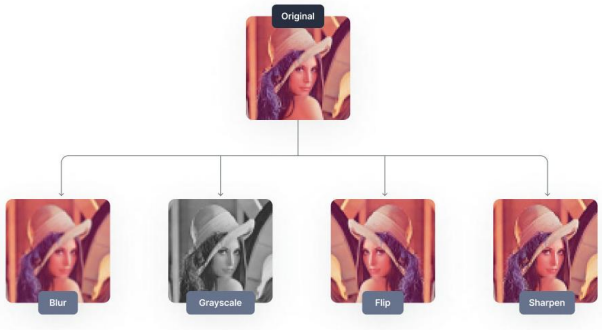
# Hyperion: A Generic and Distributed Mobile Offloading Framework on OpenCL

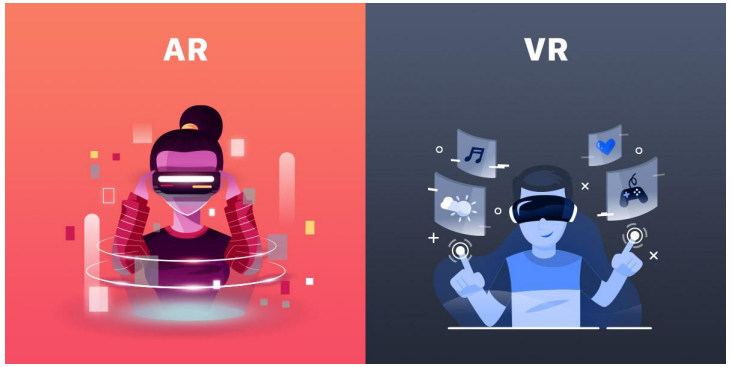
Sensys'22

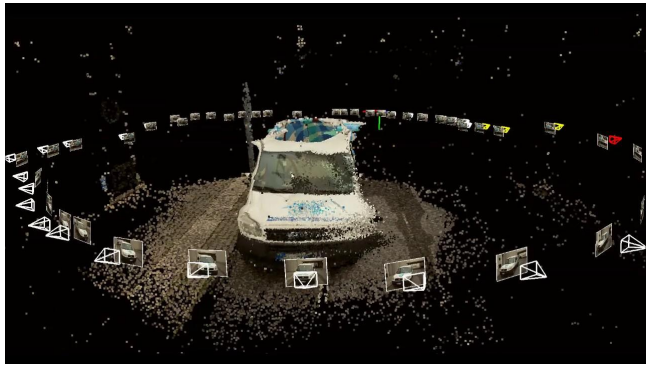
Ziyan Fu[1], Ju Ren[1,5,*], Yunxin Liu[2,6], Ting Cao[3], Deyu Zhang[4], Yuezhi Zhou[1,5], Yaoxue Zhang[1,5]
[1]Department of Computer Science and Technology, BNRist, Tsinghua University, Beijing, China
[2]Institute for AI Industry Research (AIR), Tsinghua University, Beijing, China, [3]Microsoft Research, 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

# Introduction

- More computation-intensive and real-time task processing
  - High resolution image processing, AR/VR, 3D reconstruction
- High image processing burden on mobile devices
- Affects the user experience
  - takes 4.39 s to perform 4K panoramic picture stitching  and 67.4 s for 4K image recognition



# Introduction - Solutions

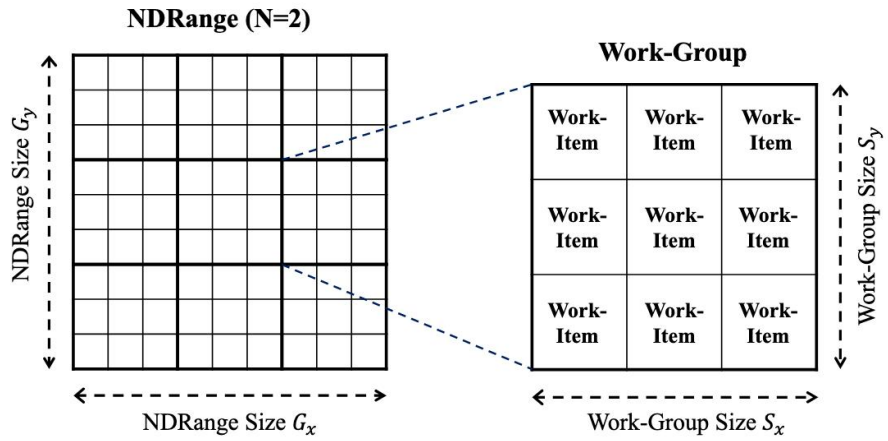
- Compress the input images
  - low accuracy or low quality of output images
- Designing new algorithms or Compressing DNN models
  - Impossible to balance the accuracy and computation efficiency
- **Offloading**
  - Partition the computation task
  - Offload the computationally intensive parts
  - Specific tasks or targeted at specific **software/hardware platforms.**
  - Seek a more **fundamental** and **generic** solution for handling the heterogeneous tasks on different platforms.

