

# CacheInf: Collaborative Edge-Cloud Cache System for Efficient Robotic Visual Model Inference

Anonymous Author(s)

Submission Id: 841

## Abstract

Visual information processing is crucial for mobile robots performing tasks such as navigation, manipulation, and human-robot interaction. However, limited computational power for local computation and unstable wireless network bandwidth for computation offloading to a GPU server on these robots lead to slow visual model inference, hindering real-time responsiveness and increasing energy consumption. Existing caching mechanisms designed for fixed edge devices reduce computation by reusing cached activations (computation intermediates) by aligning the activations to the detected movements on the input images, but they face challenges on mobile robots due to frequent camera perspective changes that accumulates error when reusing the cached activation, which leads to degraded inference accuracy or frequent discarding and recomputing the cached activations that increases latency.

We propose CacheInf, a high-performance edge-cloud caching system for efficient visual model inference on mobile robots. Instead of wholly reusing or recomputing the activations, CacheInf selectively reuses a portion of the cached activations frame and recomputes the others, which minimizes required computation, accelerates both local processing and computation offloading, and mitigates error accumulation in activations, achieving an optimal trade-off between inference accuracy and speed. Evaluation on various visual models and wireless network environments shows CacheInf reduced end-to-end inference latency by up to 48.8% and reduced average energy consumption for inference by up to 39.9% compared with the baselines.

**Keywords:** Edge computing

## 1 Introduction

Visual information is vital for various robotic tasks deployed on mobile edge devices (mobile robots), such as navigation [20], manipulation [2], and human-robot interaction [29]; and as a major visual information processing method, fast, accurate and energy-efficient visual model inference is important for the robotic tasks to timely respond to environment changes. Unfortunately, the mobile robot typically has limited computational power and limited and unstable wireless network bandwidth [31], which slow down both local computation and naive offloading of computation to GPU servers. Thus the mobile robot often suffers the problem of slow

visual model inference that interferes the robotic task performance (e.g., 1.xx seconds in human pose estimation [11] or 2.xx seconds in surrounding occlusion prediction [26]) and the prolonged inference latency naturally increases energy consumption each inference.

To address this problem, we seek enlightenment from previous work targeting fixed edge devices that decrease the amount of computation required in visual model inference with the caching mechanism [3, 4, 8]. They are based on the fact that the visual models extensively use operators (e.g., convolution) whose computation results (i.e. activations) are spatially correlated to the input image: the value of each pixel on the activations is dominated by a block of the input image (i.e., receptive field) determined by the model architecture [3]. Given a continuous stream of images, they either 1. recognize the movements of the receptive fields and interpolate (reuse) the cached activations of an operator accordingly to skip the activation computation of this and the previous operators, or 2. in cases of movements recognition failure, execute full local computation (recompute) on the input image to get the latest activations.

While introducing such mechanism to reuse the cached previous activations to visual model inference on mobile robots seems able to skip local activation computation and reduce transmission data volume in computation offloading (since only recognized movements needs to be transmitted), it faces a **dilemma** between inference accuracy and inference speed on mobile robots which feature frequent camera perspective movements [4, 8]. Changes in camera perspective typically bring new scenes or changed occlusion into the images, which cannot be covered by receptive field movements (recognition failure). To cope with this situation, the above methods can only either ignore certain recognition failure at the cost of severely degraded inference accuracy (25.89% lower in [8]), or frequently execute full local computation or full transmission of the input images during computation offloading, sacrificing inference speed.

The key reason to their dilemma is that while reusing or recomputing the activations optimizes the inference speed or accuracy and sacrifices the opposite, there lacks a tradeoff method between inference accuracy and speed. When continuously reusing activations computed with a reference frame, the difference between the new frames and the reference frame accumulates and causes either degraded accuracy or triggering of full recomputation to eliminate error in activations; but with the ability to tradeoff between accuracy and

111 inference speed (for example, partially reusing and partially  
 112 recomputing the activations), we can selectively compute on  
 113 the blocks with most differences while reusing the computa-  
 114 tion results of the others, mitigating the accumulation of  
 115 error while accelerating both local computation and compu-  
 116 tation offloading via caching.

117 To bridge this gap, in this paper, we propose CacheInf, a  
 118 high-performance collaborative edge-cloud cache system for  
 119 efficient robotic visual model. Given a continuous stream  
 120 of visual input in a robotic visual task, CacheInf selectively  
 121 reuses a portion of the cached activations of a reference  
 122 frame while recomputes the rest based on statistical metrics  
 123 such as the mean square error of pixels in corresponding  
 124 receptive fields between the reference frame and the current  
 125 frame. In this way, the amount of required computation is  
 126 minimized which accelerates both local computation and  
 127 computation offloading and the error accumulation in activa-  
 128 tions is mitigated, achieving the optimal tradeoff between  
 129 inference accuracy and inference speed for collaborative  
 130 edge-cloud visual model inference on the mobile robot.

131 The first challenge of the the design of CacheInf is to min-  
 132 imize computation using cache mechanism without compro-  
 133 mising the inference accuracy. While prioritizing recompu-  
 134 tation on receptive fields with most difference is intuitively  
 135 effective, we also notice under the same level of difference  
 136 between consecutive images, if we choose to cache the activa-  
 137 tions computed by an operator closer to the input image  
 138 (former), the portion of activations needed to be recomputed  
 139 would be reduced. This is because the former operators have  
 140 smaller receptive fields compared with the latter ones whose  
 141 activations are less affected by difference across images, but  
 142 there are more operators after the cached operator whose  
 143 computation are not skipped.

144 Based on these observations, we design a mechanism in  
 145 CacheInf to both selectively and adaptively choose the op-  
 146 erator to cache its computed activations and the portion  
 147 of activations to be recomputed according to the levels of  
 148 difference across input images detected and the amount of  
 149 possible computation reduction, achieving maximal compu-  
 150 tation reduction while maintaining high inference accuracy.

151 The second challenge is to minimize the overall inference  
 152 latency considering the interaction between the mobile robot  
 153 and the GPU server. One major obstacle is that switching  
 154 between local computation and computation offloading under  
 155 the caching mechanism (e.g., when wireless network  
 156 bandwidth changes) typically requires costly one full local  
 157 recomputation or full transmission of the input image, and  
 158 such cost hinders a greedy scheduling method from adopt-  
 159 ing such switching to gain further overall acceleration. To  
 160 overcome this issue, we schedule between local computation  
 161 and computation offloading by looking ahead several steps  
 162 with the predicted future wireless network bandwidth and  
 163 possible computation reduction based on previous records,  
 164 to minimize the overall inference latency.

166 We implement CacheInf based on python and pytorch  
 167 integrated with self-implemented C++ cuda extensions. For  
 168 the tail cases where we need to fully offload an input image  
 169 to the GPU server, we integrate a state-of-the-art offloading  
 170 method called Hybrid-Parallel (HP) [23] which mitigates the  
 171 caused latency. Our baselines include HP, a state-of-the-art  
 172 cache-based computation reduction methods called EVA2 [3],  
 173 EVA2 integrated with HP and local computation. We evalua-  
 174 ted CacheInf on a four-wheel robot equipped with a Jetson  
 175 NX Xavier [14] that is capable of computing locally with its  
 176 low-power-consumption GPU. The offloading GPU server  
 177 is a PC equipped with an Nvidia 2080ti GPU. Our datasets  
 178 include the standard datasets of video frames of DAVIS [17]  
 179 and CAD [5] each captured by a handheld camera and our  
 180 self-captured video frames using sensors on our robot. Ex-  
 181 tensive evaluation over various visual models [11, 24, 26]  
 182 and wireless network bandwidth circumstances shows that:

- CacheInf is fast and accurate. Among the baselines, CacheInf reduced the end-to-end inference time by 13.1% to 48.8% with only 0.21% to 0.96% accuracy reduction.
- CacheInf saves energy. Among the baselines, CacheInf reduced the average energy consumed to complete inference on each image by 9.5% to 39.9%.
- CacheInf is robust. Under different level of difference across consecutive input images, the advantages of CacheInf remained.

183 The contribution of this paper is twofold: 1. a new caching  
 184 mechanism for visual model inference on mobile devices  
 185 which selectively reuse and recompute fractions of cached  
 186 activations to best tradeoff between inference accuracy and  
 187 computation reduction; 2. a scheduling mechanism designed  
 188 to optimize the overall inference latency considering the  
 189 interaction between the edge (the mobile robot) and the cloud  
 190 (the GPU server to offload computation to) And the resulting  
 191 system, CacheInf, optimally reduces visual model inference  
 192 latency and energy consumption on the mobile robot. The  
 193 accelerated visual model inference and the reduced power  
 194 consumption will make real-world robots more performant  
 195 on various robotic tasks and nurture more visual models  
 196 to be deployed in real-world robots. The source code and  
 197 evaluation logs of CacheInf is available at [todo](#).

