AUTOMATED GENERATION OF DOMAIN SPECIFIC KERNELS

MEGHAN COWAN

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2021

Reading Committee:

Luis Ceze, Chair

Arvind Krishnamurthy

Zachary Tatlock

Program Authorized to Offer Degree:

Paul G. Allen School of Computer Science & Engineering

UNIVERSITY OF WASHINGTON

ABSTRACT

AUTOMATED GENERATION OF DOMAIN SPECIFIC KERNELS

Meghan Cowan

Chair of the Supervisory Committee:
Professor Luis Ceze
Paul G. Allen School of Computer Science & Engineering

Seamless gains in performance from technology scaling is coming to an end, but many applications rely on hardware and their compilation stacks to continue improving performance and efficiency. In order to keep up with application compute demands, emerging hardware is becoming more diverse, specialized, and complex. New hardware and accelerators expose programming models that have great potential performance, but are often more restrictive and difficult to program. Oftentimes, even traditional compilers struggle to generate efficient programs for these new programming models, leading to a proliferation of domain specific libraries comprised of hand-optimized kernels that are meticulously tuned to take advantage of the target hardware and avoid any bottlenecks.

This thesis argues that we can automatically generate and optimize programs by building domain specific tools that search for efficient code. I show how we can apply two search-based methods, program synthesis and autotuning, to take advantage of application specific optimizations such as efficiently utilizing vector parallelism present in many programming models.

In this dissertation, I demonstrate how we can optimize programs in machine learning and Homomorphic Encryption (HE) using search-based methods. First, I show how we can extend existing machine learning compilers to efficiently deploy unconventional neural networks, such as ultra quantized networks, by

augmenting core operators with synthesized vector code and intelligently tuned memory access behavior. Then, I present Porcupine, a synthesizng compiler for vectorized HE programs. Porcupine synthesizes vectorized code from a plaintext scalar reference program, and performs instruction selection and scheduling for HE's unique performance model. Together, these two systems show how we can use search to both tune and discover optimizations in programs.

To my family.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Luis Ceze for his support and encouragement. Luis is one of the most optimistic people I know, and he brings lots of energy and excitement to all the projects he works on. Luis gave me great freedom as a graduate student which allowed me to pursue projects involving program synthesis and Homomorphic Encryption. Even though I started working on topics he was unfamiliar with, he always supported and promoted my work. Additionally, I would like to thank my initial temporary advisor Mark Oskin. While we only worked together for a short time, Mark helped me transfer to the University of Washington and gave me a second chance at a PhD for which I am incredibly grateful.

At the University of Washington, I was incredibly fortunate to be a member of the Sampa and Sampl labs, where I had the pleasure of working with and learning from my amazing lab mates who have become great friends. Thank you to my fellow graduate students and postdocs, Armin Alaghi, James Bornholt, Tianqi Chen, Josh Fromm, Emily Furst, Vincent Lee, Liang Luo, Amrita Mazumdar, Thierry Moreau, Luis Vega, Eddie Yan, and anyone else I may have missed. Completing a PhD has many ups and downs, but I always enjoyed hanging out with everyone both inside and outside of the lab. Thank you for helping me scope out research ideas, write papers, give presentations, and so much more. Without your advice and encouragement, none of my research projects would have ever started and this dissertation would not have been possible.

During my PhD, I was fortunate to spend lots of time in industry between two internships at Microsoft Research and a temporary work position at Facebook Reality Labs Research. Thank you to my mentors at Microsoft Research, Matthai Philipose, Bodhi Priyantha, Luke Marshall, and Ishai Menache. Special thanks to Vincent Lee, who hosted me for a mostly remote year at Facebook Reality Labs Research, and my many collaborators and mentors at FRL: Armin Alaghi, Deeksha Dangwal, Hyo Jin Kim, Brandon Reagen, Caroline Trippel, and Amr Suleiman.

Finally, thank you to my father, Philip Cowan, and brother, Sean Cowan, who have been supporting my goals since I was a little kid and providing endless encouragement and support throughout my PhD.

CONTENTS

1 INTRODUCTION

Performance is a critical aspect of software programs. Faster programs allow applications to scale to larger problems, provide better quality of results, and use fewer resources. For over thirty years, two hardware scaling enabled trends that drove software performance: Moore's law and Dennard scaling. Moore's law, the doubling of transistor density roughly every eighteen months, and Dennard scaling, the constant power density as transistors shrink, enabled single-threaded performance to exponentially increase. Because new chip designs preserved the same instruction set architecture, software programs could run faster by simply waiting for the next hardware generation. However, physical barriers have ended both these trends [19, 52], and we can no longer rely on scaling to deliver performance.

1.1 TRENDS IN ARCHITECTURE AND COMPILERS

While hardware performance growth has slowed, new applications demand more performance, power, and efficiency from the hardware it runs on. Each year, machine learning and natural processing models grow in size requiring more resources to train and deploy [55, 68]. Because of privacy concerns, more complex computations shifts to low-power edge devices requiring more efficient usage of hardware. To continue improving power and performance, emerging hardware is becoming more specialized and complex.

The trend of specialization has already started; commercial system on chip (SoC) designs are comprised of an increasing number of diverse domain specific accelerators in addition to a general purpose CPU. Even general purpose processors are trending towards more specialization for a few important domains, specifically in the design of their single instruction multiple data (SIMD) units. Intel first introduced its SIMD extension to accelerate multimedia applications with special instructions that operated on multiple elements packed into a vector register. Since then an increasing number of vector extensions have been added to accelerate more workloads such as encryption, scientific computing, and machine learning that introduce more instructions operating over increasingly larger vector registers.

However, specialization comes at a cost: emerging hardware is significantly harder to program. Often times, domain specialized hardware sacrifice flexibility to reduce overhead present in general purpose hardware to accelerate compute heavy workloads. General-purpose compilers struggle to generate code that fully realizes the performance of complex hardware because they require new optimizations that are typically domain and problem specific. For

example, dense linear algebra and graph workloads benefit from a different set of optimizations and even small differences such as applying an operator to a small input requires different optimizations than applying the same operator to a large input. Building up mature compiler infrastructure with new hand crafted optimization passes is not a scalable solutions as it requires the effort of thousands of people to write millions of lines of code [48]. New methods that automatically discover and select optimizations are key for building compilers that are able to optimize for emerging hardware.

1.2 THESIS OVERVIEW

My thesis states that *search-based methods can automatically generate programs or compute kernels optimized for a particular application or hardware domain.* By sacrificing time to intelligently explore different program implementations, search-based methods can automatically discover and tune optimizations tailored to unique combinations of applications and backend targets. In this dissertation, I show how we can utilize two types of search-based optimization, program synthesis and autotuning, to automatically generate optimized code for unconventional machine learning models and Homomorphic Encryption programs.

1.2.1 *Ultra Quantized Kernels for Machine Learning*

Traditionally, machine learning frameworks [1, 98] decompose models, such as a neural network, into a series of kernels implemented by a high-performance library, such as cuDNN or MKL [9, 37, 42]. For commonly used neural networks, such as ResNets [69], this approach works well as most libraries provide efficient hand optimized implementations that accelerate these networks. However, this approach is inflexible and does not scale. It fails to support machine learning models that rely on unconventional or new primitives that are not accelerated by the library [12] leaving them to be optimized from scratch. This can lead programmers to falsely conclude that certain network designs are inherently slower than existing models because they are not optimized to the same extent as existing models.

Optimizing a neural network from scratch is a challenging task that requires knowledge of both machine learning and computer architecture, as well as time to write, debug, and benchmark various implementations. [135]. Furthermore, optimizations are not one size fit all. Each kernel needs a tailored implementations to fully utilize hardware resources. In particular scheduling optimizations require specializing for both kernel parameters, such as input shapes and datatypes, and the hardware it runs on.

Chapter 3 presents an automated approach to generating unconventional ultra quantized machine learning kernels. These kernels are crucial for implementing

models such as XNOR-Nets [107] and binarized neural networks [29, 56, 149] that operate on inputs and parameters quantized down to less than four bits. In theory, these networks should be fast; memory bottlenecks that plague full-precision kernels are gone and their computation uses efficient bitwise instructions. However, naive implementations can't compete with optimized full-precision implementations [45]. While a hand optimized ultra quantized precision convolution implementation can achieve excellent performance for a narrow range of convolutions [136], it does not scale to optimize the many convolution variations made up different combinations of shapes and precisions.

I present a two-pronged approach to optimize these kernels that uses autotuning to optimize memory access patterns and program synthesis to efficiently utilize SIMD hardware the network runs on. We autotune the memory access pattern by developing a template of the kernel that exposes scheduling parameters that affect performance through parallelism and data locality. Simulated annealing is used to intelligently sample the space of valid parameters for each unique kernel, selecting the one that empirically runs the fastest on the hardware.

Program synthesis is used to discover a vector intrinsic microkernel designed to take advantage of new intrinsics in CPU SIMD extensions. Often times autovectorizers can only handle embarrassingly parallel computation where the same computation is performed on each element in the vector. New vector intrinsics expose intra-vector instructions designed to accelerate machine learning that perform operations within a vector, such as pairwise additions, but are difficult for compilers to utilize. We developed a set of sketches that allowed us to explore different microkernel configurations that can use these intrinsics efficiently with the low-precision data. Using this approach, we were able to automatically generate a library of operators to deploy end-to-end quantized neural networks.

1.2.2 *Vectorized Kernels for Homomorphic Encryption*

Homomorphic Encrpytion (HE) is an emerging family of encryption schemes that enables computation over encrypted data or ciphertexts [25, 36, 39, 58]. Unlike traditional encryption schemes (e.g. RSA) which require ciphertexts be decrypted before computation, HE schemes provide a set of instructions that directly manipulate ciphertexts without decryption. This enables secure compute offload as both inputs, outputs, and intermediates of computation stay encrypted and secret, making it an promising technology for privacy preserving applications.

One of the key challenges facing HE applications is performance. HE ciphertexts and their operations incur orders of magnitude overheads in memory and run time over equivalent plaintext programs as data must be mapped to large mathematical structures like rings. Recently introduced HE schemes

such as BFV [53] and CKKS [36] are considered more efficient because they allow packing multiple integers into a single ciphertext that is manipulated by SIMD behaving instructions [123]. These schemes expose a low-level vector programming model[1] that is similar to the CPU SIMD programming model, and have many of the same vectorization challenges that plague CPU SIMD code generation [50]. Additionally, HE has its own unique set of constraints that include managing noise in ciphertexts for correctness and performance.

An efficient vector HE program must correctly map computation onto vector instructions to take advantage of vector parallelism while simultaneously considering the effect the instruction mix and ordering has on performance. Chapter 4, introduces Porcupine, a vectorizing compiler that addresses this problem using program synthesis. Porcupine take as input a plaintext scalar example program and synthesize a vectorized HE version by solving a constrained optimal synthesis problem. Porcupine searches a space of potential candidate programs for valid implementations that minimize a cost function that factors in instruction latency and noise accumulation.

Vectorization is difficult compilation problem as it requires global changes to move from a scalar to vector representation [85] and insertion of data shuffling instructions to align slots within a vector for certain computations. However, Porcupine does not rely on rules for recognizing and packing scalar computation into vector computation, which can be challenging to develop and safely apply. Instead a space of vector programs (including incorrect programs) is searched and the scalar reference implementation is used to guarantee correctness. Using Porcupine, we are able to synthesize HE kernels that are on average 11% faster than hand optimized vector kernels.

1.3 THESIS ORGANIZATION

The following chapters of this dissertation is organized as follows. Chapter 2 provides a survey on methods for generating optimizing programs and discusses related work in this field. In particular, I focus on search-based techniques, such as synthesis and autotuning, that search over a space of many possible implementations. Chapter 3 describes my work on automatically generating ultra quantized kernels for machine learning. In this chapter, I show that combining autotuning and synthesis to optimize vectorization and memory access behavior enables us to automatically generate a library of kernels to accelerate ultra quantized neural networks.

Switching gears, Chapters 4 presents Porcupine, a vectorizing compiler for HE. Porcupine uses synthesis to generate HE kernels from plaintext scalar kernels. By reasoning about HE's low-level vector programming and its constraints, Porcupine is able to generate vector kernels that are optimized for HE's unique

1 This vector parallelism is part of the HE scheme and is separate from hardware vector paralleism an HE implementation can exploit.

performance model. Finally, Chapter 5 concludes this dissertation with future directions in automatically generating optimized programs.

1.4 PREVIOUSLY PUBLISHED MATERIAL

This dissertation includes work published as conference papers:

- Chapter 3: Meghan Cowan et al. "Automatic generation of high-performance quantized machine learning kernels." In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020
- Chapter 4: Meghan Cowan et al. "Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption." In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2021

2

SURVEY ON OPTIMIZATION METHODS

Code optimization is a large and mature field, and optimizing compilers have existed since at least the early 1970s [4, 81, 95]. Compilers and their optimizations are critical to raising the abstraction level of programming languages so that programs are easier to write. Modern compilers such as GCC and LLVM [82] are built upon a plethora of hardware agnostic and hardware specific optimizations such as constant folding, dead code elimination, hoisting, and register allocation, that help automatically optimize code and allow programmers to focus on high-level system design and not low-level performance details.

An optimizing compiler's goal is to produce the best possible code or implementation for a particular program and hardware target pair. Even for small programs this is not feasible as there are potentially thousands of equivalent implementations, and finding the best implementation is computationally intractable. However, by combining multiple optimizations that target specific goals such as dead code elimination or autovectorization, compilers could greatly improve code performance.

This chapter provides a background survey on techniques for optimizing programs. I broadly organize techniques into two classes: traditional and search-based methods where traditional methods are those widely accepted and commercially utilized and search-based methods are emerging techniques that utilize some form of search during the optimization process. Traditional methods include rewrite rules that deterministically transform a program and are employed by almost all major compilers and hand optimization which involves a programmer bypassing some or all parts of the compiler to manually optimize code, and is commonly employed in high-performance computing. Search-based methods include polyhedral optimization, autotuning, and program synthesis which formulate compilation (or even generation) of programs as an intelligent search problem over a candidate space defined by a high-level abstraction of the program (e.g. a template or grammar defining building blocks). Each section begins with high-level overview of each of these techniques and discusses their strengths and weaknesses. For each technique, I also discuss relevant related work, focusing on the application domain (e.g. machine learning, SIMD processors, etc.) they are applied to and their optimization goal (e.g. autovectorization, scheduling, etc.).

2.1 TRADITIONAL OPTIMIZATION METHODS

This section describes what I refer to as traditional optimization methods, which are methods commonly used in commercial compilers and code bases. While rewrite rules are built into compilers to provide automatic optimization, hand optimization pushes all hardware and algorithmic complexities back to the programmer to handle.

2.1.1 *Rewrite Rules*

Traditional compiler optimizations often take the form of semantics preserving transformations or rewrite rules that are applied over a program's IR (intermediate representation). Typically, a compiler applies these optimizations in multiple passes that analyze the program's dataflow to gather information, then apply rewrite rules that gradually transform the original IR into an optimized IR. Each pass applies a particular rule by matching an IR pattern and replacing it cheaper fragment. Certain passes enable other passes making the sequence rules are applied critical. In general, which rules are applied and their ordering are determined by heuristics.

Rewrite optimizations are incredibly successful, and high-performance compilers include thousands of lines of code dedicated to them. One of the main advantages of rewriting is compilation time and scalability. A large number of passes can be performed over a large program's IR and compilation will still finish in a reasonable time.

Many successful domain specific compilers also rely on rewriting systems that operate over a high-level domain specific IR [106]. For example, machine learning compilers represent models as a graph of operators, and apply rewrite passes to improve their performance [1, 74, 98, 113]. Rewrite rules might fuse operators to reduce data movement or quantize operators to operate over lower precision integers (technically this is not a semantics preserving rewrite). EVA [49] a compiler for Homomorphic Encryption, builds up a graph of HE instructions and automates the placement of low-level HE instructions through a sequence of graph rewrites. By automating this process, EVA raises the level of abstraction level of HE programming and often provides better performance than manually inserting these instructions.

A rewrite system's efficiency relies on two key components: its set of passes or rewrite rules and the ordering rules are applied. The set of rewrite rules determines all possible transformations the compiler can make while the ordering determines which transformations are explored and ultimately used. Generating an expressive set of rules that exposes many optimizations is a challenging task. Usually, these rules are handcrafted and designed to address a specific optimization like constant propagation or dead code elimination. Given the set of rules, deciding which rules to apply and their sequence determines

the actual optimizations a program goes through. Typically, this is determined by heuristics that are also handcrafted. Developing a good set of heuristics is not a simple task and requires a significant amount of human intuition and investment. Heuristics are not guaranteed to find the best ordering of rules, and can miss out on many potential optimizations.

Furthermore, deciding whether a transformation is beneficial is not a simple task as different applications and hardware benefit from different optimizations. Compilers rely on hardware performance models for determining whether hardware dependent optimizations are beneficial. Developing an accurate performance model is challenging as modern hardware is complex and microarchitectural details that affect performance are kept secret. Because optimization passes can negatively affect performance for some programs, many optimizations are turned off to prevent accidentally hurting performance.

2.1.2 *Hand Optimization*

While hand optimization is not an automated technique, in practice it is heavily relied on. For critical programs and commonly used kernels, hand optimizing code down to the assembly level is critical for performance, and can lead to faster implementations than relying on the compiler to optimize naive code. Most hardware vendors ship handcrafted libraries to accelerate key applications such as machine learning and scientific programming. For example, Intel's MKL [42] and NVIDIA's cuDNN [37] are made up of mostly hand optimized kernels for accelerating math and deep learning based applications respectfully.

By relying on detailed knowledge of the hardware and kernel being optimized, with enough effort and manual exploration, people can often outperform compilers by taking advantage of optimizations compilers would never consider or have the time to explore [16, 136]. For example, GOTOBLAS [61] a hand-crafted BLAS (Basic Linear Algebra Subroutine) library frequently used in supercomputing, at one point contained many of the fastest BLAS implementations for a variety of CPU microarchitectures. Implementations were hand optimized down to the assembly level, and were manually tuned to avoid microarchitectural bottlenecks such as TLB (translation look-aside buffer) pressure. Existing compilers, including autotuning compilers, failed to beat GOTOBLAS partly because of their internal performance models abstracted away details such as overhead from TLBs.

Hand optimization is also common for targeting unconventional architectures such as VLIW (very long instruction word) and SIMD (single instruction multiple data) processors. Many modern compilers don't compile code for these architectures very well, partly due to the complexity of their non-scalar programming models. For example, autovectorization which transforms scalar code into vector code for SIMD machines is a common optimization where compiler rewrite rules perform poorly. Often they cannot recognize all the

vectorization opportunities or produce the global changes that are required to apply a rule [85]. These architectures have the potential for great performance because they can exploit more parallelism, but often programmers fall back to directly writing intrinsics or even assembly to achieve good performance.

With enough effort hand optimized code can beat automated techniques but at the cost of many human hours, and arguably only for people with enough domain knowledge in both the application and hardware being targeted. Furthermore, hand optimization is not a scalable technique, as it can take days or weeks to optimize a particular kernel for one hardware backend. Finally, the code is usually difficult to maintain and prone to errors. There are many combinations of programs and hardware that would benefit from this level optimization, but we do not have the time to hand optimize them.

2.2 SEARCH-BASED OPTIMIZATION METHODS

As hardware trends towards more heterogeneity and complexity, it opens up new opportunities for different types optimizations but also makes optimizations harder to develop, apply, and determine when they are beneficial. By sacrificing time and resources, search-based techniques can automatically explore different program implementations and potentially produce better code than hand-optimized implementations because they systematically explore different optimizations or constructions of programs.

One of the key features of these methods is they define a space of programs or implementations such as those defined by a grammar, set of axioms, or a template, and perform a best-effort search to minimize a cost function that estimates run time. The search can be formulated to use different solvers and optimizers, such as SAT, SMT, ILP, or simulated annealing, so that the search space is intelligently explored. Typically, search-based optimization is used in domain specific settings (e.g. optimizing machine learning kernels [33], GPU communication [30]) so that the search space can be constrained enough so that the search yields good results in a reasonable time.

2.2.1 *Polyhedral Optimization*

Polyhedral optimizers represent programs as a polyhedron, or n-dimensional geometric object, that is geometrically optimized then mapped back code [10, 62, 103, 142]. This has been successfully applied to perform optimizations such as loop scheduling [20], autovectorization [80, 134], and automatic parallelization [24, 54] which are common optimizations in image processing and machine learning applications.

For example, while optimizing nested loop schedules, the polyhedral method first represents the loop as a polyhedra. The shape of the polyhedra is determined by the loop bounds and is described by a set of affine inequalities, and

each loop statement is a point within a polyhedra. The polyhedra can then be transformed with affine functions representing different transformations such as loop reordering or tiling that change the sequence statements are executed.

Optimizing the loop schedule is formulated as a problem of finding the coefficients and offsets of the affine functions representing the transformations that obey data dependencies for correctness and minimize a cost function for optimality. The cost functions are hand-crafted mathematical expressions designed to minimize latency, maximize parallelism, enhance data locality, or some combination of different objectives. Depending on the cost function the optimization problem is usually solved with integer linear programming or a greedy algorithm [20]. Once the final transformed polyhedra is found, it is mapped back down to an optimized loop.

