Automating the Architectural Execution Modeling and Characterization of Domain-Specific Architectures

Shail Dave
School of Computing and Augmented Intelligence,
Arizona State University
Tempe, Arizona, USA
Shail.Dave@asu.edu

Aviral Shrivastava
School of Computing and Augmented Intelligence,
Arizona State University
Tempe, Arizona, USA
Aviral.Shrivastava@asu.edu

ABSTRACT

Domain-specific architectures (DSAs) are increasingly designed to efficiently process a variety of workloads, such as deep learning, linear algebra, and graph analytics. Most research efforts have focused on proposing new DSAs or efficiently exploring hardware/software designs of previously proposed architecture templates. Recent architectural modeling or simulation frameworks for DSAs can analyze execution costs, e.g., for a limited architectural templates for dense DNNs such as systolic arrays or a spatial architecture with an array of processing elements and 3-level memory hierarchy. However, they are manually developed by domain-experts, containing several 1000s of lines-of-code, and extending them for characterizing new architectures is infeasible, such as DSAs for sparse DNNs. Further, the lack of automated architecture-level execution modeling limits the design space of novel architectures that can be explored/optimized, affecting overall efficiency of solutions, and it delays time-to-market with low sustainability of design process.

To address this issue, this paper introduces *DSAProf*: a framework for automated execution modeling and bottleneck characterization by a *modular, dataflow-driven* approach. The framework uses a flow-graph-based methodology for modeling DSAs in a modular manner via a library of architectural components and analyzing their executions. The methodology can account for analytically modeling and simulating intricacies in the presence of a variety of architectural features such as asynchronous execution of workgroups, sparse data processing, arbitrary buffer hierarchies, and multi-chip or mixed-precision modules. Preliminary evaluations of modeling previously proposed DSAs for dense/sparse deep learning demonstrate that our approach is extensible for novel DSAs and it can accurately and automatically characterize their latency and identify execution bottlenecks, without requiring designers to manually build analysis/simulator from scratch for every DSA.

TECHCON '23, September 10–12, 2023, Austin, TX

CCS CONCEPTS

• **Hardware** → *Electronic design automation*; **Application specific processors**; *Hardware accelerators*; • **Computer systems organization** → *Special purpose systems*; System on a chip; Heterogeneous (hybrid) systems; Neural networks; • **General and reference** → *Measurement*; **Performance**; *Design*; • **Computing methodologies** → **Modeling methodologies**; • **Software and its engineering** → Domain specific languages; *Abstraction, modeling and modularity*; • **Social and professional topics** → *Sustainability*.

1 INTRODUCTION

Domain-specific architectures (DSAs) are increasingly designed for efficiency processing a wide range of workloads, including deep learning and artificial intelligence, linear algebra, and scientific computing, and they have been deployed from datacenters to edge [5, 7, 10, 25]. Their designs face strict execution requirements (e.g., performance, energy, or area efficiency [40]) and require agile toolflows for designing, evaluating, and optimizing the DSAs [13, 54]. Tools/techniques for architecture-level execution modeling and characterization of DSAs are paramount for the design phase, as they provide quantitative evaluations of DSA-workload configuration and insights for further improvements.

Current approaches for execution modeling of DSAs. They require heavy development efforts from experts and target a specific architectural template. For instance, analytical models for estimating latency of deep learning accelerators such as [9, 28, 38, 42, 52] were developed by domain-experts for a specific template architecture, e.g., either a systolic array or a spatial architecture with 3/4-level memory hierarchy for dense DNN operators like convolutions or matrix multiplications [30, 33, 54]. These analytical cost models contain several 1000s of lines of code (LoC), and extending them becomes very challenging, when an architecture needs to be modified with a new component for specialization, e.g., evaluating a DSA for sparse DNNs. While AI-based approaches for quantifying processor executions exist [2, 24, 32, 57], they also face same limitations. This is because, these AI models have been developed only for off-the-shelf processors and require extensive data curation and off-line training, which

again may rely on already developed in-house simulators for target DSAs or actual processors that are available in only post-design phase.

Adverse implications. Time-consuming development efforts from domain-experts are required due to lacking automatic architecture-level execution modeling for DSAs. It limits the design space of novel architectures that can be modeled or explored, affecting an agile design development (prolongs time-to-market) and overall efficiency of solutions. Further, it significantly lowers the sustainability of the DSA design process. Recent industrial studies like [19] have shown that the carbon footprint for producing processors, especially DSAs, can largely overshadow the benefits achieved through operational efficiency (e.g., 80%). And, their calculations focused only on post-design phases like manufacturing. The implications can exacerbate notably when accounting for the design-time efforts and human/compute resources.

Need. We require generic methodologies for flexible and extensible execution modeling, quantification, and characterization for a wide range of DSAs, without having to build a full analysis/simulators from scratch in entirety for new DSA templates – which is the primary objective of this work.

Underlying research challenges. There are two main challenges that prevent the automated latency/energy modeling and characterization of DSAs: 1) Unitary or non-modular development, and 2) lack of automatic determination of execution activity. Firstly, it is challenging to extend or partly reuse an existing unitary model for developing a new cost model or simulator for a new DSA. This is because, for a unitary model, designers require significant efforts to first figure out exact modeling of each component that can be reused. Plus, efforts/implications of introducing models of new components on the total latency/energy are non-trivial and might lead to erroneous modeling. Further, these unitary models only provide a single value, such as total latency, and for performance characterization, it makes infeasible to automatically track it down to the underlying design/execution factors that are likely causing inefficiencies.

Secondly, component-level and overall execution modeling requires execution activity of each component, such as invocation counts and the meta-information about the received input data. For a fixed template architecture, such information is embedded manually based on the calculations for specific workloads, e.g., in [9, 28, 45]. Otherwise, it needs to be calculated manually based on mappings of the workloads onto DSA and intra-DSA dataflow, and supplied as an auxiliary input to the execution model, e.g., in [50]. For a more generalized execution modeling, such information about execution activity needs to be supplied automatically.

Our approach. We propose *DSAProf*, a framework for automated execution modeling and bottleneck characterization, which addresses above challenges through a *modular, dataflow-driven* approach. Modular modeling/characterization approach requires a flexible abstraction for DSAs that can model DSAs of various hierarchy, spatial resources, pipelining, and heterogeneity of components. We visualize, express, and evaluate DSAs as flow graphs, which can enable modularity, while allowing to model a wide range of DSAs. Through proposed python-based embedded DSL constructs and features, DSAProf allows specifying the architecture graph of DSAs in a flexible/readable manner, as compared to prior approaches. Especially, they allow concealing low-level components and automatic interfacing of DSA inputs/outputs with a synthetic controller.

