

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Michael Canesche

Optimizing Machine Learning Models: A Droplet Search Approach to
Efficient Kernel Scheduling

Belo Horizonte
2024

Michael Canesche

**Optimizing Machine Learning Models: A Droplet Search Approach to
Efficient Kernel Scheduling**

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte
2024

Canesche, Michael.

C221o Optimizing machine learning models: [recurso eletrônico] a droplet search approach to efficient kernel scheduling / Michael Canesche – 2024.

1 recurso online (141 f. il, color.) : pdf.

Orientador: Fernando Magno Quintão Pereira.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.

Referências: f. 124-141

1. Computação – Teses. 2. Compiladores (Programas de computador) – Teses. 2. Aprendizado de computador – Teses. 3. Otimização combinatoria – Teses. 4. Recuperação da informação – Teses. I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III. Título.

CDU 519.6*21(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Optimizing Machine Learning Models: A Droplet Search Approach to
Efficient Kernel Scheduling

MICHAEL CANESCHE

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

Fernando M. Q. Pereira

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

Marcio Costa Santos

PROF. MÁRCIO COSTA SANTOS
Departamento de Ciência da Computação - UFMG

Corinne Ancourt

PROFA. CORINNE ANCOURT
MINES Paris - École des Mines de Paris

Sanjay Rajopadhye

PROF. SANJAY RAJOPADHYE
Electrical and Computer Engineering Department - Colorado State University

P. Sadayappan

PROF. PONNUSWAMY SADAYAPPAN
School of Computing - University of Utah

Belo Horizonte, 15 de novembro de 2024.

*To my laboratory friends. Living life's constant challenge of
investigating the universe.*

Acknowledgments

I am very thankful...

To my advisor Fernando for accepting me as PhD student and allowing me to realize my dream of studying at the Federal University of Minas Gerais (UFMG). His friendship and partnership through all these years have been invaluable for my growth as a person, student, and researcher.

To my friends in the compiler's laboratory, for the enriching discussions on this thesis, computers, and life in general. In particular, I would like to thank (in alphabetical order) Angélica Moreira, Bruno Kind, Breno Campos, Caio Raposo, Cecília Kind, Daniel Augusto, Everaldo Andrade, Guilherme Silva, João Amorin, João Fröhlich, Kael Augusto, Kaio Ananias, Lucas Victor, Luigi Domenico, Luiza Melo, Rafael Alvarenga, Rafael Sumitani, Roberto Rosmaninho, Thaís Damásio, Vinicius Julião, Vinicius Pacheco, Vinicius Silva. You all are truly awesome.

To my parents, Miguel and Maria, for their upbringing and unwavering support. Their unconditional support has been crucial for the accomplishment of every milestone in my life, including this one.

I am grateful to my brother Miguel and my sister-in-law Rosangela for their constant presence and support, even from afar, and for allowing me the joy of being Gabriel's uncle. Their smiles bring happiness to my life.

I am grateful to Unicamp University for the support and assistance provided throughout my journey. Especially Thaís Camacho, and Professor Edson Borin. Who helped me and contributed immensely with discussions and ideas for this work.

To the Cadence Company, I would like to express my gratitude for the financial support provided for my research. In particular, I want to extend my thanks to Vanderson and Eric for the numerous meetings and discussions about this work. I always learn a lot from both of them.

I extend my sincere appreciation to CAPES, FAPEMIG, and CNPq for their financial support, which has made the commencement of my PhD journey possible. Without this assistance, embarking on this path would not have been feasible.

In special, I would like to thank all the people in Brazil. The research is only possible with their contribution to this country. Thank you very much!

“Do what you like in life, not what you feel you should do.”
(Jim Simons)

Resumo

O escalonamento do kernel, um estágio crítico da otimização computacional, envolve a tarefa de determinar a implementação mais eficiente para um kernel computacional. Essa jornada de otimização envolve experimentar parâmetros do compilador, como tamanhos de janelas lado a lado e fatores de desenrolamento, para discernir a configuração ideal. A abordagem introduzida neste trabalho conceitua esses parâmetros como pontos dentro de um espaço de coordenadas cuidadosamente definido. Embora a função que mapeia esses pontos para os tempos de execução do kernel normalmente resulte em uma superfície não convexa, a evidência empírica apoia uma nova hipótese denominada “droplet expectation”. Esta hipótese postula que a origem desta superfície, representando um kernel não otimizado, e seu ótimo global, significando o kernel mais rápido, residem coletivamente dentro dos limites de uma região convexa. Aproveitando esse insight, o trabalho propõe a metodologia Droplet Search, baseada no algoritmo de descida por coordenadas. Assumindo a validade da expectativa do droplet, o Droplet Search demonstra a capacidade de convergir rapidamente para a configuração ideal do kernel. Implementado no Apache TVM desde abril de 2023, o Droplet Search passa por uma avaliação rigorosa por meio de extensa experimentação envolvendo seis grandes modelos de aprendizagem profunda implantados em uma variedade de dispositivos de computação, incluindo ARM, Intel, AMD e NVIDIA. Os resultados obtidos desses experimentos mostram que o Droplet Search não apenas corresponde à eficácia de outras técnicas de pesquisa do AutoTVM, mas também as supera em eficiência, apresentando uma melhoria de velocidade que varia entre duas a dez vezes. Além de seus recursos de otimização acelerada, os modelos gerados pelo Droplet Search exibem uma vantagem competitiva quando justapostos aos produzidos pelo AutoScheduler (Anso) do TVM. Esta é uma revelação digna de nota, considerando que o Anso emprega quatro a cinco vezes mais transformações de código do que o AutoTVM. As descobertas empíricas aqui apresentadas ressaltam a trajetória promissora do Droplet Search como uma ferramenta potente no domínio do agendamento do kernel, redefinindo benchmarks ao oferecer processos de otimização acelerados e resultados competitivos, tudo realizado com uma sobrecarga computacional visivelmente reduzida.

Palavras-chave: compilador de tensor; otimização; kernel scheduling; busca.

Abstract

Kernel scheduling, a critical stage of computational optimization, involves the intricate task of determining the most efficient implementation for a computational kernel. This optimization journey encompasses experimenting with compiler parameters, such as tiling window sizes and unrolling factors, to discern the optimal configuration. The approach introduced in this work conceptualizes these parameters as points within a carefully defined coordinate space. While the function mapping these points to kernel running times typically results in a non-convex surface, empirical evidence supports a novel hypothesis termed the “droplet expectation.” This hypothesis posits that the origin of this surface, representing an unoptimized kernel, and its global optimum, signifying the fastest kernel, collectively reside within the confines of a convex region. Leveraging this insight, the work proposes the Droplet Search methodology, grounded in the coordinate descent algorithm. Assuming the validity of the droplet expectation, Droplet Search demonstrates an ability to swiftly converge upon the optimal kernel configuration. Implemented in Apache TVM since April 2023, Droplet Search undergoes rigorous evaluation through extensive experimentation involving six large deep learning models deployed across a spectrum of computing devices, including ARM, Intel, AMD, and NVIDIA. Results gleaned from these experiments depict Droplet Search as not only matching the effectiveness of other AutoTVM search techniques but outperforming them with unparalleled efficiency, showcasing a speed improvement ranging between two to ten times. Beyond its accelerated optimization capabilities, models generated by Droplet Search exhibit a competitive edge when juxtaposed against those produced by TVM’s AutoScheduler (Ansor). This is a noteworthy revelation considering that Ansor employs four to five times more code transformations than AutoTVM. The empirical findings presented herein underscore the promising trajectory of Droplet Search as a potent tool in the domain of kernel scheduling, redefining benchmarks by offering accelerated optimization processes and competitive outcomes, all accomplished with a discernibly reduced computational overhead.

Keywords: tensor compiler; optimization; kernel scheduling; search.

List of Figures

1.1	Examples of search spaces defined by the same set of kernels but different neighborhood functions. The first search space (a) has three local minima (2, 4, and 6) and one global minima (8), while the second one has only one local minima (7) and one global minima (8). Finally, the third space has only one global minima (8).	21
1.2	Example of a convex downhill path. The red dots and edges highlight the convex downhill path while black dots illustrate its neighborhood.	21
1.3	Minima convex neighborhoods.	22
2.1	Main components in the Deep Learning Compiler. Adapted from [122]	29
2.2	(a) The abstract representation of the matrix multiplication kernel. (b-c) Canonical schedulings of the kernel. (d-f) Schedulings that result from applying loop tiling and loop unrolling with different parameters onto the canonical scheduling in (c).	30
2.3	The transformation vectors that produce kernels in Figure 2.2.	31
2.4	Pictographic metaphors for different search algorithms. (a,b) Grid search and Random sampling: every configuration within the gray area—and only those—will be evaluated. (c) Simulated Annealing: solid lines represent moves that go “down-hill”, <i>i.e.</i> , towards faster kernel configurations; dashed edges represent moves that go “up-hill”, <i>i.e.</i> , which accept to explore slower configurations to escape from local minima. (d) Droplet search: the darker the color of a configuration, the faster the running time of that configuration.	33
2.5	(a) Two operators of a computational graph. (b) Single operator that results from loop fusion.	35
3.1	Qualitative comparison between different previous work. See Chapter 3 for the meaning of columns. We classify the first six works as autotuning and the last six as scheduling problems.	39
4.1	Droplet Search, a Coordinate Descent variation proposed in this work to solve the Scheduling Problem from Definition 3.	44

4.2	Performance variation for the kernel in Figure 2.2 (b), considering the following parameters of the iteration space: (Left) $A = 1000, B = 800, C = 700$; and (Right) $A = 3500, B = 1400, C = 2400$. White pins show optimal tiling configurations, and gray pins show points of local minima that are not globally optimal.	47
4.3	(a) Downhill path from origin to global optimum on a three-dimensional hypersurface. (b) Expectation holds: origin and global optimum belong to the same convex region. (c) Expectation fails: origin and global optimum belong to different convex regions. Configuration v_3 is a <i>local minimum</i> , but not a global optimum.	48
4.4	(a-c) Progress of Droplet Search without speculation. Each number shows the order in which points are evaluated. Gray boxes denote points currently evaluated. Gray numbers show points in the neighborhood of the current candidate that was already evaluated. (d) Differential speculation: a subset of the extended neighborhood of the current candidate is evaluated to maximize thread occupancy.	50
4.5	The architectures evaluated in this work.	52
4.6	The machine-learning models evaluated in this work. “S” is the number of possible configurations formed by a given number “P” of transformation parameters. The largest neighborhood explored by Droplet Search at any time contains $2 \times P + 1$ points. “K” is the number of kernels that form each model. The implementation of the models vary according to the architecture. TVM’s implementation of MXNet does not run on the ARM boards.	53
4.7	Visual representation of nine different 17×17 transformation spaces, showing the path traversed by coordinate descent from origin to global optimum. Matrix multiplication is <code>mm2d</code> , and convolution is <code>conv3</code> . Performance improves from green to red. Black cells denote invalid configurations. We show, on top of each grid, the nesting order of loops, where the canonical configuration starts with the nested sequence <code>ijk</code>	55
4.8	Comparison of kernel scheduling techniques. “Model (ms)” is the running time of the best schedule for a given model. “Search (min)” is the time to find that configuration. Light-gray boxes denote statistically similar results (with a confidence level of 95%). Black boxes denote statistically significant best kernel times. Gray boxes with white fonts denote statistically significant best search times. Double borders denote the best schedule produced by an algorithm from AutoTVM only (<i>i.e.</i> , Ansor is not considered).	56
4.9	Effect of confidence level (the stop criterion of Section 4.3) on the quality of the best kernel configuration and on the running time of Droplet Search. . . .	57

4.10	Amount of memory transfers produced by Olivry et al. [103]’s cost model applied onto Listing 1 in Olivry et al.’s work: a tiled version of the kernel in Figure 2.2-b in this work. We vary tiling dimensions Ti_0 , Ti_1 , and Tk_0 . The input matrices have sides $Ni = Nj = Nk = 10K$	59
4.11	Part of the tiling search space generated for a ReLU layer by the XNNC’s analytical cost model for the P1 digital signal processor. The higher the cost (yellow), the slower the kernel is expected to behave, when deployed onto the actual hardware. To emphasize the non-convex regions, we show only the first 16 sizes of possible tiling windows.	60
4.12	Performance achieved by Droplet Search and exhaustive grid search on deep-learning models, as estimated using Tensilica’s XNNC cost model. The grid size is the size of the transformation space.	61
4.13	Hypersurfaces formed by the models in Table II of Renganarayana and Rajopadhye’s work [116].	62
4.14	(Left) Number of iterations until convergence of variations of droplet search. (Middle) Running time of different implementations of Droplet Search. (Right) P-values produced by a T-Test on populations of three executions of the best model found by each technique. The closer to 1.0 is the p-value, the more statistically similar are the two populations.	63
4.15	Comparison between Apache TVM and TensorFlow on the six kernels made available by Zhu et al. [159], on two graphics processing units. This figure uses the same notation as Figures 4.8 and 4.16: dark boxes indicate the fastest models, and gray boxes indicate the fastest search times. Light gray boxes indicate results that are statistically similar. TensorFlow does not implement search.	65
4.16	The search techniques of AutoTVM applied on individual layers of deep-learning models on the AMD R7-3700X. Evaluations average three samples. Light gray boxes mark running times that are statistically similar with confidence level of 95%. Black boxes denote statistically significant running times. Gray boxes mark statistically significant search times. The sum of search times approximates the results in Figure 4.8. The sum of kernel times, <i>i.e.</i> , “Time (ms)”, is strictly less than the running times reported in Figure 4.8.	66
4.17	Quality of the solution for kernel scheduling versus the number of evaluations (trials) used to find that solution for ResNet-18. The blue star marks the convergence of the Droplet Search.	68
5.1	(a) Abstract view of a kernel. (b) Naïve implementation of the abstract kernel. (c) Two optimization sketches for the naïve kernel. (d) Different annotations for the sketches.	72

5.2	(a) A two-dimensional view of the optimization space formed by the parameters P1 and P2 seen in Figure 5.1 c-i. (b) A two-dimensional view of the optimization space, this time involving parameters P9 and PC	72
5.3	Sketch generation rules. Ansor uses these rules to change the kernel search space. Each rule modifies a sketch, e.g., fusing, splitting or tiling loops. However, these rules do not change the annotations in the sketch.	74
5.4	(a) Sketch of the abstract kernel seen in Figure 5.1 (a). (b) Sketch that ensues from the application of the “Always inlining” rule. (c) Sketch that ensues from the initialization of the unrolling factor.	74
5.5	Rules that Ansor uses to create an initial population of kernels, which will be the starting point to the evolutionary search.	75
5.6	Mutation rules for evolutionary search. In contrast to the initialization rules seen in Figure 5.5, the mutation rules take the history of previous annotations when determining the next value of a given annotation.	76
5.7	The Droplet Search kernel scheduling algorithm. This pseudo-code is a simplified version of the original presentation of the algorithm, taken from [25]. We have removed speculation and parallelism from this version as these features are immaterial for the presentation of our ideas.	77
5.8	Summary of this proposal.	80
5.9	The files and functions in the Apache TVM code base that must be modified to incorporate the ideas proposed in this document.	81
5.10	The architectures evaluated in this report.	83
5.11	Comparative Analysis of Optimization Results on the x86 Architecture Using an AMD Ryzen 7 3700X Processor. Numbers show Ansor / DPAnsor ratios. Thus, results higher than 1.0 (in blue) denote improvements of DPAnsor (this work) over Ansor	85
5.12	Left: relative speed of kernels produced by DPAnsor over Ansor on four different architectures. Right: relative search time of DPAnsor over Ansor . Every bar is the geometric mean of relative times observed on 20 different models. Results above 1.0 represent improvements of DPAnsor over Ansor	85
5.13	The search vs quality line that characterizes four models optimized in the AMD (Left) and in the NVIDIA (Right) setting: the larger the model, the lower the slope. Numbers on the X and Y axes are speedup/slowdown relative to Ansor with a budget of 10,000 trials. Numbers for the left chart are available in Figure 5.11, and numbers for the right chart were observed on the Nvidia RTX3080. The labels on the dots (1 and 1k) refer to the number of trials given to DPAnsor	87
5.14	(Left) Variation in kernel quality (Ansor / DPAnsor)) with model size. (Right) Variation in search time (Ansor / DPAnsor)) with model size.	88

5.15	Comparison of different exploitation techniques used in tandem with Ansor on the x86-64 setting. Numbers are ratio of “ <i>SearchTechnique/DropletSearch</i> ”. Thus, numbers above 1.0 demonstrate the effectiveness of the technique proposed in this work.	90
5.16	Comparison of different exploitation techniques used in tandem with Ansor on the Cuda setting. Numbers are ratio of “ <i>SearchTechnique/DropletSearch</i> ”. Thus, numbers above 1.0 demonstrate the effectiveness of the technique proposed in this work.	91
5.17	Results of Figure 5.15, analyzed per layer of AlexNet	92
5.18	Number of trials sampled by Droplet Search until reaching convergence, considering the AMD x86-64 architecture. The average number of trials per layer divides the total number of trials per the number of layers in each deep learning model.	93
5.19	Average number of trials per model (considering 20 models) for different architectures. The averages per layer are the quotient of the total number of trials for all the models divided by the total number of layers.	93
5.20	Comparison between the running time of kernels produced via TensorFlow , PyTorch , Ansor or DPAnsor running on two different GPUs. The black cells highlight fastest results; the gray cells indicate ties (with confidence level of 95%). TensorFlow and PyTorch do not perform tuning.	94
5.21	Coefficient of Variation and P-Value Test. The lower the p-value, the more statistically significant are the results reported in this section.	95
6.1	The order of operator fusion can be encoded as sentences in a Binary Dyck Language formed by the nodes of the computational graph.	102
6.2	Comparison of different fusion algorithms. The Evolutionary Algorithm (EVL) is parameterized by a Population (Pop) of best graph candidates, plus a number of Generations (Gen).	103
6.3	Node fusion via the DFS approach, assuming a memory of five storage units. Thus, fusion is only possible if the cost of the incoming and outgoing edges of the fused node is less than five.	105
6.4	Node fusion via the MST approach, assuming a memory of five storage units. .	106
6.5	Asymptotic behavior of the different fusion algorithms discussed in this work, where V is the set of nodes in the computational graph.	108
6.6	(a) Code generated for a single operator A of the computational graph. (b) Code generated for two fused operators $(A \circ B)$	113
6.7	Ratio between estimated and stimulated cycles for models produced with the MST algorithm using DSP P6. Each point represents a different model. . . .	114
6.8	Neural networks used in this evaluation.	116

6.9	Number of cycles spent running 19 deep learning models on a cycle-accurate simulator for the DSP P6. Smaller is better.	116
6.10	Time spent to execute each of the algorithms during the generation of neural networks using the XNNC compiler for the DSP P6.	117
6.11	Reduction in the number of edges after fusion. Each bar is the number of edges before fusion divided by the number of edges after fusion. The higher the bar, the more edges were removed.	118
6.12	Binary size of the fusion algorithm against 19 neural network models for the DSP P6.	119
6.13	Behavior of fusion on different DSPs. (Left) Aggregated speedup of fusion on 19 models: higher is better. (Right) The aggregated slowdown in compilation time on 19 models: lower is better.	120

Contents

1	Introduction	18
1.1	Motivation	19
1.2	Problem formalization	20
1.3	Proposed solution	23
1.3.1	Thesis Statement	24
1.4	Outline	24
1.5	Publication	26
2	Background	28
2.1	Deep Learning Compiler Frameworks	28
2.2	The Search Space	30
2.3	Loop fusion	34
2.3.1	Factors that Prevent Loop Fusion	34
2.3.2	Fusion in Computational Graphs	35
2.4	Concluding Remarks	36
3	Literature Review	38
3.1	Autotuning and Scheduling	38
3.2	Frameworks	40
3.3	Loop Fusion Optimization	42
3.4	Concluding Remarks	43
4	Droplet Search	44
4.1	The Neighborhood Function	45
4.2	Convexity	46
4.3	Stop Criterion	49
4.4	Synchronous Parallelism and Speculation	49
4.5	Concluding Remarks	51
4.6	Evaluation	51
4.7	RQ1 – The Droplet Expectation	53
4.8	RQ2 – End-to-end Effectiveness	54
4.9	RQ3 – Stop Criteria	57
4.10	RQ4 – Analytical Models	58
4.10.1	Olivry et al.’s Cost Models	58

4.10.2	Tensilica’s Cost Models	59
4.10.3	Renganarayana and Rajopadhye et al.’s Cost Models	60
4.11	RQ5 – Parallelism and Speculation	61
4.12	RQ6 – Comparison with TensorFlow	63
4.13	RQ7 – Intra-Kernel Behavior	65
4.14	RQ8 – Convergence Rate	67
4.15	Concluding Remarks	68
5	Exploration versus Exploitation	69
5.1	Introduction	69
5.2	Exploration via Ansor ; Exploitation via Droplet Search	71
5.3	Space Exploration via Ansor	73
5.4	Space Exploitation via Droplet Search	77
5.5	Combining Ansor with Droplet Search	79
5.6	User Interaction	79
5.7	Implementation Details	81
5.8	Evaluation	82
5.9	RQ1 – On the Quality of End-to-End Models	84
5.10	RQ2 – On the Search Time	86
5.11	RQ3 – On the Impact of Model Size	86
5.12	RQ4 – On the Search Technique	88
5.13	RQ5: The Average Number of Droplet Search Samples	91
5.14	RQ6: Comparison with Well-Known Machine-Learning Frameworks	93
5.14.1	Threats to Validity – On the Statistical Significance of Reported Results	94
5.15	Conclusion Remarks	96
6	Fusion of Operators of Computational Graphs via Greedy Clustering – The XNNC Experience	97
6.1	Introduction	97
6.2	Fusion as an Optimization Problem	99
6.3	Greedy Clustering	101
6.3.1	The Linear Search	103
6.3.2	Criteria to Select Next Fusion Candidate	104
6.3.3	Asymptotic Complexity	106
6.4	Pseudocode	108
6.4.1	The DFS-Based Fusion Algorithm	108
6.4.2	The MST-Based Fusion Algorithm	109
6.5	Operator Fusion in XNNC	112
6.5.1	The XNNC Cost Model	113

6.6	Evaluation	115
6.6.1	RQ1: Impact of Fusion on Performance	116
6.6.2	RQ2: Impact of Fusion on Compilation Time	117
6.6.3	RQ3: On the Number of Fused Operators	118
6.6.4	RQ4: Impact of Fusion on Binary Size	118
6.6.5	RQ5: Fusion Across Different DSPs	119
6.7	Conclusion Remarks	121
7	Conclusion	122
	Bibliography	124

Chapter 1

Introduction

In the context of this work, a *kernel* is a function that reads and writes data indexed by a linear combination of natural numbers. Kernels are typically implemented as nests of affine loops. Examples of kernels include matrix multiplication, transposition, and convolutions. A *model* is a composition of kernels, such that the $(n - 1)^{th}$ kernel writes the data to be read by the n^{th} kernel. Following *Jin et al.* [68], we say that a deep learning model is a function implemented as alternating layers of kernels and non-linear functions (sigmoid, ReLU, *etc.*). Examples of deep-learning models include neural networks, such as BERT [47], ResNet-18 [64], VGG-16 [124], MobileNet [65], and MXNet [32].

The Space of Kernel Schedulings. A kernel is an abstract concept that supports different concrete implementations [68]. Each implementation, in this work, is called a *kernel schedule*. The set of every schedule of a kernel is called its *search space*. The choice of implementation impacts the performance of the kernel. Finding the best schedule for a kernel is an optimization problem whose objective function is running time: the faster a kernel runs, the better that schedule is. The problem of finding exact analytical solutions to kernel scheduling is open, even when fixing the computer architecture [136]. Hence, typical techniques are stochastic, take time to converge, and provide no guarantees of optimality [78].

Coordinates and Neighborhoods. Kernel schedules differ due to the application of code transformations, such as tiling, unrolling, and thread blocking. If we fix the sequence of transformations that define the search space, then each schedule is uniquely determined by the transformation parameters: unrolling factor and tiling window per loop, number of threads per block, *etc.* These parameters admit a total order: if $m < n, n, m \in \mathbb{N}$, then an unrolling factor of n is larger than an unrolling factor of m . This ordering determines *coordinates* on the space of kernel schedules; hence, yielding a notion of a neighborhood. The *neighborhood function* relates kernel configurations produced by transformation vectors that differ by a minimal difference on one parameter.

The Key Observation and the Implied Hypothesis: In a *convex optimization space* every local minimum is a global minimum. The space of kernel schedules is usually

not convex, as Example 12, in Section 2.2, will show. However, we believe that the following expectation applies to the vast majority of machine learning models: *It is possible to project the set of all kernel schedules onto a system of coordinates, such that the region between the origin of this space and the fastest kernel configuration form a convex hypersurface concerning the running time of the kernels.* Hence, the optimum configuration can be reached from the origin by deriving the running time function along a continuous neighborhood of kernel configurations. We call this observation the *Droplet Expectation* and formalize it in Chapter 4 (see Definition 1 on Page 22).

1.1 Motivation

The widespread adoption of machine-learning models across various domains of computer science has made kernel optimization a very compelling research challenge. This is evident in the abundance of discussions on “Tensor Compilers” [158] in recent programming languages publications. As a result, there are numerous motivations driving this work, and we will explore some of them below.

Performance Bottlenecks in Deep Learning Models. Deep learning models have ushered in remarkable advancements in various domains, from natural language processing to computer vision [151, 62]. However, the growing complexity of these models, coupled with the increasing scale of datasets, has brought to the forefront the issue of computational performance [82]. Many deep learning models, particularly those employed in real-time or resource-constrained applications, face significant performance bottlenecks [56]. These bottlenecks often originate from the inefficiencies in executing the numerous kernels that constitute these models.

The Quest for Efficiency and Speed. At the heart of these performance challenges is the role of energy expenditure in running the kernel. By identifying optimal parameters for a kernel, it becomes feasible to address energy consumption concerns as well. In neural network models, energy expenditure is unavoidable, given that both execution time and energy consumption affect the final outcome. Recent studies [98, 108, 109] indicate a correlation between parameter optimization for kernels and energy efficiency. Thus, the pursuit of optimized kernels proves compelling not only for enhancing performance but also for managing energy consumption more effectively.

The Complex Landscape of Kernel Scheduling. Kernels, serving as elementary computational functions, execute fundamental operations within models, including matrix multiplications, convolutions, and tensor manipulations. Optimizing kernel execution directly impacts overall model performance, given the recurrent nature of these operations in neural networks and other machine learning algorithms [130]. However, navigating The Complex Landscape of Kernel Scheduling poses challenges. Optimizing kernel execution is not straightforward due to the vast and intricate landscape of possible kernel scheduling. Each scheduling method represents a specific implementation approach, with code transformations like tiling, unrolling, and thread blocking introducing numerous parameters and options. Consequently, finding the optimal scheduling requires substantial computational resources and time, further complicating the optimization process

Therefore, the motivation behind this research stems from the critical need to address performance bottlenecks in deep learning models by optimizing the execution of kernels. Our exploration is driven by the belief that the Droplet Search can offer a novel and practical solution to this problem, potentially revolutionizing the field of kernel scheduling optimization and enhancing the efficiency of machine learning models in diverse applications.

1.2 Problem formalization

A **kernel optimization search space** is defined by a **set of kernel schedulings** $K = \{k_1, k_2, \dots, k_n\}$ and a **neighborhood function** $N(k_i)$ that defines, for a given kernel k_i , the set of kernels that are their immediate neighbors in the space. Figure 1.1 illustrates three different search spaces formed by the same set of kernels an different neighborhood functions.

The **kernel optimization search problem** consists in searching for the best kernel in the search space. The best kernel is the one that minimizes the objective function $C(k)$, also known as cost function. In our case, the cost function is the kernel execution time.

A kernel k_i is a **minima point** if $C(k_i) < C(k_j)$ for all $k_j \in N(k_i)$, *i.e.*, if the execution time of k_i is lower than all its immediate neighbors. A minima point k_i is classified as a **global minima** if $C(k_i) < C(k_j)$ for all $k_j \in K$, otherwise, it is classified as a **local minima**.

A **convex downhill path** on the search space is a path $p = \{p_1, p_2, \dots, p_m\}$, composed by a sequence of kernels, so that, $p_{i+1} \in N(p_i)$, *i.e.*, p_i and p_{i+1} are neighbors on the search space, and that for every kernel $k_j \in N(p_i)$ and $k_j \neq p_{i+1}$, $C(k_j) > C(p_{i+1})$.

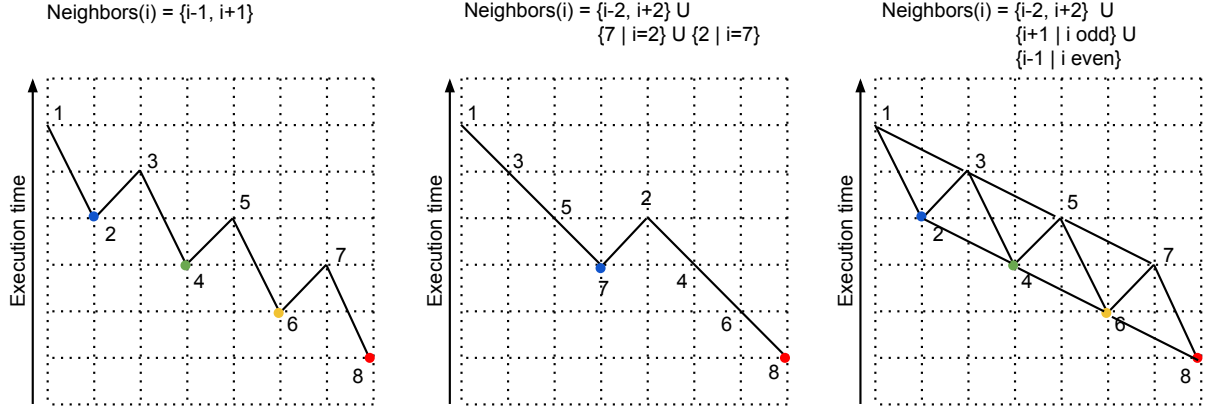


Figure 1.1: Examples of search spaces defined by the same set of kernels but different neighborhood functions. The first search space (a) has three local minima (2, 4, and 6) and one global minima (8), while the second one has only one local minima (7) and one global minima (8). Finally, the third space has only one global minima (8).

In other words, for each point p_i in path p , p_{i+1} is the neighbor with the lowest cost. The convex path is the path taken by the *Coordinate Descent* Algorithm [148] when starting at point p_1 and stopping at point p_m . Figure 1.2 illustrates a convex downhill path.

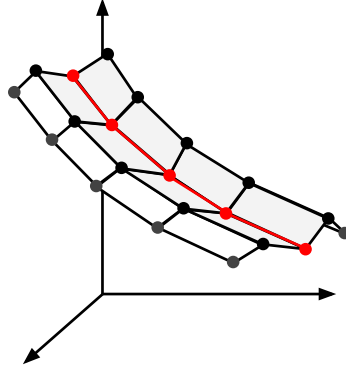


Figure 1.2: Example of a convex downhill path. The red dots and edges highlight the convex downhill path while black dots illustrate its neighborhood.

A **minima convex neighborhood** is the convex neighborhood around a minima point. This neighborhood is defined by the minima point itself, *i.e.*, k_i , and all points k_j so that there is a convex downhill path from k_j to k_i . Figure 1.3 illustrates the minima convex neighborhoods on a search space with 8 kernels and two minima points, (7) and (8). Notice that a search space may contain multiple minima points and that each one of these minima points defines on minima convex neighborhood.

The *Coordinate Descent* Algorithm [148] always reaches the minima point k_i if the search starts on a point in its minima convex neighborhood. Notice that a search space may contain multiple minima points, one global and several local ones. For each one of these minima points, there is a minima convex neighborhood. Also, assuming there are no two kernels with equal cost, then, each point in the search space belongs to exactly one

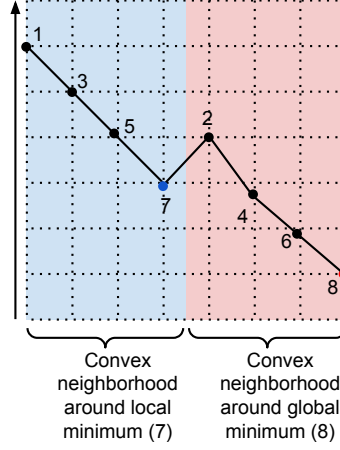


Figure 1.3: Minima convex neighborhoods.

minima convex neighborhood.

Given the above definitions, we can formalize the Droplet expectation as follows:

Definition 1 (The Droplet Expectation). *Let k_{opt} be the optimum kernel configuration, we expect the origin $\mathcal{K}$ to be located at k_{opt} minima convex neighborhood, i.e., we expect $\mathcal{K}$ to be at the global minima convex neighborhood.*

If the Droplet expectation holds, then we can successfully leverage the *Coordinate Descent* Algorithm [148] to find the optimum kernel by starting the search at $\mathcal{K}$.

Hypothesis 1: By carefully defining the neighborhood function, it is possible to shape the search space so that a) the Droplet expectation holds; and b) the neighborhood sets do not grow too much.

By defining the neighborhood function as the set of all kernels in the search space, i.e., by connecting all points as immediate neighbors, we would definitely ensure the Droplet Expectation holds; however, this space would be too costly to explore using heuristics that probe the neighborhood, such as the *Coordinate Descent* Algorithm. Hence, we would like to design neighborhood functions that produce small immediate neighborhood sets.

To validate Hypothesis 1, we might explore different ways of defining the neighborhood function. We could start by investigating the search spaces to verify whether including points that are not immediate, but are close to odd/even distances to the point. We might try to gain some gut feeling about how different parameters (e.g., loop unrolling, tiling factors, etc) affect the performances of the kernel and define special neighborhood functions for each one of these parameters.

1.3 Proposed solution

Based on this expectation and motivation, this work describes a scheduling technique called *Droplet Search*, which is currently part of Apache TVM¹. This thesis evaluates Droplet Search on six architectures (two x86 CPUs, two ARM CPUs, and two NVIDIA GPUs); and on six models (BERT, ResNet-18, VGG-16, MobileNet, MXNet, and Inception-v3). Chapter 4.6 shows that Droplet Search runs up to ten times faster than the other four search algorithms in AutoTVM [34] and the search algorithm in TVM’s Ansor [157]. The kernels produced via Droplet Search tend to outperform those produced by the other search approaches in AutoTVM, and approximate those produced by Ansor, even though the latter might use up to four times more transformation parameters. These results, summarized in Figure 4.8 (Page 56), come from the following contributions:

Intuition: the droplet expectation is not a theorem: in Section 4.10, we show that it is possible to disprove it using analytical cost models involving discontinuous functions. However, the hypothesis is expected to hold in cost models described by contiguous functions involving only positive domain and coefficients: a property that Renganarayana and Rajopadhye [116] call “*Positivity*”. These models are rather common: Table II in Renganarayana and Rajopadhye’s manuscript lists eight of them from previous work. More recent models [102, 103] share similar properties, as Section 4.10 discusses.

Simplicity: the implementation of droplet’s search in AutoTVM 0.13.0 (the pseudo code in Figure 4.1, Page 44) consists of 127 lines of commented Python code². For comparison, the implementation of TVM XGB’s search [34] uses 971 lines in three files (`xgboost_tuner.py`, `xgboost_cost_model.py` and `sa_model_optimizer.py`).

Adaptability: Droplet Search works in any setting where AutoTVM does, including settings like a Cortex A7, where Ansor cannot be used to optimize MobileNet [65].

Efficiency: Droplet Search converges more than twice as fast as the different algorithms available in AutoTVM (random sampling, grid search, genetic search, and XGBoost) and usually is more than four times faster than TVM’s AutoScheduler.

Effectiveness: in almost every experiment we ran, Droplet Search delivered kernels that either match or outperform those produced by the other schedulers in AutoTVM. Compared to Ansor, in a universe of 30 experiments, Droplet Search found faster

¹Release v0.13.0 (<https://github.com/apache/tvm/pull/14683>).

²See https://github.com/apache/tvm/blob/v0.13.0/python/tvm/autotvm/tuner/droplet_turner.py

kernels in eleven settings, and lost in twelve. However, Ansor uses up to four times more transformations.

1.3.1 Thesis Statement

We subsume the contributions previously enumerated into a single thesis, which shall direct the study presented in this proposal. The thesis statement follows below:

“A scheduling methodology based on a coordinate-descent search over the space of possible kernels is effective, because it is generally possible to project a system of coordinates onto this space with the following property: its origin and the optimal kernel lay on the same convex region with respect to the speed of kernels.”

Notice that our thesis relies on an “expectation”: it is generally feasible to project a system of coordinates onto the kernel optimization space, so that the Droplet Hypothesis holds. Such might not always be the case. Although, this study shows that the expectation is often true, in a GPU system, parallel evaluation encourages premature convergence to suboptimal regions due to insufficient exploration of the search space. For this specific case, the thesis may not be true.

1.4 Outline

This thesis is structured in the following seven chapters:

Chapter 2. This chapter explores Deep Learning Compiler concepts, emphasizing TVM tools’ role in optimizing kernels. We outline the significance of these techniques in enhancing model performance and discuss kernel optimization methods like loop unrolling and memory access optimization. The chapter also delves into autotuning methodologies and the Kernel Transformation Space. Additionally, it addresses Deep Learning Compiler frameworks, the Kernel Scheduling Problem, and differences between autotuning and scheduling techniques, offering insights into their methodologies.

Chapter 3. This chapter discusses the existing literature on deep learning model optimization with a focus on kernel scheduling and loop fusion optimization. The chapter divides the discussion into two main themes: autotuning and scheduling. Autotuning is a compiler-based method where the system is trained on various programs to optimize future implementations, while scheduling involves running the target program multiple times to gather data for optimization. Several influential works in both areas are highlighted, with specific attention to how recent research has approached loop fusion.

Chapter 4. This chapter details the Droplet Search algorithm. It selects promising configurations within a fixed number of iterations by evaluating neighboring options. The neighborhood function establishes an ordering over code transformations, aiding the search process. The goal is to converge to a globally optimal solution, with parallelization and speculation techniques used to accelerate the search by evaluating multiple configurations simultaneously. Also, Section 4.6 examines Droplet Search against AutoTVM and Ansor in five models across eight research questions. We explore confidence levels' impact on Droplet Search's performance on three different architectures (x86, ARM, and Cuda), revealing a better performance against AutoTVM and Ansor in execution and search time. Moreover, we investigate various cost models, affirming the Droplet Expectation's consistency. The study assesses parallelism and speculation effects on Droplet Search, showing improvement with speculation. It also compares Droplet Search with TensorFlow, highlighting Apache TVM's kernel performance. The analysis shows Droplet Search's efficiency on individual kernels, indicating faster convergence. Finally, we evaluate Droplet Search's convergence rate, showing stabilization before 10,000 evaluations across different models and hardware configurations.

Chapter 5 This chapter explores the balance between exploration and exploitation in neural network kernel scheduling optimization. The chapter introduces a hybrid approach where Ansor performs wide exploration of the optimization space, and Droplet Search focuses on local exploitation to refine the results further. Ansor, known for generating diverse sketches, serves as the exploration phase, while Droplet Search efficiently narrows down the best parameters within those sketches. By combining the two, the approach improves both search time and the quality of the resulting kernel configurations. Various experiments across multiple processors and deep learning models demonstrate significant speed improvements compared to using Ansor or Droplet Search alone.

Chapter 6 This chapter introduces a novel heuristic for operator fusion in computational graphs used in neural networks, specifically targeting optimization in Cadence's Xtensa Neural Network Compiler (XNNC). The chapter frames fusion as an optimization problem and discusses the challenges posed by fusing operators, which is conceptually similar to

the NP-complete problem of directed graph partitioning. It presents two main algorithms: a DFS-based approach and a Minimum Spanning Tree (MST)-based approach. The MST method proves to be more efficient, reducing the number of edges and improving performance across various deep learning models. The chapter evaluates the impact of fusion on different factors such as performance, compilation time, and binary size, showing how fusion can reduce resource usage and accelerate machine learning tasks across different processors.

Chapter 7 This chapter conclude this work.

1.5 Publication

At the beginning of my PhD, I published some papers as a second author. These experiences introduced me to different research areas and connected me with other researchers. This exposure gave me insights that have influenced the direction of my future work. The first paper [43] presents a system to analyze and compare algorithm classifiers and evaders using four adversarial games, focusing on applications like malware detection and plagiarism. The framework evaluates how well classifiers can detect algorithms and how effectively evaders can use obfuscation techniques to deceive them. The second paper [129] presents a code transformation technique called partial control-flow linearization (PCFL) to eliminate timing side channels in programs. PCFL is a vectorization technique that can be adapted to make programs operation invariant with respect to secret inputs. The technique is based on the idea of linearizing the control-flow graph of a program to remove tainted branches that depend on secret inputs. The third paper [71] is a technical report describing a new methodology for the generation of executable C benchmarks.

- Thaís Damásio, **Michael Canesche**, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando Magno Quintão Pereira. A game-based framework to compare program classifiers and evaders. In Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, pages 108–121, 2023. doi: 10.1145/3579990.3580012.
- Luigi Soares, **Michael Canesche**, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. ACM Transactions on Programming Languages and Systems, 45(2):1–43, 2023. doi: 10.1145/3594736.

- Cecilia Conde Kind, **Michael Canesche**, and Fernando Magno Quintao Pereira. Jotai: a methodology for the generation of executable c benchmarks. Technical report 02-2022. Universidade Federal de Minas Gerais, 2022.

