UNIVERSITY OF CALIFORNIA SAN DIEGO

AI for Optimized Execution of AI

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Byung Hoon Ahn

Committee in charge:

Professor Hadi Esmaeilzadeh, Chair
Professor Steven Swanson, Co-Chair
Professor Sorin Lerner
Professor Ramesh Rao
Professor Dean Tullsen

2022

The Dissertation of Byung Hoon Ahn is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2022

DEDICATION

To my wife, Hwajin.

EPIGRAPH

Times and conditions change so rapidly that
we must keep our aim constantly focused on the future.

Walt Disney

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Above all, I thank and praise God, the Lord of all creation.

”Lord, you are my God; I will exalt you and praise your name, for in perfect faithfulness you have done wonderful things, things planned long ago.” – Isaiah 25:1

This dissertation would not have been possible without the help of many people. Foremost, I owe immense thanks to my advisor Prof. Hadi Esmaeilzadeh for guiding me through the journey that initially felt as if I were walking through a deep dark void. I thank him for his patience while I was spending time exploring radical research ideas, and for the unparalleled opportunities he provided. The past four years working with him have transformed me into a better researcher and a better person, and I could never thank him enough for all he has done for me.

I would also like to thank my committee members, Prof. Steve Swanson, Prof. Sorin Lerner, Prof. Ramesh Rao, and Prof. Dean Tullsen, for their insightful comments and whole-hearted encouragements. Valuable feedbacks from them helped me improve my dissertation.

My Ph.D. journey has been an exciting ride with a unique blend of both academic and industry research. I thank all my mentors in the industry for offering me to work on exciting projects and even deep dive into the startup scene. I had the privilege of working with Dr. Jinwon Lee, Dr. Harris Teague, Dr. Chris Lott, and Dr. Jilei Hou at Qualcomm AI Research; Dr. Eiman Ebrahimi at Protopia AI; Dr. Hojin Kee, Dr. Jinmook Lee, and Cecile Foret at Apple; Dr. Abdul Wasay and Dr. Tim Mattson at Intel Labs. In particular, I would like to thank Jinwon who extended his mentorship to help me in every step of my Ph.D. studies and my job search.

My life as a Ph.D. student would not have been the same without my amazing colleagues in Alternative Computing Technologies (ACT) lab, Dr. Ahmed Taha Elthakeb, Brahmendra Yatham, Chris Priebe, Edwin Mascarenhas, Fatemeh Mireshghallah, Hanyang Xu, Joon Kyung Kim, Lavanya Karthikeyan, Parsa Assadi, Prannoy Pilligundla, Rohan Mahapatra, Sean Kinzer, Shu-Ting Wang, and Soroush Ghodrati (I would also include Dr. Kazem Taram and Mojan Javaheripi as honorary members), as well as the former graduates, Dr. Amir Yazdanbakhsh, Dr. Divya Mahajan, Dr. Hardik Sharma, and Prof. Jongse Park. I will miss all the fun, the countless

hours of stimulating discussions that led to awesome research ideas, and the sleepless nights we spent together before the deadlines. I look forward to continuing our collaboration even after graduation. A special thanks to Jongse for hosting me at KAIST during the pandemic.

I thank many of my Korean friends in San Diego for making me feel home. I will miss the barbeque parties with Jonghun Park, Joon Kyung Kim, Kyung Soo Kim, Mingyu Woo, Yujin Park, and Yun Joon Soh. I will also miss taking strolls around the Geisel library with Juno Kim. In particular, I thank Jaeyoung Kang without whom I might have starved during the pandemic. I am also grateful to everyone at San Diego Calvary Korean Church for their love. While I cannot list all of them, I thank all my friends accross the world for always cheering and praying for me.

I would also like to give thanks to many staffs at UCSD. Julie Conner in the Department of Computer Science and Engineering has been very supportive to me despite the countless questions I bugged her with. Emily Stewart in the International Student and Programs Office helped me with many difficult immigration issues that I faced over the years.

I feel deeply indebted to the people who encouraged me to pursue Ph.D. I thank Prof. Seon Wook Kim for introducing me to the field of compiler optimization during my undergraduate studies at Korea University. I also thank Prof. Sung-Jea Ko and Prof. Myo Taeg Lim at Korea University for supporting me with reference letters to come to the United States for graduate studies. I also thank Dr. Yeonbok Lee, Dr. Jonghun Lee, Dr. Shin-gyu Kim, Prof. Myungsun Kim, Dr. Sukjin Kim, and Dongjin Koh at Samsung for helping me take my first step as a researcher and encouraging me to pursue Ph.D. studies. I also thank the National Institute for International Education (NIIED) for the financial support during the first two years of my studies.

I would like to extend my deepest gratitude to my family: my father, Chan Young Ahn; my mother, Mi Sung Kim; my brother, Byung In Ahn; sister-in-law, Jinsil Kim; my nephew Woo Rim Ahn; my in-laws, Jong Hee Lee, Youjung Cho, and Jaesang Lee. I feel very fortunate to have such a wonderful family. Their love, patience, and support made it possible for me to be here and I owe a lot to them.

Last but not least, I thank the love of my life, Hwajin Lee, without whom this dissertation

would not have been possible and without whom none of this would matter. I thank my wife for her patience, encouragement, and dedication. Her endless love and unwavering support even during the most difficult times fueled me with the energy to soldier through the difficulties I faced during my Ph.D. studies. I look forward to spending more time with her and creating many wonderful memories throughout the rest of my life with her.

Chapter 2, in part, contains a re-organized reprint of the material as it appears in International Conference on Learning Representations (ICLR) 2020. Ahn, Byung Hoon; Pilligundla, Prannoy; Yazdanbakhsh, Amir; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, contains a re-organized reprint of the material as it appears in Conference on Machine Learning and Systems (MLSys) 2020. Ahn, Byung Hoon; Lee, Jinwon; Lin, Jamie Menjay; Cheng, Hsin-Pai; Hou, Jilei; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper[1].

Chapter 4, in part, contains a re-organized reprint of the material as it appears in Design Automation Conference (DAC) 2022. Ahn, Byung Hoon; Kinzer, Sean; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, contains a re-organized reprint of the material as it appears in IEEE Micro 2022. Kim, Joon Kyung; Ahn, Byung Hoon; Kinzer, Sean; Ghodrati, Soroush; Mahapatra, Rohan; Yatham, Brahmendra; Wang, Shu-Ting; Kim, Dohee; Sarikhani, Parisa; Mahmoudi, Babak; Mahajan, Divya; Park, Jongse; Esmaeilzadeh, Hadi. The dissertation author was the co-investigator and co-author of this paper.

[1]Qualcomm Technologies, Inc. ("QTI") grants Byung Hoon Ahn ("Licensee") a limited, revocable, non-transferable, non-exclusive, royalty and fee free copyright license to copy and reproduce, in whole or in part, the paper entitled "Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices" published in MLSys 2020 along with any supplemental material provided with the paper ("Content") as part of Licensee's Ph.D. thesis submission, provided that the Content is cited as belonging to Qualcomm Technologies, Inc., has the appropriate copyright notice as a footnote and is only disclosed to Licensee's PH.D. school Director and to the academic reviewers of the thesis. Any further disclosure of the Content will require additional permission or license. The Content remains the exclusive property of QTI and no other rights are granted.

VITA

2015	Bachelor of Engineering, Korea University
2020	Master of Science, University of California San Diego
2022	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh, "Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices," MLSys, 2020

Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh, "Chameleon: Adaptive Code Optimization for Expedited Deep Neural Network Compilation," ICLR, 2020

Soroush Ghodrati, **Byung Hoon Ahn**, Joon Kyung Kim, Sean Kinzer, Brahmendra Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh, "Planaria: Dynamic Architecture Fission for Spatial Multi-Tenant Acceleration of Deep Neural Networks," MICRO, 2020

Byung Hoon Ahn, DoangJoo Synn, Masih Derkani, Eiman Ebrahimi, and Hadi Esmaeilzadeh, "Protopia AI: Taking on the Missing Link in AI Privacy and Data Protection," NeurIPS Demonstrations, 2021

Byung Hoon Ahn, Sean Kinzer, and Hadi Esmaeilzadeh, "Glimpse: Mathematical Embedding of Hardware Specification for Neural Compilation," DAC, 2022

Joon Kyung Kim, **Byung Hoon Ahn**, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh, "Yin-Yang: Programming Abstraction for Cross-Domain Multi-Acceleration," IEEE Micro Special Issue on Compiling for Accelerators, 2022

ABSTRACT OF THE DISSERTATION

AI for Optimized Execution of AI

by

Byung Hoon Ahn

Doctor of Philosophy in Computer Science

University of California San Diego, 2022

Professor Hadi Esmaeilzadeh, Chair
Professor Steven Swanson, Co-Chair

In the recent decade, *Intelligent Systems*–advanced computer systems that can make useful predictions or decisions based on observations–have become increasingly ubiquitous: from personal assistants to self-driving cars. These intelligent systems are incarnations of *Artificial Intelligence (AI)*, powered by the recent advances in *Deep Neural Networks (DNNs)* that now exhibit superhuman performance in many tasks such as image classification, game playing, and protein-folding problems. Such astounding performance of DNNs depend on two key ingredients: *Data* and *Computing Power*. In the current era of big data, the rate of data generation has reached an overwhelming level that is beyond the capabilities of conventional

computing systems. The hardware design has gone through a significant change and has exploded in diversity to cope with the rate of data generation and the computational intensity of DNNs. Nevertheless, developing *Compilers* to optimize the code for them remains an open challenge.

DNNs have made significant strides in context-sensitive natural language translation. These advances can be seen as an opportunity to utilize DNNs for the compilation of DNNs, themselves, which in fact is a series of translation tasks. To this end, the dissertation begins by introducing an effort to integrate deep reinforcement learning to improve the compilers' capability to adapt to unseen search spaces in code optimization. This marks an initial step in leveraging *AI for Optimized Execution of AI* on commodity platforms. Although the exciting results from the work shows the potential for leveraging machine intelligence for compilation, it does not fully justify relinquishing the swathe of conventional optimization techniques and the foundational algorithms that have been curated with human ingenuity over decades. As such, the dissertation also explores the other end of the spectrum–*Foundational Algorithms*–to tackle the problem of memory footprint in neural execution. This work achieves memory-optimal scheduling building on dynamic programming, a well-known foundational algorithm in computer science. Having observed that both worlds–AI and Foundational Algorithms–can bring significant benefits to compiler optimization, this dissertation culminates to an ambitious effort to take advantage of the best of both worlds. The dissertation presents *Hybridization of AI and Foundational Algorithms* for optimized execution of AI, where we utilize mathematical embeddings to extract core information from the hardware specification while using meta learning to fuse those information into compilers for improved compilation performance.

Intelligent systems comprise components from various domains that are not limited to DNNs. Therefore, it naturally makes *Cross-Domain Multi-Acceleration* our next step. To this end, this dissertation devises a set of abstractions for various application domains and their hardware, then a virtual machine for execution of end-to-end applications. The work sets the foundations for cross-domain multi-acceleration to expand the scope of the aforementioned AI-enabled compilation techniques to the end-to-end intelligent systems.

Chapter 1

Introduction

In the recent decade, *Intelligent Systems*–advanced computer systems that can make useful predictions or decisions based on their observations–have become increasingly ubiquitous. For example, machine translation [68], smart factory [47], home robots [9, 149], personal assistants [8, 18, 66, 148], surveillance [12], and self-driving cars [164, 169] are no longer the figment of human imagination but are real services and products that have deeply penetrated into our daily lives (Figure 1.1). These intelligent systems are incarnations of *Artificial Intelligence (AI)*, and are powered by the recent advances in *Deep Neural Networks (DNNs)*. To demonstrate, Figure 1.2 illustrates the software architecture of the Tesla's Full Self-Driving (FSD) pipeline [169]. *Vision* includes various network DNN models and components such as RegNet [139] and Bi-directional Feature Pyramid Network [168]. In fact, their more recent updates and future plans include Spatial Attention [177] and Neural Radiance Fields (NeRF) [115]. Also, the *Neural Net Planner* includes a Monte-Carlo Tree Search (MCTS) based algorithm similar to that of AlphaGo [160]. Similarly, many other intelligent systems also rely on DNNs that exhibit superhuman performance in many tasks such as image classification [75], game playing [160], and protein-folding problems [90].

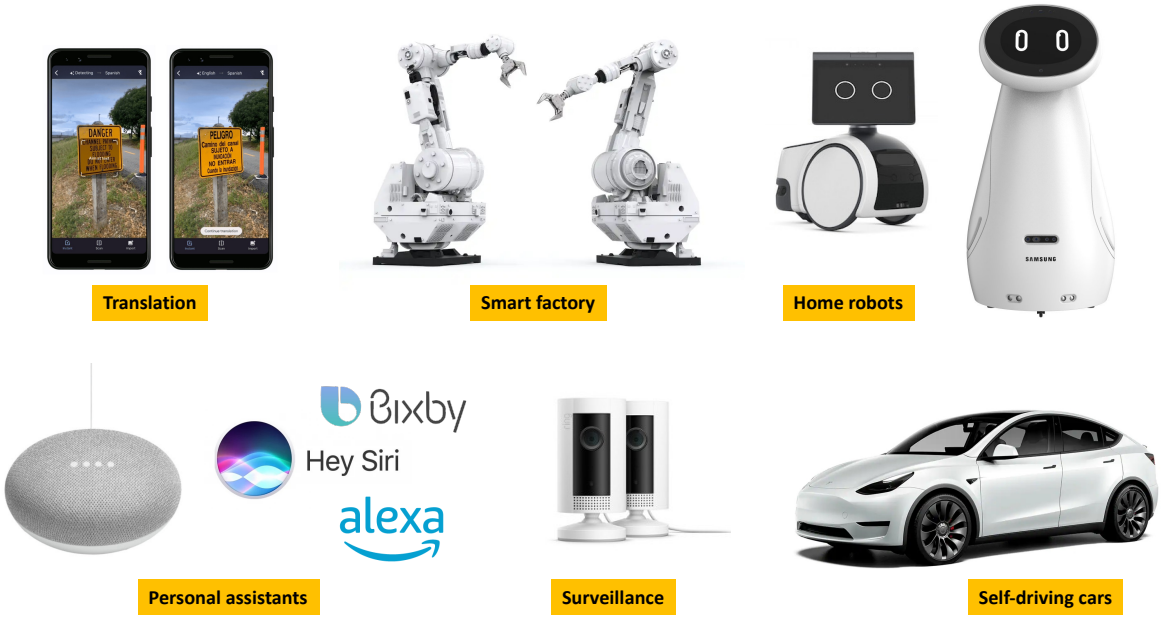


Figure 1.1. Examples of *Intelligent Systems*: translation, smart factory, home robots, personal assistants, surveillance, and self-driving cars.

1.1 Background

Inception of large-scale datasets and hardware advances.

Such astounding performance of DNNs depend on two key ingredients: *Data* and *Computing Power*. In current era of big data, the rate of data generation has reached an overwhelming level that is beyond the capabilities of the conventional computing systems [108]. In fact, large-scale datasets such as ImageNet [48], WMT [25], and LibriSpeech [133] are now readily available to train large DNN models and are growing to extreme scales. On the other hand, hardware has also experienced significant changes in the recent decade. Insatiable demand for computation in DNN applications and the Dark Silicon era [51] have coincided calling for innovations with less support from the technological advancements (Moore's Law [151] and Dennard Scaling [49]) [80]. In fact, both the industry [11, 14, 56, 88] and the academic community [36, 37, 60, 64, 72, 89, 134, 157] have opted for acceleration. Both communities have made large strides and innovations in the DNN hardware design in the *Golden Age of Domain-Specific Architectures* [78]. In fact, the Cambrian explosion of domain-specific accelerators now

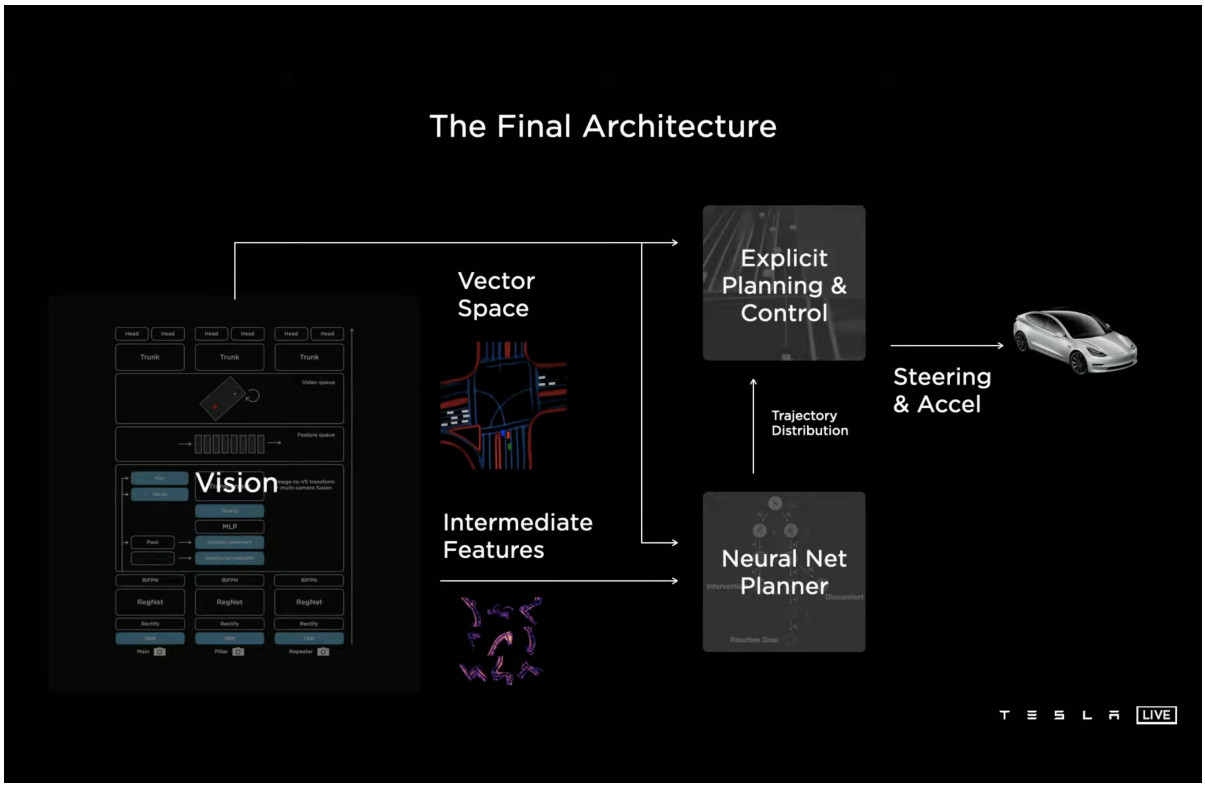


Figure 1.2. The final architecture of Tesla's FSD pipeline [Source: Tesla AI Day 2021 [169]].

enable training of large-scale deep learning models in order of minutes, over what used to be days a few years ago [114].

Cambrian explosion of deep neural networks.

Another important trend in the past decade is the unprecedented growth in the research community for deep neural networks. As shown in the Annual Report from Stanford University [189], "from 2010 to 2021, the total number of AI publications doubled, growing from 162,444 in 2010 to 334,497 in 2021." (Figure 1.3) This translates to around 916 AI publications per day in 2021. Many of these papers either propose new model architectures or build on the foundational models to solve real-world problems. This fast growth in diversity of deep learning models is a testament to the wide-adoption of DNNs. On the other hand, such diversity of DNNs raises a question to the computer architects and compiler engineers on how to optimize the execution of DNNs.

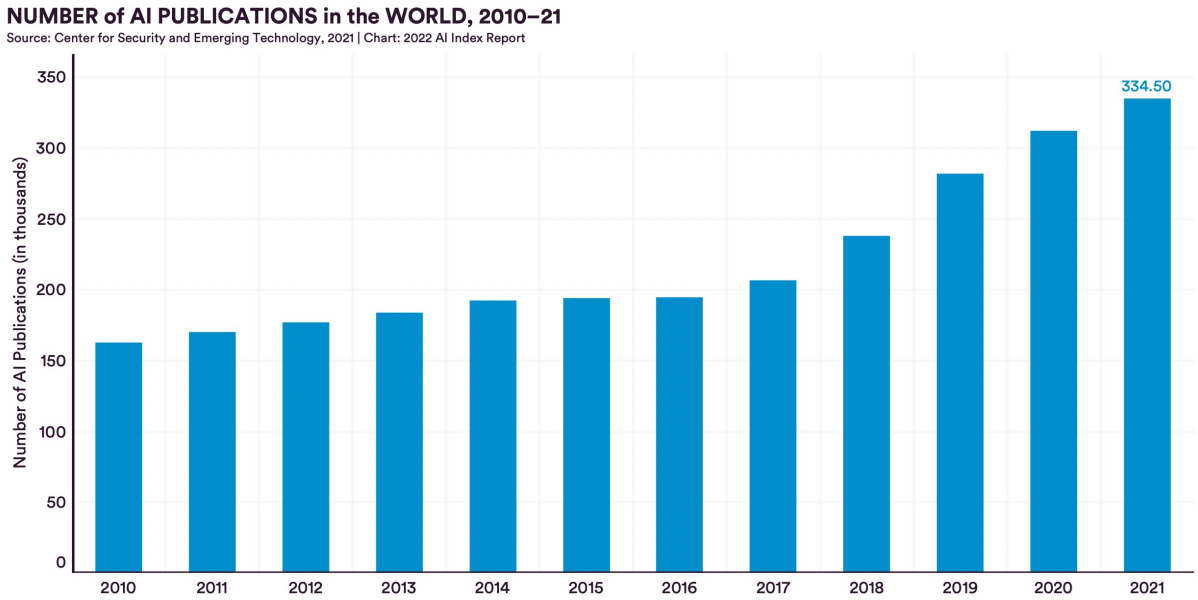


Figure 1.3. Number of AI publications in the world, 2010–21 [Source: AI Index Annual Report 2022 by Stanford University [189]].

Deep learning compilers.

Given the increasing diversity of deep learning models and the Cambrian explosion of DNN hardware, it becomes imperative that we generate efficient code for neural execution. The community initially relied on hand-optimized kernels such as NVIDIA cuDNN or Intel MKL that serve as backend for popular deep learning libraries such as TensorFlow [1] and PyTorch [135]. However, the complexity of the tensor operations in DNNs and the volatility of algorithms call for developing *Automated Compilation Frameworks* that bridge the gap between the deep learning models (*SW*) and the deep learning accelerators (*HW*). To this end, industry and academia have developed deep learning compilers such as TVM [34], TensorComprehensions [170], Glow [143], TensorFlow XLA [146], and Intel nGraph [45]. These compilers benefit from various optimization to achieve or surpass the code performance of hand-optimized libraries [104]. Nevertheless, there are still many open challenges in developing deep learning compilers.

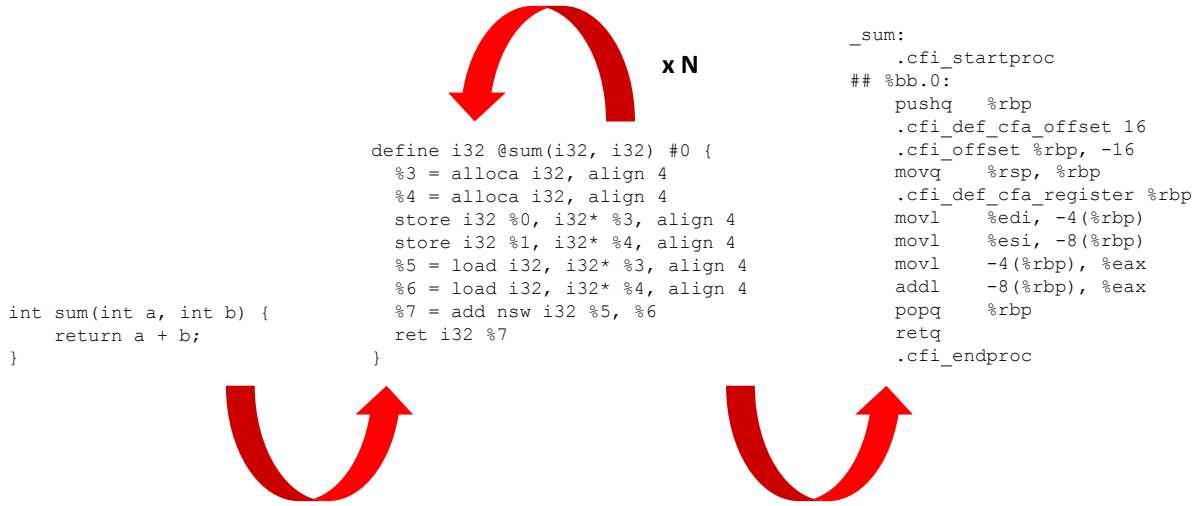


Figure 1.4. Example compiling a function `int sum(int a, int b)` with LLVM [99].

1.2 AI-Enabled Compilation for Intelligent Systems

Opportunities in integrating AI into compilers.

Figure 1.4 demonstrates an example compiling a function `int sum(int a, int b)` with LLVM [99]. Input to the compiler is a code in C which is passed to the front-end of the compiler (e.g., `clang`) to be transformed into LLVM Intermediate Representation (IR). Then, the middle-end of the compiler applies various LLVM target-independent optimizations such as loop invariant code motion and common subexpression elimination. Finally, the back-end of the compiler applies various target-dependent optimization as well as legalization, instruction selection, scheduling, and register allocation. The overall compilation process and the code optimization can be viewed as a series of transformations from the source code to the final output binary. Recently, however, the research community has made significant strides in context-sensitive natural language translation [171]. These advances can be seen as an opportunity to utilize DNNs for the compilation of DNNs, themselves, which in fact is a series of translation tasks.

AI-enabled compilation of deep neural networks.

Code optimization in deep learning compilers is a time-consuming process that navigates through an immense search space. For example, black-box optimization for deep learning kernels has around 10^{10} possibilities, and it may take over 10 hours to compile even a relatively small model such as ResNet-18 [75]. In fact, scheduling of layers permits all topological orderings of layers in the computational graph of DNNs, making the task of finding memory-optimal schedule to have exponential complexity. While the application of AI to solve these problems look appealing on surface, it not only requires using the right algorithms but also determining whether AI is really necessary. To this end, this dissertation explores the both ends of the spectrum (AI algorithms vs Foundational algorithms) then propose a hybridization of the two to develop the *AI-Enabled Compilation of Deep Neural Networks*.

AI-enabled compilation of end-to-end intelligent systems.

Accelerating DNNs have a significant impact on the overall latency and efficiency of running intelligent systems. However, the algorithms that constitute intelligent systems span domains beyond just DNNs. For example, even the above mentioned FSD pipeline in Figure 1.2 also includes many non-DNN components. To accelerate and optimize the end-to-end intelligent systems, the solutions that have been developed for DNN code optimization would not suffice. Nevertheless, each domain and their hardware comes with its own vertically-specialized domain-specific stack which by design is difficult to conjugate with other stacks. This makes it difficult to reuse the insights from the AI-enabled compilation of deep neural networks to other domains or to expand the scope of the optimization to other domains. To this end, this dissertation builds the foundations for *AI-Enabled Compilation of Intelligent Systems* by developing a horizontal system stack that can break the vertical barriers of the specialized domain-specific stacks to enable *Cross-Domain Multi-Acceleration*.

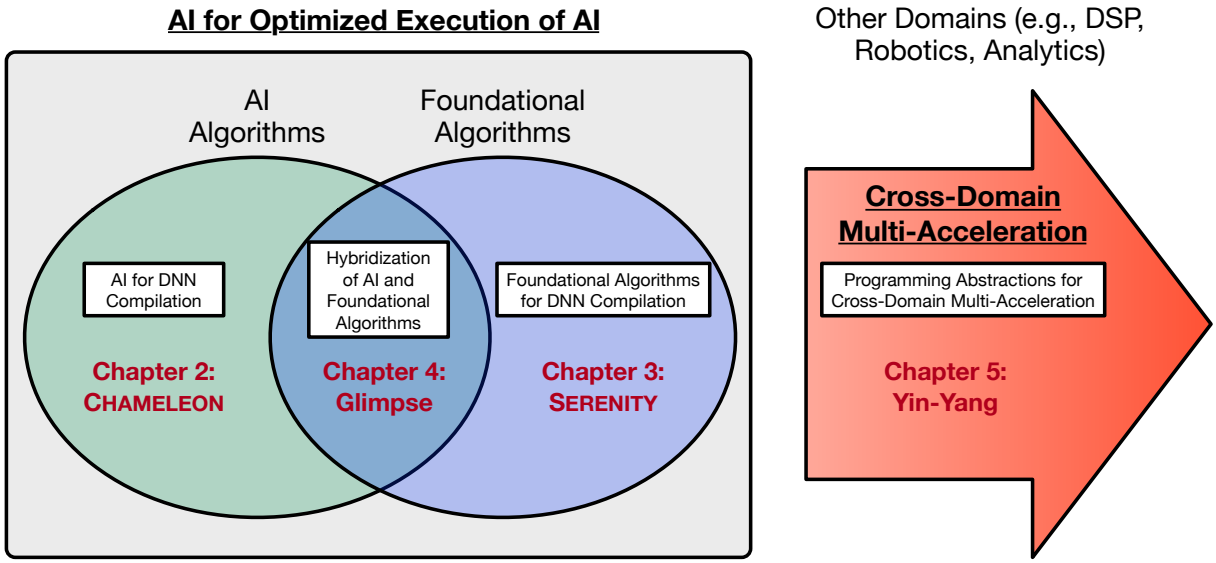


Figure 1.5. Overview of the dissertation.

1.3 Thesis Contributions

This section provides an overview of this dissertation. The dissertation first dive into AI-enabled compilation of deep neural networks, then the last part of the dissertation proposes novel programming abstractions to expand the scope of AI-enabled compilation to the end-to-end intelligent systems (Figure 1.5).

Chapter 2: AI for optimized execution of AI.
Adaptive code optimization for expedited deep neural network compilation.

We provide an adaptive code optimization framework that takes advantage of AI algorithms. We present **CHAMELEON** which can significantly reduce the compilation time and offer automation while avoiding dependence to hand-optimization, enabling far more diverse tensor operations in the next generation deep learning models. The framework comprises three key components: *Adaptive Exploration* module that utilizes reinforcement learning to adapt to unseen design space of new networks to reduce search time yet achieve better performance, *Adaptive Sampling* algorithm that utilizes clustering to adaptively reduce the number of costly hardware measurements, and *Sample Synthesis* module that takes a domain-knowledge inspired

approach to find configurations that would potentially yield better performance. Importantly, the work marks an initial effort to bring an AI algorithm–reinforcement learning–to the realm of optimizing compilers for neural networks. **Chapter 2** describes this effort in more detail.

Chapter 3: Foundational algorithms for optimized execution of AI.
Memory-aware scheduling of irregularly wired neural networks for edge devices.

We provide a memory-optimization framework that utilizes a foundational algorithms in computer science. We present **SERENITY** which can find memory-optimal schedules for neural execution. The framework automatically schedules the nodes of the deep neural network computational graph to minimize the memory footprint to meet the limitations of the edge devices. The framework includes three key components: *Dynamic Programming-based Scheduler* that takes advantage of the signatures from the repeated subpaths while searching for optimal schedules, *Adaptive Soft Budgeting* technique that performs a light-weight meta-search to find the appropriate memory budget for pruning suboptimal paths for a signficantly faster scheduling, and *Identity Graph Rewriting* that, similar to strength reduction in modern compilers, exchanges subgraphs with mathematically equivalent counterparts that can lead to a lower memory footprint. This work shows that foundational algorithms or compiler heuristics based on human ingenuity can even yield optimal solutions in deep learning compiler, where optimal solutions are commonly unattainable. **Chapter 3** describes the proposed solution in more detail.

Chapter 4: Hybridization of AI and foundational algorithms for optimized execution of AI. Mathematical embedding of hardware specification for neural compilation.

We provide a Bayesian optimization framework for code optimization that takes a hybrid approach between AI and foundational algorithms. The proposed framework **Glimpse** transforms a previously blind black-box optimization process into a gray-box optimization which takes hints from a mathematical embedding of hardware specification for faster neural compilation. The work introduces two key ideas: **Blueprint** that encodes key architectural information of the hardware specified in the public data sheet, and a hardware-aware optimization framework called **Glimpse** that takes into account the information present in the **Blueprint**. Importantly,

Blueprint builds on a widely used mathematical embedding algorithm: Principal Component Analysis (PCA), and the components in **Glimpse** (*Prior Distribution Generator*, *Hardware-Aware Exploration*, and *Hardware-Aware Sampling*) build on modern AI algorithms such as meta-learning. This work serves a case study of how AI algorithms and foundational algorithms can be assembled together seamlessly to improve deep learning compilation. **Chapter 4** describes the work in more detail.

Chapter 5: Expanding the scope to end-to-end intelligent systems.
Programming abstractions for cross-domain multi-acceleration.

As an effort to expand the scope of the compilation to the end-to-end intelligent systems, we explore how applications that comprise multiple domains can be accelerated. This effort includes two parts: devising **Yin-Yang** programming abstractions that breaks the vertical barriers of the specialized domain-specific stacks, and developing **XLVM**: a dataflow virtual machine that maps domain functions to best-fit accelerator capabilities. Overall, this work introduces the foundational framework on top of which we can expand the scope of the AI-enabled compilation (built on the learnings from **Chapter 2–4**) to the end-to-end intelligent systems. **Chapter 5** describes the work in more detail and discusses the future directions.

Chapter 2

AI for Optimized Execution of AI

2.1 Adaptive Code Optimization for Expedited Deep Neural Network Compilation

Achieving faster execution with shorter compilation time can foster further diversity and innovation in neural networks. However, the current paradigm of executing neural networks either relies on hand-optimized libraries, traditional compilation heuristics, or very recently genetic algorithms and other stochastic methods. These methods suffer from frequent costly hardware measurements rendering them not only too time consuming but also suboptimal. As such, we devise a solution that can learn to quickly adapt to a previously unseen design space for code optimization, both accelerating the search and improving the output performance. This solution dubbed **CHAMELEON**[1] leverages reinforcement learning whose solution takes fewer steps to converge, and develops an adaptive sampling algorithm that not only focuses on the costly samples (real hardware measurements) on representative points but also uses a domain-knowledge inspired logic to improve the samples itself. Experimentation with real hardware shows that **CHAMELEON** provides 4.45×speed up in optimization time over AutoTVM, while also improving inference time of the modern deep networks by 5.6%.

[1]Chameleon is an animal that is capable of *Adapting* to their environments which helps them survive. In our work, **CHAMELEON** is an entity that *Adapts* to the variations in the design space and the distribution of the candidate configurations, enabling expedited deep neural network compilation.

