HERO-ML: A Very High-Level Array Language for Executable Modelling of Data Parallel Algorithms

Björn Lisper
bjorn.lisper@mdu.se
Mälardalen University
Sweden

Linus Källberg
linus.kallberg@mdu.se
Mälardalen University
Sweden

Abstract

HERO-ML is an array language, on very high level, which is intended for specifying data parallel algorithms in a concise and platform-independent way where all the inherent data parallelism is easy to identify. The goal is to support the software development for heterogeneous systems with different kinds of parallel numerical accelerators, where programs tend to be very platform-specific and difficult to develop. In this paper we describe HERO-ML, and a proof-of-concept implementation.

CCS Concepts: • **Computing methodologies** → **Parallel programming languages**; Neural networks; • **Software and its engineering** → **Very high level languages**; **Software prototyping**.

Keywords: data parallelism, array language

ACM Reference Format:
Björn Lisper and Linus Källberg. 2023. HERO-ML: A Very High-Level Array Language for Executable Modelling of Data Parallel Algorithms. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming (ARRAY '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3589246.3595370

1 Introduction

There is an ever-growing need for computational power. An area where the need for heavy computing is increasing rapidly is embedded systems, where applications like autonomous vehicles require massive amounts of computing for tasks like machine learning, advanced signal and image processing, etc. These systems also often have strong constraints on energy consumption, memory, unit cost, etc. The response from the hardware industry has been to develop increasingly integrated, heterogeneous hardware, where *computational accelerators* are placed on the chip or board to offload computationally heavy tasks from the main processor. In this way, large computational resources can be provided at low cost.

Today we see a large proliferation of accelerator architectures: GPGPU's, many-cores, solutions involving FPGA, and even ASICs. Although these architectures are quite different, they have in common that they typically rely on massive data parallelism to obtain performance. Besides embedded systems, these kinds of accelerators are also increasingly being used in traditional HPC as well as cloud computing: an example of the latter is Microsoft's Catapult project that integrates FPGAs into servers [10].

Developing software for these heterogeneous systems provides a challenge. Utilising the accelerators well requires parallel code, but parallel programming can be very hard and error-prone. The situation is aggravated by the fact that current programming practices for the accelerators are very dependent on the type of accelerator. For instance, code for a GPGPU will typically be very different from code for a many-core. This makes the code less portable, and costly redesigns may be needed if the hardware platform is changed.

A possible way forward is to consider *model-based development*, where system and software is specified by high-level models. The models can capture different aspects such as system structure, or program logic. If the model is *executable* then it can be used to simulate the aspect of the system that it captures: such models can be used to find errors in the design early, and they can also be used as test oracles in the validation phase.

HERO-ML[1] is a data-parallel executable modelling language, intended to be used for very high level specifications of data parallel algorithms. Such executable specifications can serve as portable "blueprints" when developing accelerator code, and they can help finding flaws in the algorithms at an early stage of development. HERO-ML is inspired by data-parallel and array languages such as *lisp [20], NESL [2], ZPL [6], and HPF [16]. These languages all implement a parallel model of computation where the parallelism resides in collective operations over data structures, like arrays, rather

ARRAY '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0169-6/23/06.
https://doi.org/10.1145/3589246.3595370

[1] ML here stands for "Modelling Language", not "Meta-Language", or "Machine Learning".

than explicitly in threads or processes. This model of computation is conceptually much simpler than the control parallelism given by parallel threads and processes, and languages like the ones mentioned above introduce various high-level concepts and constructs that help writing clear and concise data parallel code. HERO-ML aims to generalise and unify these concepts. Since HERO-ML is intended for high-level modelling rather than high performance production code, its design does not have to make compromises in order to allow for efficient implementations. Thus, its design rather focuses on providing maximal support for the early modelling phase in the design of software for parallel accelerators.

In this paper we give an overview of HERO-ML, and its salient features. A full language specification, including an abstract syntax, and formal semantics, is found in [17]. HERO-ML documentation, as well as a proof-of-concept implementation, is found at

`https://hero-ml-language.github.io/`.

2 HERO-ML in a Nutshell

HERO-ML has two parts: a sequential part, which is a simple imperative language, and a data parallel part in the form of an abstract array data type where operations on the arrays can be done in parallel over the array elements. Since the focus is on the parallel part, the sequential part is deliberately kept simple: it is a standard WHILE language [18], with assignments, conditionals, simple i/o, and While loops. It is strongly and explicitly typed, with basic types *int*, $float$, *bool*. HERO-ML also has some builtin functions: these have function types similar to what is found in some higher-order functional languages.

3 Abstract Arrays, and Bounds

The arrays in HERO-ML are called *abstract arrays* (similar to the previously considered *data fields* in Data Field Haskell [13]). Their design is based on the observation that arrays really correspond to functions from some index set to the set of values for the array elements. HERO-ML extends the standard WHILE language with the following array constructs: a set of *array expressions*, a statement to *evaluate* such expressions and bind the resulting array to a program variable, and a *masked concurrent assignment* where all elements in an existing array that fulfil some condition are updated in parallel.

