# Dimension and shape invariant programming: the implementation and application

Manal Ezzat Helal
*The American University in Cairo AUC*

**The American University in Cairo**

**School of Sciences and Engineering**

# Dimension and Shape Invariant Programming: The Implementation and The Application

A Thesis Submitted to

*Department of Computer Science*

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

*The degree of Master of Science*

by

*Manal Ezzat A. Helal*

B.Sc. Computer Science, AUC, Jan. 1995

Under the Supervision of **Dr. Ahmed Sameh**

January 2001

The American University in Cairo

# DIMENSION & SHAPE INVARIANT PROGRAMMING: THE IMPLEMENTATION AND THE APPLICATION

A Thesis Submitted by Manal Ezzat A. Helal to

The Department of Computer Science

January 2001

in partial fulfillment of the requirements for

the degree of Master of Science

has been approved by

Dr. Ahmed Sameh

Thesis Committee Chair/Adviser [illegible]

Affiliation The American University in Cairo

Dr. Abdel Badie Salem

Thesis Committee Reader/examiner A. Salem

Affiliation Ain Shems University

Dr. Awad Khalil

Thesis Committee Reader/examiner Khalil

Affiliation Professor, AUC

Dr. Hoda Hosny

Thesis Committee Reader/examiner H. Hosny

Affiliation Computer Science, AUC

| | | | |
|---|---|---|---|
| [illegible] | 01.10.2001 | [illegible] | January 30, 2001 |
| Department Chair/ Program Director | Date | Dean | Date |

The American University In Cairo

## ABSTRACT

**DIMENSION AND SHAPE INVARIANT PROGRAMMING: THE IMPLEMENTATION AND THE APPLICATION**

by Manal Ezzat A. Helal

Supervised by: Dr. Ahmed Sameh

This thesis implements a model for the shape and dimension invariant programming based on the notation of the Mathematics of Arrays (MOA) algebra. It focuses on dimension and shape invariance implementation, and their effect in parallel computing. A new design for the MOA notation is implemented that eliminates the need for another PSI-compiler, or a language extension to functional programming languages. The MOA notation is designed as a library of Application Programming Interfaces (APIs), contains object oriented classes implemented in C++. The library executes array operations correctly, and is expected to enhance the performance invariant of dimension and shape. To implement these APIs, the mathematical equations of the original notation were analyzed and sometimes simplified to become more comprehensible to implement from the programming point of view, and some more operations were added. The APIs reduce the erroneous loops starts, strides, and stops used by programmers in the traditional handling of multi-dimension arrays. The library defines the dimension and shape of the arrays at runtime; and gives the source code of the problem in hand better chances to be automatically parallelized.

The MOA library testing tool developed and implemented in this thesis, can be used by mathematicians and computer arithmetic researchers to translate high level arithmetic functions in applications like image processing, video processing, fluid dynamic, … etc. to the MOA notation, utilizing its benefits. An image-processing tool is implemented using this new MOA library, proving the correctness of the design on 2D-array application, where image operations are expressed concisely in the source code and easily manipulated on the conceptual level. Image processing transformations, filtering and detections are implemented. Video processing operations like transformations on the AVI Frames after decomposing them, and motion detection scheme are implemented using the MOA library, to prove the correctness of the library on a 3D-array application. Also, the parallelisation factors inherent in the MOA library design are discussed in terms of shape polymorphism, MOA parallel architecture, data redistribution, and Tiling algorithms, in relation to the MOA notation. Furthermore, pipelining with MOA has been investigated. In addition to the above experiments, a hardware implementation of the MOA APIs was implemented using VHDL on Renoir as a package, and simulated using ModelSim. Performance analysis is conducted in terms of general benefits of programming invariant of shape and dimension as designed in this thesis, which is open to further analysis based on the application domain.

To my Parents

# ACKNOWLEDGMENTS

If it were not for the blessings of Allah (SWT), this work wouldn't have been done in the first place. Allah is the first to be praised and thanked for everything at all times.

I gratefully thank Dr. Ahmed Sameh for his sincere support, his guidance based on deep knowledge, and patience through all the years that I have worked with him. His educational supervision and belief in this work and in me, are the main drives behind this work.

I thank all my professors at the Computer Science Department in the American University in Cairo for teaching me in the undergraduate and graduate courses. The wealth of knowledge that I have received during my years of study is displayed in the pages of this thesis.

I thank Dr. Lenore Mullin, Computer Science Department, University of Albany, State University of New York, for her kind support and for answering my questions, through e-mail, about the MOA notation.

I thank all my friends at AUC, and outside, for their sincere help and support. I also thank Engineer Abdel-Moniem Abou Rehab for helping in the traditional implementation of image processing. My thanks also go to Azza Khatab for the graphic design of the defense presentation. I also appreciate the help I received from all my lab friends throughout the Renoir implementation and the C++ implementation, including Aya Saad, Rania Mameesh, Soha S. Zaghloul, and all the others that I may have forgotten to mention here, but I should not forget to thank the lab engineers for their continuous support.

Finally, I would like to express my deep gratitude for being a daughter of a great father Mr. Ezzat A. Helal, who, beside being a loving and supporting father, is an Information Technology Consultant, with very long experience in this field. I have always used his knowledge to achieve more in my career and in my studies. Without his support in this thesis, the video processing wouldn’t have been done, and without his involvement in the rest of the implementation, discussing and testing, this work would have been incomplete.

I also thank my mother for her moral support, encouragement, and patience with me. I thank my whole family for praying for me and for putting up with my long hours of studying.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# *Chapter 1*
# INTRODUCTION

## 1.1 Introduction

Arrays are the core of almost any complex computational problem. The Von Neumann computer architecture has been always more appropriate to array processing because of its linear memory structure and its addressing methods (absolute address + offset + index, base registers, offset registers, index registers). Also, its instruction set is considered a perfect base to express arrays. However, researchers addressed the Von Neumann bottleneck that is caused by processors accessing words of memory one at a time [47]. This bottleneck caused programs to handle data in small units processed individually; then gathered together into large conceptual units. It was desirable to encourage computer architects to design a machine organization that handles data in large conceptual blocks. Solving this problem, researchers developed functional languages that deal with large data items as single units, such as APL and LISP. These programming languages implement solutions more succinctly than others. This brevity does not come always with clarity. However, in many cases, these languages are more amenable to parallel processing.

In addressing real problems, programmers usually have the tendency to implement in flat programs, using complicated nested loops and GOTOs [11]. Efficiency necessitates employing higher-level concepts of programming languages, addressing real problem (frequently used metrics, image processing transformations, … etc.). One of the benefits of writing an efficient code, (among other benefits like conceptual clarity, expressiveness, … etc.), is the possibility of automatically parallelizing the sequential code. Handling boundary constraints, variables scopes, and static memory layout, are considered overheads for the programmer designing a flat program that can be transformed automatically to parallel programs.

Efficient parallel scheduling and load balancing requires the analysis of the shape and dimension of the arrays. So, the notion of shape analysis and decomposition has been researched in various aspects during the last decade. This thesis relates the shape and dimension invariance programming to the Mathematics of Arrays (MOA) notation - based on the Psi-Calculus – an algebra of arrays - that is designed to provide a technology for scientists and engineers, to do large-scale scientific and engineering computations effectively, utilizing parallel computers [7]. Previously, research was taking the direction of providing compilers that translate the code written using the MOA notation to C or Fortran languages, or providing language extensions to existing functional languages, saving the programmer the effort of writing the code of the array computation. In this research, a Dynamic Link Library (DLL) providing Application Programming Interfaces (APIs) is designed to be imported to the user's application, to use the MOA notation directly. This implementation achieves the preliminary requirements for a functional C++ language.

A MOA testing tool based on this library is designed having a complete user interface for the MOA notation. This tool can be used in the mathematical analysis process for testing and verifying equations to easily map traditional computer arithmetic problems to the MOA notation, explaining the MOA functionality, and a test bench for the correctness of the implementation. Then, the MOA library is used in representing image structures for performing image-processing functions using the new notation as a 2-Dimension illustration. Then, a 3-Dimension illustration of the use of the MOA library is presented through a Video (AVI File) being read into MOA structures, and operated on it. Some other factors of the full utilization of the notation has been discussed, like the explicit parallelisation factors in the MOA library design, discussed in terms of shape polymorphism, data redistribution algorithms, tiling algorithms, and pipelining.

The design of the library is simple and could be migrated easily to any platform or language. It is implemented in Visual C++ as a Windows DLL and tested on Visual Basic and Java. It is also implemented on hardware using VHDL packages on Renoir, and simulated using ModelSim. The hardware implementation is an important step towards the current direction of hyper-programming, mixing hardware programming with software to reach the ultimate performance possible using new faster hardware technology. Special-purpose hardware (hardware accelerators) could be deigned for different scientific and engineering applications based on array computations.

The performance of applications based on MOA is compared to the traditional methods non-quantifiably, and the effect of dimension invariance on performance as dimension increase was analyzed. Analysis covered also the effect of the MOA design on the source code, parallel computing, multi-processor environment, and employing hardware accelerators.

## 1.2 Thesis Motivation

The main motivation behind this thesis is to achieve conceptual and functional array processing invariant of dimension and shape, in C++ as found in some functional languages. Another main objective is to prove the practicability of the MOA algebra, by providing simple APIs that are added to any project in a sort of an external compiled library. A programmer can use these APIs to apply computations on arrays of arbitrary dimensions, without handling the erroneous nested loops starts, stops and strides, and the transformation equations, that require extensive testing to reach the simplest expressions.

The previous effort in this field presented several attempts to implement array operations invariant of dimensions and shapes, using the notations and equations presented in [1]. However, these attempts lacked the formality of a simple algorithm, and were quite complicated and not reduced to its simplest programming steps, and were implemented using laboratory research compilers that are in many cases not used in real-life applications. The research was directed towards designing compilers to parse the MOA notation to produce a C or Fortran languages source code, or to provide recommendations about language extensions to functional programming languages to encapsulate the array operations in their compilers. So, this thesis is motivated by formalizing these algorithms and simplifying the MOA algebraic equations to be easily implemented as a Library of APIs that can be called directly from any application and used to define the dimension and the shape of the array at run time by the calling applications, or the end-user.

Also, presenting a valid physical application (image and video processing) is a main motivation to encourage the implementation of real-life problems based on this notation. These application examples proved the correctness of the design of the MOA library, and the enhanced performance of the dependence on this programming model, making use of the parallel computing that is made easier with MOA, and the chances of having hardware accelerators for the problem in hand.

## 1.3 Thesis Objective

The main and first objective of this thesis is to support the MOA notation equations by implementing them as APIs. The second objective is to test the

MOA APIs correctness and performance on a 2-Dimension application, using images as a 2-Dimension MOA array structures, and applying the image processing operations on them based on MOA notation. A Third objective is to apply the previous tests on a 3-Dimension application, using Videos (AVI files) applying video processing functions on them.

A fourth objective is to discuss how parallelism and pipelining factors in the MOA notation are more obvious than in the traditional implementation. A fifth objective is to implement in hardware the MOA APIs, to provide hardware accelerators for the applications that can be implemented using this notation. A final objective is to conduct a general analysis of the performance of applications depending on the MOA notation implemented in this thesis.

## 1.4 Thesis Outline

The Thesis is divided into 8 chapters. Chapter two surveys the array programming theories and languages. The Third chapter presents the Mathematics of Arrays (MOA) literature review, background information, and a comprehensive description and examples of the notation in details, with some modification to the original ones. Chapter four discusses the implementation details of the model class implemented, and design issues for the MOA Library implementation. Chapter five discusses the Image Processing operations as 2-Dimension application of the MOA Library, and the Video Processing operations as a 3-Dimension application of the MOA Library. Chapter six discusses the parallel computation, arrays pipelining, and implementation of the MOA library in VHDL using Renoir. Chapter seven presents the performance analysis of the conducted experiments. Finally, chapter eight states the conclusion and outlines the future work.

# *Chapter 2*

# PROGRAMMING WITH ARRAYS

## 2.1 Introduction

Array-based programming started with APL Language as will be described later. The More's array theory (AT) was formalized evolving from the data concepts of APL, extending it to include nested arrays and systematic handling of the second order functions. AT is a formal description of nested arrays in a first order logic. It is a mathematical model of the ways in which the orthogonal arrangement of material bodies interacts with their hierarchy nesting. The theory stems out of geometric experience with finite collections. Originally, the motivation to formalize array theories and languages was to create a notation for subscripting sequences, vectors, matrices, and higher dimensional objects.

Arrays properties are defined in the array theory as: [23][27]

- arrays are complex bodies composed of mass (data) and energy (operations),
- multidimensional rectangular objects with items locations described by 0-origin addressing scheme,
- arrays have arbitrary numbers of dimensions including zero (denoting scalars),
- data elements are laid out along orthogonal axes (dimensions),
- every axis has a length (extent or upper bound),
- the extents of the array's axes forms a shape vector,
- items of arrays can be themselves arrays, allowing for nesting,

- arrays are heterogeneous, since there is no type constraints,
- operations on these bodies are simple, expressed as relationships between operations that are simple enough to be remembered and communicated,
- any operation applies to an element in an array, applies to others,
- the result of an operation is always an array subject to further operations,
- the formulae of brevity and universality of arrays are simplified by expressing the array transformations as successive operations on arrays. To preserve a pattern, array operations behave like operations on simple objects,
- absence of axes and nesting is a manifestations of a boundary-case presence of a structure,
- arrays are not a generalization of simple objects; simple objects (numbers, truth values, characters, … etc.) are special cases of arrays,
- array operations include: *valence*, *shape*, *tally*, *pick*, *tell*, *choose*, *raise*, *reshape*, *link*, take, takeright, drop, dropright, *list*, *sublist*, *transpose*, *fuse*, *rows*, *cols*, *split*, *mix*, *blend*, *each*, *pack*, *flip*, *reduce*, *leftreduce*, *cart*, *inner*, *outer*, *rest*, *front*, *append*. Second order functions – Transformers, unary pervasive operations (*abs*, *sin*, … etc.), and binary pervasive operations (*minus*, *plus*, … etc).
- the basic operations of the arrays are total in the sense that they apply for all arrays,
- the array operation are closed, in the sense that they return arrays for further operations as pointed above.

Array theory, similar to set theory, is concerned with the concepts of nesting, aggregation and membership. Array theory is the mathematics of nested, rectangularly arranged collections of data objects, having a special position relative to other data objects in a collection. So, collections of zero or more elements are held at positions in a rectangular arrangement along zero or more axes. The items of arrays are themselves arrays. Nesting is an important concept in array theory that gives the theory much of its expressive power; it is the ability to aggregate arbitrary elements in an array, by having the objects of a collection be collections themselves. [17]

Array theory conceptualize an algebra of:

- self-containing scalars,
- empty arrays having nested structures,
- strand (nested vector) notation,
- the application of operations to nested arrays, with scalar operations acting pervasively,
- pictorial representation of such objects by nested boxes,
- explicitly defined primitives operators, such as *each*,
- and the application of arbitrarily defined operators to any operation.

In array theory, arrays are constructed by three different methods: *hitch*, *reshape* and *void*. Selection in done by four main selectors: *first*, *rest*, *shape*, and *list*, where the combinations of these correspond to *cons*, *head*, and *tail*.

In traditional imperative scalar languages, arrays are accessed with the specified index of its defined dimensions, as much as these dimensions are declared. To traverse a multidimensional array, the programmer need to take care of the starts, stops, and strides of nested loops of as many as the dimensions of the array. A two-dimension array will require two nested loops in the specified lower and upper bounds of each dimension, and so forth. To retrieve a slice from the array, the programmer must traverse through all entries in the array, to locate the required slice, or portion, and may copy it in another location in memory to apply some processing on it, and rewrite it back to the array. The programmer will be overloaded with defining relations between the indices in the multi-dimension array, and testing the efficiency of the resulting relation equations. This requires a lot of tracing as much as the complexity of the problem.

## 2.2 Functional Programming

In applicative languages, it is not easy to define and implement arrays under the applicative semantics, this is due to three reasons:

1. arrays are handled as an expression result that makes the allocation of storage and filling in values alien to applicative semantics;
2. illegal array definition occurs in case of recursive definition; arrays must be defined at most once;
3. modifying arrays require copying them, while the cost of copying large arrays is prohibitive.

In procedural languages, arrays are simpler than in functional languages. Non-sequential processing requires a functional programming language (category of declarative languages based on mathematical notion of functions) that is statically scoped and based on single expressions, which are executed by evaluating the expression. Most functional languages use arrays (such as nested lists, tuples, trees) as a mean for structuring data. Functional Languages implement arrays in a variety of approaches. Usually arrays are represented as a flat contiguous area in linearly addressable memory. Another approach would represent arrays to be passed item-by-item, along data-flow links. Also, Sparse arrays can be stored as lists of items, or hash-tables. One of the approaches can be mapping arrays to fine-grain SIMD machines, such as a Connection Machine [14]. Moreover, Finite portions of infinite arrays can be represented using data-flow or sparse array techniques. Relating functional languages to lambda calculus, realized arrays such that progress was achieved compromising the four factors of 1: adhering to first principles, 2: clear semantics, 3: obvious pragmatics, and 4: high performance.

Scientific computations like fluid dynamic problems or weather forecast require enormous memory capacities operating on large data structures, like lists of records or arrays. So, programming languages and computer systems suitable for these applications need to efficiently manage space and time consumption. Programming techniques, clever algorithms, and highly optimized compilers provide means to create code that avoids temporaries, and immediate re-claim of memory after finishing using it.

Techniques like allocating exactly the required memory size in declaration, de-allocating on leaving the declaration scope, the multiple assignment concept that allows to overwrite arrays (reusing previously allocated space), indexing in array access allowing the iteration in loops with the required starts, stops, and strides, are all functional techniques that allow the efficient management of space and time for scientific applications. Side effects are observed like the naming problem for the multiple assignments, and scope problems in splitting large programs into concurrently executable parts, due to shared variables. The programmer can make concurrency explicit, or the compiler can deduce logical dependencies on variable assignments or indices. [7]

Functional programming languages provide the solution for those side effects, where variables are just placeholders for values, applying the concept of "single assignment rule", avoiding side effects in over-writing. However, that was at the expense of the efficiency of array-operations, due to the fact that they consume their operand objects, creating new result objects, rather than update existing ones. This rewriting of large arrays, for any mere restructure or a few entries being changed, is very costly in terms of memory space, and processing time. So the functional languages needed to integrate array concepts to allow operations to overwrite their operands, achieving concise programming and efficient implementation. With the Mathematics of Array (MOA), arrays can be viewed as conceptual entities that high-level functions can operate on to restructure and transform values. This method can totally decrease the use of the Do-Loops, since traversing and accessing arrays can be done in specific order using a particular primitive, handled by the system or the compiler. The idea is to map restructured arrays into existing arrays, instead of rewriting them, by moving pointers using offsets. This technique requires algebraic simplification to the array complex expressions, avoiding the creation of intermediate arrays.

Since the PSI-calculus is an algebra of array, it is based on a core set of essential array operations defined in terms of dimensionality, shapes, and indexing such as psi and gamma functions [1]. Moreover, the omega operator ($\Omega$) allows for distributing functions over the components of the array argument, to compute the values of elements of the resultant array. MOA extended the PSI Calculus to include high level constructs for array restructuring and value transformation, making it a full subsystem for array processing. Initially, the research was directed towards integrating MOA notation into functional languages. Challenging problems for an array processing subsystem were the correct treatment of boundary conditions, the mapping into other arrays of different mesh sizes, and designing a program that works invariant of dimensionality and shape of actual array supplied as an argument.

The advantages of integrating the PSI-calculus into a functional programming language lie specifically in the reduction of temporary variables created, less amount of the error-prone loops created for a multi-dimension array, folding loops with the same starts, stops, and strides into one, and freeing the code from redundancies. Thus the programmer using the MOA constructs will be able to produce enhanced program design that is abstracted from problem-specific parameters, with confidence in program correctness, and without sacrificing performance.

In the following general overview of array languages and processing environments, considering the popularity of the scalar languages like C, Pascal, Fortran, ADA, … etc., (where operations apply only on single values, and array operations require indexing and looping which is tedious and error prone), it is desirable to investigate how we can integrate the array operations found in array-based languages, to these imperative scalar languages as higher order constructs. This thesis implements array operations invariant of dimension and shape in C, as APIs in a library, based on the MOA (Psi-Calculus), and tests the applications domains that could benefit from this implementation, as well as the parallelisation, pipelining, and hardware implementation of the implemented library.

## 2.3 Array Languages and Environments Survey

Array languages have been around since the 1960. The following are some array languages that provide array operations and indexing functions:

***APL Language***: It stands for A computer Programming Language [15]. It is the first array language with powerful functional subsets that look like algebraic formulas and contain an enormous character set with concise syntax; it is famous for its succinctness, and the indefinite nesting of functions (functions evaluations are parameters for other functions). It provides a number of powerful built-in functions that are applied to multi-dimensional arrays [of arrays [of arrays …]] (denoting nesting of arrays) as easy and simple as they are with atomic items. It emerged in a number of dialects with a standard one called ISO Standard APL; the ACRON compiler is a combination of two of these dialects. APL is based on Lambda calculus like in dropping and taking elements from an array. It includes recursive data structures (boxed arrays), the rank adverb; it employs dynamic scoping, in which a called verb (function) can alter the values of the variables of its caller, and which are not localized within the called verb. On the other hand, ACRON used static scoping. APL supports using integer vectors as subscripts for arrays, by computing the outer product of the indices in each dimension to identify the elements referenced. APL is cursed for being un-maintainable, due to considering it an incomprehensible programming language (an anarchist programming language).

***Fortran*** introduced the array concepts, with static arrays of a fixed type and with a limited number of dimensions. The array-handling notation in Fortran is closer to the mathematical form. Fortran allows referencing elements using triple or slice notation to reference subsets of elements. Fortran 90 allows zero-sized arrays and explicit shape definition. Both languages (APL and Fortran) allow ignoring the subscript to denote the whole array. Fortran provides some higher level array constructs like: whole array assignment, WHERE to mask an assignment, sub-objects to allow for segmentation, Dot Product, Matrix Multiplication, array reduction, array inquiry, array construction, reshaping, and manipulation operations. Fortran 90 can support object-oriented concepts. Parallelism is much less obvious in original source code of Fortran, but it is intuitively clear.

***Algol***: stands for (ALGOrithmic Language) [27]; it is a high level language designed for programming scientific computation. It allows for an arbitrary number of dimensions. It determines the size of the array dynamically at block entry, where the subscript range is specified by variables to define the size of the array while allocating storage for it.

***Nial Language***: Array theory was formalized for development through the Nial (Nested Interactive Array Language), which is a multi-paradigm programming language combining features from APL, Lisp, and FP [23]. It implements fast powerful array operations, based on the mathematical treatment of nested arrays, with procedural language and familiar control structures, containing a rich set of primitives to rapidly develop a loop-free data-driven algorithm. Nial supports the heterogeneous arrays. ***Q'Nial*** is a highly portable interpreter for Nial, used for applications in decision support, knowledge based systems, scientific computations, and data analysis.

***VCODE*** is an intermediate vector language, where all data are vectors, including the shape vector. It compiles to a variety of parallel machines. Q'Nial and VCODE are two languages that allow multidimensional arrays to be declared in terms of a flat array and a shape, allowing operation over a variety of shape classes – This is how the MOA model is implemented in this thesis in C++.

***Sisal Array Language***: it is a general-purpose applicative language that stands for "Streams and Iterations in a Single Assignment Language" [13]. It is a high-performance parallel programming language that exposes implicit parallelism through data independence (guaranteeing determinate results), based on mathematical principles. It is a single assignment data-flow language, which support arrays. Sisal generates code for multi-processors architectures. Performance of Sisal programs is better than equivalent Fortran programs compiled using automatic vectorizing and parallelizing software, and run comparably to hand-written parallel Fortran codes. This is because of the simplicity of the functional programming model. It employs single assignment data-flow, and is designed such that arrays and array operations are included in its language definition to target the large-scale scientific applications to run on parallel-supercomputers. In Sisal, modifications of elements, and mean copying of array operands, are possible while the cost of copying of large arrays is prohibitive.

***Falafel Array Language***: it stands for (Functional Array Language for Experiments in Laziness) [14]; it is an experimental functional language that is not an array-based language, with first class arrays. Falafel relates array types by strong structured polymorphic typing, and structural inheritance of functions applied from supertypes to subtypes. It employs the lazy evaluation concept, and provides means for partial definitions of arrays. It supports only the homogenous arrays, not the heterogeneous ones. Falafel adopts the sophistication of array theory and simplifies some areas.

***NESL Language***: is a parallel functional language that supports nesting. It translates to VCODE. It is based on two main concepts which are the Nested data parallelism, and the language based performance model. The first concept resulted in data parallelism, concise code, and hence, readiness for irregular algorithms on trees and graphs or sparse matrices. The second concept gives a formal way to calculate the work and depth of a program. It is used for teaching, algorithm experimentation (planar Delaunay triangulation, the N-body problem, graph connectivity, graph separators, support tree conjugant gradient), and algorithm animation.

***Haskel Language***: it is a lazy non-strict functional language [25], where arrays are defined as a basic type, called non-strict monolithic arrays that can contain undefined elements, and monolithic arrays can define all its elements at once, using array comprehensions. Some of array functions in Haskel are: *array*, *listArray*, *accumArray, bounds, indices, elems, assoc, operator //* (copy array), *operator !* (subscripting), *accum* (batch updating), *amap* (mapping on the elements), *ixmap* (mapping on the indices), … etc. Since, Array mapping in Haskel is supported through type classes (also in ***Gofer Language***), therefore the mapping algorithm must be supported by the user. Unlike the ***Charity language***, the algorithm of mapping can be inferred from the type.

***Gofer Language*** stands for GO(od) F(or) E(quational) R(easoning), not like the Gopher (internet distributed information delivery system) [25]. It is an interpreter language based on the Haskell report. It supports lazy-evaluation, higher order functions, polymorphic typing, pattern matching, and overloading. It runs on several machines like PCs, Ataris, Amigas, Unix-based systems, and Apple Macintosh.

***Charity language*** is a categorical programming language that is based on the theory of strong categorical data types (inductive and co-inductive data types). Programming in Charity is expressed by folding (catamorphisms) for the inductive and unfolding (anamorphism) for the co-inductive data types. It supports lazy-evaluation, higher order functions, all computation terminates up to user input. It is elegant and suitable for teaching, researching, development for reasoning about programs, transformations and verifications.

***Lucid Language***: it was introduced in the mid seventies, originally with implicit one dimension (referred to informally as *time*) [28]; then later it was enhanced to allow for the existence of different dimensions and even temporary dimensions. It includes operators like first, next, fby, realign, and rank.

***FISh Language***: is an array-based, polymorphic language for array programming that supports both functional and imperative styles [24]. It includes higher-order array operations such as mapping and folding (reduction). Its array type constructor is used to represent all finite dimensional arrays, so that polymorphic functions may be applied to arrays of any number of dimensions. Shape analysis is used to determine the storage requirements of each array, so that boxing of array entries is unnecessary. Also, the shape  analysis produces an accurate portable parallel programs costing model . FISh shows dramatic speedups in array programming compared to other higher-order, polymorphic languages.

***MATLAB language***: it is a high-performance interpreted technical computing environment, designed for engineering and scientific applications, with trusted mathematics and powerful numeric computing functions, and graphical representation of data. Its application domain is wide; it applies for producing solutions to complex systems of equations, modeling, simulations, prototyping, fuzzy logic implementation,  digital signal processing, image processing, partial differential equations, curving applications, aerospace applications, decision support systems, data analysis and exploration and visualization, … etc. It provides efficient matrix and vector computations, and graphical plotting of 2-dimensional and 3-dimensional structures.

***ZPL***: it stands for the Z-level Programming Language [25] [21]; it is an efficient parallel APL language used by scientists and engineers that provides a performance model allowing for reasoning about parallel overheads, and is faster than other high level programming languages. It is characterized by good scalability, platform portability (machine independent overlapping of computation with communication optimization using its Ironman interface), message passing equivalent to C. It compiles to ANSI C, and then is compiled to the targeted machine (currently to Cray T3E, IBM SP2, Intel Paragon, SGI Origin, Sun Enterprise, High Performance Clusters, and Unix workstations). It can interface with sequential C code, and provides access to scientific libraries. It supports region-programming concept (indexing arrays by regions; i.e. slicing to compute on boundary conditions), directions, sophisticated array structures, shattered control flow. It provides a performance model that makes users define how well their programs will run on parallel machines, characterized by what-you-see-is-what-you-get (WYSIWYG) property. It is not a shared memory language, however it preserves the minmum advantages of shared memory. it eliminates the temporaries created both by the programmer and by the compiler, specially in arrays. This is done by operating on the entire array by wrapping each line in enough loops to iterate over all elements of the array.

## 2.4 Summary

In table 2-1, a summary of some basic properties of arrays are compared among the surveyed languages is presented. A hyphen in a cell in the table means that the information wasn't available at the time of editing of this table, since these languages & environments were surveyed theoretically from the available references, without experimentations. From table 2-1 and the discussions in this chapter, the need to include array operations in an imperative language with a larger domain of application, becomes more obvious. This is what this thesis hopes to achieve.

Table 2-1: Summary of Surveyed Array Languages and Environments

| **Property** / **Lang.** | **Arrays & Array Operations** | **Nesting** | **Hetro/ Homogenous** | **Dynamic Size** | **Parallel Computing** |
|---|---|---|---|---|---|
| **APL** | Array-based | Functions & Arrays nesting | Homogenous | Dynamic | Parallel Language |
| **Fortran** | Supports Arrays | Can support inheritance, but wastes efforts for source code duplication | Homogenous | Fixed | Intuitive |

Table 2-1: Summary of Surveyed Array Languages and Environments - Continued

| | | | | | |
|---|---|---|---|---|---|
| **Algol** | Array-based | Allow Nesting | Homogenous | Dynamic | - |
| **Nial** | Array-based | - | Heterogeneou s | Dynamic | - |
| **VCODE** | Vector-language | - | Homogenous | Dynamic | Compiles to parallel machines |
| **Sisal** | Support arrays | - | Homogenous | - | Parallel Language |
| **Falafel** | First class arrays | Structured inheritance | Homogenous | Partial Array definition | - |
| **NESL** | Supports arrays. | Nested Data Parallelism | Homogenous | - | Parallel Language |
| **Haskell** | Incorporates arrays | - | Homogenous | - | - |
| **Gofer** | Supports arrays | - | Homogenous | - | - |
| **Charity** | Supports arrays | - | Homogenous | - | - |
| **Lucid** | Array-based | - | Homogenous | - | - |
| **FISh** | Array-based (mapping, reduction, static shape, and efficient array access) | - | Homogenous | - | GoldFish supports parallel combinators . |
| **MATLAB** | Supports Arrays | - | Homogenous | - | - |
| **ZPL** | Supports indexing array by regions | - | - | - | Parallel APL language |

*Chapter 3*

# MATHEMATICS OF ARRAYS

## 3.1 Introduction

Since arrays are an important data structure in many applications, many researchers worked on enhancing array manipulation techniques to achieve better performance. Mathematics of Arrays (MOA) is the algebra of Reduction Rules defined to operate on arrays of arbitrary dimension. In the paradigm of MOA, arrays of arbitrary dimension and shape are the basic data structure of all types. Scalars are considered to be arrays of 0 dimension, vectors are considered to be arrays of 1 dimension, matrices are considered to be arrays of 2 dimensions, and up to any arbitrary number of dimensions and shapes.

Earlier studies on arrays went in two directions. The first direction dates back to the 19th century when Joseph Sylvester, an English Mathematician, developed "The Construction Theory of Partitions" [34], incorporating ways to partition an integer into a finite arbitrary number of parts. He used diagrams resembling arrays to visualize the odd and even partitioning. He was the first to introduce the term Matrix, and developed the theory of determinants, and the theory of invariants, and studied the quarternians (a special kind of matrix).

Sylvester collaborated with Arthur Cayley who was investigating the linear transformations on matrices [35], introducing matrix transpose and inversion and matrix methods for the geometry of n-dimensional space. Both of them developed jointly an algebraic approach to differential geometry. Their work was referenced in the work of Whitehead [48] discussing the Universal Algebra in the 1960's.

The second direction started by Iverson, who investigated similar ideas and reformulated the previous ideas [36], developed a set of operations for a programming language that used arrays. Iverson generated the APL language formulating partitions and properties of array expressions. Abrams in 1970s studied the APL operations [37], recognizing the relation between the arrays transformations and indexing operations. Then, he simplified the array expressions, relating inner and outer product to scalar operations and reductions, and discussed the lazy evaluation concept (evaluation of expressions should take place only when the values are needed). He built an APL machine, developing the basis of the Mathematics of Arrays (MOA).

Guibas and Wyatt [38] extended Abrams work, to include the outer product. Perlis, Minter, and Miller [39] included these concepts in an efficient implementation of an APL compiler. Tu and Iverson later introduced the ranking operation, using the rank as an argument to operations [40]. Tu also described the semantics of the array expressions in terms of the Psi ($\psi$)-Calculus, linking them to the functional programming languages.

Later on, array operations notation was used to describe and simulate computer architectures [41] (for example System/360 architecture was described using APL notation, and also hardware components like multipliers and adders) and as a basis for the Register Transfer Language (RTL). Examples of the RTLs based on array operations notation are the AHPL by Hill written in Fortran [42]. Also APLSIM is a circuit design language, which provides SPICE-like simulations implemented in C in a modified APL environment [43].

