PCA and Abstract Machines: 
A Uniform Way of Reasoning about Array-Based Computation in Science: 
Algebraically Connecting the Hardware-Software Boundary

Lenore M. Mullin
Computer Scientist
Advanced Scientific Computing Research
August 8, 2011
Sabbatical Research at MIT

This presentation reports on sabbatical research conducted at MIT Lincoln Laboratory: September 2001 – September 2002 and NSF Grants CCR-0105536,

Publications that report on this research may be found in:


“Intermediate Languages for Enhanced Parallel Performance”, NSF Presidential Faculty Fellowship, 1992, National Science Foundation: nsf01118 5.

Outline

• Overview
  – Motivation
    • The mapping problem
    • Application domain
    • Write once portable software
    • Current abstractions
    • MoA & $\psi$ calculus
• Algorithm in Science: in DoE, DARPA, DoD, ... , e.g. Radar
• Mechanizing *Psi Reduction*
• On Abstract Machines
• Conclusion & Future Research (from sabbatical): 2001-2010
A Mathematic of Arrays (MoA) and an index (ψ) calculus
Creating an optimization space for **Reasoning about Arrays**

**Overview**

**Algebraically Describe**
- Algorithm
- Architecture
  - Processor
  - Memory Hierarchy
- Decompositions
  - Block-cyclic
- Mappings

**Formal Verification of Designs**
- λ & ψ calculi

**Programming Languages & Compiler Theory**
- Strongly Typed
- Declarative
- Functional
- Fortran
- ZPL
- SAC

**Category Theory**
- SAC
- Category Theory

**Theory of Computation**
- Functional
- C++
- Expression Templates
- Monolithic compiler-style optimization

**Mechanized Algorithm & Environment**
- Compilation
- Optimization

**Manual design & derivation**
- Monolithic compiler optimization

**1-dimensional algorithm alone**
- PETE-Style Templates

**n-dimensional**
- Manual design & derivation

**Loop Transformation**
- AST Manipulation

**Block-cyclic**
- Formal Verification of Designs

**ψ calculus**
Write Once Portable Software

Depends on Advances in Computer Science

Theory of Computation
(Reason about programs)

Models of Computation

Turing, Gödel, Church
Scalar, Van Neuman machines
Recursive function theory
$\psi$ calculus
Church-Rosser Property

Berckling & Mullin:
$\lambda$ calculus with arrays

Hains & Mulin
Bird-Meertens Formalism & arrays

Mullin: $\psi$ calculus & $\lambda$ calculus

$\therefore$ A model of Computation with arrays

Programming Languages
Software Engineering

Compilers & Interpreters
Scalar Fortran APL LISP

Monolithic Fortran 90
C++ classes, functions, templates

Pre and/or Post optimizations
AST manipulation
– Compiler or compiler style
– Attribute Evaluation Rules
– Algebra of Arrays and $\psi$ Calculus

CS Theory

Compiler Theory
Category Theory
Generic Programming
Computational Complexity
Theory of Computation

Reason About Arrays $\rightarrow$ Reason About Science (DoE Algorithms)
Even when operations compose, they don’t compose, \(X(YZ)\) without temporary arrays.
Motivation

Whole Array Operations
Scientific Algorithms
- [multi-]Linear Algebra
- DoE algorithms

Operations

Scientific Algorithms
- [multi-]Linear Algebra
- DoE algorithms

Semantic Meaning in Cartesian coordinates

Denotational

Operational

How to implement in starts, stops, strides at all specified memory levels

DNF

Common Mathematical Abstraction

ONF

Common Machine Abstraction

Algebra

Matlab

FORTRAN

Assembly
Motivation: The Mapping Problem

Mathematics of Arrays
- Math and indexing operations in same expression
- Framework for design space search
  - Rigorous and provably correct
  - Extensible to complex architectures

Example: “raising” array dimensionality

x: < 0 1 2 ... 35 >

Map:

< 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 >
< 30 31 32 >
< 33 34 35 >
MOA Visualization

Mathematical:

Memory domain:

(Single) processor domain:

Multiple processor:
PSI Calculus

Basic Properties:
• An index calculus; psi function
• Shape polymorphic functions and operators
• Fundamental type is the array (*even scalars are 0-d arrays*)
• Denotational Normal Form = reduced form in Cartesian coordinates (independent of data format)
• Operational Normal Form = reduced form for 1-d (memory layout) representation

Example:

<table>
<thead>
<tr>
<th>Expression</th>
<th>ONF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=cat(rev(B), rev(C))</td>
<td>( \forall i: 0 \leq i &lt; B\text{.size} ) ( A[i]=B[\text{size}-1-i] ) ( \forall i:B\text{.size} \leq i &lt; (B\text{.size}+C\text{.size}) ) ( A[i]=C[\text{size}+B\text{.size}-1-i] )</td>
</tr>
</tbody>
</table>

ONF has minimum number of reads/writes

PSI Calculus rules applied mechanically to produce ONF which is easily translated to optimal loop implementation
### Some Psi Calculus Operations

**Built Using \( \psi \) & Shapes**

<table>
<thead>
<tr>
<th>Operations</th>
<th>Arguments</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>take</td>
<td>Vector A, int N</td>
<td>Forms a Vector of the first N elements of A</td>
</tr>
<tr>
<td>drop</td>
<td>Vector A, int N</td>
<td>Forms a Vector of the last (A.size-N) elements of A</td>
</tr>
<tr>
<td>rotate</td>
<td>Vector A, int N</td>
<td>Forms a Vector of the last N elements of A concatenated to the other elements of A</td>
</tr>
<tr>
<td>cat</td>
<td>Vector A, Vector B</td>
<td>Forms a Vector that is the concatenation of A and B</td>
</tr>
<tr>
<td>unaryOmega</td>
<td>Operation Op, dimension D, Array A</td>
<td>Applies unary operator Op to D-dimensional components of A (like a for all loop)</td>
</tr>
<tr>
<td>binaryOmega</td>
<td>Operation Op, Dimension Adim, Array A, Dimension Bdim, Array B</td>
<td>Applies binary operator Op to Adim-dimensional components of A and Bdim-dimensional components of B (like a for all loop)</td>
</tr>
<tr>
<td>reshape</td>
<td>Vector A, Vector B</td>
<td>Reshapes B into an array having A.size dimensions, where the length in each dimension is given by the corresponding element of A</td>
</tr>
<tr>
<td>iota</td>
<td>int N</td>
<td>Forms a vector of size N, containing values 0 . . N-1</td>
</tr>
</tbody>
</table>

