NICE: A Non-intrusive In-Storage-Computing Framework for Embedded Applications

Tianyu Wang*, Yongbiao Zhu*, Shaoqi Li*, Jin Xue†, Chenlin Ma✉*, Yi Wang✉*,
Zhaoyan Shen‡§, and Zili Shao†
*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
†Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
‡School of Computer Science and Technology, Shandong University, Qingdao, China
§Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, China
tywang@szu.edu.cn, {zhuyongbiao2023, lishaoqi2023}@email.szu.edu.cn, jinxue@cse.cuhk.edu.hk,
✉{chenlin.ma, yiwang}@szu.edu.cn, shenzhaoyan@sdu.edu.cn, shao@cse.cuhk.edu.hk

***Abstract*—Embedded machine learning applications face challenges related to massive data movement and high computational intensity, exacerbated by the limited performance of mobile devices. Computational Storage Devices (CSDs) pose huge potential for accelerating both data-intensive and computation-intensive embedded machine learning tasks by effectively reducing data movement and leveraging built-in accelerators. However, existing in-storage-computing (ISC) frameworks either require invasive customization of existing host driver layers or necessitate complex device firmware modifications, hindering the widespread deployment of CSDs. Additionally, the lack of file semantics and the constrained internal resources within CSD implicitly compromise system performance and impact normal read/write performance.**

In this paper, we aim to provide a Non-intrusive In-storage-Computing framework for Embedded applications, named NICE. This framework includes an easy-to-use in-storage-computing programming interface that bypasses the kernel stack and requires no modification to the host NVMe driver, which is achieved through a novel hyper-addressing-based programming library and a file-aware page data layout within the CSD. Additionally, we incorporate a lightweight kernel with coroutine-based command scheduling and several FPGA-based accelerators within the storage device firmware to enhance the performance of embedded machine learning applications while ensuring that the normal I/O performance remains unaffected.

NICE is implemented on real CSD hardware integrated with ARM and FPGA. Experimental results demonstrate that our NICE framework can achieve an average latency performance improvement of 43.5x (9.32x) compared to CPU- (GPU-) based embedded machine learning solutions using the state-of-the-art NVIDIA Jetson NX platform, with 27.5x (4.3x) higher energy efficiency. NICE also has 34.2x less software and I/O performance overheads than state-of-the-art ISC frameworks.

***Index Terms*—Embedded System, Solid State Drive, In Storage Computing, Computational Storage Device**

I. Introduction

As embedded devices are widely adopted in both daily life and industrial fields, embedded machine learning paves the way for real-time and privacy-sensitive AI applications [1]–[7]. However, embedded machine learning faces substantial challenges related to data movement overhead and high computational intensity. Computational Storage Devices (CSDs) exhibit great potential to address these challenges. Benefitting from in-storage-computing (ISC) capabilities, CSDs can avoid the bottleneck of extensive data transfer between processing units and storage devices. Many data-intensive applications, such as neural network inference and nearest neighbor search in recommendation systems, gain a lot of benefits from ISC technology [8]–[14], especially for embedded devices with limited computational resources.

Currently, to integrate CSDs into existing systems, three key issues need to be considered: 1) how programmers can offload programs to the accelerator, 2) how the CSD firmware handles ISC requests, and 3) how to design application-specific accelerator logic in CSDs. However, recent works mainly focus on accelerator logic design for specific applications, overlooking the minimization of modifications to existing systems during integration and the provision of programmer-friendly ISC frameworks. For instance, [9], [15], [16] enrich the NVMe protocol by utilizing some unused bits to add extra NVMe commands for ISC invocations. However, this approach requires heavy modifications to the existing NVMe driver in the host system. Other works [14], [17] send network packages to the CSD to invoke built-in accelerators. Although this approach can minimize modifications to the host system by only providing a library based on Ethernet protocols, it usually burdens the CSD by equipping it with a full-fledged Linux to receive and process network packets, putting heavy pressure on the inherently low-performance CSD controller. Additionally, existing solutions overlook the impact of ISC tasks on normal I/O performance. Therefore, in this paper, we aim to propose **a non-intrusive ISC framework** that allows programmers to easily offload various embedded machine learning applications to the CSD without requiring complex modifications to existing systems and with minimal I/O performance overhead. To achieve this, four challenging issues must be addressed.

The first challenge is how to distinguish ISC invocations from normal NVMe Read/Write requests. Without any extra NVMe commands, we propose a novel hyper-addressing-based ISC method. Specifically, our NICE framework exposes twice the actual capacity of each CSD to the host system. The extra address range is named *HyperAddress*. When programmers need to invoke ISC tasks, they can issue a normal NVMe Read/Write command with the LPA (Logical Page Address)

✉: Chenlin Ma and Yi Wang are the corresponding authors.

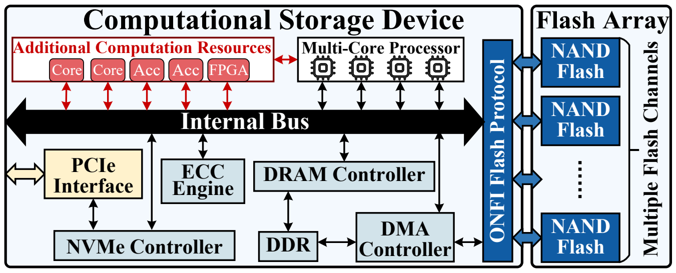


Fig. 1. A typical computational storage device architecture.

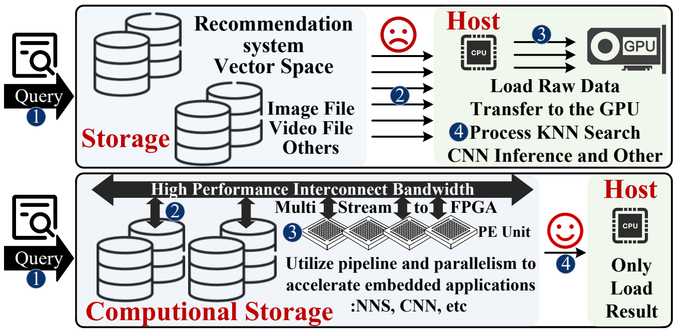


Fig. 2. The process of handling requests in computational storage devices.

located at the *HyperAddress* range, while the Read/Write command within a normal address range still acts as the original Read/Write command. Thus, the CSD can easily distinguish ISC invocations by checking if the LPA exceeds the normal address range. With such a hyper-addressing-based method, we can initiate ISC invocations **without any modifications** to the existing host NVMe driver.

The second challenge lies in bridging the semantic gap between the host file system and the device firmware. Many machine learning applications require files as inputs (e.g., images in image recognition). However, the file system on the host side translates a file read into multiple page reads, including inode pages and data blocks. From the perspective of the CSD, these are individual NVMe read commands without any concept related to files. To address this, we propose an in-CSD file-aware page data layout to identify each file used for ISC tasks. **As a non-intrusive framework**, we aim to refrain from modifying anything within the existing host file system or negatively impacting it. For instance, CSDs can be formatted normally as EXT-4, and users should be able to initiate requests through the filesystem's write-read operations seamlessly. As only write operations change file layouts on disk, we monitor each NVMe write command to detect inode page modifications, retrieve all data blocks' LPN (logical page number) from inode pages, and obtain relevant PPNs (physical page numbers) based on the mapping table. We then establish a doubly linked list stored in the Out-Of-Bound (OOB) area of each physical flash page. Through this doubly linked list, we can ensure that each file is correctly fed into the CSD's built-in accelerator following the correct order.

The third challenge is how to prevent ISC tasks from affecting the throughput of normal read and write requests. Since we offload most of the host's computations to the CSD, although we utilize FPGA resources to relieve the pressure of the CSD's built-in microcontroller, the polling-based FPGA scheduling still inevitably impacts the SSD's regular read and write performance. Therefore, we design a coroutine-based scheduling strategy and locking mechanisms within the CSD to seamlessly handle both normal I/O requests and ISC tasks. This approach ensures that **our NICE framework is non-intrusive** for normal I/O performance.

The last challenge is the accelerator logic design for embedded machine learning applications. Unlike other works dedicated solely to designing FPGA accelerators, our FPGA resources are also occupied by SSD controller logic. To efficiently utilize FPGA resources, we propose a data partitioning engine and a task allocation engine to achieve computational-level parallelism. We also introduce two ping-pong buffers in the accelerator design to achieve task-level parallelism. With this approach, we have implemented several built-in accelerators, significantly reducing data movement between the host CPU and storage devices. Finally, we have implemented NICE on real CSD hardware with ARM and FPGA resources and observed significant performance improvements.

II. Background and Motivation

A. Computational Storage Device

As many data-intensive applications emerge, traditional von Neumann architecture faces more severe data stall problems caused by limited storage performance. Instead of migrating data from the storage side to the CPU side, the CSD mitigates the data stall problem by offloading computations to the storage side close to the data. Figure 1 shows a typical CSD architecture, in which the microcontroller in the CSD takes more responsibility (e.g., ISC tasks) than the original SSD controller which only runs some Flash Translation Layer (FTL) functions (address mapping, garbage collection, etc.). Moreover, due to the limited computing resources in the microcontroller, CSDs usually equip some application-specific acceleration logic such as FPGA resources for more efficient task offloading. With such an ISC architecture, some data-intensive tasks can be directly performed near the data, effectively reducing data movements and alleviating pressure on the host CPU.

