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

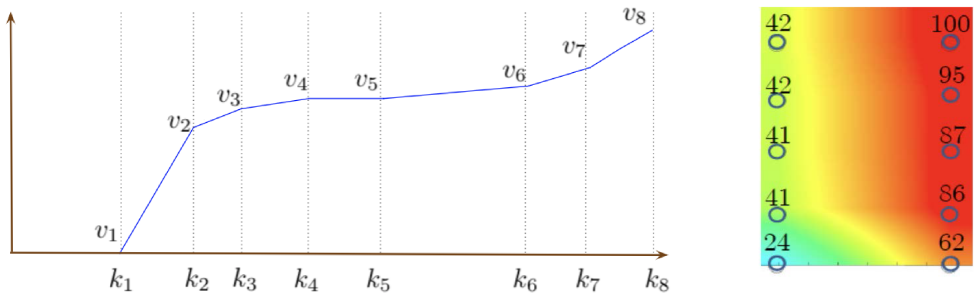
ACM Reference format:
Nathan Zhang, Kevin Canini, Sean Silva, and Maya Gupta. 2021. Fast Linear Interpolation. *J. Emerg. Technol. Comput. Syst.* 17, 2, Article 20 (April 2021), 15 pages.
https://doi.org/10.1145/3423184

1 INTRODUCTION

Linearly interpolated look-up tables (LUTs) are a core operation of piecewise-linear (PWL) functions and lattice models, which interpolate multi-dimensional look-up tables [10]; see Figure 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 [11, 28]. Recently, as concerns grow about the black-box nature of AI, PWLs and lattice models have become useful to machine learning practitioners, because they offer interpretable and semantically regularized machine learning by enabling shape constraints like monotonicity while still producing flexible, accurate models [4, 7, 12–14, 30, 32, 34].

In this article, we investigate how quickly one-dimensional and multi-dimensional LUTs can be interpolated on standard CPUs. In theory, interpolating LUTs can be very fast, because only

Authors' addresses: N. Zhang, Gates Computer Science Building, 353 Jane Stanford Way, Stanford, CA 94305; email: stanfurd@stanford.edu; K. Canini, Google Inc., 1600 Amphitheatre Pkwy, Mountain View CA 94043; email: canini@google.com; S. Silva, Google Inc., 1600 Amphitheatre Pkwy, Mountain View CA 94043; email: silvasean@google.com; M. Gupta, Google Inc., 1600 Amphitheatre Pkwy, Mountain View CA 94043; email: relativeentropy@gmail.com.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2021 Copyright held by the owner/author(s).
1550-4832/2021/04-ART20
https://doi.org/10.1145/3423184



PWL Lattice with 5×2 LUT

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

the LUT parameters nearest an input are needed to evaluate that input. This is in stark contrast to models like DNNs and CNNs, for which every model parameter might be touched repeatedly during the evaluation of a single example. However, ML models based on LUTs have many small operations (10 s of ns) that are easily bottle-necked by the cost of dispatching the computation, rather than the computation itself. We propose custom low-level optimizations and data-handling implemented using MLIR [20] and show that these optimizations achieve incredibly fast runtimes on standard CPUs.

The main contributions of this article 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 multi-dimensional linear interpolation methods (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.

In Section 2, we review some related work on compilation for ML. 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. Experiments in Section 7 show our proposals lead to $5–10\times$ speed-ups on real multi-layer machine-learned models. We conclude in Section 8 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, 25, 29, 31] 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 operations 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 article 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 [20] 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×. In the next sections, we show how to reduce the overall runtime by 5–10× 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 to accelerating these operations.

3.1 Piecewise-linear Functions (PWLs)

PWLs have been used to approximate and represent one-dimensional functions for centuries, for example, tables for logarithms [24, 27] and actuarial tables [8]. As illustrated in Figure 1, we define a PWL by N key-value pairs $(k_i, v_i)_{i=1}^{N}$, where the keys are sorted $k_i < k_{i+1}$.

