effective latency predictor to direct the operator partitioning in the online phase. The predictor can achieve both light-weight and effective because 1) it considers all the data sharing overhead including data transformation, mapping, and synchronization (Sec. 5.1); 2) it analytically formulates the non-linear latency response caused by the platform features, and only needs to learn the latency of the basic execution unit by an extremely light-weight linear regression model for each kernel implementation (Sec. 5.2).

The online phase consists of two modules, *i.e.,* the operator partitioner and the operator co-executor.

The role of operator partitioner is to work out the optimal operator partitioning plan for the input DL model. Based on the latency predictor, it employs two techniques, *i.e.,* hybrid-dimension partitioning and operator chain, to fulfill this role. The partitioner first finds the best partitioning dimension (either height or output channel) and ratio (*e.g.,* 0.1, 0.2, ...) for each operator as the base plan, by the hybrid-dimension partitioning technique (Sec. 4.1). According to the base plan, it searches for the operators to chain up by the operator chain technique, so that the operators on a chain require no shared data (Sec. 4.2). The final partitioning plan from the partitioner is the set of found chains, each with the chained operators and chain settings *i.e.,* partitioning ratio and dimension. With the plan, the model weights are pre-arranged for the GPU and CPU to avoid re-transformation for each inference invocation.

The operator co-executor coordinates the synchronized execution of operators according to the partitioning plan, using processor-friendly data type for different processors. As shown in Fig. 8, at the beginning of a chain, CoDL first shares the data between the CPU and GPU. Assuming the GPU-friendly type as the default data type, CoDL transforms the partitioned input feature map from the GPU-friendly type (*e.g.,* the image type on Adreno GPU) to the CPU-friendly type. Then, the CPU and GPU execute all operators in one chain concurrently. At the end of a chain, the GPU transforms the data generated on the CPU back to GPU-friendly type, and combine with GPU output together as the input of the following chain.

Next, we discuss the key techniques proposed in CoDL, *hybrid-type friendly data sharing* and *the non-linearity- and concurrency-aware latency prediction* in detail.

4 HYBRID-TYPE FRIENDLY DATA SHARING

CoDL supports the use of efficient data type for each processor. However, the challenge is that it further increases data sharing overhead. This section introduces the two data sharing optimization techniques of CoDL, *i.e.,* hybrid-dimension partitioning and operator chain. They accelerate operator co-execution by achieving the tradeoff between data sharing and computation overhead.

4.1 Hybrid-dimension partitioning

Performance impact analysis. It is possible to partition the tensors of an operator along different dimensions, including OC, H and W, for co-execution. Fig. 9 illustrates the partitioning along OC and H for example. Different dimensions lead to different performance impact.

Firstly, partitioning dimension impacts data sharing overhead. As discussed in Sec. 2, there is considerable overhead to ensure the coherence of shared data between processors. It is therefore important to reduce the amount of shared data. Model weights are consistent during inference. They can be pre-allocated on each processor, and no need to be shared dynamically. However, the input feature map for each operator needs to be shared dynamically, since it is generated by the last operator during inference.

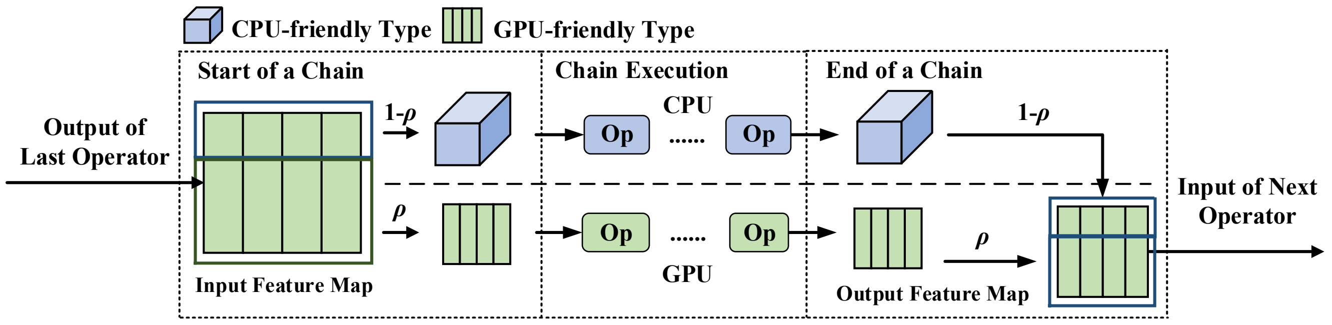


Figure 8: Co-execution of an operator chain.

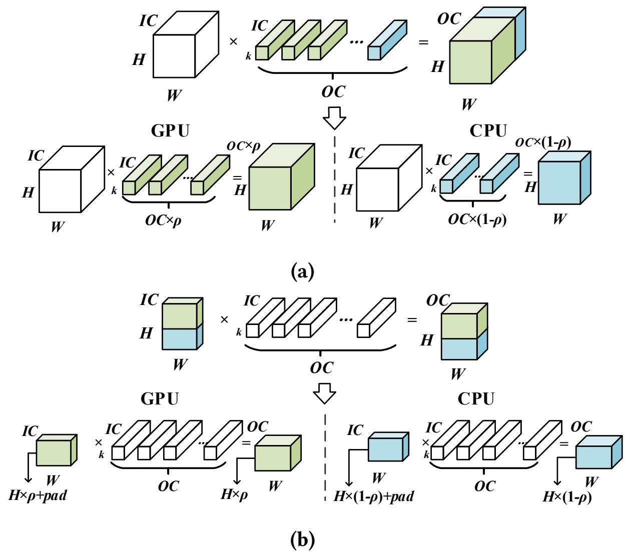


Figure 9: Data partitioning for co-execution along (a) OC and (b) H. ρ is the partitioning ratio for the GPU, and $1 - \rho$ for the CPU.

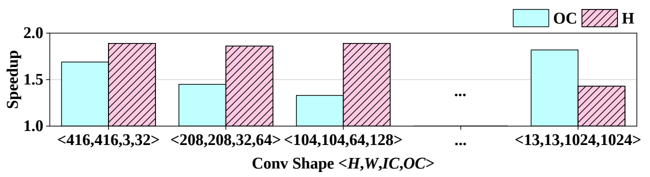


Figure 10: CPU+GPU speedup (over CPU_only) comparison between partitioning on *OC* and *H*, for convolution in YOLO model. The partitioning ratio is 0.5.

Therefore, considering data sharing overhead, partitioning on *H* dimension is preferable than *OC*. As Fig. 9 shows, for *OC* partitioning, the whole input feature map is shared between the CPU and GPU. By comparison, only a partial feature map (and also the padding data for filter processing) is shared for *H* partitioning.

