Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:
Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

1 Compiling for modern accelerators

We became interested in this paper's subject when trying to improve an implementation of Capsule networks [1] to scale up to larger datasets. Capsule networks are an exciting machine learning research idea where scalar-valued "neurons" are replaced by small matrices, allowing them to capture more complex relationships. Capsules may or may not be the "next big thing" in machine learning, but they serve as a representative example of a disruptive ML research idea.

Although our convolutional Capsule model requires around 4 times fewer floating point operations (FLOPS) with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

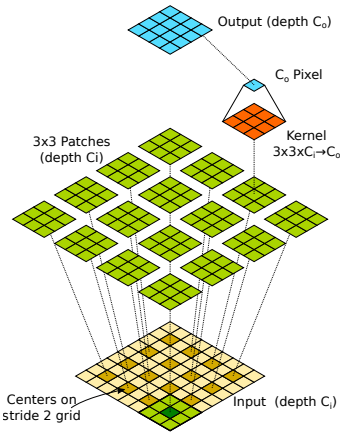


Figure 1. Conv2D operation with 3×3 kernel, stride=2

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is similar to the computation in a traditional CNN layer but operating on 4×4 matrices rather than scalars.

A basic building block of current machine learning frameworks is the strided 2D convolution. Most frameworks provide a primitive operation that accepts N input images of size $H \times W$, where each pixel has a "depth" of C_i channels[2]. Informally, for a "kernel size" K=3 and "stride" S=2, conv2d computes a weighted sum of overlapping 3×3 patches of pixels centered at every other (x, y) coordinate, to produce N smaller images with pixel depth C_o (Figure 1). Mathematically, this can be expressed as follows:

$$\forall n, x, y, c_o : O^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} I^{n,c_i}_{sx+k_x, sy+k_y} \cdot K^{c_i,c_o}_{k_x,k_y} \quad (1)$$

where $\cdot$ denotes scalar multiplication, and O, I, and K are all 4-dimensional arrays of scalars. The resulting code is little more than 7 nested loops around a multiply-accumulate operation, but array layout, vectorization,

[1] For an excellent tutorial on Capsule networks see [4].
[2] Section 4 discusses why these dimensions are used.

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05.
https://doi.org/10.1145/3317550.3321441

parallelization and caching are *extremely* important for performance [5].

The analogous computation for convolutional Capsules sums weighted "pose" matrices in 3×3 convolution patches to form "votes":

$$\forall n, x, y, c_o : \boldsymbol{V}^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} \boldsymbol{P}^{n,c_i}_{sx+k_x,sy+k_y} \cdot \boldsymbol{W}^{c_i,c_o}_{k_x,k_y} \quad (2)$$

where $\cdot$ now denotes matrix multiplication and $\boldsymbol{V}$, $\boldsymbol{P}$, and $\boldsymbol{W}$ are 4-dimensional arrays of 4×4 matrices, or equivalently, 6-dimensional arrays of scalars.

The following sections explain why ML frameworks make it hard to run the Capsule computation efficiently.

2 Compiling kernels is hard

Convolutional Capsule primitives can be implemented reasonably efficiently on CPU (see Table 1) but problems arise on accelerators (e.g., GPU and TPU). Performance on accelerators matters because almost all current machine learning research, and most training of production models, uses them. The marginal cost to perform a particular ML training or large-scale inference workload in a given time is much lower using accelerators than CPUs.

Accelerators have been very successful for machine learning workloads because the computationally expensive part of training tasks is written as dense linear algebra over multi-dimensional arrays. Dense linear algebra is regular compared to workloads that CPUs are designed for, and comparatively easy to parallelize. Consequently people have built increasingly complex accelerators designed for regular parallel computation. Example accelerator features include "warps", blocks, and grids of threads, very wide vector arithmetic units (ALUs), and systolic array multipliers (MXUs). As we explain next, it is hard to get good accelerator performance even on these regular computations. While frequently occurring computations receive attention and are well optimized, the performance of non-standard computations like convolutional Capsules suffers.