Polyhedral optimization scale with structure of the loop and not the size or number of elements it manipulates. Furthermore, multiple transformations can be compactly represented so that a large space of transformations can be explored. However, polyhedral optimization have mainly been used for optimizing static control parts or nested loops where the bounds are statically known and can be described with affine functions. Expanding polyhedral optimization to target arbitrary loops and other program structure is ongoing, but challenging because the programs and their transformation must be mapped to affine inequalities and functions [17]. Additionally, transformation must be efficiently translated back from the polyhedral representation to code which is not always straightforward.

2.2.2 *Autotuning*

Autotuners operate by tuning algorithmic strategies and parameters by empirically measuring the execution time of different configurations and selecting the best encountered. Autotuning is a commonly employed by many domain specific compilers and libraries [6, 7, 11, 121, 130, 145] where the programs being optimized have many implementations and the runtime costs of the implementations are difficult to predict, making it challenging to build an analytical cost model. This problem setting is common for highly parallel programs found in machine learning, signal processing, and scientific computing where there is lots of data independent computation that can be scheduled many ways. Selecting the best variant depends on both the problem and the hardware architecture, and there is no one size fits all strategy. Because the space of different implementations can be massive, autotuners employ intelligent search algorithms to decide what configurations to measure to avoid brute force measuring the entire space.

For example, TVM [34] a deep learning compiler includes an autotuning optimizer, AutoTVM [33]. In TVM, neural networks are decomposed into a sequence of individual operators. Each operator can have multiple algorithmic

strategies that are implemented by a schedule template that exposes parameters that control low-level implementation details such as the threading and tiling strategies of the algorithm. Together the different algorithms and their schedule templates form a large space of concrete implementations, and AutoTVM searches for the best implementation by running different configurations on the target hardware and selecting the best seen. Because the space of concrete implementations can be massive, simulated annealing is used to intelligently search the space, and execution time is used to train a learned statistical cost model that informs the search what to explore next.

Autotuners offer a low effort solution for performance portability by optimizing programs to run on different hardware. Assuming the hardware is not radically different, such as for different CPU microarchitectures, the same search space can be re-explored for an optimized implementation without any extra human effort. Additionally, pure autotuning does not require the programmer come up with a cost model of the hardware target as execution time is used to guide the search. As hardware gets more complex, building an analytical cost model becomes a challenging task which autotuning does not need to rely on. Instead by analyzing patterns in the execution time, the search algorithm indirectly builds up a cost model that helps guide it towards better implementation.

For large programs, autotuning can be very time and resource intensive as many different program variations need to be compiled and run. For this reason autotuning is usually performed offline. Additionally, the space of optimizations autotuners explore tends to be limited to pre-determined strategies and parameters. Targeting radically different hardware such as a GPU, often requires a different set of algorithmic strategies and parameters to tune.

2.2.3 *Program Synthesis*

Program synthesis is the task of automatically generating programs from a specification or a description of how they should behave, such as input-output examples or reference program. The standard synthesis problem focuses on generating any valid program; however, it can also be phrased as an optimal synthesis problem where the goal is to find the optimal solution with respect to a cost function [22, 63]. Even if a synthesis procedure does not directly optimize a cost function, synthesis can also be used to search for optimized programs by helping programmers discover optimized algorithms by completing partial programs [38, 99].

Superoptimizers are a canonical example of using program synthesis to optimize code. The goal of a superoptimizer is to find the cheapest instruction sequences for a given program or fragment by searching through the space of all instruction sequences [75, 101, 117, 118]. Superoptimizers can be useful

in automatically generating peephole optimizations for compilers or even optimizing small code fragments during compilation.

Synthesis can be divided into two main techniques, deductive and inductive synthesis, which are described below.

2.2.3.1 *Deductive Synthesis*

Deductive synthesizers derive a program by applying semantics preserving rewrite rules or axioms to the specification [28, 86]. Unlike a typical compiler optimization rewrite pass that applies a fixed sequence of rules to optimize a program, deductive synthesizers must select the axioms to apply and their ordering to derive a program. In practice, selecting the right axioms is not straightforward and requires searching over the axioms and even user guidance depending on the problem. Some deductive synthesizers, such as Denali [75], work by compactly representing all the equivalent programs that can be derived by applying the appropriate axioms and extract out the optimal sequence according to a cost function [129, 146]. Depending on the cost function, different solvers can be used such as integer linear programming [150] or SAT [75] to extract the best program.

Deductive synthesis has proven useful for both generating and optimizing domain specific code [97, 144] from DSP kernels [104, 141] to CAD models [93], and building assembly code superoptmizers [75]. One of the key challenges in building deductive synthesizers is developing a sufficiently expressive set of axioms, which often requires expert knowledge in the domain being targeted. The axioms are used to build up the program search space; if an axiom is missing the search space could preclude efficient implementations. The advantage is deductive synthesizers will search over programs that are semantically equivalent to the source program and can potentially scale better than inductive synthesizers [141].

2.2.3.2 *Inductive Synthesis*

Inductive synthesizers solve the problem of generating a program as a guess and check problem, usually through a technique called counterexample-guided inductive synthesis (CEGIS) [125]. Instead of deriving a program through axioms, a space of programs, defined by a grammar, is explored for a correct solution using a verifier to reject incorrect solutions. The CEGIS algorithm solves this by searching for a program that is correct for a few inputs then verifying it is correct for all inputs, repeating the search with feedback from verification until a solution has been found.

Optimization is usually built into this search process as typically smaller programs are explored first and the search is expanded to larger programs until successful, taking advantage of the fact smaller programs are often faster [99]. Alternatively, metasketches [22] can be used search over multiple candidate spaces with a gradient function that directs the parallel searches. CEGIS can be

used to minimize a more complex cost function by making cost a correctness requirement. By iteratively searching for a solution cheaper than the previous solution, we can guarantee the optimal program within a space is found [44, 122].

Inductive synthesizers have been used to optimize bitvector algorithms [65], GPU data movement [100], Homomorphic Encryption programs [44, 84], and more [38, 100, 117, 118]. One of the key strengths of inductive synthesis is its ability to discover optimizations without the programmer explicitly encoding the axioms that were required to reach the optimization like in deductive synthesis. Instead the synthesizer enumerates program candidates and relies on a verifier to filter out incorrect candidates which requires the user provide an *interpreter* so that it can evaluate programs.

The limiting factor that prevents inductive synthesis from being widely deployed is time and tractability. In practice, only small or constrained programs can be optimized in reasonable time. Because the search space can include many incorrect candidates, the synthesizer spends time evaluating incorrect programs. Many CEGIS based synthesizers rely on the user to provide a partial program, or *sketch* [124], to provide structure to the search space that is intended to restrict the space towards good solutions. This requires the user to have some prior idea of what optimizations are likely to be helpful, but this could also prevent the synthesizer from discovering radically different new optimizations.

2.3 SUMMARY

This chapter provides a background survey on different optimization strategies, focusing on five techniques: rewrite rules, hand optimization, polyhedral optimization, autotuning, and program synthesis. While techniques were discussed separately, in practice compilers often employ multiple methods that target different optimizations and work together to overcome their limitations.

For example, Lee et. al. proposed Lobster [84], a term rewriting system for HE programs that combines rewriting and program synthesis. Offline, Lobster synthesizes a set of rewrite rules by optimizing a large set of examples programs. The rules are then applied during compilation to optimize HE programs. By combining these two techniques, Lobster automates the process of generating rewrite rules with program synthesis and keeps the speed of term rewriting to keep compilation time reasonable.

2.3.0.1 *This Work*

In the remainder of this thesis, I present two systems for optimizing ultra quantized neural networks and HE programs that use search to optimize and generate code. When optimizing ultra quantized neural networks, I employ both autotuning and synthesis to generate efficient ultra quantized operators. Autotuning is used to refine an operator template that exposes parameters

that determine how computation is scheduled onto hardware resources. At a low level, program synthesis is used to select the SIMD intrinsic instructions used in the microkernel, or inner-most loop of computation. Together these two strategies optimize the memory access pattern of the ultra quantized operators to the exact CPU target and optimize instruction selection to utilize special SIMD intrinsics exposed by the hardware.

The second system I present is a vectorizing compiler for HE that relies on program synthesis. Synthesis is used to automatically generate a vector HE program from a scalar plaintext program. Our synthesizer is able to find optimized vector implementations that map computation onto a restricted instruction set, interleaving data shuffling with computation and tracking HE's unusual cost model.

A common theme of both these systems is that they rely on synthesis to discover vectorized code. Vectorized code tends to be more challenging for rewrite-based optimizers to handle because the rules must explicitly reason about multiple data elements simultaneously, and they are challenging to both write and determine when they can be safely applied. Using synthesis we avoid this problem of writing new rules for a new domain or unconventional CPU intrinsics. In the following chapters, I describe these two systems in depth.

3

ULTRA QUANTIZED MACHINE LEARNING MODELS

This chapter presents an automated approach to implementing quantized inference for machine learning models. We integrate the choice of how to lay out quantized values into the scheduling phase of a machine learning compiler, allowing it to be optimized in concert with tiling and parallelization decisions. After scheduling, we use program synthesis to automatically generate efficient low-level operator implementations for the desired precision and data layout. On a ResNet18 model, our generated code outperforms an optimized floating-point baseline by up to 3.9×, and a state-of-the-art quantized implementation by up to 16.6×.

3.1 INTRODUCTION

Quantization techniques reduce the scale of machine models to make them practical in resource-constrained environments such as smartphones and edge devices [71]. Quantization reduces the bit-width of both the data (weights) and the computations (additions and multiplications) used to execute the model. In the extreme, even single-bit weights and computations can still yield an effective model (with under 10% accuracy loss) while being orders of magnitude more efficient [43, 107]. This trade-off is appropriate for applications such as facial recognition, which may use a lower-accuracy, on-device model as a low-pass filter to decide whether to offload to the cloud for more expensive, more accurate prediction [88].

Quantized inference evokes a classic accuracy–performance trade-off across a large space of potential configurations. At training time, users must select quantized bit-widths to meet their performance and accuracy targets. They must then efficiently implement inference at that chosen quantization level for the target platform. Commodity hardware rarely supports ultra-low-bitwidth operations, and so naive quantized implementations can easily be slower than full-precision versions. These challenges are exacerbated by pressures across the stack. Rapid progress in new machine learning models demands new specialized quantized operations (e.g., new matrix sizes or new combinations of precisions). Meanwhile, a broad spectrum of hardware backends each require different implementations to best realize performance gains from quantization.

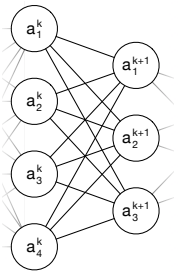
In practice, users deploying quantized inference are restricted to a small class of pre-set configurations with corresponding hand-tuned inference implementations. Developing these hand-tuned implementations is a tedious and labor-intensive process that requires intimate knowledge of the target architecture. One developer estimated that implementing and tuning a single quantization

configuration on a single smartphone platform [139] required four person-months of work [137]. Similarly, the authors of another library [136] spent about a month implementing a single configuration [135]. These hand-tuned implementations are often brittle and difficult to extend with new optimizations; both authors reported difficulty with multi-threading and new vector extensions. With the proliferation of new models and hardware backends, investing this level of effort for every configuration is not a scalable approach.

We present a new automated approach to generating efficient code for quantized machine learning inference. Our approach comprises two parts. First, we integrate the choice of how to perform *bit-slicing* (i.e., how to arrange quantized values in memory) into the scheduling phase of a machine learning compiler [34]. Our insight is that quantization is best viewed as a *scheduling* decision, in the same way that tensor libraries treat tiling and parallelism as part of the schedule space [106]. The bit-slicing choice is critical to generating efficient code: different layouts influence both spatial locality and the ability to apply different low-level operators for efficient computation (e.g., without scatter or gather operations on vector lanes). Making quantization a scheduling decision also allows us to extract better performance using holistic schedules that combine quantization with other transformations such as loop tiling and parallelization. Finally, integrating quantization into scheduling allows us to apply existing automated scheduling techniques [33] to optimize a model for a given quantization configuration and hardware architecture. This automation supports flexible experimentation and retargeting to new hardware without reengineering operator implementations by hand each time.

Second, to provide quantized low-level kernels for the machine learning compiler to target, we apply program synthesis [125] to automatically generate efficient implementations of each desired quantized operator. Our synthesis approach builds on the new notion of a *reduction sketch*, which captures domain knowledge about the structure of an operator's computation and allows the synthesis to scale to real-world implementations. The synthesis process takes as input a specification of the hardware's instruction set, the desired operator output, the layout of input data, and a cost function, and automatically synthesizes a sequence of straight-line assembly code that produces the desired output while minimizing the cost function. We implement reduction sketches and our synthesis engine in the Rosette solver-aided language [133], and integrate the generated code into the TVM machine learning compiler [34]. Using program synthesis further supports our goal of retargetability—porting the operators to a new hardware architecture (e.g., new vector extensions) requires only a specification of that hardware's instruction set, from which the necessary implementations can be automatically synthesized.

We evaluate our approach by using it to implement quantized ResNet-18 [69] image classification models. The resulting implementation on a low-power ARM CPU outperforms a floating-point version by 3.9× and is comparable to a state-of-the-art hand-written quantized implementation [136] when configured in the



$$\begin{bmatrix} w_{11}^k & w_{21}^k & w_{31}^k & w_{41}^k \\ w_{12}^k & w_{22}^k & w_{32}^k & w_{42}^k \\ w_{13}^k & w_{23}^k & w_{33}^k & w_{43}^k \end{bmatrix} \begin{bmatrix} a_1^k \\ a_2^k \\ a_3^k \\ a_4^k \end{bmatrix} = \begin{bmatrix} a_1^{k+1} \\ a_2^{k+1} \\ a_3^{k+1} \end{bmatrix}$$

Figure 3.1: In a fully connected neural network, computing the activations for layer $k+1$ involves a matrix-vector multiplication, which reduces to a series of vector dot products.

same way. However, our automated approach allows us to apply optimizations such as parallelization—whereas the hand-written implementation is single-threaded—that yield up to a 16.6× speedup over the hand-written quantized implementation. We show that our quantized implementations shift the Pareto frontier of the accuracy–performance trade-off, offering 2.3× higher throughput than existing implementations at the same 22% accuracy loss.

3.2 QUANTIZED MODELS

Quantized machine learning models promise to reduce the compute and memory demands of inference while maintaining acceptable accuracy. We focus on both fully connected and convolutional neural networks as they are two popular styles of networks. In these networks, computing the output of a layer involves a matrix multiplication followed by an activation function. For example, in a fully connected network, we can compute the activations for layer $k+1$ as the matrix-vector product of the weights for layer k and activations for layer k, as shown in Figure 3.1. A convolutional network is similar but with higher-dimension tensors for the weights and activations (i.e., more dot products required).

In a regular implementation of such a model, the weights w_{ij} and activations a_i would each be represented as a floating point number. This representation allows the machine learning compiler (e.g., TensorFlow [1]) to leverage decades of work on optimized floating point linear algebra operators in libraries such as Intel's MKL [42] and NVIDIA's cuDNN [37].

However, recent results have shown that the full dynamic range of a floating point representation is rarely necessary to achieve good inference accuracy. It is now common for inference to use 8- or 16- bit fixed point representations [72]. Taking this trend even further, reducing the bitwidths of the weights and activations to between two and four bits achieves competitive accuracy on image recognition problems [71, 149]. In the extreme, even models with single-bit weights and activations can achieve near state-of-the-art results on some problems [43, 107].

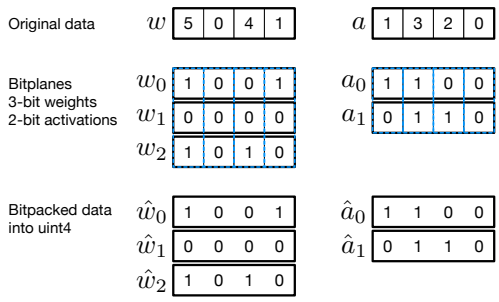


Figure 3.2: Slicing the values of weights and activations into bitplanes enables vector dot products to be computed with bitwise operations.

SINGLE-BIT QUANTIZATION. The advantage of quantized models is their lower memory and compute requirement, making them suitable for resource-constrained environments. In principle, their memory footprint is 32× smaller (going from single-precision floats to 1-bit values). They require less compute because they can use inexpensive bitwise operations rather than floating point. In Figure 3.1, computing each output activation a_i^{k+1} requires a vector dot product, which reduces to 4 floating-point multiplications and 3 floating-point additions. But if the activations and weights are quantized to one bit, we can replace each multiplication with logical ANDs and the additions with a single population count. If we pack the bits for each weight w_{ij}^k in row i into a single four-bit bit-vector $\hat{w}^k$, and likewise pack the activations a_i^k into a four-bit bit-vector $\hat{a}_i^k$, then we can compute:

$$\hat{w}^k \cdot \hat{a}^k = \text{popcount}(\hat{w}^k \,\&\, \hat{a}^k)$$

If implemented efficiently for the target architecture, this quantized version uses only simple bitwise operations and can be much faster than a floating-point version. Because it packs multiple weights and activations into a single vector, it can also exploit data parallelism for higher performance.

MULTI-BIT QUANTIZATION. Quantizing to single-bit weights and activations may yield unacceptable accuracy loss for some models. We can extend the above approach to larger weights and activations by *slicing* the bits of the weights and activations into bitplanes and then packing them into vectors. Figure 3.2 shows an example in which we quantize weights to 3 bits and activations to 2 bits. The first step is to decompose each value in the vectors w and a into their constituent bits at the corresponding bitwidth. The resulting vectors are called bitplanes; for example, the first bitplane vector w_0 holds the least-significant bit of each weight in the vector w. We can then pack each bitplane into larger elements—in this case, a single `uint4` value, but other configurations such as

two `uint2`s or four `uint1`s are possible. Given these bitpacked values $\hat{w}_i$ and $\hat{a}_i$, we can compute

$$\hat{w} \cdot \hat{a} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} 2^{n+m} \operatorname{popcount}(\hat{w}_n \,\&\, \hat{a}_m)$$

where N and M are the bitwidths for weights and activations, respectively ($N = 3$ and $M = 2$ in the example). This approach maintains the benefits of the single-bit version (bitwise operations and data parallelism), but the number of popcount operations scales with the product $O(NM)$, and so it is only practical for ultra-low-precision quantization—in this work, we consider quantizations to three bits or fewer.

The above equations assume a *unipolar* encoding of $\hat{w}$ and $\hat{a}$ values, in which bits represent $\{0, 1\}$ values. These values can instead be *bipolar* encoded, mapping $\{0, 1\}$ bits to $\{-1, +1\}$ values. Recent quantization work [29, 40, 149] uses unipolar encoding for activations and bipolar encoding for weights, making the dot product more complex:

$$\hat{w} \cdot \hat{a} = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} 2^{n+m} [\operatorname{popcount}(\hat{w}_n \,\&\, \hat{a}_m) - \operatorname{popcount}(\neg [\hat{w}_n] \,\&\, \hat{a}_m)]$$

This variation improves the accuracy of quantized models, at the cost of extra popcount operations. In this work we focus on generating implementations for this hybrid encoding, although our approach can be extended to other encodings.

3.3 QUANTIZATION SCHEDULES

Quantization promises to improve the performance of inference by using simpler bitwise operations on smaller values. However, a naive implementation of a quantized model is unlikely to be more efficient than a full-precision version. Mapping quantized computations onto commodity hardware requires careful data layout to make effective use of hardware resources and cannot leverage optimized floating point linear algebra libraries. This section introduces our approach to *scheduling* quantized computation on commodity hardware, and presents an automated workflow for identifying the optimal schedule for a given model on a chosen platform.

3.3.1 *Bit-Slicing Schedules*

A machine learning compiler such as TVM [34] reduces a model to a graph of *operator* invocations. An operator is a linear algebra primitive such as matrix multiplication or convolution, and is the fundamental building block of

a model. In a strategy introduced by Halide [106], each operator has both a declarative specification of its output and a *schedule* describing how to lower it to implementation code. The schedule captures transformations such as tiling and vectorization that preserve semantic equivalence but change the implementation to execute efficiently on a specific hardware target.

We observe that quantization, and in particular, the choice of how to lay out data by slicing values into bitplanes, is best viewed as a scheduling decision. Bitserial algorithms such as the dot products in Section 3.2 operate on each bit of a tensor element independently. They extract performance via data parallelism, packing the sliced bits of many elements together into a single element and computing on them together. But enabling this data parallelism requires choosing an axis along which to slice values into bits, in much the same way as loop tiling requires choosing which axes to tile. This choice critically influences performance, as it must balance spatial locality with the new optimization potential exposed by separating the bits.

To put this observation into action, we introduce a parameterizable bit-packing layout transformation to the scheduling process. It takes as input a d-dimensional tensor and returns a $d+1$-dimensional tensor, with a new *bit axis* that indexes the bitplanes of the original values. Bit-packing is parameterized by the reduction axis, bit axis, and datatype of the packed values (e.g. `uint4` in Figure 3.2). The reduction axis corresponds to the axis to slice and pack into bitplanes, and the bit axis is the location of the new slices. Bit-packing enables other data-layout transformations to operate at single-bit granularity by exposing the bit axis for use in scheduling.

