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 machine learning, databases, and hardware design. Programmable accelerators are making their way into each of these areas independently. As such, there is a void of solutions that enables hardware acceleration at the intersection of these disjoint fields. This paper sets out to be the initial step towards a unifying solution for in-**D**atabase **A**cceleration of Adva**n**ced **A**nalytics (DAnA). 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 advanced analytics queries to an FPGA accelerator. The accelerator implementation is generated for a User Defined Function (UDF), expressed as a part of an SQL query using a Python-embedded Domain-Specific Language (DSL). To realize an efficient in-database integration, DAnA accelerators contain a novel hardware structure, *Striders*, that directly interface with the buffer pool of the database. *Striders* extract, cleanse, and process the training data tuples that are consumed by a multi-threaded FPGA engine that executes the analytics algorithm. We integrate 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, 8.3× end-to-end speedup for real datasets, with a maximum of 28.2×. Moreover, DAnA-enhanced PostgreSQL is, on average, 4.0× faster 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.

PVLDB Reference Format:
Divya Mahajan, Joon Kyung Kim, Jacob Sacks, Adel Ardalan, Arun Kumar, and Hadi Esmaeilzadeh. In-RDBMS Hardware Acceleration of Advanced Analytics. *PVLDB*, 11(11): 1317-1331, 2018.
DOI: https://doi.org/10.14778/3236187.3236188

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 queries towards advanced analytics, especially machine learning (ML), 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) enterprise in-database analytics [3, 4] , (2) modern hardware acceleration platforms [5, 6], and (3) programming paradigms which facilitate the use of analytics [7, 8].

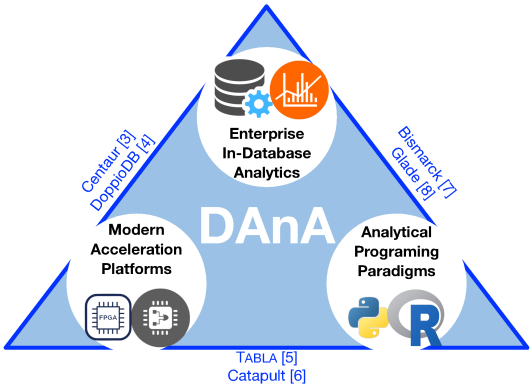


Figure 1: DAnA represents the fusion of three research directions, in contrast with prior works [3–5, 7–9] that merge two of the areas.

The database industry is investing in the integration of ML algorithms within RDBMSs, both on-premise and cloud-based [10, 11]. This integration enables enterprises to exploit ML without sacrificing the auxiliary benefits of an RDBMS, such as transparent scalability, access control, security, and integration with their business intelligence interfaces [7, 8, 12–18]. Concurrently, the computer architecture community is extensively studying the integration of specialized hardware accelerators within the traditional compute stack for ML applications [5, 9, 19–21]. 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 [22] and reconfigurable hardware, such as Field Programmable Gate Arrays (FPGAs) [3,4,23–25], for relational operations. Furthermore, cloud service providers like Amazon AWS [26], Microsoft Azure [27], and Google Cloud [28], are also offering high-performance specialized platforms due to the potential gains from modern hardware. Finally, the applicability and practicality of both in-database analytics and hardware acceleration hinge upon exposing a high-level interface to the user. This triad of research areas are currently 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 produc-

Proceedings of the VLDB Endowment, Vol. 11, No. 11

DOI: https://doi.org/10.14778/3236187.3236188

tivity of data scientists who can benefit from in-database hardware acceleration for analytics. Consider the following example.

Example 1 *A marketing firm uses the Amazon Web Services (AWS) Relational Data Service (RDS) to maintain a PostgreSQL database of its customers. A data scientist in that company forecasts the hourly ad serving load by running a multi-regression model across a hundred features available in their data. Due to large training times, she decides to accelerate her workload using FPGAs on Amazon EC2 F1 instances [26]. Currently, this requires her to learn a hardware description language, such as Verilog or VHDL, program the FPGAs, and go 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 [19, 29, 30]. However, these solutions do not interface with or support RDBMSs, requiring her to manually extract, copy, and reformat her large dataset.*

To overcome the aforementioned roadblocks, we devise DAnA, a *cohesive stack that enables deep integration between FPGA acceleration and in-RDBMS execution of advanced analytics*. DAnA exposes a high-level programming interface for data scientists/analysts based on conventional languages, such as 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 enables the user to express RDBMS User-Defined Functions (UDFs) using familiar practices of Python and SQL. The user provides their ML algorithm as an update rule using a Python-embedded Domain Specific Language (DSL), while an SQL query specifies data management and retrieval. To convert this high level ML specification into an accelerated execution without manual intervention, we develop a comprehensive stack. Thus, DAnA is a solution that breaks the algorithm-data pair into software execution on the RDBMS for data retrieval and hardware acceleration for running the analytics algorithm.

With respect to the second challenge, DAnA offers *Striders*, which avoid the inefficiencies of conventional Von-Neumann CPUs for data handoff by seamlessly connecting the RDBMS and FPGA. *Striders* directly feed the data to the analytics accelerator by walking Jacthe RDBMS buffer pool. Circumventing the CPU alleviates the cost of data transfer through the traditional memory subsystem. These *Striders* are backed with an Instruction Set Architecture (ISA) to ensure programmability and ability to cater to the variations in the database page organization and tuple length across different algorithms and training datasets. They are designed to ensure *multi-threaded acceleration* of the learning algorithm to amortize the cost of data accesses across concurrent threads. DAnA automatically generates the architecture of these accelerator threads, called execution engines, that selectively combine a Multi-Instruction Multi-Data (MIMD) execution model with Single-Instruction Multi-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 available FPGA resources. As such, this paper makes the following technical contributions:

- Merges three disjoint research areas to enable transparent and efficient hardware acceleration for in-RDBMS analytics. Data scientists with no hardware design expertise can use DAnA to harness hardware acceleration without manual data retrieval and extraction whilst retaining familiar programming environments.
- Exposes a high-level programming interface, which combines SQL UDFs with a Python DSL, to jointly specify training data and computation. This unified abstraction is backed by an extensive compilation workflow that automatically transforms the specification to an accelerated execution.
- Integrates an FPGA and an RDBMS engine through *Striders* that are a novel on-chip interfaces. *Striders* bypass CPU to directly access the training data from the buffer pool, transfer this data onto 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. This model exposes a domain specific instruction set architecture that offers automation while providing efficiency.

We prototype DAnA with PostgreSQL to automatically accelerate the execution of several popular ML algorithms. Through a comprehensive experimental evaluation using real-world and synthetic datasets, we compare DAnA against the popular in-RDBMS ML toolkit, Apache MADlib [15], on both PostgreSQL and its parallel counterpart, Greenplum. Using Xilinx UltraScale+ VU9P FPGA, we observe DAnA generated accelerators provide on average 8.3× and 4.0× end-to-end runtime speedups over PostgreSQL and Greenplum running MADlib, respectively. An average of 4.6× of the speedup benefits are obtained through *Striders*, as they effectively bypass the CPU and its memory subsystem overhead.

2. BACKGROUND

Before delving into the details of DAnA, this section discusses the properties of ML algorithms targeted by our holistic framework.

2.1 Iterative Optimization and Update Rules

During training, a wide range of supervised machine learning algorithms go through a cyclic process that demand constant iteration, tuning, and improvement. These algorithms use optimization procedures 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.

Example. Given a set of N pairs of $\{(x_1, y_1^*), ..., (x_N, y_N^*)\}$ constituting the training data, the goal is to find a hypothesis function $h(w^{(t)}, x)$ that can accurately map $x \rightarrow y$. The equation below specifies an entire update rule, where $l(w^{(t)}, x_i, y_i^*)$ is the loss function that signifies the error between the output y^* and the predicted output estimated by a hypothesis function $h(w^{(t)}, x_i)$ for input x.

