CODE GENERATION OF ARRAY CONSTRUCTS FOR DISTRIBUTED MEMORY SYSTEMS

BY

SWETA YAMINI POTHUKUCHI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

ABSTRACT

Programming for high performance systems to fully utilize the potential of the compute system is a complex problem. This is particularly evident when programming distributed memory clusters containing multiple NUMA chips and GPUs on each node since it would require a complex combination of MPI, OpenMP, CUDA, OpenCL, etc to achieve high performance even for sequentially simplistic codes. Programs requiring high performance are usually painstakingly written by hand in C/C++ or Fortran using MPI+X to target these machines.

This work presents a multi-layer code generation framework *Vaani* that takes a very high-level representation of computations, and generates C+MPI code by transforming the input through a series of intermediate representations. The very high level nature of the language greatly facilitates programming parallel systems. Additionally, the use of multiple representations provide a flexible and transparent venue for the user to interact and customize the transformation process to generate code suitable to the user and the target machine.

Experimental evaluation shows that the current implementation of *Vaani* generates code that is competitive with handwritten codes and hand-optimized libraries.

To my family, for their love and support.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser Professor David Padua, for his support, guidance and patience. He has given me the freedom to pursue my interests, and provided invaluable insights while he guided my research. Also, I could not have completed this thesis without his infinite tolerance. Having two children while pursuing a Ph.D. is difficult and almost impossible without a supportive adviser. He has given me leeway and has been very understanding of my responsibilities and choices as a mother, as I tried not to alienate my work or my children.

I would like to express gratitude to Professor Laxmikant Kale, Professor Andreas Klöckner and Professor Franz Franchetti for being on my dissertation committee and for their critical feedback that helped shape my research.

I would like to thank all my friends here at University of Illinois at Urbana-Champaign (UIUC) for being with me, supporting me and having fun together while we cruise through graduate school. I would also like to thank my old friends Manisha, Vijetha, Rasha, Aghamarshan and Bharadwaj for being with me through all these years.

I would like to thank Anine Singh for rekindling my passion for dance, and Miss Sharon, Miss Kayla and Miss Monic for taking care of my son Srirama, while I was at work.

I cannot express in words the gratitude I have for my family for their love, care and support throughout my Ph.D. I cannot fathom the sacrifices my mother made to reach out and support me in times of need, and I wonder if I could have done the same in her place. I can never forget the trouble my father had to go through to take care of the children during a pandemic that wrecked havoc across the globe. I feel very lucky to have parents-in-law who are supportive of my work, who took time out of their busy schedules, and came to the United States to help me and my husband work on our research, even while they had to battle their own health issues. My husband Raghavendra has been a pillar of support through thick and thin, and I can't thank him enough for the troubles he put himself through to give me time to finish my dissertation. I would like to thank my brother Raghu Teja, his wife Harsha Sree, my brother-in-law Viswanatha Srinivas and his wife Aishwarya, for their love, support and the fun times we spent together. I would like to thank my grandmother for her help in taking care of my children. At her age, it was no mean task.

I owe this dissertation to my two children, Srirama Prahlada and Ameya Dakshayani, who were born during my Ph.D. My son has been very patient, caring and understanding, always waiting for me to find time to spend with him. It astonishes me how maturely he has handled the situation as a four year old, and I am extremely proud of him. I can't forget the cries of my daughter, as I shut the door to get some time to work.

I would like to thank my teachers at school, college, Indian Institute of Technology (IIT) Kharagpur, and UIUC, who have taught me invaluable lessons and shaped me into who I am today.

Lastly, I would like to thank the wheel of time to weave my life the way it willed, for making me come to this point in time through a million little nudges.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Array computations form the crux of many science and engineering applications, and can be succinctly represented in a very high-level notation. However, these codes are written today in C/C++ or Fortran for distributed memory systems using MPI. Supercomputers have different types of nodes and interconnects, and program optimization needs to be tailored to suit each set of target machines. The selection of algorithms, data layout, mapping and optimizations are interdependent and modifying one of them may typically require a complete rewrite of the application.

The following sections describe the representation of computations in high level notation and in C using MPI.

1.1 ARRAY REPRESENTATION OF COMPUTATIONS

A significant portion of scientific and high performance computing use multi-dimensional arrays as primary data structures, and perform operations on the arrays that can be represented with an array view of the structure, as opposed to the scalar, element-by-element view provided in C/C++ or Fortran. These array representations are succinct, clear and easy to maintain and modify. The programmer intent and the core computation specification is available in this representation. Figure 1.1 shows a simple matrix multiplication written in MATLAB (left side) and C (right side). Straightforward C code is already much longer than the corresponding MATLAB code, but for high performance, it needs to be even longer since matrix multiplication benefits from two levels of blocking (possibly multiple levels of blocking), loop unrolling, vectorization and other optimizations, which tend to obfuscate the code. On the other hand, in C, we know exactly how the data is stored and accessed, possible

```
C = A*B
```

```
for(int i = 0; i < m; i++) {
    for(int j = 0; j < n; j++) {
        C[i][j] = 0;
        for(int k = 0; k < r; k++) {
            C[i][j] += A[i][k]*B[k][j];
        }
    }
}
```

Figure 1.1: Matrix multiplication in MATLAB and C

optimizations to perform, and their impact on performance. MATLAB does everything under the hood, and the user typically does not have much control over the performance.

1.2 CODE FOR DISTRIBUTED MEMORY SYSTEMS

Single Program Multiple Data (SPMD) programming style using MPI with C/C++ or Fortran is the most commonly adopted style to program for distributed memory systems. This requires the data to be manually partitioned across the processes, and communication needs to be explicitly specified using sends and receives for inter-process communication. The already long C code from Figure 1.1 becomes significantly more complex with explicit partitioning and data transfers as shown in Figure 1.2. It must also be noted that this example is already a simplified version, that make use of a simple algorithm (Cannon's algorithm) and assumes square matrices on a square grid. The choice of algorithms affect performance, and in this case, we can have 1D, 2D or 3D partitioning, with different algorithm choices in each of these partitioning schemes, where different matrices need to be communicated. For example, in 2D partitioning, we can keep either A, B or C in place, and move the other two matrices around to perform the computation. Again, the moving matrices can be broadcast step-by-step or moved around cyclically, as in Cannon's algorithm.

In this setting, the programmer must spend time in low level details, like index computations, buffer management, etc. in addition to the time consumed by the high level design decisions like the choice of partitioning, the choice of optimizations, etc. Further, modifying the code from one set of partitioning and optimization choices to another requires a complete rewrite of the program. The problem is exacerbated if we add the conventional optimizations mentioned above, like tiling, unrolling and vectorization to the MPI version.

Hybrid programming, combining distributed memory MPI models with shared memory paradigms like OpenMP are frequently used to find the optimum performance. MPI needs explicit communication, and can be expensive to use within the same node, while OpenMP threads are light weight. Computations which require data replication on each process benefit from this model, for example, stencil computations with ghost layers. Again, this adds another layer of optimizations, and thus complicates the design and implementation of programs significantly.

```
// Initial shuffle
MPI_Irecv(Ain[0], myrc*myrc, MPI_DOUBLE,
          (col+row)%pp, pp, commRow, &req[0]);
MPI_Isend(Aout[0], myrc*myrc, MPI_DOUBLE,
          (col+pp-row)%pp, pp, commRow, &req[1]);
MPI_Irecv(Bin[0], myrc*myrc, MPI_DOUBLE,
          (row+col)%pp, pp, commCol, &req[2]);
MPI_Isend(Bout[0], myrc*myrc, MPI_DOUBLE,
          (row+pp-col)%pp, pp, commCol, &req[3]);
MPI_Waitall(4, req, sts);

for(int l = 0; l < pp; l++) {
    swapMatrix(&Ain, &Aout);
    swapMatrix(&Bin, &Bout);

    for (int i = 0; i < myrc; i++) {
        for (int j = 0; j < myrc; j++) {
            for (int k = 0; k < myrc; k++) {
                Cl[i][j] += Aout[i][k] * Bout[k][j];
            }
        }
    }

    MPI_Irecv(Ain[0], myrc*myrc, MPI_DOUBLE,
              (col+1)%pp, l, commRow, &req[0]);
    MPI_Isend(Aout[0], myrc*myrc, MPI_DOUBLE,
              (col+pp-1)%pp, l, commRow, &req[1]);

    MPI_Irecv(Bin[0], myrc*myrc, MPI_DOUBLE,
              (row+1)%pp, l, commCol, &req[2]);
    MPI_Isend(Bout[0], myrc*myrc, MPI_DOUBLE,
              (row+pp-1)%pp, l, commCol, &req[3]);

    MPI_Waitall(4, req, sts);
}
```

Figure 1.2: Matrix multiplication in C+MPI using Canon’s algorithm

1.3 OVERVIEW OF THE IDEAS IN THIS DISSERTATION

The very high-level notation used in this thesis provides a clear and succinct representation of a computation that is easy to maintain and modify. However, this lacks the flexibility to decide the placement of data and its corresponding optimizations. The main idea of this dissertation is to define a high level array notation language that can be used to create high performance code by following through a series of intermediate representations that lower the high level input to optimized C code with MPI. The layered approach provides an intuitive and clear transformation path, providing the users with an interface to transform the code based on high-level decisions on how the computation must be partitioned and mapped, the set of optimizations or strategies to adopt, without having to deal with the low-level details such as the generation of actual send and receive calls, index manipulations, etc.

To this effect, we define four intermediate representations (IR). High Level IR (HLIR) represents computations on arrays using a set of predefined operators. At this level, our system performs optimizations using symbolic computation and simplification. Mid level IR (MLIR) represents computations based on the data access patterns. During the translation from HLIR to MLIR, operations are categorized as Map, Stencil, Reduce/Scan, Multiply, etc. Low level IR (LLIR) represents local computations and communication, after data and computations are partitioned onto a virtual process grid. This level still retains a high level notation of the computation and communication. C level IR (CLIR) takes this one step further and represents the computation and communication in near C notation, using loops with index sets and instruction blocks; and lower level communication nodes.

We provide an interactive interface to the intermediate representations to lower a computation specified in the high level notation to MPI+C code. We also provide a user-guided tuning approach to tune the generated code to a target machine, by allowing the user to select parameterized optimizations and partitioning, and using autotuning to select the best values for a given input and machine size.

1.4 CONTRIBUTIONS

This dissertation introduces a framework called *Vaani* to generate efficient distributed memory code from high level specifications, while providing flexibility and transparency to the internal transformations in the process of generating the final code. This work proposes a layered approach to code transformation tailored for distributed memory systems that provides an interactive interface to generate efficient code. The main contributions of this dissertation are:

1. Design of a high-level array notation language to represent computations, particularly recurrences and stencil computations

2. Design of a series of intermediate representations to provide an interface to code generation

3. Development of a scripting language strategy to generate code from a high-level specification

1.5 THESIS ORGANIZATION

The rest of the thesis is organized as follows. Chapter 2 details the high level input language to the framework *Vaani*. Chapter 3 provides detailed description of the intermediate representations used in *Vaani*, and the rationale for choosing them. Chapter 4 discusses the compilation process and code generation. Chapter 5 describes the process of generating a program using *Vaani*. Chapters 6 and 7 describe the experimental setup and the evaluation of *Vaani* respectively. Chapter 8 explains the related work. And finally, chapters 9 and 10 explore future possibilities and conclude the thesis.

CHAPTER 2: HIGH LEVEL LANGUAGE DESIGN

Vaani uses a very high-level array notation with a strong typing system to specify computations as input to the system. The language design takes ideas from existing languages that support array operations, like MATLAB [1] and NumPy [2], but does not adopt either of them as is, to provide a coherent notation that is succinct, clear and expressive. Figure 2.1 has a few examples to demonstrate the syntax and structure of the language.

GEMVER (Figure 2.1a) is a program that takes a matrix A of size $m \times n$, three column vectors $u1$, $u2$, and y of size m (interpreted as $m \times 1$), three column vectors $v1$, $v2$ and z of size n (interpreted as $n \times 1$), and two scalars $alpha$ and $beta$ as inputs and gives three outputs A, x and w. Here, A is specified as inout, indicating it is both an input and an output to the system. A is updated with two outer products specified by the expressions $u1 * v1'$ and $u2 * v2'$, where the '$'$' indicates a transpose. Then x is computed by performing a scaled matrix-vector multiplication with the matrix A transposed and the input vector y, and added to the input vector z. w is again computed by a scaled matrix-vector product.

SSSP (Figure 2.1b) is a combination of matrix-power and matrix-vector multiplication with custom additive and multiplicative operators. Here, a square matrix A of size $n \times n$, a column vector x of size $n \times 1$ are taken as inputs, and $sssp$ is declared as an output. $sssp$ is computed using a custom power operator '^' where the additive operator is min and the multiplicative operator is $+$. Here the matrix A is raised to the power of n with these custom operators, and again multiplied to x with the same custom operators as specified by the $*(min, +)$. This application, when provided with an adjacency matrix A where each element $A[i, j]$ indicates the cost between nodes i and j if a path exists, and a very large number (that can be treated as infinity in this context) otherwise, and the vector x has the same very large number all over, except one node k that has a 0, then the vector $A^n x$ gives the shortest path costs from the node k [3].

Activation (Figure 2.1c) represents two commonly used activation functions relu and sigmoid in deep neural networks. Here, the functions max and exp are maximum and exponential respectively. These functions and the other operators are applied to each element of the argument/operand matrices.

Blur (Figure 2.1d) is a 3×3 box filter represented as two 3×1 and 1×3 filters in the x and y dimensions. Here, it takes a matrix A as input and gives a matrix C as the output, where the intermediate matrix B is a 3-point stencil computation in the first dimension, and the final result is a 3-point stencil computation in the y direction. Here, the curly brackets are used to indicate the relative offsets of the stencil.

```
program GEMVER
inout A matrix(m, n, real64)
in u1, u2, y cvector(m, real64)
in v1, v2, z cvector(n, real64)
in alpha, beta scalar(real64)
out x, w

A = A + u1*v1' + u2*v2'
x = beta*A'*y + z
w = alpha*A*x
```

(a) GEMVER: BLAS Level 2

```
program SSSP
in A matrix(n, real64)
in x cvector(n, real64)

out sssp


sssp = A^(min, +)n *(min, +) x
```

(b) SSSP: Matrix power with custom operators

```
function Activation
in input matrix(b, m, real64)
out relu, sigmoid

relu = max(input, 0)
sigmoid = 1/(1 + exp(-input))
```

(c) Activation: Some activation functions

```
function Blur
in A matrix(m, n, real64)
out C

B = (A{-1,0} + A + A{1,0})/3
C = (B{0,-1} + B + B{0,1})/3
```

(d) Blur: 3×3 blur as 3×1 and 1×3 passes

```
program Jacobi2D
in A matrix(m, n, real64)
in iter scalar(int32)
out B

B = rec B [iter] {
  B = (B{-1} + B{-1;0,1}
      + B{-1;0,-1} + B{-1;1,0}
      + B{-1;-1,0})/5
      with boundary=periodic
} with B[0] = A
```

(e) Jacobi2D: Jacobi stencil computation

```
program Gauss2D
in A matrix(m, n, real64)
in iter scalar(int32)
out B

B = rec B [iter] {
  B = (B{-1} + B{-1;0,1}
      + B{0;0,-1} + B{-1;1,0}
      + B{0;-1,0})/5
      with boundary=none
} with B[0] = A
```

(f) Gauss2D: Gauss-Siedel stencil computation

Figure 2.1: Example specification in *Vaani*

Jacobi2D (Figure 2.1e) is a two dimensional Jacobi 5-point stencil computation for a fixed number of iterations, while Gauss2D (Figure 2.1f) is a similar computation using Gauss-Siedel iterations. Here, both programs take a matrix A of size $m \times n$ and a scalar integer *iter* as inputs. Matrix B is declared to be the output of the system. These programs use a novel *recurrence* construct to specify recurrences to compute new values of B. The recurrence of B is initialized with the matrix A in the with clause $B[0] = A$, and the value of B in the i^{th} iteration is computed from the B in $(i-1)^{th}$ iteration by using the assignment statement in the body of the recurrence. Here, the numbers in the curly braces are divided by a semicolon, and the first number indicates a temporal offset (offset in the iteration space of the recurrence), while the ones following the ';' indicate spatial offsets (offsets in the iteration space of the array). It can be observed that to create a Gauss-Siedel iteration, one needs to change the temporal offset for some of the terms from -1 to 0 to indicate current iteration. Here, the 0 in $B\{0;0,-1\}$ and $B\{0;-1,0\}$ together create a Gauss-Siedel expression from the top-left to the bottom-right corner of the matrix. Also, the *boundary* clause in Jacobi2D (Figure 2.1e) indicates that a periodic boundary condition is applied to the computation, while that in Gauss2D (Figure 2.1f) indicates that the boundary is ignored.

Vaani generates code in the form of either a stand alone complete program, reading inputs from and writing outputs to files, or as functions which assume an MPI environment already running and the inputs and outputs spread across the process grid as specified in the code generation phase. For the functions, *Vaani* generates setup and teardown functions, and a function to perform the computation. These C functions can be called from another C program, such that the setup function must be called first, and then its output is an input to the computation function, which can be called multiple times, and then the teardown function is called. In Figure 2.1, GEMVER, SSSP, Jacobi2D and Gauss2D are programs, whereas Blur and Activation are functions, as identified by the first keyword `program` or `function`.

Vaani supports element-by-element operations, stencil operations, matrix products and powers, transpose, reductions, rearrangements and recurrences. It also supports user declared and user defined functions. The following sections describe their syntax and semantic behavior in detail.

2.1 GRAMMAR

Figure 2.2 presents *Vaani*'s grammar. The terms in angular brackets (like ⟨*program*⟩) are non-terminal symbols, characters in quotes (like '`program`' and '`:`') are keywords and accepted symbols, and capital character strings (like `ID` and `CONSTANT`) are terminal symbols.

⟨*program*⟩ ::= ⟨*header*⟩ ⟨*declaration*⟩+ ⟨*statement*⟩+

⟨*header*⟩ ::= ‘`program`’ [`ID`]
  | ‘`function`’ `ID`

⟨*declaration*⟩ ::= ‘`in`’ `ID` (‘`,`’ `ID`)* ⟨*objtype*⟩
  | ‘`inout`’ `ID` (‘`,`’ `ID`)* ⟨*objtype*⟩
  | ‘`out`’ `ID` (‘`,`’ `ID`)* [⟨*objtype*⟩]

⟨*objtype*⟩ ::= ‘`scalar`’ ‘`(`’[⟨*datatype*⟩] ‘`)`’
  | [‘`vector`’ | ‘`cvector`’ | ‘`rvector`’] ‘`(`’ ⟨*expr*⟩ [‘`,`’ ⟨*datatype*⟩] ‘`)`’
  | ‘`matrix`’ ‘`(`’ ⟨*expr*⟩ [‘`,`’ ⟨*expr*⟩] [‘`,`’ ⟨*datatype*⟩] ‘`)`’
  | ‘`tensor`’ ‘`(`’ [⟨*datatype*⟩] ‘`)`’
  | ‘`tensor`’ ‘`(`’ ⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)* [‘`,`’ ⟨*datatype*⟩] ‘`)`’

⟨*datatype*⟩ ::= ‘`bool`’ | ‘`int8`’ | ‘`uint8`’ | ‘`int16`’ | ‘`uint16`’
  | ‘`int32`’ | ‘`uint32`’ | ‘`int64`’ | ‘`uint64`’
  | ‘`real32`’ | ‘`real64`’ | ‘`complex64`’ | ‘`complex128`’

⟨*statement*⟩ ::= ⟨*assignment*⟩ | ⟨*recurrence*⟩ | ⟨*function*⟩

⟨*assignment*⟩ ::= ⟨*lhs*⟩ (‘`,`’ ⟨*lhs*⟩)* ‘`=`’ ⟨*expr*⟩ (‘`,`’ ⟨*expr*⟩)* [‘`with`’ ⟨*stmtblock*⟩]

⟨*recurrence*⟩ ::= ⟨*lhs*⟩ (‘`,`’ ⟨*lhs*⟩)* ‘`=`’ ‘`rec`’ ⟨*recheader*⟩ [‘`[`’ ⟨*expr*⟩ ‘`]`’] ⟨*stmtblock*⟩ ‘`with`’ ⟨*stmtblock*⟩

⟨*recheader*⟩ ::= (`ID` (‘`,`’ `ID`)* ‘`[`’ ⟨*selector*⟩ ‘`]`’)+

⟨*stmtblock*⟩ ::= ⟨*statement*⟩ | ‘`{`’ ⟨*statement*⟩+ ‘`}`’

⟨*function*⟩ ::= ⟨*funcdecl*⟩ | ⟨*funcdef*⟩

⟨*funcdecl*⟩ ::= ‘`extern`’ ‘`func`’ `ID` ‘`(`’ ⟨*datatype*⟩ (‘`,`’ ⟨*datatype*⟩)* ‘`)`’ ‘`=>`’ ⟨*datatype*⟩

⟨*funcdef*⟩ ::= ‘`func`’ `ID` ‘`(`’ ⟨*finputs*⟩ ‘`)`’ [‘`=>`’ ⟨*datatype*⟩] ‘`=`’ ⟨*funcbody*⟩

⟨*finputs*⟩ ::= [⟨*datatype*⟩] `ID` (‘`,`’ [⟨*datatype*⟩] `ID`)*

⟨*funcbody*⟩ ::= ⟨*expr*⟩ | ‘`{`’ ⟨*statement*⟩* ⟨*expr*⟩ ‘`}`’

Figure 2.2: Grammar for the input specification of *Vaani*.

⟨*lhs*⟩ ::= `ID` [‘`[`’ ⟨*selector*⟩ (‘`,`’ ⟨*selector*⟩)* ‘`]`’]

⟨*expr*⟩ ::= ⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩
  | ⟨*unop*⟩ ⟨*expr*⟩
  | ⟨*expr*⟩ ‘'’ // transpose
  | ⟨*expr*⟩ ‘`?`’ ⟨*expr*⟩ ‘`:`’ ⟨*expr*⟩ // conditional
  | ⟨*id*⟩ ‘`(`’ [⟨*arg*⟩ (‘`,`’ ⟨*arg*⟩)*] ‘`)`’ // function call
  | ⟨*expr*⟩ ‘`[`’ ⟨*selector*⟩ (‘`,`’ ⟨*selector*⟩)* ‘`]`’ // exact indexing
  | ⟨*expr*⟩ ‘`{`’[`INT` ‘`;`’] `INT` (‘`,`’ `INT`)* ‘`}`’ // offset indexing
  | `CONSTANT` (`INT` | `FLOAT` | `BOOL`)
  | `ID`

⟨*arg*⟩ ::= ⟨*expr*⟩ | ⟨*op*⟩

⟨*selector*⟩ ::= ⟨*expr*⟩ | [expr] ‘`:`’ [expr] [‘`:`’ [expr]]

⟨*op*⟩ ::= ‘`+`’ | ‘`*`’ | ‘`&&`’ | ‘`||`’

⟨*binop*⟩ ::= ‘`+`’ | ‘`.+`’ | ‘`-`’ | ‘`.-`’
  | ‘`*`’ [⟨*genop*⟩] | ‘`.*`’ | ‘`/`’ | ‘`./`’ | ‘`%`’ | ‘`.%`’
  | ‘`^`’ [⟨*genop*⟩] | ‘`.^`’
  | ‘`==`’ | ‘`!=`’ | ‘`<=`’ | ‘`<`’ | ‘`>=`’ | ‘`>`’
  | ‘`<<`’ | ‘`>>`’
  | ‘`&&`’ | ‘`||`’

⟨*unop*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`!`’

⟨*genop*⟩ ::= ‘`(`’ ⟨*arg*⟩ ‘`,`’ ⟨*arg*⟩ ‘`)`’

`DIGIT` ::= ‘`0`’..‘`9`’

`LETTER` ::= ‘`a`’..‘`z`’‘`A`’..‘`Z`’

`INT` ::= `DIGIT`+

`DECIMAL` ::= `DIGIT`* ‘`.`’ `DIGIT`+

`EXP`::= (‘`e`’ | ‘`E`’) [‘`+`’ | ‘`-`’] `INT`

`FLOAT` ::= `INT` `EXP` | `DECIMAL` [`EXP`]

`ID` ::= (‘`_`’ | `LETTER`) (‘`_`’ | `LETTER` | `DIGIT`)*

Figure 2.2: Grammar for the input specification of *Vaani* (cont.)

Datatype	Multidimensional array size
scalar()	[]
vector(m)	[m]
rvector(m)	[m]
cvector(m)	[m, 1]
matrix(m)	[m, m]
matrix(m, n)	[m, n]
tensor()	[]
tensor(m,n)	[m, n]
tensor(m, m, m)	[m, m, m]
tensor(m, n, k)	[m, n, k]
tensor(n_1, n_2, ... , n_k)	[n_1, n_2, ... , n_k]

Table 2.1: Type description and corresponding multidimensional array

$*$ indicates that the expression preceding it can repeat 0 or more times, and square brackets '[]' represent optional expressions. The | symbol separates options for the same non-terminal.

`ID` refers to any identifier. *Vaani* identifiers contain upper or lower case letters, underscore, and digits, but cannot start with a digit, similar to the identifiers used by almost all programming languages. The constants supported in *Vaani* are integer, floating point and boolean (`true` or `false`). Integers in *Vaani* are currently a sequence of digits. Floating point numbers accept decimal or exponential notation.

A *Vaani* program starts with the keyword `program` or `function` followed by an identifier to name the program or function. It is followed by a set of declarations of inputs and outputs, discussed in detail in Section 2.3. After the input output specification, a sequence of statements specify the actual computations. There are three types of statements recognized in *Vaani*: an assignment, a recurrence or a function. An assignment statements takes a list of expressions on the left hand side, a list of expressions on the right hand side, and an optional annotation of statements using the '`with`' clause. The statements following the '`with`' clause are first executed, then the right hand side expressions are evaluated, and then assigned to the left hand side expressions. *Vaani* supports assignments to either identifiers or a subset of an identifier (a tensor represented by an identifier) using square bracket indexing (described in Section 2.6.1). Recurrences are described in Section 2.7 and functions are described in Section 2.10.

Vaani's datatype	C datatype
bool	bool (stored as a byte)
int8	int8_t
int16	int16_t
int32	int32_t
int64	int64_t
uint8	uint8_t
uint16	uint16_t
unit32	uint332_t
uint64	uint64_t
real32	float
real64	double
complex64	float complex
complex128	double complex

Table 2.2: Primitive datatypes supported in *Vaani* and their corresponding C datatypes

2.2 TYPE SYSTEM

All data in *Vaani* is represented as a multidimensional array, with a fixed primitive datatype and number of dimensions, but the actual size of each dimension can be determined at runtime. For example, matrix 'A' in Figure 2.1a is a two dimensional array with size $m \times n$, where m and n can be determined at runtime. These multidimensional arrays are represented in *Vaani* as *tensor*s with 0 or more dimensions (0 dimensions implying a scalar element). During declaration, the types can be defined using the keywords `scalar`, `cvector` (column vector), `rvector` (row vector), `vector` (default vector is a row vector), `matrix` and `tensor` (multidimensional). These keywords are provided for ease of declaring the types, but all of them are represented as *tensor*s within the context of *Vaani*. Some example usages and their corresponding array sizes are presented in table 2.1.

The currently supported primitive datatypes are boolean (bool), signed (int8, int16, int32, int64) and unsigned (uint8, uint16, uint32, uint64) integers, real (real32 (float), real64 (double)) and complex (complex64 (float), complex128 (double)) numbers. The numbers in the names indicate the number of bits used by the datatype. *Vaani*'s primitive datatypes and their corresponding C datatypes are shown in table 2.2. The array sizes are represented as a list of symbolic expressions providing the size in each of the dimensions. Default scalar types are int32 (as the most commonly added scalars are indices) and all other types are real64 (double).

The input types are explicitly provided by the user and the output and intermediate data types are inferred by *Vaani* during the compilation process. Type analysis using casting rules and symbolic computations is described in Section 4.2. Distribution of these arrays onto a multidimensional grid is described in Section 4.6.

2.3 INPUT OUTPUT SPECIFICATION

Input and output to the program/function is required to be explicitly specified in *Vaani* using the keywords `in` for inputs, `out` for outputs and `inout` for variables that are both inputs and outputs. The inputs must be type annotated, and the output types can be deduced from the input types using type analysis (Section 4.2). The type declared in the output must be compatible (as defined in Section 2.4) with the type deduced by the type analysis algorithm, else *Vaani* flags an error.

⟨*declaration*⟩ of the grammar in Figure 2.2 describes the input output specification for *Vaani*. each declaration begins with a keyword '`in`', '`out`' or inout; a list of identifiers, and an ⟨*objtype*⟩ to define the type. *Vaani* allows for the use of undeclared identifiers in the size descriptions and implicitly adds them to the list of inputs with a default type of `int32`.

For a program, scalar inputs are read from command line arguments, while arrays are read from files. For a function, inputs and outputs are passed as arguments. `inout` variables are modified in place for functions.

2.4 ELEMENT-BY-ELEMENT OPERATIONS

Vaani supports element-by-element operations on compatible arrays using MATLAB like dot '`.`' representation for certain operations. For example, '.^' represents element-by-element power operation. Most other operations are provided as functions. Table 2.3 lists all the element-by-element operations supported by *Vaani*, currently. Element-by-element operations can also be performed using user defined or declared scalar functions on compatible arrays as described in Section 2.10. Table 2.4 gives the operator precedence in *Vaani*, from high to low.

Vaani supports combining arrays of different sizes to perform element-by-element operations. This is supported by replicating the values in an array to match the other input arrays. To check for compatibility of two arrays, the array sizes are extended by appending 1s if needed, to make the two array sizes of equal dimensions. Two arrays are compatible for binary operation if, at each dimension, either the two sizes are statically determined to be equal using symbolic equivalence checking, or atleast one of the two sizes is statically equal to 1. In this case, the array with size 1 is implicitly replicated along this dimension m times to match the size m of the other array. This replication is performed in as many dimensions

Function	Description
	Basic Arithmetic
+ or .+	Addition
- or .-	Subtraction
.*	Multiplication
/ or ./	Division
.^	Power
min	Minimum
max	Maximum
% or .%	Modulo
remainder	Remainder
ceil	Round up
floor	Round down
	Relational operations
==	Equal to
!=	Not equal to
<	Less Than
<=	Less than or equal to
>	Greater Than
>=	Greater than or equal to
	Boolean Arithmetic
&&	Logical and
||	Logical or
xor	Logical xor
!	Logical not
	Bitwise Operations
bitand	Bitwise and
bitor	Bitwise or
bitxor	Bitwise xor
bitflip	Bitwise not
<<	Left shift
>>	Right shift

Function	Description
	Trigonometric functions
sin	Sine
cos	Cosine
tan	Tangent
asin	Inverse sine
acos	Inverse cosine
atan	Inverse tangent
atan2	Four quadrant inverse tan
	Hyperbolic functions
sinh	Hyperbolic sine
cosh	Hyperbolic cosine
tanh	Hyperbolic tangent
asinh	Inverse hyperbolic sine
acosh	Inverse hyperbolic cosine
atanh	Inverse hyperbolic tangent
	Complex functions
abs	Absolute value
phase	Phase computation
real	Real component
imag	Imaginary component
conjugate	Conjugate
	Exponents and Logarithms
exp	Exponent
log	Logarithm base e
log2	Logarithm base 2
log10	Logarithm base 10
sqrt	Square root
cbrt	Cube root
	Miscellaneous
? :	conditional
<id >	User functions

Table 2.3: Element-by-element operations defined in *Vaani*

Level	Operator	Description
1	( )	function call
2	[ ]	exact index
	{}	offset index
3	’	matrix conjugate transpose
	.’	matrix simple transpose
	^	matrix power
	.^	element-by-element power
4	+	unary plus
	-	unary minus
	!	not
5	*	general matrix multiplication
	.*	element-by-element multiplication
	./ or /	division
	.% or %	mod
6	.+ or +	addition
	.- or -	subtraction
7	<<	left shift
	>>	right shift
8	<	less than
	>	greater than
	<=	less than or equal to
	>=	greater than or equal to
9	==	equal to
	!=	not equal to
10	&&	logical and
11	||	logical or
12	? :	Conditional

Table 2.4: Operator precedence

Input array sizes	Extended array sizes	Combined array size
Compatible array sizes		
[m, n] [n]	[m, n] [1, n]	[m, n]
[m, n, k] [n, 1]	[m, n, k] [1, n, 1]	[m, n, k]
[m, 1] [n]	[m, 1] [1, n]	[m, n]
[m, k] [n, 1, k]	[1, m, k] [n, 1, k]	[n, m, k]
[m, n] [k, 1, 1]	[1, m, n] [k, 1, 1]	[k, m, n]
Incompatible array sizes		
[m, n] [m]	[m, n] [1, m]	X
[m, n, k] [m, k]	[m, n, k] [1, m, k]	X

Table 2.5: Example compatible and incompatible arrays

as necessary for each of the two arrays to match the two array sizes. This replication has been conventionally termed as a broadcast in NumPy and MATLAB.