\( = \) index permutation

\( = \) operators

\( = \) restructuring

\( = \) index generation
**Psi Reduction: An Example**  
**2d and Higher**

**Given**: A a 2 by 3 array  
**Perform**: $A^T$ and index row 2 in reverse

<table>
<thead>
<tr>
<th>A</th>
<th>$A^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\begin{pmatrix} 0 &amp; 1 &amp; 2 \ 3 &amp; 4 &amp; 5 \end{pmatrix}$</td>
<td>$\begin{pmatrix} 0 &amp; 3 \ 1 &amp; 4 \ 2 &amp; 5 \end{pmatrix}$</td>
</tr>
</tbody>
</table>

$\langle 2 \rangle \psi A^T = \langle 2 \ 5 \rangle$

reverse  $(\langle 2 \rangle \psi A^T) = \langle 5 \ 2 \rangle$

**Express Algebraically**: $(\phi(\langle 2 \rangle \psi (\phi A)))$

Let $\phi =$ reverse  
Let $\phi =$ transpose

| $\rho \phi \langle 2 \rangle \psi \phi A = \rho \langle 2 \rangle \psi \phi A = 1 \land \rho \phi A = 1 \land \rho \phi \phi A = \langle 2 \rangle$   |  
| $\forall \ 0 \leq i < 2 \ \langle i \rangle \psi \phi \langle 2 \rangle \psi \phi A = (\rho \langle 2 \rangle \psi \phi A)[0] \cdot i - 1 \rangle \psi \langle 2 \rangle \psi \phi A = \langle 2 - i - 1 \rangle \psi \langle 2 \rangle \psi \phi A = \langle 1 \rangle \psi \langle 2 \rangle \psi \phi A = \langle 1 \ 2 \rangle \psi A$  |  
| $\langle 0 \rangle \psi \langle 2 \rangle \psi \phi A = \langle 0 \ 2 \rangle \psi A$  |  

$\rho \phi \langle 2 \rangle \psi \phi A = \rho \langle 2 \rangle \psi \phi A = 1 \land \rho \phi A = 1 \land \rho \phi \phi A = \langle 2 \rangle$

**Psi Reduce to**

**Get DNF**

| $\langle 1 \ 2 \rangle \psi A = @A + \gamma(\langle 1 \ 2 \rangle ; \langle 2 \ 3 \rangle) = 5$  |  
| $\langle 0 \ 2 \rangle \psi A = @A + \gamma(\langle 0 \ 2 \rangle ; \langle 2 \ 3 \rangle) = 2$  |  

| @A + 5 | @A + 2 | stride  |
| start   | stop    | -3      |

**Get ONF from DNF**

**Cartesian indices to starts, stops & strides**
Definition of \( y = \text{conv}(h, x) \)

\[
y[n] = \sum_{k=0}^{M-1} h[k] x[n-k]
\]

where \( x \) has \( N \) elements, \( y \) has \( M \) elements, \( 0 \leq n < N + M - 1 \), and \( x' \) is \( x \) padded by \( M-1 \) zeros on either end.

<table>
<thead>
<tr>
<th>Algorithm and PSI Calculus Description</th>
<th>Algorithm step</th>
<th>Psi Calculus</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial step</td>
<td>( x = [1\ 2\ 3\ 4] ) \quad ( y = [5\ 6\ 7] ) \quad ( M=3, N=4 )</td>
<td>( x = [1\ 2\ 3\ 4] ) \quad ( y = [5\ 6\ 7] )</td>
</tr>
<tr>
<td>Form ( x' )</td>
<td>( x' = \text{cat} (\text{reshape} (&lt;M-1&gt;, &lt;0&gt;), \text{cat}(x, \text{reshape}(&lt;M-1&gt;,&lt;0&gt;))) = )</td>
<td>( x' = [0\ 0\ 1\ 2\ 3\ 4\ 0\ 0] )</td>
</tr>
<tr>
<td>rotate ( x' ) ( (N+M-1) ) times</td>
<td>( x'_{\text{rot}} = \text{binaryOmega}(\text{rotate},0,\text{iota}(N+M-1),1,x') )</td>
<td>( x'_{\text{rot}} = [0\ 0\ 1\ 2\ldots] ) \quad ( &lt;0\ 1\ 2\ 3\ldots&gt; )</td>
</tr>
<tr>
<td>take the “interesting” part of ( x'_{\text{rot}} )</td>
<td>( x'<em>{\text{final}} = \text{binaryOmega}(\text{take},0,&lt;M&gt;,1,x'</em>{\text{rot}}) )</td>
<td>( x'_{\text{final}} = [0\ 0\ 1] ) \quad ( &lt;0\ 1\ 2\ldots&gt; )</td>
</tr>
<tr>
<td>multiply</td>
<td>Prod = \text{binaryOmega} (*,1,y,1,x'_{\text{final}})</td>
<td>Prod = [5\ 6\ 14\ 21\ldots]</td>
</tr>
<tr>
<td>sum</td>
<td>( Y = \text{unaryOmega} (\text{sum},1,\text{Prod}) )</td>
<td>( Y = [7\ 20\ 38\ldots] )</td>
</tr>
</tbody>
</table>

PSI Calculus operators compose to form higher level operations.
Derivation

The complete derivation to the operational normal form (ONF) is in:

ONF for the Convolution Decomposition with Processors & Cache

Generic form- 4 dimensional

1. For $i_0 = 0$ to $p-1$ do:

2. For $i_1 = 0$ to $tz/p - 1$ do:

3. \[ \text{sum} \leftarrow 0 \]

4. For $\text{icache}_{\text{row}} = 0$ to $M/\text{cache}-1$ do:

5. For $i_3 = 0$ to $\text{cache} - 1$ do:

6. \[ \text{sum} \leftarrow \text{sum} + y \left[ (M - ((\text{icache}_{\text{row}} \times \text{cache}) + i_3)) - 1 \right] \]

\[ \times X' \left[ (((tz/p \times i_0) + i_1) + \text{icache}_{\text{row}} \times \text{cache}) + i_3 \right] \text{mod} \ tz \]

\[
\begin{align*}
p & \quad = \text{number of processors} \\
\text{cache} & \quad = \text{cache size}
\end{align*}
\]

Assume all variables are well defined, evenly divisible, and within bounds.