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

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 $(y = h(w, x))$ and loss function vary substantially across different ML algorithms, the optimization algorithm that iteratively minimizes the loss function remains fixed. As such, the two required components are the hypothesis function to define the machine learning algorithm and an optimization algorithm that 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 [31–37]. This offers a unique opportunity for DAnA to rapidly consume data pages brought on-chip by

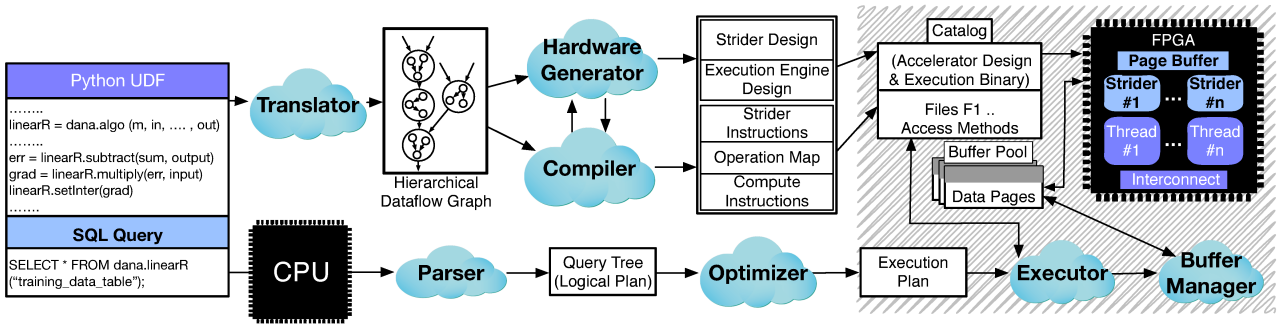


Figure 2: Overview of DAnA, that integrates FPGA acceleration with the RDBMS engine. The Python-embedded DSL is an interface to express the ML algorithm that is converted to hardware architecture and its execution schedules (stored in the RDBMS catalog). The RDBMS engine fills the buffer pool. FPGA Striders directly access the data pages to extract the tuples and feed them to the threads. Shaded areas show the entangled components of RDBMS and FPGA working in tandem to accelerate in-database analytics.

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 can be 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 rule via the DAnA DSL described in §4.1.

In addition to providing a background on properties of machine learning algorithms targeted by DAnA, Appendix A in our tech report (http://act-lab.org/artifacts/dana/addendum.pdf) provides a brief overview on Field Programmable Gate Arrays (FPGAs). It provides details about the reconfigurability of FPGAs and how they offer a potent solution for hardware acceleration.

2.2 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 need 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 criteria through *Striders*, its database-aware reconfigurable memory interface, 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-embedded 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 that directly interface with the RDBMS engine's buffer pool. The next section provides an overview of DAnA.

3. DAnA WORKFLOW

Figure 2 illustrates DAnA's integration within 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 both the ML algorithm and page specifications of the RDBMS engine. To run the hardware accelerated UDF on her training data, the user provides a SQL query. 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 the query. As shown in Figure 2, the RDBMS catalog is shared by the database engine and the FPGA. 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 data 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, database page layout, FPGA} triad. Each component of DAnA 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. It also expects a merge function that specifies how to process multiple tuples in parallel and aggregate the resulting ML 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 converted to a final synthesizable FPGA accelerator.

Translator. The user provided UDF is converted into a *h*ierarchical 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 vector 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 machine learning architecture. To target a wide range of ML algorithms, DAnA offers a parametric reconfigurable hardware design solution that is hand optimized by expert hardware designers as described in § 5. The hardware interfaces with the database engine through a specialized structure called *Striders*, that extract high-performance, and provide low-energy computation. *Striders* eliminate CPU from the data transformation process by directly interfacing with database's buffer pool to extract the training data pages. They process data at a page granularity to amortize the cost of per-tuple data transfer from memory to the FPGA. To exploit this 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 by DAnA's compiler and hardware generator in accordance to the FPGA specifications, database page layout, and the analytics function.

Instruction Set Architectures. Both *Striders* and the execution engine can be programmed using their respective Instruction Set

Architectures (ISAs). The *Strider* 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 execution engine's ISA describes the operation flow required to run the analytics algorithm in selective SIMD mode.

Compiler and hardware generator. DAnA's compiler and hardware generator ensure compatibility between the *h*DFG and the hardware accelerator. 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* to generate the final FPGA synthesizable accelerator. The compiler converts the database page configuration into a set of *Strider* instructions that process the page and tuple headers and transform user data into a floating point format. Additionally, the compiler generates a static schedule for the accelerator, a map of where each operation is performed, and execution engine instructions.

As described above, providing flexibility and reconfigurability of hardware accelerators for advanced analytics is a challenging but pertinent problem. DAnA is a multifaceted solution that untangles these challenges one by one.

4. FRONT-END INTERFACE FOR DAnA

DAnA's DSL provides an entry point for data scientists to exploit hardware acceleration for in-RDBMS advanced analytics. This section 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. This section also explains how a UDF defined in this DSL is translated into an intermediate representation, i.e., in this case a *h*ierarchical DataFlow Graph (*h*DFG).

4.1 Programming For DAnA

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

1. Incorporates language constructs commonly seen in a wide class of supervised learning algorithms.
2. Supports expression of any iterative update rule, not just variants of gradient descent, whilst conforming to the DSL constructs.
3. Segregates algorithmic specification from hardware-dependent implementation.

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 and provide 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 declarations delineate the semantics of the data types used in the ML algorithm. The DSL supports the following data declarations: input, output, inter, model, and meta. Each variable can be declared by specifying its type and dimensions. A variable is an implied scalar if no dimensions are specified. Once the analyst imports the dana package, she can express the required variables. The code snippet below declares a multi-dimensional ML model of size [5][2] using dana.model construct.

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

In addition to dana.model, the user can provide dana.input and dana.output to express a single input-output pair in the training dataset. The user can specify meta variables using dana.meta, the value of which remains constant throughout execution. As such, meta variables can be directly sent to the FPGA before algorithm execution. All variables used for a particular algorithm are linked to an algo construct.

Table 1: Language constructs of 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
	sigma, 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

```
algorithm = dana.algo (mo, in, out)
```

The algo component allows the user to link together the three functions – update rule, merge, and terminator – of a single UDF. Additionally, the analyst can use untyped intermediate variables, which are automatically labeled as dana.inter by DAnA's backend.

Mathematical operations. The 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 and its output is automatically inferred by DAnA's translator (as discussed in § 4.4) in accordance to the operands' dimensions. Group operations, such as sigma, pi, norm, perform computation across elements. Sigma refers to summation, pi indicates product operator, and norm calculates the magnitude of a multidimensional vector. Group operations require the input operands and the grouping axis which is expressed as a constant and alleviates the need to explicitly specify loops. The underlying primary operation is performed on the input operands prior to grouping.

Built-in functions. The DSL provides four built-in functions to specify the merge condition, set the convergence criterion, and link the updated model variable to the algo component. The merge(x, int, "op") function is used to specify how multiple threads of the update rule are combined. Convergence is dictated either by a specifying fixed number of epochs (1 epoch is a single pass over the entire training data set) or a user-specified condition. Function setEpochs(int) sets the number of terminating epochs and setConvergence(x) frames termination based on a boolean variable x. Finally, the setModel(x) function links a DAnA variable (the updated model) to the corresponding algo component.

All the described language constructs are supported by DAnA's reconfigurable architecture, hence, can be synthesized on an FPGA. 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 below.

