An Agile Methodology for Designing Efficient Domain-Specific Architectures

by

Shail Dave

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

August 2024

ABSTRACT

Domain-specific architectures are designed increasingly for efficiently processing compute- and data-intensive workloads, including machine learning. An effective design methodology is required for these accelerators, as application executions impose stringent constraints on architecture designs and execution costs. In a common approach, experts design an architecture template and its dataflow (how accelerator processes applications) and tools like power/performance/area cost models and simulators for each architecture. Design hyperparameters are optimized by black-box explorations that take thousands of trials or weeks. This methodology limits design's efficiency, designer productivity, and methodology's applicability. Firstly, the lack of exploring hardware/software configurations from a broad design space limits efficiency of obtained solutions, and thereby, operational efficiency and end-user experience. Lacking agility in design explorations can consume a lot more time and resources, degrade sustainability, and restrict wide-scale resource management of emerging applications that require dynamic explorations. Lack of agile design tools delays time-to-market while also creating accessibility challenges. Lacking explainability of design costs and decisions in exploration process can degrade designer productivity and design sampling/tuning.

This dissertation introduces an effective design methodology towards designing efficient next-generation accelerators in an agile, explainable manner. Firstly, it proposes to formulate comprehensive hardware/software codesign space for architectures. Holistic mapping space formulation for spatial and temporal execution enables determining adaptive dataflows, and inclusion of a broad range of architectures allows exploring efficient, constraints-meeting solutions. Second, bottleneck characterization of obtained solutions and bottleneck model-guided design optimization brings explainability in the design exploration process. It helps reason about the effectiveness of sampled solutions during the exploration process, systematically reducing execution costs. Lastly, this dissertation proposes developing design and characterization tools around an accelerator abstraction (flow graph) and a methodology towards such an automated development. The effectiveness of the proposed methodology is shown through hardware/software codesigns of edge ML accelerators for recent computer vision and language models. Evaluations show that the proposed methodology can find 6$\times$ efficient (low-latency) codesigns in minutes–hours, as compared to days–weeks taken by previous design exploration approaches.

DEDICATION

To my dearest wonderful parents, Harita and Janak Dave,

and to Krishna,

and all my well wishers.

ACKNOWLEDGMENTS

"It takes a village to raise a PhD graduate."

This dissertation would not have been possible without unflinching support, love, guidance, and companionship of many individuals and well wishers throughout my journey. First and foremost, praise be to the Almighty, for the love and blessing and guidance over all the hurdles of life, specially when there seemed no light in the sight during facing challenges personally or professionally. Likewise, any words are insufficient to express my gratitude for unconditional love and support and inspiration blessed by my parents on day-to-day basis. Mom and dad - you have motivated and supported me to pursue this journey and accomplish it, and you were always there when I thought things didn't work well. It made my journey smooth and enjoyable.

I owe tons of thanks to Professor Aviral Shrivastava for being my advisor and motivating me to pursue the doctorate degree and always supporting me all these years in numerous ways. His insightful and forward-looking comments and unwavering support throughout my graduate program journey has greatly contributed to my learning and this dissertation. He always provided me ample freedom to freely conduct research and embark on any tough challenges and try different approaches, even when needles weren't moving much. I am also grateful to all my collaborators, committee members, and topical course instructors whom I have pleasure to work with on my papers, new projects, or learn from them in-class or outside – Prof. Tony Nowatzki, Prof. Fengbo Ren, Prof. Baoxin Li, Prof. Jae-Sun Seo, Dr. Sasikanth Avancha, Prof. Michael Spear, Dr. Xiaochen Guo, Prof. Kyoungwoo Lee, and Dr. Partha Biswas.

I am also very grateful to know or work with some of the past and current lab members of Make Programming Simple / Compiler Microarchitecture Lab – especially Dr. Moslem Didehban, Dr. Yooseong Kim, Dr. Mahesh Balasubramanian, Dr. Mohammad Khayatian, Prof. MohammadReza Mehrabian, Dr. Reiley Jeyapaul, Prof.

Jongeun Lee, Dr. Hwisoo So, Dr. Youngbin Kim, Dr. Jing Lu, Matthew Szeto, Francis Mendoza, Sanggu Park, Mohsen Sheibanian, Abhinav Kumar, Rishabh Kashyap, Sai Shashank, Atharva Khedkar, Adam Awale, Edward Andert, Sumedh Joshi, Kaustubh Harapanahalli, Vinayak Sharma, Abhishek Risheekesan, Saleel Kudchadker, Bryce Holton, Rachel Dedinsky, Dheeraj Lokam, and Dipal Saluja. All these friends have guided me, helped me with brainstorming or on my projects, cheered me, shared fun moments, or extended help in various ways. I am also grateful to all the support I received from other researchers in the Center for Embedded Systems (CES) and the Center for Intelligent, Distributed, Embedded Applications and Systems (IDEAS) – Prof. Sarma Vrudhula, Prof. Heni Ben Amor, Prof. Nakul Gopalan, Dr. Carol-Jean Wu, Prof. Hokeun Kim, Prof. Aman Arora, Dr. Georgios Fainekos, Gian Singh, Ayushi Dubey, Souroush Heidari, Prof. Mahdi Ghasemirahaghi, Dr. Ankit Wagle, Dr. Elham Azari, Dr. Shakiba Yaghoubi, Dr. Akhil Arunkumar, Dr. Jhe-Yu Liou, Dr. Simon Stepputtis, Dr. Kevin Luck, Zifan Yu, and Kaustubh Mhatre. Many thanks goes to my friend Bhavin for his friendship and for always cheering me up.

I cannot thank enough to the wonderful staff at ASU, who has been helpful with program advising, payroll, travel, and many tasks – Monica Dugan, Pamela Dunn, Dr. Araxi Hovhannessian, Christina Sebring, Arzuhan Kavak, Jaya Krishnamurthy, Lisa Christian, Theresa Chai, Marianne Reyes, Vicki Fox, and SCAI IT for the IT support.

Many thanks to Graduate College, School of Computing and Augmented Intelligence (SCAI), Graduate and Professional Students Association (GPSA), National Science Foundation (NSF), Intel, Semiconductor Research Corporation (SRC), IEEE, and ACM for providing to me conference travel grants, funding my research, or recognizing my works. Last but not the least, discussions with many researchers and practitioners and learning from them or indirectly from their content have shaped my journey, which is just a mere attempt towards standing on the shoulder of the giants.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Design Methodology for Domain-Specific Architectures

1.1.1 Domain-Specific Architectures

With increased computational and memory requirements of the applications and the need for their efficient processing, domain-specific accelerators have been increasingly designed, especially for machine learning (ML) [6–8]. ML models implement intelligence in computing systems. Hence, different ML models are widely used in several important domains, including computer vision (object classification [1, 9, 10] and detection [11, 12]), natural language processing [13–15, 15], quantitative reasoning [16], media generation [17], recommendation systems [18, 19], medical diagnosis [20, 21], large-scale scientific computing [22], embedded systems [23], mobile and edge processing [24, 25], and even for designing or optimizing hardware and software systems [26–28]. *Domain-specific architectures (DSAs)* can significantly speed up their execution in an energy-efficient manner [4, 29–31].

1.1.2 Design Methodology Requirements

Domain-specific application execution scenarios impose stringent constraints on the accelerator design and the execution costs while requiring the minimization of some of these costs. Such design and execution costs include latency, energy, throughput, storage, monetary costs, or error tolerance for functionality outputs (Fig. 1.1; MLPerf [32], [33, 34]). Efficient design methodology and exploration are required, as the accelerator resources and resultant constraints on costs vary dramatically based on

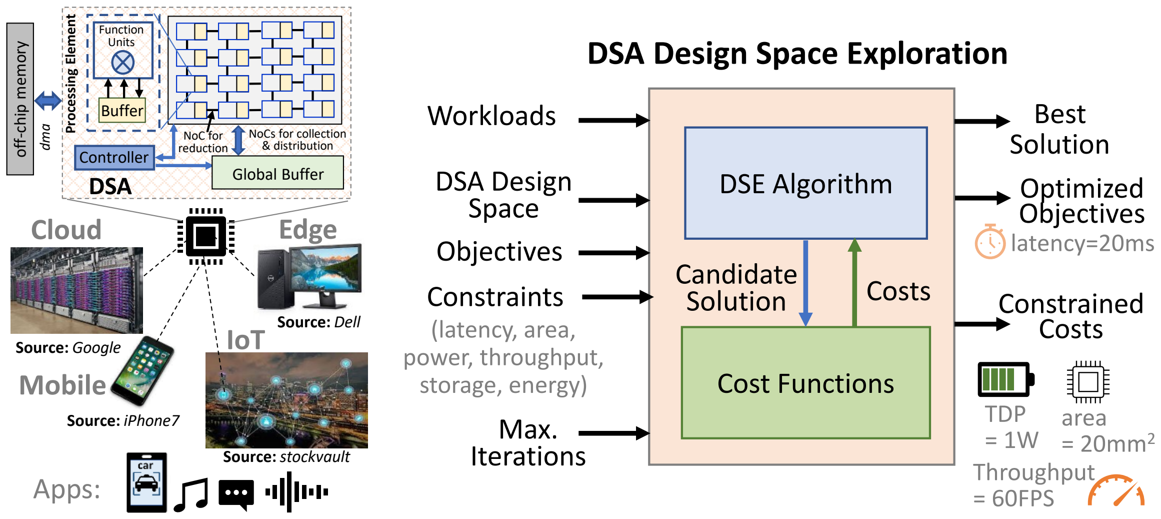


Figure 1.1: Exploring efficient hardware/software codesigns of DSAs under different application execution scenarios.

various application execution scenarios (Table 1.1, adapted from [35]). For instance, a cloud accelerator could have petabytes of memory and 100s of Watts of power and thousands of dollars as the price point, whereas a mobile DSA can only afford a few GB of memory, a few watts of power budget, and cost 100 dollars. Thus, designing efficient domain-specific accelerators is essential not just for smooth end-user experience (e.g., users of mobile phones, wearables, cloud instances) but also for low design and operation costs, and thereby, from energy consumption and sustainability [1] perspective. In fact, recent research has shown that most of the carbon footprint of processors stems from their design and manufacturing process. So, the time and energy of humans and computing that are used to design efficient accelerators need to be utilized effectively. Therefore, this dissertation advocates for the design methodology

[1]Sustainability, as referred to in this dissertation, can be improved through i) reusability of the tools and methodology, 2) agility of the development, i.e., notably lowering the usage of computational and human resources taken by the design development and exploration, and 3) efficiency of the design, i.e., considerable reduction in the operation costs. Some studies like [36, 37] indicate that a higher potential for increasing sustainability of the processors can be in lowering their development costs, as compared to reducing the operating costs after deployment.

Table 1.1: Constraints for accelerator design for CloudML, MobileML and TinyML.

Platform	Memory	Storage	Power	Price
CloudML	16-256 GB HBM	TB ~ PB SSD/Flash	100s of W	several $1000s
MobileML	8-128 GB DRAM	128-512 GB Flash	<10W	a few $100s
TinyML	100s of KB	few MBs of eFlash	<0.5 W	<10$

that not just results in an efficient accelerator but one that is "agile" by itself, i.e., it takes less time. The design and operation of accelerators need to be sustainable as well, i.e., the design phase should be reusable in the longer term and take less time/resources while producing the most efficient (e.g., high-throughput and low-energy) architecture designs.

1.1.3 Design Methodology Components

The process for efficient accelerator design involves not just coming up with the most efficient microarchitecture or hardware-level optimization (e.g., power-gating [38], zero-skipping [4, 39]) but also requires the hardware/software co-design and development of the corresponding tools. It involves 1) Defining the design space for the application's execution on the accelerator architecture, 2) Tools for quantifying and characterizing the application's execution efficacy, and 3) Techniques for effectively navigating the (vast) design space for tuning the architecture design and workload execution. The design space usually contains the hardware space – variations in the hyperparameters of the accelerator design and mapping space – different ways of processing application functionality onto the accelerator's resources spatially and temporally [40]. When co-designing the accelerator with the applications (e.g., ML models), the design space can include the "data space" or "algorithmic space" as well, corresponding to the variations in the data (storage format, bit-width, sparsity) or functionality that needs to be computed (e.g., number/organization of different ML operators), respectively. Development of the design tools involves developing PPA cost models that quantify power, performance (latency/throughput), and die area of the accelerator and simulation to ensure functionality and cycle-level execution estimates. Exploration techniques need to navigate (usually vast) design space in a systematic manner, iteratively evaluate the effectiveness of processing applications on a selected accelerator design (with chosen hardware/software configurations), and find the (Pareto) optimal solution.

1.1.4 *Current Approaches*

A commonly used approach for designing the accelerators is template-constrained development and black-box design optimization. Teams of expert designers define an *architecture template*, e.g., systolic/spatial architecture as a standalone or near-data processor. They also define how application functionality can be processed on it for desired acceleration (aka ”dataflow”). For example, Fig. 1.2(a) shows a template

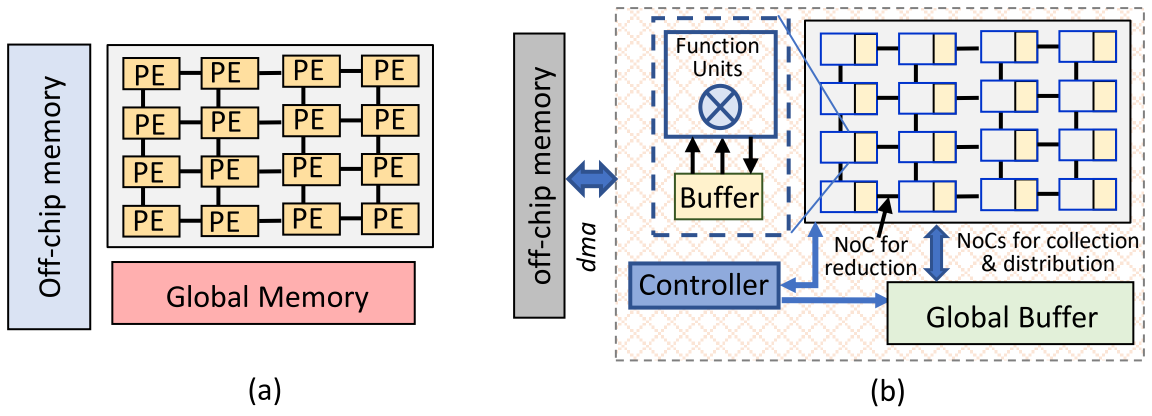


Figure 1.2: Spatial architecture templates for designing domain-specific accelerators. (a) Google TPU-like systolic array accelerator. (b) Eyeriss-like spatial architecture with configurable interconnects and buffers.

architecture for Google's TPU-like accelerator using a systolic array [30, 41], and Fig. 1.2(b) shows an Eyeriss-like spatial architecture [4], where processing elements (PEs) share a unified buffer that is filled with the required data from off-chip memory by direct memory access (DMA) transfers. Such accelerators have processed the executions of ML model operators (e.g., CONV2D, GEMM) with fixed dataflows such as weight stationary (in TPU [30], DLA [42]), output stationary (in [43], ShiDianNao [44]), or row stationary (in Eyeriss [45]).

Execution costs (power, performance, area or PPA) for these accelerators are obtained by either expert-maneuvered analytical models for the specific architecture [31, 40] or synthesizing each design (which is time-consuming). In fact, some of the commonly used analytical models (contemporary or posterior models of the one discussed in this dissertation later) do not even consider non-computation latency, e.g., on-chip/off-chip data communication latency (in Scale-Sim [46]), implications of non-contiguous data accesses (in Timeloop [47]), and stalls encountered by non-pipelined computation and communication (e.g., in reduction of partial summation vectors, which may not be pipelined with computation of multiplications or communication of inputs/outputs of an ML model's layer). In addition to manually developing power/performance/area cost models, the experts need to also build other design or evaluation tools like functionality/cycle-level simulators and mappers for the domain-specific architecture *manually*.

The design space for a domain-specific architecture or an accelerator is usually defined as the ranges of values corresponding to the hardware design hyperparameters. Techniques for optimizing the accelerator design use non-feedback such as grid search [48, 49] and random search [50] or use black-box optimizations like genetic algorithm [51–57], simulated annealing [58–60], Bayesian optimization [2, 41, 61–64], or reinforcement learning [65–68]. For narrowing down the search space, such techniques usually specify a limited number of parameter values, e.g., power-of-two-numbers or a discrete set of values, e.g., in [41, 61, 65, 69]. Further, for exploring efficient and constraints-meeting solutions in the vast design space (e.g., quadrillions of hardware configurations [41]), most techniques (e.g., [61, 65–68, 70–74]) usually fix the dataflow, i.e., how application functionality can be executed spatially and temporally on accelerator resources. Further, among possible hardware hyperparameters, such techniques (e.g., [61, 65, 69] to list a few) explore only some hyperparameters, such as those for designing the computational and memory units.

Such an template-constrained design and black-box optimization approach is common. For example, such design methodology aspects have been used by NVIDIA MAGNet [75] and NVDLA [42], Google full-stack TPU exploration [41], Intel SuSy [76], Harvard SODA [77], ARM SCALE-Sim [46], and likely in designing even several NPUs or Neural Processing Units for accelerating AI/ML models, e.g., [29, 30, 78–85].

1.1.5 Challenges and Limitations

While the current approach of expert-driven static design has led to reasonable accelerator designs – some of which even empower many applications around us today – it limits design's efficiency [2] , designer productivity, methodology's applicability, and sustainability.

1) Not exploring efficient solutions from a broad design space: A major challenge is in limiting the design space since it limits the hardware and/or software configurations that can be explored at all and consequently, the efficiency of the overall solution. For example, restricting the execution to a specific dataflow (e.g., input or row stationary) limits the spatial parallelism that can be achieved for different ML

[2]Throughout the dissertation, the efficiency of the execution/design refers to the optimization objectives such as achieving low latency or energy consumption. Whenever unspecified, assume latency minimization as an example.

operators, interleaving the communication latency with computations on an accelerator with specific NoC (network on chip) bandwidth, the data reuse that can be effectively exploited through buffers, and thereby the overall latency or energy consumption. Similarly, limiting the hardware design space to exploring just some of the hardware hyperparameters or specific architectures restricts the execution's efficiency. Further, not including the software/mapping configurations in the accelerator design space can lead to accelerator designs that are inefficient or incompatible with the targeted software configurations. Lastly, restricting the hardware space to design hyperparameters limits the accelerator design to the expert-defined template architecture. As a result, a vast space of architectures is left unexplored, even if some can be more effective. For instance, the template architecture could contain a shared, unified buffer, even though unexplored architectures with small, private buffers and unicast links could be much more efficient for no-reuse, memory-bounded workloads. Thus, limiting the space of hardware/software (co)designs impacts not just the quality of designs obtained in the price budget, but also the operational efficiency and the end-user experience.

2) Lacking agility in the design flow: Current approach requires a lot more time and resources for design explorations, degrading sustainability and increasing operation costs and time-to-market. For example, when exploring vast design space (e.g., millions to quadrillion hardware/software configurations [41, 86]), existing design space explorations (DSEs) take thousands of trials or days/weeks [2, 41, 65, 69]. They increase the time-to-design and operation costs. In fact, several applications requiring dynamic DSE cannot be supported well, e.g., deploying an ML model dynamically on a reconfigurable platform in the cloud or at the edge. Another example includes runtime resource management of city-wide, reconfigurable, edge infrastructure for processing smart-city applications like detection of traffic incidents [87] or haboobs. Moreover, the need for template-specific design and characterization tools for every accelerator architecture requires months of development efforts [88]. It notably delays the time-to-market.

3) Lack of explainability of design costs and decisions: Current approach cannot explain costs obtained from the underlying system and design decisions in the exploration flow, affecting the productivity of designers and development costs. For example, when designers use an analytical cost model [46, 47, 89] or a simulator [90, 91] to characterize an application's execution on an accelerator design, they do not reason about the inefficiencies that cause high costs. So, it becomes hard to understand the implications of variations in the workloads, hardware configurations, and software optimizations, i.e., why a specific hardware/software configuration leads to a specific execution cost and how. Further, current accelerator design exploration mechanisms cannot explain the optimality or efficiency of obtained design solutions, especially when the search space is so vast that doing the brute-force exploration is impractical to demonstrate the optimality of the solution. For instance, a random search or a black-box exploration makes it hard for designers to infer why the obtained solution is the most efficient (or among the top solutions) within the constraint budget. Thus, the lack of explainability of the design costs and design decisions affects the designer's productivity, as fine-tuning the design becomes challenging. It also usually leads to excessive sampling. Excessive sampling means not only very high exploration time (unsuitable for dynamic DSE required by several application executions) but also solutions of likely poor quality within some exploration budget. It is because the obtained valid solution may not be very efficient with the high exploration budget used by excessive and often ineffectual trials, which are generated without understanding the inefficiencies behind the obtained cost and implications of varying parameters on the design/execution costs.

4) Less reusability: The current approach develops design tools and related system stack specific to an architecture template, including PPA cost models, mapper, and simulator. As workloads evolve (e.g., new kinds of AI models are developed [33, 92]) and new architecture templates are designed, expert designers need to develop the new tools from scratch [93]. Thus, this approach limits sustainability, designer productivity, and efficiency of the design. It is primarily due to the fact that the design tools are developed for a single template, and hence, they can be incompatible with architectures from a broad space, which impacts their reusability. For instance, when architecture needs to be enhanced to facilitate evolving AI workloads [33], the design tools for the previous template *cannot be heavily reused* and needs heavy rework. A mapper for processing language models [14, 15] on systolic arrays [41] differs considerably from the one that needs to be developed for a spatial architecture for vision or graph learning models (unstructured sparse) [33]. Likewise, PPA cost models or simulators for different ML accelerators with scalar/vector PEs [4, 94] and accelerators/dataflows for different ML operators [33, 95], FFTs [96], etc. are developed separately. Such heavy development requirements also put a huge accessibility toll on resource-constrained researchers, e.g., graduate students. This marks the need for more agile design methodology.

1.2 Overview of the Proposed Approach, Contributions, and Dissertation Organization

This dissertation introduces an effective design methodology, as illustrated in Fig. 1.3, for designing and evaluating efficient domain-specific architectures in an agile and explainable manner. In particular, it makes the following key contributions.

1. **Making Design Space (More) Comprehensive (Chapter 2).** Instead of an experts-tailored design of the accelerator or its dataflow, this dissertation advocates for formulating a comprehensive design space for the hardware/soft-

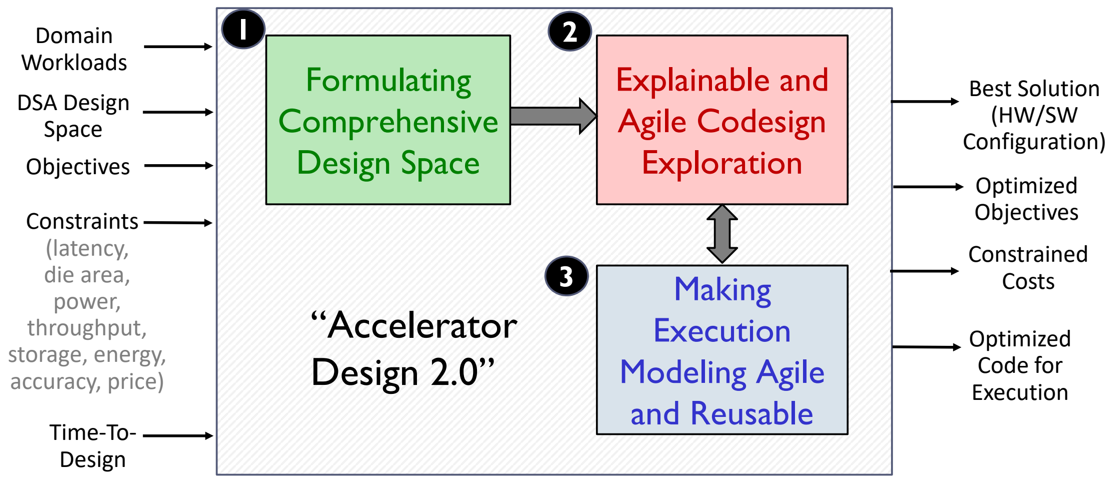

Domain Workloads
DSA Design Space
Objectives
Constraints (latency, die area, power, throughput, storage, energy, accuracy, price)
Time-To-Design
1
Formulating Comprehensive Design Space
2
Explainable and Agile Codesign Exploration
3
Making Execution Modeling Agile and Reusable
"Accelerator Design 2.0"
Best Solution (HW/SW Configuration)
Optimized Objectives
Constrained Costs
Optimized Code for Execution

Figure 1.3: Overview of the proposed accelerator design methodology.

ware codesign of domain-specific accelerators. For example, instead of using specific dataflow for processing *deep learning (DL)* models on accelerators, it formulates comprehensive mapping space for the spatial and temporal execution of the deep learning models, improving the efficiency (e.g., reduction in the energy-delay product) by an order of magnitude. Likewise, instead of limiting the optimizations of the hardware configurations to just computational or memory units, it considers various relevant parameters, e.g., interconnect links and bit-width and a variety of sparsity-exploiting components. Further, one key aspect of this work is that it demonstrates joint hardware/software codesign. The codesign is done such that the executions of the workloads are optimized to utilize the hardware's resources in the most effective manner, and the hardware exploration, in turn, is tailored for mitigating the inefficiencies left after the software-optimized execution.

2. **Enabling Explainability and Agility in the Design Space Exploration by Using Bottleneck Analysis (Chapter 3).** This dissertation proposes to

use bottleneck models to drive the design space exploration of domain-specific architectures. Proposed bottleneck model is a graph-based abstraction to analyze the costs of executing workloads on accelerators in an explicit and automated way. Applying bottleneck analysis over such a cost graph or the bottleneck model helps pinpoint bottlenecks and the associated hyparameters requiring further optimization. Further, an API and a generalized framework is proposed that can work with different bottleneck models and enable bottleneck analysis-driven design space exploration. By using the information about the cost factors through the proposed abstraction, the framework can reason why a certain hardware/software configuration of an accelerator leads to specific costs. Its applicability is demonstrated through the cost analysis and design optimizations for executing computer vision and language models on an edge accelerator with Eyeriss-like [4] spatial architecture.

3. **Improving Reusability of Design Tool Development for Accelerators (Chapter 4).** This dissertation advocates for developing design and characterization tools around an abstraction of accelerators, instead of building PPA cost models, mappers, and simulators from scratch for every new domain-specific architecture. It proposes an architectural flow graph-based abstraction and a methodology for such development, e.g., developing automated latency estimation incorporating architectural details and bottleneck characterization, including for dense/sparse tensor computations.

1.3 Dissertation Statement

The domain-specific architecture design space exploration should be: Comprehensive for efficient hardware/software codesign; Explainable for enhanced efficiency and designer productivity; Agile for enabling dynamic optimizations, better reusability, and better time-to-market.

1.4 Dissertation Impact

The contributions of this dissertation and the author's research and development efforts towards the proposed design methodology have led to many follow-up works, adoptions, invited talks, honors, and a comprehensive survey by the author on the efficient processing of the sparse and compact machine learning models on the domain-specific accelerators.

Follow-up Works and Adoptions: Techniques for efficiently mapping applications on DSAs (both for perfectly nested loops of ML operators [40, 86] and general-purpose computing [97]) have been recognized by many works and some of the top researchers in the research areas as state-of-the-art. They studied these techniques quantitatively and/or qualitatively and even used them in developing follow-up works (e.g., [98–100], to list a few). These also include master's theses and doctoral dissertations at several top universities and recent industrial research such as [41, 101–104]. The adopted or follow-up works have been regularly published at the flagship conferences and journals in the domains of computer architecture, design automation, embedded systems, parallel and distributed computing, code optimization, machine learning, and computing in general. The author's compilation and simulation frameworks implementing these techniques [40], [91] have regularly received several downloads every month and about 60 stars on GitHub. Some of the techniques and works developed throughout this dissertation have been studied in seminar classes at top research universities in the U.S. and also referenced by recent books and surveys on related topics (e.g., books in the Synthesis Lecture Series in Computer Architecture by Morgan & Claypool [105, 106]). Overall, the proposed techniques altogether have received over 400 citations as per Google Scholar, Semantics Scholar, etc.

Honors, Awards, and Invited Talks: Some of the techniques and works developed throughout this dissertation have been invited and presented in the premier international forums, including ARM Research Summits, Annual Days of NSF/Intel CAPA Research Center (Computer Assisted Programming for Heterogeneous Architectures), Future Chips Forum, SRC Techcon, IBM/IEEE CAS AI Compute Symposium. They have been highlighted on or featured by various organizations, media, magazines, and social media, including ACM Tech news, Communications of the ACM blog, ASU news, insideHPC, IEEE Bridge and IEEE Eta Kappa Nu (HKN), DeepAI, Hacker news, and Intel Labs Select Publications. Some of these works have also received several competitive awards such as Outstanding research awards at ASU and a Silver Medal in ACM Student Research Competition. The challenges laid out by this work for agile development has also affected chartering the need and vision for the sustainable computing [37].

Literature Survey: The author's comprehensive survey on the efficient processing of sparse and compact machine learning models and designing efficient accelerators for such models was published in Proceedings of the IEEE [33]. It has been received well and widely studied and recognized by various communities (computer architecture, machine learning, etc.), including senior researchers (e.g., research works [107–110]). The information from the survey has also been studied or used by several master's theses and doctoral dissertations at top universities, including ETH Zurich, Harvard, Purdue, and the University of California at Santa Barbara.

Chapter 2

MAKING DESIGN SPACE COMPREHENSIVE

The efficiency of the domain-specific architecture's design heavily depends on the architectural components that are tailored for the set of workloads as well as the configurations of the design parameters and the obtained code (mapping) optimization. A major challenge with the existing design approaches, including those for the machine learning accelerators, is that they limit the hardware/software design space. This chapter discusses the need for making the design space comprehensive so that a vast range of hardware and/or software configurations can be specified and explored, which in turn, could allow the designers to achieve highly efficient solutions for the set of target workloads.

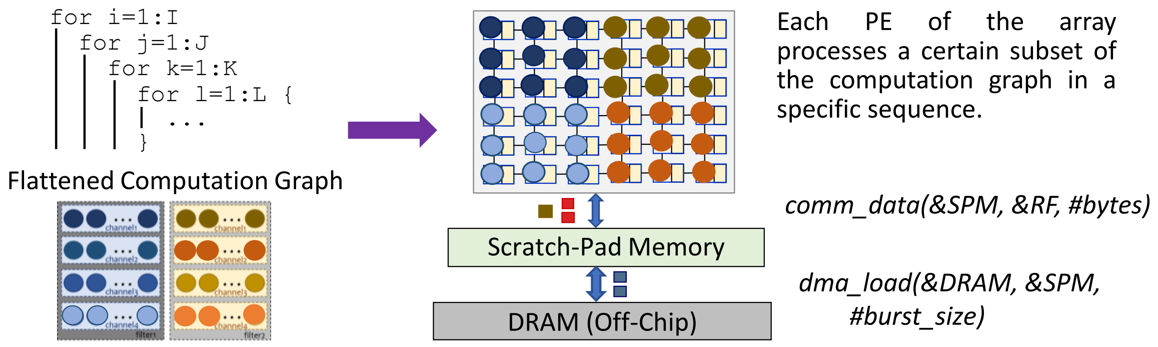
Domain-specific architectures typically accelerate loops of the workloads by executing loop iterations spatially onto PEs in parallel and managing the data accesses from local/shared register files or buffers and off-chip memory. Prior design approaches restricted the workload executions to specific dataflows, e.g., output stationary or weight stationary or row stationary when designing architectures for dense/sparse deep learning models. It limits the spatial parallelism that can be achieved for different ML operators or loop-kernels, interleaving of the NoC communication latency with computations, the data reuse that can be effectively exploited through buffers, and thereby the overall latency or the energy consumption. Contrarily, this chapter demonstrates how to formulate a comprehensive mapping space representation for optimizing the spatial and temporal execution of the loops in the workloads, which could cover the vast space of loop tilings/orderings/parallelism. It also shows how exploration of such a comprehensive design space improves the efficiency (e.g., reduction in the energy-delay product) by one order of magnitude. Such comprehensive mapping formulation can also help achieve better hardware/software codesigns that can flexibly accelerate a variety of workloads.

2.1 Background

At the heart of several important-to-accelerate applications, e.g., multimedia, imaging, and deep learning are *perfectly nested loops*, which are often compute- and memory-intensive. A perfectly nested loop is a nested loop, where all the assignment instructions are inside the innermost loop. For example, the convolution kernel (that executes for the majority of execution time in computer vision models (e.g., ResNet152 [1], ResNeXt [111], EfficientNets [112]) is a 7-deep perfectly nested loop. Likewise, general matrix multiplication (GEMM) is the primary operation in Transformers [13] and language models based on Transformers e.g., BERT [14].

Recent research efforts and commercial solutions have extensively demonstrated that these power and performance-critical loops can be efficiently accelerated on dataflow accelerators. Typically, these domain-customized accelerators feature *spatial architectures*, which are those that expose low-level aspects of the hardware's interconnect and storage to the hardware-software interface. Spatial architectures can be coarse-grained or fine-grained. Coarse-grained architectures feature arrays of interconnected PEs, and fine-grained designs are realized by programming FPGAs. *Coarse-grained spatial architectures* are a common implementation choice for designing domain-specific hardware accelerators including for ML, e.g., *systolic arrays* are used in Tensor Processing Unit [29, 30, 113–115] and specialized spatial architectures in Eyeriss [45] and [54, 84, 88, 116–120].

As Fig. 2.1 illustrates, the accelerator usually comprises an array of PEs (*processing elements*) that may contain private *register files (RFs)* and shared buffers or a



for i=1:I
for j=1:J
for k=1:K
for l=1:L {
...
}
Flattened Computation Graph
Each PE of the array processes a certain subset of the computation graph in a specific sequence.
comm_data(&SPM, &RF, #bytes)
Scratch-Pad Memory
dma_load(&DRAM, &SPM, #burst_size)
DRAM (Off-Chip)

Figure 2.1: Programming the dataflow accelerators requires explicit management of computational, memory, communication, and control resources.

scratchpad memory (SPM). PEs are simple in design (functional units with little local control), and the shared scratchpads or the buffers are non-coherent with software-directed execution. Therefore, these domain-specific architectures or accelerators are a few orders of magnitude more power-efficient than out-of-order CPU or GPU cores [4, 29, 30]. They lead to highly energy-efficient execution of ML models that are compute-intensive and memory-intensive. Performance-critical tensor computations (e.g., of ML models) are relatively simple operations like element-wise or tensor additions and multiplications. So, they can be processed efficiently with structured computations on the PE-array. Moreover, private and shared memories of PEs enable high temporal reuse of the data [33, 40, 121]; with efficient data management, PEs can be continuously engaged in tensor computations while the data is communicated via memories [45]. Additionally, interconnects like mesh or multicast enable data communication among PEs and spatial reuse of the data, lowering the accesses to off-chip memory. Thus, with minimized execution time, spatial-architecture-based hardware accelerators yield very high throughput and low latency for processing domain workloads like ML models [4, 30, 122].

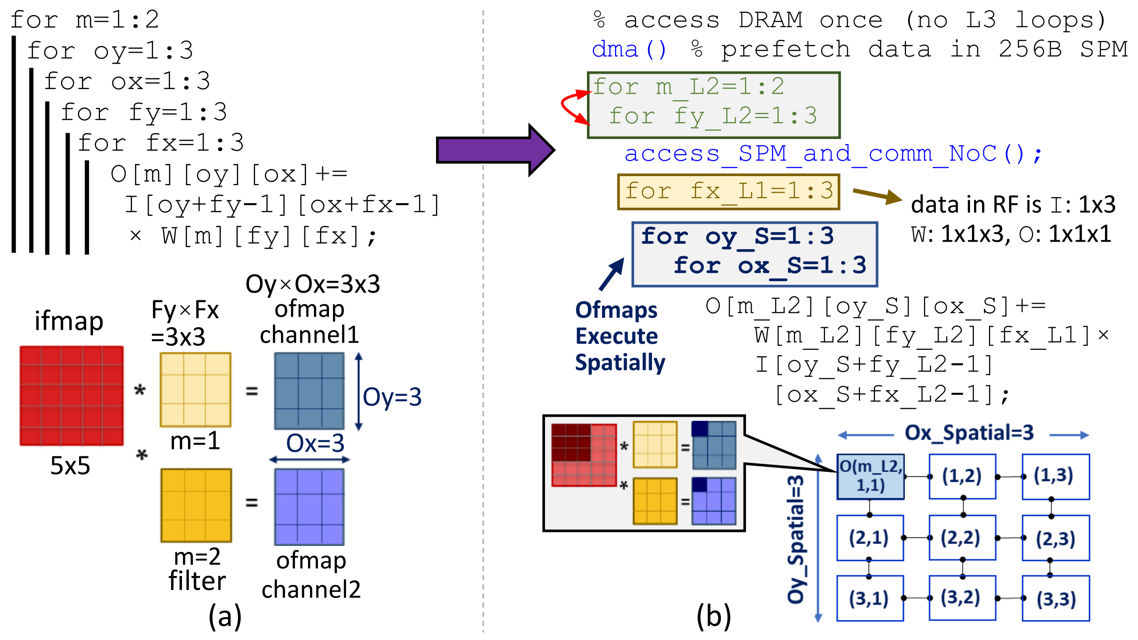
However, how to discover the most efficient way to execute a perfectly nested loop of an application onto the computational and memory resources of a given dataflow accelerator (execution method) remained an essential and yet unsolved challenge. This is because, the joint search space of hardware design of the accelerator, combined with the ways to execute the loops both spatially and temporally on it, is vast. In other words, not only the architecture can be configured in many different ways, but for each of those configurations, the number of ways to answer questions like – how to divide the loop execution among PEs, which PEs processes what subset of the data and in which sequence, when to schedule the data movement between memory-levels of the accelerator (for data prefetching), and how much buffering to do in the buffers or on-chip memories – are numerous.

The different ways in which a perfectly nested loop can be executed on the dataflow accelerator are referred herein as *execution methods*. When a programmer chooses a way of spatiotemporal execution of the loop-nest, that leads to a particular execution method. – Execution methods significantly impact the computation and communication patterns within the accelerator and therefore, the power and performance of the execution. – If they are not optimized/chosen well, acceleration benefits may even be negative! In the absence of a systematic and explicit way to capture and explore vast design space, prior techniques have considered only certain execution methods (like row-stationary [45], output-stationary [43, 44, 123] mechanisms for convolutions or GEMMs). Hence, they end up exploring only a tiny fraction of the mapping space, during manual tuning [88] or randomization-based search [54, 124].

2.2 Spatiotemporal Execution of Loops on Dataflow Accelerators

The efficiency of executing a perfectly nested loop onto a domain-specific architecture or a dataflow accelerator depends on the execution method which defines the spatiotemporal organization of loop iterations. If all loop iterations are processed simultaneously on different PEs, then execution would finish in one shot. However, due to a limited number of PEs, only some loops are (partially) executed in space, and remaining loops iterate temporally on each PE. For example, consider the loop-nest of Fig. 2.2(a), which is a simplified convolution kernel. It shows that a convolution of a 5×5 *input feature map (ifmap)* with 3×3 weights of two filters yields two output channels of 3×3 *output feature map (ofmap)*. All data elements are of 16 bits. Now, Fig. 2.2(b) shows one execution method to map the nest of Fig. 2.2(a) onto a sample dataflow accelerator consisting 3×3 PEs, where each PE accesses own 16B RF and a 256B shared SPM. For example, executing the loop with an *index variable (IV)* `ox` in space requires a row of 3 PEs in the accelerator. Similarly, spatially executing both the loops with IVs `ox` and `oy` requires 3×3 PEs. Here, each PE computes a unique ofmap value `O(m_L2,oy_S,ox_S)` while temporally executing loops with IVs m, `fx`, and `fy`. PE(1,1) corresponds to `oy_S`=1 and `ox_S`=1. So, PE(1,1) processes `O(m_L2,1,1)` and requires ifmaps I(1,1)–I(3,3) and all the weights W(1,1,1)–W(2,3,3). In contrast, if some other execution method corresponds to executing loops with IVs `fx` and `fy` in space, then each PE will maintain different weights and will generate a partial outcome. Thus, **selecting which loops are (completely or in part) executed in the space determines what subset of the data gets processed by each PE.**

The organization of the loops that execute temporally on each PE determines the exact sequence of processing the data and thus, significantly impacts the data reuse and data management of RFs and SPM. For example, for the execution method of Fig. 2.2(b), loops with IVs m, `fy`, and `fx` execute temporally. The loop corresponding to the columns of the filters (`fx`) executes at level 1. This implies that the data corresponding to the loop with IV `fx_L1` is buffered into RFs (L1 memory) of PEs. The execution method allocated data into RFs at maximum



```
for m=1:2
 for oy=1:3
  for ox=1:3
   for fy=1:3
    for fx=1:3
      O[m][oy][ox]+=
       I[oy+fy-1][ox+fx-1]
       × W[m][fy][fx];
```

```
% access DRAM once (no L3 loops)
dma() % prefetch data in 256B SPM
for m_L2=1:2
 for fy_L2=1:3
  access_SPM_and_comm_NoC();
   for fx_L1=1:3
   for oy_S=1:3
     for ox_S=1:3
       O[m_L2][oy_S][ox_S]+=
         W[m_L2][fy_L2][fx_L1]×
         I[oy_S+fy_L2-1]
           [ox_S+fx_L2-1];
```

Figure 2.2: (a) Convolution of a 5×5 input feature map with 3×3 weights of two filters. (b) An execution method, which executes a 3×3 output feature map on different PEs of the dataflow accelerator.

capacity (3 elements for I and W, 1 element for O i.e., 7 elements or 14 bytes in 16-byte RFs). Thus, each PE executes 3 times (`fx_L1=1:3`) and processes data from the registers. Now, when the remaining loops execute (a total of 6 iterations of L2 loops with IVs `fy_L2` and `m_L2`), new data is accessed from the SPM (L2 memory) and communicated to PEs via NoC. Since the operand `O` is invariant of `fy_L2`, it gets used thrice from RFs of PEs. Thus, both the ifmaps and weights are loaded from SPM 2×3 = 6 times, while ofmap is reused and written to SPM just twice. Now, after interchanging both the L2 loops, the loop with IV `m_L2` becomes innermost. Hence, with `I` being invariant of `m_L2`, ifmap gets reused.

