POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Deep Recommender Models Data Flow Optimization for AI Accelerators

Supervisors

Prof. Daniele Jahier PAGLIARI

Doc. Lukas CAVIGELLI

Doc. Renzo ANDRI

Candidate

Giuseppe RUGGERI

July 2023

Summary

Deep Learning-based Recommender Models (DLRMs) have become indispensable tools for businesses to provide effective personalized recommendations to end users. As a result, the workload introduced by these models is extremely relevant, representing, for instance, more than 79% of the AI workload in Meta's data centers. Therefore, optimizing such models is crucial and can lead to significant energy savings, increased throughput, and better real-time responsiveness. State-of-the-art DLRMs present big performance limitations due to embedding layers, which project sparse categorical features to dense, continuous embedding vectors. In particular, the bottleneck is given by the large number of random memory accesses performed to retrieve a multitude of small embedding vectors from look-up tables stored in off-chip memory. To mitigate this issue, some existing approaches exploit the large bandwidth offered by High Bandwidth Memory (HBM), while others propose to build clusters of heterogeneous nodes exploiting the advantages introduced by each platform. Furthermore, some methods propose to model embedding access patterns to place "hot rows" in a cache and/or to build an entire hierarchical memory system tailored for the embedding lookups. However, existing approaches are limited by the variable size of the models (from a few MBs to hundreds of GBs), as well as their dependency on input query distributions. This thesis aims to study and design embedding lookup dataflows for the Huawei Ascend AI processors, particularly focusing on exploiting the available software-controlled on-chip buffers (scratchpad memories) in the most effective way. More specifically, the work proposes four strategies that determine which buffer is used to store embedding tables and how lookups are performed. Two of the strategies build on the idea of persistently preloading the tables in one relatively large on-chip L1 buffer to be then able to perform fast lookups from there. Besides, since the AI accelerators are multi-core, the workload of embedding layers is split following two parallelization approaches. One exploits the classical single-instruction-multiple-data (SIMD) paradigm, which splits the input batch evenly across the cores. The other leverages a multiple-instruction-multiple-data (MIMD) paradigm, thus resulting in an asymmetric core execution and unlocking the possibility of splitting the tables into chunks to be preloaded in the L1 buffer of specific cores. Since different strategies are more

effective for differently shaped tables, two policy optimization problems are solved through heuristics and greedy solutions. Through extensive experiments using both real embedding tables from a model used in production and synthetic ones, the proposed strategies and policies are compared to a black-box baseline obtained from a sophisticated compiler (ATC) which applies various optimizations and exploits built-in operators written by experts. Results show that the baseline is extremely dependent on the input query distribution, as it faces up to 24x higher average latency when the input query is fixed. In contrast, the proposed strategies are not only independent from the input query distribution, but they also provide better throughput vs. worst-case latency trade-offs compared to the baseline. More importantly, when using a fixed and uniform distribution, the proposed method achieves up to 116x and 3.5x lower average latency w.r.t the baseline.

Acknowledgements

This thesis project represents the end of a long journey that has contributed to my personal growth, not only from the knowledge perspective but especially from the human point of view. The project was done while I was working at Huawei Zurich Research Center, in Zurich, which has left me with an unforgettable abroad experience and very good memories that I will keep with care for the rest of my life. Huge thanks are due to all my peers working at ZRC, who have always supported me throughout the whole experience with precious technical hints and their kindness and friendship. A particular mention has to be given to Filip, Alberto, and Ilias, who are working at ZRC, with whom I shared incredible moments both in the office and outside, and who have never let me feel alone. I would also like to express my sincere gratitude to my supervisors, Lukas and Renzo, working at ZRC, who have been ideal mentors, giving me tons of advice and supporting every step of the project. Likewise, I would like to thank my academic supervisor Daniele Pagliari for having supported me both technically and humanly from the beginning of this experience to the end of it, cheering me always up when I was going through difficult times. Moreover, I would like to thank my friends Amedeo, Mauro, Gabriele, and Alessandro, with whom I've shared tons of beautiful moments throughout these years and who have always believed in me. Particular credit must be given to my girlfriend Alessia, who has supported me since the beginning of my studies, always believing in me and putting light in moments of darkness. I would also like to thank my girlfriend's family, who have always been there for me in times of need. Lastly, I would like to thank all members of my family, including my brother Franco, who has helped me a lot by giving me precious hints, Nino, and my little sister Angela. I would also like to mention my grandmother Stefana, who has always believed in me and genuinely cared about me.

With all my sincere gratitude, thank you to all the people who have made my life better throughout these years, my academic achievements are only due to you.
July 2023
Giuseppe Ruggeri

Table of Contents

List of Tables VIII

List of Figures IX

Acronyms XIV

1 Introduction 1

2 Background 6
2.1 Recommender systems . . . 6
2.1.1 Deep learning-based Recommender Models (DLRMs) . . . 8
2.1.2 DLRMs inference key challenges . . . 11
2.1.3 Deployment platforms . . . 12
2.2 AI hardware accelerators . . . 13
2.3 Huawei Ascend Computing . . . 16
2.3.1 Huawei Da Vinci architecture . . . 16
2.3.2 Ascend AI accelerators . . . 19
2.3.3 Ascend software stack . . . 21
2.4 Data flow . . . 25

3 Related works 27
3.1 System-level optimizations . . . 28
3.1.1 Multiple nodes / cluster optimizations . . . 30
3.1.2 Other solutions with custom hardware . . . 31
3.2 Dataflow optimizations . . . 31
3.2.1 Cache-storage level optimization . . . 32
3.2.2 Load balancing: table sharding optimizations . . . 33

4 Accelerating embedding lookups on Ascend AI accelerators 36
4.1 Problem characterization . . . 36
4.2 ATC-based embedding lookups . . . 39

4.3 Embedding lookups strategy design . . . 40
4.4 Symmetric inter-core lookup data flow . . . 42
4.4.1 Direct lookups data flow . . . 43
4.4.2 Unified Buffer (UB) lookups . . . 45
4.4.3 Model-level symmetric lookup strategy . . . 48
4.5 Asymmetric inter-core lookup data flow . . . 51
4.5.1 Tasks and chunking . . . 51
4.5.2 Sharding and load balancing problems . . . 53

5 Experimental results 56
5.1 Experiments setting . . . 56
5.1.1 Performance metrics . . . 59
5.2 Baseline assessment . . . 61
5.3 Table-level strategies assessment . . . 67
5.3.1 Hardware utilization . . . 67
5.3.2 Assessment of input query distributions . . . 72
5.3.3 Throughput vs. latency trade-offs . . . 73
5.4 Model-level policies assessment . . . 75

6 Conclusions and future works 83

Bibliography 87

List of Tables

2.1 The table shows the parallelism of the three computational units of the Ascend Da Vinci Architecture ([17]). The parallelism is expressed as the maximum number of bytes processable by the unit in parallel. The vector unit can process up to 2048bit INT8/FP16/FP32 vector with special functions, while the mac unit can process 4096 fp16 MACs + 8192 int8 MACs. 18

4.1 The table shows the bandwidths of the different components and memories included in the Ascend Da Vinci architecture. 40

5.1 The table shows the performance achieved on the DCNv2 model on the Ascend 910 using different strategies. In particular, the values refer to the P99 worst-case latency and average latency expressed in μs using different batch sizes. The values refer to 200 different runs. 'U' stands for uniform distribution, whereas 'F' stands for fixed distribution. 82

List of Figures

2.1 Architecture of the Wide and Deep Learning for Recommender Systems released by Google ([7]). 10
2.2 DCNv2 architecture proposed by Google to build Recommender models ([6]). (left) Overall architecture. (right) Cross layer. 10
2.3 Huawei Da Vinci basic computational blocks. The image is taken from [17]. 17
2.4 Da Vinci AI core 18
2.5 Storage access relationships between the memory components both for the Cube unit instructions (left) and the Vector unit ones (right). 19
2.6 Ascend310, an AI Processor (SoC) specifically designed for accelerating AI inference. 20
2.7 Ascend910, an AI Processor (SoC) specifically designed for accelerating AI training in data centers. 20
2.8 Huawei Atlas devices for different needs and scenarios, built leveraging Ascend AI SoCs. 22
2.9 ACL standard API call flow including two working modes. One is based on executing single operators, whereas the other executes entire model inferences. 23
2.10 ATC compiler flow of operations to generate an executable offline model. On the left, we see the various optimizations that ATC applies when converting a model from a framework such as TensorFlow. On the right, a single operator, described in a proprietary IR, is instead converted. 24

5.1 Table size distribution for the 84 tables of the DCNv2 model used in production. 59
5.2 Sequence length distribution among the different tables of the DCNv2 used in production. 60
5.3 Layer-wise average latency using the baseline strategy on the DCNv2 model executing on the Ascend 910. The input distribution is uniform and all the 32 cores are enabled. 62

5.4 GatherV2 CANN operator average hardware core unit utilization considering the DCNv2 model executed using the baseline strategy on the Ascend 910. The input distribution is uniform, and all 32 cores are enabled. 63

5.5 ReduceSumD CANN operator average hardware core unit utilization considering the DCNv2 model executed using the baseline strategy on the Ascend 910. The input distribution is uniform and all the 32 cores are enabled. 64

5.6 Baseline and GM strategies used on a single table made of 32760 rows, with sequence length set to 5 and enabling all the 32 cores of the Ascend 910. The fixed input distribution aims to make multiple cores conflict on the same cache line. 65

5.7 Baseline strategy applied on the DCNv2 model used in production. All 32 cores of the Ascend 910 are enabled and the fixed input distribution is used to cause cache conflicts between the cores. . . . 66

5.8 Hardware utilization of the core units of the Ascend 910. The runs refer to the execution of the GM strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled. 68

5.9 Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the L1 strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled. 69

5.10 Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the GM-UB strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled. 70

5.11 Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the L1-UB strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled. 71

5.12 Average latency achieved on a table with 32760 rows and sequence length set to 5 on the Ascend 910. On the left, all 32 cores are enabled, whereas on the right, only one core is enabled, thus showing the presence or absence of cache conflicts. 72

5.13 Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) table with 1600 rows. (bottom) table with 32760 rows. The aggregation refers to 200 runs on the Ascend 910 performing pooling-based lookups using a uniform query distribution with the five competing strategies, a sequence length set to 5, and enabling all 32 cores. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy. 73

5.14 Each pie chart refers to the percentage of tables of the DCNv2 model considering the Ascend 910 and assigned by the policy for being computed with a specific strategy. In particular, there are two kinds of policy: asymmetric, which tries to assign the optimal strategies to get the most out of the MIMD paradigm, and symmetric, which instead aims at finding an optimal assignment of the strategies for the SIMD paradigm. From left to right, we see the effects of having fewer or more cores available, and from top to bottom, the batch size affects the number of lookups performed, thus resulting in a different policy. 76

5.15 Inter-core load-balancing assessment of the optimal asymmetric policy applied on the embedding tables of the DCNv2 model used in production. The values are the average latencies measured executing only with one core the set of tables with the strategies defined by the optimal policy. Here we refer to feeding the model queries sampled from a uniform distribution. 77

5.16 Inter-core load-balancing assessment of the optimal asymmetric policy applied on the embedding tables of the DCNv2 model used in production. The values are the average latencies measured executing only with one core the set of tables with the strategies defined by the optimal policy. Here we are referring to feeding to the model queries made of indices fixed to the value 0. 78

5.17 Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) enabling all 32 cores. (bottom) enabling only 8 cores. The aggregation refers to 200 runs of the DCNv2 model used in production on the Ascend 910, using the symmetric, asymmetric, and baseline approaches. The queries are sampled from a uniform distribution. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy. 79
5.18 Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) enabling all 32 cores. (bottom) enabling only 8 cores. The aggregation refers to 200 runs of the DCNv2 model on the Ascend 910, using the symmetric, asymmetric, and baseline approaches. The query indices are all set to the value 0 to test the problem of cache conflicts. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy. 80

Acronyms

AI
artificial intelligence

DL
deep learning

DNN
deep neural network

DLRM
deep learning-based recommender model

SLA
ervice level agreement

SIMD
single-instruction-multiple-data

SIMT
single-instruction-multiple-threads

MIMD
multiple-instruction-multiple-data

HBM
high-bandwidth memory

ATC
Ascend Tensor Compiler

CPU

central processing unit

GPU

graphics processing unit

GPU

general-purpose computing on graphics processing unit

FPGA

field programmable gate arrays

SoC

system-on-a-chip

ASIC

application specific integrated circuit

API

application programming interface

UB

unified buffer

GM

global memory

Chapter 1

Introduction

Nowadays, deep learning has become a keystone for many production-scale web services. As systems built leveraging the incredible capabilities of deep learning are becoming more and more widespread, the demand for running such services in data centers has become huge. A particular class of those services is represented by recommender systems, which are largely exploited by web services of any kind to give their users personalized recommendations. The services built around personalized recommendations span many domains, including e-commerce, music and video streaming platforms, social networks, and many more. Gigantic players in the industry, such as Amazon, Facebook, Youtube, Spotify, and many more, have invested a lot of effort in the last years in building powerful recommender systems, being at the center of their business success. In particular, deep learning recommender models, or DLRMs, have become the state-of-the-art method for providing effective recommendations to users. Thus, the widespread adoption of such models has caused a dramatic increase in the computing and storage demands in today's data centers. According to [1], the inference of DLRMs represents more than 79% of the AI workload in Meta's data centers. Optimizing the inference workloads of such models has become of uttermost importance. Even though the research community has spent a lot of effort throughout the last decade in optimizing the performance of deep learning workloads, unfortunately, these optimizations do not apply to DLRMs due to their architecture. In particular, DLRMs exploit massive datasets of both categorical and dense features to learn to solve problems such as Learning to Rank (LTR), which consists in learning to rank a set of items according to a given ranking function. To this extent, deep neural networks are exploited to learn complex feature interactions of both items and users' characteristics to be used to solve such problems effectively. However, categorical features are naturally represented as extremely sparse one-hot encoded vectors, leading to poor capabilities to solve LTR problems. For this reason, DLRMs are built around embedding layers, which are thought specifically for mapping categorical features to dense

embedding vectors of fixed size, solving the sparsity problem. To do the mapping, since the inference of such layers must be designed to have low latency, tables of precomputed embedding vectors are created and stored persistently in the memory, to be later looked up at inference time. As a result, the input to such layers becomes a query of integer indices used to look up the tables to retrieve the embedding vectors. Embedding vectors of different tables are then combined together and also with dense features, and the result is fed to the subsequent feed-forward deep neural network to make the prediction. Unfortunately, as underlined by [2], the look-up operations represent an important bottleneck for the performance of the inference of such models. More specifically, they involve a large amount of random off-chip memory accesses to retrieve many small embedding vectors. Consequently, standard off-chip memories, such as DRAMs, waste a lot of their bandwidth as the amount of data retrieved for each request is very small, thus resulting in poor memory bandwidth utilization.

For this reason, many works in the literature have recently proposed solutions to mitigate this problem. A large body of research focuses on using standard CPU or GPU-based platforms, already available in every data center, and thus not requiring any additional hardware. Others instead utilize Field Programmable Gate Arrays (FPGAs) coupled with High-Bandwidth Memory (HBM) to implement custom solutions capable of performing fast embedding lookup operations. In contrast, this work focuses on optimizing the lookup operations involved in DRLMs for programmable AI accelerators.

Unlike general-purpose hardware like CPUs or GPUs, AI accelerators focus on supporting only the operators needed by AI applications, resulting in a much more efficient and effective overall hardware design. Moreover, unlike FPGAs and ASICs, they are flexible enough to support a wide variety of different AI applications by offering a rich software stack made of tools, compilers, APIs, and integration with existing DL frameworks such as TensorFlow and PyTorch, thus resulting in a transparent acceleration of many already existing applications. However, developing compilers and tools that seamlessly lower high-level code into an efficient representation for the accelerator is a challenging task. This is due to the abundant ways to express a complex computation in hardware, which depends on many factors, including the number of available processing elements (e.g., cores), the amount of memory that the program needs w.r.t the memory hierarchy available, together with the types of memory access patterns and data reuse opportunities. As a result, it is difficult to obtain out-of-the-box performance for an arbitrary AI application just relying on a set of compilers and tools.

To this extent, this thesis proposes to design new strategies for achieving an

optimal data flow tailored to the specific needs of DLRMs. In particular, the data flow is designed for the Huawei Ascend AI accelerators, multi-core SoCs based on the Da Vinci architecture. Specifically, each Da Vinci core is equipped with software-controlled memories (scratchpad memories), including an L1 buffer and a Unified Buffer that are specific for each core, and a shared L2 cache together with an optional on-chip HBM for achieving high-throughput data transfers. To design a custom data flow for the Ascend accelerators, the rich Ascend software stack is exploited, which offers a high level of expressiveness achieved by exposing high-level APIs written in Python for the development of effective custom operators that can run efficiently on the accelerators. The thesis proposes new strategies to write custom operators for performing DLRM Look-ups in a way that mitigates the performance bottleneck introduced by the many random off-chip memory accesses. We focus on pooling-based DLRMs, where the embedding layers do not retrieve only one embedding vector for each sample of the batch but rather a sequence of embedding vectors given a vector of indices. Once the sequence of embedding vectors is retrieved, a pooling function is then applied to combine the sequence of vectors into only one vector. The pooling function is usually an associative and commutative function, such as the element-wise vector sum, subtract, max, or min. The thesis proposes four strategies that can be applied for different use cases, which differentiate from one another both for the memory used for storing the embedding tables and for the strategy used for performing the actual gathering and reduction operations. A first strategy proposes to preload an embedding table in the L1 buffer persistently, to be then able to perform fast lookups directly from there in a sequential way. Two other strategies, instead, propose to move all the chunks of an embedding table, one after the other, in the unified buffer, to be then able to perform fast lookups from there in a vectorized way. The two strategies differentiate for the source memory of the original embedding table, which can be either the global memory (or off-chip memory) or the L1 buffer as it has been previously preloaded. Since the embedding tables have a size ranging from a few KBs to hundreds of GBs, the last strategy performs the look-ups directly from the off-chip memory, thus supporting tables of any size. Results showed that the performance of each strategy greatly depends on the characteristics of embedding tables, thus setting the ground for a new problem of finding an optimal policy, that is, which strategy to use for a given table, in a given use case.

Given the multi-core nature of the Ascend accelerators, the thesis explores two different ways of splitting the workload across the many cores available. The first one is based on the classical single-instruction-multiple-data paradigm (SIMD), which consists in having only one program (or kernel) that gets executed precisely in the same way in all the cores but over a different set of data. In particular, the developed approach splits the workload evenly across the cores along the batch size

dimension and includes an optimization algorithm for finding an optimal policy. The algorithm developed is based on a greedy strategy with heuristics which tries first to find an optimal set of tables to preload in the L1 buffer and then defines the set of strategies to use for all the tables according to a simple performance model. On the other hand, since preliminary results showed that preloading the embedding tables in the L1 buffer brings huge performance improvements compared to the other strategies, the other approach tries to make the most of this idea. In particular, the approach is instead based on the multiple-instruction-multiple-data paradigm (MIMD), which consists in executing a different program in each core, thus offering the opportunity of preloading different tables, or chunks of them, in the L1 buffer of each core. As a result, with this approach, the aggregated L1 buffer reaches a size much larger than the one that uses the SIMD paradigm. Again, a policy optimization problem must be solved to define not only the strategy with which to process each table but also which tables and portion of the batch to process in each core, thus resulting in a much more complicated problem to solve. Again, since the goal was to design a strategy that can seamlessly run with any set of tables and model, a greedy solution based on heuristics is exploited so that it can run almost in real-time.

To assess the effectiveness of the proposed method, the strategies were compared with a baseline method consisting in exploiting the great capabilities of the default compiler for Ascend accelerators together with built-in operators written by experts at Huawei. The comparisons were made by running multiple iterations on the Ascend 910, which is made of 32 Da Vinci cores, and equipped with an on-chip HBM for making the baseline a strong opponent, thus resulting in a fair comparison. Moreover, the comparisons were made both at the level of single tables, synthetic and specifically crafted for evaluating the performance of the strategies, and at model-level, by using a DLRM model used in production by an anonymous company. Furthermore, two different input query distributions were used, a uniform one, which consists of sampling indices from a uniform distribution, and a fixed one, which instead fixes all the indices to the same value. The performance of the strategies was evaluated by considering different metrics. First, the average throughput, expressed as queries per second (qps), was used, which is a good indicator of the average hardware utilization and thus energy efficiency. The average latency was also measured, defined as the total time taken by the accelerator to process a batch of queries, and averaged across many different iterations, thus being a good proxy for assessing the average user experience. Lastly, since DLRMs are usually real-time systems released under Service Level Agreement (SLA) requirements, the P-99 worst-case latency was used to assess the worst-case scenarios. Moreover, the trade-offs between the average throughput vs. average and P-99 worst-case latency were evaluated to assess the overall optimality of the proposed strategies in

a deployment scenario. Experimental results show that the baseline experiences a huge drop in performance when the input query is made of fixed indices compared to when it is sampled from a uniform distribution, showing up to a 24x worsening in the average latency, and thus demonstrating to be dramatically dependent on the input query distribution. On the contrary, not only the proposed strategies demonstrated to be independent of the input distribution used, but they also delivered outstanding improvements over the baseline achieving up to a 116x and 3.5x lower average latency for the fixed and uniform distributions, respectively. Besides, the two policies provide better trade-offs between the average throughput and P-99 worst-case latency compared to the baseline for almost every batch size considered, and for both the input distributions considered.

The rest of the manuscript is articulated as follows. In chapter 2 there will be all the information that is needed for understanding the topic of the thesis, including a general description of what recommender systems are and how they are classified, what specifically Deep Learning-based Recommender Models are, and what are the deployment problems of such models. Moreover, a general introduction about the AI hardware accelerators will be given, and the specific Ascend AI accelerators and the software stack will be described. Next, in chapter 3, there will be a comprehensive analysis of the existing works that propose different solutions for accelerating the inference of DLRMs. Then, in chapter 4, the proposed strategy will be described comprehensively. Lastly, in chapter 5, all the experimental results will be discussed, and in chapter 6, some conclusions will be drawn and future works proposed.

Chapter 2

Background

This section includes all the relevant information and knowledge necessary for understanding the topics of this thesis. First, a general introduction to recommender systems and how they can build is provided, putting particular emphasis on deep learning-based recommender models (DLRMs), a core topic of this manuscript. Then, different hardware platforms for executing the inference of such models are described, with the pros and cons of each one. Among them, particular regard is given to AI hardware accelerators, a central element of this thesis. More specifically, the Huawei Ascend AI accelerators and the associated software stack are described in detail. Lastly, a few general principles for designing effective data flows are reported.

2.1 Recommender systems

Due to the ever-increasing success of web-based applications, ranging from e-commerce, social networks, music and video platforms, and many more, there has been an increasing need to provide effective personalized recommendations to users in the last decade. To this extent, big companies, such as Youtube, Amazon, Spotify, and many more, rely on sophisticated recommender systems to provide recommendations to their users. For instance, according to [1], 75% of movies watched on Netflix and 60% of videos viewed on YouTube are based on suggestions from their recommender systems. The effectiveness of such techniques has been demonstrated to increase income, motivating the importance of building better and better techniques over the years. The goal is accomplished by building recommender systems (RS), a specific information filtering system that predicts user preferences. Information filtering systems are essential nowadays, as the abundance of available data makes it hard to discriminate relevant data from unuseful ones. More specifically, recommender systems operate mainly on two entities. On the

one hand, there are the users of a specific service, each one described with a set of features, such as height, age, gender, and so forth. On the other hand, a specific service operates on a set of items with which users can interact, such as products that a user can buy on e-commerce like Amazon or songs played on a music streaming platform like Spotify. Notice that the interactions between the users and the items can be of any kind, such as a user's click on a specific product on a web page or a rating on a movie given explicitly by a user. As a result, it is evident that the number of interactions, given their granularity, explodes in size, as underlined in [3], leading to millions of interactions combined with hundreds of thousands of different users. Put it all together, given users, items, and interactions, the goal of a recommender system is to extract from all the available items, which can be millions, a set of potentially *interesting* items for a given user based on his preference.

Categories of recommender systems Many methods have been developed and evolved over the years to extract effective personalized recommendations to exploit the available data most profitably. In particular, the methods can be categorized in many ways, but all are based on modeling the users' preferences starting from their interactions with the items.

1. Content-Based Filtering methods
   First, recommender systems that fall under the category of *Content-Based Filtering* are based on extracting users' preferences by building profiles, which can be, for instance, the set of the last ten items bought, and then, based on these profiles, recommendations can be made by ranking the items for their similarity with the user's profile. A simple way of implementing such an idea is to map each item with its features to a low-dimensional embedding space. Then, using a similarity metric, we can find the nearest neighbors of an item in such embedding space and propose such items to the user.

2. Collaborative Filtering (CF) methods
   Another category of methods, categorized as *Collaborative Filtering*, tries to identify patterns and similarities by considering groups of similar users or items. In reality, these methods can be further categorized into two sub-types, called *Memory-based CF* and *Model-based CF*, based on how they model the interactions between users and items.

   The first one is based on directly modeling similar preferences. For instance, if two users have similar preferences and one has interacted with one item, which the other has not, the system could recommend that item. Moreover, considering similarities between items instead, if two items are similar, and

one user has interacted with one of the two items, a good recommendation would be the other one.

The other sub-type, called *Model-based CF*, tries to model the interaction matrix between items and users by mapping each user and item into an embedding space. Notably, the most important model for vanilla recommender systems, Matrix Factorization, falls in this category. It consists mainly in decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices and then learning such two matrices so that their product equals the original interaction matrix, as underlined in [4].

Explicit and implicit feedback To extract users' preferences, systems need to collect feedback from them. In particular, users' feedback can be explicit or implicit, based on whether the user has given explicitly a preference over an item or not. Explicit feedbacks are, for example, ratings given directly by users, such as IMBDB stars for movies or thumbs-up and thumbs-down for YouTube videos. However, this kind of feedback is only sometimes available as users might not want to rate products. On the counter, implicit feedbacks, such as user clicks or viewed videos, are often readily available. Therefore, many recommender systems are built on top of implicit feedback, which indirectly models users' preferences through their behavior.

Recommendation tasks It should now be clear that recommender systems try to recommend a set of items based on the users' preferences. In particular, various tasks can be solved using such systems, depending on the application domain and the usable feedback. For instance, considering movies with explicit ratings, recommender systems can be built by simply predicting the rating a specific user would give the movies, which can be thus ranked and recommended. Moreover, another prevalent task is called click-through rate prediction (CTR), which consists in predicting the likelihood that a user will click on an item or advertisement, thus being very used in all advertisement-based systems such as Google Ads. In general, we can see that all those tasks are based on the capacity of an algorithm to rank items based on users' preferences, thus falling under the broader category of tasks called Learning to Rank (LTR).

