Efficient Radar Processing Via Array and Index Algebras

Lenore R. Mullin Harry B. Hunt III Daniel J. Rosenkrantz *

Department of Computer Science
University at Albany, SUNY
Albany, NY 12222
17 March 2003

Abstract

Embedded software processing requirements for DSP, especially for radar, computations are expected to exceed 1×10^{12} operations per second within 5 years. Consequently, the efficient use of memory at all levels of the memory hierarchy is becoming increasingly essential in these computations. For radar, and more generally DSP, computations generally involve compositions of linear and multi-linear operators, and consequently, are array-based. Here, we illustrate how a general array algebra, together with a suitably rich compatible index calculus, can be used to develop software for radar and other DSP applications tuned to use the levels of memory hierarchies efficiently. We do this by using the array algebra, MoA, together with the index calculus ψ-Calculus, to develop a convolution algorithm tuned to use a processor/memory hierarchy efficiently. **Keywords: embedded digital systems, radar, signal processing, arrays, high performance, index calculus, shapes, psi, MoA.**

*[lenore,hunt,djr]@cs.albany.edu, NSF CCR 0105536

1 Introduction

1.1 Reasoning about Radar

Reasoning about radar, from a *computational perspective*, entails reasoning about the data structures underlying the algorithms for radar computations. Most, such algorithms, are characterized by linear and multi-linear operations, especially convolution. Thus, they are characterized computationally by matrix operations. Consequently, we believe that the future development of efficient scalable, portable algorithms, for radar, more generally for DSP applications, will be greatly facilitated by the use of a high-level array algebra during algorithm design. Additionally, since program efficiency depends critically upon the efficient use of memory/processor hierarchy, this array algebra should be combined with a suitably powerful index calculus. This calculus should facilitate data layout, movement, and manipulation at all levels of the memory/processor hierarchy. We present strong evidence that MoA and Psi Calculus[15, 8] are suitable candidates for such an algebra and calculus. This evidence consists of a detailed semi-mechanized development using MoA and the Psi-Calculus of an algorithm for convolution. During the algorithm development, we carry out array dimension lifting and data restructuring driven by the memory/processor hierarchy, coincident with array decompositions and layouts. We also show how temporary array materializations are minimized using Moa and the Psi Calculus.

Contemporary programming languages[1], often provide the expressivity but not the performance required for real time systems. One major reason for this is that they do not provide adequate methods to understand and predictively analyze the performance of algorithms given diverse memory and processor layouts[2]. Languages typically scalarize their monolithic array operations. They then use various loop transformation theories to discover structure, parallelism, etc. However, once scalarization occurs the intentional information of the algorithm in terms of compositions/products of high-level arrays is lost.

2 MoA and the Psi Calculus

The array algebra, MoA, and the index calculus *Psi Calculus* , can describe array computations on both uni- or multi-processor topologies. It is centered around a generalized array indexing function, *psi*.

All array operations in this theory are defined using the indexing function. The algebra of MoA denotes a core set of operations proven useful for representing algorithms in scientific disciplines. Unlike other theories about arrays[18, 11], 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[2].

There have been several investigations into a mathematics to describe array computations. These include More's theory of arrays (AT)[11], a formal description of AT's nested arrays in a first order logic[9] and the development of a Mathematics of Arrays (MoA) [15]. In [10] the correspondence between AT and MoA is described. 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.

2.1 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[17]. 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

[1] Matlab, C++, Fortran 90/95

[2] Even when attempts are made to communicate with the compiler, e.g. distributions for multiprocessing[6, 20], the semantics of what is desired is delivered without a guarantee of how the algorithm was built.

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[3], or a network of workstations[5, 17, 4]. 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*[12] 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 [17]. We ported and scaled these designs to a 32 processor CM5[4]. We used a similar approach, a linear processor array, to map a parallel sparse LU Decomposition to a network of RS6000s[5]. These ideas were later realized in hardware[13].

3 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[22]. 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. Whenever, possible a functional description will be given. Examples and Figures will illustrate functional descriptions. Generic designs, parameterized by array size, number of processors, and size of cache will be developed.

TD Convolution: MoA Design and Derivation

We want to perform the convolution of vector $\vec{\mathcal{X}}$ with vector $\vec{\mathcal{H}}$. Denote the length of $\vec{\mathcal{X}}$ and $\vec{\mathcal{H}}$ by $\tau\vec{\mathcal{X}}$ and $\tau\vec{\mathcal{H}}$ respectively. Denote the result of the convolution by $\vec{\mathcal{Z}}$. Denote the cache size by *cache*. **Assume:**

1. $\tau\vec{\mathcal{X}} \geq \tau\vec{\mathcal{H}}$
2. $\tau\vec{\mathcal{H}} \geq cache$
3. $\tau\vec{\mathcal{H}} \mod cache = 0$

Example:

1. $\vec{\mathcal{X}} \equiv <\ 1\ 2\ 3\ 4\ 5\ >$, so $\tau\vec{\mathcal{X}} = 5$
2. $\vec{\mathcal{H}} \equiv <\ 6\ 7\ 8\ >$, so $\tau\vec{\mathcal{H}} = 3$
3. Let $\vec{\mathcal{X}}'$ denote a vector representing how 0's are used to pad $\vec{\mathcal{X}}$ during the convolution. Here $\vec{\mathcal{X}}'$ is $<\ 0\ 0\ 1\ 2\ 3\ 4\ 5\ 0\ 0\ >$. In general, $\tau\vec{\mathcal{X}}' = \tau\vec{\mathcal{X}} + 2\ \tau\vec{\mathcal{H}} - 2$
4. The length of $\vec{\mathcal{Z}}$, denoted by $\tau\vec{\mathcal{Z}}$, is $\tau\vec{\mathcal{H}} + \tau\vec{\mathcal{X}} - 1$, in this case 7.

