Distributed and Parallel Databases (2022) 40:205–259
https://doi.org/10.1007/s10619-022-07417-7

Recursive SQL and GPU-support for in-database machine learning

Maximilian E. Schüle[1] · Harald Lang[1] · Maximilian Springer[1] · Alfons Kemper[1] · Thomas Neumann[1] · Stephan Günnemann[1]

Accepted: 21 June 2022 / Published online: 9 July 2022

Abstract

In machine learning, continuously retraining a model guarantees accurate predictions based on the latest data as training input. But to retrieve the latest data from a database, time-consuming extraction is necessary as database systems have rarely been used for operations such as matrix algebra and gradient descent. In this work, we demonstrate that SQL with recursive tables makes it possible to express a complete machine learning pipeline out of data preprocessing, model training and its validation. To facilitate the specification of loss functions, we extend the code-generating database system Umbra by an operator for automatic differentiation for use within recursive tables: With the loss function expressed in SQL as a lambda function, Umbra generates machine code for each partial derivative. We further use automatic differentiation for a dedicated gradient descent operator, which generates LLVM code to train a user-specified model on GPUs. We fine-tune GPU kernels at hardware level to allow a higher throughput and propose non-blocking synchronisation of multiple units. In our evaluation, automatic differentiation accelerated the runtime by the number of cached subexpressions compared to compiling each derivative separately. Our GPU kernels with independent models allowed maximal throughput even for small batch sizes, making machine learning pipelines within SQL more competitive.

Keywords SQL · GPU · In-database machine learning · Code-generation

1 Introduction

Typically, steps of machine learning pipelines—that consist of data preprocessing [1, 2], model training/validation [3] and finally its deployment on unlabelled data [4]—are embedded in Python scripts that call up specialised tools such as

✉ Maximilian E. Schüle
m.schuele@tum.de

Extended author information available on the last page of the article

NumPy, TensorFlow, Theano or Pytorch. Hereby, especially tensor operations and model training are co-processed on graphical processing units (GPUs) or tensor processing units (TPUs) developed for this purpose.

Integrating machine learning pipelines into database systems is a promising approach for data-driven applications [5–9]. Even though specialised tools will outperform general-purpose solutions, we argue that an integration in database systems will simplify data provenance and its lineage, and allows complex queries as input. So far, machine learning pipelines inside of database queries are assembled from user-defined functions [10–14] and operators of an extended relational algebra. This brings the model close to the data source [15] with SQL [16] as the only query language. As modern HTAP main-memory database systems such as SAP HANA [17], HyPer [18–21] and Umbra [22–27] are designed for transactional and analytical workload, this allows the latest database state to be queried [28, 29]. But for continuous machine learning based on the latest tuples, only stand-alone solutions exist [30, 31] whose pipelines retrain weights for a model partially [32] when new input data is available.

In the last decade, research on database systems has focused on GPU co-processing to accelerate query engines [33–36]. GPUs, initially intended for image processing and parallel computations of vectorised data, also allow general-purpose computations (GPGPU). In the context of machine learning, matrix operations [37] and gradient descent [38, 39] profit from vectorised processing [40] available on GPUs [41]. Vectorised instructions accelerate model training and matrix computations—as the same instructions are applied elementwise. When a linear model is trained, vectorised instructions allow the loss as well as the gradient to be computed for multiple tuples in parallel.

We argue that SQL is sufficient to formulate a complete machine learning pipeline. Our database system, Umbra, is a computational database engine that offers—in addition to standardised SQL:2003 features—a matrix datatype, a data sampling operator and continuous views [42]. A continuous view [43–45] updates precomputed aggregates on incoming input. This kind of view—combined with sampling [46–50]—can be used as source to train and retrain a model partially within a recursive table.

This work starts by expressing gradient descent as a recursive table as well as the views needed for data preprocessing in SQL. Instead of manually deriving the gradient, we propose an operator for automatic differentiation. The algorithm for automatic differentiation is type-agnostic and works on datatypes that provide the needed arithmetic expressions, for example, floating point, integer values or their aggregation into arrays. Based on automatic differentiation, we will proceed with an operator for gradient descent to be able to off-load work to GPUs (see Fig. 1). This work extends a previous publication [51] by recursive SQL to train a neural network, code-generation for GPU and additional experiments. The additional contribution consists of the theoretical background of training a neural network in SQL (Sect. 5.3), the description of just-in-time (JIT) compilation to GPU (Sect. 6.3) and an extended evaluation out of training logistic regression (Fig. 19), neural networks (Figs. 20, 21, 33b) in SQL and the integration in Umbra (Sect. 7.4). The experiments involve different model functions

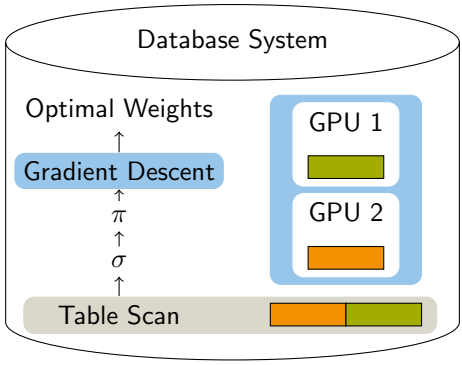


Fig. 1 In-database machine learning: gradient descent with GPU support, embedded in a query plan

and measure the performance of generated GPU kernels. In particular, this work's contributions are

- machine learning pipelines expressed in pure SQL,
- automatic differentiation in SQL that uses a lambda function [52–54] to derive the gradient and generates LLVM code,
- the integration of gradient descent as a database operator,
- fine-tuned GPU kernels that maximise the GPU specific throughput even for small and medium batch sizes,
- and an evaluation of strategies for synchronising gradient descent on processing units with different throughput.

The paper first summarises subsidiary work on machine learning pipelines, GPU co-processing and in-database machine learning (Sect. 2), before it proceeds with the integration of gradient descent inside a database system. In detail, we focus on data preprocessing for machine learning pipelines and recursive computation of gradient descent within the code generating database system Umbra (Sect. 3). During code-generation, an operator for automatic differentiation compiles the gradient from a lambda expression (Sect. 4). This allows to express arbitrary loss functions including matrix operations as used for a neural network (Sect. 5). Based on automatic differentiation and a dedicated operator for gradient descent, we compile LLVM code directly for GPUs. The generated code processes mini batches on GPUs and synchronises parallel workers on multiple devices as well as multiple learners on a single GPU (Sect. 6). We will evaluate CPU and GPU-only approaches in terms of performance and accuracy using a NUMA-server cluster with multiple GPUs (Sect. 7).

2 Related work

This work incorporates past research on deploying continuous machine learning pipelines, GPU co-processing and in-database machine learning, which is here introduced.

Machine learning pipelines To cover the life-cycle of machine learning pipelines, automatic machine learning (AutoML) tools such as Lara [55] assist in data preprocessing as well as finding the best hyper-parameters. Basically, our work ties in with the idea of continuous deployment of machine learning pipelines [30]. The idea is based on an architecture that monitors the input stream and avoids complete retraining by sampling batches.

Database systems and machine learning In the last decade, research has focused on integrating techniques of database systems into dedicated machine learning tools. One example of this kind of independent system is SystemML, with its own declarative programming language, and its successor SystemDS [56]. The integration of machine learning pipelines inside database systems would allow end-to-end machine learning [54, 57, 58] and would inherit benefits such as query optimisation and recovery by design [59]. The work of Jankov et al. [60] states that complete integration is possible by means of the extension of SQL with additional recursive statements as used in our study. As a layer above database systems that also uses SQL, LMFAO [61] learns models on pre-aggregated data.

GPU acceleration *Crossbow* [41] is a machine learning framework, written in Java, that maintains and synchronises local models for independent learners that call C++ functions to access NVIDIA's deep neural network library *cuDNN*[1]. We rely on the study when adjusting batch sizes for GPUs and synchronising multiple workers.

JIT compiling for GPU The LLVM compiler framework, often used for code generation within database engines [62–64], also offers just-in-time compilation for NVIDIA's Compute Unified Device Architecture (CUDA) [65]. Code compilation for GPU allows compilation for heterogeneous CPU-GPU clusters [66] as LLVM addresses multiple target architectures as used in this study.

3 In-database gradient descent

This section explains the mathematical background for implementing gradient descent for linear regression (Sect. 4) and neural networks (Sect. 5) in SQL or as GPU kernel (Sect. 6). Therefore, this section first introduces mini-batch gradient descent, before describing a machine learning pipeline out of preprocessing and model training expressed in SQL.

3.1 Mini-batch gradient descent

Given a set of tuples $(\mathbf{x}, y) \in X \subseteq (\mathbb{R}^m, \mathbb{R})$ with m features, a label and weights $\mathbf{w} \in \mathbb{R}^m$, then optimisation methods such as gradient descent try to find the best parameters $\mathbf{w}_\infty$ of a *model function* $m_{\mathbf{w}}(\mathbf{x})$, e.g., a linear function (Eq. 1) that approximates the given label y. A *loss function* $l_{\mathbf{x},y}(\mathbf{w})$ measures the deviation (*residual*) between

[1] https://developer.nvidia.com/cudnn.

an approximated value $m_{\mathbf{w}}(\mathbf{x})$ and the given label y, for example, mean squared error (Eq. 2):

$$m_{\mathbf{w}}(\mathbf{x}) = \sum_{i=1}^{m} x_i \cdot w_i = \mathbf{x}^T \cdot \mathbf{w} \approx y, \tag{1}$$

$$l_{\mathbf{x},y}(\mathbf{w}) = (m_{\mathbf{w}}(\mathbf{x}) - y)^2, \tag{2}$$

$$l_X(\mathbf{w}) = \frac{1}{|X|} \sum_{(\mathbf{x},y)\in X} l_{\mathbf{x},y}(\mathbf{w}) = \frac{1}{|X|} \sum_{(\mathbf{x},y)\in X} (m_{\mathbf{w}}(\mathbf{x}) - y)^2. \tag{3}$$

To minimise $l_X(\mathbf{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ until the optimal weights $\mathbf{w}_\infty$ are approximated (Eq. 6). *Stochastic gradient descent* takes one tuple for each step (Eq. 4), whereas *batch gradient descent* considers all tuples per iteration and averages the loss (Eq. 5):

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \cdot \nabla l_{\mathbf{x},y}(\mathbf{w}_t), \tag{4}$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \cdot \nabla l_X(\mathbf{w}_t) = \mathbf{w}_t - \gamma \cdot \frac{1}{|X|} \sum_{(\mathbf{x},y)\in X} \nabla l_{\mathbf{x},y}(\mathbf{w}), \tag{5}$$

$$\mathbf{w}_\infty \approx \lim_{t\to\infty} \mathbf{w}_t. \tag{6}$$

Smaller batch sizes, mini-batches, are mandatory when the entire input does not fit into memory and allows parallelism later on. Therefore, *mini-batch gradient descent* splits an input dataset X into disjoint mini-batches $X = X_0 \uplus \cdots \uplus X_o$.

Using a recursive table, gradient descent can be expressed in SQL. Listing 1 shows five iterations based on an exemplary loss function with two weights (Eq. 7): First, the weights get initialised (line 2), then each iteration updates the weights (Eq. 5, line 3) based on manually derived gradients (Eq. 8) and $\gamma = 0.05$:

$$l_{x,y}(a,b) = (a \cdot x + b - y)^2 \tag{7}$$

$$\nabla l_{x,y}(a,b) = \begin{pmatrix} \partial l/\partial a \\ \partial l/\partial b \end{pmatrix} = \begin{pmatrix} 2(ax+b-y)\cdot x \\ 2(ax+b-y) \end{pmatrix}. \tag{8}$$

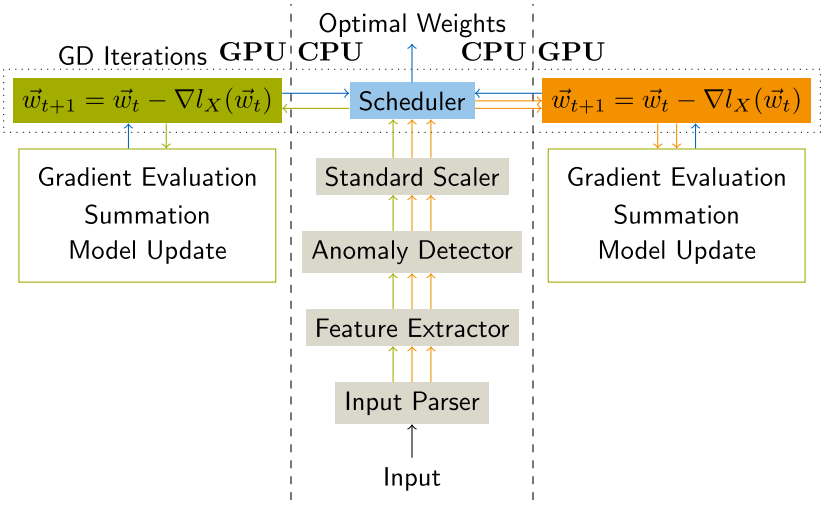


Fig. 2 Components of a machine learning pipeline: chunked input will be processed independently (potentially off-loaded to GPUs). After every iteration, the weights (blue) are synchronised

```
create table data (x float, y float); insert into data ...
with recursive gd (id, a, b) as (select 0,1::float,1::float UNION ALL
 select id+1, a-0.05*avg(2*x*(a*x+b-y)), b-0.05*avg(2*(a*x+b-y))
 from gd, data where id<5 group by id,a,b)
select * from gd order by id;
```

Listing 1: Gradient descent using a recursive table (manually derived).

To avoid overfitting, one mini-batch or an explicitly given input will serve as the validation dataset for accuracy measurements. The remaining mini-batches will be used as input for every iteration of parallel and distributed mini-batch gradient descent.

To avoid underestimation of the predicted values, Derakhshan et al. [30] propose root mean squared logarithmic error (RMSLE) on the validation dataset:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \gamma \cdot \sqrt{\frac{1}{|X|} \sum_{(\mathbf{x},y)\in X} (log((m_{\mathbf{w}}(\mathbf{x}) + 1) - log(y + 1))^2} \tag{9}$$

3.2 Machine learning pipeline in SQL

We argue that SQL offers all components needed for data preprocessing, and recursive tables allow gradient descent to be performed. Thus, we reimplemented the components of a machine learning pipeline (see Fig. 2) proposed by Derakhshan et al. [30] in SQL (see Fig. 3):

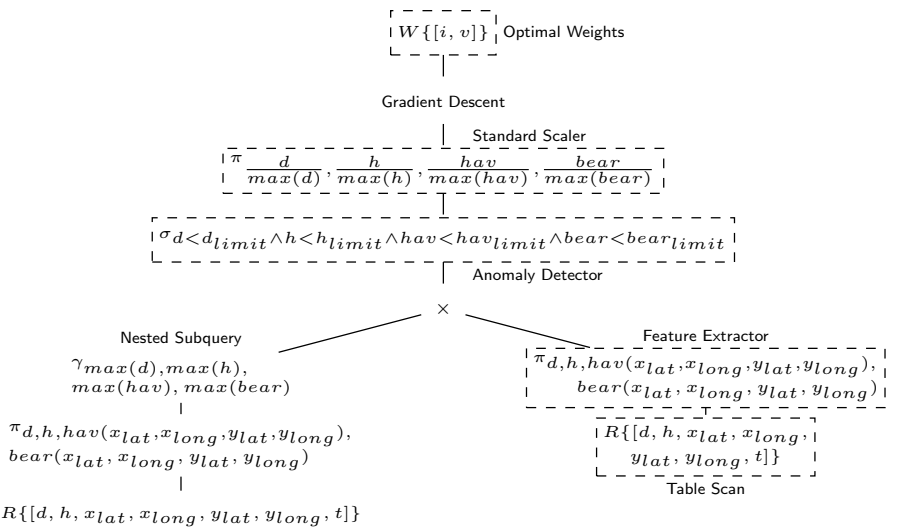


Fig. 3 Operator plan inside of a database system with linear regression on the New York taxi dataset in relational algebra: a projection extracts the features as haversine (*hav*) distance or bearing (*bear*), anomalies are deleted using predefined thresholds (denoted as *limit*)

- The **Input parser** parses input CSV files and stores the data in chunks using row-major format to allow batched processing of mini-batch gradient descent. In SQL, this corresponds to a simple table scan. In Umbra, we can also use a foreign table as input for continuous views (table `taxidata`).
- The **Feature extractor** extracts features from data chunks, which is a simple projection in SQL. For example, day and hour are extracted from timestamps, distance metrics from given coordinates (view `processed`).
- The **Anomaly detector** deletes tuples of a chunk on anomalies. An anomaly occurs when at least one attribute in a tuple passes over or under a predefined threshold. For anomalies, we filter for user-defined limits in a selection (view `normalised`).
- The **Standard scaler** scales all attributes in the range [0, 1] to equal each attribute's impact on the model, this corresponds again to a projection and nested subqueries to extract the attribute's extrema (view `normalised`).
- The **Scheduler** manages gradient descent iterations until the weights converge. This can be either done using recursive tables or using an operator that off-loads work to GPU.

