Compiling KB-Sized Machine Learning Models to Tiny IoT Devices

Sridhar Gopinath
Microsoft Research, India

Nikhil Ghanathe
Microsoft Research, India

Vivek Seshadri
Microsoft Research, India

Rahul Sharma
Microsoft Research, India

Abstract

Recent advances in machine learning (ML) have produced KiloByte-size models that can directly run on constrained IoT devices. This approach avoids expensive communication between IoT devices and the cloud, thereby enabling energy-efficient real-time analytics. However, ML models are expressed typically in floating-point, and IoT hardware typically does not support floating-point. Therefore, running these models on IoT devices requires simulating IEEE-754 floating-point using software, which is very inefficient.

We present SeeDot, a domain-specific language to express ML inference algorithms and a compiler that compiles SeeDot programs to fixed-point code that can efficiently run on constrained IoT devices. We propose 1) a novel compilation strategy that reduces the search space for some key parameters used in the fixed-point code, and 2) new efficient implementations of expensive operations. SeeDot compiles state-of-the-art KB-sized models to various microcontrollers and low-end FPGAs. We show that SeeDot outperforms 1) software emulation of floating-point (Arduino), 2) high-bitwidth fixed-point (MATLAB), 3) post-training quantization (TensorFlow-Lite), and 4) floating- and fixed-point FPGA implementations generated using high-level synthesis tools.

CCS Concepts • **Software and its engineering → Compilers**; • **Hardware → Sensor devices and platforms**.

Keywords Machine Learning, IoT device, Programming Language, Compiler, Fixed-point, Microcontroller, FPGA

ACM Reference Format:
Sridhar Gopinath, Nikhil Ghanathe, Vivek Seshadri, and Rahul Sharma. 2019. Compiling KB-Sized Machine Learning Models to Tiny IoT Devices. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314597

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06...$15.00
https://doi.org/10.1145/3314221.3314597

1 Introduction

In recent years, we have seen an increase in automation systems that deploy sensors to collect data and analyze the data using machine learning (ML) algorithms. A few examples of such systems include simple health monitoring through wearable sensors [67, 71, 72], large-scale monitoring of big cities [9, 53, 63], and so on. Typical systems use sensor devices (also referred to as IoT devices) that only *collect* data and run the ML algorithms in the cloud [29, 39, 49]. However, running the ML classifiers *directly* on the IoT device has three known advantages [30, 56]. First, it improves the security and privacy of user data by keeping it at the source rather than communicating it to the cloud. Second, it eliminates data communication between the IoT device and the cloud, thereby reducing energy consumption. Third, running the algorithms on the IoT device significantly reduces the latency of prediction, enabling real-time analysis.

Despite these benefits, there are two key challenges in running ML algorithms on IoT devices. First, IoT devices have limited compute and memory resources (few KBs). As a result, they cannot run typical ML models that are MBs or GBs in size. Even frameworks such as TensorFlow-Lite [84] that target embedded systems need MBs of memory to run. Whereas, the largest device that we consider has only 32 KBs of RAM. Second, these IoT devices do not have hardware support for floating-point operations [6]. The absence of floating-point support is problematic as most ML algorithms are expressed in floating-point.

Recent breakthroughs in ML have addressed the first challenge by proposing KB-sized ML models [30, 56] that can fit in the memory available in tiny IoT devices. While these new models may not be able to run heavy-weight tasks, they are powerful enough for tasks such as anomaly detection and activity recognition that are typically useful in IoT applications. Unfortunately, even these models are expressed in floating-point. There are two ways of dealing with the lack of floating-point support: 1) software emulation of floating-point, and 2) conversion to high-bitwidth fixed-point.

First, existing popular tool-chains (e.g., the Arduino IDE) emulate floating-point operations in software. However, faithful software emulation of floating-point must handle all the vagaries of the IEEE-754 standard [46]: ±0, NaNs, denormals, infinites, etc. Consequently, this approach results in poor performance and energy efficiency. The second approach is to convert floating-point operations to fixed-point operations. Existing work in this area focuses on digital signal processors (DSPs) [3, 5, 7, 8, 64, 68, 88]. These approaches rely on high-bitwidth arithmetic which is supported natively by DSPs but is very expensive on microcontrollers.

In this paper, we describe the first framework that generates efficient fixed-point code for ML inference algorithms that can run on constrained hardware. For the purposes of this paper, we define a device as *constrained* if it has KBs of memory and does not have hardware support for floating-point operations. We focus on the scenario where an ML model is trained in the cloud and an IoT maker wants to deploy the trained model directly on the IoT device. To this end, we make the following contributions.

First, we propose SeeDot[1], a domain-specific language for expressing ML inference algorithms. SeeDot is high-level, easy to comprehend, and has intuitive semantics. It provides language support for standard matrix operations, which are the natural abstractions used in ML algorithms. The syntax of SeeDot helps the compiler infer and track dimensions of matrices at compile time which is difficult in general purpose languages like Python/C/C++. With these features, SeeDot improves programmer productivity by making it easy for ML researchers to specify their algorithms. For instance, SeeDot can express the LeNet convolution neural network [83] for object detection on the CIFAR-10 dataset [54] in ten lines of code (Section 7.4). In contrast, the corresponding C program spans hundreds of lines. We formally define the SeeDot language and its semantics in Section 5.

Second, we design a compiler that transforms SeeDot programs to fixed-point C code for microcontrollers. The fixed-point code operates only on *low-bitwidth* integers and is much more efficient than emulating floating-point in software. Our compiler uses two key ideas. First, each fixed-point number is associated with a *scale* parameter. The naïve approach for setting the scales results in an unacceptable loss in precision. The optimal approach for setting the scales requires exploring a parameter space whose size is exponential in the size of the input program. Our compiler uses an intelligent heuristic that reduces the size of the parameter space to a constant which is independent of the size of the input program. This approach results in a precise and efficient fixed-point code in practice. Second, we observe that existing approaches to compute the exponentiation function (e^x) on constrained hardware are very inefficient. We propose an approach that computes e^x as a product of two values that are looked up from two pre-computed tables. With these techniques, SeeDot-generated fixed-point code significantly outperforms code generated by state-of-the-art float-to-fixed compilers. Section 3 provides a motivating example and Section 5.3 describes our optimizations.

[1]Implementation available at https://github.com/Microsoft/EdgeML/tree/master/Tools/SeeDot

Third, we observe that IoT devices are often deployed for specific scenarios. While the models may undergo updates, they do not change significantly in structure and complexity. This characteristic makes them an ideal target for hardware acceleration using Field Programmable Gate Arrays (FPGAs) [69]. As a result, to exhibit the generality of SeeDot, we perform a preliminary evaluation to explore SeeDot's potential to target FPGAs. We augment SeeDot with a backend that generates code to run KB-sized ML models on a low-end, power-efficient Xilinx FPGA with no floating-point support. Our compiler uses the high-level synthesis (HLS) tool provided by Xilinx along with two optimizations. First, our compiler uses a hand-optimized Verilog code for Sparse-Matrix-Vector (SpMV) multiplication, a frequently-occurring operation in ML inference. Second, our compiler automatically generates hints for the HLS compiler to parallelize other operations. To the best of our knowledge, this is the first demonstration of automatically compiling KB-sized ML algorithms specified in a high-level language to low-end FPGAs. Section 6 describes the FPGA-backend in more detail.

We evaluate SeeDot using state-of-the-art KB-sized ML inference algorithms for constrained devices. In the microcontroller setting, we compare the performance of SeeDot-generated code to the code generated by 1) the native Arduino IDE, 2) the commercial MATLAB float-to-fixed converter, and 3) post-training quantization in Tensorflow. For the FPGA setting, we compare the performance of our compiler to Xilinx's HLS tool. Our evaluations show that SeeDot-generated programs achieve comparable classification accuracy for the ML algorithms with a significant reduction in execution time compared to these prior approaches.

Finally, to evaluate the benefit of SeeDot in the real world, we consider two case studies where KB-sized ML models have been deployed in the wild: 1) a fault detection system that uses an ML model to detect whether a soil temperature/moisture sensor deployed in a remote farm has malfunctioned, and 2) a sensor pod that reacts in real-time to gestures performed by people with visual impairments using their white cane. For both scenarios, SeeDot-generated fixed-point code for the ML algorithms has comparable classification accuracy and much better performance than the deployed implementations. Thus, SeeDot is already helpful to farmers and people with visual impairments.

2 Background

In this section, we provide a brief background on machine learning and fixed-point arithmetic.

2.1 ML Preliminaries

An ML classifier takes an input data point (e.g., an image) and assigns it a *label* (e.g., "cat image" or "dog image"). A typical ML dataset has a *training* set and a *testing* set. The training set is used to learn a *model*. The performance of the trained model is judged by its *classification accuracy*, the proportion of points in the testing set that the classifier labels correctly. For example, in the linear classifier $w * x > 0$, the vector w is the trained model, the vector x is the input which needs to be classified, the possible labels are *true* and *false*, and $*$ is the inner-product operation. In this paper, we focus on running KB-sized ML *classifiers* on constrained devices. Therefore, w is stored on the device's memory, x is a run-time input, and the device performs the computation $w * x > 0$. To generate efficient code, the SeeDot compiler has access to the SeeDot program, the trained model, and the training set to learn few parameters for the compiled code. We use the testing set *only* to evaluate the performance of the code generated by our compiler and not for generating the code.

2.2 Accuracy Metric

ML classifiers are typically specified as expressions over Reals. As Real arithmetic requires infinite precision, modern processors approximate Real numbers using floating-point or fixed-point numbers for efficiency. For ML classifiers, the correctness of these implementations can be judged by two metrics: classification accuracy and numerical accuracy. The latter bounds the error between an implementation and a Real specification over all possible inputs. It is well-known that the best numerical accuracy does not necessarily result in the best classification accuracy. In fact, prior work [51, 74] observes that using fewer bits of precision can improve classification accuracy; precision reduction can be seen as a form of regularization that reduces over-fitting. However, the choice of an accuracy metric is orthogonal to the implementation of the compiler and SeeDot can work with any metric. In particular, the example in Section 3 uses numerical accuracy. Other metrics like recall, precision, and F1-score can be used as well. In this paper, we consider an implementation of a classifier to be *satisfactory* if it has good classification accuracy, regardless of the numerical accuracy.

2.3 Fixed-Point Preliminaries

Fixed-point arithmetic represents a Real number r using an integer[2] $\lfloor r * 2^{\mathsf{P}} \rfloor$. The quantity P is called the *scale*. If $\mathsf{P} > 0$ then we say that r has been scaled up by P. If $\mathsf{P} < 0$ then we say that r has been scaled down by $|\mathsf{P}|$.

The choice of scale is critical when using a fixed-point representation. E.g., consider 8-bit integers and $r = \pi = 3.1415\ldots$. A scale of $\mathsf{P} = 5$ is optimal since it produces the most accurate result: $\lfloor \pi * 2^5 \rfloor = 100$ which represents the Real $100/2^5 = 3.125$, the most precise 8-bit fixed-point representation of π. If the scale is too high, e.g., if $\mathsf{P} = 6$ then $\lfloor \pi * 2^6 \rfloor = 200$, which when written as an 8-bit integer corresponds to -56. This situation is an *overflow*. Here, the most significant bits are lost when converting to 8-bit integers and the result is garbage. If the scale is too low, e.g., if $\mathsf{P} = -2$ then $\lfloor \pi * 2^{-2} \rfloor = 0$ and all the significant bits are lost.

