Fast Linear Interpolation

NATHAN ZHANG, Stanford University
KEVIN CANINI, Google Research
SEAN SILVA, Google
MAYA GUPTA, Google Research

We present fast implementations of linear interpolation operators for piecewise linear functions and multi-dimensional look-up tables. These operators are common for efficient transformations in image processing and are the core operations needed for lattice models like deep lattice networks, a popular machine learning function class for interpretable, shape-constrained machine learning. We present new strategies for an efficient compiler-based solution using MLIR to accelerate linear interpolation. For real-world machine-learned multi-layer lattice models that use multidimensional linear interpolation, we show these strategies run $5 - 10\times$ faster on a standard CPU compared to an optimized C++ interpreter implementation.

CCS Concepts: • **Software and its engineering** → **Source code generation**.

Additional Key Words and Phrases: compiler, interpolation

ACM Reference Format:
Nathan Zhang, Kevin Canini, Sean Silva, and Maya Gupta. 2020. Fast Linear Interpolation. *ACM J. Emerg. Technol. Comput. Syst.* 1, 1 (September 2020), 16 pages.

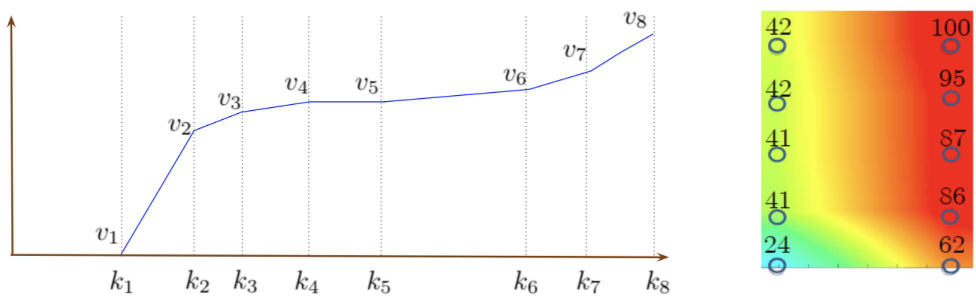
1 INTRODUCTION

Linearly-interpolated look-up tables (LUTs) are a core operation of many real-world machine-learned models, such as piecewise-linear (PWL) functions and lattice models, which interpolate multi-dimensional look-up tables [10]; see Fig. 1 for examples. Interpolating LUTs has long been a common choice for low-dimensional signal and image processing applications where fast evaluation and flexible models are needed. For example, the International Color Consortium standard for color management for printers uses two-layer models where PWLs correct each individual color channel, and then multi-dimensional look-up tables convert between 3D colorspaces [9, 27]. Recently, as concerns grow about the black-box nature of AI, lattice models have become useful to machine learning practitioners because they offer interpretable and semantically-regularized machine learning by enabling constraints on the underlying LUTs that provide guarantees about model behavior, like ensuring that selected inputs can only increase the output (monotonicity) while still producing flexible accurate models [4, 6, 12–14, 29, 31, 33]. Higher-dimensional problems are handled with multi-layer models formed by ensembling or cascading multiple layers of PWLs and lattices [4, 6, 29, 33].

In this paper, we investigate the question: Just how fast can one interpolate LUTs on standard CPUs? In theory, interpolating LUTs can be very fast because only the LUT parameters nearest an input are needed to evaluate that input. This is in stark contrast to models like DNNs and CNNs,

Authors' addresses: Nathan Zhang, nathanzhang@google.com, Stanford University; Kevin Canini, Google Research, canini@google.com; Sean Silva, Google, silvasean@google.com; Maya Gupta, mayagupta@google.com, Google Research.

1550-4832/2020/9-ART $15.00
https://doi.org/



PWL

Lattice with 5×2 LUT

Fig. 1. Left: An example of a PWL defined by 8 key-value pairs. Right: An example of a lattice model formed by bilinear interpolation of each cell of a 5×2 LUT defined on a regular grid of keys (a.k.a. knots) with 10 free LUT parameters corresponding to each key. The rainbow colormap goes from blue (0) to red (100). Inputs outside the domain of the LUT were clipped component-wise to the domain of the LUT.

for which every model parameter might be touched during the evaluation of a single example. However, ML models based on LUTs have many small operations and are thus easily bottle-necked by overhead, thus necessitating performance-oriented optimizations there as well. We show that our proposed custom low-level optimizations and data-handling implemented using MLIR [11] can enable fast runtimes on standard CPUs.

Specifically, the main contributions of this paper are: *(i)* we give two new complementary strategies to reduce the runtime of PWLs; *(ii)* we show how to optimize the compute kernels for two types of multi-dimensional linear interpolation (simplex and multilinear); *(iii)* overall we show a $5 - 10\times$ runtime speed-up on real-world multi-layer lattice models compared to a prior C++ interpreter implementation with optimized C++ code.

First, we review some related work on compilation for ML. Then in Section 3, we review one-dimensional and multi-dimensional linear interpolation and what is already known about how to make them efficient. Then we propose new strategies for more efficient PWLs in Section 4, more efficient multilinear interpolation in Section 5, and more efficient simplex interpolation in Section 6. We contextualize these contributions in Section 7 with analysis of theoretically possible performance. Experiments in Section 8 show our proposals lead to $5 - 10\times$ speed-ups on real multi-layer machine-learned models. We conclude in Section 9 with a summary and some open questions.

2 COMPILATION MATTERS DUE TO LARGE DISPATCH OVERHEAD

Recent work investigated the use of compilers to speed-up machine-learned models [5, 17, 24, 28, 30], but focused on models with *large operations*, such as convolutional networks or large matrix-multiplication models, where the operations takes hundreds of times longer than the dispatch process. For example, ResNet-34 performs 3.6 billion floating point ops across 34 layers, averaging over 100 million floating point operations per kernel [15].

In contrast, for *small operation* models like linear interpolation (technical details follow), the cost of dispatch can be bigger than the computation, and thus reducing dispatch overhead becomes key. For example, the proprietary multi-layer lattice models described in this paper execute at most a few thousand floating point ops per kernel, and many useful lattice models use a few hundred or fewer ops per kernel. We often use lattice models in latency-sensitive pipelines where single examples must be evaluated as they occur, removing the choice to amortize overhead by batching.

