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

Abstract

Real-time visual model inference is crucial for various robotic tasks deployed on mobile robots in the field, but limited onboard computation power and limited and unstable wireless network bandwidth of the mobile robot constraint the speed of either local computation or offloading (e.g., to remote GPU servers) of the visual model inference; the prolonged inference time also increases energy consumption for the inference on each input. We observe that visual model inference typically computes on local geometries and consecutive inference inputs often partially share similar local geometries, which provides opportunity to cache and reuse the computation results to both speed up local computation and offloading.

Based on these observations, we propose CacheInf, a collaborative edge-cloud cache system for efficient robotic visual model inference. CacheInf first profiles the visual model and schedules for optimal offloading / local computation plan at different ratio of reusable cache. At runtime, it analyses the incoming inference input and manages reusable cache on both the robot and the server; then CacheInf dispatches local computation and offloading on reduced input size by reusing cached computation results both on the robot and the server, which accelerates both local computation and offloading. Evaluation on various visual models and wireless network environments shows that CacheInf reduced average end-to-end inference latency by TODO% to TODO% and reduced average energy consumption for inference on each input by TODO% to TODO%.

1 Introduction

Visual information is vital for various robotic tasks deployed on real-world edge devices (typically mobile robots), such as surveillance, health care and autonomous vehicles; and as a major visual information processing method, fast visual model inference is important for the real-world robotic tasks to timely respond to environment changes. Unfortunately, the mobile robots often suffer slow visual model inference, because they are typically limited in computation power and have limited and unstable wireless network bandwidth, making both local computation and naively offloading the computation to GPU servers slow.

To tackle this problem, we observe that introducing the classic caching mechanism into robotic visual model inference has the potential to accelerate both local computation and offloading the computation to GPU servers. It is based on the facts that the input for the robotic visual models is typically a continual stream of images and these models mostly compute on the images using local operators (i.e., operators such as convolution that relies on local geometries of the input images). In such cases, computation results of similar local geometries between consecutive images can be reused to shrink the overall size of the computation, which reduce both local computation time and transmission time when offloading computation to the GPU server.

However, the existing caching systems for visual models are designed for interactive generative image edition on high-end PCs and unfit for robotic tasks on mobile robots. With their targeted scenario, they assume no perspective changes in the images, consume too much memory and does not consider the acceleration opportunity of offloading the computation to other GPU servers. A new caching system specifically designed for robotic visual model inference is desired.

To fulfill this gap, in this paper, we propose CacheInf, a collaborative edge-cloud cache system for efficient robotic visual model inference. Given a continuous stream of visual input in a robotic visual task, CacheInf analyses the overlapping area between consecutive inputs; based on the portion of overlapping area (reusable cache) and the current estimated wireless network bandwidth, CacheInf schedules on the action between reusing local cache to reduce local computation time and reusing the remote cache (e.g., the cache at the GPU server side from the robot's perspective) to reduce transmission time, to ultimately reduce the overall visual model inference latency.

The design of CacheInf is non-trivial. The first challenge is to transform cached results to local computation acceleration. While the computation result of cached areas can be reused, the sparse and fragmented remaining un-cached area can not be computed on the highly optimized local operators for dense local geometries, hindering acceleration . To make use of these highly optimized local operators, in CacheInf we search for a minimal set of rectangles that covers the uncached areas and recombine them to a new dense rectangle while preserving local geometries. Computation results on the new rectangle can then be broken up and combined with the cache to form the correct result.

The second challenge is how to reduce the cache memory consumption, especially at the robot side which typically has a tight GPU memory budget. In visual models, the computation results of local operators typically consumes

significantly more memory than the parameters of local operators and naively caching all the computation results leads to heavy GPU memory burden.

We observe that in visual models, the computation result of a local operator often serves as the input of another local operator. In this case, the computation result of the combined rectangle of uncached areas can be directly passed to the following local operators without loss of information, and cache between these two operators can be ignored to reduce memory consumption. We greedily search for a continuous sequence of local operators whose starting and ending operators incur the least memory consumption and cache the computation results of the starting and ending operators only, so as to minimize cache memory consumption.

To fully exploit the potential acceleration of cache and offloading, we also integrate an emerging offloading paradigm named Hybrid-Parallel [21]: during visual model inference on an image, Hybrid-Parallel enables splitting of the input of local operators and assigns different splits to the local robot and the remote GPU server for computation, so that local computation and data transmission of one image can be parallelized to reduce inference latency. We extend the scheduler of Hybrid-Parallel to further consider the potential acceleration with cache, such the splitting and assigning will benefit (TODO) from cache on both the robot and the server.

We implemented CacheInf using python and pytorch on Ubuntu20.04. The offloading of computation is handled by the highly optimized distributed module of pytorch with cuda-aware mpi backend which directly accesses GPU buffer, so as to minimize offloading overhead. Our baselines include plain local computation, a state-of-the-art computation offloading system named TODO, together with its counterpart with cache enabled modified by us, and Hybrid-Parallel.

We evaluated CacheInf on a four-wheel robot equipped with a Jetson NX Xavier that is capable of computing locally with the low power consumption GPU. The offloading GPU server is a PC equipped with an Nvidia 2080ti GPU. Our datasets include the standard datasets of video frames of DAVIS [15] and CAD [2] each captured by a hand held camera and our self captured video frames using sensors on our robot. Extensive evaluation on various visual models and wireless network bandwidth circumstances shows that:

- CacheInf is fast. Among the baselines, CacheInf reduced the end-to-end inference time by XX% to XX%.
- CacheInf saves energy. Among the baselines, CacheInf reduced the average energy consumed to complete inference on one image by XX% to XX%.
- CacheInf is also memory-efficient. The above advantages were obtained by only incurring XX% to XX% increase in memory consumption for CacheInf.

The major contribution of this paper is our new edge-cloud collaborative caching paradigm to accelerate robotic visual model inference, which reuses cached computation results to both accelerate local computation and computation offloading to remote GPU servers. The resulting system, CacheInf, collaboratively considers and reuses cached computation results on both the robot and the server and schedules the computation and offloading to minimize visual model inference latency. The accelerated visual model inference and the reduced power consumption will make real-world robots more performant on various robotic tasks and nurture more visual models to be deployed in real-world robots. The source code and evaluation logs of CacheInf is available at TODO.

