A Uniform Way of Reasoning About Array-Based Computation in Radar

Algebraically Connecting the Hardware/Software Boundary

Lenore R. Mullin
MIT Lincoln Laboratory
Lexington, MA 02420 [*]

August 2002

Abstract

Embedded processing requirements will exceed 10 trillion operations per second in the 2005-2010 time frame. Consequently, efficient use of processors and memory, at all levels, is essential. In a defense environment, e.g. radar, as well as medical and other real-time embedded systems, operations are primarily array-based. Although languages support high level, monolithic, array based computation through classes, functions, templates(C++), and grammars(Fortran95, ZPL), limited optimizations occur to eliminate array valued temporaries, which for embedded real time systems, is enormous. Without an underlying theory of arrays, e.g. an algebra and index calculus, it is difficult, if not impossible to provide such optimzations. This paper presents *A Mathematics of Arrays* and *Psi Calculus*. Together they are used to reason about array based computation in radar; the algorithm, decomposition, mapping to processors and memory, performance.

Keywords: embedded digital systems, radar, signal processing, arrays, high performance, index calculus, shapes, shape polymorphisms, psi, MoA.

[*]This work was done while on sabbatical from the Department of Computer Science, U Albany, SUNY, Albany, NY 12222, lenore@cs.albany.edu

Contents

List of Figures

1 Introduction

1.1 Reasoning about Radar

In a defense environment, the amount and type of information may vary but the need to respond to all types of information is immediate and constrained, e.g. real-time in a computational environment with space, power, and time limitations. This is particularly true for defense simulations, radar and other forms of signal and image processing. Consequently, the Department of Defense makes major investments to develop, maintain, and port high-performance scientific and engineering software to advanced hardware in pursuit of these endeavors. COTs simplify this task when coupled with mainstream software, e.g. C++, and software tools, e.g. PETE[20], but *guarantees* of performance or correctness depend on the solution of open questions in computer science: software engineering, compiler technology, programming languages, and theory of computation.

Reasoning about radar, from a *computational perspective*, requires a reasoning about the data structures which define algorithms for radar. Most, if not all such algorithms, are characterized by matrices, the composition of matrices: QR and FIR, and the composition of matrix algorithms: FIR(FFT($B^H \times A$)). Such a preponderance of array based computation necessitates an understanding, and ideally a theory, of how such algorithms relate to their various computational platforms. Contemporary programming languages(Matlab, C++, Fortran 90/95), often provide the expressivity but not the performance required for real time systems. They also do not provide a way to understand and predictively analyze the performance of algorithms given diverse memory and processor layouts. Even when attempts are made to communicate with the compiler, e.g. distributions for multiprocessing[33, 69], the semantics of what is desired is delivered without a guarantee of how the algorithm was built. Consequently, reasoning about radar requires reasoning about arrays: the algorithm, decompositions, levels of memory, and data-flow, and thus necessitates a theory of arrays: an algebra of arrays and a calculus of indexing. We describe how *A Mathematics of Arrays*(MoA) and the *Psi Calculus*[59, 37] are used to reason about radar by algebraically connecting an algorithm to software and its underlying hardware. We show how shapes characterize arrays and how the *psi* function,ψ, in conjunction with shapes, builds the algebra.

The first step in an MoA/Psi analysis, is to design and derive an algorithm given a single processor architecture and memory large enough to hold all arrays. The next step is to add or change architectural attributes. Applying this analysis, three algorithms commonly used in radar processing are investigated: the Time-Domain Convolution, Modified Gram-Schmidt, QR[27] and the composition of multiple Matrix Multiplications, commonly used in Beamforming. Each is first expressed in MoA then reduced *manually*, by the application of linear transformations in the *Psi Calculus*, to both a semantic(DNF) and operational(ONF) normal form. The DNF denotes a reduced form in terms of Cartesian coordinates independent of layout(row, column major, or regular sparse) or environment(levels, sizes, and speeds of memory and processors). Normal forms describe computation *without* unnecessary temporaries typical of today's programming languages. The ONF reflects the attributes necessary to build the DNF, i.e. translates Cartesian coordinates into starts, stops and strides across memories, etc. Consequently, a one to one correspondence exists between the DNF(the meaning) and ONF(how to build it). It is linked by the MoA function *gamma*[59, 37].

Although all designs begin with a manual analysis, each step in the reduction process is a linear transformation on shapes, and is thus mechanizable, at run or compile time in a deterministic amount of time. Numerous studies have been undertaken to mechanize *Psi Reduction*. The first attempt was a prototype compiler[72, 57, 71, 55, 51]. Later attempts involved the modification of existing compilers, C[53] and Fortran[62], and the use of C++ functions and classes[32]. Here we discuss the mechanization of *Psi Reduction* using C++ expression templates[78, 80, 77] and PETE[20]. We discuss and present algebraic dimension lifting and data restructuring coincident with decompositions and memory hierarchies. We then use these descriptions to represent and predict performance on the RAW[19] Polymorphic Computing Architecture(PCA).

In the future, we envision that scientific programs will be developed in an interactive development environment. Such environments will combine human judgment with compiler-like analysis, transformation, and optimization techniques. A programmer will use knowledge of problem semantics to guide the selection of the transformations at each step, but the application of the transformation and verification of its safety will be done mechanically. A given sequence of transformations will be at a point where a version of the code has been obtained such that subsequent optimizations can be left to an available optimizing compiler. This feasibility study is a step towards the development of such an interactive program development environment.

Mullin's *Psi Calculus* plays an underlying role in this feasibility study. This calculus provides a unified mathematical framework for representing and studying arrays, array operations and array indexing. It is especially useful for developing algorithms centered around transformations involving array addressing, decomposition reshaping, distribution, etc. Each of the algorithm transformations carried out here can be expressed in terms of the *Psi Calculus*, and we used the *Psi Calculus* to verify the correctness of program transformations in the development of all algorithms presented herein.

1.2 Outline

A historical perspective of computing and the origins of array based scientific computation are presented first. The *Psi Calculus* and MOA are introduced as a theoretical tool to define and reason about array based computation. Then, we show how to apply this reasoning to three important algorithms used in Radar: FIR(Time Domain Convolution), QR(Modified Gram Schmidt), and the composition of multiple matrix multiplications(Beamforming). We then *manually* algebraically reformulate the Convolution and QR to include time, processor, and cache dimensions by partitioning the shape vector of input arrays. To illustrate mechanization and performance optimizations, *Psi Reduction* of the convolution algorithm, *for uniprocessors*, was built using C++ expression templates. The Convolution algorithm, manually extended by time, processor, and cache dimensions, was used to predict and validate performance on RAW. The QR extended by processor, and cache dimensions, was built from its ONF in Fortran 90 while the composition of matrix multiply was built directly from its DNF. The later was attempted to determine to what extent current compilers could optimize. Preliminary experiments, for both the Convolution and Matrix Multiply, were performed on NCSA's SGI Origin 2000. Finally, essentials of an Abstract Machine(AM) are presented, followed by conclusions and future research.

2 Scientific Computing Then and Now

From its onset, computer science has attempted to describe ways to drive computer architectures by theorizing about computation, developing programming languages, compilers, interpreters, and operating systems, while growing software engineering disciplines. Initially, the fastest real time scientific codes were written by the engineers who designed them, primarily because they had the deepest understanding of the machine, and often the problem. To create higher level interfaces to machines, various programming languages were founded on one or more of the models of computation: Turing[74, 75], Church[14, 43] and Göedl[26].

2.1 Compilers, Interpreters, and Operating Systems

Compilers, interpreters and operating systems characterize the first attempts at systems software. Each enabled a high level interface between the user and machine. Within these environments, programming languages emerged: e.g. Fortan[4], APL[38], and Lisp[10], circa 1950-60.

2.1.1 Fortran

Fortran, was just that, a **For**mula **tran**slation language and promised, through compiler technology, to provide a high level of expressivity AND high performance. Fortran possessed(es) the data structure most common to scientific computation, the *array*, and its compilers were highly optimized to their target architectures, initially one real memory on one processor. Theorizing about compilers for Von Neumann machines was well defined[45]. However, with the addition of virtual memory, time sharing, levels of memory and processors such theories were found to be insufficient.

Libraries such as BLAS, LINPACK, LAPACK, ScaLAPACK, and ATLAS evolved with languages and compilers by paralleling the growth of language extensions, compiler improvements, and computer architectures: uni-processor, multiprocessor shared and distributed memory architectures, Non-Uniform Memory Architectures(NUMA) and PCAs, respectively. Even when finely tuned by highly skilled scientific programmers, these libraries *still pose* three major technological problems:

1. They must be *re-tuned* each time a new architecture or communications paradigm emerges.
2. Even if functions and subroutines compose, they produce intermediate, often large, array valued temporaries, `IFFT(FFT(A))` with space and performance consequences.

3. Highly skilled scientific programmers must be trained and retrained each time new software and hardware changes occur.

Despite all of this, FORTRAN is still viewed as a modern language for scientific computing[13] and thus is concerned about high performance improvements.

2.1.2 APL and Lisp

APL[38] and Lisp[10] emerged around the same time as Fortran and chose interpretation over compilation to encourage rapid prototyping. APL was based on an algebra[70] that was believed to be ideal for expressing scientific thought, Lisp was based on **Lis**t **p**rocessing and the *Lambda Calculus*[5]. The *APL-style* of programming used *monolithic arrays* and array operations and thus provided a very high level of expression to scientific programmers. **Independent of symbol set, syntax, and language environment**, the *algebra* became popular within the scientific community and continues to influence contemporary software and hardware languages, e.g. Matlab. APL's use of a supervisor was said to influence the first Unix operating system, and its shared variable processor[44], interprocess communication protocols. APL's algebra was used to define machines [1, 22, 17], hardware design languages[34] and parallel algorithms[21]. Attempts were made at compilation[11, 8, 49, 28, 65], and verification[25], concluding that APL, as a language *and* algebra, had too many anamolies. Consequently, Perlis[73] concluded that combining lambda calculus with an array algebra was the ideal model for array based computation. His insights resulted in numerous descriptions of how that could be done[7, 30, 29].

2.1.3 Functional Languages

Functional languages such as Miranda[76], or ML[48] provide little direct support for arrays. Haskell[31] includes arrays as data types but performance of the compiled code in not very competitive. SISAL[9], and all major data flow languages, e.g. ID and VAL[63, 2], provide control constructs for the traversal of array entries very similar to those of imperative languages, but strictly enforce the single assignment rule. Owing to very sophisticated compilation techniques, SISAL programs are known to outperform equivalent FORTRAN programs on multiprocessor systems[12]. However, SISAL does not offer substantial advantages in terms of programming techniques. Other than introducing another syntax, the programmer is still asked to specify array operations and iteration loops whose index variable and index ranges must be strictly adapted to array dimensionalities and shapes.

Integrating into languages a monolithic array processing concept[24], considerably improves high-level array processing. *Monolithic arrays*, are treated as conceptual entities which can be operated upon by high-level structuring and value-transforming primitives. Explicit specification of iteration loops can in many cases be avoided. Most structuring operations internally just move pointers or modify descriptor entities which specify, e.g., in the form of offsets, how restructured arrays must be mapped onto original arrays, rather than completely rewriting them.

2.2 Array Algebras

Beyond these pragmatic advantages, the monolithic array approach stimulated the development of an algebra of arrays(MoA) *and* index calculus(Psi Calculus)[59, 60]. No known algebras have an associated index calculus, nor are they free of algebraic anamolies. MoA is based on a small set of absolutely essential array operations which are solely defined in terms of dimensionalities, shapes and indexing functions. By application of the rules of this algebra and underlying calculus, complex array expressions can be consequently simplified prior to actually compiling them to code thus avoiding intermediate arrays whenever possible. The rich history of this algebra demonstrates its use throughout the history of computation and when combined with the *lambda calculus* and a universal algebra[70] is an ideal paradigm for reasoning about arrays and subsequently, radar.

3 MoA and the Psi Calculus

The MoA algebra, and its associated *Psi Calculus* , an index calculus, is a formalism that we use to describe array computations on uni- or multi-processor topologies. It is centered around a generalized array indexing function, *psi*, that selects a partition of an array described by a partial index. All array operations in this theory are defined using the indexing function.

There have been several investigations into a mathematics to describe array computations. These include More's theory of arrays (AT)[50], a formal description of AT's nested arrays in a first order logic[41] and the development of a Mathematics of Arrays (MoA) [59]. In [42] the correspondence between AT and MoA is described.

The algebra of MoA denotes a core set of operations proven useful for representing algorithms in scientific disciplines. Unlike other theories about arrays[64, 50], all operations in MoA are defined using shapes and the indexing function *psi*(ψ) which, in conjunction with the reduction semantics of the *Psi* Calculus, provides not just transformational properties, but compositional reduction properties which produce an optimal normal form possessing the Church-Rosser Property[5]. Often transformations require pattern matching and knowledge of equivalence preserving transformations while *psi reductions* are deterministic, hence mechanical.

MoA builds on Iverson's concepts and extends the transformational properties of his algebra while removing all anamolies. The *Psi Calculus* and MoA put closure on concepts introduced by Abrams[1] to optimize the evaluation of monolithic array expressions based on an algebra of indexing.

Our goal is to optimize array computation given a linear address space. The first level of optimization, termed *a denotational normal form*(DNF), is a minimal semantic form expressed in terms of selections using Cartesian coordinates. The normal form can be *built* in numerous ways. We describe a constructive way to build it by using the *Psi Calculus*.

The second level of optimization is to transform the functional normal form to its equivalent *operational normal form*(ONF) or description for implementation which describes the result in terms of starts, strides and lengths to select from the linear arrangement of the items.

3.1 Indexing and Shapes

The central operation of MoA is the indexing function

$$p\psi A$$

in which a vector of n integers p is used to select an item of the n-dimensional array A. The operation is generalized to select a partition of A, so that if q has only $k < n$ components then

$$q\psi A$$

is an array of dimensionality $n - k$ and q selects among the possible choices for the first k axes. In MoA zero origin indexing is assumed. For example, if A is the 3 by 5 by 4 array

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \\ 17 & 18 & 19 & 20 \end{bmatrix} \begin{bmatrix} 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \\ 37 & 38 & 39 & 40 \end{bmatrix} \begin{bmatrix} 41 & 42 & 43 & 44 \\ 45 & 46 & 47 & 48 \\ 49 & 50 & 51 & 52 \\ 53 & 54 & 55 & 56 \\ 57 & 58 & 59 & 60 \end{bmatrix}$$

then

$$< 1 > \psi a = \begin{bmatrix} 21 & 22 & 22 & 23 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \\ 37 & 38 & 39 & 40 \end{bmatrix}$$

$$< 2\ 1 > \psi a \;=\; < \;45\;\; 46\;\; 47\;\; 48\; >$$

$$< 2\ 1\ 3 > \psi a \;=\; 48$$

Most of the common array manipulation operations found in languages like Fortran90, Matlab, ZPL, etc., can be defined from ψ and a few elementary vector operations.

We now introduce notation to permit us to define ψ formally and to develop the *Psi Correspondence Theorem*[52], which is central to the effective exploitation of MoA in array computations. We will use $A, B, ...$

to denote an array of numbers (integers or reals). An array's dimensionality will be denoted by d_A and will be assumed to be n if not specified.

The shape of an array A, denoted by s_A, is a vector of integers of length d_A, each item giving the length of the corresponding axis. The total number of items in an array, denoted by t_A, is equal to the product of the items of the shape. The subscripts will be omitted in contexts where the meaning is obvious.

An index is a vector of n integers that describes one position in an n-dimensional array. Each item of an index for A is less than the corresponding item of s_A. There are precisely t_A indices for an array. A partial index of A is a vector of $0 \leq k \leq n$ integers with each item less than the corresponding item of s_A.

We will use a tuple notation (omitting commas) to describe vectors of a fixed length. For example,

$$< i\ j\ k >$$

denotes a vector of length three. $<>$ will denote the empty vector.

For every n-dimensional array A, there is a vector of the items of A, which we denote by the corresponding lower case letter, here a. The length of the vector of items is t_A. A vector is itself a one-dimensional array, whose shape is the one-item vector holding the length. Thus, for a, the vector of items of A, the shape of a is

$$s_a = < t_A >$$

and the number of items or total number of components a[1] is

$$t_a = t_A.$$

The precise mapping of A to a is determined by a one-to-one ordering function, *gamma*. Although the choice of ordering is arbitrary, it is essential in the following that a specific one be assumed. By convention we assume the items of A are placed in a according to the lexicographic ordering of the indices of A. This is often referred to as *row major ordering*. Many programming languages lay out the items of multidimensional arrays in memory in a contiguous segment using this ordering. Fortran uses the ordering corresponding to a transposed array in which the axes are reversed,*column major*.

Scalars are introduced as arrays with an empty shape vector.

There are two equivalent ways of describing an array A:

1) by its shape and the vector of items, i.e.
$A = \{s_A, a\}$, or

2) by its shape and a function that defines the value at every index p.

These two forms have been shown to be formally equivalent in a first order theory of arrays[40]. We wish to use the second form in defining functions on multidimensional arrays using their Cartesian coordinates (indices). The first form is used in describing address manipulations to achieve effective computation.