2.1.1 Deep learning-based Recommender Models (DLRMs)

It is well known that deep neural networks are outstanding feature extractors, making them a perfect candidate for building recommender systems. They can capture potentially infinite degrees of interactions between input features, still generating

compact, effective embeddings that can be exploited for solving complex tasks such as LTR. The key concept is to learn the interaction between different features belonging to users, items, or interactions, to obtain a representation expressive enough to solve the task effectively. This problem is called features interaction (or features cross) learning, and there are several approaches to tackle it. One approach used in the past consists of manually identifying feature crosses, thus requiring domain expertise and exploring a huge combinatorial search space. For instance, supposing to have three features, age, height, and category of the last bought item, the manual approach would consist in choosing, for example, to model the couple, or feature cross, between age and category, arguing that the joint information can offer a valuable contribution for solving the task.

Besides, the trend has changed over the past few years towards using deep neural networks for their incredible capability of capturing effective feature interactions. Many approaches have been proposed to exploit DNNs for recommender systems, all building on similar ideas and falling under the broad category of *Deep learning-based Recommender Models (DLRMs)*. Despite the many differences in the architectures of DLRMs, since most of the available features are categorical, they must be treated in a specific way. In particular, categorical features are naively represented as one-hot encoded vectors, which are extremely sparse and make learning to solve recommendation tasks such as LTR much more difficult.

To this extent, all DLRMs use embedding layers, whose goal is to map categorical features into dense, continuous embedding vectors. Moreover, all DLRMs architectures focus on devising smart topologies to capture effective feature interactions. This is motivated by the fact that standard DNNs do not directly model interactions between different input features but instead leverage very deep architectures to learn high-degree implicit feature interactions. As underlined by [5], [6], and [7], this approach is not adequate for solving recommendation tasks, as it requires training huge models and thus requiring a huge amount of high-quality data, still producing disappointing results.

To tackle this problem, many new architectures have been designed to model feature interactions explicitly. First, Google proposes an architecture called Wide and Deep Learning for Recommender Systems ([7]), which combines traditional linear models with deep ones to solve the task effectively. As figure 2.1 shows, the key concept is exploiting a traditional DNN to capture high-level interactions and wide models that apply a simple linear combination of categorical and dense features to directly model the interactions between them. Moreover, a very interesting approach is again proposed by Google and is called DCNv2 ([6]), which is an improvement over the original work DCN developed by the same authors ([5]). In

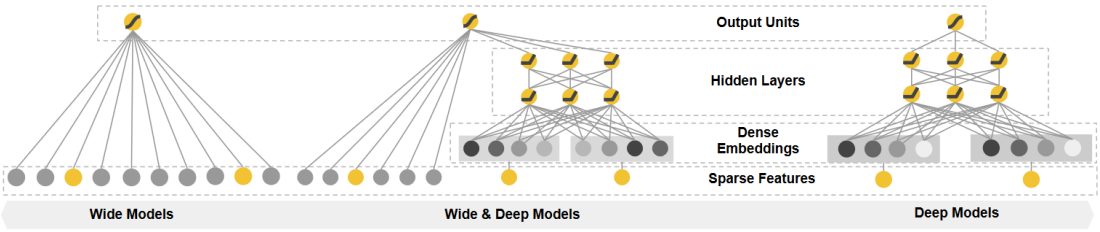

Figure 2.1: Architecture of the Wide and Deep Learning for Recommender Systems released by Google ([7]).

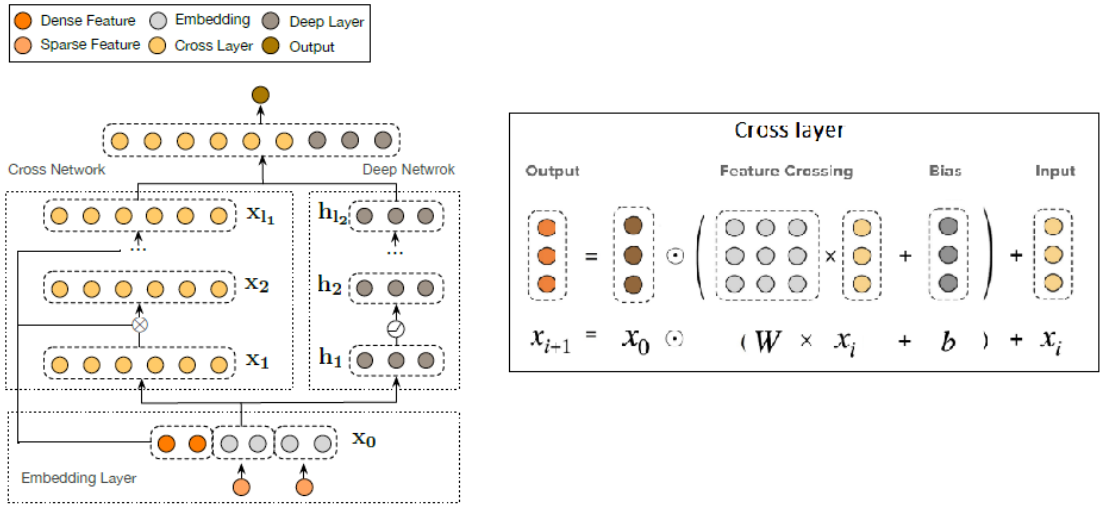

Figure 2.2: DCNv2 architecture proposed by Google to build Recommender models ([6]). (left) Overall architecture. (right) Cross layer.

particular, it is designed to learn explicit feature interactions by directly modeling feature crosses through an innovative cross-network. More details will be given in the next paragraph, as this is the architecture used for benchmarking the strategies proposed by this thesis. Besides, many other works are based on autoencoders ([8]), recurrent neural networks ([9]), transformers ([10]).

DCNv2: Deep and Cross Network As we have briefly mentioned, a very powerful approach for building DLRMs is proposed by Google, which proposes a new architecture called Deep and Cross Network ([6]). Figure 2.2 shows both the overall architecture and the innovative cross layer, which differentiates it from other approaches. In particular, the cross network is made of cross layers, which model explicitly features interactions by computing their output as $x_{i+1} = x_0 \odot (W \times x_i + b) + x_i$, where x_0 is the input vector composed of the input dense

features concatenated with the embedding vectors, x_i is the output vector of the previous layer, which corresponds to the input of the current layer, and x_{i+1} is the output of the current layer. As a result, it is evident how all input features x_0 get explicitly multiplied in each layer to model the feature interactions at various explicit degrees. In fact, stacking more cross layers actually corresponds to multiplying multiple times the input features together, thus resulting in a well-defined, bounded degree feature interaction. Moreover, the output of the last cross layer is combined with the one of a standard DNN, achieving a powerful representation that can capture both bounded-degree explicit feature interactions and high-degree implicit ones.

Embedding layer A key element of the DCNv2 architecture, and many other DLRMs, is the embedding layer. We focus on this layer in particular as it represents the main target for the optimization performed in this thesis. Given a categorical feature with N possible values, it must be converted into a usable format for neural networks. The naive solution is to use a one-hot encoded vector, that is, a vector with N binary elements, among which only the element corresponding to the categorical value is set to 1, while the others are set to 0. Formally, we have: $x_i \rightarrow v_i$, where $v_i \in \mathbf{R}^N$. It is evident that having such a sparse representation leads to model inefficiency, increases storage requirements, and worsens the generalization capability of the model, making, as a consequence solving recommendations tasks more difficult. To this extent, the embedding layer learns a projection matrix $U \in \mathbf{R}^{E,N}$, where E is the chosen dimensionality of the embedding space, to project each categorical feature into a dense, continuous embedding vector by applying a vector to matrix multiplication: $y_i = v_i * U^T$. It is worth noting that the described process happens only at training time to learn an embedding space suitable for the task to solve. At inference time, instead, for performance and efficiency reasons, embedding layers store precomputed embedding vectors, for all the possible categorical values, in a table, which is usually called *embedding table*. Thus, given a categorical feature that can have N possible values, the corresponding embedding table is composed of N rows, each having the size of the chosen embedding dimensionality E. The embedding table, stored in DRAM or other memories, is then looked up at inference time, accessing one row through an index. This implies that the input query, at inference time, is made of integer indices used to perform the look-ups from such tables and whose values range from 0 to $N - 1$.

2.1.2 DLRMs inference key challenges

The nature of the embedding lookup operations, which must be all completed before concatenating the retrieved embeddings and feeding them to the rest of

the network, introduces new performance problems compared to standard DNNs. *First*, given that usually there are tens or even hundreds of embedding tables, look-up operations involve a large number of random accesses to off-chip memory, each one to retrieve a small embedding vector whose dimension usually ranges from 4 to 64 elements. Even worse, embedding layers are usually set to perform more than one lookup for each sample in the batch, ranging from one to even hundreds of look-ups for one single sample. As a consequence, as underlined in [1], this introduces a performance bottleneck, leading to poor memory bandwidth utilization and high-latency inferences in general. Moreover, look-ups are even slower if resorting to state-of-the-art machine learning frameworks such as TensorFlow and PyTorch. Even in an inference-optimized framework such as TensorFlow Serving, many different operators are executed multiple times in the embedding layer, thus increasing the latency of such layers. *Secondly*, the size of the model, due to the varying number and size of embedding tables, ranges significantly from a few MB to hundreds of GB, leading to the impossibility of storing such tables in the memory of hardware accelerators such as FPGAs, GPUs, and so forth. *Third*, DLRMs are often real-time systems deployed under SLA requirements, which usually introduce constraints on the worst-case latency that must be guaranteed. As a result, inference systems must be devised appropriately to guarantee that SLA requirements are met. Moreover, having a boundary on the worst-case latency limits the maximum usable batch size, which is usually increased to achieve a better throughput in most hardware platforms, such as FPGAs and GPUs.

2.1.3 Deployment platforms

The inference of DLRMs is executed in a wide variety of hardware platforms in today's data centers. First, a prevalent and widely used approach is leveraging CPUs as they are already available in every system, thus not necessitating buying new hardware or upgrading existing data centers. Moreover, they are usually characterized by a lower latency than GPUs, and they can be equipped with a large quantity of DRAM, thus making possible to store very large embedding tables.

Another approach is to exploit GPUs, which, based on the single-instruction-multiple-threads (SIMT) paradigm, are well known to perform fast computation by processing large batches of data in parallel. However, they are not very suitable for the inference of DLRMs as they are dramatically limited by the memory capacity, which is often not large enough to store the embedding tables, thus causing many off-chip memory accesses that limit the overall performance.

A very promising line of work focuses instead on FPGAs, which are tailored specifically for the needs of DLRMs. They typically employ the most innovative

HBM (High Memory Bandwidth) technology to do the look-ups in parallel from as many memory interfaces as possible. However, a lot of resources are consumed for performing parallel look-ups, thus leaving scarce remaining resources for the computation of the rest of the model.

Lastly, recently there has been an emerging trend in developing ASICs specific for accelerating AI applications, opening new performance optimization opportunities at the cost of losing some flexibility. However, it is difficult to obtain out-of-the-box good performance, and the same limitations of DLRMs described so far also apply to hardware accelerators if not designed and used appropriately.

2.2 AI hardware accelerators

In the last years, as deep learning applications have become ubiquitous, dominating every domain where it is applicable, there has been an increasing interest in developing specialized hardware for accelerating AI applications. The motivations behind the development of such technologies are several, starting from the *utilization wall* that several systems impose, that is, the presence of a power constraint in a system. This, together with the fact that it was no longer possible to decrease the power consumption of the single transistors, meant that only a fraction of a chip's transistors could be utilized, while the other part had to be turned off (*dark silicon*[1]). This gives an intuition of why the development of specialized hardware has been necessary, that is, exploiting the available transistors in the most effective way while remaining within the power budget.

As opposed to general-purpose hardware, which must support a large number of logical operations, many possible patterns, and program-induced behaviors, accelerators focus only on the subset of operations needed, thus removing a lot of structural redundancy from chips. AI is a good candidate for acceleration as it involves only a handful of operations, including multiplication, addition, and some non-linear operators. Moreover, AI programs are massively parallel and natively represented in a graph-like format, thus making the control flow known at compile-time. This, together with having communication and data reuse patterns fairly confined, opens many opportunities for accelerating such programs.

Architectural components To provide the reader with a high-level view of the fundamental elements upon which AI accelerators are built, a quick summary of

[1]Dark silicon

the major categories of solutions.

The first fundamental block of any programmable hardware is the Instruction Set Architecture (ISA), which includes the set of operations supported and how instructions and operations are encoded by the compiler and later decoded to be executed (e.g., arithmetic instructions, memory and control operations). ISAs differentiate one from another for the granularity of instructions, which is strictly correlated with the complexity of the decode phase. Two very famous kinds of ISAs are RISC (Reduced Instruction Set Computer), characterized by fixed-size simple instructions together with a very simple decoding phase, and CISC (Complex Instruction Set Computer), which contrarily are made of complex, variable-size instructions that can express a complex combination of operations and conditions, at the cost of having a much more complicated decoding phase.

Besides, **domain-specific ISAs** are employed for accelerators, supporting only the minimum set of operations needed and thus obtaining a very simple instruction decoding phase, which contributes to hardware efficiency. Moreover, supporting only the necessary operations simplifies the processing cores and the hardware-software interface, resulting in an efficient accelerator design.

Some of the architectures at the base of AI accelerators are provided in the following to provide a high-level idea of how these systems can be built:

1. *Very-Long Instruction Word (VLIW) Architectures*
   Similarly to CISC, VLIW architectures combine multiple operations into a single complex instruction that dispatches data to a heterogenous data path array of arithmetic and memory units. For instance, in AI accelerators, we could point a tensor to a matrix multiply unit while in parallel sending data portions to a vector and a transpose unit. Since the data path is heterogeneous, we must guarantee that the workload is balanced between the various units, which involves complex static scheduling.

2. *Systolic Arrays*
   Systolic arrays are blocks of multiple processing elements (PEs) hardwired in a fixed order. The entire array operates in *beats*, where each element processes a portion of the data in every compute cycle and streams down the result to the next element. They are extremely efficient at performing matrix multiplications and accumulations, thus being extensively used in almost every architecture, such as Google TPUs ([11] and Nvidia's Tensor Cores ([12]).

3. *Processing in Memory (PIM)*
   One critical aspect concerning all the architectures is represented by the memory bandwidth, which is a performance bottleneck for many AI applications.

Moving data between units is a **few orders** of magnitudes more costly than doing the actual computation, both in terms of energy consumed and latency. Therefore, PIM technologies build on the idea of designing computational units directly inside the memory to avoid the data movement cost completely. Typically the computation is performed analogically (i.e., neuromorphic computing), but it still requires expensive digital-to-analog and analog-to-digital converters.

The very rich landscape Based on the just-described architectures and components, a plethora of solutions have been developed in the last decade. Nvidia is one of the leaders and has brought significant contributions to the field. First, they have developed GPGPUs, general-purpose GPUs built on top of the *single-instruction-multiple-thread* (SIMT) paradigm. They are based on multiple cores running the threads synchronously, thus simplifying the control flow and making them extremely suitable for domains such as dense linear algebra. Not less important is the Nvidia Tensor Cores ([12]), accelerator cores that use systolic operations to accelerate deep learning computation.

Another interesting approach is called Cerebras ([13]), which proposes a design of a chip the size of a pizza box to eliminate the need for inter-chip communication and to reduce latency at the cost of having a massively high power consumption. Besides, leveraging systolic arrays and VLIW architectures, Google TPUv1 ([11]), Groq [14], and Habana ([15]), have found success in the AI acceleration industry.

Moreover, a very interesting approach is the ET-SoC-1 ([16]) proposed by Esperanto Technologies, an AI accelerator made of thousands of cores interconnected in a smart way. The SoC is equipped with on-die memory to avoid the expensive data movement cost letting the chip achieve outstanding low energy characteristics together with high throughput and low latency.
Lastly, it is worth noting that also Meta has very recently moved to develop their own AI inference accelerator, called MTIA v1[2]. The accelerator is specifically designed with a hardware-software co-design process to be tailored specifically for the workloads of DLRMs running in their data centers. Consisting of a mesh of interconnected processing elements, together with on-chip and off-chip memory resources, they aim to obtain a good performance-per-watt (TFLOPS/W) for models of any scale.

[2]Meta MTIAv1

AI Accelerators and DLRMs inference We have seen that AI programs offer great opportunities for acceleration as they are massively parallelizable and make use of a limited set of operations. However, we recall that DLRMs have unique inference characteristics, and performance is dominated by the large number of random off-chip memory accesses made to retrieve small embedding vectors, regardless of which architecture we use.

To understand which accelerators are more suitable for boosting the performance of DLRMs, a few high-level requirements can be made. Firstly, chips must be equipped with enough on-chip memory, essential for the computational units to have access to data without too expensive data movements, both in terms of energy consumed and latency. Secondly, accelerators must be equipped with HBM with as many interfaces as possible for parallelizing the look-ups. Lastly, due to the incredible diversity of size of the models, ranging from megabytes to hundreds of gigabytes, there must be support for inter-chip and inter-device communication to unlock the possibility of building solutions that can scale out effectively.

2.3 Huawei Ascend Computing

Among many other domains, Huawei also specializes in artificial intelligence computing technologies. Motivated by the huge global demand for running AI workloads in various scenarios, ranging from edge computing such as mobile or automotive to high-performant data centers, they introduced the Ascend Computing series. It comprises several technologies, including various AI SoCs, boards, servers, and a highly-integrated software stack to accommodate the heterogeneous landscape.

2.3.1 Huawei Da Vinci architecture

To meet the needs of AI heterogeneous workloads, Huawei proposed an innovative domain-specific architecture, Da Vinci ([17]), thought specifically for efficiently running AI applications and algorithms.
The Da Vinci AI core's idea builds on using three basic computational blocks, and a self-explanatory representation is reported in figure 2.3.

- First, a one-dimensional, full-flexible scalar unit can execute any kind of operation.
- Second, a two-dimensional vector unit is capable of executing operations on vectors efficiently, such as computing non-linear activation functions.
- A three-dimensional matrix unit is crucial to computing in parallel multiply-and-accumulate (MAC) operations, typical of matrix multiplications. Since

AI programs involve a lot of tensor operations, this is a core component to accelerate their performance.

As can be seen from table 2.1, the computation intensity of the three units is radically different, ranging from a parallelism of 16 and 256 (16^2) fp16 elements for the scalar and vector units respectively, up to 4096 (16^3) for the cube one.

Figure 2.4 shows the high-level architecture of the core and its possible supported data flow patterns. Basically, the core consists of compute, storage, and control units. First, it is important to stress how the memory units are organized and the data flow that characterizes the execution of different kinds of instructions. A self-explanatory diagram of the storage access relationships is reported in figure 2.5.

- A first relatively large buffer of 1 MB, called **L1 Buffer**, is a software-controlled memory (scratchpad memory) and can serve as a temporary fast storage close to the computation units. Its typical usage is to host the source tensors currently being multiplied by the cube unit.
- Secondly, a smaller buffer of 256 KB, called **Unified Buffer**, is positioned very close to the vector unit so that it can access the source operands and store the result much faster.
- Moreover, there are also three additional buffers called L0A, L0B, and L0C to store the two current source operands of the cube unit and the result of its computation respectively. Lastly, there is also a scalar buffer made of registers, which are transparent for the programmer.

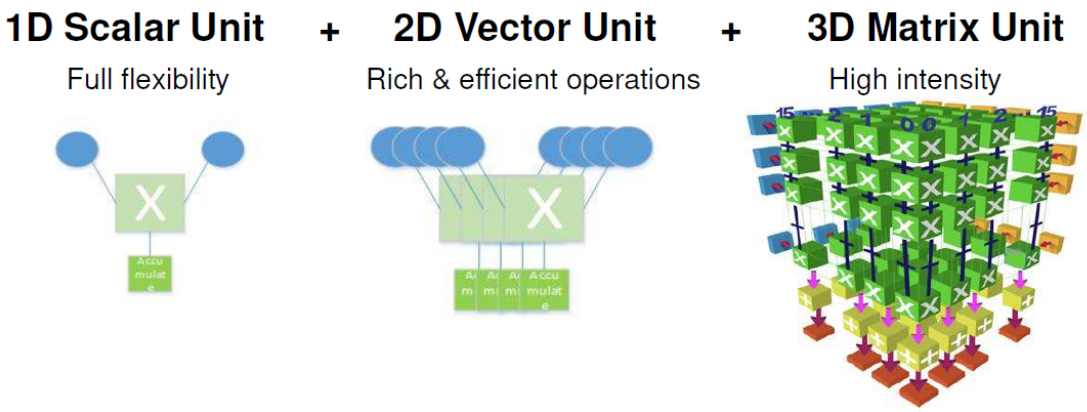

Figure 2.3: Huawei Da Vinci basic computational blocks. The image is taken from [17].

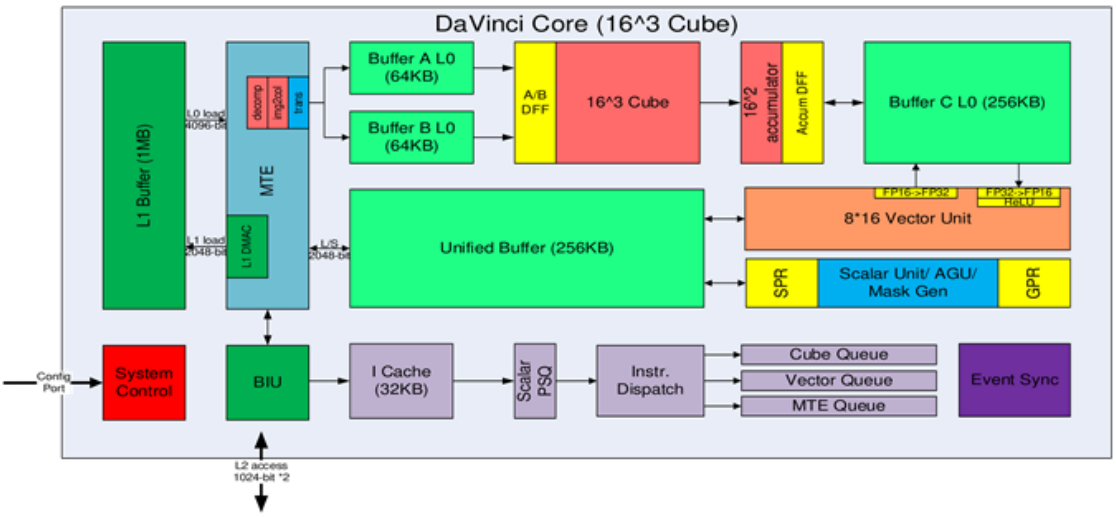

Figure 2.4: Da Vinci AI core

To manage the data transfers between the different memories, there is a DMA (Direct Memory Access) unit, called Memory Transfer Engine (MTE), which can be programmed to do the transfers asynchronously and thus in parallel to the computation units. More specifically, since data movements can be done in parallel as they include different independent memories, there are three different MTE units that handle the transfers between different memories. In particular, one handles data transfers between the different on-chip memories in the core, whereas the other two manage the transfers between on-chip and off-chip memories, considering both directions. To control the storage and computational units, a control system based on independent queues, called pipes, allows leveraging the parallelism of the available units. In particular, each instruction is decoded and then dispatched to one of five pipes, among which two are used for instructions relative to the vector

Unit	Parallelism
Scalar	16
Vector	256
Mac	8192

Table 2.1: The table shows the parallelism of the three computational units of the Ascend Da Vinci Architecture ([17]). The parallelism is expressed as the maximum number of bytes processable by the unit in parallel. The vector unit can process up to 2048bit INT8/FP16/FP32 vector with special functions, while the mac unit can process 4096 fp16 MACs + 8192 int8 MACs.

and cube units, respectively, and three for the data transfer instructions related to the three MTE units described previously. It is worth stressing that each pipe executes the instructions sequentially but is fully independent and parallel to the others. Lastly, an instruction cache of 32 KB is also available inside each core so that prefetching is possible, thus minimizing the times in which instructions must be fetched from off-chip memory. As a result, it is evident that the different units introduce data dependencies and, thus, must be synchronized. To this extent, a set of flags and a barrier are available to the programmers both for intra- and inter-pipe synchronization.

2.3.2 Ascend AI accelerators

As we have just seen, the Da Vinci core architecture is powerful and flexible, offering big opportunities to be integrated into a multi-core AI SoC. Thus, the Ascend series includes a set of SoCs specifically designed for different scenarios and requirements. On the one hand, they propose solutions for edge devices, such as the *Mobile SoC* and the *Automotive SoC*. On the other hand, they offer high-performance solutions with the *Ascend 310*, and the *Ascend 910*, designed for inference and training use cases, respectively.

While the details are out of the scope of this manuscript, a rough overview of the most important components of the two SoCs for inference and training is reported. Figures 2.6 and 2.7 show the schematics of the Ascend 310 and 910. The two SoCs have 2 and 32 Da Vinci cores respectively, and, more importantly, they are equipped with an on-chip L2 cache of 8 MB and 32 MB, shared by all the cores. Moreover, there are two DDR4 channels to add sufficient off-chip memory, which is essential for storing large quantities of data. Critically, Ascend 910 is equipped with on-chip HBM, which makes huge differences both in terms of bandwidth and

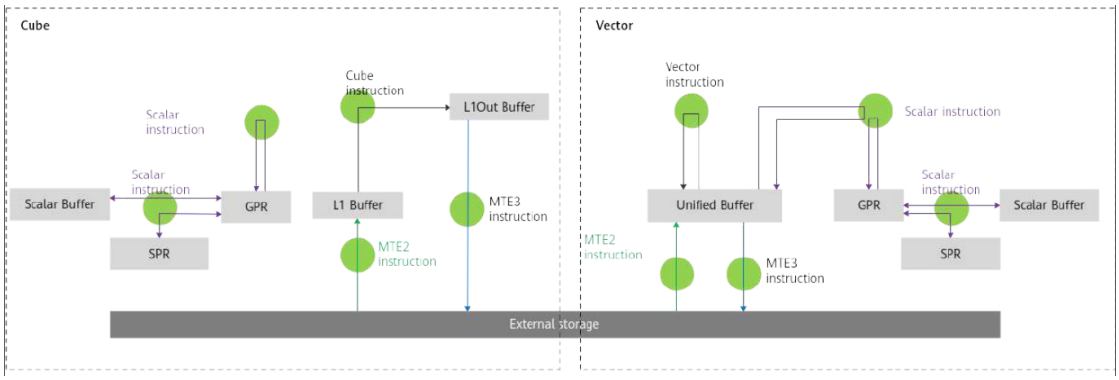

Figure 2.5: Storage access relationships between the memory components both for the Cube unit instructions (left) and the Vector unit ones (right).

