University of Bergen
Department of Informatics

Exploring Hardware Agnostic Multiarrays in Magnolia

Author: Marius Kleppe Larnøy

Supervisors: Magne Haveraaen

Bergen Language Design Laboratory

August, 2022

Abstract

We present a specification and implementation of a generic multiarray API based on A Mathematics of Arrays in the general purpose research language Magnolia. We show how we can lift the reasoning on arrays to a more abstract level, and how this enables us to precisely manipulate arrays independent of hardware memory layouts.

Acknowledgements

This thesis has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX^3), which is financially supported by the Research Council of Norway under contract 270053.

I want to thank my supervisor Magne Haveraaen for his guidance on this thesis, and for facilitating an environment for interesting discussions on generic programming and formal methods through both the Magnolia work group and BLDL. Thank you to my collegues, and especially Benjamin Chetioui for endless feedback and support on my work. Lastly I want to thank my family for continued support throughout this process.

Marius Kleppe Larnøy
Tuesday 2nd August, 2022

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Gordon Moore postulated in the 1960s that the number of transistors in a processing unit would double every two years [14]. This postulate largely holds true as we enter the 2020s, with computing power reaching exascale levels (10^{18} FLOPS) in 2018. As hardware continues to evolve we are reliant on software capable of adapting to both current and future architectures, whilst remaining maintainable.

Fields of both research and industry that deal with large volumes of data often utilize HPC – i.e. supercomputers or clusters – to process and perform calculations efficiently. This creates the need for software capable of leveraging distributed architectures, whilst remaining maintainable. MoA [40] is a calculus for working with arrays, generalizing the notion of an array to the concepts of shapes and dimensions. A big motivation behind creating this calculus was how arrays are mapped down to hardware, and how we can rearrange and manipulate the arrays independently of memory layout without losing the ability to target specific architectures.

In this thesis we will explore the MoA calculus, using the generic programming language Magnolia [2, 5] as our vehicle to implement a generic array API based on MoA. We will observe how MoA allows us to manipulate arrays on a hardware independent level without compromising neither performance or supported hardware.

1.1 Motivations

There are numerous domains in science and industry that rely on discretizations of formally defined physics models. These models are reliant on numerical solutions in order to be applicable to real-world problems, and discretization of PDEs in order to be able to compute finite solutions. A good example is the Navier-Stokes equations, which are used to model the behavior of fluids, e.g. wind and water. In fields such as meteorology and industry sectors such as wind farms, fast and accurate modeling of wind and water is essential. Work by BLDL at the University of Bergen has contributed to the idea that mapping array expressions that can run efficiently on arbitrary hardware is worth exploring. A recurring case study from this research is how to use arrays to compute numerical solutions to PDEs on different hardware, potentially bringing together domain experts in fields such as meteorology who are looking for both faster and more portable ways to simulate data independent of current computing power.

1.2 Contribution

This thesis is comprised out of existing theory in conjunction with the authors own work on applications. Mainly, A Mathematics of Arrays [40] is the work of Lenore Mullin, with subsequent publications on MoA being the work of Mullin and her co-authors. More recent literature on MoA [6, 7, 9] is the work of researchers associated with BLDL in collaboration with Mullin. Additionally, Magnolia is a research language under active development at BLDL. Among the related works are two compiler implementations [2, 5].

What this thesis aims to do is to explore the MoA calculus through the lens of formal specifications. We establish a baseline of understanding of multiarrays and Magnolia, and then we present a specification and implementation for a subset of MoA.

1.3 Thesis Outline

The thesis is structured as follows:

- Chapter 2 introduces the relevant parts of the MoA theory,
- Chapter 3 introduces the Magnolia programming language
- Chapter 4 explores arrays in convensional programming languages, and we motivate the problem at hand by highlighting a relevant domain where efficient array computations are important,

- Chapter 5 describes an implementation of a subset of MoA in the Magnolia programming language,
- Chapter 6 consists of a collaborative article highlighting current work on being carried out at BLDL, and a closer look at an experimental implementation in CUDA,
- Chapter 7 rounds off the thesis with discussion and reflection on both the work that has been done and future work.

Chapter 2

A Mathematics of Arrays

2.1 Introduction

MoA is a theoretical framework for multidimensional arrays, defining them by the notion of their shape. Its inception was the PhD thesis of Lenore Mullin in 1988 [40], and she has been the driving force behind promoting MoA and its applications in parallel computing [16], HPC [15] etc.

The original presentation of MoA draws heavy inspiration from Ken Iverson and APL [25], both in its approach to defining arrays and its notational style. More recent publications depart from the APL roots of the theory [1, 6, 7], focusing instead on its ideas of creating dense array expressions operating on single multiarrays. In this chapter we present an overview of the core theory, first as presented in the original papers and then as given in recent literature. We then draw comparisons between the two approaches.

2.2 The Original MoA Approach

Following from APL where the centerpiece data type is the multidimensional array, MoA revolves around a single array type. Unary operators and infix binary operations are used without any specified operator precedence, and parentheses are used to dictate order of application. Expressions associate implicitly to the right, following from APL.

2.2.1 Terminology

Every array has a *dimensionality*, often denoted by an integer superscript. E.g. ξ^3 is a 3-dimensional array. An arrays *shape* denotes the length of each of its dimensions, collected in a 1-dimensional array. E.g. ξ^3 would have a shape of the form $\langle i\ j\ k \rangle$, where $i, j, k \in \mathbb{N}^+$.

Some arrays with a specific dimensionality are more commonly used than others, and as such they are given unique names.

- A *scalar* refers to a 0-dimensional array, i.e. an array with zero dimensions and an empty shape. The literature denotes the empty scalar as σ.
- A *vector* in MoA is a 1-dimensional array of elements with ordered integer indices. In the literature, the empty vector is denoted Θ, and vectors in general are denoted with the typical arrow notation. E.g. $\vec{v} = \langle 1\ 2\ 3 \rangle$ is a vector with 3 elements.
- A *matrix* is a 2-dimensional array.

2.2.2 Unary Operations for Array Shapes

- δ (Delta): takes an array and returns its dimensions as a scalar. For scalar arguments $\delta\sigma = 0$.
- ρ (Rho): takes an array and returns its shape as a vector. In particular $\rho\sigma = \Theta$.
- τ (Tau): takes an array returns the total number of objects in the array as a scalar. $\tau\sigma = 1$.
  Example: if we have a vector $\vec{v} = \langle 1\ 2\ 3 \rangle, \tau\vec{v} = 3$.
- ι (iota): takes as argument a scalar $\sigma \in \mathbb{N}$ and generates a vector containing the integer sequence $0 \ldots (\sigma - 1)$. For $\sigma = 0, \iota 0 \equiv \Theta$.

2.2.3 Indexing

Indexing can be done a few different ways. $\vec{v} = \langle 1\ 2\ 3 \rangle$ $\tau\vec{v} = 3$

1. Scalar indexing

$$\begin{aligned} \vec{v}[0] &= 1 \\ \vec{v}[1] &= 2 \\ \vec{v}[2] &= 3 \end{aligned}$$

2. Vector indexing, given that the components of the index vector all are valid indices for the indexed vector

$$\begin{aligned} \vec{u} &= \langle 2\ 1\ 0 \rangle \\ \tau\vec{u} &= 3 \\ \vec{v}[\vec{u}] &= \langle 3\ 2\ 1 \rangle \end{aligned}$$

2.2.4 Psi indexing

The psi indexing function ψ takes as a left argument an index vector and right argument an n-dimensional array.

We will begin defining the few special cases for ψ, and then move on to the general form. For the empty scalar σ and the empty vector Θ acts as neutral elements:

$$\begin{aligned} &\Theta\ \psi\ \sigma \equiv \sigma \\ &\Theta\ \psi\ \vec{x} \equiv \vec{x} \\ &\Theta\ \psi\ \zeta^n \equiv \zeta^n \\ 0 \le i < (\tau\vec{x}) \qquad &\langle i \rangle\ \psi\ \vec{x} \equiv \vec{x}[i] \end{aligned}$$

In general, for an index vector $\vec{i}$ satisfying the bounds[1] $0 \le^* \vec{i} <^* (\rho\zeta^n)$:

$$\vec{i}\ \psi\ \zeta^n = \zeta^n[\vec{i}[0]; \ldots; \vec{i}[n-1]]$$

[1] In her dissertation, Mullin introduces the notation $\xi_l\ R^*\ \xi_r$ to talk about constraints on indices. It is used to express the condition that $\vec{i}$ is a valid index vector. E.g. $0\ \le^*\ \vec{i}\ <^*\ (\rho\xi^n)$.

Partial indexing

Until now we have assumed indexing arguments to ψ to be total, i.e. $(\tau\vec{i}) = \delta\zeta^n = n$. When $\vec{i}$ is a total index, $\vec{i}\ \psi\ \zeta^n$ will return a scalar with an empty shape. Now we are going to introduce a partial index, also called a *short* index, i.e. an index vector that does not access precisely to the scalar level, but rather to a subarray level. We impose some restrictions on the partial index vectors, to make it play nicely in bounds of the accessed arrays.

For an index vector $\vec{j}$ and a n-dimensional array ξ^n:

$$\begin{array}{rl} \text{Given} & 0 \ \leq^* \ \vec{j} \ <^* \ ((\tau\vec{j}) \ \uparrow \ (\rho\xi^n)) \\ \text{then} & 0 \ \leq \ (\tau\vec{j}) \ \leq \ \delta(\xi^n) \end{array}$$

When $\vec{j}$ is a partial index, the shape of the indexed subarray is given as $\rho(\vec{j}\ \psi\ \xi^n) = (\tau\vec{j}) \downarrow (\rho\xi^n)$.

2.2.5 Take and Drop

We can define the ↑ (*take*) and ↓ (*drop*) operators as shorthand for indexing on the primary axis[2]. By *take* we are accessing the first n subarrays of the given array, keeping them. Conversely, *drop* will ignore the n first subarrays and keep the rest in an array. When applied to a 1-dimensional array, *take* and *drop* accesses down to the element level.

Example (1 dimensional):

$$\begin{aligned} \vec{x} &= \langle 1\ 2\ 3\ 4\ 5\ 6 \rangle \\ \tau\vec{x} &= 6 \\ \vec{x}[\langle 0\ 1\ 2 \rangle] &= 3 \uparrow \vec{x} = -3 \downarrow \vec{x} = \langle 1\ 2\ 3 \rangle \\ \vec{x}[\langle 4\ 5 \rangle] &= -2 \uparrow \vec{x} = 4 \downarrow \vec{x} = \langle 5\ 6 \rangle \end{aligned}$$

[2] In the literature the theory is usually presented in row-major fashion, and as such when we refer to the primary axis we are talking about the outmost row-axis unless specified otherwise.

Example ($\delta(A) > 1$):

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

$$0 \uparrow A = \langle 1\ 2\ 3\ 4 \rangle$$

$$0 \downarrow A = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

2.2.6 Catenate

Given two vectors $\vec{x}$ and $\vec{y}$, their (con)catenation $\vec{x} \# \vec{y}$ yields a vector by indexing them together. The resulting vector $\tau(\vec{x} \# \vec{y}) \equiv (\tau\vec{x}) + (\tau\vec{y})$.

Catenating a vector with a scalar is legal, "promoting"s the scalar to a one-element vector.

$$\tau(\vec{x} \# \sigma) \equiv (\tau\vec{x}) + 1$$

Catenation of arrays

Arrays can also be catenated on the primary axis, given that the shapes of the rest of the dimensions match. The shape of two catenated arrays is

$$((1 \uparrow (\rho\xi^n)) + (1 \uparrow (\rho\zeta^n))) \# (1 \downarrow (\rho\zeta^n))$$

Example:

$$A = \begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 9 & 10 \end{bmatrix} \quad B = \begin{bmatrix} 3 & 4 \\ 7 & 8 \\ 11 & 12 \end{bmatrix}$$

$$A \# B = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

2.2.7 Array Transformations

Here we will introduce three operations for manipulating the indexing of existing arrays: reverse, rotate and transpose.

Reverse

The unary reverse operation ϕ takes an array argument and reverses the order of the elements on the primary axis. It does not change the shape of the array.

$$\rho(\phi\xi^n) \equiv \rho\xi^n$$

For valid indices $0 \leq i < (\rho\xi^n)[0]$:

$$\langle i \rangle \ \psi \ (\phi\xi^n) \equiv \langle(\rho\xi^n)[0] - (i+1)\rangle \ \psi \ \xi^n$$

Example:

$$\rho A = \langle 3\ 4 \rangle \qquad \rho(\phi A) = \langle 3\ 4 \rangle = \rho A$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \phi A = \begin{bmatrix} 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Rotate

Rotate - $\ominus$ - takes a scalar left argument and an array on the right. $\sigma \ominus \xi^n$ shifts the order of the elements on the primary axis by σ. $\rho(\sigma \ominus \xi^n) \equiv \rho\xi^n$.

$$\sigma \ominus \xi^n \equiv \begin{cases} (\sigma \downarrow \xi^n) \ \# \ (\sigma \uparrow \xi^n), & 0 < \sigma \leq (\rho\xi^n)[0] \\ (\sigma \uparrow \xi^n) \ \# \ (\sigma \downarrow \xi^n), & -(\rho\xi^n)[0] \leq \sigma < 0 \end{cases}$$

Example:

$$\rho A = \langle 3\ 4 \rangle \qquad \rho(1 \ominus A) = \langle 3\ 4 \rangle = \rho A$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} 1 \ominus A = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 \end{bmatrix}$$

Transpose

Transpose - $\obslash$ - reverses the order of the indices. The resulting shape of the transposed array is the reversal of the original shape.

$$\rho(\obslash\xi^n) \equiv \phi(\rho\xi^n)$$

For valid index vectors $0 \leq^* \vec{i} <^* \phi(\rho\xi^n)$,

$$\vec{i}\ \psi\ (\obslash\xi^n) \equiv (\phi\vec{i})\ \psi\ \xi^n$$

Example:

$$\rho A = \langle 3\ 4 \rangle \quad \rho(\obslash A) = \langle 4\ 3 \rangle = \phi(\rho A)$$

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \obslash A = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}$$

2.2.8 Denotational Normal Form

DNF is a semantics-only, layout-agnostic normal form for array expressions, and represents the most ”cost-effective” way to perform operations on any given array. Any MoA expression can be rewritten as a DNF expression. What we want to achieve by reducing an expression to its corresponding DNF is to end up with (ideally) only terms utilizing the ψ operator, which in essence is an array access (cheap). By layout-agnostic we mean that this reduction to normal form is performed *before* any mapping to hardware takes place, meaning optimizations for specific hardware architectures will take place later, and that we don't have to worry about different layouts when performing the conversion to DNF.

ψ-reduction

ψ-reduction is the formal process of transforming an array expression stepwise into its normal form. It is a mechanical process where each operator has its defined rewrite rules. The complete list of rewrite rules were defined and published already in the 1990s, including that ψ-reduction could be performed by a computer [33]. It is the first step in using MoA for efficient array calculations.

In this section we will be discussing the semantics of ψ-reduction. The paper presented in Chapter 6 applies ψ-reduction as part of optimizing array expressions.

Shape Analysis

In order for us to perform a reduction, the shape of both the initial variables of the array expression and the partial results needs to be computed. We also need to define the valid indices (or index vectors) of the result. The valid indices for the expression can be decribed using:

$$0 \leq i < (\tau E) \text{ for scalar indices or } 0 \leq^* \vec{i} \leq^* (\rho E) \text{ for index vectors}$$

Reduction

Using valid indices as left side argument of ψ, we start out on the form

$$\langle i \rangle \ \psi \ E \text{ for scalar indices or } \vec{i} \ \psi \ E \text{ for index vectors}$$

The reduction is performed simply by applying what definitions/properties/identities that apply to the given expression. Mullin and Thibault defines a complete list of reduction rules, and shows that the reduction process is deterministic.

2.2.9 Operational Normal Form

Memory layouts in hardware are linear. What separates different systems are how they are accessed, with factors such as the number of processors and processor cores also affecting the final physical layout. Here we will introduce the ONF, a collection of functions to both transform and work with array expressions in a hardware specific context. This part of the MoA algebra is not in the scope of this thesis, but an overview is provided for completeness.

Ravel

Ravel – rav – takes an array expression and returns it as a vector, flattening its elements into a one dimensional array. rav can be performed both row-major and column-major, depending on the target memory layout.

Reshape

Reshape - $\hat{p}$. Takes a shape s and an array A and returns A with the same elements but with the shape s. I.e. $\rho(reshape(s, A)) = s$.

Gamma

To be able to map the DNF efficiently and optimally down to ONF we need to know two things. We need knowledge about the memory layout where our array is being mapped to, and we need an array expression in DNF form. We now introduce a **family** of functions γ which we can use to express the relation between an index and an offset in flat memory.

For vectors, the γ function is layout independent, as vectors are already one-dimensional and contiguous:

$$\gamma(\langle i \rangle, \vec{v}) \equiv \vec{v}[i]$$

The γ functions for for n-dimensional arrays depend on the target memory layout. We will give an example on how one can define a γ function for a row-major architecture. Given an n-dimensional array ζ^n with shape $\rho(\zeta^n) = \langle s_0 \ \ldots \ s_{n-1} \rangle$ and valid index vectors $0 \ \leq^* \ \vec{i} \ <^* \ \rho\zeta^n$, we can give the following relation:

$$\gamma_{\text{row}}(\vec{i}, \rho(\zeta^n)) = \gamma_{\text{row}}(\langle i_0 \ \ldots \ i_{n-1} \rangle, \langle s_0 \ \ldots \ s_{n-1} \rangle) \equiv i_{n-1} + s_{n-1} \times \gamma_{\text{row}}(\langle i_0 \ \ldots \ i_{n-2} \rangle, \langle s_0 \ \ldots \ s_{n-2} \rangle)$$

We are now equipped with the operations we need to manipulate arrays to fit specific memory layouts. This concludes our small introduction to the original MoA formalism.

2.3 BLDL Approach

For the last few years, effort has been put into creating a complete pipeline for using MoA in tandem with the Magnolia programming language to generate high-performing array expressions. This work has been carried out by researchers at BLDL in collaboration with Mullin, resulting in a series of papers [6, 7, 9]. A case study on creating an efficient PDE solver serves as the recurring domain of interest. The work includes important additions to the existing theory, as well as required proofs for existing theory.

Here we will briefly introduce this approach to the theory. As the two approaches constitute the same theory, we will focus on the specific contributions of the articles.