Note that the execution method of Fig. 2.2(b) shows just one way of spatiotemporal execution and many such variations are possible. However, when execution methods are not explicitly modeled (e.g., in the code of Fig. 2.2a), a specific execution sequence is implicit, and it is impracticable to capture and explore the variations in both the spatial execution and data reuse in memory hierarchy.

2.3 Related Work

Dataflow accelerator architectures: Several dataflow accelerator designs for domain-specific acceleration are proposed recently [4, 29, 30, 114]. Google TPU [30] is a systolic-array accelerator for computer vision and language models. Chen et al. [4] proposed Eyeriss architecture that efficiently executes their novel row stationary dataflow mechanism. Cong et al. [125] used a polyhedral based analysis to generate high-performance systolic array architectures for executing loops on FPGAs. HyPar architecture [126] is an array of hybrid memory cube based accelerators for training *deep neural networks (DNNs)*. Lu et al. [114] considered various dataflow mechanisms to execute convolutions and proposed a dataflow accelerator architecture which can execute either of them.

Compilation techniques for loop optimizations: Although techniques of loop tiling and permutation are well studied over the past few decades, they have been either agnostic to hardware features or primarily researched for off-the-shelf processors [127–130]. Moreover, their cost functions are often limited to the memory subsystem of a processor with an objective to optimize the data allocation in the on-chip memory. However, minimizing DRAM accesses is not sufficient to achieve efficient mappings for dataflow accelerators, since other factors like efficient interleaving of computation with communication, efficient reuse of different operands, and higher resource utilization significantly contribute to the net acceleration. In fact, due to diverse architectural features (pipelined PEs, data buffering options, NoC configurations, memory sizes, and memory configurations), complete modeling and optimization for the entire accelerator system are required. Furthermore, these loop optimization techniques may require drastic pruning for exploring the optimal execution method. For example, loop optimization techniques of [128, 131] suffer from the vast space of loop-orderings, since up to 7!=5040 orderings (per tiling configuration) need to be explored for a 7-deep loop-nest. Besides, an alternative to MIMD-style dataflow execution is software pipelining the loops [132]; loop operations of the same or consecutive iterations concurrently execute on PEs of a coarse-grain reconfigurable array (CGRA) accelerator [97, 133–141]. Such an approach is beneficial to accelerating non-vectorizable loops through instruction-level parallelism. However, these mapping techniques were primarily evaluated for kernels with relatively small computational or memory requirements and on considerably smaller PE arrays (16–64 PEs) e.g., in [97, 142–144]. In contrast, high-performance demanding loop-kernels of GEMM, convolutions, logistic regressions, etc. exhibit abundant data- and thread-level parallelism and can be efficiently accelerated on the designs with larger arrays of PEs (e.g., from 256 to 65,536) featuring larger RFs.

Explicit modeling of all execution methods: For compute- and memory-intensive loop-nests, numerous execution methods exist for configuring tiling and ordering of the loops for their spatial execution and for accessing the data from RFs, scratchpad memory or global buffers, and DRAM. In the absence of a system to explicitly and succinctly capture the vast space of execution methods, the programmers and architects considered specific execution methods. For example, [53, 145] tiled loop-nest once (transformed a 7-deep nest to 14-deep), which specified how accelerator accesses DRAM and buffer the data in the SPM. However, they lacked tiling the loops further to explicitly model the spatial execution and RF accesses. This scenario is similar to the code of Fig. 2.2a, which implicitly assumed a sequence and offered no insight about variations in the data loaded from the scratchpad to the register files and how differently PEs can process data. Similarly, [43, 44] executed loops corresponding to ofmaps in the space, missing out exploring many execution methods. Techniques used by [43, 44] maximized the psum reuse, and [4] maximized weight reuse in the register file and psum reuse in the scratchpad and did not explore other execution methods. Likewise, techniques like [54, 145–147] considered a batch size of N=1 images, missing the opportunities for weight reuse. Thus, prior techniques organized the loops in certain ways and *without explicit modeling of the complete spatiotemporal execution*, they lacked information about different execution methods. It is demonstrated later that without a systematic approach (like the representation proposed in this chapter) that captures vast space of the execution methods, information available about the entire space is not comprehensive. Hence, the programmer/optimizer ends up with an inferior solution.

Analytical modeling of dataflow execution: For mapping perfectly nested loops onto dataflow accelerators and for design exploration, it is necessary to determine the effectiveness of an execution method statically. Since dataflow accelerators exhibit simple design and are explicitly managed, few works recently developed analytical models to either estimate energy consumption or execution time [31, 46, 53] for DNNs. MAESTRO [148] provides an analytical model for DNNs and estimates the efficiency of an execution method. However, the user needs to understand the proposed directives and write them in MAESTRO DSL, where chosen parameters for the directives can significantly impact the spatiotemporal execution. Yang et al. [31] proposed an energy model [149] for dataflow execution of DNNs and LSTMs, but it lacks estimation of the execution time. Likewise, [53, 150] proposed performance models with an assumption that PEs are always engaged in performing operations and never stall. Thus, prior analytical models either lack estimation of energy consumption or execution time, or do not accurately model data reuse or miss penalty, or lack auto-optimizer. Moreover, these models are specific to DNNs (i.e., may not be capable of analyzing nested loops

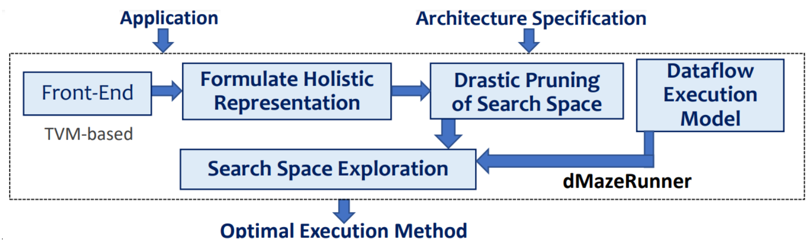


Figure 2.3: Overview of dMazeRunner framework for an efficient application mapping onto dataflow accelerators.

from various applications). Furthermore, they require the user to specify DNN layer parameters as inputs and do not provide integrated support for common ML/AI application libraries like TensorFlow, MXNet, or PyTorch. Instead, this work develops a more comprehensive analytical modeling, which is discussed later in chapter 4.

2.4 dMazeRunner Framework and Implementation

To efficiently map perfectly nested loops onto programmable dataflow accelerators, dMazeRunner framework is proposed as a comprehensive solution. Fig. 2.3 shows dMazeRunner framework. Its front-end parses the application and extracts target loop-nest. After analyzing the loop-nest, dMazeRunner formulates the holistic representation as described above, which features explicitly tiled loops for spatial execution as well as for accessing data from RFs, SPM, and DRAM. Various configurations of this representation capture the vast space of execution methods. It also estimates execution costs for processing these loops spatiotemporally on the accelerator (chapter 4). With drastic pruning of the execution methods, it can find (near)optimal execution method quickly, e.g., in seconds [86].

Framework implementation: dMazeRunner framework features analysis, transformations, and optimizations for dataflow execution of loops. Front-end of the framework leverages TVM environment [124] to support various applications and multiple ML libraries such as MXNet, keras, and TensorFlow. Using the TVM environment, dMazeRunner achieved execution method can be transformed into LLVM IR [151] for code generation. Moreover, for a rapid design space exploration on modern multi-core platforms, the framework implementation leverages the hardware features like caching of the commonly invoked analysis routines and multi-threading. The framework has been made available at `https://github.com/MPSLab-ASU/dMazeRunner`.

2.5 Holistic Representation to Capture Vast Space of Execution Methods

Execution on the accelerator architectures takes place by means of executing the loop iterations onto the PE array both spatially and temporally. To determine spatial execution onto processing elements and the data accessed from register files or local memories, scratchpad or global buffers, and DRAM, each loop of the loop-nest is explicitly tiled at these four levels. Fig. 2.4(b) shows the proposed representation, which is obtained after transforming the algorithm of Fig. 2.4(a). Thus, the proposed formulation transforms a 7-deep nested loop into a 28-deep nested loop. In this explicitly tiled form, the configurable parameters are—loop iteration counts (tiling factors like $N\_SPATIAL$, N_RF) and ordering of the loops at any of these four levels.

This representation can be configured to represent various execution methods. For example, in order to achieve the method of Fig. 2.2(b), the seven innermost loops that correspond to spatial execution are configured. The innermost two loops (`ox` and `oy`) that have tiling factors greater than 1 (`Ox_SPATIAL = 3`, `Oy_SPATIAL`=3) determine how PEs are grouped in a 2D array. For example, Fig. 2.5 shows tiling for spatial execution of three loops. Here, unrolling the third tiled loop (`M_SPATIAL`=2) for spatial execution results in two groups of 3×3 PEs. In fact, if the architecture features interconnections for 3D array (e.g., cubic or vertically-stacked 2D array), then

```
for n_L3 = 1:N_DRAM
 for m_L3 = 1:M_DRAM
  for c_L3 = 1:C_DRAM
   for oy_L3 = 1:Oy_DRAM
    for ox_L3 = 1:Ox_DRAM
     for fy_L3 = 1:Fy_DRAM
      for fx_L3 = 1:Fx_DRAM
      {
       dma( );
        for n_L2 = 1:N_SPM
         for m_L2 = 1:M_SPM
          for c_L2 = 1:C_SPM
           for oy_L2 = 1:Oy_SPM
            for ox_L2 = 1:Ox_SPM
             for fy_L2 = 1:Fy_SPM
              for fx_L2 = 1:Fx_SPM
              {
               access_SPM_and_comm_NoC();
                for n_L1 = 1:N_RF
                 for m_L1 = 1:M_RF
                  for c_L1 = 1:C_RF
                   for oy_L1 = 1:Oy_RF
                    for ox_L1 = 1:Ox_RF
                     for fy_L1 = 1:Fy_RF
                      for fx_L1 = 1:Fx_RF
                      {
                       for n_S = 1:N_SPATIAL
                        for m_S = 1:M_SPATIAL
                         for c_S = 1:C_SPATIAL
                          for oy_S = 1:Oy_SPATIAL
                           for ox_S = 1:Ox_SPATIAL
                            for fy_S = 1:Fy_SPATIAL
                             for fx_S = 1:Fx_SPATIAL
                                 O[][][][] +=
                                  I[][][][] ×
                                  W[][][][];
                      }
              }
      }
```

(b)

```
for n=1:N              % batch size
 for m=1:M             % filters
  for c=1:C            % channels
   for oy=1:Oy         % ofmap rows
    for ox=1:Ox        % ofmap cols
     for fy=1:Fy       % filter height
      for fx=1:Fx      % filter width
        O[n][m][oy][ox] +=
         I[n][c][oy+fy-1][ox+fx-1]
          × W[m][c][fy][fx];
```

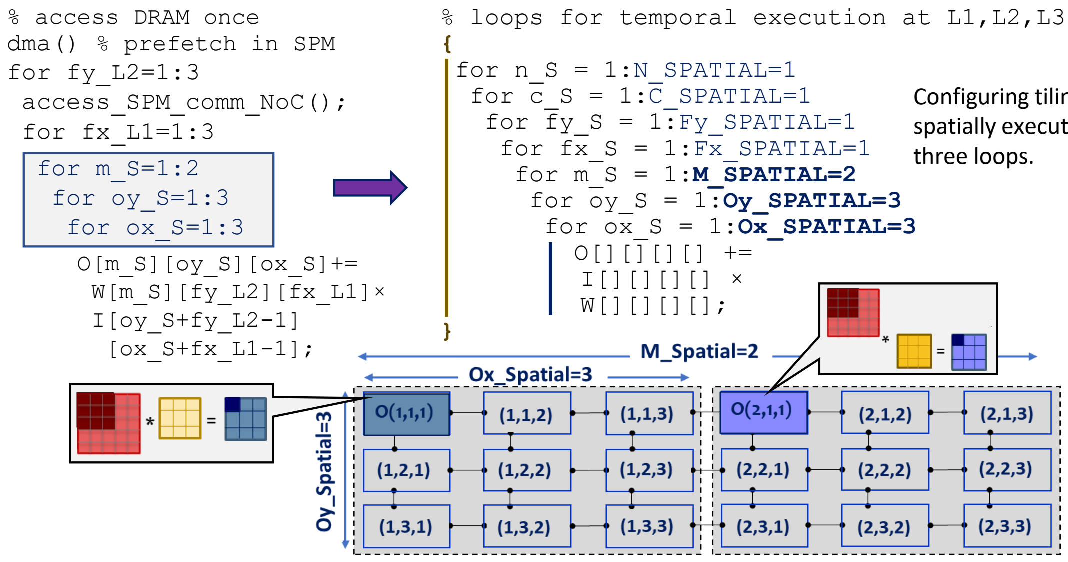
(a)

- Iteration counts (tiling factors) and orderings for L3 loops determine data communicated to (reused in) SPM.
- Iterations of L2 loops affect SPM accesses and the cost of data communication to RF via NOC.
- Iterations of L1 loops indicate data accessed by a PE from RF.
- Inner loops are unrolled in space; Spatial execution determines the subset of the data processed by each PE.

Figure 2.4: Explicitly tiled representation that comprehensively models the vast space of methods for spatiotemporal execution.

such tiling for spatially executing more than two loops can be translated into mapping onto a 3D array.

The seven loops at memory/buffer levels L1, L2, and L3 execute temporally on each PE and are configured to specify the accesses to RF, SPM, and DRAM. Here, tiling factors (e.g., `N_SPM`=2) impact the size of the data accessed from L1/L2/L3 memory



```
% access DRAM once
dma() % prefetch in SPM
for fy_L2=1:3
 access_SPM_comm_NoC();
 for fx_L1=1:3
  for m_S=1:2
   for oy_S=1:3
    for ox_S=1:3
     O[m_S][oy_S][ox_S]+=
      W[m_S][fy_L2][fx_L1]×
      I[oy_S+fy_L2-1]
       [ox_S+fx_L1-1];
```

```
% loops for temporal execution at L1,L2,L3
{
 for n_S = 1:N_SPATIAL=1
  for c_S = 1:C_SPATIAL=1
   for fy_S = 1:Fy_SPATIAL=1
    for fx_S = 1:Fx_SPATIAL=1
     for m_S = 1:M_SPATIAL=2
      for oy_S = 1:Oy_SPATIAL=3
       for ox_S = 1:Ox_SPATIAL=3
        O[][][][] +=
         I[][][][] ×
         W[][][][];
}
```

Figure 2.5: Configuring the representation of Fig. 2.4(b) for spatial execution of three loops, which results in two PE-groups in the accelerator.

(section 4.1.1 provides the exact calculation), and ordering of the loop determines the schedule of data movement i.e., data reuse/eviction. For example, in Fig. 2.4(b), the iteration counts of loops with IVs m_L2 and fy_L2 determine accesses to SPM, and for executing innermost loop (with IV fy_L2), operand "O" (psum) is reused. Thus, this explicitly tiled transformation of Fig. 2.4(a) can be configured to represent the various spatiotemporal organization of the loops (tiling/ordering/unrolling), which corresponds to different execution methods to program the dataflow accelerators. Thus, in the comprehensive mapping space formulation, since each loop of the input nest is *explicitly* modeled for spatial execution and for accessing data from L1/L2/L3 etc. memory, it captures the vast space of execution methods. Hence, the compiler can determine and explore the global space of execution methods and achieve the optimal execution method.

Note that the convolution layers from deep neural network (DNN) models are used here to explain the background and examples and for demonstrating the search space and design space exploration capabilities. This is because, convolution layers in deep learning models feature a 7-deep loop-nest (deeper than matrix multiplication or other applications), exhibiting various ways of data reuse and spatial execution. They are widely used in deep learning and media processing applications [1, 8, 9, 18, 30, 111, 152]. However, the proposed approach is more general and can optimize the execution of any perfectly nested loop (featuring direct memory accesses and statically known loop bounds) on a dataflow accelerator.

2.6 Drastically Pruning the Vast Search Space

2.6.1 Determining Valid Tiling Options

Statically pruning the search space becomes essential when it is vast. A common approach used in pruning of the accelerator design space is restricting values of the design hyperparameters based on the bounds of specified constraints. For instance, the buffer size can be restricted to less than 100kB if the total power budget is 0.1W. Such pruning usually requires no or a few evaluations of cost functions, while limiting the search of each hyperparameter to only meaningful values. In addition, infeasible solutions can be pruned statically by checking, in formulating the search space, against the validity of a solution w.r.t. constraints imposed by the accelerator design and execution. This is described herein by formulating the mapping space of execution methods while considering the validity of tiling options.

After multi-level tiling of a loop, trip-counts (TCs) of the tiled loops can be of any integer value. For example, consider a loop that iterates N=8 times. After tiling it into four levels, the trip-counts of the tiled loops are $N\_SPATIAL$, N_RF, N_SPM, and N_DRAM, which are the optimization parameters.

When off-the-shelf optimizers (constraint-solvers for non-linear programming that use simulated-annealing, newton's method, etc.) are used [54, 153], in each step, they randomly select the parameter values from all possible combinations (8^4). For large-scale optimization problems, since the valid methods are very few (e.g., 20 out of 4096 in this example), an optimizer's majority of the search time is often spent on discarding invalid solutions. However, a constraint-driven pruning of the space can be employed before beginning the exploration and analytical evaluation of execution methods, by considering only valid tiling options (e.g., 20 instead of 8^4). It ensures that for tiling of a loop into four loops, the total iterations executed by the tiled loops match the functionality of the loop-nest, i.e.,

$$cons : N\_SPATIAL \cdot N\_RF \cdot N\_SPM \cdot N\_DRAM = N$$

In general, for any loop index-variable `iv`,

$$TC[base][iv] = TC[SPATIAL][iv] \cdot TC[RF][iv] \cdot TC[SPM][iv] \cdot TC[DRAM][iv]$$

For pruning the mapping space, it can be ensured that the pruning is subjected to constraints from the architecture resources (PEs, RF, and SPM). For example, the data to be allocated by an execution method must fit into RF of a PE and in a multi-bank SPM/buffer, i.e.,

$$cons : \sum_{op=1}^{total\_Operands} data\_alloc[RF][op] \leq RF\_size$$
$$cons : \sum_{op=1}^{total\_Operands} data\_alloc[SPM][op] \leq SPM\_size$$
$$cons : \prod_{i=1}^{total\_IVs} TC[SPATIAL][IVi] \leq Total\_PEs$$

For example, when RF tiling factors $< N\_RF, M\_RF, C\_RF, Oy\_RF, Ox\_RF, Fy\_RF, Fx\_RF >$ are selected as $< 1, 1, 1, 1, 1, 1, 3 >$, allocated registers for weights are `data_alloc[RF][W]` = M_RF$\times$ C_RF $\times$ Fy_RF$\times$ Fx_RF = 3. Total allocated registers are 3 + 3 + 1 = 7 (for input activations or I, weights or W, and output activations or O), and this is a valid method for an 8-element RF (example of Fig. 2.2, no double-buffering is assumed

Table 2.1: Analyzing size of the mapping space for deep learning accelerators.

Model	**Layer**	**Tile Sizings**	**Tile Sizings with Valid Factors**	**Valid Tilings w.r.t. Hardware**	**Orderings at a Memory Level**	**Orderings with Unique/Max Data Reuse**	**Full Map. Space**	**Factorization-Constrained Mapping Space**	**Factorization-Constrained Reuse-Aware Map. Space**
		A	B	C	D	E	F: A*D^2	G: B*D^2	H: B*E^2
ResNet18	CONV_2_1a	$O(10^{25})$	$O(10^{13})$	$O(10^{7})$	$O(10^{4})$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
MobileNetV2	features.2.conv.0	$O(10^{22})$	$O(10^{12})$	$O(10^{6})$	$O(10^{4})$	15/3	$O(10^{30})$	$O(10^{19})$	$O(10^{13})$
EfficientNetB0	blks.2.expand	$O(10^{22})$	$O(10^{12})$	$O(10^{6})$	$O(10^{4})$	15/3	$O(10^{29})$	$O(10^{20})$	$O(10^{13})$
VGG-16	CONV_1_2	$O(10^{28})$	$O(10^{14})$	$O(10^{7})$	$O(10^{4})$	15/3	$O(10^{36})$	$O(10^{21})$	$O(10^{15})$
ResNet50	CONV_2_1b	$O(10^{25})$	$O(10^{13})$	$O(10^{7})$	$O(10^{4})$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
Vision Transformer	patchembeddings. CONV2D	$O(10^{25})$	$O(10^{13})$	$O(10^{6})$	$O(10^{4})$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
FasterRCNN-MobileNetV3	features.12. conv2.excite	$O(10^{26})$	$O(10^{13})$	$O(10^{6})$	$O(10^{4})$	15/3	$O(10^{33})$	$O(10^{20})$	$O(10^{14})$
YOLOv5	features.1.conv	$O(10^{27})$	$O(10^{14})$	$O(10^{7})$	$O(10^{4})$	15/3	$O(10^{34})$	$O(10^{21})$	$O(10^{15})$
Transformer	decoder. output_projection	$O(10^{27})$	$O(10^{9})$	$O(10^{4})$	$O(10^{1})$	3/3	$O(10^{28})$	$O(10^{10})$	$O(10^{10})$
BERT	encoder.layer.0. output.dense	$O(10^{26})$	$O(10^{9})$	$O(10^{5})$	$O(10^{1})$	3/3	$O(10^{27})$	$O(10^{11})$	$O(10^{10})$
Wav2Vec2	encoder.layers.0. intermediate.dense	$O(10^{28})$	$O(10^{12})$	$O(10^{6})$	$O(10^{1})$	3/3	$O(10^{29})$	$O(10^{13})$	$O(10^{12})$

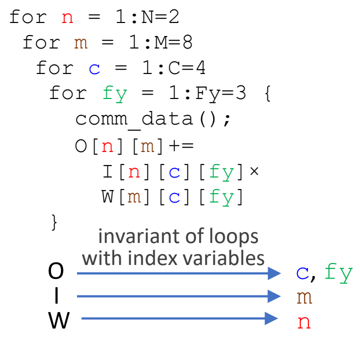
here). However, a solution with RF-level loop tiling factors $< 2, 1, 1, 1, 1, 1, 3 >$ is invalid and not considered for the exploration, since it allocates $6 + 3 + 2 = 11$ elements. Thus, the constraints discard invalid tiling options. Table 2.1 shows that such factorization based and architecture-aware pruning can discard invalid tilings in a significant way. They prune the search space of tiling configurations by a square root or even a cube root, e.g., from $O(10^{22})$–$O(10^{28})$ configurations to a much smaller range of $O(10^{9})$–$O(10^{14})$ configurations. Thus, overall pruning achieved is a factor of $O(10^{9})$ – $O(10^{18})$ for different loop-kernels and layers of different deep learning models (columns A $\rightarrow$ B $\rightarrow$ C and columns F $\rightarrow$ G).

2.6.2 Determining Loop Orderings for Unique Data Movement Costs

There can be many solutions in vast design space with the same costs as others or much higher than the most efficient solution (globally suboptimal) within a bounded subspace. For domain-specific exploration, pruning such feasible yet ineffectual solutions statically can be done by approximating the costs of different solutions statically (e.g., without an actual full evaluation of cost functions) and imposing hyperparameters for exploration that limit the search space to effectual solutions. This is described herein by taking an example of pruning the loop ordering search space.

While tiling factors for L1, L2, and L3 loops determine the size of the data accessed from RF, SPM, and DRAM, the orderings of these loops determine the data reuse and scheduling of the data movement. In a loop nest, data operands (tensors) are often invariant of specific loops and can be reused [128]. Therefore, for a given a loop-nest, it is possible to create a list of all those *loop-orderings (schedules)* that feature unique reuse of operands, and the optimizer needs to target just those orderings. For example, it is demonstrated herein that out of 7!=5040 orderings to organize 7-deep loop-nest of convolution, loop-orderings featuring unique reuse factors are just up to 15. Such reduction stems from the fact that for execution of tiled L2/L3 loops, memory management ensures the availability of the data blocks prior to the execution, and reuse factors of operands (data blocks) get limited, as compared to numerous hit/miss occurrences possible (at cache-line granularity) in a cache-based memory hierarchy.

Fig. 2.6 depicts a 4-deep loop-nest along with information about each operand being invariant of certain loops. To explain the impact of orderings, in this example, it is assumed that the current memory level (e.g., RF) can accommodate 3 data elements. Thus, during each loop iteration (total 192), the data corresponding to each operand



```
for n = 1:N=2
 for m = 1:M=8
  for c = 1:C=4
   for fy = 1:Fy=3 {
     comm_data();
     O[n][m]+=
       I[n][c][fy]×
       W[m][c][fy]
   }
```

Unique Reuse Factors for Data Operands

Loop Order	I	W	O
{…, n}	–	N = 2	–
{…, m}	M = 8	–	–
{…, m, c}	–	–	C = 4
{…, m, fy}	–	–	Fy = 3
{…, fy, c}	–	–	C×Fy = 4×3

Figure 2.6: Determining all orderings of loops that feature unique data reuse factors. Achieved orderings are five among a total of 4! = 24 orderings. Dash symbol indicates a use factor of 1 for an operand (i.e., no reuse).

can be accessed from lower memory (e.g., SPM) and brought to the current memory level. In other words, for a given ordering, for each operand, the function *comm_data*() may (not) execute in every loop iteration. Fig. 2.6 also tabulates different orderings that feature unique reuse factors. Loop IVs are in lower-case, and *trip-counts (TCs)* are in upper-case. For each schedule, a listing of loop IVs from the right- to left-hand-side indicate the order of innermost to outermost loops. For example, the first ordering indicates that loop with IV n is the innermost, and "…" indicates that the ordering of outer 3 loops does not matter for this schedule. So, selecting any one ordering among 3! combinations yields the same reuse.

To generate the schedules (algorithm 2), dMazeRunner iterates over each operand and constructs the loop-orderings for which the operand is invariant of inner loops. For example, `W` is invariant of `n`, and the first loop-ordering is the only schedule where `W` is reused for `N`=2 iterations. Thus, out of 192 iterations, `W` is accessed from memory only $192/2 = 96$ times. However, since `I` and `O` are indexed through `n`, they are communicated from lower memory during all 192 iterations (for a given ordering, algorithm 1 determines such reuse factors for operands). Similarly, `I` gets reused only in the second ordering. Now, `O` is invariant of two IVs `c` and `fy` (total_independent_IVs=2). So, more than one orderings feature unique reuse of `O` (generated by lines 5–15 of algo. 2). Two possible orderings (3^{rd} and 4^{th}) are where `O` is reused only in the innermost loop with IV as either `c` or `fy`. Similarly, `O` is reused in both the inner loops when IVs for inner loops are permutations of `c` and `fy`. Here, both the permutations ('`c`', '`fy`') or ('`fy`', '`c`') yield the same reuse factors (1 for `I` and `W` and 12 for `O`). So, any one permutation can be considered (line 16 in Algo. 2 prunes another), which is 5^{th} ordering. Thus, dMazeRunner prunes $4! = 24$ orderings to just 5. Similarly, for convolution of Fig. 2.4(a), dMazeRunner prunes 7!=5040 orderings to 15 orderings that feature unique reuse, which are listed in Table 2.2. For given tiling factors of

Algorithm 1: *Determine_Data_Reuse*(Input *loop_ordering*, Input *level*, Input *operand_list*, Output *reuse_vector*)

```
1  foreach operand op in operand_list do
2      operand_reuse_factor = 1;
3      list_op_dependent_IVs = get_op_dependencies(op);
4      foreach iv in reversed(list(loop_ordering)) do
5          tc = get_TripCounts(iv, level);
6          if (tc == 1) then
7              continue;
8          else if (iv is not in list_op_dependent_IVs) then
9              operand_reuse_factor *= tc;
10         else
11             break;
12     reuse_vector[op] = operand_reuse_factor
13 return reuse_vector
```

Algorithm 2: *Generate_Loop_Orderings*(Input *operand_list*, Output *pruned_orderings*)

```
1  foreach operand op in operand_list do
2      list_op_independent_IV = get_op_dependencies(op);
3      total_independent_IVs = len(list_op_independent_IV);
4      list_orderings = null; iter = 1;
5      while $iter \leq total\_independent\_IVs$ do
6          list_IVs = null; temp_list =
             get_combinations_IVs(list_op_independent_IV, iter);
7          foreach item in temp_list) do
8              list_permutations = get_all_permutations(item);
9              list_IVs.append(list_permutations);
10         list_IVs.remove_duplicate_items();
11         foreach item in list_IVs) do
12             temp_ordering = prepend_dependent_IVs(item,
                 list_op_dependent_IV);
13             order = prepend_missing_IVs_in_random_order (temp_ordering,
                 list_op_independent_IVs);
14             list_orderings.append(order);
15         iter++;
16     pruned_orderings= prune_orderings_same_reuse(list_orderings);
17 return pruned_orderings
```

an execution method, collective orderings (of L2 and L3 loops) to reuse the data while accessing SPM and DRAM are up to 15×15 instead of 5040×5040. Note that the list of orderings (e.g., ones in Table 2.2) are determined statically once, before

Table 2.2: Unique data reuse factors for accessing lower memory

Schedule	**Ifmap**	**Weights**	**Ofmap**
`{..., m}`	M	1	1
`{..., m, ox}`	1	Ox	1
`{..., m, oy}`	1	Oy	1
`{..., m, n}`	1	N	1
`{..., m, oy, ox}`	1	Oy × Ox	1
`{..., m, n, ox}`	1	N × Ox	1
`{..., m, n, oy}`	1	N × Oy	1
`{..., n, oy, ox}`	1	N × Oy × Ox	1
`{..., m, fx}`	1	1	Fx
`{..., m, fy}`	1	1	Fy
`{..., m, c}`	1	1	C
`{..., m, fy, fx}`	1	1	Fy × Fx
`{..., m, c, fx}`	1	1	C × Fx
`{..., m, c, fy}`	1	1	C × Fy
`{..., c, fy, fx}`	1	1	C × Fy × Fx

the exploration and evaluation of execution methods begin. Furthermore, during exploration of execution methods, for a given set of tiling factors, it is possible that one or more loops iterate(s) just once (e.g., M_SPM=1). In such a scenario, among these 15 orderings, several orderings feature the same reuse factors. In other words, unique reuse factors reduce from 15 orderings. Thus, during exploration, for each set of tiling factors, dMazeRunner dynamically prunes the list of 15 orderings (of Table 2.2) further.

dMazeRunner constructs the list of orderings depending on the operand being invariant of the loops, which is determined by analyzing the indexing expressions of the operand (e.g., `I` is invariant of IV `m`). Therefore, the proposed pruning technique is applicable to direct memory access patterns (including affine accesses), which are commonly found in many applications. Note that in determining orderings, a loop interchange is considered only when it is a legal transformation. The legality can be determined by analyzing distance- and dependence-vectors for the loops [127].

2.6.3 Additional Pruning Heuristics

Depending on the depth and iteration counts of the loops in the application, the exhaustive exploration may take even several hours. One strategy can be to pre-compile the application for common target architectures, where the optimal execution method is explored just once. However, to allow re-compiling applications by users and rapid design space explorations, the optimizer should be able to generate a highly efficient solution promptly. So, dMazeRunner embeds a pruning heuristic that achieves close-to-optimal solutions in second(s) through the following strategies:

OPT 1) Targeting execution methods featuring high resource utilization: dMazeRunner explores only those tiling factors that highly utilize (e.g., 60%) RFs, SPM, and PEs. High utilization improves data reuse and reduces DRAM accesses. Note that very high utilization does not guarantee an optimal solution, as it may not effectively interleave computation and communication cycles.

OPT 2) Discard execution methods requiring several memory accesses of non-contiguous data: Some IVs of loops correspond to a minor dimension of tensors (`fy` and `fx` for W[m][c][fy][fx]). For such IVs, when tiling factors of L3 loops (i.e., `Fy_DRAM`) are greater than 1, it requires many DMA invocations with small burst-sizes. Thus, it results in higher DMA cycles and may introduce the miss penalty for SPM management. So, dMazeRunner discards such execution methods which are susceptible to higher execution time.

OPT 3) Discard execution methods that require inter-PE communication:

Often a *read + write (r+w)* operand (O) is an invariant of few IVs (`c`, `fy`, and `fx`). If loops corresponding to these IVs execute spatially, it requires inter-PE communication (for reduction), which may introduce stall cycles and often costs higher energy. Therefore, to avoid inter-PE communication, dMazeRunner decides not to execute such loops in space. This strategy discards several dataflow mechanisms (e.g., weight-stationary, row-stationary).

OPT 4) Targeting execution methods that maximize the reuse of operands: Although dMaze-Runner determines all loop-orderings featuring unique reuse factors, space can be pruned to few orderings that maximize the data reuse. For example, in Table 2.2, only schedules #8 and #15 maximize the reuse of weights and ofmap respectively. Thus, schedules #2–#7 and #9–#14 are discarded.

OPT 5) Leveraging hardware features of compilation platform: Implementation of dMazeRunner framework integrates - (i) caching of the frequently used analysis routines and commonly referenced hash tables (e.g., loop orderings), and (ii) concurrently exploring various execution methods and evaluating their efficacy with multi-threading. Thus, on modern multi-core processors, the exploration time is significantly reduced. Note that OPT4 and OPT5 do not impact optimality and can be used for an exhaustive search.

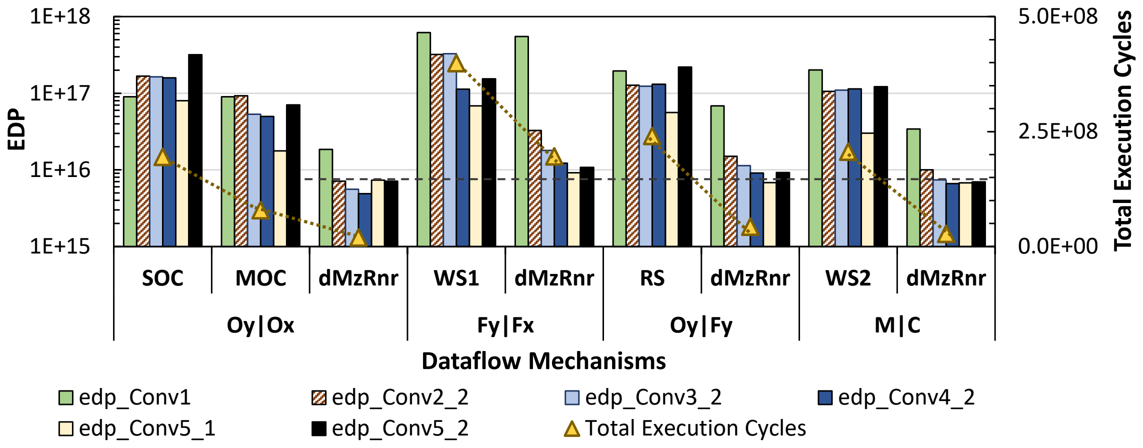
2.7 Experiment Results and Analysis

This section describes the efficiencies of execution methods obtained after exploring the comprehensive design space, that can be formulated through proposed methodology and with dMazeRunner framework. The optimization objective is set to minimize the total summation of the Energy-Delay-Product (EDP) of all the loops that are evaluated.

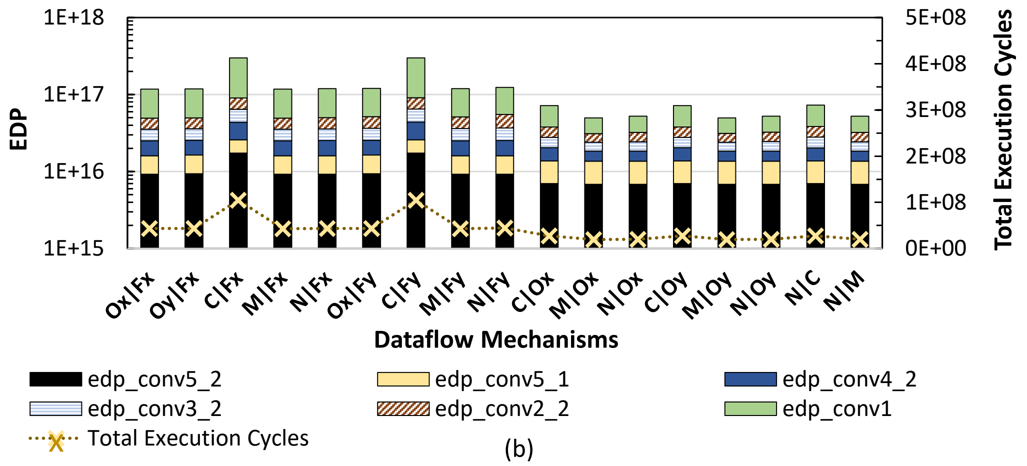
Benchmarks: For evaluating different execution methods (featuring diverse data reuse patterns and various ways of spatial execution), different convolution layers are considered from widely used DNNs - ResNet and ResNext models [1, 111] for ImageNet classification (with batch size of 4 images).

Specification of the target platform: A dataflow accelerator architecture is considered which is similar as recent works including Eyeriss [4] and [31, 114]. The accelerator consists of 16×16 PEs with 16-bit precision. Each PE accesses 512B RF and a 128 kB scratch-pad. Like Eyeriss architecture [45], each pipelined PE consists of a 2-stage multiplier and an adder. The accelerator features 4 single-cycle multi-cast networks [4] to communicate the operands to PEs and 1 such network for reduction. The shared SPM consists of 64 banks (2 kB each) that can be allocated to any data. Data is accessed from DRAM via DMA and managed in SPM with double-buffering [121, 138]. DRAM latency model for data transfers via DMA is same as Cell processors that featured SPMs [154]. Energy costs for accelerator resources were obtained from hardware evaluations by Yang et al. [31] for a 28 nm technology. The analytical cost modeling of the architecture and its validation is described later in chapter 4

Mapping techniques evaluated: To evaluate the effectiveness of the optimal solutions achieved by the comprehensive mapping space, various execution methods are determined for dataflow mechanisms described by previous techniques: (i) For output stationary mechanism (`Oy|Ox`), (i.a) **SOC** [44, 45] in which, entire PE array processes *single output channel*, and (i.b) simultaneous processing of *multiple output channels* **(MOC)** [43, 45] on different PE-groups for *ifmap reuse*. For both SOC and MOC, the data movement schedule iterates over channels for minimizing *psum* accumulation cost, (ii) **WS1** for weight stationary mechanism (`Fy|Fx`) [45, 114], (iii) *RS* [4] for row stationary (`Oy|Fy`) mechanism, which maximizes weight reuse in RF, psum accumulation in RFs/PE-array, psum reuse in SPM, and (iv) coarse weight stationary **(WS2)** for `M|C` mechanism, which is like matrix-multiplication on systolic



(a)



(b)

Figure 2.7: (a) For popular mechanisms, achieved execution methods reduce the total EDP by 9.16× on average. (b) Optimal execution methods for other dataflow mechanisms are also achieved.

arrays [30]. Any other previous technique that optimized EDP through other dataflow mechanisms were not known to the author at the time of these evaluations. However, all mechanisms that show different parallelisms and dataflows are still evaluated to demonstrate effectiveness of achieved execution methods.