The proposed execution model follows a dataflow-driven approach for supplying execution activity. The latency model of each component in the library processes input data or control logic from the input ports for deriving the latency corresponding to the activity, and populates synthetic data on output ports for forwarding the execution activity to sink nodes. The dataflow is triggered/updated by the external inputs to the DSA, which are provided by configuring the outputs from the synthetic controller. Such controller outputs (control logic or meta information about input data for DSA) can be provided by designers or automation tools through machine code related routines. The DSA components receiving inputs from the controller are invoked by such routines, which are invoked as part of the mappings of workloads on a DSA. With the modular construction of DSA and execution activity exchange among nodes via ports, overall latency for a time interval can be calculated simply by traversing the DSA's architecture graph and aggregating the latency values of individual components. Each time interval correspond to invoking one or more routines for configuring one or more controller-interfacing components, and it can be advanced as per mapping for the DSA. DSAProf uses the latency of each component and constructs a simple bottleneck graph. By traversing the latency values for the DSA components and those in its bottleneck graph, DSAProf can automatically pinpoints execution bottlenecks.

Results and broad impacts. Preliminary evaluations of popular DSAs for dense/sparse deep learning [3, 55] show that DSAProf can accurately determine the execution time and energy consumption of DNN workloads within 1%-6% of the reported results for these DSAs. The evaluations also demonstrate that with the proposed modular approach, latency modeling of DSAs for dense deep learning (e.g., Eyeriss [3]) can be easily extended to model and characterize the latency/energy of DSAs for sparse DNNs (e.g., Cambricon-X [55], EIE [21], or SIGMA [39]), without enforcing experts to develop a new latency model or a simulator from scratch.

It introduces up to only a few 10s of LoC for extending the library by introducing a few new components, while simply reusing the libraries for available components, DSA specification, and overall cost modeling methodology for modular executions, whereas current approaches of developing dedicated, unitary simulators/models for a new DSA template typically take 1000s of LoC [9, 28, 38] and weeks–months of development efforts [41]. Further, we show a study of automated characterization of Cambricon-X like accelerator for BERT, which matches prior analysis [7] and reveals how indexing modules and load imbalance due to irregular weight sparsity incur 56% excess time, dropping the speedup (over dense model computations) from 12.5× (ideal) to 8×.

2 DSAProf: MODULAR, DATAFLOW-DRIVEN EXECUTION MODELING

2.1 Architecture Graph Abstraction

Flow graph based abstraction for arbitrary hierarchy: Design abstraction impacts the architectures that can be specified by designers or ML-based automation tools and thereafter evaluated or optimized during the explorations. Our abstraction for specifying and evaluating the domain-specific architectures is a flow graph (FG). In our flow graph representation for DSAs, the nodes are primary components of computation, memory, control logic, or interconnect networks [37, 47], or even a sub-graph that represents high-level architectural components like processing engines. The edges simply represent the interconnections/bus of appropriate data widths among the inputs/outputs of the nodes. Fig. 1 represents an example flow graph for the Cambricon-X [55] like DSA. As figure illustrates, such approach can allow modular construction of an accelerator or even a multi-accelerator design, while modeling arbitrary compute/buffer hierarchy for automated analysis. For instance, such approach could allow modeling of memory hierarchy from 2 to 4-levels [52], unified or shared L1 or L2 buffers, compute engines accessing data from on-chip buffers or even directly from off-chip memory via DMAs [18, 38, 55].

Named input/output ports of the nodes enable flexibility in specifying connectivity and automatic dataflow for architectural evaluation/simulation: Each node contains some input and/or output ports that can be connected to the ports of the other nodes via creating edges. The named ports make it easier for architectural specification by designers or AI-based tools that rely on textual representation for correctly specifying the connections among the nodes. The input/output ports are implemented via pointers (or pointer-like data accesses by modifying lists in Python), which make it easier for automated flow of the data for functionality/cycle-level simulation or metadata for performance/energy analysis.

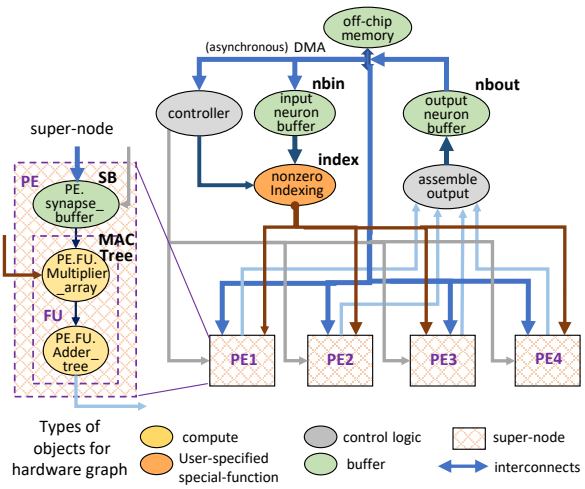

Figure 1: Flow graph of Cambricon-X like accelerator (for sparse deep learning models), which efficiently exploits unstructured sparsity of weights. With a flexible flow-graph abstraction and specification in DSAProf, designers or architecture generation tools can easily specify and characterize various DSAs of different hierarchy and specialization.

Supernodes for grouping/concealing low-level components or modules: Systematic execution analysis, simulation, or visualization of an architecture graph often require the designers to group some low-level components and treat as a high-level architecture component in the hierarchy. For instance, the datapath of a processing element may need to be defined through buffers, ALU, multiplexers, delays, address generators, and additional control logic. In our architecture graph, such a sub-graph for a PE can be defined by grouping of the preliminary components as a supernode, even if the high-level PE definition is unavailable in the imported libraries. When analysis/simulation of an architecture graph reaches such a supernode, it can iterate over the sub-graph similar to the overall architectural flow graph.

Advantages: The flow graph abstraction allows modular construction of architectures and modeling arbitrary hierarchy and grouping of components for higher level analysis. Hence, it can facilitate designs of a wide range of DSA architectures such as [3, 7, 23, 29, 49, 56], as also observed by [47, 50], and even novel architectures. Moreover, the modular, dataflow-driven latency/energy modeling enables opportunities for automated bottleneck characterization of the DSAs as detailed in section 2.5. Further, since algorithms are compiled as data flow graphs [6, 8, 12, 47], they can be conveniently mapped on flow graphs of DSAs, which could be useful in automated architecture search for novel workloads [10].

2.2 DSA Graph Specification

DSAProf uses a python-based embedded DSL, which allows importing the constructs for specifying the architecture graph, while leveraging in-built data types and language constructs in Python for flexibility and compactness in specification. This approach also makes feasible to import or integrate additional libraries (typically in Python/C++) for hardware specification/generation (e.g., from PyMTL [26], gem5 [31], CACTI [34]), design space exploration [1, 35, 36, 46], or application's dataflow graph specifications.

Our graph specification library builds upon Python's *networkx* package, which allows associating custom objects with nodes and edges for a customized DFG (dataflow graph) as well as common routines for constructing/parsing DFG. The listing below shows declaring a new architectural graph of a DSA or a multi-DSA SoC.

```
myDSA = accel(name='myDSA')
```

The newly defined DSA object contains an empty DFG object (for the overall architecture graph) and properties and/or routines for execution modeling, including latency, energy, functionality simulation, and architecture visualization [13].