4.3 Linear Regression Example

Update rule. As the code snippet below illustrates, the data scientist first declares different data types and their corresponding dimensions. Then she defines the computations performed over these variables specific to linear regression.

```
#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)

#Gradient or Derivative of the Loss Function
s = sigma ( mo * in, 1)
er = s - out
grad = er * in

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

In this example, the update rule uses the gradient of the loss function. The gradient descent optimizer updates the model in the negative direction of the loss function derivative ($\frac{\partial(l)}{\partial w^{(t)}}$). The analyst concludes with the setModel() function to identify the updated model, in this case mo_up.

Merge function. The merge function facilitates multiple concurrent threads of the update rule on the FPGA accelerator by specifying the functionality at the point of merge.

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

In the above merge function, the intermediate grad variable has been combined using addition, and the merge coefficient (merge_coef) specifies the batch size. DAnA's compiler implicitly understands that the merge function is performed before the gradient descent optimizer. 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.

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

The mo_up is calculated by each thread for tuples in its batch separately and consecutively averaged. Thus, DAnA's DSL provides the flexibility to create different learning algorithms without requiring any modification to the update rule by specifying different merge points. In the above example, the first definition of the merge function creates a linear regression running batched gradient descent optimizer, whereas, the second definition corresponds to a parallelized stochastic gradient descent optimizer.

Convergence function. The user also provides the termination criteria. As shown in the code snippet below, the convergence checks for the conv variable, which, if true, terminates the training. Variable conv compares the Euclidean norm of grad with a conv_factor constant.

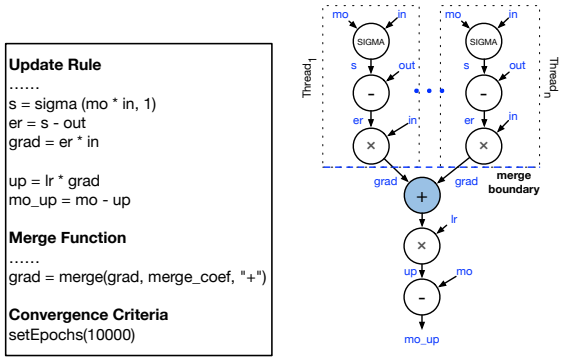
```
convergenceFactor = dana.meta (0.01)
n = norm(grad , i)
conv = n < convergenceFactor
linear.setConvergence(conv)
```

Alternatively, the number of epochs can be used for convergence using the syntax linearR.setEpochs(10000).

Query. A UDF comprising the update rule, merge function, and convergence check describes the entire analytics algorithm. The linearR UDF can then be called within a query as follows:

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

Currently, for high efficiency and low latency, DAnA's DSL and compiler do not support dynamic variables, as the FPGA and CPU do not interchange runtime values and only interact for data handoff. DAnA only supports variable types which either have been explicitly instantiated as DAnA's data declarations, or inferred as intermediate variables (dana.inter) by DAnA's translator. 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. In the next section we discuss the process of converting this UDF into a *h*DFG.



(a) Code snippet (b) Hierarchical DFG

Figure 3: Translator-generated *h*DFG for the linear regression code snippet expressed in DAnA's DSL.

4.4 Translator

DAnA's translator is the front-end of the compiler, which converts the user-provided UDF to a *h*ierarchical DataFlow Graph (*h*DFG). The *h*DFG represents the coalesced update rule, merge function, and convergence check whilst maintaining the data dependencies. Each node of the *h*DFG represents a multi-dimensional operation, which can be decomposed into smaller atomic sub-nodes. An atomic sub-node is a single operation performed by the accelerator. The *h*DFG transformation for the linear regression example provided in the previous section is shown in Figure 3.

The aim of the translator is to expose as much parallelism available in the algorithm to the remainder of the DAnA workflow. This includes parallelism within a single instance of the update rule and among different threads, each running a version of the update rule. To accomplish this, the translator (1) maintains the function boundaries, especially between the merge function and parallelizable portions of the update rule, and (2) automatically infers dimensionality of nodes and edges in the graph.

The merge function and convergence criteria are performed once per epoch. In Figure 3b, the colored node represents the merge operation that combines 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; thus, they can be readily parallelized across multiple threads. To generate the *h*DFG, the translator first infers the dimensions of each operation node and its output edge(s). For basic operations, if both the inputs have same dimensions, it translates into an element by element operation in the hardware. In case the inputs do not have same dimensions, the input with lower dimension is logically replicated, and the generated output possess the dimensions of the larger input. Nonlinear operations have a single input that determines the output dimensions. For group operations, the output dimension is determined by the axis constant. For example, a node performing sigma(mo * in, 2), where variables mo and in are matrices of sizes [5][10] and [2][10], respectively, generates a [5][2] output.

The information captured within the *h*DFG allows the hardware generator to configure the accelerator architecture to optimally cater for its operations. Resources available on the FPGA are distributed on-demand within and across multiple threads. Furthermore, DAnA's compiler maps all the operations 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 the reconfigurable architecture for the FPGA (*Strider* and execution engine).

5. HARDWARE DESIGN FOR IN-DATABASE ACCELERATION

DAnA employs a parametric accelerator architecture comprising a *multi-threaded access engine* and a *multi-threaded execution engine*, shown in Figure 4. Both engines have their respective custom Instruction Set Architectures (ISA) to program their hardware designs. 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. The access and execution engines are configured according to the page layout and UDF specification, 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 that are processed by the execution engine. Figure 5 shows a detailed diagram of this access engine. The access engine uses the Advanced Extensible Interface (AXI) interface to transfer the data to and from the FPGA, the shifters properly align the data, and the *Striders* unpack the database pages. AXI interface is a type of Advanced Microcontroller Bus Architecture open-standard, on-chip interconnect specification for system-on-a-chip (SoC) designs. It is vendor agnostic and standardized across different hardware platforms. The access engine uses this interface to transfer uncompressed database pages to page buffers and configuration data to configuration registers. Configuration data comprises *Strider* and execution engine instructions and 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 incorporates 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* process database data at a page level granularity. Training data is written to multiple page buffers, where each buffer stores one database page at a time and has access to its personal *Strider*. Alternatively, each tuple could have been extracted from the page by the CPU and sent to the FPGA for consumption by the execution engine. 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 data extraction would have a significant overhead due to the handshaking between CPU and FPGA. Offloading tuple extraction to the accelerator using *Striders* provides a unique opportunity to dynamically interleave unpacking of 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 the previous. 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 and stores it in the configuration registers. The information includes offsets, such as the beginning and size of each tuple, which is either located or computed from the data in the header. This auxiliary page information is used to trace the tuple addresses and read the corresponding data from the page buffer. After each page buffer, the shifter ensures alignment of the tuple data for the *Strider*. From the tuple data, its header is processed to extract

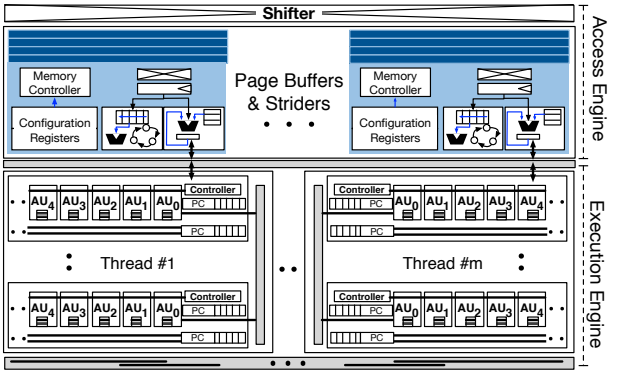


Figure 4: Reconfigurable accelerator design in its entirety. The access engine reads and processes the data via its Striders, while the execution engine operates on this data according to the UDF.

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	

and route the 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 its instructions that depend on the page layout and page size of the target RDBMS. We next discuss the novel ISA to program these *Striders*.

5.1.2 Instruction Set Architecture for STRIDERS

We devise a novel fixed-length Instruction Set Architecture (ISA) for the *Striders* that can target a range of RDBMS engines, such as PostgreSQL and MySQL (innoDB), that have similar backend page layouts. An uncompressed page from these RDBMSs, once transferred to the page buffers, can be read, extracted, and cleansed using this ISA, which comprises light-weight instructions specialized for pointer chasing and data extraction. Each *Strider* is programmed with the same instructions but operates on different pages. These instructions are generated statically by the compiler.

Table 2 illustrates the 10 instructions of this ISA. Every instruction is 22 bits long, comprising a unique operation code (opcode) as identification. The remaining bits are specific to the opcode. Instructions **Read Bytes** and **Write Bytes** are responsible for reading and writing data from the page buffer, respectively. The ISA provides the flexibility to extract data at byte and bit granularity using the **Extract Byte** and **Extract Bit** instructions. The **Clean** instruction can remove parts of the data not required by the execution engine. Conversely, the **Insert** instruction can add bits to the data, such as NULL characters and auxiliary information, which is particularly useful when the page is to be written back to memory. Basic math operations, **Add**, **Subtract**, and **Multiply**, allow calculation of tuple sizes, byte offsets, etc. Finally, the **Bentr** and **Bexit** branch instructions are used to specify jumps or loop exits, respectively. This feature invariably reduces the instruction footprint as repeated patterns can be succinctly expressed using branches while enabling loop exits that depend on a dynamic runtime variable.

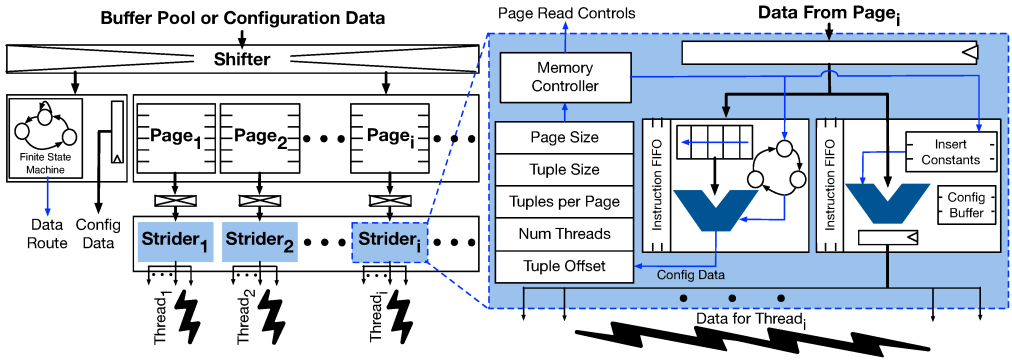


Figure 5: Access engine design uses Striders as the main interface between the RDBMS and execution engines. Uncompressed data pages are read from the buffer pool and stored in on-chip page buffers. Each page has a corresponding strider to extract the tuple data.

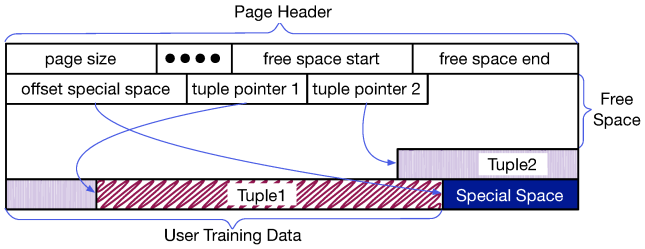


Figure 6: Sample page layout similar to PostgreSQL.

An example page layout representative of PostgreSQL and MySQL is illustrated in Figure 6. Such layouts are divided into a page header, tuple pointers, and tuple data and can be processed using the following assembly code snippet written in *Strider* ISA.