Reynolds, Gerhart, Pichumani, and Stabler verified array expressions using inductive assertion techniques [44] [45] [46], automating the verification of shape constraints. Some investigations found that APL had too many irregularities and can't perform systematic verifications, suggesting that in order to completely automate the array operations in a language, it should be functionally semantic and arrays should be described in terms of their shapes. On the other hand, others proved that APL can be used to automate the array operations, proving properties about RTLs, which appear frequently in VLSI designs, in shifting, or rotating registers, taking, or dropping segments of registers, and comparing registers segments with stored memory locations.

In the late 80's and 90s, Mullin later performed a complete arithmetic analysis to identify the array operations that have a common theme [1], and the usage of axioms and definitions to describe theorems for VLSI design verification. She identified the common theme to be the indexing function, which uses the array structuring information (dimensionality and shape). Later on, the Psi-Compiler was designed [4] [5], which parses the notation in MOA to generate C or Fortran programs, which were also introduced, then implemented in hardware using the Chameleon board (described later in this thesis). Further work discussed the Reduction semantics of the array expressions in the Psi-compiler, and the Data Parallel computation using the Psi-Calculus, and Functional Languages extensions to encapsulate the Psi-Calculus subsystem for array operations [6] [7].

That was how the MOA evolved to become a notation used for verifications of computer architectures design, and array descriptions in any physical problem.

## 3.2 MOA preliminary axioms and Definitions

MOA is a notation based on the Psi-Calculus, which corresponds to the array theory discussed in chapter 2 [27]. It encapsulates the same array properties discussed earlier, only differs in that its data elements are numeric scalars that can be extended to any homogenous scalar type. Scalars are defined to be arrays of no dimension and with empty shape vectors. The MOA algebra is a set of operations proven to be useful for scientific algorithms. All operations are based on shapes and indexing functions. The properties of the mathematics of arrays notation are as follows:

- The method of evaluating expressions only when the values are needed (Lazy Evaluation).
- The method of simplifying array expressions, to perform fewer operations, consuming less space.
- The method of updating operands (large arrays) in place, avoiding the creation of temporaries.
- The ability to shift or rotate, taking or dropping segments of registers and comparing register segments to stored memory locations.
- The fact that the array operations that are defined to access arrays in large adjacent blocks, fit nicely in the cache structure and execute faster, because caches take advantage of nearest memory locations, even on a uni-processor.
- The fact that in a multi-processor environment, sub-array operations can be scheduled over the multiple processors of the shared-memory, MIMD architectures.

As these potential speed-ups could be fully realized in a parallel architecture, it was also proven that they could be partially realized on a uni-processor. Hence, a combination of an accurate model of memory caches, concurrency of sub-array operations, and low process communication was seen to achieve good performance [1].

Arrays are read in row major order. The notations used through out this thesis are summarized in table 3-1.

Table 3-1: MOA Notation

| | |
|---|---|
| $\vec{v}$ | Denoting vectors, |
| $\xi$ | For arbitrary array, |
| [ | The opening parenthesis for a dimension in an array, |
| ] | The closing parenthesis for a dimension in an array, |
| < | The opening parenthesis for a vector, |
| > | The closing parenthesis for a vector, |
| abs | Returns absolute value of argument on right hand side. |
| div | Integer divide, returns the quotient, |
| mod | The remainder, from an integer division, |
| op | Any arithmetic operation [+, -, *, /, <, >, max, min, and, or, xor] |

The following sub-sections describe the MOA operations. The functionality of each operation is described, the equation is stated, and some examples are demonstrated. These operations are divided into seven categories: the measurement operations, the indexing operations, the array constructing operations, the scalar operations, axis transpositions and transformations operations, reduction and scan operations, and finally the higher order constructs. The diagram in figure 4-13, found in the next chapter in section 4.4, illustrates these categories and all the MOA functions, conceptualizing the MOA notation as implemented in this thesis.

### *3.2.1 The Measurement Operations*

**Dimensionality**: $\delta\xi$ a unary prefix operation returning the rank of the array:

$$\delta\xi^{n} \equiv n \qquad \textbf{Equation 3-1}$$

This function returns zero for a scalar, 1 for a vector, 2 for a matrix, and so forth.

**Shape:** $\rho\xi$ any array $\xi$ has a shape vector, denoted by $\rho\xi$**,** whose entries are the lengths of each of $\xi$**'**s dimensions.

$$\rho\xi = \left\langle i_0, \quad i_1, \quad \dots, \quad i_{(\delta\xi)-1} \right\rangle \qquad \textbf{Equation 3-2}$$

The following examples are defined to be used in all the operations to come. They are selected to be comprehensive test cases, to illustrate the behavior of the notation invariant of dimension and shape**:**

$$\vec{v}_1 \equiv \langle 1 \quad 2 \quad 3 \quad 4 \quad 5 \rangle$$

$$\rho\vec{v}_1 \equiv 5$$

$$\delta\vec{v}_1 = 1$$

$$\xi_1^2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\rho\xi_1^2 = < 3 \quad 2 >$$

$$\xi_2^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

$$\rho\xi_2^2 = < 3 \quad 4 >$$

$$\xi_3^2 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

$$\rho\xi_3^2 = < 3 \quad 2 >$$

$$\xi_4^2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

$$\rho\xi_4^2 = < 4 \quad 4 >$$

$$\xi_5^2 \equiv \begin{bmatrix} 12 & 3 & 22 & 7 & 11 \\ 5 & 17 & 9 & 13 & 6 \\ 8 & 24 & 18 & 4 & 25 \\ 23 & 19 & 15 & 1 & 21 \\ 2 & 14 & 20 & 16 & 10 \end{bmatrix}$$

$$\rho\xi_5^2 = < 5 \quad 5 >$$

$$\xi_1^3 = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} & \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \end{bmatrix}$$

$\rho\xi_1^3 = <3 \quad 2 \quad 2>$

$$\xi_1^4 = \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \\ \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} & \begin{bmatrix} 13 & 14 \\ 15 & 16 \end{bmatrix} \\ \begin{bmatrix} 17 & 18 \\ 19 & 20 \end{bmatrix} & \begin{bmatrix} 21 & 22 \\ 23 & 24 \end{bmatrix} \end{bmatrix}$$

$\rho\xi_1^4 = <3 \quad 2 \quad 2 \quad 2>$

**Total:** $\tau\xi$: is a unary operation that returns the number of elements of an array $\xi$, by multiplying the components of its shape.

$$\tau\xi \equiv \pi(\rho\xi^n) \qquad \textbf{Equation 3-3}$$

Examples using the arrays defined in section 3.2.1:

$\tau\xi_1^2 \equiv \pi(3 \quad 2) \equiv 6$

$\tau\xi_1^3 \equiv \pi(3 \quad 2 \quad 2) \equiv 12$

**Ravel**: rav $\xi$ collapses the array to one dimension, i.e. it returns the flat vector of a multidimensional array.

Given array $\xi^n$ with shape $\vec{v}$, and valid subscript vector $\vec{i}$:

$$\xi^n[i_0, \ \dots \ i_{n-1}] \equiv (rav\xi)[(((((i_0 \ \times \ v_1) + \ i_1) \ \times \ v_2) \ + \ \dots) \times \ v_{n-1}) \ + \ i_{n-1}]$$

**Equation 3-4**

Examples using the arrays defined in section 3.2.1:

$rav\xi_1^2 \equiv < 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 >$

$rav\xi_1^3 \equiv < 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11 \quad 12 >$

**Pi**: $\pi\xi$ is a unary operation that returns the product of the array components of any arbitrary vector $\vec{v}$, or the component elements of a multidimensional array $\xi$.

$$\pi\xi \equiv \prod_{i=0}^{((\rho\xi)[0]-1)} (rav \quad \xi)[i]$$ **Equation 3-5**

Example using the above defined arrays:

$\pi\xi_1^2 \equiv 720$

### *3.2.2 The Indexing Operations*

**Psi–** $\vec{i}\,\psi\xi^n$: This is the main indexing operation; it is a binary operation which accesses the selected $\vec{i}$ from $\xi^n$.

$$\vec{i}\,\psi\xi^n = \xi^n[i[0],\dots,i[n-1]]$$ **Equation 3-6**

Examples using the arrays defined in section 3.2.1:

$< 1 \quad 1 > \psi \xi_1^2 = 4$

$< 2 \quad 1 \quad 1 > \psi \xi_1^3 = 12$

$< 1 \quad 0 > \psi \xi_1^3 = < 5 \quad 6 >$

$< 2 \quad 1 > \psi \xi_1^4 = \begin{bmatrix} 21 & 22 \\ 23 & 24 \end{bmatrix}$

**Gamma**: $\gamma(\vec{a};\vec{b})$ is a mapping function that returns the index in the raveled (flattened) array $\xi$ of shape $\vec{b}$, denoted by the vector index $\vec{a}$.

Given an array of dimension j, and shape $\vec{b}$, gamma of index vector $\vec{a}$ is:

$$\gamma(< a_0,\ldots,a_{j-1} >;< b_0,\ldots,b_{j-1} >) \equiv a_{j-1} + b_{j-1} \times \gamma(< a_0,\ldots,a_{j-2} >;< b_0,\ldots,b_{j-2} >)$$

**Equation 3-7**

So this is equivalent to:

$$\vec{i}\,\psi\xi \equiv (rav\xi)[\gamma(\vec{i};\rho\xi)]$$

Examples:

$\gamma(< 2 \quad 1 >, < 3 \quad 2 >)\xi_1^2 \equiv 5$

$\gamma(< 2 \quad 1 \quad 0 >, < 3 \quad 2 \quad 2 >)\xi_1^3 \equiv 10$

Note: the shape is included in the argument list. No need to pass a complete array, just its shape is enough to compute the index in the flat array from the multidimensional index. This same note applies for the next function.

**Gamma Inverse**: $\gamma'$ this function returns the index vector of a scalar index in the raveled array. It takes the output of the previous gamma function to return its input.

$\gamma'(n;<b_0,...,b_{n+1}>) \equiv \gamma'$ (n div d; <$b_0$, …, $b_n$>), n mod d **Equation 3-8**

where d = $\pi < b_1,...,b_n >$.

So that:

$y(\gamma'(n;\vec{b});\vec{b}) \equiv n$

Examples:

$\gamma'(5;<3 \quad 2>) \equiv <2 \quad 1>$

$\gamma'(10;<3 \quad 2 \quad 2>) \equiv <2 \quad 1 \quad 0>$

### *3.2.3 The Array Constructing Operations*

**Reshape**: $\vec{s}\rho\xi$ this is a binary infix operation that constructs or reconstructs arrays.

For vectors:

$\vec{i}\ \psi(\vec{s}\rho\vec{v}) \equiv \vec{v}[\gamma(\vec{i};\vec{s}) \text{ mod } \tau\vec{v}]$ **Equation 3-9**

For arrays:

$\vec{s}\rho\xi \equiv \vec{s}\rho(rav\xi)$ **Equation 3-10**

Examples using the arrays defined in section 3.2.1:

$$<2 \quad 2> \rho\xi_1^2 \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$<2 \quad 3 \quad 2> \rho\xi_1^3 \equiv \begin{bmatrix} [1 \quad 2] & [7 \quad 8] \\ [3 \quad 4] & [9 \quad 10] \\ [5 \quad 6] & [11 \quad 12] \end{bmatrix}$$

**Catenate**: $\xi_1 \oplus \xi_2$ we will be giving this operation the sign $\oplus$. This operation concatenates two arrays over the required dimension, given that the shapes along the other axes are equal.

For vectors:

$$\rho(\vec{x} \oplus \vec{y}) \equiv (\rho\vec{x}) + (\rho\vec{y})$$ **Equation 3-11**

$$<i> \psi(\vec{x} \oplus \vec{y}) \equiv \begin{cases} <i> \psi\vec{x} & if \quad 0 \le i < \tau\vec{x} \\ <i - \tau\vec{x}> \psi\vec{y} & if \quad \tau\vec{x} \le i < (\tau\vec{x}) + \tau\vec{y} \end{cases}$$ **Equation 3-12**

For arrays, Catenate of array $\xi_1$ with shape vector $\vec{a}$ and $\xi_2$ with shape vector $\vec{b}$ on the X^th dimension, given that other dimensions bounds are equal on both arrays:

$$\rho(\xi_1 \oplus \xi_2) \equiv <\rho\xi_1[0], \ldots, (\rho\xi_1[X] + \rho\xi_2[X]), \ldots, \rho\xi_1[\delta\xi_1 - 1]>$$ **Equation 3-13**

$$<\vec{i}> \psi(\xi_1 \oplus \xi_2) \equiv \begin{cases} <\vec{i}> \psi\xi_1 & if \quad 0 \le i[X] < a[X] \\ <i[0], \ldots, i[X] - a[X], \ldots i[d-1]> \psi\xi_2 & if \quad a[X] \le i[X] < a[X] + b[X] \end{cases}$$

**Equation 3-14**

Examples using the arrays defined in section 3.2.1:

$$\xi_1^2 \oplus \xi_3^2 \equiv \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \\ 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}, \quad when \quad X = 1$$

resulting shape vector = <6 2>.

$\xi_1^2 \oplus \xi_3^2 \equiv \begin{bmatrix} 1 & 2 & 1 & 2 \\ 3 & 4 & 3 & 4 \\ 5 & 6 & 5 & 6 \end{bmatrix}, \quad when \quad X = 2$, resulting shape vector = <3 4>.

**Iota**: $\iota \quad n$ is a vector whose entries are the integers from 0 to n-1, for n ≥ 0.

$\rho(\iota \quad n) \equiv < n >$ **Equation 3-15**

$(\iota \quad n)[j] \equiv j$ **Equation 3-16**

Example:

$(\iota \quad 5) \equiv \langle 0 \quad 1 \quad 2 \quad 3 \quad 4 \rangle$

### *3.2.4 The Scalar Operations*

**Point-wise Extension**: $\xi_1 \quad op \quad \xi_2$ This kind of operation takes two arrays of the same dimension and shape and applies a point wise operation on their components. This means that elements of the same coordinate positions are treated as scalars to produce an output array of the same dimension and shape as the input arrays, containing the resulting elements.

$\rho(\xi_1 \quad op \quad \xi_2) \equiv \rho(\xi_1)$ **Equation 3-17**

$\vec{i}\,\psi(\xi_1 \quad op \quad \xi_2) \quad \equiv \quad (\vec{i} \quad \psi \quad \xi_1) \quad op \quad (\vec{i} \quad \psi \quad \xi_2)$ **Equation 3-18**

Example using the above defined arrays:

$$\xi_1^2 + \xi_3^2 \equiv \begin{bmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{bmatrix}$$

**Scalar Extension:** $\sigma \quad op \quad \xi$ This kind of operation takes a scalar and an array as inputs, and applies an operation between the scalar and every element in the array, constructing an array of the same dimension and shape as the input array.

$\rho(\sigma \quad op \quad \xi) \equiv \rho(\xi)$ **Equation 3-19**

$\vec{i}\,\psi(\sigma \quad op \quad \xi) \quad \equiv \quad \sigma \quad op \quad (\vec{i} \quad \psi \quad \xi)$ **Equation 3-20**

Example using the above defined arrays:

$$2 + \xi_1^2 \equiv \begin{bmatrix} 3 & 4 \\ 5 & 6 \\ 7 & 8 \end{bmatrix}$$

## *3.2.5 The Axis Transposition and Partition Operations*

**Take**: $\vec{i} \uparrow \xi$ This structuring or restructuring operation partition an array $\xi$ by returning an array of shape $\vec{i}$, from the front end of the axes in the shape of $\xi$, if $\vec{i}$ is positive, or from the back-end otherwise.

For vectors:

$\rho(\sigma \uparrow \vec{v}) \equiv < abs \quad \sigma >$ **Equation 3-21**

$$(\sigma \uparrow \vec{v})[i] \equiv \begin{cases} \vec{v}[i] & if \quad 0 \leq \sigma < \tau\vec{v} \\ \vec{v}[(\tau\vec{v}) - \sigma + i] & if \quad -\tau\vec{v} \leq \sigma < 0 \end{cases}$$ **Equation 3-22**

For Array $\xi$ of shape $\vec{b}$ and dimension k, where j is the upper bound of the vector $\vec{i}$:

$$\rho(\vec{i} \uparrow \xi) \equiv \begin{cases} \vec{i} & if \quad j = k \\ < \vec{i}[0], \ldots \vec{i}[j-1], \vec{b}[j], \ldots \vec{b}[k] > & if \quad 0 \leq j < k \end{cases}$$ **Equation 3-23**

$$\vec{l}\,\psi(\vec{i} \uparrow \xi) \equiv \begin{cases} \vec{l}\,\psi\xi & if \quad j = k \\ < \vec{l}[0], \ldots \vec{l}[j-1], \vec{b}[j], \ldots \vec{b}[k] > \psi\xi & if \quad 0 \leq j < k \end{cases}$$ **Equation 3-24**

Examples using the arrays defined in section 3.2.1:

$\langle 2 \quad 2 \rangle \uparrow \xi_2^2 \equiv \begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix}$

$\langle 1 \quad 1 \quad 2 \rangle \uparrow \xi_1^3 \equiv \langle 1 \quad 2 \rangle$

$\langle 2 \quad 1 \quad 2 \rangle \uparrow \xi_1^3 \equiv \left[ \langle 1 \quad 2 \rangle \quad \langle 5 \quad 6 \rangle \right]$

**Drop**: $\vec{i} \downarrow \xi$ This function returns the remaining of the array in the take function after taking the index of it, i.e. it drops from the front end of the axes in the shape of $\xi$, if $\vec{i}$ is positive, or from the back-end otherwise.

For vectors:

$$\rho(\sigma \downarrow \vec{v}) \equiv <(\tau\vec{v}) \quad - \quad abs \quad \sigma > \qquad \textbf{Equation 3-25}$$

$$(\sigma \downarrow \vec{v})[i] \equiv \begin{cases} \vec{v}[\sigma + i] & if \quad 0 \leq \sigma < \tau\vec{v} \\ \vec{v}[i] & if \quad -\tau\vec{v} \leq \sigma < 0 \end{cases} \qquad \textbf{Equation 3-26}$$

For Array $\xi$ of shape $\vec{b}$ and dimension k, where j is the upper bound of the vector $\vec{i}$:

$$\rho(\vec{i} \downarrow \xi) \equiv \begin{cases} < b[0] - i[0], \ldots, b[j-1] - i[j-1] > & if \quad 0 \leq j < k \\ < b[0] - (abs \quad i[0]), \ldots, b[j-1] - (abs \quad i[j-1]) > & if \quad -k \leq j < 0 \end{cases}$$

**Equation 3-27**

$$\vec{l}\,\psi(\vec{i} \downarrow \xi) \equiv \begin{cases} \rho(\vec{i} \downarrow \xi) \uparrow \xi & if \quad 0 \leq j < k \\ < -(b[0] - (abs \quad i[0])), \ldots, -(b[j-1] - (abs \quad i[j-1])) > \uparrow \xi & if \quad -k \leq j < 0 \end{cases}$$

**Equation 3-28**

Examples using the arrays defined in section 3.2.1:

$\langle 2 \quad 2 \rangle \downarrow \xi_2^2 \equiv \langle 11 \quad 12 \rangle$

$\langle 1 \quad 1 \quad 1 \rangle \downarrow \xi_1^3 \equiv \langle 8 \quad 12 \rangle$

$\langle 2 \quad 1 \rangle \downarrow \xi_1^3 \equiv \langle 11 \quad 12 \rangle$

**Reverse**: $d\phi\xi$ this operation reverses the elements of array $\xi$ on the required dimension d, preserving symmetry.

For vectors:

$\rho\phi\vec{v} \equiv \rho\vec{v}$ **Equation 3-29**

$(\phi\vec{v})[i] \equiv \vec{v}[(\tau\vec{v}) - (i+1)]$ **Equation 3-30**

For arrays:

$\rho\phi\xi \equiv \rho\xi$ **Equation 3-31**

$< \vec{i} > \psi(\phi\xi) \equiv < (\rho\xi)[d] - (i+1) > \psi\xi$ **Equation 3-32**

Examples using the arrays defined in section 3.2.1:

$$\phi\xi_2^2 \equiv \begin{bmatrix} 9 & 10 & 11 & 12 \\ 5 & 6 & 7 & 8 \\ 1 & 2 & 3 & 4 \end{bmatrix}, \text{ on dimension 1,}$$

$$\phi\xi_2^2 \equiv \begin{bmatrix} 4 & 3 & 2 & 1 \\ 8 & 7 & 6 & 5 \\ 12 & 11 & 10 & 9 \end{bmatrix}, \text{ on dimension 2.}$$

**Rotate**: $\vec{\sigma}\theta\xi$ this operation rotates the array $\xi$ with dimension j, on every axis k, $\vec{\sigma}[k]$ positions towards increasing indices if $\vec{\sigma}[k]$ is positive, and towards decreasing indices otherwise.

For vectors:

$\rho(\sigma\theta\vec{v}) \equiv \rho(\vec{v})$ **Equation 3-33**

$\sigma\theta\vec{v}[i] \equiv \vec{v}[(i+\sigma) \mod \tau\vec{v}]$ **Equation 3-34**

For arrays:

$\rho(\vec{\sigma}\theta\xi) \equiv \rho(\xi)$ **Equation 3-35**

$< \vec{i} > \psi(\vec{\sigma}\theta\xi) \equiv <[(\vec{i}[o]+\vec{\sigma}[o]) \mod (\rho\xi)[0]],\ldots[(\vec{i}[j-1]+\vec{\sigma}[j-1]) \mod (\rho\xi)[j-1]] > \psi\xi$

**Equation 3-36**

Examples using the arrays defined in section 3.2.1:

$$\langle 2 \quad 2 \rangle \theta \xi_4^2 \equiv \begin{bmatrix} 11 & 12 & 9 & 10 \\ 15 & 16 & 13 & 14 \\ 3 & 4 & 1 & 2 \\ 7 & 8 & 5 & 6 \end{bmatrix},$$

$$\langle 0 \quad 2 \rangle \theta \xi_4^2 \equiv \begin{bmatrix} 3 & 4 & 1 & 2 \\ 7 & 8 & 5 & 6 \\ 11 & 12 & 9 & 10 \\ 15 & 16 & 13 & 14 \end{bmatrix},$$

$$\langle 2 \quad 0 \rangle \theta \xi_4^2 \equiv \begin{bmatrix} 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix},$$

$$\langle 3 \quad 1 \rangle \theta \xi_4^2 \equiv \begin{bmatrix} 14 & 15 & 16 & 13 \\ 2 & 3 & 4 & 1 \\ 6 & 7 & 8 & 5 \\ 10 & 11 & 12 & 9 \end{bmatrix}$$

So, rotate achieves the result of a multidimensional circular shift, where the parameter index, decides the amount of shifting required on each axis.

**Transpose**: $\vec{v}\Phi\xi$ This operation applies a multi-dimensional transposition on the input array, defined by a permutation vector of length equals to the array's dimensionality. In a 2-dimensional array, a transposition of permutation vector = <2, 1> converts rows to columns. For arrays of dimensionality more than 2, a permutation vector decides the way the array is transposed, so it must be always of length equal to the arrays dimensionality, and its elements are the order of switching the multi-dimension index of the input array. A reverse order permutation order, always converts rows to columns.

This operation does not apply on Vectors. It applies only on Arrays with j dimensions greater than or equal 2:

$\rho(\vec{v}\Phi\xi) \equiv \rho(\xi)$ **Equation 3-37**

$< \vec{i} > \psi(\vec{v}\Phi\xi) \equiv < i[(\rho\xi)[v[0]]], ..., i[(\rho\xi)[v[j-1]]] > \psi\xi$ **Equation 3-38**

Examples using the arrays defined in section 3.2.1:

$$\langle 2 \quad 1 \rangle \Phi \xi_4^2 \equiv \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix},$$

$$\langle 3 \quad 2 \quad 1 \rangle \Phi \xi_1^3 \equiv \begin{bmatrix} \begin{bmatrix} 1 & 5 & 9 \\ 3 & 7 & 11 \end{bmatrix} & \begin{bmatrix} 2 & 6 & 10 \\ 4 & 8 & 12 \end{bmatrix} \end{bmatrix},$$

**Grade Up**: *gu* is a function that sorts the elements in an array in ascending order

Examples using the arrays defined in section 3.2.1:

$$gu\xi_5^2 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 \\ 21 & 22 & 23 & 24 & 25 \end{bmatrix}$$

**Grade Down**: *gd* is a function that sorts the elements in an array in descending order

Examples using the arrays defined in section 3.2.1:

$$gd\xi_5^2 = \begin{bmatrix} 25 & 24 & 23 & 22 & 21 \\ 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 & 1 \end{bmatrix}$$

**Slice**: $a \subset \xi \ b$ This function slices the multidimensional array $\xi$ into slices of arrays of the dimensionality a, and returns the specific slice b.

For vectors:

$$a \subset \vec{v} \ b \equiv a \ \rho \ \vec{v} \ [b] \qquad \textbf{Equation 3-39}$$

For arrays:

$$a \subset \xi \ \vec{b} \equiv \ \vec{b} \ \psi \ (\vec{a} \ \rho \ \xi \ ) \qquad \textbf{Equation 3-40}$$

Examples using the arrays defined in section 3.2.1:

$$0 \subset \xi_1^3 \ \langle 0 \quad 0 \quad 0 \rangle \equiv \langle 1 \quad 5 \quad 9 \rangle$$

$$1 \subset \xi_1^3 \ \langle 1 \quad 1 \quad 1 \rangle \equiv \langle 6 \quad 8 \rangle$$

$$2 \subset \xi_1^3 \ \langle 2 \quad 0 \quad 0 \rangle \equiv \langle 9 \quad 10 \rangle$$

### *3.2.6 The Reduction and Scan Operations*

**Reduction**: (*$_{op}$ red*) It is like the scalar extension operations discussed above, but requires no scalar input and the operation is applied on the components of the input array cumulatively, reducing its dimension by 1.

For vectors:

$\delta \quad {}_{op}red \quad \vec{v} \equiv 0$ ➔ a scalar, an array of dimension zero. **Equation 3-41**

$\rho({}_{op}red \quad \vec{v}) = <1>$, a scalar has a shape of 1. **Equation 3-42**

${}_{op}red \quad \vec{v} \equiv op_{c=0}^{c=\tau\vec{v}-1}\vec{v}[c]$ **Equation 3-43**

generating one element (a scalar) of the result of applying the operation on the elements of the input vector.

For arrays:

$\delta \quad {}_{op}red \quad \xi \equiv \delta\xi - 1$, the dimensionality, **Equation 3-44**

$\rho({}_{op}red \quad \xi) = <b[1],...,b[j-1]>$ **Equation 3-45**

skipping the first dimension b[0] that will be reduced.

$\vec{i}\ \psi({}_{op}red \quad \xi) \equiv op_{c=0}^{c=b[0]} <c, i[1],...,i[j-1]> \psi\xi$ **Equation 3-46**

Examples using the arrays defined in section 3.2.1:

$${}_{+}red \quad \vec{v}_1 \quad \equiv \quad 15$$

$${}_{+}red \quad \xi_2^2 \quad \equiv \quad \langle 15 \quad 18 \quad 21 \quad 24 \rangle$$

**Scan:** ${}_{op}scan\ \ \xi$ retains all the partial results obtained in a reduction operation on an array on the specified dimension.

For vectors:

$\rho({}_{op}scan\ \ \vec{v})\ \equiv\ \rho\vec{v}$ **Equation 3-47**

$({}_{op}scan\ \vec{v})[i]\ \equiv\ {}_{op}red\ ((1+i)\ \uparrow\ \vec{v})\qquad \forall\ 0 \le i < \tau\vec{v}$ **Equation 3-48**

For arrays:

$\rho({}_{op}scan\ \ \xi)\ \equiv\ \rho\xi$ **Equation 3-49**

$\vec{i}\,\psi({}_{op}scan\ \xi)\ \equiv\ {}_{op}red\ (<1+i[0],1+i[1],\ldots,1+i[\tau\rho\xi-1]>\ \uparrow\ \xi)\ \forall\ <0,\ldots,0> \le \vec{i} < \rho\xi$

**Equation 3-50**

Examples using the arrays defined in section 3.2.1:

$${}_{+}scan\quad \vec{v}_1\ \equiv\ \langle 1\quad 3\quad 6\quad 10\quad 15\ \rangle$$

$${}_{+}scan\quad \xi_2^2\ \equiv\ \begin{bmatrix} 1 & 2 & 3 & 4 \\ 6 & 8 & 10 & 12 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$, on the first dimension

$${}_{+}scan\quad \xi_2^2\ \equiv\ \begin{bmatrix} 1 & 2 & 3 & 4 \\ 6 & 8 & 10 & 12 \\ 15 & 18 & 21 & 24 \end{bmatrix}$$, on the second dimension

**Compress**: /• Compressing the array on the positions of the data elements set to 1 in the Boolean array specified. The original equation in the MOA notation:

For vectors:

$$\rho(\vec{v}_1 \quad /\bullet \quad \vec{v}_2) \equiv \langle {}_{+}red \quad \vec{v}_1 \rangle$$ **Equation 3-51**

$$(\vec{v}_1 \quad /\bullet \quad \vec{v}_2)[i] \equiv \vec{v}_2[\langle gd \quad \vec{v}_1 \rangle[i]]$$ **Equation 3-52**

For arrays:

$$\rho(\vec{v} \quad /\bullet \quad \xi) \equiv \langle {}_{+}red \quad \vec{v} \rangle \oplus (1 \quad \downarrow \quad \rho\xi)$$ **Equation 3-53**

$$\langle \vec{i} \rangle \psi(\vec{v} \quad /\bullet \quad \xi) \equiv < \langle gd \quad \vec{v} \rangle [i] > \psi\xi$$ **Equation 3-54**

Examples using the arrays defined in section 3.2.1:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \quad /\bullet \quad \xi_1^2 \equiv \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

**Expand**: $\bullet\backslash$ Expands the array on the positions of the data elements set to 1 in the Boolean array specified. Using the same Boolean array argument, this function returns the same original array (the dimension, the shape, and the values of the non-skipped elements), but with the data elements that has been skipped in the compress, will be equal to zero. The first argument is the Boolean array.

For vectors:

$$\rho(\vec{v}_1 \quad \bullet\backslash \quad \vec{v}_2) \equiv \rho\vec{v}_1$$ **Equation 3-55**

$$(\vec{v}_1 \quad \bullet\backslash \quad \vec{v}_2)[i] \equiv \begin{cases} 0 & if \quad \vec{v}_1[i] \equiv 0 \\ \vec{v}_2[({}_{+}scan\,\vec{v}_1) \quad [i]-1] & Otherwise \end{cases}$$ **Equation 3-56**

For arrays, over the zero's dimension:

$\rho(\vec{v} \quad \bullet\backslash \quad \xi) \equiv (\rho\vec{v}) \quad \oplus \quad (1 \quad \downarrow \quad \rho\xi)$ **Equation 3-57**

$\langle i\rangle\psi(\vec{v} \quad \bullet\backslash \quad \xi) \equiv \begin{cases} (1 \downarrow \rho\xi) \ \rho 0 & if \ \vec{v}[i] \equiv 0 \\ ((_{+}scan\ \vec{v}) \ [i]-1) \ \psi \ \xi & if \ \vec{v}[i] \equiv 1 \end{cases}$ **Equation 3-58**

Examples using the arrays defined in section 3.2.1:

$$\begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \quad \bullet\backslash \quad ( \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \quad /\bullet \quad \xi_1^2) \equiv \begin{bmatrix} 1 & 0 \\ 3 & 0 \\ 5 & 0 \end{bmatrix}$$

## *3.2.7 The Higher Order Operations*

**Omega**: $_{op}\Omega a \quad \xi_1 \quad b \quad \xi_2$ This is a higher order operation, that applies operation op on pairs of identically indexed sub arrays from the slice of dimension a of array $\xi_1$ and the slice of dimension b of array $\xi_2$.

$_{op}\Omega \quad a \quad \xi_1 \quad b \quad \xi_2 \equiv (a \quad \subset \quad \xi_1) \quad op \quad (b \quad \subset \quad \xi_2)$, for all slices from both arrays **Equation 3-59**

Examples using the arrays defined in section 3.2.1:

$$\xi_o \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$_{+}\Omega \quad 0 \quad \xi_O \quad 1 \quad \xi_2^3 \quad \equiv \quad \begin{bmatrix} (1 \ + \ \langle 1 \ \ 2\rangle) & (2 \ + \ \langle 3 \ \ 4\rangle) & (3 \ + \ \langle 5 \ \ 6\rangle) \\ (4 \ + \ \langle 1 \ \ 2\rangle) & (5 \ + \ \langle 3 \ \ 4\rangle) & (6 \ + \ \langle 5 \ \ 6\rangle) \end{bmatrix}$$

**Dot:** $\bullet$ This function performs the outer and inner products of arrays. It has two variations, a unary Dot operation that performs the outer product, and a binary Dot operation that performs the inner product. The outer product unary dot operation applies the unary operator on each element of the first array with each element in the second array, ending up with an expanded result of shape formed by the concatenation of the shapes of the input arrays. The inner product binary operation applies an operator on every pair of equally indexed elements from the two arrays, then reduces the resulting array by one dimension, ending up with a collapsed array than the input arrays. The functionality of the Dot operation can be decided as the problem domain requires, and different variations can be introduced. The two operations proposed in this thesis are formalized as follows:

Outer Product Unary Dot:

$$\rho\left(\xi_1 \quad \bullet_{op} \quad \xi_2\right) \equiv \rho\xi_1 \quad \oplus \quad \rho\xi_2$$ **Equation 3-60**

$$\left(\vec{i}, \quad \vec{j}\right)\psi\left(\xi_1 \quad \bullet_{op} \quad \xi_2\right) \equiv \left(\vec{i} \quad \psi \quad \xi_1\right) \quad op \quad \left(\vec{j} \quad \psi \quad \xi_2\right)$$ **Equation 3-61**

Inner Product Binary Dot:

$$\rho\left(\xi_1 \quad {}_{op_0}\bullet_{op_1} \quad \xi_2\right) \equiv \left(\overline{1} \quad \downarrow \quad \rho\xi_1\right) \quad \oplus \quad \left(1 \quad \downarrow \quad \rho\xi_2\right)$$ **Equation 3-62**

$$(\xi_1 \quad {}_{op_0}\bullet_{op_1} \quad \xi_2) \equiv {}_{op_0}red \quad (\xi_1 \quad op_1 \quad (\vec{v}\Phi\xi_2))$$ **Equation 3-63**

where $\vec{v} \equiv 1 \quad \varphi \quad (\iota(\delta\xi_2))$, which is a vector of a reverse order permutation of size equal the dimensionality of the second array input.

