DELFT UNIVERSITY OF TECHNOLOGY

MASTER'S THESIS

Cost Estimation for Factorized Machine Learning

Author:
Pepijn te MARVELDE

Supervisors:
Dr. Rihan HAI
Wenbo SUN

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

Student Number:	4886496	
Thesis Committee:	Dr. A. Katsifodimos	TU Delft, Chair
	Dr. R. Hai	TU Delft, Supervisor
	Dr. S.S. Chakraborty	TU Delft

April 14, 2024

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science

Software Technology

Master of Science

Cost Estimation for Factorized Machine Learning

by Pepijn te MARVELDE

In the realm of machine learning (ML), the need for efficiency in training processes is paramount. The conventional first step in an ML workflow involves collecting data from various sources and merging them into a single table, a process known as materialization, which can introduce inefficiencies caused by redundant data. Factorized ML strives to reduce this by maintaining the original data forms and performing model training on the separate source tables. This approach can lead to significant increases in training efficiency.

However, factorized training does not always reduce cost compared to traditional materialized training. This research tackles this issue by examining the multidimensional cost optimization problem that emerges when deciding between factorized and traditional materialized learning methods. It fills in gaps left by prior research, which is focused on CPU-based training, by investigating the cost estimation landscape for factorized ML, with a special emphasis on GPU performance compared to CPUs. The used factorized ML framework is expanded to incorporate GPU training, a topic not explored in previous research. We demonstrate that GPU training exhibits significantly different cost characteristics than CPU training, which has substantial implications for the design of cost models and the optimization of factorized ML.

Through an empirical study, an ML-based cost model is developed that can accurately predict the faster training method for a wide range of scenarios. On an extensive evaluation with real-world datasets this model boasts an average speedup of $3.8\times$, versus the state-of-the-art's $0.9\times$. We also show that it generalizes to scenarios with datasets and hardware settings on which the model is not trained, keeping 82% of training set performance.

Our innovative cost model for factorized ML enables significant time savings in training-intensive scenarios and further underlines the benefits of factorized training. However, effort should be invested into incorporating factorized training into existing ML frameworks so this method of training a model, and our cost model, can be evaluated in a larger set of realistic scenarios.

Acknowledgements

I would like to express my heartfelt gratitude to Dr. Rihan Hai for their guidance and encouragement as my supervisor throughout this thesis. I am also very thankful for Wenbo Sun's constructive feedback, valuable insights, and the guidance he provided. Their ideas and perspectives greatly enriched this study. I also express my gratitude to Dr. Asterios Katsifodimos and Dr. Soham Chakraborty for serving on my thesis committee.

My sincere thanks go to my family and friends for their unwavering support, patience, and understanding during this thesis. Their encouragement and occasional distractions were highly appreciated.

Lastly, the research reported in this work was partially facilitated by computational resources and support of the Delft AI Cluster (DAIC) at TU Delft (https://daic.tudelft.nl/), but remains the sole responsibility of the authors, not the DAIC team.

Contents

List of Figures

List of Tables

List of Abbreviations

(G)NMF	(**G**aussian)-**N**on-negative **M**atrix **F**actorization
CPU	**C**entral **P**rocessing **U**nit
DI	**D**ata **I**ntegration
F/M	**F**actorized/**M**aterialized
FK	**F**oreign **K**ey
PK	**P**rimary **K**ey
FLOP	**FL**oating-point **OP**eration
GPU	**G**raphics **P**rocessing **U**nit
LA	**L**inear **A**lgebra
ML	**M**achine **L**earning
NCU	**N**sight **C**omp**u**te
SM	**S**treaming **M**ultiprocessor
SOTA	**S**tate **O**f **T**he **A**rt
Proposed cost model types	
ANLY	**AN**a**LY**tical
LINR	**LIN**ear **R**egression
XGB	**X**treme **G**radient **B**oosting

List of Symbols

σ	Selectivity
r_t	Number of rows (samples/tuples) in table t
c_t	Number of columns (features) in table t
e_t	Sparsity (fraction of 0 values) of table t
j_t	Join type t
n	Number of base tables
ρ	Feature ratio ($\frac{c_{S_1}}{\sum_{k=2}^{n} c_k}$)
τ	Tuple ratio ($\frac{\sum_{k=2}^{n} c_k}{c_{S_1}}$)
T	Target table, result of joining tables (materialization)
S_k	Source matrix k
I_k	Indicator matrix k
M_k	Mapping matrix k

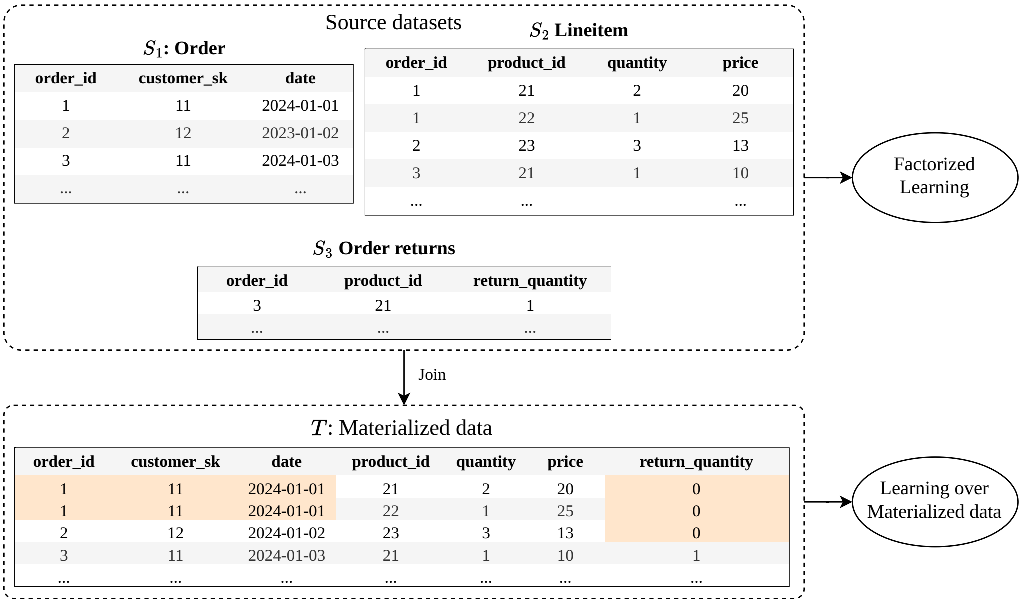
Chapter 1

Introduction

The field of Machine Learning (ML) has gained massive traction over the last few decades, with both industry and academia investing substantial efforts to enhance data availability and the performance of ML algorithms. This performance factor encompasses both the accuracy of a model and the computational and time costs associated with training these models. To make training more efficient, a novel approach called factorized learning has been proposed [1]. This approach enables models to be trained on normalized data. It is applicable to a wide range of realistic ML workflow scenarios and joinable data sources, thus opening up new possibilities for more efficient model training.

In a typical machine learning scenario, the first step for a data scientist aiming to train a model is to gather the necessary data for the training process. Often, these data come from disparate sources. Therefore, they first need to join these sources to create a single dataset to use as input for an ML model. This is achieved through a process called Data Integration (DI). The act of executing the join between the tables is called materialization [2]. Factorized learning eliminates this prerequisite to the training process by learning directly from the source datasets, without first joining them. Figure 1.1 illustrates the difference between factorized learning and learning over materialized data. The reason factorized learning can be more efficient is that the values in the materialized data (orange cells in T in the figure) do not lead to redundant computations during training. However, source datasets can also have redundant values, and this redundancy is not the only factor that affects the efficiency of factorized learning. Apart from the data-characteristics (which include redundancy), model parameters and hardware characteristics can also influence the choice between factorized learning and materialization.

Therefore, because factorization does not always increase efficiency, there is a need to be able to select those cases where factorization is beneficial. This decision between factorization and materialization is a multidimensional cost optimization problem. Creating a cost model capable of making this choice is an interesting and important problem because factorized learning is a very novel approach to the fundamentals of machine learning. It has potential to reduce the cost of model training without affecting performance. Factorized ML could also be easily extensible to federated learning in a scenario where computations involving a source dataset are executed in the silo in which the dataset is located [4].



S_1: **Order**

order_id	customer_sk	date
1	11	2024-01-01
2	12	2023-01-02
3	11	2024-01-03
...	...	...

S_2 **Lineitem**

order_id	product_id	quantity	price
1	21	2	20
1	22	1	25
2	23	3	13
3	21	1	10
...	...		...

S_3 **Order returns**

order_id	product_id	return_quantity
3	21	1
...	...	...

T: Materialized data

order_id	customer_sk	date	product_id	quantity	price	return_quantity
1	11	2024-01-01	21	2	20	0
1	11	2024-01-01	22	1	25	0
2	12	2024-01-02	23	3	13	0
3	11	2024-01-03	21	1	10	1
...	...	...	...	...	...	...

FIGURE 1.1: **Running Example: Source tables** S **and target table** T. Illustration of input data used for factorized Learning vs Learning over Materialized data, schema from TPCx-AI [3] use case 1 (unused columns not shown). Target redundancy, that is avoided by factorization, shown in orange.

However, solving this problem is challenging because the optimization space is exceptionally large and may be hardware-dependent. Previous solutions, such as Morpheus [5] and Amalur's cost estimation [6], have focused on theoretical cost or simple heuristics without considering the hardware dimension.

Figure 1.2 shows the applicability of the cost model we propose. For an ML practitioner aiming to optimize their training processes with the use of factorized learning, the data preparation and preprocessing steps do not change. Gathering source datasets and defining how to, e.g., join and clean them is still necessary. After finishing preprocessing, formalizing how the datasets should be joined, and having decided which model they want to train, our cost model predicts the optimal training method, factorized, or materialized. Using such a cost model can result in considerable time savings, particularly in scenarios where the total training time is high, such as hyperparameter tuning or training complex models. This is elaborated on in Section 5.1.

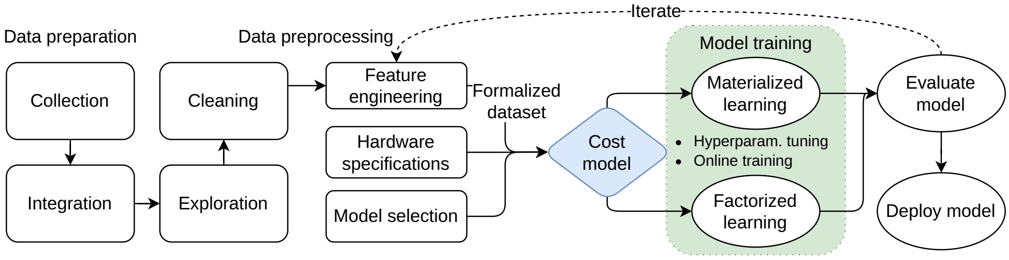


FIGURE 1.2: Function of this thesis' cost model in an ML pipeline.

1.1 Running Example

Throughout this thesis we use a running example, based on TPCx-AI [3] use case 1, to explain various concepts. The scenario is that of a data scientist working for an e-Commerce firm, who is tasked with creating a data science pipeline to perform customer segmentation. Three tables are joined to create a single table which is then used as input in a K-Means clustering model. The schema of the tables and the joined table are shown in Figure 1.1.

The three tables to be joined are the **Orders** table (S_1), the **Lineitem** table (S_2), and the **Order returns** table (S_3). The **Orders** table contains information about orders made by customers, the schema of this table is $S_1(order\_id, customer\_sk, \ date)$. The **Lineitem** table contains information about the items in each order, its schema is $S_2(order\_id, product\_id, quantity, price)$. The **Order returns** table, having schema $S_3(order\_id, product\_id, return\_quantity)$, contains information about the returns made by customers.

The joined table contains all the columns from the three source tables and is used as input for the K-Means clustering model. It is created by first left joining $S2$ with $S1$ on shared columns *order_id* and *product_id*, then joining the result of this with $S1$ on *order_id*. After this we have $T(order\_id, customer\_sk, date, product\_id, quantity, price, return\_quantity)$.

1.2 Research Questions & Contributions

This thesis focuses on facilitating the adoption of factorized machine learning. A significant part of this work involves developing a cost model designed to accurately decide the optimal approach for training a machine learning model. The cost model, considering the data, model, and hardware dimensions, decides between factorized or materialized computation.

1.2.1 Research Questions

The research questions answered in this thesis are:

RQ.1 How can we optimize and implement factorized Machine Learning for GPUs?

RQ.2 How can we accurately predict the optimal choice between factorized or materialized training of a Machine Learning model, on CPU and GPU, through leveraging knowledge about model, data, and hardware characteristics?

1.2.2 Contributions

The Research Questions are answered by providing the following contributions:

C.1 A GPU optimized implementation of Amalur's factorized Machine Learning framework.

C.2 A cost model that predicts whether factorized or materialized learning is faster, capable of accurate predictions regardless of dataset, model hyper-parameters, or hardware used. This cost model is the result of a detailed study that compares multiple cost calculation strategies.

1.3 Cost Estimation for Factorized Machine Learning

To develop a cost model capable of accurately predicting whether factorized or materialized learning is faster, we capture the runtime of training types for a set of scenarios. We vary the data used, the models trained, and the hardware settings. These collected results are then used as training data to create several types of cost models.

Evaluation on real-world datasets shows that our models outperform the state-of-the-art (SOTA), specifically our **Hybrid** model, which combines a linear regression and an XGBoost model, performs well. It outclasses the SOTA in scenarios with real-world datasets, reaching **80%** of the maximum achievable time saved. This cost model also generalizes well to scenarios with novel hardware settings and datasets, only losing **18%** of its predictive power.

1.4 Outline

This section provides an overview of the structure of the remainder of this thesis. We start with the theoretical concepts and principles that underpin our study in Chapter 2. The literature review (Chapter 3) reviews existing research relevant to our topic and identifies gaps or limitations in the existing literature. In the methodology chapter (Chapter 4), we describe our overall approach to this empirical study, including the breakdown and motivation of the independent variables chosen. In Chapter 5, we detail the statistical and analytical methods used to analyze the data, present the results of each experiment, and include visualizations to illustrate key findings. Next, in Chapter 6, the results are evaluated and discussed. It starts with an explanation of how the empirical results were gathered, followed by a discussion of the results of the experiments in relation to the research questions. Here, we also evaluate the validity and reliability of the results and compare our findings with those of the existing literature. This chapter ends with an in-depth interpretation and discussion of the results and their implications. Finally, in the conclusion chapter (Chapter 7), the main contributions and findings of this thesis are summarized, and we provide an outlook for future research.

Chapter 2

Preliminaries: Factorized Machine Learning

This chapter details the preliminary theoretical concepts that form the foundation for this thesis. First, we explain Data Integration: the process of combining data from different sources, which is a crucial step in any ML workflow, in Section 2.1. With these concepts in mind, we explore factorized Machine Learning in detail (Section 2.2). Finally, in Section 2.3, we explain GPUs and how they are crucial for the ML industry. With these concepts, we provide the theoretical foundation necessary for understanding the content presented in the next chapters of this thesis.

2.1 Data Integration

In order to comprehend the significance and complexities of factorized Machine Learning, it is necessary to have a grasp of the field of Data Integration (DI). In its broadest sense, DI details the relationships between datasets, enabling the merging of data from diverse sources into a unified dataset. This process is crucial for ML applications, as ML frameworks (such as Keras[1] and TensorFlow[2]) typically require a single table as input. An example of such a DI scenario is illustrated in Figure 1.1.

However, when merging data sets into a unified table is essential for machine learning, it may present the following significant challenges [7].

1. **Extra storage**
   The joined dataset will take extra space to store.

2. **Computational redundancy**
   Joining tables can introduce duplication of values in the materialized data (shown in orange in Figure 1.1). These values are included in any computations made during the training of an ML model in this dataset, resulting in duplicate computations.

3. **Join time**
   For complex scenarios, joining datasets can take a significant amount of time.

4. **Maintenance headaches**
   Join query needs updating when changing input table schemas.

[1] https://keras.io/

[2] https://www.tensorflow.org/

Factorized Machine Learning seeks to address issues one through three through the concept of “learning over joins” [1], which involves shifting the computations required for an ML model to the individual tables.

2.1.1 Schema Mapping

Schema mapping is an integral step in the Data Integration process and thus to factorized ML. These mappings specify how the source datasets map to the target tables. Having a formal way to specify how these datasets relate is especially important for factorized ML. It allows us to convert these relationships between the source tables and the target table to a form that can be translated to linear algebra: the normalized matrix (detailed in Subsection 2.2.1).

For the running example the schema mapping is as follows:

Given source datasets, with abbreviated column names:

$$o = order\_id, c = customer\_id, d = date,$$
$$p_{id} = product\_id, q = quantity, p = price, rq = return\_quantity$$
$$S_1(o, c, d)$$
$$S_2(o, p_{id}, q, p)$$
$$S_3(o, p_{id}, rq)$$

The mapping to target table T can be specified as follows. First, we left join S_2 with S_3 on $S_2.o = S_3.o$ and $S_2.p_{id} = S_3.p_{id}$ to get an intermediate table with schema:

$$T(o, p_{id}, q, p, rq)$$

Next, we inner join this with S_1 to get the final table T:

$$T(o, c, d, p_{id}, q, p, rq)$$

Now that we have the schema mapping, we can translate this to the normalized matrix, which we will do in the next section.

2.2 Factorized Machine Learning

As stated previously in this thesis, factorized ML is the process of training Machine Learning models on multiple tables without the need to materialize the join between these tables. This section will go in-depth on how this can be achieved, continuing the running example from Figure 1.1. We start with the definitions (Subsection 2.2.1) followed by an in-depth example of the involved linear algebra (Section 2.2.1).

2.2.1 Normalized Matrix

As Machine Learning algorithms can be expressed in Linear Algebra (LA), we need to express the Data Integration scenario of an ML use case in terms of Linear Algebra, i.e., we need to translate the Schema Mappings of an integration scenario to

Linear Algebra to allow us to achieve the goal of "pushing down" ML to the separate source tables. This is achieved through the **Normalized matrix**: A set of matrices that capture the necessary DI metadata telling us how the source tables map to the materialized Target table [4], [5].

The **Mapping matrix** and **Indicator matrix** respectively represent how the columns and rows from each source table S_k map to the Target table T.

Mapping Matrix

The Mapping Matrix M is a set of matrices M_k for each source table S_k that denotes how the source columns map to the target columns. A value of 1 in this matrix indicates that the corresponding column (via column number) in S_k corresponds to the corresponding column (via column number) in T. The formal definition is as follows.

Definition 2.2.1 (*Mapping matrix* [4])**.** Each source table S_k has a corresponding binary Mapping matrix M of shape $c_T \times c_{S_k}$, where

$$M_k[i,j] = \begin{cases} 1, & \text{if } j\text{-th column of } S_k \text{ is mapped to the } i\text{-th column of } T \\ 0, & \text{otherwise} \end{cases}$$

Note that in the case that there is no column overlap between source tables this Mapping Matrix is redundant. This affects the materialization step, as shown in Definition 2.2.3.

Indicator Matrix

Now that we have defined how to map the columns from the source tables to the target table, we need to do the same for the rows. This is done with the **Indicator** matrix.

Definition 2.2.2 (*Indicator matrix* [5])**.** Each source table S_k has a corresponding binary Indicator matrix I of shape $r_T \times r_{S_k}$, where

$$I_k[i,j] = \begin{cases} 1, & \text{if } i\text{-th row of } S_k \text{ is mapped to the } j\text{-th row of } T \\ 0, & \text{otherwise} \end{cases}$$

Materialization

Using the normalized matrix, we can now **materialize** the join to obtain the target matrix T:

Definition 2.2.3 (*Materializing the Normalized Matrix to obtain Target matrix T*)**.**

Given

k Table id $k \in [1, n]$

S_k Source tables

M_k Mapping matrices

I_k Indicator matrices

$$T = \begin{cases} \sum_{k=1}^{n} I_k S_k M_k^T, & \text{if there is column overlap between source tables} \\ \begin{bmatrix} \vdots & \vdots & \vdots \\ I_1 S_1 & \cdots & I_n S_n \\ \vdots & \vdots & \vdots \end{bmatrix}, & \text{otherwise} \end{cases}$$

The materialization case when there is no column overlap is a horizontal concatenation of each source matrix S_k multiplied by I_k: $I_k S_k, k \in [1, n]$. Intuitively the materialization process can be seen as:

```
for each $k \in [1, n]$ do                          ▷ For each source table
    $rows_k \leftarrow I_k S_k$                      ▷ Map the source table rows to the target table
    $T_k \leftarrow rows_k M_k^T$                    ▷ Map the source table columns to the target table
end for
$T \leftarrow \sum_{k=1}^{n} T_k$                   ▷ Sum the results
```

Running Example: Normalized Matrix

To translate the normalized matrix to how it is used in ML algorithms, we first show the full normalized matrix of the running example, followed by the materialized Target table T. The goal is to show how the matrices interact to allow computation without materializing the join. For completeness, we show the calculations with the Mapping matrices M_k included, but as highlighted before, this is not needed due to this scenario having no column overlap. However, it is insightful to show as it gives an idea of how this process looks when there is column overlap.