Next, we show how the scale parameter can affect the precision of fixed-point addition and multiplication. Consider the real numbers $r_1 = \pi$ and $r_2 = e = 2.71828\ldots$. The corresponding 8-bit fixed-point representation using a scale $\mathsf{P} = 5$ are $y_1 = \lfloor r_1 * 2^{\mathsf{P}} \rfloor = 100$ and $y_2 = \lfloor r_2 * 2^{\mathsf{P}} \rfloor = 86$. To add the two numbers, simply computing $y_1 + y_2$ is *unsafe*, as the operation results in an overflow ($y_1 + y_2 = -70$). The standard approach to avoid the overflow is to first scale down both the numbers by 1, and then computing the sum. With this approach, the computed fixed-point result is $\frac{y_1}{2} + \frac{y_2}{2} = 93$ with a scale $\mathsf{P} = 4$, which corresponds to the Real number $93/2^4 = 5.8125 \approx \pi + e$.

Similarly, to multiply the two numbers r_1 and r_2, computing $y_1 * y_2$ results in an overflow. The standard approach to avoid overflow while multiplying two d-bit fixed-point numbers is to scale them down by $\frac{d}{2}$ before the multiplication, i.e., we evaluate $\frac{y_1}{2^{d/2}} * \frac{y_2}{2^{d/2}}$ in d-bit arithmetic.[3] This process would avoid any overflows due to multiplication as the result of multiplying two $\frac{d}{2}$ bit numbers would fit in d bits. The result of $\frac{y_1}{2^4} * \frac{y_2}{2^4} = 30$ with scale $2 * 5 - d = 2$, i.e., $\frac{30}{4} \approx \pi * e$.

While these naïve rules of performing fixed-point arithmetic are sufficient to guarantee the absence of overflows, they can result in a significant loss of precision. We present an evaluation of this technique in Section 7.3.2. Our results show that applying these rules to ML benchmarks can result in implementations that return unacceptable results (same classification accuracy as a purely random classifier). We describe this problem further using an example.

3 Motivating Example

We use a linear classifier as a motivating example for our fixed-point compiler and to introduce the SeeDot language. The input to our example classifier described below is a vector $x \in \mathbb{R}^4$ and it returns a label $\ell \in \{true, false\}$. The classifier consists of a model $w \in \mathbb{R}^4$ and it computes $w * x > 0$, where $*$ is the inner-product of two vectors. The following program is how the classifier would be represented in SeeDot for specific values of x and w:

$$\begin{array}{l}\texttt{let}\ x = [0.0767; 0.9238; -0.8311; 0.8213]\ \texttt{in} \\ \texttt{let}\ w = [[0.7793, -0.7316, 1.8008, -1.8622]]\ \texttt{in} \\ w * x\end{array} \tag{1}$$

If we run this program in infinite precision Real arithmetic then $w*x$ evaluates to -3.64214951. Floating-point arithmetic produces the approximately correct answer -3.642149448.

Suppose we represent each Real number using a 8-bit fixed-point number (*bitwidth* = 8), then the best scale for each entry in x and w is 7 and 6 respectively. Choosing larger scales would result in overflows. If we mechanically

[2] We only consider integers with fixed number of bits (e.g., 8, 16, 32, etc.).

[3] If the hardware has support for $2d$-bit multiplication then another option is to extract top d bits of $(y_1 * y_2)$ and discard the lower d bits.

apply the rules described in Section 2.3 then 1) for addition, we must scale down the inputs by 1, and 2) for multiplication, we must scale down the inputs by 4 (half the bitwidth). The resulting fixed-point code will be,

$$\begin{aligned}&\texttt{let } x = [\lfloor 0.0767 * 2^7 \rfloor, \lfloor 0.9238 * 2^7 \rfloor \ldots] \texttt{ in}\\ &\texttt{let } w = [\lfloor 0.7793 * 2^6 \rfloor, \lfloor -0.7316 * 2^6 \rfloor, \ldots] \texttt{ in}\\ &\frac{(\frac{w_1/2^4 * x_1/2^4}{2})+(\frac{w_2/2^4 * x_2/2^4}{2})}{2} + \frac{(\frac{w_3/2^4 * x_3/2^4}{2})+(\frac{w_4/2^4 * x_4/2^4}{2})}{2}\end{aligned} \tag{2}$$

This code loses valuable significant bits and computes an imprecise result of −2.625. In contrast, the code generated by SeeDot does the following computation:

$$\begin{aligned}&\texttt{let } x = [\lfloor 0.0767 * 2^7 \rfloor, \lfloor 0.9238 * 2^7 \rfloor \ldots] \texttt{ in}\\ &\texttt{let } w = [\lfloor 0.7793 * 2^6 \rfloor, \lfloor -0.7316 * 2^6 \rfloor, \ldots] \texttt{ in}\\ &((w_1/2^4 * x_1/2^4) + (w_2/2^4 * x_2/2^4) + \ldots\end{aligned} \tag{3}$$

The computed value of $w * x$ is −98 with a scale of 5 and represents the Real value $\frac{-98}{2^5} = -3.0625$. This value is a significantly better approximation of the ideal result.

Before describing how SeeDot produces the code shown in (3), we describe an (impractical) approach to generate the optimal implementation. Suppose we non-deterministically guess the best scale that every sub-expression needs to have, we can then perform the appropriate scale-up/down operations and obtain the most accurate implementation. This non-determinism can be removed by enumerating over all possible choices of scales for all sub-expressions. This enumeration space is huge and there are over 10^{20} possibilities for our tiny example in (1). In contrast, the size of the enumeration space explored by SeeDot is a small constant independent of the input program (see Section 5.3).

4 SeeDot Design Overview

SeeDot avoids enumerating over all the possibilities by evaluating only a very small heuristically selected subset of this vast enumeration space. To this end, our heuristic identifies a parameter *maxscale*, $\mathcal{P}$, such that the upper bound for the intermediate values is $2^{d-\mathcal{P}-1}$, where d is the bitwidth. Given a $\mathcal{P}$, SeeDot uses $\mathcal{P}$ to avoid scale down operations that lose significant bits. In particular, the operands to addition and multiplication are not scaled down if their scale is below $\mathcal{P}$.

In our example, the magnitude of all intermediate values computed by the expression is less than 4 (= 2^{8-5-1}). Hence, SeeDot uses $\mathcal{P} = 5$ to generate the program in (3). Consider the sub-expression $y_1 + y_2$, where $y_i = w_i/2^4 * x_i/2^4$. Here, y_1 and y_2 both have a scale of 5. SeeDot needs to decide whether a scale down operation needs to be performed. If we are being conservative then this addition can potentially overflow. Therefore, we should perform $\frac{y_1}{2} + \frac{y_2}{2}$, thus decreasing the scale of the result to 4. However, since $\mathcal{P} = 5$, we know that the magnitude of the result is below 4 and can safely be represented using the scale of 5. Thus, we can compute $y_1 + y_2$ without performing the scale down operation and guaranteeing no overflows, thus saving significant bits.

$$\begin{aligned} e \quad ::= \quad & n \mid r \mid M_d \mid M_s \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \\ & \mid e_1 + e_2 \mid e_1 * e_2 \mid e_1 \times e_2 \mid \texttt{exp}(e) \mid \texttt{argmax}(e) \end{aligned}$$

Figure 1. Syntax of the core language of SeeDot

If $\mathcal{P} = 3$ then the intermediate results have a magnitude below $2^{8-3-1} = 16$. Hence, there is a possibility that $y_1 + y_2$ might produce overflows. To avoid this, the scale down operation is performed to reduce the scale to 4.

To identify the best $\mathcal{P}$, SeeDot generates a classifier program for *each* $\mathcal{P}$ in $\{0, 1, \ldots, d-1\}$ and then picks the program that achieves the best classification accuracy on the training set. In particular, the program in (3) corresponds to $\mathcal{P} = 5$ and (2) corresponds to $\mathcal{P} = 3$. For our example program, SeeDot picks $\mathcal{P} = 5$. In general, since ML datasets have outliers, we have observed that using a $\mathcal{P}$ that lets the outliers overflow but preserves significant bits on most inputs leads to better accuracy than using a $\mathcal{P}$ that ensures no overflows on all inputs. Therefore, evaluating all possible choices of $\mathcal{P}$ helps SeeDot pick the best program.

5 Formal Development

SeeDot is a declarative language whose expressions specify computations over Reals. It has been designed for expressing ML inference algorithms. In this section, we describe 1) the syntax of the core language of SeeDot, 2) its type system, and 3) compilation of programs written in SeeDot to fixed-point code. We describe our new implementation for computing exponentials in Section 5.3.1, and our mechanism to determine critical parameters (e.g., $\mathcal{P}$) in Section 5.3.2.

5.1 Syntax

Figure 1 describes the syntax of SeeDot using a grammar. The values in SeeDot are integer scalars n, real scalars r, matrices in dense representation M_d, and matrices in sparse representation M_s. An example of M_d is [[1, 2, 3]; [4, 5, 6]], which represents the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 3 & 4 & 6 \end{bmatrix}$$

A sparse matrix is a record consisting of two lists: a list `val` of non-zero values and a list `idx` of positions of these non-zero values in the matrix. A new identifier x is created using the `let` keyword. Run-time inputs can be modeled by free variables that are not bound by `let`. Free variables of SeeDot expressions get their values and types from environments they are executed and compiled under.

Expressions can be added or multiplied. The operator $*$ represents dense matrix multiplication and $\times$ represents multiplying a two-dimensional sparse matrix with a (dense) vector. Exponentials can be computed via the `exp` keyword and the index of the maximum element of a vector can be obtained using `argmax`.

$$\frac{x \in domain(\Gamma)}{\Gamma \vdash x : \Gamma(x)}\ T-Var \qquad \frac{}{\vdash r : \mathbb{R}}\ T-Real$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}\ T-Let$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \qquad \Gamma \vdash e_2 : \mathbb{R}[n_1, n_2]}{\Gamma \vdash e_1 + e_2 : \mathbb{R}[n_1, n_2]}\ T-Add$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \qquad \Gamma \vdash e_2 : \mathbb{R}[n_2, n_3]}{\Gamma \vdash e_1 * e_2 : \mathbb{R}[n_1, n_3]}\ T-Mult$$

$$\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2]^s \qquad \Gamma \vdash e_2 : \mathbb{R}[n_2]}{\Gamma \vdash e_1 \times e_2 : \mathbb{R}[n_1]}\ T-SparseMult$$

$$\frac{\Gamma \vdash e : \mathbb{R}[n_1, n_2] \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}}\ T-M2S$$

$$\frac{\Gamma \vdash e : \mathbb{R} \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}[n_1, n_2]}\ T-S2M$$

$$\frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \vdash \texttt{exp}(e) : \mathbb{R}}\ T-EXP \quad \frac{\Gamma \vdash e : \mathbb{R}[n]}{\Gamma \vdash \texttt{argmax}(e) : \mathbb{Z}}\ T-ArgMax$$

Figure 2. Type system

The syntax of SeeDot is designed to help the programmer as well as the compiler. In particular, the compiler infers and tracks dimensions of matrices at compile time to determine the appropriate scales and to warn the user about the dimension mismatch errors. Tracking dimensions is difficult in general-purpose languages like Python/C/C++.

The full SeeDot language has additional constructs for reshaping matrices, `for` loops, and CNN [70] specific operators such as convolutions, ReLU, and maxpool. We omit these from Figure 1 as they do not offer additional insights. SeeDot can express any ML model that can be written as a composition of primitive matrix operations. This expressiveness suffices for KB-sized models.

