In-RDBMS Hardware Acceleration of Advanced Analytics

Divya Mahajan* Joon Kyung Kim* Jacob Sacks* Adel Ardalan† Arun Kumar‡ Hadi Esmaeilzadeh‡*
*Georgia Institute of Technology †University of Wisconsin-Madison ‡University of California, San Diego
{divya_mahajan,jkkim,jsacks}@gatech.edu adel@cs.wisc.edu {arunkk,hadi}@eng.ucsd.edu

ABSTRACT

The data revolution is fueled by advances in several areas, including databases, high-performance computer architecture, and machine learning. Although timely, there is a void of solutions that brings these disjoint directions together. This paper sets out to be the initial step towards such a union. The aim is to devise a solution for the in-**D**atabase **A**cceleration of Adva**n**ced **A**nalytics (DAnA). DAnA empowers database users to leap beyond traditional data summarization techniques and seamlessly utilize hardware-accelerated machine learning. Deploying specialized hardware, such as FPGAs, for in-database analytics currently requires hand-designing the hardware and manually routing the data. Instead, DAnA automatically maps a high-level specification of in-database analytics queries to the FPGA accelerator. The accelerator implementation is generated from a User Defined Function (UDF), expressed as part of a SQL query in a Python-embedded Domain-Specific Language (DSL). To realize efficient in-database integration, DAnA-generated accelerators contain a novel hardware structure, *Striders*, that directly interface with the buffer pool of the database. DAnA obtains the schema and page layout information from the database catalog to configure the *Striders*. In turn, *Striders* extract, cleanse, and process the training data tuples, which are consumed by a multi-threaded FPGA engine that executes the analytics algorithm. We integrated DAnA with PostgreSQL to generate hardware accelerators for a range of real-world and synthetic datasets running diverse ML algorithms. Results show that DAnA-enhanced PostgreSQL provides, on average, 11.3× end-to-end speedup for real datasets, with the maximum at 58.2×. Moreover, DAnA-enhanced PostgreSQL is 5.4× faster, on average, than the multi-threaded Apache MADLib running on Greenplum. DAnA provides these benefits while hiding the complexity of hardware design from data scientists and allowing them to express the algorithm in ≈30-60 lines of Python code.

1. INTRODUCTION

Relational Database Management Systems (RDBMSs) are the cornerstone of large-scale data management in almost all major enterprise settings. However, data-driven applications in such environments are increasingly migrating from simple SQL analytics towards advanced algorithms, especially machine learning (ML), that perform sophisticated predictive analytics over large datasets [1,2]. As illustrated in Figure 1, there are three concurrent and important, but hitherto disconnected, trends in this data systems landscape: (1) adoption of enterprise in-database analytics, (2) development of modern hardware acceleration platforms, and (3) programming paradigms which facilitate the use of analytics.

First, the database industry is investing in the integration of ML algorithms within RDBMSs, both on-premise and cloud-based [9, 10]. This allows enterprises to exploit ML without sacrificing the auxiliary benefits of RDBMSs, such as transparent scalability, access control, security, and integration with their business intelligence interfaces [11–15].x Second, the computer architecture community is extensively studying the integration of specialized hardware accelerators within the traditional compute stack for ML applications [6, 16–19]. Recent work at the intersection of databases and computer architecture has led to a growing interest in hardware acceleration for relational queries as well. This includes exploiting GPUs [20] and reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs), for relational operations [3, 4, 21–23]. Even cloud database service providers, like Amazon AWS [24], Microsoft Azure [25], and Google Cloud [26], are offering such high-performance specialized platforms. Third, the applicability and practicality of both in-database analytics and hardware acceleration hinge upon exposing a high-level interface to the user. Several research projects have investigated ways of simplifying in-database analytics [7, 8, 14, 27, 28] and hardware acceleration [5] for learning by focusing on a particular subset of algorithms which use vanilla or stochastic gradient descent (SGD). This triad of research areas are being studied in isolation and are evolving independently. Little work has explored the impact of moving analytics within databases on the design, implementation, and integration of hardware accelerators. Unification of these research directions can help mitigate the inefficiencies and reduced productivity of data scientists who can benefit from hardware acceleration for in-database analytics. Consider the following example.

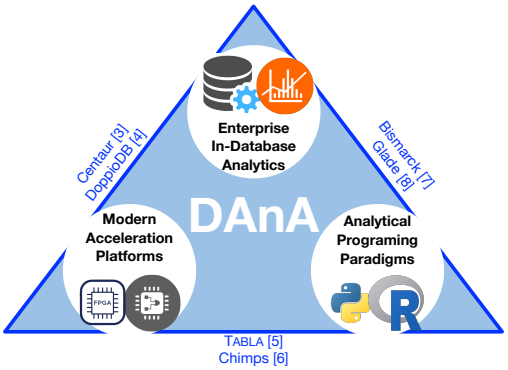


Figure 1: DAnA represents the fusion of three research directions, in contrast with prior works [3–8] that only merge two of the areas.

Example 1 *In an advertising company that uses the Amazon Web Services (AWS) Relational Data Service (RDS) to maintain a PostgreSQL database, a data scientist wants to forecast the hourly ad serving load over the next two weeks. To do so, she creates a regression model over a hundred features using historical records in their database. Due to large training times, she decides to accelerate her*

workload using FPGAs on Amazon EC2 F1 instances [24].Currently, this is highly non-trivial, and requires her to learn a hardware description language, such as Verilog or VHDL, to program the FPGAs and going through the painful process of hardware design, testing, and deployment individually for each ML algorithm. Recent research has developed tools to simplify FPGA acceleration for ML algorithms [16,17,29], however, do not interface or provide in-RDBMS support. Therefore, she would have to manually extract, copy, and reformat her large dataset to utilize such solutions.

As such, this work devises DAnA, a *cohesive system that enables deep integration between FPGA acceleration and in-RDBMS execution of advanced analytics*, while exposing a high-level programming interface to data scientists from conventional languages (SQL and Python). Building such a system requires: (1) providing an intuitive programming abstraction to express the combination of ML algorithm and required data schemas; and (2) designing a hardware mechanism that transparently connects the FPGA accelerator to the database engine for direct access to the training data pages.

To address the first challenge, DAnA extends the User Defined Functions (UDF) with a Python-embedded Domain Specific Language (DSL) to express ML algorithms as an update rule while the associated SQL query specifies data's management and retrieval. This interface allows expressing in-database analytics using familiar practices, but requires a complete software stack that can translate this high-level specification to accelerated execution. Thus, DAnA contains a full-stack solution that breaks the (algorithm, data) pair into software execution on the RDBMS for data retrieval and hardware acceleration for running the analytics algorithm.

The cohesive fusion of RDBMS and FPGA addresses the second challenge to seamlessly connect both while avoiding the inefficiencies of conventional Von-Neumann CPUs for data handoff. To achieve this, DAnA introduces a novel hardware component, *Striders*, that walk through the RDBMS buffer pools and directly feed data to the analytics accelerators. By circumventing the CPU, DAnA can alleviate the cost of data transfer through the traditional CPU memory subsystem. As the database page organization and tuple length varies across algorithms and training datasets, it is imperative to ensure the programmability of *Strider*. Thus, we devise an Instruction Set Architecture (ISA) that can program the *Strider* to cater for the variability in database page layout. *Striders* are also designed to ensure *multithreaded acceleration* of the learning algorithm to amortize the cost of data accesses across multiple concurrent threads. Besides *Striders*, DAnA automatically generates the architecture of these accelerator threads, called the execution engine, that selectively combines a Multi-Instruction Multi-Data (MIMD) execution model with novel Single-Instruction Multiple Data (SIMD) semantics to reduce the instruction footprint. While generating this MIMD-SIMD accelerator, DAnA tailors its architecture to the ML algorithm's computation patterns, RDBMS page format, and FPGA resources. By tackling the aforementioned challenges, this paper makes the following technical contributions:

- Merges three mutually disjoint research areas to enable transparent and efficient hardware acceleration for in-RDBMS analytics. This union allows DAnA to provide a means for data scientists with no expertise in hardware design to harness acceleration while retaining familiar programming environments without manual data retrieval and extraction.
- Exposes a high-level programming interface, which combines SQL UDFs with a Python DSL, to specify the data and computation jointly. This unified abstraction is backed by an extensive compilation workflow that automatically transforms the specification to an accelerated execution.
- Enables the deep integration between an FPGA and the RDBMS engine through *Striders*, which are novel on-chip hardware interfaces. *Striders* bypass the CPU to directly access the training data from the buffer pool, transfer this data to the FPGA, and unpack the feature vectors and labels.
- Offers a novel execution model that fuses thread-level and data-level parallelism to execute the learning algorithm computations. The model is exposes a domain specific instruction set architecture that offers automation while providing efficiency.

DAnA is prototyped with PostgreSQL to automatically accelerate the execution of several popular ML algorithms.Through a comprehensive experimental evaluation using both real-world and synthetic datasets, we compare DAnA against the popular in-RDBMS ML toolkit, Apache MADlib [14], on both PostgreSQL and its parallel counterpart, Greenplum. Using Arria-10 as the FPGA platform, we observe that DAnA generated accelerators provide, on average, 11.3× and 5.4× end-to-end runtime speedups over PostgreSQL and Greenplum running MADlib, respectively. An average 2.5× of the speedup benefits are obtained through *Striders*, which effectively bypass the CPU and its memory subsystem overhead.

2. BACKGROUND

Before delving into the details of DAnA, this section provides a background on FPGAs. Next, it discusses the properties of ML algorithms targeted by DAnA to provide its holistic framework.