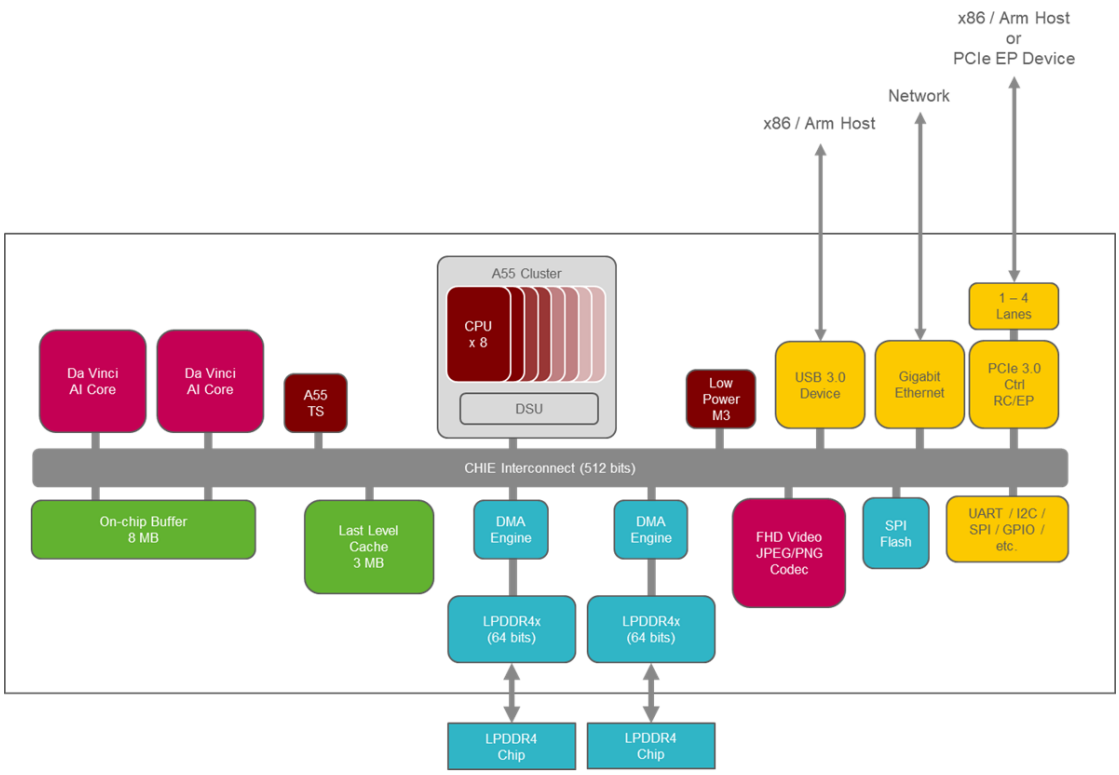

Figure 2.6: Ascend310, an AI Processor (SoC) specifically designed for accelerating AI inference.

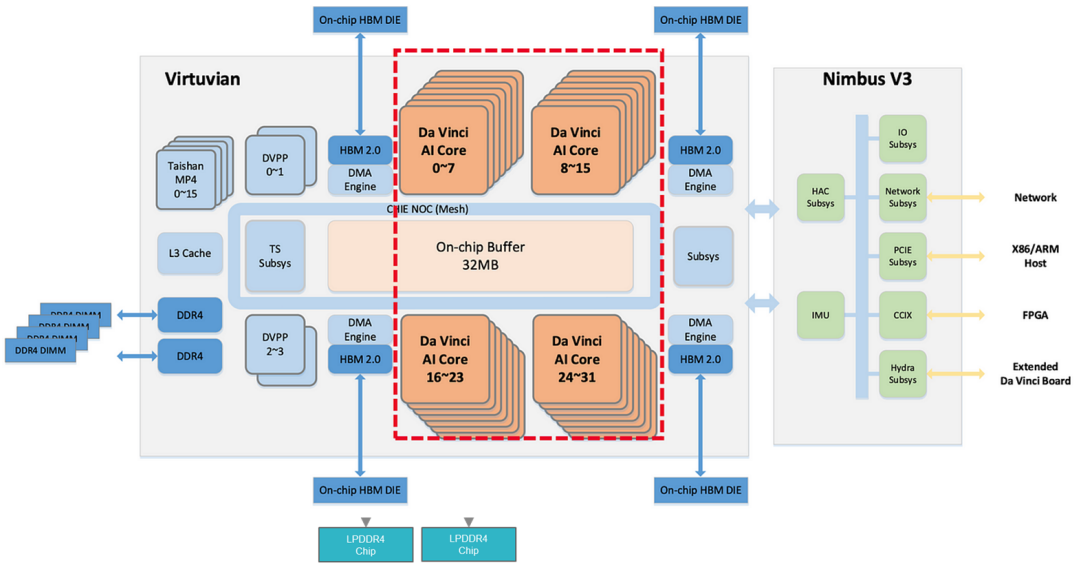

Figure 2.7: Ascend910, an AI Processor (SoC) specifically designed for accelerating AI training in data centers.

number of channels compared to DDR4 memories. Lastly, it is worth noting that there is a Gigabit Ethernet interface for enabling network connections, thus offering the possibility of building solutions that can scale out effectively.

Atlas devices To tackle different scenarios and use cases, Huawei has proposed a series of products called Atlas, whose devices are built on top of the two described SoCs. More specifically, different products integrate onboard specific quantities of memory, together with different numbers of AI SoCs mounted and interconnected in a specific way. As a result, there are accelerator modules, cards, and development kits, to mention a few. Besides, the Atlas series also includes comprehensive AI servers for training and inference scenarios, which can be put together in a cluster to be installed in high-performance data centers. Figure 2.8 shows a comprehensive picture of the Atlas series.

2.3.3 Ascend software stack

Concerning AI hardware accelerators, designing a good software stack is a crucial step to obtaining acceleration effectiveness. Specifically, Huawei has released a complex software stack made of several components, which allows AI applications to run on Atlas devices effortlessly and efficiently. To this extent, Huawei has introduced CANN (Compute Architecture for Neural Networks), a rich set of components and tools that expose hierarchical APIs to streamline the development of AI applications on the Ascend accelerators.

ACL CANN comes with a unified programming language called ACL (Ascend Computing Language), which allows writing AI applications to be executed onto Ascend hardware. Among many other things, ACL allows one to fully manage the runtime, ranging from device and stream management to model and operator loading and execution. Specifically, a stream is an abstraction created to make the parallel execution of different devices possible.

More importantly, ACL supports two working modes for running programs onto Ascend accelerators. As shown in figure 2.9, the first working mode consists in converting one single operator, written with programming abstractions that will be discussed in a moment, and then executing the converted operator directly onto Ascend hardware. The other mode, instead, converts an entire model and then executes its inference onto the accelerator. Moreover, we can see that both are based on a common format, called offline model (.om), which is a proprietary format to format binaries that can run on Ascend accelerators.

Figure 2.8: Huawei Atlas devices for different needs and scenarios, built leveraging Ascend AI SoCs.

Figure 2.9: ACL standard API call flow including two working modes. One is based on executing single operators, whereas the other executes entire model inferences.

ATC - Ascend Tensor Compiler Undoubtedly, the most crucial component of the Ascend software stack is ATC (Ascend Tensor Compiler), a sophisticated compiler that is capable of converting high-level AI programs into offline models, that is, executable binaries for the Ascend accelerators. Moreover, ATC is highly integrated with major deep learning frameworks such as TensorFlow and PyTorch, offering the possibility of seamlessly converting models written with these frameworks into offline models directly executable on Ascend accelerator.

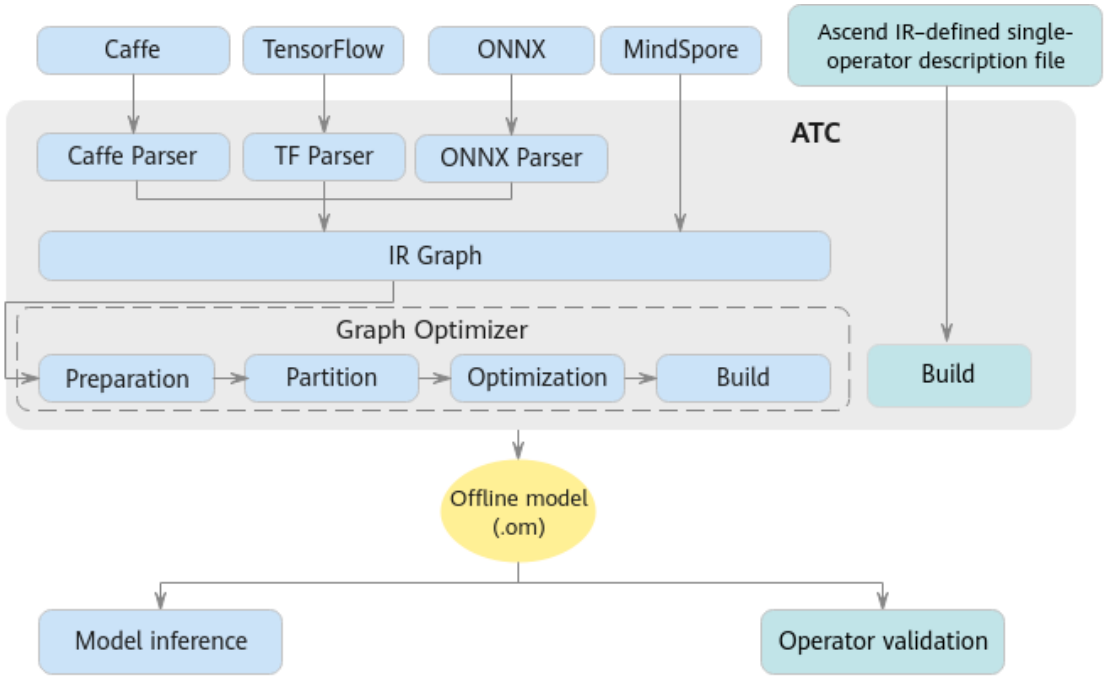

Figure 2.10: ATC compiler flow of operations to generate an executable offline model. On the left, we see the various optimizations that ATC applies when converting a model from a framework such as TensorFlow. On the right, a single operator, described in a proprietary IR, is instead converted.

This is accomplished by mapping each layer of the original model into one or more CANN operators, which are the basic kernels that are executed on the accelerators. Moreover, CANN comes with a rich library of ready-made built-in operators, implemented and highly optimized by expert developers working at Huawei.

Notice that lowering models or single operators effectively onto executables for the available hardware is not an easy task due to the many possible ways of representing a complex computation, the number of cores available, the amount and types of memory available given the memory required by the program, and many more. To this extent, as shown by figure 2.10, ATC applies extensive optimizations to the whole execution graph and to the single CANN operators. Specifically, optimizations include operator scheduling, weight data rearrangement, data movement, memory optimizations, and model-level optimizations such as graph partitioning and operator fusion.

Custom operators and programming abstractions The library of built-in operators can be extended with custom operators, which can be written using

sophisticated and effective programming abstractions. Writing custom CANN operators has been made possible thanks to CANN TBE (Tensor Boost Engine), which by exposing high-level APIs, enables custom operator development. In particular, TBE exposes two different kinds of APIs for writing custom operators, each offering a different control granularity.

- TBE DSL (Domain Specific Language), offers high-level encapsulated primitives and automatic tiling, synchronization, and scheduling, at the cost of delivering lower performance. For instance, typical usable primitives are tensors over which high-level operations can be applied. Consequently, this level of abstraction does not offer much control over the memory or the order of execution of the operations.
- TBE TIK (Tensor Iterator Kernel) is a dynamic programming framework based on Python. It exposes APIs to have complete control of the data flow and thus lets programmers focus on data movements and computation while benefiting from automatic synchronization between the core units. Moreover, it does not provide automatic tiling, giving the programmer the responsibility to do it, which leads to more efficient programs at the cost of requiring expertise and knowledge of the underlying hardware.

2.4 Data flow

Since the method proposed in this manuscript focuses deeply on designing an effective data flow for hardware accelerators, before describing in detail the design choices, it is essential to introduce some fundamental principles. The term data flow refers to the description of the whole path that data follows in a computation. More precisely, it describes how data are passed from one hardware component to another, including all the stages from the source operands to the actual results. Therefore, the problem of accelerating applications using custom hardware accelerators can be seen as finding the optimal data flow which maximizes hardware utilization while minimizing latency.

A good model for describing data flows is called *loop nest*, and consists in describing the flow as a set of nested loops that process and move data between different memory units. Given a loop, if the data of its iterations are processed in different time intervals onto the same hardware components, we classify it as a temporal loop. Instead, if the data are processed in the same time interval but on different hardware components (i.e., in parallel), we call it a spatial loop. As a result, a data flow can be expressed easily in the loop nest format as a set of nested loops, each being either spatial or temporal.

Since the memory buffers have limited size, it is evident that if a computation involves data larger than the available buffers, the processing must be split, and data must be moved between different memory units, potentially many times. Thus, the design of effective data flows is strongly related to two principles of localities. First, we refer to temporal reuse if the same consumer uses the same data more than once. Second, if more than one consumer uses the same data at different spatial locations, we refer to it as spatial reuse.

Chapter 3

Related works

Optimizing the inference of DRLMs in production-scale scenarios is still an open area of research. In fact, due to the relevance of such models in today's data centers, the research community has been investing a lot of resources for years to find suitable solutions. The difficulties lie in the intrinsic complexity of DLRMs, which present several challenges that make optimization strategies work only for specific use cases. To clarify what the challenges are and the different strategies that can be put in place to overcome them, we characterize the DLRMs in terms of the following features:

1. Size of the model, which ranges from a few MBs to hundreds of GBs since we have tens or even hundreds of embedding tables, ranging from very small to very big.

2. Constraints on worst-case latency, which hold for almost every real-time system as they are usually released under SLA requirements.

3. Energy consumption, which evaluates the throughput with respect to the consumed energy. In fact, as underlined in [16], a better performance metric is *performance per watt*, as improving the performance of a few percentage points at the cost of exponentially increasing the power requirements might not be ideal.

4. Scaling capabilities, which is very important as the requested traffic to recommendation systems is increasing limitlessly every day, not to mention the ever-increasing size of DLRMs.

Thus, it is evident that the optimization problem has many dimensions which make deployment scenarios substantially different from one another. As a result, a plethora of works has been developed in the last decade to address, or at least mitigate, some of the major issues affecting such complicated systems. It is worth

noting that a critical vision of such approaches is essential as they try to address specific use cases, thus being, most of the time, neither generalizable nor applicable in other scenarios. Moreover, some works do not consider all the dimensions but instead purposefully ignore some, such as energy consumption, restricting their applicability to use cases without strict power constraints.

The existing works tackle the challenges from different angles and use different approaches. First, a line of works tries to exploit system-level optimizations using CPUs, GPUs, FPGAs, or an ensemble of them interconnected with a proper network topology [18, 19, 20, 21]. Second, other works have invested a lot of time and resources in designing custom hardware, for example, integrating many computational units near the memory by designing custom DIMMs, which is commonly identified as *Near-Memory Processing (NMP)* ([22, 2, 23]. Other approaches, such as the one proposed by Esperanto [16], try instead to build an entirely innovative SoC for accelerating recommendation systems and other major AI applications.
However, notice that the approaches quickly introduced so far are not directly related and comparable with the approach presented in this thesis; instead, they are orthogonal solutions which are important to mention to provide a comprehensive picture of the different methodologies.

Besides, another line of work focuses instead on something more comparable to the approach presented in this manuscript, consisting in optimizing the *data flow* achieved onto existing hardware. In particular, they focus mainly on partitioning the workload evenly across the available resources or optimizing the caching policies to obtain better performance for the embedding look-ups.
Moreover, some works try to model and learn the characteristics of the query distribution, leveraging this information to provide major improvements through optimized access-aware storage.
For clarity, the different categories of approaches are described in different sections to distinguish clearly the level upon which the methods build.

3.1 System-level optimizations

To better understand the motivations behind the development of the works, we further distinguish approaches that leverage only one node from approaches that instead try to ensemble multiple, possibly heterogeneous nodes, to better tackle the architectural challenges imposed by DLRMs.

Single node optimizations Several works build on the idea of exploiting one FPGA tailored specifically to mitigate or delete the memory access rate bottleneck

introduced by the large number of off-chip random memory accesses made by embedding layers. A representative example is MicroRec [21], which leveraging one FPGA introduces two major optimizations. First, they propose redesigning the data structures used to memorize the embedding tables to reduce the number of look-ups needed. For instance, instead of storing two embedding tables separately, the cartesian product between the two is computed and stored, that is, for each row of the first table, all the couples made by the row and all the rows of the second table are stored. However, this approach is applicable only to couples of small tables, as the size of cartesian products quickly explodes. Even though the idea seems promising and could have been integrated into the proposed method, it increases a lot the storage requirements, resulting in a few significant motivations of why the cost paid is not worth it. On the one hand, even considering small tables, it is usually a better idea to store them in a memory close to the processing unit to avoid the expensive cost of moving many small vectors, both in terms of energy consumed and latency. On the other hand, wasting precious space of fast memory, such as HBM, which has limited capacity, to store cartesian products, is not optimal given that potentially larger tables would benefit from being stored there.

Second, they take advantage of an HBM to perform as many parallel look-ups from different interfaces as possible. Additionally, they carefully design a deeply pipelined data flow for obtaining low-latency inference. Thus, they obtain significant improvements with respect to a standard 16-CPUs baseline showing that CPUs are affected by high latency (order of milliseconds) compared to the FPGA-based works, which remain in the order of microseconds. However, they rely heavily on using the most innovative HBM technology to store the embedding tables, thus restricting the addressable use cases to models which fit in HBM, whose size is limited to tens of gigabytes. Moreover, they consume a lot of the FPGA resources to eliminate the memory access rate bottleneck moving the limiting factor to the computation of the rest of the model, which is starving for resources and thus becomes a new bottleneck for the performance.

CPU-based and GPU-based inference Before moving to analyze the ensembling of different architectures and strategies, it is worth spending some time to analyze the two standard architectures, CPUs and GPUs, readily available in every existing data center.

CPUs-based approaches consist in storing the embedding tables in DRAM or even in SSDs, and then performing the lookups from there. This reflects the possibility of having much flexibility in storing potentially huge models. However, CPUs are limited SIMD architectures, consisting only of a handful of general-purpose

computational cores, thus being inefficient and having limited throughput for this kind of workload. Even worse, these systems present the memory access rate bottleneck, as underlined in [2, 22].

Instead, a more suitable design is implemented in GPUs, which leverage thousands of identical cores based on the single-instruction-multiple-data (SIMD) paradigm, benefiting from very large batch sizes for increasing the achieved throughput. Unfortunately, as underlined in [18], GPUs present even worse latency compared to CPUs, not to mention the gigantic number of random memory accesses that are not suitable for the SIMD paradigm. To worsen the situation, GPUs have limited on-chip memory, resulting in being obliged to do random memory accesses from off-chip memory.

3.1.1 Multiple nodes / cluster optimizations

Having defined the pros and cons of CPUs, GPUs, and FPGAs, many works have focused on building a heterogeneous ensemble of such technologies to exploit the advantages of each technology while mitigating its problems with the others. One example is FleetRec [20], a high-performance and scalable recommendation inference system with tight latency constraints. They argue that the main challenges lie in the memory access rate bottleneck, the big size of DLRMs, and latency constraints, aligning their analysis with the one in this manuscript. Nonetheless, they reinforce the motivation for their approach by arguing that using one accelerator, such as FPGAs or GPUs, is not applicable for the limited memory capacity of these devices. Thus, their idea consists of building a heterogeneous computing cluster of several nodes of different types. In particular, they exploit (i) FPGAs equipped with cutting-edge HBM for performing fast look-ups, (ii) CPUs with large DRAM capacity for storing a few large embedding tables, and (iii) GPUs exclusively for the computation part of the DNN.

The nodes are then interconnected through a high-speed network (100 Gbps) for moving data between them. As a result, the approach can scale out to support any model size by simply adding more nodes to the cluster. Other works build on similar ideas, for example, the approach described in [19] focuses on building a homogeneous cluster of multiple FPGAs to avoid leaving the computation part in starvation for resources, as we have previously mentioned.

Even though the overall architecture is quite effective in mitigating the bottleneck introduced by the embedding look-ups, there are also some drawbacks. First, despite the concept being simple in principle, its implementation is quite complicated. In fact, coordinating the data flow of the whole cluster is a challenging task and must be carefully designed to be efficient. Second, it is important to consider that

moving data between units is orders of magnitude more expensive than the actual computation, both in terms of latency and consumed energy. This motivates even more the development of optimizations from a different perspective, such as the one presented in this manuscript.

3.1.2 Other solutions with custom hardware

It should now be clear that the memory bottleneck is the first tough challenge to solve. To address the issue, some works focused on building Near-Memory Processing (NMP) solutions. They integrate as many computational units as possible near the memory chips where the tables are stored while supporting only the minimum set of operations required for the computation. Representative examples are RecNMP [22], and TensorDIMM [2], which build a DRAM-compliant near-memory processing solution tailored for mitigating the memory access rate bottleneck introduced by the look-ups.

Lastly, a very promising approach is proposed by Esperanto Technologies, who developed an AI accelerator called ET-SoC-1 [16] with innovative characteristics suitable for DLRMs. In particular, they leverage 1000 low-power RISC-V processors on a single chip and an on-die memory system with many interfaces. The concept is relatively simple and consists in performing parallel look-ups through as many interfaces as possible while exploiting a few cores for the remaining processing. Moreover, they group the cores in a smart way to turn them off and design an interconnection between core groups and chips. In this way, they achieve massive throughput values while still consuming less energy than other solutions.

3.2 Dataflow optimizations

Designing custom hardware solutions requires years to go in production and involves enormous costs for companies. Even worse, those solutions try to optimize a few applications as much as possible, resulting in limited flexibility in supporting other applications effectively. Moreover, as we have already mentioned, DLRMs diversify a lot one from another. Besides, the requested traffic that such systems must handle is changing every day, and it is difficult to design solutions that scale up or scale out effectively and transparently. Lastly, as research advance at a fast pace, new architectures for building more accurate and efficient models are in constant development, thus making those less flexible custom solutions a risky move for the companies. These are just a few reasons that have motivated the research of new optimizations that are more flexible and easy to implement on existing architectures.

These solutions try to address the bottleneck introduced by the many random memory accesses performed by the look-up operations while taking into consideration the scaling factor of the model size. To this extent, different lines of work focused mainly on modifying the caching systems or partitioning the workload across the available hardware resources. Since these approaches align with the characteristics of the method proposed in this manuscript, a few major works are analyzed.

3.2.1 Cache-storage level optimization

The memory hierarchy typically implemented in every system is usually stratified. It starts with a slow, large storage space such as HDDs or SSDs (order of TBs), proceeding to a smaller but faster memory, called primary memory (order of GBs), and lasting in the so-called N-layered cache system. Thus, the cache system can have different levels, and its functionality is based on some basic locality principles. Spatial locality refers to the execution pattern of accessing data that are confined in terms of storage locations, whereas time locality refers to the trend of accessing the same data in the same time window. These principles usually lead to choosing a cache replacement policy of type LRU (Least-Recently Used), which consists in replacing the least used cached data with the newly accessed one. Despite these principles effectively characterize the access patterns of most executing programs, the unique and random access patterns of the embedding look-ups invalidate such principles' effectiveness, thus making the cache system ineffective when performing embedding lookup operations.

Motivated by the ineffectiveness of the standard cache replacement policy, EVStore [24] proposes an ad-hoc replacement policy tailored for the characteristics of the embedding look-ups. First, they argue that embedding tables are tripling their size every two years (order of TBs), and thus, more and more frequently, they must be stored in HDDs or SSDs.
Secondly, they show a critical property characterizing the access patterns of embedding look-ups regardless of the underlying platform. In particular, they call such property *all or nothing*, referring to the fact that given a group of indices for looking up different tables, even if only one of them does not hit the cache, the overall performance is significantly degraded. Therefore, they introduce an interesting metric called *groupability* of a key (i.e., an index) that encodes how much a key is groupable (i.e., found in the same query) with the others.

Motivated by the above properties, they design an abstract caching system made of three components or layers. (i) L1 Cache, which implements a policy that tries to maximize the times in which a perfect hit (i.e., all the indices in

the query hit the cache) happens. (ii) EVMix, which delegates some space from L1 into an L2 segment storing different lower-precision versions of the vectors. The feature is called mixed precision caching, an interesting approach that stores, instead of float32 words, reduced precision versions made of 16, 8, and even 4 bits. (iii) EVProx (L3), which tries to exploit the similarities of embedding vectors, is called surrogate vectors, which are likely found in the L1 or L2 cache.
Overall, they claim to be capable of reducing storage usage by up to 94% at minor accuracy costs.

Even though the approach presents advantages, a few negative considerations must be made. Firstly, EVMix and EVProx propose to approximate embedding vectors of floating point numbers by reducing the precision or by picking similar values, claiming that the worsening of accuracy is limited at 0.2%. Even if assuming that the declared upper bound of performance worsening applies to all the considered models and deployment scenarios, we argue that this solution is unacceptable in a large production-scale DLRM. The reason is that the potential income derived from such models has gigantic volumes, excluding the possibility of sacrificing so much accuracy on the provided personalized recommendations. However, a very good feature introduced by the approach is the L1 cache replacement policy, based on a novel groupability score of the keys. Thus, the feature could be integrated into other optimization strategies, such as the one presented in this manuscript. However, we leave this approach as a possible future improvement.

3.2.2 Load balancing: table sharding optimizations

Another interesting approach exploited by other works focuses instead on partitioning the workload associated with the embedding look-ups. The problem of partitioning the tables across multiple groups is called *table sharding* and has a long research history as it characterizes several other issues in the literature, such as database sharding [25], and blockchain sharding [26]. Even though existing works apply sharding strategies to the embedding tables of DLRMs only for optimizing the training stage, the problem is precisely equivalent when optimizing the inference phase. Moreover, to the best of our knowledge, little to no work has been proposed to optimize the inference of such models exploiting sharding strategies, thus being the first time such an approach has been utilized for this goal. For the sake of clarity, notice that the mentioned works build on the hypothesis that the embedding layers have been previously trained so that the sharding problem during training makes sense.

The sharding problem consists in providing a balanced, thus efficient partitioning of all the tables across the available compute nodes of the system, aiming at

providing the most balanced execution. Intuitively, the slowest node limits the performance, thus explaining the correlation between having a balanced execution and a performant one. Even though the proposed strategies focus on the training part, both the training and inference stages of DLRMs inherit the bottleneck coming from the embedding lookup operations. However, the sharding or partitioning problem is known to be NP-Hard[1], thus making the task complicated to solve.

An interesting existing approach, called AutoShard [27], exploits deep learning capabilities to solve the problem. Even though the work focuses on optimizing the training stage, its approach can also be applied to the inference one. Moreover, it gives a few important takeaways which align perfectly with the problems faced by the method presented in this thesis.

- First, it is hard to efficiently and precisely measure the cost of partitioning. This is because running on the device every possible set of values for the varying parameters, such as the number of tables and their characteristics, is unfeasible as it would require too much time, not to mention the waste of energy only for measuring a huge number of configurations.

- Second, the cost introduced by the execution of a single table cannot be summed to obtain the cost of execution of multiple tables due to batching, parallelism, and operator fusion, which make the execution of different tables overlapping and highly dependent on one another.