To complete our notational conventions, we assume that $p, q, ...$ will be used to denote indices or partial indices and that $u, v, ..$ will be used to denote arbitrary vectors of integers. In order to describe the i_{th} item of a vector a, either a_i or $a[i]$ will be used. If u is a vector of k integers all less than t_A, then $a[u]$ will denote the vector of length k, whose items are the items of a at positions u_j, $j = 0, ..., k-1$.

Before presenting the formal definition of the ψ indexing function we define a few functions on vectors:

$u \mathbin{+\!\!+} v$	catentation of vectors u and v
$u + v$	itemwise vector addition assuming $t_u = t_v$
$u * v$	itemwise vector multiplication
$n + u$, $u + n$	addition of a scalar to each item of a vector
$n * u$, $u * n$	multiplication of each item of a vector by a scalar
$\iota\, n$	the vector of the first n integers starting from 0[2]
$\pi\, v$	a scalar which is the product of the components of v
$k\ \triangle\ u$	when $k \geq 0$ the vector of the first k items of u,*take*
	and when $k < 0$ the vector of the last k items of u
$k\ \bigtriangledown\ u$	when $k \geq 0$ the vector of $t_u - k$ last items of u, *drop*
	and when $k < 0$ the vector of the first $t_u - \lvert k\rvert$ items of u
$k\ \theta\ u$	when $k \geq 0$ the vector of $(k \bigtriangledown u) \mathbin{+\!\!+} (k \triangle u)$
	and when $k < 0$ the vector or $(k \triangle u) \mathbin{+\!\!+} (k \bigtriangledown u)$

[1]We also use $\tau a, \delta a$, and ρa to denote total number of components, dimensionality and shape of a.

Definition 3.1 *Let A be an n-dimensional array and p a vector of integers. If p is an index of A,*

$$p\psi A = a[\gamma(s_A, p)],$$

where

$$\begin{aligned}\gamma(s_A,p) &= x_{n-1} \quad \textit{defined by the recurrence}\\ x_0 &= p_0,\\ x_j &= x_{j-1} * s_j + p_j, \quad j = 1, ..., n-1.\end{aligned}$$

If p is partial index of length $k < n$,

$$p\psi A = B$$

where the shape of B is

$$s_B = k \bigtriangledown s_A,$$

and for every index q of B,

$$q\psi B = (p +\!\!+ q)\psi A$$

The definition uses the second form of specifying an array to define the result of a partial index. For the index case, the function $\gamma(s,p)$ is used to convert an index p to an integer giving the location of the corresponding item in the row major order list of items of an array of shape s. The recurrence computation for γ is the one used in most compilers for converting an index to a memory address[45].

Corollary 3.1 $<> \psi\ A = A$.

The following theorem shows that a ψ selection with a partial index can be expressed as a composition of ψ selections.

Theorem 3.1 *Let A be an n-dimensional array and p a partial index so that $p = q +\!\!+ r$. Then*

$$p\psi A = r\psi(q\psi A).$$

Proof: The proof is a consequence of the fact that for vectors u,v,w

$$(u +\!\!+ v) +\!\!+ w = u +\!\!+ (v +\!\!+ w).$$

If we extend p to a full index by $p +\!\!+ p'$, then

$$\begin{aligned}p'\psi(p\psi A) &= (p +\!\!+ p')\psi A\\ &= ((q +\!\!+ r) +\!\!+ p')\psi A\\ &= (q +\!\!+ (r +\!\!+ p'))\psi A\\ &= (r +\!\!+ p')\psi(q\psi A)\\ &= r\psi(q\psi A)\end{aligned}$$

which completes the proof.

We can now use *psi* to define other operations on arrays. For example, consider definitions of *take* and *drop* for multidimensional arrays.

Definition 3.2 (take: $\triangle$) *Let A be an n-dimensional array, and k a non-negative integer such that $0 \leq k < s_0$. Then*

$$k \triangle A = B$$

where

$$s_B = < k > +\!\!+ (1 \bigtriangledown s_A)$$

and for every index p of B,

$$p\psi B = p\psi A.$$

(In MoA $\triangle$ is also defined for negative integers and is generalized to any vector u with its absolute value vector a partial index of A. The details are omitted here.)

Definition 3.3 (reverse: Φ) *Let A be an n-dimensional array. Then*

$$s_{\Phi A} = s_A$$

and for every integer i, $0 \leq i < s_0$,

$$< i > \psi \Phi A =< s_0 - i - 1 > \psi A.$$

This definition of Φ does a reversal of the 0th axis of A. Note also that all operations are over the 0th axis. The operator Ω[59] extends operations over all other dimensions.

Example

Consider the evaluation of the expression

$$< 2 \;\; 3 > \psi(2 \triangle \Phi A) \tag{1}$$

where A is the array given in the previous section. The shape of the result is

$$\begin{aligned}
& 2 \bigtriangledown s_{2 \triangle \Phi A} \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ (1 \bigtriangledown s_{\Phi A})) \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ (1 \bigtriangledown s_A)) \\
= \;& 2 \bigtriangledown (< 2 > +\!\!+ < 5 \; 4 >) \\
= \;& 2 \bigtriangledown < 3 \; 5 \; 4 > \\
= \;& < 4 > .
\end{aligned}$$

The expression can be simplified using the definitions:

$$\begin{aligned}
& < 2 \; 3 > \psi(2 \triangle \Phi A) \\
= \;& < 2 \; 3 > \psi \Phi A \\
= \;& < 3 > \psi(< 2 > \psi \Phi A) \\
= \;& < 3 > \psi(< 3 - 2 - 1 > \psi A) \\
= \;& < 0 \; 3 > \psi A
\end{aligned} \tag{2}$$

This process of simplifying the expression for the item in terms of its Cartesian coordinates is called *Psi Reduction.* The operations of MoA have been designed so that all expressions can be reduced to a minimal normal form[59].

3.2 Mapping Arrays to Processors

Much work has been done in describing architectures in terms of abstract models and then using the abstract model as the basis for mapping decisions[61]. However, it is still a primarily manual effort to design the mapping of the computation to the abstract model. The automation of this step is crucial if we are to make effective use of parallel architectures.

The abstract model for the organization of the processors is often in the form of a graph whereby the nodes of the graph are processors and the edges are communication links. For many practical models the graph can be represented by an array in which each processor is given an address and each processor to which a link is available is at an address one away along one of the axes. The array model can describe a list of processors, a 2 dimensional mesh, a hypercube, a balanced tree[6], or a network of workstations[16, 61, 15]. Our approach will be to view an architecture as having two array organizations, one which is the abstract model best suited for describing the problem and a second which corresponds to an enumeration of the processors as a list. For an actual architecture the latter corresponds to the list of processor identity numbers and is used to determine the actual send and receive instructions issued by the resulting program.

In doing the mapping from the data arrays of the problem to the array-like arrangements of processors, there is a need to be able to systematically determine what information to distribute to which processors. Having made those decisions, the high level algorithm expressed in terms of array operations has to be turned into low level code that selects data elements from one-dimensional memory, sends it to the appropriate processor in the one-dimensional list of processors. Each processor has to be supplied with code that is parameterized so that it operates on the data in its local memory to carry out its portion of the parallel algorithm.

The difficulties are compounded when a problem is so large that it must be attacked in slices. Thus, a vector of length $k = m * n * p$, where m is the number of slices, n is the amount of data each processor can process in one go, and p is the number of processors, can be viewed algorithmically as a 3-dimensional array of shape m by n by p, where the first axis indicates slices of work to be done one after the other.

The manipulations of the data addresses to ensure that the problem decompositions are handled correctly can be quite intricate and are difficult to get right by hand. Thus, having a formal technique for deriving the address computations, one that can be automated to a large extent, is essential if rapid progress is going to be made in exploiting parallel hardware for scientific computation.

We can also use the idea of mapping Cartesian coordinates to their lexicographic ordering when we want to partition and map arrays to a multiprocessor topology in a portable, scalable way. Consider, for example, a parallel vector-matrix multiply of vector A and matrix B, where $s_A =< n >$ and $s_B =< np >$. One effective way to organize the computation is to map each of the rows of B to a processor, send the elements of A to the corresponding processors, do an integer-vector multiply to form vectors in each processor, and then add the vectors pointwise to produce the result. The last step involves the adding together of n vectors and is best done by adding pairs of vectors in parallel. Abstractly, this last step can be seen as adding together the rows of a matrix pointwise. A hypercube topology is ideal for such a computation and at best would take $\mathcal{O}(\log n)$ on n processors to compute.

Often a hypercube topology is not available, we may only have a LAN of workstations or a linear list of processors. But, we can view any processor topology abstractly as a hypercube and map the rows to processors by imposing an ordering on the p available processors. That is we look at p_i where $0 \leq i < p$ as the lexicographically ordered items of the hypercube.

Hence, in the case of the LAN we obtain a vector of socket addresses. We then abstractly restructure the vector of addresses as a k-dimensional hypercube where $k = \lceil \log_2 n \rceil$. In order to map the matrix to the abstract hypercube we restructure the matrix into a 3-dimensional array such that there is a 1-1 correspondence between the restructured array's planes and the available processors. That is, we send the i^{th} plane of the restructured array to the i^{th} processor lexicographically. If there are more rows than processors then the planes are sequentially reduced within each processor in parallel.

We can apply the *Psi Correspondence Theorem*[52] to the data to see how to address the i^{th} planes from memory efficiently. The same methodology can be applied to address the processors effectively.

For example, suppose we want to add up the rows of a 256 by 512 matrix and we have 8 workstations connected by a LAN. We would restructure the matrix into a 8 by 32 by 512 array which we denote by $A^{'}$. The socket address of the workstations are put into a matrix P and each $< i > \psi A^{'}, i = 1, ..., 8$ is sent to the processor addressed by P_i. The sum of the rows for each plane are formed in parallel producing 8 vectors of length 512 in each processor.

We then restructure P into a 3-d hypercube implicitly and use this arrangement to decide how to perform the access and subsequent addition between the processors. In the first step we add processor plane 1 to plane 0. By the *Psi Correspondence Theorem* this implies adding the contents of processors 4 to 7 to those of processors 0 to 3. In the next step we add processor row 1 to row 0, which implies the contents of processors 2 and 3 are added to those of processors 0 and 1. Finally, we add the contents of processor 1 to the contents of processor 0. Thus, we have added up all the rows in $\log_2 8$ or 3 steps.

The method can be employed for any size matrix and can utilize an arbitrary number of homogeneous workstations connected by a LAN. It is a portable scalable design. A more detailed description of this technique (including timing results) can be found in [61]. We ported and scaled these designs to a 32 processor CM5[15]. We used a similar approach, a linear processor array, to map a parallel sparse LU Decomposition to a network of RS6000s[16]. These ideas were later realized in hardware[56].

3.3 MoA and Machine Abstractions

It is recognized that many large scale scientific and engineering problems have massive computational requirements that can utilize the collective power of large numbers of processors working together. The difficulty facing the computer science community is to provide tools to the scientists and engineers that allow them to solve such problems on large networks of workstations or specific parallel architectures in an effective and timely manner.

The approach we are espousing is that a formal methodology be developed that assists in automating the translation of high level descriptions of computationally intensive problems working from a high level array-based description of the problem. Here, we discuss a specific formalism, the *Psi Calculus*, and demonstrate how by a combination of using *Psi Reduction* and the *Psi Correspondence Theorem*, progress has been made on both the problem of expressing problems at a high level, and on using the formalism both to generate low level code, and to assist in the organization of the computation on a processor network.

Our goal is to prototype tools that can be adopted by compiler, pre-processor, and library designers to assist in the problem of parallelizing programs. We see this happening in three directions:

- by a language providing a high level notation embedded within a standard language, e.g. HPF, ZPL, Fortran90, that is preprocessed using the techniques described in the paper, or
- by the compiler extracting a high level description based on loop analysis and then using the techniques to achieve an effective translation.
- by a language that supports classes, functions, and templates, thus providing compile and run time polymorphisms for compiler pre-processing.

The next step towards our goal is to apply the methodology to a problem of practical size and complexity to demonstrate that significant scientific and engineering computations can be solved effectively in this manner; *real-time radar and signal processing.*

4 Time-Domain Convolution

Over the last 5 decades, the synthetic aperture radar(SAR) has been developed as a unique imaging instrument with high resolution, day/night and all weather operation capabilities[83]. As a result, the SAR has been used in a wide variety of applications, including target detection, continuously observing dynamic phenomena: seismic movement, ocean currents, sea ice motion and classification of vegetation. In comparison with the spectral analysis(FFT) and frequency domain convolution, the time-domain(TD) analysis has been introduced and has become the simplest and most accurate algorithm for SAR signal processing. As the time-modulated wave transmission and receiving by SAR, the TD algorithm directly processes the signal echo by using the matched filters without approximation. However, the TD algorithm is also the most computationally intensive, thus it can only be used for size-limited SAR data. As the requirement of large-size and high-resolution SAR imagery increases, the investigation and development of time-domain schemes are conducted with respect to a fast computational algorithm to implement the dime-domain analysis. What follows is a MoA design and derivation of the Time-Domain Convolution. Some MoA operations defined by *psi* are found in the Appendix.

TD Convolution: MoA Design and Derivation

Assume:

1. We want to perform the convolution of $\vec{\mathcal{X}}$, length $\tau\vec{\mathcal{X}}$, with $\vec{\mathcal{Y}}$, length $\tau\vec{\mathcal{Y}}$.
2. $(\tau\vec{\mathcal{X}}) \geq (\tau\vec{\mathcal{Y}})$
3. $(\tau\vec{\mathcal{Y}}) \mod \mathit{cache} = 0$
4. $(\tau\vec{\mathcal{Y}}) \geq \mathit{cache}$

Example:

1. $\vec{\mathcal{X}} \equiv <\ 1\ 2\ 3\ 4\ 5\ >$
2. $\vec{\mathcal{Y}} \equiv <\ 6\ 7\ 8\ >$
3. Let $ty \equiv (\tau\vec{\mathcal{Y}}) - 1 \equiv 3 - 1 \equiv 2$

4. Let $tz \equiv (\tau\vec{\mathcal{Y}}) + (\tau\vec{\mathcal{X}}) - 1 \equiv 3 + 5 - 1 \equiv 7$
5. Let $ne\vec{w}x \equiv (< ty > \ \widehat{\rho}\ 0) +\!\!+ \vec{\mathcal{X}} +\!\!+ (< ty > \ \widehat{\rho}\ 0) \equiv <\ 0\ 0\ 1\ 2\ 3\ 4\ 5\ 0\ 0\ >$. Consequently, $\rho\ ne\vec{w}x \equiv < ty + (\tau\vec{\mathcal{X}}) + ty > \equiv < 2 \times ty + (\tau\vec{\mathcal{X}}) >$

I Shift $ne\vec{w}x\ \ tz$ times: by 0, by 1, ... by $tz - 1$.

$$(\iota\ \ tz)_\theta\Omega_{<0\ 1>} <\ 0\,0\,1\,2\,3\,4\,5\,0\,0\ >$$

$$\equiv \begin{bmatrix} 0 & 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 & 0 \\ 1 & 2 & 3 & 4 & 5 & 0 & 0 & 0 & 0 \\ \vdots \\ 5 & 0 & 0 & 0 & 0 & 1 & 2 & 3 & 4 \end{bmatrix}$$

II. From each row take the necessary $\tau\vec{\mathcal{Y}}$ pieces.

$$(\tau\vec{\mathcal{Y}})\ \triangle\ (\iota\ \ tz)_\theta\Omega_{<0\ 1>} <\ 0\,0\,1\,2\,3\,4\,5\,0\,0\ >$$

$$\equiv \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 3 \\ \vdots \\ 5 & 0 & 0 \end{bmatrix}$$

III. Perform the multiplication and addition with $\vec{\mathcal{Y}}$.

$$+_{\text{red}}\Omega_{<1>}((\phi\vec{\mathcal{Y}})_\times\Omega_{<1\ 1>}((\rho\vec{\mathcal{Y}})\ \triangle\ \Omega_{<1\ 1>}((\iota\ \ tz)_\theta\Omega_{<0\ 1>}ne\vec{w}x)))$$

$$\equiv \begin{matrix} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{matrix} \begin{bmatrix} 8 & 7 & 6 \\ 8 & 7 & 6 \\ 8 & 7 & 6 \\ \vdots \\ 8 & 7 & 6 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 3 \\ \vdots \\ 5 & 0 & 0 \end{bmatrix}$$

IV. Removing all constants, i.e. the general expression to reduce:

$$+_{\text{red}}\Omega_{<(\delta\vec{\mathcal{Y}})>}((\phi\vec{\mathcal{Y}})_\times\Omega_{<(\delta\vec{\mathcal{Y}})+(\delta\vec{\mathcal{Y}})>}((\rho\vec{\mathcal{Y}})\ \triangle\ \Omega_{<(\delta\vec{\mathcal{Y}})+(\delta\vec{\mathcal{Y}})>}((\iota\ \ tz)_\theta\Omega_{<(\delta\ \ ne\vec{w}x)-1+(\delta\ \ ne\vec{w}x)>}ne\vec{w}x)))$$

V. Now ψ-reduce the expression in IV.

Let:

1. $B \equiv ((\iota\ \ tz)_\theta\Omega_{<0\ 1>}ne\vec{w}x)$
2. $A \equiv (\rho\vec{\mathcal{Y}})\ \triangle\ \Omega_{<1\ 1>}B$
3. $X \equiv ((\phi\vec{\mathcal{Y}})_\times\Omega_{<1\ 1>}A)$