2.7.1 Solutions Obtained After Comprehensive Exploration Reduce EDP by 9.16×

Fig. 2.7(a) shows the evaluation of various execution methods for popular dataflow mechanisms. The evaluations depict EDP of each convolution layer on the primary axis and total execution cycles for these six layers on the secondary axis (lower the better). For better visualization, EDP results are plotted on a logarithmic scale.
i) For each dataflow mechanism, the EDP and the total cycles are reduced significantly, when compared to execution methods achieved by prior approaches. For example, for conv5_2, EDP was reduced by 44.47× and execution cycles by 18.72×, as compared to SOC. On average, the summation of the EDP of convolution layers was reduced by 9.16× over other techniques and the total execution cycles by 5.83×.

The primary reason for such a significant scope of improvement is that *prior techniques target certain ways of spatial execution and data reuse, which are often, not very efficient.* For example, when 14×14 PEs executed ofmaps or the output channel(s) spatially, the PE utilization achieved on a 16×16 array was just 76%. Similarly, SOC and MOC maximized psum accumulation in RF, which did not always yield high RF utilization (e.g., 436B utilized for 512B RF). Moreover, *with a fixed optimization strategy to reuse certain data operand(s), no single heuristic efficiently leveraged the maximum data reuse possible.* For example, for convolution layers at beginning of ResNe(x)t (*conv1*), ifmaps are significantly larger and *weight reuse* is desired. In contrast, for later layers (*conv4_2*), weights dominate the data movement, and *ifmap reuse* yields better execution. *The execution methods obtained by prior approaches were not able to adapt to such dynamics of loop characteristics.* Thus, prior techniques neither ensured very high resource utilization, nor efficient reuse of all data operands. So, even if they somehow obtained a reasonable solution, a scope for further reduction in both execution time and energy remained.

With holistic representation, dMazeRunner captured the vast space of execution methods and after drastically pruning the search, it made the brute-force exploration feasible. Consequently, it achieved the optimal execution methods that outperformed prior techniques. For example, for executing ResNet conv5_2 with output stationary dataflow, the obtained execution method allocated 36 elements of image tensor (I), 144 elements of weights (W), and 64 elements for output tensor (O) (244 from 256) in RFs, achieving image and weight reuse through processing the data of 4 images and 16 filters simultaneously. Cycles for performing *Multiply and ACcumulate (MAC)* operations on this data in the registers were 576, which perfectly interleaved with the communication latency of 576 cycles (to communicate $4{\times}1{\times}9{\times}9 = 324$ elements of I and $64{\times}1{\times}3{\times}3 = 576$ elements of W via different interconnects). Data elements of O ($4{\times}64{\times}7{\times}7 = 12{,}544$) were reused in RFs by accumulating partial summation during 8 consequent execution passes. Thus, execution pass latency was 576 cycles, and 8 passes processed the data from the scratchpad memory (4608 cycles). Again, this execution was efficiently interleaved with the latency to communicate the data of I and W tensors from off-chip memory to a double-buffered SPM via DMA transfers (3641 cycles). Considering the stall cycles for writing back the data of output tensor, total execution cycles were 2,459,648, which exceeded an ideal execution time (2,359,296 cycles for executing 462,422,016 MACs on $4{\times}7{\times}7$ PEs) by a mere 4%. Thus, solutions optimized through comprehensive design space formulation can achieve highly optimized and non-intuitive mappings that account for multiple factors attributing to efficient accelerations.

ii) Obtained execution methods achieved various data reuses at different accelerator resources and minimized DRAM accesses for various operands. With a certain optimization strategy, prior heuristics leveraged reuse of specific operands. For example, *SOC* and *MOC* maximized psum reuse at RF and SPM levels, while *RS* maximized weight reuse in RFs and psum reuse in SPM. For executing conv5_1 layer with `Oy|Ox` mechanism, *SOC* allocated 28kB ifmaps, 18kB weights, and 1.5kB psum in SPM, which were accessed from DRAM 512, 512, and 128 times, respectively. This resulted in DRAM access of 14.74MB, 9.44MB, and 0.2MB, respectively. However, the execution method obtained from the comprehensive design space allocated 14.06kB of ifmaps, 18kB weights, and 12.25kB psum in SPM, which were accessed from DRAM 256, 256, and 16 times. So, the obtained method accessed DRAM for 3.68MB ifmaps, 4.7MB weights, and 0.2MB psum. Thus, the obtained solution exhibited a better choice of tiling factors and minimized total DRAM accesses for ifmaps by 4× and 2× for weights. In fact, it maximized *ifmap* and *filter reuse* spatially, *convolution reuse* in RF, and *psum reuse* at RF and SPM levels. Similarly, for executing conv5_2 layer with `M|C` mechanism, it reduced DRAM accesses by 16× for ofmap, as compared to WS2. By significantly reusing all operands, execution methods minimized DRAM accesses, reducing both the energy and execution cycles. Thus, although the acceleration gains during chip execution can differ from estimations, through better data reuse, reduced DRAM accesses, and efficient interleaving of computation with communication, achieved solutions from comprehensively formulated design space can outperform prior heuristics.

iii) With holistic exploration, achieved the optimal solutions which yield similar EDP and execution time for various dataflow mechanisms. Fig. 2.7(a) shows that for various mechanisms, the achieved solutions result in a very similar EDP and execution time (note the dotted line). This is because: (1) for efficient acceleration, often more than two loops are spatially executed (e.g., M and C along with Oy and Ox) and hence, two mechanisms may attain the same solution, and (2) highly efficient solutions share common characteristics like high utilization of resources, maximized reuse of various operands, efficient interleaving of computation with communication (i.e., minimum to no miss penalty). Therefore, for individual mechanisms, the achieved optimal solutions yield similar results. Moreover, Fig. 2.7(b) depicts the EDP and execution cycles for 17 more mechanisms and demonstrates similar results. However, when reduction operations are performed through inter-PE communication, it results in higher cycles in the execution model used [40]. This is because, one single-cycle multi-cast network was used for r+w operands instead of a mesh-style interconnect. This is reflected in a relatively high execution time and high EDP for mechanisms like `Fy|Fx`, `C|Fx`, and `Ox|Fy`. Such difference can be observed at least for conv1 layer that consisted of larger feature maps. Note that none of the prior heuristics pruned the space such drastically that a brute-force algorithm is applied to achieve the optimal solutions. Furthermore, no prior technique achieved the optimal solutions that minimize EDP while using a variety of dataflow mechanisms. Therefore, Fig. 2.7(b) does not feature any evaluations of prior works.

2.7.2 Obtained Solutions Reduce Energy Consumption Up to 30.84% Over Other Optimization Tools

Recently Yang et al. [31] proposed an auto-optimizer [149] to reduce the energy consumption of DNN dataflow execution. This section shows evaluations after executing the various convolution layers of ResNet with [149] and obtained optimized execution methods. Then, they are compared with the solutions achieved by dMazeRunner in order to demonstrate the impact of holistic exploration.

Fig. 2.8 shows the energy consumption of optimized methods obtained by [149] and that of dMazeRunner (holistic mapping space formulation). Here, the energy of a convolution layer is obtained from the best outcome among all execution methods explored. Execution methods achieved by dMazeRunner outperformed [149] by reducing the energy up to 30.84% and by 15.55% on average. Even for individual

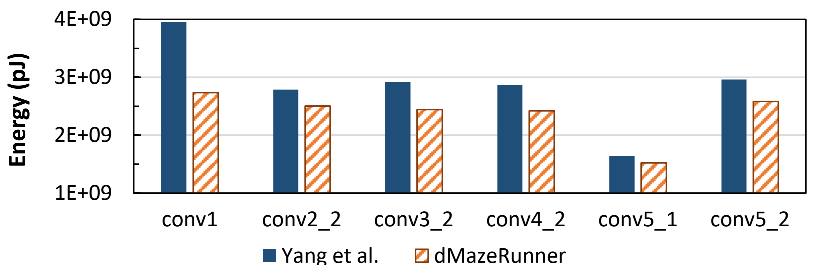


Figure 2.8: Mappings obtained from comprehensive mapping space reduce energy consumption up to 30.84% as compared to the auto-optimizer of Yang et al (2018).

dataflow mechanisms (like output- or row-stationary [4, 31]), obtained execution methods from comprehensive mapping space reduced the energy significantly over [149]. For example, they reduced the energy of executing conv4_2 with `N|Ox` by 36% and conv5_2 with `Oy|Ox` by 14%.

Note that heuristically exploring a small fraction of all execution methods may not yield efficient EDP or reasonable execution time. In fact, heuristically obtained solution may fail to efficiently interleave the computation with communication latency (high miss penalty). Since [31, 149] lacked performance model, this work did not compare the EDP or execution time of obtained methods with it.

Table 2.3: DNN layer-specific and overall best memory sizes for a 256-PE accelerator.

ResNet Layer	**Layer-Specific Best Design**	**EDP**	**Overall Top #1 Design**	**EDP (normalized)**	**Overall Top #2 Design**	**EDP (normalized)**
Conv1	<256,512k>	1.59E+16	<256, 512k>	1x	<256, 1024k>	1.16x
Conv2_2	<256,512k>	4.52E+15		1x		1.03x
Conv3_2	<256,1024k>	3.56E+15		1.10x		1x
Conv4_2	<256,1024k>	3.57E+15		1.11x		1x
Conv5_1	<256,1024k>	2.83E+15		1.08x		1x
Conv5_2	<256,128k>	6.14E+15		1.05x		1.03x
Total (6 layers)	<256,512k>	3.78E+16		1x		1.04x

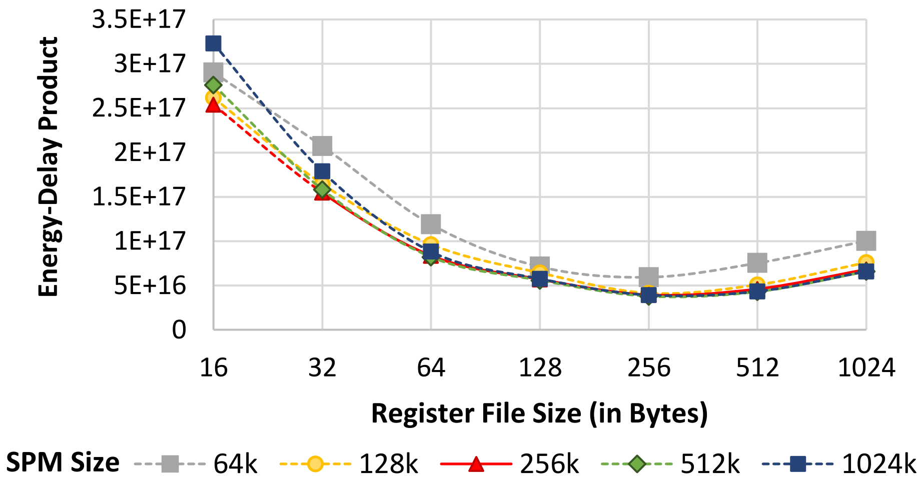


Figure 2.9: Exploration of memory sizes for a 256-PE accelerator. Designs featuring 256B RFs and 512 kB SPMs yield lower EDP.

2.7.3 Exploring Memory Configurations

As holistic design space exploration enables efficient execution methods, dMazeRunner can be leveraged to invoke a rapid DSE for landing upon better architectural design solutions. Table 2.3 lists the results of a DSE experiment which optimizes the on-chip memory sizes for the targeted 256-PE accelerator. The second-left column lists the best memory configuration for each layer and the third-left column lists corresponding EDP. The columns on the right-hand side show the two best designs that achieved the best EDP (normalized) for some layers and in total, a lower EDP as compared to other configurations. Both the designs #1 and #2 (in fact, the top four designs) featured 256B RFs per PE. Fig. 2.9 depicts the EDP (a total of all the six convolution kernels) for the variations in the RF sizes (primary horizontal axis) and SPM sizes (series in the legend).

Fig. 2.9 shows that EDP is notably lower when the RF size is 128B or larger. This is because, the convolution kernels exhibit the significant reuse of different operands, which can be better sustained with larger RFs, avoiding costly accesses to SPM and

DRAM. However, increasing the RF size beyond 512B increases the EDP again, since it is hard to efficiently utilize RF (i.e., finding a schedule that balances communication latency with computation from the RF) while the energy cost to access RF increases. The RF size of 256B demonstrates a balance in the trade-off and yield significantly lower EDPs. Similarly, an SPM size of 512kB demonstrates lower EPDs. If an SPM size is relatively small (e.g., 64kB or smaller) then, after reusing the data at the RF level, the accesses mostly go to DRAM since there is little-to-no reuse at SPM level. On the other hand, accessing a larger SPM significantly costs more energy and can yield a large increase in the EDP, if RF size is smaller (e.g., consider a 512kB or 1024kB SPM and a 16B RF). Thus, dMazeRunner can be leveraged for exploring the optimized designs.

2.8 Conclusions

This chapter has presented several aspects of enabling efficient accelerator designs, especially the effectiveness of formulating a comprehensive design space for achieving efficient executions and obtaining better hardware/software codesigns. The design space needs to be comprehensive so that a vast range of hardware and/or software configurations can be specified and explored and, consequently, highly efficient solutions can be obtained. Instead of restricting the execution of applications like deep learning models to one or a few specific dataflows, the proposed approach introduces formulation of a comprehensive mapping space by covering spatial and temporal execution at all levels of hierarchy within the architecture. The evaluations have shown that exploring such a comprehensive design space helps achieve the dataflow that can adapt to architecture configuration and functionality and shapes of tensors of the loops, thereby improving the efficiency notably (e.g., reduction in the energy-delay product by an order of magnitude). This is primarily because, as compared to a fixed mapping strategy, for an ML operator, the explored solutions can exhibit higher spatial parallelism, effective interleaving of the communication latency with computations, and high data reuse of various tensors exploited spatially and temporally, and thereby, achieve low latency or energy consumption. Moreover, considering a broad architectural design space and including them in mapping exploration for spatiotemporal execution can help achieve a highly efficient and constraints-meeting solution. Further, the ability to formulate mapping space comprehensively for a range of architectures and being able to explore the mappings quickly (e.g., in few seconds) could help achieve adaptive execution methods a.k.a. dataflows, and thereby, notably higher efficiency of hardware-software codesigns in the design exploration process.

Chapter 3

AGILE AND EXPLAINABLE DESIGN SPACE EXPLORATION

Effective design space exploration (DSE) is paramount for hardware/software codesigns of domain-specific architectures that must meet strict execution constraints. For their vast search space, existing DSE techniques can require excessive trials to obtain a valid and efficient solution because they rely on black-box explorations that do not reason about design inefficiencies. This marks the need for developing explainable design methodology.

This chapter proposes Explainable-DSE – a framework for the DSE of accelerator codesigns using bottleneck analysis. By leveraging information about execution costs from bottleneck models, the proposed DSE is able to identify bottlenecks and reason about design inefficiencies, thereby making bottleneck-mitigating acquisitions in further explorations. This chapter describes the construction of bottleneck models for DNN accelerators. Then, it also proposes an API for expressing domain-specific bottleneck models and interfacing them with the DSE framework. Acquisitions made by the proposed DSE systematically cater to multiple bottlenecks that arise in executions of multi-functional workloads or multiple workloads with diverse execution characteristics. By reasoning about obtained designs and their costs, the proposed framework optimizes both the hardware and software configurations in a tightly coupled manner.

Evaluations of the proposed framework for recent computer vision and language models show that Explainable-DSE mostly explores effectual candidates, achieving codesigns of 6× lower latency in 47× fewer iterations vs. non-explainable DSEs using evolutionary or ML-based optimizations. By taking minutes or tens of iterations, it enables opportunities for runtime DSEs.

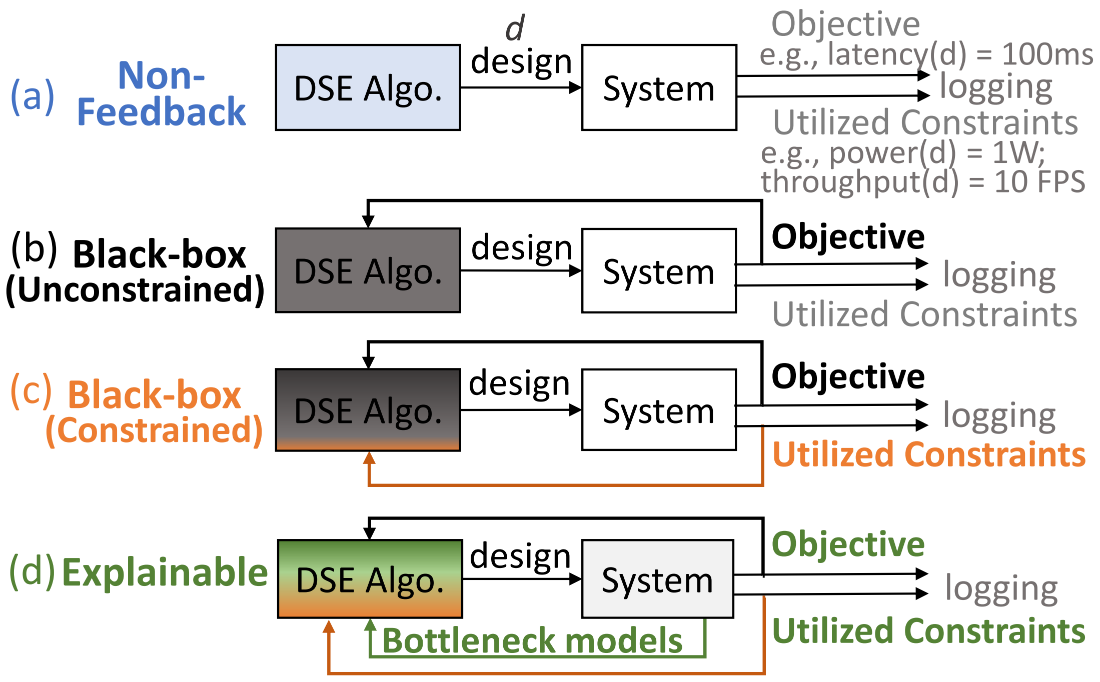


Figure 3.1: DSE with (a) Non-feedback, (b) Unconstrained black-box, (c) Constrained black-box, and (d) Explainable optimization, which leverages domain-specific bottleneck models.

3.1 Overview

Domain-specific architectures, e.g., for deep learning models, are deployed from datacenters to edge. In order to meet strict constraints on execution costs (e.g., power and area) while minimizing an objective (e.g., latency), their hardware/software codesigns must be effectively explored using an effective *design space exploration* (DSE). However, the search space is vast (e.g., contain $O(10^{29})$ solutions), with each evaluation taking milliseconds or even hours [41]. For instance, [41] showed that a TPU-like architecture has 10^{14} hardware solutions with modest options for design parameters. For every hardware configuration, software space can also be huge. For example, DNN layers can be mapped on a spatial architecture in $O(10^{15})$ ways aka dataflows [86] (also see discussion for Table 2.1 in Chapter 2), even after aggressively pruning the mapping space. Clearly, an effective exploration is needed to achieve feasible and efficient solutions [1] quickly.

Recent DSE techniques for deep learning accelerators use non-feedback or feedback-based, black-box optimizations. *Non-feedback* optimizations include grid search (in [48, 49]) and random search (in [47, 50]). They evaluate different solutions for a pre-set number of iterations and terminate (Fig. 3.1a). *Black-box* optimizations, on the other hand, consider value of the objective before acquiring [2] the next candidates (Fig. 3.1b-c). Thus, they can be more effective than non-feedback approaches. They include simulated annealing [59], genetic algorithm [51, 53, 54], Bayesian optimization [2, 41, 61–64, 155], and reinforcement learning [65–67, 72, 156]. These optimizations can be unconstrained or constrained.

For vast accelerator hardware/software codesign space, existing techniques require *excessive trials* for convergence or even finding a feasible solution. This is because of lack of explainability during the exploration. **By *explainability*, it is implied herein the ability of the DSE to reason about, at each attempt, why a certain design corresponds to specific costs, and what are the underlying inefficiencies, and how they can be ameliorated.** Existing DSE approaches are non-explainable, as they lack information and reasoning about the quality of designs acquired during DSE. They may be able to figure out which of the previous trials reduced the objective but they cannot determine *why*? Consequently, in deciding the acquisition targets for the next trial, they cannot reason about and estimate the quality of the next possible candidates. In contrast, an explainable DSE would

[1] A feasible solution meets all constraints, and its hardware and software configurations are compatible; An efficient solution minimizes objective; Agility refers to DSE's ability to find desired solutions quickly, which becomes crucial for exploring vast space in practical DSE budgets and runtime DSEs.

[2] Acquisition refers to a step in a DSE algorithm that selects next set of candidate designs to evaluate. §3.2.1 discusses the terminology for the DSE techniques.

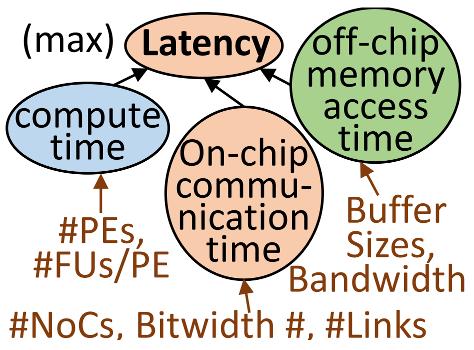

(max)
Latency
compute time
On-chip commu-nication time
off-chip memory access time
#PEs, #FUs/PE
Buffer Sizes, Bandwidth
#NoCs, Bitwidth #, #Links

Figure 3.2: Example bottleneck model of DNN accelerator latency.

identify inefficiencies in the acquired design that incur high costs and also suggest mitigation options that would improve designs and execution further. For instance, for reducing the latency of a DNN accelerator, an explainable DSE could reason that the latency is dominated by memory access time that cannot be hidden behind the time for computation or communicating data on-chip. Therefore, it could strive to reduce the latency further by increasing off-chip bandwidth or on-chip buffer size to further exploit the available data reuse.

The goal of this chapter is to develop a framework for an agile and effective DSE of hardware/software codesigns of accelerators by introducing explainablity in the DSE process.

Proposed approach to achieve explainable DSE is through the use of bottleneck analysis. Enabling explainability in DSE with bottleneck analysis requires ***bottleneck models***. Conventional DSEs evaluate *cost models* that provide just a single value like latency. In contrast, a bottleneck model is a graphical representation of which and how various design parameters and intermediate factors contribute to the total cost. For instance, a toy example tree in Fig. 3.2 illustrates how the time for computation, memory accesses, and NoC communication are intermediate factors derived from hardware/software parameters, leading to total latency. Thus, bottleneck models can provide rich information in an explicitly analyzable format. Bottleneck models can also help find mitigation, i.e., when any factor (on-chip communication) gets identified as a bottleneck, how to tune different design parameters based on the workload execution-related characteristics (e.g., increase bit-widths of NoCs by certain amount or increase physical links or time-shared unicast support). These bottleneck models can be developed based on domain-specific information, which is often embedded within experts-defined, domain-specific cost models (like [40, 121]) but *implicitly*. Having the *explicitly* analyzable bottleneck models and their driving the DSE can help DSE explain inefficiencies of acquired designs (referred to as *bottleneck analysis*) and to make mitigating acquisition decisions.

For enabling DSE of deep learning accelerators using bottleneck analysis, proposed approach overcomes the following shortcomings.

1) This chapter develops a bottleneck model for deep learning accelerators. Taking latency minimization as an example, it is described what execution characteristics of DNN accelerators need to be leveraged, how to construct a corresponding bottleneck model, how its bottleneck graph provides insights in design/execution inefficiencies, how to pinpoint bottlenecks, and what are mitigation options for identified bottlenecks. By applying bottleneck analysis on software-optimized executions of each hardware design, proposed DSE co-explores both hardware-software configurations of DNN accelerators in adaptive and tightly coupled manner.

2) This chapter proposes an API for specifying domain-specific bottleneck models and interfacing them with the DSE. Through proposed API, bottleneck model of an architecture/system can be described as a tree corresponding to the target cost. Navigating such tree enables the DSE to analyze the bottlenecks, relate the bottlenecks with the design parameters, and reason about the desired scaling for mitigation. For instance, by parsing a latency tree (Fig. 3.2), the DSE could reason that latency is a maximum value of the time taken for computations, on-chip communications, and memory accesses; if computational time exceed other factors by 3×, then the related parameters (number of functional units in PEs and number of PEs) may need to be scaled next accordingly.

The API can allow expert designers to systematically express their domain-specific bottleneck models and integrate them in DSE while leveraging constrained exploration framework. This helps overcome a limitation of previous DSEs using bottleneck analysis in other domains like multimedia or FPGA-HLS [157–160] which lack such interface; as search mechanisms were defined in domain-specific ways for their bottleneck models, they could not be decoupled or reused for other domains.

3) This chapter proposes a generic framework for constrained DSE using bottleneck models, with acquisitions accounting for multiple bottlenecks in multi-workload executions. Previous DSEs using bottleneck analysis optimize only a single task at a time, i.e., consider a single cost value of executing a loop-kernel or a whole task and iteratively mitigate its bottleneck. However, when workloads involve different functions of diverse execution characteristics, e.g., a DNN with multiple layers or multiple DNNs, changing a design parameter impacts their contribution to overall cost in distinct ways; considering just a total cost could not be useful. Also, mitigation strategies to address layer-wise bottlenecks can lead to range of different values for diverse parameters. So, proposed framework systematically aggregates parameters predicted for mitigating bottlenecks in executions of multiple functions in one or more workloads, for making next acquisitions. Lastly, the DSE exploits awareness about constraints utilization, striving to explore among feasible solutions in the vast space and finding more efficient solutions, without quickly exhausting the constraints.

Results: This chapter demonstrates proposed explainable and agile DSE framework by exploring high-performance edge inference accelerators for recent computer vision and language processing models. By iteratively mitigating bottlenecks, Explainable-DSE reduces latency under constraints in mostly every attempt (1.3× on average). Thus, it explores effectual candidates and achieves efficient codesigns in *minutes*, while non-explainable optimizations may fail to obtain even a feasible solution over days. Explainable-DSE obtains codesigns of 6× lower latency in (36× less search time on average and up to 1675×) 47× fewer iterations vs. previous DSE approaches for DNN accelerators. By achieving highly efficient solutions in only 54 iterations, Explainable-DSE enables opportunities for cost-effective and dynamic explorations in vast space.

3.2 Accelerator Hardware/Software Codesign Exploration

3.2.1 DSE Problem Formulation and Terminology

Exploration of accelerator designs is a *constrained minimization* problem, where the most efficient *solution* [3] corresponds to minimized *objective* (e.g., latency), subjected to *inequality constraints* on some *costs* (e.g., area, power) and *parameters* p of accelerator design [161]. Fig. 3.3 illustrates the DSE problem formulation. During the optimization, every solution gets evaluated by *cost models* for objectives and inequality constraints. The DSE technique needs to consider only *feasible* solutions and determine the most efficient solution by processing several *iterations* (*trials*). It is a discrete optimization since the *search space* is usually confined to presumably effective solutions, e.g., power-of-two or categorical values of parameters. It is also *derivative-free optimization* [2].

$$\begin{array}{lll} min & obj(p), & p = (p_1, p_2, ..., p_n) \in \mathbb{R}^n \\ subject\ to & cost_i(p) \leq constraint_i; & for\ i = 1, 2, ..., m \end{array}$$

[3] In the accelerator design space exploration context, terms "solutions", "designs", and "configurations" are used interchangeably.

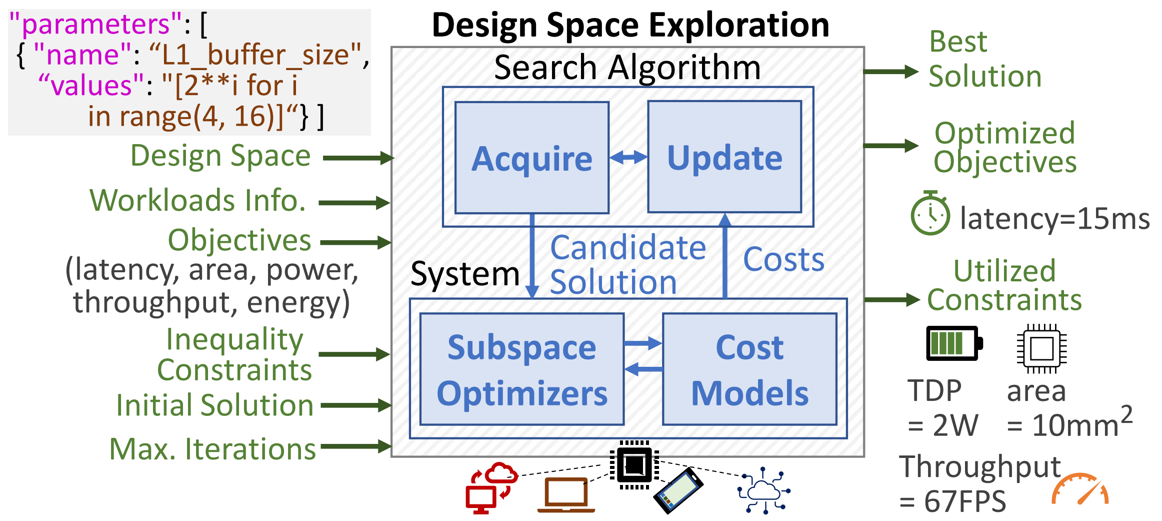


Figure 3.3: Accelerator DSE as constrained minimization.

3.2.2 Accelerator Hardware/Software Codesigns DSE

Hardware/software codesigns can be explored by partitioning the search space and optimizing *software* space as a *subspace* in a loop. So, the DSE technique needs to find the best mapping of a task onto architecture and repeat the search with different architectural configurations [162]. Partitioning enables exploration in reduced space compared to exploring parameters from multiple spaces altogether. DSE techniques for DNN accelerators explore hardware designs through non-feedback or black-box optimizations like evolutionary or ML-based [41, 51, 53, 54, 61–63, 65–67, 72, 155]. Such approaches are also commonly used for designing or optimizing computing systems in general [2, 56, 163–169]. For mapping DNNs on a design (subspace optimization), they typically fix the way of execution or dataflow, e.g., in [61, 65, 67, 70–72, 157]. Hence, for processing each functionality (nested loop such as a DNN layer), these techniques usually have just one mapping. Thus, they primarily optimize designs of accelerator architecture, i.e., parameters for buffers, processing elements (PEs), and NoCs.

3.3 Limitations of Prior DSE Approaches

Non-feedback DSE approaches search used by previous techniques either exhaustively over statically reduced space (e.g., grid search in [49, 170]) or randomly (e.g., in [47]). So, they do not consider any outputs like objective or utilized constraints and terminate after using a large exploration budget. It is illustrated by Fig. 3.1(a). On the other hand, black-box optimizations such as Bayesian Optimization (e.g., in [2, 41, 61–63]) consider values of the objective for previously tried solutions. It is illustrated by Fig. 3.1(b)–(c). Considering the objective helps them predict the likelihood of where the minima may lie; they acquire a candidate for the next trial accordingly. The process repeats until convergence or the number of trials exceeding a threshold. While black-box DSE could be more efficient than non-feedback DSE, they all face the following limitations:

Previous DSE techniques lack reasoning about bottlenecks incurring high costs: An efficient DSE mechanism should determine challenges hindering the reduction of objectives or utilized constraints. It should also determine which among the many parameters can help mitigate those inefficiencies and with what values. However, with objective as the only input, these black-box or system-oblivious DSEs can figure out only which prior trials reduced objective. But, they do not reason about what costs a solution could lead to and why – a crucial aspect in exploring enormous design space. This challenge is exacerbated by the fact that execution characteristics of different functions in workloads are diverse (e.g., memory- vs. compute-bounded DNN operators, energy consumption characteristics). By considering just the total cost, black-box DSEs cannot consider diverse bottlenecks in multi-modal or multi-workload executions, which need to be addressed systematically.

Implications: A major implication of excessive sampling caused by lacking

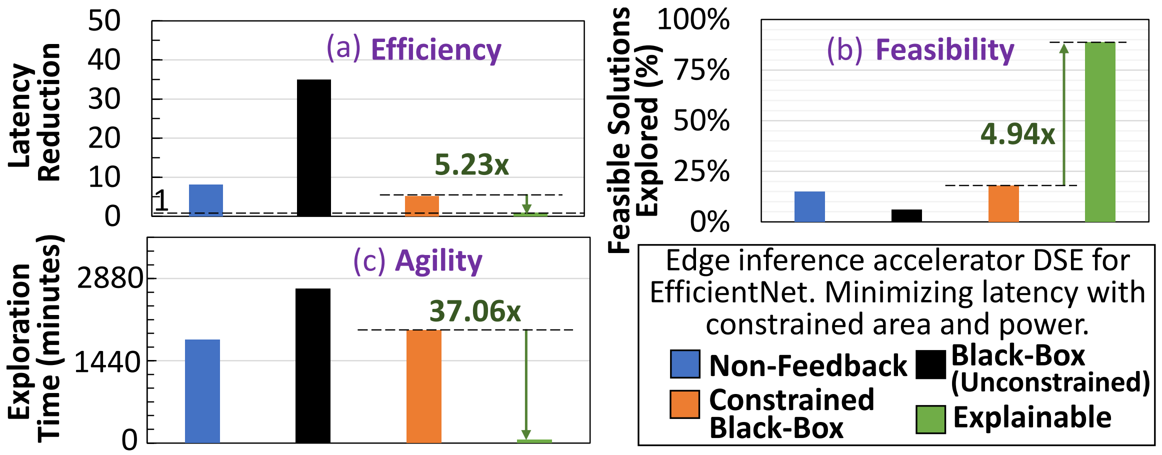


Figure 3.4: Effectiveness of non-explainable and explainable DSE frameworks for exploring efficient and feasible solutions in the vast space: (a) Efficiency (latency minimization), (b) Feasibility (in % of the total solutions evaluated); (c) Agility (exploration time in minutes). Analysis is shown here for exploring edge accelerator design for EfficientNetB0 model.

explainability is **inefficiency of obtained solutions**. Fig. 3.5(a) illustrates this through a toy scenario, i.e., exploring the number of PEs and global buffer size for a single ResNet layer. It shows an exploration done by earlier to later trials with HyperMapper 2.0 [2] – an efficient, Bayesian-based optimizer. The figure shows that even for a tiny space, acquired solutions are inefficient (high latency), as there is no reasoning about underlying bottlenecks and their mitigation. So, even though DSE has already acquired some better solutions before, the later acquisitions correspond to inefficient solutions. As the design space becomes vast, the non-explainable DSEs can require too many trials (at least, in thousands [2, 41, 65]), and they may still not find the most efficient solutions. For example, Fig. 3.4(a) shows that the latency of the solutions obtained by non-explainable DSEs can be up to 35$\times$ higher, even for 2500 trials (two days of search time). This is because, practical exploration budget is fractional (thousands) compared to vast design space (quadrillions). By generating trials without understanding bottlenecks and their mitigation, most of the search budget gets spent on excessive and likely ineffectual trials.

Lacking reasoning about inefficiencies can deprive the DSE of **a tightly coupled hardware/software codesign optimization**. For instance, DSEs in [61, 65–67, 70–72] mainly explore architectural parameters with black-box DSEs and use a fixed dataflow for executions. [4] Fixing the execution methods limit the effectual utilization of architectural resources when subjected to various tensor shapes and functionalities [33, 95]. Consequently, DSEs may achieve architecture designs that are either incompatible with the dataflow (**infeasible solutions**) or **inefficient**. Likewise, in *isolated* co-optimizations, obtained HW design and dataflow are *oblivious* of each other, leading to excessive trials and inefficient solutions.

For constrained optimizations, **lack of awareness about utilization of constraints** in black-box DSEs leads to exploring infeasible and inefficient solutions and excessive trials. This is because DSEs cannot determine which constraints are violated, which regions could exhaust constraints quickly while not optimizing objective much, and how configuring different accelerator design parameters could affect all this. Fig. 3.4(b) illustrates this for an edge accelerator DSE subjected to power and area constraints. For constrained optimizations like HyperMapper2.0 [2], out of 2500 trials, only 18% of the evaluated solutions were feasible, and up to only 52% for constrained reinforcement learning [65].

Another implication of excessive trials is **inapplicability to dynamic DSE scenarios**. Excessive trials lead to low agility, as illustrated in Fig. 3.4(c). Non-explainable DSEs consume very high exploration time, even *weeks*, while obtaining solutions of lower efficiency. It makes existing DSE approaches unsuitable for dynamic explorations (e.g., convergence within a few tens to 100 iterations). For instance,

[4]§3.2.2 and §3.11 provide background on HW/SW codesign DSE.

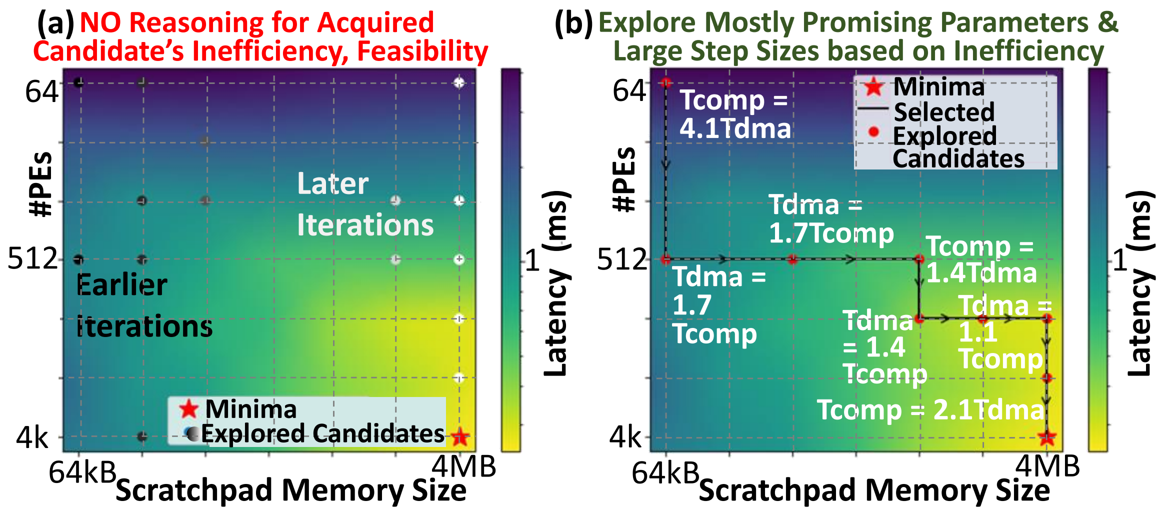


Figure 3.5: Example DSE of #PEs and shared memory sizes for ResNet CONV5_2b [1] with (a) Prior techniques (HyperMapper2.0 [2]), and (b) Explainable-DSE, which reasons about inefficiencies in achieved executions, limiting the search to crucial parameters and tuning accordingly.

unlike one-time ASIC designs, deploying accelerator overlays over FPGAs (edge/cloud; dedicated/multi-tenant) can benefit from dynamic DSEs, where constraints for DSE and resource budget may also become available just before deployment.

3.4 DSE Using Bottleneck Analysis: Motivation and Challenges

3.4.1 Making DSE Explainable Through Bottleneck Analysis

Fig. 3.5(b) illustrates the same problem of designing a DNN accelerator as in Fig. 3.5(a), but by using bottleneck analysis in the DSE. Before acquiring new candidates, the DSE analyzes current design through the bottleneck model and pinpoints the bottleneck in achieved latency. Then, it uses mitigation obtained from the bottleneck model to make next acquisitions. A bottleneck, in the context of the latency optimization for a deep learning accelerator, can be attributed to one of the execution factors, such as time consumed by computations, communication via NoCs, and off-chip memory accesses with direct memory access (DMA) controller. For instance, after evaluating the initial point *(number of PEs, shared memory size) = (64, 64kB)*, the DSE can reason that the computation time of the design is 4.14× higher than the time taken by off/on-chip data communication. From the mitigation strategy, DSE concludes and communicates to the designers that it would scale the total number of PEs next by at least 4.14×. [5] Since this is the only mitigation suggested, the newly acquired and optimized design becomes (512, 64kB). By repeating this process, the DSE informs that the previous bottleneck got mitigated and DMA-transfers is the new bottleneck. Using the bottleneck model, the DSE considers execution characteristics (like data accessed from off-chip memory and unexploited data reuse) and mitigation for the current design point, adjusting the size of shared on-chip memory or off-chip bandwidth. This iterative process continues. It not only enables the DSE to characterize and explain DSE decisions but also reduces objectives at almost every acquisition attempt, converging to efficient solutions quickly.

3.4.2 Challenges in Enabling DSE of Accelerators Using Bottleneck Analysis

Need bottleneck models for DNN accelerators. DSE using bottleneck analysis requires bottleneck models. Unlike cost models used in black-box DSEs that provide a single value, bottleneck models can provide rich information, in an explicitly analyzable manner, about 1) how design parameters contribute to different factors that lead to the total cost, and 2) mitigation options when any factor gets identified as a bottleneck. Such bottleneck/root-cause analysis have been applied for characterizing fixed designs and finding mitigation, e.g., for industry pipelines

[5] Just to note the power of explicit bottleneck mitigation strategies, if area constraint was unmet, DSE could intelligently let communication time increase but meet constraints first through reduced buffer/NoC sizes.

and production systems, hardware or software for specific applications [97, 157, 171], FPGA-based HLS [159, 160], overlapping microarchitectural events [172], and power outage [173]. Likewise, optimizing DNN accelerators with bottleneck analysis also require developing bottleneck models.