3.1.1 Evaluating a PWL. To evaluate any input $x \in [k_1, k_N]$, its two 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 compute the interpolation weight on the jth keypoint:

$$w_j(x) = \frac{k_{j+1} - x}{k_{j+1} - k_j}, \tag{2}$$

and compute the interpolated value:

$$f(x) = w_j(x)v_j + (1 - w_j(x))v_{j+1}. \tag{3}$$

One nice property of PWLs for safe AI and interpretable AI [13, 16, 32] is that a PWL can be guaranteed to be monotonically increasing if its look-up table parameters are monotonically increasing ($v_i < v_{i+1}$).

3.1.2 Quantile Keypoints for PWLs. For machine-learned PWLs it is recommended [30] that one 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. We assume these quantile keypoints are fixed and not trained; that is, only their corresponding values $\{v_i\}$ are trained to fit the data [13]. 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, then 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.

3.1.3 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 Equation (1). Linear search over the N keypoints requires an expected number of comparisons $E[\#\text{ Comparisons}] = \frac{N+1}{2}$. We estimate the cost of doing a memory-load-and-compare at four cycles, due to the two to three cycle latency of a memory load. However, as we show next, the branch mispredictions dominate even for small N.

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 predictor will predict whether the for-loop over the keypoints will terminate, 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 can be wrong only once, but will be wrong once with probability $(N - 2)/(N - 1)$. Thus, the expected number of branch mispredictions is $E[\#\text{ Mispredictions}] = (N - 2)/(N - 1)$. Each branch misprediction typically costs around 17 cycles, depending on architecture [9].

Here, we assume that all parameters are within the L1-cache, because there are few enough parameters to be loaded together. Thus, in total, we estimate the overall expected number of cycles taken by linear search for the index j in Equation (1) to be $E[\#\text{ Cycles}] = 4 \times E[\#\text{ Comparisons}] + 17 \times E[\#\text{ Mispredictions}]$. Replacing with the estimates above, that becomes $E[\#\text{ Cycles}] = 2(N + 1) + 17(N - 2)/(N - 1)$. For $N < 6$ keypoints, the expected branch mispredictions dominate the expected cycles. The number of keypoints used for PWLs in real-world machine-learned calibrators is often in the range $N \in [5, 15]$.

3.1.4 Binary Search for PWLs. In a branch-free implementation of binary search with known depth (since N is known ahead of time), 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 six 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 $N \in [3, 10]$, and is thus the baseline that any proposed indexing must beat.

3.1.5 A Map-to-Index Function for PWLs. More abstractly, the key to efficient PWL evaluation is to construct a function that can quickly map an input x to the index j from (1). An old trick is to build an auxiliary LUT over $[k_1, k_N]$ with B uniformly spaced buckets, use that auxiliary LUT to map x to a bucket, and then only linearly search through all the keypoints that fell in that bucket. However, with irregularly spaced keypoints, an auxiliary LUT bucket could 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 [23] 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.

3.2 Multilinear Interpolation

Next, we review linear interpolation of a regular D-dimensional LUT. See Figure 1 for an example $D = 2$ dimensional LUT with a regular grid of 5×2 knots (keys), and Figure 2 for more examples of $D = 2$ LUTs on a regular 2×2 grid. An interpolated LUT is called a lattice.

Our real-world models in Section 7 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* [34].

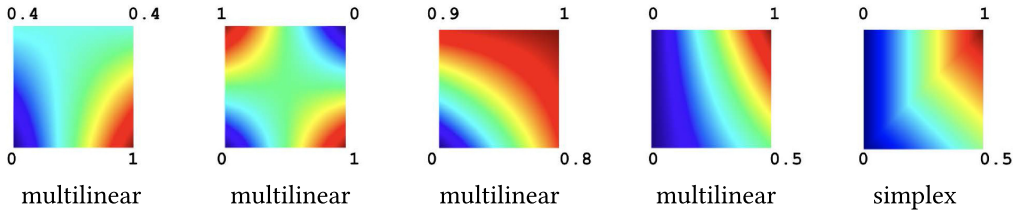


Fig. 2. Examples of two-dimensional lattices defined by 2×2 LUTs with different parameters, as shown. The functions go from 0.0 (blue) to 1.0 (red). The first four examples interpolate the LUT with multilinear interpolation, known as bilinear interpolation for the special two-dimensional case. The far-right example uses simplex interpolation instead. One can see in the simplex interpolation example that the function is linear on its two 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 an increasing function in the vertical direction, but is non-mononotonic in the horizontal direction.

