Array Languages Make Neural Networks Fast

Artjoms Šinkarovs
Heriot-Watt University
Edinburgh, Scotland, UK
a.sinkarovs@hw.ac.uk

Hans-Nikolai Vießmann
Heriot-Watt University
Edinburgh, Scotland, UK
Radboud University
Nijmegen, Gelderland, NL
h.viessmann@ru.nl

Sven-Bodo Scholz
Heriot-Watt University
Edinburgh, Scotland, UK
Radboud University
Nijmegen, Gelderland, NL
svenbodo.scholz@ru.nl

Abstract

Most implementations of machine learning algorithms are based on special-purpose frameworks such as TensorFlow or PyTorch. While these frameworks are convenient to use, they introduce multi-million lines of code dependency that one has to trust, understand and potentially modify. As an alternative, this paper investigates a direct implementation of a state of the art Convolutional Neural Network (CNN) in an array language. While our implementation requires 150 lines of code to define the special-purpose operators needed for CNNs, which are readily provided through frameworks such as TensorFlow and PyTorch, our implementation outperforms these frameworks by *factors 2 and 3* on a fixed set of hardware — a 64-core GPU-accelerated machine; for a simple example network. The resulting specification is written in a rank-polymorphic data-parallel style, and it can be immediately leveraged by optimising compilers. Indeed, array languages make neural networks fast.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Computing methodologies** → *Machine learning*.

Keywords: machine learning, array language

ACM Reference Format:
Artjoms Šinkarovs, Hans-Nikolai Vießmann, and Sven-Bodo Scholz. 2021. Array Languages Make Neural Networks Fast. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '21), June 21, 2021, Virtual, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3460944.3464312

1 Introduction

With the increasing success of machine learning in various domains, scientists attempt to solve more and more complex problems using neural networks and deep learning. Increased complexity in the context of deep learning typically means more layers of neurons and larger training sets, all of which results in the necessity to process larger amounts of data. As a result, modern networks require advanced and powerful hardware — modern machine learning applications are envisioned to run on massively parallel high-throughput systems that may be equipped with GPUs, TPUs, or even custom-built hardware.

Programming such complex systems is very challenging, specifically in an architecture-agnostic way. Therefore, there is a big demand for a system that abstracts away architectural details allowing the users to focus on the machine learning algorithms. TensorFlow or PyTorch solve exactly that problem — they provide a convenient level of abstraction, offering a number of building blocks that machine learning scientists can use to specify their problems. Productivity of such a solution is quite high as these frameworks are embedded into high-level languages such as Python or C++.

However, turning a framework-based specification into an efficient code remains challenging. There is a huge semantic gap between the specification and the hardware. Frameworks such as TensorFlow and PyTorch build on many levels of abstractions, most of which offer alternatives in the way the framework is installed and run on a given system. Typically there is a Python front-end, a core library in C++ that depends on numerous external libraries for linear algebra, tensor operations, libraries for GPUs and other specialised hardware. Such complexity makes it challenging to deliver excellent performance: optimisations across multiple layers of abstraction, as well as across multiple external libraries, inherently come with overheads and hardware-specific trade offs.

The key question we are investigating is: can we identify a single layer of abstraction where on the one hand we can express the core building blocks and generate efficient parallel code, and on the other hand that is high-level enough to be used as a front-end.

Based on the observation that neural networks can be concisely expressed as computations on high-ranked tensors, we look into using a shape-polymorphic array language *à la* APL [20] as the central layer of abstraction. While APL itself is perfectly suitable in terms of expressiveness [47] the interpreter-based implementation of operators, unsurprisingly, does not readily provide parallel performance anywhere near that of TensorFlow or PyTorch.

ARRAY '21, June 21, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8466-7/21/06.
https://doi.org/10.1145/3460944.3464312

Over the last 50 years we have seen quite some research into compilation of array languages into efficient parallel code [3, 5, 17, 33, 36]. These languages leverage whole program optimisations and they offer decent levels of parallel performance. They also offer high program portability, as inputs are typically hardware-agnostic and all the decisions on optimisations and code generation are taken by the compiler. A user can influence these decisions by passing options, but no code modifications are required.

For the purposes of this paper we use SaC [33], a functional array language, as our implementation vehicle. To be clear, the presented ideas are not specific to the choice of the array language, see Section 2 and Section 4.4 for further details. We focus on a simple yet frequently benchmarked CNN for recognising handwritten characters. First we implement building blocks that are required to define the chosen CNN in native SaC and then we use these building blocks to define the network. We compare the resulting code size and performance against TensorFlow and PyTorch. We observe that the overall problem can be expressed concisely (300 lines of native[1] SaC code) and on a GPU-accelerated 64-core machine, our solution performs *two and three times faster* than the TensorFlow- and PyTorch-based implementations. The key aspect of such good performance is first-class support for multi-dimensional arrays in a functional setting followed by a number of well-known code-generation techniques used by the chosen compiler.

This example suggests that at least for this particular domain, the trade off between conciseness, performance and development time is quite satisfying.

The individual contributions of the paper are:

- we make a case for using array languages to host a machine-learning framework,
- we provide a concise implementation of the CNN for hand-written image recognition in SaC without using any domain-specific libraries, and
- we present a performance evaluation of the CNN in SaC against multiple variants of the PyTorch- and TensorFlow-based versions of the algorithm on a high-performance cluster node.

The rest of the paper is organised as follows. Section 2 briefly introduces machine learning algorithms and state of the art frameworks. In Sections 2 and 3 the notion of functional arrays and describe our implementation of the CNN are introduced. All the implementations used in the paper can be found in [39]. Section 4 presents a performance and productivity evaluation. Section 5 reviews related work, and we conclude in Section 6.

[1]The code does not depend on any specialised numerical libraries like MKL [19], only system libraries like libc or pthreads. We relate the SaC code with TensorFlow and PyTorch code in Section 4.1.

2 Background

In the last decade machine learning has attracted a lot of attention as it offers solutions to several practical problems that mainly have to do with automatic recognition of complex patterns: objects in images or videos, automatic text translation, recommendation systems, speech recognition, *etc.* We only focus on the computational aspects of machine learning algorithms and on CNNs in particular. For an in-depth review refer to [18, 31].

All machine learning algorithms are based around the idea that we want to learn (through *guessing*) a function f that maps given input variables X to given output variables Y, *i.e.* $Y = f\ X$, in the best possible way, according to some cost function. Once f is found for given samples of X and Y, we apply it to new inputs X.

Linear Regression. The simplest example of a machine learning algorithm is linear regression [9, 22]. It is probably one of the most well-understood algorithms in the area, yet it demonstrates fundamental principles that are used in CNNs as well. Given a set of n statistical units $\{y_i\ ,\ x_{i1}, \ldots, x_{im}\}$, for $i \in \{1, \ldots, n\}$, we assume that the relationship between ys and xs is linear, so that each y_i can be computed as: $y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_m x_{im} + \epsilon_i$. This can be written in matrix form as:

$$y = X\beta + \epsilon \qquad \text{where}$$

$$y = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} \quad X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1m} \\ \vdots & & \ddots & \\ 1 & x_{n1} & \cdots & x_{nm} \end{pmatrix} \quad \beta = \begin{pmatrix} \beta_0 \\ \vdots \\ \beta_m \end{pmatrix} \quad \epsilon = \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_n \end{pmatrix}$$

There exists a large number of methods to estimate or infer parameters β and ϵ such that our model function "best" fits the data. For example, one commonly used method is *linear least squares* [22]. We assume that $\epsilon = 0$ and the cost function that we want to minimise is: $\sum_{i=1}^{n} (y_i - \hat{y}_i)^2$ where $\hat{y}_i = X\beta$. The attractiveness of this method lies in the existence of the closed solution for the parameter vector β given by the formula: $\beta = (X^\top X)^{-1} X^\top y$.