Need an interface to decouple domain-specific bottleneck models from a domain-independent exploration mechanism and express them to DSE. Once bottleneck models are developed, there needs to be a DSE framework that can integrate such a domain-specific bottleneck model to drive the iterative search. However, since bottleneck models are usually domain-specific, search mechanisms provided by prior DSE techniques using bottleneck analysis [157–159] are implemented too specifically for their domain. There needs to be an interface to decouple the domain-independent search mechanism from domain-specific bottleneck models so that designers can reuse and apply the same search mechanism for exploring designs in new domains like DNN acceleration.

Need acquisitions accounting for mitigation of multiple bottlenecks in multi-functional or multiple-workload executions. Prior DSE techniques using bottleneck analysis (in other domains) [157–160] optimize only a single task at a time, i.e., consider a single cost value of executing a loop-kernel or whole task and iteratively mitigate arising bottleneck. However, when workloads involve different functions of diverse execution characteristics, e.g., a DNN with multiple layers or multiple DNNs, changing a design parameter impacts their contribution to the overall cost in distinct ways; considering just a total cost may not be useful. Mitigation strategies to address these layer-wise bottlenecks can lead to changing diverse parameters and a range of values possible for the same parameter. Therefore, when the DSE makes its next acquisitions, it needs to ensure that multiple bottlenecks arising from executing different functions of target workloads are mitigated systematically and effectively.

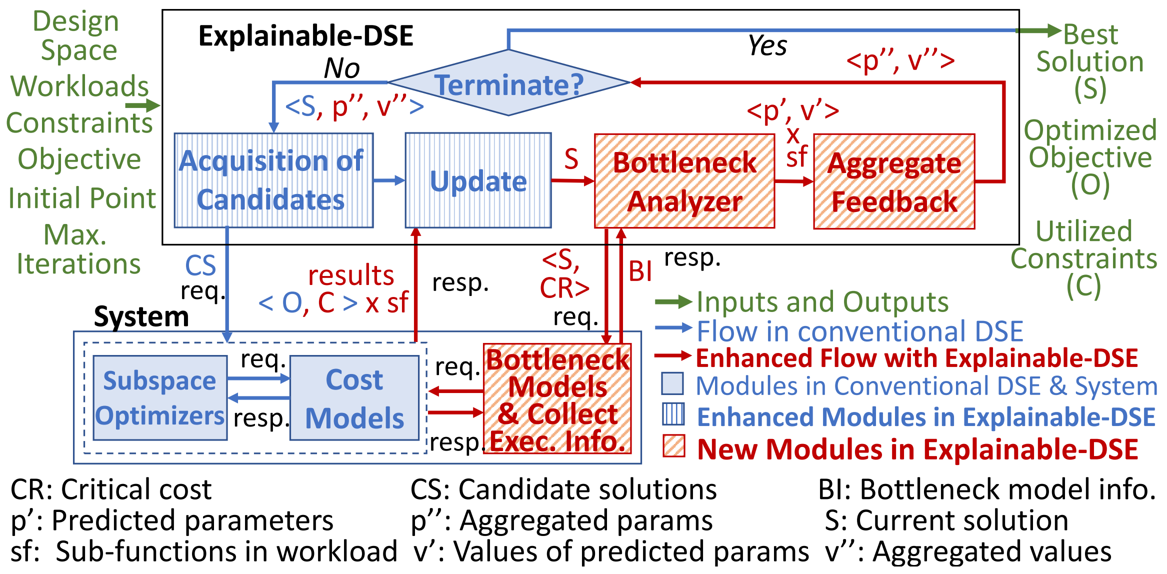
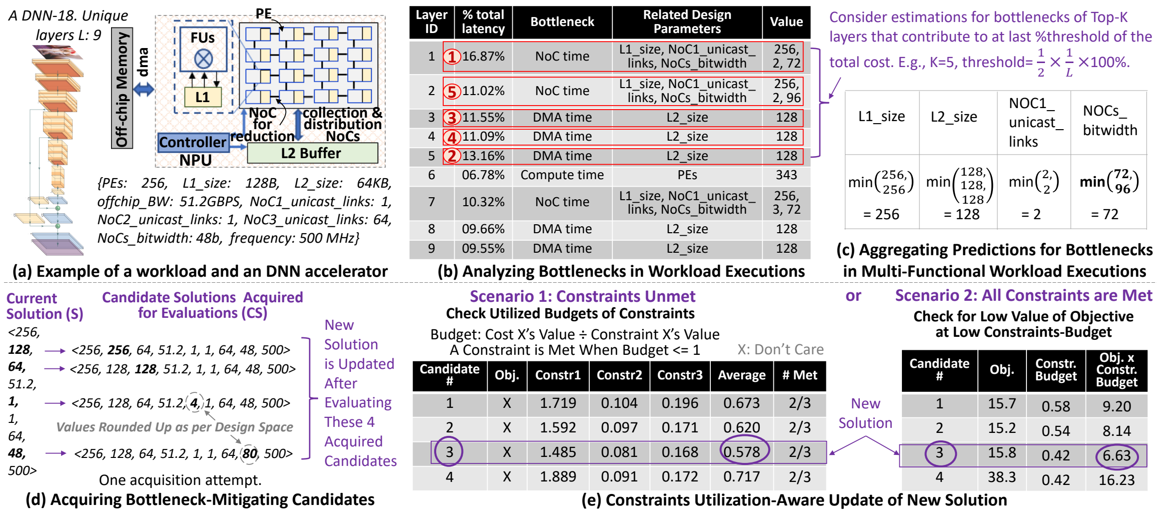


Figure 3.6: Explainable-DSE: A framework for exploring design space using domain-specific bottleneck models.

3.5 Explainable-DSE: Constraints-aware DSE Using Bottleneck Analysis

This section presents Explainable-DSE – This section presents Explainable-DSE – a framework for an agile and explainable DSE using bottleneck analysis for optimizing deep learning accelerator designs. First, this chapter discusses proposed framework's overall workflow and illustrate it with a walk-through example. Then, it describes how its bottleneck analyzer processes bottleneck models, i.e., determines factors incurring a high cost, parameters relevant to the bottleneck factors, and new values of parameters that can reduce the cost. This chapter also introduces an API through which architects can specify domain-specific bottleneck models, e.g., for analyzing accelerator execution costs and bottleneck mitigation strategies. For bottleneck analysis involving the execution of multiple workloads or multiple functions within a workload, it discusses how Explainable-DSE aggregates the obtained parameters and their new values, including considering bottlenecks of only execution-critical functions. The chapter


A DNN-18. Unique layers L: 9
Off-chip Memory
dma
FUs
L1
PE
NoC for reduction
collection & distribution NoCs
Controller
NPU
L2 Buffer
{PEs: 256, L1_size: 128B, L2_size: 64KB, offchip_BW: 51.2GBPS, NoC1_unicast_links: 1, NoC2_unicast_links: 1, NoC3_unicast_links: 64, NoCs_bitwidth: 48b, frequency: 500 MHz}
(a) Example of a workload and an DNN accelerator
Consider estimations for bottlenecks of Top-K layers that contribute to at last %threshold of the total cost. E.g., K=5, threshold=$\frac{1}{2} \times \frac{1}{L} \times 100\%$.
Current Solution (S)
Candidate Solutions Acquired for Evaluations (CS)
Values Rounded Up as per Design Space
New Solution is Updated After Evaluating These 4 Acquired Candidates
One acquisition attempt.
(d) Acquiring Bottleneck-Mitigating Candidates
Scenario 1: Constraints Unmet
Check Utilized Budgets of Constraints
Budget: Cost X's Value ÷ Constraint X's Value
A Constraint is Met When Budget <= 1
X: Don't Care
or
Scenario 2: All Constraints are Met
Check for Low Value of Objective at Low Constraints-Budget
New Solution
(e) Constraints Utilization-Aware Update of New Solution

Layer ID	% total latency	Bottleneck	Related Design Parameters	Value
1	① 16.87%	NoC time	L1_size, NoC1_unicast_links, NoCs_bitwidth	256, 2, 72
2	⑤ 11.02%	NoC time	L1_size, NoC1_unicast_links, NoCs_bitwidth	256, 2, 96
3	③ 11.55%	DMA time	L2_size	128
4	④ 11.09%	DMA time	L2_size	128
5	② 13.16%	DMA time	L2_size	128
6	06.78%	Compute time	PEs	343
7	10.32%	NoC time	L1_size, NoC1_unicast_links, NoCs_bitwidth	256, 3, 72
8	09.66%	DMA time	L2_size	128
9	09.55%	DMA time	L2_size	128

(b) Analyzing Bottlenecks in Workload Executions

L1_size	L2_size	NOC1_unicast_links	NOCs_bitwidth
$\min\binom{256,}{256}$ = 256	$\min\begin{pmatrix}128,\\128,\\128\end{pmatrix}$ = 128	$\min\binom{2,}{2}$ = 2	$\min\binom{72,}{96}$ = 72

(c) Aggregating Predictions for Bottlenecks in Multi-Functional Workload Executions

Current Solution (S)	Candidate Solutions Acquired for Evaluations (CS)
<256,	
128,	<256, **256**, 64, 51.2, 1, 1, 64, 48, 500>
64,	<256, 128, **128**, 51.2, 1, 1, 64, 48, 500>
51.2,	
1,	<256, 128, 64, 51.2, **4**, 1, 64, 48, 500>
1,	
64,	
48,	<256, 128, 64, 51.2, 1, 1, 64, **80**, 500>
500>	

Candidate #	Obj.	Constr1	Constr2	Constr3	Average	# Met
1	X	1.719	0.104	0.196	0.673	2/3
2	X	1.592	0.097	0.171	0.620	2/3
3	X	1.485	0.081	0.168	0.578	2/3
4	X	1.889	0.091	0.172	0.717	2/3

Candidate #	Obj.	Constr. Budget	Obj. x Constr. Budget
1	15.7	0.58	9.20
2	15.2	0.54	8.14
3	15.8	0.42	6.63
4	38.3	0.42	16.23

Figure 3.7: Example walkthrough. a) A DNN and accelerator architecture parameters, b) Analyzing bottlenecks for executing each layer/function, c) Aggregating bottleneck mitigation for multi-functional/multiple workloads, d) Acquiring new candidates that mitigate bottlenecks; e) Constraints utilization-aware update of the new solution.

then describes how the proposed framework considers inequality constraints when updating the obtained solutions, prioritizing exploration of feasible regions. It also provides an in-depth bottleneck model and bottlenecks mitigation strategies for exploring low-latency designs of DNN accelerators using Explainable-DSE. Lastly, it discusses how proposed approach can enable a tightly coupled accelerator/mappings co-explorations.

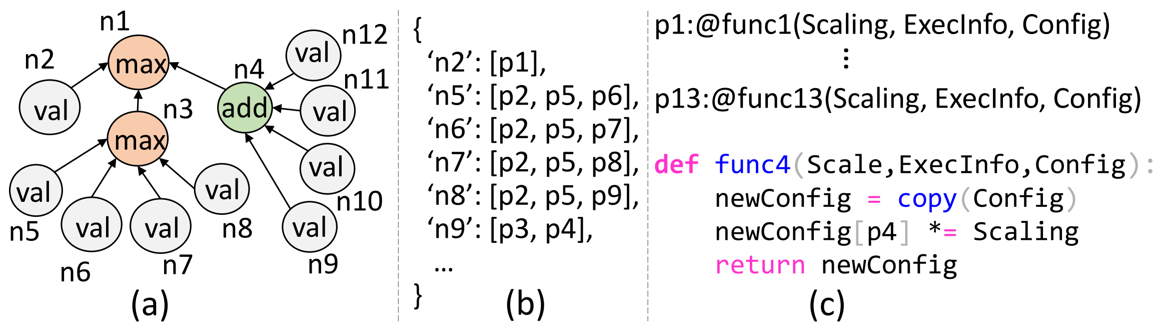
3.5.1 Framework Workflow

Fig. 3.6 illustrates the workflow of Explainable-DSE. The DSE uses bottleneck analysis to explore solutions that reduce a critical cost, denoted as CR. Critical cost is usually an objective O that needs to be minimized, and optionally an unmet inequality constraint value C. To reduce the cost, the bottleneck analyzer considers the current solution (S) and analyzes cost-related bottleneck information (I). The analyzer identifies the bottleneck factors incurring higher cost value and finds the *scaling* "**s**" by which the objective/constraint value needs to be reduced ("s" is internal to the analyzer, so not shown in Fig. 3.6). Then, the analyzer determines design parameters (p') crucial for mitigating the bottleneck and their values (v'). Workloads can be multi-modal or usually involve multiple sub-functions (sf), e.g., the accelerator needs to be optimized for different DNNs or various layers in a DNN. So, the DSE applies bottleneck analysis to the costs of each sub-function individually and aggregates the corresponding feedback obtained. This aggregation leads to a set of predicted design parameters (p'') and their respective values (v''). Based on these predictions, a new set of candidate solutions (CS) is derived for the subsequent acquisition. The process iterates, as depicted in Fig. 3.6. Acquiring and evaluating candidates in the CS is referred to herein as one "*acquisition attempt*". It is analogous to z sequential DSE iterations if a CS contains z candidates. The current solution, S, is updated once (from z candidates) at every acquisition attempt. When some inequality constraint is not met, the framework considers the utilized budgets of constraints for acquired candidates in updating the current solution. This approach enables the DSE to prioritize reaching feasible subspaces. In Fig. 3.6, the introduction of new modules for the proposed approach and corresponding information flow is illustrated through a diagonal stride pattern and a different shade (red). The workings of these modules are described next, accompanied by a walk-through example (illustrated in Fig. 3.7). Additional information regarding the capabilities of the framework, current limitations, and future works for further automation and enhancements are discussed in §3.13 and §3.14, respectively.

3.5.2 Framework Inputs and Outputs

Inputs: Information of the design space, constraints, objective, workloads, initial point, and total iterations. *Outputs* upon convergence or termination: Optimized solution and its costs.
Design Space: It defines the design parameters of type integer, real, or categorical. Their possible values can be expressed as either a list or a mathematical expression.
Constraints and Objective: Users can define inequality constraints on multiple costs. Proposed implementation currently optimizes a single objective. It can be extended for multiple objectives through existing acquisition techniques.
Target System and Cost Models: System can incorporate arbitrary cost models and subspace optimizations for populating costs. It can also provide costs at sub-functions granularity, e.g., the latency of individual DNN layers. The proposed API (§3.5.3) enables the seamless integration of the bottleneck models.

To demonstrate DNN accelerator design explorations, this work leverages existing cost models and use them to evaluate all techniques. Accelergy [89] is used to obtain the total area, energy per data access (for 45nm technology node), and maximum power. The maximum power is obtained from the maximum energy consumed by all design components in a single cycle. Accelergy provides technology-specific estimations via plugins for Aladdin [174] and CACTI [175]. dMazeRunner infrastructure [40] proposed in this disseration is used to obtain the latency and energy consumed by mappings of DNN layers and for quick mapping optimizations for each architecture design.



```
{
 'n2': [p1],
 'n5': [p2, p5, p6],
 'n6': [p2, p5, p7],
 'n7': [p2, p5, p8],
 'n8': [p2, p5, p9],
 'n9': [p3, p4],
 ...
}
```

```
p1:@func1(Scaling, ExecInfo, Config)
                 ⋮
p13:@func13(Scaling, ExecInfo, Config)

def func4(Scale,ExecInfo,Config):
    newConfig = copy(Config)
    newConfig[p4] *= Scaling
    return newConfig
```

Figure 3.8: Proposed API through which designers or design automation tools can specify a bottleneck model of a system. The information can contain: (a) Bottleneck graph containing factors contributing to a cost, (b) Different parameters impacting the factors, and (c) Handles to subroutines that calculate new values of the parameters.

3.5.3 Bottleneck Analyzer

Before each acquisition attempt, Explainable-DSE conducts bottleneck analysis on the obtained solution from previous attempt. It uses the bottleneck model, which helps pinpoint the execution bottlenecks and suggests options to mitigate them, ultimately reducing costs. For instance, Fig. 3.7 demonstrates this exploration process for an 18-layer DNN, where nine layers have unique tensor shapes for execution-critical operators (CONV and GEMM). Fig. 3.7(a) shows the architectural template and parameter values of the current solution during the DSE. Fig. 3.7(b) displays the bottleneck analyzer's ability to identify bottlenecks for each DNN layer and estimate which parameters should be updated with what specific values. This section further explains how the analyzer works and presents an API through which designers can specify their domain-specific bottleneck models for the DSE.

By evaluating the bottleneck model, the bottleneck analyzer determines (a) bottleneck factors, (b) parameters that are most critical for reducing the costs of these bottleneck factors, and (c) values of these critical parameters. Designers can provide the information for bottleneck models through an API that comprises up to three data structures, as illustrated in Fig. 3.8. The first and the key data structure is a graph of the bottleneck model, which outlines the underlying factors contributing to the total cost. The second includes a list of related parameters for each factor. The third contains handles to subroutines that predict the next values of parameters. When some information is unavailable, such as how to predict the value of a parameter, Explainable-DSE resorts to its black-box counterpart (e.g., sampling neighboring values).

(a) Determining bottleneck factors from bottleneck model graph: A bottleneck model is a graphical representation of which and how various factors contribute to the cost of executing a workload on an accelerator, as depicted in Fig. 3.8(a). It is represented as a tree whose nodes are mathematical functions like addition, multiplication, division, and maximum. Each node typically represents a cost factor, which is calculated from values of its children by applying the corresponding mathematical function. Thus, the root node of the bottleneck model represents the total cost and leaf nodes are hardware, software, or execution related design parameters.

For example, Fig. 3.9 shows a simplified bottleneck model for a DNN layer execution, where the root corresponds to the overall cost (e.g., latency). The total cost depends on child nodes representing underlying cost factors. For example, the total latency is determined as the maximum value among the computational time, the total on-chip communication time, and the total DMA time for off-chip memory accesses. The total DMA time, in turn, is additive and depends on the off-chip footprint of different tensors and the bandwidth. Similarly, the time for communicating data from on-chip buffers to PEs via NoCs is approximated with the total data packets communicated to different workgroups and NoC bus widths. Leaf

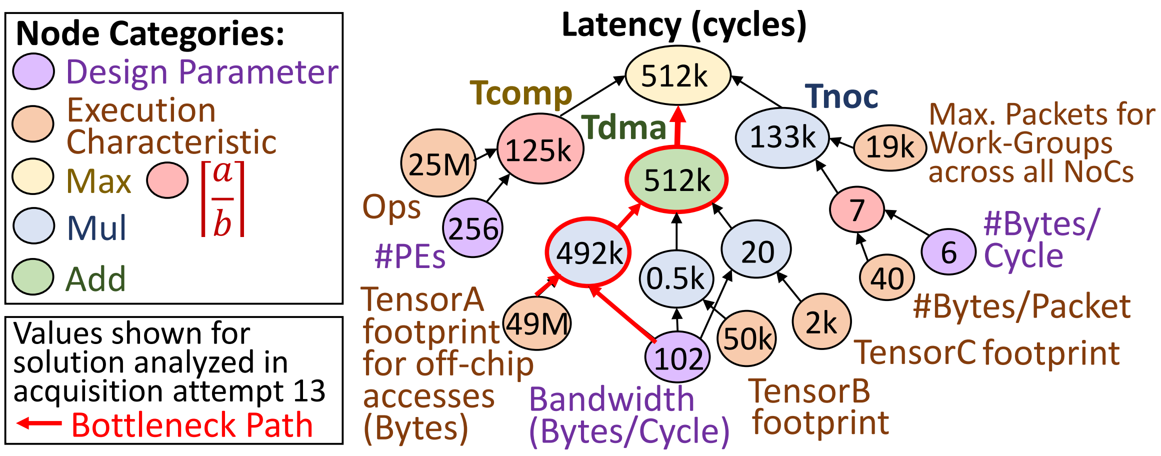


Figure 3.9: A simplified bottleneck model for analyzing the latency of a DNN layer execution on a DNN accelerator. As compared to conventional cost models that provide a single value, the graph-based bottleneck models can provide richer information in an explicitly analyzable format and outline how hardware/software parameters relate to total cost, allowing designers and the DSE to make informed decisions.

nodes in a bottleneck graph typically represent values of the *design parameters* from the design space, such as hardware design parameters, application parameters like tensor shapes and quantization bit-width, and the accelerator's execution characteristics for a given workload or application or the code optimization parameters. Execution characteristics include data allocation to buffers, on/off-chip communication of data, and unexploited reuse, etc. (§3.5.7) and obtained from mapping of a workload on the accelerator.

During each acquisition attempt, the analyzer considers current solution and populates the graph with the corresponding actual values. For each cost factor, which is an intermediate node, the analyzer calculates its contribution to the total cost as the ratio of its value to the total cost. The analyzer traverses the graph and computes contribution of each factor based on the associated mathematical operation. For instance, at a max node, it traces back to the maximum value; at an add node, it counts contributions proportionally. It identifies the factor with the highest contribution as the primary bottleneck. The analyzer then calculates the **scaling "s"**, which is the ratio by which the cost of the bottleneck factor should be reduced to alleviate bottleneck. In Fig. 3.9, DMA time dominates the total latency, whereas the computational and on-chip communication time contributes to only 24.4% and 25.9% of the total latency, respectively. The analyzer finds that the later factors can be balanced by scaling down the DMA time, e.g., by a factor of 100% ÷ 25.9% or 3.85×. Through traversal, the analyzer identifies the memory footprint of tensor A as the primary bottleneck operand. The analyzer may also determine multiple bottlenecks (based on decreasing order of their contributions) so that the acquisition function can generate an adequate number of candidates.

(b) Selecting parameters associated with the bottleneck: To determine which parameters impact specific bottleneck factors, the analyzer can traverse the bottleneck graph, or designers can provide this information through a dictionary that maps the node names/numbers to relevant parameters (Fig. 3.8b). In the example bottleneck graph of Fig. 3.8(a), nodes '$n4$' and '$n9$' correspond to DMA time and the off-chip footprint of Tensor A, respectively. They are associated with parameters '$p3$' and '$p4$' (e.g., '$L2\_size$' and '$offchip\_BW$' in Fig. 3.7a). Once the bottleneck factor and mitigating parameters are identified, DSE can obtain new values from supporting subroutines or evaluating the bottleneck path.

(c) Obtaining values of critical parameters for bottleneck mitigation: Designers can provide handles to domain-specific subroutines that contain mitigation strategies for different design parameters, as shown in Fig. 3.8(c). Each subroutine calculates the new value of a parameter based on the current parameter value, the scaling s required for reducing the bottleneck factor, and the execution characteristics of the current design configuration (§3.5.7). For example, the function '`func4`' can scale the off-chip bandwidth to reduce DMA time, and functions '`func5`' to '`func8`' can scale the bus width or NoC links to lower on-chip communication time. The DSE can leverage these subroutines to predict bottleneck-mitigating values for acquiring the next candidates.

While accelerator designers can specify bottleneck models in the proposed graphical representation, design tools or machine learning-based approaches can be developed for automatic construction of bottleneck models or mitigation options for designing new processors and off-the-shelf or large-scale architectures (§3.14).

3.5.4 Addressing Bottlenecks in Multi-Functional and Multi-Workload Executions

As Fig. 3.7(b) illustrates, the analyzer performs bottleneck analysis on each sub-function of workloads (DNN layer) one by one. Due to the diverse execution characteristics of these functionalities, the predictions obtained for each sub-function can be distinct, depending on the factors like available reuse and parallelism. Additionally, mitigation options for multiple bottlenecks in executions of various DNN layers may involve multiple values for the same parameter. Hence, an aggregation is required to determine the next set of parameters and their values (Fig. 3.7c). The DSE employs two methods for the aggregation/filtering of the predicted parameters and values:

(i) *Aggregating different values of the same parameter:* After analyzing the solution S for multiple sub-functions (identifying bottlenecks and predicting mitigation), there can be different predicted values of the same parameter. So, the final prediction can be obtained by either iterating over some of these predicted values or applying a function (maximum, minimum, average) on them. Choosing the maximum value can lead to faster convergence, but it can favor a single sub-function and be overly aggressive for others. For instance, selecting a new value as $16\times$ (from options like $4\times$, $8\times$, $16\times$) of the current number of PEs can significantly reduce latency of a non-performance-critical DNN layer but not of other layers, while consuming higher area and power. Thus, exploration can quickly exhaust the budget for constraints without getting a chance to explore a considerable range of intermediate candidates that could minimize the overall cost. Instead, selection of the minimum value as the final prediction is opted (shown in Fig. 3.7c).

(ii) *Aggregating parameters from only bottleneck sub-functions:* Not all the sub-functions or cost factors require improvement. Hence, Explainable-DSE allows focusing on only the bottleneck ones, i.e., those contributing the most to the total cost. This capability is achieved through two tunable parameters: K and *threshold*. The DSE considers predictions from up to top-K sub-functions whose fractional contributions to the total cost exceed a certain *threshold*. In target DNNs, the number of layers with unique tensor shapes (l) can range from a few to several tens. So, K is arbitrarily set to five and the *threshold* to 0.5*(1/l)*100%, considering predictions from layers that consume higher portions of the cost. For the example in Fig. 3.7, the analyzer considers mitigating bottlenecks from the top-5 layers that contribute at least 5.5% to the total latency.

3.5.5 Bottlenecks-Guided Acquisitions of New Candidates

After aggregating predicted parameter values for mitigating bottlenecks, the DSE populates the candidates CS to be acquired next. For simplicity, the acquisition function samples a new candidate for each new parameter value. As Fig. 3.7(d) shows, all but one parameter of the candidate has the same value as in the current solution. This mechanism naturally facilitates an iterative search that adaptively tunes among bottleneck parameters. It avoids a greedy local search [176] by the following means. i) It limits exploration parameters to only a few (critical for addressing the bottleneck); ii) It can predict values of larger step-size (non-neighbors) based on bottleneck mitigation analysis (whereas local search explores p immediate neighboring values for all p parameters in the selected solution). Acquisitions by addressing multiple, dynamic bottlenecks (different parameters to be optimized at each DSE iteration) and exploring larger step sizes usually help avoid over-optimization within the local neighborhood (converging to local optimal). §3.14 further discusses workarounds for overcoming the bottleneck-oriented greediness in the search. With modular framework, the designers may also specify other acquisition/update functions that act upon bottlenecks-mitigating parameters. When acquiring a candidate, if a predicted value is not present in the defined design space (e.g., non-power-of-2), the DSE rounds it up to the closest value.

3.5.6 Constraints-Budget Awareness in Updating the New Solution

When exploring a vast space under tight constraints, initially acquired solutions usually fail to meet some constraints (e.g., low-area, high-latency region). To effectively explore the space, the DSE accounts for the *constraints budget* when selecting the new solution, which, in turn, impacts the acquisitions of new candidates. In determining the new solution among the explored candidates, the DSE first checks whether the acquired candidates meet all constraints and by what margin. If any candidate does not meet all constraints, it selects a candidate that uses the least *constraints budget* as the new solution. The constraints budget is calculated as the average of the utilized constraint values that are normalized to the constraint thresholds. Such accounting is illustrated in Fig. 3.7(e) - scenario 1. Further, for monomodal cost models, when a candidate (corresponding to the new value of some parameter) violates more constraints than the obtained solution, the DSE can stop further exploration for that parameter's range. Thus, by prioritizing the feasibility of solutions, the DSE limits acquiring solutions that optimize the objective at the expense of violating constraints. When multiple candidates satisfy all constraints (scenario 2), the DSE selects the one (as the new solution) that achieves the lowest objective value with a lower constraints budget, i.e., the smallest value for *objective*$\times$*constraints budget*. Such a strategy can help avoid greedy optimization that chases marginal objective reduction, seeking more promising solutions without quickly exhausting the constraints.

3.5.7 Bottleneck Mitigation for Designing Deep Learning Accelerators

This work uses the latency of executing a DNN as an example cost for describing bottleneck mitigation for optimizing DNN accelerator/mapping codesigns. It is described herein what information about the latency can be analyzed for constructing a bottleneck model and predicting new values to mitigate various bottlenecks.

Information embedded in bottleneck model: The bottleneck model incorporates execution characteristics of an optimized mapping of a DNN layer onto an architecture design. They include:

• T_comp T_comm, T_dma: Total time consumed by computations on PEs, communicating data via NoCs, and accessing data from off-chip memory via DMA, respectively.

• $Accel\_freq$: Frequency of the accelerator (MHz)

• $data\_offchip$: Data (bytes) accessed from off-chip, per operand

• $data\_noc$: Data (bytes) communicated via NoC, per operand

• NoC_groups_needed: Maximum number of concurrent links that can be provided for communicating unique data to different PE-groups; one variable per operand.

• $NoC\_bytes\_per\_group$: Size of the data that can be broadcast to PEs within every workgroup of PEs; one variable per operand.

Using above information, a bottleneck graph can be created as illustrated in Fig. 3.9. Typically, this information is available from experts-defined cost models like [40, 89, 121]. If not, it may be obtained through similar analysis, hardware counters, or ML models.

Dictionary of affected parameters: It contains different factors contributing to the latency as keys and a list of relevant parameters as values. For example, the computation time is affected by the number of PEs and functional units in PEs. The time consumed by NoC communication is affected by the concurrent unicast links in NoCs, bit-widths of NoCs, and size of the local buffer or RF. The buffer size impacts the exploited reuse and the size of the data to be communicated. DMA time is affected by the bandwidth for off-chip memory accesses and the size of the shared memory.

Determining new values of accelerator design parameters: For a design configuration, analyzing the bottleneck model of a cost provides s, which is the scaling to be achieved by reducing a bottleneck factor's cost. $X\_current$ and X_new indicates the current and predicted value of a parameter X, respectively. X is a parameter impacting the bottleneck factor (obtained from dictionary). This chapter next describes the calculations for values of various design parameters.

• **PEs:** The number of PEs required can be calculated directly from the needed scaling. $PEs\_new = s * PEs\_current$.

• **Off-chip BW:** Bandwidth (BW) for off-chip and on-chip communication is obtained from the number of data elements communicated per operand and the target scaling factor. E.g.,

$scaled\_T\_dma = T\_dma \div s$;

$footprint = sum(data\_offchip)$;

$bytes\_per\_cycle = footprint \div scaled\_T\_dma$

$offchip\_BW\_new = bytes\_per\_cycle * Accelerator\_freq$

• **NoC Links and Bit-width:** For DNN accelerators, separate NoCs communicate different operands, each with multiple concurrent links for various PE groups. For every NoC, the maximum number of PE-groups with simultaneous access and the total bytes broadcast to each group are obtained from the cost model [40]. If communication time is a bottleneck, the operand causing it ('*op*') is available from the bottleneck analysis of the graph. Then, for the corresponding NoC, its width (bits) is scaled to make the broadcast faster based on the needed scaling. The new value is clamped to avoid exceeding the maximum width feasible for a one-shot broadcast.

$max\_width\_feasible = exec\_info[noc\_bytes\_per\_group][op] * 8$

$width\_scaled = noc\_width\_current * s$

$noc\_width\_new = min(width\_scaled, max\_width\_feasible)$

Similarly, total unicast links needed by the NoC for *op* are calculated from required concurrent accesses by PE groups.

$max\_links\_feasible = exec\_info[noc\_groups\_needed][op]$

$links\_scaled = noc\_unicast\_links\_current[op] * s$

$unicast\_links\_new[op] = min(links\_scaled, max\_links\_feasible)$

Whenever the number of PE-groups requiring different data elements exceeds the available unicast links (by $V\times$), the data is unicast with time-sharing (V times) over configurable NoC (as in Eyeriss [4]) to facilitate the mapping. Parameter *virtual_unicast_links* indicates time sharing over a unicast link, which can be set as number of time sharing instances (V).

• **Sizing RFs and Memory:** The total NoC communication time can be reduced by increasing the bottleneck operand (*op*)'s reuse in the RF (register file or local buffer) of the PEs. Increasing the reuse by R requires (R×) larger chunks of non-bottleneck operands, which need to be stored in the RF and communicated via other NoCs. Using the information about non-exploited (available) reuse of the bottleneck operand and the required scaling, the new RF size can be calculated as:

$target\_scaling = min(max\_reuse\_available\_RF[op], S)$

$RF\_size\_new = \sum_{op_i} \lceil exec\_info[data\_RF][op_i] *$

$target\_scaling \div reuse\_available\_RF[op_i]\rceil$

The calculation is similar for the global buffer (scratchpad memory), except for the targeted scaling. In off-chip data communication, multiple operands are communicated one by one via DMA (unlike simultaneously by NoCs per operand). So, the targeted scaling of the scratchpad depends on the bottleneck operand's (with remaining reuse) contribution (f) to the total off-chip footprint. The speedup/scaling achievable through exploiting reuse (A) can be approximated with the Amdahl's law as:

$A = (s * f) \div (1 - s + (s * f))$

$target\_scaling = min(max\_reuse\_available\_SPM[op], A)$

$SPM\_size\_new = \sum_{op_i} \lceil exec\_info[data\_SPM][op_i] *$

$target\_scaling \div reuse\_available\_SPM[op_i]\rceil$

Explainable-DSE workflow and bottleneck analysis and mitigation for DNN accelerators were implemented in python. It allows easy interfacing with the cost models for DNN accelerators. Since the implementation of the bottleneck analysis module and the bottleneck-guided DSE is external to the cost model, they could be extended to interface with other accelerator cost models like MAESTRO [121] that make the execution characteristics available (e.g., bandwidth, Ops, data packets to be communicated). §3.14 and §3.12 discuss such specification efforts for bottleneck models.

3.5.8 Tightly Coupled Hardware/Software Codesign Explorations

Efficient codesign requires optimizing both the hardware configurations and mappings in a coordinated manner. However, when using black-box DSEs, these configurations are typically explored in a loosely coupled manner. In other words, the acquired candidates usually do not address inefficiencies in the achieved execution with their co-optimization counterparts. For example, the acquired values of the off-chip/NoC bandwidth may be inefficient or incompatible with the selected loop tile configuration (in the same/previous trials in the mapping optimization), resulting in significantly higher communication time and total latency.

To address these inefficiencies, the DSE integrates mapping space optimizations and explores HW/SW codesign in a tightly coupled manner through bottleneck-guided exploration. The usage of bottleneck models allows reasoning about design inefficiencies for the objective optimized by a co-optimization counterpart. For example, the DSE considers software optimization as a subspace for iteratively optimizing hardware configurations. For a hardware configuration, when the DSE optimizes mappings through explorations or even a fixed schema, it mostly leads to efficient executions that can adapt to the tensor shapes and workload characteristics (reuse, batching, parallelism, etc.) for the selected hardware configuration. Then, the DSE uses bottleneck models that consist of both hardware and software/execution parameters. The DSE finds bottlenecks in the executions optimized by the mapping optimizer. Then, in the next attempt, the DSE acquires new hardware candidates such that they address bottlenecks in the executions optimized previously through software configurations. Once a new hardware design is updated as the current solution, software configurations are optimized again in tandem. Consequently, this approach leads to an efficient codesign for diverse tensor shapes and workload characteristics.

For efficient exploration of hardware/mapping codesign within practical budgets, DSE needs to explore quality mappings quickly. Proposed approach builds on previous research on mappers for DNN accelerators that eliminate infeasible and ineffective mappings by pruning loop tilings and orderings (detailed in §3.9, §3.10). For fast mapping optimizations, the proosed DSE has integrated and extended dMazeRunner [40], which can find near-optimal solutions within seconds. Mappers like dMazeRunner [40], Interstellar [177], or ZigZag [170] consider comprehensive space, optimally prune loop orderings, and prune tilings based on the utilization of architectural resources

(PEs, buffers, non-contiguous memory accesses). Then, they linearly explore the pruned space. However, one challenge with their fixed utilization thresholds for pruning is that it may lead to a search space that contains either too few mappings (e.g., tens) for some DNN layers or too many (many thousands) for others. To address this challenge, these search hyperparameters of dMazeRunner are adusted automatically to formulate the mapping search space that contains up to the top-N mappings based on utilization thresholds. N is the size of pruned mapping space formulated by adjusting thresholds for pruning the search space iteratively, which must be within a user-specified range, such as [10, 10000]. These mappings are then evaluated linearly, as in dMazeRunner [40] or Timeloop [47]. This approach helps achieve quality mappings by pruning ineffectual methods like in dMazeRunner/Interstellar, while also ensuring a reasonably large space of high-quality mappings as per specified exploration budget.

3.6 Experimental Methodology

• **Benchmarks:** Eleven deep learning models are evaluated for Computer Vision (CV) and Natural Langugage Processing (NLP) tasks [32]. CV models include ResNet18, MobileNetV2, and EfficientNetB0 [10] (light) and VGG16, ResNet50, and Vision Transformer [178] (large) for classifying ImageNet images. The light and large labels differentiate models based on inference latency and total computations. For object detection, recent models like FasterRCNN-MobileNetV3 [179] and YOLOv5 [180] (large) are evaluated. NLP models include Transformer for English-German sentence translation [13] and BERT-base-uncased [14] for Q&A on SQuAD dataset. Facebook wav2vec 2.0 [181] is also evaluated for automatic speech recognition (ASR). Their DNN layers are 18, 53, 82, 16, 54, 86, 79, 60, 163, 85, and 109 respectively. Models are obtained from PyTorch and Hugging Face [182].

• **Design space:** Table 3.1 lists the design space of a DNN accelerator for inference

Table 3.1: Design space for edge DNN accelerators.

Data: int16; Freq. 500 MHz; Constraints: Throughput>=40/10 FPS (vision light/large), 120/530/176k samples/second (NLP: Transformer/BERT/ wav2vec2); Area < 75 mm^2; Max. power < 4W. Objective: Minimize latency.

Parameter	**Values**	**Options**
PEs	64, 128, ..., 4096	7
L1 buffer (B)	8, 16, ..., 1024	8
L2 buffer (kB)	64, 128, ..., 4096	7
Offchip bandwidth (MBPS)	1024, 2048, 4096, 6400, 8192, 12800, 19200, 25600, 38400, 51200	10
NOC datawidth	16*i; i: [1, 16]	16
Physical unicast (×4)	PEs*i / 64; i: [1, 64]	64^4
Virtual unicast (×4)	2^{3i}; i: [0, 3]	4^4

at the edge. Like existing accelerators, four dedicated NoCs are considered for a total of four read/write operands [33]. The number of links for concurrent or time-shared unicasting is per each NoC. To limit the design space for related techniques, the number of unicast links are expressed as a fraction of total PEs. Execution constraints are selected based on the requirements for ML benchmarks [32] and designs of industrial edge accelerators for ML inference, e.g., [183, 184]. The objective is set as minimizing the latency of the single-stream execution [32].

• **DSE techniques:** Explainable-DSE is evaluated against previous accelerator DSE frameworks using constrained optimizations - Hypermapper 2.0 [2] and Confuciux [65] for reinforcement learning (RL). Confuciux limits the total parameters to two, works with a single constraint, and requires the same number of values for all parameters. So, its implementation is generalized for evaluations. Proposed approach is also evaluated against non-feedback or black-box approaches like Grid search, Random search, Simulated annealing (Scipy [185]), Genetic algorithm (Scikit-Opt [186]), and Bayesian optimization [187]. All techniques are evaluated on a Dell precision 5820 tower workstation. Like previous DNN accelerator DSEs, the cost models [40, 89] were used. The system for evaluating the candidates with cost models was the same for all techniques.

• **Mapping optimizations and codesign explorations:** Prior works mostly used a fixed dataflow, such that exploration time is primarily spent on optimizing hardware configurations, while getting efficient mappings with fixed schema. So, first the mapping technique is fixed as an optimized output stationary dataflow (SOC-MOP) [45] for all approaches. Then, the codesign with Explainable-DSE is demonstrated by a tightly coupled optimization of both the hardware and mapping configurations. Obtained codesigns are also compared with those obtained by black-box approaches. Black-box codesign DSE explores hardware configurations with two techniques that were found effective: random search and HyperMapper 2.0. §3.10 details setup for an effective black-box exploration of mappings in a comprehensive yet highly pruned space of feasible/effectual mappings. For mapping each DNN layer on every hardware configuration, black-box DSE evaluations use Timeloop-like random search for 10,000 mapping trials, as it was found effective in quickly obtaining high-quality mappings (§3.10).

• **Exploration budget:** A total of 2500 iterations are considered for statically finding the best solutions. Dynamic DSE capabilities are also analyzed by explorations in the total 100 iterations.

