In-Database Machine Learning with SQL on GPUs

Maximilian E. Schüle
m.schuele@tum.de

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

Technical University of Munich

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

In-Database Machine Learning, Automatic Differentiation, GPU

ACM Reference Format:
Maximilian E. Schüle, Harald Lang, Maximilian Springer, Alfons Kemper, Thomas Neumann, and Stephan Günnemann. 2021. In-Database Machine Learning with SQL on GPUs. In *33rd International Conference on Scientific and Statistical Database Management (SSDBM 2021), July 6–7, 2021, Tampa, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468791.3468840

1 INTRODUCTION

Typically, steps of machine learning pipelines—that consist of data preprocessing [6, 17], model training/validation [9] and finally its deployment on unlabelled data [64]—are embedded in Python scripts that call up specialised tools such as 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.

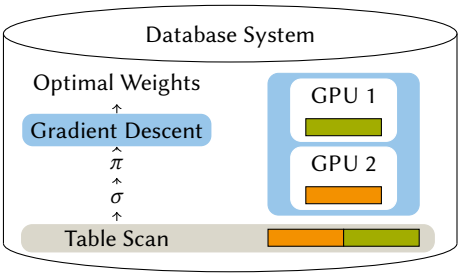


Figure 1: In-database machine learning: gradient descent with GPU support, embedded in a query plan.

Integrating machine learning pipelines into database systems is a promising approach for data-driven applications [1, 15, 19, 57, 62]. 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 [13, 30, 41, 54, 65] and operators of an extended relational algebra. This brings the model close to the data source [58] with SQL [2] as the only query language. As modern HTAP main-memory database systems such as SAP HANA [38], HyPer [20, 27, 39, 46] and Umbra [26, 40, 45] are designed for transactional and analytical workload, this allows the latest database state to be queried [24, 43]. But for continuous machine learning based on the latest tuples, only stand-alone solutions exist [3, 11] whose pipelines retrain weights for a model partially [31] when new input data is available.

In the last decade, research on database systems has focused on GPU co-processing to accelerate query engines [22, 55, 56, 63]. 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 [12, 33] and gradient descent [23, 36] profit from vectorised processing [5] available on GPUs [28]. 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.

SSDBM 2021, July 6–7, 2021, Tampa, FL, USA

ACM ISBN 978-1-4503-8413-1/21/07…$15.00
https://doi.org/10.1145/3468791.3468840

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 [7] and continuous views [59]. A continuous view [35, 67, 68] updates precomputed aggregates on incoming input. This kind of view—combined with sampling [4, 16, 18, 60, 66]—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. Based on automatic differentiation, we will proceed with an operator for gradient descent to be able to off-load work to GPUs (see Figure 1). In particular, this work's contributions are

- machine learning pipelines expressed in pure SQL,
- automatic differentiation in SQL that uses a lambda function 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 (Section 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 (Section 3). During code-generation, an operator for automatic differentiation compiles the gradient from a lambda expression (Section 4). 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 (Section 5). We will evaluate CPU and GPU-only approaches in terms of performance and accuracy using a NUMA-server cluster with multiple GPUs (Section 6).

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 [29] 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 [11]. 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 [8]. The integration of machine learning pipelines inside database systems would allow end-to-end machine learning [48, 49] and would inherit benefits such as query optimisation and recovery by design [37]. The work of Jankov et al. [21] 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 [44] learns models on pre-aggregated data.

GPU Acceleration. *Crossbow* [28] 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 [14, 51, 53], also offers just-in-time compilation for NVIDIA's Compute Unified Device Architecture (CUDA) [34]. Code compilation for GPU allows compilation for heterogeneous CPU-GPU clusters [32] as LLVM addresses multiple target architectures as used in this study.

3 IN-DATABASE GRADIENT DESCENT

This section first introduces mini-batch gradient descent, before describing a complete machine learning pipeline in SQL.

3.1 Mini-Batch Gradient Descent

Optimisation methods such as gradient descent try to find the best parameters $\vec{w}_\infty$ of a *model function* $m_{\vec{w}}(\vec{x})$, e.g., a linear function that approximates a given label y. A *loss function* $l_{X,y}(\vec{w})$ measures the deviation (*residual*) between all approximated values $m_{\vec{w}}(X)$ and the given labels $\vec{y}$, for example, mean squared error:

$$m_{\vec{w}}(\vec{x}) = \sum_{i \in [\,|\vec{w}|]} x_i \cdot w_i \approx y, \tag{1}$$

$$l_{X,\vec{y}}(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} l_{\vec{x}_i, y_i}(\vec{w}) = \frac{1}{n} \sum_{i=1}^{n} (m_{\vec{w}}(\vec{x}_i) - y_i)^2. \tag{2}$$

To minimise $l_{X,y}(\vec{w})$, gradient descent updates the weights per iteration by subtracting the loss function's gradient times the learning rate γ. Batch gradient descent considers all tuples per iteration and averages the loss:

$$\vec{w}_{t+1} = \vec{w}_t - \gamma \nabla l_{X,\vec{y}}(\vec{w}_t), \tag{3}$$

$$\vec{w}_\infty \approx \lim_{t \to \infty} \vec{w}_t. \tag{4}$$

Smaller batch sizes, mini-batches, are mandatory when the entire input does not fit into GPU memory and allows parallelism later on. Therefore, we consider mini-batch gradient descent, where we have to split our input data set X into disjoint mini-batches $X = X_0 \uplus \cdots \uplus X_o$.