In addition to these works, I have also published papers as a first author, which are not directly related to this work. The first paper [23] presents an optimal solution to the bit-aware register binding problem, which is a variation of the register assignment problem that takes into account the bit-widths of variables. I started working on High-Level Synthesis, but then shifted my research focus to scheduling and auto-tuning. The second paper [24] describes how to prepare a scientific artifact using Docker. Although these papers are not used as source material in this dissertation, working on them has greatly contributed to improving my research skills.

- **Michael Canesche**, Ricardo Ferreira, José Augusto Nacif, and Fernando Magno Quintão Pereira. A polynomial time exact solution to the bit-aware register binding problem. In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, pages 29–40, 2022. doi: 10.1145/3497776.3517773.
- **Michael Canesche**, Roland Leissa, and Fernando Magno Quintão Pereira. Preparing reproducible scientific artifacts using docker. arXiv preprint arXiv:2308.14122, 2023. doi: 2308.14122

This thesis subsumes the work described in three papers. The first paper [25] is the main contribution of this thesis: it introduces the Droplet Search. The second paper [26] shows a methodology to combine exploration and exploitation via droplet search. The third paper (currently under submission) applies the concept of convergence algorithms to operator fusion in neural networks.

- **Michael Canesche**, Vanderson M. Rosario, Edson Borin, and Fernando Magno Quintão Pereira. The droplet search algorithm for kernel scheduling. ACM Transaction Architecture Code Optimization, feb 2024. ISSN 1544-3566. doi: 10.1145/3650109.
- **Michael Canesche**, Gaurav Verma, and Fernando Magno Quintão Pereira. Explore as a storm, exploit as a raindrop: On the benefit of fine-tuning kernel schedulers with coordinate descent. arXiv preprint arXiv:2406.20037, 2024.
- **Michael Canesche**, Vanderson M. Rosario, Edson Borin, and Fernando Magno Quintão Pereira. Fusion of Operators of Computational Graphs via Greedy Clustering – The XNNC Experience. Submitted to International Symposium on Code Generation and Optimization (CGO), 2025.

Chapter 2

Background

This chapter reviews concepts related to Deep Learning Compilers, particularly emphasizing the utilization of TVM tools to optimize kernels efficiently. Beginning with a brief view of Deep Learning Compiler techniques and their significance in enhancing model performance, the chapter dives into the workings of TVM, showing its role in transforming high-level code into optimized machine instructions for diverse hardware architectures. Kernel optimization is an important aspect, where techniques such as loop unrolling, memory access optimization, and parallelization are explored to maximize computational efficiency. Also, we discuss the definition of the search space and its relation with configurations and parameters that influence optimization outcomes. Autotuning techniques are analyzed, revealing tools and methodologies designed to explore the search space and identify optimal configurations for compiled kernels. By synthesizing insights from TVM-based optimization and autotuning frameworks, this chapter not only provides an understanding of compiler optimization in deep learning but also informs future research directions aimed at advancing the efficiency and performance of deep learning workloads across diverse hardware platforms.

2.1 Deep Learning Compiler Frameworks

Deep learning compiler frameworks consist of two primary components: language front-ends and code-generating back-ends. Working in tandem, these elements efficiently compile deep learning models into executable code tailored for specific hardware targets [122]. In Figure 2.1, the main components are illustrated. The front-end takes a deep learning model as input, typically represented in a high-level description within a specific deep learning framework (e.g., TensorFlow, PyTorch, Keras) or domain-specific language. This input model, often a neural network architecture, undergoes translation into a high-level Intermediate Representation (IR), frequently graph-based. The IR provides an abstract representation of the neural network's structure and operations, facilitating

optimizations and transformations. Following IR generation, the front-end can apply target-independent optimizations to enhance overall model performance. These optimizations may include techniques such as tiling for improved cache memory utilization, operator fusion to combine multiple operations, or data layout transformation for optimal memory access patterns.

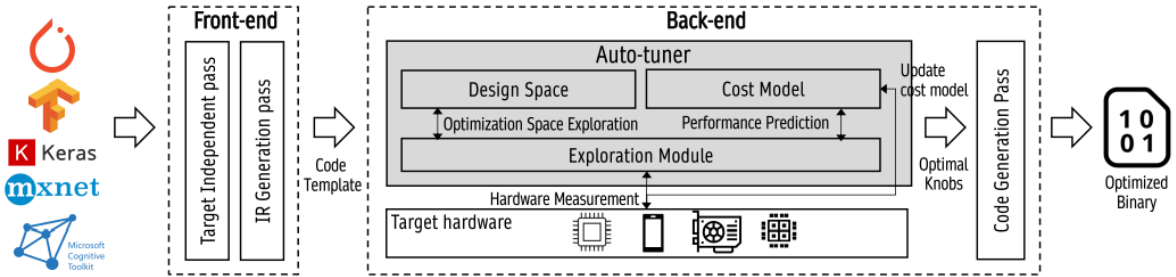


Figure 2.1: Main components in the Deep Learning Compiler. Adapted from [122]

The code-generating back-end receives the optimized high-level IR from the front-end. This IR has already undergone target-independent optimizations and is prepared for further processing. Subsequently, target-dependent optimization passes are applied to the IR. These passes are tailored to the hardware target on which the deep learning model is intended to run, to optimize the IR to better leverage the target’s hardware features and capabilities. This may involve architecture-specific optimizations and code generation. Auto-tuning, integrated into the target-dependent optimization passes, employs techniques that automatically explore various optimization strategies to identify the optimal set for a given hardware target. This approach aids in fine-tuning the model’s performance on the specific hardware. The culmination of this process is a compiled code or executable representation of the deep learning model finely tuned for the target hardware. This methodology facilitates the efficient deployment of deep learning models across diverse hardware platforms, spanning CPUs, GPUs, and specialized accelerators, harnessing the benefits of target-specific optimizations.

Machine-learning Based Auto-tuning is a technique that uses machine learning algorithms to automatically optimize the performance of software, hardware, or systems. This approach is commonly used in various fields as computer architecture, compilers, and signal processing. Specifically, in compilers, we have a deep learning compiler like Apache TVM [35]. This approach is valuable in scenarios where manually optimizing configurations is impractical due to the complexity and size of the optimization space. It can lead to significant improvements in system performance and is widely used in a range of applications, including deep learning, high-performance computing, and embedded systems [57].

2.2 The Search Space

A kernel admits an *abstract view*, formed by an iteration space, a data space, and a computation constrained into these zones¹. As mentioned in Chapter 1, this view can be implemented in multiple ways as long as the data dependencies encoded in the kernel’s computation are respected. Each implementation differs in how the iteration space is traversed. This scheduling determines how the kernel’s computation updates the data space. Example 1 clarifies these notions.

Example 1. Figure 2.2 shows an abstract view of the matrix multiplication kernel. The computations performed by the kernel can be indexed by triples (i, j, k) , which form its iteration space. The bounds R_A , C_B , and C_A that delimit this space abide by two constraints. The first, $C_a = R_B$, is mandatory for correctness; the second— R_A is even—we use for the sake of the example. Figure 2.2 also shows five implementations of the abstract kernel. These implementations produce the same matrix C ; however, the order in which the computations occur—the kernel scheduling—might vary.

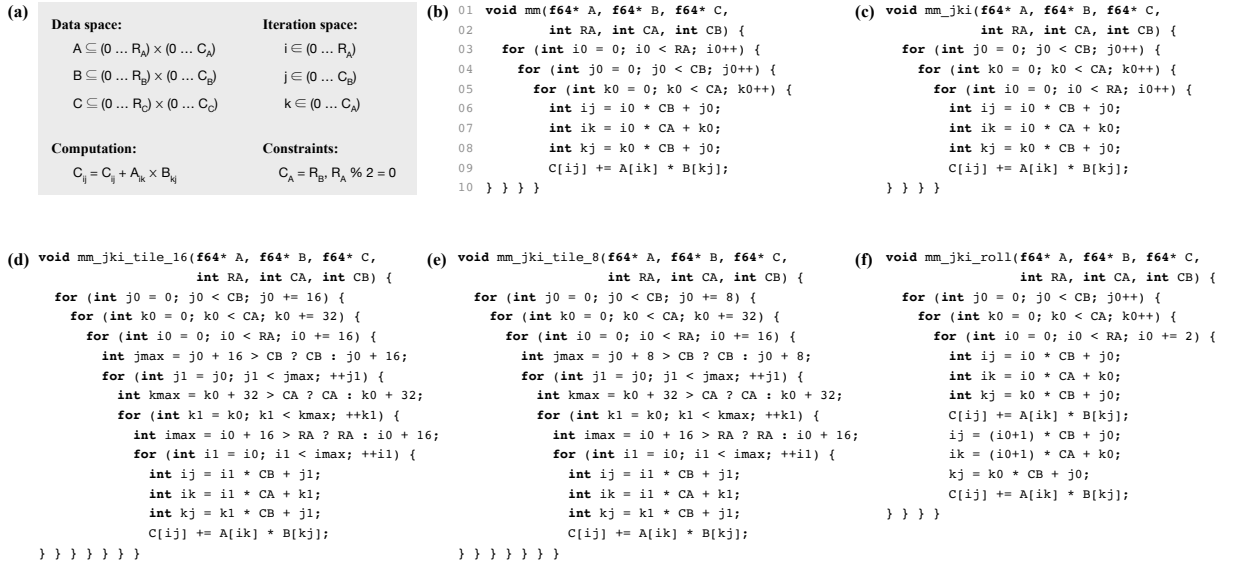


Figure 2.2: (a) The abstract representation of the matrix multiplication kernel. (b-c) Canonical schedulings of the kernel. (d-f) Schedulings that result from applying loop tiling and loop unrolling with different parameters onto the canonical scheduling in (c).

transformation Vectors. As hinted in Example 1, in the context of this work, the implementation of kernels differs concerning code transformations. These transformations

¹Notions of iteration and data space are standard in the compiler literature. Such concepts appeared independently in several works during the 80’s and 90’s [145, 143, 53, 6, 67, 111, 55, 75, 115, 114], eventually leading to the concept known today as the *Polyhedral Model*.

are guided by *parameters*. Example 2 shows parameters for two well-known transformations: tiling and unrolling.

Example 2. Figure 2.2 (e) shows the kernel that comes from Figure 2.2 (c) after the application of three instances of tiling: a transformation that partitions the iteration space into smaller regions. In this example, tiling happens along the three axes of the iteration space. The dimensions of the tiling window are 8, 32, and 16 points. Each one of these sizes is an optimization parameter. Figure 2.2 (f) shows the kernel produced after an application of unrolling onto the innermost loop of the kernel in Figure 2.2 (c). The unrolling factor, i.e., the transformation parameter, is 2.

The parameters of code transformations can be organized into *transformation vectors*. A transformation vector is a tuple whose elements represent these parameters. The order in which these parameters appear in the vector determines the order in which transformations are applied to programs. Thus, if a given code transformation has multiple parameters, then these parameters exist sequentially in the transformation vector. Example 3 shows how the parameters seen in Example 2 can be represented as transformation vectors.

Example 3. Consider the ordered application of the two loop-related compiler optimizations mentioned in Example 2 onto the abstract kernel in Figure 2.2 (a): tiling along dimensions A, B, and C, and unrolling on dimensions A, B, and C. The vectors representing any application of these code transformations have the format: $\langle \text{tile}_A : t_A, \text{tile}_B : t_B, \text{tile}_C : t_C, \text{roll}_A : r_A, \text{roll}_B : r_B, \text{roll}_C : r_C \rangle$. Figure 2.3 shows the vectors that produce the kernels in Figure 2.2.

Fig-1c $\langle \text{tile}_A : 0, \text{tile}_B : 0, \text{tile}_C : 0, \text{roll}_A : 0, \text{roll}_B : 0, \text{roll}_C : 0 \rangle$

Fig-1e $\langle \text{tile}_A : 16, \text{tile}_B : 8, \text{tile}_C : 32, \text{roll}_A : 0, \text{roll}_B : 0, \text{roll}_C : 0 \rangle$

Fig-1d $\langle \text{tile}_A : 16, \text{tile}_B : 16, \text{tile}_C : 32, \text{roll}_A : 0, \text{roll}_B : 0, \text{roll}_C : 0 \rangle$

Fig-1f $\langle \text{tile}_A : 0, \text{tile}_B : 0, \text{tile}_C : 0, \text{roll}_A : 2, \text{roll}_B : 0, \text{roll}_C : 0 \rangle$

Figure 2.3: The transformation vectors that produce kernels in Figure 2.2.

The iteration space of an abstract kernel can be traversed in many ways. If we ascribe one loop per dimension of the iteration space, any ordering of these loops that respects data dependencies is a valid traversal. The implementation of any such ordering, subject to a null transformation vector (the vector representing the absence of transformations), is called a *canonical scheduling*. A kernel may have more than one canonical scheduling, as Example 4 illustrates. Canonical schedulings lead to *Kernel Transformation Spaces*, which Definition 2 formalizes.

Example 4. Figures 2.2 (b) and (c) show canonical schedulings for the abstract kernel in Figure 2.2 (a). Any permutations of the loops in Lines 03-05 is a valid canonical scheduling. Canonical schedulings, by definition, are not subject to code transformations: they represent the application of a null transformation vector. As a consequence, they do

not show the effects of typical compiler optimizations, such as loop invariant code motion. Thus, the invariant code in Line 06 of Figure 2.2 (b) remains inside the innermost loop.

Definition 2 (Kernel Transformation Space). Let $\langle \mathcal{P}_1 : p_1, \mathcal{P}_2 : p_2, \dots, \mathcal{P}_n : p_n \rangle$ be a transformation vector, such that each parameter p_i comes from a range of parameters $\mathcal{P}_i$. The Cartesian product $\mathcal{P}_1 \times \mathcal{P}_2 \times \dots \times \mathcal{P}_n$ plus a canonical kernel $\mathcal{K}$ form a kernel transformation space. Each point of this space represents a configuration of $\mathcal{K}$ transformed by an instance of that Cartesian product. $\mathcal{K}$ is called the origin of this space and the set of transformation parameters is called the basis of this space. If the basis contains n parameters, the space is called n -dimensional.

Example 5. The vectors in Example 3, plus the canonical kernel in Figure 2.2 (c), form a six-dimensional transformation space. The origin of this space is the kernel in Figure 2.2 (c), and its basis is formed by three parameters of loop tiling, and three parameters of loop unrolling.

Not every sequence of code transformations might be valid. For instance, unrolling a loop by a factor larger than that loop’s trip count is not meaningful. Thus, Definition 2 implicitly assumes that the transformation space is produced by *valid* transformation vectors. Furthermore, the set of kernels determined by Definition 2 is non-exhaustive, even under bounded parameters. Non-exhaustiveness follows from two simplifications that we enumerate below, which are assumed in the construction of the transformation space. These simplifications are also adopted in AutoTVM [34], and have been incorporated into further work inspired by it [153]:

1. Code transformations are applied to the canonical kernel always in a fixed order. In other words, it is possible that by varying the order in which transformations are applied, the space could be extended with new concrete kernels.
2. Every point in the transformation space comes directly from the canonical kernel via the application of one transformation vector. Notice that successive applications of optimizations could, in principle, produce kernels outside the transformation space.

The performance of the kernels that form the transformation space might vary. These variations depend on the scheduling, on the target architecture and on the dimensions of the iteration and data spaces. The problem of finding the best implementation of a kernel, given these constraints, is known as the *Kernel Scheduling Problem*, a notion extensively discussed in previous work (see Chapter 3 for references). To keep this work self-contained, Definition 3 restates this concept. Search strategies for Kernel Scheduling are key to optimize deep learning models. Thus, Definition 3 is a well-researched problem. Example 6 revisits some of these techniques.

Definition 3 (Kernel Scheduling Problem). *Given a computing device, a transformation space with origin $\mathcal{K}$ and basis $(\mathcal{P}_1, \dots, \mathcal{P}_n)$, plus valid inputs for $\mathcal{K}$, kernel scheduling asks for the fastest implementation of $\mathcal{K}$ in the transformation space.*

Example 6. *Figure 2.4 provides a pictographic metaphor for different search techniques for the kernel scheduling problem implemented in AutoTVM, and contrasts them with the droplet search method that this work introduces. Chapter 4 shall describe the droplet algorithm. In regards to the other methodologies, the search proceeds as follows:*

Grid: *explores a bounded region of the transformation space. Regular ranges of transformation parameters determine this region.*

Random: *points of the transformation space are sampled randomly. Sampling usually follows a uniform distribution on predefined bounds placed onto the parameters.*

Annealing: *Simulated annealing [72] is a refinement of random search, where sampling alternates between regions that are close and distant from current best points. Points within a neighborhood are sampled, and, from time to time, the center of this neighborhood changes. The probability of such large jumps happening decreases with time.*

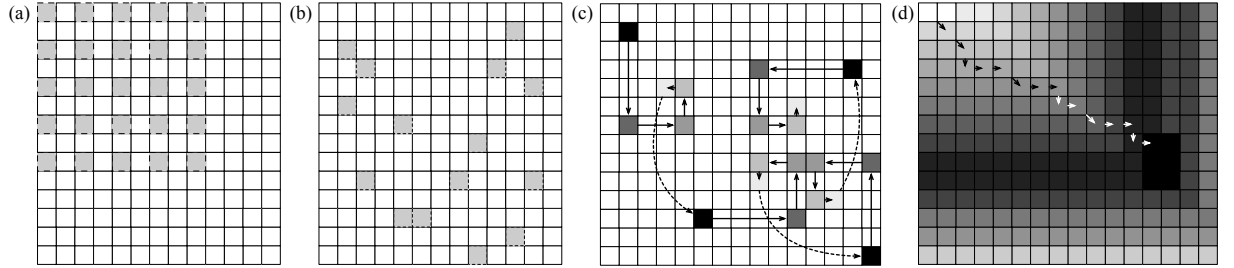


Figure 2.4: Pictographic metaphors for different search algorithms. (a,b) Grid search and Random sampling: every configuration within the gray area—and only those—will be evaluated. (c) Simulated Annealing: solid lines represent moves that go “down-hill”, *i.e.*, towards faster kernel configurations; dashed edges represent moves that go “up-hill”, *i.e.*, which accept to explore slower configurations to escape from local minima. (d) Droplet search: the darker the color of a configuration, the faster the running time of that configuration.

2.3 Loop fusion

Loop fusion is a compiler optimization where two or more loops that iterate over the same range are combined into a single loop². This code transformation is well-known in the compiler literature, being present in Watson et al. [140] classic catalog of code optimizations. Already in 1971, Watson et al. explained that loop fusion (or *jamming*, in their words), has three benefits: it amortizes the overhead of computing loop conditions; it reduces code size; and it exposes more opportunities for parallel execution. Additionally, loop fusion also tends to improve locality in the data cache, as Wolfe et al. [146] explains. Example 7 illustrates these benefits.

Example 7. Figure 2.5 (a) shows two operators of a computational graph: `kernel_mul` and `kernel_add`. These functions can be understood as implementing “layers” in a neural network, for instance. Each operator is implemented as a nest of loops. The outermost loops can be fused since they have identical ranges and there are no data or control dependencies violations. Figure 2.5 (b) shows one possible implementation of the fused loops. The function `kernel_mul_add`, in Figure 2.5 (b), is likely to be faster than the combined running time of the two routines in Figure 2.5 (a). It executes $M \times N$ fewer comparisons. Furthermore, it may avoid cache misses when updating $C[i, j]$.

2.3.1 Factors that Prevent Loop Fusion

Loop fusion is not always feasible, even when two loops have the same range. Still following Watson et al.’s classic definition of fusion, two factors prevent fusion: data and control dependencies.

Furthermore, loop fusion might not be beneficial even when feasible due to the memory hierarchy. Computer architectures typically feature multiple levels of memory, with smaller, faster storage and larger, slower storage. If the fused loop requires more data, this demand might exceed the available space in the fastest memory level. Example 8 illustrates how memory constraints can slow down fused loops.

²Loop fusion can also be applied to loops with different ranges; however, this work does not consider such cases. First, because it is not implemented in XNNC, the compiler used in our experimental evaluation. Second, fusing loops of different ranges has two drawbacks: (i) it requires inserting conditional checks within fused loops; and (ii) the fused loop must iterate over the maximum range of the original loops; thus, creating iterations that are innocuous for the smaller loop.

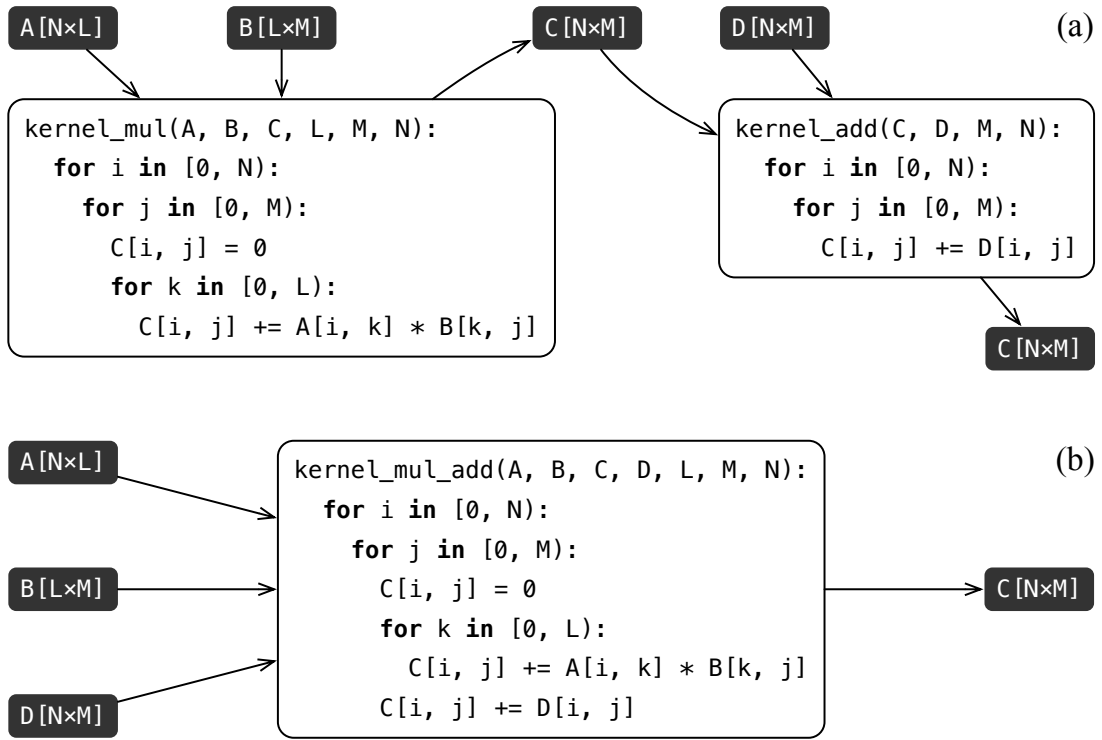


Figure 2.5: (a) Two operators of a computational graph. (b) Single operator that results from loop fusion.

Example 8. Let us assume that the operators in Figure 2.5 work on tensors containing 64-bit double-precision floating point numbers. In this case, the two operators in Figure 2.5 (a) require at least $64 \times \max(N \times L + L \times M + N \times M, 2 \times N \times M)$ bits of available memory. In contrast, the fused operator in Figure 2.5 (b) requires $64 \times (N \times L + L \times M + 2 \times N \times M)$ bits. Thus, it is possible that the data in Figure 2.5 (a) fits into the fastest memory level, but the data in Figure 2.5 (b) does not. In this case, code generated for `kernel_mul_add` in Figure 2.5 (b) will cause the system to perform more memory transfers between slower layers in the memory hierarchy.

2.3.2 Fusion in Computational Graphs

The implementation of many deep-learning models can be represented as a *Computational Graph*³. A computational graph represents a mathematical computation as

³The term “computational graph” and its formalization as a tool for representing and managing the computations in deep-learning models became more popular in the 2010s with the rise of deep learning frameworks [79] like TENSORFLOW, THEANO, and PYTORCH. However, the idea of representing computations using computational graphs can be traced back to the development of automatic differentiation (AD) techniques, which started being developed in the 1960s and 1970s [141, 63, 50]. AD is a method for efficiently computing derivatives of functions. It breaks down the function into elementary operations

a directed acyclic graph (DAG), where nodes represent operations (like addition, multiplication, *etc*) and edges represent the data (tensors, matrices, scalars, *etc*) flowing between these operations. The computational graph is used to compute a model’s output by following the graph from the input nodes to the output nodes and performing the necessary computations at each node. This process is called the *forward pass*. During training, the computational graph is used to compute the gradients of the loss function with respect to each parameter. This is done through *backpropagation*, which involves traversing the graph in reverse (from output to input) to calculate gradients.

In the context of this work, fusion is performed between the nodes that form the computational graph, as long as the conditions enumerated in Section 2.3.1 are met. The number of potential pairs of nodes that can be fused depends on the size and shape of the computational graph. As Example 9 shows, these computational graphs can be extensive; hence, they admit many different combinations of fusions.

Example 9. ALEXNET [4] is, by modern standards, a simple convolutional neural network (CNN). The XNNC representation of its computational graph contains 56 nodes and 89 edges. ALEXNET has five convolutional layers, three fully connected layers, seven ReLU activation functions, and one SoftMax activation function, in addition to pooling and normalization steps. Each layer’s operations, such as convolutions, ReLU activations and max pooling, are represented as separate nodes in the computational graph. Additionally, input data, weight parameters, and biases are represented as nodes. In contrast to ALEXNET, the XNNC representation of DENSENET169 [160] has 1,376 nodes and 2,231 edges. This difference is primarily due to the number of layers: 169 vs eight, but not only. DenseNet models introduce a unique feature called dense connectivity. In a Dense Block, each layer is connected to every other layer in a feed-forward fashion. This means that the output of each layer is passed as input to all subsequent layers within the same block. This dense connectivity significantly increases the number of nodes and edges in the computational graph.

2.4 Concluding Remarks

In this chapter, we explored the foundational concepts of register minimization, scheduling, and autotuning. We also discussed methodologies for constructing artifacts within these domains. This examination highlights the interconnectedness of these areas

and systematically applies the chain rule. In this context, the function is represented as a graph of operations—this is very similar to what we now call a computational graph in deep learning.

and their collective impact on optimizing computational processes. Understanding these relationships is crucial as we delve deeper into the subsequent chapters. The next chapter presents the literature review relating my work.

Chapter 3

Literature Review

This chapter explores existing literature and research surrounding deep learning model optimization, focusing on two key areas: kernel scheduling and loop fusion optimization. The chapter provides a comprehensive review of autotuning techniques, which optimize kernel execution through repeated tuning processes, and scheduling approaches that select the best configuration by analyzing performance data. Key works in these domains are discussed, offering insights into how modern techniques address performance bottlenecks in machine learning models. The discussion will lay the foundation for the proposed Droplet Search algorithm introduced later in the thesis.

3.1 Autotuning and Scheduling

This work aims to find the best concrete implementation for a given program. We recognize two main approaches to this theme, which we shall call *autotuning* (e.g., compiler autotuning) and *scheduling* (e.g., program autotuning, following Tollenaere et al. taxonomy [136]). The latter is the problem from Definition 3. These problems differ in two essential ways:

Training: In autotuning, the compiler is trained offline on many programs—its training set—before being applied to an unknown program. Thus, the compiler uses information acquired from general programs before optimizing a specific program. In scheduling, there is no pre-training phase: the compiler does not try to generalize the behavior of a universe of programs to predict the behavior of an individual program.

Sampling: In autotuning, the compiler typically evaluates the target program once (although there are exceptions, like in the work of Cavazos et al. [29]), using, as a guide, the behavior learned from observations made on the training set. In scheduling, the compiler is allowed to run the target program multiple times. Information acquired from these executions will guide the search for good optimizations.

Much of the current techniques employed in autotuning originate in the work of Cavazos and his collaborators [12, 11, 27, 28, 2, 95, 127, 135]. The growing availability of predictive models and benchmarks to train these models has made autotuning works common in the recent literature [17, 38, 39, 126, 42]. Figure 3.1 compares six autotuning-based techniques with six scheduling-based approaches. Whereas autotuning typically concerns the optimization of general programs, scheduling is mainly seen in the optimization of deep-learning models composed of kernels. Autotuning has been used, for instance, to find good sequences of optimizations for `clang` [41, 126]; or to fine-tune the Hotspot JIT compiler [27].

Work	Training	Sampling	Space	Guide	Target	Platform
Rapidly Selecting Good Compiler Optimizations Using Performance Counters	Y	± 25	list	Performance Counters	Programs	General
Exploring the Space of Optimization Sequences for Code-Size Reduction: Insights and Tools	Y	1	list	Program features	Programs	General
Method-specific dynamic compilation using logistic regression	Y	1	vec[n], n is 4, 9, or 20	Program features	Functions	JIT
Hybrid Optimizations: Which Optimization Algorithm to Use?	Y	1	vec[1]	Program features	Function	JIT
Mapping Computations in Heterogeneous Multicore Systems with Statistical Regression on Program Inputs	Y	1	vec[8]	Input features	Functions	Parallel
Compiler-Based Graph Representations for Deep Learning Models of Code	Y	1	vec[1]	Program features	Programs	Parallel
Ansor : Generating High-Performance Tensor Programs for Deep Learning	N	[1..]	param list	Program features	Kernels	ML
Automatic Generation of High-Performance Convolution Kernels on ARM CPUs for Deep Learning	N	[1..]	param list	Grid search	Convolutions	ML
TVM: An automated end-to-end optimizing compiler for deep learning	N	[1..]	params	Program features	Kernels	ML
Autotuning Convolutions Is Easier Than You Think	N	[1..]	params	I/O complexity model	Convolutions	ML
I/O Lower Bounds for Auto-tuning of Convolutions in CNNs	N	[1..]	params	I/O complexity model	Kernels	ML
This paper	N	[1..]	params	Coordinate descent	Kernels	ML

Figure 3.1: Qualitative comparison between different previous work. See Chapter 3 for the meaning of columns. We classify the first six works as autotuning and the last six as scheduling problems.

The transformation Space. Scheduling techniques usually fix the sequence of optimizations that form the search space, and vary their parameters [34, 136, 153, 37]. However, there are scheduling approaches that accept different transformation vectors [107, 88, 157, 51]. Within the TVM community, these different transformation vectors are called *templates*. For instance, a common technique—adopted in TVM’s Ansor—is to assume that each interchange of loops forms a different template. Figure 3.1 distinguishes fixed vectors as “params” and templated vectors as “param list”. Autotuning usually fix the parameters of each optimization; however, they have more freedom to compose sequences of optimizations. For instance, Cavazos et al. [29] form sequences of 500 optimizations drawn from a universe of 121 possible compilation flags. This approach is also adopted by Silva et al. [126]: they produce lists of up to 100 elements drawn from approximately

80 compilation flags. Figure 3.1 uses the notation “list” to denote this way to build the search space. Differently, Cavazos et al. [27] assumes a fixed sequence of optimizations but disables some of these transformations; and Cavazos et al. [28] uses vectors with just one optimization: the type of register allocator. In Figure 3.1, we denote fixed-length vectors as “vec[n]”.

The Search Guide. Techniques used in autotuning or scheduling differ in how the search for good transformation sequences is performed. In this work, we use coordinate descent. Our approach does not depend on static characteristics of the program; only on its dynamic behavior. Several search techniques use program features (static characteristics) to steer the search. These techniques are usually data-agnostic. An exception is the work of Da Silva et al. [42], who use the runtime values of inputs to choose program configurations. Da Silva et al. capitalize on the convexity of the search space; however, in their case, tuning is guided by linear regression, not coordinate descent. Recent scheduling techniques have used analytical models to prune the search space [70, 153, 136, 92, 37]. Mogers et al. [92] have shown, in the context of the LIFT compiler [132], that pruning can be very effective, as “*only 1 out of 49,000 candidates [generated by random search to optimize convolution] satisfies the constraints [hence is valid]*”. Pruning is orthogonal to the search techniques that Chapter 3 evaluates. For instance, an interesting continuation of the ideas in this work would be to use Tollenare et al.’s cost model [136] to remove from the active neighborhood kernel configurations that are unlikely to improve on the best candidate seen by coordinate descent.

3.2 Frameworks

The design and implementation of systems to run machine learning models have been experiencing constant progress. Following Ding et al. [49], we recognize two main approaches to run deep-learning models, which we call *interpretation* and *compilation*. In the former approach, frameworks such as PyTorch [105], TensorFlow [1], and the ONNX runtime [69] implement machine learning models by binding operations to libraries such as cuDNN, cuBLAS, and CUTLASS. In the latter approach, tools such as TVM, Halide [112], TorchInductor [8] or XLA generate code for the operations that constitute the model. This work concerns compilation; nevertheless, to provide some perspective to the reader about the relative effectiveness of these systems, Section 5.14 compares our approach with standard distributions of PyTorch and TensorFlow. Contrary to the findings in Section 5.14, Ansel et al. have recently shown that PyTorch, in compilation mode, con-

sistently outperforms **Ansor**. However, in those experiments, **Ansor** was used solely as a code generator without performing any autotuning. In our setup, once autotuning is enabled, **Ansor** can outperform **PyTorch** in most kernels.

Design and Construction of Autotuners The original presentations of **AutoTVM** [33] and **Ansor** [157] outline the key techniques utilized in this work. These tools explore optimizations typically described via the *polyhedral model* [54]. Said optimizations, including fusion, tiling, and fission, are commonplace in tools such as **POLLY** [61], **GRAPHITE** [137] and **PLUTO** [16]. These tools do not solve the autotuning problem; however, they offer the basic infrastructure to do so, as Tavarageri et al. [134] have demonstrated with their **POLYDL** framework.

Recent research focuses on pruning the search space to enhance the efficiency of autotuners. For instance, Tollenaere et al. [136]’s search algorithm uses a cost model to avoid exploring regions of the kernel space unlikely to yield good schedules. However, building an accurate analytical model seems an illusive problem. In the words of Ritter and Hack [119]: “*Modern processor designs use many techniques to improve overall performance that cause complex, irregular performance characteristics.*” Additionally, there has been much work into adapting autotuners to deal with tensors whose shape is not statically known [96, 106]. For instance, **DIETCODE** [155] and **SoD²** [99] use a cost model similar to **Tollenaere et al.**’s, to represent the shape of a tensor as symbols that will only be known at runtime. We notice that autotuning is not restricted to scheduling the order of computations within a kernel. For instance, autotuning techniques can be used to choose the format to store tensors (sparse vs dense) [3, 48, 147], or the bitwidth used for quantization [66, 73].

Chapter 5 does not propose a new kernel scheduling algorithm. Rather, it is bringing forward the observation that the combination of two well-established heuristics tend to bring much benefit to the generation of high-quality kernels. This benefit is measured not only in terms of the speed of the final code, but also in terms of the efficiency of the search technique. In this sense, an important consequence of our work is the fact that it makes **Ansor** more hardware-aware. By using coordinate descent as an exploitation approach, **Ansor**’s search remains circumscribed to a region that is likely to benefit more from the available hardware. Some research groups have, independently, shown that the design of hardware-centric (in contrast to input-centric) autotuners tend to accelerate the search process [159, 49, 81]. This work demonstrates this point without designing an algorithm that needs to be parameterized with hardware characteristics.

3.3 Loop Fusion Optimization

As mentioned in Section 2.3, loop fusion is a well-known compiler optimization, with descriptions dating back to the early 1970s [140]. Early literature on fusion focused on implementing the optimization at the loop level. The two critical problems addressed were as follows, typically involving two loops, L_1 and L_2 , iterating over the same space:

P1: Are there data or control dependencies that prevent the fusion of L_1 and L_2 ?

P2: How can the fusion of L_1 and L_2 be implemented to produce efficient object code?

A representative study from that time is Darte [44]’s, which analyzes the theoretical complexity of several subproblems related to **P1** and **P2**. Although graphs were central to the literature, they were completely different from the computational graphs used in this work. Instead, authors such as Darte [44], Liu et al. [85], and Xue et al. [149] focused on the “data dependence graph”¹: a structure that relates memory access operations and their dependencies, which was key for solving **P1**. This work assumes the existence of effective solutions to **P1** and **P2**—a reasonable assumption given the depth and breadth of the literature just discussed.

Fusion at the Level of Computational Graphs In recent years, the focus of loop fusion has shifted from dependence graphs to computational graphs, primarily driven by the need to generate efficient implementations for machine learning models. As an example of this line of work, the simple act of combining two high-level instructions from the MLIR [77] dialect proposed by Golin et al. [59] implies fusing their low-level representation, once these instructions are lowered to the LLVM intermediate representation. In this contemporary context, two new challenges replace the traditional concerns of **P1** and **P2**. The first challenge—addressed in this work—is the Operator Fusion Problem, stated in Definition 8. The second challenge involves designing effective cost models to guide fusion decisions [150, 87, 58].

Several solutions to operator fusion have been proposed in the more recent compiler literature. The most relevant work to this work is by Cai et al. [21, 20], whose BKT algorithm we have implemented and evaluated in Section 4.6. However, we emphasize that we did not directly reuse Cai et al., Cai et al.’s artifact, as it does not generate binary code for a neural network. Instead, Cai et al. [21, 20]’s OPTIMUS produces a report indicating which nodes of the computational graph should be fused. While we were able to reproduce

¹The earliest formal definition of a dependence graph that we are aware of appears in Section 2.2 of Kuck et al. [74]’s work. However, Kuck et al. note that this concept was already familiar to compiler writers at the time.

this artifact, we could not use it to analyze our largest networks within a 60-minute time window. Thus, we had to reimplement Cai et al., Cai et al. algorithm within the XNNC infrastructure.

3.4 Concluding Remarks

This chapter provides a survey of the existing literature on deep-learning model optimization, with a specific focus on addressing the kernel scheduling problem. The primary aim is to provide a foundation for introducing the main contribution of this work: the Droplet Search kernel scheduling algorithm. Additionally, we delve into two research domains: program autotuning and kernel scheduling, in order to contextualize how this work builds upon prior knowledge. The forthcoming chapter will further elucidate how the Droplet Search algorithm enhances kernel scheduling methods, thereby expanding the reader's understanding of the advancements presented in this work.

Chapter 4

Droplet Search

The Droplet Search is a greedy heuristic to explore the kernel transformation space. The proposed technique is a variation of the *Coordinate Descent* Algorithm¹, with extensions proposed by Richtárik and Takác [118] to enable synchronous parallelism. Figure 4.1 provides an overview of the search algorithm evaluated in this work, and Figure 2.4 (d) provides a graphical metaphor to illustrate how the algorithm works: at each step—up to a maximum fixed number of iterations—search chooses the most profitable kernel configuration that is considered a “neighbor” of the current best configuration. The rest of this section discusses each part of this algorithm shown in Figure 4.1.

```

01 def search(num_iterations):
02     candidate = origin                                # The origin of the coordinate descent is the
03     visited = set()                                    # canonical kernel from Def. 2.5
04     for i in range(num_iterations):
05         neighborhood = get_neighbors(candidate).difference(visited)    # The neighborhood function from Section 3.1
06         if len(neighborhood) < num_threads():                # grabs one kernel to evaluate per thread
07             neighborhood.union(speculate()).difference(visited)    # The `visited` set avoids double evaluation
08             visited.union(neighborhood)
09             ps = map(lambda n: Process(target=n.run).start(), neighborhood)    # Synchronous parallelism through speculation
10             map(lambda p: p.join(), ps)    # is the subject of Section 3.4
11             new_candidate = min(neighborhood, key=lambda e: e.running_time)
12             if is_statistically_faster(new_candidate, candidate, 0.05):    # Stop criteria are discussed in Section 3.3
13                 candidate = new_candidate    # We use a confidence level of 0.05 to
14             else:    # distinguish the running time of samples.
15                 return candidate

```

Figure 4.1: Droplet Search, a Coordinate Descent variation proposed in this work to solve the Scheduling Problem from Definition 3.

The successful application of Coordinate Descent depends on a suitable “neighborhood function”, subject of Section 4.1. Search tends to converge to a global optimal, based on the intuition that Section 4.2 provides. Convergence is based on criteria discussed in Section 4.3. This closes with discussions of two optimizations that speed search up: parallelization and speculation (Sec. 4.4).

¹It is not clear who invented Coordinate Descent. Descriptions of the algorithm can be found in classic textbooks [152]. For a comprehensive overview, we recommend the work of Wright [148].

4.1 The Neighborhood Function

Any code transformation used in this work is parameterized by one positive integer. Example 2 (Page 31) describes some of these parameters. This assumption—transformations defined by one positive integer—forces a total ordering over different instances of a code transformation. The *domain* of a code transformation is the set of all the values of its parameters. This work works only with numeric domains; however, a domain does not need to be contiguous, as Example 10 shows.

Example 10. *The list below shows different compiler optimizations and the ordering between their instances. The example takes liberties for the sake of the illustration; i.e., vector lengths are architecture-dependent, and not every size might be available:*

Peeling, with parameter `peel` = $n, n \in [0, 1, 2, \dots, \text{maxp}]$ being the number of iterations to peel. The set of values $\{0, 1, \dots, \text{maxp}\}$ is the domain of the transformation.

Unrolling, with parameter `roll` = $n, n \in [0, 1, 2, \dots, \text{maxr}]$ being the unrolling factor.

Thread blocking, (in graphics-processing units) with parameter `thread` = $n, n \in [0, 1, 2, \dots, \text{maxt}]$ being the number of GPU threads per block.

Tiling, with parameter `tile` = $n, n \in [N/20, N/15, N/12, N/10, N/6, N/4, N/3, N/2]$, where N is the size of the iteration space. We assume that N is a multiple of 60. Notice that this example chooses perfect divisors of the iteration space, but different ranges are possible.

Vectorization, with parameter `vect` = $n, n \in [0, 2, 4, 8, 16]$ being the vector length.

