國立臺灣大學電機資訊學院資訊網路與多媒體研究所

碩士論文

Graduate Institute of Networking and Multimedia

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

利用儲存內計算加速向量資料庫搜尋

Accelerate Search in Vector Databases with In-Storage Computing

宋沛誠

Pei-Cheng Sung

指導教授: 楊佳玲 博士

Advisor: Chia-Lin Yang Ph.D.

中華民國 114 年 2 月

February, 2025

doi:10.6342/NTU202500592

致謝

首先，感謝指導教授楊佳玲教授的悉心指導，無論是研究方向的確立、技術問題的克服，或是論文架構的規劃，都給予了寶貴的建議，使我得以順利完成本研究。

感謝口試委員張原豪博士與鄭湘筠博士撥空參與口試，並提供寶貴建議，幫助我進一步完善論文。

特別感謝政宇學長長期投入大量時間與我討論，提供技術支援與研究建議，以及閔良學長提供的寶貴想法，幫助我拓展視野。感謝胤辰在研究與口試期間的實作支援與討論，程翔、行悌、瀚坪在論文撰寫與口試準備上的協助，以及聖傑學長初入實驗室時的幫助。此外，也要感謝所有實驗室的夥伴們，在這段時間裡的陪伴與協助，這份支持讓我能夠在研究的道路上持續前進，雖無法一一列舉，但心中充滿感激。

另外，感謝 SPFresh 團隊，尤其是 Yuming Xu，提供 SPACEV dataset 及索引相關的協助，使本研究得以順利推進。

最後，感謝我的家人始終支持與鼓勵我，讓我能夠專心投入研究並完成這篇論文。

doi:10.6342/NTU202500592

中文摘要

向量資料庫（Vector Databases, VDBs）透過向量搜尋實現高效的資訊檢索，在現代資料處理中至關重要。雖然近似最近鄰搜尋（Approximate Nearest Neighbor Search, ANNS）方法（如基於圖的和基於叢集的方法）解決了大型資料集在計算上的挑戰。然而，大量資料移動所導致的效能瓶頸，限制了向量搜尋的可擴展性和效率。透過儲存內計算 (In-storage Computing) 技術能夠有效緩解此瓶頸所造成的性能損失。然而，過去的研究缺乏存儲內計算裝置內部的量化分析，既無法直觀的說明效能提升的原因，系統中也可能存在潛在的效能瓶頸。

本研究提出了一種基於 COSMOS+ OpenSSD 平台的儲存內向量搜尋裝置設計與實現。借助實際平台的分析發現直觀的設計會導致指令管線化效能下降並針對此問題提出改善方案。我們的方法在距離計算功能的效能上相較於現有的儲存內計算解決方案提升了 1.49 倍，達到了理想效能的 88%。此外，我們的方案在端到端吞吐量上，與儲存內計算方法相比，提升了 1.42 ~1.44 倍; 與使用傳統 SSD 方法相比，提升了 1.75 ~2.03 倍。最後，我們分析了引入儲存內計算所帶來的新的系統瓶頸，並且針對此瓶頸提出未來的研究建議。

關鍵字：儲存內計算、近數據計算、向量資料庫、近似最近鄰搜尋、場效可程式化邏輯閘陣列

doi:10.6342/NTU202500592

英文摘要

Vector Databases (VDBs) enable efficient information retrieval through vector search, crucial in modern data processing. While Approximate Nearest Neighbor Search (ANNS) methods address computational challenges, data movement bottlenecks limit the scalability and efficiency of vector search. Prior In-storage computing (ISC) reserch offers a solution but lacks quantitative analysis on real devices, making it difficult to explain performance gains and identify potential bottlenecks.

This study presents the design and implementation of an in-storage vector search device based on the COSMOS+ OpenSSD platform. Through real-device analysis, we identify a key bottleneck where a naive PE design leads to pipeline inefficiencies between commands. To address this, we propose an optimized approach that improves distance computation performance by 1.49x, achieving 88% of the theoretical peak performance. Additionally, our design enhances end-to-end throughput by 1.42x ~1.44x compared to existing ISC solutions and by 1.75x ~2.03x over traditional SSD-based methods. Finally,

doi:10.6342/NTU202500592

we analyze new system bottlenecks introduced by ISC and provide insights for future research directions.

Keywords: In-Storage Computing, Near Data Processing, Vector Database, Approximate Nearest Neighbor Search, FPGA

doi:10.6342/NTU202500592

目次

doi:10.6342/NTU202500592

doi:10.6342/NTU202500592

doi:10.6342/NTU202500592

圖次

doi:10.6342/NTU202500592

doi:10.6342/NTU202500592

表次

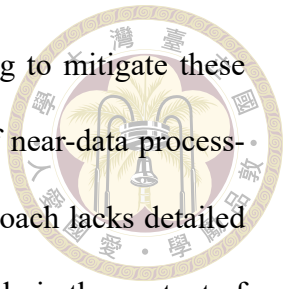
doi:10.6342/NTU202500592

第一章　Introduction

Vector databases (VDBs) have become essential in modern data processing, leveraging embedding models to transform diverse types of data into vector representations. Within such vector spaces, data points with higher similarity are positioned closer together, enabling efficient solutions to complex information retrieval problems through nearest vector search(1). These databases are widely applied in domains such as Retrieval-Augmented Generation (RAG)(2) and recommendation systems(3), where their ability to handle high-throughput, low-latency workloads is critical for meeting stringent service level agreements (SLAs).

Despite their growing prominence, VDBs face significant challenges. The large data volumes, often exceeding hundreds of gigabytes, render exact search computationally infeasible. Consequently, approximate nearest neighbor search (ANNS) methods, including graph-based and cluster-based approaches, have emerged as the preferred techniques. Among these, cluster-based methods(4; 5) offer superior SSD access patterns compared to graph-based methods(6–9) when achieving the same recall rate, making them more suitable for storage-efficient implementations. However, the substantial data movement required for these methods constitutes a major performance bottleneck, limiting the scalability and efficiency of vector search.

doi:10.6342/NTU202500592



Existing research has proposed leveraging in-storage computing to mitigate these bottlenecks. Notably, the Pyramid(10) demonstrated the potential of near-data processing to accelerate cluster-based vector search. Nevertheless, this approach lacks detailed design, implementation specifics, and quantitative analysis, particularly in the context of in-storage computing devices. Making it difficult to intuitively explain performance improvements and potentially overlooking hidden system bottlenecks.

In this work, we address these gaps by presenting a comprehensive design and implementation of an in-storage vector search device on the COSMOS+ OpenSSD platform. Through real-platform experiments, we analyze the workloads, identify pipeline inefficiencies caused by a naive processing element design between commands, and propose enhancements to optimize the processing element design. These enhancements result in a 1.49x improvement in throughput for the distance calculation primitive compare to SoTA ISC solution, and achieving 88% of the ideal performance. Furthermore, we demonstrate a 1.42x ~1.44x improvement in end-to-end throughput comparing to SoTA ISC solution, and when compared to conventional approaches, our solution achieves a 1.75x ~2.03x gain in overall performance. Additionally, this study identifies new performance bottlenecks and offers insights into future optimization opportunities, paving the way for further advancements in the field.

doi:10.6342/NTU202500592

第二章　Background & Motivation

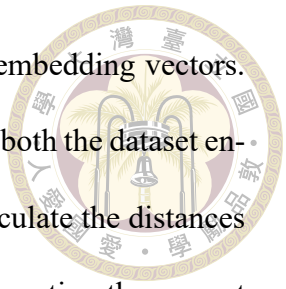
2.1　Background

In this section, we first introduce the concept of vector search and explore related work. Next, we discuss the architecture of SSD and one of the open source platform: COSMOS+ OpenSSD(11). Finally, we review the architecture of in-storage computing and its applications.

2.1.1　Vector database

In the era of big data, the volume of available data is growing exponentially and is projected to exceed 160 zettabytes by 2025 (12). Approximately 90% of the new data consists of unstructured or semi-structured formats, such as text, images, graph, etc. (13). One of the most common approaches to handling such data today is through the use of vector embeddings. Vector embedding models map unstructured data to feature vectors (14; 15), ensuring that similar original data points, such as text with similar semantics, are represented by vectors that are closer to each other in the embedding space. For instance, the embedding vectors of ”cat” and ”dog” are closer to each other than those of ”cat” and ”apple,” reflecting their semantic similarity.

doi:10.6342/NTU202500592



We can retrieve information by using the characteristics of the embedding vectors. Specifically, to retrieve data from a dataset using a query, we first map both the dataset entries and the query to the same embedding vector space. Next, we calculate the distances between the query vector and the dataset vectors. By identifying and sorting the nearest vectors to the query vector, the corresponding data of the closest vectors are considered to be the most relevant to the query; in other words, the data to be retrieved. The process of identifying the nearest vectors in a dataset is commonly referred to as k-nearest neighbor search (k-NNS), where k represents the number of nearest vectors to be found. The databases that implement this method are known as vector databases.

One of the issues in performing searches in vector databases is the huge computing overhead due not only to the large dataset, but also to the high dimension of the embedding vectors, ranging from 10 to 1000 dimensions(16). To avoid huge searching overhead, current vector databases choose not to search for the exact nearest neighbors but instead for approximate nearest neighbors. K-approximate nearest neighbor search (k-ANNS) methods require offline index construction, where each search leverages the index to guide the process and find an approximate result. K-ANNS methods targeting large-scale, high-dimensional data can primarily be categorized into two types based on their index structures: graph-based indexes (6–8) and cluster-based indexes(4; 5).

A graph-based index represents the dataset as a graph, where each vector is a vertex connected to others based on distance-related rules. Searching starts from an entry point, typically the centroid or a central vector, and iteratively selects the closest outgoing neighbor to the query vector. The process continues until no closer neighbor is found(17). A nearest neighbor buffer is maintained to track the closest k vectors during the search, with the final buffer containing the query result.

doi:10.6342/NTU202500592

The main concept of a cluster-based index is to partition the dataset into several clusters. Searching in a cluster-based index involves two steps. First, identify several clusters that are closest to the query vector. Second, conduct an exact search within the vectors of these clusters to determine and rank the closest k vectors.

We compare these methods in terms of computational cost, access patterns, and specific challenges, highlighting their trade-offs to provide a clearer understanding of their applicability. Graph-based methods excel in computational cost, as the number of hops during the search process is typically low, averaging less than 7 (8), resulting in lower computational overhead compared to cluster-based methods. However, graph-based methods exhibit poor performance in terms of access patterns. For access patterns, accessing vector and edge data from outgoing neighbors leads to frequent small random accesses. This inefficient access pattern can significantly degrade search performance, especially when the index is stored on external storage devices like SSDs. There are two main approaches in previous works that address this issue. The first approach focuses on caching small data, such as vectors. For example, DiskANN(8) quantized the vectors using Product Quantization (PQ) (18) and fully cached the quantized data in host DRAM. The second approach aims to improve the location of the data in external storage. An example is DiskANN++(9), which packs data from nearby vertices, those more likely to be accessed simultaneously, together to enhance the locality of storage access. On the other hand, while cluster-based methods excel in access patterns, they suffer from higher computational overhead. For access patterns, since distance calculations are performed at the cluster level, cluster-based methods generate largely sequential access patterns. In contrast to the small and random access patterns of graph-based methods, this spatially localized access pattern is particularly effective in reducing read/write amplification issues when accessing external

doi:10.6342/NTU202500592

storage. However, computational costs are higher in cluster-based methods because all vectors within the retrieved clusters must be processed. Cluster-based methods face a particular dilemma known as the boundary issue (4; 7).Figure 2.1 displays the problem, where the green dots represent the cluster closest to the query vector, yet the actual nearest neighbor is part of the blue dots' cluster. Consequently, if the algorithm only seeks the nearest cluster, it risks overlooking the nearest neighbor. Therefore, to improve search accuracy, cluster-based methods are often forced to include adjacent clusters in the search. This significantly increases computational costs, especially as the number of adjacent clusters grows with the dimensionality of the data. To address this problem, SPANN (4) mitigates the boundary issue by assigning vectors near cluster edges to multiple overlapping clusters and dynamically selecting which clusters to search. This approach reduces search latency while maintaining high accuracy.