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 [13, 30]. Monotonicity constraints have been shown to be useful for AI interpretability [13, 30], regularization [4], and making AI models more ethical [32]. Lattices can also be constrained to produce functions that satisfy other shape constraints [7, 14].

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 cell [28], but the most common is multilinear interpolation, which for $D = 2$ is the popular bilinear interpolation used to up-sample images. 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 Equation (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.

3.3 Simplex Interpolation

Simplex interpolation is a more efficient $O(D \log D)$ linear interpolation of a D-dimensional LUT cell that produces a locally linear surface. Simplex interpolation is the same as the Lovász extension in submodularity [2].

For each input $x \in [0, 1]^D$, the D components of x are first sorted, then 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, 26, 33].

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

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.

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 Figure 1 for a visual comparison of the differences between multilinear interpolation and simplex interpolation of the same LUT.

Gupta et al. [13] gave runtimes for lattices using either multilinear or simplex interpolation implemented in C++ on a single-threaded 3.5 GHz Intel Ivy Bridge processor: for $D = 4$ inputs both simplex and multilinear interpolation ran in about 50 ns, but for $D = 20$, simplex interpolation ran in 750 ns, and multilinear interpolation ran in 12 ms, around 15,000× slower than simplex. Despite the slower runtime for $D > 4$, multilinear interpolation might be preferred, because it produces a smoother surface. Further, 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>>`, then 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. First, in Section 4.1, we show how to construct a better auxiliary index-mapping function that takes into account the spacing of the keypoints. Then, in Section 4.2, we note that because ensemble or deep lattice models often pass the same input through multiple PWLs, one can remove redundant index searches.

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 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 j given in Equation (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

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

where $T : \mathcal{X} \to \mathbb{R}$ is some 1D 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 three operations for arithmetic and two 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 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. Note that the number of steps is dependent on the quality of the transform function T; in our experiments, we have found that the number of steps is small. Importantly, this also removes dependence on the actual distribution P in Equation (5); the procedure instead optimizes over the range of possible inputs $\mathcal{X}$.

With these choices, Equation (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,

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

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 three 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.

To solve Equation (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 Equation (6) for a given model usually takes 1–2 s 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 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 approximated 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 four PWLs will have the same k_i values. In this case, the work to determine j in Equation (1) is duplicated across all PWLs that act on the same input.

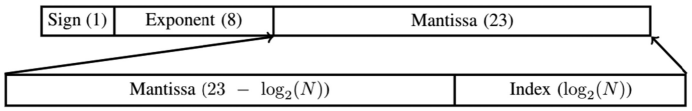


Fig. 4. We propose encoding the index in the low-order bits of the residual for higher sorting efficiency in simplex interpolation. For the double-precision floating point format the exponent instead has 11 bits and the mantissa has 53 bits.

To avoid this 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

We propose a new strategy for speeding up 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 [21, 22], 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

We show that simplex interpolation benefits from the use of *sorting networks* [3]. As described in Algorithm 2 of Gupta et al. [13], the simplex interpolation algorithm requires a sorting permutation π over the D inputs. For a D-dimensional LUT, the simplex interpolation only requires computing $D + 1$ 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. Thus, we propose using sorting networks [3], which construct sequences of max and min operations to generate a branch-free sorting implementation for inputs of fixed size. For example, at two operations per comparator, a 10-input sorting network uses 29 comparators, or 58 operations [6]. Due to superscalar execution, each comparator takes less than a single cycle, even when assuming a basic dual-issue CPU. As a result, even two mispredictions on ten inputs using a branch-based method exceeds the execution time of the sorting network.

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 Figure 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

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.