2.1 Field Programmable Gate Arrays

As machine learning is a constantly evolving field, FPGAs offer a potent solution to hardware acceleration as they can readily support new learning-based algorithms. An FPGA is an integrated circuit that consists of reprogrammable look up tables (LUTs), which mimic digital logic by storing a corresponding design configuration in the Static Random Access Memories (SRAMs). Figure 2 illustrates a small portion of the reprogrammable logic blocks and specialized hardware available on an FPGA chip. Contemporary FPGAs also include hardened memories called Block RAMs (BRAMs) and Digital Signal Processing (DSP) cores, as on-chip local memory accesses and arithmetic units are fundamental to any application. Although FPGAs are energy efficient and reconfigurable, programming them is still a challenge and requires long design cycles, even for experts.

Hardware Description Languages (HDLs), such as Verilog and VHDL, provide a means to specify hardware logic at the register-transfer level (RTL). The designer is thus expected to provide synthesizable code, i.e., a circuit description valid for the given FPGA. A typical hardware design flow demands iterative refinement of this description to verify its functional correctness. This verification process is a laborious task which involves rigorous simulations and cycle-by-cycle waveform generation to check if the logic conforms to the desired specification. As a verified design is not necessarily optimal, the programmer generally has to optimize the design manually at the register or gate level. To reduce the complexity of programming with HDLs, companies offer proprietary high-level synthesis tools that convert C/C++ programs to Verilog or VHDL. However, the designer is still expected to go through the verification and optimization process. DAnA aims to exploit the reconfigurability and high performance of FPGAs for advanced analytics without requiring the user to go through this painful design flow process. Instead, DAnA provides a parametric architecture (∼15,000 lines of Verilog code) designed by hardware experts, which can be tailored to the needs of the ML algorithm and target FPGA. DAnA only requires the analyst to provide the high-level algorithm specification via a Python-embedded DSL. In our experience, many applications, such as regression/classification models

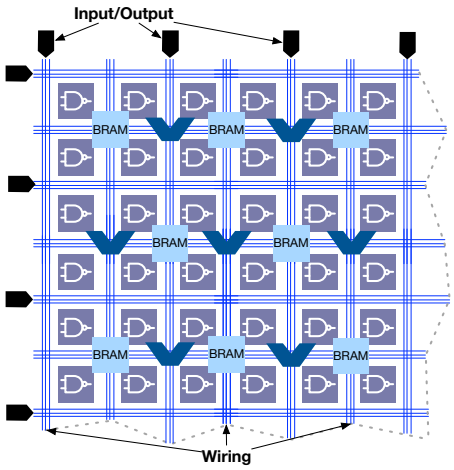


Figure 2: Illustration of a small corner of an FPGA chip. Modern FPGAs comprise a sea of binary lookup tables, augmented with hardened Block RAM and DSP slices to offer higher frequencies.

and collaborative filtering, can easily be expressed in ∼30-60 lines of DSL code. Even though the number of lines of code is not the most pertinent measure for comparison with the FPGA design process, it provides a tangible intuition for implementation complexity. The parametric architecture is automatically configured by a domain-specific compiler and hardware generator according to the analyst-provided application. Next section discusses the application domain of learning algorithms targeted by DAnA.

2.2 Iterative Optimization and Update Rules

A wide range of supervised Machine Learning (ML) training algorithms go through a cyclic process that demands constant iteration, tuning, and improvement. These algorithms use data mining techniques and optimizers that iteratively minimize a loss function - distinct for each learning algorithm - by using one tuple (input-output pair) at a time to generate updates for the learning model. Each ML algorithm has a specific loss function that mathematically captures the measure of the learning model's error. Improving the model corresponds to minimizing this loss function using an *update rule*, which is applied repeatedly over the training model, one training data tuple (input-output pair) at a time, until convergence.

An example. In supervised machine learning, given a set of N pairs of $\{(x_1, y_1^*, ..., x_N, y_N^*)\}$ constituting the training data, ultimate goal is to find a hypothesis function $h(w^{(t)}, x)$ that can accurately map $x \rightarrow y$. In the equation below, $l(w^{(t)}, x_i, y_i^*)$ is the loss function, which signifies the error between the output y^* and the predicted output estimated by a hypothesis function $h(w^{(t)}, x)$ for input x. This equation specifies an update rule in its entirety.

$$w^{(t+1)} = w^{(t)} - \mu \times \frac{\partial (l(w^{(t)}, x_i, y_i^*))}{\partial w^{(t)}} \quad (1)$$

For each (x, y^*) pair, the goal is to a find a model (w) that minimizes the loss function $l(w^{(t)}, x_i, y_i)$ using an iterative update rule. While the hypothesis and loss function vary greatly across different ML algorithms, the optimization algorithm which iteratively minimizes the loss function remains fixed. As such, the two required components are the hypothesis function $y = h(w, x)$, which defines the machine learning algorithm, and an optimization algorithm, which iteratively applies the update rule.

Amortizing the cost of data accesses by parallelizing the optimization. In Equation (1), a single (x_i, y_i^*) tuple is used to update the model. However, it is feasible to use a batch of tuples and compute multiple updates independently when the optimizer supports combining partial updates with a mathematical operator [30–36]. This property provides a unique opportunity for DAnA to rapidly consume data pages brought on-chip by *Striders* while efficiently utilizing the large, ever-growing amounts of compute resources available on the FPGAs through simultaneous multi-threaded acceleration. Examples of commonly used iterative optimization algorithms that support parallel iterations are variants of gradient descent methods, which are applied across a diverse range of ML models. DAnA is equipped to accelerate the training phase of any hypothesis and objective function that can be minimized using such iterative optimization. Thus, the user simply provides the update via the DAnA DSL described in §4.1.

3. DAnA OVERVIEW

This section discusses the insights and workflow that enable DAnA's integration of the RDBMS engine and FPGA accelerator.

3.1 Insights Driving DAnA

Database and hardware interface considerations. To obtain large benefits from hardware acceleration, the overheads of a traditional Von-Neumann architecture and memory subsystem are to be avoided. Moreover, data accesses from the buffer pool need to be at large enough granularities to efficiently utilize the FPGA bandwidth. DAnA satisfies these criterion through *Striders*, its database-aware reconfigurable memory interface, as discussed in § 5.1.

Algorithmic considerations. The training data retrieved from the buffer pool and stored on-chip must be consumed promptly to avoid throttling the memory resources on the FPGA. DAnA achieves this by leveraging the algorithmic properties of iterative optimization to execute multiple instances of the update rule. The Python-based DSL provides a concise means of expressing this update rule for a broad set of algorithms while facilitating parallelization.

DAnA leverages these insights to provide a cross-stack solution that generates FPGA-synthesizable accelerators which seamlessly directly interface with the RDBMS engine's buffer pool.

3.2 Workflow

Figure 3 shows the integration of DAnA in the traditional software stack of data management systems. With DAnA, the data scientist specifies her desired ML algorithm as a UDF using a simple DSL integrated within Python. DAnA performs static analysis and compilation of the Python functions to program the FPGA with a high-performance, energy-efficient hardware accelerator design. The hardware design is tailored to the ML algorithm and page specifications of the RDBMS engine. DAnA stores accelerator metadata (*Strider* and execution engine instruction schedules) in the RDBMS's catalog along with the name of a UDF to be invoked from a SQL query. As shown in Figure 3, the RDBMS catalog is shared by the database engine and FPGA. To run the UDF on her training data, the user provides a SQL query. The RDBMS parses, optimizes, and executes the query while treating the UDF as a black box. During query execution, the RDBMS fills the buffer pool, from which DAnA ships the training table's pages to the FPGA for processing. DAnA and the RDBMS engine work in tandem to generate the appropriate data stream, data route, and accelerator design for the {ML algorithm, RDBMS's page layout, FPGA} triad. Each component of DAnA's workflow is briefly described below.

Programming interface. The front end of DAnA exposes a Python-embedded DSL (discussed in §4.1) to express the ML algorithm as a UDF. The UDF includes an update rule that specifies how each tuple or record in the training data updates the ML model by using the operations available in the DSL. It also expects a merge

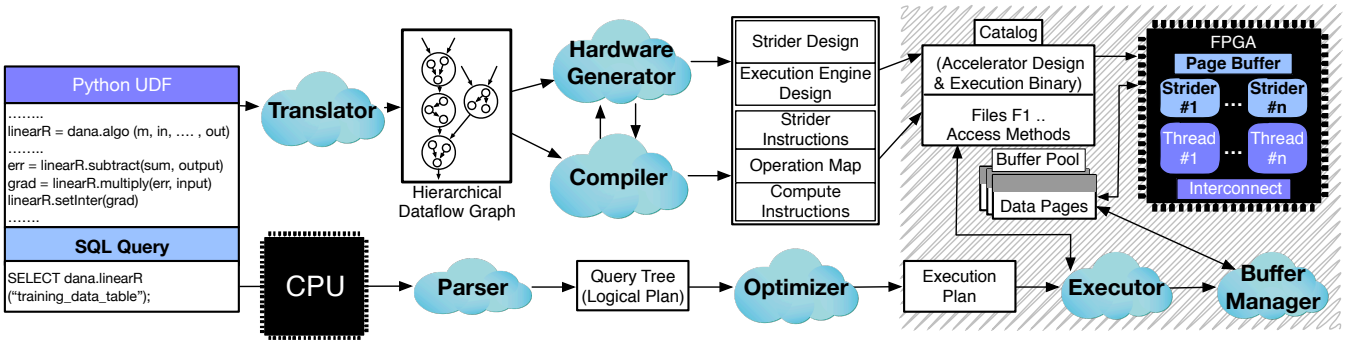


Figure 3: Overview of the DAnA workflow that integrates FPGA acceleration with the RDBMS engine. The Python-embedded DSL provides an interface to express the algorithm, which is converted to hardware and stored in the RDBMS catalog. The RDBMS engine fills the buffer pool and the FPGA Striders directly access these data pages to extract the tuples and feed to the threads. Shaded areas show the entangled components of RDBMS and FPGA concurrently working in tandem to accelerate in-database analytics.

function that specifies how to process multiple tuples in parallel and aggregate the resulting models. DAnA's DSL constitutes a diverse set of operations and data types that cater to a wide range of advanced analytics algorithms. Any legitimate combination of these operations can be automatically be converted by DAnA to a final synthesizable FPGA accelerator.