Conventional arrays correspond to functions from intervals, or products of intervals. Abstract arrays can map from a larger variety of domains, including *sparse*, possibly multi-dimensional sets of indices, and even infinite sets. Every abstract array has a *bound*, which is a set expression safely over-approximating its domain. Arrays and bounds are both first class citizens in HERO-ML, and can be freely passed around.

In general an abstract array is a pair

$$(f, bnd)$$

where *bnd* is its bound, and f is a function defining the values of the array elements for the indices within the bound. For an *evaluated* array, f is given by a table containing the array elements. Evaluating an abstract array means to first compute its bound, and then creating the table and populating it with the corresponding array elements. For this to work, the bound must represent a finite set. Some HERO-ML bounds indeed represent infinite sets, and array expressions with such bounds cannot be evaluated. (Infinite bounds may seem useless, but the evaluation of abstract arrays has a lazy flavor where such bounds can make sense.)

HERO-ML supports the following kinds of bounds:

- *dense bounds*, representing intervals,
- *sparse bounds*, representing general finite sets,
- *predicate bounds*, representing (possibly infinite) sets defined by a predicate,
- *empty* and *all*, representing the empty and universal set, respectively, and
- *product bounds*, representing cartesian products.

Some two-dimensional bounds are illustrated in Fig. 1.

The HERO-ML bounds form a *complete lattice* [18], which means that they have certain mathematical properties. Indeed there is a strong relation to static program analysis, and the computation of bounds for forall expressions (see below) can be seen as a kind of run-time value analysis.

There are three major kinds of array expressions in HERO-ML:

- *explicit array expressions*, which define arrays through listing their elements,
- *array comprehensions*, which define arrays with explicit bounds where the elements are computed according to some rule, and
- *forall expressions*, which are similar to array comprehensions but have their bounds defined implicitly from the syntax of the expression. This construct is inspired by lambda abstraction in the lambda calculus, and the rules for computing bounds are designed to (over)approximate the domain of the partial function defined by the forall expression. Forall expressions are further described in Section 8.

In addition there is a masked concurrent assignment of arrays, where array elements that fulfil some condition are concurrently updated. Together, these array constructs can express basically all data parallel constructs found in the literature.

An array access out of bounds generates a run-time error. Also array elements within bounds may be undefined, but in this case a "soft" error value, denoted "?", is returned when accessing the element. This can happen since bounds

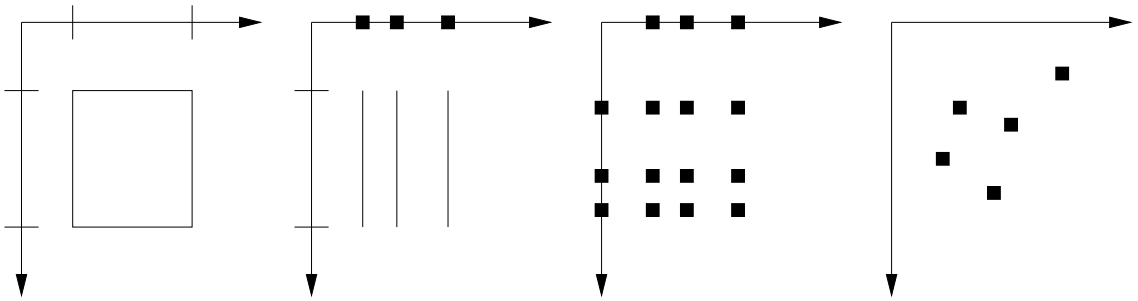

Figure 1. A dense product bound, a product of a dense and a sparse bound, a product of two sparse bounds, and a 2D sparse bound. (Adapted from [13].)

may be over-approximated: in particular, the rules for deriving implicit bounds for forall expressions may produce such bounds. This has some consequences for collective operations such as *reduce* and *scan*, which will have to skip such elements. "?" is not accessible at source code level, but HERO-ML has a predicate $isDef$ that tests if its argument is different from "?".

4 Types

HERO-ML is a *strongly*, and *explicitly typed* language. The types are similar to those found in ML-like functional languages, such as OCaml or Haskell, but with some restrictions. It has basic types int, $float$, $bool$, a polymorphic type $Array\ \iota\ \alpha$ for abstract arrays with indices of type ι and elements of type α, and a type $Bound\ \iota$ for bounds over indices of type ι. α can itself be an array data type: thus, HERO-ML supports nested arrays. Array indices are either integers (for one-dimensional arrays), or tuples of integers (for multi-dimensional arrays). There is, however, no full-fledged data type for tuples: they appear only as indices to arrays, within expressions defining bounds, or as arguments to builtin functions and operators, see below.

HERO-ML is not a full-fledged higher-order language. However, builtin functions and operators have function types. Contrary to what is customary in higher-order functional languages these functions are on uncurried form, taking a tuple of individual arguments as argument. For instance the operator "+" has the type $(int, int) \rightarrow int$, indicating that it takes two arguments of type int and returns an int. Arithmetic operators are overloaded over the numerical types int, and $float$: thus, for example, "+" also has the type $(float, float) \rightarrow float$. Implicit type conversion is not permitted: thus, numerical arguments of different type are not allowed for these operators. HERO-ML also has collective array operations *reduce*, and *scan*, which are higher-order in that they take a binary function, or operator, as an argument.