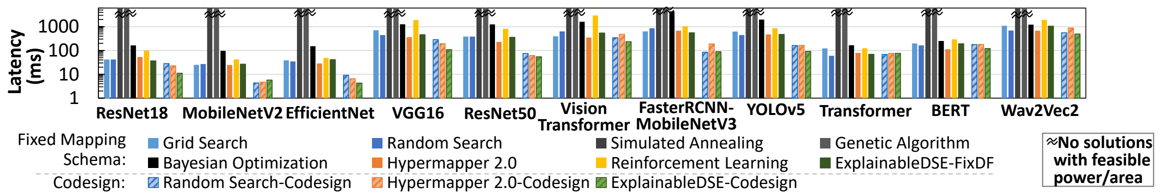


Figure 3.10: Explainable-DSE obtained codesigns of 6× lower latency.

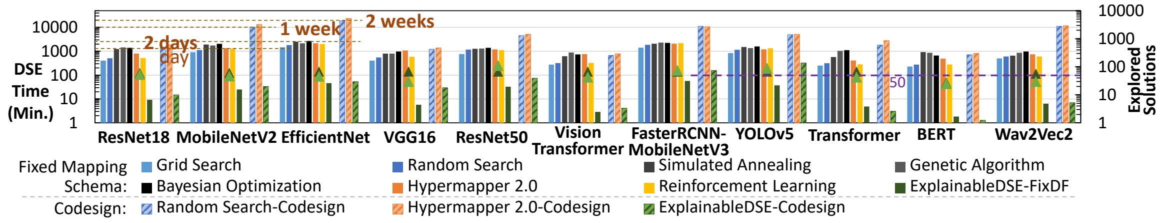


Figure 3.11: Explainable-DSE with fixed dataflow and codesigns reduce search time by 53× and 103× (minutes vs. days–weeks).

3.7 Results and Analysis

3.7.1 Explainable-DSE Obtained Codesigns of 6× Low Latency in 47× Less Iterations

Fig. 3.10 illustrates the latency obtained by different techniques for static exploration. By exploring among quality solutions, Explainable-DSE obtained 6× more efficient solutions, on average, as compared to previous approaches, and up to 9.6× over random search and 49.3× over Bayesian optimization. Even when dataflow (schema for optimized mappings) was fixed for all techniques, it obtained 1.77× lower latency on average and up to 7.89×. By applying bottleneck analysis on workload executions at every acquisition attempt, Explainable-DSE could determine parameters critical for improving efficiency. Thus, it can effectively reach high-reward subspaces among the vast space. Fig. 3.12 illustrates this with latency reduction obtained over iterations by taking examples of two models, EfficientNet for CV and Transformer for NLP.

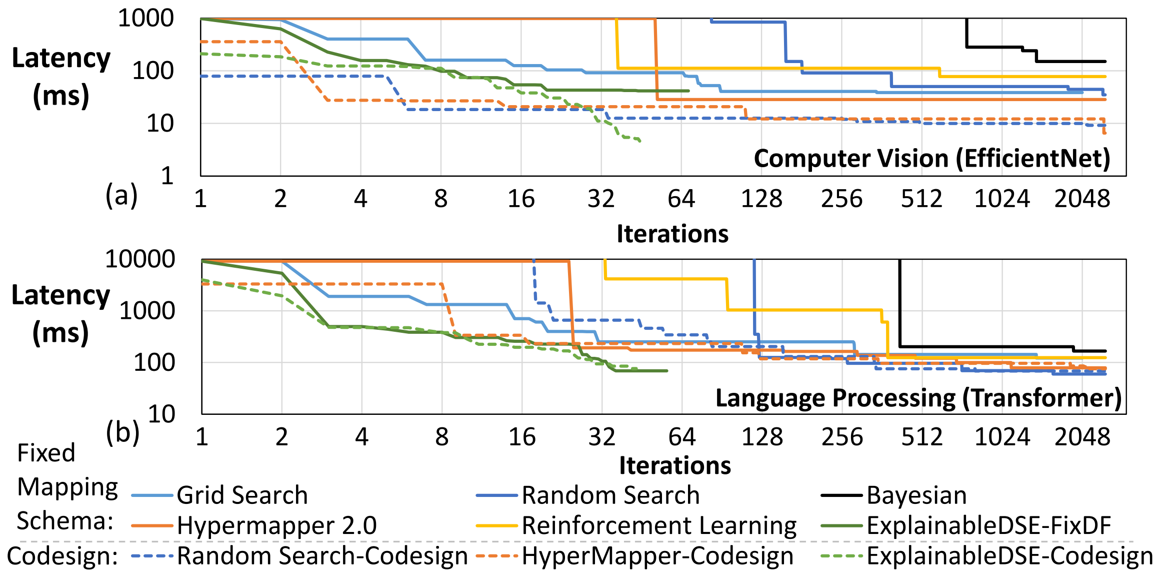


Figure 3.12: Latency reduced over iterations for (a) EfficientNet; (b) Transformer.

With objective reduction at almost every attempt, the Explainable-DSE converges to quality solutions early on (some tens of iterations) and usually of better efficiency. For instance, obtained solutions have 6.6×-35.1× lower latency for EfficientNet, as compared to the DSEs with fixed dataflow and 2.1×-9.7× as compared to black-box co-optimizations. Overall, at every attempt, it reduced the values of objective for feasible acquisitions by geomean 1.30× and 1.32× for fixed and co-explored mappings (as shown in Table 3.2). Acquisitions by non-explainable techniques, being bottlenecks-unaware, do not focus much on defacto promising subspaces. In fact, in some of the evaluations for Bayesian optimization, random search, and constrained RL, the reduction in the objective throughout the DSE iterations was negative (Table 3.2). They acquired candidates without understanding bottlenecks, out of which many were feasible but corresponded to lower efficiencies than the previously encountered best solutions.

Fig. 3.11 shows the total time (bars) taken by DSE techniques. Through constraints accommodation and systematically mitigating bottlenecks in multi-functional workload

Table 3.2: At every acquisition attempt, Explainable-DSE reduces objective by 30% vs. ∼1.4% by non-explainable techniques.

N/A is indicated when a technique could not find a single feasible hardware solution.

DSE Technique	ResNet18	MobileNetv2	EfficientNet	VGG16	ResNet50	Vision Transformer	FasterRCNN-MobileNetv3	YOLOv5	Transformer	BERT	Wav2Vec2	Average
Grid Search-FixDF	1.71%	1.03%	1.07%	1.21%	1.25%	1.41%	0.71%	0.55%	0.98%	1.04%	1.07%	1.09%
Random Search-FixDF	0.52%	-0.87%	7.34%	-2.26%	4.69%	-4.29%	-1.41%	-0.90%	0.01%	0.97%	-1.45%	0.21%
Simulated Annealing-FixDF	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Genetic Algorithm-FixDF	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Bayesian Optimization-FixDF	11.26%	26.57%	19.57%	19.22%	-1.09%	-4.89%	-10.01%	-12.28%	10.15%	-0.27%	11.33%	6.32%
HyperMapper 2.0-FixDF	5.32%	1.21%	0.44%	2.67%	4.94%	4.86%	-0.20%	0.87%	1.40%	3.35%	1.18%	2.37%
Reinforcement Learning-FixDF	-0.75%	-4.13%	5.18%	-2.51%	-2.97%	-10.47%	0.67%	1.66%	4.62%	0.50%	-0.50%	-0.79%
Random Search-Codesign	-0.07%	-0.29%	0.14%	0.44%	0.33%	0.57%	0.23%	0.91%	0.48%	-0.24%	0.02%	0.23%
HyperMapper 2.0-Codesign	0.56%	0.46%	0.59%	0.64%	0.68%	0.68%	0.72%	0.76%	0.43%	0.52%	0.73%	0.62%
ExplainableDSE-FixDF	53.54%	21.92%	20.48%	52.42%	15.32%	31.74%	23.73%	21.54%	30.96%	40.66%	21.44%	30.34%
ExplainableDSE-Codesign	30.50%	23.45%	32.10%	32.03%	18.77%	46.29%	27.03%	18.78%	26.19%	47.30%	46.70%	31.74%

executions, explorations quickly converged or terminated while achieving even more efficient solutions. For example, Explainable-DSE with fixed and optimized mappings explored about only 59 and 54 designs, respectively (shown by triangles; ∼2500 for other techniques). It led to search time reduction of 53× and 103× on average over black-box explorations, when using fixed dataflow for all techniques and hardware/mapping co-optimization, respectively. Maximum reduction in the search time was up to 501× and 1675×, respectively. Using modest information on mitigating bottlenecks, explainable DSEs consumed only 21 and 64 minutes, on average. In fact, they achieved the most efficient solutions for BERT under just two minutes!

3.7.2 Including Software Design Space in the Exploration Enables 4.24× Better Solutions

With the availability of exploration budget (by a drastic reduction in the search time), hardware/software codesigns can truly be enabled by optimizing both of them in a tightly coupled manner. Codesigns obtained with Explainable-DSE reduced objective by 4.24× on average as compared to using a single optimized mapping per DNN operator. The higher efficiency emanates from achieving better mappings tailored for processing various DNN layers (different functionality and tensor shapes of DNN operators) on the selected hardware configuration. They leverage higher spatial parallelism and more effectively hide data communication latency behind computations as compared to a pre-set dataflow. Further, mapping optimizations reduce the objective considerably without necessarily increasing hardware resources. Thus, by having a more constraints-budget on hand, the DSE was able to reduce the objective further (also evident in Fig. 3.12a).

For exploring comprehensively defined vast space of architectural configurations with non-explainable DSEs, presetting dataflow can lead to many infeasible solutions (§3.7.3). Note that infeasible solutions are not just hardware configurations with exceeding constraints like area or power. The designs can also be infeasible when a generated hardware configuration is incompatible with the used software, i.e., dataflow for mapping. For instance, in configurations generated by non-explainable DSEs, the total number of links for time-shared unicast was often lower than that needed by spatial parallelism in the dataflow used for mapping. That is exactly why a codesign or joint exploration with the software is important.

Black-box co-optimizations incorporated mapping explorations and reduced latency of obtained solutions further by 2.33× for HyperMapper 2.0 and 2.63× for random search, as compared to their DSEs using a fixed schema for optimized mappings. It is primarily because of the availability of more constraints-budget at hand, as discussed before. The co-optimizations also alleviated aforementioned challenge of mapping-hardware incompatibility. As Fig. 3.13 shows, with software optimization in the loop, the black-box co-optimizations find more feasible designs than black-box DSEs using a fixed mapping schema, when allowed to explore hardware design configurations for the same number of trials. However, even after 2500 trials for exploring hardware configurations and 10,000 trials for exploring mappings of each DNN layer on every hardware configuration, the latency of codesigns obtained by black-box approaches are still 1.6× higher than the codesigns obtained by Explainable-DSE, while consuming 103× more search time (taking 7-16 days for four workloads). Key reasons for such effective explorations by Explainable-DSE include generating fewer yet objective-reducing trials and tightly coupled codesigns. As Explainable-DSE leverages the domain knowledge, its generated designs continually address arising execution inefficiencies, converging in 47× less iterations. In black-box co-optimizations, the DSE is loosely coupled, as the generated hardware configuration is not necessarily tailored to work best with the optimized mappings (from the previous/same trial for the hardware DSE). In contrast, tightly coupled codesign exploration in Explainable-DSE finds hardware configurations that alleviate inefficiencies in the workload executions optimized previously by mappings; Once new hardware configuration is generated, mapping exploration strives to utilize hardware resources effectively, lowering costs further. And, this repeats. Thus, optimizations for both hardware and software configurations strive to reduce inefficiencies in the execution optimized by their counterpart.

Although optimizing the mappings for every hardware design requires additional search time, the overall increase for exploring codesigns with Explainable-DSE was only 3× on average (from 21 minutes to 64). In fact, for all except large object detection models, the DSE time increased from 16 minutes to only 26 minutes. One reason is that the mappings can be quickly evaluated with analytical performance models (e.g., a minute each for several hundred to a few thousand mappings) and concurrent execution with multiple threads [40] (subjected to execution on four cores at maximum in the evaluations). Moreover, applying bottleneck analysis on efficient mappings helped obtain efficient designs faster (1.1× lower iterations for hardware

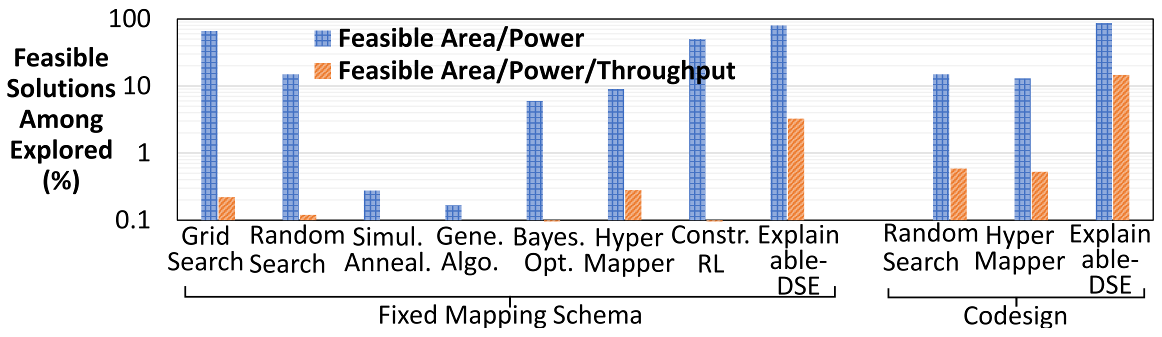


Figure 3.13: Most acquisitions by Explainable-DSE met area and power constraints, as compared to non-explainable techniques. Solutions obtained by all but Explainable-DSE mostly did not meet strict throughput requirements.

designs on average, and up to 1.9×). Whenever the DSE for codesigns evaluated a similar number of architecture designs as proposed DSE with fixed dataflow, it went on to explore even more efficient solutions (e.g., 2.33× lower latency for Vision Transformer).

3.7.3 By Considering Utilization of Constraints, DSE Mostly Acquires Feasible Solutions Without Exhausting Constraints Quickly

Non-explainable black-box optimization approaches, e.g., with Genetic Algorithm or Bayesian Optimization, did not know which configurations could likely lead to feasible subspaces. Therefore, even after exploring over days, they almost did not obtain a single feasible solution. When considering only area and power constraints, feasibility of the explored solutions was higher for mostly all techniques (Fig. 3.13), e.g., 15% for random search and 50% for constraints-aware reinforcement learning. However, when considering throughput requirement for DNN inference, the feasibility of the explored solutions was barely ∼0.1%–0.3%. By exploring mappings, the black-box codesign optimizations addressed the challenge of mappings being incompatible for the obtained hardware configurations. Thus, they improved feasibility by 2×-5×,

Table 3.3: Latency minimized by DSE techniques in 100 iterations. Explainable-DSE evaluated ∼54 solutions. Designs obtained by Non-Explainable DSEs were *low-throughput* (shaded values) and *incompatible* with used dataflow (dashes). More importantly, * denotes that none of the obtained candidates by a non-explainable DSE met even area/power constraints.

DSE Technique	ResNet18	MobileNetv2	EfficientNet	VGG16	ResNet50	Vision Transformer	FasterRCNN-MobileNetv3	YOLOv5	Transformer	BERT	Wav2Vec2
Grid Search-FixDF	278	73.4	92.0	3650	747	1973	1625	1477	251	780	1933
Random Search-FixDF	-*	197	694	41912	626	1376	3152	7754	157	1044	2357
Simulated Annealing-FixDF	-*	-*	-*	-*	-*	-*	-*	-*	-*	-*	-*
Genetic Algorithm-FixDF	-*	-*	-*	-	-*	-	-	-*	-*	-*	-*
Bayesian Optimization-FixDF	-	-	-	-	-	-	-	-	-	-	-
HyperMapper 2.0-FixDF	53.3	46.5	135	1339	493	1308	13582	1142	171	663	912
Reinforcement Learning-FixDF	-	-	360	-	-	-	21150	18082	143	1428	1428
Random Search-Codesign	69.6	12.7	9.5	870	209	857	224	218	244	240	1427
HyperMapper 2.0-Codesign	63.1	5.1	10.3	1233	87.3	1084	830	348	133	637	1945
ExplainableDSE-Codesign	11.2	5.7	4.3	109	54.9	233	89.2	92.1	76.2	121	494

but the overall feasibility was still ∼0.6%. Such low feasibility for DSE in humongous space is presumably caused by not accommodating constraints during exploration and bottlenecks-unaware acquisition trials. Contrarily, Explainable-DSE prioritized to meet the constraints for its acquisitions and update of the new solutions, which helped avoid infeasible subspaces. Plus, addressing bottlenecks in executions helped acquiring high-performance solutions. Hence, 87% and 15% of solutions explored by Explainable-DSE codesigns were feasible when considering area and power constraints and all the three constraints, respectively. For DNNs like BERT and MobileNetV2, 89%–98% of the explored solutions met area and power constraints. Once Explainable-DSE achieved a solution that met all constraints, it always ensured to optimize further with a feasible solution.

3.7.4 Enabling Efficient Dynamic Exploration in the Vast Space

Table 3.3 shows latency of solutions achieved in 100 iterations by different techniques. Under a short exploration budget, non-explainable techniques did not find a feasible solution (shaded values). Even after ignoring intense throughput requirements, most techniques could not find feasible solutions. Contrarily, by exploring spaces where candidates utilize low budget of constraints, Explainable-DSE quickly landed feasible solutions. Black-box approaches explored feasible codesigns, but they did not meet throughput requirements. On the other hand, by addressing the bottlenecks in multi-functional executions, Explainable-DSE achieved solutions of one to two orders of magnitude lower latency over other techniques.

3.8 Case Study: Efficiency of the Designs Achieved by DSE

Methodology: This study compares the efficiency of designs obtained by proposed DSE (§3.7; Fig. 3.10) to those of edge AI accelerators. The DSE can only explore designs for the spatial architecture templates used by the cost models (e.g., in [40, 47, 89, 121]). Therefore, to make a fair comparison, only those edge accelerators with similar architectural features are considered. Specifically, two edge accelerators are compared against: 1) Coral Edge TPU [3], a state-of-the-art industrial edge accelerator platform developed by Google, and 2) Eyeriss [4], an efficient edge accelerator that incorporates several special optimizations for high energy efficiency and low latency.

While novel edge accelerators have been developed recently, they typically contain several (micro)architectural features for specialization (e.g., mixed-precision computations or sparsity exploiting features [33]) that are not modeled (accurately) by the commonly used/available latency models for edge AI accelerators. The architectures considered here are suitable for the comparison, as they are commonly used as a template in the system-level tools for design and execution modeling [40, 47, 65, 103, 121, 177]. Table 3.4 compares the architectural features and execution optimizations for the Edge TPU, Eyeriss, and the template architecture used by the DSE cost models. The results for Edge TPU were obtained from the performance

Table 3.4: Comparison of architectural features and execution optimization for edge AI acceleration.

Feature	**Edge TPU**	**Eyeriss**	**DSE**
Data Precision	8b*	16b	16b
Technology	-	65nm	45nm
PEs	Scalar MAC	Scalar MAC	Scalar MAC
Temporal Reuse	Yes	Yes	Yes
Spatial Reuse	Data Distribution and Reduction	Data Distribution and Reduction	Data Distribution
Mapping Optimizations	Automatic (TFLite)	Fixed-Style	Automatic
Hardware-Mapping Co-optimization	No	Yes	Yes
Accelerator-DNN Codesign	Yes†	No	No
Frequency (MHz)	125-500	100-250	500

∗ Edge TPU results are scaled to match 16b precision used in comparison.

† For Edge TPU, the EfficientNet-EdgeTPU model is codesigned.

¶ In lack of information about the actual power consumed by edge TPU for different models, 1.4W power is considered (as reported for MobileNetV2 in the edge TPU datasheet).

benchmarking of TPU-optimized models [188], and the results for Eyeriss chip were obtained from its evaluations [4].

Results: Fig. 3.14 shows performance achieved by all three designs and the resultant energy efficiency and area efficiency. The results demonstrate that, on average, codesigns obtained by the DSE attain 3.7× higher throughput than the Edge TPU. It is due to the DSE's ability to analyze execution bottlenecks and optimize both mappings and hardware configurations, such as NoC configurations/bandwidth and buffer sizes. The DSE-achieved designs also required more than an order of magnitude less on-chip area, presumably due to allocating significantly smaller buffers and fewer MAC units (e.g., considering Edge TPU configurations studied in [189]). The overall area efficiency is higher by about 14× for MobileNetV2 and 49× on average due to a high-throughput execution for VGG-16. Although the DSE was focused on minimizing latency, energy efficiency of its designs matches that of the EfficientNet-Edge TPU codesign, and is even higher by 2.9× for VGG-16.

When compared to Eyeriss, the DSE achieves designs of lower latency (with similar area/power budgets), and improves area efficiency by up to 11.84× and energy efficiency of VGG-16 by up to 1.33×—while not incorporating additional efficiency-gaining features as in Eyeriss. For instance, Eyeriss leverages frequency scaling, power gating, zero skipping, and compression (for up to 86% sparsity in AlexNet layers). Thus, Eyeriss achieves about 2×–3.75× higher energy efficiency when compared to the designs obtained by DSE that minimized latency. In most cases, the codesigns obtained by DSE in a tightly coupled manner outperform the Eyeriss-like designs. These comparisons demonstrate the potential of the proposed capabilities for the accelerator system design. With the continued development of automated execution modeling and inclusion of more template architectures and specialization components, proposed methodology can be expected to enhance efficiency further.

3.9 Additional Related Works

This section discusses additional related work beyond the background on DNN accelerator DSE techniques described in §3.2.2, their limitations in §3.3, and previous DSEs using bottleneck analysis for different domains in §3.4.

• **Execution cost models of DNN accelerators:** The cost models of SECDA [160] and TVM/VTA [124] support end-to-end simulation and synthesis, while faster

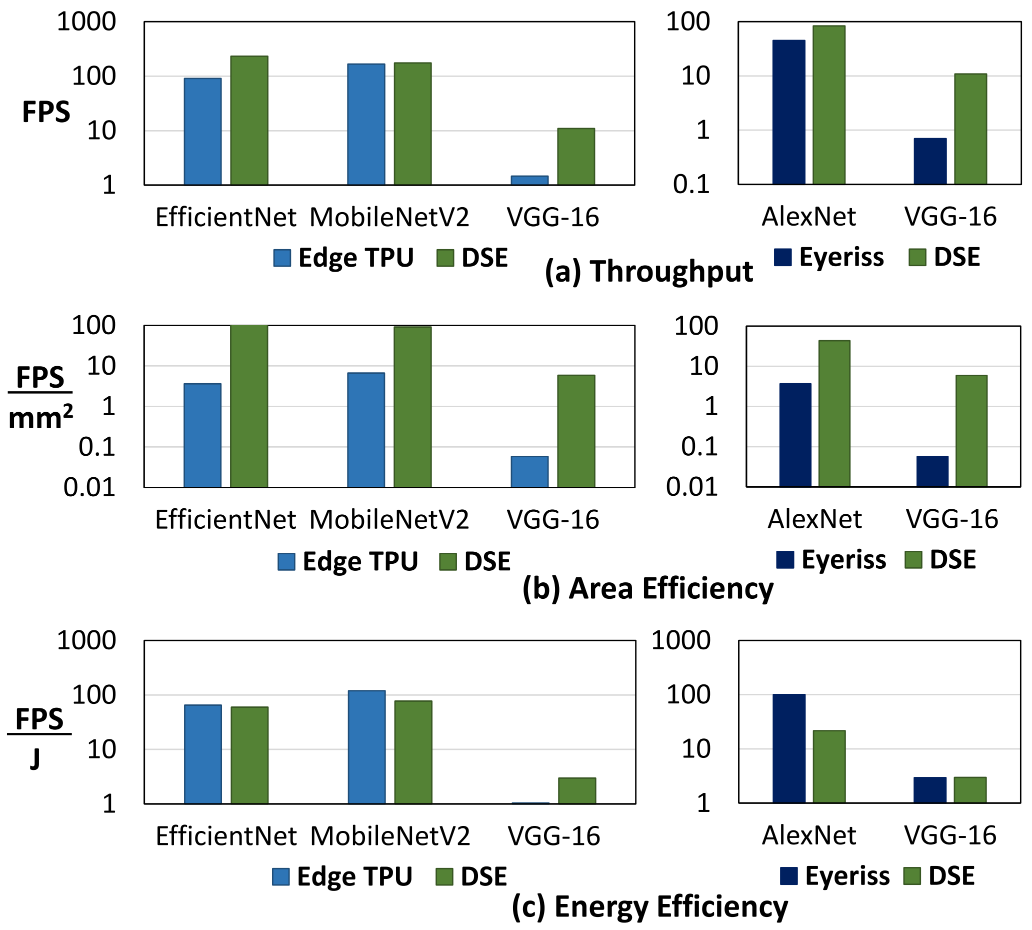


Figure 3.14: Comparison of efficiency of the designs obtained by the DSE with that of the edge accelerators Google Coral Edge TPU [3] and Eyeriss [4]: (a) Throughput (frames per second i.e., FPS) [3.7×, 8.7×], (b) Area efficiency (FPS/mm^2) [49×, 57×], and (c) Energy efficiency (FPS/Joule) [1.5×, 0.6×].

analytical models are more commonly used to optimize mappings and accelerator design configurations. Their examples include MAESTRO [121], Accelergy [89], SCALE-Sim [46], and those of Timeloop [47], dMazeRunner [40], and Interstellar [177] infrastructures. Most of these models estimate both latency/throughput and energy. In addition to computational cycles, MAESTRO, dMazeRunner, and Timeloop account for on-chip and off-chip communication latency. Table 3.5 compares their execution modeling features. For the DSE, the cost model of dMazeRunner infrastructure was used, which also considers the performance overheads of non-contiguous memory accesses and allows explicit specification of NoC bandwidths and flexibly specifying mappings through loop nest configurations.

• **Mappers for DNN accelerators:** Mappers typically target the space of all valid loop tilings and orderings. For tensor shapes of a layer, there can be many factors of loop iteration counts, and just populating the space of valid mappings could be time-consuming (*micro*seconds–several seconds) [190]. Table 3.6 compares different mappers. Timeloop [47], a commonly used mapper, explores mappings through random sampling, while GAMMA [191] uses a genetic algorithm. However, GAMMA limits the number of loops that can be executed spatially and does not prune invalid tilings before exploration, requiring several-fold more trials for convergence [192]. Without eliminating ineffectual loop tilings and orderings beforehand, black-box explorations typically require thousands of trials, generating many invalid mappings, and take hours to map a single DNN layer once [193]. Mind Mappings [103] reduces the search time by training a surrogate model that estimates costs faster than analytical models. CoSA [193] uses a prime factorization-based approach to construct the tiling space for a mixed-integer programming solver. But, many tilings corresponding to combinations of prime factors remain unexplored, potentially resulting in sub-optimal solutions. Additionally, most mappers do not support depthwise-convolutions, invoking convolutions channel-by-channel. So, they miss opportunities for exploiting parallelism across multiple channels and reducing miss penalties for accessing contiguous data of consecutive channels from the off-chip memory.

Interstellar [177] prunes ineffectual tilings by constraining the search to pre-set resource utilization thresholds. dMazeRunner [40] goes further and prunes loop orderings for unique/maximum reuse of operands and proposes heuristics that reduce the space to highly efficient mappings, which can be explored in second(s). Hence, proposed dMazeRunner infrastructure is utilized in the codesign and it was extended to construct the space of up to top-N mappings, where N is the maximum mapping trials allowed. ZigZag [170] and follow-up mappers build upon such pruning strategies. ZigZag allows uneven blockings of loops for processing different tensors, which may partially improve efficiency. However, ZigZag's search time for a DNN layer is nearly hours [170]. While works such as [170, 194–197] optimize DNN mappings on one or more hardware accelerators, they require exploring hardware parameters exhaustively or with black-box optimizations.

• **Hardware/software codesign explorations of DNN accelerators:** Previous DNN accelerator DSEs, such as [41, 51, 61, 62, 69], used black-box optimizations. They incur excessive trials and ineffectual solutions, as they lack reasoning about the higher costs of obtained candidates and the potential efficiency of candidates to be acquired next (§3.3). Further, DSEs of [61, 65–67, 70–72] used a fixed dataflow in explorations. It obviates increasing search time further but may not lead to the most efficient solutions compared to codesigns.

Recent approaches HASCO [62] and DiGamma [198] optimize both hardware and mapping configurations in a black-box manner, encountering the same challenges of ineffectual and excessive trials due to non-explainability (§3.3). Secondly, with a loosely coupled codesign exploration (§3.5.8), they acquire HW/SW configurations that may not be effective or suitable for the counterpart. Furthermore, they target a limited hardware design space comprising only buffers and PEs. Finally, they typically do not explore a single accelerator design that addresses inefficiencies in executing DNNs with many layers.

• **DSE using bottleneck analysis:** DSEs of [157, 158] use bottleneck analysis, but they are unaware of constraints utilization and optimize only a single loop-kernel.

Plus, they explored only neighboring values of parameters (instead of scaling them to mitigate bottleneck in one shot). It leads to search time comparable to black-box DSEs [157]. AutoDSE [159] and SECDA [160] proposed bottleneck analysis specific to FPGA-based HLS, and their search optimizes a single loop-kernel/task of a single workload at a time. This work proposes using bottleneck models for DNN accelerator designs; proposed DSE framework generalizes prior DSEs to the case of multiple loop-nests, multi-modal workloads, and multiple workloads through aggregation of various bottleneck mitigation (for new acquisitions of promising designs). Further, via proposed API and data structures, proposed framework decouples bottleneck models from search algorithms, allowing designers to systematically express the bottleneck models for their domain-specific architectures/systems and interface with the bottleneck-guided, explainable DSE.

3.10 Constructing Effectual Mapping Space and Black-Box Mappers

Background: The process of optimizing mappings of a DNN layer on an accelerator design involves exploring an enormous search space, as demonstrated by previous works [86, 191] and later summarized in Table 3.7. This search space is primarily composed of configurations for loop tile sizings and loop orderings. Loop tilings determine the data bursts that need to be accessed via memory hierarchy and spatial parallelism, whereas loop orderings determine the data reuse and impact the total memory accesses. For mapping DNN operators such as convolutions and multi-layer perceptrons on an accelerator with a 3-level buffer/memory hierarchy and 1-level spatial parallelism [4], the mapping space corresponds to 28-deep and 12-deep nested loops, respectively [40]. Tile sizings are configurations that represent the values for loop iterations at each temporal/spatial level in the architectural processing hierarchy. For a selected tiling configuration, processing of a 7-deep nested loop at a buffer level

Table 3.5: Execution cost models for deep learning accelerators.

Features	**Timeloop** [47]	**dMazeRunner** [40]	**MAESTRO** [121]	**Interstellar** [177]	**SCALE-Sim** [46]	**Accelergy** [89]
Performance Estimate	Yes	Yes	Yes	No	Yes	No
Energy Estimate	Yes	Yes	Yes	Yes	No	Yes
Integrated Support for ML Libraries	No	Yes	Yes	No	No	N/A
Layers Supported	GEMM, CONV	GEMM, CONV, DWCONV	GEMM, CONV, DWCONV	GEMM, CONV	GEMM, CONV	N/A
Data Precision	Variable	Variable	Variable	Variable	Fixed	Variable
Mapping Specification	Loop Nest Configuration	Loop Nest Configuration	Directives	Loop Nest Configuration	N/A	N/A
Dataflow	All	All	All	All	WS, OS, IS	N/A
Spatial and Temporal Data Reuse	Yes	Yes	Yes	Yes	Yes	N/A
Data Reuse with Striding Convolution	Yes	No	Yes	N/A	N/A	N/A
Considers On-Chip Communication Latency	Yes	Yes	Yes	No	N/A	N/A
Memory Hierarchy	Arbitrary	3-level	3-level	3/4-level	Fixed	Arbitrary
Models overhead of non-contiguous accesses	No	Yes	No	No	No	No

corresponds to 7! different loop orderings. Table 3.7 lists the DNNs and their layers with the largest search space, which can contain up to $O(10^{36})$ configurations.

Pruning invalid loop tilings: To optimize the tile sizes of a DNN layer using a black-box optimizer, designers often set the index variables of tiled loops as design parameters and specify lower/upper bounds (loop iterations) to define the range of values that the optimizer can explore [103, 191]. However, the search space for tile sizes can be enormous, containing as many as $O(10^{28})$ configurations for certain DNN layers, as shown in Table 3.7. A black-box optimizer is unlikely to find more than a few valid mappings under practical exploration budgets for exploring such a large search space [193]. This is because, for each loop in a DNN operator, only a small subset of factors of loop iterations lead to valid tile sizes. For example, a set (8, 4,

Table 3.6: Mappers for deep learning accelerators.

Features	**Timeloop** [47]	**Gamma** [191]	**Mind Mappings** [103]	**CoSA** [193]	**Interstellar** [177]	**ZigZag** [170]	**dMazeRunner** [40]
Comprehensive Mapping Space	Yes	No	Yes	No	Yes	Yes	Yes
Discard Invalid Mappings for Mapping-Space Construction	Yes	No	No	Yes	Yes	Yes	Yes
Prune Inefficient Tilings	No	No	No	No	No	Yes	Yes
Prune Inefficient Orderings	No	No	No	No	Fixed	Yes	Yes
Layers Supported	CONV, GEMM	CONV, GEMM, DWCONV	CONV, MTTKRP	CONV, GEMM	CONV, GEMM	CONV, GEMM	CONV, GEMM, DWCONV
Exploration Approach	Random	Genetic Algorithm	Gradient Descent	Mixed Integer Programming	Heuristic	Heuristic	Heuristic
Uneven Tiling for Tensors	No	No	No	No	No	Yes	No
Search Time	Minutes	Minutes	Minutes	Seconds	Minutes	Hours	Seconds
Off-line Training	No	No	Yes	No	No	No	No

2, 16) makes a valid 4-level tiling configuration for 1024 loop iterations over output activations/filters, while many more do not. Therefore, designers formulate the space of valid tiling sizes based on factorization of loop iterations [47, 193]. As shown in Table 3.7, this prunes the search space of tiling configurations by a square root or even a cube root, e.g., from $O(10^{22})$–$O(10^{28})$ configurations to a much smaller range of $O(10^{9})$–$O(10^{14})$ configurations. Invalid tiling configurations, which require more architectural resources than are available in the target hardware configuration, are usually discarded by black-box optimizers during the exploration (as indicated in column C of Table 3.7).

Table 3.7: Analyzing size of the mapping space for deep learning accelerators.

Model	**Layer**	**Tile Sizings**	**Tile Sizings with Valid Factors**	**Valid Tilings w.r.t. Hardware**	**Orderings at a Memory Level**	**Orderings with Unique/Max Data Reuse**	**Full Map. Space**	**Factorization-Constrained Mapping Space**	**Factorization-Constrained Reuse-Aware Map. Space**
		A	B	C	D	E	F: A*D^2	G: B*D^2	H: B*E^2
ResNet18	CONV_2_1a	$O(10^{25})$	$O(10^{13})$	$O(10^7)$	$O(10^4)$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
MobileNetV2	features.2.conv.0	$O(10^{22})$	$O(10^{12})$	$O(10^6)$	$O(10^4)$	15/3	$O(10^{30})$	$O(10^{19})$	$O(10^{13})$
EfficientNetB0	blks.2.expand	$O(10^{22})$	$O(10^{12})$	$O(10^6)$	$O(10^4)$	15/3	$O(10^{29})$	$O(10^{20})$	$O(10^{13})$
VGG-16	CONV_1_2	$O(10^{28})$	$O(10^{14})$	$O(10^7)$	$O(10^4)$	15/3	$O(10^{36})$	$O(10^{21})$	$O(10^{15})$
ResNet50	CONV_2_1b	$O(10^{25})$	$O(10^{13})$	$O(10^7)$	$O(10^4)$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
Vision Transformer	patchembeddings. CONV2D	$O(10^{25})$	$O(10^{13})$	$O(10^6)$	$O(10^4)$	15/3	$O(10^{32})$	$O(10^{20})$	$O(10^{14})$
FasterRCNN-MobileNetV3	features.12. conv2.excite	$O(10^{26})$	$O(10^{13})$	$O(10^6)$	$O(10^4)$	15/3	$O(10^{33})$	$O(10^{20})$	$O(10^{14})$
YOLOv5	features.1.conv	$O(10^{27})$	$O(10^{14})$	$O(10^7)$	$O(10^4)$	15/3	$O(10^{34})$	$O(10^{21})$	$O(10^{15})$
Transformer	decoder. output_projection	$O(10^{27})$	$O(10^9)$	$O(10^4)$	$O(10^1)$	3/3	$O(10^{28})$	$O(10^{10})$	$O(10^{10})$
BERT	encoder.layer.0. output.dense	$O(10^{26})$	$O(10^9)$	$O(10^5)$	$O(10^1)$	3/3	$O(10^{27})$	$O(10^{11})$	$O(10^{10})$
Wav2Vec2	encoder.layers.0. intermediate.dense	$O(10^{28})$	$O(10^{12})$	$O(10^6)$	$O(10^1)$	3/3	$O(10^{29})$	$O(10^{13})$	$O(10^{12})$

Pruning ineffectual loop orderings: The space is reduced further by discarding ineffective loop orderings. Previous black-box mappers explored all orderings, resulting in a large number of configurations (7! or $O(10^4)$) for processing a convolution at a memory level in the memory hierarchy. Instead, proposed approach builds upon previous research that has shown only a handful of loop-orderings having unique data reuse of tensors (15 for convolutions) and a few with maximum reuse of various tensors [40, 170].

Overall mapping space: The GAMMA-like mapper considers full (non-factorized) tiling space and all loop-orderings [191] (column F), while Timeloop [47] and CoSA [193] consider factorized tile sizes but all loop orderings. For evaluations with black-box mappers, factorized tile sizes (all valid factorization) were used and only loop orderings with unique data reuse were considered. In practice, there are only a few unique data reuse scenarios (vs. 15/3) for each tiling configuration, so all of them can be explored

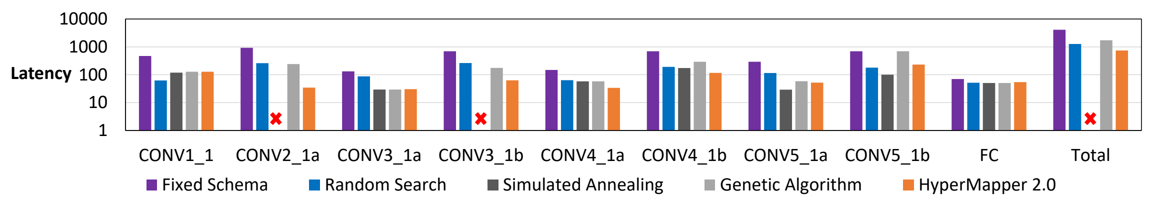


Figure 3.15: Efficiency of mappings obtained by different black-box optimizations for accelerating ResNet layers.

linearly [40]. Therefore, black-box optimizations were invoked with 10,000 trials for mapping each layer on a hardware configuration. During each trial, the optimization acquired a tiling configuration and evaluated all effectual loop-orderings through the cost model, selecting the one that minimized the objective. Thus, the mapping space formulation discarded a large number of invalid and ineffective configurations (column H) without compromising the optimality.

Selection of the mapping optimization technique: Evaluations comparing black-box DSEs of hardware configurations with fixed mapping schema showed that random search and Bayesian optimization-based HyperMapper 2.0 [2] were the most effective, as depicted in Figure 3.10. As a result, these two techniques were selected for optimizing both the hardware and mapping configurations. However, when they were applied to optimize mappings from the pruned space (column H in Table 3.7), it was found that random search obtained efficient mappings within several seconds. By contrast, HyperMapper 2.0 generated efficient mappings, but its search overhead was prohibitively high, requiring a few hours to evaluate just a single DNN layer. Consequently, random search was used for optimizing mappings during the codesign DSE.

The quality of mappings obtained by random search was also compared with those obtained by simulated annealing, genetic algorithm, and Bayesian optimization for

ResNet18 layers, as shown in Fig. 3.15. [6] The random search successfully achieved low-latency mappings for all layers, whereas simulated annealing [185] failed to map a few layers (in 10,000 trials), and genetic algorithm [186] led to a higher overall latency than random search and took almost four hours to optimize mappings for the nine layers, making it impractical for codesign search. Therefore, a Timeloop-like random search was used to quickly and efficiently explore the highly pruned mapping space.

3.11 Codesign Optimization: Multi-Stage or Joint?

The optimization of hardware and software codesigns can be done either by exploring partitioned sub-spaces in a sequential manner or simultaneously. In a partitioned or a two-stage optimization, an outer loop iterates over different hardware configurations, and an inner loop optimizes the software space for each hardware configuration selected. On the other hand, the joint or simultaneous exploration involves finding a new configuration for both the hardware and software parameters at the same time in a trial. Although approaches using simultaneous search have been proposed, they are often infeasible to apply to a multi-workload exploration, target system with diverse and time-consuming cost evaluations, and huge collective search space. Therefore, partitioned sub-space exploration is commonly used for optimizing codesigns (§3.2.2). For demonstration of Explainable-DSE, all the DSE evaluations also follow two-stage optimization.