2.3.1 Canonical rewrite system

Chetioui et al. approaches MoA with a specific goal in mind, namely to define the minimal array API decribed in Burrows et al. in terms of the MoA formalism. A subset of MoA is sufficient to investigate this API, and the paper provides rewrite rules for these operations, along with proofs that the rules form a canonical rewrite system. That is, the system is is confluent, and any expression can be rewritten to its normal form in a finite number of steps.

2.3.2 Padding

Padding in MoA was introduced by Chetioui et al. as a way to introduce reduncancy in arrays. When working with high-performance computers, a limiting factor for computational efficiency is data locality. Large distributed systems might not have sufficient global memory, relying on message passing systems such as MPI to transfer data between components running parallel computations. By prepending or appending existing data to the array, one can limit the need for interaction between different processors. Chetioui et al. demonstrated significant runtime improvements for the PDE solver by padding the arrays.

2.4 Reflection on the approaches

Mullin was heavily inspired by APL when developing the original MoA theory. This is reflected in both in choice of syntax, as well as semantics. Notation used in MoA as presented by Mullin often follows directly from APL. Some semantic rules carry over as well, such as operators being implicitly associated to the right if no parentheses are provided.

The contemporary approach makes an effort to step away from the APL roots. While faithful and equivalent to the original theory, steps are made to create a more seamless transition into a Magnolia flavoured notational style. All operations are now defined consistently throughout in terms of the ψ operator, on the form $index\ \psi\ op(args)$, leaving the recursive definitions behind. There exist multiple reasons to why this notational style could be preferable.

1. The notational style is well suited for application in parallel computing. By describing the result at each index, the computation can easily be distributed between different processes.

2. Magnolia explicitly disallows recursion, so by relying on recursive definitions there would be a notational gap between implementations and the underlying theory.

3. The recent publications are focused on a limited subset of the theory, keeping notation consistent is more practical.

The approaches are also shaped by their respective goals. In the introduction of her thesis, Mullin argues that MoA enables verification of computer architecture design, building on VLSI design, i.e. design of integrated circuits. The focus of the papers produced at BLDL has been array transformations. Designing a pipeline for creating dense, hardware-independent array expressions in DNF which then can be translated to padded hardware-specific expressions well suited for distributed computing. Combined with Magnolia this creates a platform for theoretically well founded, portable array code not limited to existing hardware.

With these areas of applications in mind, it highlights the versatility of MoA as a theoretical framework for multiarrays.

Chapter 3

Magnolia

Magnolia is a research programming and specification language based on institution theory [13], with the goal of fully capturing Stepanov-style generics [10]. This type of generics is known as *genericity by property* in terms of the Gibbons taxonomy [12], which describes structures and algorithms in terms of syntactic and semantic requirements. To describe the type of genericity provided by Magnolia, Chetioui et al. coined the term *genericity by host language*. Magnolia is dependent on being parameterized by a host language because it provides no base data types or data structures, giving rise to a minor distinction.

Here we will give an introduction to the Magnolia specification and programming language.

3.1 The Magnolia Language

In Magnolia there are four top level module types. A `signature` is a collection of generic type - and function names. We can equip a signature with **axioms**, stating behavior of the defined types and functions. Signatures with axioms together form a `concept`. The `implementation` module expands on the functionality of the signature by allowing us to provide generic implementations for the defined types and functions. This is done either by providing an implementation in a backend language of choice[1], or by providing bodies to the defined functions. Types and structures expecting an external implementation are prefixed with the `require` keyword. In addition to functions, Magnolia also supports

[1] At the time of writing this thesis, C++ and Python [8] are supported backend languages

predicates and procedures. Whereas functions are immutable, procedures can modify state of its input variables depending on its mode. Input parameters to procedures must be given explicit mode declarations, which can be either **obs**, **upd** or **out**. An `implementation` fully parameterized by the backend is called a `program` module.

Introductory Example[2]

Let us look at how to specify and implement natural numbers in Magnolia as an example. We will take advantage of the fact that natural numbers with addition and multiplication form a commutative semiring.

A commutative semiring is defined as a set S with two binary operations *plus* and *mult* such that:

- (S, plus) is a commutative monoid with identity element 0
- (S, mult) is a commutative monoid with identity element 1
- *mult* distributes over *plus*
- *mult* by 0 annihilates S

First, we will define a generic commutative semiring and then we will relate it to a specific natural number implementation. This also serves as an example to show how separating generic structures from specific implementation can decrease code duplication by reusing generic code.

We will begin by specifying a basic algebraic structure, the semigroup. A semigroup is a type together with an associative binary operation. This is straight forward to formulate in Magnolia.

```
concept Semigroup = {

    type S;

    function bop(s1: S, s2: S): S;

    axiom associative(s1: S, s2: S, s3: S) {
        assert bop(bop(s1, s2), s3) == bop(s1, bop(s2, s3));
    }
}
```

Listing 3.1: Concept of a semigroup

[2] The complete example is available online under `examples/naturalnumbers` [30]

We can then expand on our semigroup concept by adding an identity element, this gives us a monoid. For a semiring to be commutative we require that its underlying monoids are commutative. By adding the commutative property to our monoid concept, we get a commutative (or abelian) monoid. Magnolia's powerful *renaming* mechanism is also in use here. Renamings allow us to give new names to any declared type or function in a module, providing great flexibility for both specializing generic structures and avoiding unintended name overlaps when importing multiple modules.

```
concept AbelianMonoid = {

    // include Semigroup, rename type S to M
    use Semigroup[S => M];

    function identity(): M;

    axiom idAxiom(m: M) {
        assert bop(identity(), m) == m;
        assert bop(m, identity()) == m;
    }
    axiom commutative(m1: M, m2: M) {
        assert bop(m1, m2) == bop(m2, m1);
    }
}
```

Listing 3.2: Concept of an abelian monoid

We can now specify our semiring by bringing in our monoid concept in scope with the appropriate renamings, and by asserting the distribution- and annihilation properties.

```
concept Semiring = {
    // Gives us + and 0
    use AbelianMonoid[bop => _+_, identity => zero];
    // Gives us * and 1
    use AbelianMonoid[bop => _*_, identity => one];

    // Multiplication distributes over addition
    axiom multDistribution(m1: M, m2: M, m3: M) {
        assert m1 * (m2 + m3) == (m1 * m2) + (m1 * m3);
        assert (m1 + m2) * m3 == (m1 * m3) + (m2 * m3);
    }

    // Annihilation of mult by zero
    axiom multAnnihilation(m: M) {
        assert m * zero() == zero();
        assert zero() * m == zero();
    }
}
```

Listing 3.3: Concept of a semiring

This concludes our specification of a generic commutative semiring, and we proceed to a concrete implementation. We must rely on externally provided types for our implementation because Magnolia does not provide any concrete types. In this example, we'll use a C++ backend.

```
// externally defined types and functions
implementation ExternalNat = external C++ base.nat {
    type Nat;

    function zero(): Nat;
    function one(): Nat;

    function add(a: Nat, b: Nat): Nat;
    function mul(a: Nat, b: Nat): Nat;
}

program NaturalNumbers = {
    use ExternalNat;
}
```

Listing 3.4: External implementation and program in Magnolia

Now we want to relate our generic specification to our concrete implementation. The `satisfaction` construct allows us to do precisely this.

```
satisfaction NaturalNumbersModelsSemiring = NaturalNumbers
    models Semiring[M => Nat,
                 zero => zero,
                  one => one,
                  _+_ => add,
                  _*_ => mul];
```

Listing 3.5: Asserting a claim that a `program` models a `concept`

This satisfaction relation `NaturalNumbersModelsSemiring` expresses that the program `NaturalNumbers` satisfies the axioms of `Semiring` with the provided type `Nat` and functions `zero`, `one`, `add` and `mul`.

If we provide a minimal backend in C++, together with a main function with some test calls, we can compile with magnoliac and check that our small specification and implementation in fact yields executable code.

```
$ ./natnum.bin
zero(): 0
add(one(), one()) = 2
```

Listing 3.6: Example output of the Natural Numbers program

3.2 Related works on Magnolia

There has been a wide range of work done in the Magnolia ecosystem since its inception.

- Bagge lays the groundwork for the concepts explored in Magnolia, as well as providing the first compiler implementation.

- Haugsbakk uses Magnolia as a vehicle to explore program transformation techniques.
- Abusdal explores the MoA calculus in the context of Magnolia similarly to this thesis, but through the lens of array transformations.
- Chetioui et al. highlights a redesigned Magnolia compiler [5], extending it to support a Python backend, and implementing a subset of the Boost Graph Library [42] in Magnolia to demonstrate how this allows for performant code in both transpiled C++ and Python from the same Magnolia source.
- Hamre provides insights on the viability and use of third-party verification software such as SMT solvers to prove or disprove satisfaction claims in Magnolia code.

Chapter 4

Arrays in other programming languages & PyWake

4.1 Arrays in programming languages

Arrays in programming are used to conveniently allocate equal parts of contiguous memory without having to refer to individual variables for each allocation. In this section we will take a short detour to look at multiarray support in some convensional programming languages.

4.1.1 C

C does not have support for multiarrays, only allowing integer indexing. While one can both create and index C arrays with multiple integer indices – e.g. a "2-dimensional" array `int array[4][4]` – this is just syntactic sugar for making 1-dimensional array manipulation easier. Array creation in C is in itself in fact syntactic sugar for creating a pointer to a location in memory. C arrays are in reality pointers to the first block of memory, and accessing elements after that is just providing an offset to the initial pointer position. This is reflected in C11 standard [24], where it describes array memory layout as contiguous. Two arrays of different dimensionality with the same elements in identical order will be represented equally in memory. C also allows for nesting of arrays, i.e. arrays of arrays.

Abusdal showcased how efficient the C compiler can be, in an example similar to this.

```c
const int one_d[4] = {1,2,3,4};
const int two_d[2][3] = {{1,2},{3,4}};

int c_indexing() {
  if(one_d[0] == two_d[0][0] &&
     one_d[1] == two_d[0][1] &&
     one_d[2] == two_d[1][0] &&
     one_d[3] == two_d[1][1])
    {return 5;}
  else
    {return 10;}
}
```

Listing 4.1: C array example

Compiling using GCC with all optimizations on, we can see that the whole comparison in the `c_indexing` function has been reduced to a single `mov` instruction.

```
<c_indexing>:
  mov $0x5, %eax
  ret
```

Listing 4.2: Assembly output of C example

4.1.2 Fortran

Fortran arrays offer much more fine-grained manipulations out of the box. It supports arithmetic operations on arrays, reducing the need to iterate through arrays as one are used to in the C school of languages. Interestingly, unlike C, Fortran does not allow for nested arrays. It is explicitly described in the Fortran 2003 standard [23] that *a **scalar** is a datum that is not an array (..) an **array** is a set of scalar data, all of the same type (..)*. Fortran also limits the number of dimensions an array can have to seven, but poses no restrictions on the number of elements each dimension can have.

Drawing once again from Abusdal, the GCC Fortran compiler can – as the C compiler – optimize out all array accesses given the right conditions.

```

function fortran_indexing() result(r)

  integer :: r
  integer, dimension(2,2,2) :: array1
  integer, dimension(8) :: array2
  array1 = reshape([1,2,3,4,5,6,7,8], [2,2,2])
  array2  = reshape([1,2,3,4,5,6,7,8], [8])

  if(array2(1) == array1(1,1,1) .and. &
     array2(2) == array1(2,1,1) .and. &
     array2(3) == array1(1,2,1) .and. &
     array2(4) == array1(2,2,1) .and. &
     array2(5) == array1(1,1,2) .and. &
     array2(6) == array1(2,1,2) .and. &
     array2(7) == array1(1,2,2) .and. &
     array2(8) == array1(2,2,2)) then
        r = 5
  else
        r = 10
  end if
end function access
```

Listing 4.3: Fortran90 array access example

As with 4.1.1, if we compile using GCC with all optimizations enabled and inspect the assembly output we can see that all array accesses has been optimized out, and we are left with a simple `mov` instruction setting the result to 5.

```
<fortran_indexing_>:
  mov $0x5, %eax
  ret
```

Listing 4.4: Assembly output of Fortran example

4.1.3 Python

Python is not known for its arrays, but rather for its lists. While arrays typically require a type to be provided, the flexibility of lists are more suited for the dynamically typed paradigm Python follows. Python lists are dynamically sized and allows for mixing of types in a single container. As with other languages, Pythons built-in lists has – while useful – largely been replaced with library provided alternatives for performance heavy computations. While the Python standard library includes an array package [11], it is scheduled for deprecation in Python 4, most likely due to third-party libraries already meeting the demand for arrays. Let us take a look at the most prolific one: NumPy.

NumPy [18] is a Python library for array and numerical computations. It has become synonymous with array- and numerical computing in the realm of Python, and is part of the foundation of libraries such as SciPy [46] and pandas [48]. It has even proved itself worthy of application in fields with demanding computational requirements, e.g. astrophysics [35].

NumPy offers a powerful API for array manipulation, and manages to achieve higher performance than usually associated with Python by leveraging optimized C code for much of its core implementation, and Fortran libraries such as OpenBLAS. Additionally, there exists extensions to NumPy designed to leverage high performance architectures such as GPUs [34], backing up the claim that there is a demand for tooling to adapt to increasing computing power. Many of the operations provided by NumPy corresponds with operations from MoA and Fortran, both in naming and behavior. This gives an indication that many array libraries – although different design choices – to a varying degree inherit some core traits from early array languages such as APL and Fortran.

Here we give a few examples of operations that show up in NumPy as well as MoA or Fortran.

```
import numpy as np
# creating a 1-dimensional array
data = np.arange(12)
> array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

# the shape
data.shape
> (12,)

# reshape it to an array with shape (3,4)
data = data.reshape((3, 4))

> array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
data.shape
> (3,4)
# indexing
data[1,2] # total
> 6
data[2] # partial
> array([ 8,  9, 10, 11])
data[-1] # negative (in bounds)
> array([ 8,  9, 10, 11])

# map
data + 2
> array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13]])
```

Listing 4.5: NumPy basic operations

4.2 Example domain: wind farm modelling with PyWake

Simulating wind in wind farms is an application area well suited for high performance array computations. Here we take a quick look on how the PyWake library combines the computational strength of NumPy with Pythons ease of use.

Calculating wind flow is far from a new field of research. The Navier-Stokes equations on the motion of viscous fluids were formulated in the first half of the 19th century. Proofs of general solutions and their uniqueness are famously one of the open questions presented as part of the Millennium Prize Problems [4]. Leveraging the power of computers to calculate wind flow proved useful, Veers exemplifies earlier application areas of this, using simulation data to analyse the aerodynamics of wind turbines.

Arrays were an obvious representation of data to explore when it came to fluid simulations. The three spatial axes of the real world can be represented by a three-dimensional array, and the problem could then be reduced to efficiently propagating updated values between a finite collection of grid points. One also saw the potential for running much of the computation in parallel, especially since the rise of GPGPU in the mid- 2000s [32, 26] with platforms such as CUDA [36] and OpenCL [43].

Large-scale wind farm design and maintenance necessitate extensive modeling of wind and ocean conditions. The wind conditions of the locations suitable for power generation are self-explanatory: you would want to build your wind farm in a location with average wind speeds high enough to generate a sufficient amount of electricity. The addition of wind turbines complicates matters considerably. The rows of turbines disrupt the wind flow, so positioning the turbines to maximize output in the face of disrupted air flow is critical.

PyWake [38] is a Python library developed and maintained by the Technical University of Denmark, used for calculating wind fields and energy production of wind farms. As an academic effort [27, 47, 44, 41], PyWake is a powerful tool capable of simulating wind conditions in wind farms on both sea (accounting for waves) and land (accounting for terrain). PyWake uses XArray [21] for its data representation, which are labeled multiarrays. XArray is in it self built on top of NumPy arrays. With NumPy arrays serving as the underlying data structures, calculating how wind propagates through the wind farm can be reduced to numerical maps on arrays.

While highly customizable to meet the requirements of domain experts, PyWake comes with a library of pre-defined models to work with. This includes real-world wind turbines that are currently in use, as well as a collection of existing wind farms with which one can experiment. PyWake is also tightly integrated with matplotlib [22], allowing for easy visualization of aspects such as AEP, wind speeds across a wind farm, and wind speeds around single turbines.

In this example we will be using the pre-defined site Horns Rev 1, which is an offshore wind farm in Denmark. We will create a SimulationResult, which is a labeled XArray containing arrays of information such as wind speeds, wind direction and power production across the site. By invoking methods on the SimulationResult, we can easily extract and visualize information, e.g. AEP.

```
import numpy as np
import py_wake

from py_wake.examples.data.hornsrev1 import Hornsrev1Site,V80, wt_x,
    ↪ wt_y, wt16_x, wt16_y
from py_wake import NOJ

# selecting type of turbine
windTurbines = V80()
# selecting site
site = Hornsrev1Site()
# NOJ is a wake model, which combines a site with a set of turbines
noj = NOJ(site,windTurbines)

# creating a simulation result with 16 turbines
sim_res = noj(wt16_x, wt16_y)
```

Listing 4.6: PyWake example

Here we give an example plot to showcase how one can easily present information of about the wind farm. Figure 4.2 gives clear information about how the power production of the inner turbines are being affected by the surrounding ones.