Note two important aspects. First, instead of searching through all the functions from X to Y, we restrict the general shape of that function and introduce a set of parameters (β-s in our case). The search for a function reduces to the search for the parameters. Secondly, computationally, most of the involved operations can be reduced to linear algebra operations. This means that we will need a representation for vectors, matrices, tensors and common operations on them when implementing machine learning algorithms.

Neural Networks. Continuing on from linear regression, we can consider the function $f : X \to Y$ that we want to learn as a composition of functions g_i that can be further decomposed into smaller functions. Overall such a composition forms a graph (or *network*) connecting inputs X with outputs Y.

A typical function composition takes the form: $f\ x = A\ (\sum_i w_i\ (g_i\ x))$ where A is an activation function (usually it is chosen to be continuous and differentiable, *e.g.* sigmoid, hyperbolic tangent, *etc.*) and w_i are so called weights. These weights are parameters of our approximation that we want to find, similarly to β in linear regression, so that our cost function is minimised.

Usually, neural networks are designed in a way that offers slicing of the elementary functions g_i into layers, so that all the elements in the given layer can be computed independently. As a layer is an activation function of the weighted sum of other layers, most of the transitions in the network can be expressed as matrix or tensor operations.

Very often due to the size and complexity of the network, the closed solution that finds optimal weights either does not exist or is very difficult to find. Therefore, weight prediction is usually performed in an iterative manner. In this case, the concept of backpropagation — a method to calculate the gradient of the objective function with respect to the weights — is being applied. The key assumption is that we can improve an approximation of all weights w by recomputing w from the overall error and the gradient of F through $w := w - \eta\nabla F(w)$. In the cases when our objective function can be written as: $F = \sum_i F_i$, the gradient descent can be rewritten as: $w - \eta\nabla \sum_i F_i = w - \eta \sum_i \nabla F_i$. Furthermore, the stochastic gradient descent [51] approximates the true gradient as follows: $w := w - \eta\nabla F_i(w)$ which is typically more efficient. Intuitively, if we process a batch of items, we can update weights after processing one individual item. Finally, with carefully chosen activation functions A, the computation of the backpropagation can be expressed as a composition of linear algebra operations.

Chosen Problem. CNNs [18, 31], are neural networks where at least one layer is computed as a convolution of the values from the previous layers. In this paper we will implement a CNN and use it to recognise hand-written digits. We base our implementation on Zhang's network design [52]. For training and recognition we rely on the widely used MNIST data set[2] as input.

State of the Art Machine Learning Frameworks. The overall designs of state of the art machine learning frameworks such as TensorFlow [1], Caffe [21], CNTK [50], Torch [7], or PyTorch [28] are very similar. There is a core part written in C/C++ with the use of external libraries, and there is an interface part — usually a Python library. The core part contains highly-optimised kernels doing tensor operations, linear algebra operations, and convolutions, that are pre-optimised for the range of supported architectures. All these frameworks support computations on GPUs, multi-threaded and distributed executions. TensorFlow also supports custom hardware known as Tensor Processing Units (TPU).

The main difference between the frameworks lies in the number of building blocks that they provide which in turn influences the productivity of data scientists. For instance, Caffe and CNTK make it possible to specify networks via a configuration file allowing users to avoid programming entirely. Differences in the underlying libraries (BLAS, tensor libraries, GPU libraries) and optimisation techniques (XLA compiler, just-in-time compilation, kernel fusion) lead to runtime differences on the chosen hardware.

All frameworks have in common that they construct an internal representation of the dataflow graph of the network. This representation makes it possible to support automatic differentiation which automates the computation of gradient descents. Furthermore, such dataflow graphs are analysed in order to exploit natural concurrency of the network, optimise the scheduling of multiple network nodes across the available devices or threads, *etc.* In TensorFlow and CNTK the graph is statically fixed, whereas in PyTorch the graph can change at runtime.

The Essence of Array Programming. The underlying linear algebra of CNNs suggests that any implementation is amenable to a formulation based on multi-dimensional arrays. Any declarative array language as powerful as APL, the Ψ-calculus [26], or SaC can be used to express tensor operations.

Conceptually, all that is needed is an abstraction for n-dimensional arrays, with three basic primitives: selection, shape-enquiry and some form of n-dimensional map functionality. In SaC [13, 33], arrays can be constructed by using square brackets:

$$a = [1, 2, 3] \qquad b = [[1, 2], [3, 4], [5, 6]]$$
$$c = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]$$

It is assumed here, that all arrays are rectangular, *i.e.* all nestings are homogeneous, and expressions like $[[1, 2], [3]]$ are considered ill-formed. Each array has a *shape* which is a vector (1-dimensional array) denoting the number of elements per axis. For the above examples, we have:

$$\mathsf{shape}\ (a) = [3] \quad \mathsf{shape}\ (b) = [3, 2]$$
$$\mathsf{shape}\ (c) = [2, 2, 2]$$

All expressions are considered arrays — empty arrays as well as scalar values also have shapes:

$$\mathsf{shape}\ ([]) = [0] \qquad \mathsf{shape}\ ([[]]) = [1, 0]$$
$$\mathsf{shape}\ (42) = []$$

The shape of an empty vector is $[0]$; the shape of the 2D array containing one row that contains no elements is $[1, 0]$, and the shape of a scalar value is the empty vector. Selections

[2] see http://yann.lecun.com/exdb/mnist/.

have C-like syntax <array>[<iv>] (where <iv> is shorthand for <index-vector>) and the following two constraints:

1. shape (<iv>) $\dot{\leq}$ shape (shape (<array>))
   the length of the index vector can at most be as long as the array has axes, *i.e.* we are comparing two singleton vectors — the shape of the index vector must be element-wise less or equal ($\dot{\leq}$) to the shape of the shape of the array, and
2. <iv> $\dot{<}$ shape (<array>)
   the values of the index vector must be in range, *i.e.* element-wise less ($\dot{<}$) than the corresponding shape elements.

In case <iv> has maximal length, the corresponding scalar element in <array> is selected. Otherwise, the selection pertains to the first axes of <array> only and returns a sub-array whose shape corresponds to those components of the shape of <array> for which no indices were provided. In case <iv> is empty, the entire array is selected.