The rest of the paper is organized as follows. Chapter two introduces background and related work. Chapter three gives an overview of CacheInf and Chapter four presents its detailed design. Chapter five describes implemented. Chapter six presents our evaluation results and CacheInf seven concludes.

2 Background

2.1 Vision tasks on robots

Vision tasks play a crucial role in enabling robots to perceive, understand, and interact with their environment. Visual information is essential for various robotic tasks, such as object recognition, navigation, manipulation, and human-robot interaction. 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.

To address these challenges, various approaches have been proposed to optimize the inference speed of deep learning models on resource-constrained robots. These approaches include model compression techniques, such as pruning and quantization, which aim to reduce the size and computational complexity of visual models while maintaining their performance. Other techniques, such as model distillation and neural architecture search, focus on designing compact and efficient network architectures specifically tailored for robotic applications. These methods achieve faster inference speed through model modifications by potentially sacrificing some accuracy, and the trade-off between model accuracy and inference speed needs to be carefully considered, as sacrificing too much accuracy for the sake of speed may lead

to suboptimal performance in critical robotic tasks, which is orthogonal to this paper.

Despite these efforts, the deployment of deep learning models for real-time visual inference on robots remains a challenging task. The limited computational resources, power constraints, and the need for low-latency processing pose significant hurdles in achieving fast and reliable visual model inference on robots. This paper presents a new method to optimize inference process based on the cache mechanism.

In summary, vision information and fast visual model inference are crucial for the success of many robotic tasks. The deployment of deep learning models on resource-constrained robots requires careful consideration of inference speed and computational resources. 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 have become a fundamental building block in visual models (deep learning models for computer vision tasks). These layers are widely used in various vision applications, such as image classification, object detection, and semantic segmentation, due to their ability to effectively capture and learn spatial hierarchies of features from raw input images.

Inspired by the biological structure of the visual cortex, convolutional 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. This enables deep learning models to capture translation-invariant features, meaning they can detect and recognize visual patterns regardless of their position in the input image.

Convolutional layers also learn hierarchical representations of visual features, with early layers learning low-level features like edges and corners, and deeper layers learning more complex patterns and object parts. This hierarchical learning process allows deep learning models to effectively capture and represent intricate visual patterns in the input data.

The use of convolutional layers has led to significant breakthroughs in various CV tasks, with deep convolutional neural networks (CNNs) achieving state-of-the-art performance in image classification, object detection, and semantic segmentation. 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.

Notice that operators for DNN layer (e.g., convolution, ReLU, softmax) can be categorized into two types: local operators and global operators, depending on whether they can be computed independently with partial input according to [21], and the convolutional layer is a typical local operator, as it can be computed with partial input tensor (the blocks in the input tensor for convolution).

2.3 Resource Limitations of Robots

In real-world robotic Internet of Things (IoT) scenarios, devices often navigate and move around to perform tasks such as search and exploration. While wireless networks provide high mobility, they also have limited bandwidth, which can significantly impact the performance of robotic IoT systems.

wireless transmission of robots is constrained by limited bandwidth, both due to the theoretical upper limit of wireless transmission technologies and the practical instability of wireless networks. The most advanced Wi-Fi technology, Wi-Fi 6, offers a maximum theoretical bandwidth of 1.2 Gbps for a single stream [7]. However, the limited hardware resources on robots often prevent them from fully utilizing the potential of Wi-Fi 6[25]. Moreover, the actual available bandwidth of wireless networks is often reduced in practice due to various factors, such as the movement of devices [8, 14], occlusion by physical barriers [3, 19], and preemption of the wireless channel by other devices [1, 18], which demonstrate the instability of wireless transmission in [21]. This limitation on the bandwidth of robots' wireless network poses significant challenges for the efficient and reliable operation of robots in real-world scenarios, particularly in outdoor environments where the instability of wireless networks is more pronounced.

2.4 Related Work

offloading methods; intra-DP

Cache related (Guan will take it)

3 System Overview

The chapter presents an overview of the design of CacheInf.

3.1 Working Environment and Metrics

We assume that the working scenario of CacheInf is a mobile robot performing robotic tasks in a real-world field which requires real-time visual model inference on the continuous image stream captured from the on-board camera, to achieve real-time response to various environment changes. The robot itself is equipped with low-power-consumption gpu to perform visual model inference which is slow and consumes too much power; it has wireless network access to a remote powerful GPU server that provides opportunity of acceleration, but the connection suffers from limited and unstable wireless network bandwidth.

While the requirements of real-time inference does not necessarily imply the requirement of high inference frequency, we measure the real-time metric by the average end-to-end inference latency when the robot is seamlessly performing inference, which leads to high inference frequency;

the power consumption is also measured by average power consumption to finish inference on each image in the same scenario.

3.2 Architecture of CacheInf

To reduce inference latency by reusing cached previous computation results to both reduce local computation time and transmission time of offloading, CacheInf basically consists of three blocks: a scheduler, a cache tracker and an executor (TODO fix the names).

3.2.1 Scheduler (TODO fix the names). During the initialization stage of the robotic task and CacheInf, CacheInf is granted access to the visual model and an initial input image and we mainly greedily pre-compute a schedule of various situations at this stage, since scheduling at runtime affects the real-time performance of the robotic task. We first profile the model at both the robot and the remote GPU server to gather information including shapes of the computation intermediates, the execution time of each operator (e.g., convolution, linear, etc.) on various scale of the input (e.g., from one tenth of the image to full scale of the image), the local property of each operator (i.e., whether the operator performs local computation) and so on.

Based on the above information, CacheInf finds sets of continuous local operators and assign the operators with smallest output sizes to be the operators to cache their computation results to reduce memory consumption of cache. Then we coarsely iterate through the possible wireless network bandwidth, distribution of cache between the robot and the server and the portion of reusable cache and greedily compute a plan of whether to compute on cache and the portion of local computation and offloaded computation at the server at the reduced transmission data volume reusing cache. We use the greedy strategy because we assume that both the wireless network bandwidth and the portion of reusable cache is unpredictable in the real-world scenario. Note that the precomputed schedule can be reused for a same visual model with the same settings.