As a result, to tackle the sharding problem, they propose to use a deep reinforcement learning method supported by a standard DNN to learn a performance model to predict the multi-table cost effectively. More precisely, they formalize the problem of partitioning as a Markov Decision Process[2], in which in a single step, one table is assigned to a group. Thus, they exploit a deep reinforcement learning algorithm to learn an optimal policy based on the rewards corresponding to the cost estimated by the performance model. However, training a cost model in such a way requires a decent amount of samples executed on hardware, which might not be the best choice in all contexts. The learned performance model is highly dependent on the underlying parameters, including the input query distribution, the hardware utilized, and the actual configurations of tables included in the training set. Moreover, the deep reinforcement approach takes time and resources, thus not being directly usable in all the scenarios, especially in fast-changing use cases. Other works are based on a very similar approach, such as DreamShard [28] proposed by Meta, and

[1]Partition problem

[2]Markov Decision Process

others [29, 30].

Besides, the authors of [31] underline the extremely unbalanced access patterns of embedding look-ups, stressing that a few entries are accessed up to 10000x more times than the others. As a result, they propose to put only the *"hot rows"* in the on-chip memory of GPUs. However, notice that this approach is strongly application-dependent and requires a lot of effort in gathering access patterns from models already in production, not to mention that the distribution is constantly changing over time.

Chapter 4

Accelerating embedding lookups on Ascend AI accelerators

4.1 Problem characterization

As we have already discussed in section 2.1.1, almost every existing DLRM extensively uses embedding layers to extract powerful representations used to solve tasks such as Learning to Rank effectively. We recall that an Embedding layer maps categorical features, represented as one-hot encoded vectors, to a dense embedding space of fixed dimensionality. To this extent, at inference time, it is employed a pre-computed table of embedding vectors, one vector per row, each corresponding to a categorical value of the original feature.

Pooling-based embedding look-ups To be more precise, to achieve more expressive representations, embedding layers offer the possibility of handling sequences of categorical values instead of just one. Given a categorical feature as input, we do not have a single categorical value but rather a sequence of values, which means that at inference time, we will have a sequence of integer indices and, thus, a sequence of lookups to be performed on the same embedding table. As a result, the output of the embedding layer will be a sequence of dense embedding vectors for each sample in the batch, which are then pooled together to obtain a single dense embedding vector. This structure defines DLRMs as *pooling-based* DLRMs, the most widespread in today's data centers. The pooling operation is typically a sum of the embedding vectors, but also other options, such as calculating

the maximum, the minimum, or the average, are possible.
To provide a common notation for the rest of the manuscript, given a set of N categorical features as input, we define a set $C = \{c_0, \ldots, c_{N-1}\}$ of categorical features. Each categorical feature c_i has its own sequence length, which is denoted as s_i, and thus we denote the set of all the sequence lengths as $S = \{s_0, \ldots, s_{N-1}\}$. Moreover, each categorical feature c_i can assume m_i possible values; therefore, we define for convenience another set of the number of possible values of the categorical features as $M = \{m_0, \ldots, m_{N-1}\}$.
For the sake of clarity and to provide a more explicit meaning to the sequence length, we take as a small example a model with two categorical features referring to the characteristics and interactions of a person with an e-commerce service. The first feature is the sex of the person and can take only two possible values. In this case, the sequence length is one, as it would be meaningless otherwise. The second feature is the sequence of the last ten bought items, where each item is referred to with a numeric identifier. In this case, the number of possible values is given by the total number of existing items in the e-commerce database, which we hypothesize to be 10^6, and the sequence length is then set to 10. The last example also gives a practical sense of why embedding tables are so gigantic, as the number of possible items in a database could be huge. To express the small example in symbols, we would have the number of categorical features $N = 2$, then the set of possible values would be $M = \{2, 10^6\}$, and lastly, the set of sequence lengths $S = \{1, 10\}$. Notice that the usefulness of the sequence length in the case of the last ten items bought could be the possibility of aggregating multiple pieces of information instead of just relying on single, independent features.
Given the previously defined sets, we define explicitly how the embedding look-ups are built and used at inference time. For simplicity, we fix the embedding dimensionality to the same value for all the embedding layers, and we refer to this value as E. For each categorical feature c_i with m_i possible values, we build a pre-computed embedding table containing m_i rows, one for each categorical value, each made of E elements. We end up with a set of embedding tables denoted as $T = \{t_0, \ldots, t_{N-1}\}$. Each categorical feature c_i maps its m_i possible values (categorical values) to the corresponding index to access the correct row of the related table.
We denote the input indices at inference time, usually called input query list, as $Q = \{q_0, \ldots, q_{N-1}\}$, where each q_i is a vector of s_i (sequence length) integer indices, and each index can take values in the range $[0, m_i - 1]$. Taking the previous small

example, we have:

$$
\begin{aligned}
&c_0 = \{\text{“}male\text{“}, \text{“}female\text{“}\} \\
&c_1 = \{id_0, id_1, \ldots, id_{10^6-1}\} \\
&s_0 = 1 \\
&s_1 = 10 \\
&E = 16 && \text{(for hypothesis)} \\
&t_0 \in \mathbf{R}^{2,16} && \text{(i.e., a table of 2 rows, each of 16 elements)} \\
&t_1 \in \mathbf{R}^{10^6,16} \\
&q_0 \in \mathbf{N}^1 && \text{(i.e. a vector of 1 integer index)} \\
&q_1 \in \mathbf{N}^{10}
\end{aligned}
$$

Therefore, for each query q_i made of s_i integer indices, we must perform s_i look-ups from table t_i using the indices to extract s_i rows. Lastly, a pooling function, such as the vector element-wise sum, is applied to the sequence of s_i embedding vectors to obtain only one embedding vector of E elements. In symbols, we have a query vector $q_i \in \mathbf{N}^{s_i}$, which accesses the table t_i to obtain a temporary tensor $P \in \mathbf{R}^{s_i,E}$, which is lastly reduced through a pooling function to a vector $v_i \in \mathbf{R}^E$, which corresponds to the output of the embedding layer. For the sake of clarity, notice that in the small example, we assumed the batch size to be equal to 1, but in a real case, we would have a query $q_i \in \mathbf{N}^{B,s_i}$, where B is the batch size, thus obtaining as the output of the embedding layer a vector $v_i \in \mathbf{R}^{B,E}$.

Memory access rate bottleneck As underlined by several works in the literature, such as [[1], [32], [23]], we recall that the inference performance of DLRMs is largely limited by the massive number of random memory accesses from off-chip memories for retrieving small embedding vectors from tables whose size ranges from small (a few entries) to huge (GBs or TBs). More specifically, off-chip memories are usually DRAMs in standard processing architectures such as CPUs, which are limited to only a handful of independent channels for reading data in parallel. For instance, supposing to have a CPU that is trying to retrieve embedding vectors from a DRAM with only two channels, the CPU will have to retrieve sequentially only two embedding vectors at a time, regardless of the channel bandwidth, also considering the overhead implied by DRAMs to set up memory accesses at random addresses. As a result, the overall performance is limited, and the high memory reading access rate represents the bottleneck. Motivated by this, the focus of the proposed method of this thesis is confined to optimizing the embedding lookup of the models extensively, as well as the subsequent pooling-based operation, as it offers great opportunities to be fused in one single operation. We argue that this can be done without losing too much generality, as a plethora of works, such as [31],

[33], [34], have already focused on designing smart optimizations for accelerating the computation part.

4.2 ATC-based embedding lookups

To evaluate and experiment with innovative strategies and data flows for boosting embedding lookups performance, we leveraged the Huawei Ascend AI accelerators with the goal of obtaining the fairest possible comparison scenario. In particular, we have focused on the Ascend 910 AI accelerator. The accelerator is equipped with cutting-edge on-chip HBM technology, thus being characterized by low latency memory accesses, high bandwidth, and highly parallel memory accesses thanks to the many reading channels of HBM. It has 32 Da Vinci cores, useful especially because even when having a memory that supports a high memory access rate, having more cores allows generating the high amount of memory addresses required for retrieving embedding vectors, which otherwise would become a limiting factor. As a result, the combination of on-chip HBM to perform fast parallel lookups, together with the high number of cores, makes it a perfect candidate to experiment with. More importantly, considering the already existing built-in strategy for performing look-ups on Ascend accelerators, based on relying on the advanced ATC compiler together with built-in operators written by experts, the presence of HBM makes the approach much more effective, thus making the development of better data flows more challenging, and thus creating a fairer comparison. We recall that the CANN suite has a rich library of highly optimized built-in operators written by experts and thought for almost every use case required by AI applications. Moreover, operators are designed to be flexible enough to let the compiler introduce numerous optimizations, such as searching for an optimized tiling policy or rearranging the memory layout. Not only is one operator extensively optimized, but the compiler also applies numerous graph-level optimizations, such as graph partitioning and optimizations, operators fusion, and many others, as already discussed in section 2.3.3. As a result, the multi-year expertise of the developers of built-in CANN operators, together with the well-tested and proven effectiveness of the ATC compiler, makes the design of a better solution a tough challenge.
More specifically, as underlined in section 2.3.3, ATC is highly integrated with high-level DL frameworks such as TensorFlow and PyTorch. For this reason, the standard approach for accelerating DLRMs inference is to convert a model written in those frameworks into an offline model that can be executed on accelerators. To this extent, ATC converts each layer of the original model into one or more CANN operators that produce as output the same result as the original layer. ATC converts all the embedding and pooling layers, written in a high-level framework

Unit	Bandwidth	Bandwidth Ratio
Execution Engine	2048 TB/s	1
L0 Memory	2048 TB/s	1/1
L1 Memory	200 TB/s	1/10
L2 Memory	20 TB/s	1/100
HBM	1 TB/s	1/2000

Table 4.1: The table shows the bandwidths of the different components and memories included in the Ascend Da Vinci architecture.

(such as PyTorch, TensorFlow, etc.), mapping each of them into a built-in CANN operator called GatherV2 and ReduceSumD, respectively. Unfortunately, it is impossible to derive which optimizations are applied by ATC and their motivation. For this reason, the binary that ATC generates from the model conversion is treated as a black box and taken as the baseline for evaluating the effectiveness of the proposed strategies.

4.3 Embedding lookups strategy design

As we have seen, DLRMs are limited by the very high memory access rate induced by embedding lookup operations. Thus, the design of the new strategies focuses on providing an optimized execution both for the look-up operations and the subsequent pooling reduction, which offers great opportunities to be fused in a unique operation. Before digging into the details of the design of the proposed strategy, a few considerations must be made. Firstly, as underlined in table 4.1, the different memory levels in the hierarchy are characterized by bandwidths that differ by orders of magnitude. In fact, relative to the internal bandwidth of the computation data path, we can see that the L1 memory is 10x slower, whereas the L2 cache is 100x slower. Even worse, the HBM memory, which plays a key role in looking up embedding tables, is 2000x slower with respect to the internal data path. Even though the bandwidth offers an intuitive idea of the performance of a specific memory, it is worth noting that making a lot of data reads of a few bytes involves additional overhead, including the data transfer request, additional data exchanged by the reading protocol, and many others. Thus, when considering the memory accesses performed by the embedding lookup operations, the actual throughput achieved is smaller than the one reported in the memory specifications. The scenario gets worse if we consider accessing huge tables from DDR4 memories, which is necessary as HBM is limited to tens of GB in the best case and whose number of channels is limited to two. To perform look-ups efficiently, we have seen that a number of works in the literature, including [18], and [20], try to put

together a cluster of FPGAs and assign one or a few of them to perform the lookups, leaving the others for the computation of the rest of the model. Unfortunately, these solutions do not apply well in all those scenarios where the tables do not fit in the HBM memory.
Secondly, as underlined in several works in the literature, such as [31] and [24], the query distributions in real-world use cases are far different than a uniform one, resulting in extremely irregular access patterns for the embedding look-ups, and thus resulting in only a tiny part of the embedding table being responsible for the majority of the accesses. Lastly, as we will see in detail when discussing experimental results in section 5, almost every existing hardware platform used for running DLRMs inference programs makes extensive use of a cache system, which presents hidden problems as emerged by preliminary experiments. More specifically, in a multi-core environment, typical in this scenario, when two or more cores try to access the same cache line, an access conflict is made, which must be solved by serializing the accesses made by the cores. This, together with the extreme irregularity of the access patterns, cause a huge performance problem, as we will see in the related chapter.

Design key ideas Having described the major problems that emerged from previous works and preliminary experiments, a few key ideas can be extracted and considered as a base to build a better strategy.

- First, an innovative solution could be persisting in advance (*preloading*) some of the embedding tables in the L1 memory of each core. Since the L1 memory is an on-chip high-speed buffer, exclusive inside each core, we argue that its supported maximum access rate is much higher than the one of the HBM, considering the small dimension of the embedding vectors to be retrieved. As a result, this could alleviate or even cancel the memory access rate bottleneck.

- Second, instead of doing individual lookup operations followed by a partial pooling function, the parallelism of the vector unit could be exploited to access in a vectorized way multiple rows from the Unified Buffer, to be later pooled together by the vector unit.

- Third, since embedding tables are extremely variable in terms of number and size, an inter-core chunking strategy might be effective. This implies an asymmetric usage of the available cores, which, to the best of our knowledge, has not yet been explored in other works.

- Lastly, since from preliminary experiments it has emerged that the performance of embedding lookups is extremely dependent on a wide number of parameters, a single strategy that always works better than others might not exist. To

this extent, a combination of different strategies supported by performance modeling could be more effective.

4.4 Symmetric inter-core lookup data flow

Since AI accelerators are most of the time multi-core processors, we first focus on designing a new data flow for Ascend accelerators based on the single-instruction-multiple-data (SIMD) paradigm, which is largely employed in the acceleration of deep learning applications. As a consequence, the code that will be executed on each core is the same. Without loss of generality, for the design of the data flow, we will take as reference the Huawei Ascend 910 accelerator characteristics, described in section 2.3.2. Referring to the schematic shown in figure 2.7, we recall that Ascend 910 is composed of 32 Da Vinci cores, together with two software-controlled buffers (scratchpad memories), called L1 and UB (unified buffer), which are exclusive for each core and large 1 MB and 256 KB respectively.

Motivated by the problems introduced in section 4.3, we propose four possible strategies for implementing the data flow of pooling-based embedding lookup operations, each exploiting the available on-chip buffers in a different way.

1. The first one consists in performing the lookups from the global memory sequentially, thus implying off-chip memory accesses with all the related problems already discussed. This is necessary as some tables are very big, thus, it might be more effective to do the lookups directly from there. For the rest of the manuscript, we will refer to this strategy as *GM* strategy.

2. The second one, instead, is based on the idea of persistently preloading an embedding table in the L1 buffer to perform the lookup operations directly from there. The motivation lies in the fact that the bandwidth of the L1 buffer is 10x and 200x larger than the ones of the L2 cache and HBM, respectively, as underlined in table 4.1. Moreover, since the L1 buffer is placed directly inside each core, the average latency of each read operation is much lower, thus resulting in higher actual throughput when considering the memory accesses of the lookup operations. For the rest of the manuscript, this strategy will be referred to as *L1* strategy.

3. The third and fourth strategies instead transfer chunks of tables in the UB and then perform a fast vectorized lookup operation from there through the vector unit. In particular, there are two alternatives; one refers to the case in which we have previously preloaded the embedding table in L1, and therefore the chunk movements are done from there, whereas in the opposite case, the

movements are done from the global memory. The strategies will be referred to as *L1-UB* and *GM-UB*, respectively.

In the rest of the section, we will describe the data flow design for the strategies just described. Even though the actual implementation of the strategies is realized using Python exploiting the TBE TIK APIs described in section 2.3.3, for the sake of clarity, we will present the data flow using a pseudo-code based on the nested loop representation described in section 2.4. In particular, we refer to *spatial for* loop when using multiple cores in parallel, that is, each iteration refers to the execution in a different core, and we refer to *temporal for* loops when iterating through the scalar unit over the time at runtime. Lastly, we refer simply to *for* loops when we implement loop unrolling in order to compute static values to avoid the additional overhead of computing them at runtime. Moreover, the pseudo-code is based on the symbolism introduced for pooling-based lookup operations, described in 4.1. Additionally, since the data flow involves processing a batch of samples in a multi-core accelerator, we define the batch size as B and the number of maximum usable cores as K.

4.4.1 Direct lookups data flow

We have seen that the first two strategies, GM and L1, are based on performing individual look-ups of embedding vectors sequentially, gathering them directly from global memory (off-chip memory) or the L1 buffer. Since the two strategies have most of the data flow in common, with the only difference represented by the size and bandwidth offered by the two types of memory, they are described together.

The pseudo-algorithm 1 shows the data flow relative to the pooling-based lookup operations relative to only one embedding table, using either the GM or L1 strategies. First, a set of different samples of the query batch are assigned to each available core through a simple tiling computation. Then, each core executes in parallel an outer loop, whose iterations define a buffer in the UB for storing the accumulated result. Thus, inside the outer loop, an inner loop is executed in which another buffer in UB is defined for storing the query indices to use. The query indices are then moved from global memory to the UB, and a final innermost loop is then executed. In particular, each iteration of the innermost loop uses all the indices to access the correct row in the embedding table. Moreover, each look-up operation is alternated with a vector sum reduction, which can be overlapped with double-buffering. Finally, the result is moved from the unified buffer to the global memory.

It is worth underlining a few possible design choices and optimizations.

1. First, since we have four loops, defining their optimal ordering plays a critical

Algorithm 1 GM/L1 symmetric direct lookups

$t_{gm-l1} \in \mathbf{R}^{m_i,E}$ ▷ table stored either in GM or L1
$q_{gm} \in \mathbf{N}^{B,s}$ ▷ batch of vector queries of s indices, stored in GM
$o_{gm} \in \mathbf{R}^{B,E}$ ▷ output of the kernel (i.e., pooled result of looked up rows)
spatial for each $core_{id} \in \{0, \ldots, K-1\}$ **do**
  $B_{core_{id}} \leftarrow compute\_core\_samples(B, K, \ldots)$
  $B_o, B_q \leftarrow compute\_core\_tiling(B_{core_{id}}, \ldots)$
  $o_{tile\_cnt} \leftarrow ceil(B_{core_{id}}/B_o)$
  for each $tile_o \in \{0, \ldots, o_{tile\_cnt}-1\}$ **do** ▷ outer tile for the result
    $o_{ub} \in \mathbf{R}^{B_o,E}$ ▷ buffer for the result in UB
    $q_{tile\_cnt} \leftarrow ceil(B_o/B_q)$
    for each $tile_q \in \{0, \ldots, q_{tile\_cnt}-1\}$ **do** ▷ tile for the query
      $q_{ub} \in \mathbf{N}^{B_q,s}$ ▷ buffer for query indices in UB
      $q_{ub} \leftarrow q_{gm}$ ▷ data movement of query from gm to ub
      temporal for each $g_i \in \{0, \ldots, B_q-1\}$ **do** ▷ query samples iteration
        if $s > 1$ & $B_q > 1$ **then**
          ENABLE_DOUBLE_BUFFERING()
        end if
        $idx_{rel} \leftarrow B_q * tile_q + g_i$ ▷ index of current sample
        for each $z_j \in \{0, \ldots, s\}$ **do**
          $a_{ub} \in \mathbf{R}^E$ ▷ adder in UB
          $idx \leftarrow q_{ub}[g_i, z_j]$ ▷ index for accessing the row
          $a_{ub} \leftarrow t_{gm-l1}[idx, :]$ ▷ look-up of the row
          $o_{ub}[idx_{rel}, :] \leftarrow o_{ub}[idx_{rel}, :] + a_{ub}$ ▷ vector sum reduction
        end for
      end for
    end for
    $start \leftarrow tile_o * B_o$
    $end \leftarrow (tile_o + 1) * B_o$
    $o_{gm}[start : end, :] \leftarrow o_{ub}$ ▷ Move output to global memory
  end for
end for

role in how different pieces of data flow in the data path. In particular, there are two available options which consist in deciding whether to put as the outermost loop the query loading or the output buffer accumulation. In the first case, we favor an input stationary approach in which each query index is moved from off-chip to on-chip memory only once, at the expense of making many data movements of the accumulated output from on-chip to off-chip memory. In the other case, an output stationary approach is preferred, and many query data movements are thus necessary. It is worth noting that to leverage double buffering for overlapping data movements and vector pooling operations, only the second approach is feasible, thus being the preferred option.

2. Second, the tiling policy is based on a few empirically optimized hyper-parameters, which define a minimum size for the query tiles and derivate the tiles for the accumulation as a consequence.

3. Third, we choose to perform loop unrolling of the two outermost loops, plus the loop for the indices inside each sequence, in order to compute most of the scalar values at compile time, thus avoiding too much workload for the scalar unit at runtime. Empirically, from preliminary experiments, we saw that unrolling the loops of the output accumulator, query loading, and sequence length, contributes positively to decreasing considerably the scalar workload while maintaining the instruction cache hit ratio almost unaltered.

4. Lastly, double buffering on the look-ups loop is enabled only when the pooling function is executed, that is, when there is more than one index in the tile and when the sequence length is greater than one.

4.4.2 Unified Buffer (UB) lookups

Another valuable strategy builds instead on the idea of first moving chunks of the tables in the unified buffer and then leveraging the vector unit to access the rows in parallel through vectorized indexing. Not only is this strategy removing the dependency on extremely irregular access patterns typical of lookup operations, but it also avoids the expensive off-chip random memory accesses that represent a bottleneck for the performance.

The pseudo-algorithm 2 reports the data flow applied when using the GM-UB or L1-UB strategies, considering again the processing relative to a single embedding table. Unlike the direct lookups, an extra tiling loop is added to accommodate an additional buffer in UB, which stores one chunk at a time of the embedding table. In this context, since the query indices are runtime values and we are looking up one chunk of a table at a time, we must implement a method to handle indices

Algorithm 2 GM/L1->UB symmetric UB lookups

```
t_{gm-l1} ∈ R^{m_i,E}                                   ▷ table stored either in GM or L1
q_{gm} ∈ N^{B,s}                                        ▷ batch of vector queries of s indices, stored in GM
o_{gm} ∈ R^{B,E}                                        ▷ output of the kernel (i.e., pooled result of looked up rows)
spatial for each core_{id} ∈ {0,...,K-1} do
    B_o, B_c, B_q ← compute_core_tiling(B, K, ...)
    zero_{ub} ∈ N^V                                     ▷ V: elements computable in parallel by the vector unit
    for each tile_o do                                  ▷ outer tile for the result
        o_{ub} ∈ R^{B_o,E}                              ▷ buffer for the result in UB
        for each tile_c do                              ▷ tile for the chunks of the table
            c_{ub} ∈ R^{B_c+2,E}                        ▷ buffer for the chunk in UB
            c_{ub} ← t_{gm-l1}[tile_c * B_c : (tile_c + 1) * B_c]
            c_{ub}[0, :] ← 0
            c_{ub}[B_c, :] ← 0
            norm_{ub} ∈ N^V
            norm_{ub} ← -tile_c * B_c + 1
            bound ← -tile_c * B_c + 2 - 1
            bound_{ub} ∈ N^V
            bound_{ub} ← bound
            q_{cnt} ← 0
            for each tile_q do                          ▷ tile for the query
                q_{ub} ∈ N^{B_q,s}                      ▷ buffer for query indices in UB
                q_{ub} ← q_{gm}                         ▷ data movement of query from gm to ub
                q_{ub} ← q_{ub} + norm_{ub}             ▷ normalization of indices
                q_{ub} ← min(q_{ub}, bound_{ub})        ▷ clipping out-of-bound indices
                q_{ub} ← max(q_{ub}, zero_{ub})         ▷ clipping out-of-bound indices
                for each z_j ∈ {0,...,s} do
                    a_{ub} ∈ R^{B_q,E}                  ▷ accumulator in UB
                    temporal for each g_i ∈ {0,...,(B_q/16) - 1} do
                        V_{idx} ∈ N^16                  ▷ vector of 16 indices
                        for each i ∈ {0,...,15} do
                            V_{idx} ← q_{ub}[g_i * 16 + i, z_j]
                        end for
                        a_{ub} ← vectorized_lookup(t_{gm-l1}, V_{idx})
                    end for
                    start ← B_q * tile_q
                    end ← B_q * (tile_q + 1)
                    o_{ub}[start : end, :] ← o_{ub}[start : end, :] + a_{ub}   ▷ vector reduction
                end for
            end for
            start ← tile_o * B_o
            end ← (tile_o + 1) * B_o
            o_{gm}[start : end, :] ← o_{ub}             ▷ Move output to global memory
        end for
    end for
end for
```

that are out of the bounds of the current chunk. To this extent, there are two available options that add extra workload to different hardware units. The first consists of checking, through the scalar unit, if the current indices are in-bound or out-of-bound, which determines which index is used for the look-up and which is not. Moreover, since we exploit the vector unit to make parallel lookups, this option also requires a reorganization of valid query indices before doing the actual vectorized operation. The other option instead consists in adding two extra dummy rows, one on top, and one on the bottom of the chunk, with value set to 0. Then, the query indices are further normalized in three steps, consisting in subtracting the relative offset of the current chunk from the indices and then clipping the out-of-bound indices to point correctly to the dummy rows. We opted for the second option to avoid adding extra workload on the scalar unit, which reflects two advantages.

- One consists of the fact that the scalar unit must not check the single indices, which is a great advantage in a context where the scalar indexing could easily become a bottleneck.
- Secondly, normalizing and clipping the indices makes the strategy completely independent of the input query distribution, making the strategy particularly effective in scenarios where the input query distribution causes problems.

Thus, the data flow includes an additional chunk movement from global memory or L1, depending on whether the table was previously preloaded in L1. Once the query is normalized and the out-of-bound indices are clipped, a vector of 16 indices is prepared and used to perform a vectorized look-up operation through the scalar unit. Notice that 16 is the maximum number of addresses that can be fed to the vector unit of the Da Vinci architecture. Once the vectorized look-up is completed, the vector unit finishes the operation by performing the vector sum reduction to pool together the embedding vectors relative to the same sequence.

Kernel composition Having defined four possible strategies for processing one embedding table, it is also essential to focus on how the processing of different tables is organized. We recall from section 2.3.3 that the execution of programs on Ascend accelerators is based on binary files, called offline models, which contain one or more compiled CANN operators. In particular, CANN operators can be either built-in ones, developed by expert developers to be highly flexible and optimizable by the ATC compiler, or custom operators written in one of the available programming abstractions. In principle, we should execute one kernel for each categorical feature, as each table is independent of one another. Unfortunately, due to architectural limitations, to leverage the persistent preloading of some tables in the L1 buffer, we must fuse all the kernels that use an L1-preloading strategy. As a result, three different custom CANN operators are built and installed in the library.

1. One consists only in persistently preloading a set of tables in the L1 buffer so that other kernels can leverage it to do the look-ups.

2. The second one performs in a single fused kernel execution all the lookups related to tables that have been preloaded persistently in L1.

3. The last one consists of a standard operator which can execute either the GM or the GM-UB strategy, thus executing exactly one kernel for each embedding table to process.

