Ph.D. Dissertation

Hardware/Software Techniques for Efficient Embedding Operations in AI Applications

인공지능 어플리케이션의 효율적인 임베딩 연산을 위한 하드웨어 및 소프트웨어 기술

August 2023

Department of Computer Science & Engineering
College of Engineering
Seoul National University

Yejin Lee

Hardware/Software Techniques for Efficient Embedding Operations in AI Applications

Advisor Jae W. Lee

Submitting a Ph.D. Dissertation of Engineering

May 2023

Department of Computer Science & Engineering
College of Engineering
Seoul National University

Yejin Lee

Confirming the Ph.D. Dissertation written by Yejin Lee

May 2023

Chair	Sungjoo Yoo	(Seal)
Vice Chair	Jae W. Lee	(Seal)
Examiner	Youngki Lee	(Seal)
Examiner	Jaewoong Sim	(Seal)
Examiner	Hongil Yoon	(Seal)

Abstract

The capacity of deep neural networks (DNNs) is rapidly growing to capture intricate connections between entities within a dataset, while the size of the dataset is also growing large and complex to encompass complicated information from the real world. To represent such complex nature of the dataset, embedding is commonly employed. Embedding is a vector projection of a high-dimensional sparse feature space to a low-dimensional dense space that preserves the semantics of the original features. This facilitates the ability of DNNs to process the dataset efficiently as an input.

The growth in the size of both models and datasets results in a high overhead of embedding operations leading to the overall slowdown in emerging AI applications such as recommender systems, NLP, and computer vision. As a result, these applications are facing three primary challenges: 1) an increase in memory bandwidth requirement 2) an increase in computational demand 3) inefficiency of the hardware. This dissertation explores three different AI applications experiencing these challenges and identifies different optimization opportunities for different embedding operations. Based on the findings, this dissertation introduces three proposals to address these challenges.

First, we present a new memoization framework called MERCI to optimize the embedding reduction operation in recommender systems. This novel memoization framework utilizes the co-appearing structure among features in a real-world dataset. To accomplish this, we introduce Correlation-Aware Variable-Sized Clustering, which identifies frequently co-appearing clusters of features with high coverage and small memoization table size. We also propose a feature remapping scheme to efficiently locate a partially reduced embedding using a small number of instructions.

Second, we introduce a software-hardware co-design work called ELSA for efficient self-attention operation which is widely used in natural language processing (NLP), recommender systems, and computer vision and etc. ELSA

proposes a novel approximate self-attention algorithm that efficiently identifies relationships that are unlikely to impact the final outcome and skips computations related to them. We also introduce a hardware accelerator for our approximation algorithm to gain substantial speedup and energy saving by reduced computation.

Third, we present ANNA, a specialized architecture designed to accelerate compression-based approximate nearest neighbor search (ANNS). ANNA addresses the inefficiencies of commodity hardware (e.g., CPUs and GPUs) with carefully designed hardware modules and fine-grained pipelining. ANNA flexibly supports various search configurations and different ANNS libraries from large companies like Meta and Google. Also, ANNA employs a memory traffic optimization technique to support large-scale datasets more efficiently.

These proposals are highly relevant for practical use cases where models are deployed in data centers to provide services to end users. They can assist in meeting strict throughput and energy constraints associated with such use cases. Additionally, the findings of this research can serve as a valuable reference for future research and development in the field of AI.

Keywords: Machine Learning, Embedding Operation, Embedding Reduction, Self-Attention, Compression-based Approximate Nearest Neighbor Search, SW/HW Co-design, HW Accelerator, SW Optimization

Student Number: 2018-25551

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The capacity of deep neural networks (DNNs) is rapidly scaling to learn more complex and implicit relationships between entities within a dataset. At the same time, the size of the dataset is also growing rapidly to include varying and complex information in the real world. *Embedding* is one of the most popular data types widely used to represent such complicated nature of the dataset [21, 57, 138, 139, 168, 178]. It is a vector projection of high-dimensional sparse feature space to a low-dimensional dense space that preserves the semantics of the original features, for DNNs to process it easily as an input.

As a result of the rapid growth in the size of both models and datasets, many emerging AI applications tend to spend a significant portion of runtime on embedding operations. Embedding operations cause the applications to face several critical challenges in providing quality results at high speed. Considering practical use cases where models are deployed at the data center to provide services to the end users, it is very important to solve these challenges to meet the throughput and energy constraints on such use cases. Below, we introduce three primary challenges that emerging AI applications are facing, which are caused by embedding operations.

The first challenge is an increase in memory bandwidth consumption. The development of memory bandwidth has been much slower than that of computation power. Recently, as the size of the model and the dataset grows rapidly, emerging AI applications, which especially rely heavily on embedding operations, are consuming a lot of memory bandwidth making them to be easily bounded by memory bandwidth. Thus, it is crucial to ensure the required

memory bandwidth at least because otherwise the performance is bounded by the memory bandwidth no matter how many computing resources there are.

The second challenge is an increase in computation. As the amount of required computation grows in proportion to the size of the models and datasets, the computational challenge is one of the emergent challenges to address considering recent trends of rapidly advancing AI. Additionally, the end of Moore's Law has made the cost of computation more expensive than in the past, making the reduction of computation essential in terms of cost-saving.

The third challenge is the inefficiency of the hardware. Recently, hardware has been getting bigger and more expensive to support complex features. However, emerging AI applications often do not take advantage of these resources, resulting in a waste of cost and energy. Thus, producing customized hardware accelerators, which are highly designed to be optimized for the given application, could highly contribute to improving performance and energy efficiency.

The dissertation will explore three emerging AI applications that are facing these primary challenges. Different AI applications perform different embedding operations to achieve different goals, leading to unique optimization challenges. Through a detailed analysis, the dissertation characterizes each application's embedding operation and performs performance bottleneck analysis to identify optimization opportunities. By identifying the challenge that each operation faces, this dissertation aims to provide insights into the optimization opportunities for emerging AI applications. The findings of this dissertation can inform future research and development in the field of AI, particularly regarding the use of embedding to address the challenges posed by the increase in model and dataset sizes.

First, we propose MERCI to optimize embedding reduction operation in recommender systems. MERCI is a novel memoization framework that exploits co-appearing structure among features in a real-world dataset. We introduce Correlation-Aware Variable-Sized Clustering to identify clusters of frequently co-appearing features of variable length with high coverage and small memorization table size, as well as a feature remapping scheme to quickly locate a

partially reduced embedding with a small number of instructions.

Second, we propose ELSA to optimize self-attention operation, which is widely used in natural language processing (NLP), recommender systems, computer vision and etc. ELSA is a software-hardware codesign work that proposes a novel approximate self-attention algorithm that efficiently identifies relatively less important computations and hardware accelerator that supports skipping such computations to maximize the benefit of approximation.

Lastly, we propose ANNA for compression-based approximate nearest neighbor search (ANNS) acceleration. ANNA is a specialized architecture designed carefully to overcome the inefficiencies in commodity hardwares (e.g., CPUs and GPUs) and to scale well to support billion-scale ANNS. Also, ANNA adopts a memory traffic optimization technique to reduce memory traffic.

The contribution of the dissertation is summarized as follows:

- We identify opportunities for applying memoization to efficient embedding reduction for the first time (Chapter 3.2). Based on this, we introduce Correlation-Aware Variable-Sized Clustering, a novel clustering scheme that carefully weighs the benefits and costs of memoization to form clusters of co-appearing features to memorize (Chapter 3.4).

- We present MERCI, a memoization framework for efficient embedding reduction. MERCI utilizes Correlation-Aware Variable-Sized Clustering, feature ID remapping for efficient query processing to maximize the benefit of memorization (Chapter 3.3).

- We prototype MERCI on commodity platforms to demonstrate its effectiveness in reducing the number of memory accesses 44% (29%), which translates to substantial throughput gains 40.2% (28.6%) and energy savings at the expense of 8× (1×) additional memory usage (Chapter 3.7).

- We present a novel self-attention approximation algorithm composed of hardware-friendly similarity computation to substantially reduce the amount of computation in the self-attention operation (Chapter 4.3).

- We design ELSA, a specialized hardware accelerator that exploits opportunities for approximation and parallelism in the self-attention to significantly improve its performance and energy efficiency (Chapter 4.4).

- We compare ELSA with multiple representative self-attention-oriented models to demonstrate that we can achieve substantial performance improvements (58.1×) and over three orders of magnitude improvements in energy efficiency over the conventional hardware while maintaining less than 1% loss in the accuracy metric. (Chapter 4.5).

- We provide an in-depth analysis of PQ-based ANNS and its inefficiencies on conventional CPUs and GPUs (Chapter 2.3.3). Based on this, we present ANNA, a specialized architecture for ANNS which can accelerate compression-based ANNS algorithms such as Facebook Faiss [103] and Google ScaNN [77] (Chapter 5.2).

- We introduce a memory traffic optimization technique, which substantially reduces the memory traffic through data reuse (Chapter 5.3).

- We evaluate ANNA on multiple billion-scale workloads. ANNA improves the energy efficiency by multiple orders of magnitude (97×+), the search throughput (2.3-61.6×) and latency (4.3-82.1×) compared to the conventional CPU and GPU (Chapter 5.4).

In summary, the proposals in this dissertation have significant practical implications for the deployment of AI models in data centers to provide services to end users. By addressing the challenges related to throughput and energy constraints, these proposals can contribute to more efficient and effective AI systems. Moreover, the insights gained from this research can serve as a valuable resource for future efforts to improve AI applications, particularly with respect to embedding techniques and their application in handling large and complex datasets. Ultimately, the outcomes of this work have the potential to enable faster, more accurate, and more scalable AI systems.

Chapter 2

Background and Motivation

2.1 Embedding

A variety of modern machine learning and neural network algorithms represent an entity as a learned high-dimensional vector or *embedding*. An embedding is a relatively low-dimensional continuous vector that often represents a single categorical feature.

For example, in natural language processing (NLP) models [57, 122], embedding generally represents a word. Without embedding, each word in a corpus would be assigned a dimension and is represented as a one-hot vector, resulting in an extremely sparse vector. Even worse, such representation does not contain semantic or syntactic relations between words; hence, in many cases, a neural network (NN) model maps this sparse vector to a low-dimensional dense vector that captures a word's semantic meaning and keeps similar words in the close distance as Word2Vec [135] does. Such processing makes it easier for machine learning (ML) systems to infer the meaning of input.

Such a wide range of expressiveness of embeddings made them a common data structure for an entity, especially in emerging AI applications. Another use case of embedding is recommender systems [139, 178]. For instance, a recommender system at an online shopping website often employs a NN model to learn the semantics of products and create an embedding for each product. These embeddings are trained to extract the semantics of features (i.e., products); therefore, co-appearing products (e.g., a notebook and a pencil) will appear relatively adjacent in the embedding space. These embeddings are

later used as categorical feature inputs to the recommender system. In practice, modern recommender systems utilize an extensive range of embeddings for different categorical features. One example of a user-related categorical feature is a set of products a user has recently browsed.

Also, other use cases include conventional search engines such as Microsoft Bing [2] and multimedia search services where a user provides image or audio input to find similar ones. In such search applications, an entity could indicate a document, audio, image [21, 166], or video [134].

2.2 Embedding Operations

Various emerging AI applications adopt embeddings for different purposes. For each purpose, embeddings are processed differently with different operations. In this section, we go through how each operation works.

2.2.1 Embedding Reduction

Embedding reduction is widely used in various applications. A typical use case is a personalized recommender system where it performs embedding reduction to represent users or products. For instance, in a recommender system for an online shopping website, each product on sale can be treated as a feature and be represented with distinct embedding. An online user could be represented with multiple categories (e.g., age, gender, location, browsing history, etc.). In many cases, multiple features can constitute a single categorical variable. For example, browsing history may have multiple features in cases where a user has multiple recently browsed products or multiple favorite brands. To represent these categories as a single embedding vector, *embedding reduction* operation is performed. This operation loads an embedding vector for each feature (e.g., a single product), and performs a reduction operation (e.g., sum, average, max, inner product) on them.

In NLP models, embedding reduction generates a sentence or a document embedding by aggregating embedding vectors for words in a given sentence or

```
1  // N: Number of embeddings
2  // D: Dimension of each embedding vector
3  float emb_table[N][D];
4  def embedding_reduction (vector<int> query[B], float &res[B][D]):
5    for qid=0 to B-1:
6      for i=0 to query[qid].size()-1:
7        int cur_feature = query[qid][i];
8        float[D] embedding_vec = emb_table[cur_feature];
9        /* Embedding Reduction */
10       for d = 0 to D-1:
11         res[qid][d] += embedding_vec[d];
```

Figure 2.1: Pseudocode of Embedding Reduction.

document [114]. Many other ML models also adopt embedding reduction [29, 39, 47, 76], and all popular NN frameworks like Tensorflow (`embedding_lookup_sparse(...)`) [11], PyTorch (`EmbeddingBag(...)`) [150], and Caffe2 (`SparseLengthsSum`) [62] support this operation.

Figure 2.1 shows the pseudocode of embedding reduction with the sum operator. The embedding reduction operation takes a set of queries (a batch of queries) and iterates over each feature in each query (Line 6-7). For each feature, it retrieves the corresponding embedding vector from the embedding table (Line 8-9) and performs an element-wise reduction for the vector (Line 11-12). The result of the reduction for each query (i.e., `res`) is the output of this operation. Note that, although sum reduction is adopted here, other reduction operators such as min, max, or inner product can be employed.

2.2.2 Self-Attention

The attention mechanism is a relatively recently introduced neural network primitive emerging as one of the most influential ideas in the deep learning community. This mechanism allows neural networks (NNs) to identify the information relevant to the specific input and decide where to *attend*. For example, this mechanism can be used to identify the portion of the information that is relevant to the query from an extensive collection of data (e.g., knowledgebase, image). One specific case of the attention mechanism is the self-attention mechanism, where the attention mechanism is used to identify

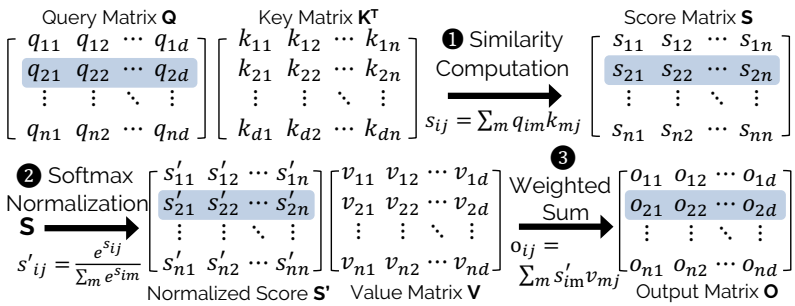


Figure 2.2: Self-attention Mechanism.

the relations among input data. Since its first introduction in the seminal paper *Attention Is All You Need* [188] that presents the Transformer NN architecture, the self-attention mechanism has been widely used to lead the breakthroughs in the field of natural language processing (NLP). Self-attention-oriented NLP models from major AI companies such as Google BERT [57], Meta RoBERTa [122], OpenAI GPT2/3 [27, 155], NVIDIA MegatronLM [168], and Microsoft Turing-NLG [163] established the state-of-the-art results for various NLP tasks. In addition to natural language processing, the self-attention is widely used for computer vision [22, 45, 149, 206] and recommendation systems [63, 107, 171, 178, 214, 215] as well.

Self-attention is essentially an operation that identifies the relations within the input entities, and Fig. 2.2 presents the required computations for it. For each input entity, three different d-dimensional dense vector representations need to be provided: query, key, and value. Assuming the input has n entities, n vectors of d dimension are grouped to form the query matrix ($\mathbf{Q}$), the key matrix ($\mathbf{K}$), and the value matrix ($\mathbf{V}$) each having $n \times d$ dimensions. Throughout the paper, we call row vectors of these matrices queries, keys, and values, respectively. ❶ The very first step of self-attention is similarity computation, which computes the dot product similarity between each query vector and each key vector. For this purpose, the query matrix is multiplied with the transposed key matrix ($\mathbf{QK}^T$). This results in $n \times n$ matrix (i.e., attention score matrix $\mathbf{S}$), where s_{ij} represents the similarity (i.e., dot product) between the ith query and the jth key vector. Note that some implementa-

tions often called *scaled* self-attention divide the resulting matrix by a scalar constant. ❷ The second step is the softmax normalization for each row of the attention score matrix ($s'_{ij} = e^{s_{ij}} / \sum_m e^{s_{im}}$). ❸ The final step computes the output of this operation for each query vector by computing the weighted sum of value matrix ($\mathbf{V}$) rows utilizing the corresponding normalized attention scores as weights. This is equivalent to multiplying the matrix $\mathbf{S}'$ to $\mathbf{V}$ (because $\text{row}_i(O) = \sum_{m=1}^{n} s'_{im} \cdot \text{row}_m(V) \Leftrightarrow o_{ij} = \sum_{m=1}^{n} s'_{im} \cdot v_{mj}$). The result of this is an output matrix ($\mathbf{O}$) where ith row represents the d-dimensional vector that represents the outcome of the self-attention operation for the ith input entity.

Application-Level Description. Each input entity (e.g., a word in a text) gets three different vector representations (query, key, and value). Then, each entity uses its query representation to find the set of other entities that are the most relevant to the current entity. For this purpose, the dot product similarity between the query representation (of the current entity) and the key representation of other entities are computed, then softmax-normalized. Since the softmax function is a differentiable approximation of the `argmax` function, this step is effectively selecting a few most similar entities to the current entity. Finally, the value representations of the selected entries are summed up utilizing the softmax-normalized attention score as the weights. This process is repeated for each input entity, and the output is passed to the next layer in a NN model. In NLP models, this operation is used to identify the specific semantic relation between tokens (e.g., words). For example, a self-attention head (i.e., sub-layer) in a layer lets the *direct objects* to attend their *verbs*, or *noun modifiers* to attend their *nouns* [43].

2.2.3 Product Quantization (PQ) Similarity Search

Similarity search is a task of finding top-k embeddings in a given set (i.e., database vectors) that are the most similar to a query embedding vector in the vector space. It is also called the nearest neighbor search (NNS) problem. The most popular application of NNS is recommender systems. For example,

YouTube recommender systems first utilize NNS to identify a set of candidate videos for a specific user and then use a separate, heavy deep neural network to select top recommendations [47]. Similarly, NNS is often used to efficiently identify a set of candidates [8] for CTR (Click-Through-Rate) prediction models, such as Meta DLRM [140], Google DCN [192], and Alibaba BST [33]. Moreover, similarity search is used for conventional search engines such as Microsoft Bing [2], and can also be used for multimedia search services where a user provides image or audio input to find similar ones.

The broad applicability of similarity search has motivated many researchers in academia and industry to explore various similarity search algorithms. For example, Meta maintains a similarity search library named Faiss [103] and Google recently released a similarity search library named ScaNN [77]. Microsoft [38], Yahoo Japan [7] also have their own similar search libraries.

Similarity Metrics. Various similarity functions can be used to define the similarity between vectors. Among them, the most common similarity metrics are inner product similarity and L2 distance similarity shown in the equation below.

$$s_{ip}(q,x) = \sum_{i=0}^{D-1} q[i]x[i] = \|q\|\|x\| \cos\theta \text{ (Inner Product)}$$
$$s_{L2}(q,x) = -\sum_{i=0}^{D-1} (q[i]-x[i])^2 \text{ (L2 Distance)}$$

Inner product is one of the most popular similarity metrics. Here, a similarity $s(q,x)$ between two D-dimensional vectors q and x is defined as the dot product of those two vectors. This metric is used for computing the similarity between embeddings, and the NNS using the inner product is called Maximum Inner Product Search (MIPS). L2 distance between vector q and x is computed as the sum of the squared value of each dimension's difference. Geometrically, this is the squared value of the length of the line connecting two points defined by each vector. Since L2 distance is technically a dissimilarity metric, the negative of squared L2 distance is utilized as a similarity metric, as shown in the above equation. A common use case of L2 distance search is image similarity search.

Approximate Similarity Search. The most naïve way to perform a similarity search is to compute the similarity between the query vector and all database vectors, then sort the similarities to obtain k vectors with the highest similarity. Assuming N database vectors having D dimensions, this requires ND multiply-and-add operations and $2ND$ bytes memory accesses, assuming 16-bit datatype for each vector element. The cost of computation and memory accesses becomes prohibitive when N is large (e.g., a billion). To avoid this problem, many approximate similarity search algorithms (ANNS) have been proposed [58, 61, 77, 93, 96, 103, 129]. Such algorithms can significantly reduce the amount of computation as well as the memory accesses at the expense of a slight reduction in search accuracy.

There exist various ANNS algorithms such as hash-based ones [52, 170], graph-based ones [87, 129, 211], and compression-based ones [77, 100, 103]. Among these solutions, compression-based ones are the most popular choice for billion-scale search scenarios. Graph-based ones and hash-based ones are very effective for million-scale searches (e.g., finding similar movies), but they are not well-suited for billion-scale searches where their memory requirement (i.e., whole dataset as well as their large index structures) often exceeds the main memory size. However, the main problem with those solutions is that they still require all data as well as additional index structures to be resident in memory. The dataset itself is 256GB, when N is a billion and D is 128. This only fits in a relatively large single-node machine. Moreover, many ANNS algorithms also require additional $O(N)$ memory capacity for index structure [130].

To avoid such huge capacity requirements, compression-based ANNS algorithms have been proposed. Such algorithms encode the original data in a compressed form and utilize the compressed form to compute the similarity between the query and the database vectors. Since these algorithms do not require the original, uncompressed vectors to be kept in memory, they require much less memory capacity and are better suited for large-scale datasets. Compression-based schemes have drawn a lot of attention from both academia and industry for their capabilities to handle billion-scale searches effectively.

Some recent work [35, 137, 204, 207] explores ways to optimize and refine compression-based similarity search and major industry players like Google and Meta have released their compression-based similarity search libraries as open-source software [77, 103]. Considering the ever-increasing data size, we expect the compression-based schemes to continue to be the most dominant scheme in the foreseeable future.

2.2.3.1 Product Quantization (PQ)

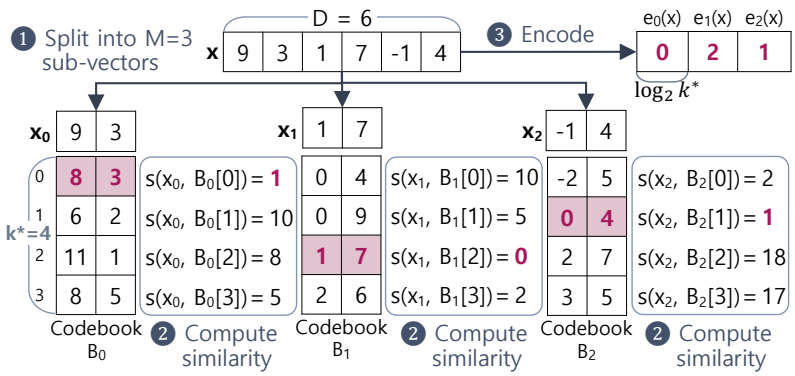


Figure 2.3: Illustration of Product Quantization.

Encoding. A product quantization scheme encodes each database vector into a compressed form as shown in Figure 2.3. For this purpose, it first divides a vector into multiple sub-vectors. For example, a D-dimensional vector x is represented as a concatenation of M sub-vectors $x_0, x_1, x_2, ...x_{M-1}$ where each x_i is a D/M-dimensional vector. In the figure, ❶ we assume $D=6$ and $M=3$, so each sub-vector is a 2-dimensional vector. Then, the product quantization scheme encodes each sub-vector into an identifier and represents the vector as a concatenation of identifiers. To obtain a corresponding identifier for each sub-vector, the product quantization scheme uses a set of vectors named codebook. A separate learning process obtains codebooks $B_0, ..., B_{M-1}$, one for each D/M dimension, and each codebook consists of k^* codeword vectors, each having D/M dimension. The figure shows $M=3$ codebooks of $k^*=4$ codeword vectors each. There exist various algorithms to obtain codebooks [67, 77, 103, 112],

and the quality of the codebook affects final search accuracy. Using a codebook B_i, a sub-vector x_i is encoded into an identifier. ❷ Specifically, product quantization scheme first computes the similarity between a sub-vector x_i and each codeword vector in the codebook B_i (i.e., $B_i[0], ...B_i[k^* - 1]$). Eventually, the codeword with the highest similarity is selected.

Assuming $B_i[j]$ is selected, x_i is now represented as $e_i(x) = j$. ❸ Repeating this process for each sub-vector x_i, the vector x is now represented as a concatenation of $x' = e_0(x), ..., e_{M-1}(x)$ where $0 \leq e_i(x) < k^*$. Assuming that the original datatype is 2 bytes (float16) per vector element, the original vector requires $2D$ bytes storage. With this encoding, each $e_i(x)$ is represented with $\log_2 k^*$ bits, and the total number of bytes for the encoded vector x' is $M \log_2 k^*/8$ bytes. In the figure, original vector x requires 12 bytes storage, but with encoding, it only requires less than 1 byte (e.g., 6/8 bytes). While the choice of smaller M and k^* significantly improves the compression rate, too much compression can negatively affect search accuracy.

Approximate Similarity Computation. To compute the approximate similarity between each encoded vector x' and a given query q, the PQ scheme utilizes the encoded identifier e_i to obtain the corresponding codeword from codebook B_i. Then, the decoded vector is represented as a concatenation of $B_0[e_0(x)], ...B_{M-1}[e_{M-1}(x)]$ where $B_i[e_i(x)]$ is a D/M-dimensional vector. The similarity between this D-dimensional decoded vector and the query vector ($q_0, ...q_{M-1}$ where q_i is a D/M-dimensional sub-vector) is computed as shown below. This process is repeated for all encoded vectors, and the top-k most similar vectors can be obtained.

$$s_{ip}(q, x') = \sum_{i=0}^{M-1} q_i B_i[e_i(x)] \text{ (Inner Product)}$$
$$s_{L2}(q, x') = -\sum_{i=0}^{M-1} \|q_i - B_i[e_i(x)]\|^2 \text{ (L2 Distance)}$$

Efficient Similarity Computation with Memoization. In the above equations, an inner product similarity computation requires D multiplications and $D-1$ additions. Similarly, a L2 distance similarity computation requires D subtractions, D multiplications, and $D-1$ additions. These are repeated for

all N database vectors. Carefully inspecting the above equations, we can see that it is more efficient to memoize $\{q_i B_i[0],\ \ldots\ , q_i B_i[k^* - 1]\}$ (inner product) or $\{-\|q_i - B_i[0]\|^2,\ \ldots\ , -\|q_i - B_i[k^* - 1]\|^2\}$ (L2 distance), and reuse them across different database vectors. Assuming those k^* values are memoized in a lookup table L_i (for all $0 \leq i < M$), the above similarity computation equation changes to the following for both the inner product similarity and the L2 distance similarity.

$$s(q, x') = \sum_{i=0}^{M-1} L_i[e_i(x)]$$

With this memoization, computing the similarity between q and x' only requires M lookup table references and $M - 1$ additions. To construct the lookup table for inner product similarity, k^*D multiplications and $k^*(D - 1)$ additions are necessary. Similarly, the L2 distance similarity computation requires k^*D subtractions, k^*D multiplications, and $k^*(D - 1)$ additions. In both cases, the required capacity is $2Dk^*$ bytes since there exist M lookup tables and each lookup table L_i has k^* entries where each entry stores D/M-dimensional sub-vector requiring $2D/M$ bytes storage. Note that the size of the lookup table or the amount of computation for construction of the lookup table is independent of N. With a large number of database vectors, the lookup table construction overhead can easily be amortized.

2.2.3.2 Two-level Product Quantization ANNS

The base product quantization scheme presented in the previous section i) reduces the memory capacity requirement and ii) reduces the amount of computation via memoization. Still, it does not reduce the number of database vectors that the similarity computation needs to be performed. As a result, the total search time is still proportional to N. To address this limitation, many popular similarity search implementations adopt a two-level product quantization scheme (also called inverted-index-based product quantization) that substantially reduces the number of database vectors for which the distance computation needs to be performed.

Encoding. In this scheme, the database vectors are first grouped into multiple clusters using a clustering algorithm. Technically, any clustering algorithm can be utilized, but *kmeans* is the most popular choice. After clustering, a representative point from each cluster is decided. A popular way to obtain this representative point is simply using the centroid of all vectors within a cluster. For cluster j, we call the centroid vectors as $c^{(j)} \in R^D$. Then, for each database vector x in cluster j, the residual $r(x) = x - c^{(j)}$ is computed. Finally, $r(x)$ is encoded using the product quantization scheme explained in Chapter 2.2.3.1 to obtain $r(x)'$ which is a concatenation of $e_0(r(x)), e_1(r(x)), ..., e_{M-1}(r(x))$. Then, these encoded vectors belonging to this specific cluster are stored together, along with the cluster centroid vector $c^{(j)}$.

$$
\begin{aligned}
s_{ip}(q, x') &= s_{ip}(q, c^{(j)} + r(x)) \\
&= q \cdot c^{(j)} + \textstyle\sum_{i=0}^{M-1} q_i B_i[e_i(r(x))] \\
&= q \cdot c^{(j)} + \textstyle\sum_{i=0}^{M-1} L_i[e_i(r(x))] \\
s_{L2}(q, x') &= s_{L2}(q, c^{(j)} + r(x)) = s_{L2}(q - c^{(j)}, r(x)) \\
&= -\textstyle\sum_{i=0}^{M-1} \|(q_i - c_i^{(j)}) - B_i[e_i(r(x))]\|^2 \\
&= \textstyle\sum_{i=0}^{M-1} L_i[e_i(r(x))]
\end{aligned}
$$

Similarity Computation. Such a cluster-wise encoding scheme changes the way to compute the similarity, as shown in the equations above. L_i for inner product similarity stores $\{q_i B_i[0], ..., q_i B_i[k^*-1]\}$ and L_i for L2 distance similarity stores $\{-\|(q_i - c_i^{(j)}) - B_i[0]\|^2, ..., -\|(q_i - c_i^{(j)}) - B_i[k^*-1]\|^2\}$, where $c^{(j)}$ is one of C . For the inner product similarity, the contents of the lookup table are invariant to the chosen clusters and the term $q \cdot c^{(j)}$ needs to be added at the end. On the other hand, for the L2 distance similarity, the contents of the lookup table are variant to the chosen clusters.