Listing 2 shows the resulting SQL queries using a taxi dataset as exemplary input and a linear function to predict a trip's duration based on its day, hour, distance and bearing. In this example, we perform 50 iterations of mini-batch gradient descent based on a sample size of ten tuples (`tablesample reservoir (10)`) and a learning rate of 0.001. In every iteration, we subtract the average gradient from the weights (line `7-14`), which we finally use to compute the loss (line `15/16`). As computing each partial derivative manually can be bothersome and error-prone for

complex loss functions, we proceed with an operator for automatic differentiation in the next section.

```
1  create foreign table taxidata(id int, pickup_datetime date, dropoff_datetime date, passengers float,
       pickup_longitude float, pickup_latitude float, dropoff_longitude float, dropoff_latitude float, duration
       float) server stream;
2  copy taxidata from './taxisample.csv' delimiter ',';
3  create view processed as (select hour,day,duration,ACOS(SIN(plat)*SIN(dlat)+COS(plat)*COS(dlat)*COS(dlong-plong)
       )*6371000 distance, ATAN2(SIN(dlong-plong)*COS(dlat),COS(plat)*SIN(dlat)-SIN(plat)*COS(dlat)*COS(dlong-
       plong))*180/PI() bearing from (select avg(hour) as hour, avg(day) as day, avg(duration) as duration, avg(
       plat) as plat, avg(plong) as plong, avg(dlat) as dlat, avg(dlong) as dlong from (select cast(extractHour(
       dropoff_datetime) as float) as hour,cast(extractDay(dropoff_datetime) as day,duration, pickup_latitude
       /180*pi() plat, pickup_longitude/180*pi() plong, dropoff_latitude/180*pi() dlat, dropoff_longitude/180*pi
       () as dlong from taxidata) group by hour, day, duration, plat, plong, dlat, dlong));
4  create view normalised(hour, day, distance, bearing, duration) as (select cast(hour as float)/(select max(hour)
       +1 from processed), cast(day as float)/(select max(day) from processed), distance/(select max(distance)
       from processed where distance < 1000), (bearing+360)%360/360.0, duration/(select max(duration) from
       processed) from processed where distance < 1000);
5  with recursive gd (id, a1, a2, a3, a4, b) as (
6  select 1, 1::float, 1::float, 1::float, 1::float, 1::float UNION ALL
7  select id+1,
8    a1-0.001*avg(2*hour*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
9    a2-0.001*avg(2*day*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
10   a3-0.001*avg(2*distance*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
11   a4-0.001*avg(2*bearing*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
12   b -0.001*avg(2*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration))
13 from gd, (select * from normalised tablesample reservoir (10))
14 where id<=50 group by id,a1,a2,a3,a4, b)
15 select id, avg(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)^2
16 from gd,normalised where id=51;
```

Listing 2: Machine learning pipeline in SQL.

4 Database operators for machine learning

This section describes the operators in Umbra, we created to facilitate machine learning in SQL. Modern database systems like Umbra generate code for processing chunks of tuples in parallel pipelines, so we first explain code generation before presenting the operators for automatic differentiation and gradient descent. The algorithm for automatic differentiation presents how to parse an expression to calculate the derivatives and is expanded for matrix operations in Sect. 5.

4.1 Code generation

With Umbra as the integration platform, an operator follows the concept of a code-generating database system. It achieves parallelism by starting as many pipelines as threads available and expects each operator in a query plan to generate code for processing chunks of tuples. Unary operators can process tuples within a pipeline, whereas binary operators have to materialise at least the result of one incoming child node first before pipelined processing begins.

Each operator of Umbra, similar to HyPer [18], provides two functions, `produce()` and `consume()` to generate code. On the topmost operator of an operator tree, `produce()` is called, which recursively calls the same method on its child operators. Arriving at a leaf operator, it registers pipelines for parallel execution and calls `consume()` on the parent node. Within these pipelines, the generated code processes data inside registers without overhead. An operator

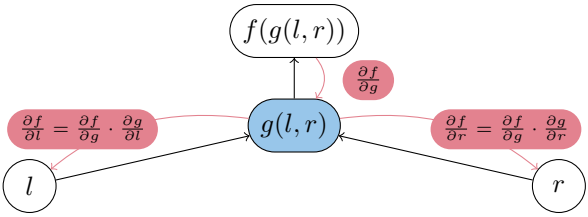


Fig. 4 Reverse mode automatic differentiation: First, the function $f(g(l, r))$ gets evaluated, then each partial derivative is computed in reverse order. Each arrow represents one cached computation

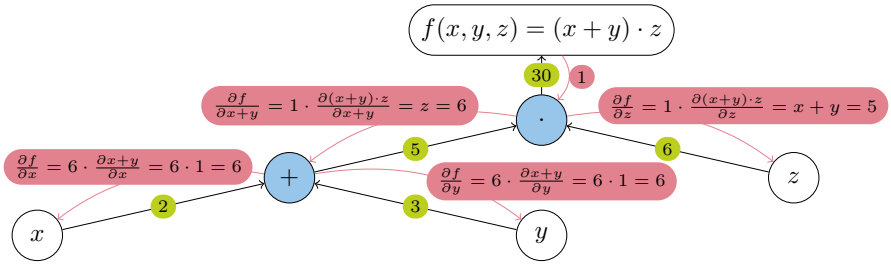


Fig. 5 Automatic differentiation $f(x, y, z) = (x + y) \cdot z$ on $x = 2, y = 3, z = 6$

for gradient descent is a pipeline breaker, as it accesses batches of tuples multiple times until the weights converge, whereas an operator for automatic differentiation is part of a pipeline as it just adds the partial derivatives per tuple.

4.2 Automatic differentiation

Reverse mode automatic differentiation first evaluates an expression, then it propagates back the partial derivative in reverse order by applying the chain rule (see Fig. 4). Each partial derivative is the product of the parent one (or 1 for the topmost node) and the derived function with its original arguments as input (see Fig. 5). This allows computing each expression's derivative in one pass by reusing each subexpression. Algorithm 1 parses an arithmetic expression to perform reverse mode automatic differentiation. It uses the evaluation of the original arguments to calculate the partial derivatives. Arriving at a variable, the propagated value is added to a hashtable with the variable as key. The hashtable allows retrieving the derivatives, which is the sum of propagated values per variable, afterwards.

As Umbra compiles arithmetic expressions to machine code as well, it is perfectly suited for automatic differentiation. Similar to how an arithmetic SQL expression is compiled during code generation, we created a function that can be used to generate the expression's partial derivatives: Once a partial derivative has been compiled, its subexpressions will be cached inside an LLVM register that can be reused to generate the remaining partial derivatives. This accelerates the runtime during execution.

Algorithm 1 Automatic Differentiation

```
1:  function DERIVE(z, z')
2:      if isBinary(z) then
3:          if z = x + y then DERIVE(x,z') DERIVE(y,z')
4:          else if z = x − y then DERIVE(x,z') DERIVE(y,−z')
5:          else if z = x · y then DERIVE(x,z' · y) DERIVE(y,z' · x)
6:          else if z = x/y then DERIVE(x, z'/y) DERIVE(y, (−z'·x)/y^2)
7:          else if z = x^y then DERIVE(x,z' · y · x^(y−1)) DERIVE(y,z' · x^y ln(x))
8:          else if z = log_y(x) then DERIVE(x, z'/(x·ln(y))) DERIVE(y, (−z'·ln(x))/(y·ln^2(y)))
9:          end if
10:     else if isUnary(z) then
11:         if z = √x then DERIVE(x, z'/(2·√x))
12:         else if z = |x| then DERIVE(x, (z'·x)/|x|)
13:         else if z = sin(x) then DERIVE(x,z' · cos(x))
14:         else if z = cos(x) then DERIVE(x,−z' · sin(x))
15:         else if z = e^x then DERIVE(x,z' · e^x)
16:         end if
17:     else if isVariable(z) then
18:         ∂/∂z ← ∂/∂z + z'
19:     end if
20: end function
```

To specify the expression, we integrated lambda functions as introduced in HyPer into Umbra. Lambda functions are used to inject user-defined SQL expressions into table operators. Originally developed to parametrise distance metrics in clustering algorithms or edges for the PageRank algorithm, lambda functions are expressed inside SQL queries and allow “variation points" [67] in otherwise inflexible operators. In this way, lambda expressions broaden the application of the default algorithms without the need to modify the database system’s core. Furthermore, SQL with lambda functions substitutes any new query language, offers the flexibility and variety of algorithms needed by data scientists, and ensures usability for non-expert database users. In addition, lambda functions allow user-friendly function specification, as the database system automatically deduces the lambda expressions’ input and output data types from the previously defined table’s attributes. Lambda functions consist of arguments to define names for the tuples (but whose scope is operator specific) and the expression itself. All provided operations on SQL types, even on arrays, are allowed:

$$\lambda(< \text{name1} >, < \text{name2} >, ...)(< \text{SQL expression} >).$$

```
select * from umbra.derivation(TABLE(select 2::float x,3::float y,6::float z),lambda(x)((x.x+x.y)*x.z));
-- x y z d_x d_y d_z
-- 2 3 6 6 6 5
select * from umbra.derivation(TABLE(select 2::float x, 3::float y, 10::float a, 10::float b), lambda (x)((x.a *
       x.x + x.b - x.y)^2));
-- x y a b d_x d_y d_a d_b
-- 2 3 10 10 540 -54 108 54
```

Listing 3: Automatic differentiation within SQL.

We expose automatic differentiation as a unary table operator called `derivation` that derives an SQL expression with respect to every affected column reference and adds its value as a further column to the tuple (see Listing 3). We can use the operator within a recursive table to perform gradient descent (see Listing 4, 5). This eliminates the need to derive complex functions manually and accelerates the computation with a rising number of attributes, as each subexpression is evaluated only once.

```
with recursive gd (id, a, b) as (select 1,1::float,1::float UNION ALL
 select id+1, a-0.05*avg(d_a), b-0.05*avg(d_b)
 from umbra.derivation(TABLE (select id,a,b,x,y from gd,data where id<5),
 lambda (x) ((x.a * x.x + x.b - x.y)^2))  group by id,a,b)
select * from gd order by id;
```

Listing 4: Gradient descent using a recursive table (automatically derived).

```
with recursive gd (id, a1, a2, a3, a4, b) as (
 select 1, 1::float, 1::float, 1::float, 1::float, 1::float
UNION ALL
 select id+1,a1-0.001*avg(d_a1),a2-0.001*avg(d_a2),a3-0.001*avg(d_a3),a4-0.001*avg(d_a4),
        b-0.001*avg(d_b)
 from umbra.derivation(TABLE (
  select * from gd, (select * from normalised tablesample reservoir (10)) where id < 51),
  lambda (x)((x.a1*x.hour + x.a2*x.day + x.a3*x.distance + x.a4*x.bearing + x.b-x.duration)^2))
 group by id, a1, a2, a3, a4, b)
select id, avg(a1*hour + a2*day + a3*distance + a4*bearing + b-duration)^2
from gd, normalised where id = 51;
```

Listing 5: Automatic differentiation for gradient descent on the taxi dataset.

4.3 Gradient descent operator

Our operator for gradient descent materialises incoming tuples, performs gradient descent and produces the optimal weights for labelling unlabelled data. The proposed operator is considered a pipeline breaker as it needs to materialise all input tuples beforehand to perform multiple training iterations. This section focuses on the operator characteristics, the design with its input queries and the actual implementation, with linear regression as an example.

4.3.1 Operator design

We design an operator for gradient descent, which requires one input for the training, one for the initial weights and optionally one for the validation set, and returns the optimal weights. If no input is given as validation set, a fraction of the training

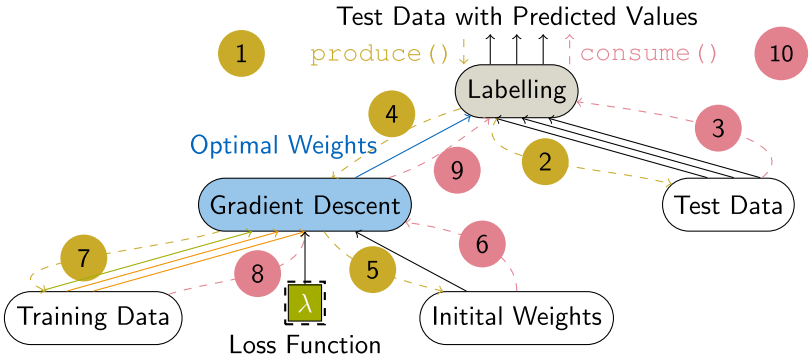


Fig. 6 Operator plan inside of a database system with one operator for training and a query for predicting labels. Dashed lines illustrate code generation, solid lines compiled code. The `gradient descent` operator materialises input from parallel pipelines within local threads, performs iterations and returns the optimal weights

set will be used for validation. The user can set the parameters for the batch size, the number of iterations and the learning rate as arguments inside the operator call (see Listing 6). Figure 6 shows gradient descent inside of an operator tree: it expects the training data set as parallel input pipelines and returns the optimal weights. These might serve as input for a query that labels a test data set. In addition, SQL lambda functions, which allow users to inject arbitrary code into operators, specify the loss function to be used for gradient descent. Gradient descent benefits from generated code as it allows user-defined model functions to be derived at compile time to compute its gradient without impairing query runtime.

```
1 select * from umbra.gd(TABLE (select * from data), TABLE (select 10::float a, 10::float b), lambda (x,y) ((y.a *
        x.x + y.b - x.y)^2), 1, 0.05, 10);
```

Listing 6: Gradient descent as operator.

This implies three parts for the integration of gradient descent: consuming all input tuples in parallel pipelines, performing gradient descent with a call to the GPU kernels and producing the weights in a new pipeline. This first separation is necessary, as we need to know the number of tuples in advance to determine when one training epoch ends. Specific to Umbra, we cannot assume the same number of available threads for training as for the parallel pipelines; we have to merge all materialised tuples before we start new parallel threads for the training iterations afterwards.

4.3.2 Implementation

The generated code runs gradient descent iterations in parallel. Devoted to batched processing on GPUs, we deduce a parallel mini-batch gradient descent operator. First, it materialises the input tuples thread locally (generated by `consume()`) and

merges them globally. Afterwards, each thread picks one mini-batch and maintains a local copy of the global weights.

Algorithm 2 depicts the training procedure without GPU support. Again, for simplicity, the validation phase with the corresponding validation input is omitted. Inside of the two loops (lines 5–9), one is unrolled during compile time in order to dispatch tasks to parallel threads, and one executed at runtime to manage gradient descent iterations, we can later off-load work to GPUs. Inside such a code fragment, we start as many CPU threads as GPU units are available with whom one CPU thread is associated.

Algorithm 2 Operator for mini-batch gradient descent.

```
1:  function PRODUCE
2:      COMPUTEGRADIENT(expression)
3:      PRODUCE(inputPipeline)
4:      GENERATE(mergeTuples)
5:      GENERATE(while !converged)
6:      for l ∈localthreads do
7:          GENERATE(l.updateWeights)
8:      end for
9:      GENERATE(w⃗ ← Σ_{l∈localthreads} l.w⃗ / |localthreads|)
10:     GENERATE(whileEnd)
11:     CONSUME(parent,w⃗)
12: end function
13: function UPDATEWEIGHTS
14:     GENERATE(w⃗ ← w⃗ − 1/|X| Σ_{(x⃗,y)∈X} ∇l_{x⃗,y}(w⃗))
15: end function
16: function CONSUME
17:     GENERATE(localthread.store(x⃗, y))
18: end function
```

5 Neural network

As the previous sections presented gradient descent and automatic differentiation on scalar values, this section expands their use to matrix operations for training a neural network. Mini-batch gradient descent can be used to train a neural network with an adjusted model function. We assume every input with m attributes serialised as one input vector $x \in \mathbb{R}^{1\times m}$ together with a categorical label $y \in L$. This corresponds to a database tuple as we later store one image as one tuple with one attribute per pixel and colour.

We consider a fully connected neural network with one hidden layer of size h, consequently we gain two weight matrices $w_{xh} \in \mathbb{R}^{m\times h}$ and $w_{ho} \in \mathbb{R}^{h\times|L|}$. With sigmoid (Eq. 10) as activation function (applied elementwise) we obtain a model

function $m_{w_{xh},w_{ho}}(x) \in \mathbb{R}^{1\times|L|}$, that produces an output vector of probabilities, also known as forward pass. The output vector is compared to the one-hot-encoded categorical label (y_{ones}). The index of the entry being one should be equal to the index of the highest probability.

$$sig(x) = (1 + e^{-x})^{-1} \tag{10}$$

$$m_{w_{xh},w_{ho}}(x) = \underbrace{sig(\overbrace{sig(x \cdot w_{xh})}^{a_{xh}} \cdot w_{ho})}_{a_{ho}}. \tag{11}$$

Although neural networks can be specified in SQL-92, the corresponding query will consist of nested subqueries, that are not intuitive to create. As we created an array data type in Umbra [54], nested subqueries can be avoided by using this data type extended by matrix algebra.

5.1 Neural network in SQL-92

Expressing neural networks in SQL-92 is possible having one relation for the weights and one for the input tuples (Listing 7). The weights relation will contain the values in normal form as a coordinate list. If one relation contains all weight matrices, it will also contain one attribute (`id`) to identify the matrix.