5.2 Static Semantics

We describe the type system of SeeDot in Figure 2. The possible types are the following:

$$\tau ::= \mathbb{Z} \mid \mathbb{R} \mid \mathbb{R}[n_1] \mid \mathbb{R}[n_1, n_2] \mid \mathbb{R}[n_1, n_2]^s$$

A SeeDot expression can have a type integer ($\mathbb{Z}$), or a scalar Real number ($\mathbb{R}$), or a k-dimensional matrix of Reals where $k \in \{1, 2\}$. The type of the matrix in Equation 5.1 is $\mathbb{R}[2, 3]$. Two dimensional sparse matrices with n_1 rows and n_2 columns are assigned the type $\mathbb{R}[n_1, n_2]^s$. We restrict the maximum dimension of matrices to two for the ease of presentation.

We use Γ to denote the typing environment, which is a map from variables to types. The judgement $\Gamma \vdash e : \tau$ is read as follows: under the typing environment Γ, the expression e is well-typed and has a type τ. The rule *T-Var* is standard: a variable x is well-typed if it belongs to the domain of Γ. The rule *T-Let* is also standard and adds variables to Γ. *T-Add* and *T-Mult* ensures that we only add and multiply matrices of compatible dimensions. *T-SparseMult* checks that the two arguments of $\times$ are a two-dimensional sparse matrix and a vector. If a SeeDot developer multiplies or adds matrices with non-compatible dimensions then a compile-time error is generated. The rules *T-M2S* and *T-S2M* coerce a 1×1-matrix to a scalar and vice versa. The `exp` operator takes a scalar argument and returns a scalar result. We only support the exponentiation of scalars. `argmax` returns an integer. These rules are expected from any strongly-typed language for matrix algebra. However, the widely used production DSLs for matrix algebra (e.g., MATLAB) are dynamically typed and can only catch the errors described above at run-time.

A well-typed SeeDot expression can be executed by targeting it to computer algebra systems (e.g., Mathematica [89]) that perform arbitrary precision Real arithmetic. Although exact Real arithmetic is useful for debugging at development time, production systems rely on approximations such as floating-point or fixed-point arithmetic to ensure high efficiency. We describe the fixed-point code generator next.

5.3 Fixed-Point Compilation

The compilation rules provided in Figure 3 translate SeeDot programs to a sequence of procedure calls. The pseudocode for the procedures is described in Algorithm 2. The compilation rules use the auxiliary functions described in Algorithm 1. Note that the compilation rules, the auxiliary functions, and the procedures crucially use the dimensions of the matrices inferred by the type system (the calls to *dim*). The auxiliary functions are parameterized. The functions used in addition and multiplication rules are parameterized by $\mathcal{P}$, which was introduced in Section 4. We use $\mathcal{B}$ to denote the bitwidth. The functions for exponentiation rule require some other parameters ($\mathbb{T}$, m, and M) that are described in Section 5.3.1. In this section, we assume that the compiler has been given a valuation of these parameters by an oracle. Given such a valuation, the compilation rules can be applied to generate a fixed-point implementation statically. We discuss our strategy to set these parameters in Section 5.3.2.

The compilation environment κ maps a variable x to a unique location η and a scale P. The judgment $\kappa \vdash e : (C, \eta, \mathsf{P})$ is read as follows: under an environment κ, an expression e is compiled to a code C, a sequence of procedure calls. The return value of C is stored at location η, which has a scale P. We use ϵ for a no-op code in Figure 3. The rules use these standard functions: function $max(W)$ returns the maximum element of a matrix W; $abs(W)$ returns a matrix containing the magnitude of each entry in the matrix W; *dim* returns the dimensions of a matrix as inferred by the type system.

We discuss Figure 3 using examples. Consider the simple SeeDot program: $\texttt{let}\ x = 1.23\ \texttt{in}\ x$. Compiling this program involves using the rules *C-Let*, *C-Var*, and *C-Val*. *C-Val* uses the auxiliary function GetP to compute the scale of 1.23. If

$$\frac{\kappa(x) = (\eta, \mathsf{P})}{\kappa \vdash x \longrightarrow (\epsilon, \eta, \mathsf{P})}\ C-Var \qquad \frac{\mathsf{P} = \mathrm{GETP}(max(abs(v_1))) \qquad v_2 = \lfloor v_1 * 2^{\mathsf{P}} \rfloor}{\kappa \vdash v_1 \longrightarrow ((\eta = v_2;), \eta, \mathsf{P})}\ C-Val$$

$$\frac{\kappa \vdash e_1 \longrightarrow (C_1, \eta_1, \mathsf{P}_1) \qquad \kappa[x \mapsto (\eta_1, \mathsf{P}_1)] \vdash e_2 \longrightarrow (C_2, \eta_2, \mathsf{P}_2)}{\kappa \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \longrightarrow ((C_1; C_2;), \eta_2, \mathsf{P}_2)}\ C-Let$$

$$\frac{\kappa \vdash \mathsf{e}_1 \longrightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \longrightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \quad (I, J) = dim(\mathsf{e}_1) \quad \mathsf{P}_2 >= \mathsf{P}_1 \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \mathrm{ADDSCALE}(\mathsf{P}_1)}{\kappa \vdash \mathsf{e}_1 + \mathsf{e}_2 \longrightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \mathrm{MATADD}(\eta_1, \eta_2, \mathsf{P}_2 - \mathsf{P}_1, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-MatAdd$$

$$\frac{\begin{array}{c}\kappa \vdash \mathsf{e}_1 \longrightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \longrightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \quad (I, J) = dim(\mathsf{e}_1) \quad (J, K) = dim(\mathsf{e}_2) \\ (\mathsf{P}_{mul}, \mathsf{S}_{mul}) = \mathrm{MULSCALE}(\mathsf{P}_1, \mathsf{P}_2) \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \mathrm{TREESUMSCALE}(\mathsf{P}_{mul}, J)\end{array}}{\kappa \vdash \mathsf{e}_1 * \mathsf{e}_2 \longrightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \mathrm{MATMUL}(\eta_1, \eta_2, \mathsf{S}_{mul}, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-MatMul$$

$$\frac{\begin{array}{c}\kappa \vdash \mathsf{e}_1 \longrightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad \kappa \vdash \mathsf{e}_2 \longrightarrow (\mathsf{C}_2, \eta_2, \mathsf{P}_2) \quad (I, J) = dim(\mathsf{e}_1) \\ (\mathsf{P}_{mul}, \mathsf{S}_{mul}) = \mathrm{MULSCALE}(\mathsf{P}_1, \mathsf{P}_2) \quad (\mathsf{P}_3, \mathsf{S}_{add}) = \mathrm{TREESUMSCALE}(\mathsf{P}_{mul}, J)\end{array}}{\kappa \vdash \mathsf{e}_1 \times \mathsf{e}_2 \longrightarrow ((\mathsf{C}_1; \mathsf{C}_2; \eta_3 = \mathrm{SPARSEMATMUL}(\eta_1, \eta_2, \mathsf{S}_{mul}, \mathsf{S}_{add});), \eta_3, \mathsf{P}_3)}\ C-SparseMatMul$$

$$\frac{\kappa \vdash \mathsf{e} \longrightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1) \quad (T_f, T_g, \mathsf{P}_{11}, \mathsf{P}_{12}, k) = \mathrm{EXPTABLE}(\mathsf{P}_1, m, M) \quad (\mathsf{P}_2, \mathsf{S}_{mul}) = \mathrm{MULSCALE}(\mathsf{P}_{11}, \mathsf{P}_{12})}{\kappa \vdash \mathsf{exp}(\mathsf{e}) \longrightarrow ((\mathsf{C}_1; \eta_2 = \mathrm{EXP}(\eta_1 - m, T_f, T_g, \mathsf{S}_{mul}, k);), \eta_2, \mathsf{P}_2)}\ C-Exp$$

$$\frac{\kappa \vdash \mathsf{e} \longrightarrow (\mathsf{C}_1, \eta_1, \mathsf{P}_1)}{\kappa \vdash \mathsf{argmax}(\mathsf{e}) \longrightarrow ((\mathsf{C}_1; \eta_2 = \mathrm{ARGMAX}(\eta_1);), \eta_2, 0)}\ C-ArgMax$$

Figure 3. Compilation rules

the bitwidth $\mathcal{B} = 16$, i.e., 16-bit integers are used to represent the Real numbers, then GETP returns 14 and the value 1.23 compiles to $\eta = 20152$, where $20152 = \lfloor 1.23 * 2^{14} \rfloor$. The rule *C-Let* updates the environment to map x to $(\eta, 14)$. The rule *C-Var* for variables is standard where under an environment κ, the variable x compiles to an empty program and the return value of the expression is stored at the location η with scale 14. Overall, this example compiles to $\eta = 20152; \epsilon$, where the return value in η has a scale of 14.

If our example is $\texttt{let } x = 1.23 \texttt{ in } x + x$ then the rule *C-MatAdd* is applicable. This example compiles to $\eta = 20152; \eta_1 = MatAdd(\eta, \eta, 0, 1)$, where the return value in η_1 has a scale of 13. The scales P_1 and P_2 in *C-MatAdd* are the scales of x, i.e., 14. The function ADDSCALE computes the scale of the result of addition, which is first set to $14 - 1 = 13$. Recall, addition can result in larger numbers that can overflow. Hence, the scale needs to be reduced. Accordingly, S_{add} specifies that both the arguments of addition need to be divided by $2^{\mathsf{S}_{add}} = 2$ before adding them together. However, if $\mathcal{P}$ is large then there would be no need to reduce scale and S_{add} would be zero. The compiled fixed-point code evaluates to 20152 with a scale of 13, i.e., 2.4599609375, which is a good approximation of the exact result 2.46.

Dense and sparse matrix multiplications follow the same pattern as addition. In particular, the intermediate results need to be scaled down before they are added or multiplied. An intermediate step of matrix multiplication requires summation over a sequence of values. We use the TREESUM procedure of Algorithm 2 that minimizes the precision loss.

5.3.1 Computing Exponentials

There are two standard techniques for computing e^x: either compute approximations in floating-point or use a look-up table [40, 78]. Both of these approaches are unsatisfactory for constrained devices. The former approach [78] simulates floating-point in software and has high latency. The latter approach [40] has low latency but consumes a lot of memory. In particular, a look-up table for 16-bit fixed-point arithmetic would have 2^{16} entries of 16-bits each and consumes 128 KB. This table cannot fit on the KB-sized resource-constrained devices that we consider in this paper. In this paper, we propose an approach that uses only 0.25KB of memory.

At a high level, our approach implements e^x as a product of two values that are looked up from two tables. Specifically, we first divide x into four parts as shown in Figure 4: the sign bit, two parts of $\mathbb{T}$ each (a and b), and remaining least

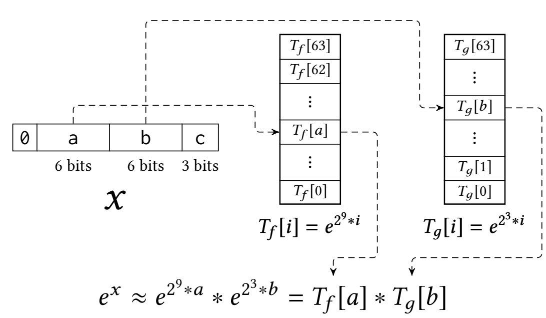


Figure 4. Computing e^x using pre-computed tables T_f and T_g, where x is a positive 16-bit fixed-point number.