198 The rest of this paper is organized as follows. Chapter  
 199 two introduces background and related work. Chapter three  
 200 gives an overview of CacheInf and Chapter four presents  
 201 its detailed design. Chapter five describes the implementa-  
 202 tion. Chapter six presents our evaluation results and Chapter  
 203 seven concludes.

## 2 Background

### 2.1 Vision tasks on robots

218 Vision tasks play a crucial role in enabling robots to per-  
 219 ceive, understand, and interact with the environment. Visual

220

information is essential for various robotic tasks, such as object recognition [6], navigation [20], manipulation [2], and human-robot interaction [29]. The rapid advancements in machine learning, particularly deep learning, have revolutionized the field of computer vision and have been widely adopted in robotic applications, which form the foundation for many high-level robotic tasks.

However, the deployment of visual models on resource-constrained robots poses significant challenges. Visual models often require significant computational resources and memory, which may not be readily available on robots, especially in mobile and embedded systems. Furthermore, real-time performance is critical for many robotic tasks, as robots need to process and respond to visual information quickly to ensure safe and effective operation. Therefore, fast visual model inference becomes a key requirement for the successful deployment of deep learning models in robotic applications. Addressing these challenges is essential for enabling robots to effectively perceive, understand, and interact with their environment in real time, paving the way for more intelligent and autonomous robotic systems.

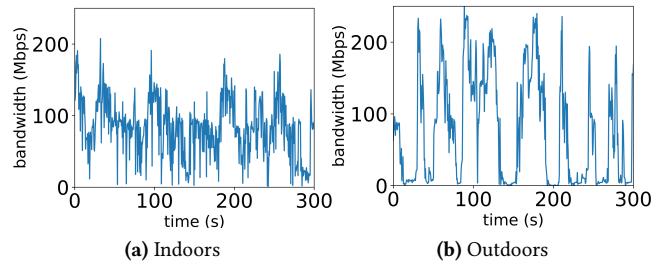
## 2.2 Visual Models

Convolutional layers [15] have become a fundamental building block in visual models, leading to significant breakthroughs in various computer vision tasks. Inspired by the biological structure of the visual cortex [25], these layers apply learnable filters to the input image, performing convolution operations to produce feature maps that highlight the presence of specific patterns at different spatial locations (i.e., local operators). This enables deep learning models to capture translation-invariant features and learn hierarchical representations [10], with early layers learning low-level features like edges and corners, and deeper layers learning more complex patterns and object parts. As a result, deep convolutional neural networks (CNNs) have achieved state-of-the-art performance in various vision applications, image classification [21], object detection [6], and semantic segmentation [27], due to their ability to effectively capture and learn spatial hierarchies of features from raw input images. As the field of computer vision continues to evolve, convolutional layers are expected to remain a crucial component in the development of advanced models for understanding and analyzing visual data.

## 2.3 Resource Limitations of Robots

To demonstrate the instability of wireless transmission in real-world situations, we conducted a robot surveillance experiment using four-wheel robots navigating around several given points at 5–40cm/s speed in our lab (indoors) and campus garden (outdoors), with hardware and wireless network settings as described in Sec. 6.6. We saturated the wireless network connection with iperf [1] and recorded the average

bandwidth capacity between these robots every 0.1s for 5 minutes.



**Figure 1.** The instability of wireless transmission between our robot and a base station in robotic IoT networks.

The results in Fig. 1 show average bandwidth capacities of 93 Mbps and 73 Mbps for indoor and outdoor scenarios, respectively. The outdoor environment exhibited higher instability, with bandwidth frequently dropping to extremely low values around 0 Mbps, due to the lack of walls to reflect wireless signals and the presence of obstacles like trees between communicating robots, resulting in fewer received signals compared to indoor environments. This limitation on the wireless network bandwidth on the robot poses significant challenges for the efficient and reliable computation offloading of robots in real-world scenarios, particularly in outdoor environments where the instability of wireless networks is more pronounced.

## 2.4 Related Work

Edge-Cloud Collaborative Inference expedites the overall inference process by leveraging a GPU server to handle a portion of the computational workload of the robot. The DSCCS approach [9] views the visual model in a layer-wise perspective and focuses on model-layer-level scheduling (layer partitioning) for rapid inference; Hybrid-Parallel [23] further offers a more fine-grained control by partitioning and scheduling the computation within local operators, so that the robot can compute on a portion of the input on local operators while at the same time transmitting the result of the input to the GPU server. It enhances parallelism and further accelerates inference. However, despite the advancements in these offloading techniques, the limited bandwidth still poses a bottleneck for data transmission, which our caching mechanism effectively mitigates and achieves a significant improvement in inference performance.

## 3 System Overview

### 3.1 Working Environment

We assume that the working environment of CacheInf is a mobile robot performing robotic tasks in a real-world field which requires seamless real-time visual model inference on

331 the continuous image stream captured from the on-board  
 332 camera, to achieve real-time response to various environment  
 333 changes. The robot itself is equipped with low-power-  
 334 consumption gpu to perform visual model inference which is  
 335 slow and consumes too much power; it has wireless network  
 336 access to a remote powerful GPU server that provides op-  
 337 portunities of acceleration, but the connection suffers from  
 338 limited and unstable wireless network bandwidth.

### 340 3.2 Architecture of CacheInf

341 CacheInf consists of four major components: CacheInf Sched-  
 342 ule, Cache Tracker, Cache-Aware Collaborative Inference  
 343 and Cache Recoverer. CacheInf Scheduler functions at the  
 344 initialization stage and we exclude it in Figure 2 which de-  
 345 scribes the runtime of CacheInf for simplicity.

346 **3.2.1 CacheInf Scheduler.** During the initialization stage  
 347 of the robotic task and CacheInf is granted access to a visual  
 348 model and an initial input image. CacheInf Scheduler first  
 349 profiles the visual model based on this initial input and its  
 350 execution statistics on both the robot and the server. Then it  
 351 decides on the set of operators involved in the visual model  
 352 that should cache their computation results and computes for  
 353 an optimized computation and offloading plan for each possi-  
 354 ble situation including different wireless network bandwidth  
 355 and different ratios of reusable cache.

356 **3.2.2 Cache Tracker.** Given a pair of consecutive image  
 357 inputs and the former one’s computation intermediates at  
 358 the selected operators are cached, Cache Tracker identifies  
 359 the reusable portion of these computation intermediates. We  
 360 use the standard image stitching method to try to stitch  
 361 the areas of the two images as much as possible, which re-  
 362 sults in a perspective transform that maps the pixels from  
 363 the former image to the latter. And the same perspective  
 364 transform can be applied to the cached computation results  
 365 since they are computed by local operators that keep the  
 366 local geometries of the input image. We then filter the dif-  
 367 ference between the mapped pixel pairs and find areas of  
 368 similar appearance whose correspondent cache is reusable.  
 369 Pixels from uncached areas are finally gathered for compu-  
 370 tation. Note that the computation involved in this process is  
 371 lightweight compared with the visual model inference that  
 372 typically involves hundreds of operators.

373 **3.2.3 Cache-Aware Collaborative Inference and Cache**  
 374 **Recoverer.** In this stage, we select a precomputed plan based  
 375 on the current estimated wireless network bandwidth and  
 376 the estimated ratio of reusable cache and execute it at both  
 377 the robot and the server. We pass the gathered sparse pixels  
 378 that need computation through the sequence of operators  
 379 involved in the visual model. When a local operator with  
 380 cache is met, we gather (depicted in Gather in Figure 2) extra  
 381 pixels from cache and form correct input (e.g., wrap around a  
 382 pixel into a 5x5 pixel block for a convolution kernel with size  
 383 384 385

386 3, detailed in Section 4.2) and feed it to the corresponding  
 387 sparse local operator and subsequently the following local  
 388 operators whose cache is merged. Offloading computation  
 389 and receiving computation results of local operators between  
 390 the robot and the server only happens at local operators with  
 391 cache because they can gather extra pixels required by the  
 392 computation of the opportunity side, where we slice the  
 393 gathered input into splits at the planned ratio and assign  
 394 them to the robot and the server.

395 When a first non-local operator (e.g., linear, flatten) is  
 396 met, Cache Recoverer transforms the cached output of the  
 397 previous local operator and merges it with the current com-  
 398 putation result on the sparse pixels to recover the global  
 399 geometry for subsequent computation.