To reduce dispatch overhead, we use a compiler constructed using the MLIR framework [11] to convert the trained models into compiler-optimized C++ code. By replacing the interpreter with a hard-coded model, the compiler removes a significant portion of the dispatch overhead and provides an overall speedup of $2-3\times$. In the next sections, we show how to reduce the overall runtime by $5-10\times$ by taking advantage of the details of the linear interpolation ops.

3 BACKGROUND AND RELATED WORK

We review PWLs, and the two most popular multidimensional interpolation methods, and what the challenges are in making them run fast.

3.1 Piecewise-Linear Functions (PWLs)

PWLs have been used to approximate and represent one-dimensional functions for centuries, for example tables for logarithms [26] [23] and actuarial tables [7]. As illustrated in Fig. 1, we define a PWL by N key-value pairs $(k_i, v_i)_{i=1}^{N}$, where the keys are sorted $k_i < k_{i+1}$.

To evaluate any input $x \in [k_1, k_N]$, the surrounding key-value pairs are linearly interpolated. That is, first find the index of the nearest keypoint to the left of x:

$$j = \max\{i : k_i \leq x\}. \tag{1}$$

Then,

$$f(x) = w_j(x)v_j + (1 - w_j(x))v_{j+1}, \tag{2}$$

where the interpolation weight is

$$w_j(x) = \frac{k_{j+1} - x}{k_{j+1} - k_j}. \tag{3}$$

One nice property of PWLs for safe and interpretable AI [13, 16, 31] is that they can be guaranteed to be monotonically increasing if every look-up table parameter is larger than its left-neighbor.

3.2 Quantile Keypoints

For machine-learned PWLs it is recommended to choose the PWL keys $\{k_i\}$ based on the quantiles of the training examples for that input: assign keypoint k_1 to the minimum possible value of the PWL's domain, assign the last keypoint k_N to the maximum possible value of the PWL's domain, and assign the remaining $N-2$ keypoints to equally-spaced quantiles of the training examples' inputs [29]. We assume the keypoints stay fixed at these values and are not trained (though their corresponding values $\{v_i\}$ are trained). Quantile keypoints are good for machine-learning because each keypoint sees roughly $1/N$ of the training examples, reducing the chance of overfitting any of the trained PWL values $\{v_i\}$. Quantile keypoints also aid in interpretability because if one plots a PWL, the keys reflect the distribution of the training data. However, we next show that quantile keypoints result in pessimal PWL runtimes because each keypoint is equally likely to be the left-keypoint for a random input x.

3.3 Linear Search For PWLs Is Slow, Even For Small PWLs

The key challenge to efficiently evaluating PWLs is to quickly find the index j in (1). We estimate the cost of a memory load and compare at 2 cycles if speculative memory loads are allowed, and 4 cycles if they are not, due to the 2-3 cycle latency of a memory load. Each branch misprediction costs an expected 15-20 cycles depending on architecture [8]. As a result, we estimate the overall expected cost to be $E[\text{\# Cycles}] = 4 \times E[\text{\# Comparisons}] + 17 \times E[\text{\# Mispredictions}]$. Here, we assume that all parameters are within the L1-cache because there are few enough parameters to be loaded together.

Linear search over the N keypoints requires $E[\text{\# Comparisons}] = \frac{N+1}{2}$. However, as we show in this section, it is the branch mispredictions that are the bigger problem.

Recall that a *branch predictor* predicts whether or not a given branch is taken, and an *optimal branch predictor* always predicts the outcome associated with the highest probability. During the linear search, a branch prediction will predict whether the for-loop over the keypoints will stop, for each $i = 1, \ldots, N$. Typically, a branch predictor is able to access a summary of its history, and any static information the compiler may be able to provide. For quantile keypoints, each of the first $N - 1$ keypoints is equally likely to be the correct index. Thus the optimal branch prediction is to continue unless the linear search has reached $i = N - 2$, in which case there is a 50-50 chance of either of the remaining two keypoints being the right one. However, $(N - 2)/(N - 1)$ of the examples will find their correct index before the linear search reaches $i = N - 2$, which means the branch prediction will be wrong once with probability $(N - 2)/(N - 1)$, producing $E[\text{\# Mispredictions}] = (N - 2)/(N - 1)$.

3.4 Binary Search for PWLs

In a branch-free implementation of binary search with known depth, the compiler is able to fully unroll the structure and thus avoid branch mispredictions. Additionally, a well-optimized binary search implementation is able to perform each step of the binary search in approximately 6 cycles [18]. It thus takes $6 \times \lceil \log_2(N) \rceil$ cycles to find the appropriate location. This makes binary search roughly 2× more efficient than linear search even for $N = 3$ to 10, and is the baseline that any proposed indexing must beat.

3.5 A Map-to-Index Function for PWLs

More abstractly, the goal is to construct an efficient function that can map an input x to the correct index j. An old trick is to build an auxiliary LUT over $[k_1, k_N]$ with B uniformly-spaced buckets, use that to map x to a bucket, and then linearly-search through all the keypoints that fell in that bucket. However, with irregularly-spaced keypoints, a uniform bucket can still have $O(N)$ keypoints to search through. Aus and Korn [1] proposed constructing a hierarchy of such auxiliary *uniformly*-spaced LUTs to better cover irregular keypoints. O'Grady and Young [22] proposed using a sufficiently large B such that no uniform bucket contains more than one keypoint, but at the cost of potentially large B. An analogous problem arises in database indexing, where recent work has proposed machine-learning a two-layer DNN to produce the map-to-index function [19]. Our proposed solution will be in a similar spirit but lighter-weight.

3.6 Multilinear Interpolation

Next we review linear interpolation for regular D-dimensional LUTs. See Fig. 1 for an example $D = 2$ dimensional LUT with a regular grid of 5×2 keys, and Fig. 2 for more examples of $D = 2$ LUTs on a regular grid of 2×2 keys.

An interpolated LUT is called a lattice. Our real-world models in Section 8 use LUTs of dimension $D = 4 - 8$. In practice, D up to 20 is reasonable, beyond $D = 20$, memory can be an issue due to the 2^D parameters for one cell of a D-dimensional LUT. Higher-dimensional feature vectors are handled with ensembles [4] and multi-layer *deep lattice networks* [33].