V.I Reduce $ne\vec{w}x$ so that it can be used in the derivation of IV.

i. Obtain shape:

$$(\rho\ \ ne\vec{w}x) \equiv ty + (\tau\vec{\mathcal{X}}) + ty \equiv 2ty + (\tau\vec{\mathcal{X}}) \tag{3}$$

ii. Determine components from shape:

$$\forall\ \ i, 0 \le i < (2ty + (\tau\vec{\mathcal{X}}))$$

$$\begin{aligned} < i > \psi ne\vec{w}x \ &\equiv\ 0 \text{ if } 0 \le i < ty && (4)\\ &\equiv\ < i - ty > \psi\vec{\mathcal{X}} \text{ if } ty \le i < (ty + (\tau\vec{\mathcal{X}})) && (5)\\ &\equiv\ 0 \text{ if } ((\tau\vec{\mathcal{X}}) + ty) \le i < (2ty + (\tau\vec{\mathcal{X}})) && (6)\end{aligned}$$

VI. Recall: $A \equiv (\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B$.

Apply definitions for Ω[59]:

Let:

1. $m \equiv (\delta\xi_l) - \sigma_l \lfloor (\delta\xi_r) - \sigma_r \equiv 1 - 1 \lfloor 2 - 1 \equiv 0 \lfloor 1 \equiv 0$
2. $\vec{x} \equiv \Theta \Longrightarrow \vec{k} \equiv \Theta$
3. $\vec{u} \equiv \Theta \Longrightarrow \vec{i} \equiv \Theta$
4. $\vec{v} \equiv < (\rho A)[0] > \Longrightarrow 0 \leq j < (\rho A)[0]$

Consequently:

$$\begin{aligned}
(\rho\xi_l) &\equiv \vec{u} +\!\!+ \vec{x} +\!\!+ \vec{y} \equiv \Theta +\!\!+ \Theta +\!\!+ \vec{y} \equiv (\rho\vec{\mathcal{Y}}) && (7)\\
&\equiv \text{ where } \vec{y} \equiv (\rho\vec{\mathcal{Y}}) && (8)\\
(\rho\xi_r) &\equiv \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{z} \equiv (\rho A[0]) +\!\!+ \Theta +\!\!+ (\rho A)[1] \equiv (\rho A) && (9)\\
&\equiv \text{ where } \vec{z} \equiv (\rho A)[1] && (10)
\end{aligned}$$

Therefore:

$$\begin{aligned}
\rho A &\equiv \vec{u} +\!\!+ \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{w} && (11)\\
& \quad \text{where } \vec{w} \equiv \rho((\phi\vec{\mathcal{Y}}) \times (< j > \psi A)) \equiv \rho(\phi\vec{\mathcal{Y}}) \equiv (\rho\vec{\mathcal{Y}})\\
& \quad \forall\ j, 0 \leq j < tz\\
&\equiv \Theta +\!\!+ (\rho A)[0] +\!\!+ \Theta +\!\!+ (\rho\vec{\mathcal{Y}}) && (12)\\
&\equiv (\rho A)[0] +\!\!+ (\rho A)[1] \equiv (\rho A) && (13)
\end{aligned}$$

Putting this together, recalling that: $\rho B \equiv < tz +\!\!+ (\rho\vec{\mathcal{Y}}) >$, and $\forall\ j, \ni 0 \leq j < tz$

$$(\phi\vec{\mathcal{Y}}) \times (< j > \psi A) \quad \equiv \quad (\phi\vec{\mathcal{Y}}) \times (< j > \psi((\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B)) \qquad (14)$$

VII. Apply Ω again.

VII.I Determine the shape:

$$\begin{aligned}
\rho((\phi\vec{\mathcal{Y}}) \times (< j > \psi((\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B))) &\equiv \rho(\phi\vec{\mathcal{Y}}) \equiv \rho\vec{\mathcal{Y}} && (15)\\
& && (16)
\end{aligned}$$

VII.II Obtain the indices: $\forall\ i' \ni 0 \leq i' < (\rho\vec{\mathcal{Y}})[0] >$

$$\begin{aligned}
&(< i' > \psi(\phi\vec{\mathcal{Y}})) + (< j > \psi(\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B) && (17)\\
\equiv\ & (< i' > \psi(\phi\vec{\mathcal{Y}})) + (< i' > \psi < j > \psi(\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B) && (18)\\
\equiv\ & < (\rho\vec{\mathcal{Y}})[0] - (i' + 1) > +(< j\ i' > \psi(\rho\vec{\mathcal{Y}}) \bigtriangleup \Omega_{<1\ 1>}B) && (19)
\end{aligned}$$

VIII. Apply Ω again. Let:

1. $m \equiv (\delta\xi_l) - \sigma_l \lfloor (\delta\xi_r) - \sigma_r \equiv 1 - 1 \lfloor 2 - 1 \equiv 0 \lfloor 1 \equiv 0$
2. $\vec{x} \equiv \Theta \Longrightarrow \vec{k}'' \equiv \Theta$
3. $\vec{u} \equiv \Theta \Longrightarrow \vec{i}'' \equiv \Theta$
4. $\vec{v} \equiv < tz > \Longrightarrow 0 \leq j'' < tz$

Consequently:

$$\begin{aligned}
(\rho\xi_l) &\equiv \vec{u} +\!\!+ \vec{x} +\!\!+ \vec{y} \equiv \Theta +\!\!+ \Theta +\!\!+ \vec{y} \equiv (\rho\vec{\mathcal{Y}}) && (20)\\
&\equiv \text{ where } \vec{y} \equiv (\rho\vec{\mathcal{Y}})\\
(\rho\xi_r) &\equiv \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{z} \equiv tz +\!\!+ \Theta +\!\!+ (\rho B)[1] && (21)\\
& \quad \text{where } \vec{z} \equiv (\rho B)[1]
\end{aligned}$$

Therefore, the shape of the Ω expression is:

$$\vec{u} +\!\!+ \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{w} \equiv < tz +\!\!+ (\rho\vec{\mathcal{Y}}) > \tag{22}$$

where

$$\vec{w} \equiv \rho((\rho\vec{\mathcal{Y}})\ \triangle\ (< j^{''} > \psi B)) \equiv (\rho\vec{\mathcal{Y}}) \tag{23}$$

Completing the reduction, we recall that: $0 \leq j < tz$ and $0 \leq i^{'} < (\tau\vec{\mathcal{Y}})$ and that

$$< j\ i^{'} > \psi(\rho\vec{\mathcal{Y}})\ {}_{\triangle}\Omega_{<1\ 1>}B$$

$$< i^{''} > \psi((\rho\vec{\mathcal{Y}})\ \triangle\ (< j^{''} > \psi B)) \quad \equiv \quad (< i^{'} > \psi < j^{''} > \psi B) \tag{24}$$

Note that $i^{''}$ ranges through the components of $\vec{\mathcal{Y}}$ and $j^{''}$ permutes $ne\vec{w}x\ tz$ times. With all this we have,

$$\sum_{i^{'}=0}^{(\tau\vec{\mathcal{Y}})}(< (\rho\vec{\mathcal{Y}})[0] - (i^{'}+1) > \psi\vec{\mathcal{Y}}) \times < i^{'} > \psi(< j^{''} > \psi B) \tag{25}$$

$$\equiv \sum_{i^{'}=0}^{(\tau\vec{\mathcal{Y}})}(< (\rho\vec{\mathcal{Y}})[0] - (i^{'}+1) > \psi\vec{\mathcal{Y}}) \times (< j^{''} > \psi B) \tag{26}$$

Now reduce B where

VIV $B \equiv ((\iota\ \ tz)_{\theta}\Omega_{<0\ 1>}ne\vec{w}x)$

Let:

1. $m \equiv 0$
2. $\vec{x} \equiv \Theta \Longrightarrow \vec{k}^{'''} \equiv \Theta$
3. $\vec{u} \equiv \Theta \Longrightarrow 0 \leq \vec{i}^{'''} < tz$
4. $\vec{v} \equiv < tz > \Longrightarrow \vec{j}^{'''} \equiv \Theta$

Consequently:

$$\begin{aligned}(\rho\xi_l) &\equiv \vec{u} +\!\!+ \vec{x} +\!\!+ \vec{y} \equiv < tz > && (27)\\ &\equiv \text{ where } \vec{y} \equiv \Theta \\ (\rho\xi_r) &\equiv \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{z} \equiv (\rho\ \ ne\vec{w}x) && (28)\\ & \quad \text{where } \vec{z} \equiv (\rho\ \ ne\vec{w}x)\end{aligned}$$

Therefore, the shape of the Ω expression is:

$$\vec{u} +\!\!+ \vec{v} +\!\!+ \vec{x} +\!\!+ \vec{w} \equiv < tz > +\!\!+ (\rho\ \ ne\vec{w}x) \tag{29}$$

where

$$\begin{aligned}\vec{w} &\equiv \rho(< i^{'''} > \psi((\iota\ \ tz)\phi\ \ ne\vec{w}x) && (30)\\ &\equiv \rho(i^{'''}\phi\ \ ne\vec{w}x) \equiv \rho\ \ ne\vec{w}x && (31)\end{aligned}$$

Rewriting the derivation of B:

$$\sum_{i^{'}=0}^{(\tau\vec{\mathcal{Y}})}(< (\rho\vec{\mathcal{Y}})[0] - (i^{'}+1) > \psi\vec{\mathcal{Y}}) \times (< i^{'} > \psi(< j^{''} > \psi B)) \tag{32}$$

$$\equiv \sum_{i'=0}^{(\tau\vec{\mathcal{Y}})} (<(\rho\vec{\mathcal{Y}})[0] - (i' + 1) > \psi\vec{\mathcal{Y}}) \times (< i' > \psi(i''' \phi ne\vec{w}x)) \tag{33}$$

$$\equiv \sum_{i'=0}^{(\tau\vec{\mathcal{Y}})} (<(\rho\vec{\mathcal{Y}})[0] - (i' + 1) > \psi\vec{\mathcal{Y}}) \times (< i' + j'' > \bmod tz > \psi ne\vec{w}x) \tag{34}$$

$$\textbf{qed} \tag{35}$$

Note that $0 \leq i' < (\tau\vec{\mathcal{Y}})$ and both i''' and j'' have the same bounds. This completes the derivation for the convolution algorithm on a sequential processor and is the generic design of how to build the code.

4.1 Adding Processors

Recall how the computation is performed on a uniprocessor[3]

$$+_{\text{red}}\Omega_{<1>}((\phi\vec{\mathcal{Y}})_\times \Omega_{<1\ 1>}((\rho\vec{\mathcal{Y}})\ \triangle\ \Omega_{<1\ 1>}((\iota\ \ tz)_\theta \Omega_{<0\ 1>} ne\vec{w}x)))$$

$$\equiv \begin{array}{c} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{array} \begin{bmatrix} 8 & 7 & 6 \\ 8 & 7 & 6 \\ 8 & 7 & 6 \\ \vdots & & \\ 8 & 7 & 6 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 2 \\ 1 & 2 & 3 \\ \vdots & & \\ 5 & 0 & 0 \end{bmatrix}$$

An analysis of the data flow shows that the best way to decompose the matrix computation is over the primary axis since breaking up the problem over rows creates no communication. If we break up over columns, $\frac{1}{2}\vec{\mathcal{Y}}$ must be used in each section AND the addition would be over multiple processors. That is, the tz dimension is broken into 2 parts(2 procs[4]) which lifts the dimensionality of the problem one dimension. That is

$$< tz > \mapsto < p +\!\!+ \frac{tz}{p} >$$

If communication is necessary, it is best to do it at lower, faster levels of the memory hierarchy.

Suppose now that $(\tau\vec{\mathcal{Y}}) \equiv 9$ and $(\tau\vec{\mathcal{X}}) \equiv 6$ where $\vec{\mathcal{Y}} \equiv < 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9 >$ and $\vec{\mathcal{X}} \equiv < 10\ 11\ 12\ 13\ 14\ 15 >$ and there are two processors. In this case $tz = ((\tau\vec{\mathcal{Y}}) - 1) + (\tau\vec{\mathcal{X}}) \equiv 5 + 9 \equiv 14$. Consequently, each processor would process 7 rows.

$$\begin{array}{c} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{array} \begin{bmatrix} 15 & 14 & 13 & 12 & 11 & 10 \\ 15 & 14 & 13 & 12 & 11 & 10 \\ \vdots & & & & & \\ 15 & 14 & 13 & 12 & 11 & 10 \end{bmatrix} \times \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 2 \\ \vdots & & & & & \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \mapsto \textbf{processor 0}$$

$$\begin{array}{c} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{array} \begin{bmatrix} 15 & 14 & 13 & 12 & 11 & 10 \\ 15 & 14 & 13 & 12 & 11 & 10 \\ \vdots & & & & & \\ 15 & 14 & 13 & 12 & 11 & 10 \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 4 & 5 & 6 & 7 & 8 \\ \vdots & & & & & \\ 9 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \mapsto \textbf{processor 1}$$

This decomposition implies we must reshape A thus adding a processor loop to our original design. $(\rho A)[0]$ is broken into two pieces and becomes $(p +\!\!+ \lceil\frac{(\rho A)[0]}{p})\rceil$ We reshape as follows:

$$+_{\text{red}}\Omega_{<1>}((\phi\vec{\mathcal{Y}})_\times \Omega_{<1\ 1>} A$$

[3]We will later see that processors, caches, etc., are simply abstract memory levels ordered by speed.

[4]For illustration we use $p \equiv 2$. The design is general for $p \equiv n$. Notice in the design we concatenate 0 to A. In general we'd need to concatenate $n - 1$ zeros (through a reshape so as to use only one).

becomes

$$+_{\mathrm{red}}\Omega_{<1>}((\phi\vec{\mathcal{Y}})_{\times}\Omega_{<1\ 1>}((p +\!\!+ \lceil\frac{(\rho A)[0]}{p}\rceil) +\!\!+ (\rho A)[1])\ \widehat{\rho}\ (A +\!\!+ 0)$$

where $A \equiv (\rho\vec{\mathcal{Y}})\ \triangle\ \Omega_{<1\ 1>}B$ and $B \equiv ((\iota\ \ tz)_{\theta}\Omega_{<0\ 1>}ne\vec{w}x)$ noting that $(\rho A)[0] \equiv tz$. There is no need to re-derive the ONF we simply recognize how γ is used to partition $(\rho A)[0]$ into two pieces.

4.2 Adding Cache

If we now want to add a cache loop we must also break $(\rho A)[1]$ into two pieces. Consequently, we add another dimension. Thus, we started with a 1-d problem, abstracted the computation to a 2-d time dimension, adding processors went to 3-d, adding a cache; 4-d. Suppose $cache = 3$. Adding cache[5] to the expressions above requires that we break $(\rho A)[1]$ into two pieces as we did for $(\rho A)[0]$. We reshape again as follows:

$$+_{\mathrm{red}}\Omega_{<1>}(\phi\vec{\mathcal{Y}})_{\times}\Omega_{<1\ 1>}((p +\!\!+ \lceil\frac{(\rho A)[0]}{p}\rceil) +\!\!+ (\rho A)[1])\ \widehat{\rho}\ (A +\!\!+ 0)$$

becomes

$$+_{\mathrm{red}}+_{\mathrm{red}}\Omega_{<1>}(\frac{(\rho A)[1]}{cache} +\!\!+ cache)\ \widehat{\rho}\ (\phi\vec{\mathcal{Y}}))_{\times}\Omega_{<1\ 1>}((p +\!\!+ \lceil\frac{(\rho A)[0]}{p}\rceil) +\!\!+ \frac{(\rho A)[1]}{cache} +\!\!+ cache)\ \widehat{\rho}\ (A +\!\!+ 0)$$

Pictorially is:

$$\begin{array}{l} \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{array} \left[\begin{array}{c} \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \\ \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \end{array}\right] \times \left[\begin{array}{c} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \\ \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 2 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \end{array}\right] \mapsto \textbf{processor 0}$$

$$\begin{array}{l} \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \\ \vdots \\ \sum_{i=0}^{\frac{(\tau\vec{\mathcal{Y}})-1}{cache}} \sum_{i=0}^{(\tau\vec{\mathcal{Y}})-1} \end{array} \left[\begin{array}{c} \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \\ \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 15 & 14 & 13 \\ 12 & 11 & 10 \end{bmatrix} \end{array}\right] \times \left[\begin{array}{c} \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \\ \begin{bmatrix} 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 9 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{array}\right] \mapsto \textbf{processor 1}$$

Let $\vec{\mathcal{Y}}' \equiv \phi\vec{\mathcal{Y}}$. Consequently, both $\vec{\mathcal{Y}}'$ and A get reshaped, the rows of A by processors, and the columns of A by cache size. Therefore, $\vec{\mathcal{Y}}$ becomes

$$Y^2 \equiv (\frac{\tau\vec{\mathcal{Y}}}{cache} +\!\!+ cache)\ \widehat{\rho}\ \vec{\mathcal{Y}}'$$

Let $\vec{s}_l \equiv \rho Y^2$. Thus, $\forall\ \vec{i} \ni 0 \leq^* \vec{i} <^* \vec{s}_l$ where $\vec{i} \equiv < icache_{row} +\!\!+ icache_{col} >$. Then,

$$\begin{aligned} \vec{i}\psi Y^2 &\equiv (\texttt{rav}\, Y^2)[\gamma(\vec{i};\vec{s}_l) \bmod (\tau Y^2)] && (36) \\ &\equiv \vec{\mathcal{Y}}'[\gamma(\vec{i};\vec{s}_l) \bmod \pi(\frac{\tau\vec{\mathcal{Y}}}{cache} +\!\!+ cache)] && (37) \\ &\equiv \vec{\mathcal{Y}}'[\gamma(\vec{i};\vec{s}_l) \bmod (\tau\vec{\mathcal{Y}})] && (38) \end{aligned}$$

[5]We assume that *cache* evenly divides (ρA).