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 Figure 2) proposed by Derakhshan et. al. [11] in SQL:

- 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

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

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, and 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 1 shows the resulting SQL queries using a taxi data set 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, which we finally use to compute the loss. 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.

```
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;
copy taxidata from './taxisample.csv' delimiter ',';
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));
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);
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(2*hour*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
  a2-0.001*avg(2*day*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
  a3-0.001*avg(2*distance*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
  a4-0.001*avg(2*bearing*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration)),
  b -0.001*avg(2*(a1*hour+a2*day+a3*distance+a4*bearing+b-duration))
from gd, (select * from normalised tablesample reservoir (10))
where id<=50 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 1: Machine learning pipeline in SQL.

3.3 Neural Networks in SQL

Expressing neural networks in SQL-92 is possible having one relation for the weights and one for the input tuples (Listing 2). 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.

```
create table w(id int, i int, j int, val float); insert into w ...
create function w_ij(id int, i int, j int) returns float language 'sql' strict
       as $$ select val from w where w.i=i and w.j=j and w.id=id $$;
create function sig(i float) returns float language 'sql' as $$ select 1.0/(1.0+
       exp(-i));$$;
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 input i;
```

Listing 2: Neural network in SQL-92.

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

In Listing 3, 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.

```
create view wm as (select id, array_agg(name) from (select id, i, array_agg(val)
        as name from w group by id,i) j group by id);
create function sig (x float[]) returns float[] language 'sql' as $$ select
       array_agg(s) from (select sig(unnest) as s from unnest(x)) tmp; $$;
select sig(array[[a,b]]*wm.val) from input, wm where wm.id=0;
```

Listing 3: Neural network with an array data type.

4 DATABASE OPERATORS FOR ML

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.

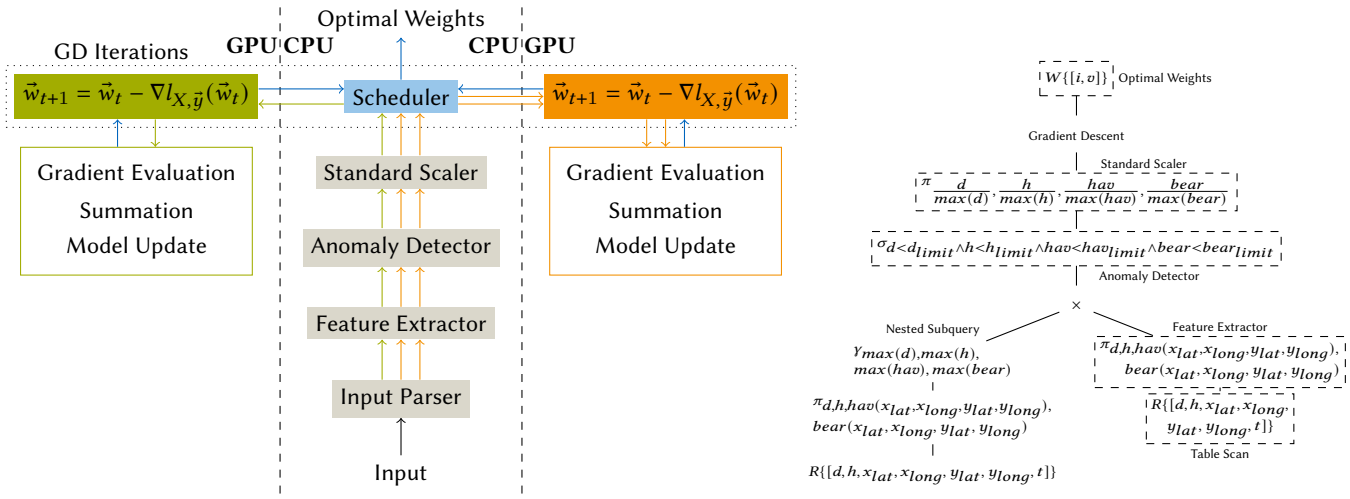
4.1 Code Generation

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 [39], 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 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

As Umbra compiles arithmetic expressions to machine code as well, it is perfectly suited for automatic differentiation (see Figure 3, Algorithm 1). Similar to how an arithmetic SQL expression is compiled during code generation, we created a function that can be used to



(a) Components of a machine learning pipeline for chunked processing. **(b) Operator plan inside of a database system.**

Figure 2: Machine learning pipelines for database systems: (a) focuses on the components of a whole machine learning pipeline: chunked input will be processed independently on GPUs. After every iteration, the weights (blue) are synchronised. (b) shows the corresponding operator plan with linear regression on the New York taxi data set in relational algebra: a projection extracts the features as haversine (*hav*) distance or bearing (*bear*), anomalies are deleted using predefined thresholds (denoted as *limit*).

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.

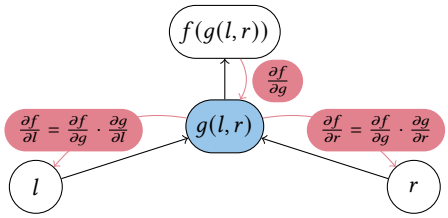


Figure 3: Reverse mode automatic differentiation: First, the function $f(g(l,r))$ gets evaluated, then each partial derivative is computed in reverse order. Each partial derivative is the product of the parent one (or 1 for the top most node) and the derived function with its original arguments as input. Each arrow represents one cached computation.

To specify the expression, we integrated lambda functions as introduced in HyPer [50, 52] into Umbra. Lambda functions are used to inject user-defined SQL expressions into table operators. They consist of arguments to define the column names (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 x, 3 y, 10 a, 10 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 4: Automatic differentiation within SQL.

Algorithm 1 Automatic Differentiation