Node Specification: New nodes can be added to the DSA's flow graph by 'add_node' routine as shown below. It allows to specify the name of the node and associate an architectural component from the imported libraries. The component class defined in the library should contain the information about its input/output ports, internal properties (e.g., buffer size/banks), and the definitions for execution modeling such as latency, power, or functionality for simulation. If a component is not specified or it lacks some features, by default, a baseline node object or its definitions are used.

```
myDSA.dfg.add_node(Name='Buf_L2_Weights',
  Object=buffer(size=1024, data_width=32, banks=8,
  double_buffering=True))
```

Each node contains input/output ports. For instance, a node corresponding to a multi-bank buffer object could have ports for reading/writing the data in each banks of the on-chip buffer. For making the specification flexible/readable, these ports are named in the class initialization of a component, which are later used to connect the nodes during the modular construction of a DSA. If port names are unspecified in the component definition, then the DSL generates and uses default names `input_port_i` and `output_port_i`, where i is the number of the input/output port. Input ports can be designated for receiving either data or control logic. Typical examples of input data ports are data values and buffer addresses generated by components within DSA, whereas control logic like read/write enable or resetting summation in MACs is usually supplied by custom FSMs or controller.

Edge Specification: Edges represent the interconnections among nodes. Interconnections are setup between output ports of source nodes and input ports of sink nodes. The ports are numbered, and with several ports for each nodes, specifying a connection correctly could be challenging. Therefore, our specification allows naming ports and connecting them accordingly. Designers or architecture generation tools could specify multiple source-sink pairs as lists, as shown below. Each item in a list is defined as a `node_name.port_name`, which helps the graph parser to locate appropriate components and their ports.

```
myDSA.dfg.add_edges(list_sources=["DRAM.rd_data0",
  "BufC_L2.rd_data0"], list_sinks=
  ["DMA.rd_data_offchip", "DMA.rd_data_onchip"],
  pipelined=True)
```

Once an edge is specified for connecting ports of components, the ports are connected via pointers (or pointer-alike data access mechanism) in the underling implementation. This represents the setting up data buses of the necessary bitwidth, which must be same for a source port and a sink port. Such setup enables the dataflow for actual data-based simulation or exchange of meta information about analytical modeling of the data transfers (section 2.3).

Pipelining. DSA graphs typically contain architectural components that correspond to different pipeline stages and work in an interleaved manner. Therefore, by default, when edges are formed, they represent a pipeline buffer for a different stage. However, designers can specify whether the connected components constitute the same pipeline stage or not.

Supernode/Grouping Specification: The listing below shows The supernode can be also be defined initially by creating a new flow graph object and associating it.

```
PE_datapath = ['Function_Unit', 'Weight_Buffer']
PE = myDSA.dfg.add_supernode('PE', PE_datapath)
```

Controller/Host Interfacing: The effectual latency/energy analysis of architectural components typically depend on the control inputs specified, which triggers the execution corresponding to certain datapath/operation. Such control logic needs to be supplied to the input ports of the components and should be configured at regular intervals, based on a workload's mapping on the target DSA. Typically, such control logic contains several Boolean signals or categorical values, provided from the host machine or a customized controller that is tightly coupled to the DSA. For a quick setup of a synthetic controller for analytical modeling, the DSL allows specifying a synthetic controller and automatically generating related input/output ports and interconnects, as listed below.

```
# Define a synthetic controller for the DSA
controller = controller()
myDSA.dfg.add_node('Controller', controller)
# Determine unconnected inputs/outputs for
  interfacing with the controller
controller_inputs, controller_outputs =
  myDSA.dfg.get_unconnected_ports()
# Populate controller with the necessary ports for
  the interfacing
controller.set_unconnected_ports(
  controller_inputs, controller_outputs)
# Connect ports between controller and DSA
  components
myDSA.dfg.connect_controller(name='Controller')
```

Once a DSA is specified with all architecture components and a synthetic controller is defined, the automatic interfacing of controller is achieved by figuring out all unconnected input/output ports of the components in the target DSA. In section A.1, we show a case study for defining Cambricon-X like DSA with DSAProf's DSL.

2.3 Dataflow-driven Execution Modeling

Our approach uses a modular, dataflow-driven modeling. The proposed approach is modular, which uses execution model of each architectural component in the DSA graph in order to populate overall quantification. With a dataflow-driven modeling, connected components sin the DSA graph exchange the meta information about data transfers/properties. Such information received by the component during each invocation represents the execution activity. Therefore, the latency model of each component processes meta information about input data or control logic from the corresponding input ports, and then it derives the latency corresponding to the activity. It also populates synthetic data (meta information) on output ports, which forwards the execution activity to the input ports of the sink nodes.

With the modular construction of DSA and execution activity exchange among nodes via ports, overall latency for a time interval can be calculated simply by traversing the DSA's architecture graph. In the DSA graph, one or more root nodes can be designated and the nodes are traversed from the root nodes based on their heights in the tree. Heights of the nodes are determined based on the heights of their predecessors. Based on whether connected components corresponds to the same/different pipeline stage, obtained latency values from models of individual components are aggregated (taking maximum/addition of values). In case of synchronization between the components, the collective latency is updated at the end of each interval.

Overall latency can be found by evaluating models of architectural components as per the DSA graph traversal and summation of aggregated value for every time interval. The time intervals can be advanced as per mapping of a workload onto the DSA. Each time interval corresponds to invoking one or more routines, which configures inputs to the one or more controller-interfacing components. The configurations/data to the interfaced components of the DSA are provided by the outputs from a synthetic controller or host (through machine code routines or a custom instruction/bitstream). Such controller outputs are usually provided by designers or generated through automation tools for the machine code generation. In section A.2, we discuss a case study for mapping SpMV operator on Cambricon-X like DSA that is specified and analyzed with DSAProf. It shows how the relevant machine code routines can be extended to supply the configurations generated by controller.

DSAProf's DSL provides routines for invoking such analysis through a simple function call. Additionally, designers can directly initialize the traversal order or visualize the DSA by invoking in-built methods as shown in the listing below.

```
myDSA.view()
myDSA.dfg.set_roots(['DMA'])
myDSA.dfg.traverse()
execution_time = myDSA.dfg.get_latency()
```

Temporal data representation: Our execution model allows considering data transfers between components over time, which are represented as a n + 1-d array/list where n represents the dimensionality of the data received/sent by the ports of each node. Such temporal representation enables the latency model for a component to consider the whole data sequence for an execution interval and estimating their latency in one shot. This reduces the invocations to the component be made and consequently, a faster analytical estimation or functionality simulation. This is typically possible for most DSAs that process vector streams, at least for several cycles or a time period. Otherwise, either the temporal processing can be disabled or components can be invoked by the generated code in a regular fashion, i.e., at a cycle-level or a single element-level granularity.

Approximating processing for repeated execution behaviors: The execution model allows marking the advancement of execution interval with a repeat factor, which makes the analysis faster. For instance, given a mapping of nested loops on DSAs, execution pattern could be repetitive and the execution latency/energy could simply be estimated by multiplying the obtained estimates with the repetition factor for the loop processing.

Data-aware analysis and functionality simulation: Execution model could be invoked by configuring it with data-aware analysis as well, where components communicate actual data based on their functionality and their latency models take the actual data values into account (e.g., for sparse data processing). While this approach could likely consume less time for analysis as compared to invoking components at every cycle in cycle-level simulators, it is usually slower than the analysis based on meta information about the input/output data (used by default).