2.2 Introduction

The enormous computational intensity of Deep Neural Networks (DNNs) have resulted in developing either hand-optimized kernels, such as NVIDIA cuDNN or Intel MKL that serve as backend for a variety of programming environment such as TensorFlow [1] and PyTorch [135]. However, the complexity of the tensor operations in DNNs and the volatility of algorithms, which has led to unprecedented rate of innovation [100], calls for developing automated compilation frameworks. To imitate or even surpass the success of hand-optimized libraries, recent research has developed stochastic optimization passes: for general code, STOKE [152], and neural network code, TVM [34] and TensorComprehensions [170]. TVM and TensorComprehensions are based on random or genetic algorithms to search the space of optimized code for neural networks. AutoTVM [35] builds on top of TVM and leverage boosted trees [33] as part of the search cost model to avoid measuring the fitness of each solution (optimized candidate neural network code), and instead predict its fitness. However, even with these innovations the optimizing compilation time can be around 10 hours for ResNet-18 [75], and even more for deeper or wider networks.

Since the general objective is to unleash new possibilities by developing automatic optimization passes, long compilation time hinders innovation and could put the current solutions in a position of questionable utility. To solve this problem, we first question the very statistical guarantees which the aforementioned optimization passes rely on. The current approaches are oblivious to the patterns in the design space of schedules that are available for exploitation, and causes inefficient search or even converges to solutions that may even be suboptimal. Also, we notice that current approaches rely on greedy sampling that neglects the distribution of the candidate solutions (configurations). While greedy sampling that passively filter samples based on the fitness estimations from the cost models work, many of their hardware measurements (required for optimization) tend to be redundant and wasteful. Moreover, we found that current solutions that rely on greedy sampling lead to significant fractions of the candidate configurations being

redundant over iterations, and that any optimizing compiler are prone to invalid configurations which significantly prolongs the optimization time. As such, this work sets out to present an *Adaptive* approach dubbed **CHAMELEON** to significantly reduce the compilation time and offer automation while avoiding dependence to hand-optimization, enabling far more diverse tensor operations in the next generation DNNs. We tackle this challenge from two fronts with the following contributions:

1. Devising an *Adaptive Exploration* module that utilizes reinforcement learning to adapt to unseen design space of new networks to reduce search time yet achieve better performance.

2. Proposing an *Adaptive Sampling* algorithm that utilizes clustering to adaptively reduce the number of costly hardware measurements, and devising a domain-knowledge inspired *Sample Synthesis* to find configurations that would potentially yield better performance.

Real hardware experimentation with modern DNNs (AlexNet, VGG-16, and ResNet-18) on a high-end GPU (Titan Xp), shows that the combination of these two innovations, dubbed **CHAMELEON**, yields 4.45×speedup over the leading framework, AutoTVM.

2.3 Challenges in Deep Neural Network Compilation

The general life-cycle of deep learning models from its birth to deployment comprises of two major stages. First stage is the designing and the training of a deep learning model by a research scientist, with the primary goal of achieving the highest feasible accuracy. Then, with a general demand to enable the intelligence on a wide range of devices (from mobile CPUs in the edge to cloud-scale GPUs), the second stage has emerged for the deployment of the pre-trained deep learning model to a target hardware by a deployment engineer. These stages are each iterative processes: research scientists iterate until it reaches the target performance in terms of accuracy whereas the deployment engineers iterate until the performance in terms of inference speed with a given hardware satisfies the given constraints. Importantly, these two stages are

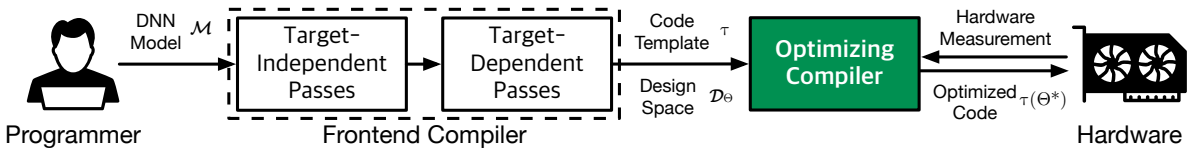


Figure 2.1. Overview of our model compilation workflow. Highlighted in green is the scope of this work and where **CHAMELEON** comes into play.

most often separate processes, and this paper mainly focuses on the second stage (deployment) of the cycle with an overarching goal of accelerating the overall deployment cycle by reducing the optimizing compilation time without compromising the performance of the output code.

2.3.1 Compilation Workflow for Deep Neural Networks

Figure 2.1 illustrates how a compiler for DNNs takes an input model $\mathcal{M}$ and emits an optimized code $\tau(\Theta^*)$ that runs the model efficiently on a given hardware. This flow is commensurate with TensorComprehensions [170] and TVM [34], using which we implement **CHAMELEON** that is available as a separate package for adoption in even other frameworks. The first phase of the workflow is the frontend compiler which performs the translation from the compiler and applies target-independent and white-box target-dependent optimizations that do not incorporate a measure of runtime. Target-independent passes transform the input DNN model without specificity to the target hardware. Operator fusion and data layout transformation in TVM are some examples of these passes, which lie in the same category as dead-code elimination or loop-invariant code motion in GCC [162] or LLVM [99]. Target-dependent passes, on the other hand, the compiler takes the hardware architecture (target) into account while optimizing the program; however, this also does not actively leverage runtime measures. The last stage is a black-box optimization pass, called *optimizing compiler*, that given a measure of performance at runtime from the hardware can further optimize the code. **CHAMELEON** falls in this class by offering an optimizing compiler that *adapts* to different design space to be more swift in optimizing deep neural networks compared to conventional approaches.

2.3.2 Optimizing Compiler for Deep Neural Networks

Optimizing compilers [92] usually take a black-box approach and use hardware measurements to configure the optimization based on a measure of fitness f of each solution. In order to make the problem tractable, the optimizing compilers for deep neural networks reduce the problem down to tuning the knobs θ for the output code template τ, and can be formulated as:

$$\Theta^* = \underset{\Theta}{\operatorname{argmax}} f(\tau(\Theta)), \qquad \text{for } \Theta \in \mathcal{D}_\Theta. \tag{2.1}$$

A combination of assignment to the knobs is said to be a configuration $\Theta = (\theta_1, \theta_2, ..., \theta_n)$ while the dimensions of the design space $\mathcal{D}_\Theta$ is defined by the knobs. As such, in Equation 2.1, an optimizing compiler starts from a code template τ for each layer, and makes use of a search algorithm and real hardware measurements to efficiently find the best configuration $\Theta^* \in \mathcal{D}_\Theta$. In this context, there are three variables that determine the effectiveness of the optimizing compiler: (1) *a large and diverse enough design space that covers a variety of transformations*, (2) *an effective search algorithm to adequately navigate this space*, and (3) *a mechanism to cut down the number of costly hardware measurements that check the fitness of a solution*. Table 2.1 lists the knobs for performing convolution on a GPU, where it is crucial that the code (1) maximizes data reuse, (2) uses the shared memory wisely, and (3) minimizes bank conflicts. The knobs optimize various aspects of the execution, including tiling (e.g., tile_x, tile_y, ...), unrolling (e.g., auto_unroll_max_step and unroll_explicit), and these knobs define a design space with 10^{10} possibilities. Given the vastness of the design space, the remaining challenges are designing *an effective search algorithm* and designing *a mechanism that reduces the cost of each step in the search* (i.e. reducing the need to measure the hardware).

2.3.3 Challenges in Deep Neural Network Compilation

As shown in Figure 2.2, optimizing compilation for DNNs may still take an eon even with the advances from prior works [34, 35, 170] With active research [7, 40, 70, 114, 185,

Table 2.1. Knobs in the design space to optimize convolution.

KNOBS	DEFINITION
tile_f, tile_y, tile_x	Factors for tiling and binding number of filters height, and width of feature maps.
tile_rc, tile_ry, tile_rx	Factors for tiling reduction axis such as number of channels, height, and width of filters.
auto_unroll_max_step	Threshold of number of steps in the loop to be automatically unrolled.
unroll_explicit	Explicitly unroll loop, this may let code generator to generate pragma unroll hint.

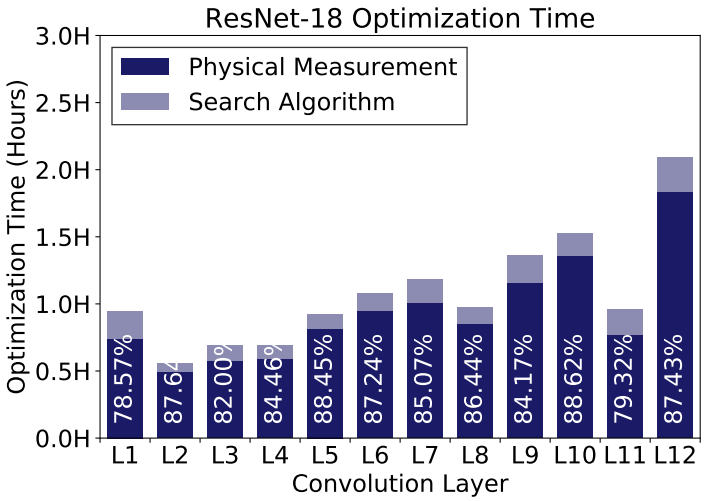


Figure 2.2. AutoTVM optimization time breakdown for ResNet-18 on Titan Xp.

186] that has been able to cut down the training time to only few hours [70, 185] and even minutes [7, 186] on big models (e.g., ResNet-50 [75]) for ImageNet, it renders the optimizing compilation time of the current solutions seem even more prominent. Especially, since the above-mentioned compilers have been integrated to the deep learning pipelines of major players in the industry [107, 143, 170], many users of these pipelines including the deployment engineers must go through the compilation workflow depicted in Figure 2.1 numerous times. Therefore, current long compilation time can be a hindrance to deploying DNN in various hardware, hence a major bottleneck in enabling intelligence on wider range of target platforms.

Furthermore, as we explore various neural topologies [178, 181] for better performance as illustrated in [4], even deeper or wider networks [167, 188], and new operations [81] to achieve higher performance [100], we are forced to optimize the networks more frequently. The long optimization times are multiplied with such trend, leaving the practical utility of the current compiler solutions to question. As such, the primary goal of this work is reducing the optimizing compilation time to meet the immediate needs of the industry for expedited DNN compilation to foster further diversity and innovation in designing DNNs.

Such long optimization time results from the inefficiency of simulated annealing which (while it stochastically guarantees a reasonable solution after huge number of iterations) *fails*

to capture the patterns in the design space that can be exploited during the search. On the other hand, we can see in the figure that *majority of the optimization time is spent on reaching for measurements on real hardware* that is used as a feedback for the aforementioned search. Also, current approach even *suffers from numerous invalid configurations* that not only wastes the limited hardware measurement budget that the compiler starts with, but also incurs serious overhead to reset the target hardware for subsequent hardware measurements. As such, it is important that a sampling mechanism that selects potential configurations for hardware measurements to be smarter to ensure that each measurement is maximizing the chances of achieving a good solution and that it evades the invalid configurations. However, the current approaches rely on greedy sampling that passively sample based on the estimations from the cost models. This not only has a tendency to overfit but also neglect that solutions are distributed non-uniformly and that there are numerous invalid configurations.

2.4 CHAMELEON: Adaptive Code Optimization for Expedited Deep Neural Network Compilation

As discussed in Section 2.3, current solutions fall short of providing a swift optimization framework for optimizing emergent deep neural networks, because of the futility of the search in adapting to the design space from a random walk based search algorithm and the inefficiency of the physical hardware measurements from the greedy sampling. Therefore, developing a new framework that can overcome current challenges to unfetter neural network innovation from a prolonged optimization times can be boiled down to two problems: ❶ improving the search algorithm to better adapt to the design space, and ❷ improving the sampling algorithm to both better adapt to the distribution of the solutions and decrease the possibility of running into invalid configurations. As such we make two innovations in the optimizing compiler for deep neural networks to develop **CHAMELEON** by applying reinforcement learning to the search that can adapt to new design spaces (*Adaptive Exploration*) and devising an *Adaptive Sampling* that replaces

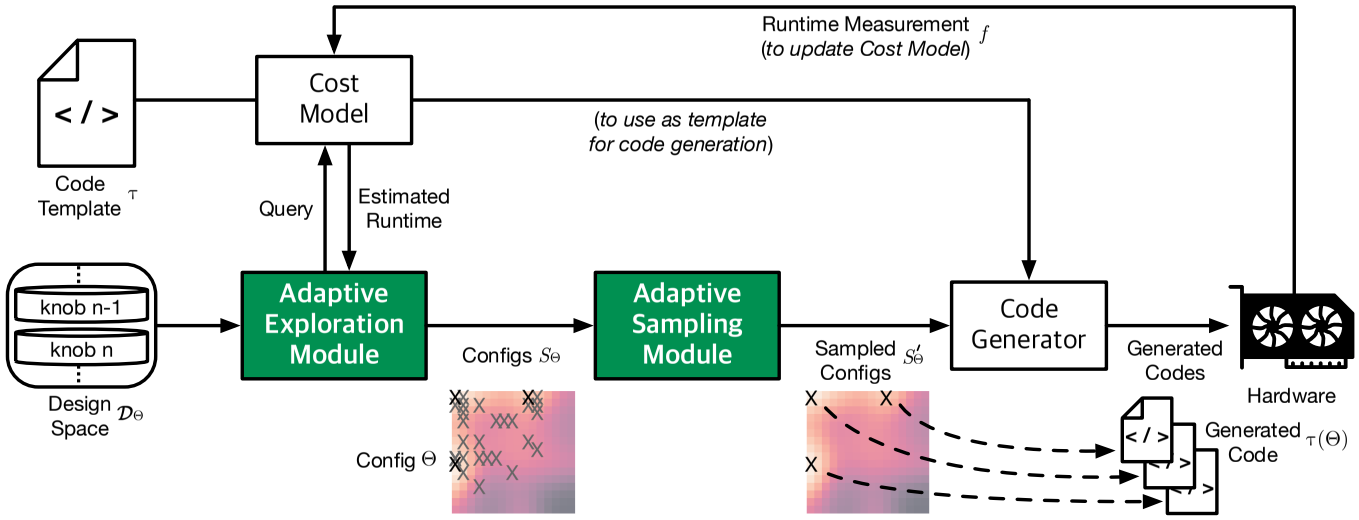


Figure 2.3. Overall design and compilation overview of the **CHAMELEON**.

the current greedy sampling.

2.4.1 Overall Design of Chameleon

Figure 2.3 outlines the overall design of our optimizing compiler, dubbed **CHAMELEON**, and gives an overview of the optimizing compilation process. **CHAMELEON** takes code template τ for each layer in the network and the corresponding design space $\mathcal{D}_\Theta$ as its input, and iteratively optimizes the code for configuration Θ to finally output $\tau(\Theta^*)$. The proposed *Adaptive Exploration* maneuvers the design space while using a cost model as a proxy for hardware measurements to the output set of candidate configurations S_Θ. These configurations are then sampled with *Adaptive Sampling* so that the sampled configurations S'_Θ subsume the initial candidate configurations while reducing its number significantly. The sampled configurations S'_Θ are then passed to the code generator which combines the input template τ and the configurations S'_Θ to create a set of $\tau(\Theta)$ that are sent to real hardware for runtime measurements. Runtimes from the hardware are used as the measure of fitness f and update the cost model to enhance the exploration of the subsequent iterations. After multiple iterations, $\tau(\Theta^*)$ with the best fitness f (shortest runtime) is selected as an output for the layer.

2.4.2 Adaptive Exploration: Learning about the Unseen Design Space to Expedite Convergence of Optimization

As stated in Section 2.3, the current state-of-the-art approach [35] that leverages simulated annealing relies on the stochastic guarantees of its random walks. Therefore, the current approach requires numerous iterations of exploration to converge to a reasonable solution causing long compilation hours, thus insufficient to enable disruptive innovations in neural networks. We take an inspiring approach that avoids naive dependence on the stochastic guarantee of simulated annealing and leverage a technique that can *learn to adapt* to unseen design space to not only accelerate convergence but also bring some performance gains. As such, we develop *Adaptive Exploration* by leveraging *Reinforcement Learning (RL)*, which is concerned with learning to

maximize reward given an environment by making good *exploration* and *exploitation* tradeoffs, in our case maximizing fitness f of the explored configurations S_Θ.

Reinforcement learning formulation.

Our RL-based Adaptive Exploration module uses an *actor-critic style RL*, where policy network learns to emit a set of directions (vector of increment/decrement/stay) for each knob in the design space that will increase f of the next configuration and the value network learns the design space $\mathcal{D}_\Theta$ to estimate the value of the action. The first layer of these networks that takes the current configuration Θ as input is shared to foster information sharing among the two networks, and its output is fed into the subsequent layers the networks. These networks not only learn the dependencies among the different knobs of the design space (which are interrelated) that helps our module navigate through the design space but also lean the potential gains of the modifications to the configurations.

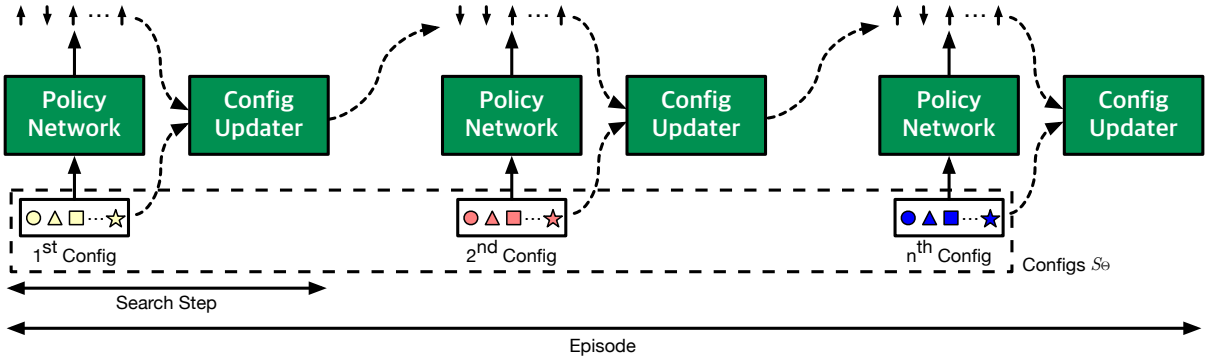
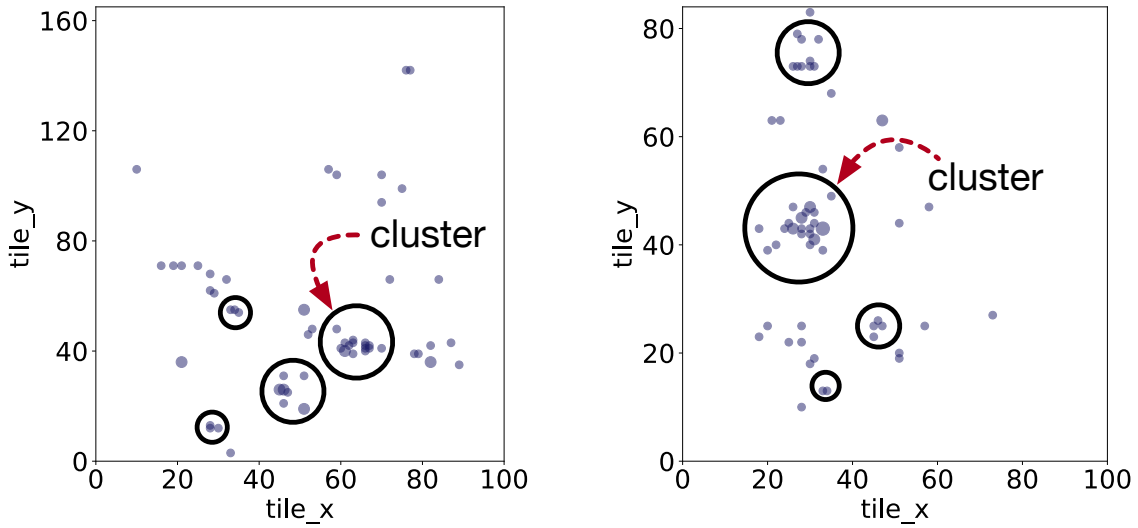


Figure 2.4. Adaptive Exploration Module of **CHAMELEON** in action.

Learning procedure.

Having formulated the RL-based Adaptive Exploration Module, an iteration of our optimization begins with a set of initial configurations and takes multiple search steps (episode) for each of the configurations. As shown in Figure 2.4, the agent makes an action and applies it to the configuration using configuration updater to get another configuration that potentially has better f. After finishing multiple search steps in the episode, all configurations S_Θ are evaluated using a cost model, which its return values are used as a surrogate reward to update our agent, to



(a) VGG-16 4th layer

(b) ResNet-18 11th layer

Figure 2.5. Clusters of candidate configurations.

reduce the number of costly hardware measurements. By taking this approach, f of S_Θ improves as our module progresses through the episodes. In other words, by repeating multiple episodes and iterations, our Adaptive Exploration Module gradually learns to locate good configurations.

2.4.3 Adaptive Sampling: Adapting to the Distribution to Reduce Costly Hardware Measurements

Reducing number of costly hardware measurements.

After the exploration step (regardless of the exploration method), we observe that the candidate configurations are clustered in subregions of the design space and these clusters are non-uniformly distributed (Figure 2.5). We also find that, while the design space's surface is discrete and un-smooth, a large fraction of configurations within each cluster achieve similar runtime (Figure 2.6). Utilizing these characteristics of the design space, we devise *Adaptive Sampling* that can sample a new set of candidates, by adapting to the shape of the design space and the non-uniformity of the distribution while leaving the performance of optimization intact. We first leverage *clustering* algorithm to find configurations that are representative of each cluster; the sampling module uses centroids as the representative configurations. Our Adaptive Sampling

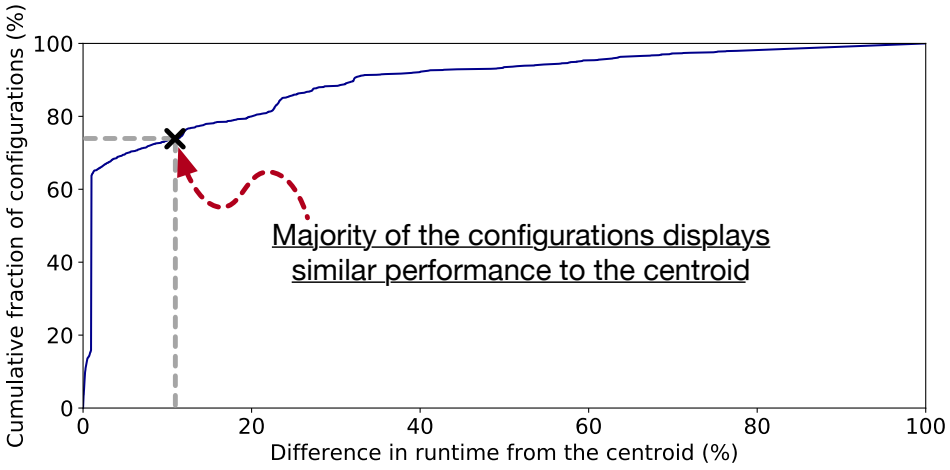


Figure 2.6. Cumulative Distribution Function (CDF) of the difference in runtime among the configurations in the cluster.

iterates over a different number of clusters for their respective centroids and the L2 loss.

In the context of optimizing compiler, selecting the number of centroids for clustering entails making the important tradeoff between selecting more centroids for better performance or fewer centroids for a reduced number of hardware measurements. As such, we must devise a method that would automatically make the tradeoff in a reasonable manner. We take advantage of the decreasing trend in the aforementioned L2 loss as we increase the number of centroids, and devise a *Threshold-based Swift Meta-Search* to determine the number of clusters. By setting the threshold (hyperparameter) it allows the compiler to determine the point of diminishing return (*knee* of the curve), inflection point beyond which fewer centroids may lead to performance degradation and more clusters would prolong the optimization substantially. Overall, our sampling curtails the number of hardware measurements so that it is just enough to subsume the entire subspace of the candidate configurations.

Improving candidate configurations using sample synthesis.

While the above sampling algorithm significantly reduces the number of hardware measurements compared to the conventional greedy sampling, without impacting the performance of the output code, we are still left with a critical issue of *redundancy among the candidate*

configurations. We find that the exploration algorithm (regardless of the type) combined with the greedy sampling frequently leads to redundancy among the candidate configurations over different iterations of optimization due to the overfitting of the cost model from the greediness of the sampling. Even though the exploration algorithm tries to explore unvisited regions of the design space, these explored (not exploited) configurations are discarded due to the greedy sampling which entirely depends on the cost model for its selections of the configurations. Therefore, the current greedy sampling algorithm has its limitation in focusing the hardware measurements to the same region over and over.

On the other hand, we find that from a code optimization point of view, we know that many of the automated approaches for black-box optimization are prone to *invalid configurations*, which results from too large a tile that goes over the input feature map boundary or errors during memory accesses (cannot be solved analytically). These invalid configurations not only blow the chances for better exploration but also leads to an extra optimization time overhead to reset the physical hardware for the subsequent hardware measurement. We try to overcome both of these limitations by devising *Sample Synthesis*. When our compiler runs into redundant samples, the proposed synthesis method analyzes the candidate samples to determine the most probable (most frequent = mode function) non-invalid choice for each knob to come up with a new configuration. This statistical combination of the most frequent knob settings yield configurations that combine the strengths of different knobs to converge to a better overall solution. In spirit, the recombination (crossover) operator in genetic algorithms also tries to combine the best features of the solutions with high fitness values. Algorithm 1 presents the integration of our Adaptive Sampling and the Sample Synthesis.

2.4.4 Implementation Details

Architecture exploration for the adaptive exploration.

We use *Proximal Policy Optimization (PPO)* [154], a policy gradient that has been shown to adapt to various problems and have good sample complexity, as our reinforcement learning

Algorithm 1. Adaptive Sampling and Sample Synthesis

```
1: procedure AdaptiveSampling(s_Θ, v_Θ)      ▷ s_Θ: candidate configs, v_Θ: visited configs
2:     new_candidates ← ∅, previous_loss ← ∞
3:     for k in range(8, 64) do
4:         new_candidates, clusters, L2_loss ← K-means.run(s_Θ, k)
5:         if Threshold × L2_loss ≥ previous_loss then break ▷ Exit loop at knee of loss curve
6:         previous_loss ← L2_loss
7:     end for
8:     for candidate in new_candidates do          ▷ Replace visited config with new config
9:         if candidate in v_Θ then new_candidates.replace(candidate, mode(s_Θ))
10:    end for
11:    return new_candidates ▷ Feed to Code Generator to make measurements on hardware
12: end procedure
```

algorithm. Since reinforcement learning could incur computational overhead that could prolong the optimization time, we optimize the actor-critic networks through architecture exploration to find good tradeoff for size of these networks (that determines the computational overhead) and the optimization performance.

Design choices for the adaptive sampling.

We use a $\mathcal{K}$*-means Clustering* to determine centroids of the configurations, because $\mathcal{K}$-means has been shown effective in practice and it only requires $\mathcal{K}$, over error ϵ or radius in other algorithms which are much more challenging to tune. For example, DBSCAN [53] or mean-shift clustering [42] are very sensitive to the above hyperparameters. On the other hand, $\mathcal{K}$ can be framed as a *lever* to balance the performance and speed of optimizing compilation which abstracts away the aforementioned challenges, enabling the Threshold-based Swift Meta-Search that identifies the optimal number of clusters.

Hyperparameter tuning.

Hyperparameter tuning is a very important task in machine learning-based tools and models. As such, we present the hyperparameters we used for the evaluation in Table 2.2, which its tuning took several days. For the hyperparameters in Table 2.3, we used the same set of values that were used in the AutoTVM paper [35] in order to conduct a fair comparison or **CHAMELEON**.

Table 2.2. Hyper-parameters uses in **CHAMELEON**.

HYPERPARAMETER	VALUE	DESCRIPTION
$iteration_{opt}$	16	number of iterations for optimization process (equivalent to 1000 hardware measurements)
$mode_{GBT}$	xgb-reg	type of loss used for cost model
b_{GBT}	64	maximum batch size of planning in GBT [33] cost model per iteration of optimization process
$episode_{rl}$	128	number of episodes for reinforcement learning
$step_{rl}$	500	maximum steps of one reinforcement learning episode
$threshold_{meta}$	2.5	threshold used for meta-search in sampling

Table 2.3. Hyper-parameters uses in AutoTVM [35].

HYPERPARAMETER	VALUE	DESCRIPTION
$\Sigma(b_{GBT})$	1000	total number of hardware measurements
$mode_{GBT}$	xgb-reg	type of loss used for cost model
b_{GBT}	64	batch size of planning in GBT [33]
n_{sa}	128	number of Markov chains in parallel simulated annealing
$step_{sa}$	500	maximum steps of one simulated annealing run

Table 2.4. Hyper-parameters used in **CHAMELEON**'s PPO [154] search agent.

HYPERPARAMETER	VALUE
Adam Step Size	1×10^{-3}
Discount Factor	0.9
GAE Parameter	0.99
Number of Epochs	3
Clipping Parameter	0.3
Value Coefficient	1.0
Entropy Coefficient	0.1

Table 2.5. Details of the DNN models used in evaluating **CHAMELEON**.

NETWORK	DATASET	NUMBER OF TASKS
AlexNet	ImageNet	5
VGG-16	ImageNet	9
ResNet-18	ImageNet	12

Table 2.6. Details of the layers used in evaluating **CHAMELEON**.

NAME	MODEL	LAYER TYPE	TASK INDEX
L1	AlexNet	convolution	1
L2	AlexNet	convolution	4
L3	VGG-16	convolution	1
L4	VGG-16	convolution	2
L5	VGG-16	convolution	4
L6	ResNet-18	convolution	6
L7	ResNet-18	convolution	9
L8	ResNet-18	convolution	11

Additionally, for parameters used in the Adaptive Exploration module, which is not present in AutoTVM, we have tuned the hyperparameters using the set of layers presented in Table 2.6. We emphasize, however, that the *hyperparameters have been tuned offline before the deployment of* **CHAMELEON**, and the hyperparameters are not changed during the use of the framework or the experimentation. So the tuning overhead is not part of the compilation after the Adaptive Exploration module is tuned once before releasing the compiler to the deployment practitioners.

2.5 Evaluation

We integrate **CHAMELEON** into TVM [34] to perform component evaluation and compare with AutoTVM [35]. We first evaluate components of **CHAMELEON** in Section 2.5.1 and Section 2.5.2 on set of convolution layers sampled from AlexNet [96], VGG-16 [161], and

Table 2.7. Details of the hardware used for evaluation of **CHAMELEON**.

SPECIFICATIONS	DETAILS
GPU	Titan Xp
Host CPU	3.4G Hz Intel Core i7
Main Memory	32GB 2400 MHz DDR3

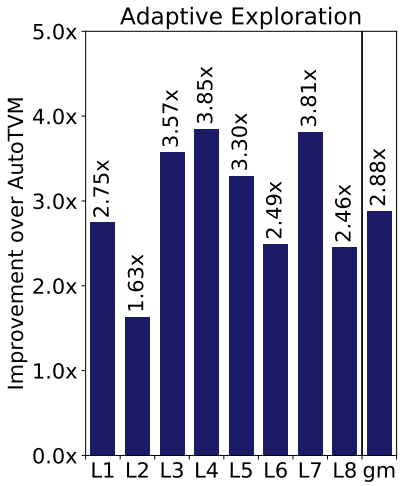
ResNet-18 [75]. Then we provide end-to-end evaluation of **CHAMELEON** on both set of layers and end-to-end deep models, in Section 2.5.3. Full details of the hardware used for the evaluation of **CHAMELEON** are provided in Table 2.7.

2.5.1 Adaptive Exploration: Improving Efficacy of Search Algorithm

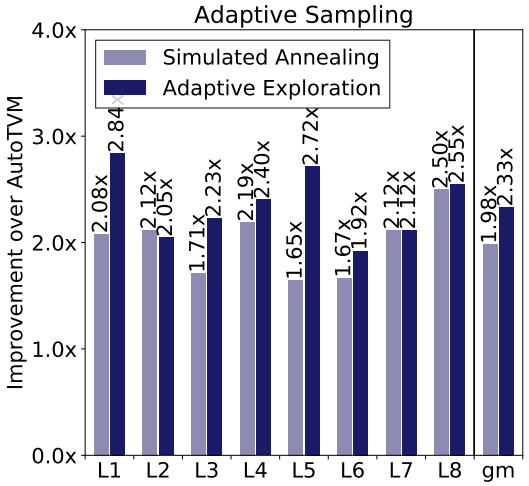
In the previous approach [35], authors have built a cost model to estimate fitness instead of performing costly measurements on real hardware, then used simulated annealing to find potentially optimal configurations. Figure 2.7(a) compares the number of search steps taken per iteration to reach or converge to the solution in simulated annealing and Adaptive Exploration, respectively. Overall, observation is that **CHAMELEON**'s Adaptive Exploration requires 2.88×less search steps compared to simulated annealing to find good solution. This comes from the ability of the reinforcement learning algorithm in Adaptive Exploration Module to (1) learn the correlation between different dimensions, and (2) reuse information across different iterations, instead of starting from scratch while naively relying on the stochastic guarantees of simulated annealing process.

2.5.2 Adaptive Sampling: Reducing Number of Costly Hardware Measurements

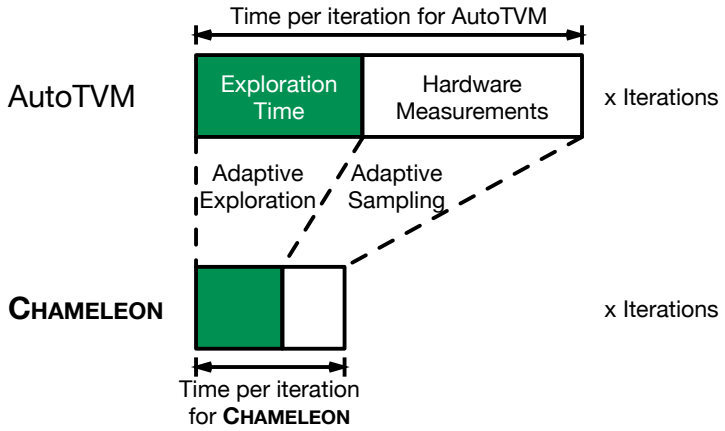
Figure 2.7(b) summarizes the effect of applying **CHAMELEON**'s Adaptive Sampling module on simulated annealing and reinforcement learning based search. First, the results show that using Adaptive Sampling helps the framework to make less hardware measurements regardless of the search algorithm used. The Adaptive Sampling algorithm reduces the number of mea-



(a) Reduction in number of steps.