Algorithm 1 Auxiliary functions

```
function GetP(n)
    return (ℬ - 1) - ⌈log_2(n)⌉
function MulScale(P_1, P_2)
    S_mul ← ℬ
    P_mul ← (P_1 - S_mul / 2) + (P_2 - S_mul / 2)
    if P_mul <= 𝒫 then
        S_mul ← max(ℬ - (𝒫 - P_mul), 0)
        P_mul ← (P_1 - S_mul / 2) + (P_2 - S_mul / 2)
    return P_mul, S_mul
function AddScale(P)
    S_add ← 1
    P_add ← P - 1
    if P_add <= 𝒫 then
        S_add ← 0
        P_add ← P
    return P_add, S_add
function TreeSumScale(P, n)
    S_add ← ⌈log_2(n)⌉
    P_add ← P - S_add
    if P_add <= 𝒫 then
        S_add ← max(S_add - (𝒫 - P_add), 0)
        P_add ← P - S_add
    return P_add, S_add
function ExpTable(P, m, M)
    k ← ⌈log_2(M − m)⌉
    P_1 ← GetP(e^m) , P_2 ← GetP(1)
    for i in 0:2^𝕋 do
        T_f[i] ← ⌊e^((m+2^(k−𝕋)*i)/2^P) * 2^(P_1)⌋
        T_g[i] ← ⌊e^((2^(k−2*𝕋)*i)/2^P) * 2^(P_2)⌋
    return T, P_1, P_2, k
```

significant k bits (c). Assuming x to be positive, we have,

$$x = 2^{\mathbb{T}+k}a + 2^k b + c$$

$$\Rightarrow\; e^x = e^{2^{\mathbb{T}+k}a+2^k b+c} \;\approx\; e^{2^{\mathbb{T}+k}a}.e^{2^k b} \;=\; f(a).g(b)$$

where $f(a) = e^{2^{\mathbb{T}+k}a}$ and $g(b) = e^{2^k b}$. Our idea is to implement the functions f and g using two look-up tables, T_f and T_g, respectively. For bitwidth $\mathcal{B} = 16$, $\mathbb{T} = 6$, and $k = 3$, each table has 64 entries with 2 bytes each. Therefore, the total cost of the two tables is just 256 bytes. We can use the same approach and use two additional tables to compute e^x for negative values of x. We present a detailed evaluation of our approach in Section 7.2.

5.3.2 Auto-Tuning Parameters

There are five parameters in SeeDot compiler: $\mathcal{P}$, $\mathcal{B}$, $\mathbb{T}$, m, and M. We show how these parameters are set. In our evaluation, we keep $\mathbb{T} = 6$ and the other parameters need to be explored. There are two main strategies for setting parameters: brute force and run-time profiling. In the former,

Algorithm 2 Codegen procedures

```
procedure MatMul(A, B, S_mul, S_add)
    (P, Q) ← dim(A), (Q, R) ← dim(B)
    var T[Q], C[P, R]
    for i in 0:P do
        for j in 0:R do
            for k in 0:Q do
                var a ← A[i][j] / 2^(S_mul/2)
                var b ← B[i][j] / 2^(S_mul/2)
                T[k] ← a * b
            C[i][j] ← TreeSum(T, S_add)
    return C
procedure SparseMatMul(A, B, S_mul, S_add)
    (P, Q) ← dim(A), Q ← dim(B)
    var C[P, 1], i_idx ← 0, i_val ← 0
    for i in 0:Q do
        j ← A.idx[i_idx++]
        while j != 0 do
            var u ← 2^(S_mul/2)
            var t ← (A.val[i_val++] / u) * (B[i] / u)
            C[j-1][0] ← C[j-1][0] + (t / 2^(S_add))
            j ← A.idx[i_idx++]
    return C
procedure TreeSum(A, S_add)
    n ← dim(A)
    var k ← n/2, s ← 1
    while k > 1 do
        if S_add- - <= 0 then s ← 0
        for i in 0:k do
            A[i] ← (A[2*i] / 2^s) + (A[2*i+1] / 2^s)
        if n%2 != 0 then A[k] ← A[2*k] / 2^s
        n ← (n+1)/2, k ← n/2
    return A
procedure MatAdd(A, B, n, S_add)
    P, Q ← dim(A)
    var C[P, Q]
    for i in 0:P do
        for j in 0:Q do
            C[i][j] ← (A[i][j] / 2^(S_add)) + (B[i][j] / 2^(n+S_add))
    return C
procedure Exp(x, T_f, T_g, S_mul, k)
    var i ← bits_𝕋(x, k), j ← bits_𝕋(x, k − 𝕋)
    var e ← (T_f[i] / 2^(S_mul/2)) * (T_g[j] / 2^(S_mul/2))
    return e
procedure ArgMax(A)
    P ← dim(A)
    var index ← 0, max ← A[0]
    for i in 0:P do
        if A[i] > max then
            max ← A[i], index ← i
    return index
```

we exhaustively try all possible values of the parameter and choose the one that works the best. This evaluation is performed by measuring classification accuracy on the *training* set. In the latter, we observe the runs of the ML classifier on the training data and set the parameters according to the observations. Brute force provides optimal performance and ideally, we want to set all parameters this way.

We set the following parameters by brute force: the bitwidth $\mathcal{B}$, i.e., the number of bits assigned to each variable and the maxscale $\mathcal{P}$. In particular, for $\mathcal{B} = 16$, SeeDot generates 16 programs for each $\mathcal{P}$ value from 0 to 15 and chooses the $\mathcal{P}$ with highest accuracy. Note that the number of programs generated by SeeDot is constant and independent of the size of the input program. The time taken for each exploration step is dependent on the size of the training set and is usually within a couple of minutes. Although we would like to set (m, M), the range of inputs for e^x, via brute force as well, this would increase the enumeration space significantly. Therefore, (m, M) are set by profiling. In particular, we run the SeeDot program using floating-point arithmetic on the training set. We monitor the calls to exponentiation and select a small range in which *most* (more than 90%) of the inputs lie. By excluding the outliers, this process produces satisfactory implementations.

6 Accelerating SeeDot Using FPGA

Field Programmable Gate Arrays (FPGA) are re-configurable chips that can be used to build custom hardware accelerators for important applications. FPGAs offer better performance and power efficiency compared to general-purpose processors; Unlike Application-Specific Integrated Circuits (ASICs), FPGAs can be *reprogrammed* to handle updates to algorithms. These features make them a natural choice for IoT devices. To this end, we explore the potential for accelerating SeeDot programs using FPGAs.

6.1 SeeDot to FPGA: Overview

Traditionally, FPGA programmers write code in Hardware Description Languages (HDL) like VHDL [44] or Verilog [45]. This process requires significant expertise in digital design and is extremely time-consuming. To improve programmer productivity, FPGA vendors have developed High-Level Synthesis (HLS) tools that can compile programs written in a language like C directly to Verilog code. A simple approach to compiling a SeeDot program to Verilog is to directly feed the SeeDot-generated fixed-point C code as input to the HLS tool. Although this approach significantly outperforms Arduino Uno (on an FPGA of comparable power consumption), it does not fully utilize the FPGA resources. To address this underutilization problem, we present two optimizations.

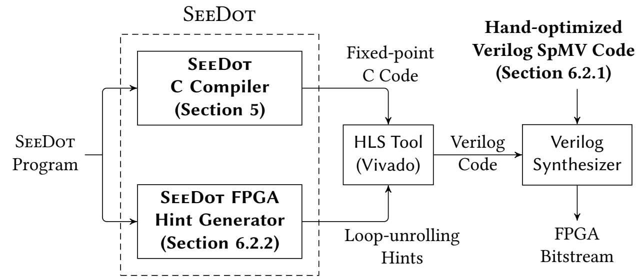


Figure 5. Flowchart for compiling SeeDot to FPGA

6.2 Optimizations to Improve FPGA Utilization

Our first optimization exploits the fact that ML algorithms heavily use Sparse-Matrix Vector (SpMV) multiplications. To this end, we use a hand-optimized Verilog code to perform SpMV multiplications, thereby reducing the execution time. Our second optimization automatically generates loop-unrolling hints for the HLS compiler, thereby enabling better utilization of the FPGA resources. Figure 5 shows the flowchart for compiling a SeeDot program to FPGA bitstream.

6.2.1 Accelerating SpMV Multiplication

Many ML algorithms use sparse matrices to compress the models [2, 27, 93]. In our evaluation, we observe that SpMV multiplication (the × operator of Section 5) consumes a significant fraction of the execution time (56% on average). To accelerate this operation, we implemented it in Verilog. Our implementation creates multiple *processing elements* (PEs) on the FPGA where each PE can perform one fixed-point multiply-accumulate operation per cycle. The columns in the sparse matrix are partitioned and assigned to PEs to compute the result. To avoid workload imbalance across the PEs, a small portion (about a quarter) of matrix columns is retained for dynamic assignment to PEs which complete the work first. The remaining portion is assigned statically. Our implementation of SpMV multiplication is significantly faster, 2.6×–14.9×, than the version generated by the HLS compiler.

6.2.2 Hints to the HLS Compiler

The HLS compiler allows programmers to provide hints to exploit the parallelism offered by FPGAs. However, prior work [24] has shown that inserting these hints requires some knowledge of hardware design, which many ML experts do not possess. Our second optimization automatically generates loop unrolling hints (`#pragma HLS UNROLL`) for the HLS compiler, thereby increasing the parallelism of the generated Verilog code. To exploit loop unrolling, we must determine 1) the loops in which the iterations are independent of each other, and 2) the degree of unrolling for each such loop. While loop unrolling improves parallelism, it also increases resource utilization. Unrestricted unrolling can result in the generated-code exceeding the resource budget on the FPGA.

Our FPGA hint generator has two aspects that are enabled by SeeDot. First, as the SeeDot program specifies operations at a high level (e.g., matrix multiplication), the hint generator can easily identify loops in the generated C-code that have no data dependence between iterations. In contrast, this analysis is harder for programs written directly in C. Second, determining the unrolling factor for each "*for*" loop so as to minimize the overall execution time is a complex optimization problem. In this preliminary exploration, we devise a simple heuristic that sequentially unrolls each loop as much as possible as long as the generated FPGA-code is within the resource budget. This heuristic is feasible to implement as the compiler knows the dimension of all matrices. The hint generator statically estimates the resource usage of operations (number of required configurable logic blocks) and then computes the unroll factor for each operation.

For example, consider a SeeDot program with a matrix subtraction followed by a matrix addition as shown below, where A, B, and C are 10×1 vectors.

$$\texttt{let } D = A - B + C \texttt{ in } D$$

During compilation, the hint generator knows that the addition and subtraction operations are independent and can be executed in parallel. Then, the hint generator determines the loop unrolling factors as follows. Consider the available resource on the FPGA as 'R', and the estimated resource usage of each iteration of subtraction and addition as $0.4 \times$ R and $0.1 \times$ R respectively. First, the hint generator greedily assigns the maximum unroll factor, 10, for A-B and computes the resource usage as $10 \times 0.4 \times$ R. Since the resource usage exceeds R, the unrolling factor is progressively reduced to bring the resource usage less than R. Thus, an unroll factor of 2 is computed for A-B and uses $0.8 \times$ R resources. Further, the hint generator applies the same heuristic to the next operation, matrix addition with C, with the remaining $0.2 \times$ R resources and computes the unroll factor as 2. The generated C-code with annotations for the unroll factors is as follows.

```
for (int i=0; i<10; i++)
    #pragma HLS UNROLL factor=2
    temp[i] = A[i] - B[i];
for (int i=0; i<10; i++)
    #pragma HLS UNROLL factor=2
    D[i] = temp[i] + C[i];
```

Our simple heuristic significantly improves resource utilization and consequently the performance of the generated FPGA code (Section 7.3.1).