```
function DERIVE(z, z')
    if z = x + y or z = x − y then DERIVE(x,z') DERIVE(y,z')
    else if z = x · y then DERIVE(x,z' · y) DERIVE(y,z' · x)
    else if z = x/y then DERIVE(x, z'/y) DERIVE(y, (−z'·x)/y^2)
    else if z = x^y then
        DERIVE(x,z' · y · x^(y−1)) DERIVE(y,z' · x^y ln(x))
    else if z = log_y(x) then
        DERIVE(x, z'/(x·ln(y))) DERIVE(y, (−z'·ln(x))/(y·ln^2(y)))
    else if z = sin(x) then DERIVE(x,z' · cos(x))
    else if z = cos(x) then DERIVE(x,−z' · sin(x))
    else if z = e^x then DERIVE(x,z' · e^x)
    else if isVariable(z) then ∂/∂z ← ∂/∂z + z'
```

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 4). We can use the operator within a recursive table to perform gradient descent (see Listing 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.

```
create table data (x float, y float); insert into data ...
with recursive gd (id, a, b) as (select 1,1::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;
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 5: Gradient descent using recursive tables: manually derived and using automatic differentiation.

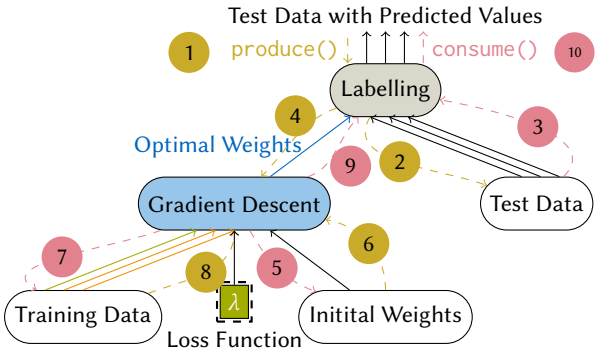


Figure 4: Operator plan inside of a database system with one operator for training and a query for predicting labels. Dashed lines illustrate code generation, solid lines compiled code. The `gradient descent` operator materialises input from parallel pipelines within local threads, performs iterations and returns the optimal weights.

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 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 4 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.

```
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:      GENERATE(w ← Σ_{l∈localthreads} l.w / |localthreads|)
9:      GENERATE(whileEnd)
10:     CONSUME(parent,w)
11: function UPDATEWEIGHTS
12:     GENERATE(w ← w − (1/n) Σ_{i∈[n]} ∇l_w(x_i, y_i))
13: function CONSUME
14:     GENERATE(localthread.store(x, y))
```

5 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 [25]. 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.

5.1 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 Figure 5) 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

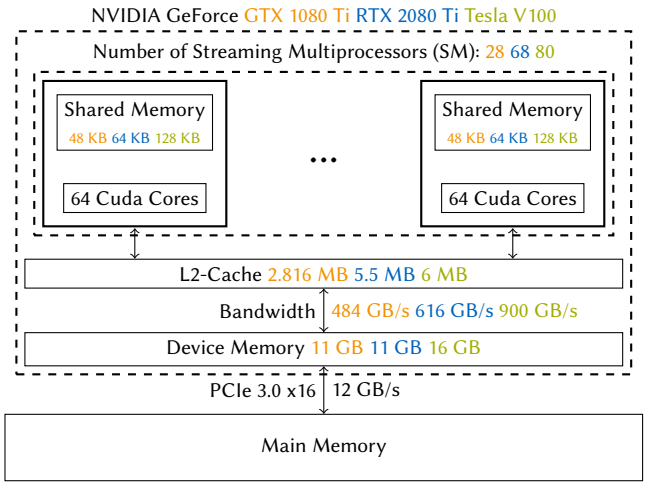


Figure 5: Simplified GPU architecture for NVIDIA GeForce GTX 1080 Ti (orange), RTX 2080 Ti (blue) and Tesla V100 (green): Each GPU transfers data via PCIe x16 from main-memory to its global/device memory. Multiple CUDA cores sharing the L1 cache (shared memory) are grouped to one streaming multiprocessor, its number is GPU specific. In-between lies the L2 cache.

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. 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.

5.1.1 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).

5.1.2 Neural Network. Our initial approach was to adapt the gradient descent kernel for linear regression to train a neural network

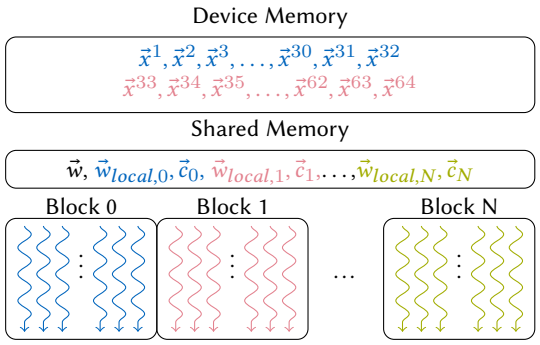


Figure 6: Multiple learners per GPU: Each block corresponds to one learner, each learner maintains local weights $\vec{w}_{local}$ and the difference $\vec{c}_{local}$ to the global weights $\vec{w}$. Each input tuple is stored in device memory and is scheduled to one GPU thread.

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.

5.1.3 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* [25] 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 lightweight 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.