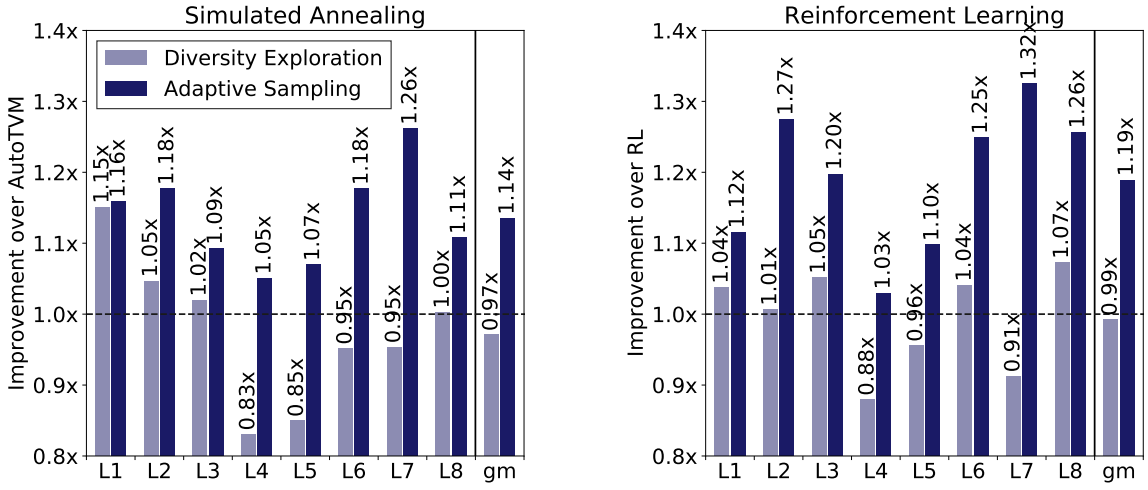
(b) Reduction in number of hardware measurements.



CHAMELEON significantly reduces optimization time

(c) Illustration of how the each component of **CHAMELEON** reduces the optimization time when compared to AutoTVM.

Figure 2.7. Component evaluation of **CHAMELEON**.



(a) Simulated Annealing.

(b) Reinforcement Learning.

Figure 2.8. Comparison to AutoTVM's diversity exploration.

surements by 1.98×when used with simulated annealing and 2.33×with reinforcement learning One observation is that the Adaptive Sampling is more effective with reinforcement learning search. This comes from the reinforcement learning agent's capacity to better localize the search to meaningful samples (*exploitation*) while still aiming to find good solution by making diverse search (*exploration*).

Diversity exploration of AutoTVM aims to spread out the candidate configurations with a regularizing effect that fosters *uniform sampling*. In contrast, our Adaptive Sampling uses a clustering algorithm to perform more measurements on the regions with higher likelihood of achieving better output performance, leading to a *non-uniform sampling*. While AutoTVM states that diversity-aware selection had no meaningful impact on most of the evaluated workloads, our Adaptive Sampling brings significant improvement as depicted in Figure 2.8. As shown, Adaptive Sampling brings an average of 13.5% and 19.0% improvement on simulated annealing and reinforcement learning, respectively.

2.5.3 Integration: Reducing Optimization Time and Output Inference Time

CHAMELEON integrates two components into the workflow: RL-based Adaptive Exploration (AE) and Adaptive Sampling (AS). This section compares the performance of **CHAMELEON** with AutoTVM [35] that leverages Simulated Annealing (SA) for its exploration.

Layer evaluation.

Figure 2.9 shows the trend of output code performance of ResNet-18's 11th layer over number of hardware measurements during optimization. The figure illustrates that our Adaptive Exploration finds better configurations than simulated annealing which results in better output code performance, and the Adaptive Sampling reduces number of hardware measurements significantly during optimization. Also, **CHAMELEON**'s Adaptive Exploration and Adaptive Sampling working in tandem emits better code with shorter optimization time than others. As such, Figure 2.10(a) compares optimization time and the performance of the output code in **CHAMELEON** and AutoTVM to confirm the observation. **CHAMELEON** achieved 1.17×better performance with 4.82×shorter optimization time compared to AutoTVM. Overall, the results suggest that our Adaptive Exploration effectively maneuvers the design space, and *Adaptive Sampling* reduces hardware measurements and the overall optimization time while even improving output performance.

End-to-end evaluation.

Up until now, we have focused on evaluation with subset of layers. Now we continue our discussion to the applicability of **CHAMELEON** to optimization of end-to-end deep neural networks. Figure 2.10(b) shows that **CHAMELEON** spends 3.59×, 5.73×, and 4.28×less time than AutoTVM to optimize AlexNet, VGG-16, and ResNet-18, respectively. On average, our work shows 4.45×optimization time speedup while achieving up to 6.4% improvement in terms of performance of output code. Inference time in Figure 2.10(b) illustrates the speedup for optimized code. Raw numbers are available in Table 2.8 and Table 2.9. All in all, such

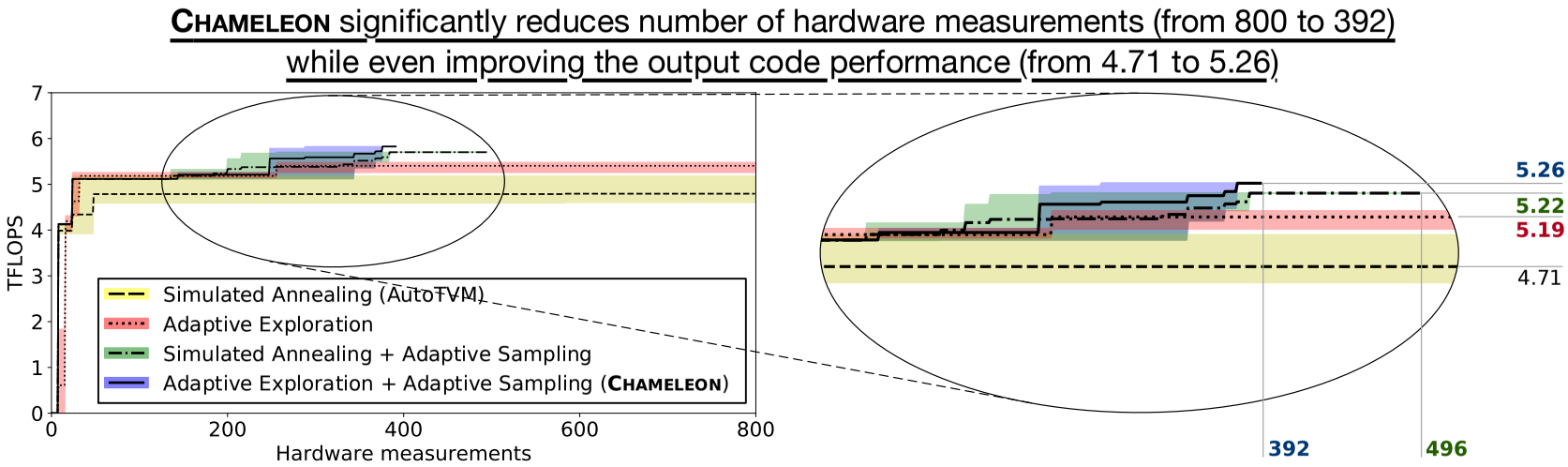
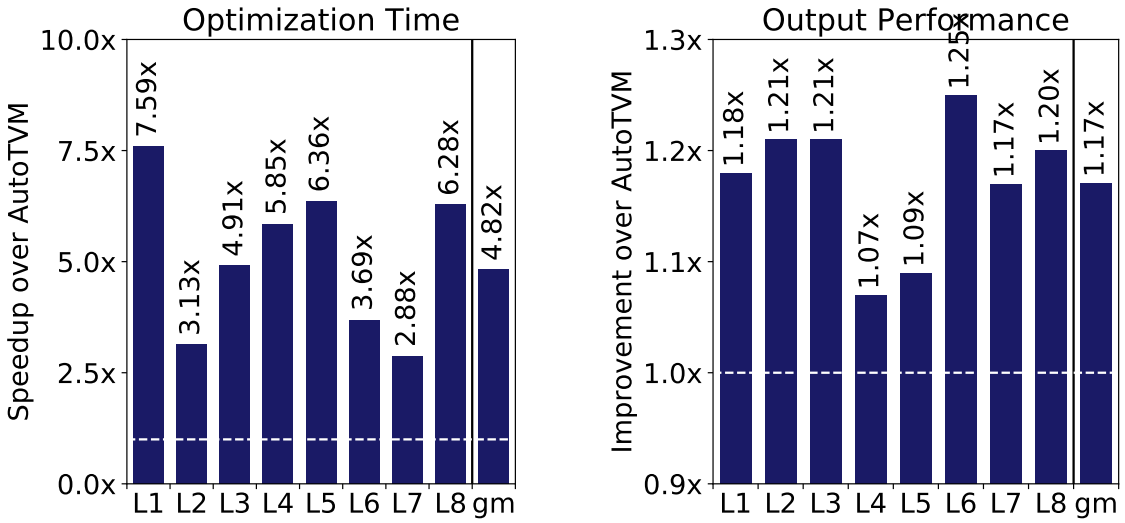
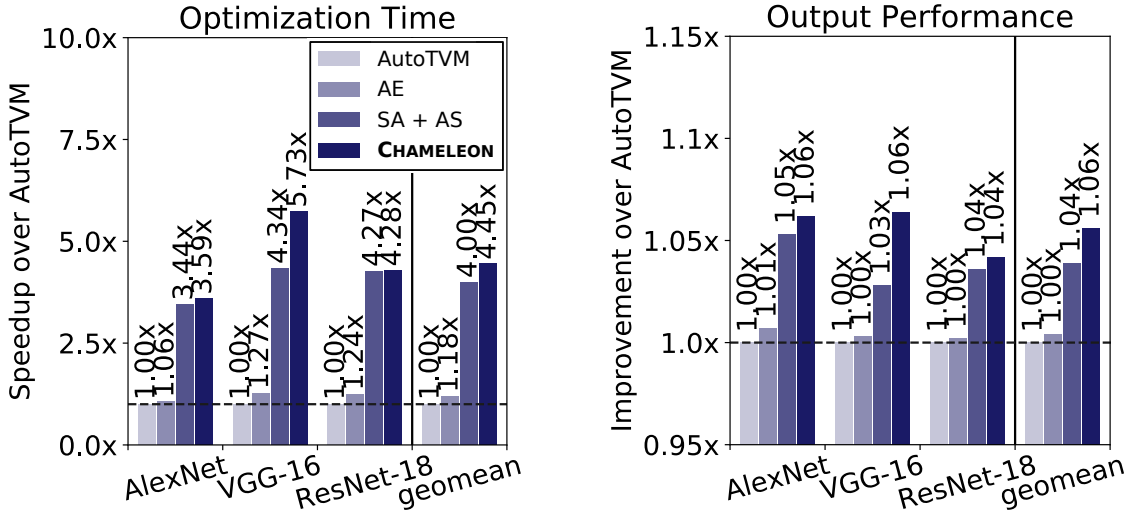


Figure 2.9. Layer evaluation of output performance for ResNet-18's 11th layer.



(a) Layer evaluation.



(b) End-to-end evaluation.

Figure 2.10. Layer and end-to-end evaluation. Dashed lines denote AutoTVM's performance.

Table 2.8. End-to-end evaluation of the optimization time for deep networks.

NETWORK	SA (AutoTVM)	AE	SA + AS	AE + AS (**CHAMELEON**)
AlexNet	4.31 Hours	4.06 Hours	1.25 Hours	**1.20 Hours**
VGG-16	11.18 Hours	8.82 Hours	2.57 Hours	**1.95 Hours**
ResNet-18	9.13 Hours	7.39 Hours	2.14 Hours	**2.13 Hours**

Table 2.9. End-to-end evaluation of the output performance for deep networks.

NETWORK	SA (AutoTVM)	AE	SA + AS	AE + AS (**CHAMELEON**)
AlexNet	1.0277 ms	1.0207 ms	0.9762 ms	**0.9673 ms**
VGG-16	3.9829 ms	3.9710 ms	3.8733 ms	**3.8458 ms**
ResNet-18	1.0258 ms	0.9897 ms	0.9897 ms	**0.9831 ms**

improvements result from efficient Adaptive Exploration and the reduced number of hardware measurements from Adaptive Sampling.

2.6 Related Works

CHAMELEON uniquely offers a solution that exclusively enables (i) *Reinforcement Learning* and (ii) *Sampling* in the context of (iii) *Optimizing Compilers* for neural networks. As such, we discuss the related work from each of the three independent research directions.

Optimizing compilers.

TensorComprehensions [170] and TVM [34] use genetic algorithm and simulated annealing to choose parameters of polyhedral optimization for neural networks. In a more general context, some computing libraries [58, 175] make use of black box optimization and also profiling-based compilation passes [31, 128] utilize runtime information to generate optimized code. Later, AutoTVM [35] incorporates learning with boosted trees within the cost model for TVM to reduce the number of real hardware measurements. While **CHAMELEON** is inspired and builds on these prior works, unlike them, it is based on reinforcement learning for *Adaptive Exploration*, and *Adaptive Sampling* that leverages clustering to reduce the number of measurements.

Reinforcement learning for hyper-parameter optimization.

There are a growing body of studies on using reinforcement learning to perform various optimizations [61, 111, 112, 120, 126, 182] for a variety of objectives including hyper-parameter optimization for neural networks. For instance, DeepArchitect [127] and NAS [195] use reinforcement learning to automate the process of designing deep neural network models and their associated parameters. HAQ [174] and ReLeQ [50] use reinforcement learning to chose levels of quantization for the layers of a given deep neural network. AMC [76] formulates neural network compression as a RL problem. [132] combined RL with graph neural networks and genetic algorithms to optimize DNN execution. Our work exclusively explores a different problem, that

is optimizing compilers using reinforcement learning.

Sampling algorithms for learning.

Active learning is a broad field [30, 41, 65, 156, 163, 180] that uses a measure of the change in the model to decide which training data elements should be used to update the model. Passive learning [131, 187] is an alternative view that independent of the model, analyze the distribution of the training data set and selects a subset. The Adaptive Sampling algorithm for **CHAMELEON** shares similarities with Passive learning but it differs in its context. The sampling is designed to reduce the number of samples (configuration) for hardware measurement from the exploration of the design space whilst performing an optimization to accelerate the process.

2.7 Conclusion

We present **CHAMELEON** to allow optimizing compilers to adapt to unseen design spaces of code schedules to reduce the optimization time. This paper is also an initial effort to bring *reinforcement learning* to the realm of optimizing compilers for neural networks, and we also develop an *Adaptive Sampling* with domain-knowledge inspired *Sample Synthesis* to not only reduce the number of samples required to navigate the design space but also augment its quality in terms of fitness. Experimentation with real-world deep models shows that **CHAMELEON** not only reduces the time for compilation significantly, but also improves the quality of the code. This encouraging result suggests a significant potential for various learning techniques to optimizing deep learning models.

Acknowledgement. Chapter 2, in part, contains a re-organized reprint of the material as it appears in International Conference on Learning Representations (ICLR) 2020. Ahn, Byung Hoon; Pilligundla, Prannoy; Yazdanbakhsh, Amir; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Foundational Algorithms for Optimized Execution of AI

Section 2 explored the use of reinforcement learning, an *AI Algorithm* in deep learning compilers. The use of reinforcement learning augmented the compiler with adaptation capability that improved both compilation time and the execution speed of the DNN model. Overall, significant benefits from applying reinforcement learning to compilation suggest significant potential in integrating AI algorithm to compilers. In fact, many research including, but not limited to, [3, 103, 144, 145, 166, 191] confirms that various AI algorithms can help explore the exponentially large search spaces for compilation. However, an important question here is whether this exciting result means that we should relinquish the conventional methods that have enabled faster computing for a long time. To answer this question, this section dives into the use of a *Foundational Algorithm: Dynamic Programming* [21] in the context of compilation. More specifically, this section utilizes dynamic programming for memory-aware scheduling of the deep learning models' computational graph.

3.1 Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices

Recent advance on automating machine learning through Neural Architecture Search and Random Network Generators, has yielded networks that deliver higher accuracy given the

same hardware resource constrains, e.g., memory capacity, bandwidth, number of functional units. Many of these emergent networks; however, comprise of irregular wirings (connections) that complicate their execution by deviating from the conventional regular patterns of layer, node connectivity, and computation. The irregularity leads to a new problem space where the schedule and order of nodes significantly affect the activation memory footprint during inference. Concurrently, there is an increasing general demand to deploy neural models onto resource-constrained edge devices due to efficiency, connectivity, and privacy concerns. To enable such a transition from cloud to edge for the irregularly wired neural networks, we set out to devise a compiler optimization that caps and minimizes the footprint to the limitations of the edge device. This optimization is a search for the schedule of the nodes in an intractably large space of possible solutions. We offer and leverage the insight that partial schedules leads to repeated subpaths for search and use the graph properties to generate a signature for these repetition. These signatures enable the use of *Dynamic Programming* as a basis for the optimization algorithm. However, due to the sheer number of neurons and connections, the search space may remain prohibitively large. As such, we devise an *Adaptive Soft Budgeting* technique that during dynamic programming performs a light-weight meta-search to find the appropriate memory budget for pruning suboptimal paths. Nonetheless, schedules from any scheduling algorithm, including ours, is still bound to the topology of the neural graph under compilation. To alleviate this intrinsic restriction, we develop an *Identity Graph Rewriting* scheme that leads to even lower memory footprint without changing the mathematical integrity of the neural network. We evaluate our proposed algorithms and schemes using representative irregularly wired neural networks. Compared to TensorFlow Lite, a widely used framework for edge devices, the proposed framework provides $1.86\times$reduction in memory footprint and $1.76\times$ reduction in off-chip traffic with an average of less than one minute extra compilation time.

3.2 Introduction

Growing body of work focuses on Automating Machine Learning (AutoML) using Neural Architecture Search (NAS) [29, 38, 43, 106, 109, 140, 195, 196] and now even, Random Network Generators [178, 181] which emit models with *irregular* wirings, and shows that such *irregularly wired neural networks* can significantly enhance classification performance. These networks that deviate from *regular topology* can even adapt to some of the constraints of the hardware (e.g., memory capacity, bandwidth, number of functional units), rendering themselves especially useful in targeting edge devices. Therefore, lifting the regularity condition provides significant freedom for NAS and expands the search space [38, 43, 181].

The general objective is to enable deployment of neural intelligence even on stringently constrained devices by trading off regular wiring of neurons for higher resource efficiency. Importantly, pushing neural execution to edge is one way to address the growing concerns about privacy [117] and enable their effective use where connectivity to cloud is restricted [179]. However, the new challenge arises regarding orchestrating execution of these irregularly wired neural networks on the edge devices as working memory footprint during execution frequently surpass the strict cap on the memory capacity of these devices. The lack of multi-level memory hierarchy in these micro devices exacerbates the problem, because the network cannot even be executed if the footprint exceeds the capacity. To that end, despite the significant potential of irregularly wired neural networks, their *complicated execution pattern*, in contrast to previously *streamlined execution of models with regular topology*, renders conventional frameworks futile in taking these networks to edge due to their large *peak memory footprint*. While peak memory footprint is largely dependent on scheduling of neurons, current deep learning compilers [34, 170] and frameworks [1, 85, 135] rely on basic topological ordering algorithms that are oblivious to peak memory footprint and instead focus on an orthogonal problem of tiling and kernel level optimization. This paper is an initial step towards embedding peak memory footprint as first-grade constraint in deep learning schedulers to unleash the potential of the emergent

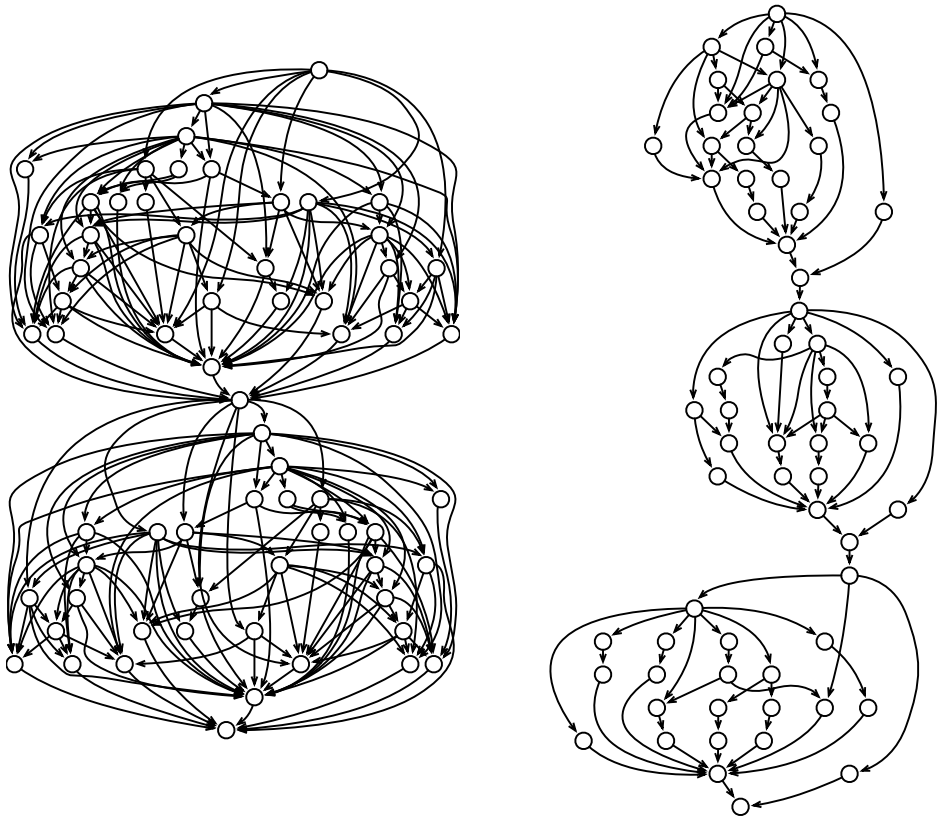
irregularly wired neural networks. As such, this paper makes the following contributions:

(1) Memory-aware scheduling for irregularly wired neural networks. Scheduling for these networks is a topological ordering problem, which enumerates an intractably large space of possible schedules. We offer and leverage the insight that partial schedules leads to repeated subpaths for search and use the graph properties to generate a signature for these repetition while embedding a notion of the running memory usage. These signatures enable the use of *Dynamic Programming* as a basis for the optimization algorithm.

(2) Adaptive soft budgeting for tractable compilation time. Even with the dynamic programming as the base, due to the sheer number of neurons and connections, the search space may remain too large (exponentially large) in practice. As such, we devise an *Adaptive Soft Budgeting* technique that uses a lightweight meta-search mechanism to find the appropriate memory budget for pruning the suboptimal paths. This technique aims to find an inflection point beyond which tighter budgets may lead to no solution and looser budget prolongs the scheduling substantially, putting the optimization in a position of questionable utility.

(3) Identity graph rewriting for enabling higher potential in memory reduction. Any scheduling algorithm, including ours, is still bound to the topology of the neural graph under compilation. To relax this intrinsic restriction, we devise an *Identity Graph Rewriting* scheme that exchanges subgraphs leading to a lower memory footprint without altering the mathematical integrity of the neural network.

Results show that our adaptive scheduling algorithm improves peak memory footprint for irregularly wired neural networks by $1.68\times$compared to TensorFlow Lite, the de facto framework for edge devices. Our graph rewriting technique provides an opportunity to lower the peak memory footprint by an additional 10.7%. Furthermore, our framework can even bring about $1.76\times$ reduction in off-chip traffic for devices with multi-level memory hierarchy, and even eliminate the traffic in some cases by confining the memory footprint below the on-chip memory capacity. These gains come at average of less than one minute extra compilation time.



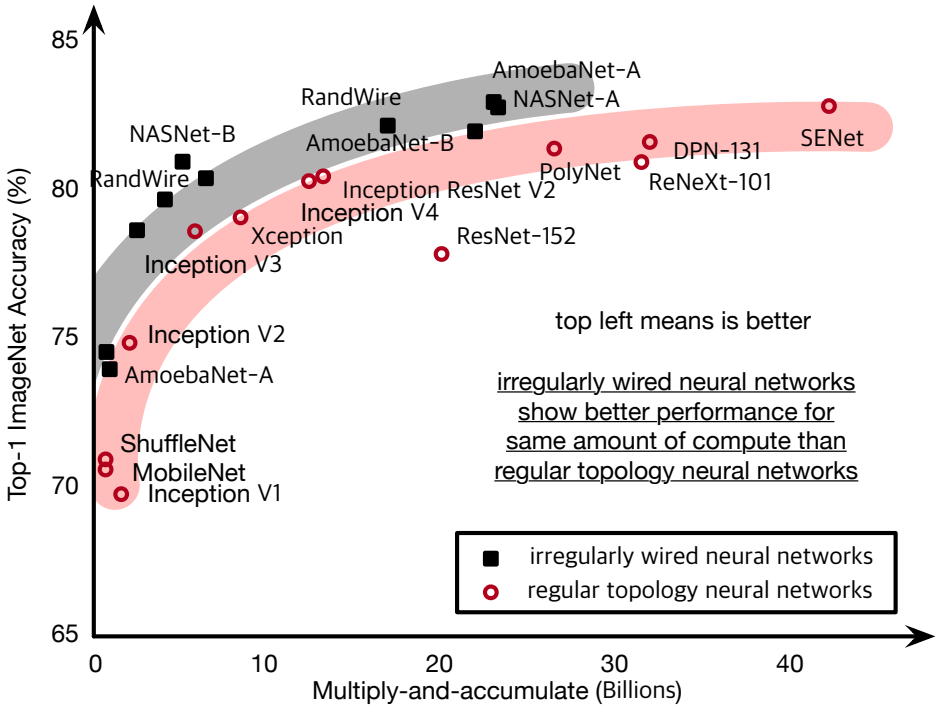
(a) RandWire (b) SwiftNet

Figure 3.1. Architecture of network models from NAS and Random Network Generators. Topology of such networks include distinctive *irregular wirings* between the nodes.

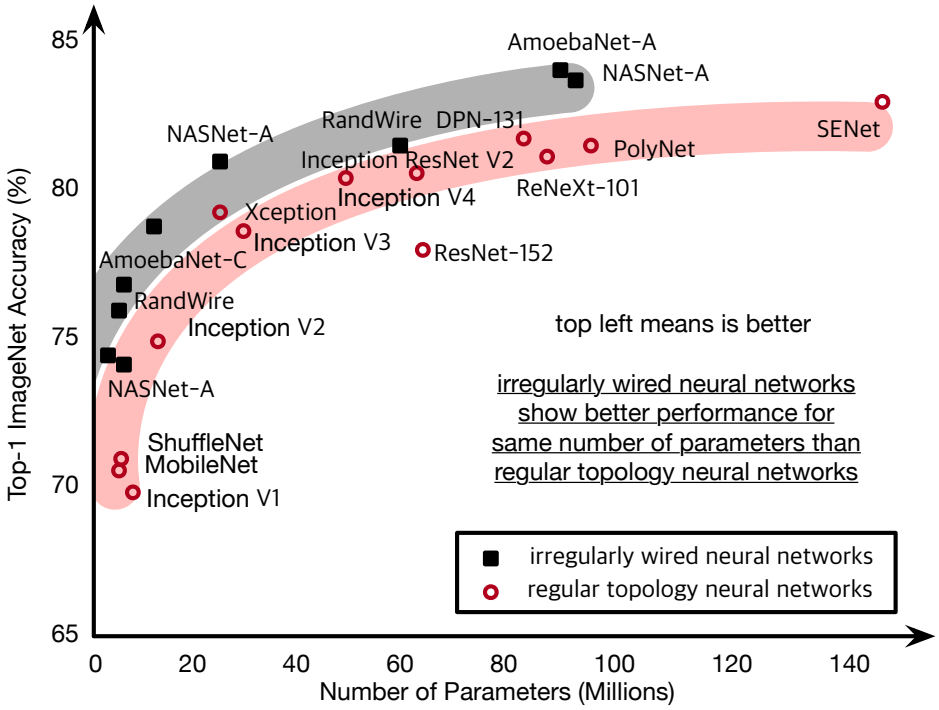
3.3 Challenges and Our Approach

3.3.1 Irregularly Wired Neural Networks

Recent excitement in Automated Machine Learning (AutoML) [46, 50, 54, 76, 98, 174] aims to achieve *human out of the loop* in developing machine learning systems. This includes Neural Architecture Search (NAS) [29, 38, 106, 140, 195, 196] and Random Network Generators [178, 181] that focus on automation of designing neural architectures. Figure 3.1 demonstrates that networks of this regime are characterized by their distinctive *irregular graph topology* with much more irregular wirings (dataflow) compared to conventional networks with regular graph topology. This paper refers to these networks as *irregularly wired neural networks*.

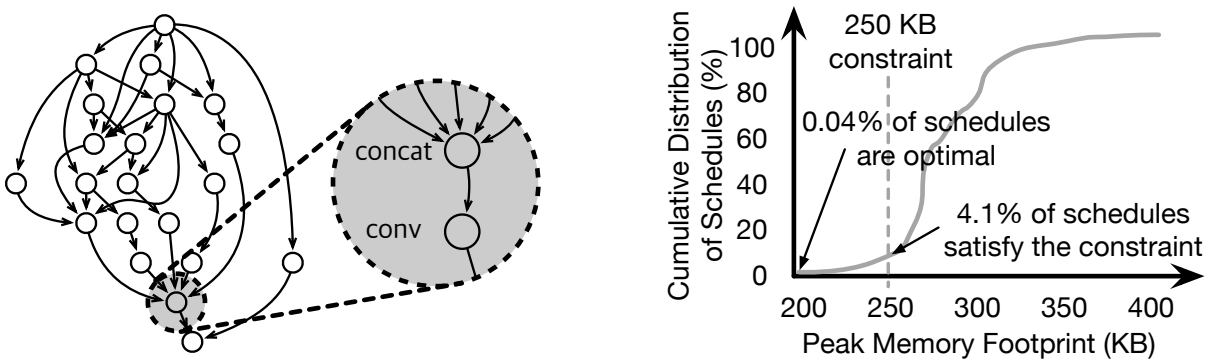


(a) ImageNet accuracy vs number of multiply-and-accumulate.



(b) ImageNet accuracy vs number of parameters.

Figure 3.2. ImageNet accuracy vs number of multiply-and-accumulate or parameters, where irregularly wired neural networks show higher performance for same amount of compute or number of parameters than regular topology neural networks.



(a) SwiftNet Cell A.

(b) CDF of peak memory for different possible schedules.

Figure 3.3. CDF of the peak memory footprint for the different possible schedules of a given *irregularly wired neural network.*

From the performance perspective, these networks have shown to outperform manually designed architectures in terms of accuracy while using less resources. In fact, majority of winning neural architectures in competitions with primary goal of reducing resources [62] rely on NAS, suggesting its effectiveness in that respect. Figure 3.2 plots the accuracy of different models given their computation. The figure clearly shows that the Pareto frontier of irregularly wired neural networks from NAS and Random Network Generators are better than the hand designed models with regular topology. This indicates that the efficiency in terms of accuracy given fixed resources are better with the irregularly wired neural networks.

3.3.2 Challenges

Many existing compilers [34, 170] and frameworks [1, 85, 135] rely on basic topological ordering algorithms to schedule the graph. While the current approach may be sufficient to run conventional networks on server-class machines, such scheme may be unfit for running irregularly wired neural networks on resource-constrained edge devices. This is because, unlike running networks with regular topology, running irregular networks results in varied range of memory footprint depending on the schedule. For instance, given the constraints of a representative edge device (SparkFun Edge: 250KB weight/activation memory and 60M MACs), Figure 3.3(b) shows

that 4.1% of the schedules barely meets the hard memory constraint, while only 0.04% would achieve the optimal peak memory. In reality, such limitation will prevent further exploration regarding the diversity and innovation of network design, and in order to allow edge computing regime to take full advantage of the irregularly wired neural networks, this limitation should be alleviated if not removed.

3.3.3 Design Objectives

Scheduling algorithm.

To address this issue, our work aims to find a schedule of nodes s^* from the search space $\mathcal{S}$ that would minimize peak memory footprint μ_{peak}. $\mathcal{S}$ enumerates all possible orderings of the nodes $v \in \mathcal{V}$ where $\mathcal{V}$ is the set of all nodes within a graph $\mathcal{G}$.

$$s^* = \underset{s}{\operatorname{argmin}}\, \mu_{peak}(s, \mathcal{G}), \qquad \text{for } s \in \mathcal{S} \tag{3.1}$$

The most straightforward way to schedule is a *brute force* approach which just enumerates $\mathcal{S}$ and picks one with the minimum peak memory footprint. While this extreme method may find an optimal solution, it is too costly in terms of time due to its immense complexity: $\Theta(|V|!)$ where $|V|$ denotes number of nodes in the graph. One way to improve is to narrow down the search space to just focus on only the *topological orderings* $\mathcal{S}_T \subset \mathcal{S}$. However, this will still suffer from a complexity with an upper bound of $\mathcal{O}(|V|!)$ (takes days to schedule DAG with merely 30 nodes). In fact, previous works [24, 27] already prove optimal scheduling for DAGs is NP-complete. On another extreme are heuristics for topological ordering such as Kahn's algorithm [91], with complexity of $\mathcal{O}(|V| + |E|)$ where V and E are number of nodes and edges. However, as demonstrated in Figure 3.3, such method may yield suboptimal schedule of nodes which will not run on the target hardware. To this end, we explore *dynamic programming* combined with *adaptive soft budgeting* for scheduling to achieve an optimal solution while keeping the graph constant s^*, without adding too much overhead in terms of time. We explain our algorithms in depth in Section 3.4.1 and 3.4.2.

Graph rewriting.

Any scheduling algorithm including ours is intrinsically bounded by the graph topology. Therefore, we explore to transform the search space through *graph rewriting* [138]. Graph rewriting is generally concerned with substituting a certain pattern in the graph with a different pattern to achieve a certain objective. For a computational dataflow graph, leveraging *distributive, associative, and commutative properties* within the computation of the graph, graph rewriting can maintain the semantics while bringing significant improvements regarding some objective. For example, in general programs, $\sum_i log x_i$ can be represented as $\sum_{oddi} log x_i + \sum_{eveni} log x_i$ or $log \prod_i x_i$, while $x + x$ can be translated to $x \times 2$ or $x << 1$. Likewise, we bring this insight to neural networks to find a set of possible transformations $\mathcal{X}$ that can rewrite the original graph $\mathcal{G}$ to a new graph $\mathcal{G}'$ that would also change our search space $\mathcal{S}$ to one with a lower peak memory footprint:

$$\mathcal{X}^* = \underset{\mathcal{X}}{\operatorname{argmin}}(\mu_{peak}(s^*, \mathcal{X}(\mathcal{G}))) \tag{3.2}$$

We identify a set of candidate patterns for transformation $\chi : g \rightarrow g'$ ($g \in \mathcal{G}$ and $g' \in \mathcal{G}'$), which constitutes $\mathcal{X}$. While transforming the graph, our method keeps the *mathematical integrity* of the graph intact, thus not an approximation method. We embed this systematic way to improve peak memory footprint and the search space as *identity graph rewriting*, and we address this technique in Section 3.4.3.