The rules stated above are identical to the rules followed by NumPy. This differs from the broadcasting rules of MATLAB, where the 1s are appended to the end (instead of at the beginning) to extend the arrays to equal dimensions, mainly due to the default internal representation of the arrays. MATLAB, being column-major, has the first dimension the quickest changing, where as NumPy, our target language C and hence *Vaani*, being row-major, has the last dimension changing fast.

Table 2.5 provides a few examples of compatible and incompatible array combinations, and the final combined size of the operation.

These rules are extended by applying them pairwise for operations with more operands, like addition and element-by-element multiplication, which support multiple operands.

2.5 MATRIX OPERATIONS

Matrix operations (multiplication ‘`*`’, power ‘^’, and transpose ‘`'`’ or ‘`.'`’) are defined for matrices of upto two dimensions. General matrix multiplication and matrix power using user specified additive and multiplicative operations are supported using the operator followed by a pair of operations or user defined scalar binary functions. This is described in the grammar

(Figure 2.2) using $\langle genop \rangle$. For example, `A*(min,+)B` performs a matrix multiplication with minimum as the additive operation and addition as the multiplicative operation, which could be used to perform shortest path computations by using adjacency matrices as the operands [3]. Also, `A^(min,+)n` performs a similar matrix product n times with A. These operations are used in the example SSSP of Figure 2.1b.

2.6 INDEXING

Vaani uses two types of indexing into the arrays, exact and offset indexing.

2.6.1 Exact Indexing

Square brackets ('[' ']') are used to denote exact indexing into the array, to access individual elements or sub-arrays. *Vaani* supports a triplet notation with [*start*] : [[*stop*][: *step*]] to refer to sub-arrays (here, the square brackets in definition imply optional parameters). The notation implies a list of numbers starting from *start*, ending at *stop* but not including *stop*, and in increments of *step*. The default *step* size is 1, default *start* is 0, and default *stop* is context-dependent, and is the end of an array in the context of indexing. Statically determined constant indices can be negative to refer to the end of the array, but dynamic indices must be positive. For example, consider a matrix `A` of size $m \times n$. `A[0, 0]` refers to the first element and `A[-1,-1]` refers to the last element `A[m-1, n-1]`. `A[:,0]` refers to the first column of `A`, `A[:, -1]` to the last column (`A[:, n-1]`), and `A[i1:j1, i2:j2]` to a sub-matrix of `A` where all the values `i1`, `i2`, `j1`, `j2` are assumed to be positive integers within the range. `A[0, ::2]` would imply alternate elements in the first row of `A`. This is similar to the notation in NumPy, however, the main difference is that only statically constant negative integers can be negative, while NumPy, being an interpreted language, allows any index to be negative.

2.6.2 Offset Indexing

Curly brackets ('{' '}') are used to denote offset indexing, which specifies the offset of the index from an abstract current index, for each element in the array. We use these offsets to represent spatial offsets for stencil computations and temporal offsets for recurrences. For example, `A{0,1}` represents the element `A[i, j+1]` for an abstract current index of `[i,j]`. Offsets are couple with boundary specifications to determine the range of the operations. Currently supported boundary conditions for *Vaani* are `none` and `periodic`. `none` denotes that the boundary values are not computed, while `periodic` indicates the offsets are wrapped around circularly. The same offset index of `A{0,1}` for an $m \times n$ matrix `A` would have indices of range `[0:m, 0:n-1]` if boundary is `none` while the full index range `[0:m, 0:n]` if boundary is `periodic`. In the case of periodic boundary condition, `A{0,1}` for an actual index of `[i, n-1]` would imply `A[i, 0]` by wrapping around to the beginning of the array.

If both spatial and temporal offsets are required, then the offsets are separated by a '`;`' to differentiate the first temporal offset, and the latter spatial offsets. An example of this indexing is used in Jacobi2D (Figure 2.1e) and Gauss2D (Figure 2.1f).

2.7 RECURRENCES

Vaani does not support traditional loops, but instead, supports recurrences. <*recurrence*> of the grammar in Figure 2.2 describes the syntax of recurrences. Components of a recurrence are described below.

2.7.1 Output Variables

Output variables are a list of identifiers that are defined within the recurrence whose values will be carried over outside the recurrence. The `B` after the keyword `rec` in Figures 2.1e and 2.1f are the output variables.

2.7.2 Selectors

Selectors specify both the number of iterations to run, and the values to be saved. An expression selector specifies the number of iterations to be run, and saves the last computed value of the variable. A selector with a ':' operator specifies both the number of iterations to be run, and that values of all iterations are to be saved. This increases the dimension of the variable by 1 from the internal recurrence size. The `[iter]` after the output variable `B` in Figures 2.1e and 2.1f are the selectors. Here, starting with the initial values of `B`, the iterations of the recurrence are run to obtain `B[iter]`. For example, if the selector had been `B[0:iter]`, then the values of `B` would be stored for each iteration of `B`, and the resulting `B` would have a size of $iter \times m \times n$, increasing the dimensions of `B`. It should be noted here that in order to obtain `B[iter]`, the computation must run for *iter* iterations starting from 1, while to compute `B[0:iter]`, the computation runs for $(iter - 1)$ iterations, as the expression `0:iter` excludes *iter*.

Alternatively, a ‘*’ can be used as a selector, specifying that the recurrence runs to convergence. Since *Vaani* uses fixed array sizes, ‘*’ cannot be combined with the ’:’ specification and only the last (converged) value can be saved.

Selectors are combined with the initialization values to obtain the recurrence iterations, and the process is described in Section 2.7.6.

2.7.3 Condition

A condition can also be specified such that the iteration is run as long as this condition is true. When combined with ‘*’ selector, the recurrence is run till this condition is false. Combined with an iteration count, it terminates when either the condition is false or iteration count is reached. Conditional execution is not allowed with ‘:’ selector for the same reason as ‘*’.

2.7.4 Body

The body of a recurrence is a list of statements, defining all the output variables in each iteration using variables from previous iterations. The values from previous iterations can be accessed using temporal offsets in the recurrence domain. For example, `A{-1}` denotes the value of the previous iteration, and `A{-2}` denotes the value 2 iterations before. For example, fibonacci numbers can be represented using the statement `F = F{-1} + F{-2}`. Temporal offsets can be combined with spatial offsets using the ‘;’ operator. Jacobi2D and Gauss 2D from Figures 2.1e and 2.1f demonstrate the usage of both temporal and spatial offsets to perform iterative stencil computations. The temporal offsets here can only be non-positive integers, while spatial offsets can be any integer constant. The use of 0 as a temporal offset for some of the terms of Gauss2D suggests that the values computed in the current iteration must be used, which implies that a Gauss-Siedel iteration must be performed. The values must be such that a consistent direction of computation can be determined, else *Vaani* throws an error.

2.7.5 Initial Values

The statement block after the ‘with’ keyword provides the initial values to the recurrence. The initial values can be specified using two ways.

Single Iteration Point

The value of a variable at a single constant integer iteration point is specified using square brackets (`B[0]` in Figures 2.1e and 2.1f). The number of values specified must equal the maximum temporal offset of that variable that is used in the body of the recurrence. For example, if a variable `A` has temporal offsets of {-1} and {-3}, then three consecutive integer points must be specified.

All Iteration Points

The value of a variable for all iteration points can be specified using a ‘:’ operator in square brackets, giving an array of values, where each element is mapped to one iteration (for example, `a[:]  = A`, where A is a 2 dimensional array, would give vector values to `a`). The number of iterations must match the leading dimension of the array. This specification is only valid for fixed iteration space.

2.7.6 Determination of the Iteration Space

Iteration space is determined by using the indices of the initial values and the selector. The initial values determine the lower bound of the iteration, and the selector specifies the upper bound. For example, in Figures 2.1e and 2.1f, the initial value of 0 specifies the iteration starts from 1, and the selector specifies that the final iteration is `iter`.

2.8 REDUCE/SCAN

Reduction and scan operations are supported in *Vaani* using functions. The function signatures are `reduce(Array, [operation, axes])` and `scan(Array, [operation, axes])`. The operation can be a built-in function (+, *, min, max, &&, ||, xor, bitand, bitor, bitxor) or a user-defined function, and the default is addition (+). The axes for reduce can be one or more integers specifying the dimension to reduce along, default is 0. Scan can only be performed in one dimension.

Reduce removes the dimensions along which it is reduced, reducing the dimension of the result array. It is planned to take a user specified boolean to retain the dimensions or not, but is currently not implemented. For example, `v = reduce(A, +, 1)` on a matrix `A` of size $m \times n$ results in a vector `v` of size m where each element `v[i]` is the sum of all the elements in the row `i` of matrix `A`.

Scan result has the same dimensions as the input array, with the elements in the specified dimension a cumulative of the values. Scan currently performs inclusive scan, and a future boolean to select inclusive or exclusive scan is planned.

2.9 REARRANGE

Rearrange functions currently supported in *Vaani* are described below.

2.9.1 Reshape

`reshape(array, sz1, sz2, ...)` function takes an array `A` and recasts it as an array of size `(sz1, sz2, ..)`. If the original size of the matrix is $m_1 \times m_2 \times ... \times m_k$, this operation flattens the original array to one dimension of size $m_1 * m_2 * ... * m_k$, and then recasts it to its new sizes $sz_1 \times sz_2 \times ... \times sz_j$. The total size of the array A and the size obtained by the new sizes must statically be equal on symbolic equivalence check, i.e., $m_1 * m_2 * ... * m_k = sz1 * sz2 * ... * sz_j$. For example, an array of size `(m, n, k)` can be cast as `(m*n, k)`, `(n, 1, k, m)` or `(m, 1, n, k)`, but not as `(m, m, k)` or `(p, q)` even if `m*n*k = p*q` at runtime, as *Vaani* cannot determine it statically. Reshape operations are of two types.

Implicit Reshape

This is possible if the original and final array shapes and distributions over the process grid remains the same before and after reshape. This is possible if the reshape adds or removes 1 length dimensions to the array, or combines adjacent sizes where at least one of them is not partitioned over the grid. This reshape does not generate any communication, and is only performed implicitly in the way underlying data is viewed.

Explicit Reshape

The reshape must be explicit if the final array shape over the process grid differs from the initial array. This reshape involves communication across the process grid.

2.9.2 Reorder

`reorder(array, order)` changes the dimensions of an array to be viewed in the order specified. Matrix transpose is a special case of reorder, where reorder(A, 1, 0) is a simple

	expression	original size	result size
Explicit	replicate(A, m, n)	[m, n]	[m*m, n*n]
	replicate(A, n)	[m, n]	[m, n*n]
	replicate(A, k, m, 1)	[m, n]	[k, m*m, n]
Implicit	replicate(A, k, 1, 1)	[m, n]	[k, m, n]
	replicate(A, n)	[m, 1]	[m, n]
	replicate(A, m, 1)	[n]	[m, n]

Table 2.6: Example replicate expressions

matrix transpose. Reorder retains the row or column distribution of the original matrix, but arranges them differently, often leading to a need to explicitly recreate the array. This often involves communication, unless the reorder only changes the order of dimensions not partitioned across the grid.

2.9.3 Flip

`flip(array, [axis])` reverses the elements in axis dimension, defaulting to 0 if axis is not specified. Flip involves communication to place the data in the correct place.

2.9.4 Replicate

`replicate(array, rep1, rep2, ...)` replicates the array by the sizes $\texttt{rep}_i$ specified. This takes the sizes list of the array and the sizes represented by rep, aligns them at the end and appends 1s to make them the same size, and multiplies the terms pairwise to create a new array. Replicate again is of two types, similar to reshape. Table 2.6 shows some examples of replicate, and their final shapes.

Implicit Replicate

Implicit replicate replicates the array along a dimension that is originally 1. This only has communication to make the array values available on the processes requiring it.

Explicit Replicate

Explicit replicate is when an array is replicated along a dimension that is not originally 1. This increases the length of a dimension, and hence requires a redistribution of the array.

2.10 USER FUNCTIONS

Vaani supports two types of user functions, user declared and user defined functions. These functions can be used as element-by-element operations, as operators for general matrix multiplication and power, and as functions for reduction and scan. Currently, *Vaani* only supports scalar functions.

2.10.1 User Declared Functions

Externally defined scalar C functions can be declared in *Vaani* using the `extern` keyword. As an example,

```
extern func modadd(int32, int32) => int32
```

introduces a function `modadd` that takes two integers and returns an integer. *Vaani* declares these functions in the final generated C code as extern functions, and must be coupled with their implementations at compile time.

2.10.2 User Defined Functions

Users can also define functions in *Vaani* using statements and expressions supported in *Vaani*. As an example,

```
func modadd(int32 a, int32 b) => int32 = { (a + b)%n }
```

introduces a function modadd that performs modulo addition. Here, the value `n`, that is not an input to the function, takes the value of `n` when the function is defined. User defined functions do not need type annotations, the types can be deduced on instantiation. For example, a function

```
func square(a) = { a*a }
```

introduces a function square that squares a given number where the return type is determined by the type of the argument at each instance.

CHAPTER 3: INTERMEDIATE REPRESENTATIONS

The intermediate representations(IR) of *Vaani* represent the specified computations at varying levels of detail. The representations are designed to enable different types of optimizations at different stages of the compilation. The high-level IR (HLIR) represents the operations on the arrays as specified by the user, the mid-level IR (MLIR) represents computations categorized by the data access patterns, the low-level IR (LLIR) represents the communication and local computation, and the C-level IR (CLIR) is the final step before generating the code, and is in the form of iterations spaces and instruction blocks. The input program is parsed and transformed through these representations as shown in Figure 3.1 before generating the final code. In this chapter, we explore the rationale for this design in Section 3.1 and look at the different IRs in sections 3.2 to 3.5. We provide details of the actual translation in chapter 4.

3.1 RATIONALE FOR MULTIPLE INTERMEDIATE REPRESENTATIONS

The design and implementation of a distributed memory program intuitively follows a general pattern. First, a high level computation specification is developed, which is independent of underlying implementation details. Then, the data and computation is distributed onto a process grid. Next, the communication necessary to carry out the computation assuming this distribution is determined. Finally, optimizations for local computations are selected and implemented. This represents a flow in which each step is dependent on the previous step, as a specification is necessary to perform distribution, a distribution is necessary to determine communication, and local computations must be extracted before performing local optimizations. These steps are typically performed in this order, even if a user wishes to backtrack or maintain multiple versions of the final code. *Vaani*'s intermediate representations follow this intuitive pattern, and thus enable the user to intuitively define and refine the computations from a high level representation, breaking them down into simpler and smaller chunks of computation, that lends itself to a varying set of operations and optimizations. In *Vaani*, the language described in chapter 2 and HLIR provide the implementation independent specification of computations. MLIR is used to partition and distribute data and computation onto a virtual process grid. Communication is introduced on translation from MLIR to LLIR, and local optimizations are performed on LLIR and CLIR.

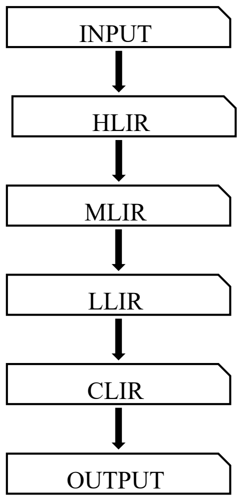


Figure 3.1: Translation through IRs

3.2 HIGH LEVEL INTERMEDIATE REPRESENTATION (HLIR)

The high level IR is used to denote the program using arrays as objects with the operations supported in the input language as-is as nodes. This representation has a list of inputs, a list of outputs, and a directed acyclic graph (DAG) connecting the inputs to the outputs, where each node is an operation. These nodes represent all possible computations defined in *Vaani* like element-by-element operations, matrix operations, indexing operations, etc.

Recurrences are treated as a single node in HLIR, thus avoiding cycles but creating a hierarchical nesting, as shown in Figures 3.6 and 3.7 for Jacobi2D and Gauss2D, respectively. Recurrences have a set of recurrence variables and a set of recurrence outputs. Recurrence variables are the variables that are considered inputs to each iteration of the recurrence and have an initialization edge from outside the recurrence. Recurrence outputs are the outputs defined in each iteration of the recurrence, and they have an edge out of the recurrence. There is a loop in the recurrence, where the outputs of the current iteration are the inputs of the next iteration. This is ignored, and the body of the recurrence is treated as a DAG for the translation. Eventually, buffer allocation and code generation make sure that the output to input match is correctly performed in the final code. This input to output match can be performed unambiguously, as the temporal offsets of recurrences are constant integers. Nested recurrences create a hierarchy of recurrence nodes, where an inner recurrence is considered a statement in the outer recurrence.

HLIR is generated by parsing the input specification into an abstract syntax tree (AST) and performing type and shape analysis to annotate the intermediate nodes with a datatype and symbolic array size. Symbolic expression simplification, strength reduction and common

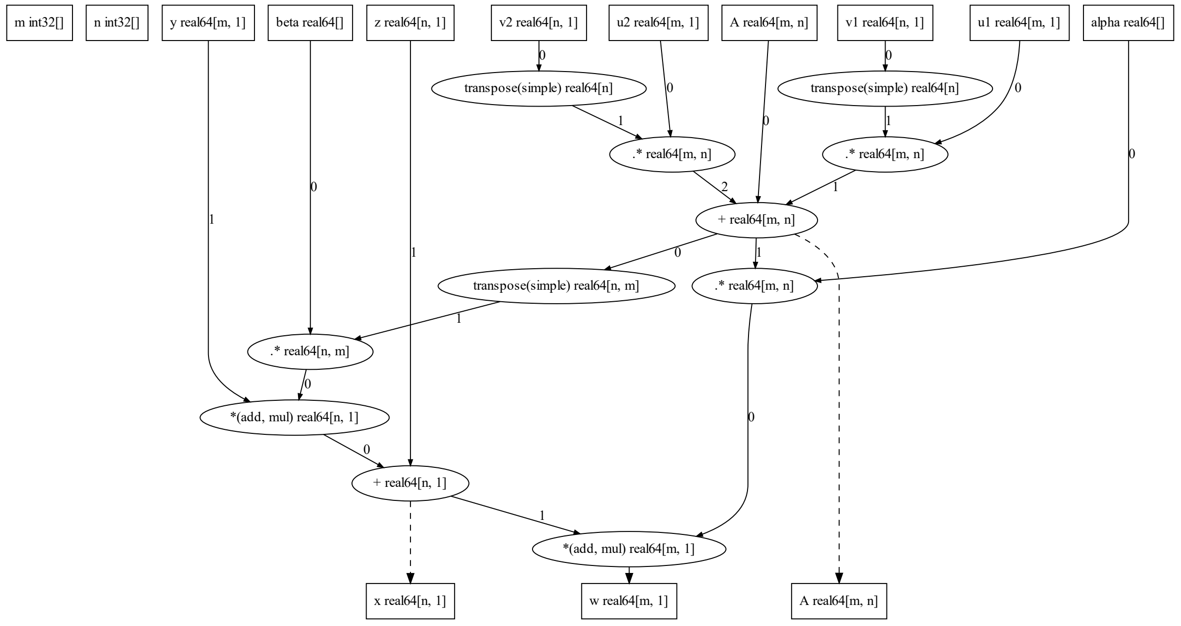


Figure 3.2: GEMVER HLIR

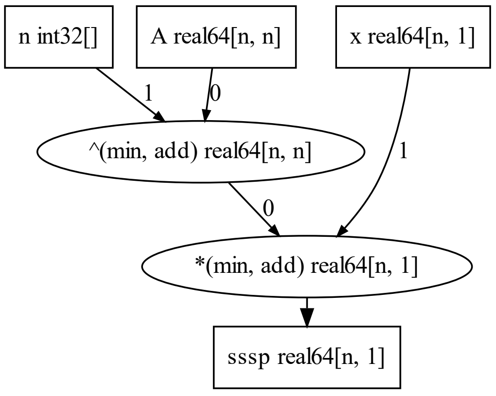


Figure 3.3: SSSP HLIR

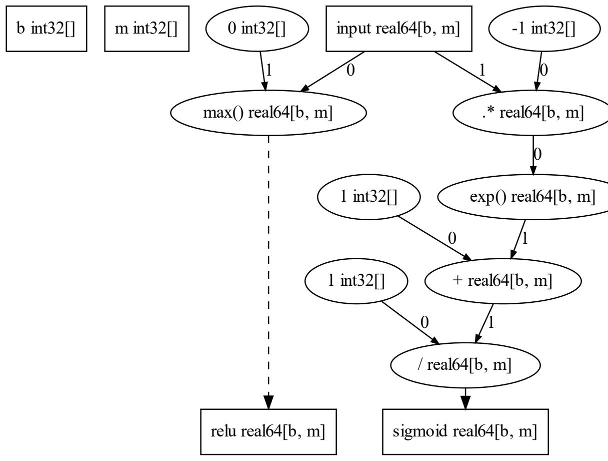


Figure 3.4: Activation HLIR

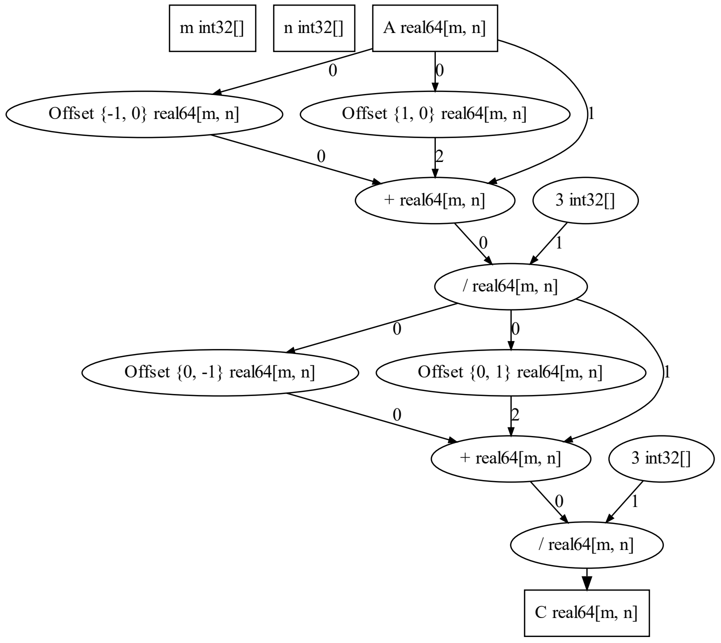


Figure 3.5: Blur HLIR

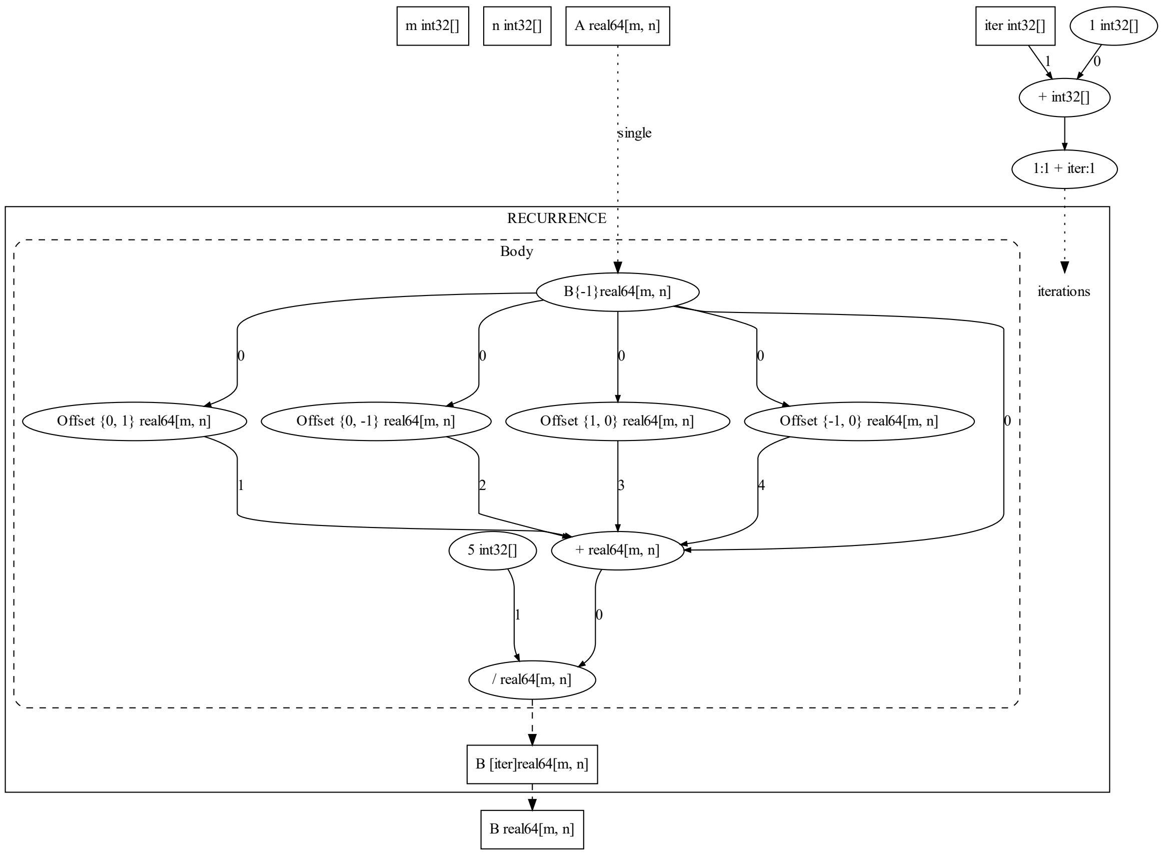


Figure 3.6: Jacobi2D HLIR

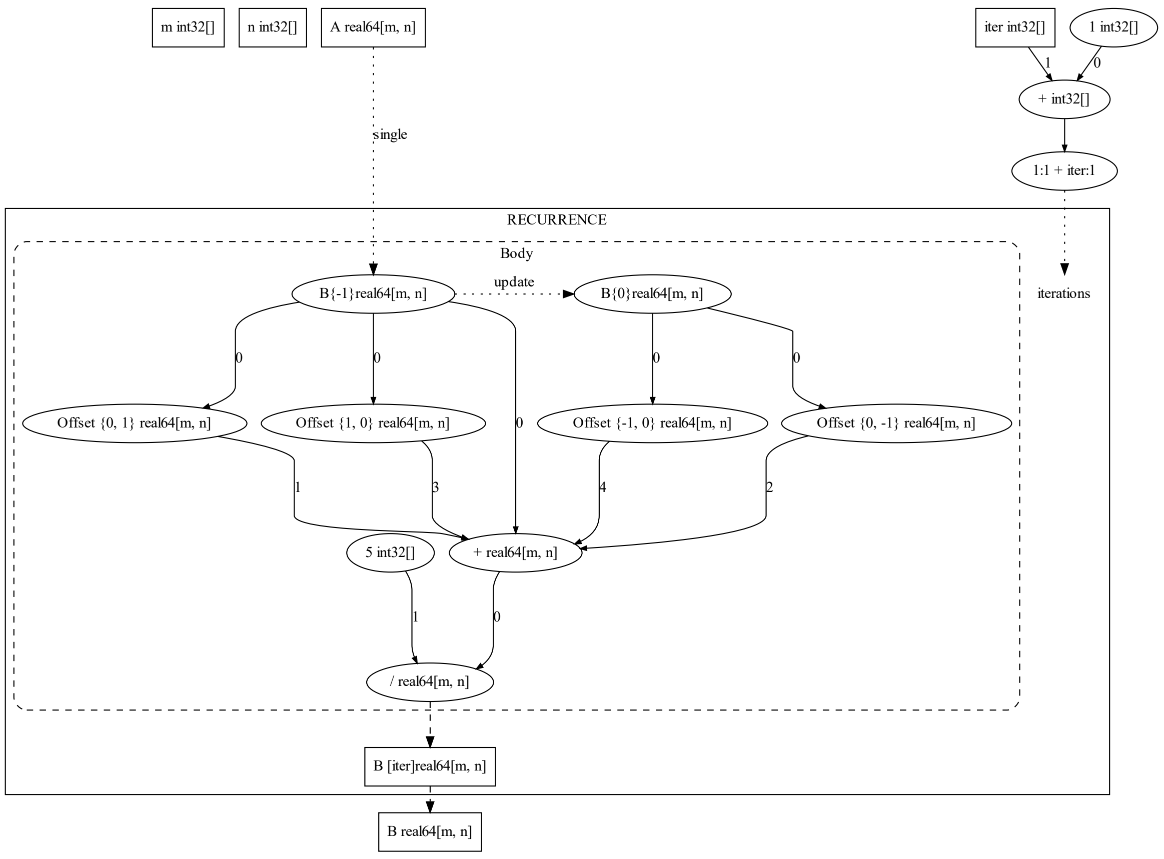


Figure 3.7: Gauss2D HLIR

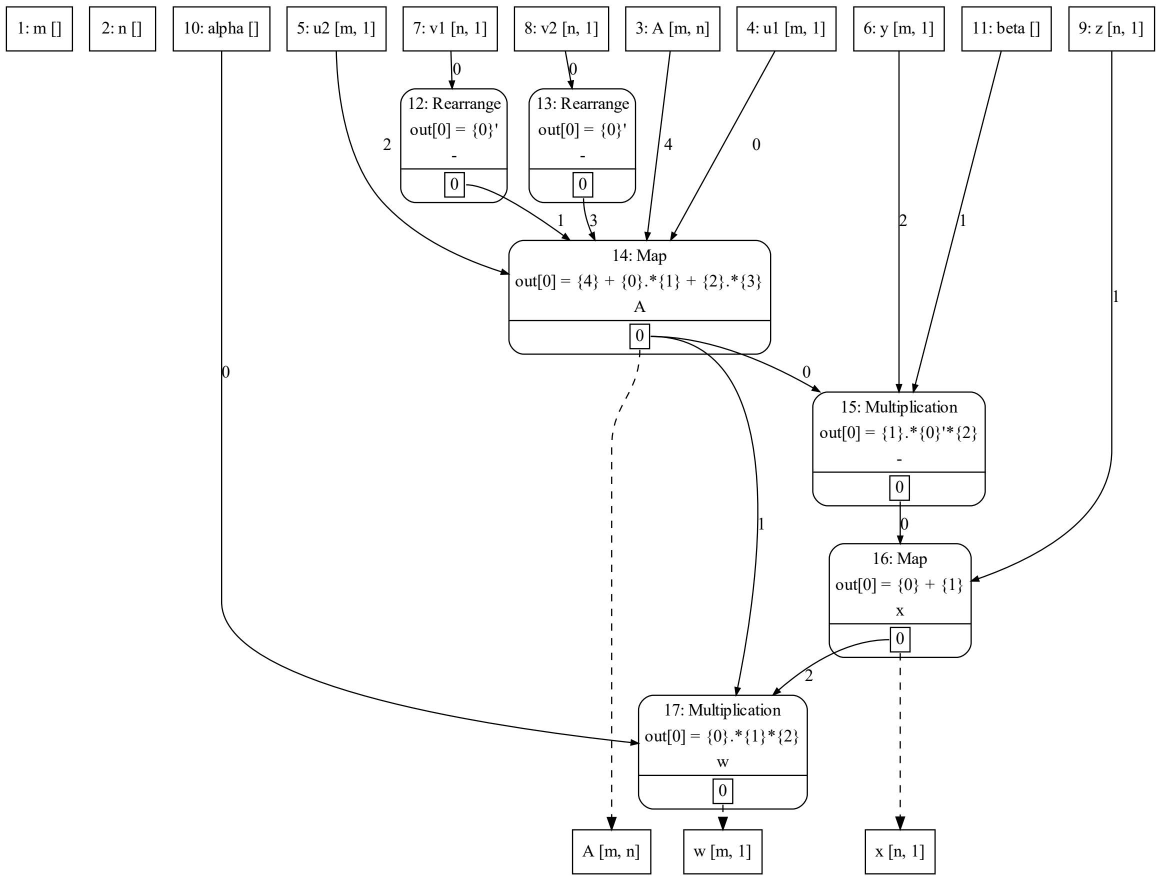


Figure 3.8: GEMVER MLIR

sub-expression elimination are possible optimizations at this level.

