Automatic Generation of High-Performance Quantized Machine Learning Kernels

Meghan Cowan
University of Washington
Seattle, WA, USA
cowanmeg@cs.uw.edu

Thierry Moreau
University of Washington
Seattle, WA, USA
moreau@cs.uw.edu

Tianqi Chen
University of Washington
Seattle, WA, USA
tqchen@cs.uw.edu

James Bornholt
University of Texas at Austin
Austin, TX, USA
bornholt@cs.utexas.edu

Luis Ceze
University of Washington
Seattle, WA, USA
luisceze@cs.uw.edu

Abstract

Quantization optimizes machine learning inference for resource constrained environments by reducing the precision of its computation. In the extreme, even single-bit computations can produce acceptable results at dramatically lower cost. But this ultra-low-precision quantization is difficult to exploit because extracting optimal performance requires hand-tuning both high-level scheduling decisions and low-level implementations. As a result, practitioners settle for a few predefined quantized kernels, sacrificing optimality and restricting their ability to adapt to new hardware.

This paper presents a new 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. We scale up synthesis using a novel *reduction sketch* that exploits the structure of matrix multiplication. 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×.

CCS Concepts • **Software and its engineering → Compilers**.

Keywords quantization, machine learning, synthesis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CGO '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7047-9/20/02…$15.00
https://doi.org/10.1145/3368826.3377912

ACM Reference Format:
Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Luis Ceze. 2020. Automatic Generation of High-Performance Quantized Machine Learning Kernels. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO '20), February 22–26, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368826.3377912

1 Introduction

Machine learning has seen successes in a wide variety of application areas. These successes come at the cost of dramatic increases in the memory and compute demands of new models such as convolutional neural networks. For example, performing a single prediction (an *inference*) with state-of-the-art image recognition models use tens of millions of parameters and billions of floating-point operations [22, 26]. *Quantization* techniques reduce the scale of these models to make them practical in resource-constrained environments such as smartphones and edge devices [17]. 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 [12, 30]. 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 [24].

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 (we do not consider training in this paper). 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 [42] required four person-months of work [40]. Similarly, the authors of another library [39] spent about a month implementing a single configuration [38]. 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.

This paper presents 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 [4]. 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 [29]. 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 [5] 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 [35] 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 [37], and integrate the generated code into the TVM machine learning compiler [4]. 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 [15] 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 [39] when configured in the 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. in edge applications such as video processing [24]. Automation also allows us to explore the space of possible quantization configurations, performing a limit study of the potential speedups available from quantization.

In summary, this paper makes the following contributions:

- Quantization-aware scheduling for compilation of machine learning models
- An automated scheduling approach to tuning quantized implementation for new models and hardware
- A new program synthesis approach to generating quantized low-level tensor kernels
- Results showing we can automatically generate code that executes quantized models up to 12.4× faster than state-of-the-art hand-written code.

2 Quantized Models

Quantized machine learning models promise to reduce the compute and memory demands of inference while maintaining acceptable accuracy. In this paper, we focus on both fully connected and convolutional neural 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 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

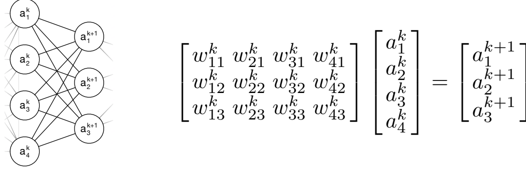


Figure 1. In a fully connected neural network, computing the activations for layer $k+1$ involves a matrix-vector multiplication, which reduces to a series of vector dot products.

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 [11] and NVIDIA's cuDNN [8].

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 [18]. 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 [17, 45]. In the extreme, even models with single-bit weights and activations can achieve near state-of-the-art results on some problems [12, 30].

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 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 2 shows an example in which we quantize weights to 3 bits and activations to 2 bits. The first step is to decompose

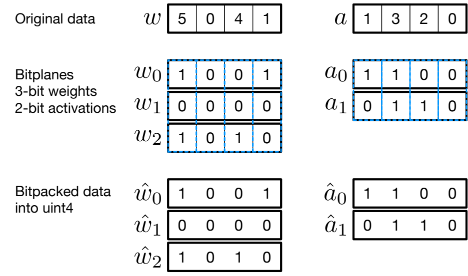


Figure 2. Slicing the values of weights and activations into bitplanes enables vector dot products to be computed with bitwise operations.

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

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

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 paper, 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 [3, 9, 45] uses unipolar encoding for activations and bipolar encoding for weights, making the dot product more complex:

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

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

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.