7 Evaluation

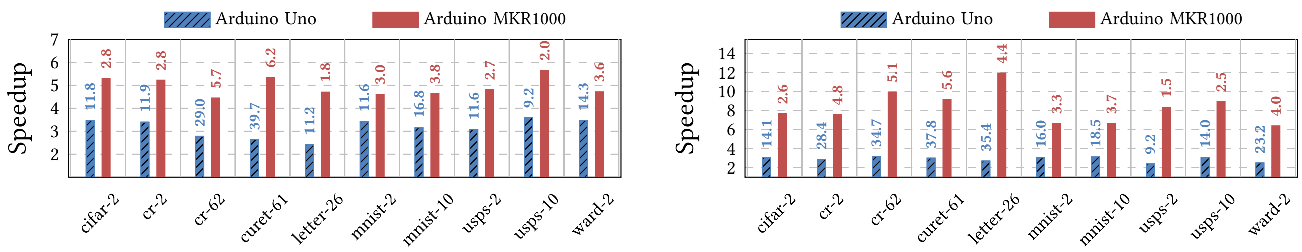
We evaluate SeeDot in three different settings: Arduino boards, FPGAs, and real IoT devices. Through empirical evaluation, we aim to justify the following claims:

- SeeDot-generated fixed-point code is much more efficient than emulating floating-point in software. In particular, we compare the performance of fixed-point and floating-point code on two Arduino boards (Uno and MKR1000).
- SeeDot's novel compilation strategy beats state-of-the-art float-to-fixed converters in compiling KB-sized ML models to resource-constrained devices. We show that SeeDot-generated code for Arduino Uno has much better performance than commercial MATLAB toolboxes that cost more than $30000 per license to achieve the same task. We also compare the performance of SeeDot-generated code with TensorFlow-Lite, a framework to generate efficient code for smartphones and embedded devices, and observe significant performance improvements.
- SeeDot's novel approach to compute exponentiation is much more efficient than the state-of-the-art approaches that compute approximations in floating-point.
- SeeDot generates FPGA implementations that are much more efficient than both microcontroller-based implementations and FPGA implementations obtained using high-level synthesis (HLS) tools directly. Moreover, SeeDot-generated fixed-point code for FPGAs performs significantly better than traditional fixed-point schemes.
- SeeDot can express various ML inference algorithms. In particular, SeeDot can express recently-published ML classifiers for constrained devices as well as convolution neural networks (CNNs) used in computer vision tasks.
- SeeDot's novel compilation technique to generate fixed-point code results in a minimum loss in accuracy. We show that exploring multiple programs with different maxscale values improves the precision of the generated code.
- SeeDot is helpful in the real-world and improves the performance of IoT devices used in the wild. We consider devices deployed on agricultural farms and pods attached to white canes of persons with visual impairments.

We use Arduino Uno and MKR1000 for our evaluation. The Uno has an 8-bit, 16-MHz Atmega328P microcontroller, with 2KB of SRAM and 32KB of read-only flash memory. MKR1000 has more powerful hardware: a 32-bit 48-MHz ARM Cortex-M0+ microcontroller, 32KB of SRAM and 256KB of read-only flash. These devices are much more resource-constrained than the floating-point equipped embedded devices considered in prior work (e.g., Raspberry Pi [66], ARMv7 [32], etc.).

The FPGA device we target is the Xilinx Arty board which has 225KB of on-chip memory, 5200 logic slices consisting of 20800 LUTs and a peak operating frequency of 450MHz. Prior systems that run ML on FPGAs require devices with much richer capabilities (e.g., Zynq [80], Virtex [28], Stratix [10] etc.). For synthesis, we use Xilinx's Vivado HLS tool. We use 10 standard ML datasets that have been used by [30, 56]: cifar [54], character recognition (cr) [18], curet [87], letter [41], mnist [59], usps [43], ward [92], and binary classification tasks of cr, mnist and usps datasets from [50].

We consider three types of KB-sized ML classifiers: Bonsai [56], ProtoNN [30], and CNNs [70]. Note that Bonsai



(a) Bonsai model. **(b) ProtoNN model.**

Figure 6. Speedup of SeeDot-generated fixed-point code over hand-written floating-point code.

and ProtoNN are the only known classifiers that are compact enough to fit in the tiny memories of Arduino Unos. Bonsai is a tree-based classification algorithm which learns a shallow and sparse tree. ProtoNN is a k-means style algorithm. The mathematical description of the classifiers and the corresponding SeeDot code can be found in the supplementary material. We trained Bonsai and ProtoNN on 10 different datasets and learned 20 different models. We present our evaluation on these 20 models.

7.1 Arduino Evaluation

We compare SeeDot-generated fixed-point code against floating-point code, MATLAB-generated fixed-point code, and post-training quantization of Tensorflow-Lite.

7.1.1 Comparison with Floating-Point

Emulating floating-point operations in software is inefficient. On an Uno, addition and multiplication operations on integers are 11.3× and 7.1× faster than the respective floating-point operations. This results in high performance of the SeeDot-generated code. Figure 6a and Figure 6b show the speedup of SeeDot-generated implementations for Bonsai and ProtoNN respectively over the baseline floating-point implementations from [30, 56]. The text on each bar shows the absolute execution time of SeeDot-generated code in milliseconds. The size of all models is within 32KB and they fit on both Uno and MKR. The mean speedup for Bonsai is 3.1× on Uno and 4.9× on MKR. For ProtoNN, the mean speedup is 2.9× on Uno and 8.3× on MKR. Thus, fixed-point implementations are much more efficient for both these ML inference algorithms. The average loss in classification accuracy on the *testing* set caused by using fixed-point arithmetic for Bonsai is 0.345% on Uno and 0.127% on MKR. Similarly, for ProtoNN, the loss is 1.855% and 0.051% respectively. The MKR implementations are more precise because they use 32-bit integers and the Uno implementations use 16-bit integers. We note that, in most cases, the MKR implementations have better classification accuracy than the corresponding floating-point implementations. The average accuracy loss reported above includes only the cases where the floating-point implementations are more precise.

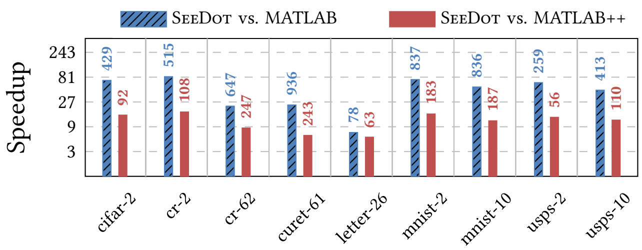
These stark performance improvements are actually not surprising. Custom fixed-point implementations of Bonsai and ProtoNN are known to outperform floating-point implementations [30, 56]. Such custom implementations are obtained after careful manual fine-tuning of scales and this effort needs to be repeated for each dataset. In contrast, SeeDot is fully automatic and improves developer productivity. Moreover, SeeDot-generated code is comparable in performance to the custom handwritten implementations [30, 56].

7.1.2 Comparison with MATLAB

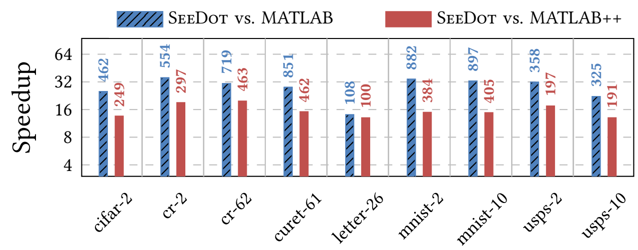
We now compare SeeDot with existing float-to-fixed converters. Most frameworks to compile ML models to fixed-point code do not target KB-sized microcontrollers and are irrelevant for comparison purposes. To the best of our knowledge, MATLAB is the only tool that compiles KB-sized ML models to fixed-point code for Arduino Uno. We use the following MATLAB toolboxes: MATLAB Coder, Embedded Coder, and Fixed-Point Designer. MATLAB uses arithmetic operations over large bitwidths to guard against overflows. Although this approach is good for DSPs, performing such operations on microcontrollers causes huge slowdowns.

Most implementations of ML algorithms support special representations of sparse matrices for performance. However, the Fixed-Point designer toolbox of MATLAB lacks support for sparse matrices which results in the generation of inefficient fixed-point code. On the other hand, SeeDot has language support for sparse matrices. As a side contribution, to be more fair to the techniques being used by MATLAB, we spent significant development effort in adding support for sparse matrices in the MATLAB tool-chain. This improves the performance of MATLAB-generated code by up to 4.8×.

Figure 7a and Figure 7b use the MATLAB-generated fixed-point code for Bonsai and ProtoNN as the baseline and shows the speedups of SeeDot-generated code on Uno. The text on each bar shows the absolute execution time of MATLAB-generated code in milliseconds. The y-axis is in log scale. MATLAB++ represents MATLAB with sparse matrix support. Without sparse matrix support, the mean speedup is 51× for Bonsai and 28.2× for ProtoNN. With sparse matrix support, the speedups are still quite high with



(a) Bonsai model.



(b) ProtoNN model.

Figure 7. Speedup of SeeDot-generated code over MATLAB-generated fixed-point code on an Arduino Uno. MATLAB++ denotes MATLAB with sparse matrix support.

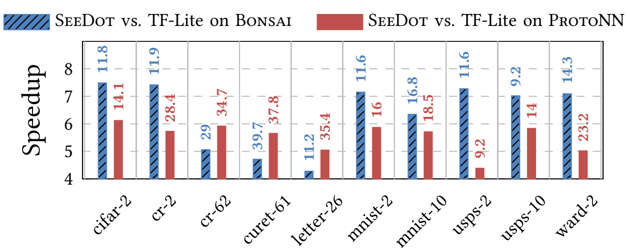


Figure 8. Speedup of SeeDot-generated code with TensorFlow-Lite-generated code on an Arduino Uno.

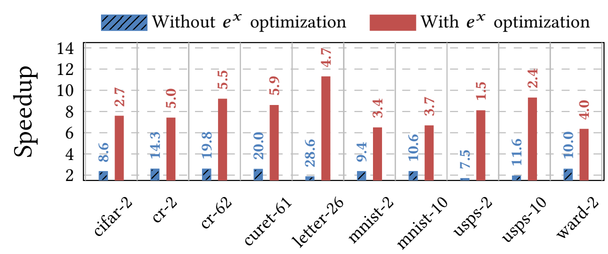


Figure 9. Performance of the SeeDot-generated code for ProtoNN with and without the exponentiation optimization described in Section 5.3.1 on MKR1000.

a mean speedup of 11.6× for Bonsai and 15.6× for ProtoNN. We note that, in some cases, the classification accuracy of MATLAB-generated code is extremely poor (similar to that of a purely random classifier). In contrast, SeeDot-generated implementations have comparable accuracy to the floating-point code for all classifiers.

7.1.3 Comparison with TensorFlow-Lite

TensorFlow-Lite (TF-Lite) [84] is TensorFlow's low resource footprint solution for running ML on smartphones and embedded devices. Using TF-Lite, a developer can deploy ML models trained using TensorFlow directly on smartphones. TF-Lite also provides "post-training-quantization" that converts trained floating-point models to 8-bit tensors. We compare SeeDot and post-training-quantizer of TF-Lite next.

A direct comparison between these two approaches is hard as they target different hardware. TF-Lite focuses on devices having MB/GB sized memories. The TF-Lite runtime itself is a few MBs and cannot fit on the KB-sized microcontrollers. In particular, TF-Lite compiled binary for Raspberry-Pi is 2.9MB. For comparison purposes, we have translated TF-Lite's quantized models [85] to C. These C models are standalone and do not need the TF-Lite runtime for execution.

TF-Lite uses a hybrid approach for quantization. The quantized tensors are converted to floating-point while performing arithmetic operations. Hence, arithmetic operations of TF-Lite code are all performed in floating-point. For devices without floating-point support, the overhead of floating-point operations and integer-to-float conversions is large.