B. In-storage-computing Framework

Many recent works are focusing on utilizing the CSD to accelerate data-intensive applications. Table I summarizes some state-of-the-art works from three perspectives: 1) modifications made to the existing host system; 2) methods for invoking ISC tasks within the CSD; and 3) the types of resources used by ISC tasks. As shown below, some works invoke in-storage functions via Ethernet-based network packets [14], [17], facilitating seamless integration into the current host system. However, they all need full-fledged Linux running inside the CSD microcontroller to handle network packets, which inherently impacts the storage performance since the microcontroller still needs to perform FTL tasks. Others either need to revise the NVMe protocol by adding some extra command to invoke in-storage functions or need a fully customized host driver for CSD products. Those works more or less require host driver changes, preventing their technologies from being widely adopted in existing systems. Thus, we aim to design a non-intrusive framework without any modification to the existing host system, while taking a lightweight kernel in the CSD to avoid performance impact. Furthermore, some existing works [9], [12], [15], [16] directly take the microcontroller in the CSD to perform in-storage-computing tasks, it is hard to strike a balance between computation tasks and FTL tasks. Thus, our NICE employs FPGA-based logic designed for in-storage-computing tasks to relieve the microcontroller's load.

TABLE I
THE COMPARISON BETWEEN NICE AND RECENT WORKS.

Related Works	Host Driver	CSD Runtime	Acceleration Logic
RecSSD [12]	Customized	Bare metal	Inside firmware
Virtual Object [18]	Extra NVMe command	Bare metal	Inside firmware
nKV [11]	Customized	Bare metal	FPGA-based
ISC Hadoop [19]	Customized	Customized kernel	Inside firmware
RM-SSD [13]	Customized	FPGA logic	FPGA-based
Newport [15]	Extra NVMe command	Full-fledged Linux	Inside firmware
SmartSAGE [9]	Extra NVMe command	Bare metal	Inside firmware
SoftSSD [16]	Extra NVMe command	Lightweight kernel	Inside firmware
V-Store [20]	Customized	Customized kernel	FPGA-based
Containerized framework [17]	Network command	Full-fledged Linux	Inside firmware
Hybrid CNN Training [14]	Network command	Full-fledged Linux	Inside firmware
PreCog [8]	Customized	Bare metal	FPGA-based
Cognitive SSD [10]	Customized	Customized kernel	FPGA-based
Optimizing NDP Operations [21]	Customized	Full-fledged File System	Inside firmware
NICE (this work)	**Unmodified**	**Lightweight kernel**	**FPGA-based**

C. Embedded Machine Learning Applications

Embedded Convolutional Neural Network. The embedded CNN has been widely deployed for real-time object detection. However, as the microcontroller in the CSD only has limited computing resources even with FPGA assisted (constrained by maximal 15W power through M.2 slot [22]), we need to judiciously consider which type of neural network is suitable for such an embedding environment. Considering the Storage Intensity (SI) defined by $SI = \frac{FLOPS}{I/Osize}$, which indicates how much computation is associated with each byte transfer from storage devices, a neural network with a smaller SI is more suitable for embedded environments due to less computation requirement [23], as well as obtaining more benefits from the CSD with less data movement. Thus, we select lightweight CNN models with fewer network layers and weights for more practical embedded CNN inference.

Related works on CNN accelerators [24]–[26] have achieved high throughput, but their power consumption is not tolerable for CSDs, making them unsuitable for deployment inside a storage system. Although a CNN inference accelerator with extremely low power consumption (0.34W) was implemented in [27], its inference latency is very high. In [28], in-storage-computing and computing-in-flash techniques are fused to accelerate CNNs utilized in video processing. However, this technique does not consider the placement in resource-limited embedded environments and is only implemented in simulators without validation on real hardware platforms. Thus, based on the above, we have designed a low-power CNN accelerator suitable for being placed inside SSDs, and have conducted evaluations on a real CSD hardware platform.

Embedded Nearest Neighbor Search. The embedded nearest neighbor search (NNS) algorithm is primarily used in privacy-sensitive recommendation systems as it prefers not to use cloud-based data processing, protecting users' privacy from leakage. With the increasing scale of recommendation systems, the size of vector space has been approaching the limit of memory capacity, especially for memory-constrained embedded devices. When the size of vectors exceeds the memory capacity, the searching process requires fetching a portion of vectors from the storage device for distance calculation. This process incurs significant data movement overhead, resulting in intolerable query latency.

Recent works [29]–[31] have designed FPGA-based accelerators for the efficient acceleration of NNS. However, they all overlook the time overhead caused by fetching the corpus from storage devices. Some works [32] improve KNN's vector retrieval speed by reducing external memory access, but they do not consider external storage accesses. Although some works also take ISC architectures to accelerate NNS [20], [33], they mainly focus on the specific accelerator design instead of the ISC framework, hindering programmers from easily incorporating a CSD into their applications. Therefore, to efficiently conduct vector retrieval and make the CSD platform easy to use, we propose the NICE framework, adopting an ISC architecture to accelerate nearest neighbor search, as illustrated in Figure 2. We leverage FPGA resources in the CSD to efficiently perform vector retrieval, with only the essential data returned to reduce time and power consumption.

III. THE OVERVIEW OF NICE

Figure 3 illustrates the overall architecture of our NICE design. As a non-intrusive framework, we allow the CSD to be formatted with various file systems and initiate data read/write requests through standard I/O operations. For ISC requests, users can easily invoke them using our NICE library without modification to the host NVMe driver. The hyper-addressing-based programming framework can distinguish specific ISC requests from regular NVMe I/O requests, which is achieved by setting the request LBA to (the first logical page address of the file + hyper-address). Within the CSD firmware, we introduce an address interpreter to identify the address range (see Section IV-A). For ISC requests, as CSDs lack file semantics, we also need to retrieve all relevant file pages through file-aware page data layout (see Section IV-B), then concatenate them in order as input and feed them to built-in accelerators. Next, to minimize normal I/O performance impacts caused by ISC tasks, we propose a coroutine-based scheduling mechanism in the CSD firmware (see Section IV-C). Finally, we have designed FPGA-based accelerators for several embedded machine learning applications (see Section IV-D). Thus, the

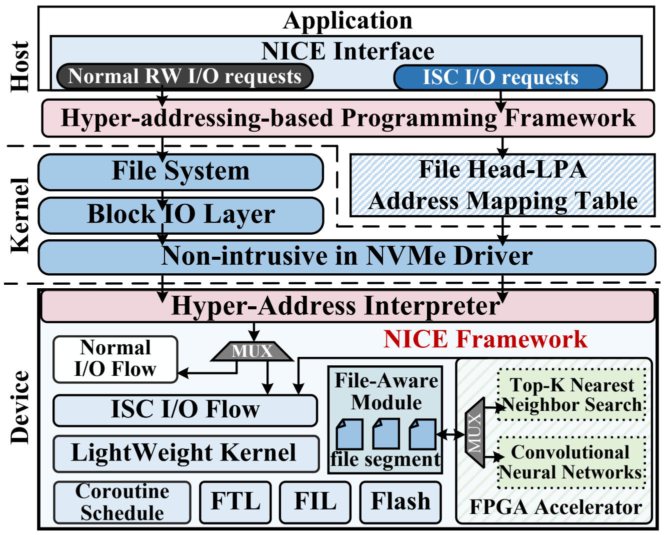


Fig. 3. NICE design overview.

CSD only returns computation results to the host, saving the overhead of raw data movements and achieving higher performance and energy efficiency.

IV. Technical Details

A. Hyper-addressing-based Programming Framework

As a non-intrusive programming framework, we can only follow existing NVMe commands to invoke built-in accelerators without introducing additional command types. Thus, we choose to manipulate the address range for different purposes (normal I/O or ISC invocation). Specifically, when integrating a CSD into the host system, it will expose twice the actual capacity to the host, resulting in two logical address ranges: the Normal Address Range (the lower half) and the *Hyper Address Range* (the upper half), as shown in Figure 4 (a 512GB CSD in this example). The original NVMe Read/Write commands continue to function as usual since all NVMe requests within the Normal Address Range are processed in the same manner as those in a standard SSD.

TABLE II
NICE APIs.

APIs	Description
open (FilePath, flag)	Re-encapsulate the POSIX API, insert the file path and first address to Hashtable after obtaining the file descriptor.
NICE_Write (FilePath, op)	Initiate ISC request via hyper-address, select different accelerators for the file.
NICE_Read (FilePath, buf, size)	Initiate ISC request via hyper-address, read ISC calculation results.