## 400 4 Design

### 401 4.1 Identifying Reusable Computation Results

402 To find and match similar local geometries between consecu-  
 403 tive images in a stream of images  $I = \{I_1, I_2, \dots, I_n\}$  to identify  
 404 reusable cache, we use the standard image stitching proce-  
 405 dure: given a pair of consecutive images  $I_j$  and  $I_{j+1}$ , their  
 406 key points and key point descriptors (or feature vectors) are  
 407 computed and matched within a distance threshold of the  
 408 feature vectors; then a homography matrix  $M$  is computed  
 409 based on the corresponding relationship between the key  
 410 points on each image which minimizes the error. The result-  
 411 ing homography matrix is then used to apply perspective  
 412 transformation to each pixel in  $I_j$  to form a new image  $\hat{I}_{j+1}$   
 413 closest to  $I_{j+1}$  as shown in Equation 1, where  $(u_j, v_j)$  and  
 414  $(\hat{u}_{j+1}, \hat{v}_{j+1})$  and pixel indices on  $I_j$  and  $\hat{I}_{j+1}$ . It is also depicted  
 415 in the Feature Based Transformed Image in Fig. 2. Since the  
 416 computation of local operators relies on local geometries,  
 417 the same transformation can be applied to intermediate com-  
 418 putation results of the following local operators.

$$(u_{j+1}, v_{j+1}, 1) = M \times (u_j, v_j, 1) \quad (1)$$

419 While the above process minimizes error between  $\hat{I}_{j+1}$   
 420 and  $I_{j+1}$ , the remaining different areas between them are the  
 421 areas of new information which are uncached and need to be  
 422 recomputed. We filter and identify these areas by applying  
 423 average pooling over the difference between  $\hat{I}_{j+1}$  and  $I_{j+1}$  and  
 424 the pixels with computed difference greater than a preset  
 425 threshold  $N$  (default to be 0.1 when we normalize the value  
 426 of each channel of a pixel to between 0 and 1) will be marked  
 427 as needed to be recomputed as in Equation 2, where  $u, v$  are  
 428 the pixel indices.

$$\mathbf{uv} = \{(u, v) | \text{AveragePooling}(|\hat{I}_{j+1} - I_{j+1}|)^{u,v} \geq N\} \quad (2)$$

429 Suppose there are  $Q$  pixels in  $\mathbf{uv}$  and  $H \times W$  total pixels in  
 430 each image, we define the cache ratio between  $I_j$  and  $I_{j+1}$  as  
 431  $r = \frac{Q}{H \cdot W}$ .

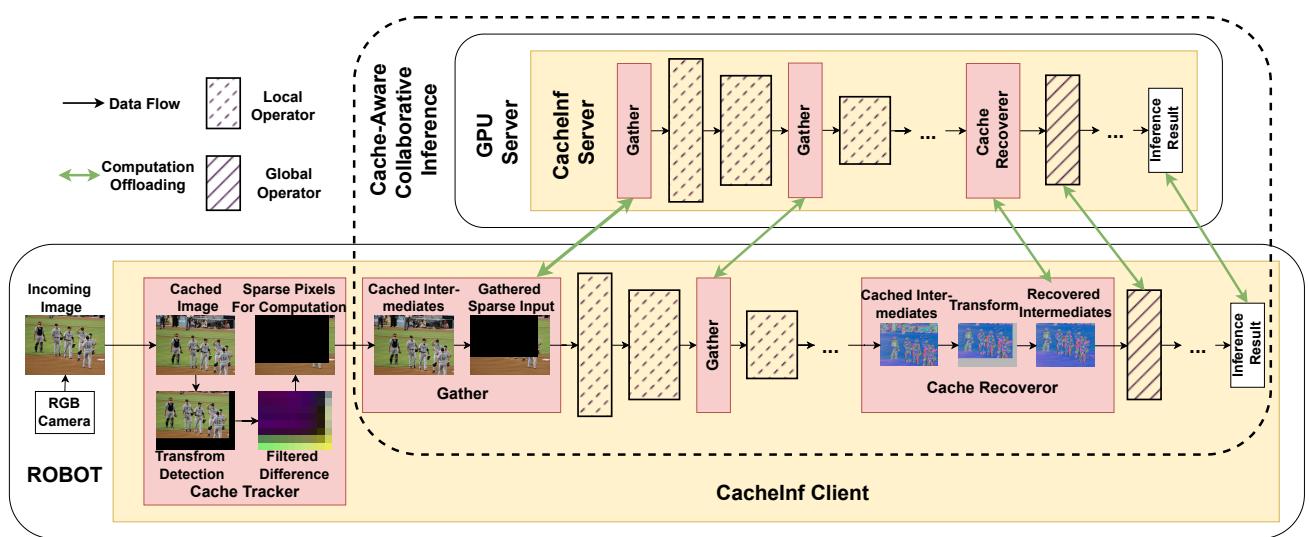


Figure 2. Architecture and workflow of CacheInf.

## 4.2 Sparse Local Operators

From the above discussion, we have identified the pixels needed to recompute  $\mathbf{uv}$  and we suppose their corresponding features  $f_{inp}$  are of size  $B \times C_1 \times Q$ , along with the cached input defined as  $I'_{inp}$  of size  $B \times C_1 \times H \times W$ . Now we focus on how to compute the correct results based on  $\mathbf{uv}$ ,  $f_{inp}$  and  $I'_{inp}$ . There are mainly two kinds of local operators: element-wise local operators such as addition, subtraction, multiplication and division, which solely depends on the value of each element; and convolution local operators such as convolution, average pooling and max pooling, which is influenced by the surrounding areas (e.g., a 2D kernel) of each element. We mainly focus on the latter type of local operators since the element-wise local operators can be viewed as a special case of convolution local operators where the surrounding area is of size one.

We first consider the scenario with dense input. Assume an image (or feature map)  $I_{inp}$  of size  $B \times C_1 \times H \times W$ , a convolution local operator  $K$  with its kernel sized  $C_2 \times C_1 \times K_1 \times K_2$ , stride 1 and no padding and its output feature map  $I_{out}$  of size  $B \times C_2 \times H' \times W'$ , then each of the value of the output feature map is determined by

$$I_{out}^{i,j,k,l} = \sum_{c=1}^{C_1} \sum_{m=1}^{K_1} \sum_{n=1}^{K_2} K^{j,c,m,n} * I_{inp}^{i,c,k+m-1,l+n-1}, \quad (3)$$

Omitting the batch dimension and the channel dimension (first two dimension) of  $I_{out}$ , we can learn from Equation 3 that an output value is determined by an area of  $K_1 \times K_2$  on  $I_{inp}$  and we define pixels in this area as

$$P_{k,l} = \{(u,v) | k \leq u < k + K_1 \wedge l \leq v < l + K_2\} \quad (4)$$

where  $(k, l)$  is the pixels indices on  $I_{out}$ .

Moving to the sparse scenario, the indices of pixels on  $I_{out}$  that have updated value with  $\mathbf{uv}$  as input would be

$$\mathbf{uv}' = \{(k, l) | \exists P_{k,l}, s.t. P_{k,l} \cap \mathbf{uv} \neq \emptyset\} \quad (5)$$

which can be view as wrapping around pixels in  $\mathbf{uv}$  by  $K_1 \times K_2$  and may involve pixels in  $I'_{inp}$ .

Note that  $\mathbf{uv}$  and cached input  $I'_{inp}$  are possibly in different planes determined by the homography matrix  $M$ . We may transform the cached intermediates every time before computation, but it will unfortunately involve computation of the whole feature map and invalidate the acceleration of sparse computation. Instead, during computation we query the original cached intermediates by transforming the pixel indices with  $M$ :

$$F(i, j, u, v, I'_{inp}, f_{inp}) = \begin{cases} f_{inp}^{i,j,u,v}, & (u, v) \in \mathbf{uv}, \\ I'_{inp}^{i,j,G(u,v,M)}, & (u, v) \notin \mathbf{uv} \end{cases} \quad (6)$$

where  $G(u, v, M) = H^{-1}(M^{-1} \times H((u, v)))$  which transforms  $(u, v)$  into the plane of cached input  $I'_{inp}$ , and  $H(\cdot)$  and  $H^{-1}(\cdot)$  means turning a vector to a homogeneous vectors and the opposite. Then for  $(u, v) \in \mathbf{uv}'$ , the corresponding computed output is

$$f_{out}^{i,j,u,v} = \sum_{c=1}^{C_1} \sum_{m=1}^{K_1} \sum_{n=1}^{K_2} K^{j,c,m,n} \cdot F(i, c, k+m-1, l+n-1, I'_{inp}, f_{inp}) \quad (7)$$

Until now we get the indices of the altered output values in output feature map  $\mathbf{uv}'$  and the corresponding features  $f_{out}$  which can then be passed to the subsequent computation.