Therefore, assuming that $(\tau\vec{\mathcal{Y}}')$ mod $cache \equiv 0$. $\forall\ \vec{i'} \ni 0 \leq \vec{i'} < (\tau\vec{\mathcal{Y}}')$.

$$\begin{aligned} < i_{cache} > \psi Y^2 &\equiv (\texttt{rav}\, Y^2)[(\gamma(< icache_{row} >; \rho Y^2) \times (\pi 1 \bigtriangledown \rho Y^2)) + \iota\ (\pi 1 \bigtriangledown \rho Y^2)] && (39) \\ &\equiv \vec{\mathcal{Y}}'[(icache_{row} \times cache) + \iota\ cache] && (40) \\ &\equiv \vec{\mathcal{Y}}[(\tau\vec{\mathcal{Y}}) - (i' + 1)][(icache_{row} \times cache) + \iota\ cache] && (41) \end{aligned}$$

Therefore,

$$< (\rho\vec{\mathcal{Y}})[0] - (i' + 1) > \psi\vec{\mathcal{Y}} \tag{42}$$

becomes $\forall\ icache_{row} \ni 0 \leq icache_{row} < \frac{(\tau\vec{\mathcal{Y}})}{cache}$

$$< (\tau\vec{\mathcal{Y}}) - ((icache_{row} \times cache) + icache_{col} - 1) > \psi\vec{\mathcal{Y}} \tag{43}$$

where $0 \leq icache_{col} < cache$. Rewriting the entire original expression to incorporate processors and cache we have:

$$((\tau\vec{\mathcal{Y}}) - ((icache_{row} \times cache) + icache_{col} - 1) > \psi\vec{\mathcal{Y}}) \times < p +\!\!+ \lceil \frac{(\rho A)[0]}{p} \rceil +\!\!+ \frac{(\rho A)[1]}{cache} +\!\!+ cache > \widehat{\rho}\ A \tag{44}$$

where

$$A \equiv < (i' + j'') \bmod tz > \psi \vec{newx}$$

That is, $\vec{newx}[(i' + j'') \bmod tz]$ recalling that $0 \leq i' < (\tau\vec{\mathcal{Y}})$ and $0 \leq j'' < tz$. Then, $\forall\ i_0, i_1, i_2, i_3 \ni 0 \leq^* i_0, i_1, i_2, i_3 <^* < p +\!\!+ \frac{(\rho A)[0]}{p} +\!\!+ \frac{(\rho A)[1]}{cache} +\!\!+ cache >$, i.e.

$$\begin{aligned} 0 \leq i_0 &< p \\ 0 \leq i_1 &< \frac{(\rho A)[0]}{p} \\ 0 \leq icache_{row} &< \frac{(\rho A)[1]}{cache} \\ 0 \leq i_3 &< cache \longmapsto \iota\ cache \end{aligned}$$

Rewriting

$$newx[(((\lceil \frac{tz}{o} \rceil \times i_0) + i_1) + (icache_{row} \times cache) + \iota\ cache) \bmod tz] \tag{45}$$

4.3 ONF for TD Convolution

Putting this all together we produce a generic design for processors and cache.

$$\sum_{icache_{row}=0}^{\frac{\tau\vec{\mathcal{Y}}}{cache}-1} \vec{\mathcal{Y}}[((\tau\vec{\mathcal{Y}}) - ((icache_{row} \times cache) + (\iota\ cache)) - 1] \times newx[((((\lceil \frac{tz}{p} \rceil \times i_0) + i_1) + icache_{row} \times cache) + \iota\ cache) \bmod tz]$$

4.3.1 ONF and a Generic Algorithm Specification

The following denotes a generic algorithm specification given the ONF above.

```
1. For $i_0 = 0$ to $p - 1$ do:
2.     For $i_1 = 0$ to $\frac{(\rho A)[0]}{p} - 1$ do:
3.         $sum \leftarrow 0$
4.         For $icache_{row} = 0$ to $\frac{\tau\vec{\mathcal{Y}}}{cache} - 1$ do:
5.             For $i_3 = 0$ to $cache - 1$ do:
6.                 $sum \leftarrow sum +\ \vec{\mathcal{Y}}[((\tau\vec{\mathcal{Y}}) - ((icache_{row} \times cache) + i_3)) - 1]$
                       $\times newx[((((\lceil \frac{tz}{p} \rceil \times i_0) + i_1) + icache_{row} \times cache) + i_3) \bmod tz]$
```

We have mechanized the above without the processor and cache extensions[58].

4.4 Algebraically Predicting Performance on RAW

Characterizing the partitioning of the shape vector as cache and processor loops is arbitrary. If another level of memory is faster, than it would be appropriately named. Having said this, we turn to using our algebraic descriptions, as an engineer's tool to assist in the design and implementation of software to drives their hardware to run at designed speeds. Let's return to our original depiction in Section 4_{III}, with a time dimension. From the picture we see that there are *tz* executions[6] of:

$$\sum_{i=0}^{(\tau\vec{y})-1} < y_2\ y_1\ y_0 > \times < x_j\ x_{j+1}\ x_{j+2} > \tag{47}$$

j an index in x. The above operation(s) could be implemented in numerous ways:

- ideal parallelism: fully pipelined, parallel multiply-add, and parallel IO.
- sequential add, parallel multiply, parallel IO.
- sequential add, sequential multiply, sequential IO.

⋮

-

Consequently, in order to predict performance, the instructions and timings which characterize the above must be defined for the ground case. Thus, implying that an Abstract Machine must do the same. We discuss this in Section 8.

In conjunction with RAW engineers, we started with their performance for an 4-Tap,4-Tile implementations, see Figures 2 and 3. The goal was to determine how to *scale down* and *scale up* this design. The RAW team achieved this implementation by using their network to stream in the input vector, *parallel IO*. They used their fully pipelined multiply-add instruction and pipelined all other assembly instructions. Their first attempts at implementing the TD Convolution using 1-Tile utilized a circular buffer which slowed[35] performance below the scalability factor desired. RAW engineers could not scale this design to their optimum. We could however characterize how this might be possible: If we started with a 4-Tap,4-Tile using parallel IO and a fully pipelined multiply-add, and we wanted our 4Tap, 1-Tile design to use memory, then we could generalize and scale by using half of the memory with half of the network. Again, we would have to know which instructions to use and which memory level to use, all chosen by speed, if we want to predict and verify performance from an algebraic abstraction.

Collaborations with Hoffmann[35, 54] led to a 4-Tap,2-Tile design that mirrored the design in Section 4.2, *adding cache*. Recall the name is arbitrary, e.g. adding *cache* here means *Tile*. Consequently, we can see from our depiction that moving an adder outside will achieve the same performance as the 4-Tap,4-Tile divided by two since we must sequentialize part of the computation, while keeping constant all other components in the design, see Figure 4. Independently, we predict performance for a 4-Tap,1-Tile design based on the same observations, see Figure 5.

RAW engineers have published what they believe to be optimal performance given an N-Tap,N-Tile configuration[19], see Figure 3. to implement the TD Convolution. What they might not have realized is that performance could increase by partitioning the *time dimension*, as described in Section 4.1. Figure 6 illustrates the MoA/Psi prediction that performance will improve if the number of Tiles is doubled, keeping all other things constant. We are awaiting the validation of this as well as our one and two Tile prediction.

5 Modified Gram-Schmidt QR

We now apply our methodology to the modified Gram-Schmidt(MGS) algorithm for QR factorization. We use the MoA framework to add processor and cache loops, reordering memory accesses regularly across separate processors and in time. The end result is a generic parallel implementation of MGS for QR factorization. The implementation accepts as parameters the number of processors and the cache size, and allows for a transparent analysis of the I/O complexity of the algorithm. In comparison, existing parallel algorithms for QR factorization usually rely on dividing a matrix into blocks of a particular size, distributing these blocks

[6]Recall that either (0) one 0 is used due to reshape or (1) a shift occurs due to the awareness of the *noop*.

over available processors [39, 23]. Memory access patterns are controlled indirectly through the block size. Finding the optimal block size can be difficult since the theory to analyze the block(tensor decomposition) analyzes these structures independent of the algorithm. We give an example of its application towards QR factorization, starting with a sequential formulation, and moving towards a formulation accounting for parallel processing and the memory hierarchy. We then use the result of this derivation to give a performance analysis, description of implementation, and preliminary experimental results. We conclude the QR section by discussing ways to use expression templates to assist in providing compiler-style optimizations.

5.1 QR Factorization

Suppose $A = \left[\begin{array}{cccc} \vec{a_0} & \vec{a_1} & \ldots & \vec{a_n} \end{array} \right]$ is an $m \times n$ matrix of maximal rank, where $m > n$. The modified Gram-Schmidt algorithm produces the factorization $A = QR$, where Q is an orthonormal $m \times n$ matrix and R is an $n \times n$ upper triangular matrix.

5.1.1 Notation

Let $A = \left[\begin{array}{cc} \vec{a_0} & A' \end{array} \right]$. Let $R = \left[\begin{array}{cc} r_{00} & \vec{r_0} \\ 0 & R' \end{array} \right]$.

```
Modified Gram-Schmidt:
If $A$ is not empty,
    1. $r_{00} \leftarrow \|\vec{a_0}\|$
    2. $\vec{a_0} \leftarrow \frac{\vec{a_0}}{r_{00}}$
    3. $\vec{r_0} = \vec{a_0}^T A'$
    4. $A' = A' - \vec{a_0} * \vec{r_0}$
    5. Run Modified Gram-Schmidt on $A'$, $R'$
```

At the end of the algorithm, the columns of A have been orthogonalized to form Q and the correct transformation matrix R has been created.

5.2 Derivation of the Algorithm

Each step of the MGS algorithm will first be examined separately. A general sequential implementation will be derived for each step of the modified Gram-Schmidt algorithm. Processor and cache loops will then be added to optimize the implementation's performance on parallel memory-constrained systems.

5.2.1 Variable storage

Assume that arrays are stored as lists of elements in row-major order[7] . After each iteration, one column of A is dropped to form A'. Since elements in a column are scattered throughout the matrix, it is more convenient for our derivation to work with the transpose. Thus, for the remainder of this discussion, we will focus on solving the problem $A^T = QR$. Note that the final implementation can be changed to work with A quite easily by substituting the desired index into A every time a memory access into A^T occurs.

5.2.2 Notation and organization

Assume that A is a two-dimensional matrix of size $m \times n$, $m < n$. Further assume for simplicity that n is evenly divisible by the cache size. [8] Let $\vec{a}$ be the vector containing the elements of A in row major order ($\vec{a} = \texttt{rav}\, A$). Let $\vec{a_i}$ denote the ith row of A. Let R be an $n \times n$ matrix initialized to zero. Let p denote the number of processors, and let *cache* denote the cache size.

[7]Column major and regular sparse layouts are also supported.

[8]This assumption can dropped, but a more extensive set of edge conditions must be applied at the end. It does not change the analysis.

Reference to an example be will helpful in the following derivation. In these examples, A will be a 5×6 matrix with arbitrary entries. The entries of A will be labeled with their offset to make memory access patterns clearer.

$$A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 \end{bmatrix}$$

5.2.3 Vector Norm

We first calculate the Euclidean norm of $\vec{a_0}$.

$$r_{00} \leftarrow \|\vec{a_0}\| \tag{48}$$

The norm is the square-root of the sum of the components squared.

$$r_{00} \leftarrow \sqrt{+_{\text{red}}(<0> \psi A)^2} \tag{49}$$

Expanding the $+_{\text{red}}$ as a sum yields an expression in DNF. r_{00} is a scalar, i.e., $\rho\ r_{00} = <>$.

$$r_{00} \leftarrow \sqrt{\sum_{i=0}^{(\rho A)[1]-1} (<0\ i> \psi A)^2} \tag{50}$$

Reduction to ONF yields an expression containing only arithmetic on elements directly accessed from the list representation of A. Transformation of this expression to code is straightforward.

$$r_{00} \leftarrow \sqrt{\sum_{i=0}^{n-1} \vec{a_0}[i]^2} \tag{51}$$

The previous expression gives a description appropriate for a uniprocessor. In a multiprocessor system, work can be divided among all available processors. Each processor computes a subsum of $\lceil \frac{n}{p} \rceil$ elements. These subsums are then combined to yield a total sum. The vector $\vec{a_0}$ is broken into p pieces along its only dimension, and each piece is assigned to one processor. In effect, the dimensionality of the problem has been lifted by one. This correspondence is made specific by reshaping the vector.

Suppose p=2.

$$\begin{aligned} B &= (p \mathbin{+\!\!+} \lceil \frac{(\rho A)[1]}{p} \rceil)\ \widehat{\rho}\ (<0> \psi A) && (52) \\ &= (2 \mathbin{+\!\!+} \frac{6}{2})\ \widehat{\rho}\ (<0> \psi A) && (53) \\ <0> \psi A &= \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} = B && (54) \end{aligned}$$

Subsums are first computed over all elements in each processor. Then, these subsums are combined over the appropriate dimensions.

$$r_{00} \leftarrow \sqrt{+_{\text{red}}(+_{\text{red}}(B^2))} \tag{55}$$

As before, $+_{\text{red}}$ is rewritten as a sum. r_{00} is a scalar, i.e. $\rho\ r_{00} = <>$.

$$r_{00} \leftarrow \sqrt{\sum_{i_1=0}^{p-1} \sum_{i_2=0}^{\lceil \frac{(\rho A)[1]}{p} - 1 \rceil} \left\langle\ i_1 \quad i_2\ \right\rangle \psi B^2} \tag{56}$$

When reduced to Operational Normal Form:

$$r_{00} \leftarrow \sqrt{\sum_{i_1=0}^{p-1} \sum_{i_2=0}^{\lceil \frac{(\rho A)[1]}{p} - 1 \rceil} \vec{a}_0[i_1 * \lceil \frac{(\rho A)[1]}{p} \rceil + i_2]^2} \tag{57}$$

The minimum functions are necessary to take care of end conditions. This completes the addition of a processor loop. Since each element of $\vec{a_0}$ is accessed only once, there is no reason to add a cache loop, and this is the full description.

5.2.4 Row Renormalization

The first row is then renormalized to unit length.

$$\vec{a_0} \leftarrow \frac{\vec{a_0}}{r_{00}}$$

Translation to MoA notation yields an expression already in DNF.

$$\rho(< 0 > \psi A) = < (\rho A)[1] > \tag{58}$$

$$\left\langle \; 0 \quad i \; \right\rangle \psi A \leftarrow \frac{< 0 > \psi A}{r_{00}} \tag{59}$$

This yields an ONF expression.

$$\vec{a}_0[i] \leftarrow \frac{\vec{a}_0[i]}{r_{00}} \tag{60}$$

$$0 \leq i < n$$

Again, there is no reason to include a cache loop. Application of the same reshaping scheme used in the last step yields the following.

$$\vec{a}_0[i_1 * \lceil \frac{(\rho A)[1]}{p} \rceil + i_2] = \frac{\vec{a}_0[i_1 * \lceil \frac{(\rho A)[1]}{p} \rceil + i_2]}{r_{00}} \tag{61}$$

$$0 \leq i_1 < p$$

$$0 \leq i_2 < min(\lceil \frac{(\rho A)[1]}{p} \rceil, (\rho A)[1] - i_1 * \lceil \frac{(\rho A)[1]}{p} \rceil)$$

This completes the description of the normalization step.

