Hyperion: A Generic and Distributed Mobile Offloading Framework on OpenCL

Ziyan Fu[1], Ju Ren[1,5,*], Yunxin Liu[2,6], Ting Cao[3], Deyu Zhang[4], Yuezhi Zhou[1,5], Yaoxue Zhang[1,5]
[1]Department of Computer Science and Technology, BNRist, Tsinghua University, Beijing, China
[2]Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China, [3]Microsoft Research, China
[4]School of Computer Science and Engineering, Central South University, Changsha, China
[5]Zhongguancun Laboratory, Beijing, China, [6]Shanghai Artificial Intelligence Laboratory, Shanghai, China
Email: [1]{fuzy17@mails., renju@, zhouyz@, zhangyx@}tsinghua.edu.cn, [2]liuyunxin@air.tsinghua.edu.cn,
[3]ting.cao@microsoft.com, [4]zdy876@csu.edu.cn

ABSTRACT

Despite the significant development of mobile device SoCs, they are still inefficient in computing computation-intensive workloads, such as high-resolution image processing and AR/VR applications. Offloading offers a promising way to leverage cloud or edge servers for acceleration, but existing offloading is limited to specific tasks or specific hardware/software platforms, resulting in significant engineering overhead. To address this problem, we focus on the underlying layer of these applications (i.e., OpenCL) and propose Hyperion, a generic and distributed mobile offloading framework built on OpenCL. To achieve high-performance distributed execution for Hyperion, we first take a deep insight into the OpenCL data structures and design *regularity-aware kernel analyzer* to analyze the data dependency of work-groups and identify the essential data to offload. Then, *context-aware execution time predictor* is proposed to estimate the computing time of a given partitioned kernel workload that is highly impacted by many runtime factors. These techniques are integrated into *pipeline-enabled and network-adaptive scheduler* to make scheduling decisions, which coordinates the kernel partition and workload scheduling to form pipeline processing between data transmission and distributed execution with flexible adaptability to network dynamics. Extensive experimental results demonstrate that Hyperion achieves superior performance with an average 3.80× speedup compared with the best baseline and flexible adaptation to dynamic network conditions and available computing resources.

CCS CONCEPTS

• **Human-centered computing → Ubiquitous and mobile computing systems and tools**.

KEYWORDS

Mobile offloading, edge computing, distributed computing, OpenCL, online scheduling.

[*]Ju Ren is the corresponding author.

ACM Reference Format:
Ziyan Fu[1], Ju Ren[1,5,*], Yunxin Liu[2,6], Ting Cao[3], Deyu Zhang[4], Yuezhi Zhou[1,5], Yaoxue Zhang[1,5]. 2022. Hyperion: A Generic and Distributed Mobile Offloading Framework on OpenCL. In *ACM Conference on Embedded Networked Sensor Systems (SenSys '22), November 6–9, 2022, Boston, MA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3560905.3568511

1 INTRODUCTION

With the rapid development and high penetration of mobile devices, increasing modern applications are relying on resource-constrained mobile devices to perform computation-intensive and real-time task processing, such as high resolution image processing, AR/VR and 3D reconstruction, etc. For example, 4K image photography is supported in mainstream devices, leading to high image processing burden on mobile devices. Although the computing capability of mobile SoCs has dramatically improved in recent years, they are still inefficient for high-quality image processing, the high computational latency of which seriously affects the user experience. Even with the state-of-the-art algorithms, it takes 4.39 s to perform 4K panoramic picture stitching and 67.4 s for 4K image recognition (YOLOv4-tiny [6] model used) on Xiaomi BlackShark2, which is unacceptable for mobile users. A compromised solution is to compress the input images to reduce the computation complexity. For example, image recognition usually resizes an input image to 416×416 resolution. However, these applications may suffer low accuracy or low quality of output images due to information loss.

To empower the modern applications, a few existing works focus on designing new algorithms with low complexity or compressing DNN models to reduce the amount of calculation [23, 32, 36]. But it is still very difficult, even impossible, to balance the accuracy and computation efficiency for many applications with rigorous performance requirements. Offloading is another way to release the computation burden of mobile devices [33, 36, 55]. It aims to partition the computation task and offload the computationally intensive parts to the cloud or edge servers for performance enhancement. However, most of the existing solutions are proposed for specific tasks or targeted at specific software/hardware platforms. For example, Li et al. [36] proposed a partition scheme for image stitching tasks but it is not applicable to DNN-based tasks, while DeepThings [55] can only be applicable to DNN-based applications but not be directly adapted to others, such as image denoising, transformation, and stitching. Moreover, the majority of DNN partition methods are based on specific usage scenario or hardware, for example, the work [18] is applicable for video object detection, and the work [51] for wearable devices. Consequently, in practical deployment, developers need to design optimization methods case by case, resulting in significant engineering overhead.

Facing the above challenges, it is necessary to seek a more fundamental and generic solution for handling the heterogeneous tasks on different platforms. This motivates us to focus on OpenCL [1], which is an open standard programming model for developing data-parallel applications in heterogeneous multicore architectures (e.g., CPU, GPU, and FPGA). In addition to the compatibility of heterogeneous hardware, it has been widely utilized in different application frameworks, such as OpenCV [9] and Tensorflow-Lite [5], as the backend for supporting parallel execution on heterogeneous platforms. The dual-compatibility for application and hardware, as well as the design principle for general-purpose of computation, make OpenCL a desired framework for coping with task offloading.

However, enabling efficient task partitioning and offloading at the OpenCL level has to address the following challenges. Firstly, the data transmission cost on mobile network is usually high in task offloading. But different from the application-level offloading where the data dependency among different computation parts can be easily identified, OpenCL offers a low-level and more general abstraction on computation without a clear clue on data dependency. Transmitting the whole input data is a straightforward approach but incurs huge transmission overhead. Thus, it is challenging to identify the minimum required data for each computation part and design a delicate computation partitioning and scheduling strategy on OpenCL to reduce the data transmission cost. Secondly, computing time prediction is very important for making offloading decisions, but it is difficult to predict because many complex factors influence computing time. For application-level offloading (like DNN partition), the partitioned computation consists of a number of specific operators that have explored performance on different computation units, while for OpenCL-level offloading, the computing time has irregular relationship with the amount of partitioned tasks. Moreover, the runtime of the computation device (e.g., available resources, concurrent executing tasks) and the unknown concurrent offloading requests from other devices both have significant impacts on the execution of a specific task, posing a further challenge on the accuracy of computing time prediction. Lastly, the dynamic network condition requires the offloading system adaptively changing the partitioning and scheduling decisions to achieve better performance. This is a common but nontrivial challenge in practical offloading systems.

In this paper, we propose Hyperion, a generic mobile offloading framework built on OpenCL, to achieve high-performance tasks offloading among distributed edge servers. The generality of Hyperion is manifested in two folds, a wide support for a large range of applications and for heterogeneous computation units. In Hyperion, each OpenCL kernel[1], such as convolution operations in DNN inference and image format conversion in image processing, is regarded as an independent execution unit on a device. To enable fine-grained computation offloading, Hyperion performs kernel-level partitioning and scheduling by dividing an OpenCL kernel into a number of slices, where each slice consists of a number of work-groups in this kernel.

[1]These terms will be further explained in Table 1 and Section 2.

Hyperion achieves generality and improves the performance through three key technical components. (1) When Hyperion receives an offloading request for an application, it first adopts the *regularity-aware kernel analyzer* to identify the data dependencies of different work-groups, which can be leveraged to direct kernel partition, i.e., deciding how many work-groups in a slice to minimize the transmission overhead. This analyzer provides a generic analysis method for data dependencies to reduce the developers' engineering efforts. (2) Then, before making offloading decisions, Hyperion uses the dedicated *context-aware computing time predictor* to estimate the computing time of a given slice. Obtaining the application-level information (e.g., the resource usage of other applications running in the background and concurrent offloading requests) is simple for application-level offloading but is a challenge in OpenCL-level. The proposed predictor can significantly improve prediction accuracy through an offline-trained predictor to capture the runtime impact and an online calibrator to reflect the impacts of concurrent offloading requests. (3) Finally, Hyperion employs the *pipeline-enabled and network-adaptive scheduler* to make scheduling decisions, including the work-group number of each slice and how many slices to be offloaded for each computation unit. The key idea of the scheduling algorithm is to form pipeline processing between data transmission and distributed execution with flexible adaptability to network dynamics. However, for multi-kernel applications, OpenCL-level scheduling lacks a global perspective of kernel structure, which is a challenge for Hyperion to decide the data placement for the intermediate kernels. To address this issue, we have proposed a lazy-transmission mechanism to transmit data on demand and thus improve pipeline parallelism.

To validate the performance of Hyperion, we build a prototype system of Hyperion and take evaluation under different types of applications including traditional image processing and DNN-based applications. Compared with a few baseline solutions, Hyperion shows superior performance with average 3.80× speedup compared with the best baseline and 4.75× speedup among all baselines. Meanwhile, Hyperion can flexibly schedule the workload assigned to the local and edge servers under dynamic network conditions and high system loads, and even achieves average 2.53× and 1.69× speedups, respectively, in terms of end-to-end execution delay. The main contributions of this paper are summarized as follows.

- To the best of our knowledge, Hyperion is the first generic and distributed mobile offloading framework on OpenCL that can support diverse applications and heterogeneous hardware.
- To address the challenges in OpenCL-level offloading, Hyperion devises three novel techniques: 1) *regularity-aware kernel analyzer* to achieve transmission-efficient kernel partition with minimum required data, 2) *context-aware computing time predictor* to estimate the execution time of a partitioned workload under the runtime of a computation unit, and 3) *pipeline-enabled and network-adaptive scheduler* to form pipeline processing among distributed computation units with network adaptability.
- We implement Hyperion as a library in OpenCL and build a prototype system. Extensive experimental results demonstrate that Hyperion achieves superior performance and flexible adaptation on dynamic network conditions and available computing resources.

Table 1: Terms frequently used in this paper.

Term	Explanation
Kernel	An OpenCL function executed in the backend device.
Work-Item	An OpenCL thread executed on a single computing unit.
Work-Group	(a.k.a WG) A collection of work-items. They execute the same codes and share local memory.
NDRange	An NDRange is an N-dimensional index space. It indicates the spatial relationship of WGs in OpenCL.
Slice	A set of WGs transferred or calculated in a single operation during pipeline processing of Hyperion.