Examples:

$\xi_d^1 \equiv \langle 1 \quad 2 \quad 3 \quad 4 \rangle$

$$(\xi_d^1 \quad \bullet_* \quad \xi_d^1) \equiv \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

$$(\xi_d^1 \quad +\bullet_* \quad \xi_d^1) \equiv 30$$

***MOA expression Example:***

Given a 4 dimensional array as follows:

$$\xi^4 \equiv \begin{bmatrix} \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} & \begin{bmatrix} 13 & 14 \\ 15 & 16 \\ 17 & 18 \end{bmatrix} \end{bmatrix} \\ \begin{bmatrix} \begin{bmatrix} 19 & 20 \\ 21 & 22 \\ 23 & 24 \end{bmatrix} & \begin{bmatrix} 25 & 26 \\ 27 & 28 \\ 29 & 30 \end{bmatrix} & \begin{bmatrix} 31 & 32 \\ 33 & 34 \\ 35 & 36 \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

With shape vector:

$$\rho\xi^4 \equiv \langle 2 \quad 3 \quad 3 \quad 2 \rangle$$

To choose the sub-array that corresponds to the first hyper-plane in the axis 0, first two planes in axis 1, last two rows in axis 2, and all the columns in axis 3, results from the following nested MOA expression:

$$1 \downarrow (\langle 1 \quad 2 \rangle \uparrow \xi^4) \equiv \begin{bmatrix} \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix} & \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \end{bmatrix}$$

## 3.3 Psi Reduction Theorem

Reduction in literature can be defined to be shape reduction, or data reduction. Shape reduction means the definition of the array shapes statically, simplifying the shape expressions, and evaluating polymorphic constants. Data reduction is expressed as the process of replacing the manipulation of a whole array with the manipulation of a part, which increases efficiency.

The Psi reduction theorem is the process of simplifying the expression for the item in terms of its Cartesian coordinates. The MOA operations defined above are all designed so that expressions can be reduced to the minimal normal form. Consider the following example:

Given the following three-dimensional array:

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

with shape vector:

$$\rho\xi^3 \equiv \langle 3 \quad 2 \quad 2 \rangle$$

The following MOA expression drops 1 from the array, then takes 1, results in:

$$1 \uparrow (1 \downarrow \xi^3) \equiv \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

This is equivalent to the following reduced expression:

$$1\psi\xi^3 \equiv \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

As shown above, MOA array expressions can be reduced to their simplest form, which does the minimal work, to an expression by only involving the $\psi$ - Psi operation. The Psi-reductions are deterministic and mechanical, and could be used in conjunction with existing parallel compiler technology.

Considering the previous definition of reduction in MOA, we can see that reduction of any MOA array over a dimension interacting with its shape vector, is a reduction over shape and data. The reduce operation discussed earlier, is a general form of reducing the size of the data in the array by applying an operation between pairs of items in the array along a specified axis reducing its dimensions by one. Another form of reduction is achieved by the compress operation, which reduces the size, by skipping some of the elements in the array. However, the expand operation does not restore the data reduced in the compress function, as shown in the examples above. Another variation for the compress function could be implemented that averages the values of the non-skipped elements by the neighboring skipped elements (using the connectivity operation), so that the compressed array can be restored later by expansion, using an inverse operation. This is illustrated in the next chapter in the image compression section 5.2.7.

The reduction entails interaction between the data and the shape. However this interaction is designed to be invariant of the values and the size of the shape vector, so that we can maintain dimension and shape invariance in our programming paradigm in general. This means that data reduction results in shape reduction also, which can affect also the dimension of the array. So, the resulting array after a reduction operation can be a different shape and dimension, than the input array.

## 3.4 Psi Correspondence Theorem (PCT)

The Psi Correspondence Theorem (PCT) is used to express the selection in terms of starts, strides and lengths. It is used to access memory efficiently. Since, all MOA operations are expressed in terms of the $\psi$ - Psi operation, the PCT can translate their definitions from one involving Cartesian coordinates into one involving selection from a list of items stored in memory, or through abstract data or processor restructurings. The PCT can contribute to reliable methods for partitioning, and mapping to multiple processors. It is used to collapse multiple loops based on three notations:

1. ***The (expr)*** $\Gamma$ ***Notation***: this notation is based on extracting slices from an array, according to their Psi index position, by defining the index of the slice required in every dimension in the array.

For example, given the same three-dimensional array defined in the previous section, the following expression results in:

$$\left\langle \vec{i} \quad \vec{j} \quad \vec{k} \right\rangle \Gamma \xi^3 \mid [\vec{i} = \langle 0 \quad 2 \rangle,\ \ \vec{j} = \langle 1 \rangle,\ \ \vec{k} = \langle 0 \rangle] \equiv \begin{bmatrix} 1 & 9 \\ 3 & 11 \end{bmatrix}$$

As shown, this operation works by defining the slices in each axis from which to extract the elements, the first index vector $\vec{i}$ defined the first axis and the last axis, which is the first and the last brackets in the above array, then the second index vector $\vec{j}$ defined the second (the last) row, and finally the last index vector $\vec{k}$ defined the first column to extract from. These slices resulted in the above array. As you see, the shape of the resulting vector is dependant on the values of the indices provided. A rule for the resulting shape can be induced to be:

$$\rho\left(\left\langle \vec{i} \quad \vec{j} \quad \vec{k} \right\rangle \Gamma \xi^3\right) \equiv \left\langle \tau\vec{i} \quad \tau\vec{J} \quad \tau\vec{K} \right\rangle$$

2. ***The (mix)*** $\Xi$ ***Notation***: this notation is based on gluing the arrays formed using $\psi$ on partial indices.

For example using the same three-dimensional array defined in the previous section, the following expression results in:

$$\vec{i}\,\Xi\,\xi^3 \mid [\vec{i} =< 0 \quad 2 >] \equiv \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \end{bmatrix}$$

which is equivalent to the following MOA expression using the $\psi$ - Psi operation and the $\oplus$ - catenate operation defined earlier in this chapter:

$$(0\psi\xi^3) \oplus (2\psi\xi^3) \equiv \begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} & \begin{bmatrix} 9 & 10 \\ 11 & 12 \end{bmatrix} \end{bmatrix},$$ which is a catenation on the second dimension.

3. ***The (link)*** $\Lambda$ ***Notation***: this is the vector formed by concatenating the items collected from the array by the previous expr notation.

The three notations are used to describe the start, stride and length of selection in an MOA array, which are used in the loop to traverse a specified selection from an MOA array, which leads to collapsing the number of loops required to traverse for the selection. PCT is not implemented in this thesis; for further details on the computation of the start, stride and length, refer to [6]. However, there is a discussion in chapter five, relating the video stream with the audio stream in an AVI file, and another discussion in chapter six, on the proposed parallel architecture-mapping scheme.

## 3.5 Summary of MOA Operations

Table 3-2 summarizes the entire MOA notation as explained in the previous sections.

Table 3-2: Summary of MOA Operations

| **Symbol** | **Name** | **Description** |
|---|---|---|
| $\delta$ | Dimensionality | Returns the number of dimensions of the array. |
| $\rho$ | Shape | Returns a vector of the upper bounds or sizes of each dimension in the array. |
| $\psi$ | Psi | The main indexing function of the MOA notation, returns either scalars of the elements in the array, if provided with a full index, or partitions of the array otherwise. |
| *rav* | Ravel | Flattens a multi-dimensional array, retrieving all its elements as one-dimensional array. |
| $\pi$ | Pi | Returns the product of the elements in a multidimensional array. |
| $\tau$ | Tau | Returns the number of elements in a multidimensional array (the Pi result of the shape vector of the array). |
| $\lambda$ | Gamma | Indexing function, converts a multidimensional index into its scalar equivalent in the flat (raveled) array. |

Table 3-2: Summary of MOA Operations - Continued

| | | |
|---|---|---|
| $\lambda'$ | Gamma Inverse | Inverse of the previous function; returns the multidimensional index from a scalar index, given the shape of the array. |
| $\vec{s}\rho\xi$ | Reshape | Changes the shape vector of the array, affecting its dimensionality, and rearranging its elements to fit symmetrically in the new shape. |
| $\xi_1 \oplus \xi_2$ | Catenate | Concatenates two arrays on the specified dimension, given that the rest of dimensions are of equal sizes. |
| $\xi_1 \; op \; \xi_2$ | Point-wise Extension | Applies an operator between every pair of equally indexed element in two arrays of the same dimensionality and shape. |
| $\sigma \; op \; \xi$ | Scalar Extension | Applies an operator between a scalar and every element in an array. |
| $\iota \quad n$ | Iota | Returns the vector containing elements from 0 to n-1. |
| $\uparrow$ | Take | Takes as much as specified in each element in the input vector from the corresponding dimension in the shape vector from the array, from the front if values are positive, and from the back, if values are negative. Returns the elements of the taken partition of the array. |
| $\downarrow$ | Drop | Drops as much as specified in each element in the input vector from the corresponding dimension in the shape vector from the array, from the front if values are positive, and from the back, if values are negative. Returns the remaining of the array. |
| $red$ | Reduce | Reduces the arrays dimension by one, by applying the specified operator on the elements of the arrays. |
| $gu$ | Grade – Up | Ascending sorting of the elements of the input array. |
| $gd$ | Grade – Down | Descending sorting of the elements of the input array. |
| $_{op}scan$ | Scan | Returns the partial results obtained from a reduction operation. |
| $\phi$ | Reverse | Symmetrically reverses the elements on the specified dimension. |

Table 3-2: Summary of MOA Operations - Continued

| | | |
|---|---|---|
| $\theta$ | Rotate | Rotates the elements of the arrays on the each dimension k, $\vec{\sigma}[k]$ positions towards increasing indices if positive, or decreasingly otherwise. |
| Φ | Transpose | Transposes the elements in the array by the permutation vector specified. |
| /• | Compress | Compressing the array on the dimensions set to 1 in the Boolean vector specified. |
| • \ | Expand | Expands the array on the dimensions set to 1 in the Boolean vector specified. |
| ⊂ | Slice | Slices the array to the specified dimensionality, returning the slice required. |
| Ω | Omega | Applies an operator over pairs of equally indexed slices of two MOA arrays. |
| • | Dot | Unary & Outer Dot operations applier the outer and inner product respectively. |

# *Chapter 4*

# MOA LIBRARY IMPLEMENTATION

## 4.1 Introduction

In this research work, MOA notation is implemented in a class named CMOA. Next chapter explains the implementation of CMOAImage, and CMOAVideo, which are based on the CMOA class for image and video processing respectively. The three classes are compiled in a Library to be included in any application. The main class CMOA is designed to operate all MOA notation described in the previous chapter, on an MOA_rec structure that contains the dimension n, shape (n tuple of sizes – extents or upper bounds - of the n dimensions), and the data elements of the MOA array, stored as a flat (raveled) array in principal row major order. The length of the flat array of the MOA array data elements is equal to the product of the bounds of its dimensions (the values of the shape vector elements). The shape and data of the MOA array are explicitly separated for the advantages of the shape theory as will be discussed in chapter six. The type of the Data is fixed in this implementation to be a DWORD. The choice of DWORD data type is influenced by the image and video applications implemented as described in chapter five. The image pixels, which are the array elements of the MOA arrays in both applications, require a minimum of one bit (black and white images) and a maximum of double word (32-bit pixel, describing the intensities of the blue, green, and red components of each pixels) space in memory to store a pixel value. However, data morphology (as explained in chapter six) can be implemented since C++ is a language that allows for overloading. Overloading can be employed to redefine the MOA functions to operate on different data types for array elements, based on different MOA structures denoting the different types. The CMOA class implementation will be explained in this chapter, while the next two classes will be explained in the next chapter. Appendix A contains the header declaration of the CMOA Class and shows the functions argument lists and return types.

### *4.1.1 MOA Data Structure*

**MOA_rec** is the main data structure used as argument and return type for most functions in the MOA library. It contains the three main attributes of the array description, which are the dimensionality, the shape, and the raveled (flattened) vector of its content data elements. It is declared as follows:

```
struct MOA_rec {
dimn;
shape;
elements;
}
```

**Restruct_rslt_rec**: this structure is used for the return types of the restructuring functions. It contains the new array and the set of indices it represent in the original array. It is declared as follows:

```
struct Restruct_rslt_rec {
MOA_rec;
orig_indices;
}
```

### *4.1.2 The CMOA Class*

The MOA library is designed (for the first time) by applying object oriented techniques, providing a class that contains the above MOA structure and the methods that perform the equations described in chapter three. The programmers can instantiate an object from the class and define its values and call the methods as the logic of the application in hand requires, or call APIs (higher-level constructs), that perform MOA notation functions on the values provided in the argument list. The Library consists of the following methods:

**The declaration and measurement functions:**

These functions return or set the private values of the MOA structure in the instantiated object of the MOA class.

- Class constructors: several constructors are defined; one takes one argument, which is a pointer to the MOA structure as defined above; another takes as argument list: the dimension, shape (which is an array of integer values of bound equals to dimension – 1, containing values defining the bound for every axis in the array), and the data elements in a flat one-dimensional array. Other constructors are defined to serve different purposes, like an empty constructor is defined to declare an object of the MOA class with no internal MOA array, which can be used to call the MOA operations for external MOA structures defined in the calling program.

- GetMOA: returns the values of the MOA structure defined above in the instance of the MOA class, by either returning a pointer to it, or a new instance equal to it.

- Dimn, Shape, Ravel are three functions, which return parts of the MOA structure values defined in the class, as the notation definition.

- SetShape, SetElements, SetMOA: change the values of the MOA structure in the MOA class, keeping a valid relationship between the dimension value, and the shape vector size on one hand, and the shape vector values and the number of components in the elements vector (flat array that contains the raveled vector of the multidimensional array) on the other.

- Pi: it returns the product of all the components of the array.

- Tau: it produces the product of the elements in the shape vector of an array, resulting in the number of elements in the array.

**The indexing functions:**

- PSI: this function returns either a scalar value (if the index argument is a full index – i.e. the number of elements in the index array is equal to the number of elements in the shape array). Or returns a portion of the axes of the multidimensional array not defined in the index vector, identified by the values of the defined axes in the index vector. The index vector is the set of values that in normal handling of arrays would have been used as subscripts for every dimension:

Example in traditional handling, a given multidimensional array defined as follows:

Integer M[4][5][6];

Used as follows:

```
value1 = M[2][4][3];
```

Will be represented by:

```
MOA_rec M;
M.dimn = 3;
M.shape[0] = 4;
M.shape[1] = 5;
M.shape[2] = 6;

Index[0] = 2;
Index[1] = 4;
Index[2] = 3;
```

When the above declarations are passed like this (notice we are passing a full index – number of elements in the index vector equals to the dimensionality of the array):

```
X = Psi (M, index)
```

a scalar value that can be assigned to the variable X is returned, defined in an MOA structure – i.e. X is defined to be an MOA structure, that holds a scalar in case a full index is passed to the Psi function, and a sub-array in case of a partial index.

It might seem that the number of statements used to declare an MOA array is much more than in traditional methods. However, the point is not in the declaration, it is in the further manipulation of the multidimensional array that gets more complicated as the dimension increases. Chapter seven has more on the Nested Loops reduction with MOA.

The Psi function returns another MOA structure as a result. If a full index is sent in the argument list, it returns a scalar (remember MOA represents scalars as arrays of zero dimension, shape of one element of value 1; and one element in the raveled array of value of the scalar). Otherwise it returns a multidimensional array as defined above.

- Gamma: as defined before, this function returns a scalar value of the index vector sent in the argument list that can be used as a subscript in the raveled vector of the elements. This function is useful in mapping the traditional handling of multidimensional arrays to the MOA notation, if direct access is required.

Example:

```
Value1 = M [2][3][4];
Is represented with MOA as:
Index[0] = 2;
Index[1] = 3;
Index[2] = 4;
Value1 = M[Gamma(M.shape, index)];
```

- Gamma_Inverse: as defined before, this function is the inverse of the previous function. It returns the multidimensional index of the scalar value argument that is an index in the raveled vector of elements in a multidimensional array.

**The Array Construction/Reconstruction Functions:**

- Iota: this function can be used to assign the elements of the array with values from 0 : n-1, where n is the number of elements decided by the Tau(shape), which is the default in the MOA testing tool provided.

- Catenate: this function as defined in the notation, generates a new MOA structure containing the result of the concatenation of two MOA Structures entered on the specified dimension.

- Reshape function transforms the data elements of the array, in addition to changing the dimensionality and the shape vector of the result.

- Pack: Adds a new dimension to the array of extent (shape) equal to two, where the two equally equivalent indexed elements of the two input arrays are packed as the new dimension. This function is defined in the Array Theory, not in the MOA notation.

**Value Transformation Operations:**

The implementation of the value transformation functions aims to:

- Determine minimal or maximal values along specific axes: min(d) and max(d).

- Apply an operator to all elements of an array (scalar) as scalar_op, or to pairs of elements from two arrays with identical shape (point-wise) as array_op.

**The Axis Transpositions and Partitioning Functions:**

Memory management enhancement is one of the basic advantages of designing the MOA notation, by reducing the temporary storage requirements, and reducing the amount of address generation of array references. A design decision must be taken in the return structure of the MOA partitioning functions. The first alternative is to return the indices of the elements in the required partition, so that the caller of the partitioning function can process the partition using the same memory reference allocated to the original array with an offset of the position of the element. The second alternative is to copy the partition as a new MOA structure with a new memory address, as a temporary variable that will be deleted after finishing the operation, or reused for other temporary calculations.

Sometimes, for some specific operations (for instance the transpose function), especially when processed sequentially, one can't change the original array in the same memory location, because as soon as we start changing the first few elements, the processing of later elements will require checking the original values of the previous elements. In parallel processing, this is not a problem as long as reading the original values happens at one clock cycle before starting to change it. In cases where parallel execution doesn't guarantee that reading will succeed to get the original values before the changes, for example when the number of processors is less than the problem size, there is no way out, except by allocating temporary memory space.

- Take: this function partitions the different axes of the shape input MOA array. For instance, it returns the first two of five in the first axis, and first 3 of six in the second axis, and so forth. Unlike the Psi function, a full valid index sent as argument to the Take function doesn't return a scalar. It returns as many slices of each dimension as defined in the position of that dimension position in the index vector. Positive values mean to slice from the front, and negative values mean to slice from the back. A partial index vector means including the full size of the missing axes.

- Drop: this function is implemented based on the take function. The input index is processed to define a new index to be passed to the take function to take the remaining part of the array after dropping the slices of the dimensions as defined in the input index vector.

- Slice: This function is implemented as explained in chapter 3. It is illustrated by figure 4-1. This figure shows how a 3-dimensional array can be sliced from either the front or the back on each dimension.

All the transformation functions are designed to return a new memory allocation for a new MOA structure, however as discussed previously, it can apply the changes in place in some cases only. The functions are: Rotate, Reverse, Grade_Up, Grade_Down, and Transpose functions transform the data elements of the input array preserving the same dimensionality and shape. The functionalities of these functions are implemented as described by the notation explained in chapter three.

- The Grade_Up and Grade_Down are implemented by traversing the elements of the input array (raveled array), then calls the min_on_dimn or max_on_dimn functions respectively (which is already one dimension, since the array is raveled), to specify the elements to be removed from the input array and placed in the current position in iteration on the output array.

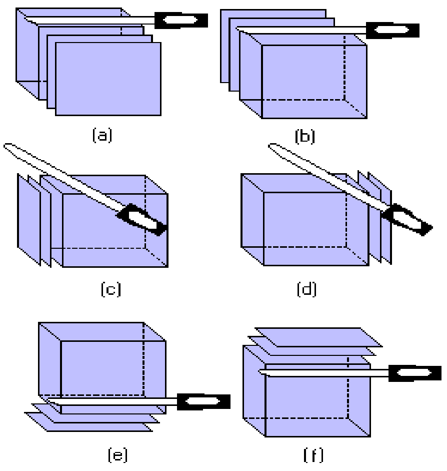


Figure 4-1: Slicing a 3-D array over: (a) the 0th dimension from the front end; (b) the 0th dimension from the back end; (c) the 1st dimension from the front end; (d) the 1st dimension from the back end; (e) the 2nd dimension from the front end; (f) the 2nd dimension from the back end

## The Reduction Operations:

- Reduction: reduces the dimensionality of an array by 1, by applying an operator along the reduced axis.

- Scan: scans the partial results of a reduction operation.

- Compress and Expand functions are implemented as required in the MOA notation. With Boolean MOA input entered by the user. The Boolean array must be of the same shape as the input array and contains ones in its space that correspond to a regular shape (i.e. constant number of ones in each axis, and this applies to all axes). Refer to chapter five, in the image compression section for better utilization of these two functions.

**Higher order primitives:**

- Omega: Value Transforming and restructuring at the same time. It works by applying an operator on pairs of elements identically indexed, of parameterized dimensionality from two arrays of different shapes, keeping the constraints discussed in chapter 3.

- Dot: computes Inner and Outer products of two MOA structures as explained in chapter 3.

**Other useful functions:**

- NextIndex: This function returns the next index vector of a given index $\vec{i}$, within a specified shape $\vec{s}$. It is used to traverse an MOA array. The programmer starts calling this function with all elements in the starting index equal to zero. It increments the argument index by one position in the bounds specified in the shape vector argument.

- Connectivity operation returns the set of indices of the connected elements (number of steps away as specified in input) in the array to the specified element, on the specified dimension. The number of the returned set of connected elements is bound according to the dimension of the array, and the dimension of the connectivity. For example a two dimension array can return 8-connected elements if the dimension required for the connectivity is the second (including connections on the diagonal), and the number of steps away from the required point is one; and it returns only 2-connected elements on dimension one and also one step away from the specified point. If it is a vector, the connected elements can't be more than 2 in case of one step away from the specified point, i.e. the element before and after the specified element in the flat array. From this we can note that the number of elements retrieved in the Connected set, is dependent on the location of the specified point. If it is at the borders of the array, it returns less neighbors than the other elements in the middle, in case of one step away, where d is the specified dimension, which must be less than the dimensionality of the input array. This function can be designed in several ways. It can be a simple partitioning using the take and the drop functions, deciding the starting index and the length indices from the index of the element required and the number of steps for neighboring as required (for direct connectivity, number of steps are 1). Otherwise, it is implemented as a transformation on the index of the element required, traversing all its neighbors only.

- GetOrientation: This function is used to suggest an orientation for the border data elements that when used to return its neighboring elements, will find empty positions. This suggests according to the position of the element in the array, the orientation to align the returned neighbors MOA structure, in the expected larger neighboring structure, which is equivalent to either the mask used (as in the convolution operation described in chapter five), or the number of steps away at which the neighbors are included.

Figure 4-2 illustrates how the neighboring and the GetOrientation function work. The input array is a 2-dimensional array, with shape vector = <3 3>. The element enclosed in the small square is the element whose neighbors are to be returned. This figure illustrates the neighbors one step away from the required element's position. So, this requires the output to be always a <3 3> shaped two-dimensional array. The GetOrientation function, decides for the border elements like in a, b, c, d, f, g, h, and i, the orientation to align the returned neighbors in an MOA structure. Figure (e) is a middle element, not a border element. The Orientations are defined in the MOA class in an enumeration as shown in Appendix A.

The summary table 4-1 in section 4.5 may be reviewed for other implemented functions. The implementation of the above functions is divided into three stages, which are repeated in almost all functions. First define the resulting dimension. Second, compute the resulting shape, which either depends on the inputs' data or inputs' shapes according to the following:

- Most cases it is defined based on the inputs' shapes.
- Reshape, the same data elements of the input array is distributed over a new shape.
- The catenate function, the resulting shape depends on the inputs' shapes and the parameterized dimension of the catenation.
- The take function, makes the resulting shape equal to the index vector if it is positive, or a variation of it, if it is negative.
- The drop, the resulting shape is a subtraction between the input's shape and the index vector.
- The compress and expand functions, analyzing the MOA Boolean input decides the resulting shape.

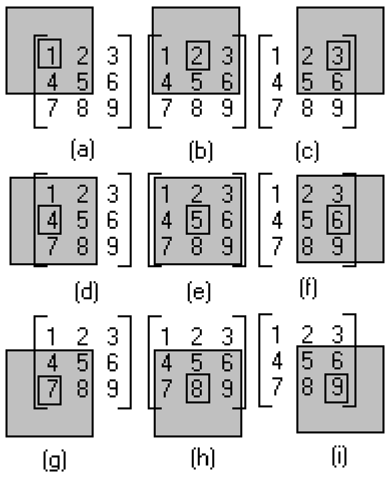


Figure 4-2: Orientation of the neighboring function related to elements' position; (a) lower right; (b) lower middle; (c) lower left; (d) middle right; (e) center; (f) middle left; (g) upper right; (h) upper middle; (i) upper left.

The third stage is the computation of the resulting element values as specified by the function definition. A general check is always performed in all functions to take care of scalar MOA structures, where the dimension is equal to zero and the shape vector of length one that is equal to value one, and hence data vector of length one that is equal to the scalar's value. Input validation (constraints) based on the functions' definition is always performed before starting to compute the function.

## 4.2 The MOA Library C++ Design Decisions

Functional languages in general, view memory locations as "box-variables", associating four basic operations to them: allocate, deallocate, fill content of box, and read content. In contrast, conventional languages view a memory location as a "place-holder-variable", which is either attached to an expression by functional application, or an occurrence of the of the place holder variable is replaced by the attached expression. This replacement makes neither the place nor the expression accessible through the variable name after that. [11] This concept makes the implementation of the MOA on functional languages more appropriate to satisfy one of the main advantages of applying it, which is avoiding the creation of temporaries, and calculation in place. However, sometimes the processing logic, given some constraints as discussed previously, requires creating temporaries during calculation to produce correct results.

The implementation of the library in Visual C++, as an imperative language in the sense that it consists of a sequence of commands, which are executed strictly one after the other, depended entirely on memory allocation at run time, which required care in the bounds of the run-time defined shapes and arrays, and efficiently implemented equations that preserve the boundary constraints. It is compiled as a Dynamic Link Library (DLL), so that applications can include it and use its notation. The compilation of the MOA library to a DLL, helped to compile the library once, and to use it by including it in other applications programmed with any language as APIs. Dynamic array allocations can be done with two different methods, either to use the MFC class CArray which allows for expanding or shrinking the array size as an array of a changing size may require, while preserving the values of the common elements between the original and new size. Another method is the simple memory allocation of the required size, and is subject to be reallocated at any point during the execution of the program. The second method proved to be more appropriate, and allows for better control over the memory allocations.

Since the design of the MOA indexing functions is based on an index component (a vector of indices), not simple scalars like in traditional array operations. This imposed a difficulty in using the indexing functions' results in conjunction with second order operations. So, the subscript selection is filled in a vector of indices to be passed to the operations, even for scalars, vectors, and less dimensional arrays. It could be investigated to provide indexing operators equivalent to the traditional bracket subscripting provided for vectors at least in the same manner it is provided for the CString MFC class. The MOA structure stores regular (rectangular shapes) that can be viewed to be indexed as follows:

given $\rho\xi^3 = \langle 3 \quad 2 \quad 2 \rangle$

then: $\tau\xi = 12$

Figure 4-3 shows how the Psi-indexing function operates. The content of every box represents the index argument to the Psi. In the figure a three dimensional array is represented, a vector index of one element returns the subtree under the first dimension below the index. The same applies to the second dimension, until the leaves of the tree contains the flat array data elements, with the same multidimensional index as specified in the diagram, applying the row-major order. Figure 4-3 also illustrates how the rectangular homogenous arrays are represented in the CMOA implementation as a balanced tree.

The MOA library implemented was referenced in another two classes that apply the MOA notation for the two examples of image and video as a 2-Dimnesion MOA, and a 3-Dimension MOA respectively. The other two classes are explained in more details in the next chapter.

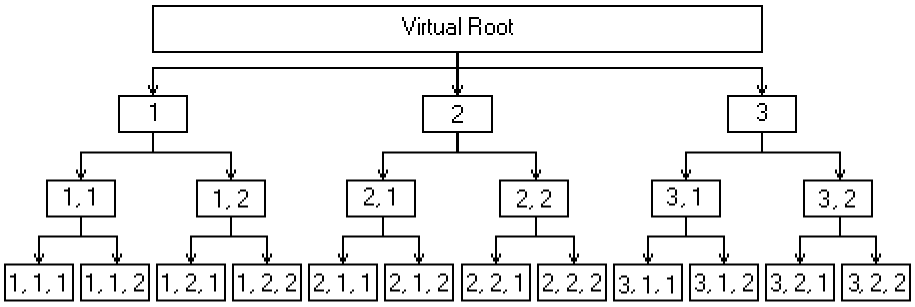


Figure 4-3: Regular Rectangular Array Indexing (Psi-operation)

## 4.3 The MOA Testing Tool

A user interface is provided for the MOA library discussed above, which allows the user to access all its operations. This MOA testing tool is a multiple document interface application, where the user can define several MOA arrays, all displayed in separate work documents. It is used to verify the computation in MOA, returning the result of operations one by one in a new document containing the resulting MOA array, and subject for further operations.

The Tool starts by the definition of a MOA array with the dialog interface shown in figure 4-4, where the dimension of the array is defined, and this automatically decides the number of elements expected in the shape vector defined in the second list box. The values in the shape list box, automatically decides the total number of data elements to be expected in the second list box. The user can check the Iota Elements check box to hide the elements list box, and assign the elements the result of calling the itoa function with argument equals to the Tau of the shape vector defined. A definition like that shown in figure 4-4 (dimensionality = 2; shape = <3 3>; elements = iota(9)), defines the MOA array displayed in figure 4-5.

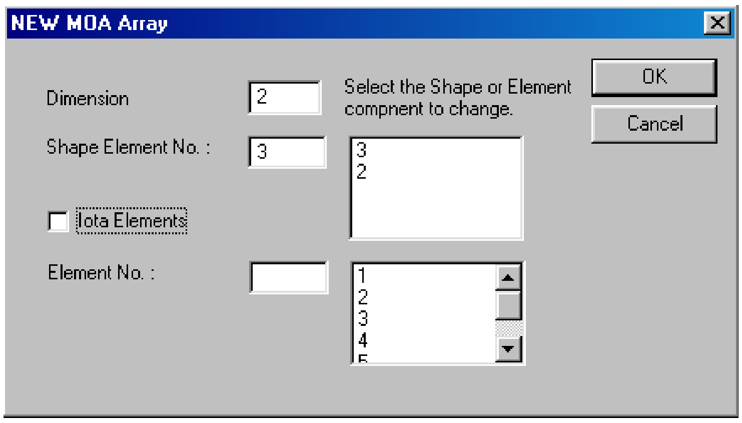
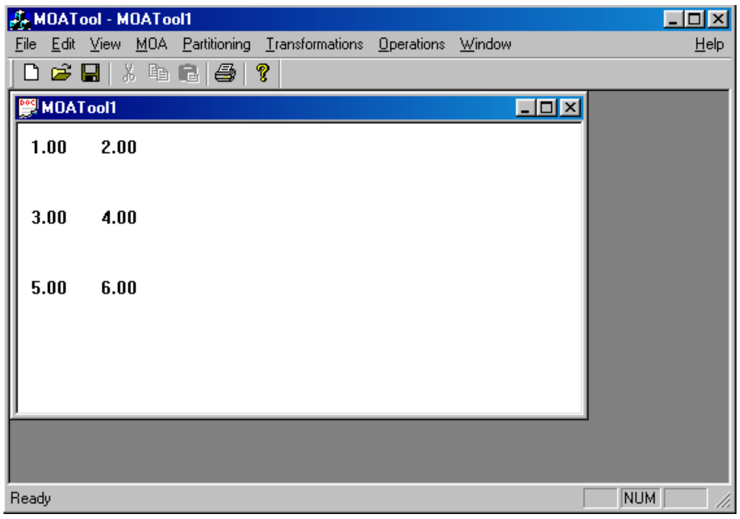


Figure 4-4: New MOA dialog interface in the MOA Testing Tool


MOATool - MOATool1
File Edit View MOA Partitioning Transformations Operations Window Help
MOATool1
Ready
NUM

```
1.00    2.00

3.00    4.00

5.00    6.00
```

Figure 4-5: MOA Testing Tool

Figure 4-6 to figure 4-9 display the Menus contained in the MOA tool, covering all the MOA notation.