For ISC requests, leveraging the extended logical address range, we offer a user-friendly ISC library including two important functions: $NICE\_Write(FilePath, op)$ and $NICE\_Read(FilePath, buf, length)$ in table II. In these functions, we seamlessly translate an ISC invocation into a standard NVMe command, with the LPA situated at the designated *Hyper Address Range*. Therefore, the CSD can easily distinguish ISC requests from regular NVMe requests based on the input logical address. For the *open* API, we re-encapsulate it in the POSIX *open* API with preprocessing of the NICE framework.

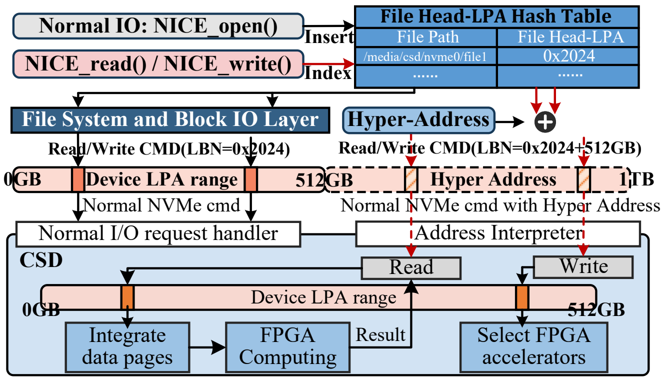


Fig. 4. The hyper-addressing-based framework.

• **NICE Open Operation.** We have extended the POSIX *open* API to incorporate file LPA information within the NICE framework. Since the NICE framework passes ISC requests via hyper-addresses, it is essential to establish relationships between files and their corresponding LPA numbers. Thus, NICE maintains a Head-LPA hash table that maps <the file path> to <the first logical address of the file>, eliminating the necessity to traverse the file path from the root directory to locate the inode number when initiating ISC requests. Thus, our NICE framework can bypass several OS kernel layers achieving higher ISC efficiency.

Such a relationship can be built when every file is created. Given that file creation and storage are prerequisites for ISC operations, the *open* function is crucial before performing ISC tasks on a specific file. Since the file creation would inevitably invoke the *open* function to return the file descriptor, we have re-encapsulated the *open* function to insert $Hash(File\_Path)$ and the first address of the file into the hash table at the time of file creation. Furthermore, as the *open* function conducts permission checks, files lacking sufficient permissions will not be included in the Head-LPA hash table to prevent malicious attacks. The hash table storage overhead (e.g., 0.34MB per 10,000 files) is also negligible for large-capacity SSDs.

• **NICE Read Operation.** To initiate ISC tasks, programmers only need to replace the $Read()$ function with our $NICE\_Read(FileName, buf, size)$, requiring no additional changes. Within the $NICE\_Read$ context, the file's first logical address is retrieved by searching the hash table based on the file name, eliminating the need to traverse the file system stack from the root directory. During this process, we modify the start LPA in the read operation with the *Hyper Address* instead of the original LPA. The calculation of the *Hyper Address* involves adding a fixed offset equivalent to the actual capacity (e.g., 512GB in Figure 4) to the original address. This establishes a one-to-one mapping between the *Normal Address Range* and the *Hyper Address Range*, which guarantees that even if the host issues a $NICE\_Read()$ request with any arbitrary *Hyper Address*, the CSD can identify the corresponding original address. Finally, we employ *ioctl* to transmit the modified read command to the CSD.

Upon receiving an NVMe read request, the CSD initially examines whether the request's start LPA is located within the *Hyper Address Range*. If yes, the CSD conducts ISC tasks using the data stored at the original LPA, which is obtainable by

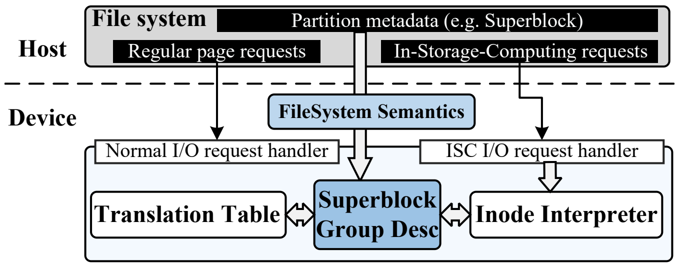


Fig. 5. The overview of file-aware in-CSD page data layout.

subtracting the actual capacity from the *Hyper Address*. Hence, by utilizing $NICE\_Read(FileName, buf, size)$, programmers can easily perform ISC tasks, obtaining result feedback directly without transferring large raw data to the host.

• **NICE Write Operation.** To support various embedded machine learning applications in the CSD, programmers must be able to select different accelerators. This selection is achieved through the $NICE\_Write(FileName, op)$ function in our NICE library. Similar to the $NICE\_Read$ function, we replace the original LPA with the corresponding *Hyper Address* when initiating the write command. As no actual data modification occurs, the identifier of the chosen application model will be written, whose size is determined by the number of embedded accelerators implemented in the CSD. To avoid data inconsistency caused by the accelerator's identifier, it cannot be stored within the data page. Thus, we leverage the OOB area in each physical flash page to store this identifier, whose size is smaller enough regarding more than 400 bytes of OOB space in a 4KB flash page.

B. File-aware In-CSD Page Data Layout

Due to the absence of file semantics within the CSD, bridging the semantic gap between the host file system and the CSD runtime is necessary for initiating ISC tasks. Thus, we propose a file-aware CSD internal page data layout to preserve relationships among file pages. This layout ensures that an ISC operation with only one LPA transferred to the CSD can be accurately identified and executed. This layout ensures that a complete file can be accurately identified in CSD and fed to the computing unit with sequential information with just one NVMe command.

Originally, obtaining file semantics involved initiating system calls in the kernel's Virtual File System (VFS) to traverse the namespace tree along the file path. Specifically, it begins by locating the corresponding inode data in the inode table based on the inode number of the root directory, and then obtaining the inode data of the next-level directory level-by-level, until reaching the target file. NICE eliminates this time-consuming path traversal process by constructing and maintaining a mapping between the file name and the logical address of its first block (as mentioned in Section IV-A: *NICE Open Operation*). When initiating ISC operations on a specific file, we simply issue a hyper-addressing-based read command to its first block according to the mapping table, saving the latency of traversing the whole path from the root directory.

After path traversal, existing ISC architectures commonly require reading the page where the inode corresponding to

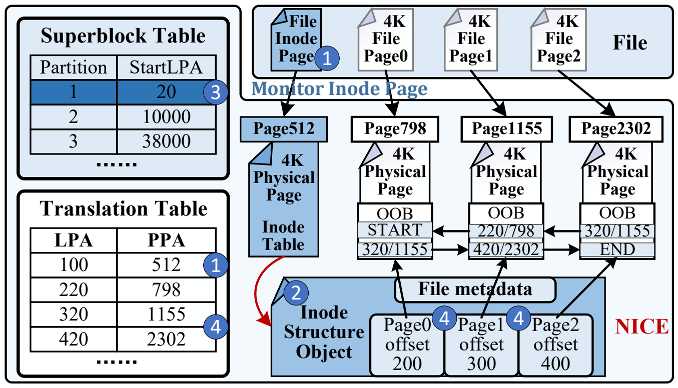


Fig. 6. The details of file-aware in-CSD page data layout.

the file is located in the host and parsing all the data blocks corresponding to the file. To avoid extra I/O caused by reading the inode page, we offload the inode page interpretation into the CSD.

Figure 5 provides an overview of our file-aware CSD internal page data layout design, demonstrating additional information that needs to be maintained in the CSD firmware (e.g., partition and filesystem metadata). When the host formats the SSD into a file system such as EXT-4, the file system's superblock, group descriptor, and other metadata are written to specific logical pages. Traditional SSD systems have this data but do not use it, while our system will monitor and use this data to obtain the semantics of the file in the CSD. As the LPA of data blocks recorded in the inode page serves solely as an offset to the partition's starting LPA rather than representing an absolute LPA, the CSD without partition information faces difficulty in accurately identifying the correct LPA for data blocks. Thus, we record the starting LPA of each partition to a *Superblock Table* in the CSD.

Considering the possibility that the file's first LPA recorded in the Head-LPA mapping table may not reference an inode page, we additionally maintain a doubly linked list within the Out-Of-Bound (OOB) area of each physical data page, recording its corresponding predecessor and successor logical/physical data pages, as depicted in Figure 6. When an ISC command (e.g., $NICE\_Read()$) is initiated, the CSD receives an NVMe Read command with its LPA being the logical address of the file's first page. Suppose the first LPA corresponds to the file's inode. In that case, we directly parse the inode page to fetch LPA offsets of all data pages, and then add the partition's StartLPA from the *Superblock Table* to obtain the absolute LPA of data pages and corresponding PPAs. If the first LPA only points to a specific data page, we can still retrieve all data pages from the doubly linked list.