ϵ is the machine precision for the datatype. As a result, the interpolation output has relative error of at most $\epsilon\lceil\log_2(D)\rceil$.

7 EXPERIMENTAL PERFORMANCE 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 one. 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 (marked *Old Interpreter* on the graphs) that is a C++ interpreter implementation of the interpolation algorithms following the descriptions in Gupta et al. [13], with an additional speed-up for our baseline of using a fixed auxiliary LUT as described in Section 4.1 for each PWL but with identity transform T, fixed $B = 50$ uniform buckets, $\alpha = k_1$, $\beta = (k_n - k_1)/50$. *New Interpreter* refers to the old interpreter with updated simplex interpolation kernels and an improved Vector/Matrix interface to reduce overhead and improve interoperability with the generated code.

For multilinear interpolation, we benchmarked the baseline C++ interpreter code at approximately 100× faster than vanilla TensorFlow Lattice [30] for single-evaluations on a two-layer calibrated lattice model with 4 inputs (the model was just four 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. As of these experiments, TensorFlow Lattice did not support simplex interpolation.

We also compared to a rough estimate of peak achievable performance by combining the theoretical requirements of each kernel in Figure 5 with a conservative machine model. This estimate is denoted by a red line in Figures 6 and 7.

7.1 Simplex Interpolation Experiments

Figure 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), using both double and single precision. The proposed