Time Domain
Notice the ONF

Notice:
0. We do lots of integer vector and arithmetic.
1. Start, Stop, Stride and Count will be used everywhere.
<table>
<thead>
<tr>
<th>Index Theory Introduced Abrams 1972</th>
<th>MoA &amp; $\psi$ calculus theory: Mullin ’88</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>Prototype compiler: output C, F90, HPF: <em>Mullin and Thibault’94</em></td>
</tr>
<tr>
<td>C</td>
<td>HPF compiler: AST manipulations: <em>Mullin, et al ‘96</em></td>
</tr>
<tr>
<td>C++</td>
<td>SAC: functional C: <em>Mullin and Bodo ’96</em></td>
</tr>
<tr>
<td></td>
<td>C++ classes: <em>Helal, Sameh and Mullin ’01</em></td>
</tr>
<tr>
<td></td>
<td>C++ expression templates: <em>Mullin, Rutledge, Bond ’02</em></td>
</tr>
<tr>
<td></td>
<td>Lifting Compiler Optimizations to Application Programmer Interface</td>
</tr>
<tr>
<td></td>
<td>PVL with the portable expression template engine(PETE)</td>
</tr>
<tr>
<td></td>
<td>Parallel and distributed processing, Abstract machine, Automate cost and determine optimizations: minimize search space</td>
</tr>
</tbody>
</table>
On Abstract Machines

• Presently
• Ideal
Presently

Questions

1. Where is morphing information?
2. How does program compiler, OS communicate with hardware?
3. How does hardware information get used by software layers?
4. How does hardware change and is the regular / the norm for PCA’s?

- Programs may or may not be “verified”
- Memory access patterns may or may not be well defined

- Must handle C++, e.g.
- With or without MPI, OPENMPI or other paradigm
- Must know how to
  – Schedule resources: memory, processors, registries, buffers, etc.
  – Handle interrupts

Input Program

C or Fortran Compiler

Executable

OS

Firebase

FORTRAN, Python, Matlab, C, …

• Must be aware of instruction set

• Morphable hardware

• Programs may or may not be “verified”
• Memory access patterns may or may not be well defined

• Must handle C++, e.g.
• With or without MPI, OPENMPI or other paradigm
• Must know how to
  – Schedule resources: memory, processors, registries, buffers, etc.
  – Handle interrupts

FORTRAN, Python, Matlab, C, …
Ideal Performance/config info

TRIPS

Ideal? machine

<table>
<thead>
<tr>
<th>C or Fortran</th>
<th>Ideal Lang?</th>
<th>Matlab</th>
</tr>
</thead>
</table>

- Spec/Requirements Info

Abstract Language Machine (ALM)

- Memory hierarchy
- Abstraction of “compile time” environment

- Interrupts
- Abstraction of “run-time” environment

Abstract VIRTUAL Machine (AVLM)

Scheduling Virtualizing
Translation of AM into configuration/instructions of real machines

Create a Plan

Create a template for sequential/parallel/distributed

Input program verified & reduced

Specify the Abstract Machine

Runtime bind to real machines

- Requirements out-

RAW

TRIPS

Ideal? machine

- Translation of “state” into abstraction information
- Update database

Abstraction Database

- Rules Invariant over architecture
- Contiguous access
- Parallel Properties

- Parallel program/counter
- Meta instructions
- Parallel scheduler

Performance/config info
• **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.

• 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 **middleware**.

• Middleware has two constituent parts: an **Abstract Language Machine (ALM)**, and an **Abstract Virtual Machine (AVM)** which is defined using the 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.

• 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. Example: Use 8 processors when there are only 4 real processors.
Subsequent Work 2001-2010

- **Continue manual designs and mappings**
  - Scientific algorithms
    - LU, QR, ... and their relationship
    - Relaxation, FFT, Wavelets, Products, ... and their relationship.
  - Identify families
  - Algebraically
    - Mapping and indexing

- **Explore optimizations for enhanced algebraic subset of all languages of interest**

- **Hardware & software co-design & verification**
The FFT, an Example
Plans, Templates, and Abstract Machines

Input: $x$ in $C^n$ and $n = 2^t$, where $t \geq 1$ is an integer.
Output: The FFT of $x$.

\begin{align*}
x & \leftarrow P_n x \\
& \text{for } q = 1 \text{ to } t \\
& \quad \begin{align*}
L & \leftarrow 2^q \\
r & \leftarrow n/L \\
x_{L \times r} & \leftarrow B_L \ x_{L \times r}
\end{align*}
\end{align*}

Here, $P_n$ is a $n \times n$ permutation matrix. Letting $L_* = L/2$, and $\omega_L$ be the $L$'th root of unity, matrix $B_L = \begin{bmatrix}
I_{L_*} & \Omega_{L_*} \\
I_{L_*} & -\Omega_{L_*}
\end{bmatrix}$, where $\Omega_{L_*}$ is a diagonal matrix with values $1, \omega_L, \ldots, \omega_L^{L_*-1}$ along the diagonal.

**FIG. 2.1. High-level algorithm for the radix 2 FFT**
What follows is Mechanizable

The following publication gives the details of mechanization.

do col = 0,r-1
    do row = 0,L-1
        if (row < L/2) then
            xx(row,col) = x(row,col) + weight(row)*x(row+L/2,col)
        else
            xx(row,col) = x(row-L/2,col) - weight(row-L/2)*x(row,col)
        end if
    end do
end do

**FIG. 3.1. Direct Scalarization of the Matrix Multiplication**
do col = 0, r-1
  do row = 0, L-1
    if (row < L/2) then
      xx(L*col+row) = x(L*col+row) + weight(row)*x(L*col+row+L/2)
    else
      xx(L*col+row) = x(L*col+row-L/2) - weight(row-L/2)*x(L*col+row)
    end if
  end do
end do

Fig. 3.2. Scalarized Code with Data Stored in a Vector

do col' = 0, n-1, L
  do row = 0, L-1
    if (row < L/2) then
      xx(col'+row) = x(col'+row) + weight(row)*x(col'+row+L/2)
    else
      xx(col'+row) = x(col'+row-L/2) - weight(row-L/2)*x(col'+row)
    end if
  end do
end do

Fig. 3.3. Striding through the Data Vector
Re-tile and Unroll Loop

do col = 0,r-1  
do row = 0,L/2-1  
do group = 0,1  
  if ( group == 0 ) then  
    xx(row,group,col) =  
      x(row,group,col) + weight(row)*x(row,group+1,col)  
  else  
    xx(row,group,col) =  
      x(row,group-1,col) - weight(row)*x(row,group,col)  
  end if  
end do  
end do  