We implement the forward pass in SQL as a query on the validation dataset that returns the probabilities for each category. It uses nested queries to extract the weights by an index and arithmetic expressions for the activation function. Listing 7 shows the forward pass with one single layer and two attributes ($m = 2$) as input. For simplicity, we use SQL functions for nested subqueries and for the sigmoid function.

```
1 create table data (a float, b float); insert into data ...
2 create table w(id int, i int, j int, val float); insert into w ...
3 -- helper functions
4 create function w_ij(id int, i int, j int) returns float language 'sql' strict as $$
5   select val from w where w.i=i and w.j=j and w.id=id $$;
6 create function sig(i float) returns float language 'sql' as $$ select 1.0/(1.0+exp(-i)); $$;
7 -- forward pass
8 select sig(i.a*w_ij(0,1,1)+i.b*w_ij(0,2,1)),sig(i.a*w_ij(0,1,2)+i.b*w_ij(0,2,2)) from data i;
```

Listing 7: Forward pass for one layer within a neural network in SQL-92.

5.2 Neural network with an array data type

Expressing matrix operations in SQL-92 has the downside of manually specifying each elementwise multiplication. For this reason, Umbra and HyPer provide an array data type that is similar to the one in PostgreSQL and allows algebraic expressions as matrix operations.

Table 1 Implemented activation functions and their derivatives

Name	Abbreviation	$f(x)$	$f'(x)$
Hyperbolic tangent	tanh	$\frac{e^x-e^{-x}}{e^x+e^{-x}}$	$1-f(x)^2$
Rectified linear unit	relu	$\begin{cases} x & \text{if } x>0 \\ 0 & \text{if } x\geq 0 \end{cases}$	$\begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x\geq 0 \end{cases}$
Logistic sigmoid	sig	$\frac{1}{1+e^{-x}}$	$f(x)(1-f(x))$

```
1  create table wm(id int, val float[][]);
2  insert into wm select id, array_agg(name) from (select id, i, array_agg(val) as name from w group by id,i) j
       group by id;
3  -- helper function
4  create function sig (x float[]) returns float[] language 'sql' as $$
5    select array_agg(s) from (select sig(unnest) as s from unnest(x)) tmp; $$;
6  -- forward pass
7  select sig(array[[a,b]]*wm.val) from data, wm where wm.id=0;
```

Listing 8: Forward pass for one layer within a neural network with an array data type.

In Listing 8, we first construct the weight matrices from its relational representation and apply the sigmoid function on arrays as a user-defined function. Hence, the forward-pass for a single layer consists of the matrix multiplication and the sigmoid function on arrays.

An array data type allows transforming a categorical data type L into a one-hot-encoded vector even if the number of categories $|L|$ is not known at compile time. In Listing 9, we first create a dictionary to assign a consistent number to each category (line 3). Afterwards, we can create an array that corresponds to a one-hot-encoded vector, whose entry is one at the corresponding index with leading and subsequent zeros (line 4).

```
1  create table categories(a text); insert into categories values ('Apple'),('Egg'),('Apple');
2  with dict as (
3   select a, cast(rank() over (order by a) as int) from (select distinct a from categories) sub)
4  select categories.a, rank, array_fill(0,array[rank-1])    1    array_fill(0,array[(select cast(count(distinct a)
         as int) from categories) - rank])
5  from categories, dict where categories.a=dict.a;
```

Listing 9: One-hot-encoding using the SQL array data type.

5.3 Training a neural network

Automatic differentiation allows training a neural network when the derivatives of matrix multiplication [68] (see Algorithm 3) and activation functions (see Table 1) are implemented. To integrate the derivatives of matrix multiplication in our existing implementation, we need to extend the derivation rule for multiplications ($Z' \cdot Y^T$ instead of $z' \cdot y$ and $Z'^T \cdot X$ instead of $z' \cdot x$) and overload the transpose operator internally, so that transpose, when called on a scalar such as real numbers like floating-point values, will be ignored. Furthermore, we need the Hadamard product

$X \circ Y$ (elementwise matrix multiplication) for the derivatives of the activation functions and the Hadamard power function $X^{\circ Y}$ (elementwise exponentiation) for mean squared error.

Algorithm 3 Automatic Differentiation (Matrices)

1: **function** DERIVE(Z, Z')
2: **if** $Z = X + Y$ **then** DERIVE(X,Z') DERIVE(Y,Z')
3: **else if** $Z = X \circ Y$ **then** DERIVE(X,$Z' \circ Y$) DERIVE(Y,$Z' \circ X$)
4: **else if** $Z = X \cdot Y$ **then** DERIVE(X,$Z' \cdot Y^T$) DERIVE(Y,$Z'^T \cdot X$)
5: **else if** $Z = f(X)$ **then** DERIVE(X,$Z' \circ f'(X)$)
6: **else** $\frac{\partial}{\partial Z} \leftarrow \frac{\partial}{\partial Z} + Z'$
7: **end if**
8: **end function**

These adaptations allow our operator for automatic differentiation to train the weights for a neural network ($m = 4, h = 20, |L| = 3$) within a recursive table (see Listing 10): We first initialise the weight matrices with random values (line 2) and then update the weights in each iterative step (line 4) using mean squared error $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$ (line 7).

```
1 with recursive gd (id,w_xh,w_ho) as (
2   select 0, (select array_agg(array_agg) from generate_series(1,4),(select array_agg(random()) from
          generate_series(1,20))), (select array_agg(array_agg) from generate_series(1,20),(select array_agg(
          random()) from generate_series(1,3)))
3 union all
4  select id+1, w_xh - 0.2 * avg(d_w_xh), w_ho - 0.2 * avg(d_w_ho)
5  from umbra.derivation(
6    TABLE(select * from (select * from data tablesample reservoir(50)),gd where id < 2000),
7    lambda(x)( (sig(sig(x.img**x.w_xh)*x.w_ho)-one_hot)^2 ))
8  group by id, w_ho, w_xh
9 ) select * from gd where id = 2000;
```

Listing 10: Training a neural network using automatic differentiation within a recursive table.

Instead of relying on an operator for automatic differentiation, we can train a neural network by hand when applying the rules for automatic differentiation. With mean squared error, the loss is equal to the difference of labels and predicted probabilities (Eq. 12). The factor 2 can be omitted when the learning rate γ is doubled. To train the neural network for a given input vector, we have to backpropagate the loss and update the weights as follows:

$$l_{ho} = 2 \cdot (m_{w_{xh},w_{ho}}(x) - y_{ones}), \tag{12}$$

$$\delta_{ho} = l_{ho} \circ sig'(a_{ho}) = l_{ho} \circ a_{ho} \circ (1 - a_{ho}), \tag{13}$$

$$l_{xh} = l_{ho} \cdot w_{ho}^T, \tag{14}$$

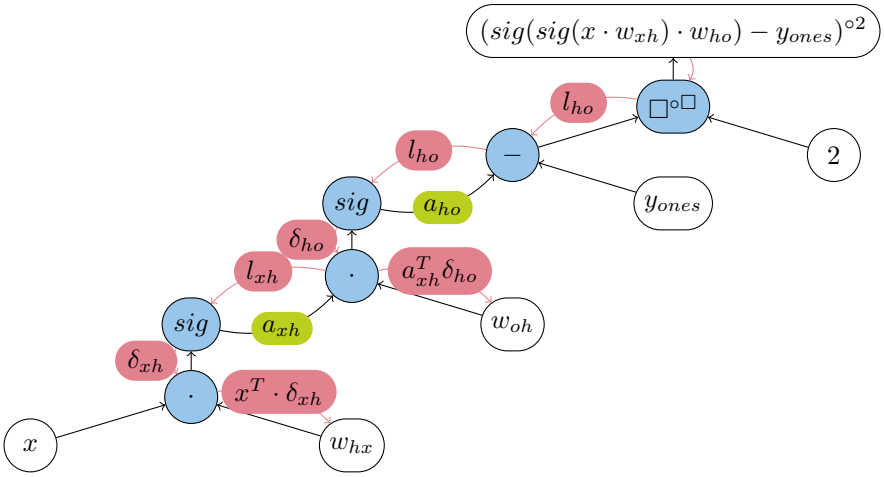


Fig. 7 Automatic differentiation for $(m_{w_{xh},w_{ho}}(x) - y_{ones})^{\circ 2}$

$$\delta_{xh} = l_{xh} \circ sig'(a_{xh}) = l_{xh} \circ a_{xh} \circ (1 - a_{xh}), \tag{15}$$

$$w'_{ho} = w_{ho} - \gamma \cdot a_{xh}^T \cdot \delta_{ho}, \tag{16}$$

$$w'_{xh} = w_{xh} - \gamma \cdot x^T \cdot \delta_{xh}. \tag{17}$$

Figure 7 shows the corresponding computational graph and Listing 11 the corresponding code in SQL when the Hadamard product ($\circ$) is exposed as SQL operation ($**$).

```
1  with recursive gd (id,w_xh,w_ho) as (
2    select 0, (select array_agg(array_agg) from generate_series(1,4),(select array_agg(random()) from
         generate_series(1,20))), (select array_agg(array_agg) from generate_series(1,20),(select array_agg(
         random()) from generate_series(1,3)))
3  union all
4    select id+1, w_xh - 0.2 * avg(transpose(img)*d_xh), w_ho - 0.2 * avg(transpose(a_xh)*d_ho)
5    from ( select l_xh ** (a_xh ** (1- a_xh)) as d_xh, *
6           from ( select d_ho*transpose(w_ho) as l_xh, *
7                  from (
8                   select l_ho ** (a_ho ** (1- a_ho)) as d_ho, *
9                   from ( select 2*(a_ho-one_hot) as l_ho, *
10                         from (
11                          select sig(a_xh*w_ho) as a_ho, *
12                          from ( select sig(img*w_xh) as a_xh, *
13                                 from (select *
14                                       from data tablesample reservoir(50)),gd
15                                       where id < 2000))))))
16   group by id, w_ho, w_xh
17 ) select * from gd where id = 2000:
```

Listing 11: Backpropagation for a neural network using a recursive table.

To process a batch of input tuples instead of a single one, the multiplications are applied on matrices instead of vectors. The multiplication during the last steps, the weight updates (Eqs. 16, 17), then sums up the delta for every tuple, so a simple

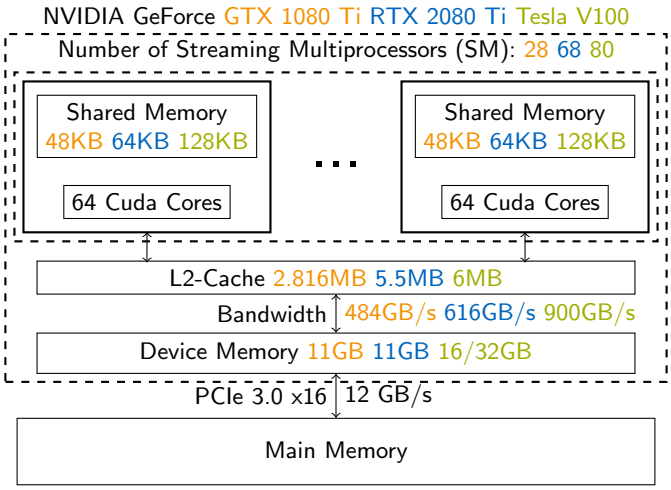


Fig. 8 Simplified GPU architecture for NVIDIA GeForce GTX 1080 Ti (orange), RTX 2080 Ti (blue) and Tesla V100 (green): Each GPU transfers data via PCIe x16 from main-memory to its global/device memory. Multiple CUDA cores sharing the L1 cache (shared memory) are grouped to one streaming multiprocessor, its number is GPU specific. In-between lies the L2 cache

division by the number of tuples is needed to construct the average gradient. This is helpful when vectorising mini-batch gradient descent in Sect. 6.1.2.

6 Multi-GPU gradient descent

This section explains our CUDA kernels for linear regression and neural networks, which one CPU worker thread starts once per GPU. We describe blocking and non-blocking algorithms so as not not to hinder faster GPUs from continuing their computation while waiting for the slower ones to finish. To synchronise multiple workers, we either average the gradient after each iteration or maintain local models as proposed by Crossbow [69]. We adapt this synchronisation technique to maximise the throughput of a single GPU as well. As a novelty, we implement learners at hardware level—each associated to one CUDA block—to maximise the throughput on a single GPU. Finally, we generate the kernels directly with LLVM to support lambda functions for model specification.

6.1 Kernel implementations

Developing code for NVIDIA GPUs requires another programming paradigm, as computation is vectorised to parallel threads that perform the same instructions simultaneously. Each GPU device owns one global memory (device memory) and an L2 cache. Core components are streaming multiprocessors with an attached shared memory (see Fig. 8) to execute specialised programs for CUDA devices (*kernels*).

In these kernels, every thread receives a unique identifier, which is used to determine the data to process. 32 threads in a bundle are called a *warp*, multiple warps form a *block* and threads inside a block can communicate through shared memory and interfere with each other. To interfere with other threads, shuffling can be used to share or broadcast values with one or more threads within a warp.

To off-load gradient descent iterations to NVIDIA GPUs, we generate specialised kernels. In detail, we have to map batches to blocks; we can vary the number of threads per block and explicitly cache values in shared memory. In the following sections, we describe our kernel implementations for gradient descent with linear regression and a fully-connected neural network, and we will introduce independent learners at block-size granularity.

6.1.1 Linear regression

As linear regression is not a compute-bound but a memory-intensive application, we initially transfer as much training data into device memory as possible. If data exceeds the memory and more GPUs are available for training, we will partition the data proportionally to multiple devices. Otherwise, we reload the mini-batches on demand.

Each thread handles one input tuple and stores the resulting gradient after each iteration in shared memory. Each iteration utilises all available GPU threads, wherefore the size of a mini-batch must be greater or equal to the number of threads per block, to ensure that compute resources are not wasted. When the batch size is larger than a block, each thread processes multiple tuples and maintains a thread-local intermediate result, which does not require any synchronisation with other threads. After a mini-batch is processed, shuffling operations summarise the gradient to compute the average for updating the weights (tree reduction).

6.1.2 Neural network

Our initial approach was to adapt the gradient descent kernel for linear regression to train a neural network and to spread each tuple of a batch to one thread. As training neural networks is based on matrix operations, we rely on libraries for basic linear algebra subprograms for CUDA devices (cuBLAS[2]), which provide highly optimised implementations. Our implementation uses the cuBLAS API for all operations on matrices or vectors. For example, the forward pass in a neural network uses matrix-vector multiplications (`cublasDger()`) for a single input tuple and, when a mini-batch is processed, matrix-matrix multiplications respectively (`cublasDgemm()`). To apply and derive the activation function, handwritten kernels are used that vectorise over the number of attributes. These kernels plus the library calls plus handwritten code build the foundation for parallelising to multiple GPUs.

[2] https://docs.nvidia.com/cuda/cublas

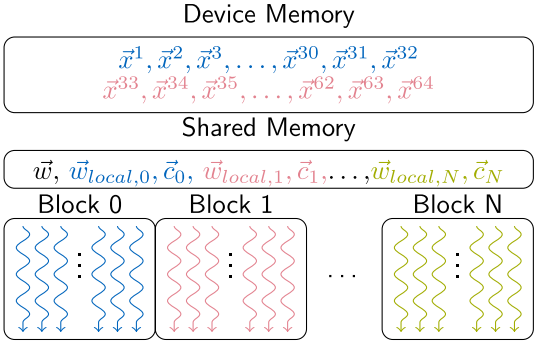


Fig. 9 Multiple learners per GPU: Each block corresponds to one learner, each learner maintains local weights $\mathbf{w}_{local}$ and the difference $\mathbf{c}_{local}$ to the global weights $\mathbf{w}$. Each input tuple is stored in device memory and is scheduled to one GPU thread

6.1.3 Multiple learners per GPU

To utilise all GPU threads even with small batch sizes, we implement multiple workers on a single GPU. These are called *learners* [69] and ensure a higher throughput. Crossbow offers a coarse-grained approach as every learner launches multiple kernels, which limits its overall number. By contrast, our light-weight approach launches only one instance of a fine-grained kernel for one entire GPU. This enables full utilisation of the GPU as the number of learners could be much higher.

In our implementation (see Fig. 9), each learner corresponds to one GPU block. We can set the block size adaptively, by which the number of learners results. Consequently, one learner works on batch sizes of at least one warp, that is the minimum block size with 32 threads, or multiple warps. Hence, the most learners that are allowed is the number of warps that can be processed per GPU.

After each learner has finished its assigned batches, the first block synchronises with the other ones to update the global weights. But for multi GPU processing as well as for multiple learners per GPU, we need to synchronise each unit.

6.2 Synchronisation methods

As we intend to run gradient descent in parallel on heterogeneous hardware, we have to synchronise parallel gradient descent iterations. Based on a single-threaded naive gradient descent implementation, we propose novel synchronisation methods to compare their performance to existing ones and benchmark different hardware.

The naive implementation uses a constant fraction of its input data for validation and the rest for training. The training dataset is split into fixed-sized mini-batches. After a specified number of mini-batches but no later than after one epoch when the whole training set has been processed once, the loss function is evaluated on the validation set and the current weights. The minimal loss is updated and the next epoch starts. We terminate when the loss falls below a threshold l_{stop} or a maximum number of processed batches ctr_{max}. Also, we terminate if the loss has not changed within the last 10 iterations.

Based on the naive implementation, this section presents three parallelisation techniques, a blocking but synchronised one and two using worker threads with multiple local models or only one global one.

6.2.1 Synchronised iterations

At the beginning of each synchronised iteration, we propagate the same weights with an individual mini-batch to the processing unit. After running gradient descent, the main worker collects the calculated gradients and takes their average to update the weights.