3.4 SERENITY: Memory-Aware Scheduling of Irregularly Wired Neural Networks

As discussed in Section 3.3, the objective is reducing the peak memory footprint while executing irregularly wired neural networks. We propose **SERENITY**, memory-aware scheduling that targets devices with restricted resources (e.g., edge devices). Figure 3.4 summarizes the overall scheduling process, highlighting the major contributions of our approach. Input to

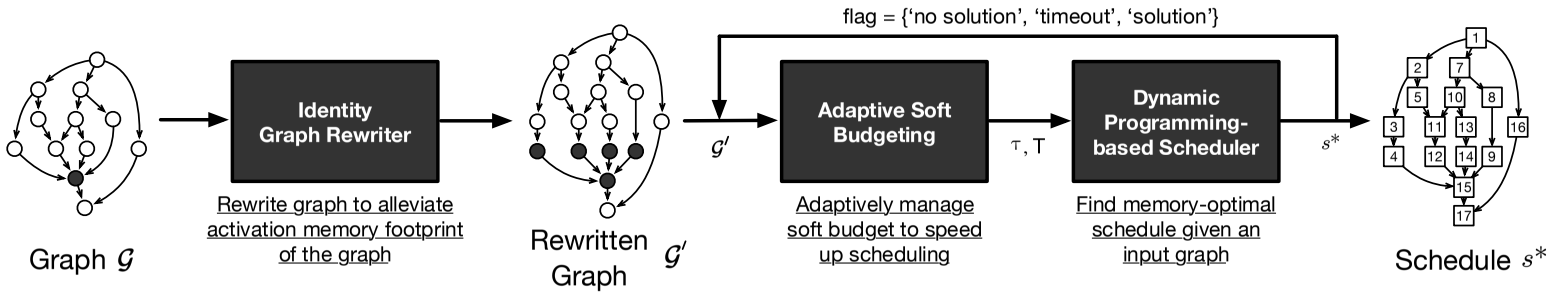


Figure 3.4. Overall workflow of **SERENITY**, memory-aware scheduling of *irregularly wired neural network*.

SERENITY is a graph of irregularly wired neural network $\mathcal{G}$, which in fact acts as an intermediate representation (IR) during the scheduling process. We augment this IR with the metadata of the nodes such as the operation type, input/output edges, input/output shapes, and memory cost. Then the *graph rewriter* transforms the graph $\mathcal{G} \rightarrow \mathcal{G}'$ to relax the memory costs of memory intensive patterns with the goal of reducing the peak memory footprint μ_{peak} of $\mathcal{G}$. **SERENITY** schedules the graph to an optimal schedule s^* using the *dynamic programming-based scheduler*. However, since the scheduling may be slow due to the complexity, we scale down search space by leveraging *divide-and-conquer* which partitions the graph into multiple subgraphs. Them, we augment the scheduler with an *adaptive soft budgeting* which prunes suboptimal paths by adaptively finding a budget for thresholding through a swift meta-search to speed up the scheduling process. This section focuses on the innovations of **SERENITY**: dynamic programming-based scheduling, divide-and-conquer, adaptive soft budgeting, and graph rewriting, which are explained in detail in Section 3.4.1, 3.4.2, and 3.4.3, respectively.

3.4.1 Dynamic Programming-based Scheduling: Achieving Optimal Peak Memory Footprint

Our goal for the scheduling algorithm is to minimize the peak memory footprint $\mu_{peak}(s, \mathcal{G})$. As stated in Section 3.3.3, recursive algorithms that covers the entire search space $\mathcal{S}$ or the subspace of all topological orderings $\mathcal{S}_T \subset \mathcal{S}$ takes impractically long time. This is primarily due to the repetitive re-computation of subproblems that upper bounds the algorithm by $\mathcal{O}(|V|!)$. Therefore, we leverage *dynamic programming* [21, 22, 77] which includes a *memoization* scheme that has been shown to be effective in reducing the complexity of time-intensive algorithms by reusing solutions from their subproblems, while still finding optimal solution by sweeping the entire search space.

Identifying signature to enable dynamic programming.

The first step to applying dynamic programming to a new problem is characterizing the structure of an optimal solution: $s^* = s^*_n$ (s^*_n is an optimal solution for n number of nodes). Then,

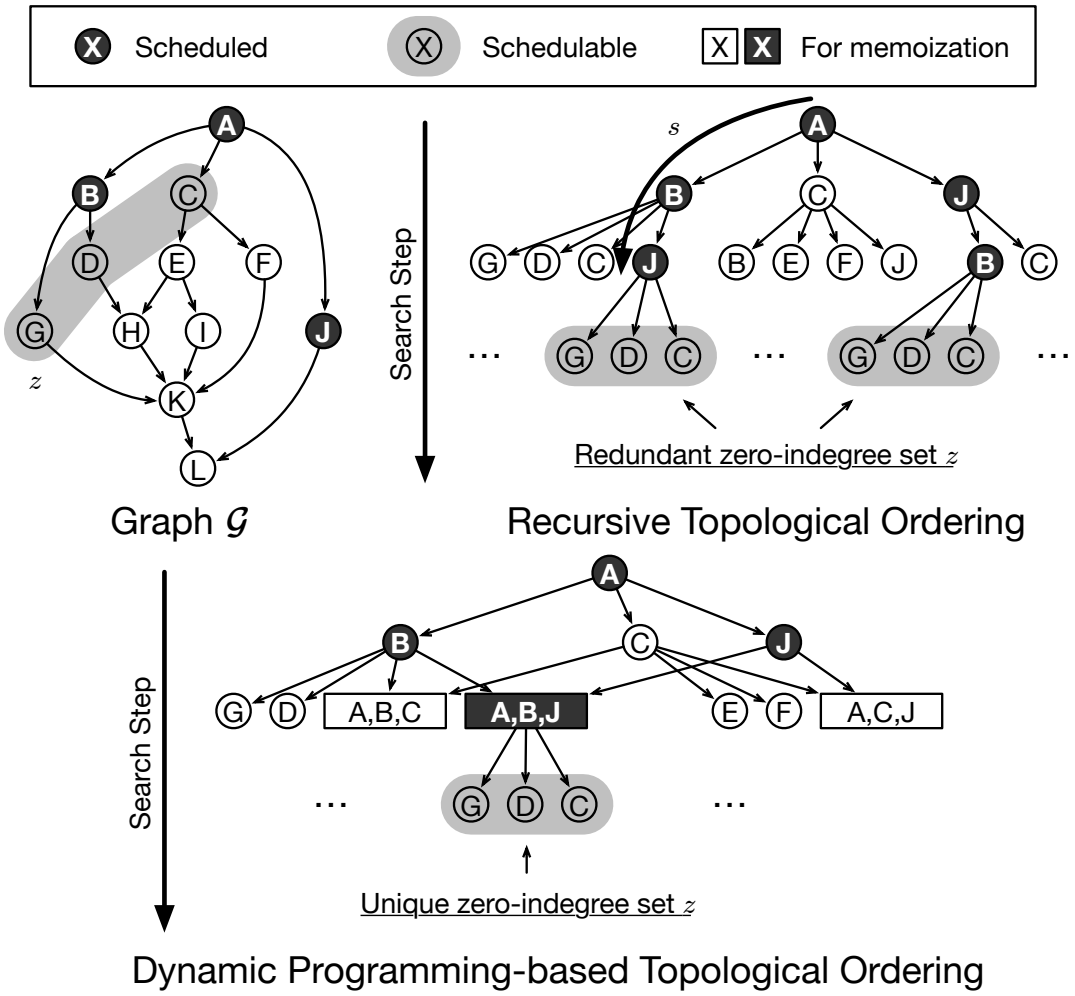


Figure 3.5. Illustration of identifying redundant *zero-indegree* set z and making z unique (*square*) throughout the topological ordering algorithm to reduce re-computation.

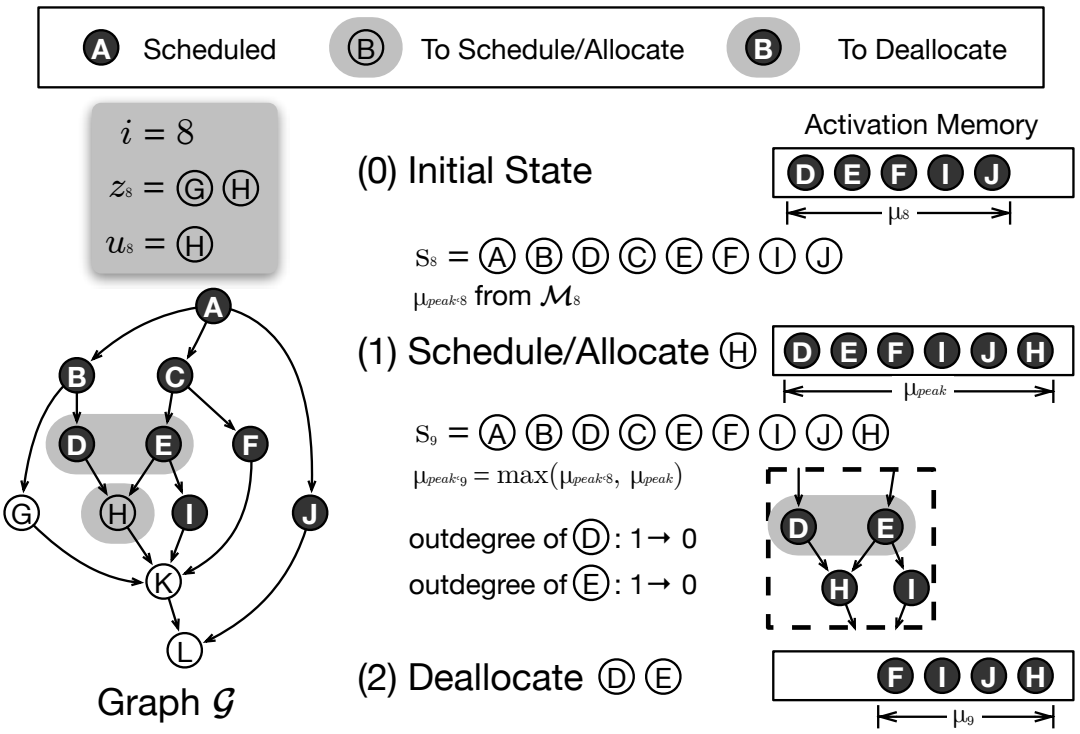


Figure 3.6. Visualization of scheduling the node $u_8 = \text{H}$ during the search step $i = 8$. Starting from s_8, μ_8, and $\mu_{peak,8}$ the figure shows how the algorithm calculates s_9, μ_9, and $\mu_{peak,9}$

it requires identifying a recursive relationship between the optimal solution of a subproblem s_i^* and the original problem s_{i+1}^*, and we do this by analyzing the straightforward *recursive topological ordering*, which while inefficient sweeps the entire search space. In essence, topological ordering algorithm is a repeated process of identifying a set of nodes that are available for scheduling and iterating the set for recursion. In graph theory such a set of nodes available for scheduling is called *zero-indegree set* z, where z is a set of nodes which all of their incoming edges and the corresponding predecessor nodes (*indegree*) have been scheduled. Figure 3.5 demonstrates the recursion tree of the different topological ordering algorithms, where the height of the tree is the search step and every path from the root to the leaf is a topological ordering $s \in S_T$. The figure highlights the redundant z in the recursive topological ordering in the recursion tree, then merges these z to make them unique, identifying it as the signature for repetition, and prevent the aforementioned re-computation. This makes the scheduling for z into a unique subproblem, that constitutes the *dynamic programming-based topological ordering*.

Integrating the peak memory footprint constraint.

On top of the dynamic programming formulation that shows potential for optimizing the search space significantly, we overlay the problem specific constraints to achieve the optimal solution. In particular, we calculate the *memory footprint* μ_{i+1} and its corresponding *peak* $\mu_{peak,i+1}$ in each search step i to select optimal path s^*_{i+1} for *memoization*. Here, we clarify the process of a search step, explaining the details of calculating $\mu_{peak,i+1}$ and saving s_{i+1} for each search step i. In each search step, we start with number of *unique* zero-indegree sets z_i (signature), saved in i^{th} entry of memoization $\mathcal{M}_i$. For each z_i, we append the schedule up to the point s_i, sum of activations in the memory μ_i for the signature z_i, and the peak memory footprint of the s_i denoted $\mu_{peak,i}$. Therefore, in each search step i, we start with s_i, μ_i, and $\mu_{peak,i}$ for s_i. Then, when we iterate z_i to schedule a new node u_i, its output activation is appended to s_i to form s_{i+1}, and is *allocated* in the memory. Size of u_i is product ($\prod$) of u_i.shape, where shape is a property of the activation tensor that includes channels, height, width, and the precision (e.g., byte, float), is added to μ_i, so $\mu_{i+1} \leftarrow \mu_i + \prod(u_i.\text{shape})$. Then we use μ_{i+1} as μ_{peak} to update $\mu_{peak,i+1}$ (peak memory footprint for s_{i+1}). Since some predecessors of u_i will not be used anymore after allocating u_i, we update the *outdegrees* of the node by decrementing them. Having updated the outdegree, we will be left with a *zero-outdegree set* that denotes the nodes that are ready for deallocation. We *deallocate* the nodes in the set and update μ_{i+1} accordingly.

To demonstrate scheduling of a node u_i, Figure 3.6 simulates scheduling a node u_8 = Ⓗ in $i = 8$. In the figure, (1) Ⓗ is appended to s_8 and allocated to memory as it is scheduled, and then the scheduler records maximum of the $\mu_{peak,8}$ and the sum of all activations in the memory at this point as $\mu_{peak,9}$. Then, it recalculates the outdegrees of the predecessor nodes of Ⓗ: Ⓓ and Ⓔ's outdegree are decremented from one to zero. (2) Then these nodes are deallocated and sum of the activation memory here is recorded as μ_9.

Finding schedule with optimal peak memory footprint.

After scheduling u_i, we save the new signature into the $\mathcal{M}_{i+1}$ for next search step $i+1$. Since the goal of this work is to minimize the overall μ_{peak}, we identify the corresponding optimal schedule s^*_{i+1} for each z_{i+1} by only saving s_{i+1} with the minimum $\mu_{peak,i+1}$. We integrate the aforementioned step of scheduling u_i and updating $\mathcal{M}_{i+1}$ to complete the proposed *dynamic programming-based scheduling algorithm*. Algorithm 2 summarizes the algorithm. As a first step, the algorithm starts by initializing the memoization table $\mathcal{M}_0$, then the algorithm iterates different search steps. In each search step i, the algorithm performs the above illustrated memory allocation for all u_i in z_i, and saving s_{i+1}, μ_{i+1}, and $\mu_{peak,i+1}$. After iterating all search steps to $n-1$, s_* is saved in $\mathcal{M}_n$ with a unique entry, for n being number of nodes in $\mathcal{G}$.

Proof of the algorithm.

Here we prove the optimality of the above dynamic programming-based scheduling algorithm.

Proof

THEOREM 1. *In order to find a schedule s^* with an optimal peak memory consumption μ^*, it is sufficient to keep just one schedule-peak memory pair (s_i, z_i) in S_{Ti} for each zero-indegree set z_i, and to append subsequent nodes on top of s_i to get s_{i+1} in each search step.*

Proof. If $i = 0$, the optimal s_0 is an empty sequence and μ_0 must be 0. On the other hand, if $i \geq 1$, assume that (*suboptimal*) v_i constitutes s^*, substituting $u^*_i \in z_i$ and achieves μ^*. In such case, let v_i be replaced with (*optimal*) u^*_i, which will result in $\mu_{peak} \leftarrow \min(\mu_i + \prod v_i.\text{shape}, \mu_i + \prod u^*_i.\text{shape})$, and μ_{i+1} is calculated by deducting $\prod p_i.\text{shape}, \forall p_i \in (u_i.\text{preds} \cap \text{zero-outdegree}(s_{i+1}, \mathcal{G}))$. By recursively applying u_k for rest of the search steps k, the algorithm should find an alternative sequence $s^{*\prime}$ with $\mu^{*\prime} \leq \mu^*$ due to the $\min$ operator above, contradicting the original assumption on the optimality of s^*. Therefore, our algorithm finds a schedule with an optimal peak memory consumption. ■

Algorithm 2. Dynamic Programming-based Scheduling

Input: graph $\mathcal{G}$
Output: optimal schedule s^*
// initialize memoization
$s_0 \leftarrow []$, $\mu_0, \mu_{peak,0} \leftarrow 0$, $z_0 \leftarrow$ zero-indegree$(s_0, \mathcal{G})$
$\mathcal{M}_0[z_0] \leftarrow (s_0, \mu_0, \mu_{peak,0})$
// iterate search step
for $i = 0$ **to** $n - 1$ **do**
    // iterate (schedule, current memory, peak memory)
    for $z_i, (s_i, \mu_i, \mu_{peak})$ **in** $\mathcal{M}_i$ **do**
        for u_i **in** z_i **do**
            $s_{i+1} \leftarrow s_i$.append(u_i) // allocate
            $z_{i+1} \leftarrow$ zero-indegree$(s_{i+1}, \mathcal{G})$
            $\mu_{i+1}, \mu_{peak} \leftarrow \mu_i + \prod(u_i.\text{shape})$
            $\mu_{peak,i+1} \leftarrow \max(\mu_{peak,i}, \mu_{peak})$
            for p_i **in** u_i.preds **do**
                if p_i **is in** zero-outdegree$(s_{i+1}, \mathcal{G})$ **then**
                    $\mu_{i+1} \leftarrow \mu_{i+1} - \prod(p_i.\text{shape})$ // deallocate
                end if
            end for
            // memoize schedule with least peak memory
            if $\mu_{peak,i+1} \leq \mathcal{M}_{i+1}[z_{i+1}].\mu_{peak,i+1}$ **then**
                $\mathcal{M}_{i+1}[z_{i+1}] \leftarrow (s_{i+1}, \mu_{i+1}, \mu_{peak,i+1})$
            end if
        end for
    end for
end for
$s^*, \mu^*_{peak} \leftarrow \mathcal{M}[\cdot]_n.s_n, \mathcal{M}[\cdot]_n.\mu_{peak,n}$ // solution

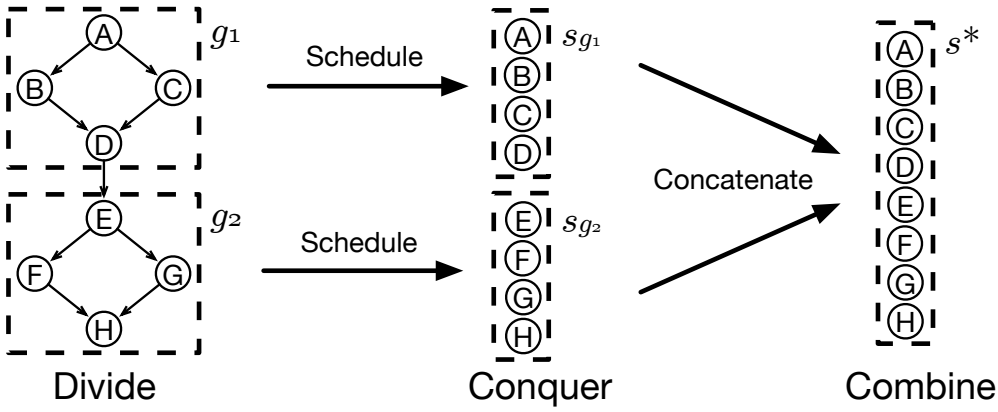


Figure 3.7. Illustration of *divide-and-conquer*, which divides the graphs into multiple subgraphs (*divide*), schedules each of them using the optimal scheduler (*conquer*), then concatenates the sub-schedules to get the final schedule (*combine*).

Complexity of the algorithm.

The complexity of the proposed dynamic programming-based scheduling is $\mathcal{O}(|V| \times 2^{|V|})$, which is significantly faster than the exhaustive search of $\mathcal{S}_T$ with an upper bound complexity of $\mathcal{O}(|V|!)$.

3.4.2 Optimizing Scheduling Speed: Speeding up the Dynamic Programming-based Scheduling

While the above scheduling algorithm improves complexity of the search, search space may still be intractable due to the immense irregularity. Therefore, we devise *divide-and-conquer* and *adaptive soft budgeting* to accelerate the search by effectively shrinking and pruning the search space.

Divide-and-conquer.

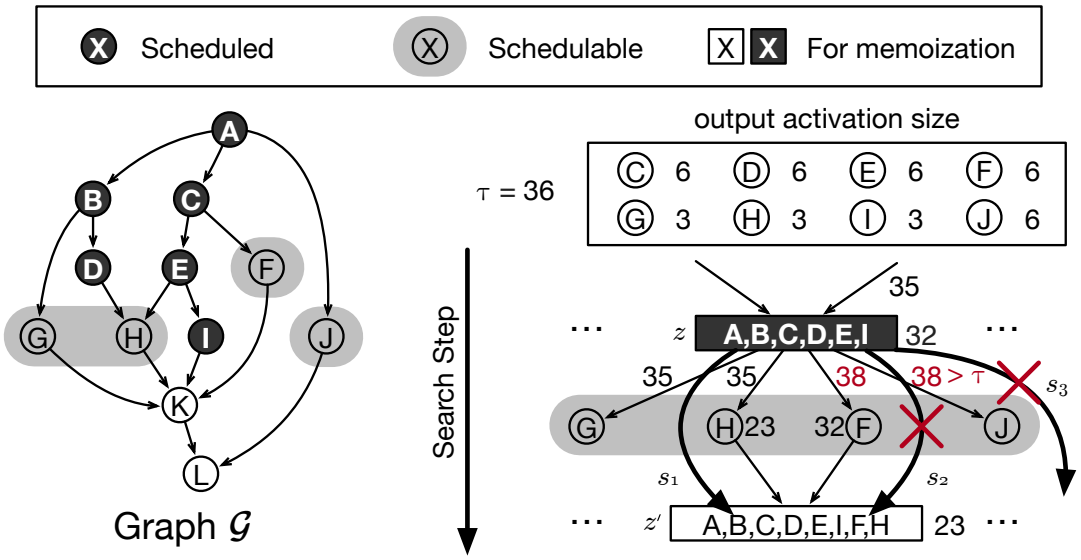
We can observe from Figure 3.1 that the topology of irregularly wired neural networks are *hourglass shaped* (⧖), because many NAS and Random Network Generators design *cells* with single input and single output then *stack them* to form an hourglass shape topology. [176] shows that, during general purpose code scheduling, graphs can be partitioned (*divide*) into multiple subgraphs and the corresponding solutions (*conquer*) can be concatenated (*combine*)

to form an optimal solution for the overall problem. While the complexity of the scheduling algorithm remains the same, this *divide-and-conquer* approach can reduce the number of nodes in each subproblem, speeding up the overall scheduling time. For instance, for a graph that can be partitioned into N equal subgraphs, the scheduling time will decrease from $|V| \times 2^{|V|}$ to $|V| \times 2^{|V|/N}$ that we can speed up scheduling by multiple orders of magnitude compared to the naive approach, depending on the size of the graph and the number of partitions.

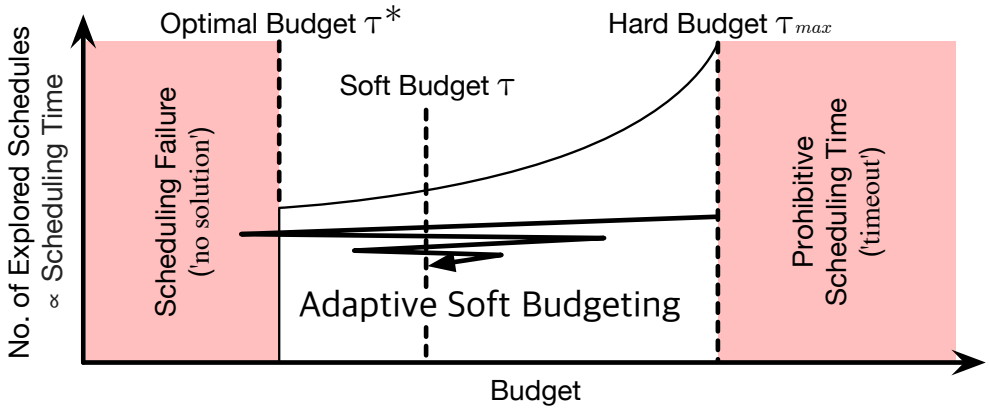
As such, Figure 3.7 shows this insight can be extended to our problem setting, where we can first perform scheduling on each cell and merge those solutions together to form the final solution. First, stage is partitioning the original graph $\mathcal{G}$ into multiple subgraphs g (*divide*). Then, utilizing the independence among the subgraphs, each subgraph g can be scheduled separately for their corresponding optimal schedule s_g (*conquer*). Considering that the number of nodes in the subgraph g is much smaller than the entire graph $\mathcal{G}$, the scheduling time will decrease significantly. Finally, the schedules of the subgraphs are concatenated to give optimal schedule s^* of the entire graph (*combine*).

Adaptive soft budgeting.

While divide-and-conquer approach scales down the number of nodes, the algorithm may still not be fast enough due to the exponential complexity of the algorithm. Therefore, we explore avoiding suboptimal solutions during the early stage of scheduling without affecting the optimality of the original algorithm. Since our goal is to find a *single* solution that can run within a given *memory budget* $\tau^* = \mu^*$ while all other solutions can be discarded, setting some budget τ that is greater or equal to μ^* and pruning suboptimal schedules with which their μ_{peak} exceeds τ can focus the search to a smaller search space $\mathcal{S}'_T \subset \mathcal{S}_T$ while still achieving the optimal schedule s^*. On top of this, we develop a *meta-search* for τ. This is inspired from engineers buying a larger memory (increase τ) if a program fails due to stack overflow (= 'no solution' due to an overly aggressive pruning) and selling out excess memory (decrease τ) if the current budget is prohibitive (= 'timeout' due to lack of pruning). **SERENITY** takes advantage of this insight to



(a) While both path s_1 and s_2 schedules lead to same z', their μ and μ_{peak} varies and we can prune schedules that yield higher μ_{peak} than a given budget τ. Numbers next to box or circle are μ and numbers next to edges are μ_{peak}



(b) *Adaptive soft budgeting* starts by setting a hard budget τ_{max} as the maximum value for the soft budget τ. Then, conducts a binary search for τ, higher than τ^* that it finds a solution yet not too high that scheduling completes quickly.

Figure 3.8. Illustration of the *adaptive soft budgeting*. (a) shows how schedules are pruned, and (b) illustrates how the *soft budget* τ relates to the number of explored schedules.

develop an *adaptive soft budgeting* scheme while scheduling to cut down the overall number of explored schedules. Figure 3.8 illustrates the overall idea by first showing how some schedules are pruned with regard to a given budget τ in Figure 3.8(a) then implication of different τ on scheduling time in Figure 3.8(b).

Algorithm 3. Adaptive Soft Budgeting

Input: graph $\mathcal{G}$
Output: optimal schedule s^*
$\tau_{max} \leftarrow \mu(\text{Kahn'sAlgorithm}(\mathcal{G}), \mathcal{G})$ // hard budget
$\tau_{old}, \tau_{new} \leftarrow \tau_{max}$
$flag \leftarrow$ 'no solution'
repeat
    // binary search for τ: decrease τ if 'timeout'
    // and increase τ if 'no solution'
    if $flag$ **is** 'timeout' **then**
        // simultaneous
        $\tau_{old} \leftarrow \tau_{new}$, $\tau_{new} \leftarrow \tau_{new}/2$
    else if $flag$ **is** 'no solution' **then**
        // simultaneous
        $\tau_{old} \leftarrow \tau_{new}$, $\tau_{new} \leftarrow (\tau_{new} + \tau_{old})/2$
    end if
    if $flag$ **is** 'solution' **then**
        $s^* \leftarrow schedule$ // optimal schedule
    end if
until $flag$ **is** 'solution'

Figure 3.8(a) depicts a certain point while scheduling $\mathcal{G}$, where nodes Ⓖ, Ⓗ, Ⓕ, and Ⓙ can be scheduled. In particular, the figure compares two possible solutions s_1 and s_2 which schedules Ⓗ→Ⓕ and Ⓕ→Ⓗ, respectively given $\tau = 36$. While s_1 and s_2 both starts from z with $\mu = 32$, scheduling Ⓗ leads to $\mu_{peak} = 32 + 3$ (H) $= 35$, whereas scheduling Ⓕ or Ⓙ leads to $\mu_{peak} = 32 + 6$ (F or J) $= 38$. Therefore, since we assume $\tau = 36$, s_2 and s_3 will fail because $\mu_{peak} = 38$ for s_2 and s_3 exceeds 36. So, as long as we set the budget τ higher than μ^*, the scheduler still finds a single optimal solution while avoiding many suboptimal paths. On the other hand, too small a $\tau < \mu^*$ leads to no solution because the optimal path would be pruned away.

Having established the possibility of pruning, our question boils down to discovering τ that is greater or equal to μ^* which we call an *optimal budget* τ^*, yet close enough to shrink the search space effectively. Figure 3.8(b) and Algorithm 3 summarizes the proposed *adaptive soft budgeting*. Since we start with no information about the approximate range for τ, we resort to a commonly used topological ordering algorithm called Kahn's algorithm [91] ($\mathcal{O}(|V|+|E|)$) to adaptively gain idea of the range for τ. We use the peak memory footprint from this sequence and use it as our *hard budget* τ_{max}, and in contrast we call adaptively changing τ as a *soft budget*. Since $\tau_{max} \geq \mu^*$, we know that any $\tau \geq \tau_{max}$ do not need to be explored. Having this upper bound for the search, *adaptive soft budgeting* implements a binary search to first run the scheduling algorithm with τ and T as input, where T is an hyperparameter that limits the scheduling time per search step. The binary search increases τ ($\tau_{new} \leftarrow (\tau_{new}+\tau_{old})/2$) if it finds 'no solution' and decreases τ ($\tau_{new} \leftarrow \tau_{new}/2$) if a search step returns 'timeout' (search step duration exceeds T). The binary search stops as soon as it finds a schedule ('solution'), and this method using binary search is guaranteed to work due to the monotonically increasing number of explored schedules with τ.

3.4.3 Identity Graph Rewriting: Improving the Search Space for Better Peak Memory Footprint

Reorganizing the computational graph of the irregularly wired neural networks may lead to *significant reduction in the peak memory footprint* μ_{peak} during computation. For example, it is notable that large stream of NAS-based works [38, 106] rely on extensive use of *concatenation* as a natural approach to merge information from multiple branches of the input activations and expand the search space of the neural architectures. However, concatenation with many incoming edges may prolong the liveness of the input activation and increase the memory pressure, which is unfavorable especially for resource constrained scenarios. To address this issue, we propose *identity graph rewriting* to effectively reduce μ_{peak} around the concatenation while keeping the arithmetic outputs identical. To this end, we present two main examples of the graph patterns in

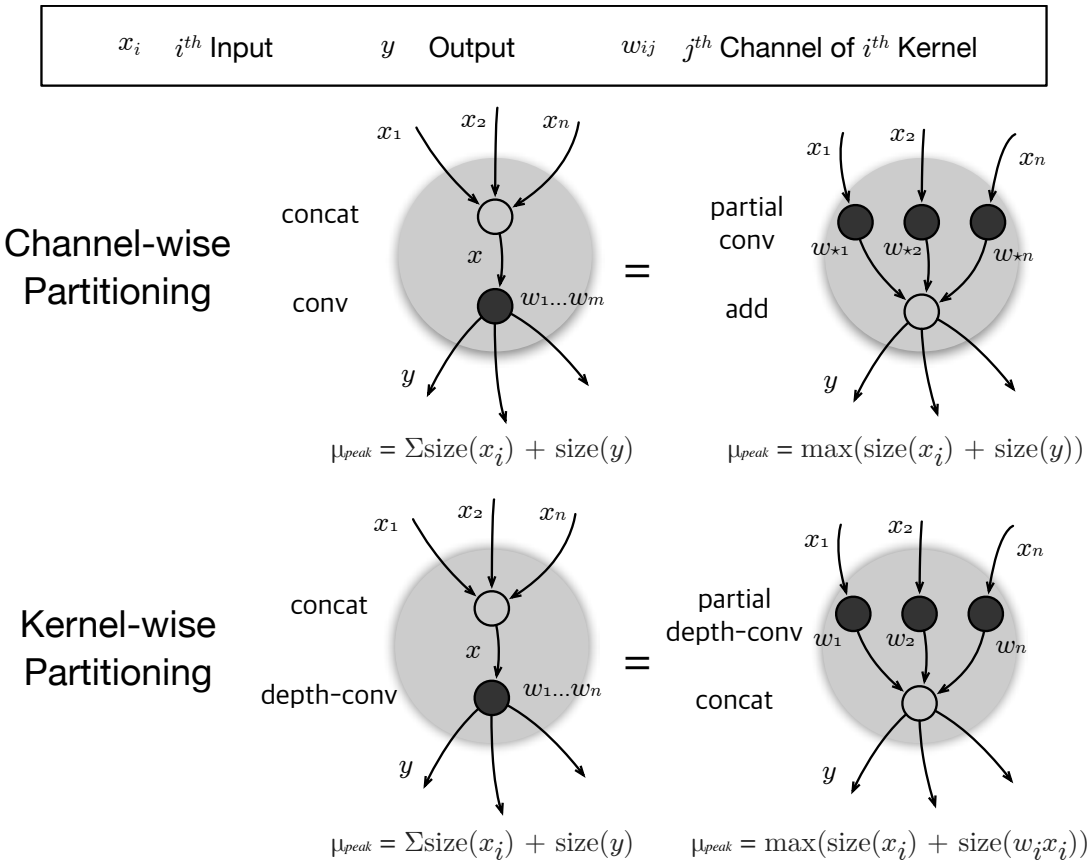


Figure 3.9. Illustration of the graph rewriting patterns: *channel-wise partitioning* and *kernel-wise partitioning* can reduce the memory cost of convolution and depthwise convolution respectively.

irregularly wired neural networks that benefits from our technique:

Channel-wise partitioning (convolution).

One typical pattern in irregularly wired neural networks is *concatenation* (*concat*: $[\cdot]$) that takes multiple branches of the input prior to a *convolution* (*conv*: $*$). While executing such pattern, peak memory footprint μ_{peak} occurs when the output $y \in \mathbb{R}^n$ is being computed while concatenated branches of input $x \in \mathbb{R}^n$ are also mandated to reside in the memory. Our objective is to achieve the same arithmetic results and logical effect as *concat* yet sidestep the corresponding seemingly excessive memory cost. To this end, we *channel-wise partition* the *conv* that follows the *concat* so that the *partitioned conv* can be computed as soon as the input x_i becomes available. Equation 3.3-3.6 detail the mathematical derivation of this substitution.

