Padding in the Mathematics of Arrays

Benjamin Chetioui
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
benjamin.chetioui@uib.no

Ole Abusdal
Western Norway University of Applied Sciences
Department of Computer science, Electrical engineering and Mathematical sciences
Bergen, Hordaland, Norway
ojab@hvl.no

Magne Haveraaen
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
https://www.ii.uib.no/~magne/

Jaakko Järvi
University of Turku
Department of Computing
Turku, Finland
jaakko.jarvi@utu.fi

Lenore Mullin
College of Engineering and Applied Sciences
University at Albany, SUNY
Albany, NY, USA
lmullin@albany.edu

Abstract

Multi-dimensional array manipulation constitutes a core component of numerous numerical methods, e.g. finite difference solvers of Partial Differential Equations (PDEs). The efficiency of such computations is tightly connected to traversing array data in a hardware-friendly way.

The Mathematics of Arrays (MoA) allows reasoning about array computations at a high level and enables systematic transformations of array-based programs. We have previously shown that stencil computations reduce to MoA's Denotational Normal Form (DNF).

Here we bring to light MoA's Operational Normal Forms (ONFs) that allow for adapting array computations to hardware characteristics. ONF transformations start from the DNF. Alongside the ONF transformations, we extend MoA with rewriting rules for padding. These new rules allow both a simplification of array indexing and a systematic approach to introducing halos to PDE solvers. Experiments on various architectures confirm the flexibility of the approach.

CCS Concepts: • **Software and its engineering**;

Keywords: Mathematics of Arrays, Finite Difference Methods, PDE Solvers, Ghost Cells, High-Performance Computing, Stencil Computations

ACM Reference Format:
Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. 2021. Padding in the Mathematics of Arrays. 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/10.1145/3460944.3464311

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

ACM ISBN 978-1-4503-8466-7/21/06…$15.00
https://doi.org/10.1145/3460944.3464311

1 Introduction

In the past few decades, a large variety of high-performance computing (HPC) architectures has appeared. On the path towards exascale computing, we can expect to see a similar medley of architectures. Software for HPC therefore needs to be highly adaptable. This includes adjusting to, among other things, different memory hierarchies and changing intra- and interprocess communication hardware.

This paper explores the Mathematics of Arrays (MoA) formalism [10] as a tool for optimizing array codes for different hardware architectures. We have previously established a means of transforming stencil-based array code to *Denotational Normal Form* (DNF) [4]—irreducible expressions in the language of MoA. Given knowledge of the targeted parallel distribution and memory layout, one can transform a DNF expression to an architecture-specific normal form, the *Operational Normal Form* (ONF), which we describe in Section 4.

ONF transformations include the *dimension lifting* operation for reshaping an array by splitting a given axis of its shape into two or more dimensions. This operation can conveniently divide an array over different computation loci (whether they be threads, cores, or even systems).

The contribution of this paper is a formalization of concepts of MoA's ONF and the extension of MoA with new

operations to deal with padding of data. With these operations, MoA provides a framework for transforming regular array stencil code to distributed code with *halo zones* — also referred to as *ghost cells* in the literature [8]. As an example, the paper shows how MoA and its ONF help in the search for more efficient stencil-based array computations in a Partial Differential Equation (PDE) solver based on Finite Difference Methods (FDMs). We obtain a 10% performance improvement with changes easily expressible in MoA.

We have started to implement MoA with our extensions in Coq, so that the formal claims we make, e.g., about the ONF transformations can be machine-checked. This effort is at an early stage; the proofs can be found in the repository at https://github.com/mathematics-of-arrays/moa-formalization.

The paper is organised as follows. Next is a motivation section, then a discussion of related work. Section 4 covers the required prerequisites in MoA and previous work on the DNF layer. Section 5 discusses dimension lifting, and defines and explains padding and data layout. We then briefly report on some experiments and conclude in Section 7.

2 Motivation

To motivate our work, we ran the PDE solver we presented in [4] on a set of experimental architectures and implemented some of the ONF transformations on the code. Table 1 shows a matrix where each column corresponds to a different version of the solver and each row to different hardware. The table makes it plain that different architectures benefit from different transformations. While on CPU 1 the *dimension lifting on 0^{th} axis and tiled memory* approach performs best, on CPU 3 it is clearly inefficient compared to the other dimension lifting-based scenarios.

The hard to predict variations in performance, and the sheer number of different memory layouts, motivate a vehicle for easy exploration of codes that implement different memory layouts. If exploring different layouts is made easy, programmers can obtain close-to-optimal performance for different architectures with little effort.

In the following, we demonstrate that MoA, with our extension of operations for padding, provides the required level of expressivity to accomplish just that.

3 Related

Ken Iverson introduced whole-array operations in the APL programming language [7]. Building on further explorations by Abrams [1], Mullin created the Mathematics of Arrays formalism [10] in order to address various shortcomings of the universal algebra underlying APL (most notably the lack of a calculus for indexing). MoA is intended to serve as a foundation for exploring and optimizing array/tensor operations. Mullin further explored MoA through case studies of scientific algorithms, including QR Decomposition [12] and Fast Fourier Transforms (FFTs) [6]. The latter paper introduced the dimension lifting operation, crucial to this work.

Table 1. Execution time (in seconds) of a PDE solver C implementation compiled with GCC 8.2.0 depending on hardware and dimension lifting (DL) parameters. The arrays involved in the computation are cubic, and each axis has length 512. The gray background marks the fastest version(s) of the solver for each row. The labels are as follows: S: Single core (no DL); MDL: Multicore (DL on 0^{th} dimension); MDLSL: Multicore (DL on $(n-2)^{\text{th}}$ dimension); MDLTM: Multicore (DL on 0^{th} dimension using tiled memory); CPU 1: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz; CPU 2: AMD EPYC 7601 32-Core; CPU 3: Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz; and CPU 4: ThunderX2 CN9980. The code of the experiments is at https://github.com/mathematics-of-arrays/padding-in-the-mathematics-of-arrays.

	S	MDL	MDLSL	MDLTM
CPU 1	225.74	70.96	66.66	61.81
CPU 2	299.42	59.16	68.14	68.70
CPU 3	172.71	85.97	85.59	117.11
CPU 4	660.53	85.06	72.80	77.86

Burrows et al. identified an array API for FDM solvers of PDEs [3]. We explored the MoA fragment corresponding to this API and concluded that stencil computations can systematically be reduced to MoA's DNF [4]. Hagedorn et al. [5] also looked into optimizing stencil computations, and augmented LIFT [13] with the same operations.

Artjom Šinkarovs studied automatic data layout transformations using a type-based approach and demonstrated that carefully chosen data layouts can greatly improve program vectorisation, therefore leading to significant performance improvements [15]. Šinkarovs et al. also implemented a Convolutional Neural Network in APL [14]—a setting in which stencil-related padding operations are relevant. Šinkarovs's formalization in Agda of multiarrays à la APL in Agda (see https://github.com/ashinkarov/agda-array) uses the latter as an example.

4 MoA Background and Notation

We give a short introduction to the MoA algebra for representing and describing operations on arrays. For more details, we refer the reader to the relevant works in our bibliography [2, 10, 11].

MoA defines the ψ-calculus, a set of rules for manipulating array shapes and expressions. By systematically applying a set of terminating rewriting rules, we can transform an array expression to a single array with standard layout and operations on the array elements, the Denotational Normal Form (DNF). DNF can further be transformed into a corresponding

Operational Normal Form (ONF), which represents array access patterns in terms of *start*, *stride* and *length*. Together with *dimension lifting*, this lets us reorganize the memory layout and data access patterns, and to thus take into account distribution of data and memory hierarchies, data locality, etc., a flexiblity needed for current and future hardware architectures.

The *dimension* of an array A corresponds to the number of axes in A and is denoted by $\mathsf{dim}(A)$. We define the *shape* of an n-dimensional array A as a vector $\langle s_0\ ,\ \ldots\ ,\ s_{n-1}\rangle$ containing at index i the length of i^{th} axis in A. The *size* of an array is the number of elements it contains, i.e. $\mathsf{size}(A) = \prod_{i=0}^{n-1} s_i$; we write this product of shape s also as Πs.

We adopt the notation $\mathsf{Fin}\, n$ for the finite set of natural numbers $\{0, \ldots, n-1\}$. An *index* into A is a vector $\langle i_0\ ,\ \ldots\ ,\ i_k\rangle$ of length $k+1$ with $k \in \mathsf{Fin}\ (\mathsf{dim}(A))$ such that $i_j \in \mathsf{Fin}\, s_j$ for all $j \in \mathsf{Fin}\, k$. If $\mathsf{dim}(A) = k+1$, we say the index is a *total index*. The indexing function that defines the content of the array at a given index differs depending on the abstraction layer we consider.

