Towards Low-Latency GPU-Aware Pub/Sub Communication for Real-Time Edge Computing

Hao-En Kuan, Yung-Hsiang Yang, Zen-Mou Jiang, Chi-Sheng Shih and Shih-Hao Hung
Dept. of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
{r12922060, r12922a09, r12944042, cshih, hungsh}@csie.ntu.edu.tw

***Abstract*—Real-time Edge AI applications often require efficient GPU-based data processing and communication. Since the applications are typically highly modularized, publish–subscribe (pub/sub) pattern is widely used to deliver data among components. However, existing pub/sub middleware introduces significant latency due to redundant memory copies between GPU and host memory. To address this, we propose GPU-Aware Pub/Sub communication (GAPS), a universal solution that integrates shared CUDA memory with existing pub/sub middleware, such as Zenoh-pico and Iceoryx. GAPS minimizes data transfer latency by enabling GPU memory sharing between publishers and subscribers, eliminating unnecessary memory copies. In our work, we propose an independent shared CUDA memory manager that creates a shared CUDA memory pool for each topic during a topic's initialization. For fine-grained allocation from the pool, we modify Two-Level Segregated Fit (TLSF), a real-time dynamic memory allocator, making it process-safe and capable of managing GPU memory. Additionally, we develop PyGAPS, an extension that accelerates publications of PyTorch tensors, eliminating serialization overhead in AI-driven applications. Our evaluation demonstrates that GAPS significantly reduces end-to-end latency and improves throughput of simplified computer vision pipelines—by up to 1.5× in the segmentation task and 3.8× in the classification task—making it a robust solution for real-time Edge AI.**

***Index Terms*—Pub/Sub Middleware, GPU, CUDA, Shared Memory, Real-Time Dynamic Memory Allocator**

I. Introduction

Artificial intelligence (AI) has been a hot topic for many years. Numerous AI technologies, such as machine learning, reinforcement learning and deep learning, have been proposed to solve problems like detecting objects, recognizing anomalies in sensor data, or generating natural chat responds. To reduce the responding latencies of AI models and enhance data privacy of users, there is a trend of deploying AI models in edge computing environments, i.e., Edge AI. There have been many researches [1], [2], [3], [4] discussing the benefits of combining edge computing or embedded systems with AI. By processing data closer to the source, Edge AI ensures faster response time, making it suitable for applications requiring real-time decision-making. Additionally, sensitive data can be processed locally, mitigating privacy concerns. For example, they are applied in the area of embodied agent, autonomous vehicle, or smart manufacturing. Deployed applications are usually modularized into multiple components for easier development, higher resource utilization and better fault isolation. As for the communication among these modules, publish–subscribe pattern, namely pub/sub pattern, is often used because it abstracts the networking details and provides semantics for delivering messages to whom needs them without having to know the exact receivers. For instance, there is RobotCore [5], an open architecture integrating hardware such as FPGA and GPU into ROS 2 contributed by the ROS 2 Hardware Acceleration Working Group (HAWG), a community dedicated to bring hardware acceleration to ROS 2.

We focus on accelerating data publications between components of GPU-accelerated applications involving pub/sub communication on edge computing devices. On such platform, there is usually only one GPU, e.g., NVIDIA's integrated GPU on Jetson series embedded system. In addition, as these applications are accelerated with NVIDIA's GPUs, CUDA is often utilized for computation. When transferring data on CUDA memory between two processes, a naive approach is to copy the data twice between GPU and the host, i.e., copying the data from CUDA memory to the host memory, sending the data to the other process via inter-process communication (IPC) or network, and copying the data from the host memory back to the CUDA memory. Nevertheless, it is evident that this approach wastes much time on copying data. A better approach is to leave the data at rest on the CUDA memory and tell the other process the location of the data. For example, existing works [6], [7] have attempted to combine the benefits of shared CUDA memory and pub/sub communication patterns in the area of robotics and autonomous navigation systems, respectively. However, the authors of [6] only made it to support fixed-sized messages and communication between multiple threads, while the pub-centric design in [7] may break the decoupled characteristics of the pub/sub pattern.

We overcame the problems in previous works and proposed GPU-Aware Pub/Sub communication (GAPS), an approach to incorporating existing pub/sub middleware with shared CUDA memory to reduce the latency of publishing data on GPU memory. In addition, our implementations have been open-sourced at https://github.com/Alan-Kuan/GAPS. The contribution of our work can be summarized as follows:

2325-1301/25/$31.00 ©2025 IEEE
DOI 10.1109/RTCSA66114.2025.00018

1) We proposed a universal method applicable to existing pub/sub middleware. For instance, we successfully applied our work on two modern pub/sub middleware implementations, including Zenoh-pico [8] and Iceoryx [9].
2) To efficiently allocate dynamically-sized memory space from the shared CUDA memory, we crafted Header-Detached Process-Safe TLSF, which is derived from the real-time dynamic memory allocator, Two-Level Segregated Fit (TLSF) [10].
3) Our work is made to support not only x86 machines but also ARM-based embedded devices like NVIDIA Jetson AGX Orin, with the potential to support any edge computers equipped with an NVIDIA GPU.
4) We even developed PyGAPS, supporting publications of PyTorch tensors, so that the communication acceleration can be easily brought to the area of AIs. Our work has revealed its ability to improve throughput in an experiment of a computer vision application.

II. Background and Related Work

A. Publish–Subscribe Pattern

Publish–subscribe pattern is a common messaging model where publishers send messages to a *topic* or *channel*, whose subscribers receive the messages. The characteristics of the pub/sub messaging model include decoupling, asynchronous communication and many-to-many relationships. For example, Eclipse Zenoh [8], Eclipse Iceoryx [9], Apache Kafka, Message Queuing Telemetry Transport (MQTT) and Data Distribution Service (DDS) are common pub/sub middleware or standards used in IoT and robotics areas. In our experiments, we chose Zenoh and Iceoryx as the foundation of our implementations.

Zenoh is a fully decentralized pub/sub middleware, providing efficient pub/sub primitives in a scalable network environment. Based on different network scales, Zenoh offers two distinct communication modes: *peer-to-peer* and *routed-and-brokered*. Zenoh nodes can establish direct connections with peers in peer-to-peer communication mode and can further establish connections with other nodes over the Internet via Zenoh routers. Studies [11], [12] have shown that Zenoh outperforms well-known pub/sub standards, such as Kafka and MQTT, in terms of latency and throughput. Moreover, Zenoh-pico, an implementation of Zenoh for microcontrollers and embedded devices, achieved lower latency than Cyclone DDS [13] on single-machine and multi-machine scenarios [11].