2.4 Library for Modeling Performance of Domain-Specific Architectures

We develop a library for estimating performance of modules based on the meta information about data transfers/properties in the execution. Several libraries/frameworks for estimating area or power for these components exist, such as DSAGen [47], Accelergy [50], CACTI [34], which can be integrated to our infrastructure for area/power models of the components. Next we describe some of the preliminary components that are modeled by our library.

Off-chip memory. We model a dummy banked DRAM for off-chip accesses. For off-chip data transfers, we model a DMA engine that can be interfaced with multiple on-chip buffers, accessing them one at a time. We follow a standard latency model that considers latency for initiating the transfer and burst communication over maximum bandwidth available [27]. We rely on external models such as CACTI for accurate estimation of latency parameters for a broad range of off-chip memory configurations.

On-chip memory. We model multi-port and multi-bank SRAM buffers or register files for accessing data on-chip. The on-chip buffers can contain several read-write banks and can be double-buffered for hiding the data access time via memory hierarchy. The execution model considers various scenarios, e.g., when data can be written into banks in either a round-robin fashion or as contiguous block within a target bank. For accurate estimation of read/write latencies for these buffers and broad range of SRAM configurations, we rely on external models such as CACTI.

Functional units. Different function units modeled in the library supports common arithmetical and logical functions on scalar or vector data. For instance, MAC units can perform scalar operations, contain SIMD lanes, or contain multiplier-arrays and/or adder-trees.

NoCs. Our library models common interconnects such as unicast, broadcast, mesh, crossbars, or configurable multicast [3, 4]. Based on the information about total inputs/outputs, communication bandwidth (links, bit-widths of links), and hops, their latency estimation is simple, i.e., the time taken to forward data in time-multiplexed manner [11, 28]. We plan to integrate NoC models for SoC-level communication and chiplets in future.

Sparse data compression and indexing. The components for extracting non-zeros from sparse data support indexing-based and intersection-based mechanisms that act upon positions of non-zeros in one or more tensors [7]. These modules synthetically generate a sample stream for accounting for sparsity structure in real-world and make estimates based on their capabilities for extracting non-zeros from the sample stream. The modules for encoding/decoding sparse data support commonly used compression formats such as COO, CSR, CSC, bitmap, RLC, etc.

Control logic. The library models commonly used components such as multiplexer, FSMs for address generators, as well as synchronization and delay elements.

2.5 Automated Bottleneck Characterization

The modular, dataflow-driven latency/energy modeling enables opportunities for automated bottleneck characterization of the DSAs. This is because, in the modular approach of execution modeling for the whole DSA or a multi-DSA SoC, the information about costs of each architectural component (and subsystem) serves as a fundamental part. This information is inherently available in an explicit manner, as compared to deriving it from the collective cost model or simulator manually written by experts for a specific DSA. Further, with the dataflow-driven execution, metadata about the tensors processed by each component (tensor shapes, sequences received over time, sparsity, etc.) also becomes inherently available in an explicit manner, which relates to the execution activity observed over time. The availability of both the information is essential in constructing a bottleneck analysis of the execution cost, which are usually absent in conventional analytical/simulation-based execution models for performance/energy, requiring explicit manual efforts by domain-experts.

In several scenarios, a simple bottleneck analysis with the information about the components, whose execution costs are excessive or prevailing the overall latency/energy, could be sufficient. For instance, consider the architectural graph of Cambricon-X like DSA in the Figure 1 for dense/sparse deep learning. The latency of such DSAs are primarily dependent on the factors like time consumed by computations (Function Units in PEs - PEFUs), time taken by DMAs for input/output activations and weights, time consumed by NoCs for communicating data between shared buffers and registers/buffers in PEs, and the time taken by decoding and extracting non-zeros for feeding to PEs (indexing and neuron broadcast synchronization). Usually, with a highly pipelined architecture design, these components are almost equally engaged in their executions, and the total execution time is

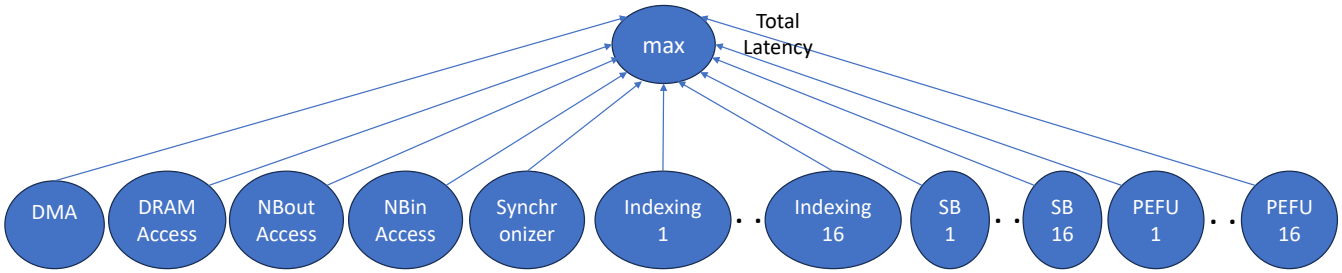

Figure 2: With modular and dataflow-driven latency modeling, DSAProf can construct simple latency bottleneck analysis. A sample bottleneck graph of latency is shown here for processing sparse ML layers on Cambricon-X [55] like DSA that is depicted in Fig. 1.

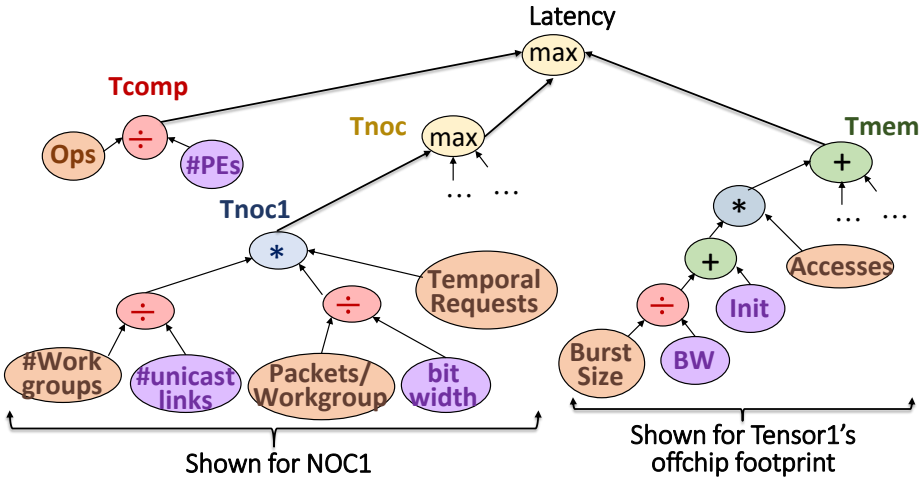

Figure 3: Detailed latency bottleneck analysis for Eyeriss [3] like DSA, which accounts for execution activity as well as execution models and design parameters of architectural components. It can help not only identify the root causes of inefficiencies but also mitigation strategies for improving workload mapping and architecture design.

determined by a maximum value among these factors, as illustrated in Fig. 2. Thus, a simple analysis determining the datapath consuming maximum value could serve to identify the primary bottleneck. We also provide an actual case study analysis for BERT layers in section 3.3.