For example, a 2-dimensional array M with shape $\langle 2, 3\rangle$ contains 6 elements and corresponds to a 2-by-3 matrix. We represent such an array using the row-major notation

$$M = \begin{pmatrix} e_{0,0} & e_{0,1} & e_{0,2} \\ e_{1,0} & e_{1,1} & e_{1,2} \end{pmatrix},$$

where $e_{i,j}$ is the element of M at total index $\langle i\ j\rangle$.

Scalars are represented as 0-dimensional arrays, i.e. arrays with shape $\langle\rangle$ and size 1. Empty arrays have size 0, i.e. at least one of their shape components is 0.

4.1 Relevant MoA Operations at the DNF level

In the following, A is an n-dimensional array with shape $\langle s_0\ ,\ \ldots\ ,\ s_{n-1}\rangle$. The rest of the paper makes use of the following core operations at the DNF level:

- the shape function ρ, that returns the shape of an array, e.g. $\rho(M) = \langle 2, 3\rangle$, where M is the 2-by-3 array from the example above;
- the indexing function ψ, that takes an index into A and returns the subarray at the indexed position. Thus, $\langle\rangle\ \psi\ A = A$ holds. For our example, we have

  $$\langle 1\rangle\ \psi\ M = \begin{pmatrix} e_{1,0} & e_{1,1} & e_{1,2} \end{pmatrix}$$

  and $\rho(\langle 1\rangle\ \psi\ M) = \langle 3\rangle$;
  When A is 1-dimensional and therefore has shape $\langle s_0\rangle$, we also use the notation

  $$A[u],$$

  where the index u is either
  - a scalar $u \in \mathsf{Fin}\, s_0$, and $A[u]$ is the element at index u in A; or
  - a vector of k indices $\langle u_0\ ,\ \ldots\ ,\ u_{k-1}\rangle$ such that $\forall j \in \mathsf{Fin}\, k, u_j \in \mathsf{Fin}\, s_0$, and $A[u]$ is the vector whose j^{th} element is the ${u_j}^{\text{th}}$ element in A;
- the reshaping function reshape, that takes a shape s with $\Pi s = \Pi\rho(A)$, and produces an array with the same size and elements as A but with shape s, i.e. $\rho(\mathsf{reshape}(s, A)) = s$. Note that reshape does not move data around in A;
- a slicing function $\triangle$ (read "take"), that takes a positive (respectively negative) integer t such that $|t| \in \mathsf{Fin}\ (s_0 + 1)$ and returns a slice containing the first (respectively last) $|t|$ subarrays of A. Thus,

  $$\rho(\triangle(t, A)) = \langle |t|\ ,\ s_1\ ,\ \ldots\ ,\ s_{n-1}\rangle$$

  and $\forall i \in \mathsf{Fin}\, |t|$,

  $$\langle i\rangle\ \psi\ \triangle(t, A) = \begin{cases} \langle i\rangle\ \psi\ A & \text{if } t \geq 0 \\ \langle s_0 - |t| + i\rangle\ \psi\ A & \text{otherwise;} \end{cases}$$

- a slicing function $\triangledown$ (read "drop"), that takes a positive (respectively negative) integer t such that $|t| \in \mathsf{Fin}\ (s_0 + 1)$ and returns a slice containing the last (respectively first) $s_0 - |t|$ subarrays of A. Thus,

  $$\rho(\triangledown(t, A)) = \langle s_0 - |t|\ ,\ s_1\ ,\ \ldots\ ,\ s_{n-1}\rangle$$

  and $\forall i \in \mathsf{Fin}\ (s_0 - |t|)$,

  $$\langle i\rangle\ \psi\ \triangledown(t, A) = \begin{cases} \langle i + t\rangle\ \psi\ A & \text{if } t \geq 0 \\ \langle i\rangle\ \psi\ A & \text{otherwise.} \end{cases}$$

- the concatenation function cat, that takes an additional array B with shape $\left\langle s_0^B,\ s_1\ ,\ \ldots\ ,\ s_{n-1}\right\rangle$ such that

  $$\rho(\mathsf{cat}(A, B)) = \left\langle s_0 + s_0^B,\ s_1\ ,\ \ldots\ ,\ s_{n-1}\right\rangle$$

  and

  $$\langle i\rangle\ \psi\ \mathsf{cat}(A, B) = \begin{cases} \langle i\rangle\ \psi\ A & \text{if } i < s_0 \\ \langle i - s_0\rangle\ \psi\ B & \text{otherwise} \end{cases}$$

  hold. In order to simplify notation along the paper, we relax the definition of cat to assume its second argument is automatically reshaped to fit the shape requirements. We use this only in cases when the required reshape operation does not require computing a non-trivial shape argument.
  $\forall t \in \mathsf{Fin}\, s_0, \mathsf{cat}(\triangle(t, A), \triangledown(t, A)) = A$ holds;
- the family of rotation functions θ_j that take a positive (respectively negative) integer o and rotate A by $|o|$ elements to the "right" (respectively left) along axis j. Formally, we have
  $\forall j \in \mathsf{Fin}\ (\mathsf{dim}(A)), o \in \{o \in \mathbb{Z} : |o| \in \mathsf{Fin}\, s_j\}$,

  $$\rho(o\ \theta_j\ A) = \rho(A)$$

  and

  $$i\ \psi\ (o\ \theta_j\ A) = \begin{cases} \mathsf{cat}(\triangle(-o, i\ \psi\ A), \triangledown(-o, i\ \psi\ A)) & \text{if } 0 \leq o \\ \mathsf{cat}(\triangledown(o, i\ \psi\ A), \triangle(o, i\ \psi\ A)) & \text{otherwise;} \end{cases}$$

  where i is a partial index of length j into A.

4.2 Relevant MoA Operations at the ONF level

The following core operations are used throughout the paper at the ONF level:

- the family of dimension lifting operations lift_j that take two natural numbers d, q with $(d, q) \in \{(d, q) : d \cdot q = s_j\}$ and split the j^{th} axis of A into two shape components. More specifically,

$$\begin{aligned}&\mathsf{lift}_j(d, q, A)\\ =\ &\mathsf{reshape}\left(\left\langle s_0\ ,\ \ldots\ ,\ s_{j-1}\ ,\ d\ ,\ q\ ,\ \ldots\ ,\ s_{n-1}\right\rangle, A\right)\end{aligned}$$

  holds. Dimension lifting is syntactic sugar for a specific reshaping operation. The intent is to use dimension lifting when the goal is to distribute computations below axis j across d computation loci. To the best of the knowledge of the authors, it is the first time a formal definition for dimension lifting in MoA is stated in the literature. We later give a formal definition for dimension lifting compatible with padding operations in Definition 10;
- the flattening function rav, which flattens an array into a unidimensional array, i.e.

$$\mathsf{rav}(A) = \mathsf{reshape}(\langle \Pi(\rho(A)) \rangle, A).$$

  We use rav to transport A into its corresponding linear representation in memory;
- the mapping function γ that takes a shape s with $\Pi s = \Pi(\rho(A))$ and a total index into s and produces the corresponding index into $\mathsf{rav}(A)$. Since $\mathsf{rav}(A)$ is 1-dimensional, the equation

$$i\ \psi\ A = (\mathsf{rav}(A))[\gamma(\rho(A), i)]$$

  holds for any total index i into A. Intuitively, γ transforms indexing operations into an abstract array representation of A into one that takes into account its concrete memory layout;
- the range function ι that given a positive integer n returns a 1-dimensional array containing the elements of $\mathsf{Fin}\ n$ in ascending order.

We recall informally the ψ-correspondence theorem [11]: $\forall k \in \mathsf{Fin}\ (\dim(A))$ and an index i of length k into A,

$$i\ \psi\ A = (\mathsf{rav}(A))[\gamma(i, \langle s_0\ ,\ \ldots\ ,\ s_{k-1}\rangle) \cdot \mathit{stride} + \iota\ \mathit{stride}]$$

holds, with $\mathit{stride} = \Pi(\langle s_k\ ,\ \ldots\ ,\ s_{n-1}\rangle)$, and $+$ is the elementwise addition operation with implicit broadcast semantics.

Formal definitions for the operations described above can be found in Mullin's original work [10].