# Introduction - OpenCL™

- **Offloading**
  - Specific tasks or targeted at specific **software/hardware platforms**.
  - Seek a more **fundamental** and **generic** solution for handling the heterogeneous tasks on different platforms.
- **OpenCL**
  - **Cross-platform**
  - **Parallel** programming framework for general-purpose computation
  - Widely supported by **diverse applications and hardware**

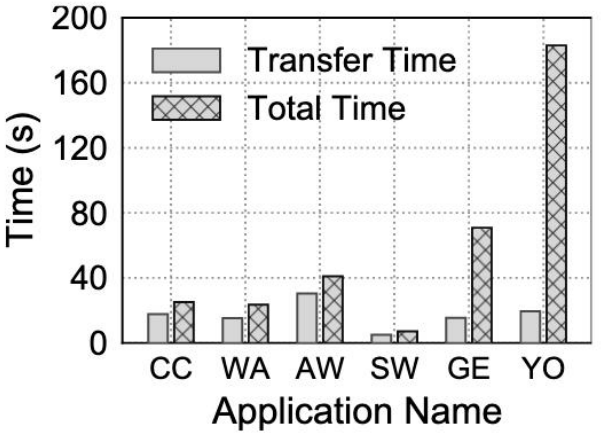
# Introduction - The idea of Hyperion

- Host & kernel
- Work items & Work group & NDRange
- **Partition the OpenCL kernels of each task**
- **Offload part of the WGs to nearby edge servers**
  - kernel(many WGs: fixed)
  - slice(some WGs)
- Fully utilizing both local and edge servers
- **Face many challenges**



# Introduction - Challenges

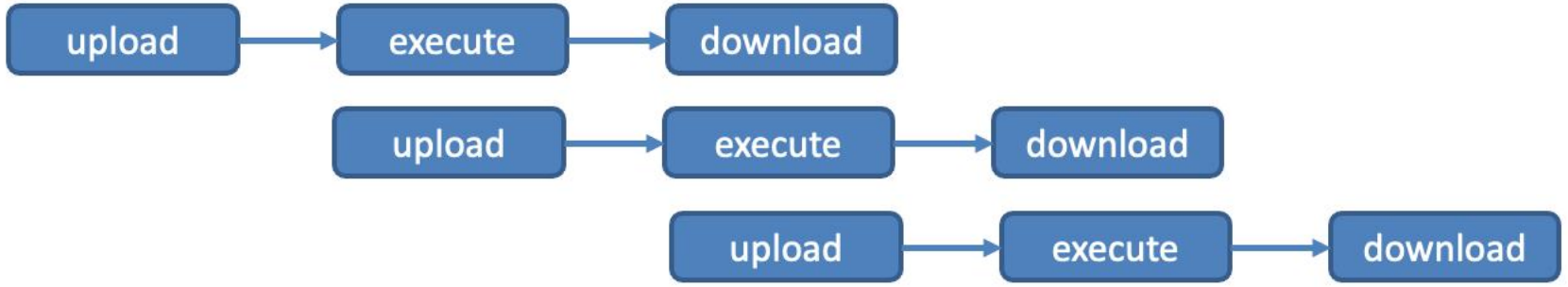
**Challenge 1: High data transmission cost for kernel-level offloading**

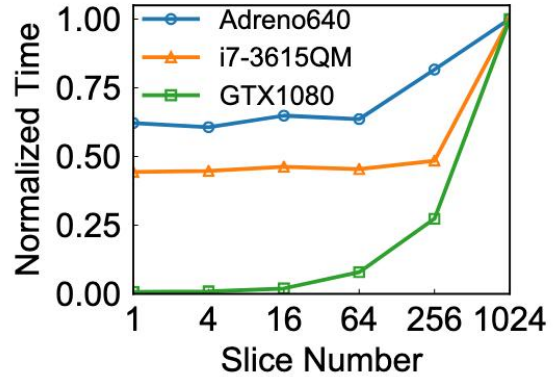




May be less efficient because of the **data transfer overhead**

**Solution: build a processing pipeline**

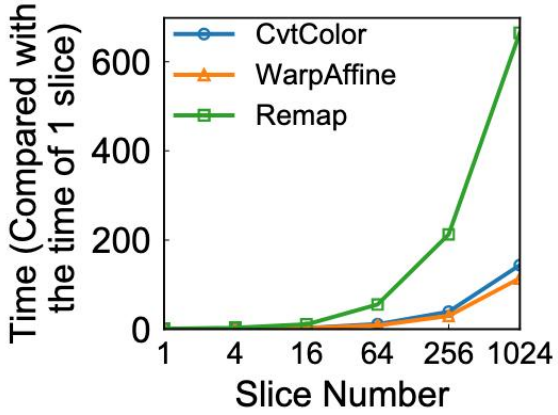
**Parallelize computation and data transmission**




**Fewer slice:** low pipeline performance will occur

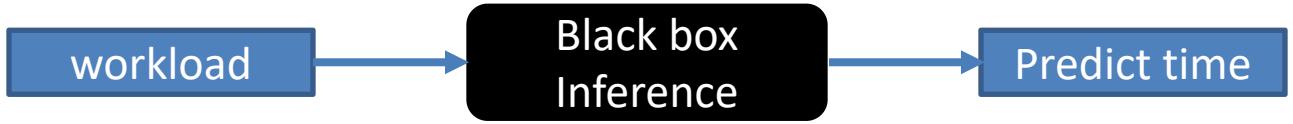
**More slice:** tremendous computational overhead will occur



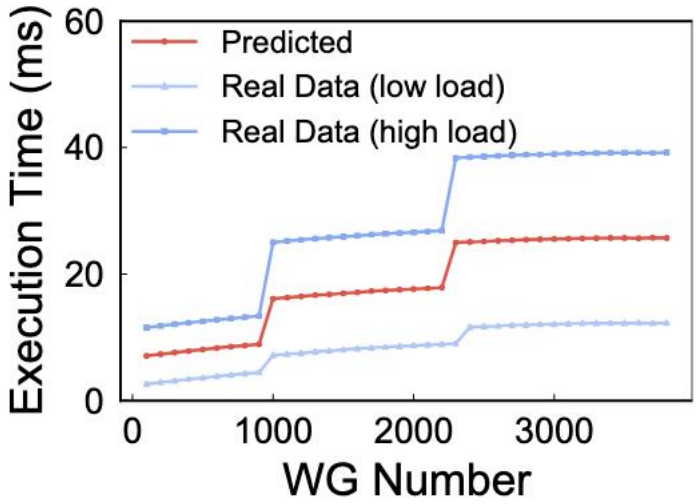
# Introduction - Challenges

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

- **Require an evaluation of the execution time based on device performance**
- **The kernel execution time is affected by many factors**
  - kernel implementation,
  - cache contention,
  - memory coalescing,
  - device load
  - workload scheduling
- **Black box inference**