```
\\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
```

Each line in the assembly code is identified by its instruction name (opcode) and its corresponding fields. The first four assembly instructions process the page header to obtain the configuration information. For example, the **(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 is a configuration register, and %t is a temporary register. Next, the first tuple pointer is read to extract the byte-offset and length (bytes) of the tuple. Only the first tuple pointer is processed, as all the training data tuples are expected to be identical. Each corresponding tuple is processed by adding the tuple size to the previous offset to generate the page address. This address is used to read the data from the page buffer, which is then cleansed by removing its auxiliary information. The above step is repeated for each tuple using the **bentr** and **bexit** instructions. The loop is exited when the tuple offset address reaches the free space in the page. Finally, cleaned data is sent to the execution engines.

5.2 Execution Engine Architecture

The execution engines execute the *h*DFG of the user provided UDF using the *Strider* processed training data pages. More and more database pages can now be stored on-chip as the BRAM capacity is rapidly increasing with the new FPGAs such as Arria 10 that offers 7 MB and UltraScale+ VU9P with 44 MB of memory. 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 both regular and irregular 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 architecturally identical and perform the same computations on different training data 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) 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 caters to commonly seen compute operations in data analytics.

Analytic cluster. An Analytic Cluster (AC), shown in Figure 7a, is a collection of AUs designed to reduce the data transfer latency between them. Thus, *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). 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 line topology bus. 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 an AC, each

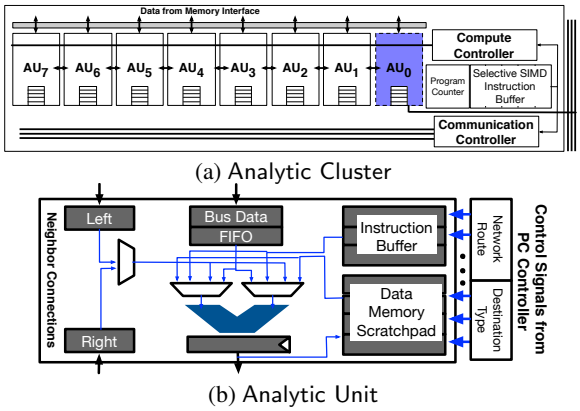


Figure 7: (a) Single analytic cluster comprising analytic units operating in a selective SIMD mode and (b) an analytic unit that is the pipelined compute hub of the architecture.

performing instructions independently. Data sharing among ACs is made possible via a shared line topology inter-AC bus.

Analytic unit. The Analytic Unit (AU) shown in Figure 7b, is the basic compute element of the execution engine. It is tailored by the hardware generator to satisfy the mathematical requirements of the *h*DFG. Control signals are provided by the AC. Data for each operation can be read from the memory according to the source type of the AC instruction. 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 Arithmetic Logic Unit (ALU), that executes both basic mathematical operations and complicated non-linear operations, such as sigmoid, gaussian, and square root. The internals of the ALU are reconfigured according to the operations required by the *h*DFG. The ALU then sends its output to the neighboring AUs, the shared bus within the AC, and/or the memory as per the instruction.

Bringing the Execution Engine together. Results across the threads are combined via a computationally-enabled tree bus in accordance to the merge function. This bus has attached ALUs to perform computations on in-flight data. The pliability of the architecture enables DAnA to generate high-performance designs that efficiently utilize the resources on the FPGA for the given RDBMS engine and algorithm. The execution engine is programmed using its own novel ISA. Due to space constraints, the details of this ISA are added to the Appendix B of our tech report (http://act-lab.org/artifacts/dana/addendum.pdf).

6. BACKEND FOR DAnA

DAnA's translator, scheduler, and hardware generator together 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. Each node of the *h*DFG comprises of sub-nodes, where each sub-node is a single instruction in the execution engine. Thus, all the sub-nodes in the *h*DFG are scheduled and mapped to the final accelerator hardware design. The hardware generator outputs a single-threaded architecture for the operations of these sub-nodes and determines the number of threads to be instantiated. The scheduler then statically maps all operations to this architecture.

6.1 Hardware Generator

The hardware generator finalizes the parameters of the reconfigurable architecture (Figure 4) for the *Striders* and the execution engine. The hardware generator obtains the database page layout information, model, and training data schema from the DBMS catalog. 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. Using this information, the hardware generator distributes the resources among access and execution engine. Sizes of the DBMS page, model, and a single training data 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 determines the number of ACs. 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 the coefficient. To decide the allocation of resources to each thread vs. number of threads, we equip the hardware generator with a performance estimation tool that uses the static schedule of the operations for each design point to estimate its relative performance. It chooses the smallest and best-performing design point which strikes a balance between the number of cycles for data processing and transfer. Performance estimation is viable, as the *h*DFG does not change, there is no hardware managed cache, and the accelerator architecture is fixed during execution. Thus, there are no dynamic irregularities that hinder estimation. This technique is commensurate with prior works [5, 19, 30] that perform a similar restricted design space exploration in less than five minutes with estimates within 5% of the physical measurements.

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 each sub-node 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", i.e., all its predecessors have been scheduled, the compiler tries to place that operation with the goal to improve throughput. Elementary and non-linear operation nodes are spread across as many AUs as 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, an element-wise vector-vector multiplication, where each vector contains 16 scalar values will be scheduled across two ACs (8 AUs per ACs). Group operations exhibit data dependencies, hence, they are mapped to minimize the communication cost. After all the sub-nodes are mapped, the compiler generates the AC and AU micro-instructions.