More accurate accounting for activity may be required when the excess time (consumed by non-perfect interleaving of these execution factors) causes synchronization at intervals, and the active time of components may not be in tandem with overall synchronized activity. Further, we envision constructing a more detailed bottleneck model by leveraging the meta-information about execution activity and the representation of the execution cost calculation for each component. For instance, for an Eyeriss-like spatial architecture, the simple bottleneck graph (e.g., of Fig. 2, which directly considers maximum/addition among latency values) could be transformed into more accurate bottleneck model as illustrated in Fig. 3. Such detailed accounting remains an in-progress/future work. Once such detailed bottleneck models can be constructed, the bottleneck identification and mitigation may be obtained by traversing the information-rich graph and scaling the DSA design parameter or execution metadata (from mapping) based on the contributions of a bottleneck factor to the total cost [11].

3 RESULTS AND ANALYSIS

We use DSAProf's DSL and execution modeling of common components to evaluate latency/energy of DSAs. For preliminary analysis, we consider two popular DSAs for dense/sparse deep learning - Eyeriss [3] and Cambricon-X [55]. We evaluate executions of operators like convolutions and MLPs for their reported results.

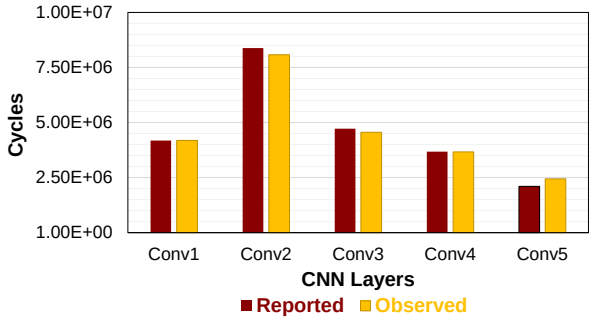

Figure 4: For processing AlexNet convolution layers on Eyeriss [3] like DSA, DSAProf estimates the total execution cycles within 1% of the reported results.

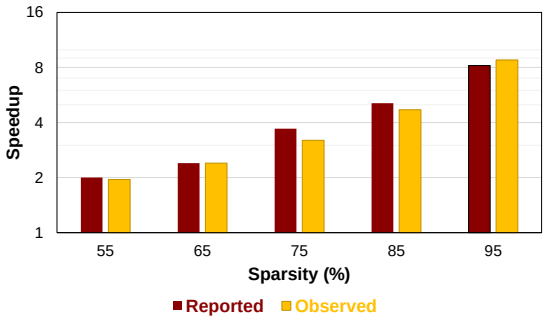

Figure 5: For accelerating fully-connected (MLP) layers of sparse DNNs on Cambricon-X [55] like DSA, DSAProf estimates performance speedups (over dense DNNs) within 6% of the reported results.

3.1 Validation 1: Dense DNN Executions

We modeled Eyeriss-like DSA [3] and evaluated its execution for their reported results, e.g., AlexNet for ImageNet classification [15]. We configure outputs from controller as per mapping-style reported in [3] for their row-stationary dataflow. Figure 4 shows the comparison of execution cycles based on the processing time reported in [3] and the estimated execution cycles. We find that our estimations closely matched execution cycles of the architecture [3], with a difference of about 1% in the total execution cycles. We were unable to validate energy consumption against [3], since it did not report energy consumption when executing layers with reported mapping configurations.

We also evaluated executions of convolutions or matrix multiplications on similar DSAs with other dataflow-style such as output stationary and compared with experts-defined tools such as dMazeRunner [14]. We find that the estimates for execution cycles and energy consumption matched closely with those reported by dMazeRunner.

3.2 Validation 2: Sparse DNN Executions

We demonstrate how proposed modular approach can be easily extended to model and characterize the latency of new DSAs, e.g., for sparse DNNs, without enforcing the development of a new analysis or a simulator for the whole DSA from scratch. After modeling the DSA for dense DNNs, e.g., Eyeriss [3], we extend the library for modeling executions of the DSA components by adding the models for two new modules, i.e., indexing and synchronization for sparse tensor computations. Such support requires adding only up to a few 10s of LoC, while simply reusing the models of available components, such as buffers, NoCs, and compute units for dense tensor processing. It also reuses the DSL for DSA specification and overall modular, dataflow-driven cost calculation approach.

After modeling the Cambricon-X [55], we validated estimated performance for various sparsity in weights of classification (MLP) layers and compared the results with reported values. Fig. 5 compares estimated speedup by DSAProf with the reported speedups in [55], when evaluating a sparse VGG-16 classification layer for various sparsity (55%-95%). It shows that on average, estimated speedups differ from the reported results by 6%. Since the details about DRAM configuration/technology was not clear from [55], we could not compare the energy consumption directly. However, when using DRAM power/energy estimates for a 28 or 45nm technology [21, 52], we find that total energy for Cambricon-X is heavily consumed by off-chip memory accesses, as also reported in [55]. We plan demonstrating modeling/analysis for more DSAs for deep learning and other domains in near future.

3.3 Bottleneck Characterization

We show an automated characterization of Cambricon-X like accelerator for BERT layers [16] (e.g., encoder classification). As discussed previously in section 2.5, the latency of such DSA is preliminary dependent on the factors like time consumed by computations on PEFUs, time taken by DMAs for input/output activations and weights, time consumed by NoCs for communicating data between shared buffers and registers/buffers in PEs, and the time taken by decoding and extracting non-zeros for feeding to PEs (indexing and neuron broadcast synchronization). Cambricon-X uses fat-tree NoCs for output collection and broadcasting of input neurons, making NoC time becomes non-important or at par with computations on PEs. Fig. 6 shows evaluation of a simple bottleneck graph for a BERT encoder layer. The bottlenecks in the total latency are in indexing and synchronization (load imbalance). For the obtained latency, only 64% represents the time required ideally for effectual computations (738 cycles, not shown), while additional 3% gets spent on indexing

