Design and Optimization for AI/ML Acceleration on Resource-constrained and Edge Systems

Jalil Boukhobza
Lab-STICC, CNRS UMR 6285, ENSTA, Institut Polytechnique de Paris
Brest, France
jalil.boukhobza@ensta.fr

Alessio Burrello
Politecnico di Torino
Turin, Italy
alessio.burrello@polito.it

Yuan-Hao Chang
National Taiwan University
Taipei, Taiwan
johnson@csie.ntu.edu.tw

Yawei Li
ETH Zurich
Zurich, Switzerland
yawli@iis.ee.ethz.ch

Daniele Jahier Pagliari
Politecnico di Torino
Turin, Italy
daniele.jahier@polito.it

Chun-Feng Wu
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
cfwu417@cs.nycu.edu.tw

Ming-Chang Yang
The Chinese University of Hong Kong
Hong Kong, China
mcyang@cse.cuhk.edu.hk

Tsun-Yu Yang
Duke University
Durham, USA
tsun-yu.yang@duke.edu

Abstract

The rapid advancement of AI (from foundational machine learning to Large Language Models) and edge computing has placed unprecedented demands on computation, memory, and storage on resource-constrained edge devices. As AI models scale, the ability to efficiently manage computing resources, utilize memory and storage, and reduce energy consumption has become critical. This paper introduces contributions on 4 topics related to deploying AI on resource-constrained edge devices: 1) unlocking training of foundational machine learning algorithms on the edge, 2) exploring hardware-aware DNN architecture and mapping co-optimization for inference on heterogeneous systems, 3) scaling RAG by leveraging advanced memory, storage, and energy-efficient designs, and 4) investigating cost-effective and high-performance large-scale graph processing.

CCS Concepts

• **Hardware → Memory and dense storage**; **External storage**; • **Computer systems organization → Embedded systems**; • **Computing methodologies → Artificial intelligence**; **Machine learning**; **Parallel computing methodologies**.

Keywords

Edge AI, Memory, Storage, K-means, GMM, DNN, heterogeneous computing, RAG, ANN, KV cache, I/O bottleneck, graph processing

ACM Reference Format:
Jalil Boukhobza, Alessio Burrello, Yuan-Hao Chang, Yawei Li, Daniele Jahier Pagliari, Chun-Feng Wu, Ming-Chang Yang, and Tsun-Yu Yang. 2025. Design and Optimization for AI/ML Acceleration on Resource-constrained and Edge Systems. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '25), September 28-October 3, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3742872.3758334

1 Introduction

IoT devices and mobile phones collect and analyze huge amounts of data to transform them to meaningful information [60]. Analyses are traditionally performed on the cloud using available large computational power. This data transmission is done at a cost of large network traffic and energy [4, 27] and an exposure to security threats. When dealing with medical [49] and industrial [68] applications, this could be problematic. Processing data near where they were generated (on the embedded device or on the edge) is a more favorable solution in several cases [21].

Edge AI refers to the deployment of artificial intelligence algorithms and models directly on local hardware devices—known as edge devices—rather than relying on centralized cloud servers. This enables real-time data processing at or near the location where the data are generated, reducing the need to constantly transmit information to the cloud and avoiding data exposure to threats. While this solution is very attractive for privacy, latency, and security, it may pose significant challenges due to resource scarcity on edge devices. An edge device can be of different sizes ranging from sensors, cameras, smartphones, IoT devices, and embedded systems, up to a micro data center with several server racks [78]. However, it is commonly accepted that edge devices are typically heterogeneous and resource-constrained platforms, in terms of computing power, memory, storage capacity, and performance [59].

Edge AI has been classified into 6 levels [59]. In levels 1 to 3 the learning is done on the cloud while the inference could be done either on both the cloud and the edge (level 1), on the edge (level 2) or on the embedded device itself (level 3). In higher levels (4 to 6), it is the learning process that could be deployed whether on both the cloud and the edge (level 4), on the edge (level 5) or on the embedded device itself (level 6). While some foundational machine learning models such as K-means, random forest, SVM, could be deployed at different levels of Edge AI (from both training and inference perspectives, levels 1 to 6), other more advanced AI models such as Large Language Models (LLMs) require huge amounts of computations for the training but could be deployed on edge devices for the inference if well optimized (levels 4 to 6). Edge AI is a very broad topic as it spans a variety of models and targets very diverse hardware platforms. This paper explores 4 venues of optimization in the scope of Edge AI:

- **Learning on the edge:** This first section explores foundational ML algorithms deployment for learning on resource constrained devices. One of the main challenges discussed is the memory scarcity that pushes the learning process to generate a huge amount of I/O operations. A solution pattern is proposed and 2 case-studies presented: K-means and Gaussian Mixture Models.
- **DNN mapping on the edge:** In this section, we focus on the optimization of DNN inference at the edge. We present a toolchain that permits to efficiently map a given DNN onto a heterogeneous hardware platform, while also adapting the DNN model architecture to better match the hardware characteristics, co-optimizing for task accuracy and inference cost (e.g. latency, energy, etc).
- **Retrieval-Augmented Generation on the edge:** One promising approach to enabling LLMs on edge devices is to narrow their scope by tailoring them to domain-specific contexts. Retrieval-Augmented Generation (RAG) supports this by retrieving only the most relevant external information for each query, reducing the computational load and improving efficiency. However, the RAG architecture introduces new challenges, particularly in memory usage, I/O performance, and system coordination. These bottlenecks become more pronounced in resource-constrained environments where storage, memory bandwidth, and compute are limited. In this section, we present key system-level optimizations designed to make RAG practical and scalable for edge deployments.
- **Graph Processing on the edge:** Graphs are commonly used for diverse applications in AI/ML, for instance in RAG systems and Graph Neural Network (GNN). However, when deployed on edge resource constrained devices, and because graph sizes grow exponentially, deploying efficient large-scale graph requires revising how memory and storage are leveraged for out-of-core graph processing. This part explores strategies to achieve cost-effective, high-performance graph processing with storage.

2 Learning on the edge

One of the main issues with on-device learning is the limitation in the size of the main memory in edge devices. In fact, memory capacity can hardly scale as fast as the volume of data to process [47]. To overcome these constraints, some studies investigate how to

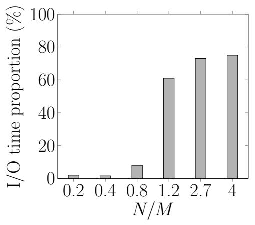

Figure 1: Proportion of time spent on I/O operations for different values of N/M [60]

trade accuracy for simpler models [32, 58]. **We argue that one can also handle memory space limitation during the learning process by revising ML algorithms in order to reduce I/O movements while maintaining comparative accuracy.**

2.1 Motivation

To motivate the relevance of our contribution, we discuss the example of the K-means algorithm [60], since it is widely used. Let us assume a dataset of size $N \cdot s$ where N is the number of data elements and s the size of a data element, and a memory workspace of $M \cdot s$, where M is the number of data elements that can fit into the memory space. Figure 1 shows the measured learning execution time proportion spent on I/O operations when increasing the ratio N/M (the higher the ratio, the larger the dataset as compared to the available memory). We used embedded boards representative of edge devices in our studies [31]. We can observe that the larger N/M, the higher the proportion of time spent in I/Os (this translates into very high execution times), up to 72% of the overall execution because of the performance difference between DRAM and flash memory [5, 6], knowing that K-means is supposed to be CPU-bound.