Finally, SaC provides a data-parallel array constructor for n-dimensional arrays named *with*-loop. For the context of this paper, we use its shorthand notation that we call *tensor comprehension* [34]. An n-dimensional array can be specified by an expression of the form:

{ <idx-var> -> <elem-expr> | <idx-var> < <shp-expr> }

where the shape of the result is determined by the value of <shp-expr>, and each element is computed by evaluating the expression <elem-expr>. SaC allows <elem-expr> to evaluate to non-scalar arrays, provided that all these expressions are of identical shape. The shape of the overall result is the concatenation of <shp-expr> and shape (<elem-expr>). For example, we have:

$$\{\,\texttt{iv -> 1} \mid \texttt{iv} < [3]\,\} = [1,1,1]$$
$$\{\,\texttt{iv -> } [1,2] \mid \texttt{iv} < [2]\,\} = [[1,2],[1,2]]$$

The index variable can be referred-to in the element expression, *e.g.* an expression of the form { iv -> a[iv]+1 | iv < shape (a)} computes an array that has the same shape as a given array a but whose elements have been increment by one. This notation is an extended version of the set-expressions in [15]; it has been implemented in the latest version of the SaC compiler and will be available in the next release.

More on SaC. We capture the set of assumptions in SaC that enable a compiler to generate efficient code. Firstly, SaC is a first-order functional language. This means that all the functions are pure, and all data is immutable. Conceptually, every assignment copies its right hand side and every function call copies its arguments. Such an assumption makes memory management completely transparent — there is no way to force a memory allocation, and there is no way to pass a pointer. The concept of pointers and references does not exist as it would break the assumption about purity. This makes all the optimisations much simpler as there is no need to solve aliasing or ownership problems. At runtime we avoid copying data that can be shared with the help of reference counting.

Secondly, the SaC compiler has multiple backends for generating code for sequential, multi-threaded and CUDA architectures from a single specification. No user annotations are needed to indicate parallel regions, as the *with*-loop per semantics exposes parallelism. Given that every iteration can be run concurrently, the compiler chooses which array comprehension will be run in parallel and generates either a multi-threaded version of the code or a CUDA kernel.

Finally, SaC uses C-like syntax for functions and comes with a rich standard library of pre-defined array operators similar to those available in APL.

3 CNN

In this section we describe our implementation of a CNN using [52] as a blueprint. It constitutes a typical CNN for recognising handwritten images of digits. Figure 1 shows the construction of the network which starts from a 28×28 pixel image of a digit and produces a 10-element $\hat{y}$ through a sequence of convolution and pooling layers. The vector $\hat{y}$ contains the probabilities of the input actually depicting the digits 0–9.

Convolution. In the first layer C_1, we compute six convolutions of the input image I with 5×5 matrices of weights $k^1_{1,i}$ producing six 24×24 arrays. One such convolution can be implemented as:

```
float[*] conv(float[*] I, float[*] k) {
  return { iv -> sum({ ov -> I[iv+ov] * k[ov]
                     | ov < shape(k) })
         | iv < shape(I) - shape(k) + 1 };
}
```

The type float[*] denotes an array of floating point numbers of arbitrary shape. For our image I of shape $[28, 28]$ and any of the weights $k^1_{1,i}$ of shape $[5, 5]$, the result is of shape $[28, 28] - [5, 5] + 1 = [24, 24]$. Each element at the index position iv is computed as a sum of 5×5 elements in I multiplied with the corresponding weights in k.

Using conv we can define a function mconv to compute the six convolutions and to add the individual biases b^1_i to each convolution (denoted as $\oplus$ in Figure 1):

```
float[*] mconv(float[*] I, float[*] k, float[*] b) {
  return { i -> conv(I, k[i]) + b[i]
         | i < shape(b)};
}
```

This function is rank-polymorphic, and in the context of the C_1 layer we chose to store all ks in a 3D array of shape $[6, 5, 5]$. One bias per convolution leads to the shape of b being $[6]$. For every index in b, mconv computes the convolution of I with k[i] that is adjusted by adding the bias

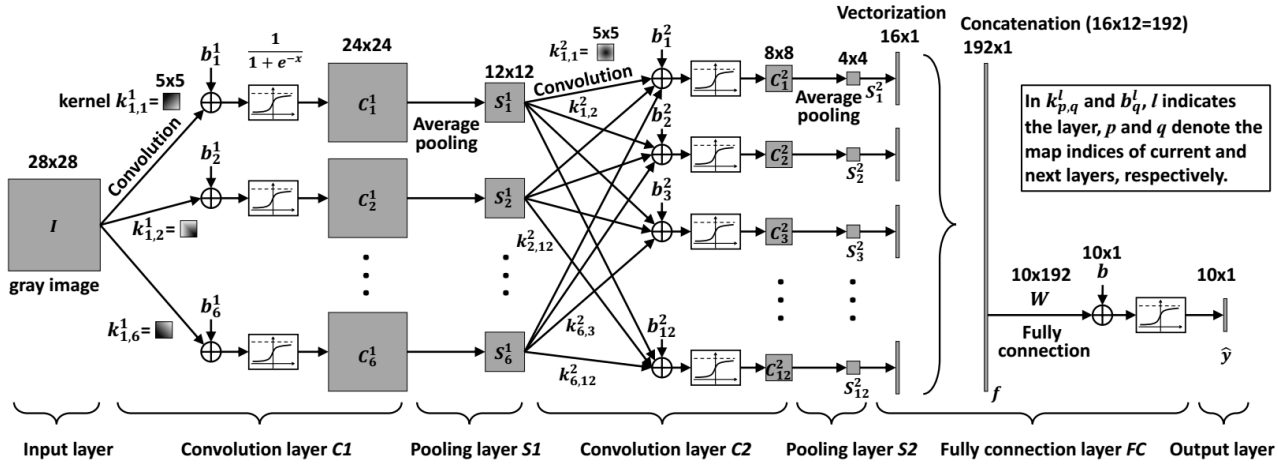


Figure 1. CNN for digit recognition. *The diagram is taken from [52]*

b[i]. The expression k[i] selects a [5, 5] sub-array at the corresponding index, and '+ b[i]' adds the scalar to every element of the [24, 24] array, resulting in the overall shape [6, 24, 24].

The last step in C_1 is the application of the sigmoid activation function to all values. We define it with the overloaded versions of mathematical functions provided in the standard library of SaC:

```
float[*] sigmoid (float[*] in) {
  return 1f / (1f + exp(-in));
}
```

This is a rank-polymorphic shape-preserving function, so its application to the result of mconv of shape [6, 24, 24] yields the desired result of the same shape.

Consider now the convolution layer C_2. If we choose to represent s as a 3D array of shape [6, 12, 12], our rank-polymorphic specification of mconv becomes immediately applicable here. Intuitively, if s is a single object, then all the left hand sides of the arrows from S_1 to C_2 in Figure 1 will merge into a single point, similarly to the first convolution. Our new input to conv is of shape [6, 12, 12], so each s^i should be of shape [6, 5, 5], producing a result of shape [1, 8, 8]. As we have 12 s^i and 12 biases, the application of the mconv would be of shape [12, 1, 8, 8]. Note that the second element in the shape can be eradicated by a simple reshape, which does not alter the data representation in memory or its computational efficiency.

Applying the same reasoning to the *FC* layer, we conclude that we can use mconv again. Without additional reshapes, the shape of the layer S_2 would be [12, 1, 4, 4]. A fully connected layer is a convolution with the weight that is identical to the shape of the input array. Therefore, as we intend to compute ten weighted sums of all the elements, W now has shape of [10, 12, 1, 4, 4]. This yields mconv to return a result of shape [10, 1, 1, 1, 1]. With these observations it becomes clear that the only parts of Figure 1 left to complete the implementation are the pooling layers.