Secondly, partitioning dimension impacts processor utilization. Though partitioning on *H* has less data sharing overhead, we find that it does not always mean less running time. We take several typical convolution operators from YOLO [23] to show the performance difference in Fig. 10. *H* partitioning does achieve higher speedup than *OC* for the first three convolutions. For example, the <104, 104, 64, 128> convolution has 1.89× speedup by co-execution (compared to CPU_only) in *H* partitioning, while only 1.33× in *OC* partitioning. Conversely, the <13, 13, 1024, 1024> convolution achieves higher speedup in *OC* partitioning than *H* (1.89× vs 1.43× respectively).

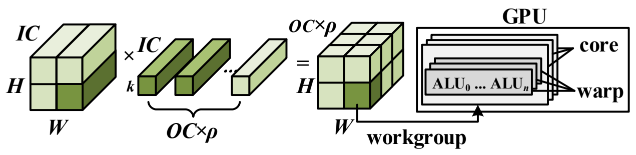


Figure 11: The output is divided into workgroups. Each work group is scheduled to run on a GPU core in the unit of warps.

The reason is that *H* partitioning may reduce processor utilization compared to *OC*, depending on the operator shape. To utilize the inter-core and intra-core parallelism of a processor, the tensors of an operator need to be divided into blocks and scheduled to run on different cores. Take the GPU as an example shown in Fig. 11. A block named as a *work group* is scheduled to run on a GPU core. To efficiently utilize the many ALUs in a core, the basic execution unit named as a *warp* executes the same instruction for a number of threads (*e.g.,* 64 or 128) simultaneously. Therefore, a work group should provide enough threads to fill up the warps. Or there will be idle ALUs.

The size of a work group on the *OC* dimension is normally small (*e.g.,* 4) to avoid stressing cache by many filters. When *H* and *W* are small, *e.g.,* <13, 13> in Fig. 10, there are not enough threads to fill up the warps, leading to lower GPU utilization and performance.

Thirdly, partitioning dimension impacts data access overhead. For faster data access, the partitioning dimension should be consistent with the tensor layout and make sure that the shared data is continuously stored in memory. For example, the $[H, W, IC]$ layout for feature maps prefers partitioning on *H* rather than *W*.

Determining the partitioning dimension. Based on the analysis above, the partitioning dimension should be determined for each operator shape, according to its impact on data sharing overhead and processor utilization. We thus propose hybrid-dimension partitioning for CoDL. It integrates the impacted factors into the

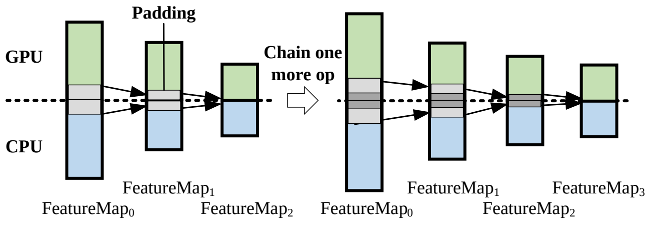


Figure 12: To chain one more operator requires more padding for each feature map in a chain.

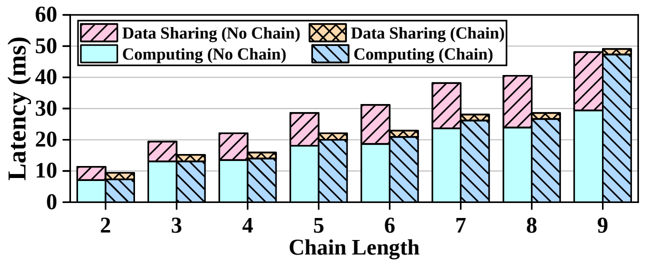


Figure 13: Latency comparison of chain vs no-chain on data sharing and computation, for operators in YOLO model. The partitioning ratio is 0.5.

latency predictor (detailed in Sec. 5). Given the input operator settings and a partitioning plan in dimension and ratio ρ, the predictor outputs the total co-execution latency (including both data sharing and computing overhead). Based on the predicted latency, CoDL can rapidly evaluate different partitioning plans online, and find the optimal partitioning dimension and ratio for each operator of a DL model. Specially, CoDL predicts the total latency of a given operator across various partitioning plans (*i.e.,* the H and OC dimension, and the ratio from 0 to 1). The determined partitioning dimension is the one that achieves the least predicted latency.

4.2 Operator chain

Instead of sharing data after each operator, we propose the operator chain technique to reduce the number of operators requiring shared data. As Fig. 8 shows, the data only needs to be shared at the beginning and ending of a chain. Other operators in a chain use locally generated data but no data from the other processor.

Performance impact analysis. The challenge for operator chain is how to rapidly decide which operators to chain up in a DL model. Improperly chaining operators can have negative impact on performance, due to two reasons.

Firstly, the partitioning ratio of a chain may not be ideal for each of its operator's performance. The ratio is a compromise of all the operators in a chain.

Secondly, the longer the chain is, the more padding and thus more additional computations there are. Convolution requires padding on the boundary of a feature map for filter processing. Take padding on the H dimension as an example, the padding size (P), shown in Eq. 1, depends on the settings of a convolution operator, *i.e.,* the input and output (H_I and H_O), filter size (F), and stride (S).

$$P = S \times (H_O - 1) - H_I + F \tag{1}$$

As Fig. 12 shows, to add one more operator in a chain, the padding will be propagated along the chain to the first operator. This adds more and more redundant computations to each operator.

Fig. 13 quantizes this padding effect using a chain of different length from YOLO. Compared with no-chain (*i.e.,* share data after each operator), our chain method significantly reduces data sharing overhead. When the chain length is 9, the data sharing overhead is only 9% of no-chain. However, due to padding, the computation latency is increased by 61%, resulting in more latency compared to no-chain. The optimal length in this example should be 8 with 42% improved performance compared to no-chain.

Chain searching algorithm. To find the chains with the best tradeoff between data sharing and computation overhead for a DL model, we design the chain searching algorithm as shown in Algorhtim 1. It is a greedy-like algorithm for this NP problem, which searches for chains with the least latency up to the current operator.