4.4.3 Model-level symmetric lookup strategy

To wrap up, four different strategies are available for the computation of pooling-based embedding look-ups of each embedding table, while the workload can be easily split evenly across the cores along the batch size dimension since we are using a SIMD paradigm. However, choosing the optimal strategy to apply on each embedding table among the four proposed is not a trivial task, and thus requires solving an optimization problem. More specifically, the problem consists in finding the optimal set of strategies under the constraint imposed by the size of the L1 buffer, that minimizes a defined objective, which can be, for example, the average latency. More formally, as described in section 4.1, we have N embedding tables, each with m_i rows composed of E elements. Thus, given a batch size B, the set of sequence lengths S, the maximum cores K, the input query distribution I, and a given space L reserved for preloading tables in L1, we define an objective function J to minimize. Therefore, the problem becomes finding the optimal policy $P = \{p_0, \ldots, p_{N-1}\}$, where p_i can take four possible values $\{gm, gm-ub, l1, l1-ub\}$, which minimizes the objective function J.

$$P = \arg\min_{P} \left(J(B, M, S, K, L, I, P)\right) \tag{4.1}$$

Moreover, given that the performance metrics are several and will be described in detail in section 5.1.1, the objective function J can be defined in many ways. For the scope of this thesis, we defined J as the P-99 worst-case latency of the entire pooling-based embedding look-up processing. In particular, after running a DLRM inference for a defined number of iterations using a specific policy, the P-99 latency refers to the 99th percentile of the acquired latencies, that is, the specific value for which it holds that 99% of the considered latencies are less or equal to it. Defining the objective in such a way allows us to achieve a policy that guarantees a specific P-99 latency boundary, thus meeting the SLA requirements which usually characterize the deployment of DRLMs.

However, as underlined by [27] and [29], the total latency achieved with a specific policy is not easy to estimate, as it depends on a big number of factors.

Performance model Since the problem is non-trivial, together with the fact that we want to build a strategy that can be applied to different use cases effortlessly, some simplifications are made. In particular, as also proposed by [27] and [29], the total estimated latency is computed as the sum of the latencies of the single tables, thus neglecting the overlap and complex hardware interactions between the processing of different embedding tables. As a result, given a new cost function W, which estimates the cost of a single table, we define a new cost function J' as:

$$J' = \sum_{i=0}^{N-1} J'_i = \sum_{i=0}^{N-1} W(B, m_i, s_i, K, I, p_i) \tag{4.2}$$

However, as opposed to [27] and [29], instead of relying on neural networks, we choose to keep a simple linear formulation for estimating the cost W due to the limited number of collected samples. The linear formulation is trivial and so defined as:

$$W = \begin{cases} c_0(p_i) + t_i(p_i) \cdot B \cdot s_i & \text{if } p_i \neq \text{'ub'} \\ c_0(p_i) + n_c \cdot [t_m(p_i) + t_i(p_i) \cdot B \cdot s_i] & \text{otherwise} \end{cases} \tag{4.3}$$

Referring to equation 4.3, c_0 captures all the constant terms which do not depend on the number of iterations, whereas t_i is the estimated iteration time, and refers to capturing how much time, on average, a single pooling-based look-up operation takes. Moreover, the time estimation of the tables which are processed using UB strategies is formulated differently, as it involves an additional chunk movement whose latency depends on the size of the transferred chunk, as well as the type of memory where it comes from, which can be either the global memory or the L1 buffer depending on the policy.

Even though the formulation of the performance model is straightforward, estimating accurate values for its parameters is not trivial. For simplicity, we collected multiple individual samples of t_i, c_0, and t_m for the four strategies by changing most of the parameters, including the batch size B, the number of rows m_i, the sequence length s_i, and the input query distribution. Specifically, to measure the iteration time t_i, we removed all the constant operations from the written operators and introduced a multiplier for averaging better the iteration time; then, we ran 100 iterations of hardware profiling using the Ascend 910 AI accelerator. Thus, the average latency was divided by the batch size multiplied by the sequence

length to obtain the iteration time t_i.

$$t_i = t_{avg}/(B \cdot s \cdot multiplier)$$

A similar procedure was performed to estimate the constant terms, this time by removing the iterations and leaving only the constant operations repeated by a multiplier factor. Lastly, in order to estimate the chunk movement time, we measured the actual throughput achieved by moving chunks of different sizes from the L1 buffer and the global memory. Then, we used such estimation to estimate the movement time of a chunk of C bytes as:

$$t_m = C/throughput_{measured}$$

The procedure for estimating the iteration time was repeated varying the previously mentioned parameters, including the batch size, the sequence length, the number of rows, and the input query distribution. Lastly, given a new set of values for the parameters, a simple 1-nearest-neighbor algorithm was used by taking the most similar point from the sampled ones. For instance, supposing to have collected two samples, one with a sequence length set to 10 and the other with a sequence length set to 50, a new estimation for a sequence length set to 20 would simply pick the iteration time t_i and the fixed term c_0 of the sample with sequence length set to 10. Even though this method is not perfectly accurate, we argue that this approximation is sufficient to solve the policy optimization problem effectively. However, given the limited space for preloading the tables in L1, which in the Da Vinci architecture is limited to 1 MB, the policy optimization can not be split into the independent optimization of the strategy of the single tables.

Greedy policy optimization As a consequence, given the performance model just described, the problem becomes finding an optimal policy according to equations 4.1 and 4.2. Due to the dependency problem introduced by the limited size of L1 memory, it is convenient to split the policy optimization problem into two equivalent simpler subproblems. The first one consists in finding the optimal set of tables to preload in the L1 buffer, and the other consists in independently choosing the optimal strategy for the remaining tables.

1. First, given a set of N tables, together with a specified batch size and an input query distribution, we use the performance model to obtain a matrix of estimated costs $U \in \mathbb{R}^{N,4}$, which includes, for each table, the estimated costs of processing it with the four strategies.

2. Then, we retain, for each table, only the strategy which provides the minimum estimated cost, thus being able to extract a set Z of potential candidates to preload in L1, which corresponds to the set of tables whose best strategy is either L1 or L1-UB.

3. As a result, the first optimization problem consists in finding the optimal set of tables included in Z, which maximizes the number of iterations involved in the computation (i.e., the sum of the sequence lengths associated with the tables). To this extent, a greedy approach is exploited by sorting the candidates in descending order by the number of iterations involved divided by the number of rows. Then, we pick the tables respecting the order to fill the maximum L1 capacity.

4. Having obtained the best set of tables for which it is better to preload them in L1, the second subproblem becomes trivial. In fact, for the hypothesis, the cost of each table can be estimated separately, thus resulting in the possibility of just choosing the best strategy from the matrix U for the remaining tables.

4.5 Asymmetric inter-core lookup data flow

From preliminary experiments, it emerged that the two strategies based on preloading the embedding table in the L1 buffer provide dramatic improvements in terms of both average latency and P-99 worst-case latency. However, in section 4.4, we have seen that exploiting the symmetric paradigm implies executing the same code on all the cores. As a result, given K cores, even though the aggregated L1 buffer is large ($K \cdot 1MB$), only 1 MB is used to store different tables, as the preloaded tables are replicated and equal for all the cores.

Motivated by this, we designed an alternative strategy to fully exploit the aggregated L1 buffer. In particular, we decided to opt for an asymmetric paradigm, which in contrast to the previous one, allows executing different instructions on each core on a possibly different set of data. Moreover, even though having a different code for each core implies a drastic increase in the code size, the approach can be applied effectively on Ascend accelerators as they equip an instruction cache exclusive for each core, thus avoiding the potential worsening of the instruction cache hit ratio which would have been introduced otherwise. To the best of our knowledge, this is the first time this approach has been implemented in AI accelerators to boost the performance of DRLMs.

4.5.1 Tasks and chunking

The main novelty introduced by the asymmetric approach is the flexibility of exploiting the cores to execute different programs. As a result, to increase the flexibility of the processing relative to each embedding table, we decided to introduce two new concepts, tasks and chunking.

1. First, we introduce the concept of *task*, which is meant to represent a pooled-based lookup operation applied on a portion of the input query. In particular, it tells which elements of the batch must be processed, as well as which elements of each sequence. For the sake of clarity, supposing to have a categorical feature with a sequence length equal to 16, and a batch size equal to 32, a task could express the computation of the first 4 samples and the last 2 elements of the sequence, which means that only 8 indices in total would be processed. Furthermore, a task includes also which strategy out of the four available must be used to perform the lookups.

2. Secondly, we designed a chunking functionality capable of splitting the embedding tables into chunks. However, we restrict the chunking of tables only to horizontal cuts, leaving the vertical ones as a possible future improvement. As a result, it is possible to split large tables into chunks to be stored in the L1 buffer of different cores.

To emphasize the importance of the chunking functionality, it is worth noting that DLRMs have tens or, in some cases, even hundreds of tables with an incredible difference in size, from a few KBs to hundreds of GBs. Thus, chunking large tables offers the possibility to set up a strategy that not only scales out to multiple cores but also potentially to multiple devices. For example, using this strategy, the Ascend 910 accelerator, equipped with 32 cores, each with 1 MB of L1 memory, can now exploit all 32 MBs of aggregated L1 memory. As a result of combining tasks and chunking, the strategy is flexible enough to process any chunk of any table while restricting the computation only to a specific subset of the input batch.

However, the adaptation of the four strategies to support the new functionalities requires some attention. In particular, concerning the direct look-ups described in section 4.4.1, if a chunk of an embedding table is preloaded in the L1 buffer, input indices can fall out of its boundaries. To tackle this issue, a similar approach to the one already described in section 4.4.2 is used with minor differences. Specifically, instead of normalizing and clipping the indices, the clipping phase is removed and substituted with a boundary check made through the scalar unit. More details can be found in section 4.4.2.

Synchronization An essential and non-negligible difference with respect to the symmetric approach is represented by the possibility of having multiple cores trying to write on the same off-chip address range. This happens when two or more cores share chunks of the same embedding table, which makes the sharing of the corresponding output address range inevitable. If multiple cores share chunks of the same table, they compute partially pooled results, thus needing an extra global pooling operation to compute the final result. Luckily, pooling-based DLRMs are all based on pooling functions which are both associative and commutative,

thus implying that partial pooling can be performed independently by each core. However, it is worth considering that if such an approach were to be extended to non-pooling DLRMs, inevitably, all the cores would have to move the whole non-pooled sequence of embedding vectors to the global memory, causing a non-negligible loss of performance due to the on-chip to off-chip memory data transfer. A logical atomic accumulation operation is used whenever multiple cores have to accumulate the partially pooled result on the same address range. For the sake of completeness, notice that the actual hardware realization of such atomic operation is undisclosed.

Kernel aspects Since with the MIMD paradigm each core is executing a different code, splitting the computation relative to different tables in multiple kernels is highly inefficient. The reason lies in the fact that whenever a kernel finishes its execution, all the cores are automatically synchronized, causing an extremely inefficient execution given that the cores operate on different tables. As a result, only one fused operator, and so kernel, operates on the pooling-based embedding lookups of all the embedding tables of a DLRM. Similarly to what happens with the symmetric approach, a first operator is executed only for preloading some tables, or chunks of them, in the L1 buffer. Notice that the preloading operation is executed only once when the model is started. Afterward, the AI recommendation application keeps waiting for new users' queries and launches the fused lookup kernel when needed.

4.5.2 Sharding and load balancing problems

Having adapted the four strategies to support the new functionalities, the effectiveness of the MIMD approach is then based on the goodness of the policy used to process a set of embedding tables. In particular, differently than the policy optimization problem relative to the SIMD approach, the many degrees of freedom introduced by the chunking and task functionalities make the problem much more complicated to solve. In fact, not only can one table be chunked horizontally, introducing the first degree of freedom, but the associated workload can also be arbitrarily split along both the batch size dimension and the sequence length one. On top of that, we also have the freedom to choose which strategy to apply for a given task.

Given a set of N tables, each one associated with a fixed cost, the basic partitioning problem[1] consists in finding an optimal partitioning of the tables in K

[1] Partioning problem

groups to minimize the unbalancing between the total costs of the groups. The problem is NP-hard, and even worse, the asymmetric policy optimization problem represents a much more complicated load-balancing problem for the presence of additional degrees of freedom. Since it is evident that the problem must be simplified in order to make it more tractable, we propose some heuristics to fix the degrees of freedom at the expense of finding suboptimal solutions.

Greedy load balancing with heuristics

1. Firstly, splitting the workload across the sequence length dimension introduces overhead as we must still load the whole query just to use a fraction of indices. For this reason, we fix the sequence length to the maximum in each task.

2. Secondly, if a table is chunked, the related input indices cannot be split into different tasks. The heuristic finds motivation in the fact that if a table is split into chunks, we must replicate each chunk a number of times equal to the number of associated tasks. For the sake of clarity, suppose to have one table split into two chunks stored in the L1 buffer of two different cores, and the corresponding batch of queries split into two tasks. Without replicating the two chunks, if at least one index of one task executed in one core falls in the chunk stored in the other core, the result would be incorrect. Therefore, in order to perform the operation correctly, the two chunks must be replicated, thus occupying double the space. As a result, since chunking is typically required for large tables, replicating such big chunks would be a huge waste of memory.

The new problem formulation with heuristics is more tractable. Still, it does not correspond to any standard sharding problem as it presents the additional freedom of splitting the workload along the batch size dimension, as well as deciding how to chunk the tables. Therefore, a few options involving a different complexity are proposed.

1. The first consists in fixing both the workload splitting and the chunk splitting possibilities. The problem is thus translated into a partitioning problem, and solutions like the one proposed in [27] can be used effectively to solve it.

2. Another approach consists in fixing only the workload splitting, transforming the problem into a sharding problem with a single objective. This kind of problem is typical of database load-balancing scenarios and is much more complicated than the first one. A solution based on deep reinforcement learning, such as the one proposed in [30], can be used to solve this task.

3. Furthermore, similarly to the previous one, we could also fix the chunks and work on the freedom of the workload splitting, thus achieving a problem with the same degrees of freedom as the previous one.

Due to time and resource constraints, we opted for the first option, making the dimensionality of the problem more reasonable to deal with and the algorithms much more performant to be executed. To this extent, we employ heuristics to statically split the tables, thus treating each chunk as an independent table. Using the same performance model formulation described in section 4.4.3, a greedy optimization algorithm is thus proposed and structured as follows.

1. First, for tables that are both large and involve a low number of look-ups (i.e., they have a small sequence length), the L1 strategies are discarded, and the best remaining strategy according to the performance model is selected. We argue that the L1 memory is precious, and thus wasting a lot of space for storing large tables which involve a small workload is not optimal. In particular, we define a first threshold as the average size of the tables considering only the tables with a size smaller than the L1 size of one core. Moreover, we define the median of all the sequence lengths as a second threshold. Thus, a large table with a small sequence length is recognized when it has a number of rows greater than the first threshold and a sequence length lower than the second one.

2. Second, we split the tables that are both large and with a large sequence length evenly into a number of chunks equal to the number of available cores. This is due to the constraint that we introduced to simplify the problem, which consists in not having the possibility of splitting the workload when we refer to a table that has been chunked. More specifically, if we were to split a large table into an arbitrary amount of chunks, the workload of the cores in which we put these chunks would be extremely unbalanced.

3. Lastly, all the chunks and tables are sorted in terms of the cost given by the performance model divided by the number of rows. The sorted chunks are then assigned one after the other to the current group with the lowest cost, using the best strategy according to the performance model.

Chapter 5

Experimental results

To assess the effectiveness of the proposed strategies in accelerating pooling-based DRLMs inference workloads, we conducted extensive experiments using the Ascend AI accelerators. To this extent, the choice of how to structure the experiments, as well as which AI accelerator to use for running the benchmarks, was made to provide the fairest possible comparison of the competing strategies, involving, as a consequence, a variety of use cases. In this chapter, we will first describe the setting under which all experiments were conducted and the performance metrics used; then, we will assess the performance of the baseline (i.e., the embedding lookup built-in kernel implementation available in CANN), focusing on discovering its weaknesses and strengths. Lastly, the proposed methods will be compared using both single synthetic embedding tables and multiple tables belonging to a model used in production.

5.1 Experiments setting

Before digging into the characteristics of the various experiments, it is necessary to describe precisely the setting under which they were conducted, which data were collected, and what performance metrics were used.
First, all the experiments made use of the major Ascend AI accelerator, the Ascend 910, which was originally designed to accelerate training workloads rather than inference ones. The reason lies in the fact that the Ascend 910 is equipped with an HBM, thus resulting in a much more performant execution of the baseline strategy, which is mainly based on optimizing the accesses to off-chip memories, compared to an accelerator that uses instead DRAM like the Ascend 310. This is because HBM has a much higher overall bandwidth, achieved by stacking multiple layers of memory dies and offering a wider channel composed of multiple channels, which can be exploited for performing multiple embedding lookup reads in parallel, as

done, for instance, in [21]. Therefore, given the higher benefits provided to the baseline strategy when using the Ascend 910 compared to others, the resulting comparison with the proposed techniques is much fairer. Furthermore, since the L2 cache speeds up considerably the accesses to frequently gathered embedding vectors, its size plays a critical role. This reinforces the motivations for choosing the Ascend 910 over the 310, as they have 32 MB and 8 MB of L2 cache, respectively.

All profiling data were collected through hardware measurements, making them much more reliable than those obtainable through a software simulation. In particular, the Ascend platform comes with a sophisticated tool called *msprof*, which not only allows running in-hardware profiling but also offers the possibility of selectively enabling the capturing of data at different granularities. Specifically, the captured data allow extracting summaries of the kernel-wise performance, including the absolute and relative average utilization of the compute and memory units inside the cores, as well as the end-to-end task duration. As a side note, msprof also offers the possibility of capturing timeline data, which was extensively used throughout the development of the operators.
For clarity and reproducibility purposes, we report the whole workflow followed for running the experiments.

1. First, a specific use case, that is, a specific set of values for the varying parameters, is prepared for making a comparison. In particular, the use case includes a specific number of embedding tables together with their characteristics, which are the number of rows and the sequence length. Another critical parameter is the distribution of the input queries, which drastically influences the access patterns of the lookups.

2. Once the specific use case is ready, we select the set of strategies we want to compare, choosing among the baseline and the proposed ones.

3. Then, the set of batch sizes for assessing the performance of the competing strategies is defined.

4. Thus, for each batch size to try and for each strategy to compare, the ATC compiler is invoked for creating a binary file called *offline model*. More details about this step can be found in section 2.3.3.

5. Then, each offline model is loaded into the accelerator and executed, with profiling enabled, for a defined number of iterations feeding queries sampled from the previously chosen query distribution.

6. Lastly, the msprof tool is invoked to parse and summarize the collected profiling data.

Two kinds of use cases were defined, one involving only one single synthetic embedding table crafted with specific characteristics and the other consisting of a set of embedding tables extracted from a model used in production. Moreover, to make the comparisons more effective, we focused on varying the most critical parameters, thus fixing the less important ones. In particular, since all the memories utilized for retrieving the embedding vectors are usually characterized by a large bandwidth, we fixed the embedding space dimensionality to 16, thus making the embedding vectors very small and leading to poor memory-bandwidth utilization. Moreover, the data type of the weights was fixed to float16, which is a very used data type in inference scenarios since it allows to achieve low latencies compared to float32. In this way, each row of the embedding tables is made of 16 float16 elements, thus making it large 32 bytes. Lastly, the pooling function of the embedding layers was fixed to the element-wise sum, which consists of summing element by element the embedding vectors belonging to a sequence. We argue that this comes without any loss of generality, as pooling-based DLRMs are mostly based on associative and commutative functions, such as max or min, thus making no practical difference in the data flow as they all involve a local vector operation on each core. For the sake of completeness, also the data type of the query indices was fixed to unsigned int64, thus not restricting the maximum table size supported.
Besides, a very important parameter characterizing a use case is the input query distribution, as it determines the access patterns of the table lookups. To this extent, all the use cases included one of the following input query distributions:

- Uniform distribution, which is simply a discrete distribution whose density value is equal in every valid point and null in all the others. The valid points range from 0 to $M_i - 1$, where M_i is the number of rows of the i^{th} table. Put in simpler terms, the probability of extracting each index to access a table row is the same.

- Fixed distribution, which puts all the density in only one point, which means that only one value will always be extracted. Specifically, we fixed the value of the index to 0, as this is not affecting the performance of the lookups in any way. For the rest of the manuscript, we will refer to this distribution as the *fixed distribution*.

DCNv2 model characteristics To assess the performance of the proposed strategies, we used the embedding tables of a model used in production by an anonymous company. The architecture of the model is the DCNv2, already described in section 2.1.1, consisting of N embedding tables, a cross-network, and a deep network. The anonymous company built a model based on such architecture starting from an undisclosed private dataset, with 84 embedding tables of various sizes, ranging from 8 to 176322 rows. In particular, from figure 5.1, it can be seen

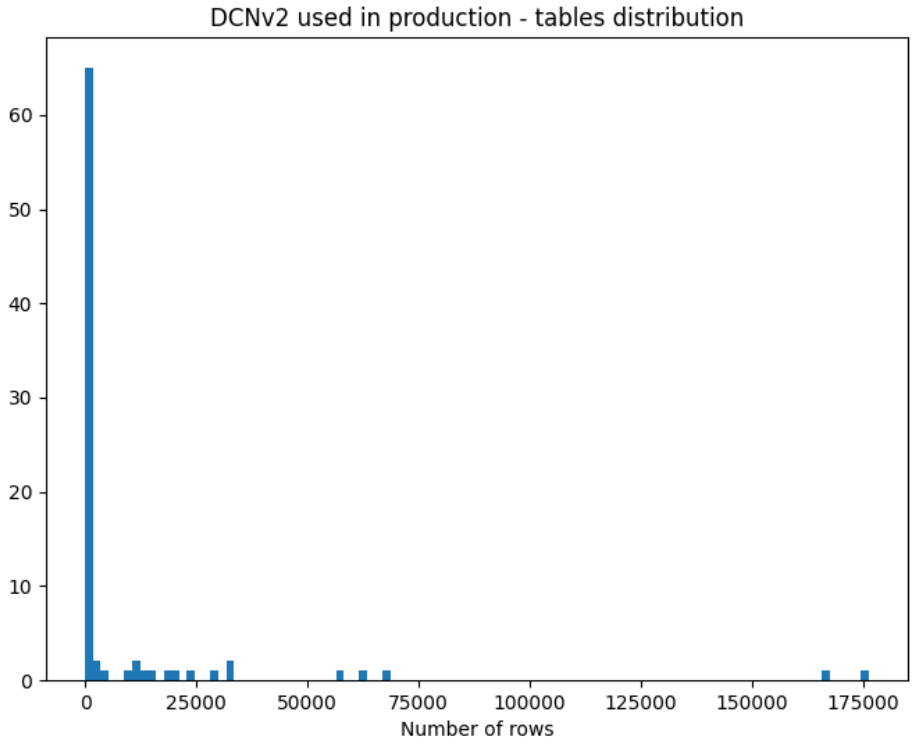

Figure 5.1: Table size distribution for the 84 tables of the DCNv2 model used in production.

that the model is mainly characterized by very small tables, with a few medium and large outliers. Furthermore, each embedding table is associated with a number called sequence length, which ranges from 1 to 172. As can be seen in figure 5.2, the majority of the lookups are performed from small and medium tables, whereas only less than 10% of them from very large ones. Lastly, the embedding space dimensionality is fixed to 16 float16 elements, as already discussed in the previous paragraph. To avoid ambiguity, even though this model will be referred to as simply *DCNv2 model*, notice that we are referring only to the specific model used in production, which has been trained on a specific dataset, and not in general to the DCNv2, which is a general DLRM architecture.

5.1.1 Performance metrics

Having defined the use cases and the structure of the experiments, another essential step is defining good performance metrics for assessing the goodness of the proposed

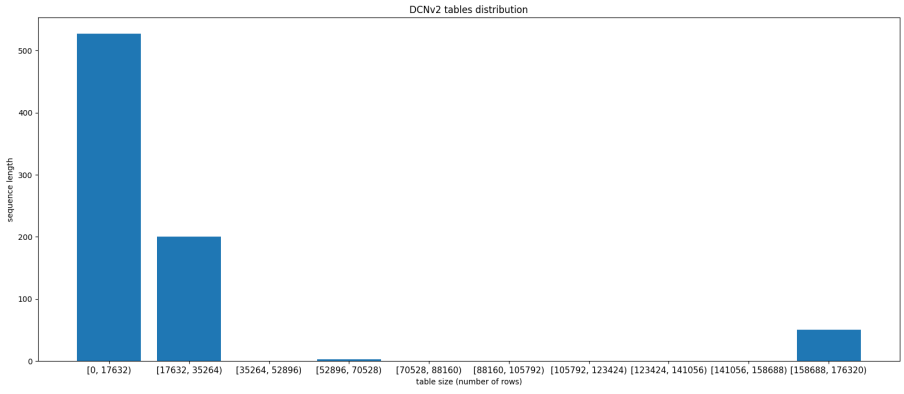

Figure 5.2: Sequence length distribution among the different tables of the DCNv2 used in production.

strategies.

A very common performance metric used to measure the performance of DLRMs is the *latency*, that is, the time elapsed between the reception of a new user's query and the availability of its result. In other words, latency is the time spent by all the executed kernels to process a batch of queries, plus the time spent transferring the query and its result from the host to the accelerator and vice versa. It is worth noting that increasing the batch size might also increase the latency, as the entire batch must be processed together, and thus the time when the result relative to the first sample becomes available increases.

Another important metric is the throughput, which in this work refers to the number of queries processed in a second (query per second, or qps). If the data flow is well designed, increasing the batch size should lead to a higher parallelization of the computation, leading to a higher throughput as a result. This is because the many processing elements in the available data path are exploited in parallel when processing more data. Notice that every system is characterized by a maximum throughput, which is limited by the maximum parallelization exploitable with a given data path and data flow.

Since DLRMs are usually deployed under strict SLA requirements, including an upper bound on the worst-case latency that must be guaranteed, assessing the trade-offs between throughput and worst-case latency is vital. In particular, increasing the batch size can lead to increased throughput at the cost of having a potentially higher latency. Therefore, assessing the trade-offs between the two

metrics is critical as it directly reflects the goodness of a given data flow in exploiting effectively the available data path. To be more precise, the worst-case latency is usually not the best metric, as rare failures do not have catastrophic consequences. A more common metric expressed in SLA requirements is the P-99 or P-95 worst-case latency, which refers to the 99^{th} or 95^{th} percentile of the considered latencies. More specifically, given a set of latencies, the X^{th} percentile is the specific value for which it holds that X% of all the considered latencies are less or equal to it.

Even though assessing the worst-case performance of such models is important to guarantee a minimum quality of service, assessing the average latency provides instead some clues about the average quality of the user experience, which is critical for the success of businesses. More specifically, a common scenario in today's data centers is represented by having an upper bound on the P-95/99 worst-case latency so that the batch size cannot be increased more than a given value to meet the requirements, thus resulting in having a corresponding average latency.