These are the corresponding Source matrices $S_{1..3}$ for the source tables shown in Figure 1.1. The orange numbers over the columns denote in which column of T they will end up. The blue numbers at the end of each row illustrate to which target table rows they are mapped.

$$S_1 = \begin{array}{|ccc|l} 0 & 1 & 2 & \\ 1 & 11 & 2024 & 0,1 \\ 2 & 12 & 2024 & 2 \\ 3 & 11 & 2024 & 3 \end{array} \quad S_2 = \begin{array}{|ccc|l} 3 & 4 & 5 & \\ 2 & 20 & 40 & 0 \\ 1 & 25 & 25 & 1 \\ 3 & 13 & 39 & 2 \\ 1 & 10 & 10 & 3 \end{array} \quad S_3 = \begin{array}{|c|l} 6 & \\ 1 & 3 \end{array}$$

The Indicator matrices denote how rows from S map to rows in T. The column number denotes the row in S, the row number denotes the row in T. The blue annotations show more clearly how this works in the form row number in $S_k \rightarrow$ row number in T.

$$I_1 = \begin{array}{|ccc|l} 1 & 0 & 0 & 0 \rightarrow 0 \\ 1 & 0 & 0 & 0 \rightarrow 1 \\ 0 & 1 & 0 & 1 \rightarrow 2 \\ 0 & 0 & 1 & 2 \rightarrow 3 \end{array} \quad I_2 = \begin{array}{|cccc|l} 1 & 0 & 0 & 0 & 0 \rightarrow 0 \\ 0 & 1 & 0 & 0 & 1 \rightarrow 1 \\ 0 & 0 & 1 & 0 & 2 \rightarrow 2 \\ 0 & 0 & 0 & 1 & 3 \rightarrow 3 \end{array} \quad I_3 = \begin{array}{|c|l} 0 & \rightarrow 0 \\ 0 & \rightarrow 1 \\ 0 & \rightarrow 2 \\ 1 & 0 \rightarrow 3 \end{array}$$

The Mapping matrices denote how columns from S map to columns in T. The row number denotes the column in T, the column number denotes the column in S. The orange annotations show this in the form: column number in $S_k \rightarrow$ column number in T.

$$M_1 = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{array}\right]\begin{array}{l} 0 \rightarrow 0 \\ 1 \rightarrow 1 \\ 2 \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 4 \\ \rightarrow 5 \\ \rightarrow 6 \end{array} \qquad M_2 = \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{array}\right]\begin{array}{l} \rightarrow 0 \\ \rightarrow 1 \\ \rightarrow 2 \\ 0 \rightarrow 3 \\ 1 \rightarrow 4 \\ 2 \rightarrow 5 \\ \rightarrow 6 \end{array} \qquad M_3 = \left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{array}\right]\begin{array}{l} \rightarrow 0 \\ \rightarrow 1 \\ \rightarrow 2 \\ \rightarrow 3 \\ \rightarrow 4 \\ \rightarrow 5 \\ 0 \rightarrow 6 \end{array}$$

For conciseness we show the calculation of one of the sub-target tables T_1.

$$T_1 = I_1 \qquad S_1 \qquad M_1^T$$

$$T_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 11 & 2024 \\ 2 & 12 & 2024 \\ 3 & 11 & 2024 \end{bmatrix} M_1^T$$

$$T_1 = \begin{bmatrix} 1 & 11 & 2024 \\ 1 & 11 & 2024 \\ 2 & 12 & 2024 \\ 3 & 11 & 2023 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$T_1 = \begin{bmatrix} 1 & 11 & 2024 & 0 & 0 & 0 & 0 \\ 1 & 11 & 2024 & 0 & 0 & 0 & 0 \\ 2 & 12 & 2024 & 0 & 0 & 0 & 0 \\ 3 & 11 & 2023 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The materialized Target table T is the element wise sum of the dot product of each tuple of Indicator, Source, and Mapping matrices. For each source table S_k the intermittent result is shown as T_k. For clarity the cells from each source table are colored in the same color in the intermittent result and in Target table T.

$$T_1 = I_1 S_1 M_1^T = \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & \cdots & 6 \end{array} \\ \begin{bmatrix} 1 & 11 & 2024 & 0 & \cdots & 0 \\ 1 & 11 & 2024 & 0 & \cdots & 0 \\ 2 & 12 & 2024 & 0 & \cdots & 0 \\ 3 & 11 & 2023 & 0 & \cdots & 0 \end{bmatrix} \end{array} \quad T_2 = I_2 S_2 M_2^T = \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 2 & 20 & 40 & 0 \\ 0 & 0 & 0 & 1 & 25 & 25 & 0 \\ 0 & 0 & 0 & 3 & 13 & 39 & 0 \\ 0 & 0 & 0 & 1 & 10 & 10 & 0 \end{bmatrix} \end{array}$$

$$T_3 = I_3 S_3 M_3^T = \begin{array}{c} \begin{array}{cccc} 0 & \cdots & 5 & 6 \end{array} \\ \begin{bmatrix} 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \end{array} \quad T = \sum_{k=1}^{3} I_k S_k M_k^T = \begin{array}{c} \begin{array}{ccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} 1 & 11 & 2024 & 2 & 20 & 40 & 0 \\ 1 & 11 & 2024 & 1 & 25 & 25 & 0 \\ 2 & 12 & 2024 & 3 & 13 & 39 & 0 \\ 3 & 11 & 2024 & 1 & 10 & 10 & 1 \end{bmatrix} \end{array} \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array}$$

2.2.2 Factorized Linear Algebra

In the previous section, we have shown the properties of the Normalized matrix. This section will show how commonly used Linear Algebra operators are rewritten for the Normalized matrix for the purpose of performing factorized ML [5]. We will show how to perform element-wise operations, reduction operations, dot-product operations, and a running example of right matrix multiplication (RMM) on the Normalized matrix. The goal is to show how (most of) these operations can be performed without materializing the join between the source tables, and how the Normalized matrix allows us to do so.

Element-wise Scalar Operations

This group of operators perform an operation on every element of a matrix independently of each other. The arithmetic operations are: $+$, $-$, $\times$, $\div$ and $^\wedge$ (these operators are denoted by $\oslash$). This can be seen as a scalar function f applied to each element of a matrix T. The rewrite rule therefore is very simple for these arithmetic operators, as well as for any other scalar function (e.g., *log*, *round*) f:

$$x \oslash T \rightarrow [x \oslash S, I, M]$$
$$T \oslash x \rightarrow [S \oslash x, I, M]$$

or more generally:

$$f(T) \rightarrow [f(S), I, M]$$

These operations all return a normalized matrix and can thus be performed without materializing the join between the source tables. In the used implementation [6], when a normalized matrix is transposed, the actual computation is not carried out, but the transpose is simply added as a flag. Then, for any downstream operators, the transpose flag is checked, and the computation is performed accordingly. For these element-wise operations the transposed rewrite is:

$$f(T^T) \rightarrow [f(S), I, M]^T$$

Aggregation

The supported aggregation operators are row-wise and column-wise summation, respectively abbreviated to rowSums and colSums. For the factorized rowSums case we sum each source table separately, then multiply with the indicator matrices and sum the results, the mapping matrix is irrelevant. This operation produces a single (column) vector of size $r_T \times 1$. For the transposed case, it is equal to a column summation. These rewrite rules are:

$$\text{rowSums}(T) \rightarrow \sum_{k=1}^{n} I_k \text{rowSums}(S_k)$$
$$\text{rowSums}(T^T) \rightarrow \text{colSums}(T)$$

Summing column-wise gives a row vector of shape $1 \times c_T$. It is equal to first summing the indicator tables column-wise, then materializing with these aggregated indicator matrices. The rewrite rule for the factorized case is:

$$\begin{aligned}\text{colSums}(T) &\rightarrow \sum_{k=1}^{n} \text{colSums}(I_k) S_k M_k^T \\ \text{colSums}(T^T) &\rightarrow \text{rowSums}(T)\end{aligned}$$

As these operations do not create normalized matrices, and in fact materialize (part of) the join the benefit of factorized computation is smaller.

Multiplication

As matrix multiplication is not commutative, there are different rewrite rules for left- and right-matrix multiplication. The rewrite rule for left matrix multiplication (LMM) with another matrix X is:

$$\begin{aligned}TX &\rightarrow \sum_{k=1}^{n} I_k S_k M_k^T X \\ T^T X &\rightarrow (X^T T)^T\end{aligned}$$

For right matrix multiplication (RMM) the rule is the same, we still essentially materialize the join, but with X on the left-hand side:

$$\begin{aligned}XT &\rightarrow \sum_{k=1}^{n} X I_k S_k M_k^T \\ XT^T &\rightarrow (TX^T)^T\end{aligned}$$

Running Example: Right Matrix Multiplication

We showcase right RMM and its rewrite rule by multiplying with X. First for the materialized Target table T:

$$\begin{aligned}XT = &\begin{bmatrix}1 & 1 & 2 & 3\end{bmatrix}\begin{bmatrix}1 & 11 & 2024 & 2 & 20 & 40 & 0\\ 1 & 11 & 2024 & 1 & 25 & 25 & 0\\ 2 & 12 & 2024 & 3 & 13 & 39 & 0\\ 3 & 11 & 2023 & 1 & 10 & 10 & 1\end{bmatrix}\\ &= \begin{bmatrix}15 & 79 & 14165 & 12 & 101 & 173 & 3\end{bmatrix}\end{aligned}$$

Now for the Normalized matrix, recall the rewrite rule for RMM:

$$XT = \sum_{k=1}^{n} X I_k S_k M_k^T$$

For conciseness we refer back to sub results $T_{0\cdots 2}$ and use them directly here. We also leave out X in the subcalculations for $T_{1,2}$.

$$= XT_0 + XT_1 + XT_2$$

$$
\begin{aligned}
&= \begin{bmatrix} 1 & 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} 1 & 11 & 2024 & 0 & \cdots & 0 \\ 1 & 11 & 2024 & 0 & \cdots & 0 \\ 2 & 12 & 2024 & 0 & \cdots & 0 \\ 3 & 11 & 2023 & 0 & \cdots & 0 \end{bmatrix} + X \begin{bmatrix} 0 & 0 & 0 & 2 & 20 & 40 & 0 \\ 0 & 0 & 0 & 1 & 25 & 25 & 0 \\ 0 & 0 & 0 & 3 & 13 & 39 & 0 \\ 0 & 0 & 0 & 1 & 10 & 10 & 0 \end{bmatrix} + X \begin{bmatrix} 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & 1 \end{bmatrix} \\
&= \begin{bmatrix} 15 & 79 & 14165 & 0 & 0 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 12 & 101 & 173 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix} \\
&= \begin{bmatrix} 15 & 79 & 14165 & 12 & 101 & 173 & 3 \end{bmatrix}
\end{aligned}
$$

2.2.3 Machine Learning Models

We use the same machine learning models as Morpheus [5] and [6]. These models are linear regression, logistic regression, K-Means clustering, and Gaussian NMF. Having demonstrated the transformation rules for the relevant Linear Algebra operations, we are now able to illustrate how these models can be adapted for the Normalized matrix. This adaptation enables us to execute these machine learning models without materializing the join between the source tables. The algorithms are detailed in the next sections, with the factorized operators used highlighted in red. Within the algorithms, we use the following conventions: X represents the matrix of independent variables, which in our scenario corresponds to the normalized matrix. The dependent variable is denoted as y, the weight vector as w, the learning rate as γ, and n represents the number of iterations.

Linear Regression

Algorithm 1 Linear regression using Gradient Descent [5]

```
Require: $X, y, w, \gamma$
  for $i \in 1 : n$ do
    $w = w - \gamma(X^T((Xw) - y))$
  end for
```

Linear regression (Algorithm 1) is an ML technique fit to find linear relationships between independent variables and a dependent variable. The algorithm utilizes gradient descent to iteratively converge towards the best solution. The LA operators performed on the normalized matrix T are Transpose T^T, and Left Matrix Multiplication TX.

Logistic Regression

Logistic regression (Algorithm 2) is very similar to linear regression, but instead of predicting a continuous value, it predicts a binary value. The rewrite rule for logistic regression uses the same operators as linear regression: Transpose and Left Matrix Multiplication.

Algorithm 2 Logistic regression using Gradient Descent [5]

```
Require: $X, y, w, \gamma$
  for $i \in 1 : n$ do
    $w = w - \gamma\left(X^T \frac{y}{1+e^{Xw}}\right)$
  end for
```

K-means Clustering

The regression algorithms discussed above are supervised, that is, they predict a value based on a set of input features. K-Means clustering is an unsupervised algorithm that groups data points into a predefined number of clusters based on their similarity. The operators used to calculate the clusters are $exp\ (X^2)$, Scalar Multiplication, Transposition, Row Summation, and Left Matrix Multiplication. The algorithm is shown in Algorithm 3.

Algorithm 3 K-Means Clustering [5]
$\mathbf{1}_{r \times c}$ denotes a matrix of size $r \times c$ filled with ones, this is used to repeat a vector to a matrix, either row- or column-wise.

```
Require: $X, k$ (number of centroids)
  $C = \text{rand}(r_X \times k)$                               ▷ Randomly initialize centroids matrix $C$
  $D_X = \text{rowSums}(X^2) \times \mathbf{1}_{1 \times k}$        ▷ Compute the $l^2$-norm of points for distances
  $T_2 = 2 \times X$
  for $i \in 1 : n$ do
    $D = D_X - T_2 C + (\mathbf{1}_{r_X \times 1} \times \text{colSums}(C^2))$     ▷ Compute distances
    $A = (D == \text{rowMin}(D) \times \mathbf{1}_{1 \times k})$        ▷ Assign points to the closest centroid
    $C = \frac{X^T A}{\mathbf{1}_{c_X \times 1} \times \text{colSums}(A)}$                 ▷ Update centroids
  end for
```

Gaussian Non-negative Matrix Factorization

Gaussian Non-negative Matrix Factorization (Gaussian NMF) is a technique used to decompose a matrix into two smaller non-negative matrices. It is used for feature extraction from data and is often used in image processing and text mining. The operators used in the algorithm are Transpose and Right Matrix Multiplication. The rank hyperparameter r controls the size of the resulting matrices. The algorithm is shown in Algorithm 4.

Algorithm 4 Gaussian Non-negative Matrix Factorization [5]

```
Require: $X, r$ (rank)
  $W = \text{rand}(r_X \times r)$                               ▷ Randomly initialize $W$
  $H = \text{rand}(r \times c_X)$                               ▷ Randomly initialize $H$
  for $i \in 1 : n$ do
    $H = H \times \left(\frac{W^T X}{W^T W H}\right)$
    $W = W \times \left(\frac{X H^T}{W (H H^T)}\right)$
  end for
```

2.2.4 Overview

In this section we have presented the rewrite rules for the Normalized matrix, for commonly used Linear Algebra operators, and how these are used in the training of Machine Learning models without the need to explicitly compute the join between the source tables. Table 2.1 provides a summary of the operators discussed, with the final column describing the specific operators used for each ML model. This underscores the importance of a robust cost model in factorized ML, since various models leverage a range of operators in distinct ways when benefiting from factorized computation.

Group	Operator	Example	2^{nd} Operand	Output	Used in models
Element-wise	Addition	$T+x$	scalar x	Normalized Matrix	—
	Multiplication	$T \times x$	scalar x		K-Means
	Division	T/x	scalar x		—
	Transposition	T^T	—		LinReg, LogReg, K-Means, G-NMF
	Generic Scalar Function	$f(T)$	f		K-Means (exp)
Aggregation	Row Summation	rowSums(T)	—	Column Vector	K-Means
	Column Summation	colSums(T)	—	Row Vector	—
Multiplication	Left Matrix Multiplication	TY	Matrix Y ($c_T \times r_Y$)	Matrix ($r_T \times c_X$)	LinReg, LogReg, K-Means
	Right Matrix Multiplication	YT	Matrix Y ($c_X \times r_X$)	Matrix ($r_Y \times c_T$)	G-NMF

TABLE 2.1: Overview of factorized ML operators.

2.3 Machine Learning on GPUs

Graphics Processing Units (GPUs) have emerged as the preferred processing units in the Machine Learning domain due to their significant advantages over Central Processing Units (CPUs). As CPUs are designed for diverse general-purpose applications, this is not surprising. Whereas CPUs excel in executing sequential tasks, such as running an operating system, GPUs are optimized for parallel tasks. This specialization enables them to efficiently execute identical operations on multiple pieces of data simultaneously with a considerably higher degree of parallelism than CPUs, exactly what is needed for the prevalent linear algebra operations in ML.

2.3.1 Architecture

This section elaborates on the architecture of a GPU, emphasizing its effectiveness in performing Linear Algebra tasks. Figure 2.1 illustrates the architectural differences between a CPU and a GPU and will serve as a point of reference throughout this section.

The heart of the GPU is the Streaming Multiprocessor (SM), which is responsible for executing thousands of parallel threads. Each SM comprises multiple CUDA cores (highlighted in blue), resembling CPU cores, that carry out arithmetic operations. These cores are optimized to manage multiple operations concurrently, enhancing their effectiveness for the matrix and vector calculations crucial in Machine Learning.

Data and code are transferred to the streaming multiprocessors (SMs) by passing through various memory layers. Initially, data is fetched from the host and stored in the GPU's DRAM, after which it is moved to the SM via the L2 cache. This facilitates quick data exchange and minimizes latency. Threads are grouped into blocks and

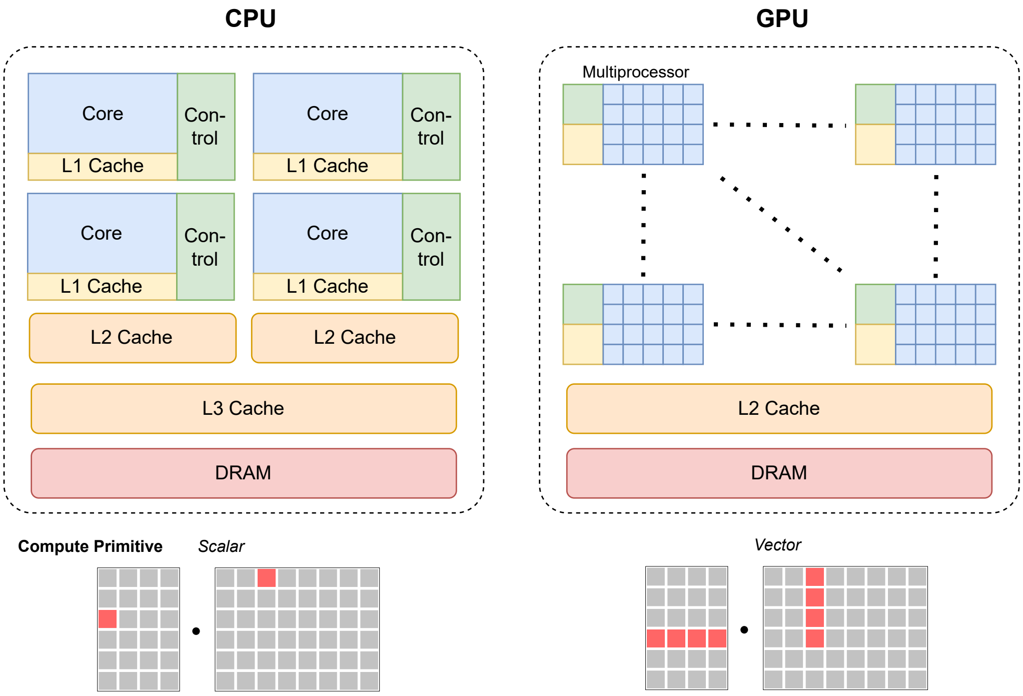


FIGURE 2.1: Simplified view of the difference in architecture between a CPU and a GPU. Compute cores are blue, control units green and L1 Cache is marked in yellow. Figure based on visualizations from [8]–[10]. Compute Primitive shows an algorithm performing sequential matrix multiplication where a scalar operation is processed at a time, and a SIMD algorithm were a vector operation is computed in parallel.

then allocated to SMs for scheduling. This organization enables efficient resource management and parallel thread execution. Each SM is responsible for managing a block of threads that are then scheduled for execution. The cores within the SM execute these threads concurrently, resulting in a high degree of parallelism. This simultaneous processing of multiple data points is known as Single Instruction, Multiple Data (SIMD) architecture. Although CPUs also support SIMD, GPUs offer a much higher degree of parallelism, due to the higher number of cores, which proves to be advantageous in Machine Learning applications, where identical operations are applied across numerous data points.

2.3.2 Estimating Performance on GPU

Following the architecture overview, it is essential to grasp the performance dynamics of a GPU in the context of Machine Learning applications, specifically focusing on Linear Algebra operations. The GPU's ability to execute thousands of threads concurrently is the basis of its computational power, particularly for tasks with high arithmetic intensity. The following passages provide valuable perspectives from [11].

The arithmetic intensity refers to the ratio of mathematical operations performed per memory operation. In the context of GPUs, it is a critical factor in determining performance constraints. The time it takes for a function to execute on a GPU can be constrained by memory bandwidth, mathematical bandwidth, or latency. To illustrate this, imagine a function that reads the input data, performs calculations, and writes the output. In an ideal case, the time spent on memory operations T_{mem} and math operations T_{math} can overlap (because many threads are running at once). In this ideal case, the total execution time will approach $\max(T_{mem}, T_{math})$. As will be shown in Section 5.2 programs on GPUs are often limited by their memory bandwidth, due to their very high computational throughput. On the contrary, for CPU operations, the total time taken is highly correlated with T_{math} (Section 5.1). This discrepancy introduces difficulties in creating a singular cost model that performs well on both the CPU and GPU scenarios.