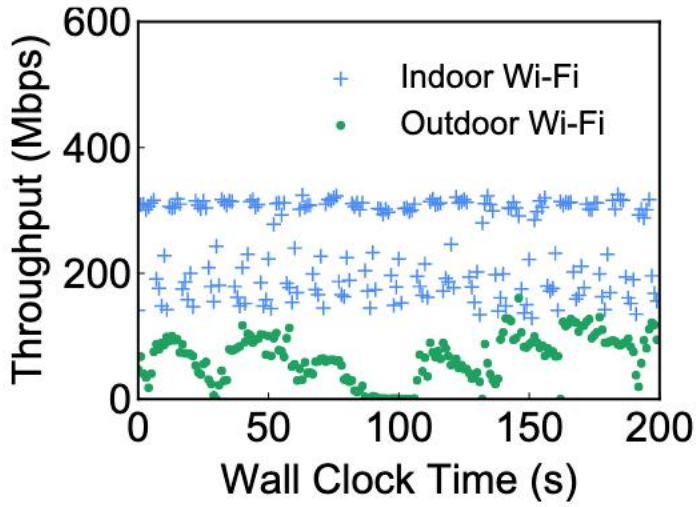

- **Ignoring the runtime status, especially the real-time load**



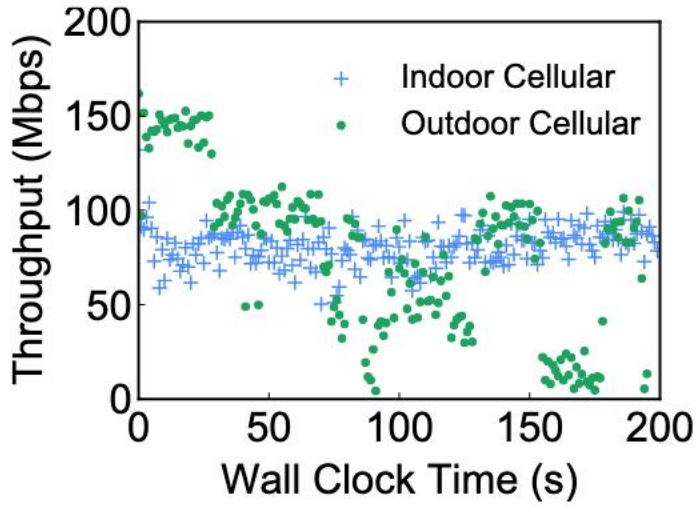
# Introduction - Challenges

## Challenge 3: Highly-dynamic network conditions

- **Dramatic network throughput fluctuations for both indoor and outdoor cases**
- **Variation of signal strength and the network congestion**
- **Need to consider the risk of network fluctuations when making scheduling decisions**




(a) Wi-Fi




(b) Cellular

# System Overview - Hyperion

- Offers a generic and distributed offloading framework for various applications and works as a library in OpenCL
- Hyperion automatically makes scheduling decisions and offloads parts of the workload to multiple edge servers

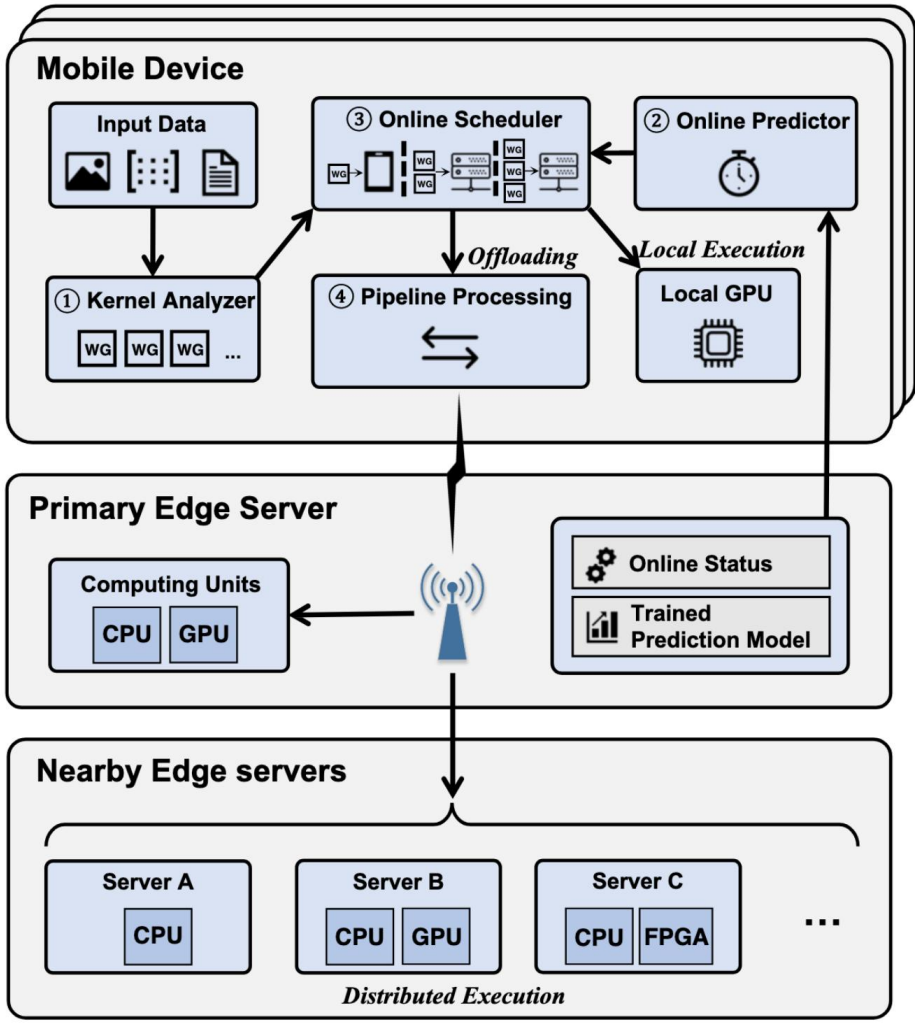



Figure 7: System overview of Hyperion.