5 Standard Functions, and Syntax

HERO-ML comes with a standard set of numerical and boolean functions, and operators. There is also a conditional function. The HERO-ML syntax is also very much standard, with standard syntax for identifiers, standard precedence and associativity rules for operators, etc. Like in Python and F#, the syntax is indentation-sensitive.

6 Functions and Operations on Bounds

HERO-ML has many functions on bounds. Many of these are used to implement the automatic derivation of implicit bounds that takes place when array-valued expressions, such as $forall$-expressions, are evaluated. They are exposed at the user level primarily to allow the manual tailoring of bounds in cases when the implicit bound derivation does not yield a satisfactory result. The functions can also be seen as providing an *interface* for bounds: any set representation that implements these functions (such that they have certain prescribed properties) can in principle be used as bounds.

Below i stands for integer variables, ie for integer expressions, ti for tuples of integer variables, or single variables, te for tuples of integer expressions, e for general HERO-ML expressions, bnd for bounds, b for boolean expressions, a for array variables, and ae for array expressions. Array expressions also include expressions to access subarrays in nested arrays, and similarly array variables may be indexed to access subarrays of nested arrays: we omit these cases below for notational simplicity.

6.1 Functions That Operate on Bounds

$member : (\iota, Bound\ \iota) \rightarrow bool$. Membership in the set defined by a bound.

$join, meet : (Bound\ \iota, Bound\ \iota) \rightarrow Bound\ \iota$. Lattice-theoretical $\sqcup$ and $\sqcap$, respectively, in the lattice of bounds over $Bound\ \iota$ (approximating set union and -intersection). Fig. 1 shows the "type" of bound for join and meet for the different

Table 1. Result "types" of *join* (⊔), and *meet* (⊓) as a function of the argument "types" (adapted from [13]). $E = empty$, $A = all$, $S = sparse$, $D = dense$, $P = predicate$, $\times$ = product bound. "S/P" in the table for *join* means that the result is *sparse* if the product bound is finite, and a *predicate* otherwise, and "–" means that the combination is not allowed.

⊔	E	A	S	D	P	$\times$
E	E	A	S	D	P	$\times$
A		A	A	A	A	A
S			S	D	P	S/P
D				D	P	–
P					P	P
$\times$						$\times$

⊓	E	A	S	D	P	$\times$
E	E	E	E	E	E	E
A		A	S	D	P	$\times$
S			S	S	S	S
D				D	S	–
P					P	S/P
$\times$						$\times$

possible combinations of bounds. Exact definitions of join and meet are found in [17].

Example (dense 1-D bounds): `meet(1..10,5..30)` = `5..10`, `join(1..10,20..30)` = `1..30`.

$isDense, isSparse, isPredicate, isProduct : (Bound\ \iota) \rightarrow bool$. Tests for the different kinds of bounds.

$finite : (Bound\ \iota) \rightarrow bool$: $finite(bnd)$ is true iff *bnd* is classified as finite (*i.e.*, if *bnd* is dense, sparse, *empty*, or a product of finite bounds).

$size : (Bound\ \iota) \rightarrow int$: returns the number of elements in a finite bound. Undefined for infinite bounds.

$enum : (Bound\ \iota) \rightarrow Array\ int\ \iota$: provides an enumeration of the elements in a finite bound, in lexicographic order, in the form of an array containing the elements in this order. Undefined for infinite bounds.

Example (*enum* of a sparse 1-D bound): `enum({3,1,7})` = `[0..2 : 1,3,7]`.

Note: *enum* is currently not visible in HERO-ML. It is however visible (and used) in intermediate layers.

6.2 Constructs That Define Bounds

empty, *all*: bottom and top in the lattice of bounds, corresponding to the empty and universal set, respectively.

$ie..ie'$: an interval bound (also called *dense bound*).

Example: `1..10`

$\{te_1, \ldots, te_m\}$: a finite, possibly irregular set, called a *sparse bound*.

Example: `{(0,1),(3,2),(0,-1),(2,2)}`, a sparse two-dimensional bound with four elements.

$\{ti : b\}$: a so-called *predicate bound*, a possibly infinite set defined by a predicate.

Example: `{(i,j) : i+j > 0}`, the set of all `(i,j)` such that `i+j > 0`.

$(bnd_1, \ldots, bnd_n)$: *cartesian product bound* formed from n 1-dimensional bounds.

Example: `(1..10,1..25)`, the 2-D bound for a 10×25 dense matrix, formed from two 1-D bounds.

7 Functions and Operations on Abstract Arrays

7.1 Functions That Work on Abstract Arrays

$bound : (Array\ \iota\ \alpha) \rightarrow Bound\ \iota.\ bound(ae)$ extracts the bound from the array expression *ae*.

Example: `bound([2..4 : 1,3,2])` = `2..4`.