5 Memory Layout in the MoA

In the rest of the paper, we use a row-major memory layout. Our running example will be the DNF expression

$$\mathsf{expr} = ((1\ \theta_0\ \mathsf{Arr}) + (-1\ \theta_0\ \mathsf{Arr}))$$

where $+$ is the elementwise addition operator. For the rest of the paper, we set

$$\mathsf{Arr} = \begin{pmatrix} 1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\\ 17 & 18 & 19 & 20\\ 21 & 22 & 23 & 24 \end{pmatrix},$$

where $\rho(\mathsf{Arr}) = \langle 6, 4\rangle$. More illustratively,

$$\begin{aligned}\mathsf{expr} &= \begin{pmatrix} 21 & 22 & 23 & 24\\ 1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\\ 17 & 18 & 19 & 20 \end{pmatrix} + \begin{pmatrix} 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\\ 17 & 18 & 19 & 20\\ 21 & 22 & 23 & 24\\ 1 & 2 & 3 & 4 \end{pmatrix}\\ &= \begin{pmatrix} 26 & 28 & 30 & 32\\ 10 & 12 & 14 & 16\\ 18 & 20 & 22 & 24\\ 26 & 28 & 30 & 32\\ 34 & 36 & 38 & 40\\ 18 & 20 & 22 & 24 \end{pmatrix}.\end{aligned}$$

The array expression expr is representative of one step of a PDE solver, as considered in the authors' previous work [4] and by Burrows et al. [3].

We can use the ψ-correspondence theorem to transform expr into the following ONF expression:

$$\begin{aligned}&\forall i \in \mathsf{Fin}\ 6,\ \langle i\rangle\ \psi\ ((1\ \theta_0\ \mathsf{Arr}) + (-1\ \theta_0\ \mathsf{Arr}))\\ &\quad = (\mathsf{rav}\ \mathsf{Arr})[\gamma(\langle (i+1) \bmod 6\rangle, \langle 6\rangle) \cdot 4 + \iota 4] +\\ &\quad\ \ (\mathsf{rav}\ \mathsf{Arr})[\gamma(\langle (i-1) \bmod 6\rangle, \langle 6\rangle) \cdot 4 + \iota 4].\end{aligned}$$

We follow up by unfolding γ:

$$\begin{aligned}&= (\mathsf{rav}\ \mathsf{Arr})[((i+1) \bmod 6) \cdot 4 + \iota 4] +\\ &\quad\ (\mathsf{rav}\ \mathsf{Arr})[((i-1) \bmod 6) \cdot 4 + \iota 4].\end{aligned}$$

By unfolding rav and turning ι into a loop we get the following generic program:

$$\begin{aligned}&\forall i \in \mathsf{Fin}\ 6,\ j \in \mathsf{Fin}\ 4,\\ &\quad (\mathsf{rav}\ \mathsf{Arr})[((i+1) \bmod 6) \cdot 4 + j] +\\ &\quad (\mathsf{rav}\ \mathsf{Arr})[((i-1) \bmod 6) \cdot 4 + j].\end{aligned}$$

The above program is written assuming, implicitly, that the target architecture is a single-core processor. We can use dimension lifting to establish a correspondence between the shape of the array and a different underlying hardware architecture.

Consider an architecture that consists of two single-core processors. We apply dimension lifting on axis 1 of Arr, to create the array

$$\mathsf{Arr}' = \mathsf{lift}_1(2, 2, A) = \mathsf{reshape}\left(\langle 6, 2, 2\rangle, A\right)$$

where axis 1 corresponds to the number of available cores. We get the following:

$$\begin{aligned}
&\forall i \in \mathsf{Fin}\,6,\ \ j \in \mathsf{Fin}\,2,\\
&\quad \langle i, j\rangle \ \psi\ ((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}'))\\
&= (\mathsf{rav}\,\mathsf{Arr}')[\gamma(\langle(((i+1) \bmod 6)\ j\rangle, \langle 6, 2\rangle) \cdot 2 + \imath 2] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[\gamma(\langle(((i-1) \bmod 6)\ j\rangle, \langle 6, 2\rangle) \cdot 2 + \imath 2]\\
&= (\mathsf{rav}\,\mathsf{Arr}')[(((i+1) \bmod 6) \cdot 2 + j) \cdot 2 + \imath 2] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[(((i-1) \bmod 6) \cdot 2 + j) \cdot 2 + \imath 2].
\end{aligned}$$

This reduces to the following generic program:

$$\begin{aligned}
&\forall i \in \mathsf{Fin}\,6, j \in \mathsf{Fin}\,2, k \in \mathsf{Fin}\,2,\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[((i+1) \bmod 6) \cdot 4 \cdot 2 + j \cdot 2 + k] +\\
&\quad (\mathsf{rav}\,\mathsf{Arr}')[((i-1) \bmod 6) \cdot 4 \cdot 2 + j \cdot 2 + k].
\end{aligned}$$

The programs before and after dimension lifting above are equivalent except for their different looping structures—they are adapted to two different hardware architectures.

Dimension lifting can be carried out across any axis (or on several axes simultaneously). The choice of axes should be guided by both the memory hierarchy and the operations involved in the expression. For example, the rotations above are applied on axis 0; dimension lifting on this axis would not allow perfectly splitting the memory between the two processors.

The example involves a modulo operation on the index. This is an expensive operation even on modern hardware [9]. We describe below a *circular padding* operation on DNF expressions that introduces data redundancy into arrays.

In Section 5.1, we define circular padding operations and observe how they eliminate the need for *modulo* operations in a single-core setting for our running example. In Section 5.2 we generalize these operations and put them to work to reduce the need for inter-process communication in a distributed computation setting for our running example.

5.1 Case of One Core and Constant Memory Access Cost

Padding an array is prepending or appending data to it. For our purposes, we want these data to be specific slices of the array itself. Here we introduce notation to define the circular prepending (referred to as *left padding*) and circular appending (referred to as *right padding*) operations in MoA.

Notation 1. Given an n-dimensional array A and an integer $i \in \mathsf{Fin}\,n$, we use the shorthand notation K_i to represent the index of length i into A whose j^{th} component is bound to variable k_j, i.e.

$$K_i = \langle k_0\ ,\ \ldots\ ,\ k_{i-1}\rangle\,.$$

To further simplify the notation of indexing, we also introduce the shorthand notation

$$A_{K_i} = A_{\langle k_0\ ,\ \ldots\ ,\ k_{i-1}\rangle} = \langle k_0\ ,\ \ldots\ ,\ k_{i-1}\rangle\ \psi\ A.$$

Definition 2. Let A be an n-dimensional array with shape $\langle s_0\ ,\ \ldots\ ,\ s_{n-1}\rangle$. We can specify an n-dimensional slice B of A by annotating each component s_i of its shape with the (inclusive) beginning of the slice $b_i \in \mathsf{Fin}\ (s_i + 1)$ and the (exclusive) end $e_i \in \mathsf{Fin}\ (s_i + 1)$, with $b_i \le e_i$. Concretely, we have

$$\rho(B) = \langle e_0 - b_0\ ,\ \ldots\ ,\ e_{n-1} - b_{n-1}\rangle$$

and

$$B_{\langle k_0\ ,\ \ldots\ ,\ k_{n-1}\rangle} = A_{\langle k_0+b_0\ ,\ \ldots\ ,\ k_{n-1}+b_{n-1}\rangle}.$$

In the rest of this section, we attach a slice annotation to each of our arrays. We write

$$\rho_{\text{ann}}(A) = \left\langle s_0^{b_0,e_0},\ \ldots\ ,\ s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle$$

the projection function that extracts both the shape and the slice annotation from an array.

We care about making a difference between padded and unpadded arrays. In the following, it is assumed that if A is unpadded, it carries the slice annotation such that $\forall i \in \mathsf{Fin}\ (\mathsf{dim}(A)), b_i = 0, e_i = s_i$.