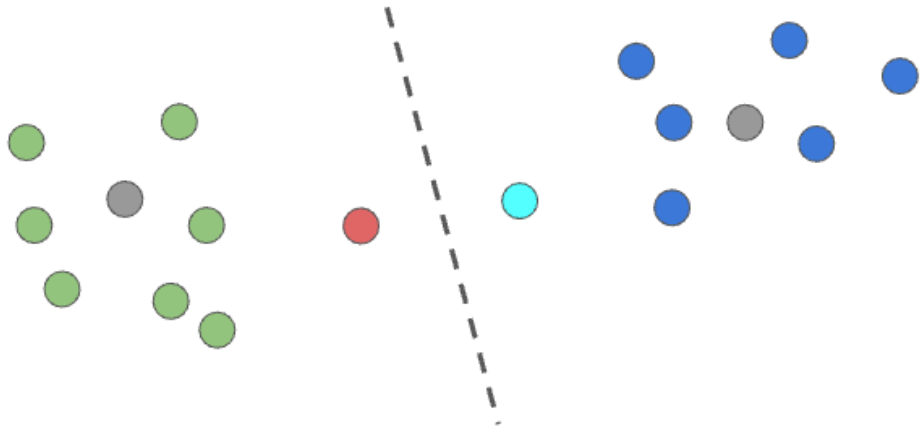


Figure 2.1: Example of the boundary issue. The red dot represents the query vector. The cyan dot indicates the nearest neighbor to this query vector. Meanwhile, blue and green dots signify vectors that belong to two separate clusters. Gray dots mark the centroids of these clusters, and the dashed line delineates the boundary between them. Despite the nearest cluster to the query vector being the green dots' cluster, the closest neighbor resides within the blue dots' cluster.

doi:10.6342/NTU202500592

2.1.2 SSD Architecture & Open Source SSD Platform

2.1.2.1 SSD Architecture

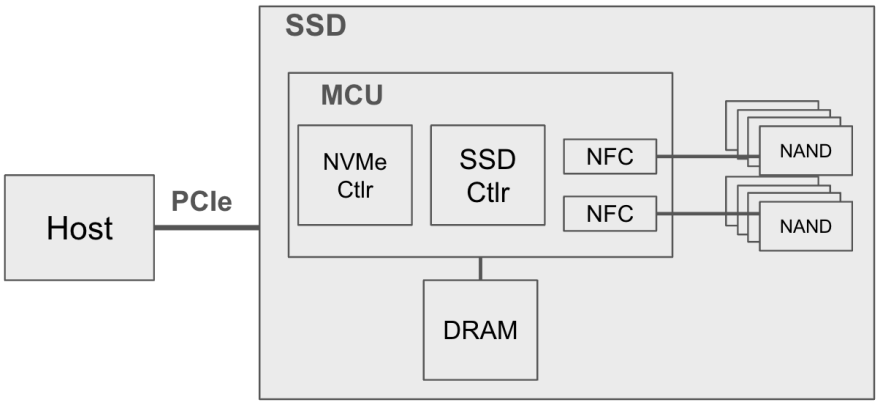


Figure 2.2: SSD Architecture

The architecture of a Solid-State Drive (SSD), as shown in Figure 2.2 consists of several critical components that enable fast and efficient data storage and retrieval. Unlike traditional hard disk drives (HDDs), SSDs use NAND Flash memory, a type of non-volatile memory. This allows SSDs to offer higher performance, and lower power consumption compared to HDDs. Key components of an SSD include the Host Interface Controller, the SSD Controller, SSD DRAM, and the NAND Flash Controller (NFC) along with the NAND Flash itself(11). The NAND Flash serves as the primary storage medium, while the other components work together to manage data flow, optimize performance, and ensure efficient operation.

The Host Interface Controller serves as the communication bridge between the host system and the SSD. In modern SSDs, this is typically achieved using the NVMe (Non-Volatile Memory Express) protocol(19), which is known for its low-latency and high-bandwidth characteristics. Operating over the PCIe (Peripheral Component Interconnect Express) bus, NVMe ensures high-speed data transfer and efficient command manage-

doi:10.6342/NTU202500592

ment.

At the heart of the SSD lies the SSD Controller, which is responsible for managing data flow between the host and the NAND Flash. Among its various functions, the Flash Translation Layer (FTL) is the most critical. The FTL maps logical addresses used by the host to physical addresses on the NAND Flash memory, providing an essential abstraction to address the limitations of NAND Flash, such as its inability to overwrite data directly.

To address this limitation, the controller performs Garbage Collection, which reclaims storage space by consolidating valid data and removing invalid blocks. When data on a NAND page becomes invalid, the page cannot be rewritten directly and must first be erased. This is achieved through the NAND erase command, which clears entire blocks of pages, making them available for new data writes. However, the erase operation comes at a cost, as it contributes to the wear of NAND Flash memory. NAND Flash has a limited number of program/erase (P/E) cycles, and frequent erasing accelerates the degradation of NAND blocks, ultimately reducing the SSD's lifespan.

To mitigate this issue, the controller employs Wear Leveling, which evenly distributes write and erase operations across all NAND blocks. By ensuring that no single block is excessively used, Wear Leveling minimizes the wear caused by repeated erasures, extending the overall lifespan of the SSD. Finally, to maximize performance, the controller implements scheduling functions that coordinate read and write operations across multiple channels and ways. By leveraging the SSD's internal parallelism, these scheduling mechanisms optimize throughput and minimize latency.

The SSD DRAM acts as a data buffer and cache that bridges the performance gap between the host interface and the NAND Flash memory. It temporarily stores write data

doi:10.6342/NTU202500592

to hide the high latency of NAND write operations and accelerates read data delivery. Additionally, the DRAM buffer supports internal housekeeping tasks, such as garbage collection.

Finally, the NAND Flash Controller (NFC) and the NAND Flash complete the architecture. The NFC is responsible for managing communication between the SSD Controller and the NAND Flash, ensuring efficient command execution and data transfer. It also performs Error Correction Code (ECC) operations to detect and correct bit errors, ensuring the integrity of data stored in the NAND Flash(20; 21). NAND Flash is organized into channels, each serving as an independent data path that can operate in parallel to increase data throughput. Within each channel, multiple ways (or NAND dies) are connected, enabling interleaved access across devices. This interleaving mechanism is designed to hide the long NAND read latency by overlapping data access operations across multiple dies, thereby improving overall throughput.

2.1.2.2 Open Source SSD Platform

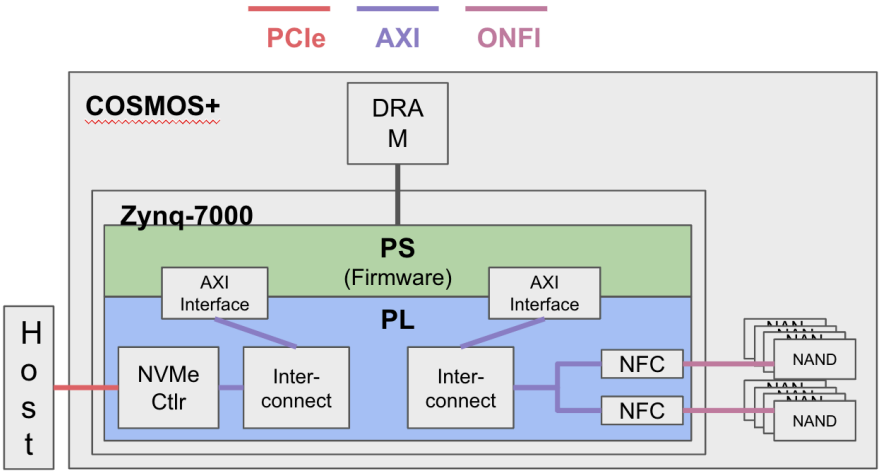


Figure 2.3: COSMOS+ Architecture

The Cosmos+ OpenSSD(11) platform is an open-source development platform for

doi:10.6342/NTU202500592

SSD prototyping and research. Its architecture is illustrated in Figure 2.3. Based on the Xilinx Zynq-7000 SoC, the platform integrates a Processing System (PS) and a Programmable Logic (PL) unit, enabling flexible system design. The PS features dual-core ARM Cortex-A9 cores, which run the firmware responsible for high-level SSD operations, such as command management and scheduling. Meanwhile, the PL provides FPGA resources that implement critical SSD hardware components, such as the NVMe Controller and NAND Flash Controller. These components interact through AXI buses. This architectural integration allows the Cosmos+ platform to serve as an adaptable and powerful tool for exploring advanced SSD functionalities.

The hardware design of Cosmos+ leverages the PL to implement the NVMe Controller and the NAND Flash Controller. The NVMe Controller ensures high-speed communication with the host via the NVMe protocol over PCIe, while the NAND Flash Controller manages data read/write operations and performs tasks like error correction and erasure. The PL also allows the integration of custom hardware components, enabling research into advanced storage concepts such as in-storage computing and hardware-accelerated scheduling.

The firmware architecture of Cosmos+ managing SSD operations through a structured and efficient stack. The firmware stack includes several layers, with each layer responsible for a specific set of tasks. The NVMe Manager interprets host commands and handles administrative tasks, while the Flash Translation Layer (FTL) performs logical-to-physical address translations, garbage collection, and wear leveling. At the lowest level, the scheduler coordinates the execution of hardware-level operations.

A key feature of the Cosmos+ firmware is its structured processing of host com-

doi:10.6342/NTU202500592

mands. Commands received from the host are first parsed into fixed-size slices that align with the page-based operations of NAND Flash. These slices are further converted into hardware-level operations, such as NAND read, NAND write, and PCIe DMA, which are then queued for execution. The low-level scheduler plays a crucial role in this process by managing the prioritization and dispatch of operations.

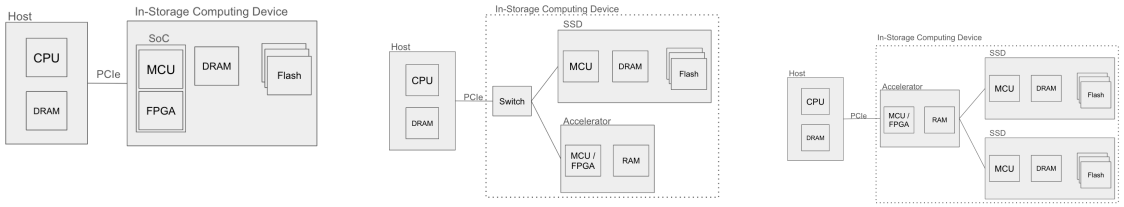
The combination of reconfigurable hardware and robust firmware makes Cosmos+ a comprehensive platform for SSD research. Its ability to support custom hardware extensions and explore new firmware algorithms positions it as a valuable tool for advancing both academic and industrial understanding of storage system design.

2.1.3 In-Storage Computing

Efficient data processing has become a significant challenge in the era of big data and machine learning, driven by two key factors: the rapid growth in data volume, as demonstrated in the previous section, and the rise of machine learning applications, which have emerged as a direct result of the availability of big data. This type of data processing often involves substantial data movement between the host and external storage devices, particularly SSDs. Examples include performing distance calculations on a cluster of vectors in vector search or multiplying large tensors in deep learning models. These data movements have become the bottleneck of the entire system, significantly impacting overall processing performance. One of the main reasons is the bandwidth mismatch between the buses within the system. For a single SSD device, the ratio between the aggregated internal bandwidth, the sum of the bandwidth of NAND flash channels, and the external bandwidth, typically the host interface such as PCIe, can range from 2x to 4x(22; 23). When scaled to an entire system with multiple SSDs, this ratio can increase dramatically, reach-

doi:10.6342/NTU202500592

ing up to 66x between the CPU's I/O interface and the combined bandwidth of all NAND flash channels (24). This is where in-storage computing, also known as computational storage, comes into play to address this issue. The core concept of in-storage computing is to move computation closer to the data, rather than transferring the data to centralized computing cores. There are three main types of in-storage computing architectures(25), as shown in Figure 2.4: Stand-alone, Bundled Together, and Interconnect. Let us discuss each type of in-storage computing device in detail.

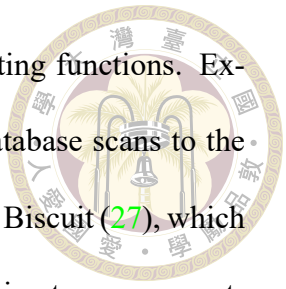


(a) Bundled Together (b) Stand-alone (c) Interconnect

Figure 2.4: Three Different Architectures of In-Storage Computing Devices

The bundled-together type (Figure 2.4a) has full control over the entire SSD system. Typically, these devices use a system-on-chip (SoC) that integrates an MCU and FPGA to replace the original SSD controller, enabling complete control over the SSD, including the SSD firmware and NAND flash controller (NFC). This design enables precise management of internal data placement, scheduling, and other related tasks. As the only type of device capable of controlling the NFCs, its primary advantage lies in the ability to fully utilize the SSD's internal bandwidth. However, with great power comes great responsibility. In addition to managing its own processing elements, this type of device also requires handling the existing SSD system components, such as the flash translation layer (FTL). Programming such devices demands significant engineering effort and cross-domain expertise. One of the most prominent platform of this type is COSMOS+(11), an NVMe-based SSD featuring an onboard Xilinx SoC. This design allows programmers to

doi:10.6342/NTU202500592



modify both firmware and hardware to implement in-storage computing functions. Examples of this architecture include YourSQL (26), which offloads database scans to the storage side to enable early filtering and reduce host-device traffic, and Biscuit (27), which utilizes a stream programming model to simplify task deployment on in-storage computing devices. However, the dataflow in the above works requires data to be fetched from NAND flash to SSD DRAM before processing, effectively doubling the traffic in and out of the SSD DRAM. ASSASIN (28) identified this dataflow cause SSD DRAM as a new bottleneck in the system and proposed the ASSASIN core. This core is attached between the NAND flash controller and SSD DRAM via interconnects and leverages the concept of computing on the dataflow to reduce traffic on the SSD DRAM.

The stand-alone type (Figure 2.4b) usually combines SSD, accelerator, and PCIe switch in a single device. The accelerator can access the SSD conventionally through the internal PCIe switch. By programming the accelerator, data access and operations are performed entirely within the device, without consuming any external system bandwidth. Although this type of device cannot fully utilize the internal NAND bandwidth of SSDs, they still have significant potential to leverage the bandwidth mismatch between the CPU's I/O interface and the combined bandwidth of all SSDs, particularly when multiple SSDs are attached to a single machine. Additionally, separating the SSD and accelerator allows programmers to treat the SSD as a black box and program the accelerator using existing standards, such as OpenCL. This approach makes programming stand-alone devices significantly easier compared to the previous type. The most prominent platform of this type is SmartSSD(29). An SmartSSD device includes an PM1733 SSD, Xilinx FPGA, and PCIe switch and can be easilly programmed in OpenCL standard. Examples of this architecture include NASCENT(30), which implements bitonic sort on an FPGA to enable

doi:10.6342/NTU202500592

in-storage sorting, significantly reducing data transfer to the host, and Smart-Infinity (31), which performs gradient updates within the SmartSSD and maintains optimizer states inside the SSD, thereby minimizing data movement.