Average Pooling. The pooling layer S_1 can be constructed in a two step process similarly to the convolution layers. An average pooling of a single image can be implemented as:

```
float[.,.] avgpool(float[.,.] in) {
  return { iv -> average({ ov -> in[iv*2+ov]
                         | ov < [2,2] })
         | iv < shape(in) / 2 };
}
```

We select sub-arrays of shape [2, 2] and compute their individual average, resulting in a matrix of half as many rows and columns as the input. Based on this definition, a generic version that applies avgpool to the two innermost axes of an n-dimensional array can be expressed as:

```
float[*] avgpool(float[*] in) {
  return { iv -> avgpool(in[iv])
         | iv < drop([-2], shape(in)) };
}
```

Note that for convenience we overload the name avgpool. With these few functions, we are now ready to define the whole network from Figure 1 as

```
float[10,1,1,1,1] forward(
                  float[28,28] I,
                  float[6,5,5] k1, float[6]  b1,
              float[12,6,5,5] k2, float[12] b2,
            float[10,12,1,4,4] fc, float[10] b) {
  c1 = sigmoid(mconv(I, k1, b1));
  s1 = avgpool(c1);
  c2 = sigmoid(mconv(s1, k2, b2));
  s2 = avgpool(c2);
  return sigmoid(mconv(s2, fc, b));
}
```

Explicit shapes in `forward` are given for documentation purposes, and they can be replaced with more generic shapes in case of more abstract networks.

The implementation so far suffices for using the network in *forward mode*, *i.e.* once suitable weights and biases are known, we can classify images. To adjust the weights, we use training inputs where we know the correct answer for every input image. The error in recognition is our cost function that we minimise by using stochastic gradient descent to adjust the weights.

Backpropagating Convolution. Our loss function has a form $\frac{1}{2}\sum(y - f(x,w))^2$, so its derivatives for w_i will have a form $\frac{\partial f}{\partial w_i} f(x,w) \sum y - f(x,w)$ according to the chain rule. That is, to adjust the weights, we multiply the error with the derivative of the network with respect to the weights. The linear nature of the convolution implies that the derivatives are constants, namely the input of the convolution itself. Consequently, we can approximate the error in the weights as a convolution of the input with the error:

```
float[*] backweights(float[*] d_out, float[*] in) {
  return conv(in, d_out);
}
```

The resulting deltas then can be used to adjust the corresponding weights for the next forward run. To cater for the imprecision, this is done by applying a factor, usually referred-to as *rate*.

```
...
weights = weights - rate*backweights(d_out,in);
...
```

Similarly, we can approximate the error of the bias as a sum of the error since the derivative of the bias is constant 1:

```
float[*] backbias(float[*] d_out) {
  return sum(d_out);
}
```

The trickiest bit of implementation is the propagation of the error back to the inputs of the convolution, which we need to feed into the computation of the next backpropagation layer. Mathematically, the derivatives are simply the weights. The challenge arises from the fact that the outer elements of the result are influenced by fewer weights than the inner elements. This distinction between inner elements and outer elements can be expressed by checking against out-of-bound accesses:

```
float[*] backin(float[*] d_out,
                float[*] k, float[*] in) {
  return {
    iv -> sum({ ov -> all(iv-ov >= 0)
                      && all(iv-ov < shape(d_out))
                         ? k[ov] * d_out[iv-ov] : 0f
              | ov < shape(k) })
    | iv < shape(in) };
}
```

All inner elements of the result are computed by summing up valid `k[ov] * d_out[iv-ov]` where `ov` ranges over weights and `iv` ranges over `in`. Recall that `shape(d_out)` is the same as `shape(in) - shape(k) + 1`. The `iv-ov` index may become negative when `iv = 0` and `ov != 0`, or larger than `shape(d_out)` when `iv` is maximal index and `ov = 0`. Therefore, for each summand, we check the validity of `d_out[iv-ov]` and replace out-of-bound elements with zero, that is a neutral element of the summation. While this code snippet is easy to understand, it computes a conditional at every index, which is harmful for performance. Fortunately, there is a standard technique to move such conditionals from the body into boundary expressions. Application of this technique to `backin` can be found in [39].

Backpropagating Average Pooling. Average Pooling usually is backpropagated by evenly spreading out the error across the indices that we have averaged across in the forward mode. In our example, we can express this as:

```
float[.,.] backavgpool(float[.,.] d_out) {
  return { iv -> d_out[iv/2] / 4f
         | iv < shape(d_out) * 2 };
}

float[*] backavgpool(float[*] d_out) {
  return { iv -> backavgpool(d_out[iv])
         | iv < drop([-2], shape(d_out)) };
}
```

With these main building blocks, the backpropagation can be implemented in a way very similar to that of the `forward` function shown above. Details can be found in the source code provided in [39].

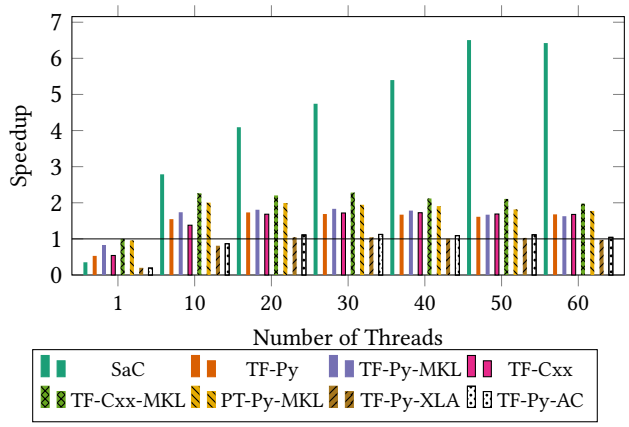
4 Evaluation

Provided is an evaluation of the CNN implementation in SaC, comparing it to semantically identical implementations in TensorFlow and PyTorch[3]. We discuss programming productivity reflecting our implementation experience, describe our experimental setup for runtime evaluation, and provide a performance analyses of the implementations.

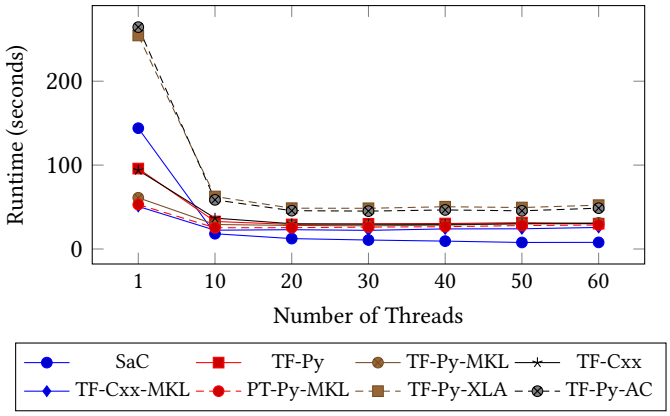
4.1 Effect on Programming Productivity

Programmer productivity is a very personalised topic as the background in tool familiarity influences the experience. The tools that we are comparing are of a different nature: SaC is a general-purpose language, whereas TensorFlow and PyTorch are specifically designed for machine learning purposes, both highly performance tuned and optimised for algorithms like the CNN. Neither of these tools executes the specification directly. Instead, the specification is analysed and translated into code that executes on parallel architectures.