To construct the doubly linked list in the OOB area of each data page, we persistently monitor each page written to determine if it is an inode page. For instance, as illustrated in Figure 6, we first recognize an inode page written with the #100 LPA and assign a physical page #512 for it through the translation table (➊). Then we read the inode page to get all data blocks of this input file (➋). As the inode LPA must be situated in a partition whose LPA space is consecutive, we can easily determine its associated partition using the *Superblock Table* (➌). Finally, by adding each data page's LPA offset to

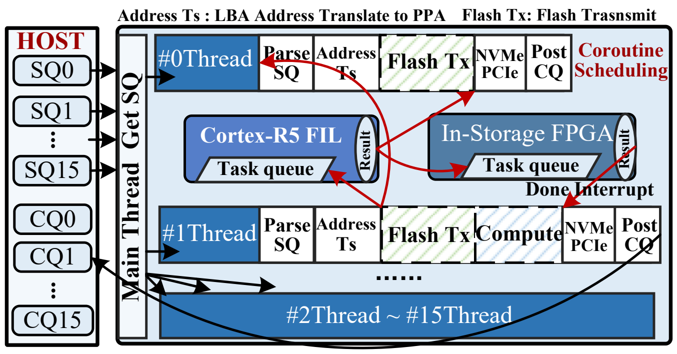


Fig. 7. I/O Schedule based on Coroutine.

the starting LPA of this partition, we can get the absolute LPA number of each data page (❹). After checking with the translation table, both logical and physical page numbers can be retrieved, which are then written to the OOB area of each data page. Thus, we can get such file semantics **without changing the host filesystem.**

Another challenge is related to garbage collection in the CSD firmware, which may change the physical locations of partial data pages of a file, leading to a broken linked list of the PPAs. However, since we also record a linked list of the LPAs that would not be changed during garbage collection, we can restore the PPA-linked list from the LPA-linked list. Specifically, each time we retrieve a data page through the PPA-linked list, we also check if it matches the LPA-linked list. Any detected mismatch will lead to a repeated read (read the correct PPA through the LPA-linked list) according to the newest translation table. The physical page with incorrect OOB information is also marked as "*vulnerable*". When all data pages of the file are marked *vulnerable*, we additionally perform a reconstruction for all data pages' OOB area to recover the correct PPA-linked list.

C. LightWeight Runtime and Coroutine-based Scheduling

NICE has a lightweight runtime based on a multi-core processor and FPGA resources on the CSD platform. It contains both coroutine-based scheduling mechanisms and lock strategies that can simultaneously handle excessive loads of both normal I/O and ISC requests, ensuring non-intrusive I/O performance of the CSD. As shown in Figure 7, the ISC request processing can be divided into multiple stages, including SQ retrieval, SQ parsing, logical address translation, flash data transmission, in-storage calculation, and CQ postage. Flash data transfer and in-storage calculation are the most time-consuming operations. Therefore, we place the flash data transfer stage on the Cortex-R5 co-processor, relieving the pressure of Cortex-A53 (main processor). R5 interacts with A53 through the coroutine-based sleep-wake mechanism and task queue. For ISC logic, we implement it using FPGA resources and utilize an interrupt-based wake-up mechanism to release computing resources in the main A53 processor, thus minimizing the impact on normal I/O performance. The major job of the master coroutine is to get SQ requests and dispatch them to each slave coroutine. We give each slave coroutine a run queue, from which it can take out requests for processing. For incoming ISC requests, we prioritize their

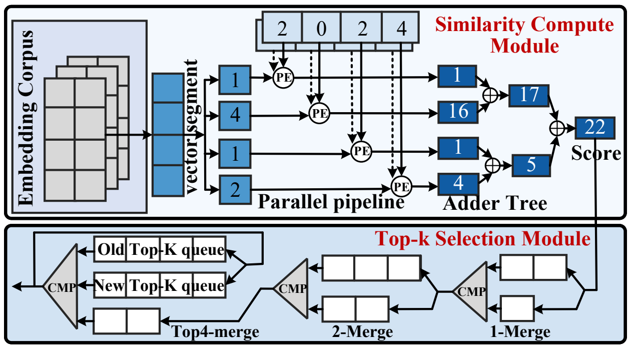


Fig. 8. The details of FPGA-based NNS accelerator in NICE.

processing to be completed as soon as possible. By offloading the two most time-consuming stages of ISC requests to other cores, we can minimize the impact on normal I/O performance while ensuring optimal ISC performance. In addition, since we expose a new hyper-address space (with the same size as the original address space) in the CSD to initiate ISC requests, during the master coroutine's task dispatching process, we will determine whether ISC requests and normal I/O requests hit the same original address space. If so, the two coroutines can share the data read from NAND flash, increasing data reusability and improving I/O performance.

D. Accelerator Design

1) In-Storage-NNS Accelerator: For data-intensive machine learning applications such as NNS, we have implemented an FPGA-based in-storage KNN accelerator into NICE, called **"NICE-K"**. We offload all computations involved in vector retrieval, including similarity calculation and top-k calculation, from the host to the CSD. The architecture of NICE-K, depicted in Figure 8, comprises the similarity calculation module and the top-K selection module.

Similarity Calculation Module. To fully leverage FPGA parallelism, we partition the embeddings in the corpus based on dimensional sizes. When a query is received, we swiftly load the partitioned vectors into the calculation module within a single clock cycle using multiple AXI buses. This enables us to compute the similarity between the query vector and the vectors in the corpus across different dimensions concurrently. Since similarity computation typically involves subtraction followed by multiplication (such as Euclidean distance), we have implemented the similarity calculation module in a pipelined manner. Finally, the results from different dimensions for the same vector are fed into a pipelined addition tree, ultimately producing the final similarity score.

Top-k Selection Module. To improve the throughput of NNS, we have implemented a pipeline-based Top-K selector as described in [34]. This selector receives a score every cycle and outputs data within the top-K range. It comprises *i-merge* and *Topk-merge* components, where multiple *i-merges* can be combined to create an *i-sorter* component. The data input starts from *1-merge* and progresses to *Top4-merge* based on the specified value of K (e.g., 4 in the Figure 8). The *Top4-merge* module includes two queues: one to store the last top-K data and the other to hold new data. This pipelined selector operates in a total of $k + log_2 k$ clock cycles. However, when

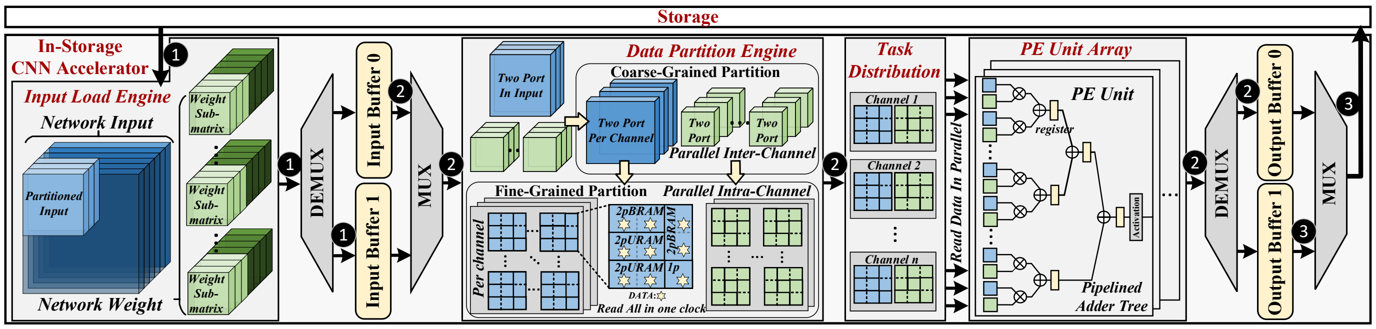


Fig. 9. The details of FPGA-based CNN accelerator in NICE.

processing multiple batches of data, a single selector may not precisely match the score output frequency of the similarity module. Therefore, we employ parallel pipeline selectors to enhance the throughput for multi-batch processing.

2) In-Storage-CNN Accelerator: Furthermore, to empower NICE in handling more intricate embedded machine learning tasks, we have implemented a dedicated neural network accelerator tailored for in-storage CNN inference, referred to **"NICE-C"**. Users can offload neural network models to the SSD, execute in-storage inference tasks, and then obtain the inference results transferred to the host instead of moving large amounts of raw data. We offload all computations of commonly used neural network layers such as convolutional layers, pooling layers, linear layers, depth-wise convolutions, pointwise convolutions, inverse residual layers, and activation layers to the CSD. The accelerator consists of five main modules: the input load engine, input partition engine, task distribution engine, processing element (PE) array, and input-output ping-pong buffers. The overview is illustrated in Figure 9, where we fully exploit the parallelism of the FPGA from both the computational and task levels.

Computation-Level Parallelism. Due to the limited resources in FPGA, preloading all input data and weights into the accelerator is impossible. Therefore, the *Input Load Engine* is needed to select and load a portion of the network inputs and weights. The read data is temporarily stored in FPGA RAMs (e.g., BRAM, URAM, LUTRAM) which can be configured by programmers. However, to achieve maximum computation-level parallelism in the PE array, higher RAM bandwidth is required, which is constrained by limited read/write ports for each RAM block. Therefore, we need a *Data Partition Engine* to divide the input data into different RAM blocks. We first perform a coarse-grained partition for feature maps and weight matrices, storing each image channel in different RAM blocks, making the data transmission benefit from more independent read/write ports. In this way, the data in multiple channels can be fetched to the PE array simultaneously. Fine-grained partition is further introduced for higher throughput, where the data in each channel is divided based on the CNN kernel size and stored in different RAM blocks. This enables multiple read/write ports in one image channel, allowing the PE array to fetch all the data in one image channel with one cycle. Then, as the data has been partitioned and stored in different RAM blocks, *Task Distribution Engine* schedules the partitioned data to idle PE units, allowing them to run simultaneously for maximum computation-level parallelism. A pipelined adder tree is utilized for result accumulation and the results are finally stored in the output buffer and transferred back.