Figures 3.2 through 3.7 present the HLIR versions of the sample examples from Figure 2.1. In each of these examples, the topmost rectangular nodes represent the inputs, the bottom rectangular nodes represent the outputs, and the intermediate ovals represent computation nodes. A solid arrow brings all the required inputs to a computation node. These arrows are numbered to specify the order of the inputs to a node.

Inout parameters and assignment operations are technically expected to introduce cycles. However, *Vaani* treats them to be unique nodes in HLIR and thus do not produce cycles. This information is used in a much later stage (buffer allocation), to possibly reduce the total memory consumption by reusing buffers.

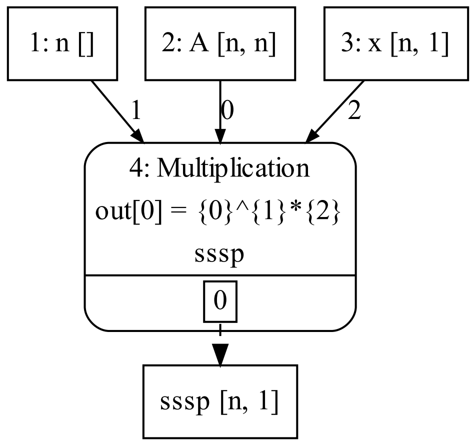


Figure 3.9: SSSP MLIR

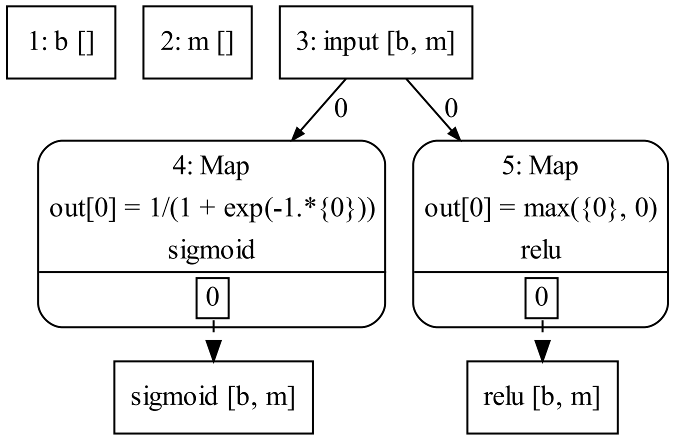


Figure 3.10: Activation MLIR

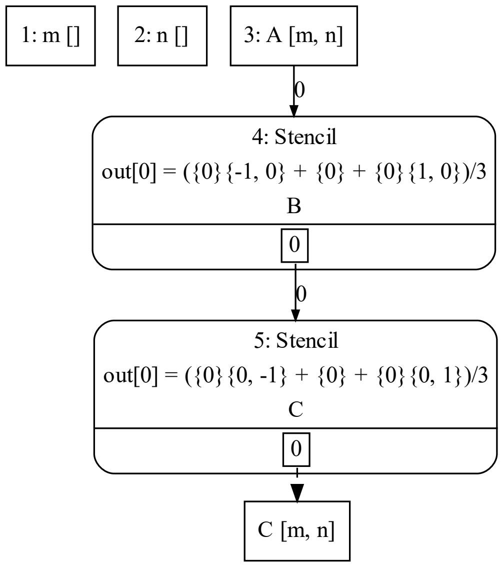

1: m []
2: n []
3: A [m, n]
0
4: Stencil
out[0] = ({0}{-1, 0} + {0} + {0}{1, 0})/3
B
0
0
5: Stencil
out[0] = ({0}{0, -1} + {0} + {0}{0, 1})/3
C
0
C [m, n]

Figure 3.11: Blur MLIR

3.3 MID LEVEL INTERMEDIATE REPRESENTATION (MLIR)

The mid-level IR represents computations grouped by the data access patterns. The main nodes are `map` nodes that represent element-by-element operations, `stencil` nodes that represent stencil computations, `multiplication` nodes that represent matrix-matrix and matrix-vector products, and matrix power operations, `rearrange` nodes that represent transpose, flip, replicate, reorder and reshape, `combine` nodes which have reduce and scan operations, and `recurrence`. Each node has a list of inputs, a list of outputs, and a function operating on the inputs to generate the outputs, which are again, represented as DAGS (a linear text version is presented in the figures).

`map` node takes a set of array inputs and performs element-by-element operations to generate a set of outputs. The actual operations are encoded as a function that map the inputs to the outputs through a set of operations, again stored as a DAG. These nodes need communication to align the arrays across the processes, and no communication for the actual computation. These nodes are obtained by translating element-by-element operations in HLIR and fusing the operations together.

`stencil` node performs the same operations as a `map` node but also includes at least one offset-indexed array. This is different from the `map` node as offset-indexed arrays require ghost regions and data exchange with neighbor nodes.

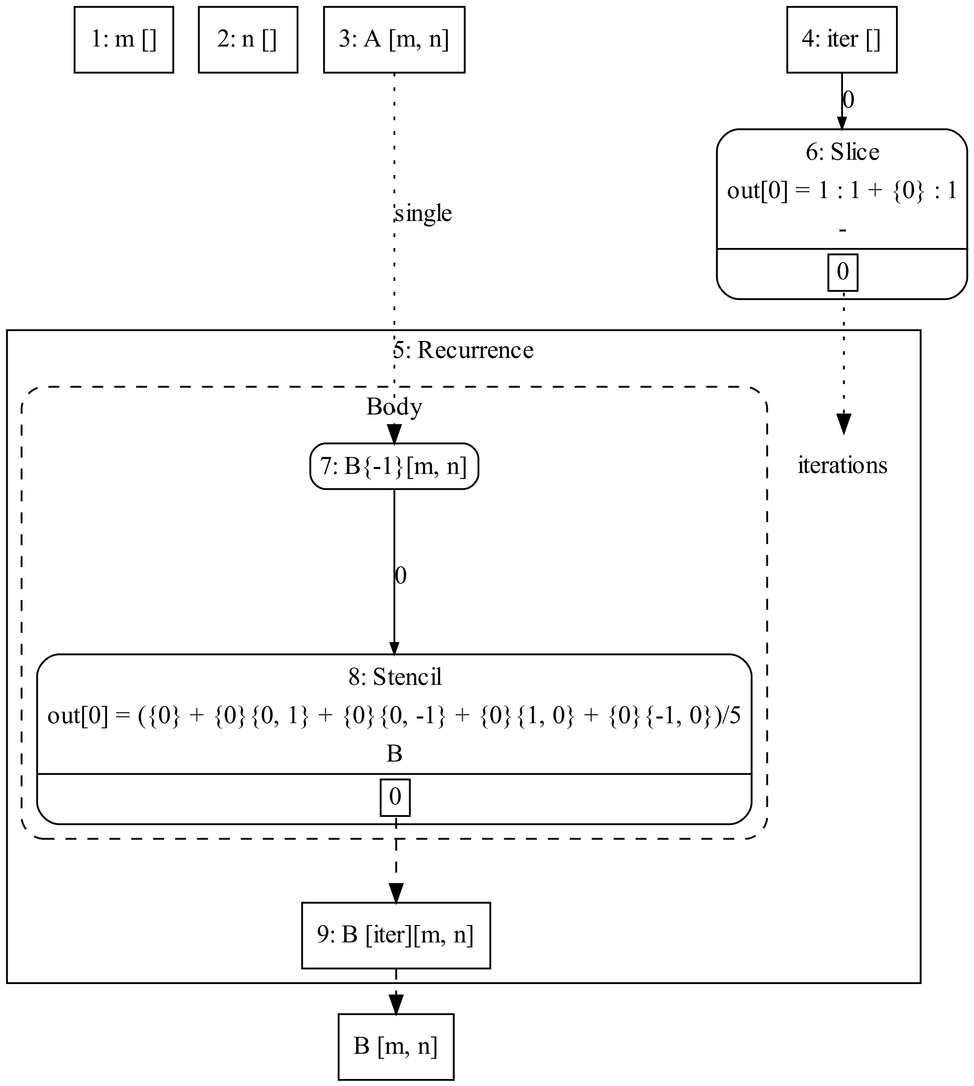


Figure 3.12: Jacobi2D MLIR

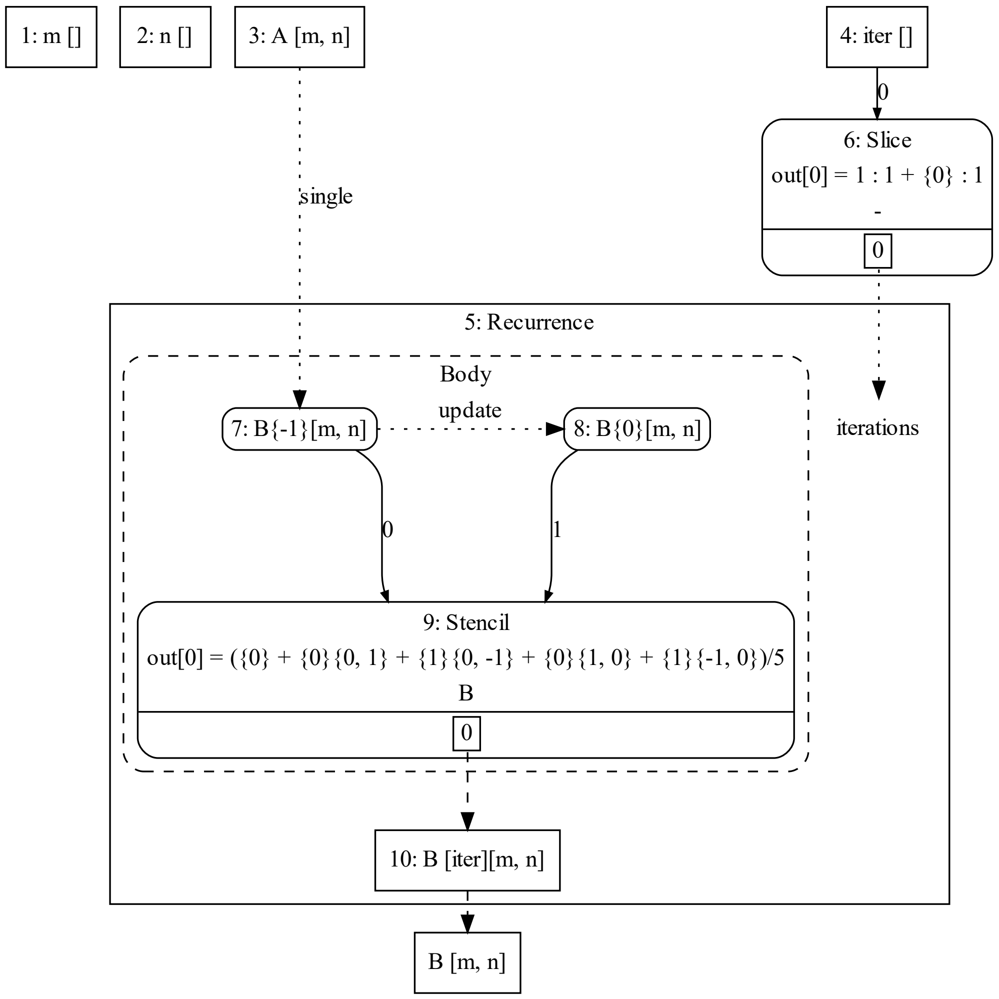


Figure 3.13: Gauss2D MLIR

`multiplication` node performs matrix-matrix and matrix-vector multiplications, and matrix power operations. This node generates different communication patterns based on the algorithm selected.

`rearrange` nodes perform data alignment and rearrangement. These include transpose, flip, replicate, reorder and reshape operations. These nodes generate communication nodes depending on the pattern of rearrangement. These nodes also represent a change in data distribution on a virtual process grid.

`combine` nodes perform reduction and scan operations. These again generate reduce or scan communication patterns.

`recurrence` nodes are hierarchical, like in HLIR and have the same mechanics as in HLIR. Only difference is that the nodes in the DAG of the iteration are MLIR nodes instead of HLIR nodes.

Since MLIR categorizes nodes based on the data access and communication patterns, it lends itself amenable to grouping computations together by merging nodes, defining the communication patterns in lower layers. Grid declaration, data partitioning and mapping (Section 4.6) are performed on the MLIR, and communication and computation strategies are selected for each node in MLIR.

MLIR undergoes a series of transformations during the translation process. The initial MLIR is generated with a one-to-one correspondence to HLIR. Then nodes are merged to group the computations. MLIR also provides an interface to allow users to manually select nodes to merge. MLIR supports each node to have multiple outputs to allow for a greater freedom to the user in merging the nodes. Then, data partitioning and mapping is specified on the merged MLIR. Consistency and propagation of mapping type is performed on MLIR and each node (input, output and intermediate computation nodes) is annotated with a data partition and map.

Figures 3.8 through 3.13 present the MLIR version of the examples from Figure 2.1.

3.4 LOW LEVEL INTERMEDIATE REPRESENTATION (LLIR)

Low-level IR (LLIR) nodes represent computation and communication operations on each process on the virtual process grid. Local data arrays are computed using the partitioning and mapping on the MLIR, to determine the local inputs and outputs. Computation nodes in LLIR specify local computations on local inputs to generate local outputs based on the MLIR. Communication nodes define communication patterns. All LLIR nodes have a mask parameter to select or filter the nodes on which the computation/communication is performed.

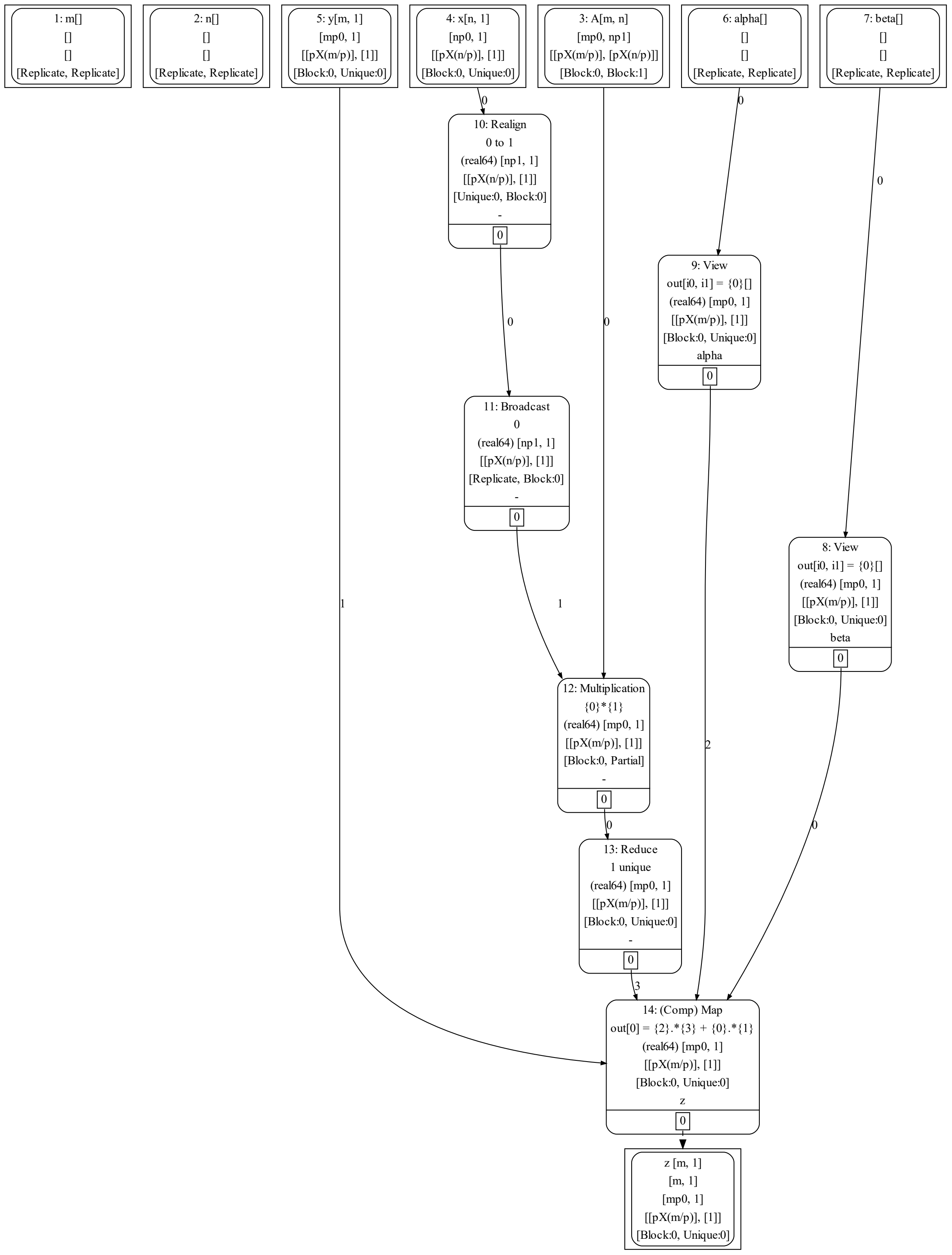


Figure 3.14: LLIR for matrix-vector multiplication

3.4.1 Computation Nodes

Computation nodes have a set of inputs, a set of outputs and a function that maps outputs to inputs using a DAG. The main difference between the computation nodes of MLIR and LLIR is that LLIR nodes represent local computation. The main types of LLIR computation nodes are Map nodes, Stencil nodes, Multiplication nodes and Combine nodes. These perform the operations as their corresponding MLIR nodes, but on local arrays only. The communication requirements and data placements are taken care of in other nodes.

3.4.2 Communication Nodes

Communication nodes in LLIR describe communication patterns in the process grid. `broadcast` node specifies broadcast of an array in a set of grid dimensions. `gather` node gathers an array from all the processes in a particular grid dimension. `scatter` node distributes an array to all the processes in a particular grid dimension. `reduce` and `scan` nodes perform reduction and scan operations on an array in a particular grid dimension. `transpose` node is used to circularly shift data among the indices. `realign` node is similar to a `transpose` node, but the data transfer is one directional. Transpose of a matrix on a two dimensional grid generates a `transpose` node as all the nodes in the grid both send and receive data, while the transpose of a row or column vector on a two dimensional grid generates a `realign` node if the data is present without replication. `boundaryexchange` node is used for halo exchange. `gather` and `reduce` nodes have an option to obtain the result on a unique process or on all the processes in the dimension.

3.4.3 View Nodes

LLIR uses `view` nodes to implicitly view an array as an array of a different shape. They take an input array, an output array and a mapping of output indices to input indices. These are used to implicitly replicate along a dimension, transpose, flip, etc. These index maps are of three types.

1. Identity: These map an index of the output to an index of the input.
2. Reverse: These map an index of the output to the reverse of an index of the input.
3. Unique: These map an index of the output to a constant value.

These nodes are used to view, for example, a vector v of size $[m, 1]$ as a matrix A of size $[m, n]$, where the first index of the output matrix is identity mapped to the first index of the vector, and the second index of the output matrix is Unique mapped to 0. Thus, $A[i, j] = v[i, 0]$ would be result of the index map.

3.4.4 Examples

Consider a simple matrix-vector multiplication $z = \alpha Ax + \beta y$. Figure 3.14 shows the LLIR version of this multiplication. Here, we assume that the computation is performed on a two dimensional square grid of processes. The matrix A is block distributed on the process grid. The second line in node 3, $[mp0, np1]$, show that the size of matrix A is $mp0 \times np1$, where $mp0$ and $np1$ are variables whose values are obtained at runtime. These represent the sizes specified in the next line, $[[p \times m/p], [p \times n/p]]$, indicating that both dimensions are blocked into p parts, of size roughly m/p and n/p, respectively. The last line in node 3, $[block : 0, block : 1]$, indicates that the array dimension 0 is block distributed on grid dimension 0, and array dimension 1 is block distributed on grid dimension 1. The partition information $[[p \times m/p], [p \times n/p]]$, combined with the distribution information $[block : 0, block : 1]$, and a symbolic index for each process in the process grid, together give the exact values of the variables $mp0$ and $np1$. Here, the naming convention is to show that the $mp0$ is obtained by splitting m onto p parts onto dimension 0 of the grid, while the $np1$ is obtained by breaking the size n into p parts on dimension 1 of the grid. The user provides the vectors x and y as column vectors available on the first column of the grid, and the user requires the result vector z, again, to be aligned on the first column of the grid. This is, again, depicted in the last lines of nodes 4, 5, and the final output node z, $[block : 0, unique : 0]$, where array dimension 0 is distributed on grid dimension 0, and the data is presently unique on grid dimension 1, with the index of the unique process being 0. This implies that the vector is distributed on the first column, block distributed across the rows. The constants `alpha` and `beta` are replicated on all the processes, depicted by the $[replicate, replicate]$ on the last line of nodes 6 and 7. Node 12 performs the actual multiplication of Ax on every process. Node 10 is a realign, which transposes x to align with the matrix A. This makes x align along a row, instead of a column. Node 11 then broadcasts the array x in grid dimension 0, using the column communicator. Node 13 reduces the partially distributed multiplication result Ax in grid dimension 1 along the row communicator, obtaining the result as a column vector on the first column of the grid. Node 14 performs the computation $z = \alpha(Ax) + \beta y$. Nodes 8 and 9 are view nodes that map index $[i0, i1]$ to no index on the input side, as α and β are scalars that don't have any indices.

Figures 3.15 through 3.20 present the LLIR versions of the examples from Figure 2.1.

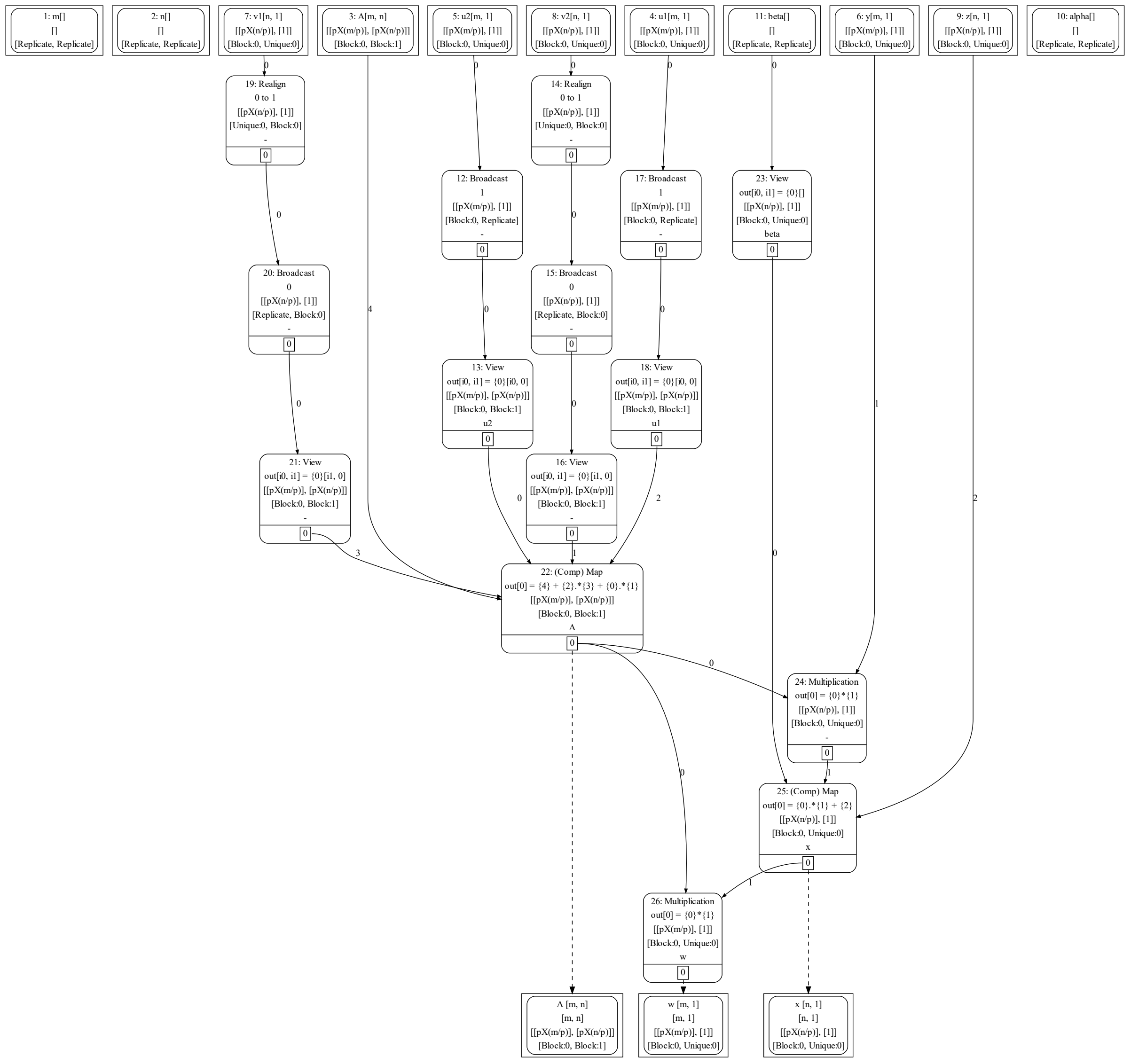


Figure 3.15: GEMVER LLIR

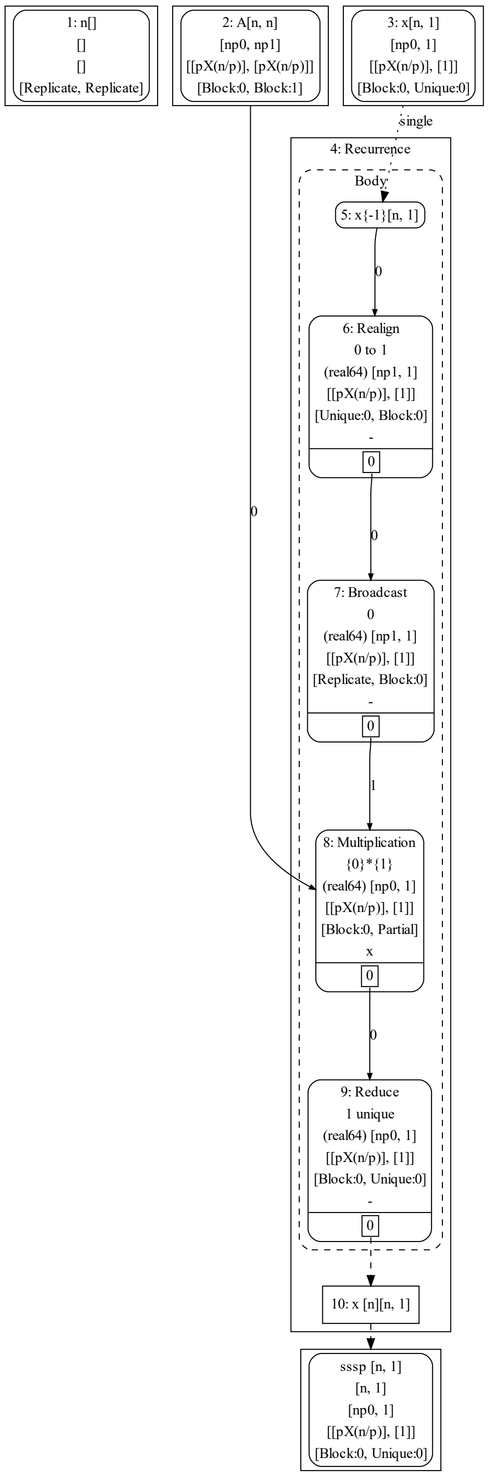


Figure 3.16: SSSP LLIR

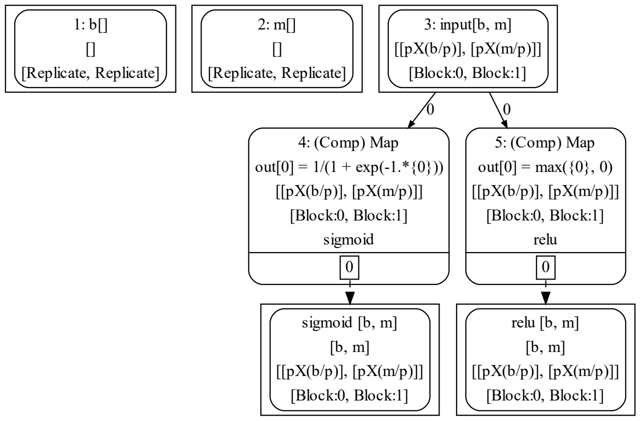


Figure 3.17: Activation LLIR

3.5 C LEVEL INTERMEDIATE REPRESENTATION (CLIR)

C-level IR (CLIR) is a DAG of instruction blocks. Each node has a mask filter, an iteration space, and a list of instructions. Setup and tear-down of the process grid, declaration of variables, allocation of memory, etc. are also represented as nodes in CLIR. This is the final representation before *Vaani* generates the final code. Currently, *Vaani* uses CLIR as a way to represent the final code internally, and does not provide its access to the user.

3.5.1 Computation node

CLIR has a single unified computation node, unlike specialized ones in previous IRs. Each node has a list of indices, an iteration space defined by a list of triplets of start, stop and step.The node also has a list of instructions that appear for each iteration point.

3.5.2 Communication nodes

These nodes represent the same communication patterns as LLIR, but each node has buffer details and, again, a list of instructions.