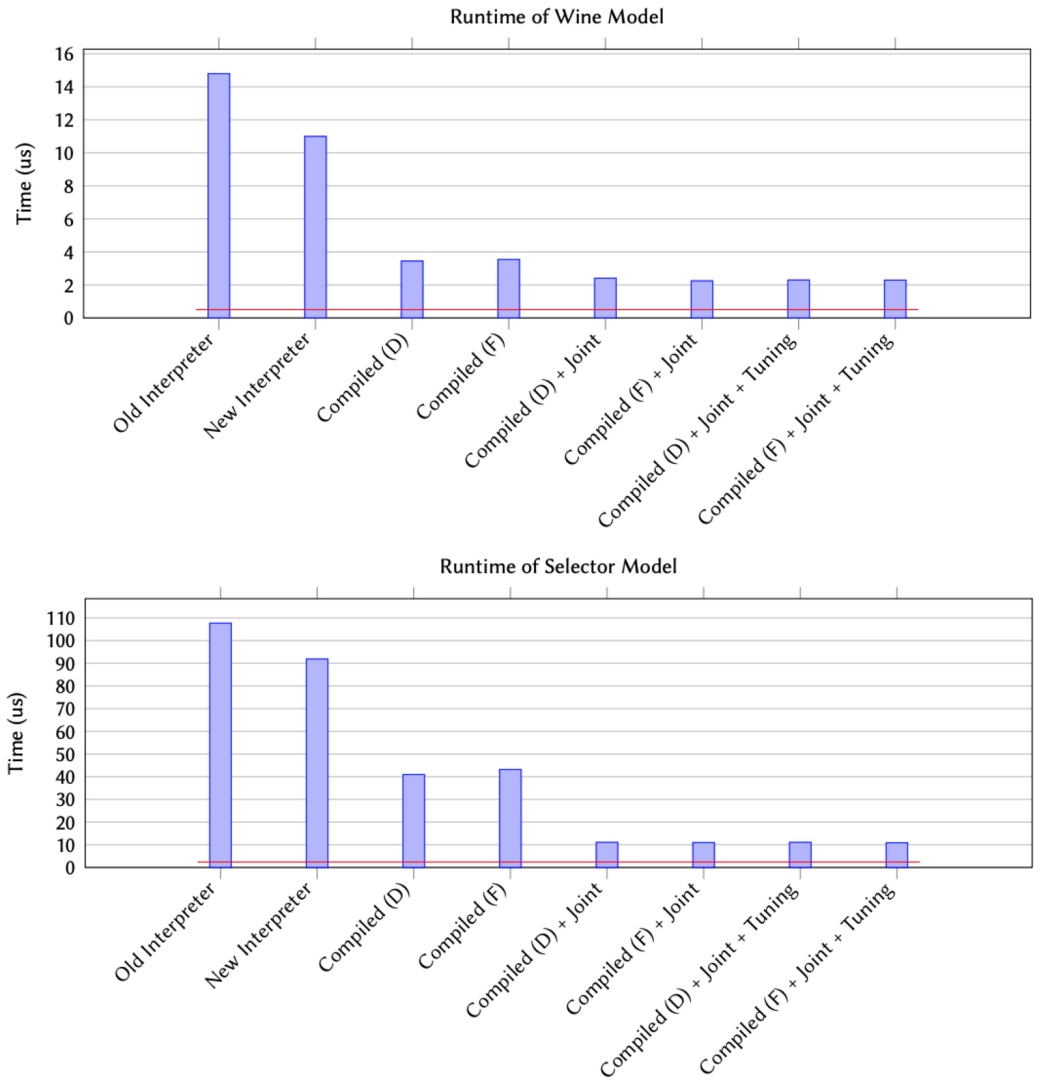


Fig. 6. **Simplex Interpolation Experiments:** The red line denotes peak performance as described in Section 7. "Joint" refers to the optimization in Section 4.2, and "Tuning" refers to the optimization in Section 4.1. (D) and (F) refer to double-precision and single-precision, respectively. **Upper:** Wine Model. The optimized solution provides a 6.5× speedup over the baseline interpreter on batch size one, 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.

bit-packing for the simplex kernel does lose a small amount of precision. The worst observed deviation in model output when compared to the C++ implementation was 10^{-13} for double-precision evaluation and 10^{-4} for single-precision. We considered these precision losses are acceptable for machine-learned predictions, which are inherently noisy.