The interconnect type (Figure 2.4c)is an accelerator equipped with multiple PCIe slots, allowing multiple SSDs to be attached simultaneously. These devices can not only process data on the dataflow but also make the underlying SSDs transparent to the host. Therefore, the greatest advantage of this type lies in the field of virtualization. For instance, BM-Store (32) demonstrates the capability to enable hardware virtualization by incorporating a Xilinx SoC-based accelerator between the host and SSDs. Without this accelerator, achieving virtualization would require OS modifications, potentially raising significant privacy concerns for cloud customers, or SSDs with specific hardware functionalities, which would increase the cost of replacing numerous SSDs.

2.2 Motivation

State-of-the-art cluster-based vector search method, SPANN(4), divides the search process into two phases: in-memory graph search and in-storage cluster search. In this approach, an in-memory graph is used to identify several clusters that are closest to the query vector, which are then fetched from the SSD into DRAM for precise searching. The graph resides entirely in DRAM, ensuring fast access, while the clusters stored in the SSD.

Figure 2.5 presents a detailed performance breakdown of SPANN when performing cluster-based vector search on the SIFT100M dataset using a commercial SSD. The results reveal that in-storage cluster search dominates the overall search time, accounting for approximately 95% of the total search latency, whereas in-memory graph search con-

doi:10.6342/NTU202500592

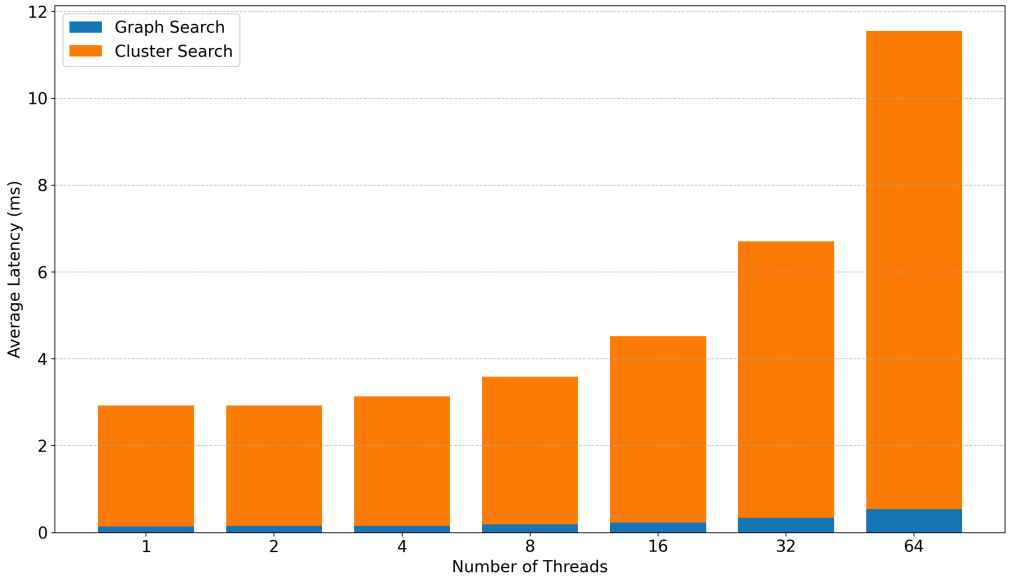


Figure 2.5: Latency breakdown of Cluster-Based Vector Search on the SIFT100M dataset using a commercial SSD. The results indicate that in-storage cluster search dominates the overall search time.

tributes only 5%. This stark contrast highlights that the in-storage cluster search phase is the primary performance bottleneck, underscoring the necessity of optimizing this stage to improve overall system efficiency.

To further analyze the system behavior, Figure 2.6 compares the performance of *distance calculations per second (DCPS)* and *SSD IOPS* under varying host software thread counts when running cluster-based vector search on a commercial SSD. The results demonstrate that as the number of host software threads increases, both DCPS and SSD IOPS eventually saturate, suggesting a strong correlation between computational workload and storage access rates. This saturation indicates a fundamental limitation in I/O bandwidth, where increased parallelism in the host software fails to translate into higher throughput due to hardware constraints.

To better understand the cause of SSD IOPS saturation, we conducted further experiments using *COSMOS+* as a conventional SSD to execute distance calculations, the primitive of in-storage cluster search. Figure 2.7 presents a breakdown of execution time

doi:10.6342/NTU202500592

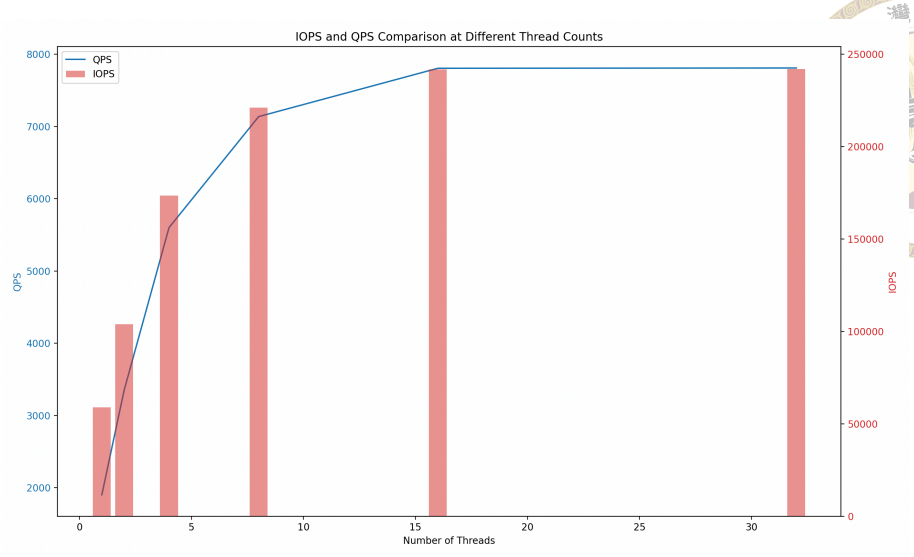


Figure 2.6: Comparison of DCPS and SSD IOPS under varying host software thread counts using a commercial SSD. Both DCPS and IOPS saturate as thread count increases, highlighting their interdependence.

across different thread counts. Notably, as the number of threads increases, the time spent on *TxDMA*, which transfers data back to host DRAM via PCIe, grows significantly. With 32 threads, the TxDMA time is observed to be **13× higher** than with just 2 threads, indicating that data transfer overhead becomes a dominant factor in performance degradation. This trend suggests that the PCIe bandwidth becomes a bottleneck, preventing further scaling of computation and leading to increased waiting times for data transfers.

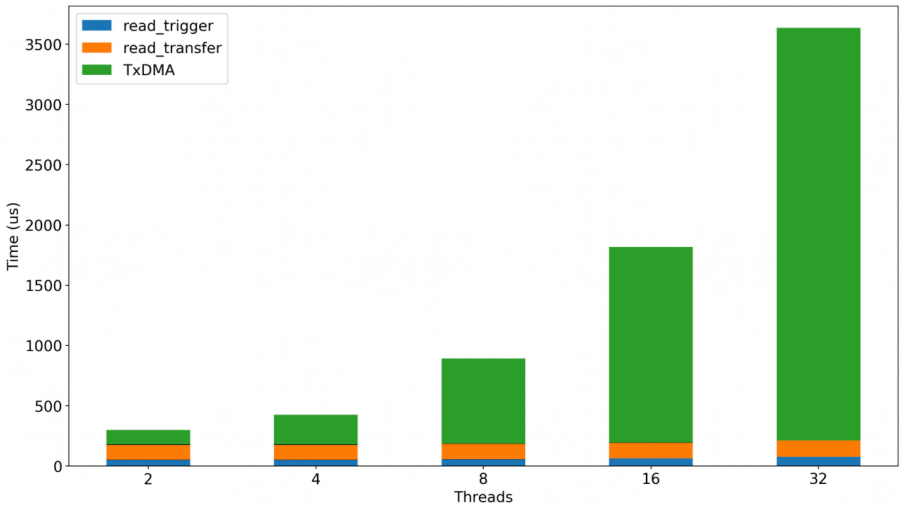


Figure 2.7: Time breakdown of distance calculation on COSMOS+ with varying thread counts. The TxDMA time increases significantly due to PCIe bandwidth limitations.

doi:10.6342/NTU202500592

Prior work such as Pyramid(10) explored the use of Process-in-Memory and In-Storage Computing to deal with the PCIe bottleneck and accelerate cluster-based vector search. While the concept of integrating computation closer to the storage layer is promising, Pyramid lacks a detailed design, implementation, and thorough analysis of its in-storage computing design. Without these critical details, the potential benefits and limitations of in-storage acceleration remain underexplored.

To address these gaps, we aim to leverage the Cosmos+ OpenSSD platform to design, implement, and thoroughly analyze an in-storage acceleration solution for cluster-based vector search. Our work will provide a comprehensive implementation, detailing both the hardware and software design aspects, and include a complete quantitative analysis of the solution's performance. Furthermore, we will propose optimizations to improve performance and efficiency. Through this effort, we aim to establish a robust and well-analyzed framework for the design and implementation of in-storage computing in cluster-based vector search systems.

doi:10.6342/NTU202500592

第三章　Implementation & Analysis

3.1　In-storage Distance Calculation

In a typical cluster search process, data is organized into clusters stored on the SSD. Each cluster comprises a list of tuples, where each tuple contains a vector identifier (VID) and the vector's raw data. The VID, a 4-byte integer, uniquely identifies each vector, while the vector's raw data size is determined by its dimensionality (D) and the scalar type used, which is typically `uint8` or `int8` in large scale dataset. For example, in the commonly used `sift1b` dataset, $D = 128$ and the data type is `uint8`, resulting in a vector size of 128 bytes.

To execute a search, the entire cluster needs to be transferred from the SSD to the host memory. This process involves moving substantial amounts of data through the PCIe interface. This data-intensive operation makes the cluster search workflow heavily reliant on I/O bandwidth, highlighting the challenges posed by PCIe bottlenecks.

Pyramid (10) proposed offloading the distance calculation to be performed within the storage system. By doing so, the data transmitted over the PCIe interface shifts from high-dimensional vectors to compact distance values. This approach dramatically reduces data transfer volume, cutting it by over 90%, thereby alleviating the PCIe bottleneck and

doi:10.6342/NTU202500592

improving system efficiency. However, Pyramid lacks detailed design and quantitative analysis, leaving critical aspects of its implementation and performance unexplored. This chapter seeks to address these gaps by presenting a practical implementation and analyzing its performance on a real hardware platform.

3.2 In-storage Distance Calculation Design

The architecture of the in-storage distance calculation device, shown in Figure 3.1, leverages a PE dedicated to performing stream distance calculation for each channel. By adding a PE to each channel, this design effectively exploits channel parallelism, allowing multiple calculations to be carried out simultaneously across channels. Furthermore, the computation is overlapped with data transfer operations, hiding the latency of calculations within the transfer process.

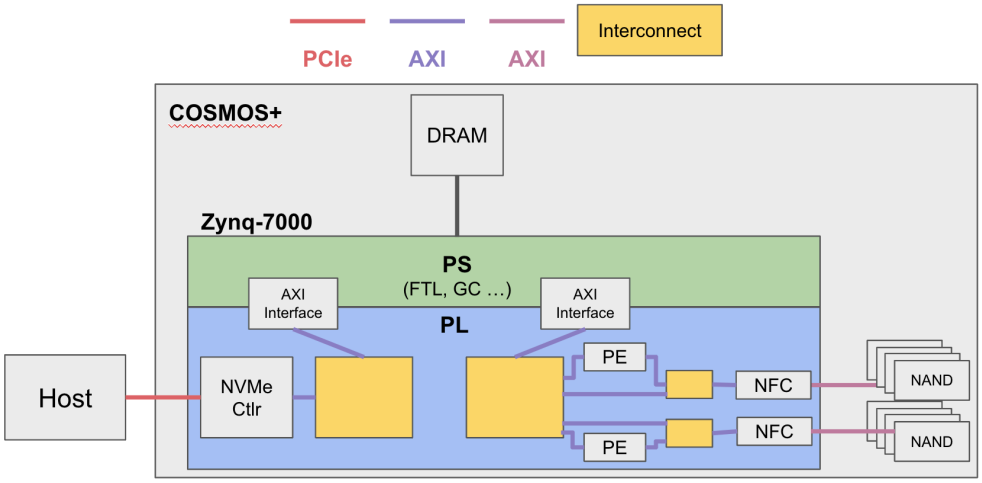


Figure 3.1: Architecture of in-storage distance calculation device

The in-storage distance calculation workflow is illustrated in Figure 3.2.