In any of the above examples, we say that an instance of a transformation is less than another instance based on the ordering between their parameters. For example, if we consider vectorization (`vect`), then `vect` = $m < \text{vect} = n$ if, and only if, $m < n$.

The order between parameters leads to a notion of neighborhood, formalized as follows:

Definition 4 (Neighborhood). *Consider two transformation vectors of an n -dimensional space: $v_1 = \langle \mathcal{P}_1 : p_1, \dots, \mathcal{P}_{i-1} : p_{i-1}, \mathcal{P}_i : x, \mathcal{P}_{i+1} : p_{i+1}, \dots, \mathcal{P}_n : p_n \rangle$; and $v_2 = \langle \mathcal{P}_1 : p_1, \dots, \mathcal{P}_{i-1} : p_{i-1}, \mathcal{P}_i : y, \mathcal{P}_{i+1} : p_{i+1}, \dots, \mathcal{P}_n : p_n \rangle$, such that each p is a transformation parameter within range $\mathcal{P}$. Vectors v_1 and v_2 are neighbors along dimension $i, 1 \leq i \leq n$, if they differ only on dimension i and if:*

1. $x < y$ (without loss of generality);

2. $\forall z \in \mathcal{P}_i, z \neq x$, if $z < y$, then $z < x$.

3. $\forall z \in \mathcal{P}_i, z \neq y$, if $z > x$, then $z > y$.

If two vectors are neighbors along one dimension, then they are called neighbors. The set of every neighbor of a given transformation vector is called the neighborhood of that vector.

Each dimension of a transformation vector v is considered separately when determining the neighbors of v . There exists at most two neighbors per transformation parameter; therefore, the number of neighbors of v grows linearly with the number of its dimensions. This observation is essential for scalability: if the neighborhood of a vector considered variations in two or more dimensions, then the size of the neighborhood would be exponential on the number of dimensions.

Example 11. Consider the following transformation vector: $v = \langle \text{tile}_A : 0, \text{tile}_B : 8, \text{tile}_C : 16, \text{roll}_A : 4, \text{roll}_B : 0, \text{roll}_C : 0 \rangle$, which applies onto the canonical kernel in Figure 2.2 (c). If we let $\text{roll}_A = [2, 4, 8, 16, 24]$, then v has two neighbors along dimension roll_A . These neighbors are $v_1 = \langle \dots, \text{roll}_A : 2, \dots \rangle$; and $v_2 = \langle \dots, \text{roll}_A : 8, \dots \rangle$.

4.2 Convexity

The goal of this work is to find implementations for abstract kernels that minimize their running times. To this purpose, we define the running time function RT as follows:

Definition 5 (The Running Time Function). *Given: (i) a canonical Kernel $\mathcal{K}$; (ii) a transformation vector v with parameters $\mathcal{P}$; (iii) a computer architecture A ; and (iv) input data I for the canonical kernel; we define the running time function $RT_{A,I,\mathcal{K}}(v) : \mathcal{P} \mapsto \mathbb{R}$ as a function that maps the implementation of $\mathcal{K}$ optimized by v to the time it takes to process input I on target A .*

The notion of neighborhood makes the transformation space a *coordinate space*: it is possible to define the distance between two transformation vectors. The running time function RT divides this coordinate space into two halves: if $RT_{A,I,\mathcal{K}}(v) = t$, then we have a region formed by $t_{\text{lower}} < t$, and a region formed by $t_{\text{higher}} \geq t$. Thus, $RT_{A,I,\mathcal{K}}$ determines a *hypersurface* onto the transformation space. This hypersurface is convex if its local minima are equal to its global minimum. We say that vector v is a *minima* of $RT_{A,I,\mathcal{K}}$ if $RT_{A,I,\mathcal{K}}(v) < RT_{A,I,\mathcal{K}}(v')$, for any v' within the neighborhood of v . A global minimum is the smallest local minimum in a set. Example 12 illustrates these concepts.

Example 12. Figure 4.2 shows hypersurfaces produced by two running time functions. Each function is parameterized by a different architecture and different inputs (the dimensions of the iteration space). The functions are parameterized by the same canonical kernel—the configuration in Figure 2.2(b). Two parameters form the transformation space: $\text{tile}_A \in [0, 20, 40, \dots, 120]$, and $\text{tile}_B \in [0, 20, 40, \dots, 120]$. The figure shows, for each running time function, its optimal configuration. The figure also shows at least one local minimum that differs from the optimum. These hypersurfaces are not convex because they contain local minima that are not globally optimal.

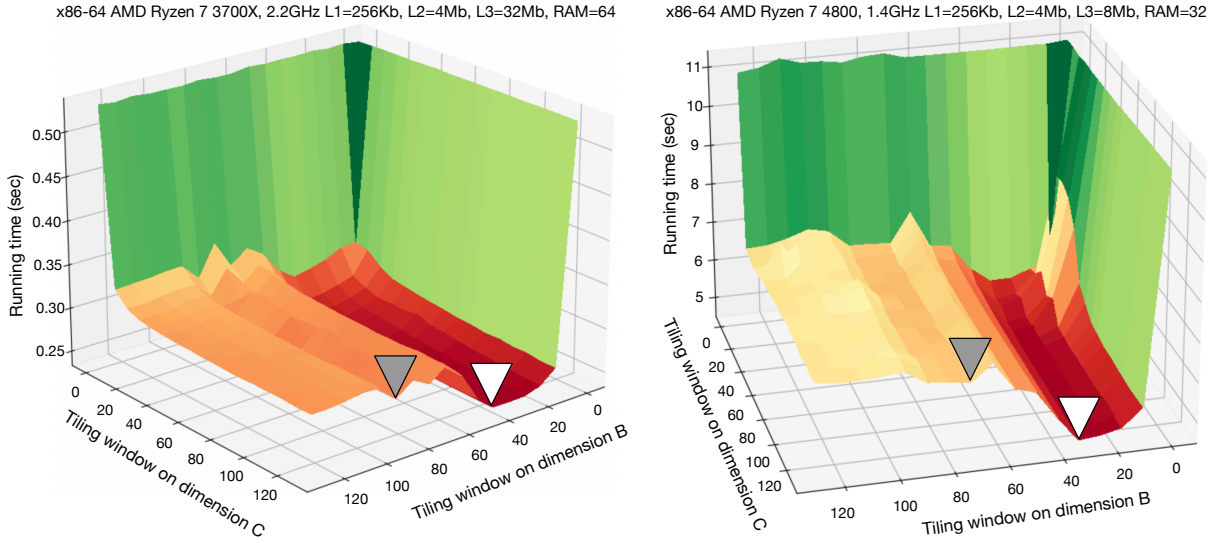


Figure 4.2: Performance variation for the kernel in Figure 2.2 (b), considering the following parameters of the iteration space: (Left) $A = 1000, B = 800, C = 700$; and (Right) $A = 3500, B = 1400, C = 2400$. White pins show optimal tiling configurations, and gray pins show points of local minima that are not globally optimal.

The hypersurfaces in Figure 4.2 are not convex. However, they have the following property: the origin and the global optimum belong to the same convex region. This property remains true, at least for the two settings in Figure 4.2, if we add more dimensions to the transformation space considered in Example 12, such as an extra tiling window, or unrolling of the innermost loop, or interchange of any pair of loops. Definition 1 states this observation as our working hypothesis.

Definition 6 (The Droplet Expectation). Let v_{opt} be the optimum kernel configuration of the running time function $RT_{A,I,K}$. We expect v_{opt} and K to belong into a convex contiguous subset of the hypersurface determined by $RT_{A,I,K}$. Therefore, we expect the existence of a chain of neighbor vectors $v_0, v_1, v_2, \dots, v_{n-1}, v_n$, where $K = v_0$ and $v_n = v_{opt}$, with the following properties:

Contiguous chain: $v_i \in \text{neighborhood}(v_{i-1}), 0 < i \leq n$; and

Descending chain: $RT(v_i) < RT(v_{i-1}), 0 < i \leq n$.

A continuous hypersurface can be partitioned into *convex regions*: maximal convex sets formed by the transitive closure of the neighborhood function. Whenever the Droplet Expectation is confirmed, origin and global optimal belong into the same convex region. Thus, there exists a “*downhill path*” from origin to optimum that can be found by Coordinate Descent (assuming an idealized running time function without statistical variations). If the Droplet Expectation fails, then the origin and global optimal belong into distinct convex regions. Figure 4.3 illustrates these ideas.

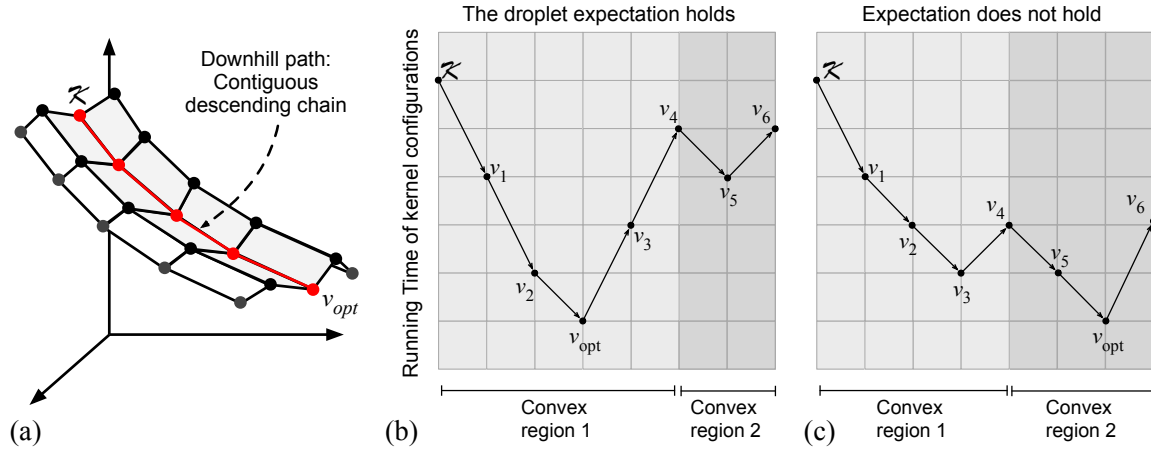


Figure 4.3: (a) Downhill path from origin to global optimum on a three-dimensional hypersurface. (b) Expectation holds: origin and global optimum belong to the same convex region. (c) Expectation fails: origin and global optimum belong to different convex regions. Configuration v_3 is a *local minimum*, but not a global optimum.

Intuition. The hypothesis stated in Definition 1 is not a certainty: it is possible to disprove it with analytical models involving discontinuous functions, as Section 4.10 shows. However, Chapter 4.6 demonstrates that the hypothesis holds on a variety of architectures and models. Indeed, the effect of many compiler optimizations can be described by polynomial equations involving only positive coefficients ranging on positive domains. The second derivative of such functions, if they exist, will be always positive. This condition is sufficient to ensure convexity [14, Pro.1.1.10]. In the words of Renganarayana and Rajopadhye [116], “*The use of polynomial functions with this property leads to convex optimization problems which can be solved for real solutions in polynomial time*”. Renganarayana and Rajopadhye call this property *Positivity*.

4.3 Stop Criterion

The existence of the convex path expected in Definition 1 does not ensure that such a path can be discovered via a coordinate descent search procedure. Running time has a stochastic nature: This nature implies that the actual evaluation of $RT_{A,I,\mathcal{K}}$ on a kernel produced by a transformation vector v is prone to variations. Thus, coordinate descent might reach suboptimal configurations that are apparently optimal due to measurement fluctuations on $RT_{A,I,\mathcal{K}}(v)$.

To increase the reliability of coordinate descent, multiple evaluations of each point of the transformation space are in order. However, each further evaluation contributes to increasing the total time necessary to solve scheduling. The implementation of Droplet Search that we analyze in Chapter 4.6 performs three evaluations of each point visited in the process of solving scheduling. Three evaluations are the default sampling procedure adopted by AutoTVM. We use Student’s Test (following Levine [80]’s implementation) with significance level $\alpha = 0.05$ to compare the two populations of three samples each². In other words, the search stops if we reach a transformation vector v with no neighbor that yields a faster kernel configuration with a confidence level of 95%.

4.4 Synchronous Parallelism and Speculation

The algorithm in Figure 4.1 keeps a current *candidate* transformation vector, which is initialized with the origin of the optimization space in Line 02 and is updated in Line 13 whenever a faster kernel configuration is found. To find a faster configuration, every point in the neighborhood of the current candidate is considered. The evaluation of these points happens in parallel, as Lines 09 and 10 of Figure 4.1 shows. However, we have observed that it is often difficult to fill up every available thread with unvisited kernel configurations. Example 13 explains how this difficulty emerges.

Example 13. *Figure 4.4 shows how parallel evaluation of kernel configurations happens. Points in the neighborhood of the current candidate are evaluated in synchronous batches. Figure 4.4 represents these points as gray boxes. Usually, there will be a non-empty*

²Previous work has observed that speedups due to compiler optimizations may not follow a normal distribution [5]. Student’s Test is parametric; hence, not recommended for non-Gaussian distributions. However, non-parametric tests also have shortcomings. In particular, they tend to require more samples. As an example, the minimum recommended number of samples for Wilcoxon’s [142] non-parametric test would be five, under a confidence level of 95%.

intersection of configurations between the neighborhood of the current candidate and the next candidate. Intersecting points will be stored in the *visited* set seen in Figure 4.1. In this example, intersecting points are marked with gray numbers in Figure 4.4 (b) and (c). Overlaps reduce parallelism: only five points are evaluated concurrently; thus, resources will be underutilized in processors with more than five cores.

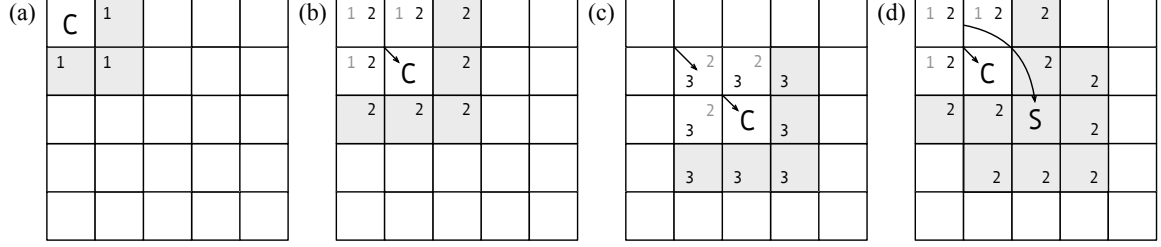


Figure 4.4: (a-c) Progress of Droplet Search without speculation. Each number shows the order in which points are evaluated. Gray boxes denote points currently evaluated. Gray numbers show points in the neighborhood of the current candidate that was already evaluated. (d) Differential speculation: a subset of the extended neighborhood of the current candidate is evaluated to maximize thread occupancy.

Differential Speculation. We resort to *speculation* to maximize thread occupancy. Speculation, in the context of this work, is the evaluation of kernel configurations in the *extended neighborhood* of the current candidate. The extended neighborhood is formed by the neighbors of neighbors. The algorithm in Figure 4.1 determines these extra points—the *Speculative Set*—in Line 07. This set is built via differential speculation: the history of previous coordinates of the best candidates determines a speculative set as follows: Let v_{n-1} be the candidate at iteration $n - 1$ of coordinate descent, and let v_n be the candidate at iteration n . By Definition 4, v_{n-1} and v_n differ in only one dimension, *e.g.*: $v_{n-1} = \langle \dots, \mathcal{P} : x_{n-1}, \dots \rangle$ and $v_n = \langle \dots, \mathcal{P} : x_n, \dots \rangle$. We choose a new transformation vector $v_s = \langle \dots, \mathcal{P} : x_n + s, \dots \rangle$, where $x_n + s$ is the value that follows x_n within the parameter $\mathcal{P}$. By the nature of Coordinate Descent, it is likely that v_s is not in the visited set. Nevertheless, if the centroid of the speculative set is in the *visited* set, then Line 07 of Figure 4.1 still ensures that multiple evaluations will not happen. Given v_s , we let the speculative set be its neighborhood. Line 07 of Figure 4.1 adds this set to the batch of configurations waiting to be evaluated.

Example 14. Figure 4.4 (d) provides some idea of how speculation works. We let S be the centroid of a speculative set. This kernel configuration, S , is chosen by extending the overall direction of the coordinate descent path speculatively: an action represented by the longer arrow in Figure 4.4 (d). In this example, speculation increases the number of active threads from five to ten.

Even though we restrict speculation to the extended neighborhood of the current candidate, this technique can still cause the line search of coordinate descent to leave a convex region. However, Section 4.11 shows that such event is not happening in practice. Had we adopted larger speculation steps, *i.e.*, outside the extended neighborhood of the current candidate, then the risk of leaving the convex region would be higher.

4.5 Concluding Remarks

This chapter provides an overview of the Droplet Search algorithm, emphasizing the significance of neighbor interactions within the convexity space search and its stop criterion, alongside discussions on synchronous parallelism and speculation. Serving as a precursor to the subsequent chapter, we aim to compare Droplet Search with AutoTVM and Ansor across five distinct neural network models. Our comparative analysis seeks to evaluate Droplet Search’s effectiveness across diverse neural network architectures and optimization scenarios, laying the groundwork for insights into optimization techniques and their adaptability across varied contexts.

4.6 Evaluation

This section compares Droplet Search with similar techniques employed in the implementation of Apache TVM. To this effect, this section investigates the following research questions:

- RQ1:** Can we demonstrate the Droplet Expectation in different architectures, at least for transformation vectors involving a small number of dimensions?
- RQ2:** How effective is Droplet Search on end-to-end models compared to other search techniques on different computer architectures?
- RQ3:** What is the average number of samples that Droplet Search takes to converge to optimal results, compared to other search techniques?
- RQ4:** Does the Droplet Search converge to a global optimum when applied to an industrial-quality analytical cost model?

RQ5: How does the number of threads used in Droplet Search influence the convergence time of the algorithm and the quality of the model that it finds, with and without speculation?

RQ6: How do kernels tuned via AutoTVM and Ansor compare to hand-written implementations of kernels in TensorFlow?

RQ7: How does the behavior of Droplet Search vary, in terms of quality and speed, on each individual kernel that constitutes a machine-learning model?

RQ8: What is the impact of the confidence level on the convergence rate of Droplet Search in terms of search speed and quality of the final model?

Hardware. This section evaluates scheduling approaches on six computer architectures, which Figure 4.5 enumerates. This mix contains two general-purpose desktop architectures (AMD and Intel); two embedded system-on-chips (ARM) and two graphics processing units (NVIDIA).

Device	Arch	ISA	Clock Max (GHz)	Memory			
				Ram (GB)	Cache L1 (KiB)	Cache L2 (MiB)	Cache L3 (KiB)
AMD R7-3700X	x86-64	x86	4.4	64	256	4	32
Intel i7-3770	x86-64	x86	3.9	16	32	2.56	8
Cortex-A7	ARMv7-32	arm	2.0	4	32	2	-
Cortex-A72	ARMv8-A	arm	1.5	4	80	4	-
RTX 3080	Ampere	ptx	1.7	12	-	-	-
GTX 1650	Turing	ptx	1.6	4	-	-	-

Figure 4.5: The architectures evaluated in this work.

Software. This section uses `Apache TVM v0.11.1`, released on March of 2023. We implemented Droplet Search as part of AutoTVM. AutoTVM also provides four other search techniques: grid, random, genetic (GA) and simulated annealing (XGB). XGB is described by Chen et al. [34]. This technique uses a cost model to guide simulated annealing. The constants that define this cost model are learned during a training phase. We also compare our implementation of Droplet Search with AutoScheduler (Ansor) [157], also available in `Apache TVM v0.11.1`.

Benchmarks. This section evaluates kernel scheduling on five convolutional neural networks and on one encoder stack of transformers (BERT). Figure 4.6 enumerates the models. The schedulers in AutoTVM, including Droplet Search, use the same optimization parameters. We have not chosen these parameters: they are pre-defined—per model—in the distribution of AutoTVM. Optimization parameters differ depending on the computer

architecture. In the CPUs, these parameters regard two optimizations: *tiling* and *unrolling*. In the GPUs, they concern two more: *thread blocking*: the ability to partition CUDA threads into blocks, and *shared memory tiling*: the ability to bring data from global to shared memory. All the parameters are *divisors* of boundaries of the iteration space. For instance, the size of a tile window must be a divisor of the size of the iteration space along the tiled dimension. This fact removes discontinuities from the search space (code that the compiler inserts to handle boundary conditions), as Section 4.10 will explain.

Arch	Auto-TVM									Auto-Schedule					
	x86			ARM			Cuda			x86		ARM		Cuda	
Network	S	P	K	S	P	K	S	P	K	P	K	P	K	P	K
Resnet-18	10K	5	12	0.5M	12	28	0.3T	9	16	27	24	43	24	39	24
vgg-16	11.3K	4	9	0.9M	12	27	2.3T	9	18	26	18	43	18	37	18
mobilenet	16.7K	5	18	0.6M	13	38	0.2T	8	19	26	22	40	22	37	22
mxnet	21.4K	5	20	-	-	-	0.4T	9	24	26	25	-	-	39	25
inception_v3	76.1K	5	43	1.0M	12	92	0.5T	9	50	26	56	41	56	37	56
bert	12.5K	4	7	-	-	-	2.7T	5	6	30	9	-	-	32	9

Figure 4.6: The machine-learning models evaluated in this work. “S” is the number of possible configurations formed by a given number “P” of transformation parameters. The largest neighborhood explored by Droplet Search at any time contains $2 \times P + 1$ points. “K” is the number of kernels that form each model. The implementation of the models vary according to the architecture. TVM’s implementation of MXNet does not run on the ARM boards.

Ansor uses more optimizations than AutoTVM. Whereas AutoTVM is restricted to a single transformation vector (in TVM’s parlance, a *kernel template*), Ansor creates several of them. The extra templates effectively give Ansor access to inter-kernel optimizations, such as loop fusion and fission: it can merge operators (*i.e.*, kernels) in the computational graph, for instance. Figure 4.6 shows the number of parameters in the largest template that Ansor explores for each network. The optimizers used in AutoTVM—Droplet Search included—are restricted to intra-kernel optimizations. Nevertheless, we hope to demonstrate that even accessing a smaller pool of optimizations, Droplet Search can be competitive with Ansor, which resorts to more extensive code transformations.

4.7 RQ1 – The Droplet Expectation

Definition 1 specifies a behavior likely to characterize typical implementations of linear kernels. This expectation is not a guarantee; thus, while we anticipate to find a path from origin to optimum along which performance improves gradually, such a path might

not exist. In this case, Droplet Search will be stuck on a local minimum. Nevertheless, this section provides some empirical evidence that the Droplet Expectation holds. Section 4.8 provides further evidence: the best kernel configurations found by coordinate descent are similar to the best configurations found via more extensive techniques, even when speculation is enabled.

Methodology. We have analyzed the behavior of two kernels: matrix multiplication and convolution on six architectures. In this experiment, we use two-dimensional spaces for the sake of visualization. On CPUs (ARM or AMD), the transformation space is formed by the tiling of the two innermost loops of each kernel. Tile sizes, in both dimensions, are 0, 8, 16, 24, $\dots$, 128; hence, we have a 17×17 transformation space. On GPUs, the transformation space is formed by the number of threads in two-dimensional thread blocks. The possible number of threads are 1, 2, 4, $\dots$, 32, also forming a 17×17 transformation space. Convolution uses a $1,024 \times 1,024$ matrix with a 3×3 filter. Multiplication uses the following matrix sizes: $1,000 \times 700$, 700×800 , and $1,000 \times 800$.

Discussion. The Droplet Expectation holds in every one of these 2×6 scenarios. Figure 4.7 shows representations for nine of these spaces, highlighting the path taken by coordinate descent. Some configurations involving thread blocking on the GPU are not valid; thus, they are not evaluated. In every scenario, the optimal kernel configuration is close to the origin, as the paths in Figure 4.7 emphasize. The Droplet Expectation holds if we increase the number of dimensions in the transformation space, or the number of kernels in the model. Section 4.8 discusses this experiment.

4.8 RQ2 – End-to-end Effectiveness

The goal of any kernel scheduling technique is to find efficient implementations of computational graphs involving kernels. This section investigates how Droplet Search fares in such task, compared with other scheduling techniques, when optimizing well-known machine learning models.

Methodology We evaluate six end-to-end models on six architectures, with a hard limit of 10,000 evaluations per search technique. This—arbitrary—limit prevents experiments from running for too long. The Droplet Search will stop after evaluating 10,000 kernel configurations; however, it tends to stabilize before (Convergence analyses are given in supplementary material, in Section 4.14). The other search techniques do not have a

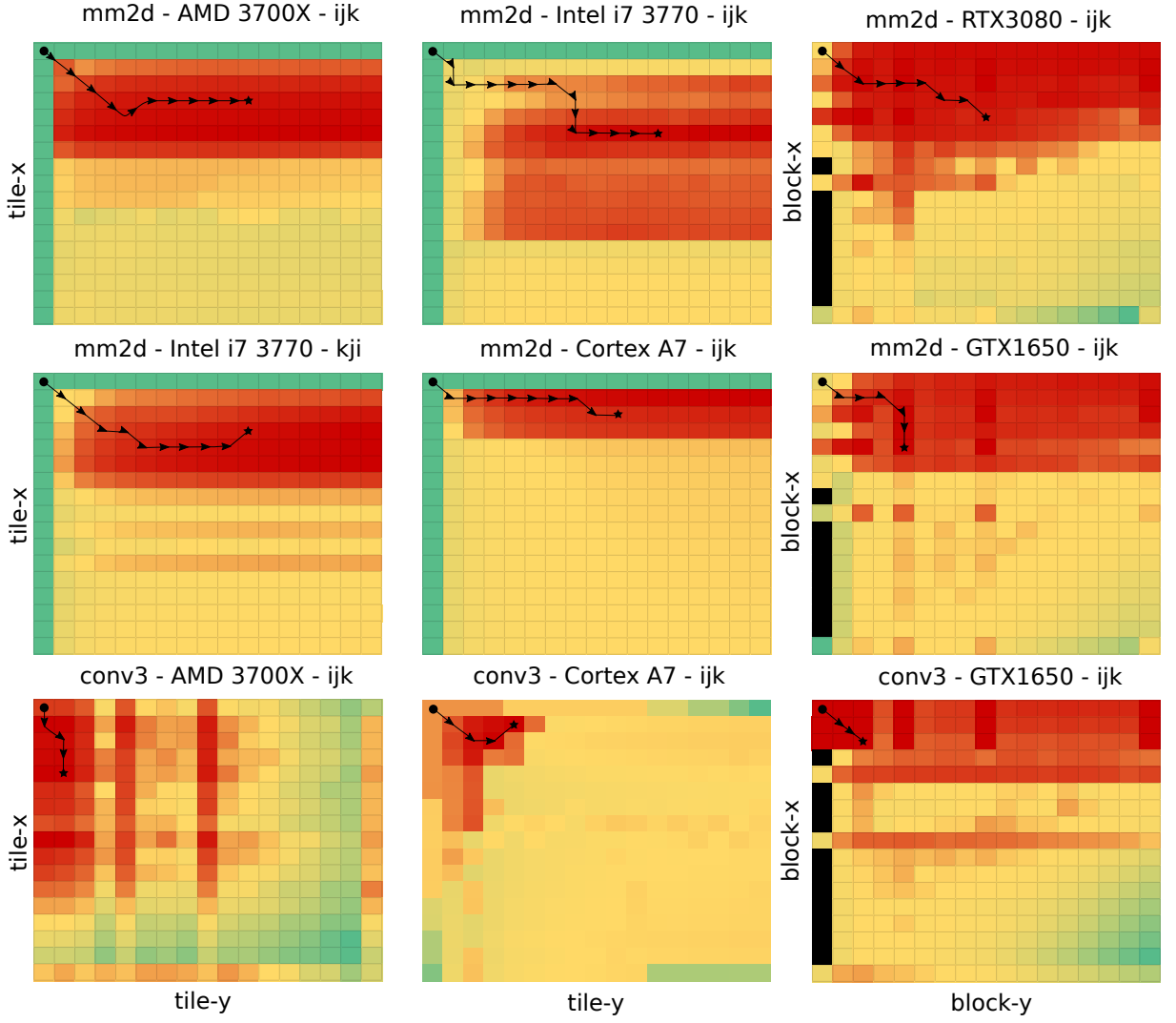


Figure 4.7: Visual representation of nine different 17×17 transformation spaces, showing the path traversed by coordinate descent from origin to global optimum. Matrix multiplication is `mm2d`, and convolution is `conv3`. Performance improves from green to red. Black cells denote invalid configurations. We show, on top of each grid, the nesting order of loops, where the canonical configuration starts with the nested sequence `ijk`.

notion of premature convergence; hence, they evaluate 10,000 configurations. Notice that Figure 4.6 shows that every transformation space contains more than 10,000 configurations. All the search algorithms evaluate kernel configurations in parallel. The parallelism in Droplet Search follows the ideas from Section 4.4. We use a confidence interval of 95% when comparing a candidate kernel configuration with kernels within its neighborhood. Section 4.9 discusses the impact of this last choice.

Discussion Figure 4.8 compares the effectiveness of different search techniques. Considering only the search techniques in AutoTVM, Droplet Search generally yields the best kernels or ties for the best (usually with XGB). Its search time, however, is faster. On the AMD CPU, for instance, Droplet Search optimizes VGG-16 in 10% of the time

Hardware	Model (ms)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)
resnet-18	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	24.48	15.15	25.4	15.31	191.1	15.66	194.1	15.80	194.6	15.38	198.1	15.63	217.0
IntelI7-3770	71.40	45.74	22.7	45.79	184.9	45.75	196.6	45.82	196.1	45.79	210.7	45.74	213.9
Cortex-A7	867.02	640.55	31.0	672.12	300.4	650.62	385.6	641.41	348.4	-	-	-	-
Cortex-A72	276.41	102.39	188.3	125.36	327.7	115.09	435.1	113.93	382.3	105.68	422.8	111.22	1.1k
RTX3080	0.76	0.53	130.6	1.75	199.5	0.63	195.2	0.53	219.9	0.53	349.9	0.70	189.0
GTX1650	3.93	2.35	88.7	5.72	204.6	2.41	192.2	2.13	191.5	2.03	450.7	2.53	202.7
vgg-16	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	243.28	127.03	16.0	128.96	189.9	128.24	199.8	127.55	195.8	129.59	204.2	122.82	212.7
IntelI7-3770	661.00	399.49	26.5	400.05	286.3	399.60	317.6	399.99	306.9	401.26	333.5	398.50	342.6
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	1.57k	0.94k	375.1	1.05k	651.5	1.51k	752.7	1.68k	769.8	1.07k	659.8	0.97k	1.2k
RTX3080	2.42	1.92	121.9	3.63	219.3	2.10	198.1	1.95	211.7	1.94	489.2	2.93	208.6
GTX1650	12.00	10.29	117.8	18.94	381.3	10.49	274.9	10.48	238.4	10.28	693.3	15.89	228.2
mobilenet	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	15.32	6.06	38.7	6.70	194.8	6.64	222.8	6.48	206.0	6.40	212.2	6.04	203.8
IntelI7-3770	28.56	17.10	38.9	17.50	354.8	17.18	420.0	17.40	412.3	17.42	427.7	15.05	171.0
Cortex-A7	342.58	293.68	42.1	296.22	266.6	295.05	368.4	293.21	347.0	-	-	-	-
Cortex-A72	240.45	84.76	144.7	241.74	414.1	208.81	532.9	253.56	490.3	261.23	1.3k	55.56	791.4
RTX3080	0.44	0.41	127.0	1.39	200.6	0.49	159.1	0.43	201.0	0.42	284.4	0.27	169.5
GTX1650	1.98	1.42	103.4	9.03	186.6	1.60	159.6	1.50	209.7	1.40	386.0	1.24	173.0
mxnet	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	58.37	33.18	46.2	34.55	193.7	33.42	208.2	33.17	204.7	33.18	205.0	34.30	214.3
IntelI7-3770	163.91	99.07	42.5	99.98	186.9	100.47	219.4	100.66	207.5	100.40	214.3	100.44	187.1
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	-	-	-	-	-	-	-	-	-	-	-	-	-
RTX3080	2.03	1.57	128.9	8.02	184.4	1.83	179.9	1.62	207.9	1.57	337.0	1.42	184.6
GTX1650	10.76	6.79	137.8	60.00	185.8	7.30	183.8	6.97	206.9	6.75	188.4	6.15	188.4
inception_v3	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	85.29	56.95	110.7	57.47	200.5	55.03	207.4	57.99	193.5	55.35	208.7	52.46	212.1
IntelI7-3770	247.59	153.89	110.5	194.44	196.4	157.57	217.2	164.65	194.5	155.50	222.4	150.20	188.4
Cortex-A7	3.42k	2.43k	123.8	2.73k	310.3	2.63k	403.6	2.58k	382.1	-	-	-	-
Cortex-A72	928.34	408.25	260.7	503.07	286.0	439.94	409.5	422.54	376.5	408.26	441.4	430.87	868.9
RTX3080	4.22	4.12	175.8	40.73	189.7	4.43	201.9	3.85	224.1	3.59	364.1	2.72	177.2
GTX1650	26.84	13.62	165.4	307.99	171.9	17.77	202.1	15.32	212.0	15.45	488.6	11.59	183.4
bert	No opt	Droplet		gridsearch		Random		GA		XGB		Ansor	
AMD3700X	205.64	119.37	31.5	120.37	156.3	121.32	205.8	121.70	164.6	121.83	177.1	85.79	60.5
IntelI7-3770	388.74	223.35	97.1	229.73	177.6	233.78	241.6	227.11	225.9	231.96	213.3	212.79	108.5
Cortex-A7	-	-	-	-	-	-	-	-	-	-	-	-	-
Cortex-A72	-	-	-	-	-	-	-	-	-	-	-	-	-
RTX3080	12.89	5.26	119.1	20.14	151.4	5.79	129.7	5.50	158.2	5.28	207.2	3.36	172.3
GTX1650	30.76	19.02	128.0	44.52	155.9	19.84	142.3	21.80	182.6	19.38	269.5	14.08	246.9

Figure 4.8: Comparison of kernel scheduling techniques. “Model (ms)” is the running time of the best schedule for a given model. “Search (min)” is the time to find that configuration. Light-gray boxes denote statistically similar results (with a confidence level of 95%). Black boxes denote statistically significant best kernel times. Gray boxes with white fonts denote statistically significant best search times. Double borders denote the best schedule produced by an algorithm from AutoTVM only (*i.e.*, Ansor is not considered).

the other scheduling approaches require. Ansor tends to produce faster kernels than the techniques implemented in AutoTVM, including Droplet Search. Ansor’s explores more optimization parameters: it has access to a large number of kernel templates, whereas

AutoTVM uses only one. Nevertheless, due to excessive memory consumption, Ansor (and also AutoTVM’s XGB) could not be used to schedule our largest models in the ARM boards. We have observed that Droplet Search does not perform well on the GPUs. In this case, the size of the search space (from 200M to 2.7T configurations, as seen in Figure 4.6) forces coordinate descent to converge due to the limit of iterations in every model, except ResNet-18. Incidentally, on ResNet-18 Droplet Search produces faster kernels than Ansor in any GPU.

4.9 RQ3 – Stop Criteria

As Section 4.3 explains, our current implementation of Droplet Search stops once it reaches a candidate kernel configuration faster than all its neighbors. Statistical significance between runtime differences is determined via Student’s T-Test applied over two populations consisting of three samples each, with a confidence level of 95%. Yet, measurements might fluctuate, and premature termination is possible. If we tighten the confidence level, termination might happen too early. If we lose it, convergence can take too long, and coordinate descent might visit configurations that are not statistically significantly faster. This section investigates how Droplet Search fares once we vary the confidence level for comparing kernel configurations.

Methodology We evaluate the five deep-learning models listed in Figure 4.6 on the Intel i7 using five different levels of confidence: 99%, 95%, 90%, 75%, and no test. In the latter case, we use the absolute arithmetic average of three samples to determine which kernel is faster.

Intel i7-3770	Droplet									
	p = 0.01		p = 0.05		p = 0.10		p = 0.25		no p-value	
	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)	Time (ms)	Tuner (min)
resnet-18	45.74	42.1	45.75	42.3	45.90	42.8	45.96	43.1	45.93	44.3
vgg-16	401.39	47.6	401.37	48.4	401.18	49.1	401.67	49.7	401.68	49.7
mobilenet	16.80	70.1	17.10	70.2	17.20	72.1	17.25	74.4	17.35	77.6
mxnet	99.07	92.9	99.07	93.8	100.72	95.6	101.13	96.9	100.59	97.6
inception_v3	153.45	172.0	153.89	173.8	153.52	176.6	153.58	176.9	154.31	181.3

Figure 4.9: Effect of confidence level (the stop criterion of Section 4.3) on the quality of the best kernel configuration and on the running time of Droplet Search.

Discussion We could observe almost no variation in the quality of the best kernel configuration depending on the confidence level. This result seems to indicate, at least for

the five models running on the Intel i7 CPU, that there exists a number of “acceptable best” kernels with very similar dynamic behavior. However, the search time increase—albeit slightly—once we move from high confidence levels towards low confidence levels. This growth in search time happens because more kernel configurations tend to be visited by the search procedure.

4.10 RQ4 – Analytical Models

Analytical cost models are systems of equations that predict the cost of a program (CPU cycles, I/O operations, cache misses, *etc.*) given a model of the hardware. Recently, different research groups have designed analytical cost models to estimate the performance of machine-learning kernels [153, 133, 102, 103, 136]. This section investigates if the droplet expectation also holds in some of these models.

Methodology This section evaluates cost models taken from three different sources. The first models were proposed by *Olivry et al.* [103]. They estimate a lower bound for the amount of data movement between slow and fast memories. The second model is part of the Xtensa Neural Network Compiler (XNNC), from Cadence Tensilica, and was made available to us through the Cadence Academic Network³. This model estimates the number of execution cycles that a program compiled by XNNC takes to execute on Cadence DSPs. Finally, we analyze the eight analytical models listed in Table II of Renganarayana and Rajopadhye et al. [116]’s work.

4.10.1 Olivry et al.’s Cost Models

We evaluate the Olivry et al.’s model on the tiled version of matrix-matrix multiplication in the authors’ original work (Listing 1 [103]). This program is the default example in Olivry et al.’s online tool⁴. We evaluate it in a system with two caches, with 64KB and 256KB—the dimensions of the first two cache levels used in the Intel i7.

³https://www.cadence.com/ko_KR/home/company/cadence-academic-network/university-program.html

⁴Available at <https://iocomplexity.corse.inria.fr/ioub> on June 2023.

Discussion Figure 4.10 shows hypersurfaces produced by the cost models generated via Olivry et al.’s online tool. Each figure relates the size of a bidimensional tiling window with the I/O cost in terms of memory transfers. The lower the cost, the faster the program is expected to run. Figure 4.10 (a) assumes one level of cache (with 64KB). The other two figures assume two levels (64K and 256KB). Figures 4.10 (b) and (c) are similar to the surfaces seen in Figure 4.2, which explore the same program, albeit on an actual machine. In the three figures, the droplet expectation (Definition 1) holds. This result is not a coincidence: Olivry et al.’s cost model involves only positive quantities (domain and coefficients); thus, form convex surfaces.

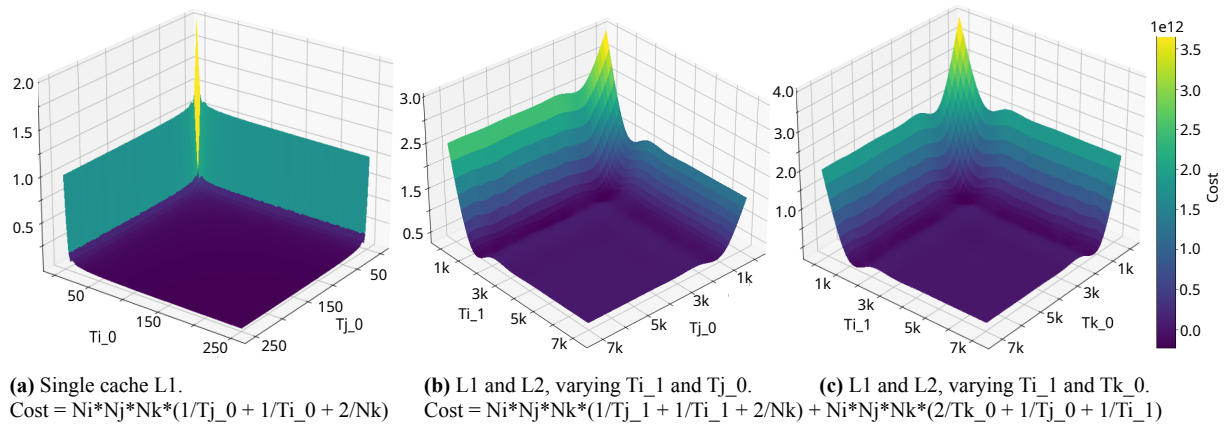


Figure 4.10: Amount of memory transfers produced by Olivry et al. [103]’s cost model applied onto Listing 1 in Olivry et al.’s work: a tiled version of the kernel in Figure 2.2-b in this work. We vary tiling dimensions T_{i_0} , T_{i_1} , and T_{k_0} . The input matrices have sides $N_i = N_j = N_k = 10K$.

4.10.2 Tensilica’s Cost Models

We evaluate the Tensilica model on an implementation of the tiled ReLU kernel on two digital signal processors, called P1 and P6. We chose these two processors because they are used as tests in the Tensilica tool. The DSPs do not have a cache; however, they contain local memory and system memory. Hence, the compiler must implement direct memory access (DMA) transfers from the system to local memory, and tiling reduces DMA operations.