Translator. The provided UDF is converted into a Hierarchical DataFlow Graph (*h*DFG) by DAnA's parser, discussed in detail in §4.4. Each node in the *h*DFG represents a mathematical operation allowed by the DSL and each edge is a multi-dimensional piece of data on which the operations are performed. The information in the *h*DFG enables DAnA's backend to optimally customize the reconfigurable architecture and schedule and map each operation for a high-performance execution.

Strider-based customizable architecture. To target a wide range of ML algorithms, DAnA offers a parametric reconfigurable hardware design that is hand optimized by expert hardware designers, as described in § 5. The hardware interfaces with the database engine to provide high-performance, low-energy computation. To extract higher performance from hardware acceleration, this design eliminates the CPU from the data transformation process. Instead, DAnA deploys a specialized structure, called a *Strider*, which directly interfaces with database's buffer pool and extracts the training data. A *Strider* processes at a page granularity to amortize the cost of per-tuple data transfer from memory to the FPGA. To exploit the vast amount of data available on-chip, the architecture is equipped with execution engines that run multiple parallel instances of the update rule. This architecture is customized in accordance to the FPGA specifications, database page layout, and the analytics function by DAnA's compiler and hardware generator.

Instruction Set Architectures. Both *Striders* and the execution engine are programmed with a novel Instruction Set Architecture (ISA). The *Strider* ISA instructions process page headers, tuple headers, and extract the raw training data from a database page. Different page sizes and page layouts can be targeted using this ISA. The independent execution engine ISA can selectively run the engine in SIMD mode to amortize the cost of instruction handling. Instructions are statically scheduled to avoid imposing the traditional shortcomings of SIMD execution that require synchronizing all the processing engines.

Compiler and hardware generator. DAnA's compiler and hardware generator work together to ensure compatibility between the *h*DFG and customizable architecture design. They generate the final FPGA synthesizable accelerator, a static schedule for this design, a map of where operations are performed, and instructions for the *Strider* and execution engine. The compiler converts the database page configuration into a set of instructions that process the page and tuple headers and transform user data into a floating point format. For the given *h*DFG and FPGA specifications (such as number of DSP Slices and BRAMs), the hardware generator determines the parameters for the execution engine and *Striders*.

As described above, simultaneously providing flexibility and reconfigurability with high performance for advanced analytics is a very challenging but a pertinent problem. DAnA is a multifaceted solution that one by one untangles these challenges.

Table 1: Language constructs for DAnA's Python-embedded DSL.

Type	Keyword	Description
Component	`algo`	To specify an instance of the learning algorithm
Data Types	`input`	Algorithm input
	`output`	Algorithm output
	`model`	Machine Learning model
	`inter`	Interim data type
	`meta`	Meta parameters
Mathematical Operations	`+,-,*, /, >, <`	Primary operations
	`sigmoid, gaussian, sqrt`	Non linear operations
	`sum, norm, pi`	Group operations
Built-In Special Functions	`merge(x, int, "operation")`	Specify merge operation and number of merge instances
	`setEpochs(int)`	Set the maximum number of epochs
	`setConvergence(x)`	Specify the convergence criterion
	`setModel(x)`	Set the model variable

4. FRONT-END INTERFACE OF DAnA

DAnA's DSL provides an entry point for data scientists to exploit hardware acceleration for in-RDBMS analytics. This section first elaborates on the constructs and features of the DSL and how they can be used to train a wide range of learning algorithms for advanced analytics. The section then explains how a UDF defined in this DSL is translated into an intermediate representation (*h*DFG) for automatic hardware generation.

4.1 Programming For DAnA

DAnA exposes a high level DSL for database users to provide their learning algorithm expressed as a UDF. Embedding this DSL within Python allows support for intricate update rules within a framework familiar to database users whilst not requiring a full language compiler. This DSL meets the following objectives:

1. Incorporate language constructs commonly seen in a wide class of supervised learning algorithms.
2. Support expression of any iterative update rule, not just variants of gradient descent, however, conform to the DSL constructs.
3. Segregate algorithmic specification from hardware-dependent implementation.

All the language constructs of this DSL, *components*, *data declarations*, *mathematical operations*, and *built-in functions*, are summarized in Table 1. Users express the learning algorithm using these constructs by providing the (1) *update rule*, to decide how

each tuple in the training data updates the model; (2) *merge function*, to specify the combination of distinct parallel update rule threads; and (3) *terminator*, to describe convergence.

4.2 Language Constructs

Data declarations. Data declaratives of the DSL delineates the semantics of the data types used in the ML algorithm. The DSL supports the following data declarations: input, output, inter, model, and meta. Once the analyst imports the dana package, he or she uses the declarations as follows.

```
mo = dana.model ([5][2])
```

In the above code snippet, dana.model defines a multi-dimensional ML model of size [5][2]. Each data type can be declared by specifying the type and its dimensions. If no dimensions are specified, the variable is considered to be a scalar. The dana.input and dana.output are used to express a single input-output pair in the training dataset. Additionally, the user can specify meta variables using dana.meta, the values of which remains constant throughout the application. As such, meta variables can be directly sent to the FPGA before algorithm execution. All variables used for a particular algorithm must be linked to an algo component.

```
algorithm = dana.algo (model, input, ...)
```

This algo component allows the user to link together the three functions of a single UDF. Learning algorithms often require a significant amount of multi-dimensional operations. To address this need, the DAnA DSL provides a dana.iterator variable which is used to specify implicit loops. Apart from the above data declarations, the analyst can use untyped intermediate variables, which are automatically labeled as dana.inter by DAnA's backend.

Mathematical operations. DAnA's DSL supports mathematical operations performed on both declared and untyped intermediate variables. Primary and non-linear operations, such as {*, +, ... , sigmoid}, only require the operands as input. The dimensionality of the operation is automatically inferred by DAnA's translator (as discussed in § 4.4) in accordance to the operand/s dimensions. Group operations, such as {sum, pi, norm}, perform computation across elements. As such, group operations require the input operands and the grouping dimension. The grouping dimension is specified via a iterator variable, which alleviates the need to explicitly specify loops. Group operations first perform the underlying primary operation on the input operands prior to this grouping.

Special Functions. The DAnA DSL provides four built-in functions which are used to specify the merge function, set the terminating convergence conditions, and update the model. The merge(x, int, "operation") function is used to specify how multiple threads of the update rule are combined. Next, the convergence is dictated by either a fixed number of epochs [1] or a user-specified condition. The setEpochs(int) function sets the number of terminating epochs. Alternatively, the setConvergence(x) frames termination based on a boolean variable x. Finally, the setModel(x) function is used to link a DAnA variable which represents the updated model to the corresponding algo component.

All of these language constructs are supported by DAnA's reconfigurable architecture, and hence, can be synthesized on the FPGA. Below, an example usage of these constructs to express the update rule, merge function, and convergence for linear regression algorithm running the gradient descent optimizer is provided.

4.3 Linear Regression Example

[1] One epoch is single pass over the entire training data set.

Update rule. As the code snippet below illustrates, the analyst first declares the different data types with their corresponding dimensions. The analyst then provides the computations performed over these variables specific to linear regression.

```
def algorithm(...):
  #Data Declarations
  mo = dana.model ([10])
  in = dana.input ([10])
  out  = dana.output ()
  lr = dana.meta (0.3) #learning rate

  linearR = dana.algo (mo, in, out, ..)
  i = dana.iterator (10)

  #Derivative of the Loss Function
  s = sum ( mo * in, i )
  er = s - out
  grad = er * in

  #Gradient Descent Optimizer
  up = lr * grad
  moupdate = mo - up
  linearR.setModel (moupdate)
```

In this example, the update rule is the gradient of the loss function. Thus, the gradient descent optimizer updates the model in the negative direction of the gradient of the loss function ($\frac{\partial(l)}{\partial w^{(t)}}$). As previously mentioned, the analyst concludes with the setModel() function to identify the updated model, in this case moupdate.

Merge Function. The merge function facilitates spawning multiple threads of the update rule on the FPGA accelerator by specifying the functionality of the point of merge. For instance, the merge function can combine the output either from the loss function definition or the optimization function.

```
mergeCoef = dana.meta (8)
grad = linearR.merge(grad, mergeCoef, "+")
```

In the above merge function, the intermediate grad variable has been combined using addition, and the merge coefficient (mergeCoef) specifies the batch size. DAnA's compiler implicitly understands that the merge function is performed before the optimizer description. Specifically, the grad variable is calculated separately for each tuple per batch. The results are aggregated together across the batches and used to update the model. Alternatively, partial model updates for each batch could be merged.

```
mergeCoef = dana.meta (8)
m1 = linearR.merge(moupdate, mergeCoef, "+")
m2 = m1/mergeCoef
lineaR.setModel(m2)
```

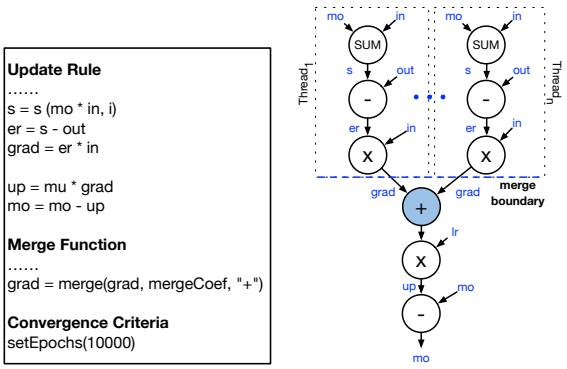
The moupdate is calculated by each thread for each tuple in its batch separately and then averaged. These above functions are just two examples of merge descriptions, which provide the flexibility to create different learning algorithms without requiring any modification to the update rule. For instance, the first definition of the merge function above creates a linear regression running batched gradient descent optimizer. In contrast, the second definition corresponds to a parallelized stochastic gradient descent optimizer.