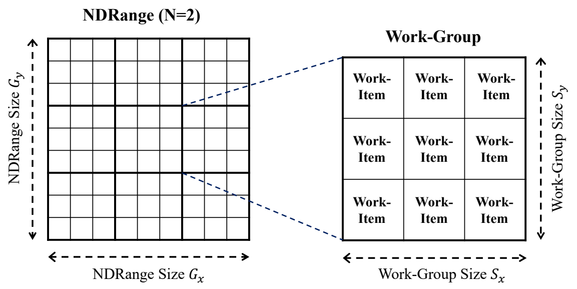


Figure 1: The relationship of NDRange, Work-Group, and Work-Item in OpenCL.

2 BACKGROUND AND MOTIVATION

The rationale of building the offloading framework on OpenCL comes from that it is a cross-platform, parallel programming framework for general-purpose computation and is widely supported by diverse applications and hardware. In OpenCL, each application includes host and kernel codes. Kernel codes are compiled at runtime and executed by the target computation unit [37], while host codes are responsible for transmitting data and control commands to the device. When launching a new kernel, OpenCL emits a bunch of work-items on the device, and some work-items are further combined into a work-group (a.k.a WG). Each work-item in the same WG shares local memory as cache and can conduct concurrent control (e.g., barrier and memory fence). Moreover, all WGs form a 1-dim to 3-dim NDRange, which indicates the spatial relationship of WGs. For example, in the matrix multiplication, each WG is responsible for the result of a sub-matrix, and the hole 2-dim NDRange generates the complete result matrix. Figure 1 shows the relationship of NDRange, WG, and work-items, and we summarize the frequently used terms in Table 1. The OpenCL kernels support not only mobile SoCs but also high-performance computation units on servers, such as CPUs, GPUs, DSPs, thus offloading at the OpenCL kernel level can well fit the mobile scenario where the high-performance and diverse accelerators on edge/cloud servers can be leveraged for computation acceleration.

The basic idea of Hyperion is to partition the OpenCL kernels of each task and offload part of the WGs to nearby edge servers, aiming to achieve distributed computing and thus maximize computing efficiency by fully utilizing both local and edge servers. However, to enhance the performance of such a generic offloading framework, we are facing significant challenges not only from the highly-dynamic mobile scenario, but also from the complex kernel characteristics of OpenCL. Based on our empirical studies, we find and summarize three key challenges as follows.

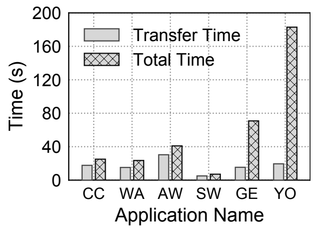


Figure 2: Comparison of data transfer time and total offloading time across different applications.

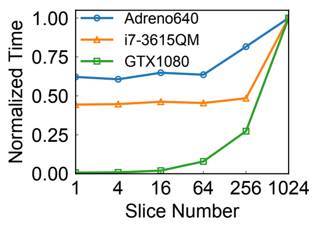


Figure 3: Computing time of CvtColor under different slice number and computation units.

Challenge 1: High data transmission cost for kernel-level offloading.

A typical offloading process includes three steps: uploading data to one of the edge servers, executing, and downloading results from the server. However, in most cases, this approach is even less efficient than computing entirely locally. This is because of the non-negligible data transfer overhead. We conduct an experiment to compare the data transfer time and the corresponding total offloading time of this typical offloading method. In this experiment, we use six widely used applications, the specifications of which are shown in Table 2 (Section 5). The Xiaomi Blackshark2 which deploys these applications transmits data through 100 Mbps Wi-Fi to an edge server with Intel i7-3615QM CPU. The experimental result shows in Figure 2, where the data transfer time takes up 10.7% to 74.1% of the total time of the whole offloading process, which indicates the high transmission cost for kernel-level offloading.

An intuitive solution for reducing the overall data transmission cost is to build a pipeline to parallelize computation and data transmission during the whole offloading process. It means that when the server executes a part of a specific kernel, the data of the next parts and the results of previous executions should be transmitted simultaneously. We use the term *slice* to denote each part for convenience. However, it is non-trivial and challenging to design an efficient pipeline along with kernel partition mechanism for diverse OpenCL kernels. In OpenCL, the WG number in a kernel is a fixed value once the kernel starts execution. If we partition the kernel into fewer slices, the average number of WGs per slice will be larger. Considering that the data uploading of the first slice and the result downloading of the last slice cannot be parallelized with computations, low pipeline performance will occur if they contain excessive WGs. On the contrary, if the number of partitioned slices is too large, tremendous computational overhead occurs on this occasion. Figures 3 and 4 show the relationship between kernel computing time and slice number under different backend devices and kernel types, respectively. The computing time increases dramatically as the slice number increases. In the worst case shown in Figure 3, when using GTX 1080 GPU as the backend computation unit and breaking the kernel into 1024 slices, the computing time increases by 143× compared with that of full offloading (i.e., only one slice), which is unacceptable in most cases. The inefficiency is because a small workload in each slice results in a very low utilization rate of hardware resources [12].

In addition, this inefficiency differs among diverse kernels and computation units, making the challenge more complicated. For example, if we partition the kernel into 64 slices, only 2.3% overhead

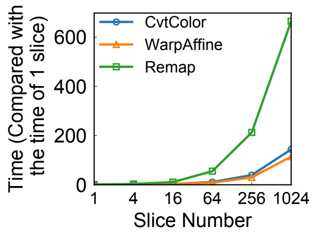


Figure 4: Computing time of GTX 1080 under diverse slice number and kernel types.

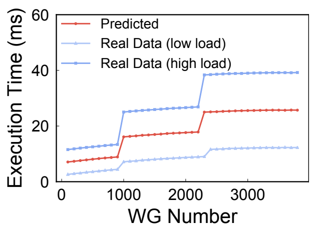


Figure 5: Exist solutions fail to predict the execution time due to ignoring system load.

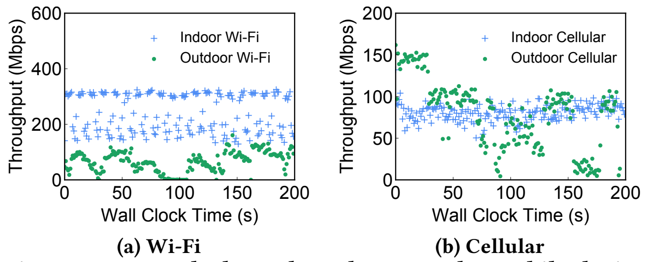


Figure 6: Network throughput between the mobile device and edge server. It changes dramatically over time.

occurs in i7-3615QM CPU while 11.3× overhead in GTX 1080 GPU. Another example is that when we partition the kernel into 16 slices, 2.8× overhead occurs in the CvtColor kernel while 10.8× overhead occurs in the Remap kernel. In summary, the high data transmission cost calls for a sophisticated kernel partition and pipeline mechanism to enhance the performance of offloading.

Challenge 2: Inconsistent kernel execution time under dynamic realtime workloads.

As we aim to low-delay computation, the scheduling decisions require a constant evaluation of the execution time based on device performance. Hyperion will assist the scheduler with this predicted time to achieve performance gain in offloading, and avoid the conditions that some devices become the performance bottleneck due to excessive computation. However, the kernel execution time is affected by many factors such as kernel implementation, cache contention, memory coalescing, device load, and workload scheduling in backend device driver [12]. Unfortunately, many of them are closed-sourced (e.g., drivers) or invisible to developers (e.g., cache contention), making the prediction complicated and challenging.

Existing solutions [35, 54] usually adopt offline training, for example, sampling the performance data with different workloads, building a regression model, and at runtime, querying the model with a given workload. These solutions usually fail to predict the execution time. This is caused by ignoring the runtime status, especially the realtime load, on different computation units. Since the mainstream computation units (i.e., the CPU and GPU) use temporal multiplexing to concurrently execute multiple services [2], the computing resources allocated on a specific application are impacted by the concurrent services in an uncertain way, leading to ever-shifting execution time. We have conducted an experiment to show this gap by using nn-Meter[54], and the experimental results are shown in Figure 5. We use WarpAffine kernel and GTX 1080 GPU as the case study in this experiment. We first train the model offline with sampled data, while the server is still concurrently processing other requests. At runtime, these requests may differ from those at the training stage, thus resulting in different computational resources allocated to our target kernel. The light-blue and blue curves shown in Figure 5 represent the real data under low and high system load, respectively, which shows up to 48.6% error.

Challenge 3: Highly-dynamic network conditions.

This challenge comes from the practical mobile offloading scenario, which is a classic problem for task scheduling and has severe impacts on the system performance. We measured the network dynamics under different scenarios in Figure 6. We found dramatic network throughput fluctuations for both indoor and outdoor cases. This fluctuation is more dramatic outdoor, which is due to many factors such as variation of signal strength and the network congestion [52]. Sometimes it even happens with network outages. Although it is possible to simply use the recent throughput data (e.g., the data transmitted in the last second) and change the workload assigned to each device adaptively [13], this does not apply to the case when the throughput fluctuates dramatically. That is, when Hyperion has decided the size of a new slice, the data transmission time may still differ from predicted value due to the throughput has changed at runtime. This inconsistency makes it more challenging for the scheduler to make decisions. Thus, we need to consider the risk of network fluctuations when making scheduling decisions and reduce the negative impact of such risk.

3 HYPERION SYSTEM OVERVIEW

To address the aforementioned challenges, we proposed Hyperion, which offers a generic and distributed offloading framework for various applications and works as a library in OpenCL. When a new kernel arrives, Hyperion automatically makes scheduling decisions and offloads parts of the workload to multiple edge servers.

Figure 7 shows the system overview of Hyperion. We consider a general edge computing scenario, where a number of mobile devices connect to a cluster of edge servers through a wireless access point. For each mobile device, the directly connected edge server acts as its primary edge server, which is responsible for receiving the offloading requests and coordinating the nearby edge servers for distributed execution. Since the cluster of edge servers is usually built on a high-speed wired network, the transmission cost among them is far less than the communication between mobile device and the primary server. Enabled by the characteristics of OpenCL, the heterogeneous computation units in each edge server, such as CPU, GPU, and FPGA, are regarded as independent computation devices by Hyperion for workload offloading.