2.1 Compiling for accelerators

A major reason that it's hard to get good performance from regular computations is that the compiler has to consider the memory system of an accelerator as well as the ALUs. In an attempt to prevent data bottlenecks, accelerators' parallel capabilities have become tightly coupled with the memory system. For example [6]: peak ALU performance on GPUs requires "coalesced loads" where all 32 threads of a warp simultaneously access different values in the same cache line; implementations must be tailored to the sizes and strides implied by the organization of memory banks; and efficient programs must make use of all values loaded in a single memory access which may have large granularity.

```
def conv_capsule(float(B, H, W, CI, MH, MW) poses,
                 float(CI, CO, KH, KW, MH, MW) weights)
                 -> (votes) {
  votes(b, h, w, co, m, n) +=!
    poses(b, h*2 + r_kh, w*2 + r_kw, r_ci, m, r_k) *
    weights(r_ci, co, r_kh, r_kw, r_k, n) where r_k in 0:4
}
```

Figure 2. Tensor Comprehensions Capsules Code

In general accelerator code must perform explicit scheduling through the memory hierarchy rather than relying on transparent multi-level caches. Often memory access granularities require threads to cooperatively load each others' values and then exchange them[3]; so that code also contains complex instruction scheduling across loop iterations. While matching memory accesses to the parallel ALUs results in good hardware utilization, any mismatch can lead to orders of magnitude of performance slowdown [6]. Avoiding this slowdown requires tuning kernel parameters for e.g., padding, strides, and dimension layout, for each generation of each accelerator.

For "stencil computations" like convolution in which input values are reused by overlapping computation windows, scheduling loads and stores to optimize memory bandwidth is very challenging and has given rise to sophisticated tools such as Halide [7]. The data-reuse pattern in a convolutional capsule has several additional dimensions of complexity.

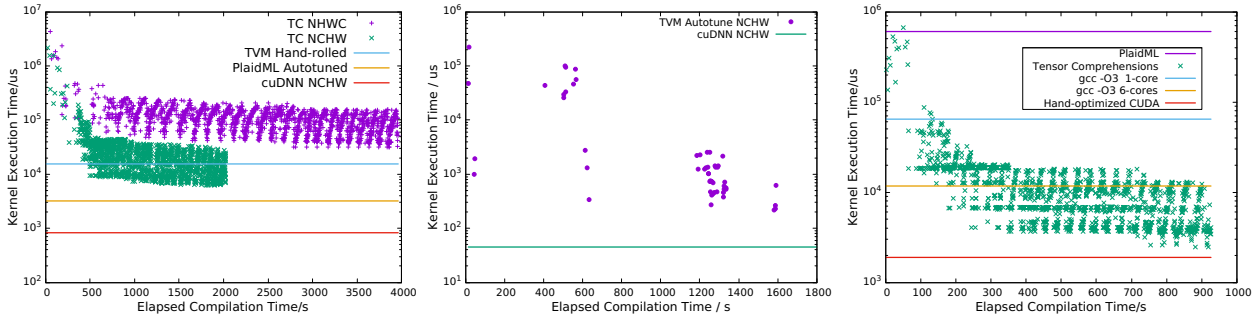
2.2 The monolithic kernel approach

Because of the difficulty of tuning parameters analytically, and the combinatorial number of choices, high-performance back ends for accelerators expend a lot of development effort on a small set of computational "kernels" (generally, isolated loop nests), such as 2D convolution and batch matrix multiplication, that dominate performance profiles of benchmarks. For each of these kernels the back end maintainers spend hours or days searching for the best algorithm and parameter settings for a small representative set of operand shapes[4], and then use heuristics or auto-tuning to select one of these pre-tuned implementations at runtime.

2.3 Compiling custom kernels

It is surprisingly common for machine learning papers to propose new primitives that cannot be computed efficiently with existing kernels. Compilers like Tensor Comprehensions (TC) [8] and PlaidML [9] have been developed to allow end-users to write such custom kernels, and both provide DSLs with concise syntax that resembles the math[5], e.g., compare the TC implementation of a capsule primitive in Figure 2 with Equation 2.