$reduce : ((\alpha, \alpha) \rightarrow \alpha, Array\ \iota\ \alpha) \rightarrow \alpha.\ reduce(f, a)$ "sums" the elements in the array expression *a* using the binary function/operator f.

Example: `reduce(+,[2..4 : 1,3,2])` = `1 + 3 + 2` = `6`.

$scan : ((\alpha, \alpha) \rightarrow \alpha, Array\ \iota\ \alpha) \rightarrow Array\ \iota\ \alpha$. Like *reduce*, but computes an array with all the "partial sums". The bound for $scan(f, a)$ is the same as for *a*, and the "partial sums" are computed in the lexicographic order over $bound(a)$.

Example: `scan(+,[2..4 : 1,3,2])` = `[2..4 : 1,4,6]`

7.2 Constructs That Define Abstract Arrays

$[e : ti\ in\ bnd]$: array comprehension (definition of n-dimensional array with explicit bound).

Example: `[2*i : i in 1..10]`, an array with bound `1..10`, and elements `2*i`.

$forall\ ti \rightarrow e$: array definition with implicit bounds, and syntax similar to lambda-abstraction. The bound is computed from the syntactic structure of *e*. Rules for how to compute the bounds are given in [17].

Example: `forall i -> a[i] + b[i]`, element-wise addition of `a` and `b`, computing the bound from those of `a` and `b`.

ae | *bnd*: subarray of *ae* defined by *bnd*.

Example: `a | 1..10`, the subarray of `a` from `1` to `10`.

$[preamble\ elist(n)]$: array with dense bounds, where the elements are explicitly listed. There are several variations that differ in how the bound is specified. "*preamble*" specifies the (possibly multi-dimensional) dense bounds, whereas "$elist(n)$" specifies the array elements.

Example 1: `[2.. : 1,3,2]`, a dense one-dimensional array with bound `2..4`, and three elements `1`, `3`, `2`.

Example 2: `[..4 : 1,3,2]`, same array as above, but with the bound specified through its upper limit.

Example 3: `[1,3,2]`, an array with the same elements as above, but with bound `0..2` (similar to a C array).

Example 4: `[(1..2,1..3) : 1,2,3; 4,5,6;]`, a 2D-array (matrix) with rows `1,2,3`, and `4,5,6`. (The last semi-colon can be dropped.)

Example 5: `[1,2,3; 4,5,6;]`, a 2D-array like in Example 4, but with default bounds `(0..1,0..2)`.

Example 6: `[(, , 98..100) : 1,2,3; 4,5,6;; 7,8,9; 10,11,12;; 13,14,15; 16,17,18;;]`, a 3D-array with bounds `(0..2,0..1,98..100)`. (The two first are default bounds.) If we name the array "`A`", then `A[1,0,99] = 8`.

$[te_1 : e_1, \ldots, te_n : e_n]$: explicit sparse array expression.

Example: `[(1,1):4.7, (2,3):0.01, (3,5):3.14]`, a sparse two-dimensional array with bound `{(1,1), (2,3), 3,5)}`, and three elements `4.7`, `0.01`, `3.14`.

7.3 Assignment of Abstract Arrays

$a = ae$: creates a new abstract array by evaluating *ae*, and sets *a* to hold it.

Example: `a = [2*i : i in 1..10]`, sets `a` to the new array defined by the array expression.

$foreach\ ti\ in\ bnd\ do\ a[te] = e$: destructive, masked, in-place update where $a[te]$ is set to *e* for all values of the tuple *ti* that belong to *bnd* and where *e* is defined.

Example: `foreach i in 1..10 do x[i+1] = y[i-1]`, sets `x[i+1]` to `y[i-1]` for all `i` in `1..10` where `y[i-1]` is defined.

Example (nested): `foreach i in 1..10 do x[i][i+1] = y[3][i-1]`, sets `x[i][i+1]` to `y[3][i-1]` for all `i` in `1..10` where `y[3][i-1]` is defined.

Note: it might be that the same element in the left-hand side is targeted by the right-hand side for more than one value of `i`. This implies a write conflict, and the value of the targeted array element is then non-deterministically set to one of the values written there. An error also occurs if, for some `i`, the target address for the left-hand side is out of bounds.

8 Forall-Expressions

The purpose of $forall$-expressions is to provide a convenient, and generic syntax for arrays, which is close to mathematical notation. It builds on the fact that arrays really are partial functions from indices to array values. In higher-order functional languages functions can be defined through λ-abstractions $\lambda i.e$, and the forall-expression $forall\ i \to e$ is similar except that it defines an array whose bound is automatically derived from the syntax of *e*. The programmer is thus relieved from the task of explicitly defining the bounds. The implicit bounds also help making array definitions more reusable across different kinds of bounds.

The guiding principle is that the bound for $forall\ i \to e$ should safely over-approximate the domain of $\lambda i.e$, that is: $dom(\lambda i.e) \subseteq bound(forall\ i \to e)$ where $dom(f) = \{ x \mid f(x) \neq ? \}$. The rationale is that if we view the function $\lambda i.e$ as the "ideal" array then $bound(forall\ i \to e)$ will surely contain all elements for which $\lambda i.e$ is defined. Functions over $\lambda i.e$ can then be computed over $forall\ i \to e$ by first filtering out the relevant appearances of "?".