The workflow of Hyperion is as follows. ① When a new kernel with its corresponding input data arrives, the *regularity-aware kernel analyzer* parses the data dependency of WGs and identifies the essential data to offload with objective of minimizing the data transmission cost. This analysis is performed for new kernels. If a kernel has been executed before, we can skip the analysis and directly take the result. ② Next, the *context-aware computing time predictor* estimates the execution time of a given slice on a specific computation unit according to the runtime conditions. ③ Then, with the kernel analytical information and the predicted execution time, the *pipeline-enabled and network-adaptive scheduler* calculates the WG number in the slice for each computation unit according to the network dynamics. The slice will be scheduled for either local execution or offloading through pipeline processing (④) that

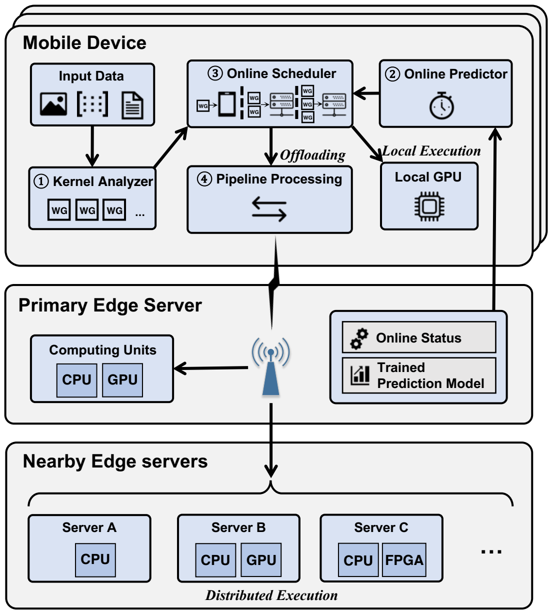


Figure 7: System overview of Hyperion.

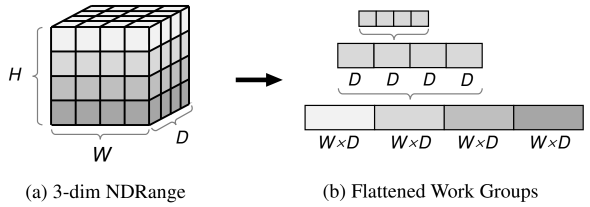


Figure 8: 3D NDRange to flattened WGs

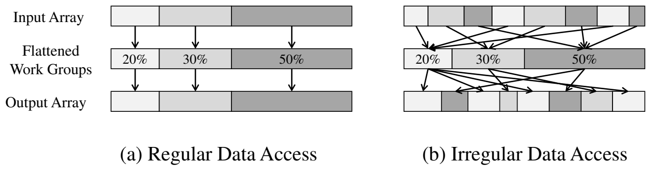


Figure 9: Regular and irregular data access patterns

enables the parallel processing of transmissions and computation to save time. Finally, each offloading slice will be sent to the target server for distributed computing.

4 SYSTEM DESIGN

4.1 Regularity-Aware Kernel Analyzer

Original OpenCL kernels are designed to be executed in a single device where the corresponding data are stored in the dedicated memory or cache. If we implement distributed computing on two or more devices at different geographic positions, these distributed devices cannot efficiently share a global memory, resulting in a huge cost for data transmission. A primitive method is to transmit the whole input data to all devices, and thus each device can efficiently obtain the required data from local memory when needed. However, this approach incurs a huge transmission overhead. Another way is asynchronous transmission [50]. When a device needs some data for computing, but the corresponding data do not exist in the local memory, the operating system generates a page fault, suspends the computing, and sends requests to other devices for data transmission. However, excessive page faults will be generated at the beginning of the kernel execution, as no data exists in the edge server at this time. Meanwhile, a waste of computing resources occurs because the computation is suspended, waiting for data transmission. Thus, the principle for kernel analyzer design in Hyperion is to identify the data dependency of different WGs, simultaneously reducing the data transmission overhead and avoiding excessive computing suspension.

In Hyperion, we classify the memory data of a kernel into two types: *common data* and *exclusive data*. Common data are frequently accessed by the kernel at runtime, and each device involved in the kernel's distributed computing should keep a copy. Thus, common data is transferred first at the beginning of kernel execution to avoid excessive network overhead caused by frequent page faults. Exclusive data is WG specific in OpenCL, which is only used by one WG exclusively. In this way, we can only transmit the exclusive data to one of the devices to reduce the network overhead. Therefore, the problem is how to identify the common and exclusive data.

To address this, we propose a regularity-aware kernel analysis method for Hyperion, which is inspired by SKMD [35]. Different from kernel-level analysis in SKMD, Hyperion classifies each data according to its data access pattern and further partitions and transmits parts of regular data to reduce network cost. The data access pattern includes regular and irregular ways. As Figure 9 shows, each WG only reads or writes data to the contiguous locations in the regular access pattern while discontiguous locations in the irregular access pattern. This data access pattern is determined by the functionality of the application. For example, the regular data access pattern is common in image processing applications, where the pixel-by-pixel processing ensures each data can be equally accessed. On the other hand, in the linear algebra domain, most kernels need to determine data access according to the input, which is irregular data access in these kernels. To design the regularity-aware kernel analyzer, we first flatten the 3-dimensional NDRange to form a WG sequence, as shown in Figure 8. Then, for each input and output array in this kernel, we detect their data access patterns and regard the regular access data as exclusive data. For irregular access pattern data, there is still no effective and general partition scheme for all kinds of kernels. Keeping complete copies to other devices is still the most efficient solution chosen in existing research, for example, SKMD [35] and SunCL [30]. Thus, we regard all of these data as common data. Similarly, if the kernel has an irregular data access pattern for output arrays, we transmit a full copy of output arrays back to the local, and merge different versions of these arrays from multiple devices by using the method proposed in SKMD [35].

To implement this data access pattern detection, we have developed an analysis pass `parseKernel` by leveraging LLVM SDK [34]. First, we decide the partition granularity G, the size selected from element numbers by partitioning the NDRange along one or several dimensions. For example, rows of the 2D NDRange, or planes of the 3D NDRange. Then, WGs are partitioned to $B = \frac{M}{G}$ blocks, where M is the number of WGs in the NDRange. Each memory object is also divided into B sub-objects with equal size. After that, for WGs in a block with index `BID`, we check whether addresses of I/O operations for each memory object are bounded within the sub-object with index `BID`. This check is conducted by using LLVM Scalar

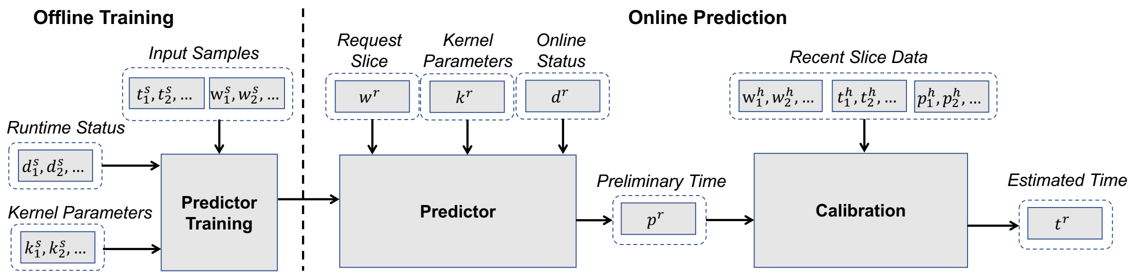


Figure 10: Data flow of context-aware computing time prediction

Evolution (SCEV) [48]. A memory object is regarded as exclusive data if it passes this check, otherwise as common data. We check all possible partition granularities and select the granularity that can partition the most memory objects. We take matrix multiplication as an example, which implements the function $C = A \times B$. Matrices A and B are the input, and the matrix C is the output. We assume all matrices with the size of 4000×4000 and focus on the data access pattern of matrix A in this example. If a WG is responsible for a single value in C, then $G = 1$ and $B = 4000 \times 4000$ in this case. In fact, the WG usually works on a tile (e.g., 8×8 sub-matrix). As we have flattened the matrix data to a single-dimensional array, the data access locations of a WG are discontiguous as the tile requires data from different rows. In this case, if we set $G = 8 \times 4000$ and $B = 500$, the matrix A is the regular pattern because the data access of the WG is restricted in the corresponding block.

All the common data are transferred at the beginning of the task. Instead of being transmitted to each server separately, common data are transmitted to the primary server while this server will further send the data to other servers asynchronously to save time.

4.2 Context-Aware Computing Time Predictor

Before making the scheduling decision, Hyperion uses a dedicated context-aware computing time predictor to estimate the computing time of a given slice on a specific computation unit under the runtime status. As Figure 10 shows, the predictor consists of two stages: *offline training* and *online prediction.*

4.2.1 Offline Training. During the offline training, we see the kernel execution on a specific computation unit as a black-box matter. Hyperion samples data under different configurations and runtime status, and records its corresponding computing time. A configuration includes the WG number w_i^s and the parameters of the kernel k_i^s ($i = 1, 2, \ldots, M$), e.g., the filter size, stride, and pad size for convolution operation kernel, and input image size for an image format conversion kernel. The runtime status includes the number and the type of existing concurrent tasks, as well as the utilization ratio of the computation unit. Since different kinds of tasks usually cause different impacts on resource competition, we can roughly divide the representative tasks into a few types, such as DNN inference, image transformation, and linear algebra. Then, we train a lightweight prediction model for each device using random forest regression, with the configurations and runtime status as the input and the corresponding computing time as the output. Such that, Hyperion can gradually learn the complex relationship between the execution time of a given slice and computation context.

Since the model training is lightweight and related to specific hardware, each edge server maintains such a model for each computation unit in an offline way. These trained models can serve as a prediction service in each edge server for mobile devices to call.

4.2.2 Online Prediction. When a mobile device initializes a new prediction request, it first calls the model to calculate a preliminary predicted time p^r. However, this predicted time is still not sufficiently accurate. This is because the concurrent services impact the computing resources allocated on a current application in an uncertain way. Although we have considered the system runtime status in offline training, during the online prediction, the server may be executing other applications which are different from those during offline training. For example, offline training was performed long ago, and new services were deployed on this server after the training. These new services may require inconsistent computing resources. Meanwhile, other mobile devices may concurrently offload workload to the computation unit. The problem is that when a mobile device predicts the execution time for a specific kernel and makes scheduling in a distributed way, it cannot accurately know how many and what types of workloads will be scheduled on the same computing unit.