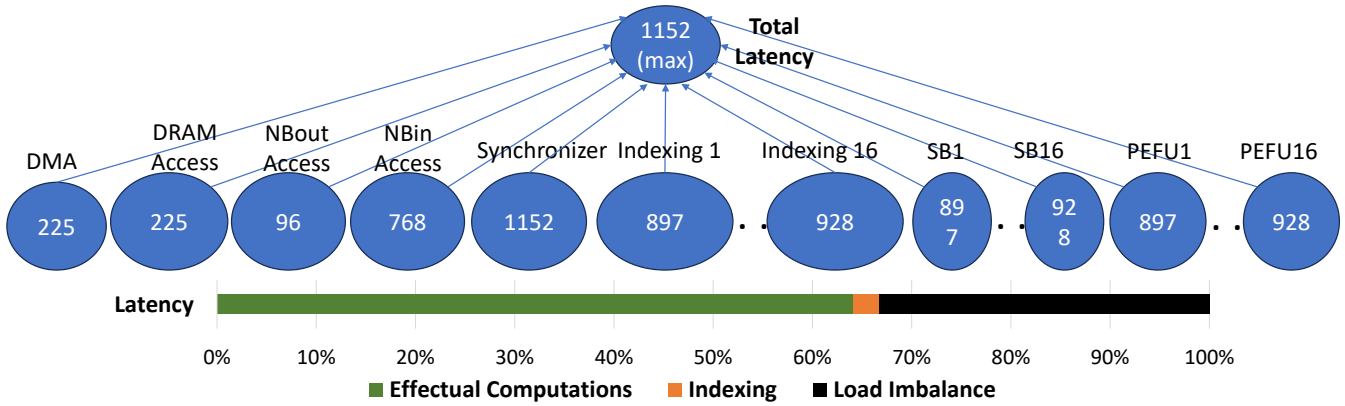

Figure 6: DSAProf's simple bottleneck analysis of latency (execution cycles) for processing a BERT encoder layer with 92% weight sparsity on a Cambricon-X [55] like DSA. Overheads of non-zero extraction and imbalance caused by irregular sparsity leads to 56% excess execution time beyond effectual computations. Such automated characterization helps determine the underlying execution inefficiencies for further optimization.

mechanism for PEs that extract non-zeros, and the rest 33% of excess time is incurred by load imbalance among indexing subunits. This is because, some indexing subunits could spend more cycles to populate sufficient non-zeros for PEs, as the distribution of non-zeros can be imbalanced and irregular. As same neurons need to be processed by all indexing units and PEs, a synchronization is required, which delays the processing depending on a tailing indexing subunit, while other modules remain idle. This obtained characterization also matches with prior analysis reported in [7]. Such overheads for indexing and load imbalance incur 56% excess time. It drops the speedup (over dense model computations) from 12.5× (ideal, considering 92% weight sparsity [7, 48]) to 8×.

4 RELATED WORKS

Graph-based Domain-Specific Architecture Specification: A vast majority of DSAs are specified through a corresponding custom architecture template, as they are application-tailored architectures for ASICs or FPGAs. Thus, their architecture specification is restrained by the underlying template, and it does not require/allow specifying modular construction of the various architectures. A few recent efforts for evaluating/exploring DSAs, such as Accelergy [50] and DSAGen [47] follow similar graph-based approach, but they limit the design space that can be specified, or their details, or offer less flexibility. For instance, Accelergy only allows specifying the nodes and their hierarchy, but not explicit connections among inputs/outputs of the nodes, which is essential in modeling performance and functionality simulation. Through its Scala-based DSL, DSAGen allows specifying connections between the nodes of architecture graph via switches (NoC nodes). However, these connections may need to be defined in a certain order for correctness, as there is no way to indicate connecting a specific input/output of a node to that of another node. Further, it lacks constructs for concealing the low-level components into a higher-level module.

Libraries and Frameworks for Modular Accelerator Construction and Execution Modeling: Frameworks like DSAGen [47] and Accelergy [50] define preliminary components for computation, memory, interconnections, and control. Their definitions typically specify execution costs like area or power; overall cost for the DSA can be obtained by simply addition of the costs of all the components. For performance modeling of the DSA, DSAGen [47] requires cycle-level simulation of the functionality or hardware synthesis of each component [47], which is highly time-consuming. Frameworks for accelerator generation such as MAGNet [45] and AutoDNNChip [51] could estimate the performance/energy, but only for a fixed template architecture.

Automated Execution Modeling of Domain-Specific Architectures: The cost models of SECDA [22] and TVM/VTA [2] support end-to-end simulation and synthesis for their DNN accelerator templates, and it could be highly time consuming. Faster analytical models are more commonly used to optimize mappings and design configurations for deep learning accelerators. Their examples include MAESTRO [28], SCALE-Sim [42], and those of Timeloop [38], dMazeRunner [9], and Interstellar [52] infrastructures. However, all these analytical cost models are developed specifically for a certain template architecture, e.g., either a systolic array or a spatial architecture with 3/4-level memory hierarchy. Therefore, extending them becomes very challenging, when an architecture needs to be modified with a new component for specialization.

Bottleneck Characterization for Performance/Energy: Characterizing bottlenecks in executions of workloads on DSAs is extremely important for optimizing the architecture, configurations of its design parameters, code optimization, as well as for evolving algorithms/workloads. Bottleneck analysis/characterization refers to identifying the underlying inefficiencies that incur higher execution costs (performance and energy) and related strategies for mitigating such inefficiencies. Such bottleneck analysis have been developed/applied for characterizing fixed designs and finding mitigation strategies, e.g., for industry pipelines and production systems, hardware or software for specific applications [44, 53], FPGA-based HLS [22, 43], overlapping microarchitectural events [17], power outage [20], and recently for deep learning accelerators [11]. However, such characterization efforts are largely manual, and they are too specific for their target processor architecture. For instance, AutoDSE [43] and SECDA [22] proposed bottleneck models specific to FPGA-based HLS. Further, current tools for simulation or analytical performance modeling of DSAs contain manually defined cost models, which provide only a single execution value, missing the richer information about how architectural design components contribute to the overall performance and potential mitigation strategies.

5 CONCLUSIONS

Techniques for automated architectural modeling and characterization of DSAs are essential for their agile design development and design optimizations. Current approaches develop unitary analysis/simulations for every DSA template architecture, requiring time-consuming efforts from domain-experts. This restricts design exploration/optimization of novel DSAs and makes the DSA design process highly unsustainable. We propose a modular, dataflow-driven methodology and framework (*DSAProf*) for flexible and extensible execution modeling, quantification, and characterization for a wide range of DSAs, without having to build a full analysis/simulators from scratch in entirety for new DSA templates. The dataflow-driven exchange of execution-related information among DSA components alleviates the need of explicit calculation of execution activity, allowing to quantify or simulate each component separately. Overall methodology builds upon such modular evaluations and aggregates the analysis globally based on pipelining/synchronization among components and specified workload mapping for the DSA. Such modularity also makes it possible to account for each component's contribution to overall execution cost, leading to an automated bottleneck characterization for the DSAs. Preliminary evaluations by modeling the DSAs for dense/sparse deep learning shows that our framework DSAProf can accurately model their performance and energy consumption. It also demonstrates the potential for modeling and characterizing DSAs in a flexible and extensible manner.

ACKNOWLEDGMENTS

This work is partially supported by Graduate College Fellowship at Arizona State University, National Science Foundation (NSF) grant CPS 1645578, and AI hardware (AIHW) program from Semiconductor Research Corporation (SRC).