Convergence Function. Finally, the user is expected to provide the termination criteria. As shown in the code snippet below, the convergence here compares the Euclidean norm of **grad** with a constant **convergenceFactor**.

```
convergenceFactor = dana.meta (0.01)
n = norm(grad , i)
conv = n < convergenceFactor
linear.setConvergence(conv)
```

When the conv variable is true, training is terminated. Alternatively, the number of epochs can be specified as the convergence criteria using the syntax **linearR.setEpochs(10000)**.

Query run. The UDF comprises the update rule, merge function, and convergence check, which defines the entire analytics algorithm. The linearR UDF can then be called in a query as follows:



(a) Code snippet (b) Hierarchical DFG

Figure 4: Translator-generated *h*DFG for the linear regression code snippet expressed in DAnA's DSL.

```
SELECT dana.linearR('training_data_table');
```

As such, this Python-embedded DSL provides a high level programming abstraction that can easily be invoked by an SQL query and extended to incorporate algorithmic advancements. We next discuss how this UDF is converted to a *h*DFG.

4.4 Translator

DAnA's front-end, called the translator, is tasked with converting the user-provided UDF to a *h*DFG. The *h*DFG represents the coalesced update rule, merge function, and convergence check and maintains all data dependencies. Each node of the *h*DFG represents a multi-dimensional operation, which can be decomposed into smaller atomic sub-nodes. Sub-nodes represent single operations performed by the accelerator. The *h*DFG transformation for the linear regression example provided in the previous section is shown in Figure 4. The aim of the translator is to expose as much parallelism available to the remainder of the DAnA workflow. This includes parallelism within a single instance of the update rule, as well as among different threads running independent update rule instances. To accomplish this, the translator (1) maintains the function boundaries, especially between the merge function and the portions of the update rule which can be parallelized, and (2) automatically infers the dimensionality of the nodes and edges in the graph.

The merge function and convergence criteria are only performed once per iteration of the update rule. However, the update rule can be readily parallelized across multiple threads. In Figure 4b, the colored node represents the merge operation, which adds together the gradients generated by separate instances of the update rule. These update rule instances are run in parallel and consume different records or tuples from the training data. Additionally, the translator infers the dimensions of each operation node and its output edge(s). For basic operations, if both the inputs have the same dimensions, it translates it to an element by element operation. In the case where the inputs do not have the same dimension, the input with lower dimension is assumed to be replicated, and the generated output has the dimensions of larger input. Nonlinear operations, on the other hand, only have a single input, which determines the output edge dimensions. With group operations, the output dimension is determined by the iterator variable. For example, an output of size [5][2] is generated for a node performing sum(mo * in, i), where variables mo and in are matrices of sizes [5][10] and [2][10], respectively. This information allows the hardware generator to configure the accelerator architecture to optimally cater for these operations. Resources are distributed on-demand within and across multiple threads. Furthermore, DAnA's compiler uses this information to map operations optimally to the accelerator architecture to exploit fine-grained parallelism within an update rule. Before delving into the details of hardware generation and compilation, we discuss reconfigurable architecture and its ISA. We also elaborate on how together, the architecture and ISA facilitate the high-performance execution of iterative optimization algorithms.

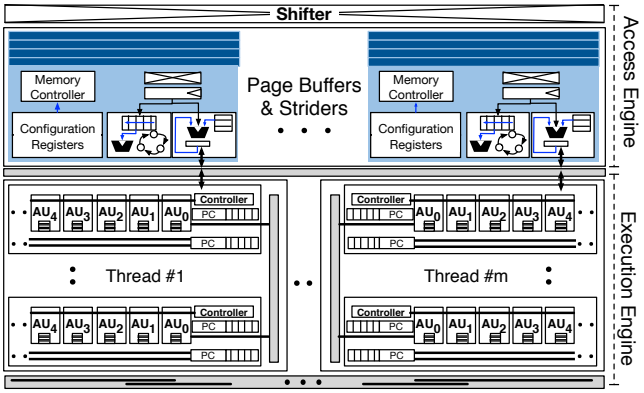


Figure 5: Reconfigurable accelerator design in its entirety. The access engine reads and processes the data via its Striders, while the execution engine operates on this data according to the UDF.

5. ARCHITECTURE DESIGN FOR IN-DATABASE ACCELERATION

DAnA employs a parametric accelerator architecture, which constitutes a *multi-threaded access engine* and a *multi-threaded execution engine* shown in Figure 5. Both the engines expose a custom Instruction Set Architecture (ISA) to generate micro-programs for the architecture. The access engine harbors *Striders* to ensure compatibility between the data stored in a particular database engine and the execution engines that perform the computations required by the learning algorithm. *Striders* and the execution engine are configured according to the page layout and UDF, respectively. The details of each of these components are discussed below.

5.1 Access Engine and Striders

5.1.1 Architecture and Design

The multi-threaded access engine is responsible for storing pages of data and converting them from a database page format to raw numbers processed by the execution engine. Figure 6 shows a detailed diagram of this access engine. The access engine uses the AXI link to transfer uncompressed database pages and configuration data to the page buffers and configuration registers, respectively. Configuration data comprises *Strider* and execution engine instructions, as well as necessary meta-data. Both the training data in the database pages and the configuration data are passed through a shifter for alignment, according to the read width of the block RAM on the target FPGA. A separate channel for configuration data uses a finite state machine to dictate the route and destination of the configuration information.

To amortize the cost of data transfer and avoid the suboptimal usage of the FPGA bandwidth, the access engine and *Striders* operate at a page level granularity to process database pages on-chip. Training data is written to multiple page buffers, where each buffer stores one database page at a time and has access to its own *Strider*. Alternatively, each tuple could have been extracted from the page by the CPU and sent to the FPGA for processing. This approach would fail to exploit the bandwidth available on the FPGA, as only one tuple would be sent at a time. Furthermore, using the CPU for

Table 2: Strider ISA to read, extract, and clean the page data.

Instruction	Instruction Code	Bits			
		21 - 18	17 - 12	11 - 6	5 - 0
Read Bytes	readB	Opcode = 0	Read Address	# of Bytes	Write Address
Extract Bytes	extrB	Opcode = 1	Byte Offset		
Write Bytes	writeB	Opcode = 2	Read Address		
Extract Bits	extrBi	Opcode = 3	Start Location	Offset	# of Bits
Clean	cln	Opcode = 4			
Insert	ins	Opcode = 5			Reserved
Add	ad	Opcode = 6	Read Address 1	Read Address 2	Immediate Operand
Subtract	sub	Opcode = 7			
Multiply	mul	Opcode = 8			
Branch Enter	bentr	Opcode = 9	0		
Branch Exit	bexit	Opcode = 10	Condition	Value	

data extraction would require a significant overhead for handshaking between it and the FPGA. Offloading tuple extraction to the accelerator using *Striders* provides a unique opportunity to dynamically interleave unpacking data in the access engine and processing it in the execution engine.

It is common for data to be spread across pages where each page requires plenty of pointer chasing. Two tuples cannot be simultaneously processed from a single page buffer, as the location of one could depend on where the previous tuple ended. Therefore, we store multiple pages on the FPGA and parallelize data extraction from the pages across their corresponding *Striders*. For every page, the *Strider* first processes the page header and extracts necessary information about the page, which is stored in the configuration registers. The information includes any important offsets, such as the beginning of the tuple, and the sizes of each tuple, which is either located or computed from the data in the header. This auxiliary page information is used to trace the tuples' addresses and read the corresponding data out from the page buffer. The shifter after each page buffer ensures alignment of the tuple data between page buffer and *Strider*. This is vital to efficient performance, as tuple size varies and is dependent on the table schema. From the tuple data, its header is processed and use to extract and route training data to the execution engine. The number of *Striders* and database pages stored on-chip can be adjusted according to the BRAM storage available on the target FPGA. The internal workings of the *Strider* are dictated by instructions which depend on the page layout and page size of the target RDBMS. We next discuss the novel ISA which enables programming these *Striders*.

5.1.2 Instruction Set Architecture

We devise a novel fixed-length Instruction Set Architecture (ISA) for the *Striders* to target a range of RDBMS engines, such as PostgreSQL and MySQL (innoDB), that have similar backend page layouts. Data inside an uncompressed page of these RDBMS engines can be read, extracted, and cleansed using the ISA's instructions. The ISA is light-weight and specialized for pointer chasing and data extraction. *Strider* instructions are generated statically by DAnA's compiler, discussed in §6.2. Each *Strider* is programmed with the exact same instructions, while the page operated on differs.

The Table 2 illustrates the 10 instructions of this ISA. Every instruction is 22 bits long, which consists of a unique operation code (opcode) to identify the instruction. The remainder of the bits are specific to the opcode. Instructions **Read Bytes** and **Write Bytes** are responsible for reading and writing data from the page buffer, respectively. This data is aligned by the shifter and written into a register in the *Strider*. It is common for database pages to be variably aligned. Therefore, we provided the flexibility in the ISA to extract at both byte and bit granularity using the **Extract Byte** and **Extract Bit** instructions. Both the extraction instructions operate on the data residing in the *Strider*. Next, the **Clean** instruction can remove parts of the data not required by the execution engine. Conversely, the **Insert** instruction enables the ISA to add bits to the data, such as NULL characters and auxiliary information, which are particularly useful when the page is to be written back to memory. Basic math operations, **Add**, **Subtract**, and **Multiply**, are also provided in the ISA and are used to calculate the tuple sizes, byte offsets, etc. Finally, the **Bentr** and **Bexit** branch instructions provide the capability to specify jumps or loop exits, respectively. This feature invariably reduces the instruction footprint when instructions are to be repeated in a pattern while enabling loop exits that depend on a dynamic runtime variable.