To address this issue, we propose a calibration method for the preliminary predicted time, which fills the gap between offline sampling and online actual data. First, we collect historical slice execution data in the recent 30 seconds, including WG numbers w_i^h and the corresponding computing times t_i^h. These computing times are recorded at runtime. We also collect the preliminary predicted time for each slice, denoting it as p_i^h. Then, we build a linear regression model from these data. The usage of the linear regression model is based on two considerations: (1) it is lightweight, and (2) the computing time of a specific slice has a linear relationship with the number of concurrent offloading tasks [33]. Based on above analysis, we use $t_i^h = x_0 p_i^h + x_1$ to fit all these data, where x_0 and x_1 is the parameters to be learned from these data. Finally, Hyperion calibrate preliminary predicted time through $t^r = x_0 p^r + x_1$. When there is no historical data at the beginning of the kernel execution, we use the value of x_0 and x_1 from other kernels as a substitute if available. Otherwise, we can directly utilize the preliminary time as the estimated time to solve the cold start problem.

4.3 Pipeline-enabled and Network-adaptive Scheduler

The scheduler decides the WG number of each slice and how many slices to be offloaded for each computation unit at runtime. Meanwhile, it can achieve pipeline processing and adapt to network

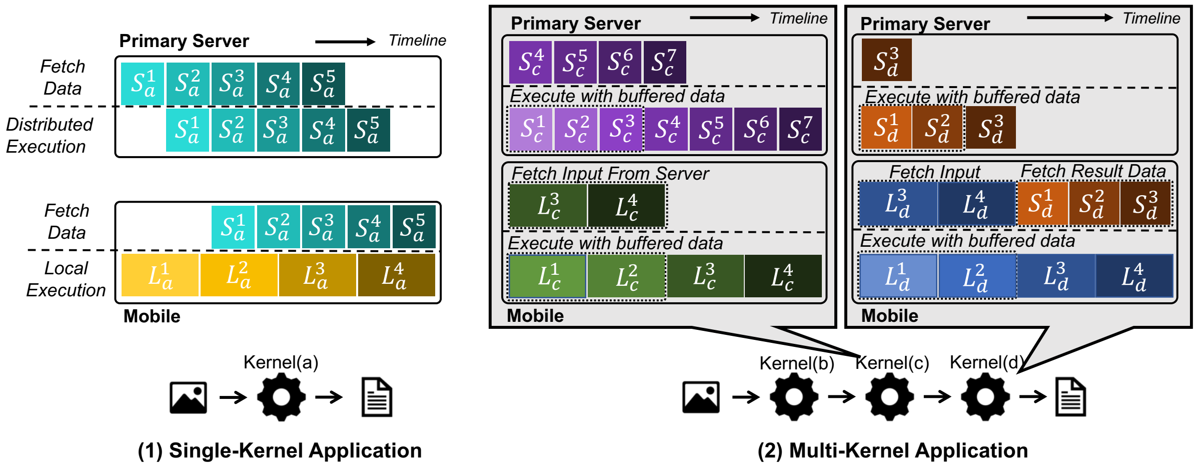


Figure 11: Examples of pipeline processing. S and L are slices assigned to the edge server and mobile, respectively.

dynamics in online scheduling. Before we introduce the scheduling strategy, we first explain the main idea of pipeline processing.

4.3.1 Pipeline Processing. As we have mentioned in the system overview, the data transmission overhead for Hyperion is mainly caused by the wireless communication between the mobile device and the primary server. The key to reduce the data transmission cost during the whole offloading process is to make the data transmission and distributed computing form a pipeline. The pipeline design is simple for single-kernel applications but is a challenge for multi-kernel applications because OpenCL-level scheduling lacks a global perspective of structures of kernels. Thus, without knowing the future kernel to be executed, it prevents the scheduler from better deciding the data placement and achieving more efficient computing. To address this, we propose a lazy-transmission mechanism to transmit data on demand for better pipeline parallelism.

We use examples to illustrate the pipeline processing idea in Hyperion. Figure 11(a) shows an example of the single-kernel application, where $L_a^i,\ i \in \{1,2,3,4\}$ denote the slices allocated to local device (namely, *local slice*), and $S_a^i,\ i \in \{1,2,\ldots,5\}$ denote the slices allocated to the edge server (namely, *server slice*). The pipeline is easy to form in this case. When executing slices in servers, the primary server can organize data upload of the next slice and result download for the previous slice. For example, when executing S_a^3, the upload of S_a^4 and the download of S_a^2 can be processed in parallel. The devices achieve balanced loads if the last execution of local slice (i.e., L_a^4) and last download of server slice (i.e., S_a^5) are simultaneously finished. However, the situation becomes complex in multi-kernel applications, as shown in Figure 11(b). In this case, to avoid unnecessary data transmissions, when a server completes a slice, we do not immediately transfer the result data back to the local device, but check if only edge servers will use these data in subsequent computations. If true, these data will be buffered in the primary edge server to reduce network overhead. Similarly, the local device also maintains a buffer to store the result data of local slices. However, it is a challenge to know the subsequent computations because conditional branches may occur. To address this, we propose a lazy-transmission mechanism: except for the last kernel, the result data are temporarily stored in the corresponding buffer and are not transmitted through the network. When executing a subsequent kernel, we first decide the workload assigned to the local and edge servers, and check whether the buffer contains all input data. If not, Hyperion sends a request to fetch missing data and starts the execution with available data in the buffer. For example, after kernel(c) in Figure 11(b) is launched, Hyperion first executes the WGs using the available input data from the buffer, such as local slice L_c^1 and L_c^2, and server slice S_c^1, S_c^2, and S_c^3. At the same time, Hyperion transmits essential data to ensure the completion of other slices, i.e., upload data of S_c^4 to S_c^7, and download data for L_c^3 and L_c^4. Similarly, for the kernel(d), Hyperion first checks the buffer and directly executes the slice with the buffer data: local slice L_d^1, L_d^2 and server slice S_d^1, S_d^2. Meanwhile, the edge server obtains the essential data for S_d^3, and the mobile obtains the data of L_d^3, L_d^4. The process of data transmission and computing can form pipeline parallelism. As kernel(d) is the last kernel of the application, the edge server should transmit the result data (i.e., S_d^1, S_d^2, and S_d^3) to the mobile device and empties its buffer. In kernel(c), Hyperion achieves load balance when both the computation of the server and mobile finish at the same time, conducing to high resource utilization. For kernel(d), we should consider the network overhead of result data transmission. In this case, load balance is achieved if the download of the last server slice and the execution of the last execution of local slice simultaneously finish.

Guided by the pipeline processing idea mentioned above, we introduce the scheduling strategy in Hyperion in the following. Before kernel execution, we need to complete the preparatory work, including kernel analysis and offline training. After the kernel execution starts, the scheduler first makes scheduling decisions for mobile and the primary edge server. A scheduling decision includes how many WGs the device should execute in the next period of time and how much data should be fetched. The primary edge server is responsible for managing the data received from mobile. This server will send data to the target edge for execution when it receives an execution command from the scheduler. The scheduler will make the next decision after completing the last execution or transmission. The workflow of the scheduler is as follows: (1) First, we evaluate the performance of distributed computing, and (2) set an upper bound of WG number in each slice to cope with dynamic network conditions. (3) During runtime, the scheduler decides the ideal WG number in each slice for pipeline processing with the latest network conditions. (4) With the derived information in the previous two steps, we can determine the WG number of the next slice and adjust the workload assigned to the local and edge servers, respectively.

4.3.2 *Computing Time Predictor for Multiple Edge Servers.*

In the scheduling process, it is common for Hyperion to predict the computing time of distributed computing across multiple edge servers and then determine whether the offloading is worthwhile. Specifically, assuming that N servers provide computing services to the mobile user, based on the prediction result in Section 4.2.2, we can calculate the computing time $t = T_i(w)$ of w WGs in server i. Meanwhile, we can also decide the number WGs that each server can complete in a give time, which denote as $w = T_i^{-1}(t)$. Thus, in this given time t, these servers can finish w' WGs through distributed computing, where

$$w' = T_1^{-1}(t) + T_2^{-1}(t) + \ldots + T_N^{-1}(t). \tag{1}$$

We use $T_s^{-1}(t) = \Sigma_{i=1}^{N} T_i^{-1}(t)$ to abbreviate Eq. (1), and $t = T_s(w)$ is the time of distributed computing.

4.3.3 *Getting the Upper Bound of WG number.*

Then, we determine the upper bound of the WG number in a single slice. This upper bound is set mainly due to the dynamic network conditions. If a slice has excessive WGs and the network condition changes at runtime, the computing time and network transmission overhead will differ from the predicted value, which harms overall performance. For example, when a slice is executed by the server, and the network throughput goes down at this time, the time for the mobile device to obtain the result data will be extended. The local has nothing to do except wait for the data. Actually, it is better to assign less workload to edge servers and more to the mobile for better-balanced loads.

A possible solution is to minimize the workload in the slice, i.e., only one WG in each slice, where Hyperion adjust the workload for each backend device for better-balanced load according to the latest network conditions. However, this solution degrades the computation efficiency due to the low occupancy of hardware resources. Contrarily, assigning a large workload improves the computation efficiency but also incurs low responsiveness due to long execution and transmission time. As a result, it is more susceptible to dynamic networks that the time is inconsistent with that of the scheduling predicted, and thus it difficult to achieve balanced loads. Thus, we appropriately allow a small amount of computational overhead and choose the smallest WG number within the overhead limit to cope with the dynamic network conditions. We use c to denote this overhead.

Specifically, we first find the minimum computing time across all possible partition plans:

$$T_c^k = min_w(\lceil \frac{M}{w} \rceil T_k(w)) \quad k \in \{l, s\}, \tag{2}$$

where M is the total number of WGs in the kernel, and the $T_l(w)$ is the computing time of local execution with a slice of w WGs. Then, we find the minimum WG number with at most c overhead:

$$w_{\mathtt{max}}^k = min\{w \mid \lceil \frac{M}{w} \rceil T_k(w) \le (c+1)T_c^k\} \quad k \in \{l, s\}, \tag{3}$$

where $w_{\mathtt{max}}^k$ is the upper bound of WG number in each slice for the local and edge, respectively.

4.3.4 *Deciding the WG Number for Pipeline Processing.*

With the upper bound, the scheduler then needs to decide the WG number in each slice to maximize the pipeline parallelism. In the ideal case, the computational overhead of each slice can just cover the data transmission overhead. Specifically, for the edge servers, we find the minimum WG number w_0^s that satisfies:

$$w_0^s = min\{w \mid \max\{\frac{ws_i}{b_u}, \frac{ws_o}{b_d}\} = T_s(w)\}. \tag{4}$$

Hyperion uses s_i and s_o to denote average input and output data size per WG, and b_u and b_d to denote network upload and download throughput. As we only perform result data transmissions for the last kernel of the application, $s_o = 0$ if the kernel is not the last. We select the minimum WG number to react to the dynamic network conditions quickly. We only consider the overhead of input data transmissions for the local device if local data is incomplete. On this occasion, we find the minimum WG number w_0^l that satisfies:

$$w_0^l = min\{w \mid \frac{ws_i}{b_d} = T_l(w)\}. \tag{5}$$

If the computing time is always larger than the data transmission time, we choose the largest possible WG number (i.e., $w_0^k = w_{\mathtt{max}}^k,\ k \in \{l, s\}$) to reduce computational overhead. Conversely, if the computing time is always less than the data transmission time, we choose the smallest possible WG number to react to dynamic network conditions quickly.

