Finite Difference Methods Fengshui: Alignment through a Mathematics of Arrays

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

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

Ole Abusdal
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
ole.abusdal@student.uib.no

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

Jaakko Järvi
University of Bergen
Department of Informatics
Bergen, Hordaland, Norway
jaakko.jarvi@uib.no

Sandra Macià
Barcelona Supercomputing Center (BSC - CNS)
Barcelona, Spain
sandra.macia@bsc.es

Abstract

Numerous scientific-computational domains make use of array data. The core computing of the numerical methods and the algorithms involved is related to multi-dimensional array manipulation. Memory layout and the access patterns of that data are crucial to the optimal performance of the array-based computations. As we move towards exascale computing, writing portable code for efficient data parallel computations is increasingly requiring an abstract productive working environment. To that end, we present the design of a framework for optimizing scientific array-based computations, building a case study for a Partial Differential Equations solver. By embedding the Mathematics of Arrays formalism in the Magnolia programming language, we assemble a software stack capable of abstracting the continuous high-level application layer from the discrete formulation of the collective array-based numerical methods and algorithms and the final detailed low-level code. The case study lays the groundwork for achieving optimized memory layout and efficient computations while preserving a stable abstraction layer independent of underlying algorithms and changes in the architecture.

CCS Concepts • **Software and its engineering → Software design engineering**;

Keywords Mathematics of Arrays, Finite Difference Methods, Partial Differential Equations, Psi calculus, Magnolia

1 Introduction

Given an address space, the data layout and the pattern of accessing that data are fundamental for the efficient exploitation of the underlying computer architecture. The access pattern is determined by a numerical algorithm, which may have been tuned to produce a particular pattern. The data layout may have to be adjusted explicitly to a given pattern and the computer hardware architecture. At the same time, high-performance environments are evolving rapidly and are subject to many changes. Moreover, numerical methods and algorithms are traditionally embedded in the application, forcing rewrites at every change. Thus the efficiency and portability of applications are becoming problematic. Under this scenario, software or hardware modifications usually lead to a tedious work of rewriting and tuning throughout which one must ensure correctness and efficiency. To face this scenario, the scientific community suggests a separation of concerns through high-level abstraction layers.

Burrows et al. identified a Multiarray API for Finite Difference Method (FDM) solvers [8]. We investigate the fragment of the Mathematics of Arrays (MoA) formalism [22, 23] that corresponds to this API. MoA gives us the ψ-calculus for optimizing such solvers. We present a full system approach from high level coordinate-free Partial Differential Equations (PDEs) to preparing for the layout of data and code optimization, using the MoA as an intermediate layer and the Magnolia programming language [5] to explore the specifications. In this framework, a clean and natural separation occurs between application code, the optimization algorithm and the underlying hardware architecture, while providing verifiable components. We fully work out a specific test case that demonstrates an automatable way to optimize the data layout and access patterns for a given architecture in the case of FDM solvers for PDE systems. We then proceed to show that our chosen fragment of the rewriting system defined by the ψ-calculus makes up a canonical rewriting subsystem, i.e. one that is both strongly normalizing and confluent.

In the proposed system, algorithms are written against a stable abstraction layer, independent of the underlying numerical methods and changes in the architecture. Tuning for performance is still necessary for the efficient exploitation of different computer architectures, but it takes place below this abstraction layer without disturbing the high-level implementation of the algorithms.

This paper is structured as follows. Section 2 presents the related work, and a concise literature review of the state of the art. Section 3 introduces the general software stack composition and design used for our purposes. Section 4 details the optimizations and transformation rules. The PDE solver test case showcasing the framework is presented in Section 5. Finally, conclusions are given in Section 6.

2 Related work

Whole-array operations were introduced by Ken Iverson [18] in the APL programming language, an implementation of his notation to model an idealized programming language with a universal algebra. Ten years later, shapes were introduced to index these operations by Abrams [1]. Attempts to compile and verify APL proved unsuccessful due to numerous anomalies in the algebra [34]. Specifically, $\iota\sigma$ was equivalent to $\iota\langle\ \sigma\ \rangle$, where σ is a scalar and $\langle\ \sigma\ \rangle$ is a one element vector. Moreover, there was no indexing function nor the ability to obtain all indices from an array's shape. This caused Perlis to conclude the idealized algebra should be a Functional Array Calculator based on the λ-calculus [34]. Even with this, no calculus of indexing was formulated until the introduction of MoA [22]. MoA can serve as a foundation for array/tensor operations and their optimization.

Numerous languages emerged with monolithic or whole-array operations. Some were interpreted (e.g. Matlab and Python), some were compiled (e.g. Fortran90 and TACO [19]) and some were Object Oriented with pre-processing capabilities (e.g. C++ with expression templates [9, 32]). Current tensor (array) frameworks in contemporary languages, such as Tensorflow [33] and Tensor Toolbox [4] provide powerful environments to model tensor computations. None of these frameworks are based on the ψ-calculus.

Existing compilers have various optimizations that can be formulated in the ψ-calculus, e.g. loop fusion (equivalent to distributing indexing of scalar operations in MoA) and loop unrolling (equivalent to collapsing indexing based on the properties of ψ and the ψ-correspondence Theorem (PCT) in MoA [23]). Many of the languages mentioned above implement concepts somewhat corresponding to MoA's concept of shape and its indexing mechanism. It is, however, the properties of the ψ-calculus and its ability to obtain a Denotational Normal Form (DNF) for any computation that make it particularly well-suited for optimization.

Hagedorn et al. [13] pursued the goal of optimizing stencil computations using rewriting rules in LIFT.

3 Background, design and technologies

We present the design of our library-based approach structured by layers. Figure 1 illustrates this abstract generic environment. At the domain abstraction layer, code is written in the integrated specification and programming language

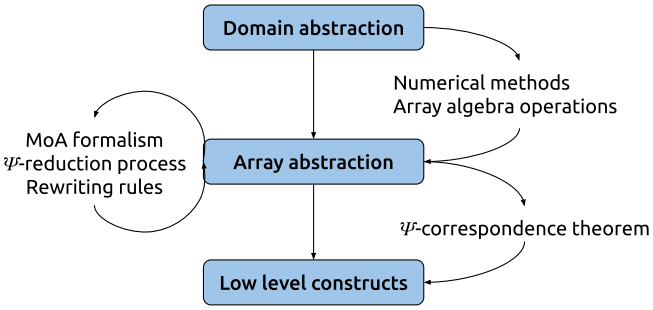

Figure 1. Layer abstraction design; generic environment approach.

Magnolia, a language designed to support a high level of abstraction, ease of reasoning, and robustness. At the intermediate level, the MoA formalism describes multi-dimensional arrays. Finally, through the ψ-correspondence theorem, the array abstraction layer is mapped to the final low-level code.

3.1 Magnolia

Magnolia is a programming language geared towards the exploration of algebraic specifications. It is being designed at the Bergen Language Design Laboratory [5]; it is a work in progress and is used to teach the Program Specification class at the University of Bergen, Norway. Magnolia's strength relies in its straightforward way of working with abstract constructs.