When the computational time T_{math} exceeds the memory access time T_{mem}, a function is classified as math-bound, indicating that the GPU's computational capabilities are the limiting factor. On the contrary, if T_{mem} is higher, it is memory-bound, which means that the memory bandwidth is the restricting element. This relationship is illustrated by the inequality $\frac{\#ops}{BW_{math}} > \frac{\#bytes}{BW_{mem}}$, which can be rearranged as $\frac{\#ops}{\#bytes} > \frac{BW_{math}}{BW_{mem}}$. Here, the left side denotes a function's arithmetic intensity, while the right side represents the GPU's $ops : byte$ ratio, i.e., the number of Floating Point Operations (FLOPs) per byte retrieved from memory.

In practice, many Machine Learning operations, such as linear layers or activation functions, often have low arithmetic intensities, sometimes executing only one operation for every two-byte element accessed from and stored in memory. This characteristic typically renders them memory-bound on GPUs. However, for operations with high arithmetic intensity, like large matrix multiplications, the GPU's mathematical bandwidth emerges as the constraining factor.

To fully leverage a GPU's capabilities, it is crucial to ensure sufficient parallelism. This is achieved by launching a significant number of thread blocks, ideally several times higher than the number of SMs, to minimize the tail effect, where only a few active thread blocks remain towards the end of a function's execution. By maintaining a high level of parallelism, GPUs can effectively hide instruction latency and maximize throughput, rendering them better suited for the parallel processing demands of Machine Learning compared to CPUs.

How this relates to the performance of factorized ML on GPUs, and the impact it has on the trade-off between materialized and factorized learning, is discussed in Section 5.2.

Chapter 3

Literature Review

Factorized machine learning is a novel technique that allows learning over normalized data without materializing the join of multiple tables. This can potentially reduce redundancy in I/O and computation and speed up the learning process. Several ways have been proposed to achieve and implement this technique. These works are discussed in Section 3.1. However, since Factorization is not always the faster choice [1], [4]–[6]. Consideration must go into choosing the right data representation for ML workflows. Works that present contributions to answering this question are laid out in Section 3.2. Finally, we draw inspiration from the SOTA Machine Learning Optimizers in Section 3.3.

3.1 Factorized Machine Learning

The concept of factorized Learning was proposed in [1]. The paper demonstrates that learning over joins can avoid redundancy in I/O and computation. The authors show that their factorized Learning framework, Orion, is faster in certain tested scenarios where materializing the join introduces significant redundancy. However, its focus on two-table joins limits its applicability to real-world scenarios. The cost model proposed in this paper is based on hardware, data characteristics, and model parameters. Despite its contributions, the scope of the model is limited since it only considers buffer memory as hardware, input table dimensions as data characteristics, and the number of iterations as the only model parameter.

Santoku [12], a toolkit that implements factorized learning in R, extends Orion. The toolkit additionally supports ML models with categorical features, such as Naive Bayes, and extends the factorized approach to ML inference. However, Orion and Santoku have some limitations:

1. Only support PK/FK joins.
2. Requires one-hot encoding of categorical features.
3. Requires manual effort to create a factorized implementation of an ML algorithm.

F [13] addresses this first limitation by extending Factorized Learning to any natural join. However, F only applies to least squares regression models. AC/DC, a system developed by the same authors, generalizes F to non-linear models and eliminates the need for one-hot encoding of categorical features. This is achieved by using sparse data representations for categorical features, which avoid the redundancy of one-hot encoding.

Morpheus [5] proposes a solution to the third problem mentioned earlier. Morpheus uses generic rewrite rules for Linear Algebra (LA) operators to factorize a large ensemble of ML models, without manually rewriting the algorithms. This is achieved by using a specific representation of normalized data called the *normalized matrix*. The rewrite rules apply this normalized matrix to generalize factorized computations. MorpheusFI [14] extends this data abstraction to the *interacted* normalized matrix which can capture non-linear interactions between features, thus extending factorized learning to ML models with quadratic feature spaces. [15] Uses this as a basis to extend MorpheusFI to Gaussian Mixture Models and Neural Networks.

Although the previously mentioned works are mostly specialized pieces of software with limited applicability to real-world ML workflows, Trinity [16] aims to enable writing factorized learning workloads once and deploying them across multiple programming languages and linear algebra tools. This means that DB and ML optimizations can be implemented once and applied to many languages or LA runtimes. However, a significant drawback is that the user must specify whether to materialize the join or perform factorized ML. How other systems alleviate this responsibility from the user is described next.

3.2 Cost Estimation for Factorized Machine Learning

Several works propose frameworks and methods to decide between factorization (F) and materialization (M). However, their cost models have limitations, as they rely on theoretical analysis, simple heuristics, or conservative assumptions. This section reviews these works and highlights their contributions and challenges.

In [1] an analytical model that compares the I/O cost and the CPU cost between F and M is used. The authors analyze the number of operations for each step of Batch Gradient Descent in relation to the input data sizes. This results in a prediction for CPU cost and I/O cost. In their experiments, the model accurately predicts the fastest approach 95% of the time.

Morpheus [5] argues that the use of specific cost models for LA operators is not feasible because it makes the cost model dependent on a single LA back-end. Therefore, they advocate for a "system-agnostic approach that does not need cost models for operators". This approach uses a decision rule based on feature and tuple ratios to determine whether to factorize. The rule is as follows:

Definition 3.2.1 (*Morpheus' Decision Rule*).

τ Tuple ratio

ρ Feature ratio

$$Optimize_{Morpheus}(\tau, \rho) = \begin{cases} Factorize & \tau > 5 \land \rho > 1 \\ Materialize, & \text{otherwise} \end{cases}$$

The conservative choice of thresholds results in Morpheus predicting materialization in cases where it is slower than factorization, but the authors show that these speed-ups are often less than 1.5x.

MorpheusFI [14] analyzes performance trade-offs and crossovers between its factorized interaction framework and materialized execution for LA operations. The authors identify sparsity as another key factor that affects runtime, along with the already known tuple ratio, and feature ratio. They propose a heuristic decision rule based on these factors to help users decide when to use their framework. The decision rule uses the cost ratio of the factorized and materialized approaches for left matrix multiplication. The decision rule considers the number of base tables, the number of sparse dimension tables, and the sparsity of each dimension table, it states that factorization should be undertaken only when a low ratio of the base tables is sparse, or if the ratio is high and the sparsity of each base table times the ratio of the number of samples in S to the number of rows in R is greater than 1. It is important to note that this rule has not been subjected to comprehensive evaluation.

Definition 3.2.2 (*MorpheusFI's Decision Rule*)**.**

q Number of base tables with sparsity $< 5\%$

p Number of base tables

e_k Sparsity of R_k

r_{S_1} Number of samples in S_1

r_k Number of rows in R_k

$$Optimize_{MorpheusFI}(q, p, e, r_{S_1}, r) = \begin{cases} Factorize & q < \lfloor \frac{p}{2} \rfloor \vee (q \geq \lfloor \frac{p}{2} \rfloor \wedge \forall k \in [1, q], e_k \frac{r_{S_1}}{r_k} > 1) \\ Materialize & \text{otherwise} \end{cases}$$

Amalur [6] implements a combination between the two previously mentioned cost estimation approaches: analytical counting of operations and a heuristic decision rule. The decision rule is based on the complexity ratio between factorization and materialization. It computes the number of FLOPs for both approaches. This involves analyzing the training algorithms of various ML models and creating formulas to compute the number of FLOPs needed with regard to the input datasets. Which approach to take is chosen as follows.

Definition 3.2.3 (*Amalur's Decision Rule*)**.**

s Standard complexity

f Factorized complexity

$$Optimize_{Amalur}(s, f) = \begin{cases} Factorize & \frac{s}{f} > 1.5 \\ Materialize & \text{otherwise} \end{cases}$$

The threshold value $t = 1.5$ was chosen as the boundary to cater to preferring false negatives to false positives. This approach shows performance comparable to that of Morpheus.

A comparison of these approaches (see Table 3.1) shows that most cost models are simple heuristic decision rules. Even Orion's analytical cost model is primarily used to count operations. The final decision is also based on a decision rule. These rules are effective at predicting cases where the answer is obvious, such as when there is substantial redundancy. However, a cost model that can accurately predict difficult cases, which are likely to occur more often, is still needed. To achieve this, a decision rule will not suffice. Some explainability may have to be traded for the benefit of creating a more accurate cost model.

System	Model	Relevant features
Orion [1]	Analytical cost model (I/O and CPU cost) → Decision Rule	• Buffer size • Input table dimensions • Model iterations
Morpheus [5]	Heuristic decision rule	• Tuple ratio • Feature ratio
MorpheusFI [14]	Heuristic decision rule	• Sparsity • Input table dimensions
Amalur [6]	Analytical cost model (FLOPs) → Decision rule	• Complexity ratio

TABLE 3.1: Overview of cost models for factorized learning

3.3 Machine Learning Optimizers

Machine learning optimizers are algorithms or techniques that improve the performance of machine learning tasks by finding the optimal configuration or schedule for a given hardware back-end. Optimizers often rely on cost models to estimate the runtime or resource consumption of different options and select the most efficient one. In this section, a selection of existing machine learning optimizers and how they approach the cost estimation problem are reviewed. How their ideas can be applied or adapted to the factorized machine learning setting is also discussed.

[17] Presents a new algorithm for optimizing the schedule of machine learning tasks compiled with Halide [18], a compiler that efficiently expresses and compiles array computations for image processing, computer vision, scientific computation and

machine learning. The algorithm uses a cost model to predict the fastest schedule and reduce runtime. The cost model, a neural network, takes two sets of features as input for each stage of the algorithm: algorithm-specific features and schedule-dependent features. These features are embedded and fed into a fully connected layer that predicts coefficients for hand-crafted terms. These terms are non-linear combinations of input features that the authors expect to be related to runtime. Examples are tasks per core, or the number of times storage is allocated. The computed coefficients are then used to predict the runtime of a given task.

TVM [10] is an end-to-end automated optimization compiler for deep learning that achieves portability of performance through graph and operator level optimizations. It uses a statistical approach to the cost model by using an ML model to predict runtime on a given hardware back-end. The model considers features such as the number of float additions and integer comparisons to make its predictions. This approach enables TVM to generate efficient code for a wide range of hardware back-ends without requiring detailed hardware information or manual tuning.

These optimizers are not directly applicable to the scenario we are creating a cost model for, as the models cannot currently be compiled with TVM or Halide and making them compatible is outside the scope of this thesis. However, insights from these optimizers can inform the cost estimation problem addressed in this research. Table 3.2 presents factors that can help create an accurate model to predict whether materialization or factorization is faster.

System	Reference	Model	Relevant features
TVM	[10]	XGBoost	• Memory access count • Memory buffer reuse ratio • Number of time kernel is called • Touched memory size
Halide	[17]	Vector of hand-crafted features multiplied by coefficients computed by Neural Network	• Total number of allocations made • Total number of bytes read • Total number of scalar instructions

TABLE 3.2: Overview of discussed ML optimizers

3.4 Research Gap

A comprehensive performance analysis of factorized machine learning, conducted through profiling and experimentation, will provide valuable information for developing an accurate cost model. This cost model aims to determine the optimal approach, factorization or materialization, in terms of training time. By comparing this analysis with an analysis of materialized machine learning, we can gain a deeper understanding of the computational differences and the factors that influence them.

Previous research has identified data, hardware, and model characteristics as factors that impact the decision to factorize or materialize. However, these works have not been able to accurately predict the optimal approach due to their limited optimization space and narrow ranges for cost model parameters. Additionally, the influence of hardware on runtime and its relationship with the trade-off between factorization and materialization have not been fully considered by the authors. Despite the widespread adoption of GPUs for ML model training, the impact of factorized training on the trade-off has not been explored in any prior work. Therefore, the main research gaps in this area can be summarized as follows.

RG.1 The trade-off between factorization and materialization in relation to hardware characteristics, particularly for training on GPUs, remains an underexplored area of study.

RG.2 Limited optimization space and narrow ranges for cost model parameters.

RG.1 is addressed by *RQ.1* on a GPU optimized factorized ML implementation, in Section 4.1.1. To answer *RQ.2*, on how to create an accurate and robust cost model, we perform experiments with a large range of independent variables increasing the optimization space when compared to related research. The details on how this fills *RG.2* are in Section 6.1.

Chapter 4

Methodology

This chapter outlines the methodology used to get an accurate cost prediction for factorized Machine Learning. We start by introducing the problem setting in Section 4.1, where we explain the choices for the independent variables. Next, we present the proposed cost estimation models in Section 4.2.

4.1 Problem Setting

As this is an empirical study, the focus is on carried out experiments and their results. Therefore, it is extremely important to design these experiments well. This starts with a look back at the problem we are trying to solve, after which we can say precisely what is needed to solve this problem. The experiments are then designed to gather the necessary results to arrive at a fitting solution.

4.1.1 Independent Variables

To reiterate, the objective of this thesis is to develop an accurate and generalizable cost model to determine whether factorized or materialized Machine Learning is the optimal choice for a given Machine Learning scenario. This requires understanding the factors that influence this decision. Prior research has already identified the three dimensions that impact cost: data characteristics [4]–[6], hardware characteristics [1], and model type & hyperparameters [4], [6]. In this section, we elaborate on the independent variables that are taken into account in this study.

Data

Existing literature has recognized the impact of certain data characteristics on factorized learning. Morpheus [5] argues that the most significant of these factors is the relationship between the number of columns/rows in the Source tables and the Target table. The authors expand on this notion in [14], demonstrating that sparsity has serious implications for the factorization vs materialization (F/M) trade-off. In this study, we incorporate the characteristics mentioned in previous research, supplemented by a new set of features. We also include a wider range of variation for each data characteristic, which allows for more insight into the relationship between these data characteristics and the training cost. The data considered characteristics are detailed in Table 4.1.

Independent Variable	Symbol	Explanation	Reason for choice
Sparsity	e	Fraction of zero-valued elements	Impacts the number of computation needed for sparse implementations. [5], [6], [14]
Table Size (rows/ columns)	c/r	Dimensions of tables. Both Target and Source.	[5], [6]
Tuple ratio	ρ	Ratio of rows from $S_{2 \cdots k}$ in S_1	Influences the number of redundant operations when computing a model [5], [6]
Feature ratio	τ	Ratio of columns from $S_{2 \cdots k}$ in S_1	Influences the number of redundant operations when computing a model [5], [6]
Join type	j_t	The join type used to join the source tables to the target table	[6]
Selectivity	σ	The fraction of rows from $S_{1 \cdots k}$ that are included in T	Can be used to estimate the computational redundancy between F/M [6], [14]

TABLE 4.1: Overview of data related features varied in this study. A reference in the column 'Reason for choice' denotes this feature is either used in the cost estimation rule in that publication, or the publication has a thorough analysis showing the impact of this feature on runtime.

Hardware

This section answers how this thesis addresses *RG.2* by addressing the hardware characteristics that represent the second dimension that influences the cost of model training. In this study, we vary these characteristics to understand their impact on training cost. The primary distinction in hardware is between CPU and GPU. This thesis places a greater focus on GPUs, given their prevalent use in the training of ML models. However, to facilitate a comparison, we also include CPUs, albeit with a lesser degree of variation. We experiment with different degrees of parallelism by altering the number of cores. As for the characteristics associated with GPUs, we vary them through experiments on different GPU types and architectures. By changing the types of GPU used, we aim to understand the effect of the following variables.

- Number of Streaming Processors
- Number of compute cores, clock speeds and floating point processing power
- Cache characteristics (L1, L2 size & bandwidth)

- Memory characteristics (bandwidth, frequency)
- GPU architecture

The specific values for these variables, along with the exact types of GPU used, are provided in Appendix B. A more comprehensive discussion of *GPU Architectures* is presented in the next paragraph.

GPU Architectures We intentionally chose a variety of GPU architectures to capture metrics of GPUs with different characteristics. Both older (Pascal, 2016) and newer (Ampere, 2020) architectures are incorporated with the aim of developing a cost model that is not confined to a single generation of hardware. Restricting the study to GPUs from a single generation would constrain the generalizability of the cost model, as they employ the same architecture, i.e., they utilize similar Streaming Multiprocessors and Cache layouts.

Model

The characteristics of the model are varied by selecting four distinct models: linear regression, logistic regression, Gaussian Non-negative Matrix Factorization, and K-Means Clustering. To avoid an exponential increase in the number of combinations of independent variables, we opt not to vary certain hyperparameters, such as k in K-Means or r in G-NMF. Nevertheless, we incorporate a multitude of features that encapsulate the variations that would otherwise be captured by altering these hyperparameters.

One of those features is the complexity of the model, that is, the number of operations needed to train a model. The hyperparameters mentioned above are parameters of the function to compute this feature; thus, we assume that our cost models will still be able to accurately predict runtime for different hyperparameter settings, as the complexity (ratio) has already been shown to be a capable predictor for the F/M trade-off.

4.1.2 Dependent Variables

In this study, the dependent variable is the cost of training a model, which is represented as the training time. The objective of the cost models is to identify the most efficient method for training a model, which is why we utilize training time as the dependent variable.

A variety of **profiling metrics** is also gathered to quantify the cost of training a model. These metrics are used to calculate the cost associated with each operation in the training process. By conducting micro-benchmarks within a representative sub-range of our dependent variables, we discern how these variables influence the execution of computations on the GPU. The collected metrics are shown in Table 4.2. They allow us to calculate the total time taken for computation and memory ($ops : byte$), and allow us to infer how changes in the independent variables affect the utilization of GPUs and the F/M trade-off.

Section name	Metric name	Metric unit
Command line profiler metrics	*`dram__bytes_read.sum`*	byte
	`dram__bytes_write.sum`	byte
GPU speed of light throughput	Duration	nsecond
	SM frequency	cycle/second
	Elapsed cycles	cycle
	SM active cycles	cycle
	Compute (SM) throughput	%
	DRAM frequency	cycle/second
	Memory throughput	%
	DRAM throughput	%
	L1 cache throughput	%
	L2 cache throughput	%
Memory workload analysis	*Memory throughput*	byte/second
	Mem busy	%
	Max bandwidth	%
	L1 hit rate	%
	L2 hit rate	%
	Mem pipes busy	%

TABLE 4.2: Collected profiling metrics and their explanation. The metrics related to the compute cost are **bold**, those related to the memory cost are *italicized*.

4.2 Proposed Cost Models

This section provides an introduction to the concepts underlying our proposed cost models, which are elaborated in Chapter 5. We first give a short introduction to the models and the reasoning behind choosing these models after which we go slightly more in-depth into each separate cost model.

4.2.1 Overview

The first model, termed the analytical model, is a formula derived from the actual cost of the operations. By using this formula to deterministically calculate the cost of training a model, for both the factorized and materialized case, we infer which is optimal. Its simplicity lends itself to high explainability, but it may not perform as well as more complex methods due to the impracticality of incorporating the effects of all independent variables.

Including more independent variables as features is straight-forward for ML-based cost models, which is why the next types are ML-based. With our second model, a linear regression model, we can incorporate more features, broadening the decision space. The third model, a tree-boosting model, uses a set of regression trees for its predictions. It is chosen as it is still explainable, but can capture the more intricate interactions between features than the linear models. It is interesting to compare these cost-based models to reason about the complexity of the problem and the generalizability of the models. If the tree-boosting model outperforms the other models, it suggests that the problem is more complex as there are non-linear feature interactions that cannot be captured in a linear model. However, if the linear models perform sufficiently well, they are likely a better choice as there is less risk of overfitting to training data. Finally, we propose a hybrid model, which integrates the insights derived from the preceding cost models and leverages the strengths of the most effective models to build a superior model.

Our cost modelling approach differs from previous research in two key areas. Firstly, we hypothesize that hardware choice affects the F/M trade-off, and thus we include hardware characteristics in our cost models. Secondly, we investigate the use of ML-based cost models as they allow for more complex feature interactions to be captured. This is in contrast to previous research, which has primarily used analytical decision rules decide whether to factorize or materialize.

4.2.2 Analytical

The analytical model is a deterministic model, constructed by examining the operations executed by the learning algorithm. This model is based on a formula derived from the actual cost of the operations. It encapsulates the critical factors of the algorithm, such as the number of matrix multiplications. For example, if an algorithm performs one addition and two multiplications, the formula for this would be $ADD + 2MULT$. The actual cost values for ADD and $MULT$ are determined through micro benchmarks, and these values are then used to complete the formula and obtain the final model. In our context, these operations are the linear algebra operations performed as part of the machine learning model training. Therefore, by capturing the profiling metrics mentioned in Table 4.2, we can compute the cost of each operation in the training process. As will be demonstrated in Chapter 5, the memory cost of an operation is a reliable predictor for its total runtime. This is the reason we concentrate on the memory cost of an operation in the analytical model. When compared to related research, this cost model is the most like Orion's cost model [1], which also explicitly incorporates multiple cost factors like I/O and Cache costs. However, Orion is highly specific to CPU training, whereas the analytical model in this study is GPU-focused.

By profiling across a broad spectrum of the selected independent variables, we can estimate their impact on memory cost. Integrating this information into the analytical model, allows us to construct a highly interpretable model that can be utilized to estimate the cost of various approaches.

4.2.3 Linear Regression

This model is grounded in empirical data and utilizes linear regression to predict runtime and make a decision between factorization and materialization. It takes into account various features known to influence performance, including the size of the input data, the complexity of the algorithm, and the hardware configuration. By examining the relationships between these features and the actual runtime of the algorithm, the linear regression model can make accurate and interpretable predictions about the cost of different approaches.