Iceoryx is an IPC pub/sub middleware leveraging POSIX shared memory. It allows a true zero-copy, end-to-end approach from publishers to subscribers, maintaining constant latency independent of payload size, which ensures predictable and reliable data transmission. In addition, Iceoryx 2 [14], evolving from Iceoryx, offers even lower data transmission latency while enhancing extensibility. In comparison with Iceoryx, it eliminates the centralized daemon connecting publishers and subscribers in favor of a more decentralized architecture. Unfortunately, as it has not provided stable C++ bindings so far, we chose not to implement our work on Iceoryx 2, but it should also benefit from the proposed method.

B. Shared CUDA Memory

CUDA is a parallel programming model developed by NVIDIA. It enables developers to leverage NVIDIA's GPUs to accelerate programs requiring heavy computing powers, such as scientific simulation, video processing and deep learning. CUDA library mainly provides two APIs with different levels of controllability, which are CUDA Runtime API and CUDA Driver API. In most cases, the former is enough for developers to accelerate a program. However, to get fine-grained control of NVIDIA's GPU, one should use the latter. For example, *virtual memory management* of CUDA memory is one of the abilities provided by the CUDA Driver API. There is a set of APIs from this category that makes it possible to create a large segment of CUDA memory shareable among processes. To realize the sharing, there are mainly three steps:

1) Export the *handle* of a newly allocated CUDA memory segment into a shareable handle, which is actually a *file descriptor* on Linux.
2) Duplicate the shareable handle to another process.
3) Import the shareable handle into a generic handle and map the same memory segment in another process.

Step 1 and 3 are done by the virtual memory management APIs. For Step 2, shareable handle can be duplicated via the *SCM_RIGHTS* message of UNIX Domain Socket. Actually, there are two other ways to duplicate file descriptors on Linux, i.e., opening pseudo files in the directory *fd* of target process in *procfs* or utilizing the *pidfd_getfd* system call. However, according to the authors of [6], none of them made it to share the CUDA memory. Besides, a process is not permitted to duplicate file descriptors of an arbitrary process with these two methods due to *Yama*, a Linux Security Module. Thus, we utilize the UNIX Domain Socket to help us share the allocated CUDA memory among processes.

In addition to virtual memory management APIs, there is also a set of CUDA IPC APIs in the category of *memory management* APIs. Similarly, they can be utilized to share CUDA memory between multiple processes but provide more flexible functionality, i.e., exporting any memory buffers allocated with the *cudaMalloc* API from CUDA Runtime API into a shareable handle. However, as of the latest version of CUDA, the CUDA IPC APIs are still not supported on Linux for Tegra (L4T) and embedded Tegra devices [15]. Therefore, we can only rely on the set of APIs from the category of virtual memory management.

C. Real-time Dynamic Memory Allocator

Mission-critical applications require real-time response time to be guaranteed for software to work correctly. A dynamic memory allocator (DynMA) that can guarantee a worst-case execution time of $\mathcal{O}(1)$ is highly desirable. The *buddy system* [16] fully meets this need and is widely adopted as a real-time allocator. However, it suffers from significant internal fragmentation [17], [18]. While the issue can be mitigated through optimizations such as the fine-grained *slab allocator* [19] used in Linux kernel, alternative real-time DynMAs like TLSF [10] and RT-Mimalloc [20] offer more effective solutions to the problem.

TLSF is nearly a best-fit algorithm, enhanced with an advanced segregated free list structure. It provides deterministic $\mathcal{O}(1)$ time for both allocation and deallocation, while striving to minimize internal fragmentation. However, it is not process-safe and lacks support for managing GPU memory. To address this limitation, we implemented a modified version of TLSF to manage a pool of shared CUDA memory in our work.

RT-Mimalloc is a more recent real-time DynMA. Its authors conducted a comprehensive analysis of existing real-time and general-purpose allocators. They observed that although TLSF offers minimal worst-case latency, it tends to exhibit higher average-case latency compared to general-purpose allocators. RT-Mimalloc leverages the low average-case latency design of Microsoft's Mimalloc [21], while also bounding its worst-case latency, thus becoming a real-time DynMA outperforms TLSF in terms of average-case latency. Unfortunately, RT-Mimalloc is not open-sourced. Besides, considering the simplicity of TLSF's design compared to RT-Mimalloc, we ultimately chose TLSF for use in our framework.

D. Related Work

We discovered three existing works on bridging the pub/-sub communication model with shared memory, including POSIX shared memory and shared CUDA memory, in different scenarios. First, a research [22] proposed an inter-container communication interface with shared memory channels for speeding up container communication in microservices. Instead of traditional TCP/UDP communication, they replaced local container communication with shared memory IPC and remote container communication with Remote Direct Memory Access (RDMA). Besides, their work can be integrated into Kubernetes, a container orchestration framework. However, this research focused only on inter-container communication, and their work does not support shared CUDA memory.

Second, another research [7] has identified the absence of effective GPU end-to-end data transfer mechanisms in the pub/sub architecture. To solve the issue, the authors proposed a GPU message communication framework in a pub/sub environment, primarily via CUDA IPC APIs. For a single-GPU scenario, shared CUDA memory is used, while for the multi-GPU scenario, the optimal communication method is chosen according to offline profiling. However, their *pub-centric* approach breaks the decoupled relationship between publishers and subscribers. There should be issues, including that subscribers cannot be established and all-set before a publisher exists, and that newly joined publishers will create new CUDA memory segments and request all the existing subscribers to map the segments, interrupting their current task. In addition, since address offsets are placed on CUDA memory in their design, their offset conversion approach utilizes a one-threaded CUDA kernel to translate the offsets into real CUDA memory addresses recursively for each process. However, such trivial conversion tasks are not suitable for GPUs but CPUs if offset values can be placed on host memory.

Last, a research [6] proposed a heterogeneity-aware zero-copy pub/sub framework, which was implemented as a ROS 2 Middleware (RMW). In their work, host memory and memory of different accelerators, including GPU, are seen as different memory domains. For communication within the same memory domain, data copying is avoided with shared memory; for communication across different memory domains, data copying is done only once by one of the subscribers, and other subscribers on the same memory domain can read the copy afterward. Nonetheless, the allocator they implemented to manage the memory pool is merely a *ring buffer*, which makes it impossible to allocate dynamically-sized memory spaces for variable-sized messages. Besides, they only made it to share CUDA memory across multiple threads rather than processes.

III. Methodology

We want to compensate what is lacked in the existing work of bridging pub/sub middleware with shared CUDA memory including the following three things:

1) The component of allocating shared GPU memory from CUDA and sharing it to other processes should be decoupled from publishers.
2) The sharing of CUDA memory should be realized not only across threads but also processes.
3) The allocator managing the shared CUDA memory should be able to allocate dynamically-sized memory blocks.

To realize the first two points, we introduce a shared CUDA memory manager to serve as an agent of allocating shared CUDA memory pool for each topic. Both publishers and subscribers can send a request to ask for a shared CUDA memory from the manager via UNIX Domain Socket. Consequently, subscribers do not depend on publishers to obtain the shared CUDA memory. In addition, both publishers and subscribers are able to join a topic at any time. As for the finer allocator, we chose the TLSF allocator, which provides real-time dynamically-sized allocation. To make it process-safe and efficient on allocating GPU memory, we made some modifications, which will be elaborated in Section IV-A. The overview of our work is shown in Fig. 1. There are three primary components, which are the shared metadata store, the shared CUDA memory manager and the publisher/subscriber node.