Magnolia relies primarily on the *concept* module, which is a list of type and function declarations (commonly called a *signature*) constrained by *axioms*. In Magnolia, an *axiom* defines properties that are assumed to hold; it however differs from the usual axioms in mathematics in that an axiom in Magnolia may define derived properties. Functions and axioms may be given a *guard*, which defines a precondition. The *satisfaction* module serves to augment our knowledge with properties that can be deduced from the premises, typically formatted to indicate that a *concept* models another one.

Magnolia is unusual as a programming language in that it does not have any built-in type or operation, requiring that everything be defined explicitly. Magnolia is transpiled to other languages, and thus, the actual types the programmer intends to use when running their program must be defined in the target language.

3.2 Mathematics of Arrays

MoA [22, 23] is an algebra for representing and describing operations on arrays. The main feature of the MoA formalism is the distinction between the *DNF*, which describes an array by its shape together with a function that defines the value at every index, and the *Operational Normal Form* (ONF), which

describes it on the level of memory layout. The MoA's ψ-calculus [23] provides a formalism for index manipulation within an array, as well as techniques to reduce expressions of array operations to the DNF and then transform them to ONF.

The ψ-calculus is based on a generalized array indexing function, ψ, which selects a partition of an array by a multi-dimensional index. Because all the array operations in the MoA algebra are defined using shapes, represented as a list of sizes, and ψ, the reduction semantics of ψ-calculus allow us to reduce complex array computations to basic indexing/selection operations, which reduces the need for any intermediate values.

By the ψ-correspondence theorem [23], we are able to transform an expression in DNF to its equivalent ONF, which describes the result in terms of loops and controls, *starts*, *strides* and *lengths* dependent on the chosen linear arrangement of the items, e.g. based on hardware requirements.

3.2.1 Motivation behind DNF and ONF

The goal behind the DNF and the ONF is to create an idealized foundation to define most — if not all — domains that use tensors (arrays). Using MoA, all of the transformations to the DNF can be derived from the definition of the ψ function and shapes.

This view has a long history [1] and, when augmented by the λ-calculus [6], provides an idealized semantic core for all arrays [26, 27]. Array computations are very prevalent. A recent Dagstuhl workshop [2, 3] reported the pervasiveness of tensors in the Internet of things, Machine Learning, and Artificial Intelligence (e.g. Kronecker [24]) and Matrix Products [11]. Moreover, they dominate science [12, 21] in general, especially signal processing [25, 28, 30, 31] and communications [29].

3.3 PDE solver framework

Figure 2 illustrates the design structured by layers for the PDE solver framework we describe. The first abstraction layer defines the problem through the domain's concepts. At this level, PDEs are expressed using collective and continuous operations to relate the physical fields involved. Through the functions encapsulating the numerical methods, the high-level continuous abstraction is mapped to a discrete array-based layer. A Magnolia specification of the array algebra defined by the MoA formalism and the ψ-calculus has been developed at this level. This algebra for arithmetic operations and permutations over multi-dimensional arrays defines the problem through collective array operations in a layout independent manner. At this point, array manipulation functions and operations may be defined in the MoA formalism and reduced according to the ψ-reduction process. This process simplifies an expression through transformational and compositional reduction properties: the rewriting rules. From the user's array-abstracted expression we obtain an equivalent

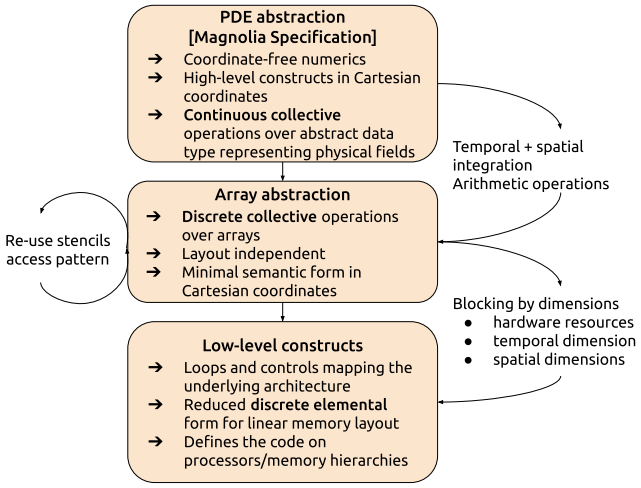

Figure 2. Layer abstraction design; detailed environment designed for a PDE solver.

optimal and minimal semantic form. Finally, the indexing algebra of the ψ-calculus relates the monolithic operations to elemental operations, defining the code on processors and memory hierarchies through loops and controls. The ψ-correspondence theorem is the theorem defining the mapping from the high-level abstracted array expressions to the operational expressions, i.e. from a form involving Cartesian coordinates into one involving linear arranged memory accesses.

4 MoA transformation rules

4.1 ψ-calculus and reduction to DNF

Multiarrays, or multidimensional arrays, have a shape given by a list of sizes $\langle s_0 \ \ldots \ s_{n-1} \rangle$. For example, a 6 by 8 matrix A has the shape $\langle 6 \ 8 \rangle$. The index for a multiarray is given by a multi-index $\langle i_0 \ \ldots \ i_{n-1} \rangle$. For position j of the multi-index, the index i_j is in the range $0 \leq i_j < s_j$. This sets the vocabulary for talking about multiarrays. In the following Magnolia code and in the rest of the paper, we will assume that the following types are declared:

- **type** `MA`, for Multiarrays;
- **type** `MS`, for Multishapes;
- **type** `MI`, for Multi-indexes;
- **type** `Int`, for Integers.

All these types will have (mapped) arithmetic operators. Important functions on a multiarray are:

- the shape function ρ, which returns the shape of a multiarray, e.g. $\rho A = \langle 6 \ 8 \rangle$;
- the ψ function, which takes a submulti-index and returns a submultiarray, e.g. $\langle\rangle \psi A = A$ and $\rho(\langle 3 \rangle \psi A) = \langle 8 \rangle$ is the subarray at position 3;

- the rotate function θ, which rotates the multiarray: $p\ \theta_x\ A$ denotes the rotation of A by offset p along axis x (rotate does not change the shape: $\rho(p\ \theta_x\ A) = \rho A$).