2.2 Problem Statement and General Pattern

The high number of I/O operations observed during the learning phase of several ML algorithms deployed on edge devices is mainly due to the swapping process. Algorithm 1 shows the overall pattern of several ML algorithms. One can observe that it needs to go through all (or a large part of) the dataset to update the model for each iteration. From a memory perspective, when the dataset fits in the available memory, there is no issue; the algorithm is CPU-bound, as expected. However, if the dataset does not fit within the memory space available, several I/Os are generated for each iteration to go through the whole data and update the model for each iteration. That is the dataset is read as many times as there are iterations, which dramatically increases the execution time of the learning phase as temporal locality is not leveraged.

Algorithm 1 General problem pattern

```
1: while stop criteria is not reached do
2:     Read all (or a large part) of the dataset
3:     perform some model related processing.
4:     update the model
5: end while
```

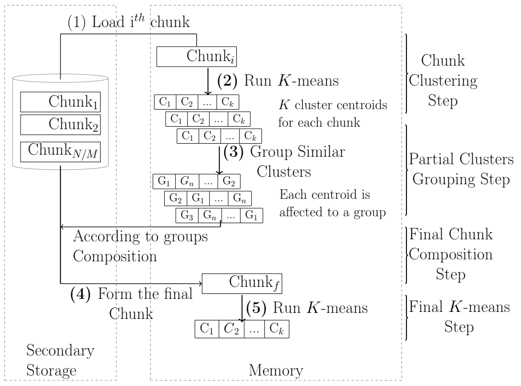

Figure 2: K-MLIO overview [60]

2.3 Proposed Solution

We propose to revise ML learning algorithms that obey the pattern shown in Algorithm 1 in a way to dissociate the computation from the need for full dataset reading. We do so by subdividing the dataset into chunks and then learning the model on chunks of data, see Algorithm 2. Once done, we need to find a way to merge the resulting sub-models into one global model (line 10) with a minimal loss in accuracy. In the next sections, we briefly summarize two case studies we considered, K-means [60] and Gaussian Mixture [7, 8].

Algorithm 2 General solution pattern

```
1: Subdivide the dataset into chunks that fit in the memory
2: for each chunk $i$ of the dataset do
3:     while stop criteria is not reached do
4:         Read all data in the chunk (part of the dataset)
5:         Perform some model related processing.
6:         Update the model
7:     end while
8:     Store submodel $m_i$
9: end for
10: Agregate all models $m_i$
```

2.4 Case Study 1: K-means

As discussed earlier, the main idea is to subdivide the dataset into chunks that can fit into the main memory space available. Then, K-MLIO [60] applies four steps summarized below, see Figure 2:

Chunk Clustering Step: Each chunk is loaded into the main memory and K-means is run on it. This results in K centroids for each chunk, see Figure 2 ((1) and (2));

Partial Clusters Grouping Step: We ran similarity analysis to group similar clusters from the different K-means models built in order to have an idea about the distribution of the elements in the different clusters among the chunks, see Figure 2 (3);

Final Chunk Composition Step: A data chunk sample is built, representative of the K-means models built beforehand with a size equal to the memory workspace available, see Figure 2 (4);

Final K-means computation: K-means is applied on the sample final chunk to get the final clustering solution for the whole dataset, see Figure 2 (5).

K-MLIO was compared to the traditional K-Means and a version optimized for resource-constrained devices for different N/M proportions, dataset sizes, and separation indexes. As compared to traditional K-Means, K-MLIO proved to be equivalent in terms of accuracy. K-MLIO reduces the execution times by 60-80% for high separation indexes and 35-50% for low ones. As compared to state-of-the-art work, for a fixed execution time, K-MLIO had a 4x better accuracy.

2.5 Case Study 2: Gaussian Mixture Models

Gaussian Mixture models (GMMs) are probabilistic machine learning models that are widely used for cluster analysis [44]. A GMM models a data distribution by a weighted sum of Gaussian densities, where each density is representative of a group in the data, and each weight associated with a density is representative of the number of observations in that group. GMM models are usually trained using the unsupervised Expectation-Maximization (EM) algorithm, where each iteration requires a pass over the entire dataset, and where several iterations are needed for the algorithm to converge. EM training quality depends on the properties of the data and, as a consequence, processing the entire dataset is not always necessary.

PIGMMaLIOn [7, 8], a **P**artial **I**ncremental **G**aussian **M**ixture **M**odel with **a L**ow **I/O** Desig**n**, was built up around two ideas: (1) just like K-means, it subdivides the dataset into chunks (called increments here) and operates a GMM on each increment and merges the results gradually, (2) rather than operating on all the dataset, PIGMMaLIOn processes only a part of the dataset, but still considers a volume that is large enough to get a good accuracy. Our method comes together in three steps:

Data shuffling: The dataset is shuffled to ensure a uniform distribution of all data points.

Dynamic subsampling: The GMM is trained on a subset of data that can fit into the memory (1st increment) to estimate the complexity of the clustering task. This estimation allows for deciding the size of the uniform sub-sample on which the final GMM will be trained. The more complex the learning, the larger the sub-sample. It is based on an online calculation of two parameters: the uncertainty factor, which refers to the proportion of data points that are clustered with relatively low certainty, and the balance factor, which is the ratio between the weight of the smallest cluster and the largest one according to the learned GMM.

Incremental learning: The dataset is divided into increments that can fit into the memory, and the GMM is trained progressively.

PIGMMaLIOn was evaluated against the original EM algorithm. It achieved a time reduction lying between 13% and 85% with an average of 63% with few to no loss of quality (fewer than 5% of experiments showing around 10% of accuracy loss).

2.6 Perspectives

Subdividing the learning process of K-Means and GMM into several incremental steps had a very positive side effect as it made it possible to learn on a limited amount of time by dropping some data chunks, trading off accuracy for learning time [1, 7]. Also, we have explored the ability to leverage DVFS to reduce the energy consumption of the learning process when the number of iterations is lower than expected, all while respecting the learning time budget.

We are working on applying the general solution pattern sketched in Algorithm 2 to different models that obey the general pattern in Algorithm 1. We did so successfully on random forest [61, 62] and we are exploring other algorithms that iteratively span all the dataset, such as boosting algorithms (AdaBoost, XGBoost, LightGBM), neural networks with gradient descent, or SVM.

3 DNN Mapping on the Edge

DNN models are notoriously compute and memory intensive, which complicates their deployment at the edge [53]. Additionally, edge devices are increasingly equipped with heterogeneous accelerators, ranging from systolic arrays to multi-core clusters or in-/near- memory computing units [11, 48, 64], and the need for extreme efficiency makes the edge AI HW ecosystem more fragmented and varied compared to the cloud. This makes deploying DNNs efficiently on edge systems particularly challenging, as it involves solving two interlinked problems: i) optimally *mapping the workload* to maximize HW utilization, accounting for accelerators' heterogeneity, ii) finding an *adapt DNN architecture*, whose geometry, pruning pattern, and data format, match the HW parallelism, sparsity and quantization support. While multiple academic and industrial tools tackle these two challenges separately [14, 23, 24, 29, 43], end-to-end solutions addressing both are rare. This section describes one such open-source toolchain, shown in Figure 3, with two key components: a DNN optimizer (PLiNIO) and a DNN compiler (MATCH). Both tools are designed to carefully balance *ease of use and extensibility*, aimed at enabling fast experimentation, and *HW-awareness* across the stack to maximize efficiency, thanks to a *unified cost modelling abstraction.*

3.1 HW-aware DNN Optimization

PLiNIO [52][1] is a PyTorch-based HW-aware DNN optimization library that focuses on lightweight gradient-based methods. As shown in the top-left of Figure 3, PLiNIO supports a complete stack of optimizations, including Neural Architecture Search (NAS) [42, 55], structured and semi-structured pruning [18, 67] and quantization and mixed-precision search (MPS) [46]. While some of the included optimizations are also available in alternative toolchains [24], others are unique, such as a joint channel-pruning and MPS solution, permitting the simultaneous exploration of layer geometry and precision, yielding up to 20% model size reduction at iso-accuracy w.r.t. optimizing them sequentially, while also reducing training cost [46]. More than by the specific methods supported, however, PLiNIO is characterized by its core design principles.