[3]All three versions can be found in [39].



(a) Speedups (higher is better) over the fastest sequential runtime TF-Cxx-MKL.



(b) Wall-clock Runtimes (lower is better).

Figure 2. CPU only results for SaC *vs.* TensorFlow and PyTorch implementations using up to 64 Opteron cores, training 10 epochs on 10k images and classifying 10k images.

All three implementations of the CNN in SaC, TensorFlow and PyTorch are very similar. In all versions about 150 lines of code are needed to specify the network and to orchestrate the reading of inputs and data initialisation. In all three versions, the programmer needs to understand the abstractions used; in TensorFlow and PyTorch, the programmer needs to learn the semantics of the available components; in SaC, the programmer needs to understand the building blocks that we described in Section 2. Another difference is that SaC does not have any built-in support for automatic differentiation.

Further experience is based on having to write the building blocks ourselves in SaC. We found the conciseness of the building blocks very satisfying. The key components are described in Section 3 and can be implemented in about 150 lines of code. For someone with reasonable familiarity in SaC, this can be achieved within a few hours, depending on the familiarity with the underlying algorithm. Overall, the time we spent on the SaC implementation was considerably smaller than the time we needed to understand sufficient details about the TensorFlow and PyTorch frameworks. Of course this experience is hard to generalise, but we are rather sure that in cases where the hardware architecture and the machine learning algorithm is fixed, figuring out the details about the frameworks and implementing the algorithm from scratch is very likely to take comparable time.

4.2 Setup

Our machine is equipped with 4*x* AMD Opteron 6376 CPUs (for a total of 64 cores) and an NVIDIA K20 GPU (CUDA driver version 410.79). We use GCC 7.2.0, `sac2c` 1.3.3, CUDA 10.1, Python 3.6.6, TensorFlow 2.2.1 and PyTorch 1.6.0 for all applications.

We compile both frameworks from sources to make sure that the architecture specific flags like `-march=native -mtune=native` are passed to the C/C++ compilers so that we get proper vectorisation and cost models. Secondly, TensorFlow and PyTorch can make use of the oneDNN[4] library [19] to accelerate linear algebra operations on various architectures and provide a significant speedup. It is not compiled in by default for TensorFlow, and though we could make our comparison without it, we would not deem it a fair comparison. Therefore, we have verified that oneDNN version 1.6.0 is correctly included in both frameworks, which has led to a noticeable difference in runtime. To further ensure that we compare against the best possible TensorFlow configuration, we provide figures for both, with and without oneDNN activated (indicated by the `-MKL` postfix), and we implement the CNN using both the Python and C++ interface (indicated by the `Cxx` postfix). The latter is to check whether static compilation has an impact on performance. Additionally, TensorFlow can make use of XLA which generates a sequence of computation kernels from the network graph [11]. Unlike oneDNN, the kernels are generated specifically from the graph meaning information specific to the model can be used to optimise the kernels, such as fusing kernels together. It can be activated in two ways, either by explicitly marking functions or by activating it for the entire program. We evaluate these two forms of XLA as well, indicated in the figures by postfixes `-XLA` for explicit use and `-AC` for whole program use (*e.g.* auto-clustering). This gives us seven framework-based implementations, plus the one in SaC. We run these on both the CPU and GPU, and measure the wall-clock runtime of the entire application.

[4]Formally known as Intel MKL Library

Table 1. Best wall-clock runtimes (seconds) on the given hardware for each framework.

Framework	SaC	TF-Py	TF-Py-MKL	TF-Cxx	TF-Cxx-MKL	PT-Py-MKL	TF-Py-XLA	TF-Py-AC
Configuration	Opteron 50 threads	NVIDIA K20	NVIDIA K20	NVIDIA K20	NVIDIA K20	NVIDIA K20	NVIDIA K20	NVIDIA K20
Runtime	7.8	17.06	18.26	17.68	14.1	24.32	21.41	22.36

With non-oneDNN TensorFlow versions we set the number of threads via the session variables; for the oneDNN version, as the library uses OpenMP, setting the TensorFlow threads this way can quickly oversubscribe the system, so per our experiments the best TensorFlow-oneDNN runtime is achieved when TensorFlow threads are set to 1 and the `OMP_NUM_THREADS` environment variable is set to the chosen thread count. With SaC we control the number of threads by passing the `-mt` flag to the binary file; this flag is automatically created by the compiler when using the multi-threaded backend.

The applications are run using the following parameters: 10 epochs, 100 images batch size, 10000 training images and labels, and 10000 test images and labels. The backpropagation has a learning rate factor of 0.05, and we do not use any momentum.

4.3 Results and Analysis

Figure 2a shows speedup compared to the fastest sequential runtime and Figure 2b shows the runtimes in seconds. From Figure 2b we can see that there are big differences in the sequential execution time of the versions tested. The oneDNN enabled versions provide the fastest sequential time, the non-oneDNN versions are a factor of roughly 2 slower, SaC is a factor of 3 slower, whereas the XLA-enabled versions are a factor of roughly 5 slower.

In terms of scaling, the picture is different. Here, the oneDNN versions stop scaling at around 10 cores with a speedup of roughly 2 over their sequential runtime. The XLA versions stop scaling at 20 cores with a 5 fold speedup over their sequential version, reaching the same performance as the sequential oneDNN versions. The SaC version scales up to 50 cores with a speedup of 20 over the sequential SaC runtime which equates a speedup of 6.4 over the sequential oneDNN time.

The relatively poor absolute performance of the XLA version most likely can be attributed to the size of our network as XLA is optimised for larger and more complex networks [10].

According to [48], the speedup plateauing that we see for the TensorFlow and PyTorch applications is likely due due to how the nodes within the network graph are translated to threads. Some nodes have dependencies on outputs of other nodes, and so are scheduled differently to nodes that have no dependencies. This limits the degree to which work can be distributed across the threads, affecting the maximum amount of scaling possible. Changes to the design of the network, or using a completely different neural network can lead to different scaling. As SaC only translates *with*-loops to threads, and *with*-loops are guaranteed to be side-effect free, this leads to better scaling.

With TensorFlow we have two levels of parallelism, as the oneDNN operations spawn their own threads independently of the TensorFlow scheduler. We tried different combinations of threading configurations looking for the best possible performance. Using the default configuration, where TensorFlow and oneDNN use all cores, leads to oversubscription and degraded performance. Using combinations of values that match the number of logical cores on the system, such as 16 oneDNN threads and 4 TensorFlow threads, did not lead to better performance compared to just setting TensorFlow thread number to 1 and having oneDNN scale to all 64 logical cores. In any case we were not able to resolve the scaling plateau by this means.

Table 1 shows the best runtimes per application and the hardware configuration this was achieved on. The best runtime for SaC is 7.8 seconds using 50 CPU threads. All other applications have their best runtime on the GPU, with the TensorFlow C++ oneDNN implementation having the best runtime at 14.01 seconds. The SaC on the GPU performed poorly compared to the other applications, running 9× slower.