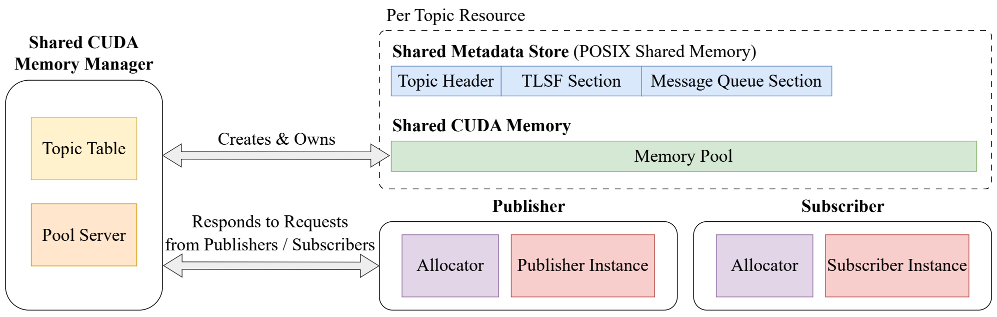


Fig. 1. Overview of our work.

A. *Shared Metadata Store*

There is a need for an efficient way to synchronize publishers and subscribers of the same topic. For example, knowledge including topic name, memory pool size, allocator states and payload descriptions should be shared between the publishers and subscribers. Therefore, POSIX shared memory was chosen since it allows multiple processes to exchange information efficiently just as they access their memory. As the allocated POSIX shared memory is responsible for storing metadata, we named it *shared metadata store*. Each topic is associated with one shared metadata store, and the topic name is utilized as the name of the shared memory so that the publishers and subscribers can easily find and map it. Furthermore, a shared metadata store is composed of three major parts: the topic header, TLSF section and message queue section.

1) Topic Header: The topic header includes topic name, padded pool size, reference count and subscriber count. First, the topic name is preserved for later uses, such as when the shared metadata store is considered useless and should be unlinked. Second, the padded pool size is the actual size of the shared CUDA memory pool. To be more specific, it is the user-specified pool size padded to a multiple of the granularity of CUDA memory. Last, reference count is the number of publishers and subscribers joining the topic, while subscriber count is only the number of subscribers joining the topic. They are increased when relevant roles join and decreased when relevant roles leave.

2) TLSF Section: The TLSF section is reserved for storing metadata and the internal states of the TLSF allocator. The metadata include an initialization flag, the aligned pool size and the maximum number of blocks in the pool, while the internal states include first level bitmap, second level bitmap, lists of free blocks and detached block headers. The detached block headers are part of our customized design, which is further explained in Section IV-A. In addition, to avoid race conditions during state updates, a binary semaphore is included in this section to serve as a *mutex lock*.

3) Message Queue Section: The message queue section serves as a simple lookup table for publishers and subscribers to update the status and location of the message's payload. To be more specific, the section is composed of a header and a ring buffer. First, the header stores the *capacity* of the ring buffer and an always-increasing variable, *next*, whose value modulo *capacity* indicates the index of the next available entry of the buffer. Noteworthy, since increment with modulo cannot be done atomically, when getting the next index, *next* is only atomically fetched and incremented by 1, and the fetched old value modulo the capacity is used. Therefore, the capacity of the ring buffer should be a power of 2, or the calculated index will be wrong when *next* overflows. Second, the ring buffer stores message queue entries. An index to an entry of the ring buffer is seen as a key, i.e., message ID, to a message queue entry. Each entry is associated with a published message, and describes the location of the message's payload, the size of the payload and the number of subscribers that have taken the payload. The first two metadata enable subscribers to find and utilize the payload, while the third value is used to determine if the space of the payload should be recycled. In Section IV-C, we will explain how publishers and subscribers update these values.

B. *Shared CUDA Memory Manager*

We noticed that a mechanism that offers *named* shared CUDA memory does not exist. Unlike POSIX shared memory, which can be opened with a specific file name and mapped to the virtual address space of a process, shared CUDA memory can only be created *anonymously*. As mentioned in Section II-B, a shared CUDA memory segment between two processes is achieved by allocating a dedicated CUDA memory with CUDA Driver API and duplicating its shareable handle via UNIX Domain Socket. Thus, we proposed a shared memory manager that accepts requests from publishers/subscribers and manages the CUDA memory pools for each topic.

In our design, the manager maintains a hash table (namely, topic table) storing the mapping from a topic name to the generic handle of its corresponding pool. The updates of the table only happen during the initialization phase of publishers/subscribers, termination phase of publishers/subscribers and the termination of the manager.

First, during the initialization phase of a publisher/subscriber, it sends a request to the manager for the shareable handle of a memory pool. The request consists of the topic name and the requested size. If the topic name is found in the topic table, the manager directly exports the handle of the corresponding pool into a shareable handle and returns it to the client. Otherwise, the manager will create a new CUDA memory pool with the requested size and update the table before replying. Second, before a publisher or subscriber terminates, the one that decreases the reference count in the shared metadata store to zero is responsible for sending a request to the manager for releasing the pool. Finally, before the manager terminates, it releases all the pools in the table.

C. Publisher and Subscriber Node

To save some implementation efforts and easily show the effects of publishing messages over shared CUDA memory, we integrated our work on existing pub/sub middleware, including Zenoh-pico and Iceoryx. In our implementation, both publishers and subscribers are composite classes of a memory allocator and a relevant pub/sub instance.

The memory allocator is responsible for managing the pool received from the shared CUDA memory manager. As part of our design, we implemented the TLSF allocator for two reasons. First, it is a low-latency real-time allocator that provides bounded allocation and deallocation time. Second, it is able to allocate dynamically sized spaces. Therefore, the allocator is suitable for procedures that require real-time guarantees while providing flexible-sized buffers for messages. Note that publishers allocate or release memory spaces via the allocator, while subscribers only release memory spaces. There is an exported allocation interface on the publisher allowing the users to allocate a shared CUDA memory space beforehand and put computing results directly on it. When the data is ready to be published, the address of the buffer is passed to the publisher through its publishing interface. The publisher will convert the address into a message ID (an index to a message queue entry) and publish it via the publishing instance. To reclaim the shared CUDA memory, the allocated buffer will be released by the last subscriber who takes the data. In case there is no subscriber while a message gets published, the space of the payload will still be released when the same message queue entry is obtained by some publisher next time. The details of payload lifecycle will be mentioned in Section IV-C2.