Search Process Step 1 - Cluster Filtering. Figure 2.4 illustrates three steps of two-level product quantization and Table 2.1 shows the notations used for two-level product quantization throughout the thesis. Given a query vector q, the first step to finding similar vectors is cluster selection. During

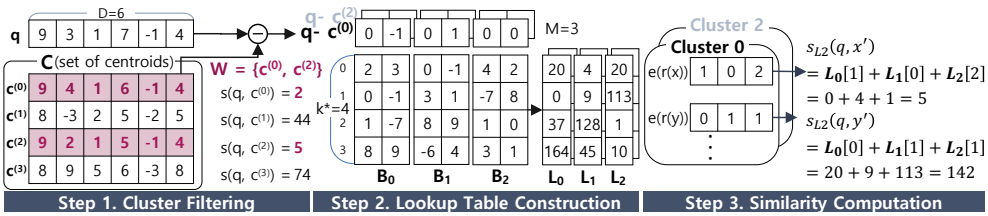


Figure 2.4: Illustration of Two-level Product Quantization ANNS (L2 distance).

Table 2.1: Notation for Two-level Product Quantization ANNS.

Notation	Description
N	# of database vectors
D	Dimension of database vectors
M	# of sub-vectors
$x[i]$	ith element of vector x
$x[i]$	ith sub-vector of vector x
$B_{0...M-1}$	Codebook for each sub-vector
k^*	# of codewords
$e_i(x)$	Encoded identifier of vector x_i
C	Set of centroids
$c^{(i)}$	Centroid vector of cluster i
W	Set of selected centroids
$c^{(s)}$	Selected centroid in W
$L_{0...M-1}$	Lookup table for each sub-vector

this step, it computes the similarity between vector q and all centroid vectors (i.e., $c^{(0)}, c^{(1)}, ..., c^{(|C|-1)}$), and then find the W most similar centroids as shown in the below equation. The set of selected centroids is denoted as W. In the figure, the most similar (in terms of L2 distance) centroids are $c^{(0)}$ and $c^{(2)}$ and thus $W = \{c^{(0)}, c^{(2)}\}$. Essentially, this step excludes vectors belonging to the cluster with centroids that are not similar to the query and hence effectively reduces the total number of candidates.

Search Process Step 2 - Lookup Table Construction. Once the nearby centroids are identified, the algorithm constructs the lookup table. This process

slightly differs for the two metrics. First, for the inner product similarity, the contents of the lookup table L_i (see s_{ip} in the above equation) are independent of the selected centroid $c^{(s)}$, where $c^{(s)}$ is one of the selected clusters in W, and thus it is sufficient to construct the lookup table once and reuse it for all clusters. On the other hand, the contents of the lookup table (see s_{L2} in the above equation) are dependent on the selected centroid $c^{(s)}$ and thus, the table needs to be constructed for each cluster. The figure assumes $D=6$, $M=3$ and L2 distance case and illustrates creating a lookup table for cluster 0.

Search Process Step 3 - Similarity Computation. Once the lookup table is ready, the similarity computation between the query vector q and database vectors in the selected clusters is performed. With the lookup table, each similarity computation is simply M additions (inner product) or M–1 additions (L2 distance). The figure shows the process of computing the similarity of vector x. Assuming encoded vector $e(r(x))$ is $(1, 0, 2)$, it computes similarity by summing up $L_0[e_0(r(x))] + L_1[e_1(r(x))] + L_2[e_2(r(x))]$ which is 5. This process is repeated for all vectors in the selected clusters, and the top-k most similar vectors are selected and then returned. Overall, this PQ-based ANNS can i) substantially reduce the number of database vectors inspected by cluster-level filtering and ii) can efficiently compute the approximate similarity between the database vector and the query with memoization.

2.3 Bottleneck Analysis for Embedding Operations

Table 2.2: Summary of Embedding Operations.

Applications	Embedding Operation	Challenge
Recommender System	Embedding Reduction	High Memory Bandwidth Requirement (Challenge 1)
Transformer-based Model (NLP, CV)	Self-Attention	High Computation Requirement (Challenge 2)
Recommender System Search Engine	Similarity Search	HW Inefficiency (Challenge 3)

Table 2.2 shows the summary of emerging AI applications and their embedding operations. Each embedding operation faces different challenges as discussed in Chapter 1, showing different performance bottlenecks.

Embedding reduction operation, which is primarily used in recommender systems, is known to be memory-bound with a relatively small amount of arithmetic computation (i.e., having a low arithmetic intensity [79, 110]) where the performance is determined by system memory bandwidth. This operation performs sparse indexing, which generates a large number of irregular memory accesses. Since off-chip memory accesses are more expensive than on-chip computation in terms of latency and energy consumption [34, 85, 191], it is critical to reduce this high cost of memory accesses for efficient embedding reduction.

Self-attention operation is one of the most widely used operations in neural network models for NLP, computer vision and etc. In contrast to its strong ability, it is a costly operation that requires an amount of computation that quadratically increases with the number of entities involved in this operation. Its high computational cost becomes a limiting factor for deployment. For example, many existing NLP models such as Google BERT limit the self-attention to be applied for up to 512 tokens (e.g., words) to avoid excessive performance and energy overhead. When the input text has more than 512 tokens, the input text needs to be divided into multiple segments (each with up to 512 tokens), and self-attention is separately applied for each segment. Unfortunately, such a scheme makes NLP models unable to capture the relation between two tokens that do not belong to the same segment.

Compression-based similarity search operation is also a well-known operation primarily used for recommender systems, search engines and etc. This operation on conventional hardware (CPU or GPU) is suboptimal due to its specific computation pattern which heavily utilizes memoization. For CPUs, the sources of inefficiency are twofold: (1) application's lack of control over the on-chip memory usage makes it difficult to keep the memoization results and frequently reused data on-chip, incurring further main memory accesses, and (2) CPU's ability to support dynamic control flow or dynamic extraction of

```
1  // for i = 0 to query[qid].size()-1;
2  REDUCTION:
3     // %rcx: feature index i; initialized with 0
4     // %r9: query[qid].size()
5  FEAT_LOOP:
6     // for d = 0 to D-1:
7     //    res[d] += embedding_vec[d];
8                ...
9     88.64%   vmovups (%rsi), %ymm2
10     6.96%   vadddps (%rax), %ymm2, %ymm0
11    1.68%    vmovups %ymm0, (%rax)
12               ... // loop unrolling
13 FEAT_REPEAT:
14               inc %rcx
15               cmp %rcx, %r9
16               jne FEAT_LOOP
```

Figure 2.5: Instruction-level Runtime Breakdown for Embedding Reduction Using Amazon-Books Dataset[1].

instruction level parallelism incurs extra energy overhead on ANNS scheme, which has relatively static control flow and easy-to-extract parallelism. On the other hand, on GPUs, (1) shared memory usage from the memoization table limits the GPU parallelism (i.e., the number of thread blocks scheduled for each streaming multiprocessor (SM)) and incurs resource underutilization, and (2) the low arithmetic intensity computation pattern results in gross underutilization of GPU computation capability.

2.3.1 Embedding Reduction

Embedding reduction has a small number of arithmetic computations compared to the number of memory reads it generates. In a CPU, which is a common deployment platform to accommodate a large embedding table [79, 88, 110], SIMD optimization (e.g., Intel AVX) is usually applied. Figure 2.5 shows the instruction-level profiling results of the embedding reduction operation on the CPU using `perf annotate`. The profiling results demonstrate that embedding reduction operation is an extremely memory bandwidth-bound operation as most of the runtime is spent on the SIMD load instruction (`vmovups`). In

[1]See Chapter 3.7.1 for details on datasets.

this code, the SIMD add instruction is very efficient with wide vector processing, but our internal profiling result shows that memory accesses to the embedding table yield a high cache miss rate (e.g., 52.8%) in L3 cache despite the existence of temporal locality to a certain degree. This inefficiency is due to the following reasons. First, embedding reduction accesses embedding table with sparse index; thus, cache miss rate increases. Second, the embedding tables are often much larger than L3 cache sizes. For example, the embedding table for one million 256B embeddings takes 256MB, which is an order of magnitude larger than a typical L3 cache size. There are cases where multiple embedding tables are accessed at the same time by different threads [110], thus further increasing the memory pressure. Third, there are other data structures that contend for the cache space (e.g., `query`, `res`). Finally, between batches of embedding reduction, other layers of the DNN model may be executed to evict most of the embedding vectors from the cache hierarchy.

In fact, the embedding reduction operation is the primary performance bottleneck in many NN models. One example is Meta's DLRM [139], in which their profiling results [80, 110] show that this operation accounts for 50% to 75% of the total runtime in their models (i.e., RMC1, RMC2). Furthermore, with a scaling of the dataset and the adoption of specialized DNN accelerators to shrink the portion of the compute-intensive layers, the bottleneck is likely to be more critical in the future.

2.3.2 Self-Attention

As described in Chapter 2.2.2, the self-attention mechanism consists of three steps, ❶ similarity computation, ❷ softmax normalization, ❸ weighted sum. The first similarity computation is matrix multiplication, which requires n^2d multiply-and-accumulate (MAC) operations (since it multiplies $n \times d$ matrix with $d \times n$ matrix). The second softmax normalization operation requires n^2 exponent operations, and the final weighted sum is also a matrix multiplication that requires n^2d MAC operations ($n \times n$ matrix is multiplied with $n \times d$).

Fig. 2.6 shows the portion of the runtime spent on self-attention in popu-

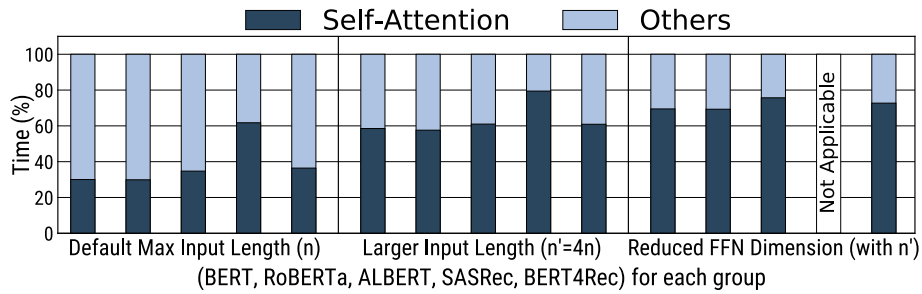


Figure 2.6: Breakdown of the Runtime Spent for Self-attention Mechanism.

lar NN models. We run SQuADv1.1 dataset [159] for NLP models (BERT, RoBERTa, ALBERT) and MovieLens-1M [86] for recommendation models (SASRec, BERT4Rec) on NVIDIA V100 GPU [142]. The details of each workload are available in Chapter 4.5.1. The left side of the figure shows that self-attention accounts for a significant portion (about 38%) of the runtime across many existing self-attention-oriented NN models. Furthermore, the figure also shows that increasing n further than the published model parameter, say, by 4×, makes the self-attention account for an even larger portion (about 64%) of the model runtime. Finally, note that several recent research works on NLP models suggest that the portion of self-attention is going to increase even further. For example, a recent research [198] demonstrates that extraneous dimensions in the feedforward layers are unnecessary and removing them hardly affects the model accuracy while significantly reducing the runtime of the feedforward layers in Transformer-style models. The right side of the figure shows that the runtime portion of the self-attention on these models reaches about 73% when the feedforward layer dimension is reduced by 4× [198]. In addition, several recent proposals investigate the idea of replacing the feedforward layer in Transformer-style models with the self-attention [117, 176] for better model accuracy. Such trends will make the self-attention take an even larger portion of the total model runtime in the future.

2.3.3 Product Quantization (PQ) Similarity Search

Recommender system is one of the representative use cases that utilize similarity search. To deal with the constantly growing amount of information, modern personalized recommender systems are generally composed of two stages: retrieval, also referred to as candidate generation, and ranking. In the retrieval stage, thousands of potential candidates are retrieved from a vast dataset in a fast and computationally efficient manner. In the ranking stage, retrieved candidates are ordered using separate neural network models or complex algorithms. Similarity search is employed during the retrieval stage to identify potential candidates from large datasets.

Also, search engine [2, 197], online advertising and e-commerce (e.g., Facebook Marketplace, Instacart, Walmart, Taobao) applications also consist of retrieval and ranking stages. Likewise, they utilize similarity search for the retrieval stage. The Google Play application's recommender system has on the order of 10 ms for the end-to-end query serving time for both retrieval and ranking stages [39]. Also, the retrieval stage for Taobao Product Search takes approximately 10 ms [65, 120]. Due to such tight constraints for the end-to-end query serving time, similarity search accounts for a considerable portion of the overall query serving time.

An algorithm best suits the specific hardware if i) the hardware fully utilizes the available compute resources while executing the algorithm, ii) the hardware can maximize data reuse and iii) the hardware fully utilizes the available memory bandwidth for data that needs to be loaded from memory. Unfortunately, we find that this is not the case when we run PQ-based ANNS on conventional hardwares.

GPU Implementation Analysis. We profile Meta Faiss [4, 103], one of the most popular implementations of the PQ-based ANNS, for GPU on NVIDIA V100 GPU. Overall, two kernels account for most (98%) of the query runtime. The first kernel simply performs the approximate similarity computation using memoization. Profiling the behavior of this kernel with Nvidia Nsight [9] tool

and Nvidia Visual Profiler [10] reveals that this kernel fails to effectively utilize the available GPU memory bandwidth as well as its floating-point units. The kernel requires a relatively large amount of shared memory per block (32KB) to store the lookup table, and this requirement limits the number of thread blocks scheduled on SM to three since each SM has 96KB shared memory. The low number of resident thread blocks per SM limits the GPU's ability to hide the memory latency via parallelism, which eventually leads to a reduction in throughput. The second kernel selects top-1000 vectors having the largest similarity out of all vectors whose similarities are computed in the previous kernels. Despite many optimizations in Faiss [103], this kernel has limited parallelism (i.e., small grid size), which prevents it from fully utilizing GPU resources. Moreover, this operation only utilizes about 4% of the total FMA units since this kernel is mostly about selecting top-scoring vectors without performing much computation.

CPU Implementation Analysis. We analyze the performance characteristics of Google ScaNN [77] and Meta Faiss [103] on Intel Skylake-X 8-core CPU. For both cases, the system spends most of its time on a loop fetching encoded vectors from the main memory and utilizes those data to read the lookup tables and performs sum reduction of the data read from the lookup tables. Although bottlenecks vary across configurations, we find two major sources of performance degradation. First, the system is often bounded by the memory bandwidth. Specifically, since an encoded vector is only utilized once per query with no reuse, it does not benefit much from the CPU cache hierarchy and consumes a large amount of system bandwidth. In certain configurations where the memory bandwidth is not a bottleneck, the primary source of performance degradation is its inability to deal with sub-byte data types effectively. Specifically, when $k^* = 16$, a single vector element is encoded as a 4-bit integer. Since the CPU does not have support for the 4-bit data type, it continuously utilizes shift instructions (e.g., `VPSRLW`) to process 4-bit data. Such use of excess instructions ends up degrading the processor's effective throughput.

Chapter 3

Efficient Embedding Reduction on Commodity Hardware via Sub-Query Memoization

3.1 Overview

MERCI is a novel memoization framework for an efficient embedding reduction on the commodity hardware. Based on the observation that there often exists a correlation structure among features in a real-world dataset, we present a new optimization opportunity in embedding reduction via memoization technique. To maximize the benefit of memoization, we propose fine-grained (sub-query) memoization to perform partial reduction when processing a query. For this purpose, we introduce Correlation-Aware Variable-Sized Clustering to identify clusters of frequently co-appearing features of variable length with high coverage and small table size, as well as a feature remapping scheme to quickly locate a partially reduced embedding with a small number of instructions.

3.2 Opportunities for Sub-query Memoization

Memoization [133] is a classic technique that stores the result of computation for an input (query) and reuses it when the same query arrives again. Memoization leverages space-time trade-offs and is the most effective when a small subset of inputs is likely to occur repeatedly. Memoization can be an effective solution to reduce memory accesses in embedding reduction by replacing N embedding table lookups with a single memoization table lookup, where N is the number of the embeddings that the given query should aggregate. However,

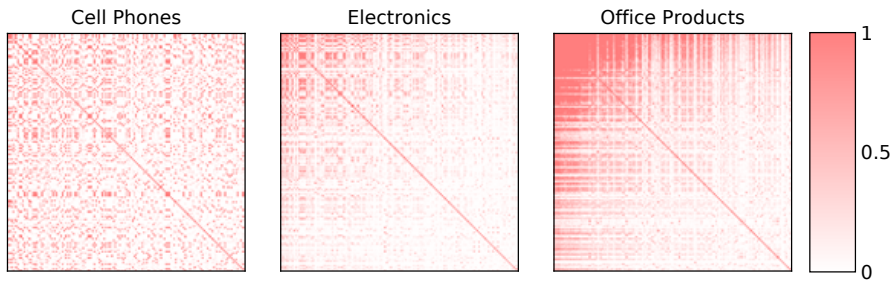


Figure 3.1: Correlation Heat Map for Product Pairs of Top 150 Items in Amazon Review Dataset.

this coarse-grained (i.e., query granularity) memoization has limited coverage as memoization can be applied only when the *exact* same query arrives again.

Instead, we identify new opportunities for *fine-grained* (i.e., sub-query) memoization to enable partial reduction by exploiting the correlation structure that exists in many real-world categorical features. This partial reduction is possible as a reduction operator (e.g., sum) is commutative and associative by definition. For example, if a query contains a user's recently browsed items, there is a high probability that if *notebook* appears in the query, then *pen* would also appear together. Similarly, (*notebook*, *pencil*) and (*pencil*, *eraser*) are co-appearing pairs that are frequently browsed. In contrast, (*pants*, *pencil*) and (*shampoo*, *apple*) are pairs that are much less likely to appear together. We can exploit such patterns and memoize the partial reduction for the frequently co-appearing features to replace two or more memory accesses with single memory access.

Figure 3.1 illustrates such opportunities indeed exist in a real-world dataset. This Amazon Review dataset contains lists of items that are bought/viewed together, which are commonly utilized as a categorical input of the recommender system. The heat map depicts the pair-wise correlation of the Top 150 frequently appearing items in this dataset. Thus, both X and Y-axis enumerate the 150 items, and a dot in the figure quantifies the correlation of a particular item pair in the range of zero (low, white) to one (high, red). This heat map shows pairs of items jointly appearing frequently in the lists (queries), which

can be excellent targets for memoization. Although the figure only shows the pair-wise co-appearance, there often exists a cluster of co-appearing features (items) that have a high chance of appearing together. Thus, we propose to exploit this correlation structure to perform memoization at a *sub-query* granularity (as few as two embeddings) to provide much greater coverage than the coarse-grained memoization scheme at a query granularity.

3.3 MERCI Overview

Although conceptually simple, building a performant memoization system for embedding reduction is a challenging task. If done naïvely using a dense array, the cost of memoization can nullify its benefit. For example, storing partial sums of all possible combinations would not be possible due to memory constraints; for N embeddings, the required memory space is $2^N \times$ {Embedding Vector Size} where N often exceeds a million in many popular NN models. Utilizing a sparse data structure (e.g., a sparse hash table) for maintaining the memoized values and only storing partial sums of frequently co-appearing features can solve this problem, but triggers considerable additional memory access to retrieve a memoized value.

MERCI proposes a way to get the best of both approaches. It moves through two phases that we call offline clustering and online query processing. Figure 3.2 shows the overall overview of MERCI.

Offline clustering. The offline clustering phase consists of two steps. ❶ MERCI first partitions N features into a set of coarse-grained, fixed-length partitions called **super-partitions** by utilizing an existing hypergraph partitioning algorithm on the training dataset (Step 1). Each super-partition contains features that are likely to appear together based on the history of queries. Then, ❷ MERCI applies Correlation-Aware Variable-Sized Clustering to each super-partition, dividing features in a super-partition into fine-grained, variable-length clusters (Step 2). Finally, ❸ MERCI creates a memoization table that holds all possible partial sum combinations for each cluster.

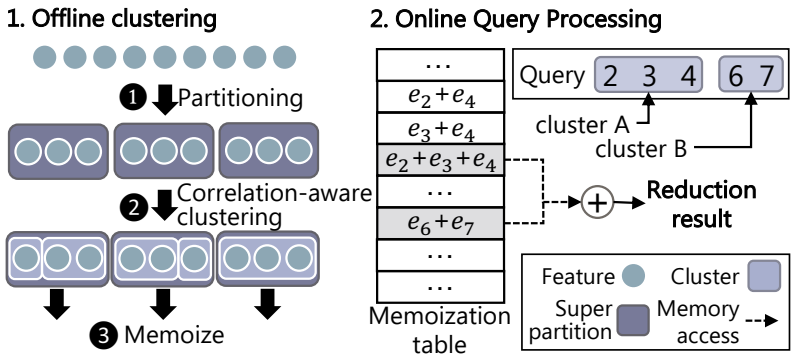


Figure 3.2: Overview of MERCI.

Chapter 3.4 describes the details of these steps.

Online query processing. MERCI utilizes the memoization table created from the previous phase to serve incoming queries. Once a query arrives—for example, one requiring access to the feature set of $\{2, 3, 4, 6, 7\}$ as in Figure 3.2—MERCI first identifies which features belong to the same cluster. In this example, features $\{2, 3, 4\}$ belong to cluster A and $\{6, 7\}$ to cluster B. Hence, two memoized partial sums (embeddings) are retrieved (i.e., $e_2 + e_3 + e_4$ and $e_6 + e_7$) and summed up to generate the final output. While the baseline scheme—performing embedding reduction without memoization—would have to load five embeddings (i.e., e_2, e_3, e_4, e_6, e_7), MERCI only loads two. Because the embedding reduction is often memory bandwidth-bound, such reduction in memory accesses can lead to performance improvement. Chapter 3.5 describes how the online query processing phase utilizes clusters to optimize the memoization table and accelerate the embedding reduction at runtime.

3.4 MERCI Offline Clustering

3.4.1 Step 1: Hypergraph Partitioning

The first step of the offline clustering phase is coarse-grained, fixed-length partitioning of all N features utilizing a hypergraph partitioning algorithm. Hypergraph partitioning is a popular algorithm that aims to generate a specified number of *equal-sized* partitions while minimizing the number of accessed

partitions for a given set of queries; that is, features in the same partition have a high chance of occurring together. We first partition all N features into equal-sized *super-partitions* of size S with an existing hypergraph partitioning algorithm implementation called PaToH [30]. As N (number of features) can often exceed millions in today's NNs, the hypergraph partitioning algorithm first reduces this large problem size to S, and the proposed fine-grained clustering algorithm called Correlation-Aware Variable-Sized Clustering (Chapter 3.4.2) is performed on each super-partition to manage the complexity. While PaToH works well for our purpose, there exists a variety of different hypergraph partitioning algorithms [30, 56, 106, 109, 165] with different time-quality trade-offs [165], and there is no fundamental limitation in using a different algorithm.

One question that might arise is whether we can use a hypergraph partitioning algorithm to create fine-grained clusters, instead of the proposed two-step algorithm, as it also groups frequently co-appearing features. It may be possible but suboptimal for the following reasons. First, it only generates equal-sized partitions, which either cannot support a set of co-appearing features larger than the (fixed) partition size or waste memory space for a smaller set. The size of a group of the co-appearing features would vary. Our algorithm can reflect diversity by generating clusters of sizes one to twenty or more. Second, the hypergraph partitioning algorithm does not give a fine-grained knob for memory constraints. Once the cluster size is set to S (embeddings), the space overhead is fixed to $N/S \times (2^S - 1) \times \{\text{Embedding Vector Size}\}$. On the other hand, our algorithm can set memory limits in the granularity of less than 1% of the original embedding table size. Thus, the hypergraph partitioning algorithm can reduce the search space while increasing the locality, but in itself, it is not sufficient for our purpose.

3.4.2 Step 2: Correlation-Aware Variable-Sized Clustering

Once the hypergraph partitioning algorithm divides all N features into super-partitions containing S features each, the next step is to apply Correlation-

Aware Variable-Sized Clustering to each super-partition individually. It is a clustering algorithm that aims to further divide the S features into fine-grained variable-sized *clusters* such that the resulting memoization table yields the maximum benefit for a given memory constraint. All possible combinations of the partial sum within a cluster are then stored for memoization.

Sketch of the Algorithm. The Correlation-Aware Variable-Sized Clustering algorithm is applied to all N/S super-partitions independently. Let c_i be a set of feature IDs that belongs to cluster i. Initially, we let each feature form a distinct cluster by itself (i.e., S clusters of size one). In this state, there is nothing to memoize since no cluster has more than one feature. The algorithm then selects and merges two clusters; to select these clusters, it considers both the *benefit* (i.e., decrease in the expected number of memory accesses if merged) and the *cost* (i.e., increase in memory usage if merged). This cost and benefit are estimated using queries in the training set, as explained below. Once two clusters are merged, the algorithm repeats the next iteration until it hits the memory size constraint.

Estimating the Cost. Since all possible combinations of features in a cluster are memoized, a cluster with a features occupies $(2^a - 1) \times \{\text{Embedding Vector Size}\}$ of memory space. $2^a - 1$ is the cardinality of power set (excluding empty set) for a set with a elements. With this in mind, we can compute the memory cost of merging cluster A having a features and cluster B with b features. The newly formed cluster would require $2^{a+b} - 1$ partial sums to be stored; since the original memory usage was $2^a - 1 + 2^b - 1$, the increase in memory usage (i.e., the cost of merge) is $(2^{a+b} - 2^a - 2^b + 1) \times \{\text{Embedding Vector Size}\}$.

Estimating the Benefit. Merging two clusters improve the chance of features in a query being in the same cluster (and hence memoized). Therefore, the number of memory accesses for processing a query would likely decrease. The amount of decrease is considered as the benefit of merging two clusters. The queries in the training set are analyzed to estimate the benefit. To explain, we define Q as the set of queries being analyzed, and I is an inverted index data structure [48] for each cluster.

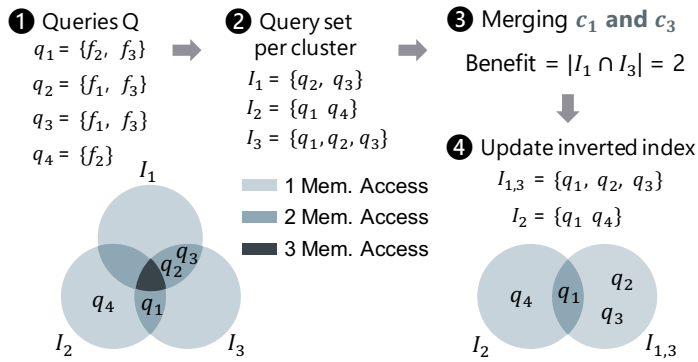


Figure 3.3: Illustration of Computing Benefits for a Particular Merge Decision.

$$Q = \{q_i \mid i = 0, 1, ..., q\},\ I_i = \{q_j \mid \exists f_k \in c_i,\ s.t.\ f_k \in q_j\} \tag{3.1}$$

Following the numbering in Figure 3.3, ❶ $Q = \{q_1,\ q_2,\ q_3,\ q_4\}$ is a set of queries used for training, and each query is identified as a set of feature IDs it accesses (e.g., f_1, f_2, ...), where each feature corresponds to a specific embedding vector. ❷ Using this information, an inverted index I_i is built and maintained for each cluster i. I_i contains IDs of queries that contain at least one feature belonging to cluster i. For instance, if f_1 appears in q_2 and q_3, I_1 becomes $\{q_2,\ q_3\}$ as in Figure 3.3. ❸ Then, the benefit of merging cluster c_1 and c_3 is computed as the cardinality of $I_1 \cap I_3$, which is equivalent to the number of queries that contain at least one feature from each of c_1 and c_3. That is, if c_1 and c_3 are merged, $c_1 \bigcup c_3$ would be memoized, and thus queries q_2 and q_3 would require one memory access instead of two saving one memory access for embedding reduction. Note that, our scheme originally chooses the pair of clusters considering both *benefit* and *cost*; however, in this example, cost is the same for all pairs of clusters ($a = 1, b = 1$) and thus we only consider the benefit.

$$benefit(c_i, c_j) = |\{\ q_k \mid q_k \in I_i \cap I_j\ \}| \tag{3.2}$$

❹ After the algorithm merges c_i and c_j, it updates the state of the clusters and the inverted index as shown below. For the next iteration, this merged cluster is treated as a single one cluster ($c_{i,j}$) for benefit-cost analysis. In

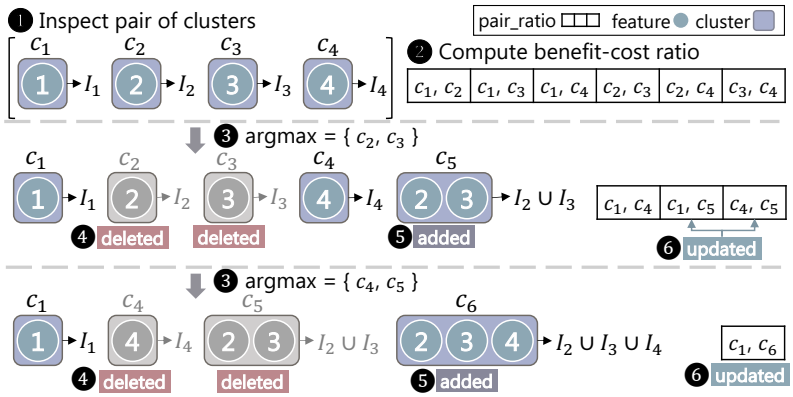


Figure 3.4: Illustration of the Correlation-Aware Variable-Sized Clustering.

consequence, various sizes of clusters can be generated by the granularity of one feature.

$$c_{i,j} = \{\ f_k \mid f_k \in c_i \cup c_j\ \},\ I_{i,j} = \{\ q_k \mid q_k \in I_i \cup I_j\ \} \tag{3.3}$$

By computing *benefit* and *cost*, we evaluate **benefit-cost ratio** (i.e., $\frac{benefit}{cost}$) of all possible pairs of clusters. With the benefit-cost ratio, we can quantify the effectiveness of every possible merge of two clusters. If the pair has a high benefit-cost ratio, merging those clusters increases the benefit of memoization with relatively low additional memory usage. To summarize, a unique feature of our clustering scheme is that it selects clusters to merge by considering the benefits of forming a larger cluster (decrease in memory accesses) and its cost (increase in memory usage).