Previous research has shown that modelling linear relations between features is sufficient for accurate predictions, e.g., Morpheus' decision rule that only takes into account tuple- and feature-ratios [5], and Amalur's complexity ratio [4]. If the cost characteristics of factorized training on GPU do not differ significantly from those of training on CPU, we expect this linear cost model to perform well. However, if the cost characteristics are more complex, a more intricate model will result in more accurate predictions.

4.2.4 XGBoost

To evaluate whether the preceding models, such as the analytical and linear regression models, are overly simplistic, we incorporate a more intricate XGBoost regressor model. If this model surpasses the performance of the other models, it suggests that more complex feature interactions occur that the other models have not been able to capture. This model is capable of modeling these complex interactions from the data and making more precise predictions about the cost of different approaches. The primary disadvantage is that this model is more expensive to train and is prone to overfitting on small training sets [19].

4.2.5 Hybrid

Finally, the insights obtained from the preceding cost models are combined and used in a hybrid model. By leveraging the strengths of multiple models, the hybrid model is able to make more accurate predictions about the cost of different approaches. Which models are chosen, and how they are combined is determined by the performance of the individual models. Through an analysis of the performance on specific scenarios, i.e., CPU and GPU scenarios, we pick the best models for the hybrid model. This model is expected to outperform the individual models, as it can leverage the strengths of each model to make more accurate predictions.

Chapter 5

Cost Estimation

In this chapter, we present the results of our experimental work and explain their application in the formulation of four different cost models. The first section (Section 5.1) provides an overview of the results of the runtime experiment, motivating why a cost model is necessary. In Section 5.2 the collected profiling metrics are aggregated and analyzed, providing information on how the fundamentals of the GPU can affect the F/M trade-off. Section 5.3 outlines how we compiled and manipulated our independent variables to produce a suitable dataset for training models. Finally, in Section 5.4, we explore various cost models created using the detailed results of the runtime and profiling experiments, and we assess the merits and drawbacks of our analytical, linear regression, XGBoost, and hybrid models.

5.1 Motivation

This section illustrates the need for precise cost estimation when deciding between factorization and materialization strategies. We structure this motivation in three stages. First, we demonstrate the advantages of factorization. Next, we examine how the characteristics of the data and models affect the outcome. By visualizing the performance ratio ($\frac{\text{Time}_M}{\text{Time}_F}$) against various independent variables, we identify initial trends that affect the F/M trade-off. Finally, we highlight the significance of GPUs in this context.

5.1.1 Benefit of Factorization

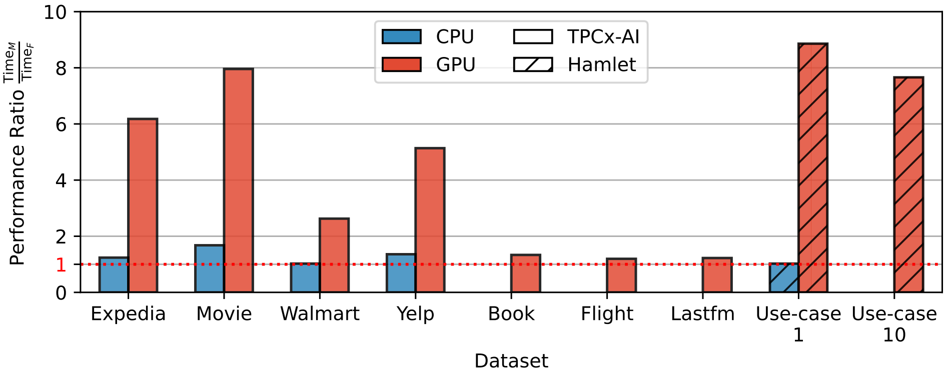


FIGURE 5.1: Average Performance ratio of ML models for positive cases ($\text{Time}_M > \text{Time}_F$), split per tested real dataset and compute type.

The aim of factorized machine learning is to minimize unnecessary operations during model training, thus enhancing efficiency and speed. The comparative performance of factorization versus materialization on actual datasets is depicted in Figure 5.1. The exploration of factorization proves advantageous; it is observed that in 18% of the instances tested on real datasets, factorization is faster, yielding an average acceleration 5.1×. In the most extreme cases, the training time is reduced by more than 20 seconds, a 27-fold reduction. Particularly in situations where training is recurrent, such as during hyperparameter tuning or online learning, this efficiency can translate into substantial time savings.

5.1.2 Data & Model Characteristics

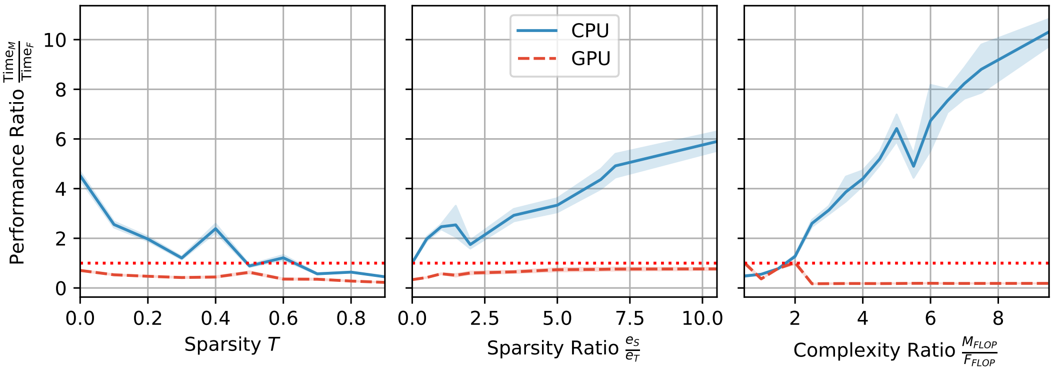


FIGURE 5.2: Performance ratio ($\frac{\text{Time}_M}{\text{Time}_F}$) against independent variables. Broken down by compute type (CPU/GPU). 99% confidence interval shown as shaded area. The sparsity ratio is defined as the sparsity of the source tables $S_k, k \in [1, n]$ divided by the sparsity of target table T. The sparsity of S is defined as the total nonzero values in the base tables divided by the total number of cells in the base tables, $\frac{\sum_{k=1}^{n} nnz(S_k)}{\sum_{k=1}^{n} r_{S_k} \times c_{S_k}}$. High sparsity ratio means the target table is relatively denser than the source tables.

The influence of different data related independent variables on the performance ratio is discussed here, and shown in Figure 5.2. The data reveals a modest inverse relationship between the performance ratio and the sparsity of the target table (T). Further analysis, shown in the second column, delves deeper into the connection between performance and sparsity. A higher sparsity ratio —meaning the base tables (S_k) have a higher count of zero-valued items relative to the target table (T)— generally results in factorization being more efficient than materialization. The plots on the far right suggest that a higher complexity ratio ($\frac{FLOP_M}{FLOP_F}$) tends to favor factorization as the optimal training approach. This aligns with the logical premise that factorization is advantageous when it eliminates redundant operations.

It is crucial to note that the relationship between these data characteristics and the acceleration provided by factorization becomes less distinct with GPU computations. The reason is that these computations are typically limited by memory capacity rather than processing power. This perspective is examined in detail in Section 5.2, which delves into the metrics gathered during profiling experiments.

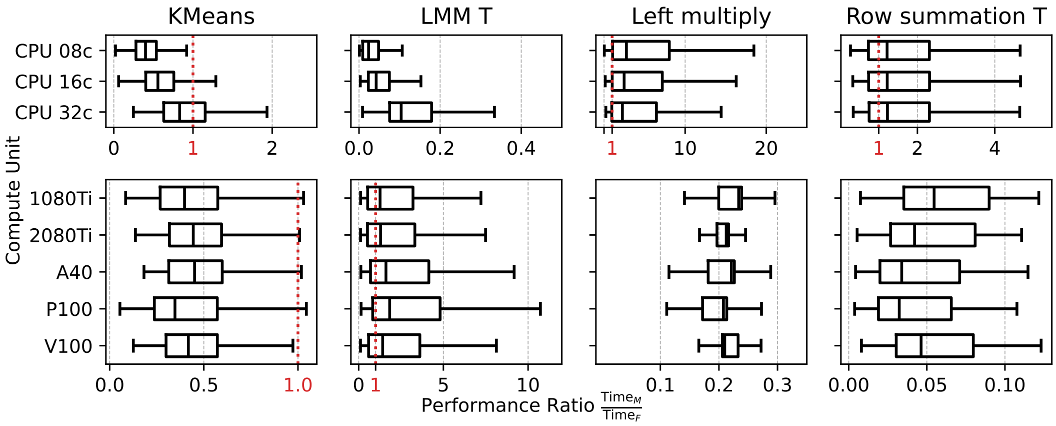


FIGURE 5.3: Box plots showing performance ratio, of various operators on synthetic data, against independent hardware variables. The performance ratio is shown to be affected by hardware choice.

5.1.3 Hardware Characteristics

The hardware used for computation not only affects a program's runtime, but also influences the F/M trade-off. Different processing units (i.e., CPU or GPU type) have unique thresholds at which factorization becomes more advantageous than materialization. This phenomenon is illustrated in Figure 5.3. The impact on the performance ratio varies depending on the hardware and the specific operation performed. For example, the average performance ratio for transposed Left Matrix Multiplication on the P100 GPU is 3.03 ± 2.70, in contrast to a marginally lower 2.32 ± 2.21. On the contrary, for left (scalar) multiplication, the V100 GPU exhibits a higher performance ratio of 0.21 ± 0.04, compared to the P100 0.19 ± 0.05.

Compute Unit	Mean	Std. Dev.	Count	% with Speedup
CPU 08c	1.27	0.25	172	1.78%
CPU 16c	1.32	0.34	579	5.99%
CPU 32c	1.48	0.46	2873	29.74%
1080Ti	2.27	1.60	432	4.47%
2080Ti	1.87	1.09	425	4.40%
A40	2.00	1.20	392	4.06%
P100	2.52	1.84	461	4.77%
V100	1.95	1.13	404	4.18%

TABLE 5.1: Mean performance ratio of ML models for cases where factorization is preferred over materialization (speedup > 1). This shows hardware choice is a large factor in when to choose factorization over materialization.

In instances where factorization is favored over materialization (when $\text{Time}_M > \text{Time}_F$), the performance varies between GPU types. Both the average performance ratio and the number of instances where factorization outperforms materialization differ, as indicated in Table 5.1. This variation underscores the importance of hardware selection in the factorization versus materialization decision. However, the variation among GPU models is less pronounced than the disparity between CPUs and GPUs. Consequently, it may be more beneficial to consider the type of computation —CPU or GPU— as an independent variable in the cost models rather than focusing on specific GPU models.

When comparing the performance ratio with the complexity ratio for different hardware settings, an interesting pattern emerges. As described in Section 3.2, the complexity ratio is the ratio of the number of floating point operations (FLOPs) needed for the factorized case divided by the FLOPs of the materialized case. According to previous research, a higher complexity ratio typically indicates that factorization is more advantageous. However, our experiments reveal that while this is generally true for CPU operations, it does not always apply to GPU computations. This pattern is illustrated in Figure 5.4, and the reasons behind it are explored further in the following section.

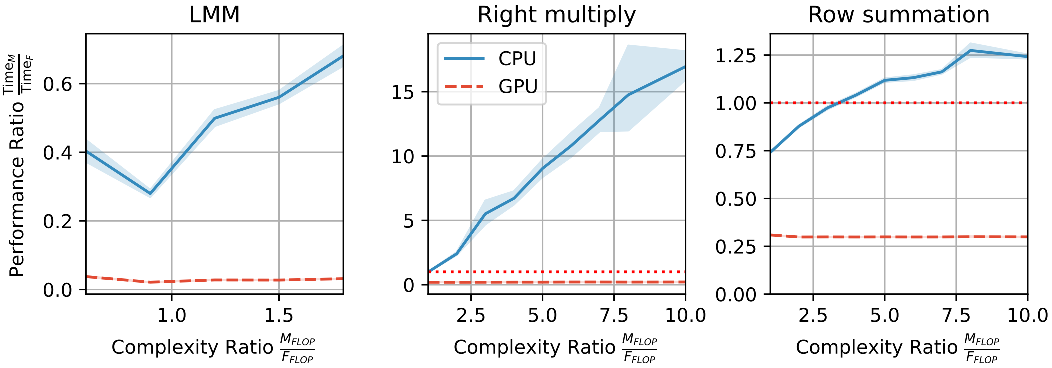


FIGURE 5.4: Performance ratio, of various operators on synthetic data, against complexity ratio, broken down by CPU and GPU. 95% Confidence interval shown as shaded area. Where a lot of operators show clear correlation between the complexity ratio and the performance ratio on CPU, this is not the case for GPU.

5.2 GPU Performance Analysis

An essential preliminary step in creating a precise cost model is understanding the performance attributes of the scenarios under examination. This section analyzes the profiling metrics collected during the experiments to understand the influence of hardware selection on the factorization versus materialization trade-off. The first analysis involves comparing the memory cost and the math cost in the profiled scenarios. According to NVIDIA, an effective method to predict the execution time of a GPU program is to calculate $max(T_{mem}, T_{math})$ [11]. In this formula T_{mem} is the time required to transfer data to and from the GPU memory, while T_{math} denotes the time needed for actual computations. This approach is in line with the inherently parallel architecture of GPUs. Should the data transfer to the GPU prove insufficiently fast, the GPU's Streaming Multiprocessors will remain idle, awaiting data. This indicates a memory-bound program. Conversely, if $T_{math} > T_{mem}$, the program is considered compute bound. This section elaborates which scenario applies to our experiments and details how this knowledge can be harnessed to estimate the runtime for machine learning training scenarios.

Figure 5.5 shows the relationship between memory time T_{mem} and computation time T_{math}, revealing a strong correlation ($\rho = 0.99$). The data predominantly shows that

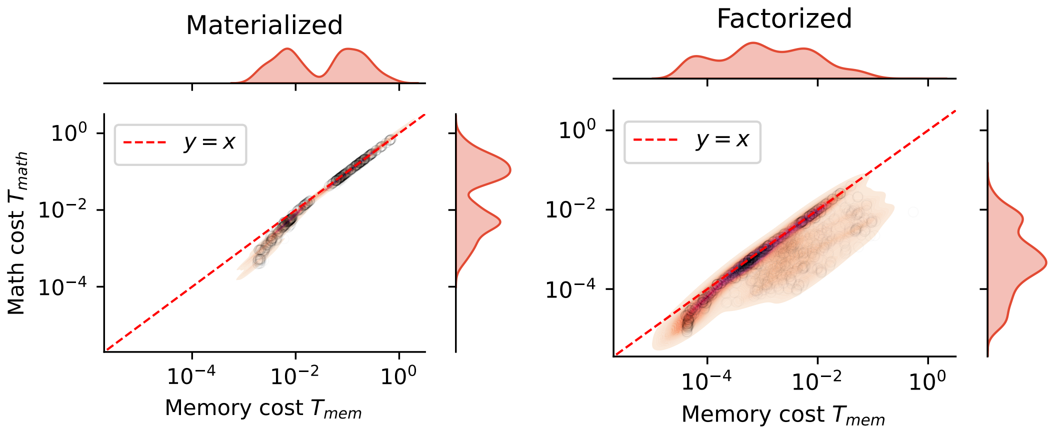


FIGURE 5.5: Memory cost (T_{mem}) vs compute cost (T_{math}) of profiled scenarios. The memory cost is computed as the total number of bytes read and written to memory divided by the measured average memory bandwidth. The math cost is the number of cycles the Streaming Multiprocessors were active divided by the measured average SM frequency.

the memory cost exceeds the computational cost, as most points lie below the $y = x$ line, indicating that the operations are memory-bound. This observation suggests that memory cost prediction should be prioritized in our cost estimation efforts. A notable distinction between factorization and materialization emerges when examining their correlation values; materialized cases exhibit a correlation of $\rho = 0.99$, while factorized cases show a significantly lower correlation of $\rho = 0.40$. This discrepancy arises because materialization typically involves handling a single matrix, or two in the case of matrix multiplication. In contrast, the factorized case on the normalized matrix involves multiple matrices ($S_k, I_k, M_k, k \in \{1 \dots n\}$), each contributing to different computations. Although this diversity reduces both the memory and the computation costs, on average, it also results in a deviation from the $T_{mem} = T_{math}$ line due to the sequential execution of computations on these matrices within the GPU.

Roofline Model

More insights into the efficiency of the tested scenarios, and the differences between factorization and materialization, and GPU types, can be gained by using a roofline model. It is a “model that offers insight ... on improving parallel software and hardware for floating point computations” [20]. The model delineates whether an operation is constrained by memory or computation. On the x-axis, it plots the arithmetic intensity of a program (in our context, this refers to an operator applied to a dataset), measured in FLOPs per byte. The y-axis represents the achievable performance in GFLOPS. The roofline itself (illustrated in gray) signifies the performance limit for a particular GPU, derived from its maximum memory bandwidth and computational capacity in FLOPs per second. The intersection point, known as the “ridge point,” indicates the minimum arithmetic intensity required to fully leverage the computational capabilities of the GPU. By plotting programs on this chart, one can infer whether a program is memory-bound (to the left of the ridge point) or compute-bound (to the right of the ridge point). This analysis is crucial because it highlights potential optimization possibilities by pinpointing performance bottlenecks.

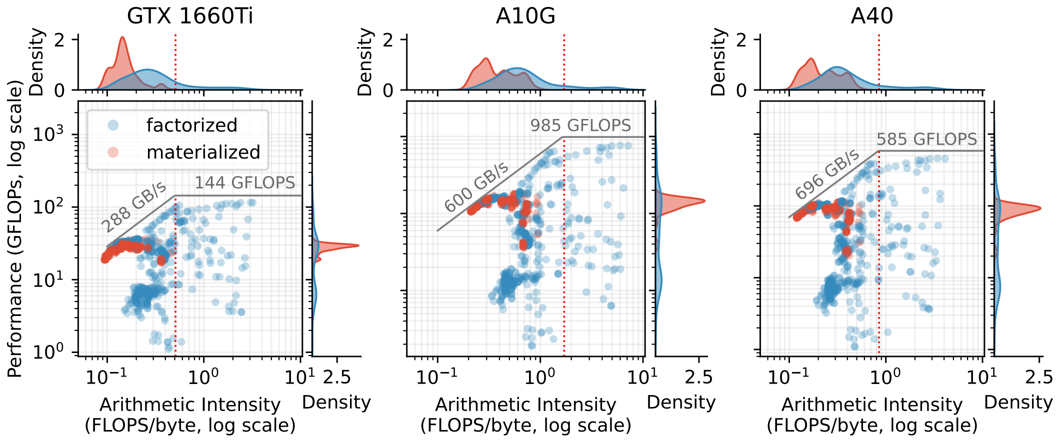


FIGURE 5.6: Roofline chart showing where the performance of the GPUs lies in the memory-bound vs compute-bound spectrum. The subplots on top and right side of each figure show the distribution along the performance (GFLOPs) and operational intensity (FLOPS/byte) axes. Similar GPU types have similar distributions.

The roofline charts depicted in Figure 5.6 validate that most scenarios are constrained by memory. However, what stands out is the impact of the GPU type on performance, as well as the contrast between factorization and materialization. The difference among GPU models is particularly evident in the distribution plots to the right of each subplot. High-performance GPUs such as the A10G and A40 demonstrate a superior memory bandwidth compared to the GTX1660Ti, which often reaches a plateau in performance due to memory bandwidth limitations. Consequently, scenarios involving the A10G and A40 achieve a higher average performance.

The divergence between factorization and materialization highlighted in these graphs is informative. Materialized operators exhibit lower arithmetic complexity than their factorized counterparts, as indicated in the top density plots. This suggests that factorized operators, on average, are less constrained by memory, allowing them to better leverage the computational resources of the GPU. However, the significant variance in performance achieved by factorized operators, as shown in the right density plots, is due to the sequential execution of operations on different matrix segments. There is potential for optimization here, which could allow these operators to more effectively utilize the GPU's computational power.

The roofline chart in Figure 5.7 provides a detailed comparison of different operators, highlighting the distinctions between factorization and materialization. For Left Matrix Multiplication (LMM), the factorized approach exhibits a large variance in performance, aligning with the previously mentioned use of a normalized matrix. Despite this variance, factorization generally appears to be advantageous for LMM, as indicated by the profiling metrics that suggest a more complete utilization of GPU resources. In contrast, the difference in performance between factorized and materialized cases for Left Scalar Multiplication is less pronounced. This is expected since the factorized operation is simpler and involves only the multiplication of source matrices by a scalar. The rightmost plot reveals a clear underutilization of GPU capabilities for Transpose Row Summation in the factorized form, compared to the materialized form. This suggests that while factorization can be beneficial for certain operations, it may not always be the most efficient use of GPU resources.