Along the local operators where local geometries are preserved, we can repeat the above process by passing only the sparse features and their indices and do not need to merge the sparse features with cache. When a non-local operator

is met (e.g., matrix multiplication), we transform its cached input with  $M$  and merge  $f_{inp}$  into the transformed input according to their sparse indices  $\mathbf{uv}$ , which recovers the correct geometries of the whole feature map. To minimize performance impact to update  $I'_{inp}$ , we update  $I'_{inp}$  by transforming  $I'_{inp}$  and merge it with  $f_{inp}$  only after the whole computation process finishes, when the system is typically idle and waiting for the next input.

Also, to save memory consumption of cached intermediates, notice that the above process is basically wrapping the sparse pixels with the kernel size  $K_1 \times K_2$  and computing on the wrapped pixels, we can merge the query process in Equation 6 of multiple convolution local operators into the first convolution local operator. For example, if a next operator is a convolution local operator with kernel size  $K'_1 \times K'_2$ , we can wrap the sparse pixels with an extended kernel size  $(K_1 + K'_1) \times (K_2 + K'_2)$  in the first local operator, and the wrapping process of the next operator is skipped (we refer to this process as merging cache). In this case, the cache for the input of the next operator is needless and can be excluded to save memory consumption and the reduced number of cached input further leverages the cost to update  $I'_{inp}$ .

### 4.3 Cache-Aware Scheduling

We define all the operators involved in a visual model as  $\mathcal{O} = \{o_1, o_2, \dots, o_n\}$  and the portion of locally executed input of each operator as  $\mathbf{X} = \{x_1, x_2, \dots, x_n\}$ ,  $0 \leq x_i \leq 1$  and  $1 - x_i$  represents the the portion of input executed on the GPU server. The indices of local operators is defined as  $\mathcal{O}_l$ . While offloading, we transmit the sparse features together with their indices encoded as a bit-mask and the transmission volume is almost inversely proportional to cache ratio  $r$ .

However, the local computation time acceleration with sparse local operators has a complex relationship with cache ratio, which is affected by the operator implementation, gpu structure and so on. Thus we profile such relationship by altering the cache ratio and  $x_i$  and record the average execution time for every operator involved in the visual model and we define the profile result as a function  $T_c(o_i, x_i, r)$  for the robot ( $c$  means client) and  $T_s(o_i, x_i, r)$  for the server, which returns the execution time of operator  $o_i$  under  $x_i$  with cache ratio  $r$ . We also profile the time cost to update cached input for each local operator and get  $U_r(o_i, x_i, r)$  and  $U_s(o_i, x_i, r)$ . Note that for non-local operators ( $\mathcal{O}_{nonlocal} = \{i | 1 \leq i \leq n \wedge i \notin \mathcal{O}_l\}$ ), we make both  $T(\cdot)$  and  $U(\cdot)$  returns time of computation on the whole input.

**4.3.1 Schedule to Merge Cache.** With the above setup, the first problem to solve will be the choices of merging the cache of sparse local operators to further accelerate computation while saving memory consumption. We define the indices of the chosen operators to cache their input as  $\hat{\mathcal{O}}_c$  and the resulting reduction of cache ratio (since extra input will be included) of each operator as  $o_l$  to be  $R(\hat{\mathcal{O}}_c, o_l)$ . Since

these choices will determine the operators that will cache their input and will be reused across different inference, these choices should be fixed during the whole inference task. Thus we start by considering only the worst case where offloading is not possible and  $\forall x_i \in \mathbf{X}, x_i = 1$ . In this case the execution time of every operator will be  $T_c(o_i, 1, r - R(\hat{\mathcal{O}}_c, o_i))$ , and the optimization problem will be

$$\min_{\hat{\mathcal{O}}_c \subset \mathcal{O}_l} \frac{1}{w} \sum_{i=1}^n \sum_{j=0}^w T_c(o_i, 1, r_j + R(\hat{\mathcal{O}}_c, o_i)) + U_{sum}(\hat{\mathcal{O}}_c, r_j) \quad (8)$$

where  $r_j = \frac{j}{w}$  with  $w > 1$  is the possible cache ratio considered (we empirically set  $w$  to 10), and  $U_{sum}(\hat{\mathcal{O}}_c, r) = \sum_{k \in \hat{\mathcal{O}}_c} U(o_k, x_k, r)$  is the total time to update cache of operators in  $\hat{\mathcal{O}}_c$ .

Solving of this optimization problem seeks the optimal choices of cache operators  $\hat{\mathcal{O}}_c$  that minimizes local execution time averaged across all possible cache ratio. Note that we do not need to explicitly consider memory consumption because the latter term in Equation 8 will naturally reduce the number of cached operators and favor operators with smaller size of input and thus shorter time to update cache.

**4.3.2 Schedule of Offloading.** Finally, we are combining all the above components to schedule for computation and offloading in a cache-aware way to optimize end-to-end inference latency for robotic visual models. With Hybrid-Parallel integrated, cache can exists partially both at the robot and the server and we analyze the cache ratio on robot  $r_c$  and the cache ratio  $r_s$  on server by enquiry the current cached pixels with the previous slice of input (i.e.,  $x_i$ ). For an  $x_i$ , we define the minimum portion of locally executed input of its parent operators (i.e., operators whose output is the input of  $o_i$ ) as  $x'_i$  and different between  $x_i$  indicates offloading to/from the server. For every operator  $o_i \in \mathcal{O}$  involved in a visual model, we define its finishing time since the first operator starts executing as  $t_i^c$  on the robot ( $c$  means client) and  $t_i^s$  on server.

We can have the finish time of each operator on the robot and the server as the following, where  $D(o_i, x'_i - x_i, r)$  is the data volume needed to be transmitted at operator  $o_i$  with cache ratios  $r_c$  and  $r_s$  and  $b$  is the estimated bandwidth:

$$t_i^c = \begin{cases} t_{i-1}^c + T_c(o_i, x_i, r_c - R(\hat{\mathcal{O}}_c, o_i)), & 1 \leq i \leq n \wedge x_i \leq x'_i \\ \max(T_c(o_i, x_i, r_c - R(\hat{\mathcal{O}}_c, o_i))) + \\ t_{i-1}^c, & \frac{1}{b} D(o_i, x'_i - x_i, r) + t_i^s, \quad 1 \leq i \leq n \wedge x_i > x'_i \end{cases}$$

$$t_i^s = \begin{cases} t_{i-1}^s + T_s(o_i, 1 - x_i, r_s - R(\hat{\mathcal{O}}_c, o_i)), & 1 \leq i \leq n \wedge x_i \geq x'_i \\ \max(T_s(o_i, 1 - x_i, r_s - R(\hat{\mathcal{O}}_c, o_i))) + \\ t_{i-1}^s, & \frac{1}{b} D(o_i, x'_i - x_i, r_s) + t_i^c, \quad 1 \leq i \leq n \wedge x_i < x'_i \end{cases}$$

The first rows of the above two equations describe the scenarios where either the robot or the server does not need to

receive data from the opposite side and thus the finishing time of this operator only depends on its local execution time. The second rows instead describe the opposite scenarios, where either the robot or the server needs to receive data from the opposite side (e.g.,  $x_i > x'_i$  for the robot) and have to wait until the same operator to finish computing at the opposite side and then be transmitted at bandwidth  $b$ .

With the above statements, optimizing the end-to-end inference latency for the visual model with cache enabled at a given bandwidth  $b$  and cache ratios  $r_c$  and  $r_s$  is to solve

$$\begin{aligned} \min_{\mathbf{x}} \quad & \mathbf{x} \\ \text{s.t.} \quad & x_1 = x_n = 1 \\ & \forall j \in \mathcal{O}_{\text{nonlocal}}, x_j \pmod{1} = 0 \\ & \forall j \notin \hat{\mathcal{O}}_c, x_j = x'_j \end{aligned} \quad (9)$$

In Equation 9, the first two constraints ensure that inference output will finally be located at the robot and non-local operators will always have full input; the third constraint ensures that offloading will not happen within an operator whose cached is merged into the cache of other operators, since we cannot recover the operator's whole feature map. We solve both optimization problems in Equation 8 and 9 with the differential evolution algorithm [19] and store the solutions of different bandwidth and cache ratios of Equation 9 in a dictionary referred to as *Schedule*.

The resulting algorithms of CacheInf at both the robot and the server are presented in Algorithm 1 and Algorithm 2. Line 1 to 3 in Algorithm 1 and Line 1 to 4 in Algorithm 2 profile the model at both the robot and the server and compute a schedule as described in Section 4.3.2, where the computation is located on the server to speed up computation. The rest of Algorithm 2 is basically mirrored from that of Algorithm 1 and thus we focus on Algorithm 1 for simplicity.