3.2.2 Cache Tracker. At runtime, the selected operators at the previous stage will cache their computation results and the cache tracker identifies the reusable portion of such cached computation results. Given an input image, we extract and store its features using classic computation vision methods (we choose Flann algorithm in our implementation, which is state-of-the-art). For a current next image, we also extract and store its features and match them with the previous features (e.g., using KNN algorithm) and compute a perspective transform between the two images, which transforms the previous image such that the transformed previous image partially overlaps the current image and the non-overlapping areas are also marked. The features of the previous images is then discarded. The same transform can also be applied to the cached computation results since they are computed by local operators that keep the local geometries of the input image, and thus the reusable cached computation results are identified. Note that the computation involved in this process is light-weight compared with the visual model inference that typically involves hundreds of operators.

3.2.3 Executor (TODO fix the names). The executor is responsible to actually select and execute an plan based on results of the above two processes at runtime. First, we further estimate the actual possible speedup by reusing cache, because the areas without cache needed for computation are often sparse and fragmented. We cluster the areas without cache into different nearest clusters and compute minimum bounding boxes for each of the clusters; then we greedily break up and recombine these bounding boxes to form a minimum new rectangle as a temporary input for the local operators and estimate its execution time based on its shape and the profile results from the initialization stage and select a precomputed plan for this input shape. If no evident speedup, we will ignore the cache and use the whole input.

With a selected plan where cache is enabled, the executor reuses the temporary input of reorganized areas without cache described above and feed it into the inference pipeline; it also handles the portion of local computation and the portion of offloaded computation to the remote GPU server. When appropriate, the executor breaks up the computation results of the temporary input and combines them with the transformed cache to recover geometries of the input image to get the correct result. With a selected plan where cache is disabled, the actions with cache involved are excluded, but note that in any cases, the cache at the remote GPU server is always reused to reduce transmission data volume.

4 Design

This chapter presents the detailed design for CacheInf to fulfill the functionality of tracking and reusing cached computation results and scheduling for actions among local computation, offloading or hybrid, with or without cache, to optimally reduce visual model inference latency for mobile robots.

4.1 Identifying Reusable Computation Results

To find and match similar local geometries between consecutive images in a stream of images $\boldsymbol{I} = \{I_1, I_2, ..., I_n\}$ to identify reusable cache, we use the standard image stitching procedure: given a pair of consecutive images I_j and I_{j+1}, their key points and key point descriptors (or feature vectors) are computed and matched within a distance threshold of the feature vectors; then a homography matrix M is computed based on the corresponding relationship between the key points on each image which minimizes the error. The resulting homography matrix is then used to apply perspective transformation to each pixel in I_j to form a new image $\hat{I}_{j+1}$

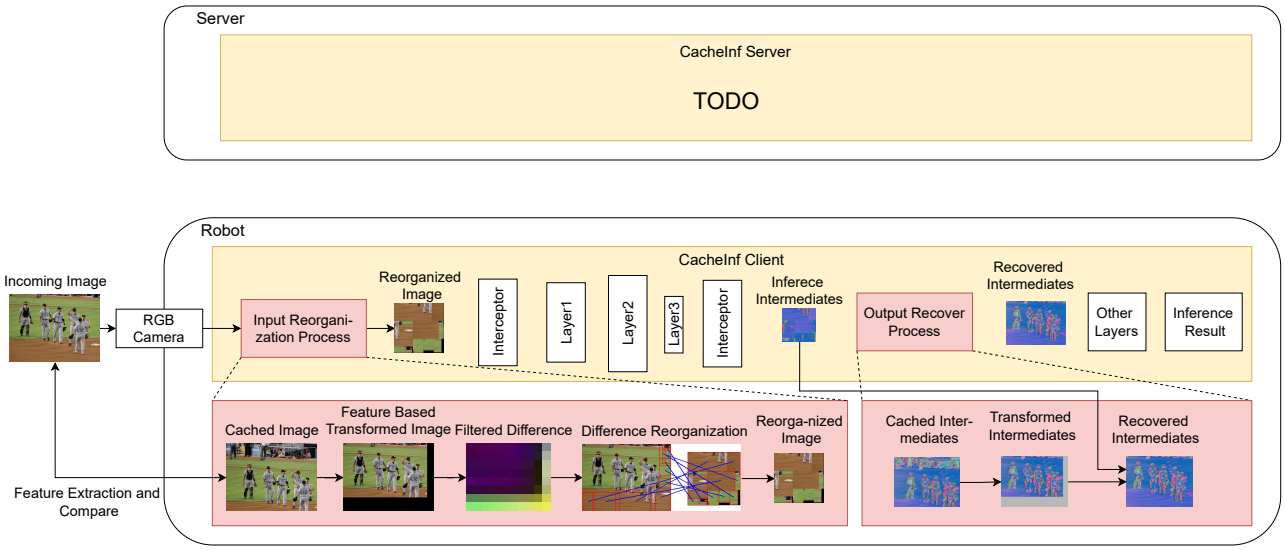


Figure 1. Architecture and working process of CacheInf. TODO...

closest to I_{j+1} as shown in Equation 1, where (u_j, v_j) and $(\hat{u}_{j+1}, \hat{v}_{j+1})$ and pixel indices on I_j and $\hat{I}_{j+1}$. It is also depicted in the Feature Based Transformed Image in Fig. 1. Since the computation of local operators relies on local geometries, the same transformation can be applied to intermediate computation results of the following local operators.

$$(\hat{u}_{j+1}, \hat{v}_{j+1}, 1) = M \times (u_j, v_j, 1) \tag{1}$$

While the above process minimizes error between $\hat{I}_{j+1}$ and I_{j+1}, the remaining difference between them are the areas of new information which are uncached and needed to be recomputed. We filter and identify these areas by applying average pooling over the difference between $\hat{I}_{j+1}$ and I_{j+1} and the pixels with computed difference greater than a preset threshold N will be marked as needed to be recomputed as in Equation 2, where u, v are the pixel indices.

