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

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, de-
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.

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

\(^1\)Implementation available at https://github.com/Microsoft/EdgeML/tree/master/Tools/SeeDot
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 \( w \ast x > 0 \), 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 \( w \ast x > 0 \). 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 \(^2\lfloor r \ast 2^p \rfloor\). The quantity \( P \) is called the scale. If \( P > 0 \) then we say that \( r \) has been scaled up by \( P \). If \( P < 0 \) then we say that \( r \) has been scaled down by \( |P| \).

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 \( P = 5 \) is optimal since it produces the most accurate result: \( \lfloor \pi \ast 2^5 \rfloor = 100 \) which represents the Real \( 100/2^5 = 3.125 \), the most precise 8-bit fixed-point representation of \( \pi \). If the scale is too high, e.g., if \( P = 6 \) then \( \lfloor \pi \ast 2^6 \rfloor = 200 \), 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 \( P = -2 \) then \( \lfloor \pi \ast 2^{-2} \rfloor = 0 \) 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 \( P = 5 \) are \( y_1 = \lfloor r_1 \ast 2^P \rfloor = 100 \) and \( y_2 = \lfloor r_2 \ast 2^P \rfloor = 86 \). 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 \( P = 4 \), 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 \( y_1 \ast y_2 \) 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 \( \frac{y_1}{2^d} \ast \frac{y_2}{2^d} \) in d-bit arithmetic. 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 \( \frac{y_1}{2^4} \ast \frac{y_2}{2^4} = 30 \) with scale \( 2 \ast 5 - d = 2 \), i.e., \( \frac{2}{4} \approx \pi + e \).

While these naive 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 \( w \ast x \ast r > 0 \), 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 \( w \):

\[
\begin{align*}
\text{let } x &= [0.0767; 0.9238; -0.8311; 0.8213] \text{ in } \\
\text{let } w &= [[0.7793, -0.7316, 1.8088, -1.8622]] \text{ in } \\
\text{w} \ast x
\end{align*}
\]

If we run this program in infinite precision Real arithmetic then \( w \ast x \) evaluates to \(-3.64214951\). Floating-point arithmetic produces the approximately correct answer \(-3.64214948\).

Suppose we represent each Real number using a 8-bit fixed-point number (\textit{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

---

\(^2\text{We only consider integers with fixed number of bits (e.g., 8, 16, 32, etc.).}\)
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,

\[
\text{let } x = \left[ 0.0767 \times 2^7, 0.9238 \times 2^7 \right] \ldots \text{in}
\]

\[
\text{let } w = \left[ 0.7793 \times 2^6, -0.7316 \times 2^6 \right] \ldots \text{in}
\]

\[
\frac{(w_1/2^4 \times x_1/2^4) + (w_2/2^4 \times x_2/2^4)}{2}
\]

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:

\[
\text{let } x = \left[ 0.0767 \times 2^7, 0.9238 \times 2^7 \right] \ldots \text{in}
\]

\[
\text{let } w = \left[ 0.7793 \times 2^6, -0.7316 \times 2^6 \right] \ldots \text{in}
\]

\[
\left( (w_1/2^4 \times x_1/2^4) + (w_2/2^4 \times x_2/2^4) \right)
\]

The computed value of \(w \times x\) is \(-98\) with a scale of 5 and represents the Real value \(-98/2^5 = -3.0625\). 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, then we can 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 \(10^{30}\) 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, \(P\), such that the upper bound for the intermediate values is \(2^{d-P-1}\), where \(d\) is the bitwidth. Given a \(P\), SeeDot uses \(P\) 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 \(P\).

In our example, the magnitude of all intermediate values computed by the expression is less than 4 \((= 2^{5-5-1})\). Hence, SeeDot uses \(P = 5\) to generate the program in (3). Consider the sub-expression \(y_1 + y_2\), where \(y_1 = w_1/2^4 + x_1/2^4\). 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 \(P = 5\), 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.

\[
e := n \mid r \mid M_d \mid M_s \mid x \mid \{ e_1 + e_2 \mid e_1 \times e_2 \mid \exp(e) \mid \argmax(e) \}
\]

