In-RDBMS Hardware Acceleration of Advanced Analytics

This is an addendum, which is a part of the paper published in 44th International Conference on Very Large DataBase (VLDB) 2018.

Appendix A

Field Programmable Gate Arrays

This section provides a brief overview of the Field Programmable Gate Arrays (FPGAs). As machine learning is a constantly evolving field, FPGAs offer a potent solution for hardware acceleration as they can be reconfigured to execute new learning-based algorithms. An FPGA is an integrated circuit that comprises reprogrammable look up tables (LUTs), which mimic digital logic by storing a corresponding design configuration in the Static Random Access Memories (SRAMs). Figure 1 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 means to specify hardware logic at register-transfer level (RTL). The designer is expected to provide a 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 that involves rigorous simulations and cycle-by-cycle waveform generation to verify the functionality of the logic. Post verification, the programmer often has

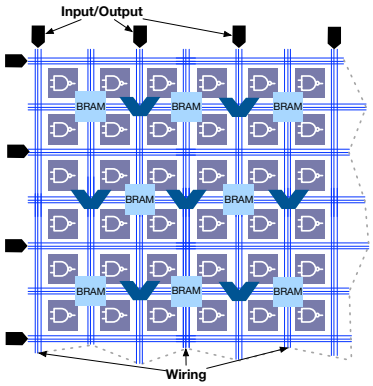


Figure 1: Illustration of a small corner of an FPGA chip. Modern FPGAs comprise a sea of binary lookup tables, augmented with hardened Block RAM and DSP slices to offer higher frequencies.

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. The designer is still expected to go through the verification and optimization processes. DAnA aims to exploit the reconfigurability and high performance of FPGAs for advanced analytics without requiring the user to endure this painful design flow. Thus, it provides a parametric architecture (∼15,000 lines of Verilog code) designed *once* by hardware experts that can be tailored for each 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 and collaborative filtering, can 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 about implementation complexity. The parametric architecture is automatically configured by a domain-specific compiler and hardware generator according to the analyst-provided

application.

Appendix B

Instruction Set Architecture for the Execution Engines

This section provides technical details about the execution engine's Instruction Set Architecture (ISA). However, before delving into the details of the ISA, we describe the execution engine architecture for the entirety of its design and programmability.

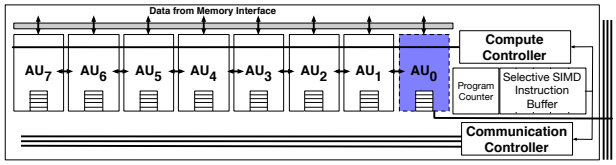
Execution Engine Architecture

The execution engines perform the *h*DFG generated from the user provided UDF on the *Strider*-processed data. As the BRAM capacity has been rapidly increasing 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 both regular and irregular parallelism in ML algorithms whilst providing the flexibility to each component of the architecture to run independently.

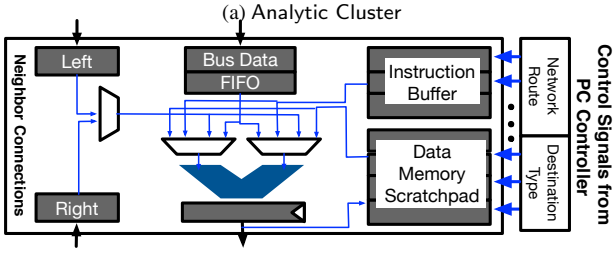
Reconfigurable compute architecture. All the threads in the execution engine are architecturally identical and perform the same computations on different tuples. DAnA balances the resources allocated per thread vs. the number of threads to ensure high performance for each algorithm. The hardware generator of DAnA 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 caters to commonly seen compute operations in data analytics.

Analytic cluster. An AC is a cluster of analytics units designed to reduce the data transfer latency between the AUs, as shown in Figure 2a. Therefore, *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 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, 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.

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,



(a) Analytic Cluster



(b) Analytic Unit

Figure 2: (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.

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 are combined across the threads via a computationally-enabled tree bus in accordance to the merge function. This tree bus has an attached ALU, to perform computations on in-flight data. This is particularly useful for aggregating results from all the threads. The pliability of this architecture enables DAnA to generate high-performance design that aims to efficiently utilize the resources on the FPGA tailored for the RDBMS engine and the algorithm. We now delve deeper into the execution engine's ISA.

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

Table 1: (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

on-chip. Despite being variable length, every instruction is self-sufficient and contains all relevant material.

As shown in Table 1, this ISA has three instruction types: compute, communication, and operand reads. These are micro-instructions generated for each operation to be performed in the *h*DFG. The blue-shaded fields in the table are mandatory, while the remaining are optional. Fields specified as N/A are not a part of the actual micro-instruction and are fillers to align the fields relevant to each AU for illustration purposes. 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 1b.

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 1c shows all the operand types – data memory, scratchpad, neighbor register, and bus FIFO – from which each AU can read or write its operands or out-

puts, respectively. The operand index instruction is only valid if the operand is of the data memory type. 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, is required and specifies whether the write back or communication component is used. For example, if the local bus Use field is 1, the Source AU uses the bus to transfer data to all AUs indicated by the "Destination AU" field. A use case of this ISA is shown with a simple group operation **sigma** shown below. This operation is performed in an AC using operations shown in Table 2.

$$s = sigma(x_i \times w_i, [8]) \tag{1}$$

Table 2: Operation flow in an AC to perform Equation 1.

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

All AUs in an AC perform the multiply operation. The results are aggregated through a 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 1b) 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. Neither the global bus nor the local bus is used, therefore, the Use field is 0, and none of the remainder fields in those micro-instructions are required. 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 indicating that only alternating AUs write.

These components ensure compatibility between the high level Python UDF and the access and execution engines by generating programs for each.