Task-Level Parallelism. To overlap the data transmission latency with the computation, we further explore task-level parallelism by utilizing two ping-pong buffers in both input/output paths. This allows for the concurrent execution of three stages: ❶ loading a portion of data from memory, ❷ partitioning data and performing parallel computation with PE array, and ❸ writing computation results back to memory. With a judicious consideration for the latency of the three stages, developers can trade off the latency of stage 2 with more or less hardware resources to match the latency of stages 1 and 3, thereby improving the overall inference performance.

E. NICE Implementaion

1) Softeware Implementation: Fig 10 shows the detailed workflow of ISC requests using the NICE framework. When calling $NICE\_Read(FilePath, buf, size)$, we need to specify the file path and set the size to indicate how many bytes of the file to perform in-storage computation (0 means the entire file). Afterward, we hash the file path and use the result as a key to find the corresponding file's first LPA in the hash table. This hash table is constructed when calling re-encapsulated $open$ to create a file and invoking regular $write$ to store file data. Once we obtain the file's first LPA, we calculate its corresponding hyper-address and then submit a read request to the device driver.

Upon receiving the read request, the CSD resolves the address back to the original LPA. Then we find the corresponding PPA through the address translation table. If it has been cached in the FTL cache, the page can be directly read, and the inode number and the doubly linked list can be obtained from the OOB. Otherwise, it needs to be read from the NAND flash. By leveraging the inode number and existing file system metadata, we can determine the location of the inode page. This allows us to get the detailed file information and instruct the FPGA to retrieve the file from NAND Flash. After ISC task execution, the FPGA transfers results via the NVMe controller to the buffer specified in the $NICE\_Read$ function.

2) Hardware Implementation: Due to limited FPGA resources, unlike other state-of-the-art works that focus solely on using all hardware resources to implement FPGA accelerators, our FPGA resources also need to be utilized for SSD controller logic. This results in relatively few resources available for accelerator design, as shown in Table III which exhibits FPGA resource utilization in detail. Therefore, we have only

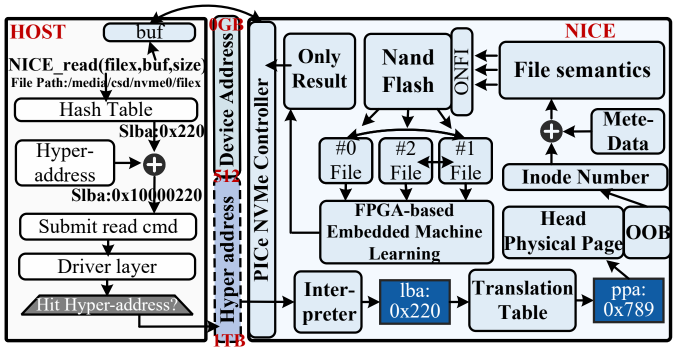


Fig. 10. The details of I/O Flow in NICE.

TABLE III
THE FPGA UTILIZATION OF NICE.

Module	LUT	FF	BRAM	DSP	URAM
NVMe Controller	11034	31287	46.5	0	0
DDR Controller	15415	17919	25.5	3	0
Flash Controller	61042	33770	28	0	0
CNN Accelerator	71862	68193	165	351	0
NNS Accelerator	9270	1247	9	33	54
Interconnect	55565	140445	15	0	0
In Total	**214918**	**223586**	**280**	**354**	**96**
Percent(%)	**97.59**	**49.07**	**92.95**	**29.81**	**56.25**

implemented two parallel IP cores for each specific machine learning application, integrating all FPGA-based optimizations in each IP core. For both of these accelerators, communication with the Flash controller or DDR controller is facilitated through the DMA controller as a bridge. The Flash controller converts the data retrieved from the flash chip, in the AXI-FULL format, into the AXI-STREAM format. This enables streaming delivery of CNN weight data and NNS corpus vectors to the accelerators. On the ARM side, the DMA controller's registers are configured to set the transfer's starting address, enabling data transfer from Flash to the accelerator or DDR to the accelerator. Ultimately, the ARM side initiates the entire computation process by changing the values of the accelerator's status registers and retrieves the execution results in the form of interrupts.

We employed Verilog HDL and High-Level Synthesis for the collaborative design of the CSD system. We synthesized it using Vivado 2020.2 and designed our lightweight kernel along with coroutine-based scheduling in Vitis 2020.2. As illustrated in Figure 11, the NICE prototype is implemented on real CSD hardware using Xilinx ZYNQ UltraScale+ ZU7EG, which features both ARM (PS) and FPGA (PL) resources, 8GB DDR4 DRAM, and 512GB MLC NAND flash. The CSD is connected to the host via a PCIe 3.0 x8 interface. Both KNN and CNN accelerators are connected to the DDR controller via the AXI bus, with their control registers exposed in the CSD microprocessor for convenient invocation by the firmware.

V. EVALUATION

A. Experiment Setup

Non-intrusive framework. To evaluate the ISC performance improvement brought by our framework under the optimization of hyper-address and file-aware in-CSD page data layout, we first conduct a comparative analysis between the NICE with the state-of-the-art work optimizing the ISC

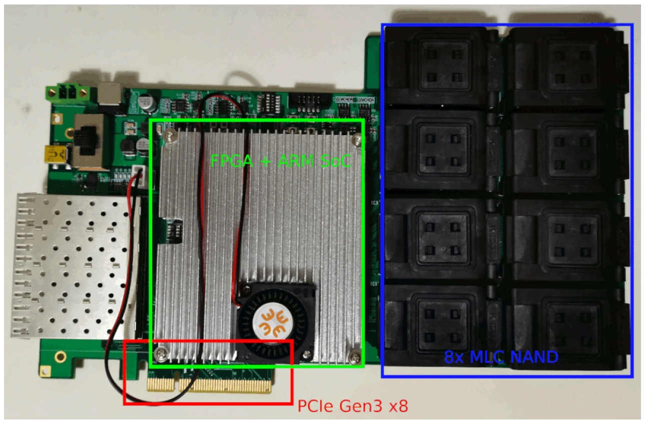


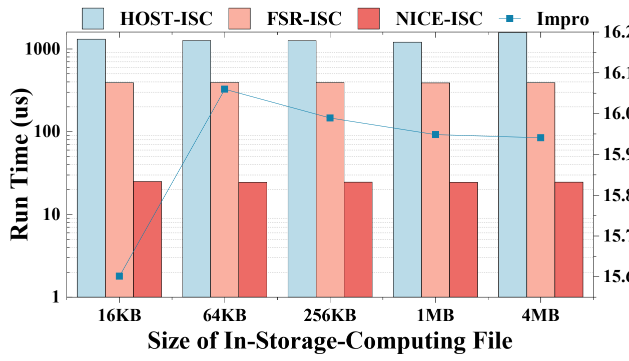
Fig. 11. The real CSD hardware.

path [21], and the widely-used data path based on the host kernel I/O stack [13]. We execute the same real-world NDP application as in [21] on NICE and set two baselines: 1) ISC path based on the host kernel I/O stack, referred to **HOST-ISC**; 2) ISC path based on FSR proposed in [21], referred to **FSR-ISC**. Given that we are focusing on evaluating the ISC path, we normalize the diverse computing performance of different processors and concentrate solely on comparing the ISC path performance in our experimental analysis. Additionally, we carefully considered the performance impact of the FTL cache in our evaluations.

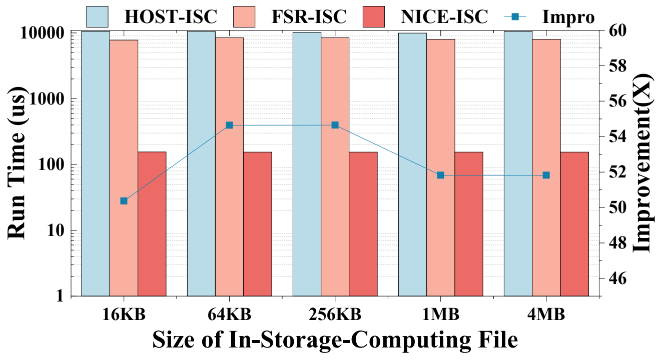
Our experiments are divided into two groups. The first group stores the first page of the requested file in the FTL cache so that the CSD can directly read the inode number corresponding to this file in memory. By leveraging the file system metadata, the CSD can calculate the inode LPN and retrieve the file's data chunks accordingly. The other group does not have the first page of the requested file in the FTL cache. As a result, the logical address of the file's first page needs to be converted into a physical address through the page table, then we can retrieve the inode number stored in the NAND flash page's OOB area, and finally obtain the file data chunks. The EXT-4 file system is used and the path depth of the file is set to 20.