To ease adoption, it exposes a *unified Python API* for all its optimizations (NAS, pruning, MPS, etc), which hides most of the complexity from users, allowing them to optimize a model with minimal modifications to their training loops, agnostic of the specific technique considered, thus simplifying research and comparison among methods [52]. PLiNIO is also designed with extensibility in mind, with reusable DNN graph analysis or rewriting passes.

PLiNIO provides an easily extensible API to define analytical **cost models** of a DNN's execution (e.g., latency, energy consumption, etc) on a particular HW platform, as a function of layers' geometry, sparsity, quantization format, etc, which in turn depend on the current state of the optimization [52]. With respect to optimizing just for model size or number of OPs, these cost models correlate better with the actual latency/energy, by encoding the HW's spatial parallelism, the bandwidth of different memory levels, etc. The library already includes models for several edge targets [48, 51, 64]. Similarly, PLiNIO simplifies the definition of **regularizers**, i.e., additional terms that must be added to the training loss to perform cost-aware optimization [55]. In particular, the DUCCIO regularizer, first proposed in [10] permits the simultaneous enforcement of *multiple cost constraints* (e.g., maximum memory and maximum latency) both at the level of individual layers and of the entire network. For instance, model size limits ensuring that each layer fits a certain memory level can be combined with global latency contrains. Results demonstrate that this multi-constraint setup permits the discovery of a fully HW-compatible DNN in a *single optimization iteration*, with no loss of accuracy w.r.t. approaches requiring multiple runs [10]. Combined, PLiNIO's unique cost models and regularizers definition allow a finer-grain and more accurate consideration of HW constraints during optimization, thus simplifying the subsequent deployment.

3.1.1 Tackling the deployment challenge at training time. When targeting a heterogeneous edge SoC, a key challenge at deployment time is how to *optimally utilize* multiple on-board accelerators [19]. Classic model parallelism or pipelining approaches work well for homogeneous accelerators, but fail to consider *incompatibilities* in heterogeous SoCs, such as different supported operation types or data formats, which cause model accuracy to be influenced by the mapping. This is a common scenario at the edge, e.g., for systems equipped with both digital and AIMC accelerators, with the lattery usually supporting only aggressively-quantized layers [64]. Our recent work ODiMO [56] (bottom-left of Figure 3) extended PLiNIO to use NAS and MPS techniques for discovering optimized *partitions* of each DNN layer, which can be mapped respectively to accelerators supporting different ops or different quantization formats. These fine-grained intra-layer partitions can be executed in parallel to maximize HW utilization, and being discovered at training time, balance the obtainable throughput or energy with the resulting accuracy (e.g., mapping the most relevant portion of each layer to an accelerator supporting high-precision, and the rest to lower-precision HW). To our knowledge, this is a unique feature of PLiNIO, which leads to up to 8x faster / 50x more efficient mappings w.r.t. heuristic layer-wise ones [56].

3.2 HW-aware DNN Compilation

MATCH [25][2], shown in the top-right of Figure 3, is built on top of Apache TVM [14]. It differentiates from the many other existing AI compilers for the edge [9, 14, 43], as it is designed to bridge the gap between highly specialized, per-platform toolchains, efficient but hardly extensible, and general-purpose compilers, with broad operators support but unable to exploit HW-specific optimizations.

To achieve this goal, MATCH employs a modular, model-driven approach. To support a new target, developers must only specify an *abstract HW model* describing: i) the platform's memory hierarchy; ii) the DNN operators supported by the HW; iii) an approximate

[1]https://github.com/eml-eda/plinio

[2]https://github.com/eml-eda/match

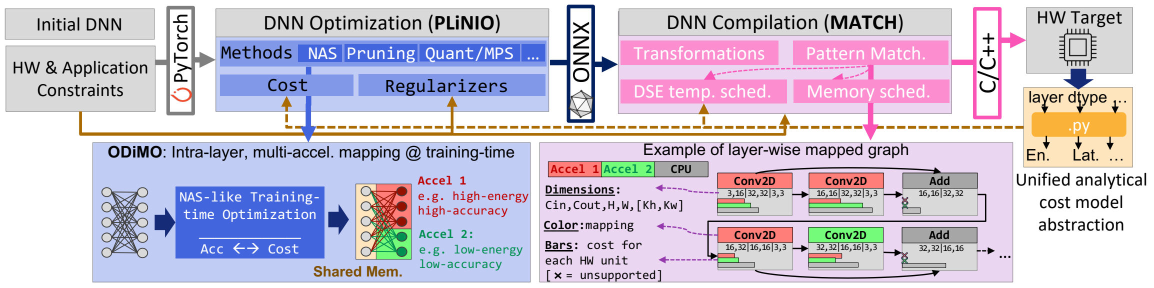

Figure 3: End-to-end DNN deployment flow for heterogeneous edge devices.

cost model for each supported operator, and iv) a set of *SoC-specific APIs* for accelerator control, memory management, etc. Importantly, the cost models in iii) are still analytical, thus entirely compatible with the PLiNIO ones, although usually expressed with a higher level of detail (e.g., carefully accounting for data transfers in a tiled layer execution). This *unified cost modelling* abstraction (Figure 3, right) is a key peculiarity of the toolchain.

Internally, MATCH augments TVM with a hardware-aware design space exploration (DSE) engine, ZigZag [45, 63], and custom graph rewrite passes that ingest the HW model, so that each DNN layer (or operator pattern) is automatically dispatched to the most suitable accelerator. This approach allows the compiler to fully exploit on-chip heterogeneity, i.e., offloading compute-intensive layers to specialized NPUs while using CPU cores for unsupported operators (i.e., not having optimized implementations in any HW unit), all without sacrificing the portability and extensibility of the compilation flow.

MATCH's performance has been demonstrated on two state-of-the-art heterogeneous edge SoCs, both leveraging a RISC-V host CPU. On GAP9 [48], which integrates an additional 8-core RISC-V cluster alongside a programmable CNN accelerator (NE16), MATCH has been shown to cut end-to-end inference times by up to 2.15× w.r.t. the custom DORY compiler [9], by synergistically executing layers on both HW units. On DIANA [65], which features two DNN accelerators (a digital one and an AIMC one), on average, MATCH outperforms by 16.9% a similar tool [20], also based on TVM.

Our current work is directed at extending MATCH to fully support the concurrent use of multiple accelerators, including ODiMO-like intra-layer partitionings (see Section 3.1.1), thus surpassing the classic "one layer - one accelerator" mapping approach. Further, MATCH is being extended to support the automatic deployment of *training graphs* (as opposed to inference-only) on these resource-constrained, OS-less devices. This goes in the direction of personalized on-device learning at the edge, which is becoming increasingly crucial, with a framework that, differently from existing ones [38], fully supports HW heterogeneity.