Discussion In contrast to *Olivry et al.*’s cost model, the equations produced by XNNC take into consideration the fact that the tiling window might not be a perfect divisor of the loop’s iteration space. If tiling does not perfectly divide the iteration space, then the

XNNC compiler generates epilogue code to fetch data outside the tiled loop. Consequently, the performance model contains conditionals. For instance, Figure 4.11 (a) was produced by a—simplified—equation like $\text{cost} = C_1 + (\text{if } W \% 63 \text{ then } C_2 \text{ else } 0) + C_3 \times (H/64)$. The coefficients C_i represent costs of particular instructions; W and H are tile sizes. Due to these conditionals, the cost models are represented by discontinuous functions. In this case, the droplet expectation does not hold.

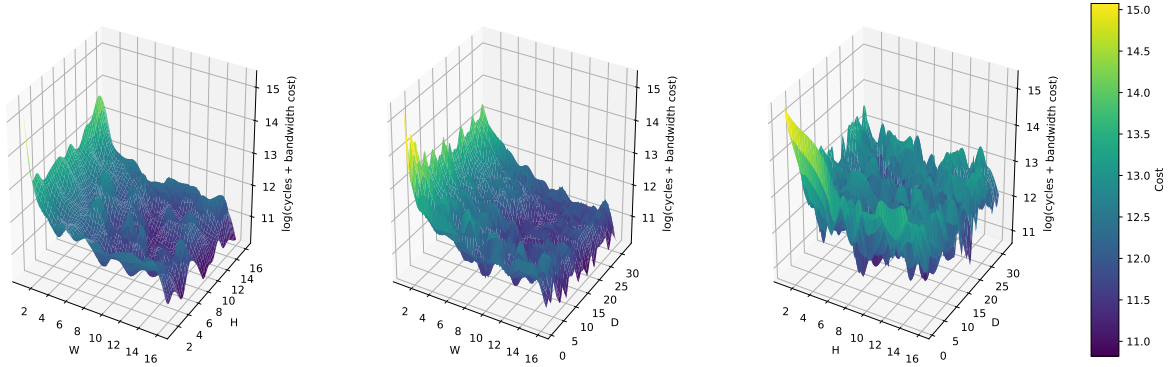


Figure 4.11: Part of the tiling search space generated for a ReLU layer by the XNNC’s analytical cost model for the P1 digital signal processor. The higher the cost (yellow), the slower the kernel is expected to behave, when deployed onto the actual hardware. To emphasize the non-convex regions, we show only the first 16 sizes of possible tiling windows.

However, if we restrict valid neighborhoods to only perfect divisors of the iteration space, then the Droplet Expectation holds. For instance, starting with tiling windows with size 16 or greater, coordinate descent achieves the optimal configuration in any of the surfaces seen in Figure 4.11 after three iterations. Notice that this restriction is unnecessary if some loss is acceptable. When applied to large deep-learning networks—under the same XNNC analytical model—Droplet Search stays very close to the global optimum, although sampling less than 1% of the space covered by an exhaustive grid search. Figure 4.12 shows these results using four deep-learning networks⁵. In contrast to the simplified setup adopted to compose Figure 4.11, these analytical models use several parameters, admitting the following optimizations: unrolling and tiling.

4.10.3 Renganarayana and Rajopadhye et al.’s Cost Models

Renganarayana and Rajopadhye et al. show that models used to solve the “*Tile Size Selection* (TSS) Problem” are represented by equations whose coefficients and domains are

⁵XNNC did not provide an analytical model for MXnet or Bert, which we evaluate in the rest of this section.

Network	DSP	Initial Cost (E+06)	Speedup Droplet	Speedup Grid Search	Avg. Steps Droplet	Avg. Grid Size
mobilenet	P1	4.93	5%	8%	28	18848
vgg-16	P1	22.4	8%	9%	34	11424
inception_v3	P1	4.84	2%	3%	32	6540
resnet-18	P1	11.1	16%	17%	23	20181
mobilenet	P6	3.95	3%	5%	23	15760
vgg-16	P6	13.6	18%	18%	31	8064
inception_v3	P6	1.59	17%	19%	18	14268
resnet-18	P6	0.62	16%	16%	33	44800

Figure 4.12: Performance achieved by Droplet Search and exhaustive grid search on deep-learning models, as estimated using Tensilica’s XNNC cost model. The grid size is the size of the transformation space.

all positive quantities. To support their observation, they list equations taken from eight different analytical models [30, 36, 52, 76, 93, 91, 144, 120]. These equations all represent instances of the bidimensional tile-size selection problem. They use variables that range on the following quantities:

C: the size of the cache in the target computer architecture;

L: the length of the cache line;

h: the height of the rectangular tiling window;

w: the width of the rectangular tiling window; and

n: the side of an $n \times n$ array.

In this section, we fix **C**, **L**, and **n**, and plot the hypersurface formed by **h** and **w**, within a contiguous range of values. This approach simulates Apache TVM’s grid search algorithm.

Discussion. Figure 4.13 shows hypersurfaces for the different equations analyzed by Renganarayana and Rajopadhye et al. To ease visualization, we remove the **h** and **w** axes (all ranging on the interval $[1, \dots, 100]$). The Droplet Expectation holds in all these analytical models. In fact, all of these equations use exclusively positive coefficients; hence, yielding convex surfaces.

4.11 RQ5 – Parallelism and Speculation

As explained in Section 4.4, the implementation of Droplet Search evaluated in this work uses synchronous parallelism and speculation to speed convergence up. This

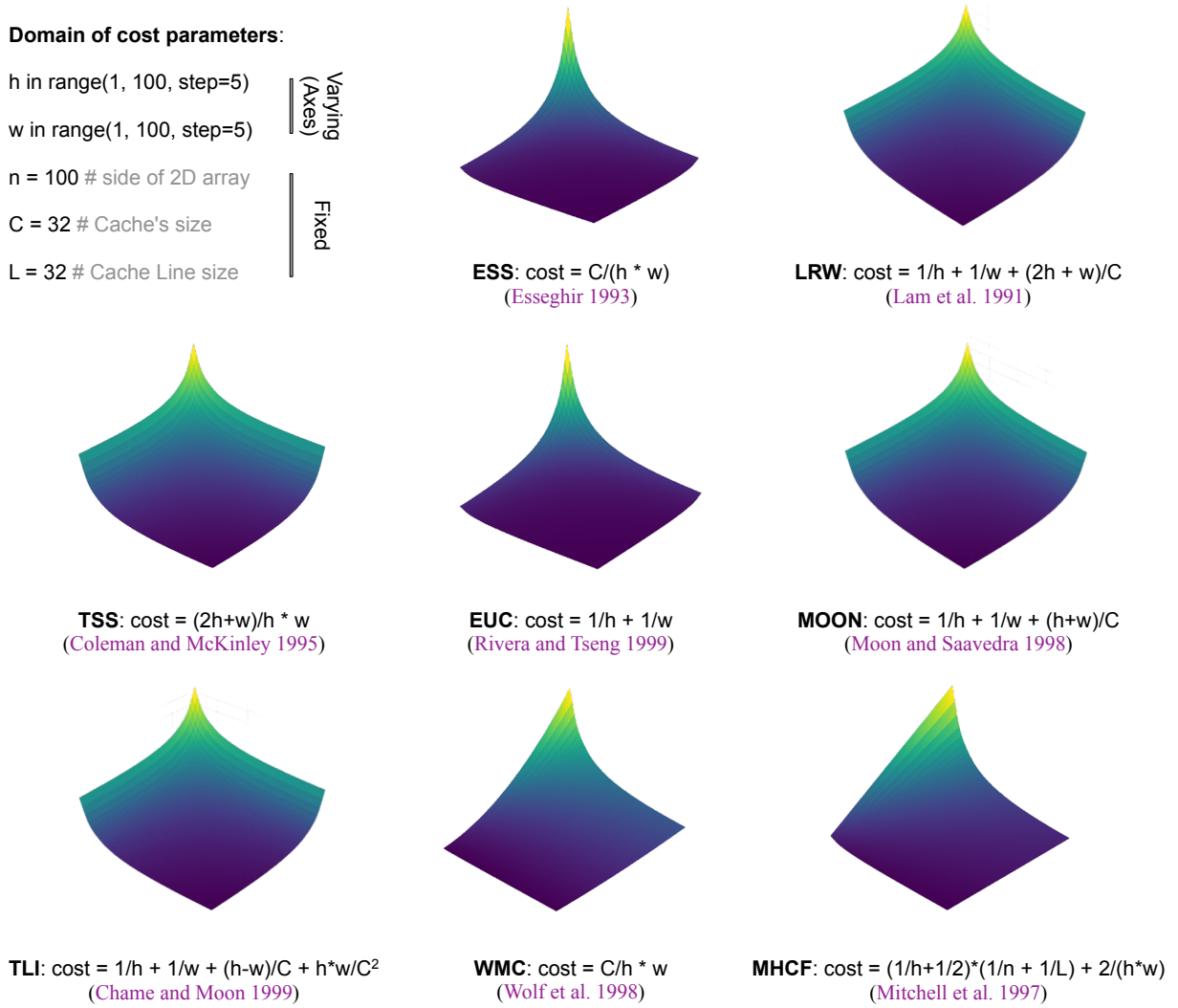


Figure 4.13: Hypersurfaces formed by the models in Table II of Renganarayana and Rajopadhye’s work [116].

section evaluates the effects of these techniques on search time and on the quality of kernel configurations.

Methodology We evaluate Droplet Search on the five different end-to-end models on the AMD 3700X CPU. This CPU runs 16 threads (two threads per core, with eight cores). This section experiments with three versions of Droplet Search: its full implementation, an implementation that features parallelism but no speculation, and an implementation that runs on a single thread. We report p-values comparing three executions of the fastest models found with each approach.

Discussion Figure 4.14 compares the different implementations of droplet search. Its single-threaded implementation takes 664 seconds to converge (sum of tuning time over five networks). The parallel version, with 16 threads, converges in 358 seconds. With speculation plus parallelism, this time goes down to 291 seconds. Parallelism is a known

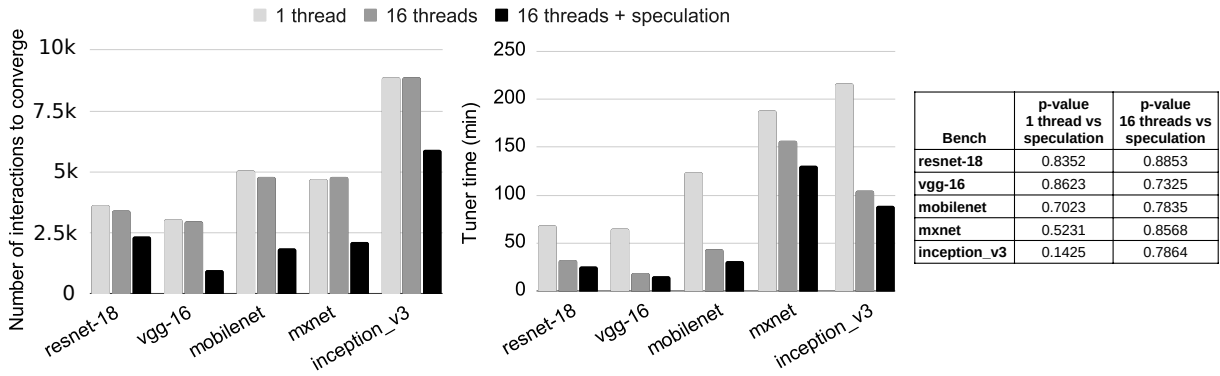


Figure 4.14: (Left) Number of iterations until convergence of variations of droplet search. (Middle) Running time of different implementations of Droplet Search. (Right) P-values produced by a T-Test on populations of three executions of the best model found by each technique. The closer to 1.0 is the p-value, the more statistically similar are the two populations.

limitation of coordinate descent. Our implementation suffers from the shortcomings mentioned by Zheng et al. [156]. The synchronous nature of coordinate descent limits concurrency: the best candidate is chosen after every point of a neighborhood is evaluated. Thus, progress only happens once the slowest point runs. Wang et al. [139] have shown that it is possible to improve parallelism if more candidate points co-exist. We believe that early preemption of slow points could also speed our implementation up: once the best candidate is found in a neighborhood, the other threads can be aborted. We leave such approaches—multiple candidates and early preemption—open for future work. Speculation improves the running time of our implementation of Droplet Search by a small margin. The parallel version of Droplet Search, with speculation, is 27% faster than the non-speculative implementation (geomean over speedups). This gain comes mostly from faster convergence.

Figure 4.14 (Right) shows that the extended neighborhood explored via speculation has no effect on the speed of the kernel configurations found via Droplet Search. The search does not find always the same final configuration for every layer of every model; however, this phenomenon is due just to statistical variations in the running time of similar kernels. As the figure shows, the p-values reported by a T-Test on the speed of the different kernels is well above 0.05. Thus, the extended neighborhood is not changing the behavior of our implementation of Droplet Search.

4.12 RQ6 – Comparison with TensorFlow

The goal of this section is to bring some perspective about our results to readers that are not familiar with the Apache TVM ecosystem. To this end, we shall present a

comparison between three different approaches to develop end-to-end models: AutoTVM, Ansor, and TensorFlow [1]. The latter is a Python-based library to write machine learning models. In contrast to AutoTVM or Ansor, TensorFlow does not do, by itself, any form of scheduling: the programmer must feed it with an optimized implementation of a machine learning model.

Methodology This section compares the different kernel implementation approaches using a benchmark collection formed by six kernels: matrix multiplication, 2D convolution, depthwise separable convolution, pooling, matrix reduction and 2D ReLU. These kernels have been taken from the artifact made publicly available by Zhu et al. [159], and are meant to run on graphics processing units. We evaluate them into our RTX 3080 GPU and on an A6000⁶. Incidentally, we have not been able to apply Zhu et al.’s tool onto these very kernels⁷.

Discussion Figure 4.15 shows the comparison between different kernel implementation systems. Notice that Figure 4.15 compares different kernel implementations. Ansor and AutoTVM (Droplet Search and XGB) receive, as input the same code: a kernel implemented in Python with libraries from Apache TVM. TensorFlow receives a different implementation: kernels also written in Python, but with libraries from the TensorFlow package. As an example, the API to invoke the ReLU (rectified linear unit) kernel is `tf.nn.relu(a.tf)` in TensorFlow, and `tvm.nn.relu(a.tvm)` in Apache TVM. In short, Figure 4.15 is comparing two different Python libraries.

In every experiment, Apache TVM has produced faster (or equally faster) kernels than TensorFlow, be it through Auto TVM, be it through Ansor. However, without scheduling, TensorFlow outperforms Apache TVM in two kernels: convolution and depthwise convolution. We have not observed a statistical difference between implementations of the reduction and the ReLU kernels, regardless of the library or the scheduling approach adopted to optimize them. In this regard, we observe that neither Ansor nor Auto TVM implement search for pooling, reduction, and ReLU. The implementation of these kernels, as provided by Zhu et al., does not come with a template of optimization parameters. The other kernels, in contrast, come with templates that enable thread blocking and tiling

⁶The A6000 GPU is only used in this section. Access to this hardware was kindly provided by the *Discovery Lab* (<https://discovery.ic.unicamp.br/>).

⁷ROLLER, the tool implemented by Zhu et al. relies on RT Cores to speedup the execution of kernels. This tool uses functions that are exclusive of Cuda v10.2 and TensorFlow v1.15.2. However, the RTX 3080—the only GPU that we have with RT cores—is only compatible with Cuda v12.0 and TensorFlow v2.13.0. We could not downgrade the version of Cuda. Furthermore, a direct change of APIs, to upgrade the versions of Cuda and TensorFlow in ROLLER was not enough to let us reuse the tool: the updated version of the tool compiles successfully; however, it does not produce kernels. We faced similar issues when trying to reuse another artifact that target RT cores: HERON [15]. HERON was implemented with the Cuda v11.0 API. We have also updated it to use Cuda v12.0. The updated version compiles and produces kernels; however, these kernels crash during execution.

Microbenchs	Model (ms)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)	Search (min)	Model (ms)
RTX3080	Original	Droplet		Autotvm (XGB)		Ansor		TensorFlow
matmul	0.01798	0.00577	2.50	0.00577	54.10	0.01266	37.96	0.24704
conv2d	414.85434	2.09061	24.38	2.18308	31.95	1.81956	72.39	43.88901
depthwise	63.78266	0.44082	44.46	0.44013	56.74	0.46296	64.67	25.03834
pooling	0.59630	0.59545	0.04	0.59525	0.04	0.96103	71.90	60.46896
reduce	0.31750	0.31340	0.02	0.31340	0.02	0.31243	2.69	9.07090
relu	0.00017	0.00017	0.02	0.00017	0.02	0.17265	0.18	4.73173
A6000	Original	Droplet		Autotvm (XGB)		Ansor		TensorFlow
matmul	0.01780	0.00626	2.61	0.00632	24.60	0.01166	39.20	0.38927
conv2d	367.64008	1.93585	60.13	1.79232	69.72	1.69408	69.72	20.15370
depthwise	52.25241	0.53334	44.05	0.53213	81.28	0.70036	67.01	14.51671
pooling	0.68824	0.68812	0.04	0.68820	0.05	1.13004	81.29	10.23547
reduce	0.38104	0.38100	0.02	0.38105	0.03	0.38019	2.97	6.10128
relu	0.00020	0.00020	0.02	0.00020	0.02	0.17265	0.38	3.76800

Figure 4.15: Comparison between Apache TVM and TensorFlow on the six kernels made available by Zhu et al. [159], on two graphics processing units. This figure uses the same notation as Figures 4.8 and 4.16: dark boxes indicate the fastest models, and gray boxes indicate the fastest search times. Light gray boxes indicate results that are statistically similar. TensorFlow does not implement search.

with shared memory. Loop unrolling is not enabled by these parameters. Nevertheless, Figure 4.15 confirms some of the results earlier observed by Zhu et al.: Kernels produced by Apache TVM tend to outperform similar kernels produced via TensorFlow. However, contrary to Zhu et al., we have observed smaller differences.

4.13 RQ7 – Intra-Kernel Behavior

The models explored in Section 4.8 consist of multiple kernels: each layer is implemented as an independent kernel. Droplet Search and all the other search techniques available in AutoTVM are intra-kernel. Thus, kernels are optimized independently from each other. This section analyzes the effects of Droplet Search on individual kernels and compares these effects with results obtained by other scheduling techniques. By showing that Droplet Search finds similar configurations as exhaustive techniques, we provide further evidence that the droplet expectation is common.

Methodology. We analyze each kernel of ResNet-18 and VGG-16 in separate, reporting search time and kernel running time. Each kernel is extracted from its encompassing model via TVM’s code generator. ResNet-18 and VGG-16 are our smallest networks, in number

of kernels. We restrict this study to only two models because the individual analysis of each kernel is time consuming (meaning human time, not machine time). Nevertheless, we believe that these results could be extrapolated to the other models, which sport similar implementations.

resnet-18	Droplet			Gridsearch			Random			GA			XGB		
Layer	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter	Time (ms)	Tuner (s)	iter
0	0.83	72.6	128	0.83	362.5	308	0.83	379.7	308	0.83	383.5	308	0.83	386.7	308
1	0.81	125.5	96	0.82	1079.2	980	0.81	1091.0	980	0.82	1098.4	980	0.82	1119.2	980
2	0.09	131.2	209	0.09	1115.8	980	0.09	1127.3	980	0.09	1128.4	980	0.09	1153.3	980
3	0.42	223.8	198	0.42	1045.9	896	0.42	1059.9	896	0.43	1052.8	896	0.42	1076.2	896
4	0.05	123.5	163	0.05	1019.4	896	0.05	1035.6	896	0.05	1033.7	896	0.05	1055.5	896
5	0.80	111.4	120	0.80	1130.1	1024	0.80	1137.3	1024	0.80	1149.0	1024	0.80	1168.1	1024
6	0.42	129.7	272	0.42	1020.4	864	0.43	1029.0	864	0.42	1026.5	864	0.42	1050.2	864
7	0.05	131.9	181	0.05	985.8	864	0.05	999.9	864	0.05	1000.4	864	0.05	1020.2	864
8	0.81	113.2	254	0.81	1086.9	972	0.81	1105.0	972	0.81	1114.9	972	0.82	1123.8	972
9	0.44	107.8	331	0.44	866.0	720	0.44	884.5	720	0.44	881.9	720	0.44	898.4	720
10	0.05	139.0	134	0.05	829.1	720	0.05	846.6	720	0.05	847.4	720	0.05	865.2	720
11	0.81	112.9	260	0.82	921.8	800	0.82	948.1	800	0.82	960.4	800	0.81	967.5	800
SUM	5.585	1522.5	2346	5.588	11463.0	10024	5.591	11643.9	10024	5.587	11677.1	10024	5.589	11884.3	10024
AVG	0.465	126.9	2346	0.466	955.3	835	0.466	970.3	835	0.466	973.1	835	0.466	990.4	835
vgg-16	Droplet			Gridsearch			Random			GA			XGB		
0	0.60	71.1	57	1.21	377.8	308	0.60	380.2	308	0.60	382.1	308	0.60	384.3	308
1	13.13	114.5	112	15.74	1095.0	1078	13.14	1116.0	1078	13.13	1110.6	1078	13.16	1148.5	1078
2	6.30	108.0	107	7.63	1163.3	1232	6.29	1212.4	1232	6.30	1201.5	1232	6.30	1242.3	1232
3	12.74	110.7	107	15.14	1380.9	1408	12.74	1439.1	1408	12.80	1413.9	1408	12.91	1473.3	1408
4	6.34	112.0	121	7.98	1347.4	1440	6.36	1457.8	1440	6.34	1392.0	1440	6.36	1478.8	1440
5	12.73	116.9	121	17.84	1611.9	1620	12.80	1731.4	1620	12.79	1682.0	1620	12.79	1762.7	1620
6	6.26	93.9	104	8.52	1419.0	1440	6.27	1533.4	1440	6.27	1482.2	1440	6.28	1565.2	1440
7	12.77	122.4	124	16.14	1742.3	1600	12.84	1830.0	1600	12.79	1791.5	1600	12.76	1874.6	1600
8	3.24	108.4	107	4.77	1258.7	1200	3.24	1286.4	1200	3.24	1291.3	1200	3.24	1325.2	1200
SUM	74.13	957.9	960	94.97	11396.2	11326	74.28	11986.8	11326	74.27	11747.1	11326	74.40	12254.9	11326
AVG	8.24	106.4	107	10.55	1266.2	1258	8.25	1331.9	1258.4	8.25	1305.2	1258	8.27	1361.7	1258

Figure 4.16: The search techniques of AutoTVM applied on individual layers of deep-learning models on the AMD R7-3700X. Evaluations average three samples. Light gray boxes mark running times that are statistically similar with confidence level of 95%. Black boxes denote statistically significant running times. Gray boxes mark statistically significant search times. The sum of search times approximates the results in Figure 4.8. The sum of kernel times, *i.e.*, “Time (ms)”, is strictly less than the running times reported in Figure 4.8.

Discussion. Figure 4.16 shows how the search techniques fare on each layer of ResNet-18 and VGG-16. We do not show results for Ansor, because, as Figure 4.6 shows, Ansor’s implementation recognizes more layers on each model. Droplet Search never yields worse configurations than the other search techniques. Furthermore, its search is faster, converging with fewer samples. In this case, column “iter” in Figure 4.16 provides the number of kernel configurations evaluated by each search technique. In contrast to Droplet Search, the other approaches used in AutoTVM do not have a notion of convergence. The search stops once a determined number of kernel configurations is visited. However, sampling is not equally divided among the kernels: AutoTVM draws more samples for kernels that run for more time. As a example, the different search procedures of AutoTVM

sample Layer Five of ResNet-18 1,024 times, as Figure 4.16 shows. Droplet Search, in contrast, stops after 120 iterations. The samples that Droplet Search evaluates depends more on the dimensions of the layer, such as the number of channels, width and height of the filter used. The four largest layers in Figure 4.16 are, in this order: the 6th, the 8th, the 9th and the 11th. Incidentally, these layers account for the largest number of samples observed by Droplet Search.

4.14 RQ8 – Convergence Rate

The convergence rate of a search mechanism used to solve the kernel scheduling problem measures how fast that technique closes on its final solution. As mentioned in Section 4.13, the different techniques that AutoTVM uses to solve kernel scheduling iterate until a fixed number of configurations are evaluated. We have observed that it is often possible to stop iterations before, once a sufficiently good configuration is reached. That is the approach adopted in Droplet Search, as Section 4.3 explains. In what follows, we investigate how the quality of the final solution to scheduling improves as the number of evaluations progresses.

Methodology We set the maximum number of iterations of AutoTVM’s grid, random, genetic and XGB search to 10,000. This number, 10K, includes the evaluations of kernel configurations in neighborhoods or speculative sets that Droplet Search uses. Ansor shall use the same limit of evaluations. We then inspect the speed of the best kernel configuration that each one of these search techniques find ResNet-18 throughout the search. We show results for ResNet-18 only; however, we have evaluated the convergence rate for the other models. Results tend to be similar.

Discussion Figure 4.17 shows the results of this experiment. Droplet Search usually converges to the best solution before 10,000 evaluations. The fastest convergence was observed on the Intel CPU: coordinate descent stabilized after 1,278 configurations were evaluated. Notice that convergence happened before 2K evaluations in every CPU. The slowest convergence happened on the GTX GPU: 7,125 evaluations. Similarly, on the RTX, 6,502 configurations were evaluated. Similar results were also observed on the other four models: Droplet Search stabilizes before 10K iterations; typically before 3K iterations on the CPUs, and before 8K iterations on the GPUs.

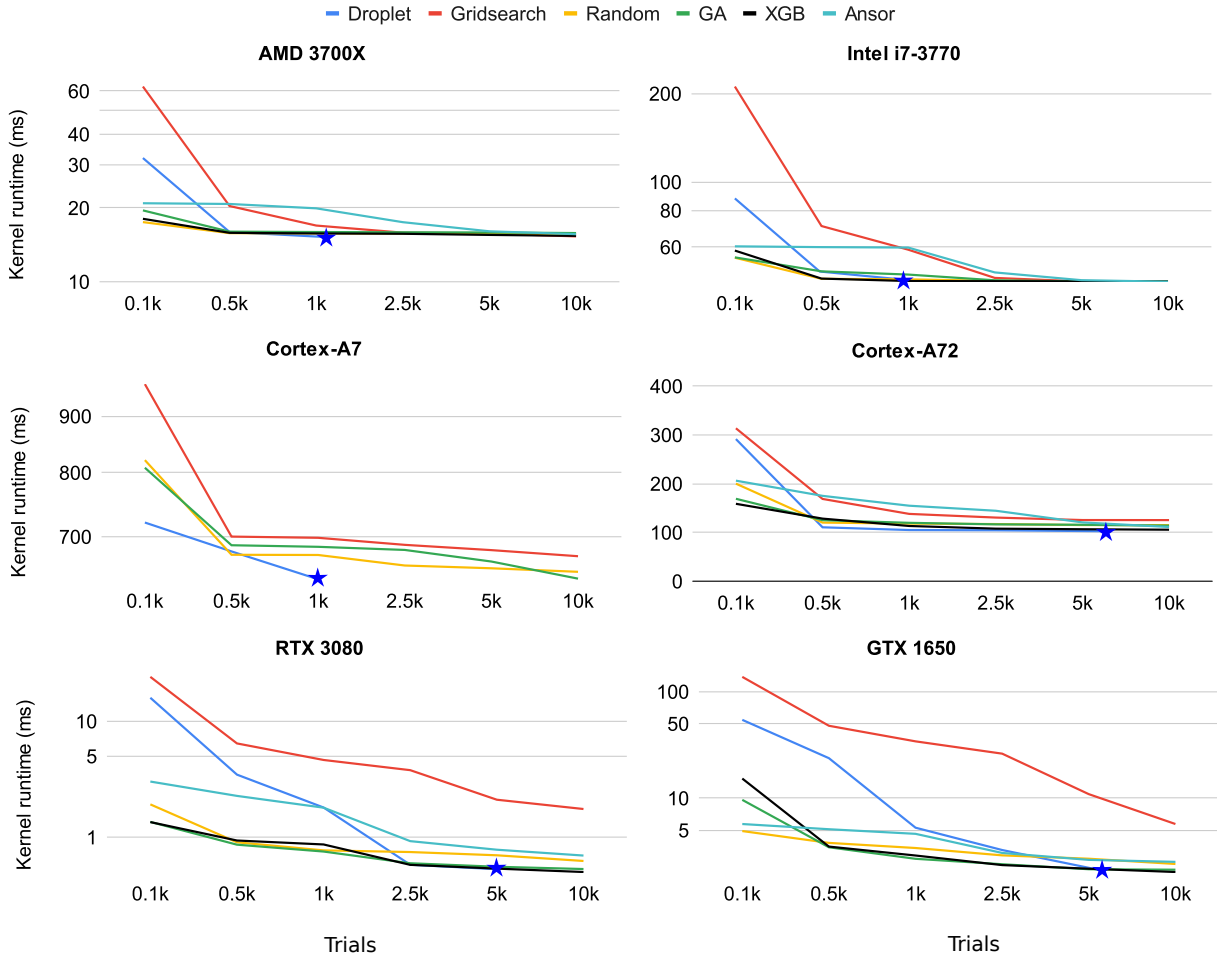


Figure 4.17: Quality of the solution for kernel scheduling versus the number of evaluations (trials) used to find that solution for ResNet-18. The blue star marks the convergence of the Droplet Search.

4.15 Concluding Remarks

This chapter provides a comparison between Droplet with AutoTVM and Ansor using five neural network models. We delve into the research questions, probing into the effectiveness of Droplet Search across diverse neural network models, and its adaptability to different optimization scenarios. The intention is to establish a link to create new insights for future work presented in the next chapter.

Chapter 5

Exploration versus Exploitation

This chapter demonstrates that using Droplet Search to exploit a reduced set of samples considered by **Ansor**, it is possible to improve upon the performance of **Ansor**. We refer to this modified version of **Ansor**, which incorporates Droplet Search, as the *Combined Approach*, henceforth denoted as **DPAnsor**, while reserving **Ansor** for the tool’s original implementation. In this new technique, **Ansor** serves as the seed for initializing parameters, with **Droplet Search** applied for focused exploitation. Seed which, being initially good, can make **Droplet Search** converge faster to an optimal solution.

5.1 Introduction

A *kernel* is an algorithm that applies operations on *tensors*: chunks of memory indexed by a linear combination of natural numbers. A *Tensor Compiler* is a compilation infrastructure that generates code for kernels. As Ansel et al. [8] explains, many tensor compilers, including TVM [33], nvFuser [123] and NNC [161], follow a design probably inspired by Halide [112]. These compilers separate the kernel’s semantics (what the kernel does) from its schedule (when the kernel does it). Since the same kernel semantics can be implemented in many different schedules, tensor compilers face a challenge called *kernel scheduling*: determining a suitable ordering for the operations a kernel performs on tensors. Kernel scheduling is typically addressed via heuristics because the *Kernel Optimization Space*—the set of all implementations of a kernel—is extremely vast, as Li et al. [84] explains.

The Apache TVM tensor compiler employs three distinct optimization infrastructures for solving kernel scheduling: AutoTVM [33], Ansor [157] and MetaSchedule [125]. AutoTVM finds parameters of *kernel sketches*. The sketch of a kernel represents the optimizations applied to the abstract description of that kernel, such as loop unrolling, splitting, interchange, and tiling. Many of these optimizations are parameterizable. Examples of parameters include the unrolling factor in loop unrolling and the width of the tiling

window in loop tiling. **AutoTVM** assigns values to these parameters using various search heuristics. One of these heuristics is part of this work: Droplet Search [25]. Droplet Search seeks the optimal configuration of an optimization template by determining a descent direction along the objective function that models the running time of the kernel. **Ansor** and **MetaSchedule** (in contrast to **AutoTVM**) have the capability to generate new sketches. In other words, these schedulers are not restricted to a single sequence of optimizations. Section 5.3 provides further details on how **Ansor** works, whereas Section 5.4 explains how Droplet Search works.

A Combined Search Infrastructure. Both **Ansor** and **MetaSchedule** typically generate higher-quality kernels compared to **AutoTVM**’s Droplet Search, as they are not limited to a single search space. Each new sketch leads to the exploration of an entirely new search space. However, when constrained to a single sketch, Droplet Search tends to outperform these schedulers. Drawing on the terminology from recent work by Ding et al. [49], Droplet Search’s coordinate descent approach is “*hardware centric*”, while **Ansor**’s genetic algorithm is “*input centric*”; better embodying this quality that Sorensen et al. [131] calls “*performance portability*”. In essence, Droplet Search, by traversing the search space contiguously, is sensitive to cache sizes and levels in the cache hierarchy. Nevertheless, despite Droplet Search’s tendency to identify optimal points within the search space, this algorithm is unable to navigate beyond this space due to its reliance on the initial sketch.

Inspired by these observations, this work addresses the following research question: “*Is it possible to combine the wide exploration phase of **Ansor**¹ with Droplet Search’s exploitation; thus, obtaining the advantages of each approach?*” By doing so, we can develop a version of **Ansor** that produces superior kernels compared to the original tool while also reducing search times. This work brings evidence that such a combination is effective. The core idea presented in this work is as follows: Initially, we allow **Ansor** to explore the kernel optimization space, leveraging its “space travel ability” to test different sequences of optimizations during this exploration. Subsequently, following this initial exploration phase, we identify the most promising kernel space discovered by **Ansor** and employ **AutoTVM**’s Droplet Search—a line search algorithm—to find a good kernel within this space.

The new search methodology that emerged out of this combination has been considered sufficiently practical to be approved into **Apache TVM**. A request for comments was submitted to the **TVM** community in December of 2023, and a patch was approved into **Ansor** in February of 2024. The patch has not yet been merged into an official release of **TVM** at the time of this submission. We have, subsequently, incorporated a coordinate descent exploitation phase into **MetaSchedule** [125], which uses different search algorithms

¹This work uses **Ansor** as the basis for experiments; however, similar results have been reproduced in the **MetaSchedule** [22]. We focus on **Ansor** because it is documented in an academic work [157].

than **Ansor**. These results are even better than those observed in **Ansor**. Consequently, a new request for comments was submitted to the TVM community in June 2024, together with a patch for **MetaSchedule** [22].

The experiments in Section 5.8 show that the combined exploration-exploitation methodology outperforms the original implementation of **Ansor** in terms of kernel quality and search speed. Positive results are reported in four different processors (AMD R7-3700X, Fujitsu ARM A64FX, NVIDIA RTX 3080, and NVIDIA A100), and in 20 popular deep-learning models, including **AlexNet**, **VGG**, **ResNet**, **MobileNet**, **Inception**, **GoogleNet**, and **DenseNet**. As an illustration, by terminating **Ansor** after sampling 300 kernels and then optimizing the best candidate with Droplet Search, we outperformed **Ansor** running with a budget of 10,000 samples in all the 4×20 architecture-model pairs. For instance, the combined search approach, on **MnasNet**, yields kernel speedups of 1.59x, 1.02x, and 1.08x on x86, ARM, and NVIDIA platforms, respectively. The search time for the combined approach correspondingly decreases by 1.25x, 1.25x, and 1.18x on these architectures.

5.2 Exploration via **Ansor**; Exploitation via Droplet Search

A kernel is an abstract concept: it can be represented by operations on memory indexed by a linear combination of natural numbers. The actual implementation of a kernel is determined by its *schedule*. In the context of this work, a schedule results from the application of code transformations onto the naïve implementation of the abstract kernel. The naïve implementation replaces each linear index with a loop. A schedule is the combination of two notions: a *sketch* and an *annotation* of the sketch. Following the terminology introduced in the original **Ansor** work [157], a sketch is the representation of a kernel that results from the application of a certain sequence of optimizations onto the naïve implementation.

Example 15. *Figure 5.1 (a) shows an example of an abstract kernel. Figure 5.1 (b) shows a naïve implementation of the abstract kernel seen in Figure 5.1 (a). Figure 5.1 (c) shows two sketches, produced after the application of different code optimizations onto the program in Figure 5.1 (b).*

Sketches are parameterized with arguments that determine how each optimization is implemented. These arguments—henceforth called *annotations*—include for instance, the number of threads assigned to a parallelizable loop, the size of the tiling window, or

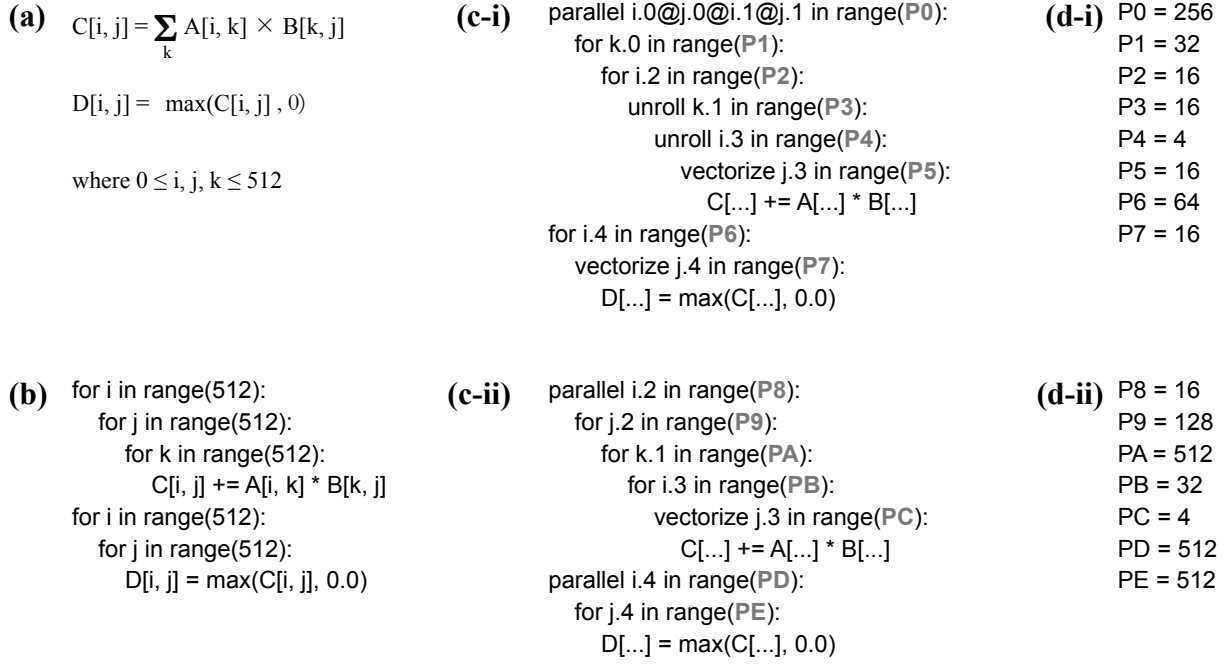


Figure 5.1: (a) Abstract view of a kernel. (b) Naïve implementation of the abstract kernel. (c) Two optimization sketches for the naïve kernel. (d) Different annotations for the sketches.

the unrolling factor of an unrollable loop. Figure 5.1 (d) shows different parameters of the sketches in Figure 5.1 (c). The set of every valid configuration of annotations for a given sketch forms the *optimization space* of that sketch. This space has one dimension for each parameter that is allowed to vary. Figure 5.2 shows two views of the optimization space of the two sketches in Figure 5.1 (c).

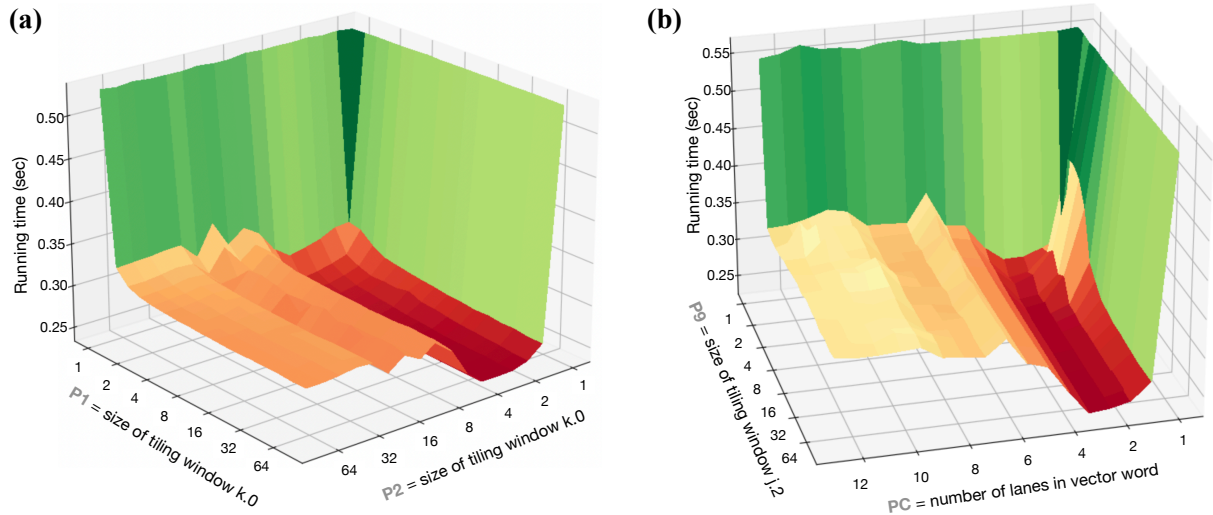


Figure 5.2: (a) A two-dimensional view of the optimization space formed by the parameters **P1** and **P2** seen in Figure 5.1 c-i. (b) A two-dimensional view of the optimization space, this time involving parameters **P9** and **PC**.