# System Design

## Regularity-Aware Kernel Analyzer

- **Original opencl kernel: execute in a single device**
- **Two or more device?**
  - **Cannot efficiently share a global memory**
  - Transmit the whole input?
  - Asynchronous transmission?
  - **Hyperion design: identify the data dependency of different WGs**
- **Classify the memory data of a kernel into two types:**
  - **common data: frequently accessed**
  - **exclusive data: only used by one WG**

**How to identify the common and exclusive data?**

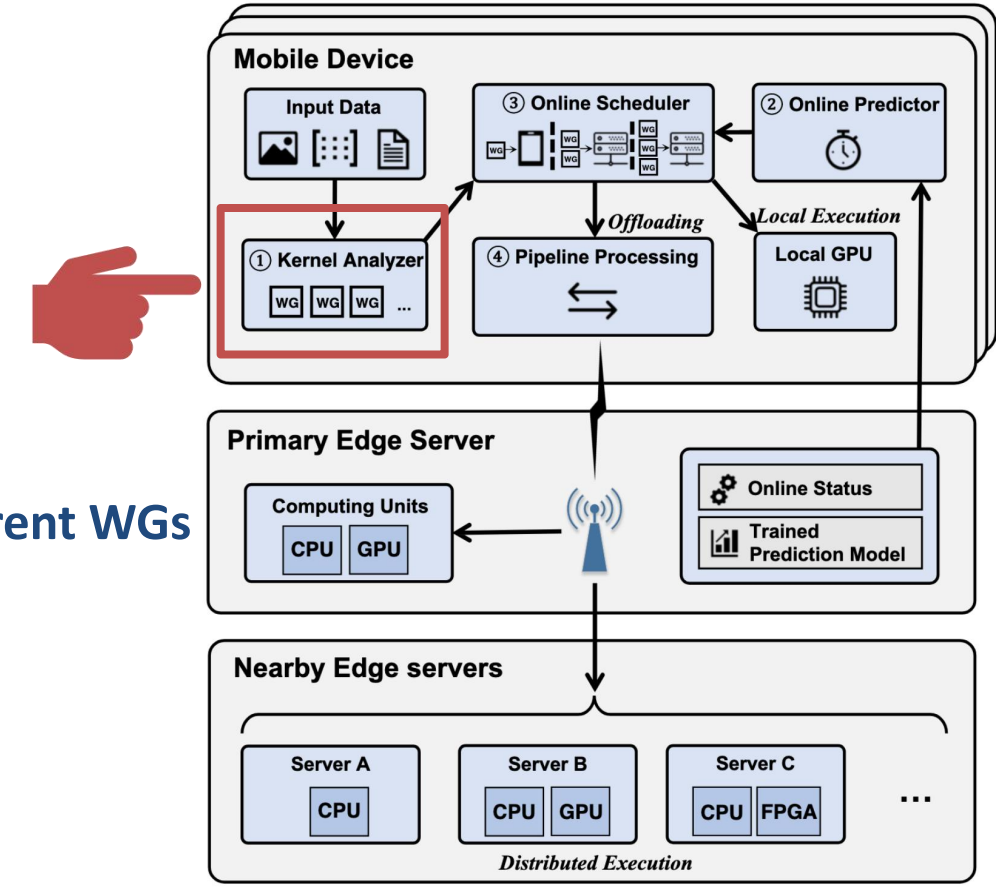



Figure 7: System overview of Hyperion.

# System Design

## Regularity-Aware Kernel Analyzer

**How to identify the common and exclusive data?**

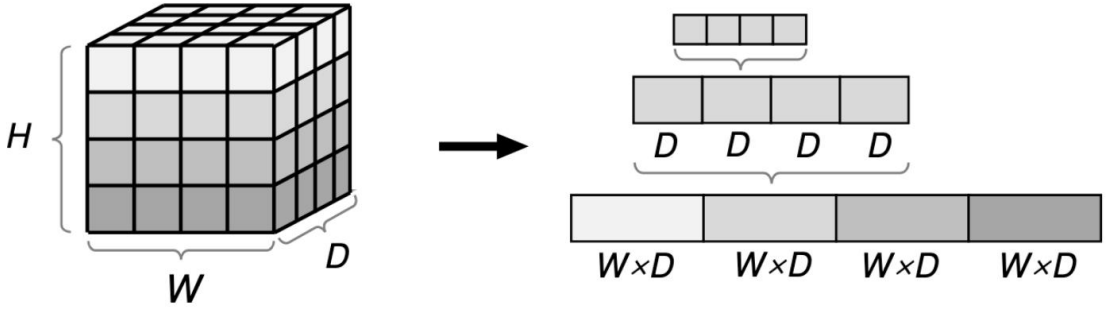
**According to its data access pattern**

**Two data access pattern:**

- **regular ways** eg: image processing (pixel by pixel)
- **irregular ways** eg: linear algebra computation