The FPGA design, its schedule, operation map, and instructions 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 it with PostgreSQL and compare the end-to-end runtime performance of DAnA generated accelerators with a popular scalable in-database advanced analytics library, Apache MADlib [14, 15], for both PostgreSQL and Greenplum RDBMSs. We compare the end-to-end runtime performance

Table 3: Descriptions of datasets and machine learning models used for evaluation. Shaded rows are synthetic datasets.

Workloads	Machine Learning Algorithm	Model Topology	Training Data		
			# of Tuples	# 32KB Pages	Size (MB)
Remote Sensing	Logistic Regression, SVM	54	581,102	4,924	154
WLAN	Logistic Regression	520	19,937	1,330	42
Netflix	Low Rank Matrix Factorization	6040, 3952, 10	6,040	3,068	96
Patient	Linear Regression	384	53,500	1,941	61
Blog Feedback	Linear Regression	280	52,397	2,675	84
S\N Logistic	Logistic Regression	2,000	387,944	96,986	3,031
S\N SVM	SVM	1,740	678,392	169,598	5,300
S\N LRMF	Low Rank Matrix Factorization	19880, 19880, 10	19,880	50,784	1,587
S\N Linear	Linear Regression	8,000	130,503	130,503	4,078
S\E Logistic	Logistic Regression	6,033	1,044,024	809,339	25,292
S\E SVM	SVM	7,129	1,356,784	1,242,871	38,840
S\E LRMF	Low Rank Matrix Factorization	28002, 45064, 10	45,064	162,146	5,067
S\E Linear	Linear Regression	8000	1,000,000	1,027,961	32,124

Table 4: Xilinx Virtex UltraScale+ VU9P FPGA specifications.

FPGA Capacity		Frequency	BRAM Size	# DSPs
1,182 K LUTS	2,364 K Flip-Flops	150 MHz	44 MB	6,840

of these three systems. Next, we investigate the impact of *Striders* on the overall runtime of DAnA and how accelerator performance varies with the system parameters. Such parameters include the buffer page-size, number of Greenplum segments, multi-threading on the hardware accelerator, and bandwidth and compute capability of the target FPGA. We also aim to understand the overheads of performing analytics within RDBMS, thus compare MADlib+PostgreSQL with software libraries optimized to perform analytics outside the database. Furthermore, to delineate the overhead of reconfigurable architecture, we compare our FPGA designs with custom hard coded hardware designs targeting a single or fixed set of machine learning algorithms.

Datasets and workloads. Table 3 lists the datasets and machine learning models used to evaluate DAnA. These workloads cover a diverse range of machine learning algorithms, – Logistic Regression (Logistic), Support Vector Machines (SVM), Low Rank Matrix Factorization (LRMF), and Linear Regression (Linear). Remote Sensing, WLAN, Patient, and Blog Feedback are publicly available datasets, obtained from the UCI repository [38]. Remote Sensing is a classification dataset used by both logistic regression and support vector machine algorithms. Netflix is a movie recommendation dataset for LRMF algorithm. The model topology, number of tuples, and number of uncompressed 32 KB pages that fit the entire training dataset are also listed in the table. Publicly available datasets fit entirely in the buffer pool, hence impose low I/O overheads. To evaluate the performance of out-of-memory workloads, we generate eight synthetic datasets, shown by the shaded rows in Table 3. Synthetic Nominal (S\N) and Synthetic Extensive (S\E) datasets are used to evaluate performance with the increasing sizes of datasets and model topologies. Finally, Table 5 provides absolute runtimes for all workloads across our three systems.

Experimental setup. We use the Xilinx Virtex UltraScale+ VU9P as the FPGA platform for DAnA and synthesize the hardware at 150 MHz using Vivado 2018.2. Specifications of the FPGA board are provided in Table 4. DAnA accelerators . The baseline experiments for MADlib were performed on a machine with four Intel i7-6700 cores at 3.40GHz running Ubuntu 16.04 xLTS with kernel 4.8.0-41, 32GB memory, a 256GB Solid State Drive storage. We run each workload with MADlib v1.12 on PostgreSQL v9.6.1 and Greenplum v5.1.0 to measure single- and multi-threaded performance, respectively.

Default setup. Our default setup uses a 32 KB buffer page size and 8 GB buffer pool size across all the systems. As DAnA operates with uncompressed pages to avoid on-chip decompression overheads, 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 for 8, 16, and 32 KB 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. We observed the most benefits with 8 segments, making it our default choice. Results are obtained for both warm cache and cold cache settings to better interpret the impact of I/O on the overall runtime. In the case of a warm cache, before query execution, training data tables for the publicly available dataset reside in the buffer pool, whereas only a part of the synthetic datasets are contained in the buffer pool. For the cold cache setting, before execution, no training data tables reside in the buffer pool.

Table 5: Absolute runtimes across all systems.

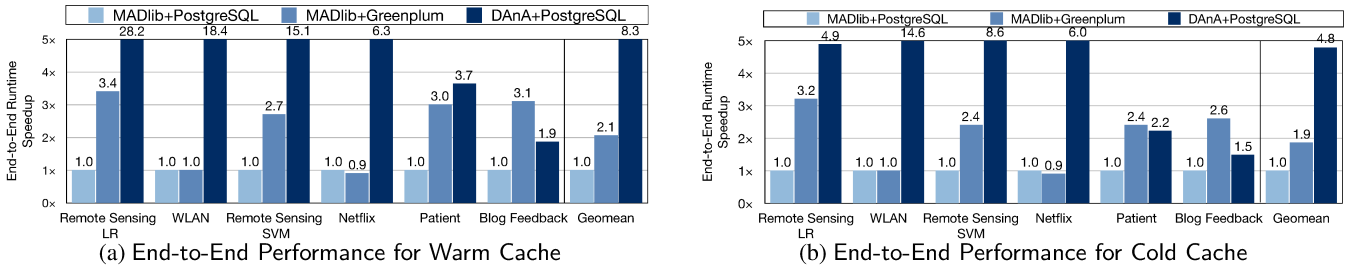
Workloads	MADlib+PostgreSQL	MADlib+Greenplum	DAnA+PostgreSQL
Remote Sensing LR	3s 600ms	1s 100ms	0s 100ms
WLAN	14s 0ms	14s 0ms	0s 610ms
Remote Sensing SVM	1s 700ms	0s 600ms	0s 90ms
Netflix	62s 300ms	69s 200ms	7s 890ms
Patient	2s 800ms	0s 900ms	1s 180ms
Blog Feedback	1s 600ms	0s 500ms	0s 340ms
S/N Logistic	54m 52s	49m 53s	2m 11s
S/N SVM	56m 26s	12m 50s	4m 4s
S/N LRMF	0m 23s	0m 3s	0m 2s
S/N Linear	29m 7s	24m 16s	5m 35s
S/E Logistic	66h 45m 0s	8h 30m 0s	0h 11m 24s
S/E SVM	0h 6m 0s	0h 5m 24s	0h 1m 12s
S/E LRMF	0h 54m 36s	0h 26m 24s	0h 39m 0s
S/E Linear	6h 36m 36s	5h 22m 12s	0h 16m 48s

7.1 End-to-End Performance

Publicly available datasets. Figures 8a and 8b illustrate end-to-end performance of MADlib+PostgreSQL, Greenplum+MADlib, and DAnA, for warm and cold cache. The x-axis represents the individual workloads and y-axis the speedup. The last bar provides the geometric mean (geomean) across all workloads. On average, DAnA provides 8.3× and 4.8× end-to-end speedup over PostgreSQL and 4.0× and 2.5× speedup over 8-segment Greenplum for publicly available datasets in warm and cold cache setting, respectively. The benefits diminish for cold cache as the I/O time adds to the runtime and cannot be parallelized. The overall runtime of the benchmarks reduces from 14 to 1.3 seconds with DAnA in contrast to MADlib+PostgreSQL.

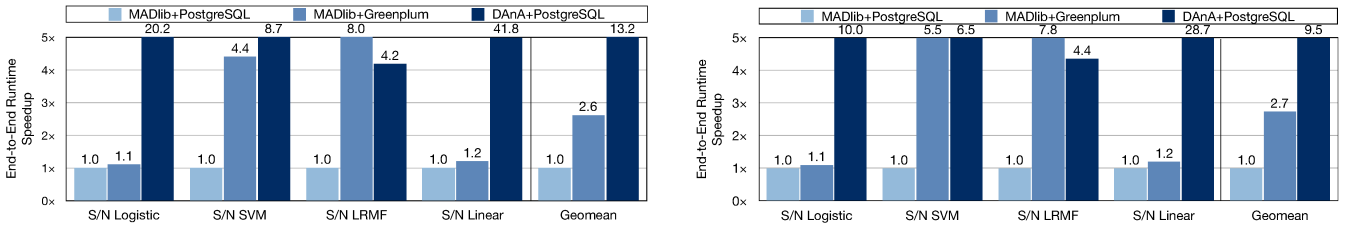
The maximum speedup is obtained by Remote Sensing LR, 28.2× with warm cache and 14.6× with cold cache. This workload runs logistic regression algorithm to perform non-linear transformations to categorize data in different classes and offers copious amounts of parallelism for exploitation by DAnA's accelerator. In contrast, Blog Feedback sees the smallest speedup of 1.9× (warm cache) and 1.5× (cold cache) due to the high CPU vectorization potential of the linear regression algorithm.