The input of the algorithm is the optimal partitioning plan $ParPlan$ for each operator without chain, stored in the data flow order (results from Sec. 4.1). The output $Chains$ is a set of operator chains with least total latency. Each chain is with settings in partitioning dimension dim, ratio ρ, and chained operators that have been properly padded according to Eq 1. The design idea is that starting from an operator not in any chain op_{head} (Line 2, 18), for each of the potential ρ, the algorithm keeps chaining up more operators and calculating the gain versus no chain. This process stops when there is no gain by adding one more operator. During the process, the ρ and the according chain with the max latency gain $Chain_{maxGain}$ will be recorded (Line 15,16). It will be added to $Chains$ after traversing all the potential ρ (Line 17). The algorithm then starts to search for the next chain (Line 18).

The potential ρ is from a range around the optimal ρ for the first operator of the chain, *i.e.,* $[\rho_{head} - \delta, \rho_{head} + \delta]$ (Line 5). To add the next operator to the chain, we need to adjust each of the preceding operators on the chain to the new padding size (Line 12), and recalculate the total latency by using the latency predictor (Line 13, the true flag is to mark a chained operator). Given that the algorithm checks each operator for a number of times equal to the size of the ratio range, the search space is $2 \times N_{op} \times (\delta/0.1)$ where N_{op} denotes the number of operators in the model. Note that δ limits the searching range of the partitioning ratio. If δ is too large, the search space of Algorithm 1 increases but barely brings improvement on inference performance. If δ is too small, it cannot find the optimal partitioning ratio. We set δ to 0.3 based on our evaluation.

5 NON-LINEARITY- AND CONCURRENCY-AWARE LATENCY PREDICTION

The partitioning and operator chain techniques of CoDL rely on latency prediction of operator co-execution. However, the challenge is how to achieve both accurate and light-weight at the same time. As discussed in Sec. 2, available latency predictors [6, 7, 10, 15, 22, 32, 33] cannot well serve this purpose due to two reasons: 1)

Algorithm 1: Chain searching algorithm

```
input  : $ParPlan$ the partitioning plan of operators in the
         model without chain.
output : $Chains$ the operator chain settings.
1  $next \leftarrow 0$;
2  $(op_{head}, dim_{head}, \rho_{head}) \leftarrow ParPlan[next++]$;
3  while $op_{head}$ is not NULL do
4      $T_{gain}, T_{maxGain} \leftarrow 0$, $i \leftarrow next$;
5      foreach $\rho \in [\rho_{head} - \delta, \rho_{head} + \delta]$ do
6          $Chain_{cur}$.initialize($op_{head}, dim_{head}, \rho$);
7          $T_{chain} \leftarrow Predictor(op_{head}, dim_{head}, \rho, false)$;
8          while $T_{gain} \geq 0$ and $i < N_{op}$ do
9              $(op_{next}, dim_{next}, \rho_{next}) \leftarrow ParPlan[i++]$;
10             $T_{noChain} \leftarrow T_{chain} +$
                 $Predictor(op_{next}, dim_{next}, \rho_{next}, false)$;
11             $Chain_{cur}.append(op_{next}, dim_{head}, \rho)$;
12             $AdjustPadding(Chain_{cur})$;
13             $T_{chain} \leftarrow$
                 $\sum_{op \in Chain_{cur}} Predictor(op, dim, \rho, true)$;
14             $T_{gain} \leftarrow T_{noChain} - T_{chain}$;
15             if $T_{gain} > T_{maxGain}$ then
16                 $Chain_{maxGain}, T_{maxGain}, next \leftarrow$
                     $Chain_{cur}, T_{gain}, i$;
17     $Chains$.append($Chain_{maxGain}$);
18     $(op_{head}, dim_{head}, \rho_{head}) \leftarrow ParPlan[next++]$;
```

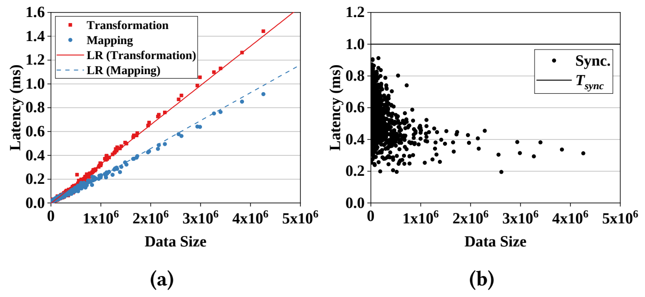


Figure 14: The latency distribution of various sizes of shared data for (a) data transformation and mapping; (b) pre-sync.

none of these predictors considers the data sharing overhead; 2) the predictors cannot be accurate and light-weight at the same time, due to the missing knowledge of underlying platforms.

This section introduces our predictor design. It can achieve both accurate and light-weight by 1) including all the data-sharing overhead for co-execution; 2) analytically formulating the non-linear latency response caused by platform features, which lowers the difficulty of learning. The input of the predictor, as shown in Alg. 1, is the operator hyperparameter (H, W, IC, OC, F, S), dim, ρ, and a flag for chain or not. The output is the predicted latency of the operator co-execution on a given platform.

5.1 Latency composition of concurrency

As shown in Fig. 3, the complete steps for an operator co-execution includes: data transformation, mapping, pre-sync, computing, post-sync, and unmapping. This subsection discusses how to model the data-sharing-related latency.

We first measure the latencies for a range of feature map configurations to learn the latency of data sharing. The measured time of transformation, mapping, and unmapping steps are the according GPU kernel/command running time. The measured time of pre-sync is the timing difference of GPU mapping completion and the CPU acknowledgement of the completion. Similarly, the measured time of post-sync is the timing difference of CPU computing completion and the GPU acknowledgement of the completion.

Our measurement shows that mapping, synchronization, and data transformation can contribute notable overhead in the operator co-execution on the CPU and GPU. They cannot be ignored in the predictor. By comparison, the overhead of unmapping and post-sync are marginal (~50 μs), which are excluded in our predictor. Therefore, the total predicted latency of an operator co-execution T is as Eq. 2, where T_{trans}, T_{map}, T_{psync}, and T_{comp} denote the latency of the data transformation, mapping, pre-sync and computation, respectively. The data sharing overhead (*i.e.,* $T_{trans} + T_{map} + T_{psync}$) is only added when the operator is not in a chain.

$$T = T_{trans} + T_{map} + T_{psync} + \max(T^{cpu}_{comp}, T^{gpu}_{comp}) \qquad (2)$$

As shown in Fig. 14a, T_{trans} and T_{map} have a clear linear relationship with the data size, because they are mostly memory operations. We use linear regression with data size as the feature to learn the latency for T_{trans} and T_{map} in our predictor. For the pre-sync overhead T_{psync}, there is no clear pattern according to our measurements, as demonstrated in Fig. 14b. T_{psync} mainly depends on the driver implementation from vendors. Thus, we use the measured upper bound (1ms) as T_{psync} in our predictor.