Like PWLs, a nice property of lattice models is they can be restricted or checked for whether their output is a monotonic response of selected inputs, simply by constraining that adjacent parameters in the underlying multidimensional LUT are increasing in the selected input directions

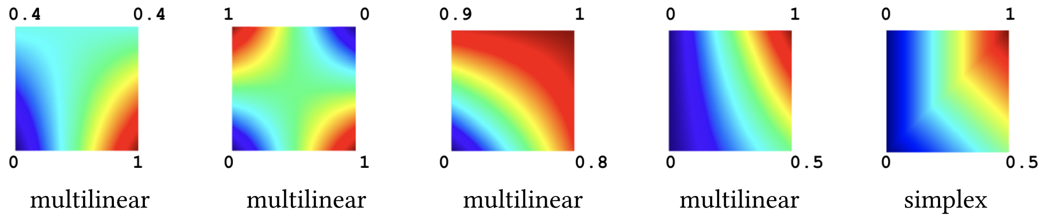


Fig. 2. Examples of $D = 2$ lattices defined by 2×2 LUTs with different parameters, which are shown at the corresponding keys. Colorbar goes from blue (0.0) to red 1.0). The first four examples interpolate the LUT with multilinear interpolation. The fifth example uses simplex interpolation instead. One can see in the simplex interpolation example that the function is linear on the $D! = 2$ simplices: the lower-right triangle and the upper left triangle. The right-most three examples are all monotonically increasing functions in both directions, which can be checked by noting the LUT parameters increase in each dimension. The left-most example is a non-decreasing function in the horizontal direction only.

[13, 29]. Monotonicity constraints have been shown to be useful for AI interpretability [13, 29], regularization [4], and making AI models more ethical [31].

The linear interpolation acts on each cell of the LUT independently. Consider one cell of a regular D-dimensional LUT, which without loss of generality is a D-dimensional unit hypercube parameterized by LUT values $v \in \mathbb{R}^{2^D}$ corresponding to the $O(2^D)$ vertices of the hypercube.

There are multiple ways to linearly interpolate a D-dimensional look-up table [27], but the most common is multilinear interpolation, which for $D = 2$ is known as bilinear interpolation. For an input $x \in [0, 1]^D$, multilinear interpolation outputs $f(x) = \sum_{i=1}^{2^D} v_i w_i(x)$, where v_i is the stored multi-d LUT value for the ith vertex in the D-dimensional unit hypercube, and $w_i(x)$ is the multilinear interpolation weight on the ith unit hypercube vertex $\xi_i \in [0, 1]^D$ taken in lexicographical order, computed from $x \in [0, 1]^D$ as:

$$w_i(x) = \prod_{d=1}^{D} x[d]^{\xi_i[d]} \, (1 - x_d)^{1-\xi_d}\,, \tag{4}$$

for all $i = 1, \ldots, 2^D$.

Gupta et al. [13] gave an $O(2^D)$ dynamic programming algorithm for computing (4). We note that while asymptotically efficient, the dynamic programming solution introduces a loop-carried dependency, and thus prevents critical compiler optimizations, a problem we show how to avoid.

3.7 Simplex Interpolation

Simplex interpolation is a more efficient linear interpolation of a D-dimensional LUT cell that produces a locally linear surface. For each input $x \in [0, 1]^D$, the D components of x are sorted, the resulting sort order determines a set of $D + 1$ vertices whose simplex is guaranteed to contain x, and then a sparse inner product is taken with the corresponding $D + 1$ LUT values to produce $f(x)$ [13, 25, 32].

This method produces a continuous function made up of $D!$ different local hyperplanes over $D!$ mutually-exclusive simplices that partition the LUT cell. See the two right-most examples of Fig. 1 for a visual comparison of the output of multilinear interpolation and simplex interpolation of the same LUT.

Simplex interpolation is the same as the Lovász extension in submodularity [2].

Gupta et al. [13] gave runtimes for single-layer models using either multilinear or simplex interpolation implemented in C++ on a single-threaded 3.5GHz Intel Ivy Bridge processor: for $D = 4$ inputs both simplex and multilinear interpolation ran in about 50 nanoseconds, but for $D = 20$, simplex interpolation ran in 750 nanoseconds, and multilinear interpolation ran in 12 milliseconds, around 15,000× slower than simplex. Despite the slower runtime for $D > 4$, multilinear interpolation might be preferred because it produces a smoother surface. The large-scale multiplications needed for multilinear are a better match for machine learning libraries like TensorFlow than the sorting operation needed for simplex interpolation.

Like with PWLs, branch prediction poses a significant challenge when implementing the simplex interpolation kernel. For example, we found that libc++ (used by LLVM) defaults to using either a hard-coded insertion sort or quicksort depending on the input size, which is determined at runtime. However, we note that this run-time decision is not needed for machine learning models, because the number of inputs D is fixed and known. If one sorts with `std::sort<std::pair<double, int>>`, we found that the sorting operation accounts for approximately 70% of the simplex kernel's overall runtime.

4 HOW TO MAKE PWLS RUN FASTER

We describe two complementary techniques for making PWLs run faster. In Section 4.1 we show how to construct a better auxiliary index-mapping function that takes into account the spacing of the keypoints. In Section 4.2 we show that ensemble or deep lattice models often pass the same input through multiple PWLs, allowing us to remove redundant index searches.

$$x \longrightarrow \underbrace{\mathrm{LUT}[\ \overbrace{\lfloor \alpha + \beta T(x) \rfloor}^{\text{Auxiliary Lookup Index}}\ ]}_{\text{Predicted Keypoint Index}} = m(x) \longrightarrow \underbrace{\mathrm{correct}(m(x))}_{\text{Keypoint Index}} = j \longrightarrow w_j(x)v_j + (1 - w_j(x))v_{j+1} = \mathrm{PLF}(x)$$

Fig. 3. Process for efficiently computing a piecewise linear function using an auxiliary lookup table and transform functions, as in section 4.1. The definitions of w and v are given in Section 3.1.

4.1 Keypoint Dependent Optimization

We propose a new way to construct an index mapping function m that first transforms the keypoints to be more uniformly-spaced, and then applies an auxiliary lookup table (LUT) to map the transformed space to an index using an optimal number of uniform buckets. The resulting implementation, summarized in figure 3, is constant-time in the number of pieces in the PWL.