Synthetic nominal and extensive datasets. Figures 9 and 10 depict end-to-end performance comparison for synthetic nominal and extensive datasets across our three systems. Across S/N datasets, shown in Figure 9, DAnA achieves an average speedup of 13.2× in warm cache and 9.5× in cold cache setting. In comparison to 8-segment Greenplum, for S/N datasets, DAnA observes a gain of 5.0× for warm cache and 3.5× for cold cache. The average speedup as shown in Figure 10, across S/E datasets in comparison to MADlib+PostgreSQL are 12.9× for warm cache and 11.9× for cold cache. These speedups reduce to 5.9× (warm cache) and 7.0× (cold cache) when compared against 8-segment MADlib+Greenplum. Higher benefits of acceleration are observed with larger datasets as DAnA accelerators are exposed to more



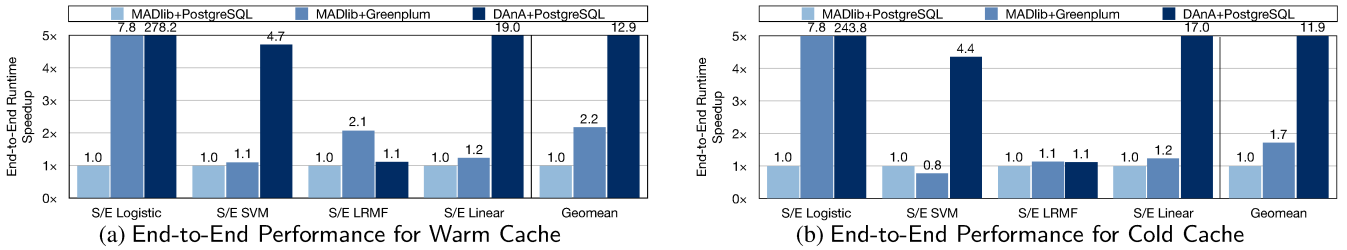
(a) End-to-End Performance for Warm Cache (b) End-to-End Performance for Cold Cache

Figure 8: End-to-end runtime performance comparison for publicly available datasets with MADlib+PostgreSQL as baseline.



(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 synthetic nominal datasets with MADlib+PostgreSQL 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 extensive datasets with MADlib+PostgreSQL as baseline.

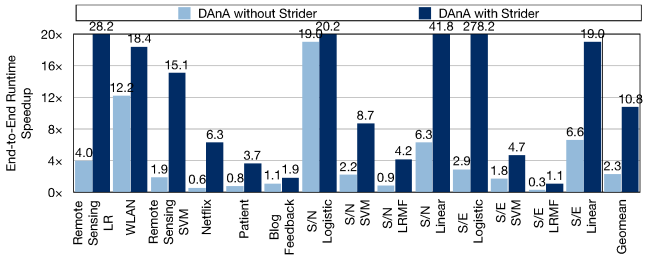


Figure 11: Comparison of DAnA with and without Striders with PostgreSQL +MADlib as the baseline.

opportunities for parallelization, which enables the accelerator to hide the overheads such as data transfer across platforms, on-chip data alignment, and setting up the execution pipeline. These results show the efficacy of the multi-threading employed by DAnA's execution engine in comparison to the scale-out Greenplum engine. The total runtime for S/N Logistic and S/E Logistic reduces from 55 minutes to 2 minutes and 66 hrs to 12 minutes, respectively. These workloads achieve high reduction in total runtimes due to their high skew towards compute time, which DAnA's execution engine is specialized in handling. For S/N SVM, DAnA is only able to reduce the runtime by 20 seconds. This can be attributed to the high I/O time incurred by the benchmark in comparison to its compute time, thus, the accelerator frequently stalls for the buffer pool page replacements to complete. Nevertheless, for S/E SVM, DAnA still reduces the absolute runtime from 55 to 39 minutes.

Evaluating Striders. A crucial part of DAnA's accelerators is their direct integration with the buffer pool via *Striders*. To evaluate the effectiveness of *Striders*, we simulate an alternate design where DAnA's execution engines are fed by the CPU. In this alternative, the CPU transforms the training tuples and sends them to the execution engines. Figure 11 compares the end-to-end runtime of DAnA with and without *Striders* using warm cache MADlib+PostgreSQL as baseline. DAnA with and without *Striders* achieve, on average, 10.7× and 2.3× speedup in comparison to the baseline. Even though raw application hardware acceleration has its benefits, integrating *Striders* to directly interface with the database engine amplifies those performance benefits by 4.6×. The *Striders* 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* in integrating with PostgreSQL.

7.2 Performance Sensitivity

Multi-threading in Greenplum. As shown in Figure 13, for publicly available datasets, the default configuration of 8-segment Greenplum provides 2.1× (warm cache) and 1.9× (cold cache) higher speedups than its PostgreSQL counterpart. The 8-segment Greenplum performs the best amongst all options and performance does not scale as the segments increase.

Performance Sensitivity to FPGA resources. Two main resource constraints on the FPGA are its compute capability and bandwidth. DAnA configures the template architecture in accordance to the algorithmic parameters and FPGA constraints. To maximize compute resource utilization, DAnA supports a multi-threaded execution engine, where each thread runs a version of the update rule. We perform an analysis for varying number of threads on the final accelerator by changing the merge coefficient. A merge coefficient of 2 implies a maximum of two threads. However, a large merge coefficient, such as 2048, does not warrant 2048 threads, as the FPGA may not have enough resources. In UltraScale+ FPGA, maximum 1024 compute units can be instantiated.

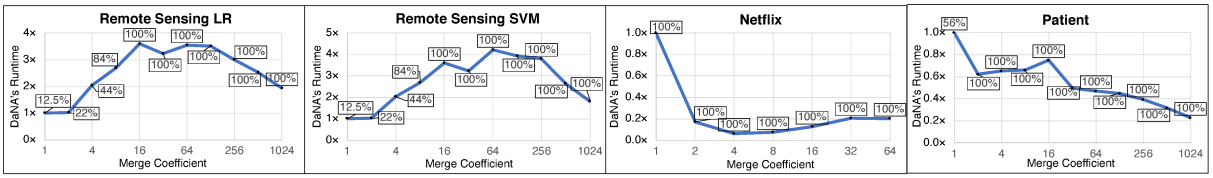


Figure 12: Runtime performance of DAnA with increasing # of threads compared with single-thread as baseline.

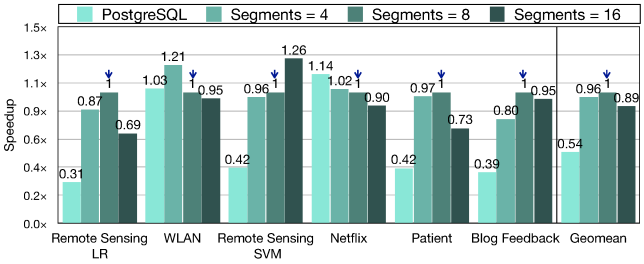


Figure 13: Greenplum performance with varying segments.

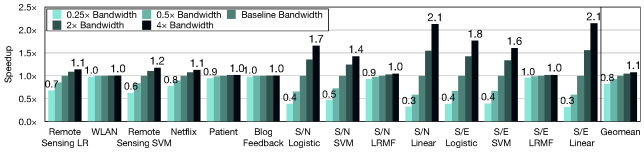


Figure 14: Comparison of FPGA time with varying bandwidth.

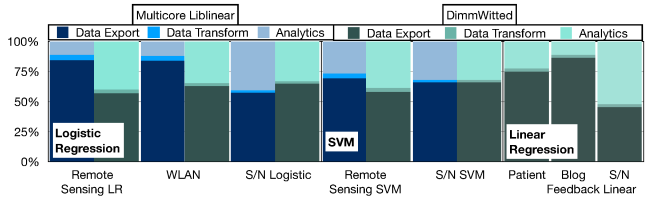
Figure 12 shows performance sensitivity with increasing compute utilization of FPGA for different workloads. Each plot shows DAnA's accelerator runtime (access engine + execution engine) in comparison to a single-thread. The sensitivity towards compute resources is a function of algorithm type, model width, and # of epochs. Thus, each workload fares differently with varying compute resources. Workloads such as Remote Sensing LR and Remote Sensing SVM have a narrow model size, thus, increasing the number of threads increases performance till they reach peak compute utilization. On the other hand, LRMF algorithm workloads do not experience a higher performance with increasing number of threads. This can be attributed to the copious amounts of parallelism available in a single instance of the update rule. Thus, increasing the number of threads reduces the ACs allocated to a single thread, whereas, merging across multiple different threads incurs an overhead. One of the challenges tackled by the compiler is to allocate the on-chip resources by striking a balance between the single-thread performance and multi-thread parallelism.

Figure 14 illustrates the impact of FPGA bandwidth (in comparison to baseline bandwidth) on the speedup of DAnA accelerators. The results show that as the size of the benchmark increases, except the ones that run LRMF algorithm, the workloads become bandwidth bound. The workloads S/N LRMF and S/E LRMF are compute heavy, thus, bandwidth increase does not have a significant impact on the accelerator runtime.

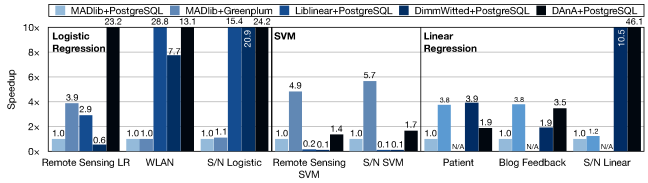
7.3 Comparison to Custom Designs

Potential alternatives to DAnA are: (1) custom software libraries [39–41] that run multi-core advanced analytics and (2) algorithm specific FPGA implementations [42–44]. For these alternatives, if training data is stored in the database, there is an overhead to extract, transform, and supply the data in accordance to each of their requirements. We compare the performance of these alternatives with MADlib+PostgreSQL and DAnA.