REFERENCES

[1] 2019. scikit-opt. https://github.com/guofei9987/scikit-opt.
[2] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 578–594.
[3] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2016), 127–138.
[4] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.
[5] William J Dally, Yatish Turakhia, and Song Han. 2020. Domain-specific hardware accelerators. *Commun. ACM* 63, 7 (2020), 48–57.
[6] Anup Das and Akash Kumar. 2018. Dataflow-based mapping of spiking neural networks on neuromorphic hardware. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. 419–422.
[7] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. 2021. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proc. IEEE* 109, 10 (2021), 1706–1752.
[8] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. 2018. Ramp: Resource-aware mapping for cgras. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[9] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. 2019. DMazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–27.
[10] Shail Dave, Alberto Marchisio, Muhammad Abdullah Hanif, Amira Guesmi, Aviral Shrivastava, Ihsen Alouani, and Muhammad Shafique. 2022. Special Session: Towards an Agile Design Methodology for Efficient, Reliable, and Secure ML Systems. In *2022 IEEE 40th VLSI Test Symposium (VTS)*. IEEE, 1–14.
[11] Shail Dave, Tony Nowatzki, and Aviral Shrivastava. 2024. Explainable-DSE: An Agile and Explainable Exploration of Efficient HW/SW Codesigns of Deep Learning Accelerators Using Bottleneck Analysis. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.
[12] Shail Dave and Aviral Shrivastava. 2018. CCF: A CGRA compilation framework.
[13] Shail Dave and Aviral Shrivastava. 2022. Design Space Description Language for Automated and Comprehensive Exploration of Next-Gen Hardware Accelerators. *in Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'22)* (2022). co-located with the 27th ACM International Conference on Architectural Support for

Programming Languages and Operating Systems (ASPLOS 2022).

[14] Shail Dave, Aviral Shrivastava, Youngbin Kim, Sasikanth Avancha, and Kyoungwoo Lee. 2020. dMazeRunner: Optimizing Convolutions on Dataflow Accelerators. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 1544–1548.

[15] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[16] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. 4171–4186.

[17] Brian A Fields, Rastislav Bodik, Mark D Hill, and Chris J Newburn. 2003. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 228–239.

[18] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 2017. Tetris: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 751–764.

[19] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. 2021. Chasing Carbon: The Elusive Environmental Footprint of Computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 854–867.

[20] Di Han, Wei Chen, Bo Bai, and Yuguang Fang. 2019. Offloading optimization and bottleneck analysis for mobile cloud computing. *IEEE Transactions on Communications* 67, 9 (2019), 6153–6167.

[21] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 243–254.

[22] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli. 2021. SECDA: Efficient Hardware/Software Co-Design of FPGA-based DNN Accelerators for Edge Inference. In *2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 33–43.

[23] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 319–333.

[24] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 2021. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 943–958.

[25] John Hennessy and David Patterson. 2018. A new golden age for computer architecture: domain-specific hardware/software co-design, enhanced. In *ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.

[26] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. 2020. PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro* 40, 4 (2020), 58–66.

[27] Michael Kistler, Michael Perrone, and Fabrizio Petrini. 2006. Cell multiprocessor communication network: Built for speed. *IEEE micro* 26, 3 (2006), 10–23.

[28] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 2019. Understanding Reuse, Performance, and Hardware Cost of DNN Dataflow: A Data-Centric Approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 754–768.

[29] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM SIGPLAN Notices* 53, 2 (2018), 461–475.

[30] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. 2020. Susy: A programming model for productive construction of high-performance systolic arrays on fpgas. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[31] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, et al. 2020. The gem5 simulator: Version 20.0+. *arXiv preprint arXiv:2007.03152* (2020).

[32] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *International Conference on Machine Learning*. 4505–4515.

[33] Marco Minutoli, Vito Giovanni Castellana, Cheng Tan, Joseph Manzano, Vinay Amatya, Antonino Tumeo, David Brooks, and Gu-Yeon Wei. 2020. Soda: a new synthesis infrastructure for agile hardware design of machine learning accelerators. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–7.

[34] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* 27 (2009), 28.

[35] Luigi Nardi, David Koeplinger, and Kunle Olukotun. 2019. Practical design space exploration. In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 347–358.

[36] Fernando Nogueira. 2014–. Bayesian Optimization: Open source constrained global optimization tool for Python. https://github.com/fmfn/BayesianOptimization".

[37] Subhankar Pal, Siying Feng, Dong-hyeon Park, Sung Kim, Aporva Amarnath, Chi-Sheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, et al. 2020. Transmuter: Bridging the efficiency gap using memory and dataflow reconfiguration. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 175–190.

[38] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. 2019. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 304–315.

[39] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.

[40] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, et al. 2020. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 446–459.

[41] Hongbo Rong. 2017. Programmatic control of a compiler for generating high-performance spatial hardware. *arXiv preprint arXiv:1711.07606* (2017).

[42] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. 2018. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883* (2018).

[43] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. 2022. AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 27, 4 (2022), 1–27.

[44] Catia Trubiani, Antinisca Di Marco, Vittorio Cortellessa, Nariman Mani, and Dorina Petriu. 2014. Exploring synergies between bottleneck analysis and performance antipatterns. In *Proceedings of the 5th ACM/SPEC International Conference on Performance engineering*. 75–86.

[45] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. MAGNet: A Modular Accelerator Generator for Neural Networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[46] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods* 17, 3 (2020), 261–272.

[47] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 268–281.

[48] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. https://www.aclweb.org/anthology/2020.emnlp-demos.6. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45.

[49] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. 2014. Q100: the architecture and design of a database processing unit. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 255–268.

[50] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[51] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 2020. AutoDNNchip: An automated dnn chip predictor and builder for both FPGAs and ASICs. In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 40–50.

[52] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. 2020. Interstellar: Using halide's scheduling language to analyze dnn accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 369–383.

[53] Yang Yang, Marc Geilen, Twan Basten, Sander Stuijk, and Henk Corporaal. 2010. Automated bottleneck-driven design-space exploration of media processing systems. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 1041–1046.

[54] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 2022. A full-stack search technique for domain optimized deep learning accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 27–42.

[55] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.

[56] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 2018. Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 15–28.

[57] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2022. Towards the co-design of neural networks and accelerators. *Proceedings of Machine Learning and Systems* 4 (2022), 141–152.

A CASE STUDY: ANALYZING SPARSE GEMMS ON CAMBRICON-X LIKE DSA

A.1 Defining the DSA

The DSA can be defined by using the DSL constructs for defining nodes, edges, and grouping of nodes as discussed in section 2.2. Listing in Fig. 7 shows such definition for Cambricon-X like DSA depicted in Fig. 1. With the proposed DSL constructs, DSAs can be defined simply in a few tens of lines of code.

A.2 Mapping for the DSA

Listing in Fig. 8 shows how an SpMV operator can be mapped on a Cambricon-X like DSA. The mapping embeds the machine code routines for setting the controller/host signals and advancing the time intervals. For instance, routines for read/write DMAs (line number 14) sets the related meta information that is provided by the controller to the DMA, which is shown by the pseudo-code listing in the Fig. 9.