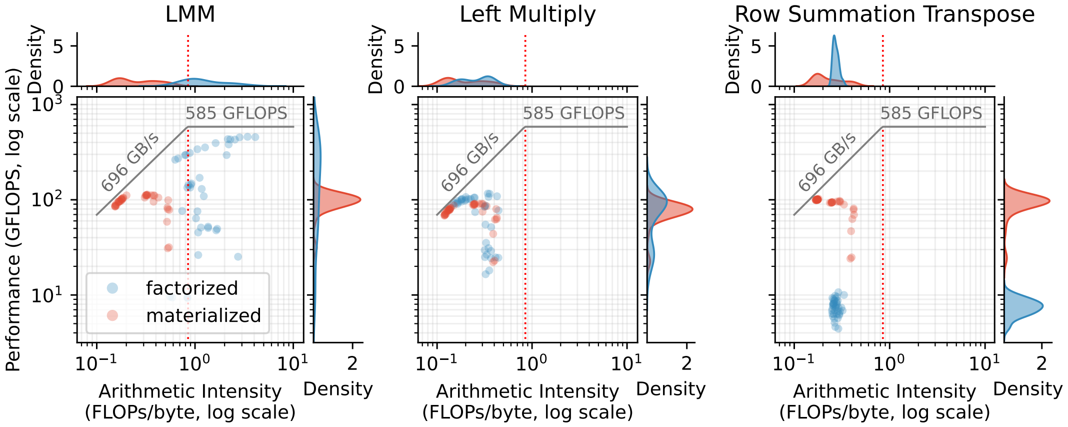


FIGURE 5.7: Roofline chart comparing factorization and materialization for Left Matrix Multiplication, Left Scalar Multiplication and Row Summation Transpose. Metrics from NVIDIA A40.

For a complete view of all operators, the full set of roofline charts is available in Appendix C, providing insight into the performance dynamics of each operator.

5.3 Feature Engineering

We proceed to construct a well-suited dataset for training our cost models. Building on the foundational work that has revealed the relations between speedup and various data and hardware characteristics in Section 5.1, and incorporating the insights from our profiling analysis we enrich and process the dataset.

This dataset, gathered through our comprehensive experiments, has a sample for each unique scenario, with the runtime and attributes of the dataset, the operator type, the model form (F/M), and the hardware configuration. The following section outlines the preprocessing and augmentation of this dataset with additional features. The enrichment process is designed to capture the complex interactions and patterns we have observed. The features are carefully crafted to enhance the models' ability to make predictions about when to use factorization or materialization. The complete set of features is detailed in Appendix D, with a focused discussion on a select subset presented in Table 5.3.

5.3.1 Preprocessing Profiling Metrics

The feature engineering process begins with the integration of the profiling metrics into the dataset. This process presents two significant challenges. The first challenge arises from the fact that these metrics are collected at the kernel level, while our focus is on operator performance. To address this, we aggregate kernel-level metrics to reflect the performance of the entire scenario. The second challenge is the incomplete coverage of the metrics, as they were collected for a subset of scenarios. The strategy to overcome these obstacles is depicted in Figure 5.8 and is further detailed below.

The procedure to aggregate kernel-level metrics to operator-level metrics is as follows, and is also depicted in Figure 5.8 (marked by the "1"). For each scenario, we compute the sum of the metrics that are totals, i.e., the metrics with units of *nanoseconds*, *bytes*, or *cycles* (refer to Table 4.2 for applicable metrics). The remaining metrics

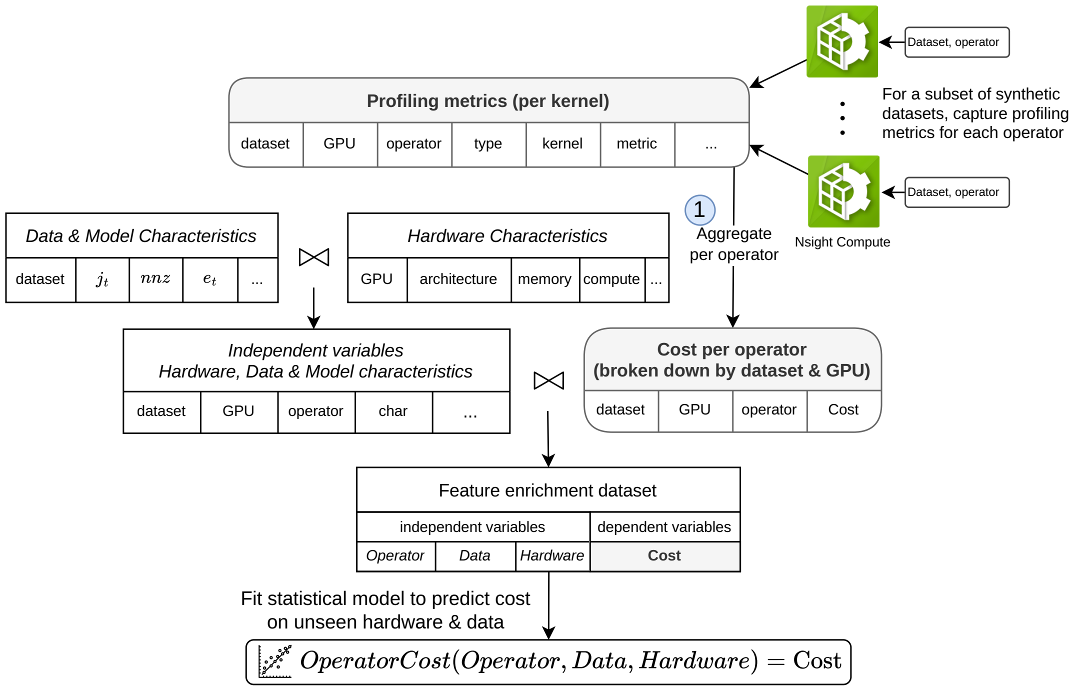

Dataset, operator
For a subset of synthetic datasets, capture profiling metrics for each operator
Dataset, operator
Nsight Compute
Profiling metrics (per kernel)
dataset
GPU
operator
type
kernel
metric
...
1
Aggregate per operator
Data & Model Characteristics
dataset
j_t
nnz
e_t
...
⋈
Hardware Characteristics
GPU
architecture
memory
compute
...
Independent variables
Hardware, Data & Model characteristics
dataset
GPU
operator
char
...
⋈
Cost per operator
(broken down by dataset & GPU)
dataset
GPU
operator
Cost
Feature enrichment dataset
independent variables
dependent variables
Operator
Data
Hardware
Cost
Fit statistical model to predict cost on unseen hardware & data
$OperatorCost(Operator, Data, Hardware) = \text{Cost}$

FIGURE 5.8: Workflow of enriching the collected data with additional features from the profiling experiments. Items related to these profiling experiments are **bolded**, while the features from the data, model & hardware characteristics are *italicized*. Description for the aggregation process ("1") is in Section 5.3.

are either *percentages* or measures *per second*. For these metrics, we calculate the weighted average, where the weight is the runtime of the kernel. This ensures that the metrics accurately represent the total runtime of the scenario. This procedure yields precise operator-level metrics for each scenario.

The second issue relates to the fact that metrics are not collected for all scenarios. This is addressed by fitting a statistical model to the collected metrics and employing this model to predict the missing metrics. The model, denoted as $OperatorCost$ in Figure 5.8, is an ensemble of linear regressors. It uses aggregated metrics, hardware characteristics, and data characteristics as features, with memory cost T_{mem} as the target variable. Each combination of operator and training type (F/M) has its own linear regression model. To enhance the model, we engineer an additional set of features derived from the metrics collected, as shown in Table 5.2. The model is trained on a subset of the metrics collected and tested on the remaining metrics. Subsequently, the model is used to predict the missing metrics. This process is crucial to ensure that we have metrics for all scenarios, as the cost models require a complete dataset for training. The true versus predicted values of this model are depicted in Figure 5.9.

5.3.2 Model-level Math- and Memory-cost

In this section, we describe the process of calculating the memory and mathematical costs for each scenario. Contrary to the previous section, which estimated the memory cost as perceived by the GPU, we now compute the theoretical costs. Memory cost is determined by dividing the total number of bytes read and written to memory by memory throughput. The mathematical cost is computed as the total number

Feature	Formula	Description
`dram_bytes_sum`	`dram_bytes_read_sum` + `dram_bytes_write_sum`	Total number of bytes read and written to DRAM.
T_{mem}	$\frac{\texttt{dram\_bytes\_sum}}{\texttt{memory\_throughput\_byte\_weighted\_mean}}$	Total memory bytes divided by the achieved memory throughput. Gives the cost of the involved memory operators in seconds.
T_{math}	$\frac{\texttt{sm\_active\_cycles\_sum}}{\texttt{sm\_frequency\_weighted\_mean}}$	Total active cycles divided by the achieved frequency of the Streaming Multiprocessors. Gives the cost of the involved math operators in seconds.
`FLOPs`	$\frac{\texttt{compute\_throughput\_weighted\_mean}}{100} \times$ gpu_processing_power	Total number of FLOPs executed in the scenario. Processing power is for double precision.
`arithmetic_-intensity`	$\frac{\texttt{FLOPs} \times \texttt{duration}}{\texttt{dram\_bytes\_sum}}$	The number of FLOPs executed per byte read or written to memory.

TABLE 5.2: Overview of features computed from the profiling metrics. Features beginning with "gpu" are constants for the specific GPU used.

of FLOPs required for a computation. These definitions align with those used in the profiling experiments. However, in this context, we derive them from the available data and model characteristics, rather than predicting them using a regressor, as was explained in the preceding section.

The complexity of an operator or model, as previously detailed, is equal to the mathematical costs and is determined by examining the algorithms and summing the computations performed. This process takes into account various data characteristics, such as the number of nonzero items and dataset sizes. For memory costs, a similar approach is employed, but we sum the number of bytes read and written to memory. For each Machine Learning (ML) model, we incorporate both the total summed costs and the costs per involved operator as features. This results in a multitude of additional features that can be utilized to train the cost models.

A subset of these features is presented in Table 5.3. The complete set of features is displayed in Appendix D. These features are used to train the cost models, as will be detailed in the subsequent section.

5.4 Cost Models

This section details the process of developing four different types of cost models, each capable of choosing when to favor factorization over materialization. These models are constructed using the enriched dataset, as described in the preceding section. We start with an introduction of the metrics used to evaluate the performance of the cost model (Subsection 5.4.1), followed by the models themselves. The first model, designed to be as interpretable as possible, is an analytical model (Subsection 5.4.2). Next, we fit a range of linear regressors to the dataset to create a linear ML-based cost model (Subsection 5.4.3). The third model is a tree-boosting model, used to demonstrate the performance of a more intricate model (Subsection 5.4.4).

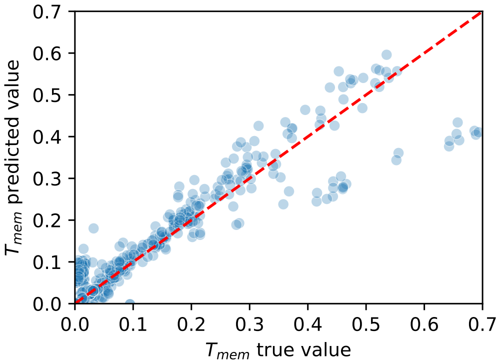


FIGURE 5.9: True T_{mem} vs predicted T_{mem} for the analytical model's *OperatorCost* function. Tested on samples not used for training. MSE: 0.0295.

The final model is a hybrid model that merges linear regression and tree-boosting models to produce a more precise model.

5.4.1 Problem Modeling and Assessing Performance

The decision whether to opt for factorization or materialization is a binary classification problem. However, it is more important to accurately predict scenarios that save more time. Therefore, we use regression models instead of classification models. This approach enables us to define the decision boundary, allowing greater flexibility in prioritizing scenarios with greater performance benefits.

Metrics to Assess Cost Model Performance

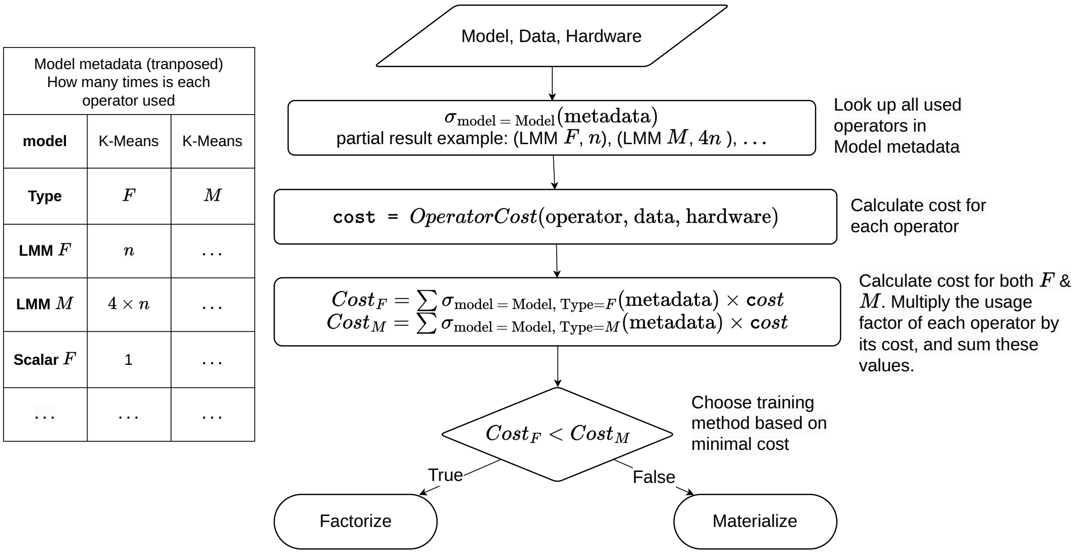
To assess the effectiveness of the cost models, we use a variety of metrics that provide insight into both the accuracy and efficiency of the models. Previous work [1], [6], [14] predominantly uses accuracy and speedup (performance ratio) to evaluate the efficacy of their cost estimation methods. However, due to the imbalanced nature of the dataset, where most scenarios lean toward materialization, accuracy is not an appropriate measure. Similarly, the performance ratio has its limitations since averaging the ratio of multiple scenarios into a single value may lead to information being lost, such as the amount of time saved, which is a more informative metric to evaluate the performance of cost models across various scenarios.

Consequently, we introduce a new metric, the time saved. It is the difference between the sum of materialized times and the sum of factorized times, for those cases where the cost models predict factorization to be faster. This metric offers a more practical assessment of the effectiveness of the model in real-world scenarios. Cases where the model predicts materialization to be faster are less relevant, as such a case will not result in a time loss compared to the normal, materialized execution. However, to evaluate whether the models exhibit adequate performance, the maximum potential time saved is also considered as a benchmark. To still allow reasoning about the achieved speedup of a set of scenarios, we adjust the performance ratio to be a weighted average according to the training time of a scenario. The final metric, used to assess generalizability as discussed in Subsection 6.2.2, is the utility of a model, which represents the proportion of time saved by using this model in relation to the maximum achievable time savings. These metrics are outlined in Table 5.4.

Dimension	Feature	Symbol	Formula
Data	Dataset size (rows, columns)	r_T, c_T	
	Sparsity	e_T	$\frac{nnz(T)}{r_T \times c_T}$
	Sparsity ratio		$\frac{e_T}{e_S}$
	Tuple ratio	τ	$\frac{\sum_{k=1}^{p} d_k}{d_S}$
	# Base tables	n	
Data & Model	Complexity ratio		$\frac{FLOP_M}{FLOP_F}$
	Memory ratio		$\frac{\text{bytes}_M}{\text{bytes}_F}$
Hardware	Compute type		
	GPU memory bandwidth		
Model	Operator		
	# Iterations	$iter$	
Dependent	Execution Time		Time_M, Time_F
	Performance ratio		$\frac{\text{Time}_M}{\text{Time}_F}$
	Time saved		$\text{Time}_M - \text{Time}_F$

TABLE 5.3: Table showing a subset of the base, and derived/engineered features used for training the cost models

5.4.2 Analytical



Model metadata (tranposed)
How many times is each operator used

model	K-Means	K-Means
Type	F	M
LMM F	n	...
LMM M	$4 \times n$	...
Scalar F	1	...
...	...	...

Model, Data, Hardware
$\sigma_{\text{model = Model}}(\text{metadata})$
partial result example: (LMM F, n), (LMM M, $4n$), ...
Look up all used operators in Model metadata
cost = $OperatorCost$(operator, data, hardware)
Calculate cost for each operator
$Cost_F = \sum \sigma_{\text{model = Model, Type}=F}(\text{metadata}) \times$ cost
$Cost_M = \sum \sigma_{\text{model = Model, Type}=M}(\text{metadata}) \times$ cost
Calculate cost for both F & M. Multiply the usage factor of each operator by its cost, and sum these values.
$Cost_F < Cost_M$
Choose training method based on minimal cost
True
False
Factorize
Materialize

FIGURE 5.10: Architecture of the Analytical model. Shows the control flow of inputs to a final decision on whether to use factorization or materialization. *OperatorCost* is the function defined in Figure 5.8.

As demonstrated in Section 5.2, the runtime of the evaluated Machine Learning scenarios is primarily determined by the time required to read and write to memory. Consequently, to generate an accurate prediction of the execution time of a scenario, we can use the memory cost as a proxy measure, an approach adopted by this analytical model.

Metric	Formula	Description
Time Saved	$\sum \text{Time}_M - \sum \text{Time}_F$ where factorization is **predicted to be faster**	The sum of the time saved for all positively predicted scenarios.
Maximum Possible Time Saved	$\sum \text{Time}_M - \sum \text{Time}_F$ where factorization **is faster**	The maximum possible time that could be saved.
Weighted Performance Ratio	$\frac{\sum \text{Time}_M}{\sum \text{Time}_F}$ where factorization is **predicted to be faster**	The ratio of the total materialization time to the total factorization time.
Utility	$U = \frac{\text{Time Saved}}{\text{Max. Time Saved}}$	The fraction of time saved when using this model, relative to the maximum time saved.

TABLE 5.4: Metrics to assess cost model performance

ANLY.1 Profiling Metrics Based model

The procedure to calculate the factorized and materialized memory cost is shown in Figure 5.10. First, we gather the operators used in the model under evaluation. Next, we employ a regression model, fitted to the collected profiling metrics (as illustrated in Figure 5.8), to predict the memory cost of the factorized and materialized operators. This step is fit, as we lack profiling metrics for the scenarios under test. Finally, we sum the costs for each training type and select the scenario with the lowest predicted runtime.

A significant limitation of this model is its dependence on a pre-trained regressor to predict the memory cost. This regressor is trained on a subset of the collected metrics and is used to predict missing metrics. As it is only fit to a subset of possible operator scenarios, it is unsuitable for predicting the cost of operations that deviate from the tested scenarios. The development of a model capable of predicting the memory cost for all scenarios is a complex undertaking and is left for future work.

ANLY.2 Model-level Memory Cost Analysis Based model

By inspecting the operations performed during ML model training, we can calculate the number of bytes read and written during computation. This is achieved by calculating the number of bytes read and written for each operation and summing these values. The memory cost is then determined by dividing the total number of bytes read and written by the GPU's memory throughput. Compared to ANLY.1, this model is more adaptable to new ML models, as it does not require a pre-trained regressor to estimate the operator cost. However, it does not consider other factors that influence memory cost, such as cache layout and hit rates.

Analytical Model Evaluation

Both analytical models yield a predicted memory cost for the factorized and materialized cases. To arrive at the final decision, we compute the ratio between the cases and opt for factorization if this ratio ($\frac{M}{F}$) exceeds a certain threshold. This threshold is fine-tuned to minimize the number of false positives while still favoring factorization in instances where it significantly outperforms materialization in terms of time efficiency. For the first analytical model (ANLY.1), we set the threshold at 1.7, and for ANLY.2, we set it at 10.0. The fact that the model only performs well with such a high ratio of materialized to factorized memory cost indicates that factorization introduces considerable overhead in areas not captured by this simplistic memory estimation.

The results of the evaluation are shown in Figure 5.11. Both models perform similarly well. The ANLY.1 model predicts much fewer false positives, but the sum of time lost by these false positives is almost the same as that of the FPs for ANLY.2. Subtracting the time lost for the FPs from the time saved for the true positives, we see that ANLY.2 saves more time than ANLY.1. This is likely due to the fact that ANLY.2 is more conservative in predicting factorization, and thus has a higher threshold for when to choose factorization over materialization. Altogether, both models save around 250s of the total 1,670s of the validation set. Breaking down the positively predicted cases by compute type, we see that a lot of time (147s & 136s respectively, shown in Figure 5.15) is lost for false positives of CPU scenarios. So, for GPU scenarios, the model is more accurate than for CPU scenarios, which is expected, as we only use memory cost as a model for runtime. We have shown this to be a valid strategy for GPUs, but for CPUs, the runtime is largely dominated by the number of FLOPs executed.

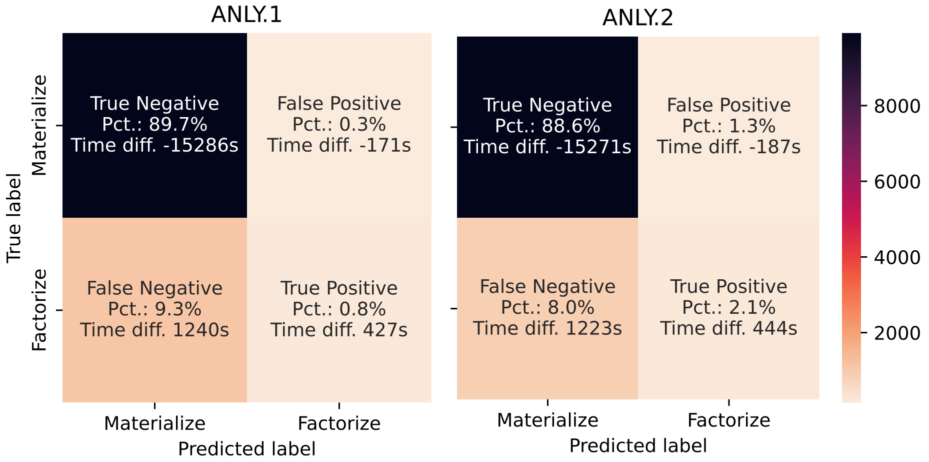


FIGURE 5.11: Confusion matrix of the analytical models' performance on the test set (CPU and GPU on synthetic data). Decision boundaries set at 1.7 and 10.0 respectively. Adding time difference of the false positive cases and the true positive cases gives the total time saved by the model. For ANLY.1 this is 256s, for ANLY.2 it is 257s.