5.2.5 Projection

Next, we compute the length of the projection of the first row on each of the other rows. [9]

$$\vec{r_0} = \vec{a_0} A' \tag{62}$$

$$\vec{r_0} = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \end{bmatrix} \begin{bmatrix} 6 & 12 & 18 & 24 \\ 7 & 13 & 19 & 25 \\ 8 & 14 & 20 & 26 \\ 9 & 15 & 21 & 27 \\ 10 & 16 & 22 & 28 \\ 11 & 17 & 23 & 29 \end{bmatrix} \tag{63}$$

In MoA, the following expression is formulated.

$$\vec{r_0} \leftarrow (< 0 > \psi A) \cdot (1 \bigtriangledown A)^T \tag{64}$$

[9]Recall that $A' \equiv (1 \bigtriangledown A.)^T$

Reduction to DNF yields:

$$\vec{r_0}[i] \leftarrow \left(\sum_{j=0}^{n-1}\left(\left\langle\; 0 \quad j \;\right\rangle \psi A\right) \times \left(\left\langle\; i+1 \quad j \;\right\rangle \psi A\right)\right) \tag{65}$$

Transformation to ONF yields:

$$\vec{r_0}[i] \leftarrow \sum_{j=0}^{n-1} \vec{a}_0[j] * \vec{a}[(i+1) * n + j] \tag{66}$$

$$0 \leq i < (\rho A)[0] - 1 \tag{67}$$

5.3 Adding Processors

The problem can be divided among processors over the primary or secondary axis. A division over the secondary axis produces a situation similar to that encountered in previous steps - each processor computes a subsum and the subsums are combined. In this case, a division over the primary axis is preferred as this division requires no communication between processors.

$$\vec{r_0} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \begin{bmatrix} 6 & 12 \\ 7 & 13 \\ 8 & 14 \\ 9 & 15 \\ 10 & 16 \\ 11 & 17 \end{bmatrix} \mapsto \textbf{processor 0}$$

$$\vec{r_0} = \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix} \begin{bmatrix} 18 & 24 \\ 19 & 25 \\ 20 & 26 \\ 21 & 27 \\ 22 & 28 \\ 23 & 29 \end{bmatrix} \mapsto \textbf{processor 1}$$

The problem has been broken into p pieces which do not communicate. The ONF can be rewritten without re-derivation.

$$\vec{r_0}[i] \leftarrow \sum_{j=0}^{n-1} \vec{a}_0[j] * \vec{a}[(i+1) * n + j] \tag{68}$$

Consequently, $(\rho A)[0]$ is partitioned as follows:

$$0 \leq i_0 < p$$

$$0 \leq i_1 < min(\lceil \frac{(\rho A)[0]}{p} - 1 \rceil, ((\rho A)[0] - 1) - i_0 * \lceil \frac{(\rho A)[0]}{p} - 1 \rceil) \tag{69}$$

5.4 Adding Cache

Each element in a_0 is accessed $m - 1$ times. For optimal performance, these accesses should occur in close succession, so each element only enters the cache once. To ensure this the second dimension of A, or the third dimension of the reshaped A, is broken into pieces and work on one vertical slice is completed before moving on to the next. Pictorially (Apply a matrix multiplication across two-dimensional sub-arrays),

$$+_{\text{red}}\Omega_{<1>} +_{\text{red}}\Omega_{<1>} \begin{bmatrix} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \end{bmatrix} ({}_{\times}\Omega_{<1\;1>})\Omega_{<2\;2>} \begin{bmatrix} \begin{bmatrix} 6 & 9 \\ 7 & 10 \\ 8 & 11 \end{bmatrix} \\ \begin{bmatrix} 12 & 15 \\ 13 & 16 \\ 14 & 17 \end{bmatrix} \end{bmatrix} \mapsto \textbf{processor 0}$$

$$+_{\text{red}}\Omega_{<1>} +_{\text{red}} \Omega_{<1>} \left[\begin{array}{c} \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \\ \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \end{array} \right] ({}_{\times}\Omega_{<1\ 1>})\Omega_{<2\ 2>} \left[\begin{array}{c} \begin{bmatrix} 18 & 21 \\ 19 & 22 \\ 20 & 23 \end{bmatrix} \\ \begin{bmatrix} 24 & 27 \\ 25 & 28 \\ 26 & 29 \end{bmatrix} \end{array} \right] \mapsto \textbf{processor 1}$$

In MoA notation, we have a matrix multiplication over two-dimensional subarrays.

$$\begin{aligned}(p + \lceil \frac{(\rho A)[1]}{cache} \rceil) \widehat{\rho}\ \vec{r_0} \quad \leftarrow \quad & +_{\text{red}}\Omega_{<1>} +_{\text{red}} \Omega_{<1>}(((\frac{n}{cache}) + cache) \widehat{\rho} \\ & < 0 > \psi A)_{({}_{\times}\Omega_{<1\ 1>})}\Omega_{<2\ 2>}(p + \frac{m-1}{p} + \frac{n}{cache} + cache) \ \ \widehat{\rho}\ 1 \bigtriangledown A \qquad (70)\end{aligned}$$

In this case, the DNF form is complicated by the number of reshape operations. Reducing directly to ONF form yields: $\forall i_1, i_2, i_3, i_4 \ni$

$$0 \leq i_1 < p$$
$$0 \leq i_2 < min(\lceil \frac{(\rho A)[0]}{p} - 1 \rceil, ((\rho A)[i] - 1) - i_0 * \lceil \frac{(\rho A)[0]}{p} - 1 \rceil)$$
$$0 \leq i_3 < \frac{(\rho A)[1]}{cache}$$
$$0 \leq i_4 < cache$$

$$\begin{aligned}\vec{r_0}[i_2 * \lceil \frac{(\rho A)[1]}{p} \rceil + i_3] \quad \leftarrow \quad & \sum_{i_3=0}^{\frac{(\rho A)[1]-1}{cache}} \sum_{i_4=0}^{cache-1} \vec{a}[i_3 * cache + i_4] * \\ & \vec{a}[(\rho A)[1] + ((i_1 * \lceil \frac{(\rho A)[0]}{p} - 1 \rceil + i_2) * \frac{(\rho A)[1]}{cache} + i_3) * cache + i_4] \qquad (71)\end{aligned}$$

This is a complete description for this step.

5.4.1 Projection Subtraction

Next, all rows are orthogonalized to the first. Let $A'' \equiv 1 \bigtriangledown A$

$$A^{'} = A^{'} - \vec{r_0}^T \vec{a_0} \qquad (72)$$

In DNF form, we have

$$\left\langle\ i+1 \quad j\ \right\rangle \psi A \leftarrow (\left\langle\ i+1 \quad j\ \right\rangle \psi A) - (\left\langle\ 0 \quad j\ \right\rangle \psi A) \times (< i > \psi \vec{r_0}) \qquad (73)$$
$$(74)$$

$$0 \leq i < (\rho A)[0] - 1$$
$$0 \leq j < (\rho A)[1]$$

Converting to ONF,

$$\vec{a}[((i+1) \times (\rho A)[1]) + j] \leftarrow \vec{a}[((i+1) \times (\rho A)[1]) + j] - \vec{a}[j] \times \vec{r_0}[i] \qquad (75)$$
$$(76)$$

$$0 \leq i < (\rho A)[0] - 1 \qquad (77)$$
$$0 \leq j < (\rho A)[1] \qquad (78)$$

We use the same processor cache divisions as in the previous step. Since there is no communication between processors and no communication between cache blocks, the ONF can again be modified directly.

$$\begin{aligned}\vec{a}[(\rho A)[1] + i_4 + cache \times (i_3 + \frac{(\rho A)[1]}{cache} \times (i_2 + \lceil \frac{(\rho A)[0]}{p} - 1 \rceil \times i_1)) \quad \leftarrow \quad & \vec{a}[(\rho A)[1] + i_4 + cache \times (i_3 + \frac{(\rho A)[1]}{cache} \times (i_2 + \lceil \frac{(\rho A)[0]}{p} - 1 \rceil \times i_1))] - \\ & (\vec{a}[(i_3 \times cache) + i_4] \times \vec{r}_0[(i_1 \times \lceil \frac{(\rho A)[0]}{p} - 1 \rceil) + i_2] \qquad (79)\end{aligned}$$

$$0 \le i_1 < p$$
$$0 \le i_2 < min(\lceil \frac{(\rho A)[0]}{p} - 1 \rceil, ((\rho A)[0] - 1) - i_0 \times \lceil \frac{(\rho A)[0]}{p} - 1 \rceil)$$
$$0 \le i_3 < \frac{(\rho A)[1]}{cache}$$
$$0 \le i_4 < cache$$

This completes the MoA specification of one iteration of the algorithm. To generalize to the full algorithm, we need only update rules for the sizes and start offsets for the recursion. In particular, during the ith iteration, we add an $i * (\rho A)[1]$ offset to any access of A, and an $i * (\rho A)[1] + i$ offset into any access of R.

5.5 Algorithm Specification

We combine the four ONF expressions from our derivation and translate them to pseudo-code. Steps 1-2 form the main loop. Steps 3-6 compute the norm of the first row. Steps 7-9 normalize the first row. Steps 10-14 compute the vector of dot products between the first row and other rows. Steps 15-19 orthogonalize all rows to the first. Input is a two dimensional array A, and integer scalars $cache$ and np. Let $m = l = (\rho A)[0]$. Let $n = (\rho A)[1]$. Let $\vec{a} =$ `rav` A

```
1.  Start: For $i \leftarrow 0$ to $l - 1$ do:
2.      $s \leftarrow n \times i$;  $\vec{r} \leftarrow (m-1) \hat{\rho}\ 0$;  $sum \leftarrow 0$;
3.      For $i_1 \leftarrow 0$ to $p - 1$ do:
4.          For $i_2 \leftarrow 0$ to $\frac{n}{p} - 1$ do:
5.              $sum \leftarrow sum + \vec{a}[s + (i_1 \times \frac{n}{p}) + i_2]^2$
6.      $r_{00} \leftarrow sum^{\frac{1}{2}}$
7.      For $i_1 \leftarrow 0$ to $p - 1$ do:
8.          For $i_2 \leftarrow 0$ to $\frac{n}{p} - 1$ do:
9.              $\vec{a}[s + (i_1 \times \frac{n}{p}) + i_2] \leftarrow \frac{\vec{a}[s + (i_1 \times \frac{n}{p}) + i_2]}{r_{00}}$
10.     For $i_1 \leftarrow 0$ to $p - 1$ do:
11.         For $i_2 \leftarrow 0$ to $\frac{m-1}{p} - 1$ do:
12.             For $i_3 \leftarrow 0$ to $\frac{n}{cache} - 1$ do:
13.                 $sum \leftarrow sum + \sum_{i_4=0}^{cache-1} \vec{a}[s + (i_3 \times cache) + i_4] \times \vec{a}[s + (n + i_4 + cache \times (i_3 + \frac{n}{cache} \times (i_2 + (\frac{m-1}{p}) \times i_1)))]$
14.             $\vec{r}[(i_1 \times (\frac{m-1}{p}) + i_2] \leftarrow sum$
15.     For $i_1 \leftarrow 0$ to $p - 1$ do:
16.         For $i_2 \leftarrow 0$ to $\frac{m-1}{p} - 1$ do:
17.             For $i_3 \leftarrow 0$ to $\frac{n}{cache} - 1$ do:
18.                 For $i_4 \leftarrow 0$ to $cache - 1$ do:
19.                     $\vec{a}[s + n + i_4 + cache \times (i_3 + \frac{n}{cache} \times (i_2 + (\frac{m-1}{p}) \times i_1))] \leftarrow \vec{a}[s + n + i_4 + cache \times (i_3 + \frac{n}{cache} \times (i_2 + (\frac{m-1}{p}) \times i_1))] - \vec{r}[(i_1 \times (\frac{m-1}{p})) + i_2] \times \vec{a}[s + (i_3 \times cache) + i_4]$
20.     $m \leftarrow m - 1$ ; goto Start
```

5.6 Analysis and Results

For the purposes of analysis, assume that we wish to run a QR factorization of a matrix of size $m \times n$ on a system with p processors and an LRU cache which holds $2 * cache + k$ matrix elements, where k is large enough to store iteration counters, matrix sizes, and other small variables. Assume the system has a shared memory architecture. We are interested in the number of memory accesses and the cache miss ratio. The analysis is be performed for one iteration of the algorithm. The total performance can be obtained by summing the following results over the array sizes specified. The speedup with additional processors is limited only by the divisibility of array dimensions.

5.6.1 Vector Length

The first element of R is accessed n times, and never leaves the cache after the first access. Each of the n elements in the first row of A is accessed once. This is a total of $2n$ accesses, with at most $n+p$ cache misses.

5.6.2 Renormalization

The first element of R is accessed n times, and never leaves the cache after the first access. Each of the n elements in the first row of A is accessed twice in succession. This is a total of $3n$ accesses, with at most $n+p$ cache misses.

5.6.3 Projection Length

Each of the n elements in the first row of A is accessed $m-1$ times in close enough succession to remain in the cache. Each of the other $n(m-1)$ elements is accessed twice in succession. This is a total of $3n(m-1)$ accesses, with at most $n(m-1)+np$ cache misses.

5.6.4 Orthogonalization

Each of the n elements in the first row of A is accessed $m-1$ times in close enough succession to remain in the cache. Each of the other $n(m-1)$ elements of A is accessed twice in succession. Each of $m-1$ entries of the first row of R is accessed n times. Each of these is accessed *cache* times in close succession at a time. This is a total of $4n(m-1)$ accesses, with at most $n(m-1)+np+\frac{n(m-1)p}{cache}$ misses.

5.6.5 Totals

For one iteration, we have a total of $5n+7n(m-1)$ memory accesses, with at most $2n(m-1)+2n+2p+2np+\frac{n(m-1)p}{cache}$ misses, or a cache miss ratio of approximately $\frac{2}{7}+\frac{p}{7*cache}$. Thus, after access pattern optimization, the performance of the algorithm is to a good extent independent of the cache size. In comparison, the blocked Modified Gram-Schmidt algorithm will in general never have a cache-miss ratio better than $\frac{m}{2*cache+k}$ which is strictly worse if the cache is small compared to m.

5.6.6 Temporaries

Examination of the implementation confirms that no array temporaries are created. Any implementation will need enough memory to store Q and R. This implementation needs, in addition, only enough memory to store a few counters and scalar temporaries. There is very little duplication of data.

5.7 Discussion

We have developed generic code for the implementation of the Modified Gram-Schmidt algorithm which adapts transparently to a wide range of different system architectures. Usage of the MoA/Psi approach has streamlined the implementation in memory. It uses almost no memory past what is needed for its output and its pattern of memory access is optimized for the cache size of the system. In essence, we have manually optimized the implementation of the algorithm. All of the code produced is very easy for a compiler to handle. This general approach applies to any array based algorithm with regular predictable patterns of memory access and can thus also be applied to many other operations necessary for signal processing.

What follows is an F90 version of the generic design. Any language could have been realized from the generic design. To validate conjectures of performance, experiments were run to compare implementations, e.g. with the LAPACK version of QR.

5.8 QR, ONF, and Fortran90

```
      program qr
c This is the f90 scalarized code
      include 'parameter.h'
```

```
! The parameter.h file inputs the following
! parameters in a general way.
!      integer, parameter :: row = 3
!      integer, parameter :: n = 4
!      integer, parameter :: cachesize = 1
!      integer, parameter :: np = 1
      real mreal,nreal,npreal,preal,cachereal
! i, i1, i2, i3, i4  are used as indicies or loop variables
      integer i, i1, i2, i3, i4, row
      integer l,s,p,cache
!
! z   <-> Gram-Schmidt QR, Input is A, and m by n array
!         This is an in-place algorithm
      real avec(0:(row*n)-1), rvec(0:row-2)
      real r00,total
!
! initializations
!
      m=row
      mreal=m
      l=m
      nreal=n
      npreal=np
      cachereal=cachesize
      preal=p
      do i_=0,(n*m)-1
         avec(i_)=i_
      end do
!
      do i=0,l-1
         s=n*i
         rvec=0
         total = 0
         p=gcd(npreal,nreal)
         do i1=0,p-1
            do i2=0,(n/p)-1
               total=total+avec(s+(i1*(n/p))+i2)**2
            end do
         end do
         r00=total**.5
         do i1=0,p-1
            do i2=0,(n/p)-1
               avec(s+(i1*(n/p))+i2)=avec(s+(i1*(n/p))+i2)/r00
            end do
         end do
         p=gcd(mreal-1,npreal)
         do i1=0,p-1
            do i2=0,((m-1)/p)-1
               cache=gcd(cachereal,nreal)
               total=0
               do i3=0,(n/cache)-1
                  total=total+sum(avec(s+(i3*cache):)*avec(s+
     $(n+cache*(i3+(n/cache)*(i2+((m-1)/p)*i1))):))
               end do
               rvec((i1*((m-1)/p))+i2)=total
            end do
         end do
         do i1=0,p-1
            do i2=0,((m-1)/p)-1
               do i3=0,(n/cache)-1
                  do i4=0,cache-1
                     avec(s+n+i4+cache*(i3+(n/cache)*(i2+((m-1)/p)*i1)))
     $=avec(s+n+i4+cache*(i3+(n/cache)*(i2+((m-1)/p)*i1)))-rvec((i1*
     $((m-1)/p))+i2)*avec(s+(i3*cache)+i4)
                  end do
               end do
            end do
         end do
         m=m-1
         mreal=m
      end do
      print *,avec
!
! Essential functions follow
!
      contains
!
      FUNCTION GCD(a,b)
      a = int(abs(a))
      b = int(abs(b))
      if (a>1e10.or.b>1e10) then
         GCD=1; return
      endif
      if (a.eq.0.or.b.eq.0) then
         GCD=1; return
      endif
      if (a<b) then
         temp=a; a=b; b=temp
      endif
 1010 r=a-b*int(a/b)
      a=b; b=r
      if (abs(r)>1e-10) goto 1010
      GCD = a
      return
      end function gcd
      end program qr
```

5.9 Hardware, Compilers, and QR

The QR was designed and derived with a view that standard compilation techniques would be used to optimize scalar operations. Consequently, since normal forms do not address arithmetic optimizations, the elimination of common scalar arithmetic subexpressions, or constant code propagation, it is assumed that such issues would be done by a current scalarizing compiler or perhaps by using expression templates and attribute evaluation rules, e.g. Pressburger Arithmetic[46](number theory). Such, a compiler[45] should know when it was necessary to pre-compute essential constant expressions, storing them in run-time registers. It would also use the control information sent from the source program to guide it to further optimizations. That is, it would not ignore cache and processor loops, e.g. by collapsing loops, that are meant to control use of memory hierarchies as illustrated in the Convolution and QR designs.