Let $C(m, x)$ be the cost of evaluating the mapping m on an input x, and $A(m, x)$ be the cost of correcting the predicted index $m(x)$ to the correct index given in (1). If we know the model will be evaluated on random examples $x \sim P$, $x \in \mathcal{X} \subseteq \mathbb{R}$, then we propose finding the mapping m^* from a family $\mathcal{M}$ of possible mapping functions that minimizes the expected cost over random examples to be evaluated:

$$\arg\min_{m \in \mathcal{M}} \mathbb{E}_{x \sim P}[C(m, x) + A(m, x)]. \tag{5}$$

We propose taking $\mathcal{M}$ to be mappings of the form

$$m(x) = \mathrm{LUT}[\lfloor \alpha + \beta T(x) \rfloor],$$

where $T : \mathcal{X} \to \mathbb{R}$ is some 1-D transform, the auxiliary lookup table LUT has B uniformly spaced buckets of size $\frac{1}{\beta}$, where each bucket maps to the smallest index encountered in the bucket's interval, and α is used as an offset in computing the lookup table index.

Estimating 3 operations for arithmetic and 2 operations for typical L1-cache hit latency, we estimate the cost $C(m, x)$ of performing $m(x)$ to be $C(T, x) + \lambda B + 5$, where λ is a parameter that represents the cache behavior cost of arrays, and is set to $\lambda = 0.05$ for all of our experiments. We note that this is an important but relatively insensitive hyperparameter as our parameters are loaded into a contiguous buffer; if a set of parameters is large then it means that the next set of parameters is unlikely to be loaded as part of the same cache line.

For most modern CPUs, the branch-misprediction penalty is high, so we only use functions that are branch-free. A consequence of making T branch free is that the cost $C(T, x) = C(T)$ is independent of x. To correct the predicted index $m(x)$ we use a fixed-step linear search that always executes the worst-case number of steps needed to achieve branch-free behavior. Importantly, this also removes dependence on the actual distribution P in (5); the procedure instead optimizes over the range of possible inputs $\mathcal{X}$.

With these choices, (5) becomes:

$$\underset{T \in \mathbb{T}, \alpha \in \mathbb{R}, \beta \in \mathbb{R}, B \in \mathbb{N}}{\arg\min} \quad C(T) + \lambda B + A(m, x) \tag{6}$$

where

$$A(m, x) = \max_{x \in \mathcal{X}} \text{Index}(x) - \text{LUT}[\lfloor \alpha + \beta T(x) \rfloor].$$

Note that $A(m, x)$ is determined by the pair of points with maximally different indices that get mapped to the same bucket. That is,

$$\begin{aligned} A(m, x) = \; & \max_{x, z \in \mathcal{X}} \text{Index}(x) - \text{Index}(z) \\ & \text{such that } \lfloor \alpha + \beta T(x) \rfloor = \lfloor \alpha + \beta T(z) \rfloor. \end{aligned}$$

Importantly, the choice of T need not be a perfect mapping; in fact, for complex distributions, it is likely that a simple transformation followed by multiple adjustment steps may be optimal. Experimentally, we approximated T by choosing the best out of a small fixed set of simple monotonic transformations, including a fast approximate $\log_2(x)$ and an approximate 2^x, which resulted in needing at most 3 steps for the linear scan for our models. We note that the problem of constructing T, or m in general, may be a problem well suited for superoptimizers.

In order to solve (6), we first choose the T that produces the most linear transform of the PWL keypoints in that it minimizes the squared distance from the set $\{(1, T(k_1)), (2, T(k_2)), \ldots, (N, T(k_N))\}$ to the best fit line through the set. We next do a grid search over the $\alpha - \beta - B$ space that checks all candidate α's and β's and B's that can produce unique values for $A(m, x)$, which can be reduced to checking $O(N^3)$ candidates. Overall, for $N = 50$ keypoints, solving (6) usually takes 1-2 seconds on a standard CPU.

4.2 Efficient Handling of Shared Index PWLs

Next, we consider models that have *shared index PWLs* such that the same input is passed through multiple PWLs in order to transform the same input in different ways. As a very simple example, the one-dimensional function $f(x) = 3\log(x) + 4\sqrt{x} + 6x + 2x^2$ for $x \in [0, 1]$ can be represented as the sum of four PWLs with the same k_i values, but different v_i values. For example, if one sets the k_i's based on the quantiles of input x, then all 4 PWLs will have the same k_i values. In this case, the work to determine j in (1) is duplicated across all PWLs that act on the same input. To avoid this

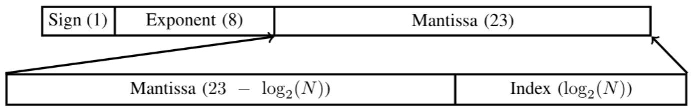


Fig. 4. We propose encoding the index in the low-order bits of the residual for higher sorting efficiency in simplex interpolation. For the double-precision floating point format the exponent instead has 11 bits and the mantissa has 53 bits.

duplication, in the compiler we transform the model to group all such PWLs into a single larger kernel.

5 LATENCY HIDING FOR FASTER MULTILINEAR INTERPOLATION

Gupta et al. [13] give a $O(2^D)$ dynamic programming algorithm for multilinear interpolation that iterates over the D inputs, and on each iteration the number of computed interpolation weights doubles. This makes the last iteration roughly half of the work. We propose that during this last and most expensive iteration, one can interleave in the next step of computing the inner product between the interpolation weights and the LUT values. This interleaving of operations helps the processor be productive while the next value is fetched. This trick is an example of *latency hiding*, a popular computing technique for performing useful work while waiting on a data fetch [20, 21], and is critical for performance using accelerators [5]. To get this latency hiding, we take the trained model parameters, and generate C++ code for the multilinear interpolation such that the compiler will do this interleaving. This provides around a 10-15% end-to-end speed-up for relevant models.

6 SORTING NETWORKS FOR FASTER SIMPLEX INTERPOLATION

As described in Algorithm 2 of [13], the simplex interpolation algorithm requires a sorting permutation π over the inputs. On a D-dimensional hypercube, the simplex interpolation requires $D + 1$ weights, but constructing the sorting permutation requires $O(D \log(D))$ comparisons which we found dominates the overall runtime.

As with piecewise linear functions, the cost of even a single branch misprediction is high, so we desire a branch-free algorithm. We use *sorting networks* [3], which construct sequences of max and min operations in order to construct a branch-free sorting implementation for inputs of fixed size. Because the size D of the LUT is fixed, each evaluation will require the same size-D sort.