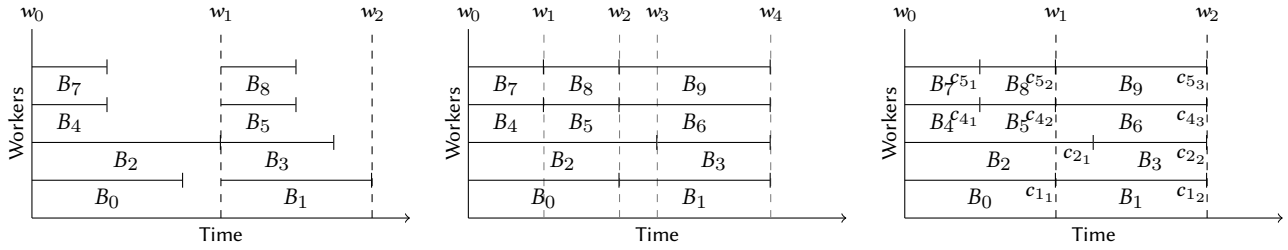
In our implementation (see Figure 6), each learner corresponds to one GPU block. We can set the block size adaptively, by which the number of learners results. Consequently, one learner works on batch sizes of at least one warp, that is the minimum block size with 32 threads, or multiple warps. Hence, 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.

5.2 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.

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



(a) Synchronised threads **(b) Worker threads (global updates)** **(c) Worker threads (local models)**

Figure 7: 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.

The naive implementation uses a constant fraction of its input data for validation and the rest for training. The training data set 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.

5.2.1 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 3 shows this gradient descent function, taking as input the data set X, labels $\vec{y}$, a learning rate γ, the batch size n and the hyper-parameter ctr_{max}. 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.

When synchronising a global model after each iteration, workers who may have finished their mini-batches earlier, are idle and waiting for input (see Figure 7a). 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 Figure 7c) or update global weights centrally (see Figure 7b).

5.2.2 Worker Threads with Global Updates (Bulk Synchronous Parallel). In Algorithm 4, 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 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 [42]. Otherwise, the critical section—gradient calculation and weights update (line 5)—has to be locked, which would result in a single-threaded process as in Algorithm 3.

Algorithm 3 Synchronised.

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

Algorithm 4 Worker threads (global updates).

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

5.2.3 Worker Threads with Local Models (Model Average). To overcome race conditions when updating the weights, we adapt local models known from Crossbow [25] 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 $\vec{c}_t$ 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 5 shows its adaption for worker threads. The main thread manages the update of the global model (line 10) that is the summation of all corrections. The critical section now consists of the computation of the corrections (lines 6-8) only, so the gradient can be computed on multiple units in parallel.

Algorithm 5 Worker threads (local models).

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

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

6 EVALUATION

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. We benchmark linear regression with synthetic data and the New York taxi data set, and a feed-forward neural network with a single hidden layer for image recognition (see Table 1). We take 2.65 GiB of the New York taxi data set[3] (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).

6.1 Automatic Differentiation

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. Figure 8 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 (Figure 9). Furthermore, we observe the approach using recursive tables computes aggregations in parallel, which accelerates computation on many input tuples with each additional thread.

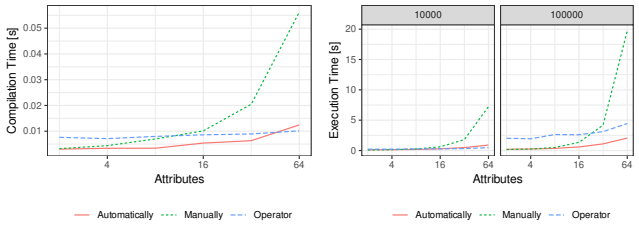


Figure 8: Compilation and execution time with increasing number of attributes (100 iterations, 10,000/100,000 tuples).

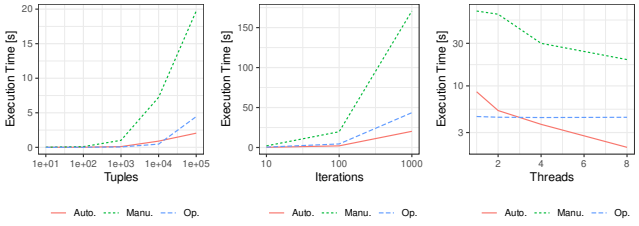


Figure 9: Execution time (64 attributes) with increasing number of tuples (100 iterations, 8 threads), iterations (100,000 tuples, 8 threads) or threads (100 iterations, 100,000 tuples).

6.2 Linear Regression

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:

[3] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

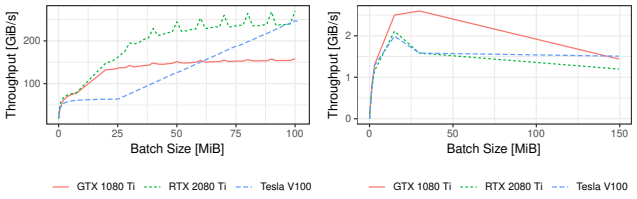
(1) *Throughput* measures the size of processed tuples per time. It includes tuples used for training as well as for validation.
(2) *Time-to-loss*, similarly to *time-to-accuracy* [10] for classification problems, measures the minimal loss on the validation data set depending on the computation time.
(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.

	#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

Table 1: Training and validation data sets used with linear regression and a neural network respectively.

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.

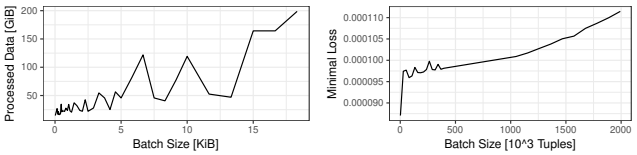
6.2.1 Throughput vs. Statistical Efficiency. To measure the throughput for linear regression on different hardware, we consider batch sizes of up to 100 MiB. We compare the performance of our kernels to that when stochastic gradient descent of the TensorFlow (version 1.15.0) library is called (see Figure 10). 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 Figure 10a) can exploit available hardware more effectively than the Python implementation (see Figure 10b). As the latter calls stochastic gradient descent, this excels on better hardware only when a larger model has to be trained.