Algorithm 4 shows this gradient descent function, taking as input the labelled dataset X, a learning rate γ, the batch size n and the parameter ctr_{max} that limits the number of iterations. In each iteration, multiple parallel workers pick a mini-batch and return the locally computed gradient. Afterwards, the weights are updated. For simplicity, the validation pass is not displayed: When the calculated loss has improved, the minimal weights together with the minimal loss are set and terminate the computation when a minimal loss l_{min} has been reached.

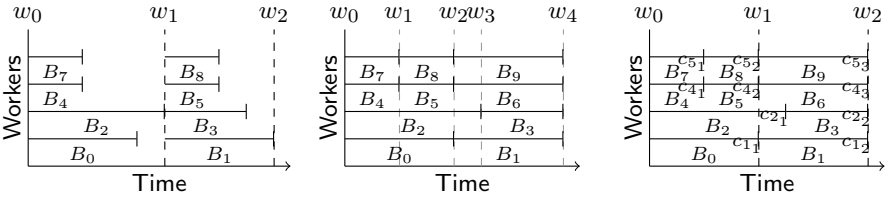
Algorithm 4 Synchronised.

```
1:  function GD(X,γ, ctr_max, n)
2:      w⃗ ← (0, ..., 0)
3:      ctr ← 0
4:      while ctr < ctr_max do
5:          for t ∈ [#workers] do
6:              batch ← ATOMIC_FETCH_ADD(ctr,1)        } in parallel
7:              X′ ← GETBATCH(batch,X,n)               } in parallel
8:              g⃗_t ← γ∇l_{X′}(w⃗)                     } in parallel
9:          end for
10:         for t ∈ [#workers] do
11:             w⃗ ← w⃗ − γg⃗_t
12:         end for
13:     end while
14: end function
```

When synchronising a global model after each iteration, workers who may have finished their mini-batches earlier, are idle and waiting for input (see Fig. 10a). To maximise the throughput, independent workers have to fetch their mini-batches on their own. These independent workers either require local weights to be synchronised frequently (see Fig. 10c) or update global weights centrally (see Fig. 10b).

6.2.2 Worker threads with global updates (bulk synchronous parallel)

In Algorithm 5, we see worker threads that fetch the next batch independently and update a global model. Each worker increments a global atomic counter as a batch



(a) Synchronised threads **(b)** Global updates **(c)** Local models

Fig. 10 Scheduling mini-batches on four different workers: **a** shows worker threads whose weights are synchronised globally after each iteration and whose averaged gradient is used to update the global weights *w*; workers are idle when others have still not finished. **b** shows worker threads that update weights globally without any synchronisation; each worker is responsible for fetching the next batch on its own. To overcome race conditions, the worker threads in (**c**) maintain their local model that is synchronised lazily when every worker is done with at least one iteration

identifier and selects the corresponding batch consisting of the attributes and the labels. The current weights are used to compute the gradient; afterwards, the weights are updated globally. Besides, the first thread is responsible for managing the minimal weights. Assuming a low learning rate, we suppose the weights are changing marginally and we omit locks similar to HogWild [70]. Otherwise, the critical section (line 5)—gradient calculation and weights update—has to be locked, which would result in a single-threaded process as in Algorithm 4.

Algorithm 5 Worker threads (global updates).

```
1:  function RUN(X, w⃗, ctr, ctr_max, γ)
2:      while ctr < ctr_max do
3:          batch ← ATOMIC_FETCH_ADD(ctr,1)
4:          X′ ← GETBATCH(batch,X,n)
5:          w⃗ ← w⃗ − γ∇l_X′(w⃗)            } critical section
6:      end while
7:  end function
8:  function GD(X,γ, ctr_max, n)
9:      w⃗ ← (0, ..., 0)
10:     ctr ← 0
11:     for t ∈ [#workers] do
12:         RUN(X, w⃗, ctr, ctr_max, γ)
13:     end for                          } in parallel
14: end function
```

6.2.3 Worker threads with local models (model average)

To overcome race conditions when updating the weights, we adapt local models known from Crossbow [69] to work with worker threads. Crossbow adjusts the number of parallel so-called learners adaptively to fully utilise the throughput on

different GPUs. Each learner maintains its local weights and the difference from the global weights. A global vector variable for every learner t called corrections $\mathbf{c}_t$ stores this difference, divided by the number of all learners. In each iteration, the weights are updated locally and these corrections are subtracted. After each iteration, the corrections of all learners are summed up to form the global weights.

Algorithm 6 shows its adaption for worker threads. The main thread manages the update of the global model (line 11) that is the summation of all corrections. The critical section now consists of the computation of the corrections (lines 7-9) only, so the gradient can be computed on multiple units in parallel.

Algorithm 6 Worker threads (local models).

```
1:  function RUN(X, w⃗, ctr, ctr_max, γ)
2:      w⃗_local ← w⃗
3:      while ctr < ctr_max do
4:          batch ← ATOMIC_FETCH_ADD(ctr,1)
5:          X′ ← GETBATCH(batch,X,n)
6:          g⃗ ← γ∇l_X′(w⃗_local)
7:          c⃗_threadid ← (w⃗_local − w⃗)/t
8:          w⃗_local ← w⃗_local − γg⃗ − c⃗_threadid
9:          c⃗_threadid ← (w⃗_local − w⃗)/t
10:         if threadid = 0 then                      } critical section
11:             w⃗ ← w⃗ + Σ_{t∈[#workers]} c⃗_t
12:         end if
13:     end while
14: end function
15: function GD(X,γ, ctr_max, n)
16:     w⃗ ← (0, ..., 0)
17:     ctr ← 0
18:     for t ∈ [#workers] do
19:         RUN(X, w⃗, ctr, ctr_max, γ)
20:     end for                                       } in parallel
21: end function
```

6.3 JIT compiling to GPU

The normal way to use the CUDA interface is to write the CUDA code, which is C++ with additional language elements to support kernel declarations. The compiled CUDA code can be invoked from the host as a special function invocation through the CUDA API. With a just-in-time architecture, which compiles the GPU code, one can keep the advantages of modularisation but also allow for more optimisations to take place during compile time. Similar to gradient computation on the CPU, the lambda function can be passed directly to customised model-specific

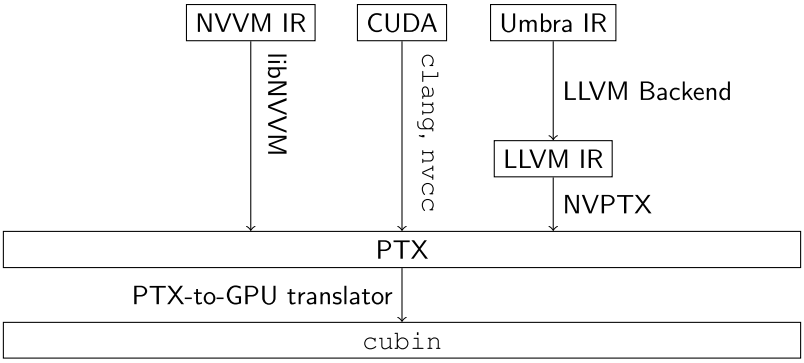


Fig. 11 Different existing paths of code generation for NVIDIA GPUs and how the translation from one layer to the next happens. `cubin` is the binary format loadable by the NVIDIA driver

kernels as it generates the gradient of a user-defined model function during compile time without impairing query time.

6.3.1 LLVM NVPTX interface

LLVM [71] is a framework that allows for just-in-time compilation via an intermediate representation in Static Single Assignment (SSA) form also called LLVM IR. There are two ways of using LLVM for generating code for GPUs (see Fig. 11). One way is to use the API that NVIDIA itself provides called *libNVVM*.[3] *libNVVM* takes NVVM IR,[4] which is based on LLVM IR, and compiles it to PTX,[5], an assembly language for NVIDIA GPUs, which can subsequently be run on the GPU. A disadvantage of this approach to using LLVM with CUDA is that the LLVM version NVVM IR is based on is older than the official version. The current NVVM IR is based on LLVM 7.0.1, while the latest version as of now is 12.0.1.[6]

The alternative is using LLVM directly for generating LLVM IR. Using the NVPTX[7] backend, we can generate PTX, which can be run on the GPU. This is the approach we can take to reuse the Umbra IR to LLVM IR backend to have a high-level API for our code generation.

[3] https://docs.nvidia.com/cuda/libnvvm-api/index.html.
[4] https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.
[5] https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.
[6] https://lists.llvm.org/pipermail/llvm-announce/2021-July/000093.html.
[7] https://llvm.org/docs/NVPTXUsage.html.

```
1 extern __shared__
2   double test_shared[];
```

Listing 12: C++

```
1 @test_shared =
2   external dso_local
3   addrspace(3) global
4   [0 x double],
5   align 8
```

Listing 13: LLVM IR

Fig. 12 Comparison of the declaration of shared memory in C++ and LLVM IR. `addrspace(3)` indicates the pointer as pointing to memory in address space 3, which the NVPTX backend reserves for shared memory

6.3.2 LLVM NVIDIA specifics

Using LLVM for NVIDIA GPU code generation introduces some specifics. In C++ we have a special keyword `__shared__` to mark memory as being shared, while the NVPTX backend is using a separate address space as can be seen in Fig. 12.

Furthermore, there are functions in LLVM IR specific to CUDA. For example in C++, the thread id in one dimension is saved as a special built-in variable[8] called `threadIdx.x`, while in LLVM IR the thread id can be retained via a function called `@llvm.nvvm.read.ptx.sreg.tid.x`, which gets subsequently mapped onto a special register called `%tid.x` in PTX again.

6.3.3 CUDA fast math optimizations

Previously, we only looked at optimisations that happen on an LLVM IR level. In a second step, we can additionally consider how to optimise the PTX that is generated from our LLVM IR. Floating-point operations *normally* cannot change in ordering or kind by optimisation. However, we can relax these rules and thus allow the optimiser to reorder and replace floating-point operations.

To allow this relaxation, LLVM IR provides *fast math flags*[9] that allow the specification of how the optimiser is allowed to modify the order of floating-point operations. The most interesting two flags are `reassoc` and `contract`. `reassoc` means that floating-point operations can be treated as associative. This is part of LLVM IR already, since operations are reordered by the optimiser. `contract` allows to combine multiple floating-point operations. There are no combined floating-point operations in LLVM IR, so setting this flag is just information for the specific backend behind LLVM (in our case, of course, NVPTX) to allow for this optimisation.

Figure 14a shows the PTX code generated from the LLVM IR shown in Fig. 13. Here we do not have any fast math flags enabled. Highlighted in blue are all the floating-point calculations that match their LLVM IR equivalents roughly. For comparison, Fig. 14b shows the resulting PTX from Fig. 13 when the `contract` fast math flag is set. We see indeed that four floating-point operations, two multiplies

[8] https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
[9] https://llvm.org/docs/LangRef.html.

```
%4 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x(), !range !8
%5 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x(), !range !9
%6 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x(), !range !10
%CUDABuiltin_cpp_97_ = mul i32 %6, %5
%CUDABuiltin_cpp_97_1 = add i32 %CUDABuiltin_cpp_97_, %4
%CodeGen_cpp_1539_ = zext i32 %CUDABuiltin_cpp_97_1 to i64
%Autodiff_cpp_616_ = getelementptr double, double* %arg0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_3 = getelementptr double, double* %arg00, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_5 = getelementptr double, double* %3, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_7 = getelementptr double, double* %2, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_9 = getelementptr double, double* %0, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_616_11 = getelementptr double, double* %1, i64 %CodeGen_cpp_1539_
%Autodiff_cpp_519_ = load double, double* %Autodiff_cpp_616_, align 8
%Autodiff_cpp_519_12 = load double, double* %Autodiff_cpp_616_5, align 8
%Autodiff_cpp_519_13 = load double, double* %Autodiff_cpp_616_7, align 8
%Autodiff_cpp_501_ = fmul double %Autodiff_cpp_519_12, %Autodiff_cpp_519_13
%Autodiff_cpp_519_14 = load double, double* %Autodiff_cpp_616_11, align 8
%Autodiff_cpp_519_15 = load double, double* %Autodiff_cpp_616_9, align 8
%Autodiff_cpp_501_16 = fmul double %Autodiff_cpp_519_14, %Autodiff_cpp_519_15
%Autodiff_cpp_493_ = fadd double %Autodiff_cpp_501_, %Autodiff_cpp_501_16
%Autodiff_cpp_493_17 = fadd double %Autodiff_cpp_519_, %Autodiff_cpp_493_
%Autodiff_cpp_519_18 = load double, double* %Autodiff_cpp_616_3, align 8
%Autodiff_cpp_497_ = fsub double %Autodiff_cpp_493_17, %Autodiff_cpp_519_18
%Autodiff_cpp_317_19 = fmul double %Autodiff_cpp_497_, 2.000000e+00
%Autodiff_cpp_302_ = fmul double %Autodiff_cpp_519_14, %Autodiff_cpp_317_19
%Autodiff_cpp_302_23 = fmul double %Autodiff_cpp_519_12, %Autodiff_cpp_317_19
%7 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1539_
store double %Autodiff_cpp_302_, double* %7, align 8
%Autodiff_cpp_623_ = add i32 %CUDABuiltin_cpp_97_1, 32768
%CodeGen_cpp_1599_26 = zext i32 %Autodiff_cpp_623_ to i64
%8 = getelementptr inbounds double, double* %result, i64 %CodeGen_cpp_1599_26
store double %Autodiff_cpp_302_23, double* %8, align 8
ret void
```

Fig. 13 Optimized LLVM IR of reverse mode automatic differentiation of a linear model with three weights and mean squared error as loss function. Red lines mark GPU specific operations to determine memory positions, blue lines mark loading operations and brown lines mark storing operations

(`mul.rn.f64`) and two adds (`add.rn.f64`), have been fused together to two fused-multiply-and-add (`fma.rn.f64`) operations.

6.3.4 CUB

Another feature that is used by our kernels is a library called *CUB*.[10] CUB provides primitives to simplify block-wide synchronisations to synchronise all threads currently running in a kernel. This can be used to efficiently execute tasks by using all currently available threads within a kernel. CUB facilitates this by offering primitives utilising some or all available threads. CUB is a header-only library, so there is no code to link. To use the features of CUB—for example, to sum up all the corrections in our `sum_corrections` kernel—we create a pre-compiled LLVM module to make this primitive available to be used by our kernel.

[10] https://nvlabs.github.io/cub/.

```
ld.param.u64         %rd1,
 [single_kernel_translation_7414561934095481619_param_0];
ld.param.u64         %rd2,
 [single_kernel_translation_7414561934095481619_param_1];
mov.u32         %r1, %tid.x;
ld.param.u64         %rd3,
 [single_kernel_translation_7414561934095481619_param_2];
mov.u32         %r2, %ntid.x;
ld.param.u64         %rd4,
 [single_kernel_translation_7414561934095481619_param_3];
mov.u32         %r3, %ctaid.x;
ld.param.u64         %rd5,
 [single_kernel_translation_7414561934095481619_param_4];
mad.lo.s32         %r4, %r3, %r2, %r1;
ld.param.u64         %rd6,
 [single_kernel_translation_7414561934095481619_param_5];
ld.param.u64         %rd7,
 [single_kernel_translation_7414561934095481619_param_6];
mul.wide.u32         %rd8, %r4, 8;
add.s64         %rd9, %rd5, %rd8;
add.s64         %rd10, %rd6, %rd8;
add.s64         %rd11, %rd4, %rd8;
add.s64         %rd12, %rd3, %rd8;
add.s64         %rd13, %rd1, %rd8;
add.s64         %rd14, %rd2, %rd8;
ld.f64         %fd1, [%rd9];
ld.f64         %fd2, [%rd11];
ld.f64         %fd3, [%rd12];
mul.rn.f64         %fd4, %fd2, %fd3;
ld.f64         %fd5, [%rd14];
ld.f64         %fd6, [%rd13];
mul.rn.f64         %fd7, %fd5, %fd6;
add.rn.f64         %fd8, %fd4, %fd7;
add.rn.f64         %fd9, %fd1, %fd8;
ld.f64         %fd10, [%rd10];
sub.rn.f64         %fd11, %fd9, %fd10;
add.rn.f64         %fd12, %fd11, %fd11;
mul.rn.f64         %fd13, %fd5, %fd12;
mul.rn.f64         %fd14, %fd2, %fd12;
add.s64         %rd15, %rd7, %rd8;
st.f64         [%rd15], %fd13;
add.s32         %r5, %r4, 32768;
mul.wide.u32         %rd16, %r5, 8;
add.s64         %rd17, %rd7, %rd16;
st.f64         [%rd17], %fd14;
ret;
```

(a) Without fast math flags.

```
ld.param.u64         %rd1,
 [single_kernel_translation_16938563922865970329_param_0];
ld.param.u64         %rd2,
 [single_kernel_translation_16938563922865970329_param_1];
mov.u32         %r1, %tid.x;
ld.param.u64         %rd3,
 [single_kernel_translation_16938563922865970329_param_2];
mov.u32         %r2, %ntid.x;
ld.param.u64         %rd4,
 [single_kernel_translation_16938563922865970329_param_3];
mov.u32         %r3, %ctaid.x;
ld.param.u64         %rd5,
 [single_kernel_translation_16938563922865970329_param_4];
mad.lo.s32         %r4, %r3, %r2, %r1;
ld.param.u64         %rd6,
 [single_kernel_translation_16938563922865970329_param_5];
ld.param.u64         %rd7,
 [single_kernel_translation_16938563922865970329_param_6];
mul.wide.u32         %rd8, %r4, 8;
add.s64         %rd9, %rd6, %rd8;
add.s64         %rd10, %rd4, %rd8;
add.s64         %rd11, %rd5, %rd8;
add.s64         %rd12, %rd3, %rd8;
add.s64         %rd13, %rd1, %rd8;
add.s64         %rd14, %rd2, %rd8;
ld.f64         %fd1, [%rd11];
ld.f64         %fd2, [%rd10];
ld.f64         %fd3, [%rd12];
ld.f64         %fd4, [%rd14];
ld.f64         %fd5, [%rd13];
fma.rn.f64         %fd6, %fd4, %fd5, %fd1;
fma.rn.f64         %fd7, %fd2, %fd3, %fd6;
ld.f64         %fd8, [%rd9];
sub.rn.f64         %fd9, %fd7, %fd8;
add.rn.f64         %fd10, %fd9, %fd9;
mul.rn.f64         %fd11, %fd4, %fd10;
mul.rn.f64         %fd12, %fd2, %fd10;
add.s64         %rd15, %rd7, %rd8;
st.f64         [%rd15], %fd11;
add.s32         %r5, %r4, 32768;
mul.wide.u32         %rd16, %r5, 8;
add.s64         %rd17, %rd7, %rd16;
st.f64         [%rd17], %fd12;
ret;
```

(b) With fast math `contract` flag.