The derivation of bounds is basically a simple value analysis by abstract interpretation [8, 18]. However, contrary to a traditional static program analysis the derivation of bounds is done at run-time, every time a $forall$-expression is evaluated.

The bound for $forall\ i \to e$ is given by a function $B(e, i, Y)$ where Y is a set of variables that are bound in enclosing expressions. On top level, $Y = \emptyset$. $B(e, i, Y)$ is then defined recursively, over the structure of *e*. A full definition is found in [17]. Here we exemplify by computing the bound for $forall\ i \to (A[i] + i)$:

$$\begin{aligned} & B(forall\ i \to (A[i] + i), i, \emptyset) \\ = \ & B(A[i] + i, i, \{i\}) \\ = \ & B(A[i], i, \{i\}) \sqcap B(i, i, \{i\}) \\ = \ & bound(A) \sqcap all \\ = \ & bound(A) \end{aligned}$$

The rationale for replacing + with $\sqcap$ is that + is "?-strict" in both arguments, that is: $? + x = x + ? = ?$. Thus $A[i] + i$ is defined only when both $A[i]$ and i are. This gives the evaluation of arrays a lazy flavour, since the initial evaluation of bounds may reduce the access of infinite arrays to finite pieces. The example above can be seen as element-wise addition of A with the infinite array $forall\ i \to i$, which yields a finite array whenever $bound(A)$ is finite even though the second argument is infinite.

Multi-dimensional arrays are extremely important in many computational applications. Different data parallel and array languages have therefore developed a multitude of constructs to express operations on matrices and arrays such as selection of row or column vectors from a matrix, reduction over all row/column vectors, and other operations. These constructs are however often a bit ad-hoc, often lacking generality, and the semantics can be somewhat unclear. `forall`-expressions provide a uniform, general, and semantically well-defined syntax to express various operations on higher-dimensional arrays. Below are some examples:

- `forall i -> A[i,c]` selection of column `c` from matrix `A`
- `forall j -> A[r,j]` selection of row `r` from `A`
- `forall i -> A[i,i]` selection of main diagonal from `A`
- `forall (i,j) -> A[j,i]` transpose of `A`

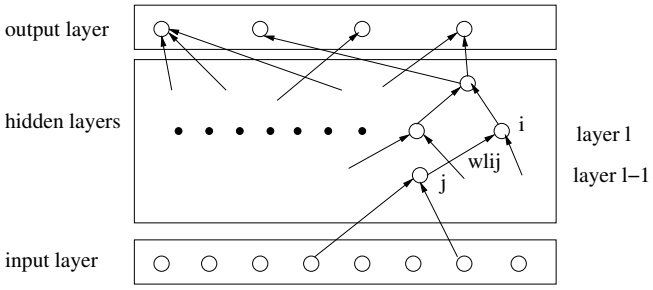

Figure 2. Layers in a feed-forward network.

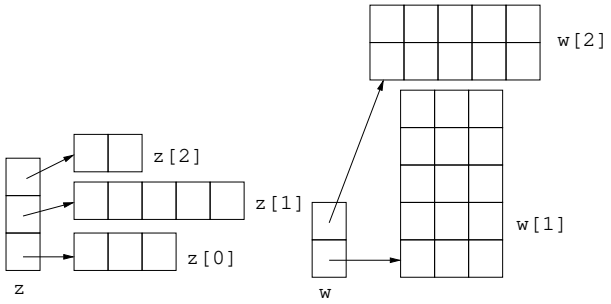

Figure 3. A nested array representation of layers in a feed-forward network.

- `forall i -> (forall j -> A[i,j])` conversion of matrix `A` into a nested array of arrays (row vectors)
- `forall i -> reduce(+,(forall j -> A[i,j]))` array of the sums of all rows in `A`

The current version of the B function derives exact bounds for all these examples. Notably this is independent of the kind of bound, so correct bounds are derived also when `A` is, for instance, a sparse matrix. However, in general there must always be cases where exact bounds cannot be automatically derived. In such cases HERO-ML provides fallback mechanisms such as subarrays, and array comprehensions, which can be used to set bounds manually.

9 A Worked Example: Feed-Forward ANN

9.1 Feed-Forward Networks

Feed-forward networks [9] is a classical example of Deep Neural Network (DNN). In general a DNN consists of many small, interconnected units, where outputs from units ("activity levels") are connected to inputs for other units. Some inputs provide the input to the network, and some outputs yield the corresponding response. The connections between units are set by *training* the DNN on a set of examples. Once trained, the DNN can be used for tasks like feature recognition.

In a feed-forward network the units are arranged in a number of *layers*, where each interconnection goes from some layer $l-1$ to layer l. See Fig. 2. There are three kinds of layers:

- an *input* layer, which provides the input to the ANN,
- a number of *hidden* layers, which contain units interconnected between layers, and
- an *output* layer, which provides the output (or response) from the ANN given a certain input.