As for the pub/sub instance, it is the core component that makes pub/sub communication possible. While publishing a message, the actual payload is left on the shared CUDA memory. The publishing instance publishes only the message ID instead of the payload of the message through the underlying middleware. By publishing only message IDs, the copies from GPU memory to host memory and vice versa are avoided. Besides, it saves much time because the message ID only takes 8 bytes on a platform with 64-bit processors. After the subscribing instance receives the message ID, the subscriber can easily find the relevant payload on the shared CUDA memory by looking up the message queue section of the shared metadata store.

IV. Implementation

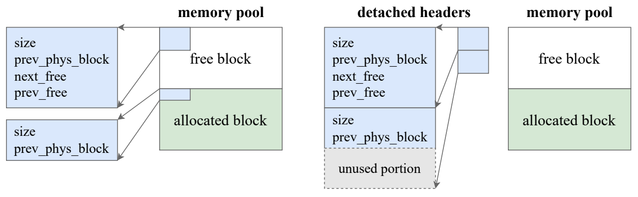
In this section, we elaborate on the details of each mechanism in our work. First, we will demonstrate the implementation details of Header-Detached Process-Safe TLSF, a modified TLSF to meet our requirements, in Section IV-A. Second, the construction steps of a publisher or a subscriber are shown in Section IV-B. Third, the process of publishing or handling a message in our framework is described in Section IV-C. Last, we will show how we even bind our work to Python in Section IV-D.

A. Header-Detached Process-Safe TLSF

Original TLSF allocator is not capable for allocating GPU memory and is not safe for being used among multiple processes. The reason is that it maintains the allocation states on the memory pool to be allocated, which is not possible or not performant to be done when the memory pool is GPU memory. In addition, there is no synchronization mechanism mentioned in the original design to make it process-safe. However, the algorithm still provides good allocation performance and its logic is easy to be implemented. Thus, we made two modifications to the TLSF allocation algorithm to fit it into our framework.

1) Detached Block Headers: In the original TLSF design, metadata of a block in a memory pool is written at the beginning of that block, which is called the block header. The header includes information such as block size, physically-previous block, next free block and previous free block. Instead of writing the information at the beginning of memory blocks, we maintain their headers in an external memory space, i.e. the shared metadata store. It is because allocated blocks from our allocator are GPU memories, and CPUs can not directly access them. Besides, utilizing a CUDA kernel to update the headers can bring a large latency overhead. By maintaining the headers on the shared metadata store as an array, they can be easily accessed by CPUs in a multiprocess scenario. Apart from that, the allocation algorithm becomes more robust as the header content will not be modified by *buffer overflow attacks* or accidental overflows. As illustrated in Fig. 2, headers of either allocated blocks or free blocks in our version are placed outside the memory pool. The detached headers, which are part of the allocator states mentioned in Section III-A2, are actually placed on the TLSF section of the shared metadata store.

As the shared metadata store is mapped at different starting addresses in different process, pointers such as *physically previous block*, *next free block* and *previous free block* in the block header should be stored in the form of relative address, i.e., indices to the corresponding detached header blocks. As for the downside of this method, it introduces memory overhead, which is proportional to the



(a) Original TLSF. (b) Header-detached TLSF.

Fig. 2. Comparison of the original TLSF and our header-detached TLSF.

TABLE I
AVERAGE LATENCY OF *TlsfMalloc* AND *TlsfFree* UNDER DIFFERENT DEGREES OF CONTENTION WITH DIFFERENT LOCK IMPLEMENTATIONS ON JETSON AGX ORIN.

#Procs	***TlsfMalloc***		***TlsfFree***	
	Ticket Lock	Semaphore	Ticket Lock	Semaphore
1	3.1	2.9	0.3	0.3
2	15.0	13.0	8.2	7.5
4	127.7	99.9	31.2	34.9
8	513.9	330.2	105.4	128.0

Unit: µs

number of memory blocks. Nonetheless, the method helps to make allocation from the shared GPU memory pool possible and efficient.

2) Critical Section Protection: Other allocation states of TLSF (e.g., first level and second level bitmaps) are also placed on the TLSF section of the shared metadata store. Consequently, we make internal operations of TLSF process-safe by protecting critical sections with a *binary semaphore*. Apart from semaphore, we also considered the *ticket lock* [23], which is a kind of *spin lock* that avoids starvation. A spin lock keeps a process busy-waiting for its turn to enter the critical section. It is a good choice if the critical section is short enough since it brings a shorter waiting time than a sleep-waiting lock. In a preliminary test on Jetson AGX Orin (see Table I), where all processes first call *TlsfMalloc* simultaneously for 100 times and then apply *TlsfFree* on all allocated spaces later, we found that when the contention is high, the implementation with a semaphore is faster while calling *TlsfMalloc* and slower while calling *TlsfFree*. However, simultaneously releasing memory space is rare in our use case. As a result, we preferred a binary semaphore in our final implementation.

B. Publisher/Subscriber Construction

In our work, a topic is associated with a shared metadata store and a shared CUDA memory pool. Either a publisher or a subscriber of a topic can play the role of initializing them. Upon construction, there are three major steps a publisher or subscriber does, including initializing the shared metadata store, constructing the allocator for the shared CUDA memory pool, and constructing the pub/sub instance.

1) Shared Metadata Store Initialization: The initialization of a shared metadata store is done by the one who starts its construction first (either a publisher or a subscriber). As mentioned in Section III-A, the shared metadata store is composed of a topic header, a TLSF section and a message queue section. Therefore, we have to know their sizes to determine the required size for the shared metadata store. First, the size of the topic header, the TLSF section header and the message queue section header are known initially. Second, the size of the TLSF section excluding its header is calculated by (1). Obviously, it is determined by the *aligned pool size*, which is the largest multiple of minimum block size that does not exceed the *padded pool size*. The padded pool size is calculated from the user-specified pool size, which is padded to a multiple of the granularity of shared CUDA memory. Therefore, the value of aligned pool size over minimum block size is essentially the maximum possible number of blocks in the pool. Third, the size of message queue section excluding its header is calculated by multiplying a user-specified queue capacity with the size of an entry. Finally, the required size for the shared metadata store is the sum of these values.

$$\frac{\textit{aligned pool size}}{\textit{minimum block size}} \times \textit{block header size} \tag{1}$$

After the metadata store is created, the publisher/subscriber fills all the mentioned headers. Besides, it increases the reference count of the topic. For other publishers or subscribers that are not responsible for the initialization, they map the shared metadata store and the shared CUDA memory pool without creating a new one and increase the reference count.

2) Memory Allocator Construction: After mapping the memory segments, the publisher or subscriber will construct an allocator for itself. During the construction of the allocator, the allocator will ask the shared CUDA memory manager for a memory pool with the padded pool size via UNIX Domain Socket. After receiving the shareable handle from the manager, it will import and map the memory pool. A publisher sets the access permission to *read-write*, while a subscriber sets *read-only* for security's sake.