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

Suppose there are Q pixels in $\boldsymbol{uv}$ and HxW total pixels in each image, we define the cache ratio between I_j and I_{j+1} as

$$r_j = \frac{Q}{H \cdot W} \tag{3}$$

4.2 Sparse Local Operators

From the above discussion, we have identified the pixels needed to recompute $\boldsymbol{uv}$ and we suppose their corresponding features f_{inp} are of size BxC_1xQ, along with the cached input defined as I'_{inp} of size BxC_1xHxW. Now we focus on how to compute the correct results based $\boldsymbol{uv}$, 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 BxC_1xHxW, a convolution local operator K with its kernel sized $C_2xC_1xK_1xK_2$, stride 1 and no padding and its output feature map I_{out} of size $BxC_2xH'xW'$, then each of the value of the output feature map is determined by

$$I_{out}^{i,j,k,l} = \sum_{c=1}^{C_1} \sum_{m=1}^{K_1} \sum_{n=1}^{K_2} K^{j,c,m,n} * I_{inp}^{i,c,k+m-1,l+n-1}, \tag{4}$$

Omitting the batch dimension and the channel dimension (first two dimension) of I_{out}, we can learn from Equation 4 that an output value is determined by an area of K_1xK_2 on I_{inp} and we define pixels in this area as

$$P_{k,l} = \{(u, v) | k \leq u < k + K_1 \wedge l \leq v < l + K_2\} \tag{5}$$

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 $\boldsymbol{uv}$ as input would be

$$\boldsymbol{uv'} = \{(k, l) | \exists P_{k,l}, s.t. P_{k,l} \cap \boldsymbol{uv} \neq \emptyset\} \tag{6}$$

which can be view as wrapping around pixels in $\boldsymbol{uv}$ by K_1xK_2 and may involve pixels in I'_{inp}.

Note that $\boldsymbol{uv}$ 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:

$$F(i,j,u,v,I'_{inp},f_{inp}) = \begin{cases} f_{inp}^{i,j,u,v}, & (u,v) \in \boldsymbol{uv}, \\ I'^{i,j,G(u,v,M)}_{inp}, & (u,v) \notin \boldsymbol{uv} \end{cases} \tag{7}$$

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. 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.

For $(u,v) \in \boldsymbol{uv'}$

$$f_{out}^{i,j,u,v} = \sum_{c=1}^{C_1} \sum_{m=1}^{K_1} \sum_{n=1}^{K_2} K^{j,c,m,n} \cdot F(i,c,k+m-1,l+n-1,I'_{inp},f_{inp}) \tag{8}$$

Until now we get the indices of the altered output values in output feature map $\boldsymbol{uv'}$ 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 $\boldsymbol{uv}$, which recovers the correct geometries of the whole feature map.

Also, to save memory consumption of cached intermediates, notice that the above process is basically wrapping the sparse pixels with the kernel size K_1xK_2 and computing on the wrapped pixels, we can merge the query process in Equation 7 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'_1xK'_2, we can wrap the sparse pixels with an extended kernel size $(K_1+K'_1)$x$(K_2+K'_2)$ 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

In the above discussion we have analyzed the opportunity for visual model inference acceleration by reusing previous computation result. In a edge-cloud collaborative system as CacheInf, reusing previous computation result has the potential to not only reduce transmission time by reducing transmission data volume, but also reduce local computation time by shrinking computation size using sparse local operators, and the extend of such reduction is determined by cache ratio of the current input r. Similar to Hybrid-Parallel [21], we define all the operators involved in a visual model as $\boldsymbol{O} = \{o_1, o_2, ..., o_n\}$ and the portion of locally executed input of each operator will be $\boldsymbol{X} = \{x_1, x_2, ..., x_n\}$, $0 \le x_i \ge 1$ and $1 - x_i$ represents the the portion of input executed on the GPU server. The indices of local operators is defined as $\boldsymbol{O}_l$.

In CacheInf, while offloading, we transmit the sparse features together with their indices encoded into a bit-mask and the transmission data volume is almost inversely proportional to cache ratio: assume wholely offloading computation of a visual model from a layer will require transmission of $m = B$xCxHxW element and each element is a float32 number, summing up to 4m bytes data; in the cached case with cache ratio r, the data volume needed to transmit will be $(1-r)4m + \frac{m}{8BxC}$ bytes, where the latter term is the bit-mask of HxW encoding the indices of the transmitted pixels on the whole feature map. When integrating Hybrid-Parallel that would slice the sparse features, we simply slice the sparse features and only encode the bit-mask for indices of the slice for the server, while preserving the features and indices for the robot.

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, input shapes and so on. Thus we profile such relationship by altering the cache ratio and x_i and recording 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 (indices $\boldsymbol{O}_{nonlocal} = \{i | 1 \le i \le n \wedge i \notin \boldsymbol{O}_l\}$), we constraint that both $T(\cdot)$ and $U(\cdot)$ returns time of x_i to be either 0 or 1 and r to be 0.

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 $\hat{\boldsymbol{O}}_c$ and the resulting reduction of cache ratio (since extra input will be included) of each operator as o_I to be $R(\hat{\boldsymbol{O}}_c, o_i)$. 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 $\forall x_i \in \boldsymbol{X}, x_i = 1$. In this case the execution time of every operator will be $T_c(o_i, 1, r - R(\hat{\boldsymbol{O}}_c, o_i))$, and the

optimization problem will be

$$\min_{\hat{\boldsymbol{O}}_c \subset \boldsymbol{O}_l} \frac{1}{w} \sum_{i=1}^{n} \sum_{j=0}^{w} T_c(o_i, 1, r_j + R(\hat{\boldsymbol{O}}_c, o_i)) + U_{sum}(\hat{\boldsymbol{O}}_c, r_j) \quad (9)$$

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

Solving of this optimization problem seeks the optimal choices of cache operators $\hat{\boldsymbol{O}}_c$ 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 9 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 $o_i \in \boldsymbol{O}$ 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. Note that we insert an operator to analyze the image input as the first operator which will always be executed at the robot.