5.3 Space Exploration via Ansor

Ansor solves kernel scheduling in an interactive process that involves three phases:

Sketch generation: in this phase, new sketches are created. As explained in Definition 2, each sketch determines one kernel search space.

Sketch annotation: in this phase, an initial population of annotated sketches is created. Following Definition 2, each annotated kernel is a point in the search space determined by the sketch that provides the annotations.

Kernel evolution: in this phase, candidate kernels are sorted according to an estimation of their performance (via **Ansor**’s cost model). The best candidates are sampled (executed and timed), and this information is used to improve the cost model.

Sketch Generation The generation of new sketches happens via the application of a small collection of rewriting rules, which Figure 5.3 enumerates². Each rewriting rule provokes a code transformation, such as tiling, parallelization, or unrolling. The sketch generation rules are hardware-dependent: some rules, like those that bind loop iterations to threads, are only well-defined for GPUs, for instance. Notice that in addition to the rules seen in Figure 5.3, users can still add custom rules to the set of available sketch transformations.

Example 16. Figure 5.4 (a) shows a variation of the sketch earlier seen in Figure 5.1 (b). Each loop has an unrolling factor, which will have to be annotated in the initialization phase of the autotuning process. Figure 5.4 (b) shows an application of the “Always inlining” rule onto Figure 5.4 (a). It is important to note that neither of these program representations qualifies as a “kernel”, since they are not executable. To transform them into concrete kernels, the annotation within the unrolling factor must be assigned an actual value.

The Initialization of Annotations Sketches are not executable programs: they have annotations which must be replaced with actual values. These values are the parameters of optimizations, as Definition 2 explains. Thus, as a second step of the iterative exploration approach of **Ansor**, an initial population of annotated sketches is produced via the application of “initialization rules”. Figure 5.5 summarizes the rules available in **Ansor** at the time this work was produced. Each initialization rule is parameterized by a probability distribution, which associates concrete values with the probability that they can be chosen. Example 18 shows how these distributions are used.

²Rules implemented in https://github.com/apache/tvm/tree/main/src/auto_scheduler/search_policy on 02/01/2024

Skip state	This rule is the identity function, which does not modify the current kernel. Its existence allows more flexibility to define rewriting rules, as they do not need to have all the same number of transformations.	Add cache write	Write data into the cache, e.g., as a way to support data layout transformations such as split, fuse and multi-level tiling.
Always inlining	This rule implements a variation of classic loop fusion (although called 'inline'). Given two loops, L1 and L2, where the former produces data that the latter consumes (without loop carried dependencies), the “inlining” operation merges these two loops.	Add reduction factor (rFactor)	This rule implements a loop as a reduction whenever such is possible, which can be run in parallel. In this case, the reduction operation (e.g.: add, mul or max) must be associative.
Multi-level tiling	This rule divides a nest of loops into various levels of subloops to enhance data locality. The goal of tiling is to arrange together data that are likely to be accessed in operations that are close in time.	Cross thread reduction	This rule determines the reduction factor for the reduction operators if they are not an intermediate stage created by cache write and there is sufficient parallelism. It also verifies the kernel fusibility and executes split and bind transformations accordingly.
Simplify tensor	This rule unrolls the indices of constant tensors or tiles that have a default tile level set to 2.	Add cache read	This rule, meant for GPUs, moves data from local to shared memory. The shared memory provides much faster access than local memory. Data is brought into shared memory by independent threads in parallel.
Multi-level tiling with fusion	This rule combines different loops in a transformation with cache-write, if the hardware supports it. It then executes multi-level tiling on the fused loops.	Special compute location	This rule identifies the unrolled outer space iterator and applies some specified transformation to compute at a designated location.

Figure 5.3: Sketch generation rules. **Ansor** uses these rules to change the kernel search space. Each rule modifies a sketch, e.g., fusing, splitting or tiling loops. However, these rules do not change the annotations in the sketch.

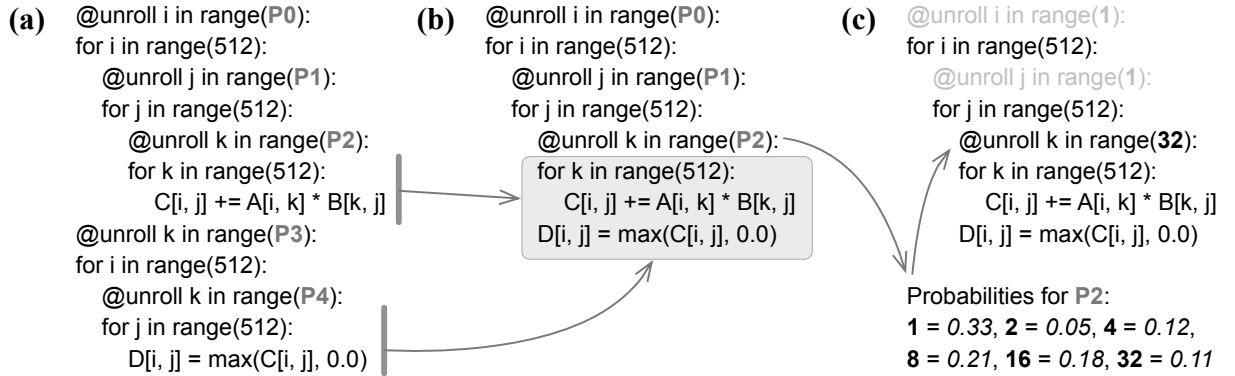


Figure 5.4: (a) Sketch of the abstract kernel seen in Figure 5.1 (a). (b) Sketch that ensues from the application of the “Always inlining” rule. (c) Sketch that ensues from the initialization of the unrolling factor.

Example 17. Figure 5.4 (c) shows the concrete kernel that comes out of the application of the “Unroll” rule (in Figure 5.5) to the sketch in Figure 5.4 (b). Figure 5.4 (c) also shows the probability distribution used to randomly choose the initial unrolling factor of 32.

Example 18. Unroll: During the initialization of unrolling, the process selects appropriate unrolling factors for iterators within a computational graph. This selection aims to optimize performance by enhancing memory access patterns and leveraging parallelism. For instance, let’s consider the predefined configuration vectors: `auto_unroll_configs_cpu = 0, 16, 64, 512` for a CPU task and `auto_unroll_configs_gpu = 0, 16, 64, 512, 1024` for a GPU task. These vectors contain potential unrolling values suitable for different hardware platforms. When applying the `InitUnroll` rule, the process randomly chooses from these

Fill tile size	This rule randomly assigns tile sizes to all tiled loops.	Unroll	This rule randomly assigns unrolling factors to the loops marked to be unrolled.
Vectorize	This rule combines and vectorizes iterations of the innermost loop. It is only applicable if memory accesses are contiguous across loop iterations.	Parallel	This rule fuses and parallelizes outermost loops. It stops upon finding reduction operations, or if the parallel annotation exceeds 16x the number of cores.
Change compute location	This rule skips all inlined, placeholder, and tiled loops. Then, it employs a random choice (0, 1, 2) to determine where computations will run.	Bind	This rule binds SIMD thread identifiers to loop iterations. By mapping computations to separate threads, the bind operation implements parallelization in graphic processing units.

Figure 5.5: Rules that **Ansor** uses to create an initial population of kernels, which will be the starting point to the evolutionary search.

configurations. For example, if the computation targets a CPU, it may randomly select a value from `auto_unroll_configs_cpu`, such as 64. Similarly, GPU computations could pick from `auto_unroll_configs_gpu`, like 1024. Additionally, the `always_unroll_inner` rule can be applied to specific iterators marked for unrolling, further refining the unrolling strategy based on the characteristics of the computation graph. Through this iterative process, the optimization framework dynamically adjusts unrolling factors.

Fill tile size: Let's say we have a transformation history with several split steps. The `InitFillTileSize` rule is applied to this history to ensure that all split steps have well-defined tile sizes for subsequent optimization. Here's a simplified example of the transformation history:

1. Split the computation into smaller tiles along the spatial dimensions.
2. Further split the tiles to increase parallelism.
3. Divide the tiles into even smaller blocks for efficient memory access.

Some of these split steps may have yet to define tile sizes, which can hinder optimization. The `InitFillTileSize` rule addresses this by filling in the missing tile sizes based on factorization schemes. For example, let's consider the second split step above, which divides the computation into smaller tiles along spatial dimensions. If the tile sizes still need to be defined, the rule will generate candidate tile sizes based on factorization schemes suitable for the extent of the split step. It then randomly selects one set of candidate sizes and updates the split step with these sizes. After applying the `InitFillTileSize` rule, the transformation history will have all split steps with well-defined tile sizes, allowing for smoother optimization and improved performance.

Evolution of the Annotated Sketch To explore different kernel spaces, **Ansor** keeps a population of promising *candidate kernels*. This population is updated in an iterative process. At each iteration, the current population of candidates evolves through the application of the mutation strategies enumerated in Figure 5.6. To select the next set of candidates, **Ansor** does not run every kernel in the current population. Instead, it

Tile size	This rule evaluates a tiling window and increases its length. The rule ensures that the increase is less than the maximum innermost split factor and conducts permutations of the tile size accordingly.	Auto parallel	This rule applies in a scenario where parallelism was added to the outermost loop, and this loop was generated by fusing outer loops. One loop in the nest is randomly selected from all the parallel loops, and a parallelism factor is chosen for it.
Compute location	This rule assigns a given transformation step to a different computing device than all the previous locations used in that computation.	Auto unroll	This rule changes the unrolling factor of a loop marked to be unrolled. It uses the previous history of that unrolling location to determine the new factor.

Figure 5.6: Mutation rules for evolutionary search. In contrast to the initialization rules seen in Figure 5.5, the mutation rules take the history of previous annotations when determining the next value of a given annotation.

uses a cost model to select candidates that are likely to run efficiently. These promising candidates are executed, and the result of these samples is used to recalibrate the cost model; hence, improving the estimates of the next candidates. Periodically, the algorithm reports statistical information regarding the search progress, including maximum and minimum scores, population size, and mutation success rates. Upon completion of the specified number of iterations, the best-performing kernels are recorded as the output.

Prior to each search round, the cost model is retrained to ensure accurate evaluation of candidate solutions. During a search round, the algorithm infers bounds and selects states for measurement, incorporating redundancy checks and an **epsilon-greedy** strategy for exploration. Retries are employed to detect if the entire search space has been traversed, halting the process if no additional valid states are found. Measured states' throughputs are updated for future rounds. Additionally, the algorithm generates sketches and samples an initial population for evolutionary search, integrating already-measured states and implementing random sampling strategies. Filter mechanisms are applied to remove states that fail initial rules or feature extraction. Finally, the target size is adjusted dynamically to optimize performance during this phase. This comprehensive approach enables efficient exploration and refinement of program configurations, facilitating the discovery of optimized solutions tailored to the target hardware platform.

Termination in Ansor The number of possible schedules is very large; hence, **Ansor** limits the amount of schedules with a *budget of trials*. Each trial consists of the observation of the execution of an actual schedule, which happens at the end of the evolutionary phase. Because a machine learning model contains many kernels, an initial round of trials is partitioned among these layers. Layers are grouped into a worklist, and receive a quota of trials in round-robin fashion. After an initial round of optimizations, layers that run for a very short time are removed from this worklist. This process ensures that layers that run for the longest time are subject to more extensive optimizations. This approach is what Zheng et al. [157] call “optimizing with gradient descent”. Because the budget of trials is fixed, **Ansor** is guaranteed to terminate.

Limitations of Ansor The main limitation of **Ansor** is the fact that it is oblivious to the structure of the search space. For instance, if we observe a performance improvement by increasing the unrolling factor of a loop from 4 to 6, it is likely that if we increase it further to 8, another improvement will also be observed. However, if going to 8 results in performance degradation, then further increases are likely to not bring improvements either. **Ansor**’s exploitation approach, via an evolutionary algorithm, is not aware of this notion of neighborhood between kernels or of potential convex regions in the optimization space.

5.4 Space Exploitation via Droplet Search

Droplet Search [25] is a kernel scheduling algorithm available in **AutoTVM**. **AutoTVM** differs from **Ansor** because it does not create new sketches. Rather, it is restricted to modifying the parameters of a single sketch—the *origin* of the optimization space. **AutoTVM** provides several independent scheduling approaches: random sampling, grid sampling, genetic sampling, etc. However, only Droplet Search will be of interest to this presentation³. Droplet Search is a variation of an exploitation algorithm called *Coordinate Descent*⁴. It relies on the premise that the parameters of a sketch can be arranged into a coordinate space. Figure 5.7 contains an annotated version of the algorithm and the full version can be found in Chapter 4.

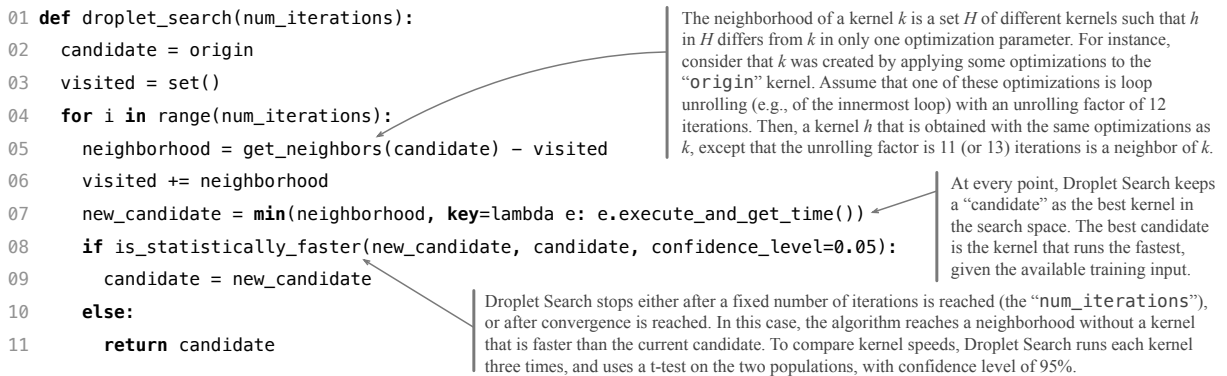


Figure 5.7: The Droplet Search kernel scheduling algorithm. This pseudo-code is a simplified version of the original presentation of the algorithm, taken from [25]. We have removed speculation and parallelism from this version as these features are immaterial for the presentation of our ideas.

³Section 5.12 shall compare Droplet Search with the other techniques available in **AutoTVM** as when they are used as **Ansor**’s exploitation mechanism.

⁴It is unclear who invented Coordinate Descent. Descriptions of the algorithm can be found in classic textbooks [152]. For a comprehensive overview, we recommend the work of Wright [148].

Example 19. *Let us assume a sketch formed by two optimizations: unrolling and tiling. For the sake of this example, unrolling supports five “unrolling factors”: $\{1, 2, 3, 4, 5\}$. These are the parameters of the loop unrolling optimization. Tiling is parameterized by the size of the tiling window. Let us assume the following sizes: $\{1, 2, 4, 8, 16\}$. The optimization space, in this case, is formed by 5×5 points, such as $(1, 1)$, which means no optimization, or $(3, 16)$, which indicates that the loop must be unrolled three times, and then tiled with a window of size 16. These points, e.g., $(1, 1)$, $(3, 16)$, etc, are the coordinates of the optimization space.*

From the notion of coordinates, Droplet Search defines a *neighborhood function*: a function that returns the *neighbors* of a given coordinate. Intuitively, the neighbors of a coordinate are the points that are the closest to it. In Example 19, the neighbors of (unrolling = 3, tiling = 8) would be the points $(2, 8)$, $(4, 8)$, $(3, 4)$ and $(3, 16)$. From this concept of neighborhood, Droplet Search works iteratively as follows:

1. At iteration zero, let the best current candidate be the set of parameters that implement no optimization.
2. Let $(c_1, c_2, \dots, c_n)$ be the best set of parameters discovered up to iteration i .
 - a) If there exists $c'_i, 1 \leq i \leq n$, such that $(c_1, \dots, c'_i, \dots, c_n)$ yields a faster kernel than $(c_1, \dots, c_i, \dots, c_n)$, then update the current best candidate to use c'_i instead of c_i .
 - b) If there is no such c'_i , then the search terminates.

Limitations of Droplet Search Droplet search is a fast search algorithm when compared to **Ansor** or to other approaches available in **AutoTVM** [25]. However, it has two fundamental limitations:

- Droplet Search is restricted to a single sketch. In other words, it can modify a sketch’s annotations, but it cannot create new sketches. This is a limitation of any search algorithm used in **AutoTVM**, but it is not a limitation of **Ansor**.
- Droplet Search depends highly on the initial schedule it receives as the *seed* of the search procedure. If this initial schedule does not exist in the same convex region as the optimal schedule, then Droplet Search cannot find the optimal schedule.

By combining **Ansor** and Droplet Search, we hope to circumvent the limitations of both search techniques. Section 5.5 explains how these two approaches can be used together.

5.5 Combining **Ansor** with Droplet Search

To combine Droplet Search and **Ansor**, we determine two parameters:

- K : the budget of trials of **Ansor**.
- $N < K$: a subset of trials.

We then proceed as follows:

1. Run **Ansor** on the target model using only N trials.
2. Give the best schedule found with N trials to Droplet Search.
3. Run Droplet Search up to convergence.

Figure 5.8 provides some intuition on this *modus operandi*. As the figure illustrates, the proposed technique seeks to use **Ansor** to explore the universe of sketches, and then use Droplet Search as the core strategy to explore concrete representations of these sketches.

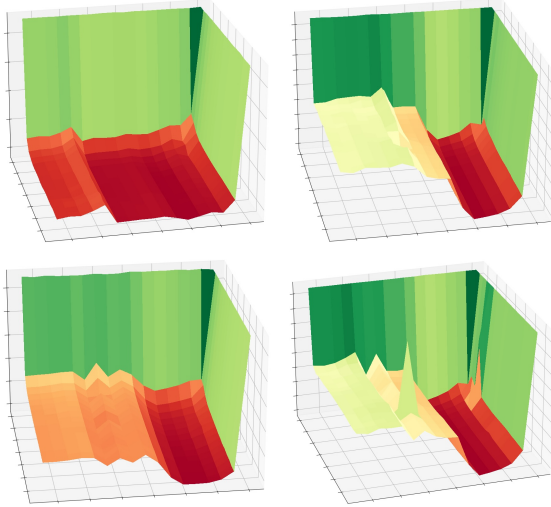
In Section 5.8 we demonstrate that by choosing proper values for K and N we can outperform **Ansor** in two ways: first, producing faster end-to-end machine learning models; second, reducing the search time of **Ansor**. The modifications needed to add this combination to the current code base of Apache TVM are relatively small. Section 5.7 describes these modifications.

5.6 User Interaction

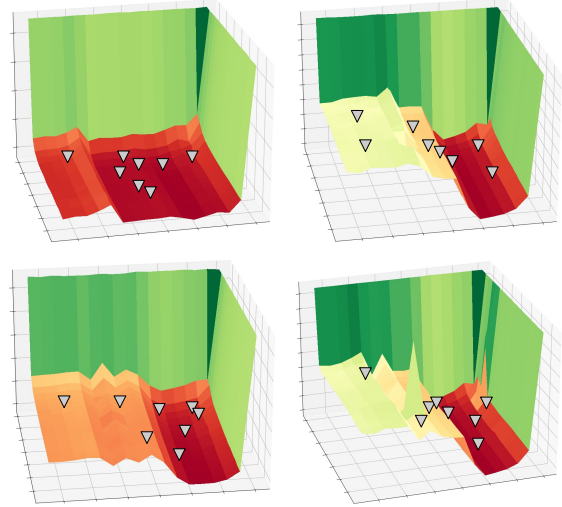
The proposed extension does not change how **Ansor** is used. Its original implementation can still be invoked via the same commands without modification. If the user wants to apply Droplet Search on the best model found by **Ansor** after several trials, they only need to run a second command. This section explains such interactions. After **Ansor** completes the tuning process, users can integrate the *droplet_exploitation* function into the solution with a single line of Python code. This functionality allows users to optimize the application of the **Ansor** solution, offering flexibility and control over the process. Example 20 shows the core commands.

Example 20. *In order to optimize an ONNX model, we can use **Ansor** with the command below. This command will run **Ansor** with a budget of 10,000 trials:*

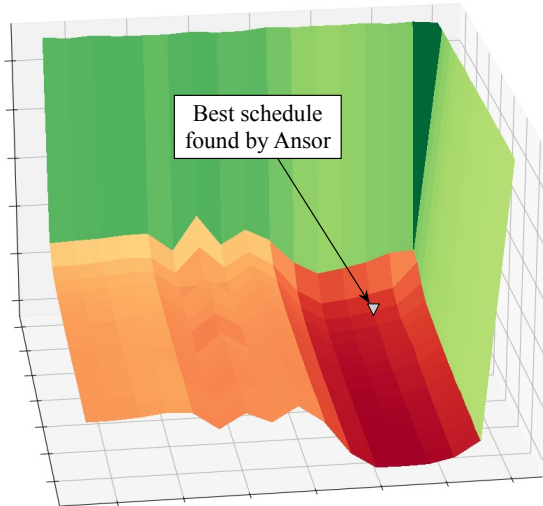
(a) Ansor uses rewriting rules to produce different sketches. Each sketch describes a sequence of optimizations



(b) Ansor uses an evolutionary algorithm to annotate the sketches; hence, producing actual schedules.



(c) We choose the best schedule (annotated sketch), considering the running time of the end-to-end model.



(d) We give the best schedule to droplet search, which uses coordinate descent to explore its neighborhood.

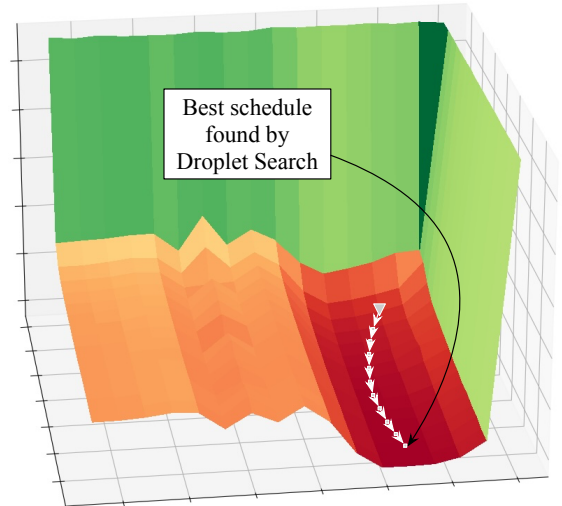


Figure 5.8: Summary of this proposal.

```
$> python3 benchmarks/models_onnx.py -m ansor -a x86 -t 10000 \
    -l results/x86_resnet18_10k.json -b models/resnet18.onnx
```

*This command has no modification in regards to the original implementation of **Ansor**. Notice that it will produce a log file called **results/x86_resnet18_10k.json**. This file can be given to Droplet Search, to improve even further the final implementation of the end-to-end model, as follows:*

```
$> python3 benchmarks/models_onnx.py -m droplet -a x86 -t 100 \
    -l results/x86_resnet18_10k.json -b models/resnet18.onnx
```

*The above command will run Droplet Search on the best candidate found by **Ansor**, limiting the number of trials given to Droplet Search by 100.*

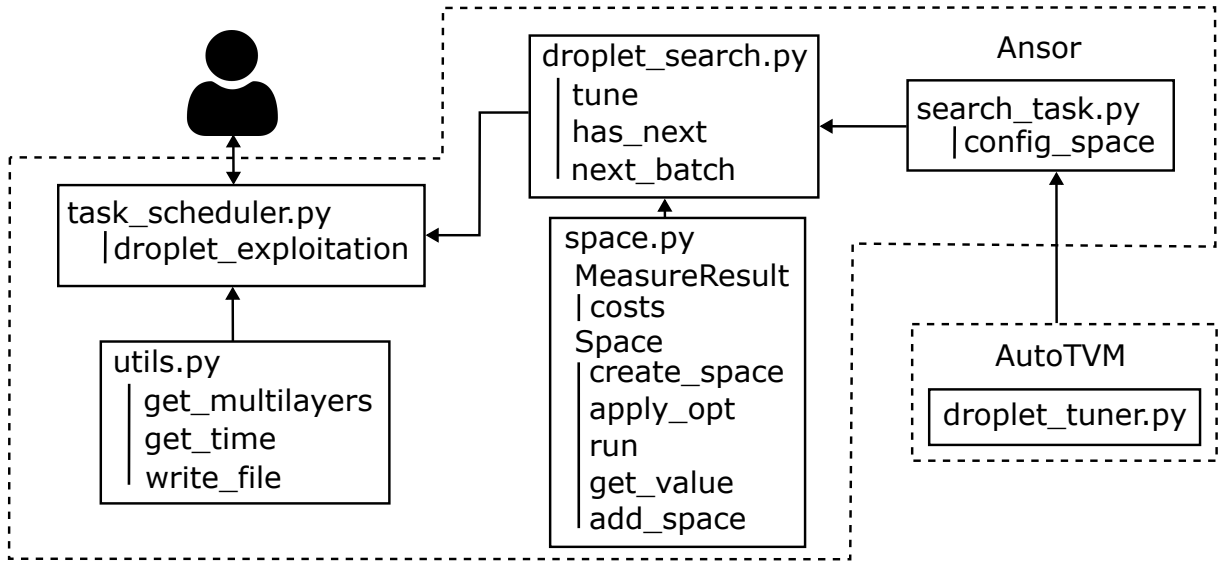


Figure 5.9: The files and functions in the Apache TVM code base that must be modified to incorporate the ideas proposed in this document.

5.7 Implementation Details

For a better understanding of the implementations, we can separate them into two parts: *Ansor infrastructure*, and *AutoTVM infrastructure*. Figure 5.9 shows a diagram of the modifications made to **Ansor**.

Ansor infrastructure Droplet Search needs a specific data structure to optimize the best solution found by **Ansor**. To bridge these two algorithms, three functions were created in the `utils.py` file:

- *get_multilayers* receives the **Ansor** log file and extracts the layers that will be optimized.
- *get_time* receives the log file from **Ansor** and returns the total time spent tuning.
- *write_file* writes the results in json format and saves it to the system.

The **Space** class has essential methods that take **Ansor**'s input format and apply optimizations to it. In the current structure, it optimizes **SplitStep** (SP) and **PragmaStep** (PR) parameters. SP focuses on optimizing tile size with a step power of 2, with the range starting from 2 and going up to $\min(64, \text{size_tile})$. PR focuses on unrolling the loop starting at 64 and going up to 512, with the values varying in a power of 2.

AutoTVM infrastructure One challenge in connecting **Ansor** and **AutoTVM** is that a class **Search Task** does not have a method called `config_space` in **Ansor**. A simple

solution to solve this problem was to add a method using a Python parameter into the `SearchTask` class. This method allows to create a link between `Ansor` and `AutoTVM` without having to change the `Ansor` structure. Thus, Droplet Search reuses what was implemented in `AutoTVM` and only implements some methods necessary for the `Space` class to be used.

5.8 Evaluation

This section evaluates the idea proposed in the chapter 5. In particular, it seeks to demonstrate that by exploiting, via Droplet Search, a reduced set of samples explored by `Ansor`, it is possible to outperform `Ansor` itself. We shall refer to this new version of `Ansor`, which uses Droplet Search, as the *Combined Approach*. Henceforth, we denote it as `DPAnsor`, reserving `Ansor` for the original implementation of that tool. In what follows, we explore six research questions:

- RQ1:** How many samples does `DPAnsor` need to observe to produce kernels that outperform those produced by `Ansor` with 10,000 trials?
- RQ2:** How many samples can `DPAnsor` observe and still outperform `Ansor` in terms of search time when the latter uses 10,000 trials?
- RQ3:** How does the size of models impact the behavior of `DPAnsor`, in terms of kernel performance and search speed?
- RQ4:** How does Droplet Search compare to other search techniques available in `AutoTVM`, in terms of their ability to exploit `Ansor`'s results?
- RQ5:** How does the average number of samples that Droplet Search gauges per layer vary with the initial budget allocated to `DPAnsor`'s exploration phase?
- RQ6:** How does `DPAnsor` compare with `PyTorch` and `Ansor` in terms of the speed of the kernels that it produces, considering a range of different machine-learning kernels?

Before diving into the research questions, we explain our experimental setup. Notice that a fully containerized version of this methodology has been organized as a `docker` image, which is publicly available at <https://github.com/lac-dcc/bennu>.

Hardware and Software. We evaluated the scheduling approaches on four different architectures, as shown in Figure 5.10. The hardware consists of a general-purpose desktop architecture (AMD Ryzen 7 [7]), a cluster-based machine (ARM A64FX [104]), and two graphics processing units (NVIDIA A100 [100] and NVIDIA RTX3080 [101]). The experiments reported in this section use versions of **Ansor** and **AutoTVM** (Droplet Search) available at **Apache TVM v0.13.0**, released in July 2023. The version used for **TensorFlow** was 2.14, and **PyTorch** was 2.0+cu118.

Device	Architecture	ISA	Clock Max (GHz)	Memory			
				RAM (GB)	Cache L1 (MiB)	Cache L2 (MiB)	Cache L3 (MiB)
AMD R7-3700X	x86-64	x86-64 (x86)	4.4	64	0.512	4	32
ARM A64FX	aarch64	ARMv8.2 + SVE	2.2	32	6	32	-
Nvidia A100	Ampere	PTX	1.4	40	-	-	-

Figure 5.10: The architectures evaluated in this report.

Benchmarks. This section evaluates kernel scheduling across twenty neural networks. The first column of Figure 5.11 contains the complete list of these models. All these models are implemented using the ONNX representation to make a comparison between **TVM**, **PyTorch** and **TensorFlow** possible, as seen in Section 5.14. The models used in our study are sourced from the ONNX model zoo available at <https://github.com/onnx/models>.

Methodology. A machine learning model forms a graph of *kernels*. **Ansor** optimizes machine learning models per kernel, assuming kernels can be independently optimized. It starts with a *budget of trials*, where each trial is a transformation that can be applied to a kernel. Let us call this budget K . **Ansor** ensures that each kernel receives a fraction of these K trials. Currently, this initial fraction is $\min(K/L, 64)$, where L is the number of layers (kernels) in the model. After an initial round of optimizations, **Ansor** applies the remaining trials onto kernels that run for the longest time. This approach directs the optimization effort to the kernels that are more likely to contribute to the overall running time of the end-to-end model. In what follows, all the results we report are relative to a baseline version of **Ansor** equipped with a budget of 10,000 trials. We shall test **DPAnsor** with either $K = 1, 10, 25, 50, 100, 200, 300$, or 1,000 trials. Suppose we choose $K = 100$, for instance. In that case, we will run **Ansor** with a budget of 100 trials, pick the best configuration (which results from the independent optimization of the kernels), and give this configuration to **AutoTVM**’s Droplet Search. We then let Droplet Search run until it reaches convergence.

Confidence Sections 5.9-5.12 show relative results. It could be possible that the running times presented for kernels and schedulers is similar enough to the point of hindering our conclusions meaningless. However, such is not the case. Scheduling runs for a very

long time. To give the reader an idea, **Ansor**, with a budget of 10,000 trials (or baseline), takes 11,195 seconds to schedule **AlexNet**, our smallest model. This time drops to 2,839 seconds considering **DPAnsor** with a budget of 300 trials. The running time of kernels is faster: end-to-end models run for a few seconds. However, in every experiment, we consider averages of three samples and only report speedups if the p-value (produced via the non-parametric Wilcoxon rank-sum test) for the difference between the two populations is below 0.01.

5.9 RQ1 – On the Quality of End-to-End Models

We consider that a version of an end-to-end model is better than another for a given architecture when it runs faster in that architecture. The execution time of a model is determined by the schedule of the kernels that constitute it. If we apply Droplet Search to the best model produced by **Ansor** after it observes 10,000 trials, we will likely improve the model (at least, we should not make it worse). However, this section shows that obtaining a better model via **DPAnsor** with a much lower budget is possible in four different architectures. For brevity, we analyze in details results obtained on an x86 CPU. For the other architectures, we show only summarization results.

Discussion: AMD Ryzen 7 (x86-64) The x86 architecture represents a widely adopted instruction set architecture (ISA), serving as the basis for Intel and AMD processors. Figure 5.11 compares kernel speed and search time of models running on an x86-64 CPU. The top part of the figure (labeled “10k speedup execution time”) compares the running time of the kernels produced by **Ansor** and **DPAnsor**. The lower part (labeled “10k speedup tuning time”) compares the search time of these two approaches, and shall be discussed in Section 5.10. In both cases, bars above 1.0 denote improvements of **DPAnsor** over **Ansor**. In terms of kernel speed, sampling 25 configurations with **DPAnsor** is sufficient—in most models—to outperform **Ansor** with 10K samples. Speedups improve gradually as more samples are added to **DPAnsor**, to the point that with 1,000 samples, we see an average speedup (geometric mean) of 34%.

Discussion: Other Architectures Figure 5.12 (Left) compares the relative speed of kernels produced by **DPAnsor** over **Ansor** in all the four architectures available for this study: in addition to the AMD x86 of Figure 5.11, we see an Nvidia A100(Ampere), an Nvidia RTX3080 (Ampere), and an ARM A64FX (aarch64). In every case, speedups of **DPAnsor** relative to **Ansor** emerge consistently with $K = 300$. However, with $K = 100$,

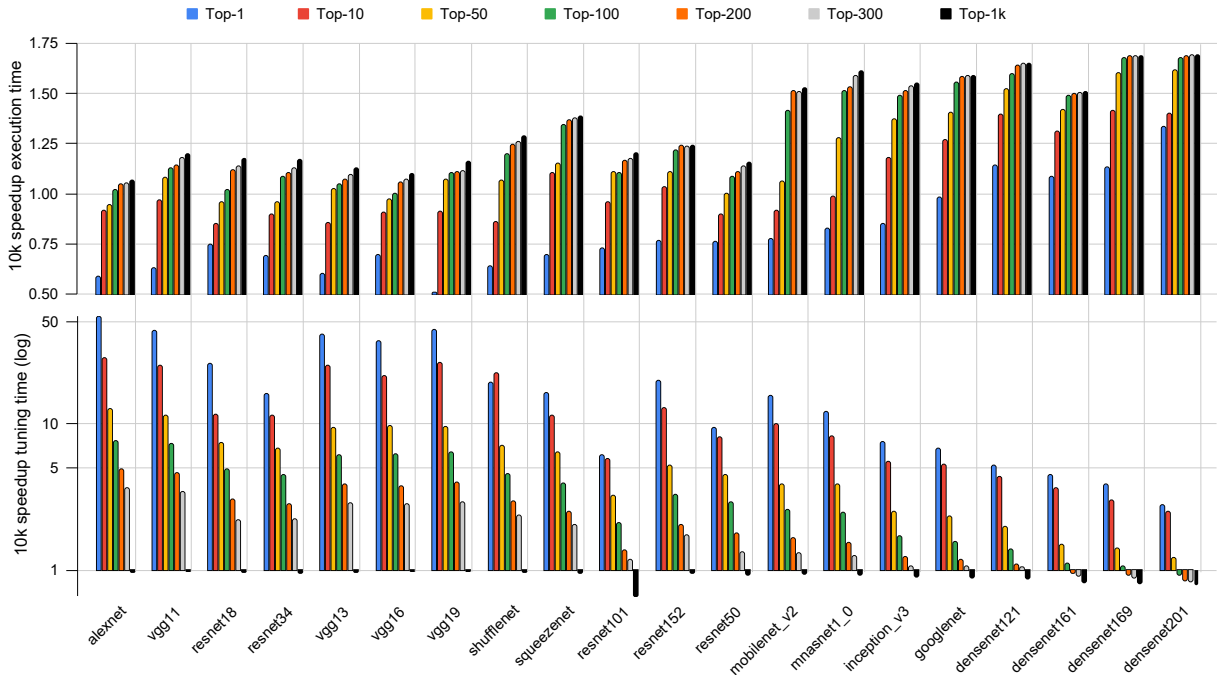


Figure 5.11: Comparative Analysis of Optimization Results on the x86 Architecture Using an AMD Ryzen 7 3700X Processor. Numbers show **AnsoR**/**DPAnsoR** ratios. Thus, results higher than 1.0 (in blue) denote improvements of **DPAnsoR** (this work) over **AnsoR**.

we have already recorded speedups on the Nvidia A100 and on the ARM A64. We failed to see meaningful speedups on the Nvidia RTX3080 because **AnsoR**, with a budget of 10K samples, seems to be very close to achieving peak performance on this GPU. Even if we increase its budget of samples, we could not obtain better kernels to the 20 different models used in this study. Nevertheless, notice that **DPAnsoR** achieves this peak performance with 300 samples (plus a few—less than 50—samples probed by Droplet Search). On the A100 GPU, in contrast, after 300 samples, **DPAnsoR** already delivers kernels 12% faster than those produced by **AnsoR**.

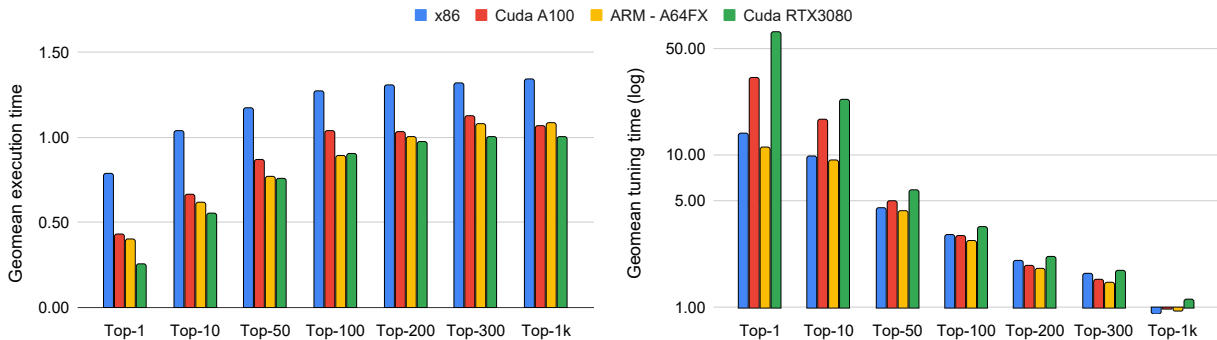


Figure 5.12: Left: relative speed of kernels produced by **DPAnsoR** over **AnsoR** on four different architectures. Right: relative search time of **DPAnsoR** over **AnsoR**. Every bar is the geometric mean of relative times observed on 20 different models. Results above 1.0 represent improvements of **DPAnsoR** over **AnsoR**.

5.10 RQ2 – On the Search Time

We define a scheduling approach as faster than another if it requires less time to converge to an end-to-end model’s final, optimized version. The search time of **Ansor** encompasses the time spent applying optimizations to kernels, deriving new optimizations, and running the kernels themselves, with a limit set at 10,000 trials. On the other hand, the search time of **DPAnsor** involves all the steps of **Ansor**, constrained to a lower number of trials, along with the time it takes to run Droplet Search until convergence on the kernels that compose a model. Although we have restricted Droplet Search to a maximum of 100 trials, it typically converges well before reaching that limit. This section compares the search time between **Ansor** and **DPAnsor**.

Discussion The lower part of Figure 5.11 compares search times on x86. For most models, **DPAnsor** is consistently faster than **Ansor** for any number of trials up to $K = 300$. At 1,000 trials, **Ansor** becomes consistently faster. Also, **Ansor** tends to outperform **DPAnsor** for very large models. This fact happens due to the longer time that Droplet Search takes to converge: the more complex the model, the more room Droplet Search will have to optimize it. Section 5.11 further discusses the impact of the model size on the behavior of **DPAnsor**. Figure 5.12 summarizes the search time comparison for the other architectures. The pattern is similar to the one observed on x86: **DPAnsor** is consistently faster when $K \leq 300$, and slower (except on the Nvidia RTX3080) at $K = 1,000$.

5.11 RQ3 – On the Impact of Model Size

The behavior of **DPAnsor**, when compared to **Ansor**, varies with the model’s size. We summarize this variation with two observations:

1. The larger the model, the less samples **DPAnsor** needs to observe to outperform **Ansor**, if **Ansor** uses a budget of 10,000 samples.
2. The larger the model, the lower the benefit, in terms of search time, of **DPAnsor** over **Ansor**.

The rest of this section provides data to support these two conclusions.

Discussion: The Search vs Quality Slope The kernel optimization technique impacts two core numbers: the search time and the speed of the final model. We can use these two quantities—search time (S) and model performance (P)—to define an $S \times P$ line characterizing the behavior of the optimization technique. Figure 5.13 shows these lines regarding four models and two architectures: AMD’s x86 and NVIDIA’s Ampere. We chose these two architectures because x86 is the scenario where **DPAnsor** performs better, and Ampere is the scenario where it performs worse.