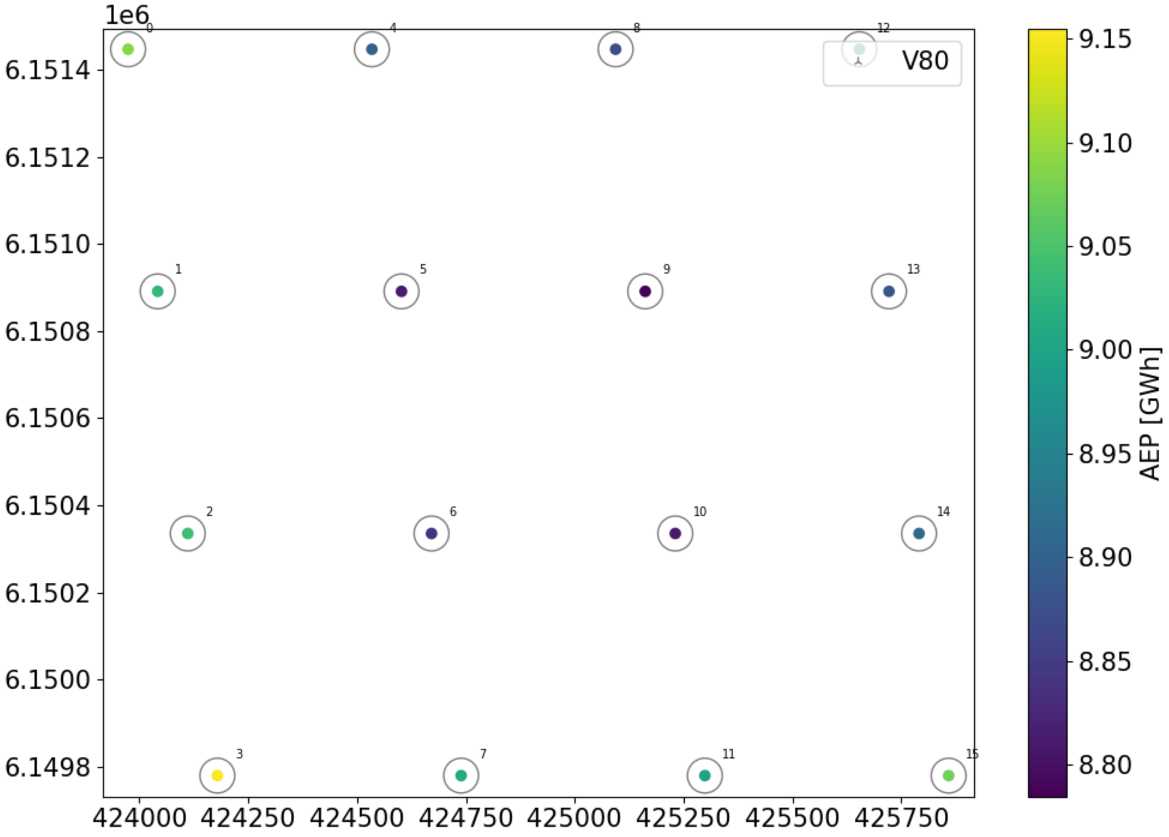

Figure 4.1: Birds eye view of the wind farm, AEP of each turbine

Chapter 5

MoA in Magnolia

In this chapter, we present an implementation of a subset of MoA in the Magnolia programming language, utilizing the magnoliac [5] compiler currently under active development. We leverage a C++ backend to provide us with the basic types and structures we need. The complete code base for the implementation presented in this chapter is publically available online [31].

Remark: This implementation is a result of work done in preparation for the paper presented Chapter 6, and reflects its intended use as a platform to explore the API presented in Burrows et al. and Chetioui et al.. As such, it is not a complete implementation of the ψ-calculus, but rather a subset.

Remark 2: Following and release of the previous Magnolia compiler [2], a large standard library was developed. magnoliac is at the time of writing this thesis incompatible with the standard library, and as such all modules used in this project have been developed independently of previous work.

5.1 Specification

Everything in the ψ-calculus revolves around a single type: the array. Vectors are 1-dimensional arrays, as are shapes and index vectors. Separating the concepts of array, shape, and index will be useful for our purposes. This is primarily due to the fact that operations (both unary and binary) are defined on legal ranges of shapes, and using the type system to constrain the functions we define will make the specification more readable and less error-prone.

```
concept ArrayBaseOps {
  // Array type
  type Array;
  // Index type
  type Index;
  // Shape type
  type Shape;
  // Element type
  require type Element;

  /*
  We separate the integer types based on
  intended use to avoid type mixups
  */
  type Axis;
  type Dim;
  type Offset;
  type Size;

  function dim(a: Array): Dim;
  function shape(a: Array): Shape;
  function total(a: Array): Size;

  // Predicates assuring that index-parameters are of the correct size
  predicate isPartialIndex(i: Index, a: Array);
  predicate isTotalIndex(i: Index, a: Array);

  function psi(i: Index, a: Array): Array guard isPartialIndex(i, a);
  function psi(i: Index, a: Array): Element guard isTotalIndex(i, a);

  function cat(a1: Array, a2: Array): Array
    guard drop(0, shape(a1)) == drop(0, shape(a2));

  function take(o: Offset, a: Array): Array;
  function drop(o: Offset, a: Array): Array;

  // transformations
  procedure rotate(obs ax: Axis, obs j: Offset, upd a: Array)
    guard ax < dim(a);
  procedure reverse(upd a: Array);
  procedure transpose(upd a: Array);
}
```

Listing 5.1: MoA Signature

Now that the core signature is defined, we can equip it with axioms to assert behavior:

```
  // rotate does not change the shape of the array
  axiom rotateShapeAxiom(ax: Axis, j: Offset, a: Array) {
    var pre_shape = shape(a);
    call rotate(ax, j, a);
    assert shape(a) == pre_shape;
  }

  // transposing an array reverses its shape
  axiom transposeShapeAxiom(a: Array) {
    var pre_shape = shape(a);
    call transpose(a);
    assert shape(a) == reverse(pre_shape);
  }
```

Listing 5.2: MoA Axioms

The API described in Burrows et al. states that our implementation is going to need mapped operations on arrays. Let us express this in our specification:

```
concept MappedOps = {

  use ArrayBaseOps;

  // Requiring functions that will be provided by the backend
  require function _+_(a: Element, b: Element): Element;
  require function _-_(a: Element, b: Element): Element;
  require function _*_(a: Element, b: Element): Element;
  require function _/_(a: Element, b: Element): Element;
  require function -_(a: Element): Element;

  require predicate _<_(a: Element, b: Element);
  require predicate _==_(a: Element, b: Element);

  // Array-Array operations
  function _+_(a: Array, b: Array): Array;
  function _-_(a: Array, b: Array): Array;
  function _*_(a: Array, b: Array): Array;
  function _/_(a: Array, b: Array): Array;
  function -_(a: Array): Array;

  predicate _==_(a: Array, b: Array);

  // Scalar-Array operations
  function _+_(a: Element, b: Array): Array;
  function _-_(a: Element, b: Array): Array;
  function _*_(a: Element, b: Array): Array;
  function _/_(a: Element, b: Array): Array;
}
```

Listing 5.3: Signature for mapped operations

With the MappedOps signature defined, we can add axioms to express the intended semantics and relate the mapped operations to their underlying element-wise operations.

```
  axiom binaryMap(a: Array, b: Array, ix: Index)
    guard isTotalIndex(ix, a) && isTotalIndex(ix, b) {

    assert psi(a+b, ix) == psi(a, ix) + psi(b, ix);
    assert psi(a-b, ix) == psi(a, ix) - psi(b, ix);
    assert psi(a*b, ix) == psi(a, ix) * psi(b, ix);
    assert psi(a/b, ix) == psi(a, ix) / psi(b, ix);
  }
  axiom scalarLeftMap(e: Element, a: Array, ix: Index)
    guard isTotalIndex(ix, a) {
    assert psi(e+a, ix) == e + psi(a, ix);
    assert psi(e-a, ix) == e - psi(a, ix);
    assert psi(e*a, ix) == e * psi(a, ix);
    assert psi(e/a, ix) == e / psi(a, ix);
  }
  axiom unaryMap(a: Array, ix: Index) {
    assert psi(-a, ix) == -psi(a, ix);
  }
```

Listing 5.4: Axioms for mapped operations

5.2 Implementation

With our specification completed, we can provide an implementation. This is accomplished by combining externally defined functions with our API and providing our own function and procedure bodies. What we rely on from the backend are:

- A looping mechanism
- An array structure with getters/setters
- Base types

Magnolia does not have built-in support for control structures such as loops, and by design does not allow recursion. While cumbersome, we can implement our own looping structures by leveraging loops present in the backend language at hand. Listing 5.5 is an example of a while-loop with one updatable *state*, and an observable *context*. The `repeat` procedure calls the `body` procedure as long as the `cond` predicate holds true given the context and current state. A simple use case would be printing the elements of a list, providing the list as context, an integer type as state, and cond as a upper bound predicate. A drawback to this approach is that Magnolia lacks support for variadics, which forces us to provide different implementations based on the number of contexts and states one would want present in the loop.

```
implementation WhileLoop1_1 =
  external C++ while_loop1_1 {
    require type Context1;
    require type State1;

    require predicate cond(context1: Context1, state1: State1);
    require procedure body(obs context1: Context1,
                           upd state1: State1);
    procedure repeat(obs context1: Context1,
                     upd state1: State1);
};
```

Listing 5.5: External While-loop in Magnolia with 1 **obs** variable and 1 **upd** variable.

We define our base types and array data structure in C++, contained in structs.

```
template <typename _Element>
struct moa {
  // defining types
  typedef _Element Element;
  typedef int Int32;
  typedef float Float32;
  typedef std::vector<Int> Index;
  typedef std::vector<Index> IndexSpace;
  typedef std::vector<Int> Shape;

  // defining the Array
  struct Array {

      Element * _content;
      Shape _shape;

    // total index, returns element
    inline Element psi(const Index i);
    // partial index, returns subarray
    inline Array psi(const Index i);
  };
  // indexing guards
  inline bool isTotalIndex(const Index i, const Array a)
    return size(i) == size(a);
  inline bool isPartialIndex(const Index i, const Array a)
    return size(i) < size(a);
};
```

Listing 5.6: Snippet of array externals in C++

These can then be put together with their corresponding Magnolia-side definitions using the `external` keyword.

```
implementation ExternalMoaOps = external C++ moa {
  require type Element;
  type Array;
  type Index;
  type IndexSpace;
  type Shape;
  type Int32;
  type Float32;
}
```

Listing 5.7: Array externals in Magnolia

It is worth noting that the required type `Element` is not defined in our `moa` struct, but is instead passed as a generic template argument. This allows us to pass different Element types to our arrays, giving us more flexibility. We provide backend definitions for an Int64 and a Float64 element type, as well as operations on the types, in this implementation.

```
struct float64_utils
{
    typedef double Float64;

    inline Float64 zero() { return 0.0; }
    inline Float64 one() { return 1.0; }

    inline Float64 binary_add(const Float64 a, const Float64 b)
        return a + b;
    inline Float64 binary_sub(const Float64 a, const Float64 b)
        return a - b;
    inline Float64 mul(const Float64 a, const Float64 b)
        return a * b;
    inline Float64 div(const Float64 a, const Float64 b)
        return a / b;
    inline Float64 unary_sub(const Float64 a)
        return -a;
    inline Float64 abs(const Float64 a)
        return std::abs(a);
};
```

Listing 5.8: Backend definition of a Float64 type with arithmetic operations

We also specify a generic NumberType-concept in Magnolia, which in combination with our backend-definition yields our candidates for types we can rename Element to:

```
concept NumberOps = {
    type NumberType;

    function zero(): NumberType;
    function one(): NumberType;

    function binary_add(a: NumberType, b: NumberType): NumberType;
    function binary_sub(a: NumberType, b: NumberType): NumberType;
    function mul(a: NumberType, b: NumberType): NumberType;
    function div(a: NumberType, b: NumberType): NumberType;
    function unary_sub(a: NumberType): NumberType;
    function abs(a: NumberType): NumberType;
}
implementation Float64Utils = external C++ float64_utils
  NumberOps[NumberType => Float64];
```

Listing 5.9: Magnolia definition of a Float64 type with arithmetic operations

Now we have all the building blocks we need to provide an implementation for our `ArrayBaseOps` concept. We make an effort to reflect the MoA notation introduced in Chetioui et al. For the complete list of definitions used to implement these functions we refer the reader to Chetioui et al., but as an informative example we will compare the definition of catenation to our implementation. The definition given in Chetioui et al. reads:

Given an arrays A and B with $\rho(A) = \langle s_0^A, s_1, \ldots, s_{n-1} \rangle$ and $\rho(B) = \langle s_0^B, s_1, \ldots, s_{n-1} \rangle$ – i.e. two arrays with identical shape except for the first shape element – we can describe the result at index $\langle i \rangle$ as

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

Notice how $\langle i \rangle$ is a index vector of length 1. For any array C with $\delta(C) > 1$ this is a partial index, and as such we are updating subarrays rather than individual elements, i.e. a map.

We provide a body to a loop with 3 observable contexts and 2 updatable states, and instantiate the loop with a valid index space, a result array with correct dimensions, and an counter variable `c`.

For each valid index, the procedure `cat_ix` is executed once.

```
procedure cat_ix(obs a: Array,
                 obs b: Array,
                 obs ix: Index,
                 upd res: Array) {

        var s0 = get(shape(a), zero());
        var i0 = get(ix, zero());

        if i0 < s0 then {
            call set(res, ix, psi(ix, a));
        } else {
            var new_ix = create_1d_index(i0 - s0);
            call set(res, ix, psi(new_ix, b));
        };
}
```

Listing 5.10: Implementation of `cat` in Magnolia

All the operations follow this pattern of utilizing an external loop to describe the result at element level or subarray level, depending on whether we have a total or partial index. We continue by providing bodies to our `take` and `drop` prototypes, again calling each procedure once for every valid index.

```
procedure take_ix(obs a: Array,
                  obs ix: Index,
                  obs t: Offset,
                  upd res: Array) {
       if zero() <= t then {
           call set(res, ix, psi(a, ix));
       } else {
           var s0 = get(shape(a), zero());
           var i0 = get(ix, zero());
           var new_ix = create_1d_index(s0 - abs(t) + i0);
           call set(res, ix, psi(new_ix, a));
       };
}
procedure drop_ix(obs a: Array,
                  obs ix: Index,
                  obs t: Int,
                  upd res: Array) {
       if zero() <= t then {
           var i0 = get(ix, zero());
           var new_ix = create_1d_index(i0 + t);
           call set(res, ix, psi(a, new_ix));
       } else {
           call set(res, ix, psi(ix, a));
       };
}
```

Listing 5.11: Implementation of `take` and `drop` in Magnolia

With `cat`, `take` and `drop` defined we can implement our transformations, which are defined in terms of these operations [7].

```
// reverse
procedure reverse_ix(obs a: Array,
                     obs ix: Index,
                     upd res: Array) {
  var sh_0 = get(a, zero());
  var ix_0 = get(ix, zero());

  var new_ix_0 = sh_0 - (ix_0 + one());
  var new_ix = cat_index(create_1d_index(new_ix_0),
                         drop_index_elem(ix, zero()));

  call set(res, new_ix, psi(ix, a));
}
// rotate
procedure rotate_ix(obs a: Array,
                    obs ix: Index,
                    obs sigma: Offset,
                    upd res: Array) {
  if zero() <= sigma then {
      var e1 = take(-sigma, psi(ix, a));
      var e2 = drop(-sigma, psi(ix, a));
      call set(res, ix, cat(e1,e2));
  } else {
      var e1 = drop(sigma, psi(ix, a));
      var e2 = take(sigma, psi(ix, a));
      call set(res, ix, cat(e1,e2));
  };
}
// transpose
procedure transpose_ix(obs a: Array,
                       obs ix: Index,
                       upd res: Array) {
    var e = psi(reverse(ix), a);
    call set(res, ix, e);
}
```

Listing 5.12: Implementation of MoA transformations in Magnolia

With transformations complete, we have successfully implemented the API specified by Burrows et al. and Chetioui et al. In order to get executable code, all that remains is to combine our external element type with our MoA implementation in a `program` module.

```
program Float64Arrays = {
  use Float64Utils;
  use MoA[Element => Float64];
}
```

Listing 5.13: Array program parameterized with a Float64 element type.

5.3 Summary

In this chapter we have provided a specification and implementation of a subset of MoA in Magnolia. We have made a case for how separating the generic API from its implementations can provide more flexibility. With the core API defined, the programmer is free to provide any number of different implementations based on domain specific needs, with guarantees of type safety by the compiler.

Utilizing an external loop to serve as the core of our implementation made it possible to describe the result of a computation at each index, closely following the notational style of recent BLDL efforts [6, 7].

While operational, the implementation presented in this chapter has not lived up to its full potential. Crucially, in order to serve as the backbone for the approach presented in section 6.1, a working implementation of circular padding [7] was needed. While effort was put into development, time constraints prevented it from reaching a satisfactory level suitable for use in the computational experiments planned for the publication. This raises an important point. While the Magnolia compiler can provide some type safety guarantees, externally provided code is no less prone to programming errors. The instability in the case of circular padding could be traced back to the C++ backend, and would require a non-trivial rewrite of the indexing code. As such, a separate, more stable implementation ended up being utilized in the article.

With this experience in mind, this brings us back to a remark from the beginning of the chapter. The implementation presented in this chapter was made without the support of the standard library due to incompatibility issues. Modularity in programming allows for greater flexibility, and it stands to reason that a trusted library of modules provides greater assurance that the types and data structures being imported function as intended.

5.4 Related works on MoA implementations

Previous partial or full implementations of MoA exist.

- In 1994, Mullin and Thibault implemented the Psi Compiler [33], demonstrating a working C implementation of ψ-reduction.
- Python MoA [37] is a proof of concept of a Python implementation of MoA funded by Quansight Labs.
- There are currently efforts to implement a MoA library for LFortran [39], headed by Mullin and a group of people connected to the Fortran community.

Chapter 6

Array Optimizations

6.1 P^3 Problem and Magnolia language: Specializing Array Computations for Emerging Architectures

This section consists of a unpublished paper showcasing array transformations using Magnolia axioms to perform rewrites.

The article is at the time of submitting this thesis undergoing peer review. A key aspect we expect to get constructive feedback on is the number of different architectures explored. The CUDA implementation discussed in section 6.2 is a work in progress to address this.

The author of this thesis contributed to the underlying Magnolia implementation of MoA, a CUDA implementation discussed in section 6.2, as well as contributing to parts of the article text.

frontiers

P^3 Problem and Magnolia Language: Specializing Array Computations for Emerging Architectures

Benjamin Chetioui [1,*], Marius Kleppe Larnøy [1], Jaakko Järvi [2], Magne Haveraaen [1], and Lenore Mullin [3]

[1] *Department of Informatics, University of Bergen, Bergen, Norway*
[2] *Department of Computing, University of Turku, Turku, Finland*
[3] *College of Engineering and Applied Sciences, University at Albany, SUNY, Albany, USA*

Correspondence*:
Benjamin Chetioui
Institutt for informatikk, Thormøhlens Gate 55, 5008 Bergen, Norway
benjamin.chetioui@uib.no

ABSTRACT

The problem of producing portable high-performance computing (HPC) software that is cheap to develop and maintain is called the P^3 (performance, portability, productivity) problem. Good solutions to the P^3 problem have been achieved when the performance profiles of the target machines have been similar. The variety of HPC architectures is, however, large and can be expected to grow larger. Software for HPC therefore needs to be highly adaptable, and there is a pressing need to provide developers with tools to produce software that can target machines with vastly different profiles.