In addition to a runtime evaluation, we also tried to measure FLOP/s for the SaC and TensorFlow implementations. For SaC we achieved 22.9 GFLOP/s using 50 CPU threads. For TensorFlow we could not make a reliable measurement as both the CPU counters and our hand-counting of operations did not match up, with orders of magnitude difference. After closer inspection, looking for instances of aggressive vectorisation and other optimisations, we could not determine the cause for this discrepancy and so leave out any measurement for TensorFlow.

4.4 Source of Performance in SaC

The SaC compiler is quite sophisticated, employing hundreds of optimisation that run in a cycle. Explaining what exactly the compiler is doing to make the code run well is challenging. All we can do here is identify a few key components, in order to potentially apply the demonstrated capabilities in the context of other compilers and programming languages.

After looking at the intermediate states of the code, we identify the following necessary optimisations: *with*-loop folding [32], *with*-loop fusion [14], memory reuse [16, 46], and statically scheduled multi-threaded execution [13].

With-Loop Folding. The main idea behind *with*-loop folding is the list equality $map\, f \circ map\, g = map\, (f \circ g)$, which can eliminate the creation of intermediate arrays. In the tensor comprehension notation this equality enables the following transformation:

```
a = {iv -> f (iv) | iv < u};
b = {iv -> g (a[K (iv)]) | iv < v}
  ⇒ b = {iv -> g (f (K (iv))) | iv < v}
```

where K is an index mapping from the legal index set of b into the legal index set of a. In case u=v and K is the identity, we get exactly the above equality. However, very often u and v do not coincide. Nevertheless, the transformation is still valid. As long as K is injective and the array a is not needed anywhere else, the transformation always improves the performance of the generated code. Even in cases where K is only partially injective, the transformation is typically beneficial due to decreased pressure on the memory system. For more details see [32].

In the case of our CNN example, the most prominent application of *with*-loop folding is the merging of the addition of biases and the computation of sigmoid functions in layers C_1, C_2 and FC.

With-Loop Fusion. Here we apply a variant of the classical loop fusion optimisation to tensor comprehensions. The optimisation combines the body of two subsequent comprehensions with an identical iteration space. In the list-based setting this is also referred to as Tupling [6]; we can capture the transformation in our *map*-based analogy as:

$$(map\, f\ a, map\, g\ a) = unzip\, (map\, (f \Delta g)\ a)$$
$$\text{where} \qquad (f \Delta g)\ x = (f\ x, g\ x)$$

This formulation exposes how the traversal of a is being shared for the computation of the two results. Key to an efficient implementation in the context of lists is an optimisation of $map \circ unzip$ to avoid the creation of a list of tuples.

In terms of the tensor-comprehensions in SaC, this transformation can be stated as:

```
a = {iv -> f (iv) | iv < u};
b = {iv -> g (iv) | iv < u};
  ⇒ a, b = {iv -> f (iv), g (iv) | iv < u}
```

Again, we can see the sharing of the traversal, *i.e.* the iteration of iv. As with lists we need to avoid the creation of an array of tuples. However, the direct access nature of the array context renders this a trivial task; the SaC compiler ensures that the results are directly written into two separate arrays a and b.

Memory Reuse. This memory analysis makes it possible to do array operations in-place. For example, when we increment all the elements by a constant:

$$b = \{\ iv \text{ -> } a[iv] + 1 \mid iv < \mathsf{shape}\ (a)\ \}$$

we can avoid allocating new memory and reuse a, even in a parallel execution, provided that a is not used further in the program. The analysis becomes challenging when we consider reusing existing, but no longer referenced, arrays within the current scope [46]. The analysis needs to handle conditionals within the set expression or the access patterns of candidate arrays.

Statically-Scheduled Parallelism. Finally, our tensor comprehensions are data parallel by design. Therefore, it is relatively straight forward to generate the code that partitions the index space into chunks and runs each chunk in parallel. Unfortunately, there is a plethora of small details that makes it very hard to implement this efficiently [12]. First of all, one needs to choose the operations we want to run in parallel, and their granularity. Secondly, choosing a schedule even for a single array operation is challenging. Finally, thread synchronisation and memory management make a significant difference. By default we use static scheduling, a custom memory allocator and for each operation we decide to run in parallel we try to choose the chunking that maximises the work each active thread is doing.

5 Related Work

5.1 Array Languages

Directly or indirectly, APL [20] has influenced all existing array languages. At its core, APL provides a set of operators with a number of rules on how they can be composed. All operators are either unary or binary, first- or second-order functions, expressed with a single symbol, which gives a lot of expressiveness. For example, all the building blocks of our CNN can be expressed in 10 lines of code [47]. APL is an untyped language, so all errors will occur at runtime only. It comes only with an interpreter and all the operators are implemented as library functions, limiting cross-operator optimisations. That setup typically inhibits performance competitive with the performance presented in this paper as shown by the performance figures presented in [47].

Other array languages can be roughly divided into three groups: direct descendants of APL, grandchildren and further relatives. Direct descendants are languages like: J [37], K [49] or Nial [25]. They treat every object as an array (except maybe functions) and provide a large subset of APL operators. Typically, these languages come only with interpreters which, again, limits the optimisation space and performance.

The grandchildren like SaC, Futhark [17], Remora [35], Qube [42] are still array-oriented languages, but instead of providing built-in APL operators natively, they offer a few low-level constructs from which the operators can be implemented as library functions. All the mentioned languages are functional and come with compilers that are focused on generating high-performance code. All these languages have strong static type systems. Futhark and SaC are capable

of generating GPU code automatically. All these languages should be suitable to generate high-performance codes from CNN specifications such as the one we looked at in this paper. Tran et al. [41] describe how to compose deep learning algorithms in Futhark. They demonstrate excellent performance competitive with that obtained in Tensorflow as well. While this may seem to be a very similar contribution as the one made in this paper, there is a key difference. The paper focuses on the use of a Futhark library, not its construction. Furthermore, in Futhark, a shape-invariant formulation as the one presented in this paper is not possible. Instead, individual versions for fixed ranks have to be provided by the library. In case the provided ranks are insufficient, manual extensions are required as needed. As shown in Section 3, our simple network already involves dealing with arrays of rank five.

Finally, further relatives like MATLAB [40], Julia [4], Python [44] with NumPy [27] have some notion of multi-dimensional arrays and a subset of APL operators, both of which are embedded in the context of the general purpose language. All the mentioned languages come with interpreters only and rarely provide exceptional levels of performance other than by relying on the use of highly optimised external libraries. Yet these languages are very useful for prototyping.

5.2 Machine Learning DSLs

Machine Learning DSLs provide a way to express neural networks using high-level specifications. Typically, the high-level specification is either handled by a machine learning framework, or transformed into machine code for performance reasons.

TypedFlow [2] is embedded in Haskell and provides a number of dependently-typed primitives that can be used to define a network. Later this specification is translated into TensorFlow calls. This approach provides type safety and a powerful syntax, but performance-wise it still relies on the underlying framework. The tensorflow-ocaml [23] and ocaml-torch [24] are similar wrappers for TensorFlow and PyTorch in OCaml.