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:

$$t_i^c = \begin{cases} T_c(o_i, x_i, r_c - R(\hat{\boldsymbol{O}}_c, o_i)), & i = 1 \\ t_{i-1}^c + T_c(o_i, x_i, r_c - R(\hat{\boldsymbol{O}}_c, o_i)), & 1 < i \le n \wedge x_i \le x_i' \\ max(T_c(o_i, x_i, r_c - R(\hat{\boldsymbol{O}}_c, o_i)) + \\ \quad t_{i-1}^c, \ \frac{1}{b} D(o_i, x_i' - x_i, r) + t_i^s), & 1 < i \le n \wedge x_i > x_i' \end{cases} \quad (10)$$

$$t_i^s = \begin{cases} 0, & i = 1 \\ t_{i-1}^s + T_s(o_i, 1 - x_i, r_s - R(\hat{\boldsymbol{O}}_c, o_i)), & 1 < i \le n \wedge x_i \ge x_i' \\ max(T_s(o_i, 1 - x_i, r_s - R(\hat{\boldsymbol{O}}_c, o_i)) + \\ \quad t_{i-1}^s, \ \frac{1}{b} D(o_i, x_i' - x_i, r_s) + t_i^c), & 1 < i \le n \wedge x_i < x_i' \end{cases} \quad (11)$$

The first two rows of both Equation 10 and 11 describes 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 third row instead describes 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

$$\begin{aligned} \min_{\forall 1 \le i \le n, 0 \le x_i \le 1} \ & t_n^c \\ s.t. \quad & x_1 = x_n = 1 \\ & \forall j \in \boldsymbol{O}_{nonlocal}, x_j (mod \ 1) = 0 \\ & \forall j \notin \hat{\boldsymbol{O}}_c, x_j = x_j' \end{aligned} \quad (12)$$

In Equation 12, 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 9 and 12 with the differential evolution algorithm [17] and store the solutions of different bandwidth and cache ratios of Equation 12 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 $Cache[1]$ 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. The 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

Algorithm 1: CacheInfClient

Input: A continual sequence of video images $\boldsymbol{I}$; DNN model M
Output: The inference results ret on each image in $\boldsymbol{I}$

// profile
$T_c, U_c = Profile(M)$
$Send(M, T_c, U_c)$
$Schedule, \hat{\boldsymbol{O}}_c = Receive()$
$Cache = InitCache(\hat{\boldsymbol{O}}_c)$
// inference
foreach I *in* $\boldsymbol{I}$ **do**
  $b = EstimateBandwidth()$
  $r_c, r_s = AnalyzeCacheRatio(I, Cache[1])$
  $inp, M = IdentifyCache(I, Cache[1])$
  $Send(b, r_c, r_s, M)$
  $x, x' = Schedule[b, r_c, r_s]$
  foreach $i = 1, 2, ..., n$ **do**
    if $i \in \boldsymbol{O}_{nonlocal}$ *and* $x_i > 0$ *and IsSparse(inp)* **then**
      $inp = DenseRecover(inp, Cache[i], M)$
      $UpdateCache(Cache[i], inp, M)$
    end
    else if $x'_i > 0$ *and* $i \in \hat{\boldsymbol{O}}_c$ **then**
      $inp = SparseGather(inp, Cache[i], M)$
      $UpdateCache(Cache[i], inp, M)$
    end
    if $x_i < x'_i$ **then**
      $inp, inp' = Slice(inp, x_i, x'_i)$
      $Send(inp')$
    end
    else if $x_i > x'_i$ **then**
      $inp = Merge(inp, Receive())$
    end
    if $x_i > 0$ **then**
      if $IsSparse(inp)$ **then**
        $inp = SparseExecute(o_i, inp)$
      end
      else
        $inp = Execute(o_i, inp)$
      end
    end
  end
  $ret[I] = inp$
end
return ret

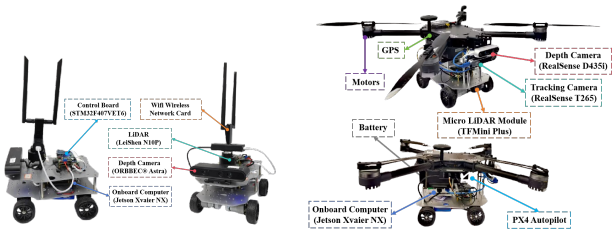
sparse input and choose the original operator for dense input at line 27 to 34.

Algorithm 2: CacheInfServer

// profile and compute schedule at the server
$M, T_c, U_c = Receive()$
$T_s, U_s = Profile(M)$
$Schedule, \hat{\boldsymbol{O}}_c = ComputeSchedule(T_s, U_s, T_c, U_c)$
$Send(Schedule, \hat{\boldsymbol{O}}_c)$
$Cache = InitCache(\hat{\boldsymbol{O}}_c)$
// inference
while *True* **do**
  $b, r_c, r_s, M = Receive()$
  $x, x' = Schedule[b, r_c, r_s]$
  foreach $i = 1, 2, ..., n$ **do**
    if $i \in \boldsymbol{O}_{nonlocal}$ *and* $x_i < 1$ *and IsSparse(inp)* **then**
      $inp = DenseRecover(inp, Cache[i], M)$
      $UpdateCache(Cache[i], inp, M)$
    end
    else if $x'_i < 1$ *and* $i \in \hat{\boldsymbol{O}}_c$ **then**
      $inp = SparseGather(inp, Cache[i], M)$
      $UpdateCache(Cache[i], inp, M)$
    end
    if $x_i > x'_i$ **then**
      $inp, inp' = Slice(inp, x_i, x'_i)$
      $Send(inp')$
    end
    else if $x_i < x'_i$ **then**
      $inp = Merge(inp, Receive())$
    end
    if $x_i < 1$ **then**
      if $IsSparse(inp)$ **then**
        $inp = SparseExecute(o_i, inp)$
      end
      else
        $inp = Execute(o_i, inp)$
      end
    end
  end
end

5 Implementation

We implemented CacheInf with python, pytorch [13] and taichi [4] on Ubuntu20.04. The communication library used is the distributed module [16] of pytorch with mpi backend. 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 [11].



(a) Four-wheeled robot **(b)** Air-ground robot

Figure 2. The composition of the four-wheeled robot and the air-ground robot used in our evaluation.

The sparse local operators were implemented based on the bitmasked sparse nodes in taichi [4], which efficiently manages the sparse pixels in a grid and preserves the spatial structure of the sparse pixels by organizing the 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 [10] can be introduced to accelerate computation; and our implemented sparse local operators achieved comparable performance with original pytorch operators with the same input size.