The *Strider* ISA provides the means to extract data from an uncompressed page. An example page layout, as offered by PostgreSQL and MySQL, is illustrated in Figure 7. This layout is often divided into a page header, tuple pointers, and the tuple data. It can be processed using the following assembly code snippet.

```
\\Page Header Processing
readB 0, 8, %cr
readB 8, 2, %cr
readB 10, 4, %cr
extrB %cr, 2, %cr

\\Tuple Pointer Processing
readB %cr, 4, %treg
extrB 0, 1 ,%cr
extrB 1, 1 ,%treg

\\Tuple extraction and processing
bentr
ad %treg, %treg, 0
readB %treg, %cr, %treg
extrB %treg, %cr, %treg
cln %treg, %cr, 2
bexit 1, %treg, %cr
```

This code is generated by DAnA's compiler using the page layout information. Each line of assembly code has an instruction name, which signifies the opcode and the its corresponding fields.

The first four assembly instructions process the page header to obtain the configuration information. For example, the Read Bytes **(readB 0, 8, %cr)** instruction reads 8 bytes from address 0 in the page buffer and adds this page size information into a configuration register. Each variable shown at %(reg) corresponds to an actual *Strider* hardware register. The %cr and %t specify a configuration and temporary register, respectively. Next, the first tuple pointer is read to the extract the byte-offset and length in bytes of the start tuple. As all training data tuples are expected to be identical, only the first tuple pointer needs to be processed. Each tuple is then processed by adding the tuple size to previous offset, reading from the stream buffer, extracting the user data, and cleaning the auxiliary information. As the above step is repeated for each tuple, the **bentr** and **bexit** branch instructions are used. The loop is completed when the tuple offset address reaches the end of free space. Once the data is cleaned, it is passed to the execution engines for consumption.

5.2 Execution Engine

5.2.1 Architecture and Design

The execution engines perform the *h*DFG generated from the user provided UDF on the *Strider*-processed data. As the BRAM capacity has been increasing rapidly with the new FPGAs (Arria 10 offers 7 MB, UltraScale+ VU9P offers 44 MB), more database pages can be stored on-chip. Therefore, the execution engine needs to furnish enough computational resources that can process this copious amount of on-chip data. Our reconfigurable execution engine architecture can run multiple threads of parallel update rules for different data tuples. This architecture is backed by a *Variable Length Selective SIMD ISA*, that aims to exploit the vast regular parallelism in ML algorithms whilst providing the flexibility to each component of the architecture to run independently.

Reconfigurable compute architecture. All the threads in the execution engine are identical and perform the same computa-

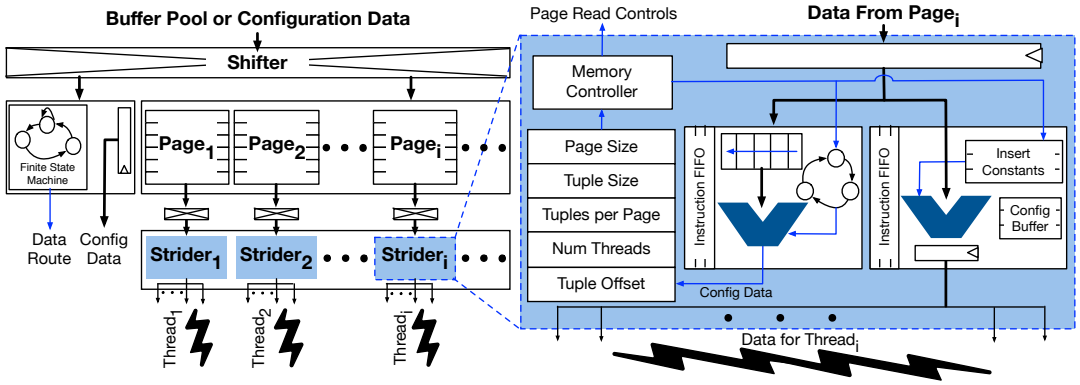


Figure 6: Access engine design which uses Striders as the main interface between the RDBMS and execution engines. Uncompressed data pages are read from the buffer pool and stored in on-chip page buffers. Each page has a corresponding strider to extract the tuple data.

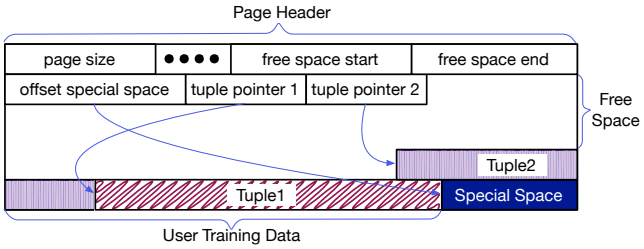


Figure 7: Sample page layout similar to PostgreSQL.

tions on different tuples.DAnA balances the resources allocated per thread vs. the number of threads to ensure high performance for each algorithm. The hardware generator of DAnA (discussed in §6.1) determines this division by taking into account the parallelism in the *h*DFG, number of compute resources available on chip, and number of striders/page buffers that can fit on the on-chip BRAM. The architecture of a single thread is a hierarchical design comprising analytic clusters (ACs), which, in turn, are composed of multiple analytic units (AUs). As discussed below, the AC architecture is designed while keeping in mind the algorithmic properties of multi-threaded iterative optimizations and the AU is constructed to cater to commonly seen compute operations in data analytics.

Analytic cluster. An AC is a group of analytics units designed to reduce the data transfer latency between the AUs, as shown in Figure 8a. Therefore, mathematical *h*DFG nodes which exhibit high data dependencies are all scheduled to a single cluster. In addition to providing greater connectivity among the AUs within an AC, the cluster serves as the control hub for all its constituent AUs. The AC runs in a selective SIMD mode, where the AC specifies which AUs within a cluster perform an operation. Each AU within a cluster is expected to execute either a cluster level instruction (add, subtract, multiply, divide, etc.) or a no-operation (NOP). The NOP is used to specify that the AU should not perform an operation at the current cycle. Finer details about the source type, source operands, and destination type can be stored in each individual AU for additional flexibility. This collective instruction technique simplifies the AU design, as each AU no longer requires a separate controller to decode and process the instruction. Instead, the AC controller processes the instruction and sends control signals to all the AUs. When the designated AUs complete their execution, the AC proceeds to the next instruction by incrementing the program counter.

To exploit the data locality among the operations performed within an AC, different connectivity options are provided. Each AU within an AC is connected to both its neighbors, and the AC has a shared bus in the form of line topology. The number of AUs per AC are fixed to 8 to obtain highest operational frequency. A single thread generally contains more than one instance of a AC,

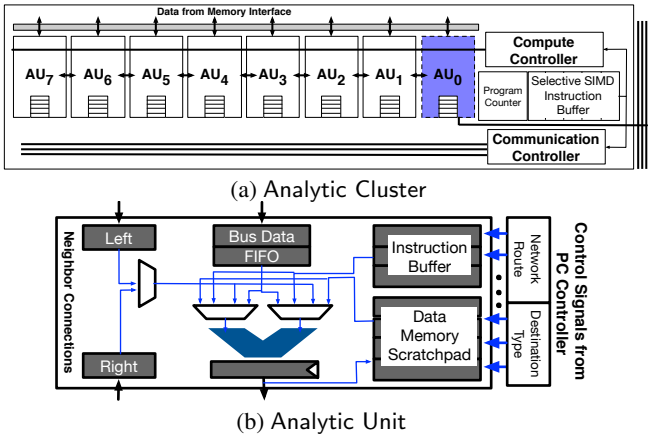


Figure 8: (a) Single analytic cluster comprising analytic units operating in a selective SIMD mode and an (b) analytic unit, which is the pipelined compute hub of the architecture.

each performing instructions independently. However, data sharing among ACs is possible via a shared inter-AC bus, which is also in the form of line topology. One AU per AC has access to this inter-AC bus, called the *staple* AU. If any other AU requires bus access, the data first needs to be sent to the staple AU.

Analytic unit. The AU is the basic compute element of the multi-threaded execution engine. It can be tailored by the hardware generator to satisfy the mathematical requirements of the *h*DFG. Control signals are provided by the AC according to its instructions. Data for each operation can be read from the data memory according to the instruction's source type. Training data and intermediate results are stored in the data memory. Additionally, data can be read from the bus FIFO (First In First Out) and/or the registers corresponding to the left and right neighbor AUs. Data is then sent to the ALU, which supports complicated non-linear operations, such as sigmoid, gaussian, and square root, in addition to basic math operations. Basic operations use the on-chip DSP provided on the FPGA, while the remainder are designed using the FPGA resources. The internals of the ALU are reconfigured according to the operations required by the *h*DFG. After AU computation, data is sent to the neighboring AUs, the shared bus within the AC, and/or the memory according to the instruction.

Bringing the Execution Engines together. Results from each thread are combined across the threads via a computationally-enabled tree bus in accordance to the merge function. This tree bus has an attached light ALU that can perform computations on in-flight data. This is particularly useful for aggregating results incoming from all the threads. We now delve deeper in the execution

Table 3: (a) A variable-length ISA for the execution engine, (b) its supported operations, and (c) the types of instruction operands.

(a) Three Categories of Instructions

Instruction Type	Description		AU0	AU1	AU2	AU3	AU4	AU5	AU6	AU7	
Compute	Opcode	N/A	Operational AU (1 Byte)								Opcode
Operand Read	Operand Type		Source Type (1 Byte - 4 Bytes)								N/A
	Operand Index		Source Index (1/2 Byte - n Bytes)								
Write Back / Communicate	Global Bus	Use	Destination AUs (1 Byte)								Source AU
	Local Bus										
	Memory		Write to Scratchpad (1 Byte)								N/A

(b) Supported Math

Basic Compute Instructions	Add	Opcode = 1
	Subtract	Opcode = 2
	Divide	Opcode = 3
	Multiply	Opcode = 4
Group Compute Instructions	Sum	Opcode = 5
	Norm	Opcode = 6
	Pi	Opcode = 7
Non Linear Compute Instructions	Sigmoid	Opcode = 8
	Gaussian	Opcode = 9
	Sqrt	Opcode = 10

(c) Operand Types

Operand Type	Code	Index
Data Memory	00	Address
Scratchpad	01	
Neighbor	02	Left/Right
Bus	03	Local/Global

engine's ISA, which programs these execution engines.