(a) Our implementation. **(b) Using TensorFlow.**

Figure 10: Throughput of (a) the dedicated kernels and (b) using the TensorFlow library.

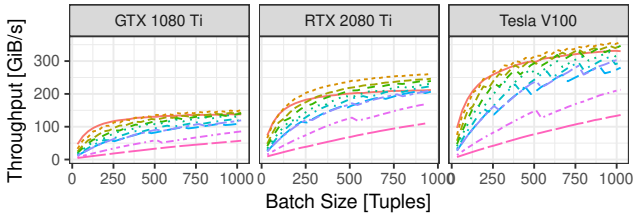
Nevertheless, training with large batch sizes does not imply statistical efficiency in terms of the volume of processed data that is needed for convergence (see Figure 11a) and the lowest reachable minimal loss (see Figure 11b). For that reason, to allow the highest throughput even for small batch sizes, we implement multiple learners per GPU.



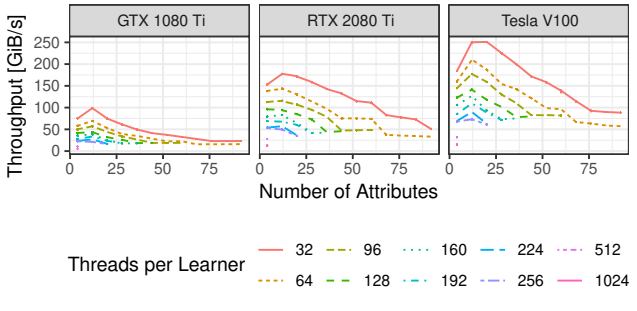
(a) Data to converge. **(b) Minimal loss.**

Figure 11: Statistical efficiency for linear regression: (a) volume of processed data needed to converge and (b) minimal reachable loss depending on the batch size.

6.2.2 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.



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



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

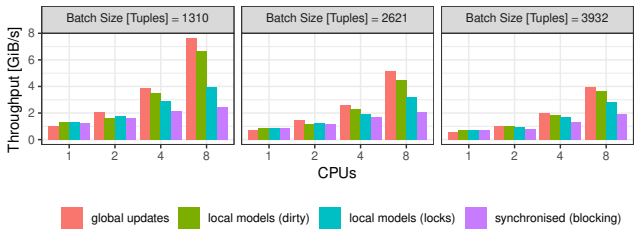
Figure 12: Throughput with multiple learners per GPU: A smaller number of threads per learner allows the maximum throughput even for small batch sizes, when small batches are processed in parallel.

The observation in Figure 12 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

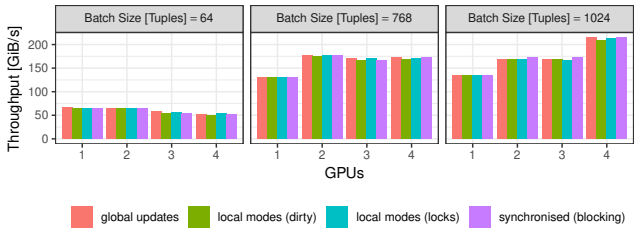
of the chosen number of threads, the throughput reaches a local maximum. Otherwise, the GPU is underutilised. These local maxima are visible as spikes in all curves except for 32 threads per block, as we increase the batch size by 32 tuples. Nevertheless, on all devices, the throughput soon converges at the possible maximum, which shows the efficiency of learners in the granularity of GPU warps.

6.2.3 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.

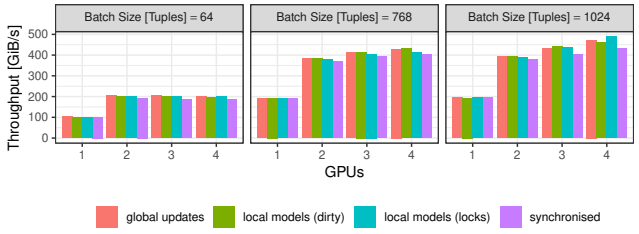
When parallelising on the CPU, each additional thread allows a linear speed-up when no synchronisation takes place (see Figure 13a). Maintaining local models costs additional computation time, which results in a lower throughput. Obviously, locks slow down the speed up, and blocking threads cause underutilisation.



(a) CPU (Intel Xeon Gold 5120)



(b) NVIDIA GeForce GTX 1080 Ti



(c) NVIDIA GeForce RTX 2080 Ti

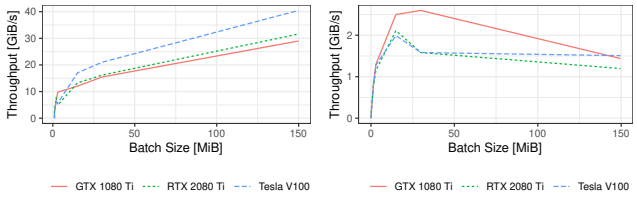
Figure 13: Scale-up for linear regression on (a) multiple CPUs, (b) multiple NVIDIA GeForce GTX 1080 Ti or (c) multiple NVIDIA GeForce GTX 2080 Ti.

Whereas parallelising for GPUs behaves differently (see Figure 13b/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.