3) Pub/Sub Instance Construction: Finally, the publisher or subscriber constructs its corresponding instance, which is responsible for publishing or handling messages. We have adopted Zenoh-pico and Iceoryx as the underlying middleware for publishing message IDs. When using Zenoh-pico, its peer-to-peer mode is utilized, which is based on UDP multicast, and publisher's express mode was enabled to prevent buffering. On the other hand, when using Iceoryx, its event-driven listening subscriber was utilized with the subscriber queue capacity set to 256. Noteworthy, in Iceoryx, when multiple data-received events comes, subsequent events will be ignored while the event callback is still running. Thus, we created a loop in the callback that keeps handling messages until no more is found.

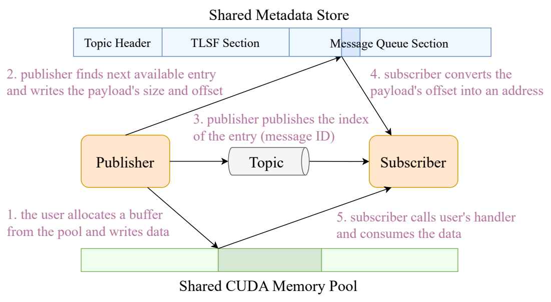

Shared Metadata Store
Topic Header
TLSF Section
Message Queue Section
2. publisher finds next available entry and writes the payload's size and offset
4. subscriber converts the payload's offset into an address
3. publisher publishes the index of the entry (message ID)
Publisher
Topic
Subscriber
1. the user allocates a buffer from the pool and writes data
5. subscriber calls user's handler and consumes the data
Shared CUDA Memory Pool

Fig. 3. The flow of publishing and handling.

C. Message Publishing and Handling

The data flow of our work can be mainly described in two folds. First, the payload of a message is placed on the CUDA memory shared among publishers and subscribers of the same topic. Second, the publisher publishes a notification (the message ID) to tell where the payload is actually placed. The whole flow of message publishing and handling is depicted in Fig. 3. At the publisher side, a memory allocation interface is exported to publishers so that users can allocate shared CUDA memory buffers via publishers. Therefore, it is possible for the users to directly store computation results in the buffer beforehand. When a publisher is about to publish a message, it mainly does the following two things. First, it finds the next message queue entry on the shared metadata store and updates it (from line 2 to line 12 in Listing 1). Then, it publishes the entry's index to all the subscribers of this topic (line 13 in Listing 1). On the other hand, at the subscriber side, the work of handling a message can be broken down into three parts. First, it finds the message queue entry on the shared metadata store with the received message index (from line 2 to line 3 in Listing 2). Second, it converts the *offset* in the entry to a valid CUDA memory address and calls the user's message handler (from line 5 to line 7 in Listing 2). Finally, it checks if it is the last subscriber using the data. The last one is responsible for freeing the space of the payload (from line 8 to line 12 in Listing 2).

```
 1  publish(payload_addr, size) {
 2      offset = payload_addr - pool_base_addr
 3      entry_idx = fetch_add(next, 1) % mq_capacity
 4      entry = entry_base_addr + entry_idx * entry_size
 5      // free an occupied entry
 6      if (entry->offset & 1 == 1) {
 7          tlsf_free(entry->offset)
 8      }
 9      // mark the entry as occupied
10      entry->offset = offset | 1
11      entry->size = size
12      entry->taken_number = 0
13      publish_via_middleware(entry_idx)
14  }
```

Listing 1. Procedure for publishing a message.

To mark if a message queue entry is empty or not, i.e., if the memory space that its offset points to is free or not, the rightmost bit of the offset value is utilized, as the size of an allocated block and the base address must be even. Additionally, because the newly created message queue entries are filled with zeros, we use a 1-bit as the mark of an occupied entry while a 0-bit as the mark of an empty entry. Otherwise, if a 0-bit is taken as the mark of an occupied entry, a new message queue entry may be mistakenly seen as occupied.

```
 1  handler(middleware_msg) {
 2      entry_idx = middleware_msg.data
 3      entry = entry_base_addr + entry_idx * entry_size
 4      // remove the mark
 5      offset = entry->offset ^ 1
 6      payload_addr = pool_base_addr + offset
 7      handler_from_user(payload_addr, entry->size)
 8      if (fetch_add(entry->taken_number, 1) ==
        subscriber_count - 1) {
 9          tlsf_free(offset)
10          // mark as an empty entry
11          entry->offset = offset
12      }
13  }
```

Listing 2. Procedure for handling a received message.

1) Address Conversion: Since each process has its own virtual address space for CUDA memory, a valid address in one process is not likely mapped to the same memory space in another process. Likewise, the base address of shared metadata store is different on different processes. Therefore, we have to convert the address into a universal form, i.e., an offset from the base address of the memory pool or shared metadata store. By sharing the offset of an allocated block from the shared CUDA memory pool, every process can recover it into a valid address by adding its own base address of the memory pool. After a subscriber converts the offsets into valid addresses, it simply calls the user-provided message handlers.

2) Payload Lifecycle: As the shared CUDA memory pool is a limited resource, measures to reclaim the allocated spaces are urgent. The spaces are released when the last subscriber finishes its handling or when a publisher retrieves an entry with unreleased payload (from line 6 to line 8 in Listing 1). The former recycles memory spaces as early as possible, while the latter recycles orphaned memory spaces. For instance, when a publisher publishes to a topic with no subscriber at the moment, the allocated space will not be released by any subscriber but some publisher in the future. Although the second measure can recycle memory spaces prone to memory leaks, if the publishing frequency is too high, a payload's space may be released too early as some subscribers are still reading the payload. As a result, the message queue size should be large enough, or the publishing frequency should be low enough. To ensure not reading corrupted data, a timestamp field can be added to each message queue entry. While publishing, a timestamp is filled to the relevant entry and its value is also sent along with the message ID. The subscriber consumes the data first before checking if the received timestamp matches the entry's one. If they do not match, the subscriber considers the data to be corrupted. Although there is a small likelihood of false positives, this method absolutely avoids corrupted data by sacrificing some memories.

D. Binding with Python

Our work is even extended to support transferring PyTorch tensors for use cases where multiple deep learning models run in different processes. We chose *nanobind* [24] to bind our library into a Python module, as it supports *DLPack* [25], a protocol of a stable in-memory cross-device data structure widely adopted by major deep learning frameworks including PyTorch. With the binding, the allocation interface exported on the publisher returns a PyTorch tensor instead. In addition, the *publish* function is modified to accept a tensor as an argument and fill a simplified DLPack structure describing it. Then, the notification it publishes becomes a message ID along with the DLPack structure. If describing a three-dimension tensor, the notification size only increases 32 bytes, barely impacting the communication latency. At the subscriber side, the *handler* function calls the user-provided handler with a new tensor constructed with the information from the DLPack structure in the notification. It is worth mentioning that while constructing a new tensor, the Python *Global Interpreter Lock* (GIL) should be acquired, or a runtime error may occur.