To exhibit the non-intrusive characteristic of our NICE framework on normal I/O performance, we conduct a test under a high load of both regular read requests and ISC requests. The host is continuously issuing ISC commands to the CSD based on various IOPS, while the SSD benchmark is also running to measure the normal I/O performance. The benchmark has two modes: 8 threads with 8 queues (8T8Q), and 16 threads with 16 queues (16T16Q). The I/O size of each read is set to 64 blocks (equivalent to 256KB). The test involves reading 615MB of data, and the final result is recorded as the ordinary read I/O performance (MB/s).

In-Stroage-NNS accelerator. To evaluate the performance of the NNS algorithm in the NICE framework, we initially examine the efficiency of our FPGA-based KNN accelerator (**NICE-K**). We compare the performance of NICE-K with a cutting-edge embedded computing platform NVIDIA Jetson NX (refer to **NX** in the following), equipped with a 6-core NVIDIA Carmel ARM CPU, a 384-core NVIDIA Volta GPU, 8GB DDR4 DRAM, and 128GB NVMe SSD. By employing Facebook's open-source similarity search library Faiss [35]

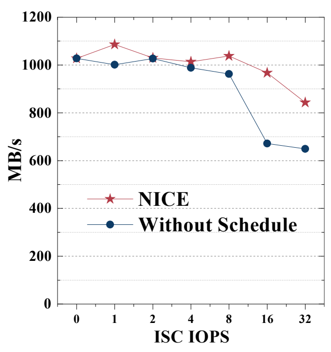


(a) FTL Cache hit

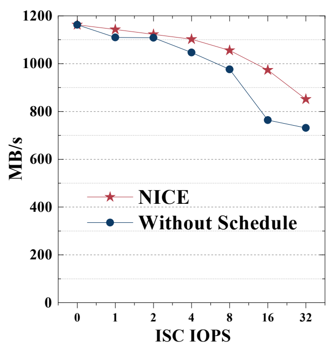


(b) FTL Cache miss

Fig. 12. The file locating performance comparison between NICE and the FSR framework [21]. (File's path depth is set to 20)



(a) 8T8Q



(b) 16T16Q

Fig. 13. The performance of NICE's coroutine-based scheduling.

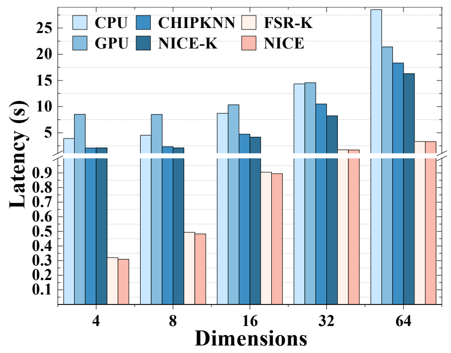
which supports both CPU and GPU on the NX platform, we maximize the throughput by simultaneously utilizing CPU and GPU in the NX platform. Furthermore, we contrast the performance of NICE-K with the state-of-the-art KNN accelerator CHIP-KNN [30]. More importantly, we also assess the performance disparities between NNS implementations based on ISC architecture and traditional architecture.

In summary, we set six distinct experimental cases: 1) **CPU-Faiss**, executing the CPU version of Faiss on the NX platform; 2) **GPU-Faiss**, operating the GPU version of Faiss on the NX platform; 3) the state-of-the-art FPGA-based accelerator **CHIP-KNN**; 4) our FPGA-only accelerator, referred to **NICE-K**, without utilizing any ISC framework; 5) NICE-K with the FSR ISC framework, labeled as the **FSR-K**; 6) NICE-K with our NICE's ISC framework, shown as **NICE**. The experimental parameters are: search space size ranged from 400 million to 6.4 billion; dimensions (D) ranged from 4 to 64; datatypes are 16-bit and 32-bit; top-k (K) is set to 128.

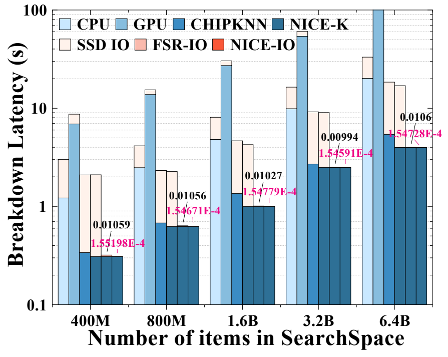
In-Stroage-CNN accelerator. For the In-Storage CNN accelerator, referred to **NICE-C**, we also conduct a set of evaluations comparing NICE-C with the embedded computing platform NVIDIA Jetson NX and another ultra-low-power CNN accelerator [27]. Due to the strict power constraints of our CSD system, we do not select state-of-the-art CNN accelerators that focus solely on inference performance without considering energy consumption. Instead, we select a low-power CNN accelerator aligning with the power requirements of our in-storage accelerator. In our evaluation, we choose two representative embedded neural networks, MobileNetV2 (α=1.0, β=1.0) and LeNet, with the ImageNet dataset (each image with 224×224×3 bytes) and the MNIST-10 dataset. Based on the different performance modes of the NVIDIA Jetson NX, we construct six scenarios: 1) **6-core ARM@1.4GHz**; 2) **4-core ARM@1.9 GHz**; 3) **384-core GPU@1.1GHz**; 4) the ultra-low-power CNN accelerator **DeepDive**; 5) NICE-C using the FSR framework as **FSR-C**, and 6) NICE-C using the **NICE** framework. We evaluate all six conditions on inference latency, I/O time, and throughput for the two CNN models at different batch sizes. We also analyze the power consumption of NICE and demonstrate its energy efficiency.

B. *NICE Framework Performance Analysis*

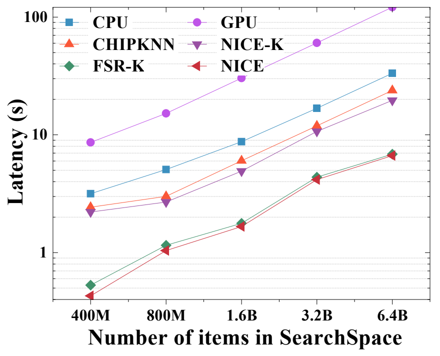
Figure 12 illustrates the execution time of the HOST-ISC, the FSR-ISC, and our NICE-ISC approach under different cache behaviors and various sizes of the requested file. Experimental results show that regardless of the FTL cache behavior, our NICE-ISC outperforms both the FSR-ISC and HOST-ISC methods across all workloads. Compared to the HOST-ISC approach, NICE uses hyper-addressing to initiate ISC commands, which prevents the user program from trapping into the kernel file system. The mapping table design also avoids the traversal of the whole file path, significantly saving data movement overhead. In comparison to the FSR-ISC approach, regarding FTL cache hit, as shown in Figure 12(a), NICE exhibits an average latency reduction of 15.6x-16x across different file sizes. This improvement is attributed to NICE's in-CSD file-aware page data layout and hyper-address-based file semantic retrieval method. We place the file inode number in the OOB area of the file's first physical page, during the file retrieving, we only need to read the file's first physical page and then locate the inode blocks based on the file system's metadata and inode number, thereby obtaining the file's data chunks. This eliminates the need to traverse 20 directories starting from the root directory based on the file name, as in the case of FSR. Therefore, the time it takes for our system to retrieve a file only includes the time to *look up the hash table* + the time of *the IO stack involved in the command* + the time to *read the first page and inode page* based on the hyper-address and the file system semantics + the time of *the inode interpreter parsing*. As shown in Figure 12(b), during FTL cache miss, our NICE-ISC further reduces access latency by 50.3x-54.6x compared to FSR-ISC. This is due to the inefficiency of flash reads, which may reach 20 times in the worst case of a 20-depth file path in FSR-ISC. By contrast, NICE-ISC only needs a maximum of two flash reads to determine the file's data chunks. Thus, NICE



(a) Single-query latency comparison with N = 400M, K = 128, DataType = 16bit

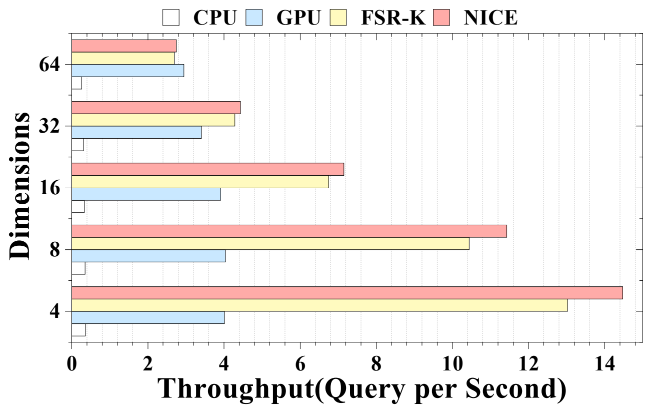


(b) Single-query latency breakdown with D = 4, K = 128, DataType = 16bit

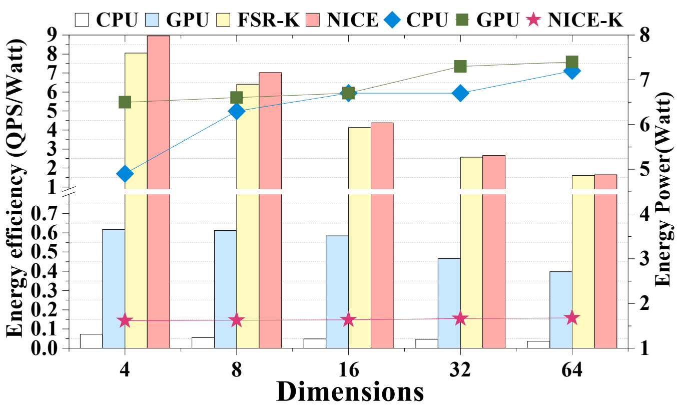


(c) Single-query latency comparison with D = 4, K = 128, DataType = 32bit

Fig. 14. The comparison of KNN performance for NVIDIA Jetson NX, FSR [21], and NICE.