Line 6 to 8 in Algorithm 1 identifies the reusable cache by matching features between the input image  $I$  and its cached counterpart  $\text{Cache}[1]$  and gets the homography matrix  $M$  and the sparse uncached input that needs to be recomputed. After communicating info of bandwidth, cache ratio and homography matrix with the server, we query the *Schedule* to get input ratio  $x$  and parent operator input ratio  $x'$  as described in Section 4.3.2. Then we start executing each operator  $o_i$  involved in the model sequentially. We recover the whole input by combining sparse input  $inp$  with cache for non-local operators or gather extra pixels from cache for  $inp$  for sparse local operator computation at cached operators at line 12 to 19. When offloading is required to accelerate inference, we send a slice of  $inp$  to the server or merge received partial input from the server to  $inp$  at line 20 to 26. When  $inp$  is finally ready and not empty, we execute the operator  $o_i$  with  $inp$  where we choose the sparse local operator for sparse input and choose the original operator for dense input at line 27 to 34.

---

**Algorithm 1:** CacheInfClient
 

---

```

Input: A continual sequence of video images  $I$ ; DNN
      model  $M$ 
Output: The inference results  $ret$  on each image in  $I$ 
      // profile
1  $T_c, U_c = \text{Profile}(M)$ 
2  $\text{Send}(M, T_c, U_c)$ 
3  $\text{Schedule}, \hat{\mathcal{O}}_c = \text{Receive}()$ 
4  $\text{Cache} = \text{InitCache}(\hat{\mathcal{O}}_c)$ 
      // inference
5 foreach  $I$  in  $I$  do
6    $b = \text{EstimateBandwidth}()$ 
7    $r_c, r_s = \text{AnalyzeCacheRatio}(I, \text{Cache}[1])$ 
8    $inp, M = \text{IdentifyCache}(I, \text{Cache}[1])$ 
9    $\text{Send}(b, r_c, r_s, M)$ 
10   $x, x' = \text{Schedule}[b, r_c, r_s]$ 
11  foreach  $i = 1, 2, \dots, n$  do
12    if  $i \in \mathcal{O}_{\text{nonlocal}}$  and  $x_i > 0$  and  $\text{IsSparse}(inp)$ 
13      then
14         $inp = \text{DenseRecover}(inp, \text{Cache}[i], M)$ 
15         $\text{UpdateCache}(\text{Cache}[i], inp, M)$ 
16    end
17    else if  $x'_i > 0$  and  $i \in \hat{\mathcal{O}}_c$  then
18       $inp = \text{SparseGather}(inp, \text{Cache}[i], M)$ 
19       $\text{UpdateCache}(\text{Cache}[i], inp, M)$ 
20    end
21    if  $x_i < x'_i$  then
22       $inp, inp' = \text{Slice}(inp, x_i, x'_i)$ 
23       $\text{Send}(inp')$ 
24    end
25    else if  $x_i > x'_i$  then
26       $inp = \text{Merge}(inp, \text{Receive}())$ 
27    end
28    if  $x_i > 0$  then
29      if  $\text{IsSparse}(inp)$  then
30         $inp = \text{SparseExecute}(o_i, inp)$ 
31      end
32      else
33         $inp = \text{Execute}(o_i, inp)$ 
34      end
35    end
36     $ret[I] = inp$ 
37  end
38 return  $ret$ 

```

---

## 5 Implementation

We implemented CacheInf with python, pytorch [16] and taichi [7] on Ubuntu20.04. The communication library used is the distributed module [18] of pytorch with mpi backend.

716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770

**Algorithm 2:** CacheInfServer

---

```

771 // profile and compute schedule at the server
772 1  $M, T_c, U_c = \text{Receive}()$ 
773 2  $T_s, U_s = \text{Profile}(M)$ 
774 3  $\text{Schedule}, \hat{O}_c = \text{ComputeSchedule}(T_s, U_s, T_c, U_c)$ 
775 4  $\text{Send}(\text{Schedule}, \hat{O}_c)$ 
776 5  $\text{Cache} = \text{InitCache}(\hat{O}_c)$ 
777 // inference
778 6 while True do
779 7    $b, r_c, r_s, M = \text{Receive}()$ 
780 8    $x, x' = \text{Schedule}[b, r_c, r_s]$ 
781 9   foreach  $i = 1, 2, \dots, n$  do
782 10    if  $i \in O_{\text{nonlocal}}$  and  $x_i < 1$  and  $\text{IsSparse}(inp)$ 
783    11    |  $inp = \text{DenseRecover}(inp, Cache[i], M)$ 
784    12    |  $\text{UpdateCache}(Cache[i], inp, M)$ 
785    13    end
786 14    else if  $x'_i < 1$  and  $i \in \hat{O}_c$  then
787    15    |  $inp = \text{SparseGather}(inp, Cache[i], M)$ 
788    16    |  $\text{UpdateCache}(Cache[i], inp, M)$ 
789    17    end
790 18    if  $x_i > x'_i$  then
791    19    |  $inp, inp' = \text{Slice}(inp, x_i, x'_i)$ 
792    20    |  $\text{Send}(inp')$ 
793    21    end
794 22    else if  $x_i < x'_i$  then
795    23    |  $inp = \text{Merge}(inp, \text{Receive}())$ 
796    24    end
797 25    if  $x_i < 1$  then
798    26    | if  $\text{IsSparse}(inp)$  then
799    27    | |  $inp = \text{SparseExecute}(o_i, inp)$ 
800    28    | end
801    29    | else
802    30    | |  $inp = \text{Execute}(o_i, inp)$ 
803    31    | end
804    32    end
805 33  end
806 34 end

```

---

We compiled pytorch with cuda-aware mpi enabled so that the mpi backend can directly read and write to cuda buffer to minimize communication overhead. We use mpi backend instead of the popular nccl backend because nccl is unavailable on the Jetson robot we used due to structural limitation [13].

The sparse local operators were implemented based on the bitmasked sparse nodes in taichi [7], which efficiently manages the sparse pixels in a grid and preserves the spatial structure of the sparse pixels by organizing them in a tree structure. With the spatial structure preserved, common optimization methods for cuda operators based on computation

locality such as block shared memory [12] can be introduced to accelerate computation; our implemented sparse local operators achieved comparable performance with the original pytorch operators with the same input size.

## 6 Evaluation

### 6.1 Evaluation Settings

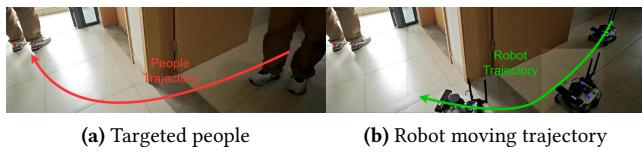
**Testbed.** We conducted experiments on a four-wheeled robot and a air-ground robot. Both robots are equipped with a Jetson Xavier NX [14] 8G onboard computer with cuda acceleration capability and a MediaTek MT76x2U USB wireless network interface card for wireless connectivity. The Jetson Xavier NX is connected to a Leishen N10P LiDAR, an ORBBEC Astra depth camera and an STM32F407VET6 controller via USB serial ports, which are managed and driven using ROS Noetic. The GPU server used in our experiments is equipped with an Intel(R) i5 12400f CPU @ 4.40GHz and an NVIDIA GeForce GTX 2080 Ti 11GB GPU, connected to our robot via Wi-Fi 6 over 80MHz channel at 5GHz frequency.

**Workload.** We chose two real-world visual robotic applications as our major workloads: 1. Kapao [11] depicted in Figure 3, a RGB-image-based real-time people key point detection applications used to guide our four-wheeled robot to track and follow a walking people; 2. AGRNav [26] depicted in Figure 4, an autonomous air-ground robot navigation application that predicts unobserved obstacles by semantic prediction on point clouds and optimizes the navigation trajectory for the air-ground robot. We also verified CacheInf's performance on a broader range of visual models common to mobile devices: VGGNet [22], ConvNeXt [28], RegNet [30] using their default implementation of torchvision [24].

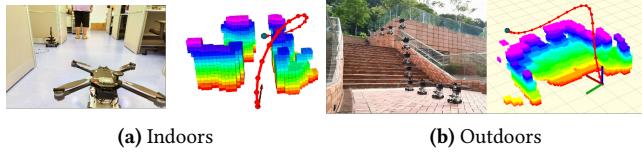
**Dataset.** For AGRNav we used the officially available sequence of point clouds input [26] and for Kapao, we used the Collective Activity Dataset (CAD) [5] which are sequences of video images of people doing different activities captured using hand-held cameras. For the rest of the models from torchvision, we used the DAVIS [17] dataset which are sequences of video images of different objectives captured also using hand-held cameras.