3.3 The Road Ahead: On-device Adaptation.

Edge AI applications, and deployment tools as a consequence, have been progressively shifting from inference to on-device training [38]. A natural next step is *on-device DNN optimization/adaptation*, i.e., enabling resource-constrained edge devices to autonomously requantize, prune, or change the architecture of their local DNNs. This would adapt AI models to *user- or device-specific data distributions*, alike biological brains, whose structure is modified by the environment [50], possibly yielding orders of magnitude higher efficiency. However, achieving this goal entails significant challenges: optimization passes currently supported by tools like PLiNIO shall be lightened as much as possible to be deployed on memory-limited edge devices, and AI compilers like MATCH shall be extended to support the generation of optimization code. Controlling the start/end of the optimization phase, to balance accuracy and efficiency, is also an open problem. Overall, achieving this level of adaptability will be a key challenge to enable next-generation Edge AI applications, and we believe that automated deployment tools like the ones we presented will play a key role in it.

4 Retrieval-Augmented Generation (RAG) on the Edge

4.1 Overview

With the growing demand for on-device intelligence, enabling Retrieval-Augmented Generation (RAG) on edge devices has become increasingly important to reduce latency, preserve privacy, and ensure offline availability. However, RAG's end-to-end pipeline introduces significant system-level challenges when deployed on resource-constrained platforms. These include high storage I/O latency for retrieving high-dimensional vectors, memory bandwidth saturation during ANN search, and inefficient GPU utilization due to the growing key-value (KV) cache in Transformer decoding. To address these bottlenecks, we propose a set of edge-aware solutions: page-aware prefetching and CPU-side optimizations for I/O efficiency, near-memory processing for scalable vector search, and a system-algorithm co-design to optimize GPU memory management. Together, these techniques lay the groundwork for efficient and scalable RAG inference on edge platforms.

4.2 Background of RAG

Retrieval-Augmented Generation (RAG) enhances language models by incorporating external knowledge retrieval into the generation process, allowing outputs to reflect real-time or domain-specific information. The pipeline begins with a query that triggers an approximate nearest neighbor (ANN) search to retrieve relevant documents, which are then concatenated with the query to form augmented input prompts. These prompts are processed by Transformer models through prefill and decode stages, with the latter relying on

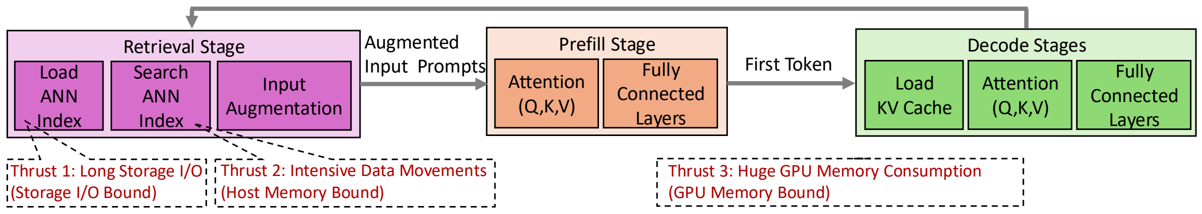

Figure 4: RAG flow.

cached key-value (KV) pairs for efficient token-by-token generation. While effective in large-scale cloud environments, RAG poses significant challenges for edge deployment due to limited memory capacity, constrained compute resources, and slower storage interfaces. Adapting RAG to such environments requires rethinking retrieval strategies, memory management, and model execution to optimize latency, memory footprint, and compute efficiency within the tight resource budgets of mobile and embedded systems.

4.3 Key Challenges

While RAG enhances model performance by enabling dynamic knowledge retrieval, its architecture introduces several system-level bottlenecks, as shown in Figure 4. Thrust 1 highlights the latency and throughput limitations caused by loading large-scale external indexes and documents, which are constrained by storage I/O performance. Thrust 2 points to the intensive data movement between storage, host memory, and accelerator buffers, often becoming a bottleneck in memory-bound systems. Finally, Thrust 3 captures the excessive GPU memory footprint due to long context lengths, augmented input size, and persistent KV cache usage, posing challenges for scaling across larger batches or longer documents. These challenges necessitate co-optimization of hardware, memory, and model execution paths to fully realize RAG's potential.

4.4 Thrust 1: Storage I/O Bottlenecks for Retrieving Multidimensional Vectors

Thrust 1 underscores the critical latency and throughput limitations that emerge when loading large-scale external indexes and documents—an operation fundamental to the retrieval phase of RAG. These indexes often consist of high-dimensional feature vectors representing textual or semantic embeddings [72], which require fast and frequent access to enable effective nearest neighbor search. In practical deployments, especially under ultra-low-latency requirements, the efficiency of retrieving such multidimensional data becomes tightly coupled with the performance characteristics of the underlying storage subsystem [37]. Several software-level and algorithm-level solutions exist.

In addition to software optimizations, emerging ultra-low-latency (ULL) storage devices offer faster data access, yet still suffer from overhead introduced by operating system synchronous I/O interfaces—especially during frequent small reads or random-access retrievals across large embedding tables. To further accelerate I/O performance when storage latency approaches the cost of asynchronous I/O handling, we propose three main solutions. First, we design a virtual-address-based page prefetching mechanism [69, 70] that fetches pages adjacent to the faulting page in virtual memory. Second, we introduce an adaptive synchronous page fault handler that reworks the Linux fault handling path to reduce overhead, improve cache hit rates, and enable memory-aware CPU scheduling [15]. Third, we implement a CPU pre-execution mechanism [71] that speculatively executes future instructions during I/O stalls, using idle CPU cycles to preload cache data—particularly effective at the microsecond scale of page fault latency.

4.5 Thrust 2: Memory-bound while Serving Intensive Vector Computations

Recently, RAG usually adopts cluster-based (e.g., Inverted File) or graph-based (e.g., Hierarchical Navigable Small World graph) ANN search. Generally, the procedure of each ANN search contains an index lookup stage to quickly select a group of neighboring vectors by looking up the indexing structure and a distance calculation stage to keep vectors closer to the query vector among the neighboring vectors. Several profiling results show that concurrent search requests saturate memory bandwidth due to poor CPU cache reuse and low compute intensity, making ANN search engines memory-bound and limiting multi-threaded scalability [13, 35, 39, 40].

To address DRAM bandwidth bottlenecks in vector similarity search, UPMEM's near-memory processing (NMP) device offers a scalable solution with thousands of DRAM Processing Units (DPUs), each with isolated memory. However, naive deployment of ANN search on UPMEM often yields limited gains due to hardware constraints. Our recent work, UPMEM-aware Vector Similarity Search (UPVSS) [40], introduces two key components: a DPU-aware Cluster Partitioning strategy that distributes vectors within IVF clusters while accounting for DPU architecture, and a Search Coordinator that offloads distance computations to DPUs with task pipelining. These designs collectively improve vector processing efficiency and parallelism across the DPU array.

4.6 Thrust 3: Towards Efficient GPU Utilization in Managing KV Cache

RAG process will then augment the original input prompt with vectors selected by the ANN search. After that, the GPU processes augmented input prompts through the Transformer-based LLM blocks, that is, a series of prefill and decode stages [12]. Transformer-based LLMs face a memory challenge due to their autoregressive self-attention, which requires storing all past token representations in high-bandwidth memory (HBM) as key/value (KV) caches. As batch size and sequence length increase, the KV cache grows, limiting GPU utilization and reducing throughput. While Huggingface's Transformers (i.e., HF-Transformers) constantly allocates more memory to the KV cache and stalls the computation until it is complete, trading usability for latency, NVIDIA's FasterTransformer pre-allocates memory for maximum sequence lengths, which improves latency but wastes memory and constrains batch size, making efficient memory management increasingly critical for serving long-sequence models [30, 33, 57].