5.2 Non-linearity-extracted computing latency prediction

As discussed in Sec. 2, the high complexity of current accurate latency predictor is to capture the non-linear latency response to the scaling of operator hyperparameters. To reduce the complexity, we first analyze the reasons for the non-linearity and formulate it directly in the predictor. The non-linear latency response is mainly due to two reasons.

Firstly, **different algorithms have different latency response to hyperparameter scaling**. Convolution operators employ different algorithms depending on the hyperparameters, such as Winograd for 3×3 and direct convolution for 5×5 convolution.

Secondly, **data blocking on different levels leads to stepped latency response**. As discussed in Sec. 4.1, there are two levels of blocking for a GPU kernel execution *i.e.,* workgroup and warp for inter-core and intra-core parallelism respectively. If the tensor size cannot be evenly divided by the blocking size, there will be idle ALUs or cores. For example, if the size of a work group is (2,5,10) *i.e.,* 100 threads, and the warp size is 64, the work group has to be executed in two warps. This blocking can thus cause stepped latency response.

Table 1: Latency prediction of convolution algorithms for a partitioned operator on a given processor.

Params.	H, W: height and width, IC, OC: the number of input and output channels, F: filter size, Ω: the number of cores of the processor, Φ_c: the blocking size for a core c, ϕ_c: the size of a basic execution unit in c, Φ_{wt}: the size of a Winograd tile, Ψ: the number of Winograd tiles, t_c: time of a basic unit for direct convolution, t_p: time of a basic unit for data packing, t_{mm}: time of a basic unit for matrix multiplication, t_{it}, t_{ot}: time of a basic unit for transforming in-&out-feature map.
Direct	$T_{comp} = \lceil \frac{H \cdot W \cdot OC}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_c$
GEMM	$T_{comp} = T_{packing} + T_{MM}$ $T_{packing} = \lceil \frac{H \cdot W \cdot IC \cdot F^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_p$ $T_{MM} = \lceil \frac{H \cdot W \cdot IC \cdot OC \cdot F^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{mm}$
Winograd	$T_{comp} = T_{inputTrans} + (\Phi_{wt} + 2)^2 T_{GEMM} + T_{outputTrans}$ $\Psi = \lceil \frac{H}{\Phi_{wt}} \rceil \lceil \frac{W}{\Phi_{wt}} \rceil$ $T_{inputTrans} = \lceil \frac{IC \cdot \Psi \cdot (\Phi_{wt}+2)^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{it}$ $T_{outputTrans} = \lceil \frac{OC \cdot \Psi \cdot \Phi_{wt}^2}{\Phi_c \cdot \Omega} \rceil \lceil \frac{\Phi_c}{\phi_c} \rceil t_{ot}$

Similarly, for the CPU, the tensor has to be divided into blocks and assigned to run on each CPU core. Then, to utilize the SIMD (Single Instruction Multiple Data) units, there is also a basic execution unit within a core, such as 8×8 implemented in TFLite framework [26].

Formulating non-linearity. Based on the analysis above, we formulate the non-linearity caused by different algorithms and blocking, and abstract the kernel latency prediction in Eq. 3.

$$T_{kernel} = \lceil \frac{Size_{output}}{Size_{block} \cdot Core\#} \rceil \cdot \lceil \frac{Size_{block}}{Size_{basicUnit}} \rceil \cdot t_{basicUnit} \quad (3)$$

$Size_{output}$ is the partitioned output size, which is calculated for each processor by the predictor input *i.e.,* operator hyperparameter, dim and ρ. $Size_{block}$ is the blocking size for inter-core parallelism given by the inference runtime. $Size_{basicUnit}$ is the size of a basic execution unit for intra-core parallelism given by the inference runtime or hardware parameters. Table 1 shows the deduced equations and detailed calculation process from Eq. 3 for each convolution algorithm.

The only learned variable in the equation is $t_{basicUnit}$, the time of executing a basic unit on a given processor. It depends on the operator hyperparameter, kernel implementation, hardware resources, and scheduling, which is difficult to be analytically modeled. We thus learn it for each kernel implementation from real processor profiling. Since the non-linearity has been extracted, an extreme light-weight linear regression model can be used to learn $t_{basicUnit}$ and achieve high accuracy. The features for this linear model is (H, W, IC, OC). Since it is very light, the model can even cheaply be trained on the target mobile device through gradient descent (results in Sec. 7.5).

Depending on the algorithm, one operator can be implemented in several kernels. The computing time on a processor T_{comp} in Eq. 2 is thus the sum of all the kernels. For example as shown in Tab. 1, the Winograd algorithm on the GPU includes the input and output transformation kernels before and after the matrix multiplication kernel, and thus in Eq. 2, $T_{comp}^{GPU} = T_{inputTrans} + T_{outputTrans} + T_{GEMM}$. For the GEMM algorithm on the CPU, there is normally a data packing kernel before the computation to increase the data locality, and thus $T_{comp}^{CPU} = T_{packing} + T_{GEMM}$. Direct convolution algorithm is normally implemented by one kernel.

6 IMPLEMENTATION

We implement CoDL based on MACE [19], which is a widely used DL inference framework on mobile devices and also has the SOTA performance shown in our evaluation. We integrate the core functionalities of CoDL *i.e.,* the hybrid-type-friendly data sharing and the light-weight latency predictor into Mace. We deliver such functionalities via a pre-compiled shared library. Thus, CoDL can be also easily adapted into other inference frameworks, *e.g.,* Tensorflow Lite [17], MNN [20], etc. In total CoDL consists of 3292 lines of C++ code. CoDL supports the co-execution of commonly used operators in DL models, including convolution, fully connection, and pooling.

To enable the hybrid-type-friendly data sharing on both CPU and GPU, we build the shared library based on OpenCL APIs [21]. Specifically we use `Buffer` and `Image2D` to create the buffer and image type data, respectively. We implement an OpenCL kernel to transform the data between buffer and image type. Once the data is transformed, we use `enqueueMapBuffer` to map the data and `enqueueUnmapMemO- object` to unmap.

To implement the light-weight latency predictor, we make use of the multi-feature linear regression model [27]. To train the predictor, we collect the samples of data transformation, mapping, and computation. For data transformation and mapping, we use the data size as the feature. For computation, we use the (H, W, IC, OC) as the features.

Summarily, we collect about 6,000 samples from five DL models in Sec. 7 in total. We fix the CPU and GPU frequency as maximum value during the sampling. We run the operators from H and OC dimensions with partitioning ratio from 0.1 to 1. The sample ranges of HW, IC, OC, F and S are (1,640), (3,4096), (32,4096), (1,7) and (1,2), respectively. We use 70% of the samples as the training set, the rest is used for testing. We set the learning rate to 0.1, and we train the predictor for around 1,000 epochs.