3.4.3 Algorithm Details

Figure 3.4 illustrates an example walk-through of our Correlation-Aware Variable-Sized Clustering algorithm with a running example. First, ❶ it inspects every pair of the clusters, and ❷ records the benefit-cost ratio of the merged cluster (i.e., `pair_ratio`). Then, ❸ it selects the cluster pair with the highest benefit-cost ratio (i.e., clusters `2` and `3`) and merges them into a single cluster. In doing so, ❹ it deletes the two original clusters (i.e., c_2 and c_3) and ❺ adds the new aggregated cluster c_5 ($= c_{2,3}$). Finally, ❻ the cluster pairs and their

ratios are updated along with the inverted index of the merged cluster. Note that merging two clusters implies that queries with a feature in either of the two clusters now access the merged cluster c_5. Thus, we set an inverted index of the merged cluster as a union of two existing inverted indexes (i.e., $I_5 = I_2 \cup I_3$). The process of ❸ - ❻ is repeated with newly updated `pair_ratio`, treating c_5 as a single cluster like others. In Figure 3.4, cluster `4` and `5` are selected to be merged to become c_6. Although not shown in the figure, the current memory usage is also updated after a merge. The algorithm exits if the memory usage reaches the user-specified limit.

Figure 3.5 presents a pseudocode of the Correlation-Aware Variable-Sized Clustering algorithm. Function `correlation_aware_clustering` is the top-level function, which merges clusters until the current memory usage reaches the user-defined memory limit (i.e., `CAPACITY_LIMIT`) (Line 36). It first calculates the benefit-cost ratios for all possible pairs of clusters of size one (Line 28-32) by calling `getBCRatio` (Line 9-14). It then finds the pair with the maximum benefit-cost ratio (Line 38) and merges the selected clusters (Line 40). Function `merge` assigns a new ID to the merged cluster and updates its size, features, and inverted index, as discussed before (Line 15-23). Finally, the original cluster pair is erased from the cluster set (`c`) and the benefit-cost ratio map (`pair_ratio`) (Line 41-44), and the newly merged cluster is inserted with benefit-cost ratios calculated against all the other clusters (Line 45-50).

3.4.4 Parallelization of Correlation-Aware Variable-Sized Clustering Algorithm

By exploiting the super-partition-level parallelism, our clustering algorithm can be effectively parallelized. However, naïvely scheduling each thread to execute on its own super-partition may end up violating the memory usage constraint without coordination. Thus, we address this by employing a minimum benefit-cost ratio; each thread will stop merging once the benefit-cost ratio of the next merge fails to exceed this minimum ratio, and once clustering completes for all super-partitions, the total memory consumption is

```
struct Cluster{
  int cid; int size;
  set<int> I; // Queries including any feature in this cluster
  set<int> features; // Features in this cluster
};
float getBCRatio(Cluster a, Cluster b){
  benefit = intersection(a.I, b.I);
  cost = pow(2, a.size + b.size) - pow(2,a.size) - pow(2,b.size) + 1;
  return benefit/cost;
}
Cluster merge(Cluster a, Cluster b, int &nextcid){
  /* Merge cluster i and cluster j */
  Cluster merged;
  merged.cid = nextcid++;
  merged.size = a.size + b.size;
  merged.features = union(a.features, b.features);
  merged.I = union(a.I, b.I);
  return merged;
}
// S: The number of initial clusters(features)
void correlation_aware_clustering (set<Cluster> &c){
  /* Pair of clusters and its benefit-to-cost ratio */
  map<int, map<int,float>> pair_ratio;
  /* Initial evaluation of benefit-cost ratio for cluster pairs */
  for i=0 to S-1:
    for j=i+1 to S-1:
      pair_ratio[i][j] = getBCRatio(c[i], c[j]);
  int nextcid = S;
  /* Repeat merging until user-specified capacity limit */
  capacity = S;
  while (capacity < CAPACITY_LIMIT):
    /* Returns cluster indices for the maximum ratio */
    (i, j) = argmax(pair_ratio);
    /* Merge two selected clusters */
    Cluster merged = merge(c[i], c[j], nextcid);
    /* Remove two clusters from c */
    c.erase(c[i]), c.erase(c[j]);
    /* Remove pairs containing c[i] or c[j] from pair_ratio*/
    pair_ratio.erase(c[i].cid), pair_ratio.erase(c[j].cid);
    /* Update ratios for cluster pairs including merged*/
    for cl in c:
      pair_ratio[cl.cid][merged.cid] = getBCRatio(cl, merged);
    /* Add merged cluster info */
    c.insert(merged);
    /* Recompute memory usage here */
}
```

Figure 3.5: Pseudocode of the Correlation-Aware Variable-Sized Clustering Algorithm.

computed. The minimum ratio is automatically adjusted according to the calculated memory consumption, and the process continues until the expected

memory consumption converges to the user-specified limit. This parallelization technique enables our clustering algorithm to utilize multiple cores in parallel, reducing its runtime by a significant factor (e.g., 9.7× on a 16-core machine with 32 hardware threads).

3.5 MERCI Online Query Processing

This section explains how MERCI utilizes the clusters identified by the offline clustering scheme in Chapter 3.4 to create a table structure for memoization (Chapter 3.5.1). Note that this is a one-time process performed when new clusters are formed. Then we present how MERCI exploits this data structure to serve incoming queries for embedding reduction (Chapter 3.5.2).

3.5.1 Preprocessing

Preprocessing for memoization consists of two steps. First, MERCI remaps feature IDs to identify the cluster for each feature with only a few additional memory access. Second, the memoization table that stores partial reduction for various combinations of embedding vectors is constructed. Below, we explain each step in detail.

Step 1: Remapping Feature IDs. At runtime, MERCI needs to identify the cluster that a specific feature within a query belongs to. The naïve solution would be to maintain a mapping table that maps each feature ID to a pointer to its cluster information. However, such an approach will incur additional memory access, since it is likely that not all mapping tables and data structures containing information on each cluster are cached. Instead, our approach statically remaps feature IDs before deployment to minimize the information needed to access at runtime.

Figure 3.6 (❶, ❷) illustrates the feature ID remapping process. ❶ First, clusters with the same size are grouped into a cluster group, and all cluster groups are sorted by descending order of cluster size. The order of the clusters within a cluster group is irrelevant. ❷ Then, starting from the first feature

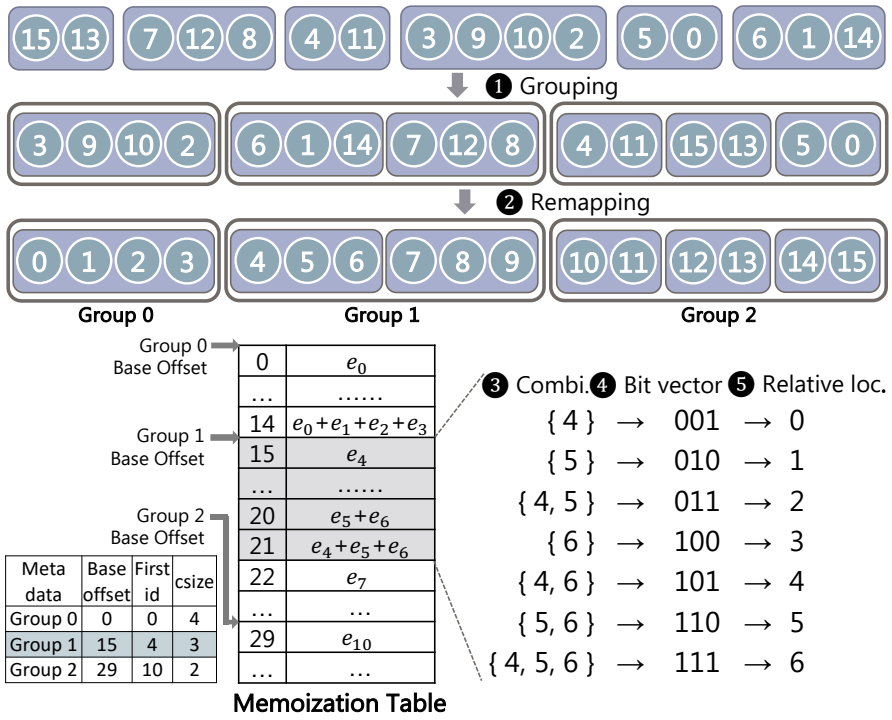


Figure 3.6: Preprocessing Phase of Online Query Processing.

(ID 3 in the first cluster) to the last feature (ID 0 in the last cluster), we assign new IDs 0 to N-1 in order, where N is the total number of features. With this feature ID remapping, features in the same cluster or a cluster group have contiguous feature IDs.

Step 2: Memoization Table Construction. Once the remapping completes, MERCI constructs the memoization table containing partial sums of all $2^{cluster\_size} - 1$ feature combinations in each cluster. The memoization table is a 2D array (implemented as a 1D array) storing R vectors ($R = \sum_{\forall clusters}(2^{cluster\_size} - 1)$), each having D dimensions (i.e., identical to the embedding vector dimension). This memoization table follows the order of clusters determined during the feature ID remapping step.

Within a single cluster, we utilize the following mechanism to determine the partial sum's location for a specific combination. Figure 3.6 (❸-❺) shows a simple example of filling out memoization table for cluster {4, 5, 6}. ❸ First, we generate all possible combinations for every cluster. ❹ Then we

represent the combination as a one-hot bit vector whose kth least significant bit is set when it contains the kth feature. For example, a combination of 2nd, 3rd feature (i.e., {5, 6}) is represented as $110_{(2)}$. ❺ Then, the integer representation of this number minus one (except for the empty set) is utilized as the relative offset of the partial sum within the cluster. This relative offset is then added to the base offset of the cluster to find the absolute location in the memoization table. For example, in the figure, seven partial sums are stored in the memoization table consecutively starting from base offset 15.

To facilitate the retrieval of the partial reduction results during the runtime, MERCI utilizes a tiny additional metadata array, as shown in Figure 3.6. For each cluster group, the base offset within the memoization table for this cluster group, as well as the first feature ID and cluster size (`csize`) of this cluster group are stored.

3.5.2 Query Processing

When a batch of queries arrives, they are distributed to each thread, and all threads run a subset of queries in parallel. For each query, MERCI iterates through features and identifies the clusters they belong to using remapped feature IDs and the cluster group meta-data array. If multiple features in a query fall into the same cluster, this implies an opportunity for partial reduction as their partial sum is already memoized. Thus, MERCI retrieves sub-query partial sums for all clusters and calculates final reduction results.

Figure 3.7 illustrates how MERCI processes a query in detail. Given a query {0, 3, 4, 6, 7, 9, 12}, MERCI processes each feature sequentially starting from the first feature in the query (i.e., feature 0). For each feature, MERCI finds out each feature's cluster group by comparing its ID with groups' first IDs in group meta-data array. Figure 3.7 assumes features 0, 3, 4 and 6 are already processed, and feature 7 is about to be processed. Feature 7 is in Cluster Group 1 because ID 7 is less than Group 2's first ID 10 but greater than Group 1's first ID 4. Then, its cluster ID can be obtained by dividing offset within a cluster group (feature ID - group first ID) by cluster size (`csize`) of

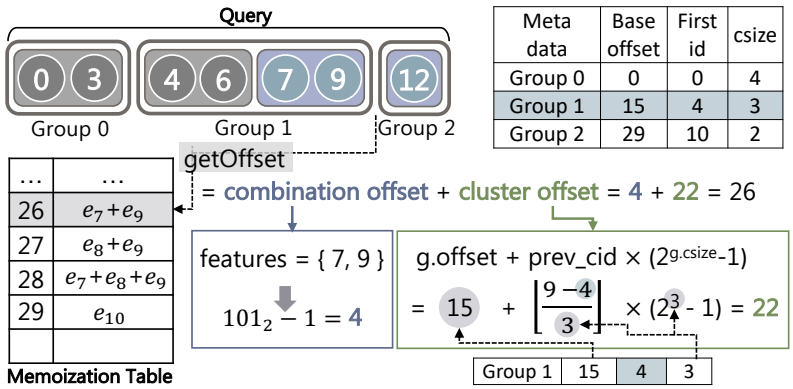


Figure 3.7: Illustration of MERCI Query Processing.

that group. Hence, Feature 7 is in Cluster 1 ($= \lfloor \frac{7-4}{3} \rfloor$). Likewise, Feature 9 is in Cluster Group 1, Cluster 1, and Feature 12 is in Cluster Group 2, Cluster 0. Then, MERCI detects a change in the cluster and collects previous features in the same cluster ($\{7, 9\}$) to calculate the memoization table offset of their reduction. The location of cluster's reduction results (i.e., *cluster offset*) is computed by {cluster group base offset} + cluster ID$\times(2^{cluster_size} - 1)$. For instance, Cluster Group 1, Cluster 1's *cluster offset* is 22 ($= 15 + 1\times\ (2^3 - 1)$). The exact offset is *cluster offset*+*combination offset*, which can be obtained by representing the current feature set using a bit vector, as explained in Chapter 3.5.1 and Figure 3.6.

Figure 3.8 again shows this process in a pseudocode. Function `query_processing` stores reduction results of B queries. For features in a query, it identifies their cluster group IDs (`g.gid`) and the cluster IDs (`cid`) (Line 23, 36). Then it compares `g.gid` and `cid` of the current feature (`fid`) with that of the previous feature to collect features in the same cluster (Line 24-26). If the cluster has changed, combination offset (Line 3-7) and the previous feature set's cluster offset (Line 9) is calculated in function `getOffset`, and the reduction result is updated (Line 31-32). Note that the last feature set of the query needs to be handled after the loop, but we omit that part for brevity.

```
int getOffset (GroupInfo& g, int prev_cid, vector<int> features){
  int combi_offset = 0;
  int first_f = features[0];
  /* Iterate features for bit vector representation */
  for f in features:
    combi_offset |= (1 << (f-first_f));
  /* Compute cluster offset */
  int cluster_offset = g.offset + prev_cid*(pow(2, g.csize)-1);
  return combi_offset + cluster_offset;
}
void query_processing (vector<int> query[B], float &res[B][D]){
  for qid = 0 to B-1:
    /* Initialize cid & gid with first feature */
    GroupInfo g = getGroup(query[qid][0]);
    int prev_cid = (query[qid][0] - g.first_id)/g.csize;
    int prev_gid = g.gid;

    vector<int> features; // Features in the same cluster
    for fid in query[qid]:
      int cid = (fid - g.first_id)/g.csize;
      /* cluster not changed */
      if (prev_cid == cid && g.gid == prev_gid):
        features.push_back(fid);
      /* cluster changed */
      else:
        /* Get memoization table index for features */
        int offset = getOffset(g, prev_cid, features);
        for d=0 to D-1:
          res[qid][d] += memoization_table[offset][d];
        prev_cid = cid;
        prev_gid = g.gid;
        features = {fid};
        GroupInfo new_g = getGroup(fid)
        /* Group changed, update group info */
        if(prev_gid != new_g.gid)
          g = new_g
    /* Omitted: Handle the last cluster here */
}
```

Figure 3.8: Query Processing Pseudocode (omitting the details for multi-threading).

3.6 Discussion

3.6.1 Time Complexity of Correlation-Aware Variable-Sized Clustering

Correlation-Aware Variable-Sized Clustering is performed on each super-partition, and hence it only considers clusters within the same super-partition as potential merge candidates. The time complexity of our clustering scheme is

$O(NS|Q|)$, where N is the number of embedding vectors (i.e., the number of features), S the size of each super-partition, and $|Q|$ the number of queries in the training set. For each merge, the scheme needs to evaluate the benefit and the cost of merging the different pairs of clusters. Here, there exist at most S remaining clusters, and evaluating the benefit of merging two clusters requires at most $|Q|$ operations as it is simply an intersection of two inverted indices whose size is bounded to $|Q|$. As a result, each merge requires at most $O(S|Q|)$ operations. In the worst case, the merge needs to be performed N times (i.e., S merges for N/S super-partitions), making the total time complexity $O(NS|Q|)$. We have empirically confirmed that a choice of small S (e.g., 128) is nearly as effective as a larger S such as 1024, and expect that even larger S does not substantially boost the performance. This implies that the number of co-appearing features for a single feature does not exceed 128 on average in the datasets used for evaluation.

If we assume that there was only one super-partition (e.g., $N = S$), the time complexity of a single merge becomes $O(N|Q|)$. In the worst case, the merge needs to be performed for N times, meaning that the time complexity without the hypergraph partitioning step (Chapter 3.4.1) is $O(N^2|Q|)$. This is impractical, especially given that N is often an order of millions. Therefore, the step of hypergraph partitioning is justified to keep the time complexity manageable.

3.6.2 Capacity Cost

Some large-scale recommendation systems [212, 213] already require an enormous capacity to store embeddings, and at a glance, it may seem applying MERCI on such systems is impractical due to the additional capacity cost of memoization. For such models, naively using MERCI for all embedding tables may incur an excessive capacity cost. To avoid such a huge capacity cost, we envision that it is possible to selectively apply MERCI for some embedding tables (or a subset of a single embedding table) that are i) frequently accessed, ii) reasonably sized, and iii) have high locality. In fact, several existing

literature state that the large-scale recommendation model utilizes multiple separate embedding tables [127], and some embedding tables exhibit higher locality than others [60]. Furthermore, MERCI allows users to specify the limit of capacity overhead from the memoization table.

3.6.3 Handling Embedding Table Updates and Query Access Pattern Changes

Embedding vectors are known to be frequently retrained every few hours [60]. In that case, the memoization table needs to be updated as well. However, the time to update the memoization table is relatively smaller than the time to retrain the embedding vectors. And simply updating the embedding vectors does not require MERCI to perform offline clustering (i.e., Hypergraph partitioning and Correlation-Aware Variable-Sized Clustering) again. In contrast, when the query access pattern changes, hypergraph partitioning (Chapter 3.4.1) and clustering (Chapter 3.4.2) need to be performed again. In practice, this happens much less frequently than the embedding vector change in many recommender systems. For all our workloads, clustering and partitioning are completed within 10 minutes with a 16-core machine (the same as in Chapter 3.7). Naturally, this time can be further reduced with the use of a better machine or the use of multiple machines.

3.7 Evaluation

3.7.1 Datasets

Real-world Datasets. For the assessment of our algorithm, we utilize popular public datasets for the recommender systems: the Amazon Review dataset (books, electronics, clothing, shoes, and jewelry, sports and outdoors, office products, and home and kitchen) [89], Last.fm Million Songs dataset [25], and DBLP Co-Authors Network dataset [164]. Although it would be ideal to use the feature traces from actual recommender models such as Meta DLRM [139], such production traces are not publicly released.

Table 3.1: Dataset Analysis.

Name	# of Features	# of Queries	Avg. Query Len.	Embedding Tbl. Size	MemTable Size (+8×)
Synthetic datasets					
Synthetic 1	1,000K	1,000K	60.0	244MB	2.08GB
Synthetic 2	1,000K	1,000K	54.0	244MB	2.06GB
Synthetic 3	1,000K	1,000K	51.0	244MB	2.19GB
Synthetic 4	2,000K	2,000K	60.0	488MB	4.18GB
Synthetic 5	2,000K	2,000K	54.0	488MB	4.09GB
Synthetic 6	2,000K	2,000K	51.0	488MB	4.36GB
Real-world datasets					
Books	3,187K	32,305K	72.796	568MB	5.20GB
Electronics	759K	10,711K	55.746	115MB	1.06GB
Clothing	2,345K	4,137K	81.953	224MB	2.06GB
Sports	1,506K	5,998K	96.019	196MB	1.75GB
Office Products	599K	3,736K	64.088	85MB	0.73GB
Home & Kitchen	1,806K	11,270K	51.476	248MB	2.23GB
Last.fm	636K	534K	95.611	104MB	0.88GB
DBLP	540K	479K	61.780	102MB	0.87GB

Each dataset was parsed into a format suitable for our use. Queries and features were defined in a way they would be in a recommender system. For instance, in the Amazon Review dataset, we defined a feature as a product for sale on Amazon, and query as a group of products (features) a reviewer bought or viewed together. Then, queries were randomly partitioned into train and test sets at the ratio of 8:2. Queries in the train set are used by Correlation-Aware Variable-Sized Clustering to calculate the benefit and cost during offline clustering, and queries in the test set were utilized to simulate query processing with the memoization table. Specific statistics regarding each dataset are delineated in Table 3.1.

Synthetic Datasets. We also evaluate MERCI on synthetic datasets. For synthetic dataset generation, we employ a technique named Stochastic Block Model (SBM) [195], which is a well-known approach for creating a random graph with community structures. We configured the parameters such that 128 features form a single correlative group, and the total number of features be N. Furthermore, each query comes with an average of p features from the same group and an average of q features from different groups. We evaluated datasets with (p, q) pairs of (48, 3), (48, 6), (48, 12) for $N = 1M$ and $N = 2M$. Queries are randomly partitioned into train and test sets at the ratio of 8:2.

3.7.2 Methodology

We implemented baseline and MERCI's embedding reduction operation in C++. In both cases, queries are distributed to multiple threads. We functionally verified the correctness of the implementation and checked that the baseline implementation achieves high performance by confirming that it fully utilizes the system's memory bandwidth. MERCI implementation is available at `https://github.com/SNU-ARC/MERCI`. We measured the runtime mostly on Amazon Web Services (AWS) EC2 m5.8xlarge instance [16], which provides 16 Intel Xeon Platinum 8259CL CPU cores with 128GiB of DRAM. We also checked the performance sensitivity to machines by evaluating MERCI on desktop-class Intel Core i7-10700K CPU with 64GiB of DRAM. Note that accessing hardware counters is possible only on the local desktop, but not on the Amazon server. Thus, we perform energy and LLC miss analysis on the desktop machine. All evaluations were performed on Ubuntu 18.04 LTS.

3.7.3 Performance Evaluation

Throughput. Figure 3.9 delineates the throughput improvement of MERCI. The x-axis denotes the size of the memoization table over the original embedding table. We limited the additional memory usage incurred by MERCI's memoization table to 0.25, 0.5, 1 and 8 times the size of the original embedding table. Note that MERCI uses both the original embedding table and the

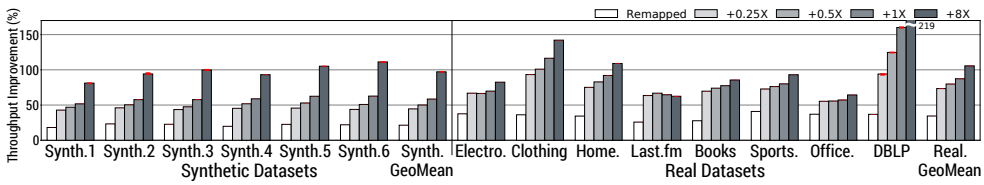


Figure 3.9: Throughput Improvement.

memoization table, so this implies pure memory consumption from memoization. All configurations in the figure (*Remapped*, +0.25x, +0.5x, +1x, +8x) hypergraph-partitioned N features into super-partitions of size 128. The embedding dimension is set to a constant value of 64 (i.e., 64 elements per embedding vector), and all measurements were repeated five times and averaged. Error bars are expressed in red lines but too minuscule to notice.

The bars labeled *Remapped* refer to *remapped-baseline* whose height denote runtime speedup without memoization. In *Remapped*, the N features are hypergraph-partitioned into superpartitions of size 128 and remapped so that those in the same super-partition are assigned consecutive IDs. Hence, *Remapped* shows the pure effect of locality-aware ID remapping without our clustering algorithm and memoization. As shown in Figure3.9, considering locality at coarse-grained granularity improves by 29%. Clearly, clustering and memoization give substantial extra speedup on top of ID remapping.

Across all datasets, MERCI manifests significant throughput improvement of 62%–262%, and achieves a geomean speedup of 102% when the memoization table size is at +8×. As shown in the figure, it is possible to obtain a decent speed up of 52%–160%, and 74% on average when the table size is limited at +1×. Even for smaller table size which is limited at +0.25× and +0.5×, MERCI achieves 60% and 66% on average, respectively. In general, increasing the memoization table size leads to further speedup, but with diminishing returns. This is because MERCI first utilizes the capacity for the most popular co-appearing combinations and then utilizes extra capacity for the less frequently co-appearing ones.

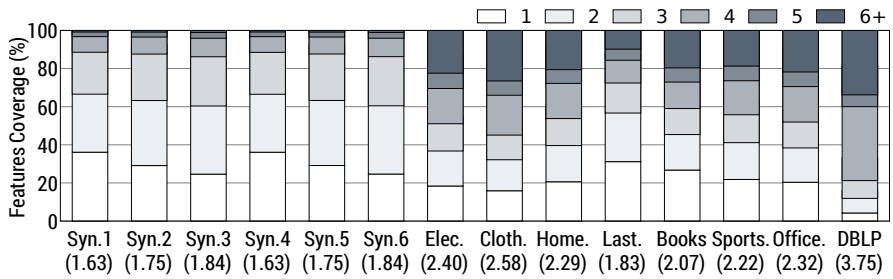


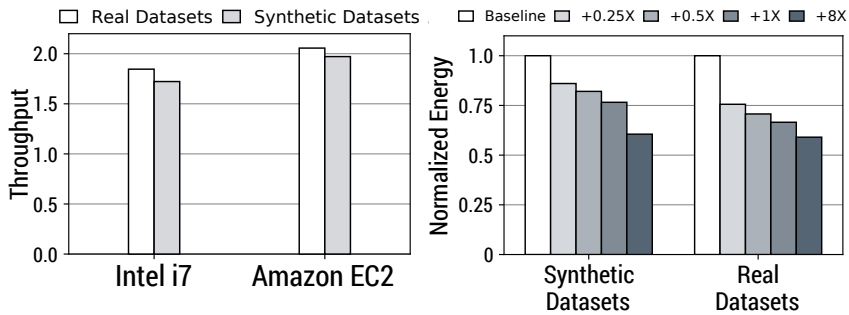
Figure 3.10: Feature Coverage Per Memoization Size.

Memoization Size Analysis. Figure 3.10 analyzes feature coverage of memoization table access by memoization size (i.e., the number of features aggregated together). This is an actual clustering result that derived the speedup in Figure 3.9. Each section in a stacked bar represents the percentage of features covered by a given memoization size, summing up to the total number of demanded features in the test set (i.e., $\sum_{qid=1}^{|Q|} query[qid].size()$). For instance, in the Amazon Office Products dataset, 18.6% of all features were retrieved as a reduction of four features, thus quartered table access count for that portion. On average, 75.4% of features were accessed by a memoization size greater than one. In the x-axis, the number in parenthesis indicates the average number of features retrieved per memoization table access. The result shows that, and single table access covers from 1.63 to 3.75 features on average.

The figure illustrates that MERCI effectively memoized the embedding table as a large portion of memory accesses reads amassed features. Even when MERCI retrieves the memoization value of size 1, it is still superior to the baseline because it benefits from locality-aware feature ID remapping, as discussed in Figure 3.9.

3.7.4 Evaluation on Desktop Platform

Machine Sensitivity. Figure 3.11a explores the MERCI performance sensitivity on different machine configurations with +8× memoization table. As shown in Figure 3.9, MERCI demonstrates largely similar performance improvements by 102% on the server system (AWS EC2 m5.8xlarge instance)



(a) Machine Sensitivity. (b) Energy Consumption.

Figure 3.11: Machine Sensitivity and Energy Efficiency.

and 78% on the desktop system (Intel Core i7-10700K CPU with two memory channels). The result indicates that MERCI can achieve speedup on any system whose embedding reduction performance is bound by memory bandwidth. Technically, MERCI may not show a performance improvement on some systems with very abundant memory bandwidth and a small number of cores. In practice, however, such systems are rare as they would heavily underutilize the memory bandwidth in many conventional operations.

Energy Savings. Figure 3.11b shows total energy consumption normalized to baseline energy consumption. We measured energy consumption with Intel Running Average Power Limit (RAPL) interface [53]. MERCI significantly saves energy consumption by 40.2% on average (up to 63.5%) at +8× configuration.

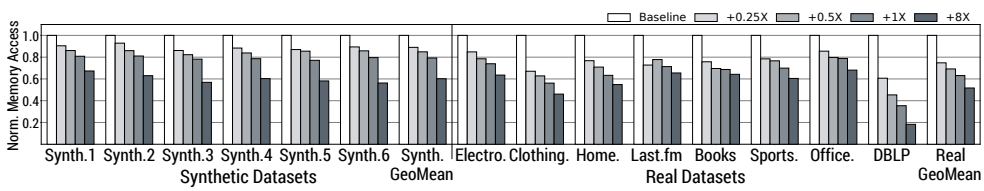


Figure 3.12: Memory Access Count Reduction.

Memory Access Count. Figure 3.12 shows memory access count (i.e., reads +writes) during MERCI's query processing normalized to baseline memory access count. For memory access count measurement, we utilized Intel VTune

Profiler [46]. As in previous sections, memoization table sizes were set at +0.25, +0.5, +1 and +8 times the original embedding table, and the embedding dimension was set to a constant value of 64. The graph shows that MERCI's total memory access count decreases by 48%, 40% for real and synthetic datasets at +8× memoization table. MERCI successfully accelerated memory-bound embedding reduction operation by reducing the actual count of memory accesses, which decreases as we use more memory for storing memoization results. We also measured the memory bandwidth utilization of both the baseline and MERCI using Intel VTune Profiler. Both systems almost fully utilize the available memory bandwidth (e.g., 90+% of the theoretical peak bandwidth), and this indicates that the memory access reduction shown in Figure 3.12 directly translates to the throughput improvement.

3.8 Related Works

3.8.1 Frequent Pattern Mining

Frequent pattern mining algorithms such as apriori [13], FP-growth [83], and DHP [146] algorithm can be utilized to identify sets of frequently co-appearing features. However, the main drawback of these algorithms is that they simply find multiple sets of co-appearing features, allowing a single feature to belong in multiple sets. In such a case, unlike our clustering approach, retrieving the reduction results becomes much more difficult. Specifically, i) identifying the relevant partial sums for a given query and ii) finding where they are located become serious challenges. It is our design choice to give up some extra reduction opportunities for the efficient retrieval of partial sums.

3.8.2 Hardware Solutions for the Embedding Reduction

Recently, Meta [60, 79, 139, 147], Google [54], and Alibaba [190] emphasize that embedding reduction is memory-bound and takes a significant portion of runtime. Several works addressed this problem with hardware support. For example, the work [110, 115] adopted near-memory processing (NMP) archi-

tecture to exploit the abundant internal bandwidth to perform reduction, and only passes the reduction outcome to the external device through links with lower bandwidth. Centaur [92] is a chiplet-based hybrid accelerator that also includes embedding reduction as its target. These solutions report that they achieve up to an order of magnitude performance improvements or traffic reductions based on their simulation results. However, solutions that require hardware support are often expensive. On the other hand, our proposal is an immediately deployable solution that provides a substantial speedup at the cost of extra memory capacity.