(a) Throughput comparison among different architectures with N=0.4B, K=128, DataType=16bit.



(b) Comparison of energy efficiency among different architectures with N=0.4B, K=128, DataType=16bit.

Fig. 15. The comparison of KNN energy efficiency for NVIDIA Jetson NX, FSR [21], and NICE.

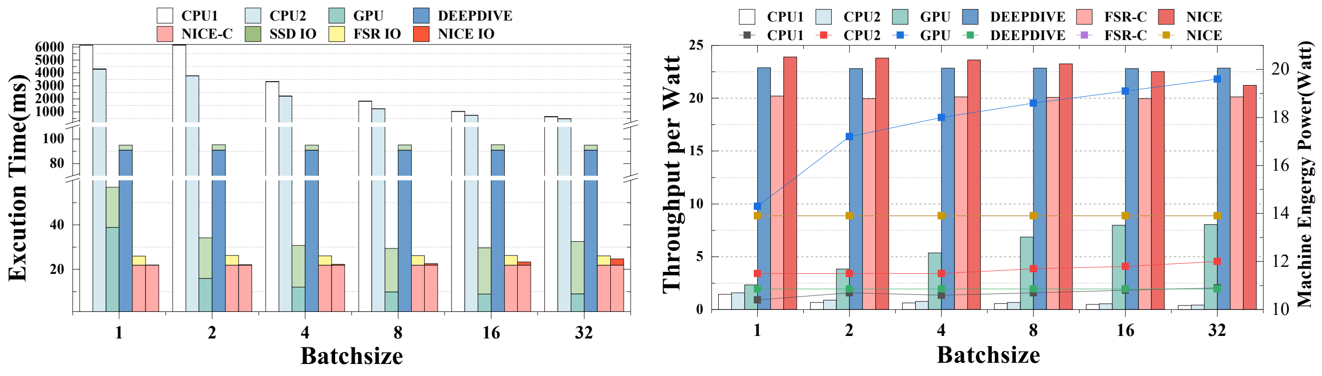
provides high-performance ISC command execution efficiency and does not suffer from performance degradation as the file depth increases.

The impact on regular I/O performance under excessive loads is shown in Figure 13. Figure 13(a) illustrates the effect of using NICE's coroutine-based scheduling vs. without scheduling under different ISC request loads (8T8Q mode). It can be observed that with coroutine-based scheduling, NICE has a lower impact on read performance, even with heavy workloads.The results for the 16T16Q mode are also similar. With coroutine-based scheduling, the read performance is improved. Note that there is no free lunch in the world, thus the scheduling mechanism has slightly influenced the ISC task performance. However, the ISC task performance reduction is also minimal (19.5ms vs. 20.78ms), which is only around 5%. In summary, NICE's coroutine-based scheduling mechanism effectively strikes a balance between normal I/O performance and ISC task latency.

C. In-Storage-NNS Performance Analysis

1) Latency: We compare the latency of a single KNN query between NICE and five other competitors under different experimental setups, as depicted in Figure 14. Firstly, Figure 14(a) illustrates the retrieval time of a single vector when the number of embeddings in the corpus is 0.4 billion, and the vector dimensions range from 4 to 64, with a 16-bit datatype. It can be observed that regardless of the dimensions, our NICE query performance outperforms all other competitors. Compared to the CPU on the NX platform, benefiting from our NICE framework and built-in FPGA accelerator, the query latency is reduced by 8.3x-12.6x. Regarding the NX platform's GPU, due to a single query's inability to fully leverage the GPU's parallel computing performance (throughput will be discussed in the next subsection), NICE's pipeline-based FPGA design results in a 6.4x-27.4x latency reduction. NICE also performs 4.8x-6.7x faster than the state-of-the-art NNS accelerator CHIPKNN. There are two reasons: firstly, CHIPKNN does not consider the overhead of data movement, focusing solely on accelerating the computing process, whereas we utilize ISC architecture to avoid the latency of data movement; secondly, we fully exploit the pipeline parallelism for higher top-K selection performance. Furthermore, the overall performance of our NICE framework is also improved compared with the FSR framework.

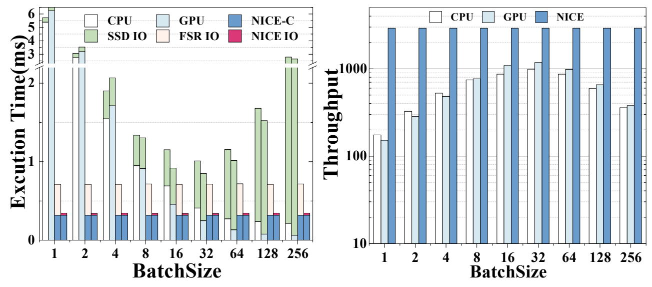
To better illustrate the performance improvement, Figure 14(b) shows the breakdown analysis of NNS query performance under various corpus sizes. There are six different combinations of accelerator logics (CPU, GPU, CHIPKNN, NICK-K) with ISC frameworks (SSD-IO, FSR-IO, NICE-IO): CPU+SSD-IO, GPU+SSD-IO, CHIP-KNN+SSD-IO, NICE-K+SSD-IO, NICK-K+FSR-IO, and NICE-K+NICE-IO. We observe two insights: firstly, our NICE-ISC framework shows significant performance improvements compared to the FSR-ISC framework and traditional (SSD-IO) architectures. Sec-



(a) Latency comparison

(b) Throughput comparison

Fig. 16. The comparison of MobileNet's performance and energy efficiency for NVIDIA Jetson NX, FSR [21], and NICE.



(a) Latency comparison

(b) Throughput comparison

Fig. 17. The comparison of LeNet's performance.

ondly, even without using the ISC framework, our NNS accelerator (NICK-K) performs better than other FPGA accelerators. Finally, Figure 14(c) demonstrates the performance under a 32-bit data type.

2) Throughput: In terms of throughput, we exhibit the performance differences between NICE, Faiss CPU and GPU implementations, and accelerators in the FSR framework (FSR-K) across various dimensions in Figure 15(a). All competitors utilize batched queries to achieve maximum throughput. At low dimensions (4-32), our NICE's throughput matches or even exceeds that of the GPU, attributed to our accelerator's optimization of throughput through parallel distance calculation and pipeline-based top-K selector. However, due to limitations in FPGA resources and AXI bandwidth, our accelerator experiences constraints on pipeline parallelism and the number of top-k modules when operating in high dimensions (D=64). This results in a significant decrease in throughput, causing the QPS (Queries Per Second) to be lower than that of GPUs. Moreover, our NICE framework still achieves an average of 1.11x higher performance than that using the FSR framework. Additionally, the CPU's throughput is consistently the lowest across all dimensions.

3) Energy Efficiency: Figure 15(b) illustrates the power consumption of different architectures across various dimensions. It can be observed that although GPUs achieve higher throughput than NICE-K when dealing with high-dimensional embedding vectors, their power consumption is also significantly higher than that of NICE-K. Since our NICE-K is integrated into the SSD, it must strictly adhere to power consumption constraints. Ultimately, in terms of energy efficiency (QPS/Watt), NICE consistently outperforms the GPU with an improvement ranging from 4.12x to 14.5x.

D. In-Storage-CNN Performance Analysis

1) Latency: Figure 16(a) illustrates the average execution time of MobileNetV2 inference obtained by our NICE and the five competitors under different batch sizes. It can be observed that regardless of the batch size, our NICE achieves lower inference latency compared to the other solutions. In the case of a batch size of 1, our NICE-C accelerator achieves lower latency than the GPU implementation, which is mainly due to maximally exploring the FPGA parallelism with pipelined executions. However, at batch sizes 2, 4, 8, 16, and 32, the parallelism of the GPU is fully utilized, leading to a slightly higher computation time for NICE-C due to limited FPGA resources allocated to the accelerator. However, the data transfer latency of the GPU dominates the overall latency at this point. Benefiting from our ISC framework, which eliminates the raw image data movement from SSD to GPU, the overall inference latency of NICE is faster than that of the GPU, despite the longer computation time. Since our NICE only needs to return the inference results to the host, its I/O time can be neglected compared to the GPU and CPU. Overall, NICE has 1.56x faster inference latency on average than the GPU. Furthermore, compared to the FSR-ISC framework, NICE also outperforms by 15%. Things are getting much better when a more lightweight CNN model is adopted, as Figure 17(a) shows, in which the raw image data transmission occupies more than 90% of the overall latency under the CPU/GPU solution when the batch size reaches 256. Thus, at all batch sizes, NICE outperforms the other solutions since they are hindered by SSD I/O latency.