6 Evaluation

6.1 Evaluation Settings

Testbed.

The evaluation was conducted on a custom four-wheeled robot (Fig 2), and a custom air-ground robot(Fig **??**). They are equipped with a Jetson Xavier NX [12] 8G onboard computer that is capable of AI model inference with local computation resources. The system runs Ubuntu 20.04 with ROS Noetic and a dual-band USB network card (MediaTek MT76x2U) for wireless connectivity. The Jetson Xavier NX interfaces with a Leishen N10P LiDAR, ORBBEC Astra depth camera, and an STM32F407VET6 controller via USB serial ports. Both LiDAR and depth cameras facilitate environmental perception, enabling autonomous navigation, obstacle avoidance, and SLAM mapping. The GPU server is a PC 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 in our experiments.

Tab. 1 presents the overall on-board energy consumption (excluding motor energy consumption for robot movement) of the robot in various states: inference (model inference with full GPU utilization, including CPU and GPU energy consumption), transmission (communication with the GPU server, including wireless network card energy consumption), and standby (robot has no tasks to execute). Notice that different models, due to varying numbers of parameters, exhibit distinct GPU utilization rates and power consumption during inference.

	inference	transmission	standby
Power (W)	13.35	4.25	4.04

Table 1. Power consumption (Watt) of our robot in different states.

Workload. We evaluated two typical real-world robotic applications on our testbed: Kapao, a real-time people-tracking application on our four-wheeled robot (Fig 3), and AGRNav, an autonomous navigation application on our air-ground robot (Fig 4). These applications feature different model input and output size patterns: Kapao takes RGB images as input and outputs key points of small data volume. In contrast, AGRNav takes point clouds as input and outputs predicted point clouds and semantics of similar data volume as input, implying that AGRNav needs to transmit more data during distributed inference. And we have verified several models common to mobile devices on a larger scale to further corroborate our observations and findings: DenseNet [5], VGGNet [20], ConvNeXt [23], RegNet [24].

Experiment Environments. We evaluated two real-world environments: indoors (robots move in our laboratory with desks and separators interfering with wireless signals) and outdoors (robots move in our campus garden with trees and bushes interfering with wireless signals, resulting in lower bandwidth). The corresponding bandwidths between the robot and the GPU server in indoors and outdoors scenarios are shown in Fig. **??**.

Baselines. We selected two SOTA inference acceleration methods as baselines: DSCCS [6], aimed at accelerating inference, and Hybrid-Parallel [21] (referred to as HP), that parallelizes local computation and offloading to further acceleratio inference. We also combined DSCCS with our cache mechanism (referred to as DSCCS-C) to present another perspective about CacheInf's performance gain.

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: What are the limitations and potentials of CacheInf?

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

Inference Time.

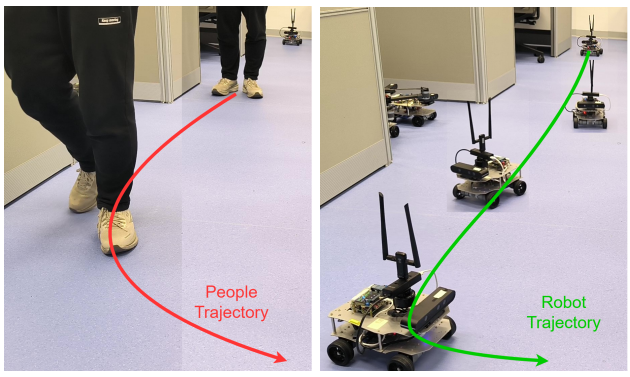
Energy Consumption.

Model(number of parameters)	Local computation time/s	System	Transmission time/s		Inference time/s		Percentage(%)	
			indoors	outdoors	indoors	outdoors	indoors	outdoors
kapao(77M)	1.01(±0.03)	DSCCS	0.21(±0.1)	0.24(±0.12)	0.36(±0.2)	0.40(±0.17)	58.33	60.21
		DSCCS-C	0.18(±0.14)	0.22(±0.12)	0.33(±0.25)	0.37(±0.18)	66.67	67.57
		Hybrid-Parallel	0.24(±0.15)	0.28(±0.13)	0.31(±0.14)	0.34(±0.12)	77.42	82.35
		CacheInf	0.16(±0.13)	0.21(±0.18)	0.20(±0.16)	0.24(±0.20)	80.09	87.56
agrnav(0.84M)	0.60(±0.04)	DSCCS	0.10(±0.05)	0.15(±0.05)	0.41(±0.11)	0.47(±0.12)	24.39	31.91
		DSCCS-C	0.13(±0.07)	0.16(±0.06)	0.38(±0.10)	0.43(±0.13)	34.21	37.21
		Hybrid-Parallel	0.24(±0.08)	0.26(±0.07)	0.30(±0.09)	0.33(±0.07)	78.65	79.47
		CacheInf	0.18(±0.08)	0.20(±0.08)	0.21(±0.16)	0.25(±0.18)	86.71	80.01

Table 2. Average transmission time, inference time, percentage that transmission time accounts for of the total inference time and their standard deviation ($\pm n$) of Kapao and AGRNav in different environments with different systems. "Local computation" refers to inference the entire model locally on the robot.

Model(number of parameters)	System	Power consumption(W)		Energy consumption(J) per inference	
		indoors	outdoors	indoors	outdoors
kapao(77M)	Local	10.61(±0.49)	10.61(±0.49)	9.79(±0.03)	9.79(±0.03)
	DSCCS	6.38(±2.21)	6.63(±2.38)	2.30(±0.55)	2.65(±0.55)
	DSCCS-C	6.30(±2.15)	6.53(±2.12)	2.08(±0.50)	2.42(±0.53)
	HP	7.05(±1.63)	6.94(±0.98)	2.19(±0.62)	2.35(±0.42)
	CacheInf	7.53(±1.62)	7.30(±0.96)	1.51(±0.60)	2.75(±0.41)
agrnav(0.84M)	Local	8.11(±0.25)	8.11(±0.25)	4.86(±0.01)	4.86(±0.01)
	DSCCS	6.21(±1.50)	7.29(±1.55)	2.55(±0.19)	3.43(±0.18)
	DSCCS-C	6.17(±1.56)	7.00(±1.43)	2.34(±0.20)	3.01(±0.20)
	HP	7.52(±0.51)	8.04(±0.45)	2.26(±0.15)	2.63(±0.15)
	CacheInf	7.83(±0.57)	8.23(±0.56)	1.64(±0.17)	2.06(±0.16)

Table 3. The power consumption against time (Watt) and energy consumption per inference (Joule) with standard deviation ($\pm n$) of Kapao and AGRNav different environments with different systems. "Local" represents "Local computation".