A sorting permutation would typically be constructed by sorting $\langle key, index \rangle$ pairs, but such paired min-max operations require a comparison followed by six conditional moves. We note that sorting over basic datatypes is much more efficient: min-max on basic datatypes only requires a min and max operation, requiring fewer than half the cycles. This is particularly important for the small sorting problems that arise in simplex interpolation: D is unlikely to be bigger than 25, because the number of parameters to define a D-dimensional LUT cell is 2^D. Since small sorting problems can be handled almost entirely in registers, optimal sorting code is effectively purely computation. To leverage this more efficient sorting, we adopt the bit-packing technique described in Fig. 4, encoding the index in the low-order bits. The lost precision on the key is $\lceil \log_2(D) \rceil$ bits; a lattice with 2^{32} parameters would lose 5 bits of precision. By definition the interpolation weights are each $w_i \in [0, 1)$, so we can bound the absolute error on each w_i by $\epsilon \lceil \log_2(D) \rceil$, where ϵ is the machine precision for the datatype. As a result, the interpolation output has relative error of at most $\epsilon \lceil \log_2(D) \rceil$.

Operation	Compute	Memory
Simplex	$\underbrace{D\log(D)}_{\text{sort}} + \underbrace{D+1}_{\text{index calculation}} + \underbrace{2D+1}_{\text{sparse product}}$	$D+1$
Multilinear	$\underbrace{2^D}_{\text{interpolation weights}} + \underbrace{2\times 2^D}_{\text{interpolation}}$	2^D
PWL Lookup	$C_T(x)+5$	1
PWL Interpolation	3	3

Fig. 5. Approximate computation and memory requirements of core kernels. D is the dimension of the lattice for simplex and multilinear interpolation. C_T refers to the cost of the transform function used in section 4.1.

7 PEAK PERFORMANCE MODELING

We summarize the theoretical performance needs for these interpolation methods in Fig. 5. As these kernels are far smaller than the typical granularity for roofline analyses, the values presented here are rough estimates. For simplex, memory access is typically serially dependent with other compute or memory access; array access locations are computed right before they are needed and the array value is required immediately after. As a result, the cache latency is difficult to hide.

8 EXPERIMENTAL PEFORMANCE EVALUATION

To illustrate the overall value of these proposals, some of which synergize, we ran experiments on four machine-learned multi-layer lattice models: one model trained on a public benchmark dataset, and three proprietary models from Google. Each is detailed below.

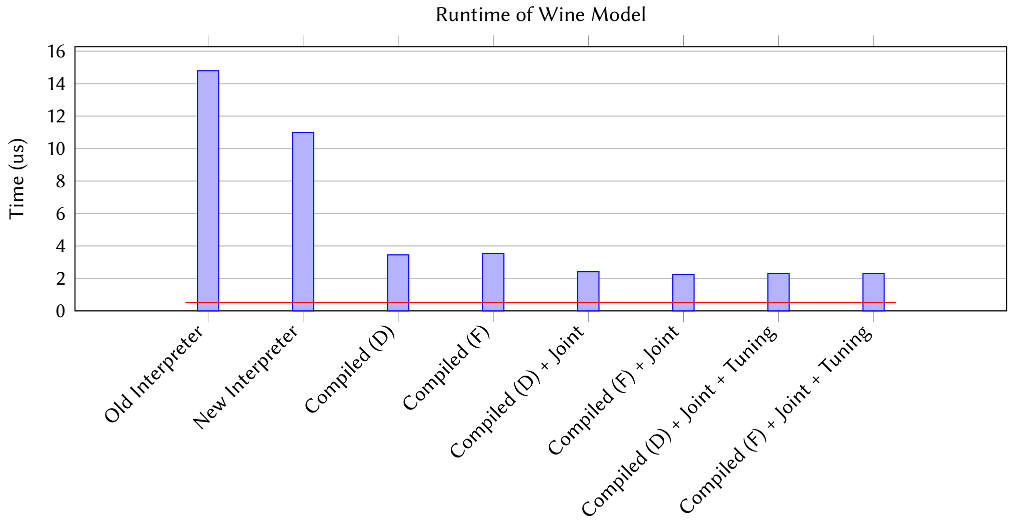
For the following evaluations, we use the batched interface provided by the compiled library using a batch size of 1. Higher performance for single evaluations could have been achieved using a dedicated single-evaluation interface, but would have proven more expensive to maintain in the long term.

We compare to a baseline that is a C++ interpreter implementation of the interpolation algorithms following the descriptions in Gupta et al. [13], with the additional speed-up that our baseline uses a fixed auxiliary LUT as described in Section 4.1 for each PWL but with no transform T, fixed $B = 50$ uniform buckets, $\alpha = k_1$, $\beta = (k_n - k_1)/50$.

We benchmarked this baseline C++ interpreter code at approximately 100× faster than vanilla TensorFlow Lattice [29] for single-evaluations on a two-layer calibrated lattice model with 4 inputs (the model was just 4 PWLs followed by a four-dimensional lattice interpolated with multilinear interpolation). TensorFlow does get more efficient when evaluated on batches: for a batch size of 4,000 examples, the amortized runtime was only 13× slower than our C++ interpreter baseline. Additionally, TensorFlow Lattice does not support simplex interpolation. As a result, we did not perform direct comparisons against TensorFlow Lattice.

8.1 Simplex Interpolation Experiments

Fig. 6 shows runtime results for two models where the lattices are interpolated with simplex interpolation. For these we show the runtime of the baseline interpreter, the interpreter with our proposed simplex kernel that uses bit-packing, and a compiled implementation with all of our proposals (faster PWLs and faster simplex), all for both double and float. The proposed bit-packing for the simplex kernel does lose a small amount of precision. The worst observed deviation in model