3.1 Bit-Slicing Schedules

A machine learning compiler such as TVM [4] 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 [29], 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 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 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 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,

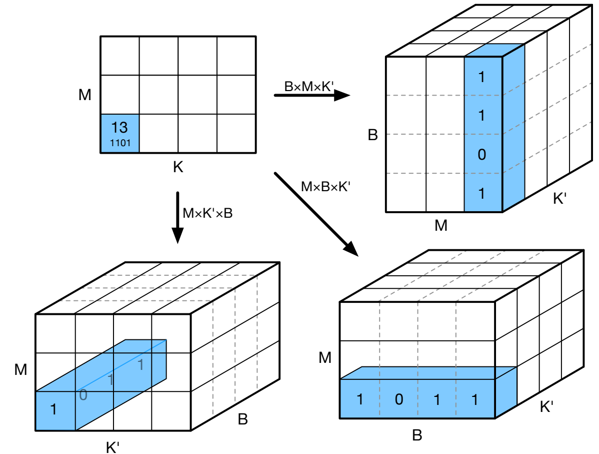


Figure 3. The bit-packing transformation transforms a d-dimensional tensor into a $d+1$-dimensional one by slicing the bits of elements into a new *bit axis*. Here the two-dimensional tensor $M \times K$ is transformed in different ways depending on the chosen location of the new bit axis B.

with a new bit axis B that indexes the bits of each original element. The K' axis after transformation has $K' \leq K$, 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 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.

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 [4], which is similar to those of Halide [29] and TACO [21]. 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 4 how to automatically synthesize implementations for each such precision.

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 [39]. This

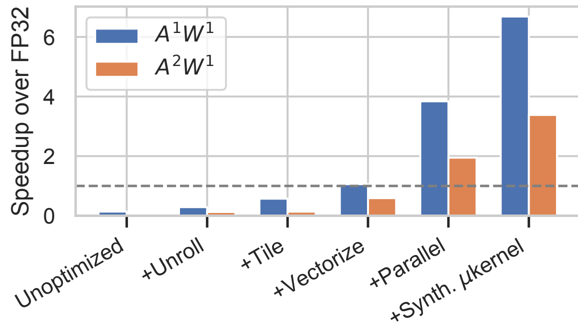


Figure 4. The cumulative effects of adding scheduling primitives, starting from an un-optimized quantized convolution. $A^X W^Y$ denotes an x-bit activation, y-bit weight quantized convolution. Speedup is relative to 32-bit floating point on ResNet-18 layer C6.

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.

We reuse the scheduling primitives of TVM [4], which itself uses the scheduling primitives of Halide [29]. 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 4 shows the importance of these optimizations to the performance of a quantized model (Section 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 4) improves performance by another 1.5–2× over the optimized quantized model; Section 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 [5]. 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.

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 [14, 35]. 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.

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 paper. Our goal is to synthesize 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 [36, 37], 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 [31], 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 [23, 27]). The synthesis engine thus takes as input a cost function, mapping each program to an integer cost, and finds the program with minimum cost [2]. 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.

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* [35], 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 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))
    ((??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.

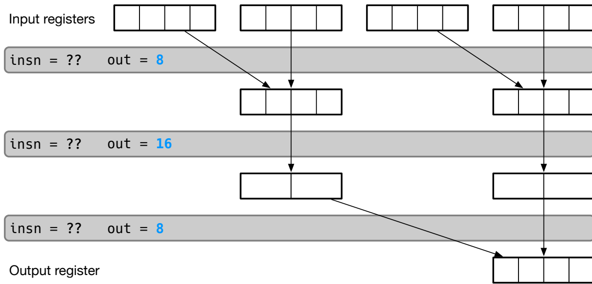
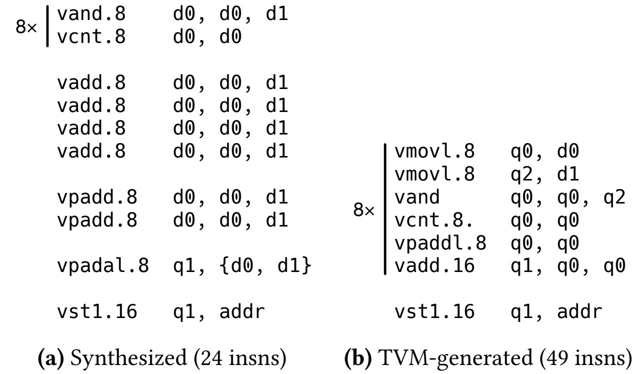


Figure 5. A reduction sketch is a tree that reduces input registers down to a single register. Each level applies an instruction to each pair of live registers. Here the output types have already been synthesized; each level is free to decide the types (vector lane widths) of its outputs.