With respect to ψ, rotate works as:

$$\langle\ i_0\ \ldots\ i_x\ \rangle\ \psi\ (p\ \theta_0\ A) \quad = \quad \langle\ (i_0 + p)\ \bmod\ s_0\ \ldots\ i_x\ \rangle\ \psi\ A$$

The rotate operation can be used to calculate, for each element, the sum of the elements in the adjacent columns, $(1\ \theta_0\ A) + ((-1)\ \theta_0\ A)$, which is a multiarray with the same shape as A. Applying ψ to the expression gives the following reduction:

$$\langle i_0\rangle\ \psi\ ((1\ \theta_0\ A) +\ ((-1)\ \theta_0\ A)) = \langle (i_0 + 1)\ \bmod\ s_0\rangle\ \psi\ A + \langle (i_0 - 1)\ \bmod\ s_0\rangle\ \psi\ A$$

These above MOA functions can be declared in Magnolia, with axioms stating their properties.

```
/** Extract the shape of an array. */
function rho(a:MA) : MS;
/** Extract subarray of an array. */
function psi(a:MA, mi:MI) : MA;
/** Rotate distance p along axis. */
function rotate(a:MA, axis:Int, p:Int) : MA ;
axiom rotateShape(a:MA, ax:Int, p:Int) {
  var ra = rotate(a,ax,p);
  assert rho(ra) == rho(a);
}
axiom rotatePsi(a:MA, ax:Int, p:Int, mi:MI) {
  var ra = rotate(a,ax,p);
  var ij = pmod(get(mi,ax)+p,get(rho(a),ax));
  var mj = change(mi,ax,ij);
  assert psi(ra,mi) == psi(a,mj);
}
axiom plusPsi(a:MA, b:MA, mi:MI)
  guard rho(a) == rho(b) {
  assert rho(a+b) == rho(a);
  assert psi(a+b,mi) == psi(a,mi) + psi(b,mi);
}
```

Note how we are using ρ and ψ to define operations on multiarrays. The ρ operator keeps track of the resulting shape. The ψ operator takes a partial multi-index and explains the effect of the operation on the subarrays. In this way the ψ operator moves inward in the expression, pushing the computation outwards towards subarrays and eventually to the element level. The concatenation property for ψ-indexing is important for this,

$$\langle\ j\ \rangle\ \psi\ (\langle\ i\ \rangle\ \psi\ A) \equiv \langle\ i\ j\ \rangle\ \psi\ A.$$

```
axiom psiConcatenation(ma:MA, q:MI, r:MI) {
    var psiComp = psi( psi( ma,q ), r );
    var psiCat = psi( ma, cat( q,r ) );
    assert psiComp == psiCat;
}
```

The rules above, for rotation and arithmetic, show how ψ moves inwards towards the multiarray variables. When this process stops, we have reached the DNF. All other multiarray functions have then been removed and replaced by their ψ definitions. What is left to figure out and what we will tentatively in this paper is how to build the DNF.

Burrows et al [8] made the case that the operations defined above augmented with mapped arithmetic constitute a sufficient basis to work with any FDM solver of PDE systems. It does not matter what language the original expression comes from (Python, Matlab, Fortran, C, etc). With the syntax removed and the tokens expressed as an AST, the DNF denotes the reduced semantic tree and could be returned to the syntax of the originating language, with interpretation or compilation proceeding as usual.

4.2 Transformation rules

The MoA defines many rewriting rules in order to reduce an expression to its DNF. Working with those, we got the insight that the goal of the reduction is to move the call to ψ inwards to apply it as early as possible in order to save computations, and that there are enough rules to allow us to move ψ across any type of operation (Multiarray on Multiarray, scalar on Multiarray).

For the sake of this particular example, we limited ourselves to a subset of the transformation rules in the MoA. We show that this constitutes a rewriting system that is canonical.

Let us first introduce the rules we are using. In the rules, the metavariables $index_i$, u_i and sc_i respectively denote multi-indexes, multiarrays and scalars. The metavariable op is used for mappable binary operations such as $\times$, $+$ and $-$, that take either a scalar and a multiarray or two multiarrays as parameters and return a multiarray.

$$\frac{index\ \psi\ (u_i\ op\ u_j)}{(index\ \psi\ u_i)\ op\ (index\ \psi\ u_j)}\ \text{R1}$$

$$\frac{index\ \psi\ (sc\ op\ u)}{sc\ op\ (index\ \psi\ u)}\ \text{R2}$$

$$\frac{k \geq i \implies \langle\ sc_0\ \ldots\ sc_i\ \ldots\ sc_k\ \rangle\ \psi\ (sc\ \theta_i\ u)}{\langle\ sc_0\ \ldots\ ((sc_i + sc)\ mod\ (\rho\ u)[i])\ \ldots\ sc_k\ \rangle\ \psi\ u}\ \text{R3}$$

Proving that a rewriting system is canonical requires proving two properties [20]:

1. the rewriting system must be confluent;
2. the rewriting system must be strongly normalizing (reducible in a finite number of steps).

For a rewriting system, being confluent is the same as having the Church-Rosser property [20], i.e. in the case when reduction rules overlap so that a term can be rewritten in more than one way, the result of applying any of the overlapping rules can be further reduced to the same result. If

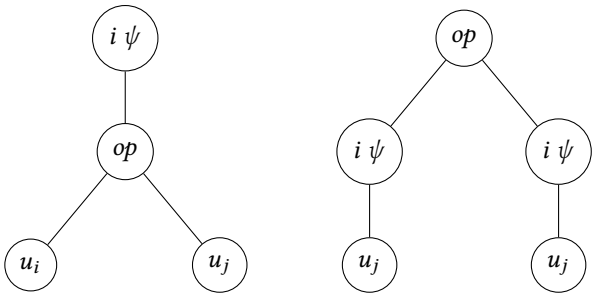

Figure 3. Rule 1 and its application.

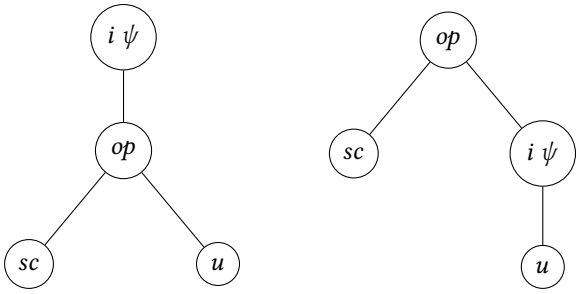

Figure 4. Rule 2 and its application.

a term can be derived into two different terms, the pair of the two derived terms is called a critical pair. Proving that a rewriting system is confluent is equivalent to proving that every critical pair of the system yields the same result for both of its terms.

Our rules above of the rewriting system can not generate any critical pair; the system is thus trivially confluent.

Now, we must prove that the rewriting system is strongly normalizing: the system must yield an irreducible expression in a finite number of steps for any expression. To that end, we assign a weight $w \in \mathbb{N}$ to the expression such that w represents the "weight" of the expression tree. We define the weight of the tree as the sum of the weight of each $(index\ \psi)$ node. The weight of each one of these nodes is equal to 3^h, where h is the height of the node.

Since $\mathbb{N}$ is bounded below by 0, we simply need to prove that the application of each rule results in w strictly decreasing to prove that our rewriting system is strongly normalizing.

For each one of our three rules, we draw a pair of trees representing the corresponding starting expression on the left and the resulting expression from applying the rule on the right. Then, we verify that w strictly decreases from the tree on the left to the tree on the right. We call w_l the weight of the left tree and w_r the weight of the right tree. Figures 3, 4 and 5 illustrate these trees.

In the three figures, we assume that the tree rooted in the $i\psi$ node has height h'. Since the $i\psi$ node has a parameter, it is never a leaf and we have $h' > 0$.