Just as with tiling and other scheduling decisions, the application of the bit-packing transformation does not affect the semantics of the operator it is applied to, only how it is computed. The bit-packing transformation does *not* determine the desired quantization bitwidths of the values in the model. That choice is made at training time and is a known constant that cannot be changed during scheduling (as we would expect, since changing the bitwidth changes the output of the operator).

Figure 3.3 shows an example of applying the bit-packing transformation to a two-dimensional $M \times K$ tensor, with K as the reduction axis. Each resulting tensor is three-dimensional, with a new bit axis B that indexes the bits of each original element. The K' axis after transformation has $K' \leqslant K$, as bits from multiple contiguous values in the K dimension may be packed together into a single element in the transformed tensor according to the schedule's chosen datatype for packed values (e.g., in Figure 3.2, four contiguous values were packed into a single `uint4` element). The different results of the transformation reflect different choices of the location of the bit axis within the tensor. For example, the $M \times K' \times B$ layout indexes the bits as the innermost dimension, placing bits from the same original element contiguously in the tensor.

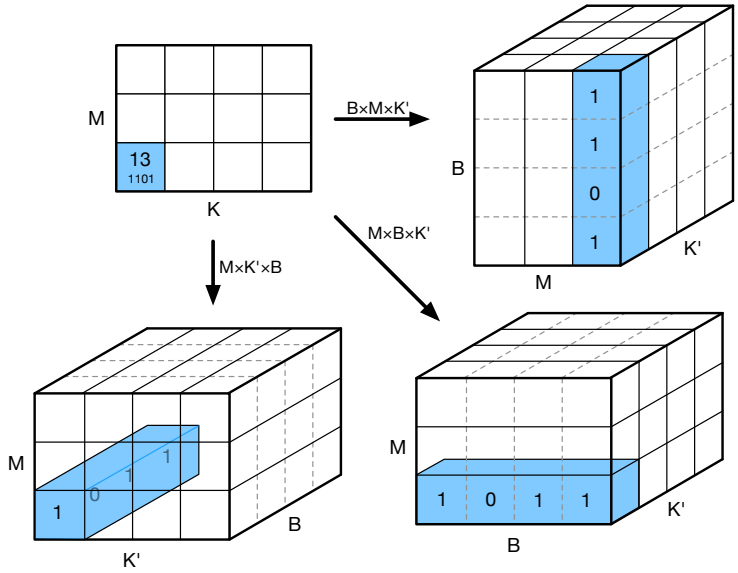


Figure 3.3: The bit-packing transformation transforms a d-dimensional tensor into a $d+1$-dimensional one by slicing the bits of elements into a new *bit axis*. Here the two-dimensional tensor $M \times K$ is transformed in different ways depending on the chosen location of the new bit axis B.

OPERATORS SUPPORTING BIT-PACKING SCHEDULING. We have implemented a library of operators that support the bit-packing transformation as part of their schedules. The library targets common neural network operators such as 2D convolutions and dense matrix multiplication (GEMM). The operators are implemented in TVM's tensor expression language [34], which is similar to those of Halide [106] and TACO [79]. For convolutions, our library contains variants that accept different high-level data layouts such as NCHW and NHWC, two common data layouts used in machine learning compilers. All operators are also parameterized by their quantization precision. We show in Section 3.4 how to automatically synthesize implementations for each such precision.

3.3.2 *End-to-End Scheduling*

Making quantization a scheduling decision allows us to integrate it with other scheduling choices and make holistic optimizations that consider the entire computation. In contrast, existing approaches to implementing quantized models intertwine the application of optimizations such as vectorization with the semantics of the quantized operator [136]. This ad hoc approach to scheduling tightly couples the implementation to a specific model and platform; new models or platforms might benefit from different optimizations, but exploring them would require new hand-tuned implementations.

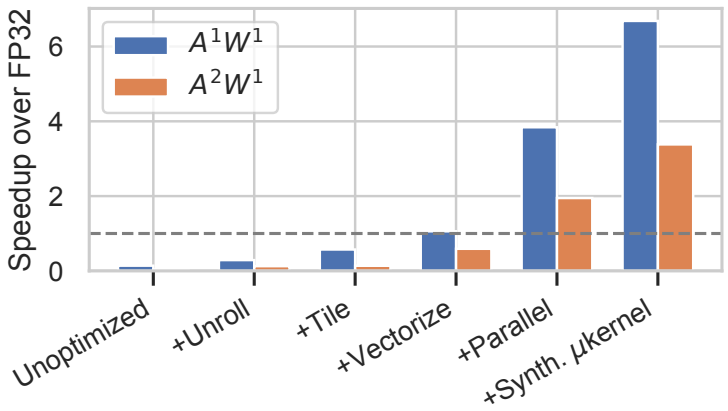


Figure 3.4: The cumulative effects of adding scheduling primitives, starting from an un-optimized quantized convolution. A^XW^Y denotes an x-bit activation, y-bit weight quantized convolution. Speedup is relative to 32-bit floating point on ResNet-18 layer C6.

We reuse the scheduling primitives of TVM [34], which itself uses the scheduling primitives of Halide [106]. In particular, a number of common scheduling primitives are useful in conjunction with bit-packing:

- **Loop tiling** splits input tensors into tiles that each fit in the cache, improving temporal locality and reducing memory traffic.
- **Loop unrolling** replicates loop bodies to reduce the overhead of maintaining induction variables and checking exit conditions
- **Vectorization** takes advantage of hardware SIMD instructions to operate on multiple elements at once
- **Parallelization** takes advantage of hardware MIMD facilities—in our low power use cases, this means multiple cores on a single multiprocessor

Each of these primitives is parameterized by the tensor axis along which to transform the implementation.

Figure 3.4 shows the importance of these optimizations to the performance of a quantized model (Section 3.5 will detail the methodology). Without any optimizations, quantized models perform much worse than an optimized floating-point baseline. By integrating quantization into scheduling, we can compose these standard optimizations without rewriting the quantized implementation. Almost all these optimizations are necesssary for the quantized versions to outperform floating point. Finally, adding a synthesized microkernel (the final column of Figure 3.4) improves performance by another 1.5–2× over the optimized quantized model; Section 3.4 presents our approach to generating this kernel.

AUTOMATED SCHEDULING. Bringing quantization into the scheduling domain also allows us to use *automated* scheduling techniques (e.g., autotuning)

to tune a given quantized model for a particular hardware architecture. Each scheduling primitive has a number of parameters (e.g., which axis to transform) that have critical influence on the performance of the resulting code. We use the AutoTVM automatic scheduling framework to choose the values of these parameters [33]. AutoTVM searches the space of possible schedules (tiling, vectorization, etc.) and learns a cost model that maps schedules to predicted performance based on trials executed on the target hardware. Using AutoTVM allows practitioners to experiment with quantization on new hardware and new models without manual effort to hand-tune the schedule each time.

We do not use AutoTVM to choose the bit axis for the bit-packed schedule primitive because AutoTVM requires output tensor shapes to be static. Our schedules all use fixed bit axes we found to work well. AutoTVM was used during this process to quickly optimize over different iterations, reducing much of the work in selecting the axis.

3.4 MICROKERNEL SYNTHESIS

After scheduling a computation, the machine learning compiler must map it down to the target hardware architecture. This process requires lowering the computation onto available low-level *microkernels* that implement primitives such as matrix multiplication. For a quantized model, these primitives must operate at ultra low precision (e.g., multiplying matrices with 2-bit values). Off-the-shelf linear algebra libraries do not offer kernels with such low precisions, and writing them by hand is tedious since each additional bit of precision requires a different implementation to extract maximal performance.

This section introduces a new automated approach to implementing low-level primitives for the compiler to target. The key idea is to reduce the problem to *program synthesis*, the task of automatically generating a program that implements a desired specification [65, 125]. We show how to decompose the matrix multiplication primitive into orthogonal parts that can be synthesized separately, and layer a cost function over the synthesis process that drives it towards efficient kernels. Together with automated scheduling, using program synthesis for automated low-level primitives enables an end-to-end automated pipeline for compiling quantized models on any desired hardware platform.

3.4.1 *Specifications*

A program synthesizer takes as input a specification of the desired program behavior, and outputs a program that implements that behavior in a chosen language. We focus on synthesizing implementations of matrix multiplication, since it is the common low-level primitive for the convolutional and fully connected neural network models we consider in this work. Our goal is to syn-

thesize vectorized kernels that efficiently implement the matrix multiplication primitive with the desired shapes and precisions.

KERNEL SPECIFICATION. To define the desired behavior of the synthesized microkernel, our synthesis engine takes as input an unoptimized bitserial matrix multiplication implementation written in the target assembly language. This *reference implementation* fully specifies the desired behavior, including the shape and precision of each input and output matrix as determined by the schedule. For example, the schedule might require a matrix multiplication between an 8×16 matrix with 2-bit values (the weights) and a 16×1 matrix with 1-bit values (the activations). We can easily generate a reference implementation to use as the specification for this kernel by translating the declarative tensor expression generated by the compiler. To avoid reasoning about memory layout during synthesis, we use the reference implementation to pre-define the registers that hold the input matrix and the registers that should hold the output matrix.

ARCHITECTURE SPECIFICATION. Our synthesis engine outputs straight-line assembly code that implements the desired microkernel, and so it requires a specification of the target architecture. We follow the Rosette *solver-aided language* approach [132, 133], in which we specify the target architecture by writing an interpreter for *concrete* assembly syntax in Racket. Rosette uses symbolic execution to automatically lift this concrete interpreter for use in program verification and synthesis. The interpreter also serves a second role, giving a semantics to the reference implementation that we use as the kernel specification.

While writing an interpreter for an instruction set may seem more expensive than writing a microkernel, ISA designers are increasingly publishing reference semantics for their instruction sets that can be used as interpreters [111], and in practice only a small subset of the instruction set is required. For example, to synthesize code for the ARM NEON vectorized instruction set, we write an interpreter in Racket for NEON instructions:

```
(define (interpret prgm state)
  (for ([insn prgm])
    (match insn
      [(vand dst r1 r2)
       (state-set! state dst
                   (bvand (state-ref state r1)
                          (state-ref state r2)))]
      [(vor dst r1 r2)
       ...]
      ...)))
```

The interpreter iterates over the instructions in the input program `prgm`. For each instruction, it uses pattern matching to dispatch to an implementation that manipulates the machine state `state`. The implementation of the `vand` instruction updates the destination register `dst` to hold the bitwise-and of the two source registers `r1` and `r2` (in fact, `vand` is a vector operation and so should

update every element of the *vector* register `dst`, but we elide the details for simplicity).

COST FUNCTION. We want the synthesizer to find the *most efficient* implementation of the desired kernel on the target architecture (i.e., we are essentially superoptimizing the reference implementation [87, 100]). The synthesis engine thus takes as input a cost function, mapping each program to an integer cost, and finds the program with minimum cost [22]. Statically estimating program performance is difficult, so out cost function is simply program length. More realistic cost functions might guide the synthesizer to more efficient implementations, but this simple cost function still generates high-quality code in practice.

3.4.2 *Compute and Reduction Sketches*

Given the inputs above, our synthesis engine searches for a program (a straight-line assembly sequence) that implements the desired specification. To define the search space for the synthesis tool to explore, we write a *sketch* [125], a syntactic program template containing *holes* that the synthesizer will try to fill in with program expressions.

To make the synthesis problem tractable, we decompose it into two separate phases and write a sketch for each. The *compute sketch* implements the initial computation for the matrix multiplication, while the *reduction sketch* implements the reduction that collects the computed results into the right locations in registers. This separation echoes the two phases of matrix multiplication: first compute the dot products for each necessary pair of vectors, and then arrange them in the right places in memory.

COMPUTE SKETCH. The compute sketch performs the necessary vector dot products on quantized values, as defined in Section 3.2. The sketch is a straight-line sequence of assembly code, where each instruction can be either a bitwise operation (and, or, not, addition, etc.) or a special population count intrinsic instruction. In both cases, the synthesizer is free to choose any live registers as the inputs to the instruction, and any register as the output. To break symmetries in the search, output registers must be written to in increasing order. In Rosette, we represent this sketch with functions that evaluate nondeterministically using the `choose*` built-in form:

```
(define (??insn)  ; choose an arbitrary instruction
  (choose* vadd vand vcnt ...))
(define (??reg n)  ; choose a register in range [0,n)
  (reg (apply choose* (range n))))

; sketch of length k with n inputs
(define (sketch k n)
  (define (r i) (??reg (+ i n))) ; input or live reg
  (list
    ((??insn) (r 1) (r 1) (r 1))
```

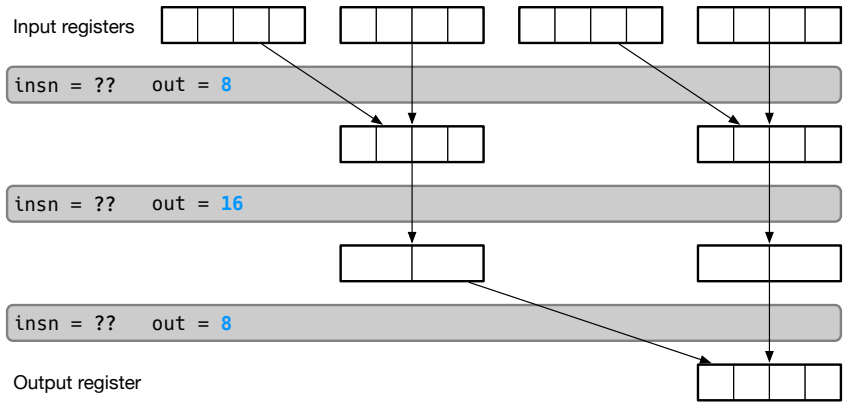


Figure 3.5: A reduction sketch is a tree that reduces input registers down to a single register. Each level applies an instruction to each pair of live registers. Here the output types have already been synthesized; each level is free to decide the types (vector lane widths) of its outputs.

```
((??insn) (r 2) (r 2) (r 2))
((??insn) (r 3) (r 3) (r 3))
...))  ; k lines
```

The compute sketch has considerable freedom to find novel efficient implementations. For example, it can manipulate packed quantized values at the bit level, decide when and how to reuse intermediate values, etc.

REDUCTION SKETCH. The reduction sketch takes as input the dot products from the compute sketch and sums them along the reduction axis of the matrix multiplication. This reduction phase is difficult because we are targeting vectorized code. To avoid overflowing hardware vector lanes, additions must be performed in the correct order, and values promoted to different vector lane widths as necessary. This datatype polymorphism is challenging for synthesis.

To scale up this reasoning, the reduction sketch exploits the tree-like structure of the reduction computation, similar to the common reduction tree parallel programming pattern. The reduction sketch, illustrated in Figure 3.5, is a tree that takes as input the live registers, the datatype of each live register, and the depth of the tree to be synthesized. At each level of the tree, the reduction sketch contains a single hole defining the instruction to apply at that level. The sketch applies the same instruction, in order, to every register pair that remains live at that level. The selected instruction dictates the output datatypes and the number of output registers (which varies depending on whether the instruction promotes to a wider bit-width); in Figure 3.5, the output types are already synthesized, to show their effects on the output registers. The final output of the reduction sketch is a single vector register that holds the entire (packed) result of the matrix multiplication.

```
8×  vand.8    d0, d0, d1
    vcnt.8    d0, d0

    vadd.8    d0, d0, d1
    vadd.8    d0, d0, d1
    vadd.8    d0, d0, d1
    vadd.8    d0, d0, d1

    vpadd.8   d0, d0, d1
    vpadd.8   d0, d0, d1

    vpadal.8  q1, {d0, d1}

    vst1.16   q1, addr
```

(a) Synthesized (24 insns)

```
    vmovl.8   q0, d0
    vmovl.8   q2, d1
8×  vand      q0, q0, q2
    vcnt.8.   q0, q0
    vpaddl.8  q0, q0
    vadd.16   q1, q0, q0

    vst1.16   q1, addr
```

(b) TVM-generated (49 insns)

Figure 3.6: Microkernels for 8×8 by 8×1 matrix multiply with 1-bit values, generated by (a) our synthesis tool and (b) TVM's tensorization. The synthesized version is half the length ("8×" code is unrolled 8 times) and twice as fast.

3.4.3 *Implementation*

We implement our synthesis engine in the Rosette solver-aided language [133], which extends Racket with support for verification and synthesis. After synthesizing a desired microkernel, we integrate it into TVM's microkernel library for use when compiling models. Rather than directly integrating assembly code, we instead emit the corresponding SIMD intrinsics inside a C function that TVM compiles using LLVM. This abstraction allows the compiler to perform register allocation and interprocedural optimizations such as inlining.

ARM NEON. To target low-power ARM processors, we synthesize code in a subset of the ARM NEON vectorized instruction set. NEON machine state consists of 16 128-bit vector registers known as *quad* registers. Each quad register also has two aliases from 64-bit *double* registers that point to its upper and lower halves. Most NEON instructions can reference either register type; our synthesized kernels have the freedom to use the two interchangeably, and mixing the two modes leads to shorter, more efficient code.

Figure 3.6 shows an example of our synthesized microkernel for multiplying two matrices of shape 8×8 and 8×1, each with 1-bit quantized values. It also shows an equivalent microkernel generated by TVM, which lowers tensor operations to LLVM IR for code generation. Our synthesized kernel is half the length of TVM's version, and is more efficient for two main reasons. First, the TVM-generated version operates only on quad-precision registers because it eagerly promotes all values to 16 bits, giving up some of the benefit of quantization. Second, TVM cannot vectorize across the reduction axis, and so needs to perform broadcast loads into the vector lanes for at least one of the inputs. The last column of Figure 3.4 shows that the synthesized kernel is 1.5–2.5× faster on an ARM CPU.

3.5 EVALUATION

To evaluate the effectiveness of our automated approach to implementing quantized models, we address three research questions:

1. Do our automatically generated implementations outperform non-ultra-low-precision versions?
2. How do our implementations compare to hand-written quantized kernels?
3. Does our automation help to explore new quantization configurations efficiently?

METHODOLOGY. We test our quantized implementations on a low-power Raspberry Pi 3B with an ARM Cortex-A53 processor. The ARM processor has four cores at 1.2 GHz and supports NEON SIMD extensions. All experiments report the average of 10 runs with 95% confidence intervals.

We focus our evaluation on quantized versions of the ResNet18 model [69], because it is small enough to deploy in resource-constrained environments. ResNet18 is a neural network model for image classification comprising 18 layers. Its compute time is dominated by the convolutional layers described in Table 3.1.

Name	Operator	H, W	IC, OC	K, S
C2	conv2d	56, 56	64,64	3, 1
C3	conv2d	56, 56	64,64	1, 1
C4	conv2d	56, 56	64,128	3, 2
C5	conv2d	56, 56	64,128	1, 2
C6	conv2d	28, 28	128,128	3, 1
C7	conv2d	28, 28	128,256	3, 2
C8	conv2d	28, 28	128,256	1, 2
C9	conv2d	14, 14	256,256	3, 1
C10	conv2d	14, 14	256,512	3, 2
C11	conv2d	14, 14	256,512	1, 2
C12	conv2d	7, 7	512,512	3, 1

Table 3.1: Configurations of 2D-convolution operators in ResNet18 [69]. H and W are height and width, IC and OC are input and output channels, K is kernel size, and S is stride size. Layer 1 is omitted as its input channel depth is too small to allow efficient packing.

We refer to quantized kernels generated by our approach as A^xW^y, where x is the bitwidth of activations and y the bitwidth of weights [139]. Each kernel we generate, including floating-point baselines, is optimized independently using AutoTVM [33]. For each convolutional layer, we run AutoTVM for a total

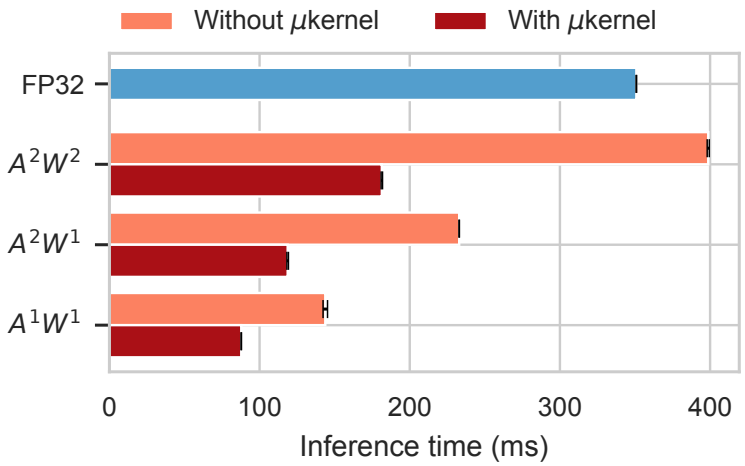


Figure 3.7: End-to-end inference times for quantized ResNet18 on an ARM CPU, with and without synthesized microkernels.

of 100 trials in parallel across 10 hosts. Our performance measurements include the cost of bitpacking the activation values, but not the weights, as they can be done offline before deployment.

Ultra quantized model architectures differ slightly from floating point models (e.g., by adding quantization layers), but maintain the number and sizes of convolutions, which dominate the compute cost. Quantized models are trained from scratch, and each desired quantization level requires retraining to maintain accuracy (so an A^2W^1 model cannot be quantized down from a A^2W^2 model). However, the shape of the model stays the same, and we so can compare run-time performance across quantization levels.