4.3.5 *Workload Scheduling.*

Algorithm 1 shows the process of our scheduling algorithm. Hyperion first decides the remaining workload assigned to the local and edge servers, respectively (Lines 1-11). This information is used by the lazy-transmission mechanism to decide necessary input data to be transmitted. At the same time, we detect whether the next slice is the last one, and if so, we need to treat it specially to ensure balanced loads. The second part is to decide the next WG numbers for requesting input data, executing, and sending output data (Lines 12-23). This decision should consider several conditions to maintain high pipeline parallelism. For example, some input data have been buffered in devices for multi-kernel applications, and meanwhile, the value of w_0^k is in constant change due to dynamic network conditions.

Specifically, Hyperion enumerates all possible schedule plans (Lines 2-3) and estimates the finish time of the kernel (Lines 4-10). This estimation includes analysis of computational overhead and data transmission overhead. The computational overhead is estimated based on the computing efficiency (Line 1), which is the WG execution speed with $w_{\mathtt{max}}^k$ ($k \in \{l, s\}$) WGs in slices. Meanwhile, the data transmission overhead is estimated by the size of data and the latest network throughput. In particular, we need to check whether the current kernel is the last one of the application, and if so, we need to consider the result data transmission to local (Line 5). Finally, we select the schedule plan with the shortest finish time as the target plan (Lines 10-11).

In the second part, Hyperion first checks available WGs (w_a^k) whose data is already received by the local or edge but not yet executed (Line 12). Then, Hyperion adopt different scheduling policies based on the relationship between w_a^k, w_0^k, and w_t^k ($k \in \{l, s\}$). We first check if all input data is ready for the current device, and no data transmission is required in this case (Lines 13-14). Then, if the

Algorithm 1: Scheduling Algorithm

```
Input  : Latest execution or transmission finish time for local
         and server: $f_l$, $f_s$
         The number of WGs of which data is ready: $w_r^l$, $w_r^s$
         The number of WGs exeucted: $w_e^l$, $w_e^s$
         Upper bound of WG number in a slice: $w_{\max}^l$, $w_{\max}^s$
         Ideal WG numbrer for pipeline processing: $w_0^l$, $w_0^s$
         Total WGs currently unexecuted: $w_u$
         The number of WGs of which result data have
transmitted to local: $w_d^s$
Output : $w_e, w_i, w_o$ : executing $w_e$ WGs for next slice, and
         requiring input of $w_i$ WGs and output of $w_o$ WGs.
1  $E_c^l \leftarrow \frac{T_l(w_{\max}^l)}{w_{\max}^l}$, $E_c^s \leftarrow \frac{T_s(w_{\max}^s)}{w_{\max}^s}$, $F_0 \leftarrow \infty$
2  for $i = 0 \rightarrow w_r$ do
3      $w_l \leftarrow i + w_e^l$, $w_s \leftarrow w_r - i + w_e^s$
4      if the kernel is the last kernel in the application then
5          $C_d^l \leftarrow (w_l - w_r^l)\frac{s_i}{b_d} + (w_s - w_d^s)\frac{s_o}{b_d}$
6      else
7          $C_d^l \leftarrow (w_l - w_r^l)\frac{s_i}{b_d}$
8      $F_l \leftarrow f_l + max\{C_d^l, (w_l - w_e^l)E_c^l\}$
9      $F_s \leftarrow f_s + max\{(w_s - w_r^s)\frac{s_i}{b_u},\ (w_s - w_e^s)E_c^s\}$
10     if $max\{F_l,\ F_s\} < F_0$ then
11         $F_0 \leftarrow max\{F_l,\ F_s\}$, $w_t^l \leftarrow w_l$, $w_t^s \leftarrow w_s$
12 $w_a^k \leftarrow w_r^k - w_e^k$ ($k = l$ if scheduling for the local, or $k = s$)
13 if $w_t^k \le w_a^k$ then
14     $w_e \leftarrow \min(w_t^k, w_{\max}^k)$
15 else if $w_a^k < w_0^k$ then
16     $w_i \leftarrow \min(w_0^k - w_a^k, w_t^k - w_a^k)$
17 else if $w_0^k <= w_a^k$ then
18     $w_e \leftarrow \min(w_a^k, w_{\max}^k)$
19     if scheduling for local device then
20         $w_i \leftarrow \min(\frac{b_d T_l(w_e)}{s_i}, w_t^l - w_a^l)$
21     else
22         $w_i \leftarrow \min(\frac{b_u T_s(w_e)}{s_i}, w_t^s - w_a^s)$, $w_o \leftarrow \frac{b_d T_s(w_e)}{s_o}$
23 Return $w_i,\ w_e,\ w_o$
```

available WGs cannot meet ideal pipeline parallelism, we continue to transmit the input data until $w_a^k = w_0^k$ and do not allocate slice execution on edge servers (Lines 15-16). This policy will improve the pipeline parallelism of the next slice. The most complicated condition is when more than w_0^k WGs are available, we allocate the maximum feasible WGs w for execution and, meanwhile, progress the data transmissions within the predicted execution time (i.e., $T_k(w)$) (Lines 17-22).

In addition, Hyperion estimates the remaining number of slices to finish the kernel computation based on the scheduling algorithm. If there are only two slices left, we decide whether to merge these slices. This is because the last slice may have small workloads, which leads to high overhead. Finally, Hyperion predicts the finish time of these two plans and selects the efficient one to execute.

4.4 Overhead Analysis

The overall complexity is $O(NM + Mf)$, $O(NM)$ for predicting distributed computing performance and $O(Mf)$ for the scheduler. N is the number of servers, M is the total number of WGs, and f is the scheduling frequency, which equals the number of slices generated during online scheduling. This scheduling frequency equals the total WG number in the worst and rare cases. Meanwhile, M can be thousands to millions level in some kernels. To address this issue, if a kernel has excessive WGs, we adopt a coarse-grained scheduling approach, which further merges some WGs into *chunks*, and perform the scheduling algorithm at chunk granularity instead of WG granularity. We limit the total number of chunks which satisfies that each schedule latency is less than the execution time of a single WG on the mobile. Thus, the scheduling latency can be overlapped by the kernel execution in the worst case.

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We have implemented a prototype of Hyperion, which consists of client side on Android and edge server side on Ubuntu 20.04 LTS. The testbed in our evaluations consists of two edge servers and ten mobile phones. These device numbers can be adjusted according to actual conditions. For the edge servers, one with Intel i7-3615QM CPU acts as the primary edge server and the other provides two NVIDIA GTX 1080 GPU. They receive requests and directly call the native OpenCL library for computing. For the mobiles, we mainly use a XiaoMi BlackShark 2 for evaluation, and the others concurrently request services to change the load of servers. We have implemented Hyperion with C++ codes such that developers can easily access it through Android Java Native Interface [3]. We use gRPC [4] as the tool for communication between servers and clients, which is a high-performance and lightweight RPC framework developed by Google. The mobile phones communicate with servers through an IEEE 802.11ac Wi-Fi connection.

5.2 Methodology

Applications: We select 5 representative image processing applications that are OpenCV-based single kernel applications [9] which are widely deployed in modern mobile phones, and two multiple-kernel applications, YOLOv4-tiny [6] and ResNet-50 [20] which are DNN inference applications widely used in image object detection and classification. Table 2 shows the application specifications, including input configurations and function notes. We use default data in OpenCV profiling tool as the input.

Baselines: Some baselines are implemented for comparison.

- **SKMD [35]:** SKMD is a task scheduling framework on OpenCL for high-performance computing platforms, which implements distributed computing across multiple servers. Before executing a new kernel, the scheduler of SKMD figures out the workload for each server targeting load balance. Each server is executed only once (i.e., one slice).
- **Greed [42]:** The Greed algorithm is based on the task scheduling method proposed in [42]. It breaks the task into several slices (e.g., 64 slices in our experiments), each of which has the same

Table 2: Application specifications

Applications	Abbr.	Input Parameters	Function
CvtColor	CC	3840 × 2160 image, code=COLOR_Lab2BGR, scn=3, dcn=4	Color Space Conversion
WarpAffine	WA	3840 × 2160 image, type=8UC3, interpolation=INTER_CUBIC	Affine Transformation
AddWeighted	AW	3840 × 2160 images, depth=8UC4	Blending Two Images
StitchingWarper	SW	3840 × 2160 image, warper=PlaneWarperType	Image Mapping
GEMM	GE	4096 × 4096 matrices, type=CV_32FC1	Generialized Matrix Multiplication
YOLOv4-tiny	YO	4000 × 4000 image	Image Detection
ResNet-50	RS	2000 × 2000 image	Image Classification

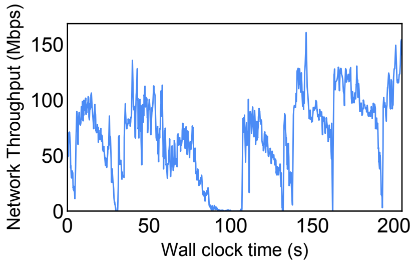


Figure 12: Network throughput variation during our walk.

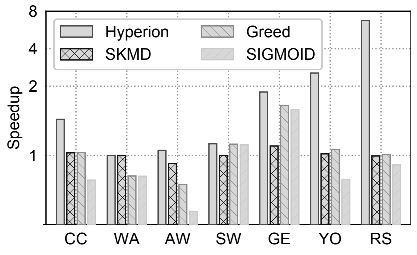


Figure 13: Speedup comparison under network dynamics (higher is better).

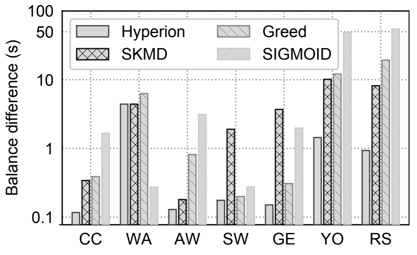


Figure 14: Balance difference comparison under network dynamics.

workload. The scheduler monitors each device and allocates a new slice if one device completes the previous slice.