Fig. 3.4. Re-tiled Loop

do col = 0,r-1  
do row = 0,L/2-1  
  xx(row,0,col) = x(row,0,col) + weight(row)*x(row,1,col)  
  xx(row,1,col) = x(row,0,col) - weight(row)*x(row,1,col)  
end do  
end do  

Fig. 3.5. Unrolled Inner Loop
Optimized Basic Block

\[
\begin{align*}
d & \col = 0, n-1, L \\
d & \row = 0, L / 2 - 1 \\
& xx(\col + \row) = x(\col + \row) + \weight(\row) \times x(\col + \row + L / 2) \\
& xx(\col + \row + L / 2) = x(\col + \row) - \weight(\row) \times x(\col + \row + L / 2) \\
\end{align*}
\]

end do
end do

**Fig. 3.6. Unrolled Inner Loop With Data Stored as a Vector**

\[
\begin{align*}
c & = \weight(\row) \times x(\col + \row + L / 2) \\
x & \times(x(\col + \row)) = x(\col + \row) + c \\
x & x(\col + \row + L / 2) = x(\col + \row) - c \\
\end{align*}
\]

**Fig. 3.7. Optimized Basic Block**
do q = 1,t
L = 2**q
do row = 0,L/2-1
  weight(row) = EXP((2*pi*i*row)/L)
end do
do col' = 0,n-1,L
  do row = 0,L/2-1
    c = weight(row)*x(col'+row+L/2)
    xx(col'+row) = x(col'+row) + c
    xx(col'+row+L/2) = x(col'+row) - c
  end do
end do
end do
x = xx
end do

**Fig. 3.8. Loop with Optimized Basic Block**

do q = 1,t
L = 2**q
do row = 0,L/2-1
  weight(row) = EXP((2*pi*i*row)/L)
end do
do col' = 0,n-1,L
  do row = 0,L/2-1
    c = weight(row)*x(col'+row+L/2)
    d = x(col'+row)
    x(col'+row) = d + c
    x(col'+row+L/2) = d - c
  end do
end do
end do

**Fig. 3.9. Loop with In-Place Butterfly Computation**
do col' = 0, n-1, L
  cvec(0:L/2-1) = weight(0:L/2-1) * x(col'+L/2:col'+L-1)
  dvec(0:L/2-1) = x(col':col'+L/2-1)
  x(col'+L/2:col'+L-1) = dvec(0:L/2-1) + cvec(0:L/2-1)
  x(col'+L/2:col'+L-1) = dvec(0:L/2-1) - cvec(0:L/2-1)
end do

FIG. 3.10. Vectorizing the Butterfly Computation
do q = 1, breakpoint - 1
    L = 2**q
    do row = 0, L/2-1
        weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = 0, n-1, L
        do row = 0, L/2-1
            c = weight(row)*x(col'+row+L/2)
            d = x(col'+row)
            x(col'+row) = d + c
            x(col'+row+L/2) = d - c
        end do
    end do
end do

do q = breakpoint, t
    L = 2**q
    do row = 0, L/2-1
        weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = 0, n-1, L
        do row = 0, L/2-1
            c = weight(row)*x(col'+row+L/2)
            d = x(col'+row)
            x(col'+row) = d + c
            x(col'+row+L/2) = d - c
        end do
    end do
end do

**Fig. 4.1. Loop Splitting the Butterfly Computation**
do q = 1, breakpoint - 1
  L = 2**q
  do row = 0, L/2 - 1
    weight(row) = EXP((2*pi*i*row)/L)
  end do
  do p = 0, m - 1
    do col' = p*psize, (p+1)*psize - 1, L
      do row = 0, L/2 - 1
        c = weight(row)*x(col'+row+L/2)
        d = x(col'+row)
        x(col'+row) = d + c
        x(col'+row+L/2) = d - c
      end do
    end do
  end do
end do

do q = breakpoint, t
  L = 2**q
  do row = 0, L/2 - 1
    weight(row) = EXP((2*pi*i*row)/L)
  end do
  do p = 0, m - 1
    do col' = 0, n - 1, L
      do row = p, L/2 - 1, m
        c = weight(row)*x(col'+row+L/2)
        d = x(col'+row)
        x(col'+row) = d + c
        x(col'+row+L/2) = d - c
      end do
    end do
  end do
end do

Fig. 4.2. Initial Parallel Computation Plan
do p = 0,m-1
  do q = 1,breakpoint - 1
    L = 2**q
    do row = 0,L/2-1
      weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col’ = p+psize,(p+1)*psize-1,L
      c = weight(row)*x(col’+row+L/2)
      d = x(col’+row)
      x(col’+row) = d + c
      x(col’+row+L/2) = d - c
    end do
  end do
end do

do p = 0,m-1
  do q = breakpoint,t
    L = 2**q
    do row = 0,L/2-1
      weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col’ = 0,n-1,L
      c = weight(row)*x(col’+row+L/2)
      d = x(col’+row)
      x(col’+row) = d + c
      x(col’+row+L/2) = d - c
    end do
  end do
end do

FIG. 4.3. Local Memory Parallel Computation Plan with Processor Loops Outermost
do p = 0, m-1
    do q = breakpoint, t
        L = 2**q
        do row' = 0, L/(2*m)-1, 1
            weightcyclic(row') = EXP((2*pi*i*(m*row'+p))/L)
        end do
        do col' = 0, n-1, L
            do row = p, L/2-1, m
                c = weightcyclic((row-p)/m)*x(col'+row+L/2)
                d = x(col'+row)
                x(col'+row) = d + c
                x(col'+row+L/2) = d - c
            end do
        end do
    end do
end do

FIG. 4.4. Second q Loop with Contiguous Access of Needed Weights
Plan for Second q Loop with Contiguous Access

```
do p = 0, m-1
   CENTRALIZED_TO_CYCLIC_PARTITIONED(x, xcyclic, m, psize, p)
   do q = breakpoint, t
      L = 2**q
      do row' = 0, L/(2*m)-1, 1
         weightcyclic(row') = EXP((2*pi*i*(m*row'+p))/L)
      end do
      do col' = 0, n-1, L
         do row = p, L/2-1, m
            c = weightcyclic((row-p)/m) * xcyclic(col'/m+(row-p)/m+L/(2*m))
            d = xcyclic(col'/m+(row-p)/m)
            xcyclic(col'/m+(row-p)/m) = d + c
            xcyclic(col'/m+(row-p)/m+L/(2*m)) = d - c
         end do
      end do
   end do
end do

CYCLIC_PARTITIONED_TO_CENTRALIZED(xcyclic, x, m, psize, p)
end do
```