The process begins with the host storing the query vector in the data buffer and issuing a distance calculation command to the SSD via NVMe vendor command **(1)**. Upon

doi:10.6342/NTU202500592

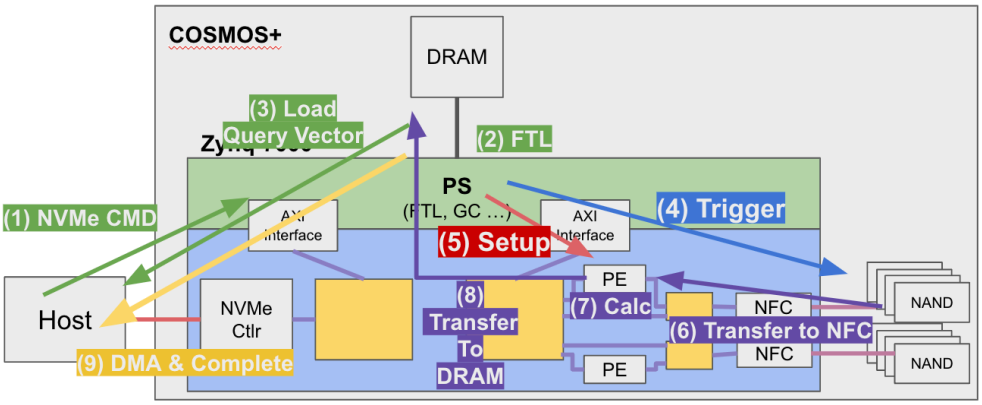


Figure 3.2: Sequence diagram of in-storage distance calculation command

receiving the command, the SSD firmware first utilizes the FTL to locate the physical address of the target cluster **(2)**. Also, the it fetches the query vector from the data buffer in the host DRAM to the SSD DRAM using RxDMA **(3)**.

Following this, the firmware initiates a standard NAND read procedure by sending a read trigger to the target NAND die through the NFC **(4)**. Once the trigger operation is completed, the firmware sets up the corresponding channel's PE via MMIO and issues a read transfer command to the NFC **(5)**. This enables the NFC to start reading and streaming data through the PE **(6)**. As the data flows through the PE **(7)**, the distance calculation is performed in real time, and the results are written back to the SSD DRAM **(8)**.

Finally, the firmware initiates a TxDMA operation to transfer the calculated distances back to the host and completes the process by issuing a NVMe complete **(9)**.

3.3 Processing Element Design

Stream Distance Calculation is the core function of the In-Storage Distance Calculation (ISDC) device. The key concept of Stream Distance Calculation lies in integrating the PE directly into the data flow. As data streams through the PE, it performs distance cal-

doi:10.6342/NTU202500592

culations in a pipelined manner, effectively hiding computation overhead under the data transfer process.

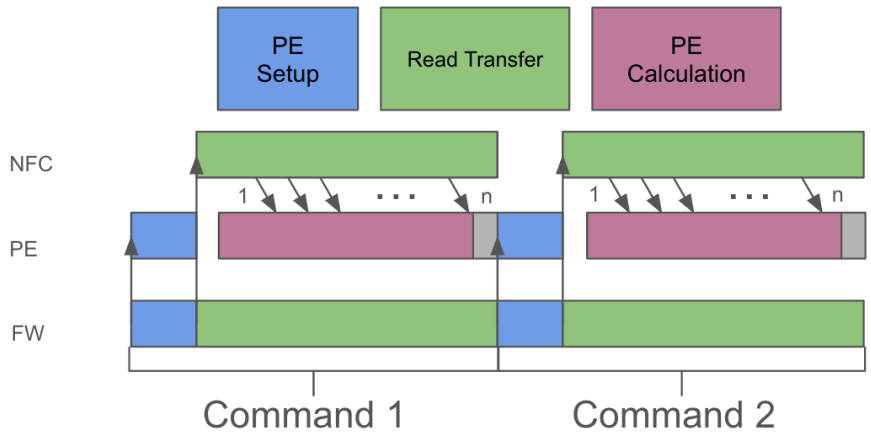


Figure 3.3: Illustration of stream distance calculation pipelining. The firmware sets PE parameters between read transfers to ensure computation accuracy.

To ensure accurate computation, each command requires distinct parameters for the PE. As illustrated in Figure 3.3, every command corresponds to a read transfer, making it essential to configure the PE parameters between consecutive transfers. Since the firmware controls the initiation of each read transfer, it dictates the timing of data movement. Additionally, the NFC notifies the firmware upon the completion of each transfer. Consequently, the firmware can easily manage argument updates, ensuring the correct parameters are set before the next command begins.

To set up a new command, the firmware provides the PE with the necessary parameters, including the number of vectors to process. Since the control logic is entirely managed by the firmware, the PE does not need to handle command transitions directly. Instead, once it completes the computation for the specified number of vectors, it simply returns to an idle state and waits for the firmware to configure new parameters before resuming operation. This design simplifies the PE's control logic, ensuring a streamlined and efficient execution flow while maintaining computational accuracy.

doi:10.6342/NTU202500592

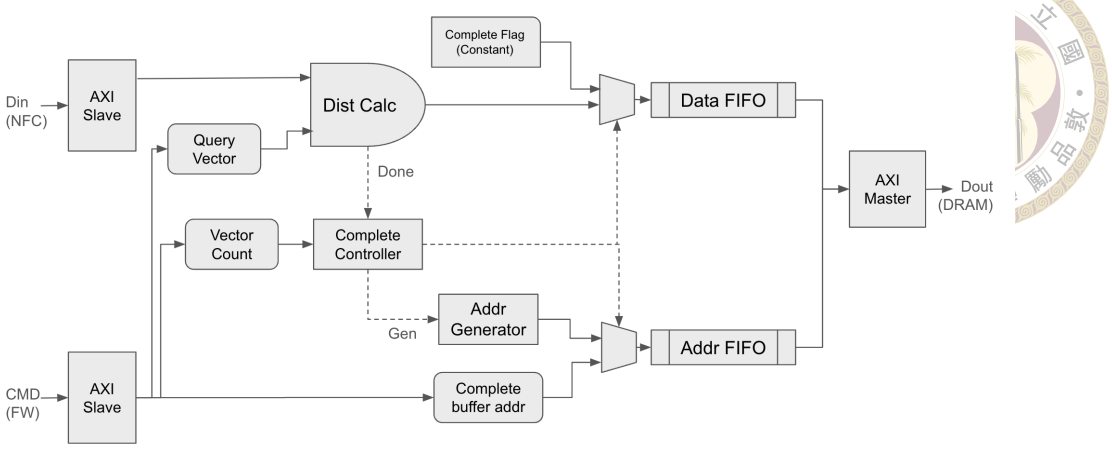


Figure 3.4: Architecture of stream distance calculator

The detailed design of the Processing Element (PE) is illustrated in Figure 3.4.

To facilitate configuration, the PE includes multiple registers that can be set up by the firmware through the **CMD interface**, enabling the storage of command arguments.

The primary function of the PE, distance calculation, is executed within the **Dist Calc** component in a pipelined manner, shown in Figure 3.5. The **Dist Calc** component retrieves data from the Din interface, computes the distances between incoming vectors, and outputs the results via the Dout interface. After the Dist Calc module completes the calculation of each vector, it triggers both the **Addr Generator** and the **Complete Controller**.

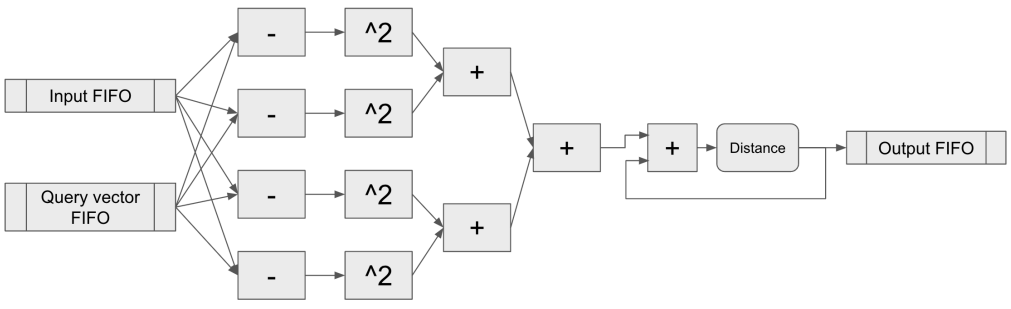
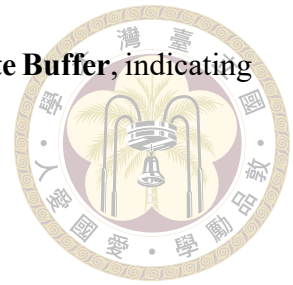


Figure 3.5: Architecture of Dist Calc

The destination addresses for the computed distances are generated dynamically by the **Addr Generator** as each vector's distance calculation is completed. Upon processing the specified number of vectors, the **Complete Controller** signals the completion of the

doi:10.6342/NTU202500592

command by writing a flag to a pre-config DRAM buffer, the **Complete Buffer**, indicating that the operation has concluded.



3.4 In-storage Distance Calculation Implementation

The implementation of in-storage distance calculation functionality on the COSMOS+ platform involves integrating hardware, firmware, and software components to enable processing within the storage device.

We utilize the Programmable Logic (PL) section of the Xilinx SoC on the COSMOS+ platform to implement the Processing Element (PE). The PE is integrated into the existing architecture through the AXI bus, with the detailed topology illustrated in Figure 3.1. By introducing an interconnect between the NFC and the PE, we can easily control whether the data stream passes through the PE by simply configuring the destination address of read transfer operations.This mechanism distinguishes standard read commands from distance calculation commands. Additionally, the PE can access the SSD DRAM through its connection to the AXI bus linking the PE and the PS, enabling it to read the query vector from and write the calculated distances to the SSD DRAM .

To support the newly added NVMe distance calculation command, modifications were made to the firmware running on the Processing System (PS). The firmware is enhanced to recognize the new command and decompose it into smaller operations, which are then scheduled and dispatched using COSMOS+'s existing low-level scheduler. To ensure the safety of data access during distance calculation commands that require simultaneous reading from the query vector buffer and writing to the data buffer, we updated COSMOS+'s internal dependency model. This enhancement enables the scheduler to ver-

doi:10.6342/NTU202500592

ify dependencies across multiple buffers, preventing operations from erroneously accessing data buffers being used by other operations. Additionally, we implemented several NVMe commands not originally supported by COSMOS+, such as flush and Namespace Identification, to ensure the system operates correctly and adheres to standard NVMe specifications.

On the host software side, we defined our custom NVMe distance calculation command by referencing the format of the existing NVMe read command. One of the main challenges in implementing this command was how to pass the query vector, typically over hundred of bytes in size, within the limited parameter fields of the NVMe command. To address this, we decided to first write the query vector into the Host Data Buffer, which was originally designed to receive data from NVMe read commands. The SSD then uses the Physical Region Page (PRP) entries provided by the NVMe command to obtain the address of the data buffer and fetch the query vector into the SSD via RxDMA.

To utilize the in-storage distance calculation, we modified the existing cluster search function to take advantage of the new NVMe distance calculation command for distance computation. The original method, shown in Algorithm 1, computes distances on the host by reading vectors from the SSD, calculating distances for each vector, and then sorting the results.

In contrast, the modified method, presented in Algorithm 2, leverages the in-storage distance calculation functionality by offloading the distance computation to the SSD. This approach significantly reduces data movements over PCIe, as only the results, rather than the raw vectors, are transferred back to the host. By doing so, the system achieves improved performance while minimizing I/O overhead.

doi:10.6342/NTU202500592

Algorithm 1 Original Method

Require: fd: file descriptor of cluster index file, C: list of LBA of clusters, qv: query vector, K: number of vectors to be found
Ensure: Top-K closest vectors

```
Distances ← {}
for c ∈ C do
    vecs ← read(fd, c)
    for (vid, data) ∈ vecs do
        d ← dist_calc(data, qv)
        Distances.push((vid, d))
    end for
end for
Distances.sort()
return Distances[: K]
```

Algorithm 2 In-storage Distance Calculation

Require: fd: file descriptor of cluster index file, C: list of LBA of clusters, qv: query vector, K: number of vectors to be found
Ensure: Top-K closest vectors

```
Distances ← {}
for c ∈ C do
    vector_count ← number of vectors in cluster c
    data_buffer ← malloc(vector_count × (sizeof(float) + sizeof(int)))
    memcpy(data_buffer, qv, qv.size())
    cmd.opcode ← DIST_CALC
    cmd.cdw10 ← c
    cmd.addr ← data_buffer
    cmd.data_len ← vector_count × (sizeof(float) + sizeof(int))
    ioctl(fd, NVME_IOCTL_IO_CMD, &cmd)
    for (vid, data) ∈ data_buffer do
        Distances.push((vid, data))
    end for
end for
Distances.sort()
return Distances[: K]
```


doi:10.6342/NTU202500592

3.5 Performance Upper Bound

To establish the performance upper bound of the system, we model the potential bottlenecks in the in-storage distance calculation process. The two most likely bottleneck components are the PCIe bus and the NAND channels. From the perspective of Distance Calculation Per Second (DCPS), we can model the system's overall DCPS as $DCPS_{\text{SSD}} = \min(DCPS_{\text{PCIe}}, DCPS_{\text{NAND}})$. Here, $DCPS_{\text{PCIe}}$ and $DCPS_{\text{NAND}}$ can be estimated as $DCPS = \frac{\text{Bandwidth}}{\text{BPDC}}$, where **BPDC (Bytes per Distance Calculation)** is defined as the amount of data (in bytes) that needs to be transferred over the bus to process a single distance calculation.