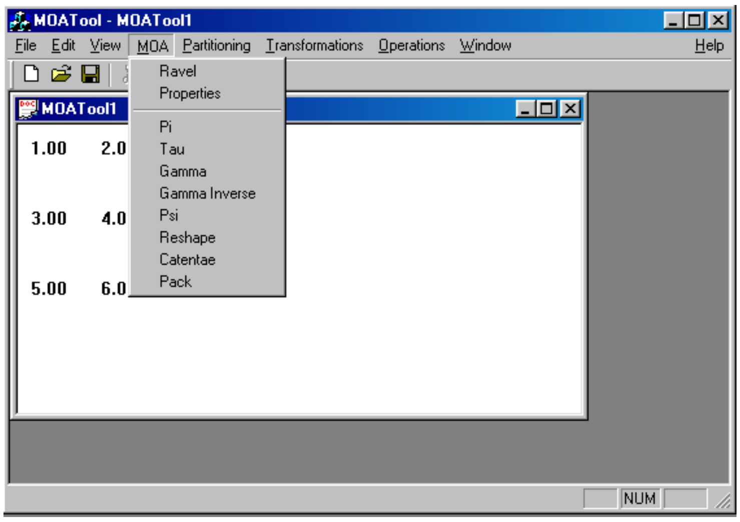


Figure 4-6: MOA Menu in the MOA Testing Tool

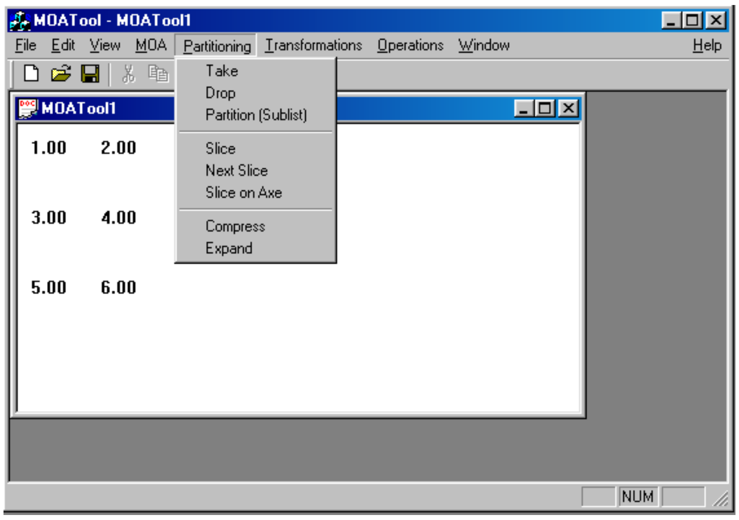
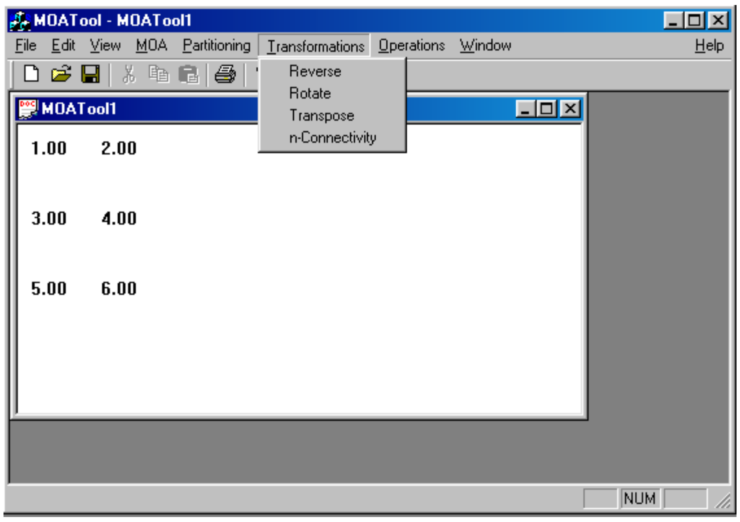
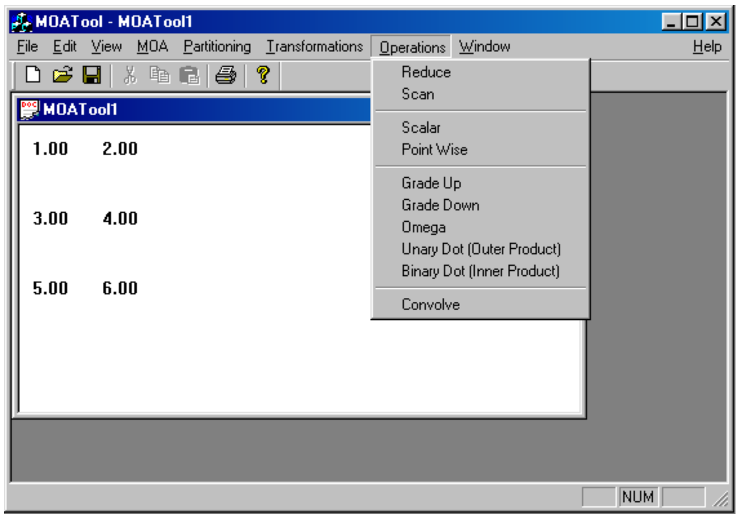


Figure 4-7: Partitioning Menu in the MOA Testing Tool


MOATool - MOATool1
File Edit View MOA Partitioning Transformations Operations Window Help
Reverse
Rotate
Transpose
n-Connectivity
MOATool1
NUM

```
1.00    2.00

3.00    4.00

5.00    6.00
```

Figure 4-8: Transformations Menu in the MOA Testing Tool


MOATool - MOATool1
File Edit View MOA Partitioning Transformations Operations Window Help
Reduce
Scan
Scalar
Point Wise
Grade Up
Grade Down
Omega
Unary Dot (Outer Product)
Binary Dot (Inner Product)
Convolve
MOATool1
NUM

```
1.00    2.00

3.00    4.00

5.00    6.00
```

Figure 4-9: Operations Menu in the MOA testing Tool

For example: to do a drop operation for the above defined MOA array, the user clicks the Partitioning menu, and choose drop, the dialog in figure 4-10 will be displayed to enter the index dimension and values for the drop function. Then, the result is displayed in a new document as shown in figure 4-11

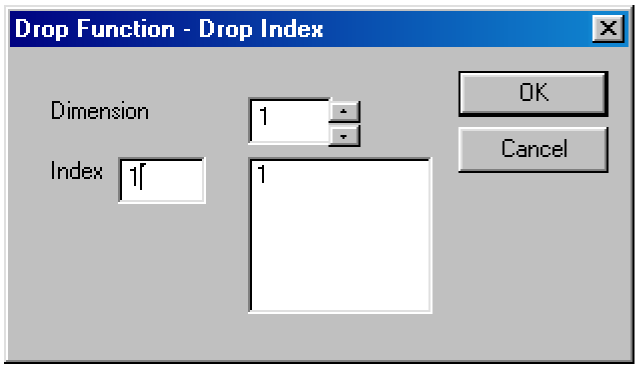
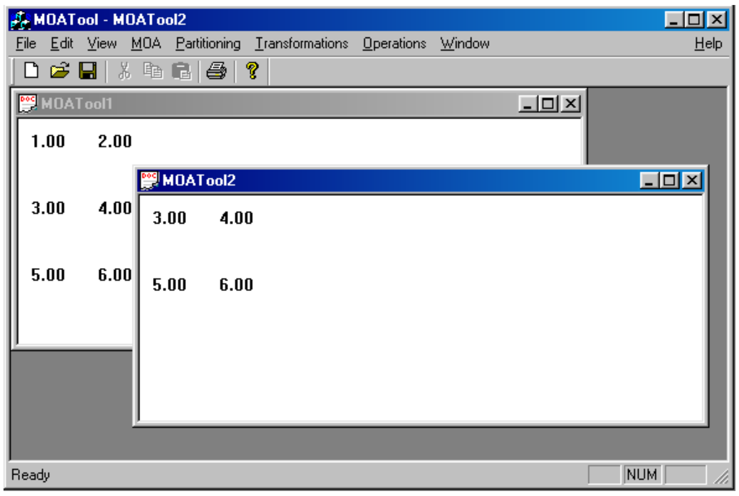


Figure 4-10: Drop Index Interface Dialog in the MOA testing Tool

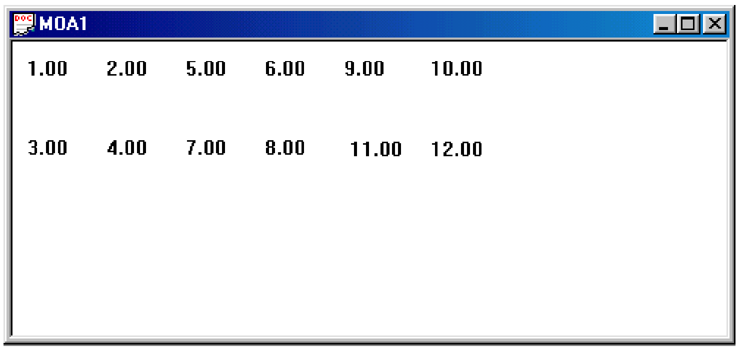

MOATool - MOATool2
File Edit View MOA Partitioning Transformations Operations Window Help
MOATool1
MOATool2
Ready
NUM

```
MOATool1
1.00    2.00

3.00    4.00

5.00    6.00

MOATool2
3.00    4.00

5.00    6.00
```

Figure 4-11: The Drop result displayed in the MOA Testing Tool

The elements of a MOA structure are displayed as data diagrams showing the data of the result in a structured way. The diagrams give immediate feedback on the effect of operations or definitions in development on trial data, showing the arrangement of the data elements flat array based on the dimension and shape defined, or resulting from a computation. Testing in this style is very effective because the results of the intermediate functions being developed is checked as you go along, to map a computation to a series of MOA operations.

The arrangement of data elements on the screen reflects the shape vector of the MOA structure in hand. Diagrams like the ones generated by Nial, (with tables enclosing the slices of the different dimensions in the multidimensional arrays) can be achieved by further enhancement on the drawing function, to be able to draw tables corresponding to the shape vector as well. The interface of a MOA structure of shape vector = $\langle 3 \quad 2 \quad 2 \rangle$, looks as displayed in figure 4-12.



```
1.00    2.00    5.00    6.00    9.00    10.00

3.00    4.00    7.00    8.00    11.00   12.00
```

Figure 4-12: A Three-dimensional MOA array displayed in the MOA Testing Tool

Figure 4-12 understood as:

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

## 4.4 Completeness Discussion

The MOA library implemented in this thesis operates on regular (rectangular) infinite-dimensional arrays. The completeness claim stems from the fact that all notation introduced in [1] is implemented in this thesis and is tested to prove its correctness. Initial functions are the dimn, shape, ravel, Psi, gamma, and NextIndex. All other functions are based on these basic functions and on each other. Higher constructs built calling these functions, could be countless and/or domain dependant. Take is based on iterations of indices, and Drop is based on the take function, while reshape, transpose, rotate and reverse all call the gamma function. The reduction function is based on internal iteration using the NextIndex function, and the scan function depends on the reduction. Inclusion operations are already tested on several functionalities using function IsContainElement. Scalar and Point-wise operations are implemented, while projection operations can be implemented using the shape vector to allocate projections. Figure 4-13 displays the CMOA class methods in a diagram. The hierarchical relationship is implicit in the diagram, since the detailed relations are too complicated to include in one diagram.

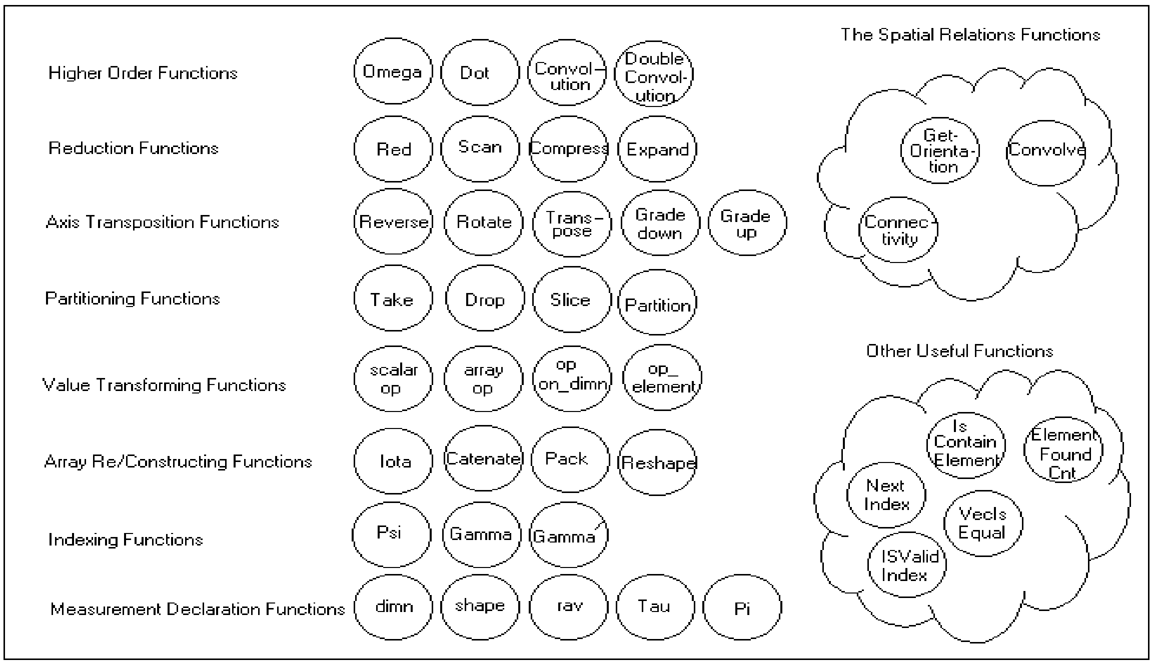


Figure 4-13: CMOA Methods Diagram

The transitive closure of the MOA library implemented in this thesis is proved by the totality of the operations implemented, meaning that these operations can be applied to all arrays invariant of dimension and shape, and terminates with a return of a value. The closure is expressed in the fact that all functions return MOA structures that are subject to further MOA operations.

## 4.5 Summary of MOA Class Methods

Table 4-1 lists the functions implemented in the CMOA class, and relating them to the MOA notation defined in chapter three, and the Array Theory (AT) constructs as explained in chapter two.

Table 4-1 Summary of MOA Class Methods corresponding to MOA & AT

| **Function** | **Comment** | **MOA Symbol** | **Array Theory** |
|---|---|---|---|
| MOA_cls | Class Constructor | | |
| ~MOA_cls | Class Destructor | | |
| GetMOA, SetMOA, CopyMOA | | | |
| Dimn | See chapter 3. | $\delta$ | *valence* |
| Shape; SetShape | See chapter 3. | $\rho$ | *shape* |
| Psi | See chapter 3. | $\psi$ | *Pick, choose* |
| Ravel; SetElements | See chapter 3. | $rav$ | |
| Pi | See chapter 3. | $\pi$ | |
| Tau | See chapter 3. | $\tau$ | *tally* |
| Gamma | See chapter 3. | $\lambda$ | |
| Gamma_Inverse | See chapter 3. | $\lambda'$ | |

Table 4-1 Summary of MOA Class Methods corresponding to MOA & AT - Continued

| | | | |
|---|---|---|---|
| Reshape | See chapter 3. | $\vec{s}\rho\xi$ | *reshape* |
| Catenate | See chapter 3. | $\xi_1 \oplus \xi_2$ | *Link, append* |
| Pack | | | *Pack* |
| array_op | See chapter 3. | $\xi_1\ op\ \xi_2$ | *minus, plus, … etc* |
| scalar_op | See chapter 3. | $\sigma\ op\ \xi$ | *minus, plus, … etc* |
| op_on_dimn | Same as above, but returns the result of an input operator. | | |
| min_on_dimn | Returns the minimum value among the elements on the specified dimension in the input array, calling the previous function. | | |
| max_on_dimn | Same as above, but returns the maximum. | | |
| average_on_dimn | Same as above, but returns the average. | | |
| sum_on_dimn | Same as above, but returns the sum. | | |
| Op_element | Applies an operator on all the elements of an MOA array., also, found in min_element, max_element, avg_element, and sun_element. | | |
| Iota | See chapter 3. | $\iota\ \ n$ | *Tell* |
| Take | See chapter 3. | $\uparrow$ | *Take, takeright* |
| Drop | See chapter 3. | $\downarrow$ | *Drop, dropright, rest* |
| Partition | Takes two indices, one specifies the starting point in the array, and the other specifies the size of each dimension to take. Based on take and drop functions. | | *Sublist* |
| Red | See chapter 3. | $red$ | *reduce, leftreduce* |
| Grade_Up | Ascending sorting. | $gu$ | |

Table 4-1 Summary of MOA Class Methods corresponding to MOA & AT - Continued

| | | | |
|---|---|---|---|
| Grade_Down | Descending sorting. | *gd* | |
| Scan_op | See chapter 3. | | |
| Reverse | See chapter 3. | $\phi$ | *Reverse* |
| Rotate | See chapter 3. | $\theta$ | |
| Transpose | See chapter 3. | $\Phi$ | *Transpose, fuse, rows, cols* |
| Compress | See chapter 3. | /● | |
| Expand | See chapter 3. | ● \\ | |
| Slice | See chapter 3. | $\subset$ | *Split* |
| Omega | See chapter 3. | $\Omega$ | |
| Dot | See chapter 3. | ● | *INNER, OUTER, cart* |
| IsValidIndex | Tests the input index to be a valid complete index in the input array or not. | | |
| Next Index | See chapter 3. | | |
| VecAssignTo | Assigns all the elements values in input array 1 to input array 2. | | |
| VecIsEqual | Checks if two arrays are equal in size and elements values. | | |
| IsContainElement | Checks the availability of an input value in any of the elements in the input array. | | |
| ElementFoundCnt | Returns the number of availability of an input value in the elements of an input array. | | |
| Connectivity | See chapter 3. | | |
| GetOrientation | Returns the recommended orientation to be used in the align and the convolve functions, based on the position of the center-point of the structure and the borders identification. | | |

Table 4-1 Summary of MOA Class Methods corresponding to MOA & AT - Continued

| | | | |
|---|---|---|---|
| Convolve | Works as convolution, except that it returns the next partition around the center point provided as input, so it convolves the partitions of an input array one by one, so multiple processing on the required partition can take place. | | |
| Convolution | Applies convolution of a mask on an input array. | | |
| DoubleConvolution | Works as convolution using two input masks, and use the resulting value, according t an input operation. | | |

# *Chapter 5*

# APPLICATIONS OF THE MOA LIBRARY

## 5.1 Introduction

The MOA Library can be used by any scientific or engineering application that operates on large data-structures and arrays. The library is useful in structuring the code and avoiding the array computation details that cause so many errors, and in obtaining implicit concurrency processing in the executable code. Since it is noticed that more than half of the total design effort is consumed by the verification and the simulation, especially in the area of the multidimensional signals, that exhibit a large amount of related control flow expressed in terms of loops. Memory in these cases is considered a bottleneck, requiring an optimization for the global memory use and communication, leading to the fact that loops and index manipulation become crucial issues. This problem is addressed in the MOA library design, and relating DSP (convolution, FIR filters, FFT, … etc.) problems to the MOA notation provide solutions to some of these problems. Image processing applications, fluid dynamic management, and weather forecasts applications can benefit from the MOA notation. Besides the LAN distributed systems organization, and the Internet and Intranet IP address generation and resource management, all can make use of the library. In the experimentations of this thesis, Image and Video files are mapped to the MOA structure and processed using the MOA notation. The nesting of the MOA structure in these two examples provide a high order functionality efficiently.  The first section of this chapter discuss the CMOAImage implementation, and the second section discusses the CMOAVideo class implementation.

## 5.2 Image Processing Using MOA

Image processing is one field of application where MOA can be extensively used to represent the image and apply processing on it using the above-explained notation. The image can be represented in 1, 2, or 3-dimensional MOA structures. However, in this thesis only the 2-dimensional images will be experimented. Almost all the image processing operations can be applied using the MOA calculus as will be demonstrated in the coming sections. The MOA structure design is suitable for the spatial domain methods, since it allows the direct manipulation of image pixels using Cartesian coordinates. The frequency domain, which is based on modifying the Fourier Transform of an image, can be investigated, by using a Fast Fourier Transform function on the image pixels, after defining the real and imaginary parts. Fourier Transform can be defined in terms of point-wise operators, maintaining the shapely operations by having the dimension and shape as parameters. However, this was out of the scope of this research.

### *5.2.1 Image MOA Format*

The experimentation in this thesis used the bitmap (bmp) format as the base for the testing. Other image specifications were considered in the analysis of the image format to read in an MOA structure of 2-Dimension, like different color tables, compression, and file formats. The bitmap file consists of header information represented in the BITMAPFILEHEADER, the BITMAPINFOHEADER, and the RGBQUAD structures, followed by the stream of image pixels. This structure is read and an MOA structure is formed with dimension equals 2, and a shape vector of two elements the first represents the height of the image, and the second represents the width, as illustrated in figure 5-1.

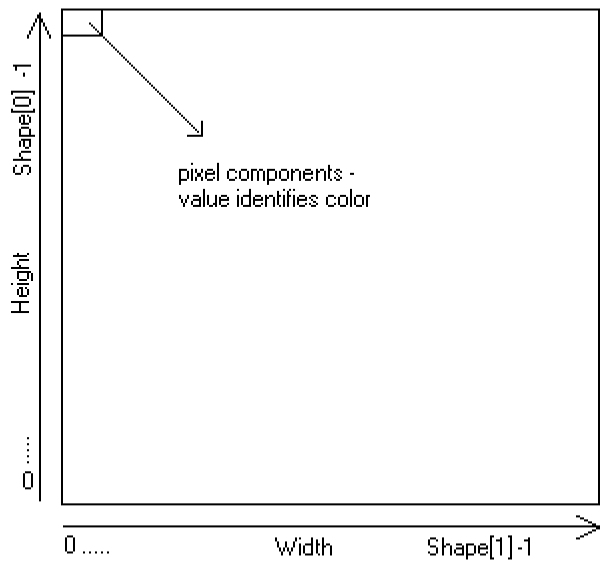


Figure 5-1: MOA Image Representation

The values of the image pixels (The MOA elements) are the pixels color values, decided by the image specification, like an 8-bit color image (256 colors), which is read such that every byte in the memory pixels stream read from the bitmap file is an element in the MOA structure. A 16-bit color image is read such that every two bytes in the pixels stream, represents one element in the MOA structure elements, and so forth. The spatial representation of the image in MOA and the connectivity function discussed above, can be augmented by notions of north, south, east, or west, in correspondence to right, left, up, or down. These notions can be used for navigation and image interpretation as will be discussed below. The following image-processing functions are implemented using the MOA library:

### *5.2.2 Transformation*

Transformations are required to visualize the image from different perspectives based on the special relationships between the pixels in an image.

Using the MOA functions reverse and transpose, several types of transformations can be applied on the image MOA structure. A reverse on dimension 1, causes a vertical flip, and a reverse on dimension 2, causes a horizontal flip. Transpose causes a flip on diagonal. The transpose function can be changed to be able to identify which diagonal to transpose on. Figure 5-2 demonstrates several transformations operated on an image using its MOA structure.

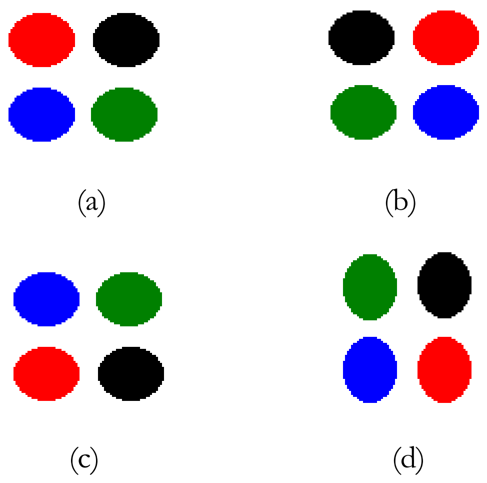


Figure 5-2: Image Transformations by MOA; (a) original image; (b) Horizontally flipped image; (c) vertically flipped image; (d) Transposed image (Rows & Columns Transpositions)

Other types of operation can be applied on an image, for example stretching can be done by the reshape function after changing it to assign the elements of the new 2-dimension MOA array with values identical to the original image in the pixels, which are relatively mapped, thus keeping the ratio in size with the original image. In the newly added pixels, between the original ones, the function assigns values after applying morphology with different dilation and erosion percentages based on the distance between the new image and the eight adjacent pixels from the original image coming from all directions around the new pixel. The symmetrical relations that are kept during the transformations can enhance graphic design in general.

## 5.2.3 Convolution and Applying Filters

Filters are applied for a specific image enhancement required by an application. Image convolution is based on the general array convolution as expressed in [2] MOA notation with the following equations:

$M^2$ is a Mask array with shape <3 3>:

$$rav \quad M^2 \quad = \quad \langle m_0 \quad m_1 \quad m_2 \quad m_3 \quad \ldots \quad m_8 \rangle$$

and $D^2$ is a data array with shape <100 100>:

$$rav \quad D^2 \quad = \quad \langle d_0 \quad d_1 \quad d_2 \quad d_3 \quad \ldots \quad d_{\tau D-1} \rangle$$

The result is array $R^2$ with shape <100 100> calculated by the following equation:

$$R \quad = \quad {}_{+}red \quad T_i^2 \qquad \textbf{Equation 5-1}$$

where:

$$T_i^2 \quad = \quad \vec{i} \quad \psi \quad M \quad * \quad (\vec{b} \quad \Delta \quad \vec{i} \quad \nabla \quad D) \quad \forall \quad 0 \quad \le \quad \vec{i} \quad < \quad *\rho M$$

and $\vec{b} \quad = \quad \langle 98 \quad 98 \rangle$

The equation can be written in programming form of calling the above defined MOA APIs and structures as inputs and output of functions to be as follows:

Convoluted_Array = Red + (Array_Op *, (take ( drop (Input_Array, dr_vec), tk_vec), 0 , Mask_Array) ➔ for all elements in the Input_Array ($a_0$, $a_1$, … , $a_{Tau(A)-1}$).

Where dr_vec, and tk_vec are the indices vectors defined for the drop and take functions for every element in the array A, according to its position in the array to iterate through the partitions of the Array matching the Mask size, and with center point being the point at iteration, covering all points in the input array. Figure 5-3 graphically sketches the convolution operation.

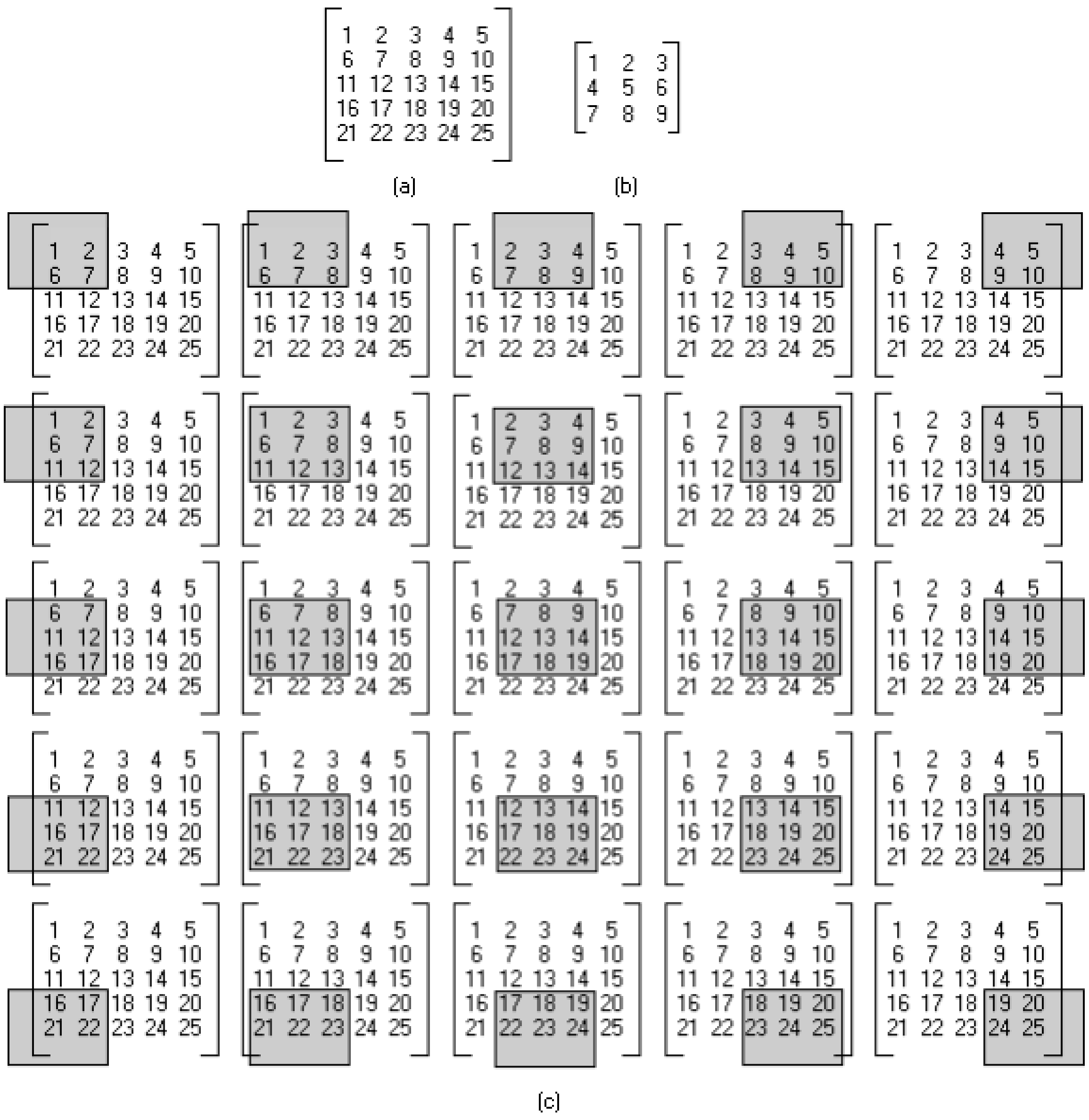


Figure 5-3: Convolution Operation: (a) Input Array – Image Pixels; (b) Mask Array; (c) Mask spanning Input array to compute the value of the center point of every spanned region.

The low pass, high pass, prewitt, and sobel filters are all variations to the above convolution with different Masks, to produce different effects. In the low and high pass filters a simple convolution operation is called as described above with the Masks defined below as input. In the prewitt and sobel filter a more complicated convolution is required because we have two masks in each filter to be convoluted with the input array at the same time, and the resulting point in every time is compared and the smaller value of both convolutions is the value chosen for the resulting array. To use the same function defined above without changing it, we can call the convolution procedure twice, and then call the array_op function (the point wise extension) with operation "<" as input, which chooses the smaller value of the two equally indexed elements in both input arrays to be the values of the equally indexed elements in the resulting array. This will result in computing time equal to O(3n), where n is the number of elements in the input array (Tau of input shape : $\tau\xi$). This is because the elements of the input array will be traversed two times in the two convolution calls, and then a third time in the point wise extension.

To enhance the performance, another function was designed that takes the two masks, and applies the two convolutions at the same iteration, and applies the comparison to include in the result the smaller value, all at one shot. The computing time will then be enhanced to O(n + f), where f is extra time consumed in the second convolution (multiplication and the addition reduction), and the comparison. This enhancement can be expanded to allow the input of variable number of Masks, defined at run time, and all processes at one shot, having the operation that decides how the results of these convolutions are reduced to the resulting array as an argument, like "<", ">", "+", … etc.

The following are the Masks that apply the above-explained filters. Figure 5-4 illustrates the original image that most of the effects in this chapter will operate on.

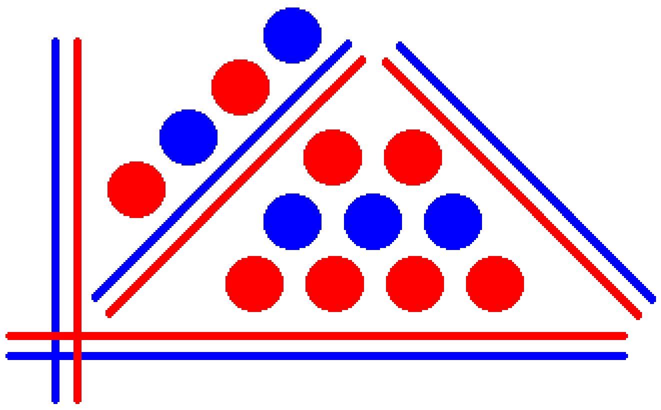


Figure 5-4: Original Image

Figure 5-5 is the effect of the low pass filter using the following mask on the image of figure 5-4.

Low Pass Mask:

$$M = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

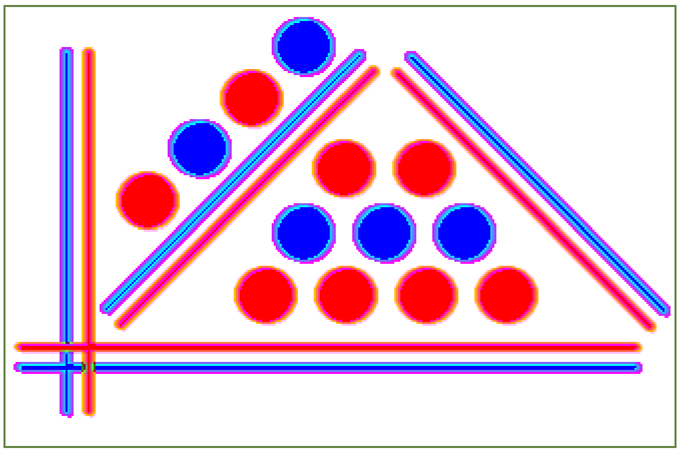


Figure 5-5: Low Pass Filter Effect

Figure 5-6 is the effect of the high pass filter using the following mask on the image of figure 5-4.