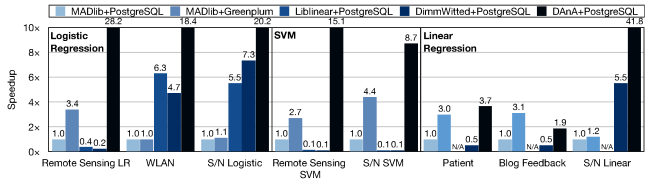
Optimized software libraries. We compare C++-optimized libraries DimmWitted and Liblinear-Multicore classification with MADlib+PostgreSQL, Greenplum+MADlib, and DAnA accelerators. Liblinear supports Logistic Regression and SVM, and



(a) Runtime breakdown for Liblinear and DimmWitted



(b) Compute Time Comparison



(c) End-to-End Runtime Comparison

Figure 15: Comparison to external software libraries.

DimmWitted supports SVM, Logistic Regression, Linear Regression, Linear Programming, Quadratic Programming, Gibbs Sampling, and Neural Networks. Logistic Regression, SVM, and Linear Regression (only DimmWitted), overlap with our benchmarks, thus, we compare multi-core versions (2, 4, 8, 16 threads) of these libraries and use the minimum runtime. We maintain the same hyper-parameters, such as tolerance, and choice of optimizer to compare runtime of 1 epoch across all the systems. We separately compare the compute time and end-to-end runtime (data extraction from PostgreSQL + data transformation + compute), as well as provide a runtime breakdown. Figure 15a illustrates the breakdown of Liblinear and DimmWitted into the different phases that comprise the end-to-end runtime. Data exporting and reformatting for these external specialized ML tools is an overhead specific to performing analytics outside RDBMS. Results suggest that DAnA is uniformly faster, as it (1) does not export the data from the database, (2) employs *Striders* in the FPGA to walk through the data, and (3) accelerates the ML computation with an FPGA. However, different software solutions exhibit different trends, as elaborated below.

- **Logistic regression:** As Figure 15b shows, Liblinear and DimmWitted provide 3.8× and 1.8× speedup over MADlib+PostgreSQL for logistic regression-based workloads in terms of compute time. With respect to end-to-end runtime compared to MADlib+PostgreSQL (Figure 15c), the benefits from Liblinear reduce to 2.4× and increase for DimmWitted to 2.1×. On the other hand, DAnA outperforms Liblinear by 2.2× and DimmWitted by 4.7× in terms of compute time. For overall runtime, DAnA is 9.1× faster than Liblinear and 10.4× faster than DimmWitted. The compute time of Remote sensing LR benchmark receives the least benefit from LibLinear and DimmWitted

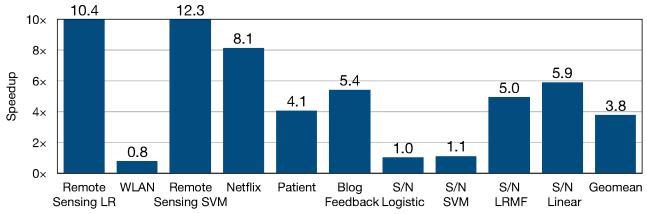


Figure 16: Performance comparison of DAnA over TABLA.

and exhibits a slowdown in end-to-end runtime. This can be attributed to the small model size, which, despite a large dataset, does not provide enough parallelism that can be exploited by these libraries. Specifically sparse datasets, such as WLAN, are handled more efficiently by these libraries.

- **SVM:** As shown in Figure 15b, that compares compute time of SVM-based workloads, Liblinear and DimmWitted are 18.1× and 22.3× slower than MADlib+PostgreSQL, respectively. For end-to-end runtime (Figure 15c), the slowdown is reduced to 14.6× for Liblinear and 15.9× for DimmWitted, due to the complex interplay between data accesses and UDF execution of MADlib+PostgreSQL. In comparison, DAnA outperforms Liblinear by 30.7× and DimmWitted by 37.7× in terms of compute time. For overall runtime, DAnA is 127× and 138.3× faster than Liblinear and DimmWitted, respectively.
- **Linear regression:** For linear regression-based workloads, DimmWitted is 4.3× faster than MADlib+PostgreSQL in terms of compute time. For end-to-end runtime compared to MADlib+PostgreSQL (Figure 15c), the speedup of DimmWitted is reduced to 12%. DAnA outperforms DimmWitted by 1.6× and 6.0× in terms of compute and overall time, respectively.

Specific FPGA implementations. We compare hand-optimized FPGA designs created specifically for one algorithm with our reconfigurable architecture. DAnA's execution engine performance is on par with Parallel SVM [42], is 44% slower than Heterogeneous SVM [43], and is 1.47× faster than Falcon Logistic Regression [44]. In addition to the speedup, we compare Giga Ops Per Second (GOPS), to measure the numerical compute performance of these architectures. In terms of GOPS, DAnA performs, on average, 16% less operations than these hand-coded designs. In addition to providing comparable performance, DAnA relieves the data scientist of the arduous task of hardware design and testing whilst integrating seamlessly within the database engine. Whereas, for these custom designs, designer requires hardware design expertise and long verification cycles to write ≈15000 lines of Verilog code.

Comparison with TABLA. We compare DAnA with TABLA [5], an open-source framework [45] that generates optimized FPGA implementations for a wide variety of analytics algorithms. We modify the templates for UltraScale+ and perform design space exploration to present the best case results with TABLA. Figure 16 shows that DAnA generated accelerators perform 4.7× faster than TABLA accelerators. DAnA's benefits can be attributed to the: (1) interleaving of *Striders* in the access engine with the execution engine to mitigate the overheads of data transformation and (2) the multithreading capability of the execution engines to exploit parallelism between different instances of the update rule. TABLA on the other hand, offers only single threaded acceleration.

8. RELATED WORK

Hardware acceleration for data management. Accelerating database operations is a popular research direction that connects modern acceleration platforms and enterprise in-database analytics as shown in Figure 1. These prior FPGA-based solutions aim to accelerate DBMS operations (some portion of the query) [3, 4, 23, 24, 46], such as join and hash. LINQits [46] accelerates database queries but does not focus on machine learning. Centaur [3] dynamically decides which particular operators in a MonetDB [47] query plan can be executed on FPGA and creates a pipeline between FPGA and CPU. Another work [24] uses FPGAs to provide a robust hashing mechanism to accelerate data partitioning in database engines. In the GPU realm, HippogriffDB [22] aims to balance the I/O and GPU bandwidth by compressing the data that is transferred to GPU. Support for in-database advanced analytics for FPGAs in tandem with *Striders* set this work apart from the aforementioned literature, which does not focus on providing components that integrate FPGAs within an RDBMS engine and machine learning.