Multi-dimensional array manipulation constitutes a core component of numerous numerical methods, such as finite difference solvers of Partial Differential Equations (PDEs). The efficiency of these computations is tightly connected to traversing and distributing array data in a hardware-friendly way. The Mathematics of Arrays (MoA) allows for formally reasoning about array computations and enables systematic transformations of array-based programs, e.g. to use data layouts that fit to a specific architecture.

This paper shows a general methodology for solving the P^3 problem in a well-specified domain using Magnolia, a language designed to embody generic programming. The Magnolia programmer can restrict the semantic properties of abstract generic types and operations by defining so-called axioms. Axioms can be used to produce tests for concrete implementations of specifications, for formal verification, or to perform semantics-preserving program transformations.

We leverage Magnolia's semantic specification facilities to extend the Magnolia compiler with a term rewriting system. We implement MoA's transformation rules in Magnolia, and demonstrate through a case study on a finite difference solver of PDEs how our rewriting system allows exploring the space of possible optimizations.

Keywords: Partial Differential Equations, Generic Programming, Magnolia Language, Mathematics of Arrays, Term Rewriting, High-Performance Computing

1 INTRODUCTION

The quest for higher performance fuels innovation on hardware architectures; we have seen a wide variety of high-performance computing (HPC) architectures in the past and can expect new ones to keep appearing. Long-lived and successful HPC software must thus be highly adaptable, adjustable to different memory hierarchies and changing intra- and interprocess communication hardware.

The problem of producing portable HPC software that is easy, or at least not unreasonably difficult, to develop and maintain is called the P^3 (performance, portability, productivity) problem. Good solutions to the P^3 problem have been achieved when the performance profiles of the target machines have been similar (Wolfe, 2021). As more new hardware architectures emerge, there is a pressing need to provide developers with tools to produce such software for targets with vastly different profiles. This includes architectures within Wolfe's P^3 machine performance model (CPUs, GPUs, or other accelerators, possibly distributed) (Wolfe, 2021) but also those that do not (e.g., Groq's Tensor Streaming Processor (Abts et al., 2020)).

Multidimensional array manipulation is at the core of numerous numerical methods. The topic of optimizing the performance of array computations is therefore extremely relevant to the P^3 problem. We have previously explored the Mathematics of Arrays (MoA) formalism (Mullin, 1988) as a tool to optimize array computations for different hardware architectures (Chetioui et al., 2019, 2021). A thorough mathematical understanding of a given domain is key to enabling domain-specific semantic-preserving rewrites — and therefore optimizations.

The portability and productivity pillars of P^3 are both strongly related to the notion of code reuse. Portability as meant here is the ability to run the same code with high performance on different machines. Productivity means that applications can be developed and maintained with a reasonable and predicable effort. Research unequivocally shows that productivity increases through reuse (Nazareth and Rothenberger, 2004; Basili et al., 1996; Frakes and Succi, 2001). Generic programming has proven to be an effective method of constructing libraries of reusable software components. The Magnolia programming language (Bagge, 2009) is designed as an embodiment of generic programming (Chetioui et al., 2022). It allows the flexible intermixing of specifications and implementations. Specifications can additionally be restricted by semantic requirements (called *axioms*) in the form of assertions. These axioms can be used for testing (Bagge et al., 2011), but also for optimization when used as directed rewrite rules, in the case of equational or conditional equational axioms (Bagge and Haveraaen, 2009).

1.1 Schedules as Hardware Abstractions

In their 2012 paper on Halide, Ragan-Kelley et al. introduce the term *schedule* to refer to decisions about storage and about the order of computations in a program (Ragan-Kelley et al., 2012). The insight is that the essence of an algorithm is distinct from its schedule — allowing the advent of a programming model where both kinds of computations are not anymore intertwined but instead expressed independently from each other.

Stepanov-style generic programming abstracts algorithms and data structures by specifying minimum syntactic and semantic requirements on instantiations. Said differently, the types and operations underlying a generic implementation are only characterized by the part of their observable behavior that is relevant to the generic algorithm.

When observed through the lens of generic programming, a schedule is an abstraction for the kind of hardware architecture underlying the computations. We consider only the information about the hardware that is relevant for executing our algorithm efficiently: how computations should be ordered, and how data should be stored. Similar hardware architectures are then valid instantiations for the same schedule.

Scheduling, in the case of array computations, relates particularly to the access patterns of the arrays. As a motivating example, consider an array program running on a single CPU with memory, the classical model of a computer. We may have three standard traversal patterns for computations over our arrays:

1. a row-major traversal;
2. a column-major traversal;
3. a tiled traversal.

While the original algorithm can be expressed without making any assumption about the underlying hardware, the choice of a particular hardware will dictate which traversal pattern is most efficient. In other cases, the choice of a particular schedule may be desirable. E.g., on hardware consisting of several distributed CPUs connected through some communication network, we may want the schedule to handle inter-CPU communication using MPI. If each one of these CPUs is connected to several GPUs, we may also want the schedule to load data on and off the GPUs as needed. Such choices will affect the desired data layout, and consequently the data access patterns so as to match the distribution of the data. These changes will have to be reflected in the presentation of the algorithm.

The execution time for an algorithm adapted to its schedule may be dramatically shorter than for an algorithm exhibiting inadapted data access patterns. While an algorithm and its schedule can be expressed independently, choices in the latter may affect what is an appropriate expression of the former, and vice versa. Our approach uses rewriting technology to adapt a unique algorithm to adequately exploit the data traversal pattern of a schedule, and underlying hardware characteristics.

Throughout the rest of the paper, we view schedules as hardware abstractions. This view is fully compatible with Ragan-Kelley et al.'s definition of schedules, but conveys our intent more accurately.

1.2 Contribution and Structure of the Paper

The contribution of this paper is a general methodology for solving the P^3 problem in a well-specified domain, that keeps the essence of the algorithm separate from its schedule. We perform a case study on a Partial Differential Equation (PDE) solver based on Finite Difference Methods (FDM). We extend the Magnolia compiler with code generation and term rewriting facilities based on axioms. We implement our solver in Magnolia, using MoA as an underlying basis for the code, giving us both generic and hardware-specific formally verified optimization rules — also directly implemented in Magnolia.

The paper is structured as follows. Section 2 provides necessary background on MoA and Magnolia. Section 3 describes our methodology in detail, and illustrates it with a PDE solver based on FDM. Section 4 reflects on our work and ties it together with relevant related work.

2 BACKGROUND

2.1 Magnolia

The phrase *generic programming* has over decades of programming language development come to have a variety of intepretations, depending on the main type of genericity considered. Gibbons gives a taxonomy of interpretations (Gibbons, 2006). Stepanov-style generic programming (Dehnert and Stepanov, 1998) corresponds to what Gibbons calls *genericity by property*, where one describes data structures and algorithms in terms of syntactic and semantic requirements. This is the essence of Stepanov's and Musser's *concepts* (Musser and Stepanov, 1988). They are the direct inspiration behind C++0x concepts (Gregor et al., 2006); the C++20 concepts are a scaled back realization of those that only allow syntactic requirements on instantiations. (In this latter case, we talk of *genericity by structure*.)

Magnolia is a programming language designed as an embodiment of Stepanov-style generic programming (Bagge, 2009). Magnolia code is structured into modules that mix abstract specifications of operations and their concrete implementations flexibly, following the work of Goguen and Burstall on the theory of institutions (Goguen and Burstall, 1984). The language does not offer any primitive types aside from predicates: every data structure is implemented in a configurable host programming language. As of today, Magnolia can target C++ and Python (Chetioui, 2021). Our prior work coins the term *genericity by host language* to refer to this axis of parameterization, in the style of Gibbons' taxonomy (Chetioui et al., 2022). Composite operations can be implemented in Magnolia, while the base types and operations, including loop structures, are implemented in the host language. The programmer can freely decide where to set the boundary between the operations implemented in Magnolia, and those implemented in the base library written in the host language — depending on what is more convenient. An appropriate choice of underlying data structures results in code that is as performant as if implemented directly in the host language (Chetioui et al., 2022). Because the axiom formalism is semantically compatible with the program code, Magnolia avoids the semantic gap common in approaches to formal software verification (Sannella and Tarlecki, 1996).

A Magnolia **signature** declares types and operations. A **signature** can be augmented with **axiom**s that restrict the properties of its types and operations: the resulting module is a **concept**. An **implementation** allows the same declarations as a **signature**, but also (generic) implementations for the declared operations. The last kind of module in Magnolia is a **program**, a specific kind of **implementation** in which all the specified operations and types are matched with implementations. Crucially, types and operations in a **program** are no longer generic but instead fully concrete. An **implementation** can be a model of a **concept**; a **concept** can also be a model of another **concept**. Such modeling relations can be specified directly in Magnolia using the **satisfaction** language construct.

Magnolia operations can be **function**s, **procedure**s, and **predicate**s. The arguments to functions and predicates are immutable, while arguments to procedures are given explicit modes: **obs** (read-only), **upd** (read/write), and **out** (write-only, and the procedure promises to initialize the argument). Procedures do not return a value. Calls to procedures are prefixed with the **call** keyword.

Listing 1 gives a general overview of the different kinds of Magnolia modules. We first specify the signature of a magma (a set `T` with a closed binary operation `bop`). By asserting the associativity property on a magma, we get a semigroup. The `ConcretePartialSemigroup` implementation describes an external C++ API providing a guarded multiplication operator over integer matrices, where the guard is intended to ensure the argument matrices have compatible dimensions. `ExampleProgram` builds `multiplyThreeMatrices` off of the primitive building blocks provided by `ConcretePartialSemigroup`. The `ExampleProgramHasMulPartialSemigroup` satisfaction relation indicates that `ExampleProgram` satisfies the semigroup axioms, with the set of integer matrices and guarded multiplication on it. The guard provided on the multiplication operation in the left-hand side of the satisfaction is propagated to the right-hand side. The resulting satisfaction relation asserts the `ExampleProgram` has a partial semigroup structure. A block of renamings (`[ T => IntMatrix, bop => _*_ ]`) is applied to `Semigroup`. Magnolia's renamings allow changing the names of types and operations in places where a module is "opened". This is a powerful feature which allows normalizing the names exposed by modules when we open them in a given scope, independently of how their types and operations were initially named.

Listing 1. Multiplying three matrices in Magnolia.

```
signature Magma = {
  type T;
  function bop(a: T, b: T): T;
}

concept Semigroup = {
  use Magma;
  axiom associativity(a: T, b: T, c: T) {
    assert bop(bop(a, b), c) == bop(a, bop(b, c));
  }
}

implementation ConcretePartialSemigroup =
  external C++ lib.int_matrices {
    type Nat;
    type IntMatrix;

    predicate lhsNrowsIsRhsNcols(m1: IntMatrix, m2: IntMatrix);

    function _*_(m1: IntMatrix, m2: IntMatrix): IntMatrix
      guard lhsNrowsIsRhsNcols(m1, m2);
  }

program ExampleProgram = {
  use ConcretePartialSemigroup;

  function multiplyThreeMatrices(
    A: IntMatrix, B: IntMatrix, C: IntMatrix): IntMatrix = A * B * C;
}

satisfaction ExampleProgramHasMulPartialSemigroup = ExampleProgram
  models Semigroup[ T => IntMatrix, bop => _*_ ];
```

2.1.1 Exploiting Magnolia axioms

Concept axioms have previously found use as test oracles (Bagge et al., 2011) and as generic optimization rules (Tang and Järvi, 2015; Bagge and Haveraaen, 2009). We implement two module transformations called *rewrite* and *generate* in the Magnolia compiler under active development (Chetioui, 2021).

The *rewrite* transformation extracts all assertions of equations from a given concept, and uses them as directed rewrite rules within a target module expression. The maximum allowed number of applications of these directed rewrite rules is provided as an argument to the transformation.

The *generate* transformation highlights a third possible use case for Magnolia axioms, i.e. code generation. The transformation extracts all the assertions of equations from a given concept where the left-hand side is a call to a declared function (or predicate) with two-by-two distinct universally quantified arguments, and generates an implementation for the function where the body is the right-hand side of the assertion. Intuitively, an assertion with the properties we outlined describes the behavior of the function on the left-hand side at every point. Therefore, such assertions are not only a way to specify the intended behavior of a function, but also a way to derive an actual implementation for it in case one was not already provided. Figure 1 describes the grammar for the *rewrite* and *generate* transformations.

Figure 1. The grammar for the *rewrite* and *generate* module transformations in Magnolia.

⟨*transformation*⟩ ::=‘`rewrite`’ ⟨*module-expr*⟩ ‘`with`’ ⟨*module-expr*⟩ ⟨*int*⟩
| ‘`generate`’ ⟨*module-expr*⟩ ‘`in`’ ⟨*module-expr*⟩

Consider the `multiplyThreeMatrices` function in Listing 1. The function is intended to multiply three matrices together — and its body `A * B * C` desugars to the expression `_*_(_*_(A, B), C)`. Due to the associativity property, the order in which the multiplications are executed does not matter when it comes to the correctness of the result. However, it matters a lot when it comes to performance: suppose `A` is of dimensions 100×2, `B` of dimensions 2×20, and `C` of dimensions 20×90. Executing `A * B` requires $100 \times 2 \times 20$ scalar multiplications, and executing `(A * B)* C` thus requires $100 \times 2 \times 20 + 100 \times 20 \times 90 = 184000$ scalar multiplications. On the other side, executing `B * C` requires $2 \times 20 \times 90$ scalar multiplications, and executing `A * (B * C)` requires executing $2 \times 20 \times 90 + 100 \times 2 \times 90 = 21600$ scalar multiplications, nearly ten times fewer.

Suppose that a developer wants to use the `multiplyThreeMatrices` function in their program. They care about efficiency, and know that the input matrices `A`, `B`, and `C` have the same dimensions as specified above. They can use the assertion provided in the associativity property of the `Semigroup` concept as a rewrite rule in `multiplyThreeMatrices` to optimize the expression from `(A * B)* C` to `A * (B * C)`. Listing 2 shows how.

Listing 2. Demonstration of the Magnolia *rewrite* transformation.

```
program DevProgram = rewrite ExampleProgram
  with Semigroup[ bop => _*_, T => IntMatrix ] 1;
```

The Magnolia *rewrite* module transformation takes three arguments: the module on which to perform the rewrite (`ExampleProgram` in the example), the module from which to extract rewriting rules (`Semigroup` with some renamings applied in the example), and a maximum allowed number of rule applications (`1` in the example).

Here, `multiplyThreeMatrices` is a toy example, and defined directly in the **program** being transpiled — it would therefore be very easy to reimplement it manually. However, this is not always the case: the function one wants to transform could be very complicated, and hidden deep inside an external dependency. Without the ability to perform rewritings on functions that have been previously defined, the developer would have to write their own version of this function.

3 METHODOLOGY AND CASE STUDY

We describe here our proposed methodology for writing performant and portable code productively using the Magnolia programming language. Each step is first described from a high-level perspective, and then concretely demonstrated for our PDE solver example.

3.1 Identifying and Formalizing the Domain

The first step of our methodology is to build a thorough understanding of the targeted problem. We do that by identifying and formalizing the domain underlying the problem. Formalizing the domain gives us a mathematical understanding of the properties expected of the types and operations involved in the problem. These in turn allow specifying semantics-preserving optimization rules on them, whose correctness can be proven.

PDE solvers using FDM are based on multi-dimensional array computations. In 2018, Burrows et al. identified an array API for FDM solvers. In 2019, Chetioui et al. followed up with a formalization of the identified array API using MoA. We will first give an overview of PDE solvers as described by Burrows et al., and an introduction to the corresponding MoA theory. With this background in place, we will reimplement the PDE solver based on FDM from the work of Chetioui et al., and implement hardware-agnostic and hardware-dependent rewriting rules. We show how they can be applied to our Magnolia program, and measure the resulting performance improvements.

3.1.1 PDEs

PDE solvers have many application areas. One example is numerical simulations of wind flow — e.g. for optimizing windmill positioning in large-scale wind farms.

Computing solutions to PDEs numerically requires discretizing continuous equations to a discrete domain. This approach to PDE solvers is often illustrated in the literature with Burgers' equation (Burgers, 1948). Equation 1 presents the equation in its coordinate-free form.

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} = \nu \nabla^2 \vec{u}, \tag{1}$$

where $\vec{u}$ is velocity, t time, and ν the viscosity coefficient.

Assuming a 3D space, we can use a Cartesian coordinate system to rewrite Equation 1 as the following system of equations

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = \nu\frac{\partial^2 u}{\partial x^2} + \nu\frac{\partial^2 u}{\partial y^2} + \nu\frac{\partial^2 u}{\partial z^2} \tag{2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = \nu\frac{\partial^2 v}{\partial x^2} + \nu\frac{\partial^2 v}{\partial y^2} + \nu\frac{\partial^2 v}{\partial z^2} \tag{3}$$

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = \nu\frac{\partial^2 w}{\partial x^2} + \nu\frac{\partial^2 w}{\partial y^2} + \nu\frac{\partial^2 w}{\partial z^2}, \tag{4}$$

where $\vec{u} = (u, v, w)$.

To discretize the domain, we describe a $N_x \times N_y \times N_z$ grid of velocity values bounded by L_x (respectively L_y and L_z) on axis x (respectively y and z) such that the u component of the velocity at index (i, j, k) and timestep n is given by

$$u^n_{i,j,k} = u(i\Delta x, j\Delta y, k\Delta z, n\Delta t), \tag{5}$$

with $\Delta x = \frac{L_x}{N_x}$, $\Delta y = \frac{L_y}{N_y}$, and $\Delta z = \frac{L_z}{N_z}$.

Similarly, the partial derivative of u in the x direction at index (i, j, k) and timestep $n + 1$ is

$$\frac{\partial u}{\partial x}(i\Delta x, j\Delta y, k\Delta z, (n + 1)\Delta t). \tag{6}$$

In the FDM, we compute a partial derivative as a weighted sum of neighbouring grid points — where the weights are given by a list of factors called a *stencil*. The stencil is chosen by a numerical expert. This paper, following the work of Burrows et al. uses the numerical stencils $(-\frac{1}{2}, 0, \frac{1}{2})$ and $(1, -2, 1)$ for the first and second order partial derivatives respectively.

Given these stencils, the partial derivative of u in the x direction at index (i, j, k) and timestep $n + 1$ is approximated by

$$\frac{\partial u}{\partial x}(i\Delta x, j\Delta y, k\Delta z, (n + 1)\Delta t) \approx \frac{\Delta t}{2\Delta x}(u^n_{i+1,j,k} - u^n_{i-1,j,k}), \tag{7}$$

which is accurate to $O((\Delta x)^2, \Delta t)$. Computing the partial derivative along the y (respectively z) axis follows a similar pattern, where j (respectively k) varies instead of i.