This allows us to express $DCPS_{\text{SSD}}$ as:

$$DCPS_{\text{SSD}} = \begin{cases} DCPS_{\text{PCIe}}, & \text{if } DCPS_{\text{PCIe}} < DCPS_{\text{NAND}} \\ DCPS_{\text{NAND}}, & \text{otherwise.} \end{cases}$$

We can determine the bottleneck of the system using the condition $\frac{BW_{PCIe}}{BPDC_{PCIe}} < \frac{BW_{NAND}}{BPDC_{NAND}}$.To simplify this analysis, we introduce amplification factors, which can be easily derived from the hardware configuration and workload:

$$AMP_{\text{BPDC}} = \frac{\text{BPDC}_{\text{NAND}}}{\text{BPDC}_{\text{PCIe}}}, \quad AMP_{\text{BW}} = \frac{\text{BW}_{\text{NAND}}}{\text{BW}_{\text{PCIe}}}.$$

Using these factors, the condition can be rewritten as $AMP_{\text{BPDC}} < AMP_{\text{BW}}$. This allows us to express $DCPS_{\text{SSD}}$ as:

doi:10.6342/NTU202500592

$$DCPS_{\text{SSD}} = \begin{cases} DCPS_{\text{PCIe}}, & \text{if } AMP_{\text{BPDC}} < AMP_{\text{BW}}, \\ DCPS_{\text{NAND}}, & \text{otherwise.} \end{cases}$$

To calculate the performance gain $\frac{DCPS_{\text{ISDC}}}{DCPS_{\text{Conv}}}$, we start with the assumptions: NAND Page/Cluster size is 16KB, vector size is 128 bytes (following sift1b dataset), and $AMP_{\text{BW}} = 4$ (23).

For a conventional SSD, $AMP_{\text{BPDC}} = \frac{16\text{KB}}{16\text{KB}} = 1 < 4$, therefore, the system is PCIe-bound.

For an ISDC SSD, $AMP_{\text{BPDC}} = \frac{16\text{KB}}{992\text{B}} = 16.52 > 4$, therefore, the system is NAND-bound.

The performance gain is Gain $= \frac{DCPS_{\text{ISDC}}}{DCPS_{\text{Conv}}} = \frac{DCPS_{\text{NAND}}^{\text{ISDC}}}{DCPS_{\text{PCIe}}^{\text{Conv}}}$, which can be expressed as Gain $= \frac{\text{BPDC}_{\text{PCIe}}^{\text{Conv}} \times AMP_{\text{BW}}}{\text{BPDC}_{\text{NAND}}^{\text{ISDC}}} = \frac{16\text{KB} \times 4}{16\text{KB}} = 4$.

Thus, the performance gain of ISDC over a conventional SSD is bounded by a factor of 4.

This model provides a theoretical upper bound for system performance based on the relationships between PCIe and NAND bandwidth, as well as the data transfer requirements per query.

3.6 Analysis

The experiment evaluates the performance of the distance calculation primitive under varying numbers of host software threads. The configuration is shown in Section 5.1

doi:10.6342/NTU202500592

3.6.1 Impact of In-Storage Distance Calculation on PCIe Bottleneck

In this section, we analyze the impact of in-storage distance calculation on the PCIe bottleneck through DCPS and breakdown analysis, showcasing its ability to alleviate PCIe limitations and its potential issues.

Figure 3.6 presents the DCPS performance comparison under varying thread counts for conventional and in-storage distance calculation methods. The results demonstrate that, compared to conventional methods, in-storage computing saturates at higher thread counts and achieves up to a 2.36x performance improvement. This indicates that in-storage distance calculation can handle higher IO pressure. However, the performance improvement only reaches 59% of the theoretical upper limit of 4x compared to conventional distance calculation, suggesting room for further optimization.

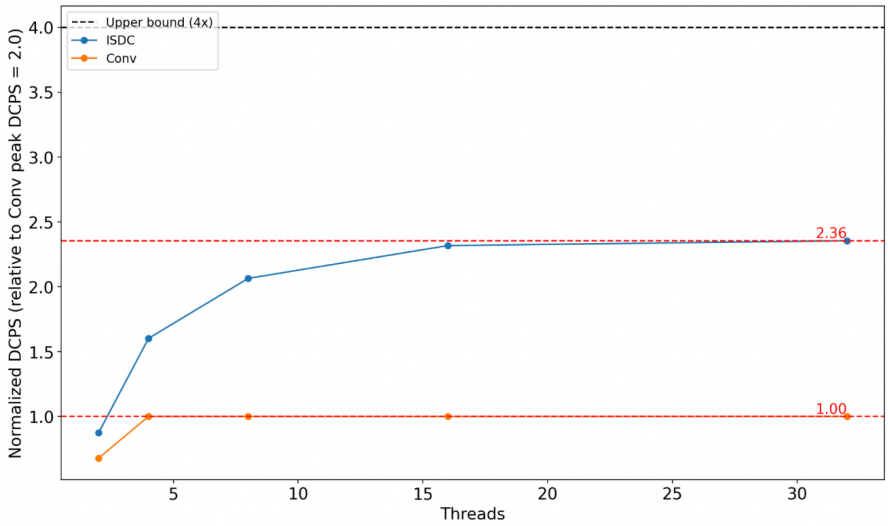


Figure 3.6: DCPS performance comparison between conventional and in-storage distance calculation under varying thread counts. In-storage distance calculation achieves up to a 2.36x improvement over conventional methods but still leaves room for optimization.

Figure 3.7 shows the breakdown analysis of in-storage distance calculation. The results reveal a significant reduction in TxDMA time, which no longer exhibits a dramatic upward trend. This demonstrates the effectiveness of in-storage distance calculation in

doi:10.6342/NTU202500592

mitigating PCIe bottlenecks.

However, compared to conventional distance calculation, we observe a noticeable increase in idle time between the read trigger and read transfer phases. We hypothesize that this idle time is the primary contributor to the performance gap between in-storage distance calculation and the ideal performance. Addressing this idle time could be key to unlocking further performance improvements and achieving closer alignment with the theoretical performance limit.

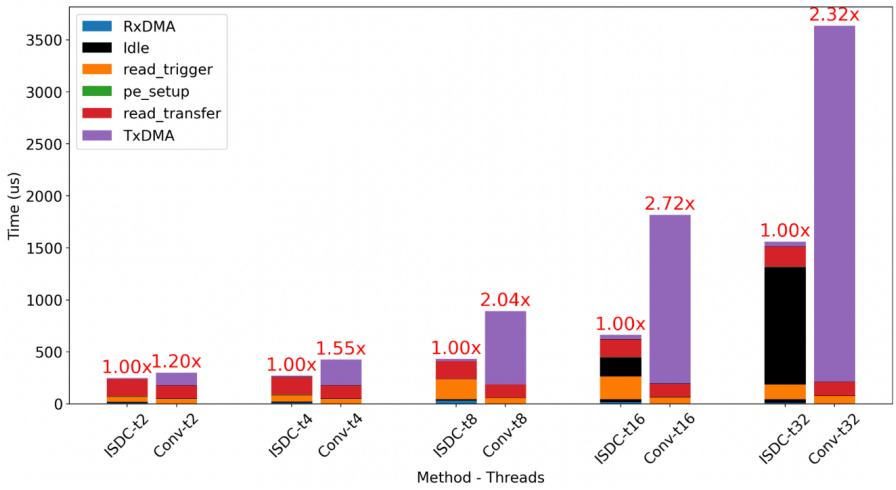


Figure 3.7: Breakdown analysis of in-storage distance calculation under varying thread counts. The analysis highlights a reduction in TxDMA time and identifies idle time between read trigger and read transfer as a potential performance bottleneck.

3.6.2 Root cause analysis of the delay between read trigger and read transfer

In this subsection, we analyze the root cause of the delay between read trigger and read transfer operations.

As illustrated in the upper part of Figure 3.8, a read transfer process can be roughly divided into two steps. In the **first step**, the NFC reads data from the corresponding NAND die via the ONFI bus and performs ECC checking. In the **second step**, the NFC transfers

doi:10.6342/NTU202500592

data out through the AXI bus. The first step and the second step can be executed in a pipelined manner.

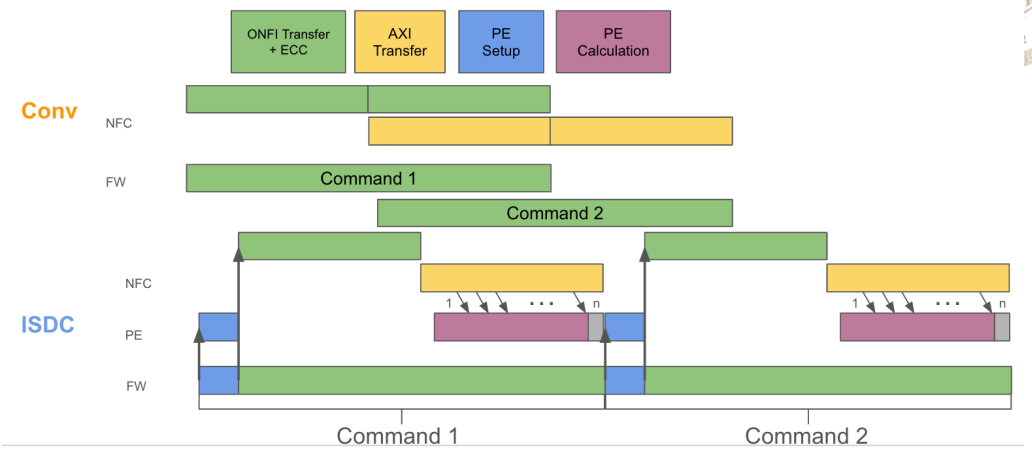


Figure 3.8: Comparison of NFC read transfer with and without PE. The upper part illustrates the pipeline in the NFC without PE, while the lower part highlights the disruption of the pipeline due to PE synchronization.

However, the incorporation of the PE disrupts this pipelining characteristic. Since each read transfer corresponds to a different distance calculation command and utilizes different parameters, the firmware must intervene between read transfer commands to update the parameters for the next command, which is illustrated in the lower part of Figure 3.8. This necessitates synchronization between the read transfer commands and the firmware, preventing the asynchronous initiation of subsequent read transfer commands as was previously possible. This synchronization extends the waiting time between read transfer commands, leading to noticeable delays observed in the breakdown. Thus, we qualitatively attribute this latency to the current PE design.

doi:10.6342/NTU202500592

第四章　Methodology

4.1　Insight

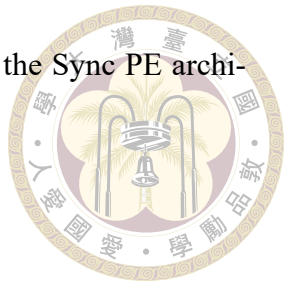
The primary issue in the current NFC design lies in the excessive latency caused by the Sync PE. In this design, the PE relies on frequent synchronization with the firmware to distinguish between commands and ensure the correct setup of arguments, thereby avoiding context errors. While this synchronization is essential for maintaining correctness, it creates a critical bottleneck by blocking the progression of operations. This dependency disrupts pipelining opportunities within the NFC, significantly degrading overall system performance.

To address this issue, this work proposes transitioning from a synchronous to an asynchronous PE (Async PE). A key feature of this design is the adoption of an **Async Setup** approach combined with an **Auto Argument Switch** mechanism. The Async Setup enables the PE to operate independently of firmware, removing the need for synchronization. Meanwhile, the Auto Argument Switch allows the PE to distinguish between commands and seamlessly switch contexts, ensuring efficient handling of concurrent operations.

By enabling asynchronous operation and equipping the PE with independent command handling capabilities, the proposed design restores pipelining opportunities in the

doi:10.6342/NTU202500592

NFC and eliminates the synchronization-induced latency inherent in the Sync PE architecture.



4.1.1 Async Setup

The primary issue in the previous design lies in the need for synchronization between the PE and the firmware during the setup phase. This synchronization creates delays, as the NFC and the PE must wait for firmware acknowledgment before proceeding, limiting the system's ability to handle multiple operations efficiently. By adopting an async setup mechanism, this dependency is removed, allowing the PE to operate independently.

The idea of async setup is the firmware can preemptively setup the arguments for the $(n+1)$th command before the nth command is fully completed. This design enables the NFC to initiate the $(n+1)$th command's ONFI transfer and ECC processing while the nth command is still undergoing AXI transfer. By overlapping these operations, the async setup mechanism effectively restores the NFC's ability to perform pipelined execution, which was hindered by the serialization bottleneck of the sync setup. This concurrent handling of commands enhances hardware utilization, improving throughput and reducing data transfer latency.

4.1.2 Auto Argument Switch

In the proposed design, the transition to an asynchronous setup requires the PE to independently determine the completion of each NAND Read Transfer. Unlike the synchronous setup, where this decision is managed by the firmware, the asynchronous setup offloads this responsibility to the PE itself. This independence is essential for enabling

doi:10.6342/NTU202500592

concurrent operations and restoring pipelining efficiency in the NFC.

For the PE to operate asynchronously, it must recognize when a NAND Read Transfer has completed. This information is critical to ensure that the PE transitions seamlessly between commands without misinterpreting their context. If the context is not properly maintained, the PE may use incorrect arguments for computations or erroneously drop data that requires processing. The Auto Argument Switch mechanism addresses this by allowing the PE to dynamically switch between arguments associated with different commands based on the transfer completion status. By doing so, the PE consistently processes operations with the correct parameters, maintaining the integrity of data processing and supporting high hardware utilization.