These limitations underscore the need for adaptive memory management strategies that balance latency, efficiency, and throughput in dynamic-generation scenarios. To tackle these challenges, our design S^3 [30] introduces a system-algorithm co-design that improves GPU utilization by predicting output sequence lengths for smarter memory allocation and batching. By increasing batch size and sharing model weights across inputs, S^3 mitigates inefficiencies caused by unknown sequence lengths. It comprises three components: (1) a predictor that classifies expected output lengths using a fine-tuned DistilBERT, (2) a scheduler that performs length-aware, bin-packing-based batching within GPU HBM constraints, and (3) a supervisor that monitors memory usage and handles mispredictions by preempting and recycling overgrown sequences. Through length-aware preallocation, S^3 reduces redundant memory operations in HF-Transformers and avoids memory waste in FasterTransformer, ultimately boosting throughput and GPU efficiency for RAG workloads.

4.7 Perspective: Toward Scalable and Efficient Edge RAG

Looking ahead, several promising directions can further advance RAG deployment on edge devices. On the algorithmic side, techniques like quantization, pruning, dynamic beam search, and early-exit strategies can reduce memory usage and inference latency. From a system and hardware perspective, future work may explore dynamic model offloading across heterogeneous compute units [57] and memory tiering [28] to balance local and remote execution. Additionally, emerging accelerators such as photonic processors [12] and compute-in-memory architectures present new opportunities for ultra-low-latency, energy-efficient RAG inference. Collectively, these directions aim to make RAG both accurate and scalable on resource-constrained platforms.

5 Graph Processing on the Edge

5.1 Overview

Graphs are a powerful data structure for representing real-world relationships and have been widely adopted across various domains. Over time, their applications have spanned from traditional areas (e.g., social network analysis and recommendation systems) to emerging AI-related fields (e.g., Retrieval-Augmented Generation (RAG) [22] and Graph Neural Network (GNN [73]).

As graph sizes grow exponentially, the demand for efficient large-scale graph processing at low cost has become increasingly critical and challenging, especially for edge devices or computing systems with limited memory. To this end, **graph processing with storage** (a.k.a., *out-of-core graph processing* [34, 41, 74–77]) has emerged as a viable option. As illustrated in Figure 5, this approach offloads the massive graph data into storage and loads parts of them into

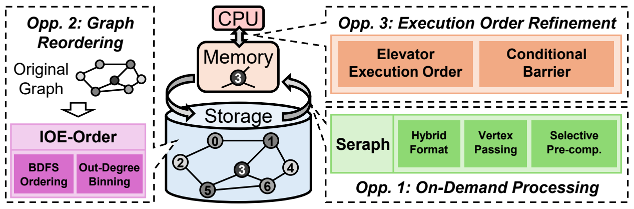

Figure 5: Graph processing with storage overview.

memory for processing on the fly. Nevertheless, despite the merit of reduced cost, the unreasonably-low efficiency often renders it uncompetitive and impractical compared to other graph processing approaches. To address this performance-cost dilemma, this section explores a series of opportunities, which collectively pave a way for enabling cost-effective and high-performance large-scale graph processing across various scenarios and applications.

5.2 Opportunity 1: On-Demand Processing

While existing out-of-core graph processing systems are inherently well-suited for handling ever-growing, large-scale graphs, they often suffer from poor efficiency, making them uncompetitive or even impractical in some real-world scenarios. The root cause lies in a common design choice: most existing graph systems overly prioritize sequential access to graph data to maximize storage bandwidth. Although this approach does deliver high throughput, it frequently results in reading excessive and irrelevant data, significantly reducing overall efficiency in handling large-scale graphs. At the same time, while modern storage (e.g., solid-state drive) offers comparable speeds for sequential and random access, traditional approaches in existing out-of-core systems fail to fully exploit the advantages of modern storage.

To address this, we build *Seraph* [74], an out-of-core graph system based on **on-demand processing**, which retrieves only the necessary graph data, effectively reducing I/O overhead by taking advantage of fast random access. In particular, Seraph employs three practical techniques specially tailored for the framework of on-demand processing to substantially improve the graph processing efficiency. First, we observe that the traditional method for representing edge data has its pros and cons: it creates a good locality for accessing vertices, yet increases the overhead of locating and reading edges. Hence, we present a new format, called *hybrid format*, to store the graph data by striking a good balance between locality for vertices and overhead for edges. Second, based on the hybrid format, we further propose *vertex passing* to enable efficient vertex updates by delaying and aggregating the vertex updates to the same subgraph via in-memory buffers. Third, although on-demand processing reads the necessary data, a common I/O block is typically way larger than an edge list. This mismatch inspires us with the opportunity of I/O reusing and the proposal of *selective pre-computation*, which asynchronously computes the current and future vertices during the same I/O operation.

Implication for AI/ML: Seraph is designed to support a broad range of graph algorithms and can cost-effectively process terabyte-scale graph data using less than 100 GB of memory. This makes it well-suited for resource-constrained environments, where it can support AI/ML pipelines by handling various tasks such as cluster preprocessing [16], neighborhood searching [22], or even generating training data for GNN [26, 73].

5.3 Opportunity 2: Graph Reordering

This section introduces a pre-processing technique, **graph reordering**, to further enhance performance by alleviating the severe I/O bottlenecks inherent in resource-constrained, out-of-core environments. Such I/O bottlenecks come from the poor access locality exhibited by many graph algorithms, which often access only a subset of vertices in a scattered manner. This behavior is especially problematic in out-of-core settings, where the storage access granularity is much larger than the actual data required, forcing the system to issue numerous random I/O requests that hinder performance significantly. Given this observation, we found that the graph's layout plays a critical role in determining I/O efficiency. By applying graph reordering to co-locate vertices and edges that are likely to be accessed together, the system can reduce random access patterns and improve spatial locality, thereby substantially mitigating the I/O bottleneck.

To this end, we present *IOE-Order* [76], an I/O-efficient graph ordering to efficiently pre-process and optimize the graph layout, while better I/O efficiency in loading storage-resident graph data can be delivered at runtime to boost the overall processing efficiency. IOE-Order consists of two main pre-processing steps: ① *Breadth-First Degree-Second (BFDS) Ordering*, which leverages both graph traversal patterns and vertex degree information to co-locate vertices and edges that are likely to be accessed together, thereby improving I/O locality. ② *Out-Degree Binning*, which partitions the BFDS-ordered graph into multiple sorted bins based on the vertices' out-degrees. This step further enhances I/O efficiency during graph processing and enables flexible caching of vertices depending on available memory.

Implication for AI/ML: Overall, IOE-Order enhances I/O efficiency by reorganizing the graph layout to improve spatial locality by using *BFDS Ordering* and *Out-Degree Binning*. By significantly reducing random I/O and improving data locality, IOE-Order holds great potential to benefit various neighborhood-centric tasks commonly found in AI/ML pipelines like random walk [54] and k-hop neighbors extraction [73].