Firstly, approaches using simultaneous search [2, 198] typically optimize configurations for individual loop kernels such as a single DNN layer, as they optimize both the hardware and software parameters at every search attempt. It does not necessarily lead to a single accelerator design that is most efficient for the entire workload or a

[6]The mappings were evaluated for the initial hardware configuration, corresponding to the lowest values of design parameters in Table 3.1.

set of workloads, as layer-specific designs may not be optimal overall for the entire DNN or multiple DNNs.

Furthermore, simultaneously optimizing both hardware and software parameters can be very time-consuming. A target system often involves different cost functions or modules for different metrics that could consume different evaluation times. For example, evaluating area and power of each hardware configuration via Accelergy [89] alone could take a few seconds, whereas the cost models of dMazeRunner [40] or Timeloop [47] could estimate latency/energy for hundreds–thousands of mappings in a second. For exploring codesigns for a DNN with L=50 unique layers, consider a black-box DSE that is budgeted H=2,500 trials for hardware configurations and M = 10,000 trials for mapping each DNN layer on each hardware configuration. Simultaneous exploration of hardware and software configurations in $H \times M$ trials for each of the L layers requires the system to evaluate power/area costs for $H \times M \times L$ times, which would take more than 0.7 million hours, or 79 years! In contrast, a two-stage partitioned exploration evaluates power/area costs only for H trials and if the DSE samples infeasible mappings for a hardware configuration, they can be discarded promptly without further detailed evaluation. The experiments show that the black-box DSEs obtained codesigns in a few days to a few weeks with exploring the partitioned subspaces.

Finally, in addition to the design parameters such as the total PEs or buffer sizes, hardware configurations can have various parameters, such as bandwidth, reconfiguration of NoCs (time-multiplexed communication of data packets, bus widths), and those for architectural specialization/heterogeneity, which further increase the search space for both the hardware and software/mapping configurations. With the vast space for both the hardware and software/mapping configurations, the collective search space becomes huge, compounding the already challenging exploration of feasible and effective solutions for either of the hardware and software parameters. Additionally, in the DSE trials, simultaneously acquired hardware and software configurations may not be compatible with each other or may not mitigate execution inefficiencies corresponding to their counterpart.

3.12 Discussion on Specification Needed by DSE Approaches

Black-box optimizations such as random search, simulated annealing, or Bayesian optimization are easy to deploy and require minimal tuning and specification for meaningfully curating and constraining the design space, which may take some hours or days. However, due to their non-explainable nature, they may mostly explore inefficient or infeasible solutions while consuming significant search time for excessive sampling. Applying these black-box exploration approaches to a domain-specific design optimization problem may still need some additional specification efforts [65, 191], depending on their implementation. For instance, Confuciux [65] is a reinforcement learning-based DSE framework that uses an LSTM/MLP-based policy network. For the targeted evaluations, it was extended to allow an arbitrary number of parameters, a different environment (target setup), different numbers of possible options for different parameters (different list sizes), and an arbitrary number of constraints. The reward calculation was also extended to consider an arbitrary number of constraints and their utilization. This extension required adding/modifying a few tens of lines of code (LoC) and a few days of work.

For bottleneck-guided DSE, a bottleneck model was developed for the target domain of DNN accelerator's hardware/mapping codesign. This is similar to efforts made previously in other domains [157–159, 172, 173]. The bottleneck model was expressed to the DSE framework via proposed API. It led to about tens of LoC that specified how different factors contribute to overall cost and a few to several lines for integrating first-hand information about bottleneck mitigation that provided predictions for new values of each parameter. It is worth noting that this is significantly less code and development efforts compared to domain-specific analytical cost models [40, 121], which typically require thousands of lines of code and provide only a single value for the total cost. The bottleneck model used in the DSE is generally simpler than the full analytical model, as it only considers major execution-related factors. And to infer new parameter values for bottleneck mitigation, it incorporates simple estimates or performs a walk-through of the populated values/paths in the bottleneck graph. In general, domain experts can derive the bottleneck model from either the graphical representation of the analytical model they develop/use or the sensitivity of a cost to the design parameters, which may take several days. §3.14 discusses how bottleneck mitigation may be generalized for arbitrary systems.

3.13 Discussion on Capabilities and Distinguished Features

This section highlights the capabilities of the proposed DSE using bottleneck models for agile and explainable explorations.

• **Efficient designs.** Explainable-DSE finds better solutions since it investigates costs and bottlenecks that incur higher costs; by exploring candidates that can mitigate inefficiencies in obtained designs, DSE provides efficient designs.

• **Quick/runtime DSE.** The DSE can reduce objective values at almost every acquisition attempt; it searches mostly in feasible/effectual solution spaces. Thus, DSE achieves efficient solutions quickly, which is beneficial for early design phase and for dynamic DSEs, e.g., deploying accelerator overlays at run time. Additionally, it can help when acquisition budgets are limited, e.g., due to evaluation of a solution consuming minutes to hours [41]. Further, when designers optimize designs offline with hybrid optimization methodologies [65] comprising multiple optimizations, quickly found efficient solutions can serve as high-quality initial points.

• **Explainability in the DSE and design process.** This work shows the need for explainability in the design process, e.g., in exploring the vast design space of deep learning accelerators and how DSE driven by bottleneck models can achieve explainability. Unlike cost models that provide a single value, bottleneck models can provide rich information in an explicitly analyzable format. Consequently, explorations based on bottleneck analysis can help explain why designs perform well/poorly and which regions are well-explored/unexplored in the vast space and why.

• **Generalized bottleneck-driven DSE for multiple micro-benchmarks and workloads.** In acquiring new candidates, DSE accounts for various bottlenecks in executing multiple loop nests (e.g., DNN layers) of diverse characteristics. Thus, the DSE can provide a single solution that is most effective overall, in contrast to previous DSEs that provide loop-kernel-specific solutions.

• **Specification for expressing domain-specific bottleneck models to the DSE.** This work proposes an API for expressing domain-specific bottleneck models so that the designers and/or design automation tools can integrate them to bottleneck-driven DSE frameworks and reuse the DSE.

• **Comprehensive design space specification.** In the DSE, appropriate values of a parameter is selected through bottleneck models. Thus, the DSE can alleviate the need for fine-tuning the design space; users can comprehensively define/explore vast space, e.g., more parameters and large ranges of values (arbitrary instead of power-of-two).

• **Bottleneck analysis for hardware/software codesign of deep learning accelerators.** By taking the latency of accelerators as an example, this work shows how to construct bottleneck models (for designing deep learning accelerators) and bottleneck analysis for improving the accelerator designs based on their execution characteristics. It shows how bottleneck models, as compared to conventional cost models, can be inherently analyzable and information-rich, allowing to make informed decisions for the design optimizations.

3.14 Further Opportunities and Research Directions

• **Improving efficiency further through better acquisitions:** By using bottleneck models and considering available budgets of constraints, the DSE can find outperforming solutions. However, it may still converge to a suboptimal solution (e.g., for VGG-16) due to the greed for resolving the bottlenecks for further optimizations. This challenge can be addressed by making the acquisitions more exploratory, e.g., exploring distant promising subspaces. It can be achieved by exploring multiple spaces side-by-side by targeting a pool of various initial points [2]. Alternatively, the acquisition function can incorporate both self-supervised learning or inducing randomness [103] and bottleneck/constraint considerations.

• **Exploring and addressing implications of specifying ineffectual bottleneck analysis for an arbitrary system:** In Explainable DSE, the bottleneck model helps explain design cost and guide the DSE. It considers which factors constitute overall cost and how design parameters and their scaling could impact each factor. For various domain-specific systems, designers usually characterize them manually and develop domain-specific bottleneck models/analyses or have first-hand information [157–159, 171–173]. Even for designing deep learning accelerators, designers already develop cost models or closely work with them [40, 41, 47, 51, 53, 54, 59, 61–63, 121]. Thus, they can determine bottleneck model and mitigation in various ways. For instance, designers can obtain a bottleneck model by simplifying their analytical cost model, which they can provide along with the cost model. Alternately, designers could estimate bottleneck mitigation through characterization or sensitivity analysis of design parameters. Designers can also opt for automation techniques, as discussed later.

A domain-specific bottleneck model constructed by domain experts or possibly learned from domain-specific data can lead to an effective DSE. However, in the absence of an effectual analysis for an arbitrary system, the resulting exploration may be slower or suboptimal. For instance, the DSE may require more acquisitions and be slower if the designer-provided mitigation scales the parameter values conservatively or associates irrelevant parameters with bottleneck factors (e.g., the total number of PEs to the DMA time). Conversely, the DSE may converge to suboptimal solutions if the designer either skips associating a critical parameter with a bottleneck factor or scales values aggressively. The examples include a) a user not suggesting explorations of NOC links or bit-widths when on-chip communication time is the bottleneck and b) the DSE scaling the number of PEs by 2× or more, even if the required scaling was only 1.2×. Either can cause the DSE to miss a range of efficient and constraints-satisfying solutions.

• **Automating construction of bottleneck models and bottleneck mitigation for arbitrary or large-scale domain-specific systems:** For expert-defined cost models, such as those of DNN accelerators, manual bottleneck analysis by designers with first-hand domain information can be possible. In general, when domain-specific architectures can be described and evaluated as flow graphs, the analysis of costs/bottlenecks may be automated by parsing execution information for architectural components [34, 199, 200].

There can be scenarios where designers may want to optimize the application processing for off-the-shelf processors or can only access pre-existing design models and simulators that are large-scale or complicated. In such cases, designers may not be able to provide the domain-information for constructing bottleneck models and available designs or large-scale simulation models (e.g., in C++/RTL) need to be used for deriving bottleneck models. Hence, learning-based approaches can be developed [201], which could be broadly applicable, while still leveraging the proposed structure/organization of the bottleneck models and their usage in gray-box/white-box design space exploration. Graph-based ML models or self-supervised ML models can be more suitable, including but not limited to decision trees, graph neural networks, and reinforcement learning. Likewise, bottleneck mitigation in complicated scenarios can be estimated using gray-box optimization functions that approximate the relevance and contributions of each parameter to the total cost [202] or through surrogates [103].

The dynamic DSE evaluation demonstrates the potential of incorporating explainability into the DSE. Efficiency, agility, and generalization of the DSE can be improved even further through improving the predictions for bottleneck mitigation, the decision mechanism that uses the feedback, aggregation and acquisition functions, and parallel evaluation/exploration of candidates and promising subspaces.

3.15 Conclusions

Agile and efficient exploration in the vast design space, e.g., for hardware/software codesigns of deep learning accelerators, require techniques that not just should consider objectives and constraints but are also explainable. They need to reason about obtained costs for acquired solutions and how to improve underlying execution inefficiencies. Non-explainable DSE with black-box optimizations (evolutionary, ML-based) lack such capability; obtaining efficient solutions even after thousands of trials or days can be challenging. To overcome such challenges, Explainable-DSE is proposed, which analyzes execution through bottleneck models. As compared to cost models that provide a total value, a bottleneck model can graphically express which and how various design parameters and intermediate factors contribute to the total cost.

Thus, it can provide rich information in an explicitly analyzable format, allowing the designers and DSE to identify the bottleneck factors for the obtained costs and acquire mitigating solutions. Proposed API can allow designers and/or automation tools to express their domain-specific bottleneck models and interface with the DSE. Through aggregation of predictions for bottleneck mitigation, the DSE facilitates a single effective solution for multi-functional or multiple workloads. In addition, awareness of utilized constraints in the decision making allows the DSE to prioritize exploration among feasible solutions and find more efficient solutions without quickly exhausting the constraints. The demonstration of optimizing codesigns of DNN accelerators showed how Explainable-DSE could effectively explore feasible and efficient candidates (6× low-latency solutions). By obtaining most efficient solutions in short exploration budgets (47× fewer iterations or minutes/hours vs. days/weeks), it opens up opportunities for cost-effective and dynamic explorations.

Chapter 4

AGILE EXECUTION MODELING

A comprehensive exploration of hardware/software/data codesigns in an agile manner can yield efficient accelerator designs and application executions. Such explorations, however, require developing tools for the accelerator system, including power/performance/area (PPA) cost models, a compiler, and a simulator. Towards their automated and sustainable development, this chapter first presents case study about the execution modeling of processing nested loops on dataflow accelerators for dense and sparse tensor computations. Then, it presents challenges in sustainable design development, followed by describing how a generic accelerator abstraction and corresponding automation methodology towards developing an agile design methodology.

4.1 Case Study: Modeling Execution of Nested Loops on Dataflow Accelerators

This section provides background about execution modeling of the accelerators by describing a case study about (manual) execution modeling for processing nested loops on dataflow accelerators. The described execution model has been in use in dMazeRunner framework [40] for evaluating the execution costs of processing DNN operators on accelerators. The architecture template is the same as the one used in chapter 2.7. To determine the goodness of an execution method statically, the methodology in this study explicitly models computation and communication costs for various architectural features and estimates the execution time and energy consumption. From an input loop-nest, the model analyzes indexing expressions and data dependencies of operands. Then, the model determines various data reuse factors, DMA invocations and burst sizes for managing non-contiguous data in SPM, miss penalty, data communication through NoC, and any stall cycles inter/intra-PE-group communication (for reduction operations).

4.1.1 Determining Data Allocation

For the given tiling factors of an execution method, the data to be allocated in RF of a PE, in SPM, and the data communicated to the PE array is determined as:

$$data\_alloc[option][op] = evaluate\_index\_expr(op, effective\_TC)$$

where, for each iv in the list `IV`,

$$effective\_TC[iv] = \begin{cases} TC[RF][iv] & ;\text{ option} = \text{RF} \\ TC[Spatial][iv] \times TC[RF][iv] & ;\text{ option} = \text{PE\_Array} \\ TC[Spatial][iv] \times TC[RF][iv] \times TC[SPM][iv] & ;\text{ option} = \text{SPM} \end{cases}$$

For example, to determine the data allocated in RFs of PEs, it is needed that

$$effective\_TC[iv] = TC[RF][iv]$$

i.e.,

$$effective\_TC[n] = TC[RF][n] = N\_RF, effective\_TC[fy] = TC[RF][fy] = Fy\_RF$$

and so forth.

Then, $data\_alloc[RF][W]$ is calculated by evaluating the indexing expression for operand W where, the value for index iv is used as $effective\_TC[iv]$. Thus, after analyzing index expressions of `W[m][c][fy][fx]`, the following can be obtained.

$$\begin{aligned} data\_alloc[RF][W] &= effective\_TC[m] \times effective\_TC[c] \times effective\_TC[fy] \times effective\_TC[fx] \\ &= M\_RF \times C\_RF \times Fy\_RF \times Fx\_RF \end{aligned}$$

Table 4.1: Notation for analytical modeling of dataflow execution.

Term	Interpretation
`IV=['n',..,'fx']`	List of loop index variables (from outermost to innermost loop).
`total_IVs`	Length of list IV (same as depth of the loop-nest).
`level`	Either of {Spatial, RF, SPM, DRAM, base}.
`TC[level][iv]`	2D array of loop iteration counts. For example, TC[RF]['n'] refers to N_RF = 4.
`effective_TC[iv]`	Vector of effective loop Trip-Counts per *iv*. Calculated to find the data allocation.
`data_alloc[option][op]`	option = {RF, PE_Array, SPM}; op is a data operand.
`Data_Reuse[level][op]`	level = {SPM, DRAM}; op is a data operand.
`Energy[option]`	option = {RF, Operation_Type, NoC, SPM, DRAM}. Operation_Type corresponds to operations supported by PEs (e.g., MAC, ADD).

When the RF tiling factors are $< 1, 1, 1, 1, 1, 1, 1, 3 >$, registers allocated in a PE for `W` are determined as $1\times1\times1\times3 = 3$. Similarly, after analyzing indexing expressions of W, the model determines the weights communicated to PE array as

$$data\_alloc[PE\_Array][W] = [M\_SPATIAL \times M\_RF] \times [C\_SPATIAL \times C\_RF] \times [Fy\_SPATIAL \times Fy\_RF] \times [Fx\_SPATIAL \times Fx\_RF]$$

4.1.2 Estimating Energy Consumption

Total energy for executing the nested loop consists of the energy consumed in RF accesses, in performing useful operations on PEs, in communicating data via interconnect, and in accessing the data from SPM and DRAM, i.e.,

$$\begin{aligned} Total\_Energy = {} & e\_Ops + e\_RF + (comm\_energy\_1\_SPM\_pass \times total\_SPM\_pass) \\ & + e\_DRAM \end{aligned}$$

In the proposed execution model of the dataflow accelerator, during each loop iteration, an operand is read/written from/to RF of a PE for the execution of an operation, i.e.,

$$total\_loop\_iterations = \prod_{i=1}^{total\_IVs} TC[base][IV_i]$$
$$e\_RF = total\_loop\_iterations \times \sum_{op=1}^{total\_Operands} Energy[RF]$$

In the example of Fig. 2.2(b), there are 2 read operands and 1 read+write (r+w) operand. So, the cost for RF accesses during each loop iteration is approximated as $4\times Energy[RF]$. Energy (pJ) of various operations and for accessing data elements from memory are obtained from the literature [4, 31] and provided as an input to the model. Moreover, energy for operations performed by PEs is:

$$e\_Ops = total\_loop\_iterations \times \sum_{opr=1}^{total\_Operations} Energy[Operation\_Type[opr]]$$

In the example of Fig. 2.4(b), just 1 Multiply-and-ACcumulate (MAC) operation is performed on a PE in executing a loop iteration. The model currently does not support loops with conditional statements. However, since each loop iteration sequentially executes on a PE, the model can be extended to accommodate them by taking the maximum latency and energy consumption of the true and false paths.

Based on tiling factors for L1 loops (level-1 buffer or RF), each PE executes a certain number of loop iterations to process the data from allocated registers. It is referred herein as one *RF pass*. During an RF pass, while PEs process data from RFs, new data for the next RF pass can be accessed from SPM and communicated to PEs via an interconnect network.

$$energy\_access\_SPM\_1\_RF\_pass[op] = data\_alloc[PE\_array][op] \times Energy[SPM]$$
$$energy\_NOC\_1\_RF\_pass[op] = data\_alloc[RF][op] \times p[op] \times Energy[NOC]$$
$$energy\_1\_RF\_pass[op] = energy\_NOC\_1\_RF\_pass[op] + energy\_access\_SPM\_1\_RF\_pass[op]$$

Although total data communicated to PE array is determined by `data_alloc[PE_array]`, many PEs may process the same data. Such spatial reuse is modeled by finding the total PEs that read/write the same operand. If an operand *op* belongs to a write operation, only those PEs are considered that produce the outcome. Thus,

$$p[op] = \begin{cases} \prod_{i=1}^{total\_IVs} TC[Spatial][IV_i] & \text{; op belongs to read operation} \\ \prod_{i=1}^{len(list\_dependent\_IV[op])} TC[Spatial][list\_dependent\_IV[op][i]] & \text{; op belongs to write operation} \end{cases}$$

Based on the ordering of the L2 loops (that correspond to SPM accesses), the reuse of data operands for the consecutive RF passes is determined and communication energy for one SPM pass is found.

$$comm\_energy\_1\_SPM\_pass = \sum_{op=1}^{total\_operands} energy\_1\_RF\_pass[op] \times (total\_RF\_pass \div Data\_Reuse[SPM][op])$$

After determining the data allocated in SPM (processed in 1 SPM pass) and the reuse of the data in SPM, the energy consumption for communicating data from DRAM can be determined as follows:

$$e\_DRAM = \sum_{op=1}^{total\_operands} data\_alloc[SPM][op] \times Energy[DRAM] \times (total\_SPM\_pass \div Data\_Reuse[DRAM][op])$$

4.1.3 Estimating Execution Time

During processing the data in a RF pass, PEs execute certain number of iterations and perform all operations within each loop iteration. So, estimated cycles are:

$$cycles\_UsefulOps = loop\_iterations\_RF\_pass \times latency\_1\_loop\_iteration$$

$$loop\_iterations\_RF\_pass = \prod_{i=1}^{total\_IVs} TC[RF][IVi]$$

$$latency\_1\_loop\_iteration = \sum_{opr=1}^{total\_operations} l$$

$$l = \begin{cases} 1 & \text{; if PE is pipelined} \\ latency[operation_{opr}] & \text{; PE is nonpipelined} \end{cases}$$

During an RF pass, while PEs process data from RFs, new data for the next RF pass can be accessed from SPM and communicated to PEs via interconnect. This interleaving of the communication latency with the computation being performed by PEs can be either achieved by double-buffering the RFs or through software scheduling scheme. If no such support is available, the PE array completely stalls to obtain the necessary data from SPM for the next RF pass. Total cycles required to communicate operands during a RF pass is:

$$comm\_cycles\_operand[op] = data\_alloc[PE\_array][op] \ / \ B$$

where `B` is the width of the data bus for interconnect. Depending on the ordering of L2 loops (that correspond to accessing the data from SPM), some operands are not reused after an RF pass and communicated between SPM and the PE array at every RF pass. However, some operand(s) can be reused and are communicated at every xth RF pass. For example, for an ordering where the loop with index variable `c_L2` is innermost, the ofmap O (or the psum) is reused for `C_SPM`=4 consecutive RF passes. Taking that into account, the communication latency is determined as:

$$comm\_cycles[RF\_pass\#][network\#] = map\_operands\_to\_NOC(\\ comm\_cycles\_operand, Data\_Reuse[SPM])$$

In a default setup, popular single-cycle multi-cast interconnect is supported. The networks to communicate read and write operands between SPM and PE array are three and one, respectively [4, 31]. There is one network to communicate r+w operands among PEs (used for reduction operations). Often the total operands in the loop-nest are few and are simultaneously communicated to/from the PEs via interconnect (including executing common kernels like matrix multiplication, convolution, regression, and sequence models). If not, they need to be sequentially broadcast to PEs via available interconnect. For example, when the total data operands are more than available networks, the communication can be scheduled onto networks via a round-robin mechanism. In fact, for performing design space exploration through dMazeRunner, architects can extend the model to accommodate various interconnect topologies. Total cycles required to process the data of SPM (*1 SPM pass*) are:

$$cycles\_SPM\_pass = \sum_{i=1}^{total\_RF\_pass} max(cycles\_UsefulOps,$$

$$\max_{1 \le j \le total\_networks} comm\_cycles[i][j])$$

Usually, the execution requires several SPM passes. During each SPM pass, the PE array processes the data from one buffer of SPM, and DMA controller accesses DRAM for the data of another buffer. After calculating the size of the data allocated in SPM, the total DMA invocations required and the burst size (of contiguous data) per invocation can be determined. To calculate DMA cycles, a latency model of Cell processors [154] (that featured SPMs) is considered, i.e.,

$$DMA\_Model(u) = 291 \text{ (initiation latency)} + 0.24 \times u; \qquad \text{u: burst width (bytes)}$$
$$cont\_data\_alloc\_spm[op], DMA\_initiations[op] \leftarrow data\_alloc[SPM][op]$$
$$DMA\_cycles[op] = DMA\_Model(cont\_data\_alloc\_spm[op]) \times DMA\_initiations[op] \times (accel\_freq \div dma\_freq)$$

Based on the data being reused in consecutive SPM passes, the cycles required for accessing the DRAM during each SPM pass can be calculated as follows:

$$DRAM\_access\_cycles[SPM\_pass\ \#] = \sum_{op=1}^{total\_operands} DMA\_cycles[op] \quad if(SPM\_pass\# \bmod Data\_Reuse[DRAM][op] == 0)$$
$$total\_cycles = \sum_{i=1}^{total\_SPM\_pass} max(DRAM\_access\_cycles[i], cycles\_SPM\_pass)$$

Note: Implementation of the proposed execution model deals with the various complex scenarios including:

• *Modeling stall cycles and energy consumption for inter-PE communication:* When a r+w operand (e.g., O) is invariant of a loop (c, `fy`, `fx`) that executes spatially (e.g., `C_SPATIAL` > 1), computing the output requires inter-PE communication. Depending on the data buffering mechanism of the RF, PE array may not start processing new data from RF or cannot get new data from interconnect while PEs perform the reduction operations onto previously computed data. Therefore, depending on the spatial execution of loops and data reuse factors, stall cycles and energy consumed are accounted.

• *Accurate modeling of continuous data reuse through several RF+SPM passes:* Depending on the ordering of the loops, some operand gets reused continuously throughout all RF passes of an SPM pass and through several such SPM passes. For example, for an ordering of L2 loops with IVs {`n_L2,m_L2,oy_L2,ox_l2,c_L2,fy_L2,fx_L2`} (outermost to innermost) with TCs $< 1, 1, 1, 1, 4, 3, 3 >$, total RF passes in an SPM pass are $4\times3\times3 = 36$. In each RF pass, operands I and W are communicated from SPM to RFs via NoC while O is reused in RFs. Now, for an ordering of L3 loops with IVs {`oy_L3,ox_l3,fy_L3,fx_L3, n_L3,m_L3,c_L3`} with TCs $< 1, 1, 1, 1, 2, 32, 16 >$, O gets reused in consecutive 16 SPM passes. Thus, write-back of O occurs just once after every 16 SPM passes; each SPM pass consists of 36 RF passes. Such reuse of data at consecutive memory levels is referred herein as a continuous reuse and it is accurately modeled for various operands.

• *Detailed model of data reuse and communication for r+w operands:* Processing of a r+w operand on PE array may require to read previously computed value (e.g., input psum) from SPM and interconnect. Furthermore, such read operation can be skipped some times, if the operand is zero-initialized. Thus, the calculation of the miss penalty and energy consumption are offset accordingly.

4.1.4 Validation Experiments

Specification of the target platform The accelerator architecture modeled is the same as the one used in the chapter 2.7.

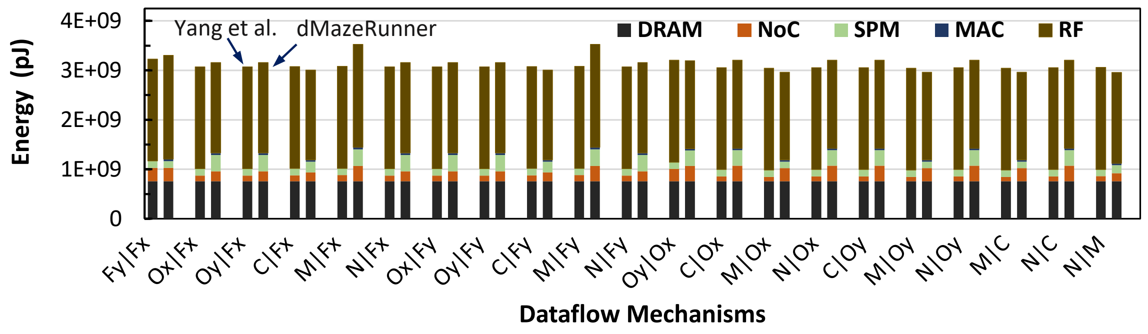


Figure 4.1: Validation results for ResNet conv5_2. The energy consumption estimated by the execution model is close to the energy model of Yang et al. (2018).

Validation against Yang et al. (2018) [149]: To determine the accuracy of the proposed dataflow execution model, it is validated against evaluations of a recent work [31, 149]. Validating the execution model is often challenging since it requires (i) the same architecture specification, (ii) the information about the adopted execution method, and (iii) the absolute values of execution time and energy consumed by the platforms. Therefore, the execution methods used are those ones that are obtained by the tool [149] and the same methods are evaluated with the proposed analytical model.

This validation experiment covers various *dataflow mechanisms* which represent *how different loops are executed spatially.* For example, `Fy|Fx` represents a weight stationary mechanism where PEs are grouped based on unrolling Fy and Fx loops for spatial execution [31]. Note that these dataflow mechanisms also incorporate the variations in temporal execution (different data reuse patterns) and the spatial execution of more than two loops.

It is found that the proposed model achieves the same PE utilization as Yang et al. [31, 149]. Moreover, Fig. 4.1 shows that for various dataflow mechanisms, the energy consumption (in pJ) estimated by the model closely matches to the energy estimation tool [149] (the difference is 4.19%). In fact, for commonly used dataflow mechanisms

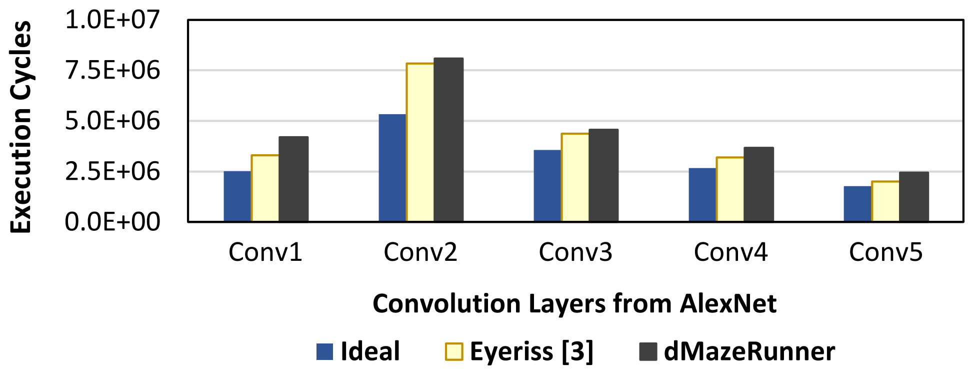


Figure 4.2: Performance validation against Eyeriss architecture.

like output-stationary (`Oy|Ox`), the difference is 0.3%. A higher difference (about 14%) was observed for $M|Fx$ mechanism. A possible reason is that the model of [149] is more accurate for the interconnect organization (e.g., per-hop communication cost) while the proposed model considers the same cost for multicast communication.

Furthermore, Fig. 4.1 shows the breakdown of the energy for system resources where, each bar on the left-hand side represents the evaluation from Yang et al. [149], and the second bar for each mechanism represents the estimation from the proposed execution model. It is found that energy estimated for system resources is similar to that obtained by [149]. In fact, for optimized execution methods, the estimated energy for DRAM accesses is very low, and most of the energy consumption is attributed to accessing data from RFs.

Validation against Eyeriss architecture: The proposed dataflow execution model is extended for modeling Eyeriss accelerator [4] that executed AlexNet for ImageNet classification [9]. Execution methods reported in [4] are evaluated. For modeling, the following Eyeriss-specific enhancements are considered: (i) separate and larger bit-widths of different input, output, and reduction networks, (ii) a dedicated mesh-style network for reduction, and (iii) in communicating the data (e.g., a row of 1×3 ifmap), reusing the neighborhood data (1×2 ifmap) from RFs in the sliding-window execution.

Fig. 4.2 shows the execution cycles considering (a) ideal acceleration (i.e., total MAC operations ÷ total PEs), (b) processing time reported for Eyeriss architecture [4], and (c) estimation of execution cycles. It is found that the estimations closely matched execution cycles of the architecture [4], with a difference of 11% in the total execution cycles. Thus, the execution model can effectively estimate the execution costs of processing loops on dataflow accelerators.

4.2 Case Study: Execution Modeling of Sparse DNN Accelerators

4.2.1 Need for Efficiently Exploiting Sparsity

Deep learning models achieve high task accuracy for various applications but incur high execution time, energy, storage, and environmental implications due to data-intensive models [33]. So, compact models are developed with sparsification, shape reduction, and quantization of (often over-parameterized) tensors [203]. They can enable notable efficiency (20× fewer operations in a GEMM with 90%/50% sparsity of matrices) while achieving target accuracy. Many other domains also face challenges in exploiting sparsity, and accelerators have been proposed for some of the more processing-intensive domains; this includes graph processing [204, 205], database operations [206], genomics [207, 208], and compression [209]. In some cases, computation primitives even extend across domains. For example, finding intersecting non-zeros is analogous to joins in a database context [210]. Therefore, developing techniques for efficient sparse tensor processing in an ML context can likely speed up innovation in a broader context.

While commodity and emerging AI accelerators support different quantizations, exploiting acceleration opportunities due to sparse and irregular-shaped tensors remains challenging. For hardware accelerators, *leveraging sparsity necessitates mechanisms to* *store, extract, communicate, compute, and load-balance the non-zeros (NZs) that are scattered in tensors* [33]. Otherwise, execution is inefficient by orders of magnitude, e.g., on conventional AI engines [4, 29, 30, 42, 115, 125, 147] or CPUs/GPUs [5, 33, 211–213]. These NZ-processing mechanisms can be designed at hardware/software/model levels and their availability and implementation choices determine the kinds of sparse models that can be accelerated and the obtained efficiency. Using such mechanisms, *accelerators nowadays can leverage sparsity, but they are efficient for processing sparsity of only limited range or fixed patterns. This is because of current ad hoc design approach where designers make certain choices of design features for targeting some workloads. As a result, existing accelerators leave notable acceleration on the table when processing diverse* [1] *sparsity range/patterns present within and across deep learning models of various application domains.*

4.2.2 Accelerators for Sparse DNNs

Sparse, size-reduced, and quantized tensors of ML models offer various opportunities for storage, performance, and energy efficiency. Hence, several accelerators have provided marginal or comprehensive support and leveraged some or all the opportunities. Table 4.2 lists such common objectives and corresponding accelerator solutions that meet these objectives.

Different accelerators for inference and learning exploit W-sparsity, IA-sparsity, or both, which impacts acceleration gains [239]. Several accelerators, including Cambricon-X [5], exploit only static sparsity (Table 4.3), e.g., when locations of zeros in weights are known beforehand for inference. Static sparsity allows offline encoding

[1] Depending on the application domain, tensors contain low, e.g., (0%–30%), moderate (30%–75%), high (75%–99%), or hyper (99%+) sparsity. The need to support diverse sparsity levels/patterns of deep learning models [33], unlike conventional hyper-sparsity in HPC, and exploiting the achievable efficiency of accelerators impose the challenge.

Table 4.2: Accelerators for processing sparse tensor computations.

Objective	Techniques
Compressed data in off-chip memory (storage)	[4, 5, 38, 42, 94, 95, 112, 122, 212, 214–232]
Compressed data in on-chip memory (storage)	[5, 38, 94, 95, 112, 122, 212, 214–220, 222–230, 232–234]
Skip processing zeros (energy efficiency)	[4, 5, 94, 95, 112, 122, 212, 214, 216–243]
, Reduce ineffectual computation cycles (performance & energy)	[5, 94, 95, 112, 122, 212, 214–220, 222–230, 232–234, 238, 239, 243]
Load balancing (performance)	[94, 95, 122, 219, 222–224, 227, 230, 232, 235, 236, 239]

and data transformations for arranging structured computations (e.g., for systolic arrays [235, 244, 247]). Recent accelerators, including ZENA [239], SNAP [218], and EyerissV2 [95], leverage dynamic sparsity also. It requires determining locations of intersecting NZs in both tensors at run-time to feed functional units, on-the-fly decoding (encoding) NZs, and often balancing computations on PEs. Table 4.3 lists different accelerators that support static and dynamic sparsity of tensors.

Previous accelerator designs for compact ML models varied in terms of the hardware design space, e.g., a certain type of architectural component for sparsity-decoding

Table 4.3: Accelerators leveraging sparsity of different tensors for different ML models.

Dynamicity of Sparsity	Static		[5, 112, 215, 219, 228, 230–232, 235, 236, 244]
	Dynamic		[4, 38, 94, 95, 122, 212, 214, 216–218, 220–227, 229, 233, 234] [237–242], [240, 245, 246]
Tensors Treated as Sparse	Weight	Unstructured	[5, 112, 215, 231, 232, 236], [247]
		Structured	[84, 94, 219, 227, 230, 235, 244, 248], [249]
	Activation		[4, 38, 222, 229, 233, 234, 237, 242], [240, 245, 246]
	Both		[94, 95, 122, 212, 214, 216–218, 220, 221, 223–228, 238, 239, 241, 243]
Primitive Operation	Matrix-Vector Multiply		[5, 94, 95, 122, 216, 218–220, 222, 223, 225, 228, 238, 242]
	Matrix-Matrix Multiply		[5, 212, 223, 225, 235, 236, 241, 243]
	Convolution		[4, 5, 38, 94, 95, 112, 214–219, 221–224, 226, 227, 229, 231–233, 238]
	Recurrent / Attention Layer		[230, 234, 237, 250], [240, 245, 246, 249]
Accelerators for Learning			[212, 222]

or non-zero extraction. For instance, for existing accelerators, hardware structures for processing NZs are designed with some choices, which, with the configuration of their design hyperparameters, impacts the acceleration achieved. For example, in Cambricon-X/S [5, 251], the data extraction mechanism processes a 256-element stream to feed 16 multipliers of dot product engine every cycle. So, they efficiently process up to ~90% sparsity, but for sparser NLP models like BERT-base-uncased [14] (92% sparse weights [182]) on SQuAD [252], get only ~8× from 12× achievable speedup. [2] Their efficiency falls steeply with increased sparsity, or when multiple tensors are highly or hyper sparse [18, 253–257]. Further, Cambricon-X only process static sparsity of weights and Cambricon-S requires weights with a certain block-sparsity [258] structure. ExTensor [225], an accelerator for hyper-sparse computations, used an intersection mechanism for extracting NZs that compares two streams of positions of NZs. Such choice is effective when positions of NZs in two tensors are identical or within a short range (e.g., low/moderate sparsity); obtained speedup on their evaluated benchmarks fell short by ~4× and ~8×, as compared to the achievable and ideal speedups, respectively. SIGMA [212] extracts data similarly, and doing so as pre-processing worsens the latency. Function units in SCNN [217] (supports only unit-strided convolutions) and SNAP [218] contain multipliers and adder trees, which are poorly utilized (below 20%) for high sparsity (90%) [218], so they can be inefficient for sparse workloads as MobileNetV2 [259] or Transformers [13, 14]. Further, all accelerators encode tensors in some fixed format that is efficient for a limited sparsity range [33] (e.g., CSR at hyper-sparsity and bitmaps/RLC-4 for low/moderate sparsity [33]).

A few approaches optimized SpMV on CPU/GPU [260–263] but lead to specific

[2] Discussion about the speedup of an accelerator, unless mentioned otherwise in this section, refers to a reduction in execution time by leveraging sparsity as compared to dense tensor processing on an accelerator with a similar configuration at its peak.

dataflow (suitable only when one tensor is hyper-sparse and another is dense). With specific hardware design components for sparsity, execution methods for accelerators are maneuvered, often degrading the efficiency. For instance, EIE [122] and EyerissV2 [95] use weight stationary dataflow [33, 45] for processing tensors on PEs, where weights (sparser) are indexed (looked-up) with a position of an NZ activation (denser). So, the obtained efficiency vs. peak can be low.

To overcome inefficiencies in processing unstructured sparsity, some techniques induce coarse-grain sparse structures, e.g., by pruning entire tensor dimensions [211] or blocks of elements [94, 248, 258, 264]. When being accelerator-unaware, such pruning does not yield the desired efficiency and is also inapplicable for conventional accelerators. This is because, no synergy is established between attained pruning and the target accelerator's requirements for execution and efficiency. For instance, density-bounded blocks need to be encoded similar to blocks with unstructured sparsity [33], e.g., with bitmap [244], one-dimensional coordinates [219], or run-length coding. So, for the same sparsity and block size, the storage overhead is similar to processing a tensor that has unstructured sparsity. Coarse-grain block-sparse tensors can be encoded at block-granularity, which can significantly reduce the metadata size (almost eliminated for dimensional pruning [265]). Coarse-grain pruning, however, has some limitations in the sparsity that can be achieved without reducing the task accuracy. Moreover, some hardware/software support is still needed to process NZs, including determining their positions. Recently, accelerator designs have been explored to support some sparsity structures [219, 251], and they also share similar inefficiencies. For example, sparse tensor cores in nVidia A100 [84] support a fixed 50% block sparsity in a single tensor, and Cambricon-S [94] is inefficient in exploiting high or hyper sparsity.

4.2.3 Overview of Components for Sparse Tensor Accelerators

In contrast to settling upon arbitrary design choices, this work develops design abstractions and taxonomy to establish a systematic representation of the wide space at each hardwarw/software/model design level and quantify them to determine their efficiency for varying sparsity. For the hardware design space, abstractions can be developed for specifying mechanisms for a variety of encoding, decoding, extraction, communication, buffering, computation, and load-balancing of NZs. For instance, abstractions for data extraction can define various ways for locating the matching NZs in tensors based on the positions of NZs, along with corresponding functionality and execution cost. Likewise, abstractions for buffering could cover different models of accessing streaming/stationary data and metadata (for tensors in different sparsity-encoded formats), and so forth. Altogether, these abstractions can capture a wide range of microarchitectural features involved in processing NZs of varying patterns, from obtaining the input data to storing the produced output (and encoding it). Therefore, they could enable systematic modeling of the accelerator functionality and execution efficiency for a wide design space, including some common and previously proposed designs. Plus, implementation of these decoupled abstractions could allow hardware designers to configure the accelerator by selecting from the available features or even define special-purpose features and evaluate it quickly with better modularity and reusability. Next, different hardware and software aspects of the accelerator system are described that help in leveraging sparsity effectively.