3.5.0.1 *Quantization versus Floating Point*

To demonstrate the performance benefits of quantization, Figure 3.7 shows end-to-end inference times on the ARM platform for both non-quantized and quantized models. The quantized results use our synthesized kernels and bit-packed schedules, while the 32-bit floating point result uses a heavily optimized pre-existing schedule in TVM. We find that the quantized model outperforms the floating-point version by up to 3.9×, confirming that quantization yields speedups in practice and a significant memory footprint reduction that is 32× to 16× smaller, depending on the weight bit-width.

Figure 3.7 also shows the importance of our synthesized microkernels for extracting performance from quantized models. Without the synthesized microkernels, TVM follows its default code generation strategy (lowering tensor expressions to LLVM IR). This strategy yields inefficient implementations that is actually slower then floating point for A^2W^2. End-to-end inference is an average of 1.9× faster using our synthesized microkernels, and all three configurations are faster than the floating point implementation.

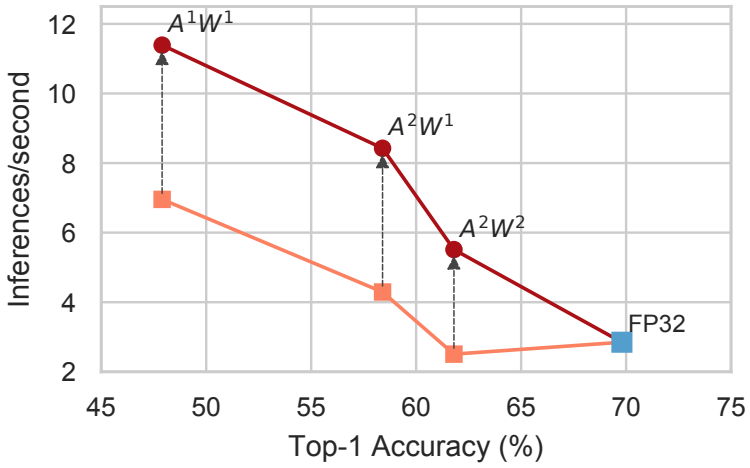


Figure 3.8: Accuracy versus performance for quantized ResNet18 on an ARM CPU, with and without synthesized microkernels. Accuracy versus performance. Higher is better on both axes.

The improved performance of our synthesized microkernels shift the Pareto frontier of the accuracy–performance trade-off for quantization. Figure 3.8 shows the image classification accuracy (measured as top-1 accuracy, i.e., the fraction of images correctly classified) of the quantized ResNet18 models against their inference performance. Without synthesized microkernels, the accuracy loss of quantization offers little performance benefit. Our synthesized implementations shift the Pareto frontier outwards, offering a choice of points in the design space for quantized models. For example, an 8% accuracy loss from FP32 yields 1.9× higher inference throughput (A^2W^2), and an additional 14% loss yields a further 2.1× throughput (A^1W^1).

LIMITATIONS. While our generated implementations offer significant speedups over the floating-point baseline, they are lower than the theoretical performance gain we would expect, due to a number of inefficiencies. First, quantized models still execute some layers in floating point, including the initial convolution (which we exclude from Table 3.1) , that limit potential speedups. For example, in the A^1W^1 model, the initial convolution is performed in floating-point and consumes 32% of total running time, versus only 8% in the floating-point model. Similarly, the operations between convolutions are often performed in floating point, requiring conversions from integer to floating point and back.

Second, our implementations must spend instructions re-packing bits into the appropriate data layout, whereas ARM has native support for single-precision floating-point sized data. This bitpacking consumes 2–3% of the end-to-end run time, and 13–21% of an individual convolution's run time.

Finally, in the floating-point implementation, the model compiler can take advantage of hardware fused multiply-add instructions and of alternative floating-point convolution algorithms. We could recoup some of these inefficiencies with more work on higher-level optimizations.

3.5.0.2 *Comparison to Hand-Written Code*

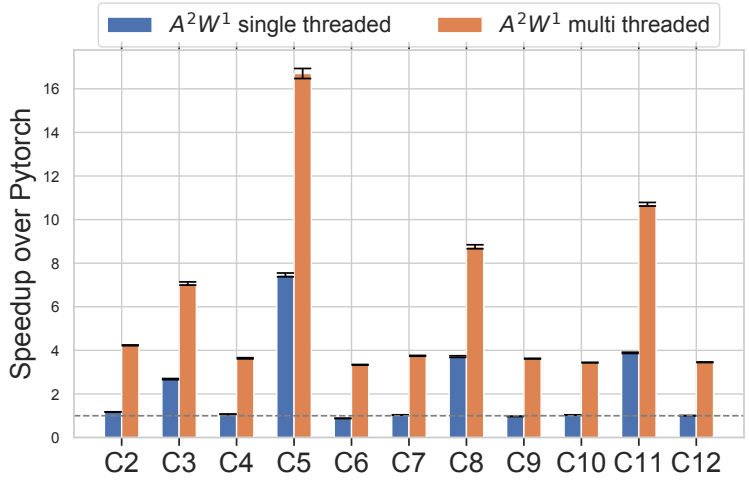


Figure 3.9: Layer-by-layer speedups for A^2W^1 convolutions normalized to PyTorch's hand-optimized A^2W^1 kernel for ARM [136]. The PyTorch kernels are single-threaded, so we compare against both single- and multi-threaded versions generated by our approach.

Our automatically generated quantized kernels a outperform hand-written quantized implementations developed by experts. Figure 3.9 compares our implementation to the hand-written ultra-low-precision convolution library in PyTorch [136]. The library was written specifically for the ARM architecture, and makes extensive use of NEON SIMD intrinsics and loop tiling for performance; it reaches about 70% of peak theoretical single-thread performance. The library focuses exclusively on A^2W^1 quantized convolutions and does not provide an end-to-end implementation of ResNet, so we compare performance on the individual convolutions rather than end-to-end inference time.

When we restrict our approach to single-threaded implementations, the performance of our synthesized kernels is generally comparable to the expert-written code, which is single-threaded. Our version does outperform PyTorch on some layers by up to 7× (C5, C8, C11); these layers are "down-sampling" convolutions that the PyTorch library does not optimize for. In contrast, our scheduling abstraction allows us to independently optimize each convolution.

Because our approach integrates scheduling and code generation, we can easily generate an optimized multi-threaded implementation without manual code changes. In Figure 3.9, our multi-threaded A^2W^1 convolution outperforms PyTorch by an average of 5.3× and up to 16.6×. This result demonstrates the flexibility of our automated approach to adapt to new hardware resources and new optimizations that were missing from hand-written kernels.

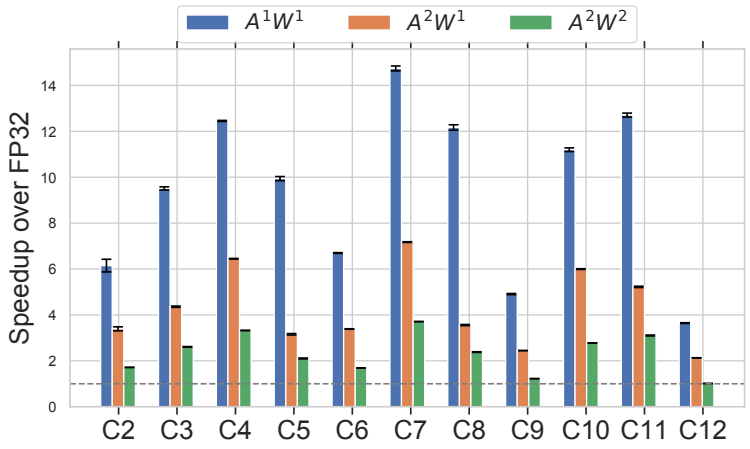


Figure 3.10: Quantized convolution speedups for various quantization configurations, normalized to 32-bit floating point.

3.5.0.3 *Exploring Quantization Configurations*

Our automation allows us to rapidly explore the potential benefits of different quantization configurations without hand-tuning for each new configuration and platform. We exploited this automation to perform a limit study of the possible speedups available at reduced precisions.

Figure 3.10 shows the performance of different configurations generated with our approach compared to a 32-bit floating-point baseline. The results support the intuition that smaller quantizations (e.g., A^1W^1) perform better. However, the performance benefit of quantization varies between layers—for example, A^1W^1 speedups vary from 3.6× to 14.7×. The speedup of a layer is correlated with the number of input channels in the convolution (Table 3.1). In our generated schedules, tensors are both bit-packed and vectorized along the input channel axis. Increasing the input channels therefore improves utilization of the available hardware resources. The larger convolutions also expand the working set beyond the ARM CPU's cache size in the floating-point version.

To further explore the limits of quantization, Figure 3.11 shows the performance of all configurations up to A^3W^3 on layer C2, which is representative of the average performance profile in Figure 3.10. The missing configurations require combining 8-bit and 16-bit arithmetic during the compute phase, which our compute sketch does not support. The potential performance benefit of quantization degrades rapidly as the number of bits increases, due to the $O(NM)$ scaling shown in Section 3.2. But even at A^2W^2, quantization offers a considerable speedup over floating point; coupled with the reduction in memory footprint, this result confirms the value of quantization in resource-constrained environments.

Figure 3.11 also shows the performance of our synthesis engine for each configuration, with more detail in Table 3.2. Synthesis performance worsens as the number of bits increases—from 17 seconds at A^1W^1 (two bits total) to

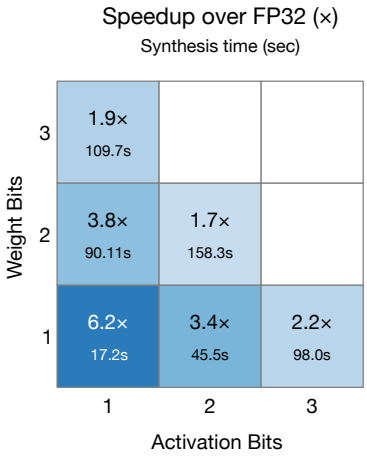


Figure 3.11: Relative speedup over 32-bit floating point, and kernel synthesis time, for convolutional layer C2 at different quantization configurations.

Configuration	Synthesis Time (s)				Scheduling Time (s)
	Compute	Reduction	Verify	Total	
A^1W^1	2.4	9.0	5.8	17.2	344
A^2W^1	21.0	13.9	10.6	45.5	328
A^1W^2	62.2	12.7	15.2	90.1	352
A^3W^1	62.2	18.9	15.2	158.3	435
A^1W^3	63.8	32.1	15.4	109.7	460
A^2W^2	90.4	52.4	15.5	158.3	334

Table 3.2: Microkernel synthesis and scheduling times. Synthesis time is broken down into time spent solving the compute and reduction sketches and verifying the solution. Scheduling time is the total across all convolutional layers.

158 seconds at A^2W^2 (four bits total). This poor scaling is because we allow the synthesizer to reason about individual bits, which gives it maximum freedom to find optimizations, but makes reasoning expensive.

Increases in the number of weight bits cause more dramatic performance degradation than increases in activation bits (e.g., A^1W^2 is slower to synthesize than A^2W^1), because weight matrices are larger than activation matrices and so more total bits are necessary.

Table 3.2 also shows the time required for AutoTVM scheduling of each configuration. Each time is the total scheduling time across all convolutional layers. Scheduling is largely independent of the number of bits, and so does not degrade as dramatically as the number of bits increases. However, scheduling time dominates synthesis time in all cases, and so we could narrow the scope of AutoTVM's search if faster end-to-end synthesis was necessary.

3.6 RELATED WORK

QUANTIZED NEURAL NETWORKS. Whereas our work focuses on efficiently *executing* quantized networks, most prior work focuses on the orthogonal problem of *training* such models to minimize accuracy loss due to quantization. BinaryNet [43] presents a training algorithm for *binary neural network* with weights and activations quantized to 1-bit. The resulting networks are competitive in accuracy to floating point networks on simple image recognition tasks such as hand written digit recognition. XNORNet [107] improved the accuracy of binary neural networks through architectural changes, but had much worse accuracy than floating point on complex recognition tasks, such as classification on ImageNet [116].

More recent quantized neural networks focus on reducing the accuracy gap on ImageNet by increasing the precision. DoReFa-Net [149] and HWGQ [29] quantize activations down to 2- to 4-bits while keeping 1-bit weights, and trained models with accuracy within 5% to 9% of floating point on the same model architecture. By quantizing both weights and activations to 2 bits, Choi et al. [40] further reduced the accuracy gap to 3.4% below floating point.

QUANTIZED MACHINE LEARNING KERNELS. Quantized neural networks are trained in floating point so that iterative small adjustments to the model can be made; even the forward pass during training only simulates quantization. There has been comparably little work on efficient inference for quantized neural networks, or on implementing the quantized operators they depend on. Umuroglu and Jahre [139] and Tulloch and Jia [136] present implementations of quantized machine learning kernels for ARM CPUs with hand-optimized code. Both implementations use ARM NEON instrinsics and rely on careful tiling to match register and cache sizes of their target devices. As a result, their peak performance is closely tied to their target device's microarchitecture.

BitFlow [70] is a hierarchical framework for implementing binary neural networks on CPUs. They present a code generator that efficiently maps 1-bit quantized operators to the appropriate SIMD compute kernels based on the operator dimensions, and divides work among the available cores to exploit MIMD parallelism. In contrast, we take a more holistic approach: our scheduling phase encompasses a variety of optimizations, including choices about how to bit-pack quantized data, that can be applied in any (valid) combination. Our approach also extends easily to quantizations beyond 1 bit.

Earlier versions of TVM [34] supported ultra-low-precision quantization using hand-written microkernels for ARM's NEON vector extensions [46]. These implementations are no longer supported because of the difficulty of maintaining them; recent TVM versions instead use the LLVM code generation approach to quantization that we outlined in Section 3.5. Performance of the old implementations is not directly comparable to ours, because they used unipolar encodings for quantization, whereas we use the more modern hybrid

unipolar–bipolar encoding (which requires twice the popcounts). Nonetheless, our synthesized microkernels are an average of 10% and up to 2.2× faster than TVM's previous hand-written kernels.

AUTOMATIC GENERATION OF KERNELS. Automatic generation of efficient floating-point tensor kernels is a long-standing problem. Tensor compilers such as TACO [79] and Halide [106] automatically generate kernels for sparse linear algebra and image processing, respectively. Machine learning compilers such as Tensor Comprehensions [142], GLOW [114], and TVM [34] generate efficient code specifically for machine learning models using domain-specific optimizations at both the graph and operator level. Our work extends this approach by focusing on quantized models, allowing us to exploit domain-specific knowledge (e.g., the bit-packing axis) to generate efficient implementations.

SPECIALIZED HARDWARE BACKENDS. The emergence of machine learning accelerators in ASICs [32, 35, 76] and FPGAs [41, 91, 138] has led to increased specialization of both hardware intrinsics (e.g. single instruction matrix multiplication) and data type selection. Accelerators that expose a quantized programming interface [77, 120, 140] offer new opportunities for our synthesis approach to extract performance. For example, an accelerator could offer specialized operations for A^nW^m quantizations, and our synthesis engine could compose these operations to reach the desired configuration for a given model. Since the optimal scheduling of a workload on an accelerator is a function of compute, memory bandwidth, and on-chip storage, our automated scheduling approach would benefit accelerators by delivering the best performance at all quantization settings.

PROGRAM SYNTHESIS. Our approach uses program synthesis to generate quantized tensor kernels, following the lead of many existing tools. Spiral [104] is a tool for generating high-performance implementations of fast Fourier transforms and other DSP primitives. Chlorophyll [100] is a superoptimizer for a low-power spatial architecture. Synapse [22] is a program synthesis framework for compiling low-level programs *optimally* with respect to a cost function. Our work further demonstrates the potential for program synthesis as a tool for generating efficient implementations of small performance-critical kernels that a regular compiler is unable to find.

3.7 CONCLUSION

Our automated approach to compiling quantized models combines the strengths of both machine learning (for scheduling [33]) and program synthesis (for code generation [133]). The result is a workflow that generates kernels that outperform than both optimized floating-point and state-of-the-art quantized imple-

mentations. Automation also helps machine learning practitioners experiment with new quantization regimes, model designs, and hardware architectures, all without having to re-engineer low-level kernels with each change. Our work makes quantized models practical to deploy in resource-constrained settings, bringing the successes of machine learning to a plethora of new environments.

4

HOMOMORPHIC ENCRYPTION

This chapter presents Porcupine, an optimizing compiler that generates vectorized HE code using program synthesis. Homomorphic encryption (HE) is a privacy-preserving technique that enables computation directly on encrypted data. Despite its promise, HE has seen limited use due to performance overheads and compilation challenges. Recent work has made significant advances to address the performance overheads but automatic compilation of efficient HE kernels remains relatively unexplored.

HE poses three major compilation challenges: it only supports a limited set of SIMD-like operators, it uses long-vector operands, and decryption can fail if ciphertext noise growth is not managed properly. Porcupine captures the underlying HE operator behavior so that it can automatically reason about the complex trade-offs imposed by the challenges and generate optimized, verified HE kernels. We evaluate Porcupine using a set of kernels and show speedups of up to 51% (11% geometric mean) compared to heuristic-driven hand-optimized kernels. Analysis of Porcupine's synthesized code reveals that optimal solutions are not always intuitive, underscoring the utility of automated reasoning in this domain.

4.1 INTRODUCTION

Homomorphic encryption (HE) is a rapidly maturing privacy-preserving technology that enables computation directly on encrypted data. HE enables secure remote computation, as cloud service providers can compute on data without viewing the data's contents. Despite its appeal, two key challenges prevent widespread HE adoption: performance and programmability. Today, most systems-oriented HE research has focused on overcoming the prohibitive performance overheads with high-performance software libraries [102, 119] and custom hardware [109, 112]. The performance results are encouraging with some suggesting that HE can approach real-time latency for certain applications with sufficiently large hardware resources [109]. Realizing the full potential of HE requires an analogous compiler effort to alleviate the code generation and programming challenges, which remain less explored.

Modern ring-based HE schemes pose three programming challenges: (i) they only provide a limited set of instructions (add, multiply, and rotate); (ii) operands are long vectors, on the order of thousands; (iii) ciphertexts have noise that grows as operations are performed and causes decryption to fail if too much accumulates. For instance, the Brakerski/Fan-Vercauteren (BFV) crpytosystem [53] operates on vectors that *pack* multiple data elements into a

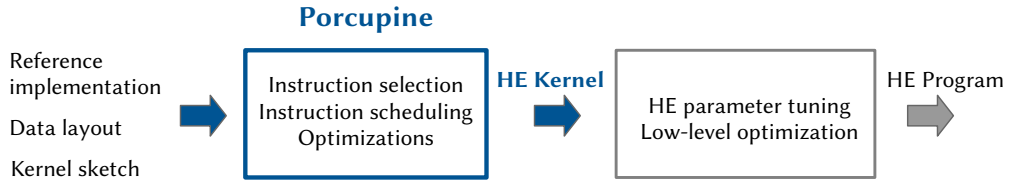


Figure 4.1: A high-level HE compiler flow. Porcupine performs vectorization and computation optimization.

single ciphertext to improve performance. Instructions operating on packed-vector ciphertexts can be abstracted as a SIMD (single instruction, multiple data) instruction set, which introduces vectorization challenges.

To target the instruction set, the programmer must break down an input kernel into SIMD addition, multiply, and rotation instructions, while minimizing noise accumulation. These challenges introduce a complex design space when implementing HE kernels. As a result, HE kernels are currently written by a limited set of experts fluent in "HE-assembly" and the details of ciphertext noise. Even for experts, this process is laborious. As a result, hand-writing HE programs does not scale beyond a few kernels. *Thus, automated compiler support for HE is needed for it to emerge as a viable solution for privacy-preserving computation.*

A nascent body of prior work exists and has investigated specific aspects of compiling HE code. For example, prior work has shown HE parameter tuning, which determines the noise budget, can be automated and optimized to improve performance [8, 31, 49, 50]. Others have proposed mechanisms to optimize data layouts for neural networks [50]. Prior solutions have also used a mix of symbolic execution [8] and rewrite rules [18, 31, 49] for code generation and optimizations for logic minimization (e.g., Boolean logic minimization [31, 84]. Each of these lines of work have advanced the field and addressed notable HE compilation challenges. Figure 4.1 depicts where Porcupine fits in a HE compiler framework. In contrast to related work (see Section 4.8),we are the first to automate compiling and optimizing vectorized HE kernels, a crticial component towards compiling efficient HE programs.

In this chapter we propose Porcupine, a synthesizing compiler for HE. Users provide a reference implementation of their plaintext kernel and Porcupine synthesizes a vectorized HE kernel that performs the same computation. Internally, Porcupine models instruction noise, latency, behavior, and HE program semantics with Quill: a novel HE DSL. Quill enables Porcupine to reason about and search for HE kernels that are (verifiably) correct and minimizes the kernel's cost, i.e., latency and noise accumulation. With Porcupine and Quill, we develop a synthesis procedure that automates and optimizes the mapping and scheduling of plaintext kernels to HE instructions.