The standard 3D explicit finite difference approximation of Equation 2 is then given by

$$\begin{aligned} u^{n+1}_{i,j,k} = u^n_{i,j,k} &- \frac{\Delta t}{2\Delta x} u^n_{i,j,k}(u^n_{i+1,j,k} - u^n_{i-1,j,k}) + \frac{\nu\Delta t}{(\Delta x)^2}(u^n_{i+1,j,k} + u^n_{i-1,j,k} - 2u^n_{i,j,k}) \\ &- \frac{\Delta t}{2\Delta y} v^n_{i,j,k}(u^n_{i,j+1,k} - u^n_{i,j-1,k}) + \frac{\nu\Delta t}{(\Delta y)^2}(u^n_{i,j+1,k} + u^n_{i,j-1,k} - 2u^n_{i,j,k}) \\ &- \frac{\Delta t}{2\Delta z} w^n_{i,j,k}(u^n_{i,j,k+1} - u^n_{i,j,k-1}) + \frac{\nu\Delta t}{(\Delta z)^2}(u^n_{i,j,k+1} + u^n_{i,j,k-1} - 2u^n_{i,j,k}). \end{aligned}$$

The discretization of Equations 3 and 4 follows the same pattern.

The API of Burrows et al. is sufficient to compute numerical solutions to PDEs using FDM. It consists of elementwise arithmetic operations at the array level ($+$, $-$, $\star$), a rotation operation on arrays (called “shift”), and arithmetic operations at the scalar level — corresponding to the behavior of the elementwise operations at each index of the array.

3.1.2 MoA

MoA (Mullin, 1988; Mullin and Jenkins, 1996) is an algebra for describing operations on arrays. MoA distinguishes between two abstraction levels: the *Denotational Normal Form* (DNF), which describes an array by its shape together with a function describing its value at every index, and the *Operational Form* (OF) which describes it on the level of the memory layout. Programs written at the DNF level do not presume knowledge of a hardware architecture. Reasoning at the DNF level is thus completely hardware agnostic. By repeatedly applying a set of terminating rewrite rules, any array expression can be reduced to its DNF (Mullin and Thibault, 1994; Chetioui et al., 2019) — where the resulting array is described at each index by indexing into the input arrays and scalar-level operations.

Given information about the hardware architecture and the memory layout of the arrays, the ψ-correspondence theorem (Mullin and Jenkins, 1996) allows transforming a DNF expression into a corresponding hardware-dependent OF — in which the access patterns on the array are described in terms of *start*, *stride*, and *length*.

Chetioui et al. investigate the fragment of MoA corresponding to the API identified by Burrows et al., and show that for programs based on it, DNF reduction is indeed canonical, which draws appeal to MoA as a framework for the optimization of PDE solvers based on FDM.

We give an informal overview of some operations at the DNF and OF levels below. We refer the interested reader to previous work for formal definitions (Chetioui et al., 2021; Mullin, 1988).

DNF Operations

The *dimension* of an array A is denoted $\mathsf{dim}(A)$. It corresponds to the number of axes of the array. For $\mathsf{dim}(A) = n$, the *shape* of A is an n-element vector $\rho(A) = \langle s_0, \ \ldots, \ s_{n-1} \rangle$ where s_i is the length of axis i. The total number of elements (or *size*) of A is given by the product of the shape, $\Pi\rho(A) = \Pi_{i=0}^{n-1} s_i$.

In the definitions below A stands for an arbitrary array with dimension n and shape as defined above. Further, we use the following array in examples:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

Thus, $\rho(M) = \langle 3, 2 \rangle$.

The relevant MoA operations on the DNF level are:

- the indexing function ψ, which takes an index i into an array and returns the subarray at the indexed position. When i's length equals the dimension of the array, i is a *total* index. Otherwise, it is *partial*. $\langle\rangle \ \psi \ A = A$ holds. For our example, we have

$$\langle 2 \rangle \ \psi \ M = \langle 5, 6 \rangle$$
$$\rho(\langle 2 \rangle \ \psi \ M) = \langle 2 \rangle$$

- the reshape function that takes an array A and a shape s such that $\Pi s = \Pi\rho(A)$, and creates a new array with shape s containing the elements of A. Thus, $\rho(\text{reshape}(s, A)) = s$ holds. For example,

$$\text{reshape}(M, \langle 2, 3 \rangle) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

- a rotation function rotate that takes an array A, an axis j and an offset o, and shift A by o along its j^{th} axis. The shape is unchanged, i.e. $\rho(\mathsf{rotate}(A, j, o)) = \rho(A)$ holds. We give a few examples of how rotation behaves on axis 0 and 1 of M:

$$\mathsf{rotate}(M, 0, 1) = \begin{pmatrix} 5 & 6 \\ 1 & 2 \\ 3 & 4 \end{pmatrix},$$

$$\mathsf{rotate}(M, 0, -1) = \begin{pmatrix} 3 & 4 \\ 5 & 6 \\ 1 & 2 \end{pmatrix},$$

$$\mathsf{rotate}(M, 1, 1) = \begin{pmatrix} 2 & 1 \\ 4 & 3 \\ 6 & 5 \end{pmatrix}.$$

ψ-Reduction

Mullin and Thibault described a rewriting system for MoA expressions at the DNF level, referred to as ψ-reduction. They conjectured that ψ-reduction is canonical — and thus takes any expression to its unique DNF. This conjecture was proven to hold by Chetioui et al. for the fragment of the rewriting system required by the array API identified by Burrows et al. (Chetioui et al., 2019). ψ-reduction essentially consists of rules that move indexing operations inwards — until eventually, the expression does not contain any collective operation, but consists only of indexing and scalar operations. As a consequence, it is guaranteed that the resulting array expression can be computed without the need to materialize any intermediate array. Because the rewriting system is canonical, another consequence is that the form in which we choose to express our computation is irrelevant: all equivalent expressions in the language of MoA reduce to the same DNF expression.

OF Operations

At the OF level, we assume knowledge of the target architecture, and an intended memory layout of the array. The central MoA operations on the OF level are:

- the family of lifting operations lift_j that take two natural numbers d, q such that $d \cdot q = s_j$, and reshape A into the shape $\langle s_0,\ \ldots,\ s_{j-1},\ d,\ q,\ s_{j+1},\ \ldots,\ s_{n-1}\rangle$;
- the flattening function rav that transforms a multidimensional array into its linear representation in memory. Thus, $\rho(\mathsf{rav}(A)) = \langle \Pi\rho(A)\rangle$ holds;
- the mapping function γ, which takes a shape s with $\Pi s = \Pi\rho(A)$ and a total index into A and returns the corresponding index into $\mathsf{rav}(A)$. In this paper, we assume a row-major ordering.

The OF operations presented here are crucial to the theory of MoA. We thus include them for the sake of completion. These operations however do not appear explicitly in the development of our methodology.

3.1.3 Initial Magnolia Implementation

We implemented a PDE solver using the MoA array API. The implementation consists of four components:

1. a specification of the necessary MoA types and operations, with axioms asserting that they respect the relevant properties;
2. a foreign API exposing the core types and operations of the MoA specification;

3. an external implementation of the foreign API in a host language (C++);
4. an implementation of the PDE solver built upon the external MoA building blocks.

The ψ-calculus conflates arrays, indices, shapes, and scalars into a single array type. While convenient in the formalism, we distinguish these types in our Magnolia implementation for ease of reasoning, and to leverage the language's type system to avoid programming errors.

Listing 3 shows the API from Burrows et al. in the language of MoA.

Listing 3. An array API for FDM solvers in Magnolia.

```
signature ArrayAPI = {
  type Array;
  type E;

  type Axis;
  type Index;
  type Offset;

  /* Scalar-Scalar operations */
  function _+_(lhs: E, rhs: E): E;
  function _-_(lhs: E, rhs: E): E;
  function _*_(lhs: E, rhs: E): E;
  function _/_(lhs: E, rhs: E): E;

  /* Scalar-Array operations */
  function _+_(lhs: E, rhs: Array): Array;
  // ... prototypes as above for _-_, _*_, _/_

  /* Array-Array operations */
  function _+_(lhs: Array, rhs: Array): Array;
  // ... prototypes as above for _-_, _*_, _/_

  /* Rotation */
  function rotate(array: Array, axis: Axis, offset: Offset): Array;

  /* Indexing */
  function psi(ix: Index, array: Array): E;
}
```

The declaration of the types and operations form an algebraic *signature*. We augment that signature with semantic properties in the form of *axioms* to obtain a *concept*. Listing 4 relates each array-level arithmetic operation in the API to its corresponding scalar-level operation (Burrows et al., 2018; Chetioui et al., 2019). The axioms for all binary operations follow the same pattern, we hence only show axiom bodies for the + operation for the sake of brevity.

Listing 4. Axioms for the arithmetic operations of our array API.

```
concept ArrayAPI_ArithmeticAxioms = {
  require ArrayAPI;

  /* Scalar-Array Axioms */
  axiom scalarBinaryPlusAxiom(lhs: E, rhs: Array, ix: Index) {
    assert psi(ix, lhs + rhs) == lhs + psi(ix, rhs);
  }
  // axiom scalarBinarySubAxiom(lhs: E, rhs: Array, ix: Index)
  // axiom scalarMulAxiom(lhs: E, rhs: Array, ix: Index)
  // axiom scalarDivAxiom(lhs: E, rhs: Array, ix: Index)

  /* Array-Array Axioms */
  axiom arrayBinaryPlusAxiom(lhs: Array, rhs: Array, ix: Index) {
    assert psi(ix, lhs + rhs) == lhs + psi(ix, rhs);
  }
  // axiom arrayBinarySubAxiom(lhs: Array, rhs: Array, ix: Index)
  // axiom arrayMulAxiom(lhs: Array, rhs: Array, ix: Index)
  // axiom arrayDivAxiom(lhs: Array, rhs: Array, ix: Index)
}
```

The specifications in Listing 3 are (straightforwardly) implemented as external C++ functions and types, not shown here. Lastly, Listing 5 shows our implementation of one full step of the PDE.

Listing 5. Implementation of one full step of the PDE solver in Magnolia.

```
/* Solver */
procedure step(upd u0: Array, upd u1: Array, upd u2: Array,
               obs nu: Float, obs dx: Float, obs dt: Float) {
  var _1 = one(): Float;
  var _2 = two(): Float;

  var c0 = _1/_2/dx;
  var c1 = _1/dx/dx;
  var c2 = _2/dx/dx;
  var c3 = nu;
  var c4 = dt/_2;

  call allSubsteps(u0, u1, u2, c0, c1, c2, c3, c4);
}

procedure allSubsteps(upd u0: Array, upd u1: Array, upd u2: Array,
                      obs c0: Float, obs c1: Float, obs c2: Float,
                      obs c3: Float, obs c4: Float) {
  var v0 = u0;
  var v1 = u1;
  var v2 = u2;

  v0 = substep(v0, u0, u0, u1, u2, c0, c1, c2, c3, c4);
  v1 = substep(v1, u1, u0, u1, u2, c0, c1, c2, c3, c4);
  v2 = substep(v2, u2, u0, u1, u2, c0, c1, c2, c3, c4);
  u0 = substep(u0, v0, u0, u1, u2, c0, c1, c2, c3, c4);
  u1 = substep(u1, v1, u0, u1, u2, c0, c1, c2, c3, c4);
  u2 = substep(u2, v2, u0, u1, u2, c0, c1, c2, c3, c4);
}

function substep(u: Array, v: Array, u0: Array,
                 u1: Array, u2: Array, c0: Float,
                 c1: Float, c2: Float, c3: Float,
                 c4: Float): Array =
  u + c4 * (c3 * (c1 *
    (rotate(v, zero(), -one(): Offset) +
     rotate(v, zero(), one(): Offset) +
     rotate(v, one(): Axis, -one(): Offset) +
     rotate(v, one(): Axis, one(): Offset) +
     rotate(v, two(): Axis, -one(): Offset) +
     rotate(v, two(): Axis, one(): Offset)) - three() * c2 * u0) -
  c0 * ((rotate(v, zero(), one(): Offset) -
         rotate(v, zero(), -one(): Offset)) * u0 +
        (rotate(v, one(): Axis, one(): Offset) -
         rotate(v, one(): Axis, -one(): Offset)) * u1 +
        (rotate(v, two(): Axis, one(): Offset) -
         rotate(v, two(): Axis, -one(): Offset)) * u2));

/* Float utils */
require function -_(f: Float): Float;
require function one(): Float;
require function two(): Float;
require function three(): Float;

/* Axis utils */
require function zero(): Axis;
require function one(): Axis;
require function two(): Axis;

/* Offset utils */
require function one(): Offset;
require function -_(o: Offset): Offset;
```

3.2 Deriving Optimization Rules

Armed with a thorough understanding of the problem, we can now derive semantics-preserving optimization rules — hardware-specific or otherwise.

Before we can apply rewriting rules defined using MoA to our program, we need to change its level of abstraction, i.e. go from an implementation that describes the resulting array using whole-array operations to one that describes its value at every index. Consider the `ToIxwiseGenerator` concept in Listing 6.

The `toIxwiseGenerator` axiom consists of a single assertion, which describes the behavior of the `substepIx` function when all of its arguments are universally quantified distinct variables. The right-hand side of the equation is thus a valid implementation for `substepIx`. Because this function is not implemented in the original program, we can use the *generate* transformation with `ToIxwiseGenerator` to generate an implementation of `substepIx` in the implementation given in Listing 5. So as to enable further optimizations, *generate* unfolds function calls in the right-hand side of the equation. The resulting index-level code is shown in Listing 7.

Listing 6. A generator for an index-level implementation of substep.

```
concept ToIxwiseGenerator = {
    type Array;
    type Float;
    type Index;

    function substepIx(u: Array, v: Array, u0: Array,
                       u1: Array, u2: Array, c0: Float,
                       c1: Float, c2: Float, c3: Float,
                       c4: Float, ix: Index): Float;

    function substep(u: Array, v: Array, u0: Array,
                     u1: Array, u2: Array, c0: Float,
                     c1: Float, c2: Float, c3: Float,
                     c4: Float): Array;

    function psi(ix: Index, array: Array): Float;

    axiom toIxwiseGenerator(u: Array, v: Array, u0: Array,
                            u1: Array, u2: Array, c0: Float,
                            c1: Float, c2: Float, c3: Float,
                            c4: Float, ix: Index) {
      assert substepIx(u, v, u0, u1, u2, c0, c1, c2, c3, c4, ix) ==
             psi(ix, substep(u, v, u0, u1, u2, c0, c1, c2, c3, c4));
    }
}
```

Listing 7. Index-level implementation of substep in Magnolia.

```

require function psi(ix: Index, array: Array): Float;

function substepIx(u: Array, v: Array, u0: Array,
                   u1: Array, u2: Array, c0: Float,
                   c1: Float, c2: Float, c3: Float,
                   c4: Float, ix: Index): Float =
  psi(ix, u + c4 * (c3 * (c1 *
            (rotate(v, zero(): Axis, -one(): Offset) +
             rotate(v, zero(): Axis, one(): Offset) +
             rotate(v, one(): Axis, -one(): Offset) +
             rotate(v, one(): Axis, one(): Offset) +
             rotate(v, two(): Axis, -one(): Offset) +
             rotate(v, two(): Axis, one(): Offset)) -
          three() * c2 * u0) - c0 *
            ((rotate(v, zero(): Axis, one(): Offset) -
              rotate(v, zero(): Axis, -one(): Offset)) * u0 +
             (rotate(v, one(): Axis, one(): Offset) -
              rotate(v, one(): Axis, -one(): Offset)) * u1 +
             (rotate(v, two(): Axis, one(): Offset) -
              rotate(v, two(): Axis, -one(): Offset)) * u2)));
}
```

To make use of `substepIx` within the program, we need to replace calls to `substep` with calls to a scheduling function that uses `substepIx` to describe the value of the array at every index. We use the program transformation **rewrite ... with** `ToIxwise 1` to achieve that, with `ToIxwise` a concept of Listing 8. Throughout the rest of the paper, we use the term *schedule* like in Halide (Ragan-Kelley et al., 2012).

Listing 8. A concept with a rewrite rule from substep to a new scheduling function.

```
concept ToIxwise = {
  type Array;
  type Float;

  function substep(u: Array, v: Array, u0: Array,
                   u1: Array, u2: Array, c0: Float,
                   c1: Float, c2: Float, c3: Float,
                   c4: Float): Array;

  function schedule(u: Array, v: Array, u0: Array,
                    u1: Array, u2: Array, c0: Float,
                    c1: Float, c2: Float, c3: Float,
                    c4: Float): Array;

  axiom toIxwiseRule(u: Array, v: Array, u0: Array,
                     u1: Array, u2: Array, c0: Float,
                     c1: Float, c2: Float, c3: Float,
                     c4: Float) {
    assert substep(u, v, u0, u1, u2, c0, c1, c2, c3, c4) ==
           schedule(u, v, u0, u1, u2, c0, c1, c2, c3, c4);
  }
}
```

Magnolia does not expose native looping constructs. For that reason, the implementation of `schedule` is done in the host language. The `schedule` function uses the `substepIx` function in Listing 7 to describe the content of the result array at every index.

From that point onwards, we can use MoA to derive transformation rules on our program.

3.2.1 Hardware-Agnostic Transformation Rules

In their work on embedding Burrows et al.'s array API for FDM solvers in MoA, Chetioui et al. outline a rewriting system sufficient to transform a program based on this API to its DNF. This rewriting system is canonical, i.e. rewriting always terminates, and the order in which the rules are applied is inconsequential.

Rewriting rules at the DNF level do not require hardware knowledge, and therefore constitute hardware-agnostic transformation rules. We show an implementation of these rules in Magnolia in Listing 9.