3.8.3 Feature-aware Optimizations

Bandana [60] utilizes hypergraph partitioning to place embedding vectors that are likely to be accessed together in the same 4KB NVM block. Bandana aims to reduce DRAM capacity consumption under the same number of memory access counts while our work aims to reduce the number of memory access count itself.

3.8.4 Memoization

Since the first introduction of memoization [133], memoization is widely adopted as a key technique to accelerate specific target. COREx [66] scales datacenter accelerators via memoization. Specifically, it proposes an accelerator and a storage layer that memoize and reuse the outcome of previously accelerated computations when the accelerator needs to compute the same thing. Other proposals [28, 44, 124, 162, 187] identify computation redundancy caused by similarities in the input within the various granularity (e.g., instruction, function, task level) and memoize them. Thus these works avoid processing the same set of instructions and rather replace such memoized regions with much simpler operations. In the work [205], they propose a technique that can be used to accelerate memoization.

Chapter 4

Hardware-Software Co-design for Efficient, Lightweight Self-Attention Mechanism in Neural Networks

4.1 Overview

The self-attention mechanism has a strong ability to capture relations within input entities at the cost of high computational cost. Based on the intuition that irrelevant relations in self-attention operation, which barely impact the final outcome, can be effectively filtered out by computing approximate similarity, we present ELSA, a hardware-software co-designed solution for efficient, lightweight self-attention. With a novel approximation algorithm, ELSA substantially reduces computational waste in a self-attention operation. Unlike conventional hardware such as GPUs, which fails to benefit from the proposed approximation, our specialized hardware directly translates this reduction to further improve performance and energy efficiency. This reduced cost of self-attention enables us to apply self-attention to larger data, which can uncover distant relations within the data that today's models cannot handle effectively.

Despite several other methods [40, 42, 51] attempting to reduce the computational cost of self-attention operations, their approach is often limited to manipulating operations such as changing the order of operations or factorizing/tiling the operation. As a result, the computation reduction ratio remains static, as these methods do not account for the unique properties of datasets. In contrast, ELSA considers the relationships among input entities to filter out unnecessary relationships, rather than treating self-attention as a simple

computation of numbers. This approach effectively reduces computational requirements by taking into account the properties of datasets, which can lead to greater reductions in datasets with strong local relationships among input entities. In addition, ELSA presents new potentials for research focused on decreasing the computational cost of self-attention operations. It does so by illustrating how customized hardware accelerators can convert the reduction in computation into an acceleration in speed and an improvement in energy consumption beyond what GPUs are capable of achieving.

4.2 Opportunities for Approximation

All three input matrices ($\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$) of the self-attention are dense. In other words, they mostly consist of nonzero elements. However, not all elements of these matrices contribute equally to the output. This is because the softmax operation maps most of the values in the attention score matrix ($\mathbf{S}$) to zeros or near-zero values except for the few largest values of the row. It effectively makes $\mathbf{S}'$ a sparse matrix with many near-zero values, and hence the final matrix $\mathbf{S}'\mathbf{V}$ as well. Simply performing the sparse matrix multiplication for the second matrix multiplication ($\mathbf{S}'\mathbf{V}$) does not completely mitigate the high cost of the self-attention, since the first matrix multiplication $\mathbf{Q}\mathbf{K}^T$ still requires n^2d multiplications. To fully exploit the approximation potential in the self-attention, there should be a way to identify the set of keys (for each query) that will result in large attention scores, without performing expensive n^2d multiplications.

Our intuition is that it is possible to achieve this by performing an approximate and lightweight similarity computation. Instead of performing d multiplications and the softmax operation to identify whether the ith query and the jth key will be relevant or not (i.e., if s'_{ij} will be near-zero or not), an approximate similarity can be computed to quickly filter out a key that is expected to be not very relevant to the query. If this approximate similarity computation indicates that they are potentially relevant, the exact dot product

similarity is computed. If not, this similarity computation and all subsequent computations can be skipped. With this scheme, it is possible to eliminate a large amount of computational waste, and our specialized hardware can translate this reduction into performance improvement as well as energy savings.

4.3 Approximate Self-Attention

4.3.1 Overview

Our approximate self-attention scheme consists of three sub-operations. First, we estimate the angle between two vectors (e.g., a key and a query) with minimal computation by utilizing the concise representations (e.g., k-bits hash, also called binary embedding) of the key and the query (Chapter 4.3.2, Chapter 4.3.3). Second, an estimated angle is utilized to compute the approximate similarity between a query and a key (Chapter 4.3.4), based on the intuition that the dot product is directly proportional to the cosine of the angle between two vectors. Finally, the approximate similarity is compared with a certain threshold (Chapter 4.3.5) to identify whether a specific key is relevant to the query or not.

4.3.2 Binary Hashing for Angular Distance

Sign Random Projection. Sign random projection (SRP) [32] is a well-known technique that effectively maps each input vector to a binary hash vector in a way that allows the original angular distance between two vectors to be efficiently estimated with the two corresponding binary hash vectors. This mapping is often utilized for locality-sensitive hashing schemes, which is an approximate nearest neighbor search algorithm that utilizes SRP, but our work focuses on its use as an efficient estimator for the angular distance.

For this process, a random d-dimensional vector v is initialized by setting each of its components to a value sampled from the normal distribution $N(0, 1)$. Then, for an input vector x, the hash bit value of 1 is assigned if $v \cdot x \geq 0$ and assigned 0 otherwise. This is repeated for k times with k random

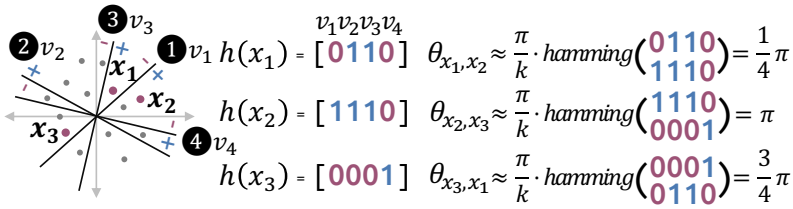


Figure 4.1: Visualization of Sign Random Projection (SRP).

vectors $v_1, ...v_k$ to construct k-bits binary hash $h(x)$ for the input vector x. Formally, the hash function is defined as follows.

$$h(x) = (h_{v1}(x), h_{v2}(x), ...h_{vk}(x)) \text{ where } h_v(x) = sign(v \cdot x)$$

Here, $sign(x)$ is a function whose value is 1 if $x \geq 0$ and 0 otherwise. It is proven that the Hamming distance between hashes of the vector x and y (i.e., $hamming(h(x), h(y))$) is an unbiased estimator of their angular distance [32]. Intuitively, if two vectors are on the same side for many of the random hyperplanes each defined by one of k random vectors $v_1, ...v_k$, they are more likely to have a smaller angle. For example, Fig. 4.1 shows that x_1 and x_2 are on the same side of three random hyperplanes out of four, and thus have a small hamming distance as well as angular distance. The following equation is used to estimate the angle between vector x and y [32].

$$\theta_{x,y} \approx \frac{\pi}{k} \cdot hamming(h(x), h(y))$$

Our work, in fact, employs the slight variant of SRP that utilizes the k orthogonal vectors generated with the modified Gram-Schmidt Process [71]. Utilizing the orthogonal vectors prevents two or more random vectors from pointing to a similar direction, which leads to an unnecessary emphasis on that specific direction. This method is proven to reduce the error of the angular distance approximation [102].

Angle Correction. The estimated angle computed from the hamming distance is not biased but still has errors. For this reason, if we simply utilize this estimator without any correction, the estimated angles will be larger than the true angle in about half of the cases. Since overestimating the angle (i.e.,

underestimating the similarity between two vectors) can result in our scheme missing the keys that have relations with the query, we subtract the bias θ_{bias} from this estimator. Specifically, we set θ_{bias} to be the 80th percentile error of this estimator so that subtracting this bias from the angle makes this estimator underestimate angles in 80% of the cases. The 80th percentile error is obtained by experiments on a synthetic dataset with standard random normal vectors. For a specific case $d = 64$ and $k = 64$, θ_{bias} is 0.127.

4.3.3 Efficient Hash Computation

Cost of Hash Computation. To obtain the k-bits hash value for a d-dimensional vector x, a $k \times d$ orthogonal matrix (i.e., a matrix whose row vectors are unit vectors orthogonal to each other) is multiplied to x, and then each element is assigned a hash bit (i.e., 1 if it is positive; 0 if not). With this scheme, computing the hash values for n vectors requires ndk multiplications (as well as $n(d-1)k$ additions), and since our scheme requires computing hashes for all queries and keys, the total number of multiplications required for hash computation is $2ndk$. This cost is negligible compared to $2n^2d$ (cost of dot product similarity computation and value matrix computation) when $n \gg k$. However, at least for current neural networks with the limited n (e.g., 128 for small models), this is not always the case. To minimize the amount of computation for hash computation, our work exploits Kronecker product, a technique to efficiently compute the matrix multiplication using orthogonal matrices [68, 210].

Kronecker Product. The key intuition of our approach is that we can utilize a structured orthogonal matrix for hash computation. Specifically, we utilize an orthogonal matrix which can be computed by the *Kronecker product* of smaller matrices. A Kronecker product of a $m \times n$ matrix $\mathbf{A}$ and $p \times q$ matrix $\mathbf{B}$ produces the $pm \times qn$ matrix as shown below.

$$\text{Kronecker Product: } \mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & \dots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & \dots & a_{mn}\mathbf{B} \end{bmatrix}$$

It is well known that Kronecker product of orthogonal matrices results in an orthogonal matrix. Thus, it is possible to obtain the $k \times d$ orthogonal matrix through Kronecker products of smaller orthogonal matrices. This characteristic allows us to utilize the technique [68, 210] to efficiently compute the hash value of the vector x, which is obtained by computing $\mathbf{A}x$.

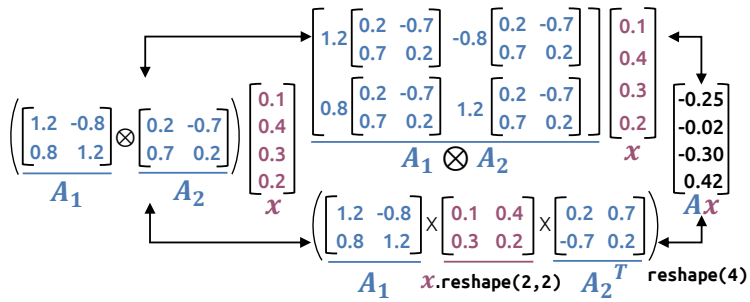


Figure 4.2: An Example of Efficient Computation with Kronecker Product.

$$\mathbf{A}x = (\mathbf{A}_1 \otimes \mathbf{A}_2)x = (\mathbf{A}_1 x\texttt{.reshape(8,8)}\mathbf{A}_2^T)\texttt{.reshape(64)}$$

Efficient Computation with Kronecker Product. Fig. 4.2 visualizes an example case of computing matrix $\mathbf{A}x$ with much fewer computations for a 4×4 matrix $\mathbf{A}$, which is represented as Kronecker product of two 2×2 matrices $\mathbf{A_1}$ and $\mathbf{A_2}$. Similarly, the above equation shows the case for $k = d = 64$ where the 64×64 matrix $\mathbf{A}$ is represented as Kronecker product of two matrices. Here, `x.reshape(8,8)` represents the operation of reshaping 64-dimensional vector x to a 8×8 matrix by dividing the vector by 8 slices and stacking them. With this technique, the amount of multiplications involved in this operation is now reduced to 1024 (i.e., $2d^{3/2}$) from 4096 (i.e., d^2).

$$\begin{aligned}\mathbf{A}x &= (\mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \mathbf{A}_3)x \\ &= (\mathbf{A}_2(x\texttt{.reshape(4,4,4)}\mathbf{A}_3^T)^{T(0,2)}\mathbf{A}_1^T)^{T(0,2)}\texttt{.reshape(64)}\end{aligned}$$

Similarly, the technique can be applied to obtain orthogonal matrix A by computing Kronecker product of three smaller 4×4 matrices $\mathbf{A}_1, \mathbf{A}_2, \mathbf{A}_3$ using the above equation. Here, $T(0, 2)$ means the tensor transpose which maps element with index (i, j, k) to (k, j, i). With this scheme, three batched (with

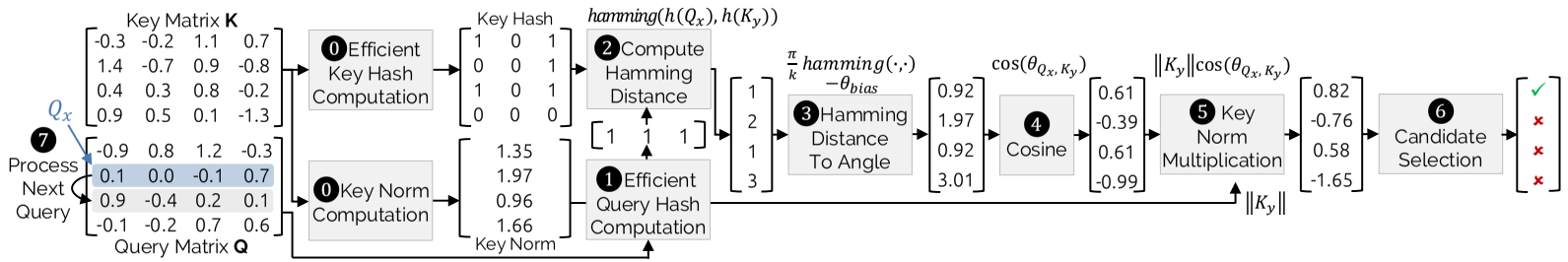


Figure 4.3: Approximate Self-attention Algorithm.

batch size = 4) 4×4 multiplications are required to compute Ax. In other words, this requires a total of twelve 4×4 matrix multiplications which involve 768 (i.e., $3d^{4/3}$) multiplications. Note that the explained efficient computation mechanism also works for cases where $k \neq d$ or A is not a square matrix [210].

4.3.4 Approximate Self-attention Algorithm

Fig. 4.3 illustrates our approximate self-attention algorithm. Below, we explain each sub-operation of the approximate self-attention algorithm in detail.

Preprocessing. ❶ At the beginning, k-bits hash values for keys (Chapter 4.3.2) are computed with the efficient hash computation scheme (Chapter 4.3.3). At the same time, the norm of each key is computed and stored as well. This preprocessing requires $3nd^{4/3}$ multiplications for the hash computation and nd multiplications as well as n square root computations for the norm computation. Note that it is possible to compute query hashes during this phase. However, for now, we assume that the query hash is computed when that query is processed so that it matches well with the hardware architecture explained in the next section. Also, note that preprocessing phase is done only once for n queries since n queries share the same set of n keys. Thus, even though preprocessing incurs an extra computational cost, it is trivial compared to the cost of original self-attention (i.e., $O(n^2d)$) and also note that a significant amount of computational cost will be saved later by skipping operations related to the keys that are not relevant to the query.

Approximate Similarity Computation. Once the preprocessing ends, the approximate dot product similarity between a query and each key needs to be computed to determine whether they are relevant or not. For a query (Q_x) and each key ($K_y \in \{K_1, ...K_n\}$), the following computations are performed. ❶ First, the query hash value $h(Q_x)$ is obtained using the efficient computation scheme in Chapter 4.3.3. ❷ Second, the Hamming distances between a query hash and all keys are computed. ❸ Third, these Hamming distances are translated to angles θ_{Q_x,K_y} for all $1 \leq y \leq n$ using the equation in Chapter 4.3.2, and the θ_{bias} is applied. ❹ Fourth, the cosine function is applied to

each of these approximate angles, and then ❺ the corresponding key norm is multiplied to each of them. Note that the resulting value is the estimate of the dot product between the normalized query and the key, which represents the (query-normalized) similarity of those two vectors. The following equations illustrate this relation.

$$Sim(Q_x/\|Q_x\|, K_y) = (Q_x/\|Q_x\|) \cdot K_y = \|K_y\| \cos(\theta_{Q_x,K_y})$$
$$\approx \|K_y\| \ \cos\Big(\max(0, \frac{\pi}{k} \cdot hamming(h(Q_x), h(K_y)) - \theta_{bias})\Big)$$

❻ Finally, once the above values are computed, we inspect these values and compare them with a constant threshold to determine whether these values are relevant to the query or not. The method to determine this threshold is explained in the next subsection. ❼ At this point, the candidates for the current query have been selected, and the next query is processed (starting from step ❶). Each approximate similarity computation between a key and a query involves i) single Hamming distance computation, ii) a multiplication ($\frac{\pi}{k}$) and a subtraction (θ_{bias}), iii) a cosine function, iv) and another multiplication ($\|K_y\|$). This cost is substantially lower than d multiplications required to compute the exact dot-product similarity. Furthermore, Chapter 4.4.3 shows we can avoid some of these computations in hardware using a lookup table.

4.3.5 Candidate Selection Threshold

Motivation. There can be several different ways to filter out irrelevant keys for a particular query based on the approximate similarity. One possible way is to sort the score and select a certain number of top-scoring elements. However, sorting has $nlogn$ time complexity and is difficult to efficiently implement in hardware, especially when n is large. For these reasons, our work focuses on filtering out potentially irrelevant keys by comparing those keys' approximate (query-normalized) similarities with a pre-defined threshold. One major issue is that different layers and sub-layers utilizing self-attention often require different thresholds since each (sub-)layer often exhibits a different distribution of attention scores. However, it is impractical to leave these layer-specific

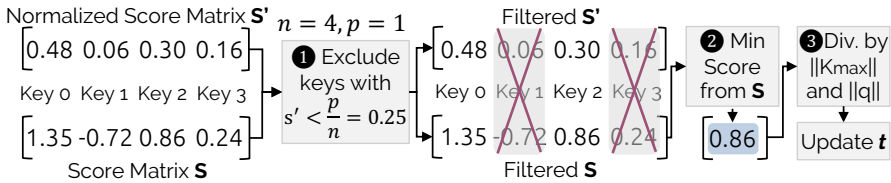


Figure 4.4: Process of Identifying Layer-specific Thresholds.

threshold values as user-defined hyperparameters, especially for models like BERT-large which has 384 sub-layers utilizing the self-attention mechanism. To avoid such an impracticality, we let a user specify a single hyperparameter that represents the degree of approximation, and present a scheme that automatically finds the (sub-)layer-specific thresholds that correspond to the user-specified degree of approximation.

Learning Layer-Specific Thresholds. To find the layer-specific threshold, our scheme runs target neural network model inference on the training set and inspects the characteristics of each layer utilizing self-attention. Fig. 4.4 illustrates this process. First, for each invocation of the self-attention operation for a particular (sub-)layer, our scheme inspects the softmax-normalized attention scores for each query. Then, ❶ we identify the set of keys whose softmax-normalized attention score exceeds $p \cdot \frac{1}{n}$ where p is a user-specified hyperparameter, and n is the number of input entities. Here, the hyperparameter p represents the degree of approximation. For example, if $p = 2$ when $n = 200$, this means that the user considers entities whose softmax-normalized score exceeds 0.01 to be relevant. The selection of a larger p implies aggressive approximation and a smaller p means conservative approximation. ❷ Among those keys, we focus on the key with the minimum softmax-normalized attention score[1]. ❸ Then, we normalize its original attention score by dividing it with the query norm $\|q\|$ and the maximum key norm $\|K_{max}\| = max(\|K_1\|, ..., \|K_n\|)$. We denote the resulting value as the threshold t. This process is repeated for multiple input data in the training set to find the average of this value for each

[1]Note that there exists a case where all softmax-normalized attention scores are below $p \cdot 1/n$ (this can happen when $p > 1$). In such a case, we simply take the maximum score among all keys.

(sub-)layer. During an actual inference run, the threshold t multiplied by the maximum key norm ($t \cdot \|K_{max}\|$) is compared with the approximate similarity (Chapter 4.3.4) to determine whether a key (in the key matrix $\mathbf{K}$) is relevant to the current query. Specifically, the following equation specifies the condition to determine if the computation for the key K_y can be skipped for query Q_x.

$$t \cdot \|K_{max}\| \geq \|K_y\| \cdot \cos\left(\max(0,\ \frac{\pi}{k} \cdot hamming(h(Q_x), h(K_y)) - \theta_{bias})\right)$$

4.4 ELSA Hardware Architecture

4.4.1 Motivation

Hardware specialization is a well-known approach to improving performance and energy efficiency of a specific type of computation. Naturally, this idea can be applied to the self-attention operation, which accounts for a substantial portion of total execution time in many emerging NN models of today. However, we also emphasize more important, often overlooked, benefits of building specialized hardware–exposing unique optimization opportunities for the specific operation that cannot be exploited profitably by the conventional hardware.

We make this point with the proposed approximation algorithm as an example. As explained in Chapter 4.3.4, the key idea of ELSA approximate attention is to avoid d-dimensional dot product through a hamming distance computation between binary embeddings, multiplication, and a cosine function. Unfortunately, the conventional GPU is not suited for many of these operations, and our internal experiments have found that the approximation scheme results in a 3.14× slowdown because simply performing d-dimensional dot product is faster than performing the approximate similarity computation, even with various manual/automated optimizations for CUDA implementation (e.g., TorchScript Tracing [153]). We find that the true benefits of the proposed approximation scheme can be harnessed only by a specialized hardware that is co-designed with this approximation algorithm. This is where a software-hardware co-optimization uncovers the unique opportunity that pure hardware or software-only optimizations fail to exploit.

4.4.2 Hardware Overview

For the efficient processing of the self-attention operation, we design a specialized hardware accelerator that exploits the novel approximation scheme introduced in Chapter 4.3.4. One can view the ELSA accelerator as a specialized functional unit for the self-attention mechanism, which can be integrated with various computing devices such as CPUs, GPUs, and other NN accelerators. The host device can issue a simple command to initiate the ELSA accelerator and pass the inputs (i.e., key/query/value matrix and n). When a device with scratchpad memories such as GPUs or NN accelerators is used, matrix inputs (and output buffer) can be passed by reference so that the accelerator can directly read those inputs without making another copy. Once inputs are ready, the accelerator goes through the preprocessing/execution phase and then writes the output matrix to the output memory and notifies the host.

Operation Overview. Figure 4.5 shows the block diagram of the ELSA accelerator pipeline, which also presents its high-level dataflow. The ELSA accelerator takes a key matrix, a query matrix, and a value matrix as inputs for self-attention to generate the output matrix. As soon as inputs are ready, the preprocessing phase begins. This phase computes k-bits hash values of each row in the key matrix using a *hash computation module*, and stores them in the key hash memory. Similarly, the norm of each key vector is computed using a *norm computation module* and stored in the key norm memory. Once this phase ends, the execution phase begins where each row of the query matrix is processed in sequence to output a single row of the output matrix at a time. Specifically, for each query, P_c *candidate selection modules* retrieve P_c keys' hashes and norms (along with the query hash) every cycle and outputs up to P_c selected candidate key IDs (i.e., row IDs) to each module's output queue. Then, these selected key IDs are arbitrated and passed to the *attention computation module*, which computes and accumulates the selected key's contribution to the output (for the current query) every cycle. Once all selected keys for this

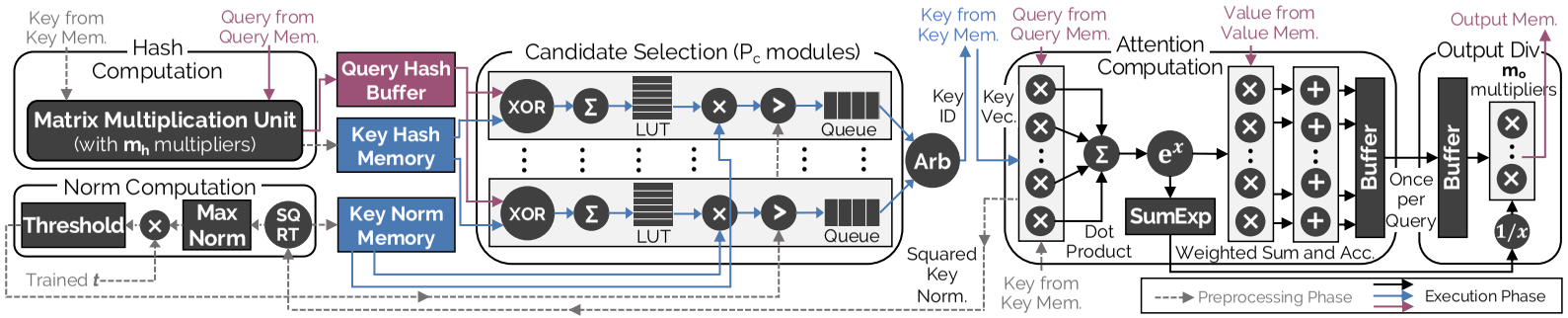


Figure 4.5: ELSA Pipeline Block Diagram.

particular query are computed, *output division module* performs the division on this output. This process is repeated for each row of the query matrix (i.e., each query), and the operation ends when the last query is processed.

4.4.3 Design of Hardware Modules

(1) Modules for Approximate Self-attention Computation

Candidate Selection Module. The candidate selection module performs the approximate self-attention mechanism (Chapter 4.3.4) to identify the set of potentially relevant rows in the key matrix (i.e., candidates), and then outputs the indices of such elements to the attention computation module. Every cycle, this module takes three inputs: i) k-bits hash value of a key from the key hash memory and ii) the norm of this key from the key norm memory, and iii) k-bits hash value of the current query from the query hash buffer. Then, this module utilizes k-bits XOR unit followed by an adder to compute the Hamming distance between the key hash value and the query hash value. The resulting Hamming distance value is then used as an index to access the pre-populated lookup table, which stores $\cos(\pi/k \cdot d_{Hamming} - \theta_{bias})$. Since the Hamming distance takes an integer value between zero and k, this lookup table has $k+1$ entries. Once this value is retrieved, it is multiplied with the norm of the current key to compute the approximate similarity (Chapter 4.3.4). This value is compared with the product of threshold t (Chapter 4.3.5) and the largest vector norm of the key matrix (i.e., $t \cdot \max(\|K_1\|, ..., \|K_n\|)$). If the approximate similarity is greater than this value, the key in question is selected as a potentially relevant key, and the index of this key is then passed to this module's output queue. Multiple (i.e., P_c) candidate selection modules process different keys in parallel, and then their outputs are arbitrated and passed to the attention computation module. The candidate selection module is fully-pipelined and processes one key per cycle.

Attention Computation Module. A single attention computation module is in charge of computing a single row of the final output matrix, along with the output division module. Figure 4.6 represents this module's operation in

```
1  def attention_computation (float q[], float key[][], float val[][],
2                                                vector<int> candidates):
3     for keyid in candidates:
4        /* Dot-Product*/
5        parallel for i = 0 to d-1:
6           temp[i] = key[keyid][i] * q[i]
7        score = ParallelSum(temp)
8        /* Exponent Computation */
9        score = exp(score)
10       sumexp += score
11       /* Weighted Sum */
12       parallel for i = 0 to d-1:
13          output[i] += score * val[keyid][i]
14 def output_division (float output[], float sumexp):
15    reciprocal = 1/sumexp
16    /* Division */
17    for i = 0 to d/mo-1:
18       parallel for j = 0 to mo-1:
19          output[i * mo + j] *= reciprocal
```

Figure 4.6: Pseudocode of Attention Computation and Output Division Modules.

pseudocode. Each cycle, this module takes a key as input from the arbiter with the longest-queue-first scheduling policy. Then, it first computes the dot product between a key (K_y) and a query (Q_x) using its d multipliers and an adder tree (Line 5-7 in Figure 4.6). After that, for the softmax normalization of the resulting attention score, the exponent of this value is computed using a lookup table (explained in Chapter 4.4.5). The resulting exponentiated value is i) accumulated in the sum of the exponent register (Line 10), and ii) multiplied with all components of the corresponding value matrix row using the other set of d multipliers and accumulated with d adders (Line 12-13). This module is fully-pipelined and can process a single candidate every cycle. Assuming c candidates are selected for the query Q_x by the candidate selection modules, this module can process them in about c cycles. The resulting output vector and the sum of exponentiated values are then passed to the output division module when it finishes processing all selected keys for the current query.

Output Division Module. Once all (selected) keys are processed, all components of the output vector need to be divided by the accumulated exponentiated score to complete the softmax normalization. For this purpose, the hardware first utilizes a reciprocal unit (Chapter 4.4.5) to compute the reciprocal of the sum of the exponentiated score (Line 15), and then multiply each component of the output vector with m_o multipliers (Line 18-19). Since

this module is fully pipelined, it can handle a single query every d/m_o cycle. This module operates in parallel with the rest of the pipeline (e.g., candidate selection and attention computation modules). However, when other modules are processing the ith query, this module is processing the $(i-1)$th query.

(2) Modules for Key/Query Hash & Norm Computation

Hash Computation Module. This module is in charge of computing hashes for the keys and the queries by performing a series of matrix multiplications as described in Chapter 4.3.3. Specifically, if we assume the specific case presented in Chapter 4.3.3 (i.e., utilizing three-way Kronecker products of 4×4 matrices for $k = d = 64$), the hash computation for a vector requires a total of twelve $(4 \times 4,\ 4 \times 4)$ matrix multiplications (the last paragraph in Chapter 4.3.3). Assuming m_h multipliers for this unit, we carefully design the matrix multiplication unit so that it fully utilizes all m_h multipliers to perform this operation and complete the hash computation in $768/m_h$ (i.e., $3d^{4/3}/m_h$) cycles. For these matrix multiplications, this module contains 48 ($3d^{2/3}$) registers, where each register value is an element of three pre-defined (4×4) matrices for the hash computation (i.e., A_1, A_2, A_3 in Chapter 4.3.3). Once the matrix multiplications are finished, the sign bits of each component (a total of k-bits) are concatenated and stored in the key hash memory. During the preprocessing phase, this module computes all key hashes ($768n/m_h$ or $3nd^{4/3}/m_h$ cycles) and the first query hash (extra $768/m_h$ or $3d^{4/3}/m_h$ cycles). During the execution phase, this model computes the hash value for the next query while the rest of the pipeline (e.g., candidate selection and attention computation module) is processing the current query.