Sparsity encodings: Sparse tensors are compressed using encodings, where only NZ values are stored in a "data" tensor and one or more "metadata" tensors encode locations of NZs. There are different formats and associated costs for encoding and decoding. For different sparsity levels, their effectiveness can vary in terms of storage efficiency. E.g., tensors can be compressed by 1.8× and 2.8× for 50% and 70% sparsity (bitmap or RLC-2) and 7.6× and 55×–60× for 90% (RLC-4) and 99% sparsity (CSC or RLC-7). Structured sparsity (coarse-grain block-sparse) can alleviate the overheads of metadata and fine-grained data extraction by encoding indices for only large dense blocks. For accelerating ML models, sparse tensors are also quantized i.e., their precisions are lowered (typically int8 or int16 for inference [95, 217, 239] and FP16 for learning [212, 222]) and often approximated by clustering data of similar values [38, 94, 122]. Therefore, encoded sparse data contains quantized values of NZs.

NZ detection and data extraction: In processing sparse tensors of different primitives, corresponding elements of the weight and activation tensors are multiplied and accumulated. Depending on the sparsity, accelerators need to use data extraction logic that decodes compressed tensors, search within a window of NZs or index the buffer, and obtain matching pairs of NZs to feed the functional units for computation. Up to moderate IA-sparsity and high W-sparsity, these indexing or intersection-based mechanisms efficiently extract sufficient NZs at every cycle for keeping functional units engaged. For efficient compute-bounded executions at such sparsity, accelerators reported achieving near-ideal speedups (e.g., about 80%–97% of the speedup corresponding to reduced operations, i.e., *sparsity-speedup ratio*) [5, 122, 239]. However, extraction becomes challenging at high (e.g., 90%+) or hyper sparsity as NZs are scattered at distant locations [254], and execution is usually memory-bounded with low arithmetic intensity. The data extraction mechanism can be either shared among PEs or employed in PEs, which also impacts overall efficiency.

Memory management: Compressed tensors are often stored in the shared on-chip memory that is non-coherent, multi-banked, and often non-unified. For a pre-determined sequence of execution, a controller or PEs initiates the accesses between off-chip and on-chip memory; their latency needs to be hidden behind computations on

PEs. The techniques for hiding miss penalty for sparse tensors include double-buffering or asynchronous computation and memory accesses. The data reuse opportunities can vary for various sparsity and dimensions of tensors of common DNNs and sparsity can lower the reuse. Further, techniques could leverage cross-layer reuse of intermediate output layers and reduce the overall latency.

Communication networks: Once tensor blocks are fetched from memory, they are distributed to appropriate PEs via interconnect networks (often one per operand). Efficient designs ensure that sufficient data can be fed to PEs while they perform computations. Reuse is leveraged spatially by multicast or mesh networks that communicate common data blocks to multiple PEs. It lowers accesses to memory hierarchy and communication latency. However, spatial reuse opportunities vary depending on the sparsity, NZ extraction mechanism, and mapping of the functionality on the accelerator. Executing inter-PE communications can be challenging, as it may become unstructured due to sparsity and the temporal and spatial mechanisms for reduction/collection of the outputs. Configurable designs can support various communication patterns for different sparsity, reuse, and functionality.

PE architecture: Several accelerators consist of scalar PEs with fused MAC units (e.g., EIE [122], LNPU [222], and Envision [221]). Others contain SIMD PEs (multiple functional units) (e.g., EyerissV2 [95]) or vector PEs consisting of multiplier-arrays and adder-trees (e.g., Cambricon-X [5] and SNAP [218]). PE architectures either directly process pairs of matching NZs extracted from tensors or use hardware logic for data extraction or coordinate computation. Effectively utilizing functional units can be challenging for variations in sparsity, precisions, and functionality, and it may require configurable designs. Further, sparsity-aware dataflow mechanisms (mapping of tensor computations on accelerator resources) can be used by different accelerators. Accelerators have leveraged value similarity of tensors and the corresponding modifications in the PE architecture.

Load balancing: Depending on the distribution of zeros, the execution may end up with processing a different amount of NZs on different PEs or their functional units, which creates inter-PE or intra-PE load imbalance. accelerators achieve load balance through either software techniques (e.g., structured pruning or data reorganization) or by providing a hardware module for dynamic work balance (through asynchronous execution or work sharing), which provides further accelerations. For example, ZENA [239] leveraged the sparsity of both activation and weight tensors for AlexNet and VGG-16 models and reported about 32% additional performance gains through load balancing. Dynamic load balancing can provide notable speedups for high, unstructured sparsity [254].

Write-back and post-processing: Tensor elements produced by PEs need to be collected, post-processed for further operations, and written back to the memory. PEs in different accelerators either write back sequentially or asynchronously through a shared bus or via point-to-point links. In addition, accelerators usually contain a post-processing unit that re-organizes the data (as per the dataflow mechanism of the current and next layer of the model) and encodes sparse output on the fly.

More details on categorization for these components, their trade-offs are available in [33]. Next accelerations achieved by some template architectures are discussed along with their analysis.

4.2.4 Analysis: Accelerations Achieved

This section analyzes the sparsity of recent DNN models (for NLP and CV) and the acceleration that can be achieved with some of the popular accelerators-alike architectures.

DNN models: Table 4.4 summarizes analyzed DNN models and their overall

Table 4.4: Sparsity of some popular DNNs.

Model	Domain	Dataset	GOps (dense)	Sparsity %			Sparse Model
				IA	W	Ops	
MobileNetV2 [259]	CV	ImageNet	0.3	34	52	81	[266]
EfficientNetB0 [112]	CV	ImageNet	0.5	0	68	60	[266]
Transformer [13]	NLP	WMT En-De	4.6	0	79	79	[267]
BERT-base-uncased [14]	NLP	SQuAD	9.3	0	92	92	[182]

sparsity across all CONV, GEMM, and DW-CONV operations. For each of these DNN operations, W-sparsity was obtained from sparse DNN models (listed in the last column). IA-sparsity was obtained by performing inference with sample data (images and text sequences).

Acceleators: Table 4.5 summarizes analyzed accelerators and their sparsity-centered features. Their architectures targeted unstructured or block-sparse sparsity of activations and/or weights. Their features represent variations across data encoding, data extraction, vector processing, memory hierarchy, NoC, and load balancing.

Methodology: To determine the impact of sparsity on achievable acceleration, a data-driven analysis of the execution latency was performed. For each DNN layer, zeros (or blocks of zeros) were induced randomly according to the sparsity of its tensors. The overall execution time was determined from the latency of processing on functional units, data decoding, extraction of non-zeros, work synchronization, and off-chip memory transfers, which were calculated based on analytical modeling of the microarchitectural features. The speedups were calculated over oracle processing of dense tensors at the accelerator's peak utilization of computational resources and off-chip bandwidth. In this study, the processing of DW-CONV on these accelerators

Table 4.5: Architectural features of analyzed accelerators for sparse DNNs.

ID	Reference Architecture	Supported Operators	Sparsity Leveraged		Non-zero Data Extraction			PE Architecture	Work Synch-ronization	Freq. (GHz)	DRAM BW (GBPS)	Bit-width			
												data		metadata	
			IA	W	Encoding	Discovery	Loc.	FU				IA / O	W	IA	W
A1	EIE [122]	GEMM	unstructured		CSR	Indexing	in-PE	Scalar	Prefetch	0.8	256	16	4	N/A	4
A2	Cambricon-X [5]	CONV, GEMM	dense	unstru-ctured	COO-1D		central (per PE)	Vector (16 multipliers & adder tree)	Every Output Activation	1	256	16	16	N/A	8
A3	Cambricon-S [94]		unstru-ctured	block-sparse	Bitmap	Inter-section	central, shared			1	256	16	8	1	1
A4	ZENA-IA-W [239]	CONV	unstructured				in-PE	Scalar	Intra-Workgroup	0.2	12	16	16	1	1

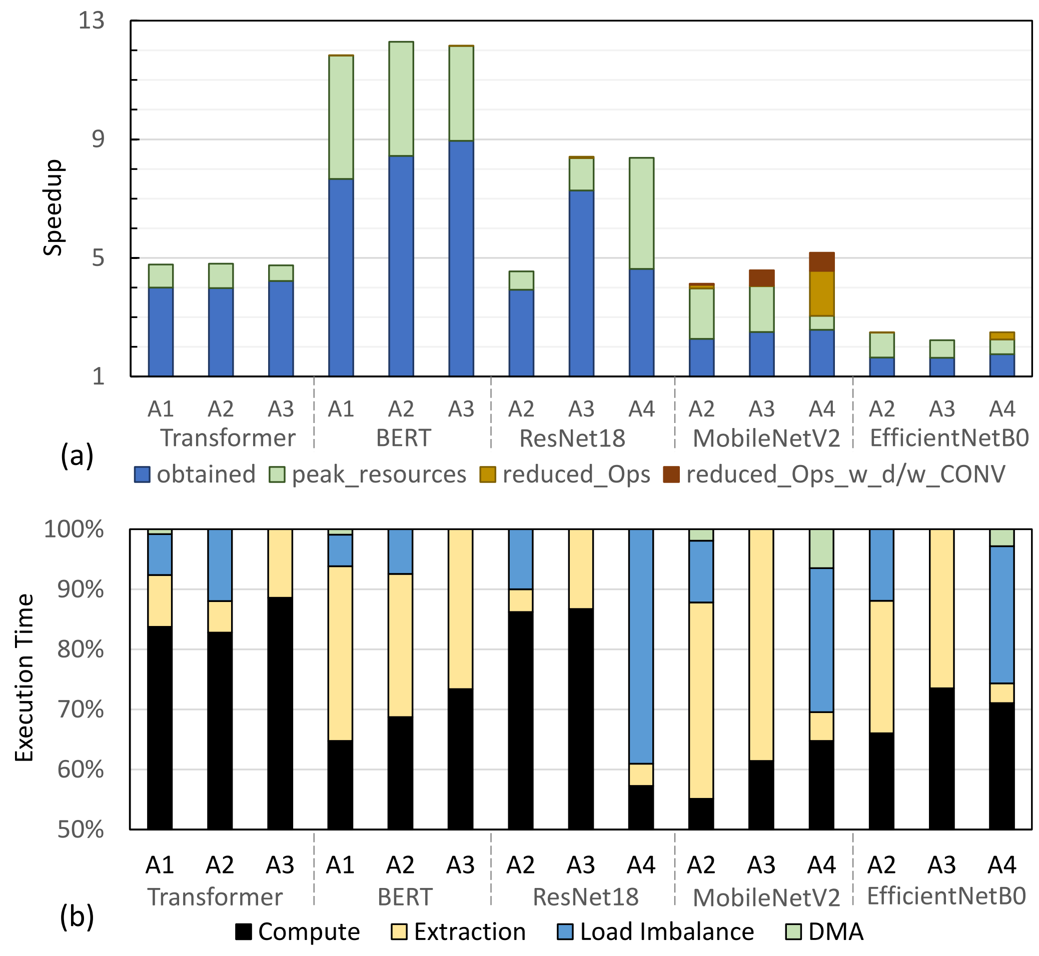


Figure 4.3: (a) Obtained speedups for accelerators listed in Table 4.5. (b) Analysis of execution time overheads for obtained accelerations.

is not considered, since they are often not pruned, and their execution needs to be group-wise, which is extremely inefficient. Such unsupported performance-critical operators were assumed to be processed with dense tensors at peak utilization of hardware resources.

Analysis: Fig. 4.3(a) shows speedups of accelerators for targeted DNN models, for leveraging the sparsity of supported DNN operators. It illustrates speedups for (i) reduction in the operations due to sparsity (desired), (ii) peak utilization of accelerator's computational resources and off-chip bandwidth while leveraging sparsity, over such oracle processing of dense tensors (potential), and (iii) actual processing on accelerator over oracle processing of dense tensors (obtained). For understanding implications of execution overheads including those incurred by metadata processing and load imbalance, Fig. 4.3(b) illustrates fractions for desired computation time and execution overheads in a stacked format. The overheads were extracted for layer-wise processing and then accumulated to determine the overall impact. Fractions include:

• Computation time: Minimum execution time required for processing at peak on accelerator's functional units.

• NZ extraction: Time required for decoding NZs from communicated operands and extracting matching operands for feeding the functional units. It also corresponds to balanced computations.

• Load imbalance: Time required for on-chip processing on the accelerator, considering the imbalanced computations subjected to the accelerator's work synchronization and work sharing schemes.

• DMA time: Time required for off-chip data communication via DMA transfers, in addition to on-chip processing.

Fig. 4.3(a) shows that accelerators efficiently exploited moderate sparsity. E.g., for 4.8× reductions in operations of Transformer due to W-sparsity, they achieved about 4×–4.2× speedup. The exploitation of speedup lowers when activations are dense and weights are highly or hyper-sparse. This is because accelerators like EIE and Cambricon-X broadcast activations to PEs and extract matching pairs corresponding to NZ weights. So, communication of activations and extraction of matching NZ operands consume significant execution time, while there are fewer operations to feed the functional units (Fig. 4.3b). E.g., for BERT-base-uncased [14] (92% sparse weights [182]) on SQuAD [252], they achieved about 7.7×–8.9× speedup out of 12.2× speedup for processing at peak. Due to block-sparse weights, computations on PEs of Cambricon-S are always balanced. Therefore, it achieved higher speedups. By using blocks of 16×16 or even 1×16 (across input and output channels) for pruning, inducing similar sparsity is not possible sometimes. So, the reduction in operations and potential for the speedup was slightly lower for Cambricon-S (e.g., for EfficientNetB0). In general, due to high DRAM bandwidth, overheads incurred by DMA transfers were hidden (for Cambricon-X/S) or negligible for non-interleaved transfers (e.g., for EIE).

Fig. 4.3(a) also shows that Cambricon-S and ZENA-IA-W achieved higher speedups for CV models by leveraging unstructured sparsity of activations. High IA-sparsity amplified total sparsity during processing several layers (e.g., MobileNetV2), incurring considerable excess processing in data extraction for Cambricon-X/S and in load imbalance for ZENA-IA-W. With zero-aware static sorting of filters and dynamic load balance, ZENA [239] could overcome such imbalance. But, it would suffer through high on-chip communication time since it used only one shared bus for multicast via NoC and collecting outputs. Such communication overhead was disregarded for ZENA-IA-W in this study, as most accelerators use separate NoCs or buses for alleviating communication overheads. Also, due to low DRAM bandwidth, overheads incurred by DMA transfers were higher for ZENA-IA-W, mainly for executing DW-CONVs with dense tensors.

To summarize, efficiently exploiting sparsity requires dedicated support through new hardware and/or software modules in domain-specific architectures that need to be tailored as per the sparsity and its pattern. Due to such newer modules, their execution modeling and analysis requires developing additional tools.

4.3 Automating the Architectural Execution Modeling and Characterization

Most research efforts have focused on proposing new domain-specific architectures a.k.a. 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 chapter 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 proposed 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.

4.3.1 Overview

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 [40, 46, 47, 121, 177] 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 [41, 76, 77]. 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 [66, 103, 124, 268], they also face same limitations. This is because, the 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.

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 [36] 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. Generic methodologies are required 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 for a highly customized architecture template, 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 [40, 75, 121]. 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 [89]. For a more generalized execution modeling, such information about execution activity needs to be supplied automatically.

Proposed approach. This chapter proposes *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. This work proposes to 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 [4, 5] 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 [4]) can be easily extended to model and characterize the latency/energy of DSAs for sparse DNNs (e.g., Cambricon-X [5], EIE [122], or SIGMA [212]), 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 [40, 47, 121] and weeks–months of development efforts [88]. Further, this work shows a study of automated characterization of Cambricon-X like accelerator for BERT, which matches prior analysis [33] 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×.

4.3.2 *DSAProf: Modular, Dataflow- driven Execution Modeling*

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. Proposed abstraction for specifying and evaluating the domain-specific architectures is a flow graph (FG). In the proposed flow graph representation for DSAs, the nodes are primary components of computation, memory, control logic, or interconnect networks [74, 199], 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. 4.4 represents an example flow graph for the Cambricon-X [5] 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 [177], unified or shared L1 or L2 buffers, compute engines accessing data from on-chip buffers or even directly from off-chip memory via DMAs [5, 47, 269].

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.

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 the proposed 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.

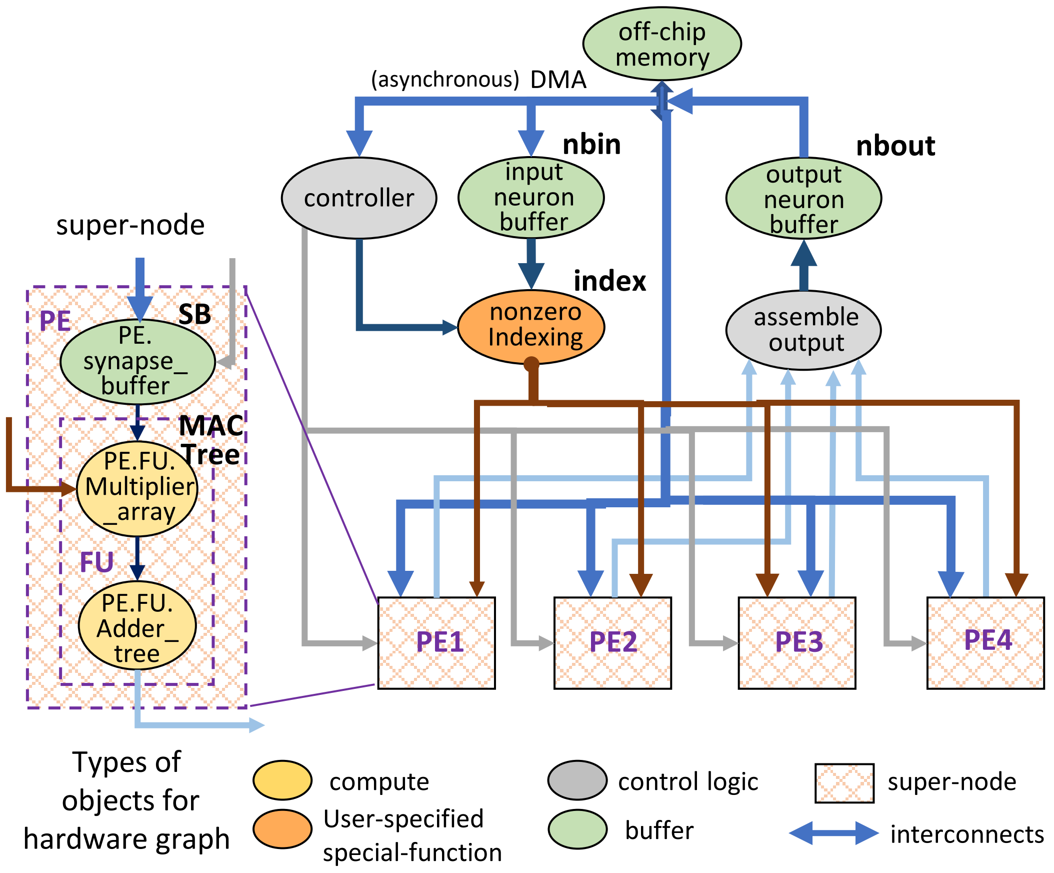


Figure 4.4: Flow graph of Cambricon-X like accelerator (for sparse deep learning models), which efficiently exploits unstructured sparsity of weights. With a flexible flow-graph abstraction and specification in DSAProf, designers or architecture generation tools can easily specify and characterize various DSAs of different hierarchy and specialization.

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 [4, 33, 94, 118, 206, 225], as also observed by [89, 199], 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 4.3.2. Further, since algorithms are compiled as data flow graphs [91, 97, 141, 199, 270] or application code can be decomposed into task graphs [271], they can be conveniently mapped on flow graphs of DSAs, which could be useful in automated architecture search for novel workloads [34].

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 [272], gem5 [90], CACTI [175]), design space exploration [2, 185–187], or application's dataflow graph specifications.

The proposed 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 [200].

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, proposed 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 4.3.2).

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 4.3.6, this chapter presents a case study for defining Cambricon-X like DSA with DSAProf's DSL.

Dataflow-driven Execution Modeling

The proposed 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 in 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/bit-stream). Such controller outputs are usually provided by designers or generated through automation tools for the machine code generation. In section 4.3.6, a case study is discussed 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: The proposed 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).

Library for Modeling Performance of Domain-Specific Architectures

This work develops 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 [199], Accelergy [89], CACTI [175], which can be integrated to proposed infrastructure for area/power models of the components. Next the chapter describes some of the preliminary components that are modeled by the proposed library.

Off-chip memory. This work models a dummy banked DRAM for off-chip accesses. For off-chip data transfers, a DMA engine is modeled that can be interfaced with multiple on-chip buffers, accessing them one at a time. A standard latency model is followed that considers latency for initiating the transfer and burst communication over maximum bandwidth available [154]. External models such as CACTI are used for accurate estimation of latency parameters for a broad range of off-chip memory configurations.

On-chip memory. Multi-port and multi-bank SRAM buffers or register files are modeled 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, external models such as CACTI are used.

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. The proposed library models common interconnects such as unicast, broadcast, mesh, crossbars, or configurable multicast [4, 95]. 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 [121, 273].

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 [33]. 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,

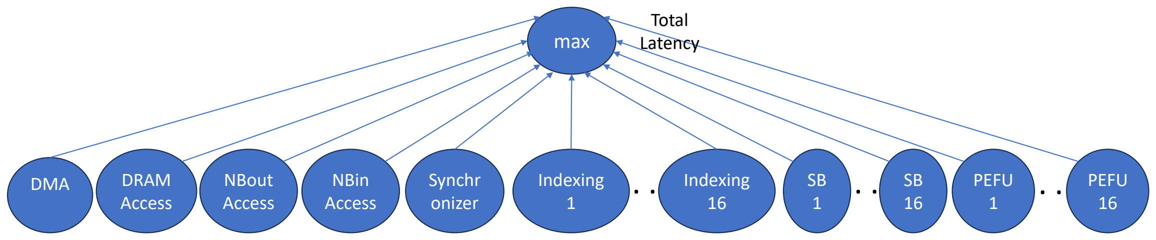


Figure 4.5: With modular and dataflow-driven latency modeling, DSAProf can construct simple latency bottleneck analysis. A sample bottleneck graph of latency is shown here for processing sparse ML layers on Cambricon-X [5] like DSA that is depicted in Fig. 4.4.

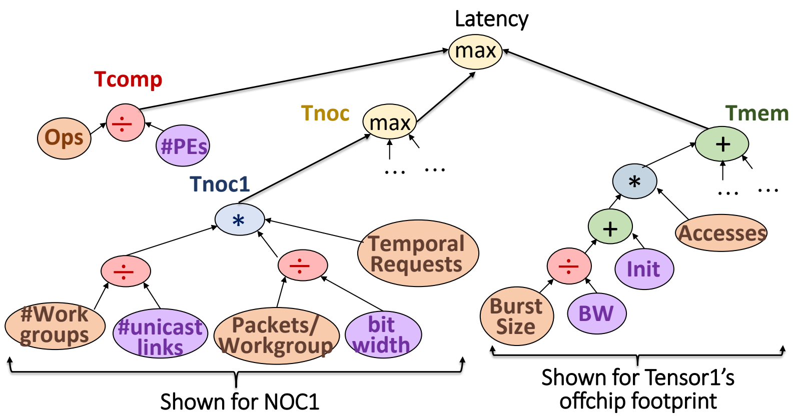


Figure 4.6: Detailed latency bottleneck analysis for Eyeriss [4] like DSA, which accounts for execution activity as well as execution models and design parameters of architectural components. It can help not only identify the root causes of inefficiencies but also mitigation strategies for improving workload mapping and architecture design.

FSMs for address generators, as well as synchronization and delay elements.

Note that the proposed methodology targets performance or energy modeling of the modules, but designers could extend the modeling for other important metrics such as resilience and security of domain-specific architectures [34].

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 Fig. 4.4 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 determined by a maximum value among these factors, as illustrated in Fig. 4.5. Thus, a simple analysis determining the datapath consuming maximum value could serve to identify the primary bottleneck. An actual case study analysis for BERT layers is provided in section 4.3.3.

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, constructing a more detailed bottleneck model is envisioned 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. 4.5, which directly considers maximum/addition among latency values) could be transformed into more accurate bottleneck model as illustrated in Fig. 4.6. Such detailed accounting remains an open opportunity for future research. 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 [273].

4.3.3 Results and Analysis

DSAProf's DSL and execution modeling of common components are used to evaluate latency/energy of DSAs. For preliminary analysis, two popular DSAs are considered for dense/sparse deep learning - Eyeriss [4] and Cambricon-X [5]. Evaluations include executions of operators like convolutions and MLPs for their reported results.

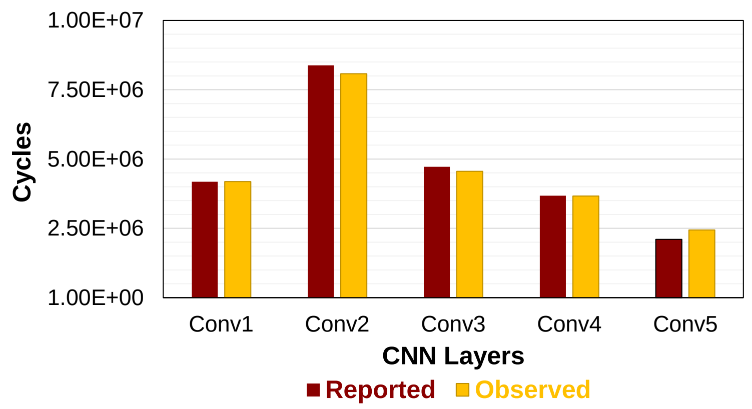


Figure 4.7: For processing AlexNet convolution layers on Eyeriss [4] like DSA, DSAProf estimates the total execution cycles within 1% of the reported results.

Validation 1: Dense DNN Executions

Eyeriss-like DSA [4] is modeled and its execution is evaluated, comparing their reported results, e.g., AlexNet for ImageNet classification [274]. The outputs from controller are configured as per mapping-style reported in [4] for their row-stationary dataflow. Fig. 4.7 shows the comparison of execution cycles based on the processing time reported in [4] and the estimated execution cycles. It is found that the proposed estimations closely matched execution cycles of the architecture [4], with a difference of about 1% in the total execution cycles. It was infeasible to validate energy consumption against [4], since it did not report energy consumption when executing layers with reported mapping configurations.

The evaluations also included 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 [86]. Such estimates for execution cycles and energy consumption matched closely with those reported by dMazeRunner.

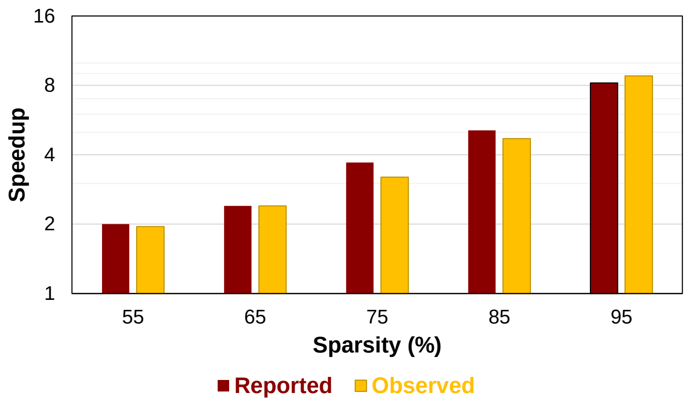


Figure 4.8: For accelerating fully-connected (MLP) layers of sparse DNNs on Cambricon-X [5] like DSA, DSAProf estimates performance speedups (over dense DNNs) within 6% of the reported results.

Validation 2: Sparse DNN Executions

This section demonstrates 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 [4], the library was extended 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 [5], estimated performance was validated for various sparsity in weights of classification (MLP) layers and compared the results with reported values. Fig. 4.8 compares estimated speedup by DSAProf with the reported speedups in [5], when evaluating a sparse VGG-16 classification layer for various

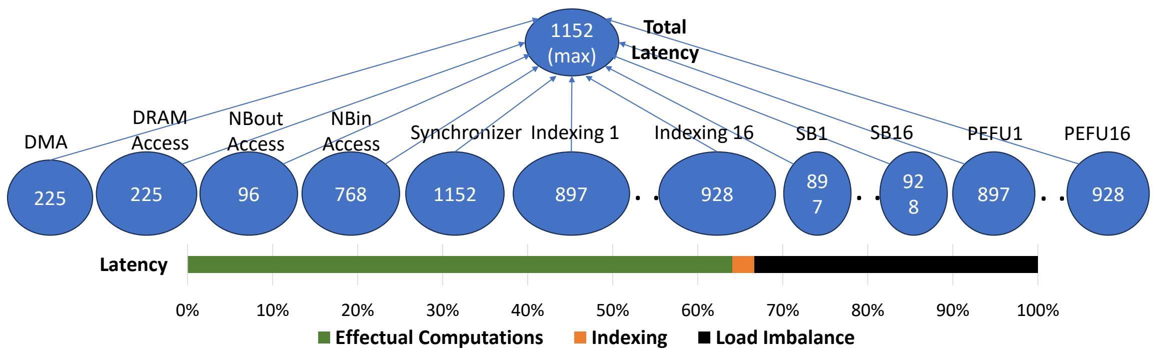


Figure 4.9: DSAProf's simple bottleneck analysis of latency (execution cycles) for processing a BERT encoder layer with 92% weight sparsity on a Cambricon-X [5] like DSA. Overheads of non-zero extraction and imbalance caused by irregular sparsity leads to 56% excess execution time beyond effectual computations. Such automated characterization helps determine the underlying execution inefficiencies for further optimization.

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 [5], the energy consumption could not be compared directly. However, when using DRAM power/energy estimates for a 28 or 45nm technology [122, 177], it is found that total energy for Cambricon-X is heavily consumed by off-chip memory accesses, as also reported in [5].

Bottleneck Characterization

This section shows an automated characterization of Cambricon-X like accelerator for BERT layers [14] (e.g., encoder classification). As discussed previously in section 4.3.2, 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. 4.9 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 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 [33]. 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 [33, 182]) to 8×.

4.3.4 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 [89] and DSAGen [199] 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 [199] and Accelergy [89] 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 [199] requires cycle-level simulation of the functionality or hardware synthesis of each component [199], which is highly time-consuming. Frameworks for accelerator generation such as MAGNet [75] and AutoDNNChip [275] could estimate the performance/energy, but only for a fixed template architecture.

Automated Execution Modeling of Domain-Specific Architectures: The cost models of SECDA [160] and TVM/VTA [124] 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 [121], SCALE-Sim [46], and those of Timeloop [47], dMazeRunner [40], and Interstellar [177] 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 [157, 171], FPGA-based HLS [159, 160], overlapping microarchitectural events [172], power outage [173], and recently for deep learning accelerators [273]. However, such characterization efforts are largely manual, and they are too specific for their target processor architecture. For instance, AutoDSE [159] and SECDA [160] 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.

4.3.5 Impact on Sustainability

Improving sustainability of computing-systems involves design-process and processor-manufacturing. Even for domain-specific accelerators(DSAs), expert-designers develop system-stack specific to architecture-template. It includes architectural-models for energy/performance and functionality-simulators. As workloads change and new architecture-templates are developed, designers must create new architectural-modeling tools from scratch. It is time/resources-intensive; limits sustainability and designer-productivity; increases carbon-footprints of DSAs. Plus, tools designed for specific architecture-templates are difficult to extend/reuse for exploring broader design-space, limiting design-efficiency. The proposed approach uses accelerator abstraction (flow graph) for automatic and modular accelerator modeling/characterization, spanning wide range of architectural-features. Thus, designers can define/analyze accelerators in 10s of lines-of-code (vs.1000s) and build-upon common methodology/infrastructure.

4.3.6 Case Study: Analyzing Sparse GEMMs on Cambricon-X like DSA

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 4.3.2. Listing in Fig. 4.10 shows such definition for Cambricon-X like DSA depicted in Fig. 4.4. With the proposed DSL constructs, DSAs can be defined simply in a few tens of lines of code.

Mapping for the DSA

Listing in Fig. 4.11 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. 4.12.