3.5.3 Instructions

Instructions are C-like instructions that are used in CLIR. Currently, *Vaani* uses assignment, assert, copy and MPI instructions. Loops and if statements are represented in iteration spaces

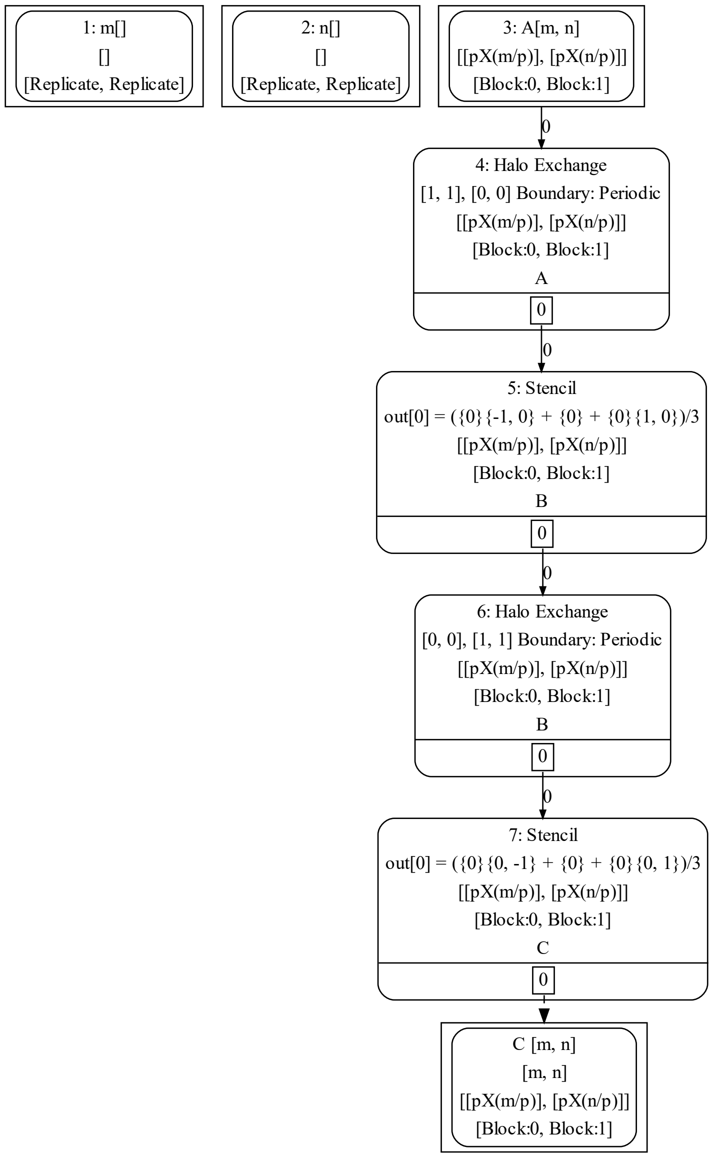


Figure 3.18: Blur LLIR

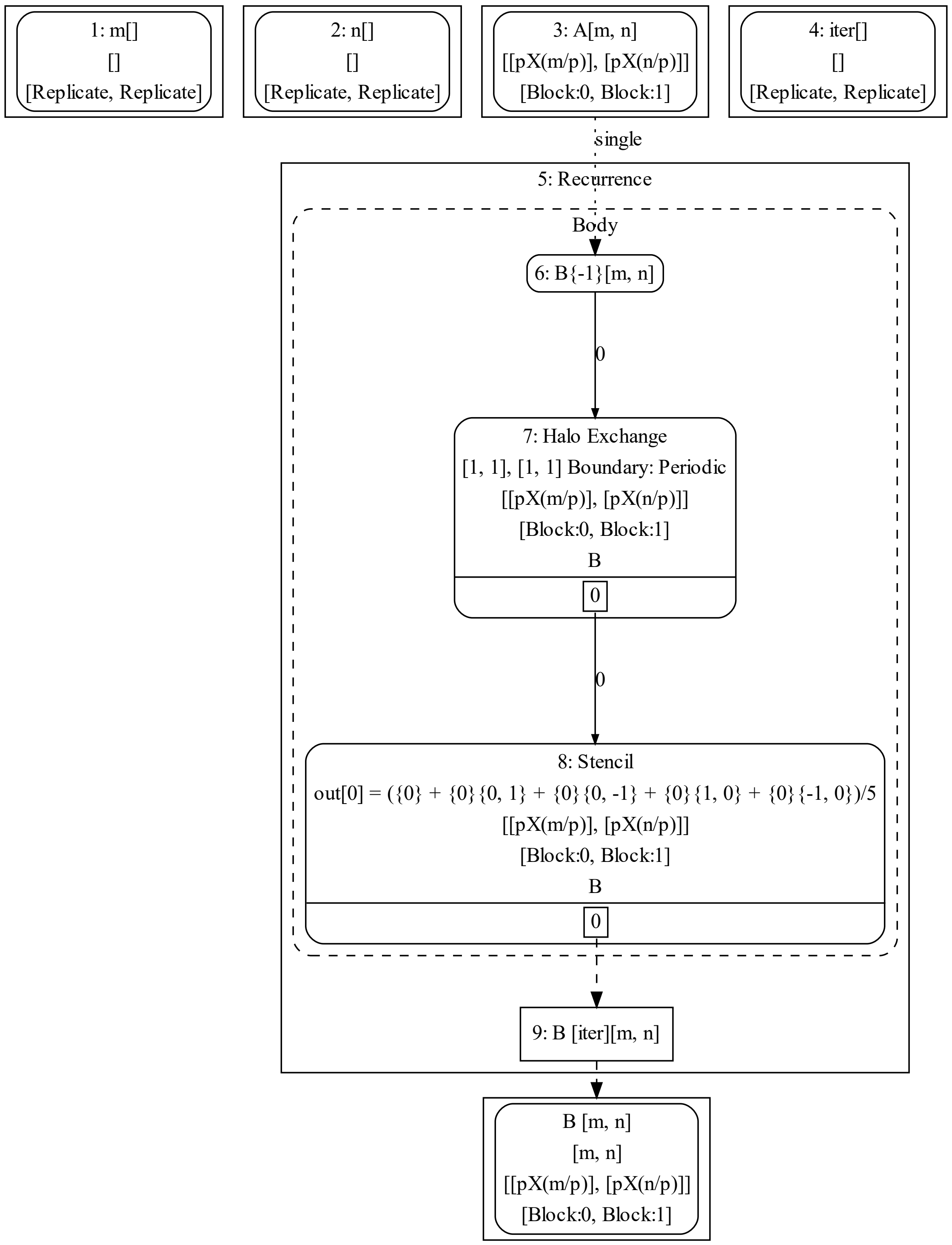


Figure 3.19: Jacobi2D LLIR

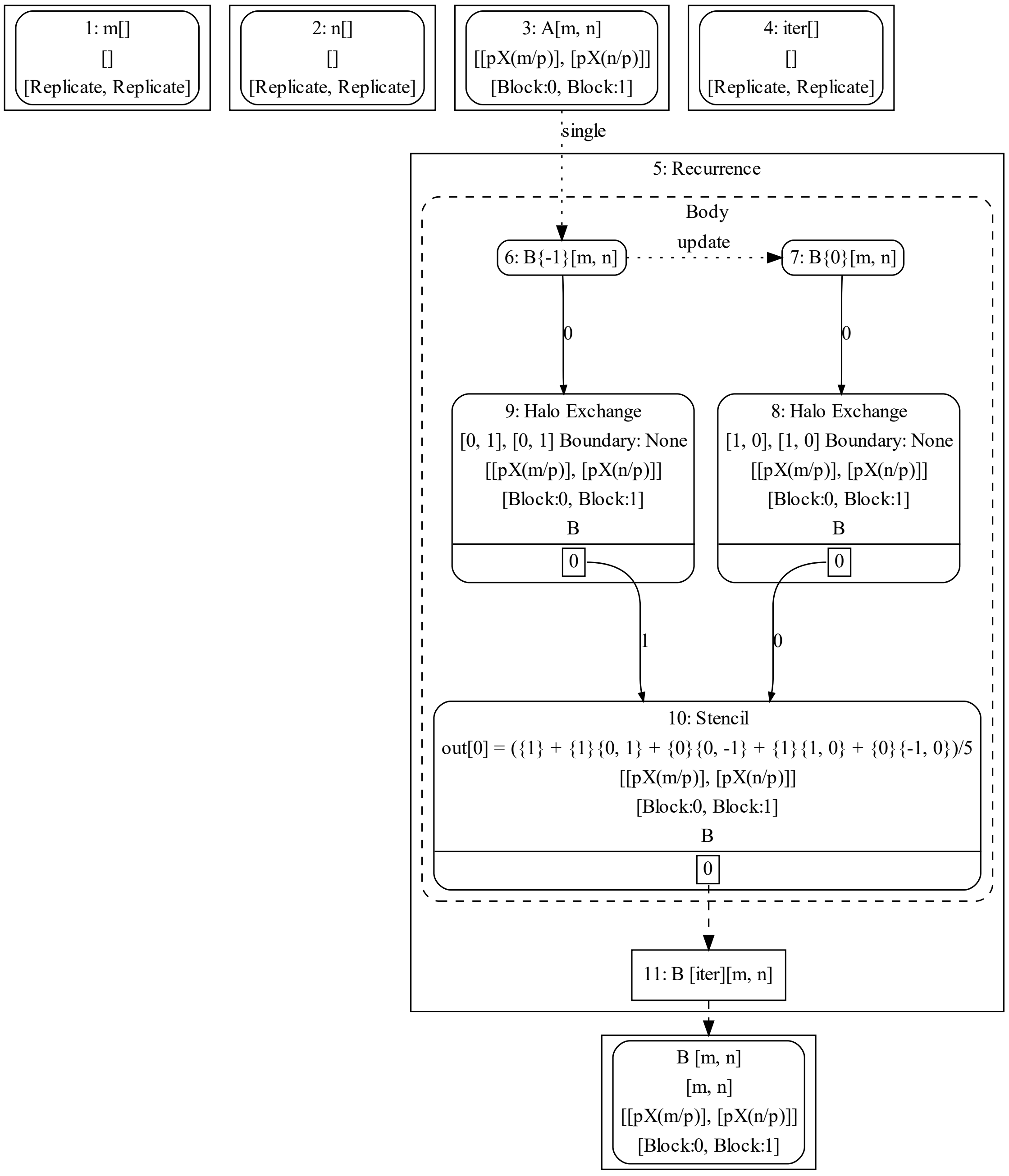


Figure 3.20: Gauss2D LLIR

and masks, respectively. Assignment takes two arguments, an LHS and an RHS, where LHS is an indexed array (or a scalar) and RHS is an expression. MPI instruction is an instruction that calls an MPI function. CLIR has functions defined for all the MPI functions currently used by *Vaani*.

CHAPTER 4: COMPILATION PROCESS

This chapter describes the compilation process in *Vaani* and its implementation, from the input specified in Chapter 2 to C code using MPI. The generated C code conforms to C99 standard. *Vaani* is written in python 3.

4.1 LEXER AND PARSER

Vaani uses the grammar specified in Figure 2.2 and the precedence rules of table 2.4 to generate an LALR parser using lark-parser [4]. The abstract syntax tree (AST) generated by the parser is converted into a DAG. A symbol table keeps track of all defined variables and is updated with each assignment. For every assignment, the statements in the `with` clause are first processed, and then the actual statements.

Constant folding optimization is performed on the resultant HLIR, where at each node, if all its children are constants, the node is replaced by a recomputed constant.

4.2 TYPE ANALYSIS

Type analysis is performed on HLIR. Inputs to the program are already type annotated. In this step, every intermediate node is annotated with a type. The type in *Vaani* constitutes a datatype and a list of symbolic sizes. Datatype for each internal node depends on the datatype of its inputs. Datatype determination is done using standard C type-casting rules and C standard library definitions of corresponding functions. Size determination is done depending on the type of the node being processed. Algorithm 4.1 describes the size computation for some of the nodes in *Vaani*.

Element computes the output size for binary element-by-element operations. Unary operations just retain the shape of the input, while multi-child operations can be assumed to have the same function called iteratively. The inputs to the function, *lsizes* and *rsizes* are lists of sizes, of the two operands to an element-by-element operation. *reslen* is the size of the return type, which is maximum of the two input sizes. Pad pads a list of sizes with 1s to obtain an array of requested dimensions. For example, a 2-d array of sizes $[m, n]$ is padded with two 1s to obtain a 4-d array of sizes $[1, 1, m, n]$. Then, going down the lists of sizes, if both *lsizes* and *rsizes* are equal, then that is the size of the output array in that dimension. And if either of them is 1, then the size would be the other. If both are not equal, and atleast one of them is not 1, then it raises an error.

Algorithm 4.1 Array Shape Analysis

```
  procedure ELEMENT($lsizes$, $rsizes$)
      $reslen \leftarrow max(size(lsizes),$
                  $size(rsizes))$
      $lsizes \leftarrow$ PAD($lsizes$, $reslen$)
      $rsizes \leftarrow$ PAD($rsizes$, $reslen$)
      $osizes \leftarrow list(reslen)$
      for $i \leftarrow 1, reslen$ do
          if $lsizes[i] = rsizes[i]$ or
                $rsizes[i] = 1$ then
              $osizes[i] \leftarrow lsizes[i]$
          else if $lsizes[i] = 1$ then
              $osizes[i] \leftarrow rsizes[i]$
          else
              ERROR("Incompatible sizes")
          end if
      end for
      return $osizes$
  end procedure

  procedure MATMUL($lsizes$, $rsizes$)
Require: $size(lsizes) \leq 2$, $size(rsizes) \leq 2$
      $lsizes \leftarrow$ PAD($lsizes$, 2)
      $rsizes \leftarrow$ PAD($rsizes$, 2)
      ASSERT($lsizes[2] = rsizes[1]$)
      return $[lsizes[1], rsizes[2]]$
  end procedure

  procedure REDUCE($sizes$, $axes$)
      $osizes \leftarrow list()$
      for $i \leftarrow 1, size(sizes)$ do
          if $i$ not in $axes$ then
              $osizes.append(sizes[i])$
          end if
      end for
      return $osizes$
  end procedure

  procedure PAD($sizes$, $len$)
      $pad \leftarrow len - size(sizes)$
      return $[1] * pad + sizes$
  end procedure

  procedure TRANSPOSE($sizes$)
Require: $size(sizes) \leq 2$
      $sizes \leftarrow$ PAD($sizes$, 2)
      return $[sizes[2], sizes[1]]$
  end procedure

  procedure REPLICATE($sizes$, $repvals$)
      $reslen \leftarrow max(size(sizes),$
                  $size(repvals))$
      $sizes \leftarrow$ PAD($sizes$, $reslen$)
      $repvals \leftarrow$ PAD($repvals$, $reslen$)
      $osizes \leftarrow list(reslen)$
      for $i \leftarrow 1, reslen$ do
          $osizes[i] \leftarrow sizes[i] * repvals[i]$
      end for
      return $osizes$
  end procedure

  procedure RESHAPE($oldsizes$, $newsizes$)
      $oldsize \leftarrow$ TOTALSIZE($oldsizes$)
      $newsize \leftarrow$ TOTALSIZE($newsizes$)
      ASSERT($oldsize = newsize$)
      return $newsizes$
  end procedure

  procedure REORDER($sizes$, $order$)
      $dims \leftarrow size(sizes)$
      $osizes \leftarrow list(dims)$
      for $i \leftarrow 1, dims$ do
          $osizes[i] \leftarrow sizes[order[i]]$
      end for
      return $osizes$
  end procedure

  procedure TOTALSIZE($sizes$)
      $size \leftarrow 1$
      for $sz$ in $sizes$ do
          $size \leftarrow size * sz$
      end for
      return $size$
  end procedure
```

MatMul is for matrix multiplication. Here, the two sizes must be less than or equal to 2, as matrix multiplication is only defined for vectors and matrices. Here, the two sizes are again padded to length 2, to make vectors also matrices. Then, it checks that the inner two sizes are equal, and creates an output size by taking the outer two sizes.

Reduce is for reduction, while scan retains the shape of its input. Reduce removes the dimensions in which the reduction is performed. For example, a reduction in dimension 2 of a three dimensional array of size $[m, n, k]$ yields a size of $[m, k]$.

Transpose is for matrix transpose, and interchanges the two indices after padding to length 2. Replicate, Reshape and Reorder are for the replicate, reshape and reorder functions supported in *Vaani*. Replicate takes a list of sizes, *sizes*, and a list of values *repvals*, pads both to get the same length, and performs element by element multiplication in each dimension to replicate the array *repvals* times. Reshape recasts the entire array, so the only check in Reshape is to assertain that the total size of the array before and after reshape is the same. Here, TotalSize computes the total size of the array by multiplying the sizes in each dimension. Reorder views the old dimensions in a new order, so the output sizes are a permutation of the input sizes, based on the order specified.

4.3 HLIR TO MLIR TRANSLATION

Each node in HLIR is converted into a node in MLIR. Offset indexing nodes are converted into `Stencil` nodes; all element-by-element nodes are converted into `Map` nodes; transpose, flip, replicate, reshape and reorder are converted into `Rearrange` nodes; matrix multiplication and matrix power are converted into `Multiplication` nodes; reduce and scan operations into `Combine` nodes and recurrences remain recurrences, with each internal node converted into an MLIR node. This is depicted in Algorithm 4.2. MLIR graph at this stage is shown for GEMVER and Jacobi in Figures 4.1 and 4.2.

4.4 MLIR NODE MERGING

Nodes of MLIR are merged to generate the nodes represented in the MLIR Figures 3.8 through 3.13. This merge is done in two steps, to account for the boundary value specifications in the computation. In the first iteration, only nodes that belong to a single line of source code are merged. The boundary conditions are then validated, and a second round of merging is performed. The merge in this stage is not aggressive, only nodes that have a direct parent-child relationship are merged. This is to allow greater flexibility in later stages, such

Algorithm 4.2 HLIR to MLIR translation

```
procedure TRANSLATE($HLIR$)
    $MLIR = newMLIR$
    for $output$ in $HLIR.outputs$ do
        $MLIR.outputs.append($
            TRANSLATENODE($output$))
    end for
end procedure

procedure TRANSLATENODE($node$)
    if $node.type$ in $[Input, Constant]$ then
        return $node$
    else if $node.type$ is $Expression$ then
        return TRANSLATEEXPR($node$)
    else if $node.type$ is $Recurrence$ then
        return TRANSLATEREC($node$)
    end if
end procedure

procedure TRANSLATEREC($node$)
    for $v$ in $node.recvars$ do
        $v.init$ =TRANSLATENODE($v.init$)
    end for
    for $o$ in $node.recoutputs$ do
        $o.def$ =TRANSLATENODE($o.def$)
    end for
    return $node$
end procedure

procedure TRANSLATEEXPR($node$)
    $newinputs = []$
    for $input$ in $node.inputs$ do
        $newinput.append($
            TRANSLATENODE($input$))
    end for
    if $node.type$ in $[MatMul, MatPow]$ then
        $mlirtype = Multiplication$
    else if $node.type$ in
            $[Transpose, Flip, Reshape,$
            $Reorder, Replicate]$ then
        $mlirtype = Rearrange$
    else if $node.type$ in $[Reduce, Scan]$ then
        $mlirtype = Combine$
    else if $node.type$ in $[Offset]$ then
        $mlirtype = Stencil$
    else
        $mlirtype = Map$
    end if
    $func$ =GENERATEFUNCTION($node$)
    $mlirnode = mlirtype(newinputs,$
                    $func)$
    return mlirnode
end procedure
```

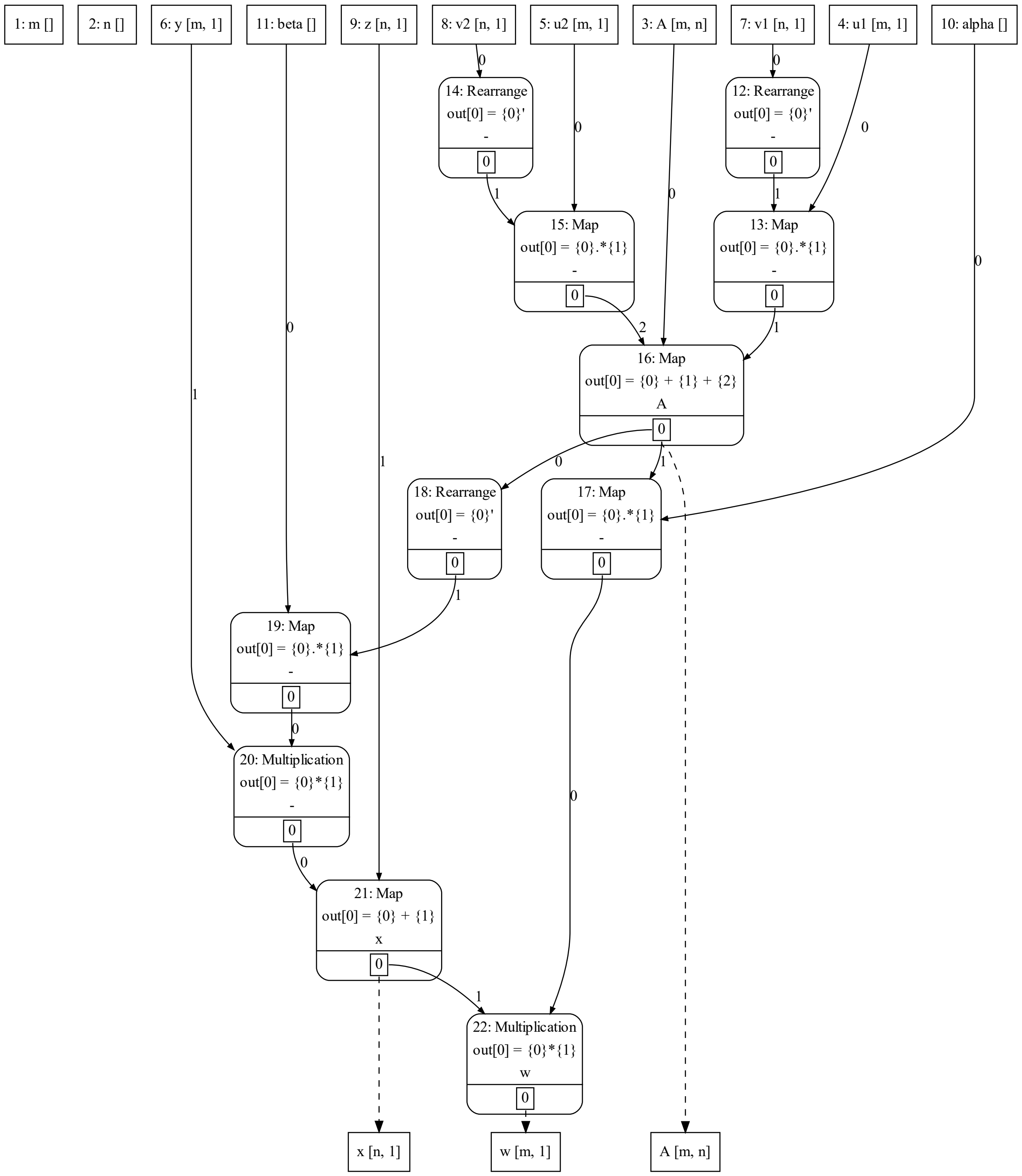


Figure 4.1: GEMVER MLIR before Node Merging

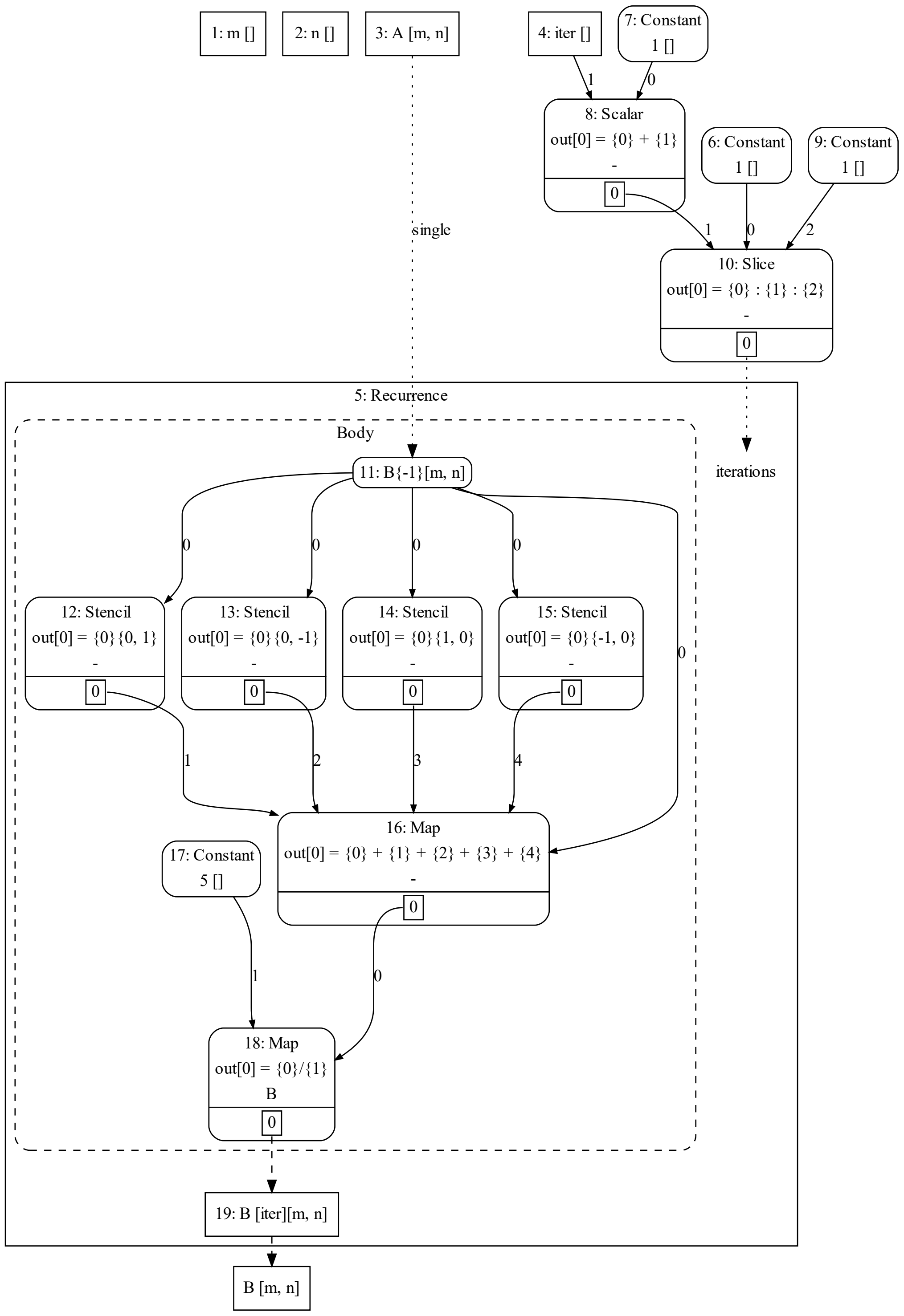


Figure 4.2: Jacobi2D MLIR before Node Merging

as possible overlap of communication and computation. However, this stage exposes the IR for user to manually select nodes to be merged, if desired.

Algorithm 4.3 MLIR merge

```
procedure MERGE($MLIR$)
    for $output$ in $MLIR.outputs$ do
        $output$ =TRYMERGE($output$)
    end for
end procedure

procedure TRYMERGE($node$)
    if $node$ is $Input$ or $Constant$ then return node
    end if
    $mergenodes$ = [$inp$ for $inp$ in $node.inputs$ if CANMERGE($inp$, $node$)]
    $mergenodes$ =CHECKCYCLE($node$, $mergenodes$)
    if non-empty($mergenodes$) then
        $newnode$ =MERGENODES($mergenodes \cup \{node\}$)
        return TRYMERGE($newnode$)
    else
        for $inp$ in $node.inputs$ do
            $inp$ =TRYMERGE($inp$)
        end for
        return $node$
    end if
end procedure
```

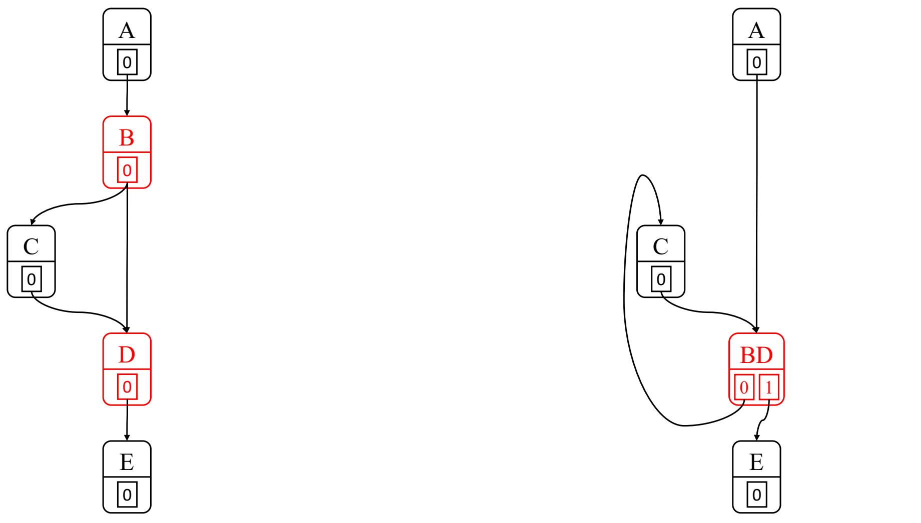
Both iterations of merge begin with the output nodes, and walk backwards towards the inputs after performing all possible merges at a node recursively. This is described in Algorithm 4.3. Here, the TRYMERGE function returns if the node is a leaf node. Else, it tries to merge the node with as many children as it can legally merge. This is checked in two steps.

First, each input node is checked to see if a merge is possible with the base node, that is the node at which the merge is performed. Table 4.1 displays the nodes that are allowed to be merged, and any conditions they must satisfy for the merge to be valid.

Another important requirement to merge nodes is to ensure that no cycle is formed on merging the nodes. Figure 4.3 illustrates how merging two nodes in a DAG may lead to a cycle. Given a set of nodes to be merged, if there is path from one node in the set to another through a node that is not present in the set, a cycle will be formed. In the example, merging B and D is not valid as there is a path from B to D via the node C which is not also merged with B and D. To detect these cycles, we start with the inputs to the set as described in

Base Node	Allowed merge nodes	Condition
Map	Map	-
	Stencil	-
	Scalar	-
	Constant	-
Stencil	Map	-
	Stencil	-
	Scalar	-
	Constant	-
Multiplication	Multiplication	-
	Map	Scalar multiplication
	Rearrange	Matrix transpose
	Constant	-
Rearrange	Constant	-
Combine	Constant	-

Table 4.1: Allowed merge nodes



(a) Original graph (b) Graph after merging

Figure 4.3: Formation of a cycle on merging nodes B and D

Algorithm 4.4 Cycle detection while merging nodes

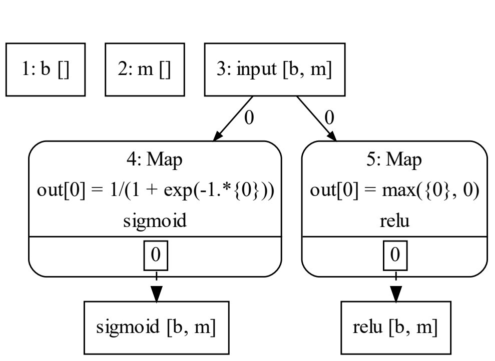
```
procedure CheckCycle($node$, $mergenodes$)
    $inputs$ = UniqueInputs($mergenodes \cup \{node\}$)
    while NotEmpty($inputs$) do
        $newinputs = inputs.pop().inputs$
        if $empty(newinputs \cap mergenodes)$ then
            $inputs = inputs \cup newinputs$
        else
            $mergenodes = mergenodes \setminus newinputs$
            return CheckCycle($node$, $mergenodes$)
        end if
    end while
    return $mergenodes$
end procedure
```

the Algorithm 4.5, and walk backwards, seeing if any of its previous inputs are present in the set of nodes to be merged. Algorithm 4.4 demonstrates this method for cycle detection. Walking through the algorithm for the example in Figure 4.3, the *node* would be D and the set of *mergenodes* would be $\{B\}$. The function UniqueInputs creates a list of inputs to the set of nodes that remove redundancies, and also eliminate inputs that are in the set itself. In the example, node B which is an input to D is disregarded as B and D will be merged. The unique inputs to B and D would be A and C. In the *while* iteration, if the first input node considered is A, it has no inputs and thus does not change anything. The next input to be popped is C, whose input is B. And since B is in *mergenodes*, it creates a cycle. B is removed and the algorithm is called again without B, and it terminates as there are no more *mergenodes*. It should be noted that we could as well have started with the outputs of the nodes and walked forwards through the dependency graph, instead.