Norm Computation Module. Norms of the keys are computed during the preprocessing phase in addition to the hashes of the keys. The Euclidean (L2) norm of the key vector $\|K_y\|$ is obtained by computing the dot product with itself $(K_y \cdot K_y)$ and then taking its square root. For this purpose, instead of having its own set of multipliers, this unit utilizes the d multipliers and the adder tree in the attention computation module (Figure 4.5). Then, this module utilizes its own square root units (see Chapter 4.4.5 for details) to

compute the final result and store it in the key norm memory. In addition, this module also identifies the maximum key norm and multiplies the trained t by that value to compute the threshold that is used for the candidate selection modules.

(3) Memory Modules

Key Hash/Norm Memory. These memory modules are implemented as SRAM structures placed within the ELSA accelerator. These structures are initialized during the preprocessing phase and then utilized by the candidate selection module during the execution phase. Key Hash SRAM requires a total of $nk/8$ bytes storage, and Key Norm SRAM requires a total of n bytes assuming an 8-bit representation for the norm. In $n = 512$ and $k = 64$ configuration, the key hash SRAM requires 4KB, and the key norm SRAM requires 512 bytes.

Query/Key/Value/Output Matrix Memory. These matrices are inputs (Query, Key, Value) and output of the self-attention. They can be placed within the ELSA accelerator using the SRAM structures. However, since the ELSA accelerator is expected to be utilized in conjunction with a host device such as GPUs or other neural network accelerators targeting other parts of the neural network models, it is also possible to utilize scratchpad memory structures in those devices (e.g., GPU shared memory) to store these matrices. At $n = 512$ and $d = 64$, each of these matrices requires about 36KB storage space assuming 9-bits representation (including the sign bit).

4.4.4 Pipeline Design

Pipeline Configuration. For a given n and d, this pipeline takes $3d^{4/3}(n + 1)/m_h$ cycles for the preprocessing. Figure 4.7 shows the high-level view of the pipeline during the execution phase and lists each hardware module's latency to process a single query (also explained in Chapter 4.4.3). As illustrated in the figure, four hardware modules can potentially bottleneck the pipeline. It takes $\max(3d^{4/3}/m_h,\ n/P_c,\ c,\ d/m_o)$ to process a single query when c is the number of candidates selected by candidate selection modules. To avoid in-

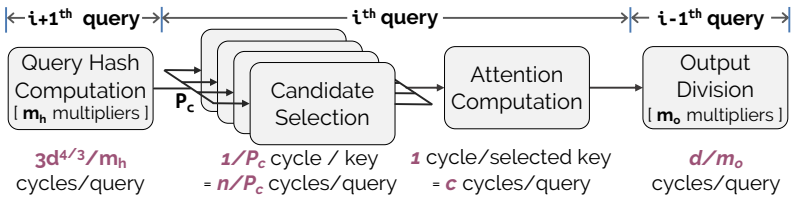


Figure 4.7: ELSA Accelerator Pipeline During the Execution Phase.

troducing the bottleneck and maximize the throughput, one should carefully select P_c, m_h, m_o to properly balance the pipeline. Specifically, it is ideal to configure parameters in a way that modules other than the attention computation module (takes c cycles) do not become a pipeline bottleneck. For example, if one aims to design a pipeline that can achieve up to 8× speedup (i.e., it takes $n/8$ or more cycles to process a query) with approximation, each of $3d^{4/3}/m_h$, n/P_c, and d/m_o should be less than or equal to $n/8$. When d is 64, a configuration such as $P_c = 8$, $m_h = 64$, $m_o = 8$ satisfies this requirement as long as $n \geq 96$. With this configuration, the achieved speedup is $min(n/c, 8)$. That is, the speedup is often (i.e., $c \geq n/8$) determined by the effectiveness of the approximation scheme, which reduces the number of keys to process (i.e., c) for the attention computation module.

Parallel Pipeline. We extend the pipeline so that ELSA can utilize multiple attention computation modules in parallel by exploiting the fact that each row of the key/value matrix can be processed independently. To extend the pipeline to utilize P_a attention computation modules in parallel, the key matrix, the value matrix, and the key hash/norm need to be stored in a banked on-chip memory where each bank contains n/P_a keys, values, and key hashes/norms. Then, for each bank, P_c candidate selection modules and a single attention computation module are connected so that they process the set of keys (and values) within a single bank and compute the partial sum of the output as well as the exponentiated score. At the end of each query, such partial sums are passed to the output division module, which sums up these values using an adder tree (requires an extra set of $(P_a - 1) \cdot m_o$ adders) and computes the final output. To avoid a specific stage of the pipeline or the spe-

cific phase from forming a bottleneck, pipeline configuration parameters such as m_h (# of multipliers in hash computation module) and m_o (# of multipliers in output division module) may need to be adjusted. This is because the throughput of candidate selection modules and attention computation modules are increased by $P_a\times$ compared to the ones shown in Figure 4.7. We find that $m_h = 256$ and $m_o = 16$ work well for $P_a = 4$. For further throughput, the whole ELSA accelerators (including its memory elements) can be replicated to exploit batch-level parallelism as well (e.g., our evaluation utilizes a set of twelve ELSA accelerators to exploit batch-level parallelism).

4.4.5 Design Details

Number Representation. The elements of key, query, value matrix are represented in a fixed-point form with a single sign bit, five integer bits, and three fraction bits. The elements of predefined matrices for the hash computation are represented with a fixed-point form with a single sign bit and five fraction bits. The rest of the pipeline utilizes the minimal necessary integer bitwidth to avoid overflow while maintaining the number of fraction bits. We use custom floating-point representations (e.g., a single sign bit, ten exponent bits, and five fraction bits) to represent the output of the exponent function as well as following computations on it to cover their huge value range. We empirically verified that the use of these number representations has a negligible impact ($<$0.2%) on model evaluation metric loss across various models when compared to the FP32 baseline.

Choice of n and d. n represents the maximum number of input entities for self-attention. For a model running very small NLP micro-benchmarks like GLUE [189], a small n (e.g., 128) is sufficient. For longer text such as question-answering benchmarks [116, 159], a larger n (e.g., 512) is often utilized to capture the relation between distant tokens. An even larger n (e.g., 800, 1024) is utilized for tasks like text summarization [121], and text generation [155, 168]. For evaluation, we configure the hardware to fit the largest workload we run, which has $n = 512$. We utilize $d = 64$, which all our evaluated models

originally used. ELSA accelerator can be designed for any n or d, and once synthesized, it efficiently run with any model or input that has smaller n or d.

Choice of Hash Length k. In general, higher k results in a better approximation since the estimate for the angle between two vectors becomes more accurate. However, too large k increases i) the cost of hash computation, ii) key hash storage area, and iii) area/power of the candidate selection modules. For such reasons, we find that $k = d$ is a choice that works well as long as k is not too small (e.g., less than 16). In case where $k > d$, batches of orthogonal vectors are utilized to generate k hash bits [102]. Since all our evaluated workloads use $d = 64$, we set $k = 64$ as well.

Hyperparameter Tuning. Our main hyperparameter p (Chapter 4.3.5) determines the degree of approximation. We recommend the user tune p with the validation dataset so that the model maintains a user's desirable accuracy while improving the performance and energy efficiency. Note that this tuning process is simple since p is a hyperparameter that (almost) monotonously increases accuracy as its value decreases. Finally, a user can set p to 0 to easily fall back to the exact version when the highest accuracy is desired.

Special Functional Units. The exponential computation unit computes e^x with $e^x = 2^{(\log_2 \mathrm{e})x} = 2^{\mathrm{frac}((\log_2 \mathrm{e})x)} \cdot 2^{\mathrm{floor}((\log_2 \mathrm{e})x)}$. For $2^{\mathrm{frac}((\log_2 \mathrm{e})x)}$, it utilizes 32-entry lookup table where fractional exponents of 2 are stored. For reciprocal unit, a simple lookup table with 32-entry is used to obtain the reciprocal of a floating point with 5 fraction bits. For the square root unit, a Taylor-expansion-oriented scheme named tabulate and multiply [95, 184] is utilized.

4.5 Evaluation

4.5.1 Workloads

We evaluate several representative self-attention-oriented NN models to demonstrate the effectiveness of the ELSA. For natural language processing models, we select three of the most popular ones: Google BERT (large) [57], Meta RoBERTa (large) [122], and Google ALBERT (large) [118]. We utilize open-source implementations of those models from HuggingFace [196] (BERT,

RoBERTa), FairSeq [143] (RoBERTa), and Google ALBERT repository [70] (ALBERT). For all three NLP models, we run Stanford Question Answering Dataset (SQuAD) [159] 1.1 & 2.0, and RACE dataset [116], which is a large-scale reading comprehension dataset from examinations. For RoBERTa, we additionally run IMDB review sentiment analysis dataset [128]. In addition to these NLP models, we also evaluate ELSA with self-attention-oriented sequential recommendation models such as SASRec (3-layers model) [107] and BERT4Rec (3-layers, 2-head model) [178] with MovieLens 1M dataset [86].

4.5.2 Accuracy Evaluation

Methodology. We extend the self-attention layer in each NN model with our approximation scheme and measure the model's end-to-end accuracy metric (i.e., F1 score for SQuAD, raw accuracy for RACE/IMDB, and NDCG @10[194] for recommendation models) on the test set or the validation set (for the workloads whose test set is not publicly available). For conciseness, we simply refer to these metrics as *accuracy* throughout this section.

Impact of Approximation. Fig. 4.8 shows the impact of approximation on end-to-end model accuracy (lines) as well as the portion of selected candidates[2] (bars) across a varying degree of approximation hyperparameter p. In general, the small p implies conservative approximation with a relatively small accuracy degradation, while the larger p implies more aggressive approximation. For most of the model-workloads combinations, it is possible to achieve sub-1% accuracy loss by only inspecting less than 40% of the total entities as candidates (i.e., $p = 1$). Furthermore, the figure also shows that it is possible to achieve sub-2% accuracy loss by inspecting about 26% of the total entities on average ($p = 2$). As discussed in Chapter 4.4.5, the user can experiment with the train set or the validation set to determine the degree of approximation (p) that provides a reasonable accuracy loss.

[2]Many software implementations operate with the fixed size n (e.g., 512). If the input text has fewer than n tokens, the software implementation pads the input so that it gets n tokens. We exclude such paddings for the normalization, and the figure shows the portion of selected candidates among the real tokens.

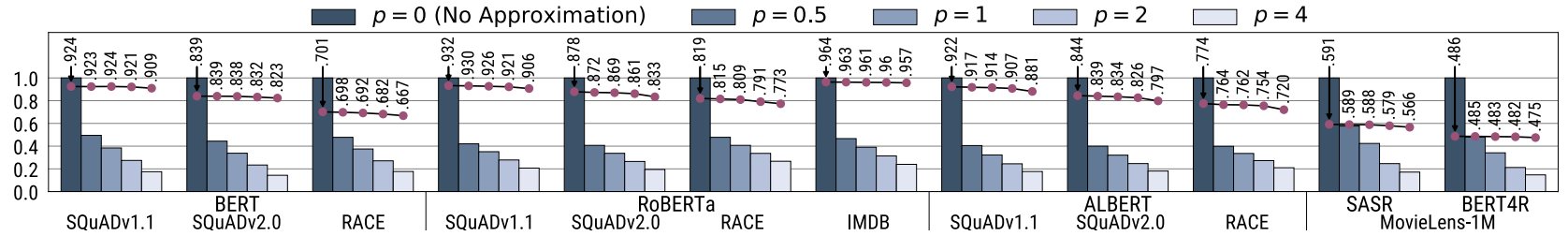


Figure 4.8: Impact of Approximation. Model accuracy (**lines**) and portion of selected candidates (**bars**) across varying degree of approximation.

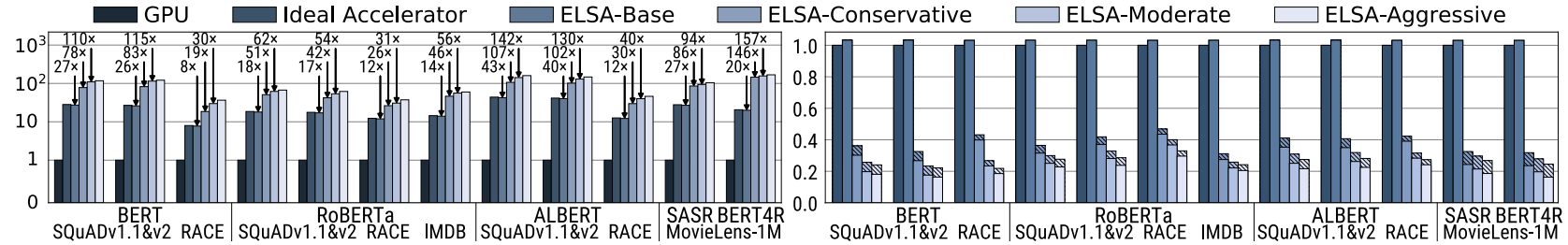


Figure 4.9: Throughput and Latency Improvement. (a) Normalized self-attention throughput (left) and (b) Normalized self-attention operation latency (right) on various devices. Hatched area on the right figure represents the time spent on preprocessing.

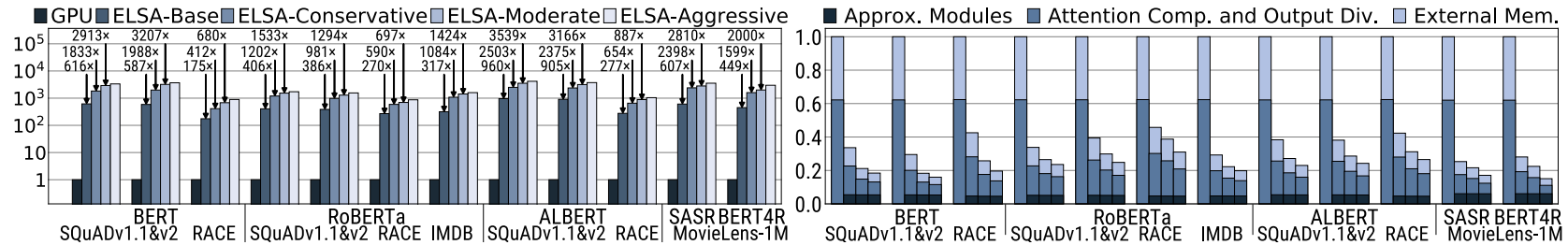


Figure 4.10: Energy Improvement. (a) Normalized energy efficiency (performance/W) (left) and (b) Energy consumption breakdown (right) of the ELSA accelerators: from left-to-right, each bar represents ELSA-base, conservative, moderate, and aggressive.

4.5.3 Performance Evaluation

Methodology. For performance evaluation, we implement a custom simulator for ELSA that is integrated with the PyTorch/TensorFlow implementations of the NN models. For GPU performance evaluation, we utilize a system with the six-core Intel Xeon Gold CPU [94] and the Nvidia V100 GPU [142] with 16GB memory. For each workload, the batch size achieving the best throughput is selected. For ELSA performance evaluation, we use a set of twelve ELSA accelerators, each running at 1GHz and configured as follows: $P_a = 4$, $P_c = 8$, $m_h = 256$, and $m_o = 16$. We specifically evaluate twelve ELSA accelerators so that their peak throughput (= 1.088 TOPS/accelerator $\times$ 12 $\approx$ 13 TOPS) approximately matches with the Nvidia V100 GPU having 14 TFLOPS peak throughput (with FP32[3]).

We select p for each NLP model-workload combination whose worst-case accuracy loss is bounded by 1%, 2.5%, 5% to call them ELSA-conservative, moderate, aggressive, respectively. For recommendation models, 0.5%, 1.0%, 2.0% drop in NDCG@10 metric is used to determine p for those configurations. We also evaluate ELSA-base configuration with no approximation. Finally, we compare ELSA configurations with an *ideal* accelerator, which can sustain 100% peak FP throughput at 1GHz frequency, while having the same number (i.e., 528) of multipliers with the ELSA-base accelerator. This is effectively an upper-bound of performance for the other matrix multiplication accelerators *without* approximation.

Throughput. Fig. 4.9(a) presents the throughput of the self-attention across different platforms. The figure shows that a set of ELSA-base accelerators achieve substantially better throughput (i.e., 7.99-43.93×) than the GPU, indicating that its specialized architecture can effectively accelerate the self-attention operation. Overall, the speedup of the ELSA-base over GPU varies

[3]Nvidia GPUs can achieve better raw inference throughput by utilizing the FP16 format accelerated with the tensor core. However, its iso-peak-FLOPS throughput will be lower in this case since the actual throughput increase from FP16 inference is often much lower than 8× increase in *peak throughput*. Thus, using FP32 throughput gives an advantage to GPU in calculating the normalized throughput.

across workloads and models. Variations across workloads are mostly attributable to the actual number of input entities. There are some inputs where the data has fewer entities than the maximum number of entities the model supports (i.e., n). In such cases, the GPU implementations pad the data and perform the matrix multiplications with n rows. However, ELSA accelerators (and the ideal accelerator) avoid computation for the padded rows and achieve higher speedup. Speedup differences across NLP models for the same dataset are mostly due to the GPU performance differences across different models and implementations. The figure also demonstrates that the conservative, moderate, and aggressive approximation scheme enables ELSA to achieve much higher geomean speedups over GPU (57×, 73×, 81×, respectively) than the ELSA-base accelerator. We find that moderate or aggressive approximation performance is sometimes bounded by the pipeline bottleneck caused by the candidate selection modules. Adjusting pipeline configuration parameters such as P_c (Chapter 4.4.4) will result in extra speedups in these cases at the expense of extra area/power.

Latency. Fig. 4.9(b) compares the average latency of performing a single self-attention operation on various models across ELSA accelerators and the ideal accelerator. As shown in the figure, ELSA-base latency is nearly identical (1.03×) to the ideal accelerator. ELSA with the approximation scheme achieves latency reduction over the ideal accelerator by exploiting the approximation opportunities. The average (geomean) normalized latency of ELSA-conservative, ELSA-moderate, and ELSA-aggressive are 0.38×, 0.29×, 0.26× of the ideal accelerator latency. Fig. 4.9(b) also shows that all workloads spend a small amount of time on preprocessing. If a further reduction in preprocessing time is desired, one can increase the m_h or use multiple hash computation modules.

Impact on End-to-End Performance. Figure 4.9 compares the throughput and latency for the *self-attention mechanism* (not the end-to-end model throughput or latency). As shown in Figure 2.6, the portion of the time spent on self-attention varies greatly across models, sequence length (i.e., input

Table 4.1: Area and (Peak) Power Characteristics of ELSA.

Module Name	Area (mm^2)	Dynamic Pwr(mW)	Static Pwr(mW)
Modules for Approximate Self-attention			
Hash Computation ($m_h = 256$)	0.202	115.08	2.23
Norm Computation	0.006	9.91	0.07
32× Candidate Selection	0.180	78.41	1.95
Modules for Attention Computation			
4× Attention Computation	0.666	566.42	7.53
Output Division ($m_o = 16$)	0.022	11.42	0.19
Internal Memory Modules			
Key Hash Memory (4KB)	0.141	139.91	1.05
Key Norm Memory (512B)	0.038	34.9	0.29
External On-Chip Memory Modules			
Key/Value Mem. (36KB ea.)	0.253	167.39	2.29
Query/Output Mem. (36KB ea.)	0.193	91.03	1.72
ELSA Accelerator			
ELSA Accelerator (1×)	1.255	956.05	13.31
External Memory Modules (1×)	0.892	516.84	8.02
ELSA Accelerators (12×)	15.06	11472.6	159.72
External Memory Modules(12×)	10.704	6202.08	96.24

length), and the model configuration (e.g., FFN dimension). With ELSA-conservative's 57× average speedup, the use of ELSA accelerators makes the time spent on self-attention to be negligible compared to the time spent on the other operations. The ELSA-conservative accelerators achieve about 1.4-2.5× end-to-end speedup across five models when the default max input length is utilized, and 2.4-5.0× speedup when the 4× larger input length is utilized. Furthermore, if other types of accelerators are utilized to accelerate the rest of the network (e.g., FC layers), the end-to-end speedup from the use of ELSA accelerators becomes even larger, since the portion of the time spent on the self-attention layer becomes larger.

4.5.4 Area/Energy Evaluation

Methodology. For area and energy evaluation, we implement the ELSA accelerator with Chisel hardware description language [41], and perform functional verification. Then, we synthesize, place and route the Chisel-generated Verilog

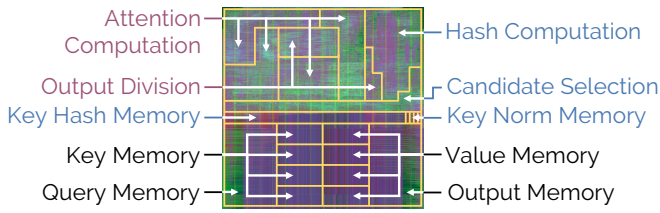


Figure 4.11: Post-layout Image of ELSA Accelerator.

code with the 1GHz target frequency using Synopsys Design Compiler [182] and TSMC 40nm standard cell library. For logic synthesis, we assume the following pipeline configuration: $n = 512$, $d = 64$, $P_a = 4$, $P_c = 8$, $m_h = 256$, $m_o = 16$.

Area. Table 4.1 reports the ELSA accelerator area characteristics and Fig. 4.11 shows the layout of the ELSA accelerator. As shown in the table, the single ELSA accelerator utilizes about 1.3mm^2 area (2.1mm^2 with external memory modules), and twelve ELSA accelerators utilize about 15.1mm^2 area (25.8mm^2 with external memory modules). On the other hand, Nvidia V100 GPU has a total die size of 815mm^2 [141]. This implies that integrating the ELSA accelerator to GPU incurs a very little area cost, and such a cost becomes even lower considering that the reported ELSA area is estimated from the 40nm technology node, while the Nvidia V100 GPU die area is from the 12nm technology node. Another important point from the area table is that candidate selection modules (32 copies) utilize a relatively little area. This proves that our approximation mechanism is very hardware-friendly.

Power and Energy Consumption. Table 4.1 shows that a single ELSA accelerator consumes about 1.49W (including power consumption from the external memory modules) and twelve ELSA accelerators consume about 17.93W at its peak. This is substantially lower than that of the Nvidia V100 GPU, which has 250W thermal design power (TDP). Furthermore, we measured the actual GPU power consumption with *nvidia-smi* tool and confirmed that the GPU is in fact operating at the power level very close to its peak (e.g., 240W+) while performing the self-attention operation in our workloads. Fig. 4.10(a) presents the energy efficiency comparison of the ELSA accelerators and the

GPU. Combining the power efficiency (over 13×) and the speedup (shown in Fig. 4.9), the ELSA-base accelerator achieves over two orders of magnitude improvements in energy efficiency (geomean 442×) over the GPU for the self-attention computation. Moreover, approximation-enabled configurations further increase the energy efficiency improvements: 1265× (conservative), 1726× (moderate), and 2093× (aggressive). Finally, Fig. 4.10(b) shows the energy consumption breakdown of the ELSA accelerators. The figure shows that our approximation scheme, despite the introduction of new hardware modules, results in the total energy reduction by significantly reducing the energy spent on attention computation and output division modules and external memory modules.

4.5.5 Discussion

Comparison with the A^3 accelerator. A^3 [82] is a recent proposal that also applies approximation to the attention. However, A^3 architecture has the following key limitations that make it not well-suited for self-attention. First, its approximation scheme requires an expensive preprocessing (i.e., sorting all columns of the key matrix). Its preprocessing relies on external hardware (e.g., GPU) that incurs significant performance/energy overheads. Unfortunately, when multiple attention accelerators are used in parallel, the preprocessing time linearly increases while the execution time linearly decreases, to make this preprocessing take the dominating portion of the runtime. Also, storing the outcome of the preprocessing requires a memory that is twice larger than the original key matrix. Second, the A^3's approximation scheme is complex (occupying over 1.7× larger area than ELSA's attention computation module) and has a very low degree of parallelism. A^3's approximation scheme can only select up to two keys (and often fewer) every cycle and is not further parallelizable. This significantly limits its ability to achieve the desired accuracy on time and prevents the use of multiple attention computation modules in parallel. For example, A^3 evaluation results state that it achieves a 1.85× speedup over its baseline accelerator without the approximation on the BERT model

running the SQuADv1.1 dataset at the expense of 1.3% accuracy loss. On the other hand, for a similar setting, ELSA-conservative/moderate configurations achieve 2.76×/3.72× speedup over the ELSA-base without approximation with lower than 1%/2.5% accuracy loss. Considering this difference in baseline configurations, ELSA approximate configurations achieve 5.96×/8.04× better raw speedup over the A^3 approximation configuration. Finally, ELSA presents a more scalable, area-efficient attention computation module design that does not require multiple n-element buffers.

Comparison with Google TPU. Google Tensor Processing Unit (TPU) [69] is specialized hardware that targets neural network training as well as inference tasks. To check its effectiveness in self-attention operation, we run ALBERT model [118] that natively supports TPU execution on Google Cloud TPUv2. Our experimental results show that ELSA-base achieves 8.3×, 6.4×, 2.4× better (peak-FLOPS-normalized) throughput[4] on self-attention operations of ALBERT running SQuADv1.1/2, and RACE datasets. For the same workloads, ELSA-moderate achieves 27.8×, 20.9×, 8.0× speedup, respectively. For references, the measured TPU (peak-FLOPS-normalized) throughput was 5.5×, 6.7×, and 5.4× better than GPU throughput for the same workloads.

NN Models with Lightweight Self-Attention. Several recent works propose changes in the NNs to reduce the computational demand of the self-attention operation. For example, some [23, 40, 42, 49, 74, 111, 154, 156, 175, 180, 193, 198, 200, 202] augment the architecture of the self-attention layer to efficiently capture the relation between a large number of entities. Our work is compatible with most of them [23, 40, 74, 154, 180, 198, 202] because they decompose a very large self-attention operation (e.g., sequence length ≥ 4096) into a sequence of multiple, smaller conventional self-attentions.

Moreover, ELSA is fundamentally different from these software approaches in that it takes a more model-agnostic approach without requiring retraining,

[4]TPUv2 has a peak throughput of 180 TFLOPS with its bfloat16 internal representation. We assume that it has 1/4× peak throughput with FP32 (45 TFLOPS) and then compute its iso-peak-FLOPS throughput by dividing the actual TPU throughput by 45/13 as twelve ELSA accelerators we used for the comparison with GPU has 13 TOPS peak throughput (instead of 180/13).

which can be very expensive computationally on a large-scale language model. Finally, most software-only approaches [23, 40, 111, 154, 156, 200] in fact fail to achieve the inference speedup for reasonable sequence length (e.g., <2048), despite a theoretical reduction in the number of operations. Specifically, a recent work [193] finds that sparse attention techniques achieve very little speedup (e.g., 20% speedup for 2% accuracy loss), and Reformer [111] fails to achieve any speedup for sequence length less than 2048, due to its huge constant in their time complexity. Even in the case of concurrent works achieving speedup on the commercial hardware for sequence length <2048, their reported speedup from approximation is around 1.3×-1.7× [42, 193], which is far less than what ELSA achieves with approximation.

4.6 Related Works

4.6.1 Hardware Support for Attention Mechanisms

A few hardware accelerators related to the attention mechanism are recently proposed. A^3 is the most closely related work, which is discussed in Chapter 4.5.5. MnnFast [99], Manna [174], and Mann Dataflow accelerator [148] are also relevant in that they contain modules that can potentially be utilized to accelerate the attention mechanism. However, their focus is on the end-to-end hardware implementation of particular neural network models without fully exploiting approximation opportunities, such as Google NTM/DNC [72, 73] for Manna and Meta End-to-End Memory Network [177] for MnnFast.

4.6.2 NN Approximation with Hardware Support

There are prior works presenting various forms of approximation strategies to improve neural network performance and energy efficiency. Specifically, works [78, 91, 97, 98, 108, 145] investigate the efficient use of quantization and low-precision operations for neural networks. Furthermore, other works [123, 158] propose the approximate MAC unit to achieve a similar goal. More closely related works are ones focusing on finding values that are less likely to affect

the final output of the neural network models. SnaPEA [14], ComPEND [119], ZAP [169], and RnR (Reduce and Rank) [157] are representive examples.

4.6.3 Hardware Accelerators for NN

Various hardware accelerators [31, 34, 36, 37, 59, 64, 81, 104, 105, 160] have been proposed to accelerate key neural network operations represented as matrix multiplications. Specifically, several proposals [15, 55, 84, 85, 113, 144, 209] focus on the sparsity of the activation and weight matrices to further accelerate such operations. Our work differs from these works in that i) we provide the unique approximation scheme that dynamically sparsifies the key matrix, and ii) specifically targets the self-attention mechanism.

Chapter 5

Specialized Architecture for Approximate Nearest Neighbor Search

5.1 Overview

Product quantization similarity search is an attractive idea with many potential benefits, but the existing hardware fails to fully harness its potential. We identify this algorithm as a great target for specialized hardware due to its unique computation and data access patterns. With the goal of efficiently finding the most similar vectors out of multi-billion database vectors, we present ANNA (Approximate Nearest Neighbor search Accelerator), a specialized hardware accelerator that can substantially improve the throughput and energy efficiency of the PQ-based approximate similarity search while flexibly supporting various search configurations.