7 EVALUATION

In this section, we evaluate CoDL. We first introduce our evaluation setup. Then we present the overall performance of CoDL. Next we evaluate each key component in detail *i.e.,* the hybrid-dimension partitioning, the operator chain, and the latency predictor. We also discuss the system overhead of CoDL, including the energy consumption and the memory usage.

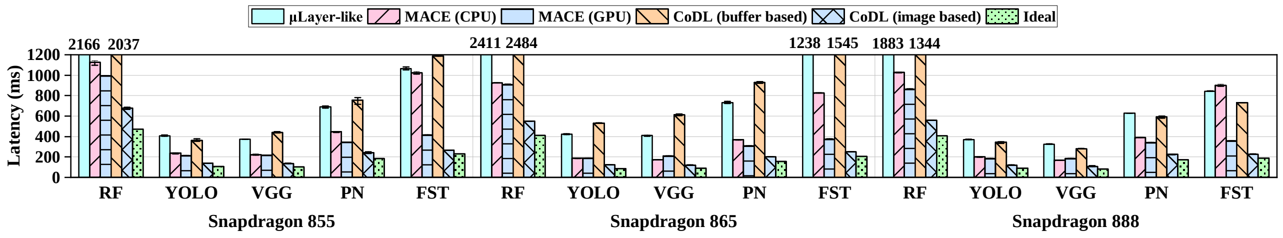


Figure 15: Overall performance of CoDL and baselines on Snapdragon platforms.

Table 2: Hardware used in the evaluation.

Device	SoC	CPU	GPU
Xiaomi 9	Snapdragon 855	Kyro 485	Adreno 640
Redmi K30 Pro	Snapdragon 865	Kyro 585	Adreno 650
Xiaomi 11 Pro	Snapdragon 888	Kyro 680	Adreno 660
Honor V30 Pro	Kirin 990	Cortex-A76	Mali-G76

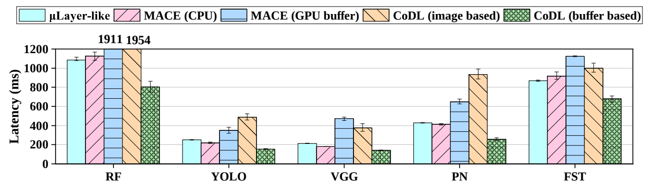


Figure 16: Performance of CoDL and baselines on Kirin 990.

7.1 Experiment setup

Platforms. We evaluate CoDL on various devices as detailed in Table 2. They are equipped with the dominated mobile SoCs, particularly Snapdragon 855 [1], 865 [2], 888 [3] and Kirin 990 [4].

Models. To evaluate the performance of CoDL, we choose various DL models that are widely deployed in real applications, including RetinaFace (RF) [8], YOLOv2 (YOLO) [23], VGG-16 (VGG) [25], PoseNet (PN) [35] and Fast Style Transfer (FST) [5]. They consist of various numbers of operators, ranging from 14 to 61. We execute these models in float32. We run each experiment for 50 times and obtain the averaged results.

Baselines. We compare CoDL with multiple baselines, particularly 1) μLayer-like. We follow the implementation proposed in μLayer [15]. We partition the workload using a FLOPs-based predictor following the design in μLayer [14], and execute the partitioned workload across CPU and GPU using the buffer type data; 2) the inference framework using the single processor, *i.e.,* MACE. we execute the models solely on CPU or GPU with the processor-friendly data type, *i.e.,* image type and buffer type for Adreno GPU and Mali GPU, respectively. All CPUs use buffer type; 3) we also calculate the theoretical performance upper bound of the co-execution system. We exclude the data sharing overhead, and to each operator we always apply the optimal partitioning ratio, which is calculated from the offline profiling.

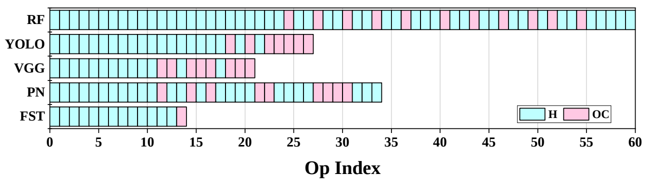


Figure 17: Distribution of partitioning dimension on each operator of selected models.

7.2 Overall performance

First we present the overall performance of CoDL on Snapdragon platforms. Unless otherwise stated, CoDL uses the image type on Snapdragon platforms. Fig. 15 illustrates the results.

Compared to μLayer-like, CoDL accelerates the inference by 3.43× on average, and up to 4.93× speedup is achieved on the tested platforms. The main contributions to such improvement are 1) CoDL exploits hardware-friendly data type on CPU and GPU to make full utilization of the processors; 2) CoDL leverages the hybrid-dimension and the operator chain to reduce the data sharing overhead; 3) CoDL uses the accurate latency predictor to balance workload partitioning.

Compared to the inference on CPU and GPU with MACE, CoDL achieves 2.08× and 1.56× speedup on average, respectively. The acceleration is due to the co-execution of each operator. Compared to CoDL (buffer based), CoDL achieves 3.64× speedup on average because it uses the efficient data type.

What's more, CoDL approaches the theoretical performance upper bound, because CoDL successfully reduce the data sharing overhead and applies the optimal partitioning ratio by the accurate latency predictor. For instance, on the fast style transfer model, CoDL reduces the latency by 73% while the co-execution approach theoretically can reduce by 77% on tested platforms, compared to the inference on CPU with MACE.

We also evaluate the performance of CoDL on the Kirin 990 platform, which is equipped with a Mali GPU [9]. The Mali GPU has the L1 cache, making it more efficient to access the buffer type data. Thus, CoDL makes use of the buffer type on both CPU and GPU for the Kirin platform. Fig. 16 demonstrates the inference latency of the selected models. Compared to μLayer-like, CoDL achieves up to 1.67× and on-average 1.49× speedup on Kirin, because it 1) reduces the overhead of data mapping and synchronization, 2) best balances workload partitioning via the accurate predictor.

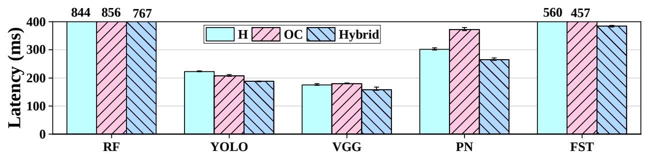


Figure 18: Inference latency with and without hybrid-partitioning dimension.