Definition 3. Given an array A such that

$$\rho_{\text{ann}}(A) = \left\langle s_0^{b_0,e_0},\ \ldots\ ,\ s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle$$

and an integer $i \in \mathsf{Fin}\ n$ we define the right circular padding operation on axis i as padr_i such that

$$\mathsf{padr}_i(A)_{K_i} = \mathsf{cat}(A_{K_i}, A_{\langle k_0\ ,\ \ldots\ ,\ k_{i-1}\ ,\ b_i+s_i-e_i\rangle})$$

for j, k_j integers such that $0 \le j < i$, $0 \le k_j < s_j$. Notice that this uses our overloaded definition of cat, where the second parameter is implicitly reshaped as needed. The shape of the result is given by

$$\rho_{\text{ann}}(\mathsf{padr}_i(A)) = \left\langle s_0^{b_0,e_0},\ \ldots\ ,\ (s_i+1)^{b_i,e_i},\ \ldots\ ,\ s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.$$

As an example, assume $\rho_{\text{ann}}(A) = \left\langle 2^{0,2}, 2^{0,2}\right\rangle$,

$$A = \begin{pmatrix} 1 & 2\\ 3 & 4\end{pmatrix}$$

then

$$\mathsf{padr}_0(A)_{K_0} = \mathsf{padr}_0(A)_{\langle\rangle} = \mathsf{cat}(A_{\langle\rangle}, A_{0+2-2}) = \begin{pmatrix} 1 & 2\\ 3 & 4\\ 1 & 2\end{pmatrix}.$$

In the same way, we define the left circular padding operation on axis i as padl_i such that

$$\mathsf{padl}_i(A)_{K_i} = \mathsf{cat}(A_{\langle k_0\ ,\ \ldots\ ,\ k_{i-1}\ ,\ e_i-b_i-1\rangle}, A_{K_i}),$$

whose shape is given by

$$\begin{aligned}
&\rho_{\text{ann}}(\mathsf{padl}_i(A))\\
&= \left\langle s_0^{b_0,e_0},\ \ldots\ ,\ (s_i+1)^{b_i+1,e_i+1},\ \ldots\ ,\ s_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.
\end{aligned}$$

Finally, we write padl^{-1} (respectively padr^{-1}) for the left inverse function of padl (respectively padr).

Recall our running example

$$\text{expr} = ((1\ \theta_0\ \text{Arr}) + (-1\ \theta_0\ \text{Arr})),$$

which we reduced to

$\forall i \in \text{Fin}\,6,\ j \in \text{Fin}\,4,$

$$(\text{rav}\,\text{Arr})[((i+1) \bmod 6)\cdot 4 + j] + (\text{rav}\,\text{Arr})[((i-1) \bmod 6)\cdot 4 + j].$$

using the ψ-correspondence theorem while assuming a single-core processor as the underlying hardware architecture.

We apply Definition 3 to obtain the following:

$$\begin{aligned}\text{expr} &= \text{padr}_0^{-1}(\text{padl}_0^{-1}(\text{padl}_0(\text{padr}_0(\text{expr}))))\\ &= \text{padr}_0^{-1}(\text{padl}_0^{-1}(\text{padl}_0(\text{padr}_0((1\ \theta_0\ \text{Arr}) + (-1\ \theta_0\ \text{Arr})))))).\end{aligned}$$

Proposition 4. For any axis i, padl_i and padr_i commute, i.e.

$$\text{padl}_i \circ \text{padr}_i = \text{padr}_i \circ \text{padl}_i\,.$$

Proposition 4 follows from the associativity of cat.

Proposition 5. Let A be an array without right padding, i.e. an array such that

$$\rho_{\text{ann}}(A) = \left\langle s_0^{b_0,s_0},\ \ldots\ ,\ s_{n-1}^{b_{n-1},s_{n-1}} \right\rangle.$$

For all $i \in \text{Fin}\,n$ and $m \in \text{Fin}\ (s_i + 1)$,

$$\text{padr}_i^m(A)_{K_i} = \text{cat}(A_{K_i}, \triangle(m, \triangledown(b_i, A_{K_i}))).$$

In the same way, for A an array without left padding, we have

$$\text{padl}_i^m(A)_{K_i} = \text{cat}(\triangledown(e_i - b_i - m, \triangle(e_i - b_i, A_{K_i})), A_{K_i}).$$

Both cases of can be shown by induction on m.

Proposition 6. Let B, C be n-dimensional MoA expressions with $\rho(B) = \rho(C)$ and $\oplus$ a binary map operation. Then $\forall i \in \text{Fin}\,n$, padr_i is distributive over $\oplus$, i.e.

$$\text{padr}_i(B) \oplus \text{padr}_i(C) = \text{padr}_i(B \oplus C)$$

holds. Similarly, for padl_i

$$\text{padl}_i(B) \oplus \text{padl}_i(C) = \text{padl}_i(B \oplus C)$$

holds. This idea is easily extensible to n-ary map operations.

Proof. To improve readability, we write

$$T_i = \langle k_0\ ,\ \ldots\ ,\ k_{i-1}\ ,\ b_i + s_i - e_i \rangle\,.$$

In the case of padr_i, since $\oplus$ is a binary map operation, we have:

$$\begin{aligned}&(\text{padr}_i(B) \oplus \text{padr}_i(C))_{K_i} = \text{cat}(B_{K_i}, B_{T_i}) \oplus \text{cat}(C_{K_i}, C_{T_i})\\ \Leftrightarrow\ &(\text{padr}_i(B) \oplus \text{padr}_i(C))_{K_i} = \text{cat}(B_{K_i} \oplus C_{K_i}, B_{T_i} \oplus C_{T_i})\\ \Leftrightarrow\ &(\text{padr}_i(B) \oplus \text{padr}_i(C))_{K_i} = \text{padr}_i(B \oplus C)_{K_i}\\ \Leftrightarrow\ &\text{padr}_i(B) \oplus \text{padr}_i(C) = \text{padr}_i(B \oplus C).\end{aligned}$$

The case for padl_i can be shown using the same reasoning. □

By applying Proposition 6 in our example, we get:

$$\begin{aligned}\text{expr} = \text{padr}_0^{-1}(\text{padl}_0^{-1}(&\text{padl}_0(\text{padr}_0(1\ \theta_0\ \text{Arr})) +\\ &\text{padl}_0(\text{padr}_0(-1\ \theta_0\ \text{Arr})))).\end{aligned}$$

Proposition 7. Let B be a n-dimensional unpadded MoA expression, j an axis of B and r an integer. Then, on any axis i of B, we have that

$$r\ \theta_j\ B = \text{padr}_i^{-m_2}(\text{padl}_i^{-m_1}(r\ \theta_j\ \text{padl}_i^{m_1}(\text{padr}_i^{m_2}(B))))$$

holds if either one of the following cases holds:

(i) $j \neq i$;
(ii) $r = 0$;
(iii) $r < 0$ and $m_2 \geq |r|$;
(iv) $r > 0$ and $m_1 \geq r$.

Proof. Cases (i) and (ii) are trivial. In case (i), padding does not affect the rotation. In case (ii), $0\ \theta_j\ B = B$ holds. We thus want to prove that

$$i = j, r < 0, m_2 \geq |r|$$

implies

$$r\ \theta_j\ B = \text{padr}_i^{-m_2}(\text{padl}_i^{-m_1}(r\ \theta_j\ \text{padl}_i^{m_1}(\text{padr}_i^{m_2}(B))))$$

Using Proposition 5, we can write $\text{padl}_i^{m_1}(\text{padr}_i^{m_2}(B))$ as an array A such that

$$A_{K_i} = \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, \triangledown(b_i, B_{K_i}))))$$

where p represents the left-padding of the array. Since B is originally unpadded, we rewrite A as:

$$\begin{aligned}A_{K_i} &= \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, \triangledown(0, B_{K_i}))))\\ &= \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, B_{K_i}))).\end{aligned}$$

Since $r < 0$, we have:

$$\begin{aligned}(r\ \theta_i\ A)_{K_i} &= \text{cat}(\triangledown(|r|, \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, B_{K_i})))),\\ &\qquad \triangle(|r|, \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, B_{K_i})))))\\ &= \text{cat}(\triangledown(|r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(m_2, B_{K_i}))),\\ &\qquad \triangle(|r|, \text{cat}(\text{p}, \text{cat}(B_{K_i}, \triangle(m_2, B_{K_i}))))).\end{aligned}$$

Using Proposition 4, we get

$$\text{padr}_i^{-m_2}(\text{padl}_i^{-m_1}(r\ \theta_i\ A)) = \text{padl}_i^{-m_1}(\text{padr}_i^{-m_2}(r\ \theta_i\ A)),$$

and since $m_2 \geq |r|$, we have

$$\begin{aligned}&\text{padr}_i^{-m_2}(r\ \theta_i\ A))_{K_i}\\ &\quad= \triangledown(|r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(m_2 - (m_2 - |r|), B_{K_i})))\\ &\quad= \triangledown(|r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(|r|, B_{K_i}))).\end{aligned}$$

We can thus write:

$$\begin{aligned}&\text{padl}_i^{-m_1}(\text{padr}_i^{-m_2}(r\ \theta_i\ A)))_{K_i}\\ &\quad= \triangledown(m_1, \triangledown(|r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(|r|, B_{K_i}))))\\ &\quad= \triangledown(m_1 + |r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(|r|, B_{K_i}))).\end{aligned}$$