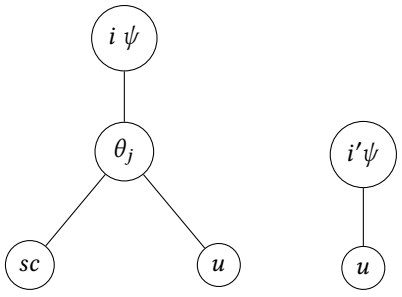

Figure 5. Rule 3 and its application.

In Figure 3, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R1$, however, has the weight $w_r = 2 \times 3^{h'-1} = \frac{2}{3} w_l$, which is less than w_l. In Figure 4, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R2$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3} w_l$, which is less than w_l. In Figure 5, the starting expression has the weight $w_l = 3^{h'}$. The resulting expression from applying $R3$, however, has the weight $w_r = 3^{h'-1} = \frac{1}{3} w_l$, which is less than w_l.

Since w strictly decreases with every rewrite, the system is strongly normalizing. Since it is also confluent, it is canonical.

4.3 Adapting to hardware architecture using ONF

Once we have reduced an expression to its DNF, if we know about the layout of the data it uses, we can build its ONF. Assuming a row major layout, let us turn
$\langle\ i\ \rangle\ \psi\ ((1\ \theta_0\ A) + ((-1)\ \theta_0\ A))$ into its ONF.
To proceed further, we need to define three functions: γ, ι and rav.

- rav is short for Ravel, which denotes the *flattening* operation, both in APL and in MoA. It takes a multi-array and reshapes it into a vector. We therefore use rav to deal with the representation of the array in the memory of the computer.
- γ takes an index and a shape and returns the corresponding index in the flattened representation of the array[1]. γ is not computable unless a specific memory layout is assumed, which is why this decision has to be taken before building the ONF.
  One can note that rav and γ are tightly connected in defining flattened array accesses as γ encodes the layout while rav is defined in terms of γ. For FDM, it is important therefore to figure out the right memory layout such that rotations are completed in an efficient fashion.
- ι is a unary function, which takes a natural number n as its parameter and returns a 1-D array containing

[1]Here, only γ on rows is considered, but other γ functions exist

the range of natural numbers from 0 to n excluded. It is used to build *strides* of indexes needed by the ONF.

With these operations defined, we can proceed. We first apply the ψ-correspondence theorem followed by applying γ.

$$\begin{aligned}
&\forall i \;\; s.t. \;\; 0 \leq i < 6 \\
&\quad \langle\, i \,\rangle\; \psi\; ((1\ \theta_0\ A) + ((-1)\ \theta_0\ A)) \\
&\qquad \equiv (rav\ A)[\gamma(\langle\, (i+1)\ mod\ 6 \,\rangle\ ;\ \langle\, 6 \,\rangle) \times 8 + \imath 8] + \\
&\qquad\quad (rav\ A)[\gamma(\langle\, (i-1)\ mod\ 6 \,\rangle\ ;\ \langle\, 6 \,\rangle) \times 8 + \imath 8] \\
&\qquad \equiv (rav\ A)[((i+1)\ mod\ 6) \times 8 + \imath 8] + \\
&\qquad\quad (rav\ A)[((i-1)\ mod\ 6) \times 8 + \imath 8]
\end{aligned}$$

Secondly, we apply rav and turn $\imath$ into a loop to reach the following generic program:

$$\begin{aligned}
&\forall j \;\; s.t. \;\; 0 \leq j < 8 \\
&\quad A[((i+1)\ mod\ 6) \times 8 + j] + \\
&\quad A[((i-1)\ mod\ 6) \times 8 + j]
\end{aligned}$$

The ONF is concerned with performance, and is where cost analysis and *dimension lifting* begins.

Regarding pure cost analysis, at this point, it is still possible to optimize this program: unfolding the loops gives us the insight that the modulo operation is only ever useful on the 0^{th} and 5^{th} row. Thus, by splitting the cases into those that require the modulo operation to be run and those that do not, we may achieve better performance.

Now imagine breaking the problem over 2 processors. Conceptually, the dimension is lifted. It is important to note that the lifting may happen on any axis, especially in the current case where we are dealing with rotations on a given axis. If we happen to apply dimension lifting on the axis on which we are rotating, we may not be able to split the memory perfectly between the different computing sites. This could require inter-process communication, or duplication of memory.

In this case, since we are rotating on the 0^{th} axis, we pick axis 1 as the candidate to be lifted. The loop on j is then split into 2 loops because we now view the 2-D resultant array as a 3-D array A' with shape $\langle\, 6\ 2\ 8/2 \,\rangle = \langle\, 6\ 2\ 4 \,\rangle$ in which axis 1 corresponds to the number of processors. Therefore, we get:

$$\begin{aligned}
&\forall i, j \;\; s.t.\; 0 \leq i < 6, \;\; 0 \leq j < 2 \\
&\quad \langle\, i\ j \,\rangle\; \psi\; ((1\ \theta_0\ A') + ((-1)\ \theta_0\ A')) \\
&\qquad \equiv (rav\ A')[\gamma(\langle\, ((i+1)\ mod\ 6)\ j \,\rangle\ ;\ \langle\, 6\ 2 \,\rangle) \times 4 + \imath 4] + \\
&\qquad\quad (rav\ A')[\gamma(\langle\, ((i-1)\ mod\ 6)\ j \,\rangle\ ;\ \langle\, 6\ 2 \,\rangle) \times 4 + \imath 4] \\
&\qquad \equiv (rav\ A')[(((i+1)\ mod\ 6) \times 2 + j) \times 4 + \imath 4] + \\
&\qquad\quad (rav\ A')[(((i-1)\ mod\ 6) \times 2 + j) \times 4 + \imath 4]
\end{aligned}$$

This reduces to the following generic program:

$$\begin{aligned}
&\forall k \;\; s.t. \;\; 0 \leq k < 4 \\
&\quad A'[((i+1)\ mod\ 6) \times 4 \times 2 + j \times 4 + k] + \\
&\quad A'[((i-1)\ mod\ 6) \times 4 \times 2 + j \times 4 + k]
\end{aligned}$$

As discussed above, there are other ways to achieve splitting of the problem across several computing sites. In general, the size of the array and the cost of accessing different architectural components drive the decision to break the problem up over processors, GPUs, threads, etc. [16, 17].

If a decision was made to break up the operations over different calculation units, the loop would be the same but the cost of performing the operation would be different. This decision is therefore completely cost-driven.

Continuing with *dimension lifting*, a choice might be made to use vector registers. This is, once again, a cost-driven decision, which may however be decided upon statically, prior to execution.

If we were to break our problem up over several processors and using vector registers, it would conceptually go from 2 dimensional to 4 dimensional, using indexing to access each resource. The same process can be applied to hardware components [11], e.g. pipelines, memories, buffers, etc., to achieve optimal throughput.