Fig. 14 Optimized PTX generated by the LLVM IR **a** without and **b** with enabling the fast math `contract` flag

Table 2 Some supported dimensions of A ($m \times k$), B ($k \times n$) and C ($m \times n$) for performing $C = A \times B$ on the tensor cores via the WMMA API. `tf32` is similar to a normal 32 bit `float`, but with 10 bit instead of 23 bit precision

A type	B type	C type	m	n	k
`__half`	`__half`	`__half`	16	16	16
`tf32`	`tf32`	`float`	16	16	8
`double`	`double`	`double`	8	8	4

`__half` is a 16 bit floating-point type

6.3.5 WMMA

As a downside of cuBLAS, it cannot be part of the JIT compilation, since the different functions are called by the host and not on the GPU. Beginning with the *Volta* architecture, NVIDIA integrated hardware acceleration of matrix operations in the

form of dedicated tensor cores.[11] We can use tensor cores for array operations within gradient descent while still allowing for just-in-time compilation.

WMMA is an CUDA API to access the tensor cores since the *Volta* architecture.[12] Instead of allowing for arbitrary matrix computations, WMMA only supports matrix-matrix multiplication of specific sizes depending on the type.

Table 2 shows some of the supported dimensions for different floating-point types available in CUDA. The API works by loading data into matrix `fragments`. Those `fragments` can be seen as special registers that hold the data for the tensor cores. After loading data into those `fragments`, we can execute the multiplication and store the result from a `fragment` into memory again. The WMMA instructions block all the threads in a warp. Therefore we can have one matrix multiplication per warp at the same time. Hence, one matrix multiplication per block as we use the warp size as block size.

For our simple example by Derakhshan et al. [30] with four weights, we can calculate the model or derivation for more than one tuple by loading the data in the appropriate shape into `fragment`. Examples with bigger dimensions are possible to be mapped onto WMMA because matrix multiplication is generally decomposable, for example, via Strassen's algorithm [72].

7 Evaluation

This section presents the evaluation of the operators in Umbra, in detail the performance increase through automatic differentiation in Umbra and code-generation for GPU, and the performance of our fine-grained learners on hardware level.

7.1 Automatic differentiation in umbra

Using synthetic data, we first compare three CPU-only approaches to compute batch gradient descent (the batch size corresponds to the number of tuples) on a linear model within SQL: Recursive tables with either manually or automatically derived gradients, and a dedicated (single-threaded) operator. All experiments were run multi-threaded on a 20-core Ubuntu 20.04.01 machine (Intel Xeon E5-2660 v2 CPU) with hyper-threading, running at 2.20 GHz with 256 GB DDR4 RAM.

Figure 15 shows the compilation and execution time depending on the number of involved attributes. As we see, automatically deriving the partial derivatives speeds up compilation time, as fewer expressions have to be compiled, as well as execution time, as subexpressions are cached in registers for reuse. This performance benefit is also visible when the batch size, the number of iterations or the number of threads is varied (see Figs. 16, 17). Furthermore, we observe the approach using recursive

[11] https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.
[12] https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

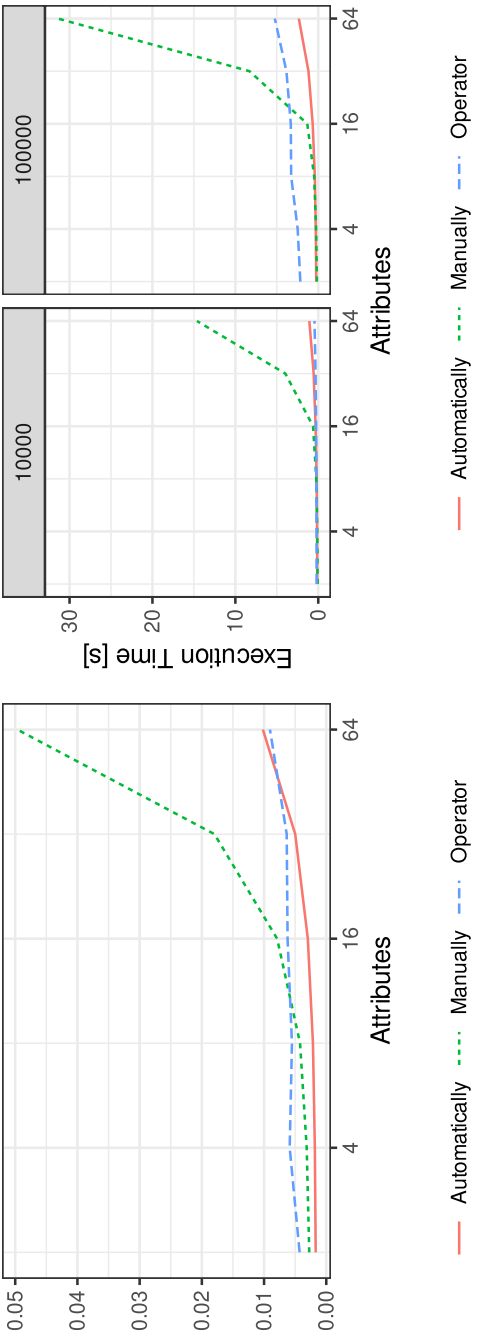


Fig. 15 Linear regression: Compilation and execution time with increasing number of attributes (100 iterations, 10, 000/100, 000 tuples)

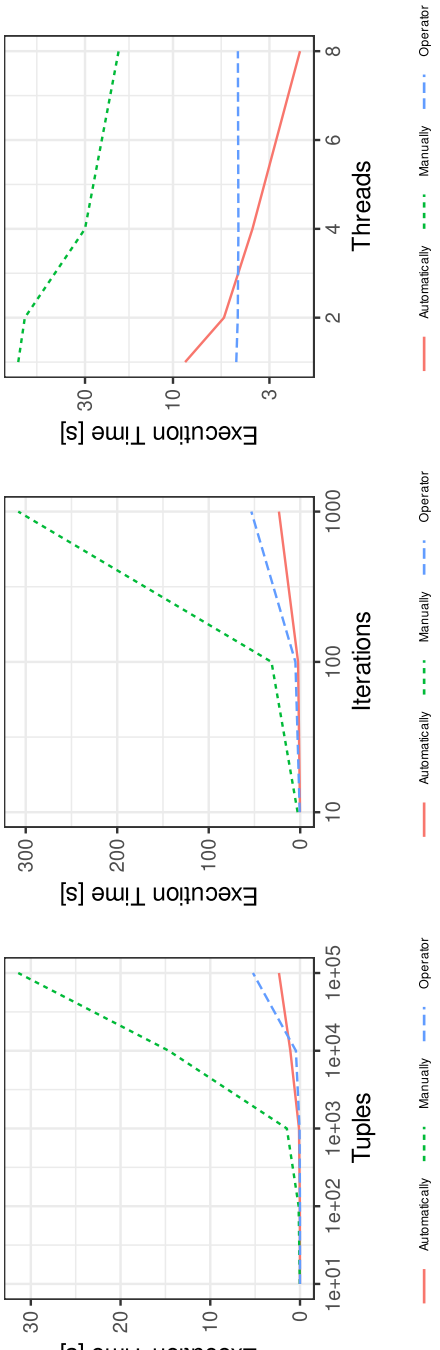


Fig. 16 Linear regression: Execution time (64 attributes) with increasing number of tuples (100 iterations, 8 threads), iterations (100, 000 tuples, 8 threads) or threads (100 iterations, 100, 000 tuples)

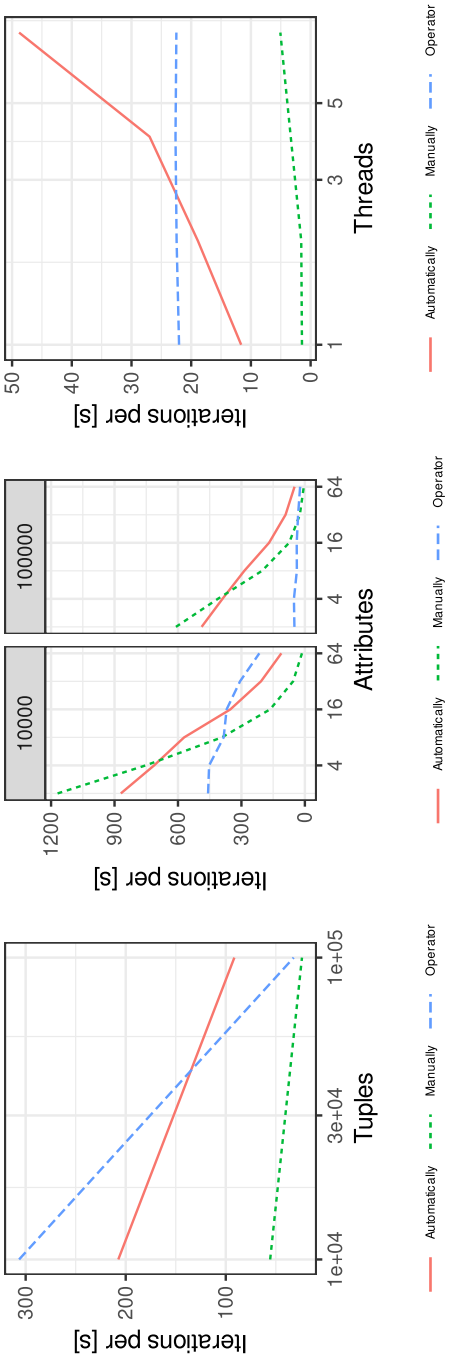


Fig. 17 Iterations per second for linear regression

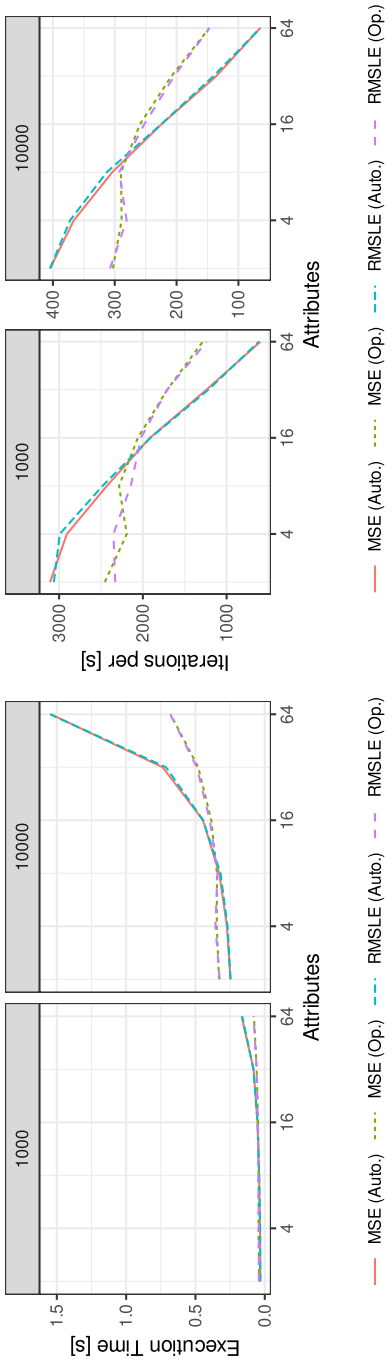


Fig. 18 Different loss functions for linear regression (mean squared error *MSE*, mean squared logarithmic error *RMSLE*): with increasing number of attributes (100 iterations, 1000/10, 000 tuples)

tables computes aggregations in parallel, which accelerates computation on many input tuples with each additional thread.

The implemented derivation rules for automatic differentiation also allow deriving the gradient for arbitrary loss functions. Therefore, we also tested two different loss functions for linear regression, namely mean squared error (*MSE*, Eq. 2) and mean squared logarithmic error (*RMSLE*, Eq. 9). Figure 18 shows that the choose of the loss function does not influence the runtime significantly.

```
with recursive gd (id, a1, a2, b) as (
   select 1,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(2*x1*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   a2-0.5*avg(2*x2*(sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b)))),
   b -0.5*avg(2*   (sig(a1*x1+a2*x2+b)-y2)*(sig(a1*x1+a2*x2+b)*(1-sig(a1*x1+a2*x2+b))))
   from gd, data where id < 50 group by id,a1,a2,b)
select * from gd order by id;

with recursive gd (id, a1, a2, b) as (
   select 1,1::float,1::float,1::float
UNION ALL
   select id+1,
   a1-0.5*avg(d_a1), a2-0.5*avg(d_a2), b-0.5*avg(d_b)
   from umbra.derivation(TABLE (
      select * from gd, (select * from data) where id < 50),
      lambda (x) ((sig(b+x.a1 * x.x1 + x.a2 * x.x2 ) - x.y2)^2))
   group by id,a1,a2,b)
select * from gd order by id;

select * from umbra.gd(TABLE (select * from data), TABLE (select 1::float a1, 1::float a2, 1::float b), lambda (
      x,y) ((sig(y.b + y.a1 * x.x1 + y.a2 * x.x2) - x.y2)^2), 50, 0.5, 0);
```

Listing 14: Logistic regression in SQL.

Using a recursive table, we can also solve logistic regression in SQL (see Listing 12). Using the sigmoid function (Eq. 18), we know its derivative (see Table 1), which we can use for numeric differentiation:

$$m_{\mathbf{w}}(\mathbf{x}) = sig(\mathbf{x}^T \cdot \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{x}^T \cdot \mathbf{w}}}. \tag{18}$$

The evaluation in Fig. 19 shows similar results as for linear regression, although function calls for the sigmoid function slowed down the runtime especially when derived manually.

To benchmark training a neural network, we are using the Fisher's Iris flower dataset [73] and a fully connected, two-layered neural network. In this experiment, we rather focus on the performance characteristics of the operator for automatic differentiation embedded in relational algebra than on the quality of different models. Figure 20 shows the compilation and execution time when training the neural network depending on the size of the hidden layer. As the size of the hidden layer depends on the weight matrices created at runtime, it does not affect the compile time. Whereas the runtime directly depends on the number of operations involved in matrix multiplications and thus increases with the size of the matrices. When we compare automatic differentiation to manually derived gradients, we applied the backpropagation rules by hand. So fewer operations were executed when using automatic differentiation, which currently computes the derivative for every variable involved. Hence, both approaches show similar performance when varying the size of the hidden layer, the batch size, the number of

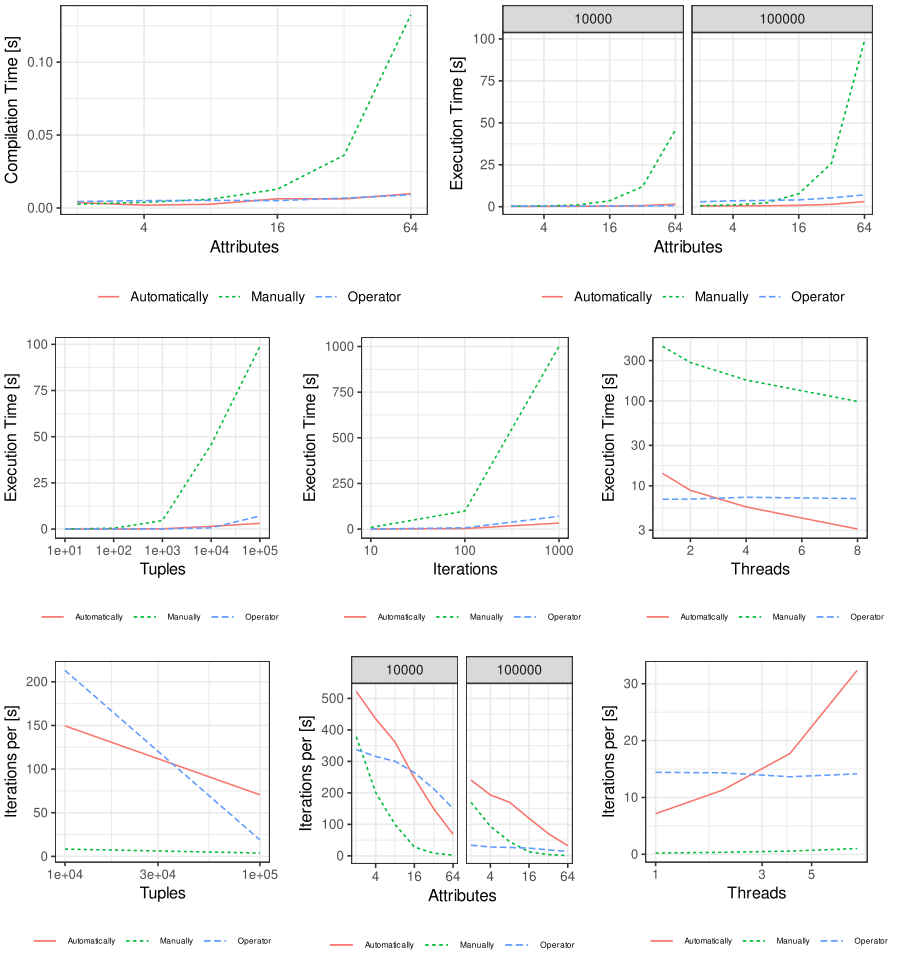


Fig. 19 Logistic regression: Performance with increasing number of attributes (100 iterations, 10, 000/100, 000 tuples), increasing number of tuples (64 attributes, 100 iterations, 8 threads), iterations (64 attributes, 100, 000 tuples, 8 threads) or threads (64 attributes, 100 iterations, 100, 000 tuples)

iterations and threads (see Fig. 21). Nevertheless, automatic differentiation eliminates the need for manually derived gradients and nested subqueries.