- **SIGMOID [43]:** SIGMOID proposes a progressive method, which assigns large slices at the beginning of the kernel execution, and then gradually decreases the workload to ensure load balance.
- **Full-offloading:** It means that all input data is sent to one of the edge servers. This server executes all the workload and returns the result to the local. We use *Full-CPU* and *Full-GPU* to denote the full offloading using the Intel i7-3615QM CPU and NVIDIA GTX 1080 GPU as the computation unit, respectively.
- **No-offloading:** All workloads are executed in the mobile device without offloading.

The SKMD, Greed, and Hyperion have implemented distributed computing for multiple servers. Besides these baselines, we also implemented a DNN custom offloader, SPINN [33], for performance comparison, which will be described in the following subsection.

Metric: We calculate the speedup values of these methods by comparing their overall time with the no-offloading scheme to present the performance gains among baselines.

5.3 Experimental Results

5.3.1 Evaluation under Dynamic Network Conditions.

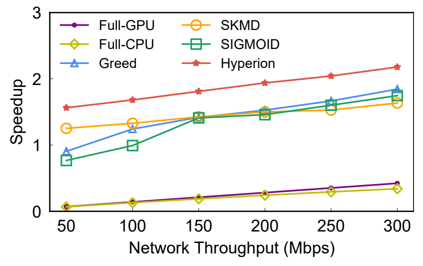
Setups: We walk around our building on the campus. Our walking is performed at the speed of 5.0 km/h by taking the Xiaomi Blackshark 2. This mobile consecutively communicates with edge servers we have deployed in this building through Wi-Fi connections. We have recorded network throughput values for each position, which are shown in Figure 12. The average latency to our server is 14.5 ms. Then we conduct trace-driven experiments by replaying the network throughput sequence, which is implemented by using the TC command [7], a widely used network traffic control tool. For each test case, this replay is recycled for 24 hours.

Experimental Results: Figures 13 and 14 show these results. The results contain the average system speedup values and balance difference. We define the balance difference that $max(\mathcal{F}) - min(\mathcal{F})$, where $\mathcal{F}$ is the vector containing the execution completion time of the last slice for each device. We find that Hyperion can flexibly adapt to dynamic conditions and achieve average 2.28×, 2.26×, 2.27×, and 3.32× speedup (2.53× on average) compared with no-offloading, SKMD, Greed, and SIGMOID, respectively. The SKMD cannot cope with runtime network changes because it makes scheduling decisions before execution and cannot adjust workloads for each backend device at runtime. Thus, it has high balance difference values. The Greed works better in some cases (e.g., GEMM) and worse in others (e.g., AddWeighted). This is because of different computational overheads for kernels. The SIGMOID does not consider the network transmission costs. In contrast, Hyperion maintains the balanced loads between the local and edge servers, and thus achieves high performance and low balance difference.

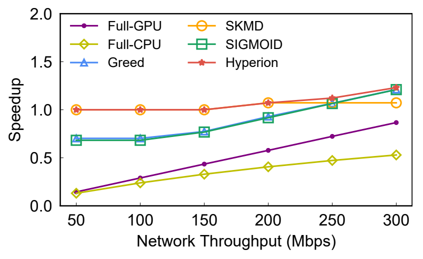
5.3.2 Performance under different network throughput.

Setups: In this section, we take the network throughput as the case study to show the performance of Hyperion without the interference of dynamic network conditions. Figure 15 shows the experimental results, where network throughput is limited to 50 − 300 Mbps with 10 ms latency by using the TC command.

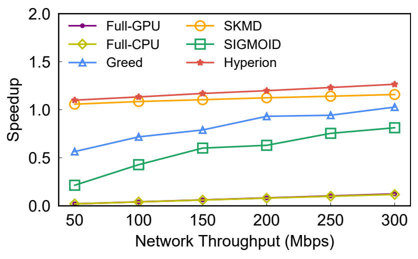
System Performance of Hyperion: Hyperion has the best performance among baselines, which can outperform no-offloading, SKMD, Greed, and SIGMOID with average 5.80×, 3.80×, 4.36×, and5.05× speedup, respectively (4.75× speedup on average for this three baselines). Intuitively, the improvement of Hyperion is not much relative to the sub-optimal method, which is 10.7% on average. However, this sub-optimal method is different across network throughputs and kernel types, and this sub-optimal method may turn into the worst case in the other conditions. For example, the sub-optimal method of CvtColor at 50 Mbps is SKMD, but as the network throughput increase, the sub-optimal method turns to the Greed method. Hyperion can achieve the best performance under all network throughput and kernel types. In addition, the speedup of Hyperion differs across kernels and network throughput. For example, under 100 Mbps, Hyperion achieves 3.80× speedup compared



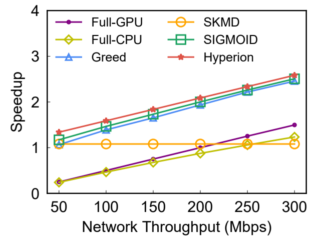
(a) CvtColor



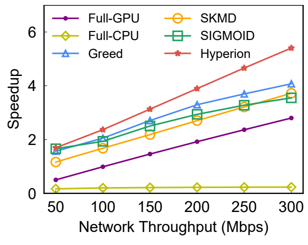
(b) WarpAffine



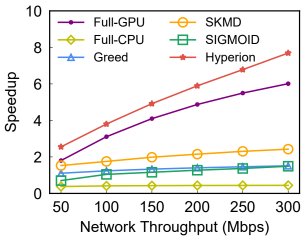
(c) AddWeighted



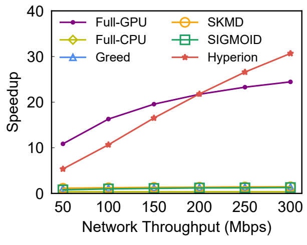
(d) StitchingWarper



(e) GEMM



(f) YOLOv4-tiny



(g) ResNet-50

Figure 15: Evaluation results of speedup of local GPU (higher is better) with respect to different network throughput.

with the no-offloading method in YOLOv4-tiny while no speedup occurs in WarpAffine. This is because WarpAffine requires large data transmissions (94.9 MB of common data). On this occasion, the local GPU completes all workloads before Hyperion finishes common data transmission. As the increase of network throughput, Hyperion will gradually assign more workload to edge servers for better performance. For example, 1.23× speedup occurs in the WarpAffine under 300 Mbps. In ResNet-50, the performance of Hyperion is not so good as full-GPU. This is because Hyperion lacks the global perspective of kernel structures in multi-kernel applications. As Hyperion focuses on OpenCL-level scheduling, it tends to fall into the local optimal for the current kernel, ignoring transmission scheduling for future kernels. Thus, the performance may degrade compared with application-specific methods (e.g., SPINN in section 5.3.5) and full offloading under some conditions.

Comparing other baselines: (1) Full-offloading method transmits all input data to the backend device, which suffers high network overhead. In most cases, the performance even cannot outperform the no-offloading method. In contrast, Hyperion can flexibly adjust the workloads assigned to each device to fully utilize the resources of network and computing units, and thus outperforms this method. (2) Compared with SKMD, Hyperion implements pipeline processing, where data transmissions and execution can be parallelized for better performance. A concern may be raised that Hyperion partition the kernel into several small workloads and thus introduce substantial computational overhead. Hyperion can control and offset this overhead in pipeline processing, thus achieving high performance. (3) Greed method partitions the kernel into the same number of slices (i.e., 64 in our experiment) and organizes pipeline processing. This slice number is suitable for some kernels (i.e., StitchingWarper), which incurs less computational overhead and high pipeline parallelism but cannot fit all kernels. For example, in AddWeighted, the substantial computational overhead makes the Greed method even cannot outperform the no-offloading method. (4) Although SIGMOID targets high load balance, it does not consider the data transmission costs and thus suffers high latency.

5.3.3 *Performance of Distributed Computing.*

In this experiment, we evaluate the performance of Hyperion with the different number of servers. We employ five edge servers, each using Intel i9-10940X CPU as the backend device. Furthermore, we use GEMM as a case study, and the results are summarized in Figure 16. When the server number is small, the performance increases as the server number increases. But when a specific value is reached, the performance arrives at the peak and cannot increase further. In this case, the network throughput becomes the performance bottleneck. For example, at 50 Mbps, all workloads are executed locally, regardless of the server number, as throughput is the bottleneck. At 150 Mbps, the performance increases when using 1 to 3 servers, and the performance no longer increases when using more than 3 servers because the network throughput becomes the bottleneck.

5.3.4 *Component-wise evaluations.*

Kernel Analyzer: We have evaluated the performance of Hyperion with or without the kernel analyzer, and Figure 17 shows the result. The network throughput is 200 Mbps in this evaluation. The analyzer partitions the kernel data and enables the partial data transmission feature to reduce network costs. Without the analyzer, the performance of Hyperion has an average 35.3% reduction in our evaluation. We also find that some applications (e.g., WA and SW) have no performance gain. This is because the analyzer considers that all data in these kernels are common data and cannot partition.

Lazy-Transmission Pipeline: This evaluation shows the performance gains in the lazy-transmission pipeline. To achieve this, we first design a multi-kernel application that consists of several convolution kernels. We can manually adjust the kernel number. Then, the execution times with and without the lazy-transmission pipeline are obtained and summarized in Figure 18. The network throughput is 200 Mbps. Without the lazy-transmission pipeline, all data need to transmit through the network, causing the delay.

Predictor: First, we evaluate the accuracy of the predictor and some other methods (i.e., LGBM [29], XGBoost[11]) with offline

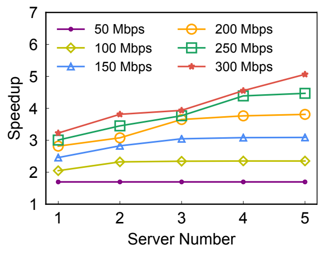


Figure 16: Impact of the server number and network throughput.

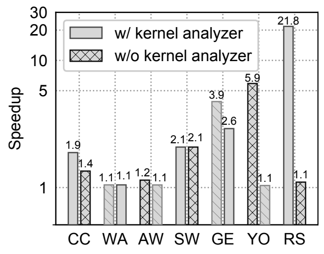


Figure 17: Performance of Hyperion with and without the kernel analyzer.

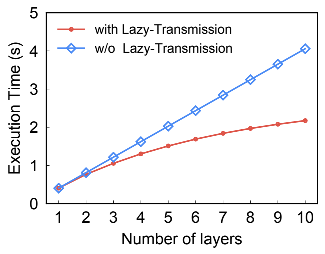


Figure 18: Performance of Hyperion with or without the lazy-transmission pipeline.