**regular access data -> exclusive data**

**irregular access data -> common data**

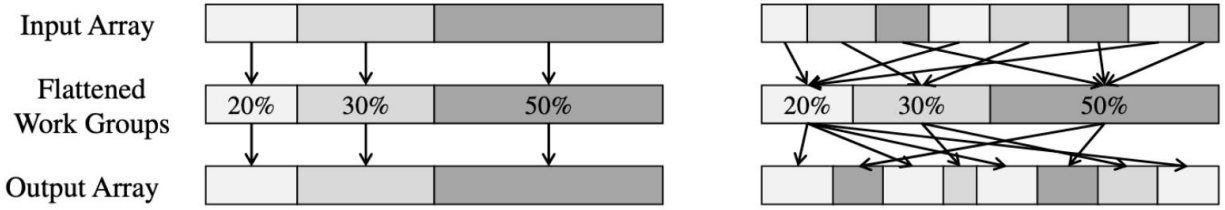



(a) 3-dim NDRange (b) Flattened Work Groups

**Figure 8: 3D NDRange to flattened WGs**

**Developed an analysis pass parseKernel**

- select G
- WGs partitioned to B = M/G blocks
- memory is also divided to B sub-objects
- check whether addresses of I/O operations are bounded with sub-object




(a) Regular Data Access (b) Irregular Data Access

**Figure 9: Regular and irregular data access patterns**

# System Design

## Context-Aware Computing Time Predictor

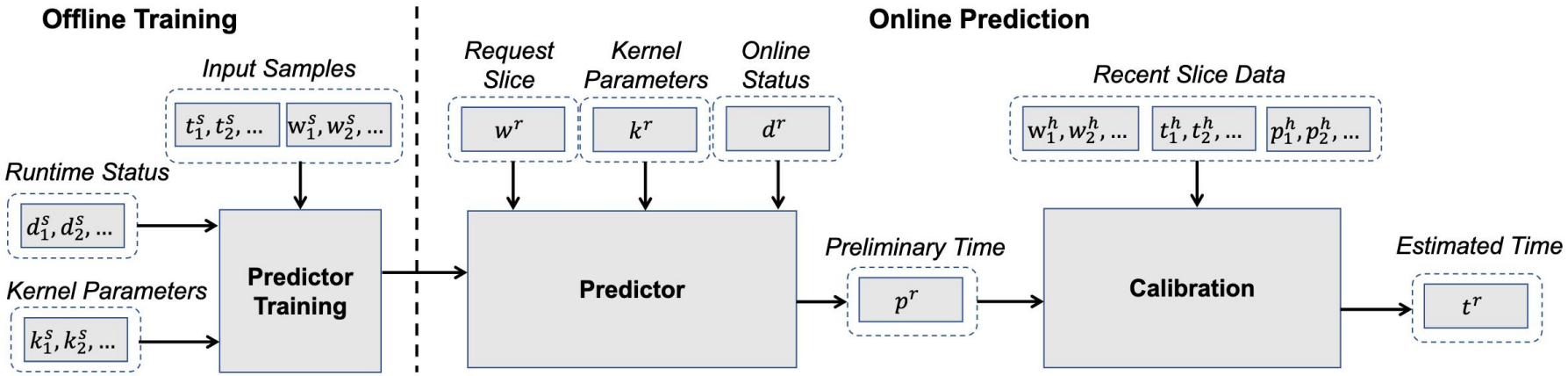



**Figure 10: Data flow of context-aware computing time prediction**

### Offline Training

- Samples data under different configurations and runtime status
- Records its corresponding computing time
- Train a lightweight prediction model for each device using random forest regression

# System Design

## Context-Aware Computing Time Predictor

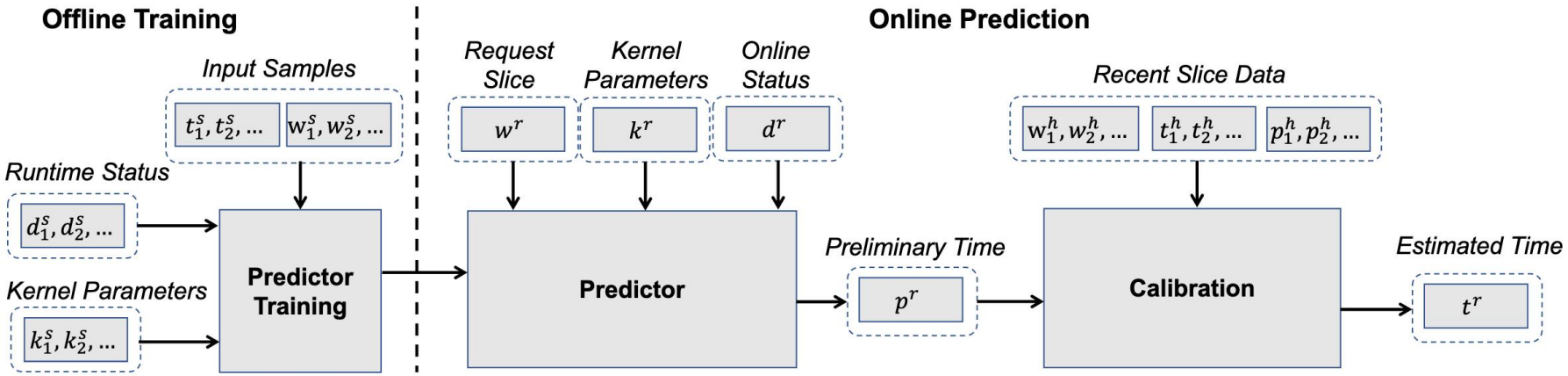



**Figure 10: Data flow of context-aware computing time prediction**

**Online Prediction**

- first calculate a preliminary predicted time pr

**Calibration**

- collect historical slice execution data in the recent 30 seconds
- build a linear regression model $t^r = x_0 p^r + x_1$