5.2.2 Instruction Set Architecture

Our variable-length ISA for the execution engines supports a Selective SIMD processing model, which targets two types of nodes in the *h*DFG: those easily vectorized and those which exhibit limited parallelism due to high data dependencies. A variable length ISA elongates the decoding process but reduces its overall memory footprint, leaving more room for the data pages to be stored on on-chip. Despite being variable length, every instruction is self-sufficient and contains all relevant material.

As shown in Table 3, this ISA has three instruction types: compute, communication, and operand reads. These are micro-instructions, which are low-level instructions generated for each operation in the *h*DFG to be performed. The shaded blue fields in the table are mandatory, while the remaining are optional. Fields specified as N/A are filler to align the fields relevant to each AU for illustration purposes and are not a part of the actual micro-instruction. Each operation to be performed always has the compute and write back/communication micro-instructions stored inside the AC's instruction buffer. In contrast, the operand read micro-instructions are optional and are stored inside the AU's instruction buffer. For the **compute** micro-instructions, the first field is a byte which indicates the operational AUs. This field is required and specifies which AUs perform the operation. The next field is the mathematical operation identifier specified using an opcode. Operations currently supported by DAnA's DSL and hardware are listed in Table 3b. All AUs that participate in executing the operation have the corresponding **operand read** micro-instructions in their instruction buffer. These operand read micro-instructions have two types: operand type and operand index. The operand type for any AU specifies the source type of the operation. Figure 3c shows all the operand types – data memory, scratchpad, neighbor register, and bus FIFO – from which each AU can read or write its operands or outputs, respectively. The operand index instruction is only valid if the operand is of the data memory type, otherwise no index is provided. Finally, the **Communicate** and **Write Back** micro-instructions itemize where the data is to be written by the AUs performing the operation. The first field, Use, in these micro-instructions is always required and specifies whether the write back or communication component is being used. For example, if the local bus use field is equal to 1, the Source AU uses the bus to transfer data to all AUs indicated by the "Destination AU" field. An example of this ISA can be given using a simple set of operations to be performed in a single AC. The group operation **sum** shown below is performed in an AC using operations shown in Table 4.

$$s = sum(x_i \times w_i, [8]) \quad (2)$$

Table 4: The operation flow in the AC to perform Equation 2.

AU0	AU1	AU2	AU3	AU4	AU5	AU6	AU7
*	*	*	*	*	*	*	*
+		+		+		+	
+				+			
+							

All AUs in an AC perform the multiply operation. The results are aggregated through reduction tree to generate the final result in AU 0. The first multiply operation is converted to a compute micro-instruction with opcode 100 (from Table 3b) and operational AU value of 11111111, as all the AUs perform the operation. The operand read micro-instruction for each AU points towards data memory, encoded as 00 00 for both operands. Operand indices are set to 0000 0001, assuming both the multiplicand and the multiplier are in consecutive memory locations. Regarding the communication micro-instructions, neither the global bus nor the local bus is used. Therefore the Use field for both is 0, and none of the other fields in those micro-instruction are required. However, for AUs 0, 2, 4, and 8, a subsequent add operation needs to be performed on the previous result. Hence, the multiply output needs to be stored in scratchpad via a memory micro-instruction. The Use field corresponding to the scratchpad is set to 1, while the Write to Scratchpad field is set to 10101010 to indicate that only alternating AUs need to write.

Next, we discuss the inner workings of DAnA's compiler and hardware generator. These components are responsible for ensuring compatibility between the high level Python UDF, the ISA, and the reconfigurable access and execution engines.

6. HARDWARE GENERATOR AND COMPILER

DAnA's translator, scheduler, and hardware generator work in tandem to configure the accelerator design for the UDF and create its runtime schedule. As discussed in § 4.4, the translator converts the user-provided UDF, merge function, and convergence criteria into a *h*DFG, where each *h*DFG node may consist of sub-nodes. For instance, the **sum** node shown in Table 4 is expressed as a single node in the *h*DFG, which is comprised of 15 sub-nodes. All sub-nodes are scheduled and mapped to the final accelerator hardware design based on the *h*DFG. The hardware generator outputs a single-thread architecture for the operations of these sub-nodes and determines the number of threads to be instantiated. The compiler then runs the mapping and scheduling algorithms to statically map all operations to AUs.

6.1 Hardware Generator

The hardware generator finalizes the parameters of the reconfigurable architecture for both the *Striders* and the execution engine, shown in Figure 5. From the catalog, the hardware generator retrieves the database page layout information, model, and training data schema. FPGA-specific information, such as the number of DSP slices, the number of BRAMs, the capacity of each BRAM, the number of read/write ports on a BRAM, and the off-chip communication bandwidth are provided by the user. With this information, the hardware generator can modify the amount of allocated resources to the access and execution engine.

Page size, model size, and the size of a single training record determine the amount of memory utilized by each *Strider*. Specifically, a portion of the BRAM is allocated to store the extracted raw training data and model. The remainder of the BRAM memory is assigned to the page buffer to store as many pages as possible to maximize the off-chip bandwidth utilization. Once the number of resident pages is determined, the hardware generator uses the FPGA's DSP information to calculate the number of AUs which

can be synthesized on the target FPGA. Within each AU, the ALU is customized to contain all the operations required by the *h*DFG.

The number of AUs, in turn, determines the number of ACs that can fit on the FPGA. Each thread is allocated a number of ACs determined by the merge coefficient provided by the programmer. It creates at most as many threads as this coefficient. The ACs dedicated to a thread are catered to by a single *Strider*, that feeds them the raw training data. Using these specifications, the hardware generator converts the final architecture into a functional and synthesizable design that can efficiently run the analytics algorithm.

6.2 Compiler

The compiler schedules, maps, and generates the micro-instructions for both ACs and AUs for all the sub-nodes in the *h*DFG. For scheduling and mapping a node, the compiler keeps track of the sequence of scheduled nodes assigned to each AC and AU on a per-cycle basis. For each node which is "ready" ($node_{ready}$), i.e., all its predecessors have been scheduled, the compiler tries to place that operation with the goal to improve throughput. Often, a single node in the *h*DFG has dimensions which are greater than the number of AUs in an AC. Elementary and non-linear operation nodes are spread across as many AUs required by the dimensionality of the operation. As these operations are completely parallel and do not have any data dependencies within a node, they can be dispersed. For instance, in an element-wise vector-vector multiplication, each vector with 16 scalar values, will be scheduled across two ACs (8 AUs per ACs). However, group operation nodes, like (sum, pi, norm), exhibit data dependencies. Hence, they are mapped with a goal to minimize the communication cost. Once all the nodes and sub-nodes are mapped, the code generation converts each operation into AC micro-instructions and corresponding AU micro-instructions.

The FPGA design its schedule, operation map, and instructions generated by the hardware generator and compiler are then stored in the RDBMS catalog. These components are executed when the query calls for the corresponding UDF.

7. EVALUATION

We prototype DAnA by integrating with PostgreSQL and compare the end-to-end runtime performance of DAnA-generated accelerators with a popular scalable in-database analytics library, Apache MADlib [13, 14], for both PostgreSQL and Greenplum RDBMSs. We aim to understand the impact of the training dataset size on performance by using both real and synthetic datasets for a diverse set of machine learning models. Additionally, we investigate the impact of *Striders* on the overall runtime of the system and how DAnA's performance varies with the system parameters. These parameters include the buffer page size, number of Greenplum segments, and multi-threading on the hardware accelerators.

Datasets and Workloads. Table 5 lists the datasets and machine learning models used to evaluate DAnA. These workloads cover a diverse range of machine learning models, such as logistic regression, support vector machines, low rank matrix factorization, and linear regression. The Remote Sensing, WLAN, Patient, and Blog Feedback datasets are real and from the UCI repository [37]. Remote Sensing is a classification dataset and runs both logistic regression and support vector machine algorithms. Netflix is a movie recommendation dataset used by the low rank matrix factorization algorithm. Additionally, the model topology of each benchmark is provided in the table, along with the number of tuples in the training dataset. The table also lists the number of uncompressed 32 KB pages it takes to fit the entire training dataset.

Table 5: Descriptions of datasets and machine learning models used for evaluation. Shaded rows are synthetic datasets.

Real Dataset	Machine Learning Algorithm	Model Topology	Training Data	
			# of Tuples	# of 32KB Pages
Remote Sensing	Logistic Regression, Support Vector Machine	54	581102	4924
WLAN	Logistic Regression	520	19937	1330
Netflix	Low Rank Matrix Factorization	6040 x 3952 x 10	6040	1345
Patient	Linear Regression	384	53500	1941
Blog Feedback	Linear Regression	280	52397	2675
Synthetic Logistic	Logistic Regression	2000	387944	96986
Synthetic SVM	Support Vector Machine	1740	678392	169598
Synthetic Linear	Linear Regression	8000	130503	130503

Table 6: Arria10 FPGA specifications.

	FPGA Capacity		Operational Frequency	BRAM Capacity	# DSPs	Technology
Altera Arria 10 GX115	427K LUTS	1708K Flip-Flops	150 MHz	7 MB	1,518	TSMC 20nm

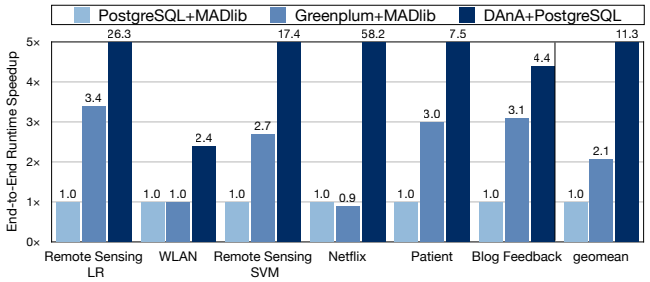
For real datasets, almost all data can be brought into the buffer pool at once, and hence do not impose high I/O overheads. To evaluate the performance with out-of-memory workloads, we generate three synthetic datasets, shown by the shaded rows of Table 5.