Figure 8 shows the comparison of SeeDot-generated code and our TF-Lite implementation for Bonsai and ProtoNN. The numbers on top of each bar show the absolute execution time of TF-Lite generated code in milliseconds. The observed average speedup is 6.4× and 5.5× for Bonsai and ProtoNN respectively. The high speedups are due to the floating-point arithmetic operations performed by TF-Lite. Moreover, since TF-Lite performs integer-to-float operations at runtime, its performance is worse than our floating-point baseline described in Section 7.1.

7.2 Exponentiation Evaluation

Existing approaches for computing e^x compute approximations in floating-point. We evaluate our approach described in Section 5.3.1 against two approaches: `math.h` implementations in Arduino IDE and fast exponentiation technique in [78]. We ran the three implementations on 100 random inputs on an Arduino Uno, and recorded the average time per e^x computation. SeeDot performs 23.2× faster than `math.h` implementation, which performs an inefficient simulation of floating-point in software. The fast exponentiation technique [78] uses a clever floating-point-based technique to reduce computation and performs significantly better than `math.h`. However, since the computation is still in floating-point, SeeDot outperforms this implementation by 4.1×.

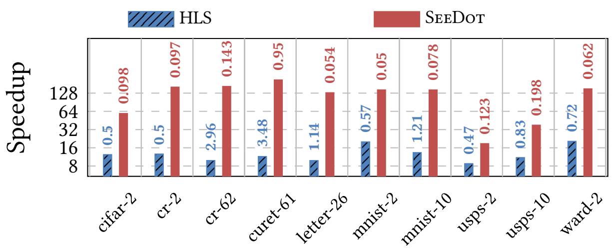


Figure 10. Performance of FPGA implementations generated by HLS and SeeDot (with our optimizations) for Bonsai, with SeeDot-generated Uno implementations as baseline.

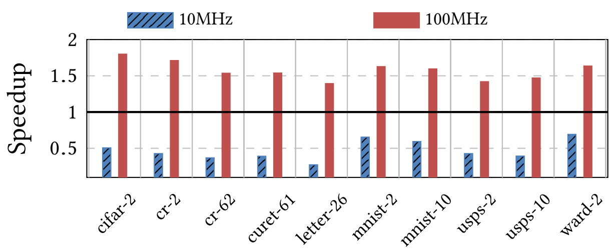


Figure 11. Performance of FPGA implementations for ProtoNN generated by SeeDot (w/o our optimizations) at 10MHz and 100MHz, with HLS as baseline.

Figure 9 shows the performance improvement of using our exponentiation technique in SeeDot-generated code for ProtoNN on an MKR1000. The numbers on top of each bar show the absolute execution time in milliseconds. The speedups in blue use `math.h` for computing e^x. The increase in speedup from the exponentiation technique is 3.8×–9.4×.

7.3 FPGA Evaluation

This section describes our experience of accelerating ML models on low-end FPGAs. We delve into a preliminary evaluation of SeeDot's potential to accelerate KB-sized models.

7.3.1 Comparison with HLS Tools

We compare SeeDot-generated FPGA implementations with Uno implementations described in the previous section and handwritten floating-point Vivado HLS C code.

The evaluation on Bonsai models is shown in Figure 10. The results for ProtoNN are similar and are omitted. The FPGA implementations are bit-wise equivalent to the Uno implementations and have the same classification accuracy. The text on each bar shows the absolute execution time of the baseline HLS-generated and SeeDot-generated code respectively in milliseconds. We observe that the FPGA implementations are 33.1×–235.7× faster than the Uno implementations. To the best of our knowledge, this is the first empirical comparison of KB-sized ML models running on constrained devices versus low-end FPGAs. We observe that acceleration

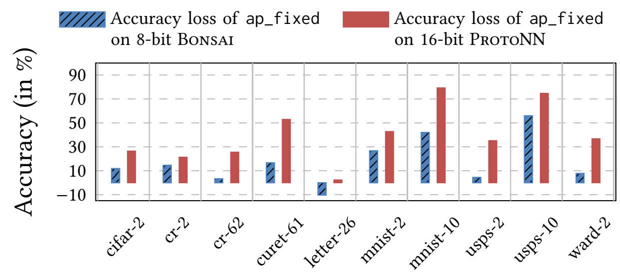


Figure 12. Comparison of accuracy loss of HLS baseline with ap_fixed type with SeeDot-generated code.

using FPGAs can provide significant benefits in this setting. The optimizations described in Section 6 have a significant impact on performance. In particular, the SeeDot-generated FPGA implementation with loop-unrolling hints and hand optimized sparse multiplication are 3.6×–21× faster than the HLS-generated floating-point FPGA implementations.

At the clock frequency of 10MHz that we use in this evaluation, both a floating-point operation and a fixed-point operation take one clock cycle. The benefits at higher clock frequencies are even larger. At higher frequencies, floating-point operations consume multiple cycles whereas fixed-point operations can still be completed within a single cycle. For instance, consider fixed-point ProtoNN code generated by SeeDot with all optimizations described in Section 6 disabled. At low clock frequencies, we expect this fixed-point code to be slower as it performs more number of operations than a floating-point code. Indeed, at 10MHz, this code is about 2× slower than the HLS implementation (Figure 11). However, at 100MHz, the same code is about 1.5× faster.

These promising results demonstrate that SeeDot can generate programs for low-end/low-cost FPGAs, thereby laying an initial groundwork for targeting MB/GB-sized ML (e.g., compressed/quantized/binary neural nets [1, 26, 34, 35, 38, 42, 61, 95]) and server-grade FPGAs in the future.

7.3.2 Comparison with HLS Fixed-Point Types

We evaluate the arbitrary precision `ap_fixed` type from the fixed-point library provided by Vivado HLS [91]. In this library, `ap_fixed<W,I>` represents a fixed-point type, where `W` is the word length, and `I` is the number of bits representing the integer part. For example, `ap_fixed<8,6>` represents a Real number r as an 8-bit integer $\lfloor r \cdot 2^{(8-6)} \rfloor$. We use the default quantization (truncation) and overflow (wrap around) modes. A developer can use this type instead of floating-point to reduce resource utilization and latency on FPGAs. In this evaluation, we compare the classification accuracy of `ap_fixed` type and SeeDot-generated code. We replace the floating-point type in our HLS baseline with `ap_fixed`. Then, for a particular bitwidth `W`, we evaluate different configurations of `ap_fixed<W,I>` by sweeping `I` from 0 to (`bitwidth-1`). We evaluate each configuration on the

Table 1. Comparison of SeeDot-generated code with floating-point code for LeNet models on MKR1000.

Model size	Bitwidth	Accuracy loss	Speedup
50K	16	2.45%	2.5×
50K	32	0.00%	3.3×
105K	16	1.16%	∞

testing set to record classification accuracy and then report the configuration with the best accuracy.

Figure 12 compares the classification accuracy loss of the `ap_fixed` type with SeeDot-generated code for Bonsai and ProtoNN across various datasets. For ProtoNN, 16-bit `ap_fixed` type loses 39.69% accuracy on average. In most cases, `ap_fixed` type has trivial accuracy (~50% for binary and ~10% for decimal classification tasks). However, 32-bit `ap_fixed` type achieves comparable accuracy with SeeDot. The trend with Bonsai models is similar: 8-bit `ap_fixed` type loses 17.26% accuracy on average, and 16-bit `ap_fixed` type has comparable accuracy. Thus, at lower bitwidths, SeeDot-generated code significantly outperforms `ap_fixed` type. These results highlight the drawback of traditional fixed-point arithmetic that quickly loses precision.

7.4 Expressiveness

SeeDot can express a variety of ML models. Bonsai and ProtoNN can be expressed in 11 lines and 5 lines of SeeDot code respectively. Since SeeDot provides language support for standard operations in linear algebra, we believe that it can express most ML inference algorithms.

To demonstrate the expressiveness of SeeDot, we implemented a KB-sized convolution neural network (CNN) [70] in SeeDot. CNNs are widely used in computer vision and are being deployed on embedded devices for various applications: pedestrian detection [82], enhancing driver safety [65, 79], traffic management [62, 96], etc. For our evaluation, we use LeNet [83], a popular CNN architecture, which passes an input image through a number of convolution layers followed by a number of fully-connected layers. We trained KB-sized LeNet models for the CIFAR-10 dataset and deployed them on an MKR. Since these models are large, they did not fit on an Uno. CIFAR-10 requires labeling RGB images with ten possible labels (e.g., cat, dog, truck, etc.) and is one of the most widely used datasets in computer vision [20, 33, 55]. LeNet can be expressed in 10 lines of SeeDot code, whereas the hand-written C code is several hundred lines long.

Table 1 summarizes the results on two LeNet models with different sizes. On the smaller model with 50K parameters, SeeDot-generated 16-bit fixed-point code performs 2.5× better than the baseline floating-point code with a small loss in accuracy (2.45%). To obtain better precision, we tested the model with SeeDot-generated 32-bit fixed-point code which has no accuracy loss and performs 3.3× better. For testing the larger network with 105K parameters, the floating-point model is too large to fit on an MKR. Therefore, we measured its accuracy by running it on an x86 processor. In contrast, the fixed-point model can fit on an MKR. To the best of our knowledge, this is the first implementation of a KB-sized ML inference algorithm running on such a small microcontroller that provides high accuracy (above 70%) on a practical computer vision task. Therefore, we believe that SeeDot can facilitate new ML applications in the future.

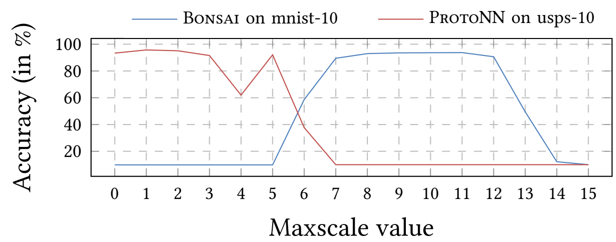


Figure 13. Significance of the maxscale parameter on the accuracy of the generated fixed-point code.

7.5 Significance of Maxscale

In this section, we study how accuracy varies with the maxscale parameter. For a given SeeDot program, the compiler generates multiple fixed-point programs with different values for the maxscale parameter. We measure the classification accuracy of the generated program using the training set. Figure 13 shows the accuracy of the Bonsai model on `mnist-10` and the ProtoNN model on `usps-10` across various maxscale values. For ProtoNN, SeeDot achieves maximum accuracy at maxscale=8. For Bonsai, there is a huge change in accuracy for maxscale=3,4,5. Thus, the accuracy of the generated fixed-point code depends heavily on the maxscale parameter and exploring it is critical to generating programs with minimum accuracy loss.

7.6 Real-World Case Studies

Our evaluation till now has focused on standard ML datasets. Next, we show how SeeDot improves the performance of ML inference algorithms that have been deployed on real IoT devices using two case studies.

7.6.1 Farm Sensors

Chakraborty et al. [11] have recently deployed 20 IoT devices on a few agricultural farms to enable data-driven farming. Each device contains multiple sensors, deployed at different soil depths, to collect soil moisture and soil temperature data. Given the likelihood of sensor failures, it is necessary to ensure the fidelity of the collected data. Hence, the device contains an Arduino Uno which runs ML inference to detect whether some sensor has malfunctioned. Since the farms are large, the devices neither have network connectivity nor are connected to power supplies. Thus, the devices need to be power efficient and use constrained hardware like the Uno.