Fig. 4.5. Plan for Second q Loop with Contiguous Data Access
Plan for Second q Loop with Contiguous Access and Simplified Loop Control

```
do p = 0,m-1
    CENTRALIZED_TO_CYCLIC_PARTITIONED(x,xcyclic,m,psize,p)
do q = breakpoint,t
    L = 2**q
    do row' = p,L/2-1,m
        weightcyclic(row') = EXP((2*pi*i*(m*row'+p))/L)
    end do
    do col'' = 0,psize-1,L/m
        do row' = 0,L/(2*m)-1,1
            c = weightcyclic(row')*xcyclic(col''+row'+L/(2*m))
            d = xcyclic(col''+row')
            xcyclic(col''+row') = d + c
            xcyclic(col''+row'+L/(2*m)) = d - c
        end do
    end do
    CYCLIC_PARTITIONED_TO_CENTRALIZED(xcyclic,x,m,psize,p)
end do
```

**Fig. 4.6. Plan for Second q Loop with Contiguous Data Access and Simplified Loop Control**
Plan for First $q$ Loop

\begin{verbatim}
do p = 0,m-1
  CENTRALIZED_TO_BLOCK_PARTITIONED(x,xblock,m,psize,p)
do q = 1,breakpoint - 1
  L = 2**q
  do row = 0,L/2-1
    weight(row) = EXP((2*pi*i*row)/L)
  end do
  do col' = p*psize,(p+1)*psize-1,L
    do row = 0,L/2-1
      c = weight(row)*xblock(col'+row+L/2-p*psize)
      d = xblock(col'+row-p*psize)
      xblock(col'+row-p*psize) = d + c
      xblock(col'+row+L/2-p*psize) = d - c
    end do
  end do
  BLOCK_PARTITIONED_TO_CENTRALIZED(xblock,x,m,psize,p)
end do
\end{verbatim}

\textit{Fig. 4.7. Plan for First $q$ Loop}
Plan for First q Loop with Simplified Loop Control

```
do p = 0, m-1
   CENTRALIZED_TO_BLOCK_PARTITIONED(x, xblock, m, psise, p)
   do q = 1, breakpoint - 1
      L = 2**q
      do row = 0, L/2-1
         weight(row) = EXP((2*pi*i*row)/L)
      end do
      do col'' = 0, psise-1, L
         do row = 0, L/2-1
            c = weight(row)*xblock(col''+row+L/2)
            d = xblock(col''+row)
            xblock(col''+row) = d + c
            xblock(col''+row+L/2) = d - c
         end do
      end do
   end do
end do
```

Fig. 4.8. Plan for First q Loop with Simplified Loop Control
Combined Plan using Partitioned Data

do p = 0,m-1
    CENTRALIZED_TO_BLOCK_PARTITIONED(x,xblock,m,psize,p)
do q = 1,breakpoint - 1
    L = 2**q
    do row = 0,L/2-1
        weight(row) = EXP((2*pi*i*row)/L)
    end do
do col'' = 0,psize-1,L
    do row = 0,L/2-1
        c = weight(row)*xblock(col''+row+L/2)
        d = xblock(col''+row)
        xblock(col''+row) = d + c
        xblock(col''+row+L/2) = d - c
    end do
end do
BLOCK_PARTITIONED_TO_CENTRALIZED(xblock,x,m,psize,p)
end do

do p = 0,m-1
    CENTRALIZED_TO_CYCLIC_PARTITIONED(x,xcyclic,m,psize,p)
do q = breakpoint,t
    L = 2**q
    do row' = p,L/2-1,m
        weightcyclic(row') = EXP((2*pi*i*(m+row'+p))/L)
    end do
    do col'' = 0,psize-1,L/m
        do row' = 0,L/(2*m)-1,1
            c = weightcyclic(row')*xcyclic(col''+row'+L/(2*m))
            d = xcyclic(col''+row')
            xcyclic(col''+row') = d + c
            xcyclic(col''+row'+L/(2*m)) = d - c
        end do
    end do
end do
CYCLIC_PARTITIONED_TO_CENTRALIZED(xcyclic,x,p,m,psize)
end do

FIG. 4.9. Combined Plan Using Partitioned Data
do q = 1, breakpoint - 1
    L = 2**q
    do row = 0, L/2-1
        weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = myid*psize, (myid+1)*psize-1, L
        do row = 0, L/2-1
            c = weight(row)*x(col'+row+L/2)
            d = x(col'+row)
            x(col'+row) = d + c
            x(col'+row+L/2) = d - c
        end do
    end do
end do
SYNCHRONIZE
do q = breakpoint, t
    L = 2**q
    do row = 0, L/2-1
        weight(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = 0, n-1, L
        do row = myid, L/2-1, m
            c = weight(row)*x(col'+row+L/2)
            d = x(col'+row)
            x(col'+row) = d + c
            x(col'+row+L/2) = d - c
        end do
    end do
end do

**Fig. 5.1.** Partial Per-Processor Parallel Code Template Obtained from Plan of Figure 4.3
Centralized to Block Distribution – Block to Cyclic Redistribution

! DISTRIBUTCENTRALIZEDTOBLOCK(x_myid,m,psize)
if myid = 0 then
  do otherp = 1,m-1
    SEND(x_myid(mysize*otherp),psize,otherp)
  end do
else
  RECEIVE(x_myid(mysize*myid),psize,0)
endif

FIG. 5.2. Centralized to Block Distribution

! DISTRIBUTBLOCKTOCYCLIC(x_myid,m,psize)
do otherp = 0,m-1
  if otherp ≠ myid then
    SEND(x_myid(mysize*myid+otherp:psize*(myid+1)-1:m),psize/m,otherp)
    RECEIVE(x_myid(mysize*otherp+myid:psize*(otherp+1)-1:m),psize/m,otherp)
  endif
end do

FIG. 5.3. Block to Cyclic Redistribution
!  DISTRIBUTE_BLOCK_TO_CYCLIC(x_myid,m,psize)
DISTRIBUTE_BLOCK_TO_CENTRALIZED(x_myid,m,psize)
DISTRIBUTE_CENTRALIZED_TO_CYCLIC(x_myid,m,psize,n)