Listing 9. The DNF rewriting rules in Magnolia.

```
concept GenericBinopRules = {
  type E;
  type Array;
  type Index;

  function binop(lhs: E, rhs: E): E;
  function binop(lhs: E, rhs: Array): Array;
  function binop(lhs: Array, rhs: Array): Array;
  function psi(ix: Index, array: Array): E;

  // Rule 1
  axiom binopArrayRule(ix: Index, lhs: Array, rhs: Array) {
    assert psi(ix, binop(lhs, rhs)) ==
           binop(psi(ix, lhs), psi(ix, rhs));
  }

  // Rule 2
  axiom binopScalarRule(ix: Index, lhs: E, rhs: Array) {
    assert psi(ix, binop(lhs, rhs)) == binop(lhs, psi(ix, rhs));
  }
}

concept DNFRules = {
  use GenericBinopRules[ binop => _+_
                       , binopScalarRule => addScalarRule
                       , binopArrayRule => addArrayRule
                       ];
  use GenericBinopRules[ binop => _-_
                       , binopScalarRule => subScalarRule
                       , binopArrayRule => subArrayRule
                       ];
  use GenericBinopRules[ binop => _*_
```

```
                       , binopScalarRule => mulScalarRule
                       , binopArrayRule => mulArrayRule
                       ];
  use GenericBinopRules[ binop => _/_
                       , binopScalarRule => divScalarRule
                       , binopArrayRule => divArrayRule
                       ];

  type Axis;
  type Offset;

  function rotate(array: Array, axis: Axis, offset: Offset): Array;
  function rotateIx(ix: Index, axis: Axis, offset: Offset): Index;

  // Rule 3
  axiom rotateRule(ix: Index, array: Array, axis: Axis,
                   offset: Offset) {
    assert psi(ix, rotate(array, axis, offset)) ==
           psi(rotateIx(ix, axis, offset), array);
  }
}[ E => Float ];
```

As explained in Subsubsection 3.1.2, applying the DNF rules pushes computations down from the array-level to the index-level, i.e. the resulting computations are devoid of whole-array operations and contain only indexing and scalar arithmetic operations.

Table 1 shows runtime results for our PDE solver implementation in Magnolia, before and after full DNF reduction using the DNF rewriting rules. DNF reduction speeds up the code by a factor of roughly $4.18\times$ and significantly reduces memory usage. At the DNF level, the expression is written in terms of scalar and indexing operations, eliminating the need to compute temporary arrays, and increasing computational density. This experiment shows that such a rewriting system gives the ability to write programs using whole-array operations without losing out on the benefits of writing index-level code. The ability to write algorithms in different ways without inducing a loss of performance is key to the productive development of performant code.

	Wall time (in seconds)
Before DNF reduction	323.02
After DNF reduction	42.26

Table 1. Execution time (in seconds) of the 3-dimensional PDE solver Magnolia implementation compiled to C++, with and without reduction to DNF. The code is compiled with gcc 10.2.1 with optimization level O3. The space dimensions are $256 \times 256 \times 256$ and the solver is run for 50 timesteps. The code is run 10 times on the Intel Xeon Silver 4112 CPU, and the time measurements are averaged.

3.2.2 Hardware-Specific Transformation Rules

Which hardware-specific transformation rules are relevant to implement is by nature dependent on the underlying hardware architecture we are interested in. For example, Chetioui et al.'s previous work on formalizing PDE computations in MoA gave rise to rules for introducing padding into array expressions. Their work also discusses rewrites rules that use the *dimension lifting* operation, which is a *reshape* operation with the explicit purpose of matching the shape of arrays with characteristics of the underlying hardware. E.g. lifting by d_1 across the first axis allows one to *scatter* the resulting subarrays across d_1 processes; or, lifting by 4 across the last axis of an array of 32-bit floats allows one to vectorize computations on an architecture with 128-bit vector registers. The hardware architecture combined with the data dependencies of the algorithm determine the shape and layout of the arrays.

We discuss two examples of such hardware-dependent rewriting systems below.

Example: Dimension lifting over several cores

At the DNF level, our concern was to express our algorithm in a canonical form, without paying any mind to hardware-related concerns. A contrario, our concern at the OF level is to make the best use of the hardware available. The rewrites we express are thus often concerned with changing the schedule of our computations. Scheduling is handled outside of Magnolia in our example, by the `schedule` function.

Listing 10 showcases a rewriting rule for moving from our initial scheduling function to one that parallelizes the computation over several cores, the number of which can be parameterized externally.

Listing 10. The rewriting rules for distributing the computation on several cores.

```
concept OFLiftCores = {
  type Array;
  type Float;
  type Axis;
  type Nat;

  function nbCores(): Nat;

  function scheduleThreaded(
              u: Array, v: Array,
              u0: Array, u1: Array, u2: Array,
              c0: Float, c1: Float, c2: Float, c3: Float, c4: Float,
              nbThreads: Nat
            ): Array;

  function schedule(
              u: Array, v: Array,
              u0: Array, u1: Array, u2: Array,
              c0: Float, c1: Float, c2: Float, c3: Float, c4: Float
            ): Array;

  axiom liftCoresRule(
              u: Array, v: Array,
              u0: Array, u1: Array, u2: Array,
              c0: Float, c1: Float, c2: Float, c3: Float, c4: Float
            ) {
    assert schedule(u, v, u0, u1, u2, c0, c1, c2, c3, c4) ==
            scheduleThreaded(u, v, u0, u1, u2, c0, c1,
                             c2, c3, c4, nbCores());
  }
}
```

The implementation of the new `scheduleThreaded` function must also be provided externally. Because the schedule is separate from the algorithm, the cost of expressing scheduling rewrites is mostly the cost of implementing a new schedule. Once a schedule is implemented, it can be reused for algorithms exhibiting similar data dependency patterns, and to target similar hardware. The cost of implementing scheduling rewrites thus decreases as more schedules are implemented, and more problems are explored.

Example: Padding computations

Figure 2 shows the dependency patterns for one third of a half-step of the PDE across the last axis of the array. The element at index i at time $t+1$ depends on the elements at index i, $(i-1) \bmod N$, and $(i+1) \bmod N$ at time t. The modulo operation serves to index the right dependencies for the first (respectively last) element of the array, where decrementing (respectively incrementing) the index would create an out-of-bounds index. Modulo operations are still expensive, even on modern hardware (Lemire et al., 2019). Additionally, if N is large, the computations at the boundary need to access elements that are far apart in memory — therefore benefitting less from data locality than the computations in the middle of the array.

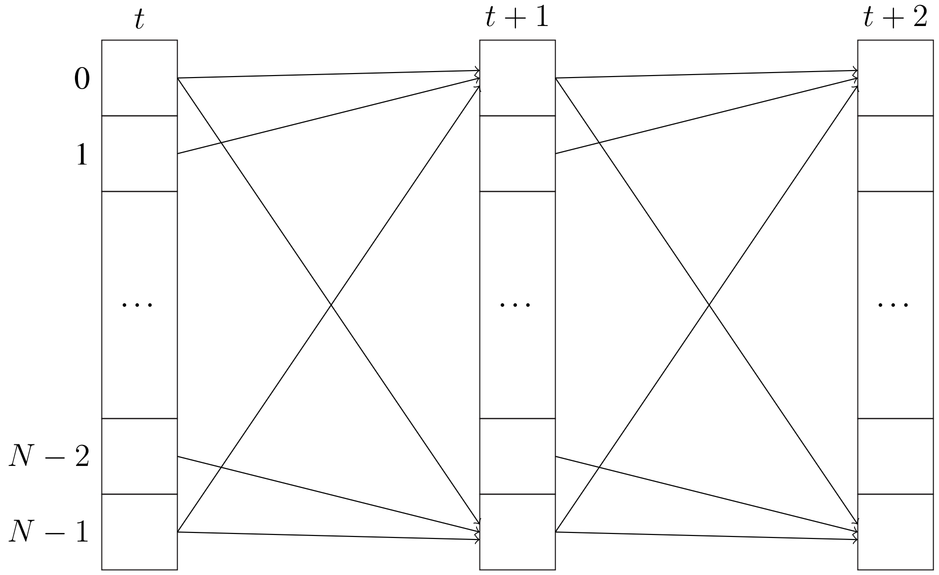

Figure 2. The dependency pattern for one third of a half-step of the PDE across the last axis of the array. Each column represents an array of length N indexed from 0 to $N-1$ for a given timestep. The element at index i of the array at time $t+1$ depends on the elements at indices i, $(i-1) \bmod N$ and $(i+1) \bmod N$ of the array at time t.

Chetioui et al. previously showed that padding is a way to eliminate these modulo computations and to increase data locality, at the cost of duplicating data in memory (Chetioui et al., 2021).

Figure 3 shows the dependency patterns for one third of a half-step of the PDE across the last axis of the array when the array is padded. In that case, the computation at the boundaries of the array can be rewritten to depend on three adjacent elements in the array. The modulo computation can also be eliminated. We pay for these improvements by using more space, and by refilling the padding before every timestep.

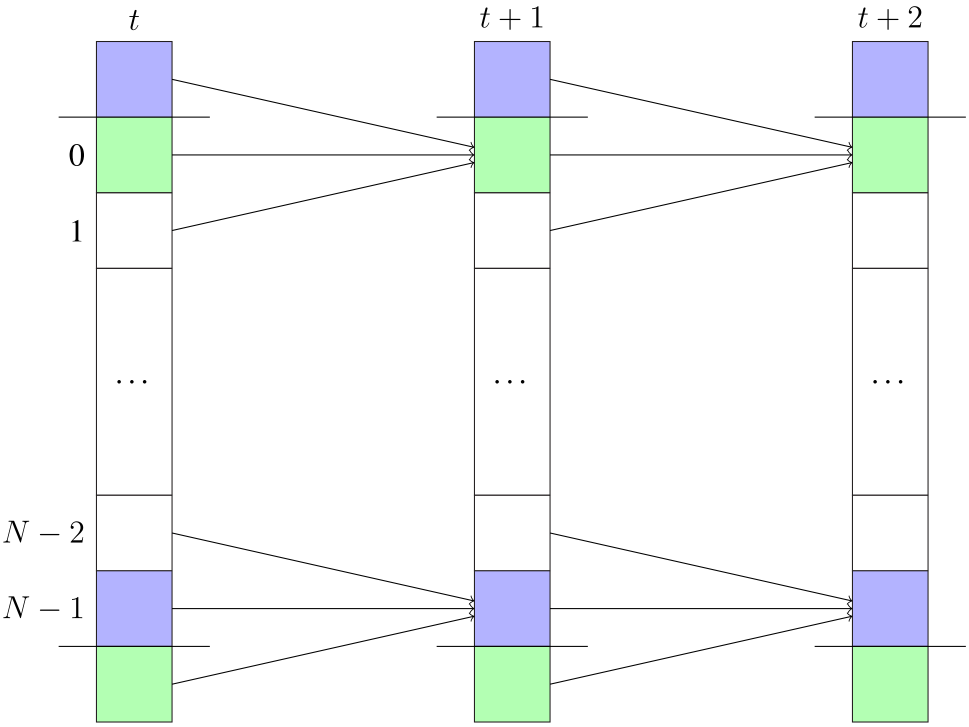

Figure 3. The dependency pattern for one third of a half-step of the PDE across the last axis of the array when the array is padded once on each side on the last axis. Each column represents an array of length N indexed from 0 to $N-1$ for a given timestep. The elements colored in the same color have the same value. The element at index i of the array at time $t+1$ depends on the elements at indices i, $i-1$ and $i+1$ of the array at time t.

Listing 11 shows an implementation of the padding transformation rules in Magnolia.

Listing 11. The rewriting rules for padding.

```
concept OFPad = {
  type Array;
  type Float;

  procedure allSubsteps(upd u0: Array, upd u1: Array, upd u2: Array,
                        obs c0: Float, obs c1: Float, obs c2: Float,
                        obs c3: Float, obs c4: Float);

  procedure refillPadding(upd a: Array);

  function schedulePadded(u: Array, v: Array,
    u0: Array, u1: Array, u2: Array, c0: Float,
    c1: Float, c2: Float, c3: Float, c4: Float): Array;

  function schedule(u: Array, v: Array,
                    u0: Array, u1: Array, u2: Array,
                    c0: Float, c1: Float, c2: Float,
                    c3: Float, c4: Float): Array;

  axiom padRule(u: Array, v: Array, u0: Array, u1: Array, u2: Array,
                c0: Float, c1: Float, c2: Float, c3: Float,
                c4: Float) {
    assert schedule(u, v, u0, u1, u2, c0, c1, c2, c3, c4) ==
      { var result =
              schedulePadded(u, v, u0, u1, u2, c0, c1, c2, c3, c4);
        call refillPadding(result);
        value result;
      };
  }

  type Index;
  type Axis;
  type Offset;
  function rotateIx(ix: Index, axis: Axis, offset: Offset): Index;
  function rotateIx_padded(ix: Index, axis: Axis, offset: Offset)
    : Index;

  axiom rotateIxPadRule(ix: Index, axis: Axis, offset: Offset) {
    assert rotateIx(ix, axis, offset) ==
           rotateIx_padded(ix, axis, offset);
  }
}
```

The implementation in Listing 11 assumes that the input arrays are padded arbitrarily across each axis in the host language, in a way that is compatible with the new `rotateIx_padded` function. Details such as the amount of padding across each axis are therefore not visible in Magnolia. This is however purely a design choice, insofar as we have chosen to make the `Index` type completely opaque. This has the benefit of making the program naturally shape polymorphic to a degree — though the program is not as interesting for input arrays with initial number of dimensions different than three.

We can control padding across each axis more explicitly by specializing our code further. This can also be achieved using transformation rules — we describe the steps below.

Listing 12 shows an axiom following the generator pattern to specialize the shape polymorphic `substepIx` to three dimensions. As previously, the call to `substepIx` on the right-hand side of the equation is unfolded to enable additional optimizations.

Listing 12. A generator for a 3D implementation of substepIx.

```
concept OFSpecializeSubstepGenerator = {
  type Index;
  type Array;
  type Float;
  type ScalarIndex;

  function mk_ix(i: ScalarIndex, j: ScalarIndex, k: ScalarIndex)
      : Index;

  function substepIx(u: Array, v: Array, u0: Array,
              u1: Array, u2: Array, c0: Float, c1: Float,
              c2: Float, c3: Float, c4: Float, ix: Index): Float;

  function substepIx3D(u: Array, v: Array, u0: Array,
              u1: Array, u2: Array, c0: Float, c1: Float, c2: Float,
              c3: Float, c4: Float, i: ScalarIndex, j: ScalarIndex,
              k: ScalarIndex): Float;

  axiom specializeSubstepRule(u: Array, v: Array, u0: Array,
              u1: Array, u2: Array, c0: Float, c1: Float, c2: Float,
              c3: Float, c4: Float, i: ScalarIndex, j: ScalarIndex,
              k: ScalarIndex) {
    assert substepIx3D(u, v, u0, u1, u2, c0, c1, c2,
                        c3, c4, i, j, k) ==
           substepIx(u, v, u0, u1, u2, c0, c1, c2, c3, c4,
                     mk_ix(i, j, k));
  }
};
```

Recall the original implementation of substepIx given in Listing 7. Every indexing operation of some array a in the resulting implementation of substepIx3D is now either of the form psi(mk_ix(i, j , k), a), or of the form psi(rotateIx(mk_ix(i, j, k), x, o), a) for some axis x and some offset o.

Listing 13 introduces a specialized psi function for 3D arrays. It does that by introducing three projection functions ix0, ix1, and ix2 on Indexes. General indexing operations of the form psi(mk_ix(i, j, k), a) are first specialized to expressions of the form psi(ix0(mk_ix(i, j, k)), ix1(mk_ix(i, j, k)), ix2(mk_ix(i, j, k)), a) by an application of specializePsiRule — which can then be reduced to psi(i, j, k, a) via three applications of reduceMakeIxRule.

Listing 13. Specializing calls to the indexing function ψ.

```
concept OFSpecializePsi = {
  type Index;
  type Array;
  type E;
  type ScalarIndex;

  /* 3D index projection functions */
  function ix0(ix: Index): ScalarIndex;
  function ix1(ix: Index): ScalarIndex;
  function ix2(ix: Index): ScalarIndex;

  /* 3D index constructor */
  function mk_ix(i: ScalarIndex, j: ScalarIndex, k: ScalarIndex)
    : Index;

  function psi(ix: Index, array: Array): E;
  function psi(i: ScalarIndex, j: ScalarIndex, k: ScalarIndex,
               array: Array): E;

  axiom specializePsiRule(ix: Index, array: Array) {
    assert psi(ix, array) == psi(ix0(ix), ix1(ix), ix2(ix), array);
  }

  axiom reduceMakeIxRule(i: ScalarIndex, j: ScalarIndex,
                         k: ScalarIndex) {
    var ix = mk_ix(i, j, k);
    assert ix0(ix) == i;
    assert ix1(ix) == j;
    assert ix2(ix) == k;
  }
}[ E => Float ];
```

We also want to call our specialized version of `psi` instead of the general one in expressions now of the form `psi(ix0(rx), ix1(rx), ix2(rx), a)` where `rx = rotateIx(mk_ix(i, j, k), x, o)`. For that purpose, we can apply the rewriting rules defined in Listing 14. These rewriting rules essentially unfold `rotateIx`. All the indexing operations in `substepIx3D` now use the specialized form of `psi`, and the scalar indices are either constants or of the form `(i + o)% s`, with `i` a scalar index, `o` an offset, and `s` the length of the relevant axis of the array.

Listing 14. A transformation rules to specialize the index rotation operation.

```
concept OFReduceMakeIxRotate = {
  use signature(OFSpecializePsi);

  type Axis;
  type Offset;

  function zero(): Axis;
  function one(): Axis;
  function two(): Axis;

  function rotateIx(ix: Index, axis: Axis, offset: Offset): Index;

  type AxisLength;

  function shape0(): AxisLength;
  function shape1(): AxisLength;
  function shape2(): AxisLength;

  function _+_(six: ScalarIndex, o: Offset): ScalarIndex;
  function _%_(six: ScalarIndex, sc: AxisLength): ScalarIndex;

  axiom reduceMakeIxRotateRule(i: ScalarIndex, j: ScalarIndex,
      k: ScalarIndex, array: Array, o: Offset) {
    var ix = mk_ix(i, j, k);
    var s0 = shape0();
    var s1 = shape1();
    var s2 = shape2();

    assert ix0(rotateIx(ix, zero(), o)) == (i + o) % s0;
    assert ix0(rotateIx(ix, one(), o)) == i;
    assert ix0(rotateIx(ix, two(), o)) == i;

    assert ix1(rotateIx(ix, zero(), o)) == j;
    assert ix1(rotateIx(ix, one(), o)) == (j + o) % s1;
    assert ix1(rotateIx(ix, two(), o)) == j;

    assert ix2(rotateIx(ix, zero(), o)) == k;
    assert ix2(rotateIx(ix, one(), o)) == k;
    assert ix2(rotateIx(ix, two(), o)) == (k + o) % s2;
  }
}
```

At this point, we can reintroduce padding using the rules previously defined in Listing 11, and renaming `schedulePadded` to `schedule3DPadded`, which will need to be pulled into scope from an external implementation somewhere down the line.