High Pass Mask:

$$M = \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

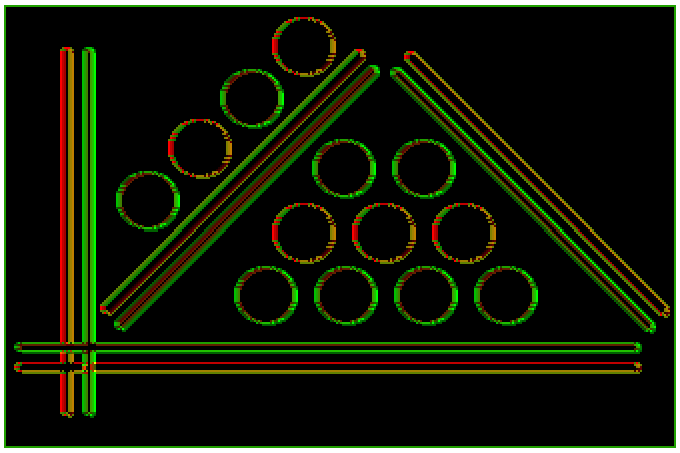


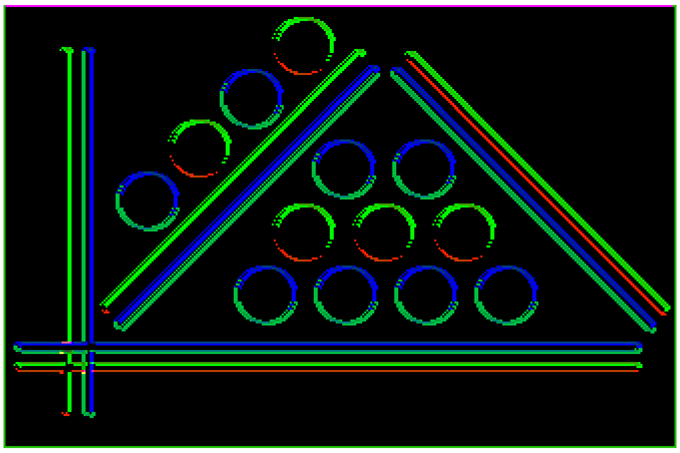
Figure 5-6: High Pass Filter Effect – Point Detection

Figure 5-7 is the effect of the Prewitt filter using the following masks in a double convolution function on the image of figure 5-4.

Prewitt Masks:

$$M_1 = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$



Figre 5 -7: Prewitt Filter Effect

Figure 5-8 is the effect of the Sobel filter using the following masks in a double convolution function on the image of figure 5-4.

Sobel Masks:

$$M_1 = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

$$M_2 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

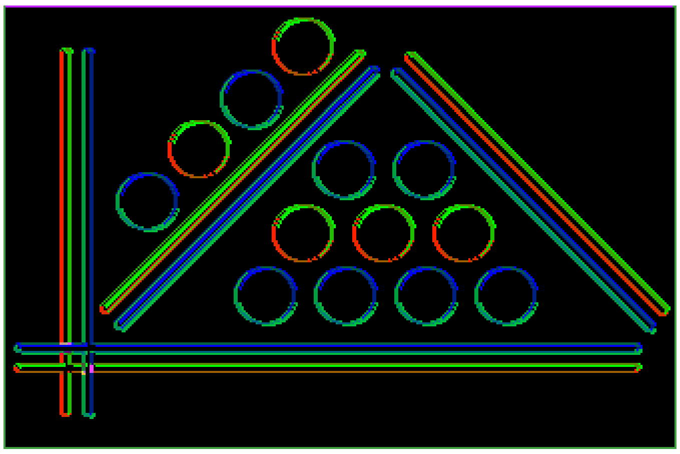


Figure 5-8: Sobel Filter Effect

Figure 5-9 is the effect of the following custom mask on the image of figure 5-4.

Custom Filter Mask:

$$M_2 = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & 2 \\ -1 & -1 & -1 & -1 & -1 & 2 & -1 \\ -1 & -1 & -1 & -1 & 2 & -1 & -1 \\ -1 & -1 & -1 & 2 & -1 & -1 & -1 \\ -1 & -1 & 2 & -1 & -1 & -1 & -1 \\ -1 & 2 & -1 & -1 & -1 & -1 & -1 \\ 2 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

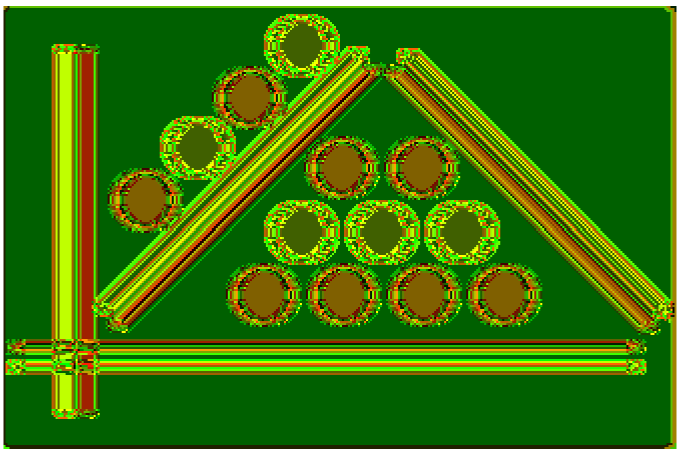


Figure 5-9: Custom Filter Effect

Morphology operations are also variations of the convolution operation described here. The difference is that no Mask is passed as an argument to the operation. The input array is scanned like the traversing scheme presented above, selecting the minimum pixel value of the selected region (in case of erosion) to be assigned to the value of the output image. Erosion removes spurious pixels (noise) and thin boundaries of objects on a dark background (object of pixel values greater than the background values). Dilation is applied the same way, but the maximum value of the selected region is the value assigned to the output center pixel value. Dilation fills up holes and thickens boundaries of objects on a dark background (object of pixel values greater than the background values). Applying erosion followed by dilation is called opening the image that has the effect of eliminating small and thin objects, breaking objects at thin points, and generally smoothing the boundaries of larger objects without significantly changing their area. Also, applying dilation followed by erosion is called closing the image that has the effect of filling small and thin holes in objects, connecting nearby objects, and generally smoothing the boundaries of objects without significantly changing their areas.

Figure 5-10 is the effect of the dilation by spanning one neighboring level on the image of figure 5-4.

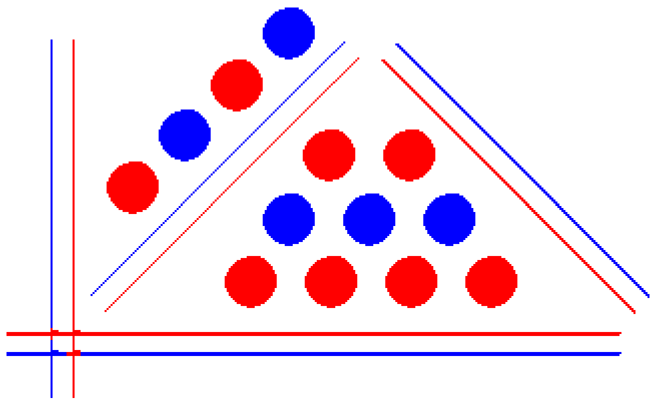


Figure 5-10: Dilation Effect

Figure 5-11 is the effect of the erosion by spanning one neighboring level on the image of figure 5-4.

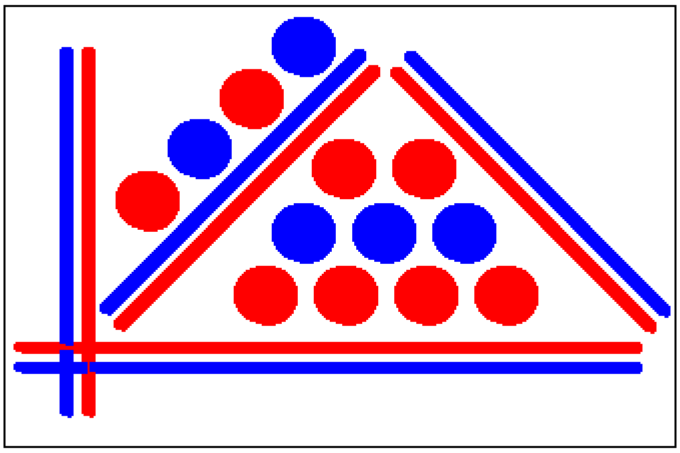


Figure 5-11: Erosion Effect

Figure 5-12 is the effect of opening the image of figure 5-4, by spanning one neighboring level.

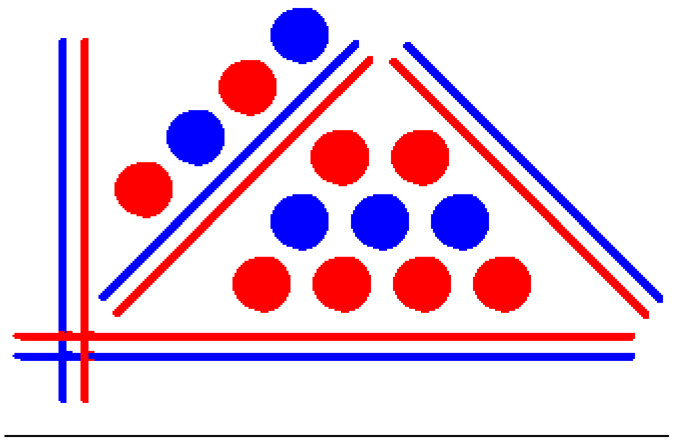


Figure 5-12: Opening Effect

Figure 5-13 is the effect of closing the image of figure 5-4, by spanning one neighboring level.

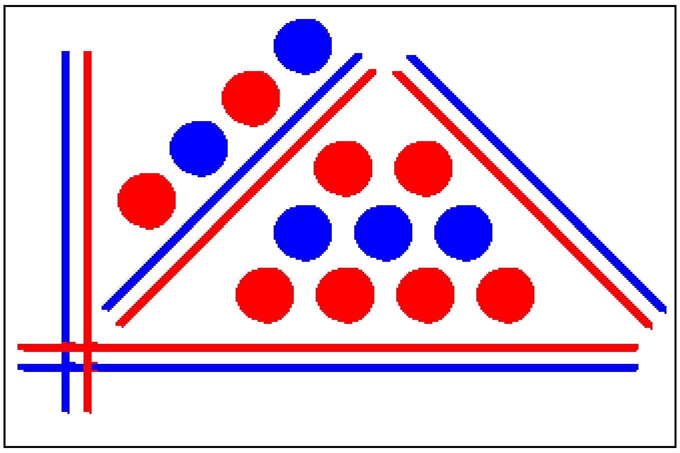


Figure 5-13: Closing Effect

### *5.2.4 Segmentation*

Segmentation of the image leads to image analysis, so that we can extract information from it. It is the process of dividing the image into constituent parts or objects. From the basic definition of image segmentation, how to apply it on MOA can be visualized. Point, Line (horizontal, vertical, $+45^\circ$, and -$-45^\circ$), Edge and Combined Detections are applied by using Masks that are convoluted on the image plane to detect discontinuities in the specified manner described in the provided mask.

The Point Detection Mask uses the same mask as the high pass filter mask explained before, and has the same effect as shown in figure 5-6. Figure 5-14 shows the effect of detectingthe horizontal line using the following mask on the image of figure 5-4.

Horizontal Line Detection Mask:

$$M = \begin{bmatrix} -1 & -1 & -1 \\ 2 & 2 & 2 \\ -1 & -1 & -1 \end{bmatrix}$$

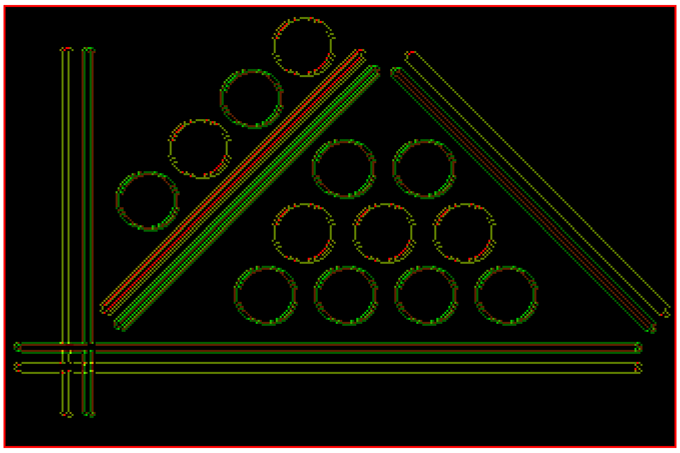


Figure 5-14: Horizontal Line Detection

The $+45^{\circ}$ Angle line is detected using the following mask, and is illustrated in figure 5-15, as applied on the original image of figure 5-4.

$+45^{\circ}$ Line Detection Mask:

$$M = \begin{bmatrix} -1 & -1 & 2 \\ -1 & 2 & -1 \\ 2 & -1 & -1 \end{bmatrix}$$

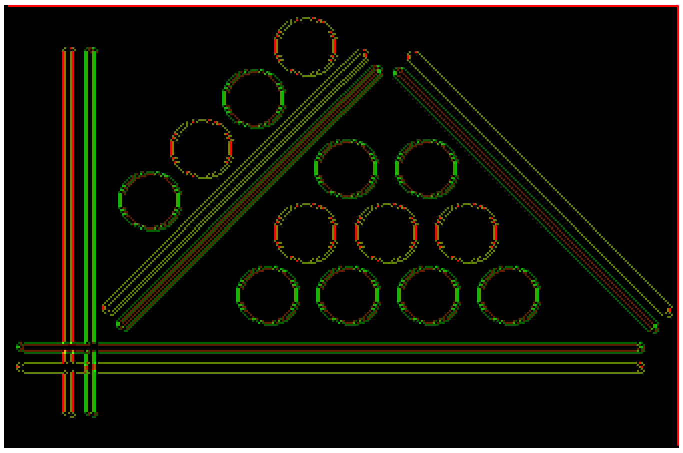


Figure 5-15: 45 Angle Line Detection

Figure 5-16 shows the effect of detecting the horizontal line using the following mask on the image of figure 5-4.

Vertical Line Detection Mask:

$$M = \begin{bmatrix} -1 & 2 & -1 \\ -1 & 2 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

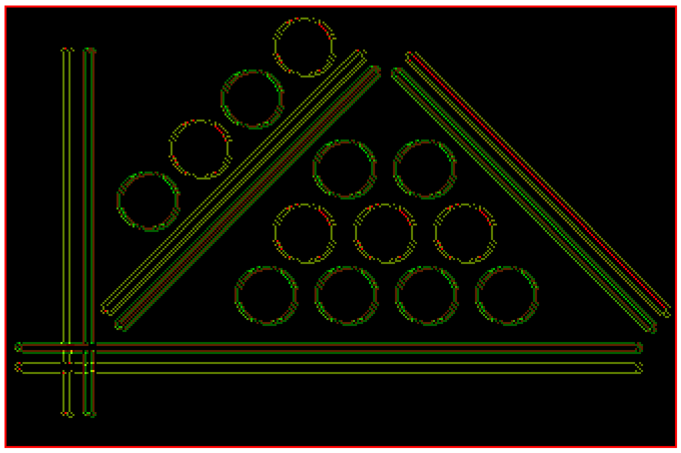


Figure 5-16: Vertical Line Detection

The $-45^{\circ}$ or 315 Angle line is detected using the following mask, and is illustrated in figure 5-17, as applied on the original image of figure 5-4.

$-45^{\circ}$ Line Detection Mask:

$$M = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}$$

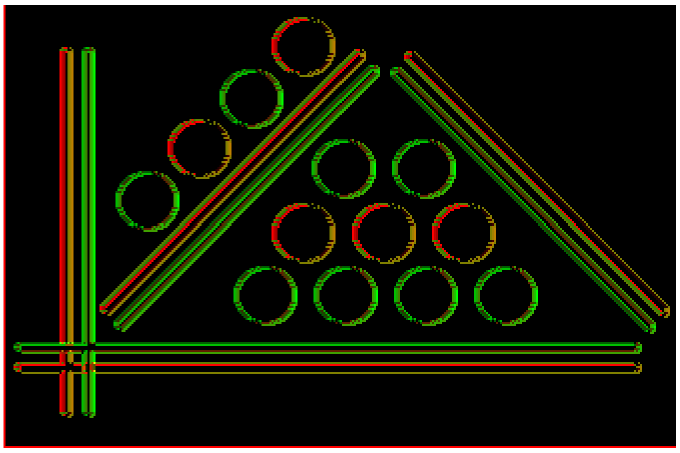


Figure 5-17: 315 Angle line Detection

The Edge Detection is implemented through discontinuities detection, or thresholds (to eliminate noise) or Region detection. The Region oriented segmentation is based on finding boundaries between regions using intensity discontinuities or on thresholds based on distribution of pixel properties (intensity, color, gray-level, texture). Region segmentation is achieved by finding regions directly by the Region Growing method through pixel aggregation, which starts by "seed" points and adds to them neighboring pixels with similar properties. Another direct method is the Region splitting and Merging, where an image is sub-divided initially into a set of arbitrary, disjoined regions and then merges and/or splits them in order to satisfy the region segmentation conditions (complete, connected, disjoint, same properties within a region, and different properties among the regions). [8]

The region segmentation is a process that associates different MOA Image classes for every region detected. These different regions can be operated on separately as independent images, and then mapped to the original image. Otherwise, Regions can be represented as a reference to the indices of the regions' pixels in the original image, where any further computations are applied directly to that specific region in the original image.

### *5.2.5 Representation and Description*

The regions segmented in the above section now need to be represented either on the shape of its boundaries (externally), or in the characteristics of the pixels in a region (internally). A Representation scheme is required to be chosen, then a description of the representation of a region is to be selected according to the external or internal features of the region (invariant of transformations like change in size, translation, rotation, … etc.). Representation schemes are Chain Codes, Polygonal Approximations, Signatures, Boundary Segments, and the Region's Skeleton. Chain Codes is a connected sequence of straight-line segments of specified length and direction that describes the boundary of an object based on its 4-connectivity or 8-connectivity. Chain codes are represented on a grid of equally spaced x and y coordinates, where every pixel is assigned a value of the direction connecting every pair of pixels. Evidently, the implementation of the chain code is just another 2-dimension MOA structure that their elements contain the representation of an object. The same understanding of image representation in MOA can be applied with other representation Schemes as discussed in [8]. The representation MOA structures can be formed by object segmentation of images to define object boundaries, and fed to a database of objects, to which images are compared to be identified and described.

The image description can be expressed as the length of the contour, the value and orientation of the diameter, curvature (rate of change of slope), shape numbers, moments, number of pixels in a region, compactness of a region ((perimeter)$^2$ / area), topology of a region, statistical texture (smoothness, coarseness, and granularity), structural texture (regularity), spectral texture (based on Fourier spectrum), or relational descriptors. For further details on these kinds of image descriptors, refer to [8]. The description can be implemented in MOA the same way it is implemented traditionally. The experimentation in this thesis did not cover the image description for recognition purposes in depth. However, the experimentation conducted can conclude that it is feasible to implement image description based on the MOA basic class and Image class implemented in this thesis.

### *5.2.6 Recognition and Interpretation*

Image recognition and interpretation is the process of analyzing the image to discover, identify and understand image patterns. So, the computer needs to be fed with patterns of the interested image-based task, and recognition is the process of approximating an input pattern to one of the base patterns. Recognition can be implemented by decision-theoretic methods, structural methods, or image interpretation methods. Then interpretation (a process of assigning meaning to an ensemble of recognized image patterns or elements) is implemented as a knowledge base for a problem domain based on predicate logic, semantic networks, or production (expert) systems. The implementation of knowledge representation in MOA is an emulation of the Nial AI toolkit described in [17]. The array methods used in Nial are the same implemented in the MOA library, and hence, AI applications designed based on Nial can be experimented in MOA. So far, the image is represented in its spatial structure as a matrix of pixels on the image plane. Another representation of hierarchical structures for complex images can be added, where layers of the image are entered as another dimensions in the MOA structure. Lower Levels in the hierarchy represents details information about the image (this could be useful also for GIS – Geographic Information System – applications where complex images denote maps). Image interpretation can be implemented as literal or prepositional knowledge, where literal knowledge describes the meaningful parts in the image and their locations, constructing an MOA structure attached to the image structure, and prepositional knowledge captures the non-spatial features of the image. This method is cognitive and captures the inherent parallelism as will be discussed in the next chapter.

In Nial, complex images (hierarchical, or multi-dimensional images as can be represented in MOA), are considered recursive. In MOA, recursion is totally avoided because of the explicit structure of the multidimensionality of the arrays. The north (up), south (down), … etc, notion discussed earlier, can be used now in relation with the recognized patterns, to extract relations of which object is *above* / *below* which. Other special relationships can be added relative to the problem-domain. The approach discussed in [17] allows for three categories of imagery cognitive inferences, which are image construction, image transformation, and image access. The image construction can be extracted from a frame data, from pattern database by description attached to these patterns, or through a perception and recognition process.

Then, object extractions from background, learning from examples to generalize knowledge, and inference from incomplete information can be implemented based on arrays as discussed in [17].

### *5.2.7 Image Compression*

Image compression can be expressed in MOA using the explicit compress function. However, in using the expand function to restore the image, this will be a lossy compression as the examples in chapter three demonstrate, in which the skipped elements in the compress functions, will be restored as zeroed elements in the expand function, with the a Boolean vector describing the original shape. Averaging functions can be defined for averaging the skipped elements values with the non-skipped ones using the Connectivity function in MOA, so that the expansion using an inverse operation will be non-lossy. To fully utilize these two functions (compress and expand), further functions need to be implemented to construct a valid Boolean input MOA array, instead of depending on the awareness of the user of the expected valid input. Several Boolean array construction functions can be defined to serve different compression and expansion requirements. These functions are not implemented in this thesis, because they will require the definition of a new file format for the images based on the MOA structure.

### *5.2.8 Summary of MOA Image Class Methods*

The above image processing operations were implemented in a new class called CMOAImage that interfaces the bmp file format with the MOA structure, and performs the image processing functions by calling MOA notation as described in previous chapters. The CMOAImage class performs only on 2-dimensional MOA arrays. Appendix B contains the header file for this class, showing the argument lists and the return types of functions. Table 5-1 summarizes the CMOAImage Methods.

Table 5-1: Summary CMOAImage Class Methods

| Name | Description |
|---|---|
| ImageToMOA | Converts different formats of an Image (handle to bmp file, or BITMAPINFOHEADER and image pixels stream) to MOA Image structures. |
| MOAToImage | Converts MOA Image structures to different formats of an Image (handle to bmp file, or BITMAPINFOHEADER and image pixels stream). |
| VerticalFlip | Calls MOA reverse operation on dimension 1 |
| HorizontalFlip | Calls MOA reverse operation on dimension 2 |
| TransposeImage | Calls MOA transpose operation to switch rows to columns and vice versa. |
| LowPassFilter | Applies convolution with low pass filter mask defined before. |
| HighPassFilter | Applies convolution with high pass filter mask defined before. |
| PrewittFilter | Applies convolution with Prewitt filter mask defined before. |
| SobelFilter | Applies convolution with Sobel filter mask defined before. |
| CustomFilter | Prompts the user to define an MOA structure to be used in convolution over the current image. |
| DetectPoints | Convolves the image using Point detection mask, defined to be the high pass filter in this thesis. |
| DetectHLine | Convolves the image using horizontal line detection mask, defined before. |
| DetectVLine | Convolves the image using vertical line detection mask, defined before. |
| Detect45Line | Convolves the image using 45-angle line detection mask, defined before. |
| Detect315Line | Convolves the image using 315-angle line detection mask, defined before. |
| Dilation | Convolves to choose the maximum pixel value in a specified span area. |
| Erosion | Convolves to choose the minimum pixel value in a specified span area. |
| Opening | Calling the erosion function followed by dilation function on the specified span area. |
| Closing | Calling the dilation function followed by erosion function on the specified span area. |

## 5.3 Video Processing Using MOA

Video processing is another field where MOA can be beneficial. The AVI file format is designed to contain Audio Video Interleaved streams. This means the images forming the frames of a video file, are stored in the form of streams, then another layer of an audio stream can be combined with file. This structure complicates the video processing operation. Since streams will need to be processed one by one, and will require more effort from the programmer to isolate frames while being able to process them collectively in a symmetric manner. The MOA representation for a video file is a 3-dimensional MOA array structure, where the first dimension is the time sequencer or the frame number, and the remaining two are the height and width of the frames images. Video processing operation on the Video frames can be processed individually using the psi MOA indexing function to return the image of the frame number required, or collectively by working on all the frames at one shot (by applying all operations on the second dimension).

### *5.3.1 Video MOA Format*

The CMOAVideo class provides the main interfacing between the MOA structure defined in the CMOA class and the AVI file format. The main objective achieved from this class is the decomposition of the AVI video stream into the frames of the images, which the video displays. The pixels of all these frames are mapped to a 3-dimensional MOA structure as shown in figure 5-18. The first dimension is the frame number, and the remaining two dimensions are the height and the width of the images respectively, assuming that all images in a Video file has the same height and width.

### 5.3.2 Transformations

All image transformations applied in the previous section are applied on the frames of the video stream stored in a 3-D MOA structure. The forward and rewind operations are a simple positioning of the index of the first dimension in MOA video structure. The reverse function can be applied on the image frames, by reversing the images display, keeping the same order (as the horizontal and vertical flipping illustrated in figures 5-2 (b) and 5-2(c) respectively). The reverse function can be applied on the first dimension only, to reverse the display of the sequence of the frames. The same applies to the transpose function, filters, and morphology functions, as explained in the previous section.

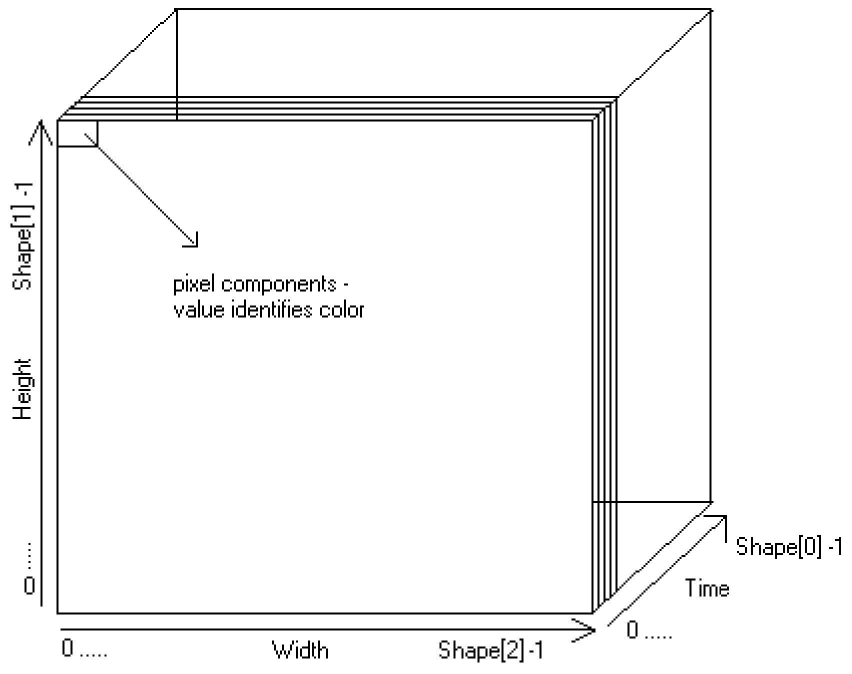


Figure 5-18: MOA Video structure

### 5.3.3 Object Tracking

Object tracking is one important video processing operation. It is based on the image detection process discussed above. Object tracking in general is the motion detection, where motion is the process of extracting objects of interest from a background of irrelevant detail. Motion is defined to be a relative displacement between the sensing system and the scene being viewed. Again this can be implemented in the special domain and in the frequency domain. Since in this thesis, we are more interested in the special domain, only the methods of the special domain will be discussed. As the Video MOA diagram shows how the image in the video stream is represented by the function f(t, x, y) where t is the time where the image is taken or is displayed, and x and y are the spatial coordinates. We must keep also a reference image that captures the stationary components. So, we need to compare two images (frames in a video file in two different time slots) pixel by pixel with the reference image to generate a difference image canceling the stationary components and leaving only non-zero entries that correspond to the non-stationary image components. This can be defined as:

$$d_{ij}(x, \quad y) = \begin{cases} 1 & if \quad |f(t_i, x, y) - f(t_j, x, y)| \quad > \quad \theta \\ 0 & otherwise \end{cases}$$ **Equation 5-2**

where $\theta$ is a threshold that is to be chosen larger than the background intensity. [8]

The difference image will denote the leading and the trailing edges of the moving object, i.e. this shows the area that was occupied by the object in the previous images, and is recently occupied by the object in the new image. Another method to detect motion is the Accumulative difference, which is more suitable to the video frames. This method is based on recognizing the first frame in the video as the reference image, and comparing every frame in the video with the first one, producing an image that corresponds to the accumulative difference using the above equation. In the accumulative difference resulting image, higher values will be given to the areas that were occupied by the moving object in earlier frames and moved away from them, and lesser values will be given to the newly occupied area. This can be realized in three types of accumulative difference images: Absolute Accumulative Difference Image (AADI), Positive Accumulative Difference Image (PADI), and Negative Accumulative Difference Image (NADI). The Absolute will show the path the moving object took and the new object's location, the positive will show only the new location, and the negative will show only the path.

The motion detection scheme implemented using the MOA class, used an xoring algorithm to have better display of the differential image. This updates equation 5-1, to have an xor sign instead of the subtraction minus sign. Figure 5-20 illustrates the frames of an AVI displaying a clock's pointers moving, as decomposed by the CMOAVideo class.

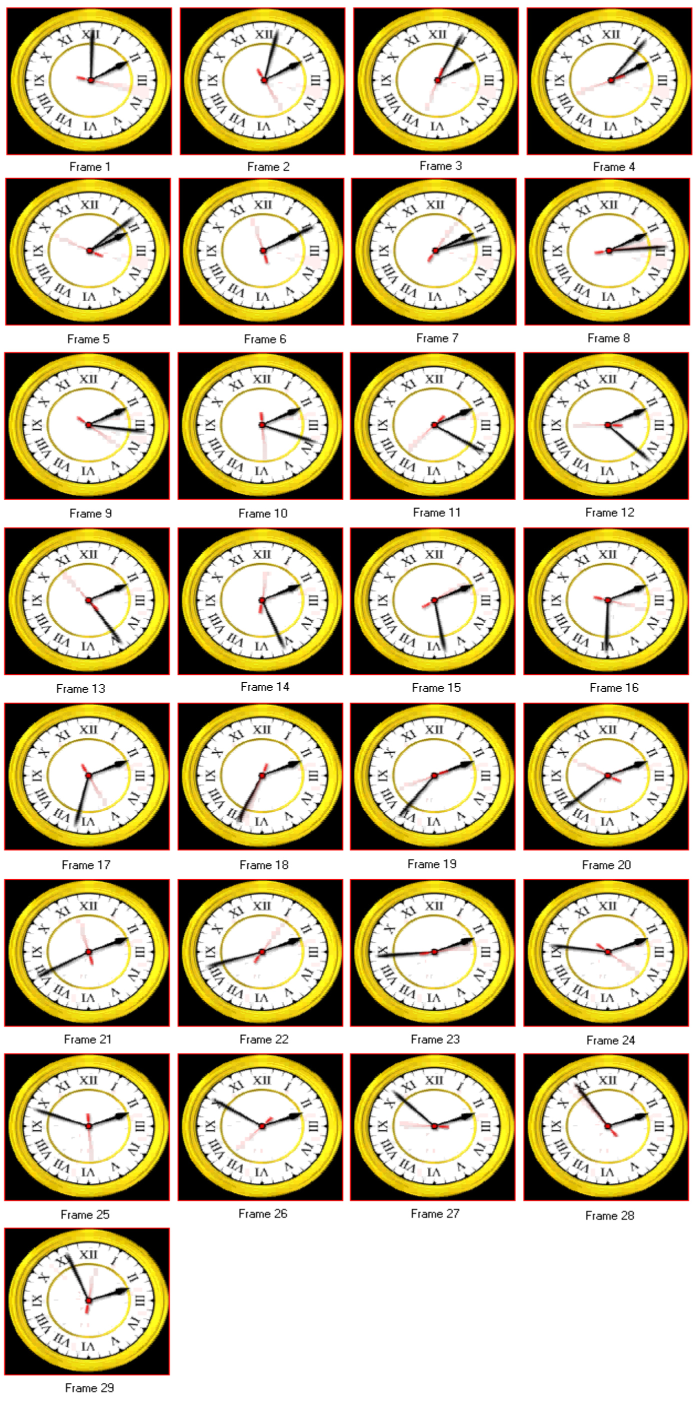


Figure 5-19: Clock AVI Frames Decomposed by the CMOAVideo Class

The motion in the frames of figure 5-19 is detected using the updated equation, resulting in the differential image displayed in figure 5-20.

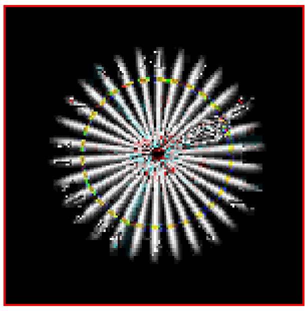


Figure 5-20: Clock Pointers Motion Detection

Another example is illustrated by the frames decompositions of a file copy AVI in figure 5-21, and its motion detected producing the differential image in figure 5-22.