Experimental Setup We use the Altera Arria10 as DAnA's acceleration FPGA and synthesize the hardware with Qaurtus II v14.1. Specifications of the FPGA board are provided in Table 6. The baseline experiments for MADlib were performed on a machine with four Intel i7-6700 cores at 3.40GHz running Ubuntu 16.04 LTS with kernel 4.8.0-41, 32GB memory, a 256GB Solid State Drive storage. We measure the runtime of each workload for MADlib running on both PostgreSQL and Greenplum for single- and multi-threaded performance, respectively.

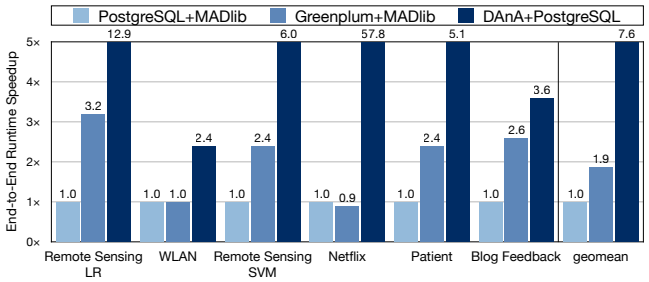
Default setup. Our default setup uses a 32 KB buffer page size and 128MB buffer pool size for all three systems. As DAnA operates with uncompressed pages to avoid decompression overheads on-chip, 32 KB pages are used as a default to fit at least 1 tuple per page for all the datasets. To understand the performance sensitivity by varying the page size on PostgreSQL and Greenplum, we measured end-to-end runtimes across 8KB, 16KB, and 32KB page sizes. We found that page size had no significant impact on the runtimes. Additionally, we did a sweep for 4, 8, and 16 segments for Greenplum but found the most benefits from 8 segments. Thus our default choice for Greenplum is to run 8 segments.Most results have been obtained for both warm cache and cold cache settings to better understand the impact of I/O on the overall runtime. In the case of a warm cache, training data tables for the real dataset are already in the buffer pool, whereas part of the synthetic datasets reside in the buffer pool before query execution. For the cold cache setting, none of the training data tables reside in the buffer pool.

7.1 End-to-End Performance

Real Datasets. Figures 9a and 9b illustrate the end-to-end performances of MADlib on PostgreSQL, MADlib on Greenplum, and DAnA, for warm and cold cache, respectively. These figures show individual workloads on the x-axis and speedup on the y-axis, the last bar representing the geometric mean (geomean) of the workloads. On average, DAnA provides 11.3× and 7.6× end-to-end speedup over PostgreSQL and 5.4× and 4.1× speedup over 8-segments of Greenplum for real datasets in warm and cold cache setting, respectively. The benefits diminish for cold cache, as the I/O time adds significantly to the runtime in all the systems. The speedup is limited given that this overhead cannot be parallelized.

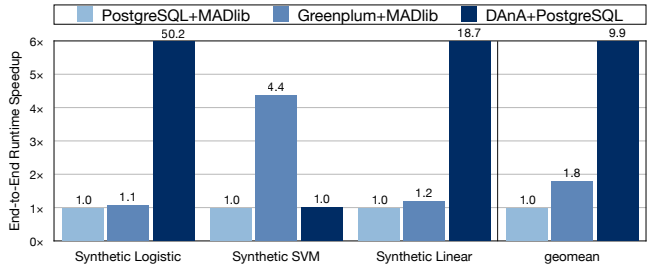


(a) End-to-End Performance for Warm Cache

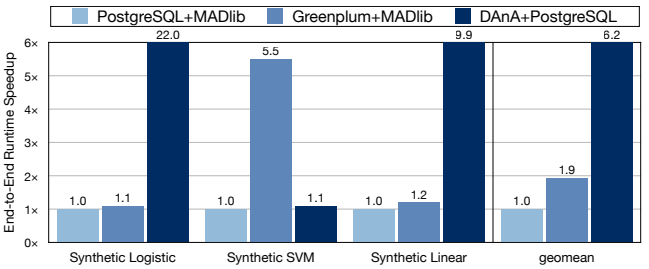


(b) End-to-End Performance for Cold Cache

Figure 9: End-to-end runtime performance comparison for real-world datasets with PostgreSQL +MADlib as baseline.



(a) End-to-End Performance for Warm Cache



(b) End-to-End Performance for Cold Cache

Figure 10: End-to-end runtime performance comparison for synthetic datasets with PostgreSQL +MADlib as baseline.

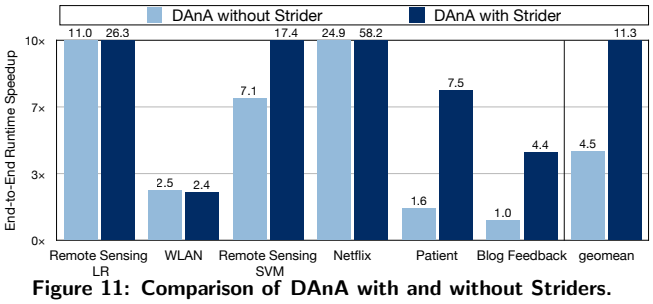


Figure 11: Comparison of DAnA with and without Striders.

The maximum speedup is obtained by Netflix, 58.2× in warm cache and 57.8× in cold cache case. Netflix workload runs low rank matrix factorization, the aim of which is to generate an approximating matrix with reduced rank given an input matrix. This algorithm generally has a large model topology with copious amounts of parallelism that can be exploited by DAnA's accelerator. In constrast, WLAN and Patient see the smallest speedups, with WLAN experiencing only 2.4× speedup in both warm cache and cold cache cases. These workloads have a relatively large number of features but relatively fewer training data tuples. This makes execution engine time disproportionately higher than the access engine time. Thus, the benefits of pipelining between both engines diminishes, and execution becomes the bottleneck.

Synthetic Datasets. Figure 10a and 10b depict end-to-end performance comparison for synthetic datasets across our three systems. Datasets Synthetic Logistic, Synthetic SVM, and Synthetic Linear take 24, 31, and 41 transfers, respectively, to bring the entire data inside the 128 MB buffer pool once. Across all synthetic workloads, DAnA achieves an average speedup of 9.9× and 6.2× for warm and cold cache, respectively. Comparing with 8 segment Greenplum, DAnA observes a gain of 5.5× for warm cache and 3.2× for cold cache. These results show the efficacy of the multi-threading employed by DAnA's execution engine in comparison to the scale-out Greenplum engine. DAnA's Synthetic Logistic and Synthetic Linear achieve much higher speedups than Synthetic SVM. This anomaly can be attributed to the high I/O time incurred by Synthetic SVM compared to its compute time. As the compute time is significantly smaller than the I/O time for Synthetic SVM, the accelerator frequently stalls for the buffer pool page replacement to finish. On the other hand, Synthetic Logistic and Synthetic Linear incur similar compute and I/O times, hence the execution engines can extract performance benefits.

Evaluating Striders. A crucial part of DAnA's hardware acceleration is the direct integration with the buffer pool via *Striders*. To evaluate the effectiveness of using *Striders*, we create an alternate design where DAnA's execution engines are fed by the CPU. In this case, the CPU is responsible to transform the data and send it to DAnA's execution engines. Figure 11 compares the end-to-end runtime of DAnA with and without *Striders* using a baseline of PostgreSQL +MADlib in the warm cache setting. DAnA with and without *Striders* achieves, on average, 11.3× and 4.5× speedup in comparison to the PostgreSQL MADlib setting, respectively. As we can see, even though raw application hardware acceleration has its benefits, integrating *Striders* to directly interface with the database engine amplifies those performance benefits by 2.5×. The *Striders* both bypass the bottlenecks in the memory subsystem of CPUs and provide an on-chip opportunity to intersperse the tasks of the access and execution engines.

The above evaluation demonstrates the effectiveness of DAnA and *Striders* to integrate with PostgreSQL.

7.2 Performance Sensitivities

This section aims to understands the impact of multi-threading within the accelerator and Greenplum on the overall performance of DAnA and MADlib, respectively.

Multi-threading in hardware. DAnA supports a multi-threaded execution engine architecture, with each thread running a version of the update rule. We perform a sensitivity analysis by varying the number of threads instantiated on the final accelerator. Specifically, we increment the number of parallel instances of the update rule, specified by the merge coefficient in the UDF, to increase the number of accelerator threads. For instance, with a merge coefficient of 2, the hardware generator does not instantiate more than 2 threads

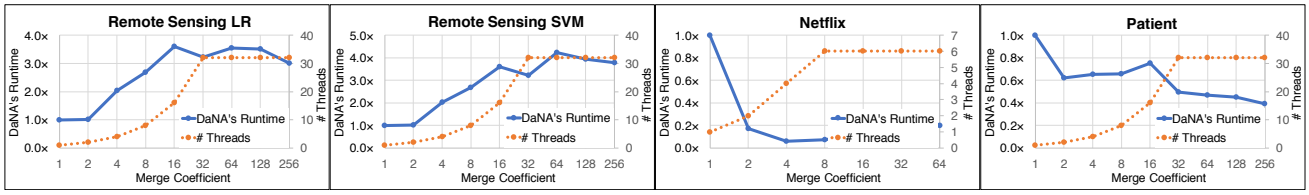


Figure 12: Runtime performance and # of threads obtained by varying the merge coefficient, compared to a single-threaded baseline.