Specifically, as shown in Equation 3.3, each kernel iterates and sums up the result of convolving channels in conv. However, using the *distributive property* of $\sum_i$ and $*$, these transform to summation of *channel-wise partitioned convolution*, which we call *partial conv*. This *partial conv* removes *concat* from the graph leading to lower *memory cost*. As illustrated in Figure 3.9, the memory cost of same computation reduces from $\sum x_i + y$ to $max(w_{\star i} * x_i) + y$, which becomes more effective when there are more incoming edges to concat.

$$y = \Big[\sum_i w_{1i} * x_i, ..., \sum_i w_{mi} * x_i\Big] \quad \text{(concat+conv)} \tag{3.3}$$

$$= \sum_i \Big[w_{1i} * x_i, ..., w_{mi} * x_i\Big] \tag{3.4}$$

$$= \sum_i \Big[w_{1i}, ..., w_{mi}\Big] * x_i \tag{3.5}$$

$$= \sum_i \Big[w_{\star i} * x_i\Big] \qquad \text{(partial conv+add)} \tag{3.6}$$

Kernel-wise partitioning (depthwise convolution).

Depthwise convolution (*depthconv*) [81, 159] has been shown to be effective in reducing computation yet achieve competitive performance, hence its wide use in networks that target extreme efficiency as its primary goal. For *concatenation* (*concat*) followed by a *depthwise convolution* (*depthconv*), similar to above *concat+conv* case, peak memory footprint μ_{peak} occurs when the concatenated x is inside the memory and the result y additionally gets saved to the memory before x is deallocated. This time, we leverage the *independence* among different kernels to *kernel-wise partition* the *depthconv* that follows the *concat* so that each input x_i is computed to smaller feature maps without residing in the memory too long. As such, Equation 3.7-3.8 derives this substitution. Equation 3.7 shows that every component in the y is *independent* (different subscript index) and is viable for partitioning. In other words, this rewriting simply exposes the *commutative property* between *depthconv* and *concat* plus *kernel-wise partitioning* to reduce μ_{peak} significantly.

$$y = \Big[w_1 * x_1, ..., w_n * x_n\Big] \qquad \text{(concat+depthconv)} \tag{3.7}$$

$$= \Big[[w_1 * x_1], ..., [w_n * x_n]\Big] \quad \text{(partial depthconv+concat)} \tag{3.8}$$

Implementation.

Following the general practice of using pattern matching algorithms in compilers [87, 99, 143], we implement *identity graph rewriting* using pattern matching to identify regions of the graph which can be substituted to an operation with *lower computational cost*. Likewise, we make use of this technique to identify regions that leads to *lower memory cost*.

3.5 Evaluation

We evaluate **SERENITY** with four representative *irregularly wired neural networks* graphs. We first compare the peak memory footprint of **SERENITY** against TensorFlow Lite [69] while using the same linear memory allocation scheme[1] for both. Furthermore, we also experiment the impact of such peak memory footprint reduction on off-chip memory communication. We also conduct an in-depth analysis of the gains from the proposed *dynamic programming-based scheduler* and *graph rewriting* using SwiftNet Cell A [38]. Lastly, we study the impact of *adaptive soft budgeting* on the scheduling time.

3.5.1 Methodology

Benchmarks and datasets.

Table 3.1 lists the details of the networks–representative of the irregularly wired neural networks from Neural Architecture Search (NAS) and Random Network Generators (RAND)–used for evaluation: DARTS [106] for ImageNet, SwiftNet [38] for a dataset comprised of human presence or absence (HPD), and RandWire [181] for CIFAR10 and CIFAR100. DARTS [106] is

[1]TensorFlow Lite implements a linear memory allocator named simple memory arena: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/simple_memory_arena.cc

Table 3.1. Specification of the networks used for evaluation.

NETWORK	TYPE	DATASET	# MAC	# WEIGHT	TOP-1 ACCURACY
DARTS	NAS	IMAGENET	574.0M	4.7M	73.3%
SWIFTNET	NAS	HPD	57.4M	249.7K	95.1%
RANDWIRE	RAND	CIFAR10	111.0M	1.2M	93.6%
RANDWIRE	RAND	CIFAR100	160.0M	4.7M	74.5%

a gradient-based NAS algorithm. In particular we focus on the learned normal cell for image classification on ImageNet: only the first cell because it has the highest peak memory footprint and the reset of the network is just repeated stacking of the same cell following the practice in NASNet [196]. SwiftNet [38] is network from NAS by targeting human detection dataset. RandWire [181] are from Random Network Generators for image classification on CIFAR10 and CIFAR100. The table also lists their dataset, multiply-accumulate count (# MAC), number of parameters (# WEIGHT), and top-1 accuracy on their respective dataset.

3.5.2 Experimental Results

Comparison with TensorFlow Lite.

Figure 3.10 evaluates **SERENITY** over TensorFlow Lite on different cells of the aforementioned networks in terms of reduction in memory footprint. The figures illustrate that **SERENITY**'s dynamic programming-based scheduler reduces the memory footprint by a factor of 1.68×without any changes to the graph. In addition, the proposed graph rewriting technique yields an average of 1.86×(extra 10.7%) reduction in terms of peak memory footprint. The results suggest that **SERENITY** yields significant reduction in terms of the peak memory footprint for irregularly wired neural networks.

Improvement in off-chip memory communication.

We also show how **SERENITY** affects the off-chip memory communication, which largely affects both power and inference speed [36, 60, 157]. To this end, Figure 3.11 sweeps different

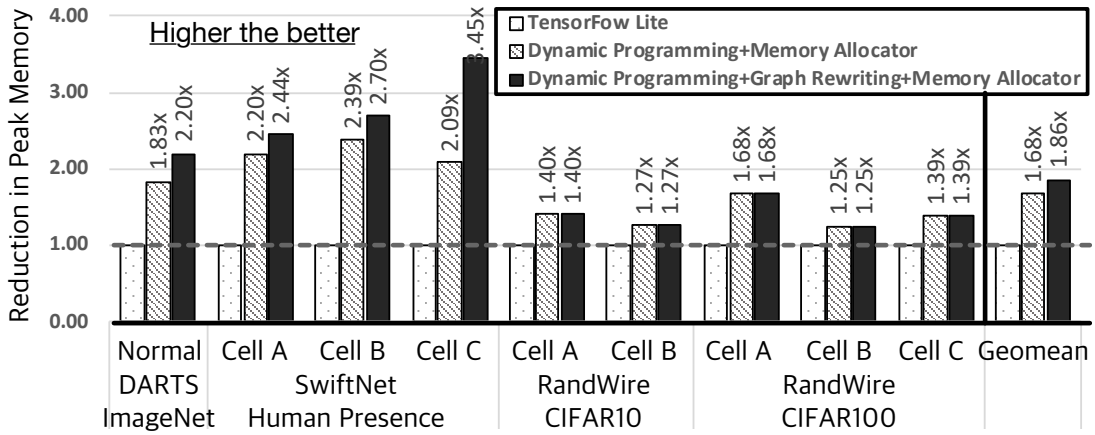


Figure 3.10. Reduction in peak memory footprint of **SERENITY** against TensorFlow Lite (no memory hierarchy).

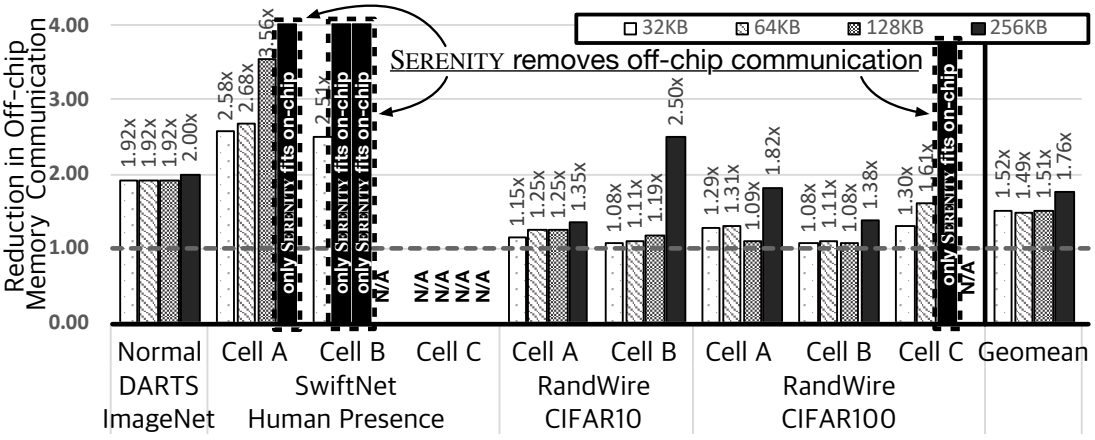
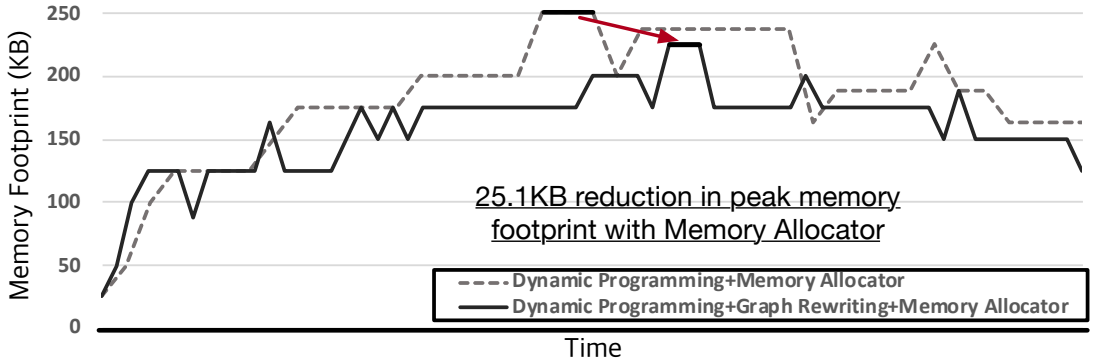


Figure 3.11. Reduction in off-chip memory communication of **SERENITY** against TensorFlow Lite (with memory hierarchy).

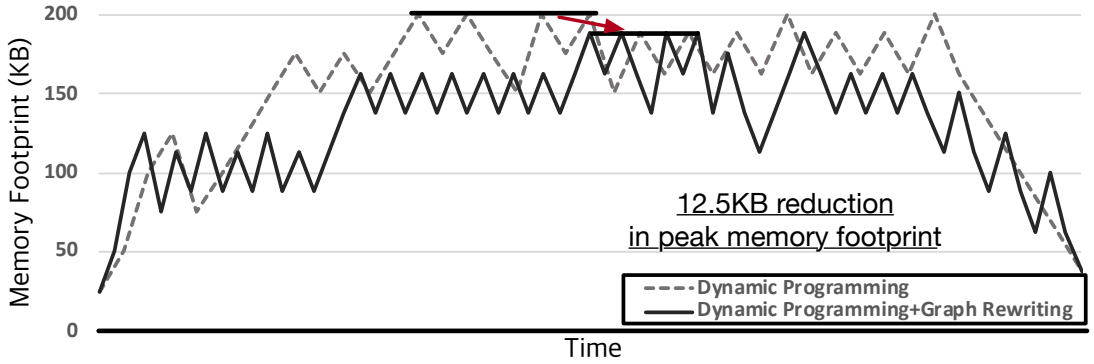
on-chip memory configurations to measure the reduction in off-chip communication on systems with multi-level memory hierarchy. Since we know the entire schedule *a priori*, we use Belady's optimal algorithm [20], also referred to as the clairvoyant algorithm for measuring the off-chip memory communication, to distill the effects of the proposed scheduling. The results show that **SERENITY** can reduce the off-chip memory communication by 1.76× for a device with 256KB on-chip memory. In particular, while there were few cases where peak memory footprint was already small enough to fit on-chip (N/A in figure), there were some cases where **SERENITY** eradicated the off-chip communication by successfully containing the activations in the on-chip memory while TensorFlow Lite failed to do so (marked in figure). This suggests that **SERENITY**'s effort of reducing memory footprint is also effective in reducing the off-chip memory communication in systems with memory hierarchy, hence the power consumption and inference speed.

Improvement from dynamic programming-based scheduler and identity graph rewriting.

To demonstrate where the improvement comes from, Figure 3.12 plots the memory footprint while running Swiftnet Cell A. Figure 3.12(a) shows the memory footprint of **SERENITY** with the memory allocation. The figure shows that **SERENITY**'s dynamic programming-based scheduler brings significant improvement to the peak memory footprint (551.0KB→250.9KB), and the graph rewriting further improves this by 25.1KB (250.9KB→225.8KB) by utilizing patterns that alleviate regions with large memory footprint. In order to focus on the effect of the scheduler and graph rewriting, Figure 3.12(b) presents the memory footprint of **SERENITY** without the memory allocation: the sum of the activations while running the network. The figure shows that the proposed scheduler finds a schedule with the optimal (minimum) peak memory footprint without changes to the graph. Then, it shows that the proposed graph rewriting can further reduce the peak memory footprint by 12.5KB (200.7KB→188.2KB). The results suggest that the significant portion of the improvement comes from the proposed dynamic programming-based scheduler and the graph rewriting.



(a) Memory footprint with the memory allocator (peak memory footprint of TensorFlow Lite = 551.0KB).



(b) Memory footprint without the memory allocator.

Figure 3.12. Memory footprint while running SwiftNet Cell A with and without the memory allocator (*red arrow* denotes reduction).

Scheduling time of SERENITY.

Figure 3.13 summarizes the (static) scheduling time taken for **SERENITY** to schedule the networks. Results show that the average scheduling time is 40.6 secs without the graph rewriting and 48.8 secs with graph rewriting, which the difference comes from the increase in the number of nodes from graph rewriting. The results show that all the above gains of **SERENITY** come at the cost of less than one minute average extra compilation time. While the dynamic programming-based scheduling suffers from an exponential time complexity, **SERENITY** manages to make the scheduling tractable through the proposed divide-and-conquer and adaptive soft budgeting.

Speed up from divide-and-conquer and adaptive soft budgeting.

Table 3.2 summarizes the scheduling time of SwiftNet [38] for different algorithms to demonstrate the speed up from divide-and-conquer and adaptive soft budgeting techniques. As such, the table lists different combination of algorithms, number of nodes, and the corresponding scheduling time. Straightforward implementation of the aforementioned ❶ *dynamic programming-based scheduling* leads to an immeasurably large scheduling time regardless of the *graph rewriting*. However, additional application of the ❷ *divide-and-conquer* (❶+❷) leads to a measurable scheduling time: 56.53 secs and 7.29 hours to schedule without and with the graph rewriting, respectively. Furthermore, we observe that further applying ❸ *adaptive soft budgeting* (❶+❷+❸) significantly reduces the scheduling time 37.9 secs and 111.9 secs to schedule without and with the graph rewriting, respectively. Above results indicate that applying the proposed algorithms leads to a scheduling time of practical utility.[2]

[2]Initial implementation presented in [4] was developed using Python which took around a minute to schedule. However, an alternative implementation in C++ resulted in a significantly faster scheduling in the order of seconds, or even sub-second.

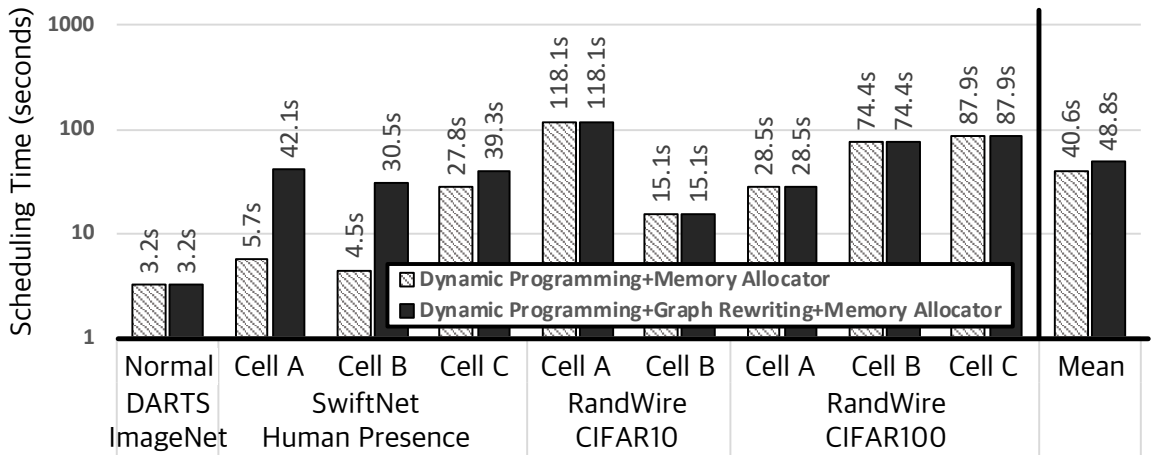


Figure 3.13. Scheduling time evaluation for **SERENITY**.

Table 3.2. Comparison of the scheduling time for different algorithms to schedule SwiftNet. ❶, ❷, and ❸ represent *dynamic programming*, *divide-and-conquer*, and *adaptive soft budgeting* respectively. N/A denotes infeasible within practical time.

GRAPH REWRITING	ALGORITHM	# NODES AND PARTITIONS	SCHEDULING TIME
✗	❶	62 ={62}	N/A
✗	❶+❷	62={21,19,22}	56.5 secs
✗	❶+❷+❸	62={21,19,22}	**37.9 secs**
✓	❶	92={92}	N/A
✓	❶+❷	92={33,28,29}	7.2 hours
✓	❶+❷+❸	92={33,28,29}	**111.9 secs**

3.6 Related Works

The prevalence of neural networks has led to the development of several compilation frameworks for deep learning [1, 45, 135, 143]. However, even industry grade tools, mostly focus on tiling and fine-grained scheduling of micro-operations on the conventional hardware [69, 129] or accelerators [36, 37, 56, 60, 72, 88, 89, 134, 157]. However, these framework are mostly designed for the common regular patterns that have dominated deep learning from almost its conception. As such, these tools inherently had no incentive to deal with the form of irregularities

that the emerging NAS [29, 38, 43, 106, 140, 195, 196] and Random Networks [178, 181] bring about. This paper, in contrast, focuses on this emergent class that breaks the regularity convention and aims to enable their execution on memory constrained edge devices.

Scheduling and tiling for neural networks.

While prior works on scheduling [93, 102, 176] focus on classical computing workloads, there have been limited study about the implications of scheduling in the neural networks domain. There is also a significant body of work on scheduling operations on hardware accelerators [2] that also considers tiling [5, 34, 107, 170]. However, graph scheduling for irregularly wired neural network, specially with memory constraints, is an emerging problem, which is the focus of this paper.

Graph rewriting for neural networks.

It has been a common practice to rewrite parts of the graph using rule-based [1, 45, 129, 135, 143] or systematic approaches to expose parallelism and make models more target-aware [86, 87, 153]. While these approaches may alleviate the complexity of the graph and reduce the peak memory footprint as a side effect, these frameworks do not explore and are not concerned with scheduling. Our work exclusively explores graph rewriting in the context of improving the peak memory footprint.

Optimizing neural networks.

There are different optimization techniques that aim to simplify the neural network indifferent dimensions. Sparsification/compression [17, 74, 101, 194], quantization [44, 52, 73, 121, 193], activation compression [84], and kernel modifications reduce the complexity of the individual operations or remove certain computations. However, our focus, the problem of memory-aware graph scheduling still remains orthogonal to these inspiring efforts.

3.7 Conclusion

As the new forms of connectivity emerges in neural networks, there is a need for system support to enable their effective use, specially for intelligence at the edge. This paper took an initial step toward orchestrating such network under stringent physical memory capacity constraints. We devised signatures to enable dynamic programming and adaptive soft budgeting to make the optimization tractable. Even more, an identity graph writing was developed to further the potential for gains. The encouraging results for a set of emergent networks suggest that there is significant potential for compiler techniques that enables new forms of intelligent workloads.

3.8 Future Directions

Improving scalability of memory-aware scheduling.

While **SERENITY** [4] focused on the scheduling (or sequencing) for irregularly wired neural networks with up to 100 nodes, neural networks may have orders of magnitude larger number of nodes in its computational graphs. In fact, if we consider a tiled version of the original graph, the graph can grow to even larger [59]. In such case, the proposed dynamic programming-based solution, despite the heuristics, may take eons to find optimal solutions. [59] and [16] have explored the combination of graph neural network with attention and GFlowNet [23], respectively, to improve the scalability of scheduling computational graphs. Further investigations into this direction to achieve near-optimal solutions for scheduling large-scale computational graphs can bring significant benefits.

Extending insights to memory placement.

Another important optimization problem in neural execution is the memory placement [94]. As the computational graphs of the DNNs are static, memory placement can be determined at compile time over online cache management algorithms [158, 190] that offer sub-optimal performance. While some recent works that leverage reinforcement learning [120], domain knowledge [119], and graph neural networks [94], its performance are far from the

optimal performance that is offered by dynamic programming [22] or constraint programming [113, 142]. As the use of the algorithms that offer optimal solutions are often limited due to its large overhead, the experiences from [4] to reduce its scheduling time can provide opportunities to make the optimal approaches more pragmatic.

Memory management of irregularly wired neural networks.

While **SERENITY** [4] focused on the scheduling (or sequencing) for irregularly wired neural networks, there is another aspect in memory management: memory allocation. Even if we manage to find optimal schedules of the neural network, lack of intelligent memory allocation algorithm may cause *memory fragmentation* that may lead to a bloated memory usage. At the time of publication, TensorFlow Lite [69] implemented a linear memory allocation named simple memory arena. However, an effort [155] from IBM Research explored a mixed-integer programming to find optimal memory allocation for small graphs and a profile-guided optimization to make this more scalable. Furthermore, an effort [137] from the Google TensorFlow Lite team explored various heuristics to improve memory allocation. While both directions made significant strides in improving memory allocation, their target is still limited to relatively simple models and sometimes fails to achieve optimal memory allocation for large-scale irregularly wired neural networks. Developing practical solutions that can achieve optimal memory allocation with negligible compilation overhead would provide significant benefits.

Acknowledgement. Chapter 3, in part, contains a re-organized reprint of the material as it appears in Conference on Machine Learning and Systems (MLSys) 2020. Ahn, Byung Hoon; Lee, Jinwon; Lin, Jamie Menjay; Cheng, Hsin-Pai; Hou, Jilei; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper[3].

[3]Qualcomm Technologies, Inc. ("QTI") grants Byung Hoon Ahn ("Licensee") a limited, revocable, non-transferable, non-exclusive, royalty and fee free copyright license to copy and reproduce, in whole or in part, the paper entitled "Ordering Chaos: Memory-Aware Scheduling of Irregularly Wired Neural Networks for Edge Devices" published in MLSys 2020 along with any supplemental material provided with the paper ("Content") as part of Licensee's Ph.D. thesis submission, provided that the Content is cited as belonging to Qualcomm Technologies, Inc., has the appropriate copyright notice as a footnote and is only disclosed to Licensee's PH.D. school Director and to the academic reviewers of the thesis. Any further disclosure of the Content will require additional permission or license. The Content remains the exclusive property of QTI and no other rights are granted.

Chapter 4

Hybridization of AI and Foundational Algorithms for Optimized Execution of AI

Section 2 is a case study of *AI Algorithm* improving the adaptiveness of deep learning compiler, and Section 3 is a case study of *Foundation Algorithm* achieving optimal solution. Having observed that both worlds–AI and Foundational Algorithms–can bring significant benefits to compiler optimization, this section presents an ambitious effort to take advantage of the *Best of Both Worlds*. To this end, this section explores a *Hybridization of AI and Foundational Algorithms* for optimized execution of AI. In more detail, this section utilizes *Principal Component Analysis (PCA)*, a widely used dimensionality reduction technique, to embed hardware specifications into a hardware representation. Then, this section explores a combination of *Meta Learning* with *Hypernetworks* to improve neural compilation.

4.1 Mathematical Embedding of Hardware Specification for Neural Compilation

Success of Deep Neural Networks (DNNs) and their computational intensity has heralded Cambrian explosion of DNN hardware. While hardware design has advanced significantly, optimizing the code for them is still an open challenge. Recent research has moved past traditional compilation techniques and taken a stochastic search algorithmic path that *blindly* generates rather stochastic samples of the binaries for real hardware measurements to guide the search.

This paper opens a new dimension by incorporating the mathematical embedding of the hardware specification of the GPU accelerators dubbed **Blueprint** to better guide the search algorithm and focus on sub-spaces that have higher potential for yielding higher performance binaries. While various sample efficient yet blind hardware-agnostic techniques have been proposed, none of the state-of-the-art compilers have considered hardware specification as hints to improve the sample efficiency and the search. To mathematically embed the hardware specifications into the search, we devise a Bayesian optimization framework called **Glimpse** with multiple exclusively unique components. We first use the **Blueprint** as an input to generate prior distributions of different dimensions in the search space. Then, we devise a light-weight neural acquisition function that takes into account the **Blueprint** to conform to the hardware specification while balancing the exploration-exploitation trade-off. Finally, we generate an ensemble of predictors from the **Blueprint** that collectively vote to reject invalid binary samples. We compare **Glimpse** with hardware-agnostic compilers. Comparison to AutoTVM [35], Chameleon [5], and DGP [165] with multiple generations of GPUs shows that **Glimpse** provides 6.73×, 1.51×, and 1.92× faster compilation time, respectively, while also achieving the best inference latency.

4.2 Introduction

Prevalent adoption of Deep Neural Networks (DNNs) in voice assistants, smart speakers, and enterprise applications has triggered a Cambrian explosion of DNN hardware to cope with the colossal computational intensity of DNNs. While the hardware designs have advanced significantly, inseparable task of generating optimized code for them is still an open challenge. In fact, hand-optimized libraries such as NVIDIA cuDNN or Intel MKL that serve backend for programming interfaces such as TensorFlow [1] and PyTorch [135] have been the go-to solutions for higher performance DNN execution. However, recent research in neural compilers has taken a leap beyond hand-optimized libraries and traditional compilation techniques, and embraced stochastic search algorithms such as simulated annealing to improve the search. These search

algorithms navigate an exponentially large search space for the optimized code, which is one of the main reason behind the success of optimizing compilers [34]. To traverse the search space in a sample efficient manner, recent innovations in optimizing compilers strived to reduce the compilation time with cost models to approximate the large search space [35, 145] and effective search algorithms [5, 165]. However, these search algorithms [5, 35, 103, 145, 165, 191], classified as *black-box optimization*, are *blindly* and solely guided by the real hardware measurements. These measurements, however, comes at a large cost in terms of time yet barely provides any *architectural hints* to effectively guide the search algorithms due to their *blindness*. As such, although these neural compilers have made their way into the deep learning pipelines of major deep learning solutions providers including Amazon, Xilinx, and Qualcomm, the current paradigm of *hardware-agnostic* neural compilers takes hours to optimize even a small model. In fact, this even grows to *days on GPUs* to optimize multitude of models on many GPU accelerators[1], which curtails the overall productivity in DNN model deployment.

This paper sets out to explore a new path where we provide neural compilers with *perception* such that it can take a *glimpse* of the *mathematical embedding* of the *hardware blueprints* to better guide the search algorithm. We devise a Bayesian optimization framework called **Glimpse** that uniquely explores the mathematical embedding of the GPU specifications dubbed **Blueprints** to expedite the neural compilation while also improving the resulting binary performance. We first use **Blueprints** to generate a set of prior distributions of different dimensions of the search space. Then, we devise a light-weight neural acquisition function learned using *meta-learning-based algorithm* that takes into account the **Blueprint** to conform to the hardware while balancing the exploration-exploitation trade-off. Finally, we generate an ensemble of predictors from the **Blueprint** that collectively vote to reject invalid binary samples. We compare **Glimpse** with state-of-the-art *hardware-agnostic* neural compilers AutoTVM [35], Chameleon [5], and DGP [165] with modern DNNs including AlexNet [96], ResNet-18 [75],

[1]For example, 10 DNN models on 100 different GPUs would take around 10,000 GPU hours to optimize which translates to $9,000 with Amazon EC2 instances (on-demand, p2.xlarge). This is an exorbitant (per model update) cost for businesses considering the swift evolution of the neural architectures deployed in real world applications.

VGG-16 [161] on multiple generations of GPUs including Titan Xp, RTX 2070 Super, RTX 2080 Ti, RTX 3090. Integration of **Glimpse** to TVM [34] shows that **Glimpse** provides 6.73×, 1.51×, and 1.92× faster compilation time over AutoTVM, Chameleon, and DGP, respectively, while also achieving the best inference latency. Further analysis show up to 2.18× improvement in the initial configurations over transfer learning, 5.07× and 2.55× reduction in the number of search steps compared to AutoTVM and Chameleon. **Glimpse** also reduces invalid configurations by 5.56× and 4.53× over AutoTVM and Chameleon.

4.3 Challenges in Neural Compilation

After the models are trained using programming interfaces such as TensorFlow [1] or PyTorch [135], they are sent to the deployment engineers whose goal is to make sure the models meet various Quality-of-Service (QoS) requirements such as inference latency in end-to-end applications. The deployment engineers utilize optimizing compilers such as TVM [34] to tune the performance on a given target hardware, we use the term *Neural Compilers* throughout the paper. In fact, major deep learning solution providers such as Amazon, Xilinx, and Qualcomm incorporate these neural compilers within their Software Development Kit (SDK).

4.3.1 Neural Compilation for Model Deployment

Current neural compilers generally try to optimize $s \in S$ while considering the target hardware as a *black-box* function $f(x_s)$, where x and s are the code templates (e.g., Conv2D, Dense, and etc.) and their configuration (sampled from combinations of tiling, bindings, unrolling, and etc.), respectively. Usually the size of the overall search spaces S is astronomically large, which render simple grid search algorithms impractical. For example, the first layer of VGG-16 has over 200 million combinations. To make this worse, these search spaces are *not differentiable*, and the optimal configurations are sparsely distributed throughout the search space making it a complex problem to solve. Recent advances in neural compilation [5, 35, 103, 145, 165, 191] have introduced a cost model $\hat{f} \approx f$ that approximates the vast search space and proposed

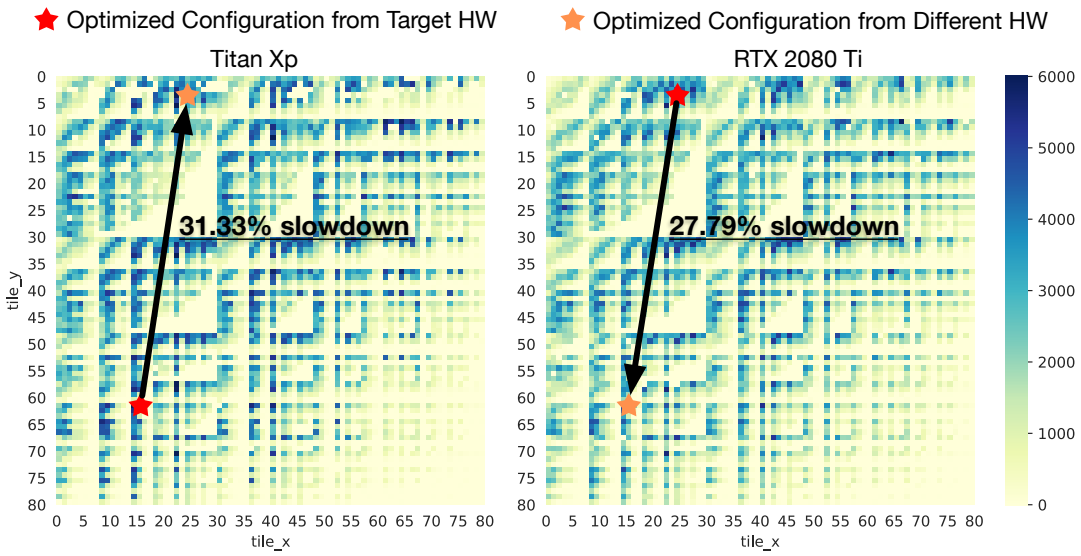


Figure 4.1. Visualization of ResNet-18 7$^{\text{th}}$ layer's search space on different generation of GPUs (Titan Xp vs. RTX 2080 Ti). While the overall search space may look similar, the optimal configuration is different. We cannot just reuse the optimal binary from one hardware to run DNN on another hardware.

intelligent search algorithms that better navigates the search space. However, the neural compilers still suffer from long compilation times of over tens of hours to days for even a single neural network.

4.3.2 Challenges and Opportunities in Neural Compilation

Although the problem of neural compilation as stated in Section 4.3.1 is already difficult, current neural compilation formulation has a narrow focus on a *single* hardware. However, in reality, there are *multiple* generations of hardware that are embedded in the intelligent devices. For instance, if we consider GPUs that are widely used to execute DNNs, generations of the GPU (e.g., Pascal, Turing, Ampere, etc.) vary machine by machine. To this end, the deployment engineers are left with a formidable task of tuning the DNN model for *multiple* not *single* target hardware, meaning n repetitions of the overall neural compilation for n hardware. In other words, considering $\theta \in \Theta$ (where Θ encodes the hardware configurations such as number of different

cores, clocks, bandwidth, bus types, and etc.), problem formulation must be updated to:

$$s^* = \underset{s}{\operatorname{argmax}}\, f(x_s|\Theta_k), \qquad \text{for } s \in S \text{ \& many } k \in \mathbb{N} \tag{4.1}$$

Simplest approach to cope with the variations in hardware is to just *ignore and reuse* the optimized configuration from another hardware. For example, using s^* from Titan Xp to compile DNN on RTX 2080 Ti. However, this may not result in the optimized performance we desire. In fact, Figure 4.1 shows that while the overall search space takes a similar shape for different hardware, the optimal configuration differs among them. For ResNet-18 7^{th} layer, reusing s^* led to 27.79% slowdown of the output code for Titan Xp$\rightarrow$RTX 2080 Ti, and 31.33% for RTX 2080 Ti$\rightarrow$Titan Xp. On the other hand, *transfer learning* [35] is the most common way of reusing the *compilation experiences*. However, this also suffers from similar degradation in the performance of the resulting binary. An alternative approach would be to develop multiple neural compilers, one for each hardware, but this is neither cost-effective nor scalable solution to the long neural compilation time problem. Most importantly, such approach cannot cope with the constant evolution of the hardware. Simply put, *current hardware-agnostic techniques are not scalable*. On the other end of the spectrum, some analytical model or a simulator within the neural compilation loop to give *full view* of the hardware to run a *white-box optimization*, the confidentiality of the hardware design and the potential slow down of the compilation process from the complex hardware prohibits this.

However, silver lining here is that: (i) while the precise blueprints of the hardware are difficult if not impossible to get and use in neural compilation, *some features or the specification of the hardware are available in public data sheets* [130], and (ii) despite the fact that optimal solutions are different for different hardware, *their search spaces have similar characteristics that open up opportunities to transfer the optimization experiences*. Overall, the macro view of the problem of *neural compilation for multiple hardware* makes the problem more challenging, yet introduces a new unexplored dimension in designing neural compilers: *hardware-awareness*.