Porcupine uses syntax-guided synthesis [5] so that our synthesizer completes a sketch, or HE kernel template. We introduce a novel *local rotate* that treats ciphertext rotation as an input to HE add and multiply instructions rather

than an independent rotation instruction; this makes the synthesis search more tractable by limiting the space of possible programs. Furthermore, we develop several HE-specific optimizations including rotation restrictions for tree reductions and stencil computations, multi-step synthesis, and constraint optimizations to further improve synthesis run time (details in Section 4.6).

We evaluate Porcupine using a variety of image processing and linear algebra kernels. Baseline programs are hand-written and attempt to minimize multiplicative and logical depth, the current best practice for optimizing HE programs [8, 31, 84]. For small kernels, Porcupine is able to find the same optimized implementations as the hand-written baseline. On larger, more complex kernels, we show Porcupine's programs are up to 51% faster. Upon further analysis, we find that Porcupine can discover optimizations such as factorization and even application-specific optimizations involving separable filters. Our results demonstrate the efficacy and generality of our synthesis-based compilation approach and further motivates the benefits of automated reasoning in HE for both performance and productivity.

This chapter makes the following contributions:

1. We present Porcupine, a program synthesis-based compiler that automatically generates vectorized HE programs, and Quill, a DSL for HE. Porcupine includes a set of optimizations needed to effectively adopt program synthesis to target HE.

2. We evaluate Porcupine using nine kernels to demonstrate it can successfully translate plaintext specifications to correct HE-equivalent implementations. Porcupine achieves speedups of up to 51% (11% geometric mean) over hand-written baselines implemented with best-known practices. We note situations where optimal solutions cannot be found with existing techniques (i.e., logic depth minimization), further motivating automated reasoning-based solutions.

3. We develop a set of optimizations to improve Porcupine's synthesis time and compile larger programs. First, we develop a domain-specific local rotate that considers rotations as inputs to arithmetic instructions, narrowing the solutions space without compromising quality. We further restrict HE rotation patterns and propose a multi-step synthesis process.

4.2 HOMOMORPHIC ENCRYPTION BACKGROUND

This section provides a brief background on homomorphic encryption. We refer the reader to [23, 25, 26, 53, 59] for the more technical details of how HE works.

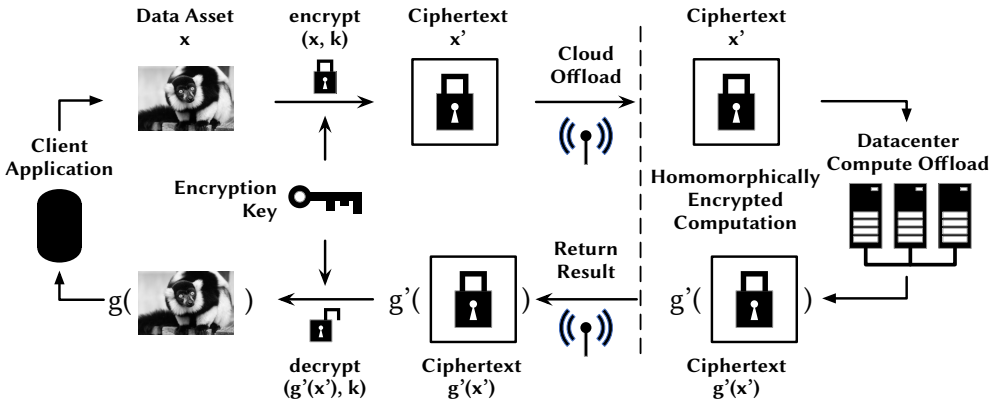


Figure 4.2: HE system for to an untrusted third party cloud. A plaintext data asset x is encrypted with a key k to generate ciphertext x' and transmitted to the cloud. The cloud service applies HE computation g' to the ciphertext *without* decrypting the data. The result $g'(x')$ is transmitted back to client where decryption yields the result $g(x)$.

4.2.1 *Homomorphic Encryption Basics*

Homomorphic encryption enables arbitrary computation over encrypted data or ciphertexts [58]. This enables secure computation offload where an untrusted third party, such as a cloud provider, performs computation over a client's private data without gaining access to it.

Figure 4.2 shows a canonical HE system for secure cloud compute. First, the client locally encrypts their data asset x using a private key k. The resulting ciphertext x' is then sent to the cloud where an HE function g' is applied to it. The output of the computation $g'(x')$ is then sent back to the client and decrypted using the same key k to reveal the plaintext output: $g(x)$. HE allows us to define a function g' that operates over ciphertext $x' = \text{encrypt}(x, k)$ such that:

$$\text{decrypt}(g'(x'), k) = g(x)$$

The private key k never leaves the client, ensuring the client's data asset is secure throughout the computation. Additionally, the client does not learn g, which could be a secret that the cloud wants to protect. Porcupine's goal is to synthesize g' given a definition of the kernel g.

This chapter focuses on the BFV cryptosystem, a specific HE scheme that targets integers [53]. In the remainder of this section, we provide an overview of the BFV scheme and focus on the vector programming model, instructions, and noise considerations it exposes. For a more technical description see [3, 53].

4.2.2 *BFV*

BFV is an HE scheme that operates over encrypted integers. In BFV, integers are encrypted into a ciphertext polynomial of degree N with integer coefficients that are modulo q. A key property of BFV is batching; this allows a vector of up to N integers to be encrypted in a single ciphertext with operations behaving in a SIMD manner.

For the most part, ciphertext polynomials behave as a vector of N slots with bitwidth q. N and q are BFV HE parameters set to provide a desired security level and computational depth, not the number of raw integers that are encrypted. Regardless of whether we encrypt a single integer or N integers in a ciphertext, a vector of N slots is allocated for security purposes. N is required to be a large power of two and is often in the tens of thousands, which makes batching crucial to efficiently utilizing ciphertext space.

INSTRUCTIONS. BFV provides three core ciphertext instructions that behave like element-wise SIMD instructions: SIMD add, SIMD multiply, and SIMD (slot) rotate. Additionally, BFV supports variants of add and multiply that operate on a ciphertext and plaintext instead of two ciphertexts.

Consider two vectors of integers $X = \{x_0, x_1, ..., x_{n-1}\}$ and $Y = \{y_0, y_1, ..., y_{n-1}\}$ with ciphertext representation X' and Y' respectively. SIMD add and multiply both perform element-wise operations over slots. SIMD add computes $\mathtt{add}(X', Y')$ such that $\mathrm{decrypt}(\mathtt{add}(X', Y')', k) = \{x_0 + y_0, x_1 + y_1, ..., x_{n-1} + y_{n-1}\}$, where k is the key used for encryption of X' and Y'. Similarly, the SIMD multiply instruction processes $\mathtt{mult}(X, Y)$ so that $\mathrm{decrypt}(g'(X', Y'), k) = \{x_0 \times y_0, x_1 \times y_1, ..., x_{n-1} \times y_{n-1}\}$. Note that the underlying operations that implement $\mathtt{add}(X', Y')$ and $\mathtt{mult}(X', Y')$ over the ciphertext representations are *not* simple vector addition or multiplication instructions.

ROTATE. Additionally, HE provides rotate instructions that circularly shift slots in a ciphertext by an integer amount (similar to bitwise rotations). Rotations occur in unison: given a rotation amount, all slots shift by the same amount in the same direction and the relative ordering of slots is preserved. For example, rotating a ciphertext $X' = \{x_0, x_1, x_2, ..., x_{n-1}\}$ by one element to the left returns $\{x_1, x_2, ..., x_{n-1}, x_0\}$.

Note the ciphertext is not a true vector, so slots cannot be directly indexed or re-ordered. Slot rotation is necessary to align slot values between vectors because add and multiply instructions are element-wise along the same slot lanes. For example, reductions that sum many elements within a ciphertext will need to rotate slots so that elements can be summed in one slot. Arbitrary shuffles also have to be implemented using rotates and multiplication with masks which can require many instructions and quickly become expensive to implement.

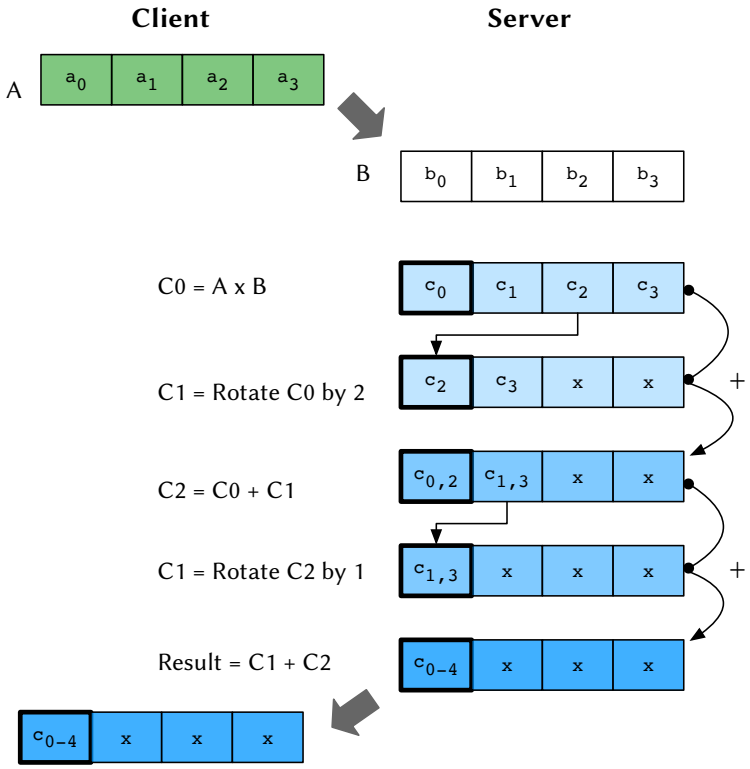


Figure 4.3: HE vectorized dot product implementation. Given an encrypted input from the client (A), the server performs an element-wise multiplication with server-local data (B). A reduction is performed using a combination of rotation and add instructions. The resulting ciphertext is then returned the client for decryption.

NOISE. During encryption ciphertexts are injected with random noise to prevent threats such as replay attacks [128]. During computation this noise grows. The ciphertext bitwidth q needs to be large enough to contain this noise growth or else the ciphertext becomes corrupted and upon decryption returns an random value (i.e., garbage value). However, larger values of q increase the memory footprint of ciphertext and requires more compute resource to perform the larger bitwidth arithmetic calculations that back HE instructions.

Specifically, add and rotate additively increase noise, and multiplication multiplicatively increases noise. Because multiplication dominates noise growth, the multiplicative depth of a program can be used as a guide to select q or as a minimization target.

4.3 HE COMPILATION CHALLENGES

Handwriting efficient HE kernels is a tedious and error-prone process as HE provides limited instructions, intra-ciphertext data movement must be done using vector rotation, and the noise budget adds additional sources of error. As a result, HE code is today is typically written by experts [50, 78, 109].

Porcupine's goal is to automate the generation of vectorized HE kernels to lower HE's high barrier-to-entry to non-experts as well as time-to-solution for experts. This section motivates the need for automated reasoning in HE compilers using a vectorized dot product (see Figure 4.3) as a running example.

4.3.1 *Data Packing*

To compute an HE dot product, a client sends an encrypted vector of elements to be computed with a server's vector; the encrypted result is then sent back to the client. A client could encrypt each element of the input vector into individual ciphertexts, but this uses only a single slot of each ciphertext vector, wasting the other slots. Another solution is to batch N independent tasks into a single ciphertext to amortize the cost of the ciphertext and HE program. However, HE vectors can hold tens of thousands of elements and most applications cannot rely on batching of this scale.

Instead, a client can pack the input data vector in a single ciphertext, as shown in Figure 4.3. In our example of a four element dot product, this requires only one ciphertext, not four. *Porcupine assumes kernels operate over packed inputs to efficiently utilize memory.*

4.3.2 *HE Computation*

One of the key challenges for building optimized HE kernels is breaking down scalar computation to efficiently use the limited HE instruction set. In ciphertext vectors, the relative ordering of packed data elements is fixed; thus, while element-wise SIMD addition and multiplication computation is trivial to implement, scalar-output calculations such as reductions require proper alignment of slots between ciphertext operands. The only way to align different slot indices between two ciphertexts is to explicitly rotate one of them such that the desired elements are aligned to the same slot.

Figure 4.3 illustrates how this is done for an HE dot product reduction operation using packed vectors. The client's and server's ciphertext operands are multiplied together and reduced to a single value. The multiplication operation is element-wise, so it can be implemented with a HE SIMD multiply operation. However, the summation within the vector must be performed by repeatedly rotating and adding ciphertexts together such that the intermediate operands are properly aligned to a slot in the vector (in this case the slot at index 0). The rotations and arithmetic operations are interleaved to take advantage of the SIMD parallelism and enable reduction to be computed with only two HE add operations for four elements.

For more complex kernels, simultaneously scheduling computations and vector rotations is non-trivial to implement efficiently. Arbitrary slot swaps or shuffles (e.g., instructions like `_mm_shuffle_epi32`) that change the relative or-

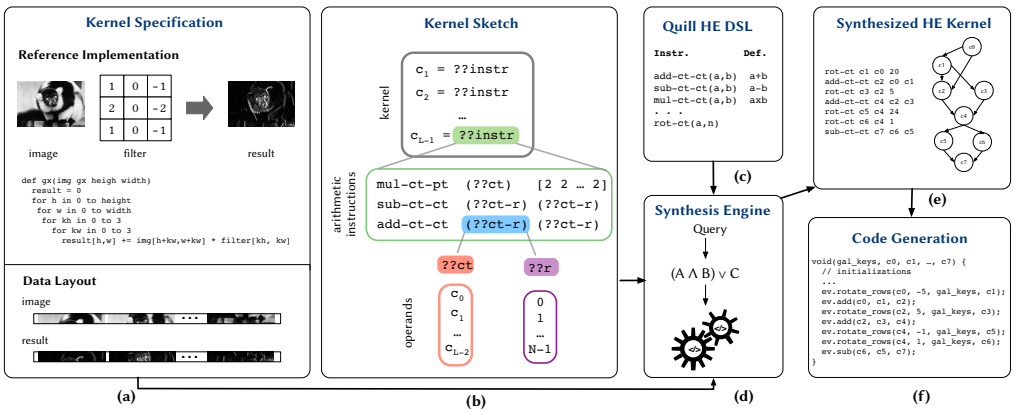


Figure 4.4: The Porcupine compiler. The user provides: (a) a kernel specification and (b) a kernel sketch with ?? denoting holes in the sketch. (c) The Quill DSL encodes the semantics of the HE instruction set and noise models. (d) Porcupine's synthesis engine completes the sketch and synthesizes a program that implements the specification using the Quill DSL. Porcupine uses an SMT solver to automatically solve the vectorization and scheduling challenges so that (e) the synthesized program is optimized. (f) The optimized HE kernel is consumed by code generation to target the SEAL library [119].

dering of elements in a vector are even more tedious to implement. While these arbitrary shuffles can be implemented in HE by multiplying with masks and rotation operations, this is undesirable since it requires dramatically increasing the multiplicative depth and hence noise budget requirements.

4.3.3 *Performance and Noise*

The vectorization challenges are further complicated by HE's compound-cost model that must consider both performance and noise. Performance and noise costs cannot be reasoned about independently; the performance cost must be aware of the noise growth since the noise budget parameter q defines the bitwidth precision of the underlying mathematical HE instruction implementations. Thus, a larger q increases the latency cost of each HE instruction. This means any sort of optimization objective for program synthesis will have to consider both noise and performance together.

4.4 PORCUPINE COMPILER AND SYNTHESIS FORMULATION

This section introduces the Porcupine compiler, Quill DSL, and the program synthesis formulation used to optimize HE kernels.

⟨*kernel*⟩ ::= (`list` ⟨*instr*⟩+) | ⟨*instr*⟩

⟨*instr*⟩ ::= ⟨*ct*⟩

⟨*pt*⟩ ::= ⟨*vector of integers*⟩

⟨*x*⟩ ::= ⟨*integer*⟩

⟨*ct*⟩ ::= (`add-ct-ct` ⟨*ct*⟩, ⟨*ct*⟩) | (`add-ct-pt` ⟨*ct*⟩, ⟨*pt*⟩)
| (`sub-ct-ct` ⟨*ct*⟩, ⟨*ct*⟩) | (`sub-ct-pt` ⟨*ct*⟩, ⟨*pt*⟩)
| (`mul-ct-ct` ⟨*ct*⟩, ⟨*ct*⟩) | (`mul-ct-pt` ⟨*ct*⟩, ⟨*pt*⟩)
| (`rot-ct` ⟨*ct*⟩, ⟨*x*⟩)

Figure 4.5: The Quill DSL. A Quill kernel is made up of a list instructions that produce a final ciphertext (ct). Each instruction produces a ciphertext and takes as input at least one ciphertext and possibly a plaintext (pt) or integer.

4.4.1 *Compiler Overview*

Porcupine is a program synthesis-based compiler that searches for HE kernels rather than relying on traditional rewrite rules. By searching for programs, Porcupine can discover optimizations that are potentially difficult to identify by hand. At a high level, Porcupine takes a kernel specification (Figure 4.4a) and sketch (Figure 4.4b) as input, and outputs an optimized HE kernel (Figure 4.4f). Section 4.4.2 defines our Quill DSL (Figure 4.4c) which is used to model the noise and latency semantics of each HE instruction. Section 4.4.3 defines what composes the specification that Porcupine as input (Figure 4.4a). Section 4.4.4 explains our sketch formulation and design decisions behind them (Figure 4.4b). Section 4.5 details our synthesis engine [5] which takes the specification, sketch, and HE kernel, and emits a synthesized HE kernel (Figure 4.4).

4.4.2 *Quill: A DSL for HE*

The Quill DSL serves as a specification for HE programs and models HE ciphertext, instructions, and their latency-noise behavior. This enables Porcupine to reason about HE instruction behavior as well as verify correctness. When synthesizing an HE kernel, Porcupine first synthesizes a Quill kernel which is then translated into code for an HE library such as SEAL. Quill currently supports BFV [53] HE, however the techniques are general and can be extended to other ring-based HE schemes, e.g., BGV [26] and CKKS [36].

Quill is used to describe straight-line HE programs that manipulate state initially defined by input vectors (either ciphertext and plaintext) and returns a ciphertext. Figure 4.5 defines Quill's grammar. Quill programs are behavioral models and not true HE programs. The ciphertext operands are implemented as unencrypted vectors that can only be manipulated according to HE instruction

Table 4.1: Quill instructions and their affect on the data (denoted by .data) and multiplicative depth (denoted by .depth) of the resulting ciphertext.

Instruction	Computation	Multiplicative depth
Add(ct_x, ct_y) $\rightarrow$ ct_z	$ct_x.data + ct_y.data$	$max(ct_x.depth, ct_y.depth)$
Add(ct, pt) $\rightarrow$ ct_z	$ct.data + pt.data$	$ct.depth$
Subtract(ct_x, ct_y) $\rightarrow$ ct_z	$ct_x.data - ct_y.data$	$max(ct_x.depth + ct_y.depth)$
Subtract(ct, pt) $\rightarrow$ ct_z	$ct.data - pt.data$	$ct.depth$
Multiply(ct_x, ct_y) $\rightarrow$ ct_z	$ct_x.data \times ct_y.data$	$max(ct_x.depth, ct_y.depth) + 1$
Multiply(ct, pt) $\rightarrow$ ct_z	$ct.data \times pt.data$	$ct_x.depth + 1$
Rotate(ct, x) $\rightarrow$ ct_z	$ct.data[i] \leftarrow$ $ct.data[(i+x) mod N]$	$ct.depth$

rules, which are captured by Quill's semantics. This provides the benefit that we can compile code without considering the implementation details of true HE.

STATE IN QUILL In a Quill program, state is defined by plaintext and ciphertext vectors. All ciphertexts are associated with metadata that tracks each operand's multiplicative depth, which models noise accumulation. An input or fresh ciphertext has zero multiplicative depth and increases each time a multiplication is performed. We track only multiplicative depth as it simplifies the objective of noise minimization without sacrificing accuracy as other instructions - add and rotate - contribute relatively negligible noise.

THE QUILL INSTRUCTIONS. Quill supports a SIMD instruction set with a one-to-one mapping to BFV HE instructions. Table 4.1 describes each instruction's type signature and how they transform state. Instructions include addition, multiplication, and rotations of ciphertext instructions as well as variants that operate on ciphertext-plaintext operands, e.g., multiplication between a ciphertext and plaintext. Each instruction is associated with a latency derived by profiling its corresponding HE instruction with the SEAL HE library [119].

4.4.3 *Kernel Specification*

A *specification* completely describes a target kernel's functional behavior, i.e., it defines what the synthesized HE kernel must compute. In Porcupine, a specification comprises a *reference implementation* of the computation (in plaintext) and *vector data layout* that inputs and outputs must adhere to.

REFERENCE IMPLEMENTATION. Reference implementations are programs written in Racket [105] that define the plaintext computation. We later use

Rosette [131] to automatically lift the Racket program to a symbolic input-output expression that defines the program's behavior. An example reference implementation for the G_x kernel is shown below. The code takes as input a 2D gray-scale image and calculates the x-gradient by summing and weighting neighboring pixels according to a 3×3 filter.

```
(define (Gx img height width filter):
    for h in 0 to height
        for w in 0 to width:
            for kh in 0 to 3:
                for kw in 0 to 3:
                    result[h,w] += img[h+kw, w+kw] *
                                   filter[kh, kw]
```

Porcupine uses the reference implementation to verify synthesized ones are correct; the quality of the reference program does not impact synthesized code quality. As a result, users can focus on writing correct code without the burden of performance tuning.