The evaluation results are presented in Figure 5.11. Both models, ANLY.1 and ANLY.2, exhibit comparable performance. Although ANLY.1 predicts fewer false positives, the cumulative time lost due to these false positives is nearly identical to that of ANLY.2. By subtracting the time lost due to false positives from the time saved by true positives, it is observed that ANLY.2 saves more time than ANLY.1. This is likely because ANLY.2 adopts a more conservative approach to predicting factorization, thus having a higher threshold for choosing factorization over materialization.

Both models save approximately 250 seconds of the total 1,670 seconds of the validation set. When we break down the positively predicted cases by compute type, we find that a significant amount of time (147 seconds and 136 seconds, respectively, as shown in Figure 5.15) is lost due to false positives in CPU scenarios. Therefore, these models are more accurate for GPU scenarios than for CPU scenarios, which aligns with our expectation, as we use memory cost as a model for runtime. We have demonstrated this to be a valid strategy for GPUs, but for CPUs, the runtime is predominantly determined by the number of Floating Point Operations Per Second (FLOPs) executed.

5.4.3 Linear Regression

In this section, we investigate the second type of cost model, the linear regression model. The objective of this model is to maintain explainability while delivering superior performance compared to the manually tuned decision rules found in related work. We employ a range of models, all of which fundamentally rely on linear regression. We start with a single regressor and, by partitioning the dataset according to the categorical variables, we eventually arrive at more intricate models comprising ensembles of linear regression models.

The architecture of each of the linear regression models is depicted in Figure 5.12. In short, before training, the dataset is partitioned according to several categorical variables (operator type, hardware type, or F/M), or by filtering a portion of the dataset. Each final model comprises a set of linear regression models, each of which is fit to a subset of the training data. For example, for LINR.3, we train two regressors: one for the factorization scenarios and the other for the materialization scenarios. During inference, each regressor is used to predict the runtime of the scenario under test. The output is the difference between the predicted materialized time and the predicted factorized time, which is used to estimate the time saved by opting for factorization over materialization. If a categorical variable is not used to divide the dataset, it is incorporated as a feature of the model.

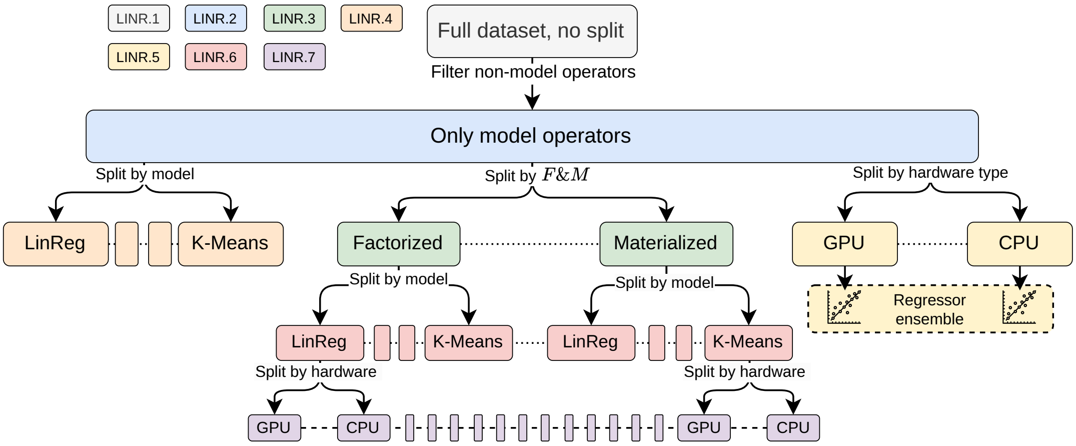


FIGURE 5.12: Architecture of the linear regression models. Shows how the data, and models, are split for each model. The final split-level belonging to each respective model is colored in the same color. For LINR.5 we show the linear regression ensemble fit to the data. For clarity, we leave this out for the other models. Each box represents a regression model, the same-colored boxes, connected via dotted lines, are combined into an ensemble to end up with the linear regression cost models.

LINR.1 Linear Regressor Fit to Full Training Set

The first linear regressor is trained on the complete training set, which includes all operators. The rationale behind this approach is the probable existence of a correlation between the performance of individual operators and the performance of the models in which they are used. By training the regressor on the entire set of operators, we strive to capture these correlations and leverage them to enhance accuracy. This model predicts the time conserved by opting for factorization over materialization.

LINR.2 Linear Regressor Fit to Model Runtimes

The second linear regressor is trained only on the runtimes of the models, and the individual operators are left out of the training set. This model helps determine whether the inclusion of operators contributes to utility. Similarly to LINR.1, this model forecasts the time conserved by opting for factorization over materialization.

LINR.3 Separate Regressors for F and M

This model is a composite of two linear regressors, one for factorization and another for materialization. By training distinct regressors for factorization and materialization, we aim to more explicitly capture the correlations between independent variables and runtime than the preceding models. Each internal regressor predicts the runtime of the scenario under test, and the final prediction is the faster predicted scenario.

LINR.4 Separate Regressors for each Model Type

Similar to LINR.3, this model is also a composite of multiple internal regressors. However, instead of having a single regressor for each factorization and materialization, we have a distinct regressor for each type of model. This is done to capture the differences in the correlations between independent variables and runtime for different model types. Like the first two models, this model forecasts the time saved by choosing factorization over materialization (by predicting whether time is conserved by choosing F).

LINR.5 Separate Regressors for CPU and GPU

In previous sections, we have shown that the choice of hardware plays a significant role in the trade-off we investigate. Therefore, it is likely that there are differences in the correlations between the independent variables and the runtime between CPU and GPU. A singular linear regression model is probably incapable of capturing these differences. Consequently, we test the performance of a set of models, one which is only fit to CPU scenarios and another which is only fit to GPU scenarios.

LINR.6 Separate Regressors for F, M and Model Type

The next version of the linear regression model we create is a combination of LINR.4 and LINR.3. By training separate regressors for every combination of factorization, materialization, and model type, we enable the models to more freely capture differences between groups.

LINR.7 Separate Regressors For Each Combination of Categories

The final, most granular, ensemble is one that has a distinct regressor for each combination of factorization, materialization, model type, and hardware. This is done to capture the differences in the correlations between the independent variables and runtime for each combination of the dimensions.

Linear Regression Model Evaluation

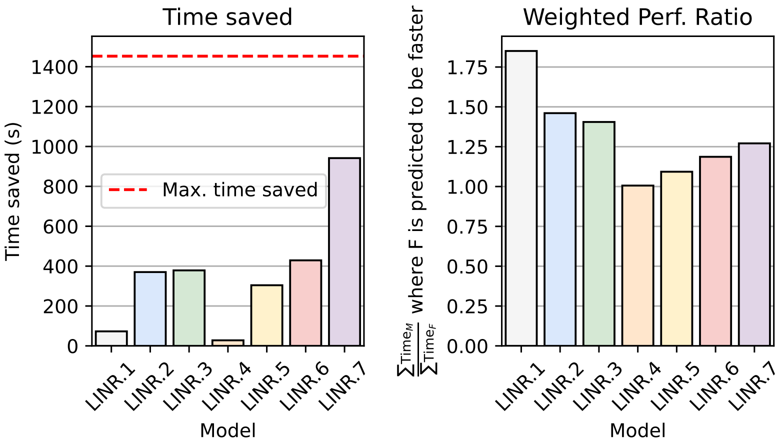


FIGURE 5.13: Evaluation of the linear regression models on the validation set (synthetic data, only models). The plots show statistics of those scenarios where the model predicts that factorization is faster. The plot on the left shows $\sum \mathrm{Time}_M - \sum \mathrm{Time}_F$, the total time saved in positively predicted scenarios. The right plots show $\frac{\mathrm{Time}_M}{\mathrm{Time}_F}$ highlighting that conservative models can result in high performance ratios with a low amount of time saved, for example LINR.7.

A performance comparison for the linear regression models is shown in Figure 5.13. We choose to highlight the cases where the model predicts factorization to be faster and evaluate the saved time in these cases. Overall, models that fit more distinct regressors perform better, showing that each of the chosen categorical variables impacts the F/M trade-off.

Models LINR.1 through LINR.3 demonstrate little time saved, but the average speedup of positively classified cases is high. This is attributable to these models producing either a substantial number of false negatives, thereby missing out on a significant amount of time saved (LINR.1), or many false positives, which reduces the total time saved (LINR.2 and LINR.3). LINR.4, which is split solely by model type, performs the worst among the models, indicating a correlation between the performance ratios of the different model types. The LINR.5 and LINR.6 models both identify a relatively large fraction of positive samples, albeit at the cost of numerous false positives and negatives. LINR.7, which is the most granular, performs the best on the **validation** set. This is likely because it can capture the differences in the relationships between the independent variables and the runtime for each combination of the dimensions. This model is the most complex, but also the most accurate, with a utility value of 0.65, indicating that it saves 65% of the maximum possible time saved.

However, when evaluating the **test** datasets (which includes real-world datasets, the train-test split is elaborated in Subsection 6.1.5), the performance of the models is considerably lower (visualized in Figure 5.15 for LINR.1 and LINR.5). This is probably due to the models overfitting to the training data (which contains only synthetic dataset scenarios). LINR.1 and LINR.5 show the best utility, which can be explained by the fact that they do not use extensive partitioning of the training data, thereby preventing the regression from fitting to the training data. To address this, we choose to evaluate the performance of a more complex model.

5.4.4 XGBoost

The third type of model we evaluate is XGBoost [19], specifically an `XGBRegressor`[1]. This model is a gradient-boosting algorithm, an ensemble learning method that utilizes a series of decision trees. We employ XGBoost due to its excellent performance demonstrated in related cost prediction scenarios [10], as well as its excellent ability to handle unbalanced datasets [21]. The model is trained on a dataset identical to the linear regression models and employs the same features.

Model	Target	Pruning	Decision Boundary
XGB.1	Runtime	All operators	$Time_M > Time_F$
XGB.2	Runtime	Only models	
XGB.3	Speedup	All operators	`speedup` > 1.0
XGB.4	Speedup	Only model	
XGB.5	Time saved	All operators	`time_saved` > 0.0
XGB.6	Time saved	Only models	

TABLE 5.5: Overview of the different configurations for the XGBoost models.

We explore a variety of configurations for this model, varying along two dimensions: their target variable, and whether all operators are included, or just the model operators, in the training dataset. The target variable is either the runtime of the scenario (with separate targets for factorized and materialized), the speedup of a scenario, or the time saved by opting for factorization over materialization. The dataset can be pruned to retain only training samples, where the operator is one of the model types (K-Means, logistic regression, linear regression, or GNMF), or all samples can be retained. By evaluating multiple models with different configurations, we aim to identify the model that best captures the relationships between independent variables and the factorization/materialization (F/M) trade-off. An overview of which model uses which configuration and how the decision to materialize or factorize is made based on the predicted value(s), is presented in Table 5.5.

A comparative evaluation of the XGB models is shown in Figure 5.14. As expected, these models, being more complex, outperform the linear regression models. Among the XGB models, there is minimal variation in performance on the validation set. There are no significant differences in performance based on the target variable used or whether all operators are included in the training set. XGB.3, which predicts the speedup of factorization over materialization, marginally exceeds the other models, accurately predicting 98% of the validation scenarios.

[1] https://xgboost.readthedocs.io/en/stable/python/python_api.html#xgboost.XGBRegressor

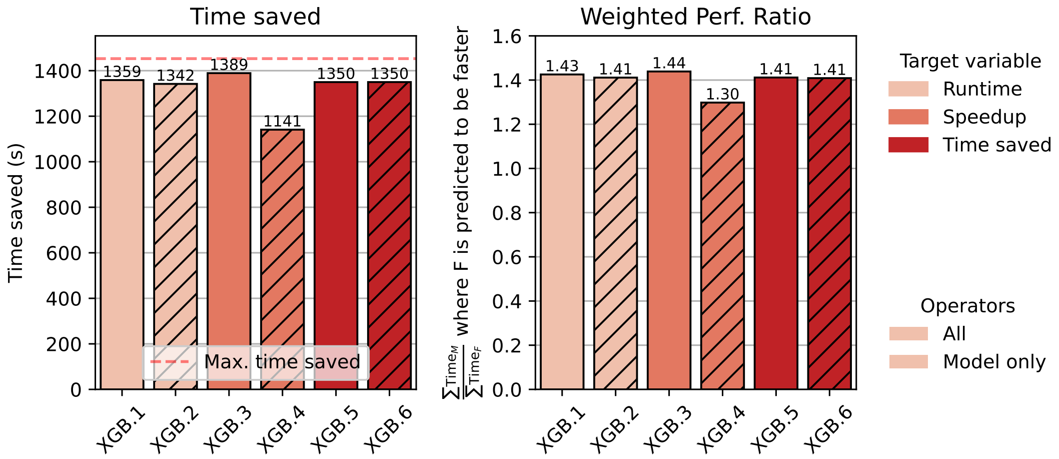


FIGURE 5.14: Evaluation of the XGBoost models on the validation set.

5.4.5 Quantitative Comparison Using Real-world Data

The performance of the two most effective models for each type is illustrated in Figure 5.15. We plot the total time saved on the test scenarios, which includes the scenarios on the Hamlet and TPCx-AI datasets. As expected, the most complex XGBoost models perform best. Specifically, XGB.3 and XGB.5 save 600 and 700 seconds, respectively, of the total 1670 seconds. The linear regression model performs the worst, probably due to overfitting to the synthetic data in the training set.

Interestingly, when we dissect the performance by compute type, the XGBoost models perform significantly better on GPU scenarios than on CPU scenarios, whereas LINR.5 performs better on CPU scenarios. This can be attributed to the fact that LINR.5 comprises two separate regressors, one for the CPU and one for the GPU, enabling it to capture the differences in the relationships between independent variables and runtime for each compute type. The process of combining these models to create a more accurate hybrid model is discussed in the following section.

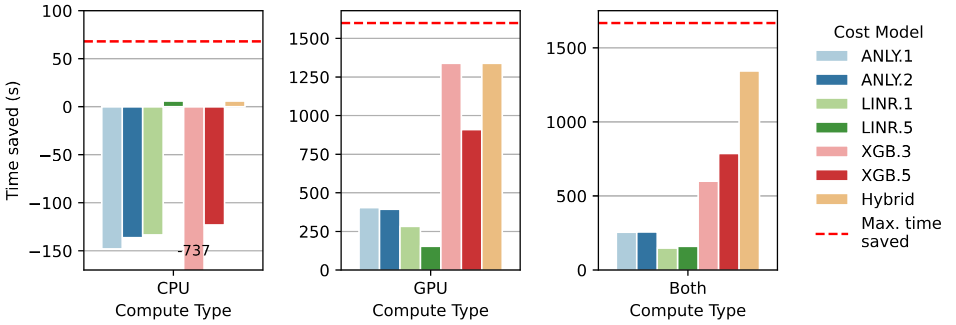


FIGURE 5.15: Comparison of the best models, broken down by compute type. Evaluated on the test set.

Combining Cost Models

To construct the final cost model, we combine the best performing models. We employ the XGB.3 model for GPU scenarios and the LINR.5 model for CPU scenarios.

This approach is adopted to capture the differences in the relationships between independent variables and runtime for each compute type. The final model is evaluated on the test set, and the results are presented in Figure 5.15. The model achieves 80% of the performance of an ideal cost model, saving 1344 seconds of the total 1670 seconds. This represents a significant improvement over the best individual models, which achieved a maximum of 47% of theoretical performance.

Qualitative Comparison

In the Machine Learning (ML) framework proposed in [4], for which this thesis develops a cost model, the decision between factorization and materialization is made at runtime. Consequently, this should not introduce any overhead to the training process. We briefly touch upon the overhead introduced by the cost of inference of the models, as well as other factors that could influence the choice between cost models. An overview of all aspects is presented in Table 5.6.

Model	Training time	Extensibility	Inference Speed	Performance
Analytical 1	−	+	−	−
Analytical 2		+	+	−
linear regression	+	−	+	−
XGBoost	+/−	−	+/−	+/−
Hybrid	+/−	−	+/−	+

TABLE 5.6: Qualitative comparison of the different models.

The first aspect we examine is the training time. For linear regression models, this is fast, but it is slower for the XGBoost model. The outliers in this case are the analytical models. ANLY.1 is notably slow as it requires fitting a large set of regressors, one for each combination of operator and F/M. In contrast, ANLY.2 does not have a traditional training time as it employs a handcrafted formula.

In terms of extensibility, the ANLY.1 models are the easiest to extend for new ML models, as they already possess an inherent "understanding" of the operators used, provided by the included profiling metrics on LA operators. For the remaining models, an updated training set is required, which includes the new ML models.

Arguably, the most crucial factor, other than performance, is inference speed, as we want to avoid overhead in the training process. ANLY.2 and the linear regression model are very quick as they are simple equations with no more than 30 terms. XGBoost, and consequently the hybrid model, is slightly slower, but still fast. ANLY.1 is the slowest here, as computing the *OperatorCost* of all involved operators is costly.

The final aspect is performance, which was discussed in the previous section.

5.4.6 Evaluating Feature Importance

To determine which characteristics are most influential in the factorization/materialization (F/M) trade-off, we examine the final hybrid model, with a particular focus on the gain per feature from the GPU leg of the model.

We present the gain of the ten most influential features of the XGBoost model in Figure 5.16. The gain quantifies the impact of a feature based on its contribution to reducing training error in the underlying decision trees. It can be interpreted as a measure of the influence a feature exerts on the model's prediction, which is insightful, as we are interested in identifying which features most significantly impact the F/M trade-off.

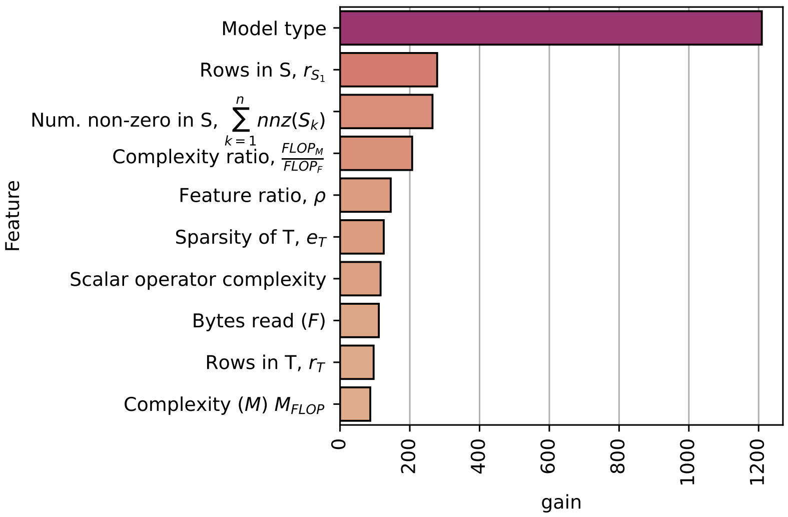


FIGURE 5.16: Feature importances of the Hybrid model, XGBoost (GPU) leg.

From the figure, we can deduce that the model type is the most impactful. This aligns with the results shown in Section 5.1, where we demonstrate that the choice of model significantly influences the F/M trade-off. Due to the operators used in each model, the speedup varies considerably between models, even when applied to the same dataset. For instance, factorized Gaussian training is beneficial twice as often as factorized K-Means. Other important features include data characteristics such as the number of rows in tables (r_{S_1}, r_T), features related to the number of zeros in a table $(e_T, nnz(S))$, or features related to the differences between the materialized and factorized case $(\rho, \frac{FLOP_M}{FLOP_F})$. We observe that although the relationship between complexity and speedup was not as clear in Section 5.1, features related to the number of operations are important, even when considering only GPUs. This is evident from the relatively high gain of the materialized and scalar complexity. This is logical, as scalar operations have high speed-ups compared to other operators.

From these gain values we can conclude that, in our experiments, differences between GPUs are less impactful than differences in data characteristics and models. The GPU characteristic with the highest gain is a categorical feature for the GPU architecture, with a gain of 7, followed by the L1 cache size and the number of Streaming Multiprocessors with gain values of 2.5 and 1.4, respectively.

Chapter 6

Evaluation

This chapter presents Contribution **C.2**: A robust cost model for Amalur's factorized ML framework, and a comparison with the SOTA. The previous chapter described the process of developing four types of cost models, and evaluated how they compared in terms of performance. This chapter extends this evaluation by comparing our best performing cost models with the SOTA, in Section 6.2, and we add a discussion on the generalizability of our best cost model, as well as a thorough analysis of the results. We end with a critical perspective on the implications and constraints of this research in Section 6.3. Prior to that, the methodology for collecting results is detailed in Section 6.1.