5.4 Opportunity 3: Execution Order Refinement

This section further introduces a runtime optimization called **execution order refinement** that alters the execution order to enhance performance based on the technique of *future value computation* [66]. Specifically, future value computation is a powerful I/O optimization for out-of-core graph systems, as it reuses already-loaded edge data to compute future vertex values in advance, allowing the system to skip previously processed edges in subsequent iterations and thereby reduce redundant I/O. Nevertheless, it is challenging to take full advantage of future value computation.

To address this effectively, we present a new *elevator execution order* [75] to significantly increase the number of future-computed vertices so as to achieve considerable I/O reduction at runtime. We can prove that the elevator execution order achieves the optimal I/O savings possible for future value computation. In addition, unlike prior approaches that rely on expensive global barriers to maintain data dependencies, we introduce a lightweight *conditional barrier* mechanism. It adaptively eliminates unnecessary barriers while ensuring correct execution, reducing computational overhead. Together, these runtime designs substantially improve the efficiency and practicality of future value computation in out-of-core graph processing.

Implication for AI/ML: These runtime designs are also especially beneficial for AI/ML pipelines that involve iterative, message-passing graph processing such as label propagation. For example, [36] shows how label propagation can help address over-smoothing and over-fitting issues in graph-based semi-supervised learning. As a result, our approach not only improves the performance of out-of-core graph systems but also advances graph-centric AI/ML workloads.

5.5 Perspectives: Approximate Graph Processing for AI/ML

While numerous advanced techniques have been proposed to optimize out-of-core graph processing, a fundamental performance limitation of computing exact answers cannot be broken through. However, since exact answers are not always necessary in many AI/ML applications, *approximate graph processing* offers a promising opportunity to trade accuracy for substantial reductions in execution time. Among available approximate techniques, *sketching* stands out as a compelling strategy that *trades cheap storage space for significant computational and energy savings*. Specifically, sketching creates a compact summary (i.e., a sketch) of large data that preserves key properties, and thus, computations can be approximated via sketches at runtime efficiently with provable accuracy guarantees. This behavior makes them especially attractive for resource-constrained environments such as edge devices, where memory, compute, and energy budgets are all limited. Nevertheless, although many sketching techniques have been proposed [2, 3, 17], most of them are only theoretically-sound and difficult to integrate into real-world AI/ML pipelines. As such, applying sketching to AI/ML presents a challenging yet promising direction that could fundamentally enhance the scalability and efficiency.

6 Conclusion

This paper highlights four research directions related to the deployment of Edge AI. While the first contribution set targets the learning phase of foundational ML algorithms, the 3 other sets of contributions discuss different compute-intensive inference models. We presented a set of tools for DNN mapping, a set of optimizations for RAG deployment on the Edge and a set of storage-based optimizations for graph-based processing.

References

[1] Hafsa Kara Achira, Camélia Slimani, and Jalil Boukhobza. 2023. Training K-Means on Embedded Devices: A Deadline-Aware and Energy Efficient Design. In *2023 31st International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 1–8. doi:10.1109/MASCOTS59514.2023.10387589

[2] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. 2012. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (Scottsdale, Arizona,

USA) *(PODS '12)*. Association for Computing Machinery, New York, NY, USA, 5–14. doi:10.1145/2213556.2213560

[3] Takuya Akiba and Yosuke Yano. 2016. Compact and Scalable Graph Neighborhood Sketching. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. Association for Computing Machinery, New York, NY, USA, 685–694. doi:10.1145/2939672.2939762

[4] Rasim M. Alguliyev and Rashid G. Alakbarov. 2023. Constrained k-means algorithm for resource allocation in mobile cloudlets. *Kybernetika* 59, 1 (2023), 88–109. doi:10.14736/kyb-2023-1-0088

[5] Jalil Boukhobza and Pierre Olivier. 2017. *Flash Memory Integration: Performance and Energy Issues* (1st ed.). ISTE Press - Elsevier, London/Oxford, GBR.

[6] Jalil Boukhobza, Pierre Olivier, Wen Sheng Lim, Liang-Chi Chen, Yun-Shan Hsieh, Shin-Ting Wu, Chien-Chung Ho, Po-Chun Huang, and Yuan-Hao Chang. 2025. A Survey on Flash-Memory Storage Systems: A Host-Side Perspective. *ACM Trans. Storage* (March 2025). doi:10.1145/3723167 Just Accepted.

[7] Meriem Bouzouad, Yasmine Benhamadi, Camélia Slimani, and Jalil Boukhobza. 2024. Adapting Gaussian Mixture Model Training to Embedded/Edge Devices: A Low I/O, Deadline-Aware and Energy Efficient Design. *SIGAPP Appl. Comput. Rev.* 24, 2 (Aug. 2024), 5–18. doi:10.1145/3687251.3687252

[8] Meriem Bouzouad, Yasmine Benhamadi, Camelia Slimani, and Jalil Boukhobza. 2024. PIGMMaLIOn: a Partial Incremental Gaussian Mixture Model with a Low I/O Design. In *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing* (Avila, Spain) *(SAC '24)*. Association for Computing Machinery, New York, NY, USA, 428–435. https://doi.org/10.1145/3605098.3635909

[9] Alessio Burrello, Angelo Garofalo, Nazareno Bruschi, Giuseppe Tagliavini, Davide Rossi, and Francesco Conti. 2021. DORY: Automatic end-to-end deployment of real-world DNNs on low-cost IoT MCUs. *IEEE Trans. on Comput.* (2021).

[10] Alessio Burrello, Matteo Risso, Beatrice Alessandra Motetti, Enrico Macii, Luca Benini, and Daniele Jahier Pagliari. 2024. Enhancing Neural Architecture Search With Multiple Hardware Constraints for Deep Learning Model Deployment on Tiny IoT Devices. *IEEE Transactions on Emerging Topics in Computing* 12, 3 (2024), 780–794. doi:10.1109/TETC.2023.3322033

[11] Michele Caon, Clément Choné, Pasquale Davide Schiavone, Alexandre Levisse, Guido Masera, Maurizio Martina, and David Atienza. 2025. Scalable and RISC-V Programmable Near-Memory Computing Architectures for Edge Nodes. *IEEE Transactions on Emerging Topics in Computing* (2025), 1–15. doi:10.1109/TETC.2025.3555869

[12] Wen-Tse Chang, Chun-Feng Wu, and Yun-Chen Lo. 2025. P-DAC: Power-Efficient Photonic Accelerators for LLM Inference. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference*. 1–6.

[13] Sitian Chen, Amelie Chi Zhou, Yucheng Shi, Yusen Li, and Xin Yao. 2024. MemANNS: Enhancing Billion-Scale ANNS Efficiency with Practical PIM Hardware. *arXiv preprint arXiv:2410.23805* (2024).

[14] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. arXiv:1802.04799

[15] Yin-Chiuan Chen, Chun-Feng Wu, Yuan-Hao Chang, and Tei-Wei Kuo. 2022. Exploring synchronous page fault handling. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 3791–3802.

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. *CoRR* abs/1905.07953 (2019). arXiv:1905.07953 http://arxiv.org/abs/1905.07953

[17] Edith Cohen. 2014. All-distances sketches, revisited: HIP estimators for massive graphs analysis. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Snowbird, Utah, USA) *(PODS '14)*. Association for Computing Machinery, New York, NY, USA, 88–99. doi:10.1145/2594538.2594546

[18] Francesco Daghero, Daniele Jahier Pagliari, Francesco Conti, Luca Benini, Massimo Poncino, and Alessio Burrello. 2025. Lightweight Software Kernels and Hardware Extensions for Efficient Sparse Deep Neural Networks on Microcontrollers. In *Eighth Conference on Machine Learning and Systems (MLSyS)*.