5.2 ANNA Hardware Accelerator

ANNA is a stand-alone hardware accelerator that can be configured to perform various product-quantization-based similarity searches. ANNA supports both inner product and L2 distance search cases. Moreover, ANNA can accommodate various search configurations (e.g., metric, k^*, $|C|$, M). Before performing similarity search with ANNA accelerator, a host device first needs to i) configure ANNA by sending a search configuration and ii) place the set of necessary data structures in ANNA main memory (centroids C and encoded vectors) and ANNA's on-chip SRAM (codebook B). Then, the host sends a search command to ANNA with a query or a batch of queries as

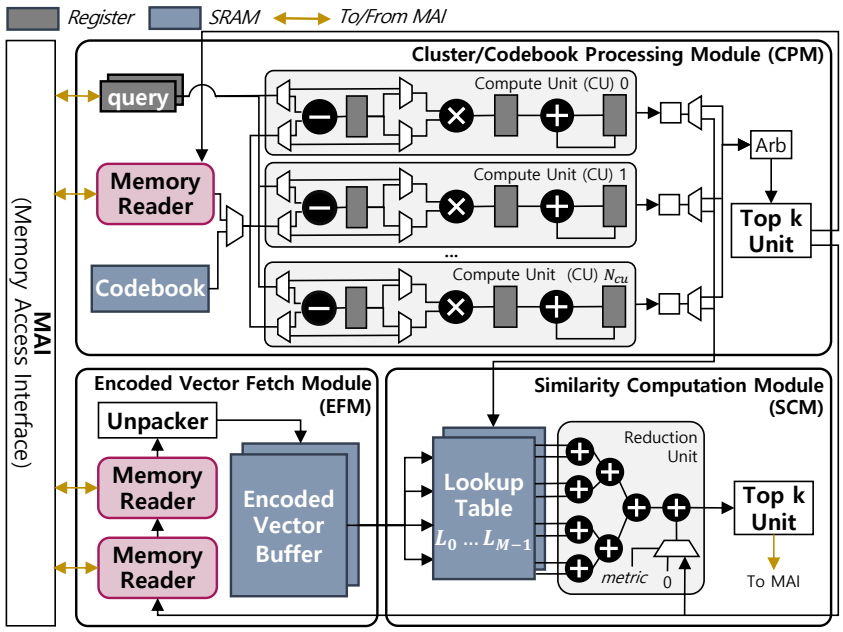


Figure 5.1: Illustration of ANNA Hardware Overview.

well as the number of similar vectors (top-k) to search for. ANNA performs a similarity search and finally returns the result to the host device. Figure 5.1 shows the overview of ANNA hardware accelerator. At a high level, ANNA consists of three main entities named Cluster/Codebook Processing Module (CPM), Encoded Vector Fetch Module (EFM), and Similarity Computation Module (SCM). Each module contains computation units, memory readers, and SRAM blocks. When provided a query, ANNA performs three steps of PQ-based ANNS outlined in Chapter 2.2.3.2.

5.2.1 Hardware Operations

Search Process (Step 1). The first step is cluster filtering, which is handled by the Cluster/Codebook Processing Module (CPM). ANNA utilizes a memory reader to read centroid vectors in a streaming manner, and these read centroid vectors and the query are passed to the compute units to compute the exact distance between two vectors. The outcome is then passed to the top-k selection unit, which selects and holds top-$|W|$ most similar centroid vectors.

Search Process (Step 2 & 3)-Inner Product. The second step is lookup table construction. In the case of the inner product, the ith lookup table L_i needs to store $\{q_i B_i[0], ..., q_i B_i[k^* - 1]\}$. In turn, the inner products between each codeword in the codebook and the corresponding sub-vector of the query are computed using the compute units, and the resulting outcome is stored in the lookup table. Once all lookup tables are filled, the third step, similarity computation, is performed by the SCM. The EFM loads the encoded vectors within a cluster selected in Step 1 from main memory, unpacks them with its unpacker hardware, and buffers them in the encoded vector buffer. The SCM reads encoded identifiers (i.e., $e_0(r(x)), ..., e_{M-1}(r(x))$) in the encoded vector buffer, and then use those identifiers as addresses to retrieve specific data in the lookup table. The loaded values are fed to the reduction unit, which computes the reduction outcome. Finally, the term $q \cdot c^{(s)}$ is added to the reduction outcome (see Chapter 2.2.3.2) to obtain the final similarity between the query and an encoded vector. This similarity value, as well as the encoded vector's ID are passed to the top-k selection unit in the SCM. This process is repeated for all encoded vectors in the selected clusters, and the final top-k selected vectors in the SCM top-k selection unit are stored in memory as the search outcome.

Search Process (Step 2 & 3)-L2 Distance. For L2 distance search, the process is slightly different. Unlike the inner product-based similarity search, the lookup table needs to be reconstructed for each selected cluster. The centroid reader first loads the selected centroid once again from memory and computes the difference between the query vector and this selected centroid vector (i.e., $q - c^{(s)}$) using the compute units. Then, the contents for the lookup table are computed, again using the compute units. At this point, the SCM is utilized to compute the similarity between the encoded vector in the currently selected cluster and the query using the lookup table. The lookup table construction and the similarity computation are repeated for selected clusters, and finally, the top-k selection unit stores the top-k selected vectors in the memory as in the inner product-based similarity search case.

Since L2 distance search requires a lookup table to be constructed for each selected cluster, such a process can potentially account for a substantial amount of runtime. To improve the performance, ANNA overlaps lookup table construction on the CPM and similarity computation on the SCM through double buffering. Specifically, ANNA maintains two copies of lookup tables and lets the CPM fill the lookup table for the $(i+1)$th most similar cluster while the SCM computes the approximate similarity between the query and the database vectors for the ith similar cluster. Once both are complete, SCM now operates for the $(i+1)$th most similar cluster using the lookup table that the CPM just filled, and the CPM fills the lookup table that the SCM stopped using. This way, lookup table construction time and similarity computation time can overlap and effectively reduce the total query processing time.

5.2.2 Design of Hardware Modules

(1) Cluster/Codebook Processing Module (CPM)

Figure 5.2 illustrates the hardware design of CPM. This module utilizes N_{cu} compute units to perform various computations. Overall, this module is utilized for three purposes: ❶ to compute the similarity between query q and each centroid in C during the cluster filtering step, ❷ to compute the residual vector $r(x) = q - c^{(s)}$ for the L2 distance search, and ❸ to compute the similarity between each codeword and the query (inner product) or residual (L2 distance).

Mode ❶ - Similarity Computation for Cluster Filtering. For the first case, the goal is to compute $s(q, c)$ between query q and every c in C. Every cycle, an element of the query vector is broadcasted into N_{cu} compute units and used as an operand. At the same time, an element of N_{cu} different centroids is supplied to each compute unit every cycle. Then, depending on the metric, the partial similarity (i.e., $q[i]c[i]$ or $(q[i]-c[i])^2$) is accumulated at the last register of the compute units. Assuming D-dimensional query vector and centroids vector, this module requires D cycles to compute the similarity between the query vector and N_{cu} centroids. To complete the cluster filtering step, this

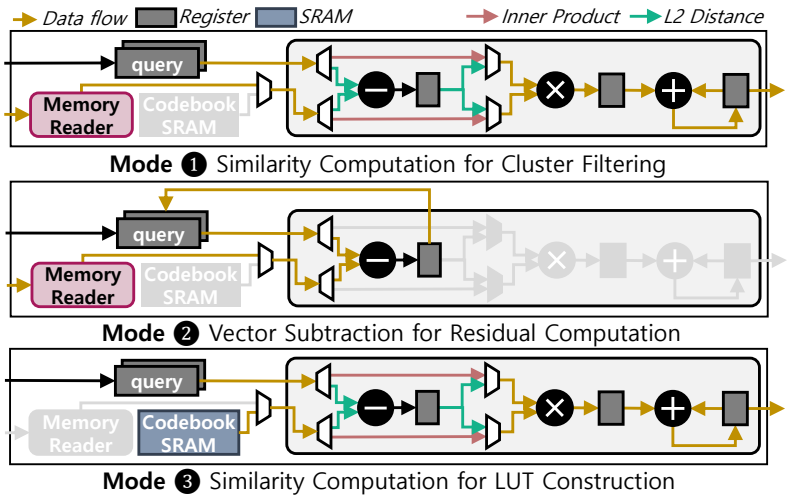


Figure 5.2: Illustration of ANNA Cluster/Codebook Processing Module.

module overall requires $D|C|/N_{cu}$ cycles.

Mode ❷ - Vector Subtraction for Residual Computation on L2 Distance Search. For the second case, this module performs a vector subtraction $(q - c^{(s)})$. Every cycle, different elements of each query, as well as the selected centroid vector (i.e., $c^{(s)}$), are sent to different compute units. For example, $q[0]$ and $c^{(s)}[0]$ are sent to the first compute unit, and $q[N_{cu}-1]$ and $c^{(s)}[N_{cu}-1]$ are sent to the last compute unit. The resulting outcome is stored in the residual register. Since each cycle processes N_{cu} elements of D-dimensional vector, this module processes residual computation in D/N_{cu} cycles.

Mode ❸ - Similarity Computation for LUT Construction. For the last case, the module is responsible for computing the values that will be stored in the lookup table. In this case, a single compute unit (CU) is responsible for computing all values that will be stored in one lookup table L_i. Specifically, in each cycle, an element of the different query sub-vectors (e.g., $q_0, ..., q_{M-1}$) is supplied to each compute unit. Also, a codeword from each codebook is supplied to different compute units. Each compute unit then utilizes these two values to compute the value that is stored in the lookup table. Over the first D/M cycles, the ith compute unit computes the value for $L_i[0]$ by utilizing sub-vector q_i and $B_i[0]$. This process is repeated for k^* times to fill up the

lookup tables, each having k^* entries. A total of $D/M \cdot k^*$ cycles are necessary to fill up N_{cu} lookup tables. Since a total of M lookup table needs to be filled up, the total number of cycles necessary to fill the whole lookup table is $D/M \cdot k^* \cdot M/N_{cu} = Dk^*/N_{cu}$ cycles.

(2) Encoded Vector Fetch Module (EFM)

This module is in charge of fetching encoded vectors of a selected cluster from the main memory and then buffering the fetched data in the on-chip buffer (named encoded vector buffer). Specifically, this module first receives the selected cluster IDs from the top-k unit. Then, its memory reader reads the cluster metadata (i.e., start address for the data within the cluster and the size of the cluster) from the main memory. The following memory reader utilizes the start address to fetch the encoded identifiers of the cluster from the main memory. The read data is passed to the unpacker hardware, which utilizes hardware shifters to unpack the packed data, and stores them in the encoded vector buffer. To overlap the data fetch and the later similarity computation, this module keeps two encoded vector buffers. When the SCM is utilizing one encoded vector buffer, this module fetches the data for the next cluster on the other data buffer. In some cases, a cluster's encoded vectors may be larger than the encoded vector buffer size. In that case, a contiguous portion of the cluster's data is first fetched, and the next contiguous portion of the cluster's data is fetched on the other buffer while the current buffer is utilized.

(3) Similarity Computation Module (SCM)

The main role of this module is to perform the approximate similarity computation, which is essentially a sum reduction of data retrieved from multiple lookup tables (See Step 3 in Figure 2.4). For this purpose, this module maintains lookup tables and an adder tree with $N_u - 1$ adders so that it can reduce N_u values every cycle. This module retrieves the set of N_u identifiers from the encoded vector buffer. Each of these identifiers is used as an address for a lookup table, and the total of N_u data is read from multiple lookup tables. These read data are passed to the reduction unit, which is a pipelined adder tree. In the case of the inner product similarity search, the reduction result is

added with the $q \cdot c^{(s)}$ supplied from the top-k unit by an extra adder. This module can perform similarity computation with a single database vector per M/N_u cycles. For example, when $M = 128$ and $N_u = 64$, the module will take two cycles to process a single entry with pipelining.

(4) Top-k Selection Unit

This unit tracks the k largest data ($k = 1000$ in our configuration) that this unit has taken as inputs during its operation. Essentially, this is a hardware priority queue. Every cycle, this unit takes a similarity score for the specific vector as an input. If the provided input is larger than the minimum of the currently tracked ones, the input is added to the structure, and the already tracked entry with the smallest score is discarded. Otherwise, the input is simply discarded and the structure remains intact. We implement P-heap hardware priority queue [26], which utilizes a binary-heap-like structure for high-throughput design. The unit consists of several SRAM buffers which can store k data and a set of comparators. This unit is designed to process a single input every cycle. It can also initialize its contents from the main memory or flush its contents to the main memory. This unit also maintains two copies of buffers so that a set of buffers can be utilized for top-k processing, while the other set of buffers can simultaneously flush/initialize its contents to/from the main memory.

(5) Memory Module

Memory Access Interface (MAI). MAI takes read requests from memory readers and issues memory read requests to the memory controller. When issuing a memory request, it reserves one of its 64B buffers and records the requested reader ID there. Then, MAI adds an entry to its associative table structure, which maintains the list of outstanding read addresses (as a key) and destination buffer ID (as a value). When the memory request returns from the main memory, it finds the matching address from the associative table and stores the read value to the destination buffer. Every cycle, the MAI utilizes an arbiter to forward one of the values in the MAI destination buffer to the memory reader that issued this memory request. For memory write requests,

MAI buffers the write data until the write completes in the main memory. In general, this is quite similar to the MSHR in CPUs.

Memory Readers. Memory readers are in charge of issuing memory requests through the memory access interface (MAI) and buffering the received inputs. The reader is configured with the start address and the amount of data that it needs to fetch from the start address. When configured, this module prefetches the data from the start address as long as the MAI can accept its read requests. Once the MAI returns the data for issued read requests, the module buffers this 64B granularity read data and forwards the portion or all of the data to the next module at the requested granularity. There are three memory readers in ANNA. Memory readers in CPM are used to read centroid vectors, and memory readers in EFM are used to fetch cluster metadata and cluster's encoded vectors.

SRAM. ANNA has three SRAM structures. First, the Codebook SRAM is sized so that it can buffer the whole codebook which is $2k^*D$ bytes (e.g., 64KB in our evaluation). This SRAM is structured to read up to $2N_{cu}$ consecutive bytes (e.g., 64B) data every cycle. Next, the lookup table SRAMs have a total capacity of $2k^*M$ (e.g., 32KB in our evaluation) bytes for a single SCM. It can handle up to N_u (e.g., 64) lookups in parallel, where each lookup returns one of k^* entries. As explained above, we maintain two copies of the lookup tables to overlap the filling of the lookup table and the similarity computation. Finally, the encoded vector buffer is used to buffer the encoded vectors in the single cluster. As with the lookup tables, two copies of the encoded vector buffer are maintained to overlap data fetch and the similarity computation. The size of this SRAM structure is a design parameter (e.g., 1MB in our evaluation). The encoded vector buffer is structured to supply N_u data per cycle.

5.3 ANNA Memory Traffic Optimization

By design, one can easily adjust ANNA's computation capability by adjusting design parameters such as N_{cu} (the number of compute units in CPM) or N_u (the number of entries which can be sum-reduced in a cycle). Alternatively,

it is possible to utilize multiple copies of ANNA modules to improve ANNA's computation throughput. In such cases, the system's performance bottleneck eventually shifts to the memory bandwidth. At that point, the only way to further improve the throughput is to reduce the traffic between ANNA and the main memory. In this section, we present an optimization scheme that can significantly reduce the memory traffic consumption of ANNA on batched similarity search scenarios.

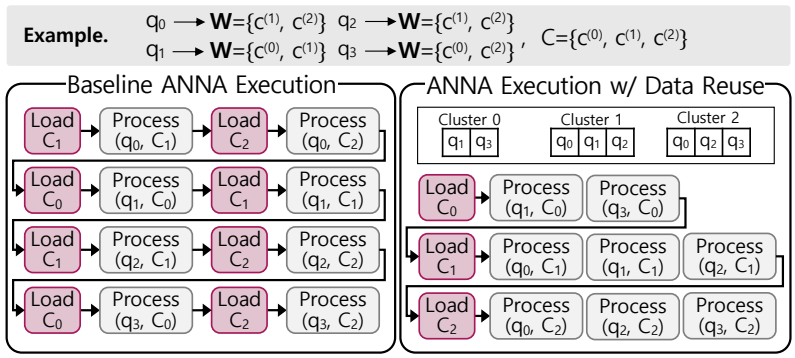


Figure 5.3: Illustration of ANNA Memory Traffic Optimization.

The key idea is to reuse the encoded vectors for a specific cluster across multiple queries. Figure 5.3 illustrates how this optimization can substantially reduce the memory traffic. In the figure, we define C_i as a set of encoded vectors in cluster i. The left of the figure shows conventional execution, where a single query is processed at a time. In such case, a query scans the encoded vectors in clusters belonging to W, whose centroids are closest to the query. This same process is repeated for the following queries. On the other hand, the right side of the figure shows the optimized execution. In this case, the set of $|W|$ relevant clusters for all queries are identified first. Based on that information, queries visiting a specific cluster are identified for all clusters. Then, each cluster is processed in series. Specifically, a specific cluster's encoded vectors are loaded and buffered on-chip, and queries visiting the cluster process the cluster using the buffered data. Once all queries visiting the cluster finish processing the cluster, the next cluster is processed for a different set of queries visiting the next cluster. Assuming each query visits $|W|$ clusters out of $|C|$

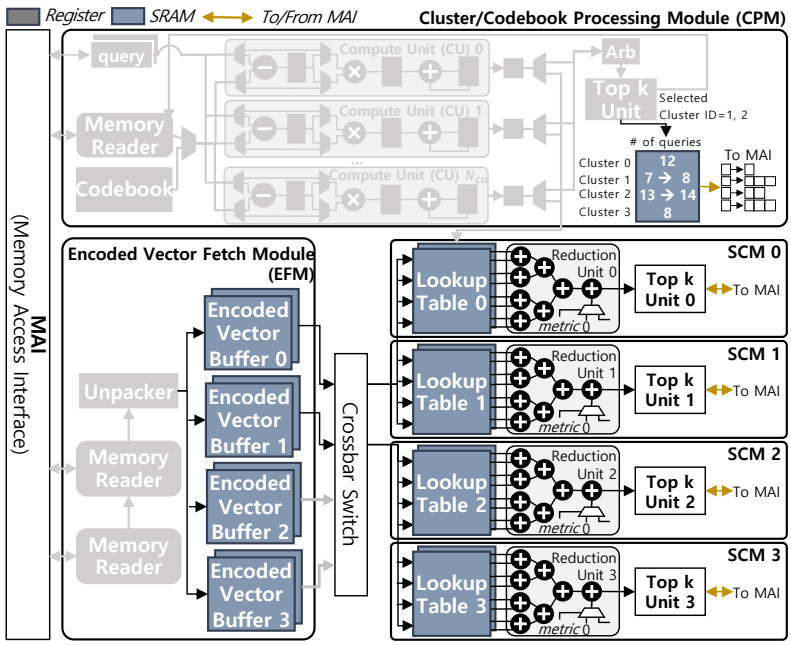


Figure 5.4: Diagram of ANNA Hardware with Memory Traffic Optimization.

clusters, and the system processes B queries at a time, the original execution scheme requires loading $B \cdot |W|$ cluster's encoded vectors. On the other hand, the optimized execution scheme requires loading mere $|C|$ clusters' encoded vectors even in the worst case. When $B = 1000, |C| = 10000, |W| = 128$, this technique leads to a 12.8× traffic reduction.

5.3.1 Hardware Extensions

Recording Queries Visiting a Specific Cluster. In the baseline ANNA, which processes a query at a time, the top-k module in CPM keeps the cluster IDs in W and then passes the cluster ID of those selected clusters to EFM so that EFM can fetch the encoded vectors for the selected cluster. On the other hand, to implement the optimization represented in Figure 5.3, ANNA first performs cluster filtering step for all queries and stores the list of queries visiting each cluster in the memory. Specifically, for this purpose, ANNA maintains an array of arrays in the main memory, where ith array keeps the IDs of the queries visiting cluster i. Also, ANNA utilizes an on-chip SRAM whose ith row stores the base address (8B) for the ith array in the main memory and

the number of queries visiting ith cluster (3B) as shown in Figure 5.4.

Once top-$|W|$ clusters for a query are identified, one of the selected cluster IDs is retrieved from the top-k module, and this ID is used to access the mentioned SRAM structure. The base address for the array associated with the selected cluster is obtained, and the number of queries in this cluster is used to compute the exact address where the query ID needs to be written. Next, the write request is passed to MAI, which performs a masked write to the main memory. This process is repeated for all $|W|$ selected clusters, and the CPM performs cluster filtering for the remaining queries.

Storing and Retrieving Intermediate Top-k Vectors. The baseline ANNA utilizes a top-k module in SCM to track the top-k most similar vectors to a particular query. This was possible since the baseline ANNA handles one query at a time. However, with the optimization, the same top-k module is now used by multiple queries processing the same cluster. As a result, once a query processes a cluster's data, its intermediate top-k results need to be stored in memory. In addition, before a query processes a cluster's data, the intermediate top-k results need to be loaded from the memory so that the top-k module correctly identifies whether the vectors in the currently processed cluster belong to top-k or not.

Improving Throughput with Multiple SCMs. Reduction in memory traffic means that more computations can happen without being bounded by the memory bandwidth. To improve the throughput, ANNA utilizes multiple SCMs (N_{SCM}) in parallel. Figure 5.4 shows the hardware extensions to support multiple SCMs. Encoded vector buffers in EFM are structured accordingly to supply data from EFM to SCMs at a higher rate. Also, a configurable crossbar switch is added to connect multiple encoded vector buffers with multiple SCMs. There are two different ways to utilize multiple SCMs at once. First, SCMs can be used to process multiple queries processing the same cluster in parallel (*inter*-query parallelism). In this case, the EFM simply broadcasts the same encoded vector to all SCMs so that each SCM having different, query-specific LUT content can process the data for each query. Alternatively, it is

possible to allocate multiple SCMs to a single query (*intra*-query parallelism). For this purpose, each encoded vector buffer, storing a portion of the cluster's encoded vectors, passes data to different SCMs. In this case, each SCM then processes a subset of the cluster's encoded vectors with its own top-k modules. Once all clusters are processed, each SCM's top-k results are merged using top-k modules.

In ANNA, a user can specify the number of SCMs that a single query utilizes during the execution. In general, it is slightly better to utilize SCMs across multiple queries since a single query utilizing multiple SCMs tends to increase the traffic for saving/restoring intermediate top-k results to the main memory. On the other hand, when a single cluster is processed by a very small number of queries, it is better to utilize multiple SCMs for a single query so that ANNA hardware fully utilizes its available SCMs. One can easily compute the average number of queries for each cluster to find the required number of SCMs per query. For example, when $B = 1000$, $|C| = 10000$, and $|W| = 40$, 4 ($= B|W|/|C|$) queries are expected for each cluster. Thus, for ANNA with 16 SCMs, we allocate four SCMs to a single query.

5.3.2 ANNA Execution with Optimization

Figure 5.5 visualizes the timeline for each compute module in ANNA (i.e., CPM and multiple SCMs) as well as its memory system in a steady-state (i.e., Step 2 and 3 of the search process outlined in Chapter 5.2.1). When the SCMs perform similarity computation for the ith cluster (takes $|C_i|M/N_u$ cycles), multiple things happen in parallel. First, the CPM works on lookup table construction for the $(i+1)$th cluster in the case of the L2 distance similarity search, which requires the CPM to compute the lookup table for every cluster. Constructing a lookup table takes k^*D/N_{cu} cycles and when there exist N_{scm} SCM modules each running a different query, the CPM needs to construct N_{scm} separate lookup tables in $N_{scm}Dk^*/N_{cu}$ cycles. Meanwhile, on the memory side, each top-k unit in SCMs stores the intermediate top-k results from its previous operations to the main memory and loads the inter-

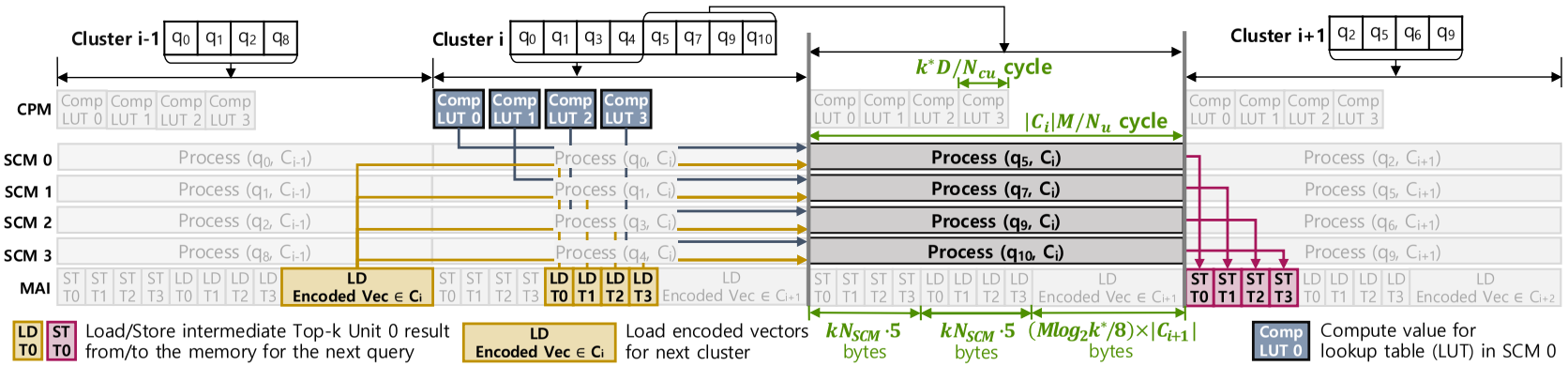


Figure 5.5: Visualization of ANNA Execution Timeline with Optimization.

mediate top-k results from the main memory for its upcoming operations. The memory traffic is $2kN_{SCM} \cdot 5B$, since each top-k unit in SCM loads/stores k entries which are 5B each (3B for vector ID, 2B for similarity score). Also, the EFM prefetches the encoded vectors for the cluster, which are $|C_{i+1}|$ encoded vectors each represented with M identifiers that are $\log_2 k^*$ bits each. Overall, the computation time for the specified time period in the figure is $max(N_{scm}k^*D/N_{cu}, |C_i|M/N_u)$ cycles, and $10kN_{SCM} + (M \log_2 k^*/8) \cdot |C_{i+1}|$ bytes needs to be fetched during this time. One should carefully set ANNA design parameters (e.g., N_u, N_{cu}, N_{scm}) so that the system is not heavily bottlenecked by computations or memory accesses.

5.4 Evaluation

5.4.1 Methodology

Dataset. We evaluate on representative similarity search datasets: SIFT1M [101] (N=1M, D=128, L2 Distance), Deep1M [20] (N=1M, D=96, L2 Distance), GloVe [151] (N=1M, D=100, Inner Product), SIFT1B [101] (N=1B, D=128, L2 Distance), Deep1B [20] (N=1B, D=96, L2 Distance), and TTI1B [1] (N=1B, D=128, Inner Product). In general, each vector within these datasets represents an embedding of an image feature or a word.

Software ANNS Implementations. We utilize two representative open-source implementations of PQ similarity search algorithms: Meta Faiss [103] and Google ScaNN [77]. Both algorithms utilize different objective functions to train codebook and thus generate different codebooks as well as the encoding for each database vector. Faiss has both CPU and GPU implementations, and ScaNN only has CPU implementation. We train each dataset for each algorithm across varying configurations and obtain trained models where each is a set of i) a list of centroids, ii) codebooks, and iii) encoded vectors. We run Faiss or ScaNN similarity search with the trained models on 8-core Intel i7-7820X with 128GB memory (both Faiss and ScaNN) and NVIDIA V100 GPU [142] with 32GB memory (Faiss) to measure their performance, energy consumption (measured with Intel RAPL & `nvprof`), and model recall $X@Y$

(i.e., the portion of retrieved top X items among submitted Y candidates). Specifically, throughout the evaluation, we utilize the following search configurations: Faiss16 (CPU), Faiss256 (CPU), Faiss256 (GPU), and ScaNN16 (CPU). Here, the number after Faiss or ScaNN (i.e., 16 or 256) represents the k^* value that the configuration utilizes. ScaNN does not support $k^* = 256$ configuration, and FaissGPU does not support $k^* = 16$ configuration since their implementations are tightly coupled with the specific k^*. Throughout the experiment, we use $|C| = 10000$ and $|C| = 250$ for billion-scale and million-scale datasets, respectively, and M varies across experiments.

ANNA Evaluation Methodology. For performance comparison of ANNA with the software implementations, we implement a custom cycle-level simulator for ANNA. We evaluate four ANNA configurations, each utilizing the trained model from the corresponding software implementations. We compare the throughput/latency of queries at a given search recall, which is our quality metric. Each ANNA configuration is assumed to be paired with the memory system providing the 64GB/s bandwidth, which is identical to the evaluated CPU-based system's memory bandwidth. For the area and energy evaluation, we implement the ANNA accelerator with Chisel HDL [41], and perform functional verification with Synopsys VCS. Then, we synthesize RTL implementations of the ANNA with the TSMC 40nm GP standard cell library with 1GHz frequency to obtain its area and power consumption. Then, we post-process power consumption from each component to obtain the system energy consumption. ANNA design parameter is set as follows: $N_{cu} = 96$, $N_{SCM} = 16$, and $N_u = 64$. ANNA can support both $k^* = 16$ and $k^* = 256$.

5.4.2 Performance Evaluation

Throughput Improvements. Figure 5.6 shows the throughput comparison of ANNA and several software ANNS implementations across different configurations and workloads. Each configuration is represented as a solid or a dotted line because the model recall changes across user-specified search parameter W, the number of clusters inspected. The higher W means higher recall at

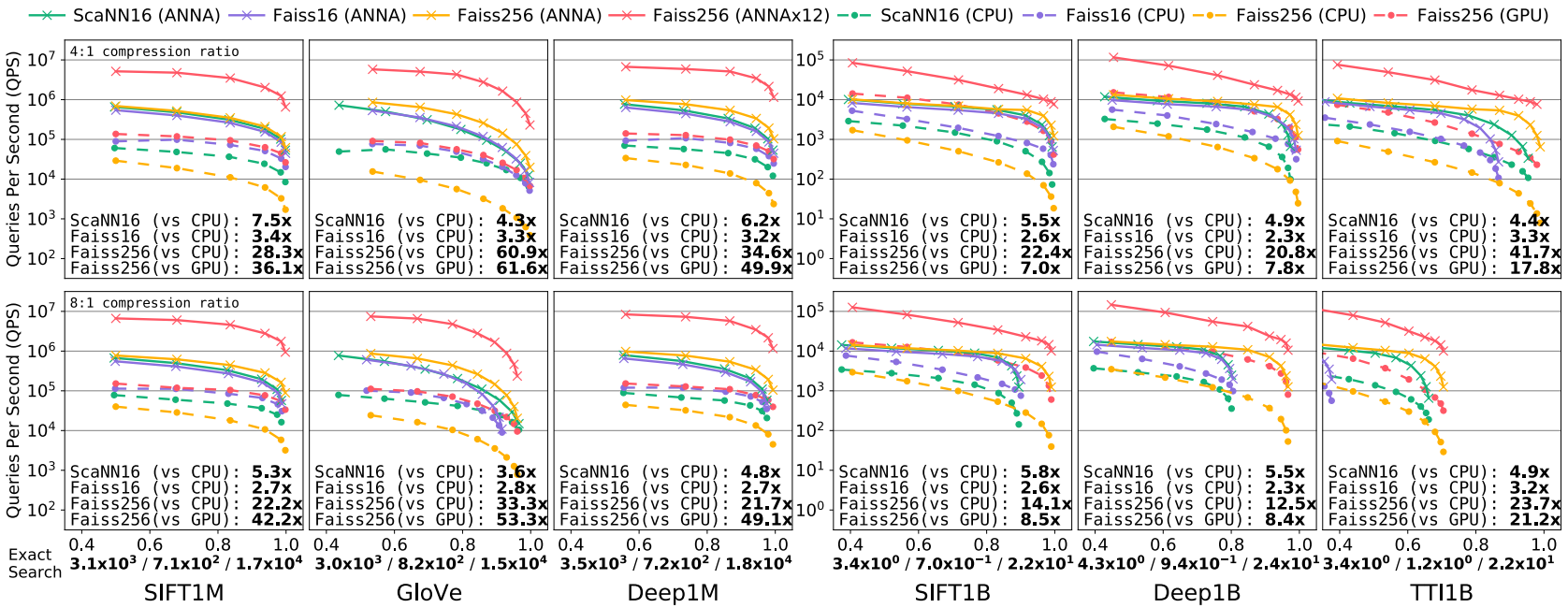


Figure 5.6: Throughput Comparison of ANNA and Various Software ANNS Implementations on CPU/GPU. X-axis is recall 100@1000 (portion of true top 100 items included in 1000 candidates obtained by ANNS algorithms). Y-axis represents the queries processed per second in a **log scale**. At the bottom of each plot, we compare each ANNA configuration with its corresponding software implementation and report the geomean speedup. The three numbers below plots represent the QPS of exhaustive, exact nearest neighbor search on ScaNN (CPU), Faiss (CPU), and Faiss (GPU), respectively.

the expense of lower throughput. The upper six plots in Figure 5.6 show the results for 4:1 compression ratio, meaning that the main memory needs to hold one-fourth of the original data size, which is $0.5N$ bytes. For $k^* = 256$ configurations which represent a single vector element as $log_2 k^* = 8$ bits, $M = D/2$ is utilized to compress the data by 4×. On the other hand, for $k^* = 16$ configurations, which represent a vector element as 4 bits, $M = D$ is utilized to compress the data by 4×. Similarly, the lower six plots in Figure 5.6 show the results for the 8:1 compression ratio. For that case, the smaller M (e.g., $D/4$) is utilized to further compress the data. In general, a higher compression rate trades off model recall for less memory usage.