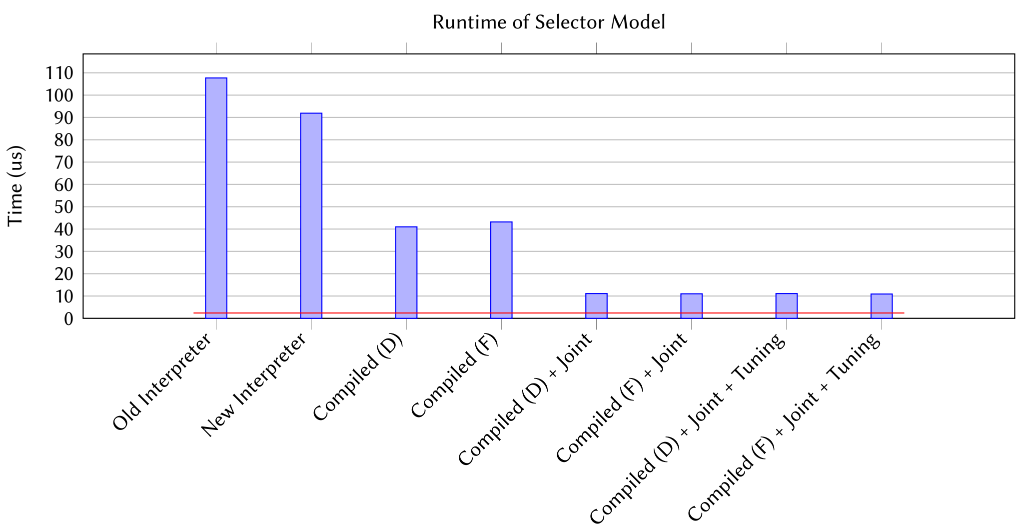
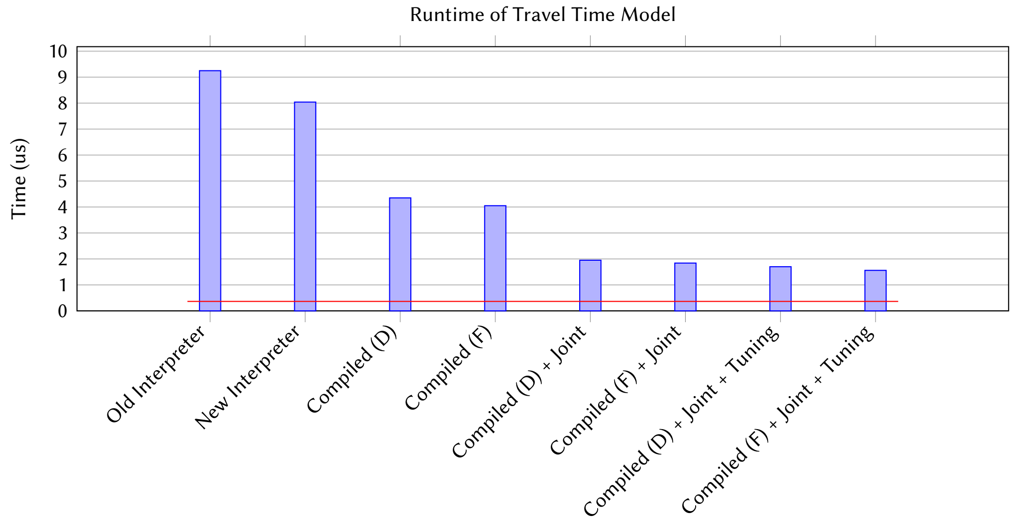


Fig. 6. The red line denotes a theoretically optimal implementation, based on the measured instructions per cycle from performance counters. "Joint" refers to the optimization in Section 4.2, and "Tuning" refers to the optimization in Section 4.1. **Upper:** Wine Model. The optimized solution provides a 6.5× speedup over the original interpreter on batch size 1, increasing to 9.1× on large batches (see Figure 8). **Lower:** Selector Model. The optimized solution provides a 9.88× speedup over the original interpreter on batch size 1, decreasing to 5.95× on large batches (see Figure 8). We note that this is due to the reference interpreter amortizing away significant amounts of overhead, while the compiled version's improvements are smaller.

output when compared to the C++ implementation was 10^{-13} for double-precision evaluation and 10^{-4} for single-precision.

The left results in Fig. 6 are for the Kaggle *Wine* dataset [12]. There are 150 inputs, but all of them are Boolean features except for one continuous feature, which passes through five 40-piece PWLs. The second-layer is an ensemble of 50 lattices, each of which acts on 8 first-layer outputs. In



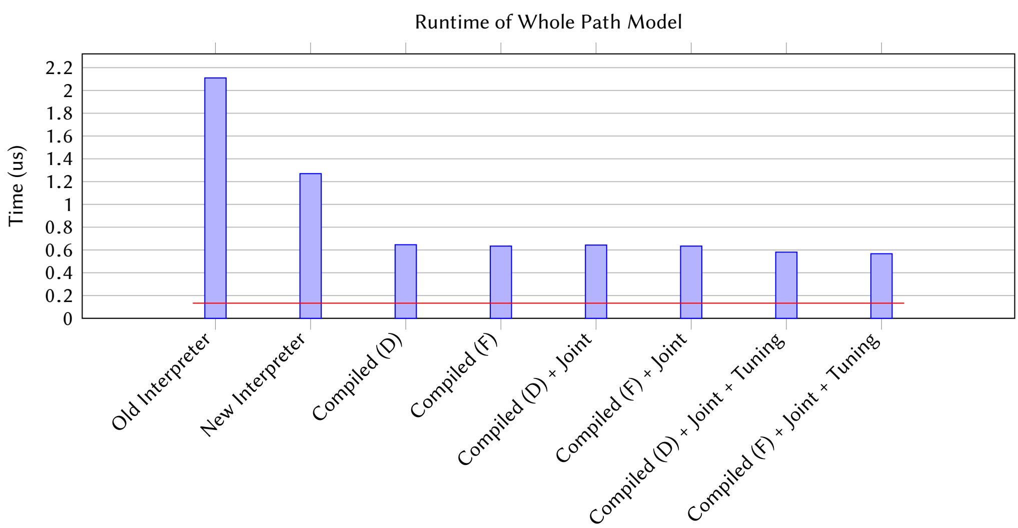


Fig. 7. The red line denotes a theoretically optimal implementation, based on the measured instructions per cycle from performance counters. “Joint" refers to the optimization in Section 4.2, and “Tuning" refers to the optimization in Section 4.1. **Upper:** Travel Time Estimation Model. The optimized solution provides a 5.9× speedup over the original interpreter on batch size 1, increasing to 9.1× on large batches (see Figure 8). **Lower:** Whole Path Model. The optimized solution provides a 3.7× speedup over the original interpreter on batch size 1, increasing to 4.3× on large batches (see Figure 8).

total, the model has roughly 3,240 parameters. Runtime was compared on 84.6k IID examples, and the proposals delivered a speed-up of 6.5×.

The right results in Fig. 6 are for a proprietary *selector* model that predicts whether a certain database should be queried for results in response to a given query. The model is an ensemble of 200 calibrated lattices, and each lattice acts on 8 out of the 30 possible features, so each of the 30 inputs is mapped through an average of 53.33 different PWLs in the model’s first layer. The

1,600 PWLs each have an average of 15 pieces each. The 200 lattices are each 2^8 multi-dimensional look-up tables. In total, the model has roughly 36,800 parameters. Runtime was compared on 650k IID examples, and the proposals delivered a speed-up of 9.88×.