**Figure 1. Syntax of the core language of SeeDot**

If \(P = 3\) then the intermediate results have a magnitude below \(2^{5-3-1} = 16\). 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 \(P\), SeeDot generates a classifier program for each \(P\) 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 \(P = 5\) and (2) corresponds to \(P = 3\). For our example program, SeeDot picks \(P = 5\). In general, since ML datasets have outliers, we have observed that using a \(P\) that lets the outliers overflow but preserves significant bits on most inputs leads to better accuracy than using a \(P\) that ensures no overflows on all inputs. Therefore, evaluating all possible choices of \(P\) 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., \(P\)) 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 \(v\) of non-zero values and a list \(i\) of positions of these non-zero values in the matrix. A new identifier \(x\) is created using the \(\text{let}\) keyword. Run-time inputs can be modeled by free variables that are not bound by \(\text{let}\). 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 \(\arg\max\).
We describe the type system of SeeDot as follows: under the typing environment $\Gamma$ map from variables to types. The judgement presentation.

\[
\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \\
\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2
\]

$\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \quad \Gamma \vdash e_2 : \mathbb{R}[n_1, n_2] \\
\Gamma \vdash e_1 + e_2 : \mathbb{R}[n_1, n_2]$

$\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \quad \Gamma \vdash e_2 : \mathbb{R}[n_2, n_3] \\
\Gamma \vdash e_1 \times e_2 : \mathbb{R}[n_1, n_3]$

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

$\Gamma \vdash e : \mathbb{R} \\
\Gamma \vdash \exp(e) : \mathbb{R}$

$\Gamma \vdash e : \mathbb{R}[n] \\
\Gamma \vdash \arg\max(e) \in \mathbb{Z}$

\[
\frac{x \in \text{domain}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \quad T \vdash \text{Var} \quad r \vdash \mathbb{R} \quad T \vdash \text{Real} \\
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad T \vdash \text{Let} \\
\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \quad \Gamma \vdash e_2 : \mathbb{R}[n_1, n_2]}{\Gamma \vdash e_1 + e_2 : \mathbb{R}[n_1, n_2]} \quad T \vdash \text{Add} \\
\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \quad \Gamma \vdash e_2 : \mathbb{R}[n_2, n_3]}{\Gamma \vdash e_1 \times e_2 : \mathbb{R}[n_1, n_3]} \quad T \vdash \text{Mult} \\
\frac{\Gamma \vdash e_1 : \mathbb{R}[n_1, n_2] \quad \Gamma \vdash e_2 : \mathbb{R}[n_2]}{\Gamma \vdash e_1 \times e_2 : \mathbb{R}[n_1]} \quad T \vdash \text{SparseMult} \\
\frac{\Gamma \vdash e : \mathbb{R}[n_1, n_2] \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}[n_1, n_2]} \quad T \vdash \text{M2S} \\
\frac{\Gamma \vdash e : \mathbb{R}[n_1, n_2] \quad n_1 = n_2 = 1}{\Gamma \vdash e : \mathbb{R}[n_1, n_2]} \quad T \vdash \text{S2M} \\
\frac{\Gamma \vdash e : \mathbb{R} \quad \Gamma \vdash \exp(e) : \mathbb{R}}{\Gamma \vdash \exp(e) : \mathbb{R}} \quad T \vdash \text{EXP} \\
\frac{\Gamma \vdash e : \mathbb{R}[n] \quad \Gamma \vdash \arg\max(e) \in \mathbb{Z}}{\Gamma \vdash \arg\max(e) \in \mathbb{Z}} \quad T \vdash \text{ArgMax}
\]

**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:

\[
r := \mathbb{Z} \mid \mathbb{R} \mid \mathbb{R}[n_1] \mid \mathbb{R}[n_1, n_2] \mid \mathbb{R}[n_1, n_2]^k
\]

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 $\mathbb{R}[n_1, n_2]^k$. We restrict the maximum dimension of matrices to two for the ease of presentation.

We use $\Gamma$ to denote the typing environment, which is a map from variables to types. The judgement $\Gamma \vdash e : r$ is read as follows: under the typing environment $\Gamma$, the expression $e$ is well-typed and has a type $r$. The rule $T-\text{Var}$ is standard: a variable $x$ is well-typed if it belongs to the domain of $\Gamma$.

The rule $T-\text{Let}$ is also standard and adds variables to $\Gamma$. $T-\text{Add}$ and $T-Mult$ ensures that we only add and multiply matrices of compatible dimensions. $T-\text{SparseMult}$ 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 $1 \times 1$-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. $\arg\max$ 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 pseudo-code 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 $P$, which was introduced in Section 4. We use $B$ to denote the bitwidth. The functions for exponentiation rule require some other parameters ($T$, $m$, 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 $\kappa$ maps a variable $x$ to a unique location $\eta$ and a scale $P$. The judgment $\kappa \vdash e : (C, \eta, P)$ is read as follows: under an environment $\kappa$, an expression $e$ is compiled to a code $C$, a sequence of procedure calls. The return value of $C$ is stored at location $\eta$, which has a scale $P$. We use $e$ for a no-op code in Figure 3. The rules use these standard functions: function $\text{max}(W)$ returns the maximum element of a matrix $W$; $\text{abs}(W)$ returns a matrix containing the magnitude of each entry in the matrix $W$; $\text{dim}$ returns the dimensions of a matrix as inferred by the type system.