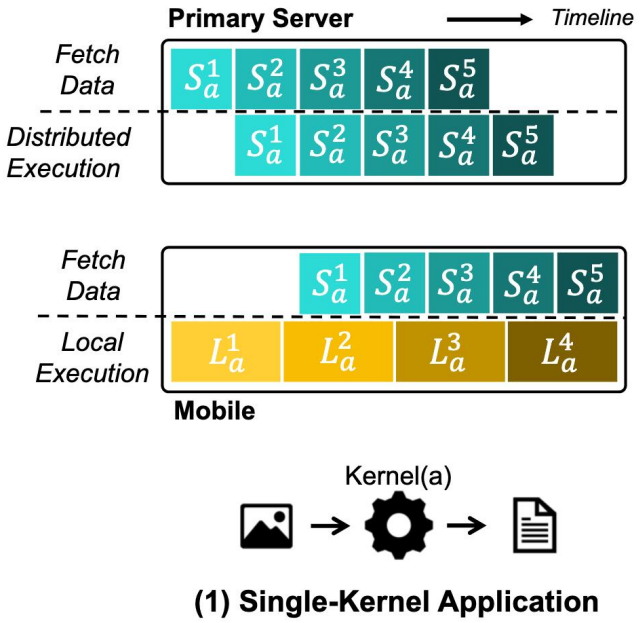
# System Design

## Pipeline-enabled and Network-adaptive Scheduler

**Decides the WG number of each slice and how many slices to be offloaded for each computation unit at runtime**

### Pipeline Processing - reduce the data transmission cost

Simple for single-kernel applications



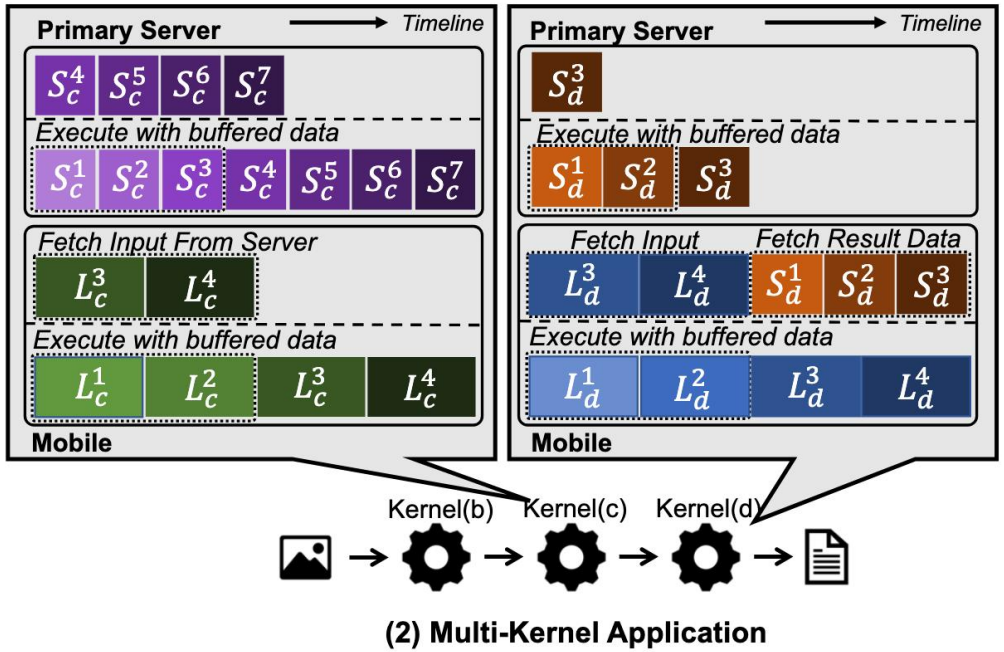

**(1) Single-Kernel Application**

# System Design

## Pipeline-enabled and Network-adaptive Scheduler

### Pipeline Processing - reduce the data transmission cost

Complex in multi-kernel applications




(2) Multi-Kernel Application

- Not immediately transfer the result data back to the local device
- Except for the last kernel, the result data are temporarily stored in the corresponding buffer

# System Design

## Pipeline-enabled and Network-adaptive Scheduler

### Computing Time Predictor for Multiple Edge Servers

$$t = T_i(w) \qquad w = T_i^{-1}(t)$$

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

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.

# System Design

## Pipeline-enabled and Network-adaptive Scheduler

### Getting the Upper Bound of WG number

This upper bound is set mainly due to the dynamic network conditions
It is better to assign less workload to edge servers and more to the mobile for better-balanced loads

minimize the workload in the slice? set 1?

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

Allow a small amount of computational overhead

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

# System Design

## Pipeline-enabled and Network-adaptive Scheduler

### Deciding the WG Number for Pipeline Processing

**Decide the WG number in each slice to maximize the pipeline parallelism**

Ideal: the computational overhead of each slice can just cover the data transmission overhead

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

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

# System Design

## Pipeline-enabled and Network-adaptive Scheduler

### Workload Scheduling

**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.

$E_c^l \leftarrow \frac{T_l(w_{\max}^l)}{w_{\max}^l}$, $E_c^s \leftarrow \frac{T_s(w_{\max}^s)}{w_{\max}^s}$, $F_0 \leftarrow \infty$
**for** $i = 0 \rightarrow w_r$ **do**
  $w_l \leftarrow i + w_e^l$, $w_s \leftarrow w_r - i + w_e^s$
  **if** *the kernel is the last kernel in the application* **then**
    $C_d^l \leftarrow (w_l - w_r^l)\frac{s_i}{b_d} + (w_s - w_d^s)\frac{s_o}{b_d}$
  **else**
    $C_d^l \leftarrow (w_l - w_r^l)\frac{s_i}{b_d}$
  $F_l \leftarrow f_l + max\{C_d^l, (w_l - w_e^l)E_c^l\}$
  $F_s \leftarrow f_s + max\{(w_s - w_r^s)\frac{s_i}{b_u}, (w_s - w_e^s)E_c^s\}$
  **if** $max\{F_l, F_s\} < F_0$ **then**
    $F_0 \leftarrow max\{F_l, F_s\}$, $w_t^l \leftarrow w_l$, $w_t^s \leftarrow w_s$