The input is basically an array of numbers. It can, for instance, be a pixel matrix encoding a picture. The output is also an array of numbers that encodes the output. It can, for instance, represent a classification of some object in the picture.

Each unit in a hidden layer computes its output as a weighted, thresholded sum of the outputs of the connected units in the previous layers. Mathematically, output z_{lj} from unit j in layer l is computed as

$$z_{li} = s(\sum_j w_{lij} z_{l-1j}) \tag{1}$$

where the sum ranges over the units j in layer $l-1$ that are connected to unit i in layer l. w_{lij} is the *weight* of the connection from j in layer $l-1$ to i in layer l. s is commonly chosen as the *sigmoid function*, defined by

$$s(x) = \frac{1}{1+e^{-x}} \tag{2}$$

The interconnections between layers are typically *sparse*: only a few units in layer $l-1$ are connected to each unit in layer l.

9.2 Modelling of Feed-Forward Network Computing with HERO-ML

We will now show how to model one particular way of computing the output from a trained feed-forward network, given some input. We will use nested arrays, where the nesting reflects the structuring of the network into layers. More specifically `z` will be an array of arrays, where `z[l]` holds the output values of the units in layer `l`. The weights will be stored in an array of matrices `w`, where `w[l]` is a matrix where each element `w[l][i,j]` holds the weight for the connection from unit `j` in layer `l-1` to unit `i` in layer `l`. The types of `z` and `w` are as follows:

```
z : Array int (Array int float)
w : Array int (Array (int,int) float)
```

Note that `w[l]` might be a sparse matrix, with a sparse bound. We assume that the arrays `z` and `w` themselves have dense bounds `0..n-1` and `1..n-1`, respectively, where `n` is the number of layers. Fig. 3 shows an example with an input layer with three units, a hidden layer with five units, and an output layer with two units.

We now give HERO-ML code for the computation. We use the following declarations[2]:

[2]HERO-ML does not have function definitions, but we can see these declarations as macros.

```
s(x) = 1/(1 + exp (-x)) // sigmoid function
sum(a) = reduce(+,a)    // sum over abstract array
```

We assume that the input to the computation is stored in the array `input`. First, the input layer `z[0]` is assigned this array. Then the code loops over the other layers, computing `z[l]` from a matrix-array multiplication of `z[l-1]` and `w[l]` followed by a thresholding of the elements in the resulting array:

```
z[0] = input;
l = 1;
while l < n do
  z[l] =
    forall i ->
      s(sum(forall j ->
        (w[l][i,j] * z[l-1][j])));
  l = l + 1;
```

This version creates a new abstract array for each array assignment. As an alternative we can instead use a `foreach` statement, which performs an in-place update:

```
foreach i in bound(z[0]) do z[0][i] = input[i];
l = 1;
while l < n do
  foreach i in bound(z[l]) do
    z[l][i] =
      s(sum(forall j ->
        (w[l][i,j] * z[l-1][j])));
  l = l + 1;
```

The versions are very similar, and which one to choose is a matter of taste. Possibly the first version is somewhat more clear, whereas the second version avoids some dynamic memory handling due to the in-place updates of the arrays.

10 Formal Semantics

HERO-ML has been given an operational semantics [17]. The semantics is structured into two parts:

1. For statements, the semantics is given as rules for state transitions, and
2. for expressions, an *eval* function is defined that evaluates expressions relative to the current state.

The semantics for statements, excluding assignments involving abstract arrays, is quite standard. The rules for state transitions have the form $(s, S) \rightarrow S'$ where s is a statement, S is a function from program variables to values (a so-called *store*), expressing the current contents of the memory before executing the statement, and S' is a store expressing the memory contents after having executed s.

Computations can return the undefined value "?". The semantics has special rules covering the cases where boolean conditions that affect the program flow become undefined. In such a case the program goes into a particular error state, where it halts.

HERO-ML expressions are evaluated using the *eval* function. Usually this function takes two arguments: the expression to be evaluated, and a store that gives the current memory contents. Here *eval* also takes a third argument, which is a set of variables that are bound on some outer level but appear as free variables in the evaluated expression. This turns out to be needed since HERO-ML has three variable-binding constructs: predicate bounds, array comprehensions, and *forall* expressions. Within such expressions *eval* might return symbolic variables rather than values.

We exemplify with evaluation of the predicate bound $\{i : i < n\}$ in an environment where $S(n) = 10$:

$$\begin{aligned} & eval(\{i : i < n\}, S, \emptyset) \\ = \; & \{i : eval(i < n, S, \{i\})\} \\ = \; & \{i : eval(i, S, \{i\}) < eval(n, S, \{i\})\} \\ = \; & \{i : i < S(n)\} \\ = \; & \{i : i < 10\} \end{aligned}$$

This illustrates how symbolic expressions, like predicate bounds, can be evaluated dynamically.