!  DISTRIBUTE_BLOCK_TO_CENTRALIZED(x_myid,m,psize)
if myid = 0 then
  do otherp = 1,m-1
    RECEIVE(x_myid(psize*otherp),psize,otherp)
  end do
else
  SEND(x_myid(psize*myid),psize,0)
endif

!  DISTRIBUTE_CENTRALIZED_TO_CYCLIC(x_myid,m,psize,n)
if myid = 0 then
  do otherp = 1,m-1
    SEND(x_myid(otherp:n-1:m),psize,otherp)
  end do
else
  RECEIVE(x_myid(myid:n-1:m),psize,0)
endif

FIG. 5.4. Block to Cyclic Redistribution Using Centralized Site as Intermediary
Cyclic to Central Redistribution

!  DISTRIBUT_CYCLIC_TO_CENTRALIZED(x_{myid}, m, psize, n)
if myid = 0 then
  do otherp = 1, m-1
    RECEIVE(x_{myid}(otherp:n-1:m), psize, otherp)
  end do
else
  SEND(x_{myid}(myid:n-1:m), psize, 0)
endif

**Fig. 5.5. Cyclic to Centralized Redistribution**
DISTRIBUTE CENTRALIZED TO BLOCK(x_myid, m, psize)
 do q = 1, breakpoint - 1
   L = 2**q
   do row = 0, L/2-1
     weight_myid(row) = EXP((2*pi*i*row)/L)
   end do
   do col' = myid*psize, (myid+1)*psize-1, L
     do row = 0, L/2-1
       c = weight_myid(row)*x_myid(col'+row+L/2)
       d = x_myid(col'+row)
       x_myid(col'+row) = d + c
       x_myid(col'+row+L/2) = d - c
     end do
   end do
 end do
DISTRIBUTE BLOCK TO CYCLIC(x_myid, m, psize)
 do q = breakpoint, t
   L = 2**q
   do row = 0, L/2-1
     weight_myid(row) = EXP((2*pi*i*row)/L)
   end do
   do col' = 0, n-1, L
     do row = myid, L/2-1, m
       c = weight_myid(row)*x_myid(col'+row+L/2)
       d = x_myid(col'+row)
       x_myid(col'+row) = d + c
       x_myid(col'+row+L/2) = d - c
     end do
   end do
 end do
DISTRIBUTE CYCLIC TO CENTRALIZED(x_myid, m, psize, n)

Fig. 5.6. Per-Processor Distributed Code Template Based on Partial Template of Figure 5.1
Distribute centralized to block partitioned ($x_0$, $x_{block_{mid}}, m, psize$

```c
do q = 1, breakpoint - 1
  L = 2**q
  do row = 0, L/2 - 1
    weight_{mid}(row) = EXP((2*pi*i*row)/L)
  end do
  do col = 0, psize - 1, L
    do row = 0, L/2 - 1
      c = weight_{mid}(row) * $x_{block_{mid}}(col' + row + L/2)
      d = $x_{block_{mid}}(col' + row)
      $x_{block_{mid}}(col' + row) = d + c
    end do
  end do
end do
```

Distribute block partitioned to centralized ($x_{block_{mid}}, x_0, m, psize$

```c
Distribute centralized to cyclic partitioned ($x_0$, $x_{cyclic_{mid}}, m, psize$

```c
```
```c
do q = breakpoint, t
  L = 2**q
  do row' = 0, L/(2*m) - 1, 1
    weight_{cyclic_{mid}}(row') = EXP((2*pi*i*(m*row'+midid))/L)
  end do
  do col' = 0, psize - 1, L/m
    do row' = 0, L/(2*m) - 1, 1
      c = weight_{cyclic_{mid}}(row') * $x_{cyclic_{mid}}(col' + row' + L/(2*m))
      d = $x_{cyclic_{mid}}(col' + row')
      $x_{cyclic_{mid}}(col' + row') = d + c
    end do
  end do
end do
```
```
```
Distribute cyclic partitioned to centralized ($x_{cyclic_{mid}}, x_0, m, psize$

```

**Fig. 5.7.** Per Processor Distributed Code Template Obtained from Combined Plan Using Partitioned Data of Figure 4.9
Code for Distribution Command occurring in Figure 5.7s per processor Template using Partitioned Data

! DISTRIBUTED_TO_BLOCK_PARTITIONED(x0,xblock_myid,m,psize)
if myid = 0 then
    xblock_myid = x0(0:psize-1)
    do otherp = 1,m-1
        SEND(x0(psise+otherp),psize,otherp)
    end do
else
    RECEIVE(xblock_myid,psize,0)
endif

! DISTRIBUTED_TO_BLOCK_PARTITIONED(xblock_myid,x0,m,psize)
if myid = 0 then
    x0(0:psize-1) = xblock_myid
    do otherp = 1,m-1
        RECEIVE(x0(psise+otherp),psize,otherp)
    end do
else
    SEND(xblock_myid,psize,0)
endif

! DISTRIBUTED_TO_CIRCULAR_PARTITIONED(x0,xcyclic_myid,m,psize)
if myid = 0 then
    xcyclic_myid = x0(0:n-1:m)
    do otherp = 1,m-1
        SEND(x0(otherp:n-1:m),psize,otherp)
    end do
else
    RECEIVE(xcyclic_myid,psize,0)
endif

! DISTRIBUTED_TO_CIRCULAR_PARTITIONED(xcyclic_myid,x0,m,psize)
if myid = 0 then
    x0(0:n-1:m) = xcyclic_myid
    do otherp = 1,m-1
        RECEIVE(x0(otherp:n-1:m),psize,otherp)
    end do
else
    SEND(xcyclic_myid,psize,0)
endif

Fig. 5.7. Code for Distribution Commands Occurring in Figure 5.7s Per-Processor Template Using Partitioned Data
! DISTRIBUTEBLOCK_PARTITIONED_TO_CYCLIC_PARTITIONED(xblock$_{\text{myid}}$, 
\hspace{1cm} x\text{cyclic}$_{\text{myid}}$, m, psize) 

= \hspace{1cm} x\text{cyclic}$_{\text{myid}}$(myid*psize/m: (myid+1)*psize/m-1) 
xblock$_{\text{myid}}$(myid: psize-1:m) 
do otherp = 0, m-1 
\hspace{1cm} if otherp \neq \text{myid} then 
\hspace{2cm} \text{SEND}(xblock$_{\text{myid}}$(otherp: psize-1:m), psize/m, otherp) 
\hspace{2cm} \text{RECEIVE}(x\text{cyclic}$_{\text{myid}}$(otherp*psize/m), psize/m, otherp) 
\hspace{1cm} endif 
end do 