[19] Ismet Dagli, Alexander Cieslewicz, Jedidiah McClurg, and Mehmet E. Belviranli. 2022. AxoNN: energy-aware execution of neural network inference on multi-accelerator heterogeneous SoCs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 1069–1074. doi:10.1145/3489517.3530572

[20] Josse Van Delm et al. 2023. HTVM: Efficient Neural Network Deployment On Heterogeneous TinyML Platforms. In *Proc. 60th ACM/IEEE Design Autom. Conf. (DAC)*. 1–6.

[21] Shuiguang Deng, Hailiang Zhao, Weijia Fang, Jianwei Yin, Schahram Dustdar, and Albert Y. Zomaya. 2020. Edge Intelligence: The Confluence of Edge Computing and Artificial Intelligence. *IEEE Internet of Things Journal* 7, 8 (Aug. 2020), 7457–7469. doi:10.1109/jiot.2020.2984887

[22] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2025. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130 [cs.CL] https://arxiv.org/abs/2404.16130

[23] Giuseppe Franco, Alessandro Pappalardo, and Nicholas J Fraser. 2025. *Xilinx/brevitas*. doi:10.5281/zenodo.3333552

[24] Yury Gorbachev, Mikhail Fedorov, Iliya Slavutin, Artyom Tugarev, Marat Fatekhov, and Yaroslav Tarkan. 2019. OpenVINO Deep Learning Workbench: Comprehensive Analysis and Tuning of Neural Networks Inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*.

[25] Mohamed Amine Hamdi, Francesco Daghero, Giuseppe Maria Sarda, Josse Van Delm, Arne Symons, Luca Benini, Marian Verhelst, Daniele Jahier Pagliari, and Alessio Burrello. 2025. MATCH: Model-Aware TVM-based Compilation for Heterogeneous Edge Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025).

[26] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *CoRR* abs/1706.02216 (2017). arXiv:1706.02216 http://arxiv.org/abs/1706.02216

[27] Bing He, FengXiang Qiao, Weijun Chen, and Ying Wen. 2018. Fully convolution neural network combined with K-means clustering algorithm for image segmentation. In *Tenth International Conference on Digital Image Processing (ICDIP 2018)*, Xudong Jiang and Jenq-Neng Hwang (Eds.), Vol. 10806. International Society for Optics and Photonics, SPIE, 108062O. doi:10.1117/12.2502814

[28] Junliang Hu, Zhisheng Hu, Chun-Feng Wu, and Ming-Chang Yang. 2025. Demeter: A Scalable and Elastic Tiered Memory Solution for Virtualized Cloud via Guest Delegation. In *Proceedings of the ACM SIGOPS 31th Symposium on Operating Systems Principles*.

[29] Shawn Hymel, Colby Banbury, Daniel Situnayake, Alex Elium, Carl Ward, Mat Kelcey, Mathijs Baaijens, Mateusz Majchrzycki, Jenny Plunkett, David Tischler, Alessandro Grande, Louis Moreau, Dmitry Maslov, Artie Beavis, Jan Jongboom, and Vijay Janapa Reddi. 2023. Edge Impulse: An MLOps Platform for Tiny Machine Learning. arXiv:2212.03332 [cs.DC]

[30] Yunho Jin, Chun-Feng Wu, David Brooks, and Gu-Yeon Wei. 2023. S^3: Increasing GPU Utilization during Generative Inference for Higher Throughput. *Advances in Neural Information Processing Systems* 36 (2023), 18015–18027.

[31] Oumayma Jouini, Kaouthar Sethom, Abdallah Namoun, Nasser Aljohani, Meshari Huwaytim Alanazi, and Mohammad N. Alanazi. 2024. A Survey of Machine Learning in Edge Computing: Techniques, Frameworks, Applications, Issues, and Research Directions. *Technologies* 12, 6 (2024). doi:10.3390/technologies12060081

[32] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient machine learning in 2 KB RAM for the internet of things. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 1935–1944.

[33] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 611–626.

[34] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 31–46. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/kyrola

[35] Yiwei Li, Yuxin Jin, Boyu Tian, Huanchen Zhang, and Mingyu Gao. 2025. ANSMET: Approximate Nearest Neighbor Search with Near-Memory Processing and Hybrid Early Termination. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 1093–1107.

[36] Zhao Li, Yixin Liu, Zhen Zhang, Shirui Pan, Jianliang Gao, and Jiajun Bu. 2020. Cyclic Label Propagation for Graph Semi-supervised Learning. arXiv:2011.11860 [cs.LG] https://arxiv.org/abs/2011.11860

[37] Che-Wei Lin and Chun-Feng Wu. 2024. ALISA: An Adaptive Learned Index Structure for Spatial Data on Solid-State Drives. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*. 1–9.

[38] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. 2022. On-Device Training Under 256KB Memory. In *Annual Conference on Neural Information Processing Systems (NeurIPS)*.

[39] Chaoqiang Liu, Haifeng Liu, Dan Chen, Yu Huang, Yi Zhang, Wenjing Xiao, Xiaofei Liao, and Hai Jin. 2025. HeterRAG: Heterogeneous Processing-in-Memory Acceleration for Retrieval-augmented Generation. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 884–898.

[40] Chun-Chien Liu, Chun-Feng Wu, and Yunho Jin. 2025. UPVSS: Jointly Managing Vector Similarity Search with Near-Memory Processing Systems. In *Proceedings of the 62nd ACM/IEEE Design Automation Conference*. 1–6.

[41] Hang Liu and H. Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 285–300. https://www.usenix.org/conference/fast17/technical-sessions/presentation/liu

[42] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2019. DARTS: Differentiable Architecture Search. arXiv:1806.09055

[43] Hsin-I. Cindy Liu, Marius Brehler, Mahesh Ravishankar, Nicolas Vasilache, Ben Vanik, and Stella Laurenzo. 2022. TinyIREE: An ML Execution Environment for Embedded Systems from Compilation to Deployment. doi:10.48550/arXiv.2205.14479 arXiv:2205.14479 [cs]

[44] Geoffrey J McLachlan, Sharon X Lee, and Suren I Rathnayake. 2019. Finite mixture models. *Annual review of statistics and its application* 6 (2019), 355–378.

[45] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. 2021. ZigZag: Enlarging Joint Architecture-Mapping Design Space Exploration for DNN Accelerators. *IEEE Trans. on Comput.* 70, 8 (Feb. 2021), 1160–1174.