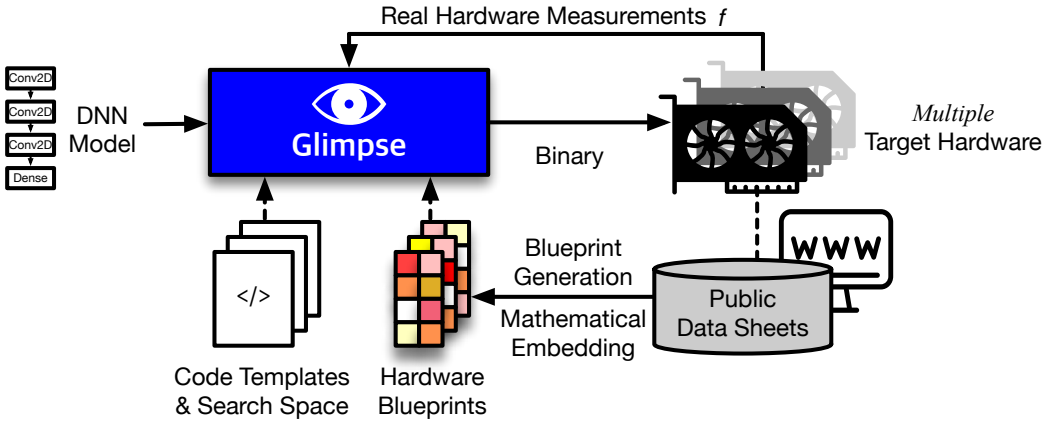


Figure 4.2. Overview of compilation with **Glimpse**. Unlike current *hardware-agnostic* approaches which navigate the search space *blindfolded*, **Glimpse** takes hints from *glimpse* of hardware **Blueprints** for faster neural compilation.

4.4 Glimpse: Mathematical Embedding of Hardware Specification for Faster Neural Compilation

Deviating from the current *blind* and *hardware-agnostic* neural compilers, we propose **Glimpse**, a novel neural compiler with *perception* to take a sneak peek of the hardware specifications in the form of mathematical embedding dubbed **Blueprint**. We first devise a mathematical embedding **Blueprint** to encapsulate the hardware specifications. Then, we develop a *hardware-aware* neural compiler dubbed **Glimpse** that takes the **Blueprints** to take a glimpse of the hardware blueprint to adaptively and quickly optimize the input DNNs to the target hardware. To this end, this work can be subdivided into two main components that work together: (i) **Blueprint** a mathematical embedding that encodes key specifications of the hardware, and (ii) **Glimpse** that translates the embedding into useful knowledge such as prior distributions to guide the search, search strategy in the form of neural acquisition function that can expedite the optimizing compilation, and ensemble of predictors to reject the invalid configurations. Figure 4.2 illustrates the overall flow of the compilation with **Glimpse** and **Blueprint**.

4.4.1 **Blueprint**: Mathematically Embedding Architectural Features of Hardware

To provide *hardware-awareness* to the neural compiler, we need to feed the neural compiler with the specification about the target hardware. However, unlike with white-box optimization where we would have the full view of the design and the specification of the hardware enabling explicit description of the hardware within the neural compiler, the complexity of the hardware designs as well as the confidentiality of the designs make it hard if not impossible to get the design. To close the structural gap between the demand for faster DNN deployment hence faster neural compilation and the practical difficulty in incorporating hardware information, ***Glimpse*** *utilizes the architectural specifications provided by the vendors in public data sheets* [130]. *The data sheet lists the number of different processors/cores, bus interfaces, cache size, clock cycles, and the compute capacity in GFLOPS provided by the manufacturer. We create a mathematical embedding of these specifications*. These mathematical embeddings can provide neural compilers with a sneak peek of the architecture, and as a result provide hints about the search space and assist compiler while learning to quickly optimize tensor programs to better optimality.

Design.

We devise a novel abstraction of the hardware dubbed ***Blueprint*** *which is a mathematical embedding vector that summarizes the important features of the target hardware.* Two key considerations while developing the **Blueprint** are (i) minimizing the loss of information while (ii) maintaining low overhead. While (i) is an obvious objective, (ii) is one of the key subtleties. As suggested in Section 4.3, one of the key challenges we face in developing neural compilers is the eons of time required for optimization. Therefore, one of the key design consideration was reducing the size of the embeddings that can impact the compilation time. In fact, parsing overhead for the neural compilers to gain architectural insights from the **Blueprint** may accrue to constitute a significant fraction of the neural compilation time. We perform a *dimensionality*

reduction of the original feature vectors using Principal Component Analysis (PCA) to get get the minimal mathematical embedding vector that summarizes the hardware. We use PCA over neural autoencoders as PCA provides an intuitive knob that allows us to balance the size with the information loss. On the other hand, using neural autoencoders would require more complex design space exploration of the neural model. Also, neural networks required more computation to achieve the same dimensionality reduction.

Prior distribution generation from Blueprint.

We consider the neural compilation as a Bayesian optimization problem where the optimization begins with a prior distribution and updates the distribution over multiple iterations to gradually improve the posterior distribution, improving the quality of sampled binaries as we progress through the compilation. While this prior distribution can be learned from scratch, this has been shown to be very inefficient [5, 35]. As such, *we use the aforementioned hardware **Blueprint** and the network specification to generate the prior distribution that can speed up the compilation significantly*. We use a parametric neural model $f'_k(\pi) \approx f_k$ instead of non-parametric Gaussian processes to approximate the spaces. Then, taking inspiration from HyperNetworks [71], we devise a prior distribution generator $\mathcal{H}$ that takes a layer specification and **Blueprint** as input and outputs the parameters π for the prior distribution $f'_k(\pi)$. To train $\mathcal{H}$, we gathered a large scale dataset similar to [192] of s and f. One important design choice for $\mathcal{H}$ was generating n distributions for n dimensions of the search space. $\mathcal{H}$ generates $f_{k,\text{tile\_x}}$ and $f_{k,\text{tile\_y}}$ for the dimensions tile_x and tile_y, respectively. To get the initial samples from the search space, **Glimpse** enumerates combinations of the $\text{argmax}(f_{k,*})$, weighted by the $\Pi f_{k,*}$. Overall, this prior distribution generator $\mathcal{H}$ serves an effective initialization for the optimizing compilation procedure, reducing the number of costly hardware measurements to locate optimal configuration s^*. Importantly, as prior distribution generation from **Blueprint** is a one-off process per layer, the computational cost of $\mathcal{H}$ was negligible.

4.4.2 Hardware-Aware Exploration: Adapting Optimization Steps with Meta-learning

Current *hardware-agnostic* techniques [5, 35, 103, 145, 165, 191] take *black-box* approach and utilize stochastic optimization algorithms. To transfer the experience among different compilation instances, above method such as AutoTVM [35] uses the cost model as a proxy to transfer knowledge among similar layers. While these approaches allow the users to reuse the cost model, they still require significant number of real hardware measurements before they start yielding satisfactory output code. Likewise, reusing cost models among different hardware usually yield sub-optimal output code as stated in Section 4.3.2. The main reason for such sub-optimal performance is because the subtle differences in the architecture leads to significant, yet nonlinear, changes in the performance for the target hardware. *Unlike these naive approaches to transfer experiences,* ***Glimpse*** *leverages the information encapsulated in* ***Blueprints*** *to improve the hardware-awareness of the exploration process.* The main insight is that, while the exact locations of the optimal configuration in the search spaces may be different among multiple hardware, the know-hows on how to achieve that optimal configuration may be transferable. ***Glimpse*** *incorporates a hardware-aware strategy to conduct the search. In particular, we take inspiration from MetaBO [172] to learn the Meta-Optimizer in the Figure 4.3 to emit neural acquisition functions $f(\cdot|\theta)$ for Hardware-Aware Exploration that dictates the exploration and exploitation strategy.*

Training.

Training first begins by sampling the maximums X_s from the prior distribution from Section 4.4.1. Then, we follow the natural Bayesian optimization pass of (i) sampling initial solutions from the surrogate cost model f, (ii) *Hardware-Aware Exploration* to determine the configurations X_s to explore, and (iii) *Hardware-Aware Sampling* to prune invalid configurations to determine the candidates for real measurements X'_s. Measurements f (*reward*), Tuples of configuration and the optimization budget (X_s, t, T) (*state*) where t and T are the optimization

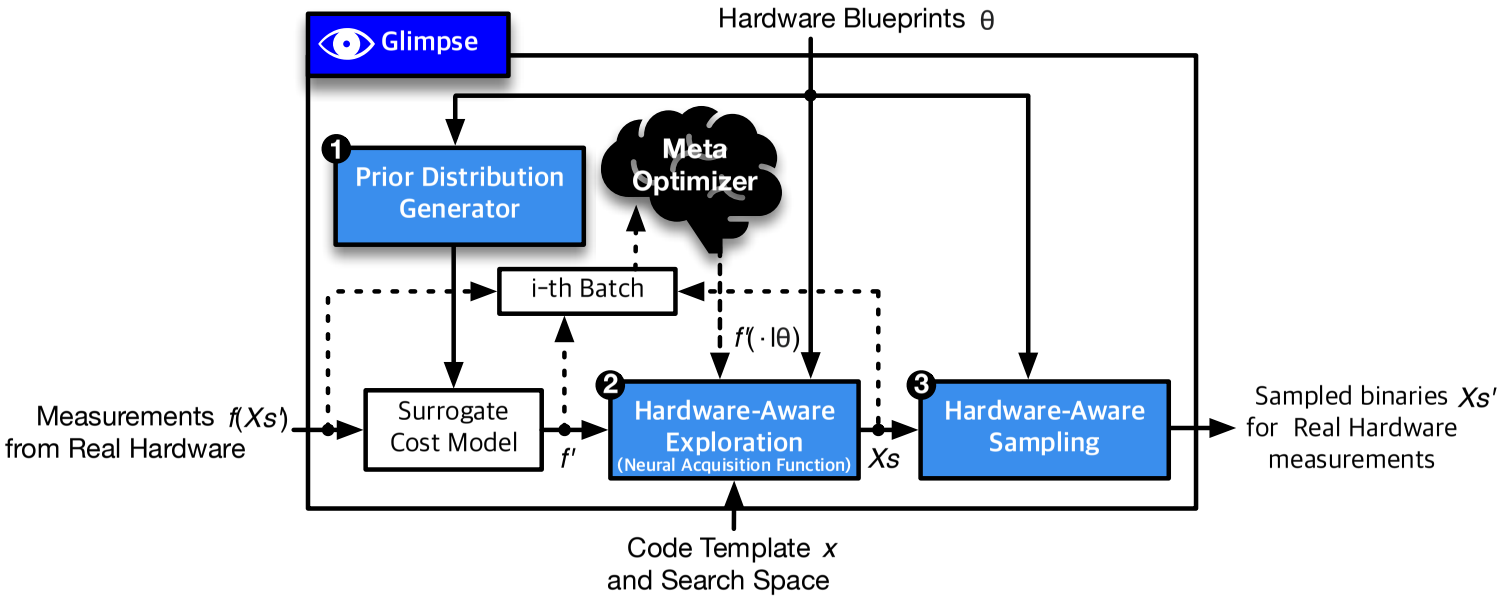


Figure 4.3. Detailed diagram of **Glimpse** and its components. Dotted arrows are *offline* training procedure.

step and the budget, respectively, and the optimal configuration $x_s \in X_s$ (*action*) are collected as the dataset to train the *Meta-Optimizer*. Highlighted inside the brackets translates the **Glimpse** training setting into the reinforcement learning parlance, similar to the [172]. We iterate through various hardware and networks to train our *Meta-Optimizer*. *As we progress through the Meta-Optimizer training, the Hardware-Aware Exploration that gets emitted gradually improves and learns to (i) make the optimal trade-off between exploration-exploitation and, more importantly, (ii) learn how to incorporate the hardware-awareness in the Hardware-Aware Exploration module. Final outcome of this off-line process is the hardware-aware optimization strategy ingrained in the Hardware-Aware Exploration module*.

4.4.3 Hardware-Aware Sampling: Using Statistics to Minimize Invalid Configurations

Besides the above innovations, **Glimpse** tackles an innate issue in neural compilers: frequent invalid configurations. Chameleon [5] suggested using clustering that samples the centroids to reject invalid configurations. However, clustering-based sampling is *hardware-agnostic*, and it fails to filter out many of the invalid configurations, leading to significant waste in GPU time and low (real measurements) sample efficiency. In contrast, **Glimpse** incorporates the *hardware-guided* approach to reject invalid configurations. ***Glimpse*** *generates an ensemble of predictors* p *for different dimensions of the search space from the* ***Blueprints***. For example, $p_{\text{tile\_x}}$ and $p_{\text{tile\_y}}$ are generated for tile_x and tile_y, respectively. For each configuration sampled from the *Hardware-Aware Exploration*, ensemble predictors *vote* the validity of the configuration. Sampler rejects the configuration if considered invalid by more than τ^2 of the predictors. As each of these predictors are *hardware-aware*, their accuracy is significantly higher than other *hardware-agnostic* approaches.

[2]We use $\tau = \frac{1}{3}$, found through a gridsearch hyperparameter search.

Algorithm 4. Overall flow of **Glimpse** with **Blueprint**.

Data: Π: Layer specification, Θ: **Blueprint**
Result: x^*: Optimal configuration
// **Section 4.4.1**: Generate prior distributions
$\hat{f} \leftarrow \mathcal{H}(\Pi, \Theta)$
for $i \leftarrow 0$ **to** n **do**
    // **Section 4.4.2**: Hardware-Aware Exploration
    $xs \leftarrow$ simulated annealing with $\hat{f}$ as energy function
    $xs_{pruned} \leftarrow$ meta-optimizer with Θ_k as hints
    // **Section 4.4.3**: Hardware-Aware Sampling
    $xs_{sampled}\ gets$ sampling to minimize invalid configs.
    // Run real hardware measurements
    for $x \in xs_{sampled}$ **do**
        $y \leftarrow f(x)$; $\mathcal{O} \leftarrow (x, y)$; $x^* \leftarrow x$ with maximum y
    end for
    // Update cost model
    update f using $\mathcal{O}$
end for

Design.

Instead of a *large and complex monolithic* predictor could be an alternative design point for *Hardware-Aware Sampling* in **Glimpse**, we use an *ensemble* of *light-weight* predictors for two reasons. First, statistically speaking, ensemble methods have been shown to yield a better predictive performance than could be obtained from any of the constituent predictor alone. In this case, comparable to a *large complex monolithic* predictor. In fact, smaller predictors are more appropriate considering the dearth amount of data. Furthermore, as key design consideration for neural compilers is the compilation speed for higher overall productivity, *ensemble* of *light-weight* predictors were used to minimize computational overhead of prediction. These predictors are super fast as they are threshold-based: their time complexity is $\mathcal{O}(1)$ over Chameleon [5]'s $\mathcal{O}(nkI)$, where n is the number of samples, k is the number of clusters, and I is the number of iterations.

Table 4.1. Details of the DNN models.

DNN Models	Dataset	Number of Tasks
AlexNet	ImageNet	12 (5 conv2d, 4 winograd conv2d, 3 dense)
VGG-16	ImageNet	21 (9 conv2d, 9 winograd conv2d, 3 dense)
Resnet-18	ImageNet	17 (12 conv2d, 4 winograd conv2d, 1 dense)

Table 4.2. Details of the GPUs.

Hardware	Generation (gencode)
Titan-Xp	Pascal (sm_61)
RTX 2070 Super	Turing (sm_75)
RTX 2080 Ti	Turing (sm_75)
RTX 3090	Ampere (sm_86)

Integration and implementation.

Algorithm 4 summarizes the overall flow of **Glimpse** with **Blueprint**. We use PyTorch [135] to implement $\mathcal{H}$ for prior generation and the meta-optimization.

4.5 Evaluation

We integrate **Glimpse** with Apache TVM v0.8 [34] to perform evaluation of both component and end-to-end scenario. We ran our framework on host machine with AMD Ryzen 7 3700X, 64GB DDR4, with NVIDIA RTX 2070 Super, and used CUDA 11.3 to program DNNs onto GPUs as summarized in Table 4.2. We compare **Glimpse** against the state-of-the-art optimizing compilers: AutoTVM [35], Chameleon [5], and DGP [165]. We optimize AlexNet [96], VGG-16 [161], and ResNet-18 [75] on multiple generations of GPUs connected via RPC (Titan Xp, RTX 2070 Super, RTX 2080 Ti, RTX 3090) as summarized in Table 4.1.

4.5.1 Blueprint

Design space exploration of Blueprint.

Unlike *hardware-agnostic* proposals [5, 35, 103, 145, 165, 191], **Glimpse** utilizes the information embedded in **Blueprint** to speed up the neural compilation. As such, minimizing the information loss about the architectural specifications listed in the data sheets [130] is imperative. Importantly, the **Blueprint** needs to be designed to have as low overhead as possible. Figure 4.4 summarizes the design space exploration of **Blueprint**. Our design of **Blueprint** strikes balance between the amount of information in the vector (¡ 0.5% *for minimal information loss in terms of Root-Mean-Squared-Error (RMSE) while using* ***Blueprint***) versus the size of the embedding (*for fast compilation*).

Prior distribution generation with Blueprint.

Figure 4.5 plots the distribution of the initial configurations sampled with and without **Blueprint** for representative GPU / DNN Model / Layer combinations. The results show that using **Blueprint** improves the initial configuration. In fact, some layers even reach the optimal configuration within first few steps of optimization, enabling *sub-minute* compilation time. In contrast, AutoTVM [35] and Chameleon [5] reports that it takes at least few hundred steps (around hour per layer) to reach a similar performance.

We also compare against transfer learning which is the core mechanism used in AutoTVM [35] to reuse knowledge from prior optimization runs. We used logs from all but

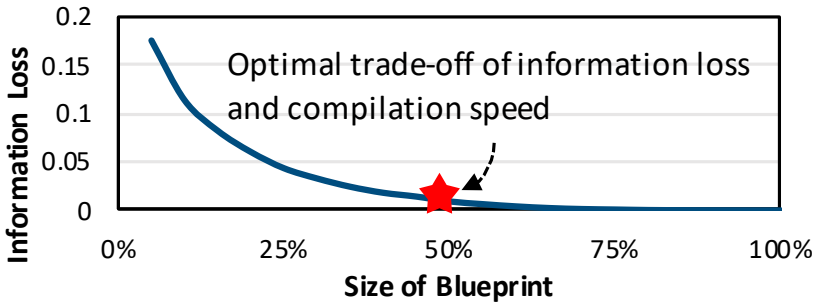


Figure 4.4. Design space exploration of **Blueprint**. Point marked with red star strikes balance between the information loss from compression and the compilation time.

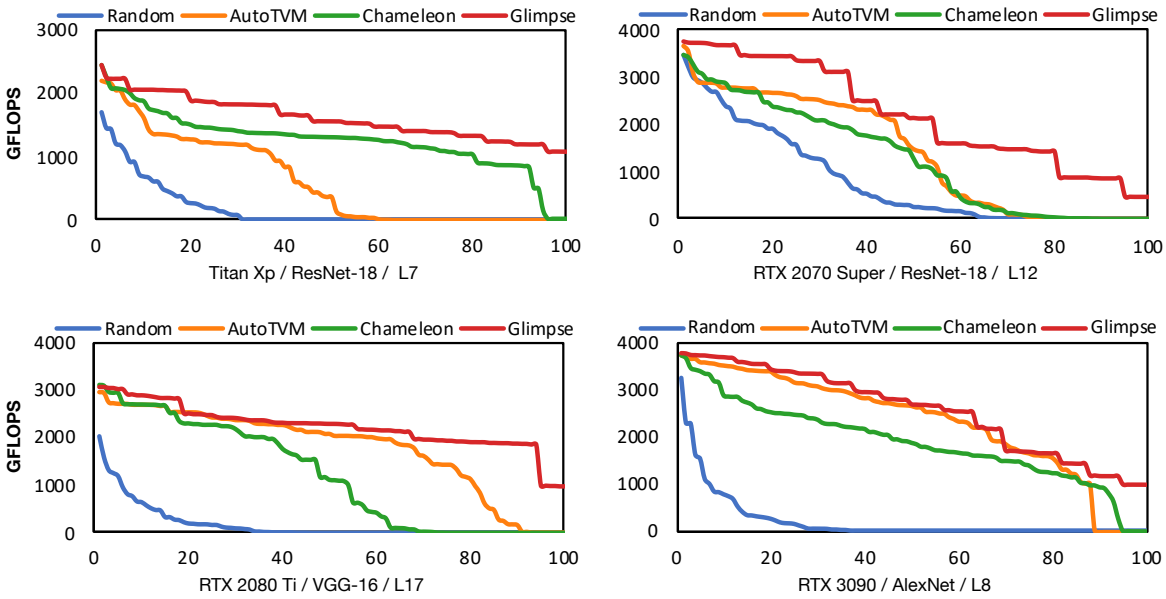


Figure 4.5. Comparison of initial sampled configurations from random search, AutoTVM, Chameleon, and Glimpse for representative combinations of DNN layers and GPUs. There are 100 configurations in each set and are sorted in descending order.

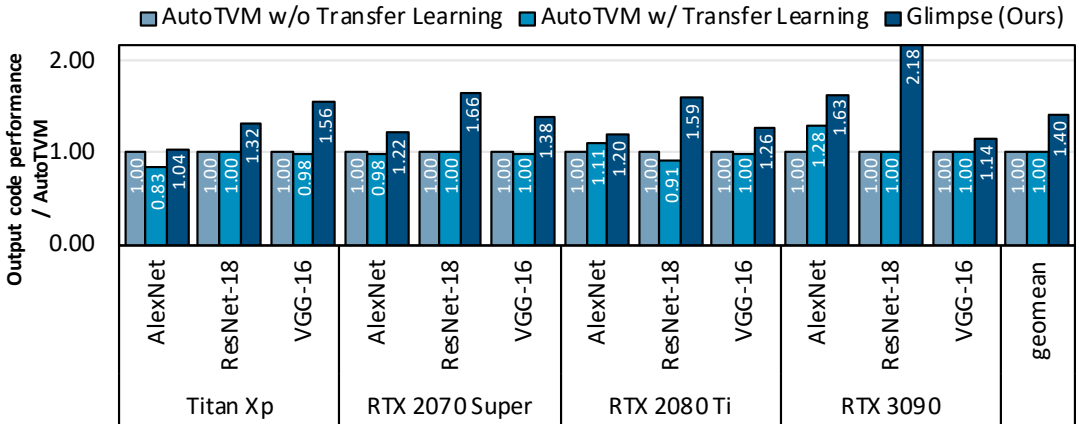


Figure 4.6. Comparison to AutoTVM transfer learning, provided 100 seconds optimization time budget per layer.

combination of target network and hardware for transfer learning, and plot the output code performance when provided 100 seconds of budget per layer. Figure 4.6 shows that **Blueprint** outperforms both AutoTVM with and without transfer learning by 40.0%. Despite the belief that transfer learning would be sufficient to transfer knowledge among tasks, it sometimes performed worse than baseline AutoTVM. In fact, the results in AutoTVM [35] also suggests that transfer learning only achieves fraction of the final binary performance that are achieved with hundreds to thousands of hardware measurements. These results suggest that knowledge from transfer learning not only necessitates significant number of additional real hardware measurements but also is prone to being misguided. In contrast, **Blueprint** provides effective initializations to the **Glimpse** compiler and consistently yields the best performance.

4.5.2 Hardware-Aware Explorer

Speed of convergence.

In AutoTVM [35] and Chameleon [5], authors formulate a cost minimization with a batch of Markov chains and use optimization algorithms such as simulated annealing and reinforcement learning. While the output code performance is determined by the final cost the optimization achieves, the number of updates or steps these Markov chains take is the key factor that determines the optimization time. Figure 4.7 compares the number of search steps among the three works: AutoTVM [35], Chameleon, and **Glimpse**[3]. **Glimpse** achieves 5.07× and 2.55× speed-up against AutoTVM and Chameleon, which shows that **Glimpse**'s Hardware-Aware Explorer may converge significantly faster than optimizing compilers for single hardware. This notable reduction in the number of search steps come from the **Glimpse** compiler's ability to take hints from the mathematical embeddings of the **Blueprints** about the optimization steps, on when and where to explore and exploit.

[3]Here, we do not provide comparisons against acquisition functions such as Expected Improvement (EI), and Upper Confidence Bound (UCB). AutoTVM's experimental results show that they yielded no improvement.

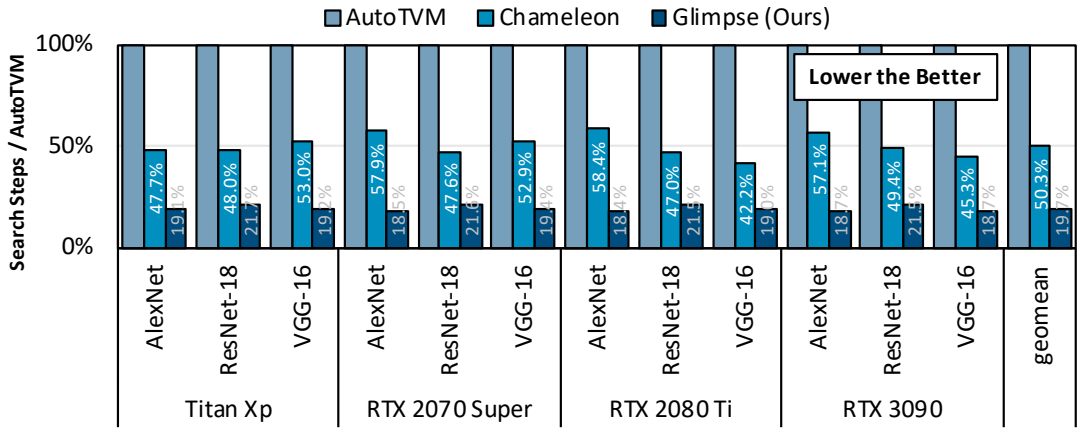


Figure 4.7. Comparison in number of search steps. Results show **Glimpse** provides significant reduction.

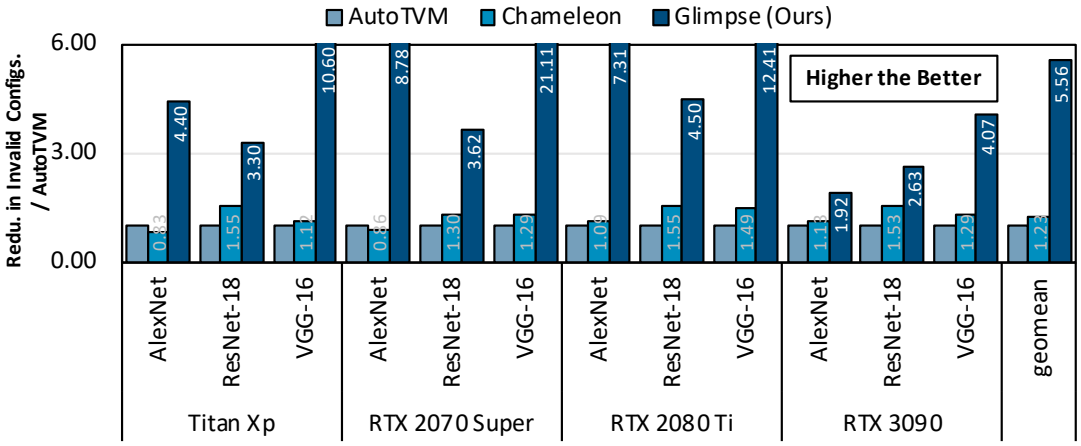


Figure 4.8. Comparison to *hardware-agnostic* sampling approaches in reduction of invalid configurations.

4.5.3 Hardware-Aware Sampling

There is an intrinsic issue of the search space provided by TVM [34] where there exists numerous invalid configurations leading to large delays in compilation speed and waste in GPU hours. In current compilers, around 10% of the measurements made were invalid. Figure 4.8 presents the reduction in fraction of invalid configurations with respect to the number of hardware measurements for sampling in Chameleon [5] and **Glimpse** compared to AutoTVM [35]. **Glimpse** reduces the invalid configurations by 5.56× and 4.53× compared to AutoTVM and Chameleon, respectively. The results suggest, that weak statistical guarantees of the sample synthesis and the adaptive sampling to reduce the frequency of these invalid configurations are insufficient to cope with the above issue. Instead, Hardware-Aware Sampling in **Glimpse** effectively reduces the number of hardware measurements using the statistical approach.

4.5.4 Putting It All Together

Figure 4.9 and Figure 4.10 compares the end-to-end compilation time and the output binary performance of **Glimpse** compared to state-of-the-art hardware-agnostic techniques: AutoTVM [35], Chameleon [5], and DGP [165]. First, **Glimpse** cuts down the search time 6.73×, 1.51×, and 1.92× compared to AutoTVM, Chameleon, and DGP respectively, while achieving the best inference latency of the output binary. The gains come from the collaboration of (i) prior distributions generated from **Blueprint**, (ii) effective balance of exploration-exploitation as well as hardware-awareness of Hardware-Aware Exploration, and (iii) hardware measurements reduction with statistical Hardware-Aware Sampling. Table 4.3 summarizes the search reduction (GPU time), inference time improvement. Also, following [165], we present Hyper-Volume (HV) to measure the efficacy of different approaches considering multi-objectives.

$$\text{HV} = \text{Search Reduction} \times \text{Inference Reduction} \times 100 \tag{4.2}$$

Glimpse cuts down the search time significantly compared to *hardware-agnostic* tech-

Table 4.3. Comparisons to state-of-the-art optimizing compilers [5, 35, 165] for Hyper-Volume (HV), a metric that summarizes the multiple objectives of optimizing compilation: search time (GPU Hours) and end-to-end model inference latency (milliseconds).

Model	AutoTVM (NeurIPS'19)					Chameleon (ICLR'20)			DGP (ICCV'21)			Glimpse (Ours)		
	Σ_{GPU} Search (GPU Hours)	Mean Inference (ms)				Search Redu. (%)	Inference Redu. (%)	HV	Search Redu. (%)	Inference Redu. (%)	HV	Search Redu. (%)	Inference Redu. (%)	HV
		Titan Xp	RTX 2070 Super	RTX 2080 Ti	RTX 3090									
AlexNet	18.65	1.0277	0.9662	0.7872	0.4799	72.16	5.88	4.2430	65.96	6.91	4.5578	**82.84**	**6.94**	**5.7492**
ResNet-18	36.53	1.0258	1.3305	0.9282	0.5518	76.67	4.16	3.1895	70.43	5.17	3.6412	**84.85**	**5.18**	**4.3954**
VGG-16	49.08	3.9829	4.5751	3.1865	1.8926	82.56	3.44	2.8401	76.83	4.24	3.2576	**87.37**	**4.24**	**3.7045**

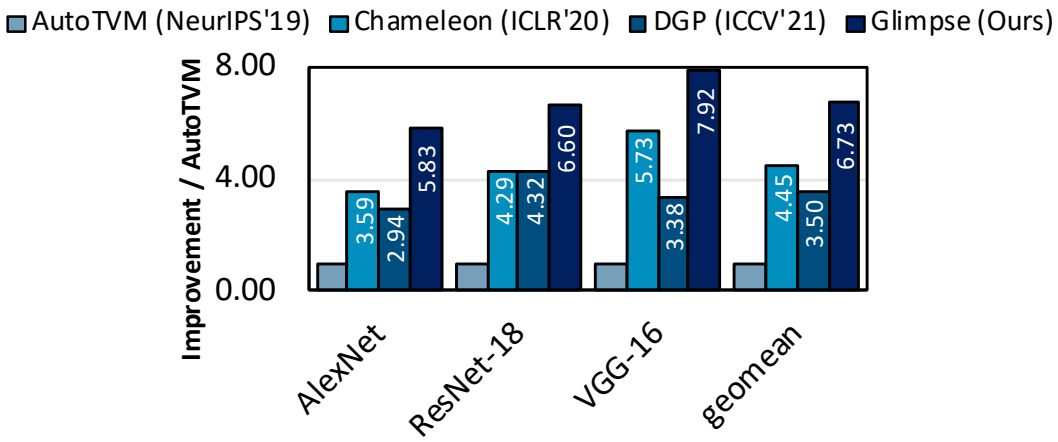


Figure 4.9. End-to-end improvement in optimization time.

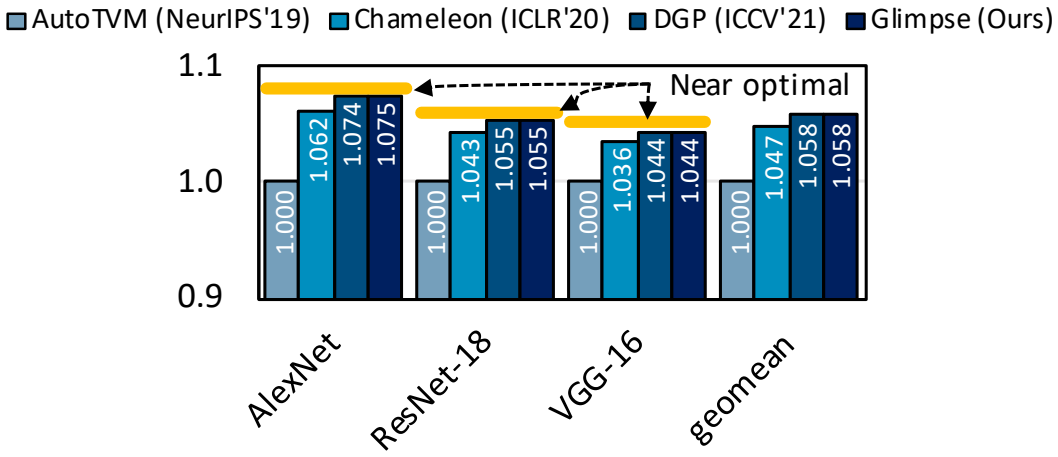


Figure 4.10. End-to-end improvement in inference speed.

niques while achieving the fastest inference. Therefore, **Glimpse** shows the highest HV score: the *best trade-off between search time and inference speed*. Even if inference speed is the main criterion [165], **Glimpse** provides the *best inference speed*.

4.6 Related Works

A large body of inspiring works on neural compilers have been introduced to generate high-performance binaries for innovative neural accelerators [104]. While many neural compilers such as TVM [34] *blindly* rely on the statistical guarantees of stochastic optimization, this paper uniquely explores the use of hardware blueprints, a proxy of the complete architecture description

to improve the initialization, exploration, and the sampling to improve neural compilation. Below, we discuss the most related works:

Neural compilers.

While TVM [34] significantly improves inference speed of DNNs, it comes with an intractable search space. AutoTVM [35] develops learned cost models and TenSet [192] provides large scale dataset to improve the cost models to approximate this large search space To find optimal configurations, TVM [34] builds on random search and genetic algorithms while AutoTVM [35], GGA [124], and Chameleon [5] explored simulated annealing, guided genetic algorithm, and reinforcement learning to further improve the search efficacy. [165] explored deep Gaussian process to transfer knowledge to different layers on a single target GPU. Prior works were *blind* about the hardware during optimization, discarding the opportunity to transfer experiences between optimization runs on different hardware. While these blind approaches incur large GPU hours for compilation, this paper explores the use of **Blueprint** as a mechanism to let compilers *perceive* the target hardware and predict the search space landscape to expedite the search, reducing the overall GPU hours while also achieving faster inference.

Meta-learning for neural compilation.