We discuss Figure 3 using examples. Consider the simple SeeDot program: $\text{let } x = 1.23 \text{ in } x$. Compiling this program involves using the rules $C-\text{Let}$, $C-\text{Var}$, and $C-\text{Val}$. $C-\text{Val}$ uses the auxiliary function $\text{GetP}$ to compute the scale of 1.23. If
procedure of Algorithm 2 that minimizes the precision loss. To achieve this, the variable \( x \) needs to be scaled down before they are added or multiplied. In particular, the intermediate results \( \eta \) of the addition can result in larger numbers that can overflow the available resources. Therefore, we need to scale the arguments of addition before adding them together. However, if \( \eta \) is applicable, this example compiles to an empty program and the return value of the expression is stored at the location \( \eta \), where \( \eta \) returns 14 and the value \( 20152 \) is stored with a scale of 13, i.e., 2.4599609375, which is a good approximation of the exact result 2.46.

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 \( 2^{S_{add}} = 2 \) before adding them together. However, if \( P \) 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.

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 in 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 the function \( \text{ArgMax} \) 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 significant bits. For each of these parts, we compute two look-up tables \( T_f \) and \( T_g \), where \( x \) is a positive 16-bit fixed-point number.

$$e^x \approx e^{2^i a} \cdot e^{2^i b} = T_f[a] \cdot T_g[b]$$

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

1: function GetP(n)
2:   retun $(\mathcal{B} - 1) \cdot \lfloor \log_2(n) \rfloor$
3: function MultiScale($P_1$, $P_2$)
4:   $S_{mul} \leftarrow \mathcal{B}$
5:   $P_{mul} \leftarrow (P_1 - S_{mul} / 2) + (P_2 - S_{mul} / 2)$
6:   if $P_{mul} = P$ then
7:     $S_{mul} \leftarrow \max(\mathcal{B} - (P - P_{mul}), 0)$
8:   $P_{mul} \leftarrow (P_1 - S_{mul} / 2) + (P_2 - S_{mul} / 2)$
9:   return $P_{mul}, S_{mul}$
10: function AddScale($P$)
11:   $P_{add} \leftarrow P - 1$
12:   if $P_{add} = P$ then
13:     $S_{add} \leftarrow 0$
14:   $P_{add} \leftarrow P$
15:   return $P_{add}, S_{add}$
16: function TreeSumScale($P$, $n$)
17:   $S_{add} \leftarrow \lfloor \log_2(n) \rfloor$
18:   $P_{add} \leftarrow P - S_{add}$
19:   if $P_{add} = P$ then
20:     $S_{add} \leftarrow \max(S_{add} - (P - P_{add}), 0)$
21:     $P_{add} \leftarrow P - S_{add}$
22:   return $P_{add}, S_{add}$
23: function ExpTable($P$, $m$, $M$)
24:   $k \leftarrow \lfloor \log_2(M - m) \rfloor$
25:   $P_1 \leftarrow GetP(e^m), P_2 \leftarrow GetP(1)$
26:   for $i$ in $0:2^k$ do
27:     $T_f[i] \leftarrow \left\lfloor e^{(m+2^{k-i})/2^k} \cdot 2^P_1 \right\rfloor$
28:     $T_g[i] \leftarrow \left\lfloor e^{(2^{k-i} - 2^i)/2} \cdot 2^P_2 \right\rfloor$
29:   return $T, P_1, P_2, k$

