Design Space Description Language for Automated and Comprehensive Exploration of Next-Gen Hardware Accelerators

Shail Dave
Arizona State University
USA

Aviral Shrivastava
Arizona State University
USA

ABSTRACT
Exploration of accelerators typically involves an architectural template specified in architecture description language (ADL). It can limit the design space that can be explored, reusability and automation of system stack, explainability, and exploration efficiency. We envision Design Space Description Language (DSDL) for comprehensive, reusable, explainable, and agile DSE. We describe how its flow graph abstraction enables comprehensive DSE of modular designs, with architectural components organized in various hierarchies and groups. We discuss automation of characterizing, simulating, and programming new architectures. Lastly, we describe how DSDL flow graphs facilitate bottleneck analysis, yielding explainability of costs and selected designs and super-fast exploration.

1 NEED FOR DESIGN SPACE DESCRIPTION
Design space exploration (DSE) of accelerators, especially for machine learning [5, 10], require efficient HW/SW co-designs that meet strict execution constraints [20, 21, 23]. The need for a single accelerator for multiple workloads necessitates bottom-up exploration.

ADL-based design approach: Recent frameworks explore designs of a certain architecture (e.g., systolic arrays, PEs sharing unified buffer that is filled by DMAs) [12, 19, 33, 35]. They describe architectural template in the ADL [4, 15, 17]. So, design process focuses on specific architectural organization (i.e., specific types of computational and memory units interconnected in certain ways and hierarchy), and hardware design space is limited to values of architecture’s hyperparameters [16, 37]. Execution costs are provided by either expert-manuevered analytical models for the architecture [7, 34] or synthesizing each design (which is time-consuming). Space of algorithm-to-accelerator mappings is also formulated based on the template [7, 19]. Thus, DSE frameworks lack following capabilities:

- **Exploring efficient solutions from broad design space:** Since design space gets restricted to the template architecture (e.g., consider one-level, shared buffer as a memory), a vast space of architectures is left unexplored (multi-level buffers, unified buffers, DMA ports instead of buffers), even if some can be more effective.
- **Reusability of design flow for novel, wide range of architectures:** Since design tools are developed for a single template, they can be incompatible with architectures from a broad space, which impacts their reusability. Because, when design space is broadened, such as by integrating new functionality or novel implementation of existing architectural components, previously maneuvered tools—execution cost models, simulators, and algorithm-to-architecture compilers are incompatible with enhanced architectures.
- **Explainability of explored designs:** Execution cost models and simulators [7, 8, 14, 19, 32] typically do not provide designers insights about obtained costs. Also, recent approaches for accelerator designs (e.g. [2–4, 11, 16, 17, 22, 25, 26]) use non-feedback or black-box optimization. It makes challenging to reason about the efficacy of the acquisition mechanism of DSE for lowering costs.
- **Quick DSE for dynamic invocations:** In addition to obtaining an efficient solution that meets tight constraints on execution costs, DSE needs to be quick when it is dynamic, e.g., deploying a new DNN on a reconfigurable platform (cloud and end-user). However, due to underlying non-feedback or black-box optimization, DSE requires thousands of trials (or several days) for vast design space.

2 ENVISIONED APPROACH: DSDL

2.1 Specifying Architectural Design Space
Comprehensive DSE needs an abstraction that can allow describing various architectures. So, we envision design space of accelerators as flow graphs, which are specified and explored through Design Space Description Language (DSDL). In the flow graph of an architecture, each node represents an object from primary components for computation, buffer, and control logic [5, 18, 24, 28–30], user-defined special functionality, or even a sub-graph consisting of such components (Fig. 1 illustrates an example). Edges represent interconnects of various bandwidths for a fixed or configurable communication from X source nodes to Y destinations; communication can be concurrent, sequential, or asynchronous. The node corresponding to the beginning and termination of execution (memory, storage, or I/O) is denoted as the root. Such abstraction allows building compute/buffer/communication hierarchy of arbitrary levels, integrating special functionality like for sparsity [5], formulating workgroups for synchronization and load balancing, and specifying

Figure 1: Flow graph of Cambricon-X [36] like accelerator.
super-nodes (top-level modules) for better interpretability. Plus, as algorithms are compiled as data flow graphs [1, 4, 6, 13, 30, 31], they can be conveniently mapped on flow graphs of accelerators.

Designers can specify the architectural space in DSDL (Fig. 2a) with: (1) set of components for nodes and edges; (2) parameters, which are counts of each component and lists of values of hyperparameters of the components; (3) legality constraints that specify, when formulating architecture, which kind of component must (not) be connected with which component and how. Designers can also specify constraints between hyperparameters of components (e.g., sizes of the buffers should double for a buffer-hierarchy towards the root); (4) optimization constraints for meaningful and optimized exploration, e.g., homogeneous architectures through alike child super-nodes, hierarchy (specific counts and interconnection of units); (5) IRs or dataflow graphs of target workloads. Moreover, like conventional DSE with ADL [16, 17, 27], DSDL users should specify (6) constraints on execution costs; (7) DSE objectives.