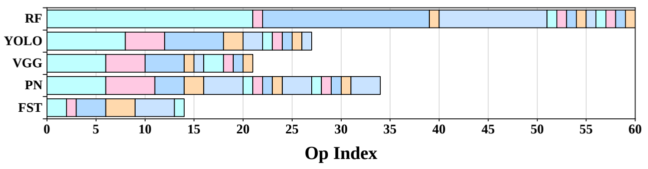


Figure 19: Generated operator chains on the selected models. The operators with same color are in the same chain.

Compared to the inference on CPU and GPU with MACE, CoDL also can accelerate the inference by 2.43× and 1.41× on Kirin, respectively. Compared to CoDL (image based), CoDL achieves 2.66× speedup on average due to the hardware-friendly data type.

7.3 Performance of the hybrid-dimension partitioning

Next we breakdown the system and discuss the performance of each key component, beginning with the hybrid-dimension partitioning. We use the Snapdragon 855 as the platform. We do not apply the operator chain.

We first show the distribution of the partitioning dimension determined by CoDL on the selected models. Shown in Fig. 17, CoDL selects the partitioning dimension by taking account of the redundant concurrency and computing overhead. For the selected models, CoDL determines that 77% and 23% operators are partitioned on *H* and *OC* dimensions, respectively. We also notice that *H* dimension is more likely picked for the up layers, while for the bottom layers we trend to use *OC* dimension to partition. It is because common models usually have large feature maps on the up layers, where partitioning on *H* is more efficient to avoid the redundant data sharing between CPU and GPU.

Fig. 18 illustrates the inference latency with and without the proposed hybrid-dimension partitioning on Snapdragon 855. As the baselines, we use the unified partitioning dimension, *i.e., H* or *OC*. We use the proposed latency predictor to set the optimal partitioning dimension. As shown in Fig. 18, the hybrid-dimension brings 1.2× speedup on average, compared to the unified partitioning on *H* and *OC*, respectively. For the large models, *e.g.,* the fast style transfer, the speedup is much more significant, up to 1.51×, reducing the inference latency from 560ms and 457ms to 384ms according to our evaluation.

7.4 Performance of the operator chain

Next we evaluate the performance of operator chain. Fig. 19 demonstrates the generated operator chains on the selected models. For instance, on RetinaFace, CoDL successfully organizes operators together as chains, the length of top three longest chains are 21 (op 0-20), 18 (op 22-39), 12 (op 40-51). On the selected models, more than 72% of operators on average can be chained together, which significantly reduce the data sharing overhead.

Fig. 20 illustrates the data sharing overhead with and without the operator chain. Due to the reduced redundant data transformation and data sharing times, CoDL with the operator chain reduces the overhead by 55% on average for the selected models. For instance, on RetinaFace, the overhead reduction is up to 45%, from 118ms to 65ms.

7.5 Performance of the latency predictor

In this subsection, we discuss the performance of the latency predictor proposed in CoDL. Table 3 shows the details. We take the FLOPs-based predictor as the baseline, which is used in the SOTA co-execution system [15]. The FLOPs-based predictor considers the number of FLOP of the model, and build a simple linear model to predict the latency. As shown, the FLOPs-based predictor only achieves 8.95% accuracy on average on the tested platforms. By considering the non-linear features and concurrency overhead, our latency predictor achieves 84.03%, 85.17% and 82.96% accuracy on-average for the tested platforms, respectively. The accurate latency prediction leads to the balance workload partitioning between CPU and GPU, which in turn speedups the inference.

Table 3: ±10% accuracy and model size of the latency predictors.

Predictor	Device	±10% Accuracy	Model Size
FLOPs-based	Xiaomi 9	10.46%	8B
	Redmi K30	6.99%	
	Xiaomi 11	9.40%	
nn-Meter	-	~90%	~800MB
Ours	Xiaomi 9	84.03%	500B
	Redmi K30	85.17%	
	Xiaomi 11	82.96%	

Table 4: ±10% prediction accuracy on typical convolution implementations.

Device	CPU-Direct	GPU-Direct	GEMM	Winograd
Xiaomi 9	77.77%	84.64%	91.14%	93.27%
Redmi K30	94.80%	78.39%	90.06%	93.72%
Xiaomi 11	84.56%	80.37%	85.71%	87.10%

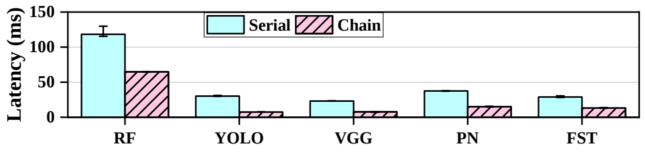
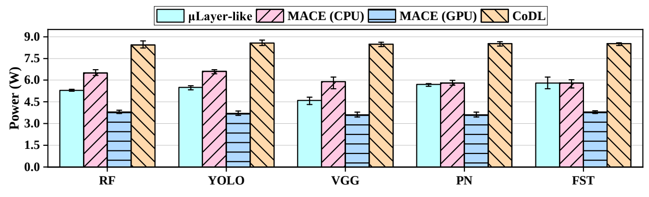
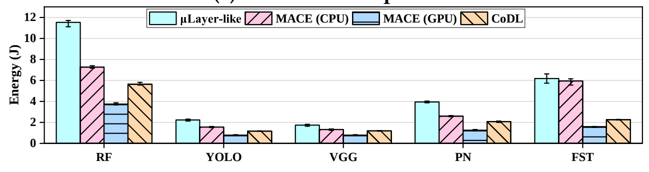


Figure 20: Data sharing overhead with and without the operator chain.



(a) Power consumption



(b) Energy consumption per inference

Figure 21: Power and energy consumption of CoDL and baselines on Snapdragon 855.

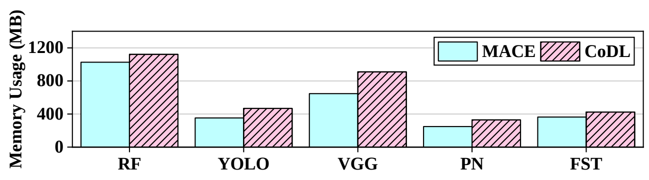


Figure 22: Memory usage of CoDL and MACE on Snapdragon 855.

What's more, the achieved accuracy is even comparable to the black-box latency prediction model *e.g.,* nn-meter (~ 90%) [33]. By understanding the hardware characteristics, we design the small but effective predictor, specially for mobile SoCs. The model size of our predictor is only about 500 bytes, while nn-meter's size is nearly 800 MB which is not suitable for the online prediction on mobile devices.