**Experiment Environments.** The experiments across all systems and all workloads were conducted in two different real-world environments (depicted in Figure 1): 1. indoors, where the robot was moving in our office with desks and separators interfering wireless bandwidth; 2. outdoors, where the robot was moving in a garden with trees and bushes interfering with wireless signals and less reflection, resulting in lower bandwidth.

**Baselines.** We selected two SOTA inference acceleration methods as baselines: DSCCS [9] (referred to as DS), which searches for optimal layer partition strategy of a visual model to offload layers to the GPU server to accelerate inference, and Hybrid-Parallel [23] (referred to as HP), which enables parallelization of local computation and offloading by also



**Figure 3.** A real-time people-tracking robotic application on our robot based on a state-of-the-art human pose estimation visual model, Kapao [11].



**Figure 4.** By predicting occlusions in advance, AGRNav [26] gains an accurate perception of the environment and avoids collisions, resulting in efficient and energy-saving paths.

partitioning within the output of local operators besides layer partitioning to further accelerate inference. We also combined DSCCS with our cache mechanism (referred to as DS-C) to present another perspective about our cache mechanism. We refer to CacheInf as Ours in the tables. Each result in the tables is followed with standard deviation ( $\pm n$ ).

The evaluation questions are as follows:

- RQ1: How much does CacheInf benefit real-world robotic applications by reducing inference time and energy consumption?
- RQ2: How does CacheInf perform on more models common to mobile devices?
- RQ3: How is the above gain achieved in CacheInf and what affects it?
- RQ4: The limitations and potentials of CacheInf.

## 6.2 End-to-End Performance on Real-World Applications

Table 1 shows the inference latency and the ratio that transmission time takes up the inference latency and compared with the baselines (we include results of local computation for comparison, referred to as Local), CacheInf reduced inference latency by 35.5% to 44.4% indoors and 29.4% to 40.0% outdoors for Kapao and 30.0% to 48.8% indoors and 24.2% to 46.8% outdoors for AGRNav. Compared with HP, while CacheInf reduced transmission time by 6 to 8 ms, CacheInf further reduced inference latency by 8 to 10 ms, confirming the effectiveness of the acceleration of the used sparse local operators. CacheInf's highest percentage that transmission time takes up the inference latency across all cases shows that with shrunk transmission data volume with cache enabled and the integration of HP, CacheInf tend to offload

computation to the GPU server more often. This can also be validated by the increased transmission time and reduced inference latency of DSCCS-C compared with DSCCS.

The reduced inference latency of CacheInf leads to reduction of energy consumed per inference by 25.2% to 34.3% indoors and 21.2% to 34.0% outdoors for Kapao and 27.4% to 35.7% indoors and 21.7% to 39.9% outdoors for AGRNav, as shown in Table 2, while the runtime power consumption was increased due to higher frequency of inference.

We report the peak GPU memory consumption on the robot under different strategy in Table 3: CacheInf (includes both the systems of CacheInf and DSCCS-C), No Cache (includes local computation and HP) and Cache All (a naive strategy that caches the output of every layer). The results show that CacheInf increased peak GPU memory consumption by 64.6% for Kapao and 58.5% for AGRNav compared with no cache, which is however 72.2% and 81.6% lower than the cases of Cache All, demonstrating the effectiveness of CacheInf's strategy to reduce the number cached operators.

## 6.3 Performance on Various Common Models

The above conclusions can be further validated by results of a wider range of visual models in Table 4 and Table 5. Across different visual models, CacheInf reduced the inference latency by 13.4% to 43.6% indoors and 13.1% to 45.9% outdoors, and it results in the reduction in energy consumed per inference to be 11.1% to 46.7% indoors and 9.5% to 42.2% compared with the baselines. Note that although CacheInf's gain is still evident, the lower bound of CacheInf's gain decreased on these models compared with Kapao and AGRNav; the reason could be that these models are less computation-intensive, which can be implied from their shorter time for local computation compared with Kapao and AGRNav. When inference of a visual models is not computation-intensive, the gain of using sparse local operators in CacheInf will be limited since execution of each local operator will no longer be the bottleneck. In terms of GPU memory consumption, CacheInf increased GPU memory consumption by 3.2% to 24.8% compared with No Cache, while reducing 12.8% to 39.5% GPU memory consumption compared with Cache All.

## 6.4 Micro-Event

We first present the micro-events about the real-time inference latency of Kapao of different systems under fluctuating bandwidth in Figure 5. And we can learn that CacheInf consistently achieved the lowest inference latency among all the systems and the gain was most significant under lower bandwidth. Then we fixed the wireless network bandwidth to 48Mb/s and examined different systems's performance at varied cache ratios from a sequence of video images in Figure 6: at high cache ratios, CacheInf dramatically reduced inference latency compared with other baselines; at low cache ratios, CacheInf degraded to Hybrid-Parallel or even slightly increased inference latency compared with Hybrid-Parallel,

Model(number of parameters)	Local computation time/s	System	Transmission time/s indoors	Transmission time/s outdoors	Inference time/s indoors	Inference time/s outdoors	Percentage(%) indoors	Percentage(%) outdoors
Kapao(77M)	1.01( $\pm 0.03$ )	DS	0.21( $\pm 0.1$ )	0.24( $\pm 0.12$ )	0.36( $\pm 0.2$ )	0.40( $\pm 0.17$ )	58.33	60.21
		DS-C	0.22( $\pm 0.14$ )	0.25( $\pm 0.12$ )	0.32( $\pm 0.25$ )	0.34( $\pm 0.18$ )	68.75	73.53
		HP	0.24( $\pm 0.15$ )	0.28( $\pm 0.13$ )	0.31( $\pm 0.14$ )	0.34( $\pm 0.12$ )	77.42	82.35
		Ours	0.16( $\pm 0.13$ )	0.21( $\pm 0.18$ )	0.20( $\pm 0.16$ )	0.24( $\pm 0.20$ )	80.09	87.56
AGRNav(0.84M)	0.60( $\pm 0.04$ )	DS	0.10( $\pm 0.05$ )	0.15( $\pm 0.05$ )	0.41( $\pm 0.11$ )	0.47( $\pm 0.12$ )	24.39	31.91
		DS-C	0.13( $\pm 0.07$ )	0.16( $\pm 0.06$ )	0.38( $\pm 0.10$ )	0.43( $\pm 0.13$ )	34.21	37.21
		HP	0.24( $\pm 0.08$ )	0.26( $\pm 0.07$ )	0.30( $\pm 0.09$ )	0.33( $\pm 0.07$ )	78.65	79.47
		Ours	0.18( $\pm 0.08$ )	0.20( $\pm 0.08$ )	0.21( $\pm 0.16$ )	0.25( $\pm 0.18$ )	86.71	80.01

**Table 1.** Average transmission time, inference time, percentage that transmission time accounts for of the total inference time of Kapao and AGRNav in different environments with different systems.

System	Power consumption(W)		Energy consumption(J) per inference		
	indoors	outdoors	indoors	outdoors	
Kapao	Local	9.91( $\pm 0.49$ )	9.91( $\pm 0.49$ )	9.79( $\pm 0.03$ )	9.79( $\pm 0.03$ )
	DS	6.38( $\pm 2.21$ )	6.63( $\pm 2.38$ )	2.30( $\pm 0.55$ )	2.65( $\pm 0.55$ )
	DS-C	6.30( $\pm 2.15$ )	6.53( $\pm 2.12$ )	2.02( $\pm 0.50$ )	2.22( $\pm 0.53$ )
	HP	7.05( $\pm 1.63$ )	6.94( $\pm 0.98$ )	2.19( $\pm 0.62$ )	2.35( $\pm 0.42$ )
	Ours	7.53( $\pm 1.62$ )	7.30( $\pm 0.96$ )	1.51( $\pm 0.60$ )	1.75( $\pm 0.41$ )
AGRNav	Local	8.11( $\pm 0.25$ )	8.11( $\pm 0.25$ )	4.86( $\pm 0.01$ )	4.86( $\pm 0.01$ )
	DS	6.21( $\pm 1.50$ )	7.29( $\pm 1.55$ )	2.55( $\pm 0.19$ )	3.43( $\pm 0.18$ )
	DS-C	6.17( $\pm 1.56$ )	7.00( $\pm 1.43$ )	2.34( $\pm 0.20$ )	3.01( $\pm 0.20$ )
	HP	7.52( $\pm 0.51$ )	8.04( $\pm 0.45$ )	2.26( $\pm 0.15$ )	2.63( $\pm 0.15$ )
	Ours	7.83( $\pm 0.57$ )	8.23( $\pm 0.56$ )	1.64( $\pm 0.17$ )	2.06( $\pm 0.16$ )

**Table 2.** The power consumption against time (Watt) and energy consumption per inference (Joule) of Kapao and AGRNav different environments with different systems.