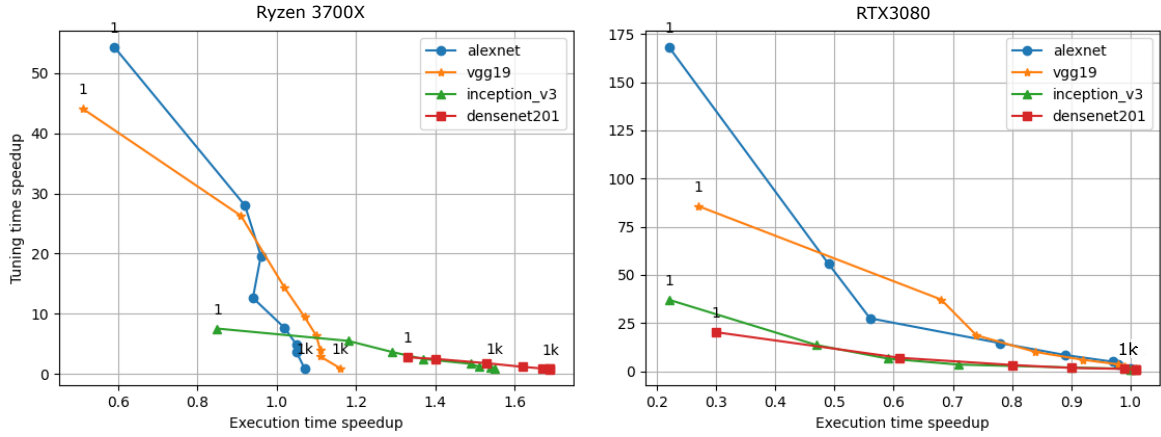


Figure 5.13: The search vs quality line that characterizes four models optimized in the AMD (Left) and in the NVIDIA (Right) setting: the larger the model, the lower the slope. Numbers on the X and Y axes are speedup/slowdown relative to **Ansor** with a budget of 10,000 trials. Numbers for the left chart are available in Figure 5.11, and numbers for the right chart were observed on the Nvidia RTX3080. The labels on the dots (1 and 1k) refer to the number of trials given to **DPAnsor**.

Figure 5.13 uses our two smallest and two largest models. The numbers on the axes show ratios between **DPAnsor** and **Ansor**; the latter using a budget of 10,000 trials. Each dot in Figure 5.13 refers to the number of trials that **DPAnsor** is allowed to observe before shifting to Droplet Search. The figure labels dots that refer to **DPAnsor** with one sample (its most restrictive scenario) and dots that refer to 1,000 samples (its least restrictive scenario).

The slopes of the lines in Figure 5.13 are always negative, meaning that as more samples are given to **DPAnsor**, the difference between its search time and **Ansor**’s reduces, but the quality of the kernels that it finds improves. However, the inclination changes with the size of the model. The larger the model, the lower the benefit of **DPAnsor** over **Ansor** in terms of search time; but the higher the relative benefit in terms of kernel speed. This result is due to **Ansor**’s fixed budget of 10,000 trials. In a small model, more trials are distributed to each layer; in a large model, each layer receives only a handful of trials.

Figure 5.14 provides further data that supports the previous observations. The figure shows how kernel quality and search time vary with the size of models. If we consider **AlexNet**, which has only 13 layers, each layer might receive, on average, $10^4/13$ trials. Inevitably, good kernels will emerge from this search. Thus, the benefit of the exploitation

phase, which uses Droplet Search, tends to be smaller. On the other hand, if we consider **DenseNet201**, which has 113 layers, then **Ansor** allocates, on average, $10^4/113$ trials per layer—less than 100 samples per each kernel of the computational graph. This number is too low to effectively explore the space of possible kernel implementations. In this case, Droplet Search has more opportunity to improve the kernels that **Ansor** finds. Even using only one trial to find the origin of the optimization space is already enough to have **DPAnsor** outperforming **Ansor** in terms of the quality of the model.

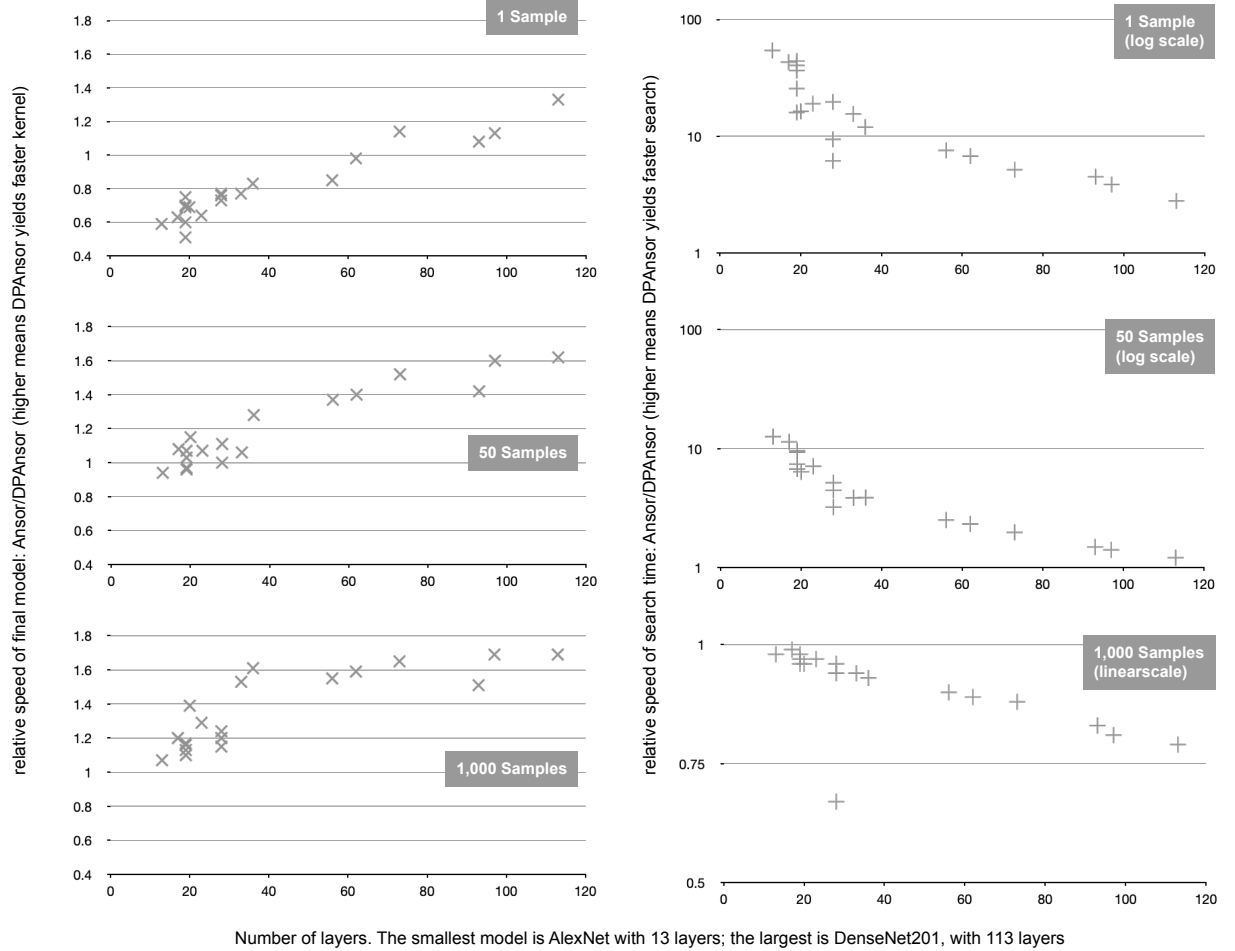


Figure 5.14: (Left) Variation in kernel quality (**Ansor/DPAnsor**) with model size. (Right) Variation in search time (**Ansor/DPAnsor**) with model size.

5.12 RQ4 – On the Search Technique

The core idea of this work consists in using Droplet Search to exploit results produced by **Ansor**. Therefore, an immediate question from this proposal is: *what if other search techniques are used instead of Droplet Search, as the basic exploitation technique?* Indeed,

AutoTVM, the framework hosting Droplet Search, provides four other search techniques that could fill the same role as Droplet Search. These techniques are described as follows:

Random: a random search on the space of valid optimization parameters. In this case, each sample is randomly produced. Search is stateless, meaning that the results of a kernel bear no influence on the choice of the next kernel.

Grid: a grid search on the space of valid parameters of the optimizations. In this case, each sample derives from a regular and exhaustive variation of each optimization parameter. Search keeps a minimum of state, namely, a counter per optimization parameter, that goes over the range of acceptable values.

GA: a genetic algorithm that treats the sequence of optimization parameters as chromosomes. Search is stateful: the running times of already seen kernels provide information to guide the synthesis of the next kernels via operations such as mutation, crossover and pruning.

XGB: search is based on gradient boosting, as implemented by the XGBoost Library [31]. Similar to the genetic algorithm, the search is stateful, as the running time of kernels guides the construction of the search tree.

In the rest of this section, we analyze the behavior of DPAnsoR, once its search technique (the “DP” part of AnsoR) is replaced by each one of the other four algorithms available in AutoTVM.

Methodology Every experiment in this Section allocates a budget of 300 samples to AnsoR for each model that we autotune. Exploitation, via one of the different search techniques in AutoTVM, uses a budget of 100 trials to optimize each layer of AlexNet. Droplet Search converged before 100 trials in every layer, but all the other approaches run 100 samples, as they do not have a notion of convergence. Thus, each other approach samples 1,200 kernel configurations. Each search technique in AutoTVM exploits the best sequence of optimizations found by AnsoR, which is the same for all of them. Following the methodology adopted in Section 5.11, we report results for the AMD x86 and the NVIDIA 3080 boards. These are the best and worst scenarios observed for DPAnsoR in Sections 5.9 and 5.10.

Discussion: AMD Ryzen 7 (x86-64) Figure 5.15 compares Droplet Search against other AutoTVM methods on the x86 architecture. As an exploitation technique, Droplet Search demonstrates superior performance in terms of execution time and kernel speedup. The **random** search technique is completely oblivious to the seed best provided by AnsoR. The **grid** search is almost oblivious to it: although the grid search starts from a likely good kernel, it diverges quickly from that configuration. The other techniques can benefit

from **Ansor**’s exploration phase. We feed both **GA** and **XGB** with a promising seed; however, these search techniques still need random seeds to diversify the initial population of kernels from where they depart. Because the other search techniques lack a convergence criterion, the relative performance of Droplet Search—in turning time—is even better: it is approximately 3x faster.

x86		Execution speedup				Tuning speedup			
Model	L	grid / droplet	random / droplet	ga / droplet	xgb / droplet	grid / droplet	random / droplet	ga / droplet	xgb / droplet
alexnet	13	3.04	2.55	2.27	2.04	3.55	5.05	3.62	3.93
vgg11	17	6.97	2.42	1.69	1.82	4.80	3.53	3.61	3.53
resnet18	19	8.28	2.22	1.97	2.40	2.29	2.81	3.05	3.09
resnet34	19	9.63	2.34	2.00	2.05	1.97	2.77	2.61	2.85
vgg13	19	6.51	3.15	2.27	2.19	4.86	4.37	4.26	3.72
vgg16	19	6.90	2.29	1.98	2.47	4.62	4.17	3.78	3.62
vgg19	19	6.60	2.20	1.93	2.32	4.46	3.71	4.18	3.76
shufflenet	20	2.30	2.41	2.34	2.23	4.78	7.50	6.74	7.21
squeezenet	23	2.40	3.31	2.52	2.25	3.24	4.89	4.65	4.58
resnet101	28	7.02	2.69	3.62	3.34	2.01	2.95	2.87	2.65
resnet152	28	6.87	2.21	2.63	2.09	2.40	3.10	3.08	3.11
resnet50	28	7.43	2.09	2.38	2.39	2.34	3.16	3.52	3.65
mobilenet_v2	33	3.27	2.67	3.19	2.69	3.61	5.60	5.67	5.47
mnasnet1_0	36	3.09	2.57	3.31	2.55	3.41	5.43	5.81	5.54
inception_v3	56	2.49	2.24	2.64	2.41	2.54	4.18	4.13	3.96
googlenet	62	2.71	2.12	2.29	2.33	2.61	4.05	3.77	3.88
densenet121	73	2.50	1.90	1.87	2.23	2.16	3.18	3.28	3.11
densenet161	93	2.65	2.30	2.49	2.47	2.06	3.39	3.18	3.24
densenet169	97	2.66	1.92	2.30	2.30	1.91	2.87	2.99	2.90
densenet201	113	2.58	2.07	2.54	2.52	1.84	2.82	2.76	2.77
GEO	-	4.21	2.36	2.37	2.34	2.90	3.82	3.75	3.70

Figure 5.15: Comparison of different exploitation techniques used in tandem with **Ansor** on the x86-64 setting. Numbers are ratio of “*SearchTechnique/DropletSearch*”. Thus, numbers above 1.0 demonstrate the effectiveness of the technique proposed in this work.

Discussion: CUDA RTX 3080 (Ampere) In the GPU setting, Droplet Search is still a superior exploitation technique in terms of search time and kernel speed, as Figure 5.16 demonstrates. In contrast to the CPU setting, invalid kernels are common in the GPU scenario. **AutoTVM** does not provide any way to constrain the search technique to using parameters that yield correct kernels. We observe, for instance, that the **Grid** search often generates invalid kernels, as the number of threads on the X, Y, and Z dimensions exceeds the total number of threads in the stream multiprocessor.

Discussion: Universality of these Results Droplet Search is not universally superior to the other search techniques available in **AutoTVM**, when used as an exploitation method in **Ansor**. For instance, if we analyze the behavior of these different search approaches per layer of **AlexNet**, we observe situations where Droplet Search yields slower kernels than

RTX 3080		Execution speedup				Tuning speedup			
Model	L	grid / droplet	random / droplet	ga / droplet	xgb / droplet	grid / droplet	random / droplet	ga / droplet	xgb / droplet
alexnet	13	6.10	2.74	1.28	1.22	2.85	5.43	3.14	3.63
vgg11	17	6.92	1.50	1.56	1.50	3.03	3.89	3.98	3.88
resnet18	19	15.74	3.35	2.06	2.12	1.75	2.36	2.45	2.53
resnet34	19	18.91	2.99	2.56	2.08	1.58	2.26	2.24	2.30
vgg13	19	9.98	1.89	2.12	1.82	2.68	5.15	5.05	4.17
vgg16	19	9.12	2.47	1.90	1.49	2.82	5.52	5.36	5.60
vgg19	19	10.25	2.84	4.09	1.58	3.25	4.91	5.09	5.47
shufflenet	20	4.38	3.62	2.42	2.37	2.96	3.94	3.82	3.88
squeezenet	23	9.36	3.53	3.28	2.44	2.64	3.24	3.16	3.47
resnet101	28	22.06	5.27	3.13	2.35	1.80	2.53	2.46	2.63
resnet152	28	21.36	4.90	3.95	2.66	2.10	2.80	2.99	3.08
resnet50	28	23.91	3.06	2.57	2.50	2.01	2.91	3.19	3.34
mobilenet_v2	33	8.99	3.21	2.56	2.66	2.14	2.82	2.64	2.70
mnasnet1_0	36	8.73	2.97	2.79	2.40	2.28	3.09	3.26	3.12
inception_v3	56	11.59	2.89	2.35	2.22	2.44	3.16	3.23	3.16
googlenet	62	10.44	3.22	2.54	2.32	1.97	2.63	2.56	2.65
densenet121	73	14.22	2.78	2.23	2.42	2.08	2.29	2.43	2.31
densenet161	93	12.60	3.95	3.14	2.21	2.01	2.13	2.00	2.04
densenet169	97	12.54	3.52	2.40	1.99	1.91	2.14	2.23	2.16
densenet201	113	12.31	3.48	2.49	2.17	1.84	2.12	2.08	2.10
GEO	-	11.42	3.10	2.48	2.08	2.26	3.10	3.03	3.08

Figure 5.16: Comparison of different exploitation techniques used in tandem with **Ansor** on the Cuda setting. Numbers are ratio of “*SearchTechnique/DropletSearch*”. Thus, numbers above 1.0 demonstrate the effectiveness of the technique proposed in this work.

the other methods. Figure 5.17 demonstrates this point. These subpar results happen due to the stochastic nature of all these autotuning techniques. Droplet Search might stop on a suboptimal kernel because its convergence criterion is statistical in nature: if the running time of kernels is considered similar with a confidence level of 95%, then the coordinate descent procedure stops. However, regarding search speed, we have not identified one single layer of **AlexNet** where Droplet Search would be slower than the other search approaches. Additionally, if we repeat the same analysis per layer of **AlexNet** on the GPU setting, we observe that Droplet Search is superior to the other search approaches in every layer. We omit these results from this work for the sake of space.

5.13 RQ5: The Average Number of Droplet Search Samples

As hinted in Section 5.12, Droplet Search has a convergence criterion discussed in Section 3.3 of its description [25]. In this case, the search stops once there is no

x86	Execution speedup				Tuning speedup			
layer	grid / droplet	random / droplet	ga / droplet	xgb / droplet	grid / droplet	random / droplet	ga / droplet	xgb / droplet
0	5.16	7.99	10.19	5.16	2.25	8.11	3.43	4.06
1	1.27	1.26	1.31	1.27	10.74	10.77	3.29	3.28
2	4.38	5.71	3.07	3.07	2.24	3.42	5.54	5.73
3	6.06	10.79	4.78	4.97	2.43	2.56	2.34	2.31
4	1.16	1.15	1.10	1.03	3.66	3.70	3.60	3.64
5	4.65	3.70	1.58	1.58	2.44	4.55	4.70	4.24
6	0.99	0.98	0.95	0.95	10.68	13.37	3.25	3.32
7	5.57	5.32	5.70	5.32	2.77	3.25	3.11	3.49
8	4.28	1.96	3.12	1.96	2.31	4.07	4.40	4.12
9	0.99	0.99	1.08	0.99	3.97	3.99	2.08	2.08
10	3.30	0.74	0.70	0.70	7.10	8.44	7.02	8.41
11	4.32	1.66	1.55	1.55	2.54	8.69	4.20	8.43
12	6.10	5.90	6.20	6.01	2.47	2.09	2.74	2.80
GEO	3.04	2.55	2.27	2.04	3.55	5.05	3.62	3.93

Figure 5.17: Results of Figure 5.15, analyzed per layer of AlexNet.

statistically significant difference between the current kernel and the kernels within its neighborhood. Thus, the more optimized the seed of the coordinate descent algorithm, the fewer iterations it is intuitively expected to take until convergence. This section investigates if this hypothesis is true.

Discussion We count the number of iterations of Droplet Search on our different architectures, considering different budgets for DPAnsoR. Figure 5.18 shows this number for each model evaluated on the x86 setting. The figure reports a total number of trials and an average number of trials per layer. The general tendency is that the larger the budget of trials allocated to DPAnsoR, the faster Droplet Search converges.

Figure 5.19 summarizes, for each architecture, the numbers earlier seen in Figure 5.18. Figure 5.19 reports averages per model (considering the 20 available models), and averages per layer. In the latter case, we divide the total sum of trials observed for all the models by the sum of the number of layers present in every model. In every case, the same tendency is evident: more trials sampled during exploration imply fewer trials sampled during exploitation. This result is intuitive: as previously mentioned, Droplet Search tends to reach stability the closer to a local optimum it starts.

x86-64		Sum of trials							Average trials per layer						
Model	L	Top-1 Anzor	Top-10 Anzor	Top-50 Anzor	Top-100 Anzor	Top-200 Anzor	Top-300 Anzor	Top-1k Anzor	Top-1 Anzor	Top-10 Anzor	Top-50 Anzor	Top-100 Anzor	Top-200 Anzor	Top-300 Anzor	Top-1k Anzor
alexnet	13	187	201	205	199	169	162	171	14	15	16	15	13	12	13
vgg11	17	232	210	185	172	158	158	160	14	12	11	10	9	9	9
resnet18	19	428	437	441	351	316	328	318	23	23	23	18	17	17	17
resnet34	19	427	419	359	387	428	350	351	22	22	19	20	23	18	18
vgg13	19	245	274	245	213	190	177	187	13	14	13	11	10	9	10
vgg16	19	263	228	230	193	218	185	186	14	12	12	10	11	10	10
vgg19	19	216	240	215	233	207	189	204	11	13	11	12	11	10	11
shufflenet	20	295	297	256	287	271	275	272	15	15	13	14	14	14	14
squeezenet	23	489	557	390	421	381	370	364	21	24	17	18	17	16	16
resnet101	28	818	860	759	694	674	684	655	29	31	27	25	24	24	23
resnet152	28	841	726	784	699	687	704	720	30	26	28	25	25	25	26
resnet50	28	925	839	784	757	643	657	644	33	30	28	27	23	23	23
mobilenet_v2	33	524	532	606	603	486	539	524	16	16	18	18	15	16	16
mnasnet1_0	36	706	678	696	706	653	678	685	20	19	19	20	18	19	19
inception_v3	56	1410	1393	1344	1239	1272	1263	1270	25	25	24	22	23	23	23
googlenet	62	1430	1384	1350	1333	1335	1311	1310	23	22	22	22	22	21	21
densenet121	73	2070	1823	1768	1648	1756	1696	1736	28	25	24	23	24	23	24
densenet161	93	2530	2381	2359	2269	2282	2271	2280	27	26	25	24	25	24	25
densenet169	97	2665	2513	2419	2435	2387	2485	2484	27	26	25	25	25	26	26
densenet201	113	3110	3004	2846	2955	3063	2890	2946	28	27	25	26	27	26	26
AVG	-	991	950	912	890	879	869	873	22	21	20	19	19	18	18

Figure 5.18: Number of trials sampled by Droplet Search until reaching convergence, considering the AMD x86-64 architecture. The average number of trials per layer divides the total number of trials per the number of layers in each deep learning model.

Arch	Sum of trials							Average trials per layer						
	Top-1 Anzor	Top-10 Anzor	Top-50 Anzor	Top-100 Anzor	Top-200 Anzor	Top-300 Anzor	Top-1k Anzor	Top-1 Anzor	Top-10 Anzor	Top-50 Anzor	Top-100 Anzor	Top-200 Anzor	Top-300 Anzor	Top-1k Anzor
x86-64	19811	18996	18241	17794	17576	17372	17467	24	23	22	22	22	21	21
ARM A64FX	19610	17785	15768	14450	13506	13683	13072	24	22	19	18	17	17	16
Cuda RTX3080	17803	13982	11886	10965	10779	10751	10819	22	17	15	13	13	13	13
Cuda A100	24831	20278	17147	14655	13731	14110	13832	30	25	21	18	17	17	17

Figure 5.19: Average number of trials per model (considering 20 models) for different architectures. The averages per layer are the quotient of the total number of trials for all the models divided by the total number of layers.

5.14 RQ6: Comparison with Well-Known Machine-Learning Frameworks

Thus far, the results we evaluated in this section are constrained to the Apache TVM software stack. To provide the reader with some perspective on these results, we shall compare **Anzor** and **DPAnzor** with two well-known deep-learning frameworks: PyTorch 2.0 [105] and TensorFlow v2.15.0 [128]. Recently, Ansel et al. [8] have shown that PyTorch is able to outperform **Anzor** in many different workloads. However, in Ansel et al.’s setting, **Anzor** was used as a backend, without autotuning the target kernels. In this section we activate autotuning for **Anzor** and **DPAnzor**. To this end, we give **Anzor** a budget of 1,000 trials. In contrast, we use **DPAnzor**, either with a budget of 100 or 300 trials before

activating Droplet Search.

RTX 3080	Tensorflow	Pytorch	Ansor (1000 samples)		DPAnsor (100 samples)		DPAnsor (300 samples)	
Microkernel	exec time (ms)	exec time (ms)	exec time (ms)	tuning time (min)	exec time (ms)	tuning time (min)	exec time (ms)	tuning time (min)
matmul	0.49759126	0.123663	0.012660	37.96	0.0112784	3.60	0.0081228	8.02
conv2d	10.43869758	2.146128	1.819560	72.39	2.0869700	8.19	1.7976000	19.76
depthwise	4.06337643	3.824702	0.650381	64.67	0.6952420	10.72	0.6530620	21.67
pooling	1.34876990	0.957267	0.974534	71.90	0.9545340	8.66	0.9545340	21.91
reduce	1.31569195	0.314016	0.312430	2.69	0.3124030	2.70	0.3124030	2.72
relu	1.21619201	0.165822	0.172650	0.18	0.1721830	0.19	0.1721830	0.19
A100	Tensorflow	Pytorch	Ansor (1000 samples)		DPAnsor (100 samples)		DPAnsor (300 samples)	
matmul	0.1473048	0.032737	0.013206	41.88	0.0144076	6.3	0.0103285	11.4
conv2d	8.6722476	2.401592	2.025246	73.21	2.2853300	12.09	2.0056600	25.3
depthwise	3.7513964	2.881968	0.422472	93.08	0.6085967	15.75	0.4215467	29.59
pooling	3.1504140	0.935589	0.881824	84.95	0.9825180	9.1	0.8918100	25.33
reduce	1.7370601	0.192991	0.157589	3.92	0.1562760	2.9	0.1562760	2.9
relu	1.0123856	0.119100	0.172235	0.46	0.1972600	0.25	0.1972600	0.25

Figure 5.20: Comparison between the running time of kernels produced via TensorFlow, PyTorch, Ansor or DPAnsor running on two different GPUs. The black cells highlight fastest results; the gray cells indicate ties (with confidence level of 95%). TensorFlow and PyTorch do not perform tuning.

Discussion Figure 5.20 summarizes the results of the comparison of different deep-learning tools. In our setting, PyTorch is consistently faster than TensorFlow on the two different graphics processing units available for these experiments. However, Ansor and DPAnsor outperform PyTorch on large workloads (matmul, conv2d, depthwise and pooling). Corroborating these results, the benefit of Ansor over PyTorch was also observed by Li et al. [84]. Notice that these results do not include two kernels, reduce and ReLU. In these two cases, neither Ansor nor DPAnsor have much room to perform optimizations. These kernels show minimal memory reuse: they visit each memory cell only once, and little benefit can be acquired from autotuning. As a consequence, the search time is very short when compared with the search time spent on the larger kernels.

5.14.1 Threats to Validity – On the Statistical Significance of Reported Results

The discussion presented in Sections 5.9 and 5.10 center around the running time of end-to-end models and on the search time of different scheduling techniques. It is possible that the running times presented for kernels and schedulers is similar enough to

the point of hindering our conclusions meaningless. It is easy to demonstrate that such is not the case for search times. In this case, each scheduling technique runs for a very long time. To give the reader an idea, **Ansor**, with a budget of 10,000 trials (or baseline), takes 11,195 seconds to schedule **AlexNet**, our smallest model. This time drops to 2,839 seconds considering **DPAnsor** with a budget of 300 trials. In this section, we show that the running times of the kernels produced by **Ansor** and **DPAnsor** also present statistically significant differences. To this end, we proceed as follows: we choose two end-to-end models (**AlexNet** and **ResNet34**), and present their mean running times, plus the standard deviations of such means, and the p-values that result from comparing the original version of **Ansor** with our extended version of it.

Discussion. Figure 5.21 presents an analysis of deviation and p-value for **ResNet34** and **AlexNet**, employing **DPAnsor** across varied hardware configurations. The networks are characterized by the number of layers, designated as “Num of Layers”. **ResNet34** comprises 19 layers, and **AlexNet** consists of 13 layers. The table includes the average runtime of the best kernels proposed by **DPAnsor** in seconds, denoted as “Mean Kernel Runtime,” and the average of standard deviation, “Mean Std Dev,” across all layers of the best schedule of a network, in seconds. Additionally, Figure 5.21 presents the count of layers with a p-value less than 0.01, labeled as “Num of Layers with p-value < 0.01.” We compare the results using “Coefficient of Variation” (CV), presented in percentage (%). It is the standard deviation expressed as a percentage of the mean, as a measure of relative variability. For each layer in a network, it is noteworthy that the best-performing kernel is selected.

Network	Layers	Hardware	Mean Runtime (sec)	Mean std (sec)	Coeff. of Variation (%)	Layers pvalue < 1%
ResNet34	19	AMD R7-3700X	5.43E-04	5.35E-05	9.86E+00	4
		A64FX	3.22E-04	1.31E-05	4.07E+00	12
		A100	2.95E-05	1.03E-07	3.48E-01	6
Alexnet	13	AMD R7-3700X	8.84E-04	4.63E-05	5.24E+00	3
		A64FX	4.51E-04	1.26E-06	2.79E-01	6
		A100	4.66E-05	4.46E-08	9.58E-02	8

Figure 5.21: Coefficient of Variation and P-Value Test. The lower the p-value, the more statistically significant are the results reported in this section.

The *Standard deviation* measures how individual values deviate from the mean—a lower standard deviation indicating proximity to the mean. The p-value serves as a probability measure, assessing the strength of evidence against a null hypothesis, i.e., our techniques yield no effect. A lower p-value signifies stronger evidence against the null hypothesis. Conversely, a high p-value indicates a lack of statistical significance at the selected threshold. As shown in Figure 5.21, both the networks under test performed similarly on the four different hardware. We could observe a CV of 10% in the average runtime of the best kernel for **ResNet34** and 5% for **AlexNet**, which **DPAnsor** suggested for x86. ARM A64FX showed a lower CV with 4% for **ResNet34** and less than 1% for

AlexNet. The lowest **CV** was observed on the Nvidia A100, with values below 1% for both networks. We observed a noteworthy trend where many layers in each network exhibited **p-values** of less than 0.01. This indicates a statistically significant pattern per layer, suggesting that the observed results are unlikely to have occurred by chance.

5.15 Conclusion Remarks

This chapter demonstrated that by using Droplet Search to explore a reduced set of samples from **Ansor**, it is possible to improve its performance. We reduced the search time within the space generated by **Ansor**, and using a well-chosen seed as a starting point allowed Droplet Search to further decrease search time in neural networks. In the next chapter, we will explore how convergence algorithms can be applied to other areas, such as operator fusion in neural networks, and introduce a heuristic aimed at reducing the search space in operator fusion.

Chapter 6

Fusion of Operators of Computational Graphs via Greedy Clustering – The XNNC Experience

The utilization of convergence algorithms, such as Droplet Search, can expedite the discovery of optimal solutions. As demonstrated in Chapter 5, leveraging the seed provided by Ansor for the Droplet algorithm enables further refinement of the results through an exploitation search within the space generated by Ansor. This concept of convergence algorithms can be broadly applied to various domains, including operator fusion in neural networks. In this chapter, we introduce a novel heuristic designed to reduce the search space associated with operator fusion in neural networks. By diminishing the number of operators within a neural network, this heuristic decreases the search time required for convergence algorithms like Droplet Search. The generated space can be integrated with such algorithms, thereby enhancing their efficiency and efficacy. Our proposed heuristic is grounded in the idea that a judicious reduction of the search space can facilitate more effective exploration and exploitation of the solution landscape. By pruning the operator space, we can guide the search towards more promising regions, ultimately leading to improved convergence rates and enhanced solution quality. The synergy between our heuristic and convergence algorithms like Droplet Search has the potential to accelerate the optimization process, making it a contribution to the field of neural network optimization.

6.1 Introduction

Tensor compilers [11, 83, 117] represent machine-learning models as “*computational graphs*”, where vertices denote operations and edges represent the data flow among these operations. In this context, “*fusion*” means combining multiple operations of a computational graph into a single node [154, 110, 138, 20]. This form of fusion is a common

transformation in many tensor compilers, including Google’s XLA [60]; Apache TVM [35]; the ONNX runtime [90]; the PYTORCH just-in-time compiler [9] (via TORCHSCRIPT); NVIDIA’s TENSORRT [45]; Apache MXNET [97]; Intel’s NGRAPH [40]; and Meta’s GLOW [121].

As pointed out by Cai et al. [21, 20], fusion is conceptually equivalent to a problem called “*Directed graph partitioning with acyclic quotient graph*” (DGPAQ). This problem is NP-complete, as demonstrated by Moreira et al. [94]. Thus, compilers either exhaustively search all the ways to perform fusion, or resort to faster heuristics, as Section 2.3 explains. Exhaustive search was previously evaluated by Cai et al. [20, 21]. This algorithm, henceforth called the “*Backtracking*” approach, inspects all combinations of fused kernels. The algorithm uses dynamic programming to avoid recomputing graph configurations. Nevertheless, even augmented with memoization, the backtracking approach is expensive. Thus, industrial compilers use more straightforward and faster exploration techniques. As an example, Cadence’s Xtensa Neural Network Compiler (XNNC) explores fusion candidates via a depth-first search (DFS) of the computational graph. Decisions are greedy: for each edge visited in the computational graph, if fusion improves the estimated cost of the whole model, then it happens. In this case, weights are updated locally, as it impacts only edges that either reach or leave the fused kernels. Henceforth, we shall call this algorithm the “*DFS*” Approach. Although the DFS approach is fast and tends to generate relatively efficient code, this work shows that it can be improved upon.

Contributions This work introduces a new fusion algorithm that groups edges from the computational graph into clusters of fusion candidates. As Section 6.2 explains, at each stage, one of the clusters is augmented with a new edge. To this end, new cluster candidates are arranged into a priority queue sorted by estimates given by a cost model. As an analogy, the proposed algorithm is to XNNC’s old fusion methodology as a minimum-spanning tree is to a depth-first search. Even though this algorithm adds a linear component over the DFS approach when running on dense computational graphs, we can still demonstrate that it is only marginally slower, staying within a factor of 50% in machine-learning models whose computational graph contains more than 2,000 nodes.

In the convex space, the greedy choice is optimal, meaning that a greedy algorithm will always find the best solution without requiring complex global search strategies. This principle is particularly useful in operator fusion within neural networks, where convexity ensures that progressively combining operators into a single, more efficient operation leads to an optimal solution, simplifying the overall optimization process.

Summary of Results This proposed algorithm, henceforth called the *Minimum Spanning Tree* (MST) approach, is currently XNNC’s default fusion algorithm. In addition to describing our experience deploying this algorithm onto XNNC, this work also presents

the first overview of XNNC’s original algorithm (Sec. 6.2)—the DFS method—together with a brief overview of XNNC’s cost model (Sec. 6.5). Section 4.6 evaluates fusion on 19 machine-learning models deployed on three of Cadence’s digital signal processors, in a comparison that involves four different fusion techniques. The new MST algorithm yields machine code that is 39% faster than the old DFS approach (geometric mean across 19 models). In terms of quality of the machine code that it produces, this algorithm ties with the evolutionary approach that we have implemented after the TVM public-domain algorithm; however, it is twice as fast in terms of compilation time. And, in contrast to Cai et al. [20] optimal algorithm, the MST technique does not face timeouts; terminating within minutes in every one of our benchmarks.

6.2 Fusion as an Optimization Problem

As mentioned in Section 2.3.1, fusion tends to improve performance, because it improves data locality and reduces the overhead of computing loop conditions. However, fusion can also increase cache pressure, as the data required by a combined kernel might exceed the space available in fast memory. There are two ways to base fusion decisions: the *sampling approach* or the *static approach*. The former consists in running the model before and after fusion, and keeping the best result. This technique is common in the so called *Auto Schedulers*, such as Ansor [157] or Halide [113]. Static decisions, in contrast, are based on a *cost model*: a function that associates programs with estimates of their running time. Cost models are used in OPTIMUS [21, 20], TIRAMISU [13], LOOPER [89] and in XNNC, as Example 21 shows.

Example 21. *In XNNC’s cost model, the cost associated with the edges of the computational graph corresponds to cycles to transfer data around memory banks. Values associated with nodes, in turn, represent an estimate of cycles required to execute those nodes. This estimate includes cycles spent on computation and data transfer in case data cannot fit into the fast memory level.*

The cost of an entire computational graph is determined by the application of the cost model over all the operations (fused or not) and edges that form that graph. Costs are not *additive*: the cost of a fused node, henceforth denoted as $C(L1 \oplus L2)$, is not necessarily $C(L1) + C(L2)$. The value of $C(L1 \oplus L2)$ may be smaller or greater $C(L1) + C(L2)$. Additionally, cost models are expected to be neither commutative nor associative. Therefore, to find the cost model of a computational graph in a non-ambiguous

way, we compute it over a *Binary Dyck Language*¹ that determines the order in which kernels in the computational graph are fused.

Definition 7 (Binary Dyck Language). *Let $V = \{n_1, n_2, \dots, n_k\}$ be the k nodes that form a computational graph. We define a Binary Dyck Language L_V over V as the set of all the permutations of V parenthesized in the following way:*

$$\begin{aligned} S &::= C \mid S, C \\ C &::= n_1 \mid n_2 \mid \dots \mid n_k \mid (C, C) \end{aligned}$$

We shall define the *Operator Fusion Optimization Problem* (Definition 8) over Binary Dyck Languages because if $s \in L_v$ is a sentence in such a language, then it determines the exact order in which the cost of fused nodes should be evaluated. Computing the costs in the correct ordering is important because, as mentioned, cost functions are neither associative nor commutative. Example 22 shows instances of different sentences in a Binary Dyck Language.

Example 22. *If $s \in L_V$, then s determines the order in which nodes in V are fused. If we have a computational graph with three nodes, e.g., $\{n_1, n_2, n_3\}$, then $s = n_1, n_2, n_3$ represents a graph with no fused nodes; $s = (n_1, n_2), n_3$ represents a graph where n_1 and n_2 are fused; and $s = ((n_1, n_2), n_3)$ represents a graph where n_1 and n_2 are fused, and then the resulting node is fused with n_3 .*

Definition 8 (The Operator Fusion Problem). *Let $G = (V, E)$ be a computational graph; let $cost_e$ be a cost function for edges, e.g., $cost_e : E \mapsto \mathbb{R}$; and let $cost_v$ be computed over the Binary Dyck representation L_v of V as follows:*

$$\begin{aligned} cost_v C &= cost_e C \\ cost_v S, C &= cost_v S \oplus cost_e C \\ cost_e n &= C_v(n), \text{ cost of single node} \\ cost_e (C_0, C_1) &= cost_e C_0 \oplus cost_e C_1 \end{aligned}$$

Under the constraints of a non-additive, non-associative cost model, solving the operator fusion problem optimally involves inspecting an exponentially large number of sentences in L_v . First, because any permutation of the elements in V could be parenthesized. Nevertheless, even if we fix the order of the nodes that can be fused (a valid assumption, once we see the nodes of a computational graph as sequential code), we still have an exponentially large number of fusing orders, as Theorem 1 shows. Each of these orderings might yield a different cost, as Example 23 explains.

¹Broadly, a “Dyck sentence” is a balanced string of brackets, following the notation used by *Walther von Dyck* on formal languages.

Theorem 1. *The number of sentences of a Binary Dyck Language with n nodes is:*

$$C(n) = \sum_{k=1}^{n-1} \binom{k-1}{n-1} \cdot C(k) \cdot C(n-k)$$

Proof. Base case: $C(1) = 1$. There is only one way to arrange a single node.

Recursive case: If $|V| = n$, then we can split the elements into two non-empty parts:

- The first part has k elements.
- The second part has $n - k$ elements.

The number of valid binary trees (or binary Dyck strings) formed by these two parts is $C(k) \cdot C(n - k)$.

Since the n elements are distinct, the number of ways to assign k elements to the first part (and the remaining $n - k$ elements to the second part) is given by the binomial coefficient $\binom{k-1}{n-1}$, which accounts for choosing $k - 1$ elements from $n - 1$ elements (since one element is fixed at the root of the binary structure). $\square$

Example 23. *Figure 6.1 shows different implementations of fusion on a computational graph formed by four kernels. Numbers on vertices and edges denote the cost of implementing these elements, assuming a hypothetical cost model. The transformation that converts the graph in Fig. 6.1 (a) to the graph in Fig. 6.1 (b) reduces the estimated cost of the entire model; hence, it is profitable—as per the cost model. However, fusing nodes **BC** and **D**, as seen in Fig. 6.1 (c), increases the overall cost of the computational graph; hence, it is unprofitable. Fusion can also be impossible, as explained in Section 2.3.1. When such is the case, the cost model sets costs to infinite, as seen in Fig. 6.1 (d).*

6.3 Greedy Clustering

The main contribution of this work is a new operator fusion algorithm designed for computational graphs. This algorithm addresses the problem defined in Definition 8. To illustrate how the algorithm works, this section discusses it in the context of three other algorithms that are well-established both in industry and in the programming languages literature:

DFS: The algorithm previously used in the XNNC compiler, before being replaced by the MST-based approach that this work introduces.

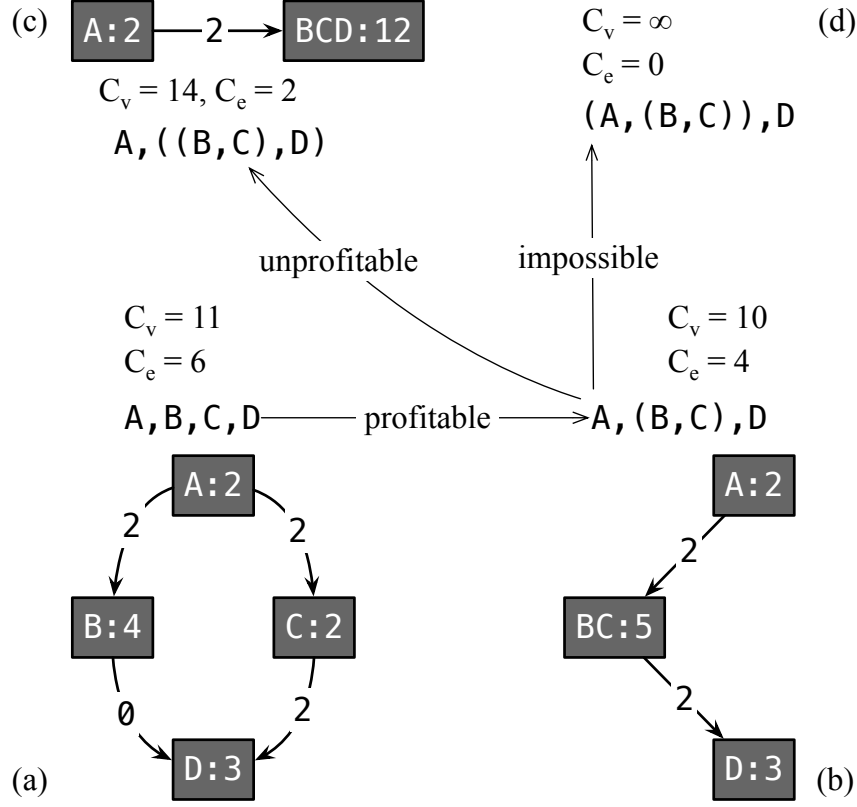


Figure 6.1: The order of operator fusion can be encoded as sentences in a Binary Dyck Language formed by the nodes of the computational graph.