Since r is a valid rotation offset in B, we can write

$$\begin{aligned}&\triangledown(m_1 + |r|, \text{cat}(\text{cat}(\text{p}, B_{K_i}), \triangle(|r|, B_{K_i})))\\ &\quad= \text{cat}(\triangledown(m_1 + |r|, \text{cat}(\text{p}, B_{K_i})), \triangle(|r|, B_{K_i}))\end{aligned}$$

$$= \mathrm{cat}(\triangledown(|r|, B_{K_i}), \triangle(|r|, B_{K_i}))$$
$$= (r\ \theta_i\ B)_{K_i}$$

and thus

$$r\ \theta_j\ B = \mathrm{padr}_i^{-m_2}(\mathrm{padl}_i^{-m_1}(r\ \theta_j\ \mathrm{padl}_i^{m_1}(\mathrm{padr}_i^{m_2}(B))))$$

by function extensionality.
The proof for case 4 follows the same pattern as case 3 on the opposite side of the array. □

Proposition 8. Given an array expression A with

$$\rho(A) = \langle s_0\ ,\ \ldots\ ,\ s_{n-1}\rangle\,,$$

some $i \in \mathsf{Fin}\, n$, and two positive integers m_1, m_2,

$$\mathrm{padl}_i^{m_1}(\mathrm{padr}_i^{m_2}(A))_{\langle k_0\ ,\ \ldots\ ,\ m_1+k_i\ ,\ \ldots\ ,\ k_{n-1}\rangle} = A_{\langle k_0\ ,\ \ldots\ ,\ k_{n-1}\rangle}$$

with $\forall j, k_j \in \mathsf{Fin}\, s_j$ holds.

Proposition 8 can be proven by definition of padding.

The insight behind Proposition 8 is that the content of A is maintained in the padded expression B, but that the evaluation of A within B may behave differently due to the shift in indexing and duplication of data brought by padding.

We call the values m_1 and m_2 in Proposition 8 *consumption speed* for a given axis i in the following, and define a function speed_i on expressions such that

$$\mathrm{speed}_i(B) = (m_1, m_2).$$

Note that in practice, the choice of m_1 and m_2 is made by the user of MoA based on their goals.

We apply Proposition 7, and get:

$$\mathsf{expr} = \mathrm{padr}_0^{-1}(\mathrm{padl}_0^{-1}((1\ \theta_0\ \mathrm{padl}_0(\mathrm{padr}_0(\mathsf{Arr}))) + (-1\ \theta_0\ \mathrm{padl}_0(\mathrm{padr}_0(\mathsf{Arr})))))$$

In order to get rid of the mod operation in the ONF expression we built for expr, we create a new array Arr′ defined by

$$\mathsf{Arr}' = \mathrm{padl}_0(\mathrm{padr}_0(\mathsf{Arr}))$$

From Definition 3, we have that

$$\rho_{\mathrm{ann}}(\mathsf{Arr}') = \left\langle 8^{1,7}, 4\right\rangle$$

and

$$\mathsf{Arr}' = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

We rewrite our example with the new definition of Arr′, and get

$$\mathsf{expr} = \mathrm{padr}_0^{-1}(\mathrm{padl}_0^{-1}((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}'))).$$

We can now transform it to ONF again. The bounds of the relevant indices i and j are given by Proposition 8. We obtain the following:

$$\begin{aligned}&\forall i \in \{i \in \mathsf{Fin}\, 8\ :\ 1 \le i < 7\},\\ &\quad \langle i, j\rangle\ \psi\ ((1\ \theta_0\ \mathsf{Arr}') + (-1\ \theta_0\ \mathsf{Arr}'))\\ &\equiv (\mathrm{rav}\,\mathsf{Arr}')[\gamma(\langle i+1\rangle ; \langle 8\rangle) \times 4 + \iota 4] +\\ &\quad (\mathrm{rav}\,\mathsf{Arr}')[\gamma(\langle i-1\rangle ; \langle 8\rangle) \times 4 + \iota 4].\end{aligned}$$

We then apply γ, rav, and turn ι into a loop to get the following generic program:

$$\begin{aligned}&\forall i \in \{i \in \mathsf{Fin}\, 8\ :\ 1 \le i < 7\},\ j \in \mathsf{Fin}\, 4,\\ &\quad (\mathrm{rav}\,\mathsf{Arr}')[(i+1) \times 4 + j] + A'[(i-1) \times 4 + j].\end{aligned}$$

Finally, we apply the composition $\mathrm{padr}_0^{-1} \circ \mathrm{padl}_0^{-1}$ and retrieve the exact same result as we would have gotten by directly evaluating expr. Notice that thanks to the notion of consumption speed, we avoided performing the computation on irrelevant indices. In the end, both of the expressions had 6 loop iterations, but we managed to get rid of the expensive mod operation by adding data redundancy into Arr.

5.2 Case of Non-Uniform Memory Access

Consider now that expr is embedded within a loop and must be executed several times. Then, in order to avoid the mod operation at each iteration, the array must be padded at each iteration as well.

Considering hardware and software implementations, it is reasonable to investigate a case in which the application of a big padding operation $p = \mathrm{padl}_i^n \circ \mathrm{padr}_i^m$ for some natural numbers n, m, i at a given point in the program is cheaper than applying parts of p in different parts of the program.

For example, if the padding operation depends on interprocess communication, it is usually significantly cheaper to open one socket and send four elements than to open two sockets each sending two elements (each opened connection probably also requiring synchronization of some sort, etc). We however consider unpadding to have negligible cost.

To reduce the number of loci where a padding operation is required and to use the resulting padding efficiently, we need to define a slightly more complex padding function as well as further notation. We also need to define the notion of *padding exhaustion*.

Informally, padding exhaustion corresponds to reaching a state where there is not enough unconsumed padding left to evaluate the expression and achieve our goals of using padding. Padding exhaustion is related to consumption speed. In our example, padding is exhausted when both of the equivalent evaluation strategies stated in Proposition 8 fail to get rid of all mod operations.

Ideally, this state is reached at the end of the computation of all the expressions; if reached in the middle of execution, the padding must be replenished to proceed.

Definition 9. Let A be an n-dimensional array with shape $\langle s_0\ ,\ \ldots\ ,\ s_{n-1}\rangle$. We can specify a $2n$-dimensional reshaping D of A by annotating each component s_i of its shape with a divisor d_i such that $s_i \equiv 0 \bmod d_i$. We then reshape A into an array D such that

$$\rho(D) = \langle d_0\ ,\ q_0\ ,\ \ldots\ ,\ d_{n-1}\ ,\ q_{n-1}\rangle$$

where $q_i = \frac{s_i}{d_i}$. Assume we have

$$\rho_{\text{ann}}(D) = \left\langle d_0^{0,d_0}\ ,\ q_0^{b_0,e_0}\ ,\ \ldots\ ,\ d_{n-1}^{0,d_{n-1}}\ ,\ q_{n-1}^{b_{n-1},e_{n-1}}\right\rangle.$$

Then, we can specify an n-dimensional slice B of A such that

$$\rho(B) = \langle (e_0 - b_0)\times d_0\ ,\ \ldots\ ,\ (e_{n-1} - b_{n-1})\times d_{n-1}\rangle$$

and

$$B_{K_n} = D_{\left\langle \frac{k_0}{e_0-b_0}\ ,\ k_0 \bmod (e_0-b_0)\ ,\ \ldots\ ,\ \frac{k_{n-1}}{e_{n-1}-b_{n-1}}\ ,\ k_{n-1}\bmod (e_{n-1}-b_{n-1})\right\rangle}.$$

We write

$$\rho_{\text{ann}^+}(A) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle$$

the projection function that extracts this "distributed slice annotation" from the array.
In the following definitions, we reuse the present definition of q_i.

Definition 10. Let A be an array such that

$$\rho_{\text{ann}^+}(A) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle,$$

We define the padding-compatible dimension lifting operation on axis i $\text{liftp}_i(A) = B$ such that B has $n+1$ dimensions,

$$\rho_{\text{ann}^+}(B) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ d_i\ ,\ q_i^{1,b_i,e_i}\ ,\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle,$$

and

$$B_{K_i} = \triangle(q_i, \triangledown(k_{i-1}\times q_i, A_{K_{i-1}})).$$

liftp is a version of lift that extracts its parameters from and modifies the "distributed slice annotation" of the array.

Definition 11. Consider an array A such that

$$\rho_{\text{ann}^+}(A) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle.$$

In a MoA setting without any notion of padding, any array is implicitly annotated with $d_i = 1$, $b_i = 0$ and $e_i = s_i$ on any given axis i. To properly use liftp as it is defined above, we do the following: assuming $b_i = 0$ and $e_i = s_i$ for a given axis i of A, we define the prelift operation on that axis for any $d \in \{d : s_i \equiv 0 \bmod d\}$ as $\text{prelift}_i(d, A) = B$,

$$\rho_{\text{ann}^+}(B) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_i^{d,0,\frac{s_i}{d}},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle$$

and

$$B_{K_n} = A_{K_n}.$$

The precondition on prelift_i means that it can only be applied to arrays that are unpadded on axis i.