5 PDE solver test-case

Coordinate-free numerics [10, 14] is a high-level approach to writing solvers for PDEs. Solvers are written using high-level operators on abstract tensors. Take for instance Burgers' equation [7],

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u},$$

where vector $\vec{u}$ denotes a time and space varying velocity vector, t is time, and the scalar ν is a viscosity coefficient. Burgers' equation is a PDE involving temporal ($\frac{\partial}{\partial t}$) and spatial ($\nabla$) derivative operations. Applying an explicit second order Runge-Kutta time-integration method, the coordinate-free time-integrated equation can be coded in Magnolia as follows.

```
procedure burgersTimestep
(upd u:Tensor1V, obs dt:R, obs nu:R) = {
  var u_t = nu * laplacian(u) )
          - dot(u,gradient(u));
  var u_substep = u + dt/2 * u_t;
  u_t = nu * laplacian(u_substep)
      - dot(u_substep,gradient(u_substep));
  u = u + dt * u_t;
};
```

Note how close this code follows the mathematical high-level formulation (5). We can lower the abstraction level of this code by linking it with a library for 3D cartesian coordinates based on continuous ringfields [15]. Next it can be linked with a library for finite difference methods choosing, e.g., stencils $\langle -\frac{1}{2}, 0, \frac{1}{2} \rangle$ and $\langle 1, -2, 1 \rangle$ for first and second order partial derivatives, respectively. This takes us to a code at

the MoA level, consisting of rotate and maps of arithmetic operations [8]. With some reorganisation, we end up with the solver code below, expressed using MoA. The code calls the `snippet` six times forming one full time integration step, one call for each of the three dimensions of the problem times two due to the half-step in the time-integration. The variables `dt,nu,dx` are scalar (floating point). The first two come from the code above, while `dx` was introducd by the finite difference method. The variables `u0,u1,u2` are multiarrays (3D each), for each of the components of the 3D velocity vectorfield. These variables will be updated during the computation. The variables `c0,c1,c2,c3` and `c4` are numeric constants. Three temporary multiarray variables `v0,v1,v2` are computed in the first three `snippet` calls, due to the half-step. They are then used in the last three `snippet` calls to update `u0,u1,u2`.

```
procedure step
(upd u0:MA, upd u1:MA, upd u2:MA,
 obs nu:Float, obs dx:Float, obs dt:Float) {

  var c0 = 0.5/dx;
  var c1 = 1/dx/dx;
  var c2 = 2/dx/dx;
  var c3 = nu;
  var c4 = dt/2;

  var v0 = u0;
  var v1 = u1;
  var v2 = u2;
  call snippet(v0,u0,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v1,u1,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(v2,u2,u0,u1,u2,c0,c1,c2,c3,c4);
  call snippet(u0,v0,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u1,v1,v0,v1,v2,c0,c1,c2,c3,c4);
  call snippet(u2,v2,v0,v1,v2,c0,c1,c2,c3,c4);
};
```

In the actual `snippet` code, `d1a,d2a,d1b,d2b,d1c,d2c` and `shift_v` are temporary multiarray variables. The `shift` function takes as first argument the multiarray being shifted, then the direction of the shift, and lastly the distance for the rotational shift.

```
procedure snippet
(upd u:MA, obs v:MA,
 obs u0:MA, obs u1:MA, obs u2:MA,
 obs c0:Float, obs c1:Float, obs c2:Float,
 obs c3:Float, obs c4:Float) {

  var shift_v = shift ( v, 0, -1 );
  var d1a = -c0 * shift_v;
  var d2a = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 0, 1 );
  d1a = d1a + c0 * shift_v;
  d2a = d2a + c1 * shift_v;

  shift_v = shift ( v, 1, -1 );
  var d1b = -c0 * shift_v;
  var d2b = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 1, 1 );
  d1b = d1b + c0 * shift_v;
  d2b = d2b + c1 * shift_v;

  shift_v = shift ( v, 2, -1 );
  var d1c = -c0 * shift_v;
  var d2c = c1 * shift_v - c2 * u0;
  shift_v = shift ( v, 2, 1 );
  d1c = d1c + c0 * shift_v;
  d2c = d2c + c1 * shift_v;

  d1a = u0 * d1a + u1 * d1b + u2 * d1c;
  d2a = d2a + d2b + d2c;
  u = u + c4 * ( c3 * d2a - d1a);
};
```

In essence, `snippet` is computing 1/3 of the half-step of the PDE, using common calls to rotate to compute one first and one second order partial derivative.

5.1 Reduction using MoA

Using the reduction rules defined in the ψ-calculus, and turning our `snippet` code into an expression, we can reduce the code to a DNF representation. In the following, we spell out some of the transformation steps. The equation

$$
\begin{aligned}
snippet = u + c_4 \times \\
&(c_3 \times (c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v)\ + ((-1)\ \theta_1\ v) + \\
&(1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) - 3c_2u_0)\ - c_0 \times \\
&(((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 + \\
&((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1 + \\
&((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2))
\end{aligned}
$$

is a transcription of the `snippet` code above.

We use the notation θ_x to denote a rotation around the x^{th} axis, represented in Magnolia by calls to `shift(multiarray, axis, offset)`.

The Magnolia implementation of the snippet makes heavy use of the multiarrays `d1x` and `d2x`, where `x` denotes the axis around which the multiarray is rotated in lexicographical order (`a` corresponds to the 0^{th} axis, `b` to the 1^{st} and so on). For the sake of easing into it, let us start by building a generic DNF representation for `d2x`. All the steps will be detailed explicitly in order to gain insights on what is needed and what is possible.

$$\langle\ i\ j\ k\ \rangle\ \psi\ d_{2x} = \langle\ i\ j\ k\ \rangle\ \psi\ (c_1 \times ((-1)\ \theta_x\ v) + c_1 \times (1\ \theta_x\ v) - c_2 \times u_0)$$

(distribute ψ over +/-)

$$= \langle\ i\ j\ k\ \rangle\ \psi\ (c_1 \times ((-1)\ \theta_x\ v))\ + \langle\ i\ j\ k\ \rangle\ \psi\ (c_1 \times (1\ \theta_x\ v))\ - \langle\ i\ j\ k\ \rangle\ \psi\ (c_2 \times u_0)$$

(extract constant factors)

$$= c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_x\ v)) + c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi(1\ \theta_x\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0)$$

(factorize by c_1)

$$= c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_x\ v)\ + \langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_x\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0)$$

Using the MoA's concatenation of index property, we can now define $\langle\ i\ \rangle\ \psi\ d_{2x}$. However, this is only reducible if $x = 0$. The reason is that to reduce an expression using a rotation on the x^{th} axis further, one needs to apply ψ with an index of at least $x + 1$ elements. Therefore, to reduce d_{21}, we need an index vector with at least 2 elements, while we need a total index containing 3 elements to reduce d_{22}. With that in mind, we can try to reduce d_{21}:

$$\langle\ i\ j\ \rangle\ \psi\ d_{21} = c_1 \times (\langle\ i\ j\ \rangle\ \psi((-1)\ \theta_1\ v)\ + \langle\ i\ j\ \rangle\ \psi\ (1\ \theta_1\ v)) - c_2 \times (\langle\ i\ j\ \rangle\ \psi\ u_0)$$