The deployed devices use a floating-point ProtoNN classifier that can detect sensor failures with an accuracy of 96.9%. For this classifier, the SeeDot-generated code uses 32-bit integers and has an accuracy of 98.0%, which is higher than the floating-point classifier. Moreover, the SeeDot-generated code is 1.6× faster than the floating-point implementation.

7.6.2 Interactive Cane

Gesturepod [73] is an IoT device that can be attached to white canes carried by people with visual impairments (VIs). When a person makes a gesture with the cane, e.g., taps it twice on the ground, the pod uses ML to recognize the gesture and communicates it to a smart-phone app. The smart-phone can then perform a task, e.g, read the recent notifications. User studies with 12 people with VIs have shown that the pod can recognize gestures with high accuracy and help complete smart-phone tasks up to 9 times faster [73].

The classification accuracy of the floating-point ProtoNN model used by the pod is 99.86%, which is comparable to the 99.79% accuracy of SeeDot's 16-bit fixed-point implementation. The pod uses an MKR1000 on which SeeDot-generated code is 9.8× faster than the deployed implementation.

8 Related Work

Using quantization to compress ML models is an active research area. A number of works modify the training algorithm to incorporate quantization [13, 14, 23, 31, 36, 48, 60, 75, 90, 97]. More recently, extremely low bitwidth quantization techniques [13, 75, 97] can learn 1, 2, 3-bit weights and biases. All of these techniques require running a non-standard training algorithm to generate quantized models with good accuracy. The modified training techniques are tailored to specific ML tasks (CNNs) or datasets. Instead, SeeDot is a post-training quantization framework that can generate fixed-point code for a pre-trained floating-point model and thus does not require the training algorithm to be modified or re-run.

SeeDot can be considered as an approximate computing framework [4, 76, 77, 81, 98]. However, the prior frameworks do not deal with fixed-point arithmetic and their techniques are complementary.

SeeDot is a compiler that translates Real expressions to fixed-point code. The previous compilers for this task are too restrictive to be useful for ML tasks. In particular, Darulova et al. [15–17] can only express arithmetic over scalar variables and provide no support for matrix operations.

Many frameworks exist for running ML on smart-phones [47, 84]. However, these frameworks require MB-sized memory to run and are irrelevant to KB-sized devices. There are tools in digital signal processing (DSP) that convert floating-point expressions to fixed-point [3, 5, 7, 8, 64, 68, 88]. Such tools do not target the KB-sized devices we consider and are far from ideal in compiling ML inference algorithms to KB-sized microcontrollers. These tools use high-bitwidth operations to compute intermediate results. Unlike DSPs, microcontrollers do not have hardware support for such operations and hence such operations cause huge slowdowns.

Developing ML classifiers for constrained hardware is an active research area [19, 30, 37, 56–58, 94]. There have been several efforts to explore acceleration of ML inference with MB/GB sized models on FPGAs [10, 21, 22, 25, 80]. Since we focus on ML inference with KB-sized models on low-end, low-cost FPGAs, these works are inapplicable at this scale. Under this setting, the research in the area is in its infancy. The closest works to SeeDot are by Guan et al. [28] and Sharma et al. [80]. The former uses a hybrid RTL + HLS framework to accelerate DNN inference on FPGAs and the latter uses a template architecture to map various DNN layers onto it. However, quantization, if necessary, needs to be performed by the user. SeeDot automatically generates fixed-point FPGA implementations that accelerate ML inference on low-end FPGAs that lack floating-point support.

An ML programmer who attempts to use high-level synthesis tools such as Intel's FPGA SDK for OpenCL and Xilinx's Vivado HLS faces a steep learning curve. The SeeDot compiler hides this complexity and makes FPGAs more accessible to an end-user. For example, Embedded FPGAs (eFPGAs) are gaining traction in real-world embedded systems. They are present in various domains such as bio-medical [86], computer vision [52], traffic monitoring [96], and industrial safety [12]. We believe that SeeDot would be useful in making eFPGAs more accessible to programmers who are inexperienced in digital design.

9 Conclusion

SeeDot is a framework for generating precise and efficient fixed-point code for ML inference algorithms that can run on microcontrollers and FPGAs. To this end, SeeDot compiler uses novel techniques like auto-tuning key parameters used in fixed-point code. With these techniques, SeeDot-generated code significantly outperforms existing alternatives for microcontrollers and FPGAs by 2.4×–82.2× and 3.6×–21×, respectively. We believe that SeeDot can facilitate new ML applications.

Acknowledgments

Thanks to the anonymous reviewers for their feedback on this paper. We also thank Tusher Chakraborty, Chirag Gupta, Prateek Jain, Ashish Kumar, Aditya Kusupati, Wonyeol Lee, Akshay Nambi, Bhargavi Paranjape, Shishir Patil, Rahul Anand Sharma, Harsha Simhadri, Sanjay Singapuram and Manik Varma for helpful discussions and feedback.

References

[1] S. Anwar, K. Hwang, and W. Sung. 2015. Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 1131–1135. https://doi.org/10.1109/ICASSP.2015.7178146

[2] T. Araki. 2017. Accelerating Machine Learning on Sparse Datasets with a Distributed Memory Vector Architecture. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. 112–121. https://doi.org/10.1109/ISPDC.2017.21

[3] Jonathan Babb, Martin C. Rinard, Csaba Andras Moritz, Walter Lee, Matthew I. Frank, Rajeev Barua, and Saman P. Amarasinghe. 1999. Parallelizing Applications into Silicon. In *7th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99), 21-23 April 1999, Napa, CA, USA*. 70.

[4] Woongki Baek and Trishul M. Chilimbi. 2010. Green: a framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 198–209.

[5] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe. 2003. Automatic conversion of floating point MATLAB programs into fixed point FPGA based hardware design. In *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2003. FCCM 2003.* 263–264. https://doi.org/10.1109/FPGA.2003.1227262

[6] Massimo Banzi and Michael Shiloh. 2014. *Getting started with Arduino: the open source electronics prototyping platform.* Maker Media, Inc.

[7] M Bečvář and P Štukjunger. 2005. Fixed-point arithmetic in FPGA. *Acta Polytechnica* 45, 2 (2005).

[8] David M. Brooks and Margaret Martonosi. 1999. Dynamically Exploiting Narrow Width Operands to Improve Processor Power and Performance. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, Orlando, FL, USA, January 9-12, 1999*. 13–22.

[9] Pablo Samuel Castro, Daqing Zhang, and Shijian Li. 2012. Urban traffic modelling and prediction using large scale taxi GPS traces. In *International Conference on Pervasive Computing*. Springer, 57–72.

[10] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 7.

[11] T. Chakraborty, S.N. Akshay Uttama Nambi, R. Chandra, R. Sharma, Z. Kapetanovic, M. Swaminathan, and J. Appavoo. 2018. Fall-curve: A novel primitive for IoT Fault Detection and Isolation. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (SenSys '18)*.

[12] L. M. Contreras-Medina, R. J. Romero-Troncoso, J. R. Millan-Almaraz, and C. Rodriguez-Donate. 2008. FPGA based multiple-channel vibration analyzer embedded system for industrial applications in automatic failure detection. In *2008 International Symposium on Industrial Embedded Systems*. 229–232. https://doi.org/10.1109/SIES.2008.4577705

[13] Matthieu Courbariaux and Yoshua Bengio. 2016. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR* abs/1602.02830 (2016). arXiv:1602.02830 http://arxiv.org/abs/1602.02830

[14] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Low precision arithmetic for deep learning. *CoRR* abs/1412.7024 (2014). arXiv:1412.7024 http://arxiv.org/abs/1412.7024

[15] Eva Darulova and Viktor Kuncak. 2014. Sound compilation of reals. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. 235–248.

[16] Eva Darulova and Viktor Kuncak. 2017. Towards a Compiler for Reals. *ACM Trans. Program. Lang. Syst.* 39, 2, Article 8 (March 2017), 28 pages. https://doi.org/10.1145/3014426

[17] Eva Darulova, Viktor Kuncak, Rupak Majumdar, and Indranil Saha. 2013. Synthesis of Fixed-point Programs. In *Proceedings of the Eleventh ACM International Conference on Embedded Software (EMSOFT '13)*. IEEE Press, Piscataway, NJ, USA, Article 22, 10 pages. http://dl.acm.org/citation.cfm?id=2555754.2555776

[18] Teófilo Emídio de Campos, Bodla Rakesh Babu, and Manik Varma. 2009. Character Recognition in Natural Images. In *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 2*. 273–280.

[19] Don Dennis, Chirag Pabbaraju, Harsha Vardhan Simhadri, and Prateek Jain. 2018. Multiple Instance Learning for Efficient Sequential Data Classification on Resource-constrained Devices. In *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*. 10976–10987. all$_{\mathrm{p}}$apers/DennisPSJ18.pdf slides/-DennisPSJ18.pdf.

[20] Peter Eckersley and Yomna Nasser. [n. d.]. AI Progress Measurement | Electronic Frontier Foundation. https://www.eff.org/ai/metrics. (Accessed on 08/04/2018).

[21] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. 2009. CNP: An FPGA-based Processor for Convolutional Networks. In *International Conference on Field Programmable Logic and Applications*. 32–37. https://doi.org/10.1109/FPL.2009.5272559

[22] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[23] Josh Fromm, Shwetak Patel, and Matthai Philipose. 2018. Heterogeneous Bitwidth Binarization in Convolutional Neural Networks. *CoRR* abs/1805.10368 (2018). arXiv:1805.10368 http://arxiv.org/abs/1805.10368

[24] N.P. Ghanathe, A. Madorsky, H. Lam, D.E. Acosta, A.D. George, M.R. Carver, Y. Xia, A. Jyothishwara, and M. Hansen. 2017. Software and firmware co-development using high-level synthesis. *Journal of Instrumentation* 12, 01 (2017), C01083. http://stacks.iop.org/1748-0221/12/i=01/a=C01083

[25] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello. 2014. A 240 G-ops/s Mobile Coprocessor for Deep Neural Networks. In *2014 IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 696–701. https://doi.org/10.1109/CVPRW.2014.106

[26] Yunchao Gong, Liu Liu, Ming Yang, and Lubomir D. Bourdev. 2014. Compressing Deep Convolutional Networks using Vector Quantization. *CoRR* abs/1412.6115 (2014). arXiv:1412.6115 http://arxiv.org/abs/1412.6115

[27] P. Grigoras, P. Burovskiy, E. Hung, and W. Luk. 2015. Accelerating SpMV on FPGAs by Compressing Nonzero Values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. 64–67. https://doi.org/10.1109/FCCM.2015.30

[28] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 152–159. https://doi.org/10.1109/FCCM.2017.25

[29] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future generation computer systems* 29, 7 (2013), 1645–1660.

[30] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN:

compressed and accurate kNN for resource-scarce devices. In *International Conference on Machine Learning*. 1331–1340.

[31] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep Learning with Limited Numerical Precision. *CoRR* abs/1502.02551 (2015). arXiv:1502.02551 http://arxiv.org/abs/1502.02551

[32] Karen Zita Haigh, Allan M. Mackay, Michael Cook, and Li Lin. 2015. Machine Learning for Embedded Systems : A Case Study.

[33] Ben Hamner. [n. d.]. Popular Datasets Over Time | Kaggle. https://www.kaggle.com/benhamner/popular-datasets-over-time/code. (Accessed on 08/04/2018).

[34] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. *CoRR* abs/1602.01528 (2016). arXiv:1602.01528 http://arxiv.org/abs/1602.01528

[35] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[36] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015). arXiv:1510.00149 http://arxiv.org/abs/1510.00149

[37] Song Han, Huizi Mao, and William J. Dally. 2015. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR* abs/1510.00149 (2015).