The figure shows that ANNA achieves substantial speedup over CPU or GPU implementations across varying recalls. Among various CPU implementations, we find that Faiss256 (CPU) achieves lower performance than other CPU implementations. This is because Faiss16 (CPU) and ScaNN16 (CPU) utilize low-level code optimizations to pin 16-entry lookup tables on vector registers. On the other hand, when $k^* = 256$, 256-entry lookup tables do not fit on vector registers, and thus the Faiss256 (CPU) achieves much lower speedup. Between Faiss16 (CPU) and ScaNN16 (CPU), Faiss16 (CPU) achieves better performance since Faiss16 (CPU) implementation processes queries in a way that is similar to ANNA memory traffic optimization represented in Figure 5.3. ANNA implementation of Faiss16 outperforms Faiss16 (CPU) while consuming a much smaller area and energy (presented in the following section).

The major drawback of Faiss16 or ScaNN16 configuration is that the use of $k^* = 16$ sometimes fails to achieve high recall on challenging scenarios. For example, on Deep1B dataset (8:1 compression ratio), all $k^* = 16$ configurations cannot achieve recall beyond 0.9. Moreover, although not presented in the figure, those configurations fail to achieve 0.5 recall on 16:1 compression ratio scenarios for the same dataset. The same issue is observed in TTI1B dataset (8:1 compression ratio), but only with Faiss16 in this case. On the other hand, Faiss256 (CPU) can achieve substantially better maximum recall, but is much slower. Unlike those software implementations, Faiss256 (ANNA) can provide

a very high recall and throughput at the same time. Finally, Faiss256 (GPU) shows very promising performance in some cases. However, this is because the V100 GPU has 900 GB/s memory bandwidth. For the fair comparison, we compare Faiss256 (GPU) with the Faiss 256 (ANNA ×12), which utilizes twelve ANNA instances, each paired with a 75GB/s memory system. It is clear that ANNA ×12 achieves a substantially larger throughput than the V100 GPU.

Impact of ANNA Memory Traffic Optimization. We compare throughput of ANNA without optimization and ANNA with memory traffic optimization (Chapter 5.3). On average (across multiple datasets), ANNA with the memory traffic optimization achieves 5.1×/5.0×/6.9× throughput compared to ANNA without the optimization for ScaNN16/Faiss16/Faiss256 configurations on 4:1 compression rate cases, respectively. Similarly, the extra speedup from optimization is 3.9×/3.9×/4.6× on cases with the 8:1 compression ratio. Memory traffic optimization reduces the total amount of memory traffic and significantly improves performance in scenarios where the throughput is memory-bound. The additional performance gains are greater at the 4:1 compression ratio as it generates more memory traffic than 8:1 compression ratio, and thus it is more memory bandwidth-bound. It is also possible to apply the idea of ANNA memory traffic optimization to software schemes. As discussed before, Faiss16 (CPU) already utilizes a conceptually similar data reuse optimization technique to achieve better performance than ScaNN16 (CPU). However, such techniques are often very challenging to implement on CPUs since they lack a mechanism to pin the loaded data to their on-chip memory (i.e., cache).

Latency Improvements. Figure 5.7 compares the average latency of processing a single query between ANNA and the other workloads on configurations whose compression ratio is 4:1. The figure shows that ANNA enables very low-latency processing of similarity search queries. For example, ANNA achieves high recall (0.8+) at sub-ms latency in billion-scale datasets. On the other hand, the fastest CPU/GPU implementations achieve such recall

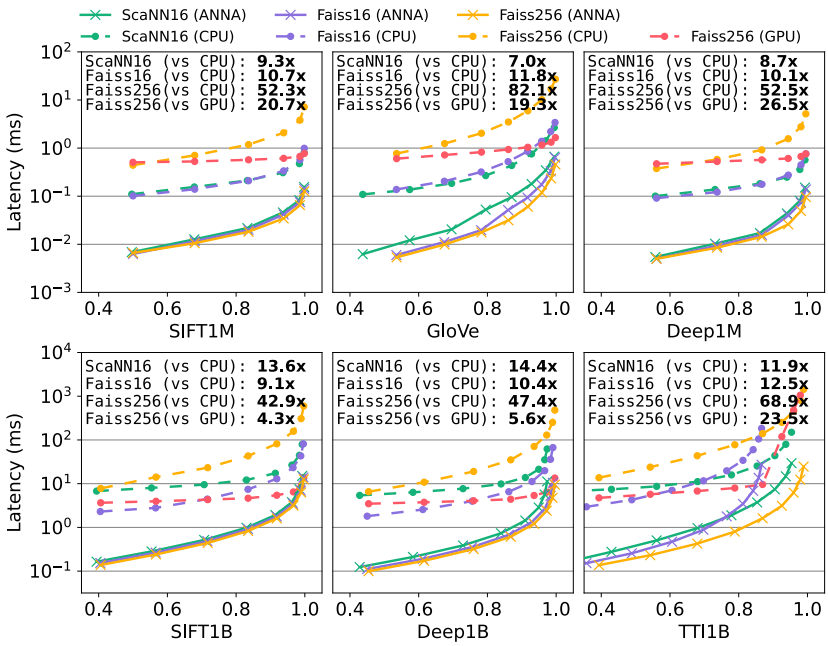


Figure 5.7: Latency Comparison of ANNA and Various Software ANNS Implementations. Y-axis represents the latency for a single query represented in a **log scale**. Lower is better. (4:1 compression ratio).

around 5-10ms latency. Overall, ANNA reduces an average query latency by 10.5×/10.7×/56.2×/13.5× over ScaNN16(CPU)/Faiss16(CPU)/Faiss256(CPU) /Faiss256(GPU), respectively. This demonstrates that ANNA exploits intra-query parallelism more effectively than the others.

5.4.3 Area/Energy Evaluation

Area. Table 5.1 reports the ANNA accelerator area usage. We find that a large portion of ANNA modules' area results from their SRAM structures. A single ANNA accelerator requires a 17.51mm^2 area at 40nm technology. In comparison, the evaluated CPU die size is 325.4mm^2 at 14nm technology [6] (effectively 151× larger), and the GPU die size is 815mm^2 at 12nm technology [142] (effectively 517× larger).

Power and Energy Consumption. Table 5.1 shows that a single ANNA accelerator consumes about 5.398W at its peak. In practice, not all modules are fully utilized at the same time, and thus the actual power usage (2-3W)

Table 5.1: Area and (Peak) Power of ANNA.

Module Name	Area (mm^2)	Peak Pwr(W)
Codebook/Cluster Processing Module	1.17	0.391
Encoded Vector Fetch Module	2.87	1.065
Similarity Computation Module (16×)	13.30	3.795
Memory Access Interface (MAI)	0.17	0.147
ANNA Accelerator	17.51	5.398
ANNA Accelerators (12×)	210.12	64.776

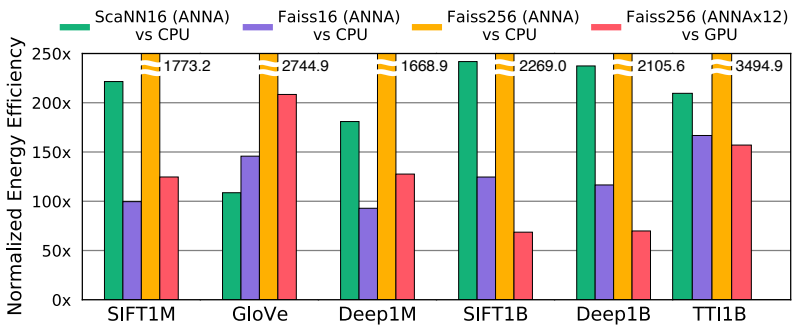


Figure 5.8: Normalized Energy Efficiency of ANNA over Corresponding CPU/GPU Implementations. (4:1 compression ratio, $W = 32$ configuration).

is lower than the peak. We also measure the GPU and CPU power consumption. Their power vary across datasets and configurations, but on average, the CPU utilizes 116W/139W power (ScaNN/Faiss), and the GPU utilizes 151.8W power during their operations. Figure 5.8 presents the energy efficiency comparison between ANNA accelerator and corresponding CPU/GPU implementations on a specific configuration. Combining the substantially lower power consumption and runtime reduction, ANNA achieves orders of magnitude energy efficiency improvements (97×+ across all configurations) over both CPU and GPU. The main sources of energy efficiency are i) the use of specialized computation modules and memory structures, ii) the use of dataflow pipeline, and iii) efficient data reuse.

5.5 Related Works

5.5.1 Hardware Acceleration of Nearest Neighbor Search

Abdelhadi et al. [12] design specialized FPGA implementation for PQ-based ANNS, exploiting the large on-chip memory on FPGA. This design achieves

high throughput on a million-scale dataset whose compressed vectors fit in on-chip buffer, but is not readily applicable to billion-scale datasets. Zhang et al. [208] also present FPGA-accelerated ANNS, which achieves 50K QPS for 0.94 recall (1@10) on SIFT1M dataset. However, the configuration used for this result cannot achieve high recall (e.g., over 90%) for most of the billion-scale datasets. Note that, for a similar recall on the same dataset, ANNA achieves around 256K QPS with a single accelerator instance, which uses multiple orders of magnitude smaller area, energy, and on-chip SRAM.

Both FPGA implementations lack ANNA's data reuse optimization presented in Chapter 5.3 and thus cannot efficiently utilize the limited off-chip memory bandwidth. Moreover, they use their own ANNS mechanism customized for their hardware. The metric performance of those custom mechanisms is not verified as rigorously as widely known software implementations like ScaNN and Faiss. In contrast, ANNA presents a hardware accelerator that is compatible with both ScaNN and Faiss while achieving much higher energy efficiency and overall speedups. Gemini APU [5] is a proprietary architecture that utilizes LSH-based ANNS. Their white paper states that it achieves 800 QPS for 0.92 recall (1@160) on Deep1B dataset, whereas ANNA achieves over 4096 QPS for a similar recall.

Note that a fair and extensive quantitative comparison of the ANNS hardware accelerators is quite difficult due to the throughput-recall tradeoff. For instance, a configuration with a relatively high compression ratio can often achieve high throughput, but the maximum recall that can be reached by such a configuration is relatively low. To make a meaningful comparison of hardware platforms, it is best to evaluate the performance of the same search configuration (e.g., M, k^*, $|W|$, etc.) across different hardware as in Chapter 5.4.

In addition, several prior works accelerate NNS in hardware, which are not directly comparable to our work. For example, Tigris [199] targets accelerating KD-tree ANNS. The presented implementation is specific to a point cloud registration task, which involves NNS for 3-dimensional data. KD-tree-based

ANNS is known to have a very limited search accuracy for high-dimensional data, and thus the presented hardware cannot be utilized for similarity search on billion-scale embeddings where each vector has over 100-dimensions. Also, Sun et al. [179] proposes a DSVS algorithm and FPGA-based design for 3-dimensional NNS for a similar task. As with KD-tree ANNS, the presented DSVS algorithm is not suitable for high-dimensional data. Danopoulos et al. [50] utilizes high-level-synthesis-based hardware to accelerate general matrix multiplication (gemm) in *kmeans* clustering algorithm, which is used for the NNS index construction that needs to be performed prior to the search process. Specifically, it targets to accelerate the process of obtaining centroid vectors (i.e., $c^{(0)}$, ... ,$c^{(|C|-1)}$), however ANNA targets the search process that happens after obtaining centroid vectors and codebooks is done. Thus, this work, as well as other gemm accelerators, are orthogonal to ANNA and are not applicable to the PQ-based ANNS query processing. Finally, several prior work [126, 131, 152] explore the hardware acceleration of *exact* nearest-neighbor search. Even with the hardware support, exhaustive searches are not a practical approach for billion-scale datasets because they require an excessive amount of computation, off-chip memory accesses, and energy consumption.

5.5.2 ANNS Techniques

ANNS on software is a well-studied topic with a wide range of related works. Graph-based ANNS [58, 87, 96, 129] exploit nearest neighbor graph, a graph structure whose each node is a vector that is connected to its nearby nodes. Although those algorithms achieve high performance on million-scale datasets [24], they are impractical for billion-scale searches as they require a large graph to be resident in memory. There exist other ANNS techniques such as ones that utilize tree-structures [61, 136] or locality-sensitive-hashing [17, 52, 170], but these algorithms exhibit lower performance than alternatives [24]. There exist several variants of the product quantization ANNS algorithms that aim to improve the codebook quality. Improving codebook quality is also critical for improving search performance because high-quality codebooks require the

inspection of a smaller number of database vectors while maintaining a high recall. For that purpose, ScaNN [77] utilizes a novel object function, DPQ [112] utilizes deep-learning-based training, and OPQ [67] applies rotation to the original database. ANNA can support all these variations since their computation pattern for the search remains the same. ANNA can also be readily extended to support other PQ variations such as AQ [19], which utilizes M identifiers with each associated with D-dimensional codeword. The ANNA implementation might need to be slightly extended to support some PQ-based similarity search algorithms of the future, but we expect the core concepts and components of the ANNA design remain relevant since almost all PQ-based similarity search algorithms operate in a similar manner: load encoded data, exploit lookup tables to simplify the similarity computation with the encoded data, and select top k candidates to return.

Chapter 6

Conclusion

6.1 Summary

Embedding is one of the most popular data types to represent complicated information. As the size of the deep neural networks and the dataset are growing rapidly, emerging AI applications spend a significant portion of runtime on embedding operations. Thus, it is crucial to optimize these embedding operations to meet the strict constraints on throughput and energy consumption in practical use cases. This dissertation introduces three primary challenges these emerging AI applications are facing and provides solutions for them.

The first challenge is an increase in memory bandwidth consumption. Embedding reduction operation is a representative example of operations facing this challenge. For embedding reduction operation, we introduce MERCI, a novel memoization framework that exploits co-appearing structure among features in a real-world dataset.

The second challenge is an increase in computation. Self-attention operation for transformer-based neural networks is a well-known computationally heavy operation. We introduce ELSA to optimize self-attention operation, which is a software-hardware codesign work that proposes a novel approximate self-attention algorithm that efficiently identifies relatively less important computations and hardware accelerator that supports skipping such computations to maximize the benefit of approximation.

The third challenge is the inefficiency of the hardware. Compression-based approximate nearest neighbor search is a representative example that suffers performance degradation due to hardware inefficiencies. Thus, we introduce

ANNA, which is a specialized architecture designed carefully to overcome the inefficiencies in commodity hardwares (e.g., CPUs and GPUs).

6.2 Discussion

Sustainability. As AI applications continue to evolve at a rapid pace, with model architectures and algorithms advancing in tandem, it is crucial to ensure that research works in this field remain sustainable and adaptable to these changes. Here, I will discuss the ability of my research to remain sustainable in the face of rapid changes in the field of AI, and also discuss the challenges that must be overcome in order to adapt to these changes effectively.

MERCI name is a software optimization technique that is easily deployable on the existing commodity hardwares. The embedding reduction operation that MERCI targets is an extremely simple operation that involves randomly sparse memory access with indices, and it is a widely used operation that I expect will remain popular in the foreseeable future, making MERCI a useful optimization method for many applications that perform this operation. However, it should be noted that MERCI cannot be used when the contents of the embedding tables are kept updated, such as during training.

The suitability of hardware accelerators, such as ELSA and ANNA, for future AI algorithms may be questioned. For both, I focused on designing modular and reusable hardware that can be easily combined and configured with deep understanding of the data and control flow of the target algorithms as well as providing scalable design. In particular, ELSA is compatible with recent transformer variants that address the computational demand of self-attention operations, such as Longformer [23], BigBird [202], and Open AI Block-sparse [75]. ANNA also supports several variants of product quantization algorithms, such as DPQ [112] and OPQ [67], and need only a slight modification to support other variants, such as AQ [19].

Despite the adaptability of ELSA and ANNA to many recent algorithm variants, there are still some algorithms that cannot be supported by these hardware designs, and there is a possibility that future algorithms may not

be compatible either. To address this challenge, we may need to extensively modify the existing hardware designs or even design new customized hardware. The key is to identify the new performance bottleneck of the new algorithm and carefully address them while ensuring efficient support for existing algorithms. Even though, thanks to the flexible and modular design of ELSA and ANNA, I believe that the core concepts and components of my works will remain relevant and impactful in the face of rapid change, while also helping to shape the future of this actively evolving field.

In particular, in the edge device environment with strict constraints on latency and energy consumption, extremely lightweight models have been proposed as a solution, such as MobileNet [90] and EfficientNet [185]. These models are already highly optimized, making it difficult to optimize them further and achieve significant performance improvements since there is limited room for optimization. Although these lightweight models could still benefit from the acceleration provided by my works, the performance improvement may not be as effective as standard-sized models.

To achieve greater acceleration and energy efficiency improvements, it is necessary to make modifications that leverage the unique characteristics of lightweight models. For instance, there is a research area that focuses on providing quantization method for (ultra-)low precision self-attention [125, 167, 201], to enable efficient model processing with minimal loss of accuracy. Also, there is plenty of dimension reduction works [132, 161, 203] that explores a way of reducing the dimension effectively while minimizing the information loss from the original dimension. When combined with product quantization ANNS, it leads to a substantial reduction in D and potentially smaller m.

For such optimized cases, the current hardware accelerator designs may be overkill in terms of processing power or memory bandwidth, which leads to inefficient energy and resource costs. Consequently, there are new opportunities to further optimize these lightweight models through scaling down or simplifying existing hardware accelerator designs. For (ultra-)low precision self-attention, the bitwidth of hardware units in ELSA could be narrowed

down according to the reduced precision. This can reduce the area required for attention computation modules, creating more space for placing additional attention computation modules. As the attention computation becomes faster, the number of hash functions may need to be reduced in order to prevent ELSA preprocessing phase from becoming a bottleneck. This will also result in smaller SRAM sizes (e.g., key hash/norm memory) and faster hamming distance computation. Also, ANNA should be scaled down by reducing the SRAM size of lookup tables, codebook and encoded vector buffer and the size of the buffer in memory reader accordingly as m gets smaller. Also, the number of compute units in CPM and adders in the reduction unit of SCM could be decreased until it does not cause a performance bottleneck in ANNA execution pipeline. By simplifying the hardware accelerators in this manner, it might bring imbalance to the current execution pipeline and create new performance bottlenecks. Thus, we also need to come up with a new configuration to balance the execution pipeline. With modified hardware designs, it can lead to enhanced performance in latency and energy-constrained environments. Moreover, the saved area can be utilized for various purposes to achieve further performance improvements specifically tailored to lightweight models.

Productization. Here, I discuss the advantages and obstacles in bringing my works to the market. The benefits of productizing my works are the notable enhancements in throughput, latency, and energy efficiency. As AI services in data centers have strict demands for these performance metrics, integrating my work into products could shorten the service time, reduce energy costs, and promote environmentally-friendly data centers.

However, there exist several challenges in productizing my work. The primary challenge in productizing ELSA and ANNA is the high cost of manufacturing hardware and scaling up the designs to the datacenter level. For ELSA, accuracy loss is another significant challenge, as it is an accuracy and computation trade-off work by approximating the self-attention operation. As the accuracy loss can only be determined after completing the model run for a given hyperparameter p (which is a user-defined value between 0 and 1 that

sets the degree of approximation), there is no guarantee for the accuracy lower bound. To make ELSA more accessible and adaptable to a wider range of use cases, a solution that provides an accuracy knob to transparently control the model's accuracy would be beneficial. While it is challenging to provide a precise accuracy knob that can adjust the accuracy at fine-grained granularity, there is a simple way to provide rough guidelines for setting hyperparameter p by leveraging the threshold training process of ELSA. ELSA performs this process to obtain layer-specific thresholds that reflect the characteristics of each self-attention layer with a training dataset. At the same time, it outputs the model's train accuracy for a given hyperparameter p since it performs inference with the training dataset to obtain thresholds. Normally, this process is performed only once for a user-specified p. However, we can modify it to repeat this process for multiple p values and provide the train accuracy result to users. As a result, users are now allowed to observe how the train accuracy changes according to p. This approach provides users with an approximate understanding of the model's sensitivity to the hyperparameter p and high-level guidelines on setting hyperparameter p to obtain desired test accuracy.

Additionally, there are recent works [172, 173, 183, 186] proposing techniques that predict the impact of approximation on the overall model accuracy based on the information about the model accuracy when approximation is applied partially to the model. Another promising future work is modifying these prediction techniques to be tailored to ELSA by leveraging information obtained from the threshold training process. For instance, BERT (large) model we used in ELSA evaluation has 24 layers with 16 multi-head self-attention, and the threshold training process generates 384 (24×16) (sub-)layer-specific thresholds. These thresholds could be potentially used for analyzing the sensitivity of each layer to the approximation. The layers with higher thresholds tend to have very few keys having high attention scores, leading to more approximation opportunities since the majority of keys with low attention scores will have near zero or zero values after softmax normalization. With this modified prediction technique, ELSA provides an accuracy knob by providing model

accuracy predictions for a given hyperparameter p. However, providing an accurate and transparent accuracy knob for ELSA remains an open research question that requires further exploration.

For ANNA, another challenge in productizing and deploying ANNA in a data center is creating an automated distributed system capable of handling extremely large-scale datasets. This system should include a process for dealing with datasets that are too large to fit into the memory of a single host, such as CPU DRAM. The process involves splitting the dataset into multiple shards that can fit into the host memory, distributing the shards to multiple nodes, and having each node process a single shard. Each node then finds the top-k results from its shard, and when all nodes have finished processing, the selected top-k vectors from each node are aggregated into one node. Finally, the final result is computed by selecting the top-k among them.

The main obstacle to productizing MERCI is to come up with an efficient way of incorporating newly added features into the existing memoization table. Currently, if new features are introduced, the entire offline processing phase needs to be redone. This can be very costly for large-scale datasets used in recommendation services by big companies. Therefore, a lightweight approach is needed to learn the patterns of new features and update the existing clusters incrementally without having to perform the entire offline processing phase again. Note that MERCI does not require redoing the entire offline processing phase in case where embedding vectors for existing features are updated.

6.3 Future Work

Accelerating Pretraining-based Language Models with Software/Hardware Co-design. Following up on ELSA, I also plan to accelerate emerging large-scale pretraining-based language models (PLMs) such as OpenAI GPT-4 with hardware/software co-design. For this purpose, I will investigate how to i) downsize the fine-tuned pretraining-based natural language processing models for the specific target hardware using teacher-student method, and ii) design the hardware accelerator that can be flexibly reconfigured for efficient

end-to-end acceleration of such models. Specifically, I would like to address the limitations of the existing distillation techniques. Many existing proposals often do not have a clear understanding of the hardware and rather focus on abstract objectives (e.g., number of parameters or operations). And for this reason, they often fail to demonstrate the end-to-end performance or energy efficiency benefits. In contrast, I plan to expose the underlying hardware architecture to the model distillation technique and also augment the existing hardware with an extension to achieve maximum benefits.

Efficient Embedding Reduction on Emerging Memory System. Following up on MERCI, I am planning to optimize embedding reduction operations through memoization on the emerging memory system. Specifically, MERCI exploits novel clustering algorithm to identify sets of co-occurring values (e.g., co-appearing movie genres for a single movie), and store the partial reduction result (e.g., partial sum) for those sets in CPU memory. However, CPU memory is a costly device thus it is not very cost-efficient to store embeddings and memoized partial reduction results there for large-scale recommendation models. Thus, I am planning to explore the potential of storing the partial reduction result in the emerging storage-class-memory (SCM) such as Intel Optane DC which provides extremely high capacity at the expense of the lower bandwidth. Considering that the memoization scheme trades off capacity for fewer memory accesses, SCM is a perfect device for our purpose. The main challenge that needs to be addressed here is i) how we can minimize the potential performance degradation from the use of slower, lower-bandwidth SCM memory, and ii) how we can utilize both the DRAM and the SCM efficiently (e.g., use DRAM to store more frequently utilized embedding reduction results) to accelerate the embedding reduction.

Augmenting MERCI with Bundle Recommendation. There is a research domain called *bundle recommendation* [3, 18, 181] that explores the concept of recommending bundles of items that are often purchased together. For example, when recommending items to the users who are making a hamburger, a bundle of meat, bread, cheese, lettuce, and tomato would be sug-

gested since there is a high probability that these users would purchase these items together. This strategy of recommending items at a bundle level is widely used in e-commerce marketing. Building on the work of MERCI, I plan to enhance MERCI by incorporating bundle recommendation. However, there are two significant challenges that need to be addressed in this approach.

First, I conducted an experiment to investigate the relationship between clusters of items generated by MERCI and bundles of items generated by bundle recommendation [3]. To achieve this, I calculated the *overlapping_ ratio*, defined as *{# of bundles with two or more items overlapping with bundles}÷{# of bundles}*, for the Clothing Shoes & Jewelry and Electronics datasets from the Amazon Review dataset, which were used in the evaluation of MERCI. The *overlapping_ ratio* for both datasets was less than 10%, indicating that the chance of items in a single bundle belonging to the same cluster is low. To identify the reason behind this result, I checked a bundle and a cluster that contained bracelet A. The bundle that included bracelet A also had three other types of bracelets, while the cluster that contained bracelet A had a pair of sunglasses and a fairy costume. This indicates that, in real-world scenarios, people tend to purchase other items along with a bracelet instead of buying four different types of bracelets at once. Thus, the primary challenge is to develop a method to adjust bundles to include items that are frequently purchased together in the embedding reduction operation.

The second challenge pertains to the fact that bundles allow an item to appear more than once across multiple bundles. This poses difficulties in two aspects: i) determining whether the relevant partial sums for a given query are in the memoization table, and ii) locating where they are stored. In contrast, MERCI allows an item to appear in only one cluster, which is an essential design choice that enables efficient retrieval of memoized results with minimal additional memory accesses, maximizing the benefits of memoization. Therefore, the challenge in this case is to devise a method to choose bundles that are not only mutually exclusive but also contain items that frequently occur together during the embedding reduction operation.

Bibliography

[1] Benchmarks for Billion-Scale Similarity Search. https://research.yandex.com/datasets/biganns.

[2] Bing Vector Search. https://www.microsoft.com/en-us/ai/ai-lab-vector-search.

[3] Context2bundle: Diversified personalized bundle recommendation. https://github.com/wubinzzu/Context2Bundle.

[4] Faiss. https://github.com/facebookresearch/faiss.

[5] Gemini APU: Enabling High Performance Billion-Scale Similarity Search. https://www.gsitechnology.com/sites/default/files/Whitepapers/GSIT-APU-Enabling-High-Performance-Billion-Scale-Similarity-Search-WP.pdf.

[6] LCC SoC - Skylake (server) Microarchitectures Intel. https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)#LCC_SoC.

[7] Neighborhood Graph and Tree for Indexing High-dimensional Data. https://github.com/yahoojapan/NGT.

[8] Nvidia Merlin. https://developer.nvidia.com/nvidia-merlin.

[9] NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[10] NVIDIA Visual Profiler. https://developer.nvidia.com/nvidia-visual-profiler.

[11] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[12] Ameer MS Abdelhadi, Christos-Savvas Bouganis, and George A Constantinides. Accelerated approximate nearest neighbors search through hierarchical product quantization. In *Proceedings of the International Conference on Field-Programmable Technology*, ICFPT, 2019.

[13] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the International Conference on Very Large Data Bases*, VLDB, 1994.

[14] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. SnaPEA: Predictive early activation for reducing computation in deep convolutional neural networks. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, 2018.

[15] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA, 2016.

[16] Amazon. Amazon EC2 M5 Instances. `https://aws.amazon.com/ec2/instance-types/m5/`.

[17] FALCONN - FAst Lookups of Cosine and Other Nearest Neighbors. `https://github.com/FALCONN-LIB/FALCONN`.

[18] Tzoof Avny Brosh, Amit Livne, Oren Sar Shalom, Bracha Shapira, and Mark Last. Bruce: Bundle recommendation using contextualized item embeddings. In *Proceedings of the 16th ACM Conference on Recommender Systems*, RecSys, 2022.

[19] Artem Babenko and Victor Lempitsky. Additive quantization for extreme vector compression. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2014.

[20] Artem Babenko and Victor Lempitsky. Efficient indexing of billion-scale datasets of deep descriptors. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

[21] Björn Barz and Joachim Denzler. Hierarchy-based image embeddings for semantic image retrieval. In *Proceedings of the IEEE Winter Conference on Applications of Computer Vision*, WACV, 2019.