Meta-learning [55] proposes a mechanism to learn to learn that guides and expedites optimization. For example, MetaBO [172] explored meta-learning in the context of Bayesian optimization for more sample efficient optimiation. In the context of neural compilation, MetaTune [145] leverages meta-learning to expedite the convergence of the cost models. In contrast, **Glimpse** incorporates a unique blend of meta-optimizer that takes domain-knowledge about the architectures as input. Specifically, we develop a mechanism that feeds the *Hardware-Aware Exploration* with information in **Blueprint**, which led to significant reduction in compilation time as well as the inference latency.

4.7 Conclusion

This paper presents **Glimpse**, a neural compiler that exclusively explores *mathematical embeddings* of the hardware **Blueprints** to improve both the speed and the performance of neural compilation. Experiments on modern DNNs on a multiple generations of hardware shows that *hardware-awareness* of **Glimpse** significantly reduces the compilation time while achieving the best inference latency. Encouraging results with **Glimpse** of **Blueprint** for neural compilation suggest significant potential in abstractions that encode domain knowledge to improve optimization.

4.8 Future Directions

Glimpse [3] was an effort to (1) develop *abstractions* for hardware and (2) apply it to deep learning compilers as a case study of *hardware-aware optimization*. Encouraging results suggest a significant potential in diving deeper into each of directions.

Abstractions for heterogenous hardware.

While **Glimpse** focused on GPUs as the target hardware, there are many different types of hardware the enable modern computing. For example, despite the wide-spread use of GPUs for training and inference at server scale such as Inference-as-a-Service (INFaaS), many edge devices still rely on CPUs or small micro-controllers. Also, recent Cambrian explosion of deep learning hardware and their heterogeneity limits the naive extension of the work to all hardware. To this end, one possibility is to explore graph neural networks as a means to develop a generalized embedding of hardware.

Hardware-aware optimization.

Hardware-aware neural architecture search [28, 29, 173], model compression [76], and quantization [50, 174] have shown significant potential in leveraging the feedback from hardware for various optimization problems. In fact, the very baseline of the **Glimpse** [3], AutoTVM [35]

and Chameleon [5] both leverage the feedback from the hardware for the code optimization problem. While naive black-box optimization provide a simple and intuitive way to utilize the feedback from the target hardware, it can be an inefficient way to perform optimization. To this end, researching the hardware-aware gray-box optimization for the above model design and compression problems can potentially lead to significant benefits. Likewise, it would be interesting to extend the ideas in [3] to various hyperparameter tuning, network scheduling, hardware architecture exploration, and even physical design flow.

Acknowledgement. Chapter 4, in part, contains a re-organized reprint of the material as it appears in Design Automation Conference (DAC) 2022. Ahn, Byung Hoon; Kinzer, Sean; Esmaeilzadeh, Hadi. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Expanding the Scope to End-to-End Intelligent Systems

Previous sections suggest exciting potential of *Hybridization of AI and Foundational Algorithms in Optimizing the Execution of Deep Neural Networks (DNNs)*. However, the *End-to-End Intelligent Systems consists of Algorithms from More Domains beyond just DNNs*. This section expands the scope beyond accelerating DNNs to accelerating the end-to-end applications that constitute intelligent systems. To this end, this section explores the acceleration of end-to-end applications in the intelligent systems and introduces *Cross-Domain Multi-Acceleration* and develops a set of abstractions with a execution engine to showcase its benefits. This work lays the foundation for further investigations into *AI-Enabled Compilation of Intelligent Systems*.

5.1 Programming Abstractions for Cross-Domain Multi-Acceleration

Field-Programmable Gate Array (FPGA) accelerators offer performance and efficiency gains by narrowing the scope of acceleration to one algorithmic domain. However, real-life applications are often not limited to a single domain, which naturally makes *Cross*-Domain *Multi*-Acceleration a crucial next step. The challenge is, existing FPGA accelerators are built upon their specific vertically-specialized stacks, which prevents utilizing multiple accelerators from different domains. To that end, we propose a pair of dual abstractions, called **Yin-Yang**, which

work in tandem and enable programmers to develop cross-domain applications using multiple accelerators on a FPGA. The **Yin** abstraction enables cross-domain algorithmic specification, while the **Yang** abstraction captures the accelerator capabilities. We also develop a dataflow virtual machine, dubbed **XLVM**, that transparently maps domain functions (**Yin**) to best-fit accelerator capabilities (**Yang**). With six real-world cross-domain applications, our evaluations show that **Yin-Yang** unlocks 29.4× speedup, while the best single-domain acceleration achieves 12.0×.

5.2 Introduction

Field-Programmable Gate Arrays (FPGAs) have emerged as a promising acceleration platform for diverse application domains both at the edge and on the cloud (Amazon F1 instances [15] and Microsoft SmartNICs [56]). Despite the benefits, the accelerators by definition limit the scope of acceleration to an algorithmic domain, while real-life applications [97, 105, 150] often extend beyond a single domain. It is evident that for such cross-domain applications, utilizing multiple accelerators–even on a single FPGA–from different domains can unlock new capabilities and offer higher performance and efficiency. However, each accelerator often comes with its own vertically-specialized domain-specific stack, as illustrated in Figure 5.1, which by design is difficult to conjugate with other stacks. Thus, there is a need for a *horizontal programming abstraction* that enables programmers to develop end-to-end applications without delving into the isolated accelerator stacks.

To that end, this paper sets out to devise such abstractions by building upon a collection of programmer-transparent layers. We first devise a pair of dual abstractions, called **Yin-Yang**, where 1) the **Yin** abstraction allows *domain experts* to concisely describe the capabilities of each domain, and 2) the **Yang** abstraction enables *hardware designers* to abstractly denote compute capabilities and data interfaces for their FPGA accelerators, henceforth referred to as *engines*. The **Yin** abstraction also offers a lightweight programming interface that allows *programmers* to aggregate **Yin**-defined cross-domain capabilities together as a single program,

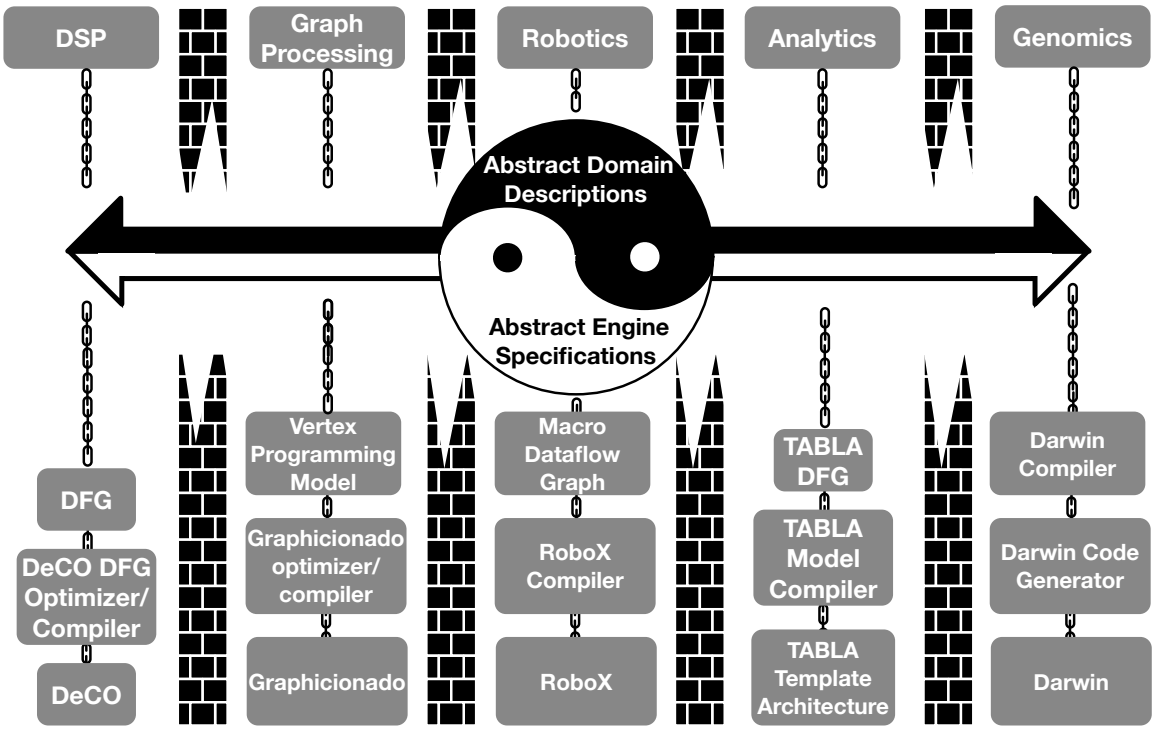


Figure 5.1. Yin-Yang dual abstractions break the vertical barriers of domain-specific stacks and enable cross-domain multi-acceleration in the heterogeneous cloud.

while preserving the domain boundaries. Then, to enable the two abstractions to work in tandem, we develop **XLVM** (Accelerator-Level Virtual Machine) and its execution workflow is delineated in Figure 5.2. **XLVM** is a dataflow virtual machine that builds and executes the program as a Queued-Fractalized Dataflow Graph (QF-DFG). In QF-DFG, like fractals, each node is another QF-DFG but at a progressively finer granularity, until a node is a primitive scalar operation. For given QF-DFG, the **XLVM**'s Engine Selector chooses components of the application (i.e., nodes of QF-DFG) to appropriate engines, using the engine specifications from **Yang** abstraction. **XLVM** also comes with Engine Compiler that compiles the individual engines into runnable executables and links them as a unified execution flow by automatically converting dependencies (i.e., edges of QF-DFG) to inter-engine communication between FPGA accelerators.

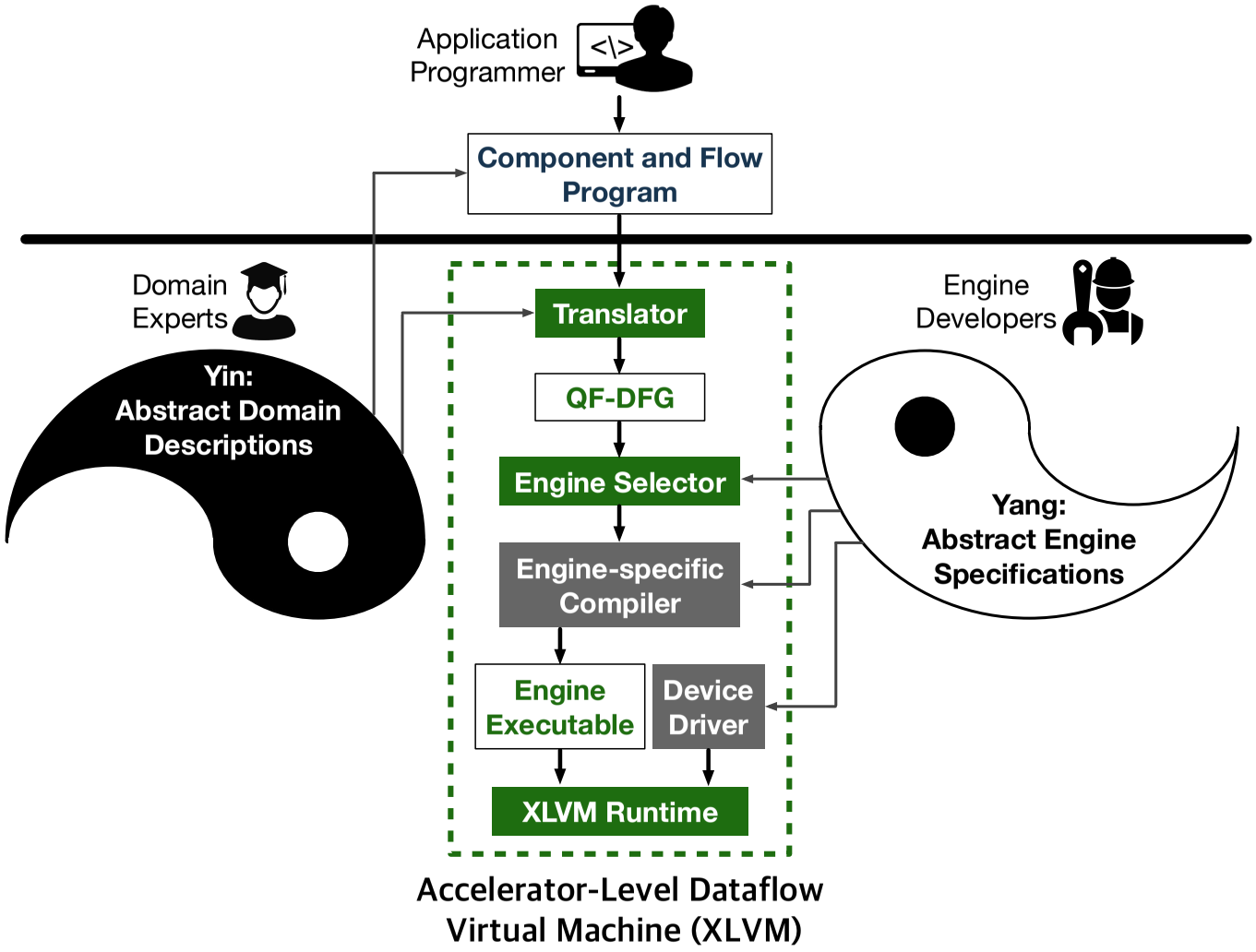


Figure 5.2. **Yin-Yang** dual abstractions and **XLVM** for cross-domain multi-acceleration.

We collect diverse real-world applications and offer it as an open-source benchmark suite for cross-domain multi-acceleration. These applications range from deep brain stimulation, geological exploration, film captioning, stock exchange, medical imaging, and surveillance. Each of these benchmarks comprises algorithms from more than one domain where each is accelerated across multiple domain-specific accelerators. Using this benchmark suite, we evaluate the proposed abstractions and its concrete system implementation. By enabling cross-domain multi-acceleration, our work improves speedup from 12.0× to 29.4× (i.e., 145% extra benefits on average) compared to an end-to-end execution with a single FPGA accelerator that offers the highest gain. These results suggest the effectiveness of the **Yin-Yang** abstractions and their associated system framework in enabling cross-domain multi-acceleration.

5.3 Yin Abstraction

5.3.1 Abstract Domain Description

The goal of **Yin** abstraction is to delineate the capabilities of each domain, without any accelerator or application specific constructs. For every domain, a unique abstract definition, called domain description, is provided independently by the domain experts to pre-define domain's common capabilities. As the objective is to allow multi-acceleration, a domain description consists of a set of capabilities, each of which is a potential agent for acceleration. However, note that the capabilities defined in domain descriptions are accelerator-agnostic and not linked to a specific accelerator. Yet, these capabilities can be mapped to various accelerators through our virtual machine **XLVM** (Section 5.5). To enable programmers to use the domain capabilities in their applications, we also provide a set of lightweight programming model (Section 5.3.2). Using the programming interface, programmers can develop their applications by importing the domain descriptions and instantiating the domain capabilities, while still preserving the domain specificity, interface, and boundary of each instantiation.

```
1   import domain.abstract_data.array as array
2
3   domain digital_signal_processing {
4       default reference fftw
5
6       capability fft(input array in_array,
7                      output array out_array,
8                      param int length,
9                      param int axis,
10                     param string norm)
11      capability band_pass_filter(input array in_array,
12                                  output array out_array,
13                                  param array frequency_bands)}
14      capability convolution(input array in_array,
15                             output array out_array,
16                             state array weight,
17                             param string mode,
18                             param string boundary)}
```

Figure 5.3. Domain description for Digital Signal Processing (DSP).

Example domain description for digital signal processing.

Figure 5.3 illustrates an example domain description for the digital signal processing domain. The domain descriptions are composed of the following: domain name, a set of capabilities with input/output semantics, a default reference implementation, and a cost model. The domain name is specified using the keyword domain on line 3. On line 4-18, the domain capabilities and their input/output are specified. For instance, lines 14-18 define the convolution capability, a frequently used operation in digital signal processing and is often accelerated by DSP accelerators. The capability keyword denotes the computational capability supported in this domain, and is followed by a unique denotation. Each capability has required input (input) and output (output) specifications as the arguments to its definition. The input and output data type and dimensions form the interface to the computation capability. We also have the state keyword that semantically stores the state across multiple executions. State is necessary for domains that share a temporal component such as robotics, data analytics, and deep learning. The param keyword denotes data whose values remain constant across executions.

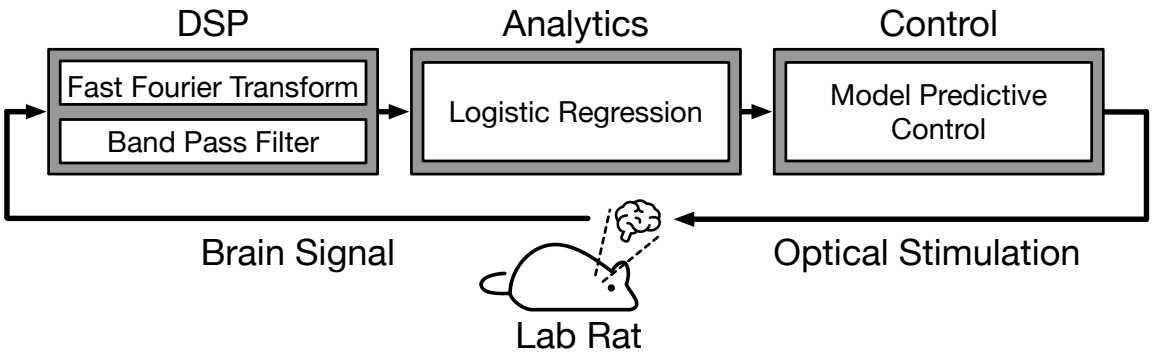


Figure 5.4. Deep brain stimulation.

5.3.2 Component & Flow Programming Model

The domain descriptions define the capabilities of individual domains that constitute end-to-end applications, but there is a need for programming interface that enables programmers to use the capabilities for application development without concerning the low-level hardware details. Due to the modular nature of the **Yin** abstraction, the Component and Flow programming model (CNF) is built upon lightweight annotations to create the linkage between domain description capabilities and end-to-end application kernels. Components and Flows in CNF represent the computation and dataflow in between, respectively. In particular, Component is a language construct that is explicitly used within the code, whereas the Flow is implicitly present in between.

Deep brain stimulation.

To demonstrate the use of CNF programming model, we take a cross-domain application, *deep brain stimulation* [150], as an example. Figure 5.4 illustrates the algorithms and the workflow of this cross-domain application. This application takes and processes the electrophysiological response of the brain (DSP) to measure the biomarkers (Analytics), and generates a set of optical stimulations (Control) for memory enhancement in rats. In this setup, the electrophysiological activity of the brain is collected in real-time, and passed through Fast Fourier Transform (FFT) and a set of Band Pass Filters (BPF) of distinctive frequency bands (i.e., delta

(0.5-4 Hz), theta (4-8 Hz), alpha (8-12 Hz), beta (12-30 Hz), and gamma (30-100 Hz)). Next, the pipeline uses Logistic Regression (LR) to decode and classify these brain waves to be used as biomarkers. Based on the classification output, a Model Predictive Control (MPC) process configures the synthesized brain waves (i.e., amplitude, frequency, and duration). Offloading the major compute-heavy algorithms to the corresponding accelerators–FFT to DeCO [83], logistic regression to Tabla [110], and control optimization to RoboX [147]–will provide runtime performance improvements[1].

Example CNF code for deep brain stimulation.

Figure 5.5 illustrates a CNF implementation of deep brain stimulation. First, lines 1–3 import domain descriptions into the application and bring the available pre-defined capabilities. Then, CNF enables programmers to express 1) the component boundaries, 2) its interfaces, and 3) the hierarchical structure. On line 5, the components are defined using the keyword Component along with its inputs (e.g. wave) and outputs (e.g. stimuli), which is followed by the code for its computation on lines 6–15. CNF also allows the programmer to express arbitrary levels of component hierarchy, where Components may be defined inside a Component. Once a component has been defined, the programmer can instantiate it as many times as needed to express the algorithm.

5.4 Yang Abstraction

5.4.1 Abstract Engine Specification

To manage various accelerators and to allow flexible additions of newly developed accelerators, we devise the **Yang** abstraction, as the counterpart to the **Yin** abstraction. The **Yang** abstraction offers a means for the accelerator developers to abstractly describe the accelerator specifications. In this paper, an *engine* denotes an abstract compute platform, which exclusively

[1]The original work [147] proposed RoboX as an ASIC but it is straightforward to develop the architecture as an FPGA accelerator.

```
1  import domain.digital_signal_processing as dsp
2  import domain.data_analytics as analytics
3  import domain.controls as controls
4
5  with Component(inputs=[wave], outputs=[stimuli]) as DBS:
6      with Component(inputs=[wave], outputs=[filtered]) as DSP:
7          signals = dsp.fft(wave)
8          filtered = dsp.band_filter(signals)
9      with Component(inputs=[filtered], outputs=[logit]) as Analytics:
10         logit = analytics.logistic_regression(filtered)
11     with Component(inputs=[logit], outputs=[stimuli]) as Control:
12         stimuli = controls.model_predictive_control(logit)
13     bands = DSP(wave)
14     logit = Analytics(bands)
15     stimuli = Controls(logit)
16
17 while True:
18     wave = mouse.measure()
19     stimuli = DBS(wave)
20     mouse.apply(stimuli)
```

Figure 5.5. Implementation of deep brain stimulation with CNF programming model.

supports a single domain and is able to serve a subset of the capabilities defined in the corresponding domain description. Thus, **Yang** abstraction allows the engine developers to specify the provided capabilities and communication interfaces of an engine as a structured specification, called engine specification.

Example engine specification describing an digital signal processing accelerator.

Figure 5.6 illustrates an example engine specification of DeCO [83] using our engine specification language. To provide a flexible abstraction that can be used by a variety of engines, but also to ensure multi-acceleration, an engine specification needs to express 1) its own capabilities, and 2) the interface it exposes to connect with different engines. Line 3 shows that the engine name is specified using the keyword engine and domain (digital_signal_processing). On Line 4–7, how the engine communicates its input and output with the outside world is specified using the keyword interface. The engine specification provides pre-defined interfaces such as FIFO, SRAM, or BRAM, etc. Also, its capabilities (i.e., fft and band_pass_filter) and their

```
1   import engine.hw_interfaces as inouts
2
3   engine deco implements digital_signal_processing {
4     interface input_mem = inouts.bram
5     interface output_mem = inouts.bram
6     interface weight_mem = inouts.bram
7     interface config = inouts.bram
8
9     capability fft(input_mem int8[N][M] { mem[n*N+m] } in_array,
10                   output_mem int8[N][M]  { mem[m*M+n] } out_array,
11                   config int8 length,
12                   config int8 axis,
13                   config char[ ] norm)
14     capability band_pass_filter(input_mem int8[N][M]   { mem[m*M+n] } in_array,
15                                 output_mem int8[N][M]   { mem[m*M+n] } out_array,
16                                 config int8[ ] frequency_bands)
17
18     fusion {fft: [band_pass_filter], band_pass_filter : [ ] }
19     cost {path: "deco_model"}
```

Figure 5.6. Engine specification of DeCO engine.

semantics for input, output, weight, and configuration memory are specified in lines 9–16.

5.4.2 Hints for Engine Selection

Our **Yang** abstraction also offers two keywords, fusion and cost, to allow the engine developers to provide engine-specific information, which can be used as *hints* for the engine selection process later in **XLVM**. The keyword, fusion, denotes a set of capabilities that can be sequentially executed internally in an engine, while avoiding external data communication with the host or other engines. For instance, line 18 illustrates that the fft capability can be fused with the band_pass_filter capability, while band_pass_filter cannot be fused with any other capabilities on DeCO. The cost construct lets engine developers specify a means to estimate the latency of capabilities. This can be mapped to a cycle-accurate simulator, hard-coded metric, or a machine learning-based cost models as in AutoTVM [35].

5.5 XLVM: Accelerator-Level Virtual Machine

The **Yin-Yang** abstractions need to be realized as a unified execution flow so that the application is executed efficiently and the maximal gains can be achieved from cross-domain multi-acceleration. To accomplish this objective, we devise Accelerator-Level Dataflow Virtual Machine (**XLVM**), which is at the confluence of the **Yin-Yang** abstractions. **XLVM** preserves and translates the CNF program as a queued-fractalized dataflow graph (QF-DFG) intermediate representation (IR). Then, we develop Engine Selector that selects the engines for the application, maximizing acceleration gains. Finally, we also develop Engine Compiler, which compiles the individual engines into the corresponding runnable executables and links them as a unified execution flow.

5.5.1 Queued-Fractalized Dataflow Graph (QF-DFG)

For effective engine selection and runtime orchestration, it is crucial to have an intermediate representation (IR) that 1) preserves the program and domain semantics (input, output, interface, and hierarchy of components), and 2) is flexible to support any granularity required for multi-acceleration. As such, we devise queued-fractalized dataflow graph (QF-DFG), which is designed to capture the details of the program such as dependency (order of execution), functionality (operation), and compositionality (hierarchy) of the CNF programs. In QF-DFG, each edge denotes a dataflow and node denotes an operation of multiple levels of granularity, progressing from coarse granular nodes to finer nodes until primitive scalar operations are reached. Figure 5.7 and Figure 5.8 shows a snippet of the QF-DFG IR, which corresponds to the CNF program in Figure 5.5.

5.5.2 Engine Selector

QF-DFG is a target-independent IR which, when created from CNF, is oblivious to the target engines for execution, similar to target-independent IR stages of the traditional compilers.

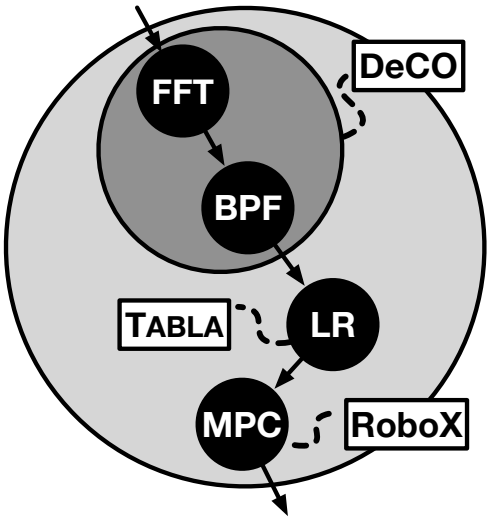


Figure 5.7. Visualization of QF-DFG of deep brain stimulation.

```
1   Component:
2     cid: 0, name: DBS,
3     sub_cids: [1, 4, 5]
4     inputs: (name: wave, flow_id: 0)
5     outputs: (name: stimuli, flow_id: 1)
6   Component:
7     cid: 1, name: DSP,
8     sub_cids: [2, 3], super_cids: 0
9     inputs: (name: wave, flow_id: 0)
10    outputs: (name: bands, flow_id: 2)
11  Component:
12    cid: 2, name: fft,
13    sub_cids: null, super_cids: 1
14    inputs: (name: wave, flow_id: 0)
15    outputs: (name: signals, flow_id: 2)
16    domain: dsp, engine_name: DeCO,
17    capability: fft
18    ....
19  Flow:
20    flow_id: 0, name: wave, is_queue: true
21    source_cid: 3, dest_cid: 0
22  Flow:
23    flow_id: 1, name: stimuli, is_queue: true
24    source_cid: 0, dest_cid: 3
```

Figure 5.8. Textual form of QF-DFG of deep brain stimulation.

Unlike traditional compilation processes where the target platform is explicitly known, the duality of domains and engines provided by **Yin-Yang** opens a new avenue for optimal target engine determinations as it exhibits the following properties: 1) a domain possibly has multiple engines that can support different subsets of its capabilities; and 2) every engine, even within the same domain, has different performance energy tradeoffs. Thus, to choose an optimal combination of engines for a given QF-DFG and a set of engine specifications, **XLVM** is equipped with Engine Selector, which exploits 1) a simple cost model as a proxy to estimate the performance of engine assignment, and 2) an optimized objective function.

Cost model and objective function.

We model the execution time of end-to-end multi-acceleration applications using three cost functions: 1) computation latency ($\mathcal{T}$), 2) data copy overhead ($\mathcal{C}$), and 3) data format conversion cost ($\mathcal{D}$). To simplify the design, we model the overall cost for the given QF-DFG as a sum of these functions, which does not consider the dynamic runtime factors such as pipelined execution and bandwidth contention with other applications. Using this cost model, we formulate the objective function of Engine Selector as a combination of engines S for the QF-DFG from the candidate engine set E, that minimizes total execution latency:

$$\underset{S \subset E}{\operatorname{argmin}} \ \text{Cost} = \sum_{i} \mathcal{T}_i + \sum_{ij} \mathcal{C}_{ij} + \sum_{ij} \mathcal{D}_{ij}, \quad \text{for } i, j \in S$$

Algorithm 5 illustrates the engine selection process, which takes a QF-DFG and maps the graph nodes to a set of available engines that minimize the expected latency by optimizing the cost function. Engine Selector conducts a brute-force search of all possible engine assignment combinations to the QF-DFG and formulates the candidate set for the engine selections (Line 4–5). Then, Engine Selector evaluates the cost function per each candidate selection and chooses the candidate that imposes the minimum latency cost (Line 7–15). The selection results are augmented to the QF-DFG as metadata. While we demonstrated the engine selection process optimizing for the execution time, the objective function can be updated for other objectives such

Algorithm 5. Engine selection algorithm for QF-DFG

```
1: Input: QF-DFG G(N, E)
2: Output: Engine Selection S
3: candidates ← ∅, cost ← {}
4: while new_candidate_exists() do
5:     candidates ← candidates ∪ {(n, e) | ∀ n ∈ N, ∃ e ∈ n.domain.engines}
6: end while
7: for c in candidates do
8:     cost[c] ← 0
9:     for (n, e) in c do
10:        cost[c] ← cost[c] + 𝒯(n, e)
11:        for (n_child, e_child) in children((n, e)) do
12:            cost[c] ← cost[c] + 𝒞(e, e_child) + 𝒟(e, e_child)
13:        end for
14:    end for
15: end for
16: S ← find_engine_selection_with_minimum_cost(cost)
17: return S
```

as energy efficiency or SLO.

5.5.3 Engine Compiler

Once every node has been assigned to an engine, Engine Compiler individually invokes the engine-specific compiler to obtain the engine executable. The canonical set of operations in engine executables constitutes loading the input data to engines, setting the configuration registers, triggering the computation, observing the runtime status, and receiving the output data. The underlying implementations of these operations for accelerators are all disparate, which makes the runtime orchestration difficult. To unify the interfaces, **XLVM** abstracts the engines as files that can perform computation and formalizes the engine interfaces as a set of file management APIs. Similar to the Unix I/O, these APIs include (1) open a new engine, (2) read data back from the engine, (3) write data to the engine, (4) initiate compute of a capability, and (5) close the engine. Thus, to link this *computational* file abstraction with the low-level hardware interfaces, the engine developers are asked to provide engine-specific device drivers.

Table 5.1. Cross-domain benchmark suite.

Name	Description	Domains	Used Capabilities
memory-enhance	Deep Brain Stimulation closed-loop control pipeline to optimize stimulation signals for memory enhancement	DSP	FFT
		Data Analytics	LR
		Optimized Control	MPC
robot-explorer	KinectFusion for 3D map generation with model predictive control for cave exploration	Robotics	KF
		Computer Vision	MPC
video-sync	Calculate correct offset to sync a movie and subtitle file	DSP	MPEG-Decode FFT
stock-market	Text sentiment classification on stock market news articles to estimate call option price	Data Analytics	LR
		Finance	Black-Scholes
leukocyte	Detect and tracks rolling leukocytes (white blood cells) in a microscopy of blood vessels vivo video	Computer Vision	GICOV MGVF
security-camera	Real-time object detection system which decodes MPEG encoded video	Deep Learning	Tiny-Yolo-v2
		DSP	MPEG-Decode

5.6 Evaluation

5.6.1 Experimental Setup

Benchmarks.

Cross-domain multi-acceleration is an emerging field and there is a lack of established workloads that span multiple domains. We take real-life applications comprising well known algorithms to create a benchmark suite that can evaluate cross-domain multi-acceleration. Table 5.1 summarizes these benchmarks, the domains they contain, and the accelerated kernels: (1) memory-enhance is the deep brain stimulation introduced in Section 5.3.2; (2) robot-explorer is a four-wheeled robot equipped with a Kinect sensor to find its way through a cave and requires a KinectFusion (KF) algorithm to reconstruct a 3D map of the cave and MPC algorithm to navigate

Table 5.2. Domains and engines used in the evaluation.

Domain	Capabilities	Engine	Platform
DSP	FFT, MPEG-Decode	FFTW [48]	CPU
		ffmpeg [55]	CPU
		DeCO [46]	FPGA
		LogiCore [59]	FPGA
Data Analytics	LR	MLPack [56]	CPU
		InAccel [60]	FPGA
		Tabla [36]	FPGA
Robotics	MPC	ACADO [68]	CPU
		RoboX [19]	FPGA
Computer Vision	GICOV, MGVF, KF	OpenCV [53]	CPU
		SLAMBench [54]	FPGA
		Iron [62]	FPGA
Finance	Black-Scholes	QuantLib [57]	CPU
		HyperStreams [58]	FPGA
Deep Learning	Tiny-Yolo-v2	TensorFlow [44]	CPU
		TVM [45]	CPU
		DnnWeaver [16]	FPGA

through the cave; (3) video-sync synchronizes subtitles with speech segments for video files, and requires MPEG-decoding and FFT to boost the speech-text pattern matching process; (4) stock-market predicts the call option price in the stock market, and requires sentiment analysis using LR on news articles to extract market signals with Black-Scholes model to predict call option pricing; (5) leukocyte detects leukocytes from video microscopy of blood vessels, and uses Gradient Inverse Coefficient of Variation (GICOV) score to perform detection in the frame and Motion Gradient Vector Flow (MGVF) matrix to track the leukocytes; (6) security-camera detects suspicious objects from its input video stream by decoding the MPEG encoded video stream using MPEG-Decoding and performing an object detection using deep learning (Tiny-Yolo-v2).

Compute platforms.

Table 5.2 summarizes the domains used in the benchmarks, the accelerated capabilities, the used engines, and their platforms. Our system is equipped with a host CPU, Intel Xeon E3 (3.50 GHz). For fair comparison, we use optimized software libraries to obtain the best performance, including Intel MKL 2020, OpenBlas v0.3, and OpenCV 3.4.2. For FPGA, we use Xilinx KCU1500 with open-source hardware accelerators [110, 125]. Accelerators are attached to the host via a PCIe interconnect.

Runtime measurements.

We run the experiments for ten times and attain the average to report. When open-source RTL implementations of existing FPGA accelerators are unavailable, we use the author-provided simulators to measure the performance. Using the kernel execution time on the platforms, we estimate the end-to-end application runtime.

Energy measurements.

To measure the energy consumption, we use the Intel Running Average Power Limit (RAPL) for CPU and use the simulators for the FPGA accelerators.