Fig. 5.9. Block-Partitioned to Cyclic-Partitioned Redistribution
Simple Shared Memory All Processor Template obtained from Plan of Figure 4.3

parallel do p = 0,m-1
  do q = 1,breakpoint - 1
    L = 2**q
    do row = 0,L/2-1
      weight_p(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = p*psize,(p+1)*psize-1,L
      do row = 0,L/2-1
        c = weight_p(row)*x(col'+row+L/2)
        d = x(col'+row)
        x(col'+row) = d + c
        x(col'+row+L/2) = d - c
      end do
    end do
  end do
end parallel do

parallel do p = 0,m-1
  do q = breakpoint,t
    L = 2**q
    do row = 0,L/2-1
      weight_p(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = 0,n-1,L
      do row = p,L/2-1,m
        c = weight_p(row)*x(col'+row+L/2)
        d = x(col'+row)
        x(col'+row) = d + c
        x(col'+row+L/2) = d - c
      end do
    end do
  end do
end parallel do

Fig. 6.1. Simple Shared Memory All-Processor Template Obtained from Plan of Figure 4.3
All processor Shared and Processor Memory Template based on Template from Figure 6.1

```
parallel do p = 0,m-1
  COPY CENTRALIZED TO PRIVATE BLOCK(x,x,p,psize)
  do q = 1,breakpoint - 1
    L = 2**q
    do row = 0,L/2-1
      weight_p(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = p*psize,(p+1)*psize-1,L
      do row = 0,L/2-1
        c = weight_p(row)*x_p(col'+row+L/2)
        d = x_p(col'+row)
        x_p(col'+row) = d + c
        x_p(col'+row+L/2) = d - c
      end do
    end do
  end do
  COPY_PRIVATE_BLOCK_TO_CENTRALIZED(x_p,x,p,psize)
end parallel do
parallel do p = 0,m-1
  COPY CENTRALIZED TO PRIVATE CYCLIC(x,x,p,m,psize)
  do q = breakpoint,t
    L = 2**q
    do row = 0,L/2-1
      weight_p(row) = EXP((2*pi*i*row)/L)
    end do
    do col' = 0,n-1,L
      do row = p,L/2-1,m
        c = weight_p(row)*x_p(col'+row+L/2)
        d = x_p(col'+row)
        x_p(col'+row) = d + c
        x_p(col'+row+L/2) = d - c
      end do
    end do
  end do
  COPYPRIVATE CYCLIC TO CENTRALIZED(x_p,x,p,m,psize)
end parallel do
```

**FIG. 6.2.** All-Processor Shared and Private Memory Template Based on Template of Figure 6.1
Data Copying for All-Processor Shared and Private Memory Template of Figure 6.2

```
! COPY_CENTRALIZED_TO_PRIVATE_BLOCK(x,x_p,p,psize)
x_p(psige*p:psize*(p+1)-1:1) = x(psige*p:psize*(p+1)-1:1)

! COPY_PRIVATE_BLOCK_TO_CENTRALIZED(x_p,x,p,psize)
x(psige*p:psize*(p+1)-1:1) = x_p(psige*p:psize*(p+1)-1:1)

! COPY_CENTRALIZED_TO_PRIVATE_CYCLIC(x,x_p,p,m,psize)
x_p(p:n-1:m) = x(p:n-1:m)

! COPY_PRIVATE_CYCLIC_TO_CENTRALIZED(x_p,x,p,m,psize)
x(p:n-1:m) = x_p(p:n-1:m)
```

Fig. 6.3. Data Copying for All-Processor Shared and Private Memory Template of Figure 6.2
parallel do p = 0,m-1
  COPY.CENTERIALIZED_TO_PRIVATE_BLOCK_PARTITIONED(x,xblock_p,m,psize,p)
  do q = 1,breakpoint - 1
    L = 2**q
    do row = 0,L/2-1
      weight_p(row) = EXP((2*pi+i)row)/L
    end do
    do col'' = 0,psize-1,L
      do row = 0,L/2-1
        c = weight_p(row) + xblock_p(col''+row+L/2)
        d = xblock_p(col''+row)
        xblock_p(col''+row) = d + c
      end do
    end do
  end do
end parallel do

parallel do p = 0,m-1
  COPY.CENTERIALIZED_TO_PRIVATE_CYCLIC_PARTITIONED(x,xcyclic_p,m,psize,p)
  do q = breakpoint,t
    L = 2**q
    do row' = 0,L/(2*m)-1,1
      weightcyclic_p(row') = EXP((2*pi+i)*m*row'+p))/L
    end do
    do col'' = 0,psize-1,L/m
      do row' = 0,L/(2*m)-1,1
        c = weightcyclic_p(row') + xcyclic_p(col''+row'L/(2*m))
        d = xcyclic_p(col''+row')
        xcyclic_p(col''+row'L/(2*m)) = d + c
      end do
    end do
  end do
end parallel do

Fig. 4.4. All-Processor Shared Memory and Private Partitioned Data Template Obtained from Combined Plan of Figure 4.9
! COPY_CENTRALIZED_TO_PRIVATE BLOCK PARTITIONED(x, xblock_p, p, psize)
xblock_p = x(psize*p:psize*(p+1)-1:1)

! COPY_PRIVATE BLOCK PARTITIONED TO_CENTRALIZED(xblock_p, m, psize, p)
x(psize*p:psize*(p+1)-1:1) = xblock_p,

! COPY_CENTRALIZED_TO_PRIVATE CYCLIC PARTITIONED(x, xcyclic_p, m, psize, p, n)
xcyclic_p = x(p:n-1:m)

! COPY_PRIVATE CYCLIC PARTITIONED TO_CENTRALIZED(xcyclic_p, x, m, psize, p, n)
x(p:n-1:m) = xcyclic_p

Fig. 6.5. Data Copying for All-Processor Shared Memory and Private Partitioned Data Template of Figure 6.4
• Primarily a manual effort to design the mapping of the computation to an abstract model
  – This step is crucial if we are to make effective use of exascale architectures. Everything we just saw is “mechanizable”.
  • Not just FFT but FFT type of accesses. Access patterns are simple variants of a theme.
    – Haar Wavelet and other wavelets
    – Finite difference

• Abstract model for the organization of processors is often in the form of a graph whereby the nodes 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 step away along one of the axes.
  – Array model can describe a list of processors, a two-dimensional mesh, a hypercube, a balanced tree, or a network of workstations.
• View an architecture as having two array organizations.
  – One is the abstract model suited for describing the problem.
  – Second, corresponds to the list of processor identity numbers and is used to determine the actual send and receive instructions issued by the resulting program. Moreover, each level of memory, bandwidths, etc., become higher dimensions in this abstraction.
  – 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 discussed a specific formalism, the Psi Calculus, and we demonstrated 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.
We see this happening in three directions:

- by a language providing a high level notation embedded within a standard language, e.g., HPF, ZPL, Fortran, Matlab, ... that is preprocessed using the techniques described in the paper.
  - An enhanced subset that is 1-1 and onto semantically.