DEFINE [8] mainly focuses on liberating data scientists from the necessity to deal with general-purpose languages, such as Python, when describing the networks. The proposed syntax focuses exclusively on the machine learning primitives, and the accompanying tools take care of performance and portability, still using state of the art machine learning frameworks as a backend.

DeepDSL [53], OptiML [38] and Latte [43] focus on optimisations that are specific to machine learning such as kernel-fusion and parallelisation. They generate code to C++ and CUDA, using highly-optimised libraries.

The XLA [11] is a domain-specific compiler that focuses on accelerating linear algebra operations in machine-learning applications. The compiler is a part of the TensorFlow framework, and it works by analysing dataflow graph of the network and turning it into fast machine code by fusing pipelined nodes, inferring tensor shapes and performing memory optimisations based on these data sizes.

Tensor Comprehensions [45] has a very similar idea: it is a DSL that is integrated into existing machine learning frameworks and it provides a common ground to implement machine learning operators for further cross-optimisation. A distinctive feature for this approach is the use of the polyhedral model to perform the actual fusion, blocking, non-trivial scheduling and parallelisation. In a way the approach is very similar to Halide [30], except the domain is different and the number of optimisations is larger.

6 Conclusions

This paper makes an argument for an alternative design of machine learning frameworks. Instead of using a large number of interconnected specialised libraries, we consider using a compiled array-oriented language to host both the framework and the specification of the actual networks. To justify the viability of the proposed approach, we implement a minimalistic framework in SaC without building on any pre-existing libraries and use it to define a state of the art CNN. We compare its performance and expressiveness against implementations in TensorFlow and PyTorch.

Our solution is concise: about 150 lines of code to define the building blocks of the network, and another 150 lines to define the network itself — which is about the same amount as is needed for the network definition in both TensorFlow and PyTorch. The basic building blocks in SaC are rank-polymorphic functions that can be easily reused in other contexts. Rank polymorphism is a key to expressiveness here.

Our performance experiments on a 64-core machine with an NVIDIA K20 GPU show that the SaC implementation of the CNN is on par with that of TensorFlow and PyTorch. The TensorFlow and PyTorch implementations perform well on the GPU, but they did not make good use of the multi-core CPU. SaC performed poorly on the GPU, but the multi-threaded version outperforms both the CPU the GPU versions of TensorFlow and PyTorch by a factor between 2 and 3.

As for productivity, Python-based setups, with a large set of readily available libraries, are more elaborate than most existing array languages. The dynamic nature of the language gives rise to powerful abstractions, but poses challenges for getting good performance from native code. Performance is typically achieved by using special-purpose libraries implemented in low-level languages. When the number of such libraries grows, correct system setup turns into a daunting task, as we have experienced in our multiple runtime experiments with TensorFlow and PyTorch. By expressing frameworks and algorithms in the same array language, it

is possible to eliminate complex dependencies, reducing the setup time, and helping compilers to achieve competitive performance.

There seems to be no *conceptual* problem in bringing Python-level experience to an array language of choice. In the case of SaC, besides generating a proper machine learning library, two system improvements would be desirable: interactive behaviour and automatic differentiation. As a compiled language, SaC does not offer the same interactivity as Python. However, there exists a Jupyter-based frontend[5] that mimics REPL interactivity. Adding automatic differentiation to a compiler is a well-understood problem, as demonstrated by Stalin∇ [29]. Bringing it to the context of SaC is mainly an implementation task.

By no means do we suggest that existing machine learning frameworks can be readily replaced by array languages. However, a clean design that eliminates a number of abstraction layers, supported by the fact that a prototypical research compiler can significantly outperform two industrial frameworks suggests that the proposed approach is interesting enough to be further investigated.

Acknowledgments

We would like to thank our reviewers for their constructive suggestions. This work is supported by the Engineering and Physical Sciences Research Council through grants EP/N028201/1 and EP/L016834/1.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 265–283.

[2] Jean-Philippe Bernardy. 2017. TypedFlow. https://github.com/GU-CLASP/TypedFlow.

[3] Robert Bernecky. 1997. *An Overview of the APEX Compiler*. Technical Report 305/97. Department of Computer Science, University of Toronto.

[4] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. https://doi.org/10.1137/141000671

[5] David Cann and John Feo. 1990. SISAL Versus FORTRAN: A Comparison Using the Livermore Loops. In *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing* (New York, NY, USA) *(Supercomputing '90)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 626–636. https://doi.org/10.5555/110382.110593

[6] Wei-Ngan Chin and Siau-Cheng Khoo. 1993. Tupling functions with multiple recursion parameters. In *Static Analysis*, Patrick Cousot, Moreno Falaschi, Gilberto Filé, and Antoine Rauzy (Eds.). Springer, Berlin, Heidelberg, 124–140.

[7] Ronan Collobert, Koray Kavukcuoglu, Clément Farabet, et al. 2011. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS workshop*, Vol. 5. Granada, 10.

[8] Nina Dethlefs and Ken Hawick. 2017. DEFIne: A Fluent Interface DSL for Deep Learning Applications. In *Proceedings of the 2nd International Workshop on Real World Domain Specific Languages* (Austin, TX, USA) *(RWDSL17)*. ACM, New York, NY, USA, Article 3, 10 pages. https://doi.org/10.1145/3039895.3039898

[9] Carl F. Gauss. 1809. *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. sumtibus F. Perthes et I. H. Besser. https://books.google.co.uk/books?id=ORUOAAAAQAAJ

[10] Google. 2017. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html. [Accessed 2020/03].

[11] Google. 2020. https://www.tensorflow.org/xla. [Accessed 2020/02].

[12] Stuart Gordon and Sven-Bodo Scholz. 2015. Dynamic Adaptation of Functional Runtime Systems Through External Control. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages* (Koblenz, Germany) *(IFL '15)*. ACM, New York, NY, USA, Article 10, 13 pages. https://doi.org/10.1145/2897336.2897347

[13] Clemens Grelck. 2005. Shared Memory Multiprocessor Support for Functional Array Processing in Sac. *Journal of Functional Programming* 15, 3 (2005), 353–401. https://doi.org/10.1017/S0956796805005538

[14] Clemens Grelck, Karsten Hinckfuß, and Sven-Bodo Scholz. 2006. With-Loop Fusion for Data Locality and Parallelism. In *Implementation and Application of Functional Languages*, Andrew Butterfield, Clemens Grelck, and Frank Huch (Eds.). Springer, Berlin, Heidelberg, 178–195.

[15] Clemens Grelck and Sven-Bodo Scholz. 2003. Axis Control in Sac. In *Implementation of Functional Languages, 14th International Workshop (IFL'02), Madrid, Spain, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 2670)*, Ricardo Peña and Thomas Arts (Eds.). Springer, 182–198. https://doi.org/10.1.1.540.8938

[16] Clemens Grelck and Kai Trojahner. 2004. Implicit Memory Management for SaC. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL'04*, Clemens Grelck and Frank Huch (Eds.). University of Kiel, 335–348. Technical Report 0408.

[17] Troels Henriksen, Niels GW Serup, Martin Elsman, Fritz Henglein, and Cosmin E Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 556–571.