To work correctly, the implementation must describe computation that is directly implementable in HE. Implementations cannot contain data dependent control flow such as conditional statements or loops that depend on a ciphertext, since we cannot see the values of encrypted data. This is a limitation of HE, and while it is possible to approximate this behavior, e.g., using a polynomial function, this is beyond the scope of our work.

DATA LAYOUT. A data layout defines how the inputs and outputs are packed into ciphertext and plaintext vectors. In the G_x example, we pack the input and output image into one ciphertext as a flattened row-order vector with zero-padding around the borders. The data layout is an input to the synthesizer only, and the reference implementation does not need to consider it. Together, the reference implementation and data layout define the inputs and outputs to the HE program, and Porcupine will synthesize an HE program that achieves that transformation.

4.4.4 *Sketch*

The user also provides a *sketch*, which is a template for describing partial Quill kernels that are used to guide the synthesis engine towards a solution. It allows the user to articulate required features of the HE kernel to the synthesis engine while leaving other components unspecified as *holes*, indicated by `??`, for the engine to fill in. The synthesizer then completes the sketch by filling in the holes to match the functionality of the reference implementation. We introduce a *local rotate* sketch to help the user convey hints about ciphertext rotations. An example of a local rotate sketch for the G_x kernel is shown below:

```
; Program sketch of L components
; ct0 is a ciphertext input
(define (Gx-Sketch ct0 L)
    ; choose an existing ciphertext
    (define (??ct)  (choose* ct))
```

```
; choose a rotation amount in range (0,N)
(define (??r)
  (apply choose* (range 0 N)))
; choose a rotation of an existing ciphertext
(define (??ct-r)
    (rot-ct ??ct ??r))
; choose an opcode with operand holes
(for/list i = 1 to L
    (choose*
        (add-ct-ct (??ct-r) (??ct-r))
        (sub-ct-ct (??ct-r) (??ct-r))
        (mul-ct-pt (??ct) [2 2 ... 2])))
```

The sketch describes a kernel template that takes as input a single ciphertext (encrypted image) and applies a kernel composed of `L` *components* or arithmetic instructions. In this example the components are: add two ciphertexts, subtract two ciphertexts or multiply a ciphertext by a plaintext of 2s. Each component contains holes for their instruction dependent operands. Specifically, `??ct` is ciphertext hole that can be filled with the ciphertext input or a ciphertexts generated by previous components. `??ct-r` is a ciphertext-rotation that introduces two holes: a ciphertext hole and a rotation hole. The ciphertext hole can be filled with any previously generated ciphertexts and the rotation hole indicates the ciphertext can be rotated by any legal amount (1 to $N-1$) or not at all. Ciphertext-rotation holes indicate the kernel performs a reduction operation over elements and requires rotation to align vector slots.

Writing sketches of this style is relatively simple, with most sketches taking only a few minutes to write and debug. The arithmetic instructions can be extracted from the specification. In this case add, subtract, and multiplication by 2 were used in the reference implementation. The set of arithmetic instructions is treated like a multiset of multiplicity L, and the synthesizer will determine which instructions and how many are needed. In other words, the sketch does not have to be exact as the synthesizer can choose to ignore instructions; this once again eases the burden on the user. Additionally, the user must specify whether instruction operands should be ciphertexts or ciphertext-rotations, and what rotations are allowed. As a fall back, all ciphertext holes can be made ciphertext-rotation holes; however, this will increase solving time as the sketch describes a larger space of programs. Furthermore, the effort of sketch writing can potentially be amortized by re-using or tweaking a sketch from a kernel with similar compute patterns. For example, when writing a sketch for a different 2D convolution, we could start from this `Gx-sketch` and either re-use it or change the plaintext constants.

A key feature of our sketches is that we treat rotation as an input to arithmetic instructions rather than a component of the sketch. This is because rotations are only useful when an arithmetic instruction needs to re-align operands; in isolation, rotations do not perform meaningful computation. This excludes programs that contain nested rotations since rotations can be combined. For

instance, we disallow (rot (rot c0 1) 2) since this can be more succinctly expressed as (rot c0 3).

The sketches must describe loop-free programs so that Quill can interpret them. Porcupine requires sketches to be parameterized by the number of components in the program. Porcupine first explores small (in terms of L programs and iteratively explores larger programs by incrementing L until a solution is found.

SOLUTION. A *solution* is a completed sketch that matches the reference implementation. Porcupine's synthesis engine generates solutions by filling instruction and operand holes such that the resulting program satisfies the specification and optimizes the objective functions (minimize instruction count and noise). The solution Porcupine synthesizes for the above example uses three arithmetic instructions and four rotations [1]:

```
Ciphertext c1 = (add-ct-ct (rot-ct c0 -5) c0)
Ciphertext c2 = (add-ct-ct (rot-ct c1 5) c1)
Ciphertext c3 = (sub-ct-ct (rot-ct c2 1)
                           (rot-ct c2 -1))
```

4.5 SYNTHESIS ENGINE

This section describes how Porcupine's synthesis engine (see Algorithm 1) searches the program space (described by our local rotate) to find an optimized HE solution that satisfies the kernel specification. Porcupine's synthesis engine operates by first synthesizing an initial solution. It then optimizes the solution by iteratively searching for better solutions until either the best program in the sketch is found or a user-specified time out is reached.

Porcupine's synthesis engine is a counter-example guided inductive synthesis (CEGIS) loop [73, 126]. It leverages Rosette's built-in support for translating synthesis and verification queries to constraints that are solved by an SMT solver.

4.5.1 *Synthesizing an Initial Solution*

The first step in Porcupine's synthesis procedure is to synthesize an initial program that satisfies the user's specification. In particular, Porcupine first attempts to complete a sketch $sketch_L$ that encodes programs using L components. Specifically, Porcupine searches for a solution sol_0 contained in $sketch_L$ that minimizes L and satisfies the specification for all inputs. We follow a synthesis procedure similar to those proposed in [64, 73, 126], and avoid directly solving the above query because it contains a universal quantifier over inputs. Instead, we synthesize a solution that is correct for one random input then

1 Rotation amounts are adjusted to be relative in example.

verify it is correct for all inputs, applying feedback to the synthesis query if verification fails.

SYNTHESIZE. The engine starts by generating a concrete input-output example, (x_0, y_0), by evaluating the specification using a randomly generated input, x_0 (line 6). The engine attempts to synthesize a program that transforms x_0 into y_0 by completing the sketch and finding a binding for the L arithmetic instructions and operand holes (line 10). We generate a synthesis query expressing $solve(sketch_L(x_0) = y_0)$, which is then compiled to constraints and solved by an SMT solver.

VERIFY. If successful, the synthesis query described above returns a program that satisfies the input specification for the input x_0, but not necessarily for all possible inputs. To guarantee that the solution is correct, Porcupine verifies the solution matches the specification for all inputs. Porcupine leverages Rosette's symbolic evaluation and verification capabilities to solve this query. First, a formal pre-condition and post-condition is lifted from reference specification with symbolic execution, capturing the kernel's output for a bounded set of inputs as a symbolic input-output pair $(\hat{x}, \hat{y})$. Rosette then solves the verification query $verfiy(sol(\hat{x}) = spec(\hat{x}))$.

RETRY WITH COUNTER-EXAMPLE. If verification fails, it returns a counter-example, (x_1, y_1), that causes the synthesized kernel to disagree with the specification. Porcupine then makes another attempt to synthesize a program; this time trying to satisfy both the initial example and counter-example. This process repeats until Porcupine finds a correct solution.

If the engine cannot find a solution, indicated when the solver returns `unsat` for any synthesis query, the engine concludes that for the given sketch, a program that implements the specification with L components does not exist. The engine tries again with a larger sketch $sketch_{L+1}$ that contains one more component and this process repeats until a solution is found. By exploring smaller sketches first, our algorithm ensures that the solution using the smallest number of components is found first.

4.5.2 *Optimization*

Once an initial solution is found, Porcupine's synthesis engine attempts to improve performance by searching for better programs contained in the sketch. Programs are ranked according to a cost function that Porcupine attempts to minimize.

COST FUNCTION. Porcupine uses a cost function that multiplies the estimated latency and multiplicative depth of the program: $cost(p) = latency(p) \times$

Algorithm 1 Synthesis engine

```
1:  Input
2:    spec      Kernel reference program
3:    sketch    Partial HE program
4:  Synthesize first solution
5:  function SYNTHESIZE
6:    y_0 ← spec(x_0)                          ▷ Random input-output example
7:    ŷ = spec(x̂)                              ▷ Symbolic input-output
8:    examples = [(x_0, y_0)]
9:    while true do
10:     sol ← solve(sketch s.t. y=sketch(x))
11:     if sol is unsat then
12:       return False                          ▷ Sketch too restrictive
13:     cex ← verify(ŷ = solution(x̂))
14:     if cex = unsat then
15:       return sol
16:     (x, y) ← extract(cex)                   ▷ Get counterexample
17:     examples.append((x, y))
18: Minimize cost
19: function OPTIMIZE
20:   sol ← synthesize()
21:   c′ ← cost(sketch)
22:   sol′ ← sol
23:   while sol′ is sat do
24:     c ← cost(sol), sol ← sol′
25:     sol′ ← solve(sketch s.t. y=sketch(x) & c′ < c)
26:     <verify sol′ and add cex if needed>
27:   return sol
```

$(1 + \mathtt{mdepth}(p))$. We include multiplicative depth to penalize high-noise programs, which can lead to larger HE parameters and lower performance.

COST MINIMIZATION. Once a solution $\mathtt{sol}_0$ with cost $\mathtt{cost}_0$ is found, we iteratively search for a new program with lower cost (line 19), as described in [122]. Porcupine does this by re-issuing the successful synthesize query with an additional constraint that ensures a new solution $\mathtt{sol}_1$, has lower cost: $\mathtt{cost}_1 < \mathtt{cost}_0$ (line 25). This process repeats until the solver proves there is no lower cost solution and it has found the best solution or the compile time exceeds the user-specified time out. The initial solution is only used to provide an upper-bound on cost and is not used during the optimization synthesis queries. This forces the engine to consider completely new programs with different instruction mixes and orderings. In practice, we find that initial solutions perform well given the savings in compile time (see Section 4.7.5 for discussion).

4.5.3 *Code Generation*

The synthesis engine outputs a HE kernel described in Quill and Porcupine then translates the Quill program into a SEAL program [119]. SEAL is a HE library that implements the BFV scheme. Quill instructions map directly to SEAL instructions, so this translation is simple, but the code generation handles a few post-processing steps. For example, Porcupine inserts special *relinearization* instructions after each ciphertext-ciphertext multiplication. Relinearization does not affect the results of the HE program but is necessary to handle ciphertext multiply complexities.

During code generation we also map fixed rotations to variable rotations to support variable sized inputs for stencil kernels. In stencil kernels the amount of computation per individual element depends on the window size, not the size of the input. Therefore, we can translate fixed-sized rotations to variable sized rotations to support variable sized images. For example, the final code generated G_x kernel is shown below, where `h` and `w` are the height and width of the ciphertext image in `c0`. While the synthesis step assumed a small fixed-size 4x4 image, we can translate the fixed rotations into variable rotations to support generic sized images.

```
Ciphertext gx(Ciphertext c0, int h, int w)
    Ciphertext c1 = rotate(c0, w)
    Ciphertext c2 = add(c0, c1)
    Ciphertext c3 = rotate(c2, -w)
    Ciphertext c4 = add(c2, c3)
    Ciphertext c5 = rotate(c4, 1)
    Ciphertext c6 = rotate(c4, -1)
    return sub(c5, c6)
```

4.6 SYNTHESIS FORMULATION OPTIMIZATIONS

Scaling Porcupine to handle larger kernels requires optimizing the synthesis formulation. Since the search space grows super exponentially, it quickly becomes intractable—a five instruction HE program can have millions of candidate programs. This section describes optimizations developed to scale up our formulation and their impact on the results.

4.6.1 *Rotation Restrictions*

HE Rotation instructions are used to align different vector slots within a ciphertext to perform computation such as reductions. Ciphertext slots can be rotated by up to N, the size of the ciphertext vector, which introduces a large number of possible rotations for the synthesizer to select from. In practice, we observe that of all possible rotations only a few patterns are ever used. For example, in our G_x kernel each output element only depends on its neighbors in the 3×3 window, implying rotations that align input elements from outside this window

are not necessary. By restricting rotations, we can scale up the synthesis process by pruning away irrelevant potential programs.

To optimize for this, we introduce two types of rotation restrictions for tree reductions and sliding windows. For sliding window kernels, which are commonly used in image processing, we use the restriction described above to restrict rotation holes `??rot` to align elements covered by the window. The tree reduction restricts rotations to powers of two and is used for kernels that implement an internal reduction within the ciphertext. For example, in a dot product elements in the vector are summed to produce one value. Restricting the rotations to powers of two constrains the output programs to perform the summation as a reduction tree.

4.6.2 *Constraint Optimizations*

We also apply a number of common constraint optimization techniques to improve synthesis speed and scalability. We employ symmetry breaking to reduce the search space for add, multiply, and rotate. For example, the programs $a + b$ and $b + a$ are functionally equivalent but appear as two unique solutions to a solver. Restricting operands to occur in increasing order eliminates redundant candidate solutions and improves synthesis speed. For rotations we impose symmetry breaking by forcing only left rotations, since a left rotation by x is equivalent to a right rotation by $N - x$. We also enforce solutions use static single assignment to instill an ordering and break symmetries between programs that are functionally equivalent but write to different destination ciphertexts.

Our synthesis formulation also uses restricted bitwidth instead of full precision bit vectors to reduce the number of underlying variables the solver needs to reason about. Ordinarily, the number of solver variables scales linearly with bitwidth; however, we do not need the bit accurate behavior, only the operator functionality, so this optimization does not affect correctness of the solution.

4.6.3 *Multi-step Synthesis*

One of the limitations of program synthesis is its inability to scale to large kernels [66]. With the above optimizations, Porcupine scales to roughly 10-12 instructions, but beyond that the program space becomes intractable to search. Many applications in image processing, neural networks, and machine learning have natural break points. For instance, an image processing pipeline may have cascaded stencil computations for sharpening, blurring, and edge detection which have natural boundaries. To scale beyond the limitations of program synthesis, we leverage these natural breakpoints to partition larger programs into segments and synthesize them independently. In Section 4.7, we show how this partitioning into a multistep synthesis problem can allow Porcupine to scale to longer kernels.

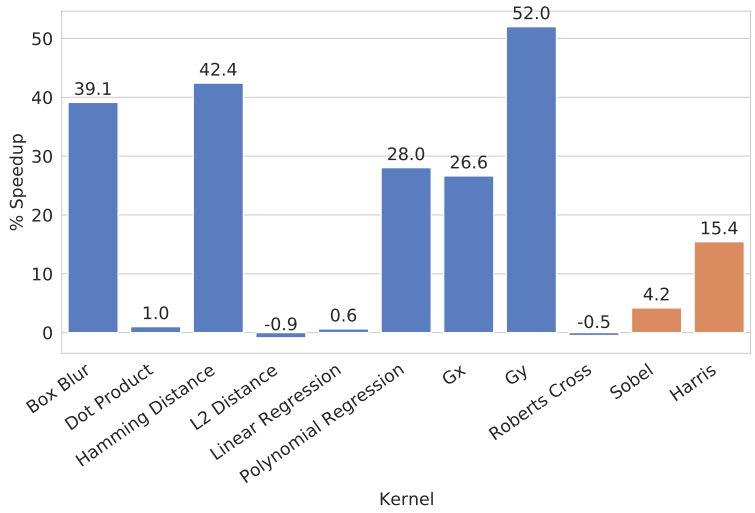


Figure 4.6: Speedup of Porcupine synthesized kernels compared to the baseline, results are averaged over 50 runs. Kernels in blue are directly synthesized while kernels in orange use multi-step synthesis.

4.7 EVALUATION

This section evaluates Porcupine's synthesized programs and compares them against expert-optimized baselines (see Section 4.7.2). We also report how long Porcupine takes to synthesize kernels (see Section 4.7.5). We find that Porcupine is able to synthesize a variety of kernels that are at least as good or better than an expert-written version, and in most cases can synthesize a kernel in a few minutes.

4.7.1 *Methodology*

Porcupine is implemented with Rosette v3.1 [131], and configured to use Boolector [27] as its backend SMT solver. Synthesized kernels are compiled down to SEAL v3.5's BFV library [119]. When running Porcupine's kernels, security parameters are set to guarantee a 128-bit security level; both baseline and synthesized kernels use the same settings. All experiments are conducted on a 3.7 GHz Intel Xeon W-2135 CPU with 64 GB of memory.

WORKLOADS. We evaluate Porcupine using common kernels found in linear algebra, machine learning, and image processing listed in Table 4.3. Since there is no standardized benchmark for compiling HE kernels, we attempt to be as diverse and representative in our selection as possible. For example, dot product, L2 distance, and linear and polynomial regression kernels are building blocks of machine learning applications, while the x/y-gradient (G_x/G_y) and Roberts Cross kernels are used in image processing applications.

Table 4.2: A comparison of instruction count, multiplicative depth (M. Depth), and logical depth (L. Depth) of synthesized and baseline kernels.

Kernel	Synthesized/Baseline		
	Instr.	M. Depth	L. Depth
Box Blur	4/6	0/0	4/3
Dot Product	7/7	1/1	7/7
Hamming Distance	9/13	1/1	9/9
L2 Distance	7/7	1/1	7/7
Linear Regression	4/4	1/1	4/4
Polynomial Regression	7/9	2/2	5/5
Gx	7/12	0/0	4/4
Gy	7/12	0/0	4/4
Roberts Cross	10/10	1/1	5/5
Sobel	19/25	1/1	9/7
Harris	43/59	3/3	17/14

Kernels are modified to omit operations not directly supported by HE. For instance, the canonical L2 distance kernel uses a square root, but many applications (e.g., k-nearest neighbors) can use squared distance with negligible effect on accuracy [92]. Finally, because BFV cannot implement data-dependent branches or conditionals, applications that require these operations are calculated up to a branch. For example, our Harris corner detector implementation returns an image of response values that the user must decrypt and apply a threshold over to detect the corners.

BASELINES. We compare Porcupine's code quality against an expert's hand-written implementation that seeks to minimize multiplicative, then logical depth. Minimizing multiplicative depth was chosen to reflect the state-of-the-art solution that was recently proposed for optimizing HE kernels under Boolean HE schemes [84]. The paper suggests that optimizing multiplicative depth also minimizes noise, as fewer successive operations compound less noise between any input-output. Since some of our baseline kernels require few or no multiplications, the baselines further minimize noise growth by minimizing logical depth after multiplicative depth. To minimize depth, these programs attempt to perform as much computation as possible in early levels of the program and implement all reductions as balanced trees. In addition, all our baseline implementations use packed inputs (i.e., are not scalar implementations) to minimize latency.

4.7.2 *Synthesized Kernel Quality*

To understand the quality of Porcupine's synthesized programs, we compare instruction count, multiplicative depth, logical depth, and run time against the hand-optimized baseline. We report run time speedups in Figure 4.6, with all times averaged over 50 independent runs and instruction counts in Table 4.2.

The results show that Porcupine's kernels have *similar or better performance* compared to the hand-written baselines. For some kernels such as dot product, L2 distance, and Roberts Cross, Porcupine generates roughly the same kernel as the hand-written implementation. The synthesized and baseline implementations may have different orderings of independent instructions, resulting in small performance differences.

For more complex kernels (G_x, G_y), polynomial regression, and box blur), we observe Porcupine's programs have notably better run times, up to 51% and use fewer instructions. Our speedups are a result of Porcupine being able to identify different types of optimizations. For example, our synthesized polynomial regression kernel found an algebraic optimization that factored out a multiplication similar to $ax^2 + bx = (ax + b)x$, resulting in a kernel that used 7 instructions instead of 9 and was 27% faster than the baseline. We analyze more of these optimizations in Section 4.7.3.

For these kernels, each handwritten baseline took on the order of a few hours to a day to implement, debug, and verify; for a non-expert unfamiliar with HE and SEAL, this would take much longer. The results show that Porcupine can effectively automate the tedious, time-consuming task of handwriting these kernels without sacrificing quality.

MULTI-STEP SYNTHESIS EVALUATION. We also used Porcupine's synthesized kernels to compile larger HE applications. Specifically, Porcupine's G_x and G_y kernels are used to implement the Sobel operator, and G_x, G_y, and box blur kernels were used to implement the Harris corner detector, shown in orange in Figure 4.6. By leveraging Porcupine synthesized kernels, our Sobel operator and Harris corner detector were 6% and 13% faster than the baseline, and used 10 and 16 fewer instructions respectively. These results show that we can speedup larger applications by synthesizing the core computational kernels these applications rely on.