8.2 Multilinear Interpolation Experiments

Fig. 7 shows runtime results for two models where the lattices are interpolated with multilinear interpolation. For these, we show the baseline interpreter runtime, and the runtime for a compiled implementation with all of our proposals (faster PWLs and latency hiding for multilinear) for both double and float.

The left results in Fig. 7 are for a proprietary model that predicts how long it will take a car to travel a stretch of road. The model is a 4-layer model on 39 inputs, where the first layer passes the 39 inputs through 156 PWLs (each input goes through 4 different PWLs), and each PWL has 50 pieces. The second layer is a linear embedding that maps the 156 calibrated inputs down to four dimensions, followed by another calibrator layer of 4 PWLs, then the fourth layer takes those four inputs and fuses in another four inputs using a 2^8 multidimensional LUT and multilinear interpolation. In total, the model has roughly 8,688 parameters. Runtime was compared on 94k IID examples, and the proposals delivered a speed-up of 5.9×.

The right results in Fig. 7 are for a proprietary model that fuses travel time estimates for different parts of a route into a travel time estimate for the whole route. This model is a 2-layer calibrated lattice model on 8 inputs, where the 8 PWLs each have 100–168 pieces, followed by a 2^8 multidimensional LUT with multilinear interpolation. In total, the model has roughly 1,264 parameters. Runtime was compared on 4 million IID examples, and the proposals delivered a speed-up of 3.7×.

8.3 Batch Performance

Fig. 8 shows that our proposals provide similar performance improvements when applied to batches of samples as well. The figure also shows that larger batch sizes do not substantially reduce runtime in this setting. This data as well as other profiling information suggest that the amortizable overhead is likely around $20 - 30\%$, based on the decrease in time-per-inference batch size increases. These typically come from function call overhead and other bookkeeping. Also, the Instructions Per Cycle for these models range from $1.8 - 2.4$, but are consistent across batch sizes. This indicates an opportunity to obtain increased performance across all batch sizes by performing fine-grained interleaving of kernels in order to expose more parallelism.

9 CONCLUSIONS AND OPEN QUESTIONS

This paper presents a set of state-of-the-art techniques for fast implementations of linear interpolation, using both operation-level optimizations and compiler transformations, together producing $3.7 - 10\times$ speed-ups compared to an interpreter-based implementation on several benchmark and real-world models. Our speed-ups reduced both fixed overhead costs as well as improved the efficiency of per-example computations.

ML models composed of interpolated lookup tables are attractive for their interpretability and model guarantees. Here we show such models can be evaluated on the order of microseconds and even nanoseconds without custom hardware, and are thus also well-suited to latency-sensitive tasks. We note here that this ultra-low latency setting poses a challenge not just for accelerator design, but also for the integration of the accelerator with the main processing units.

We focused on CPUs, which are cheap and readily available. Faster solutions may be possible with GPUs, but this is unlikely to be a net win due to the kernel launch latency of a GPU. Additionally, GPUs and other similar coarse-grained accelerators are likely to struggle with achieving good

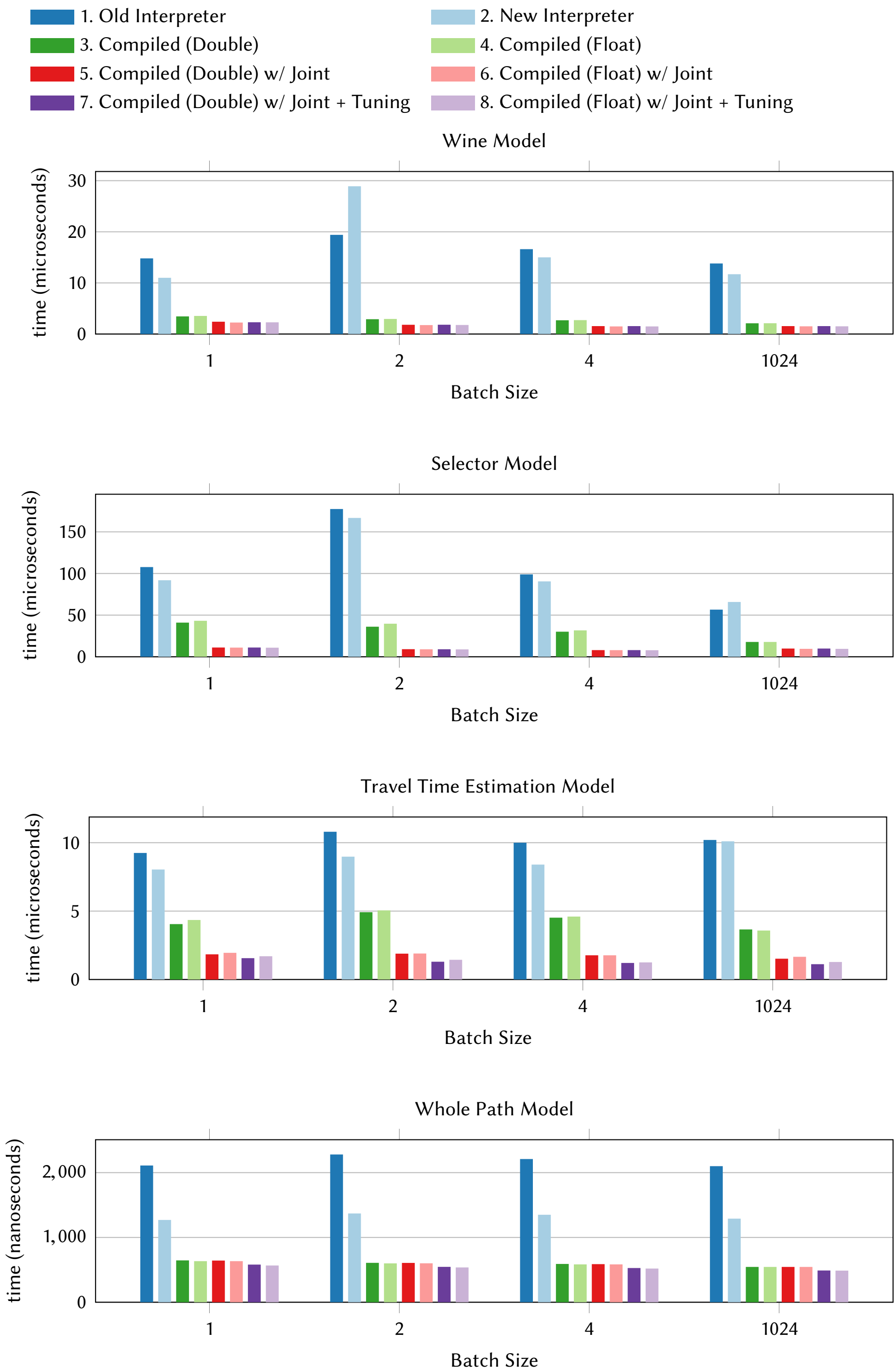


Fig. 8. Comparisons for average evaluation time per sample when evaluating batches of size 1,2,4, and 1024 samples. Bars in the plots are ordered as per the legend.