4.2 Processing Element Design

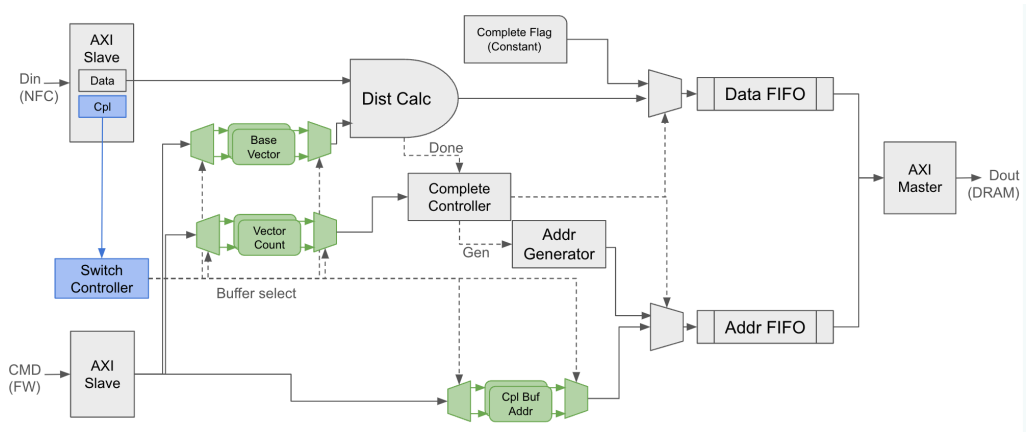
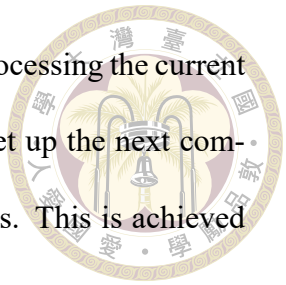


Figure 4.1: Async PE Design

4.2.1 Async Setup Implementation

The core of the proposed asynchronous setup lies in enabling the PE to operate independently of firmware intervention. In the traditional synchronous setup, the firmware

doi:10.6342/NTU202500592



can safely set up the next command only after the PE has completed processing the current command, whereas the asynchronous setup allows the firmware to set up the next command in advance, even while the current command is still in progress. This is achieved through the introduction of a **Ping Pong Buffer** mechanism.

The Ping Pong Buffer, shown in the green components of Figure 4.1, consists of two alternating buffers that decouple the setup phase from the execution phase. While one buffer, referred to as the Ping Buffer, manages the arguments for the current command, the other, called the Pong Buffer, can be asynchronously prepared by the firmware with arguments for the next command. The dual-buffer design ensures that the setup of the Pong Buffer does not interfere with the arguments in the Ping Buffer, allowing the ongoing command to execute without disruption. Crucially, this setup process no longer requires synchronization with the firmware, allowing the PE to prepare for subsequent commands without blocking.

By removing the need for synchronization, the firmware can issue the $(n+1)$th NAND read transfer to the NFC while the nth command is still undergoing data transfer, thereby restoring pipelining opportunities within the NFC.

The Ping Pong Buffer mechanism ensures efficient hardware utilization and enhances overall throughput. However, while asynchronous setup is now possible, determining the appropriate timing to switch between the Ping and Pong Buffers remains a challenge. To address this, the Auto Argument Switch mechanism is introduced, ensuring seamless and accurate buffer transitions during command execution.

doi:10.6342/NTU202500592

4.2.2 Auto Argument Switch Implementation

Maintaining the correct context for each command without firmware synchronization necessitates a reliable method for dynamic context switching. To address this issue, we propose the Auto Argument Switch mechanism, represented by the blue components in Figure 4.1.

Implementing the Auto Argument Switch mechanism in an asynchronous setup presents several challenges. Unlike the synchronous setup, where the firmware explicitly controls transitions, the asynchronous design requires the PE to independently determine when to switch contexts. This is particularly difficult because each NAND Read Transfer is treated as an independent command, and the Din interface only perceives a continuous stream of incoming data without any inherent context.

The key challenge lies in enabling the PE to accurately identify when a NAND Read Transfer is complete. Without this information, the PE risks either prematurely switching contexts or processing the data with incorrect arguments. Thus, a robust solution is needed to reliably signal the end of each transfer and facilitate seamless transitions between command contexts.

In the current NFC design, NAND Read Transfers involve large-scale data movement to DRAM, which serves as the primary data buffer. To ensure that all data has been successfully written to the buffer before notifying the firmware of transfer completion, the NFC uses a complete flag written into the buffer's designated memory location. To address this, the **Transfer Complete Register (TCR)** is introduced. The TCR is a MMIO register located on the PE's Din interface that allows the PE to monitor transfer completion in real time. During runtime, when the firmware issues a Read Transfer command

doi:10.6342/NTU202500592

to the NFC, it must provide the address of the complete buffer where the complete flag will be written. To enable direct detection by the PE, the firmware supplies the TCR's address as the complete buffer address to the NFC. As a result, the NFC writes the complete flag directly to the TCR, enabling the PE to determine when the transfer is complete without additional synchronization steps. By observing updates to the TCR, the PE gains the ability to accurately determine when each NAND Read Transfer is complete.

The **Switch Controller** ensures seamless transitions between commands in the asynchronous setup by dynamically switching the context of the PE. With the introduction of the TCR, the PE can reliably determine when a NAND Read Transfer is complete. This information serves as the trigger for the Switch Controller to adjust arguments to match the next command.

In the proposed design, the TCR signals a transfer's completion by notifying the Switch Controller, a dedicated module within the PE. Upon receiving this notification, the Switch Controller verifies that all data associated with the current command has been fully processed. Once this condition is met, it transitions the PE from the Ping Buffer to the Pong Buffer (or vice versa), ensuring that the correct arguments are applied for the next command.

The Auto Argument Switch mechanism eliminates the need for firmware synchronization to manage context transitions. By automating this process, the PE can maintain the integrity and correctness of computations while fully leveraging the benefits of the asynchronous setup.

doi:10.6342/NTU202500592

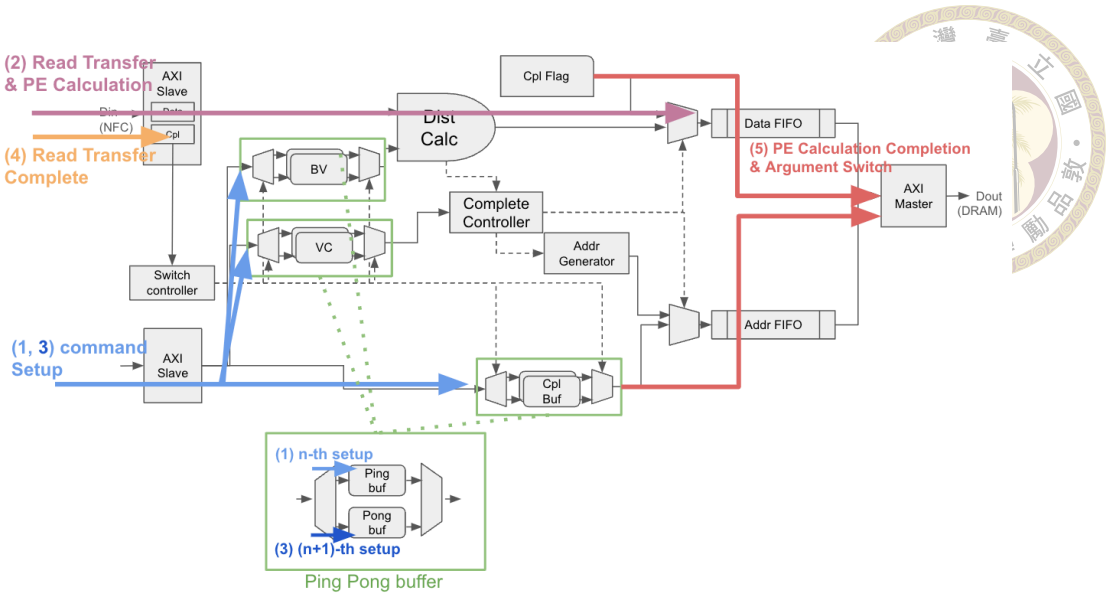


Figure 4.2: Async PE workflow

4.3 Async PE Workflow

The workflow of the Async PE is illustrated in Figure 4.2. It highlights how the system handles commands and calculations in a highly efficient, asynchronous manner. The process involves five key events:

1. **n-th Command Setup** The setup phase for the n-th command begins by configuring the required parameters through the Command Interface. These parameters are loaded into the Ping Buffer, preparing the PE to execute the command.

2. **Read Transfer and PE Calculation** After the firmware sets up the n-th command, it issues a Read Transfer command to the NFC. The PE then begins receiving data through the Din interface and performs the necessary calculations, ensuring continuous operation.

3. **$(n+1)$-th Command Setup** During the Read Transfer and calculation for the n-th command, the firmware utilizes the Command Interface to asynchronously configure the parameters for the $(n+1)$-th command. The Switch Controller automatically

doi:10.6342/NTU202500592

determines that the arguments should be written to the Pong Buffer, ensuring correct setup. Once the setup is complete, the firmware can issue the $(n+1)$-th Read Transfer command to the NFC.

4. **Read Transfer Complete** When the n-th command's Read Transfer is complete, the **Transfer Complete Register (TCR)** detects the event and signals the **Switch Controller**. The Switch Controller ensures all data associated with the n-th command is fully processed before proceeding to the next step. Meanwhile, after sending the complete flag for the n-th command, the NFC can immediately proceed with the $(n+1)$-th command's Read Transfer.

5. **PE Calculation Completion and Context Switch** After completing the calculations for the n-th command, the PE sends a complete flag to the PE complete buffer, signaling the end of its processing. At the same time **Switch Controller** transitions the PE from the Ping Buffer to the Pong Buffer. This transition ensures that the PE is ready to process the $(n+1)$-th command using the correct arguments.

The workflow in Figure 4.2 demonstrates the effectiveness of the Async PE design. By leveraging the **Ping Pong Buffer**, **TCR**, and **Switch Controller**, the system achieves non-blocking operations, restores NFC pipelining, and eliminates the need for firmware synchronization. This design ensures accurate and efficient command handling in an asynchronous environment.

doi:10.6342/NTU202500592

第五章　Evaluation

5.1　Experimental Setup

The evaluation of the proposed system involves three distinct methods to assess its effectiveness and performance. The first method, **Conventional (Conv)**, employs COSMOS+ as a conventional SSD. In this configuration, data retrieval is handled by the SSD, while distance calculations are performed on the host system. This serves as a baseline for comparison against in-storage processing methods. The second method, **In-Storage Distance Calculation (ISDC)**, shifts the computational workload of distance calculation into the COSMOS+ SSD, leveraging its internal processing capabilities to mitigate the PCIe bottleneck. Finally, the third method, **Async In-Storage Distance Calculation (Async)**, further optimizes this approach by incorporating an asynchronous mechanism into the processing element (Async PE). This mechanism is designed to exploit pipelining between the NAND Flash Controller (NFC) and the PE, enhancing overall performance.

To evaluate the system under realistic and controlled scenarios, two workloads were selected. The first workload,**Stress Test** employs a synthetic workload designed to perform distance calculations on data across consecutive addresses, maximizing the utilization of all parallelism within the SSD. The second workload employs **SPTAG** (33), a vector search library developed by Microsoft, which is representative of real-world appli-

doi:10.6342/NTU202500592

cations involving large-scale vector search.

Table 5.1: Experimental Configuration

Parameter	**Value**
Dataset	
Cluster size	16KB
Dimension	128
Data type	uint8
COSMOS+	
Page size	16KB
Aggregated NAND Bandwidth	536 MB/s
PCIe Bandwidth	133 MB/s

The experiments were conducted using the COSMOS+ SSD, configured with a 1:4 ratio of external to internal bandwidth. This ratio reflects the disparity between the PCIe interface bandwidth and the internal NAND flash channels. The hardware specifications and configurations for the experimental setup are detailed in Table 5.1.

The datasets used in the evaluation were tailored to match the requirements of the workloads. For the **Stress Test**, a synthetic dataset as specified in Table 5.1 was used. For **SPTAG**, the workload utilized the **SIFT100M** and **SPACEV100M** dataset, a downsampled version of the widely recognized SIFT1B (34) and SPACEV1B (35) dataset. Due to the storage capacity constraints of COSMOS+, the 100M datasets was generated by sampling the original datasets using the method described in (5). Although the original paper does not provide detailed instructions on this method, its implementation was based on the publicly available code provided by the authors.

doi:10.6342/NTU202500592

5.2 Stress Test Evaluation

5.2.1 Throughput

To evaluate the effectiveness of the proposed Async In-Storage Distance Calculation (Async) method, we conducted a stress test to compare its throughput with that of the In-Storage Distance Calculation (ISDC) method. The throughput, measured in Distance Calculations Per Second (DCPS), is normalized to the peak DCPS achieved by the Conventional (Conv) method for consistency in comparison.

Figure 5.1 illustrates the normalized throughput across different thread counts. The Async method demonstrates a significant improvement, achieving a **1.49x** increase in throughput compared to ISDC. Furthermore, Async reaches **88%** of the ideal performance, which is bounded by the 4x theoretical improvement over Conv's peak throughput.

This improvement is primarily attributed to the incorporation of an asynchronous mechanism in the Processing Element (PE). The Async PE effectively resolves the bottleneck between the NAND Flash Controller (NFC) and the PE, enabling better pipelining and enhancing the overall throughput.