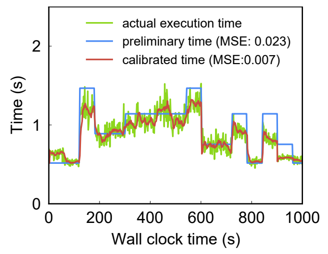


Figure 19: The predicted time under dynamic system loads (lower MSE is better)

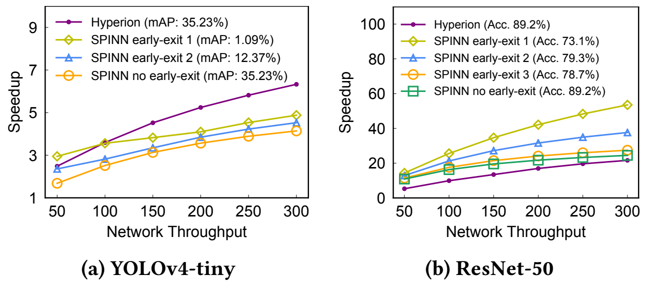


Figure 20: Performance of Hyperion and SPINN. The mAP and Acc. is the accuracy (higher is better)

data, which have 0.061, 0.078, 0.083 in mean square error, respectively. Hyperion is more accurate in offline training. Second, we conduct an online evaluation to show online accuracy. We take GEMM as a case study and record the prediction results under dynamic system loads. The dynamic system loads are simulated through the way that other mobiles concurrently send computing requests to the server, and we randomly change the loads every minute. Figure 19 shows the result. The preliminary time only considers concurrent task numbers and types, ignoring the actual resource status at runtime. Thus, after the calibration of Hyperion, the accuracy can be greatly improved.

5.3.5 *Comparison with the DNN Offloading Method.*

Setups: Hyperion is applicable to general OpenCL applications, including DNN inference. In this experiment, we take YOLOv4-tiny and Resnet-50 as the case study and compare Hyperion with SPINN [33], the state-of-the-art offloading framework for DNN inference. SPINN proposed a progressive inference strategy that selected the *early-exit point* and *offloading point* according to the timeliness requirement and network conditions. Early-exit means skipping some DNN layers according to the timeliness requirement and feeds the intermediate results into the classifier to get a coarse result. Although the performance is improved, the result is less accurate than that of the original model. The offloading point partitions the DNN network into two parts: the first part is executed locally, then SPINN transmits the intermediate data to the server, and finally, the server executes the remaining part. As SPINN did not implement distributed computing across backend devices, we use a single edge server (i.e., GTX 1080) for fair comparisons. The YOLOv4-tiny has two early-exit points at layers 10 and 18, respectively, and Resnet-50 has three early-exit points at layers 11, 23, and 41, respectively. To have a better comparison between these early-exit policies and Hyperion, we test each of them and the no early-exit point cases, respectively. Figure 20 shows the results.

Experimental Results: In YOLOv4-tiny, Hyperion outperforms SPINN in all conditions except when SPINN takes early-exit point 1 at 50 Mbps. In this case, Hyperion still outperforms SPINN in terms of accuracy because we do not modify the model. However, in Resnet-50, SPINN outperforms Hyperion at all times. This is because Hyperion tends to fall into the local optimal in multi-kernel applications, which we have discussed in Section 5.3.2. The application-specific method can obtain more information about the structures, and thus can have better performance.

6 DISCUSSION

Applicable scenarios and applications. Hyperion is a generic framework for mobile offloading. Developers do not have to design application-specific scheduling algorithms for every application to achieve high-performance offloading, which can save developers' engineering efforts. In terms of applicable application types, Hyperion shows performance advantages in computing-intensive applications. Thus, the processing of high-resolution images is a typical application scenario. Apart from this, Hyperion is also suitable for linear algebraic algorithms, such as correlation coefficients and matrix multiplication. Although audio processing and NLP applications are also computing-intensive, the data size of audio streams or texts is usually small. Thus, full offloading to the server is a good choice. In this condition, the performance of Hyperion is similar to that of full offloading because Hyperion also offloads all workloads to the edge server. Before using Hyperion, developers can roughly estimate performance gain through the execution time on each device and the data transmission time, thus determining whether it is worth using Hyperion.

Advantages. Hyperion is designed with the goal of high performance and heterogeneity-compatible. It comprehensively judges the impact of network conditions, device computing capability, and kernel characteristics on performance, and chooses the ideal offloading strategy. Thus, performance improvement is affected by a variety of factors. Generally speaking, Hyperion has higher performance when it has better network conditions, higher computing capacity of edge devices, fewer data amounts, and more computation amounts of the kernel. Under the inferior cases, Hyperion computes tasks in the mobile GPU. Some existing methods can be used to mitigate the burden of GPU, such as input compressing for image processing applications and model compression or early-exit technique for DNN-based applications.

Limitations. Hyperion is unsuitable for irregular OpenCL kernels, where WGs have inconsistent workloads. For example, in ray tracing, the workload of each WG depends on the reflection and refraction times. Meanwhile, it is still a challenge to predict the computing time of such kernels at low overhead. In addition, kernels with global barriers or atomic operations are not recommended in Hyperion. For the global barrier, kernels should be organized into Bulk Synchronous Parallel (BSP) form [10]. Specifically, the kernel is split into two kernels with a synchronization point in the middle. Synchronization of data is required when all devices reach this point before continuing to the other kernel. For the atomic operation, the modification of local memory from one device cannot immediately be seen by another device, given that our WGs are distributed across different devices. Developers need to know the reason for atomic operations and transfer the kernel to BSP form.

Requirements for developers. Compatibility is one of design principles of Hyperion, i.e., developers can use Hyperion without modifying the kernel codes. When designing new applications, developers need to follow the official manual of OpenCL, know the basic principle of designing OpenCL kernels, and exploit the massive parallelism of multi-core computing devices. After that, developers need to conduct offline training of Hyperion to have a better prediction performance during task scheduling. Hyperion can automatically make scheduling decisions during online execution.

7 RELATED WORK

Scheduling strategy for data parallelism. Several works were devoted to scheduling the data-parallel kernel to other devices to alleviate exhausted GPU [8, 41, 44, 53]. These devices are generally with inconsistent computing capacity. Pandit et al. [41] proposed a dynamic method in which work-groups were flatted into a one-dimension vector, and then, CPU and GPU started their work from two ends of the vector. For integrated CPU/GPU architectures, Zhang et al. [53] proposed a static and machine learning-based method to distribute tasks to the most suitable device. For embedded FPGA, Rodrıguez et al. [44] designed a shared memory architecture and unified access to mitigate the transmission overhead. However, all of them are designed for offloading to specific devices, which is not generic enough and unsuitable for mobile offloading scenarios.

Heterogeneity-adaptive offloading mechanism. Optimizing the performance under heterogeneous conditions was also studied by many works. Zhou et al. [56] classified heterogeneous networks into three types: cloud, cloudlet, and mobile ad-hoc cloud. Offloading strategies could be automatically selected based on the resources of different types of networks. Wang et al. [49] proposed an offloading method for video object detection, which could dynamically adjust video configurations according to real-time network conditions. Hao et al. [19] studied the heterogeneities of computing capacity and storage capacity on edge server and proposed an offloading mechanism to reduce system delay. However, these works are not designed for data-parallel applications, which can be further partitioned and optimized for collaborative computing.

Performance Prediction. Performance prediction plays an important role in task scheduling to ensure timeliness. Some works simply use FLOPs or MAC values to predict[21, 47], which are inaccurate because they ignore the actual runtime state of devices. Some other works are application-specific, including the prediction for data analytics tasks [17, 40] and neural network tasks [54, 57]. Similar to Hyperion, some works [28, 31, 54] use historical data and build a regression model to predict the performance. However, they do not consider the system load variation during the prediction, which may incur errors in the result.

Pipeline optimization. Pipeline optimization is a classic method and is extensively used in many areas. Streaming applications are the typical usage scenarios [14, 16, 26], and the pipeline technique is used to overlap the data transmission costs among computing nodes. For DNN inference tasks, pipeline technique mitigates the insufficient hardware resources of resource-constrained devices[22, 27], for example, executing large-size model that exceeds the memory limit[22]. In the domain of neural network training, two kinds of pipeline techniques implement efficient distributed training. Layer-wise pipeline [38, 46] assigns different sub-sequences of layers on separate computing nodes, while distributed tensor pipeline[25, 45] focuses on a single tensor partition and assignment. In our work, we have improved the pipeline technique (i.e., the lazy-transmission pipeline) to better work with multi-kernel applications.

Distributed offloading mechanism. Offloading workloads to distributed edge devices can also be frequently found in the literature. Gong [15] improved the performance by finding an optimal workload allocation and communication order. Im et al. [24] proposed a new scheduler for non-preemption workloads. Ning et al. [39] enabled multi-edge cooperation by jointly considering constraints of storage capacity and execution delay. However, they are all application-level scheduling solutions, which cannot be directly applied to optimize a single inefficient data-parallel kernel.

8 CONCLUSION

In this paper, we propose Hyperion, which takes the first step towards generic and distributed mobile offloading. To achieve high-performance distributed computing, Hyperion integrates three techniques of addressing the challenges in workload partition and scheduling at the OpenCL layer. Specifically, the regular-aware kernel analyzer can identify the data dependency of WGs and discover the necessary data to offload, while the context-aware predictor is capable of perceiving the dynamic runtime status to predict the computing time of a given slice. Based on them, the pipeline-enabled and network-adaptive scheduler is designed to make offloading decisions with the aim of forming pipeline processing and adapting to network dynamic. Experimental results show that Hyperion can achieve an average 3.80× speedup compared with baselines and is highly adaptive to dynamic network conditions and system loads.

ACKNOWLEDGMENT

This research was supported in part by the National Key R&D Program of China under Grant No. 2022YFF0604502, the National Natural Science Foundation of China under Grant No. 62122095, 62072472, U19A2067, and 62172439, the Guoqiang Institute, Tsinghua University and the Natural Science Foundation of Hunan Province, China under Grant No. 2020JJ2050, Tsinghua University - AsiaInfo Technologies (China) Inc. Joint Research Center under Grant No. 20203910074, the Key-Area Research and Development Program of Guangdong Province under Grant No. 2019B010137005.

REFERENCES

[1] 2013. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opencl/.

[2] 2021. Multi-Process Service. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf

[3] 2022. Android NDK | Android Developers. https://developer.android.com/ndk.