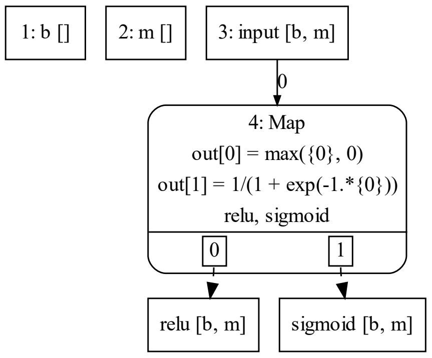
Algorithm 4.5 describes how two nodes are merged in MLIR. First, the constant nodes are separated from the other nodes to be merged, and are folded into the functions of each node. Then, the inputs and outputs of the merged node are determined. A node is an input to the merged node if it is an input to any of the nodes to be merged, and it is not the output of any of the nodes to be merged. And, a node is an output to the merged node if it is an output to a node to be merged, and it has dependencies beyond the nodes to be merged. The function is merged by taking the new selected outputs and tracing them backwards to the new set of inputs through the old set of functions. A new node is created depending on the type of the nodes merged, and the dependency graph is updated accordingly. The procedure GetResultType describes how the type of the merged node is determined.

Algorithm 4.5 Merging nodes

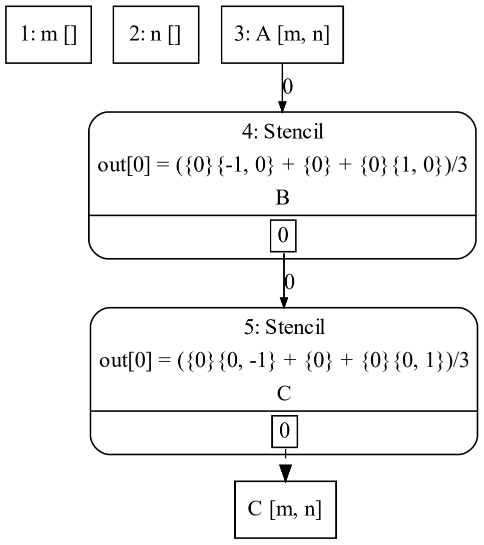
```
procedure MergeNodes(nodelist)
    constants = [node for node in nodelist if node is Constant]
    nodelist = nodelist $\setminus$ constants
    FoldConstants(nodelist, constants)
    inputs = $\bigcup_{node \in nodelist} node.inputs$
    outputs = $\bigcup_{node \in nodelist} node.outputs$
    inputs = inputs $\setminus$ outputs
    outputs = [op for op in outputs if nonempty(dep(op) $\setminus$ nodelist)]
    function = MergeFunction(nodelist, inputs, outputs)
    nodetype = GetResultType([node.type for node in nodelist])
    node =Create(nodetype, function, inputs, outputs)
    UpdateDependencies()
    return node
end procedure

procedure GetResultType(typelist)
    if any(typelist = Multiplication) then
        return Multiplication
    else if all(typelist = Scalar) then
        return Scalar
    else if all(typelist = Stencil or typelist = Map) then
        if any(typelist = Stencil) then
            return Stencil
        else
            return Map
        end if
    else
        return “Unable to determine type”
    end if
end procedure
```



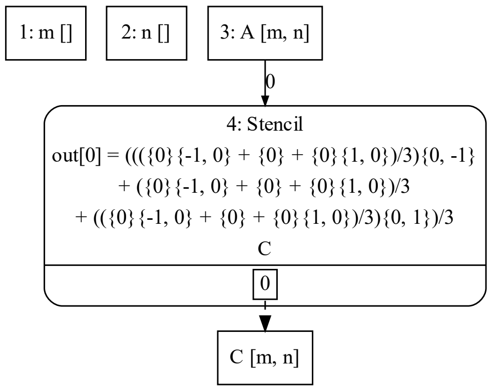
(a) Activation MLIR before merge



(b) Activation MLIR after merge


1: m []
2: n []
3: A [m, n]
0
4: Stencil
out[0] = ({0}{-1, 0} + {0} + {0}{1, 0})/3
B
0
0
5: Stencil
out[0] = ({0}{0, -1} + {0} + {0}{0, 1})/3
C
0
C [m, n]

(c) Blur MLIR before merge



(d) Blur MLIR after merge

Figure 4.4: Examples of user initiated merge nodes

4.4.1 User Initiated Node Merging

The user can initiate merging of nodes, by specifying a list of nodes in the MLIR to be merged. For example, the two activation functions `sigmoid` and `relu` in activation function from Figure 3.10 can be merged in *Vaani* by calling `merge(4, 5)`, where 4, 5 are the node numbers in MLIR. Similarly, Blur in 3.11 has two stencil nodes, with indices 4 and 5, and they can be merged in *Vaani* using `merge(4, 5)`. The effect of these merges are shown in Figure 4.4. The merge follows the same steps given in 4.5.

4.5 GRID CREATION

The generated program is intended to run as a set of autonomous processes executing their own code in an MIMD style with communication across the processes. *Vaani* views these processes as a multi-dimensional grid. The creation of a symbolic grid in *Vaani* is explained in this section.

A grid is created in MLIR using a command `grid(<dims>, <sizes>, <indices>)`, where `<dims>` mentions the number of dimensions in the grid, and must be a constant integer. `<sizes>` and `<indices>` are lists of strings, where the strings represent variable names that are not yet present in the program. The `<sizes>` specify the size of the grid in each dimension, while the `<indices>` specify the index of a process in the grid. These sizes and indices, in the final program, are initialized to the grid dimensions and to uniquely identify a process in the multidimensional grid, respectively, and are used in *Vaani* as symbolic placeholders. If `<sizes>` or `<indices>` are not specified, *Vaani* generates them automatically, and tries to keep them consistent and readable.

A command `grid(1)` creates a 1-dimensional grid with auto-generated sizes and indices, while `grid(1, ['p'], ['rank'])` creates a 1-dimensional grid where `p` would hold the total number of processes, and `rank` would hold the index of a given process (also known as a rank in MPI terminology).

A command `grid(2, ['p', 'q'])` would create a 2-dimensional rectangular grid, while the command `grid(2, ['p', 'p'])` would create a 2-dimensional square grid. This only works if the total number of available nodes is a perfect square, else the program would terminate.

Vaani supports higher order grid creation, using `grid(3, ['p', 'q', 'r')` or `grid(3, ['p', 'p', 'p'])` where the latter terminates if the number of nodes is not a perfect cube. While *Vaani* also allows for creation of grids like `grid(3, ['p', 'q', 'p')`, *Vaani* currently relies on automatic grid dimension creation and returns an error if the partition does not match the expected split. So even if a split was possible, it is not guaranteed that *Vaani* would find it. For instance, if the total number of nodes is 12, a partition of $[2,3,2]$ would fit the model and such a grid would be successful, but if the default grid created returns $[3,2,2]$, the program would take that to be an error.

Since the generated program uses MPI for communication, we use the concept of communicators from MPI, which is a channel to communicate with a set of processes. *Vaani* also provides handles to symbolic communicators that can be referenced to define communication patterns. In MPI, the global communicator is called `MPI_COMM_WORLD`. *Vaani* provides a handle to a copy of the global communicator, and in the case of multi dimensional grids, a handle to communicators in each dimension. So, for a 2-dimensional grid, *Vaani* has handles for a global, row and column communicators.

4.6 PARTITIONING AND MAPPING

This section describes how data is partitioned and mapped onto the symbolic grid created in the previous section 4.6.

4.6.1 Data Partitioning

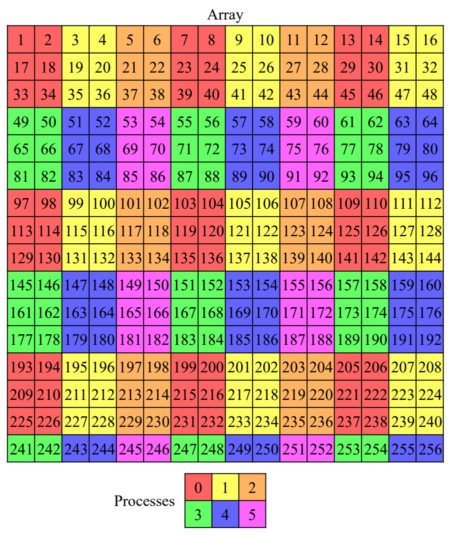
A dimension of a data can be partitioned in two ways, either by specifying a block size b, or by specifying the number of pieces k. A dimension i of a d-dimensional array thus partitioned is treated as two dimensions i_1 and i_2, such that, if the original dimension i is of size n, the new dimensions are $n/b \times b$, or $k \times n/k$.

The case where expressions n/b or n/k do not generate integer values (which is the assumption in general) are handled as:

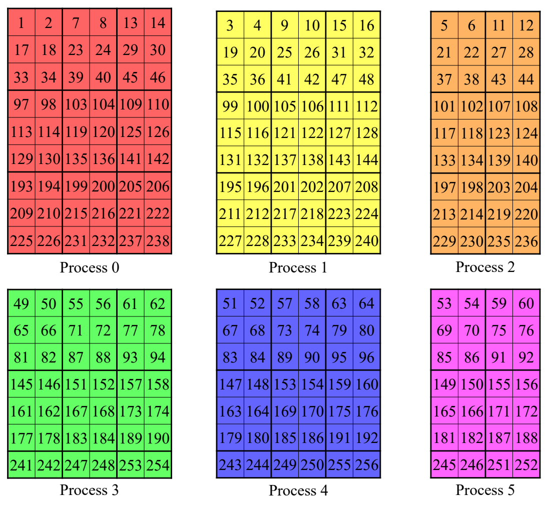
1. If the block size b is specified, all the blocks have size b except the last block which will have $n \mod b$ elements.

2. If the number of pieces k is specified, then let $n = q * k + r$, where $0 \leq r < k$. The first r pieces have $q + 1$ elements, while the next $k - r$ pieces have q elements.

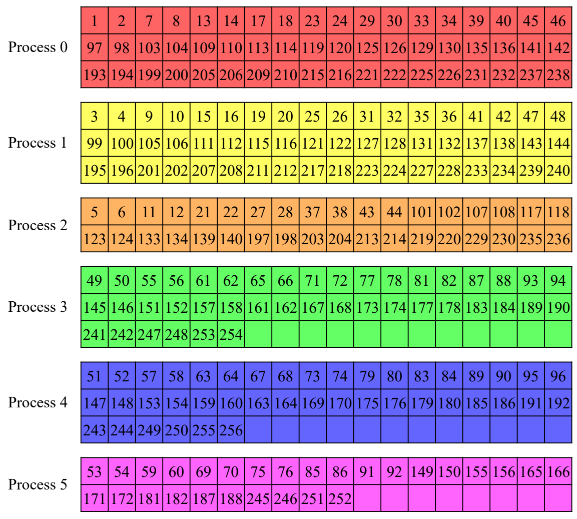
In *Vaani*, partitioning can be performed on MLIR by using the commands `block(<name>, <dim>, <size>)`, which splits `<dim>` into blocks of size `<size>`, or `chunk(<name>, <dim>, <size>)`, which splits `<dim>` into `<size>` number of pieces. `<name>` is a string specifying the name of a variable.



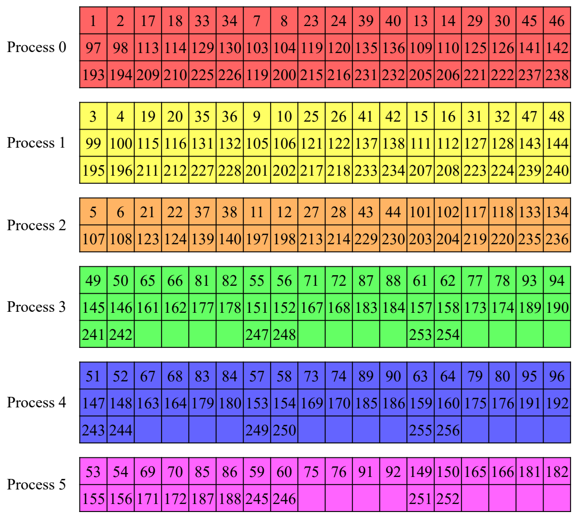
(a) Distribution from array view



(b) Distribution from process view



(c) Layout with `group=False`



(d) Layout with `group=True`

Figure 4.5: Block cyclic distribution of 16×16 array on a 2×3 grid with a block size of 3×2

4.6.2 Mapping

A d-dimensional array (or a partitioned array) `A` can be mapped to a k dimensional process grid `G`, by specifying which dimension of `A` is mapped to which dimension of the process grid, and if the distribution is block or cyclic.

Block partitioning a dimension is equivalent to data partitioning the dimension into k pieces as described earlier, and mapping one-to-one onto the grid dimension, assuming the size of the grid is k in that dimension. For example, a mapping of `[('block', 0, 0), ('block', 1, 1)]` partitions a 2 dimensional array onto a 2 dimensional grid by blocking in the x and y direction.

Cyclic partitioning assigns indices of a dimension of data in a round robin fashion to the indices of a process grid. It must be noted that both block and cyclic distributions produce equal partition sizes, only the index mapping is different.

Block cyclic partitions can be specified by first blocking the dimension, and then applying a cyclic mapping. For example, a matrix A of size $m \times n$ is partitioned using `block('A', 0, 'b')`, `block('A', 1, 'b')` and a mapping of `[('cyclic', 0, 0), ('cyclic', 1, 1)]` gives a block cyclic distribution on a two dimensional process grid. Figures 4.5a and 4.5b show block cyclic distribution for a 16×16 array distributed on a 2×3 grid with a block size of 3×2.

If the dimension of the grid is larger than the mapped dimensions of the array, the user can specify whether the data is placed on a unique node in that dimension or replicated along that dimension. For example, a mapping of `[('block', 0, 0), ('unique', 1, 1)]` maps a one dimensional array onto a two dimensional grid where the array is distributed in dimension 0 (x), and present on the first index of dimension 1 (y). A mapping of `[('block', 0, 0), ('block', 1, 1), ('replicate', 2)]` maps a 2 dimensional array onto a three dimensional grid by blocking the two dimensions of the array on two of the dimensions of the grid, and replicating the array in the third dimension. This can be used, for example, to perform matrix multiplication using a 3D grid.

You can choose to group the data of an array, which takes the dimensions not distributed on the grid and makes that a unit. Figures 4.5c and 4.5d shows the data layout for block cyclic distribution of an array on a 2 dimensional grid as a flattened array in memory. It must be noted that the data has not actually been laid out at this stage, and grouping does not correspond to any communication or data movement, it just transforms the internal representation of the data.

4.6.3 Representation in MLIR

Partition and mapping of an array A of type t onto a grid G is represented in *Vaani* using an `ArrayGridMap`. An `ArrayGridMap` has partitioning information for each dimension of the array (as a recursive block or chunk partitioned size), and a mapping information for each dimension of the grid. These mapping are of the following types.

1. Block: block map a dimension of the array to this grid dimension.
2. Cyclic: cyclic map a dimension of the array to this grid dimension.
3. Unique: the array is present on only one process in this grid dimension.
4. Replicate: the array is replicated on every process in this grid dimension.
5. Partial: the array values are distributed on every process in this grid dimension, and they need an operation to collect the actual values.

4.7 MLIR ALGORITHM SELECTION

Vaani has a provision to annotate some nodes in MLIR with special instructions on lowering to LLIR. An example is to choose simple halo exchange or to overlap communication and computation in a stencil node. In a simple halo exchange, the stencil node in LLIR would create one computation node and a halo exchange node, while the LLIR for overlap would create a computation node for the boundaries, a computation node for the internal values, and a communication node that is dependent only on the boundary computation. *Vaani* is built to easily support and extend other choices, but are currently not supported.

Another way for the user to select algorithms is for multiplication node by specifying partitioning and mapping information for the node.

4.8 PARTITION AND MAP TYPE ANALYSIS

Once the user specifies the grid, selected partitioning and mapping, and possible algorithm choices, *Vaani* performs type analysis to align partitioned and mapped data at each node. This step fills details that are missing in user specification, verifies consistency in case of existing specifications, and adds rearrangement nodes as required.

To perform this analysis, *Vaani* tries to start from points that have an `ArrayGridMap` specified, and spread this mapping outwards at each of these places. This algorithm is

Algorithm 4.6 Mapping Type Analysis (Overall)

```
procedure Analyze($MLIR$)
    BuildLists($MLIR$)
    while MLIR.processnodes do
        Iterate()
    end while
end procedure

procedure Iterate($MLIR$)
    $progress = False$
    $next = []$
    for $node$ in $MLIR.processnodes$ do
        if TryMap($node$) then
            $progress = True$
        else
            $next.append(node)$
        end if
    end for
    if $progress$ then
        $MLIR.processnodes = next$
        $MLIR.leaves = [lf$ for $lf$ in
                $MLIR.leaves$ if not $lf.map]$
    else
        $node =$ FindNode($MLIR.leaves$)
        ForceMap($node$)
        $MLIR.leaves.remove(node)$
    end if
end procedure

procedure BuildLists($MLIR$)
    $nodes =$ all intermediate nodes in MLIR
    $MLIR.processnodes = set(nodes)$
    $lvs = MLIR.inputs \cup MLIR.outputs$
    $MLIR.leaves = [lf$ for $lf$ in $lvs$ if not
$lf.map]$
end procedure

procedure FindNode($nodes$)
    $maxdims = -1$
    $possible = None$
    for $node$ in $nodes$ do
        $dims = count(dims$ in $node$ where
                        $size[dim] > 1)$
        if $dims > maxdims$ then
            $maxdims = dims$
            $possible = node$
        end if
    end for
    return $possible$
end procedure

procedure ForceMap($node$)
Require: $node$ is an output or an input
    $node.map =$ Default
end procedure
```

Algorithm 4.7 Mapping Type Analysis (Individual Nodes)

```
  procedure MAP(node)
Require: Atleast one input or output map
      ipmaps = [ip.map for ip in
                node.inputs if ip.map]
      opmaps = [op.map for op in
                node.outputs if op.map]
      cmap = COMMAPS(ipmaps, opmaps)
      for n in $node.inputs \cup node.outputs$ do
          if n.map then
              if not CHECKSIMILAR(cmap,
                          n.map) then
                  Add Rearrange
              end if
          else
              n.map = GETSIMILAR(cmap,
                          n.sizes)
          end if
      end for
  end procedure

  procedure MULTIPLICATIONMV(node)
Require: Matrix map
      if not CHECKMULT(node.inputs)
              then
          Add Rearrange
      end if
      mulmap = GETMULT(node.inputs)
      if node.output.map then
          if not CHECKSIMILAR(mulmap,
              node.output.map) then
              Add rearrange
          end if
      else
          node.output.map = mulmap
      end if
  end procedure

  procedure COMBINE(node)
Require: Atleast one input or output map
      opmap = node.output.map
      ipmap = node.input.map
      if node.operation = "scan" then
          if ipmap & opmap then
              if not CHECKSIMILAR(ipmap,
                          opmap) then
                  Add Rearrange
              end if
          else
              Copy map
          end if
      else
          if ipmap & opmap then
              imr = DROP(ipmap,
                          node.axis)
              if not CHECKSIMILAR(imr,
                          opmap) then
                  Add Rearrange
              end if
          else if ipmap then
              node.output.map = DROP(
                          ipmap, node.axis)
          else
              node.input.map = ADD(opmap,
                          node.axis)
          end if
      end if
  end procedure

  procedure REARRANGE(node)
Require: Atleast one input or output map
      opmap = node.output.map
      ipmap = node.input.map
      if not (ipmap or opmap) then
          ipmap, opmap = GETRAR(ipmap,
                          opmap, node)
      end if
  end procedure
```

presented in Algorithm 4.6. Here, we build two sets, one of the intermediate MLIR nodes, and one of the leaf nodes, both input and output nodes in the program. Here, only the leaf nodes that do not have a map from the user are considered. For each node in the *processnodes*, *Vaani* tries to complete the mapping using the maps of its inputs and/or outputs. If none of these maps are present, then the attempt fails. The actual process of completing these maps are presented in Algorithm 4.7 for all the major types of nodes in MLIR. If no node completes the map in an iteration, then one of the inputs or outputs that have not yet been mapped are selected and a default map is created for that node, and the iteration repeats.

It can be observed that all the node types require some map, either an input or an output map. `Map` and `Stencil` nodes require at least one input or output node already be mapped. ComMaps combines a set of maps of compatible array sizes, and generates one map. This typically looks at each array and grid dimension, and if that array dimension is similarly mapped on all the present maps, it will pick that map. Else, it will look for a majority of the maps, or, give preference to an output map, or in worst case, pick one arbitrarily. If the combined array size is larger that the parts, it tries to map unmapped array dimensions to grid dimensions that are unique or replicated on the smaller arrays. Once a combined map is determined, for each input and output, if it has a map, CheckSimilar checks if the map is compatible with the combined map, and rearrange nodes are inserted otherwise. If there is no map, a compatible map is generated.

`Multiplication` nodes are preprocessed before the analysis to only perform a single multiplication at each node. Matrix powers are transformed into recurrences of matrix multiplications. Thus, the `Multiplication` nodes only have two array inputs with or without transpose. For matrix-vector multiplication, it requires the map for the matrix input. In this case, the vector is the only one that undergoes rearrange, and the matrix is kept intact. For matrix-matrix multiplication, *Vaani* requires at least two of the three involved matrices, the two inputs and the output. However, *Vaani* tries to delay mapping of these nodes to obtain all three maps independently, if possible. This is not possible only if the output of one such multiplication is the input to another, in which case the third map is determined by combining the mapping of the required two maps.

`Combine` nodes are either reduction or scan operations. Scan operations have the same array size in input and output, and thus can have the same map, if either of them is not present. If both the maps are present, then a rearrange is needed if they don't represent the same partitioning. In case of reduce operation, the array dimension of the reduction is removed in the result. Thus, if the output map is present, the axis dimension is added, and if the input map is present, it is removed. If both maps are present, the input map with the

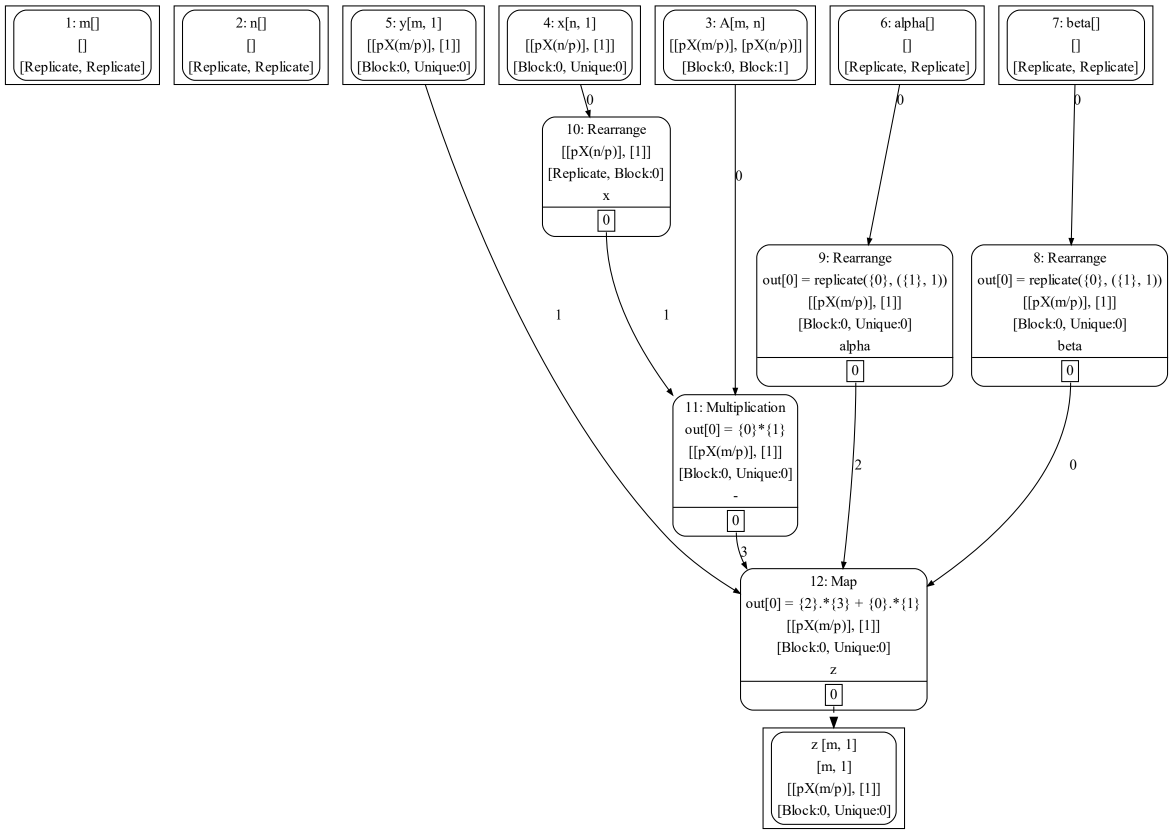


Figure 4.6: GEMV MLIR after partition and map type analysis

axis removed is aligned with the output map and rearrangements are added accordingly.

`Rearrange` nodes perform a specified rearrange operation. This node can take any mapping as input and any mapping as output. Like with multiplication nodes, these nodes also delay their decision to see if both the input and output maps are obtained through other nodes. If it is not possible, then the mapping with the least data movement is created for the other map.

Figure 4.6 shows the MLIR of matrix-vector multiplication GEMV after the partition and map type analysis is completed. Here, a rearrange node is added for each of the scalars *alpha* and *beta* to replicate them to match the vector y, and a rearrange node to transpose and replicate the vector x to perform the matrix multiplication.

4.9 MLIR TO LLIR TRANSLATION

Each node in MLIR translates to a set of nodes in LLIR. The nodes in LLIR are classified as computation nodes and communication nodes. For each array in LLIR, a local array type and mask are generated, depending on the partition and mapping of each node. For example, consider a 2 dimensional grid of size $p \times q$. Let each process be identified by its index in the two grid dimensions $(rowrank, colrank)$. If an array of size $m \times n$ is block distributed on the grid, then the dimensions of the local array will be $mp0 \times nq1$, where $mp0 = m/p + ((rowrank < m\%p)?1 : 0)$ and $nq1 = n/q + ((colrank < n\%q)?1 : 0)$; and the $mask$ will be empty. If a column vector of size $m \times 1$ is distributed across the rows to the first column of the grid (`(block, 0, 0), (unique, 1, 0)`), then the local array will be $mp0 \times 1$, where $mp0$ is previously defined, and a mask of $colrank == 0$. In this context, mask is used as the condition satisfying which a node will have a portion of the array. Here, the array is present on all the nodes, while the vector is only present on the first column.

`Map` node has all the data aligned with rearrange nodes, and thus, only generates a computation node (LLIR Map node).

`Stencil` node also has data aligned with rearrange nodes. However, a boundary exchange is necessary to have all the data required for computation, and thus a BoundaryExchange communication node is created in LLIR, followed by a computation node (LLIR Stencil node).

`Combine` node first generates a local reduction computation node. If the operation is a scan, it generates a Scan communication node, while it generates a Reduce communication node if it is a reduction. Scan nodes also need a third node, a local scan node.

`Multiplication` node, in case of matrix-vector multiplication, first has a rearrange node to align the vector along the right dimension of the array, if needed. Then a multiplication computation node is created. Finally, if the multiplication results in a partial distribution of the final array, then a Reduce communication node is created.

`Rearrange` node looks at the input map, the output map, and the rearrange operation encoded in its function to generate communication nodes. To this effect, it first looks at the indices in the input and output that are mapped to each other in the rearrange (for example, in a transpose, input index 0 is mapped to output index 1 and vice versa). If the aligned index in both the input and output are not mapped to the same grid dimension, then a tuple of array indices and grid dimensions are marked to be realigned. Chains of such realignments are created where the source of one dimension is the destination of another and each chain is processed separately. If the array dimension in the input is not mapped to a grid dimension, but it is mapped to a grid dimension in the output, it implies that the array

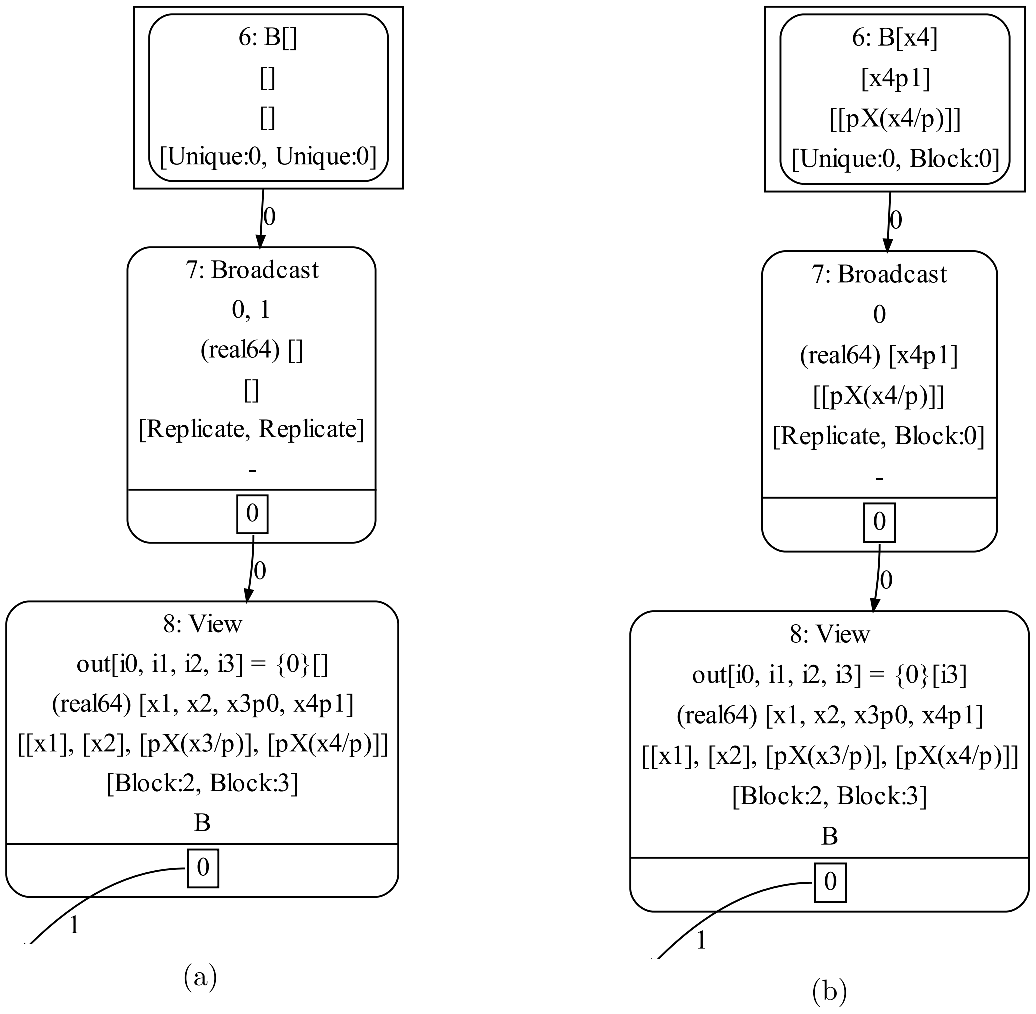
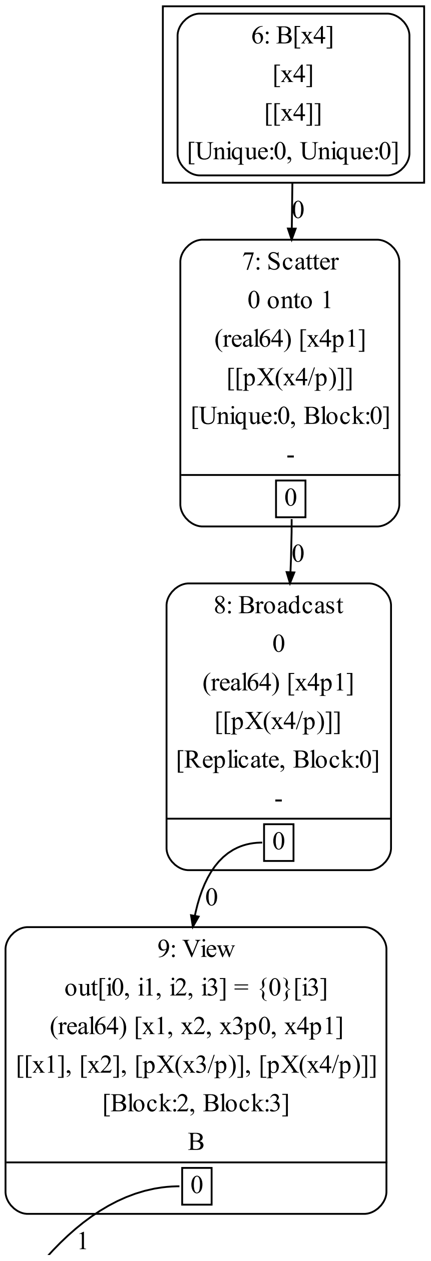


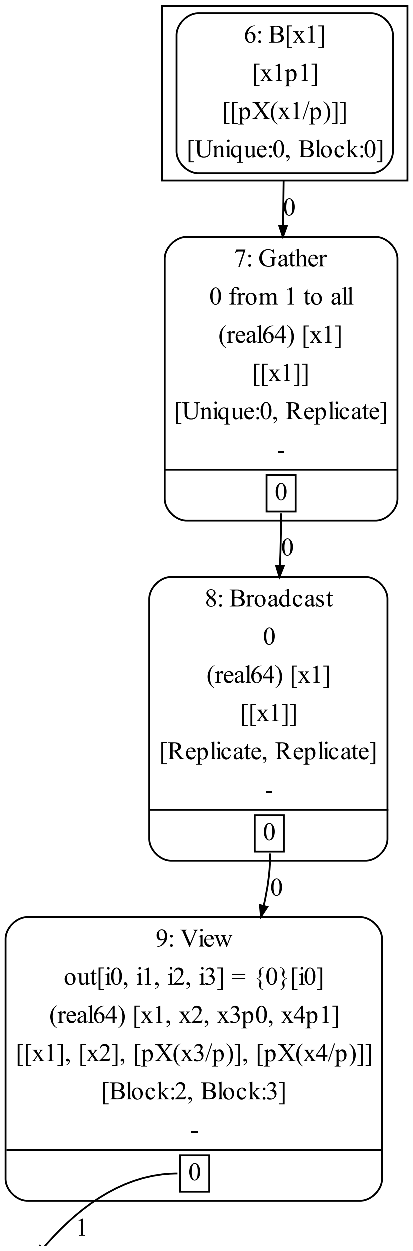
Figure 4.7: Rearrange node generation

must be scattered in that dimension. Similarly, if the array dimension is mapped to a grid dimension in the input, but it is not in the output array, then the array must be gathered. If the chain is circular, like in an array transpose on 2 dimensional grid, where rows and columns are to be interchanged, it is a circular transpose. If the chain only requires the data to transfer from one dimension to another, then it is a realignment. Then, the result from these operations and the final output is considered, and any broadcasts that are necessary are generated. Finally, the partition methods of any array dimension are considered, and if they are not the same, then a repartition node is created. Finally, a view node is created, if needed,to provide a handle to convert from the expected output to the actual output.