BKT: The exhaustive search algorithm proposed by Cai et al. [20], which combines backtracking and dynamic programming.

EVL: The evolutionary search technique which we reimplemented in XNNC after the scheduling algorithm currently deployed in Apache TVM [35].

BKT [20, 21] and EVL [35] are the subject of previous work, and shall not be further discussed. The rest of this section focuses on high-level properties of the MST and the DFS fusion algorithms. Pseudocode is available in Section 6.4. All these algorithms receive a computational graph as input and produce, as output, another computational graph in which some kernels might be fused. Nevertheless, these algorithms differ in four key aspects of their design, which are enumerated below and summarized in Figure 6.2:

Candidates: The number of active fusion candidates the algorithm keeps alive at any given point of the search.

Criteria: How an edge of the computational graph is chosen among the candidates to be fused; hence, producing a new computational graph.

Exploration: The number of different concretizations of the computational graph that are actually explored during the search.

Memory: The number of different computational graphs that must be kept resident in memory at any given time of the search.

	MST	DFS	BKT	EVL
Candidates	$O(E)$ clusters C of vertices, $C \subseteq V$	$O(V)$ outgoing neighbors of the last visited vertex v in depth-first		Profitable graphs
Criteria for choosing next graph	$(C + w \rightarrow u)$, for $w \in V$, and $w \rightarrow u$ most reduces cost	$(v + v \rightarrow u)$, for $v \rightarrow u \in E$ that most reduces cost	$(v + v \rightarrow u)$, for every $v \rightarrow u \in E$	Profitable crossovers
Number of explored graphs	1	1	$O(E !)$	$\text{Pop} \times \text{Gen}$
Active graphs	1	1	1	Pop
Internal State	Clusters + G	$v + G$, $v \in V$	$O(E !) + G$	$\text{Pop} \times G$

Figure 6.2: Comparison of different fusion algorithms. The Evolutionary Algorithm (EVL) is parameterized by a Population (Pop) of best graph candidates, plus a number of Generations (Gen).

6.3.1 The Linear Search

We call a *concretization* of the computational graph a version that can be translated into machine code and deployed onto actual hardware. Each concretization of the computational graph can be described by a sentence of a Binary Dyck Language over the set of nodes it contains, as formalized in Definition 7. The four algorithms in Figure 6.2 explore different configurations of the computational graph that must be optimized. Two of these algorithms, DFS and MST, perform a linear search over the universe of possible concretizations of the computational graph. Linearity, in this context, implies that these two algorithms satisfy the following properties:

Single graph: these algorithms keep a single active graph G resident in memory at any given point in the search process.

Strict Reduction: If a new graph G' is produced by fusing two kernels of G , then G' is necessarily smaller than G : it contains one less node.

The first property, single graph, is also present in Cai et al. [20]’s backtracking algorithm. Thus, while the BKT algorithm might explore multiple different graphs, it only keeps one of them resident in memory at any given point². However, the second property, reduction, is not present in BKT: fused kernels can be split due to backtracking. In contrast, none of these properties is present in EVT: the evolutionary algorithm keeps a population of graphs in memory. Furthermore, although the same graph might never be visited twice due to caching, crossover might split previously fused kernels from one generation to the next.

6.3.2 Criteria to Select Next Fusion Candidate

Each of the four algorithms in Figure 6.2 maintains an *internal state*, which they use to select the next concretization of the computational graph that must be explored. In this regard, the most frugal algorithm is DFS, as its internal state consists exclusively of the current concretization of the computational graph plus the current active vertex $v \in V$. This vertex is the last node visited by the depth-first search. MST, in turn, keeps in memory *clusters* of fused nodes, in addition to the current concretization of the computational graph. Given this internal state, when choosing the next concretization, the algorithms proceed as follows:

DFS: The algorithm traverses edges of the computational graph in topological order.

Whenever it visits an edge $v \rightarrow u$, it fuses v and u if the fusion is profitable, reducing the estimated cost of the entire computational graph. Example 24 illustrates the approach.

MST: Let $\{C_1, \dots, C_n\}$ be the set of clusters in MST’s internal state. The algorithm chooses the edge $v_i \rightarrow v_j, v_i \in C_i, v_j \in C_j, i \neq j$ that minimizes the estimated cost of the entire computational graph. If no such edge exists (*e.g.*, the current cost estimate cannot be reduced by merging any two clusters), then the algorithm stops. Example 25 illustrates this step.

Examples 24 and 25 explain how DFS and MST decide which kernels are fused. As discussed in Section 6.2, these algorithms rely on a cost model to guide these decisions. The actual implementation of the cost model is not essential. However, for the sake of illustration, we shall assume an abstract cost model that meets the following properties:

²Notice that BKT is still memory hungry, for it holds a memoization table with the costs of different operator fusions, to avoid recomputations.

Additive: $C_v(a \oplus b) = C_v(a) + C_v(b)$

Perfect: If a and b are fused, then $C_e(a \rightarrow b) = 0$

Bounded: Fusion is only possible if the sum of incoming and outgoing edges of the fused node is less than a constant limit

The **Bounded** constraint simulates a running environment with one memory level of bounded capacity. Thus, fusion is only possible if it does not overload the available memory.

Example 24. Figure 6.3 illustrates how fusion decisions are taken in the DFS approach, assuming a memory capacity of five storage units. The example assumes the following topological ordering of nodes: A, B, D, E, C . Thus, the depth-first search starts visiting kernel A and traversing the edge $A \rightarrow B$. The fusion of nodes A and B is possible, because the sum of incoming and outgoing edges of the resulting node AB is only four. Fusion has a positive effect, as it eliminates the cost of edge $A \rightarrow B$. The algorithm can perform yet another fusion, producing a combined node ABD . It yields a concrete computational graph of estimated cost 18; i.e., $C_v = 8$ and $C_e = 10$.

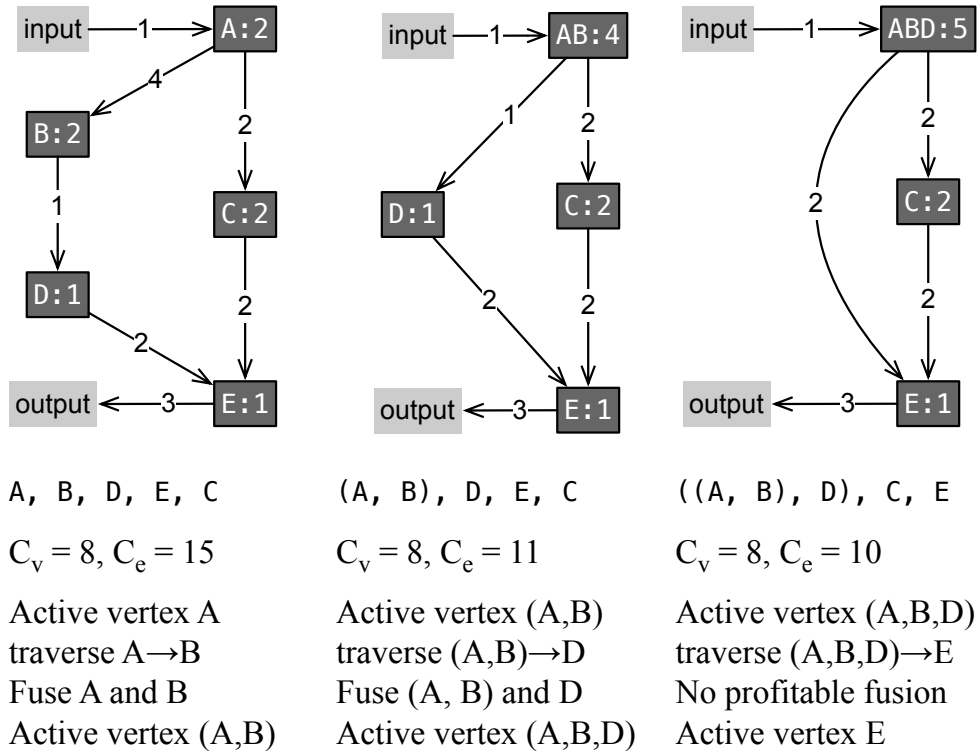


Figure 6.3: Node fusion via the DFS approach, assuming a memory of five storage units. Thus, fusion is only possible if the cost of the incoming and outgoing edges of the fused node is less than five.

Example 25. Figure 6.4 shows how the MST algorithm takes fusion decisions. The algorithm starts with five clusters, each a singleton containing one kernel of the computational graph. In the first step, it fuses clusters $\{A\}$ and $\{B\}$, as this fusion yields the most drastic cost reduction: four. It is still able to fuse $\{AB\}$ and $\{C\}$, for a new profit of two memory transfer units. In this example, the MST outperforms DFS by one cost unit, as its final estimated cost is 17; i.e., $C_v = 8$ and $C_e = 9$.

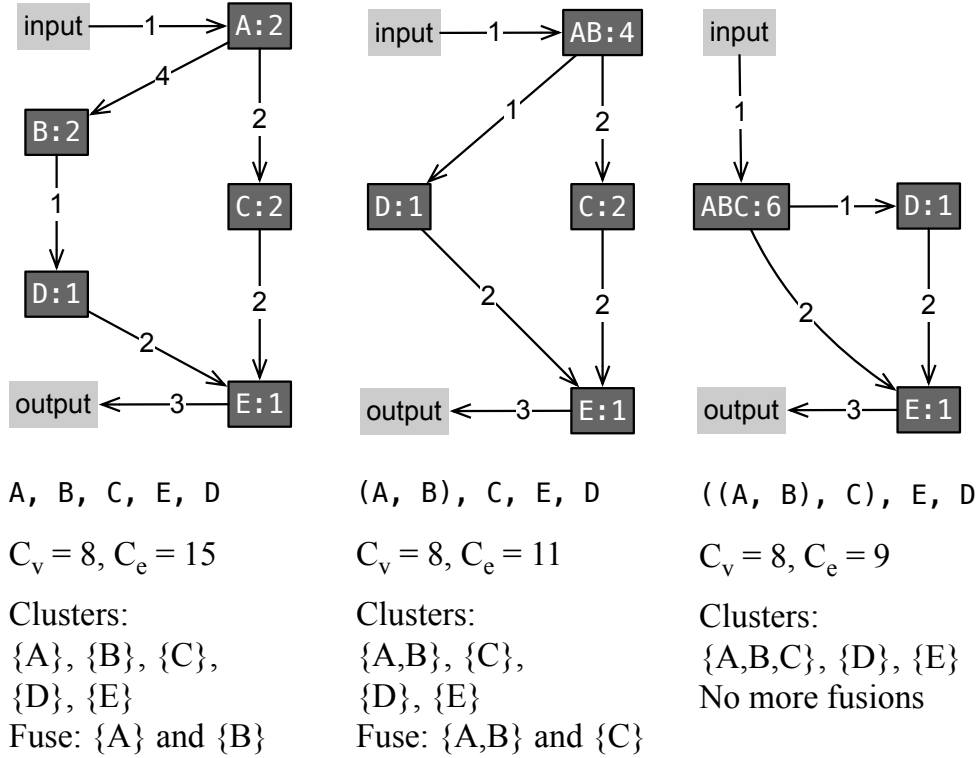


Figure 6.4: Node fusion via the MST approach, assuming a memory of five storage units.

6.3.3 Asymptotic Complexity

As previously explained, all the algorithms in Figure 6.2 are parameterized by the same cost model. To calculate this cost, XNNC needs to find the best implementation for the operation, a task that involves determining tiling sizes and doing micro-kernel selection. This process is costly, because it might involve testing a large amount of code configurations. In our experience, the invocation of the cost model is the most expensive single operation that any of these algorithms perform. As we will explain in Section 6.5.1, XNNC's cost model runs a static analysis on a subset of the computational graph's nodes, including expensive components such as points-to analysis and liveness analysis. Thus,

the more often the algorithm consults the cost model, the longer its running time will be. Theorem 2 puts a bound on the number of queries performed by DFS, and Theorem 3 does the same for MST.

Theorem 2 (Complexity of DFS). *In the worst case, DFS runs $O(|V|)$ queries to the cost model.*

Proof. DFS visits each vertex of the computational graph once, and for each vertex v , it chooses a neighbor u in $O(1)$, as given by the graph's topological ordering. When edge $v \rightarrow u$ is visited, the cost model is invoked once to estimate the cost of the fused node, *e.g.*, $C(v \oplus u)$. Thus, in total, DFS runs $O(|V|)$ queries to the cost model. $\square$

Notice that the bound in Theorem 2 still applies to dense computational graphs that might contain $O(|V|^2)$ edges: the cost model is queried only when the search moves from one visited node to a non-visited node (which then becomes visited). MST, in turn, might visit the same cluster more than once; however, the algorithm is guaranteed to terminate in $O(V)$ steps, as Lemma 1 shows.

Lemma 1 (Termination of Clustering). *When processing a computational graph $G = (V, E)$, the MST search terminates within at most $O(|V|)$ steps.*

Proof. At any given time of the search procedure, MST chooses two clusters C_i and C_j , based on the Criteria discussed in Section 6.3.2, and merges them. Merging only happens if $C(C_i \oplus C_j) < C(C_i) + C(C_j)$. If no such pair of clusters exists, the algorithm stops. The transformed computational graph necessarily contains one less vertex. Therefore, this step can happen at most $O(|V|)$ times. $\square$

Theorem 3 (Complexity of Clustering). *MST runs $O(|V|^3)$ queries to the cost model without memoization. It runs $O(|V|^2)$ queries with memoization.*

Proof. MST starts the search procedure with $|V|$ clusters, each formed by a single vertex. It runs $O(|E|) \approx O(|V|^2)$ queries to find the first pair of nodes to merge. At any given step, it must run all these queries again to find the pair of nodes to merge. As indicated by Lemma 1, it runs $O(|V|)$ steps, for a total of $O(|V|) \times O(|V|^2) = O(|V|^3)$ queries. However, cost estimates only change once, for the merged nodes; hence, they can be cached. In this case, we have one query at any given step for a final worst-case complexity of $O(|V|^2)$ queries. $\square$

Figure 6.5 summarizes the asymptotic behaviors discussed in this section. Notice that Figure 6.5 is only concerned with the number of queries each algorithm sends to the cost model. MST still must maintain a priority queue of edges, which contain $O(|E|)$ elements at any given step of the algorithm. The priority of each edge $u \rightarrow v$ is given by $C(u \oplus v)$. Building the initial priority list is $O(|E| \times \ln(|E|))$. Updating it is $O(\ln |E|)$. Notice also that the results stated in Theorem 3 refer to worst-case results. In practice, computational graphs tend to be sparse, meaning that $O(|V|) \approx O(|E|)$. In this case, the

MST algorithm's asymptotic behavior becomes much closer to that of the DFS approach. This behavior is corroborated by the empirical data discussed in Section 6.6.2.

	DFS		MST	
	Worst	Expected	Worst	Expected
Graph rewrites (Concrete graphs)	$O(V)$			
Cost estimates (with memoization)	$O(V)$		$O(V ^3)$	$O(V)$
Memoized costs	–		$O(V ^2)$	$O(V)$

Figure 6.5: Asymptotic behavior of the different fusion algorithms discussed in this work, where V is the set of nodes in the computational graph.

6.4 Pseudocode

This section provides a high-level description of the different algorithms that this work discusses. Section 6.4.1 focuses on the DFS-based approach, while Section 6.4.2 presents the pseudocode of the MST-based approach.

6.4.1 The DFS-Based Fusion Algorithm

Algorithm 1 performs a Depth-First Search (DFS) on a graph G to identify clusters of edges that can be fused for optimization. It begins by initializing a list of all graph edges (line 3) and a set of visited edges (visited). While there are unprocessed edges (line 6), the algorithm pops an edge from `list_edges` and adds it to `ready_set` (lines 7-8). It then iteratively processes each edge in `ready_set` (line 11), checking if the edge has not been visited (line 13). If not, it adds the edge to a potential fusion cluster (line 14) and evaluates the fusion cost (line 16). The cost model (as seen in Section 6.5.1) is invoked at line 15. This cost is invoked once per edge, as it is guarded by the test in line 13. If the fusion cost is lower than the non-fusion cost (line 16), Algorithm 1 adds the edge to the visited set and the cluster to `best_cluster_cut` (lines 17-18). This process continues until all edges are evaluated for potential fusion, and the final list of fused clusters is returned (line 26). Notice that termination is assured, because edges are not inserted in the list of

Algorithm 1 Depth-First Search (DFS)

```

1: Input: Graph  $G$ 
2: Output: List clusters
3:  $list\_edges \leftarrow G.edges$  (in DFS order)
4:  $visited \leftarrow Set()$ 
5:  $fusion\_cluster \leftarrow List()$ 
6: while  $list\_edges$  is not empty do
7:    $next\_edge \leftarrow list\_edges.pop()$ 
8:    $ready\_set \leftarrow Deque([next\_edge])$ 
9:    $cluster \leftarrow List()$ 
10:   $best\_cluster\_cut \leftarrow None$ 
11:  while  $ready\_set$  is not empty do
12:     $edge \leftarrow ready\_set.popleft()$ 
13:    if  $edge \notin visited$  then
14:       $cluster.append(edge)$ 
15:       $val \leftarrow evaluate\_cluster(cluster)$ 
16:      if  $val \leq cost\_of\_no\_fusion(cluster)$  then
17:         $visited.add(edge)$ 
18:         $best\_cluster\_cut \leftarrow cluster$ 
19:      end if
20:    end if
21:  end while
22:  if  $best\_cluster\_cut$  is not  $None$  then
23:     $fusion\_cluster.append(best\_cluster\_cut)$ 
24:  end if
25: end while
26: return  $fusion\_cluster$ 

```

edges ($list_edges$) within the loop in line 6. Similarly, the $ready_set$ is never augmented within the loop in line 11.

6.4.2 The MST-Based Fusion Algorithm

The algorithm **MST** (Algorithm 2) clusters edges of a graph G with the goal to maximize the cost function (Equation 6.5.1). The goal of this algorithm is to fuse operators in a computational graph G by identifying clusters of operators that can be combined into a single operator. The algorithm returns a list of such clusters.

Algorithm 2 starts by listing all edges (line 4) and initializing max-heap data (line 5) and a set of visited clusters (line 6) to avoid revisiting clusters. The use of a heap allows the algorithm to prioritize clusters based on their cost-effectiveness. It stores clusters along with their associated costs for later processing. For each edge (line 7), Algorithm 2

Algorithm 2 Minimum Spanning Fusion Tree (MST)

```

1: Input: Graph  $G$ 
2: Output: List clusters
3:  $list\_edges \leftarrow G.edges$ 
4:  $data \leftarrow HeapMAX()$ 
5:  $visited \leftarrow Set()$ 
6: for each  $edge$  in  $list\_edges$  do
7:    $cluster \leftarrow [edge]$ 
8:   for each  $next\_edge$  in  $list\_edges$  do
9:      $cluster.append(next\_edge)$ 
10:    if not  $is\_fusible(cluster)$  then
11:      break
12:    end if
13:  end for
14:   $val \leftarrow cost\_function(cluster)$ 
15:  if  $val \leq cost\_of\_not\_fusion(cluster)$  then
16:     $data.insert((val, cluster))$ 
17:  end if
18: end for
19:  $clusters \leftarrow []$ 
20: for each  $(val, cluster)$  in  $data$  do
21:   if  $cluster \notin visited$  then
22:      $visited.add(cluster)$ 
23:      $new\_edge \leftarrow G.fusion(cluster)$ 
24:      $new\_edge.val \leftarrow val$ 
25:      $clusters.append(cluster)$ 
26:   end if
27: end for
28: return  $clusters$ 

```

creates a cluster and iteratively adds subsequent edges (lines 8-10) to maintain a “fusible” cluster (line 11). Once a cluster is formed, it evaluates the cluster (line 14) and inserts it into the heap (line 15). The algorithm calculates the cost of the current cluster using the cost model from Section 6.5.1. It compares this cost with the cost of not performing the fusion ($cost_of_not_fusion(cluster)$). If the fusion cost is less than or equal to the cost of not fusing, the cluster and its cost are inserted into the data heap.

After processing all edges, an empty list cluster is initialized (line 17). The algorithm then iterates through the clusters in the heap (line 18), and if a cluster C has not been visited (line 19), it runs the following sequence of steps:

1. Marks C as visited (line 20)
2. Fuses C into a new edge with the graph G (line 21)
3. Assigns the C ’s evaluation value to the new edge (line 22)

4. Appends C to the result list (line 23).

Once every cluster has been visited and expanded, Algorithm 2 returns the list of clusters (line 26). The termination of the Minimum Spanning Fusion Tree (MST) algorithm is guaranteed by the following factors:

1. The algorithm operates on a computational graph G , which consists of a finite number of edges.
2. The list of edges (`list_edges`) that the algorithm processes in the outer loop (Line 6) is finite. Since the graph G has a finite number of edges, the loop will execute a finite number of times.
3. Within the inner loop (Lines 8-13), the algorithm iterates over `list_edges` again, but it stops expanding the cluster as soon as it encounters a situation where the cluster is no longer fusible (`is_fusible(cluster)` returns false).
4. Since `list_edges` is finite, the maximum number of iterations of the inner loop is also finite. Furthermore, the early termination of this loop when the cluster becomes non-fusible ensures that it does not run indefinitely.
5. The `HeapMAX` data structure (`data`), which stores potential clusters and their associated costs, is finite because it can only contain a finite number of clusters (each cluster is a combination of edges, and the number of such combinations is finite).
6. The operations performed on the data heap, such as insertion and retrieval, are also finite and bounded by the number of edges in the graph.
7. The final loop (Lines 18-24) that processes the clusters stored in `data` will also terminate since it iterates over a finite set of cluster-cost pairs stored in the heap.
8. Each iteration of the algorithm either adds a new cluster to the visited set or moves on to the next edge if the cluster has already been visited.
9. Therefore, since the number of edges (and thus potential clusters) in the graph is finite, the algorithm will eventually exhaust all possible clusters and terminate.

Lemma 1 (Page 107) subsumes the above list of steps, e.g.: the MST algorithm is guaranteed to terminate because it processes a finite number of edges, performs a bounded number of operations on each edge, and iterates over a finite set of possible clusters.

6.5 Operator Fusion in XNNC

The ideas described in this work are currently implemented in the Xtensa Neural Network Compiler (XNNC), a toolchain for deploying machine learning models on Cadence’s Tensilica processors [18]. These edge-device processors are popular in applications such as automotive, consumer electronics, communications, LiDAR and Radar systems. XNNC’s first implementation was released in 2017 to complement Tensilica’s Vision 7 processors³. That initial design featured quantization capabilities, allowing neural networks to utilize 8-bit data types while maintaining less than 1% loss in model accuracy [10]. Currently, XNNC is at version 3.0 and its code base consists of about 1.15 million lines written in C++. It has already been used to compile thousands of different neural networks to a wide range of XTENSA architectures[19].

XNNC takes an ML model in either the ONNX or the TensorFlow format as input and converts it into a computational graph. It optimizes the graph by applying techniques like quantization and operator fusion [10]. XNNC uses these transformations to generate optimized C code for each node in the graph, tailored to a specific Tensilica digital signal processor. One of these optimizations combines loop tiling with DMA operations to transfer input and output tensor tiles between main memory and local memory, as Example 26 shows.

Example 26. *Figure 6.6 (a) shows a high-level view of the code that implements a kernel for an operator of a computational graph. The kernel generated by XNNC consists of a set of nested loops that iterate over tiles in the input and output tensors, transferring data between main and local memory. This allows the micro-kernel operators to process the data stored in the local memory efficiently. Graph operator fusion allows the micro-kernels that implement different operators to be executed on the same tiled loop. In the XNNC setting, the main benefit of fusion is the reduction in memory transfers between main and local memory, as shown in Figure 6.6 (b). Additionally, when operators are fused, a post-optimization process called “micro-kernel” selection might replace multiple micro-kernels with a single one.*

³<https://www.eetimes.com/cadence-last-holdout-for-vision-ai-programmability/>

```

# For each tile (ti, tj)
for ti in W by tw # Iterate by tile width (tw)
  for tj in H by th # Iterate by tile height (th)
    # Copy the tile into the local memory
    DMA_in(in_tensor[ti*tw:ti*(tw+1)][tj*th:tj*(th+1)], local_tensor_1)
    # Perform the node operation via a micro kernel:
    operator_A_microkernel(local_tensor_1, local_tensor_2)
    # Copy the resulting tile to global memory
    DMA_out(local_tensor_2, out_tensor[ti*tw:ti*(tw+1)][tj*th:tj*(th+1)])

```

(a)

```

# For each tile (ti, tj)
for ti in W by tw # Iterate by tile width (tw)
  for tj in H by th # Iterate by tile height (th)
    # Copy the tile into the local memory
    DMA_in(in_tensor[ti*tw:ti*(tw+1)][tj*th:tj*(th+1)], local_tensor_1)
    # Perform the node operations via two, instead of one, micro kernel:
    operator_A_microkernel(local_tensor_1, local_tensor_2)
    operator_B_microkernel(local_tensor_2, local_tensor_1)
    # Copy the resulting tile to global memory
    DMA_out(local_tensor_1, out_tensor[ti*tw:ti*(tw+1)][tj*th:tj*(th+1)])

```

(b)

Figure 6.6: (a) Code generated for a single operator A of the computational graph. (b) Code generated for two fused operators ($A \circ B$).

6.5.1 The XNNC Cost Model

As already discussed in Section 2.3, fusion, once implemented as a static code optimization, must be guided by a cost model. XNNC uses a cost model to determine which pairs of graph operators are most beneficial to fuse. This cost model, henceforth referred to as XNNC’s cost model, is based on Equation 6.5.1, which is parameterized by two constants: the memory weight (w_m) and the cycle weight (w_c). The cost model is defined as the weighted sum of two components: memory and cycle costs.

Equation 6.5.1. *Estimated Cost Function*

$$\begin{aligned}
 C_e &= \frac{\sum_i^{neigh(e)} e_i.in_byte + e_i.out_byte}{bytes_per_cycle} \\
 C_v &= \sum_i^{neigh(v)} v_i.cycles \\
 cost &= w_m * C_e + w_c * C_v
 \end{aligned}$$

The memory cost is computed by dividing the bandwidth of an edge in the computational graph by the `bytes_per_cycle` parameter from the DMA, and then multiplying

by the memory weight. The cycle cost is the product of the cycle weight and the number of cycles required to produce the data that flows through the edge. To compute this cost, XNNC uses a performance model representing the operations necessary to produce this data. This performance model is implemented as a Python function, which is synthesized as the product of a proprietary static analysis.

Figure 6.7 shows the correlation between the cycle estimates produced by the cost model and the actual observations obtained from cycle-accurate simulations across the 19 ML models listed in Figure 6.8 (Page 116). The correlation coefficient between the estimated and observed results is 0.99. The correlation is not perfect in absolute terms—one estimated cycle does not directly correspond to one simulated cycle; however, the rank correlation is nearly perfect. Regardless of the fusion algorithm used, when the cost model predicts that one version of a neural network, nn_1 , is faster than another version, nn_2 , this behavior is consistently observed. To validate this, the Spearman rank correlation between the estimates and simulation results—comparing the performance of the original and optimized models across all 19 models and four fusion algorithms—remains above 0.95.

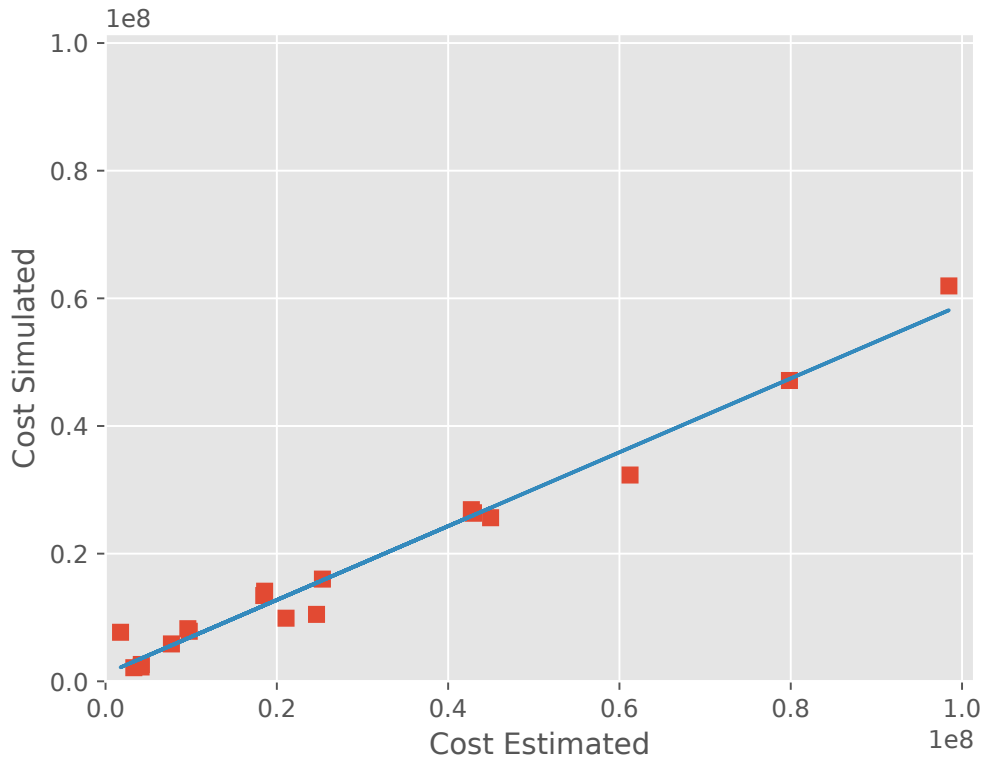


Figure 6.7: Ratio between estimated and stimulated cycles for models produced with the MST algorithm using DSP P6. Each point represents a different model.

6.6 Evaluation

This section presents empirical results that justify the deployment of MST as the default fusion algorithm of XNNC, instead of the other algorithms listed in Figure 6.2. To this end, this section evaluates the five research questions below:

RQ1: What is the impact of fusion on the speed of different deep-learning models?

RQ2: What is the time required to run the different fusion algorithms?

RQ3: What is the number of operations (layers) fused by the algorithms evaluated in this work?

RQ4: What is the impact of fusion on the size of the binary code that implements the deep-learning models?

RQ5: Does the relative impact of the different fusion algorithms vary across different Cadence DSPs?

Hardware and Software. XNNC runs on general-purpose CPUs and the code that it produces runs on Cadence Tensilica processors. The experiments in this section run XNNC 3.0.0 on an Intel(R) Xeon(R) Gold 6248, running at 2.5GHz, with 40 cores, data and memory cache instructions of 32K each (first level), and 28MB L3 cache with 754GB of RAM. To evaluate the models, these experiments use the XTENSA XPLOER 10.1.11 cycle-accurate simulator with three different digital signal processors: P1, P6 and Q8 [46]. Section 6.6.5 presents some specifications of these DSPs. Notice that we report the performance of different deep-learning models in CPU Cycles via cycle-accurate simulation.

Benchmarks. This section evaluates kernel fusion on 19 neural networks obtained from Torchvision version 0.15.0 [86]. Figure 6.8 shows the number of operators and edges per model. Models are represented in the ONNX format. All the models are tested with the same input, which consists of images of shape: batch=1, channels=3, height=224, width=224.

The Competing Algorithms. Experiments in this section use the four algorithms in Figure 6.2, in addition to the baseline approach, which consists of disabling fusion. All the algorithms are implemented on XNNC, use the same cost model, and process neural networks on the ONNX representation.

network	nodes	edges	network	nodes	edges
alexnet	56	89	resnet50	296	496
vgg11	76	122	googlenet	460	743
vgg13	84	136	shufflenet	484	800
vgg16	96	154	resnet101	568	955
vgg19	108	178	resnet152	568	955
resnet18	132	217	inception_v3	684	1116
squeezenet	160	264	densenet121	996	1613
resnet34	228	377	densenet161	1312	2127
mnasnet1_0	264	464	densenet169	1376	2231
mobilenet_v2	264	464			

Figure 6.8: Neural networks used in this evaluation.

6.6.1 RQ1: Impact of Fusion on Performance

The benefit of fusion is performance, measured in CPU cycles. The faster the code produced for computational graphs, the better the code generation algorithm. This section evaluates this parameter—performance—on 19 models simulated on Cadence Tensilica’s P6 digital signal processor.

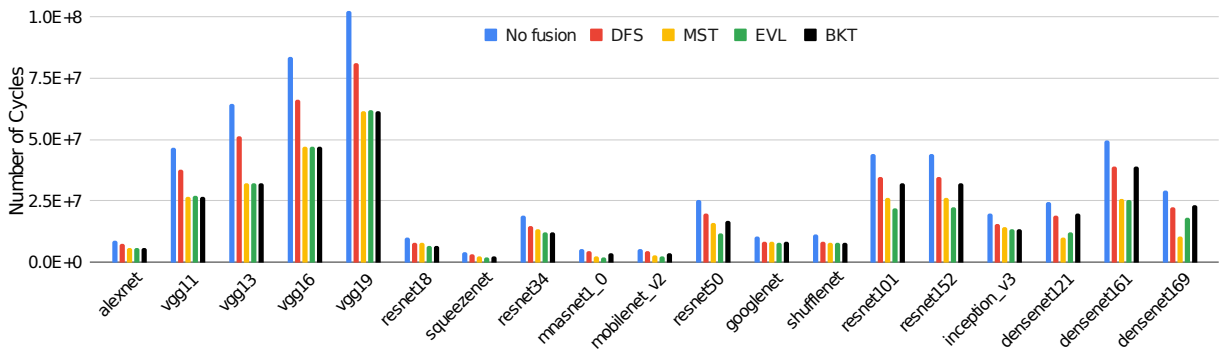


Figure 6.9: Number of cycles spent running 19 deep learning models on a cycle-accurate simulator for the DSP P6. Smaller is better.

Discussion Figure 6.9 compares the performance of the code produced by the fusion algorithms in Figure 6.2. The benefits of fusion vary depending on the model. Modest improvements were observed with the various DENSENET architectures, where MST-based fusion boosted performance by only 26%. In contrast, networks such as MNASNET1_0 and MOBILENET_V2 experienced speedups exceeding 2x. Nonetheless, in every case, MST consistently delivers the best results or is tied to the evolutionary approach. Both of these algorithms outperformed BKT, which was hampered by timeouts. As discussed in

Section 6.6.2, the exhaustive search failed to terminate within the allocated time for 12 models.

6.6.2 RQ2: Impact of Fusion on Compilation Time

This section analyzes the time each fusion algorithm takes to generate code for neural networks. The reported compilation time includes time to perform fusion and to generate code. We report the latter because fusion choices impact it: the fewer operators remain on the computational graph, the faster XNNC can produce machine code for it. Fusion’s search time, in this section, is limited to 60 minutes. After this time, the search stops, and the code generator returns the best candidate found thus far. Timeouts occurred only when running BKT.

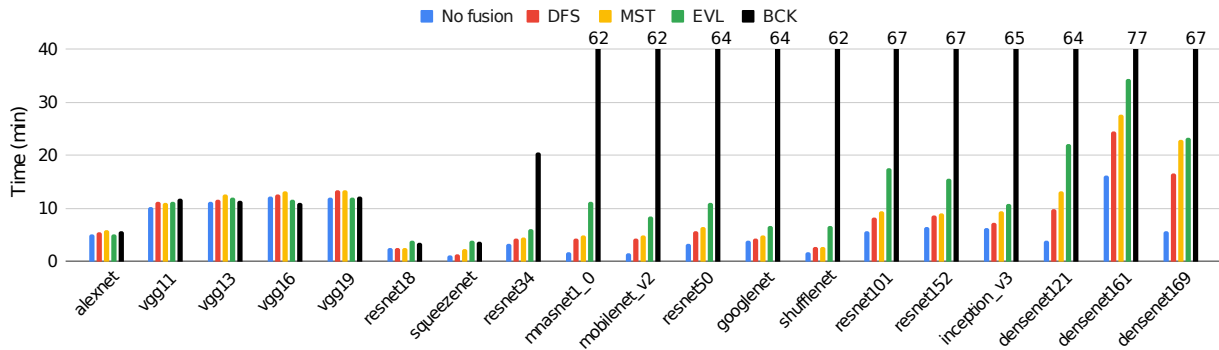


Figure 6.10: Time spent to execute each of the algorithms during the generation of neural networks using the XNNC compiler for the DSP P6.

Discussion Figure 6.10 reports compilation times. The time reported by the “no fusion” approach corresponds to code generation only. Except for “no fusion”, the original XNNC’s algorithm, DFS, is consistently the fastest approach. MST is 11% slower than DFS, and EVL is 29% slower than MST. BKT is the slowest algorithm, experiencing timeouts in larger neural networks. We emphasize that these results do not contradict findings reported by Cai et al. [20, 21]. As we explain in Section 3.3, our attempts to run Cai et al. [21]’s artifact on our largest networks met with non-termination.

6.6.3 RQ3: On the Number of Fused Operators

The number of operators and edges that remain in the deep-learning model’s final implementation varies depending on the fusion algorithm. This section analyzes this number. We report results obtained using the P6 DSP’s cost model; however, the other DSPs yield similar results. More fused nodes (Fig. 6.11) might lead to faster implementations of deep-learning models (Fig. 6.9); however, as explained in Section 6.2, this correlation is not perfect.

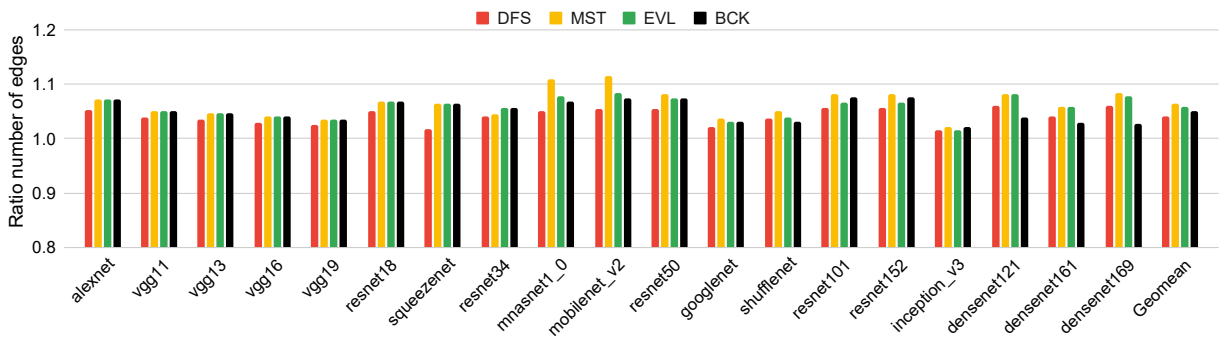


Figure 6.11: Reduction in the number of edges after fusion. Each bar is the number of edges before fusion divided by the number of edges after fusion. The higher the bar, the more edges were removed.

Discussion Figure 6.11 shows the ratio of edge reduction achieved by the different fusion algorithms. The reductions in the number of edges are relatively small, with SMT having a slight advantage: DFS = 4.2%, MST = 6.4%, EVL = 5.8%, and BKT = 5.2% (geometric mean of the percentage reduction across 19 models). Fusion is constrained by the amount of on-chip memory available to the DSPs: 128kb local memory for P6. Nevertheless, the effects of fusion are significant. For example, MST eliminates 172 edges from DENSENET169, reducing its edge count from 2,231 to 2,059.

6.6.4 RQ4: Impact of Fusion on Binary Size

We define a deep-learning model’s “binary size” as the size, in bytes, of the object file that implements this model when deployed on a given digital signal processor. This section analyzes this size, using as reference the P6 DSP. The more extensive the application of fusion, the larger the reduction in size. Nevertheless, this reduction tends to be small, even in settings where fusion can be applied aggressively. Code-size reduction ensues

from deleted loop instructions (counter initialization, guard and increment), and from optimizations that fusion enables, such as more effective constant propagation. Fusion can also lead to code expansion, as it might enable optimizations such as loop unrolling and tiling, which add instructions in the final binary.

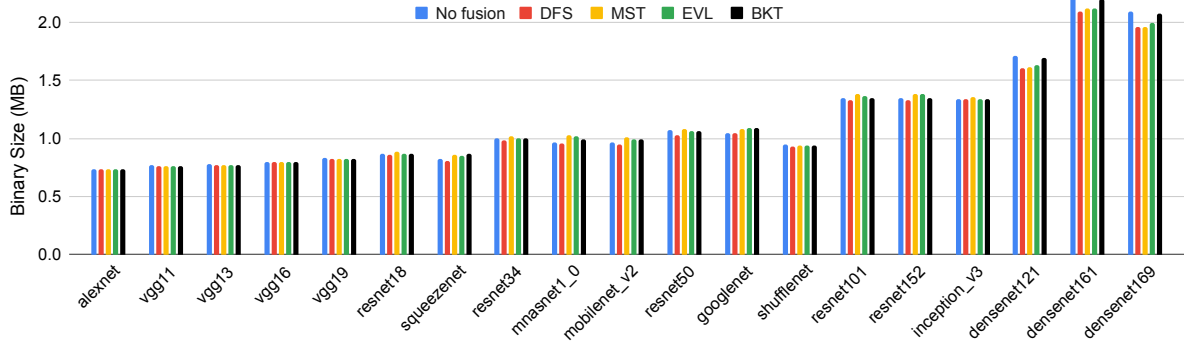


Figure 6.12: Binary size of the fusion algorithm against 19 neural network models for the DSP P6.