Flow graphs can be described with a library and APIs for modular construction of architectures (Fig. 2c). For reusable, automated system stack, architectural components are supplied with their definitions for various tools through a library (Fig. 2b). For each component class, different subroutines define various design flow steps (e.g., analytical cost, simulation functionality, program representation). Then, DSDL can construct these tools for each flow graph, as needed. Developers can extend the library with new components for special-purpose or high-level architectures to enable DSE of new architectures. Following principles drive our approach: • Modularity: An accelerator or a multi-accelerator architecture can be described with pre-defined or user-specified components. • Design Flexibility: Users can specify a comprehensive design space with various hierarchies and grouping of components. • Extensibility: Extending support for new components or new implementations of existing components should be possible. • Explainability: Extending support for new components or new implementations of existing components should be possible. • Explainability: Users can reason about contributions of components to overall execution costs and selection of optimal solutions. • Easy-to-Use: Users can specify, characterize, simulate, program, and explore designs with a few mouse clicks or lines of code (LOC). • Compatibility: Generated outputs (e.g., program representations) should be compatible with relevant tools (LLVM/MLIR).

2.2 Comprehensive DSE with DSDL

DSDL framework enables comprehensive DSE of accelerators by exploring various flow graphs for target functionalities. It invokes DSE with inputs from the designers, and it outputs an HW/SW design with minimized objective cost (or a Pareto front) while satisfying all constraints. Besides enforcing specified legality and optimization constraints, it uses in-built legality constraints, such as ensuring that a candidate architecture is suitable for target functionality. DSE is dual-mode: vertical DSE yields new architectures, and horizontal DSE optimizes hyperparameters of each. Explainable DSE with bottleneck analysis can enable smooth joint exploration.

2.3 Full-stack Automation for a Flow Graph

In-built methods for flow graphs (Fig. 2d) can automate visualization, performance/area/power characterization, program representation and mapping space generation, functionality simulation, and synthesis of an accelerator design. Some methods may require additional information, such as invocation of components (for analytical characterization), actual data (simulation), and an algorithm to be executed (compilation). DSDL automates a tool’s construction by parsing the flow graph and invoking subroutines module-wise.

Automating Execution Cost Modeling. For a flow graph, execution costs can be generated automatically from costs consumed by underlying nodes during execution. A parent’s cost is an aggregation of the costs of children. Aggregation depends on cost functions of edges (e.g., maximum/addition of latency, for concurrent/sequential execution over parallel paths; addition for area or power). After propagating costs through parents, the root provides the total cost. Automating Mapping Space Generation, Code Generation, and Simulation. Each node can be associated with program representation, and edges can be associated with invoking subroutines for communication (e.g., DMA). For instance, a buffer represents memory accesses, and a computational unit represents an arithmetic/logical operation(s) on a data stream (Fig. 2b). Combining such representations meaningfully can realize a collective code representation, transforming to which becomes necessary for executing any algorithm on the accelerator. For instance, consider accelerating matrix multiplication (triply nested loop) on an architecture with PEs sharing a unified buffer where PEs have a local buffer, and DMA fills data in the shared buffer from DRAM. The DSDL-generated mapping space contains all possible transformations (e.g., 12 tiled loops with (3)4 orderings [7, 19]). Simulation of an accelerator occurs as a dataflow through the flow graph. Control triggers for a looped execution and live-in/out data are communicated through a synthetic controller, which also simulates non-accelerator functionality.

2.4 Explaining Costs by Bottleneck Analysis

Execution cost models get available by processing a flow graph. Such processing can provide information about the costs consumed by different datapaths of a design’s graph (e.g., computation time vs. DMA time [5, 9]) and relevant execution characteristics (e.g., total loop iterations invoked, size of transferred data, data reuse). Thus, it facilitates a systematic and accurate bottleneck analysis of the design, improving reasoning about the design’s efficacy.

2.5 Dynamic DSE Empowered by Explainability

Explainability informs bottlenecks behind the high costs of a design and how to mitigate them. DSE using bottleneck analysis iteratively explores among effective candidates and avoids arbitrary trials. Hence, it can be super-fast, especially for optimizing an architecture.
ACKNOWLEDGMENTS

This work was supported in part by NSF under Grant CCF 1723476 — NSF/Intel Joint Research Center for Computer Assisted Programming for Heterogeneous Architectures (CAPA).

REFERENCES