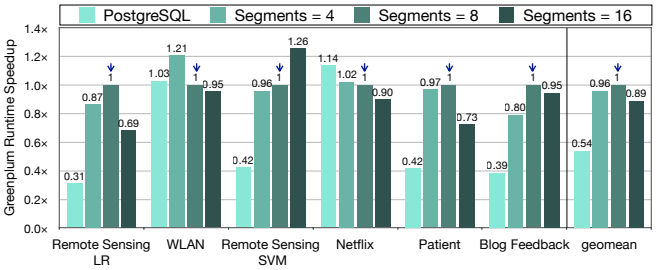


Figure 13: Multi-threading with Greenplum.

to avoid wasting FPGA resources. On the other hand, a large merge coefficient, such as 256, does not necessarily mean 256 threads, as the FPGA may not have enough resources. Therefore, the hardware generator strikes a balance between the resources allocated to a single thread vs. number of threads to obtain the maximum throughput. As depicted in Figure 12, each plot shows DAnA's accelerator runtime (access engine + execution engine) compared to single-threaded runtime. We plot both the runtime comparison and how the thread count scales with the merge coefficient. Each workload runs as many epochs as required to maintain the same final training error. For all the workloads except Netflix, the number of threads stagnate at 32, as the Arria10 FPGA cannot support more AUs and ACs. For Netflix, the maximum thread count supported by the FPGA are 6. Its performance declines when the number of threads is increased due to fact that large portion of operations in the LRMF algorithm are matrix-matrix multiplications. Therefore, even a single instance of the update rule has enough parallelism. Increasing the number of threads does not provide benefit, as less ACs can be allocated to a single thread and results from different threads need to be aggregated.

Multi-threading in Greenplum. As depicted in Figure 9, for real datasets, the default configuration (8-segment) Greenplum provides 2.1× (warm cache) and 1.8× (cold cache) higher speedups than its PostgreSQL counterpart, which is running a single thread. However, as Figure 13 shows, 8-segment Greenplum performs the best, and the performance does not scale as the segments increase.

8. RELATED WORK

Hardware acceleration for data management. Accelerating database operations is a popular research direction. In contrast, we focus on accelerating in-RDBMS advanced analytics through ML. Prior FPGA-based solutions aim to accelerate DBMS operations (some portion of the query) [3, 4, 21, 22, 38], such as join and hash. LINQits [38] accelerates database queries but does not focus on machine learning. For instance, Centaur [3] dynamically decides which particular operators in a MonetDB [39] query plan can be executed on FPGA and creates a pipeline between FPGA and CPU. Another work [22] uses FPGAs to provide a robust hashing mechanism to accelerate data partitioning in database engines. In the GPU realm, HippogriffDB [20] aims to balance the I/O and GPU bandwidth by compressing the data that is transferred to GPU. This work also does not deal with advanced analytics. Support for in-database analytics and *Striders* set this work apart from the aforementioned literature, which does not focus on providing components that integrate FPGAs within a RDBMS engine and machine learning.

Hardware acceleration for advanced analytics. There is an increasing number of hardware accelerators that target machine learning [5, 19, 23, 40] and deep neural networks [41–45]. These works either only focus on a fixed set of algorithms or do not offer the reconfigurability of the architecture. Among these, several works [16, 17, 46] provide frameworks to automatically generate hardware accelerators for stochastic gradient descent. However, none of these works provide a hardware structure or software components to embed FPGAs within RDBMS engines. DAnA's Python DSL builds upon the mathematical language in the prior work [5, 17]. However, the integration with both conventional (Python) and data access (SQL) languages provides a significant extension by enabling support for UDFs which include general iterative update rules, merge functions, and convergence functions.

In-Database advanced analytics. Recent work at the intersection of databases and machine learning are extensively trying to facilitate efficient in-database analytics and have built frameworks and systems to realize it [7, 13, 14, 27, 47–51] (see [52] for a survey of various methods and systems). DAnA takes a step forward and exposes FPGA acceleration for in-Database analytics by providing a specialized component,*Striders*, that directly integrates with the database with the aim to alleviate some of the shortcomings of traditional Von-Neumann architecture in general purpose compute systems. Past work in Bismarck [7] provides a unified architecture for in-database analytics, facilitating UDFs as a convenient interface for the analyst to describe their desired analytics models. However, unlike DAnA, Bismarck lacks the hardware acceleration backend and support for general iterative optimization algorithms.

9. CONCLUSION

This paper aims to bridge the power of well established and extensively researched means of structuring, defining, protecting, and accessing data, i.e. relational database management systems, with the low-power FPGA accelerators for compute-intensive advanced data analytics. DAnA provides the initial bridge between these two paradigms and empowers data scientists, with no knowledge about hardware design, to use accelerators within their current in-database analytics procedures.

10. ACKNOWLEDGMENTS

We thank Yannis Papakonstantinou and Balaji Chandrasekaran for insightful discussions. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, CCF#1553192, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, and Qualcomm.

11. REFERENCES

[1] Gartner Report on Analytics. gartner.com/it/page.jsp?id=1971516.
[2] SAS Report on Analytics. sas.com/reg/wp/corp/23876.

[3] M. Owaida, D. Sidler, K. Kara, and G. Alonso. Centaur: A framework for hybrid cpu-fpga databases. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 211–218, April 2017.

[4] David Sidler, Muhsen Owaida, Zsolt István, Kaan Kara, and Gustavo Alonso. doppiodb: A hardware accelerated database. In *27th International Conference on Field Programmable Logic and Applications, FPL 2017, Ghent, Belgium, September 4-8, 2017*, page 1, 2017.

[5] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kim, and Hadi Esmaeilzadeh. TABLA: A unified template-based framework for accelerating statistical machine learning. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016. *(Distinguished Paper Award)*.

[6] Andrew R. Putnam, Dave Bennett, Eric Dellinger, Jeff Mason, and Prasanna Sundararajan. CHiMPS: A high-level compilation flow for hybrid CPU-FPGA architectures. In *FPGA*, 2008.

[7] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336. ACM, 2012.

[8] Yu Cheng, Chengjie Qin, and Florin Rusu. GLADE: Big Data Analytics Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 697–700. ACM, 2012.

[9] Amazon web services postgresql.

[10] Azure sql database.

[11] Oracle Data Mining.

[12] Oracle R Enterprise.

[13] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. MAD Skills: New Analysis Practices for Big Data. *Proc. VLDB Endow.*, 2(2):1481–1492, August 2009.

[14] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, August 2012.

[15] Microsoft SQL Server Data Mining.

[16] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[17] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learnng. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, October 2017.

[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pages 609–622. IEEE, 2014.

[19] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. In *ASPLOS*, 2015.

[20] Jing Li, Hung-Wei Tseng, Chunbin Lin, Yannis Papakonstantinou, and Steven Swanson. Hippogriffdb: Balancing i/o and gpu bandwidth in big data analytics. *Proc. VLDB Endow.*, 9(14):1647–1658, October 2016.

[21] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2(1):910–921, August 2009.

[22] Kaan Kara, Jana Giceva, and Gustavo Alonso. Fpga-based data partitioning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 433–445. ACM, 2017.

[23] Kaan Kara, Dan Alistarh, Gustavo Alonso, Onur Mutlu, and Ce Zhang. Fpga-accelerated dense linear machine learning: A precision-convergence trade-off. *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 160–167, 2017.

[24] Amazon EC2 F1 instances: Run custom FPGAs in the AWS cloud, 2017.

[25] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James R. Larus, Eric Peterson, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *to appear in the Communications of the ACM Research Highlights (invited)*, 2015.

[26] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, Richard C. Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. *CoRR*, abs/1704.04760, 2017.

[27] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen Wright. Hogwild : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*, 2011.

[28] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. Learning generalized linear models over normalized data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1969–1984. ACM, 2015.

[29] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and

Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, October 2016.

[30] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. On parallelizability of stochastic gradient descent for speech dnns. In *ICASSP*, 2014.

[31] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, 2010.

[32] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research*, 13(Jan):165–202, 2012.

[33] J. Langford, A.J. Smola, and M. Zinkevich. Slow learners are fast. In *NIPS*, 2009.

[34] Gideon Mann, Ryan McDonald, Mehryar Mohri, Nathan Silberman, and Daniel D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, 2009.

[35] Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey. Distributed deep learning using synchronous stochastic gradient descent. *arXiv:1602.06709 [cs]*, 2016.

[36] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. In *ICLR Workshop Track*, 2016.

[37] A. Frank and A. Asuncion. UCI machine learning repository, 2010.

[38] Eric S. Chung, John D. Davis, and Jaewon Lee. LINQits: Big data on little clients. In *ISCA*, 2013.

[39] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[40] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[41] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, 2015.

[42] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: efficient inference engine on compressed deep neural network. *CoRR*, abs/1602.01528, 2016.

[43] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.

[44] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *ISCA*, 2016.

[45] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, Jose Miguel Hernandez-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ISCA*, 2016.

[46] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, Oct 2016.

[47] Boriana L. Milenova, Joseph Yarmus, and Marcos M. Campos. Svm in oracle database 10g: Removing the barriers to widespread adoption of support vector machines. In *VLDB*, 2005.

[48] Daisy Zhe Wang, Michael J. Franklin, Minos Garofalakis, and Joseph M. Hellerstein. Querying probabilistic information extraction. *Proc. VLDB Endow.*, 3(1-2):1057–1067, September 2010.

[49] Michael Wick, Andrew McCallum, and Gerome Miklau. Scalable probabilistic databases with factor graphs and mcmc. *Proc. VLDB Endow.*, 3(1-2):794–804, September 2010.

[50] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 13–24, New York, NY, USA, 2013. ACM.

[51] M. Levent Koc and Christopher Ré. Incrementally maintaining classification using an rdbms. *Proc. VLDB Endow.*, 4(5):302–313, February 2011.

[52] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1717–1722, New York, NY, USA, 2017. ACM.