5.2.2 Time Breakdown

To validate the effectiveness of the proposed methods in addressing previously identified issues, we analyzed the time breakdown of Conventional (Conv), In-Storage Distance Calculation (ISDC), and Async In-Storage Distance Calculation (Async) across different thread counts. Specifically, this evaluation focuses on demonstrating improvements in re-

doi:10.6342/NTU202500592

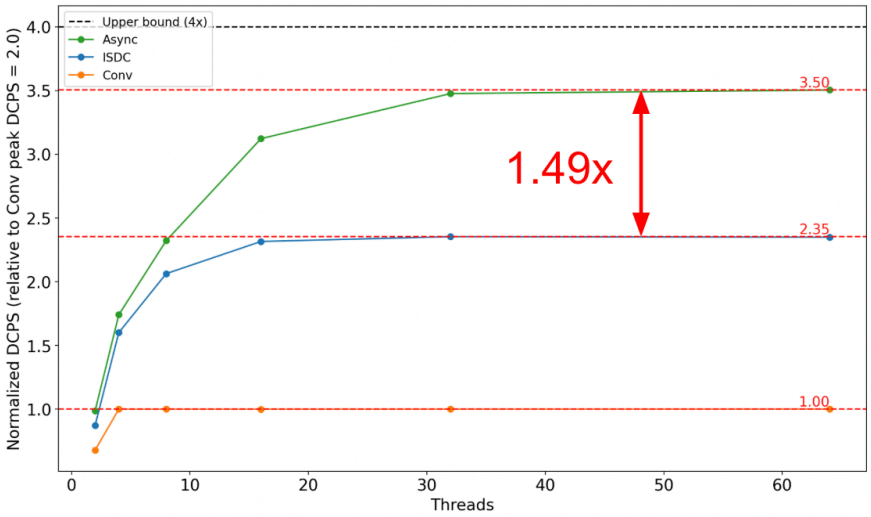


Figure 5.1: Normalized DCPS (relative to Conv peak DCPS) across varying thread counts. Async achieves a 1.49x improvement over ISDC and reaches 88% of the ideal performance.

solving the *Idle Before Transfer* issue, which was characterized by prolonged idle periods between the *Read Trigger* and *Read Transfer* stages.

The results, as shown in Figure 5.2, confirm that the Async method significantly reduces idle time (black segments) between these stages (orange and read segments). This improvement stems from the asynchronous mechanism incorporated into the Processing Element (PE), which facilitates better pipelining between the NAND Flash Controller (NFC) and the PE. By allowing different read transfer operations on the same channel to be efficiently pipelined, the Async method effectively minimizes idle periods and optimizes resource utilization.

doi:10.6342/NTU202500592

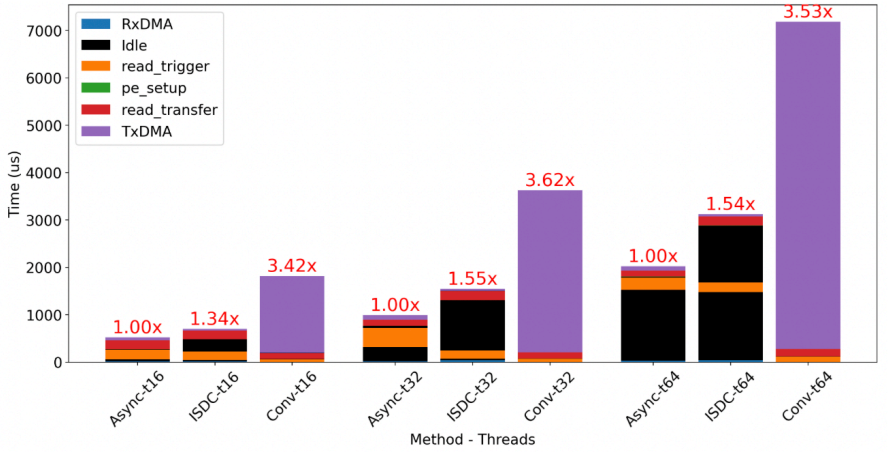


Figure 5.2: Time breakdown of Conventional (Conv), ISDC, and Async across different thread counts. Async significantly reduces the idle time (*Idle Before Transfer*) and improves overall pipelining efficiency.

5.3 End-to-End Evaluation

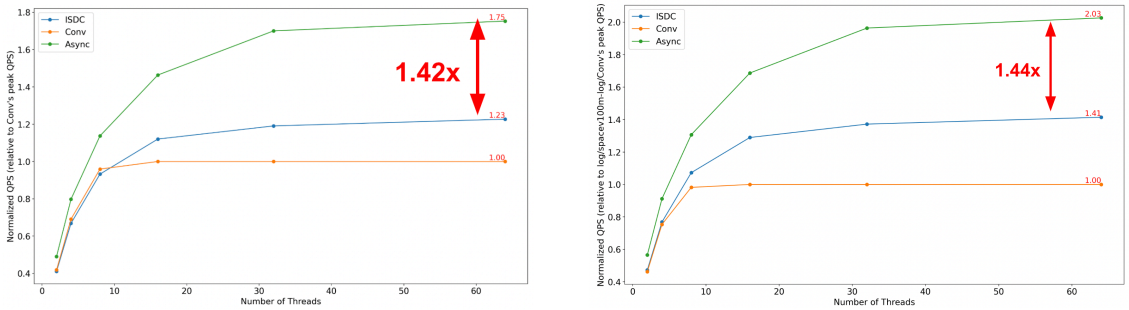
5.3.1 Throughput

To evaluate the performance of the proposed system under real-world workloads, we ran SPTAG on the SIFT100M dataset to simulate vector search operations. This workload represents practical applications involving large-scale vector similarity searches, where each query requires multiple distance calculations. The throughput, measured in Queries Per Second (QPS), is normalized to the peak QPS achieved by the Conventional (Conv) method to enable consistent comparison.

As shown in Figure 5.3, the Async In-Storage Distance Calculation (Async) method consistently improves end-to-end performance across different datasets. For SIFT100M, Async achieves a **1.75x** speedup over Conv and a **1.42x** improvement over ISDC. Similarly, in SpaceV100M, Async demonstrates a **2.03x** performance gain compared to Conv and a **1.44x** improvement over ISDC. These results highlight the effectiveness of the asynchronous mechanism in enhancing system throughput under real-world workloads. How-

doi:10.6342/NTU202500592

ever, despite these improvements, there remain gaps of **56%** and **49%** to the ideal (4x) performance for SIFT100M and SpaceV100M, respectively.



(a) SIFT100M (b) SpaceV100M

Figure 5.3: Normalized QPS (relative to Conv's peak QPS) for end-to-end evaluation using SPTAG on the SIFT100M and SPACEV100M datasets. Async achieves a 1.75x performance improvement over Conv and a 1.42x improvement over ISDC, with a 56% gap to the ideal performance.

5.3.2 Cluster Size and Performance Correlation

The **56% performance gap** relative to the ideal throughput arises due to variations in cluster size. In real-world workloads, clusters are not uniform in size, leading to differences in how data is accessed and processed. This difference in cluster size results in varying throughput, primarily due to the mismatch in access granularity between NVMe (4KB) and NAND flash (16KB). In the Conv method, when accessing smaller clusters, the system must retrieve entire 16KB pages from NAND flash, even if only a fraction of the data is needed. However, since only the required portion of the data is transferred over PCIe, the overall PCIe traffic per request is reduced. This lower PCIe bandwidth usage allows for a higher number of distance calculations per second (DCPS), ultimately improving system throughput, as PCIe remains the bottleneck in this scenario.

To quantify this effect, we used the stress test workload to compare the speedup achieved by the Async method over the Conventional (Conv) method across different

doi:10.6342/NTU202500592

cluster sizes. The results, shown in Figure 5.4, illustrate the speedup obtained by Async under varying cluster sizes (4KB to 16KB). The figure demonstrates that smaller cluster sizes yield lower speedups compared to larger cluster sizes, with the speedup ranging from a minimum of **0.9x** for 4KB clusters to a maximum of **3.5x** for 16KB clusters. This highlights the significant impact of cluster size on throughput improvement.

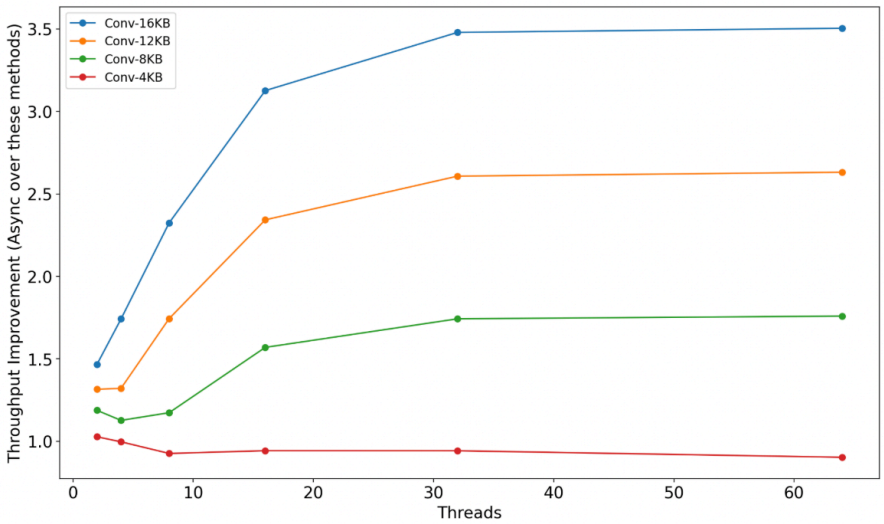


Figure 5.4: Throughput improvement (Async over Conv) for different cluster sizes (4KB to 16KB). Smaller cluster sizes result in reduced speedup due to less effective utilization of PCIe bandwidth.

To further illustrate the correlation between cluster size and end-to-end performance, Table **??** shows the proportion of different cluster sizes in the SIFT100M and SPACEV100M dataset. It can be observed that the majority of clusters in SIFT100M are 4KB (38.83%) and 8KB (48.09%), whereas in SPACEV100M, the distribution is more balanced, with 4KB (33.75%), 8KB (31.03%), and 12KB (31.24%) clusters each occupying a significant portion of the dataset. This indicates that cluster sizes tend to be small in the current setup.

The prevalence of smaller clusters can be attributed to the limitations of larger clusters. Due to the following issues with larger clusters, previous indexing methods tend to construct smaller clusters (4). Larger clusters require more I/O overhead to retrieve data from the SSD, as described in the PCIe bottleneck we previously introduced. Addi-

doi:10.6342/NTU202500592

tionally, larger clusters result in more distance calculations per cluster, further increasing computational demands. As a result, overly large cluster sizes negatively impact search performance, and prior indexing works have focused on reducing cluster sizes to mitigate these challenges.

The introduction of in-storage distance calculation brings new characteristics that mitigate the limitations of smaller clusters. Since in-storage distance calculation mitigates the PCIe bottleneck, the current bottleneck shifts to the NAND Flash. Regardless of cluster size, data access is performed in NAND Page units on the new bottleneck. Computation is also executed directly in these units, enabling the processing of larger clusters without incurring additional overhead. These characteristics eliminate the need for indexing to favor small cluster sizes and open up new directions for exploration.

Cluster Size	SIFT100M (%)	SPACEV100M (%)
4K	38.83	33.75
8K	48.09	31.03
12K	11.02	31.24
16K	2.06	3.98

Table 5.2: Cluster size distribution in the SIFT100M and SPACEV100M datasets. The majority of clusters are 4KB and 8KB, indicating that existing indexing methods tend to create smaller clusters.

5.4 Future Direction

Building on the observation that larger clusters do not introduce additional overhead due to in-storage distance calculation, we propose the following research directions to leverage this characteristic.

The first direction involves refining the indexing method to form clusters that closely match the NAND page size by aligning cluster sizes with the underlying hardware' s

doi:10.6342/NTU202500592

access granularity. We hypothesize that, given a similar number of examined vectors, it is possible to achieve comparable accuracy by checking a few large clusters instead of many small clusters. Therefore, increasing the cluster size to match the NAND page size allows for fewer clusters to be checked while maintaining a high recall rate. This approach minimizes unnecessary data movement, offering the potential to improve the overall query efficiency without compromising the recall rate in large-scale search systems.

The second direction explores the use of cluster packing techniques to enhance data locality on NAND. Inspired by packing methods proposed in (9), this approach leverages the structure of in-memory graph indices and the distance characteristics between clusters to predict the likelihood of clusters being accessed simultaneously. By placing clusters that are likely to be accessed together on the same NAND page, this method effectively improves locality on NAND, reducing the number of NAND reads required. By carefully arranging data in this manner, it becomes possible to minimize redundant data access, thereby improving overall performance.

These directions highlight how the unique characteristics enabled by in-storage distance calculation can inspire new indexing and data organization strategies. By addressing the limitations of conventional approaches and optimizing for hardware-specific characteristics, these strategies pave the way for more efficient and scalable solutions in the future.

doi:10.6342/NTU202500592

第六章　Related Works

Recent advances in Approximate Nearest Neighbor Search (ANNS) algorithms have leveraged innovative hardware architectures to address the challenges posed by billion-scale vector datasets. These approaches can be broadly classified into those utilizing Solid-State Drives (SSDs) and those based on in-storage computing (ISC).

6.1　SSD-Based ANNS

The deployment of ANNS algorithms on SSDs aims to balance storage efficiency, query latency, and scalability by leveraging hybrid memory-disk architectures. Among graph-based approaches, *DiskANN*(8) and *DiskANN++*(9) stand out as prominent solutions. *DiskANN* introduces a hybrid index that combines a Product Quantization (PQ)-based compressed memory index with a graph structure stored on SSDs. This design reduces memory overhead while achieving high recall and low query latency. Building upon *DiskANN*, *DiskANN++* addresses inherent I/O inefficiencies by introducing a query-sensitive entry vertex selection and employing isomorphic mapping to optimize SSD layouts. These enhancements significantly improve queries per second (QPS) performance, making it more suitable for large-scale applications.