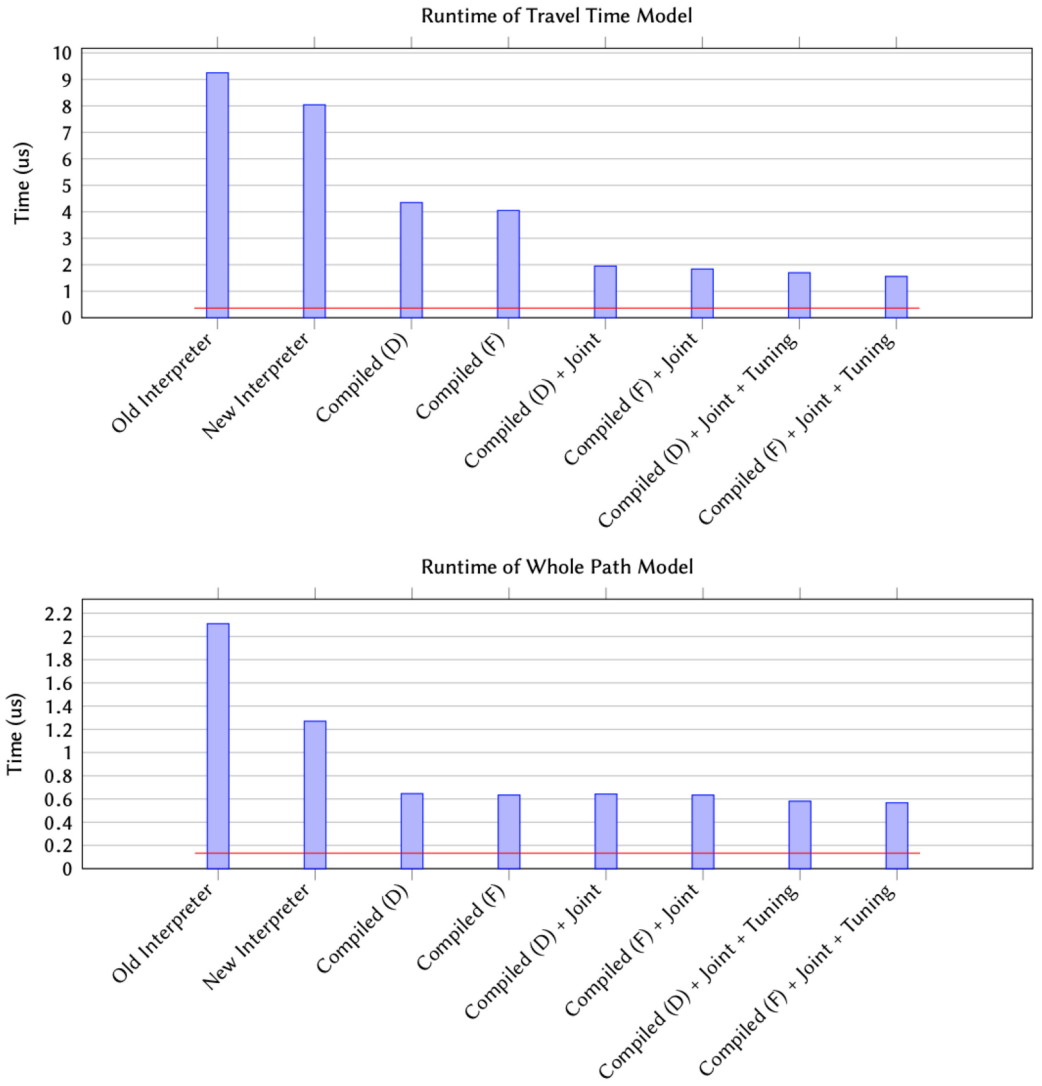


Fig. 7. **Multilinear Interpolation Experiments:**The red line denotes peak performance as described in Section 7. "Joint" refers to the optimization in Section 4.2, and "Tuning" refers to the optimization in Section 4.1. (D) and (F) refer to double-precision and single-precision, respectively. **Upper:** 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).

The top results in Figure 6 are for the Kaggle *Wine* dataset [12]. Our Wine model took 150 inputs, and all of them are Boolean features except for one continuous feature. The continuous feature was passed through five different PWLs to nonlinearly transform it five different ways; each of those five PWL's had 40 keypoints. The second-layer of this machine-learned model is an ensemble of 50 lattices. Each of the 50 lattices acts on a random draw of eight of the 150 inputs. In total, the model has roughly 3,240 parameters. Runtime was compared on 84.6k IID examples, and using all our proposals delivered a speed-up of 6.5× over the baseline interpreter.