This process is demonstrated by the following examples. Consider a 4 dimensional array A of size $[x1, x2, x3, x4]$ mapped onto a two dimensional grid such that the third and fourth dimension (of sizes $x3$ and $x4$) are block distributed on grid dimensions 0 and 1. Consider

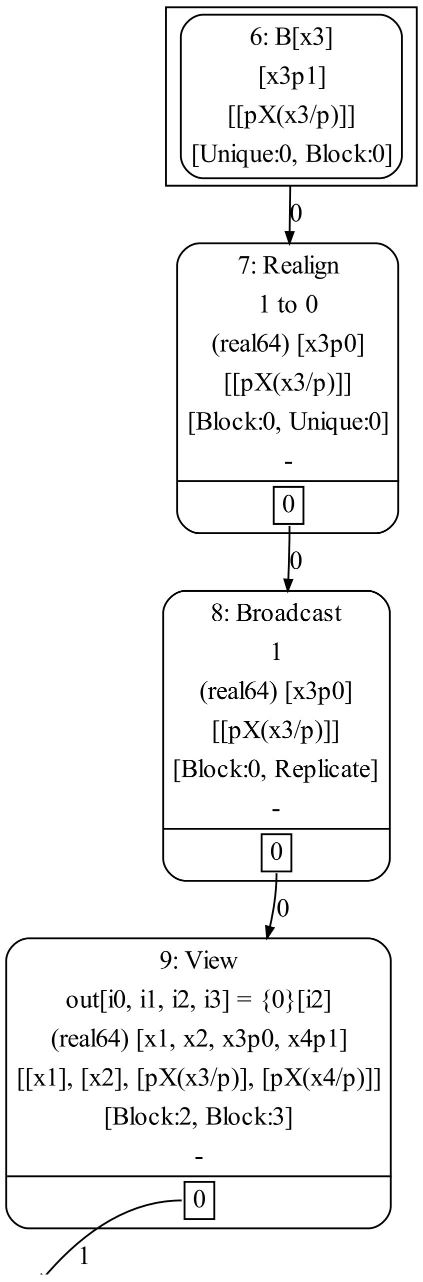


(c)

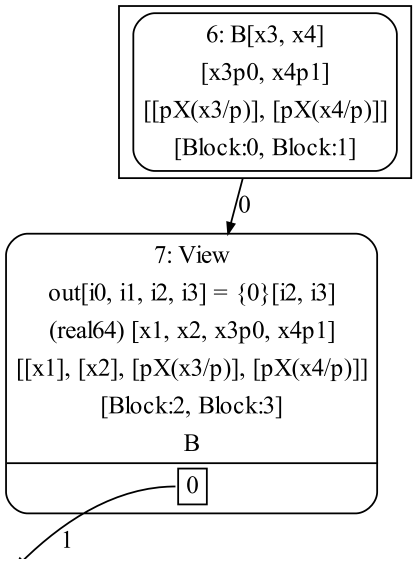


(d)

Figure 4.7: Rearrange node generation (cont.)

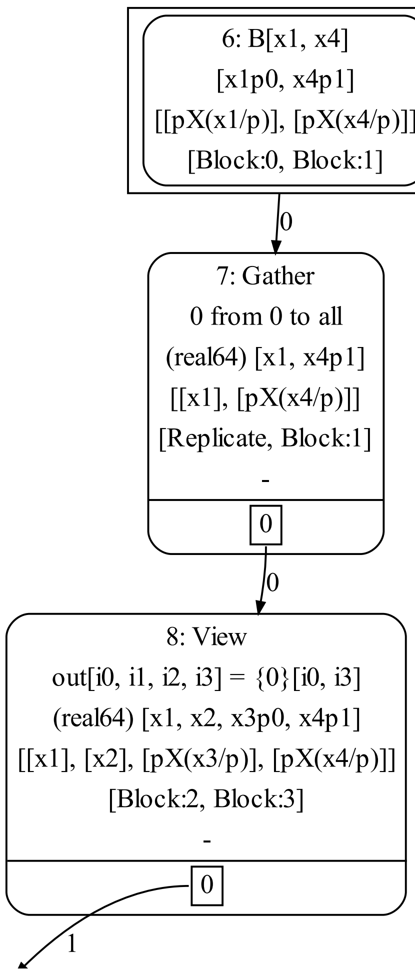


(e)

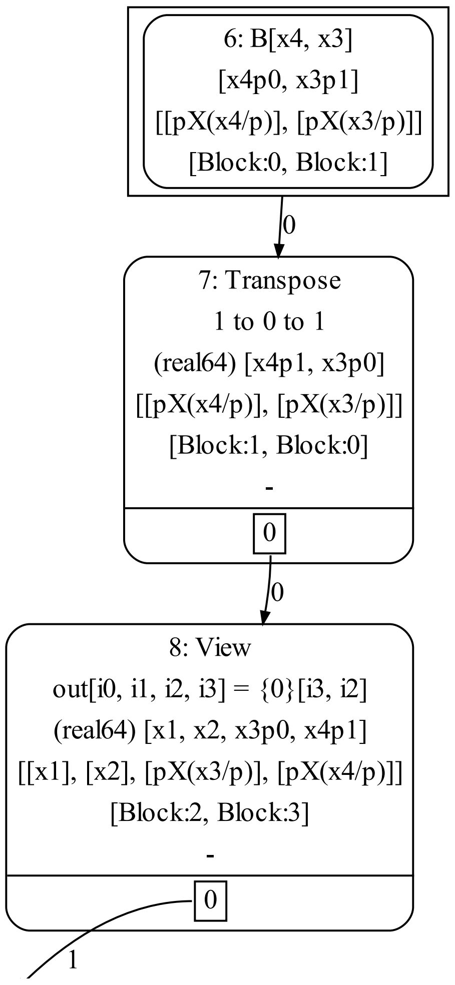

6: B[x3, x4]
[x3p0, x4p1]
[[pX(x3/p)], [pX(x4/p)]]
[Block:0, Block:1]
0
7: View
out[i0, i1, i2, i3] = {0}[i2, i3]
(real64) [x1, x2, x3p0, x4p1]
[[x1], [x2], [pX(x3/p)], [pX(x4/p)]]
[Block:2, Block:3]
B
0
1

(f)

Figure 4.7: Rearrange node generation (cont.)

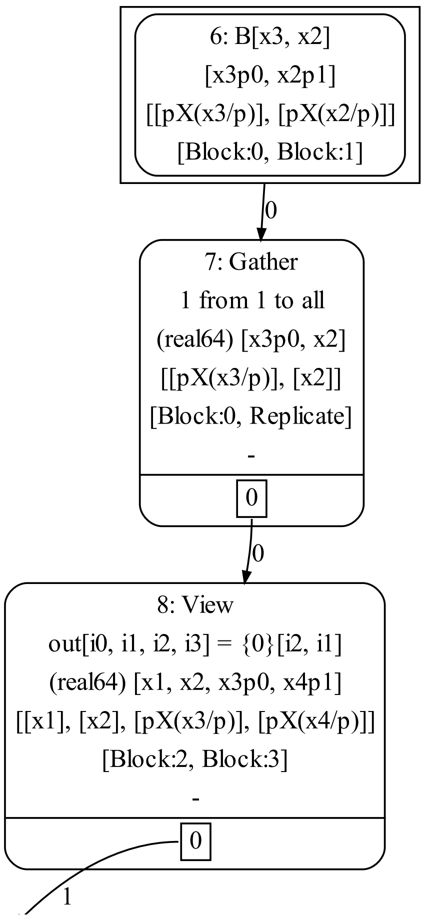


(g)

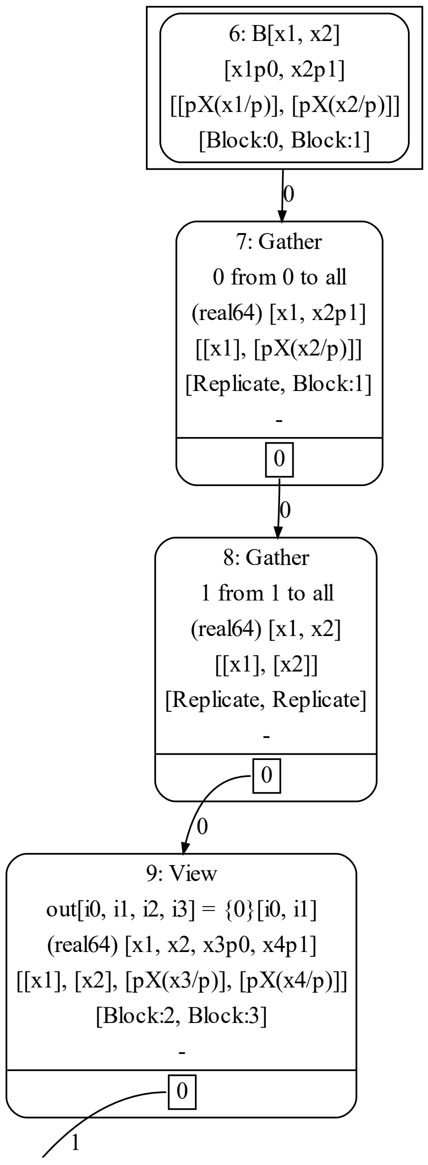


(h)

Figure 4.7: Rearrange node generation (cont.)

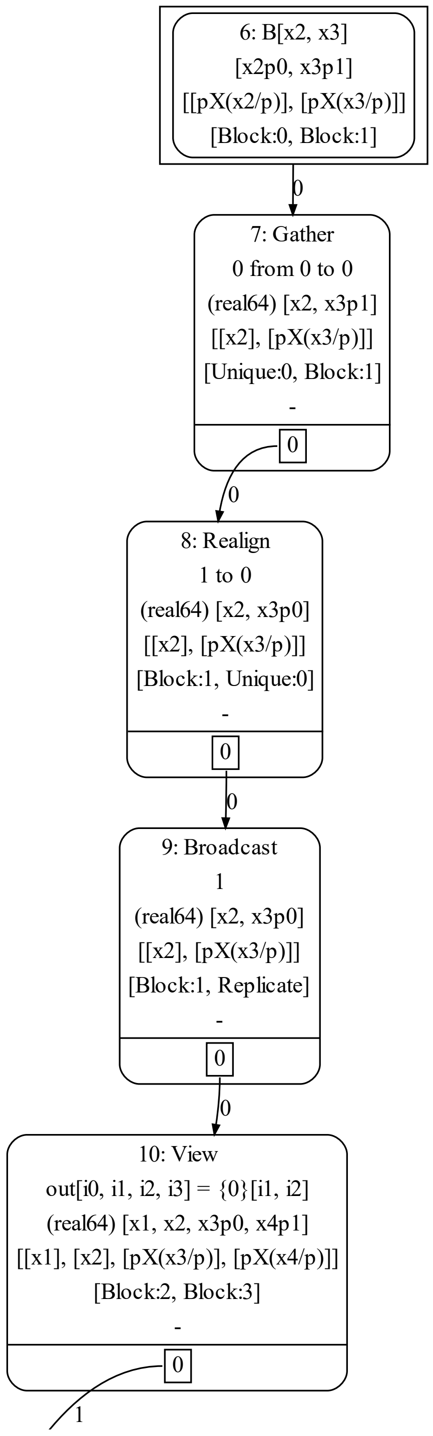


(i)

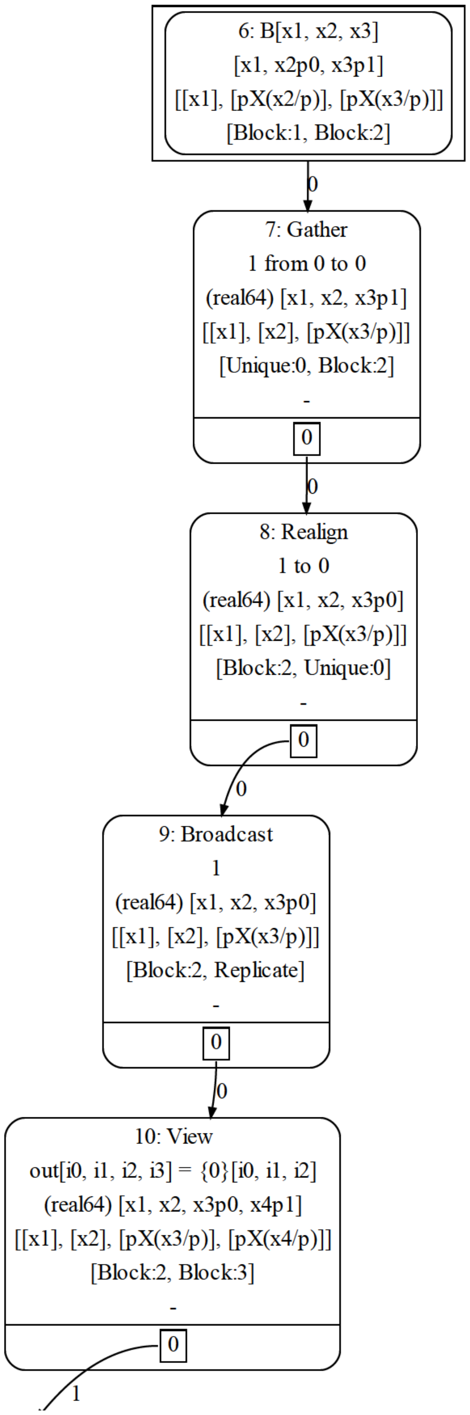


(j)

Figure 4.7: Rearrange node generation (cont.)

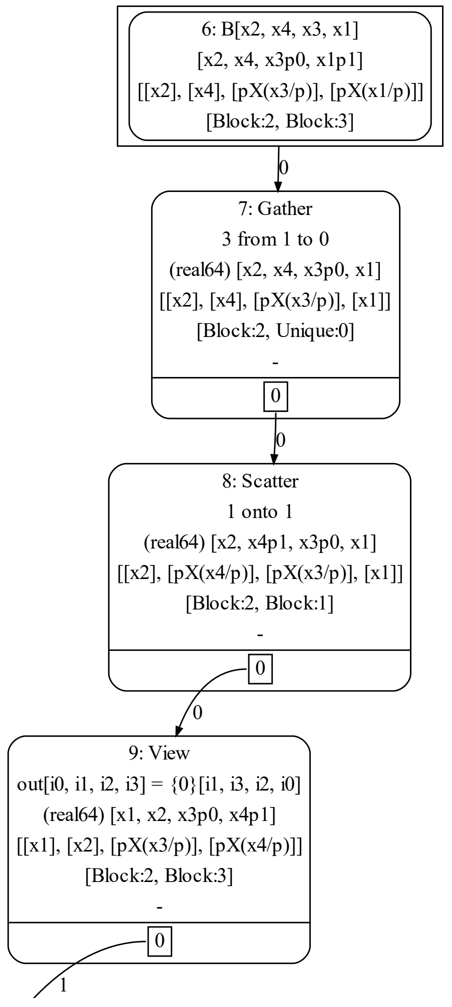


(k)



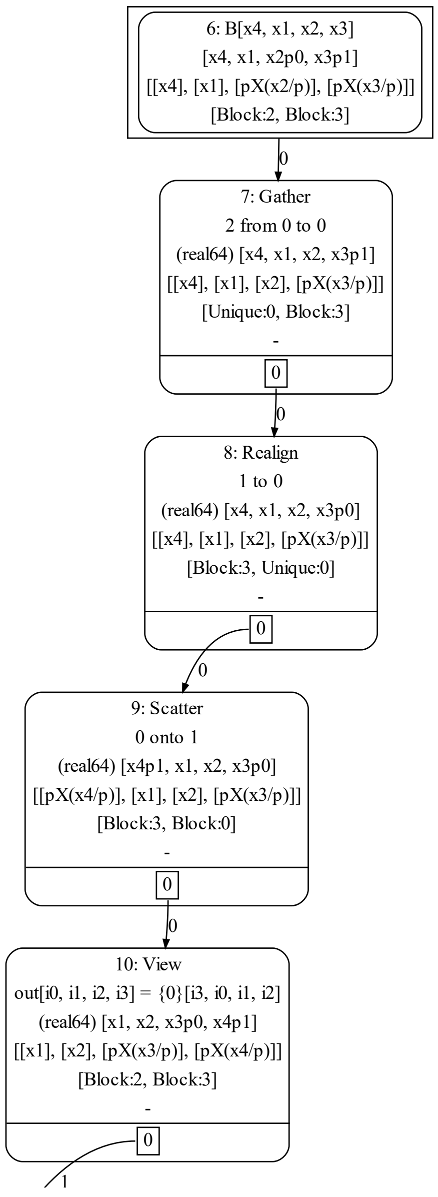
(l)

Figure 4.7: Rearrange node generation (cont.)



6: B[x2, x4, x3, x1]
[x2, x4, x3p0, x1p1]
[[x2], [x4], [pX(x3/p)], [pX(x1/p)]]
[Block:2, Block:3]
0
7: Gather
3 from 1 to 0
(real64) [x2, x4, x3p0, x1]
[[x2], [x4], [pX(x3/p)], [x1]]
[Block:2, Unique:0]
-
0
0
8: Scatter
1 onto 1
(real64) [x2, x4p1, x3p0, x1]
[[x2], [pX(x4/p)], [pX(x3/p)], [x1]]
[Block:2, Block:1]
-
0
0
9: View
out[i0, i1, i2, i3] = {0}[i1, i3, i2, i0]
(real64) [x1, x2, x3p0, x4p1]
[[x1], [x2], [pX(x3/p)], [pX(x4/p)]]
[Block:2, Block:3]
-
0
1

(m)



6: B[x4, x1, x2, x3]
[x4, x1, x2p0, x3p1]
[[x4], [x1], [pX(x2/p)], [pX(x3/p)]]
[Block:2, Block:3]
0
7: Gather
2 from 0 to 0
(real64) [x4, x1, x2, x3p1]
[[x4], [x1], [x2], [pX(x3/p)]]
[Unique:0, Block:3]
-
0
0
8: Realign
1 to 0
(real64) [x4, x1, x2, x3p0]
[[x4], [x1], [x2], [pX(x3/p)]]
[Block:3, Unique:0]
-
0
0
9: Scatter
0 onto 1
(real64) [x4p1, x1, x2, x3p0]
[[pX(x4/p)], [x1], [x2], [pX(x3/p)]]
[Block:3, Block:0]
-
0
0
10: View
out[i0, i1, i2, i3] = {0}[i3, i0, i1, i2]
(real64) [x1, x2, x3p0, x4p1]
[[x1], [x2], [pX(x3/p)], [pX(x4/p)]]
[Block:2, Block:3]
-
0
1

(n)

Figure 4.7: Rearrange node generation (cont.)

element-by-element addition of A with an array B for the following cases.

(a) B is a scalar uniquely distributed on both the dimensions. Then, as in Figure 4.7a, B is broadcast along dimensions 0 and 1, and a view is created to map the 4 dimensional index to no index.

(b) B is a vector of size $x4$ and is distributed along grid dimension 1. In this case, as in Figure 4.7b, B is broadcast along dimension 0 to replicate, and a view node is created to map the 4 dimensional index $[i0, i1, i2, i3]$ to the one dimensional index $[i3]$.

(c) B is a vector of size $x4$ and is uniquely distributed on both dimensions. In this case, as in Figure 4.7c, since the original array dimension is not mapped to the grid, and the final dimension is mapped to grid dimension 1, a scatter node is generated to map the array dimension 0 onto the grid dimension 1. Then the resultant array is replicated in dimension 0, with a similar view as before.

(d) B is a vector of size $x1$ distributed along grid dimension 1. In this case, the addition cannot be performed as is, as the two arrays are incompatible. We use `reshape(B, x1, 1, 1, 1)` to make the arrays compatible. Then, as in Figure 4.7d, the vector that is distributed along the dimension 1, is gathered onto all nodes (all gather) to replicate it along grid dimension 1, and then broadcast in grid dimension 0 to align with the matrix A.

(e) B is a vector of size $x3$ distributed along grid dimension 1. In this case, again, we need to reshape it to size $[x3, 1]$ to make it compatible to the array A. Then, as in Figure 4.7e, The array, which is distributed along dimension 1 is realigned onto dimension 0, and replicated along dimension 1 to match matrix A.

(f) B is a matrix of size $[x3, x4]$ distributed along grid dimensions 0 and 1. This aligns perfectly with A, and only a view node is generated, as in Figure 4.7f.

(g) B is a matrix of size $[x1, x4]$ distributed along grid dimensions 0 and 1. This needs a reshape to change the size to $[x1, 1, 1, x4]$. Here, as in Figure 4.7g, index 0 is gathered from dimension 0 to all processes on dimension 0, and a view node is created to correctly index the array.

(h) B is a matrix of size $[x4, x3]$ distributed along grid dimension 0 and 1, respectively. This needs a transpose to align with A. This is shown in Figure 4.7h.

(i) B is a matrix of size $[x3, x2]$ distributed along grid dimensions 0 and 1. Here, we transpose B and reshape the result to $[x2, x3, 1]$. This transpose is to ensure that the dimensions of $x2$ and $x3$ align correctly with A, instead of an explicit reshape. Here, $x2$ is gathered from grid dimension 1 to all processes and a view is created to incorporate the transpose. It must be noted that the creation of the communication nodes (a gather) are dependent on the complexity of the actual transformation, instead of the specified operations (a transpose and reshape), as in Figure 4.7i.

(j) B is a matrix of size $[x1, x2]$ distributed along grid dimensions 0 and 1. It is reshaped to size $[x1, x2, 1, 1]$ to align with A. The entire matrix must be present on all processes, and *Vaani* does this by performing two successive all gathers in both the dimensions 0 and 1, as in Figure 4.7j.

(k) B is a matrix of size $[x2, x3]$ distributed along grid dimensions 0 and 1. We, again, reshape B to size $[x2, x3, 1]$ to match A. Here, $x2$ is gathered in grid dimension 0 uniquely to process 0, then the matrix is realigned from grid dimension 1 to grid dimension 0 (to match $x3$), and then the matrix is broadcast in grid dimension 1 to make it available for the addition, as in Figure 4.7k. Here, even though the specification only has a reshape, the distribution of $x3$ on a dimension different than the matrix A, calls for a more complex transformation.

(l) B is a three dimensional tensor of size $[x1, x2, x3]$ such that $x2$ and $x3$ are distributed along grid dimensions 0 and 1. It is then reshaped to size $[x1, x2, x3, 1]$. The LLIR nodes for this transformation is given in Figure 4.7l, where $x2$ is gathered from grid dimension 0 onto the first column, then matrix is realigned to be present on the first row instead, and then broadcast down the column to align with matrix A.

(m) B is a 4 dimensional tensor of size $[x2, x4, x3, x1]$. We use `reorder(B, 3, 0, 2, 1)` to align with A. Here, $x1$ is gathered from grid dimension 1, and then $x4$ is scattered on the same dimension 1, as in Figure 4.7m.

(n) B is a 4 dimensional tensor of size $[x4, x1, x2, x3]$ and we use `reorder(B, 1, 2, 3, 0)` to order the tensor to size $[x1, x2, x3, x4]$ aligning with A. Here, $x2$ is gathered from grid dimension 0, then the array is realigned to distribute $x3$ along dimension 0 instead of 1, and finally, $x4$ is scattered along the grid dimension 1. This is presented in Figure 4.7n.

As a complete program example, Figure 4.8 is the LLIR obtained from the MLIR for GEMV in Figure 4.6. The two rearrange nodes for *alpha* and *beta* need not perform any

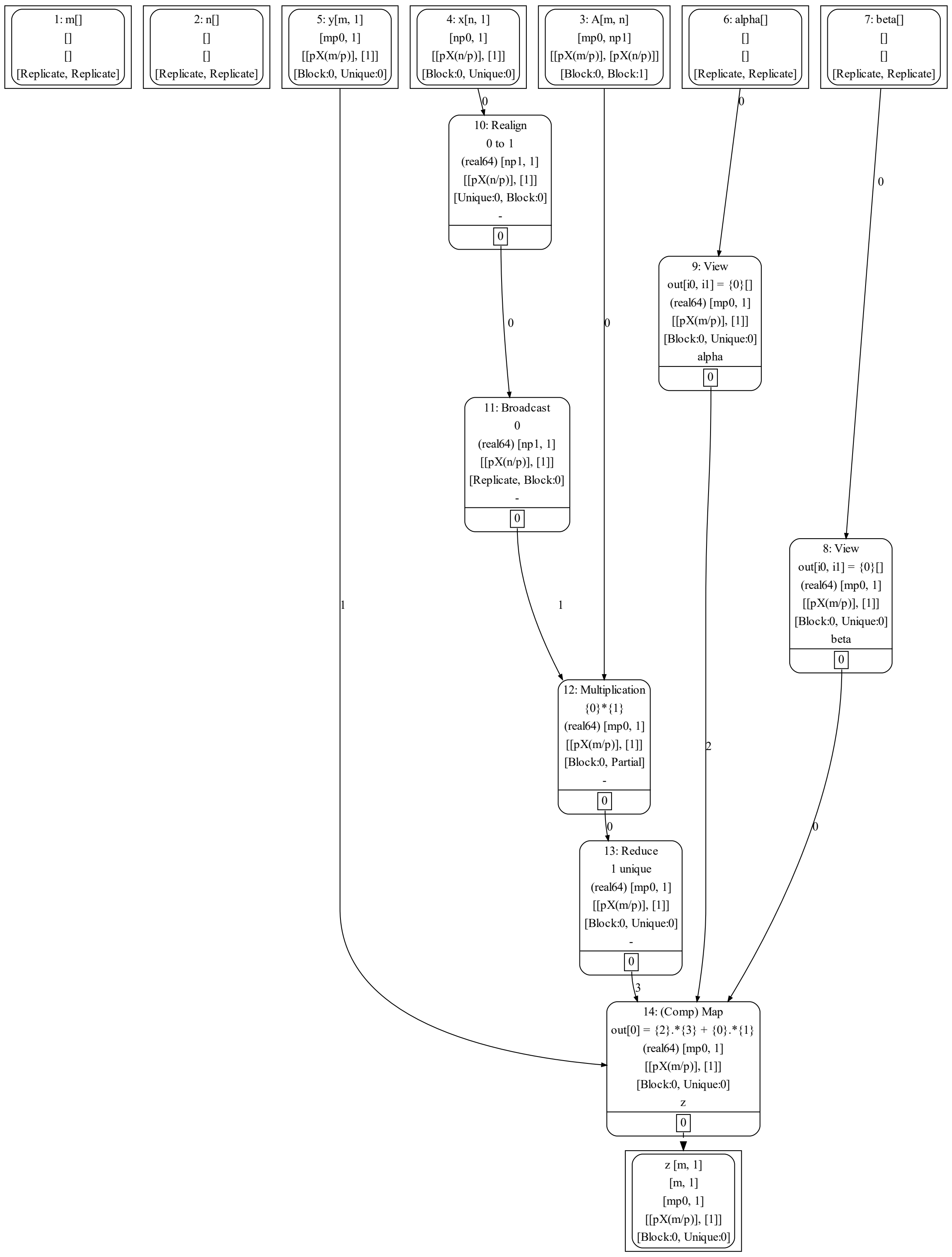


Figure 4.8: GEMV LLIR

```
tag(6,                                         # Node number in LLIR
    ['x', 'y'],                                # Iteration indices
    [('block', 'x', ['x0', 'x1'], 'b'),        # Split x into blocks
     ('block', 'y', ['y0', 'y1'], 'b'),        # Split y into blocks
     ('parallel', 'x0'),                       # Parallelize x0
     ('vectorize', 'x1', 4),                   # Vectorize x1 by 4
     ('unroll', 'y1', 2),                      # Unroll y1 by 2
     ('interchange', 'x1', 'y0')])             # Interchange x1 and y0
```

(a) Tag specification

```
# pragma omp parallel for private(y0, x1, y1)
for(x0 = 0; x0 < m; x0 += b) {
    for (y0 = 0; y0 < n; y0 += b) {
        # pragma omp simd simdlen(4)
        for(x1 = x0; x1 < min(x0+b, m); x1++) {
            for(y1 = y0; y1 < min(y0+b, n); y1 += 2) {
                // Code for (x1, y1), and (x1, (y1 + 1))
            }
        }
    }
}
```

(b) Loop nest for $m \times n$ iteration space

Figure 4.9: Sample LLIR computation tag

communication or computation, and they only generate a `view` node to indicate that all indices are ignored and a scalar value is returned. The rearrange node of the vector x generates a `realign` node to convert it from being distributed on the first column on the process grid to distributed along the first row. Then, the vector is broadcast down the column along the column communicator to replicate the vector on all the rows. The multiplication node is split into two nodes, a local computation multiplication node and a reduce node. The map node generates a local computation map node without any other nodes necessary.

4.10 LLIR NODE TAGGING

Vaani provides for tagging nodes in LLIR. Some of the features are described in the following.

4.10.1 Computation Nodes

Computation nodes in LLIR can be tagged to indicate thread parallelism, tiling, vectorization, loop unroll and loop reorder. Figure 4.9 shows a sample tag in LLIR and its corresponding loop nest generated. The code shown is simplified for clarity, and assumes both n and b are multiples of the unroll factor 2.