(a) Synthesized (24 insns) **(b)** TVM-generated (49 insns)

Figure 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.

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 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 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.

4.3 Implementation

We implement our synthesis engine in the Rosette solver-aided language [37], 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 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's *tensorization* schedule primitive, which lowers tensor operations to LLVM IR in a template-driven fashion. 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 4 shows that the synthesized kernel is 1.5–2.5× faster on an ARM CPU.

Other platforms. We have also implemented a synthesis backend for x86's AVX2 vector instruction set. The implementation took only a few days of work for one author, and only required developing a specification of relevant AVX2 instructions. While we can successfully synthesize microkernels for this architecture, AVX2 lacks a vectorized popcount instruction, and so the kernels are not competitive in performance with floating-point implementations. However, our experience with x86 suggests that synthesis enables rapid porting to new architectures, including potentially to programmable accelerators (as we discuss more in Section 6).

Table 1. Configurations of 2D-convolution operators in ResNet18 [15]. 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.

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

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 [15], 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 1.

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 [42]. Each kernel we generate, including floating-point baselines, is optimized independently using AutoTVM [5]. For each convolutional layer, we run AutoTVM for a total 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

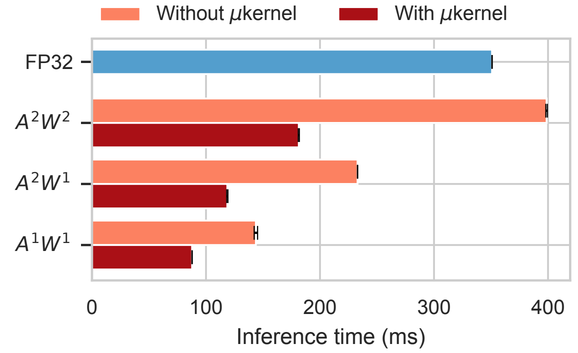


Figure 7. End-to-end inference times for ResNet18 on an ARM CPU, with and without synthesized microkernels.

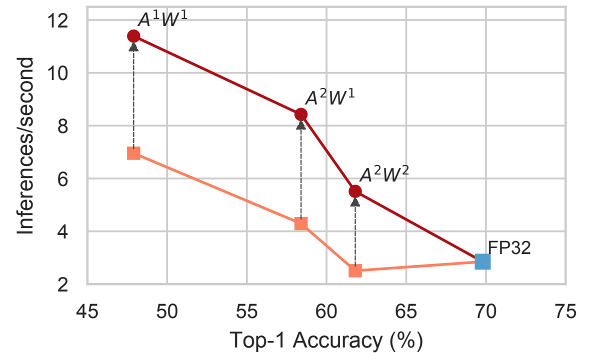


Figure 8. Accuracy versus performance for quantized ResNet18 models, with and without synthesized microkernels. Higher is better on both axes.

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.

5.1 Quantization versus Floating Point

To demonstrate the performance benefits of quantization, Figure 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 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.

Figure 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 barely outperform floating point at A^2W^2. End-to-end inference is an average of 1.9× faster using our synthesized microkernels.

The improved performance of our synthesized microkernels shift the Pareto frontier of the accuracy–performance trade-off for quantization. Figure 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). Newer training techniques can further reduce the accuracy loss, as we survey in Section 6.

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 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 repacking 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.

5.2 Comparison to Hand-Written Code

Our automatically generated quantized kernels a outperform hand-written quantized implementations developed by experts. Figure 9 compares our implementation to the hand-written ultra-low-precision convolution library in PyTorch [39]. 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

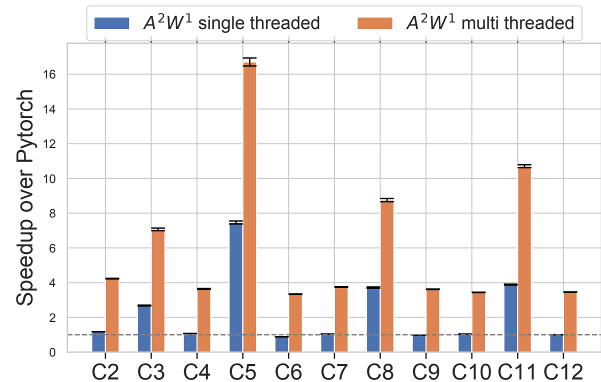


Figure 9. Layer-by-layer speedups for A^2W^1 convolutions normalized to PyTorch's hand-optimized A^2W^1 kernel for ARM [39]. The PyTorch kernels are single-threaded, so we compare against both single- and multi-threaded versions generated by our approach.