Figure 22 displays further experiments such as varying the number of hidden layers and the training time depending on the batch size for one epoch, so when all tuples have been processed once, using the MNIST dataset. As expected, the runtime increases with additional layers and the training time for one epoch decreases with a higher batch size. We discuss the influence of the batch size on the statistical efficiency in Sect. 7.

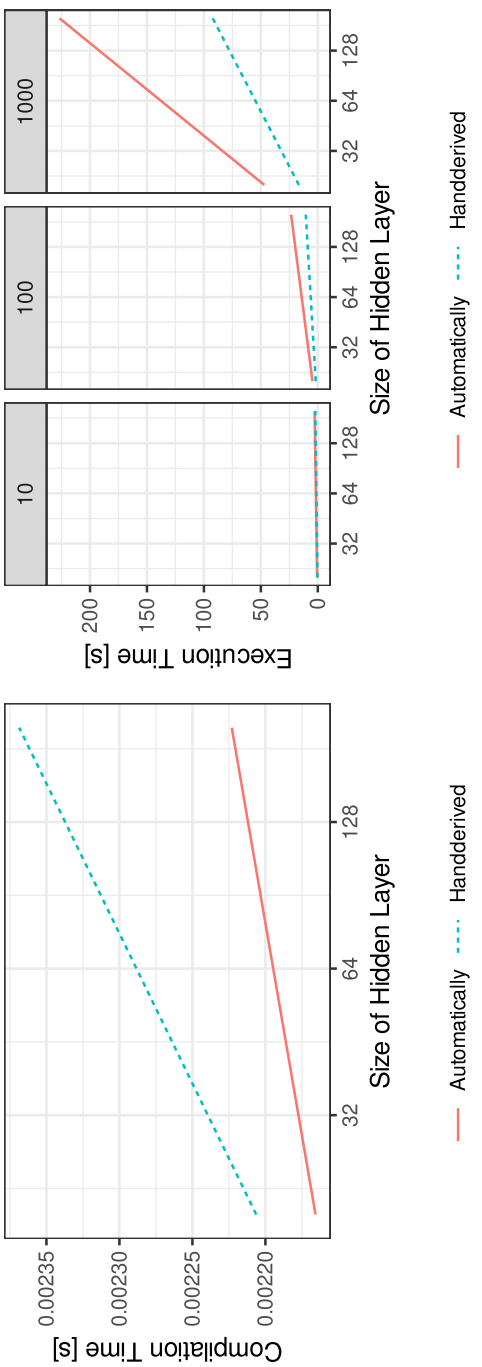


Fig. 20 Neural network: Compilation and execution time with increasing size of the hidden layer (2000 iterations, batch size 10/100/1000 tuples)

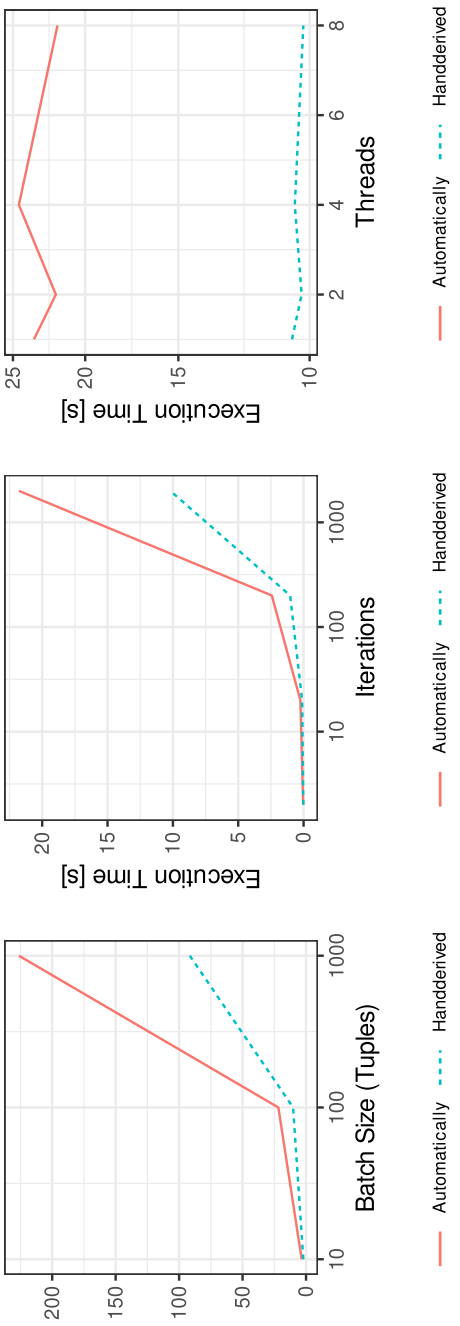


Fig. 21 Neural network: Execution time (hidden layer $h = 200$) with increasing batch size (2000 iterations, 8 threads), iterations (batch size 100 tuples, 8 threads) or threads (2000 iterations, batch size 100 tuples)

7.2 Linear regression (GPU)

We benchmark linear regression with synthetic data and the New York taxi dataset, and a feed-forward neural network with a single hidden layer for image recognition (see Table 3). We take 2.65 GiB of the New York taxi dataset[13] (January 2015), on which we perform linear regression to forecast the taxi trip duration from the trip's distance and bearing, the day and the hour of the ride's beginning (four attributes). We tested on servers with four Intel Xeon Gold 5120 processors, each with 14 CPUs (2.20 GHz) running Ubuntu 20.04.01 LTS with 256 GiB RAM. Each server is connected to either four GPUs (NVIDIA GeForce GTX 1080 Ti/RTX 2080 Ti) or one NVIDIA Tesla V100-PCIE-32GB.

We measure the performance and the quality of the different parallelisation models on the CPU as well as the GPU according to the following metrics:

1. *Throughput* measures the size of processed tuples per time. It includes tuples used for training as well as for validation.
2. *Time-to-loss*, similarly to *time-to-accuracy* [74] for classification problems, measures the minimal loss on the validation dataset depending on the computation time.
3. *Tuples-to-loss* describes how many tuples are needed to reach a certain minimal loss. In comparison to *time-to-loss*, it is agnostic to the hardware throughput and measures the quality of parallelisation and synchronisation methods.

We perform gradient descent with a constant learning rate of 0.5 to gain the optimal weights. After a predefined validation frequency, every 3,000 batches, the current loss is computed on a constant validation set of 20 % the size of the original one. We vary the hyper-parameters of our implementation, i.e., the batch size and the number of workers. A thread records the current state every second to gather loss metrics.

7.2.1 Throughput vs. statistical efficiency

To measure the throughput for linear regression on different hardware, we consider batch sizes of up to 100 MiB. We compare the performance of our kernels to that when stochastic gradient descent of the TensorFlow (version 1.15.0) library is called (see Fig. 23).

The higher the batch size, the better the throughput when running gradient descent on GPUs as all concurrent threads can be utilised. Hardware-dependent, the maximal throughput converges to either 150 GiB/s (GeForce GTX 1080 Ti), 250 GiB/s (GeForce RTX 2080 Ti) or more than 300 GiB/s (Tesla V100). As developed for batched processing, our dedicated kernels (see Fig. 23a) can exploit

[13] https://s3.amazonaws.com/nyc-tlc/trip+data/yellow_tripdata_2015-01.csv.

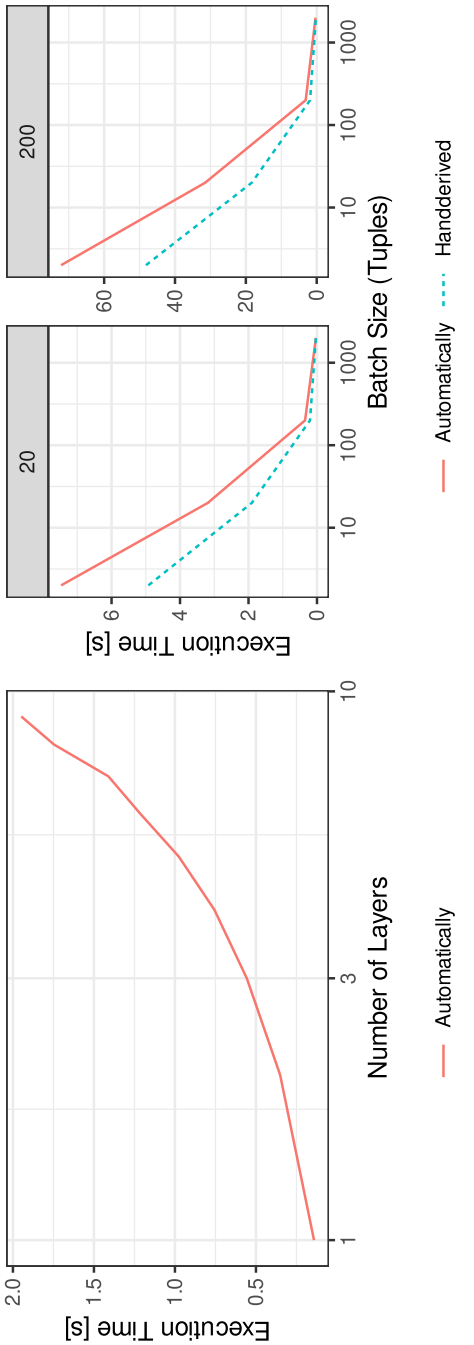


Fig. 22 Neural network: Execution time depending on the number of hidden layers (each with a size of 20, batch size 2 tuples, 100 iterations) and training time for one epoch on the MNIST dataset depending on the batch size (hidden layer $h = 20/200$)

Table 3 Datasets used with linear regression and a neural network respectively

	#attr.	*#training*	*#validation*
New York Taxi	4 + 1	61,664,460	15,416,115
Synthetic	99 + 1	10	10
MNIST	784 + 1	60,000	10,000
Fashion-MNIST	784 + 1	60,000	10,000

available hardware more effectively than the Python implementation (see Fig. 23b). As the latter calls stochastic gradient descent, this excels on better hardware only when a larger model has to be trained.

Nevertheless, training with large batch sizes does not imply statistical efficiency in terms of the volume of processed data that is needed for convergence (see Fig. 24a) and the lowest reachable minimal loss (see Fig. 24b). For that reason, to allow the highest throughput even for small batch sizes, we implement multiple learners per GPU.

7.2.2 Multiple learners per GPU

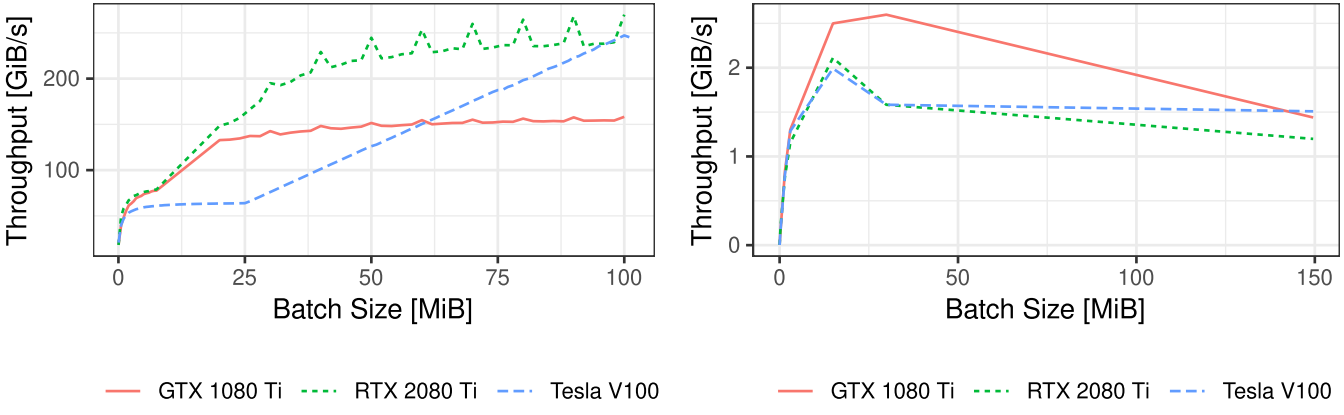
As the GPU is only fully utilised when the number of concurrently processed tuples is greater or equal to the number of parallel GPU threads, we benchmark multiple learners per GPU. As each learner corresponds to one GPU block consisting of a multiple of 32 threads, our implementation allows the highest throughput for every batch size, as a multiple of the block size. Therefore, we vary the number of threads per block (equal to a learner) between 32 and 1,024 and measure the throughput dependent on the batch size in multiples of 32 threads.

The observation in Fig. 25 corresponds to the expectation that a small number of threads per learner allows a higher throughput for small batch sizes. When the batch size is equal to a multiple of the chosen number of threads, the throughput reaches a local maximum. Otherwise, the GPU is underutilised. These local maxima are visible as spikes in all curves except for 32 threads per block, as we increase the batch size by 32 tuples. Nevertheless, on all devices, the throughput soon converges at the possible maximum, which shows the efficiency of learners in the granularity of GPU warps.

7.2.3 Scalability

When running gradient descent in parallel, we benchmark the four implementations for synchronising weights: no synchronisation with global updates (*global updates*), maintaining local models either with locking of the critical section (*local models (locks)*) or without locking (*local models (dirty)*), or synchronised updates that block until every worker has finished (*synchronised (blocking)*). We ran the experiments on the CPU as well as the GPU.

When parallelising on the CPU, each additional thread allows a linear speed-up when no synchronisation takes place (see Fig. 26a). Maintaining local models costs



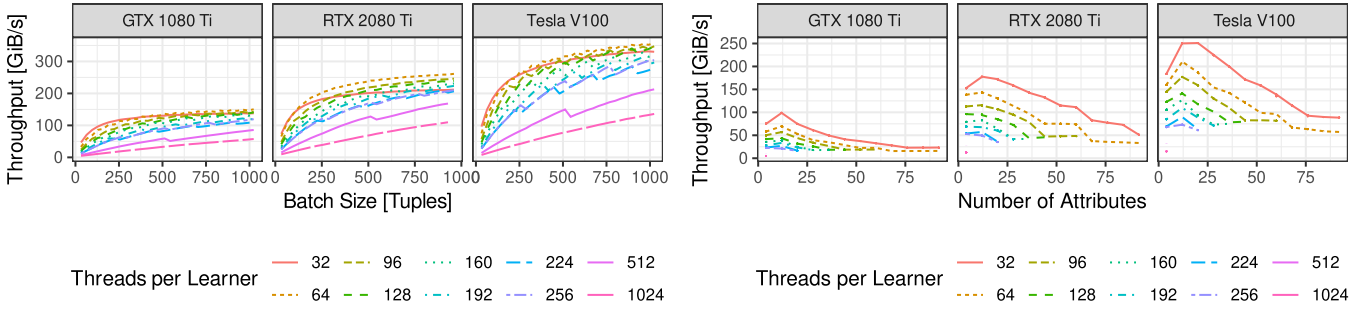
(a) Our implementation.

(b) Using TensorFlow.

Fig. 23 Throughput of **a** the dedicated kernels and **b** using the TensorFlow library

(a) Data to converge. **(b)** Minimal loss.

Fig. 24 Statistical efficiency for linear regression: **a** volume of processed data needed to converge and **b** minimal reachable loss depending on the batch size



(a) Varying the batch size (four attributes).

(b) Number of attr. (64 tuples per batch), synthetic data.

Fig. 25 Throughput with multiple learners per GPU: A smaller number of threads per learner allows the maximum throughput even for small batch sizes when small batches are processed in parallel

additional computation time, which results in a lower throughput. Obviously, locks slow down the speed up, and blocking threads cause underutilisation.

Whereas parallelising for GPUs behaves differently (see Fig. 26b/c): the larger the batch size, the higher the scale-up. This is obvious, as less synchronisation is necessary for larger batch sizes and the parallel workers can compute the gradients independently. Also on GPUs, the implementation without any synchronisation and global updates scales best, even though not as linearly as on CPUs. In all implementations, one additional GPU allows a noticeably higher throughput. Maintaining local models requires inter-GPU communication of the local corrections to form the global weights, which decreases the performance significantly with the third additional device. To minimise this effect, the weight computation could be split up hierarchically.