[3]Via so-called "warp shuffles".
[4]We use "shape" to mean the cardinality of an array's dimensions.
[5]Based on Einstein Notation.



(a) Strided conv2d (batch=32) **(b)** TVM autotuned conv2d (batch=1) **(c)** Capsule Kernel

Figure 3. Performance comparison of autotuned kernels

Despite the obvious appeal, the current state-of-the-art is that these tools are only really suitable for compiling small code fragments: compilation times are often long and the resulting code quality frequently does not come close to peak performance.

Figure 3a illustrates the difficulty that current compiler frameworks have generating GPU code for *conventional* 2D convolution whose performance competes with the carefully tuned library implementation in cuDNN [10]. In each case we wrote the `conv2d` implementation (Eqn. 1) using the lowest-level primitives available in the framework. TC uses a genetic search algorithm over optimization parameters and comes within a factor of 8 of cuDNN's performance, but only after an hour of search. Final performance is also very dependent on the memory layout of the input operands (NCHW vs NHWC). TVM [11] has a similar autotuned convolution template[6] that does not match cuDNN performance after 30 minutes of search (Fig. 3b). PlaidML [9] gets the best performance, just under 4× slower than cuDNN, with rapid compilation time[7], but uses heuristics that, as we show next, are brittle when the computation is more complex than simple convolution. TVM also has a hand-scheduled `conv2d` kernel for these operand shapes, but it is almost 19× slower than cuDNN.

Returning to our motivating Capsules example, we next tried implementing custom kernels for the core Capsules primitives (Eqn. 2). As a baseline, compiling the obvious C++ loop nests around a 4×4 `matmul` function with `gcc` produces good quality vectorized code that runs in around 60ms on a single x86 core and 11.7ms when parallelized across 6 cores with OpenMP. A hand-written CUDA implementation runs in 1.9ms but took over two days to manually tune.

Though PlaidML compiles as fast at `gcc`, the resulting kernel executes *much* slower[8]. Tensor Comprehensions takes nearly 3 minutes to find a kernel that outperforms the CPU, but eventually discovers a schedule that runs in 1.8ms (see Table 1 and Fig. 3c).

Compiler	Device	Compilation	Execution
gcc	x86 (1 core)	500ms	64.3ms
gcc -fopenmp	x86 (6 cores)	500ms	11.7ms
PlaidML	GTX1080	560ms	604ms
Tensor Comp.	GTX1080	3.2s	225ms
Tensor Comp.	GTX1080	64s	18.3ms
Tensor Comp.	GTX1080	1002s	1.8ms
CUDA	GTX1080	48h	1.9ms

Table 1. Convolutional Capsules Microbenchmark

Our interpretation of these results is that current frameworks excel at workloads where it makes sense to manually tune the small set of computations used by a particular model or family of models. Unfortunately, frameworks become poorly suited to research, because there is a performance cliff when experimenting with computations that haven't previously been identified as important. While a few hours of search may be acceptable before production deployment, it is unrealistic to expect researchers to put up with such compilation times (recall this is just one kernel in what may be a large overall computation); and even if optimized kernels were routinely cached locally, it would be a major barrier to disseminating research if anyone who downloaded a model's source code had to spend hours or days compiling it for their hardware before being able to experiment with it.

2.4 ML framework APIs are inflexible

As we will discuss in more detail in later sections, it is not straightforward to use custom computations in ML frameworks like TensorFlow and PyTorch. As a consequence, the easiest and best performing way to implement convolutional Capsules in both TensorFlow and Pytorch is to target high-level operations that are already supported by those frameworks. The best implementation we have found is the same for both frameworks:

[6]The `autotvm` template for `conv2d` does not support batching.
[7]PlaidML uses an analytical performance model to guide its search.
[8]We speculate that PlaidML's heuristics and performance model are not a good fit for more esoteric code.