Figure 4.10: 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: 1 synapse buffer, 1 function unit; 1 central neuron indexing subunit
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))
# sync. neuron broadcast
dfg.add_node('synchronizer', synchronizer(num_inputs=num_PEs))
# 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 4.11: 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 4.12: 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
```

4.4 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. This chapter proposes 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 the proposed 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.

Chapter 5

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation advocated for an effective design methodology towards designing efficient next-generation accelerators in an agile, explainable manner. Firstly, formulation of a comprehensive hardware/software codesign space for architectures is required. Holistic mapping space formulation for spatial and temporal execution enables determining adaptive dataflows, and inclusion of a broad range of architectures allows exploring efficient, constraints-meeting solutions. Second, bottleneck characterization of obtained solutions and bottleneck model-guided design optimization brings explainability in the design exploration process. It helps reason about the effectiveness of sampled solutions during the exploration process, systematically reducing execution costs. Lastly, the design and characterization tools need to be developed around an accelerator abstraction (e.g., a flow graph), which introduces modular and extensible development for a broad range of domain-specific architectures. Next, several open challenges and research directions are discussed that can help achieve better designs and advance the automation further in designing efficient domain-specific architectures.

• **Automating construction of bottleneck models and bottleneck mitigation for arbitrary or large-scale domain-specific systems:** For expert-defined cost models, such as those of DNN accelerators, manual bottleneck analysis by designers with first-hand domain information can be possible. When domain-specific architectures can be described and evaluated as flow graphs, the analysis of costs/bottlenecks may be automated by specifying the bottleneck models of architectural components as graph of their analytical costs and having a centralized methodology for integrating individual graph models into overall bottleneck model.

There can be scenarios where designers may want to optimize the application processing for off-the-shelf processors or can only access pre-existing design models and simulators that are large-scale or complicated. In such cases, designers may not be able to provide the domain-information for constructing bottleneck models and available designs or large-scale simulation models (e.g., in C++/RTL) need to be used for deriving bottleneck models. Hence, learning-based approaches can be developed [201], which could be broadly applicable, while still leveraging the proposed structure/organization of the bottleneck models and their usage in gray-box/white-box design space exploration. Graph-based ML models or self-supervised ML models can be more suitable, including but not limited to decision trees, graph neural networks, and reinforcement learning. Likewise, bottleneck mitigation in complicated scenarios can be estimated using gray-box optimization functions that approximate the relevance and contributions of each parameter to the total cost [202] or through surrogates [103].

• **Automating the mapping space formulation and code generation for arbitrary domain-specific architectures:** The compiler for a flow-graph based architectural description and design methodology needs to take the flow graph representation of a DSA, dataflow graph of the targeted workload-functionality, and provide the optimized machine code (actual/virtual ISA). Its design would need to deal with several challenges, including (but not limited to) developing IRs/DFGs for DSA components and generating overall functionality and ensuring its correctness, checking the compatibility of workloads for execution on a DSA and apply necessary transformations to workload's source code and/or input data for correctness and efficiency, developing virtual ISA for simulation and as a common front for lowering vendor-specific ISAs, and programming/synthesizing accelerator host controller for non-accelerated code and supporting control logic. Further, there remains opportunities for identifying data/code transformations that could make the programs amenable to accelerated execution. For instance, source code of a workload may be written in a way that it is not executable on the accelerator but applying some transformation can make it possible to execute the workload's functionality on the DSA (e.g., identifying and transforming convolution into matrix multiplication).

While some tasks have been investigated in the mainstream compiler research for individual DSAs, designing a compiler for a flow-graph representation faces unique challenges/steps involving development of future methodologies and representations that can be applied to families of DSAs for different domains.

• **Automated, template-free architecture exploration for domain workloads:** As new kinds of domain workloads emerge or become execution-critical for computing tasks (graph learning, federated learning, Fourier transformations, reinforcement learning), automatic exploration of suitable hardware template architectures can be useful. This differs from today's scenarios, where experts manually come up with a suitable template from scratch or follow templates developed previously (e.g., systolic arrays). This could be addressed by developing a Design Space Description Language (DSDL) for architecture-level exploration [34, 200]. It can build upon existing libraries for accelerator designs that could provide cost modeling of accelerator's components or their compilation support. It also requires developing constructs for specifying architectural space for hardware components, rules for constructing feasible and efficient architectures from components, etc. Once such a language can help explore novel architectures from scratch, the vast space of hardware components and their dataflows would make agile exploration of efficient and novel architectures challenging. Therefore, it needs to be equipped with AI-based techniques such that it focuses on formulating feasible and effectual hardware architectures and their dataflows – for both when defining the design space to begin with (off-line pruning) and when formulating candidate hardware architectures during exploration (on-the-fly pruning).

REFERENCES

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

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

[3] Google LLC. Coral Edge TPU Accelerator. `https://coral.ai/products/accelerator-module`.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2016.

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

[6] William J Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63(7):48–57, 2020.

[7] Norman P Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Communications of the ACM*, 61(9):50–59, 2018.

[8] Yann LeCun. 1.1 deep learning hardware: Past, present, and future. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 12–19. IEEE, 2019.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[10] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114, 2019.

[11] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[12] Liang-Chieh Chen, George Papandreou, Florian Schroff, and Hartwig Adam. Rethinking atrous convolution for semantic image segmentation. *arXiv preprint arXiv:1706.05587*, 2017.

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 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)*, pages 4171–4186, 2019.

[15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[16] Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[18] Jongsoo Park, Maxim Naumov, Protonu Basu, Summer Deng, Aravind Kalaiah, Daya Khudia, James Law, Parth Malani, Andrey Malevich, Satish Nadathur, et al. Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications. *arXiv preprint arXiv:1811.09886*, 2018.

[19] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[20] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen Awm Van Der Laak, Bram Van Ginneken, and Clara I Sánchez. A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88, 2017.

[21] Ying Wei, Jun Zhou, Yin Wang, Yinggang Liu, Qingsong Liu, Jiansheng Luo, Chao Wang, Fengbo Ren, and Li Huang. A review of algorithm & hardware design for ai-based biomedical applications. *IEEE transactions on biomedical circuits and systems*, 14(2):145–163, 2020.

[22] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. Exascale deep learning for climate analytics. In

SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pages 649–660. IEEE, 2018.

[23] Nesma M Rezk, Madhura Purnaprajna, Tomas Nordström, and Zain Ul-Abdin. Recurrent neural networks: an embedded computing perspective. *IEEE Access*, 8:57967–57996, 2020.

[24] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. Deep learning in mobile and wireless networking: A survey. *IEEE Communications Surveys & Tutorials*, 2019.

[25] Colby R Banbury, Vijay Janapa Reddi, Max Lam, William Fu, Amin Fazel, Jeremy Holleman, Xinyuan Huang, Robert Hurtado, David Kanter, Anton Lokhmotov, et al. Benchmarking tinyml systems: Challenges and direction. *arXiv preprint arXiv:2003.04821*, 2020.

[26] Jeffrey Dean. 1.1 the deep learning revolution and its implications for computer architecture and chip design. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 8–14. IEEE, 2020.

[27] Kunle Olukotun. Designing computer systems for software 2.0. In *Presentation at 2018 Conference on Neural Information Processing Systems*, 2018.

[28] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[29] Bruce Fleischer, Sunil Shukla, Matthew Ziegler, Joel Silberman, Jinwook Oh, Vijavalakshmi Srinivasan, Jungwook Choi, Silvia Mueller, Ankur Agrawal, Tina Babinsky, et al. A scalable multi-teraops deep learning processor core for ai trainina and inference. In *2018 IEEE Symposium on VLSI Circuits*, pages 35–36. IEEE, 2018.

[30] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[31] Xuan Yang, Mingyu Gao, Jing Pu, Ankita Nayak, Qiaoyi Liu, Steven Emberton Bell, Jeff Ou Setter, Kaidi Cao, Heonjae Ha, Christos Kozyrakis, et al. Dnn dataflow choice is overrated. *arXiv preprint arXiv:1809.04070*, 2018.

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

[33] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights. *Proceedings of the IEEE*, 109(10):1706–1752, 2021.

[34] Shail Dave, Alberto Marchisio, Muhammad Abdullah Hanif, Amira Guesmi, Aviral Shrivastava, Ihsen Alouani, and Muhammad Shafique. Special Session: Towards an Agile Design Methodology for Efficient, Reliable, and Secure ML Systems. In *2022 IEEE 40th VLSI Test Symposium (VTS)*, pages 1–14. IEEE, 2022.

[35] Colby Banbury, Chuteng Zhou, Igor Fedorov, Ramon Matas, Urmish Thakker, Dibakar Gope, Vijay Janapa Reddi, Matthew Mattina, and Paul Whatmough. Micronets: Neural network architectures for deploying tinyml applications on commodity microcontrollers. *Proceedings of Machine Learning and Systems*, 3:517–532, 2021.

[36] Udit Gupta, Young Geun Kim, Sylvia Lee, Jordan Tse, Hsien-Hsin S Lee, Gu-Yeon Wei, David Brooks, and Carole-Jean Wu. Chasing carbon: The elusive environmental footprint of computing. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 854–867. IEEE Computer Society, 2021.

[37] A report for US National Science Foundation (NSF) On Sustainability in Computing Based on NSF Sponsored Workshop Series on Sustainability in Computing. 2023. `https://nsf-suscomp.org/`.

[38] Jeff Jun Zhang, Parul Raj, Shuayb Zarar, Amol Ambardekar, and Siddharth Garg. Compact: On-chip compression of activations for low power systolic array based cnn acceleration. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):47, 2019.

[39] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278. IEEE, 2016.

[40] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. Dmazerunner: Executing perfectly nested loops on dataflow accelerators. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–27, 2019.

[41] Dan Zhang, Safeen Huda, Ebrahim Songhori, Kartik Prabhu, Quoc Le, Anna Goldie, and Azalia Mirhoseini. 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*, pages 27–42, 2022.

[42] Nvidia deep learning accelerator (nvdla). `http://nvdla.org`, 2018.

[43] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015.

[44] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM, 2015.

[45] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 367–379. IEEE Press, 2016.

[46] Scale-sim. `https://github.com/ARM-software/SCALE-Sim`. Accessed: November 5, 2018.

[47] 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. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 304–315. IEEE, 2019.

[48] Nirmal Prajapati, Sanjay Rajopadhye, Hristo Djidjev, Nandakishore Santhi, Tobias Grosser, and Rumen Andonov. Optimization approach to accelerator codesign. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(6):1300–1313, 2019.

[49] Christian Heidorn, Frank Hannig, and Jürgen Teich. Design space exploration for layer-parallel execution of convolutional neural networks on cgras. In *Proceedings of the 23th International Workshop on Software and Compilers for Embedded Systems*, pages 26–31, 2020.

[50] David Koeplinger, Christina Delimitrou, Raghu Prabhakar, Christos Kozyrakis, Yaqi Zhang, and Kunle Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *Proceedings of the 43rd International Symposium on Computer Architecture*, page 115–127. IEEE Press, 2016.

[51] Jie Wang and Jason Cong. Search for optimal systolic arrays: A comprehensive automated exploration framework and lessons learned. *arXiv preprint arXiv:2111.14252*, 2021.

[52] Oliver Bringmann, Wolfgang Ecker, Ingo Feldner, Adrian Frischknecht, Christoph Gerum, Timo Hämäläinen, Muhammad Abdullah Hanif, Michael J Klaiber, Daniel Mueller-Gritschneder, Paul Palomero Bernardo, et al. Automated hw/sw co-design for edge ai: State, challenges and steps ahead: Special session paper. In *2021 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 11–20. IEEE, 2021.

[53] Ye Yu, Yingmin Li, Shuai Che, Niraj K Jha, and Weifeng Zhang. Software-defined design space exploration for an efficient dnn accelerator architecture. *IEEE Transactions on Computers*, 70(1):45–56, 2020.

[54] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.

[55] Liang Wang and Kevin Skadron. Lumos+: Rapid, pre-rtl design space exploration on accelerator-rich heterogeneous architectures with reconfigurable logic. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 328–335. IEEE, 2016.

[56] Takuya Kojima, Nguyen Anh Vu Doan, and Hideharu Amano. Genmap: A genetic algorithmic approach for optimizing spatial mapping of coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(11):2383–2396, 2020.

[57] Hongxiang Fan, Martin Ferianc, Zhiqiang Que, He Li, Shuanglong Liu, Xinyu Niu, and Wayne Luk. Algorithm and hardware co-design for reconfigurable cnn accelerator. In *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 250–255. IEEE, 2022.

[58] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. Lisa: Graph neural network based portable mapping on spatial accelerators. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2022.

[59] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47. IEEE, 2016.

[60] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. Cgra-me: A unified framework for cgra modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 184–189, 2017.

[61] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. A case for efficient accelerator design space exploration via bayesian optimization. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2017.

[62] Qingcheng Xiao, Size Zheng, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, and Yun Liang. Hasco: Towards agile hardware and software co-design for tensor computation. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1055–1068. IEEE, 2021.

[63] Maryam Parsa, Aayush Ankit, Amirkoushyar Ziabari, and Kaushik Roy. Pabo: Pseudo agent-based multi-objective bayesian hyperparameter optimization for

efficient neural accelerator design. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[64] Atefeh Mehrabi, Aninda Manocha, Benjamin C Lee, and Daniel J Sorin. Prospector: Synthesizing efficient accelerators via statistical learning. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020.

[65] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. Confuciux: Autonomous hardware resource assignment for dnn accelerators using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 622–636. IEEE, 2020.

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

[67] Kanghyun Choi, Deokki Hong, Hojae Yoon, Joonsang Yu, Youngsok Kim, and Jinho Lee. Dance: Differentiable accelerator/network co-exploration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 337–342. IEEE, 2021.

[68] Guihong Li, Sumit K Mandal, Umit Y Ogras, and Radu Marculescu. Flash: Fast neural architecture search with hardware optimization. *ACM Transactions on Embedded Computing Systems (TECS)*, 20(5s):1–26, 2021.

[69] Yujun Lin, Mengtian Yang, and Song Han. Naas: Neural accelerator architecture search. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1051–1056. IEEE, 2021.

[70] Liu Ke, Xin He, and Xuan Zhang. Nnest: Early-stage design space exploration tool for neural network inference accelerators. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 1–6, 2018.

[71] Giulia Santoro, Mario R Casu, Valentino Peluso, Andrea Calimera, and Massimo Alioto. Energy-performance design exploration of a low-power microprogrammed deep-learning accelerator. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1151–1154. IEEE, 2018.

[72] Lei Yang, Zheyu Yan, Meng Li, Hyoukjun Kwon, Liangzhen Lai, Tushar Krishna, Vikas Chandra, Weiwen Jiang, and Yiyu Shi. Co-exploration of neural architectures and heterogeneous asic accelerator designs targeting multiple tasks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[73] Guanwen Zhong, Alok Prakash, Siqi Wang, Yun Liang, Tulika Mitra, and Smail Niar. Design space exploration of fpga-based accelerators with multi-level parallelism. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1141–1146, 2017.

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

[75] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[76] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, et al. 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)*, pages 1–9. IEEE, 2020.

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

[78] Ian Bratt and John Brothers. Arm's first-generation machine learning processor. In *2018 IEEE Hot Chips 30 Symposium (HCS)*, pages 1–27. IEEE Computer Society, 2018.

[79] Eitan Medina and Eran Dagan. Habana labs purpose-built ai inference and training processor architectures: Scaling ai training systems using standard ethernet with gaudi processor. *IEEE Micro*, 40(2):17–24, 2020.

[80] Paolo D'Alberto, Victor Wu, Aaron Ng, Rahul Nimaiyar, Elliott Delaye, and Ashish Sirasao. xdnn: Inference for deep convolutional neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(2):1–29, 2022.

[81] Karam Chatha. Qualcomm® cloud al 100: 12tops/w scalable, high performance and low latency deep learning inference accelerator. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–19. IEEE, 2021.

[82] Jun-Seok Park, Heonsoo Lee, Dongwoo Lee, Jewoo Moon, Suknam Kwon, SangHyuck Ha, MinSeong Kim, Junghun Park, Jihoon Bang, and Sukhwan Lim Inyup Kang. Samsung neural processing unit: An ai accelerator and sdk for flagship mobile ap. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–21. IEEE Computer Society, 2021.

[83] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Bengseng Phuah, et al. Aquabolt-xl: Samsung hbm2-pim with in-memory processing for ml accelerators

and beyond. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–26. IEEE, 2021.

[84] Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. Nvidia ampere architecture in-depth. `https://devblogs.nvidia.com/nvidia-ampere-architecture-in-depth/`, 2020.

[85] Yuan Xie. A Brief Guide of xPU for AI Accelerators. `https://www.sigarch.org/a-brief-guide-of-xpu-for-ai-accelerators/`, April 2018. ACM SIGARCH Computer Architecture Today.

[86] Shail Dave, Aviral Shrivastava, Youngbin Kim, Sasikanth Avancha, and Kyoungwoo Lee. dmazerunner: Optimizing convolutions on dataflow accelerators. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1544–1548. IEEE, 2020.

[87] Bob Iannucci, Aviral Shrivastava, and Mohammad Khayatian. Ticktalk–timing api for dynamically federated cyber-physical systems. *arXiv preprint arXiv:1906.03982*, 2019.

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

[89] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

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

[91] Shail Dave and Aviral Shrivastava. Ccf: A cgra compilation framework, 2018.

[92] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2021.

[93] Shail Dave and Aviral Shrivastava. Automating the architectural execution modeling and characterization of domain-specific architectures. *Semiconductor Research Corporation (SRC) Techcon*, 2023.

[94] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. 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)*, pages 15–28. IEEE, 2018.

[95] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 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):292–308, 2019.

[96] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. C ir cnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408. ACM, 2017.

[97] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ramp: Resource-aware mapping for cgras. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[98] Jungi Lee and Jongeun Lee. Specializing cgras for light-weight convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[99] Jonathan Dickerson, Ioannis Galanis, Zois-Gerasimos Tasoulas, Lincoln Kinley, and Iraklis Anagnostopoulos. Adaptive approximate computing on hardware accelerators targeting internet-of-things. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pages 1–6. IEEE, 2020.

[100] Zhongyuan Zhao, Weiguang Sheng, Qin Wang, Wenzhi Yin, Pengfei Ye, Jinchao Li, and Zhigang Mao. Towards higher performance and robust compilation for cgra modulo scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2201–2219, 2020.

[101] Prasanth Chatarasi, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. Marvel: a data-centric approach for mapping deep learning operators on spatial accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–26, 2021.

[102] Mark Horeni, Pooria Taheri, Po-An Tsai, Angshuman Parashar, Joel Emer, and Siddharth Joshi. Ruby: Improving hardware efficiency for tensor algebra accelerators through imperfect factorization. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 254–266. IEEE, 2022.

[103] Kartik Hegde, Po-An Tsai, Sitao Huang, Vikas Chandra, Angshuman Parashar, and Christopher W Fletcher. 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*, pages 943–958, 2021.

[104] Dennis Rieber, Axel Acosta, and Holger Fröning. Joint program and layout transformations to enable convolutional operators on specialized hardware based on constraint programming. *ACM Transactions on Architecture and Code Optimization (TACO)*, 19(1):1–26, 2021.

[105] Andres Rodriguez. Deep learning systems: Algorithms, compilers, and processors for large-scale production. *Synthesis Lectures on Computer Architecture*, 15(4):1–265, 2020.

[106] Tushar Krishna, Hyoukjun Kwon, Angshuman Parashar, Michael Pellauer, and Ananda Samajdar. Data orchestration in deep learning accelerators. *Synthesis Lectures on Computer Architecture*, 15(3):1–164, 2020.

[107] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.*, 22(241):1–124, 2021.

[108] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. Capstan: A vector rda for sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1022–1035, 2021.

[109] Jong Hoon Shin, Ali Shafiee, Ardavan Pedram, Hamzah Abdel-Aziz, Ling Li, and Joseph Hassoun. Griffin: Rethinking sparse optimization for deep learning architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 861–875. IEEE, 2022.

[110] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Indirection stream semantic register architecture for efficient sparse-dense linear algebra. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1787–1792. IEEE, 2021.

[111] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[112] Liqiang Lu, Jiaming Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 17–25. IEEE, 2019.

[113] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 821–834. ACM, 2019.

[114] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[115] Kamran Khan. Xilinx dnn processor (xdnn), accelerating ai in datacenters. `https://www.xilinx.com/publications/events/developer-forum/2018-frankfurt/accelerating-ai-in-datacenters-xilinx-ml-suite.pdf`, 2018.

[116] Michael Pellauer, Angshuman Parashar, Michael Adler, Bushra Ahsan, Randy Allmon, Neal Crago, Kermin Fleming, Mohit Gambhir, Aamer Jaleel, Tushar Krishna, et al. Efficient control and communication paradigms for coarse-grained spatial architectures. *ACM Transactions on Computer Systems (TOCS)*, 33(3):10, 2015.

[117] Tony Nowatzki, Michael Sartin-Tarm, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. In *ACM SIGPLAN Notices*, volume 48, pages 495–506. ACM, 2013.

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

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

[120] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14. IEEE, 2018.

[121] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. 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*, pages 754–768, 2019.

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

[123] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Shixuan Zheng, Tianyi Lu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. A high energy efficient reconfigurable hybrid neural network processor for deep learning applications. *IEEE Journal of Solid-State Circuits*, 53(4):968–982, 2017.

[124] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX}*

Symposium on Operating Systems Design and Implementation ({OSDI} 18), pages 578–594, 2018.

[125] Jason Cong and Jie Wang. Polysa: polyhedral-based systolic array auto-compilation. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[126] Linghao Song, Jiachen Mao, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Hypar: Towards hybrid parallelism for deep learning accelerator array. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 56–68. IEEE, 2019.

[127] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, & tools.* Pearson Education India, 2007.

[128] Steve Carr, Kathryn S McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. *ACM SIGPLAN Notices*, 29(11):252–262, 1994.

[129] Ilya Issenin, Erik Brockmeyer, Miguel Miranda, and Nikil Dutt. Drdu: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 12(2):15, 2007.

[130] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Acm Sigplan Notices*, volume 43, pages 101–113. ACM, 2008.

[131] Florin Balasa, Per Gunnar Kjeldsberg, Arnout Vandecappelle, Martin Palkovic, Qubo Hu, Hongwei Zhu, and Francky Catthoor. Storage estimation and design space exploration methodologies for the memory management of signal processing applications. *Journal of Signal Processing Systems*, 53(1-2):51, 2008.

[132] B Ramakrishna Rau. Iterative module scheduling: An algorithm for software pipelining loops. In *Proceedings of MICRO-27. The 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 63–74. IEEE, 1994.

[133] Zhongyuan Zhao, Yantao Liu, Weiguang Sheng, Tushar Krishna, Qin Wang, and Zhigang Mao. Optimizing the data placement and transformation for multi-bank cgra computing system. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1087–1092. IEEE, 2018.

[134] Frank Bouwens, Mladen Berekovic, Bjorn De Sutter, and Georgi Gaydadjiev. Architecture enhancements for the adres coarse-grained reconfigurable array. In *International Conference on High-Performance Embedded Architectures and Compilers*, pages 66–81. Springer, 2008.

[135] Taewook Oh, Bernhard Egger, Hyunchul Park, and Scott Mahlke. Recurrence cycle aware modulo scheduling for coarse-grained reconfigurable architectures. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 21–30, 2009.

[136] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. Operation and data mapping for cgras with multi-bank memory. *ACM Sigplan Notices*, 45(4):17–26, 2010.

[137] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. Egra: A coarse grained reconfigurable architectural template. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 19(6):1062–1074, 2010.

[138] Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, Jonghee W Yoon, Doosan Cho, and Yunheung Paek. High throughput data mapping for coarse-grained reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of integrated circuits and systems*, 30(11):1599–1609, 2011.

[139] Shouyi Yin, Xianqing Yao, Dajiang Liu, Leibo Liu, and Shaojun Wei. Memory-aware loop mapping on coarse-grained reconfigurable architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1895–1908, 2015.

[140] Satyajit Das, Kevin JM Martin, Philippe Coussy, Davide Rossi, and Luca Benini. Efficient mapping of cdfg onto coarse-grained reconfigurable array architectures. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 127–132. IEEE, 2017.

[141] Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, and Reiley Jeyapaul. Laser: A hardware/software approach to accelerate complicated loops on cgras. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1069–1074. IEEE, 2018.

[142] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. Cascade: High throughput data streaming via decoupled access-execute cgra. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s):1–26, 2019.

[143] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. Ureca: Unified register file for cgras. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1081–1086. IEEE, 2018.

[144] Cheng Tan, Tong Geng, Chenhao Xie, Nicolas Bohm Agostini, Jiajia Li, Ang Li, Kevin Barker, and Antonino Tumeo. Dynpac: Coarse-grained, dynamic, and partially reconfigurable array for streaming applications. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 33–40. IEEE, 2021.

[145] Arthur Stoutchinin, Francesco Conti, and Luca Benini. Optimally scheduling cnn convolutions for efficient memory access. *arXiv preprint arXiv:1902.01492*, 2019.

[146] Zhongyuan Zhao, Hyoukjun Kwon, Sachit Kuhar, Weiguang Sheng, Zhigang Mao, and Tushar Krishna. mrna: Enabling efficient mapping space exploration for a reconfiguration neural accelerator. In *2019 IEEE International Symposium*

on Performance Analysis of Systems and Software (ISPASS), pages 282–292. IEEE, 2019.

[147] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[148] Hyoukjun Kwon, Michael Pellauer, and Tushar Krishna. MAESTRO: an open-source infrastructure for modeling dataflows within deep learning accelerators. *CoRR*, abs/1805.02566, 2018.

[149] Xuan Yang et al. Dnn energy model and optimizer. `https://github.com/xuanyoya/CNN-blocking/tree/dev`. Accessed: November 5, 2018.

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

[151] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[152] Kartik Hegde, Rohit Agrawal, Yulun Yao, and Christopher W Fletcher. Morph: Flexible acceleration for 3d cnn-based video understanding. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 933–946. IEEE, 2018.

[153] fmincon. `https://www.mathworks.com/help/optim/ug/fmincon.html`. Accessed: November 5, 2018.

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

[155] Qijing Huang, Charles Hong, John Wawrzynek, Mahesh Subedar, and Yakun Sophia Shao. Learning a continuous and reconstructible latent space for hardware accelerator design. In *2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 277–287. IEEE, 2022.

[156] Nan Wu, Yuan Xie, and Cong Hao. Ironman: Gnn-assisted design space exploration in high-level synthesis via reinforcement learning. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI*, pages 39–44, 2021.

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

[158] Gennette Gill and Montek Singh. Bottleneck analysis and alleviation in pipelined systems: A fast hierarchical approach. In *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pages 195–205. IEEE, 2009.

[159] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4), 2022.

[160] Jude Haris, Perry Gibson, José Cano, Nicolas Bohm Agostini, and David Kaeli. 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, 2021.

[161] Andy D. Pimentel. Exploring exploration: A tutorial introduction to embedded systems design space exploration. *IEEE Design & Test*, 34(1), 2017.

[162] Soonhoi Ha, Jürgen Teich, Christian Haubelt, Michael Glaß, Tulika Mitra, Rainer Dömer, Petru Eles, Aviral Shrivastava, Andreas Gerstlauer, and Shuvra S Bhattacharyya. Introduction to hardware/software codesign. *Handbook of Hardware/Software Codesign*, pages 3–26, 2017.

[163] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[164] Ayse K. Coskun, Jose L. Ayala, David Atienza, Tajana Simunic Rosing, and Yusuf Leblebici. Dynamic thermal management in 3d multicore architectures. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 1410–1415, 2009.

[165] Björn Forsberg, Maxim Mattheeuws, Andreas Kurth, Andrea Marongiu, and Luca Benini. A synergistic approach to predictable compilation and scheduling on commodity multi-cores. In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 108–118, 2020.

[166] Aryan Deshwal, Nitthilan Kanappan Jayakodi, Biresh Kumar Joardar, Janardhan Rao Doppa, and Partha Pratim Pande. Moos: A multi-objective design space exploration and optimization framework for noc enabled manycore systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 18(5s), 2019.

[167] Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. Mapping on multi/many-core systems: Survey of current and emerging trends. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, 2013.

[168] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. Lattice-traversing design space exploration for high level synthesis. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 210–217. IEEE, 2018.

[169] Joydeep Dey and Sudeep Pasricha. Robust perception architecture design for automotive cyber-physical systems. *arXiv preprint arXiv:2205.08067*, 2022.

[170] Linyan Mei, Pouya Houshmand, Vikram Jain, Sebastian Giraldo, and Marian Verhelst. Zigzag: Enlarging joint architecture-mapping design space exploration for dnn accelerators. *IEEE Transactions on Computers*, 70(8):1160–1174, 2021.

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

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

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

[174] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 97–108. IEEE, 2014.

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

[176] David L Poole and Alan K Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010.

[177] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. 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*, pages 369–383, 2020.

[178] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

[179] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for mobilenetv3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 1314–1324. IEEE.

[180] YOLOv5 Classification. `https://pytorch.org/hub/ultralytics_yolov5`, 2019.

[181] Alexei Baevski, Yuhao Zhou, Abdelrahman Mohamed, and Michael Auli. wav2vec 2.0: A framework for self-supervised learning of speech representations. *Advances in Neural Information Processing Systems*, 33:12449–12460, 2020.

[182] 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. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[183] ARM Machine Learning Processor. `https://en.wikichip.org/wiki/arm_holdings/microarchitectures/mlp`, 2018.

[184] Intel Nervana NNP-I 100. `https://en.wikichip.org/wiki/nervana/nnp/nnp-i_1100`, 2019.

[185] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, Ilhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[186] scikit-opt. `github.com/guofei9987/scikit-opt/`, 2019.

[187] Fernando Nogueira. Bayesian Optimization: Open source constrained global optimization tool for Python, 2014–.

[188] Coral. Edge TPU Performance Benchmarks. `https://coral.ai/docs/edgetpu/benchmarks/`.

[189] Kiran Seshadri, Berkin Akin, James Laudon, Ravi Narayanaswami, and Amir Yazdanbakhsh. An evaluation of edge tpu accelerators for convolutional neural networks. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 79–91. IEEE, 2022.

[190] Enrico Russo, Maurizio Palesi, Davide Patti, Salvatore Monteleone, Giuseppe Ascia, and Vincenzo Catania. Multi-objective end-to-end design space exploration of parameterized dnn accelerators. *IEEE Internet of Things Journal*, 2022.

[191] Sheng-Chun Kao and Tushar Krishna. Gamma: Automating the hw mapping of dnn models on accelerators via genetic algorithm. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[192] Sheng-Chun Kao, Angshuman Parashar, Po-An Tsai, and Tushar Krishna. Demystifying map space exploration for npus. *arXiv preprint arXiv:2210.03731*, 2022.

[193] Qijing Huang, Aravind Kalaiah, Minwoo Kang, James Demmel, Grace Dinh, John Wawrzynek, Thomas Norell, and Yakun Sophia Shao. Cosa: Scheduling by constrained optimization for spatial accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 554–566. IEEE, 2021.

[194] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. Heterogeneous dataflow accelerators for multi-dnn workloads. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–83. IEEE, 2021.

[195] Xiaofan Zhang, Yuan Ma, Jinjun Xiong, Wen-Mei W. Hwu, Volodymyr Kindratenko, and Deming Chen. Exploring hw/sw co-design for video analysis on cpu-fpga heterogeneous systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(6):1606–1619, 2022.

[196] Shixuan Zheng, Xianjue Zhang, Leibo Liu, Shaojun Wei, and Shouyi Yin. Atomic dataflow based graph-level workload orchestration for scalable dnn accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 475–489.

[197] Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, and P Sadayappan. Comprehensive accelerator-dataflow co-design optimization for convolutional neural networks. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 325–335.

[198] Sheng-Chun Kao, Michael Pellauer, Angshuman Parashar, and Tushar Krishna. Digamma: Domain-aware genetic algorithm for hw-mapping co-optimization for dnn accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 232–237. IEEE, 2022.

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

[200] Shail Dave and Aviral Shrivastava. 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).

[201] Aviral Kumar, Amir Yazdanbakhsh, Milad Hashemi, Kevin Swersky, and Sergey Levine. Data-driven offline optimization for architecting hardware accelerators. In *International Conference on Learning Representations*, 2021.

[202] Roberto Santana. Gray-box optimization and factorized distribution algorithms: where two worlds collide. *arXiv preprint arXiv:1707.03093*, 2017.

[203] B. L. Deng, G. Li, S. Han, L. Shi, and Y. Xie. Model compression and hardware acceleration for neural networks: A comprehensive survey. *Proceedings of the IEEE*, 108(4):485–532, 2020.

[204] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[205] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015.

[206] Lisa Wu, Andrea Lottarini, Timothy K Paine, Martha A Kim, and Kenneth A Ross. 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*, pages 255–268, 2014.

[207] Yatish Turakhia, Gill Bejerano, and William J Dally. Darwin: A genomics co-processor provides up to 15,000 x acceleration on long read assembly. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213, 2018.

[208] Daichi Fujiki, Arun Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. Genax: A genome sequencing accelerator. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 69–82. IEEE, 2018.

[209] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. A scalable high-bandwidth architecture for lossless compression on fpgas. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 52–59. IEEE, 2015.

[210] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.

[211] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*, pages 2074–2082, 2016.

[212] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020*

IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 58–70. IEEE, 2020.

[213] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. Sparse gpu kernels for deep learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2020.

[214] Patrick Judd, Alberto Delmas, Sayeh Sharify, and Andreas Moshovos. Cnvlutin2: Ineffectual-activation-and-weight-free deep neural network computing. *arXiv preprint arXiv:1705.00125*, 2017.

[215] Shixuan Zheng, Yonggang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. An efficient kernel transformation architecture for binary-and ternary-weight neural network inference. In *Proceedings of the 55th Annual Design Automation Conference*, page 137. ACM, 2018.

[216] Rastislav Struharik, Bogdan Vukobratović, Andrea Erdeljan, and Damjan Rakanović. Conna–compressed cnn hardware accelerator. In *2018 21st Euromicro Conference on Digital System Design (DSD)*, pages 365–372. IEEE, 2018.

[217] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE, 2017.

[218] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W Keckler, and Zhengya Zhang. Snap: A 1.67—21.55 tops/w sparse neural acceleration processor for unstructured sparse deep neural network inference in 16nm cmos. In *2019 Symposium on VLSI Circuits*, pages C306–C307. IEEE, 2019.

[219] Hyeong-Ju Kang. Accelerator-aware pruning for convolutional neural networks. *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.

[220] Asit K Mishra, Eriko Nurvitadhi, Ganesh Venkatesh, Jonathan Pearce, and Debbie Marr. Fine-grained accelerators for sparse machine learning workloads. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 635–640. IEEE, 2017.

[221] Bert Moons, Roel Uytterhoeven, Wim Dehaene, and Marian Verhelst. 14.5 envision: A 0.26-to-10tops/w subword-parallel dynamic-voltage-accuracy-frequency-scalable convolutional neural network processor in 28nm fdsoi. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 246–247. IEEE, 2017.

[222] Jinsu Lee, Juhyoung Lee, Donghyeon Han, Jinmook Lee, Gwangtae Park, and Hoi-Jun Yoo. 7.7 lnpu: A 25.3 tflops/w sparse deep-neural-network learning processor with fine-grained mixed precision of fp8-fp16. In *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 142–144. IEEE, 2019.

[223] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. Sparten: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165. ACM, 2019.

[224] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. Squeezeflow: A sparse cnn accelerator exploiting concise convolution rules. *IEEE Transactions on Computers*, 68(11):1663–1677, 2019.

[225] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333. ACM, 2019.

[226] Zhe Yuan, Jinshan Yue, Huanrui Yang, Zhibo Wang, Jinyang Li, Yixiong Yang, Qingwei Guo, Xueqing Li, Meng-Fan Chang, Huazhong Yang, et al. Sticker: A 0.41-62.1 tops/w 8bit neural network processor with multi-sparsity compatible convolution arrays and online tuning acceleration for fully connected layers. In *2018 IEEE Symposium on VLSI Circuits*, pages 33–34. IEEE, 2018.

[227] Sharad Chole, Ramteja Tadishetti, and Sree Reddy. Sparsecore: An accelerator for structurally sparse cnns. In *SysML Conference*, 2018.

[228] Leonid Yavits and Ran Ginosar. Accelerator for sparse machine learning. *IEEE Computer Architecture Letters*, 17(1):21–24, 2017.

[229] Alessandro Aimar, Hesham Mostafa, Enrico Calabrese, Antonio Rios-Navarro, Ricardo Tapiador-Morales, Iulia-Alexandra Lungu, Moritz B Milde, Federico Corradi, Alejandro Linares-Barranco, Shih-Chii Liu, et al. Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps. *IEEE transactions on neural networks and learning systems*, 30(3):644–656, 2018.

[230] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84, 2017.

[231] Adam Page, Ali Jafari, Colin Shea, and Tinoosh Mohsenin. Sparcnet: A hardware accelerator for efficient deployment of sparse convolutional networks. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 13(3):1–32, 2017.

[232] Liqiang Lu and Yun Liang. Spwa: an efficient sparse winograd convolutional neural networks accelerator on fpgas. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[233] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. *ACM SIGARCH Computer Architecture News*, 44(3):1–13, 2016.

[234] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. Deltarnn: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 21–30, 2018.

[235] Bahar Asgari, Ramyad Hadidi, Hyesoon Kim, and Sudhakar Yalamanchili. Eridanus: Efficiently running inference of dnns using systolic arrays. *IEEE Micro*, 39(5):46–54, 2019.

[236] HT Kung, Bradley McDanel, and Sai Qian Zhang. Adaptive tiling: Applying fixed-size systolic arrays to sparse convolutional neural networks. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 1006–1011. IEEE, 2018.

[237] Shouyi Yin, Peng Ouyang, Shibin Tang, Fengbin Tu, Xiudong Li, Shixuan Zheng, Tianyi Lu, Jiangyuan Gu, Leibo Liu, and Shaojun Wei. A high energy efficient reconfigurable hybrid neural network processor for deep learning applications. *IEEE Journal of Solid-State Circuits*, 53(4):968–982, 2017.

[238] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks. In *SysML Conference*, 2018.

[239] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zena: Zero-aware neural network accelerator. *IEEE Design & Test*, 35(1):39–46, 2017.

[240] Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: a fast and scalable system architecture for memory-augmented neural networks. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 250–263, 2019.

[241] Bradley McDanel, Sai Qian Zhang, HT Kung, and Xin Dong. Full-stack optimization for accelerating cnns using powers-of-two weights with fpga validation. In *Proceedings of the ACM International Conference on Supercomputing*, pages 449–460, 2019.

[242] Paul N Whatmough, Sae Kyu Lee, David Brooks, and Gu-Yeon Wei. Dnn engine: A 28-nm timing-error tolerant sparse deep neural network processor for iot applications. *IEEE Journal of Solid-State Circuits*, 53(9):2722–2731, 2018.

[243] Ganesh Venkatesh, Eriko Nurvitadhi, and Debbie Marr. Accelerating deep convolutional networks using low-precision and sparsity. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2861–2865. IEEE, 2017.

[244] Zhi-Gang Liu, Paul N Whatmough, and Matthew Mattina. Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference. *IEEE Computer Architecture Letters*, 19(1):34–37, 2020.

[245] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. *arXiv preprint arXiv:2012.09852*, 2020.

[246] Tae Jun Ham, Sung Jun Jung, Seonghak Kim, Young H Oh, Yeonhong Park, Yoonho Song, Jung-Hun Park, Sanghee Lee, Kyoung Park, Jae W Lee, et al. A^3: Accelerating attention mechanisms in neural networks with approximation. *arXiv preprint arXiv:2002.10941*, 2020.

[247] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-tpu: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[248] Gaurav Srivastava, Deepak Kadetotad, Shihui Yin, Visar Berisha, Chaitali Chakrabarti, and Jae-sun Seo. Joint optimization of quantization and structured sparsity for compressed deep neural networks. In *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1393–1397. IEEE, 2019.

[249] Runbin Shi, Peiyan Dong, Tong Geng, Yuhao Ding, Xiaolong Ma, Hayden K-H So, Martin Herbordt, Ang Li, and Yanzhi Wang. Csb-rnn: a faster-than-realtime rnn acceleration framework with compressed structured blocks. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[250] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M Rush, Gu-Yeon Wei, and David Brooks. Masr: A modular accelerator for sparse rnns. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14. IEEE, 2019.

[251] Xi Zeng, Tian Zhi, Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Bingrui Wang, Yuanbo Wen, Chao Wang, Xuehai Zhou, et al. Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. *IEEE Transactions on Computers*, 2020.

[252] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, 2016.

[253] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*, pages 173–182, 2017.

[254] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[255] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools. `http://frostt.io/`, 2017.

[256] Jiajia Li, Mahesh Lakshminarasimhan, Xiaolong Wu, Ang Li, Catherine Olschanowsky, and Kevin Barker. A parallel sparse tensor benchmark suite on cpus and gpus. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 403–404, 2020.

[257] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[258] Dharma Teja Vooturi, Dheevatsa Mudigree, and Sasikanth Avancha. Hierarchical block sparse neural networks. *arXiv preprint arXiv:1808.03420*, 2018.

[259] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018.

[260] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 521–532, 2015.

[261] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 300–314. ACM, 2019.

[262] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Barbara Kreaseck. Sparse tiling for stationary iterative methods. *The International Journal of High Performance Computing Applications*, 18(1):95–113, 2004.

[263] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[264] Jiecao Yu, Andrew Lukefahr, David Palframan, Ganesh Dasika, Reetuparna Das, and Scott Mahlke. Scalpel: Customizing dnn pruning to the underlying hardware parallelism. *ACM SIGARCH Computer Architecture News*, 45(2):548–560, 2017.

[265] Xiaolong Ma, Sheng Lin, Shaokai Ye, Zhezhi He, Linfeng Zhang, Geng Yuan, Sia Huat Tan, Zhengang Li, Deliang Fan, Xuehai Qian, et al. Non-structured dnn weight pruning–is it beneficial in any platform? *arXiv preprint arXiv:1907.02124*, 2019.

[266] Erich Elsen, Marat Dukhan, Trevor Gale, and Karen Simonyan. Fast sparse convnets. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 14629–14638, 2020.

[267] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. *arXiv preprint arXiv:1902.09574*, 2019.

[268] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505–4515, 2019.

[269] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. 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*, pages 751–764, 2017.

[270] Anup Das and Akash Kumar. Dataflow-based mapping of spiking neural networks on neuromorphic hardware. In *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pages 419–422, 2018.

[271] Benjamin R Willis, Aviral Shrivastava, Joshua Mack, Shail Dave, Chaitali Chakrabarti, and John Brunhaver. Cyclebite: Extracting task graphs from unstructured compute-programs. *IEEE Transactions on Computers*, 2023.

[272] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. Pymtl3: A python framework for open-source hardware modeling, generation, simulation, and verification. *IEEE Micro*, 40(4):58–66, 2020.

[273] Shail Dave, Tony Nowatzki, and Aviral Shrivastava. 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)*, 2024.

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

[275] Pengfei Xu, Xiaofan Zhang, Cong Hao, Yang Zhao, Yongan Zhang, Yue Wang, Chaojian Li, Zetong Guan, Deming Chen, and Yingyan Lin. 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*, pages 40–50, 2020.

BIOGRAPHICAL SKETCH

Shail Dave is a final-year Ph.D. candidate in the School of Computing and Augmented Intelligence (SCAI) at Arizona State University. His research techniques and infrastructures enable efficient processing of critical applications like machine learning on hardware accelerators in an agile manner. More specifically, his research develops compilation and mapping optimizations for accelerators, execution cost modeling and bottleneck characterization, language for automated accelerator designs, techniques for efficient dense/sparse tensor computations, accelerator simulators, and exploration of hardware/software codesigns through systematic heuristics and machine learning. His research is regularly published in and referred by the flagship ACM/IEEE conferences and journals in design automation, embedded systems, and computer architecture and invited to premier international and industrial forums.

Shail's industry experiences range from compiler optimizations for wide-scale commodity embedded systems to digital design and verification for FPGA-based accelerators and ASICs. He has piloted several projects in competitive programs with academics and industry researchers, including for heterogeneous and AI computing systems. He is a recipient of several honors, awards, and fellowships, including Silver Medal from ACM for Student Research Competition, Outstanding Research and Teaching Assistant Awards, and Doctoral Dissertation Fellowship from Graduate College at ASU. His research has been featured by various organizations, media, magazines, and social media, including ACM Tech news, Communications of the ACM blog, ASU news, insideHPC, IEEE Bridge and IEEE Eta Kappa Nu (HKN), DeepAI, Hacker news, and Intel Labs Select Publications. Shail regularly serves in mentorship/student programs, vision workshops for future computing systems research, and as a reviewer and organization/program committee member for the leading ACM/IEEE conferences, workshops, and journals.

PUBLICATIONS

1. Explainable-DSE: An Agile and Explainable Exploration of Efficient Hardware/-Software Codesigns of Deep Learning Accelerators Using Bottleneck Analysis. Shail Dave, Tony Nowatzki, Aviral Shrivastava, in ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2024.

2. Cyclebite: Extracting Task Graphs From Unstructured Compute-Programs, Benjamin Willis, Aviral Shrivastava, Joshua Mack, Shail Dave, Chaitali Chakrabarti, John Brunhaver, in IEEE Transactions of Computers (TC), 2023.

3. Automating the Architectural Execution Modeling and Characterization of Domain-Specific Architectures. Shail Dave, Aviral Shrivastava. In Semiconductor Research Corporation TECHCON 2023.

4. Learning-Oriented Reliability Improvement of Computing Systems From Transistor to Application Level. Behnaz Ranjbar, Florian Klemme, Paul R. Genssler, Hussam Amrouch, Jinhyo Jung, Shail Dave, Hwisoo So, Kyongwoo Lee, Aviral Shrivastava, Ji-Yung Lin, Pieter Weckx, Subrat Mishra, Francky Catthoor, Dwaipayan Biswas, Akash Kumar, in Proceedings of the 26th International Conference on Design Automation and Test in Europe (DATE), 2023.

5. Towards an Agile Design Methodology for Efficient, Secure, and Reliable ML Systems. Shail Dave, Alberto Marchisio, Muhammad Abdullah Hanif, Amira Guesmi, Aviral Shrivastava, Ihsen Alouani, Muhammad Shafique, in Proceedings of the 40th IEEE VLSI Test Symposium (VTS), 2022.

6. Design Space Description Language for Automated and Comprehensive Exploration of Next-Gen Hardware Accelerators. Shail Dave and Aviral Shrivastava, in

Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE), co-located with ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022.

7. Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights. Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, Baoxin Li, in Proceedings of the IEEE (PIEEE), volume 109, issue 10, 2021.

8. SPX64: A Scratchpad Memory for General-Purpose Microprocessors. Abhishek Singh, Shail Dave, PanteA Zardoshti, Robert Brotzman, Chao Zhang, Xiaochen Guo, Aviral Shrivastava, Gang Tan, Michael Spear, in ACM Transactions on Architecture and Code Optimization (TACO), Vol. 18, No. 1, 2021.

9. dMazeRunner: Optimizing Convolutions and GEMMs on Dataflow Accelerators. Shail Dave, Aviral Shrivastava, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, in Proceedings of the 45th International Conference on Acoustics, Speech, and Signal Processing (ICASSP), 2020.

10. dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators. Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, Aviral Shrivastava, in ACM Transactions on Embedded Computing Systems (TECS), Vol. 18, No. 5s, 2019 [Special Issue on ESWEEK 2019 - ACM/IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)].

11. RAMP: Resource-Aware Mapping for CGRAs. Shail Dave, Mahesh Balasubramanian, Aviral Shrivastava, in Proceedings of the 55th Annual Design Automation Conference (DAC), 2018.

12. CCF: CGRA Compilation and Simulation Framework. Shail Dave, Aviral Shrivastava, in University Booth Demonstration at the 21st International Conference on Design Automation and Test in Europe (DATE), 2018.

13. LASER: A Hardware/Software Approach to Accelerate Complicated Loops on CGRAs. Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, Reiley Jeyapaul, in Proceedings of the 21st International Conference on Design Automation and Test in Europe (DATE), 2018.

14. URECA: A Compiler Solution to Manage Unified Register File for CGRAs. Shail Dave, Mahesh Balasubramanian, Aviral Shrivastava, in Proceedings of the 21st International Conference on Design Automation and Test in Europe (DATE), 2018.

Advisory Report

1. Chapter: Sustainable Computing Architectures. In a report for US National Science Foundation (NSF) on Sustainability in Computing based on NSF Sponsored Workshop Series on Sustainability in Computing. 2023.

Patents

1. Systems and methods for agile and explainable optimization of efficient hardware/software codesigns for domain-specific computing systems using bottleneck analysis. Shail Dave, Aviral Shrivastava, Tony Nowatzki. US Non-Provisional Patent Application 2023.

2. Hybrid and efficient approach to accelerate complicated loops on coarse-grained reconfigurable arrays (CGRA) accelerators. Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, (ASU), Reiley Jeyapaul (ARM), US Non-Provisional Patent Application 2020.