- 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.
• **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.
  - Chunks may be contiguous or 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.
    • At shared and distributed memory we think of block or cyclic distributions.
    • All of these movements will be handled by the PCA’s middleware which conceptually subsumes the work typically done in a compiler, OS, and firmware.
Definition 8.1. Let $P_p = \langle \text{shape},^{12} \text{speed} \rangle$ where $p = 1, np$, and $np$ = number of different processors available in the PCA. For each $P_p \exists$ an instruction set $I_{pj} = \langle op, na \rangle$ where $j = 1, nfu$, and $nfu$ = number of functional units supported by processor $P_p$. $op$ denotes an instruction and $na$, the number of arguments required.

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

8.1.2. Example

$P_0 = @\text{Intel}, P_1 = @\text{IBM}, P_2 = @\text{SGI}$, etc.
On an Abstract Machine: Thoughts and Requirements

Definition 8.2. Let \( p, M_{m_1} = (\text{shape, speed}), m = 1, nm \), where \( nm \) = number of levels in the memory hierarchy numbered from lowest to highest, \( t = 1, nt \), where \( nt \) = number of types of \( p, M_{m_1} \) memory. The \( p, M_{m_1} \)'s include registers, cache, local memory, page 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. \( \text{shape}[0] = \text{the dimensionality of } p, M_{m_1}: 1 = \text{vector}, 2 = \text{matrix}, \text{etc.} \)
2. \( \text{shape}[0] \Delta (1 \lor \text{shape}) = \text{shape of } p, M_{m_1}. \)

Hence, \( (- (\text{shape}[0] - 1)) \Delta \text{shape} = \text{shape of each } p, M_{m_1} \) component and \( \pi (\text{shape}[0] \Delta (1 \lor \text{shape})) = \text{the total number of components in } p, M_{m_1}. \)

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

\[ p, M_{O_0} = \langle (1 16), \text{speed} \rangle \]  \hspace{1cm} (84)

and

\[ p, M_{O_1} = \langle (2 16 8), \text{speed} \rangle, \]  \hspace{1cm} (85)

respectively. Hence, in the register memory defined by expression (84):

\( \text{shape} = (1 16): \text{shape}[0] = 1 \), which is the dimensionality of \( p, M_{O_0} \).
\( \text{shape}[0] \Delta (1 \lor \text{shape}) = 1 \Delta (1 \lor (1 16)) = 1 \Delta (16) = (16) \), which is the shape of \( p, M_{O_0} \).
\( - (\text{shape}[0] - 1)) \Delta \text{shape} = (- (1 - 1)) \Delta (1 16) = 0 \Delta (1 16) = \Theta. \) That is, the shape of each component is the empty vector, i.e., each component is a scalar. Consequently, we have 16 scalar registers.
\( \pi (\text{shape}[0] \Delta (1 \lor \text{shape})) = \pi (1 \Delta (1 \lor (1 16))) = \pi (1 \Delta (16)) = \pi (16) = 16 \), the total number of components in \( p, M_{O_0} \).
Caches, \( p_p M_1 \), 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.

Classes of Algorithms

- Not skeletons
- Not mini-apps
- Classes of algorithms: I observed and Van Loan validated.
- Fast Transforms (FFT, Haar Wavelet), Preconditioners, data sparse approximate factorizations, and others.
  - Algorithms can be parameterized easily.
  - Study of multiple Kronecker products emphasized the need for an array machine operation, the outer product plus, that works on integers to calculate addresses in parallel.
  - Parallel address calculation kept coming up in ONFs over the years.
Pattern of index vector

\[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
256 & 257 & 258 & 259 \\
512 & 513 & 514 & 515 \\
768 & 769 & 770 & 771 \\
4 & 5 & 6 & 7 \\
260 & 261 & 262 & 263 \\
516 & 517 & 518 & 519 \\
772 & 773 & 774 & 775 \\
8 & 9 & 10 & 11 \\
264 & 265 & 266 & 267 \\
520 & 521 & 522 & 523 \\
776 & 777 & 778 & 779 \\
12 & 13 & 14 & 15 \\
268 & 269 & 270 & 271 \\
524 & 525 & 526 & 527 \\
780 & 781 & 782 & 783 \\
1024 & 1025 & 1026 & 1027 \\
1280 & 1281 & 1282 & 1283 \\
1536 & 1537 & 1538 & 1539 \\
1792 & 1793 & 1794 & 1795 \\
1028 & 1029 & 1030 & 1031 \\
1284 & 1285 & 1286 & 1287 \\
1540 & 1541 & 1542 & 1543 \\
1796 & 1797 & 1798 & 1799 \\
1032 & 1033 & 1034 & 1035 \\
1288 & 1289 & 1290 & 1291 \\
1544 & 1545 & 1546 & 1547 \\
1800 & 1801 & 1802 & 1803 \\
1036 & 1037 & 1038 & 1039 \\
1292 & 1293 & 1294 & 1295 \\
1548 & 1549 & 1550 & 1551 \\
1804 & 1805 & 1806 & 1807 \\
2048 & 2049 & 2050 & 2051 \\
\end{array}
\]
Conclusions

• Domain specific languages, especially matrix, array, or tensor based, have a chance of “total” mechanization.
  – Write once code.
  – Portable and Scalable
  – Verifiable
  – Analyzable
  – Array operations map easily to an array view of the processor memory architectures and to the underlying machine instructions, array operations.
  – Studies indicate a need for an array operation on integers to calculate addresses.
  – Studies also indicate the need to access the length of vector/array registers.
  – Such experiences and insights could influence the design of future hardware if systems programmers, theorists, and engineers worked collectively. It has worked for me over the years.
Thank You

Lenore M. Mullin
lenore.mullin@science.doe.gov