Figure 7: Code listing for defining a Cambricon-X like DSA

```
myDSA = accel(name='Cambricon-X', freq(MHz)=1024, technology(nm)=65)
dfg = myDSA.dfg
num_PEs = 16; num_multipliers = 16

# ********************************** Define nodes (components) for DSA **********************************
dfg.add_node('DRAM', DRAM(size=5))                                  # size in GB
dfg.add_node('DMA', DMA(bandwidth=256000))                          # BW in MB/second
dfg.add_node('NBin', buffer(size=8192, read_ports=1, write_ports=1))
dfg.add_node('NBout', buffer(size=8192, read_ports=1, write_ports=1))
# DMA writes data to buffers in PEs and on-chip NBin via demux
dfg.add_node('demux1', demux(num_outputs=num_PEs+1))                # write data for sink
dfg.add_node('demux2', demux(num_outputs=num_PEs+1))                # write address for sink
# Per PE: one synapse buffer, one function unit, one central indexing subunit for neurons
for i in range(num_PEs):
    dfg.add_node(f'BufferController_{i+1}_indexing', indexNonZerosLookupVector(input_length=num_multipliers,
      ↪ lookup_window_size=num_multipliers*16))
    dfg.add_node(f'SB_{i+1}', buffer(size=1024, read_ports=num_multipliers, write_ports=1))
    dfg.add_node(f'PEFU_{i+1}', vectorMAC(multipliers=16, datawidth=16))
dfg.add_node('synchronizer', synchronizer(num_inputs=num_PEs))  # sync. neuron broadcast
# assemble outputs from PEs
dfg.add_node(f'BufferController_assemble', buffer(read_ports=1, write_ports=num_PEs))

# ********************************** Add connections between ports **********************************
# DRAM inputs, outputs, and control signals
dfg.add_edges(["DRAM.read_data_0", "DMA.write_data_offchip", DMA.read_write_addr_offchip",
  ↪ "DMA.read_write_addr_offchip"], ["DMA.read_data_offchip", "DRAM.write_data_0", "DRAM.write_addr_0",
  ↪ "DRAM.read_addr_0"])
dfg.add_edges(["DMA.read_enable_offchip", "DMA.write_enable_offchip", "DMA.read_enable_onchip",
  ↪ "DMA.write_enable_onchip"], ["DRAM.read_enable_0", "DRAM.write_enable_0", "NBout.read_enable_0",
  ↪ "NBout.write_enable_0"])

# DMA inputs, outputs, and control signals
dfg.add_edges(["NBout.read_data_0", "DMA.write_data_onchip", "DMA.read_write_addr_onchip"],
  ↪ ["DMA.read_data_onchip", "demux1.input_data", "demux2.input_data"])
dfg.add_edges(["DMA.out_dest_onchip", "DMA.out_dest_onchip"], ["demux1.select", "demux2.select"])

# L2 (shared) buffers: inputs, outputs, and control signals
dfg.add_edges([f"BufferController_assemble.read_data_0"], [f"NBout.write_data_0"])
dfg.add_edges(["demux1.output_data_0", "demux2.output_data_0"], ["NBin.write_data_0", "NBin.write_addr_0"])
for i in range(num_PEs):
    dfg.add_edges([f"demux1.output_data_{i+1}"], [f"SB_{i+1}.write_data_0"])
    dfg.add_edges([f"demux2.output_data_{i+1}"], [f"SB_{i+1}.write_addr_0"])

# Connect indexing subunits with L2 buffer, their outputs to PEs, and connect datapath in PEs
for i in range(num_PEs):
    dfg.add_edges([f"NBin.read_data_{i}"], [f"BufferController_{i+1}_indexing.input_data"])
    dfg.add_edges([f"BufferController_{i+1}_indexing.output_data"], [f"synchronizer.input_data_{i}"])
    dfg.add_edges([f"synchronizer.output_data_{i}"], [f"PEFU_{i+1}.input_data_0"])
    dfg.add_edges([f"SB_{i+1}.read_data_0"], [f"PEFU_{i+1}.input_data_1"])
    dfg.add_edges([f"PEFU_{i+1}.output_data"], [f"BufferController_assemble.write_data_{i}"])
```

Figure 8: Pseudo-code listing for defining mapping of SpMV operator on a Cambricon-X like DSA. Mapping embeds the machine code routines for configuring inputs to the DSA via controller interface at different intervals.

```
def SpMV(myDSA, J, K, neuron_vector, weights_matrix, output_vector):
  # Inputs: 1xK input neurons, JxK weights, 1xJ output neurons
  # DSA has 8kB NBin, 2B data, double buffered, 2k elements
  # DSA has 2kB SBs, 2B data, double buffered, 512 elements
  tc_j_S = largest_factor_within_threshold(J, num_PEs)        # Number of active PEs
  tc_k_L1 = largest_factor_within_threshold(K, 512)           # Iterations for processing data from weight
    ↪ buffer in PEs (L1)
  tc_k_L2 = largest_factor_within_threshold(K // tc_k_L1, 4)  # Iterations for processing elements from shared
    ↪ (L2) buffer
  tc_k_L3 = K // (tc_k_L1 * tc_k_L2)                          # Iterations for off-chip memory (L3) accesses
  tc_j_L3 = J // tc_j_S                                       # Iterations for off-chip memory (L3) accesses

  for k_L3 in range(tc_k_L3):
    # DMA transfers for input neurons
    start_address, burst_size = get_neuron_indices_L3(K, tc_k_L3, k_L3)
    read_dma(myDSA, neuron_vector, start_address, burst_size)
    # DMA transfers for weights (multiple accesses due to non-contiguous bursts for different PEs)
    for j_L3 in range(tc_j_L3):
      start_address, burst_size, offset, num_invocations = get_weights_indices_L3(J, K, tc_j_L3, j_L3, tc_k_L3,
        ↪ k_L3)
      for access in num_invocaitions:
        read_dma(myDSA, weights_matrix, start_address + access*offset, burst_size)
      for k_L2 in range(tc_k_L2):
        address_neurons = get_neuron_address_L2(K, tc_k_L3, tc_k_L2, k_L2)
        read_buffer_NBin(myDSA, address_nuerons)
        for j_S in range(tc_j_S):
            address_synapes = get_weights_address_L1(J, K, tc_k_L3, tc_k_L2, tc_j_S)
            # No L1 loop. Execution for a PE's data processing from synapse buffer is modeled in one shot.
            read_buffer_SB(myDSA, address_synapses)
            address_outputs = get_outputs_address_L2(J, tc_J_L3, j_S)
            write_buffer_NBout(myDSA, address_outputs)

      # DMA transfers for output neurons
      start_address, burst_size = get_outputs_indices_L3(J, tc_j_L3, j_L3)
      write_dma(myDSA, output_vector, start_address, burst_size)
```

Figure 9: Pseudo-code listing for configuring outputs from the controller for setting a DMA transfer.

```
def set_DMA_transfer(controller_obj, read_write_addr_offchip, read_write_addr_onchip, write_enable_n,
  ↪ burst_size, onchip_dest=0):
    controller_obj['DMA.write_enable_n'] = write_enable_n
    controller_obj['DMA.burst_size'] = burst_size
    controller_obj['DMA.read_write_addr_offchip'] = read_write_addr_offchip
    controller_obj['DMA.read_write_addr_onchip'] = read_write_addr_onchip
    if write_enable_n == True:
        controller_obj['DMA.in_dest_onchip'] = onchip_dest
```