4.7.3 *Analysis of Synthesized Kernels*

We now analyze the synthesized and baseline implementations of the box blur and G_x kernels to demonstrate the trade-offs explored by Porcupine. Figure 4.7 compares Porcupine's and the baseline's box blur. The baseline implements this kernel in six instructions with three levels of computation. In the first level, elements are aligned in the window with rotations and then summed in a

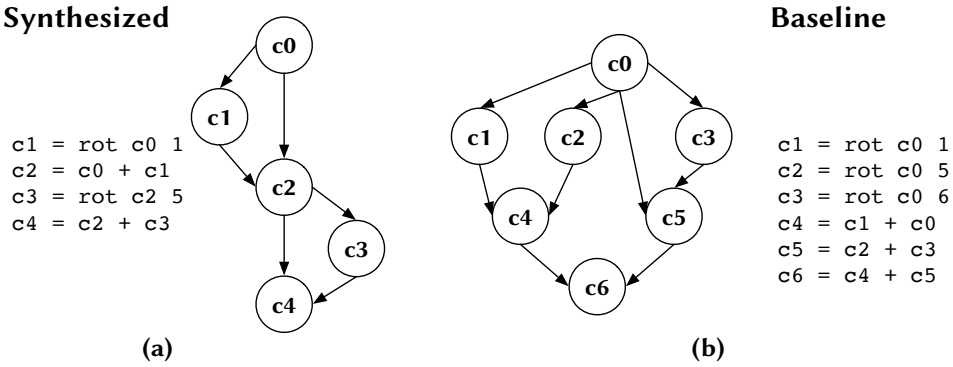


Figure 4.7: HE kernels for box blur. (a) Synthesized kernel with minimal number of instruction (b) Hand-optimized minimal depth kernel. Porcupine achieves a much higher performing kernel by separating kernels and use fewer instructions which, even though the logical depth increases, results in a 39% speedup.

reduction tree. Porcupine's synthesized kernel uses four instructions with five levels; decomposing the 2D convolution into two 1D convolutions to perform the same computation with fewer instructions. Furthermore, despite having a greater logical depth, the synthesized solution consumes the same amount of noise as the baseline. By focusing on minimizing logical depth, the baseline misses the separable kernel optimization because it was not the minimum depth solution.

We observe similar results for the G_x kernel and show the synthesized and baseline programs in Figure 4.8. The depth-optimized baseline applies the same strategy as the box blur kernel, first aligning elements in the sliding window then combining them in a balanced reduction tree. The G_x kernel weights some of the neighbor elements by two, and the baseline substitutes the multiplication with a cheaper addition (operand c11 in Figure 4.8b). The synthesized G_x kernel has a very different program structure from the baseline. Porcupine discovers the filter is separable and decomposes the kernel into two 1D filters, requiring a different set of rotations and schedule to implement correctly as depicted in Figure 4.9. Porcupine's synthesized solutions automatically also substitutes the multiplication by two with an addition which is performed at c4 (see Figure 4.9) in parallel with other additions.

While minimizing for logical depth is a good guideline for minimizing noise in scalar HE programs, our results show it is not guaranteed to find the optimal implementations for vector HE constructions, like BFV, and can leave significant unrealized performance (e.g., up to 51% for G_y). Because Porcupine searches for vectorized implementations, and tracks program latency and multiplicative depth it can outperform the heuristic for programs with more complex dependencies.

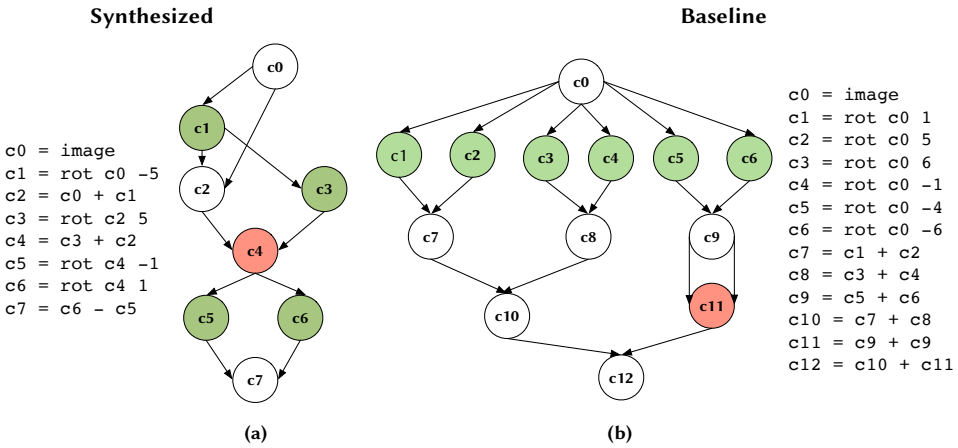


Figure 4.8: (a) Synthesized and (b) baseline G_x kernel. The synthesized kernel uses 7 instructions while the baseline uses 12 instructions. The synthesized kernel optimizes the computation to separate the 2D convolution into two 1D convolutions and interleaves rotation and computation. Ciphertexts generated by rotations are marked in green and the ciphertext where multiplication by 2 is implemented with an addition is in red.

4.7.4 *Handling Instructions not in the Instruction Set.*

Porcupine can only synthesize kernels that exactly obey the specification, and cannot synthesize an approximate kernel. This means that kernels requiring an operation such as a square-root cannot be synthesized because we cannot calculate square-roots with adds and multiplies. However, for instructions that can be exactly implemented with adds and multiplies, Porcupine can discover these translations as well as optimize them according to HE's cost model.

An example of a kernel that required instructions not directly supported is the Hamming distance kernel. The Hamming distance kernel takes as input two vectors of binary values and returns the number of elements that did not match. In plaintext computation this is easily expressed as $\sum_{i=0}^{N} a_i \oplus b_i$ where bitwise-XOR is used to efficiently determine if the slots a_i and b_i do not match. Since BFV does not directly support bitwise values, we left Porcupine to discover how to implement bitwise-XOR with multiplication and additions, and it returned the following result below, where N is the length of the binary vectors:

```
Plaintext p0(N, 2)
Ciphertext c1 = add(a, b)
Ciphertext c2 = sub(p0, c1)
Ciphertext c3 = multiply(c1, c2)
```

From analyzing works in stochastic computing [2], we found XOR could be implemented as $(1-a) = b + (1-b)a$, and requires two ciphertext-ciphertext multiplies. Distributing the multiplies results in a slightly more efficient $a + b - 2ab$, which still requires two multiplications but one is a more efficient ciphertext-plaintext multiplication. However, the result Porcupine discovers uses

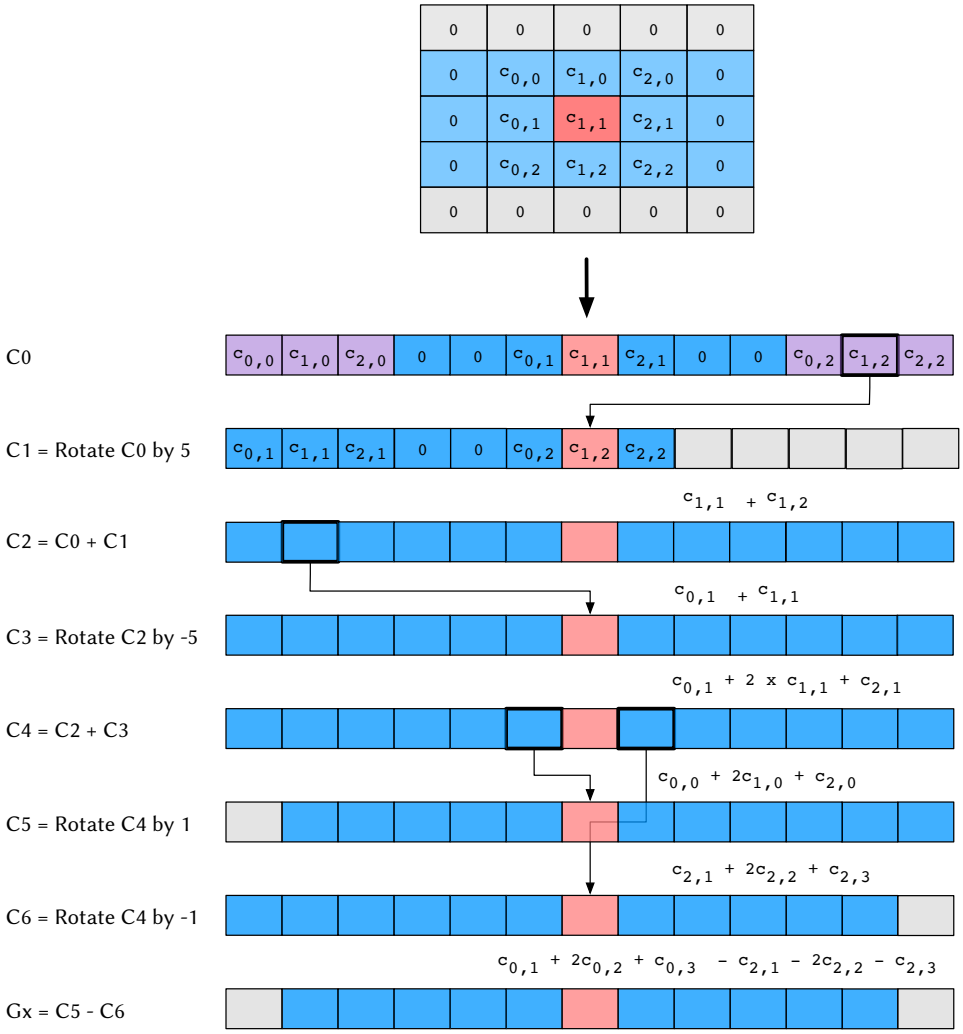


Figure 4.9: Porcupine optimized G_x kernel. An optimized implementation packs the entire image into one ciphertext and schedules computation with rotations. Purple slots contain elements that are used to compute the final red slot. The value contained in the red slot are tracked on the right hand side.

Table 4.3: Synthesis time and number of examples used by Porcupine. Initial time is the time to synthesize a solution and total time includes time spent optimizing. Reported values come from the median of three runs.

Kernel	Examples	Initial Time (s)	Total Time (s)
Box Blur	1	1.99	9.88
Dot Product	2	1.27	15.16
Hamming Distance	10	38.98	77.21
L2 Distance	2	27.57	114.28
Linear Regression	2	0.50	0.69
Polynomial Regression	2	24.59	47.88
Gx	1	14.87	70.08
Gy	1	9.74	49.52
Roberts Cross	1	212.52	609.64

only one multiplication and is far more efficient for HE: $(2 - a + b) \times (a + b)$. In stochastic computing multiplications are cheap and additions are expensive, so the implementation $a + b - 2ab$ is efficient in stochastic computing, but not optimal for HE.

4.7.5 *Synthesis Analysis*

SYNTHESIS TIME. Table 4.3 reports the median time it took to synthesize each kernel over three runs. We report how long it took to find an initial solution and the cumulative time it took to find an optimized solution. For most of the kernels we were able to synthesize an initial solution in under 30 seconds and synthesize an optimized solution under 2 minutes. The Roberts Cross kernel required more time, taking over 2 minutes to synthesize an initial solution and in total to 27 minutes to optimize. This is because the Roberts Cross kernel required a sketch with 10 instructions, which took longer to search over. Additionally, the optimization phase of the synthesis engine must prove it found the best solution contained in the sketch, requiring the SMT solver explore the entire search space.

In terms of input-output examples required by Porcupine during the synthesis process, we typically only require one example to synthesize a solution; however, for some kernels such as Hamming distance we required up to 10 input-output examples be generated during synthesis. We find kernels that produce a single-valued output, like Hamming distance, require more examples than kernels that produce a vector output (e.g., image processing kernels). This is because the synthesis engine can find many input-dependent (but not general) programs.

COST TRAJECTORY. Figure 4.10 reports the cost of the initial and final solutions found by Porcupine. For some kernels, the initial and first solution Porcupine finds are the same. This indicates that there was only one correct solution in the minimum L-sized sketch, or that Porcupine found the best solution on the first try. The time between Porcupine reporting the initial and final solution is spent proving that it found the best solution in the sketch. After the initial solution is found, users can terminate Porcupine early to shorten compile times. While this does not guarantee the best solution was found, it will minimize arithmetic instructions.

ANALYSIS OF LOCAL ROTATE SKETCHES. Our local rotate sketches treat rotations as operands instead of components. We could have alternatively required users explicitly add rotations to the list of components supplied in the sketch (which we refer to as *explicit rotation* sketches). However, explicit rotation sketches describe a larger space of programs that includes the space described by our local rotate sketches.

In small kernels, e.g., box blur, the synthesis time using local rotate sketches was slower than the explicit rotation sketch; the explicit rotation sketch took only 3 seconds to synthesize verses 10 seconds when using a local rotate sketch. However, when we synthesize larger programs the explicit rotation sketch scales poorly. Using the explicit rotation sketch, synthesizing the G_x kernel took over 400 seconds to find an initial solution then over 30 minutes total when using the explicit rotation sketch. On the other hand, the local rotate sketches found the same solution in about 70 seconds, showing that local rotate does improve synthesis scalability and search time.

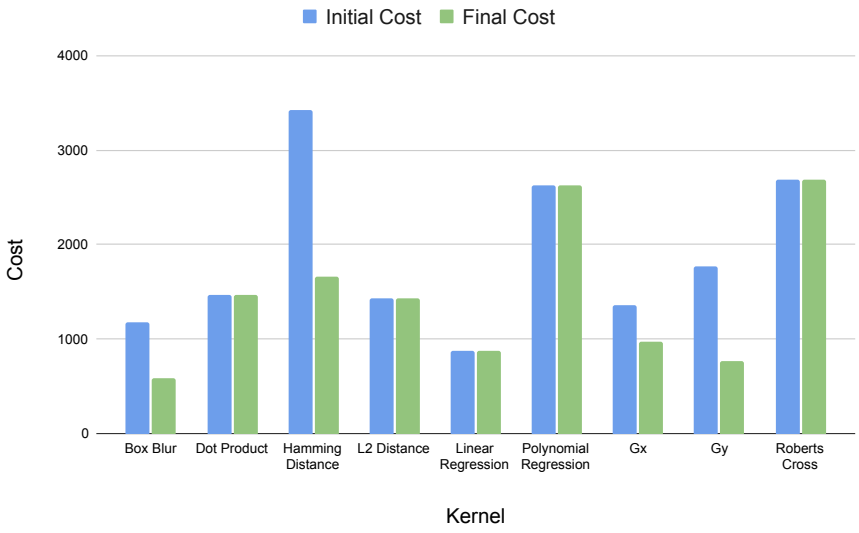


Figure 4.10: Synthesized cost of the initial kernel and final optimized kernel Porcupine discovers.

4.8 RELATED WORK

4.8.1 *Compilers for Homomorphic Encryption*

Recent work proposes domain-specific and general compilers for HE [8, 18, 31, 36, 47, 49, 50]. Prior work such as CHET [50] and nGraph-HE [18] are domain-specific HE compilers for deep neural networks (DNNs). CHET optimizes the data layout of operations in DNNs while nGraph-HE added an HE extension to an existing DNN compiler with new graph-level optimizations. Cingulata [31] and Lobster [84] target Boolean HE schemes and propose compilation strategies that rely on multiplicative depth minimization and synthesizing rewrite rules.

Other HE compilers such as EVA [49] and Alchemy [47] automate parameter selection and placement of low-level scheme HE instructions that control ciphertext properties necessary for correctness, but have no affect on the result of computation (e.g., mod-switch). For example, EVA achieves this for the CKKS scheme using custom rewrite rules but requires a hand-crafted HE kernel as input. On the other hand, Porcupine tackles the orthogonal problem of synthesizing vectorized kernels and optimizes the computational instructions.

The closest work to ours is Ramparts [8] which is a HE compiler that translates plaintext Julia programs to equivalent HE implementations. Unlike Porcupine, Ramparts does not support packed vectorization (i.e., one task cannot use multiple slots in a ciphertext) which is required for taking advantage of SIMD parallelism within a task and improving latency. In contrast, Porcupine supports packed data inputs and can generate kernels with rotations. Furthermore, Ramparts relies on the structure of the input Julia program to serve as the seed for symbolic execution-based methodology which produces a computational circuit that is optimized and lowered to HE instruction with rewrite rules. In contrast, Porcupine places essentially no constraints on the structure of the programs it synthesizes other than the number of instructions it can contain. This enables Porcupine to consider a wider range of programs when optimizing.

Overall, Porcupine is the first compiler that applies program synthesis to optimize vectorization for integer HE constructions.

4.8.2 *Compilers for Privacy-Preserving Computation*

Compiler support has also been proposed for other privacy-preserving techniques, such as differential privacy (DP) [51] and secure multi-party computation (MPC) [60, 147] to automatically enforce or reason about restrictions and constraints by these technologies. For instance, DP requires adding noise to the algorithm and deriving that the effect of an individual's information is in fact differentially private (i.e., has indistinguishable effect on the aggregate data). In DP, there are proposals for using type systems to enforce differential privacy [57, 94, 110]. Other programming language techniques [14] include

dynamic approaches [89, 90, 115], static checking [57, 96, 110], and machine-checked proofs [15]. A similar trend is occurring in MPC where implementations must also comply with design constraints to collaboratively compute functions while still protecting private inputs from other users. Recent work by [67, 108, 127, 143, 148] proposes and/or evaluates general-purpose compiler for MPC.

4.8.3 *Synthesizing Kernels*

Prior work has shown program synthesis to be effective for compiling and optimizing programs for various goals and targets. For example, Chlorophyll [100] introduced a synthesis-based compiler to target an scalar spatial architecture with a stack-based language. By pairing a naive code generation with a synthesis based superoptimizer they were able to quickly build an optimizing compiler. Spiral [104] generates optimized DSP kernels using both inductive and deductive synthesis techniques

Swizzle Inventor [99] synthesized optimized data movement for GPU kernels from a sketch that specified that computation strategy and left data movement unspecified. Because their objective was to only optimize data movement, they relied on canonicalization for verification (not an SMT solver) which does not allow their synthesis formulation to optimize algebraic expressions but improves synthesis time. On the other hand, our synthesis formulation needs to optimize algebraic expressions as part of selecting arithmetic instructions so requires an SMT solver.

Program synthesis has also been used to auto-vectorize code. For example, Barthe et al. introduced an auto-vectorization method [13] that transformed scalar loops into SIMD implementations (Intel SSE4) by restructuring loops to expose parallelism and then synthesizing a straight-line SIMD loop body using an enumerative search. Porcupine does not rely on a loop restructuring phase and our synthesis procedure optimizes the entire kernel, allowing us to handle nested loops. Furthermore, our search optimizes an HE cost model that accounts for multiplicative depth and handles vectors larger than four lane CPU SIMD vectors.

4.9 CONCLUSION

This chapter described Porcupine, a program synthesis-based compiler that automatically generates vectorized HE kernels. Porcupine automatically performs the instruction selection and scheduling to generate efficient HE kernels and minimize the HE noise budget. By automating these tasks, Porcupine abstracts away the details of constructing correct HE computation so that application designers can concentrate on other design considerations. HE is still a rapidly maturing area of research and there is limited related work in this space. As

a result, we expect that in future work we will see rapid improvements to compilation infrastructure such as ours.

5

CONCLUSION

This thesis argues that search-based optimization methods, in particular program synthesis, are promising methods for optimizing programs. Building domain specific optimizers that use automated search-based methods to discover and tailor optimizations to a particular problem is a promising solution to achieving good performance for hardware and applications that are becoming increasingly more complex. My dissertation demonstrates this by describing how we can automatically generate efficient kernels for machine learning and Homomorphic Encryption (HE).

First, I showed how we could combine autotuning scheduling parameters and synthesizing low-level vector intrinsics to optimize and deploy ultra quantized neural networks. Second, I introduced Porcupine, a synthesis based vectorizing compiler for HE. Porcupine used program synthesis to search for optimized and correct HE kernels that operate over packed vectors to overcome performance and programmability challenges of HE. Together, these two compilation systems demonstrated that we could search for optimized programs instead of manually optimizing them.

Looking forward, as we continue down the route of specialization in hardware and by construction the compilers that generate code for them, I think we will see search-based optimization techniques begin to take a larger role. Existing compilers provide a framework for lowering and applying transformations that worked extremely well for scalar CPU code. But, developing the set of optimizations and heuristics that make them so successful required a huge expenditure of human effort. As hardware becomes more diverse and specialized, compilers will need to become more flexible and employ new optimization tactics. I think in the future we will want compilers to not only provide this infrastructure for lowering and optimizing code but also discovering the optimizations they should use.

COMBINING MULTIPLE OPTIMIZATION TECHNIQUES Optimization techniques have different strengths and weaknesses. While program synthesis can potentially discover new optimizations without explicit programmer direction, to my knowledge, no commercial compiler uses synthesis in an online setting because of the compile time costs. Most people don't want to wait hours for programs to compile, which makes the relatively fast and consistent compile time of heuristic driven rewrite rules appealing.

I think combining multiple optimization techniques that complement each other by operating on different abstraction levels will become more common. Prior work has combined the two by synthesizing a library of peephole opti-

mizations or rewrites [84, 101, 118] and using them online to get the benefit of synthesized rules with the speed of rewriting. In Chapter 3, we took a similar approach and offline synthesized a set of small microkernels that were presented as options for an autotuner to select while optimizing larger convolution operators. The autotuner didn't concern itself with low-level instructions and accepted the microkernels as input and could scale to optimize entire operators, while the synthesizer focused on instruction selection for a very small microkernel and didn't model memory behavior. Combining these two techniques allowed us to benefit from both specialized intrinsic and tuned parallelism and locality.