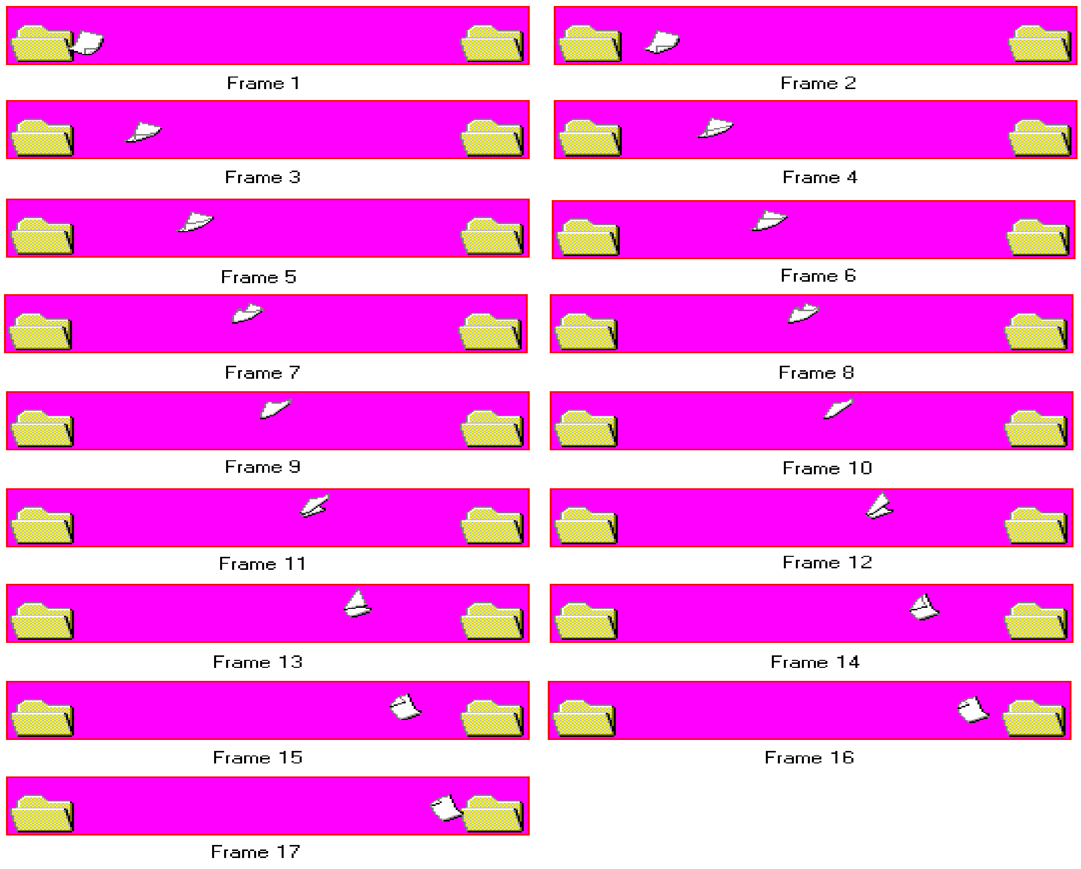


Figure 5-21: File Copy AVI Frames Decomposed by CMOAVideo Class

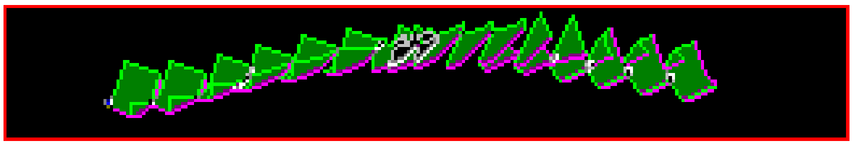


Figure 5-22: File Copy Motion Detection

### *5.3.4 Video and Audio Correspondence*

The Psi-Correspondence theorem (PCT) discussed in chapter three, can be applied by relating the audio stream and the video stream coded in the AVI file format as two MOA structures in correspondence to each other. Streaming is a technique of using a small buffer to play a large file by filling the buffer with data from the file at the same rate that data is taken from the buffer and played.

### *5.3.5 Summary of MOA Video Class Methods*

CMOAVideo is implemented in this thesis to perform the interfacing between the MOA structure and the AVI file format and the Video processing operations by calling MOA notation as described in previous chapters. This class performs only on 3-dimensional MOA arrays. Refer to Appendix C for the header file of this class, showing the argument lists and return types. Table 5-2 summarizes the CMOAVideo Methods.

## 5.4 MOA Applications

The MOA library implemented in this thesis has a wide domain of applications. Only two domains were tested in this thesis. Table 5-3 summarizes the possible domains and their applications including those tested in this thesis, and the other feasible applications.

Table 5-2: Summary CMOAVideo Class Methods

| **Name** | **Description** |
|---|---|
| CMOAVideo | Constructor: Analysis the AVI file |
| InitVideo | Decomposes the AVI file into its streams, and decomposes the Video stream into its images. |
| AVIBitMapInfoHeader | Returns the Info. Header of a specified frame in the video stream |
| AVIGetBitMapImage | Returns the Image pixels stream of a specified frame in the video stream. |
| AviToMoa | Converts she AVI file format to MOA Video structure. |
| MoaToAvi | Converts the MOA Video structure to AVI file format. |
| VideoMOAIntoImage-MOA | Converts a specific frame in the video stream to an MOA Image structure . |
| ReverseFramesOrder | Calls MOA reverse operation on dimension 1 |
| FlipFramesVertically | Calls MOA reverse operation on dimension 2 |
| FlipFramesHorizontally | Calls MOA reverse operation on dimension 3 |
| VideoSubtract | Performs a bit-wise MOA operation (array_op) using subtraction operator among the 2-D Frames in a Video 3-D MOA structure. |
| MotionDetection | Performs a bit-wise MOA operation (array_op) using xor operator among the 2-D Frames in a Video 3-D MOA structure (producing better illustrations for motion). |

Table 5-3: Summary of MOA Applications

| **Application** | **Description** |
|---|---|
| Image Processing & Medical Imaging | Spatial arrangement of pixels in the images plane (width and height), applying all processing required on an image. |
| Video Processing | Frames of images arrangement in a specified time sequencer. |
| GIS Applications | Hierarchical arrangements of maps with different layers of details. |
| Routing Algorithms | Where points on the map are digitized having a defined scale between the spatial coordinates in reality and the coordinates in the MOA structure. |
| Robotics | Visual Perception, autonomous navigation |
| Knowledge Base Systems | Reasoning about spatial knowledge like: recognition of complex objects, motion planning. |

Table 5-4: Summary of MOA Applications - Continued

| | |
|---|---|
| Scientific Applications | Depending on multidimensional array structures and requiring aggregation and proration. |
| Processor Alignment | Distributed Systems, very useful to load balancing, pipelining, distribution and redistribution. |
| Parallelisation Agents | To be embedded in Compilers. |
| Document Editing & Layout | Where characters are the data elements of the array, and the shape is the document width and height. |

## 5.5 Summary

This chapter presented two applications that can benefit from the MOA library. The image processing operations are implemented, representing the image as a 2-Dimensional MOA structure, and The Video processing operations are implemented, representing the video as a 3-Dimensional MOA structure, where operations in both applications are implemented by calling MOA functions as explained in chapter 4. Interfacing with the file formats for the image and the video files, was implemented, and some operations were implemented to prove the applicability of the MOA structure and operations. Moreover, the anticipated applications domains for the MOA library are presented.

# *Chapter 6*

# PARALLELISM, PIPELINING, AND HARDWARE IMPLEMENTATION

## 6.1 Introduction

This chapter investigates further experimentation on the programming invariant of dimension and shape environment implemented in this thesis. Sections 6.2 – 6.4 discuss parallelism, pipelining, and hardware implementation respectively, all related to the MOA library implemented in this thesis. Parallelism and pipelining are discussed theoretically, by relating previous research fields to the implementation of the MOA class. Only section 6.4.2 (on the Hardware implementation), includes a complete implementation in VHDL using Renoir, and simulated by ModelSim, of the MOA class implemented in C++.

## 6.2 Parallelism in MOA programming

The idea behind the MOA is to allow programmers writing array computations sequentially, to write their code in higher-level constructs that allow the compiler to extract parallelism implicitly from the overall structure of arrays and the array algorithms. The process is based on rewriting a sequential solution on a single processor to run on a specific parallel architecture. Since it seems difficult to develop a technology that automatically parallelizes a sequential code, the sequential code will instead need to be rewritten using these higher-level constructs, thus eliminating the use of nested loops and detailed subscript handling.

The performance of parallel computing depends on two factors: i) dividing the work on the available processors, and keeping the overall computing time small, and ii) placing the data, and minimizing the communication between the processors. So, it can be summarized in computation, distribution, and communication. Communication can be several orders of magnitude higher than the computation costs. Arrays are considered one of the major problems that decrease the parallelism effectiveness, the thing that caused recognizing the need to provide language extensions for a more direct expression of the problem in terms of array data structures. There has been a lot of research focusing on providing a mathematical formalization analyzing the array objects introduced into High Performance Fortran (HPF), and High Performance C (HPC), providing a systematic way of describing array operations to one that does minimal work. HPF provides semi-automatic parallelisation through the use of programming directives. These directives describe the data partitioning and dependencies that are communicated to the compiler in conjunction with the parallel tools to achieve an effective solution. The Psi-calculus research direction is based on the belief that the data placement can be deduced from the array expression, using mathematical formalization to describe the parallel architecture as an array of processors, and hence to achieve a reasonable partitioning solution for a non-trivial problem. [6]

The introduction of parallelism into sequential computation using arrays operations was previously studied, by viewing arrays of data and arrays of processors using an equational theory. This way, data is manipulated more efficiently through equivalence preserving transformations. Alternatively, the Psi-calculus defines everything in terms of structure and indexing, allowing the simplification of an arbitrary array expression even on one processor. The unified theory of arrays does not rely on graph theory, set theory, nor data flow analysis. This unified theory is designed to represent the computation and the computational organization in a network of processors, through a high-level mathematical treatment of the problem to capture the inherent parallelism of the computation, and by providing a high level description of the available architecture. Eventually, this allows the use of transformation techniques to convert a high level description into a program that utilizes the architecture effectively.

The objective of the MOA algebra and the associated Psi-calculus is to optimize array computation given a linear address space on two levels. The first level is the *functional normal form*, which is a minimal semantic form expressed in terms of selections using Cartesian coordinates, which employ the Psi Reduction theorem. The second level is the transformation of the *functional normal form* to it's equivalent *operational normal form*, which describes the implementation results in terms of starts, strides, and lengths to select from the linear arrangements of the items. The latter form employs the Psi Correspondence Theorem (PCT). [6]

Ongoing research has been attempting to describe abstract models for architectures, to be used as the basis for mapping decisions. However, the mapping of a computation to an abstract model based on manual efforts is always researched for better automation. There are several models, which provide automation solutions, and enhancement to existing ones. It is crucial to provide automation to this step, making effective use of parallel architecture. The graph theory is often used in designing the abstract model using nodes as processors, and edges as the communication links. The graph theory itself in this manner can be represented as an array of the addresses of the processors, where processors having a link are one address away along one of the axes. This way can describe a list of processors, a 2-dimensional mesh, a hypercube, a balanced tree, or a network of workstations. [6]

A DoAll loop is one form of parallel execution of arrays, where all iterations can be executed in parallel, causing additional communication costs in case the array is not properly distributed across the processors. Some cases of the DoAll loop allow for communication-free array partitioning. These cases are derived in an equations as discussed in [31], using a constant distance vector and a set of independent array elements expressed in a lattice. If the set of independent array elements is the whole array, then the problem is trivial. Other cases can be handled by one block communication partitioning, locating all remote data to one processor, requiring at most one block communication for each processor to get the remote data during computation.

In this thesis, parallelism is discussed in terms of the MOA library implemented in C++, which is considered a step towards enhancing and simplifying the parallelism in C that was proven difficult, since the language was designed to be sequential in nature. So, it is an overhead for the C programmer targeting parallel computing to encode the programs' synchronization, communication operations, and safeguard against race conditions. The separate shape in the MOA structure is discussed in terms of parallel computing. Then a discussion of a previously proposed parallel architecture-mapping scheme follows. Another discussion will address the Block-Cyclic redistribution problem and the algorithm presented in [16] for the Multi-dimensional Arrays. Then, a discussion about the implementation of the Tiling algorithms based on the MOA notation is presented.

### *6.2.1 Shapely Types*

Parallel programming requires defining the shapes of the data structure separately from the data itself, in order to distribute the data among containers before operating on them [18]. This, as well as the fundamental study of the semantics of matrices and arrays, raised a formalization of shape theory. This theory is based on whether the data types are defined as shapely data types or not, where shapely data types are those whose values can be decomposed into their shape and their data. Regular and irregular arrays, graphs, records, variant records, lists and trees are shapely; functions and sets are not shapely. Arrays were considered peripheral to the central functions in functional programming. To parallelize the computing of the arrays, dependent types were introduced, which require intolerable penalty for type checking during execution. So, the shapely types have the right amount of structure for data parallel computing.

Besides, the fixed geometry entailed in physical systems, requiring the conversion from graphs to sparse matrix (decoding and encoding), the programmer using shapely types can replace this process, by low-level processes for handling shapes as discussed in previous chapters. Another advantage for separating shapes from data, is the independence of shape, for which changes such as dimension increase, generalizing from a mesh to an arbitrary graph, does not affect the computation. This is called shape polymorphism, which is a novel form of parametric polymorphism, which allows operations to be parameterized over shapes. Interaction between the shape and the data can be found in the reduction operations (where shape is affected), and matrix multiplication (where shape of the inputs is validated and affects the resulting shape), whereas other functions may not be interacting with the shape. However, this interaction is designed in this thesis such that it is not dependant on the values of the shape vector in order to maintain dimension and shape invariance.

Hence, shapes (which can also be defined to be data structures, containers, indexing systems) are needed in parallel computing to determine communication strategies, load balancing, evaluation strategies, … etc, to the limit that data parallel computing can be defined as shape-based computing. The greater the separation between the shape and the data and the more stable is the shape, the greater is the parallelism achieved. The library implemented in this thesis eliminates the static and the dynamic shape analysis phase (computing shape and assigning values to be used in the computation and performing the shape constraints as entered to the library). Previous research assigns the compiler and the optimizer to do the shape analysis as in [18]. For further details about the shape theory and semantics, refer to [18] and [19]. To summarize the importance of the shape separation from data, and the availability of shape information, we may state the following about it:

- It allows programming of graphs and topologies to be handled explicitly, while avoiding embedding within a structure, and supporting reusability for variant geometries.
- As will be discussed in the next section, it can also be more expressible in terms of processor architecture, so that the compilation can map the shape of the problem to the shape of the processors.
- Since shapes carry size information, this can be useful in pre-determining the load-balancing.
- It allows complexity estimates to be made for various sub-tasks, leading to improved scheduling, or determination to some non-deterministic algorithms.
- Because of shape polymorphism, error detection, and optimization are simplified.

- Error detection is useful in identifying the impossibilities and the improbabilities (wild constraints).
- The MOA design provides the shape analysis (no need for static nor dynamic shape analysis), which allows for compiler optimization, and consequently the computation to be straightforward.
- Global synchronization is made available due to the availability of shape information and reshaping algorithms.
- Reshaping to partition is already implemented, and using the Psi-Correspondence theorem (PCT), reshaping points are required, initially during the realization of the input data, and later during the data distribution, with as much redistributions as will be required.
- Shape independence is explicit in the MOA design, and can be addressed separately by the optimizer.

### *6.2.2 A Proposed Parallel Architecture Mapping Scheme*

The approach is based on two array organizations, one for the abstract model describing the problem, and the other for the enumeration of the processors in the form of a list, corresponding to the list of processors identity numbers, which is used to determine the actual send and receive instructions issued by the resulting program [6]. There must be a systematic determination method about what information to distribute to which processor, and to map from the data arrays of the problem, to the processors' array-like arrangements. The next step, is to design the low level code that selects data elements from one-dimensional memory, sending to the appropriate processor in the one dimensional list of processors. The code is to be based on the algorithm using high-level array-operations. In the second array (list of processors), each processor needs to have its own parameterized code to process its data in its local memory.

Slicing can be used to tackle the problem of large components. Slicing can be done by having a three dimensional array, where the first dimension represents the number of slices handled at one run, and the second dimension represents the amount of data each processor can process in one run, and the third dimension is the number of processors. There must be an automatic address computation formal technique to ensure that the problem decomposition is handled correctly, because of the difficulty in the manual manipulation of data addresses. [6]

Mapping Cartesian coordinates to their lexicographic ordering can be useful when we want to partition and map arrays to a multi-processor topology in a portable, scalable way. For instance, a vector A of shape (n), multiplied by matrix B of shape (n p), where p is the number of available processors, can be performed in parallel by mapping each of the rows in B to a processor, and the elements of A to the corresponding processor. An integer-vector multiplication takes place in each processor, and then the vectors are added point-wise, producing the result. Adding the n vectors together, is best done by adding pairs of vectors in parallel, or abstractly, adding the rows of a matrix point-wise. This computation ideally needs a hypercube topology, taking at best O(log n)on n processors to compute, which is often not available. Having a LAN of workstations, a linear list of processors, or any processor topology, we can view abstractly as a hypercube and map the rows to processors by ordering on the p available processors, looking at the $p_i$ where $0 \leq i < p$ as the lexicographically ordered items of the hypercube.

This process is based on obtaining a vector of socket addresses in case of a LAN, and abstractly restructuring it to a k-dimensional hypercube, where $k = [\log_2 n]$ [6]. Then the matrix needs to be restructured to a 3-dimensional array to be mapped to an abstract hypercube, having a 1-1 correspondence between the restructured array's planes and the available processors. This method utilizes the Psi Correspondence Theorem (PCT), sending the i$^{th}$ plane of the array to the i$^{th}$ processor lexicographically, and allowing for efficient memory addressing. In case there are more rows than processors, we can reduce the planes sequentially within each processor in parallel.

For example, an addition of the rows of 256 x 512 matrix denoted by A, on 8 workstations connected by a LAN, we would restructure the matrix into an array of shape <8 32 512> denoted by A. A matrix P for socket addresses is created with 8 addresses of workstations. Addition of the rows is performed in parallel producing 8 vectors of length 512 in each processor. Then, P is restructured into a 3-dimensional hypercube implicitly to be used in deciding how to perform accessing the subsequent addition between processors. Thus, addition can be done in 3 steps taking ($\log_2 8$) time: [6]

1. Add processor plane 1 to 0. By applying Psi-Correspondence Theorem, this means adding the contents of processors 4 to 7 to the contents of processors 0 to 3.
2. Add processor row 1 to row 0, (i.e. add the contents of processors 2 & 3, to the contents of processors 0 and 1).
3. Add the contents of processor 1 to the contents of processor 0.

The above method can be applied on any size matrix and any arbitrary number of homogenous workstations connected by a LAN. This scheme is portable and scalable.

### *6.2.3 Block Cyclic Redistribution with MOA*

Data distributions are managed to enhance the data locality in a distributed system, since accessing local data is much faster than accessing remote data [16]. Data aligning is a research area concerned with the redistribution algorithms, which are automatically embedded in compilers of parallel languages like in High Performance Fortran (HPF) as described in [20]. The block-cyclic distribution problem is a classical research area that matches the data access patterns of many High Performance Computing (HPC) applications, such as radar, and sonar signal processing. Also, there is ScaLAPACK, a mathematical software for dense linear Algebra computations, which uses block-cyclic distribution for load balancing and computation efficiency. The redistribution is required since data access may change during computation (due to failure of some resources in the distributed system, or whatever). This means data will be reorganized in order to minimize the remote access overhead, leading to scalable performance.

Specifying data distribution and redistribution varies in level of details in application programs. It is specified as a high level compiler directives, when parallel compilers are used, like in HPF: ALIGN, DISTRIBUTE, and REDISTRIBUTE directives. In explicit parallel algorithms, the programmer manages the data distribution and movements between the processors. Message Passing Interface (MPI) can be used to perform inter-processor communications. These algorithms need to be efficiently implemented or else overheads will offset the data locality performance benefits.

A Cartesian representation of processor assignment to each dimension in a multi-dimensional block-cyclic redistribution is discussed as a process topology to implement the redistribution of multi-dimensional arrays [16]. Changing the block size requires the reapplication of a 1-d (for dimension) redistribution algorithm along each dimension of the array. However, this is applicable for the process topology changes, which require the matrix transpose operation. This used to be enhanced by reducing the index computation overhead, or reducing the actual communication cost of redistribution, which could make the redistribution cost very high if not implemented efficiently. Communication schedules are classified to be 1) direct scheduling – array elements are sent directly to their destination, or 2) multiphase scheduling – arrays are moved in phases using direct scheduling within the phases, reducing the startup costs.

In [16], a uniform framework for block-cyclic redistribution is presented, utilizing the generalized circulant matrix, exploiting the regular characteristics of block-cyclic redistribution. This approach minimized the communication time and the index computation overhead, deriving direct, indirect and hybrid scheduling, and eliminating node contention. The algorithm exploits a two-dimensional table (called Destination Processor Table - dpt), which relates the global block indices with the destination processor depicting the actual local memory layout of the blocks. The number of the columns equals the number of processors in the architecture, and the number of rows equals the number of the communication events. This structure makes the determination of every block location a function of the block size along each dimension (shape), number of processors, and the global block index assignment index, reducing the redistribution (conceptually) to be a table conversion process – row (inter-processor communications) and column (local memory) transformations. The indirect scheduling algorithm in [16] aligns the diagonal entries vertically in logarithmic number of steps by cyclically shifting the rows of the dpt, thus reducing the number of communication steps. The scheduling then proceeds as mentioned earlier either direct, indirect, or hybrid, using a permutation.

As discussed above, the shape information (holding size information) available in the MOA design can be made available to the distribution algorithm to be used in the dynamic data distribution, which is optimized based on the shape values. The dynamic cycle of shape analysis, and optimization required for the redistribution of the data until the program ends, is reduced due to the availability of the shape information from the beginning as discussed in section 6.2.1. The index computation overhead can be further reduced by applying the indexing notions discussed in section 6.2.2, using a mapping scheme between the indices in the MOA's data elements, and the processors MOA structure. The destination processor table used in this algorithm and the row and column transformations discussed, can be all mapped to the MOA structure and the PCT theory, using the operations implemented in this thesis to provide the index computation invariant of dimension and shape, which leads to a full implementation of the circulant matrix form using the same notation of the MOA library, either as a higher construct based on the available operations, or by adding an extra operation to handle this concept. The full implementation of the algorithm based on the MOA notation is beyond the scope of this thesis.

### *6.2.4 Tiling*

Tiling is the process of applying geometric transformations in the iteration space in order to restructure the loop nest (execution of statements within the loop, or the loop iterations), in order to improve the performance, preserving a reasonable tradeoff between communication and computation, and allowing automatic parallelisation [10]. It is a way to make an orderly subdivision of space, using two or more shapes assembled somehow to cover the whole plane without overlapping. The transformation can be any of the following types, which will be discussed in relation to their correspondence to the MOA notation and will cause the same kind of effect on the final performance.

Fusion in Tiling is combining two adjacent loops, which is achieved by the Catenate function in MOA. Fission is dividing two loops into two separate loops, which can be achieved by several functions in MOA according to the type of separation (psi, take, drop, and slice). Interchange is that the order of the nested loops can be interchanged. This can be achieved in MOA by the transpose function, which will also take care of the legality and the dependency and will preserve the performance of the original code automatically. Strip Mining in tiling is partitioning the one-dimensional array into vectors of certain size and operates on it as if it is a two dimensional array with two nested loops. This can be achieved in MOA, by the slicing function, returning partitions of the array in the form of slices (number of slices is the required size) of the required dimension – in this case vectors (arrays of one dimension). Then, the resulting vectors are reshaped to operate on them as two-dimensional arrays.

Loop Skewing is the process of transforming the inner loop indices to the sum of difference of old inner indices and an integer multiple of the outer loop index [10]. This adding of a skewing factor to the indices of the iteration is what is totally eliminated in the MOA notation, since there is no need to have a relation between indices identified with this skewing factor, when you have the psi function. In MOA, all that the programmers need to do is to send the indices in a vector with the multidimensional array as arguments to the psi function, and that defines the relation. Another possibility is to send the vector of indices to the Gamma function and it returns the index in the flat array, without going through the trouble of identifying the skewing factor. Hence, in relating arrays of different dimension or shapes to each other in one loop, indices are used incrementally according to the shape and bounds of dimensions of each array.

Rectangular partitioning in Tiling is the process of forcing the tile to be of rectangular shapes having the borders of the tiles parallel to the iteration space boundaries, not to the iteration space axes. The reshaping in general, whether to rectangular or any other shape, is achieved with a simple function in MOA, which is reshape.

The data dependence relations can be identified straightforwardly in MOA notation from the multidimensional index vector. The Hyper-planes defined in tiling is the main data type of the MOA notations, since it is all based on a variant number of dimensions and shapes. The Tiling parameters (size and shape) are the same notation used in MOA with Tau (Shape) and shape, where the psi, reshape, take, drop, and slice functions can be used to perform the tiling algorithm in the MOA notation, keeping the number of computation nodes (elements of components of the flat array) bounded by the cache size.

MOA allows for better parallelisation because of the lowered dependency in its logic. In Tiling the main factors were the communication and computation, while in MOA, we can focus only on communication, since computation can be easily divided into independent blocks of code that address specified locations in memory.

## 6.3 Arrays Pipelining

Pipelining is one of the parallel processing techniques. Since arrays are accessed by loops, therefore pipelining can be a process of scheduling technique that makes use of the repetitive nature of loops. This means, an iteration of a loop starts before its preceding iteration is completed, thus executing multiple iterations concurrently. This makes it an NP-complete problem, which is overcome by using expensive hardware features such as in the polycyclic architecture or the FPS 164 approach that restricted software pipelining to loops containing a single Fortran statement in their bodies. Previous work addressing this issue includes Warp systolic array architecture, the fine-grain scheduling technique, and the time and space efficient general framework for fine-grain code scheduling in pipelined machines, exploiting fine-grain parallelism through dataflow software pipelining as discussed in [12].

The software pipelining accomplished by arranging the mapping of the array selection operations to flow in a pipelined fashion into a code block (units of program text that define the major structured values involved in a computation), in the right order, while discarding the unused values. This structure is achieved by having an index generator subgraph (IGEN) that generates the indices required from an array in a computation. Besides, an array generator subgraph (AGEN) is needed to use the indices and control values generated from the IGEN to structure an array (assemble the results elements into a resulting array) depending on the base address of the array. Also, the framework requires a selection operation with an index calculation expression. The use of the IGEN is optimized by deciding whether the array elements selected are processed in the same order, in which case, there is no need to dismantle them and assemble them after the computation, and thus create an overhead of storing intermediate array values in structure memory. [12]

The scheduling for pipelined architecture is addressed by a constraint precedence graph, where nodes represent operations, directed edges represent precedence relations due to data dependence hazard, and undirected edges represent constraints due to possible collisions. This scheduling aims to find an execution order of the instructions to minimize the total run-time. The code scheduling for the dataflow software pipelining is the process of defining a dataflow of a computation, which shows the successive waves of elements of the input arrays to visualize how the computation can proceed in a pipelined fashion. Work in this field resulted in a (fully) balanced acyclic dataflow graph containing equal number of operations on every path between a pair of nodes. This balancing can be achieved by introducing a minimum amount of FIFO buffering on certain unbalanced arcs. Full balancing vs. limited balancing concepts were discussed in this work, causing less space requirements in the case of limited balancing. Also, the register allocation task is not done separately from the loop-scheduling task, with a total transparency of the runtime storage management to the user (complier), and reduced synchronization acknowledgement signals. So, the scheme is based on unraveling the loop at compile-time for pipelined execution, using the same code and data memory space for the entire loop pipelining, avoiding the overhead of allocation and management of space and tags. [12]

From the abstract description of the monolithic array pipelining, we can visualize where the MOA notation fit in the scheme. IGEN, AGEN, and index calculation expression are the three main constructs in this scheme that can be easily implemented in the MOA notation. The index generation in the IGEN can be implemented using the Gamma and Gamma Inverse functions in MOA. The array generation is the result of many of the MOA partitioning and restructuring functions discussed in MOA in previous chapters (take, drop, reshape, catenate, reduce, slice, … etc.). The index calculation construct is the heart of the MOA notation in all its functions.

## 6.4 MOA Hardware Implementation

### *6.4.1 VLSI Verification*

The MOA axioms and definitions represent one of the techniques used in the VLSI verification. Since arrays are considered an important data structure for the description of any physical problem, the array operations and notations are used to describe and simulate computer architectures, and as a basis for register transfer languages. The entire System/360 Architecture is described using APL notation. The APL notation is used to describe hardware components like multipliers and adders. Also, there is a register transfer language called AHPL, written in Fortran, which simulates hardware design. Moreover, a circuit design language, APLSIM, provides SPICE-like simulations in a modified APL environment written in C. The properties of the mathematics of arrays notation discussed previously, are considered useful to the VLSI design verification.

As defined previously, the mathematics of arrays is based on believing that all data structures can be represented using arrays. A scalar is an array of zero dimension, a vector is an array of one dimension, and a matrix is an array of two dimension, and so forth. Forming a notation to handle all operations independent of dimension, resulted in having a new paradigm of computation, which takes the advantages of the mathematics of arrays, and is applicable to all domains, specially the image processing applications. This notation was first implemented in software using the Psi compiler, and achieved a speedup on the performance of the conventional software computation on both uni and distributed processors, by reducing array expressions to eliminate unnecessary computation and temporary storage. It was also desirable to apply this notation on hardware to achieve higher performance. In [2] and [3], a Chameleon reconfigurable coprocessor board is presented to try various approaches. The board consists of:

1. One 4013-4 Xilinx FPGA for SBUS interface,
2. 10 Mbits of 15 ns static RAM (SRAM),

3. An Analog device ADSP-21020 Digital Signal Processor (DSP) for high speed floating point calculations.

4. Two application reconfigurable FPGAs, programmed through the SBUS to control the DSP chip and to manipulate the data between the SBUS, external SRAMs, and the DSP chip,

The system implementation is done manually in VHDL, aiming to use the Psi compiler to generate syntheizable VHDL code for high performance array address generation and microcode for the DSP chip. In [2] and [3], the authors presented an application example of array convolution using MOA to derive a hardware algorithm for optimal memory address generation on the Chameleon coprocessor, thus achieving optimal performance for all resources involved (bus bandwidth, DSP speed, supporting architecture, array shapes, etc.). The array convolution is achieved using the following algorithm:

1. Define the initial data array D, and the multiplier mask M, and flatten them using the rav operator.

2. Iterate step 3 until the difference between iterations for all points is less than some predefined epsilon or until a predetermined iteration count is reached, depending on the application on hand.

3. Apply the mask to all sets of points in the data array to compute the resultant data array with boundary conditions held constant. A single resultant point in the resultant data array is computed by taking each element of the mask and multiplying with each element of a corresponding region in the data array centered about the point of interest and summing the results to compute the resultant point. This is achieved by applying the summation reduction (+ red) on all vectors i greater than or equal zero, and less than or

equal the shape of the mask, as the Cartesian offset of the mask (i $\psi$ M), multiplied by the vector (sub-array) b taken from the vector i, and dropped from the initial data array D.