(reducing rotation)

$$= c_1 \times (\langle\ i\ ((j-1)\ mod\ s_1)\ \rangle\ \psi\ v + \langle\ i\ ((j+1)\ mod\ s_1)\ \rangle\ \psi\ v) - c_2 \times (\langle\ i\ j\ \rangle\ \psi\ u_0)$$

For $x = 2$, we apply the same process with a total index:

$$\langle\ i\ j\ k\ \rangle\ \psi\ d_{22} = c_1 \times (\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v) + \langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v)) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0)$$

(reducing rotation)

$$= c_1 \times (\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ \psi\ v + \langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ \psi\ v) - c_2 \times (\langle\ i\ j\ k\ \rangle\ \psi\ u_0)$$

Now we can define the ONF of the expression, which is the form we will use in our actual code. Let's define it for d_{21}:

$$(rav\ d_{21})[\gamma(\langle\ i\ j\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \imath s_2] = c_1 \times ((rav\ v)[\gamma(\langle\ i\ ((j-1)\ mod\ s_1)\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \imath s_2] + (rav\ v)[\gamma(\langle\ i\ ((j+1)\ mod\ s_1)\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \imath s_2]) - c_2 \times (rav\ u_0)[\gamma(\langle\ i\ j\ \rangle\ ;\ \langle\ s_0\ s_1\ \rangle) \times s_2 + \imath s_2]$$

(apply γ on both sides)

$$(rav\ d_{21})[(i \times s_1 \times s_2 + j \times s_2 + \imath s_2] = c_1 \times ((rav\ v)[i \times s_1 \times s_2 + ((j-1)\ mod\ s_1) \times s_2 + \imath s_2] + (rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + \imath s_2]) - c_2 \times (rav\ u_0)[i \times s_1 \times s_2 + j \times s_2 + \imath s_2]$$

The optimization can be done similarly for d_{22}. The fact that d_{22} can only be reduced using a total index means that *snippet* too can only be fully reduced using a total index.

$$\begin{aligned}
&\langle\ i\ j\ k\ \rangle\ \psi\ snippet \\
&\quad = \langle\ i\ j\ k\ \rangle\ \psi\ (u + c_4 \times (c_3 \times (c_1 \times \\
&\quad (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) + ((-1)\ \theta_1\ v) + \\
&\quad (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) - \\
&\quad 3c_2u_0)\ - c_0(((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 + \\
&\quad ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1\ + ((1\ \theta_2\ v) + \\
&\quad ((-1)\ \theta_2\ v))\ u_2)))
\end{aligned}$$

(distribute ψ over + and -)

$$\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + (\langle\ i\ j\ k\ \rangle\ \psi\ c_4 \times (c_3 \times \\
&\quad (c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) + \\
&\quad ((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + \\
&\quad (1\ \theta_2\ v)) - 3c_2u_0)))\ - \langle\ i\ j\ k\ \rangle\ \psi \\
&\quad (c_0 \times (((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 + \\
&\quad ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1\ + \\
&\quad ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2)))
\end{aligned}$$

(extract constant c_4, c_3, and c_0)

$$\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (\langle\ i\ j\ k\ \rangle\ \psi \\
&\quad (c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) + \\
&\quad ((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + \\
&\quad ((-1)\ \theta_2\ v) + (1\ \theta_2\ v)) - 3c_2u_0)) - \\
&\quad c_0 \times (\langle\ i\ j\ k\ \rangle\ \psi \\
&\quad (((1\ \theta_0\ v) - ((-1)\ \theta_0\ v))\ u_0 + \\
&\quad ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v))\ u_1\ + \\
&\quad ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v))\ u_2)))
\end{aligned}$$

(distribute ψ over +, ×, and -)

$$\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (\langle\ i\ j\ k\ \rangle\ \psi \\
&\quad (c_1 \times (((-1)\ \theta_0\ v) + (1\ \theta_0\ v)\ +
\end{aligned}$$

$$
\begin{aligned}
&((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + \\
&((-1)\ \theta_2\ v) + (1\ \theta_2\ v))) - \\
&\langle\ i\ j\ k\ \rangle\ \psi\ (3c_2u_0)) - c_0\times \\
&((\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_0\ v) - ((-1)\ \theta_0\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_0 + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_1 + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_2))
\end{aligned}
$$

(extract constant factors c_1 and $3 \times c_2$)

$$
\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times \\
&(\langle\ i\ j\ k\ \rangle\ \psi\ (((-1)\ \theta_0\ v) + (1\ \theta_0\ v) + \\
&((-1)\ \theta_1\ v) + (1\ \theta_1\ v) + ((-1)\ \theta_2\ v) + \\
&(1\ \theta_2\ v))) - 3c_2(\langle\ i\ j\ k\ \rangle\ \psi\ u_0))\ - c_0 \times \\
&((\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_0\ v) - ((-1)\ \theta_0\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_0 + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_1\ v) - ((-1)\ \theta_1\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_1 + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((1\ \theta_2\ v) - ((-1)\ \theta_2\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_2))
\end{aligned}
$$

(distribute ψ over + and -)

$$
\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times \\
&(\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_0\ v) + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_0\ v) + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_1\ v) + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_1\ v) + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v) + \\
&\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v)) - 3c_2 \\
&(\langle\ i\ j\ k\ \rangle\ \psi\ u_0))\ - c_0 \times \\
&((\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_0\ v) - \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_0\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_0 + \\
&(\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_1\ v) - \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_1\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_1 + \\
&(\langle\ i\ j\ k\ \rangle\ \psi\ (1\ \theta_2\ v) - \\
&\langle\ i\ j\ k\ \rangle\ \psi\ ((-1)\ \theta_2\ v)) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_2))
\end{aligned}
$$

(translate rotations into indexing)

$$
\begin{aligned}
&= \langle\ i\ j\ k\ \rangle\ \psi\ u + c_4 \times (c_3 \times (c_1 \times \\
&(\langle\ ((i-1)\ mod\ s_0)\ j\ k\ \rangle\ \psi\ v + \\
&\langle\ ((i+1)\ mod\ s_0)\ j\ k\ \rangle\ \psi\ v + \\
&\langle\ i\ ((j-1)\ mod\ s_1)\ k\ \rangle\ \psi\ v + \\
&\langle\ i\ ((j+1)\ mod\ s_1)\ k\ \rangle\ \psi\ v + \\
&\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ \psi\ v + \\
&\langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ \psi\ v) - \\
&3c_2(\langle\ i\ j\ k\ \rangle\ \psi\ u_0))\ - c_0 \times \\
&((\langle\ ((i+1)\ mod\ s_0)\ j\ k\ \rangle\ \psi\ v - \\
&\langle\ ((i-1)\ mod\ s_0)\ j\ k\ \rangle\ \psi\ v) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_0 + \\
&(\langle\ i\ ((j+1)\ mod\ s_1)\ k\ \rangle\ \psi\ v - \\
&\langle\ i\ ((j-1)\ mod\ s_1)\ k\ \rangle\ \psi\ v) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_1 + \\
&(\langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ \psi\ v - \\
&\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ \psi\ v) \times \\
&\langle\ i\ j\ k\ \rangle\ \psi\ u_2))
\end{aligned}
$$