(a) Targeted people **(b)** Robot moving trajectory

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 [9].

References

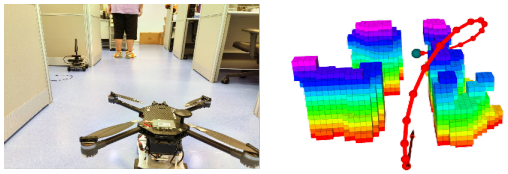
[1] Toni Adame, Marc Carrascosa-Zamacois, and Boris Bellalta. Time-sensitive networking in ieee 802.11 be: On the way to low-latency wifi 7. *Sensors*, 21(15):4954, 2021.

[2] 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 (VSWS09) in conjuction with ICCV*, 2009.

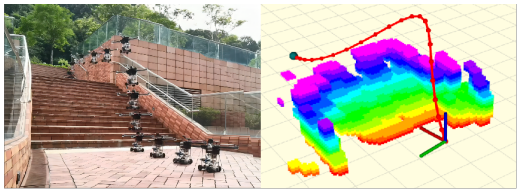
[3] Ming Ding, Peng Wang, David López-Pérez, Guoqiang Mao, and Zihuai Lin. Performance impact of los and nlos transmissions in dense cellular networks. *IEEE Transactions on Wireless Communications*, 15(3):2365–2380, 2015.

Model(number of parameters)	Local computaion time/ms	System	Transmission time/ms indoors	Transmission time/ms outdoors	Inference time/ms indoors	Inference time/ms outdoors	Percentage(%) indoors	Percentage(%) outdoors
DenseNet121(7M)	74.5(±18.7)	DSCCS	16.2(±40.9)	20.8(±51.9)	81.4(±27.2)	86.6(±27.7)	19.95	24.07
		DSCCS-C	20.4(±43.5)	25.8(±56.9)	85.5(±27.9)	89.6(±29.3)	23.86	28.80
		HP	53.4(±34.5)	52.9(±23.9)	74.5(±85.7)	55.1(±15.6)	71.70	96.05
		CacheInf	56.3(±37.5)	57.5(±43.5)	76.3(±90.6)	78.1(±33.6)	73.79	73.62
RegNet(54M)	175.0(±23.6)	DSCCS	47.6(±47.8)	60.5(±54.0)	77.8(±39.3)	86.2(±37.9)	61.22	70.22
		DSCCS-C	50.7(±49.8)	62.5(±53.6)	70.8(±33.3)	79.5(±39.2)	71.61	78.61
		HP	49.6(±21.7)	59.9(±23.4)	55.0(±24.8)	64.2(±25.2)	90.18	93.34
		CacheInf	44.2(±27.7)	48.5(±25.3)	45.3(±35.0)	49.2(±37.2)	97.57	98.58
ConvNeXt(88M)	160.2(±21.0)	DSCCS	46.9(±43.1)	56.7(±52.1)	72.4(±35.7)	84.7(±36.3)	64.78	66.95
		DSCCS-C	48.0(±45.0)	53.2(±50.1)	56.8(±28.1)	70.8(±39.0)	84.51	75.14
		HP	50.4(±32.2)	61.9(±34.8)	53.9(±26.2)	65.7(±27.7)	93.51	94.23
		CacheInf	40.7(±40.0)	50.7(±40.3)	46.7(±35.4)	56.8(±45.0)	87.15	89.26
VGG19(143M)	118.0(±18.9)	DSCCS	38.9(±47.1)	41.6(±53.8)	65.2(±28.1)	75.5(±27.1)	59.75	55.09
		DSCCS-C	42.7(±30.2)	52.0(±50.3)	53.2(±33.0)	60.3(±30.9)	80.26	86.24
		HP	44.8(±20.9)	51.5(±15.0)	47.6(±18.1)	53.6(±14.7)	94.15	96.07
		CacheInf	37.8(±31.2)	43.5(±13.2)	41.1(±20.3)	46.6(±12.8)	94.26	93.34
ConvNeXt(197M)	316.7(±31.0)	DSCCS	56.0(±36.1)	67.0(±37.6)	79.2(±35.9)	90.6(±35.4)	70.72	73.98
		DSCCS-C	56.0(±39.0)	63.0(±30.2)	64.7(±40.2)	68.6(±35.0)	86.55	91.84
		HP	56.4(±34.7)	66.5(±33.7)	59.7(±26.6)	68.0(±26.6)	94.43	97.88
		CacheInf	40.4(±37.8)	46.9(±40.0)	44.7(±33.3)	49.0(±30.8)	90.38	95.71

Table 4. Average transmission time, inference time, percentage that transmission time accounts for of the total inference time and their standard deviation ($\pm n$) of common AI models in different environments with different systems.



(a) Indoors



(b) Outdoors

Figure 4. By predicting occlusions in advance, AGRNav [22] gains an accurate perception of the environment and avoids collisions, resulting in efficient and energy-saving paths.

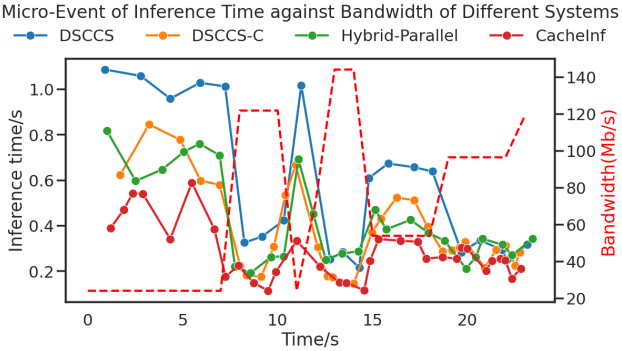


Figure 5. Micro-events about inference latency of different systems at different wireless network bandwidth.