6.3 Neural Network

To benchmark the feed-forward neural network, we perform image classification using the MNIST and Fashion-MNIST [61] data set. We train the neural network with one hidden layer of size 200 to recognise a written digit 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).

6.3.1 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 Figure 14b), our implementations 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 Figure 14a), but allows a higher throughput, the larger the batch size.

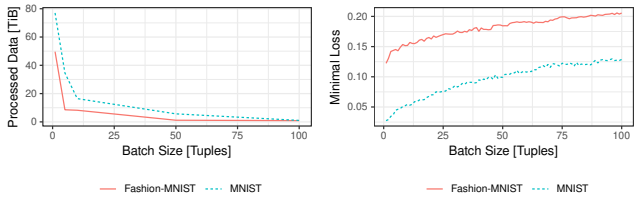


(a) Own implementation. **(b) Using Keras.**

Figure 14: Throughput of the implementation (a) using cuBLAS and (b) using Keras when training a neural network with the MNIST data set.

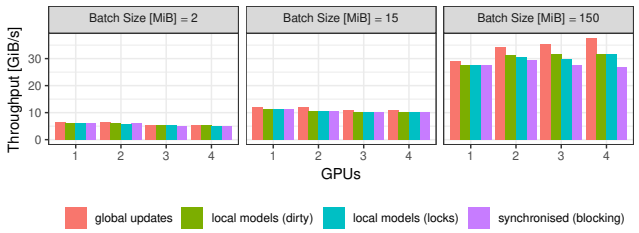
As is the case for linear regression, training models with small batch sizes results in a higher accuracy (see Figure 15b). This once again makes the case for multiple learners per single GPU. Nevertheless, the larger the chosen batch size is, the faster training iterations converge (see Figure 15a).

6.3.2 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 Figure 16). 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.

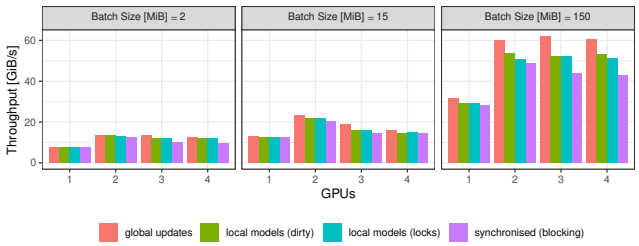


(a) Data to converge. **(b) Minimal loss.**

Figure 15: Statistical efficiency for the neural network: (a) volume of processed data needed to converge and (b) minimal reachable loss dependending on the batch size.



(a) NVIDIA GeForce GTX 1080 Ti



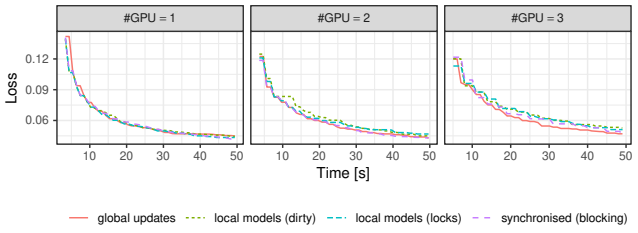
(b) NVIDIA GeForce RTX 2080 Ti

Figure 16: Scale-up for training a neural network (with the MNIST data set).

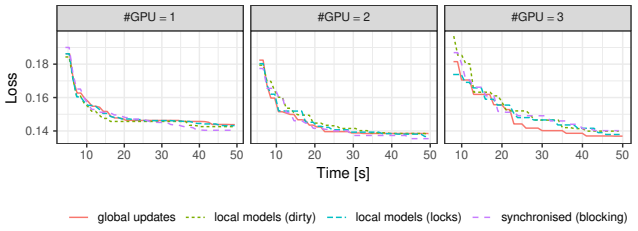
6.3.3 Time/Tuples-to-loss. Regarding the time to reach a certain accuracy (see Figure 17), all implementations perform similarly when running on a single worker. As the MNIST data set converges fast, adding a GPU device for computation has no significant impact. Whereas the Fashion-MNIST data set converges 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.

6.4 End-to-End Analysis

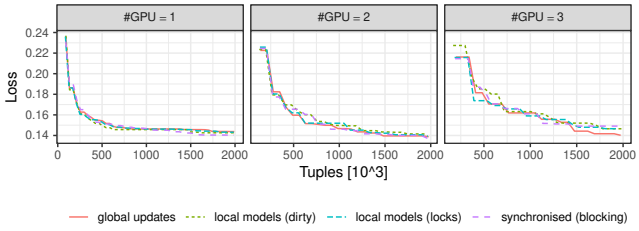
Figure 18 compares the time needed to train one epoch (New York taxi data: $13 \cdot 10^6$ tuples) within a complete machine learning pipeline in Python using Keras to a corresponding operator tree within the database system Umbra. The pipeline consists of data loading from CSV, feature extraction and normalisation either with NumPy or SQL-92 queries, and training. We observe that much time



(a) Time-to-loss: MNIST data set



(b) Time-to-loss: Fashion-MNIST data set



(c) Tuples-to-loss: Fashion-MNIST data set

Figure 17: Time-to-loss when training the neural network for the (a) MNIST and (b) Fashion-MNIST data set with a batch size of 5 tuples (NVIDIA GeForce GTX 2080 Ti). For Fashion-MNIST, also tuples-to-time (c) is provided.

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.