I To see the time domain using arrays, shift $\vec{\mathcal{X}}'$ $\tau\vec{\mathcal{Z}}$ times: by 0, by 1, ... by $\tau\vec{\mathcal{Z}} - 1$. The following array denotes $\tau\vec{\mathcal{Z}}$ time steps.

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

II. From each row, i.e. time step, take the necessary $\tau\vec{\mathcal{H}}$ pieces.

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

III. To obtain $\vec{\mathcal{Z}}$ perform the multiplication and addition with $\vec{\mathcal{H}}$.

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

IV. With this generally formulated in MoA, then reduced by the Psi Calculus, the following *normal form* is produced:

Element j of $\vec{\mathcal{Z}}$ is

$$\sum_{i'=0}^{\tau\vec{\mathcal{H}}} < \tau\vec{\mathcal{H}} - (i^{'}+1) > \psi\vec{\mathcal{H}} \times < i^{'} + j > \psi\vec{\mathcal{X}} \tag{1}$$

qed (2)

where $0 \leq j < \tau\vec{\mathcal{Z}}$.

This completes the derivation for the convolution algorithm on a sequential processor and denotes the generic design of how to build the code.

3.1 Adding Processors

Recall how the computation is performed on a uniprocessor[3]. Suppose the number of processors, p, is 2.

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

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, $\frac{1}{2}\vec{\mathcal{H}}$ must be used in each section AND the addition would be over multiple processors. That is, the $\tau\vec{\mathcal{Z}}$ dimension is broken into 2 parts, p and $\frac{\tau\vec{\mathcal{Z}}}{p}$, Consequently, each processor computes half the elements of $\vec{\mathcal{Z}}$. This lifts the dimension of the problem by one.

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

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

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

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

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

This decomposition implies we must reshape our array abstraction thus adding a processor loop to our original design. Thus $\tau\vec{\mathcal{Z}}$ becomes $(p +\!\!+ \lceil \frac{\tau\vec{\mathcal{Z}}}{p} \rceil)$. We consequently reshape by partitioning the 0th dimension, or time domain portion of the array abstraction, thus increasing the dimension of the abstraction by one.

3.2 Adding Cache

If we now want to add a cache loop we must partition the 1st dimension of our abstraction , i.e. $(\tau\vec{\mathcal{H}})$ thus adding yet another dimension. 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. We reshape again by partitioning the 1st dimension. Note how the adder is pulled outside.

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

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

3.3 ONF for TD Convolution

Putting this all together we produce a generic design for processors and cache. Each element of $\vec{\mathcal{Z}}$ is obtained by:

$$\sum_{icache_{row}=0}^{\frac{\tau\vec{\mathcal{H}}}{cache}-1} \vec{\mathcal{H}}[((\tau\vec{\mathcal{H}}) - ((icache_{row} \times cache) + (\iota\ \ cache)) - 1] \times \vec{\mathcal{X}}`[((((\lceil \frac{\tau\vec{\mathcal{Z}}}{p} \rceil \times i_0) + i_1) + icache_{row} \times cache) + \iota\ \ cache)] \quad (3)$$

qed (4)

3.3.1 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:
This is the processor loop.
2. For $i_1 = 0$ to $\frac{\tau\vec{\mathcal{Z}}}{p} - 1$ do:
3. $sum \leftarrow 0$
4. For $icache_{row} = 0$ to $\frac{\tau\vec{\mathcal{H}}}{cache} - 1$ do:
5. For $i_3 = 0$ to $cache - 1$ do:
This is the cachesize loop.
6. $sum \leftarrow sum + \vec{\mathcal{H}}[((\tau\vec{\mathcal{H}}) - ((icache_{row} \times cache) + i_3)) - 1]$
$\times \vec{\mathcal{X}}`[(((((\lceil \frac{\tau\vec{\mathcal{Z}}}{p} \rceil \times i_0) + i_1) + icache_{row} \times cache) + i_3)]$

Note how $\vec{\mathcal{H}}$ is accessed from the rear.

4 Conclusion

4.1 Benefits of Moa and Psi Calculus

- A processor/memory hierarchy can be modeled by reshaping data using an extra dimension for each level.
- Composition of monolithic operations can be reexpressed as compositions of operations on smaller data granularities that
  - Match memory hierarchy levels.
  - Avoid materialization of intermediate arrays.
- Algorithms can be automatically, algebraically, transformed to reflect array reshapings above.
- Facilitate programming expressed at a high level
  - Facilitate intentional program designs and analysis.
  - Facilitate portability and scalability.
- This approach is applicable to many problems in radar.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

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

[9] 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.

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

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

[12] 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.

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

[14] 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.

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

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

[17] 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.