$w_a^k \leftarrow w_r^k - w_e^k$ ($k = l$ if scheduling for the local, or $k = s$)
**if** $w_t^k \leq w_a^k$ **then**
  $w_e \leftarrow \min(w_t^k, w_{\max}^k)$ $\quad w_t^k > w_a^k$
**else if** $w_a^k < w_0^k$ **then**
  $w_i \leftarrow \min(w_0^k - w_a^k, w_t^k - w_a^k)$ $\quad w_0^k > w_a^k$
**else if** $w_0^k <= w_a^k$ **then**
  $w_e \leftarrow \min(w_a^k, w_{\max}^k)$ $\quad w_t^k > w_a^k > w_0^k$
  **if** *scheduling for local device* **then**
    $w_i \leftarrow \min(\frac{b_d T_l(w_e)}{s_i}, w_t^l - w_a^l)$
  **else**
    $w_i \leftarrow \min(\frac{b_u T_s(w_e)}{s_i}, w_t^s - w_a^s)$, $w_o \leftarrow \frac{b_d T_s(w_e)}{s_o}$
Return $w_i$, $w_e$, $w_o$

# Evaluation

**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 |

- **Baseline**
  - **SKMD, Greed, Sigmoid, Full-offloading, No-offloading**
- **Metric**
  - **comparing their overall time with the no-offloading scheme**

# Evaluation

## Evaluation under Dynamic Network Conditions

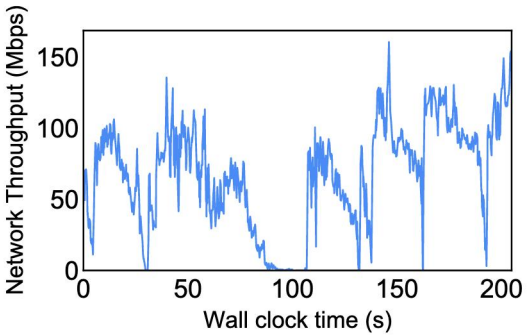



Figure 12: Network throughput variation during our walk.

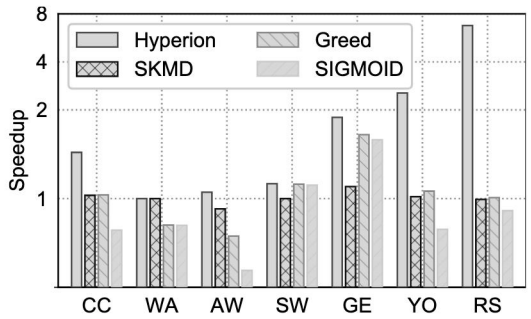



Figure 13: Speedup comparison under network dynamics (higher is better).

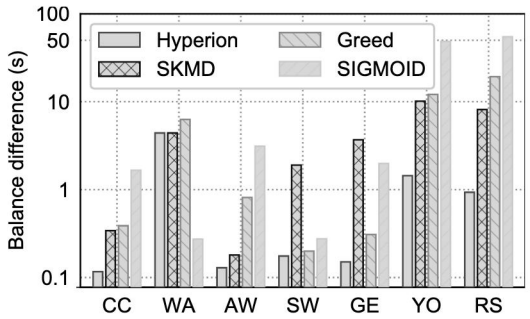



Figure 14: Balance difference comparison under network dynamics.