V. Evaluation

A. Environment Configuration

To evaluate our framework, we tested it on an x86 workstation and NVIDIA Jetson AGX Orin, an ARM-based embedded computing device with an integrated NVIDIA GPU. The latter features a physically unified memory between CPU and GPU, which makes copying between GPU and CPU faster. Nonetheless, we will show that our method still outperforms by saving the time on copying data. The details of our evaluation environment are shown in Table II.

TABLE II
The evaluation environment.

Attribute	Value
Platform 1	NVIDIA Jetson AGX Orin 64 GB
CPU	Arm Cortex-A78AE v8.2 64-bit CPU (12 cores)
GPU	NVIDIA Ampere architecture with 2048 NVIDIA CUDA cores and 64 Tensor Cores
RAM	64 GB (256-bit LPDDR5)
OS	Jetson Linux 36.4 (kernel: 5.15.148-tegra)
Platform 2	x86 Workstation
CPU	AMD Ryzen 9 5950X (16 cores / 32 threads)
GPU	NVIDIA GeForce RTX 4080 SUPER (10240 cores, 16 GB)
RAM	128 GB (Kingston KF3600C18D4/32GX DDR4 * 4)
OS	Ubuntu 22.04 (kernel: 6.5.0-25-generic)
Compiler	g++ 11.4.0
CUDA	12.6
Python	3.10.12
Zenoh-pico	1.1.0
Iceoryx	2.0.6

B. End-to-End Latency

In this experiment, we measured the end-to-end latency of publishing the data that were already placed on GPU memory from a publisher to a subscriber. In the configuration of GAPS, the data were assumed to already reside on shared CUDA memory. The first time point was set at the publisher side before it published the data, and the other time point was set as soon as the user-provided handler was called. For comparison, the conventional method copied GPU data to host memory, published the data through either Zenoh-pico or Iceoryx, and copied the data back to the CUDA memory on the subscriber side. Thus, this method involved at least two more copies. The first time point was set before the data were copied from CUDA memory to host memory on the publisher's side, and the other time point was set right after the data were copied from host memory to CUDA memory. Additionally, in a preliminary test comparing regular CUDA memory, *managed* memory and *mapped-pinned* memory, we found *mapped-pinned* CUDA memory provided the fastest copying on both platforms. Therefore, we allocated mapped-pinned CUDA memory for Zenoh-pico/Iceoryx in the experiments.

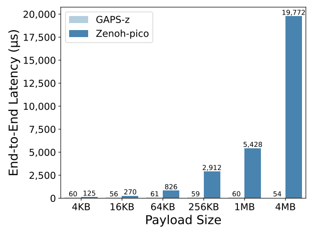
Apart from comparing GAPS with Zenoh-pico/Iceoryx, we compared PyGAPS with Zenoh-python/Iceoryx-python. Zenoh-python is an official Python binding of Zenoh (the Rust version), while Iceoryx-python is a simple Python binding we created for Iceoryx, as it does not have an official one. When publishing PyTorch tensors with Zenoh-python or Iceoryx-python, the tensors have to be serialized into byte strings and deserialized back at subscriber sides. We considered three tensor serialization libraries: Python's *pickle*, PyTorch's *torch.save/load* and HuggingFace's *safetensors* and eventually chose *safetensors* for Zenoh-python and Iceoryx-python because it provided the best performance, as shown in Table III. The table only shows the results on the x86 workstation, but the order of the performance of these libraries are the same on Jetson AGX Orin.

We conducted experiments by varying the number of publishers and subscribers. In the following sections, we discuss the end-to-end latency in one-to-one, one-to-many and many-to-one scenarios.

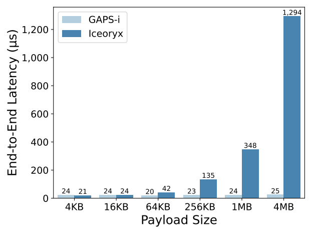
1) One-to-One Scenario: The objective of this experiment is to compare end-to-end latency of different methods under various payload sizes. In the experiment, a publisher

TABLE III
Performance of different serialization libraries on converting a 1000×1000 float tensor into a byte string and vice versa on the x86 workstation.

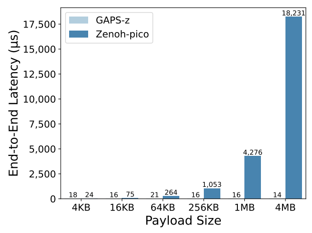
Library	T2B Latency (ms)	B2T Latency (ms)	Total Latency (ms)	Output Length (bytes)
pickle	3.93	2.19	6.12	4,000,418
torch.save	2.36	0.66	3.02	4,001,180
safetensors	2.03	0.48	**2.51**	**4,000,088**



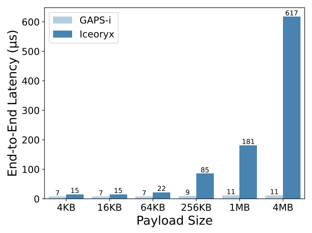
(a) Zenoh-pico vs. GAPS-z (Jetson).



(b) Iceoryx vs. GAPS-i (Jetson).

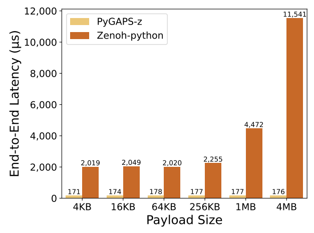


(c) Zenoh-pico vs. GAPS-z (x86).

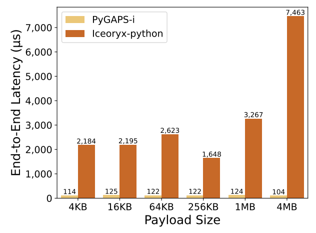


(d) Iceoryx vs. GAPS-i (x86).

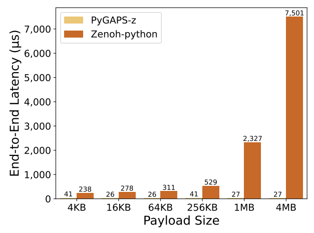
Fig. 4. End-to-end latency on Jetson AGX Orin and the x86 workstation.



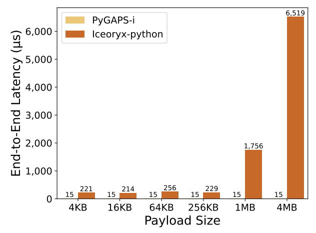
(a) Zenoh-python vs. PyGAPS-z (Jetson).



(b) Iceoryx-python vs. PyGAPS-i (Jetson).



(c) Zenoh-python vs. PyGAPS-z (x86).



(d) Iceoryx-python vs. PyGAPS-i (x86).

Fig. 5. End-to-end latency with Python binding on Jetson AGX Orin and the x86 workstation.