[18] Sakshi Indolia, Anil Kumar Goswami, S.P. Mishra, and Pooja Asopa. 2018. Conceptual Understanding of Convolutional Neural Network-A Deep Learning Approach. *Procedia Computer Science* 132 (2018), 679–688. https://doi.org/10.1016/j.procs.2018.05.069 International Conference on Computational Intelligence and Data Science.

[19] Intel. 2009. *Intel Math Kernel Library. Reference Manual*. Santa Clara, USA. ISBN 630813-054US.

[20] Kenneth E. Iverson. 1962. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA.

[21] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia* (Orlando, Florida, USA) *(MM '14)*. ACM, New York, NY, USA, 675–678. https://doi.org/10.1145/2647868.2654889

[22] A.M. Legendre. 1805. *Nouvelles méthodes pour la détermination des orbites des comètes*. F. Didot. https://books.google.co.uk/books?id=FRcOAAAAQAAJ

[23] Laurent Mazare. 2019. tensorflow-ocaml. https://github.com/LaurentMazare/tensorflow-ocaml.

[24] Laurent Mazare. 2020. ocaml-torch. https://github.com/LaurentMazare/ocaml-torch.

[5] Available at https://github.com/SacBase/sac-jupyter

[25] C. D. McCrosky, J. J. Glasgow, and M. A. Jenkins. 1984. Nial: A Candidate Language for Fifth Generation Computer Systems. In *Proceedings of the 1984 Annual Conference of the ACM on The Fifth Generation Challenge (ACM '84)*. ACM, New York, NY, USA, 157–166. https://doi.org/10.1145/800171.809618

[26] L.M.R. Mullin and M.A. Jenkins. 1996. Effective data parallel computation using the Psi calculus. *Concurrency: Practice and Experience* 8, 7 (1996), 499–515.

[27] Travis Oliphant. 2006. NumPy: A guide to NumPy. http://www.numpy.org. [Accessed 2020/02].

[28] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Workshop*. https://openreview.net/forum?id=BJJsrmfCZ

[29] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a Functional Framework: Lambda the Ultimate Backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 7 (March 2008), 36 pages. https://doi.org/10.1145/1330017.1330018

[30] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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* (Seattle, Washington, USA) *(PLDI '13)*. ACM, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[31] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[32] Sven-Bodo Scholz. 1998. With-loop-folding in Sac — Condensing Consecutive Array Operations. In *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, UK, Selected Papers (Lecture Notes in Computer Science, Vol. 1467)*, Chris Clack, Tony Davie, and Kevin Hammond (Eds.). Springer, 72–92. https://doi.org/10.1007/BFb0055425

[33] Sven-Bodo Scholz. 2003. Single Assignment C: Efficient Support for High-level Array Operations in a Functional Setting. *J. Funct. Program.* 13, 6 (Nov. 2003), 1005–1059. https://doi.org/10.1017/S0956796802004458

[34] Sven-Bodo Scholz and Artjoms Šinkarovs. 2021. Tensor Comprehensions in SaC. In *Proceedings of the 31st Symposium on the Implementation and Application of Functional Programming Languages* (Singapore) *(IFL 2019)*. ACM, New York, NY, USA. to appear.

[35] Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 27–46.

[36] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*. ACM, 74–85.

[37] Roger Stokes. 15 June 2015. Learning J. An Introduction to the J Programming Language. http://www.jsoftware.com/help/learning/contents.htm. [Accessed 2020/02].

[38] 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 domainspecific language for machine learning. In *in Proceedings of the 28th International Conference on Machine Learning, ser. ICML*.

[39] SaC Development Team. 2021. Supplementary materials for the ARRAY'21 workshop. https://github.com/SacBase/array-2021-supmaterial.

[40] The Mathworks, Inc., Natick, MA. 1992. *MATLAB Reference Guide.*

[41] Duc Minh Tran, Troels Henriksen, and Martin Elsman. 2019. Compositional Deep Learning in Futhark. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Functional High-Performance and Numerical Computing* (Berlin, Germany) *(FHPNC 2019)*. ACM, New York, NY, USA, 47–59. https://doi.org/10.1145/3331553.3342617

[42] Kai Trojahner and Clemens Grelck. 2009. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming* 78, 7 (2009), 643 – 664. https://doi.org/10.1016/j.jlap.2009.03.002 The 19th Nordic Workshop on Programming Theory (NWPT 2007).

[43] Leonard Truong, Rajkishore Barik, Ehsan Totoni, Hai Liu, Chick Markley, Armando Fox, and Tatiana Shpeisman. 2016. Latte: A Language, Compiler, and Runtime for Elegant and Efficient Deep Neural Networks. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. ACM, New York, NY, USA, 209–223.

[44] G. van Rossum. 1995. *Python tutorial.* Technical Report CS-R9526. Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

[45] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730

[46] Hans-Nikolai Vießmann, Artjoms Šinkarovs, and Sven-Bodo Scholz. 2018. Extended Memory Reuse: An Optimisation for Reducing Memory Allocations. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages* (Lowell, MA, USA) *(IFL 2018)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3310232.3310242

[47] Artjoms Šinkarovs, Robert Bernecky, and Sven-Bodo Scholz. 2019. Convolutional Neural Networks in APL. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Phoenix, AZ, USA) *(ARRAY 2019)*. ACM, New York, NY, USA, 69–79. https://doi.org/10.1145/3315454.3329960

[48] Yu Emma Wang, Carole-Jean Wu, Xiaodong Wang, Kim Hazelwood, and David Brooks. 2020. Exploiting Parallelism Opportunities with Deep Learning Frameworks. arXiv:1908.04705 [cs.LG]

[49] Arthur Whitney. 2001. K. http://archive.vector.org.uk/art10010830.

[50] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Zhiheng Huang, Brian Guenter, Huaming Wang, Jasha Droppo, Geoffrey Zweig, Chris Rossbach, Jie Gao, Andreas Stolcke, Jon Currey, Malcolm Slaney, Guoguo Chen, Amit Agarwal, Chris Basoglu, Marko Padmilac, Alexey Kamenev, Vladimir Ivanov, Scott Cypher, Hari Parthasarathi, Bhaskar Mitra, Baolin Peng, and Xuedong Huang. 2014. *An Introduction to Computational Networks and the Computational Network Toolkit.* Technical Report MSR-TR-2014-112. https://www.microsoft.com/en-us/research/publication/an-introduction-to-computational-networks-and-the-computational-network-toolkit/

[51] Tong Zhang. 2004. Solving Large Scale Linear Prediction Problems Using Stochastic Gradient Descent Algorithms. In *Proceedings of the 21st International Conference on Machine Learning* (Banff, Alberta, Canada) *(ICML '04)*. ACM, New York, NY, USA, 116. https://doi.org/10.1145/1015330.1015332

[52] Zhifei Zhang. 2016. *Derivation of Backpropagation in Convolutional Neural Network.* Technical Report. University of Tennessee, Knoxvill, TN. http://web.eecs.utk.edu/~zzhang61/docs/reports/2016.10%20-%20Derivation%20of%20Backpropagation%20in%20Convolutional%20Neural%20Network%20(CNN).pdf

[53] Tian Zhao and Xiaobing Huang. 2018. Design and implementation of DeepDSL: A DSL for deep learning. *Computer Languages, Systems & Structures* 54 (2018), 39–70. https://doi.org/10.1016/j.cl.2018.04.004