[38] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. *CoRR* abs/1506.02626 (2015). arXiv:1506.02626 http://arxiv.org/abs/1506.02626

[39] Moeen Hassanalieragh, Alex Page, Tolga Soyata, Gaurav Sharma, Mehmet Aktas, Gonzalo Mateos, Burak Kantarci, and Silvana Andreescu. 2015. Health monitoring and management using Internet-of-Things (IoT) sensing with cloud-based processing: Opportunities and challenges. In *2015 IEEE international conference on services computing (SCC)*. IEEE, 285–292.

[40] Hannes Hassler and Naofumi Takagi. 1995. Function Evaluation by Table Look-up and Addition. In *12th Symposium on Computer Arithmetic (ARITH-12 '95), July 19-21, 1995, Bath, England, UK*. 10–16. https://doi.org/10.1109/ARITH.1995.465382

[41] Chih-Wei Hsu and Chih-Jen Lin. 2002. A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks* 13, 2 (2002), 415–425.

[42] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized Neural Networks. In *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (Eds.). Curran Associates, Inc., 4107–4115. http://papers.nips.cc/paper/6573-binarized-neural-networks.pdf

[43] Jonathan J. Hull. 1994. A database for handwritten text recognition research. *IEEE Transactions on pattern analysis and machine intelligence* 16, 5 (1994), 550–554.

[44] IEEE. 2000. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2000* (2000), i–290. https://doi.org/10.1109/IEEESTD.2000.92297

[45] IEEE. 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 01–560. https://doi.org/10.1109/IEEESTD.2006.99495

[46] IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008* (Aug 2008), 1–70. https://doi.org/10.1109/IEEESTD.2008.4610935

[47] Apple Inc. [n. d.]. Core ML | Apple Developer Documentation. https://developer.apple.com/documentation/coreml. (Accessed on 11/07/2018).

[48] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. *CoRR* abs/1712.05877 (2017). arXiv:1712.05877 http://arxiv.org/abs/1712.05877

[49] Zhanlin Ji, Ivan Ganchev, Máirtín O'Droma, Li Zhao, and Xueji Zhang. 2014. A cloud-based car parking middleware for IoT-based smart cities: Design and implementation. *Sensors* 14, 12 (2014), 22372–22393.

[50] Cijo Jose, Prasoon Goyal, Parv Aggrwal, and Manik Varma. 2013. Local deep kernel learning for efficient non-linear SVM prediction. In *International conference on machine learning*. 486–494. http://manikvarma.org/code/LDKL/download.html

[51] Jongbin Jung, Connor Concannon, Ravi Shroff, Sharad Goel, and Daniel G Goldstein. 2017. Simple rules for complex decisions. *arXiv preprint arXiv:1702.04690* (2017).

[52] A. S. Khalil, M. Shalaby, and E. Hegazi. 2017. A hardware design and implementation for accelerating motion detection using (System On Chip) SOC. In *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. 411–416.

[53] JM Ko and YQ Ni. 2005. Technology developments in structural health monitoring of large-scale bridges. *Engineering structures* 27, 12 (2005), 1715–1725.

[54] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[55] Alex Krizhevsky and G Hinton. 2010. Convolutional deep belief networks on CIFAR-10. *Unpublished manuscript* 40, 7 (2010).

[56] Ashish Kumar, Saurabh Goyal, and Manik Varma. 2017. Resource-efficient Machine Learning in 2 KB RAM for the Internet of Things. In *International Conference on Machine Learning*. 1935–1944.

[57] Matt J. Kusner, Stephen Tyree, Kilian Q. Weinberger, and Kunal Agrawal. 2014. Stochastic Neighbor Compression. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*. 622–630.

[58] Aditya Kusupati, Manish Singh, Kush Bhatia, Ashish Kumar, Prateek Jain, and Manik Varma. 2018. FastGRNN: A Fast, Accurate, Stable and Tiny Kilobyte Sized Gated Recurrent Neural Network. In *Proceedings of the Thirty-first Annual Conference on Neural Information Processing Systems (NeurIPS)*. 9031–9042. all$_p$apers/KusupatiSBKJV18.pdf slides/fastgrnn.pdf.

[59] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[60] Hao Li, Soham De, Zheng Xu, Christoph Studer, Hanan Samet, and Tom Goldstein. 2017. Training Quantized Nets: A Deeper Understanding. *CoRR* abs/1706.02379 (2017). arXiv:1706.02379 http://arxiv.org/abs/1706.02379

[61] Zhouhan Lin, Matthieu Courbariaux, Roland Memisevic, and Yoshua Bengio. 2015. Neural Networks with Few Multiplications. *CoRR* abs/1510.03009 (2015). arXiv:1510.03009 http://arxiv.org/abs/1510.03009

[62] Xiaolei Ma, Zhuang Dai, Zhengbing He, Jihui Ma, Yong Wang, and Yunpeng Wang. 2017. Learning traffic as images: a deep convolutional neural network for large-scale transportation network speed prediction. *Sensors* 17, 4 (2017), 818.

[63] Nicolas Maisonneuve, Matthias Stevens, Maria E Niessen, Peter Hanappe, and Luc Steels. 2009. Citizen noise pollution monitoring. In *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government*. Digital Government Society of North America, 96–103.

[64] Daniel Menard, Daniel Chillet, François Charot, and Olivier Sentieys. 2002. Automatic Floating-point to Fixed-point Conversion for DSP Code Generation. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '02)*. ACM, New York, NY, USA, 270–276. https://doi.org/10.1145/581630.581674

[65] Pavlo Molchanov, Shalini Gupta, Kihwan Kim, and Jan Kautz. 2015. Hand gesture recognition with 3D convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*. 1–7.

[66] A. Monteiro, M. de Oliveira, R. de Oliveira, and T. da Silva. 2018. Embedded application of convolutional neural networks on Raspberry Pi for SHM. *Electronics Letters* 54, 11 (2018), 680–682. https://doi.org/10.1049/el.2018.0877

[67] Subhas Chandra Mukhopadhyay. 2015. Wearable sensors for human activity monitoring: A review. *IEEE sensors journal* 15, 3 (2015), 1321–1330.

[68] Anshuman Nayak, Malay Haldar, Alok N. Choudhary, and Prithviraj Banerjee. 2001. Precision and error analysis of MATLAB applications during automated hardware synthesis for FPGAs. In *Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2001, Munich, Germany, March 12-16, 2001*. 722–728.

[69] John V Oldfield and Richard C Dorf. 1995. *Field-programmable gate arrays: reconfigurable logic for rapid prototyping and implementation of digital systems*. Wiley.

[70] Keiron O'Shea and Ryan Nash. 2015. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458* (2015).

[71] Alexandros Pantelopoulos and Nikolaos G Bourbakis. 2010. A survey on wearable sensor-based systems for health monitoring and prognosis. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 40, 1 (2010), 1–12.

[72] Sungmee Park and Sundaresan Jayaraman. 2003. Enhancing the quality of life through wearable technology. *IEEE Engineering in medicine and biology magazine* 22, 3 (2003), 41–48.

[73] Shishir Patil, Don Kurian Dennis, Chirag Pabbaraju, Rajanikant Deshmukh, Harsha Simhadri, Manik Varma, and Prateek Jain. 2018. *GesturePod: Programmable Gesture Recognition for Augmenting Assistive Devices*. Technical Report. Microsoft. https://www.microsoft.com/en-us/research/publication/gesturepod-programmable-gesture-recognition-augmenting-assistive-devices/

[74] Aswin Raghavan, Mohamed R. Amer, and Sek M. Chai. 2017. BitNet: Bit-Regularized Deep Neural Networks. *CoRR* abs/1708.04788 (2017).

[75] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. *CoRR* abs/1603.05279 (2016). arXiv:1603.05279 http://arxiv.org/abs/1603.05279

[76] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning Assistant for Floating-point Precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 27, 12 pages. https://doi.org/10.1145/2503210.2503296

[77] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2014. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 53–64.

[78] Nicol N Schraudolph. 1999. A fast, compact approximation of the exponential function. *Neural Computation* 11, 4 (1999), 853–862.

[79] Pierre Sermanet and Yann LeCun. 2011. Traffic sign recognition with multi-scale convolutional networks. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*. IEEE, 2809–2813.

[80] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From High-level Deep Neural Models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Press, Piscataway, NJ, USA, Article 17, 12 pages.

[81] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133

[82] Mate Szarvas, Akira Yoshizawa, Munetaka Yamamoto, and Jun Ogata. 2005. Pedestrian detection with convolutional neural networks. In *Intelligent vehicles symposium, 2005. Proceedings. IEEE*. IEEE, 224–229.

[83] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going Deeper with Convolutions. In *Computer Vision and Pattern Recognition (CVPR)*. http://arxiv.org/abs/1409.4842

[84] TensorFlow. [n. d.]. Introduction to TensorFlow Lite. https://www.tensorflow.org/lite/overview. (Accessed on 11/07/2018).

[85] TensorFlow. [n. d.]. Post-training quantization | TensorFlow Lite | TensorFlow. https://www.tensorflow.org/lite/performance/post$_t$raining$_q$uantization. (Accessed on 03/21/2019).

[86] N. M. Thamrin, M. A. Haron, and Fazlina Ahmat Ruslan. 2011. A field programmable gate array implementation for biomedical system-on-chip (SoC). In *2011 IEEE 7th International Colloquium on Signal Processing and its Applications*. 187–191. https://doi.org/10.1109/CSPA.2011.5759870

[87] Manik Varma and Andrew Zisserman. 2005. A statistical approach to texture classification from single images. *International journal of computer vision* 62, 1-2 (2005), 61–81.

[88] M. WILLEMS. 1997. FRIDGE : Floating-point programming of fixed-point digital signal processors. *Proc. International Conference on Signal Processing Applications and Technology 1997 (ICSPAT-97), Sept.* (1997). https://ci.nii.ac.jp/naid/10018558547/en/

[89] Stephen Wolfram et al. 1996. *Mathematica*. Cambridge university press Cambridge.

[90] Shuang Wu, Guoqi Li, Feng Chen, and Luping Shi. 2018. Training and Inference with Integers in Deep Neural Networks. *CoRR* abs/1802.04680 (2018). arXiv:1802.04680 http://arxiv.org/abs/1802.04680

[91] Xilinx. 2015. Vivado Design Suite User Guide, High-level Synthesis. https://www.xilinx.com/support/documentation/sw$_m$anuals/xilinx2012$_2$/ug902-vivado-high-level-synthesis.pdf. (Accessed on 08/04/2018).

[92] Jingjing Yang, Yuanning Li, Yonghong Tian, Lingyu Duan, and Wen Gao. 2009. Group-sensitive multiple kernel learning for object categorization. In *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE, 436–443.

[93] L. Yavits and R. Ginosar. 2018. Accelerator for Sparse Machine Learning. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 21–24.

[94] Kai Zhong, Ruiqi Guo, Sanjiv Kumar, Bowei Yan, David Simcha, and Inderjit S. Dhillon. 2017. Fast Classification with Binary Prototypes. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017, 20-22 April 2017, Fort Lauderdale, FL, USA*. 1255–1263.

[95] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights. *CoRR* abs/1702.03044 (2017). arXiv:1702.03044 http://arxiv.org/abs/1702.03044

[96] Y. Zhou, Z. Chen, and X. Huang. 2016. A system-on-chip FPGA design for real-time traffic signal recognition system. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1778–1781. https://doi.org/10.1109/ISCAS.2016.7538913

[97] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. 2016. Trained Ternary Quantization. *CoRR* abs/1612.01064 (2016). arXiv:1612.01064 http://arxiv.org/abs/1612.01064

[98] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 2012. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 441–454.