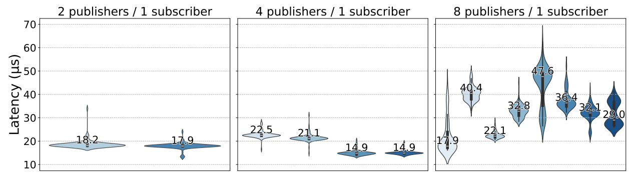
publishes messages for 100 times with a 100-µs interval. The content of the messages are different in each iteration and the median of the measured results is taken. From the results shown in Fig. 4 and Fig. 5, we have four observations.

- *Our works outperform the original middleware in all cases.* It is because our methods avoid copies between GPU and CPU, publications of the whole payload and even serialization/deserialization in the case of publishing PyTorch's tensors. The end-to-end latencies of GAPS/PyGAPS remain constant since the payloads are kept on the shared CUDA memory. Thus, the latency is simply the time of publishing a notification, whose size is independent of payload size.
- *The latency of GAPS-i is lower than that of GAPS-z.* The reason is that Iceoryx utilizes POSIX shared memory and condition variable to realize the pub/sub communication, while Zenoh utilizes UDP multicast. Therefore, the former can deliver messages faster.
- *The experiments show lower latencies on the x86 workstation than on Jetson AGX Orin.* It is because publishing and handling messages involve multiple CPU operations, such as updating shared metadata store and sending UDP messages. As a result, the more powerful the CPUs are, the faster the deliveries are.
- *The experiments on publishing PyTorch tensors take longer time than those on publishing simple data.* For PyGAPS, it is due to the overhead of Python bindings and larger notifications (message ID plus DLPack structure). For Zenoh-python and Iceoryx-python, it is due to the overhead of Python bindings and serialization.

2) Many-to-One and One-to-Many Scenarios: The objective of this experiment is to find out how the number of nodes affects the end-to-end latency. In the experiment, each publisher publishes a 64 KiB message simultaneously for 100 times with a 100-µs interval. The content of the messages are different in each iteration. As shown in Fig. 6a, in the many-to-one scenario, the more publishers there are, the more variant the latencies among pairs are. It is because when there are more publishers, there will be more messages to be handled per subscriber. Each subscriber has only one message-handling thread, causing the coming messages handled one after one. Similarly, in the one-to-many scenario, the more subscribers there are, the more variant the latencies among pairs are, but in a lighter degree. We speculate that there is an underlying portion done sequentially although the behavior of publishing via UDP multicast is nearly parallel. Suppose one subscriber takes t more µs than the other in a one-to-two scenario. In one-to-many scenario, the unluckiest subscriber will wait $(N-1)\times t$ more µs than the luckiest one, where N is the total number of subscribers. By fitting the measured results in Fig. 6b, we can derive the value of $t \approx 0.8$. If there are 32 subscribers, the latency of the slowest pair would be about $31 \times 0.8 = 24.8$ µs slower than the fastest pair. Note that the same situation happens to Zenoh-pico and Iceoryx, where the slowest pair also has slightly higher latency than the fastest pair, but this extra delay should not matter unless the application has very stringent real-time requirements.

C. Computer Vision Pipeline Throughput

Finally, we compared PyGAPS with Python counterparts in a simplified computer vision (CV) application. This application demonstrates a CV pipeline executed on an autonomous vehicle. There are two pipelines, one of which does instance segmentation and the other does scene classification. In the middle of the process, detected human


(a) Many-to-one scenario.

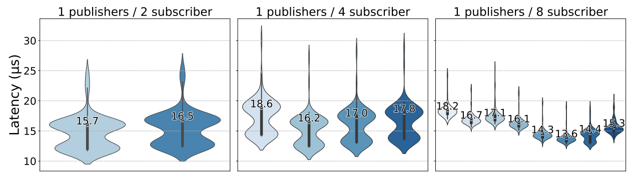
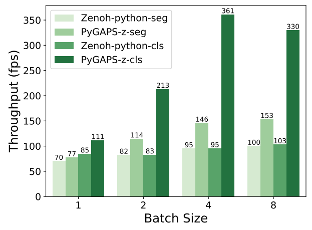

(b) One-to-many scenario.

Fig. 6. End-to-end latency distribution excluding first 10 iterations of GAPS-z with multiple nodes on the x86 workstation. Different colors represent different pairs.


(a) Zenoh-python vs. PyGAPS-z.

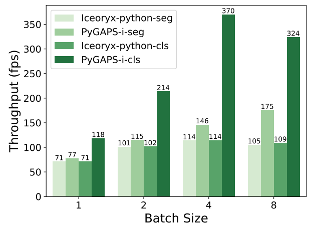

(b) Iceoryx-python vs. PyGAPS-i.

Fig. 8. Throughput comparison on the x86 workstation.

faces are blurred for privacy concerns and transformed into required input format of the models. As shown in Fig. 7, there are four nodes in total. The left node simulates a node processing images from a camera, decoding JPEG files and transforming them into 512×512 float16 frames with GPU. The other three nodes run YOLO11 [26] models on GPU with different pre-trained weights for different tasks. For simplicity, the pre-trained face detection weights are from the yolo-face [27] project, while weights for segmentation and classification are from the authors of YOLO11. As for the testing data, we picked first 100 images with at least two human faces from the testing set of TJU-DHD [28]. We measured the throughput of the pipelines under different small batch sizes.

As shown in Fig. 8, PyGAPS-z and PyGAPS-i outperform Zenoh-python and Iceoryx-python respectively. Comparing PyGAPS-z with Zenoh-python, when the batch size is 4, the segmentation task achieved about 1.5× higher throughput, and the classification task achieved about 3.8× higher throughput. Similarly, comparing PyGAPS-i with Iceoryx-python, when the batch size is 4, the segmentation task achieved about 1.3× higher throughput, and the classification task achieved about 3.2× higher throughput. PyGAPS improved the pipelines because it reduced the need to (de)serialize PyTorch tensors and copy data between GPU and CPU memories. In addition, we can observe that the throughput between two CV tasks are similar when applying either Zenoh-python or Iceoryx-python. It is because the completion time in these scenarios are dominated by the serialization and copies. On the other hand, when applying PyGAPS, the completion time is now dominated by the computations of the models, and the throughput is improved due to reduced communication time.

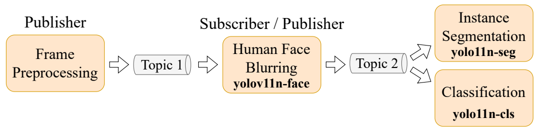


Fig. 7. Experiment setup of the computer vision pipeline.

VI. Conclusion

We present GPU-Aware Pub/Sub communication (GAPS), which is a universal solution to incorporate various pub/-sub middleware, like Zenoh-pico and Iceoryx, with shared CUDA memory for efficient GPU data publications. First, we developed the shared CUDA memory manager to enable the creation and management of shared CUDA memory pools for different topics. Additionally, we adjusted the famous TLSF allocator to be capable of allocating finer memory blocks from such pool. Furthermore, we made GAPS support publishing PyTorch's tensor, enabling faster pub/sub communication between processes running AI models accelerated by GPU.