6.1 Experiment Setup

This section provides a comprehensive explanation of all the necessary steps to reproduce the results. It covers the experimental setup (software, datasets & hardware), as well as the data processing methods employed to guarantee reliable results for the cost models (validation strategy). For further details and implementations, please consult the GitHub repository[1].

6.1.1 Software

The factorized ML framework (Amalur [4]) is implemented in Python (3.10.4) and uses SciPy (1.8.0), NumPy (1.22.4) and CuPy (12.1.1). All experiments were run in a Docker container with an image based on NVIDIA's base image with CUDA 12.1.1 and Ubuntu 20.04[2].

The choice to use CuPy as the backend for the factorized ML framework was made to ensure that the experiments could run on both CPU and GPU. CuPy is a GPU-accelerated library for numerical computations that is compatible with NumPy and SciPy [22]. This allows for minimal changes to the codebase, whether you are using a GPU or a CPU. To allow for exploitation of multiple cores for sparse matrix multiplication[3] we use MKL (Intel Math Kernel Library) [23] as NumPy's backend for the CPU experiments.

For collecting the GPU metrics, we use NVIDIA's Nsight Compute (`ncu`)[4] which is a command-line profiling tool that collects detailed performance metrics from the

[1] `https://github.com/ptemarvelde/amalur-experiments`

[2] nvidia/cuda:12.1.1-devel-ubuntu20.04

[3] `https://github.com/flatironinstitute/sparse_dot`

[4] `https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html`

GPU. The metrics are collected in a CSV file for downstream analysis, detailed in Section 5.3. The cost models were created using Scikit-learn [24].

6.1.2 Datasets

The datasets used in the experiments are a mix of synthetic and real-world datasets. The synthetic datasets are used to generate a training set to train the cost models. The real-world datasets are used to validate the cost models on unseen data.

Synthetic Datasets

To create synthetic datasets with a wide variety of data characteristics, the data generator from [6] was used, which in turn is an adaptation of the data generator[5] from [25]. In total, we generate 2415 datasets, each being a two-table join. All other parameters were varied; the values are shown in Table 6.1.

Data Characteristic	Symbol	Range
S_1 (Entity) table rows	r_{S_1}	$[40,000,\ 1,000,000]$
S_2 (Attribute) table rows	r_{S_2}	$[526,\ 1,000,000]$
S_1 (Entity) table columns	c_{S_1}	$[1,\ 50]$
S_2 (Attribute) table columns	c_{S_2}	$[2,\ 50]$
Target table rows	r_T	$[60,000,\ 1,000,000]$
Target table columns	c_T	$[11,\ 100]$
Target Sparsity	e_T	$[0.0,\ 0.9]$
Tuple ratio	ρ	$[1,\ 190]$
Feature ratio	τ	$[0.2,\ 1]$
Join Type	j_T	Inner, left or outer.
Selectivity	σ	$[1.0,\ 2.0]$

TABLE 6.1: Ranges of data characteristics for the generated synthetic datasets

The data is organized in a star schema. S_1 is the Entity (transactional) table, which is connected to the attribute table S_2. This attribute table holds the features of the entities. The target table T is the result of the join between S_1 and S_2. An example with an inner join is shown in Figure 6.1.

Real-world Datasets

The synthetic datasets are convenient for testing and training purposes. However, to assess whether the cost models are generalizable to real-world data, we use real-world datasets for validation.

Project Hamlet [26] The Hamlet datasets are widely used in the relevant literature [1], [4], [5], [26]. Comprising a collection of seven datasets, the Hamlet datasets are tailored to simulate data integration scenarios within a machine learning workflow. Initially developed to assess inner join scenarios, certain rows from various source tables were excluded to accommodate other join types. The data attributes of these datasets are detailed in Table 6.2.

[5] https://github.com/delftdata/valentine-data-fabricator

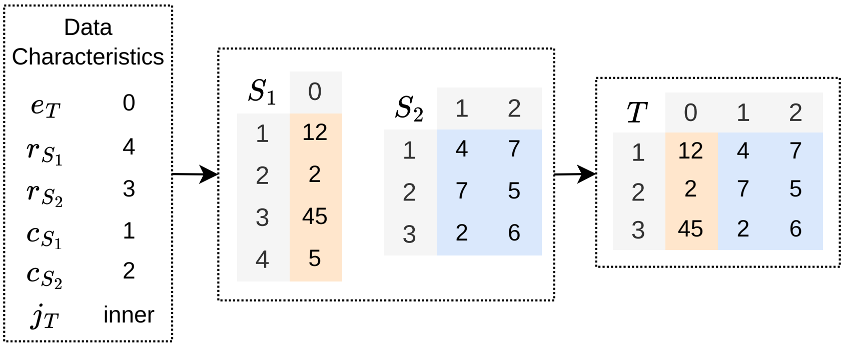


FIGURE 6.1: Example of synthetic dataset generation. The entity table S_1 is joined with the attribute table S_2 to create the target table T.

Dataset→ Characteristic ↓	Book	Expedia	Flight	Lastfm	Movie	Walmart	Yelp
r_T	253K	942K	66.5K	344K	1M	422K	216K
c_T	81.7K	52.3K	13.7K	55.3K	13.3K	2.44K	55.6K
n	2	3	4	2	2	3	2
r_{S_1}	27.9K	942K	66.5K	5K	6.04K	422K	11.5K
r_{S_2}	50K	11.9K	540	50K	3.71K	2.34K	43.9K
r_{S_3}		37K	3.17K			45	
r_{S_4}			3.17K				
c_{S_1}	28K	27	20	5.02K	9.51K	1	11.7K
c_{S_2}	53.6K	12K	718	50.2K	3.84K	2.39K	43.9K
c_{S_3}		40.2K	6.46K			53	
c_{S_4}			6.47K				

TABLE 6.2: Hamlet dataset characteristics. r is the number of rows, c is the number of columns, and n is the number of tables. Subscripts denote which table the characteristic belongs to.

TPCx-AI [3] We also assess a more practical scenario than what the Hamlet datasets provide, which is derived from a real-world benchmark used for evaluating end-to-end ML platforms. Since this aspect is not the primary focus of this study, we utilized only two of the ten use cases provided, specifically the first and the tenth. This benchmark includes a data generator with the ability to scale generation by adjusting scale factors ranging from 0.01 to 0.5, leading to the creation of 18 datasets for each use case. Details on the data characteristics of these datasets can be found in Figure 6.2.

The first use case, which involves a join operation among three tables, has been covered in the running example. First, the lineitem table is combined with the order returns data, which is subsequently linked with the orders table. This process yields a table with $c_T = 7$ columns, each row representing a product sold in an order. In the TPCx-AI benchmark, customer segmentation is accomplished through K-Means clustering on this dataset. However, we evaluate all four machine learning models outlined in this thesis, focusing on the training procedure rather than the final classifier's effectiveness. The following scenario involves a join between two tables. Here, the transaction table is merged with the customer table, resulting in a table

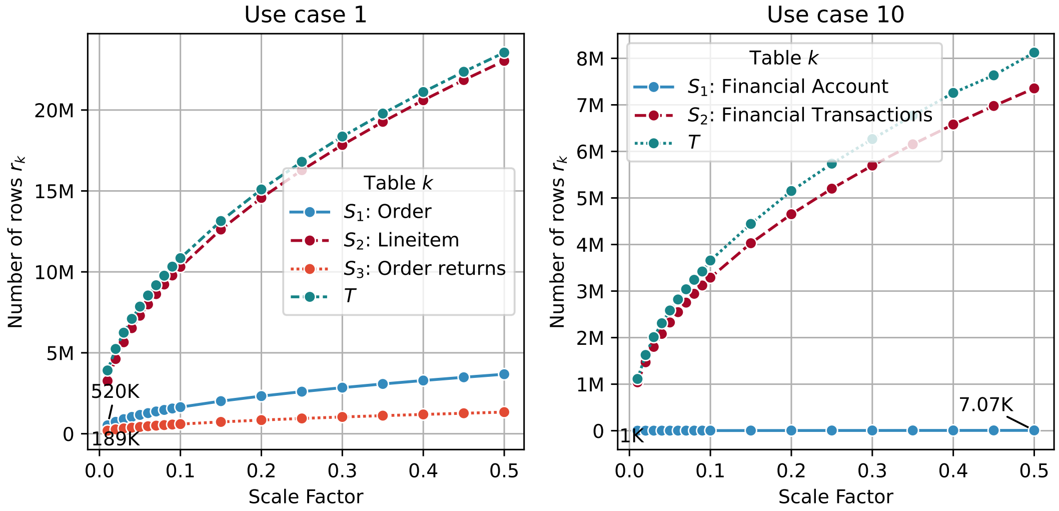


FIGURE 6.2: TPCx-AI dataset sizes for used scale factors. The number of columns is independent of the scale factors. For use case 1: $c_1 = 3, c_2 = 4, c_3 = 3, c_T = 7$. For use case 10: $c_1 = 2, c_2 = 7, c_T = 5$.

with $c_T = 5$ columns aimed at identifying fraudulent transactions. The same models utilized in the first scenario are applied in this context as well. For a complete overview of the dataset schema, including the relevant tables and columns, refer to Figure A.1.

6.1.3 Hardware

The experiments were carried out on a wide range of machines to test various hardware configurations. Most of the experiments were run on the Delft AI Cluster[6], allowing for testing across different GPU architectures. As profiling was not feasible on this cluster, some profiling experiments were carried out on a local machine, AWS, and resources from the Web Information Systems group[7]. Details of which experiments were carried out on the specific machines can be found in Table 6.3.

6.1.4 Experiment Setting

To guarantee the reliability of the training data, each experiment was performed with a repetition count of 30. The profiling experiments, on the other hand, were not replicated, as NCU guarantees consistent and actionable results by replaying kernel launches [27]. All experiments were carried out in a containerized setting to ensure reproducibility. The profiling experiments used the same image as the runtime experiments to maintain consistency in the environment. Docker[8] served as the container runtime, except on the DAIC cluster, where the usage of Apptainer[9] images was mandated.

[6] https://daic.tudelft.nl/
[7] https://www.wis.ewi.tudelft.nl/
[8] https://www.docker.com/
[9] https://apptainer.org/

Experiment type	Machine	Architecture	Compute Unit	Experiment
profile	WIS ST4	Ampere	GPU A40	`GPU-P-1`
	AWS G5.xlarge	Ampere	GPU A10G	`GPU-P-2`
	Personal Workstation	Turing	GPU 1660Ti	`GPU-P-3`
runtime	DAIC	Ampere	GPU A40	`GPU-T-1`
		Volta	GPU V100	`GPU-T-2`
		Pascal	GPU P100	`GPU-T-3`
		Turing	GPU 2080Ti	`GPU-T-4`
		Pascal	GPU 1080Ti	`GPU-T-5`
	WIS ST4	—	EPYC 7H12 CPU 8 cores	`CPU-T-1`
			EPYC 7H12 CPU 16 cores	`CPU-T-2`
			EPYC 7H12 CPU 32 cores	`CPU-T-3`

TABLE 6.3: Experiment to machine mapping. The experiment type is either profiling or runtime. Profiling experiments are used to collect the hardware specific metrics for our training data. Runtime experiments are used to gather data on the runtime of the factorized ML framework compared to materialized learning.

6.1.5 Validation Strategy

Given the vast range of potential data, model, and hardware features, it is crucial to have confidence in our model's ability to provide accurate predictions for new situations. To test this, we used a rigorous train-validate-test split. 70% of the data from synthetic datasets is allocated to the training set, while the remaining 30% forms the validation set. The real-world datasets are exclusively reserved for testing purposes. Since these datasets are not used in training the cost models, they offer a reliable assessment of performance in novel scenarios, as discussed in Subsection 6.2.2. Similarly, to ensure robustness in the hardware aspect, we adopt a comparable strategy by isolating the samples of `GPU-T-5` for testing. Lastly, to explore the dimension of model characteristics, we perform an ablation study (see Section 6.2.2) to investigate the impact of excluding various models on the cost model's performance.

6.2 Cost Model Performance and Comparative Analysis

In this section we answer

RQ.2 How can we accurately predict the optimal choice between factorized or materialized training of a Machine Learning model, on CPU and GPU, through leveraging knowledge about model, data, and hardware characteristics?

We first illustrate the comparison of the cost models with the state-of-the-art in cost estimation for factorized Machine Learning training. Next, we demonstrate the ability of the best performing cost models to generalize effectively to novel scenarios.

6.2.1 Cost Model Comparison

We compare our best performing cost models from each type with the SOTA. Using the test set described previously, which includes real datasets and new hardware configurations, we evaluate the performance of the cost models. Figure 6.3 presents the results of this comparative assessment. The plot on the left shows the precision, accuracy, f1 score and recall score of the cost models. The right plot quantifies the total time saved by the cost models across the set of scenarios.

From the figure we can infer how each model behaves with regard to the precision/recall trade-off. The SOTA cost models show diminished precision, yet enhanced recall, showing a tendency toward favoring predicting a positive label, which in this case means that the factorized training is faster. MorpheusFI [14] shows the best recall, explained by precision and accuracy, labeling all cases as positive, resulting in a total time loss of 13,000 seconds. Morpheus [1] presents a smaller time loss of approximately 7000 seconds, reflecting a slightly more conservative approach than MorpheusFI. Amalur [4] shows a total time loss of approximately 500 seconds, outperforming Morpheus and MorpheusFI. The Hybrid cost model introduced in this thesis has a net time savings of **1350** seconds, with an accuracy of **95**%, which is a significant improvement over the SOTA.

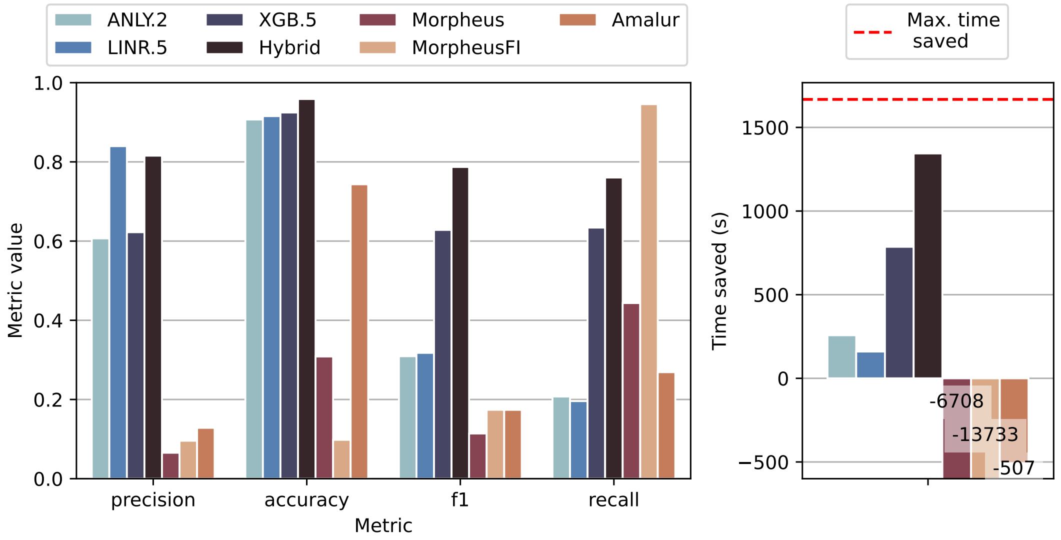


FIGURE 6.3: Comparison of the cost models. Performance evaluated on the test set (real datasets and new hardware).

We have shown stellar performance across the test set; next we will explore the generalizability of our two best cost models, XGB.5 and the Hybrid model.

6.2.2 Exploring Generalizability

The aim of this thesis is to create a cost model capable of reliably predicting the most suitable option between factorized and materialized training methods for a Machine Learning model. To ensure that this cost model maintains good performance in unfamiliar situations, i.e., it is not overfitting to the training data, we evaluate the cost models on the real-world datasets, and on new hardware. The results are shown in Figure 6.4.

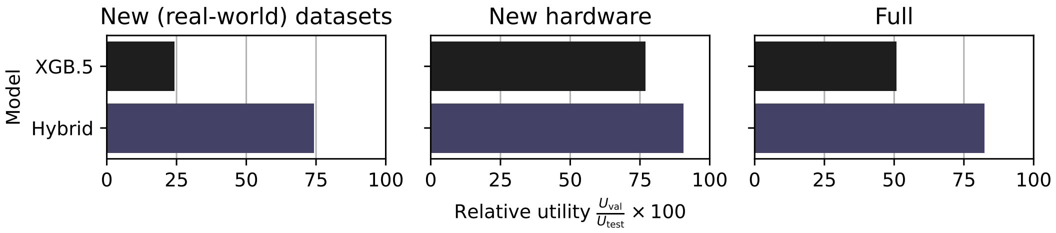


FIGURE 6.4: Evaluation and comparison of cost model's relative utility on new scenarios. The x-axis shows the relative utility. Recall that utility is defined as $U = \frac{\text{Time Saved}}{\text{Max. Time Saved}}$. With the relative utility we can see how a model performs on the test set compared to the validation set, with regard to the time saved. Relative utility is defined as $\frac{U_{\text{val}}}{U_{\text{test}}} \times 100$. A value of 100% means that the cost model performs equally well on the test set as on the training set.

Significant disparities exist among the models in terms of their ability to generalize to unfamiliar situations. The XGBoost model excels in terms of relative performance on new hardware, but underperforms on real-world datasets. The hybrid model combines the strengths of the XGBoost and linear regression models, achieving good relative performance of 82% on the full dataset.

Ablation Study

To assess the generalizability of our cost model on new model types, we performed an ablation study. In this study, we systematically exclude individual model types and evaluate the model performance in test set scenarios exclusive to the omitted model type.

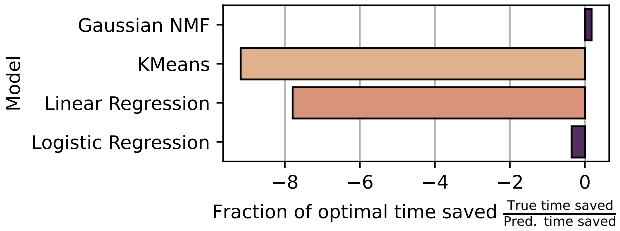


FIGURE 6.5: Result of the leave-one-out ablation study of the Hybrid model. The x-axis shows the relative performance of the cost model, defined as total time saved, with regard to the test set, for each group.

The results of this ablation study are illustrated in Figure 6.5. The x-axis represents the relative performance compared to an ideal model. The findings indicate that our hybrid model exhibits suboptimal performance when applied to new model types. While the Gaussian Non-negative Matrix Factorization (NMF) demonstrates marginal utility, facilitating some time savings, the application of our cost model to other models, notably K-Means and linear regression, would lead to considerable increases in computational time. This suggests that our cost model is specifically tailored to the model types included in its training dataset. The inclusion of linear algebra operators within the training set confers additional benefits, as evidenced in Figure 5 (Figure 5.14); however, these benefits are insufficient to achieve model independence in cost estimation.

To mitigate this limitation, we propose two potential strategies: First, refining the feature engineering process to incorporate a broader spectrum of model-agnostic derived features, necessitating a comprehensive analysis of the factorized machine learning framework and its supported models. Second, and perhaps more effectively, expanding the training dataset to include emerging models as they are integrated into the factorized machine learning framework.

6.3 Discussion

Our research demonstrates that the Hybrid cost model adeptly predicts the most efficient training method -factorized or materialized- for machine learning models across CPU and GPU scenarios. This model shows commendable generalizability to novel scenarios and surpasses state-of-the-art (SOTA) models in the domain of cost estimation for factorized ML training. However, this study is not without limitations.

The first important note is that the SOTA cost models lack design considerations for GPU-based training. This oversight is non-trivial, as our findings reveal a significant divergence in the factorization-materialization trade-off between CPU and GPU environments. But this still skews the comparison in favor of our models, as we do include GPU-specific features in our cost models.

Another limitation is that the cost models were trained on synthetic data. While synthetic data generation aims to replicate real-world datasets, discrepancies with real-world scenarios are inevitable. Our models have demonstrated proficiency with actual data; however, the potential for overfitting to synthetic data cannot be dismissed, and the robust performance observed with datasets such as Hamlet and TPCx-AI may not be entirely indicative of broader applicability. This concern extends to the hardware aspect, where the models show promising adaptability to new hardware configurations, yet the question of how representative these configurations are of real-world settings remains uncertain.

Despite these limitations, we believe that our contributions to cost estimation for ML training are substantial. The Hybrid cost model yields encouraging results and consistently outperforms SOTA in a multitude of scenarios. We hope that this work will inspire continued exploration in this field.

6.4 Conclusion

This chapter has presented a comprehensive evaluation of our hybrid cost model for factorized and materialized ML training. We have shown the model's ability to accurately discern the optimal training approach and its superiority over SOTA in real-world scenarios. Although the model adapts well to new hardware and data parameters, its predictive utility for scenarios involving new ML models is limited.

The next chapter will explore future work and potential improvements for our cost model.

Chapter 7

Conclusion

In closing, we consolidate the key contributions and discoveries of this thesis in Section 7.1. Additionally, we acknowledge the limitations and outline potential areas for future research in Section 7.2.