For array expressions a the evaluation phase above is followed by a second phase. First the bound is computed. Then, if a is applied to an index viz. $a[i]$, it is first checked whether i is within the bound. If it is then $a[i]$ is computed without evaluating the full array a. Otherwise the full array is evaluated by first checking whether its bound is finite, and then tabulating the array for all elements defined by the bound.

11 Proof-of-Concept Implementation

A proof-of-concept HERO-ML interpreter has been implemented. The interpreter is a command line application written entirely in F#, and so should be supported on all major platforms without modifications to the source code. It can be freely downloaded from the HERO-ML web site[3].

The overall design of the interpreter is straightforward. It first parses a HERO-ML source file to generate an abstract syntax tree (AST). Then the AST undergoes a validation step where any remaining static correctness checks that could not practically be performed during parsing (such as type checking) are carried out. Finally, if the program passed the previous two steps, it is executed from start to finish using the AST as the program format.

Throughout the program execution, a representation of the current HERO-ML program state is maintained in memory as a dictionary data structure (implemented using the type *Map* from the standard library of F#), which keeps an entry for each program variable holding its current run-time value. Variables that are local to, e.g., forall expressions are only added to the dictionary temporarily while the expression in which they are bound is being evaluated. The actual

[3] https://hero-ml-language.github.io/

program execution is an iterative process analogous to applying the semantic rules described in Section 10 to the AST nodes and the program state in a repeated fashion, generating updated program states until the program terminates. HERO-ML provides a simple *out* statement for writing values: Such a statement, when executed, will print a textual representation of the specified output value to the console.

To represent the run-time values of variables as well as any intermediate results generated during expression evaluations, a custom type $Value$ is used, which is an F# discriminated union with one case for each HERO-ML type. The scalar types of HERO-ML are represented using the corresponding primitive data types of F#. Bounds and arrays are represented using custom aggregate types, described in more detail below. In addition the $Value$ union includes cases for certain symbolic values, such as the special value "?", as well as an "ERROR" value which represents a program error generated during expression evaluation. In most cases, occurrences of ERROR immediately cause execution to terminate with an error message printed to the console.

The run-time representation used for bounds is fairly simple, using an F# discriminated union for the different types of bounds. Some bounds require additional parameters for their definition: A dense bound $l..u$ is given by a pair of integers, and a product bound $(b_1, \ldots, b_m)$ stores the factors $b_1, \ldots, b_m$ in a list. For a sparse multi-dimensional bound (see [17] for details), its set S uses the Set datatype from the F# standard library. The individual indices of S are represented using a custom type $Index$, which is simply a type alias for an integer list. Predicate bounds are stored as an F# function of type $Index \rightarrow bool$ which, when applied to an index value, evaluates the predicate with the local index variable(s) of the predicate temporarily bound to the given index.

To represent abstract arrays, two pieces of data are combined: a bound, and a mechanism that looks up the array element corresponding to some given index. Depending on the kind of array expression this mechanism is either a function, of type $Index \rightarrow Value$, or a table lookup.

Accessing a single array element `a[i]` now proceeds as follows. First the index is checked for membership in the bound. If this check fails, then the operation immediately returns the symbolic ERROR value to signal an access out-of-bounds. Otherwise, the lookup mechanism of the array is used to retrieve the element.

Accessing a whole array `a` is a bit different. First a check is done whether the bound is finite: if not, ERROR is returned. If the test succeeds then the array is tabulated by evaluating and storing the elements `a[i]` for all `i` in `bound(a)`.

The data structure used for storing a tabulated array depends on the kind of bound. In all cases the elements are stored in a one-dimensional array. For dense arrays there is a 1-1 mapping between the HERO-ML array indices and the low-level indices in the array holding the elements: accessing an element will take $\Theta(n)$ time, where n is the dimension of the HERO-ML array. For sparse bounds the one-dimensional array is augmented with a "cluster dictionary", which is space-efficient and still allows access in $\Theta(n) + O(\log m)$ time where m is the number of array elements. For details, see [17].

12 Related Work

Array languages have a long history. Already APL [15] had many of the array primitives that are found in later array languages. Later, Fortran dialects such as Fortran 90 [14], and HPF [16] were equipped with array primitives. Of these, the masked array assignment of HPF has inspired the $foreach$ masked concurrent assignment of HERO-ML. Also *lisp [20], a production language for the massively parallel CM-200 SIMD machine, could execute most of its array primitives conditionally relative to a bit mask selecting which array elements that should participate.

ZPL [6] is another source of inspiration for HERO-ML. In ZPL bounds (called regions) are first-class citizens. The recent parallel language Chapel [5] has inherited from ZPL.

HERO-ML's nested arrays provide the ability to express nested data parallelism. Nested data parallelism first appeared in NESL [2]. A dialect of Haskell that supports nested data parallelism is Nepal [3].

There have been several attempts to integrate functional and array programming. One way of doing it is like in [11], where lambda-calculus is used as an abstract specification language for array algorithms. A more concrete approach is to extend some existing functional language with array primitives. This can yield domain-specific languages such as Obsidian [19], Feldspar [1], and Futhark [12], or general purpose languages like Data Parallel Haskell [4] and Data Field Haskell [13]. The latter is a direct predecessor to HERO-ML, with a very similar array concept but in a purely functional setting.