Programming effort.

Accurately measuring human effort is impossible, yet similar to prior works, FlexJava and EnerJ, we count the number of lines of code (LoC) and the number of annotations to quantify the human effort.

5.6.2 Experimental Results

Performance improvement.

Figure 5.9 shows the speedup gains as the number of accelerator engines increases compared to the CPU-only baseline. All the benchmarks provide benefits even from a single-engine acceleration, which yields a 12.0× speedup when the best-performing engine is used for accelerating the benchmarks. However, the results show that the speedup increases to 29.4× on

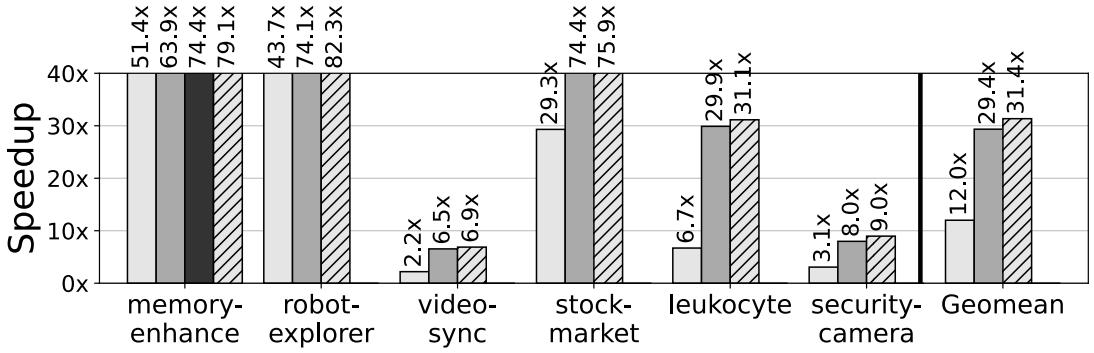


Figure 5.9. Speedup with various number of accelerator engines against CPU baseline.

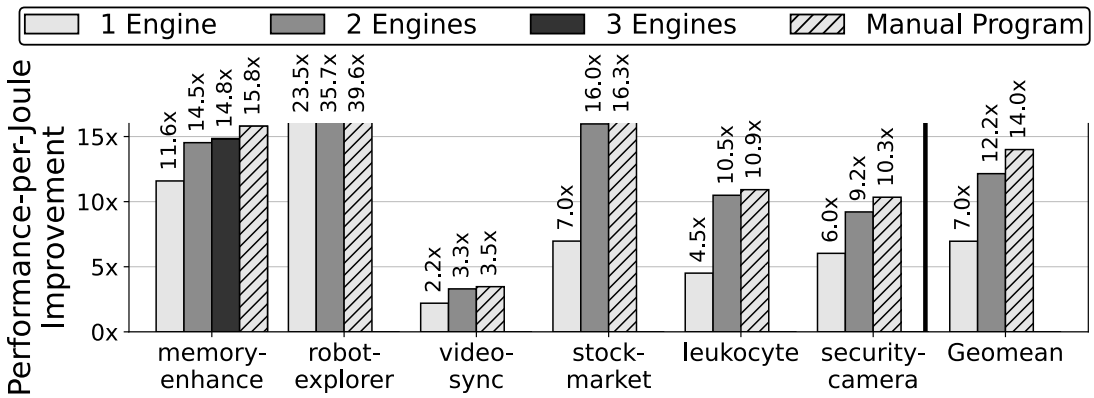


Figure 5.10. Performance-per-Joule improvement achieved by multi-acceleration.

average when leveraging more engines, which amounts to a 145% extra speedup. Thus, there is untapped potential in accelerating multiple kernels, which is unleashed by our dual abstractions and **XLVM**. The rightmost bar ("Manual Program") also shows the speedup when the maximal number of accelerators are enabled manually by programmers, which represent the ideal speedup that **Yin-Yang** would be able to achieve. The results show that **Yin-Yang** almost reaches this ideal speedup, while requiring less programming effort. Overall, our system attests to the common wisdom that "the more accelerators, the better".

Performance-per-Joule improvement.

Figure 5.10 illustrates the performance-per-Joule improvement of multi-acceleration over the CPU baseline. As the figure shows, acceleration using a single engine achieves an overall Performance-per-Joule improvement of $7.0\times$ against the baseline. By leveraging more engines

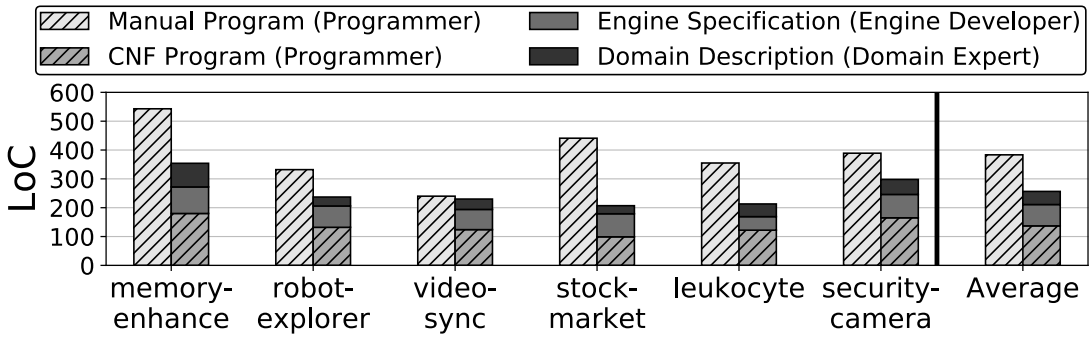


Figure 5.11. LoC improvements of **Yin-Yang** in comparison with the baseline manual programming.

through the **Yin-Yang** abstractions, we can achieve higher performance-per-Joule improvements of 12.2×, which is translated to 74.2% extra efficiency. Similar to the performance we observed above, we also report the manual multi-acceleration result, which shows that **Yin-Yang** closely reaches to the ideal efficiency gain, only leaving a marginal room for improvement.

Programmability.

Figure 5.11 shows Lines of Code (LoC) improvements of the dual abstractions when compared to manual programming. The bar on the left represents LoC that programmers write, while the stacked bar on the right delineates the summated LoC that programmers, domain experts, and engine developers should write in aggregate. The results show that **Yin-Yang** effectively reduces the LoC by 33.1% on average while obtaining the same functionality and performance. The human efforts needed for domain descriptions and engine specifications is imposed only once when registering the domains and engines. From the programmers perspective, the LoC is reduced from 383 to 137, which increases the reduction rate to 64.2%. These results suggest that the proposed dual abstractions allow domain experts and engine developers to take part in enabling multi-acceleration with minimum effort, and CNF emancipates application programmers from the onerous task of hardware development and low-level programming for orchestrating multiple accelerators.

5.7 Related Works

Abstractions for heterogenous platforms.

Although various general purpose abstractions for accelerators such as OpenCL exist, they do not incorporate the algorithmic domain knowledge. Intel oneAPI [82] provides libraries and compilation tools that can target multiple accelerators. The libraries of oneAPI contain fine-grained constructs that allow programmers to focus on their domain of interest and optimize it. SysML [57] is a system architecture modeling tool built upon Unified Modeling Language (UML). While SysML has a similarity to our CNF programming model, it offers a general-purpose and unified abstraction that lacks the notion of domains. In contrast, this work provides *cross-domain* programming abstractions and necessary mechanisms to make it easy for programmers to harmoniously combine existing accelerators from different domains together to develop a single application.

Domain-specific abstractions.

There are a plethora of one-sided acceleration solutions [83, 147] for a single domain, which is either *algorithm-centric* or *hardware-centric*. Our approach differs from these works in providing dual abstractions that move away from one-sided representation of a single domain and links multiple domains. This enables us to utilize disjointly pre-designed accelerators to be used in tandem for cross-domain applications.

FPGA acceleration.

High-Level Synthesis (HLS) is an effective tool that allows programmers to use a high-level language for accelerator development. While HLS improves programmability, its performance gains are usually lower than the custom-designed accelerators, as shown in prior works [32]. In contrary, **Yin-Yang** is an alternative programming tool that offers three different abstractions for three parties–(1) domain experts, (2) engine developers, and (3) application programmers, which allow them to collaboratively enable multi-acceleration for cross-domain applications.

5.8 Conclusion

Cross-domain multi-acceleration can unlock new capabilities. For this emerging direction, we uniquely provide dual abstractions which preserve domain knowledge while linking algorithmic representations to hardware capabilities. As a mechanism to provide this linkage, we develop **XLVM**, which represents the program as QF-DFG and determines efficient engine-to-capability mappings. Experimental results using a real-life benchmark suite show significant improvements in performance and energy when multiple accelerators from different domain are used. This paper also provides an open-source benchmark suite for the emerging area of cross-domain multi-acceleration, which is available at https://github.com/he-actlab/cross-domain-benchmarks.

5.9 Future Directions

To cope with the fact that the end-to-end intelligent systems consists of algorithms from more domains beyond just DNNs. **Yin-Yang** [95] expands the scope beyond accelerating DNNs to accelerating the end-to-end applications that constitute intelligent systems. This work lays the foundation for further investigations into AI-enabled compilation of intelligent systems.

AI-enabled compilation for intelligent systems.

To this end, exploring the use of various AI algorithms to optimize the execution of end-to-end applications will be an interesting research. For example, finding optimal solutions in the exponentially large search spaces for the compilation and execution becomes more pronounced for end-to-end applications. A promising starting point will be the extension of the works introduced in Chapter 2–4. Then, as the AI algorithms constantly evolve to be able to solve more problems, more effectively, and in a more scalable manner, it would be timely to adopt new algorithms to improve cross-domain multi-acceleration.

Acknowledgement. Chapter 5, in part, contains a re-organized reprint of the material as it appears in IEEE Micro 2022. Kim, Joon Kyung; Ahn, Byung Hoon; Kinzer, Sean; Ghodrati,

Soroush; Mahapatra, Rohan; Yatham, Brahmendra; Wang, Shu-Ting; Kim, Dohee; Sarikhani, Parisa; Mahmoudi, Babak; Mahajan, Divya; Park, Jongse; Esmaeilzadeh, Hadi. The dissertation author was the co-investigator and co-author of this paper.

Chapter 6

Other Works by This Author

While the main thrust of the dissertation focused on the *AI-Enabled Compilation for Intelligent Systems*, there are many other directions that are imperative to enabling machine intelligence. This section outlines some of the research problems explored including (1) multi-tenancy and (2) AI privacy.

Spatial multi-tenant acceleration of deep neural networks.

The main stream of the dissertation focused on how to leverage AI algorithms to find better configurations for the kernels, mainly focusing on tiling and loop unrolling. The AI-enabled compilation explored throughout the dissertation makes it possible to achieve near-maximum utilization of various accelerators for Deep Neural Networks (DNNs). However, without multi-tenancy, it is difficult to imagine leveraging cost-effective and scalable DNN services. In fact, the community has developed both software and hardware solutions to unlock multi-tenancy in CPUs and GPUs. To extend this multi-tasking capability to the deep learning accelerators, PREMA [39] and AI-MT [19] focused on temporal multi-tenant acceleration of deep neural networks. In Alternative Computing Technologies (ACT) Lab, we developed the very first spatial multi-tenant acceleration of deep neural networks dubbed **Planaria** [64] for better throughput, SLA satisfaction, and fairness. The work explored the topic of spatial multi-tenancy through a novel approach of dynamic architecture fission, and provided a concrete architecture, **Planaria**, and its respective scheduling algorithm.

Software-only solution for AI privacy.

Services such as voice assistants [18, 66], smart speakers [8], and many enterprise applications [10, 13, 67] consume a staggering amount of data. Much of the data are not necessarily required to perform the task, yet the data are consumed by the services un-encrypted. These sensitive and private information are high-bounty targets for the data thieves who actively seek to sniff and use these information against the users of the aforementioned services. While data is protected at rest and in motion through encryption [79, 141], it is exposed during inference as the data needs to be processed in an un-encrypted fashion due to latency requirements. For instance, Fully Homomorphic Encryption [63, 136, 184] and Secure Multiparty Computations [26, 122, 123, 183] slow down the AI services to yet preclude them from practical applications [116, 118]. To this end, at **Protopia AI**, we developed a software only solution called for AI privacy **Stained Glass Transform™** [6]. This solution utilizes curated stochastic perturbations to redact private information with minimal overhead as well as minimal changes to the original service.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *FPL*, 2018.

[3] Byung Hoon Ahn, Sean Kinzer, and Hadi Esmaeilzadeh. Glimpse: mathematical embedding of hardware specification for neural compilation. In *DAC*, 2022.

[4] Byung Hoon Ahn, Jinwon Lee, Jamie Menjay Lin, Hsin-Pai Cheng, Jilei Hou, and Hadi Esmaeilzadeh. Ordering chaos: Memory-aware scheduling of irregularly wired neural networks for edge devices. In *MLSys*, 2020.

[5] Byung Hoon Ahn, Prannoy Pilligundla, Amir Yazdanbakhsh, and Hadi Esmaeilzadeh. Chameleon: Adaptive code optimization for expedited deep neural network compilation. In *ICLR*, 2020.

[6] Byung Hoon Ahn, DoangJoo Synn, and Masih Derkani Eiman Ebrahimi Hadi Esmaeilzadeh. Protopia AI: Taking on the missing link in AI privacy and data protection. In *NeurIPS Demonstrations*, 2021.

[7] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch SGD: training ResNet-50 on ImageNet in 15 minutes. *arXiv*, 2017.

[8] Amazon. Amazon alexa. https://developer.amazon.com/en-US/alexa/.

[9] Amazon. Amazon astro. https://www.amazon.com/Introducing-Amazon-Astro/dp/B078NSDFSB.

[10] Amazon. Amazon case studies. https://aws.amazon.com/solutions/case-studies/.

[11] Amazon. Amazon inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[12] Amazon. Amazon ring. https://www.amazon.com/stores/Ring/Ring/page/77B53039-540E-4816-BABB-49AA21285FCF.

[13] Amazon. Amazon sagemaker customers. https://aws.amazon.com/sagemaker/customers/.

[14] Amazon. Amazon trainium. https://aws.amazon.com/machine-learning/trainium/.

[15] Amazon. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/, 2017.

[16] Anonymous. Robust scheduling with GFlowNets. In *ICLR (Under Review)*, 2023.

[17] Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. Structured pruning of deep convolutional neural networks. *JETC*, 2017.

[18] Apple. Apple siri. https://www.apple.com/siri/.

[19] Eunjin Baek, Dongup Kwon, and Jangwoo Kim. A multi-neural network acceleration architecture. In *ISCA*, 2020.

[20] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966.

[21] Richard Bellman. Dynamic programming. *Science*, 1966.

[22] Richard Ernest Bellman. Dynamic programming treatment of the traveling salesman problem. *Journal of the ACM*, 1961.

[23] Emmanuel Bengio, Moksh Jain, Maksym Korablyov, Doina Precup, and Yoshua Bengio. Flow network based generative models for non-iterative diverse candidate generation. *NeurIPS*, 2021.

[24] David Bernstein, Michael Rodeh, and Izidor Gertner. On the complexity of scheduling problems for parallel/pipelined machines. *TC*, 1989.

[25] Ondřej Bojar, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. Findings of the 2014 workshop on statistical machine translation. In *WMT*, 2014.

[26] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, 2017.

[27] John Bruno and Ravi Sethi. Code generation for a one-register machine. *Journal of the ACM*, 1976.

[28] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. *ICLR*, 2020.

[29] Han Cai, Ligeng Zhu, and Song Han. ProxylessNAS: Direct neural architecture search on target task and hardware. In *ICLR*, 2019.

[30] Wenbin Cai, Ya Zhang, and Jun Zhou. Maximizing expected model change for active learning in regression. In *ICDM*, 2013.

[31] Pohua P Chang, Scott A Mahlke, and Wen-Mei W Hwu. Using profile information to assist classic code optimizations. *Software: Practice and Experience*, 1991.

[32] Sung-En Chang, Yanyu Li, Mengshu Sun, Runbin Shi, Hayden K-H So, Xuehai Qian, Yanzhi Wang, and Xue Lin. Mix and match: A novel FPGA-centric deep neural network quantization framework. In *HPCA*, 2021.

[33] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *KDD*, 2016.

[34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.

[35] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *NeurIPS*, 2018.

[36] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *JSSC*, 2016.

[37] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *MICRO*, 2014.

[38] Hsin-Pai Cheng, Tunhou Zhang, Yukun Yang, Feng Yan, Shiyu Li, Harris Teague, Hai Helen Li, and Yiran Chen. SwiftNet: Using graph propagation as meta-knowledge to search

highly representative neural architectures. *arXiv*, 2019.

[39] Yujeong Choi and Minsoo Rhu. Prema: A predictive multi-task scheduling algorithm for preemptible neural processing units. In *HPCA*, 2020.

[40] Valeriu Codreanu, Damian Podareanu, and Vikram Saletore. Achieving deep learning training in less than 40 minutes on ImageNet-1K & best accuracy and training time on ImageNet-22K & Places-365 with scale-out Intel® Xeon®/Xeon Phi™ architectures, 2017.

[41] David A Cohn, Zoubin Ghahramani, and Michael I Jordan. Active learning with statistical models. *JAIR*, 1996.

[42] Dorin Comaniciu and Peter Meer. Mean shift: A robust approach toward feature space analysis. *TPAMI*, 2002.

[43] Corinna Cortes, Xavier Gonzalvo, Vitaly Kuznetsov, Mehryar Mohri, and Scott Yang. AdaNet: Adaptive structural learning of artificial neural networks. In *ICML*, 2017.

[44] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv*, 2016.

[45] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An intermediate representation, compiler, and executor for deep learning. *arXiv*, 2018.

[46] Jeff Dean. Machine learning for systems and systems for machine learning. In *NIPS Workshop on ML Systems*, 2017.

[47] Deloitte. Smart factory for smart manufacturing. https://www2.deloitte.com/us/en/pages/consulting/solutions/the-smart-factory.html.

[48] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[49] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *JSSC*, 1974.

[50] Ahmed T Elthakeb, Prannoy Pilligundla, Fatemehsadat Mireshghallah, Amir Yazdan-

bakhsh, and Hadi Esmaeilzadeh. ReLeQ: A reinforcement learning approach for deep quantization of neural networks. *IEEE Micro*, 2020.

[51] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[52] Steven K. Esser, Jeffrey L. McKinstry, Deepika Bablani, Rathinakumar Appuswamy, and Dharmendra S. Modha. Learned step size quantization. In *ICLR*, 2020.

[53] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[54] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning. In *NIPS*, 2015.

[55] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *ICML*, 2017.

[56] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale DNN processor for real-time AI. In *ISCA*, 2018.

[57] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A practical guide to SysML: the systems modeling language*. Morgan Kaufmann, 2014.

[58] Matteo Frigo and Steven G Johnson. FFTW: An adaptive software architecture for the FFT. In *ICASSP*, 1998.

[59] Mukul Gagrani, Corrado Rainone, Yang Yang, Harris Teague, Wonseok Jeon, Herke Van Hoof, Weiliang Will Zeng, Piero Zappi, Christopher Lott, and Roberto Bondesan. Neural topological ordering for computation graphs. *NeurIPS*, 2022.

[60] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *ASPLOS*, 2017.

[61] Yuanxiang Gao, Li Chen, and Baochun Li. Post: Device placement with cross-entropy minimization and proximal policy optimization. In *NeurIPS*, 2018.

[62] Kent Gauen, Rohit Rangan, Anup Mohan, Yung-Hsiang Lu, Wei Liu, and Alexander C Berg. Low-power image recognition challenge. In *ASP-DAC*, 2017.

[63] Craig Gentry. *A fully homomorphic encryption scheme*. Stanford university, 2009.

[64] Soroush Ghodrati, Byung Hoon Ahn, Joon Kyung Kim, Sean Kinzer, Brahmendra Reddy Yatham, Navateja Alla, Hardik Sharma, Mohammad Alian, Eiman Ebrahimi, Nam Sung Kim, Cliff Young, and Hadi Esmaeilzadeh. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks. In *MICRO*, 2020.

[65] Jack Goetz, Ambuj Tewari, and Paul Zimmerman. Active learning for non-parametric regression using purely random trees. In *NeurIPS*, 2018.

[66] Google. Google assistant. https://assistant.google.com/.

[67] Google. Google cloud customers. https://cloud.google.com/customers/.

[68] Google. Google translate. https://translate.google.com/.

[69] Google. TensorFlow Lite. https://www.tensorflow.org/mobile/tflite.

[70] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv*, 2017.

[71] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. In *ICLR*, 2017.

[72] 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 *ISCA*, 2016.

[73] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *ICLR*, 2016.

[74] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *NIPS*, 2015.

[75] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[76] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. AMC: AutoML for model compression and acceleration on mobile devices. In *ECCV*, 2018.

[77] Michael Held and Richard M Karp. A dynamic programming approach to sequencing problems. *Journal of the SIAM*, 1962.

[78] John L Hennessy and David A Patterson. A new golden age for computer architecture.

CACM and Turing Lecture, 2019.

[79] Simon Heron. Advanced encryption standard (AES). *Network Security*, 2009.

[80] Mark D Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CACM*, 2021.

[81] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient convolutional neural networks for mobile vision applications. *arXiv*, 2017.

[82] Intel. Intel oneAPI: Programmable inference accelerator. https://software.intel.com/content/www/us/en/develop/tools/oneapi.html, 2019.

[83] Abhishek Kumar Jain, Xiangwei Li, Pranjul Singhai, Douglas L Maskell, and Suhaib A Fahmy. DeCO: A DSP block based FPGA accelerator overlay with low overhead interconnect. In *FCCM*, 2016.

[84] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, 2018.

[85] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *MM*, 2014.

[86] Zhihao Jia, Sina Lin, Charles R Qi, and Alex Aiken. Exploring hidden dimensions in parallelizing convolutional neural networks. In *ICML*, 2018.

[87] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. Optimizing DNN computation with relaxed graph substitutions. In *SysML*, 2019.

[88] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *ISCA*,

2017.

[89] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *MICRO*, 2016.

[90] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andrew J Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. *Nature*, 2021.

[91] Arthur B Kahn. Topological sorting of large networks. *CACM*, 1962.

[92] Ken Kennedy and John R Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., 2001.

[93] Christoph Keßler and Andrzej Bednarski. A dynamic programming approach to optimal integrated code generation. In *LCTES*, 2001.

[94] Shauharda Khadka, Estelle Aflalo, Mattias Marder, Avrech Ben-David, Santiago Miret, Shie Mannor, Tamir Hazan, Hanlin Tang, and Somdeb Majumdar. Optimizing memory placement using evolutionary graph reinforcement learning. *ICLR*, 2021.

[95] Joon Kyung Kim, Byung Hoon Ahn, Sean Kinzer, Soroush Ghodrati, Rohan Mahapatra, Brahmendra Yatham, Shu-Ting Wang, Dohee Kim, Parisa Sarikhani, Babak Mahmoudi, Divya Mahajan, Jongse Park, and Hadi Esmaeilzadeh. Yin-Yang: Programming abstractions for cross-domain multi-acceleration. In *IEEE Micro*, 2022.

[96] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *NIPS*, 2012.

[97] F Künhe, J Gomes, and W Fetter. Mobile robot trajectory tracking using model predictive control. In *LARS*, 2005.

[98] David Laredo, Yulin Qin, Oliver Schütze, and Jian-Qiao Sun. Automatic model selection for neural networks. *arXiv*, 2019.

[99] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[100] Yann LeCun. Deep learning hardware: Past, present, and future. In *ISSCC*, 2019.

[101] Yann LeCun, John S Denker, and Sara A Solla. Optimal brain damage. In *NIPS*, 1990.

[102] Chingren Lee, Jenq Kuen Lee, Tingting Hwang, and Shi-Chun Tsai. Compiler optimization on vliw instruction scheduling for low power. *TODAES*, 2003.

[103] Menghao Li, Minjia Zhang, Chi Wang, and Mingqin Li. AdaTune: Adaptive tensor program compilation made efficient. In *NeurIPS*, 2020.

[104] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The deep learning compiler: A comprehensive survey. In *TPDS*, 2020.

[105] Alexander Liniger, Alexander Domahidi, and Manfred Morari. Optimization-based autonomous racing of 1: 43 scale rc cars. *Optimal Control Applications and Methods*, 2015.

[106] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: Differentiable architecture search. In *ICLR*, 2019.

[107] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing CNN model inference on CPUs. In *USENIX ATC*, 2019.

[108] Steve Lohr. The age of big data. *New York Times*, 2012.

[109] Mohammad Loni, Ali Zoljodi, Amin Majd, Byung Hoon Ahn, Masoud Daneshtalab, Mikael Sjödin, and Hadi Esmaeilzadeh. Faststereonet: A fast neural architecture search for improving the inference of disparity estimation on resource-limited platforms. *IEEE TSMC*, 2021.

[110] Divya Mahajan, Jongse Park, Emmanuel Amaro, Hardik Sharma, Amir Yazdanbakhsh, Joon Kyung Kim, and Hadi Esmaeilzadeh. Tabla: A unified template-based framework for accelerating statistical machine learning. In *HPCA*, 2016.

[111] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *HotNets*, 2016.

[112] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *SIGCOMM*, 2019.

[113] Kim Marriott, Peter J Stuckey, and Peter J Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[114] Peter Mattson, Christine Cheng, Gregory Diamos, Cody Coleman, Paulius Micikevicius,

David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, David Brooks, Dehao Chen, Debo Dutta, Udit Gupta, Kim Hazelwood, Andy Hock, Xinyuan Huang, Daniel Kang, David Kanter, Naveen Kumar, Jeffery Liao, Deepak Narayanan, Tayo Oguntebi, Gennady Pekhimenko, Lillian Pentecost, Vijay Janapa Reddi, Taylor Robie, Tom St John, Carole-Jean Wu, Lingjie Xu, Cliff Young, and Matei Zaharia. MLPerf training benchmark. *MLSys*, 2020.

[115] Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. In *CACM*, 2021.

[116] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Ali Jalali, Ahmed Taha Taha Elthakeb, Dean Tullsen, and Hadi Esmaeilzadeh. Not all features are equal: Discovering essential features for preserving prediction privacy. In *WWW*, 2021.

[117] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. Shredder: Learning noise distributions to protect inference privacy. In *ASPLOS*, 2020.

[118] Fatemehsadat Mireshghallah, Mohammadkazem Taram, Prakash Ramrakhyani, Ali Jalali, Dean Tullsen, and Hadi Esmaeilzadeh. Shredder: Learning noise distributions to protect inference privacy. In *ASPLOS*, 2020.

[119] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. In *ICLR*, 2018.

[120] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *ICML*, 2017.

[121] Asit Mishra and Debbie Marr. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. In *ICLR*, 2018.

[122] Payman Mohassel and Peter Rindal. Aby3: A mixed protocol framework for machine learning. In *CCS*, 2018.

[123] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *S&P*, 2017.

[124] Jiandong Mu, Mengdi Wang, Lanbo Li, Jun Yang, Wei Lin, and Wei Zhang. A history-based auto-tuning framework for fast and high-performance DNN design on GPU. In *DAC*, 2020.

[125] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly,

Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham Riley, Nigel Topham, and Steve Furber. Introducing slambench, a performance and accuracy benchmarking methodology for slam. In *ICRA*, 2015.

[126] Alexander Nareyek. Choosing search heuristics by non-stationary reinforcement learning. In *Metaheuristics: Computer Decision-Making*. Springer, 2003.

[127] Renato Negrinho and Geoff Gordon. DeepArchitect: Automatically designing and training deep architectures. *arXiv*, 2017.

[128] Diego Novillo. SamplePGO - the power of profile guided optimizations without the usability burden. In *LLVM Compiler Infrastructure in HPC*, 2014.

[129] NVIDIA. TensorRT: Programmable inference accelerator. https://developer.nvidia.com/tensorrt, 2017.

[130] NVIDIA. List of nvidia graphics processing units, since 1993.

[131] Jack O'Neill, Sarah Jane Delany, and Brian MacNamee. Model-free and model-based active learning for regression. In *Advances in Computational Intelligence Systems*. Springer, 2017.

[132] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. Reinforced genetic algorithm learning for optimizing computation graphs. In *ICLR*, 2020.

[133] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *ICASSP*, 2015.

[134] 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 *ISCA*, 2017.

[135] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

[136] Le Trieu Phong, Yoshinori Aono, Takuya Hayashi, Lihua Wang, and Shiho Moriai. Privacy-preserving deep learning via additively homomorphic encryption. *TIFS*, 2017.

[137] Yury Pisarchyk and Juhyun Lee. Efficient memory management for deep neural net inference. *MLSys Workshop on Resource-Constrained Machine Learning*, 2020.

[138] Detlef Plump. Term graph rewriting. In *Handbook Of Graph Grammars And Computing By Graph Transformation: Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[139] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. In *CVPR*, 2020.

[140] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2019.

[141] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, 2018.

[142] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[143] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. Glow: Graph lowering compiler techniques for neural networks. *arXiv*, 2018.

[144] Jaehun Ryu, Eunhyeok Park, and Hyojin Sung. One-shot tuner for deep learning compilers. In *CC*, 2022.

[145] Jaehun Ryu and Hyojin Sung. MetaTune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. In *arXiv*, 2021.

[146] Amit Sabne. Xla : Compiling machine learning for peak performance, 2020.

[147] Jacob Sacks, Divya Mahajan, Richard C Lawson, and Hadi Esmaeilzadeh. Robox: an end-to-end solution to accelerate autonomous control in robotics. In *ISCA*, 2018.

[148] Samsung. Samsung bixby. https://www.samsung.com/us/apps/bixby/.

[149] Samsung. Samsung bot care. https://research.samsung.com/robot.

[150] Parisa Sarikhani, Svjetlana Miocinovic, and Babak Mahmoudi. Towards automated patient-specific optimization of deep brain stimulation for movement disorders. In *EMBC*, 2019.

[151] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 1997.

[152] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, 2013.

[153] Andreas Schösser and Rubino Geiß. Graph rewriting for hardware dependent program optimizations. In *AGTIVE*, 2007.

[154] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv*, 2017.

[155] Taro Sekiyama, Takashi Imamichi, Haruki Imai, and Rudy Raymond. Profile-guided memory optimization for deep neural networks. *arXiv*, 2018.

[156] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[157] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Vikas Chandra, and Hadi Esmaeilzadeh. Bit Fusion: Bit-level dynamically composable architecture for accelerating deep neural networks. In *ISCA*, 2018.

[158] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *MICRO*, 2019.

[159] Laurent Sifre and Stéphane Mallat. Rigid-motion scattering for image classification. *Ph.D. dissertation*, 2014.

[160] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. In *Nature*, 2016.

[161] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *ICLR*, 2015.

[162] Richard M Stallman and GCC DeveloperCommunity. *Using the GNU compiler collection: a GNU manual for GCC version 4.3.3*. CreateSpace, 2009.

[163] Masashi Sugiyama. Active learning in approximately linear regression based on conditional expectation of generalization error. *JMLR*, 2006.

[164] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset. In *CVPR*, 2020.

[165] Qi Sun, Chen Bai, Tinghuan Chen, Hao Geng, Xinyun Zhang, Yang Bai, and Bei Yu. Fast

and efficient DNN deployment via deep gaussian transfer learning. In *ICCV*, 2021.

[166] Qi Sun, Chen Bai, Hao Geng, and Bei Yu. Deep neural network hardware deployment optimization via advanced active learning. In *DATE*, 2021.

[167] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, 2015.

[168] Mingxing Tan, Ruoming Pang, and Quoc V Le. EfficientDet: Scalable and efficient object detection. In *CVPR*, 2020.

[169] Tesla. Tesla AI Day, 2021.

[170] 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*, 2018.

[171] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NeurIPS*, 2017.

[172] Michael Volpp, Lukas P Fröhlich, Kirsten Fischer, Andreas Doerr, Stefan Falkner, Frank Hutter, and Christian Daniel. Meta-learning acquisition functions for transfer learning in bayesian optimization. In *ICLR*, 2020.

[173] Hanrui Wang, Zhanghao Wu, Zhijian Liu, Han Cai, Ligeng Zhu, Chuang Gan, and Song Han. Hat: Hardware-aware transformers for efficient natural language processing. *ACL*, 2020.

[174] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. HAQ: Hardware-aware automated quantization with mixed precision. In *CVPR*, 2019.

[175] R Clinton Whaley and Jack J Dongarra. Automatically tuned linear algebra software. In *SC*, 1998.

[176] Kent Wilken, Jack Liu, and Mark Heffernan. Optimal instruction scheduling using integer programming. In *PLDI*, 2000.

[177] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *ECCV*, 2018.

[178] Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. In *NeurIPS*, 2019.

[179] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, Tommer Leyvand, Hao Lu, Yang Lu, Lin Qiao, Brandon Reagen, Joe Spisak, Fei Sun, Andrew Tulloch, Peter Vajda, Xiaodong Wang, Yanghan Wang, Bram Wasti, Yiming Wu, Ran Xian, Sungjoo Yoo, and Peizhao Zhang. Machine learning at facebook: Understanding inference at the edge. In *HPCA*, 2019.

[180] Dongrui Wu, Chin-Teng Lin, and Jian Huang. Active learning for regression using greedy sampling. *Information Sciences*, 2019.

[181] Saining Xie, Alexander Kirillov, Ross Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *ICCV*, 2019.

[182] Zhongwen Xu, Hado P van Hasselt, and David Silver. Meta-gradient reinforcement learning. In *NeurIPS*, 2018.

[183] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *SFCS*, 1986.

[184] Ryo Yonetani, Vishnu Naresh Boddeti, Kris M Kitani, and Yoichi Sato. Privacy-preserving visual learning using doubly permuted homomorphic encryption. In *ICCV*, 2017.

[185] Yang You, Igor Gitman, and Boris Ginsburg. Large batch training of convolutional networks. *arXiv*, 2017.

[186] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet training in minutes. In *ICPP*, 2018.

[187] Hwanjo Yu and Sungchul Kim. Passive sampling for regression. In *ICDM*, 2010.

[188] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *BMVC*, 2016.

[189] Daniel Zhang, Saurabh Mishra, Erik Brynjolfsson, John Etchemendy, Deep Ganguli, Barbara J. Grosz, Terah Lyons, James Manyika, Juan Carlos Niebles, Michael Sellitto, Yoav Shoham, Jack Clark, and C. Raymond Perrault. The AI index 2021 annual report. *arXiv*, 2021.

[190] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal data placement for heterogeneous cache, memory, and storage systems. *POMACS*, 2020.

[191] Minjia Zhang, Menghao Li, Chi Wang, and Mingqin Li. DynaTune: Dynamic tensor program optimization in deep neural network compilation. In *ICLR*, 2021.

[192] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. TenSet: A large-scale program performance dataset for learned tensor

compilers. In *NeurIPS Track on Datasets and Benchmarks*, 2021.

[193] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv*, 2016.

[194] Michael Zhu and Suyog Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. In *ICLR Workshop*, 2018.

[195] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017.

[196] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.