To sum up, the common goal of all the strategies that try to maximize the performance of DLRM inference is usually to produce a data flow that, given a data path, delivers good performance considering all three metrics:

1. Average throughput, which is critical in every deployed system to achieve a high hardware utilization and thus a good energy footprint.

2. P95/99 worst-case latency, which allows to meet the imposed SLA requirements.

3. Average latency, which directly reflects the end-user experience.

5.2 Baseline assessment

Before comparing the proposed strategies, it is essential to understand the weaknesses and strengths of the baseline method. As we have seen in chapter 4.2, we have defined the baseline as a way of performing pooling-based embedding lookups by exploiting the well-tested ATC compiler of the Ascend platform. In particular, the compiler introduces extensive optimizations, both intra- and inter-kernel, and uses built-in CANN operators written by experts to define the data flow for the Ascend AI accelerators.

Memory access rate bottleneck As we have already underlined in chapter 4.1, DLRMs are limited by the high number of random memory accesses to retrieve many small embedding vectors. Therefore, it is fundamental to assess through

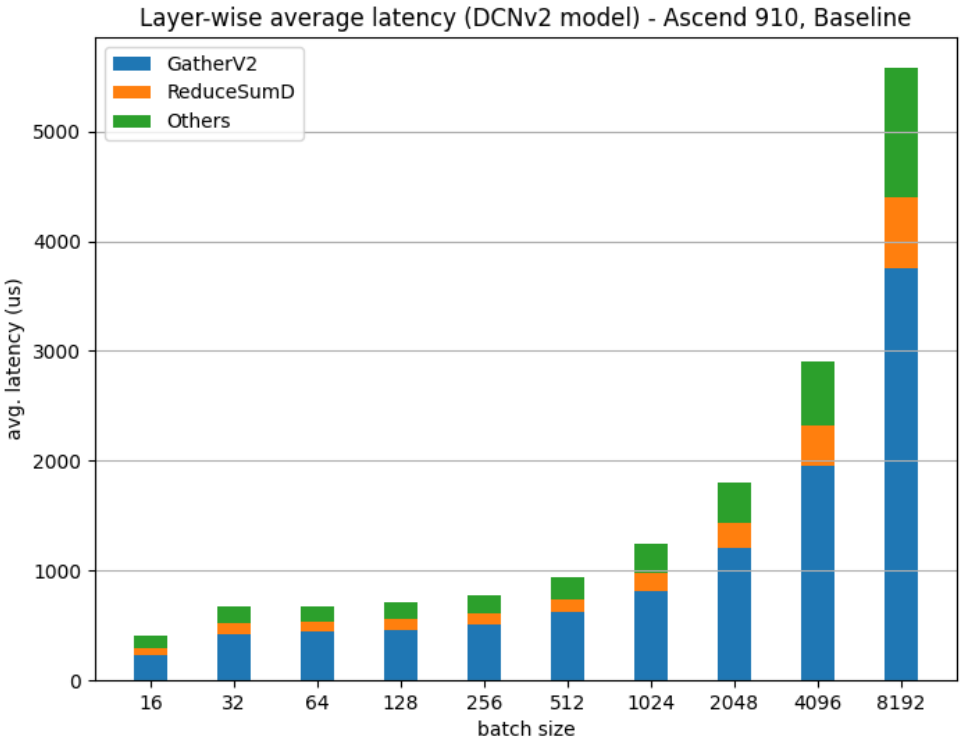

Figure 5.3: Layer-wise average latency using the baseline strategy on the DCNv2 model executing on the Ascend 910. The input distribution is uniform and all the 32 cores are enabled.

experiments if this represents an actual problem when using the baseline strategy on the Ascend 910 accelerator. To this extent, since the baseline method converts each layer of the original model to a corresponding individual CANN operator to execute, we first evaluate the layer-wise performance considering the layers of the DCNv2 model.
As can be seen in figure 5.3, not only do the gathering layers (i.e., lookups) take around two-thirds of the total model execution time, but even worse, the whole pooling-based lookup operation makes up three-quarters of the total time. Moreover, increasing the batch size does not improve the situation, showing that the pooling-based embedding lookups limit the overall performance.

Hardware utilization One effective way of characterizing the performance of a given data flow strategy is analyzing the hardware utilization of the many units composing the data path. To this extent, using the Ascend 910 AI accelerator, we

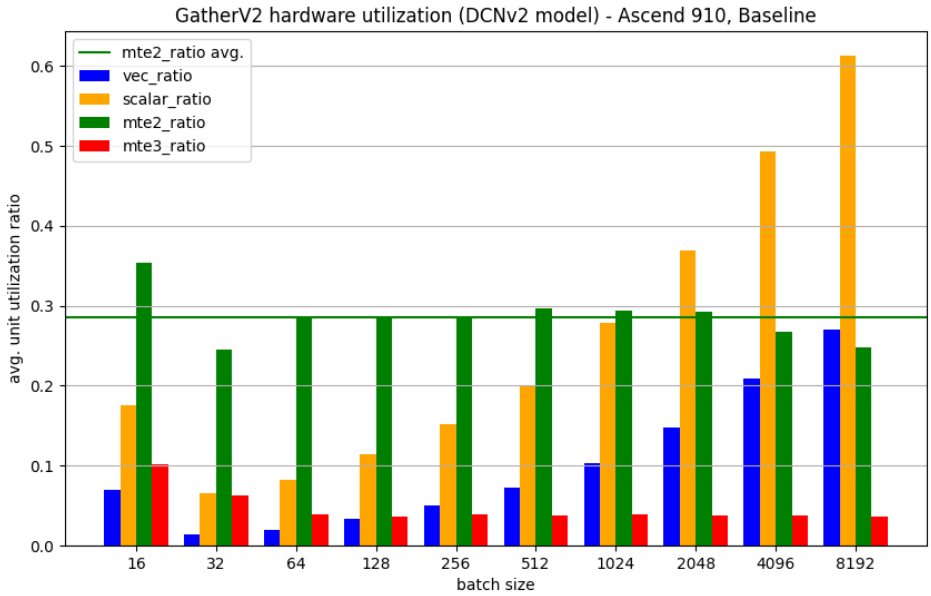

Figure 5.4: GatherV2 CANN operator average hardware core unit utilization considering the DCNv2 model executed using the baseline strategy on the Ascend 910. The input distribution is uniform, and all 32 cores are enabled.

assessed through experiments the hardware utilization of each unit composing the Da Vinci's core, which are shown in figure 2.4 and have been already described in section 2.3.1.

We briefly recall that the three MTE units, MTE1, MTE2, and MTE3, are executing DMA data transfers between on-chip buffers, from off-chip to on-chip memories, and from on-chip to off-chip memories, respectively. Moreover, the vector unit can perform operations in parallel on the elements of vectors, such as a vector sum, whereas the scalar unit handles any other instructions, including the computation of scalar values such as runtime addresses.

To better understand what are the causes of having such a high workload introduced by the GatherV2 and ReduceSumD CANN operators, we compared the layer-wise hardware utilization of the units inside the core, that is, the time spent in that specific unit divided by the total kernel execution time. From figure 5.4, we can see that the MTE2 unit, responsible for DMA transfers of data from off-chip to on-chip memories, makes up around 30% of the average latency of the GatherV2 layers. Besides, another very interesting discovery is that increasing

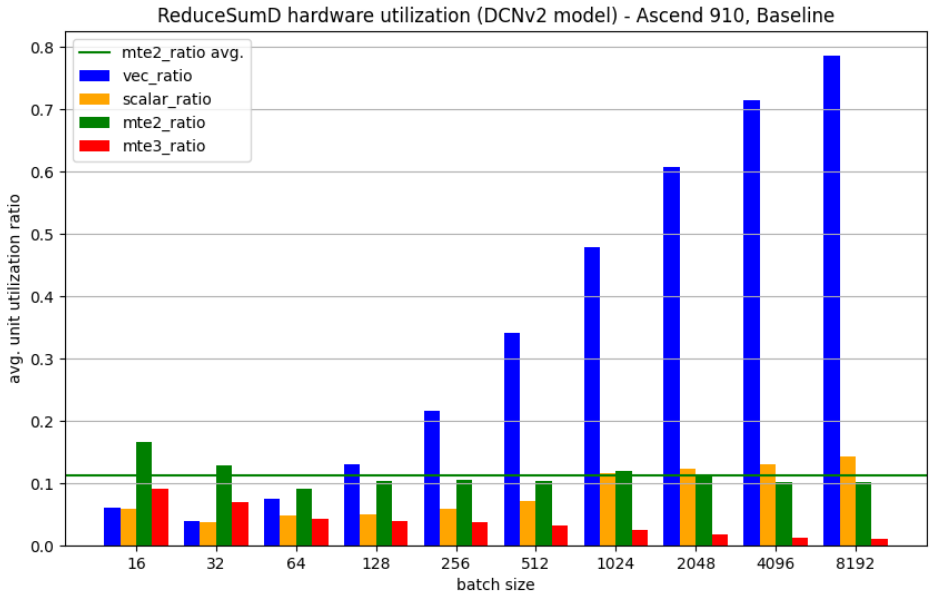

Figure 5.5: ReduceSumD CANN operator average hardware core unit utilization considering the DCNv2 model executed using the baseline strategy on the Ascend 910. The input distribution is uniform and all the 32 cores are enabled.

the batch size leads to an increase in the utilization of both the scalar and vector units. We argue that this is accountable to the indexing operation, which requires the scalar unit to generate a lot of addresses at runtime. Moreover, the baseline strategy arguably exploits the vector unit to make vectorized operations on the indices and/or embedding vectors.
As shown in figure 5.5, not so surprisingly, the utilization of the vector unit is dominant in the ReduceSumD layers, as it involves many vector reduction operations. Therefore, considering the dominance of the gathering and reducing operations shown in Figure 5.3, together with the hardware usage shown in figures 5.4 and 5.5, not only are the pooling-based lookups making up more than three-quarters of the average model latency but also the bottleneck is represented by different units depending on the batch size used. In particular, from figure 5.4, we can see that up to a batch size equal to 1024, the gathering operations are limited by the transfers of the embedding vectors, whereas afterward, the operators become indexing-bound, making the scalar unit the new performance bottleneck.

Cache conflicts Assessing the performance using only a uniform distribution is not ideal as the real distribution may differ a lot from it. Thus, it is important

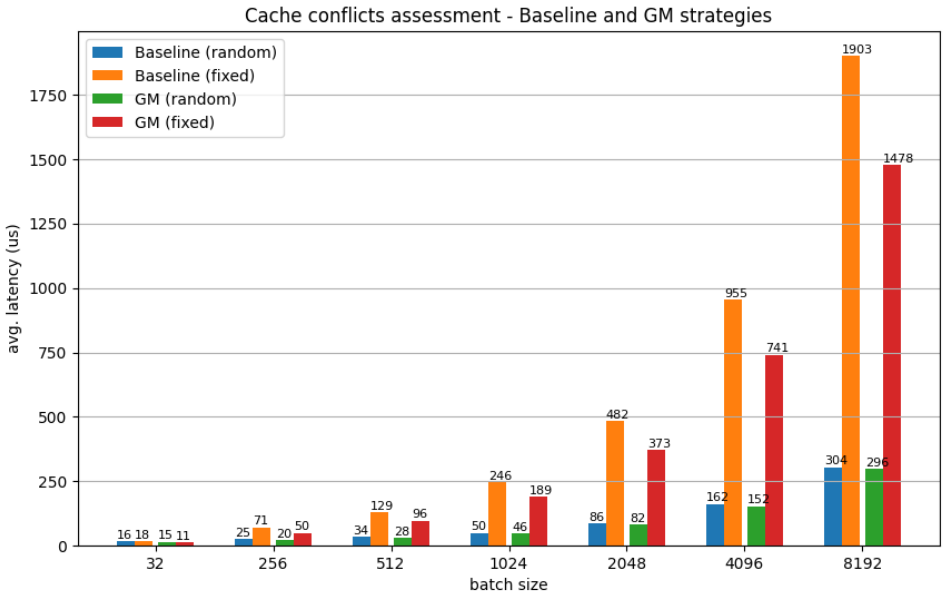

Figure 5.6: Baseline and GM strategies used on a single table made of 32760 rows, with sequence length set to 5 and enabling all the 32 cores of the Ascend 910. The fixed input distribution aims to make multiple cores conflict on the same cache line.

to evaluate the performance using problematic distributions for the given data flow. Since the Ascend AI accelerators support caching functionalities relying on L2 memory cache, another problem arises. In particular, since the cache is shared among all the cores, when two or more cores try to access the same cache line simultaneously, a data-access conflict must be solved, thus resulting in the serialization of the accesses involved. It is worth noting that also the channels of HBM are naturally subject to conflicts, but the penalty is less visible due to the difference in speed between the two memories. For this reason, we decided to introduce an ad-hoc input query fixing all the indices to the same value, previously described in section 5.1, which should be able to create the highest possible number of conflicts between the cores.

As shown by figure 5.6, both the baseline and GM strategies experience huge drops in performance when using a fixed input distribution compared to a uniform one. More specifically, considering a single embedding table, the baseline provides a worsening of performance when considering a fixed input distribution w.r.t a uniformly distributed one ranging from 2.8x for a batch size equal to 256 up to

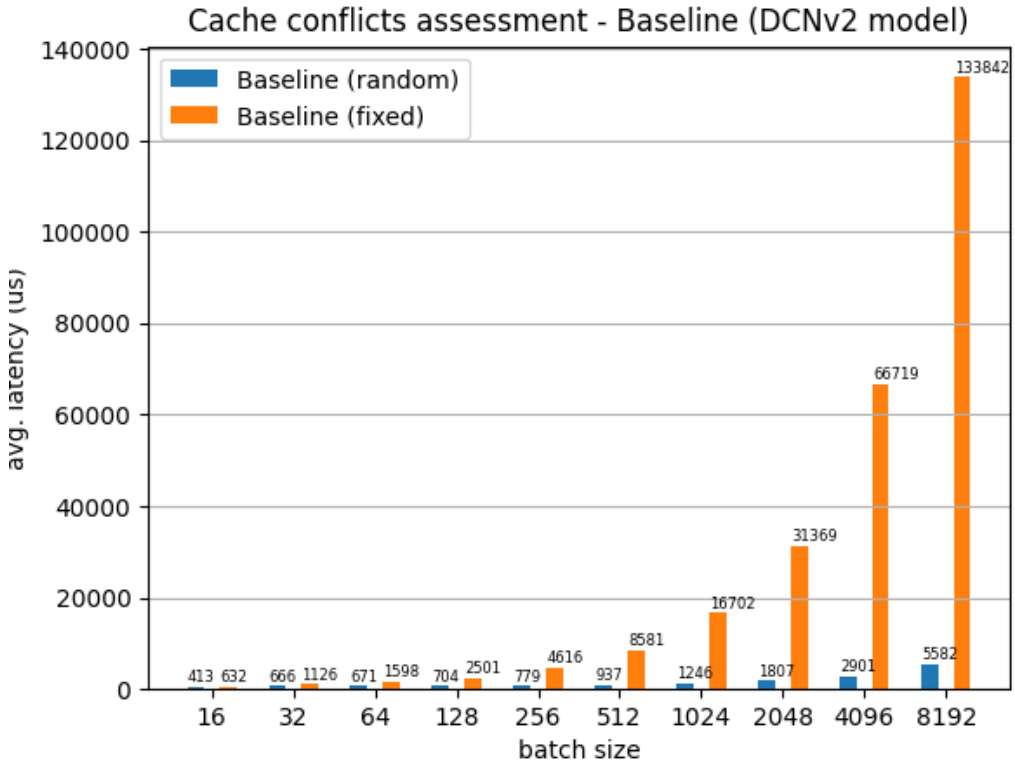

Figure 5.7: Baseline strategy applied on the DCNv2 model used in production. All 32 cores of the Ascend 910 are enabled and the fixed input distribution is used to cause cache conflicts between the cores.

around 6x for a batch size of 8192. Besides, the GM strategy, which performs the lookups sequentially from off-chip memory while splitting the workload across the available cores along the batch size dimension, experiences similar problems resulting in being up to 5x slower when using a fixed input distribution.

The problem of having a fixed query becomes even worse when considering an entire model instead of a single table. As depicted in figure 5.7, the average latency of the model execution when using a fixed query compared to a uniformly distributed one on the DCNv2 model grows significantly as the batch size increases. In particular, the performance drops ranging from a worsening of 1.5x with a batch size equal to 16, stepping to 6x when reaching a batch size of 256, and reaching up to 24x when using a batch size of 8192. Not only is the relative worsening of performance massive when using a fixed query distribution, but also the magnitude of the average latency dramatically increases from hundreds of microseconds when

using small batch sizes up to hundreds of milliseconds for large ones.
As a result, the need to design new data flow strategies less dependent on the input query distribution is further motivated.

5.3 Table-level strategies assessment

Before comparing the proposed data flow strategies used to perform the pooling-based lookup operations, characterizing their performance could reveal valuable insights into how they operate. It is worth noting that the sequence length, or pooling factor, not only increases the number of embedding lookups for each sample in the batch but also increases the number of vector reduction operations that must be performed to obtain the result. Therefore, we compare two cases characterized by a sequence length set to 1 and 5, respectively, thus showing how the presence or absence of vector reduction operations affects the overall performance. Moreover, the workload is split evenly across the cores along the batch size dimension. Lastly, evaluating the performance when sampling the input query from different distributions is critical, as shown in paragraph 5.2. Again, a uniform distribution is compared to a fixed one with all the indices set to the same value in order to cause as many cache conflicts as possible.

5.3.1 Hardware utilization

To perform the assessment, we run hardware profilings of the proposed strategies for 100 iterations using only one table of 32760 rows ($\approx 1MB$). In this case, since we are interested in assessing the hardware utilization of each strategy, the input query distribution was fixed to the uniform one in order to avoid the problem of cache conflicts already described in section 5.2.
We consider the utilization ratio of each unit, that is, the time spent in that specific unit divided by the total kernel execution time. Since we have 32 cores, the utilization ratios are averaged between all the cores. Lastly, the ratios are averaged across the 100 iterations of the execution to make the data more reliable. Notice that the assessment of the hardware utilization relative to the baseline has already been discussed in section 5.2, whereas the one relative to the proposed strategies will be described in the following.

GM strategy We start by assessing the hardware utilization of a standard strategy consisting of performing the lookups directly from the global memory, which on the Ascend 910 accelerator corresponds to the HBM or L2 cache memory. First, figure 5.8 shows that the GM strategy is unquestionably dominated by the utilization of the MTE2 unit, responsible for the DMA data transfer from off-chip to

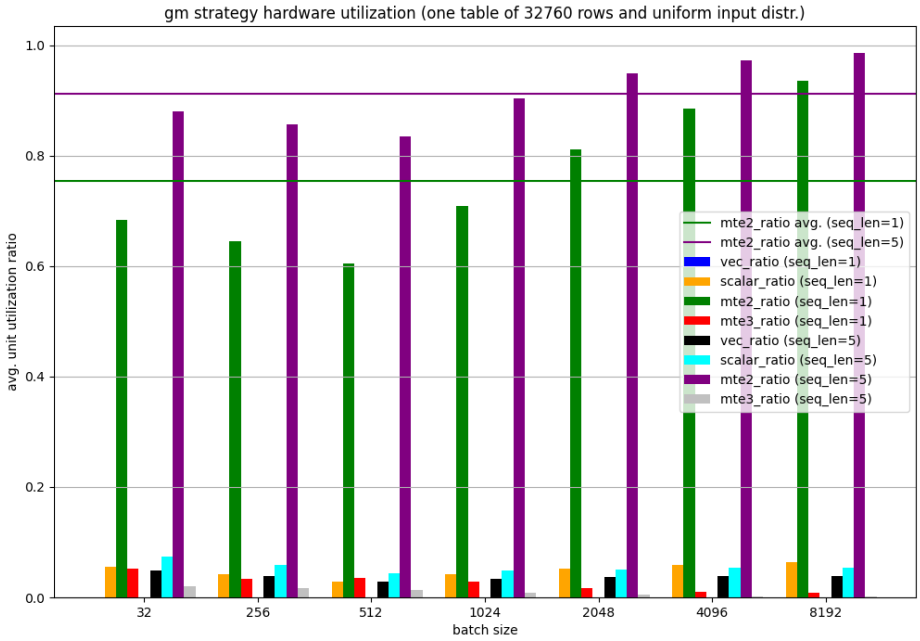

Figure 5.8: Hardware utilization of the core units of the Ascend 910. The runs refer to the execution of the GM strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled.

on-chip memory. This is not surprising since, as underlined by several works in the literature, such as [2], and as also shown by the limitations of the baseline method already described in paragraph 5.2, the many random memory accesses involved in the lookups represent a bottleneck for the overall performance. Moreover, we can see that increasing the batch size and the sequence length leads to a further increase in the MTE2 utilization, as one could expect since the bottleneck of the data transfers from off-chip memory worsens when increasing the number of lookups to perform.

L1 strategy We are now interested in assessing the hardware utilization of another approach, which consists in first preloading the table in the on-chip L1 buffer to be then able to perform individual lookups directly from there. In this case, the MTE1 unit of the Da Vinci core, in charge of making DMA transfers between on-chip memories, is of critical importance. In fact, the L1 buffer, where the table has been previously preloaded, is an on-chip buffer exclusive for each core.

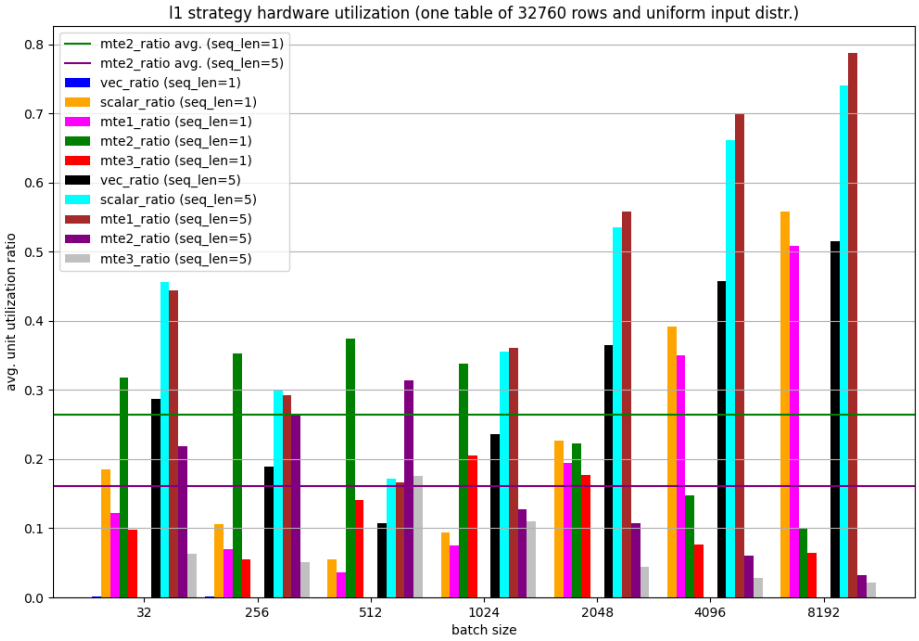

Figure 5.9: Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the L1 strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled.

Thus, the many embedding vectors are transferred in DMA from the L1 buffer to the Unified buffer, reflecting the high MTE1 utilization ratio shown in figure 5.9. Besides, we can see that since the transfers between on-chip memories are much faster than the ones involving off-chip ones, the MTE1 utilization remains relatively low compared to the MTE2 one up to a batch size equal to 2048. Notice that the shown MTE2 utilization is due to the transfers of the input queries, which are still much slower than on-chip individual lookups. Moreover, the scalar utilization is quite similar to the MTE1 one due to the generation of the many runtime addresses needed for the lookups. Lastly, as one could expect, since the sequence length determines the number of lookups to perform, using a sequence length set to 5 considerably increases the utilization of the scalar and MTE1 units compared to a sequence length set to 1.

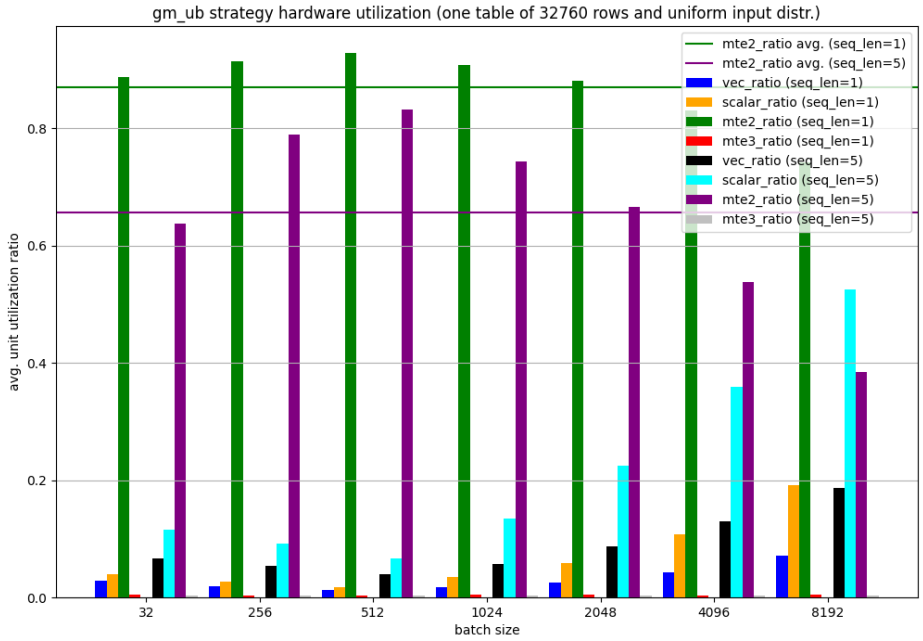

Figure 5.10: Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the GM-UB strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled.

GM-UB and L1-UB strategies The last two strategies exploit the Unified Buffer to make the lookups in a vectorized way. More precisely, the table is transferred in chunks either from the global memory or from the L1 buffer to the Unified buffer, depending on whether the table has been previously preloaded in L1 or not, and then the individual chunks are looked up in the unified buffer.