- Materialize all of the 3x3 image patches (of 4x4 matrices), which with stride 2 almost doubles the input size.
- Shuffle this large tensor to rearrange dimension order to suit the matrix multiplication operator.
- Perform a large yet inefficient batch matrix multiplication (of many tall/skinny matrices).
- Shuffle the data layout back.
- Sum the resulting tensor over 3 (non-adjacent) dimensions.

To make things worse, between each layer of a Capsules model, votes are "routed" using the Expectation-Maximization (EM) algorithm [12], which repeatedly computes weighted means and variances of the votes using reductions over additional dimensions that could theoretically be fused into the above code (i.e., simply change a $\forall$ to a $\sum$ in Eqn. 2).

Unfortunately, neither framework is able to fuse the final reduction into the batch matrix multiplication (a pre-optimized kernel) which greatly increases the required memory bandwidth as well as intermediate storage requirements[9]. To compute two relatively small quantities the APIs force us to copy, rearrange and materialize to memory two orders of magnitude more data than strictly necessary.

It is issues like the above that have prevented us from finding any high performance implementation of convolutional Capsules to date.

3 Compiling programs is harder

In the preceding section we discussed the difficulty of performance-tuning a non-standard kernel. In practice programs must evaluate large graphs of kernels, and strategies for evaluating these graphs introduce more opportunities for optimization. In this section we discuss some of those strategies and point out ways in which the use of inflexible monolithic kernel implementations can constrain their optimizations.

3.1 Layout

Even for ML models as simple as ResNet [13], a sequential chain of 2D convolutions that has been extensively benchmarked, operands have different shapes so different convolution invocations may have different optimal parameters. If the layout differs between the producer and consumer operations of an intermediate value then the value must be expensively "transposed" (converted to a different layout). The use of pre-optimized kernels makes good layout assignment harder by constraining every operator to use one of a small number of layouts that have been chosen to be optimal *in isolation*. If kernel layouts were not constrained in this way a layout assignment algorithm might choose to use a "compromise" layout that is suboptimal for any given operator, but preferable to transposing intermediate values. In practice there are so few choices of layout available that frameworks like XLA [14] and TVM [11] do not attempt a global layout assignment, and instead choose fixed layouts for expensive operators like convolution, then propagate those layouts locally through the operator graph inserting transposes where necessary.

[9]TensorFlow's XLA compiler can fuse most operators, including reductions, but its GPU backend must still call cuBLAS kernels for matrix multiplication operations to be performance competitive.

3.2 Numerical precision

ML computations are unusual compared to many computing workloads in that they typically involve approximate floating- or fixed-point numbers. A target metric such as test set prediction accuracy may be attainable in more or less time, using more or less energy, by varying the choice of precision and/or non-uniform quantization used to represent intermediate values [15]. Precision can also affect the computation/communication bottleneck. When kernel implementations are fixed in library code it is not practical to include versions for all combinations of precision of inputs and outputs, reducing the opportunity to experiment with and optimize for different choices of quantized and low-precision types.

3.3 Interdependent global optimizations

We also note some additional whole-program optimizations which would be particularly useful for machine learning computations. The following optimizations are not made more difficult by the use of monolithic kernels, but are perhaps neglected because of the peephole optimization mindset adopted by current frameworks.

Common-subexpression elimination (CSE): CSE is surprisingly important for ML frameworks because of the computational structure introduced by backpropagation. Many backward pass operations use "activations" computed in the forward pass, and in the case of deep or recurrent neural networks the consumer of the activation is often far from the producer. Standard CSE favors materializing the activations to memory, but the limited amount of accelerator memory and its slowness relative to ALUs means that it may frequently be preferable to recompute activations instead. Thus CSE introduces another combinatorial search problem for frameworks: choosing which values should be materialized. Heuristics have been proposed to choose between materialization and recomputation [14, 16], but we are not aware of a system that tries to automatically make globally optimal choices about which values to materialize.