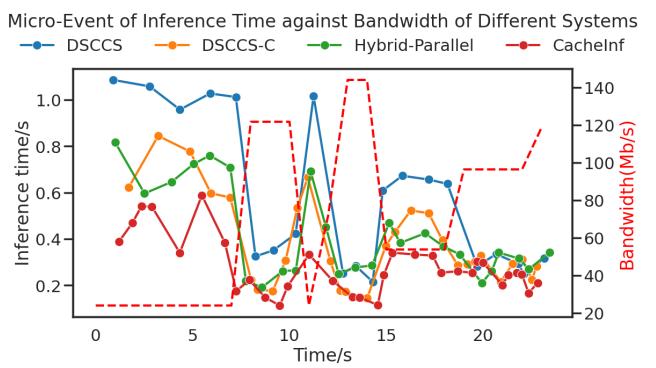
Model(number of parameters)	Memory Consumption(MB)		
	No Cache	Cache All	CacheInf
Kapao(77M)	300.6	1782.5	494.7
AGRNav(0.84M)	82.8	713.3	131.2

**Table 3.** Peak GPU memory consumption of different caching strategy on Kapao and AGRNav.

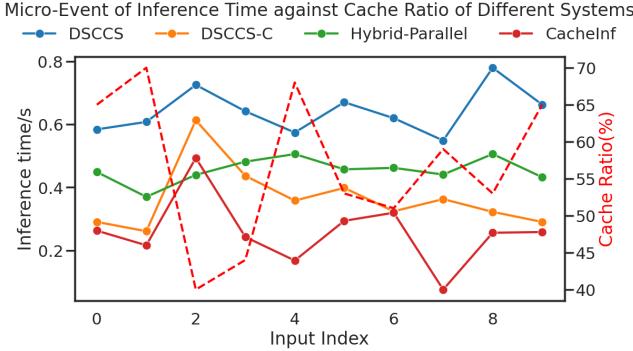
and the reason could be the overhead to analyze reusable cache and update cache. We can also observe when cache ratios fluctuated, the inference latency of CacheInf was more stable than DSCCS-C, which can be attributed to CacheInf's ability to adjust input ratio ( $x$ ) to reduce inference latency.

## 6.5 Sensitivity

With higher difference filtering threshold ( $N$ ), CacheInf will mark cached computation result for areas with more difference as reusable, and we present its influence on the accuracy performance of the selected representative visual models in



**Figure 5.** Kapao: inference latency of different systems at different wireless network bandwidth.



**Figure 6.** Kapao: inference latency of different systems at different cached ratio with fixed wireless network bandwidth.

Table 7. We used the output of the same model with the same input under local computation as the groundtruth to compute accuracy. Each model has a different accuracy metric: AP stands for average precision for people detection for Kapao; F1 examines the portion of points in a point cloud that is close to the groundtruth; Acc@1 is the percentage of the predicted classification results matching with the groundtruth. From Table 7 we can learn that loosening the constraint of

	Model(number of parameters)	Local computation time/ms	System	Transmission time/ms indoors	Transmission time/ms outdoors	Inference time/ms indoors	Inference time/ms outdoors	Percentage(%) indoors	Percentage(%) outdoors
RegNet(54M)	175.0( $\pm 23.6$ )	DSCCS	47.6( $\pm 47.8$ )	60.5( $\pm 54.0$ )	77.8( $\pm 39.3$ )	86.2( $\pm 37.9$ )	61.22	70.22	
		DSCCS-C	50.7( $\pm 49.8$ )	62.5( $\pm 53.6$ )	70.8( $\pm 33.3$ )	79.5( $\pm 39.2$ )	71.61	78.61	
		HP	49.6( $\pm 21.7$ )	59.9( $\pm 23.4$ )	55.0( $\pm 24.8$ )	64.2( $\pm 25.2$ )	90.18	93.34	
		CacheInf	44.2( $\pm 27.7$ )	48.5( $\pm 25.3$ )	45.3( $\pm 35.0$ )	49.2( $\pm 37.2$ )	97.57	98.58	
VGG19(143M)	118.0( $\pm 18.9$ )	DSCCS	38.9( $\pm 47.1$ )	41.6( $\pm 53.8$ )	65.2( $\pm 28.1$ )	75.5( $\pm 27.1$ )	59.75	55.09	
		DSCCS-C	42.7( $\pm 30.2$ )	52.0( $\pm 50.3$ )	53.2( $\pm 33.0$ )	60.3( $\pm 30.9$ )	80.26	86.24	
		HP	44.8( $\pm 20.9$ )	51.5( $\pm 15.0$ )	47.6( $\pm 18.1$ )	53.6( $\pm 14.7$ )	94.15	96.07	
		CacheInf	37.8( $\pm 31.2$ )	43.5( $\pm 13.2$ )	41.1( $\pm 20.3$ )	46.6( $\pm 12.8$ )	94.26	93.34	
ConvNeXt(197M)	316.7( $\pm 31.0$ )	DSCCS	56.0( $\pm 36.1$ )	67.0( $\pm 37.6$ )	79.2( $\pm 35.9$ )	90.6( $\pm 35.4$ )	70.72	73.98	
		DSCCS-C	56.0( $\pm 39.0$ )	63.0( $\pm 30.2$ )	64.7( $\pm 40.2$ )	68.6( $\pm 35.0$ )	86.55	91.84	
		HP	56.4( $\pm 34.7$ )	66.5( $\pm 33.7$ )	59.7( $\pm 26.6$ )	68.0( $\pm 26.6$ )	94.43	97.88	
		CacheInf	40.4( $\pm 37.8$ )	46.9( $\pm 40.0$ )	44.7( $\pm 33.3$ )	49.0( $\pm 30.8$ )	90.38	95.71	

**Table 4.** Average transmission time, inference time, percentage that transmission time accounts for of the total inference time of common visual models in different environments with different systems.

System	Power consumption(W)		Energy consumption(J) per inference		
	indoors	outdoors	indoors	outdoors	
RegNet	Local	9.0( $\pm 0.3$ )	9.0( $\pm 0.3$ )	1.37( $\pm 0.02$ )	1.37( $\pm 0.02$ )
	DS	5.84( $\pm 1.79$ )	5.36( $\pm 1.34$ )	0.45( $\pm 0.14$ )	0.46( $\pm 0.12$ )
	DS-C	6.04( $\pm 1.88$ )	5.96( $\pm 1.45$ )	0.43( $\pm 0.16$ )	0.47( $\pm 0.19$ )
	HP	5.24( $\pm 1.43$ )	5.28( $\pm 1.52$ )	0.29( $\pm 0.08$ )	0.34( $\pm 0.1$ )
	Ours	5.20( $\pm 1.51$ )	5.23( $\pm 1.77$ )	0.24( $\pm 0.08$ )	0.26( $\pm 0.09$ )
VGG19	Local	9.78( $\pm 0.34$ )	9.78( $\pm 0.34$ )	0.95( $\pm 0.02$ )	0.95( $\pm 0.02$ )
	DS	6.58( $\pm 2.14$ )	6.93( $\pm 2.35$ )	0.43( $\pm 0.14$ )	0.52( $\pm 0.18$ )
	DS-C	6.82( $\pm 2.10$ )	7.23( $\pm 2.45$ )	0.36( $\pm 0.18$ )	0.43( $\pm 0.30$ )
	HP	6.51( $\pm 1.74$ )	7.32( $\pm 1.52$ )	0.31( $\pm 0.08$ )	0.39( $\pm 0.08$ )
	Ours	6.70( $\pm 1.88$ )	7.22( $\pm 1.36$ )	0.27( $\pm 0.10$ )	0.34( $\pm 0.09$ )
ConvNeXt	Local	9.92( $\pm 0.38$ )	9.92( $\pm 0.38$ )	3.12( $\pm 0.03$ )	3.12( $\pm 0.03$ )
	DS	5.06( $\pm 0.31$ )	5.02( $\pm 0.37$ )	0.4( $\pm 0.02$ )	0.45( $\pm 0.03$ )
	DS-C	4.86( $\pm 0.44$ )	4.99( $\pm 0.39$ )	0.31( $\pm 0.05$ )	0.34( $\pm 0.09$ )
	HP	4.57( $\pm 0.23$ )	4.54( $\pm 0.25$ )	0.27( $\pm 0.01$ )	0.31( $\pm 0.02$ )
	Ours	5.26( $\pm 0.40$ )	5.39( $\pm 0.27$ )	0.24( $\pm 0.05$ )	0.26( $\pm 0.04$ )

**Table 5.** The power consumption against time (Watt) and energy consumption per inference (Joule) of common visual models in different environments with different systems.