7.3 Neural network (GPU)

To benchmark the feed-forward neural network, we perform image classification using the MNIST and Fashion-MNIST [75] dataset. We train the neural network with one hidden layer of size 200 to recognise a written digit (Fashion-MNIST: a piece of clothing) given as a single tuple representing an image with 784 pixels. We take 0.025 as learning rate, perform a validation pass every epoch and measure the throughput and the time to reach a certain accuracy (with the loss defined as the number of incorrectly classified tuples).

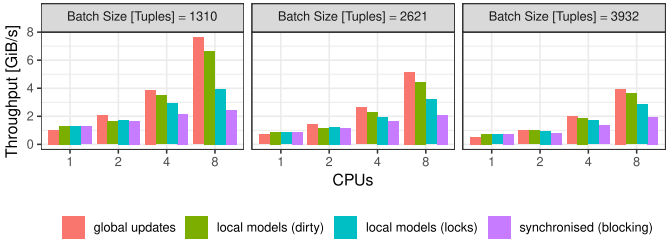
7.3.1 Throughput vs. statistical efficiency

Even though stochastic gradient descent using Keras (version 2.2.4) with TensorFlow allows a higher bandwidth than for linear regression due to more attributes per tuple (see Fig. 27b), our implementations, which call the cuBLAS library, process tuples batch-wise, which results in a higher bandwidth. As training a neural network is compute-bound involving multiple matrix multiplications, the throughput is significantly lower than for linear regression (see Fig. 27a), but allows a higher throughput, the larger the batch size.

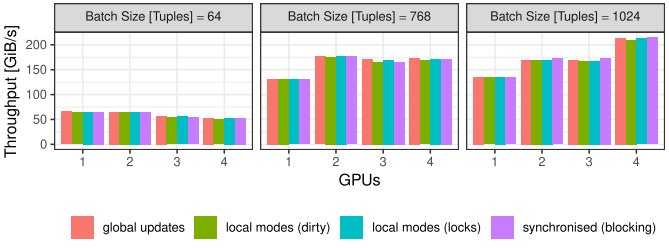
As is the case for linear regression, training models with small batch sizes results in a higher accuracy (see Fig. 28b). This once again makes the case for multiple learners per single GPU. Nevertheless, the larger the chosen batch size is, the faster training iterations converge (see Fig. 28a).

7.3.2 Scalability

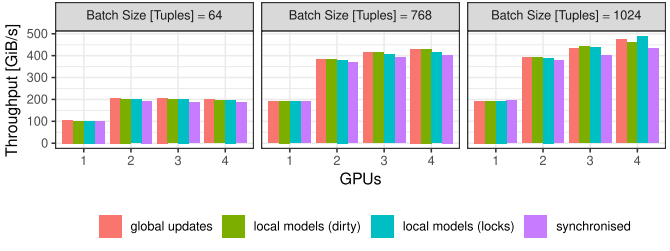
The scalability of parallel workers computing backpropagation resembles the scalability for training linear regression on GPUs: one additional worker increases the throughput, for any further workers, the inter-GPU communication decreases the runtime (see Fig. 29). For small batch sizes, training on two GPU devices has the best results, while for larger batch sizes, every additional device allows a higher throughput.



(a) CPU (Intel Xeon Gold 5120)

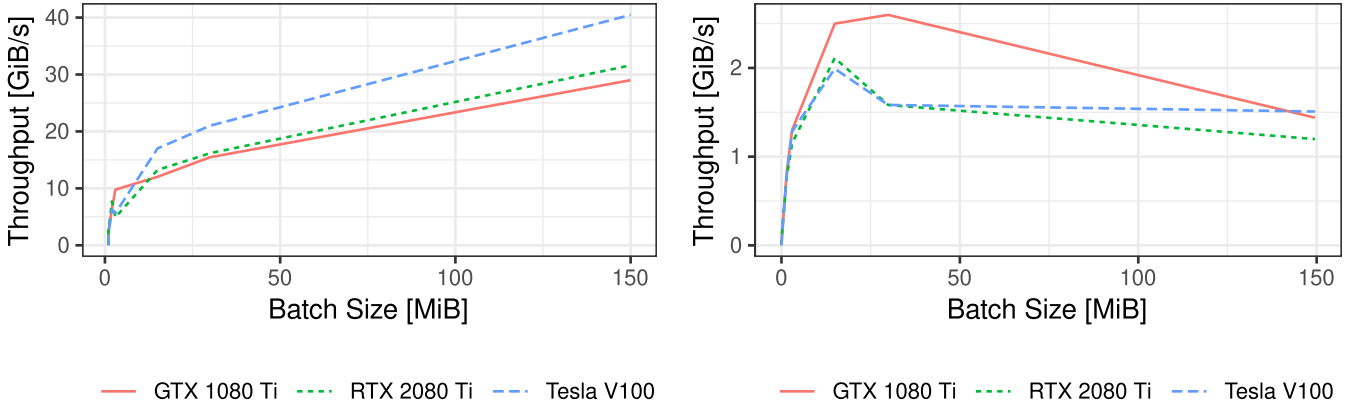


(b) NVIDIA GeForce GTX 1080 Ti



(c) NVIDIA GeForce RTX 2080 Ti

Fig. 26 Scale-up for linear regression on **a** multiple CPUs, **b** multiple NVIDIA GeForce GTX 1080 Ti or **c** multiple NVIDIA GeForce GTX 2080 Ti



(a) Own implementation. **(b)** Using Keras.

Fig. 27 Throughput of the implementation **a** using cuBLAS and **b** using Keras when training a neural network with the MNIST dataset

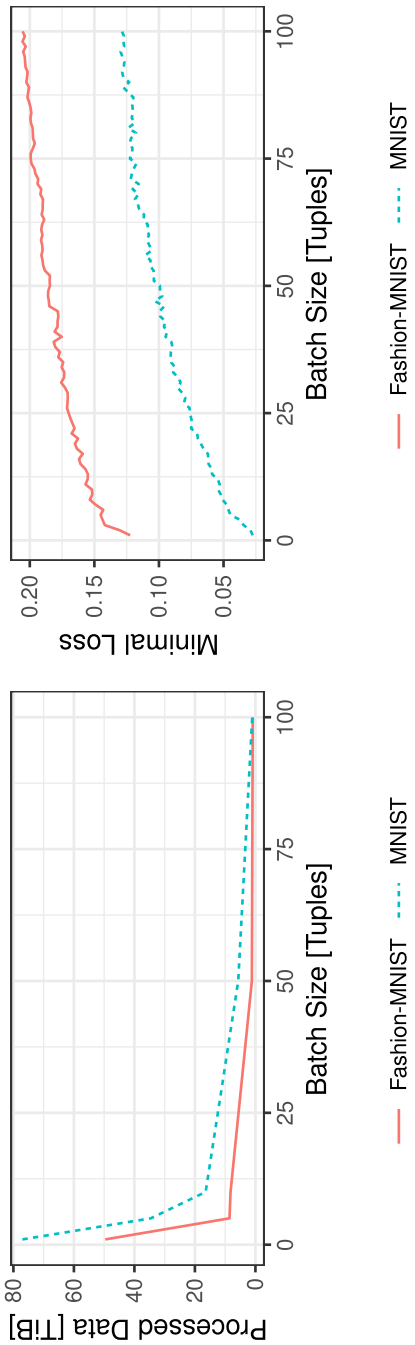


Fig. 28 Statistical efficiency for the neural network: **a** volume of processed data needed to converge and **b** minimal reachable loss depending on the batch size

7.3.3 Time/tuples-to-loss

Regarding the time to reach a certain accuracy (see Fig. 30), all implementations perform similarly when running on a single worker. As the MNIST dataset converges fast, adding a GPU device for computation has no significant impact. Whereas the Fashion-MNIST dataset converges slower, the higher throughput when training with an additional worker results in the minimal loss being reached faster. We train with a small batch size as it allows faster convergence. Hereby, a scale-up is only measurable when training with up to two devices.

7.4 Integration into umbra

We want to evaluate code generation for GPU into Umbra in several ways. We evaluate code-generation having multiple learners and the performance increase for matrix multiplication using WMMA.

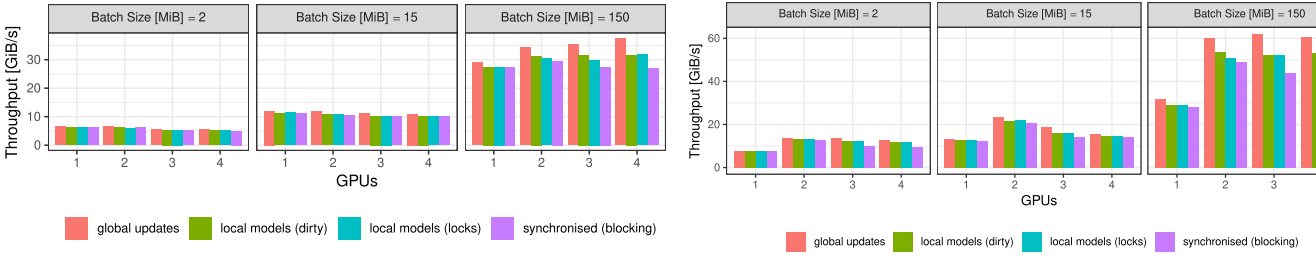
7.4.1 Code-generation: multiple learners

To investigate multiple learners per GPU, we want to compare the implementation generated by Umbra IR against the CUDA C++ implementation, which trains the taxi model. That way, we confirm that we reach a similar performance when using code-generation with Umbra IR.

Figure 31 compares three different versions of the Crossbow implementation, having 3328 workers, 32 threads per block, one batch per worker, 25199507 tuples on a single NVIDIA GTX 970. The CUDA C++ implementation corresponds to the Crossbow approach but prebuilt in C++ via CUDA. The naive implementation is a one-to-one translation of the CUDA C++ kernels into Umbra IR. Finally, the optimised version builds upon the naive version by integrating optimisations like loop unrolling. For testing, we focus on the training part of Crossbow. As we can see, we come very close to the CUDA C++ implementation with our optimised solution.

7.4.2 WMMA

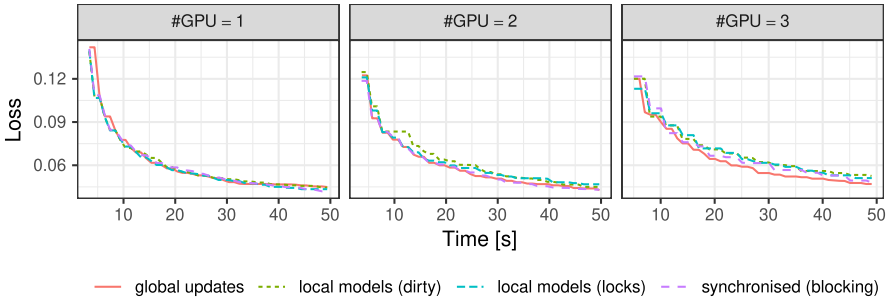
We want to evaluate whether the matrix multiplication of WMMA is more feasible than cuBLAS for training. Figure 32 shows the performance comparison of throughput for the taxi model matrix multiplication using WMMA and cuBLAS with `__half` as data type on a NVIDIA GeForce RTX 3050 Ti. The cuBLAS performance is worse than using `doubles`, which might be explained by the fact the cuBLAS is not working well with the `__half` type in this case. For the evaluation of the taxi model, we had to arrange the matrix in a specific way to maximise the number of tuples evaluated per tensor core invocation. Despite this rearrangement, we achieve a much higher throughput using WMMA.



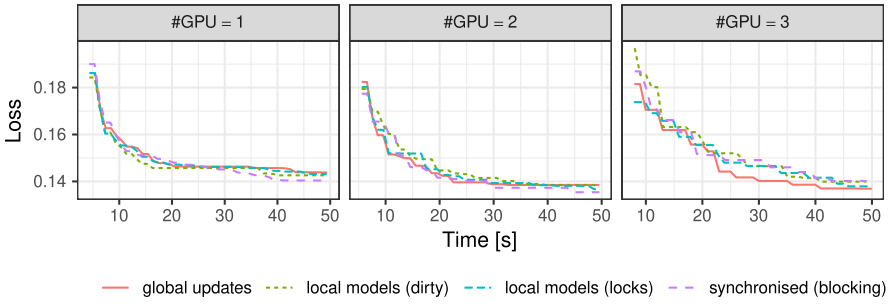
(a) NVIDIA GeForce GTX 1080 Ti

(b) NVIDIA GeForce RTX 2080 Ti

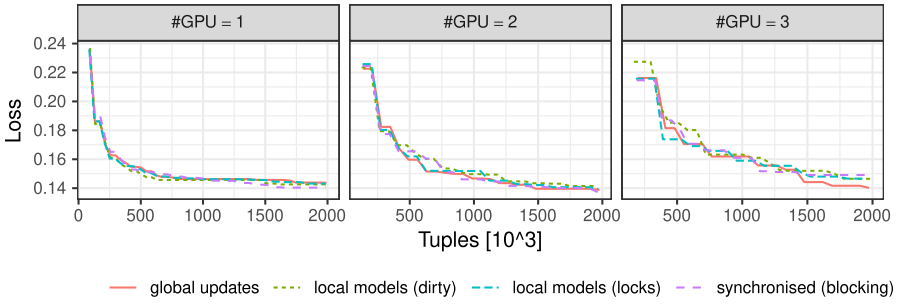
Fig. 29 Scale-up for training a neural network (with the MNIST dataset)



(a) Time-to-loss: MNIST dataset



(b) Time-to-loss: Fashion-MNIST dataset



(c) Tuples-to-loss: Fashion-MNIST dataset

Fig. 30 Time-to-loss when training the neural network for the **a** MNIST and **b** Fashion-MNIST dataset with a batch size of 5 tuples (NVIDIA GeForce GTX 2080 Ti). For Fashion-MNIST, also tuples-to-time (**c**) is provided

7.5 End-to-end analysis

Figure 33 compares the time needed to train one epoch (New York taxi data: $13 \cdot 10^6$ tuples, MNIST: $6 \cdot 10^4$ tuples) within a complete machine learning pipeline in Python

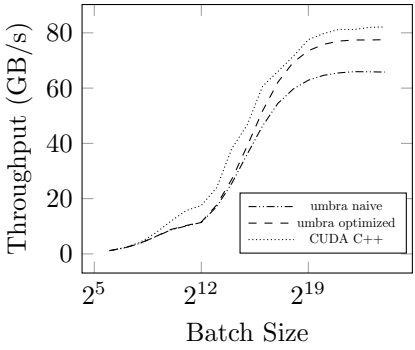


Fig. 31 Umbra integration

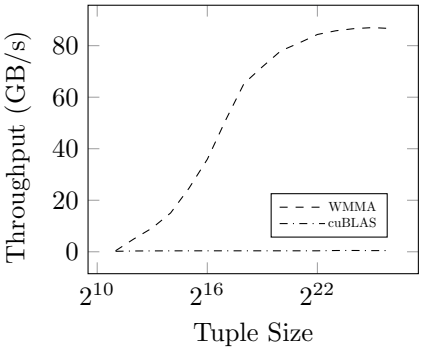


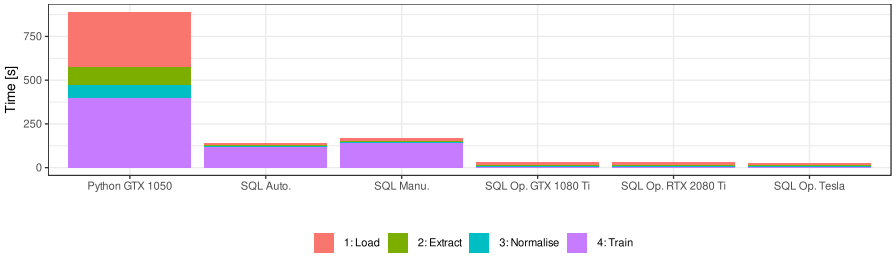
Fig. 32 cuBLAS vs. WMMA

using Keras to a corresponding operator tree within the database system Umbra, either using a recursive table with a manually or automatically derived gradient or using an operator that off-loads to GPU. The pipeline consists of data loading from CSV, feature extraction (only for the New York taxi data) and normalisation either with NumPy or SQL-92 queries, and training. Measurements using the MNIST dataset were performed on a Ubuntu 20.04 LTS machine with four cores of Intel i7-7700HQ CPU, running at 2.80 GHz clock frequency each and 16 GiB RAM. For the taxi data, a machine with Intel Xeon Gold 5120 processors, each with 14 CPUs (2.20 GHz) and 256 GiB RAM, was used.

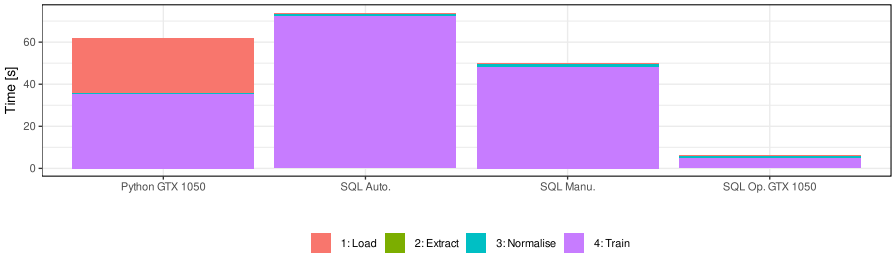
We observe that much time is spent on data loading and preprocessing. These tasks are either no longer required if the data is already stored inside the database system, or can easily be processed in parallel pipelines. Furthermore, gradient descent using recursive tables showed comparable performance to library functions used, which is still outperformed by our operator that off-loads training to GPU.