Using the MOA notation, the above algorithm can be defined by the following expression [3]:

$$\vec{i} \;\; \psi \;\; \xi_r \;\; = \;\; {}_{+}red \;\; \left[ \xi_m \;\; * \;\; \left[ \rho\xi_m \;\; \Delta \;\; \left( i \;\; - \;\; \frac{\rho\xi_{m-1}}{2} \right) \;\; \nabla \;\; \xi_i \right] \right]$$

**Equation 6-1**

with $\frac{\rho\xi_{m-1}}{2} \;\; \leq \;\; * \vec{i} \;\; \leq \;\; * \rho\xi_m \;\; - \;\; \frac{(\rho\xi_{m-1})}{2}$

### *6.4.2 Hardware Implementation of the MOA Library in VHDL*

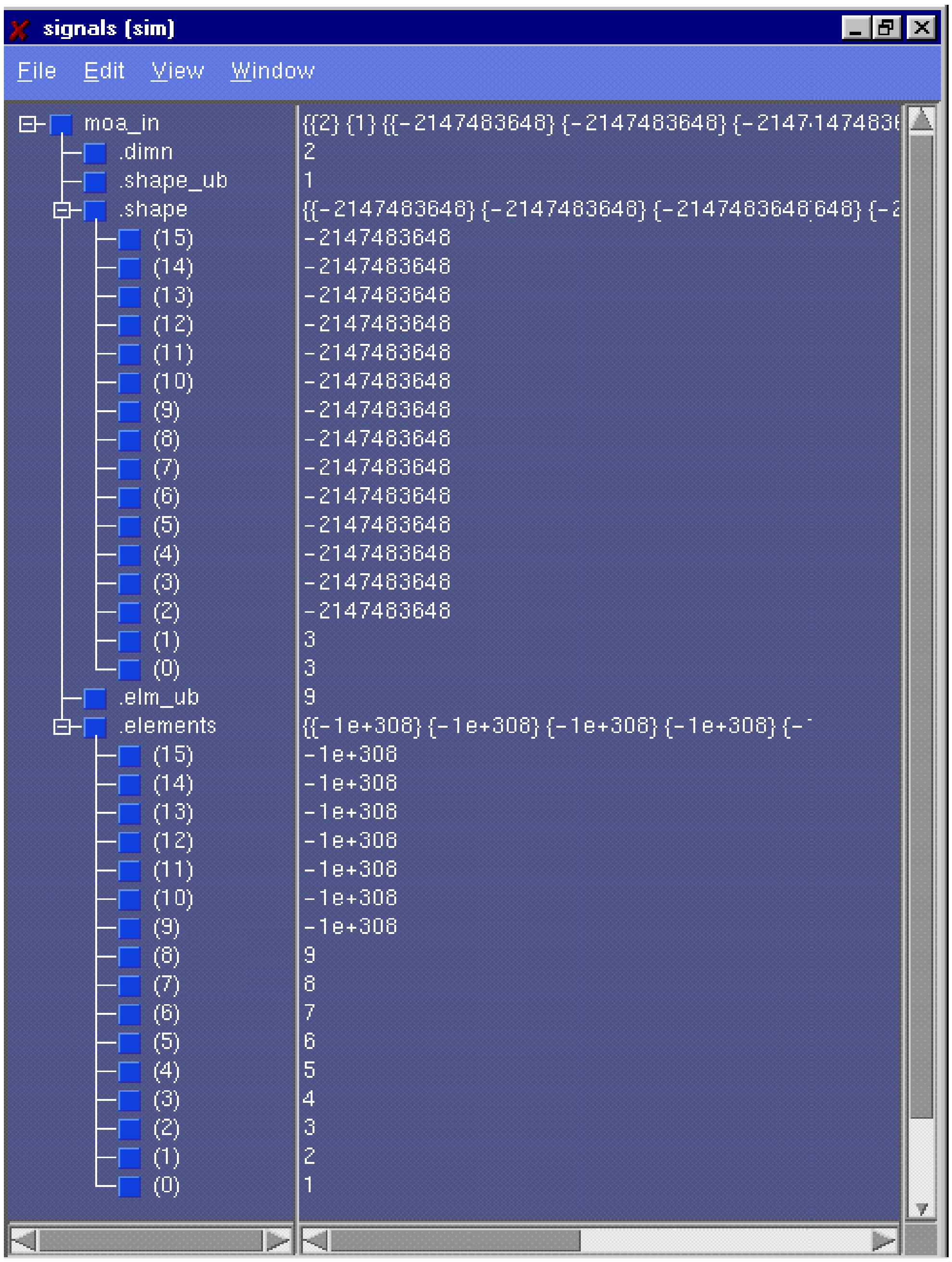
It was also desirable to test the MOA notation implementation in VHDL as a hardware description language, in order to test the performance of the notation in hardware. The implementation was designed in the form of a library package to be included in any project to use the pre-compiled operations. The only limitation in the hardware implementation was the absence of the dynamic allocation of memory as it was available in C++ implementation, or the absence of the unconstrained arrays the way it is used in other software programming languages. There must be a constant upper bound for the memory reserved for the MOA structure, and the application uses as much variable space needed beneath this upper bound. For example the MOA structure displayed in figure 6-1, displays a VHDL MOA defined in the input signals simulated with ModelSim. A static upper bound is defined for this package to be 15, and only one upper bound is used for both the shape array and the elements array (this is of course for simplicity, but engineering wise, it can't be). The package traverses the arrays up to the defined upper bound and skips the rest.

The static allocation of array sizes limited the main advantage of MOA, which is the efficient memory manipulation. Solutions like cascading or designing different size packages can be applied to allow for applications requiring larger memory allocations to set an upper bound.

The same design applied in the C++ implementation for the MOA, was again used in VHDL, minor changes were required to take care of the hardware processing. The simulation showed the same results as the C++, and the synthesis required further changes to avoid the use of the for loops. The implementation on Renoir using VHDL, and producing a package, was intended to test the performance of the MOA notation on hardware and study the possibilities of having special-purpose hardware (hardware accelerators) for scientific applications. Appendix D contains the VHDL package header. The difficulties lie in the synthesis of the package, since some of the functions used in the design are not synthesizable. So, they need to be rewritten in a more primitive syntax to allow for synthesis.

In the case of figure 6-1, the shape is traversed up to index 1, since it is a 2-dimensional array, and the elements array is traversed up to the index 8, since the shape vector is defined to be < 3 3 > as seen in the shape (0) and shape (1) signals. The MOA array in figure 6-1 denotes an array as follows:

$$\delta\xi \equiv 2,\ \rho\xi^2 \equiv \langle 3 \quad 3 \rangle, \text{ and } \xi^2 \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

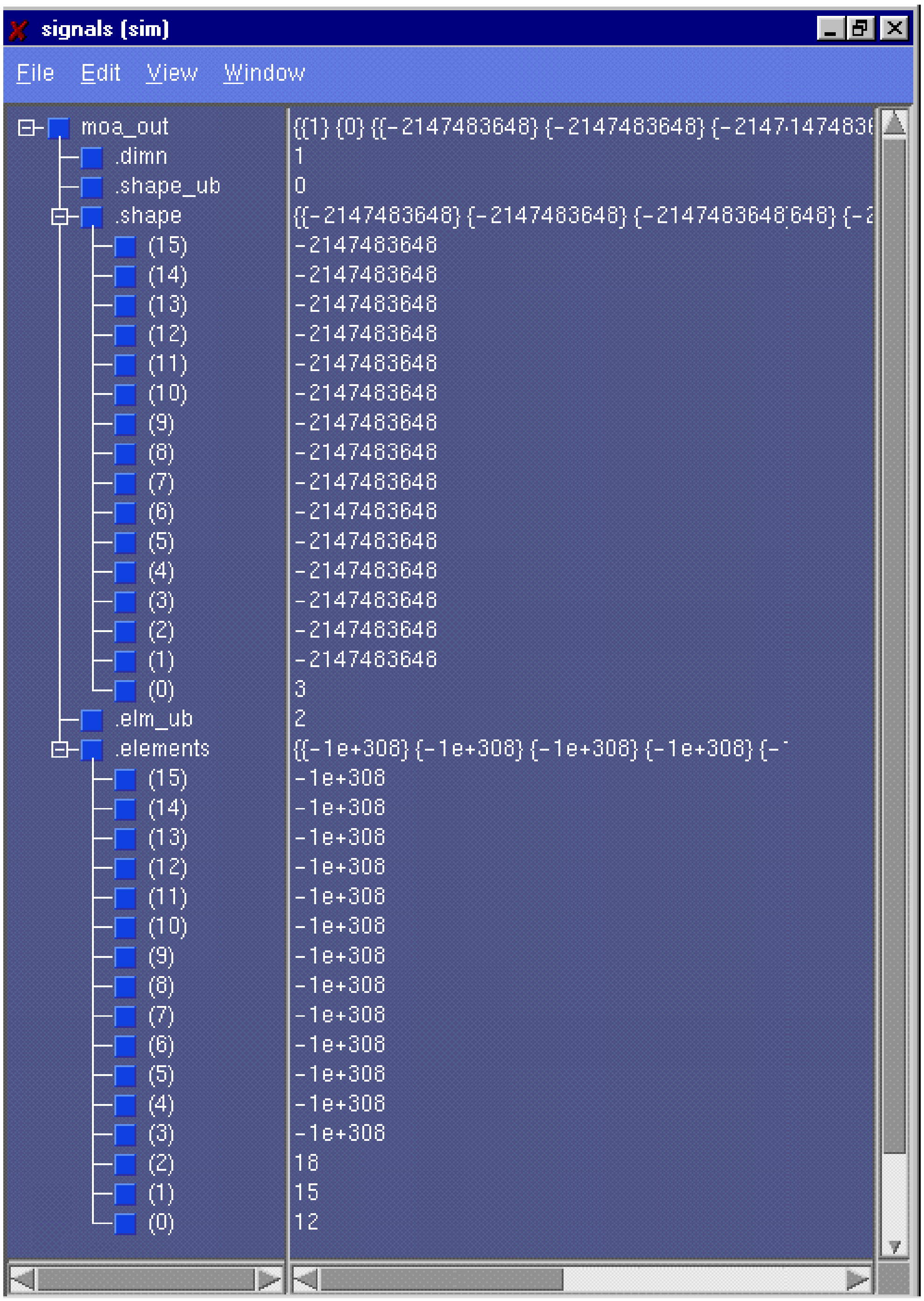


```
moa_in          {{2} {1} {{-2147483648} {-2147483648} {-2147.1474836
  .dimn         2
  .shape_ub     1
  .shape        {{-2147483648} {-2147483648} {-2147483648.648} {-2
    (15)        -2147483648
    (14)        -2147483648
    (13)        -2147483648
    (12)        -2147483648
    (11)        -2147483648
    (10)        -2147483648
    (9)         -2147483648
    (8)         -2147483648
    (7)         -2147483648
    (6)         -2147483648
    (5)         -2147483648
    (4)         -2147483648
    (3)         -2147483648
    (2)         -2147483648
    (1)         3
    (0)         3
  .elm_ub       9
  .elements     {{-1e+308} {-1e+308} {-1e+308} {-1e+308} {-
    (15)        -1e+308
    (14)        -1e+308
    (13)        -1e+308
    (12)        -1e+308
    (11)        -1e+308
    (10)        -1e+308
    (9)         -1e+308
    (8)         9
    (7)         8
    (6)         7
    (5)         6
    (4)         5
    (3)         4
    (2)         3
    (1)         2
    (0)         1
```

Figure 6-1: MOA Array with defined upper bound in VHDL

The testing in the simulation, showed accurate results as shown in figure 6-2 demonstrating the result of a reduction operation with + operator, and on dimension 1, over the MOA input array displayed in figure 6-1. The MOA array in figure 6-2 denotes the following array:

$\delta\xi \equiv 1$, $\rho\xi^2 \equiv \langle 3 \rangle$, and $\xi^2 \equiv [12 \quad 15 \quad 18]$



```
moa_out        {{1} {0} {{-2147483648} {-2147483648} {-2147·1474836
  .dimn        1
  .shape_ub    0
  .shape       {{-2147483648} {-2147483648} {-2147483648 648} {-2
    (15)       -2147483648
    (14)       -2147483648
    (13)       -2147483648
    (12)       -2147483648
    (11)       -2147483648
    (10)       -2147483648
    (9)        -2147483648
    (8)        -2147483648
    (7)        -2147483648
    (6)        -2147483648
    (5)        -2147483648
    (4)        -2147483648
    (3)        -2147483648
    (2)        -2147483648
    (1)        -2147483648
    (0)        3
  .elm_ub      2
  .elements    {{-1e+308} {-1e+308} {-1e+308} {-1e+308} {-
    (15)       -1e+308
    (14)       -1e+308
    (13)       -1e+308
    (12)       -1e+308
    (11)       -1e+308
    (10)       -1e+308
    (9)        -1e+308
    (8)        -1e+308
    (7)        -1e+308
    (6)        -1e+308
    (5)        -1e+308
    (4)        -1e+308
    (3)        -1e+308
    (2)        18
    (1)        15
    (0)        12
```

Figure 6-2: ModelSim MOA output after reduction operation using + operator on dimension 1

## 6.5 Summary

The experimentation in the implemented library can take different directions, which is what this chapter has proven. The work in this thesis focused primarily on the implementation of the basic notation, as well as validating the notation behavior on 2-dimension and 3-dimension, using an image and video processing applications respectively, proving the correctness of the design. However, the main advantages of the MOA class can be clearly observed when experimenting with parallel computing, pipelining, and even when augmenting hardware accelerators. The parallel computing was discussed, presenting the shape polymorphism of the implemented MOA structure, the MOA proposed parallel architecture scheme, MOA data redistribution using the block cyclic redistribution algorithm, and the tiling MOA implementation. The MOA array pipelining was discussed based on a general framework for fine-grain code scheduling in pipelined machines. Only the hardware experimentation in this chapter was implemented on VHDL using Renoir and simulated using ModelSim.

# *Chapter 7*

# PERFORMANCE ANALYSIS

## 7.1 Introduction

In this chapter, a discussion of the performance of the applications, designed based on the MOA library implemented in this thesis is presented. It is difficult to give a quantified analysis for all applications, since the library has a wide variety of application domains, and is still open for further investigation, more fine tuning, and for adding more exceptions to be handled as required by applications. Hence, the analysis presented in this chapter will be a general estimate on how far an application based on the data structures of the MOA library, can be enhanced due to the nature of the design, and how far it is dependant on the complexity of the applications' computation.

## 7.2 MOA Versus Traditional Methods

From the experiments presented in the previous chapters, it is obvious that the declaration of the MOA arrays requires more statement and initialization of the MOA structure because of the dimension and shape invariance and separation. This complication can be reduced if handled by the compiler (in case of augmenting the semantics of the library to the compiler), where extra declarations are added by the compiler, not by the programmer. The division of work between the compiler and the programmer is out of the scope of this thesis. However the array declaration is not the end of the story. All the complications of the traditional methods come from traversing and manipulating a multidimensional array specially when there are some defined relationships among its elements. The equations that define these relationships require lots of validations and verifications until they are efficiently implemented. The implementation of these equations might add extra complexity and overhead to the performance of the design of the solution. Therefore, the programmer will start reducing and simplifying these equations to reach the optimal form. These steps are all eliminated when designing the data structure of the solution on hand based on the MOA library implemented in this thesis.

The MOA library contains numerous types of relations that can be extended in any further experimentation on new application domains, as well as traversing methods that are already simplified and reduced using the Psi-Reduction theorem and the Psi-Calculus. Dependence on these operations will eliminate the effort of redesigning them every time an application requires their implementation for a fixed dimension and shape, since they represent a repeated pattern between different fields of application. Moreover, MOA decreases the number of nested loops required in the solution, the fact that will eliminate the erroneous design of loop starts, stops, and strides and boundary constraints, leaving the source code of the application on hand reflecting the design of the application specific solution. The shape polymorphism achieved in the MOA library allows all operations to reduce the general case to that of lists, so that these operations can be interpreted by a single algorithm for every function, i.e. only one transpose algorithm is used for all shapes, one dot operator, one reduction operators, … etc.

## 7.3 Nested Loops Reduction with MOA

The MOA notation reduces the number of nested loops involved in traversing arrays of variant dimensions. All types of arrays (different dimensions and different shapes) can be traversed in one loop either on the sequence of their raveled (flattened) array, or on the sequence of their shapes and dimensions. This reduction is useful for the performance, and efficiency of the solution design.

**An example of an array traversed on its variable dimension and shape**

```
Array[A1][A2][A3];
L1: For (I = 0 to A1 )
L2: For (J = 0 to A2 )
L3: For (K = 0 to A3 )
Array[I][J][K] := I * J + (K*3)
End L3
End L2
End L1
```

**An example of an array traversed on its flattened array**

```
Array[A1][A2][A3];
L1: For (i = 0 to (A1 * A2 * A3) )
Indices = Gamma_Inverse (i, Shape(A));
Array[Indices[0]] [ Indices[1]] [ Indices[2]] := Indices[0] * Indices[1] +
(Indices[2]*3)
End L1
```

Where Shape is a function that returns a vector containing the A1, A2, A3 bounds of each dimension or axis in the multidimensional array, and

Gamma_Inverse is a function that returns the values of the multidimensional indices for a specific index in the flattened array.

Note also that the reference:

```
Array[Indices[0]] [ Indices[1]] [ Indices[2]]
```

is equivalent to:

Array[i], where i is the loop counter. Also it is equivalent to:

Array[Gamma(Indices)], where this will return the same i mentioned above.

The second example contains one loop, and the gamma inverse function contains just another one loop on the array, so, we end up with one loop reduction from the three loops used in the first example. That is in case of three-dimensional arrays. For more dimensions, the second method will keep on using those two loops only for an arbitrary number of dimensions, while the traditional method shown in the first example will use as many loops as the dimensions of the array. The graph presented in figure 7-1 shows how dimension invariance programming affects performance as dimension increases.

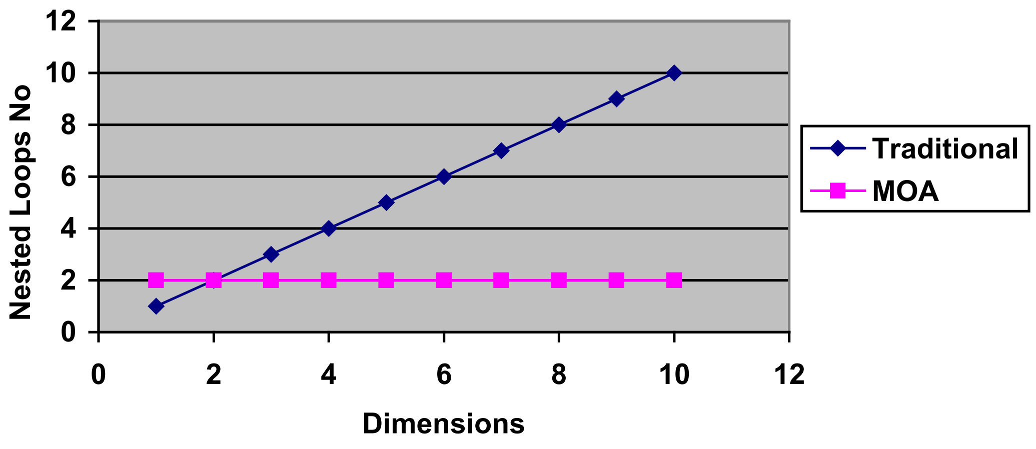


Figure 7-1: Increasing Dimension in MOA vs. Traditional Methods

## 7.4 Invariant Dimension & Shape Analysis

From the above analysis, we arrive at the fact that the declaration of the MOA arrays are complicated than in the traditional multidimensional array handling. We can also see how easy it was to design the video MOA class. This easiness comes from the fact that most of the functionality of video processing can be readily implemented in the image MOA class. Hence, processing the video operations can be just a matter of calling the MOA image class to apply some image operations on the several image frames found in the video files. This property in general visualizes the hierarchical architecture of the MOA arrays, which provides reusability of functions implemented on one level to be applied on higher or lower levels in the hierarchy by manipulating the MOA structure correctly. Therefore, the extra complexity in the MOA array declaration more than the traditional methods noticed earlier, is compensated for and even better enhanced by the increasing dimension, and the complicated traditional manipulation of multidimensional arrays.

This basic property of MOA is the main justification of using it. If an application expects its data structure's shape and size to be defined, grows, or shrinks at run time, or when operation may need to be re-applied on different levels, slices, or partitions of its array structures, it will considerably benefit from designing its structures based on the MOA library. On the other hand, applications that decides for sure that there is no expansion in the dimensionality of their data structures, can still benefit from the parallelisation inherited from the shape polymorphism in the design of the library.

## 7.5 Parallel Processing

Parallel processing with MOA was investigated in the previous chapter. It was proven that MOA inherently contains parallel computing factors that lie in the shape invariance and separation from data, and in the partitioning and decomposition of arrays into smaller ones, which can be processed concurrently on several processors or threads, then reconstructed to form the larger array result. This main concept can be expanded in several ways to include more control over the parallelisation process, and solving many of its problems in the scheduling, data redistribution, and load balancing problems. The shape analysis and its effect on the parallelisation, and the data redistribution problem, and the tiling algorithm implementation based on the MOA library are examples for how MOA can be a base for an automatic parallelisation agent that parses a sequentially written source code to generate its parallel version. Further investigation and experimentation is required in this field, since only discussions were presented in this thesis.

## 7.6 Time of Processing

The time of processing is expected to be exponentially decreasing as the dimension increases on one hand as illustrated in the graph of figure 7-1, and is expected to decrease again when parallelized and executed on a multi-processor environment in order of the number of processors involved. The shape knowledge decides the computation time, by deciding the depth of the trees or graphs connectivity denoting the complexity of the operation. Hence, a general estimate of the time of processing of any computation based on the MOA library is:

$$O(2 * \frac{\tau(\rho\xi)}{N})$$

Where $\xi$ is the data array of invariant dimension and shape (could be scalar, vector, matrix, arrays of any dimension), and N is the number of processors used in the computation, $\tau$ is the Tau operation that returns the product of the shape vector $\rho\xi$ denoting the number of elements in the array $\xi$. The size of the data array is multiplied by two, since it was decided previously that it takes two nested loops to traverse an array of arbitrary dimension and shape using the MOA paradigm, and then divided by the number of processors involved in the computation. Obviously this equation is a general estimate of the simple traversing problem of arrays in MOA. The computation specification of an application based on MOA, rewrites this equation depending on its requirements and solution design.

The image transformations implemented in this thesis in the image and video processing tools take more time than the traditional methods. This is due to the full traversal of the image in pixels in the image class, and to all pixels in all frames in the video class, all on one processor. Furthermore, another reason for the slower performance is the overhead of mapping the bmp file format to the MOA structure to do the operation, then mapping back to the bmp stream of pixels to draw on the screen the result using the StretchDIBits function. The performance can be considerably enhanced if a file format of the MOA image structure is designed and a new drawing function is implemented to read from the MOA description directly.

## 7.7 Hardware Accelerators

The implementation of the MOA library as a VHDL package to test its performance on hardware succeeded to produce correct results. However the array definition in VHDL requires a constant upper bound. So this constraint affected the complete dynamic implementation found in the software version. However, it can be compensated for by designing packages of defined upper bounds, and the application uses a suitable bound on its data size.

## 7.8 Summary

The performance analysis of the classes implemented in this thesis is difficult to quantify, since it will always depend on the application employing it, and the correct handling of the MOA structure and operation. The work on this thesis is rather abstract and proposes a new paradigm of computing that is still open for further enhancements and experimentations that will tune the performance. Generally, this chapter discussed the factors affecting the performance of applications designed using the MOA structure. These factors are: the expressiveness, validity and conceptual clarity achieved better than in the traditional methods, the nested loops reduction for multi-dimensional arrays, the programming invariant of dimension and shape polymorphism benefits, which is more amenable to parallel computing, and finally work can be more easily divided between software and hardware accelerators.

# *Chapter 8*

# SUMMARY

## 8.1 Introduction

This thesis investigated the array programming invariant of dimension and shape, using the Mathematics of Arrays (MOA) - PSI-Calculus. The main emphasis through out the experimentations conducted was to form a complete understanding of how far we can utilize the MOA notation and the PSI-Calculus, which were previously defined in the literature but never applied in real applications. The gap between the mathematical equations defined earlier and the complexity of programming these equations without being analyzed and simplified from a programming point of view was the main reason behind ignoring this complete notation. This thesis took a different direction than the previous research efforts, by working on implementing the array operation invariant of dimension and shape, as APIs, not as a compiler that parses a new language lexicon of the basic notation to produce a traditional array handling source code in the targeted language, nor as a language extension to an existing functional language.

This thesis proved that the MOA notation implementation neither requires a separate complier to compile its notation to a functional language, nor does it requires extensions to the existing functional languages. It was implemented as a library of APIs on an object oriented class using C++, to be used by programmers as higher-level constructs, operating on a flat array (one dimensional array), comprehending it as an array of dimension and shape defined at run time. The choice of C++ as a powerful object oriented imperative language was meant to add some functional array interactions to C++ as in array-based functional languages that are based on a mathematical description of the problem. The CMOA class implemented in this thesis satisfies the preliminary requirements towards achieving a functional C++ array interaction, which were implemented in sort of higher level constructs (APIs) that can be used for scientific and engineering applications in an object oriented environment that allows for overloading, inheritance and polymorphism. These concepts helped in achieving different variations of the array operations, and can allow for nesting of the data structures containing the arrays.

Also, C++ implementation achieves more effectiveness and better memory control. The design of the library allows compilers to capture the implicit parallelism due to shape polymorphism. This design of the MOA proved its correctness and showed better programming style, and generally better performance. It eliminated a considerable amount of nested loops, and allowed for reusing the functions defined for any dimension and any shape.

The Library implemented in this thesis is comprehensive and covers the entire notation presented in previous work with a few simplifications to some of the equations, and adding some more functions that were required for the application of the library, like the slicing, iteration, connectivity, and orientation functions. The application of the MOA library design could be infinite, since it forms a programming paradigm that even scalar programming can be based on. First of all, as claimed in previous work, it can be an extension to compilers, where users don't have to change their programming styles, but the compiler parses their defined structures as MOA structures, with scalars treated as arrays of zero dimension, vectors as arrays of one dimension, and matrices as arrays of 2 dimensions, and so forth. This eliminates the shape analysis phase from the compilers' job, as required for parallel computing. It also makes the translation to low-level languages and Register Transfer Languages (RTL) as well as the parallelisation inherited from the shape polymorphism, much easier and more straightforward.

The second class of application, which was more experimented in this research, is the graphic design (image and video processing). MOA showed better performance and easier manipulation of graphics (2D and 3D images) and video files. Most of the processing of both images and videos are a matter of calling notation already implemented in MOA library, with higher efficiency, enhanced performance, and much less effort than traditional methods.

Third class of application represents the scientific and engineering applications in general, which require excessive computations, and complicated data structures with defined relations among them. The shape polymorphism is useful for divide and conquer algorithms, Gaussian elimination on a sparse matrix (pivot choosing), Partial Differential Equations (PDEs) which is graph polymorphic (only depends on the notion of neighborhood), systolic algorithms, and recursion in trees construction and parsing (defining the path to a node in terms of an index in a list).

The fourth class of application of the MOA library is the parallel computing and pipelining experiments discussed in chapter six. Some of the applications of this class are:

1. Parallel matrix multiplication.
2. Mapping arrays by Cartesian Coordinates to a Multi-processor topology.
3. Matrix addition on workstations connected by a LAN.
4. MOA block cyclic data redistribution.
5. Tiling Algorithm MOA implementation.
6. MOA arrays pipelining scheduling algorithms

This class of application also includes the operating systems, and distributed systems in particular. The MOA library can be used for the scheduling of processes on processors, load balancing, mesh networks communications or other types of networks merging and designing, … etc.

Designing the data structures of the various applications based on the MOA library, gives the implementation efficiency, conceptual clarity, and expressiveness, due to the elimination of a considerable amount of equation verifications and validations, allowing for better performance. Besides, all the above discussed features of dimension and shape invariance, reduction of nested loops as dimension increases, pre-compiled and efficient index computations, parallelisation, pipelining, and ability to augment hardware accelerators to the design are added values to the design.

## 8.2 Experimentations Results - Contributions

The main contribution of this thesis is the full implementation of the MOA notation in C++ as APIs, which is done for the first time in a powerful Object Oriented imperative language. This implementation required a comprehensive analysis of the MOA operations equations and their simplification. The CMOA class is the base for other experimentation in this thesis. This contribution is verified through the implementation of the MOA testing tool, which presents a comprehensive user interface for all notations that allows the user to define the test cases and call the MOA methods. It graphically displays the results on the screen in a style amenable for further operations. Test cases applied covered a variety of MOA arrays of different dimensions and shapes, and applied all the methods on each example. The library showed a steady performance invariant of dimension and shape.

A second contribution is the two other testing stages, which were performed to prove the applicability of the MOA library in two real life applications, namely the image and the video applications. The image-processing class implemented using the MOA basic class, proved the correctness of the design, and presented an easier and neater manipulation of the image, and the operations that were processed on it. The MOA image class processed image transformations that were programmed in an expressive and concise source code. The MOA based class for Video processing was implemented to read video files, store them in MOA structures, and operate on them using the MOA notation and the implemented MOA based image class. This class showed easier manipulation of video files and more control over the frames in the video stream processing operations. Most of the functionalities of video processing are already implemented in the image MOA class, and are just called to be applied on the several image frames found in the video files. This property visualizes the hierarchical architecture of the MOA arrays that provide reusability of functions implemented on one level to be applied on higher or lower levels in the hierarchy by manipulating the MOA structure correctly. Thus, the extra complexity in the array definition over the traditional methods, is soon compensated for and even enhanced when the dimension and the shape changes at run time, and the designed algorithm still performs correctly, and the complicated traditional manipulation of multidimensional arrays.

A third contribution is the definition of the parallelisation and pipelining factors inherently defined in the MOA notation. These were discussed in terms of shape polymorphism, MOA parallel architecture scheme, data redistribution algorithms, the tiling algorithm related to the MOA class, and the MOA array pipelining. These discussions shed some light on the benefits of employing the MOA structure in problem solutions, related to parallel computing.

A fourth contribution is the hardware implementation of the library using VHDL on Renoir. The package implemented in this thesis can be considered a base for more specific hardware accelerators of data-intensive computations, specifically in the DSP, image and video processing fields.

## 8.3 Future Work

Future work based on this thesis can be classified into two categories. The first category is the possible enhancements of the current implementation of the MOA library to include more array interactions or concepts, or even to enhance its current performance. This category includes adding more direct subscripting operators to the library that allow for accessing the scalar elements, partitions or the whole array (by ignoring the index) for reading and writing. Also, graphical representation of data elements in the MOA tool can be enhanced to match the ones produced by Nial.

Moreover, investigation can proceed in the direction of the implementation of irregular array structures also invariant of dimension and shape. In this thesis only the regular (rectangular) arrays have been implemented for the MOA notation. This research can focus on allowing higher dimensions in the shape vector not to be multiplied by the lower dimensions in the shape vector to contribute to the final number of the elements in the MOA array; i.e. upper bounds defined in the shape vector could be defined for each dimension, where each can define its own lower-level number of dimensions and the extent of each (shape). It will also require shape records for every dimension in the MOA array. Therefore, the final number of elements on the flat array, could be the recursive summation of all these shape values (with a termination condition reached when the dimension of the current nesting is equal to 1 – vector - or zero - scalar -). The theory is clear for implementation, however it will require revising and altering all implemented functions to reflect this change in the data structure, to maintain invariance, and certainly will need the rewriting of all equations for higher-level generalization. This can be programmed such that:

```
struct shape_rec {
dimn;
shape;
}
struct MOA_rec {
dimn;
array_of_shape_rec;
data_elements_flat_array;
}
```

where the shape_rec in the MOA_rec structure is a structure as shown above. This structure contains dimn, and shape vector. The shape_rec is a pointer to an array whose size is defined at run time by the MOA dimensions. Thus, we can have a shape record containing another dimension and shape vector for each dimension in the main MOA structure. The size of the flat data elements array, is decided by the summation of the product of all these shape vectors as mentioned above, while maintaining the data listed in row major order in a flat array.

This enhancement will update the tree diagram of the MOA structure in chapter 4, figure 4-3, to be as follows:

Given an MOA array of base dimension = 3;

Dimension 1 ➔ dimn = 1; shape = 4;

Dimension 2 ➔ dimn = 3; dimension 1➔ dimn = 1; shape = 2;

dimension 2➔ dimn = 1; shape = 3;

dimension 3➔ dimn = 1; shape = 4;

Dimension 3 ➔ dimn = 2; dimension 1➔ dimn = 1; shape = 2;

dimension 2➔ dimn = 1; shape = 2;

Figure 8-1 represents this change, where the total number of elements (leaves in the tree) in the data elements flat array is equal to 17, instead of 12 in figure 4-3.

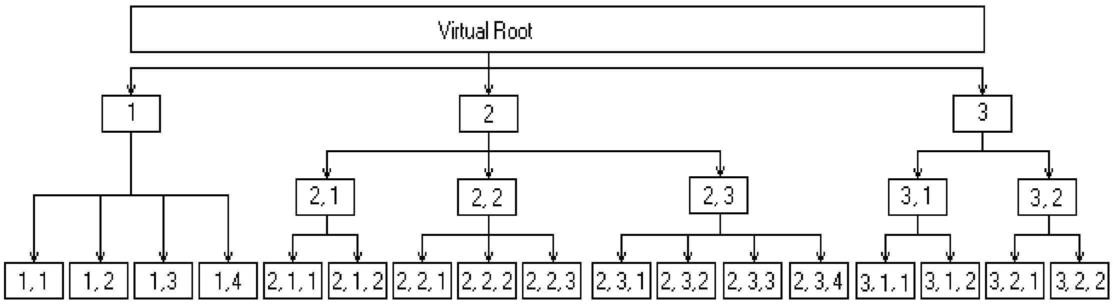


Figure 8-1: Irregular Array Indexing Tree Structure

Hence, if it is possible to do irregular shapes, why don't we also include nesting in the MOA programming paradigm. This means that data elements could have any type, and these types can include also, other MOA structures. This opens the door for testing for data polymorphism, where the computation is done invariant of the data type or size. Also, future work can test having the data elements flat array be of different types within the same MOA structure.

Also, implementing the parallel and pipelining experiments discussed in this thesis is another suggested future work. The parallelisation factors discussed in relation to the MOA library can create a new direction in the research for designing an automatic parallelisation agent, which takes a sequentially implemented source code, and automatically produces a parallel implementation of it. This new direction can be based on the library implemented in this thesis, after enhancing it, or adding missing utilities.

The second category of the future work is based on deploying the library implemented in this thesis to different problem domains. This category includes further investigation of the image and video-processing applications, starting by defining new file formats for the image and the video files based on the MOA structure. It also includes the investigation of the types of applications, which can be implemented based on the MOA notation, and quantifiably analyzing the performance enhancement over the traditional methods. Different organizational applications, like in LAN distributed systems implementing concurrency, and Internet and intranet IP address generators can be investigated to see how much MOA notation can enhance their performance and/or add more functionality to them.

This category of future work can focus on forming finite state automata for some of the functions that can be implemented using the MOA. The implementation can take either directions of the software or the hardware. The software approach can be based on the MOA library designed in this thesis. Also the hardware approach can utilize the MOA package implemented in this thesis and the Chameleon board designed in previous research, by reprogramming the application FPGA to handle other functions [2] [3].

Another approach could be taken by designing a hardware board that takes an input of operands of variable dimension and shape, and a specific operation from the set of operations that the chip supports, and use the MOA approaches of flattening and partitioning arrays, and of decomposing the operations into a series of primitive additions and offsets based on the array shape. The Block Diagram in figure 8-2 represents the suggested board.