Recall once again our running example

$$\text{expr} = ((1\ \theta_0\ \text{Arr}) + (-1\ \theta_0\ \text{Arr})),$$

which we previously reduced to

$$\forall i \in \text{Fin}\,6,\ j \in \text{Fin}\,4,\quad (\text{rav Arr})[((i+1) \bmod 6)\cdot 4 + j] + (\text{rav Arr})[((i-1) \bmod 6)\cdot 4 + j].$$

using the ψ-correspondence theorem while assuming a single-core processor as the underlying hardware architecture.

We wish to distribute the computation over two machines. We will achieve this through a combination of dimension lifting and padding. To distribute the computation over two machines, it is natural to perform dimension lifting along the 0^{th} axis of Arr, taking $d_0 = 2$. We thus start out by creating a new array Arr′ such that

$$\text{Arr}' = \text{prelift}_0(2, \text{Arr}).$$

From Definition 11, we have that

$$\rho_{\text{ann}^+}(\text{Arr}') = \left\langle 6^{2,0,3}, 4\right\rangle.$$

Since prelift_0 does not modify the layout of the array it operates on in any way, we have

$$\text{expr} = (1\ \theta_0\ \text{Arr}') + (-1\ \theta_0\ \text{Arr}').$$

Definition 12. Given an array A with shape

$$\left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle$$

and an integer $i \in \text{Fin}\,n$ we define the right pre-dimension lifting padding operation on axis i as $\text{dpadr}_i(A) = R$ such that

$$\text{liftp}_i(R)_{K_i} = \text{cat}(\triangle(q_i, \triangledown(k_{i-1}\times q_i, A_{K_{i-1}})), A_{\langle k_0\ ,\ \ldots\ ,\ ((k_{i-1}+2)\times q_i + b_i - e_i) \bmod s_i\rangle})$$

for $j \in \text{Fin}\,i$ and $k_j \in \text{Fin}\,s_j$.

Note that we consider operations on the axis i to be done in $\text{Fin}\,n$, e.g. for $i = 0$, we have $k_{i-1} = k_{n-1}$.

The shape $\rho_{\text{ann}^+}(R)$ is as in:

$$\left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ (s_i + d_i)^{d_i,b_i,e_i},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle.$$

In the same way, we define the left pre-dimension lifting padding operation on axis i as $\text{dpadl}_i(A) = L$ such that

$$\text{liftp}_i(L)_{K_i} = \text{cat}(A_{\langle k_0\ ,\ \ldots\ ,\ ((k_{i-1}-1)\times q_i + e_i - b_i - 1) \bmod s_i\rangle}, \triangle(q_i, \triangledown(k_{i-1}\times q_i, A_{K_{i-1}})))$$

The shape of L is as in:

$$\left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ (s_i + d_i)^{d_i,b_i+1,e_i+1},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}}\right\rangle.$$

Finally, we call dpadl^{-1} (respectively dpadr^{-1}) the left inverse function of dpadl (respectively dpadr).

We are now ready to start padding Arr′. In this case, we would like the two workers to only communicate at the start and at the end of the computation. To do that, we need to provide each machine with enough padding to do all of the required computations in one go.

Similarly to the approach we took in the previous section, we create a new array Arr′′ such that

$$\mathsf{Arr}'' = \mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr}')).$$

From Definition 12, we have that

$$\rho_{\mathrm{ann}^+}(\mathsf{Arr}'') = \left\langle 10^{2,1,4}, 4 \right\rangle$$

and

$$\mathsf{Arr}'' = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

Finally, we create an array $\mathsf{Arr}^+ = \mathsf{liftp}_0(\mathsf{Arr}'')$. From Definition 10, we have:

$$\rho_{\mathrm{ann}^+}(\mathsf{Arr}^+) = \left\langle 2, 5^{1,4}, 4 \right\rangle$$

and

$$\mathsf{Arr}^+_{\langle 0 \rangle} = \left(\begin{array}{cccc} 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \end{array}\right),$$

$$\mathsf{Arr}^+_{\langle 1 \rangle} = \left(\begin{array}{cccc} 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \end{array}\right).$$

By definition of dpadr and dpadl, we have the following:

$$\begin{aligned} &\mathsf{expr} \\ &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{expr})))) \\ &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{expr}))))))). \end{aligned}$$

Proposition 13. For a given axis i, the functions dpadl_i and dpadr_i commute, i.e.

$$\mathsf{dpadl}_i \circ \mathsf{dpadr}_i = \mathsf{dpadr}_i \circ \mathsf{dpadl}_i\,.$$

Proposition 13 can be proven using the associativity of cat.

Proposition 14. Let A be an array and $i \in \mathsf{Fin}\ (\dim(A))$. Let

$$R = \mathsf{dpadr}_i(A)$$
$$L = \mathsf{dpadl}_i(A)$$

then

$$\mathsf{padr}_i^{-1}(\mathsf{liftp}_i(R)_{K_i}) = \triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}})) \quad (1)$$
$$\mathsf{padl}_i^{-1}(\mathsf{liftp}_i(L)_{K_i}) = \triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}})) \quad (2)$$

hold.

Proof. We give a proof for Equation 1:

$$\begin{aligned} &\mathsf{padr}_i^{-1}(\,\mathsf{liftp}_i(R)_{K_i}) \\ &\quad = \mathsf{padr}_i^{-1}(\mathsf{cat}(\triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}})), \\ &\qquad\qquad A_{\langle k_0\ ,\ \ldots\ ,\ ((k_{i-1}+2)\times q_i + b_i - e_i) \bmod s_i \rangle})) \\ &\quad = \triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}})). \end{aligned}$$

The proof for Equation 2 follows the same pattern as the above. □

Informally, Proposition 14 tells us that for a given array A' resulting from padding and dimension lifting an array A on an axis i, unpadding and concatenating all the subarrays resulting from the dimension lifting operation is the same as concatenating them and *unpadding* the result.

Proposition 15. Let A be an array without right padding on its i^{th} axis, i.e. an array such that

$$\rho_{\mathrm{ann}^+}(A) = \left\langle s_0^{d_0,b_0,e_0},\ \ldots\ ,\ s_{n-1}^{d_{n-1},b_{n-1},e_{n-1}} \right\rangle$$

with $e_i = s_i$, $i \in \mathsf{Fin}\ n$. Given an integer $m \in \mathsf{Fin}\ (s_i + 1)$, let $A' = \mathsf{dpadr}_i^m(A)$. Then, the following holds:

$$\begin{aligned} &\mathsf{liftp}_i(A')_{K_i} \\ &= \mathsf{cat}(\triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}})), \\ &\qquad \triangle(m, \nabla(((k_{i-1}+1) \times q_i + b_i) \bmod s_i, A_{K_{i-1}}))) \end{aligned}$$

In the same way, for A an array without left padding and $A'' = \mathsf{dpadl}_i^m(A)$, then

$$\begin{aligned} &\mathsf{liftp}_i(A'')_{K_i} \\ &= \mathsf{cat}(\nabla(e_i - b_i - m, \\ &\qquad \triangle(e_i - b_i, \nabla((k_{i-1}-1) \times q_i \bmod s_i, A_{K_{i-1}}))), \\ &\quad \triangle(q_i, \nabla(k_{i-1} \times q_i, A_{K_{i-1}}))) \end{aligned}$$

holds.

A proof for Proposition 15 may be written using induction on m, like the proof for Proposition 5.

Proposition 16. Let B, C be n-dimensional MoA expressions with identical shapes and $\oplus$ a binary map operation. Then, liftp_i distributes over $\oplus$, i.e.

$$\mathsf{liftp}_i(B \oplus C) = \mathsf{liftp}_i(B) \oplus \mathsf{liftp}(C) \quad (3)$$

for any axis i of B and C. This idea is trivially extensible to n-ary map operations.

Proposition 16 can be proven using the definition of liftp, and the shape-conserving property of n-ary map operations.

Proposition 17. Let B, C be n-dimensional MoA expressions with identical shapes and $\oplus$ a binary map operation. Then, dpadr_i distributes over $\oplus$, i.e.

$$\mathsf{dpadr}_i(B \oplus C) = \mathsf{dpadr}_i(B) \oplus \mathsf{dpadr}_i(C). \tag{4}$$

Similarly, we have that

$$\mathsf{dpadl}_i(B \oplus C) = \mathsf{dpadl}_i(B) \oplus \mathsf{dpadl}_i(C). \tag{5}$$

This idea is trivially extensible to n-ary map operations.