8 Conclusion

This paper has created an in-database machine learning pipeline expressed in pure SQL based on sampling, continuous views and recursive tables. To facilitate gradient descent, we proposed an operator for automatic differentiation and one for



(a) Linear regression (NY taxi).



(b) Neural network (MNIST).

Fig. 33 End-to-end analysis of a machine learning pipeline: **a** linear regression (*New York taxi*, 64 tuples per batch and **b** a neural network (*MNIST*, stochastic gradient descent)

gradient descent. The derivation rules allow deriving arbitrary models needed for training, for example, deep neural networks with multiple hidden layer. We used the operator for automatic differentiation for training within a recursive table in SQL. SQL provided flexibility as the iteration variable allows varying the learning rate or when accessing further tables for user-specific parameters. To off-load training to GPU units, we have implemented training algorithms as GPU kernels and fine-tuned learners at hardware level to increase the learning throughput. These kernels were integrated inside the code-generating database system Umbra. In comparison to handwritten derivatives, automatic differentiation as a database operator accelerated both the compile time and the execution time by the number of cached expressions. Furthermore, our evaluation benchmarked GPU kernels on different hardware, as well as parallelisation techniques with multiple GPUs. The evaluation has shown that GPUs traditionally excel the bigger the chosen batch sizes, which was only worthwhile when a slow-converging model was being trained. In addition, larger batch sizes interfered with statistical efficiency. For that reason, our fine-tuned learners at hardware level allowed the highest possible throughput for small batch sizes equal to a multiple of a GPU warp, so at least 32 threads. Our synchronisation techniques scaled up learning with every additional worker, even though this was not as linear for multiple GPU devices as for parallel CPU threads. Finally, our end-to-end machine learning pipeline in SQL showed comparable performance to traditional machine learning frameworks. In the future, translation from Python into SQL could increase the acceptance for in-database machine learning.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Belhajjame, K.: On discovering data preparation modules using examples. In: ICSOC. Lecture Notes in Computer Science, vol. 12571, pp. 56–65. Springer (2020)
2. Filatov, M., Kantere, V.: PAW: a platform for analytics workflows. In: EDBT. OpenProceedings.org, pp. 624–627 (2016)
3. Chen, Y., Biookaghazadeh, S., Zhao, M.: Exploring the capabilities of mobile devices in supporting deep learning. In: SEC. ACM, pp. 127–138 (2019)
4. Yuan, M., Zhang, L., Li, X., Xiong, H.: Comprehensive and efficient data labeling via adaptive model scheduling. In: ICDE, pp. 1858–1861. IEEE (2020)
5. Hadian, A., Kumar, A., Heinis, T.: Hands-off model integration in spatial index structures. In: AIDB@VLDB (2020)
6. Yang, Y., Meneghetti, N., Fehling, R., Liu, Z.H., Kennedy, O.: Lenses: an on-demand approach to ETL. Proc. VLDB Endow. **8**(12), 1578–1589 (2015)
7. Andrejev, A., Orsborn, K., Risch, T.: Strategies for array data retrieval from a relational back-end based on access patterns. Computing **102**(5), 1139–1158 (2020)
8. Eslami, M., Tu, Y., Charkhgard, H., Xu, Z., Liu, J.: PsiDB: a framework for batched query processing and optimization. In: IEEE BigData, pp. 6046–6048. IEEE (2019)
9. Villarroya, S., Baumann, P.: On the integration of machine learning and array databases. In: ICDE, pp. 1786–1789. IEEE (2020)
10. Dong, B., Wu, K., Byna, S., Liu, J., Zhao, W., Rusu, F.: ArrayUDF: user-defined scientific data analysis on arrays. In: HPDC, pp. 53–64. ACM (2017)
11. Prasad, S., Fard, A., Gupta, V., Martinez, J., LeFevre, J., Xu, V., et al.: Large-scale predictive analytics in vertica: fast data transfer, distributed model creation, and in-database prediction. In: SIGMOD Conference. ACM, pp. 1657–1668 (2015)
12. Stockinger, K., Bundi, N., Heitz, J., Breymann, W.: Scalable architecture for Big Data financial analytics: user-defined functions vs. SQL. J. Big Data **6**, 46 (2019)
13. Zhang, C., Toumani, F.: Sharing computations for user-defined aggregate functions. In: EDBT, pp. 241–252. OpenProceedings.org (2020)
14. LeFevre, J., Sankaranarayanan, J., Hacigümüs, H., Tatemura, J., Polyzotis, N., Carey, M.J.: Opportunistic physical design for big data analytics. In: SIGMOD Conference, pp. 851–862. ACM (2014)
15. Wagner, J., Rasin, A., Heart, K., Malik, T., Grier, J.: DF-toolkit: interacting with low-level database storage. Proc. VLDB Endow. **13**(12), 2845–2848 (2020)
16. Arzamasova, N., Böhm, K., Goldman, B., Saaler, C., Schäler, M.: On the usefulness of SQL-query-similarity measures to find user interests. IEEE Trans. Knowl. Data Eng. **32**(10), 1982–1999 (2020)
17. May, N., Böhm, A., Lehner, W.: SAP HANA—the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. In: BTW. vol. P-265 of LNI. GI; 2017. p. 545–563
18. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. Proc. VLDB Endow. **4**(9), 539–550 (2011)
19. Hubig, N.C., Passing, L., Schüle, M.E., Vorona, D., Kemper, A., Neumann, T.: HyPerInsight: data exploration deep inside HyPer. In: CIKM, pp. 2467–2470. ACM (2017)
20. Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206. IEEE Computer Society (2011)

21. Schüle, M.E., Schliski, P., Hutzelmann, T., Rosenberger, T., Leis, V., Vorona, D., et al.: Monopedia: staying single is good enough—the HyPer way for web scale applications. Proc. VLDB Endow. **10**(12), 1921–1924 (2017)
22. Neumann, T., Freitag, M.J.: Umbra: a disk-based system with in-memory performance. In: CIDR. www.cidrdb.org (2020)
23. Karnowski, L., Schüle, M.E., Kemper, A., Neumann, T.: Umbra as a time machine. In: BTW. vol. P-311 of LNI, pp. 123–132. Gesellschaft für Informatik, Bonn (2021)
24. Schmeißer, J., Schüle, M.E., Leis, V., Neumann, T., Kemper, A.: B^2-tree: cache-friendly string indexing within B-trees. In: BTW. vol. P-311 of LNI, pp. 39–58. Gesellschaft für Informatik, Bonn (2021)
25. Schmeißer, J., Schüle, M.E., Leis, V., Neumann, T., Kemper, A.: B^2-tree: page-based string indexing in concurrent environments. Datenbank-Spektrum. **22**(1), 11–22 (2022)
26. Schüle, M.E., Götz, T., Kemper, A., Neumann, T.: ArrayQL for linear algebra within umbra. In: SSDBM, pp. 193–196. ACM (2021)
27. Schüle, M.E., Götz, T., Kemper, A., Neumann, T.: ArrayQL integration into code-generating database systems. In: EDBT. OpenProceedings.org, pp. 1:40–1:51 (2022)
28. Sadoghi, M., Bhattacherjee, S., Bhattacharjee, B., Canim, M.: L-Store: a real-time OLTP and OLAP system. In: EDBT, pp. 540–551. OpenProceedings.org (2018)
29. Kang, W., Lee, S., Moon, B.: Flash as cache extension for online transactional workloads. VLDB J. **25**(5), 673–694 (2016)
30. Derakhshan, B., Mahdiraji, A.R., Rabl, T., Markl, V.: Continuous deployment of machine learning pipelines. In: EDBT, pp. 397–408. OpenProceedings.org (2019)
31. Bär, A., Casas, P., Golab, L., Finamore, A.: DBStream: an online aggregation, filtering and processing system for network traffic monitoring. In: IWCMC, pp. 611–616. IEEE (2014)
32. Li, Z., Ge, T.: Stochastic data acquisition for answering queries as time goes by. Proc. VLDB Endow. **10**(3), 277–288 (2016)
33. Tu, Y., Kumar, A., Yu, D., Rui, R., Wheeler, R.: Data management systems on GPUs: promises and challenges. In: SSDBM, pp. 33:1–33:4. ACM (2013)
34. Terenin, A., Dong, S., Draper, D.: GPU-accelerated Gibbs sampling: a case study of the Horseshoe Probit model. Stat Comput. **29**(2), 301–310 (2019)
35. Jiang, P., Hong, C., Agrawal, G.: A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In: PPoPP, pp. 376–388. ACM (2020)
36. Yu, F., Harbor, J.M.: CSTAT+: a GPU-accelerated spatial pattern analysis algorithm for high-resolution 2D/3D hydrologic connectivity using array vectorization and convolutional neural network operators. Environ. Model. Softw., p. 120 (2019)
37. Dolmatova, O., Augsten, N., Böhlen, M.H.: A relational matrix algebra and its implementation in a column store. In: SIGMOD Conference. ACM, pp. 2573–2587 (2020)
38. Jiang, P., Agrawal, G.: Accelerating distributed stochastic gradient descent with adaptive periodic parameter averaging: poster. In: PPoPP, pp. 403–404. ACM (2019)
39. Ma, Y., Rusu, F., Torres, M.: Stochastic gradient descent on modern hardware: multi-core CPU or GPU? Synchronous or asynchronous? In: IPDPS, pp. 1063–1072. IEEE (2019)
40. Beldianu, S.F., Ziavras, S.G.: On-chip vector coprocessor sharing for multicores. In: PDP, pp. 431–438. IEEE Computer Society (2011)
41. Koliousis, A., Watcharapichat, P., Weidlich, M., Mai, L., Costa, P., Pietzuch, P.R.: Crossbow: scaling deep learning with small batch sizes on multi-GPU servers. Proc. VLDB Endow. **12**(11), 1399–1413 (2019)
42. Winter, C., Schmidt, T., Neumann, T., Kemper, A.: Meet me halfway: split maintenance of continuous views. Proc. VLDB Endow. **13**(11), 2620–2633 (2020)
43. Zhu, C., Zhu, Q., Zuzarte, C., Ma, W.: Developing a dynamic materialized view index for efficiently discovering usable views for progressive queries. J. Inf. Process Syst. **9**(4), 511–537 (2013)
44. Zhou, Y., Salehi, A., Aberer, K.: Scalable delivery of stream query results. Proc. VLDB Endow. **2**(1), 49–60 (2009)
45. Lv, Y., Jin, P.: RotaryDS: fast storage for massive data streams via a rotation storage model. In: CIKM, pp. 3305–3308. ACM (2020)
46. Fazzinga, B., Flesca, S., Furfaro, F., Parisi, F.: Interpreting RFID tracking data for simultaneously moving objects: an offline sampling-based approach. Expert Syst. Appl. **152**,(2020)

47. Baveja, A., Chavan, A., Nikiforov, A., Srinivasan, A., Xu, P.: Improved Bounds in Stochastic Matching and Optimization. Algorithmica **80**(11), 3225–3252 (2018)
48. Guo, T., Zhu, X., Wang, Y., Chen, F.: Discriminative sample generation for deep imbalanced learning. In: IJCAI, pp. 2406–2412. ijcai.org (2019)
49. Wu, J., Cai, Z., Chen, X., Ao, S.: Active AODE learning based on a novel sampling strategy and its application. Int. J. Comput. Appl. Technol. **47**(4), 326–333 (2013)
50. Zhang, M., Li, H., Pan, S., Liu, T., Su, SW.: One-shot neural architecture search via novelty driven sampling. In: IJCAI, pp. 3188–3194. ijcai.org (2020)
51. Schüle, M.E., Lang, H., Springer, M., Kemper, A., Neumann, T., Günnemann, S.: In-database machine learning with SQL on GPUs. In: SSDBM, pp. 25–36. ACM (2021)
52. Schüle, M.E., Huber, J., Kemper, A., Neumann, T.: Freedom for the SQL-lambda: just-in-time-compiling user-injected functions in PostgreSQL. In: SSDBM, pp. 6:1–6:12. ACM (2020)
53. Schüle, M.E., Vorona, D., Passing, L., Lang, H., Kemper, A., Günnemann, S., et al.: The power of SQL lambda functions. In: EDBT, pp. 534–537. OpenProceedings.org (2019)
54. Schüle, M.E., Simonis, F., Heyenbrock, T., Kemper, A., Günnemann, S., Neumann, T.: In-database machine learning: gradient descent and tensor algebra for main memory database systems. In: BTW. vol. P-289 of LNI, pp. 247–266. Gesellschaft für Informatik, Bonn (2019)
55. Kunft, A., Katsifodimos, A., Schelter, S., Breß, S., Rabl, T., Markl, V.: An intermediate representation for optimizing machine learning pipelines. Proc. VLDB Endow. **12**(11), 1553–1567 (2019)
56. Boehm, M., Antonov, I., Baunsgaard, S., Dokter, M., Ginthör, R., Innerebner, K., et al.: SystemDS: a declarative machine learning system for the end-to-end data science lifecycle. In: CIDR. www.cidrdb.org (2020)
57. Schüle, M.E., Bungeroth, M., Vorona, D., Kemper, A., Günnemann, S., Neumann, T.: ML2SQL—compiling a declarative machine learning language to SQL and Python. In: EDBT, pp. 562–565. OpenProceedings.org (2019)
58. Schüle, M.E., Bungeroth, M., Kemper, A., Günnemann, S., Neumann, T.: MLearn: a declarative machine learning language for database systems. In: DEEM@SIGMOD, pp. 7:1–7:4. ACM (2019)
59. Makrynioti, N., Vasiloglou, N., Pasalic, E., Vassalos, V.: Modelling machine learning algorithms on relational data with Datalog. In: DEEM@SIGMOD, pp. 5:1–5:4. ACM (2018)
60. Jankov, D., Luo, S., Yuan, B., Cai, Z., Zou, J., Jermaine, C., et al.: Declarative recursive computation on an RDBMS. Proc. VLDB Endow. **12**(7), 822–835 (2019)
61. Schleich, M., Olteanu, D., Khamis, M.A., Ngo, H.Q., Nguyen, X.: A layered aggregate engine for analytics workloads. In: SIGMOD conference, pp. 1642–1659. ACM (2019)
62. Duta, C., Hirn, D., Grust, T.: Compiling PL/SQL away. In: CIDR. www.cidrdb.org (2020)
63. Schüle, M.E., Schmeißer, J., Blum, T., Kemper, A., Neumann, T.: TardisDB: extending SQL to support versioning. In: SIGMOD conference, pp. 2775–2778. ACM (2021)
64. Schüle, M.E., Karnowski, L., Schmeißer, J., Kleiner, B., Kemper, A., Neumann, T.: Versioning in main-memory database systems: from MusaeusDB to TardisDB. In: SSDBM, pp. 169–180. ACM (2019)
65. Lustig, D., Sahasrabuddhe, S., Giroux, O.: A formal analysis of the NVIDIA PTX memory consistency model. In: ASPLOS, pp. 257–270. ACM (2019)
66. Liang, T., Li, H., Chen, B.: A distributed PTX compilation and execution system on hybrid CPU/GPU clusters. In: ICS. Frontiers in Artificial Intelligence and Applications, vol. 274, pp. 1355–1364. IOS Press (2014)
67. Passing, L., Then, M., Hubig, N., Lang, H., Schreier, M., Günnemann, S., et al.: SQL- and operator-centric data analytics in relational main-memory databases. In: EDBT, pp. 84–95. OpenProceedings.org (2017)
68. Murray, I.: Machine learning and pattern recognition (MLPR): backpropagation of derivatives. https://www.inf.ed.ac.uk/teaching/courses/mlpr/2017/notes/w5a_backprop.pdf
69. Karanasos, K., Interlandi, M., Psallidas, F., Sen, R., Park, K., Popivanov, I., et al.: Extending relational query processing with ML inference. In: CIDR. www.cidrdb.org (2020)
70. Recht, B., Ré, C., Wright, S.J., Niu, F.: Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In: NIPS, pp. 693–701 (2011)
71. Lattner, C, Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88. IEEE Computer Society (2004)
72. Strassen, V.: Gaussian elimination is not optimal. Numerische mathematik. **13**(4), 354–356 (1969)
73. Fisher, R.A.: The use of multiple measurements in taxonomic problems. Ann Eugenic. **7**(2), 179–188 (1936)

74. Coleman, C., Kang, D., Narayanan, D., Nardi, L., Zhao, T., Zhang, J., et al.: Analysis of DAWN-Bench, a time-to-accuracy machine learning performance benchmark. ACM SIGOPS Oper. Syst. Rev. **53**(1), 14–25 (2019)
75. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-MNIST: a novel image dataset for benchmarking machine learning algorithms

Authors and Affiliations

Maximilian E. Schüle[1] · Harald Lang[1] · Maximilian Springer[1] · Alfons Kemper[1] · Thomas Neumann[1] · Stephan Günnemann[1]

Harald Lang
harald.lang@tum.de

Maximilian Springer
max.springer@tum.de

Alfons Kemper
kemper@in.tum.de

Thomas Neumann
neumann@in.tum.de

Stephan Günnemann
guennemann@in.tum.de

[1] TUM, Boltzmannstr. 3, 85748 Garching, Bavaria, Germany