2) Throughput and Energy Efficiency: Finally, we analyze the power consumption of CNN inference and the energy efficiency (Throughput/Watt, TPW) of NICE and five competitors under various batch sizes. As shown in Figure 16(b), the TPW of NICE is significantly higher than that of CPU and GPU solutions, even comparable to DEEPDIVE. Although in some batch sizes, the throughput of the NICE accelerator is slightly lower than that of the GPU, its power consumption is also significantly lower than that of the GPU, resulting in higher energy efficiency compared to GPU-based solutions (4.94x on average). Compared to the CPU, due to the significant improvement in throughput provided by our built-in accelerators, the energy efficiency is increased by 33.17x, making the slightly higher power consumption negligible. For the FSR-

ISC framework, NICE's energy efficiency is on average 14% higher than it. For LeNet, the experimental results are even better. The inference performance is 2x higher than that based on FSR-ISC and 2.4x-19x better than the GPU-based solution.

VI. Conclusion

In this paper, we introduce a non-intrusive in-storage-computing framework for embedded machine learning applications. With an easy-to-use hyper-addressing-based method and an in-CSD file-aware page data layout, programmers can easily perform ISC tasks without any modification to the existing host system. Inside the CSD firmware, our coroutine-based scheduling mechanism also helps reduce the impact on the normal I/O performance. We also introduce several FPGA-based optimizations to fully exploit both computation- and task-level parallelism. Experimental results show that we can achieve significant performance improvement and better energy efficiency than the state-of-the-art embedded platform, as well as less overhead than the state-of-the-art ISC framework.

Acknowledgments

The work described in this paper is supported in part by NSFC (62122056, 62102263, and U23B2040), in part by Guangdong Basic and Applied Basic Research Foundation (2022A1515010180), in part by Shenzhen Science and Technology Program (RCJC20221008092725019, JCYJ20210324094208024, and 20220810144025001), in part by CCF-Tencent Rhino-Bird Open Research Fund (RAGR20230126), in part by Guangdong Province Key Laboratory of Popular High Performance Computers (2017B030 314073), in part by Guangdong Province Engineering Center of China-made High Performance Data Computing System and Shenzhen Key Laboratory of Service Computing and Applications, in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics (2022WNLOKF018), and in part by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14219422, GRF 14202123).

References

[1] Y. Song *et al.*, "Dancing along battery: Enabling transformer with run-time reconfigurability on mobile devices," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1003–1008.

[2] W. Jiang, L. Yang *et al.*, "Hardware/software co-exploration of neural architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4805–4815, 2020.

[3] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "Didacache: an integration of device and application for flash-based key-value caching," *ACM Transactions on Storage (TOS)*, vol. 14, no. 3, pp. 1–32, 2018.

[4] Z. Shen, F. Chen, G. Yadgar *et al.*, "One size never fits all: A flexible storage interface for ssds," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 293–303.

[5] Q. Wei, Y. Li, Z. Jia, M. Zhao, Z. Shen *et al.*, "Reinforcement learning-assisted management for convertible ssds," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[6] W. Niu, X. Ma *et al.*, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," in *Proceedings of the 25th International Conference on ASPLOS*, 2020, pp. 907–922.

[7] Y. Li, G. Yuan *et al.*, "Efficientformer: Vision transformers at mobilenet speed," *Advances in Neural Information Processing Systems*, 2022.

[8] J. An, E. Aliaj, and S.-W. Jun, "Precog: Near-storage accelerator for heterogeneous cnn inference," in *2023 IEEE 34th International Conference on ASAP*. IEEE, 2023, pp. 45–52.

[9] Y. Lee, J. Chung, and M. Rhu, "Smartsage: Training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th ISCA*, 2022, pp. 932–945.

[10] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive {SSD}: A deep learning engine for {In-Storage} data retrieval," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 395–410.

[11] T. Vincon *et al.*, "nkv: near-data processing with kv-stores on native computational storage," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020, pp. 1–11.

[12] M. Wilkening, U. Gupta, S. Hsia *et al.*, "Recssd: near data processing for solid state drive based recommendation inference," in *Proceedings of the 26th ACM International Conference on ASPLOS*, 2021, pp. 717–729.

[13] X. Sun, H. Wan, Q. Li, C.-L. Yang, T.-W. Kuo *et al.*, "Rm-ssd: In-storage computing for large-scale recommendation inference," in *2022 IEEE International Symposium on HPCA*. IEEE, 2022, pp. 1056–1070.

[14] C.-Z. Zheng and C.-H. Wu, "A hybrid computational storage architecture to accelerate cnn training," in *2020 ICS*. IEEE, 2020, pp. 203–208.

[15] J. Do, V. C. Ferreira, H. Bobarshad *et al.*, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Transactions on Storage (TOS)*, vol. 16, no. 4, pp. 1–37, 2020.

[16] J. Xue, R. Chen, and Z. Shao, "Softssd: Software-defined ssd development platform for rapid flash firmware prototyping," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022.

[17] D. Gouk, M. Kwon, H. Bae, and M. Jung, "Containerized in-storage processing model and hardware acceleration for fully-flexible computational ssds," *IEEE Computer Architecture Letters*, 2023.

[18] I. F. Adams, J. Keys, and M. P. Mesnier, "Respecting the block interface–computational storage using virtual objects," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[19] D. Park *et al.*, "In-storage computing for hadoop mapreduce framework: Challenges and possibilities," *IEEE Transactions on Computers*, 2016.

[20] S. Liang *et al.*, "Vstore: in-storage graph-based vector search accelerator," in *Proceedings of the 59th ACM/IEEE DAC*, 2022, pp. 997–1002.

[21] L. Li, X. Chen, J. Li, J. Wang, D. Liu, Y. Tan, and A. Ren, "Optimizing the performance of ndp operations by retrieving file semantics in storage," in *2023 60th ACM/IEEE DAC*. IEEE, 2023, pp. 1–6.

[22] "Ssd power consumption and how it's managed," https://sabrent.com/blogs/storage/ssd-power-consumption, 2022.

[23] T. Wang *et al.*, "Cnn acceleration with joint optimization of practical pim and gpu on embedded devices," in *2022 IEEE 40th International Conference on Computer Design (ICCD)*. IEEE, 2022, pp. 377–384.

[24] S. Yan, Z. Liu, Y. Wang, C. Zeng, Q. Liu *et al.*, "An fpga-based mobilenet accelerator considering network structure characteristics," in *2021 31st International Conference on FPL*. IEEE, 2021, pp. 17–23.

[25] J. Knapheide, B. Stabernack, and M. Kuhnke, "A high throughput mobilenetv2 fpga implementation based on a flexible architecture for depthwise separable convolution," in *2020 30th International Conference on FPL*. IEEE, 2020, pp. 277–283.

[26] C. Yang, Y. Wang, X. Wang, and L. Geng, "Wra: A 2.2-to-6.3 tops highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, 2019.

[27] M. Baharani *et al.*, "Deepdive: An integrative algorithm/architecture co-design for deep separable convolutional neural networks," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, 2021, pp. 247–252.

[28] I. Okafor *et al.*, "Fusing in-storage and near-storage acceleration of convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems*, vol. 20, no. 1, pp. 1–22, 2023.

[29] C. Zeng, L. Luo *et al.*, "{FAERY}: An {FPGA-accelerated} embedding-based retrieval system," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 841–856.

[30] A. Lu, Z. Fang, N. Farahpour, and L. Shannon, "Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas," in *2020 ICFPT*. IEEE, 2020, pp. 139–147.

[31] K. Liu, A. Lu, K. Samtani, Z. Fang, and L. Guo, "Chip-knnv2: A configurable and high-performance k nearest neighbors accelerator on hbm-based fpgas," *ACM TRETS*, vol. 16, no. 4, pp. 1–26, 2023.

[32] X. Song, T. Xie, and S. Fischer, "Accelerating knn search in high dimensional datasets on fpga by reducing external memory access," *Future Generation Computer Systems*, vol. 137, pp. 189–200, 2022.

[33] Y. Wang *et al.*, "In-storage acceleration of graph-traversal-based approximate nearest neighbor search," *arXiv preprint arXiv:2312.03141*, 2023.

[34] N. Matsumoto *et al.*, "Optimal parallel hardware k-sorter and top k-sorter, with fpga implementations," in *2015 14th International Symposium on Parallel and Distributed Computing*. IEEE, 2015, pp. 138–147.

[35] J. Johnson *et al.*, "Billion-scale similarity search with gpus," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535–547, 2019.