Discussion Figure 6.12 shows the size, in megabytes, of the object file deployed on the P6 DSP, when compiled with different fusion algorithms. The sum of binary sizes of the 19 deep-learning models produced by the different approaches is: no fusion=21.67 MB; DFS = 21.12 MB; MST = 21.61 MB; EVT = 21.54 MB and BKT = 21.71 MB. In general, fusion leads to some code-size reduction independent of the approach. However, only DFS caused consistent reductions. All the other fusion approaches met with cases, among the 19 benchmarks, where we could observe some code expansion. Expansion, when it happens, is small. The most significant ratio observed for MST was an increase from 0.97 MB to 1.03 MB when compiling MNASNET1_0.

6.6.5 RQ5: Fusion Across Different DSPs

The cost model that guides fusion decisions depends on hardware parameters, such as the size of memory banks and the latency of memory operations. This section aims to provide evidence that the MST approach is competitive across different hardware and, hence, across different cost models. To this end, it analyzes the fusion behavior on three DSPs: the P1, the P6 and the Q8 from Cadence Tensilica. These products are SIMD machines designed to run computer-vision applications that require analyzing images and identifying objects (such as pedestrians in an autonomous vehicle). While they are configured to support memory banks of the same size, they differ in how much data they

can load or store via vector operations, which is why the cost model varies depending on the hardware. The P1, P6 and Q8 have 128-bit, 512-bit and 1024-bit SIMD lanes, respectively. These SIMD registers can be used to implement 16-bit and 32-bit floating point operations (e.g., four 32-bit or eight 16-bit registers, in the case of P1). The Q8 can also implement sixteen 64-bit registers using one 1024-bit SIMD register.

Discussion Figure 6.13 compares the performance of the different fusion algorithms on the three digital signal processors. The figure presents summarized data. Each summary is the ratio of absolute quantities using “no fusion” as the dividend. For instance, Figure 6.13 (left) shows speedups as the sum of cycles taken for all the 19 models compiled without fusion divided by the sum of cycles of the 19 models compiled with fusion. Thus, the higher, the better. In this setting, MST and EVL show very similar results. The former is slightly superior on Q8, and somewhat inferior on P6.

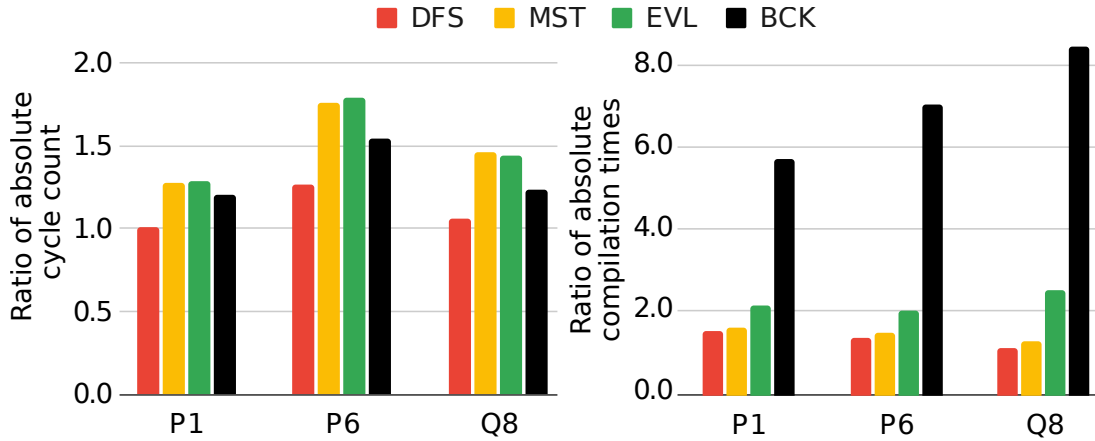


Figure 6.13: Behavior of fusion on different DSPs. (Left) Aggregated speedup of fusion on 19 models: higher is better. (Right) The aggregated slowdown in compilation time on 19 models: lower is better.

Figure 6.13 (right) shows the ratio between compilation times, i.e., the total time taken to compile all 19 models with fusion divided by the total time to compile the models without fusion. Therefore, the lower the bar, the faster the fusion algorithm is. These ratios, on the P1 are DFS = 1.6x slower; MST = 1.7x; EVL = 2.2x and BKT = 5.8x. Ratios observed on the P6 are DFS = 1.4x slower; MST = 1.5x; EVL = 2.1x and BKT = 7.1x. Finally, ratios observed on the Q8 are DFS = 1.1x slower; MST = 1.3x; EVL = 2.6x and BKT = 8.5x.

6.7 Conclusion Remarks

This chapter shows the contributions of the proposed fusion algorithms and the impact they had on optimizing the computational graph performance. Specifically, the evaluation of the Minimum Spanning Tree (MST)-based fusion approach revealed improvements in edge reduction and memory usage. The next chapter concludes this thesis with a summary of all the contributions of this work.

Chapter 7

Conclusion

This work has introduced a new kernel scheduling technique—droplet search—and has demonstrated its effectiveness by optimizing the code of six different deep learning models on six different hardware architectures. Droplet Search relies on the following observation: the optimization space formed by code transformation parameters usually determines a convex region that includes the origin of this space (no optimization) and the best kernel configuration that this space contains. Experiments show that Droplet Search tends to find kernels as efficient as any of the other search techniques available in AutoTVM; however, it does it faster. Droplet Search also compares well with TVM’s Ansor, although using a much smaller pool of optimizations.

The combination of Droplet Search with TVM’s Ansor algorithm highlighted the potential of hybrid optimization strategies. While Ansor uses a broader set of code transformations, often resulting in optimized kernels, the addition of Droplet Search allowed for more focused exploitation within localized regions of the search space. This approach enabled the production of kernels that were competitive with those generated by Ansor alone. In some experimental settings, Droplet Search converged to high-performing solutions more quickly than Ansor, while using fewer transformations. This hybrid approach indicates that combining broad exploration techniques with localized exploitation strategies can lead to more efficient solutions for optimizing neural network models across a range of architectures.

Convexity opens up new avenues to the design and implementation of code optimizations. Testimony to this statement is the new fusion algorithm proposed in the context of this work. Chapter 6 introduced heuristic-based techniques, including Greedy Clustering and MST-based fusion, aimed at minimizing the number of edges in computational graphs and optimizing memory usage. Fusion techniques were effective in reducing binary size and improving performance on models like MnasNet1 and MobileNet v2, with speedups observed in some cases. However, the benefits of fusion were more limited for certain models, such as DenseNet, indicating that the effectiveness of fusion depends on the specific architecture and computational graph structure. While the MST-based fusion algorithm produced significant improvements in performance, it also highlighted some of the limitations of fusion in resource-constrained environments, particularly on DSPs. Increased

cache pressure and memory constraints were observed in certain settings, which reduced the overall benefits of fusion. Furthermore, the exhaustive search method, represented by the BKT algorithm, struggled with larger models, indicating a need for more scalable fusion approaches.

In conclusion, this thesis presents advancements in both kernel scheduling and operator fusion. The Droplet Search methodology, especially when combined with Ansor, offers a path for future research in neural network optimization, providing both efficiency and scalability. Additionally, the exploration of fusion techniques, while effective in certain contexts, highlights areas where further refinement is needed, particularly in balancing memory usage and computational efficiency. As machine learning models grow in complexity, the insights from this work can help guide the development of more effective compilation frameworks, ensuring that deep learning workloads can be deployed efficiently across diverse hardware platforms.

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, page 265–283, New York, USA, 2016. USENIX Association. ISBN 9781931971331.
- [2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O’Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. In CGO, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: 10.1109/CGO.2006.37.
- [3] Willow Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. Autoscheduling for sparse tensor algebra with an asymptotic cost model. In PLDI, page 269–285, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523442. URL <https://doi.org/10.1145/3519939.3523442>.
- [4] Md Zahangir Alom, Tarek M Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C Van Esesn, Abdul A S Awwal, and Vijayan K Asari. The history began from alexnet: A comprehensive survey on deep learning approaches, 2018.
- [5] Andrei Rimsa Álvares, José Nelson Amaral, and Fernando Magno Quintão Pereira. Instruction visibility in SPEC CPU2017. J. Comput. Lang., 66:101062, 2021. doi: 10.1016/j.cola.2021.101062. URL <https://doi.org/10.1016/j.cola.2021.101062>.
- [6] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In The ACM SIGPLAN ‘93 Conference on Programming Language Design and Implementation, pages 126–138, Albuquerque, N.M., Jun 1993. ACM Press.
- [7] AMD. AMD Ryzen™ 7 3700X. <https://www.amd.com/en/product/8446>, 2019. [Online; accessed 18-Jan-2024].
- [8] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison,

- Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, CK Luk, Bert Maher, Yunjie Pan, Christian Puhersch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Michael Suo, Phil Tillet, Eikan Wang, Xiaodong Wang, William Wen, Shunting Zhang, Xu Zhao, Keren Zhou, Richard Zou, Ajit Mathews, Gregory Chanan, Peng Wu, and Soumith Chintala. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In ASPLOS, pages 623–630, New York, USA, 2024. ACM.
- [9] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, pages 929–947, CA, La Jolla, USA, 2024. ACM.
- [10] Volodymyr Arbatov, Pedro Vaz Artigas, and Xianmin Chen. Xtensa neural network compiler: Optimizer. https://r6.ieee.org/scv-cis/wp-content/uploads/sites/6/2019/03/Cadence_Tensilica_XNNC_Optimizer_3_4_2019_Final.pdf, 2019. IEEE ML Compiler Workshop.
- [11] Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A survey on compiler autotuning using machine learning. Comput. Surv., 51(5):96:1–96:42, 2018. doi: 10.1145/3197978.
- [12] Amir Hossein Ashouri, Giovanni Mariani, Gianluca Palermo, Eunjung Park, John Cavazos, and Cristina Silvano. COBAYN: Compiler autotuning framework using bayesian networks. TACO, 13(2):21:1–21:25, 2016. ISSN 1544-3566. doi: 10.1145/2928270.
- [13] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, and Saman Amarasinghe. A deep learning based cost model for automatic code optimization, 2021. URL <https://arxiv.org/abs/2104.04955>.
- [14] D. Bertsekas. Convex Optimization Theory. Athena Scientific optimization and computation series. Athena Scientific, Nashua, NH, USA, 2009. ISBN 9781886529311. URL <https://books.google.com.br/books?id=0H1iQwAACAAJ>.
- [15] Jun Bi, Qi Guo, Xiaqing Li, Yongwei Zhao, Yuanbo Wen, Yuxuan Guo, Enshuai Zhou, Xing Hu, Zidong Du, Ling Li, Huaping Chen, and Tianshi Chen. Heron: Automatically constrained high-performance library generation for deep learning

- accelerators. In ASPLOS, page 314–328, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399180. doi: 10.1145/3582016.3582061. URL <https://doi.org/10.1145/3582016.3582061>.
- [16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In PLDI, PLDI '08, page 101–113, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375595. URL <https://doi.org/10.1145/1375581.1375595>.
- [17] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In CC, page 201–211, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371209. doi: 10.1145/3377555.3377894.
- [18] Cadence. Tensilica xnnv2 full course. https://www.cadence.com/en_US/home/training/all-courses/86277.html, 2024.
- [19] Cadence. Xtensa neural network compiler user guide [white paper], 2024.
- [20] Xuyi Cai, Ying Wang, and Lei Zhang. Optimus: towards optimal layer-fusion on deep learning processors, 2021.
- [21] Xuyi Cai, Ying Wang, and Lei Zhang. Optimus: An operator fusion framework for deep neural networks. ACM Transactions on Embedded Computing Systems, 22(1): 1–26, 2022.
- [22] Michael Canesche. [RFC] adding an exploitation phase to metaschedule to improve scheduling. <https://discuss.tvm.apache.org/t/rfc-metaschedule-adding-an-exploitation-phase-to-metaschedule-to-improve-scheduling/17365>, 2024. Experimental results are listed on a report that is publicly available at https://homepages.dcc.ufmg.br/~michaelcanesche/paper/bennu_meta_version.pdf.
- [23] Michael Canesche, Ricardo Ferreira, José Augusto Nacif, and Fernando Magno Quintão Pereira. A polynomial time exact solution to the bit-aware register binding problem. In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, pages 29–40, 2022. doi: 10.1145/3497776.3517773.
- [24] Michael Canesche, Roland Leissa, and Fernando Magno Quintão Pereira. Preparing reproducible scientific artifacts using docker. arXiv preprint arXiv:2308.14122, 2023. doi: 2308.14122.

- [25] Michael Canesche, Vanderson M. Rosario, Edson Borin, and Fernando Magno Quintão Pereira. The droplet search algorithm for kernel scheduling. *ACM Trans. Archit. Code Optim.*, feb 2024. ISSN 1544-3566. doi: 10.1145/3650109. URL <https://doi.org/10.1145/3650109>. Just Accepted.
- [26] Michael Canesche, Gaurav Verma, and Fernando Magno Quintao Pereira. Explore as a storm, exploit as a raindrop: On the benefit of fine-tuning kernel schedulers with coordinate descent. *arXiv preprint arXiv:2406.20037*, 2024.
- [27] John Cavazos and Michael F. P. O’Boyle. Method-specific dynamic compilation using logistic regression. In *OOPSLA*, page 229–240, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933484. doi: 10.1145/1167473.1167492.
- [28] John Cavazos, J. Eliot B. Moss, and Michael F. P. O’Boyle. Hybrid optimizations: Which optimization algorithm to use? In *Proceedings of the 15th International Conference on Compiler Construction, CC’06*, page 124–138, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 354033050X. doi: 10.1007/11688839_12. URL https://doi.org/10.1007/11688839_12.
- [29] John Cavazos, Grigori Fursin, Felix Agakov, Edwin Bonilla, Michael F. P. O’Boyle, and Olivier Temam. Rapidly selecting good compiler optimizations using performance counters. In *CGO*, page 185–197, USA, 2007. IEEE Computer Society. ISBN 0769527647. doi: 10.1109/CGO.2007.32. URL <https://doi.org/10.1109/CGO.2007.32>.
- [30] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *ICS*, page 492–499, New York, NY, USA, 1999. ACM. ISBN 158113164X. doi: 10.1145/305138.305245. URL <https://doi.org/10.1145/305138.305245>.
- [31] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342322. doi: 10.1145/2939672.2939785. URL <https://doi.org/10.1145/2939672.2939785>.
- [32] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015. URL <http://arxiv.org/abs/1512.01274>.
- [33] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and

- Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In OSDI, pages 579–594, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-47-8.
- [34] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm: An automated end-to-end optimizing compiler for deep learning. In OSDI, OSDI’18, page 579–594, USA, 2018. USENIX Association. ISBN 9781931971478.
- [35] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, 2018.
- [36] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In PLDI, page 279–290, New York, NY, USA, 1995. ACM. ISBN 0897916972. doi: 10.1145/207110.207162. URL <https://doi.org/10.1145/207110.207162>.
- [37] Gianpietro Consolaro, Zhen Zhang, Harenome Razanajato, Nelson Lossing, Nassim Tchoulak, Adilla Susungi, Artur Cesar Araujo Alves, Renwei Zhang, Denis Barthou, Corinne Ancourt, et al. Polytops: Reconfigurable and flexible polyhedral scheduler. In 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 28–40. IEEE, 2024.
- [38] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, Michael F P O’Boyle, and Hugh Leather. ProGraML: A graph-based program representation for data flow analysis and compiler optimizations. In ICML, volume 139, pages 2244–2253, Baltimore, Maryland, USA, 18–24 Jul 2021. PMLR.
- [39] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. CompilerGym: Robust, performant compiler optimization environments for AI research. CoRR, abs/2109.08267, 2021.
- [40] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning, 2018.

- [41] Anderson Faustino da Silva, Bruno Conde Kind, José Wesley de Souza Magalhães, Jerônimo Nunes Rocha, Breno Campos Ferreira Guimarães, and Fernando Magno Quintão Pereira. AnghaBench: A suite with one million compilable C benchmarks for code-size reduction. In CGO, pages 378–390, Los Alamitos, CA, USA, 2021. IEEE. doi: 10.1109/CGO51591.2021.9370322.
- [42] Junio Cezar Ribeiro Da Silva, Lorena Leão, Vinicius Petrucci, Abdoulaye Gamatié, and Fernando Magno Quintão Pereira. Mapping computations in heterogeneous multicore systems with statistical regression on program inputs. ACM Trans. Embed. Comput. Syst., 20(6), oct 2021. ISSN 1539-9087. doi: 10.1145/3478288. URL <https://doi.org/10.1145/3478288>.
- [43] Thaís Damásio, Michael Canesche, Vinícius Pacheco, Marcus Botacin, Anderson Faustino da Silva, and Fernando M Quintão Pereira. A game-based framework to compare program classifiers and evaders. In Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization, pages 108–121, 2023. doi: 10.1145/3579990.3580012.
- [44] Alain Darté. On the complexity of loop fusion. In PACT, page 149, USA, 1999. IEEE Computer Society. ISBN 0769504256.
- [45] Pooya Davoodi, Chul Gwon, Guangda Lai, and Trevor Morris. Tensorrt inference with tensorflow, 2019.
- [46] Pulin Desai. Vision and ai dsps for ultra-high-end and always-on applications. In Embedded Vision Summit, San Jose, US, 2021. Cadence Design Systems. URL https://www.edge-ai-vision.com/wp-content/uploads/2021/05/ET067_Desai_Cadence.pdf.
- [47] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, NAACL-HLT, pages 4171–4186, New York, US, 2019. Association for Computational Linguistics. doi: 10.18653/v1/n19-1423. URL <https://doi.org/10.18653/v1/n19-1423>.
- [48] Sunidhi Dhandhanía, Akshay Deodhar, Konstantin Pogorelov, Swarnendu Biswas, and Johannes Langguth. Explaining the performance of supervised and semi-supervised methods for automated sparse matrix format selection. In ICPP, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384414. doi: 10.1145/3458744.3474049. URL <https://doi.org/10.1145/3458744.3474049>.

- [49] Yaoyao Ding, Cody Hao Yu, Bojian Zheng, Yizhi Liu, Yida Wang, and Gennady Pekhimenko. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In ASPLOS, page 370–384, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575702. URL <https://doi.org/10.1145/3575693.3575702>.
- [50] Carl Engelman. Mathlab: a program for on-line machine assistance in symbolic computations. In AFIPS, page 117–126, New York, NY, USA, 1965. Association for Computing Machinery. ISBN 9781450378864. doi: 10.1145/1464013.1464034. URL <https://doi.org/10.1145/1464013.1464034>.
- [51] Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. Code generation for in-place stencils. In CGO, page 2–13, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701016. doi: 10.1145/3579990.3580006. URL <https://doi.org/10.1145/3579990.3580006>.
- [52] Karim Essegir. Improving data locality for caches. Master’s thesis, Rice University, 1993.
- [53] Paul Feautrier. Dataflow analysis of array and scalar references. Int. J. Parallel Program., 20(1):23–53, 1991. doi: 10.1007/BF01407931. URL <https://doi.org/10.1007/BF01407931>.
- [54] Paul Feautrier. Automatic parallelization in the polytope model. In The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications, page 79–103, Berlin, Heidelberg, 1996. Springer-Verlag. ISBN 3540617361.
- [55] J. A. B. Fortes and D. Moldovan. Data broadcasting in linearly scheduled array processors. In Proceedings, 11th Annual Symposium on Computer Architecture, pages 224–231, 1984.
- [56] Justin Fu, Aviral Kumar, Matthew Soh, and Sergey Levine. Diagnosing bottlenecks in deep q-learning algorithms. In International Conference on Machine Learning, pages 2021–2030. PMLR, 2019.
- [57] Davide Gadioli, Emanuele Vitali, Gianluca Palermo, and Cristina Silvano. Margot: a dynamic autotuning framework for self-aware approximate computing. IEEE transactions on computers, 68(5):713–728, 2018.
- [58] Michael Gilbert. LoopTree: Enabling Systematic and Flexible Exploration of Fused-layer Dataflow Accelerators. PhD thesis, Massachusetts Institute of Technology, 2023.

- [59] Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke. Towards a high-performance ai compiler with upstream mlir, 2024. URL <https://arxiv.org/abs/2404.15204>.
- [60] Google. Xla overview. <https://openxla.org/xla>, 2024.
- [61] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Process. Lett.*, 22(4), 2012. doi: 10.1142/S0129626412500107. URL <https://doi.org/10.1142/S0129626412500107>.
- [62] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, et al. Recent advances in convolutional neural networks. *Pattern recognition*, 77:354–377, 2018.
- [63] James W Hanson, Jane Shearin Caviness, and Camilla Joseph. Analytic differentiation by computer. *Communications of the ACM*, 5(6):349–355, 1962.
- [64] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL <http://arxiv.org/abs/1512.03385>.
- [65] 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. *CoRR*, abs/1704.04861, 2017. URL <http://arxiv.org/abs/1704.04861>.
- [66] Itay Hubara, Yury Nahshan, Yair Hanani, Ron Banner, and Daniel Soudry. Accurate post training quantization with small calibration sets. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 4466–4475, Maastricht, NL, 18–24 Jul 2021. PMLR.
- [67] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.
- [68] Charles Jin, Phitchaya Mangpo Phothilimthana, and Sudip Roy. Neural architecture search using property guided synthesis. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022. doi: 10.1145/3563329. URL <https://doi.org/10.1145/3563329>.
- [69] Tian Jin, Gheorghe-Teodor Bercea, Tung D. Le, Tong Chen, Gong Su, Haruki Imai, Yasushi Negishi, Anh Leu, Kevin O’Brien, Kiyokuni Kawachiya, and Alexandre E. Eichenberger. Compiling onnx neural network models using mlir, 2020.

- [70] Samuel J. Kaufman, Phitchaya Mangpo Phothilimthana, Yanqi Zhou, Charith Mendis, Sudip Roy, Amit Sabne, and Mike Burrows. A learned performance model for tensor processing units. In Alex Smola, Alex Dimakis, and Ion Stoica, editors, *MLSys*, Indio, CA, USA, 2021. mlsys.org.
- [71] Cecilia Conde Kind, Michael Canesche, and Fernando Magno Quintao Pereira. Jotai: a methodology for the generation of executable c benchmarks. Technical report, Technical Report 02-2022. Universidade Federal de Minas Gerais, 2022.
- [72] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. *Optimization by Simulated Annealing*, page 551–567. MIT Press, Cambridge, MA, USA, 1988. ISBN 0262010976.
- [73] Eliska Kloberdanz and Wei Le. Mixquant: Mixed precision quantization with a bit-width optimization search, 2023.
- [74] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *POPL*, page 207–218, New York, NY, USA, 1981. Association for Computing Machinery. ISBN 089791029X. doi: 10.1145/567532.567555. URL <https://doi.org/10.1145/567532.567555>.
- [75] M. S. Lam. *A Systolic Array Optimizing Computer*. Kluwer Academic (Springer), 1989. doi: 10.1007/978-1-4613-1705-0. ISBN:-13: 978-14612-8961-6.
- [76] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS IV*, page 63–74, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897913809. doi: 10.1145/106972.106981. URL <https://doi.org/10.1145/106972.106981>.
- [77] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *CGO*, page 2–14, Virtual Event, Republic of Korea, 2021. IEEE Press. ISBN 9781728186139. doi: 10.1109/CGO51591.2021.9370308. URL <https://doi.org/10.1109/CGO51591.2021.9370308>.
- [78] Mikhail Lebedev and Pavel Belecky. A survey of open-source tools for fpga-based inference of artificial neural networks. In *IVMEM*, pages 50–56, New York, US, 2021. IEEE. doi: 10.1109/IVMEM53963.2021.00015.
- [79] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521 (7553):436–444, 2015.
- [80] David A. Levine. Algorithm 344: Student’s t-distribution [s14]. *Commun. ACM*, 12(1):37–38, jan 1969. ISSN 0001-0782. doi: 10.1145/362835.362841. URL <https://doi.org/10.1145/362835.362841>.

- [81] Chendi Li, Yufan Xu, Sina Mahdipour Saravani, and Ponnuswamy Sadayappan. Accelerated auto-tuning of gpu kernels for tensor computations. In ICS, page 549–561, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706103. doi: 10.1145/3650200.3656626. URL <https://doi.org/10.1145/3650200.3656626>.
- [82] Hang Li. Deep learning for natural language processing: advantages and challenges. National Science Review, 5(1):24–26, 2018.
- [83] 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. IEEE Transactions on Parallel and Distributed Systems, 32(3):708–727, 2020.
- [84] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. Analytical characterization and design space exploration for optimization of cnns. In ASPLOS, page 928–942, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383172. doi: 10.1145/3445814.3446759. URL <https://doi.org/10.1145/3445814.3446759>.
- [85] Meilin Liu, Qingfeng Zhuge, Zili Shao, and Edwin H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In CASES, page 190–201, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138903. doi: 10.1145/1023833.1023860. URL <https://doi.org/10.1145/1023833.1023860>.
- [86] Sébastien Marcel and Yann Rodriguez. Torchvision the machine-vision package of torch. In Proceedings of the 18th ACM international conference on Multimedia, pages 1485–1488, Firenze, Italy, 2010. ACM.
- [87] Linyan Mei, Koen Goetschalckx, Arne Symons, and Marian Verhelst. Defines: Enabling fast exploration of the depth-first scheduling space for dnn accelerators through analytical modeling. In HPCA, pages 570–583, New York, US, 2023. IEEE.
- [88] Jintao Meng, Chen Zhuang, Peng Chen, Mohamed Wahib, Bertil Schmidt, Xiao Wang, Haidong Lan, Dou Wu, Minwen Deng, Yanjie Wei, and Shengzhong Feng. Automatic generation of high-performance convolution kernels on ARM cpus for deep learning. IEEE Trans. Parallel Distributed Syst., 33(11):2885–2899, 2022. doi: 10.1109/TPDS.2022.3146257. URL <https://doi.org/10.1109/TPDS.2022.3146257>.
- [89] Massinissa Merouani, Khaled Afif Boudaoud, Iheb Nassim Aouadj, Nassim Tchoulak, Islem Kara Bernou, Hamza Benyamina, Fatima Benbouzid-Si Tayeb, Karima Be-

- natchba, Hugh Leather, and Riyadh Baghdadi. Looper: A learned automatic code optimizer for polyhedral compilers, 2024. URL <https://arxiv.org/abs/2403.11522>.
- [90] Microsoft. Onnx runtime. <https://onnxruntime.ai/>, 2024.
- [91] Nicholas Mitchell, Larry Carter, Jeanne Ferrante, and Karin Högstedt. Quantifying the multi-level nature of tiling interactions. In LCPC, page 1–15, Berlin, Heidelberg, 1997. Springer-Verlag. ISBN 3540644725.
- [92] Naums Mogers, Lu Li, Valentin Radu, and Christophe Dubach. Mapping parallelism in a functional ir through constraint satisfaction: A case study on convolution for mobile gpus. In CC, page 218–230, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391832. doi: 10.1145/3497776.3517777. URL <https://doi.org/10.1145/3497776.3517777>.
- [93] Sungdo Moon and Rafael Saavedra. Hyperblocking: A data reorganization method to eliminate cache conflicts in tiled loop nests. Technical Report USC-CS-98-671, University of Southern California, 01 1998.
- [94] Orlando Moreira, Merten Popp, and Christian Schulz. Graph partitioning with acyclicity constraints, 2017.
- [95] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In NIPS, page 929–935, Cambridge, MA, USA, 1997. MIT Press. doi: 10.5555/3008904.3009034.
- [96] Girish Mururu, Sharjeel Khan, Bodhisatwa Chatterjee, Chao Chen, Chris Porter, Ada Gavrilovska, and Santosh Pande. Beacons: An end-to-end compiler framework for predicting and utilizing dynamic loop characteristics. Proc. ACM Program. Lang., 7(OOPSLA2), oct 2023. doi: 10.1145/3622803. URL <https://doi.org/10.1145/3622803>.
- [97] Apache MxNet. Apahe mxnet: A flexible and efficient library for deep learning. <https://mxnet.apache.org/>, 2024.
- [98] Tan Nguyen, Samuel Williams, Marco Siracusa, Colin MacLean, Douglas Doerfler, and Nicholas J Wright. The performance and energy efficiency potential of fpgas in scientific computing. In 2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 8–19. IEEE, 2020.
- [99] Wei Niu, Gagan Agrawal, and Bin Ren. Sod2: Statically optimizing dynamic deep neural network, 2024.

- [100] Nvidia. Nvidia A100 Tensor Core GPU. <https://www.nvidia.com/en-in/data-center/a100/>, 2020. [Online; accessed 18-Jan-2024].
- [101] Nvidia. Nvidia RTX3080 Tensor Core GPU. <https://www.nvidia.com/en-in/geforce/graphics-cards/30-series/>, 2020. [Online; accessed 18-Jan-2024].
- [102] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. Automated derivation of parametric data movement lower bounds for affine programs. In *PLDI*, page 808–822, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3385989. URL <https://doi.org/10.1145/3385412.3385989>.
- [103] Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. Ioopt: Automatic derivation of i/o complexity bounds for affine programs. In *PLDI*, page 1187–1202, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383912. doi: 10.1145/3453483.3454103. URL <https://doi.org/10.1145/3453483.3454103>.
- [104] Ookami. ACCESS Research Provider, A64FX Cluster. <https://www.stonybrook.edu/ookami/>, 2022. [Online; accessed 18-Jan-2024].
- [105] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach 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 *NIPS*, Red Hook, NY, USA, 2019. Curran Associates Inc.
- [106] Philip Pfaffe, Tobias Grosser, and Martin Tillmann. Efficient hierarchical online-autotuning: a case study on polyhedral accelerator mapping. In *ICS*, ICS '19, page 354–366, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360791. doi: 10.1145/3330345.3330377. URL <https://doi.org/10.1145/3330345.3330377>.
- [107] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Reza Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake A. Hechtman, Bjarke Røne, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. A flexible approach to autotuning multi-pass machine learning compilers. In Jaejin Lee and Albert Cohen, editors, *PACT*, pages 1–16, New York, 2021. IEEE. doi: 10.1109/PACT52795.2021.00008. URL <https://doi.org/10.1109/PACT52795.2021.00008>.

- [108] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H Jones. Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In 2019 IEEE international conference on embedded software and systems (ICESS), pages 1–8. IEEE, 2019.
- [109] Murad Qasaimeh, Joseph Zambreno, Phillip H Jones, Kristof Denolf, Jack Lo, and Kees Vissers. Analyzing the energy-efficiency of vision kernels on embedded cpu, gpu and fpga platforms. In 2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 336–336. IEEE, 2019.
- [110] Bo Qiao, Oliver Reiche, Frank Hannig, and Jürgen Teich. From loop fusion to kernel fusion: A domain-specific approach to locality optimization. In 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pages 242–253, Washington DC USA, 2019. IEEE, IEEE Press.
- [111] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. Journal of VLSI Signal Processing, 1(2):95–113, 1989.
- [112] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. SIGPLAN Not., 48(6):519–530, jun 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL <https://doi.org/10.1145/2499370.2462176>.
- [113] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. Acm Sigplan Notices, 48(6):519–530, 2013.
- [114] S. V. Rajopadhye. Synthesis, Optimization and Verification of Systolic Architectures. PhD thesis, University of Utah, Salt Lake City, Utah 84112, December 1986.
- [115] S. V. Rajopadhye, S. Purushothaman, and R. M. Fujimoto. On synthesizing systolic arrays from recurrence equations with linear dependencies. In Proceedings, Sixth Conference on Foundations of Software Technology and Theoretical Computer Science, pages 488–503, New Delhi, India, December 1986. Springer Verlag, LNCS 241.
- [116] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In SC, Austin, Texas, 2008. IEEE Press. ISBN 9781424428359.
- [117] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators.

- In 2019 IEEE high performance extreme computing conference (HPEC), pages 1–9, Waltham, MA, USA, 2019. IEEE, IEEE.
- [118] Peter Richtárik and Martin Takác. Parallel coordinate descent methods for big data optimization. CoRR, abs/1212.0873, 2012. URL <http://arxiv.org/abs/1212.0873>.
- [119] Fabian Ritter and Sebastian Hack. Explainable port mapping inference with sparse performance counters for AMD’s zen architectures, 2024.
- [120] Gabriel Rivera and Chau-Wen Tseng. A comparison of compiler tiling algorithms. In CC, page 168–182, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3540657177.
- [121] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks, 2018.
- [122] Jaehun Ryu and Hyojin Sung. Metatune: Meta-learning based cost model for fast and efficient auto-tuning frameworks. arXiv preprint arXiv:2102.04199, 2021.
- [123] Christian Sarofeen, Piotr Bialecki, Jie Jiang, Kevin Stephano, Masaki Kozuki, Neal Vaidya, and Stas Bekman. Introducing nvfuser, a deep learning compiler for pytorch. <https://pytorch.org/blog/introducing-nvfuser-a-deep-learning-compiler-for-pytorch/>, 2022. [Online; accessed 15-Mar-2024].
- [124] Abhronil Sengupta, Yuting Ye, Robert Wang, Chiao Liu, and Kaushik Roy. Going deeper in spiking neural networks: VGG and residual architectures. CoRR, abs/1802.02627, 2018. URL <http://arxiv.org/abs/1802.02627>.
- [125] Junru Shao. [RFC] meta schedule (autotensorir). <https://discuss.tvm.apache.org/t/rfc-meta-schedule-autotensorir/10120>, 2021.
- [126] Anderson Faustino da Silva, Bernardo N. B. de Lima, and Fernando Magno Quintão Pereira. Exploring the space of optimization sequences for code-size reduction: Insights and tools. In Compiler Construction, page 47–58, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383257. doi: 10.1145/3446804.3446849.
- [127] Douglas Simon, John Cavazos, Christian Wimmer, and Sameer Kulkarni. Automatic construction of inlining heuristics using machine learning. In CGO, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society. doi: 10.1109/CGO.2013.6495004.

- [128] Pramod Singh, Avinash Manure, Pramod Singh, and Avinash Manure. Introduction to tensorflow 2.0, 2020.
- [129] Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. ACM Transactions on Programming Languages and Systems, 45(2):1–43, 2023. doi: 10.1145/3594736.
- [130] Huan Song, Jayaraman J Thiagarajan, Prasanna Sattigeri, and Andreas Spanias. Optimizing kernel machines using deep learning. IEEE transactions on neural networks and learning systems, 29(11):5528–5540, 2018.
- [131] Tyler Sorensen, Sreepathi Pai, and Alastair F. Donaldson. One size doesn’t fit all: Quantifying performance portability of graph applications on gpus. In IISWC, pages 155–166, New York, US, 2019. IEEE. doi: 10.1109/IISWC47752.2019.9042139. URL <https://doi.org/10.1109/IISWC47752.2019.9042139>.
- [132] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Matrix multiplication beyond auto-tuning: Rewrite-based gpu code generation. In CASES, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450344821. doi: 10.1145/2968455.2968521. URL <https://doi.org/10.1145/2968455.2968521>.
- [133] Rafael Sumitani, Lucas Silva, Frederico Campos, and Fernando Pereira. A class of programs that admit exact complexity analysis via newton’s polynomial interpolation. In SBLP, page 50–55, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400716287. doi: 10.1145/3624309.3624311. URL <https://doi.org/10.1145/3624309.3624311>.
- [134] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagan-deep Goyal, and Ramakrishna Upadrasta. Polydl: Polyhedral optimizations for creation of high-performance dl primitives. Transactions on Architecture and Code Optimization, 18(1), jan 2021. ISSN 1544-3566. doi: 10.1145/3433103. URL <https://doi.org/10.1145/3433103>.
- [135] John Thomson, Michael O’Boyle, Grigori Fursin, and Björn Franke. Reducing training time in a one-shot machine learning-based compiler. In Guang R. Gao, Lori L. Pollock, John Cavazos, and Xiaoming Li, editors, Languages and Compilers for Parallel Computing, pages 399–407, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-13374-9.
- [136] Nicolas Tollenaere, Guillaume Iooss, Stéphane Pouget, Hugo Brunie, Christophe Guillon, Albert Cohen, P. Sadayappan, and Fabrice Rastello. Autotuning convolutions is easier than you think. ACM Trans. Archit. Code Optim., 20(2), mar 2023. ISSN 1544-3566. doi: 10.1145/3570641. URL <https://doi.org/10.1145/3570641>.

- [137] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In GROW, Pisa, Italy, January 2010. HAL. URL <https://inria.hal.science/inria-00551516>.
- [138] Mohamed Wahib and Naoya Maruyama. Scalable kernel fusion for memory-bound gpu applications. In SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 191–202, New Orleans, LA, USA, 2014. IEEE, IEEE.
- [139] Xiao Wang, Amit Sabne, Sherman Kisner, Anand Raghunathan, Charles Bouman, and Samuel Midkiff. High performance model based image reconstruction. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '16, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450340922. doi: 10.1145/2851141.2851163. URL <https://doi.org/10.1145/2851141.2851163>.
- [140] Thomas J. Watson, F. E. Allen, and J. Cocke. A catalogue of optimizing transformations, 1971.
- [141] Robert Edwin Wengert. A simple automatic derivative evaluation program. Communications of the ACM, 7(8):463–464, 1964.
- [142] Frank Wilcoxon. Individual Comparisons by Ranking Methods, pages 196–202. Springer New York, New York, NY, 1992. ISBN 978-1-4612-4380-9. doi: 10.1007/978-1-4612-4380-9_16. URL https://doi.org/10.1007/978-1-4612-4380-9_16.
- [143] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation, PLDI '91, page 30–44, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897914287. doi: 10.1145/113445.113449. URL <https://doi.org/10.1145/113445.113449>.
- [144] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. Int. J. Parallel Program., 26(4):479–503, aug 1998. ISSN 0885-7458.
- [145] M. J. Wolfe. Iteration space tiling for memory hierarchies. Parallel Processing for Scientific Computing (SIAM), pages 357–361, 1987.
- [146] Michael Joseph Wolfe, Carter Shanklin, and Leda Ortega. High Performance Compilers for Parallel Computing. Addison-Wesley Longman Publishing Co., Inc., USA, 1995. ISBN 0805327304.

- [147] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. Waco: Learning workload-aware co-optimization of the format and schedule of a sparse tensor program. In ASPLOS, page 920–934, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450399166. doi: 10.1145/3575693.3575742. URL <https://doi.org/10.1145/3575693.3575742>.
- [148] Stephen J. Wright. Coordinate descent algorithms. Math. Program., 151(1):3–34, jun 2015. ISSN 0025-5610. doi: 10.1007/s10107-015-0892-3. URL <https://doi.org/10.1007/s10107-015-0892-3>.
- [149] Jingling Xue, Minyi Guo, and Daming Wei. Improving the parallelism of iterative methods by aggressive loop fusion. J. Supercomput., 43(2):147–164, feb 2008. ISSN 0920-8542. doi: 10.1007/s11227-007-0124-6. URL <https://doi.org/10.1007/s11227-007-0124-6>.
- [150] Minyong Yoon and Jungwook Choi. Architecture-aware optimization of layer fusion for latency-optimal cnn inference. In AICAS, pages 1–4, New York, US, 2023. IEEE.
- [151] Tom Young, Devamanyu Hazarika, Soujanya Poria, and Erik Cambria. Recent trends in deep learning based natural language processing. ieee Computational intelligenCe magazine, 13(3):55–75, 2018.
- [152] W. Zangwill. Nonlinear Programming, A Unified Approach. Prentice Hall, USA, 1st edition, 1969. ISBN 0136235794.
- [153] Xiaoyang Zhang, Junmin Xiao, and Guangming Tan. I/O lower bounds for auto-tuning of convolutions in cnns. In PPoPP, page 247–261, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450382946. doi: 10.1145/3437801.3441609. URL <https://doi.org/10.1145/3437801.3441609>.
- [154] Jie Zhao, Jinchun Xu, Peng Di, Wang Nie, Jiahui Hu, Yanzhi Yi, Sijia Yang, Zhen Geng, Renwei Zhang, Bojie Li, et al. Modeling the interplay between loop tiling and fusion in optimizing compilers using affine relations. ACM Transactions on Computer Systems, 41(1-4):1–45, 2024.
- [155] Bojian Zheng, Ziheng Jiang, Cody Hao Yu, Haichen Shen, Joshua Fromm, Yizhi Liu, Yida Wang, Luis Ceze, Tianqi Chen, and Gennady Pekhimenko. Dietcode: Automatic optimization for dynamic tensor programs. In MLSys, page 848–863, New York, NY, USA, 2022. Marculescu and Chi. doi: 10.1145/3575693.3575702. URL <https://doi.org/10.1145/3575693.3575702>.
- [156] Jun Zheng, Suhail S. Saquib, Ken D. Sauer, and Charles A. Bouman. Parallelizable bayesian tomography algorithms with rapid, guaranteed convergence. IEEE Trans.

- Image Process., 9(10):1745–1759, 2000. doi: 10.1109/83.869186. URL <https://doi.org/10.1109/83.869186>.
- [157] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In OSDI, pages 863–879, New York, US, 2020. USENIX.
- [158] 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 Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1), 2021.
- [159] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, Fan Yang, Mao Yang, Lidong Zhou, Asaf Cidon, and Gennady Pekhimenko. ROLLER: fast and efficient tensor compilation for deep learning. In Marcos K. Aguilera and Hakim Weatherspoon, editors, OSDI, pages 233–248, New York, USA, 2022. USENIX Association. URL <https://www.usenix.org/conference/osdi22/presentation/zhu>.
- [160] Yi Zhu and Shawn Newsam. Densenet for dense flow. In 2017 IEEE international conference on image processing (ICIP), pages 790–794, New York, US, 2017. IEEE.
- [161] Mikhail Zolotukhin. Nnc walkthrough: how pytorch ops get fused. <https://dev-discuss.pytorch.org/t/nnc-walkthrough-how-pytorch-ops-get-fused/125>, 2021. [Online; accessed 15-Mar-2024].