Algorithm 2 Codegen procedures

1: procedure MatMul($A$, $B$, $S_{mul}, S_{add}$)
2:   $(P, Q) \leftarrow \dim(A), (Q, R) \leftarrow \dim(B)$
3:   var $T[Q], C[P, R]$
4:   for $i$ in $0:P$ do
5:     for $j$ in $0:Q$ do
6:       var $a \leftarrow A[i][j] / 2^S_{mul}$
7:       var $b \leftarrow B[i][j] / 2^S_{mul}$
8:       $T[k] \leftarrow a * b$
9:     $C[i][j] \leftarrow TreeSum(T, S_{add})$
10: return $C$
11: procedure SparseMatMul($A$, $B$, $S_{mul}, S_{add}$)
12:   $(P, Q) \leftarrow \dim(A), Q \leftarrow \dim(B)$
13:   var $C[P, 1], i_{idx} \leftarrow 0, i_{val} \leftarrow 0$
14:   for $i$ in $0:Q$ do
15:     $j \leftarrow A.i_{idx}[i_{idx}]$ do
16:       $j \leftarrow i_{idx++}$
17:       while $j != 0$ do
18:         var $u \leftarrow 2^S_{mul}$
19:         var $t \leftarrow A.val[i_{val++}] / u * (B[i] / u)$
20:         $C[j-1][0] \leftarrow C[j-1][0] + (t / 2^S_{add})$
21:       $j \leftarrow A.i_{idx[i_{idx++}]}
22: return $C$
23: procedure TreeSum($A$, $S_{add}$)
24:   $n \leftarrow \dim(A)$
25:   var $k \leftarrow n/2, s \leftarrow 1$
26:   while $k > 1$ do
27:     if $S_{add} - s <= 0$ then $s \leftarrow 0$
28:     for $i$ in $0:k$ do
30:     $n \leftarrow (n+1)/2, k \leftarrow n/2$
31: return $A$
32: procedure MatAdd($A$, $B$, $n$, $S_{add}$)
33:   $P \leftarrow \dim(A)$
34:   var $C[P, Q]$
35:   for $i$ in $0:P$ do
36:     for $j$ in $0:Q$ do
37:       $C[i][j] \leftarrow (A[i][j] / 2^S_{add}) + (B[i][j] / 2^S_{add})$
38: return $C$
39: procedure Exp($x$, $T_f$, $T_g$, $S_{mul}, k$)
40:   var $i \leftarrow bits_{s_f}(x, k), j \leftarrow bits_t(x, k - T_f)$
41:   var $e \leftarrow (T_f[i] / 2^S_{mul}) * (T_g[j] / 2^S_{mul})$
42: return $e$
43: procedure ArgMax($A$)
44:   $P \leftarrow \dim(A)$
45:   var $index \leftarrow 0, max \leftarrow A[0]$
46:   for $i$ in $0:P$ do
47:     if $A[i] > max$ then
48:       max $\leftarrow A[i], index \leftarrow i$
49: 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 $B$, i.e., the number of bits assigned to each variable and the maxscale $P$. In particular, for $B = 16$, SeeDot generates 16 programs for each $P$ value from 0 to 15 and chooses the $P$ 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.

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 $\times$ 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, $2.6 \times-14.9 \times$, 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 $10 \times 1$ vectors.

\[
\text{let } D = A - B + C \text{ in } D
\]

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.

```c
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
We compare SeeDot and ProtoNN on 10 different datasets and learned 20 different models. We reported above includes only the cases where the floating-point implementations. The average accuracy loss have better classification accuracy than the corresponding models. We note that, in most cases, the MKR implementations are more precise because they use 32-bit integers and the Uno implementations use 16-bit integers. Thus, fixed-point implementations are much more efficient for both these ML inference algorithms. The average loss in classification accuracy on the absolute execution time of SeeDot-generated code in milliseconds. The y-axis is speedup is 2.9 times faster than the respective floating-point implementations from [30, 56]. The text on each bar shows the speedups of SeeDot-generated code over hand-written floating-point code. 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 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.

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
We note that, in some cases, the classification accuracy of TensorFlow-Lite-generated code on an Arduino Uno. TF-Lite code are all performed in floating-point. For devices 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.

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.

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×.
We compare the results for ProtoNN implementations at 10MHz and 100MHz, with HLS as baseline. We observe that acceleration of KB-sized ML models running on constrained devices versus low-end FPGAs. We observe that the FPGA implementations and have the same classification accuracy. To the best of our knowledge, this is the first empirical comparison of KB-sized ML models on low-end FPGAs. We delve into a preliminary evaluation of SeeDot’s potential to accelerate KB-sized models.

### 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 in milliseconds. The speedup in blue use math.h for computing $e^x$. The increase in speedup from the exponentiation technique is $3.8 \times 9.4x$.

**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](image)

**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 $3.8 \times 9.4x$.

#### 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 $[r \cdot 2^{6-6}]$. 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 ($W$-1). We evaluate each configuration on the

![Figure 12](image)

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

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 1.6x 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 9.8x 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 smartphones [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 2.4x–82.2x and 3.6x–21x, 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 Singapurnam and Manik Varma for helpful discussions and feedback.
References