Proof. To improve readability, we write

$$T_i = \langle k_0\ ,\ \ldots\ ,\ ((k_{i-1}+2) \times q_i + b_i - e_i) \bmod s_i \rangle\ .$$

Since $\oplus$ is a binary map operation, we have:

$$\begin{aligned}
&(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} = \\
&\mathsf{cat}(\triangle(q_i, \nabla(k_{i-1} \times q_i, B_{K_{i-1}})), B_{T_i}) \oplus \\
&\mathsf{cat}(\triangle(q_i, \nabla(k_{i-1} \times q_i, C_{K_{i-1}})), C_{T_i}) \\
\Leftrightarrow\ &(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} = \\
&\mathsf{cat}(\triangle(q_i, \nabla(k_{i-1} \times q_i, B_{K_{i-1}})) \oplus \triangle(q_i, \nabla(k_{i-1} \times q_i, C_{K_{i-1}})), \\
&\quad B_{T_i} \oplus C_{T_i}) \\
\Leftrightarrow\ &(\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)))_{K_i} = \\
&(\mathsf{liftp}_i(\mathsf{dpadr}_i(B \oplus C)))_{K_i} \\
\Leftrightarrow\ &\mathsf{liftp}_i(\mathsf{dpadr}_i(B)) \oplus \mathsf{liftp}_i(\mathsf{dpadr}_i(C)) = \\
&\mathsf{liftp}_i(\mathsf{dpadr}_i(B \oplus C)) \\
\Leftrightarrow\ &\mathsf{dpadr}_i(B) \oplus \mathsf{dpadr}_i(C) = \mathsf{dpadr}_i(B \oplus C).
\end{aligned}$$

The proof for Equation 5 follows the same pattern as above. Since it does not provide any additional insight, we do not develop it here. □

By applying Propositions 16 and 17 in our example, we get:

$$\begin{aligned}
\mathsf{expr} = \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\\
\mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(1\ \theta_0\ \mathsf{Arr}))) + \\
\mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(-1\ \theta_0\ \mathsf{Arr})))))).
\end{aligned}$$

Proposition 18. Let A be a n-dimensional unpadded MoA expression, j an axis of A and r an integer. Then, on any axis i of A, we have that

$$\begin{aligned}
r\ \theta_j\ A\ &=\ \mathsf{dpadr}_i^{-m_2}(\mathsf{dpadl}_i^{-m_1}(\mathsf{liftp}_i^{-1}(\\
&\qquad \mathsf{liftp}_i(\mathsf{dpadl}_i^{m_1}(\mathsf{dpadr}_i^{m_2}(r\ \theta_j\ A)))))) \\
&=\ \mathsf{dpadr}_i^{-m_2}(\mathsf{dpadl}_i^{-m_1}(\mathsf{liftp}_i^{-1}(\\
&\qquad r\ \theta_j\ \mathsf{liftp}_i(\mathsf{dpadl}_i^{m_1}(\mathsf{dpadr}_i^{m_2}(A)))))).
\end{aligned}$$

holds if either one of the following cases holds:

(i) $j \neq i$;
(ii) $r = 0$;
(iii) $r < 0$ and $m_2 \geq |r|$;
(iv) $r > 0$ and $m_1 \geq r$.

The proof for Proposition 18 follows the same pattern as the proof for Proposition 7.

We can now apply Proposition 18 in our example, and get:

$$\begin{aligned}
&\mathsf{expr} \\
=\ &\mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\\
&(1\ \theta_0\ \mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr})))) + \\
&(-1\ \theta_0\ \mathsf{liftp}_0(\mathsf{dpadl}_0(\mathsf{dpadr}_0(\mathsf{Arr})))))) \\
=\ &\mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}((1\ \theta_0\ \mathsf{Arr}^+) + (-1\ \theta_0\ \mathsf{Arr}^+)))).
\end{aligned}$$

We can now transform the resulting expression expr to ONF for each machine. The bounds of i and j are once again given by Proposition 8. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned}
&\forall i \in \{i \in \mathsf{Fin}\ 5\ :\ 1 \leq i < 4\}, \\
&\qquad \langle i \rangle\ \psi\ ((1\ \theta_0\ \mathsf{Arr}^+_{\langle c \rangle}) + (-1\ \theta_0\ \mathsf{Arr}^+_{\langle c \rangle})) \\
&\quad \equiv (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[\gamma(\langle i+1 \rangle ; \langle 5 \rangle) \times 4 + \iota 4] + \\
&\qquad (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[\gamma(\langle i-1 \rangle ; \langle 5 \rangle) \times 4 + \iota 4].
\end{aligned}$$

We then apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned}
&\forall i \in \{i \in \mathsf{Fin}\ 5\ :\ 1 \leq i < 4\}, j \in \mathsf{Fin}\ 4, \\
&\qquad (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[(i+1) \times 4 + j] + \\
&\qquad (\mathsf{rav}\,\mathsf{Arr}^+_{\langle c \rangle})[(i-1) \times 4 + j].
\end{aligned}$$

Finally, we join the results using liftp_0^{-1} and apply dpadl_0^{-1} and dpadr_0^{-1} to obtain the same result as we would have gotten evaluating expr directly. Moreover, in this case, both expressions had the same number of loop iterations, and *exactly* all the padding was consumed in the computation.

In practice however, what we studied above corresponds to a single step of the PDE solver. Assume the same scenario as above, except that the solver actually runs this step two times. For simplicity, we generalize expr to a function step such that, for any array A, $\mathsf{step}(A) = \mathsf{expr}[\mathsf{Arr} := A]$. Two sequential executions of step would then be written as $\mathsf{step}^2(A)$.

According to Proposition 8, we have $\mathsf{speed}_0(\mathsf{expr}) = (1, 1)$. Thus, for the padding to last two steps and thus avoid padding exhaustion before the end of the full computation, we need to pad the 0^{th} axis of A m_l times on the left and m_r times on the right, where m_l and m_r are given by:

$$(m_l, m_r) = 2 \times \mathsf{speed}_0(\mathsf{expr}) = 2 \times (1, 1) = (2, 2).$$

We start again by creating an array Arr'' such that

$$\mathsf{Arr}'' = \mathsf{dpadl}_0^2(\mathsf{dpadr}_0^2(\mathsf{Arr}')).$$

From Definition 12, we have that

$$\rho_{\mathrm{ann}^+}(\mathsf{Arr}'') = \left\langle 14^{2,2,5}\ 4 \right\rangle$$

and

$$\mathsf{Arr}'' = \left(\begin{array}{cccc} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{array}\right).$$

We create an array $\mathsf{Arr}^{++} = \mathsf{liftp}(\mathsf{Arr}'')$. From Definition 10:

$$\rho_{\mathsf{ann}+}(\mathsf{Arr}^{++}) = \left\langle 2, 7^{2,5}, 4 \right\rangle$$

and

$$\mathsf{Arr}^{++}_{\langle 0 \rangle} = \left(\begin{array}{cccc} 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{array}\right),$$

$$\mathsf{Arr}^{++}_{\langle 1 \rangle} = \left(\begin{array}{cccc} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ \hline 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 \\ \hline 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{array}\right).$$

Once again, using Proposition 17 and Proposition 18, we get

$$\mathsf{step}^2(\mathsf{Arr}) = \mathsf{dpadr}_0^{-2}(\mathsf{dpadl}_0^{-2}(\mathsf{liftp}_0^{-1}(\mathsf{step}^2(\mathsf{Arr}^{++})))).$$

We can now transform the resulting expression expr to ONF for each machine. The bounds of i and j are once again given by Proposition 8. Thus, for $c \in \{0, 1\}$, we have the following:

$$\begin{aligned} &\forall i \in \{i \in \mathsf{Fin}\,7 \;:\; 1 \le i < 6\}, \\ &\quad \langle i \rangle \;\psi\; ((1\;\theta_0\;\mathsf{Arr}^{++}_{\langle c \rangle}) + (-1\;\theta_0\;\mathsf{Arr}^{++}_{\langle c \rangle})) \\ &\equiv (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[\gamma(\langle i+1 \rangle ; \langle 7 \rangle) \times 4 + \iota 4] + \\ &\quad (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[\gamma(\langle i-1 \rangle ; \langle 7 \rangle) \times 4 + \iota 4]. \end{aligned}$$