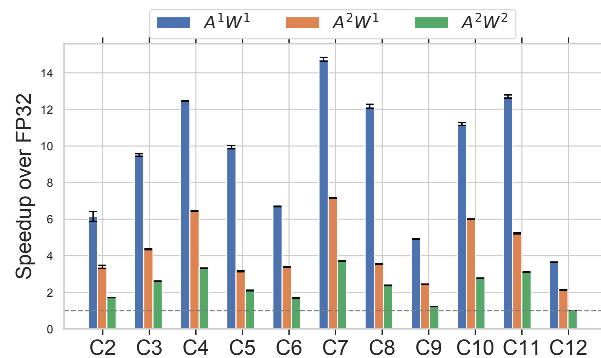


Figure 10. Quantized convolution speedups for various quantization configurations, normalized to 32-bit floating point.

single-threaded. Our version does outperform PyTorch on some layers by up to 7× (C5, C8, C11); these layers are "downsampling" 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 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.

5.3 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

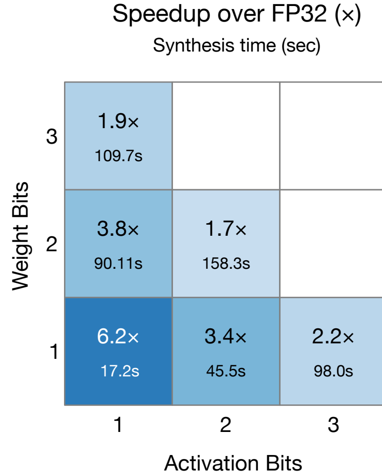


Figure 11. Relative speedup over 32-bit floating point, and kernel synthesis time, for convolutional layer C2 at different quantization configurations.

Table 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.

Config	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

exploited this automation to perform a limit study of the possible speedups available at reduced precisions.

Figure 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 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 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 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 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 11 also shows the performance of our synthesis engine for each configuration, with more detail in Table 2. Synthesis performance worsens as the number of bits increases—from 17 seconds at A^1W^1 (two bits total) to 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 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.

6 Related Work

Quantized Neural Networks. Whereas this paper 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 [12] 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 [30] 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 [33].

More recent quantized neural networks focus on reducing the accuracy gap on ImageNet by increasing the precision. DoReFa-Net [45] and HWGQ [3] 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. [9] 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 [42] and Tulloch and Jia [39] 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 [16] 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 [4] supported ultra-low-precision quantization using hand-written microkernels for ARM's NEON vector extensions [13]. 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 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 [21] and Halide [29] automatically generate kernels for sparse linear algebra and image processing, respectively. Machine learning compilers such as Tensor Comprehensions [44], GLOW [32], and TVM [4] 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 [6, 7, 19] and FPGAs [10, 25, 41] 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 [20, 34, 43] 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 [28] is a tool for generating high-performance implementations of fast Fourier transforms and other DSP primitives. Chlorophyll [27] is a superoptimizer for a low-power spatial architecture. Synapse [2] 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.

7 Conclusion

Our automated approach to compiling quantized models combines the strengths of both machine learning (for scheduling [5]) and program synthesis (for code generation [37]). The result is a workflow that generates kernels that outperform than both optimized floating-point and state-of-the-art quantized implementations. 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.

Acknowledgements

We thank Andrew Tulloch and Yaman Umuroglu for sharing their experiences implementing quantized inference; Josh Fromm for his help in training models; and our shepherd Christophe Dubach and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF under grant CCF-1518703; by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by gifts from Xilinx, Intel (under the CAPA program), Oracle, Amazon, Qualcomm, and other anonymous sources; and by a Facebook PhD Fellowship.

References

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

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

[3] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization.

In *CVPR*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26] Omkar M Parkhi, Andrea Vedaldi, Andrew Zisserman, et al. 2015. Deep face recognition. In *BMVC*.

[27] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. In *PLDI*.

[28] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, Jianxin Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. 2005. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* (2005).

[29] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*.

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

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

[32] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. 2018. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907* (2018).

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

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

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

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

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

[38] Andrew Tulloch. 2019. Private communication. (2019).

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

[40] Yaman Umuroglu. 2018. Private communication. (2018).

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

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

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

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

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