[4] 2022. gRPC. https://grpc.io/.

[5] 2022. TensorFlow Lite. https://www.tensorflow.org/lite?hl=zh-cn.

[6] Alexey. 2022. Darknet: Open Source Neural Networks in Python. https://github.com/AlexeyAB/darknet.

[7] Werner Almesberger. 1999. *Linux Network Traffic Control—Implementation Overview.*

[8] Mehmet E. Belviranli, Laxmi N. Bhuyan, and Rajiv Gupta. 2013. A Dynamic Self-Scheduling Scheme for Heterogeneous Multiprocessor Architectures. *ACM Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 1–20. https://doi.org/10.1145/2400682.2400716

[9] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal: Software Tools for the Professional Programmer* 25, 11 (2000).

[10] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. 1996. Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software. In *Tools and Environments for Parallel and Distributed Systems*, Amr Zaky and Ted Lewis (Eds.). Springer US, Boston, MA, 61–76.

[11] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[12] Thanh Tuan Dao and Jaejin Lee. 2018. An Auto-Tuner for OpenCL Work-Group Size on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 29, 2 (Feb. 2018), 283–296. https://doi.org/10.1109/TPDS.2017.2755657

[13] Ziyan Fu, Yuezhi Zhou, Chao Wu, and Yaoxue Zhang. 2021. Joint Optimization of Data Transfer and Co-Execution for DNN in Edge Computing. In *ICC 2021 - IEEE International Conference on Communications*. 1–6. https://doi.org/10.1109/ICC42927.2021.9500513

[14] Buğra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. 2014. Elastic Scaling for Data Stream Processing. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (June 2014), 1447–1463. https://doi.org/10.1109/TPDS.2013.295

[15] Xiaowen Gong. 2020. Delay-Optimal Distributed Edge Computing in Wireless Edge Networks. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2629–2638. https://doi.org/10.1109/INFOCOM41043.2020.9155272

[16] Michael I. Gordon, William Thies, and Saman Amarasinghe. 2006. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. *ACM SIGOPS Operating Systems Review* 40, 5 (Oct. 2006), 151–162. https://doi.org/10.1145/1168917.1168877

[17] Andrea Gulino, Arif Canakoglu, Stefano Ceri, and Danilo Ardagna. 2020. Performance Prediction for Data-driven Workflows on Apache Spark. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8. https://doi.org/10.1109/MASCOTS50786.2020.9285944

[18] Mengxi Hanyao, Yibo Jin, Zhuzhong Qian, Sheng Zhang, and Sanglu Lu. 2021. Edge-Assisted Online On-device Object Detection for Real-time Video Analytics. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*. 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488741

[19] Hao Hao, Changqiao Xu, Lujie Zhong, and Gabriel-Miro Muntean. 2020. A Multi-update Deep Reinforcement Learning Algorithm for Edge Computing Service Offloading. In *Proceedings of the 28th ACM International Conference on Multimedia (MM '20)*. Association for Computing Machinery, New York, NY, USA, 3256–3264. https://doi.org/10.1145/3394171.3413702

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, Las Vegas, NV, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[21] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 784–800.

[22] Xueyu Hou, Yongjie Guan, Tao Han, and Ning Zhang. 2022. DistrEdge: Speeding up Convolutional Neural Network Inference on Distributed Edge Devices. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1097–1107. https://doi.org/10.1109/IPDPS53621.2022.00110

[23] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. DeepMon: Mobile GPU-based Deep Learning Framework for Continuous Vision Applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '17*. ACM Press, Niagara Falls, New York, USA, 82–95. https://doi.org/10.1145/3081333.3081360

[24] S. Im, M. Naghshnejad, and M. Singhal. 2016. Scheduling Jobs with Non-Uniform Demands on Multiple Servers without Interruption. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 1–9. https://doi.org/10.1109/INFOCOM.2016.7524417

[25] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. *Proceedings of Machine Learning and Systems* 1 (April 2019), 1–13.

[26] Basri Kahveci and Buğra Gedik. 2020. Joker: Elastic Stream Processing with Organic Adaptation. *J. Parallel and Distrib. Comput.* 137 (March 2020), 205–223. https://doi.org/10.1016/j.jpdc.2019.10.012

[27] Daniel Kang, Ankit Mathur, Teja Veeramacheneni, Peter Bailis, and Matei Zaharia. 2020. Jointly Optimizing Preprocessing and Inference for DNN-based Visual Analytics. arXiv:2007.13005 [cs]

[28] 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. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, Xi'an China, 615–629. https://doi.org/10.1145/3037697.3037698

[29] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems*, Vol. 30. Curran Associates, Inc.

[30] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. 2012. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing - ICS '12*. ACM Press, San Servolo Island, Venice, Italy, 341. https://doi.org/10.1145/2304576.2304623

[31] 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 *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, Dresden Germany, 1–15. https://doi.org/10.1145/3302424.3303950

[32] 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. https://doi.org/10.1109/IPSN.2016.7460664

[33] Stefanos Laskaridis, Stylianos I. Venieris, Mario Almeida, Ilias Leontiadis, and Nicholas D. Lane. 2020. SPINN: Synergistic Progressive Inference of Neural Networks over Device and Cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking (MobiCom '20)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3372224.3419194

[34] C. Lattner and V. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[35] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. 2015. SKMD: Single Kernel on Multiple Devices for Transparent CPU-GPU Collaboration. *ACM Transactions on Computer Systems* 33, 3 (Aug. 2015), 1–27. https://doi.org/10.1145/2798725

[36] Jing Li, Wei Xu, Jianguo Zhang, Maojun Zhang, Zhengming Wang, and Xuelong Li. 2015. Efficient Video Stitching Based on Fast Structure Deformation. *IEEE Transactions on Cybernetics* 45, 12 (2015), 2707–2719. https://doi.org/10.1109/TCYB.2014.2381774

[37] Sparsh Mittal and Jeffrey S. Vetter. 2015. A Survey of CPU-GPU Heterogeneous Computing Techniques. *Comput. Surveys* 47, 4 (July 2015), 1–35. https://doi.org/10.1145/2788396

[38] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized Pipeline Parallelism for DNN Training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, Huntsville Ontario Canada, 1–15. https://doi.org/10.1145/3341301.3359646

[39] Z. Ning, P. Dong, X. Wang, S. Wang, X. Hu, S. Guo, T. Qiu, B. Hu, and R. Y. K. Kwok. 2021. Distributed and Dynamic Service Placement in Pervasive Edge Computing Networks. *IEEE Transactions on Parallel and Distributed Systems* 32, 6 (June 2021), 1277–1292. https://doi.org/10.1109/TPDS.2020.3046000

[40] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for Performance Clarity in Data Analytics Frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 184–200. https://doi.org/10.1145/3132747.3132766

[41] Prasanna Pandit and R. Govindarajan. 2014. Fluidic Kernels: Cooperative Execution of OpenCL Programs on Multiple Heterogeneous Devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization - CGO '14*. ACM Press, Orlando, FL, USA, 273–283. https://doi.org/10.1145/2581122.2544163

[42] Borja Pérez, José Luis Bosque, and Ramón Beivide. 2016. Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit - GPGPU '16*. ACM Press, Barcelona, Spain, 42–51. https://doi.org/10.1145/2884045.2884051

[43] Borja Pérez, E. Stafford, J.L. Bosque, and R. Beivide. 2021. Sigmoid: An Auto-Tuned Load Balancing Algorithm for Heterogeneous Systems. *J. Parallel and Distrib. Comput.* 157 (Nov. 2021), 30–42. https://doi.org/10.1016/j.jpdc.2021.06.003

[44] Alfonso Rodrıguez, Juan Valverde, and Eduardo de la Torre. 2015. Design of OpenCL-compatible Multithreaded Hardware Accelerators with Dynamic Support for Embedded FPGAs. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, Riviera Maya, Mexico, 1–7. https://doi.org/10.1109/ReConFig.2015.7393297

[45] Noam Shazeer, Youlong Cheng, Niki Parmar, Dustin Tran, Ashish Vaswani, Penporn Koanantakool, Peter Hawkins, HyoukJoong Lee, Mingsheng Hong, Cliff Young, Ryan Sepassi, and Blake Hechtman. 2018. Mesh-TensorFlow: Deep Learning for Supercomputers. In *Advances in Neural Information Processing Systems*, Vol. 31. Curran Associates, Inc.

[46] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs]

[47] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[48] Robert A. van Engelen. 2001. Efficient Symbolic Analysis for Optimizing Compilers. In *Compiler Construction (Lecture Notes in Computer Science)*, Reinhard Wilhelm (Ed.). Springer, Berlin, Heidelberg, 118–132. https://doi.org/10.1007/3-540-45306-7_9

[49] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, and M. Xiao. 2020. Joint Configuration Adaptation and Bandwidth Allocation for Edge-based Real-time Video Analytics. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 257–266. https://doi.org/10.1109/INFOCOM41043.2020.9155524

[50] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, Stephen McCamant, Youfeng Wu, and Jayaram Bobba. 2017. Enabling Cross-ISA Offloading for COTS Binaries. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '17*. ACM Press, Niagara Falls, New York, USA, 319–331. https://doi.org/10.1145/3081333.3081337

[51] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. 2020. DeepWear: Adaptive Local Offloading for On-Wearable Deep Learning. *IEEE Transactions on Mobile Computing* 19, 2 (2020), 314–330. https://doi.org/10.1109/TMC.2019.2893250

[52] Lei Yang, Jiannong Cao, Shaojie Tang, Di Han, and Neeraj Suri. 2016. Run Time Application Repartitioning in Dynamic Mobile Cloud Environments. *IEEE Transactions on Cloud Computing* 4, 3 (July 2016), 336–348. https://doi.org/10.1109/TCC.2014.2358239

[53] Feng Zhang, Jidong Zhai, Bingsheng He, Shuhao Zhang, and Wenguang Chen. 2017. Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (March 2017), 905–918. https://doi.org/10.1109/TPDS.2016.2586074

[54] 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 *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. Association for Computing Machinery, New York, NY, USA, 81–93.

[55] 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. https://doi.org/10.1109/TCAD.2018.2858384

[56] Bowen Zhou, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, Satish Narayana Srirama, and Rajkumar Buyya. 2017. mCloud: A Context-Aware Offloading Framework for Heterogeneous Mobile Cloud. *IEEE Transactions on Services Computing* 10, 5 (Sept. 2017), 797–810. https://doi.org/10.1109/TSC.2015.2511002

[57] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for {DNN} Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 337–352.