[22] Irwan Bello, Barret Zoph, Ashish Vaswani, Jonathon Shlens, and Quoc V. Le. Attention augmented convolutional networks. In *IEEE/CVF International Conference on Computer Vision*, ICCV, 2019.

[23] Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv:2004.05150*, 2020.

[24] Benchmarking nearest neighbors. `https://github.com/erikbern/ann-benchmarks`.

[25] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *In proceedings of the International Conference on Music Information Retrieval*, ISMIR, 2011.

[26] Ranjita Bhagwan and Bill Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proceedings of the IEEE Conference on Computer Communications*, INFOCOM, 2000.

[27] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv:2005.14165*, 2020.

[28] I. Brumar, M. Casas, M. Moreto, M. Valero, and G. S. Sohi. Atm: Approximate task memoization in the runtime system. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium*, IPDPS, 2017.

[29] Miguel Campo, Cheng-Kang Hsieh, Matt Nickens, J. J. Espinoza, Abhinav Taliyan, Julie Rieger, Jean Ho, and Bettina Sherick. Competitive analysis system for theatrical movie releases based on movie trailer deep video representation. *arXiv:1807.04465*, abs/1807.04465, 2018.

[30] Ümit V. Çatalyürek and Cevdet Aykanat. Patoh (partitioning tool for hypergraphs). In *Encyclopedia of Parallel Computing*. 2011.

[31] Cerebras systems: Achieving industry best ai performance through a systems approach. `https://secureservercdn.net/198.12.145.239/a7b.fcb.myftpupload.com/wp-content/uploads/2020/03/Cerebras-Systems-Overview.pdf?time=1584807908`.

[32] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, STOC, 2002.

[33] Qiwei Chen, Huan Zhao, Wei Li, Pipei Huang, and Wenwu Ou. Behavior sequence transformer for e-commerce recommendation in alibaba. In

Proceedings of the International Workshop on Deep Learning Practice for High-Dimensional Sparse Data with KDD, DLP-KDD, 2019.

[34] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.

[35] Wei Chen, Xiao Ma, Jiangfeng Zeng, Yaoqing Duan, and Grace Zhong. Hierarchical quantization for billion-scale similarity retrieval on gpus. *Computers & Electrical Engineering*, 2021.

[36] Y. Chen, T. Krishna, J. S. Emer, and V. Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 2017.

[37] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DaDianNao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2014.

[38] Sptag: A library for fast approximate nearest neighbor search. `https://github.com/microsoft/SPTAG`, 2018.

[39] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & deep learning for recommender systems. In *Proceedings of Workshop on Deep Learning for Recommender Systems*, DLRS, 2016.

[40] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv:1904.10509*, 2019.

[41] Chisel. https://chisel.eecs.berkeley.edu.

[42] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers. *arXiv:2009.14794*, 2020.

[43] Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, 2019.

[44] D. A. Conners and W. W. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *Proceedings of the Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO, 1999.

[45] Marcella Cornia, Matteo Stefanini, Lorenzo Baraldi, and Rita Cucchiara. Meshed-memory transformer for image captioning. In *The IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR, 2020.

[46] Intel Corporation. Intel vtune profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[47] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the ACM Conference on Recommender Systems*, RecSys, 2016.

[48] W. Bruce Croft, Donald Metzler, and Trevor Strohman. *Search engines: information retrieval in practice.* Addison-Wesley, 2010.

[49] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, ACL, 2019.

[50] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. Fpga acceleration of approximate knn indexing on high- dimensional vectors. In *Proceedings of 14th International Symposium on Reconfigurable Communication-centric Systems-on-Chip*, ReCoSoC, 2019.

[51] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

[52] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Symposium on Computational Geometry*, SoCG, 2004.

[53] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory power estimation and capping. In *ACM/IEEE International Symposium on Low-Power Electronics and Design*, ISLPED, 2010.

[54] J. Dean, D. Patterson, and C. Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 2018.

[55] Alberto Delmas Lascorz, Patrick Judd, Dylan Malone Stuart, Zissis Poulos, Mostafa Mahmoud, Sayeh Sharify, Milos Nikolic, Kevin Siu, and Andreas Moshovos. Bit-Tactical: A software/hardware approach to exploiting value and bit sparsity in neural networks. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[56] Karen D Devine, Erik G Boman, Robert T Heaphy, Rob H Bisseling, and Umit V Catalyurek. Parallel hypergraph partitioning for scientific computing. In *In proceedings of IEEE International Parallel and Distributed Processing Symposium*, IPDPS, 2006.

[57] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language un-

derstanding. In *In proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, NAACL, 2019.

[58] KGraph: A Library for Approximate Nearest Neighbor Search. `https://github.com/aaalgo/kgraph`.

[59] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting vision processing closer to the sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA, 2015.

[60] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim M. Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In *Proceedings of Machine Learning and Systems*, MLSys, 2019.

[61] Annoy. `https://github.com/spotify/annoy`.

[62] Facebook. Caffe2. `https://caffe2.ai`, 2016.

[63] Yufei Feng, Fuyu Lv, Weichen Shen, Menghan Wang, Fei Sun, Yu Zhu, and Keping Yang. Deep session interest network for click-through rate prediction. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, IJCAI, 2019.

[64] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the 45th International Symposium on Computer Architecture*, ISCA, 2018.

[65] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast approximate

nearest neighbor search with the navigating spreading-out graph. VLDB Endowment, 2019.

[66] Adi Fuchs and David Wentzlaff. Scaling datacenter accelerators with compute-reuse architectures. In *Proceedings of the Annual International Symposium on Computer Architecture*, ISCA, 2018.

[67] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2013.

[68] Y. Gong, S. Kumar, H. A. Rowley, and S. Lazebnik. Learning binary codes for high-dimensional data using bilinear projections. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2013.

[69] Cloud TPU system architecture. `https://cloud.google.com/tpu/docs/system-architecture`.

[70] ALBERT official implementation. `https://github.com/google-research/ALBERT`.

[71] Gram–schmidt process. `https://en.wikipedia.org/wiki/Gram%E2%80%93Schmidt_process`.

[72] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv:1410.5401*, 2014.

[73] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.

[74] Scott Gray, Alec Radford, and Diederik P. Kingma. Gpu kernels for block-sparse weights. `https://cdn.openai.com/blocksparse/blocksparsepaper.pdf`.

[75] Scott Gray, Alec Radford, and Diederik P Kingma. Gpu kernels for block-sparse weights. *arXiv:1711.09224*, 2017.

[76] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A factorization-machine based neural network for CTR prediction. In *Proceedings of International Joint Conference on Artificial Intelligence*, IJCAI, 2017.

[77] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *Proceedings of the International Conference on Machine Learning*, ICML, 2020.

[78] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning*, ICML, 2015.

[79] U. Gupta, C. Wu, X. Wang, M. Naumov, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, M. Hempstead, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. The architectural implications of facebook's DNN-based personalized recommendation. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, HPCA, 2020.

[80] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S. Lee, David Brooks, and Carole-Jean Wu. DeepRecSys: A system for optimizing end-to-end at-scale neural recommendation inference. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA, 2020.

[81] Goya hl-10x datasheet. https://habana.ai/wp-content/uploads/2019/06/Goya-Datasheet-HL-10x.pdf.

[82] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H. Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W. Lee, and Deog-Kyoon Jeong. A^3: Accelerating attention mechanisms in neural networks with approximation. In *Proceedings of the 26th IEEE International Symposium on High Performance Computer Architecture*, HPCA, 2020.

[83] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 2000.

[84] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA, 2017.

[85] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient inference Engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA, 2016.

[86] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Transactions on Interactive Intelligent Systems*, 2015.

[87] Ben Harwood and Tom Drummond. Fanng: Fast approximate nearest neighbour graphs. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2016.

[88] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noord-

huis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *IEEE International Symposium on High Performance Computer Architecture*, HPCA, 2018.

[89] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *Proceedings of the International Conference on World Wide Web*, WWW, 2016.

[90] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv:1704.04861*, 2017.

[91] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *Journal of Machine Learning Research*, 2017.

[92] R. Hwang, T. Kim, Y. Kwon, and M. Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA, 2020.

[93] Ville Hyvönen, Teemu Pitkänen, Sotiris Tasoulis, Elias Jääsaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. Fast k-nn search. *arXiv:1509.06957*, 2015.

[94] Intel Xeon gold 6128 processor. `https://ark.intel.com/products/120482/Intel-Xeon-Gold-6128-Processor-19-25M-Cache-3-40-GHz-`, 2017.

[95] Matei Istoan and Bogdan Pasca. Fixed-point implementations of the

reciprocal, square root and reciprocal square root functions. In *HAL Open Archive*, 2015.

[96] Masajiro Iwasaki. Pruned bi-directed k-nearest neighbor graph for proximity search. In *Proceedings of the International Conference on Similarity Search and Applications*, SISAP, 2016.

[97] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *The IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2018.

[98] Shubham Jain, Swagath Venkataramani, Vijayalakshmi Srinivasan, Jungwook Choi, Pierce Chuang, and Leland Chang. Compensated-dnn: Energy efficient low-precision deep neural networks by compensating quantization errors. In *Proceedings of the 55th Annual Design Automation Conference*, DAC, 2018.

[99] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. MnnFast: A fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA, 2019.

[100] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2010.

[101] Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP, 2011.

[102] Jianqiu Ji, Jianmin Li, Shuicheng Yan, Bo Zhang, and Qi Tian. Super-bit locality-sensitive hashing. In *Advances in Neural Information Processing Systems*, NIPS, 2012.

[103] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[104] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, 2017.

[105] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2016.

[106] Igor Kabiljo, Brian Karrer, Mayank Pundir, Sergey Pupyrev, Alon Shalita, Yaroslav Akhremtsev, and Alessandro Presta. Social hash partitioner: A scalable distributed hypergraph partitioner. *In proceedings of the VLDB Endowment*, 2017.

[107] Wang-Cheng Kang and Julian J. McAuley. Self-attentive sequential recommendation. In *Proceedings of the IEEE International Conference on Data Mining*, ICDM, 2018.

[108] Supriya Kapur, Asit K. Mishra, and Debbie Marr. Low precision rnns: Quantizing rnns without losing accuracy. *arXiv:1710.07706*, 2017.

[109] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel hypergraph partitioning: applications in vlsi domain. *IEEE Transactions on Very Large Scale Integration Systems*, 1999.

[110] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim M. Hazelwood, Bill Jia, Hsien-Hsin S. Lee, Meng Li, Bert Maher, Dheevatsa Mudigere, Maxim Naumov, Martin Schatz, Mikhail Smelyanskiy, Xiaodong Wang, Brandon Reagen, Carole-Jean Wu, Mark Hempstead, and Xuan Zhang. RecNMP: Accelerating personalized recommendation with near-memory processing. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture*, ISCA, 2020.

[111] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations*, ICLR, 2020.

[112] Benjamin Klein and Lior Wolf. End-to-end supervised product quantization for image search and retrieval. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR, 2019.

[113] H.T. Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2019.

[114] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger.

From word embeddings to document distances. In *Proceedings of the International Conference on Machine Learning*, ICML, 2015.

[115] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. TensorDIMM: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[116] Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. RACE: Large-scale ReAding comprehension dataset from examinations. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2017.

[117] Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. *Advances in Neural Information Processing Systems*, 2019.

[118] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. ALBERT: A lite bert for self-supervised learning of language representations. In *International Conference on Learning Representations*, ICLR, 2020.

[119] Dongwoo Lee, Sungbum Kang, and Kiyoung Choi. ComPEND: Computation pruning through early negative detection for ReLU in a deep neural network accelerator. In *Proceedings of the International Conference on Supercomputing*, ICS, 2018.

[120] Sen Li, Fuyu Lv, Taiwei Jin, Guli Lin, Keping Yang, Xiaoyi Zeng, Xiao-Ming Wu, and Qianli Ma. Embedding-based product retrieval in taobao search. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD, 2021.

[121] Yang Liu and Mirella Lapata. Text summarization with pretrained encoders. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2019.

[122] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized BERT pretraining approach. *arXiv:1907.11692*, 2019.

[123] Z. Liu, A. Yazdanbakhsh, T. Park, H. Esmaeilzadeh, and N. S. Kim. SiMul: An algorithm-driven approximate multiplier design for machine learning. *IEEE Micro*, 2018.

[124] Zhenhong Liu, Amir Yazdanbakhsh, Dong Kai Wang, Hadi Esmaeilzadeh, and Nam Sung Kim. Axmemo: Hardware-compiler co-design for approximate code memoization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA, 2019.

[125] Zhenhua Liu, Yunhe Wang, Kai Han, Wei Zhang, Siwei Ma, and Wen Gao. Post-training quantization for vision transformer. *Advances in Neural Information Processing Systems*, 2021.

[126] Alec Lu, Zhenman Fang, Nazanin Farahpour, and Lesley Shannon. Chip-knn: A configurable and high-performance k-nearest neighbors accelerator on cloud fpgas. In *International Conference on Field-Programmable Technology*, ICFPT, 2020.

[127] Michael Lui, Yavuz Yetim, Özgür Özkan, Zhuoran Zhao, Shin-Yeh Tsai, Carole-Jean Wu, and Mark Hempstead. Understanding capacity-driven scale-out neural recommendation inference. *arXiv:2011.02084*, 2020.

[128] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics*, ACL, 2011.

[129] Yu A Malkov and Dmitry A Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

[130] Yusuke Matsui, Yusuke Uchida, Hervé Jégou, and Shin'ichi Satoh. A survey of product quantization. *ITE Transactions on Media Technology and Applications*, 2018.

[131] Takazumi Matsumoto and Man Lung Yiu. Accelerating exact similarity search on cpu-gpu systems. In *IEEE International Conference on Data Mining*, 2015.

[132] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv:1802.03426*, 2020.

[133] Donald Michie. "memo" functions and machine learning. *Nature*, 1968.

[134] Antoine Miech, Jean-Baptiste Alayrac, Lucas Smaira, Ivan Laptev, Josef Sivic, and Andrew Zisserman. End-to-end learning of visual representations from uncurated instructional videos. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, CVPR, 2020.

[135] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Proceedings of the International Conference on Neural Information Processing Systems*, NIPS, 2013.

[136] Marius Muja and David G Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2014.

[137] S. Muthusundari, Gurram V. B. S. Yaswanth, Jaludu R. Kumar, and Chamarthi Nagarjuna. Quantization of product using collaborative filtering based on cluster. *Annals of the Romanian Society for Cell Biology*, 2021.

[138] Maxim Naumov. On the dimensionality of embeddings for sparse features and data. *arXiv:1901.02103*, 2019.

[139] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *arXiv:1906.00091*, 2019.

[140] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv:1906.00091*, 2019.

[141] NVIDIA TITAN V. `https://www.nvidia.com/en-us/titan/titan-v/`, 2018.

[142] NVIDIA V100 Tensor Core GPU. `https://images.nvidia.com/content/technologies/volta/pdf/volta-v100-datasheet-update-us-1165301-r5.pdf`, 2020.

[143] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. fairseq: A fast, extensible toolkit for sequence modeling. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, NAACL, 2019.

[144] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA, 2017.

[145] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA, 2018.

[146] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash-based algorithm for mining association rules. In *In proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD, 1995.

[147] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Shanker Khudia, James Law, Parth Malani, Andrey Malevich, Nadathur Satish, Juan Pino, Martin Schatz, Alexander Sidorov, Viswanath Sivakumar, Andrew Tulloch, Xiaodong Wang, Yiming Wu, Hector Yuen, Utku Diril, Dmytro Dzhulgakov, Kim M. Hazelwood, Bill Jia, Yangqing Jia, Lin Qiao, Vijay Rao, Nadav Rotem, Sungjoo Yoo, and Mikhail Smelyanskiy. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv:1811.09886*, 2018.

[148] S. Park, J. Jang, S. Kim, and S. Yoon. Energy-efficient inference accelerator for memory-augmented neural networks on an fpga. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, DATE, 2019.

[149] Niki Parmar, Prajit Ramachandran, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. Stand-alone self-attention in vision models. In *Advances in Neural Information Processing Systems*, NeurIPS, 2019.

[150] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit

Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*. 2019.

[151] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2014.

[152] Yuliang Pu, Jun Peng, Letian Huang, and John Chen. An efficient knn algorithm implemented on fpga based heterogeneous computing system using opencl. In *IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015.

[153] Torchscript. `https://pytorch.org/docs/stable/jit.html`, 2020.

[154] Jiezhong Qiu, Hao Ma, Omer Levy, Scott Wen tau Yih, Sinong Wang, and Jie Tang. Blockwise self-attention for long document understanding. *arXiv:1911.02972*, 2019.

[155] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. `https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`, 2019.

[156] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, ICLR, 2020.

[157] A. Raha, S. Venkataramani, V. Raghunathan, and A. Raghunathan. Energy-efficient reduce-and-rank using input-adaptive approximations. *IEEE Transactions on Very Large Scale Integration Systems*, 2017.

[158] Arnab Raha and Vijay Raghunathan. qLUT: Input-aware quantized

table lookup for energy-efficient approximate accelerators. *ACM Transactions on Embedded Computing Systems*, 2017.

[159] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2016.

[160] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA, 2016.

[161] G Thippa Reddy, M Praveen Kumar Reddy, Kuruva Lakshmanna, Rajesh Kaluri, Dharmendra Singh Rajput, Gautam Srivastava, and Thar Baker. Analysis of dimensionality reduction techniques on big data. *Ieee Access*, 2020.

[162] Stephen E. Richardson. Caching function results: Faster arithmetic by avoiding unnecessary computation. Technical report, 1992.

[163] Corby Rosset. Turing-NLG: A 17-billion-parameter language model by Microsoft. `https://www.microsoft.com/en-us/research/blog/turing-nlg-a-17-billion-parameter-language-model-by-microsoft/`.

[164] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[165] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. K-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of Workshop on Algorithm Engineering and Experiments*, ALENEX, 2016.

[166] Ali Sharif Razavian, Hossein Azizpour, Josephine Sullivan, and Stefan Carlsson. Cnn features off-the-shelf: an astounding baseline for recogni-

tion. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2014.

[167] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2020.

[168] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv:1909.08053*, 2019.

[169] Gil Shomron, Ron Banner, Moran Shkolnik, and Uri Weiser. Thanks for nothing: Predicting zero-valued activations with lightweight convolutional neural networks. *arXiv:1909.07636*, 2019.

[170] Anshumali Shrivastava and Ping Li. Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS). In *International Conference on Neural Information Processing Systems*, NIPS, 2014.

[171] Weiping Song, Chence Shi, Zhiping Xiao, Zhijian Duan, Yewen Xu, Ming Zhang, and Jian Tang. Autoint: Automatic feature interaction learning via self-attentive neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM, 2019.

[172] Ourania Spantidi and Iraklis Anagnostopoulos. How much is too much error? analyzing the impact of approximate multipliers on dnns. In *23rd International Symposium on Quality Electronic Design*, ISQED, 2022.

[173] Ourania Spantidi, Georgios Zervakis, Iraklis Anagnostopoulos, and Jörg Henkel. Energy-efficient dnn inference on approximate accelerators through formal property exploration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[174] Jacob R. Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. Manna: An accelerator for memory-augmented neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[175] Sainbayar Sukhbaatar, Edouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, ACL, 2019.

[176] Sainbayar Sukhbaatar, Edouard Grave, Guillaume Lample, Hervé Jégou, and Armand Joulin. Augmenting self-attention with persistent memory. *arXiv:1907.01470*, 2019.

[177] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *International Conference on Neural Information Processing Systems*, NIPS, 2015.

[178] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. *arXiv:1904.06690*, 2019.

[179] Hao Sun, Xinzhe Liu, Qi Deng, Weixiong Jiang, Shaobo Luo, and Yajun Ha. Efficient fpga implementation of k-nearest-neighbor search algorithm for 3d lidar localization and mapping in smart vehicles. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2020.

[180] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. Treegen: A tree-based transformer architecture for code generation. In *The Thirty-Fourth Conference on Artificial Intelligence*, AAAI, 2020.

[181] Zhu Sun, Jie Yang, Kaidong Feng, Hui Fang, Xinghua Qu, and Yew Soon Ong. Revisiting bundle recommendation: Datasets, tasks, challenges and opportunities for intent-aware product bundling. In *Proceedings of the*

45th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR, 2022.

[182] Synopsys Design Compiler. `https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html`.

[183] Mahdi Taheri, Mohammad Riazati, Mohammad Hasan Ahmadilivani, Maksim Jenihhin, Masoud Daneshtalab, Jaan Raik, Mikael Sjodin, and Bjorn Lisper. Deepaxe: A framework for exploration of approximation and reliability trade-offs in dnn accelerators. *arXiv:2303.08226*, 2023.

[184] N. Takagi. Powering by a table look-up and a multiplication with operand modification. *IEEE Transactions on Computers*, 1998.

[185] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv:1905.11946*, 2020.

[186] Marcello Traiola, Alessandro Savino, Mario Barbareschi, Stefano Di Carlo, and Alberto Bosio. Predicting the impact of functional approximation: from component- to application-level. In *IEEE 24th International Symposium on On-Line Testing And Robust System Design*, IOLTS, 2018.

[187] Tomoaki Tsumura, Ikuma Suzuki, Yasuki Ikeuchi, Hiroshi Matsuo, Hiroshi Nakashima, and Yasuhiko Nakashima. Design and evaluation of an auto-memoization processor. In *Proceedings of the IASTED International Multi-Conference: Parallel and Distributed Computing and Networks*, PDCN, 2007.

[188] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS, 2017.

[189] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. GLUE: A multi-task benchmark and analysis

platform for natural language understanding. In *International Conference on Learning Representations*, ICLR, 2019.

[190] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. *arXiv:1803.02349*, 2018.

[191] P. Wang, Z. Liu, H. Wang, and D. Wang. Data-centric computation mode for convolution in deep neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, IJCNN, 2017.

[192] Ruoxi Wang, Rakesh Shivanna, Derek Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed Chi. Dcn v2: Improved deep & cross network and practical lessons for web-scale learning to rank systems. In *Proceedings of the International World Wide Web Conference*, WWW, 2021.

[193] Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv:2006.04768*, 2020.

[194] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Tie-Yan Liu, and Wei Chen. A theoretical analysis of NDCG type ranking measures. In *Proceedings of the 26th Annual Conference on Learning Theory*, COLT, 2013.

[195] Yuchung J Wang and George Y Wong. Stochastic blockmodels for directed graphs. *Journal of the American Statistical Association*, 1987.

[196] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, R'emi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *arXiv:1910.03771*, 2019.

[197] Tao Wu, Ellie Ka-In Chio, Heng-Tze Cheng, Yu Du, Steffen Rendle, Dima Kuzmin, Ritesh Agarwal, Li Zhang, John Anderson, Sarvjeet

Singh, Tushar Chandra, Ed H. Chi, Wen Li, Ankit Kumar, Xiang Ma, Alex Soares, Nitin Jindal, and Pei Cao. Zero-shot heterogeneous transfer learning from recommender systems to cold-start search retrieval. In *Proceedings of the ACM International Conference on Information & Knowledge Management*, CIKM, 2020.

[198] Zhanghao Wu*, Zhijian Liu*, Ji Lin, Yujun Lin, and Song Han. Lite transformer with long-short range attention. In *International Conference on Learning Representations*, ICLR, 2020.

[199] Tiancheng Xu, Boyuan Tian, and Yuhao Zhu. Tigris: Architecture and algorithms for 3d perception in point clouds. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[200] Zihao Ye, Qipeng Guo, Quan Gan, Xipeng Qiu, and Zheng Zhang. Bp-transformer: Modelling long-range context via binary partitioning. *arXiv:1911.04070*, 2019.

[201] Ofir Zafrir, Guy Boudoukh, Peter Izsak, and Moshe Wasserblat. Q8bert: Quantized 8bit bert. In *Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition*, EMC2-NIPS, 2019.

[202] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences. *arXiv:2007.14062*, 2020.

[203] Rizgar Zebari, Adnan Abdulazeez, Diyar Zeebaree, Dilovan Zebari, and Jwan Saeed. A comprehensive review of dimensionality reduction techniques for feature selection and feature extraction. *Journal of Applied Science and Technology Trends*, 2020.

[204] Jingtao Zhan, Jiaxin Mao, Yiqun Liu, Jiafeng Guo, Min Zhang, and

Shaoping Ma. Jointly optimizing query encoder and product quantization to improve retrieval performance. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management*, CIKM, 2021.

[205] Guowei Zhang and Daniel Sanchez. Leveraging caches to accelerate hash tables and memoization. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2019.

[206] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *Proceedings of the 36th International Conference on Machine Learning*, ICML, 2019.

[207] Haowen Zhang, Yabo Dong, Jing Li, and Duanqing Xu. Dynamic time warping under product quantization, with applications to time series data similarity search. *IEEE Internet of Things Journal*, 2021.

[208] Jialiang Zhang, Soroosh Khoram, and Jing Li. Efficient large-scale approximate nearest neighbor search on opencl fpga. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, 2018.

[209] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-X: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO, 2016.

[210] X. Zhang, F. X. Yu, R. Guo, S. Kumar, S. Wang, and S. Chang. Fast orthogonal projection based on kronecker product. In *IEEE International Conference on Computer Vision*, ICCV, 2015.

[211] Weijie Zhao, Shulong Tan, and Ping Li. Song: Approximate nearest neighbor search on gpu. In *Proceedings of the IEEE International Conference on Data Engineering*, ICDE, 2020.

[212] Weijie Zhao, Deping Xie, Ronglai Jia, Yulei Qian, Ruiquan Ding, Mingming Sun, and Ping Li. Distributed hierarchical gpu parameter server for massive scale deep learning ads systems. *arXiv:2003.05622*, 2020.

[213] Weijie Zhao, Jingyuan Zhang, Deping Xie, Yulei Qian, Ronglai Jia, and Ping Li. Aibox: Ctr prediction model training on a single node. In *In proceedings of the ACM International Conference on Information and Knowledge Management*, CIKM, 2019.

[214] Chang Zhou, Jinze Bai, Junshuai Song, Xiaofei Liu, Zhengchao Zhao, Xiusi Chen, and Jun Gao. ATRank: An attention-based user behavior modeling framework for recommendation. In *AAAI Conference on Artificial Intelligence*, AAAI, 2018.

[215] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. Deep interest evolution network for click-through rate prediction. In *AAAI Conference on Artificial Intelligence*, AAAI, 2019.

국문초록

심층신경망 (DNN)은 데이터 내에 객체간 복잡한 관계를 파악하기 위해서 빠르게 크기가 증가하고 있고, 데이터셋 또한 현실 세계에 존재하는 복잡한 정보들을 담기위해서 크기와 복잡도가 증가하고 있다. 이러한 복잡한 정보들을 담기위해서 임베딩이라는 자료구조가 가장 흔히 쓰이고 있다. 임베딩은 고차원 희소 공간을 원래 특징의 의미를 보존하는 저차원 밀도 공간으로 벡터 투영시킨 자료구조이다. 이를 통해 심층신경망이 데이터를 효율적으로 처리할 수 있다.

이렇게 모델과 데이터셋의 크기가 모두 증가하면서 임베딩 관련 연산들에 큰 오버헤드가 발생하여 추천 시스템, NLP 및 컴퓨터 비전과 같은 인공지능 애플리케이션들의 성능 저하를 야기시킨다. 이에 따라, 이 어플리케이션들은 크게 세가지의 어려움에 처해있다. 첫번째는 메모리 대역폭 사용량 증가, 두번째는 계산량의 증가, 세번째는 하드웨어의 비효율성이다. 이 논문은 이러한 문제를 갖고 있는 세 가지 다른 인공지능 애플리케이션을 분석하여 다양한 최적화 기회를 찾아낸다. 그리고 이에 근거하여 각 문제점을 해결하기 위한 세 가지 제안을 소개한다.

첫째로, 추천 시스템에서 임베딩 리덕션 연산을 최적화하기 위해 MERCI 라는 새로운 메모아이제이션 프레임워크를 제시한다. 이 새로운 메모아이제이션 프레임워크는 실제 데이터셋의 특징들 중에 서로 같이 자주 등장하는 특징들의 집합 구조를 활용한다. 이를 위해 적은 추가 용량으로 높은 커버리지를 얻을 수 있는 특징 집합 구조를 식별하는 Correlation-Aware Variable-Sized Clustering 을 제안한다. 또한, 다양한 테크닉을 도입하여 줄인 메모리 접근량의 효과를 최대화할 수 있도록 한다.

둘째로, 자연어 처리, 추천 시스템, 컴퓨터 비전 등에서 널리 사용되는 셀프 어텐션 연산의 효율적인 계산을 위한 ELSA 라는 소프트웨어 하드웨어 최적화 연구를 소개한다. ELSA 는 최종 결과에 영향을 미칠 가능성이 적은 관계들을 효율적으로 식별하고 관련된 계산들을 생략할 수 있는 새로운 근사 알고리즘을 제시한다. 또한, 이렇게 줄인 계산량으로 많은 속도 향상과 에너지 절약을 얻을 수 있는 하드웨어 가속기 디자인을 소개한다.

셋째로, 압축 기반 근접 이웃 검색 (ANNS)을 하드웨어 가속기를 통해 최적화하는 ANNA 라는 연구를 소개한다. ANNA 는 신중하게 설계된 모듈과 세분화된 파이프라이닝을 통해 기존 하드웨어 (예: CPU 및 GPU)의 비효율성을 해결한다. ANNA 는 메타, 구글과 같은 대기업의 라이브러리들 다양한 검색 세팅을 유연하게 지원하고 메모리 트래픽 최적화 기술을 사용하여 대규모 데이터셋을 보다 효율적으로 지원한다.

본 논문에서 제시된 제안들은 모델이 데이터 센터에 채택되어 최종 사용자에게 서비스를 제공하는 현실적인 사용 사례에서 매우 관련성이 높다. 본 논문의 제안들은 이러한 사용 사례와 관련된 엄격한 처리량 및 에너지 제약 조건을 충족하는 데 도움이 될 수 있다. 또한, 이 연구 결과는 AI 분야에서 미래 연구 및 개발에 유용한 참고 자료로 활용될 수 있다.

주요어: 머신러닝, 임베딩 연산, 임베딩 리덕션, 셀프 어텐션, 압축 기반 유사도 검색, 소프트웨어/하드웨어 공동 디자인, 하드웨어 가속기, 소프트웨어 최적화
학번: 2018-25551