First, as shown in figure 5.10, since the GM-UB strategy is moving chunks from the global memory to the unified buffer, the utilization of the MTE2 unit is dominant. Moreover, the utilization of the MTE2 is compensated by the scalar and vector utilization when using both large batch sizes and sequence lengths. This is accountable to the fact that the high cost paid for the chunk movements is only amortized if the number of lookups is sufficiently large. Besides, due to the limitations of the programming abstraction of the Ascend accelerators, the vector of addresses needed by the vector unit for performing the lookups in a vectorized way is generated by the scalar unit instead of the vector one. This leads the scalar

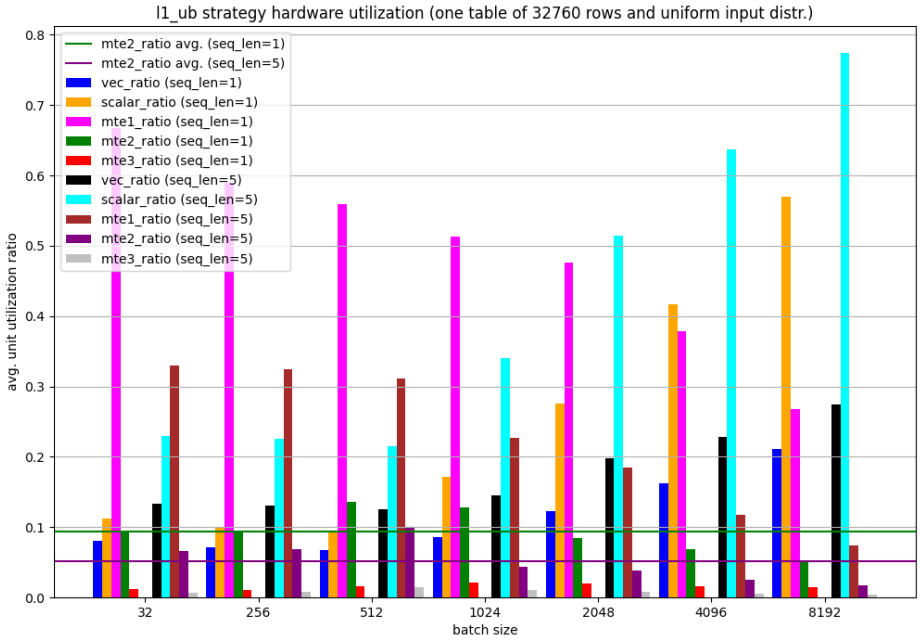

Figure 5.11: Hardware utilization of the Da Vinci core units of the Ascend 910. The runs refer to the execution of the L1-UB strategy on only one embedding table made of 32760 rows, using a uniform input distribution and varying the sequence length between 1 and 5. All 32 cores were enabled.

utilization to dominate the others when the number of lookups is high, showing that a potential further improvement could be achieved if the vector unit could generate addresses.

Considering instead the L1-UB strategy, consisting in moving the chunks from preloaded tables in L1 to the unified buffer, figure 5.11 shows a very similar utilization of the units compared to the one of the GM-UB strategy. More specifically, the role of the MTE2 unit shown in the GM-UB strategy is replaced by the MTE1 unit. Since the on-chip chunk movements between the L1 and the unified buffer are much faster compared to the ones made from the global memory, the cost of the chunk movements is compensated by the cost of the lookups much quicker, as confirmed by the unquestionable dominance of the utilization of the scalar and vector units with respect to the MTE1 when using large batch sizes.

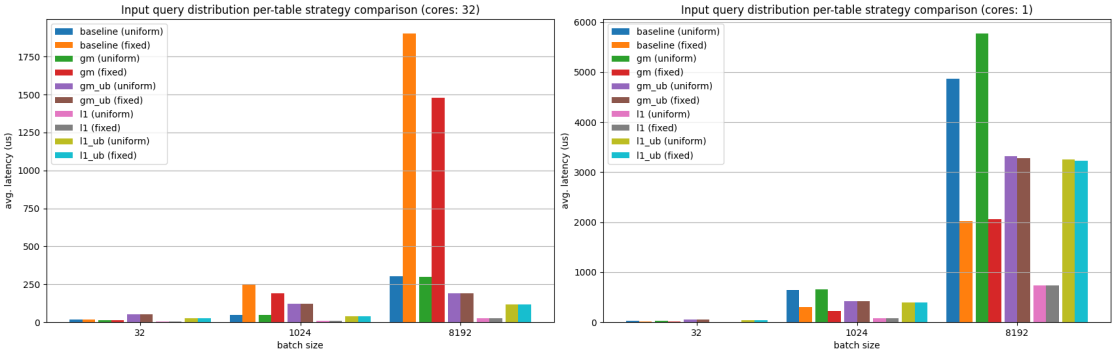

Figure 5.12: Average latency achieved on a table with 32760 rows and sequence length set to 5 on the Ascend 910. On the left, all 32 cores are enabled, whereas on the right, only one core is enabled, thus showing the presence or absence of cache conflicts.

5.3.2 Assessment of input query distributions

As shown in section 5.2, the baseline, as well as the GM strategy, experience a massive drop in performance when using a query with all the indices fixed to the same value compared to a query sampled from a uniform distribution, as shown in figure 5.6. This is caused by the high number of L2 cache conflicts that oblige the multiple cores involved to serialize the accesses to the same cache line. Even though the other proposed approaches cannot be affected by cache conflicts by design, it is still useful to understand to what extent they are independent of the input query distribution.
Figure 5.12 confirms the problem of the cache conflicts which affect both the baseline and the gm strategies, already discussed and visible also in figure 5.6. In particular, in the plot on the left, we can see that enabling all 32 cores of the Ascend 910 causes a dramatic drop in performance when using a fixed query compared to using a uniformly distributed one. On the contrary, from the plot on the right, it emerges that enabling only one core not only removes completely the cache conflicts as one could expect but also decreases the average latency making it less than half when using a fixed query compared to using a uniformly distributed one. This is explained by the much higher effectiveness of the cache usage when using a fixed query, as we are accessing the same address repeatedly. More importantly, the three proposed strategies, GM-UB, L1, and L1-UB, show the same performance for both considered distributions. This demonstrates not only the absence of cache conflicts, which was expectable by design, but also the full independence from the input query distribution, a very desirable and essential property as the access patterns of look-ups represent a big issue for existing approaches ([23, 24, 32]).

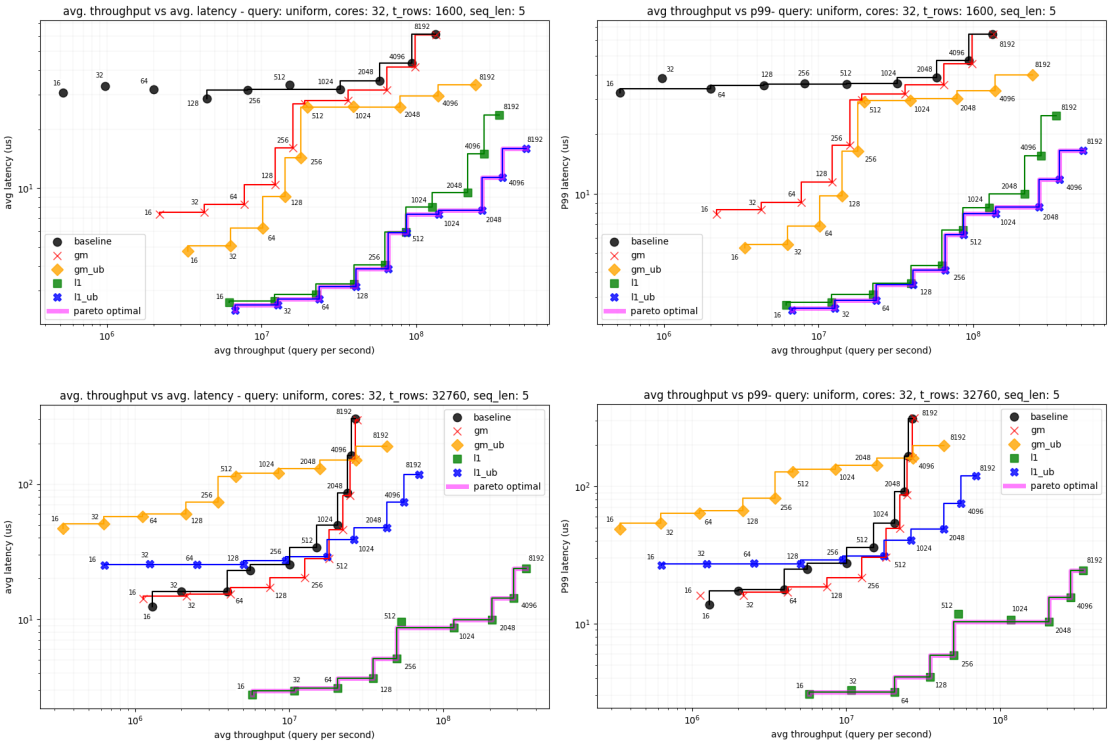

Figure 5.13: Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) table with 1600 rows. (bottom) table with 32760 rows. The aggregation refers to 200 runs on the Ascend 910 performing pooling-based lookups using a uniform query distribution with the five competing strategies, a sequence length set to 5, and enabling all 32 cores. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy.

5.3.3 Throughput vs. latency trade-offs

Even though the average latency represents a good indicator of the performance of the proposed strategies, as we have already mentioned in section 5.1.1, in a real deployment scenario, the throughput achieved with a specific target latency is also a critical metric for achieving a good hardware utilization, and so a good energy footprint. To this extent, we again performed multiple experiments on a single table exploiting the Ascend 910. In particular, we run 200 iterations using all the competing strategies for each configuration tried, where each configuration is characterized by a fixed set of values for the table parameters, while the batch size is changed to evaluate the reachable trade-offs between throughput and latency. More specifically, we are interested in assessing the trade-offs considering both

the average latency, which is a good indicator of the average user experience, and the P-99 worst-case latency, necessary in real-time systems released under SLA requirements.

Concerning the trade-offs with differently sized tables, figure 5.13 shows that with a table made of 1600 rows (50 KB), so relatively small, the L1-UB strategy provides the best trade-offs and match exactly the Pareto front, both in terms of average latency and P-99 worst-case latency. This is expectable since having a small table means making a small number of chunk movements from L1 to UB. Furthermore, we can see that the L1 strategy is the next optimal one, delivering very similar performance trade-offs compared to the L1-UB strategy up to a batch size of 1024, while the gap increases as the batch size grows. This is because the L1-UB strategy pays the cost of moving the chunks of the table only once to be able to perform the lookups directly from UB in a vectorized way, which is faster than performing individual lookups from L1. For the same reason, we can also see that not only the GM-UB strategy consistently outperforms the GM one, but again the gap between the two increases as the batch size grows. The baseline provides the worst throughputs and latencies overall, having all its sampled points Pareto-dominated by all the other strategies, thus demonstrating to be far from optimal, at least for small tables.

Still referring to figure 5.13, considering now a larger table made of 32760 rows ($\approx$ 1 MB), we can see that the relative trade-offs between the strategies change drastically. First, the L1 strategy becomes optimal, consistently outperforming all the others, including the L1-UB one, which was the best for a smaller table. This is because having a table 20x larger than before leads to many more chunk movements, which are costly operations. Moreover, in this case, the baseline and the GM strategy show good performance trade-offs, outperforming the GM-UB and the L1-UB up to specific batch sizes due to the many chunk movements already discussed. On the contrary, increasing the batch size leads the GM-UB and L1-UB strategies to outperform the baseline and the GM one again. More specifically, the L1-UB one outperforms the others, except for the L1 strategy, starting from a batch size equal to 512.

Considering the two analyzed tables, the gap between the optimal strategy, the L1 or L1-UB strategies, is very big and gets even more accentuated when considering the larger table. However, it is worth noting that the optimality of the L1 strategies also comes with constraints. In fact, they are based on persistently preloading the table in the L1 buffer, thus being feasible only if there is enough space left, considering that we have many tables. Moreover, the huge gap in performance with the other strategies provides an immediate view of the penalty that we pay if we use other strategies, making the choice of which tables to persist

in L1 critically important as a consequence.

The trade-offs achieved considering the average latency and P-99 worst-case latency are very similar and always lead to the same Pareto-optimal strategy. This means that choosing a strategy based on a target P-99 worst-case latency bound gives not only a good average throughput but also a good average latency, which means providing a good user experience on average.

5.4 Model-level policies assessment

From the results discussed in section 5.3, it is unquestionable that a globally optimal strategy for every use case does not exist, but instead, there exists an optimal strategy for a given use case. For this reason, finding an optimal policy, that is, choosing which strategy to use for each table, is critically important. To this extent, we evaluated through multiple experiments the performance given by the optimal policies obtained with the optimization algorithms described in sections 4.4 and 4.5. In particular, one policy refers to using the SIMD paradigm, while the other concerns the MIMD one. The two will be referred to as the symmetric and asymmetric strategies. We again used the Ascend 910 AI accelerator and the DCNv2 model used in production as a benchmark, running the optimal policies found and the baseline for 100 iterations with queries sampled from two distributions, a fixed one and a uniformly distributed one.

Policy statistics Before comparing the performance achieved by the two policies, it is useful to analyze how they assign an optimal strategy to each embedding table composing the DCNv2 model. As shown in figure 5.14, the percentage of tables assigned to each strategy varies according to the number of available cores and batch size. In particular, the asymmetric policy assigns more than 58% of the tables to the L1-UB strategy when using 8 cores, and this figure increases to more than 65% with 32 cores available. On the contrary, only less than 1% of the tables are assigned to the L1 strategy. This finds an explanation in figure 5.1, where it is evident that the majority of the tables of the model have less than 2000 rows, and according to figure 5.13, tables with such a small amount of rows perform way better when using the L1-UB strategy compared to the others. The increase in the percentage when considering 32 cores w.r.t 8 cores is simply due to the fact that having 32 cores means having an aggregated L1 buffer which is four times larger. Furthermore, still referring to the optimal asymmetric policy, figure 5.14 also shows that a percentage between 34% and 40% of the tables are assigned to the GM-UB strategy when using 8 cores, and that this figure decreases to less than 30% when enabling all 32 cores.

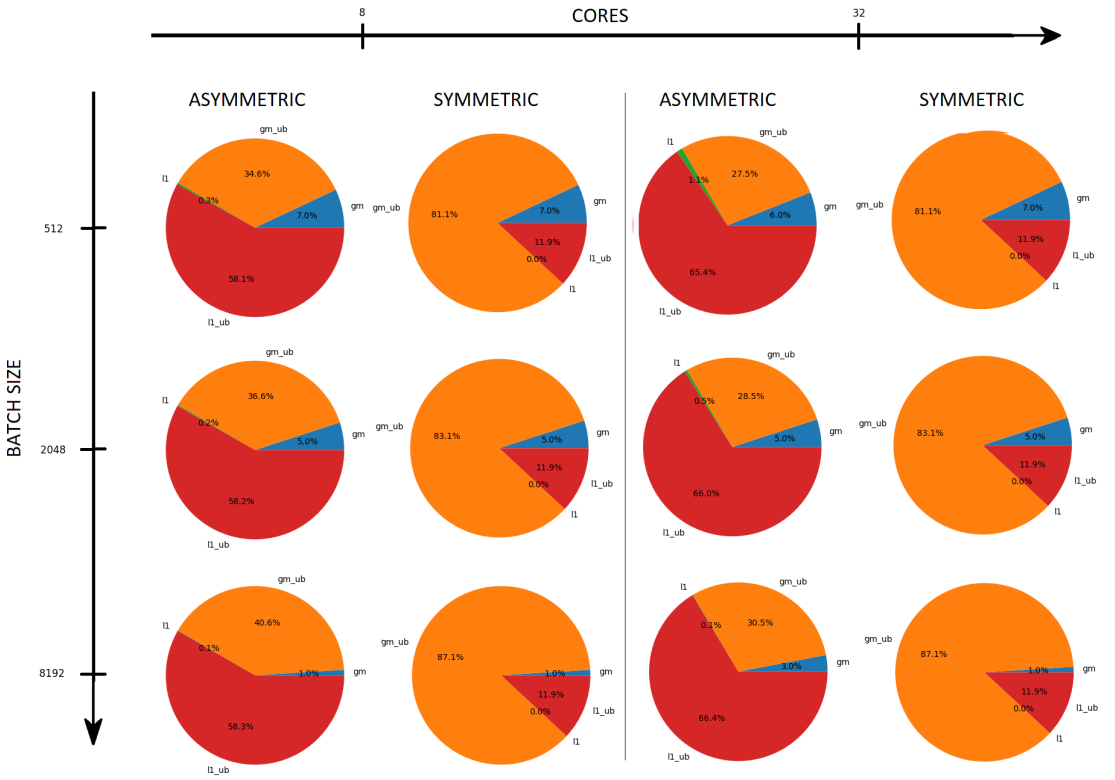

Figure 5.14: Each pie chart refers to the percentage of tables of the DCNv2 model considering the Ascend 910 and assigned by the policy for being computed with a specific strategy. In particular, there are two kinds of policy: asymmetric, which tries to assign the optimal strategies to get the most out of the MIMD paradigm, and symmetric, which instead aims at finding an optimal assignment of the strategies for the SIMD paradigm. From left to right, we see the effects of having fewer or more cores available, and from top to bottom, the batch size affects the number of lookups performed, thus resulting in a different policy.

Even though the L1 strategies perform much better than the GM-UB one when using small tables, the L1 buffer has a limited size, and therefore only a limited number of tables can be preloaded there. For this reason, when the L1 strategies are not available, the GM-UB one is much faster than the standard gm one for small tables, as shown in figure 5.13.
Still referring to the asymmetric policy, as the batch size increases, the percentages related to the GM and L1 strategies decrease while favoring a higher percentage for the respective UB versions. Again, as we have already discussed in section 5.3.3, the reason lies in the fact that the UB strategies perform much better as the batch size increases since the cost of moving the chunks of the table in UB is paid only once.

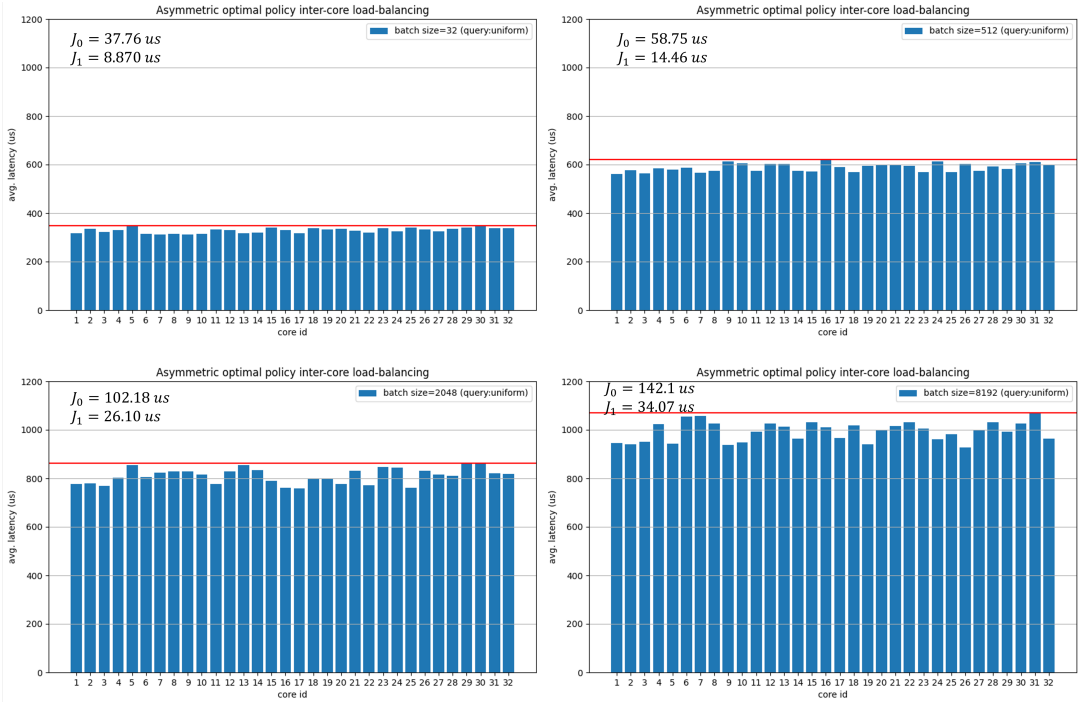

Figure 5.15: Inter-core load-balancing assessment of the optimal asymmetric policy applied on the embedding tables of the DCNv2 model used in production. The values are the average latencies measured executing only with one core the set of tables with the strategies defined by the optimal policy. Here we refer to feeding the model queries sampled from a uniform distribution.

Concerning the symmetric policy, it is worth recalling that the workload is split along the batch size dimension evenly across the cores, thus resulting in the impossibility of putting different tables in different cores. As a result, each core has assigned the same set of tables to process, as well as the same set of strategies defined by the policy. Besides, one significant difference shown by the symmetric policy compared to the asymmetric one is represented by the much lower percentage of the tables preloaded in L1. This is due to the fact that it can use the size of the L1 buffer of only one core compared to the aggregated L1 buffer usable by the asymmetric one. Consequently, due to the lack of enough L1 memory, the decrease in the percentage of the usage of the L1 strategies lead to an increase in the usage of the GM-UB one, which represents more than 81% of the total usage.

Asymmetric load-balancing assessment Before making the final comparison between the strategies using the DCNv2 model, it is important to evaluate to

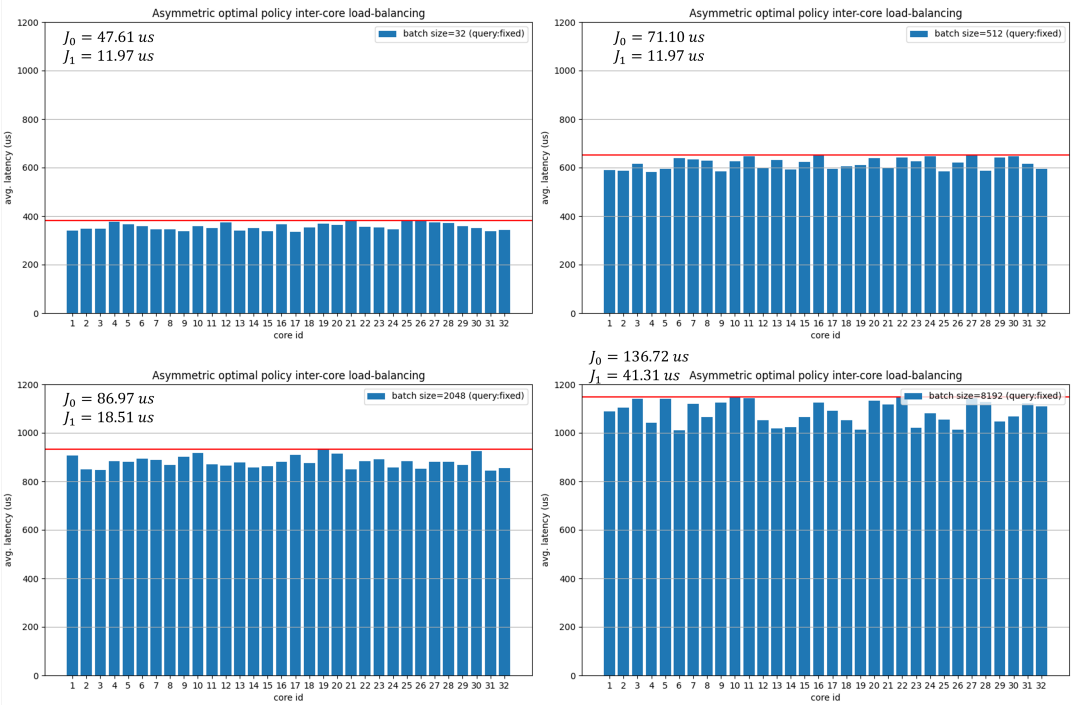

Figure 5.16: Inter-core load-balancing assessment of the optimal asymmetric policy applied on the embedding tables of the DCNv2 model used in production. The values are the average latencies measured executing only with one core the set of tables with the strategies defined by the optimal policy. Here we are referring to feeding to the model queries made of indices fixed to the value 0.

what extent the optimal asymmetric policy obtained with the greedy strategy described in section 4.5 provides a balanced workload among the cores. Apart from a qualitative assessment of load balancing, to evaluate more precisely the extent of the unbalancing, given the number of cores K, and the set of costs $C = \{c_0, c_1, \ldots, c_{K-1}\}$, each representing the average latency of each core, we define two objectives,

$$J_0 = \max_{0 \leq c_i \leq K-1} c_i - \min_{0 \leq c_i \leq K-1} c_i \tag{5.1}$$

$$J_1 = \frac{1}{K} \sum_{i=0}^{K-1} \left| c_i - \left(\frac{1}{K} \sum_{j=0}^{K-1} c_j \right) \right| \tag{5.2}$$

The first objective, J_0, represents the difference between the maximum and the minimum, showing immediately the extent of the absolute unbalancing, whereas the second objective, J_1, shows the mean absolute deviation, that is, how spread the costs are with respect to the average value, giving an intuition of how the workload

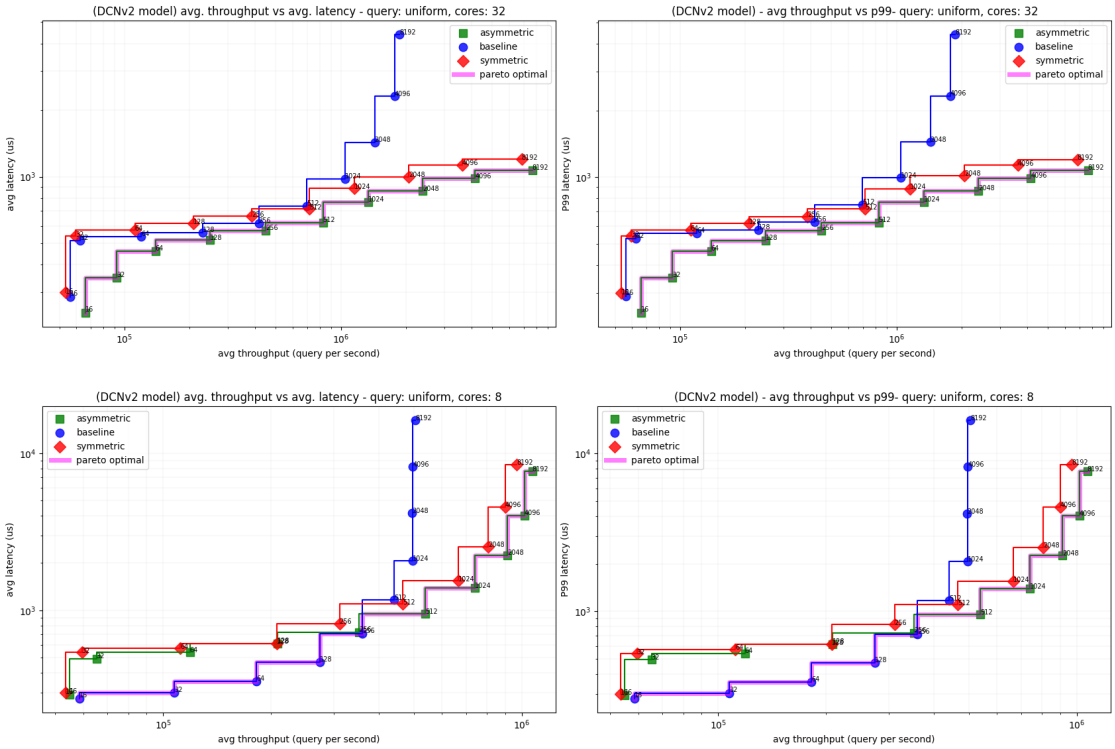

Figure 5.17: Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) enabling all 32 cores. (bottom) enabling only 8 cores. The aggregation refers to 200 runs of the DCNv2 model used in production on the Ascend 910, using the symmetric, asymmetric, and baseline approaches. The queries are sampled from a uniform distribution. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy.

of all the cores differ from one another. Notice that the standard deviation gives another valuable possibility, but we chose to focus on the mean absolute deviation to be less influenced by large outliers. Figures 5.15 and 5.16 show the workload of each core according to the asymmetric policy, as well as the values of the two objectives for the specific use cases. For both input distributions considered, we can see that increasing the batch size leads to a more unbalanced scenario, doubling or tripling the values of the two objectives when using a batch size of 8192 compared to a batch size equal to 32. Still, overall the balancing achieved is good, considering the complexity of the partitioning problem.