We also evaluate the prediction accuracy on typical DL kernels *i.e.,* convolution implemented via Direct, GEMM and Winograd with different devices. Detailed in Table 4, our latency predictor obtains up to 94.8% accuracy. On Winograd and GEMM, the averaged accuracy achieved is about 91.36% and 88.97%, respectively.

In addition, the proposed predictor is very light-weight. It only costs 0.2-0.5ms for each prediction. To the predictor, we collect around 6,000 samples. This one-time effort costs around 1.5 hours on the target devices. The training process takes about 1-2 seconds according to our evaluation.

7.6 System overhead

Next we evaluate the system overhead of CoDL, in terms of the power, energy consumption and memory usage.

Fig. 21 illustrates the power and energy consumption of CoDL as well as the baselines on Snapdragon 855. Since CoDL exploit the co-execution of CPU and GPU. the averaged power of CoDL increases by 28.9% and 129.5%, compared to the inference with CPU and GPU on MACE, respectively. CoDL makes use of the friendly data type for CPU and GPU, leading to a higher hardware utilization, thus the power consumption increases by 22.2%, compared to μLayer-like which uses the unified data type, as shown in Fig. 21a.

Although the consumed power is higher, the energy consumption of CoDL might get even lower than baselines thanks to the significant inference speedup achieved by CoDL. Fig 21b shows the averaged energy consumption per inference of CoDL as well as baselines on the selected models. Compared to the inference with CPU on MACE, we reduce the energy consumption by nearly 38.7%. CoDL can also save 62.3% more energy cost on average than μLayer-like. But compared to the GPU on MACE, CoDL increase the energy consumption by 52.0%.

Fig. 22 demonstrates the memory usage of CoDL on Snapdragon 855. To support the co-execution, CoDL allocates extra memory. As shown, CoDL consumes about 25.97% more memory than MACE, on the selected models. For instance, when executing VGG, CoDL uses about 909.02 MB memory in total, while MACE takes about 645.55 MB memory.

8 DISCUSSION

Generality of CoDL. CoDL is a software optimization solution to address the efficiency issue caused by the unawareness of platform features and data sharing overhead to enable efficient concurrent run of on-device CPU and GPU. Although we implement the CoDL on MACE, the idea of concurrency-aware latency prediction and hybrid data sharing between CPU and GPU can be readily applied to other mobile deep learning frameworks, such as TensorFlow Lite [18] and MNN [20]. CoDL can adaptively select the processor-friendly data type for various GPUs by profiling commonly used operators. CoDL also works for on-demand scenarios. If all operators in a new DL model are supported, CoDL performs online prediction and partitioning in a prompt way because the predictor in CoDL is very light-weight with low latency of online partitioning (< 1 sec). However, if there are some unsupported operators in the new model, CoDL requires extensive latency profiling and modeling that takes more time.

Limitation and future work. There are mainly two limitations for CoDL. First, due to the concurrent run of CPU and GPU, CoDL consumes more power than single processor solution. We plan to extend the predictor to model the power consumption behavior of CPU and GPU on DNN model inference, such that CoDL can strike the balance between energy consumption and latency. Second, CoDL hardly achieves speedup for lightweight DL models such as MobileNet [24], since the data sharing overhead caused by partitioning easily dominates the gain brought by the concurrent run of CPU and GPU for small operators.

For the future work, to adapt to more dynamic workloads, we plan to extend the predictor to adjust the predicted latency according to the CPU and GPU utilization. Such that the partition plan can be more flexible. This is however not a simple issue due to the complex resource scheduling and competition in the background. For the real-time workload such as the models leveraging early termination, we could record the termination probability for each termination point in the model. After executing models for many times, the termination probability at each point converges. Then CoDL can make a partitioning plan according to the termination probability at each point, such that partitioned execution is more efficient on-average.

Extended discussion. This work indicates that the specific design of the mobile SoC, *i.e.,* the sharing memory of CPU and GPU, is the root cause that the concurrent run of CPU and GPU can accelerate the DL model inference. However, it involves data sharing overhead including the data transforming and synchronization. To address this issue, a dedicated hardware-based data transformer can be quite helpful to decrease the data transforming overhead. Furthermore, a better implementation of OpenCL can decrease the synchronization overhead.

9 RELATED WORK

Co-execution of heterogeneous processors. Some existing works explore to concurrently utilize the on-device CPU and GPU to tune the DL performance on mobile devices [13, 15, 16, 29]. OPTiC [29] tunes the workload partitioning and core frequencies of CPU and GPU, to keep it within the thermal constraint. μlayer [15] uses both CPU and GPU to accelerate the DL inference by data quantization and workload partitioning. However compared to CoDL, it overlooks the real performance issues in the co-execution such as the data sharing overhead and non-linear latency response caused by hardware characteristics. By taking these factors into account, CoDL successfully reduces the data sharing overhead and best balance the workload partitioning. Furthermore, CoDL helps to decrease the latency for each DL model inference, while it cannot be achieved by the pipeline co-execution.

Performance prediction for DL inference. Existing works design latency prediction models to find efficient model architecture [6, 7, 10, 15, 22, 32, 33]. Cai *et. al* [6] propose a polynomial regression model with the configuration parameters of models as features, but it cannot capture the impacts of underlying platform features, which can be dominant in practice. nn-Meter [33] considers the fusion of multiple operators in runtime, and uses black-box random forest model to predict the latency. Different from the existing works, the predictor in CoDL considers the latency pattern caused by the characteristics of the processors, and the concurrency-specific overhead, which makes the predictor significantly more light-weight and accurate.

Acceleration of DL inference on mobile devices. Numerous works make efforts to better utilize the on-device processors to speedup DL inference, *e.g.,* CPU [28, 30] and GPU [11, 12]. Pipe-It [30] pipelines the consecutive inferences, and split the convolution operator onto the big.LITTLE CPU cores to achieve higher throughput. Asymo [28] partitions workload according to the computing power of the big and little CPU cores, thus improve the scheduling fairness between the asymmetric CPUs. DeepMon [11] caches the convolution results between consecutive frames to reduce the inference latency on video streams. Jiang *et. al* [12] heuristically determine the optimal work group size to improve the utilization of mobile GPUs. The design of CoDL is orthogonal to these works, they can be incorporated with CoDL to further accelerate the inference. DeepThings [34] proposes a fused tile partitioning method which also leverages the idea of operator chain to partition operators of DL models. Different from this work, on one hand, CoDL proposes hybrid-dimension partitioning that considers partitioning alone not only H but also OC dimension to achieve the best co-execution efficiency. On the other hand, CoDL considers the data sharing overhead and the trade-off between the data sharing overhead and computing overhead, which is non-trivial for the co-execution efficiency of operator chains.