We decide to implement this function externally such that the array is always circularly padded at least once on each side of each axis — a decision made based on the width of the stencil. With that knowledge, we can completely eliminate the modulo operations in `substepIx3D`. Listing 15 defines the relevant rewriting rules.

Listing 15. Elimination of the modulo operations in the program.

```
// We suppose here that the amount of padding is sufficient across
// each axis for every indexing operation.
concept OFEliminateModuloPadding = {
  use signature(OFReduceMakeIxRotate);

  type Array;
  type Float;

  function psi(i: ScalarIndex, j: ScalarIndex, k: ScalarIndex,
               a: Array): Float;

  axiom eliminateModuloPaddingRule(i: ScalarIndex, j: ScalarIndex,
      k: ScalarIndex, a: Array, o: Offset) {
    var s0 = shape0();
    var s1 = shape1();
    var s2 = shape2();

    assert psi((i + o) % s0, j, k, a) == psi(i + o, j, k, a);
    assert psi(i, (j + o) % s1, k, a) == psi(i, j + o, k, a);
    assert psi(i, j, (k + o) % s2, a) == psi(i, j, k + o, a);
  }
}
```

Listing 16 shows how we apply the rewriting rules defined above using Magnolia's rewriting system to build a new program. Note that, as we are in the case when an implementation for `schedulePadded` is not in scope before the rules defined in `OFPad` are applied, we can replace the rewrite by a simple renaming — as we do in the example. To build a valid program, we also need to pull in scope external functions, such as the relevant schedules, and `psi`. These come from `ExternalNeededFunctions` in the example.

Listing 16. Putting all the rewriting rules together.

```
program SpecializedAndPaddedProgram = {
  use (rewrite
        (rewrite
          (rewrite
            (rewrite
              (generate OFSpecializeSubstepGenerator in
                        DNFImplementation)
            with OFSpecializePsi 10)
          with OFReduceMakeIxRotate 20)
        with OFPad[schedulePadded =>
                   schedule3DPadded] 1)
      with OFEliminateModuloPadding 10);

  use ExternalNeededFunctions; // pulling in psi, schedules, etc...
}
```

Table 2 gives an overview of the performance improvements brought by the rewriting rules. On the specific processor considered, padding does not seem to enable any significant speedup for our original implementation. Specializing the code to our specific 3D indexing function makes the code run faster in the unpadded case, and seems to allow a significant speedup from the unpadded case to the padded case — the generated code runs nearly twice as fast in that case.

Crucially, this performance improvement did not require any reimplementation of the core algorithm. Building our core algorithm generically allows us to introduce specialized underlying types and operations, once more information is known about our input data or the underlying hardware architecture. The Magnolia term rewriting engine then allows us to introduce new operations and to replace calls to existing concrete implementations with calls to other functions with possibly different argument lists.

	Unpadded case	Padded case
Non-specialized indexing	689.46	675.12
Specialized indexing	540.3	285.55

Table 2. Execution time (in seconds) of the 3-dimensional PDE solver Magnolia implementation compiled to C++ with specialized indexing and with or without padding. The code is compiled with gcc 10.2.0 with optimization level O3. The space dimensions are $512 \times 512 \times 512$ and the solver is run for 50 timesteps. The code is run on the ThunderX2 CN9980 CPU. In the padded case, each axis is padded circularly exactly once on both ends.

This is another twist of generic programming: *rewrite* and *generate* allow to replace operations (or combinations of operations) in a generic module with others that have potentially different argument lists — so long as we can describe the behavior of the former at all points in terms of calls to the new operation(s).

4 DISCUSSION AND RELATED WORK

We presented a methodology for solving the P^3 problem on existing and emerging architectures and applied it to the domain of array computations. Instead of developing one program to target n hardware architectures, we implement a single program, along with hardware-specific rewriting rules. By relating the high-level problem to a mathematical basis, we ensure that the set of optimization rules we implement is correct, and reusable for problems that can be embedded within the same formalism.

Magnolia gives developers the tools to write high-level, domain-specific compilers with custom optimization rules, and a custom target language. The ability to choose flexibly to which opaque building blocks a Magnolia program reduces allows the application of optimization rules at various abstraction levels, until the boundary between Magnolia and the external primitives implemented in the host language is reached. Our approach is centered around the idea of expressing generic algorithms independently from any particular schedule, i.e. independently from any hardware abstraction.

As we mentioned in Section 1.1, the term *schedule* as used throughout the paper originates in the work of Ragan-Kelley et al. on Halide (Ragan-Kelley et al., 2012). SPIRAL (Puschel et al., 2005) and Sequoia (Fatahalian et al., 2006) predate Halide, but make a similar distinction between an algorithm and its mapping to the underlying hardware architecture. Halide exposes a set of scheduling primitives from which developers can build their own schedules. TVM (Chen et al., 2018) follows this idea and extends Halide's set of scheduling primitives. The set of schedules that can be expressed in such systems is necessarily limited by the set of available scheduling primitives. Extending this set requires modifications to the language and its compiler, and is thus costly. Recent work by Liu et al. shows that carefully choosing high-level rewriting rules on schedules allows optimizing tensor programs beyond what is currently possible in these languages (Liu et al., 2022). In our system, schedules are fully specified by the developer. Compared to the approach taken by Halide or TVM, the developer has full control over how their computations are executed, but incur a higher implementation cost when no scheduling algorithm exists for their particular flavor of target hardware architecture. Adding "default" scheduling primitives to Magnolia as a convenience could improve the developer experience, and is therefore a consideration for future work.

MLIR (Lattner et al., 2021) makes heavy use of rewrite rules through the MLIR *PatternRewrite* infrastructure (Vasilache et al., 2022). Their design is influenced by LIFT (Steuwer et al., 2017, 2015), another programming language exploiting rewrite rules for high-performance array computations. In LIFT, the application of rewrite rules is automated by a stochastic search method. Hagedorn et al. extend LIFT specifically for optimizing stencil programs (Hagedorn et al., 2018). Such rewrite approaches are so far limited in that they do not always deliver high enough performance for real-world use (Hagedorn et al., 2020). This is in contrast to autoscheduling in Halide, which outperforms human experts on average (Adams et al., 2019). Automatic scheduling techniques are key to improving solutions to the P^3 problem, and are thus an important topic to further explore also for rewrite rules-based optimizers.

Approaches to optimization based on rewrite rules, such as the one presented here, can benefit from rewriting strategies, e.g. for localizing rewrites to only a particular chunk of the input program or for traversing the AST in a specific order. Kirchner gives a recent survey of strategic rewriting (Kirchner, 2015). Example of tools implementing such strategies include Maude (Clavel et al., 2007; Martí-Oliet et al., 2005, 2009) and Stratego (Visser, 2005). Hagedorn et al. introduce a functional approach to high-performance code generation based on rewriting strategies (Hagedorn et al., 2020): computations are expressed in the RISE programming language, and rewrite rules and strategies in the ELEVATE strategy language. Fu et al. (2021) later added a type system to ELEVATE to ensure statically that rewrites are composed correctly. As shown throughout the paper, our rewriting system today only provides the ability to apply sets of rewrite rules a certain number of times, in sequence. Given a rule $e_1 = e_2$, the sequence $e_1;\ e_1$ can be rewritten to $e_2;\ e_1$, but not directly to $e_1;\ e_2$. Such a transformation can be expressed today by applying the rule $e_1 = e_2$ twice, and then applying the opposite rule $e_2 = e_1$ once, but this is both embarassingly verbose and inefficient. Adding rewriting strategies to Magnolia will unlock those rewrites that are not easily accessible today, and thus further improve the system's code reuse capabilities. The implementation of Magnolia strategies is of particular interest, and fits into our larger project of exploring module transformations through the lens of *Syntactic Theory Functors* (Haveraaen and Roggenbach, 2020).

For future work, we also envision the implementation of an extension to the Magnolia rewriting system that supports conditional rewrite rules. Conditional equations can already be expressed in Magnolia, but the rewriting system is not yet able to exploit them.

Whether axioms constitute valid rewriting rules is verifiable by extending Magnolia with formal verification tools — insofar as the relevant properties that a program must satisfy can be derived from the stated axioms about its external building blocks. The properties asserted about externally implemented code can however only be assumed to hold, and constitute the trusted computing base of the whole program. Work on connecting verification tools with Magnolia's specification facilities is already underway, with encouraging results (Hamre, 2022).

ACKNOWLEDGEMENTS

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

Abts, D., Ross, J., Sparling, J., Wong-VanHaren, M., Baker, M., Hawkins, T., et al. (2020). Think fast: A tensor streaming processor (TSP) for accelerating deep learning workloads. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. 145–158. doi:10.1109/ISCA45697.2020.00023

Adams, A., Ma, K., Anderson, L., Baghdadi, R., Li, T.-M., Gharbi, M., et al. (2019). Learning to optimize halide with tree search and random programs. *ACM Trans. Graph.* 38. doi:10.1145/3306346.3322967

Bagge, A. H. (2009). *Constructs & Concepts: Language Design for Flexibility and Reliability*. Ph.D. thesis, Research School in Information and Communication Technology, Department of Informatics, University of Bergen, Norway, PB 7803, 5020 Bergen, Norway

Bagge, A. H., David, V., and Haveraaen, M. (2011). Testing with axioms in C++ 2011. *Journal of Object Technology* 10, 10:1–32. doi:10.5381/jot.2011.10.1.a10

Bagge, A. H. and Haveraaen, M. (2009). Axiom-based transformations: Optimisation and testing. In *Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, eds. J. J. Vinju and A. Johnstone (Elsevier), vol. 238, 17–33. doi:10.1016/j.entcs.2009.09.038

Basili, V. R., Briand, L. C., and Melo, W. L. (1996). How reuse influences productivity in object-oriented systems. *Commun. ACM* 39, 104–116

Burgers, J. M. (1948). A mathematical model illustrating the theory of turbulence. In *Advances in applied mechanics* (Elsevier), vol. 1. 171–199

Burrows, E., Friis, H. A., and Haveraaen, M. (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* (New York, NY, USA: ACM), ARRAY 2018, 59–66. doi:10.1145/3219753.3219761

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., et al. (2018). TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA: USENIX Association), OSDI'18, 579–594

[Dataset] Chetioui, B. (2021). The Magnolia Compiler. `https://github.com/magnolia-lang/magnolia-lang`. [Online; accessed 31-January-2022]

Chetioui, B., Abusdal, O., Haveraaen, M., Järvi, J., and Mullin, L. (2021). *Padding in the Mathematics of Arrays* (New York, NY, USA: Association for Computing Machinery). 15–26

Chetioui, B., Järvi, J., and Haveraaen, M. (2022). Revisiting language support for generic programming: the case of Magnolia. Under review

Chetioui, B., Mullin, L., Abusdal, O., Haveraaen, M., Järvi, J., and Macià, S. (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* (New York, NY, USA: Association for Computing Machinery), ARRAY 2019, 2–13. doi:10.1145/3315454.3329954

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et al. (2007). *All about Maude - a High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic* (Berlin, Heidelberg: Springer-Verlag)

Dehnert, J. C. and Stepanov, A. A. (1998). Fundamentals of generic programming. In *Selected Papers from the International Seminar on Generic Programming* (Berlin, Heidelberg: Springer-Verlag), 1–11

Fatahalian, K., Horn, D. R., Knight, T. J., Leem, L., Houston, M., Park, J. Y., et al. (2006). Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA: Association for Computing Machinery), SC '06, 83–es. doi:10.1145/1188455.1188543

Frakes, W. B. and Succi, G. (2001). An industrial study of reuse, quality, and productivity. *Journal of Systems and Software* 57, 99–106

Fu, R., Qin, X., Dardha, O., and Steuwer, M. (2021). Row-polymorphic types for strategic rewriting. *CoRR* abs/2103.13390

Gibbons, J. (2006). Datatype-generic programming. In *Proceedings of the 2006 International Conference on Datatype-Generic Programming* (Berlin, Heidelberg: Springer-Verlag), SSDGP'06, 1–71

Goguen, J. A. and Burstall, R. M. (1984). Introducing institutions. In *Logics of Programs*, eds. E. Clarke and D. Kozen (Berlin, Heidelberg: Springer Berlin Heidelberg), 221–256

Gregor, D., Järvi, J., Siek, J., Stroustrup, B., Reis, G. D., and Lumsdaine, A. (2006). Concepts: linguistic support for generic programming in C++. *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* , 291–310doi:http://doi.acm.org/10.1145/1167473.1167499

Hagedorn, B., Lenfers, J., Kundefinedhler, T., Qin, X., Gorlatch, S., and Steuwer, M. (2020). Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.* 4. doi:10.1145/3408974

Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., and Dubach, C. (2018). High performance stencil code generation with lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (New York, NY, USA: ACM), CGO 2018, 100–112. doi:10.1145/3168824

Hamre, H.-C. (2022). *Automated Verifications for Magnolia Satisfactions*. Master's thesis, The University of Bergen

Haveraaen, M. and Roggenbach, M. (2020). Specifying with syntactic theory functors. *Journal of Logical and Algebraic Methods in Programming* 113, 100543. doi:https://doi.org/10.1016/j.jlamp.2020.100543

Kirchner, H. (2015). *Rewriting Strategies and Strategic Rewrite Programs* (Cham: Springer International Publishing). 380–403. doi:10.1007/978-3-319-23165-5_18

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., et al. (2021). Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

Lemire, D., Kaser, O., and Kurz, N. (2019). Faster remainder by direct computation: Applications to compilers and software libraries. *Softw., Pract. Exper.* 49, 953–970. doi:10.1002/spe.2689

Liu, A., Bernstein, G. L., Chlipala, A., and Ragan-Kelley, J. (2022). Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.* 6. doi:10.1145/3498717

Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2005). Towards a strategy language for maude. *Electron. Notes Theor. Comput. Sci.* 117, 417–441

Martí-Oliet, N., Meseguer, J., and Verdejo, A. (2009). A rewriting semantics for maude strategies. *Electron. Notes Theor. Comput. Sci.* 238, 227–247. doi:10.1016/j.entcs.2009.05.022

Mullin, L. (1988). *A Mathematics of Arrays*. Ph.D. thesis, Syracuse University

Mullin, L. and Thibault, S. (1994). *Reduction Semantics for Array Expressions: The Psi Compiler*. Tech. Rep. CSC 94-05, Dept. of CS, University of Missouri-Rolla

Mullin, L. M. R. and Jenkins, M. A. (1996). Effective data parallel computation using the psi calculus. *Concurrency - Practice and Experience* 8, 499–515. doi:10.1002/(SICI)1096-9128(199609)8:7⟨499::AID-CPE230⟩3.0.CO;2-1

Musser, D. R. and Stepanov, A. A. (1988). Generic programming. In *Symbolic and Algebraic Computation, International Symposium ISSAC'88, Rome, Italy, July 4-8, 1988, Proceedings*, ed. P. M. Gianni (Springer), vol. 358 of *Lecture Notes in Computer Science*, 13–25. doi:10.1007/3-540-51084-2_2

Nazareth, D. L. and Rothenberger, M. A. (2004). Assessing the cost-effectiveness of software reuse: A model for planned reuse. *Journal of Systems and Software* 73, 245–255

Puschel, M., Moura, J., Johnson, J., Padua, D., Veloso, M., Singer, B., et al. (2005). Spiral: Code generation for dsp transforms. *Proceedings of the IEEE* 93, 232–275. doi:10.1109/JPROC.2004.840306

Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31. doi:10.1145/2185520.2185528

Sannella, D. and Tarlecki, A. (1996). Mind the gap! Abstract versus concrete models of specifications. In *Mathematical Foundations of Computer Science 1996, 21st International Symposium, MFCS'96, Cracow, Poland, September 2-6, 1996, Proceedings*, eds. W. Penczek and A. Szalas (Springer), vol. 1113 of *Lecture Notes in Computer Science*, 114–134. doi:10.1007/3-540-61550-4_143

Steuwer, M., Fensch, C., Lindley, S., and Dubach, C. (2015). Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (New York, NY, USA: Association for Computing Machinery), ICFP 2015, 205–217. doi:10.1145/2784731.2784754

Steuwer, M., Remmelg, T., and Dubach, C. (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* (IEEE Press), CGO '17, 74–85. doi:10.1109/CGO.2017.7863730

Tang, X. and Järvi, J. (2015). Axioms as generic rewrite rules in C++ with concepts. *Science of Computer Programming* 97, 320–330. doi:https://doi.org/10.1016/j.scico.2014.05.006. Object-Oriented Programming and Systems (OOPS 2010) Modeling and Analysis of Compositional Software (papers from EUROMICRO SEAA'12)

Vasilache, N., Zinenko, O., Bik, A. J. C., Ravishankar, M., Raoux, T., Belyaev, A., et al. (2022). Composable and modular code generation in mlir: A structured and retargetable approach to tensor compiler construction doi:10.48550/ARXIV.2202.03293

Visser, E. (2005). A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation* 40, 831–873. doi:https://doi.org/10.1016/j.jsc.2004.12.011. Reduction Strategies in Rewriting and Programming special issue

Wolfe, M. (2021). Performant, portable, and productive parallel programming with standard languages. *Computing in Science Engineering* 23, 39–45. doi:10.1109/MCSE.2021.3097167

6.2 A CUDA implementation

To showcase the versatility of the approach to generic code transformations and PDE solvers given in section 6.1, a couple of different versions of the solver were implemented. Among these were OpenMP and CUDA [36] implementations, to highlight how one could parallelize key parts of the algorithm to achieve a significant decrease in computing time on different hardware. In this section we take a closer look at two iterations of the solver implemented in CUDA C++ to leverage the SIMD parallel processing approach utilized by Nvidias line of GPUs.

CUDA is an API developed by Nvidia to facilitate general purpose programming on its lines of GPUs. It is designed to work with high-performance languages such as C, C++ and Fortran. The GPU does not share most of its memory with the CPU, and as such we need to introduce a separation between CPU(host) code and GPU(device) code. In CUDA C++ this is achieved using annotations to mark function as either callable from CPU only, GPU only or both.

```

// executes on the CPU
__host__ int addTwo_cpu(int a);
// executes on the GPU
__device__ int addTwo_gpu(int a);
// can be executed on both CPU and GPU
__host__ __device__ int addTwo_gpu_cpu(int a);
// entry point from CPU to GPU
__global__ void kernel();
```

Listing 6.1: CUDA Annotations

Program execution starts on the CPU, and in order to access the device side code we need a way to call device side code from the CPU. This is done via global functions, also called kernels. Functions annotated as global can be invoked from the CPU and run device side code. In kernel calls you specify the number of blocks of memory and threads per block the GPU has available for computation. Since CUDA 5.0, device side kernel launches are allowed, paving the way for multiple layers of non-uniform device side memory allocation.