Distributed execution: Because of their data-parallel structure, machine learning computations can often usefully be distributed over multiple accelerators. In practice, compilers and machine learning frameworks typically expose distributed parallelism using mechanisms that are disjoint from those used within a device, for example offering only collective reduction and manual point-to-point messaging between devices. Despite initial research in this area [17], to our knowledge no framework tries to jointly optimize the choice of which fragments of a computation should run on which device with the choice of how to structure the subcomputations within a device.

3.4 Manual vs automatic search strategies

As explained in the preceding section, it is already hard to compile a single kernel in isolation, on a single device, with layout fixed in advance and relatively few choices for materialization. Optimizing an entire computation means also picking layouts and materialization points, and potentially distribution strategies, making the search space much larger.

Machine learning training algorithms are particularly dependent on automatic optimization strategies because the majority of the computation is typically performed in gradient operators that are synthesized from the "forward pass" implementation that appears in the source code. Since the code to compute the gradient is not written by the programmer, there is limited opportunity for programmers to guide its optimization: for example, there are no identifiers in scope for intermediate values computed as part of the gradient, to which programmers could manually assign layouts. Even in the absence of auto-generated gradients it is hard to write modular code with the manual optimization annotations used by, e.g., Halide [7] and TVM [11].

Recent research shows growing interest in automatic whole-program optimization techniques [18–20], but approaches are preliminary and typically focus on optimizing only one aspect of a program at a time. There is no doubt that multi-dimensional whole program optimization is a hard task, but we can perhaps take some hope from the recent success of hybrid search/learning approaches such as AlphaGo [21] that show promise in finding good solutions within huge combinatorial search spaces. It seems likely that it will be necessary to architect machine learning frameworks with automatic optimizers in mind before it will be possible to make the best use of whole-program optimization.

4 Evolving programming models

Thus far we have concentrated on code generation for accelerators, without much attention to programming models. We first observe that numerical computation benefits from features that are not present in traditional programming languages. Automatic differentiation is one such feature, and numerical programs are also unusual in that they are naturally written using functions that are polymorphic over the rank (number of dimensions) of their arguments. Consider again the standard convolution expression (Eqn. 1). The computations for each element of the batch (n) and output channel (C_o) dimensions are independent, and a natural way to express convolution would be in terms of a subcomputation

$$\forall x, y : O_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} I^{c_i}_{sx+k_x, sy+k_y} \cdot K^{c_i}_{k_x, k_y} \quad (3)$$

written in terms of 3-dimensional inputs I and K. A language could then automatically "lift" the function across batch and output channels if it were applied to inputs with more dimensions. Numerical languages at least as far back as APL [22] have included lifting for rank polymorphism, but there are plenty of open research questions on how to integrate such polymorphism with modern modular types and languages.

Recall that back ends are structured around calls to large monolithic kernels. In this section we argue that this back-end design approach is slowing progress in the maintainability, debuggability, and expressiveness of programming models. Worse, the resulting brake on innovation in languages is in itself reducing the incentive for back-end developers to improve on the current situation.

4.1 Opaque operators hurt extensibility

One consequence of monolithic back-end kernels is that front ends choose the kernel or "operator" as a point of abstraction. In popular frameworks like TensorFlow [23] and PyTorch [3], user programs are written in Python and call into operators[10] that are written in terms of back end-specific languages and libraries such as C++, MKL [24], CUDA [25], and cuDNN [10], or sometimes lower-level but portable domain-specific languages such as Tile [9] and Tensor Comprehensions [8]. When existing operators are not sufficient for a task, the user must descend into a lower-level language to write a new operator, and typically also manually write its gradient, a process which can be difficult and error-prone.

There are frameworks, such as Julia [26], which nominally use the same language to represent both the graph of operators and their implementations, but back-end designs can diminish the effectiveness of such a front end. In Julia, while 2D convolution is provided as a native Julia library, there is an overloaded `conv2d` function for GPU inputs which calls NVidia's cuDNN kernel. Bypassing this custom implementation in favor of the generic

[10] Current ML frameworks closely follow numpy/MatLab APIs.

code essentially hits a "not implemented" case and falls back to a path that is many orders of magnitude slower.