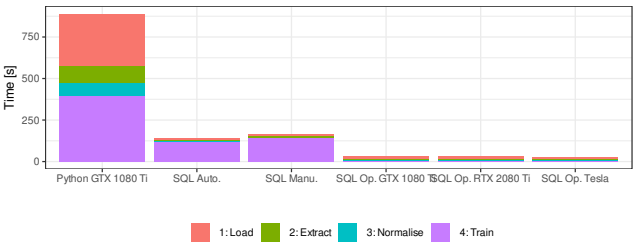


Figure 18: End-to-end analysis of a machine learning pipeline: linear regression (*NY taxi*, 64 tuples per batch).

7 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 gradient descent to off-load training to GPU units. Therefore, 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.

REFERENCES

[1] Andrej Andrejev, Kjell Orsborn, and Tore Risch. 2020. Strategies for array data retrieval from a relational back-end based on access patterns. *Computing* (2020).
[2] Natalia Arzamasova, Klemens Böhm, et al. 2020. On the Usefulness of SQL-Query-Similarity Measures to Find User Interests. *IEEE TKDE* 32, 10 (2020).
[3] Arian Bär et al. 2014. DBStream. In *IWCMC*. IEEE.
[4] Alok Baveja, Amit Chavan, et al. 2018. Improved Bounds in Stochastic Matching and Optimization. *Algorithmica* 80, 11 (2018), 3225–3252.
[5] Spiridon F. Beldianu and Sotirios G. Ziavras. 2011. On-chip Vector Coprocessor Sharing for Multicores. In *PDP*. IEEE, 431–438.
[6] Khalid Belhajjame. 2020. On Discovering Data Preparation Modules Using Examples. In *ICSOC (Lecture Notes in Computer Science, Vol. 12571)*. Springer, 56–65.
[7] Altan Birler. 2019. Scalable Reservoir Sampling on Many-Core CPUs. In *SIGMOD Conference*. ACM, 1817–1819.
[8] Matthias Boehm et al. 2020. SystemDS. In *CIDR*. www.cidrdb.org.
[9] Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. 2019. Exploring the capabilities of mobile devices in supporting deep learning. In *SEC*. ACM, 127–138.
[10] Cody Coleman et al. 2019. Analysis of DAWNBench, a Time-to-Accuracy Machine Learning Performance Benchmark. *ACM SIGOPS Oper. Syst. Rev.* 53, 1 (2019).
[11] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Tilmann Rabl, and Volker Markl. 2019. Continuous Deployment of Machine Learning Pipelines. In *EDBT*. 397–408.
[12] Oksana Dolmatova et al. 2020. A Relational Matrix Algebra and its Implementation in a Column Store. In *SIGMOD Conference*. ACM, 2573–2587.
[13] Bin Dong et al. 2017. ArrayUDF. In *HPDC*. ACM, 53–64.
[14] Christian Duta et al. 2020. Compiling PL/SQL Away. In *CIDR*. www.cidrdb.org.
[15] Mehrad Eslami, Yicheng Tu, Hadi Charkhgard, Zichen Xu, et al. 2019. PsiDB: A Framework for Batched Query Processing and Optimization. In *BigData*. IEEE.
[16] Bettina Fazzinga et al. 2020. Interpreting RFID tracking data for simultaneously moving objects. *Expert Syst. Appl.* 152 (2020), 113368.
[17] Maxim Filatov and Verena Kantere. 2016. PAW: A Platform for Analytics Workflows. In *EDBT*. 624–627.
[18] Ting Guo, Xingquan Zhu, Yang Wang, and Fang Chen. 2019. Discriminative Sample Generation for Deep Imbalanced Learning. In *IJCAI*. ijcai.org, 2406–2412.
[19] Ali Hadian, Ankit Kumar, and Thomas Heinis. 2020. Hands-off Model Integration in Spatial Index Structures. In *AIDB@VLDB*.
[20] Nina Hubig, Linnea Passing, et al. 2017. HyPerInsight. In *CIKM*. ACM, 2467–2470.
[21] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, et al. 2019. Declarative Recursive Computation on an RDBMS. *PVLDB* 12, 7 (2019), 822–835.
[22] Peng Jiang et al. 2020. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *PPoPP*. ACM, 376–388.
[23] Peng Jiang and Gagan Agrawal. 2019. Accelerating distributed stochastic gradient descent with adaptive periodic parameter averaging. In *PPoPP*. ACM, 403–404.
[24] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2016. Flash as cache extension for online transactional workloads. *VLDB J.* 25, 5 (2016), 673–694.
[25] Konstantinos Karanasos et al. 2020. Extending Relational Query Processing with ML Inference. In *CIDR*. www.cidrdb.org.
[26] Lukas Karnowski et al. 2021. Umbra as a Time Machine. In *BTW (LNI)*. GI.
[27] Alfons Kemper and Thomas Neumann. 2011. HyPer. In *ICDE*. IEEE, 195–206.
[28] Alexandros Kolioussis et al. 2019. Crossbow. *PVLDB* 12, 11 (2019), 1399–1413.
[29] Andreas Kunft et al. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *PVLDB* 12, 11 (2019), 1553–1567.
[30] Jeff LeFevre, Jagan Sankaranarayanan, et al. 2014. Opportunistic physical design for big data analytics. In *SIGMOD Conference*. ACM, 851–862.
[31] Zheng Li and Tingjian Ge. 2016. Stochastic Data Acquisition for Answering Queries as Time Goes by. *PVLDB* 10, 3 (2016), 277–288.