Initial experiments using SGI's Fortran 90 on NCSA's Origin 2000[10] indicate anamolous optimizations of the *psi* derived QR going between compiler optimization flags, -O2 and -Ofast, as compared to LAPACK's QR, whose performance remained constant. When common scalar subexpressions were manually removed, performance increased, thus implying the compiler failed on this particular optimization.

Consequently, the next step is to predictably achieve the high performance indicated by the design using a contemporary programming language, e.g. C pointers, or expression templates, ideally mapping directly to index registers, and other assembler operations. It may be possible to use templates to apply more compiler-like optimizations prior to compilation.

6 Composing Matrix Multiply

6.1 Beamforming

6.1.1 Matrix Multiplication and Radar

Electronic methods of steering beams, communications between radars, and advanced processing, give modern radar systems considerable flexibility. Phased arrays are often considerably more expensive than reflector antennas but they have the advantage that the beam can be steered electronically, rather than mechanically by heavy and expensive(but reliable) turning gears. If the phasing of a receiving antenna is carried out digitally by computer calculation, rather than by analogue devices (which may be controlled by a computer), the process is known as digital beamforming, and there is considerable flexibility for beamshaping and the formation of multiple beams. Beamforming relies on matrix multiplication and would benefit from a *psi* analysis especially if composition of beamforming operations is desired.

6.1.2 Algorithm Specification

$$A \cdot X \cdot B \equiv \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \cdot \begin{bmatrix} 9 & 10 & 11 \\ 12 & 13 & 14 \\ 15 & 16 & 17 \end{bmatrix} \cdot \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix}$$

6.1.3 Adding Processors

We want to algebraically represent the above where each row of each n by n matrix is block distributed over n processors, see Figure 1.

$$A \cdot X \cdot B \equiv +_{red}\Omega_{<2>}(A_{\times}\Omega_{<0\ 1>}(+_{red}\Omega_{<2>}X({}_{\times}\Omega_{<0\ 1>}B)))$$

The shape of $A \cdot B \cdot X$ is $< n\ n >$.

$$\text{Let } Y \equiv (A_{\times}\Omega_{<0\ 1>}(+_{red}\Omega_{<2>}X({}_{\times}\Omega_{<0\ 1>}B)))$$

$$\text{Let } W \equiv (+_{red}\Omega_{<2>}X({}_{\times}\Omega_{<0\ 1>}B))$$

$$\rho(A_{\times}\Omega_{<0\ 1>}(+_{red}\Omega_{<2>}X({}_{\times}\Omega_{<0\ 1>}B))) \equiv < n\ n\ n >$$

So, $\forall i, j, k \ni 0 \leq^* < i, j, k > <^* n$

$$\begin{aligned} < k > \psi +_{red} \Omega_{<2>}Y &\equiv +_{red} < k > \psi Y & (80)\\ &\equiv +_{red} < k > \psi(A_{\times}\Omega_{<0\ 1>}(+_{red}\Omega_{<2>}X({}_{\times}\Omega_{<0\ 1>}B))) & (81)\end{aligned}$$

[10]This work was partially supported by National Computational Science Alliance and utilized the NCSA SGI/CRAY Origin2000.

$$z(0,:) \equiv \sum_{i=0}^{n-1} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \end{bmatrix} \times \begin{matrix} \left[\sum_{i=0}^{n-1} \begin{bmatrix} 9 & 9 & 9 \\ 10 & 10 & 10 \\ 11 & 11 & 11 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 12 & 12 & 12 \\ 13 & 13 & 13 \\ 14 & 14 & 14 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 15 & 15 & 15 \\ 16 & 16 & 16 \\ 17 & 17 & 17 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \end{matrix}$$

$$z(1,:) \equiv \sum_{i=0}^{n-1} \begin{bmatrix} 3 & 3 & 3 \\ 4 & 4 & 4 \\ 5 & 5 & 5 \end{bmatrix} \times \begin{matrix} \left[\sum_{i=0}^{n-1} \begin{bmatrix} 9 & 9 & 9 \\ 10 & 10 & 10 \\ 11 & 11 & 11 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 12 & 12 & 12 \\ 13 & 13 & 13 \\ 14 & 14 & 14 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 15 & 15 & 15 \\ 16 & 16 & 16 \\ 17 & 17 & 17 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \end{matrix}$$

$$z(2,:) \equiv \sum_{i=0}^{n-1} \begin{bmatrix} 6 & 6 & 6 \\ 7 & 7 & 7 \\ 8 & 8 & 8 \end{bmatrix} \times \begin{matrix} \left[\sum_{i=0}^{n-1} \begin{bmatrix} 9 & 9 & 9 \\ 10 & 10 & 10 \\ 11 & 11 & 11 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 12 & 12 & 12 \\ 13 & 13 & 13 \\ 14 & 14 & 14 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \\ \left[\sum_{i=0}^{n-1} \begin{bmatrix} 15 & 15 & 15 \\ 16 & 16 & 16 \\ 17 & 17 & 17 \end{bmatrix} \times \begin{bmatrix} 18 & 19 & 20 \\ 21 & 22 & 23 \\ 24 & 25 & 26 \end{bmatrix} \right] \end{matrix}$$

Figure 1: Visualization of Processor and Time Dimensions for Multiple Matrix Multiplications.

$$\equiv \quad +_{red}(< k\ j > \psi A) \times (< j > \psi W) \tag{82}$$
$$\equiv \quad +_{red}(< k\ j > \psi A) \times (< j > \psi (+_{red}\Omega_{<2>} X({}_{\times}\Omega_{<0\ 1>} B))) \tag{83}$$
$$\equiv \quad +_{red}(< k\ j > \psi A) \times (+_{red} < j > \psi X({}_{\times}\Omega_{<0\ 1>} B)) \tag{84}$$
$$\equiv \quad \sum_{k=0}^{n-1}(< k\ j > \psi A) \times \sum_{k=0}^{n-1}((< j\ i > \psi X) \times (< i > \psi B)) \tag{85}$$
$$\equiv \quad \sum_{k=0}^{n-1}((< k\ j > \psi A) \times (< j\ i > \psi X) \times (< i > \psi B)) \tag{86}$$
$$\quad \mathbf{qed} \tag{87}$$

This denotes the DNF. To illustrate the need to control ONF's we built the DNF without a manual translation to its ONF. From the DNF, we created a generic form to build software, initially in Fortran90. Performance measurements were run against Fortran90's Intrinsic, `Matmul`. These preliminary experiments also indicate the need to control optimizations through preprocessings steps, perhaps through templates.

6.2 DNF to Generic Algorithm

The following denotes the generic algorithm specification based on the DNF above:

```
1. $Z \leftarrow 0$
2. For $i = 0$ to $n - 1$ do:
3.     For $j = 0$ to $n - 1$ do:
4.         For $k = 0$ to $n - 1$ do:
5.             $Z[k;] \leftarrow Z[k;] + A[k;j] \times X[j;i] \times B[i;]$
```

6.3 DNF to Fortran 90 Implementation

We built the following program from the generic specification above.

```
      program mm
c This is the f90 scalarized code
!      include "parameter.h"
      integer, parameter :: n = 1000
! i, j, k, are indices
      integer i, j, k, l
! z   <-> output of MM
      real a(0:n-1,0:n-1), b(0:n-1,0:n-1), c(0:n-1,0:n-1)
      real z(0:n-1,0:n-1)
!      real xtmp, srand, rand
! see batchit.pl to see how nvecl is calculated
! also see how the parameter.h file is created.
!
! initializations
!
!
      xtmp = srand(10001)
            a=1
            b=2
            c=3
      z=0
      do i = 0,n-1
         do j = 0,n-1
            do k = 0,n-1
!               z(k,0:n-1)= z(k,0:n-1)+ (a(k,j)*(c(j,i)*b(i,0:n-1)))
```

```
              do l = 0,n-1
              z(k,l)= z(k,l)+ (a(k,j)*(c(j,i)*b(i,l)))
! Removing the f90 looping
              end do
           end do
        end do
     end do
     print *,z(0,0)
     end program mm
```

7 Mechanizing Psi Reduction

Numerous attempts to mechanize the *Psi Calculus* include using C++ templates[58]. But only hand derived designs have included processor and cache analysis. A feasibility study for the FFT[36] complemented by our use of expression templates indicates the potential for mechanizing the algebraic decomposition and mapping of sequential, shared, and distributed monolithic array software.

7.1 C++

C++ extended C to include type definitions, building upon languages such as Pascal and Algol 60, Figure 9. C++ attempts to bring programming languages to the next generation of programming, through abstractions such as classes, functions, and templates. Through these features, a monolithic style of programming is possible. Furthermore, these features enable compiler-style optimizations, e.g. Abstract Syntax Tree(AST) manipulations on expressions, to occur prior to compilation.

7.1.1 Expression Templates, PETE, and Psi Calculus

Shapes, like types, are important in the specification and semantics of programs, and should support tools for static(or dynamic) error detection and improved compilation techniques. Clearly, shape is clearest in those object-oriented languages that support a container class, e.g. C++, where attributes of the container are clearly separated from those of its data. However, the container class plays no special role, and so the full benefits of shape analysis have gone unrealized. In most imperative and functional languages, the shapes are either fixed in advance, or computed and checked at runtime, e.g. bounds checking, which increase overhead. The exploitation of shape requires a calculus in which values can be decomposed into shape and data so that each can be manipulated separately. The semantic and operational foundations have been laid by Mullin[59] and provide the shape polymorphisms essential for real time computation, e.g. radar, given today's complex computational environments. Shape polymorphic operations are operations such that the shape of the output is a function of the shape of the input arguments alone without regard for the contents. When composing operations using operations built upon the *psi function*, ψ we can create higher order operations, FIR, QR, that are composable *without the creation of array valued temporaries.*

7.1.2 Expression Templates

Passing an expression to a function, in languages such as C, requires passing a pointer to a call-back function containing the expression. A common example is passing mathematical expressions to functions. and when they are called repeatedly, generate an enormous amount of overhead, especially when large array operations occur. Expression templates allow expressions to be passed to functions as arguments that get in-lined into the function body. Expression templates also solve an open problem in the design of vector, matrix, or array classes libraries. That is, this technique allows developers to write code such as:

```
Vector y(100), a(100), b(100), c(100), d(100)
y = (a+b)/(c-d)
```

and compute the result without three intermediate(temporary) arrays, e.g.

```
temp0 = c-d
temp1 = a+b
temp2 = temp/temp1
```

As arrays get large, such temporaries interfere with performance and the management of memory. When expression templates are used, the expression is parsed at compile time, and stored as nested template arguments of an *expression type*. There is a strong relationship between the expression type generated by the compiler and the *parse tree for the expression*. Nodes in the parse tree become types in an applicative template class. To write a function which accepts an expression as an argument, a template parameter is included for the expression type. Note that although the expression object is not instantiated until run time, the expression type is inferred at compile time. For example, let's look at an operator/() which produces a type representing the division of two subexpressions `DExpr<A>` and `DExpr<B>`:

```
template<class A, class B>
DExpr<DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> >
operator/(const DExpr<A>& a, const DExpr<B>& b)
{
typedef DBinExprOp<DExpr<A>, DExpr<B>, DApDivide> ExprT;
return DExpr<ExprT>*ExprT(a,b));
}
```

The return type contains a `DBinExprOp`, a binary operation, with two template parameters, `DExpr<A>` and `DExpr<B>`, the two subexpressions to be divided, and `DApDivide`, which is an applicative template class encapsulating the division operations. The return type is a `DBinExprOp<...>` disguised in a `DExpr<...> class`. Without this disguise we would be required to define eight different operators to handle the combinations of `DBinExprOP<>, DExprIdentity`, and `DExprLiteral` which can occur with operator/(); more for unary.

Due to such optimization potential, numerous scientific libraries have been built exploiting expression templates with a goal of speed and space performance comparable to Fortran[47, 81]. Unlike Fortran90/95, which *extended the grammar of its language* to include *monolithic* arrays and operations on arrays, C++ achieves similar semantic capabilities using class libraries and operator overloading.

Returning to our original example:

```
Vector y(100), a(100), b(100), c(100), d(100)
y = (a+b)/(c-d)
```

Usually, each line in the expression above is evaluated with a loop. Thus, four loops are needed. Ideally, we want to evaluate the expression in a single pass. We want an effect similar to the compiler optimization, *loop fusion*. Actually, we want to distribute indexing over all nodes, a definition in the *Psi Calculus*. Although the effect is similar to compiler optimizations, the compiler optimizers do not see anything but the reduced expression which has all such rules applied. Similarly, each node could have other indexing rules,*i.e. attribute evaluation rules*, reverse, shift, etc., or they could be higher order functions built from these, etc. By combining the ideas of expression templates and iterators, it is possible to generate this code automatically, by building an expression object which contains vector iterators rather than place-holders.

7.1.3 Iterators

Iterators are pointer-like objects that allow programs to step through the elements of a container sequentially without exposing the underlying representation. Iterators can be advanced from one element to the next by incrementing them. Some iterators can also be decremented or allow arbitrary jumps from one element to another, as we will see later. When they are dereferenced, iterators yield a reference to a container element. In addition, they can be compared to each other for equality or inequality.

Iterators interact seamlessly with built-in C++ types. In particular, native C++ pointers are treated as iterators to C++ arrays. Naturally, all containers in the Standard C++ Library define an iterator type, i.e., a nested type iterator that represent their respective pointer-like type.

7.1.4 Iterator Categories

Iterators fall into categories. This is because different algorithms impose different requirements on an iterator they use. For example, the drop(),e.g.($\bigtriangledown$) algorithm needs an iterator that can be advanced by incrementing it, whereas the reverse(), e.g. (Φ), algorithm needs an iterator that can be decremented as well, etc. An iterator category is an abstraction. It represents a set of requirements to an iterator.

There are five categories of Iterators in STL[82] and the Standard C++ Library:

- Input iterators allow algorithms to advance the iterator and give "read only" access to the value.
- Output iterators allow algorithms to advance the iterator and give "write only" access to the value.
- Forward iterators combine read and write access, but only in one direction (i.e., forward).
- Bidirectional iterators allow algorithms to traverse the sequence in both directions, forward and backward.
- Random access iterators allow jumps and "pointer arithmetics."

Each category adds new features to the previous one. The iterator categories obey the following order:

```
input           output
  |                |
  v                v
  +---------------+
         |
         v
      forward
         |
         v
   bidirectional
         |
         v
   random access
```

Using the knowledge of an iterator's category one can provide optimized implementations of an algorithm. The `advance()` operation is an example. It increments (or decrements for negative n) an iterator.

```
template <class Iterator, class Distance>
inline void advance (Iterator& i, Distance n);
```

Obviously, there are many ways to do this. For a C++ array, one would simply perform pointer arithmetic, i.e., add n to the C++ pointer.

```
i += n;
```

For a list, Iterators must step through the sequence and advance step-by-step.

```
if (n >= 0)
while (n--)
++i;
else
while (n++)
--i;
```

7.1.5 Observations

The iterator category, which is an abstraction that represents a set of requirements to an iterator, is information related to an iterator. It is useful for providing optimized versions of an operation like `advance()`. Certain type information is related to an iterator. One of the questions that must be answered when designing the iterators in STL and the Standard C++ Library is: how can information related to a type be expressed in C++ iterators? If we view shapes as types, then the reduction rules of the *Psi Calculus* may provide an answer.

7.1.6 Types Associated with an Iterator

There are two types that might vary depending on the iterator type:

The Distance Type. An operation like `advance()` obviously needs an argument that indicates how far to advance the iterator:

```
template <class Iterator, class Distance>
inline void advance (Iterator& i, Distance n);
```

The type of this distance argument must represent the distance between any two iterators. Hence the distance type depends on the iterator type. For C++ pointers the distance type is the C++ type $\texttt{ptrdiff}_t$, which can represent the difference between any two C++ pointers.

Also, $\texttt{ptrdiff}_t$ is the distance type of all other iterators in STL and Standard Library. However, the distance type in STL and the Standard C++ Library is not limited to $\texttt{ptrdiff}_t$.

7.2 PETE and Scientific Libraries

Consequently, the combination of expression templates and iterators yield optimizations previously possible only through advanced compilation methods. Furthermore, this paradigm creates an environment ideal for *Psi Reduction* and movement between DNFs, ONFs, and dimension lifting.

Although we've seen scientific libraries develop around the use of expression templates and iterators their perceived complexity is due to the compiler like strategies needed to exploit them. Consequently, without the underlying knowledge of recursive decent parsers, grammars, attribute evaluation rules, etc. their exploitation will not be fully realized.

7.2.1 PETE and Compiler-Like Strategies

A step towards providing this concept to a wider user population is PETE[20] which uses classical compiler techniques: a recursive descent parser, attribute evaluation rules, a push down automata, etc.