[4] 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.

[5] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks, 2018.

[6] 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.

[7] Ruofeng Liu and Nakjung Choi. A first look at wi-fi 6 in action: Throughput, latency, energy efficiency, and security. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1):1–25, 2023.

[8] Antoni Masiukiewicz. Throughput comparison between the new hew 802.11 ax standard and 802.11 n/ac standards in selected distance windows. *International Journal of Electronics and Telecommunications*, 65(1):79–84, 2019.

Model(number of parameters)	System	Power consumption(W) indoors	Power consumption(W) outdoors	Energy consumption(J) per inference indoors	Energy consumption(J) per inference outdoors
DenseNet121(7M)	Local	8.2(±0.27)	8.2(±0.27)	0.46(±0.04)	0.46(±0.04)
	DSCCS	6.91(±0.45)	6.86(±0.46)	0.56(±0.04)	0.59(±0.04)
	DSCCS-C	7.01(±0.43)	6.96(±0.43)	0.60(±0.07)	0.52(±0.06)
	HP	5.36(±0.79)	5.79(±0.24)	0.4(±0.06)	0.32(±0.01)
	CacheInf	6.01(±0.92)	6.31(±0.56)	0.46(±0.12)	0.49(±0.01)
RegNet(54M)	Local	9.0(±0.3)	9.0(±0.3)	1.37(±0.02)	1.37(±0.02)
	DSCCS	5.84(±1.79)	5.36(±1.34)	0.45(±0.14)	0.46(±0.12)
	DSCCS-C	6.04(±1.88)	5.96(±1.45)	0.43(±0.16)	0.47(±0.19)
	HP	5.24(±1.43)	5.28(±1.52)	0.29(±0.08)	0.34(±0.1)
	CacheInf	5.20(±1.51)	5.43(±1.77)	0.24(±0.08)	0.327(±0.09)
ConvNeXt(88M)	Local	9.7(±0.34)	9.7(±0.34)	1.34(±0.02)	1.34(±0.02)
	DSCCS	6.01(±0.27)	5.71(±1.56)	0.43(±0.05)	0.48(±0.13)
	DSCCS-C	6.20(±0.33)	5.91(±0.21)	0.35(±0.17)	0.42(±0.25)
	HP	6.68(±1.23)	6.68(±1.21)	0.36(±0.07)	0.44(±0.08)
	CacheInf	6.70(±0.55)	6.63(±0.26)	0.31(±0.07)	0.38(±0.08)
VGG19(143M)	Local	9.78(±0.34)	9.78(±0.34)	0.95(±0.02)	0.95(±0.02)
	DSCCS	6.58(±2.14)	6.93(±2.35)	0.43(±0.14)	0.52(±0.18)
	DSCCS-C	6.82(±2.10)	7.23(±2.45)	0.36(±0.18)	0.43(±0.30)
	HP	6.51(±1.74)	7.32(±1.52)	0.31(±0.08)	0.39(±0.08)
	CacheInf	6.70(±1.88)	7.22(±1.36)	0.27(±0.10)	0.34(±0.09)
ConvNeXt(197M)	Local	10.72(±0.38)	10.72(±0.38)	3.12(±0.03)	3.12(±0.03)
	DSCCS	5.06(±0.31)	5.02(±0.37)	0.4(±0.02)	0.45(±0.03)
	DSCCS-C	4.86(±0.44)	4.99(±0.39)	0.31(±0.05)	0.34(±0.09)
	HP	4.57(±0.23)	4.54(±0.25)	0.27(±0.01)	0.31(±0.02)
	CacheInf	5.26(±0.40)	5.39(±0.27)	0.24(±0.05)	0.26(±0.04)

Table 5. The power consumption against time (Watt) and energy consumption per inference (Joule) with standard deviation ($\pm n$) of common AI models in different environments with different systems. "Local" represents "Local computation".

Model	Statistics	Difference Filter Parameter (n)		
		50	70	130

Table 6. How different difference filter parameter (n) for identifying reusable cache affects the inference latency of CacheInf and the accuracy of visual models.

Model	Statistics	Sampling rate		
		1	2	4

Table 7. How the sampling rate of video frames influence the performance of CacheInf.

[9] 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.

[10] NVIDIA. Using Shared Memory in CUDA C/C++. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/, January 2013.

[11] 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.

[12] NVIDIA. The world's smallest ai supercomputer. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/, 2024.

[13] 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.

[14] Yuanteng Pei, Matt W Mutka, and Ning Xi. Connectivity and bandwidth-aware real-time exploration in mobile robot networks. *Wireless Communications and Mobile Computing*, 13(9):847–863, 2013.

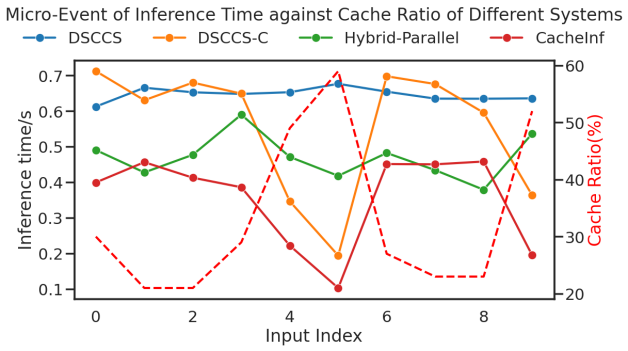


Figure 6. Micro-events about inference latency of different systems at different cached ratio with fixed wireless network bandwidth.

[15] 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.

[16] PyTorch. Distributed communication package - torch.distributed — PyTorch 2.3 documentation. https://pytorch.org/docs/stable/distributed.html/, 2024.

[17] 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.

[18] Yi Ren, Chih-Wei Tung, Jyh-Cheng Chen, and Frank Y Li. Proportional and preemption-enabled traffic offloading for ip flow mobility: Algorithms and performance evaluation. *IEEE Transactions on Vehicular Technology*, 67(12):12095–12108, 2018.

[19] Nurul I Sarkar and Osman Mussa. The effect of people movement on wi-fi link throughput in indoor propagation environments. In *IEEE 2013 Tencon-Spring*, pages 562–566. IEEE, 2013.

[20] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2015.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.