Compiler frameworks can make this task easier by providing infrastructure for applying different techniques and *composing* them in a modular fashion.

IMPROVED TOOLS AND INFRASTRUCTURE Search-based optimizers, and in particular synthesizers, are still fairly niche tools. Previously, synthesizers took years to build, as they relied on custom synthesis formulations that required expertise in formal methods, software engineering, programming languages, and the application domain the synthesizer was targeting.

Instead the synthesizers employed in this dissertation only required months of work by a single person. I relied on the development of symbolic-aided languages like Rosette [131] to quickly build up synthesizers, as well as symbolic profilers [21] to help debug performance bottlenecks. Infrastructure and tools are crucial for lowering the programming barrier to developing optimizers [83, 131, 146]. However, even with these tools, scaling up these synthesizers still required time and effort in designing DSLs in a synthesis friendly way and developing useful sketches. I think future tools that helps provide feedback or more automation when designing DSLs and sketches will continue to make synthesis even more accessible.

BIBLIOGRAPHY

[1] Martin Abadi et al. "TensorFlow: A system for large-scale machine learning." In: *OSDI*. 2016.

[2] Armin Alaghi and John P Hayes. "On the functions realized by stochastic computing circuits." In: *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*. 2015.

[3] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018.

[4] Frances Allen and John Cocke. "A catalogue of optimizing transformations." In: *Design and Optimizations of Compilers* (1972).

[5] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. *Syntax-guided synthesis*. 2013. DOI: 10.1109/FMCAD.2013.6679385.

[6] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. "Petabricks: A language and compiler for algorithmic choice." In: *ACM Sigplan Notices* (2009).

[7] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "Opentuner: An extensible framework for program autotuning." In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014.

[8] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. "RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications." In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'19. 2019. DOI: 10.1145/3338469.3358945. URL: https://doi.org/10.1145/3338469.3358945.

[9] Arm. *Compute Library*. https://github.com/ARM-software/ComputeLibrary.

[10] Riyadh Baghdadi, Massinissa Merouani, Mohamed-Hicham Leghettas, Kamel Abdous, Taha Arbaoui, Karima Benatchba, et al. "A Deep Learning Based Cost Model for Automatic Code Optimization." In: *Proceedings of Machine Learning and Systems* (2021).

[11] Prasanna Balaprakash, Jack Dongarra, Todd Gamblin, Mary Hall, Jeffrey K Hollingsworth, Boyana Norris, and Richard Vuduc. "Autotuning in high-performance computing applications." In: *Proceedings of the IEEE* (2018).

[12] Paul Barham and Michael Isard. "Machine learning systems are stuck in a rut." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019.

[13] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, Cesar Kunz, and Mark Marron. "From Relational Verification to SIMD Loop Synthesis." In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. 2013. DOI: 10.1145/2442516.2442529. URL: https://doi.org/10.1145/2442516.2442529.

[14] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin Pierce. "Programming Language Techniques for Differential Privacy." In: *ACM SIGLOG News* (2016). DOI: 10.1145/2893582.2893591. URL: https://doi.org/10.1145/2893582.2893591.

[15] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. "Probabilistic Relational Reasoning for Differential Privacy." In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. 2012. DOI: 10.1145/2103656.2103670. URL: https://doi.org/10.1145/2103656.2103670.

[16] Eli Ben-Sasson, Matan Hamilis, Mark Silberstein, and Eran Tromer. "Fast multiplication in binary fields on gpus via register cache." In: *Proceedings of the 2016 International Conference on Supercomputing*. 2016.

[17] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. "The polyhedral model is more widely applicable than you think." In: *International Conference on Compiler Construction*. Springer. 2010.

[18] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. "NGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data." In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'19. 2019. DOI: 10.1145/3338469.3358944. URL: https://doi.org/10.1145/3338469.3358944.

[19] Mark Bohr. "A 30 year retrospective on Dennard's MOSFET scaling paper." In: *IEEE Solid-State Circuits Society Newsletter* (2007).

[20] Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. "Pluto: A practical and fully automatic polyhedral program optimization system." In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. Citeseer. 2008.

[21] James Bornholt and Emina Torlak. "Finding code that explodes under symbolic evaluation." In: *Proceedings of the ACM on Programming Languages* (2018).

[22] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. "Optimizing synthesis with metasketches." In: *POPL*. 2016.

[23] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. "Improved security for a ring-based fully homomorphic encryption scheme." In: *IMA International Conference on Cryptography and Coding*. Springer. 2013, pp. 45–64.

[24] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. "Loop parallelization algorithms: From parallelism extraction to code generation." In: *Parallel Computing* (1998).

[25] Zvika Brakerski. "Fully homomorphic encryption without modulus switching from classical GapSVP." In: *Advances in cryptology–crypto 2012*. Springer, 2012.

[26] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping." In: *ACM Trans. Comput. Theory* (2014). DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600.

[27] Robert Brummayer and Armin Biere. "Boolector: An efficient SMT solver for bit-vectors and arrays." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2009, pp. 174–177.

[28] R. M. Burstall and John Darlington. "A Transformation System for Developing Recursive Programs." In: *J. ACM* (1977). DOI: 10.1145/321992.321996.

[29] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. "Deep learning with low precision by half-wave gaussian quantization." In: *CVPR*. 2017.

[30] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. "Synthesizing optimal collective algorithms." In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2021, pp. 62–75.

[31] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: a compilation chain for privacy preserving applications." In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.

[32] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks." In: *JSSC* (2017).

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

[34] Tianqi Chen et al. "TVM: An automated end-to-end optimizing compiler for deep learning." In: *OSDI*. 2018.

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

[36] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic encryption for arithmetic of approximate numbers." In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2017, pp. 409–437.

[37] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cuDNN: Efficient primitives for deep learning." In: *arXiv preprint arXiv:1410.0759* (2014).

[38] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. "Optimizing database-backed applications with query synthesis." In: *ACM SIGPLAN Notices* 48.6 (2013), pp. 3–14.

[39] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. *TFHE: Fast Fully Homomorphic Encryption Library*. https://tfhe.github.io/tfhe/. August 2016.

[40] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. "Accurate and efficient 2-bit quantized neural networks." In: *SysML conference*. 2019.

[41] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, et al. "Serving DNNs in real time at datacenter scale with project brainwave." In: *IEEE Micro* (2018).

[42] Intel Corporation. *Intel Math Kernel Library reference manual*. 2009.

[43] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1." In: *arXiv preprint arXiv:1602.02830* (2016).

[44] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T Lee, and Brandon Reagen. "Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption." In: *Proceedings of the 42nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2021.

[45] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. "Automatic generation of high-performance quantized machine learning kernels." In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. 2020.

[46] Meghan Cowan, Thierry Moreau, Tianqi Chen, and Luis Ceze. "Towards automating generation of low precision deep learning operators." In: *MLPCD*. arXiv preprint arXiv:1810.11066. 2018.

[47] Eric Crockett, Chris Peikert, and Chad Sharp. "ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. 2018. DOI: 10.1145/3243734.3243828. URL: https://doi.org/10.1145/3243734.3243828.

[48] Christopher Edward Cummins. "Deep learning for compilers." In: (2020).

[49] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. "EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '20. 2020. DOI: 10.1145/3385412.3386023. URL: https://doi.org/10.1145/3385412.3386023.

[50] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. "CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing." In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '19. 2019. DOI: 10.1145/3314221.3314628. URL: https://doi.org/10.1145/3314221.3314628.

[51] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. "Calibrating noise to sensitivity in private data analysis." In: *Theory of cryptography conference*. Springer. 2006, pp. 265–284.

[52] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling." In: *2011 38th Annual international symposium on computer architecture (ISCA)*. IEEE. 2011.

[53] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption." In: *IACR Cryptol. ePrint Arch.* 2012 (2012), p. 144.

[54] Paul Feautrier. "Automatic parallelization in the polytope model." In: *The Data Parallel Programming Model*. Springer, 1996.

[55] William Fedus, Barret Zoph, and Noam Shazeer. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." In: *arXiv preprint arXiv:2101.03961* (2021).

[56] Joshua Fromm, Meghan Cowan, Matthai Philipose, Luis Ceze, and Shwetak Patel. "Riptide: Fast end-to-end binarized neural networks." In: *Proceedings of Machine Learning and Systems* (2020).

[57] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C. Pierce. "Linear Dependent Types for Differential Privacy." In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. 2013. DOI: 10.1145/2429069.2429113. URL: https://doi.org/10.1145/2429069.2429113.

[58] Craig Gentry. "Fully Homomorphic Encryption Using Ideal Lattices." In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC '09. 2009. DOI: 10.1145/1536414.1536440. URL: https://doi.org/10.1145/1536414.1536440.

[59] Craig Gentry. "Computing Arbitrary Functions of Encrypted Data." In: *Commun. ACM* (2010). DOI: 10.1145/1666420.1666444. URL: https://doi.org/10.1145/1666420.1666444.

[60] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play any mental game, or a completeness theorem for protocols with honest majority." In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 307–328.

[61] Kazushige Goto and Robert A van de Geijn. "Anatomy of high-performance matrix multiplication." In: *ACM Transactions on Mathematical Software (TOMS)* (2008).

[62] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. "Polly-Polyhedral optimization in LLVM." In: *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*. 2011.

[63] Sumit Gulwani, William R Harris, and Rishabh Singh. "Spreadsheet data manipulation using examples." In: *Communications of the ACM* (2012).

[64] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of Loop-Free Programs." In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. 2011. DOI: 10.1145/1993498.1993506. URL: https://doi.org/10.1145/1993498.1993506.

[65] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of loop-free programs." In: *PLDI*. 2011.

[66] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. "Program synthesis." In: *Foundations and Trends® in Programming Languages* (2017).

[67] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. "SoK: General Purpose Compilers for Secure Multi-Party Computation." In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019. DOI: 10.1109/SP.2019.00028.

[68] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. "Applied machine learning at facebook: A datacenter infrastructure perspective." In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2018.

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

[70] Yuwei Hu, Jidong Zhai, Dinghua Li, Yifan Gong, Yuhao Zhu, Wei Liu, Lei Su, and Jiangming Jin. "BitFlow: Exploiting vector parallelism for binary neural networks on CPU." In: *IPDPS*. 2018.

[71] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. "Quantized neural networks: Training neural networks with low precision weights and activations." In: *JMLR* 18 (2017).

[72] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. "Quantization and training of neural networks for efficient integer-arithmetic-only inference." In: *CVPR*. 2018.

[73] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. "Oracle-guided component-based program synthesis." In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. 2010. DOI: 10.1145/1806799.1806833.

[74] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. "TASO: optimizing deep learning computation with automatic generation of graph substitutions." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019.

[75] Rajeev Joshi, Greg Nelson, and Keith Randall. "Denali: A goal-directed superoptimizer." In: *ACM SIGPLAN Notices* (2002).

[76] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In: *ISCA*. 2017.

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

[78] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "Gazelle: A low latency framework for secure neural network inference." In: *arXiv preprint arXiv:1801.05507* (2018).

[79] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman P. Amarasinghe. "The tensor algebra compiler." In: *OOPSLA*. 2017.

[80] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. "When polyhedral transformations meet SIMD code generation." In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013.

[81] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. "Dependence graphs and compiler optimizations." In: *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1981.

[82] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004.

[83] 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: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2021, pp. 2–14.

[84] DongKwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. "Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '20. 2020. DOI: 10.1145/3385412.3385996. URL: https://doi.org/10.1145/3385412.3385996.

[85] Saeed Maleki, Yaoqing Gao, Maria J Garzar, Tommy Wong, David A Padua, et al. "An evaluation of vectorizing compilers." In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011.

[86] Zohar Manna and Richard Waldinger. "Synthesis: dreams→ programs." In: *IEEE Transactions on Software Engineering* (1979).

[87] Henry Massalin. "Superoptimizer: A look at the smallest program." In: *ASPLOS*. 1987.

[88] Amrita Mazumdar, Thierry Moreau, Sung Kim, Meghan Cowan, Armin Alaghi, Luis Ceze, Mark Oskin, and Visvesh Sathe. "Exploring computation-communication tradeoffs in camera systems." In: *IISWC*. 2017.

[89] Frank McSherry and Ratul Mahajan. "Differentially-Private Network Trace Analysis." In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. 2010. DOI: 10.1145/1851182.1851199. URL: https://doi.org/10.1145/1851182.1851199.

[90] Frank D. McSherry. "Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis." In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. 2009. DOI: 10.1145/1559845.1559850. URL: https://doi.org/10.1145/1559845.1559850.

[91] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, and Carlos Guestrin. "A hardware–software blueprint for flexible deep learning specialization." In: *IEEE Micro* (2019).

[92] Marius Muja and David Lowe. "Flann-fast library for approximate nearest neighbors user manual." In: *Computer Science Department, University of British Columbia, Vancouver, BC, Canada* (2009).

[93] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. "Synthesizing structured CAD models with equality saturation and inverse transformations." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020.

[94] Joseph P. Near et al. "Duet: An Expressive Higher-Order Language and Linear Type System for Statically Enforcing Differential Privacy." In: *Proc. ACM Program. Lang.* OOPSLA (2019). DOI: 10.1145/3360598. URL: https://doi.org/10.1145/3360598.

[95] David A Padua and Michael J Wolfe. "Advanced compiler optimizations for supercomputers." In: *Communications of the ACM* (1986).

[96] Catuscia Palamidessi and Marco Stronati. "Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems." In: *arXiv preprint arXiv:1207.0872* (2012).

[97] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. "Automatically improving accuracy for floating point expressions." In: *ACM SIGPLAN Notices* (2015).

[98] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In: *arXiv preprint arXiv:1912.01703* (2019).

[99] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. "Swizzle Inventor: Data Movement Synthesis for GPU Kernels." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. 2019. DOI: 10.1145/3297858.3304059. URL: https://doi.org/10.1145/3297858.3304059.

[100] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. "Chlorophyll: Synthesis-Aided Compiler for Low-Power Spatial Architectures." In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. 2014. DOI: 10.1145/2594291.2594339. URL: https://doi.org/10.1145/2594291.2594339.

[101] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, and Dinakar Dhurjati. "Scaling up superoptimization." In: *ASPLOS*. 2016.

[102] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. "PALISADE lattice cryptography library user manual." In: *Cybersecurity Research Center, New Jersey Institute ofTechnology (NJIT), Tech. Rep* (2017).

[103] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. "GRAPHITE: Polyhedral analyses and optimizations for GCC." In: *Proceedings of the 2006 GCC Developers Summit*. Citeseer. 2006.

[104] M. Puschel et al. "SPIRAL: Code Generation for DSP Transforms." In: *Proceedings of the IEEE* (2005). DOI: 10.1109/JPROC.2004.840306.

[105] Racket. *The Racket programming language*. URL: racketlang.org.

[106] 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." In: *PLDI*. 2013.

[107] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "Xnor-net: Imagenet classification using binary convolutional neural networks." In: *ECCV*. 2016.

[108] A. Rastogi, M. A. Hammer, and M. Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations." In: *2014 IEEE Symposium on Security and Privacy*. 2014. DOI: 10.1109/SP.2014.48.

[109] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent Lee, Gu-Yeon Wei, Hsien-Hsin S Lee, and David Brooks. "Cheetah: Optimizations and Methods for Privacy Preserving Inference via Homomorphic Encryption." In: *arXiv preprint arXiv:2006.00505* (2020).

[110] Jason Reed and Benjamin C. Pierce. "Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy." In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. 2010. DOI: 10.1145/1863543.1863568. URL: https://doi.org/10.1145/1863543.1863568.

[111] Alastair Reid. "Who guards the guards? Formal validation of the Arm V8-m architecture specification." In: *OOPSLA*. 2017.

[112] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. "HEAX: An Architecture for Computing on Encrypted Data." In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. 2020. DOI: 10.1145/3373376.3378523. URL: https://doi.org/10.1145/3373376.3378523.

[113] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. "Relay: A new ir for machine learning frameworks." In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2018.

[114] 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." In: *arXiv preprint arXiv:1805.00907* (2018).

[115] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. "Airavat: Security and privacy for MapReduce." In: *NSDI*. Vol. 10. 2010, pp. 297–312.

[116] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. "Imagenet large scale visual recognition challenge." In: *IJCV* 115 (2015).

[117] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Gratian Lup, Jubi Taneja, and John Regehr. "Souper: A synthesizing superoptimizer." In: *arXiv preprint arXiv:1711.04422* (2017).

[118] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic Superoptimization." In: ASPLOS '13 (2013). DOI: 10.1145/2451116.2451150. URL: https://doi.org/10.1145/2451116.2451150.

[119] *Microsoft SEAL (release 3.5)*. https://github.com/Microsoft/SEAL. Microsoft Research, Redmond, WA. Apr. 2020.

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

[121] Jaewook Shin, Mary W Hall, Jacqueline Chame, Chun Chen, Paul F Fischer, and Paul D Hovland. "Speeding up nek5000 with autotuning and specialization." In: *Proceedings of the 24th ACM International Conference on Supercomputing*. 2010.

[122] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. "Automated Feedback Generation for Introductory Programming Assignments." In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. 2013. DOI: 10.1145/2491956.2462195. URL: https://doi.org/10.1145/2491956.2462195.

[123] Nigel P Smart and Frederik Vercauteren. "Fully homomorphic SIMD operations." In: *Designs, codes and cryptography* (2014).

[124] Armando Solar-Lezama and Rastislav Bodik. *Program synthesis by sketching*. Citeseer, 2008.

[125] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. "Combinatorial sketching for finite programs." In: *ASPLOS*. 2006.

[126] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. "Combinatorial Sketching for Finite Programs." PhD thesis. 2006. DOI: 10.1145/1168857.1168907. URL: https://doi.org/10.1145/1168857.1168907.

[127] E. M. Songhori, S. U. Hussain, A. Sadeghi, T. Schneider, and F. Koushanfar. "TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits." In: *2015 IEEE Symposium on Security and Privacy*. 2015. DOI: 10.1109/SP.2015.32.

[128] Paul Syverson. "A taxonomy of replay attacks [cryptographic protocols]." In: *Proceedings The Computer Security Foundations Workshop VII*. IEEE. 1994, pp. 187–191.

[129] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. "Equality saturation: a new approach to optimization." In: *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2009.

[130] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K Hollingsworth. "A scalable auto-tuning framework for compiler optimization." In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE. 2009.

[131] Emina Torlak and Rastislav Bodik. "Growing Solver-Aided Languages with Rosette." In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. Onward! 2013. 2013. DOI: 10.1145/2509578.2509586. URL: https://doi.org/10.1145/2509578.2509586.

[132] Emina Torlak and Rastislav Bodik. "Growing solver-aided languages with Rosette." In: *Onward!* 2013.

[133] Emina Torlak and Rastislav Bodik. "A lightweight symbolic virtual machine for solver-aided host languages." In: *PLDI*. 2014.

[134] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. "Polyhedral-model guided loop-nest auto-vectorization." In: *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2009.

[135] Andrew Tulloch. *Private communication*. 2019.

[136] Andrew Tulloch and Yangqing Jia. "High performance ultra-low-precision convolutions on mobile devices." In: *arXiv preprint arXiv:1712.02427* (2017).

[137] Yaman Umuroglu. *Private communication*. 2018.

[138] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. "Finn: A framework for fast, scalable binarized neural network inference." In: *FPGA*. 2017.

[139] Yaman Umuroglu and Magnus Jahre. "Towards efficient quantized neural network inference on mobile devices." In: *CASES*. 2017.

[140] Yaman Umuroglu, Lahiru Rasnayake, and Magnus Själander. "Bismo: A scalable bit-serial matrix multiplication overlay for reconfigurable computing." In: *FPL*. 2018.

[141] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. "Vectorization for digital signal processors via equality saturation." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2021.

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

[143] Xiao Wang, Alex J Malozemoff, and Jonathan Katz. *EMP-toolkit: Efficient MultiParty computation toolkit*. 2016.

[144] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. "SPORES: sum-product optimization via relational equality saturation for large scale linear algebra." In: *arXiv preprint arXiv:2002.07951* (2020).

[145] R Clinton Whaley and Jack J Dongarra. "Automatically tuned linear algebra software." In: *SC'98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*. IEEE. 1998.

[146] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. "egg: fast and extensible equality saturation." In: *Proceedings of the ACM on Programming Languages* POPL (2021).

[147] Andrew Chi-Chih Yao. "How to generate and exchange secrets." In: *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*. IEEE. 1986.

[148] Samee Zahur and David Evans. "Obliv-C: A Language for Extensible Data-Oblivious Computation." In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 1153.

[149] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients." In: *arXiv preprint arXiv:1606.06160* (2016).

[150] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter Ma, Qiumin Xu, Hanxiao Liu, Mangpo Phitchaya Phothilimtha, Shen Wang, Anna Goldie, et al. "Transferable graph optimizers for ml compilers." In: *arXiv preprint arXiv:2010.12438* (2020).

COLOPHON

This dissertation was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić.

Final Version as of June 23, 2021 (classicthesis v4.6).