Array languages are typically used for data-parallel programming. A systematic literature review on parallel languages [7] gives quantitative evidence that data parallelism is a major approach to problem decomposition in parallel computing. This indicates that the topic of array languages is well worth further study in the future.

13 Conclusions and Further Research

We have described HERO-ML, a very high level array language that supports a rich set of array operations, over a variety of dense, sparse, nested, and multidimensional arrays, in a unified framework. A major goal is to support the early modelling of data parallel algorithms where the HERO-ML code can serve as a platform-independent, executable specification. The proof-of-concept implementation can also serve as a workbench for carrying out array language experiments.

HERO-ML is an experimental language. Thus, some features are currently missing. An obvious omission is user-defined functions, which should be present in a production version of the language. A richer set of numerical types would facilitate, e.g., experiments with limited precision arithmetic for deep neural networks. There is also much "syntactic sugar" that could be added. An example is "elemental intrinsics overloading", where scalar operators are lifted to operate on arrays. Another example is syntax for selecting subarrays, e.g., writing `a[k,*]` for the `k`'th row of the matrix `a`. These kinds of syntactic conveniences are straightforward to resolve into forall expressions.

Acknowledgments

This research was funded by the KK-foundation, through the HERO project, under grant no. 20180039.

References

[1] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajdax. 2010. Feldspar: A domain specific language for digital signal processing algorithms. In *Proc. Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*. 169–178. https://doi.org/10.1109/MEMCOD.2010.5558637

[2] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. 1994. Implementation of a Portable Nested Data-Parallel Language. *J. Parallel Distrib. Comput.* 21, 1 (April 1994), 4–14.

[3] Manuel M. T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. 2001. Nepal — Nested Data Parallelism in Haskell. In *Proc. Euro-Par 2001 Parallel Processing*, Rizos Sakellariou, John Gurd, Len Freeman, and John Keane (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 524–534.

[4] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. 2007. Data Parallel Haskell: A Status Report. In *Proc. 2007 Workshop on Declarative Aspects of Multicore Programming* (Nice, France) *(DAMP '07)*. ACM, New York, NY, USA, 10–18. https://doi.org/10.1145/1248648.1248652

[5] Bradford L. Chamberlain. 2015. Chapel. In *Programming Models for Parallel Computing*, Pavan Balaji (Ed.). MIT Press, Cambridge, MA, Chapter 6, 129–159.

[6] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. 1998. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering* 5, 3 (1998), 76–86.

[7] Federico Ciccozzi, Lorenzo Addazi, Sara Abbaspour, Björn Lisper, Abu Naser Masud, and Saad Mubeen. 2022. A Comprehensive Exploration of Languages for Parallel Computing. *Comput. Surveys* 55, 2 (Jan. 2022), 1–39. http://www.es.mdu.se/publications/6282-

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages*. Los Angeles, 238–252.

[9] Terrence L. Fine. 1999. *Feedforward Neural Network Methodology*. Springer-Verlag, New York.

[10] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steve Reinhardt, Adrian Caulfield, Eric Chung, and Doug Burger. 2018. A Configurable Cloud-Scale DNN Processor for Real-Time AI. In *Proc. 45th International Symposium on Computer Architecture, 2018* (proc. 45th international symposium on computer architecture, 2018 ed.). ACM.

[11] Per Hammarlund and Björn Lisper. 1993. On the Relation between Functional and Data Parallel Programming Languages. In *Proc. Sixth Conference on Functional Programming Languages and Computer Architecture*. ACM Press, 210–222.

[12] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In *Proc. 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

[13] Jonas Holmerin and Björn Lisper. 2000. Data Field Haskell. In *Proc. Fourth Haskell Workshop*, Graham Hutton (Ed.). Montreal, Canada, 106–117.

[14] International Organization for Standardization and International Electrotechnical Commission 1991. *Fortran 90 [ISO/IEC 1539: 1991 (E)]*. International Organization for Standardization and International Electrotechnical Commission.

[15] K. E. Iverson. 1962. *A Programming Language*. Wiley, London.

[16] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele, Jr., and Mary E. Zosel. 1994. *The High Performance Fortran Handbook*. MIT Press, Cambridge, MA.

[17] Björn Lisper and Linus Källberg. 2023. *HERO-ML Specification*. Technical Report. http://www.es.mdh.se/publications/6649-

[18] Flemming Nielson, Hanne Ries Nielson, and Chris Hankin. 2005. *Principles of Program Analysis, 2nd edition*. Springer.

[19] Joel Svensson, Koen Claessen, and Mary Sheeran. 2010. GPGPU kernel implementation and refinement using Obsidian. *Procedia Computer Science* 1, 1 (2010), 2065–2074. https://doi.org/10.1016/j.procs.2010.04.231

[20] Thinking Machines Corporation 1991. *Getting Started in *Lisp*.* Thinking Machines Corporation, Cambridge, MA.

Received 2023-03-31; accepted 2023-04-21