7.1 Cost Estimation for Factorized Machine Learning

Our research explores the dynamics of cost estimation for factorized machine learning, focusing on the comparative performance of GPUs versus CPUs. We find that GPU training exhibits distinct cost characteristics from CPU training, which significantly influences cost model design and optimization of factorized machine learning processes.

Previous cost estimation methodologies have predominantly centered on CPU contexts, resulting in inaccuracies when extrapolated to GPU environments. Our analysis reveals a pronounced difference in the speedup of factorized model training between CPU and GPU platforms. This discrepancy is due to the distinct architectural designs and processing capabilities of GPUs, which require a customized approach to cost estimation.

Through empirical research and extensive experimentation, we have formulated an innovative cost model that is tuned to the nuances of GPU computation. Our model diverges from existing methods by incorporating a deeper understanding of GPU architecture, and by leveraging a more comprehensive set of features to decide the training approach.

By accounting for the unique cost factors associated with GPU usage, we provide a more reliable framework for predicting whether factorization is beneficial. The results of our comparative analysis demonstrate that our cost model outperforms existing methods, both for CPU and GPU scenarios. In the tested scenarios on real-world datasets, for which a perfect model would result in 1667 seconds saved, the SOTA cost model achieves a time loss of 507s. Using our hybrid cost model would save 1350s, reaching almost 80% of the maximum possible utility.

This progress in cost estimation facilitates the broader adoption of factorized machine learning within the industry, enabling considerable time savings in scenarios that use intensive training. Such scenarios include training large models, hyperparameter optimization, and real-time training. The impact of using our cost model in the ML workflow is minimal, but when factorization is faster, we achieve an average speedup of 3.8×. The largest hurdle for a Data Scientist to use this approach is the adaption of their data integration workflow, so it fits into the factorized ML framework, which currently is a manual process.

Despite promising results, our model comes with certain limitations. The need to account for the unique characteristics of GPU computation introduces complexity into our model, which makes its predictions less explainable than the state-of-the-art models. Another risk associated with the added complexity is overfitting to the current implementation. How our cost model performs when another implementation of factorized learning is used for training is uncertain. Furthermore, the introduction of new machine learning models requires additional work to adapt our cost model accordingly. These challenges highlight areas for future research and improvement. Nevertheless, the benefits of our model in terms of accuracy and efficiency make it a valuable contribution to the field of factorized machine learning.

7.2 Future Work

As we look forward, there are several promising directions for future work. This thesis has demonstrated the value of factorized machine learning in real-world settings, but to facilitate its adoption in the industry, steps must be taken. We believe there are two valuable areas for future research: the integration of factorized machine learning into widely used frameworks and the continuation of cost estimation efforts.

7.2.1 Integration of Factorized Training into ML Frameworks

Integrating factorized training in a widely used ML framework such as TensorFlow or PyTorch would improve the practicality and reach of factorized machine learning, as well as open up opportunities for investigating cost estimation. Given the maturity of these frameworks and the extensive research already conducted to optimize their training processes, this could significantly advance our understanding of cost dynamics in factorized machine learning.

Moreover, this integration also aids in the exploration of factorized machine learning in a distributed setting. This would be a significant advancement, as it would allow us to leverage the power of distributed computing to further enhance the efficiency and scalability of factorized machine learning models. This could potentially lead to advances in handling larger datasets and more complex computations, thus broadening the scope and impact of factorized machine learning.

7.2.2 Exploring Cost Estimation Further

In terms of future steps specifically for cost estimation in factorized machine learning, we propose two main areas of focus. Firstly, other types of cost models could be explored, such as those based on micro benchmarking. This would involve performing performance tests on individual operations, at training time, right before running the training algorithm. The insights gained from these benchmarks could then be used to make informed decisions between materialization and factorization. This could improve accuracy as the actual datasets can be used for these benchmarks. However, consideration must be given to keeping the overhead of such an approach low. Another direction that could complement our proposed approach is the exploration of online training. This would involve continuously updating the model as new scenarios are tested, leading to a continuously improving model. Such an approach would be particularly valuable in a real-world factorized machine learning framework, allowing users to benefit from other users' insights.

Secondly, one could expand on the profiling experiments conducted in this thesis. By conducting more extensive profiling experiments, on model training scenarios instead of individual operators, we could gain a deeper understanding of the cost dynamics of factorized machine learning. This would allow us to refine our cost model further and potentially identify new cost factors that could be incorporated into the model. However, this would require a significant investment in time and resources, as profiling experiments are time-consuming and computationally expensive.

Despite the challenges, such as higher complexity and the need for additional work with the introduction of new machine learning models, our model makes a valuable contribution to the field in terms of accuracy and efficiency. These future directions highlight the potential for continued refinement and expansion of our cost model, contributing to the ongoing advancement of factorized machine learning.

Appendix A

TPCx-AI Dataset Schema

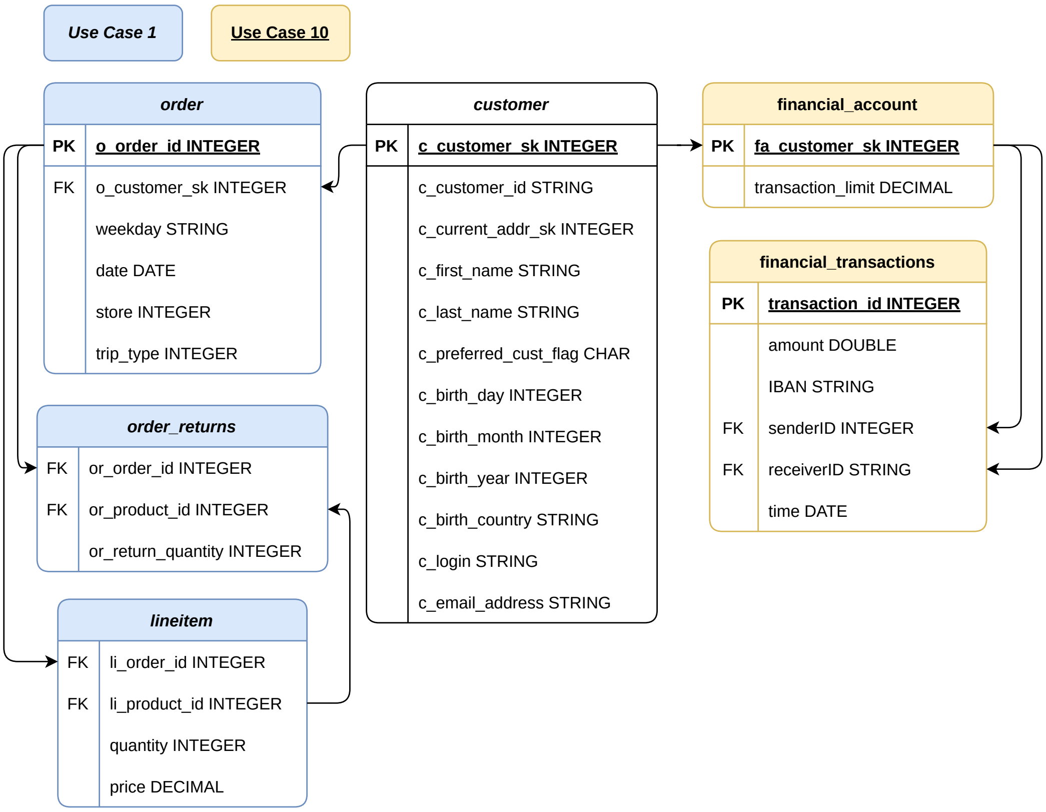


FIGURE A.1: Simplified schema from the TPCx-AI [3] benchmark. Only schemas used in experiments are shown.

Appendix B

GPU Characteristics

	GPU →		P100	1080Ti	V100	2080Ti	1660Ti	A40	A10G
Group ↓	Unit ↓	Characteristic↓							
		Architecture	Pas.	Pas.	Vol.	Tur.	Tur.	Amp.	Amp.
		Number of SM	56	28	80	68	24	84	72
		Cores	3,584	3,584	5,120	4,352	1,536	10,752	9216
Cache Size	KB/SM	L1	24	48	128	64	64	128	128
	MB	L2	4.0	2.8	6.2	5.5	1.5	6.0	6
Clock Speed	MHz	Base	1,126	1,480	1,230	1,350	1,500	1,305	1320
		Max Boost	1,303	1,582	1,370	1,545	1,770	1,740	1710
Memory	bit	Bus Width	4,096	352	4,096	352	192	384	384
	GB	Size	16	11	32	11	6	48	24
	MT/S	Clock	1,430	11,000	1,750	14,000	12,000	7,248	6,252
	GB/s	Bandwidth	732	484	900	616	288	696	600
Processing Power	TFLOPS	Half Precision	21.20	0.17	112.22	23.50	9.22	149.68	31.52
		Single Precision	10.60	10.61	14.03	11.75	4.61	37.42	31.52
		Double Precision	5.30	0.33	7.01	0.32	0.14	0.58	0.99

TABLE B.1: Hardware characteristics used GPUs. Architecture abbrevations: Pas. = Pascal, Vol. = Volta, Tur. = Turing, Amp. = Ampere.

Appendix C

GPU analysis additional figures

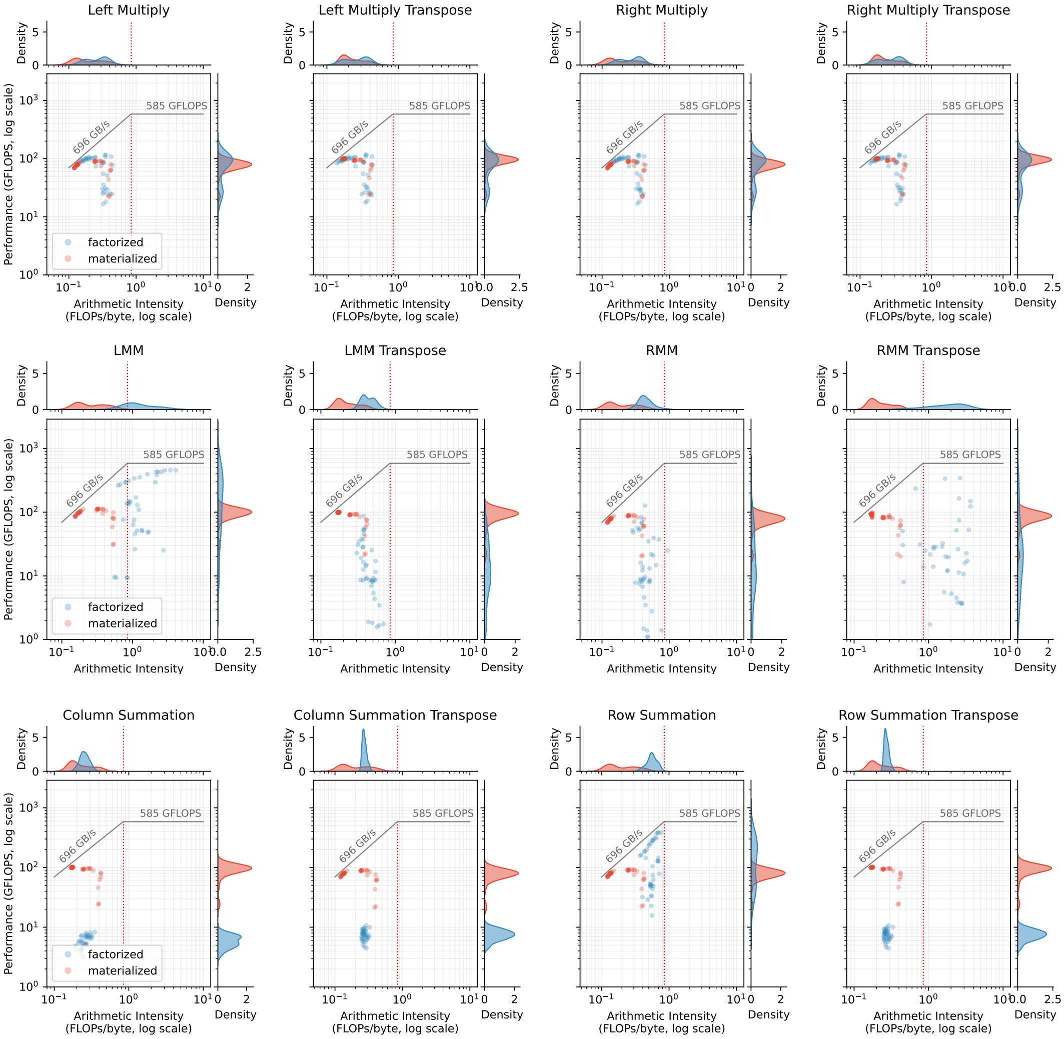


FIGURE C.1: Full set of visualizations showing roofline charts per operator (A40 GPU). Note: the transpose column summation figure should look like the row summation (non transpose) figure. It is similar to the regular column sums due to a big in the implementation that has since been resolved. This does not affect the runtime scenarios for ML models as this operator is not used in any of the tested models.

Appendix D

Full Cost Model Feature Set

Dimension	Feature	Symbol	Formula	Type	Notes
Data	Dataset size (rows, columns)	r_T, c_T		N	
	Feature ratio	ρ	$\frac{n_S}{\sum_{k=1}^{p} n_k}$	N	
	Join type	j_t		C	
	Selectivity	σ	$\frac{\sum_{k=1}^{n} r_{S_k}}{r_T}$	N	
	Sparsity	e_T	$\frac{nnz(T)}{r_T \times c_T}$	N	
	Sparsity ratio		$\frac{e_T}{e_S}$	N	
	Tuple ratio	τ	$\frac{\sum_{k=1}^{p} d_k}{d_S}$	N	
	# Base tables	n		N	
	# Nonzero values	$nnz(T)$	$nnz(S) = \sum_{k=1}^{n} nnz(S_k)$	N	
	# Sparse base tables ($e < 0.05$)	q	$\|\{S_k \in S \| e_{S_k} < 0.05\}\|$	N	From [14]
Data & Model	Complexity	M_{FLOP}, F_{FLOP}		N	For each operator
	Complexity ratio		$\frac{FLOP_M}{FLOP_F}$	N	
	Memory bytes sum	bytes$_M$, bytes$F$		N	For each operator
	Memory ratio		$\frac{\text{bytes}_M}{\text{bytes}_F}$	N	For each operator
Hardware	Arithmetic intensity		$\frac{\#ops}{\#bytes}$	N	
	Compute type			C	CPU, GPU
	Compute unit			C	CPU with number of cores or GPU type
	FLOPs	$\#ops$		N	
	GPU memory bandwidth			N	
	GPU processing power (double precision)			N	
	Math cost	T_{math}		N	
	Memory cost	T_{mem}		N	
	Total bytes	$\#bytes$		N	
Model	Operator			C	
	Rank	r		N	GNMF
	# Clusters	k		N	K-MEans
	# Iterations	$iter$		N	
Dependent	Execution Time		Time$_M$, Time$_F$	N	
	Performance ratio		$\frac{\text{Time}_M}{\text{Time}_F}$	N	
	Time saved		Time$_M$ − Time$_F$	N	

TABLE D.1: Table showing base, and derived/engineered features used for training the cost models. N stands for numerical, C for categorical. For each operator means that for each operator used in the model, the feature is calculated.

Bibliography

[1] A. Kumar, J. Naughton, and J. M. Patel, "Learning generalized linear models over normalized data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 1969–1984, ISBN: 9781450327589. DOI: 10.1145/2723372.2723713. [Online]. Available: https://doi.org/10.1145/2723372.2723713.

[2] C. J. Date, *The Relational Database Dictionary: A Comprehensive Glossary of Relational Terms and Concepts, with Illustrative Examples*, English, Paperback. O'Reilly Media, Sep. 7, 2006, p. 128, ISBN: 978-0596527983. [Online]. Available: https://lead.to/amazon/com/?op=bt&la=en&cu=usd&key=0596527985.

[3] C. Brücke, P. Härtling, R. D. E. Palacios, H. Patel, and T. Rabl, "Tpcx-ai - an industry standard benchmark for artificial intelligence and machine learning systems," *Proc. VLDB Endow.*, vol. 16, no. 12, pp. 3649–3661, Sep. 2023, ISSN: 2150-8097. DOI: 10.14778/3611540.3611554. [Online]. Available: https://doi.org/10.14778/3611540.3611554.

[4] R. Hai, C. Koutras, A. Ionescu, *et al.*, "Amalur: Data integration meets machine learning," in *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, 2023, pp. 3729–3739. DOI: 10.1109/ICDE55515.2023.00301.

[5] L. Chen, A. Kumar, J. Naughton, and J. M. Patel, "Towards linear algebra over normalized data," *PVLDB*, vol. 10, no. 11, 2017.

[6] Z. Li, W. Sun, D. Zhan, *et al.*, "Amalur: The convergence of data integration and machine learning," *IEEE Transactions on Knowledge and Data Engineering*, 2024.

[7] A. Kumar, M. Boehm, and J. Yang, "Data management in machine learning: Challenges, techniques, and systems," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 1717–1722, ISBN: 9781450341974. DOI: 10.1145/3035918.3054775. [Online]. Available: https://doi.org/10.1145/3035918.3054775.

[8] N. Jangamreddy, "A survey on specialised hardware for machine learning," en, 2019. DOI: 10.13140/RG.2.2.20697.26725. [Online]. Available: http://rgdoi.net/10.13140/RG.2.2.20697.26725.

[9] *Cuda c++ programming guide*, https://docs.nvidia.com/cuda/archive/11.2.0/pdf/CUDA_C_Programming_Guide.pdf, (Accessed on 03/07/2024).

[10] T. Chen, T. Moreau, Z. Jiang, *et al.*, "Tvm: An automated end-to-end optimizing compiler for deep learning," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594, ISBN: 9781931971478.

[11] *Gpu performance background user's guide — nvidia docs*, https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html, (Accessed on 02/20/2024), Feb. 2023.

[12] A. Kumar, M. Jalal, B. Yan, J. Naughton, and J. M. Patel, "Demonstration of Santoku: Optimizing machine learning over normalized data," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1864–1867, Aug. 2015. DOI: 10.14778/2824032.2824087.

[13] M. Schleich, D. Olteanu, and R. Ciucanu, "Learninglinear regression models over factorized joins," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 3–18, ISBN: 9781450335317. DOI: 10.1145/2882903.2882939. [Online]. Available: https://doi.org/10.1145/2882903.2882939.

[14] S. Li, L. Chen, and A. Kumar, "Enabling and Optimizing Non-linear Feature Interactions in Factorized Linear Algebra," in *Proceedings of the 2019 International Conference on Management of Data*, Amsterdam Netherlands: ACM, Jun. 25, 2019, pp. 1571–1588. DOI: 10.1145/3299869.3319878. [Online]. Available: https://dl.acm.org/doi/10.1145/3299869.3319878.

[15] Z. Cheng, N. Koudas, Z. Zhang, and X. Yu, "Efficient construction of nonlinear models over normalized data," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, IEEE, 2021, pp. 1140–1151. DOI: 10.1109/ICDE51399.2021.00103. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00103.

[16] D. Justo, S. Yi, L. Stadler, N. Polikarpova, and A. Kumar, "Towards a polyglot framework for factorized ML," *Proceedings of the VLDB Endowment*, vol. 14, no. 12, pp. 2918–2931, Jul. 1, 2021, ISSN: 2150-8097. DOI: 10.14778/3476311.3476372. [Online]. Available: https://doi.org/10.14778/3476311.3476372 (visited on 06/15/2022).

[17] A. Adams, K. Ma, L. Anderson, *et al.*, "Learning to optimize halide with tree search and random programs," *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019, ISSN: 0730-0301. DOI: 10.1145/3306346.3322967. [Online]. Available: https://doi.org/10.1145/3306346.3322967.

[18] *What is halide? why should i use it? — learning halide by examples documentation*, https://people.csail.mit.edu/tzumao/tmp/learning_halide/what_is_halide.html, (Accessed on 06/15/2023).

[19] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16, San Francisco, California, USA: ACM, 2016, pp. 785–794, ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939785. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939785.

[20] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[21] P. Zhang, Y. Jia, and Y. Shang, "Research and application of xgboost in imbalanced data," *International Journal of Distributed Sensor Networks*, vol. 18, no. 6, p. 15 501 329 221 106 935, 2022. DOI: 10.1177/15501329221106935. eprint: https://doi.org/10.1177/15501329221106935. [Online]. Available: https://doi.org/10.1177/15501329221106935.

[22] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. [Online]. Available: http://learningsys.org/nips17/assets/papers/paper_16.pdf.

[23] E. Wang, Q. Zhang, B. Shen, *et al.*, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*. Cham: Springer International Publishing, 2014, pp. 167–188, ISBN: 978-3-319-06486-4. DOI: 10.1007/978-3-319-06486-4_7. [Online]. Available: https://doi.org/10.1007/978-3-319-06486-4_7.

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[25] C. Koutras, G. Siachamis, A. Ionescu, *et al.*, "Valentine: Evaluating matching techniques for dataset discovery," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, 2021, pp. 468–479. DOI: 10.1109/ICDE51399.2021.00047.

[26] A. Kumar, J. F. Naughton, J. M. Patel, and X. Zhu, "To join or not to join?: Thinking twice about joins before feature selection," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds., ACM, 2016, pp. 19–34. DOI: 10.1145/2882903.2882952. [Online]. Available: https://doi.org/10.1145/2882903.2882952.

[27] *Nsight compute :: Nsight compute documentation*, https://docs.nvidia.com/nsight-compute/NsightCompute/index.html, (Accessed on 06/20/2023).