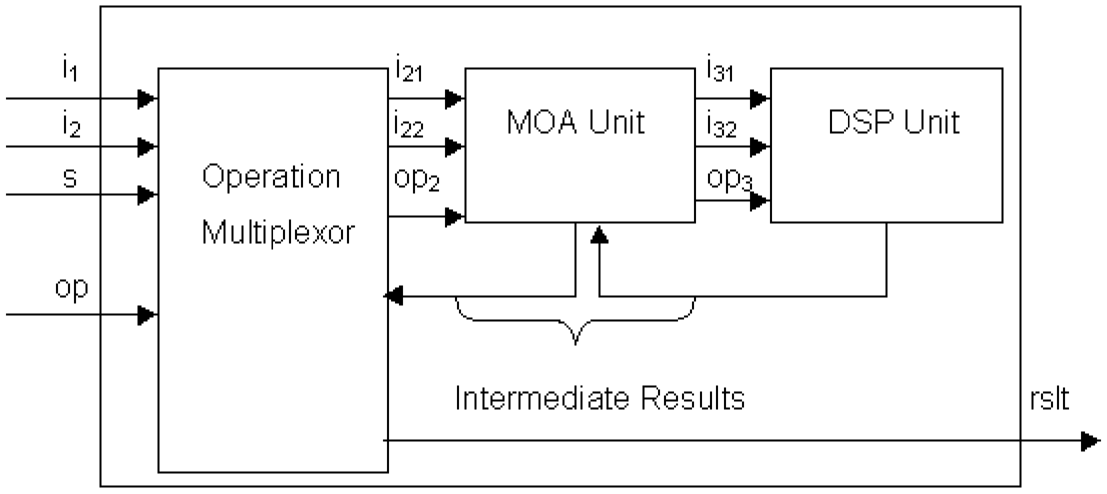


Figure 8-2: Block Diagram for a hardware MOA operations Board

In figure 8-2, the operation multiplexor takes two input operands ($i_1$, and $i_2$), the shape of the operands "s", and the operation code as input "op". This unit decides the expression simplification operations that need to be applied on the operands to enhance the performance, using the MOA unit. It performs the operation needed using the DSP processor, after decomposing it to its primitive operations using predefined finite state automata for each operation. The MOA unit in the above diagram is responsible for the simplification of the input operands by applying the MOA techniques of flattening, partitioning, indexing, etc, and performing the MOA functions on the input operands before sending them to be processed by the DSP processor. Thus, the MOA unit can be designed based on the VHDL package implemented in this thesis, which can play the role of a hardware implementation of the PSI compiler.

This board can be applied to any list of functions, by either feeding the operation multiplexor unit with finite state automata for any operation that is already analyzed externally (manually or using the MOA test bench provided in this thesis), or by designing an intelligent algorithm that parses the operation definition in the traditional programming style, or taking an input of the traditional equation as a string, and analyzing it to produce its equivalent equation in MOA notation.

# BIBLIOGRAPHY

[1] Mullin, Lenore, "A Mathematics of Arrays", Ph. D. Dissertation, Syracuse University, December 1988.

[2] H. Pottinger, W. Eatherton, J. Kelly, L. Mullin and T. Schifelbein, "Hardware Assists for High Performance Computing Using a Mathematics of Arrays", Proceedings of the ACM Symposium on Field Programmable Gate Arrays (FPGA95), Monterey, California, p39-45, February 1995.

[3] W. Eartherton. J. Kelly, T. Schiefelbein. H. Pottinger. L.R. Mullin, and R. Ziegler, "An FPGA Based Reconfigurable Coprocessor Board Utilizing a Mathematics of Arrays", Proceedings of the third international ACM symposium on Field-programmable gate arrays, 1995

[4] Lenore Mullin, and Scott Thibault, "A Reduction Semantics for Arrays Expressions: The PSI Compiler", Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri 65401, CSC-94-05, March 9, 1994.

[5] "The PSI Compiler v0.4 for MOAL to C – User's Guide", July 28, 1994.

[6] Lenore Mullin, and Michael A. Jenkins, "Effective Data Parallel Computation using the PSI-Calculus", July 25, 1995.

[7] Lenore Mullin (PA), and Werner Kluge, "On Programming Scientific Applications in a Functional Language Extended by a - $\psi$ Calculus Subsystem for Array Operations", March, 13, 1995

[8] Rafael C. Gonzalez, and Richard E. Woods, "Digital Image Processing", Addison-Wesley Publishing Company, 1993.

[9] Yi Pan, Keqin Li, and Si-Qing Zheng, "Fast Nearest Neighbor Algorithms on Linear Array with a Reconfigurable Pipelined Bus System", Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks, 1997.

[10] Hoda A. Khalil, "Tiling as a Loop Parallelisation Technique", A Master

of Science Thesis Dissertation, American University In Cairo, May 2000.

[11] Klaus Berkling, “Arrays And The LAMBDA Calculus”, CASE Center and School of Computer and Information Science, Syracuse University, May 1990.

[12] Guang R. Gao, “Compiling Issues of Monolithic Arrays”, School of Computer Science. McGill University, in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[13] John T. Feo, “Arrays in Sisal”, Lawrence Livermore National Laboratory, in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[14] C. McCrosky, K. Roy and K. Sailor, "Falafel: Arrays in a Functional Language", Montreal Workshop on Arrays, Functional Programming and Parallelism, Montreal, 1990.

[15] Robert Bernecky, “Compiling APL”, Snake Island Research, Inc., in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[16] Young Won Lim, Neungsoo Park, and Viktor K. Prasanna, “Efficient Algorithms for Multi-dimensional Block-Cyclic Redistribution of Arrays”, Proceedings of the 1997 International Conference on Parallel Processing (ICPP’97), IEEE, 1997.

[17] Janice Glasgow, “Array Theory and Knowledge Representation”, Department of Computing and Information Science, Queen’s University Kingston, in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[18] C. Barry Jay, “Shape Analysis for Parallel Computing”, Proceedings of the fourth international parallel computing workshop: {I}mperial {C}ollege {L}ondon, September 1995.

[19] C. Barry Jay, “A Semantics for Shape”, School of Computing Sciences, University of Technology, Sydney, 1995.

[20] C. Barry Jay, “Costing Parallel Programs as a function of Shapes” , School of Computing Sciences, University of Technology, Sydney, April 1999.

[21] Bradford L. Chamberlain, E. Christopher Lewis, Calvin Lin, and Lawrence Snyder, “Regions: An Abstraction for Expressing Array Computation”, University of Washington, ACM, 1999.

[22] Robert Bernecky, “Array Morphology”, Proceedings of the international conference on APL, 1993.

[23] Trenchard More, “An array-theoretic look beyond APL2 and Nial”, Conference proceedings on APL 90, 1990

[24] C. Barry Jay, “The FISh Language Definition”, School of Computing Sciences, University of Technology, Sydney, Oct. 1998.

[25] Guy Lapalme, “Arrays in Haskell”, Département d’informatique et de recherché opérationnelle, Université de Montréal, in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[25] Lawrence Snyder. “A ZPL Programming Guide (Version 6.3)”, Technical Report, University of Washington, January, 1999.

[26]Lenore Mullin, “Psi, The indexing Function: A Basis for FFP with Arrays”, Department of Computer Science and Electrical Engineering, University of Vermont, appeared in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, June 1990.

[27] M. A. Jenkins, and Lenore Mullin, “A Comparison of Array Theory and A Mathematics of Arrays”, in Arrays Functional Languages, Parallel Systems, Kluwer Academic Publishers, 1991.

[28] Ed Ashcroft, “Multidimensional Declarative Programming: The New Lucid of The New Book”, Computer Science and Engineering Department, Arizona State University. Sept. 1994.

[29] Parthasarathy Ranganathan, Sarita Advc, and Norman P. Jouppi, “Performance of Image and Video Processing with General-Purpose

Processors and Media ISA Extensions", Proceedings of the 26th annual international symposium on Computer architecture, 1999.

[30] H. Samsom, F. Franssen, F. Catthoor, and H. De Man, "System Level Verification of Video and Image Processing Specifications", Proceedings of the eighth international symposium on System synthesis, 1995.

[31] Eric Hung-Yu Tseng, and Jean-Luc Gaudiot, "Two Techniques for Static Array Partitioning on Message Passing Parallel Machines", University of Southern California, Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT' 97), IEEE, 1997.

[32] Bo Einarsson and Yurij Shokin, "Fortran 90 for the Fortran 77 Programmer", August 1996.

[33] C. Barry Jay, "Shape in computing", ACM Computing Surveys, 1996.

[34] Sylvester, J. J., "A Constructive Theory of Partitions", American Journal of Mathematics, V (1882), VI (1884). Reprinted in Mathematical Papers, 4, (1).

[35] Cayley, A., "The Theory of Linear Transformations", An Elementary Treatise on Elliptic Functions, First Edition (1876) Cambridge, Second Edition (1895) George Bell and Sons. Reprinted by Dover Publications, New York, 1961.

[36] Iverson, K. E., " A Programming Language", John Wiley and Sons, New York, 1962.

[37] Abrams, P.S., "An APL Machine", TR SLAC-114 UC-32(MISC), Stanford Linear Accelerator Center, February, 1970.

[38] Guibas, L. J., and Wyatt, D. K., "Compilation and Delayed Evaluation in APL", In Conference Record of the fifth Annual ACM Symposium on the Principles of Programming Languages, ACM, January 1978.

[39] Perlis, A. J., "Steps toward an APL Compiler - Updated", TR No. 24, Department of Computer Science, Yale University, March 1975.

[40] Tu, H., "FAC: A Functional Array Calculator and it's Application to APL and Functional Programming", Ph.D. Dissertation, Yale University, New Haven, Conn., 1985.

[41] Falkoff, A. D., Iverson, K. E, and Sussenguth, E. H., "A Formal Description of System/360", IBM Systems Journal, 3, (198), 1964.

[42] Hill, F. T., "Introducing AHPL", Computer, Dec., 1974.

[43] Schneider, R. "APLSIM: An Interactive Timimg Simulator for Large Integrated Circuits", Miconex '87, Microelectronics Conference, Winnepeg, Canada, 1987.

[44] Reynolds, J. C., "Reasoning About Arrays", CACM, 22, 1979.

[45] Gerhart, S. L., "Verification of APL Programs", Ph.D. Dissertation, CMU, November 1972.

[46] Pitchumani, V. and Stabler, E. "A Formal Method for Design Verification", 19th Design Automation Conference, IEEE, 1982.

[47] Backus, John, "Can Programming be liberated from the Von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ACM, Vol. 21(8): 613:641, August 1978.

[48] Whitehead, A. N. "A Treatise on Universal Algebra with Applications", Hafner Publishing, New York, 1960.

# APPENDIX A: CMOA Class

The following is the declaration of the CMOA class in the MOA.h file. Overloading allowed the implementation of several forms of the same function. Some functions are overloaded to operate on MOA structures passed as parameters from the calling user, and repeated again without the input MOA structure, to operate on the private content of the CMOA object (such as take and drop functions). Other functions are overloaded to allow for different indexing methods, like accessing the MOA structure data elements using the multi-dimensional index, or the scalar index in the flat array (such as Convolve function). Other functions are overloaded to accept different argument lists, leading to different operations (such as Binary and Unary Dot functions).

```
// MOA.h: interface for the CMOA class.
//
//////////////////////////////////////////////////////////////////////

#include <afxtempl.h>

#if !defined(AFX_MOA_H__68246453_8717_11D4_9CB7_0050DA465A48__INCLUDED_)
#define AFX_MOA_H__68246453_8717_11D4_9CB7_0050DA465A48__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include <math.h>
#include <stdlib.h>
#include <search.h>

typedef struct tagMOA_rec {
int dimn;
int * shape;
DWORD elements_ub;
DWORD * elements;
int lelm_size;

} MOA_rec ;

typedef struct tagRestruct_rec {
MOA_rec  MOA_ret;
```

```
int * indexes;
} Restruct_rec  ;

typedef enum tagOrientations   // Declares Orientation Names
{
   UpperLeft = 0,    // upper left corner,
   UpperMiddle = 1,      // upper middle corner
   UpperRight = 2,     // upper right corner
   RightMiddle = 3,    // right middle corner
   LowerRight = 4,     // lower right corner
   LowerMiddle = 5,      // lower middle corner
   LowerLeft = 6,    // lower left corner
   LeftMiddle = 7,   // left middle corner
   Center = 8,       // center
   Undefined = 10      // can't define Orientation
} Orientations;


// --------------- MOA Class Definetion
class CMOA
{
public:

   CMOA(int dimn_param, int * shape_param, DWORD * elements_param, DWORD elements_ub);
   CMOA(MOA_rec * MOA_param);
   virtual ~CMOA();

   MOA_rec * GetMOA(void);
   void CopyMOA (MOA_rec * rslt);

   int Dimn (); //GetDimn
   int * Shape (); //GetShape
   void Ravel (DWORD * rslt); //GetElements

   bool SetDimn (int dimn);
   bool SetShape (int * Shape, int shape_ub);
   bool SetElements(DWORD * Elements, DWORD elements_ub);
   void SetMOA (int dimn_param, int * shape_param, DWORD * elements_param, DWORD elements_ub);
   void SetMOA(MOA_rec * MOA_param);

   bool AssignDataElement (int * mlt_index, DWORD new_value);
   bool AssignDataElement (DWORD flat_index, DWORD new_value);

   void Iota (DWORD N, DWORD * rslt);
   DWORD Tau (int * array_in, int array_ub);
   DWORD Pi (DWORD * array_in, long array_ub);
   DWORD Gamma (int * ind, int ind_ub, int * arr_shape, int shape_ub, bool Front = false);
   void Gamma_Inverse (DWORD ind, int * arr_shape, int shape_ub, int * rslt);
   MOA_rec *  Psi (int * ind, int ind_ub);
```

```
  void Psi (int * ind, int ind_ub, MOA_rec * rslt);
  void Psi (int * ind, int ind_ub, MOA_rec * MOA_in, MOA_rec * rslt);

  bool IsValidIndex (int * shape, int shape_ub, int * index);
  void NextIndex (int * Prev_Index, int p_ind_ub);
  void NextIndex (int * shape, int shape_ub, int * Prev_Index, int p_ind_ub);
  void NextIndex (int * shape, int shape_ub, int * Start_Index, int * Prev_Index, int p_ind_ub);

  void Slice (int slice_size, int Slice_Pos, bool Front, MOA_rec * MOA_in, MOA_rec * rslt);
  void GetFirstSlice (int slice_size, int * start_slice_ind, bool Front, MOA_rec * MOA_in, MOA_rec * rslt);
  void GetNextSlice (int slice_size, int * prev_slice_ind, bool Front,  MOA_rec * MOA_in, MOA_rec * rslt);

  void SliceOnAxe (int slice_Axe, int * slice_ind, MOA_rec * MOA_in, MOA_rec * rslt);


  void GetFirstSliceOnAxe (int slice_size, int * slice_pos, int * start_slice_ind, MOA_rec * MOA_in, MOA_rec *
rslt);
  void GetNextSliceOnAxe (int slice_size, int * slice_pos, int * prev_slice_ind, MOA_rec * MOA_in, MOA_rec *
rslt);

  void VecAssignTo (int * Array_1, int array1_ub, int * Array_2, int array2_ub);
  bool VecIsEqual (DWORD * array, DWORD array_ub);
  bool VecIsEqual (int * array, int array_ub);
  bool VecIsEqual (int * Array1, int array1_ub, int * Array2, int array2_ub);

  bool IsContainElement(int element);
  bool IsContainElement(int element, int * array_in, int array_ub);
  bool IsContainElement(DWORD element);
  bool IsContainElement(DWORD element, DWORD * array_in, DWORD array_ub);

  int ElementFoundCnt(int element);
  int ElementFoundCnt(int element, int * array_in, int array_ub);
  DWORD ElementFoundCnt(DWORD element);
  DWORD ElementFoundCnt(DWORD element, DWORD * array_in, DWORD array_ub);

  void Take (int * ind, int ind_ub, Restruct_rec * rslt);
  void Take (int * ind, int ind_ub, MOA_rec * MOA_in, Restruct_rec * rslt);

  void Partition (int * From_ind, int * Length_ind, Restruct_rec * rslt);
  void Partition (int * From_ind, int * Length_ind, MOA_rec * MOA_in, Restruct_rec * rslt);

  void Drop (int * ind, int ind_ub, Restruct_rec * rslt);
  void Drop (int * ind, int ind_ub, MOA_rec * MOA_in, Restruct_rec * rslt);

  void Reshape (int * new_shape, int n_shape_ub, MOA_rec * rslt);
  void Reshape (int * new_shape, int n_shape_ub, MOA_rec * MOA_in, MOA_rec * rslt);

  void Catenate (MOA_rec * MOA_in, int Cat_DIM, MOA_rec *  rslt);
  void Catenate (MOA_rec * MOA_1, MOA_rec * MOA_2, int Cat_DIM, MOA_rec * rslt);
void Pack (MOA_rec * MOA_1, MOA_rec * MOA_2, MOA_rec * rslt);

DWORD op_on_dimn (char Op, int dimn, DWORD * index = NULL);
DWORD op_on_dimn (char Op, int dimn, MOA_rec * MOA_in, DWORD * index = NULL);

DWORD min_on_dimn (int dimn, DWORD * index = NULL);
DWORD max_on_dimn (int dimn, DWORD * index = NULL);
DWORD average_on_dimn (int dimn);
DWORD sum_on_dimn (int dimn);

DWORD min_on_dimn (int dimn, MOA_rec * MOA_in, DWORD * index = NULL);
DWORD max_on_dimn (int dimn, MOA_rec * MOA_in, DWORD * index = NULL);
DWORD average_on_dimn (int dimn, MOA_rec * MOA_in);
DWORD sum_on_dimn (int dimn, MOA_rec * MOA_in);

DWORD min_element (DWORD * elements, DWORD elm_ub);
DWORD max_element (DWORD * elements, DWORD elm_ub);
DWORD average_element (DWORD * elements, DWORD elm_ub);
DWORD sum_element (DWORD * elements, DWORD elm_ub);
DWORD op_element (char Op, DWORD * elements, DWORD elm_ub);

void scalar_op(char Op, DWORD scalar, bool FirstArgument = false);
void scalar_op(char Op, MOA_rec * MOA_in, DWORD scalar, MOA_rec * rslt);
void scalar_op(char Op, DWORD scalar, MOA_rec * MOA_in, MOA_rec * rslt);

void array_op(char Op, MOA_rec * MOA_in, MOA_rec * rslt);
void array_op(char Op, MOA_rec * MOA_1, MOA_rec * MOA_2, MOA_rec * rslt);

void Grade_Up (DWORD * array_in, DWORD array_ub);
void Grade_Down (DWORD * array_in, DWORD array_ub);

void Red (char Op, int Dim_No, MOA_rec * rslt);
void Red (char Op, MOA_rec * MOA_in, int Dim_No, MOA_rec * rslt);
void Scan_op (char Op, MOA_rec * MOA_in, int Dim_No, MOA_rec * rslt);

void Reverse(int Rev_dimn);
void Reverse(int Rev_dimn, MOA_rec * rslt);
void Reverse(MOA_rec * MOA_in, int Rev_dimn, MOA_rec * rslt);

void Rotate(int * Rot_ind, int Rot_ind_ub, MOA_rec * rslt);
void Rotate(int * Rot_ind, int Rot_ind_ub, MOA_rec * MOA_in, MOA_rec * rslt);

void Transpose(int * Trns_ind, int Trns_ind_ub);
void Transpose(int * Trns_ind, int Trns_ind_ub,  MOA_rec * MOA_in, MOA_rec * rslt);

void Omega(char Op, int dimn_1, MOA_rec * MOA_1, int dimn_2, MOA_rec * MOA_2, MOA_rec * rslt);
// Unary Dot Operation performs Outer Product Operation
void Dot(char Op, MOA_rec * MOA_1, MOA_rec * MOA_2, MOA_rec * rslt);
// Binary Dot Operation performs Inner Product Operation
```

```
  void Dot(char Dot_Op, char Red_Op, MOA_rec * MOA_1, MOA_rec * MOA_2, MOA_rec * rslt);


  void MatrixMultiplication (MOA_rec * MOA_1, MOA_rec * MOA_2, MOA_rec * rslt);


  void Connectivity(MOA_rec * MOA_in, int conn_dimn, int * elm_ind, DWORD * * connected_set, DWORD * *
connected_indices, DWORD * ubound);
  void Neighbors(MOA_rec * MOA_in, int * mlt_elm_ind, int num_steps_away, DWORD * * neighbors_set,
DWORD * * neighbors_indices, DWORD * ubound);
  void Neighbors(MOA_rec * MOA_in, DWORD flat_elm_ind, int num_steps_away, DWORD * * neighbors_set,
DWORD * * neighbors_indices, DWORD * ubound);


  void Compress (MOA_rec * MOA_in, MOA_rec * Bool_MOA, MOA_rec * rslt);
  void Expand (MOA_rec * MOA_in, MOA_rec * Bool_MOA, MOA_rec * rslt);


  Orientations GetOrientation (MOA_rec * MOA_in, int * mlt_elm_index, int num_steps_away);
  Orientations GetOrientation (MOA_rec * MOA_in, DWORD flat_elm_index, int num_steps_away);


  void Convolve (int * mlt_index, int num_steps_away, Orientations Orientatation, MOA_rec * MOA_in, MOA_rec
* rslt);
  void Convolve (DWORD flat_index, int num_steps_away, Orientations Orientatation, MOA_rec * MOA_in,
MOA_rec * rslt);


  void Convolution (MOA_rec * Mask, DWORD Div_Factor, DWORD Multip_Factor);
  void Convolution (MOA_rec * MOA_in, MOA_rec * Mask, DWORD Div_Factor, DWORD Multip_Factor,
MOA_rec * rslt);


  void DoubleConvolution (MOA_rec * Mask1, MOA_rec * Mask2, char MaskRedOp, float Div_Factor, float
Multip_Factor);
  void DoubleConvolution (MOA_rec * MOA_in, MOA_rec * Mask1, MOA_rec * Mask2, char MaskRedOp, float
Div_Factor, float Multip_Factor, MOA_rec * rslt);


private:
  MOA_rec * MOA_val;


};


#endif // !defined(AFX_MOA_H__68246453_8717_11D4_9CB7_0050DA465A48__INCLUDED_)
```

# APPENDIX B: CMOAImage Class

The CMOAImage class provides an interface between the MOA structure implemented in the CMOA class and the bmp file format, by containing an object of the CMOA class. Some Image processing operations were implemented only to demonstrate the effectiveness of the design on 2-D applications.

```
// MOAImage.h: interface for the CMOAImage class.
//
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_MOAIMAGE_H__A9E9F29E_A582_11D4_9BE5_0080AD97CBA2__INCLUDED_)
#define AFX_MOAIMAGE_H__A9E9F29E_A582_11D4_9BE5_0080AD97CBA2__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "MOA.h"
#include <math.h>
#include <mmsystem.h>
#include <windowsx.h>

class CMOAImage
{
public:
   CMOAImage();
   CMOAImage(MOA_rec * Image_MOA, WORD bit_Count_Param);
   CMOAImage(BYTE * ImagePtr, LPBITMAPINFOHEADER bih);
   CMOAImage(LPCTSTR lpszPathName);
   virtual ~CMOAImage();

   CString PathName;
   HANDLE h_Image;
   BYTE * ImagePtr;
   BITMAPFILEHEADER bmfh;
   BITMAPINFOHEADER bmih;
   float WidthScale,HeightScale;
   CMOA * cls_MOA;

   bool ImageToMOA(HANDLE h_Image);
   bool ImageToMOA(HANDLE h_Image, MOA_rec * p_MOA);
   bool ImageToMOA(BYTE *ImagePtr, LPBITMAPINFOHEADER bmih);
```

```
  bool ImageToMOA(BYTE *ImagePtr, LPBITMAPINFOHEADER bmih, MOA_rec * p_MOA);

  bool MOAToImage(BYTE *ImagePtr, LPBITMAPINFOHEADER bmih);
  bool MOAToImage(HANDLE h_Image);
  bool MOAToImage(MOA_rec * p_MOA, HANDLE h_Image);
  bool MOAToImage(MOA_rec * p_MOA, BYTE *ImagePtr, LPBITMAPINFOHEADER bmih, WORD
bit_Count_Param);

  // Transformations
  void VerticalFlip ();
  void HorizontalFlip ();
  void TransposeImage ();

  // Filtering
  void LowPassFilter();
  void HighPassFilter();
  void PrewittFilter();
  void SobelFilter();
  void CustomFilter(MOA_rec * Mask);

  // Segmentation
  void DetectPoints();
  void DetectHLine();
  void DetectVLine();
  void Detect45Line();
  void Detect315Line();

  // Morphology
  void Dilation (int num_steps_neighborhood);
  void Erosion (int num_steps_neighborhood);
  void Opening (int num_steps_neighborhood);
  void Closing (int num_steps_neighborhood);

};

#endif // !defined(AFX_MOAIMAGE_H__A9E9F29E_A582_11D4_9BE5_0080AD97CBA2__INCLUDED_)
```

# APPENDIX C: CMOAVideo Class

The CMOAVideo class provides an interface between the MOA structure implemented in the CMOA class, by containing an object of the CMOA class, like the case in the CMOAImage class. Some Video processing operations were implemented only to demonstrate the effectiveness of the design.

```
// MOAVideo.h : interface of the CMOAVideo class
//
//////////////////////////////////////////////////////////////////////////////

#if !defined _VIDEO_DOC_H
#define _VIDEO_DOC_H

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "MOA.h"
#include "MOAImage.h"
#include <windowsx.h>
#include <mmsystem.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include <ctype.h>
#include <vfw.h>
#include "Muldiv32.h"
#include <wingdi.h>

//--------------------------------------------------------
class CMOAVideo
{
protected:

// Attributes
public:

   UINT gwZoom;        // zoom (divide by 4)
   LONG timeStart;     // cached start, end, length
   LONG timeEnd;
   LONG timeLength;
   LONG curTime;
   CString PathName;
```

```
  AVIFILEINFO * pfi;
  PAVIFILE pfile;
  PAVISTREAM gapavi[MAXNUMSTREAMS];   // the current streams
  PGETFRAME gapgf[MAXNUMSTREAMS]; // data for decompressing
  HDRAWDIB  ghdd[MAXNUMSTREAMS];// drawdib handles
            // video
  int gcpavi;         // # of streams
  PAVISTREAM gpaviVideo;                // 1st video stream found
  int giFirstVideo;               // index of gapavi for 1st Video stream
  PAVISTREAM gpaviAudio;                // 1st audio stream found
  DWORD gdwMicroSecPerPixel;     // scale for video

  AVICOMPRESSOPTIONS gaAVIOptions[MAXNUMSTREAMS];
  LPAVICOMPRESSOPTIONS galpAVIOptions[MAXNUMSTREAMS];

  LPVOID lpAudio;     // buffer for painting
  bool IsPlay, IsMOAPlay;
  LONG startPlayTime;

// Operations
public:

  void InitVideo(LPCTSTR lpszPathName);
  void FreeVideo();
  LPBITMAPINFOHEADER AVIBitMapInfoHeader(int iStream, LONG lTime);
  BYTE * AVIGetBitMapImage(int iStream, LONG lTime);
  LPBITMAPINFOHEADER AVIFrameBitMapInfoHeader(int iStream, LONG lFrame);
  BYTE * AVIFrameGetBitMapImage(int iStream, LONG lFrame);
  bool AviToMoa ();
  bool GetFirstFrameMoa ();
  void AviFrameIntoMOA (DWORD FrameTime, LPBITMAPINFOHEADER lpbi, BYTE * ImagePtr, MOA_rec
* VideoMOA_FramePixels);
  bool MOAFirstFrametoAVI ();
  void MOAIntoAviFrame (MOA_rec * VideoMOA_FramePixels, DWORD FrameTime,
LPBITMAPINFOHEADER lpbi, BYTE * ImagePtr);
  void ReplaceFrameInMOA (MOA_rec * VideoMOA_FramePixels, DWORD Frame_No, MOA_rec *
FrameImage);

  bool MoaToAvi();
  void VideoMOAIntoImageMOA (MOA_rec * VideoMOA_FramePixels, DWORD FrameTime, MOA_rec *
FrameImage);
  MOA_rec *  FrameFromMOAVideo(int Frame_No);

  bool FlipFrame (int Frame_No);
  void FlipFramesVertically ();
  void FlipFramesHorizontally ();
  void ReverseFramesOrder ();

  MOA_rec * VideoSubtract(void);
```

```
   // This function performs the Xoring operator over the frames in
   // MOA structure, producing a 2-dimensional MOA structure of the
   // resulting differential image, denoting the motion.

   CMOAImage * MotionDetection ();

   CMOA * cls_MOA;
   CMOAImage * m_pMOAImage;
   DWORD * VedioSubtract;
   MOA_rec * MOAVideoImage;

// Implementation
public:
   CMOAVideo();
   virtual ~CMOAVideo();

};

#endif // !defined _VIDEO_DOC_H
/////////////////////////////////////////////////////////////////////////////
```

# APPENDIX D: MOA VHDL Package

The hardware implementation did not elaborate to implement all the MOA notation like the case in the C++ implementation. The following important operations were implemented to prove the correctness of the design in the hardware VHDL implementation using Renoir.

```
--
-- VHDL Package Header MOA_Lib.Hdr
--
-- Created:
--          by - mhelal.student (ultra)
--          at - 15:18:34 07/18/00
--
-- Generated by Mentor Graphics' Renoir(TM) 99.5 (Build 108)
--

package MOA_Lib is
  CONSTANT ub:integer := 1023;
  TYPE integers IS ARRAY (ub downto 0) OF integer;

  TYPE MOA_rec IS RECORD
     dimn: integer;
     shape: integers;
     elm_ub: integer;
     elements: integers;
  END RECORD;

  TYPE TK_DRP_rec IS RECORD
      MOA_ret : MOA_rec;
      Gamma_indexes: integers;
  END RECORD;

  FUNCTION Tau(array_in: integers; ub:integer) RETURN integer;
  FUNCTION Pi(array_in: integers; ub:integer) RETURN integer;
  FUNCTION Psi(ind: integers; ind_ub: integer; MOA_in: MOA_rec) RETURN MOA_rec;
  FUNCTION Gamma(ind: integers; ind_ub: integer; arr_shape: integers; ub:integer) RETURN integer;
  FUNCTION Gamma_reverse(ind: integer; shape: integers; ub:integer) RETURN integers;

  FUNCTION Iota (N: integer) RETURN integers;
  FUNCTION scalar_op(Op: CHARACTER; MOA_in: MOA_rec; scalar: integer)  RETURN MOA_rec;
  FUNCTION array_op(Op: CHARACTER; MOA_1: MOA_rec; MOA_2 :MOA_rec) RETURN MOA_rec;
  FUNCTION SliceOnAxe (slice_Axe: integer; slice_ind: integers; MOA_in: MOA_rec) RETURN MOA_rec;
  FUNCTION ElementFoundCnt(element: integer; array_in: integers; array_ub: integer) RETURN integer;
  FUNCTION VecIsEqual (Array_1: integers; array1_ub: integer; Array_2: integers; array2_ub: integer) RETURN
boolean;


  FUNCTION Reverse(MOA_in: MOA_rec; Rev_dimn: integer) RETURN MOA_rec;
  FUNCTION Scan_op (Op: CHARACTER; MOA_in: MOA_rec; Dim_No: integer) RETURN MOA_rec;
  FUNCTION Rotate(Rot_ind: integers; Rot_ind_ub: integer; MOA_in: MOA_rec) RETURN MOA_rec;
  FUNCTION Transpose(Trns_ind: integers; Trns_ind_ub: integer;  MOA_in: MOA_rec) RETURN MOA_rec;
  FUNCTION op_element (Op: CHARACTER; elements: integers; elm_ub: integer)  RETURN integers;
  FUNCTION Grade_Up (array_in: integers; array_ub: integer) RETURN integers;
  FUNCTION Grade_Down (array_in: integers; array_ub: integer) RETURN integers;
  FUNCTION Compress (MOA_in: MOA_rec; Bool_MOA: MOA_rec) RETURN MOA_rec;
  FUNCTION Expand (MOA_in: MOA_rec; Bool_MOA: MOA_rec) RETURN MOA_rec;
  FUNCTION Omega(Op: CHARACTER; dimn_1: integer; MOA_1: MOA_rec; dimn_2: integer; MOA_2
:MOA_rec) RETURN MOA_rec;


  -- Unary Dot Operation performs Outer Product Operation
  FUNCTION UDot(Op: CHARACTER; MOA_1: MOA_rec; MOA_2 :MOA_rec)  RETURN MOA_rec;


  -- Binary BDot Operation performs Inner Product Operation
  FUNCTION BDot(Dot_Op: CHARACTER; Red_Op: CHARACTER; MOA_1: MOA_rec; MOA_2
:MOA_rec)  RETURN MOA_rec;


  FUNCTION NextIndex(shape: integers; shape_ub: integer; Prev_Index :integers) RETURN integers;


  FUNCTION Take(ind: integers; ind_ub: integer; MOA_in :MOA_rec) RETURN TK_DRP_rec;
  FUNCTION Drop(ind: integers; ind_ub: integer; MOA_in :MOA_rec) RETURN TK_DRP_rec;
  FUNCTION Reshape(new_shape: integers; new_shape_ub: integer; MOA_in :MOA_rec) RETURN MOA_rec;
  FUNCTION Catenate(MOA_1: MOA_rec; MOA_2 :MOA_rec; CAT_dimn: integer) RETURN MOA_rec;
  FUNCTION Red(Op: CHARACTER; MOA_in: MOA_rec; Dim_No: integer) RETURN MOA_rec;


end MOA_Lib;
```