In Magnolia, the DNF can be captured as such:

```
procedure snippetDNF(
upd u:MA, obs v:MA,
obs u0:MA, obs u1:MA, obs u2:MA,
obs c0:Float, obs c1:Float,
obs c2:Float, obs c3:Float, obs c4:Float,
obs mi:MI) {

  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);

  u =
  psi(mi,u) + c4*(c3*(c1*(
  psi(mod0(mi-d0,s0),v) +
  psi(mod0(mi+d0,s0),v) +
  psi(mod1(mi-d1,s1),v) +
  psi(mod1(mi+d1,s1),v) +
  psi(mod2(mi-d2,s2),v) +
  psi(mod2(mi+d2,s2))) - 3*c2* psi(mi,u0)) -
  c0 * ((psi(mod0(mi+d0,s0),v) -
  psi(mod0(mi-d0,s0),v)) * psi(mi,u0) + (
  psi(mod1(mi+d1,s1),v) -
  psi(mod1(mi-d1,s1),v)) * psi(mi,u1) + (
  psi(mod2(mi+d2,s2),v) -
  psi(mod2(mi-d2,s2),v)) * psi(mi,u2) ));
}
```

Now, we can transform `snippet` into its ONF form:

$$(rav\ snippet)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] =$$

$$
\begin{aligned}
&(rav\ u)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + c_4 \times (c_3 \times (c_1 \times \\
&(rav\ v)[\gamma(\langle\ ((i-1)\ mod\ s_0)\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + \\
&(rav\ v)[\gamma(\langle\ ((i+1)\ mod\ s_0)\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + \\
&(rav\ v)[\gamma(\langle\ i\ ((j-1)\ mod\ s_1)\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + \\
&(rav\ v)[\gamma(\langle\ i\ ((j+1)\ mod\ s_1)\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + \\
&(rav\ v)[\gamma(\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] + \\
&(rav\ v)[\gamma(\langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]) - \\
&3c_2(rav\ u)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]\ - c_0 \times \\
&(((rav\ v)[\gamma(\langle\ ((i+1)\ mod\ s_0)\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] - \\
&(rav\ v)[\gamma(\langle\ ((i-1)\ mod\ s_0)\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]) \times \\
&(rav\ u_0)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]\ + \\
&((rav\ v)[\gamma(\langle\ i\ ((j+1)\ mod\ s_1)\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] - \\
&(rav\ v)[\gamma(\langle\ i\ ((j+1)\ mod\ s_1)\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]) \times \\
&(rav\ u_1)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]\ + \\
&((rav\ v)[\gamma(\langle\ i\ j\ ((k+1)\ mod\ s_2)\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)] - \\
&(rav\ v)[\gamma(\langle\ i\ j\ ((k-1)\ mod\ s_2)\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]) \times \\
&(rav\ u_2)[\gamma(\langle\ i\ j\ k\ \rangle\ ;\ \langle\ s_0\ s_1\ s_2\ \rangle)]))
\end{aligned}
$$

This is how far we can go without specific information about the layout of the data in the memory and the architecture. The current form is still fully generic, with γ and rav parameterized over the layout. The Magnolia implementation of this generic form is as follows:

```
procedure moaONF (
upd u:MA,
obs v:MA,
obs u0:MA, obs u1:MA, obs u2:MA,
obs c0:Float, obs c1:Float,
obs c2:Float, obs c3:Float, obs c4:Float,
obs mi:MI ){

  var s0 = shape0(v);
  var s1 = shape1(v);
  var s2 = shape2(v);

  var newu =
  get(rav(u),gamma(mi,s)) + c4*(c3*(c_1*
  get(rav(v),gamma(mod0(mi-d0,s0),s)) +
  get(rav(v),gamma(mod0(mi+d0,s0),s)) +
  get(rav(v),gamma(mod1(mi-d1,s1),s)) +
  get(rav(v),gamma(mod1(mi+d1,s1),s)) +
  get(rav(v),gamma(mod2(mi-d2,s2),s)) +
  get(rav(v),gamma(mod2(mi+d2,s2),s))) -
  3 * c_2 get(rav(u),gamma(mi,s)) - c_0 *
  ((get(rav(v),gamma(mod0(mi+d0,s0),s)) -
  get(rav(v),gamma(mod0(mi-d0,s0),s))) *
  get(rav(u_0),gamma(mi,s)) +
  (get(rav(v),gamma(mod1(mi+d1,s1),s)) -
  get(rav(v),gamma(mod1(mi-d1,s1),s))) *
  get(rav(u_1),gamma(mi,s)) +
  (get(rav(v),gamma(mod2(mi+d2,s2),s)) -
  get(rav(v),gamma(mod2(mi-d2,s2),s))) *
  get(rav(u_2),gamma(mi,s))));

  set(rav(u),gamma(mi,s),newu);
}
```

In Section 4.3, we defined the layout of the data as row-major. Thus we can optimize the expression further by expanding the calls to γ:

$$
\begin{aligned}
&(rav\ snippet)[i \times s_1 \times s_2 + j \times s_2 + k] = \\
&(rav\ u)[i \times s_1 \times s_2 + j \times s_2 + k] + c_4 \times (c_3 \times (c_1 \times \\
&(rav\ v)[((i-1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
&(rav\ v)[((i+1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] + \\
&(rav\ v)[i \times s_1 \times s_2 + ((j-1)\ mod\ s_1) \times s_2 + k] + \\
&(rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + k] + \\
&(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1)\ mod\ s_2)] + \\
&(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1)\ mod\ s_2)]) - \\
&3c_2(rav\ u)[i \times s_1 \times s_2 + j \times s_2 + k]\ - c_0 \times \\
&(((rav\ v)[((i+1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k] - \\
&(rav\ v)[((i-1)\ mod\ s_0) \times s_1 \times s_2 + j \times s_2 + k]) \times \\
&(rav\ u_0)[i \times s_1 \times s_2 + j \times s_2 + k]\ + \\
&((rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + k] - \\
&(rav\ v)[i \times s_1 \times s_2 + ((j+1)\ mod\ s_1) \times s_2 + k]) \times \\
&(rav\ u_1)[i \times s_1 \times s_2 + j \times s_2 + k]\ + \\
&((rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k+1)\ mod\ s_2)] - \\
&(rav\ v)[i \times s_1 \times s_2 + j \times s_2 + ((k-1)\ mod\ s_2)]) \times \\
&(rav\ u_2)[i \times s_1 \times s_2 + j \times s_2 + k]))
\end{aligned}
$$

At this point, as indicated in section 4.3, we can convert our expression into several subexpressions in order to distinguish the general case from anomalies (i.e cases that require the modulo operation to be applied on any axis). This general case is in ONF and we can use it for code generation or to perform additional transformations, specifically dimension lifting.