A recursive descent parser, is a general form of top-down parsing,, that may involve backtracking, that is, making repeated scans of the input. However, backtracking parsers are not seen frequently. One reason is that backtracking is rarely needed to parse programming language constructs. In situations like natural language parsing, backtracking is still not very efficient, and tabular methods such as the dynamic programming algorithm or the method of Earley[18] are preferred, Aho and Ullman's basic compiler text[3], describe other general parsing methods. Consequently, we could imagine packages, such as PETE, emerging across languages that support expression templates, e.g. JAVA[79].

PETE is a portable C++ framework that allows users to easily add expression-template functionality to their container classes and to simply perform complex expression manipulations. Unlike conventional overloaded operators, which immediately apply an operation and typically generate temporaries to hold intermediate results, PETE operator functions return objects that can be combined to incrementally build up the parse tree of an expression, which is encoded in the object's type. PETE knows how to store non-terminal nodes and scalars in the expression parse tree, but it must be told how to store the container objects that are passed as arguments to the operator functions, these are the needed attribute evaluation rules[45] and constitute where shape analysis and *Psi Reduction* rules reside. By default, PETE supports 45 built-in operators including all the C++ mathematical operators along with a collection of common mathematical functions, e.g. $sin()$. PETE includes an easy-to-use tool to automatically generate any desired functions. These functions are written to the `Listing1Operators.h`. Iterators are stored in the expression tree.

PETE is a powerful, yet easy-to-use, framework for adding expression-template functionality to a container class, thus providing a high level API to exploit expression templates. PETE has been used in several large scientific codes ranging from hydrodynamics, linear accelerator modeling, fusion device modeling[20] and signal/image processing[69]. Consequently, we used PETE to implement *Psi Reduction*.

7.3 Pete, MoA, Psi Calculus, and the TD Convolution

We studied and prototyped compiler-style optimizations for array indexing operations using definitions in the *Psi Calculus*. Such optimizations resulted in very efficient implementations in which intermediate array valued temporaries were eliminated in complex array expression. We tested the performance of our optimizations by applying them to compositions of 1-d indexing operations that define the Time-Domain Convolution. We chose this algorithm both because it is common in signal processing, and because it requires a combination of several basic array operations, such as take, drop, and concatenate, as well as the arithmetic operations plus and times, for its implementation. Thus, the composition of these operations illustrate how *Psi Reduction*, eliminates intermediate arrays, by combining low-level indexing operations to build efficient high level operations defined by indexing.

We tested the performance of our optimizations to that of a hand coded C/C++ loop implementation, and a typical C++ implementation where intermediate array valued temporaries are produced. Our implementation achieved performance nearly as good as the hand coded loop implementation, whose interface achieves the high level of abstraction of a typical C++ implementation, and whose performance far surpasses that of the class C++ implementation[58]. Figures 7 shows the performance of three implementations of the following expression:

```
a = rev(take(N,drop(M,rev(b))))
```

using hand coded pointer C loops, enhanced PETE, and typical C++. The horizontal axis is $log_2 M$. The vertical axis is the average time of each implementation to perform the operation, normalized to the average time of the hand coded loop implementation. Note that the loop and expression template implementation significantly outperform the typical C++ approach, with the latter taking roughly 200 times as long as the loop implementation for small(M< 16) arrays. Figure 8 compares only the performance of the hand coded loop and enhanced PETE implementations. The small overhead, apparent for small vector sizes, may be removed in two ways: by creating a *setup step* to preallocate dynamic arrays at system startup time(run-time polymorphism) for each subsequent assignment in the system, and by statically sizing arrays at compile time(compile-time polymorphism). Each approach has advantages and disadvantages, but in either case, once the indexing in expressions is reduced, no intermediate array valued temporaries are created.

8 On An Abstract Machine: Thoughts and Requirements

A polymorphic computing environment[19] is facilitated by a uniform algebraic framework for arrays. Furthermore, since an array's structure can include architectural components, performance cost functions may be realized. PCAs denotes a new model of parallel computing and optimization based on the idea that application functionality and requirements direct and refine the use of resources on the physical machine. To achieve this, an abstraction layer must be present that analyzes what the programmer has specified. The abstraction layer must also determine how to configure the machine given the request and application software. We call this abstraction layer *middle-ware* . Middle-ware has two constituent parts: an Abstract Language Machine(ALM), and an Abstract Virtual Machine(AVM) which are defined using an Abstract Machine(AM).

An ALM is a replacement for a compiler and is essential since it must interface the AVM and AM. What differentiates the ALM from a compiler is that it can map to all levels of the memory hierarchy by using explicit instructions of the AM. Through algebraic and index calculi methods, a normal form(or *psi reduced form*) is produced from the input program that characterizes semantically how the computation and data flow moves through the memory hierarchy of the PCA. This normal form expresses the required and essential mathematics in the program, first in terms of Cartesian coordinates(semantic meaning,DNF), then in terms of starts, stops, and strides, mappings and data-flow(operational meaning or how it should be built,ONF). From the normal form a mapping is made to the instruction set of an AM. These instructions would then be bound to real instructions supported by a PCA. A PCA is a collection of fundamental micro-architectural building blocks, i.e. memories, execution units, interconnections, etc., that can be configured into different execution models that communicate performance and state information up to and down from an application.

An Abstract Machine(AM) characterizes processors, instruction sets supported by that processor, memory hierarchy, connectivity of memory to processors and memory mappings(decompositions). An Abstract Virtual Machine(AVM) characterizes the virtualness of an AM, for example, use 8 processors when there are only 4 real processors.

Initially, we assume that

1. processors and memory are configured in a regular way, i.e, an n-dimensional topology with one or many processors. Such a topology subsumes meshes, n-cubes, etc..
2. data can be moved as one component, groups of contiguous components(block) and groups of strided components(cyclic).
3. all memory components are integral multiples of each other.

8.1 Abstract Virtual Machine Instructions

An instruction set for an AVM realizes an arbitrary PCA configuration. Processors and instructions support variable formats and lengths to reflect a changing and flexible AVM. Instructions must provide a way to synchronize and restructure resources as well as move data from one level of the memory hierarchy to another.

8.1.1 Processors and Instructions

Instructions must characterize all operations typical of an arbitrary PCA. In order to generalize movement of data through various levels of the memory hierarchy, data movement(loads, stores, copy, send, receive, etc.) must be abstracted. Consequently, data can be moved one component at a time or in chunks. The chunks may be contiguous one strided. Historically, these movements are characterized differently and they are often performed by separate functional units of the machine or OS. At the lowest level exists scalar and vector registers(single component or block). At the cache level there exists associativity: one way(or direct), two way, etc., and tiling. At the local memory level, DMA reads, which may be block contiguous or block cyclic, and at shared and distributed memory levels we think of block or cyclic distributions. All of these movements will be handled by the PCA's middle-ware which conceptually subsumes the work typically done in a compiler, OS, and firmware.

Definition 8.1 *Let $P_p =< shape^{11}, speed >$ where $p = 1, np$ and np =number of different processors available in the PCA. For each P_p $\exists$ an instruction set $I_{p_j} =< op, na >$ where $j = 1, nfu$, and nfu = number of functional units supported by processor P_p.*

We assume that all arguments are accessed through address pointers that provide shape and component information.

8.1.2 Example:

P_0 =@ Intel, P_1 = @IBM, P_2 =@ SGI, etc.

8.2 Levels of Memory

Definition 8.2 *Let ${}_{P_p}M_{m_t} =< shape, speed >$, $m = 1, nm$, where nm =number of levels in the memory hierarchy numbered from lowest to highest, $t = 1, nt$, where nt = number of types of ${}_{P_p}M_{m_t}$ memory. The ${}_{P_p}M_{m_t}$'s include registers, cache, local memory, paged memory, disk memory, shared memory, distributed memory, etc. ordered from lowest to highest. For each level, there are types at that level: integer, float, complex, etc..*

1. $shape[0]$ = the dimensionality of ${}_{P_p}M_{m_t}$: 1 = vector, 2= matrix, etc..
2. $shape[0] \bigtriangleup (1 \bigtriangledown shape)$ = shape of ${}_{P_p}M_{m_t}$.

[11]In the case of a one-dimensional view, the size is the shape.

Hence, $(-(shape[0]-1)) \bigtriangleup shape =$ shape of each ${}_{P_p}M_{m_t}$ component and $\pi(shape[0] \bigtriangleup (1 \bigtriangledown shape)) =$ the total number of components in ${}_{P_p}M_{m_t}$.

8.2.1 Example

Suppose we have 2 level 0 memories. That is, we have two sets of registers: 16 integer scalar registers and 16 integer vector registers with vector length = 8, denoted by

$$ {}_{P_p}M_{0_0} = << 1\ 16 >, speed > \tag{88}$$

and

$$ {}_{P_p}M_{0_1} = << 2\ 16\ 8 >, speed > \tag{89}$$

respectively. Hence, in the register memory defined by expression(88):

$shape =< 1\ 16 >$: $shape[0] = 1 =$, which is the dimensionality of ${}_{P_p}M_{0_0}$.

$shape[0] \bigtriangleup (1 \bigtriangledown shape) = 1 \bigtriangleup (1 \bigtriangledown < 1\ \ 16 >) = 1 \bigtriangleup < 16 >=< 16 >=$ which is the shape of ${}_{P_p}M_{0_0}$.

$(-(shape[0]-1)) \bigtriangleup shape = (-(1-1)) \bigtriangleup < 1\ 16 >= 0 \bigtriangleup < 1\ 16 >= \Theta$. That is, The shape of each component is the empty vector, i.e. each component is a scalar. Consequently, we have 16 scalar registers.

$\pi(shape[0] \bigtriangleup (1 \bigtriangledown shape)) = \pi(1 \bigtriangleup (1 \bigtriangledown < 1\ 16 >) = \pi(1 \bigtriangleup < 16 >) = \pi < 16 >= 16$, the total number of components in ${}_{P_p}M_{0_0}$.

Caches, ${}_{P_p}M_{1_t}$ are defined similarly. Cache associativity is characterized as decompositions and subsequently is abstracted in the shape. Similarly, decompositions are specified algebraically by abstractions of shapes associated with each level of the memory hierarchy.

8.3 RAW, Performance Predictions, and Machine Abstractions

Abstractions for a machine are essential if reliable performance predictions are desired. As seen in our analysis for the TM Convolution, we reliably predicted the performance for variants of the implementation built by the RAW team. However, our prediction was based on our knowledge of their implementation and how it would scale algebraically. Notice that our design recognized the need to pull a multiplier outside. This is precisely what the RAW team did independent of our prediction. However, there could have been many variations of how they built their 4 Tape 4 Tile implementation. Instead of an ideal parallelism due to pipelining: a parallel multiply-add, and parallel IO; they could have used a serial adder, no pipelining, or any other combination. This implies the need to *order instructions* in the Abstract Machine as outlined above.

9 Conclusion

We have investigated numerous ways to apply MoA and the *Psi Calculus* to array based computation in radar:

1. As a mathematical formalism to *manually* reason about algorithms in radar: TM Convolution, Modified Gram Schmidt QR Decomposition, and composition of matrix multiplication, used in Beamforming.
2. As a programmer's tool through the mechanization of MoA and *Psi Calculus* introduced here through expression templates and PETE.
3. To describe how to build assembler and configure hardware.
4. To describe an Abstract Machine.

Together the above items demonstrate the viability of our approach.

10 Future Research

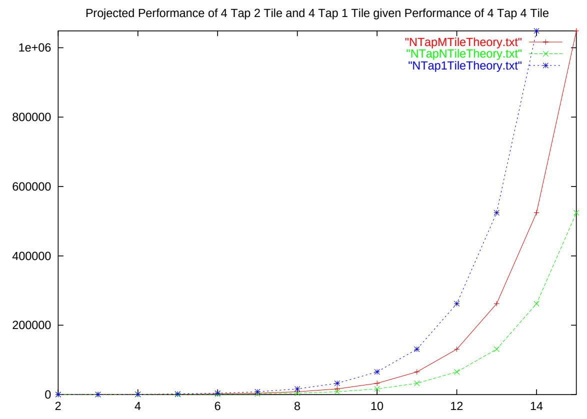
Future research will focus on identifying those real-time systems that can benefit from the analyses presented herein; disciplines whose dominant data structure is the array. We will continue to explore manual designs and derivations of algorithms to architectures in an attempt to discover common index patterns across disciplines and in so doing develop high performance libraries with the mathematical reliability needed for defense applications. Such common access patterns transcend all levels of memory and should be able to adapt to changes. Manual designs will also be used to predict performance on other PCA style architectures, aid in the design of optimal assembler instructions to drive such architectures, and develop Machine Abstractions to reason. This direction of research also implies using our analysis for hardware design and verification[67, 66].

We will continue to explore how templates can be used to preprocess scientific software. Presently, only C++ expression templates are used and they are used infrequently. PETE is the only known *software package* developed that creates an API such that tree traversals, node attribute evaluation rule applications, etc. are kept transparent from the user. Although extremely useful, PETE is built upon fundamental algorithms in compiler theory. As noted in our discussion on PETE, other tree traversal algorithms could be explored. We also believe that templates can be used for program blocks optimizations[68, 36], i.e. through graph templates. Similarly, we plan to investigate how JAVA's templates can be used for run-time polymorphic applications of MOA and the Psi Calculus. Finally, we plan to investigate how templates might be added to Fortran95.

11 Acknowledgments

Special thanks go the MIT Lincoln Laboratory and the University at Albany, SUNY, for making this sabbatical possible, and in particular to Bob Bond who shares an interest in this research. Eddie Rutledge helped to build the PETE implementation of psi reduction for 1-d TD Convolution, Janice McMahon and Hank Hoffman worked on PCA and RAW, Paul Chang and Gil Raz ; QR. Without their help this research would not have been possible. Finally, I thank Dan Rosenkrantz and Harry Hunt, my treasured colleagues at UA, who have worked with me on these concepts for over seven years.

Appendix



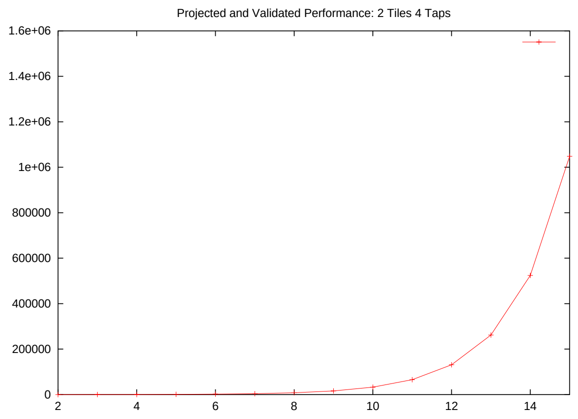
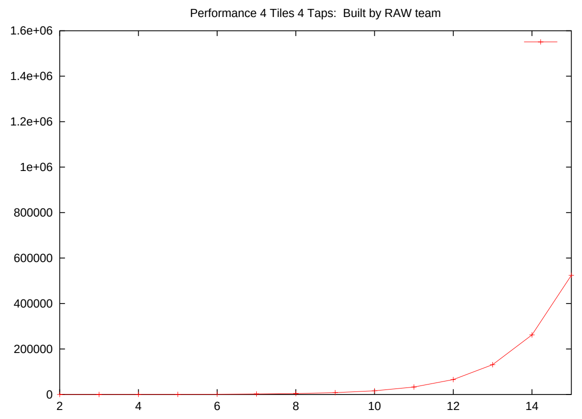


Figure 2: Predicted performance for 2 tile 4 tap, 1 tile 4 tap, given 4 tap 4 tile performance

Figure 4: Predicted and Validated Performance and Design: 4 Taps 2 Tiles



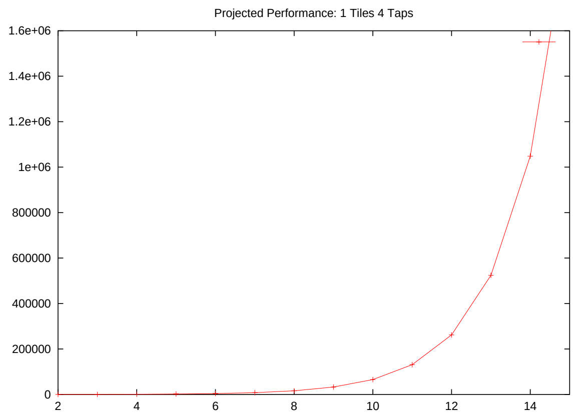


Figure 3: Actual Performance on RAW: 4 Taps 4 Tiles

Figure 5: Predicted Performance and Design: 4 Taps 1 Tiles

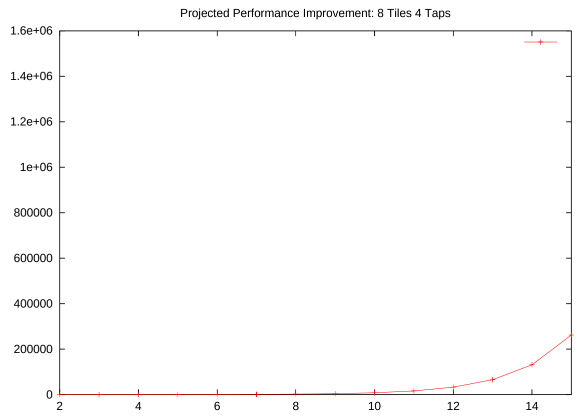


Figure 6: Predicted Performance Increase and Design: 4 Taps 8 Tiles

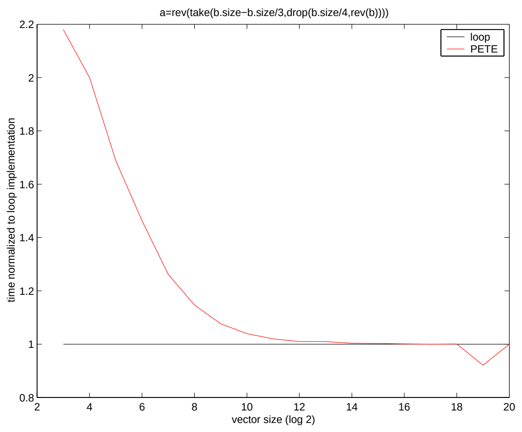


Figure 8: PETE, Psi Reduction, and C pointers: `a = rev(take(N,drop(M,rev(b))))`

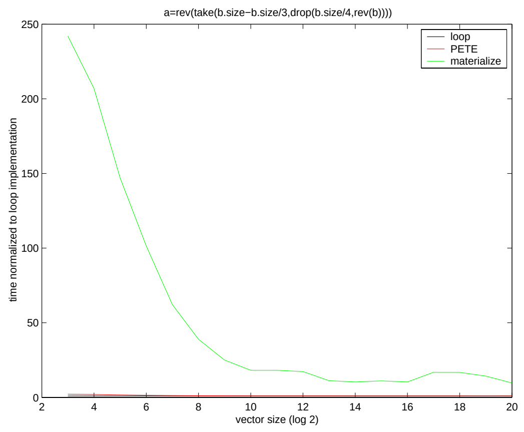


Figure 7: PETE, Psi Reduction, C pointers, C++: `a = rev(take(N,drop(M,rev(b))))`