4.2 Opaque operators hurt modularity

A more subtle problem with monolithic kernels is that frameworks "commit" to a particular interface for an operator. As we observed in the introduction to this section, the `conv2d` operator includes a batch dimension *n* as well as the expected height, width and channels of the input image. Historically, `conv2d` has been used in mini-batch stochastic gradient descent when training CNNs and the kernel parameters can potentially be reused for each batch element of the image. Supplying a batch of images rather than a single image began as a performance optimization that has become fixed in the API.

Convolutions are in fact used in many other computations whose input may naturally have 3 or more than 4 dimensions, but to call `conv2d` the programmer must first transform the input into a "view" in which all the data-parallel dimensions are placed in the batch "slot", and afterwards re-transform the output to restore the computation's meaningful dimensions. These transforms can involve shuffling elements in memory, and have potentially large performance cost. Users thus add transforms in as few places as possible, which can make it very hard to keep track of the meaning of the dimensions of intermediate quantities.

As we suggested earlier, an alternative at the language level would be to supply a lower-dimensional `conv2d` function and lift it to higher dimensions where necessary. We believe this would greatly improve readability and maintainability. Monolithic back-end kernels are an obstacle to such an approach, because the language would have to automatically discover patterns that were equivalent to the monolithic kernels under some dimension ordering, and then automatically reorder dimensions before calling the kernel. Any failure of this pattern-matching would cause a severe performance cliff. We are encouraged by recent attempts to support a restricted form of lifting [27] but we believe progress would be much more rapid if there were a compiler that could emit efficient code for general-purpose lifted functions.

4.3 Ordered dimensions considered harmful

People have frequently advocated for "named dimensions", whereby the dimensions of an array are associated with a textual name along with, or instead of, their numeric position [28–31]. Named dimensions improve readability by making it easier to determine how dimensions in the code correspond to the semantic dimensions described in, e.g., a research paper. We believe their impact could be even greater in improving code modularity, as named dimensions would enable a language to move away from fixing an order on the dimensions of a given tensor, which in turn would make function lifting more convenient. (In APL, for example, where dimensions are strictly ordered, the rightmost dimensions are always lifted and an argument must be transposed to ensure the correct ordering before the function is called.)

In order to efficiently implement programs with named or unordered dimensions, we believe it will be necessary to rethink the design of the back end, for example adopting an IR that operates over unordered *sets* of dimensions, followed by a device-specific lowering to a concrete (ordered and optionally padded/tiled) memory layout. We are pleased to note that several projects [11, 14, 32] have taken preliminary steps in this direction by decoupling to some extent the dimension order of the source code from that of the lowering, although they still require the front end to specify ordered dimensions for every array.

5 A way forward

It is hard to experiment with front end features like named dimensions, because it is painful to match them to back ends that expect calls to monolithic kernels with fixed layout. On the other hand, there is little incentive to build high quality back ends that support other features, because all the front ends currently work in terms of monolithic operators. An end-to-end tool chain for machine learning requires solutions from many specialist disciplines. Despite impressive and sometimes heroic efforts on some of the sub-problems, we as a community should recognize that we aren't doing a great job of tackling the end-to-end problem in an integrated way. There are many sub-problems that we could be working on in an independent but coordinated way. In this spirit we present a possible research agenda:

- Language design, including automatic differentiation, using purely named dimensions and kernels expressed within the language syntax.
- A back end IR defining a graph of layout-agnostic general-purpose loop nests.
- Transformation passes for the above IR that lower it to a concrete CSE strategy, with the layout of each materialized intermediate.
- Compilation passes that generate accelerator code given the lowered IR above, producing adequate code quickly, and close to peak performance after searching.

We do not want to minimize the thought and engineering that has gone into current machine learning tool chains, and clearly they are valuable to many. Our main concern is that the inflexibility of languages and back ends is a real brake on innovative research, that risks slowing progress in this very active field. We urge our colleagues to bear this in mind when designing accelerators, tool chains, and especially benchmarks.