Our experimental results show that GAPS successfully reduced the end-to-end latency by up to around 100% compared to the traditional two-copy way for delivering GPU data, as it keeps the computation results on GPU's memory during publications, resulting in nearly constant end-to-end communication latency regardless of the payload's size. In the scenario of multiple publishers/subscribers, the end-to-end latency is slightly increased. In our simplified computer vision application, PyGAPS achieved up to 1.5× and 3.8× higher throughput for the segmentation and classification tasks, respectively, compared with existing middleware.

In future work, we would like to maximize the potential for GAPS to accelerate real-time pub/sub GPU-based applications. The shared CUDA memory manager can be improved by sharing the shared CUDA memory pool among multiple topics rather than one per topic for better memory utilization. Challenges include allocating memory space from the pool by publishers of diverse topics and handling the lifecycle of the pool. Apart from that, GAPS-z can be extended to support multi-GPU memory sharing via an efficient memory synchronization method across GPUs over NVLink or RDMA-capable high-speed networks.

References

[1] H. Hua, Y. Li, T. Wang, N. Dong, W. Li, and J. Cao, "Edge computing with artificial intelligence: A machine learning perspective," *ACM*

Comput. Surv., vol. 55, no. 9, jan 2023. [Online]. Available: https://doi.org/10.1145/3555802

[2] R. Singh and S. S. Gill, "Edge ai: A survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667345223000196

[3] H. Lin, "Embedded artificial intelligence: Intelligence on devices," *Computer*, vol. 56, no. 09, pp. 90–93, sep 2023.

[4] F. Oliveira, D. G. Costa, F. Assis, and I. Silva, "Internet of intelligent things: A convergence of embedded systems, edge computing and machine learning," *Internet of Things*, vol. 26, p. 101153, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660524000945

[5] V. Mayoral-Vilches, S. M. Neuman, B. Plancher, and V. J. Reddi, "Robotcore: An open architecture for hardware acceleration in ros 2," in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 9692–9699.

[6] O. Bell, C. Gill, and X. Zhang, "Hardware acceleration with zero-copy memory management for heterogeneous computing," in *2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2023, pp. 28–37.

[7] H. Wu, J. Jin, J. Zhai, Y. Gong, and W. Liu, "Accelerating gpu message communication for autonomous navigation systems," in *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, 2021, pp. 181–191.

[8] A. Corsaro, L. Cominardi, O. Hecart, G. Baldoni, J. E. P. Avital, J. Loudet, C. Guimares, M. Ilyin, and D. Bannov, "Zenoh: Unifying communication, storage and computation from the cloud to the microcontroller," in *2023 26th Euromicro Conference on Digital System Design (DSD)*, 2023, pp. 422–428.

[9] Eclipse-Iceoryx, "Github - eclipse-iceoryx/iceoryx: Eclipse iceoryxtm - true zero-copy inter-process-communication," 2019. [Online]. Available: https://github.com/eclipse-iceoryx/iceoryx

[10] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "Tlsf: a new dynamic memory allocator for real-time systems," in *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, 2004, pp. 79–88.

[11] W.-Y. Liang, Y. Yuan, and H.-J. Lin, "A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds," 2023. [Online]. Available: https://arxiv.org/abs/2303.09419

[12] J. Zhang, X. Yu, S. Ha, J. P. Queralta, and T. Westerlund, "Comparison of dds, mqtt, and zenoh in edge-to-edge and edge-to-cloud communication for distributed ros 2 systems," 2023. [Online]. Available: https://arxiv.org/abs/2309.07496

[13] Eclipse-Cyclonedds, "Github - eclipse-cyclonedds/cyclonedds: Eclipse cyclone dds project," 2019. [Online]. Available: https://github.com/eclipse-cyclonedds/cyclonedds

[14] Eclipse-Iceoryx, "Github - eclipse-iceoryx/iceoryx2: Eclipse iceoryx2tm - true zero-copy inter-process-communication in pure rust," 2023. [Online]. Available: https://github.com/eclipse-iceoryx/iceoryx2

[15] NVIDIA, "Cuda for tegra release 12.8," 2025, please refer to Section 3.6. CUDA Features Not Supported on Tegra; Accessed: 2025-03-23. [Online]. Available: https://docs.nvidia.com/cuda/pdf/CUDA-For-Tegra-AppNote.pdf

[16] D. E. Knuth, "The art of computer programming. volume 1: Fundamental algorithms." *Journal of the American Statistical Association*, vol. 64, no. 325, p. 401, mar 1969. [Online]. Available: https://doi.org/10.2307/2283757

[17] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: solved?" *SIGPLAN Not.*, vol. 34, no. 3, p. 26–36, oct 1998. [Online]. Available: https://doi.org/10.1145/301589.286864

[18] M. Masmano, I. Ripoll, J. Real, A. Crespo, and A. Wellings, "Implementation of a constant-time dynamic storage allocator," *Software: Practice and Experience*, vol. 38, pp. 995 – 1026, 08 2008.

[19] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, ser. USTC'94. USA: USENIX Association, 1994, p. 6.

[20] R. Giannessi, A. Biondi, and A. Biasci, "RT-Mimalloc: A New Look at Dynamic Memory Allocation for Real-Time Systems," in *2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2024, pp. 173–185. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/RTAS61025.2024.00022

[21] D. Leijen, B. Zorn, and L. De Moura, "Mimalloc: Free list sharding in action," in *Programming Languages and Systems: 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1–4, 2019, Proceedings 17*. Springer, 2019, pp. 244–265.

[22] M. Khasgiwale, V. Sharma, S. Mishra, B. Thadichi, J. John, and R. Khanna, "Shimmy: Accelerating inter-container communication for the iot edge," in *GLOBECOM 2023 - 2023 IEEE Global Communications Conference*, 2023, pp. 4461–4466.

[23] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, p. 21–65, feb 1991. [Online]. Available: https://doi.org/10.1145/103727.103729

[24] W. Jakob, "nanobind: tiny and efficient c++/python bindings," 2022, https://github.com/wjakob/nanobind.

[25] Distributed (Deep) Machine Learning Community, "dlpack: common in-memory tensor structure," 2017, https://github.com/dmlc/dlpack.

[26] G. Jocher, J. Qiu, and A. Chaurasia, "Ultralytics YOLO," Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[27] A. Kanametov, "yolo-face: Yolo face in pytorch," 2024, https://github.com/akanametov/yolo-face.

[28] Y. Pang, J. Cao, Y. Li, J. Xie, H. Sun, and J. Gong, "Tju-dhd: A diverse high-resolution dataset for object detection," *IEEE Transactions on Image Processing*, 2021.