We one again apply γ, rav and turn ι into a loop, and we get the following generic program:

$$\begin{aligned} &\forall i \in \{i \in \mathsf{Fin}\,7 \;:\; 1 \le i < 6\}, j \in \mathsf{Fin}\,4, \\ &\quad (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[(i+1) \times 4 + j] + \\ &\quad (\mathsf{rav}\,\mathsf{Arr}^{++}_{\langle c \rangle})[(i-1) \times 4 + j]. \end{aligned}$$

At that point, we can rewrite our expression as such:

$$\begin{aligned} &\mathsf{step}^2(\mathsf{Arr}) \\ &= \mathsf{dpadr}_0^{-2}(\mathsf{dpadl}_0^{-2}(\mathsf{liftp}_0^{-1}(\mathsf{step}^2(\mathsf{Arr}^{++})))) \\ &= \mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{step}(\\ &\quad \mathsf{liftp}_0(\mathsf{dpadr}_0^{-1}(\mathsf{dpadl}_0^{-1}(\mathsf{liftp}_0^{-1}(\mathsf{step}(\mathsf{Arr}^{++}))))))))). \end{aligned}$$

But here, as given by Proposition 14, applying

$$\mathsf{liftp}_0 \circ \mathsf{dpadr}_0^{-1} \circ \mathsf{dpadl}_0^{-1} \circ \mathsf{liftp}_0^{-1}$$

to $\mathsf{step}(\mathsf{Arr}^{++})$ is equivalent to applying padr_0^{-1} and padl_0^{-1} once to both $\mathsf{Arr}^{++}_{\langle 0 \rangle}$ and $\mathsf{Arr}^{++}_{\langle 1 \rangle}$. As a result, for $c \in \{0, 1\}$,

$$\rho_{\mathsf{ann}+}(\mathsf{padr}_0^{-1}(\mathsf{padl}_0^{-1}(\mathsf{Arr}^{++}_{\langle c \rangle}))) = \left\langle 5^{1,4}, 4 \right\rangle.$$

The rest of the computation follows the single step distributed case presented above. Note that in this case four intermediate rows of the result were computed twice (once on each machine), resulting in four additional outer loop iterations compared to the equivalent single machine unpadded two-step case. Thus, getting rid of inter-process communication involved both data redundancy and duplicated calculations. Whether performing calculations several times instead of exchanging states between different computation loci is beneficial, must be determined based on hardware-dependent cost functions.

Table 2. Execution time (in seconds) of a 3-dimensional PDE solver C implementation compiled with GCC 8.2.0 with different padding parameters on a single core. CPU 1: Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz; CPU 2: AMD EPYC 7601 32-Core; CPU 3: Intel(R) Xeon(R) Silver 4112 CPU @ 2.60GHz; and CPU 4: ThunderX2 CN9980. The code of the experiments is at https://github.com/mathematics-of-arrays/padding-in-the-mathematics-of-arrays.

	No padding	Padding axis 2	Padding axis 3
CPU 1	225.74	168.59	167.84
CPU 2	299.42	119.61	120.12
CPU 3	172.71	160.51	192.70
CPU 4	660.53	347.47	357.32

6 Experiments

We extended the scenario depicted in Section 5.1 to our implementation of a PDE solver using a $(-1, 0, 1)$ stencil along every axis; that is, at every derivation step the left and right padding operation are applied once along the specified axis. The memory overhead of such padding is roughly 0.4% in our example. The execution times of 50 derivation steps given different padding parameters are gathered in Table 2.

We see a performance improvement on CPU 1, 2, and 4 between the original code in which no padding was applied and the cases with padding on either axis. The difference is particularly striking on CPU 2 and 4, where the program runs roughly twice as fast when padding is applied.

This large difference seems to indicate that padding allows the compiler to perform better optimizations on these architectures. This analysis is corroborated by the output of *perf stat*: on CPU 2 and 4, the runs without padding execute 2 to 2.5 times as many instructions as their padded counterparts.

On CPU 3, padding the last axis makes execution slower. Looking at the output of *perf stat* tells us that both padded programs execute roughly 87% as many instructions as the unpadded one. When the last axis is padded, the number of executed instructions per cycle (IPC) drops to 81%. That run should last roughly $\frac{87}{81} = 1.07$ times as long as the unpadded one. This is in line with our measurements. One possible explanation is that GCC does not properly take into account the costs of the instructions involved in the computation.

The number of instructions run on CPU 1 and CPU 3 are close. However, the drop in IPC is much smaller on CPU 1, resulting in a slight performance improvement.

It is hard to quantify the impact of the padding on data locality and cache line usage. The percentage of measured cache misses in all the programs is roughly the same for all three runs for a given architecture.

These experiments further confirm the need for a vehicle for easy exploration of codes that implement different memory layouts.

Further work is needed to explore tiling in this setting. This is because tiling requires reorganizing data within arrays using *transpose*, which we did not study here.

7 Conclusion

We showed that MoA provides the required building blocks to discuss padding as well as data distribution given an arbitrary architecture. It is thus well-suited to explore the space of optimal computations for array expressions at a high level of abstraction. Along the way, we built two examples demonstrating exactly how to use these notions to optimize stencil computations. Our approach could be implemented as a compiler optimization to automatically rewrite array expressions based on hardware and known operational costs. We expect future work to focus both on better quantifying the benefit of using this approach instead of existing solutions and on implementing MoA and its properties using proof assistants. For the latter, effort is already underway at https://github.com/mathematics-of-arrays/moa-formalization.

Acknowledgments

We thank Jonathan Prieto-Cubides and Wrya Kadir for their helpful feedback on early drafts of the paper. We also give our thanks to Jeremy Gibbons and our anonymous reviewers for their constructive and insightful comments of our paper. The research presented in this paper has benefited from the Experiment Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053.

References

[1] Philip Samuel Abrams. 1970. *An APL machine*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA.

[2] Ole Jørgen Abusdal. 2020. *Transformations for Array programming*. Master's thesis. The University of Bergen. https://doi.org/10.13140/RG.2.2.33970.73923

[3] Eva Burrows, Helmer André Friis, and Magne Haveraaen. 2018. An Array API for Finite Difference Methods. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (Philadelphia, PA, USA) *(ARRAY 2018)*. ACM, New York, NY, USA, 59–66. https://doi.org/10.1145/3219753.3219761

[4] Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. 2019. Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming* (Phoenix, AZ, USA) *(ARRAY 2019)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/3315454.3329954

[5] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. High Performance Stencil Code Generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO 2018)*. ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[6] Harry B. Hunt III, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds. 2008. A Transformation–Based Approach for the Design of Parallel/Distributed Scientific Software: the FFT. *CoRR* abs/0811.2535 (2008). arXiv:0811.2535 http://arxiv.org/abs/0811.2535

[7] K. E. Iverson. 1962. *A Programming Language*. Wiley, New York.

[8] Fredrik Berg Kjolstad and Marc Snir. 2010. Ghost Cell Pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns* (Carefree, Arizona, USA) *(ParaPLoP '10)*. Association for Computing Machinery, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/1953611.1953615

[9] Daniel Lemire, Owen Kaser, and Nathan Kurz. 2019. Faster remainder by direct computation: Applications to compilers and software libraries. *Softw., Pract. Exper.* 49, 6 (2019), 953–970. https://doi.org/10.1002/spe.2689

[10] Lenore Mullin. 1988. *A Mathematics of Arrays*. Ph.D. Dissertation. Syracuse University.

[11] Lenore M. R. Mullin and Michael A. Jenkins. 1996. Effective data parallel computation using the Psi calculus. *Concurrency - Practice and Experience* 8, 7 (1996), 499–515. https://doi.org/10.1002/(SICI)1096-9128(199609)8:7<499::AID-CPE230>3.0.CO;2-1

[12] Paul Chang and Lenore R. Mullin. 2002. *An Optimized QR Factorization Algorithm based on a Calculus of Indexing*. Technical Report. MIT Lincoln Laboratory. https://doi.org/10.13140/2.1.4938.2722

[13] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. Lift: A Functional Data-Parallel IR for High-Performance GPU Code Generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 74–85. https://doi.org/10.1109/CGO.2017.7863730

[14] 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)*. Association for Computing Machinery, New York, NY, USA, 69–79. https://doi.org/10.1145/3315454.3329960

[15] Artjoms Šinkarovs. 2015. *Data Layout Types: a type-based approach to automatic data layout transformations for improved SIMD vectorisation*. Ph.D. Dissertation. School of Mathematical and Computer Sciences, Heriot-Watt University, Heriot-Watt University, Edinburgh Campus, Edinburgh, Scotland, EH14 4AS. https://www.ros.hw.ac.uk/handle/10399/2880