References

[1] G. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with em routing," in *International Conference on Learning Representations (ICLR)*, 2018.

[2] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, (Berkeley, CA, USA), pp. 265–283, USENIX Association, 2016.

[3] "PyTorch." https://pytorch.org/. Accessed 2019-01-09.

[4] "Understanding Matrix Capsules with EM Routing." https://jhui.github.io/2017/11/14/Matrix-Capsules-with-EM-routing-Capsule-Network. Accessed 2019-01-09.

[5] E. Georganas, S. Avancha, K. Banerjee, D. Kalamkar, G. Henry, H. Pabst, and A. Heinecke, "Anatomy of high-performance deep learning convolutions on simd architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, (Piscataway, NJ, USA), pp. 66:1–66:12, IEEE Press, 2018.

[6] "How to access global memory efficiently in CUDA C/C++ kernels." https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/. Accessed 2019-01-09.

[7] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 519–530, ACM, 2013.

[8] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018.

[9] "PlaidML." https://github.com/plaidml/plaidml. Accessed 2019-01-09.

[10] "cuDNN." https://developer.nvidia.com/cudnn. Accessed 2019-01-09.

[11] "End to end deep learning compiler stack." https://tvm.ai/. Accessed 2019-01-09.

[12] A. P. Dempster, N. M. Laird, and D. B. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *Journal Of The Royal Statistical Society, Series B*, vol. 39, no. 1, pp. 1–38, 1977.

[13] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *CoRR*, vol. abs/1512.03385, 2015.

[14] "XLA: Accelerated linear algebra." https://www.tensorflow.org/xla/. Accessed 2019-01-09.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.

[16] T. Chen, B. Xu, C. Zhang, and C. Guestrin, "Training deep nets with sublinear memory cost," *CoRR*, vol. abs/1604.06174, 2016.

[17] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *CoRR*, vol. abs/1706.04972, 2017.

[18] Z. Jia, M. Zaharia, and A. Aiken, "Beyond data and model parallelism for deep neural networks," in *Proc. Conference on Systems and Machine Learning*, SysML '19, 2019.

[19] Z. Jia, J. Thomas, T. Warszawski, M. Gao, M. Zaharia, and A. Aiken, "Optimizing DNN computation with relaxed graph substitutions," in *Proc. Conference on Systems and Machine Learning*, SysML '19, 2019.

[20] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, S. Johnson, M. Gharbi, B. Steiner, K. Fatahalian, F. Durand, and J. Ragan-Kelley, "Learning to optimize halide with tree search and random programs," in *ACM Transactions on Graphics*, SIGGRAPH '19, 2019.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, pp. 484–503, 2016.

[22] "APL programming language." https://en.wikipedia.org/wiki/APL_(programming_language). Accessed 2019-01-09.

[23] "TensorFlow." https://www.tensorflow.org/. Accessed 2019-01-09.

[24] "Intel Math Kernel Library." https://software.intel.com/en-us/mkl. Accessed 2019-01-09.

[25] "About CUDA." https://developer.nvidia.com/about-cuda. Accessed 2019-01-09.

[26] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *CoRR*, vol. abs/1411.1607, 2014.

[27] "Auto-vectorization with vmap." https://github.com/google/jax#auto-vectorization-with-vmap. Accessed 2019-01-09.

[28] "colah/LabeledTensor." https://github.com/colah/LabeledTensor. Accessed 2019-01-09.

[29] "Labels for TensorFlow." https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/labeled_tensor. Accessed 2019-01-09.

[30] "Tensor considered harmful." http://nlp.seas.harvard.edu/NamedTensor. Accessed 2019-01-09.

[31] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, and B. A. Hechtman, "Mesh-tensorflow: Deep learning for supercomputers," *CoRR*, vol. abs/1811.02084, 2018.

[32] "Multi-level intermediate representation." https://github.com/tensorflow/mlir. Accessed 2019-04-05.