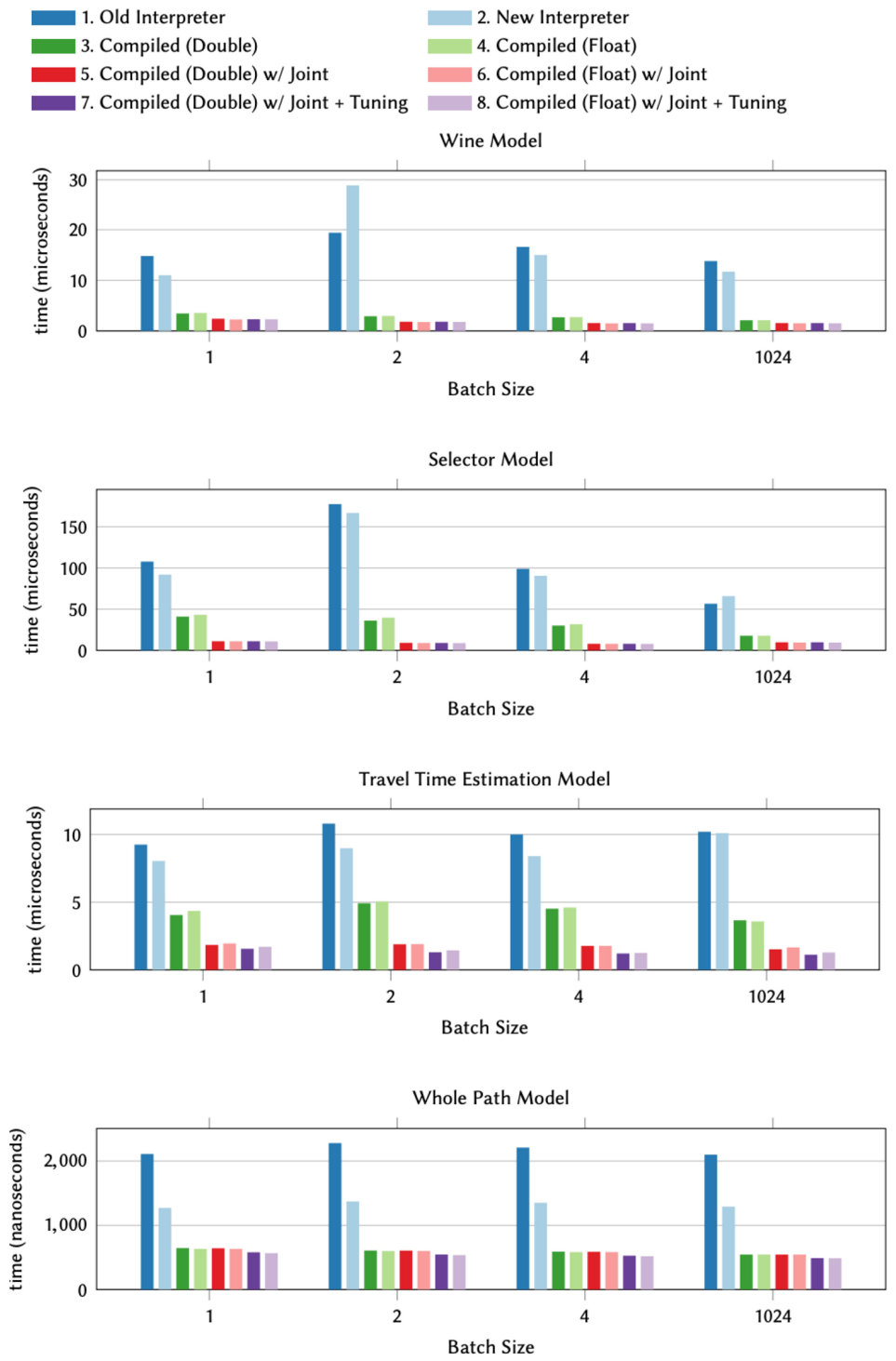


Fig. 8. Comparisons for average evaluation time per sample when evaluating batches of size 1, 2, 4, and 1,024 samples. Bars in the plots are ordered as per the legend.

The bottom results in Figure 6 are for a Google-proprietary *selector* machine-learned model that predicts whether a certain database should be queried for results in response to a given query. This is a three-layer model formed by summing an ensemble of 200 calibrated lattices, where each two-layer calibrated lattice acts on a random selection of 8 out of the 30 input 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 using all our proposals delivered a speed-up of 9.88× over the baseline interpreter. We note that tuning did not contribute significantly as the input keypoints for these models were already approximately equally spaced.

7.2 Multilinear Interpolation Experiments

Figure 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 top results in Figure 7 are for a Google-proprietary model that predicts how long it will take a car to travel a stretch of road. The model is a four-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 four 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 using all our proposals delivered a speed-up of 5.9× over the baseline.

The bottom results in Figure 7 are for a Google-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 two-layer calibrated lattice model on eight inputs, where the eight 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 using all our proposals delivered a speed-up of 3.7× over the baseline. We note here that tuning contributes an 11% speedup.

7.3 Batch Performance

Figure 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 to expose more parallelism.

8 CONCLUSIONS AND OPEN QUESTIONS

This article 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× 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 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. Comput.* 18 (1969), 745–749.

[2] F. Bach. 2013. Learning with submodular functions: A convex optimization perspective. *Found. Trends Mach. Learn.* 6, 2 (2013).

[3] K. E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of AFIPS*. ACM, New York, NY, 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. *Adv. Neural Info. Process. Syst.* 29, 1 (2016), 2927–2935.

[5] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of OSDI*. USENIX, 578–594. Retrieved from https://www.usenix.org/conference/osdi18/presentation/chen.

[6] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. 2014. Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten). In *Proceedings of ICTAI*. IEEE Computer Society, 186–193.