utilization on small operations, especially when considering the small-batch setting. We hypothesize that significantly faster speeds can be achieved with FPGAs and other spatial accelerators in settings where a high-throughput streaming solution is desired, due to far finer-grained reconfigurability compared to GPUs while eliminating the overhead due to flexibility in CPUs.

REFERENCES

[1] H. M. Aus and G. A. Korn. 1969. Table-Lookup/Interpolation Function Generation for Fixed-Point Digital Computations. *IEEE Trans. Computers* 18 (1969), 745–749.

[2] F. Bach. 2013. Learning with submodular functions: A convex optimization perspective. *Foundations and Trends in Machine Learning* 6, 2 (2013).

[3] K. E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference.* ACM, 307–314.

[4] K. Canini, A. Cotter, M. M. Fard, M. R. Gupta, and J. Pfeifer. 2016. Fast and Flexible Monotonic Functions with Ensembles of Lattices. *Advances in Neural Information Processing Systems (NeurIPS)* (2016).

[5] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation.* 578–594.

[6] A. Cotter, M. R. Gupta, H. Jiang, E. Louidor, J. Muller, T. Narayan, S. Wang, and T. Zhu. 2019. Shape Constraints for Set Functions. *ICML* (2019).

[7] W. Farr. 1860. On the Construction of Life Tables, illustrated by a New Life Table of the Healthy Districts of England. *Journal of the Institute of Actuaries* 9 (1860), 121–141.

[8] A. Fog. 2018. https://www.agner.org/optimize/microarchitecture.pdf.

[9] E. K. Garcia, R. Arora, and M. R. Gupta. 2012. Optimized Regression for Efficient Function Evaluation. *IEEE Trans. on Image Processing* 21, 9 (2012), 4128–4140.

[10] E. K. Garcia and M. R. Gupta. 2009. Lattice Regression. *Advances in Neural Information Processing Systems (NeurIPS)* (2009).

[11] Google. 2019. MLIR: Multi-Level Intermediate Representation for Compiler Infrastructure. https://github.com/tensorflow/mlir

[12] M. R. Gupta, D. Bahri, A. Cotter, and K. Canini. 2018. Diminishing Returns Shape Constraints for Interpretability and Regularization. *Advances in Neural Information Processing Systems (NeurIPS)* (2018).

[13] M. R. Gupta, A. Cotter, J. Pfeifer, K. Voevodski, K. Canini, A. Mangylov, W. Moczydlowski, and A. Van Esbroeck. 2016. Monotonic Calibrated Interpolated Look-Up Tables. *Journal of Machine Learning Research* 17, 109 (2016), 1–47.

[14] M. R. Gupta, E. Louidor, N. Morioka, T. Narayan, and S. Zhao. 2020. Multi-dimensional Shape Constraints. *In review, ICML* (2020).

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. 770–778. https://doi.org/10.1109/CVPR.2016.90

[16] A. Howard and T. Jebara. 2007. Learning monotonic transformations for classification. In *Advances in Neural Information Processing Systems.*

[17] Zhihao Jia, Sina Lin, Charles R. Qi, and Alex Aiken. 2018. Exploring Hidden Dimensions in Parallelizing Convolutional Neural Networks. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018.* 2279–2288. http://proceedings.mlr.press/v80/jia18a.html

[18] P. Khuong. 2012. Binary Search *Eliminates* Branch Mis-predictions. https://www.pvk.ca/Blog/2012/07/03/binary-search-star-eliminates-star-branch-mispredictions/

[19] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. 2018. The Case for Learned Index Structures. *SIGMOD* (2018).

[20] N. Manjikian. 1997. Combining loop fusion with prefetching on shared-memory multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing.* 78–82.

[21] T. Mowry. 1991. Tolerating Latency Through Software Controlled Data Prefetching. *PhD Thesis* (1991).

[22] E. P. O'Grady and B.-K. Young. 1991. A Hardware-Oriented Algorithm for Floating-Point Function Generation. *IEEE Trans. Computers* 40 (1991), 237–241.

[23] J. Perry. 1899. *Practical Mathematics.* Wiley and Sons.

[24] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907

[25] R. Rovatti, M. Borgatti, and R. Guerrieri. 1998. A geometric approach to maximum-speed n-dimensional continuous linear interpolation in rectangular grids. *IEEE Trans. on Computers* 47, 8 (1998), 894–899.

[26] E. Sang. 1875. On Last-Place Errors in Vlacq's Table of Logarithms. *Proceedings of the Royal Society of Edinburgh* 8 (1875), 371–376.

[27] G. Sharma and R. Bala. 2002. *Digital Color Imaging Handbook.* CRC Press, New York.

[28] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Hassan Chafi, Michael Wu, Anand R. Atreya, Kunle Olukotun, Tiark Rompf, and Martin Odersky. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML.*

[29] TensorFlow Blog. 2020. TensorFlow Lattice: Flexible, Controlled, and Interpretable ML. https://blog.tensorflow.org/2020/02/tensorflow-lattice-flexible-controlled-and-interpretable-ML.html

[30] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[31] S. Wang and M. R. Gupta. 2020. Deontological Ethics by Monotonicity Shape Constraints. In *AIStats*.

[32] A. Weiser and S. E. Zarantonello. 1988. A Note on Piecewise Linear and Multilinear Table Interpolation in Many Dimensions. *Math. Comp.* 50, 181 (Jan. 1988), 189–196.

[33] S. You, K. Canini, D. Ding, J. Pfeifer, and M. R. Gupta. 2017. Deep Lattice Networks. *Advances in Neural Information Processing Systems (NeurIPS)* (2017).