Model(number of parameters)	Memory Consumption(MB)		
	No Cache	Cache All	CacheInf
RegNet(54M)	207.5	427.7	258.9
VGG19(143M)	548.1	668.7	582.8
ConvNeXt(197M)	765.4	1152.7	789.8

**Table 6.** Peak GPU memory consumption of different caching strategy on common visual models.

cache identification by increasing  $N$  slightly decreased accuracy of visual models, with the advantage of further reduced

Model	Statistics	N		
		0.1	0.2	0.3
Kapao	Inference Latency/s	0.20	0.18	0.17
	accuracy (AP)	75.8	74.6	72.5
AGRNav	Inference Latency/s	0.21	0.19	0.17
	accuracy (F1)	98.9	98.5	98.4
ConvNeXt	Inference Latency/ms	44.7	38.6	34.8
	accuracy (Acc@1)	100.0	100.0	99.2

**Table 7.** How different difference filtering threshold (N) for identifying reusable cache affects the inference latency of CacheInf indoors and the accuracy of visual models.

inference latency. And for visual models with pixel level output (e.g., predicted people pose of Kapao and predicted point cloud of AGRNav), such influence will be perceptible, while it does not significantly affect the accuracy of the visual models with comprehensive output (e.g., classification results of ConvNeXt).

## 6.6 Discussion

From the above results we can learn that CacheInf is fundamentally trading-off between GPU memory with inference latency, just as systems in other domains with cache enabled. Since the resulting increased GPU memory consumption may be unfavorable for devices with tight memory budget, adjusting such trade-off to further reduce extra GPU memory consumption to fit in these devices will be our future work. Another limitation of CacheInf is that it relies on continuity of input and thus is unsuitable for scenarios where the perspective of the robot changes dramatically.

## 1211 7 Conclusion

1212 In this paper, we present CacheInf, a collaborative edge-cloud  
 1213 cache system for efficient robotic visual model inference.  
 1214 Based on the continuity of visual input of robots in the field  
 1215 and the local operators commonly used in visual models,  
 1216 we introduce cache mechanism to visual model inference in  
 1217 CacheInf. By reusing computation results of similar local  
 1218 geometries between consecutive inputs, CacheInf accelerates  
 1219 visual model inference by reducing both local computation  
 1220 time and transmission time when offloading computation  
 1221 to the GPU server. The more real-time visual model infer-  
 1222 ence on robots enabled by CacheInf will nurture more visual  
 1223 models to be deployed in real-world robots.

## 1226 References

- [1] iPerf - Download iPerf3 and original iPerf pre-compiled binaries.
- [2] Belhassen Bayar and Matthew C Stamm. Constrained convolutional neural networks: A new approach towards general purpose image manipulation detection. *IEEE Transactions on Information Forensics and Security*, 13(11):2691–2706, 2018.
- [3] Mark Buckler, Philip Bedoukian, Suren Jayasuriya, and Adrian Sampson. EVA<sup>2</sup>: Exploiting temporal redundancy in live computer vision. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 533–546. ISSN: 2575-713X.
- [4] Lukas Cavigelli, Philippe Degen, and Luca Benini. CBinfer: Change-based inference for convolutional neural networks on video data.
- [5] Wongun Choi, Khuram Shahid, and Silvio Savarese. What are they doing? : Collective activity classification using spatio-temporal relationship among people. In *Proc. of 9th International Workshop on Visual Surveillance (VWS09) in conjunction with ICCV*, 2009.
- [6] Reagan L Galvez, Argel A Bandala, Elmer P Dadios, Ryan Rhay P Vicerra, and Jose Martin Z Maningo. Object detection using convolutional neural networks. In *TENCON 2018-2018 IEEE Region 10 Conference*, pages 2023–2027. IEEE, 2018.
- [7] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.*, 38(6), nov 2019.
- [8] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 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*, pages 82–95. ACM.
- [9] Huanghuang Liang, Qianlong Sang, Chuang Hu, Dazhao Cheng, Xiaobo Zhou, Dan Wang, Wei Bao, and Yu Wang. Dnn surgery: Accelerating dnn inference on the edge through layer partitioning. *IEEE transactions on Cloud Computing*, 2023.
- [10] Chao Ma, Jia-Bin Huang, Xiaokang Yang, and Ming-Hsuan Yang. Hierarchical convolutional features for visual tracking. In *Proceedings of the IEEE international conference on computer vision*, pages 3074–3082, 2015.
- [11] William McNally, Kanav Vats, Alexander Wong, and John McPhee. Rethinking keypoint representations: Modeling keypoints and poses as objects for multi-person human pose estimation. In *European Conference on Computer Vision*, pages 37–54. Springer, 2022.
- [12] NVIDIA. Using Shared Memory in CUDA C/C++. <https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/>, January 2013.
- [13] NVIDIA. Can Jetson Orin support nccl? - Jetson & Embedded Systems / Jetson Orin NX. <https://forums.developer.nvidia.com/t/can-jetson-orin-support-nccl/232845>, November 2022. Section: Autonomous Machines.
- [14] NVIDIA. The world's smallest ai supercomputer. <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/>, 2024.
- [15] Keiron O’shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.
- [16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
- [17] F. Perazzi, J. Pont-Tuset, B. McWilliams, L. Van Gool, M. Gross, and A. Sorkine-Hornung. A benchmark dataset and evaluation methodology for video object segmentation. In *Computer Vision and Pattern Recognition*, 2016.
- [18] PyTorch. Distributed communication package - torch.distributed – PyTorch 2.3 documentation. <https://pytorch.org/docs/stable/distributed.html>, 2024.
- [19] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *IEEE transactions on Evolutionary Computation*, 13(2):398–417, 2008.
- [20] Lingyan Ran, Yanning Zhang, Qilin Zhang, and Tao Yang. Convolutional neural network-based robot navigation using uncalibrated spherical images. *Sensors*, 17(6):1341, 2017.
- [21] Waseem Rawat and Zhengui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural computation*, 29(9):2352–2449, 2017.
- [22] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.
- [23] Zekai Sun, Xiuxian Guan, Junming Wang, Haoze Song, Yuhao Qing, Tianxiang Shen, Dong Huang, Fangming Liu, and Heming Cui. Hybrid-parallel: Achieving high performance and energy efficient distributed inference on robots, 2024.
- [24] torchvision. torchvision – Torchvision 0.13 documentation. <https://pytorch.org/vision/0.13/>, 2024.
- [25] Bryan Tripp. Approximating the architecture of visual cortex in a convolutional network. *Neural computation*, 31(8):1551–1591, 2019.
- [26] Junming Wang, Zekai Sun, Xiuxian Guan, Tianxiang Shen, Zongyuan Zhang, Tianyang Duan, Dong Huang, Shixiong Zhao, and Heming Cui. Agrnav: Efficient and energy-saving autonomous navigation for air-ground robots in occlusion-prone environments. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2024.
- [27] Panqu Wang, Pengfei Chen, Ye Yuan, Ding Liu, Zehua Huang, Xiaodi Hou, and Garrison Cottrell. Understanding convolution for semantic segmentation. In *2018 IEEE winter conference on applications of computer vision (WACV)*, pages 1451–1460. Ieee, 2018.
- [28] Sanghyun Woo, Shoubhik Debnath, Ronghang Hu, Xinlei Chen, Zhuang Liu, In So Kweon, and Saining Xie. Convnext v2: Co-designing and scaling convnets with masked autoencoders. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16133–16142, 2023.
- [29] Min Wu, Wanjuan Su, Luefeng Chen, Zhentao Liu, Weihua Cao, and Kaoru Hirota. Weight-adapted convolution neural network for facial expression recognition in human–robot interaction. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(3):1473–1484, 2019.
- [30] Jing Xu, Yu Pan, Xinglin Pan, Steven Hoi, Zhang Yi, and Zenglin Xu. Regnet: self-regulated network for image classification. *IEEE Transactions on Neural Networks and Learning Systems*, 2022.
- [31] Xinlei Yang, Hao Lin, Zhenhua Li, Feng Qian, Xingyao Li, Zhiming He, Xudong Wu, Xianlong Wang, Yunhao Liu, Zhi Liao, et al. Mobile access bandwidth in practice: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 114–128, 2022.