Implementation [1]

The backend for the PDE solver presented in section 6.1 is implemented in C++. In the standard pipeline, magnoliac generates C++ code for the defined Magnolia concepts and

[1] The code for both implementations are publicly available online [28, 29].

programs, then compiles the generated code and the user-provided backend using GCC `g++`. CUDA-annotated C++ code is not compatible with ISO standard C++, and as such we we in need of a compiler with CUDA support. For this implementation Nvidias proprietary NVCC was chosen. It is also a feature-complete C++ compiler up to C++17, so for our purposes we replaced `g++` with NVCC for compilation of all C++ code.

Results

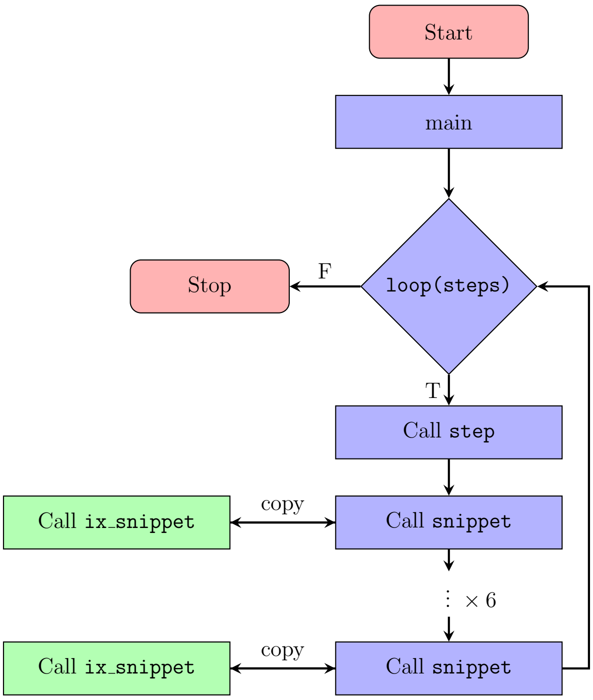

Figure 6.1: Dataflow between CPU(blue) and GPU(green), 1st iteration of the solver.

The first iteration of runtime results fell short of expectations. One would expect much faster speeds from a cutting-edge GPU like the Nvidia A100. The times in Table 6.1 are after 10 steps of the solver, whereas the benchmarks in section 6.1 are after 50 steps.

`real`	1m12.703s
`usr`	0m36.333s
`sys`	0m34.199s

Table 6.1: 10 steps of the CUDA PDE solver with array dimensions 512^3, ran on a Nvidia Volta A100/80GB. Timed in bash with `time`.

Profiling the binary produced by NVCC, we identified the main causes for the bad performance.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Name
45.8	21931544537	840	26108981.6	`cudaMemcpy`
38.7	18537571949	720	25746627.7	`cudaMalloc`
15.4	7377761005	60	32944.5	`cudaDeviceReset`

This is highly ineffective, to stay on the GPU as long as possible

Table 6.2: Snippet of `gpumemtimesum` result generated from `nsys profile <pde.bin>`.

We see that calls to `cudaMemcpy` and `cudaMalloc` account for over 80% of the execution time. CUDA memory allocations and copies are expensive operations. In the first iteration of the implementation, memory is allocated and copied between host and device memory 6 times per solver step. Figure 6.1 depicts the dataflow between CPU and GPU in the first iteration of the CUDA implementation.

Improvements

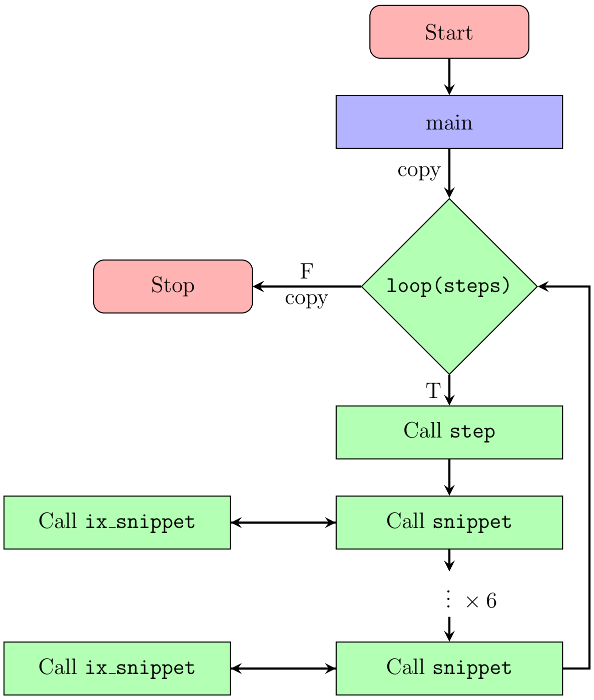

Figure 6.2: Improved dataflow between CPU(blue) and GPU(green)

Following the implementation from section 6.1, we theorized that we could reduce the number of copies between host and device to a single copy to GPU before solver execution and a single copy to CPU after, resulting in a significant reduction in execution time. Due to time constraints, this improved implementation was not completed before the paper submission deadline. Figure 6.2 depicts the theorized improved dataflow between GPU and CPU.

Remark on earlier work

The work presented in Burrows et al. includes runtime experiments for two CUDA versions implemented in the process. Analyzing the source code for these implementation provided us with valuable insights in how to approach the the problem. An important difference between the approach presented in Burrows et al. and the the one presented in this thesis is the density of the kernel computations. Previous implementations opted for implementing each operation as separate kernel calls, resulting in low density of computation. In other words, quite large parts of available GPU cores remain unused per computation. With this in mind, the approach taken in section 6.1 relies on a dense, inlined `step` function presented in Chetioui et al. With proper memory management, the theorized improvement depicted in Figure 6.2 should leverage the cores available to us on the GPU to a larger degree.

Chapter 7

Future Work & Conclusion

7.1 Future Work

Even though the thesis process draws to an end, there are multiple places to start if one would wish to continue the work discussed here. An natural first step would be to continue the work on the MoA API. Improving the existing code should be prioritized, as the current state of the code base suffers from a focus on producing compilable code on a tight deadline. In particular, this meant sacrificing time that could have been spent fleshing out the array specification. The API in its current form is also prioritizing a limited subset of the theory to explore the specific API proposed by Burrows et al. Continuing to investigate the parts of the ψ-calculus not covered in this thesis would come as a natural continuation. I believe that a logical first extension of the API would be to fully specify and implement padding, taking it a step closer to its intended state of usability. One could also move towards extending it to coverONF, both specified in the generic API and as a step toward observing how implementations on different hardware may differ.

Although developed independently of existing Magnolia library packages, once in an acceptable state, integration of the API with the standard library may be useful for future Magnolia projects.

Completion of the improved CUDA implementation discussed in section 6.2 could add weight to the argument presented in section 6.1 that generating performant code from generic languages like Magnolia can compete with existing compiler tools. Chetioui et al. have already demonstrated promising results for this approach in another domain.

7.2 Conclusion

In this thesis we have explored the MoA calculus and how it can serve as a foundation for generic multiarrays. We have presented relevant background theory, creating an arena for discussion around generic programming and API design. We have showed that one can abstract hardware specific details away, without losing the ability to target specific architectures. Implementing a subset of the MoA theory in Magnolia, we have gained knowledge about how we can leverage the Magnolia type system to go from generic specifications and concrete implementations while retaining type safety.

As hardware improves, it is critical to keep software portable and maintainable while retaining performance. This principle is followed by leveraging concepts from generic programming to abstract away from hardware specific implementations. We have seen that Magnolia allows us to reason about programs at a higher level, capturing this philosophy. The modularity gained by designing software with a clear separation between specification and implementation also serves the purpose of forcing developers away from the typical monolithic structure of large systems, where dependencies are often difficult to swap out. Haveraaen argues that allocating more time to domain exploration is a cost effective way to design software long term, and we have been given a taste of this through the lens of re-use mechanisms in Magnolia such as renamings. We hope to see a gradual shift in the software design philosophy, towards a more modular future for programming.

Glossary

ψ-reduction The process of transforming an array expression into an equivalent expression only using the ψ operator.

AEP Annual Energy Production.

API Application Programming Interface.

APL A Programming Language (APL), an array programming language developed in the 1960s.

BLDL Bergen Language Design Laboratory.

CPU Central Processing Unit.

CUDA Parallel computing platform developed by Nvidia for their line of GPUs.

DNF Denotational Normal Form.

FLOPS Floating Point Operations Per Second.

GCC GNU Compiler Collection.

GPGPU General-purpose Programming on Graphical Processing Units.

GPU Graphical Processing Unit.

HPC High-performance computing.

Magnolia Magnolia is a programming language based on the theory of institutions.

magnoliac A Magnolia compiler under active development at BLDL.

MoA A Mathematics of Arrays.

MPI Message Passing Interface.

NVCC Nvidia CUDA Compiler.

ONF Operational Normal Form.

OpenBLAS Open Basic Linear Algebra Subprograms.

OpenCL Open Computing Language.

OpenMP Open Multi-Processing, shared-memory multiprocessing programming API.

PDE Partial Differential Equation.

SIMD Single Instruction, Multiple Data.

SMT Satisfiability Modulo Theories.

VLSI Very Large Scale Integration.

Bibliography

[1] Ole Jørgen Abusdal. Transformations for array programming. Master's thesis, University of Bergen, 2020.

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

[3] Eva Burrows, Helmer André Friis, and Magne Haveraaen. An array api for finite difference methods. In *roceedings of the 5th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* Association for Computing Machinery, 2018. doi: 10.1145/3219753.3219761.
URL: `https://doi.org/10.1145/3219753.3219761`. 59–66.

[4] J. Carlson, J.A.C.A.J.A. Wiles, J.A. Carlson, A. Jaffe, A. Wiles, Clay Mathematics Institute, and American Mathematical Society. *The Millennium Prize Problems.* American Mathematical Society, 2006. ISBN 9780821836798.

[5] Benjamin Chetioui. magnoliac: A magnolia compiler, December 2020.

[6] Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià. 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.* Association for Computing Machinery, 2019. doi: 10.1145/3315454.3329954.
URL: `https://doi.org/10.1145/3315454.3329954`. 2–13.

[7] Benjamin Chetioui, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Lenore Mullin. Padding in the mathematics of arrays. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array*

Programming. Association for Computing Machinery, 2021. doi: 10.1145/3460944.3464311.
URL: `https://doi.org/10.1145/3460944.3464311`. 15–26.

[8] Benjamin Chetioui, Jaakko Järvi, and Magne Haveraaen. Revisiting language support for generic programming: The case of magnolia. Under review, 2022.

[9] Benjamin Chetioui, Marius Kleppe Larnøy, Jaakko Järvi, Magne Haveraaen, and Lenore Mullin. Pˆ3 problem and magnolia language: Specializing array computations for emerging architectures. Under review, 2022.

[10] James Dehnert and Alexander Stepanov. Fundamentals of generic programming. volume 1766, pages 1–11, 01 1998. ISBN 978-3-540-41090-4. doi: 10.1007/3-540-39953-4_1.

[11] Python Software Foundation. array - efficient arrays of numeric values.
URL: `https://docs.python.org/3/library/array.html`. [Online; accessed 31.05.22].

[12] Jeremy Gibbons. Datatype-generic programming. In *Proceedings of the 2006 International Conference on Datatype-Generic Programming*, SSDGP'06, page 1–71, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540767851.

[13] Joseph A. Goguen and Rod M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, January 1992. ISSN 0004-5411. doi: 10.1145/147508.147524.
URL: `https://doi.org/10.1145/147508.147524`.

[14] John L. Gustafson. *Moore's Law*, pages 1177–1184. Springer US, Boston, MA, 2011. ISBN 978-0-387-09766-4. doi: 10.1007/978-0-387-09766-4_81.
URL: `https://doi.org/10.1007/978-0-387-09766-4_81`.

[15] John L. Gustafson and Lenore M. Mullin. Tensors come of age: Why the ai revolution will help hpc, 2017.

[16] G. Hains and L. M. R. Mullin. Parallel functional programming with arrays. *The Computer Journal*, 36(3):238–245, 01 1993. ISSN 0010-4620. doi: 10.1093/comjnl/36.3.238.
URL: `https://doi.org/10.1093/comjnl/36.3.238`.

[17] Hans-Christian Hamre. Automated verifications for magnolia satisfactions. Master's thesis, The University of Bergen, 2022.

[18] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2.
URL: `https://doi.org/10.1038/s41586-020-2649-2`.

[19] Kristoffer Haugsbakk. Program transformations in magnolia. Master's thesis, The University of Bergen, 2017.

[20] Magne Haveraaen. Domain engineering the magnolia way. In Alexander K. Petrenko and Andrei Voronkov, editors, *Perspectives of System Informatics*, pages 196–210, Cham, 2018. Springer International Publishing. ISBN 978-3-319-74313-4.

[21] S. Hoyer and J. Hamman. xarray: N-D labeled arrays and datasets in Python. *Journal of Open Research Software*, 5(1), 2017. doi: 10.5334/jors.148.
URL: `https://doi.org/10.5334/jors.148`.

[22] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. doi: 10.1109/MCSE.2007.55.

[23] ISO. *Draft International Standard ISO/IEC 1539-1:2004(E): Information technology — Programming languages — Fortran Part 1: Base Language.* May 2004.
URL: `https://wg5-fortran.org/N1601-N1650/N1601.pdf`.

[24] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C.* December 2011.
URL: `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`.

[25] Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., USA, 1962. ISBN 0471430145.

[26] Neo Shih-Chao Kao and Tony Wen-Hann Sheu. Development of a finite element flow solver for solving three-dimensional incompressible navier–stokes solutions on multiple gpu cards. *Computers & Fluids*, 167:285–291, 2018. ISSN 0045-7930. doi: https://doi.org/10.1016/j.compfluid.2018.03.033.

[27] I G W Krabben, M P van der Laan, M. Koivisto, T J Larsen, M M Pedersen, and K S Hansen. Why curved wind turbine rows are better than straight ones. *Journal of Physics: Conference Series*, 1256(1):012028, jul 2019. doi: 10.1088/1742-6596/1256/1/012028.
URL: `https://doi.org/10.1088/1742-6596/1256/1/012028`.

[28] Marius Larnøy. mariuslarnoy/magnolia-lang at cuda-dynamic, .
URL: `https://github.com/mariuslarnoy/magnolia-lang/tree/cuda-dynamic`. [Online; accessed 03.05.22].

[29] Marius Larnøy. mariuslarnoy/magnolia-lang at cuda, .
URL: `https://github.com/mariuslarnoy/magnolia-lang/tree/cuda`. [Online; accessed 03.05.22].

[30] Marius Larnøy. mariuslarnoy/magnolia-lang at naturalnumbers, .
URL: `https://github.com/mariuslarnoy/magnolia-lang/tree/naturalnumbers`. [Online; accessed 25.05.22].

[31] Marius Larnøy. mariuslarnoy/magnolia-lang at moa-ndim, .
URL: `https://github.com/mariuslarnoy/magnolia-lang/tree/moa-ndim`. [Online; accessed 20.05.22].

[32] Lars Moastuen. Real-time simulation of the incompressible navier-stokes equations on the gpu. Master's thesis, The University of Oslo, 2007.

[33] Lenore Mullin and Scott Thibault. A reduction semantics for array expressions: The psi compiler. 1994.

[34] Preferred Networks. Cupy.
URL: `https://cupy.dev/`. [Online; accessed 12.05.22].

[35] NumPy. Case study: First image of a black hole.
URL: `https://numpy.org/case-studies/blackhole-image/`. [Online; accessed 12.05.22].

[36] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. Cuda, release: 10.2.89, 2020.
URL: `https://developer.nvidia.com/cuda-toolkit`. [Online; accessed 03.05.22].

[37] C. Ostrouchov and L. Mullin. python-moa, 2019.
URL: `https://github.com/Quansight-Labs/python-moa`. [Online; accessed 03.05.22].

[38] Mads M. Pedersen, Paul van der Laan, Mikkel Friis-Møller, Jennifer Rinker, and Pierre-Elouan Réthoré. Dtuwindenergy/pywake: Pywake. Feb 2019. doi: 10.5281/zenodo.2562662.

[39] Wileam Phan, Jeremie Vandenplas, Arjen Markus, and Lenore Mullin. Mathematics of arrays library for modern fortran, 2021.
URL: `https://github.com/wyphan/moa-fortran`. [Online; accessed 03.05.22].

[40] Lenore Marie Restifo Mullin. *A Mathematics of Arrays*. Thesis, Syracuse University, 1988.

[41] Riccardo Riva, Jaime Liew, Mikkel Friis-Møller, Nikolay Dimitrov, Emre Barlas, Pierre-Elouan Réthoré, and Arvydas Beržonskis. Wind farm layout optimization with load constraints using surrogate modelling. *Journal of Physics: Conference Series*, 1618(4):042035, sep 2020. doi: 10.1088/1742-6596/1618/4/042035.
URL: `https://doi.org/10.1088/1742-6596/1618/4/042035`.

[42] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-201-72914-8.

[43] John E. Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science & Engineering*, 12(3):66–73, 2010. doi: 10.1109/MCSE.2010.69.

[44] M P van der Laan, S J Andersen, and P-E Réthoré. Brief communication: Wind speed independent actuator disk control for faster aep calculations of wind farms using cfd. 2019. doi: 10.5194/wes-2019-27.

[45] Paul Veers. Three-dimensional wind simulation. Sandia National Laboratories, 01 1988.

[46] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C. J. Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert-Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin

Probst, Jörg P. Dietrich, Jacob Silterra, James T. Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vinícius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, Yoshiki Vázquez-Baeza, and SciPy 1.0 Contributors. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature Methods*, 17(3):261–272, Mar 2020. ISSN 1548-7105. doi: 10.1038/s41592-019-0686-2. **URL:** `https://doi.org/10.1038/s41592-019-0686-2`.

[47] J.A. Vitulli, G.C. Larsen, M.M. Pedersen, S. Ott, and M. Friis-Møller. Optimal open loop wind farm control. *Journal of Physics: Conference Series*, 1256(1):012027, jul 2019. doi: 10.1088/1742-6596/1256/1/012027. **URL:** `https://doi.org/10.1088/1742-6596/1256/1/012027`.

[48] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010. doi: 10.25080/Majora-92bf1922-00a.