4.10.2 Communication Nodes

Vaani does not yet support tagging of communication nodes, but possible directions are to use varying styles of communication patterns available in MPI like selecting blocking or non-blocking communication. Halo exchange, for example, can be performed using non-blocking send/recv, persistent communication if it appears in a recurrence, one sided communication, synchronized exchange in each dimension (that takes care of diagonal elements if needed), etc.

4.11 CODE ORDER

An order is determined for the nodes in LLIR, in the order they will appear in the final source code. This order is selected so that all dependencies are satisfied in the LLIR, starting from the outputs and walking backwards to the inputs, such that a node is only placed in the order after its parents are placed. This is obtained by performing a topological sort of the nodes in LLIR.

4.12 BUFFER ALLOCATION

Once a schedule is determined, and the dependencies marked, buffers are allocated for each of the arrays. Notably, inputs and outputs are allocated their respective buffers. Each internal node retains the name assigned to it, if it has a name in the original program, else a *Vaani* generated temporary name is assigned.

4.12.1 Recurrences

Buffers in recurrences depend on the type of the recurrence variable. If all intermediate iterations of the variable are saved, an output buffer for the combined array is allocated, and each iteration of the recurrence accesses the corresponding section of the array. If only the last

```
memcpy(&vn_B[0][0][0], &A[0][0], sizeof(double)*mp0*nq1);
for(int32_t rc0 = 1; rc0 < iter + 1; rc0++) { // Recurrence
    for(int32_t i = 0; i < mp0; i++) { // Inside map loop
        for(int32_t j = 0; j < nq1; j++) {
            vn_B[rc0%2][i][j] = vn_B[(1 + rc0)%2][i][j] + A[i][j];
        }
    }
}
memcpy(&B[0][0], &vn_B[iter%2][0][0], sizeof(double)*mp0*nq1);
```

(a) `B = B{-1} + A`, saving only the last iteration `B[iter]`

```
memcpy(&vn_B[1][0][0], &A[0][0], sizeof(double)*mp0*nq1);
memcpy(&vn_B[0][0][0], &A[0][0], sizeof(double)*mp0*nq1);
for(int32_t rc0 = 2; rc0 < iter + 1; rc0++) {
    for(int32_t i = 0; i < mp0; i++) {
        for(int32_t j = 0; j < nq1; j++) {
            vn_B[rc0%3][i][j] = vn_B[(2 + rc0)%3][i][j] +
                                vn_B[(1 + rc0)%3][i][j];
        }
    }
}
memcpy(&B[0][0], &vn_B[iter%3][0][0], sizeof(double)*mp0*nq1);
```

(b) `B = B{-1} + B{-2}`, saving only the last iteration `B[iter]`

```
memcpy(&B[0][0][0], &A[0][0], sizeof(double)*mp0*nq1);
memcpy(&B[1][0][0], &A[0][0], sizeof(double)*mp0*nq1);
for(int32_t rc0 = 2; rc0 < iter; rc0++) {
    for(int32_t i = 0; i < mp0; i++) {
        for(int32_t j = 0; j < nq1; j++) {
            B[rc0][i][j] = B[-1 + rc0][i][j] + B[-2 + rc0][i][j];
        }
    }
}
```

(c) `B = B{-1} + B{-2}`, saving all iterations `B[0:iter]`

Figure 4.10: Buffer allocation and use in a recurrence for an array of size $m \times n$ with initialization from array `A`

iteration is needed, then the total number of iteration values required for the recurrence are saved. For example, in the Jacobi2D example from 2.1e, since it uses the previous iteration (-1) and writes to the current iteration (0), only 2 copies of the array suffice, and they are indexed in the code accordingly. If a fibonacci-like operation was performed, it would require 3 iterations and thus, 3 copies of the buffer would be required. Figure 4.10 shows code snippets for these examples.

4.12.2 Ghost Regions

Buffers with extended ghost regions are allocated for nodes requiring ghost regions, and all indices are remapped to point to the correct locations. For example, an array `A` of local size $m \times n$ with 1 layer ghost boundary on all sides would have an index shift of $(1, 1)$, thus `A[i][j]` would be addressed as `A[i+1][j+1]`, where as a more complex ghost layer with two to the left and one on top would index as `A[i+2][j+1]`. The bottom and right ghost layers would change the stride length, but not the index values.

4.13 CODE GENERATION

Each node in LLIR is transformed into CLIR by explicitly defining iteration spaces and C instruction representations. These representations are then generated as the final C code, in the order determined by the schedule.

4.13.1 Boilerplate

During transformation from LLIR to CLIR, a list of headers, the function name (`main` for a program) and arguments to the function (`argc` and `argv` for a program) are determined. Commandline arguments, file inputs and file outputs are marked for code generation. These are used to create the basic structure of the final code.

4.13.2 Declarations

All variables encountered in the translation from LLIR to CLIR are tracked by type and declared in the program. Array sizes depend on the grid size, and thus they are not allocated with declarations.

```
/*** Grid Setup Begin ***/
MPI_Init(&argc, &argv);
MPI_Comm_dup(MPI_COMM_WORLD, &vaani_comm);
MPI_Comm_size(vaani_comm, &p);
MPI_Comm_rank(vaani_comm, &rank);
/*** Grid Setup End ***/
```

Figure 4.11: 1-dimensional grid creation

4.13.3 Grid Creation

Grid dimensions, ranks and communicator handles are created during the grid creation phase (Section 4.5). *Vaani* uses these handles to create MPI grids. Figure 4.11 shows the generated code for a 1 dimensional grid, while Figure 4.12 shows the grid creation for both rectangular and square grids. Figure 4.13 shows some 3d grid creations.

4.13.4 Array Allocations

After the grid has been created, command line arguments are parsed to obtain scalar inputs to the program like array sizes. These, combined with the grid dimensions and ranks, are used to generate local array sizes. Once local array sizes are determined, arrays are allocated. *Vaani* supports two ways of allocating and using the arrays:

- Multi-dimensional arrays: These have a space overhead to support the multidimensional indexing feature, but allow for readable indexing schemes. Figure 4.14a shows allocation and usage of a 3 dimensional array.
- Flat arrays: These have flat arrays, and need complex indexing schemes. Figure 4.14b shows allocation and usage of a flattened 3 dimensional array.

A way to define array indexing using compile time definitions (`#define`) is being explored to support simpler indexing without the memory overhead.

4.13.5 Computation

The LLIR computation tag, if specified, or a default tag is used to generate the loop nests for each computation node. Each output in the node is assigned an expression that maps it to its input. *Vaani* currently does not perform optimizations in expression generation, and

```
/*** Grid Setup Begin ***/
MPI_Init(&argc, &argv);
MPI_Comm_dup(MPI_COMM_WORLD, &vaani_comm);
MPI_Comm_size(vaani_comm, &nprocs);
MPI_Comm_rank(vaani_comm, &rank);
MPI_Dims_create(nprocs, 2, dimsizes);
MPI_Cart_create(vaani_comm, 2, dimsizes, periodic, 0, &comm2d);
MPI_Cart_coords(comm2d, rank, 2, coords);
p = dimsizes[0];
q = dimsizes[1];
rowrank = coords[0];
colrank = coords[1];
MPI_Comm_split(comm2d, colrank, rowrank, &colcomm);
MPI_Comm_split(comm2d, rowrank, colrank, &rowcomm);
/*** Grid Setup End ***/
```

(a) Rectangular grid ($p \times q$)

```
/*** Grid Setup Begin ***/
...
assert(dimsizes[0] == dimsizes[1]);
p = dimsizes[0];
rowrank = coords[0];
colrank = coords[1];
...
/*** Grid Setup End ***/
```

(b) Square grid ($p \times p$)

Figure 4.12: 2-dimensional grid creation

```
/*** Grid Setup Begin ***/
MPI_Init(&argc, &argv);
MPI_Comm_dup(MPI_COMM_WORLD, &vaani_comm);
MPI_Comm_size(vaani_comm, &nprocs);
MPI_Comm_rank(vaani_comm, &rank);
MPI_Dims_create(nprocs, 3, dimsizes);
MPI_Cart_create(vaani_comm, 3, dimsizes, periodic, 0, &comm3d);
MPI_Cart_coords(comm3d, rank, 3, coords);
p = dimsizes[0];
q = dimsizes[1];
r = dimsizes[2];
gridrank0 = coords[0];
gridrank1 = coords[1];
gridrank2 = coords[2];
MPI_Comm_split(comm3d, gridrank2 + r*gridrank1, gridrank0, &dim0comm);
MPI_Comm_split(comm3d, gridrank2 + r*gridrank0, gridrank1, &dim1comm);
MPI_Comm_split(comm3d, gridrank1 + q*gridrank0, gridrank2, &dim2comm);
/*** Grid Setup End ***/
```

(a) grid of size $p \times q \times r$

```
/*** Grid Setup Begin ***/
...
assert(dimsizes[0] == dimsizes[1]);
assert(dimsizes[1] == dimsizes[2]);
p = dimsizes[0];
...
/*** Grid Setup End ***/
```

(b) grid of size $p \times p \times p$

```
/*** Grid Setup Begin ***/
...
assert(dimsizes[0] == dimsizes[2]);
p = dimsizes[0];
q = dimsizes[1];
...
/*** Grid Setup End ***/
```

(c) grid of size $p \times q \times p$

Figure 4.13: 3-dimensional grid creation

```c
/* Allocation */
A = malloc(sizeof(double**)*t);
A[0] = malloc(sizeof(double*)*t*m);
A[0][0] = malloc(sizeof(double)*t*m*n);
for(int32_t i = 0; i < t; i++) {
    A[i] = A[0] + i*m;
    for(int32_t j = 0; j < m; j++) {
        A[i][j] = A[0][0] + i*m*n + j*n;
    }
}
/* Usage */
for(int32_t i = 0; i < t; i++) {
    for(int32_t j = 0; j < m; j++) {
        for(int32_t k = 0; k < n; k++) {
            ... A[i][j][k] ...
        }
    }
}
```

(a) Multidimensional array `A` of local size $t \times m \times n$

```c
/* Allocation */
A = malloc(sizeof(double)*t*m*n);
/* Usage */
for(int32_t i = 0; i < t; i++) {
    for(int32_t j = 0; j < m; j++) {
        for(int32_t k = 0; k < n; k++) {
            ... A[k + n*j + i*m*n] ...
        }
    }
}
```

(b) Flattened array `A` of size $t \times m \times n$

Figure 4.14: Array allocation and indexing

leaves it to the underlying C compiler. Code for expressions are generated in a recursive manner. Argument strings are first generated, and then they are combined to create the final expression. Arguments are parenthesized if needed following standard C operator precedence.

4.13.6 Communication

Vaani currently uses blocking collective operations and non-blocking send and receive operations. Figure 4.15 shows sample code generated to broadcast, reduce to a unique index, all reduce and realign vectors on a two dimensional square process grid. Here, it can be observed that if *Vaani* allocates the same buffer for the reduction during buffer allocation, the generated code uses `MPI_IN_PLACE` to reuse the buffer. However, a new temporary buffer is allocated for the realignment operation, as the size and distribution of the data would be different, and hence a new buffer is preferable.

```
MPI_Bcast(&v[0], np1, MPI_DOUBLE, 0, colcomm);
```

(a) Broadcast vector v along the column communicator

```
if(rowrank == 0) {
    MPI_Reduce(MPI_IN_PLACE, &y[0], np1, MPI_DOUBLE, MPI_SUM,
               0, colcomm);
}
else {
    MPI_Reduce(&y[0], &y[0], np1, MPI_DOUBLE, MPI_SUM, 0, colcomm);
}
```

(b) Reduce vector y along the column communicator to rank 0

```
MPI_Allreduce(MPI_IN_PLACE, &v[0], mp0, MPI_DOUBLE, MPI_SUM, rowcomm);
```

(c) All reduce vector v in place along the row communicator

```
rqid = 0;
if(rowrank == 0) {
    MPI_Irecv(&temp1[0], np1, MPI_DOUBLE, colrank*p, 0, vaani_comm,
              &req[rqid++]);
}
if(colrank == 0) {
    MPI_Isend(&x[0], np0, MPI_DOUBLE, rowrank, 0, vaani_comm,
             &req[rqid++]);
}
if(rqid>0) MPI_Waitall(rqid, req, MPI_STATUSES_IGNORE);
```

(d) Realign vector x from row distributed to column distributed as vector $temp1$

Figure 4.15: Some example communication codes

CHAPTER 5: USING VAANI

In this chapter, we take a basic matrix-vector multiplication operation $z = \alpha Ax + \beta y$ and walk through the process of using *Vaani* to obtain final C code.

5.1 SPECIFICATION

The first step is to write a specification in *Vaani*. Figure 5.1 shows the program specification for matrix-vector multiplication. We use the file extension '.vn' to denote *Vaani* files by convention, and save the specification as 'gemv.vn'. The parser, however, can take any text file and try to parse it as a *Vaani* specification.

5.2 COMPILATION SCRIPT

A python script can be used to compile and generate a C code file from a specification. Figure 5.2 shows a sample compilation script. The interactive module of *Vaani* has functions to interface into the *Vaani* compiler. This module is first imported as `it`. The bare minimum steps to generate C code are the function calls, `parse` to parse the input file and generate HLIR, `hlirtomlir` to convert HLIR to MLIR, `grid` to specify a grid for the computation, `mlirtollir` to convert to LLIR, `llirtoclir` to convert to CLIR, `clirtocode` to internally generate the code, and `generate` to generate the final C program. The function `plot` plots the current intermediate representation to a file. This process uses `pygraphviz` module, and typically supports most common formats like '.jpg', '.png', '.ps', '.pdf', etc, detected by the extension provided in the filename. We use the '.pdf' format in the example.

```
program GEMV
in m, n scalar(int64)
in A matrix(m, n, real64)
in x cvector(n, real64)
in y cvector(m, real64)
in alpha, beta scalar(real64)
out z

z = alpha * A * x + beta * y
```

Figure 5.1: Matrix-vector multiplication specification in *Vaani*

```python
import vaani.interactive as it

name = 'gemv'
it.parse(name + '.vn')
it.plot(name + '-hlir.pdf')
it.hlirtomlir()
it.plot(name + '-mlir.pdf')
it.grid(2, ['p', 'p'])
it.mlirtollir()
it.plot(name + '-llir.pdf')
it.llirtoclir()
it.clirtocode()
it.generate(name + '.c')
```

Figure 5.2: Python script to invoke *Vaani*

5.3 EXECUTION

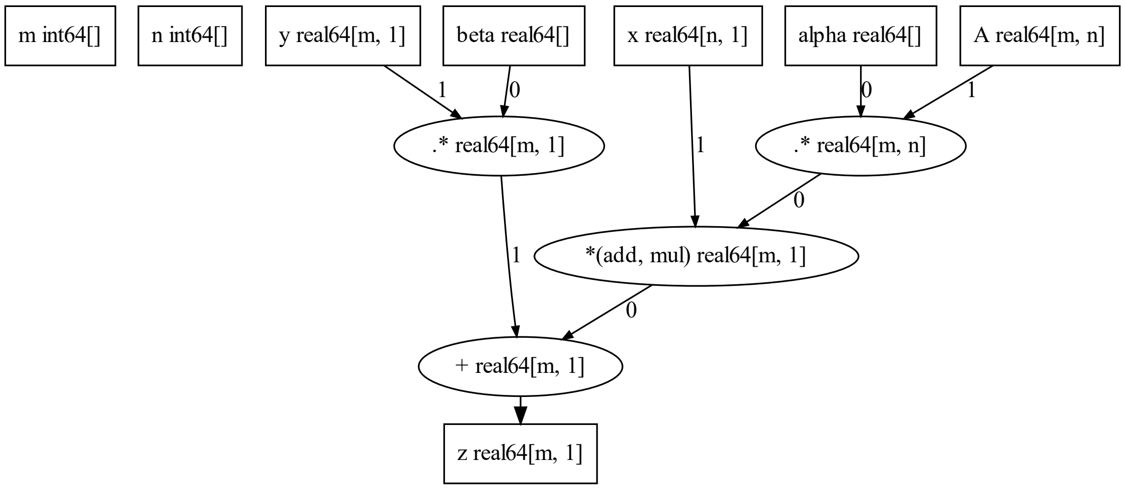
Running the basic script from Figure 5.2 as `python gemv-script.py` generates three IR Figures and a C code file. Figure 5.3 shows the three intermediate representations generated by the program.

HLIR shows each operation as a node, and has two scalar multiplications, one matrix-vector multiplication and one addition. MLIR combines the nodes to generate a multiplication node for the matrix-vector multiplication, and a map node for the two scalar multiplications and the addition. LLIR shows that the column vector x must be transposed (realign node performs vector transpose) to align to the first row of processes, then broadcast down the columns. Then the matrix-vector multiplication is performed locally on all the nodes, and the result is marked as partially distributed in the column dimension. Then the results are reduced to a single column, and finally the map is performed to generate the final result.

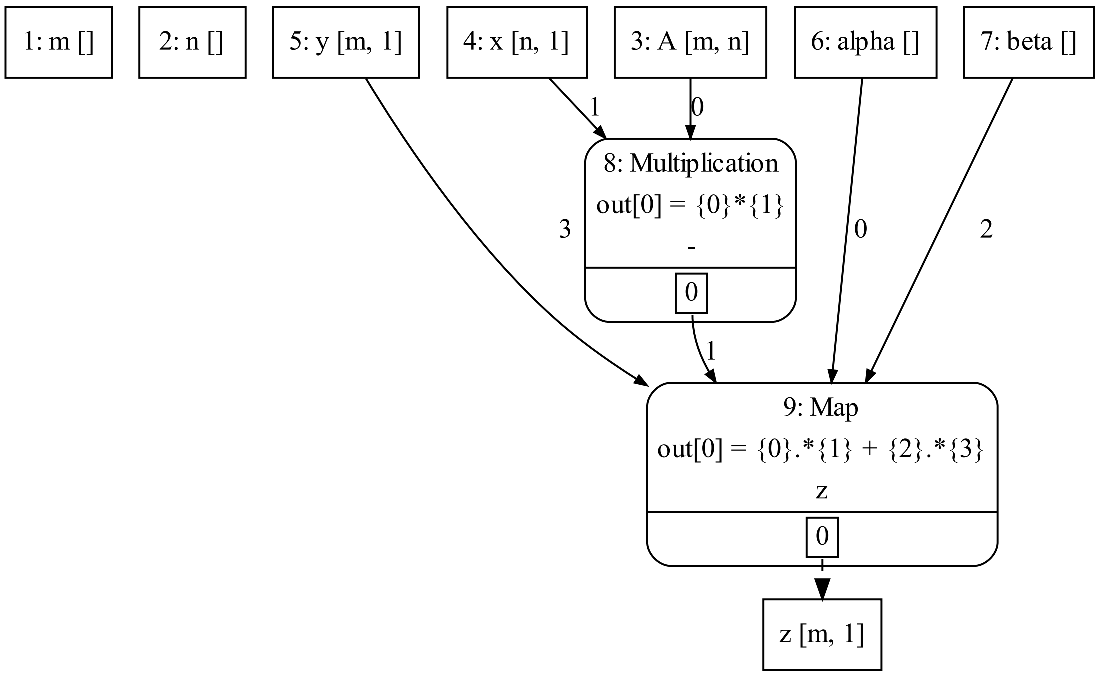
Figure 5.4 shows the complete code generated by *Vaani*. Note that the input arrays are initialized with fixed values, the actual computation is repeated *reps* times and the computation is timed, to ease evaluation of the generated code.

5.4 ALTERNATE VERSIONS

Alternate versions of the same computation can be generated by altering the compilation script. Figure 5.5 shows the modified script for a 1D grid distribution and its corresponding default partitioning in LLIR. Here, the matrix A is column distributed onto the process grid,

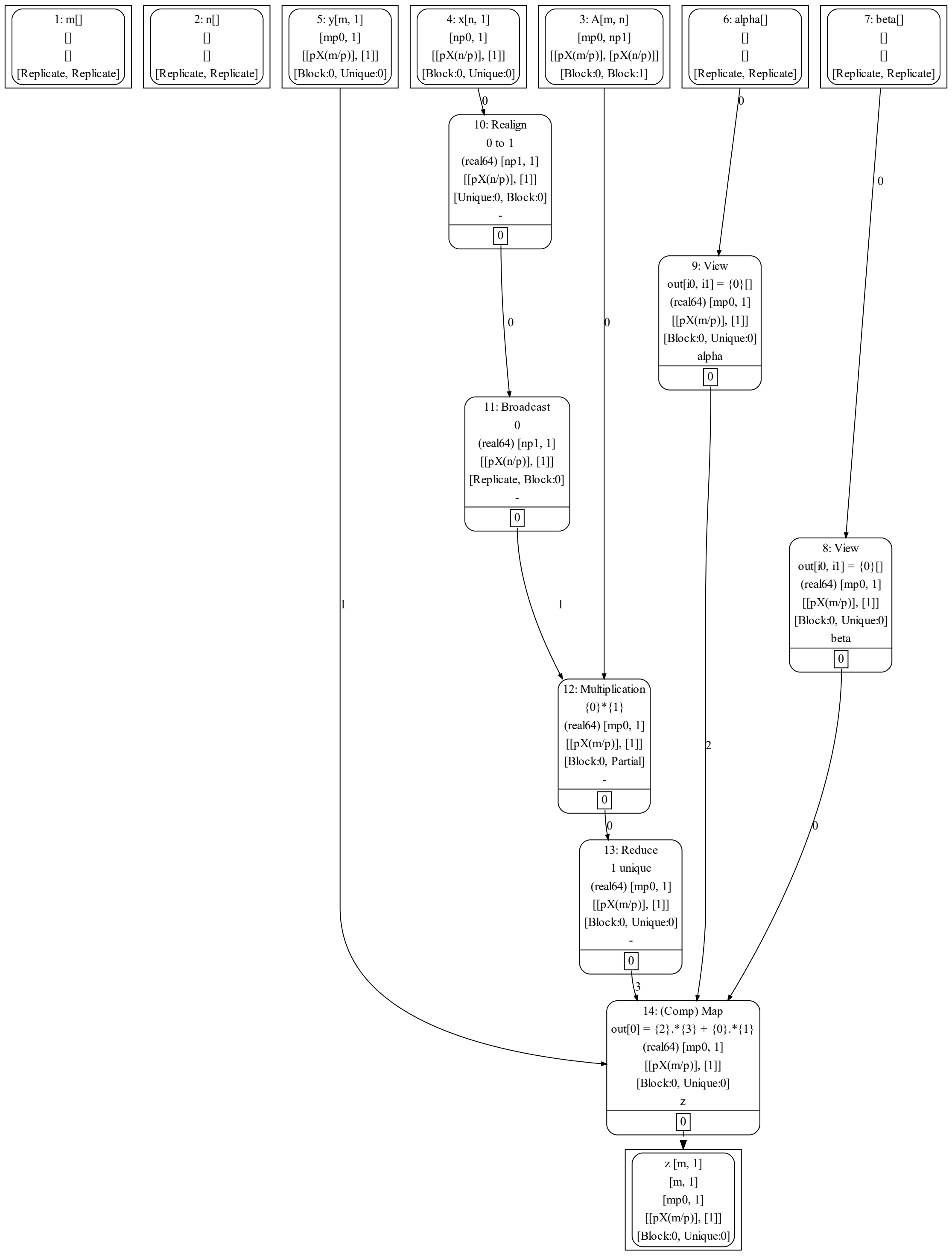


(a) HLIR



(b) MLIR

Figure 5.3: Intermediate representations of GEMV



(c) LLIR

Figure 5.3: Intermediate representations of GEMV (cont.)

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <mpi.h>
#include <math.h>
#include <assert.h>

int32_t main(int32_t argc, char** argv) {
    /*** Declarations Begin ***/
    int64_t m, n, mp0, np1, np0;
    double alpha, beta;
    int32_t nprocs, rank, p, rowrank, colrank;
    MPI_Comm vaani_comm, comm2d, colcomm, rowcomm;
    int32_t dimsizes[2], periodic[2], coords[2];
    double **A, *x, *y, *z, *temp0, *temp1;
    MPI_Request req[2];
    int32_t rqid;
    double ts, te;
    int reps;
    /*** Declarations End ***/

    /*** Grid Setup Begin ***/
    MPI_Init(&argc, &argv);
    MPI_Comm_dup(MPI_COMM_WORLD, &vaani_comm);
    MPI_Comm_size(vaani_comm, &nprocs);
    MPI_Comm_rank(vaani_comm, &rank);
    dimsizes[0] = 0;
    dimsizes[1] = 0;
    periodic[0] = 0;
    periodic[1] = 0;
    MPI_Dims_create(nprocs, 2, dimsizes);
    MPI_Cart_create(vaani_comm, 2, dimsizes, periodic, 0, &comm2d);
    MPI_Cart_coords(comm2d, rank, 2, coords);
    assert(dimsizes[0] == dimsizes[1]);
    p = dimsizes[0];
    rowrank = coords[0];
    colrank = coords[1];
    MPI_Comm_split(comm2d, colrank, rowrank, &colcomm);
    MPI_Comm_split(comm2d, rowrank, colrank, &rowcomm);
    /*** Grid Setup End ***/
```

Figure 5.4: GEMV *Vaani* generated code

```
/*** Command Line Inputs Begin ***/
m = strtol(argv[1], NULL, 10);
n = strtol(argv[2], NULL, 10);
alpha = strtod(argv[3], NULL);
beta = strtod(argv[4], NULL);
reps = atoi(argv[5]);
/*** Command Line Inputs End ***/

/*** Local Size Computation Begin ***/
mp0 = m/p + ((rowrank<m%p)?1:0);
int64_t mp0_start = mp0*rowrank + ((rowrank<(m%p))?0:(m%p));
np1 = n/p + ((colrank<n%p)?1:0);
int64_t np1_start = np1*colrank + ((colrank<(n%p))?0:(n%p));
np0 = n/p + ((rowrank<n%p)?1:0);
int64_t np0_start = np0*rowrank + ((rowrank<(n%p))?0:(n%p));
/*** Local Size Computation End ***/

/*** Allocate Arrays Begin ***/
A = malloc(sizeof(double*)*mp0);
A[0] = malloc(sizeof(double)*mp0*np1);
for(int32_t i = 0; i < mp0; i++) {
    A[i] = A[0] + i*np1;
}
x = malloc(sizeof(double)*np0);
y = malloc(sizeof(double)*mp0);
z = malloc(sizeof(double)*mp0);
temp0 = malloc(sizeof(double)*mp0);
temp1 = malloc(sizeof(double)*np1);
/*** Allocate Arrays End ***/

/*** Read Input Arrays Begin ***/
//read_2d_double(vaani_comm, A, m, n, mp0, np1, "A.txt");
for(int32_t i = 0; i < mp0; i++) {
    for(int32_t j = 0; j < np1; j++) {
        A[i][j] = (mp0_start + i + 1)*(np1_start + j + 1);
    }
}
```

Figure 5.4: GEMV *Vaani* generated code (cont.)

```
//read_2d_double(vaani_comm, x, n, 1, np0, 1, "x.txt");
if(colrank == 0) {
    for(int32_t i = 0; i < np0; i++) {
        x[i] = (np0_start + i + 1);
    }
}
//read_2d_double(vaani_comm, y, m, 1, mp0, 1, "y.txt");
if(colrank == 0) {
    for(int32_t i = 0; i < mp0; i++) {
        y[i] = (mp0_start + i + 1);
    }
}
/*** Read Input Arrays End ***/

ts = MPI_Wtime();
for(int32_t repid = 0; repid < reps; repid++){
    rqid = 0;
    if(rowrank == 0) {
        MPI_Irecv(&temp1[0], np1, MPI_DOUBLE, colrank*p, 0,
                  vaani_comm, &req[rqid++]);
    }
    if(colrank == 0) {
        MPI_Isend(&x[0], np0, MPI_DOUBLE, rowrank, 0,
                  vaani_comm, &req[rqid++]);
    }
    if(rqid>0) MPI_Waitall(rqid, req, MPI_STATUSES_IGNORE);

    MPI_Bcast(&temp1[0], np1, MPI_DOUBLE, 0, colcomm);

    for(int32_t i = 0; i < mp0; i++) {
        for(int32_t j = 0; j < np1; j++) {
            temp0[i] = temp0[i] + A[i][j]*temp1[j];
        }
    }
```

Figure 5.4: GEMV *Vaani* generated code (cont.)