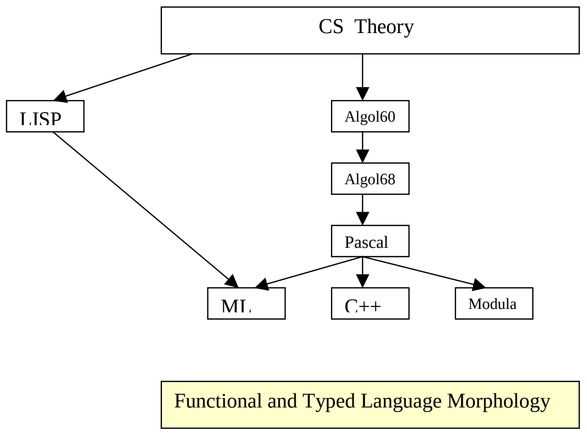


Figure 9: On Object Oriented Programming

Symbol	**Name**	**Description**
δ	Dimensionality	Returns the number of dimensions of an array.
ρ	Shape	Returns a vector of the upper bounds or sizes of each dimension in an array.
$\iota\xi^n$	Iota	When $n = 0$(scalar), returns a vector containing elements 0, to $\xi^0 - 1$. When $n = 1$(vector), returns an array of indices defined by the shape vector ξ^1
ψ	Psi	The main indexing function of the Psi Calculus which defines all operations in MoA. Returns a scalar if a full index is provided, a sub-array otherwise.
rav	Ravel	vectorizes a multi-dimensional array based on an array's layout$(\gamma_{row}, \gamma_{col}, \gamma_{sparse}, ...)$
γ	Gamma	Translates indices into offsets given a shape.
γ'	Gamma Inverse	Translates offsets into indices given a shape.
$\vec{s} \hat{\rho}\ \xi$	Reshape	Changes the shape vector of an array, possibly affecting its dimensionality. Reshape depends on layout(γ).
$\pi\vec{x}$	Pi	Returns a scalar and is equivalent to $\prod_{i=0}^{(\tau x)-1} x[i]$
τ	Tau	Returns the number of components in an array,$(\tau\xi \equiv \pi(\rho\xi))$
$\xi_l +\!\!\!+ \xi_r$	Catenate	Concatenates two arrays over their primary axis.
$\xi_l f \xi_r$	Point-wise Extension	A data parallel application of f is performed between all elements of the arrays.
$\sigma f \xi_r$ $\xi_l f \sigma$	Scalar Extension	σ is used with every component of ξ_r in the data parallel application of f.
$\triangle$	Take	Returns a sub-array from the beginning or end of an array based on its argument being positive or negative.
$\triangledown$	Drop	The inverse of Take
$_{op}$red	Reduce	Reduce an array's dimension by one by applying op over the primary axis of an array.
Φ	Reverse	Reverses the components of an array.
Θ	Rotate	Rotates, or shifts cyclically, components of an array.
$\bigcirc\!\!\!\!\backslash$	Transpose	Transposes the elements of an array based on a given permutation vector
Ω	Omega	Applies a unary or binary function to array argument(s) given partitioning information. Ω is used to perform all operations (defined over the primary axis only) over all dimensions.

Figure 10: Summary of MoA Operations

References

[1] P. S. Abrams. *An APL Machine.* PhD thesis, Stanford University, 1970.

[2] W. Ackerman and J. Dennis. Val - a value-oriented algorithmic language: Preliminary reference manual. Technical Report 218, Department of CSEE, MIT, 1979.

[3] J. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 2000.

[4] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference, February 26–28, 1957, Los Angeles, CA, USA*, pages 188–198, 1957.

[5] H. P. Barendregt. Introduction to Lambda Calculus. In *Aspenæs Workshop on Implementation of Functional Languages, Göteborg.* Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1988.

[6] N. Belanger, L. Mullin, and Y. Savaria. Formal methods for the partitioning, scheduling, and routing of arrays on a hierarchical bus multiprocessing architecture. In *ATABLE92.* Universite d'Montreal, 1992.

[7] K. Berkling. Arrays and the lambda calculus. Technical report, CASE Center and School of CIS, Syracuse University, 1990.

[8] R. Bernecky. Compiling APL. *Arrays, Functional Languages and Parallel Systems*, 1991.

[9] A. Bohm, D. Cann, R. Oldehleft, and J. Feo. Sisal reference manual language version 2.0. Technical Report 91-118, Department of CS, Colorado State University, 1991.

[10] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *Journal of the ACM*, 22(1):129–144, 1975.

[11] T. A. Budd. An APL compiler for a vector processor. *ACM Transactions on Programming Languages and Systems*, 6(3):297–313, July 1984.

[12] D. Cann. Retire Fortran? A Debate Rekindles. *CACM*, 35(8), 1992.

[13] W. Celmaster. Modern Fortran revived as the language of scientific computing. *Digital Technical Journal*, 8(3), 1996.

[14] A. Church. *The Calculi of Lambda-Conversion.* Princeton University Press, 1941.

[15] L. Coffin. Designing a new programming methodology for optimizing array accesses in complex scientific problems, 1994. UMR Undergraduate Research Program (OURE).

[16] D. Dooling and L. Mullin. Indexing and distributing a general partitioned sparse array. *Proceedings of the Workshop on Solving Irregular Problems on Distributed Memory Machines*, 1995.

[17] E. V. Dyke. A dynamic incremental compiler for an interpretive language. *Hewlett-Packard Journal*, July 1977.

[18] J. Earley. An efficient context-free parsing algorithm. *CACM*, 13(2), 1970.

[19] D. et al. The RAW architecture: Signal processing on a scalable composable computation fabric. In *Proceedings of High Performance Embedded Computing Workshop 2001*, 2001.

[20] J. et al. Genereric programming in POOMA and PETE. Technical report, Los Alamos National Laboratory, 1998.

[21] A. Falkoff. Algorithms for Parallel Search Memories. *JACM*, 9(4), 1962.

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

[23] K. Gallivan, R. Plemmons, and A. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, March 1990.

[24] G. Gao, R. Yates, J. Dennis, and L. Mullin. An efficient monolithic array constuctor for scientific computation. In *Proceedings of the Third Workshop on Programming Languages and Compilers for parallel Computing.* MIT Press, 1990.

[25] S. Gerhart. *Verification of APL Programs.* PhD thesis, CMU, 1972.

[26] K. Gödel. 'Die Vollständigkeit der Axiome des logishchen Funktionenkalkuls'. *Monatschefte für Mathematik und Physik*, 37:349–360, 1930. Translated by Stefan Bauer-Mengelberg in van Heijenoort (1967), pp.582–591.

[27] G. Golub and C. VanLoan. *Matrix Computations.* Johns Hopkins University Press, 1983.

[28] L. J. Guibas and D. K. Wyatt. Compilation and delayed evaluation in APL. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8, Jan. 1978.

[29] G. Hains and L. Mullin. Parallel functional programming with arrays. *The British Computer Journal of the British Computer Society: the society of information systems engineering*, 36(22), 1993.

[30] G. Hains and L. M. Mullin. An algebra of multidimensional arrays. Technical Report 782, Université de Montréal, 1991. Presented at the 2nd Montreal Workshop on Programming Language Theory - Algebraic and Logical Approaches in Programming Languages, December 1991.

[31] K. Hammond, L. Augustsson, and B. Boutel. Report on the programming language Haskell: A non-strict purely functional language. Technical report, University of Glasgow, 1994.

[32] M. A. Helal. Dimension and shape invariant programming: The implementation and the application. Master's thesis, The American University in Cairo, Department of Computer Science, 2001.

[33] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1*, May 1993.

[34] F. Hill and G. Peterson. Digital systems: Hardware organization and design, 1973.

[35] H. Hoffmann. Fir filter performance on Raw. MIT Laboratory for Computer Science, July 2002.

[36] H. Hunt, L. Mullin, and S. Rosenkrantz. A transformation-based approach for sequential/parallel/distributed scientific software: the FFT. *submitted to Journal of Computational and Applied Harmonic Analysis*, August 2002.

[37] H. B. Hunt III, L. Mullin, and D. J. Rosenkrantz. Towards an indexing calculus for efficient distributed array computation. Technical Report 97–4, University at Albany, Department of Computer Science, 1997.

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

[39] J. D. J. Choi, L. Ostrouchov, A. Petitet, D. Walker, and R. Whaley. The design and implementation of the ScaLapack, LU, QR, and Cholesky factorization routines. Technical report, Oak Ridge National Laboratory, 1994. TM-12470.

[40] M. Jenkins, 94. Research Communications.

[41] M. Jenkins and J. Glasgow. A logical basis for nested array data structures. *Computer Languages*, 14, 1989.

[42] M. Jenkins and L. Mullin. A comparrison of Array Theory and a Mathematics of Arrays. In *Arrays, Functional Languages, and Parallel Systems.* Kluwer Academic Publishers, 1991.

[43] S. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2, 1936.

[44] R. Lathwell. System formulation and APL Shared Variables. *IBM Journal of Research and Development*, 17(4), 1973.

[45] P. Lewis, D. Rosenkrantz, and R. Stearns. *Compiler Design Theory.* Addison-Wesley, 1976.

[46] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. A formal approach to domain-oriented software design environments. In *Proc. of the 9th Knowledge-Based Software Engineering Conference (KBSE'94)*, pages 48–57, Monterey, CA, 1994.

[47] A. Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *Proceedings of International Symposium on Computing in Object-Oriented Parallel Environments*, 1998.

[48] D. MacQueen, r. Harper, and R. Milner. Functional programming in ML. Technical report, University of Edinburgh, 1987.

[49] T. Miller. Tentative compilation: A design for an APL compiler. Technical Report 133, Yale University, 1979.

[50] T. More. Axioms and theorems for arrays. *IBM Journal of Research and Development*, 17(2), 1973.

[51] L. Mullin. The Psi compiler project. In *Workshop on Compilers for Parallel Computers*. TU Delft, Holland, 1993.

[52] L. Mullin and M. Jenkins. Effective data parallel computation using the Psi calculus. *Concurrency – Practice and Experience*, September 1996.

[53] L. Mullin, W. Kluge, and S. Scholtz. On programming scientific applications in SAC – a functional language extended by a subsystem for high level array operations. In *Proceedings of the 8th International Workshop on Implementation of Functional Languages, Bonn, Germany*, 1996.

[54] L. Mullin, J. McMahon, and R. Bond. Abstract machines for polymorphous computing architectures for signal/image processing. In *Proceedings of the High Performance Embedded Computing Workshop*, MIT Lincoln Laboratory, Lexington, MA, 2002.

[55] L. Mullin and T. McMahon. Parallel algorithm derivation and program transformation in a preprocessing compiler for scientific languages. Technical Report CSC-94-29, University of Missouri-Rolla, Dept of CS, 1994.

[56] L. Mullin, D. Moran, and D. Dooling. Method and apparatus for indexin patterned sparse arrays for microprocessor data cache. US Patent No. 5878424, March 1999.

[57] L. Mullin, N. Nemer, and S. Thibault. The Psi compiler v4.0 for HPF to Fortran 90 user's guide. Department of CS, UMR, 1994.

[58] L. Mullin, E. Rutledge, and R. Bond. Monolithic compiler experiments using C++ Expression Templates. In *Proceedings of the High Performance Embedded Computing Workshop HPEC 2002*, MIT Lincoln Laboratory, Lexington, MA, 2002.

[59] L. M. R. Mullin. *A Mathematics of Arrays*. PhD thesis, Syracuse University, Dec. 1988.

[60] L. R. Mullin. Psi, the indexing function: A basis for FFP with arrays. In *Arrays, Functional Languages, and Parallel Systems*. Kluwer Academic Publishers, 1991.

[61] L. R. Mullin, D. Dooling, E. Sandberg, and S. Thibault. Formal methods for scheduling, routing and communication protocol. In *Proceedings of the Second International Symposium on High Performance Distributed Computing (HPDC-2)*. IEEE Computer Society, July 1993.

[62] L. R. Mullin, D. Eggleston, L. J. Woodrum, and W. Rennie. The PGI–PSI project: Preprocessing optimizations for existing and new F90 intrinsics in HPF using compositional symmetric indexing of the Psi calculus. In M. Gerndt, editor, *Proceedings of the 6th Workshop on Compilers for Parallel Computers*, pages 345–355, Aachen, Germany, Dec. 1996. Forschungszentrum Jülich GmbH.

[63] R. Nikhil. ID version 88.1 reference manual. Technical Report CSG Memo 284, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[64] E. Palvast. *Programming for Parallelism and Compiling for Efficiency*. PhD thesis, Delft University of Technology, June 1992.

[65] A. Perlis. Steps towards an APL compiler. Technical Report 24, Yale University, 1975.

[66] H. Pottinger, W. Eatherton, J. Kelly, L. Mullin, and T. Schifelbein. Hardware assits for high performance computing using A Mathematics of Arrays. In *Proceedings of INEL94*, 1994.

[67] H. Pottinger, W. Eatherton, J. Kelly, L. Mullin, and R. Ziegler. An FPGA based reconfigurable co-processor board utilizing A Mathematics of Arrays. In *Proceedings of the IEEE Circuits and Systems Symposium ISCAS95*, May 1995.

[68] D. Rosenkrantz, L. Mullin, and H. B. H. III. On materializations of array–valued temporaries. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing 2000 (LCPC'00)*, Yorktown Heights, NY, Aug. 2000. Lecture Notes in Computer Science, 2002, Springer–Verlag.

[69] E. Rutledge and J. Kepner. PVL: An object oriented software library for paralell signal processing. In *Proceedings of IEEE Cluster 2001*, 2001.

[70] J. Sylvester. Lectures on the principles of universal algebra. In *American Journal of Mathematics: VI*, volume 4. reprinted in Mathematical Papers, 1884.

[71] S. Thibault and L. Mullin. Generating indexing functions of regularly sparse arrays for array compilers. Technical report, University of Missouri-Rolla, 1994. TR 94-08.

[72] S. Thibault and L. Mullin. A reduction semantics for array expressions: The Psi compiler. Technical report, University of Missouri-Rolla, 1994. TR 95-05.

[73] H. Tu and A. Perlis. FAC: A Functional APL Language. *IEEE Software*, 3(1):36–45, 1986.

[74] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 1936.

[75] A. Turing. Intelligent machinery. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*. Edinburgh University Press, 1969.

[76] D. Turner. An overview of Miranda. *SIGPLAN Notices*, 21(12), 1986.

[77] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[78] T. Veldhuizen. C++ templates as partial evaluation. In O. Danvy, editor, *Proceedings of PEPM'99, The ACM SIGPLAN Workshop on Partial Evaluation and semantics-based program manipulations*, 1999.

[79] T. Veldhuizen. Just when you thought your little language was safe: Expression Templates in Java. Technical report, Indiana University Computer Science Department, 2001.

[80] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in *C++ Gems*, ed. Stanley Lippman.

[81] T. L. Veldhuizen. Arrays in Blitz++. In D. Caromel, R. R. Oldehoeft, and M. Tholburn, editors, *Proc. Second International Symposium on Scientific Computing in Object–Oriented Parallel Environments (ISCOPE '98)*, volume 1505 of *Lecture Notes in Computer Science*, Santa Fe, NM, Dec. 1998. Springer–Verlag.

[82] M. Vilot. An introduction to the Standard Template Library. *C++ Report*, 6(8), 1994.

[83] B. Wu, Y. Zhang, and J. Kong. Time-domain computer simulation of Synthetic Apeture Radar(SAR) for rough surfaces. In *Proceedings of Progress in Electromagnetism Research Symposium*. Research Laboratory of Electronics, MIT, Cambridge, MA, July 2000.