10 CONCLUSION

In this paper, we propose CoDL, a CPU-GPU co-execution framework for efficient DL inference. CoDL intelligently balances the overhead of data sharing and computation by adaptively partitioning the operators from hybrid-dimensions, and chaining operators up to reduce the data sharing times. Furthermore, CoDL fairly distributes the workload onto the heterogeneous processors by designing a light-weight but accurate latency predictor, that considers the overhead of co-execution and non-linear platform features. We evaluate CoDL on dominated hardware platforms and the results show that CoDL achieve 3.43× speedup on average, and 62.3% energy saving, compared with the SOTA co-execution system.

11 ACKNOWLEDGMENTS

We sincerely appreciate our shepherd Prof. Kyunghan Lee and anonymous reviewers for their valuable comments, and we thank Bangwen He for his help on the experiments. This work is supported by National Natural Science Foundation of China (62172439, 61702561, 62122095, 62072472, U19A2067), National Key R&D Program of China (2019YFA0706403), Natural Science Foundation of Hunan Province (2020JJ5774, 2020JJ2050), Tsinghua University - AsiaInfo Technologies (China) Inc. Joint Research Center (20203910074), and the Young Talents Plan of Hunan Province of China (2019RS2001), and a grant from the Guoqiang Institute, Tsinghua University.

REFERENCES

[1] Snapdragon 855. 2021. https://www.qualcomm.com/products/snapdragon-855-mobile-platform
[2] Snapdragon 865. 2021. https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform
[3] Snapdragon 888. 2021. https://www.qualcomm.com/products/snapdragon-888-5g-mobile-platform
[4] Kirin 990. 2021. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-990-5G
[5] Jie An, Haoyi Xiong, Jiebo Luo, Jun Huan, and Jinwen Ma. 2019. Fast Universal Style Transfer for Artistic and Photorealistic Rendering. arXiv:1907.03118 [cs.CV]
[6] Ermao Cai, Da-Cheng Juan, Dimitrios Stamoulis, and Diana Marculescu. 2017. NeuralPower: Predict and Deploy Energy-Efficient Convolutional Neural Networks. In *ACML*. 622–637.
[7] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI 18*. USENIX Association, Carlsbad, CA, 578–594.
[8] Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. 2019. RetinaFace: Single-stage Dense Face Localisation in the Wild. arXiv:1905.00641 [cs.CV]
[9] Mali G76. 2021. https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus/mali-g76-gpu
[10] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. 2010. Predicting Execution Time of Computer Programs Using Sparse Polynomial Regression.. In *NIPS*.
[11] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-Based Deep Learning Framework for Continuous Vision Applications. In *MobiSys '17*. Association for Computing Machinery, New York, NY, USA, 82–95.
[12] Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and Optimizing Deep Learning Inference on Mobile GPUs. In *APSys '20*. Association for Computing Machinery, New York, NY, USA, 75–81.

[13] Woosung Kang, Kilho Lee, Jinkyu Lee, Insik Shin, and Hoon Sung Chwa. 2021. LaLaRAND: Flexible Layer-by-Layer CPU/GPU Scheduling for Real-Time DNN Tasks. In *2021 IEEE Real-Time Systems Symposium (RTSS)*. 329–341.

[14] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative Intelligence Between the Cloud and Mobile Edge. *SIGARCH Comput. Archit. News* 45, 1 (April 2017), 615–629.

[15] Youngsok Kim, Joonsung Kim, Dongju Chae, Daehyun Kim, and Jangwoo Kim. 2019. μLayer: Low Latency On-Device Inference Using Cooperative Single-Layer Acceleration and Processor-Friendly Quantization. In *EuroSys '19*. Association for Computing Machinery, New York, NY, USA, Article 45, 15 pages.

[16] Nicholas D. Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. DeepX: A Software Accelerator for Low-Power Deep Learning Inference on Mobile Devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. 1–12.

[17] Tensorflow Lite. 2020. https://www.tensorflow.org/lite/

[18] Tensorflow Lite. 2020. https://www.tensorflow.org/lite/

[19] MACE. 2020. https://github.com/XiaoMi/mace

[20] MNN. 2020. https://github.com/alibaba/MNN

[21] OpenCL. 2021. https://www.khronos.org/opencl/

[22] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. 2017. Paleo: A Performance Model for Deep Neural Networks. In *ICLR*.

[23] J. Redmon and A. Farhadi. 2017. YOLO9000: Better, Faster, Stronger. In *CVPR*. 6517–6525.

[24] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[25] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

[26] Xiaohu Tang, Shihao Han, Li Lyna Zhang, Ting Cao, and Yunxin Liu. 2021. To Bridge Neural Network Design and Real-World Performance: A Behaviour Study for Neural Networks. In *MLSys*. https://www.microsoft.com/en-us/research/publication/to-bridge-neural-network-design-and-real-world-performance-a-behaviour-study-for-neural-networks/

[27] TinyML. 2021. https://github.com/BurnellLiu/TinyML

[28] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu. 2021. AsyMo: Scalable and Efficient Deep-Learning Inference on Asymmetric Mobile CPUs. In *MobiCom '21*. Association for Computing Machinery, New York, NY, USA, 215–228.

[29] S. Wang, G. Ananthanarayanan, and T. Mitra. 2019. OPTiC: Optimizing Collaborative CPU–GPU Computing on Mobile Devices With Thermal Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 3 (2019), 393–406.

[30] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra. 2020. High-Throughput CNN Inference on Embedded ARM Big.LITTLE Multicore Processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 10 (2020), 2254–2267.

[31] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2019. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. In *CVPR*. 10726–10734.

[32] Jinrui Zhang, Deyu Zhang, Xiaohui Xu, Fucheng Jia, Yunxin Liu, Xuanzhe Liu, Ju Ren, and Yaoxue Zhang. 2020. MobiPose: Real-Time Multi-Person Pose Estimation on Mobile Devices. In *SenSys '20*. Association for Computing Machinery, New York, NY, USA, 136–149.

[33] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *MobiSys 2021*. https://www.microsoft.com/en-us/research/publication/nn-meter-towards-accurate-latency-prediction-of-deep-learning-model-inference-on-diverse-edge-devices/

[34] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[35] Christian Zimmermann and Thomas Brox. 2017. Learning to estimate 3d hand pose from single rgb images. In *Proceedings of the IEEE international conference on computer vision*. 4903–4911.