[32] Tyng-Yeu Liang et al. 2014. A Distributed PTX Compilation and Execution System on Hybrid CPU/GPU Clusters. In *ICS (FAIA, Vol. 274)*. IOS Press, 1355–1364.
[33] Shangyu Luo et al. 2017. Scalable Linear Algebra on a Relational Database System. In *ICDE*. IEEE, 523–534.
[34] Daniel Lustig et al. 2019. A Formal Analysis of the NVIDIA PTX Memory Consistency Model. In *ASPLOS*. ACM, 257–270.
[35] Yanqi Lv and Peiquan Jin. 2020. RotaryDS: Fast Storage for Massive Data Streams via a Rotation Storage Model. In *CIKM*. ACM, 3305–3308.
[36] Yujing Ma, Florin Rusu, and Martin Torres. 2019. Stochastic Gradient Descent on Modern Hardware: Multi-core CPU or GPU?. In *IPDPS*. IEEE, 1063–1072.
[37] Nantia Makrynioti et al. 2018. Modelling Machine Learning Algorithms on Relational Data with Datalog. In *DEEM@SIGMOD*. ACM, 5:1–5:4.
[38] Norman May et al. 2017. SAP HANA. In *BTW (LNI)*. GI.
[39] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550.
[40] Thomas Neumann et al. 2020. Umbra. In *CIDR*. www.cidrdb.org.
[41] Shreya Prasad, Arash Fard, Vishrut Gupta, et al. 2015. Large-scale Predictive Analytics in Vertica. In *SIGMOD Conference*. ACM, 1657–1668.
[42] Benjamin Recht, Christopher Ré, Stephen J. Wright, et al. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 693–701.
[43] Mohammad Sadoghi et al. 2018. L-Store. In *EDBT*. 540–551.
[44] Maximilian Schleich et al. 2019. A Layered Aggregate Engine for Analytics Workloads. In *SIGMOD Conference*. ACM, 1642–1659.
[45] Josef Schmeißer et al. 2021. B^2-Tree. In *BTW (LNI)*. GI.
[46] Maximilian E. Schüle et al. 2017. Monopedia. *VLDB* 10, 12 (2017), 1921–1924.
[47] Maximilian E. Schüle et al. 2019. In-Database Machine Learning: Gradient Descent and Tensor Algebra for MMDBS. In *BTW (LNI)*. GI.
[48] Maximilian E. Schüle et al. 2019. ML2SQL. In *EDBT*. 562–565.
[49] Maximilian E. Schüle et al. 2019. MLearn. In *DEEM@SIGMOD*. ACM, 7:1–7:4.
[50] Maximilian E. Schüle et al. 2019. The Power of SQL Lambda Functions. In *EDBT*.
[51] Maximilian E. Schüle et al. 2019. Versioning in Main-Memory Database Systems: From MusaeusDB to TardisDB. In *SSDBM*. ACM, 169–180.
[52] Maximilian E. Schüle et al. 2020. Freedom for the SQL-Lambda: Just-in-Time-Compiling User-Injected Functions in PostgreSQL. In *SSDBM*. ACM, 6:1–6:12.
[53] Maximilian E. Schüle et al. 2021. TardisDB. In *SIGMOD*. ACM.
[54] Kurt Stockinger et al. 2019. Scalable architecture for Big Data financial analytics: user-defined functions vs. SQL. *J. Big Data* 6 (2019), 46.
[55] Alexander Terenin, Shawfeng Dong, et al. 2019. GPU-accelerated Gibbs sampling: a case study of the Horseshoe Probit model. *Stat. Comput.* 29, 2 (2019), 301–310.
[56] Yi-Cheng Tu, Anand Kumar, et al. 2013. Data management systems on GPUs: promises and challenges. In *SSDBM*. ACM, 33:1–33:4.
[57] Sebastián Villarroya and Peter Baumann. 2020. On the Integration of Machine Learning and Array Databases. In *ICDE*. IEEE, 1786–1789.
[58] James Wagner et al. 2020. DF-Toolkit. *PVLDB* 13, 12 (2020), 2845–2848.
[59] Christian Winter et al. 2020. Meet Me Halfway: Split Maintenance of Continuous Views. *PVLDB* 13, 11 (2020), 2620–2633.
[60] Jia Wu et al. 2013. Active AODE learning based on a novel sampling strategy and its application. *Int. J. Comput. Appl. Technol.* 47, 4 (2013), 326–333.
[61] Han Xiao et al. 2017. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms.* arXiv:cs.LG/1708.07747 [cs.LG]
[62] Ying Yang, Niccolò Meneghetti, et al. 2015. Lenses: An On-Demand Approach to ETL. *PVLDB* 8, 12 (2015), 1578–1589.
[63] Feng Yu and Jonathan M. Harbor. 2019. CSTAT+: A GPU-accelerated spatial pattern analysis algorithm. *Environ. Model. Softw.* 120 (2019).
[64] Mu Yuan, Lan Zhang, Xiang-Yang Li, and Hui Xiong. 2020. Comprehensive and Efficient Data Labeling via Adaptive Model Scheduling. In *ICDE*. IEEE, 1858–1861.
[65] Chao Zhang and Farouk Toumani. 2020. Sharing Computations for User-Defined Aggregate Functions. In *EDBT*. 241–252.
[66] Miao Zhang, Huiqi Li, Shirui Pan, et al. 2020. One-Shot Neural Architecture Search via Novelty Driven Sampling. In *IJCAI*. ijcai.org, 3188–3194.
[67] Yongluan Zhou, Ali Salehi, and Karl Aberer. 2009. Scalable Delivery of Stream Query Results. *PVLDB* 2, 1 (2009), 49–60.
[68] Chao Zhu et al. 2013. Developing a Dynamic Materialized View Index. *J. Inf. Process. Syst.* 9, 4 (2013), 511–537.