Hardware acceleration for advanced analytics. Both research and industry have recently focused on hardware acceleration for machine learning [5, 21, 25, 48] especially deep neural networks [49–53] connecting two of the vertices in Figure 1 traid. These works either only focus on a fixed set of algorithms or do not offer the reconfigurability of the architecture. Among these, several works [19, 29, 54] provide frameworks to automatically generate hardware accelerators for stochastic gradient descent. However, none of these works provide hardware structures or software components that embed FPGAs within the RDBMS engine. DAnA's Python DSL builds upon the mathematical language in the prior work [5, 19]. 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 such an integration [7, 14, 15, 17, 55–61] (see [62] 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, *Strider*, that directly interfaces with the database to alleviate some of the shortcomings of the 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 an 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., RDBMS with FPGA accelerators for compute-intensive advanced data analytics. DAnA provides the initial coalescence between these paradigms and empowers data scientists with no knowledge of hardware design, to use accelerators within their current in-database analytics procedures.

10. ACKNOWLEDGMENTS

We thank Sridhar Krishnamurthy and Xilinx for donating the FPGA boards. We thank Yannis Papakonstantinou, Jongse Park, Hardik Sharma, and Balaji Chandrasekaran for providing insightful feedback. This work was in part supported by NSF awards CNS#1703812, ECCS#1609823, Air Force Office of Scientific Research (AFOSR) Young Investigator Program (YIP) award #FA9550-17-1-0274, and gifts from Google, Microsoft, Xilinx, Qualcomm, and Opera Solutions.

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 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.

[6] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 13–24, 2014.

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

[10] Amazon web services postgresql. https://aws.amazon.com/rds/postgresql/.

[11] Azure sql database. https://azure.microsoft.com/en-us/services/sql-database/.

[12] Oracle Data Mining. http://www.oracle.com/technetwork/database/options/advanced-analytics/odm/overview/index.html.

[13] Oracle R Enterprise. http://www.oracle.com/technetwork/database/database-technologies/r/r-enterprise/overview/index.html.

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

[15] 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. *PVLDB*, 5(12):1700–1711, 2012.

[16] Microsoft SQL Server Data Mining. https://docs.microsoft.com/en-us/sql/analysis-services/data-mining/data-mining-ssas.

[17] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*. Curran Associates, Inc., 2011.

[18] 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.

[19] Jongse Park, Hardik Sharma, Divya Mahajan, Joon Kyung Kim, Preston Olds, and Hadi Esmaeilzadeh. Scale-out acceleration for machine learning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 367–381, New York, NY, USA, 2017. ACM.

[20] 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.

[21] 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 *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[23] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *PVLDB*, 2(1):910–921, 2009.

[24] 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.

[25] 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 FCCM*, pages 160–167, 2017.

[26] Amazon EC2 F1 instances: Run custom FPGAs in the amazon web services (aws) cloud. https://aws.amazon.com/ec2/instance-types/f1/, 2017.

[27] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[28] 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.

[29] 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.

[30] 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.

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

[32] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *Neural Information Processing Systems*, 2010.

[33] 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.

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

[35] 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.

[36] 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.

[37] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous SGD. In *International Conference on Learning Representations Workshop Track*, 2016.

[38] A. Frank and A. Asuncion. University of california, irvine (uci) machine learning repository, 2010.

[39] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. Strads: A distributed framework for scheduled model parallel machine learning. In *Proceedings of the 11th European Conference on Computer Systems*, pages 5:1–5:16, New York, NY, USA, 2016. ACM.

[40] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[41] Ce Zhang and Christopher Ré. Dimmwitted: A study of main-memory statistical analytics. *Computing Research Repository (CoRR)*, abs/1403.7550, 2014.

[42] S. Cadambi, I. Durdanovic, V. Jakkula, M. Sankaradass, E. Cosatto, S. Chakradhar, and H. P. Graf. A massively parallel fpga-based coprocessor for support vector machines. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, pages 115–122, April 2009.

[43] M. Papadonikolakis and C. S. Bouganis. A heterogeneous fpga architecture for support vector machine training. In *2010 18th IEEE FCCM*, pages 211–214, May 2010.

[44] Falcon computing. http://cadlab.cs.ucla.edu/~cong/slides/HALO15_keynote.pdf.

[45] TABLA source code. http://www.act-lab.org/artifacts/tabla/.

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

[47] 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 Technical Committee on Data Engineering*, 35(1):40–45, 2012.

[48] 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 *Proceedings of 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[49] 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 *42nd International Symposium on Computer Architecture (ISCA)*, 2015.

[50] 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. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, Piscataway, NJ, USA, 2016. IEEE Press.

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

[52] 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.

[53] 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.

[54] 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.

[55] Boriana L. Milenova, Joseph S. Yarmus, and Marcos M. Campos. Svm in oracle database 10 g : Removing the barriers to widespread adoption of support vector machines. *PVLDB*, pages 1152–1163, 2005.

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

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

[58] 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.

[59] M. Levent Koc and Christopher Ré. Incrementally maintaining classification using an rdbms. *PVLDB*, 4(5):302–313, 2011.

[60] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An architecture for compiling udf-centric workflows. *PVLDB*, 8(12):1466–1477, 2015.

[61] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. January 2017.

[62] 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.