$$max(\mathcal{F}) - min(\mathcal{F})$$

# Evaluation

## Performance under different network throughput

### Network throughput is limited to50 – 300 Mbps with 10 ms latency

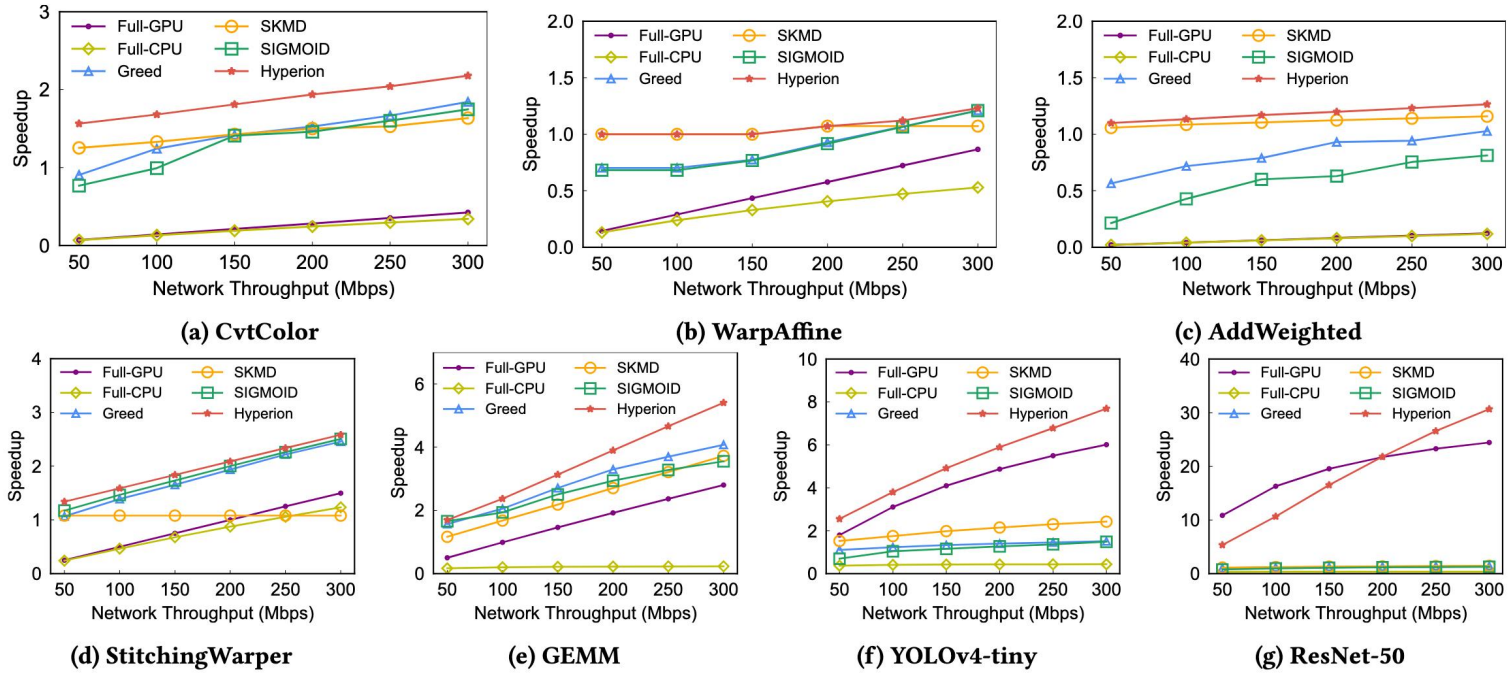



Figure 15: Evaluation results of speedup of local GPU (higher is better) with respect to different network throughput.

# Evaluation

## Performance of Distributed Computing

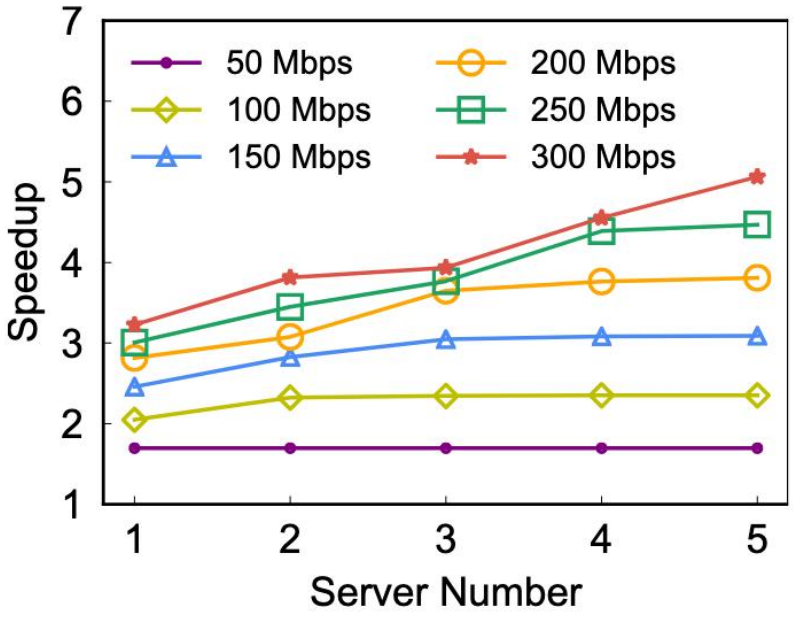



**Figure 16: Impact of the server number and network throughput.**

# Evaluation

## Component-wise evaluations

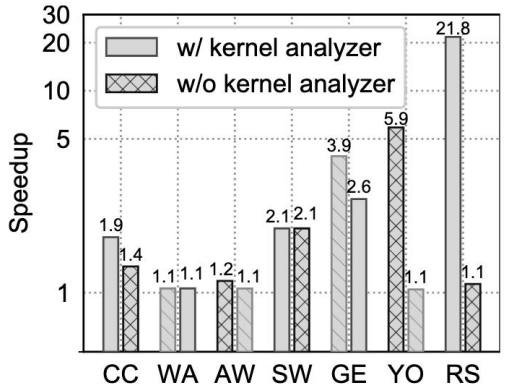



**Figure 17: Performance of Hyperion with and without the kernel analyzer.**

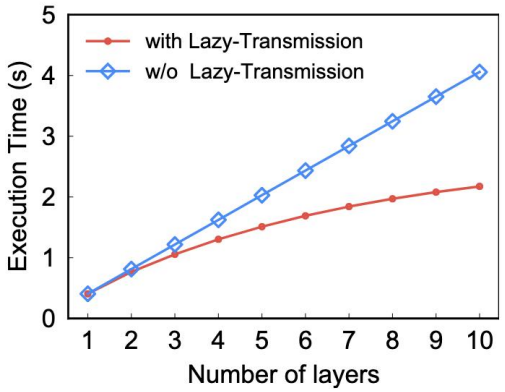



**Figure 18: Performance of Hyperion with or without the lazy-transmission pipeline.**

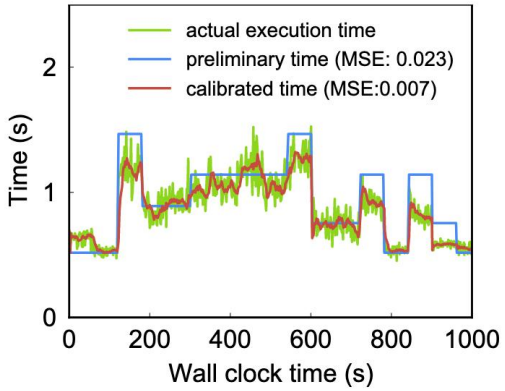



**Figure 19: The predicted time under dynamic system loads (lower MSE is better)**

# Conclusion

- Design a generic and distributed mobile offloading
- High performance and heterogeneity compatible
- Integrates three techniques
  - regular-aware kernel analyzer
  - context-aware predictor
  - the pipeline-enabled and network-adaptive scheduler
- Limitations
  - Unsuitable for irregular OpenCL kernels

# *Thank You*

*Presented by Ye Wan*

*2023-03-23*