[7] Andrew Cotter, Maya Gupta, Heinrich Jiang, Erez Louidor, James Muller, Tamann Narayan, Serena Wang, and Tao Zhu. 2019. Shape constraints for set functions. In *Proceedings of MLR*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. 1388–1396. Retrieved from http://proceedings.mlr.press/v97/cotter19a.html.

[8] W. Farr. 1860. On the construction of life tables, illustrated by a new life table of the healthy districts of England. *J. Inst. Actuar.* 9 (1860), 121–141.

[9] A. Fog. 2018. Retrieved from https://www.agner.org/optimize/microarchitecture.pdf.

[10] Eric Garcia and Maya Gupta. 2009. Lattice regression. In *Advances in Neural Information Processing Systems*, vol. 22, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta (Eds.). Curran Associates, 594–602. Retrieved from http://papers.nips.cc/paper/3694-lattice-regression.pdf.

[11] E. K. Garcia, R. Arora, and M. R. Gupta. 2012. Optimized regression for efficient function evaluation. *IEEE Trans. Image Process.* 21, 9 (2012), 4128–4140.

[12] M. R. Gupta, D. Bahri, A. Cotter, and K. Canini. 2018. Diminishing returns shape constraints for interpretability and regularization. *Adv. Neural Info. Process. Syst.* 31, 1 (2018), 6835–6845.

[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. *J. Mach. Learn. Res.* 17, 109 (2016), 1–47.

[14] M. R. Gupta, E. Louidor, N. Morioka, T. Narayan, and S. Zhao. 2020. Multi-dimensional shape constraints. In *Proceedings of ICML* (2020).

[15] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep residual learning for image recognition. In *Proceedings of CVPR*.770–778. DOI: 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*. MIT Press.

[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'18)*. 2279–2288. Retrieved from http://proceedings.mlr.press/v80/jia18a.html.

[18] P. Khuong. 2012. Binary Search *Eliminates* Branch Mis-predictions. Retrieved from 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. In *Proceedings of SIGMOD'18*.

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. Retrieved from https://arxiv:cs.PL/2002.11054.
[21] N. Manjikian. 1997. Combining loop fusion with prefetching on shared-memory multiprocessors. In *Proceedings of the International Conference on Parallel Processing*. 78–82.
[22] T. Mowry. 1991. Tolerating latency through software controlled data prefetching. PhD Thesis, Stanford University.
[23] E. P. O'Grady and B.-K. Young. 1991. A hardware-oriented algorithm for floating-point function generation. *IEEE Trans. Comput.* 40 (1991), 237–241.
[24] J. Perry. 1899. *Practical Mathematics*. Wiley and Sons.
[25] 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. Retrieved from http://arxiv.org/abs/1805.00907.
[26] R. Rovatti, M. Borgatti, and R. Guerrieri. 1998. A geometric approach to maximum-speed n-dimensional continuous linear interpolation in rectangular grids. *IEEE Trans. Comput.* 47, 8 (1998), 894–899.
[27] E. Sang. 1875. On last-place errors in Vlacq's table of logarithms. *Proc. Roy. Soc. Edinburgh* 8 (1875), 371–376.
[28] G. Sharma and R. Bala. 2002. *Digital Color Imaging Handbook*. CRC Press, New York.
[29] 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 *Proceedings of ICML*.
[30] TensorFlow Blog. 2020. TensorFlow Lattice: Flexible, Controlled, and Interpretable ML. Retrieved from https://blog.tensorflow.org/2020/02/tensorflow-lattice-flexible-controlled-and-interpretable-ML.html.
[31] 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. Retrieved from http://arxiv.org/abs/1802.04730.
[32] S. Wang and M. R. Gupta. 2020. Deontological ethics by monotonicity shape constraints. In *Proceedings of AIStats*.
[33] 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.
[34] S. You, K. Canini, D. Ding, J. Pfeifer, and M. R. Gupta. 2017. Deep lattice networks and partial monotonic functions. *Adv. Neural Info. Process. Syst.* 30, 1 (2017), 2985–2993.

Received April 2020; accepted September 2020