6 Conclusion

Through the full analysis of an FDM solver of a PDE, we were able to extract a rewriting subsystem most relevant to our specific problem out of the rewriting rules provided by the ψ-calculus. Then, we proved that this particular set of rewriting rules constitutes a canonical rewriting system, getting one step closer to fully automating the optimization of array computations using the MoA formalism.

We are now working on the implementation of our optimizations to measure their impact on the performance of

the solver for different architectures, and can report results in the near future.

By working out an approach from high level coordinate-free PDEs down to preparing for data layout and code optimization using MoA as an intermediate layer through the full exploration of a relevant example, we pave the way for building similar systems for any problem of the same category. High-efficiency code can thus easily be explored and generated from a unique high-level abstraction and potentially different implementation algorithms, layouts of data or hardware architectures.

Because tensors dominate a significant portion of science, future work may focus on figuring out what properties can be deduced from the complete ψ-calculus rewriting system with a goal to extend this currently problem-oriented approach towards a fully automated problem-independent optimization tool based on MoA.

Given the scale of the ecosystem impacted by this kind of work, such prospects are very attractive.

References

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

[2] Evrim Acar, Animashree Anandkumar, Lenore Mullin, Sebnem Rusitschka, and Volker Tresp. 2016. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Reports* 6, 4 (2016), 57–79. https://doi.org/10.4230/DagRep.6.4.57

[3] Evrim Acar, Animashree Anandkumar, Lenore Mullin, Sebnem Rusitschka, and Volker Tresp. 2018. Tensor Computing for Internet of Things (Dagstuhl Perspectives Workshop 16152). *Dagstuhl Manifestos* 7, 1 (2018), 52–68. https://doi.org/10.4230/DagMan.7.1.52

[4] Brett W. Bader, Tamara G. Kolda, et al. 2015. MATLAB Tensor Toolbox Version 2.6. Available online. http://www.sandia.gov/~tgkolda/TensorToolbox/

[5] Anya Helene Bagge. 2009. *Constructs & Concepts: Language Design for Flexibility and Reliability.* Ph.D. Dissertation. Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway. http://www.ii.uib.no/~anya/phd/

[6] Klaus Berkling. 1990. *Arrays and the Lambda Calculus.* Technical Report 93. Electrical Engineering and Computer Science Technical Reports.

[7] Johannes Martinus Burgers. 1948. A mathematical model illustrating the theory of turbulence. In *Advances in applied mechanics.* Vol. 1. Elsevier, 171–199.

[8] 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 (ARRAY 2018).* ACM, New York, NY, USA, 59–66. https://doi.org/10.1145/3219753.3219761

[9] J. A. Crotinger et al. 2000. Generic Programming in POOMA and PETE. *Lecture Notes in Computer Science* 1766 (2000).

[10] Philip W. Grant, Magne Haveraaen, and Michael F. Webster. 2000. Coordinate free programming of computational fluid dynamics problems. *Scientific Programming* 8, 4 (2000), 211–230. https://doi.org/10.1155/2000/419840

[11] Ian Grout and Lenore Mullin. 2018. Hardware Considerations for Tensor Implementation and Analysis Using the Field Programmable Gate Array. *Electronics* 7, 11 (2018). https://doi.org/10.3390/electronics7110320

[12] John L. Gustafson and Lenore M. Mullin. 2017. Tensors Come of Age: Why the AI Revolution will help HPC. *CoRR* abs/1709.09108 (2017). arXiv:1709.09108 http://arxiv.org/abs/1709.09108

[13] 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 (CGO 2018).* ACM, New York, NY, USA, 100–112. https://doi.org/10.1145/3168824

[14] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. 1999. Formal Software Engineering for Computational Modelling. *Nord. J. Comput.* 6, 3 (1999), 241–270.

[15] Magne Haveraaen, Helmer André Friis, and Hans Munthe-Kaas. 2005. Computable Scalar Fields: a basis for PDE software. *Journal of Logic and Algebraic Programming* 65, 1 (September-October 2005), 36–49. https://doi.org/10.1016/j.jlap.2004.12.001

[16] H. B. Hunt III, L. Mullin, and D. J. Rosenkrantz. 1998. *Experimental Design and Development of a Polyalgorithm for the FFT.* Technical Report 98–5. University at Albany, Department of Computer Science.

[17] 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

[18] K. Iverson. 1962. *A Programming Language.* John Wiley and Sons, Inc. New York.

[19] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[20] J. W. Klop, Marc Bezem, and R. C. De Vrijer (Eds.). 2001. *Term Rewriting Systems.* Cambridge University Press, New York, NY, USA.

[21] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (September 2009), 455–500. https://doi.org/10.1137/07070111X

[22] Lenore Mullin. 1988. *A Mathematics of Arrays.* Ph.D. Dissertation.

[23] Lenore Mullin and Michael Jenkins. 1996. Effective Data Parallel Computation Using the Psi-Calculus. *Concurrency Journal* (1996).

[24] L Mullin and J Raynolds. 2014. *Scalable, Portable, Verifiable Kronecker Products on Multi-scale Computers.* Constraint Programming and Decision Making. Studies in Computational Intelligence, Vol. 539. Springer, Cham.

[25] L. Mullin, E. Rutledge, and R. Bond. 2002. Monolithic Compiler Experiments using C++ Expression Templates. In *Proceedings of the High Performance Embedded Computing Workshop (HPEC 2002).* MIT Lincoln Lab, Lexington, MA.

[26] Lenore M. Restifo Mullin, Ashok Krishnamurthi, and Deepa Iyengar. 1988. The Design And Development of a Basis, alpha$_L$, for Formal Functional Programming Languages with Arrays Based on a Mathematics of Arrays. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software.*

[27] L.M. R. Mullin. 1991. Psi, the Indexing Function: A Basis for FFP with Arrays. In *Arrays, Functional Languages, and Parallel Systems.* Kluwer Academic Publishers.

[28] Lenore R. Mullin. 2005. A uniform way of reasoning about array-based computation in radar: Algebraically connecting the hardware/software boundary. *Digital Signal Processing* 15, 5 (2005), 466–520.

[29] L. R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. 1993. Formal Methods for Scheduling, Routing and Communication Protoc ol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2).* IEEE Computer Society.

[30] L. R. Mullin, D. J. Rosenkrantz, H. B. Hunt III, and X. Luo. 2003. Efficient Radar Processing Via Array and Index Algebras. In *Proceedings First Workshop on Optimizations for DSP and Embedded Systems (ODES).* San Francisco, CA, 1–12.

[31] Paul Chang and Lenore R. Mullin. 2002. An Optimized QR Factorization Algorithm based on a Calculus of Indexing. DOI: 10.13140/2.1.4938.2722.

[32] Jeremy G. Siek and Andrew Lumsdaine. 1998. The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra. In *Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=646894.709706

[33] Google Brain Team. 2015. https://www.tensorflow.org/.

[34] Hai-Chen Tu and Alan J. Perlis. 1986. FAC: A Functional APL Language. *IEEE Software* 3, 1 (Jan. 1986), 36–45.