[46] Beatrice Alessandra Motetti, Matteo Risso, Alessio Burrello, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. 2024. Joint Pruning and Channel-Wise Mixed-Precision Quantization for Efficient Deep Neural Networks. *IEEE Trans. Comput.* 73, 11 (2024), 2619–2633. doi:10.1109/TC.2024.3449084

[47] Onur Mutlu, Saugata Ghose, and Rachata Ausavarungnirun. 2018. Recent Advances in Overcoming Bottlenecks in Memory Systems and Managing Memory Resources in GPU Systems. arXiv:1805.06407 [cs.AR] https://arxiv.org/abs/1805.06407

[48] Hanna Müller, Victor Kartsch, and Luca Benini. 2024. GAP9Shield: A 150GOPS AI-capable Ultra-low Power Module for Vision and Ranging Applications on Nano-drones. arXiv:2407.13706 [cs.RO] https://arxiv.org/abs/2407.13706

[49] Peter O. Olukanmi, Fulufhelo Nelwamondo, and Tshilidzi Marwala. 2018. k-Means-Lite: Real Time Clustering for Large Datasets. In *2018 5th International Conference on Soft Computing Machine Intelligence (ISCMI)*. 54–59. doi:10.1109/ISCMI.2018.8703210

[50] Maya Opendak and Elizabeth Gould. 2015. Adult Neurogenesis: A Substrate for Experience-Dependent Change. *Trends in Cognitive Sciences* 19, 3 (March 2015), 151–161. doi:10.1016/j.tics.2015.01.001

[51] Gianmarco Ottavi, Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. 2020. A Mixed-Precision RISC-V Processor for Extreme-Edge DNN Inference. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 512–517. doi:10.1109/ISVLSI49217.2020.000-5

[52] Daniele Jahier Pagliari, Matteo Risso, Beatrice Alessandra Motetti, and Alessio Burrello. 2023. PLiNIO: A User-Friendly Library of Gradient-Based Methods for Complexity-Aware DNN Optimization. In *2023 Forum on Specification & Design Languages (FDL)*. 1–8. doi:10.1109/FDL59689.2023.10272045

[53] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D. Lawrence. 2022. Challenges in Deploying Machine Learning: A Survey of Case Studies. *ACM Comput. Surv.* 55, 6, Article 114 (Dec. 2022), 29 pages. doi:10.1145/3533378

[54] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA) *(KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. doi:10.1145/2623330.2623732

[55] Matteo Risso, Alessio Burrello, Francesco Conti, Lorenzo Lamberti, Yukai Chen, Luca Benini, Enrico Macii, Massimo Poncino, and Daniele Jahier Pagliari. 2023. Lightweight Neural Architecture Search for Temporal Convolutional Networks at the Edge. 744-758 pages. doi:10.1109/TC.2022.3177955

[56] Matteo Risso, Alessio Burrello, and Daniele Jahier Pagliari. 2025. Optimizing DNN Inference on Multi-Accelerator SoCs at Training-time. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2025), 1–1. doi:10.1109/TCAD.2025.3543715

[57] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. 2023. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*. PMLR, 31094–31116.

[58] Jamie Shotton, Sebastian Nowozin, Toby Sharp, John Winn, Pushmeet Kohli, and Antonio Criminisi. 2013. Decision jungles: compact and rich models for classification. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'13)*. Curran Associates Inc., Red Hook, NY, USA, 234–242.

[59] Raghubir Singh and Sukhpal Singh Gill. 2023. Edge AI: A survey. *Internet of Things and Cyber-Physical Systems* 3 (2023), 71–92. doi:10.1016/j.iotcps.2023.02.004

[60] Camélia Slimani, Stéphane Rubini, and Jalil Boukhobza. 2019. K -MLIO: Enabling K -Means for Large Data-Sets and Memory Constrained Embedded Systems. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 262–268. doi:10.1109/MASCOTS.2019.00037

[61] Camélia Slimani, Chun-Feng Wu, Yuan-Hao Chang, Stéphane Rubini, and Jalil Boukhobza. 2021. RaFIO: a random forest I/O-aware algorithm. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 521–528. doi:10.1145/3412841.3441932

[62] Camélia Slimani, Chun-Feng Wu, Stéphane Rubini, Yuan-Hao Chang, and Jalil Boukhobza. 2023. Accelerating Random Forest on Memory-Constrained Devices Through Data Storage Optimization. *IEEE Trans. Comput.* 72, 6 (June 2023), 1595–1609. doi:10.1109/TC.2022.3215898

[63] Arne Symons, Linyan Mei, and Marian Verhelst. 2021. LOMA: Fast Auto-Scheduling on DNN Accelerators through Loop-Order-based Memory Allocation. In *IEEE 3rd Int. Conf. on Art. Int. Circ. and Sys. (AICAS)*. 1–4.

[64] Kodai Ueyoshi, Ioannis A. Papistas, Pouya Houshmand, Giuseppe M. Sarda, Vikram Jain, Man Shi, Qilin Zheng, Sebastian Giraldo, Peter Vrancx, Jonas Doevenspeck, Debjyoti Bhattacharjee, Stefan Cosemans, Arindam Mallik, Peter Debacker, Diederik Verkest, and Marian Verhelst. 2022. DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 1–3. doi:10.1109/ISSCC42614.2022.9731716

[65] Kodai Ueyoshi, Ioannis A. Papistas, Pouya Houshmand, Giuseppe M. Sarda, Vikram Jain, Man Shi, Qilin Zheng, Sebastian Giraldo, Peter Vrancx, Jonas Doevenspeck, Debjyoti Bhattacharjee, Stefan Cosemans, Arindam Mallik, Peter Debacker, Diederik Verkest, and Marian Verhelst. 2022. DIANA: An End-to-End Energy-Efficient Digital and ANAlog Hybrid Neural Network SoC. In *IEEE Int. Solid-State Circ. Conf. (ISSCC)*, Vol. 65.

[66] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 429–442. https://www.usenix.org/conference/atc19/presentation/vora

[67] Alvin Wan, Xiaoliang Dai, Peizhao Zhang, Zijian He, Yuandong Tian, Saining Xie, Bichen Wu, Matthew Yu, Tao Xu, Kan Chen, et al. 2020. Fbnetv2: Differentiable neural architecture search for spatial and channel dimensions. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 12965–12974.

[68] Yan Wang, Zihang Lai, Gao Huang, Brian H. Wang, Laurens van der Maaten, Mark Campbell, and Kilian Q. Weinberger. 2019. Anytime Stereo Image Depth Estimation on Mobile Devices. In *2019 International Conference on Robotics and Automation (ICRA)*. 5893–5900. doi:10.1109/ICRA.2019.8794003

[69] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. Joint management of CPU and NVDIMM for breaking down the great memory wall. *IEEE Trans. Comput.* 69, 5 (2020), 722–733.

[70] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2020. When storage response time catches up with overall context switch overhead, what is next? *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 4266–4277.

[71] Chun-Feng Wu, Yuan-Hao Chang, Ming-Chang Yang, and Tei-Wei Kuo. 2024. How to steal CPU idle time when synchronous I/O mode becomes promising. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[72] Chun-Feng Wu, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. 2022. A joint management middleware to improve training performance of deep recommendation systems with SSDs. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 157–162.

[73] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596 http://arxiv.org/abs/1901.00596

[74] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. USENIX Association, Santa Clara, CA, 373–387. https://www.usenix.org/conference/fast24/presentation/yang-tsun-yu

[75] Tsun-Yu Yang, Cale England, Yi Li, Bingzhe Li, and Ming-Chang Yang. 2024. Grafu: Unleashing the Full Potential of Future Value Computation for Out-of-core Synchronous Graph Processing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 467–481. doi:10.1145/3620665.3640409

[76] Tsun-Yu Yang, Yuhong Liang, and Ming-Chang Yang. 2022. Practicably Boosting the Processing Performance of BFS-like Algorithms on Semi-External Graph System via I/O-Efficient Graph Ordering. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 381–396. https://www.usenix.org/conference/fast22/presentation/yang

[77] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. 2015. FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 45–58. https://www.usenix.org/conference/fast15/technical-sessions/presentation/zheng

[78] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing. *Proc. IEEE* 107, 8 (2019), 1738–1762. doi:10.1109/JPROC.2019.2918951