In contrast, cluster-based methods focus on organizing datasets into partitions that

doi:10.6342/NTU202500592

simplify storage and retrieval operations. *SPANN*(4) adopts a hierarchical balanced clustering algorithm to partition datasets into clusters, where centroids are stored in memory and vector data resides on SSDs. This strategy ensures balanced disk access and minimizes query latency by dynamically pruning irrelevant clusters during the search phase. Extending this paradigm, *SPFresh*(5) introduces a lightweight incremental rebalancing protocol (LIRE), enabling in-place vector updates. By addressing the challenges of data distribution skew and partition imbalance, *SPFresh* achieves superior query latency and accuracy while significantly reducing resource overhead compared to traditional global index rebuilds.

6.2 Processing-in-Memory for ANNS

Processing-in-Memory (PIM) architectures address the memory bandwidth bottleneck in billion-scale ANNS by integrating computation within memory modules. Pyramid-M (10)operates at the DRAM level, performing graph-based ANNS in main memory. It leverages distributed distance calculations across memory ranks and centralized sorting in the memory controller hub (MCH) to optimize query processing. By storing graph structures in DRAM and employing content-addressable memory (CAM)-based filtering, Pyramid-M reduces redundant distance computations and minimizes data transfers, significantly improving efficiency.

Similarly, MemANNS (36) utilizes commercial PIM hardware by implementing an optimized IVFPQ-based ANNS framework on UPMEM’ s DIMM-integrated processing units. It optimizes query execution by introducing architecture-aware workload distribution, ensuring balanced memory access across PIM chips to maximize aggregate

doi:10.6342/NTU202500592

bandwidth. Additionally, it employs a co-occurrence-aware encoding strategy that reduces redundant memory accesses by precomputing frequently accessed vector components. These architectural refinements enable MemANNS to effectively handle large-scale ANNS workloads while reducing computational overhead.

6.3 Compute Express Link (CXL) for ANNS

Compute Express Link (CXL) enables scalable memory disaggregation, addressing the high memory demands of billion-scale Approximate Nearest Neighbor Search (ANNS). CXL-ANNS(37) leverages CXL memory pools to expand working memory while mitigating access latency through relationship-aware caching, which stores frequently accessed graph nodes in local DRAM. Additionally, ANNS-aware prefetching predicts future node accesses to hide CXL memory delays.

To reduce data movement, CXL-ANNS offloads distance calculations to CXL endpoint (EP) devices, avoiding the need to transfer full feature vectors. Furthermore, fine-grained query scheduling optimizes resource utilization by overlapping computation tasks between the host and EPs. By integrating memory disaggregation, near-data processing, and intelligent caching, CXL-ANNS efficiently scales ANNS while maintaining high performance.

6.4 In-Storage Computing-Based ANNS

In-storage computing (ISC) approaches represent a paradigm shift by integrating computational capabilities directly within storage devices, thereby minimizing data move-

doi:10.6342/NTU202500592

ment and enhancing energy efficiency. *VStore*(38), a graph-based vector search accelerator, exemplifies this approach. By embedding computation within the SSD controller, *VStore* eliminates the overhead associated with transferring data to host memory. The system employs query orchestration and result caching to exploit data reuse effectively, resulting in substantial improvements in speed and energy efficiency. Similarly, computational storage platforms, such as those built on programmable logic within SmartSSDs, enable efficient execution of graph-based ANNS algorithms. These systems leverage near-data processing to address critical bottlenecks in traditional architectures, including limited memory bandwidth and high data transfer costs (39).

6.5 Other Hardware Accelerators for ANNS

ANNA (40) is a dedicated accelerator designed for product quantization(PQ)-based ANNS, which is widely used in large-scale ANNS applications. It introduces a specialized dataflow pipeline that maximizes computation parallelism while minimizing memory traffic. By efficiently handling PQ operations, ANNA achieves high throughput and low latency, making it well-suited for billion-scale vector search tasks.

DF-GAS (41) is a distributed FPGA-as-a-service architecture, extends the benefits of hardware acceleration to large-scale graph-based ANNS. DF-GAS employs a hybrid parallel search scheme that combines full-graph and sub-graph traversal strategies, optimizing memory access and reducing remote communication overhead. Additionally, its feature-packing memory engine and delayed processing techniques enhance memory bandwidth utilization, making it highly efficient for distributed ANNS applications.

doi:10.6342/NTU202500592

第七章　Conclusion

This work marks the first implementation of in-storage cluster-based vector search on a real hardware platform, bridging the gap between theoretical proposals and practical applications. Through comprehensive quantitative analysis, we addressed limitations in prior research, particularly in understanding the performance characteristics of in-storage computing devices.

We also identify the loss of pipelining between processing element and NAND flash controller and proposed the Async Processing Engine to mitigate this issue. This enhancement significantly improved pipeline efficiency and reduced bottlenecks, achieving a 3.50x throughput improvement in the distance calculation primitive. Notably, this throughput reached 88% of the theoretical maximum performance, underscoring the effectiveness of our design.

In the context of end-to-end vector search workloads, our system demonstrated a 1.42x ~1.44x improvement in throughput compared to the state-of-the-art in-storage computing solution, further validating the benefits of in-storage computation in real-world applications. Alongside these advancements, we identified and analyzed new system bottlenecks. While the adoption of ISC has significantly reduced PCIe traffic, the primary bottleneck has now shifted to the NAND itself. To address this challenge, we proposed

doi:10.6342/NTU202500592

new directions for improvement, focusing on optimizing NAND-level operations and enhancing parallelism in NAND access. These recommendations aim to further unlock the potential of in-storage computing systems and advance their performance in large-scale workloads.

In conclusion, this study provides a foundational step toward more efficient and scalable in-storage computing systems, setting the stage for further innovations in the field of large-scale vector search and beyond. Future work can build on these findings to refine in-storage designs and tackle the newly uncovered challenges.

doi:10.6342/NTU202500592

參考文獻

[1] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," Advances in Neural Information Processing Systems, 2013.

[2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," Advances in Neural Information Processing Systems, 2020.

[3] R. Chen, B. Liu, H. Zhu, Y. Wang, Q. Li, B. Ma, Q. Hua, J. Jiang, Y. Xu, H. Deng et al., "Approximate nearest neighbor search under neural similarity metric for large-scale recommendation," in ACM International Conference on Information & Knowledge Management, 2022.

[4] Q. Chen, B. Zhao, H. Wang, M. Li, C. Liu, Z. Li, M. Yang, and J. Wang, "Spann: Highly-efficient billion-scale approximate nearest neighborhood search," Advances in Neural Information Processing Systems, 2021.

[5] Y. Xu, H. Liang, J. Li, S. Xu, Q. Chen, Q. Zhang, C. Li, Z. Yang, F. Yang, Y. Yang et al., "Spfresh: Incremental in-place update for billion-scale vector search," in Symposium on Operating Systems Principles, 2023.

doi:10.6342/NTU202500592

[6] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," IEEE Transactions on Pattern Analysis and Machine Intelligence, 2018.

[7] C. Fu, C. Xiang, C. Wang, and D. Cai, "Fast approximate nearest neighbor search with the navigating spreading-out graph," arXiv, 2017.

[8] S. Jayaram Subramanya, F. Devvrit, H. V. Simhadri, R. Krishnawamy, and R. Kadekodi, "Diskann: Fast accurate billion-point nearest neighbor search on a single node," Advances in Neural Information Processing Systems, 2019.

[9] J. Ni, X. Xu, Y. Wang, C. Li, J. Yao, S. Xiao, and X. Zhang, "Diskann++: Efficient page-based search over isomorphic mapped graph index using query-sensitivity entry vertex," arXiv, 2023.

[10] Z. Zhu, J. Liu, G. Dai, S. Zeng, B. Li, H. Yang, and Y. Wang, "Processing-in-hierarchical-memory architecture for billion-scale approximate nearest neighbor search," in ACM/IEEE Design Automation Conference, 2023.

[11] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ openssd: Rapid prototype for flash storage systems," ACM Transactions on Storage, 2020.

[12] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: The evolution of data to life-critical; don' t focus on big data, focus on the data that' s big," IDC, Tech. Rep., 2017.

[13] E. Mehmood and T. Anees, "Distributed real-time etl architecture for unstructured big data," Knowledge and Information Systems, 2022.

[14] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, M. Ranzato, and

doi:10.6342/NTU202500592

T. Mikolov, "Devise: A deep visual-semantic embedding model," Advances in Neural Information Processing Systems, 2013.

[15] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," Advances in Neural Information Processing Systems, 2013.

[16] J. Wang, X. Yi, R. Guo, H. Jin, P. Xu, S. Li, X. Wang, X. Guo, C. Li, X. Xu et al., "Milvus: A purpose-built vector data management system," in International Conference on Management of Data, 2021.

[17] M. Wang, X. Xu, Q. Yue, and Y. Wang, "A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search," arXiv, 2021.

[18] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," IEEE Transactions on Pattern Analysis and Machine Intelligence, 2010.

[19] I. NVM Express, "Nvm express base specification, revision 2.1," https://nvmexpress.org/specifications/, 2024.

[20] K.-C. Ho, P.-C. Fang, H.-P. Li, C.-Y. M. Wang, and H.-C. Chang, "A 45nm 6b/cell charge-trapping flash memory using ldpc-based ecc and drift-immune soft-sensing engine," in IEEE International Solid-State Circuits Conference Digest of Technical Papers, 2013.

[21] S. Li and T. Zhang, "Improving multi-level nand flash memory storage reliability using concatenated bch-tcm coding," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2009.

doi:10.6342/NTU202500592

[22] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart ssd," in Symposium on Mass Storage Systems and Technologies, 2013.

[23] G. Koo, K. K. Matam, T. I, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in IEEE/ACM International Symposium on Microarchitecture, 2017.

[24] J. Do, S. Sengupta, and S. Swanson, "Programmable solid-state storage in future cloud datacenters," Communications of the ACM, 2019.

[25] A. Barbalace and J. Do, "Computational storage: Where are we today?" in Conference on Innovative Data Systems Research, 2021.

[26] I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong, "Yoursql: a high-performance database system leveraging in-storage computing," Proceedings of the VLDB Endowment, 2016.

[27] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho et al., "Biscuit: A framework for near-data processing of big data workloads," ACM SIGARCH Computer Architecture News, 2016.

[28] C. Zou and A. A. Chien, "Assasin: Architecture support for stream computing to accelerate computational storage," in IEEE/ACM International Symposium on Microarchitecture, 2022.

[29] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "Smartssd: Fpga accelerated near-storage data analytics on ssd," IEEE Computer Architecture Letters, 2020.

doi:10.6342/NTU202500592

[30] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "Nascent: Near-storage acceleration of database sort on smartssd," in ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, 2021.

[31] H. Jang, J. Song, J. Jung, J. Park, Y. Kim, and J. Lee, "Smart-infinity: Fast large language model training using near-storage processing on a real system," in IEEE International Symposium on High-Performance Computer Architecture, 2024.

[32] Y. Chen, J. Xu, C. Wei, Y. Wang, X. Yuan, Y. Zhang, X. Yu, Y. Chen, Z. Wang, S. He et al., "Bm-store: A transparent and high-performance local storage architecture for bare-metal clouds enabling large-scale deployment," in IEEE International Symposium on High-Performance Computer Architecture, 2023.

[33] Q. Chen, H. Wang, M. Li, G. Ren, S. Li, J. Zhu, J. Li, C. Liu, L. Zhang, and J. Wang, "SPTAG: A library for fast approximate nearest neighbor search," https://github.com/Microsoft/SPTAG, 2018.

[34] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg, "Datasets for approximate nearest neighbor search," http://corpus-texmex.irisa.fr/, 2011.

[35] Microsoft, "SPACEV1B: A Billion-Scale Vector Dataset for Text Descriptors," https://github.com/microsoft/SPTAG/tree/main/datasets/SPACEV1B, 2020.

[36] S. Chen, A. C. Zhou, Y. Shi, Y. Li, and X. Yao, "Memanns: Enhancing billion-scale anns efficiency with practical pim hardware," arXiv, 2024.

[37] J. Jang, H. Choi, H. Bae, S. Lee, M. Kwon, and M. Jung, "CXL-ANNS:Software-Hardware collaborative memory disaggregation and computation for Billion-Scale approximate nearest neighbor search," in USENIX Annual Technical Conference, 2023.

doi:10.6342/NTU202500592

[38] S. Liang, Y. Wang, Z. Yuan, C. Liu, H. Li, and X. Li, "Vstore: in-storage graph based vector search accelerator," in ACM/IEEE Design Automation Conference, 2022.

[39] J.-H. Kim, Y.-R. Park, J. Do, S.-Y. Ji, and J.-Y. Kim, "Accelerating large-scale graph-based nearest neighbor search on a computational storage platform," IEEE Transactions on Computers, 2022.

[40] Y. Lee, H. Choi, S. Min, H. Lee, S. Beak, D. Jeong, J. W. Lee, and T. J. Ham, "Anna: Specialized architecture for approximate nearest neighbor search," in IEEE International Symposium on High-Performance Computer Architecture, 2022.

[41] S. Zeng, Z. Zhu, J. Liu, H. Zhang, G. Dai, Z. Zhou, S. Li, X. Ning, Y. Xie, H. Yang et al., "Df-gas: a distributed fpga-as-a-service architecture towards billion-scale graph-based approximate nearest neighbor search," in IEEE/ACM International Symposium on Microarchitecture, 2023.

doi:10.6342/NTU202500592