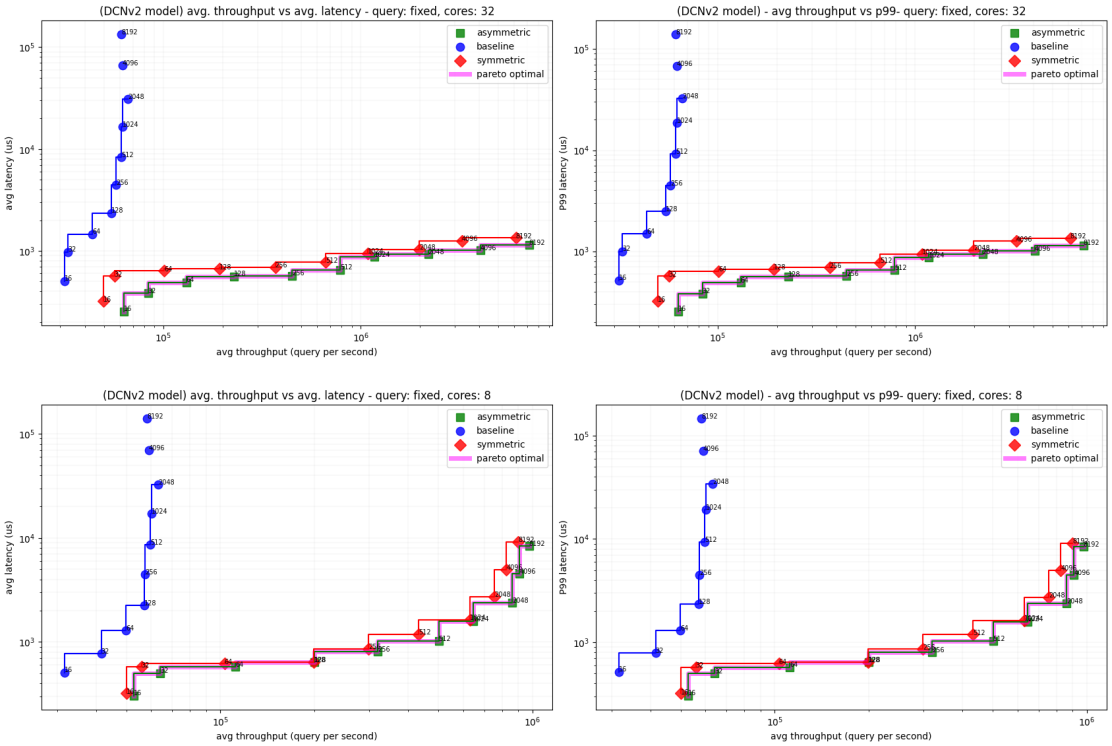

Figure 5.18: Avg. throughput vs. avg. latency (left). Avg. throughput vs. P-99 worst-case latency (right). (top) enabling all 32 cores. (bottom) enabling only 8 cores. The aggregation refers to 200 runs of the DCNv2 model on the Ascend 910, using the symmetric, asymmetric, and baseline approaches. The query indices are all set to the value 0 to test the problem of cache conflicts. The varying parameter is the batch size, and each sampled point in the plot reports its value. The thicker purple line refers to the global Pareto front considering all the strategies, whereas the others are the Pareto fronts considering only the related strategy.

Model-level comparison Having acquired a general understanding of how the policies allocate the strategies to the embedding tables of the DCNv2 model, as well as having assessed the performance of the single strategies on differently shaped tables, we finally want to compare the end-to-end performance achieved by the proposed policies compared to the baseline. To this extent, we analyzed the trade-offs between the average throughput and average latency, as well as the P-99 worst-case latency, using the two input distributions described so far in two different settings, one consisting in enabling all 32 cores and the other where only 8 cores are available, thus emulating the performance of a smaller accelerator with a 4x smaller aggregate L1 buffer.

Concerning the usage of a uniform query distribution, figure 5.17 shows a comprehensive assessment of the previously mentioned trade-offs. First, when all 32 cores are available, the asymmetric policy outperforms both the baseline and the symmetric policy by a convincing margin for all the batch sizes considered. We argue that the better performance of the asymmetric policy is accountable to the much higher percentage of tables being preloaded in the L1 buffer, as shown in figure 5.14 and already discussed in paragraph 5.4. In fact, as can be seen in figure 5.13, both small and large tables provide a much faster lookup operation when the L1-UB and L1 strategies are used compared to all the others. This also explains why the asymmetric policy consistently outperforms the symmetric one, as it can benefit from an aggregated L1 buffer 32x larger than the one available to the symmetric policy. Furthermore, the gap between the baseline and the proposed policies grows considerably starting from a batch size of 4096 due to the much higher throughput given by the GM-UB strategy when a large batch is used, as shown in figure 5.13. Not less important is the fact that the baseline starts to reach the maximum throughput capacity when the batch size is equal to 8192, whereas the two policies continue to provide significant improvements to the throughput thanks to the more amortization of the cost paid once to move chunks of tables when using the UB strategy. Besides, we can see that the symmetric policy provides slightly worse trade-offs compared to the baseline up to a batch size of 256, which is expectable since more than 80% of the tables exploit the GM-UB strategy according to the policy.

Considering instead a scenario with only 8 cores, the trade-offs shown by the strategies change. First, it is immediately visible how the baseline provides much better trade-offs than both policies up to a batch size of 256. On the contrary, starting from a batch size of 512, the two policies deliver much better trade-offs than the baseline, which instead quickly reaches the maximum throughput capacity. This is accountable to the much less aggregated L1 space available to the asymmetric policy, which relies heavily on it to deliver good performance. Instead, we can see that the gap between the baseline and the symmetric policy increases a lot up to a batch size of 256, probably due to optimizations introduced by the baseline strategy when using a lower number of cores. Once again, the asymmetric policy shows better trade-offs than the symmetric one for all the batch sizes considered, thus demonstrating its superiority, most probably given by the much better usage of the L1 buffer of each core.

Instead, the situation changes drastically when considering a fixed input distribution, as expectable for the cache conflicts that we have already described in section 5.2. In fact, as shown in figure 5.18, the baseline approaches the maximum throughput capacity quite early when the batch size is equal only to 128. Even

worse, the baseline starts even to lose throughput for batch sizes larger than 2048, showing not only its inadequacy to handle properly fixed input queries due to the cache conflicts but also the degeneration of the problem as the amount of samples increases. On the contrary, once again, the strategies handle the fixed queries without problems, delivering almost the same trade-offs shown for the uniform distribution.

Recalling that the baseline experiences an increase in up to 24x of the average latency when using fixed queries, as shown in figure 5.7, it is worth quantifying the improvement introduced by the proposed policies. Thus, from table 5.1, we can see that considering all 32 cores enabled and fixed queries, the baseline completes the entire pooling-based lookup operations of all the tables in around $1000\mu s$ on average with a batch size of 32, compared to the $567\mu s$ and $382\mu s$ taken by the symmetric and asymmetric policies respectively, which thus show to be 1.5x and 2.6x faster than the baseline. On the other extreme, considering a batch size of 8192, the gap drastically increases, showing that the baseline is taking more than $133ms$ compared to the $1.3ms$ and $1.1ms$ taken by the symmetric and asymmetric policies, respectively, thus reflecting 102x and 116x faster executions.
Concerning instead the execution of the pooling-based lookup operations sampling the queries from a uniform distribution, we can see that the baseline takes around $4.3ms$ on average with a batch size of 8192, whereas the symmetric and asymmetric policies take around $1.2ms$ and $1.1ms$, which reflect roughly 3.5x faster execution.

strategy	query	cores	p99 (B-32)	avg (B-32)	p99 (B-8192)	avg (B-8192)
baseline	U	32	526.8	513.1	4410.0	4396.8
symmetric	U	32	540.18	538.8	1199.5	1198.9
asymmetric	U	32	**350.2**	**348.5**	**1075.1**	**1070.1**
baseline	F	32	1000.0	973.5	139960.3	133717.1
symmetric	F	32	567.2	567.1	1339.6	1339.5
asymmetric	F	32	**382.6**	**382.6**	**1146.9**	**1146.8**

Table 5.1: The table shows the performance achieved on the DCNv2 model on the Ascend 910 using different strategies. In particular, the values refer to the P99 worst-case latency and average latency expressed in μs using different batch sizes. The values refer to 200 different runs. 'U' stands for uniform distribution, whereas 'F' stands for fixed distribution.

Chapter 6

Conclusions and future works

The thesis focuses on the study and design of custom data flow strategies for the Ascend AI accelerators with the goal of accelerating the inference of Deep Learning-based Recommender Models (DLRMs). This is justified as many production-scale web services, such as Youtube, Amazon, or Spotify, rely on DRLMs for providing effective personalized recommendations to their users, thus causing, for instance, that more than 79% of the AI workload in Meta's data centers is represented by the inference of DLRMs, as underlined in [1]. Despite the existence of a plethora of optimizations already available for improving the efficiency of deep learning workloads, unfortunately, they are not applicable to DLRMs due to their unique characteristics. In particular, DLRMs base their effectiveness on learning powerful representations both from dense and categorical features associated with users and items. As a consequence, since the categorical features are represented as one-hot encoded vectors, which induce an extremely sparse vector space, DLRMs make extensive use of embedding layers to map the sparse vectors to dense, continuous vectors of fixed size, called embedding vectors. Even though this mapping solves the sparsity problem, letting DLRMs achieve outstanding results in providing effective personalized recommendations, the embedding layers introduce a big performance bottleneck. In fact, embedding vectors are organized in tables, which range from a few KBs to hundreds of GBs, stored persistently in memory, and used during inference to perform look-ups to retrieve the embedding vectors. As a consequence, given that there are tens, or in some cases even hundreds of embedding tables, the large amount of look-ups that must be performed is large, inducing a large number of random memory accesses to retrieve many small embedding vectors. As underlined by [2], this represents a bottleneck for the inference performance of DLRMs, thus motivating the development of many solutions that try to mitigate

this problem. Some solutions try to exploit FPGAs with on-chip cutting edge High-Bandwidth Memory (HBM) to perform as many parallel look-ups as possible, while others propose to build clusters of heterogeneous nodes, including FPGAs, GPUs, and CPUs for separating the tasks and achieving better performance. Others try to build new caching systems, including new ideas such as modifying the cache replacement policy to better suit the access patterns of embedding look-ups. Lastly, some solutions try to build new memory chips, better suited for the look-ups, exploiting the Near-Memory Processing (NMP) idea by integrating the required operators near the memory chips to avoid the expensive data movement operations. Even though existing approaches partially mitigate the problem introduced by embedding look-ups, they are limited to specific use cases, as real-world DLRMs are characterized by highly-varying hyper-parameters. In fact, embedding tables vary both in number and dimension, and even worse, each real-world deployment scenario is characterized by a different input query distribution, which makes the access patterns unpredictable and different for every use case considered, thus making the existing strategies not always applicable and effective. For this reason, the thesis proposes to exploit AI accelerators to boost the performance of DLRMs, particularly by designing custom data flows for the Huawei Ascend AI accelerators, tailored for the embedding look-up characteristics. Moreover, since preliminary results showed that the best data flow strategy depends strongly on the characteristics of the specific embedding table, two different policies are obtained to establish the optimal strategy to apply to a given table. In particular, the thesis proposes two different approaches to split the workload across the many cores of the accelerators, one based on the single-instruction-multiple-data (SIMD) paradigm and the other based on the multiple-instruction-multiple-data (MIMD) one. The SIMD approach splits the workload evenly across the available cores along the batch size dimension, and its related policy tries to find the optimal strategy to use for each embedding table. The MIMD one executes a different program on each core, thus allowing its policy to define with arbitrary granularity, which embedding tables and portions of the batch must be processed in each core. Through extensive experiments conducted on the Ascend 910 AI accelerator, the strategies were compared with a strong baseline obtained from a smart compiler called ATC which exploits built-in operators written by experts.

Take-away messages To wrap up, a summary of the main contributions of this thesis is reported.

- First, we provided, for the first time, up to our knowledge, a comprehensive assessment of the performance of DRLMs using the Huawei Ascend AI accelerators.
- In particular, we assessed the inference performance achieved by the baseline,

which is based on the ATC compiler with built-in operators written by experts, showing that the average latency on the Ascend 910 worsens by up to 24x when using a query made of fixed indices compared to a query sampled from a uniform distribution. This demonstrates the extreme dependence of the baseline methodology on the input query distribution, making its usage potentially unsuitable for some real-world use cases.

- Thus, we proposed a completely new data flow design for Ascend accelerators, which is independent of the input query distribution, thus making it potentially a better candidate in some use cases.

- We demonstrated empirically through extensive experiments that the proposed strategy delivers much better performance compared to the baseline, achieving a 116x and 3.5x smaller average latency when using fixed indices and uniform distribution, respectively.

- Moreover, we proved that the trade-offs between average throughput and P-99 worst-case latency achieved by the proposed method are consistently better than the ones provided by the baseline.

Limitations and future works Even though the results shown by the proposed method seem promising, there is still room for large improvements. First, one major improvement could be brought by a better policy optimization algorithm to replace the current greedy one based on heuristics. Although this applies to both the proposed approaches, SIMD and MIMD, a better policy could bring much more benefits to the MIMD approach, as it involves many degrees of freedom, making the problem much harder to solve. For instance, an approach like the one used in [27, 29, 30], which exploits a deep reinforcement learning algorithm to learn the optimal policy, could be very effective in this case. Moreover, the performance model utilized to estimate the performance of the given policies could be also improved. In fact, the current linear model might be inaccurate, leading to a suboptimal solution even with the best optimization algorithm. Again, an approach like the one proposed in [27] could work well, consisting in learning a simple DNN for estimating the performance of the single embedding tables. Furthermore, since this thesis proposes only horizontal cuts of the tables to create chunks with the MIMD approach, a possible extension is the support of vertical cuts, thus making the strategy fully general and potentially more effective in some use cases. Lastly, given the fact that the proposed data flow strategies are designed to be possibly scaled out to multiple devices, we leave the adaptation of the implementation for supporting this possibility to future works. In particular, one possibility could be extending the policy of the MIMD approach to multiple devices in order to

achieve an aggregate L1 buffer with an arbitrary size, thus allowing to scale out the performance achieved based on the number of devices used.

Bibliography

[1] Udit Gupta et al. «The Architectural Implications of Facebook's DNN-Based Personalized Recommendation». In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. San Diego, CA, USA: IEEE, Feb. 2020, pp. 488–501. ISBN: 978-1-72816-149-5. DOI: 10.1109/HPCA47549.2020.00047. (Visited on 07/10/2023) (cit. on pp. 1, 6, 12, 38, 83).

[2] 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 the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 740–753. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358284. (Visited on 07/11/2023) (cit. on pp. 2, 28, 30, 31, 68, 83).

[3] Zied Zaier, Robert Godin, and Luc Faucher. «Evaluating Recommender Systems». In: *2008 International Conference on Automated Solutions for Cross Media Content and Multi-Channel Distribution*. Florence, Italy: IEEE, Nov. 2008, pp. 211–217. ISBN: 978-0-7695-3406-0. DOI: 10.1109/AXMEDIS.2008.21. (Visited on 07/10/2023) (cit. on p. 7).

[4] Yehuda Koren, Robert Bell, and Chris Volinsky. «Matrix Factorization Techniques for Recommender Systems». In: *Computer* 42.8 (Aug. 2009), pp. 30–37. ISSN: 0018-9162. DOI: 10.1109/MC.2009.263. (Visited on 07/10/2023) (cit. on p. 8).

[5] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. «Deep & Cross Network for Ad Click Predictions». In: *Proceedings of the ADKDD'17*. Halifax NS Canada: ACM, Aug. 2017, pp. 1–7. ISBN: 978-1-4503-5194-2. DOI: 10.1145/3124749.3124754. (Visited on 07/10/2023) (cit. on p. 9).

[6] Ruoxi Wang, Rakesh Shivanna, Derek Z. Cheng, Sagar Jain, Dong Lin, Lichan Hong, and Ed H. Chi. «DCN V2: Improved Deep & Cross Network and Practical Lessons for Web-scale Learning to Rank Systems». In: *Proceedings of the Web Conference 2021*. Apr. 2021, pp. 1785–1797. DOI: 10.1145/

3442381.3450078. arXiv: 2008.13535 [cs, stat]. (Visited on 07/10/2023) (cit. on pp. 9, 10).

[7] Heng-Tze Cheng et al. «Wide & Deep Learning for Recommender Systems». In: *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. Boston MA USA: ACM, Sept. 2016, pp. 7–10. ISBN: 978-1-4503-4795-2. DOI: 10.1145/2988450.2988454. (Visited on 07/10/2023) (cit. on pp. 9, 10).

[8] Xiaopeng Li and James She. «Collaborative Variational Autoencoder for Recommender Systems». In: *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Halifax NS Canada: ACM, Aug. 2017, pp. 305–314. ISBN: 978-1-4503-4887-4. DOI: 10.1145/3097983.3098077. (Visited on 07/11/2023) (cit. on p. 10).

[9] Qiang Cui, Shu Wu, Qiang Liu, Wen Zhong, and Liang Wang. «MV-RNN: A Multi-View Recurrent Neural Network for Sequential Recommendation». In: *IEEE Transactions on Knowledge and Data Engineering* 32.2 (Feb. 2020), pp. 317–331. ISSN: 1041-4347, 1558-2191, 2326-3865. DOI: 10.1109/TKDE.2018.2881260. (Visited on 07/11/2023) (cit. on p. 10).

[10] N. Nikzad-Khasmakhi, M.A. Balafar, M. Reza Feizi-Derakhshi, and Cina Motamed. «BERTERS: Multimodal Representation Learning for Expert Recommendation System with Transformers and Graph Embeddings». In: *Chaos, Solitons & Fractals* 151 (Oct. 2021), p. 111260. ISSN: 09600779. DOI: 10.1016/j.chaos.2021.111260. (Visited on 07/11/2023) (cit. on p. 10).

[11] Norman P. Jouppi et al. «In-Datacenter Performance Analysis of a Tensor Processing Unit». In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Toronto ON Canada: ACM, June 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246. (Visited on 07/11/2023) (cit. on pp. 14, 15).

[12] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. «NVIDIA Tensor Core Programmability, Performance & Precision». In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Vancouver, BC, Canada: IEEE, May 2018, pp. 522–531. ISBN: 978-1-5386-5555-9. DOI: 10.1109/IPDPSW.2018.00091. (Visited on 07/11/2023) (cit. on pp. 14, 15).

[13] Jean-Philippe Fricker. «The Cerebras CS-2: Designing an AI Accelerator around the World's Largest 2.6 Trillion Transistor Chip». In: *Proceedings of the 2022 International Symposium on Physical Design*. Virtual Event Canada: ACM, Apr. 2022, pp. 71–71. ISBN: 978-1-4503-9210-5. DOI: 10.1145/3505170.3511036. (Visited on 07/11/2023) (cit. on p. 15).

[14] Dennis Abts et al. «Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads». In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 145–158. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00023. (Visited on 07/11/2023) (cit. on p. 15).

[15] Eitan Medina and Eran Dagan. «Habana Labs Purpose-Built AI Inference and Training Processor Architectures: Scaling AI Training Systems Using Standard Ethernet With Gaudi Processor». In: *IEEE Micro* 40.2 (Mar. 2020), pp. 17–24. ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2020.2975185. (Visited on 07/11/2023) (cit. on p. 15).

[16] David R. Ditzel. «Accelerating ML Recommendation With Over 1,000 RISC-V/Tensor Processors on Esperanto's ET-SoC-1 Chip». In: *IEEE Micro* 42.3 (May 2022), pp. 31–38. ISSN: 0272-1732, 1937-4143. DOI: 10.1109/MM.2022.3140674. (Visited on 07/10/2023) (cit. on pp. 15, 27, 28, 31).

[17] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. «DaVinci: A Scalable Architecture for Neural Network Computing». In: *2019 IEEE Hot Chips 31 Symposium (HCS)*. Cupertino, CA, USA: IEEE, Aug. 2019, pp. 1–44. ISBN: 978-1-72812-089-8. DOI: 10.1109/HOTCHIPS.2019.8875654. (Visited on 07/11/2023) (cit. on pp. 16–18).

[18] 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: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 982–995. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00084. (Visited on 07/10/2023) (cit. on pp. 28, 30, 40).

[19] Yu Zhu, Zhenhao He, Wenqi Jiang, Kai Zeng, Jingren Zhou, and Gustavo Alonso. «Distributed Recommendation Inference on FPGA Clusters». In: *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. Dresden, Germany: IEEE, Aug. 2021, pp. 279–285. ISBN: 978-1-66543-759-2. DOI: 10.1109/FPL53798.2021.00057. (Visited on 07/10/2023) (cit. on pp. 28, 30).

[20] Wenqi Jiang et al. «FleetRec: Large-Scale Recommendation Inference on Hybrid GPU-FPGA Clusters». In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. Virtual Event Singapore: ACM, Aug. 2021, pp. 3097–3105. ISBN: 978-1-4503-8332-5. DOI: 10.1145/3447548.3467139. (Visited on 07/10/2023) (cit. on pp. 28, 30, 40).

[21] Wenqi Jiang et al. «MicroRec: Efficient Recommendation Inference by Hardware and Data Structure Solutions». In: ETH Zurich, Apr. 2021. DOI: 10.3929/ETHZ-B-000470540. (Visited on 07/10/2023) (cit. on pp. 28, 29, 57).

[22] Liu Ke et al. «RecNMP: Accelerating Personalized Recommendation with Near-Memory Processing». In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, May 2020, pp. 790–803. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00070. (Visited on 07/10/2023) (cit. on pp. 28, 30, 31).

[23] Hongju Kal, Seokmin Lee, Gun Ko, and Won Woo Ro. «SPACE: Locality-Aware Processing in Heterogeneous Memory for Personalized Recommendations». In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, June 2021, pp. 679–691. ISBN: 978-1-66543-333-4. DOI: 10.1109/ISCA52012.2021.00059. (Visited on 07/10/2023) (cit. on pp. 28, 38, 72).

[24] Daniar H. Kurniawan, Ruipu Wang, Kahfi S. Zulkifli, Fandi A. Wiranata, John Bent, Ymir Vigfusson, and Haryadi S. Gunawi. «EVStore: Storage and Caching Capabilities for Scaling Embedding Tables in Deep Recommendation Systems». In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Vancouver BC Canada: ACM, Jan. 2023, pp. 281–294. ISBN: 978-1-4503-9916-6. DOI: 10.1145/3575693.3575718. (Visited on 07/10/2023) (cit. on pp. 32, 41, 72).

[25] Sikha Bagui and Loi Tang Nguyen. «Database Sharding: To Provide Fault Tolerance and Scalability of Big Data on the Cloud». In: *International Journal of Cloud Applications and Computing* 5.2 (Apr. 2015), pp. 36–52. ISSN: 2156-1834, 2156-1826. DOI: 10.4018/IJCAC.2015040103. (Visited on 07/11/2023) (cit. on p. 33).

[26] Faiza Hashim, Khaled Shuaib, and Nazar Zaki. «Sharding for Scalable Blockchain Networks». In: *SN Computer Science* 4.1 (Oct. 2022), p. 2. ISSN: 2661-8907. DOI: 10.1007/s42979-022-01435-z. (Visited on 07/11/2023) (cit. on p. 33).

[27] Daochen Zha et al. «AutoShard: Automated Embedding Table Sharding for Recommender Systems». In: *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Washington DC USA: ACM, Aug. 2022, pp. 4461–4471. ISBN: 978-1-4503-9385-0. DOI: 10.1145/3534678.3539034. (Visited on 07/10/2023) (cit. on pp. 34, 49, 54, 85).

[28] Daochen Zha, Louis Feng, Qiaoyu Tan, Zirui Liu, Kwei-Herng Lai, Bhargav Bhushanam, Yuandong Tian, Arun Kejariwal, and Xia Hu. *DreamShard: Generalizable Embedding Table Placement for Recommender Systems*. Oct. 2022. arXiv: 2210.02023 [cs]. (Visited on 07/10/2023) (cit. on p. 34).

[29] Geet Sethi, Pallab Bhattacharya, Dhruv Choudhary, Carole-Jean Wu, and Christos Kozyrakis. *FlexShard: Flexible Sharding for Industry-Scale Sequence Recommendation Models*. Jan. 2023. arXiv: 2301.02959 [cs]. (Visited on 07/10/2023) (cit. on pp. 35, 49, 85).

[30] Tamer Eldeeb, Zhengneng Chen, Asaf Cidon, and Junfeng Yang. «Neuroshard: Towards Automatic Multi-Objective Sharding with Deep Reinforcement Learning». In: *Proceedings of the Fifth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. Philadelphia Pennsylvania: ACM, June 2022, pp. 1–12. ISBN: 978-1-4503-9377-5. DOI: 10.1145/3533702.3534908. (Visited on 06/25/2023) (cit. on pp. 35, 54, 85).

[31] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J. Nair. «Accelerating Recommendation System Training by Leveraging Popular Choices». In: *Proceedings of the VLDB Endowment* 15.1 (Sept. 2021), pp. 127–140. ISSN: 2150-8097. DOI: 10.14778/3485450.3485462. (Visited on 07/10/2023) (cit. on pp. 35, 38, 41).

[32] Samuel Hsia, Udit Gupta, Mark Wilkening, Carole-Jean Wu, Gu-Yeon Wei, and David Brooks. «Cross-Stack Workload Characterization of Deep Recommendation Systems». In: *2020 IEEE International Symposium on Workload Characterization (IISWC)*. Beijing, China: IEEE, Oct. 2020, pp. 157–168. ISBN: 978-1-72817-645-1. DOI: 10.1109/IISWC50251.2020.00024. (Visited on 07/10/2023) (cit. on pp. 38, 72).

[33] Koen Goetschalckx and Marian Verhelst. «Breaking High-Resolution CNN Bandwidth Barriers With Enhanced Depth-First Execution». In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (June 2019), pp. 323–331. ISSN: 2156-3357, 2156-3365. DOI: 10.1109/JETCAS.2019.2905361. (Visited on 07/10/2023) (cit. on p. 39).

[34] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. «Efficient Processing of Deep Neural Networks: A Tutorial and Survey». In: *Proceedings of the IEEE* 105.12 (Dec. 2017), pp. 2295–2329. ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2017.2761740. (Visited on 07/10/2023) (cit. on p. 39).