```
        if(colrank == 0) {
            MPI_Reduce(MPI_IN_PLACE, &temp0[0], mp0, MPI_DOUBLE,
                       MPI_SUM, 0, rowcomm);
        }
        else {
            MPI_Reduce(&temp0[0], &temp0[0], mp0, MPI_DOUBLE,
                       MPI_SUM, 0, rowcomm);
        }
        if(colrank == 0) {
            for(int32_t i = 0; i < mp0; i++) {
                z[i] = alpha*temp0[i] + beta*y[i];
            }
        }

    }
    te = MPI_Wtime() - ts;
    MPI_Allreduce(MPI_IN_PLACE, &te, 1, MPI_DOUBLE,MPI_SUM, vaani_comm);
    if(rank == 0) printf("GEMV\tGEN\t%d\t%lf\n",nprocs,te/(reps*nprocs));
    /*** Write Output Arrays Begin ***/
    //write_2d_double(vaani_comm, z, m, 1, mp0, 1, "z.txt");
    /*** Write Output Arrays End ***/

    /*** Free Arrays Begin ***/
    free(A[0]);
    free(A);
    free(x);
    free(y);
    free(z);
    free(temp0);
    free(temp1);
    /*** Free Arrays End ***/

    /*** Grid Teardown Begin ***/
    MPI_Finalize();
    /*** Grid Teardown End ***/

    return 0;
}
```

Figure 5.4: GEMV *Vaani* generated code (cont.)

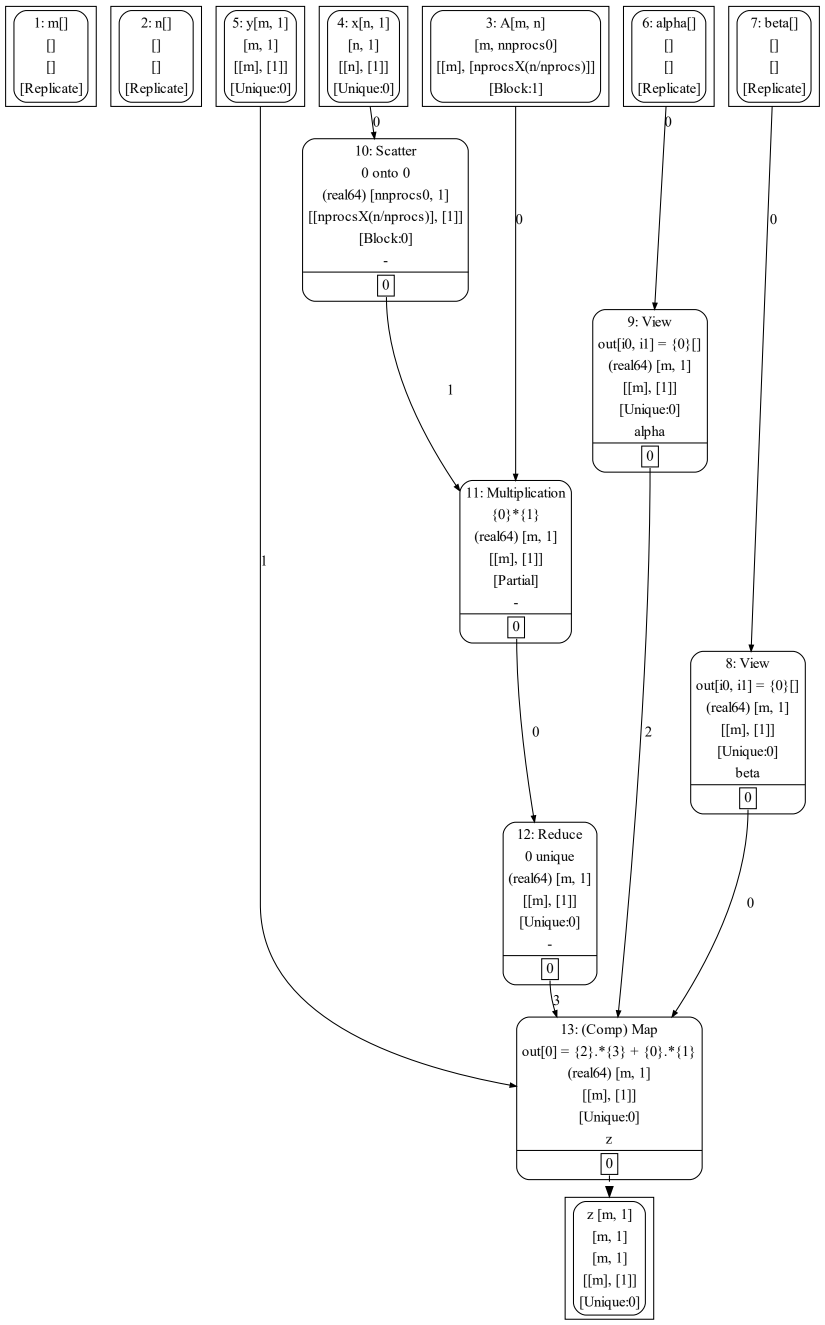
```python
import vaani.interactive as it

name = 'gemv'
it.parse(name + '.vn')
it.plot(name + '-hlir.pdf')
it.hlirtomlir()
it.plot(name + '-mlir.pdf')
it.grid(1)
it.mlirtollir()
it.plot(name + '-llir.pdf')
it.llirtoclir()
it.clirtocode()
it.generate(name + '.c')
```

(a) Python script to invoke *Vaani*

Figure 5.5: GEMV distributed on a 1D grid

and the vector x is present on the first process of the grid. To perform the multiplication, x is scattered along the grid, the multiplication is performed, and the result is generated by summing the individual multiplication results using reduction.

Now, a user could decide to row distribute the matrix, distribute the vectors, and modify the script to explicitly do so. This is shown in Figure 5.6. Here, the vector x is gathered to be present on all processes, then the multiplication is performed. The result of the multiplication is already distributed across the processes, and the scalar multiply and addition can be performed directly.

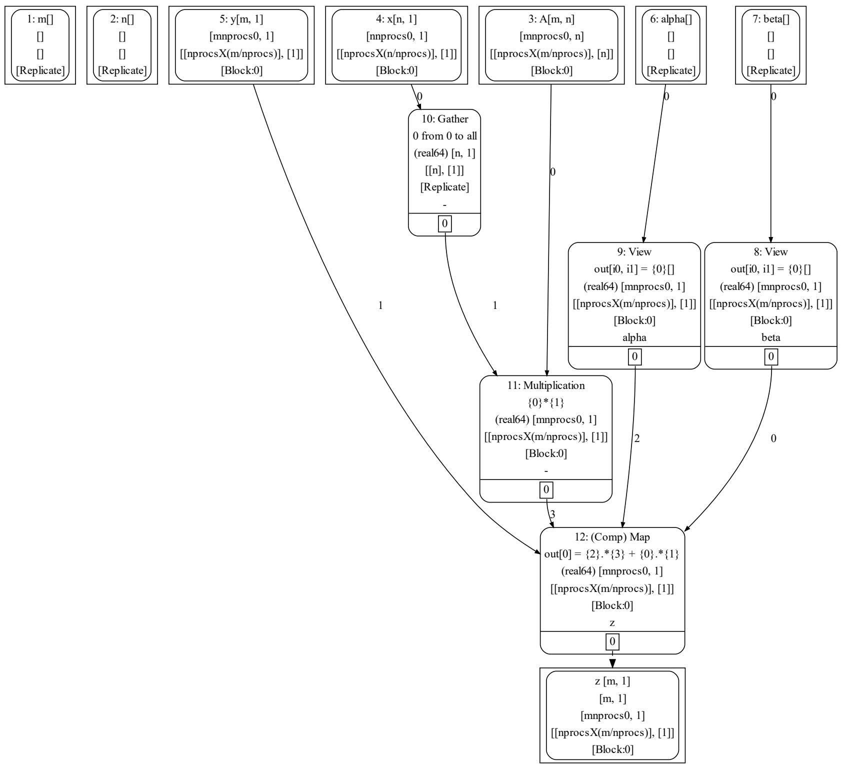


(b) LLIR

Figure 5.5: GEMV distributed on a 1D grid(cont.)

```python
import vaani.interactive as it

name = 'gemv'
it.parse(name + '.vn')
it.plot(name + '-hlir.pdf')
it.hlirtomlir()
it.plot(name + '-mlir.pdf')
it.grid(1)
it.gridmap('A', ('block', 0, 0))
it.gridmap('x', ('block', 0, 0))
it.gridmap('y', ('block', 0, 0))
it.gridmap('z', ('block', 0, 0))
it.mlirtollir()
it.plot(name + '-llir.pdf')
it.llirtoclir()
it.clirtocode()
it.generate(name + '.c')
```

(a) Python script to invoke *Vaani*



1: m[] [] [] [Replicate]
2: n[] [] [] [Replicate]
5: y[m, 1] [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0]
4: x[n, 1] [nnprocs0, 1] [[nprocsX(n/nprocs)], [1]] [Block:0]
3: A[m, n] [mnprocs0, n] [[nprocsX(m/nprocs)], [n]] [Block:0]
6: alpha[] [] [] [Replicate]
7: beta[] [] [] [Replicate]
0
10: Gather 0 from 0 to all (real64) [n, 1] [[n], [1]] [Replicate] - 0
0
0
0
9: View out[i0, i1] = {0}[] (real64) [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0] alpha 0
8: View out[i0, i1] = {0}[] (real64) [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0] beta 0
1
1
11: Multiplication {0}*{1} (real64) [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0] - 0
2
0
3
12: (Comp) Map out[0] = {2}.*{3} + {0}.*{1} (real64) [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0] z 0
z [m, 1] [m, 1] [mnprocs0, 1] [[nprocsX(m/nprocs)], [1]] [Block:0]

(b) LLIR

Figure 5.6: 1D grid

CHAPTER 6: EXPERIMENTAL FRAMEWORK

We evaluate the generated code on 8 nodes of a cluster, where each node has two Intel® Xeon® CPU E5-2670v2 processors with 10 cores each, operating at 2.50 GHz connected together via Infiniband. Each processor has 30 MB L3 cache and a node has 64 GB of main memory and runs CentOS 6.9 operating system. All code is compiled using Intel® compilers version 18.0.1 and Intel® MPI library with -O3 optimization flag.

CHAPTER 7: EVALUATION

In this chapter, we evaluate code generated by *Vaani* on the distributed memory cluster described in chapter 6. Three types of computations are considered for evaluation. First, Basic Linear Algebra Subroutines (BLAS) like computations, particularly level-2 operations are evaluated and compared to implementations using Intel® Math Kernel Library (MKL) ScaLAPACK library routines. Then, a 9-point star Jacobi stencil is evaluated and compared to Parallel Research Kernels (PRK) stencil implementation in MPI. Finally, power iteration computation using the novel recurrence construct with iteration count and termination condition is compared, again, to MKL routines.

Name	Operations
ATAX	$y = A'Ax$
BATAX	$y = \beta A'Ax$
BICGK	$q = Ap$ $s = A'r$
GEMV	$z = \alpha Ax + \beta y$
GEMVER	$A = A + u_1v_1' + u_2v_2'$ $x = \beta A'y + z$ $w = \alpha Ax$
GEMVT	$x = \beta A'y + z$ $w = \alpha * A * x$
GESUMMV	$y = \alpha Ax + \beta Bx$
HESSBLK	$A = A - u_1v_1' + u_2v_2'$ $v = A'x$ $w = Ax$
TRILAZY	$y = y - YU'u - UY'u$

Table 7.1: Example level 2-BLAS programs

7.1 BLAS-LIKE OPERATIONS

Table 7.1 shows some level-2 BLAS operations using matrix-vector multiplication and vector-vector outer products. Figure 7.1 shows the specification of these routines as programs in *Vaani*, except GEMVER which has been a running example throughout the thesis. We compare these operations with Intel® Math Kernel Library (MKL) ScaLAPACK library routines. Some operations have a single library call, while some operations are a series of calls to the library. Figure 7.2 shows the MKL function call sequences for these operations. It must be noted that the user must generate a complete program that creates a grid, allocates and initializes arrays and creates descriptions for MKL function calls before the actual computation sequence, and we show a sample of this in Figure 7.3. We repeat the iteration in the program 5 times and take the average execution time for one iteration for each of the applications. We run the applications 3 times and take the minimum execution time among multiple runs.

Figure 7.4 shows the execution time and speedup of *Vaani* generated code and MKL for matrix size of $50,000 \times 60,000$ (except for HESSBLK which requires a square matrix and we used $50,000 \times 60,000$. Speedup is computed with respect to the execution time of MKL. It can be seen that *Vaani* generates competitive code. Although MKL is faster for some of the test cases on a single process, *Vaani* outperforms MKL for higher number of processes and scales better than MKL for most of the test cases.

We also evaluate the same code for a small matrix of size 5000×6000, and the results are presented in Figure 7.5.

7.2 STENCIL COMPUTATIONS

Parallel Research Kernels (PRK) Stencil [5] is a benchmark that applies a radius-2 star stencil to a distributed 2D array. We use the MPI1 version from the suite and compare it against the code generated by *Vaani*. The original code is 337 lines of code. The *Vaani* program is in Figure 7.6. It must be noted that the *Vaani* code adds a constant to the matrix A, which is what PRK stencil code does to make sure that data needs to be sent on every iteration. The computation is repeated for 5 times and the average time per iteration is computed, for both PRK stencil and *Vaani* generated stencil. Such applications are run thrice, and the minimum time is reported. Figure 7.7 show the execution time and speedup compared to 1 process execution of PRK for a matrix dimension of 50,000. It can be observed that *Vaani* generated code performs as well as PRK.

7.3 ITERATIVE COMPUTATIONS

Power iteration is a simple eigenvalue algorithm that produces an eigenvector of a diagonalizable matrix. The recurrence relation for the computation is

$$v_{k+1} = \frac{Av_k}{||Av_k||} \tag{7.1}$$

Starting with a random vector v_0, the vector is multiplied by a matrix A and normalized in each iteration. The *Vaani* specification is given in Figure 7.8. The same iteration in

```
program ATAX
in A matrix(m, n, real64)
in x cvector(n, real64)
out y

y = A'*A*x
```

(a) ATAX

```
program BATAX
in beta scalar(real64)
in A matrix(m, n, real64)
in x cvector(n, real64)
out y
y = beta*A'*A*x
```

(b) BATAX

```
program BICGK
in A matrix(m, n, real64)
in p cvector(n, real64)
in r cvector(m, real64)
out q, s

q = A*p
s = A'*r
```

(c) BICGK

```
program GEMV
in A matrix(m, n, real64)
in x cvector(n, real64)
in y cvector(m, real64)
in alpha, beta scalar(real64)
out z

z = alpha*A*x + beta*y
```

(d) GEMV

```
program GEMVT
in A matrix(m, n, real64)
in y cvector(m, real64)
in z cvector(n, real64)
in alpha, beta scalar(real64)
out x, w

x = beta*A'*y + z
w = alpha*A*x
```

(e) GEMVT

```
program HESSBLK
inout A matrix(n, real64)
in u1, u2, v1, v2, x
    cvector(n, real64)
out v, w

A = A - u1*v1' + u2*v2'
v = A'*x
w = A*x
```

(f) HESSBLK

```
program GESUMMV
in A, B matrix(m, n, real64)
in x cvector(n, real64)
in alpha, beta scalar(real64)
out y
y = alpha*A*x + beta*B*x
```

(g) GESUMMV

```
program TRILAZY
in U, Y matrix(m, n, real64)
in y, u cvector(m, real64)
out y

y = y - Y*U'*u - U*Y'*u
```

(h) TRILAZY

Figure 7.1: Example specification in *Vaani*

```
pdgemv(&NT, &m, &n, &done, A[0], &one, &one, descA,
            x, &one, &one, descn, &one,
            &dzero, temp0, &one, &one, descm, &one);
pdgemv(&T, &m, &n, &done, A[0], &one, &one, descA,
            temp0, &one, &one, descm, &one,
            &dzero, y, &one, &one, descn, &one);
```

(a) ATAX

```
pdgemv(&NT, &m, &n, &done, A[0], &one, &one, descA,
            x, &one, &one, descn, &one,
            &dzero, temp, &one, &one, descm, &one);
pdgemv(&T, &m, &n, &beta, A[0], &one, &one, descA,
            temp, &one, &one, descm, &one,
            &dzero, y, &one, &one, descn, &one);
```

(b) BATAX

```
pdgemv(&NT, &m, &n, &done, A[0], &one, &one, descA,
            p, &one, &one, descn, &one,
            &dzero, q, &one, &one, descm, &one);
pdgemv(&T, &m, &n, &done, A[0], &one, &one, descA,
            r, &one, &one, descm, &one,
            &dzero, s, &one, &one, descn, &one);
```

(c) BICGK

```
pdgemv(&NT, &m, &n, &alpha, A[0], &one, &one, descA,
            x, &one, &one, descn, &one,
            &beta, y, &one, &one, descm, &one);
```

(d) GEMV

Figure 7.2: Computation sections for MKL programs

```
pdger(&m, &n, &done, u1, &one, &one, descm, &one,
                    v1, &one, &one, descn, &one,
                    A[0], &one, &one, descA);
pdger(&m, &n, &done, u2, &one, &one, descm, &one,
                    v2, &one, &one, descn, &one,
                    A[0], &one, &one, descA);
pdgemr2d(&n, &one, z, &one, &one, descn,
                   x, &one, &one, descn, &context);
pdgemv(&T, &m, &n, &beta, A[0], &one, &one, descA,
                    y, &one, &one, descm, &one,
                    &done, x, &one, &one, descn, &one);
pdgemv(&NT, &m, &n, &alpha, A[0], &one, &one, descA,
                    x, &one, &one, descn, &one,
                    &dzero, w, &one, &one, descm, &one);
```

(e) GEMVER

```
pdgemr2d(&n, &one, z, &one, &one, descn,
                   x, &one, &one, descn, &context);
pdgemv(&T, &m, &n, &beta, A[0], &one, &one, descA,
                    y, &one, &one, descm, &one,
                    &done, x, &one, &one, descn, &one);
pdgemv(&NT, &m, &n, &alpha, A[0], &one, &one, descA,
                    x, &one, &one, descn, &one,
                    &dzero, w, &one, &one, descm, &one);
```

(f) GEMVT

```
pdgemv(&NT, &m, &n, &alpha, A[0], &one, &one, descA,
            x, &one, &one, descn, &one,
            &dzero, y, &one, &one, descm, &one);
pdgemv(&NT, &m, &n, &beta, B[0], &one, &one, descA,
            x, &one, &one, descn, &one,
            &done, y, &one, &one, descm, &one);
```

(g) GESUMMV

Figure 7.2: Computation sections for MKL programs (cont.)

```
pdger(&n, &n, &dmone, u1, &one, &one, descn, &one,
                    v1, &one, &one, descn, &one,
                    A[0], &one, &one, descA);
pdger(&n, &n, &done, u2, &one, &one, descn, &one,
                    v2, &one, &one, descn, &one,
                    A[0], &one, &one, descA);
pdgemv(&T, &n, &n, &done, A[0], &one, &one, descA,
                    x, &one, &one, descn, &one,
                    &dzero, v, &one, &one, descn, &one);
pdgemv(&NT, &n, &n, &done, A[0], &one, &one, descA,
                    x, &one, &one, descn, &one,
                    &dzero, w, &one, &one, descn, &one);
```

(h) HESSBLK

```
pdgemv(&T, &m, &n, &done, U[0], &one, &one, descA,
                    u, &one, &one, descm, &one,
                    &dzero, temp1, &one, &one, descn, &one);
pdgemv(&NT, &m, &n, &dmone, Y[0], &one, &one, descA,
                    temp1, &one, &one, descn, &one,
                    &done, y, &one, &one, descm, &one);
pdgemv(&T, &m, &n, &done, Y[0], &one, &one, descA,
                    u, &one, &one, descm, &one,
                    &dzero, temp1, &one, &one, descn, &one);
pdgemv(&NT, &m, &n, &dmone, U[0], &one, &one, descA,
                    temp1, &one, &one, descn, &one,
                    &done, y, &one, &one, descm, &one);
```

(i) TRILAZY

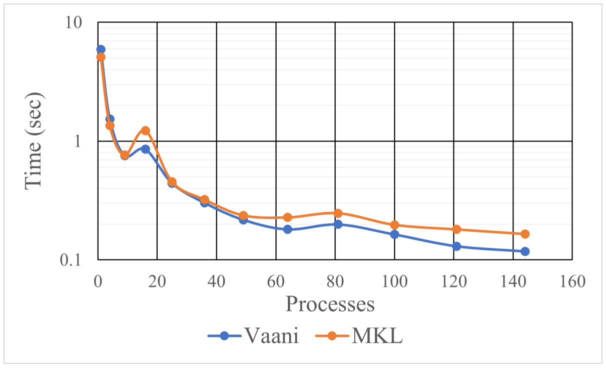
Figure 7.2: Computation sections for MKL programs (cont.)

```
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <mpi.h>

#include "mkl_pblas.h"
#include "mkl_blacs.h"
#include "mkl_scalapack.h"

int32_t main(int32_t argc, char** argv) {
    /*** Declarations Begin ***/
    MKL_INT m, n, mp0, np1, np0, mb, nb;
    MKL_INT nprocs, rank, p, rowrank, colrank;
    int32_t dimsizes[2];
    double ...;
    /*** Declarations End ***/

    /*** Grid Setup Begin ***/
    MKL_INT context;
    MKL_INT info, one = 1, zero = 0;
    blacs_pinfo(&rank, &nprocs);
    dimsizes[0] = dimsizes[1] = 0;
    MPI_Dims_create(nprocs, 2, dimsizes);
    assert(dimsizes[0] == dimsizes[1]);
    p = dimsizes[0];
    blacs_get(&zero,&zero,&context);
    blacs_gridinit(&context, "R", &p, &p);
    blacs_gridinfo(&context, &p, &p, &rowrank, &colrank);
    /*** Grid Setup End ***/

    /*** Local Size Computation Begin ***/
    mb = m/p + ((m%p==0)?0:1);
    nb = n/p + ((n%p==0)?0:1);
    mp0 = (m%p == 0)? mb : ((rowrank == p-1)?m%mb : mb);
    np0 = (n%p == 0)? nb : ((rowrank == p-1)?n%nb : nb);
    /*** Local Size Computation Begin ***/
```

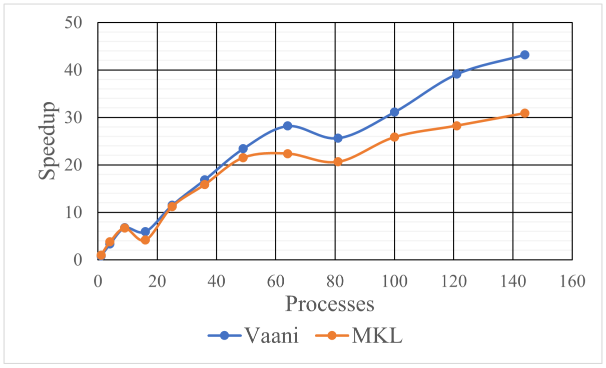
Figure 7.3: Boilerplate for MKL programs

```
    /*** MKL Description Begin ***/
    MKL_INT descA[9], descm[9], descn[9];
    char NT = 'N';
    char T = 'T';
    double dzero = 0.0, done = 1.0, dmone = -1.0;
    descinit(descA, &m, &n, &mb, &nb, &zero, &zero,
                &context, &mp0, &info);
    descinit(descm, &m, &one, &mb, &one, &zero, &zero,
                &context, &mp0, &info);
    descinit(descn, &n, &one, &nb, &one, &zero, &zero,
                &context, &np0, &info);
    /*** MKL Description End ***/

    /*** Array allocation and initialization Begin ***/
    ...
    /*** Array allocation and initialization End ***/

    /*** Computation Begin ***/
    ...
    /*** Computation End ***/

    /*** Free Arrays Begin ***/
    ...
    /*** Free Arrays End ***/

    blacs_exit(0);
}
```

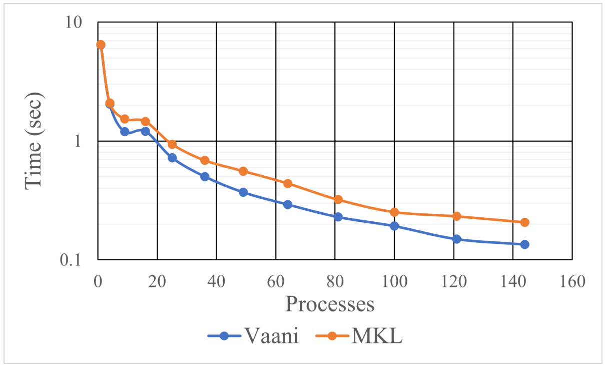
Figure 7.3: Boilerplate for MKL programs (cont.)



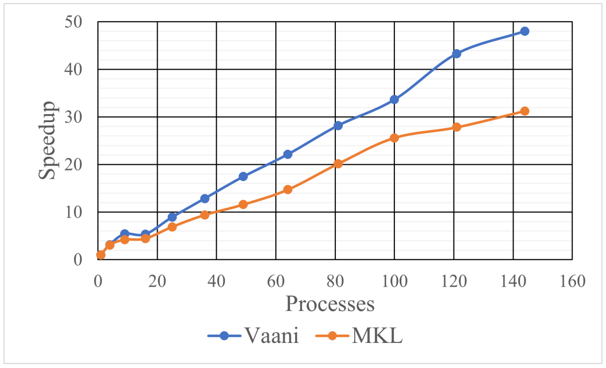
(a) ATAX execution time



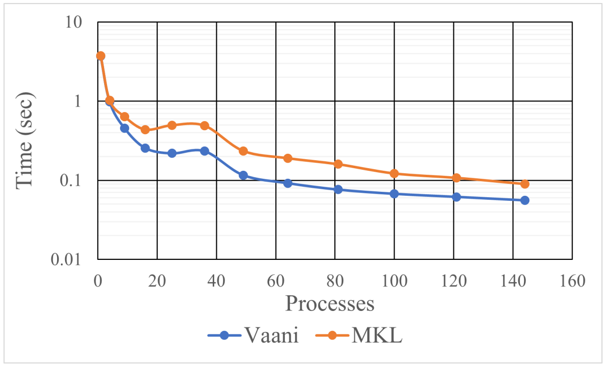
(b) ATAX speedup



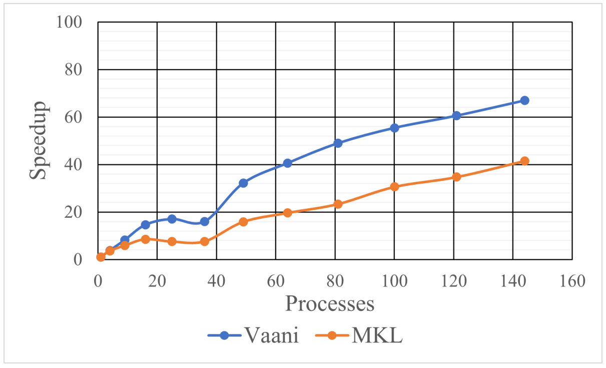
(c) BATAX execution time



(d) BATAX speedup

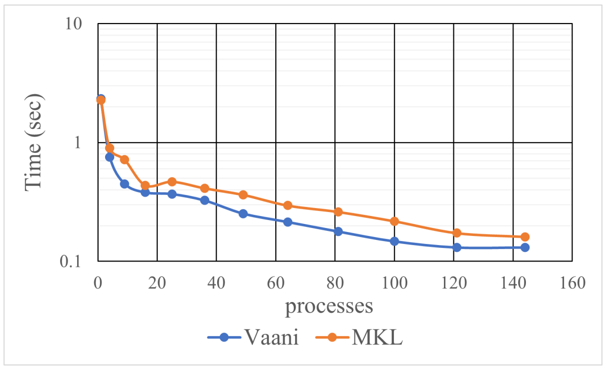


(e) BICGK execution time

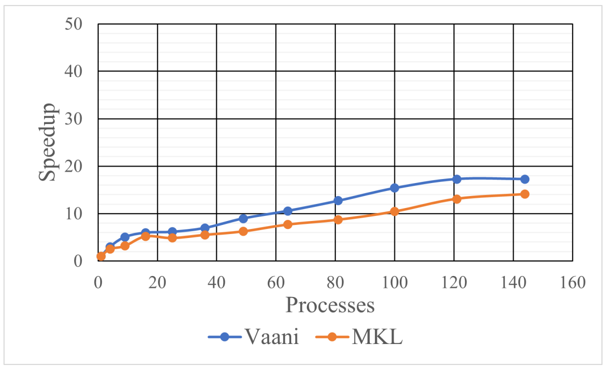


(f) BICGK speedup

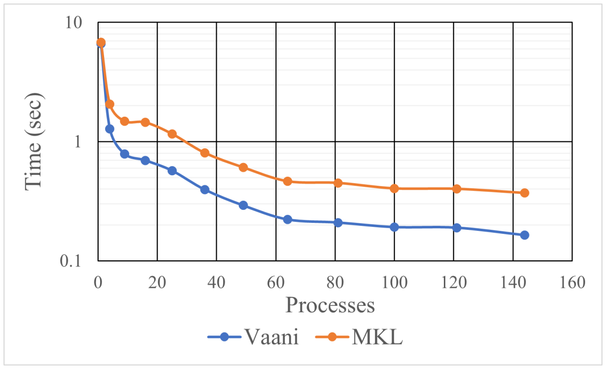
Figure 7.4: Comparison to Intel® MKL



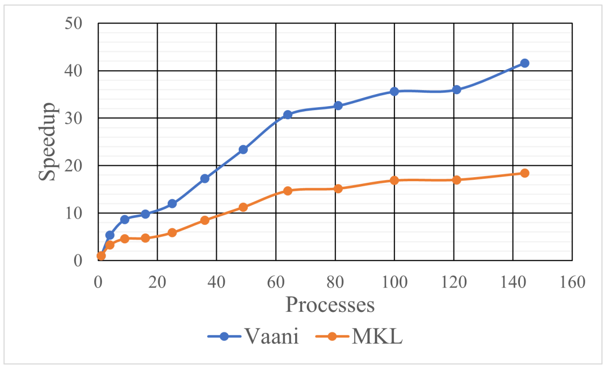
(g) GEMV execution time



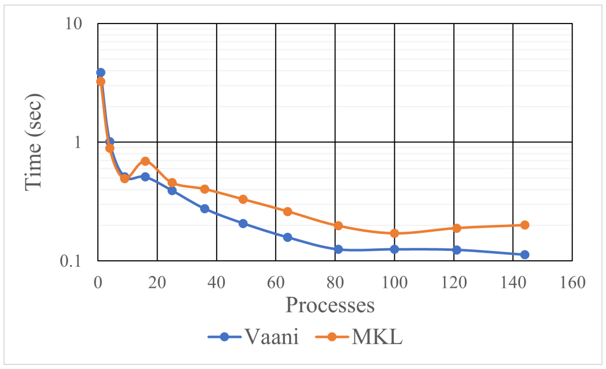
(h) GEMV speedup



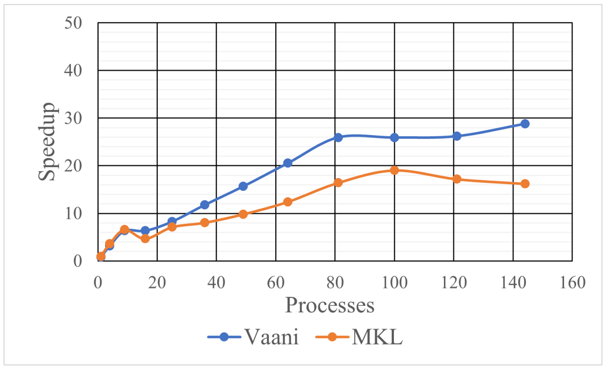
(i) GEMVER execution time



(j) GEMVER speedup

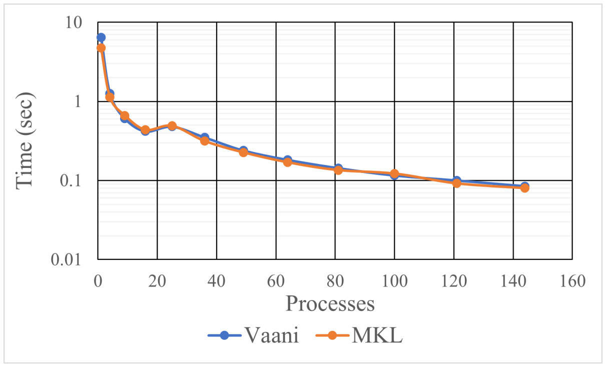


(k) GEMVT execution time

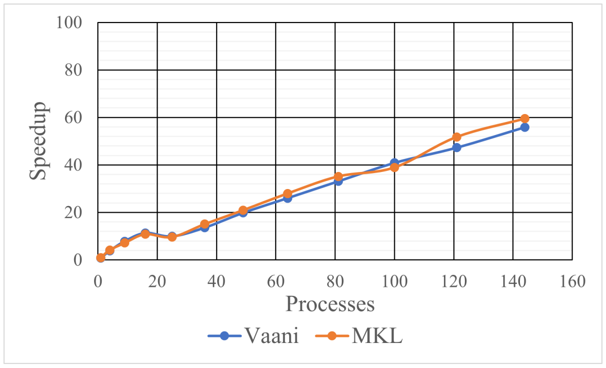


(l) GEMVT speedup

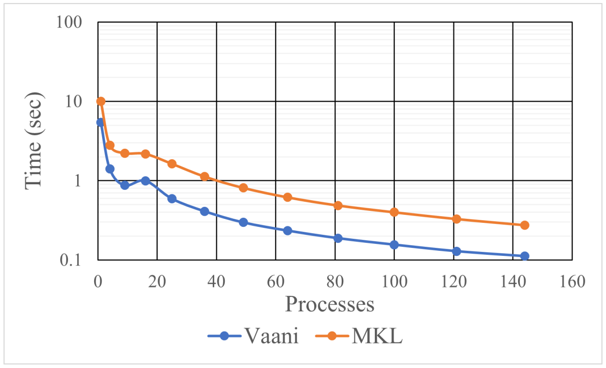
Figure 7.4: Comparison to Intel® MKL (cont.)



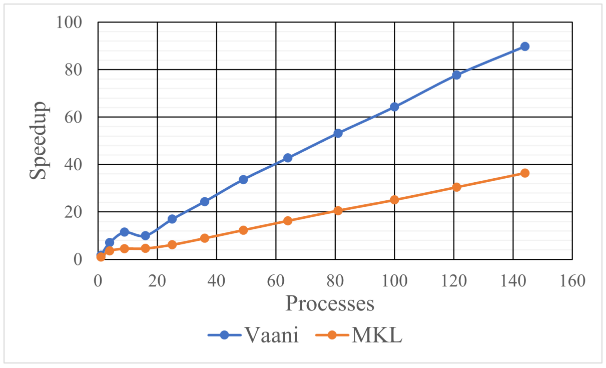
(m) GESUMMV execution time



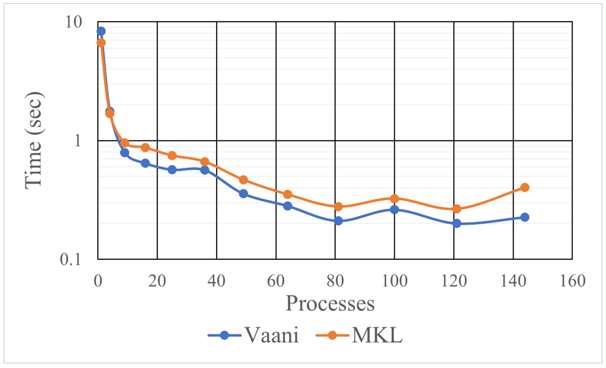
(n) GESUMMV speedup



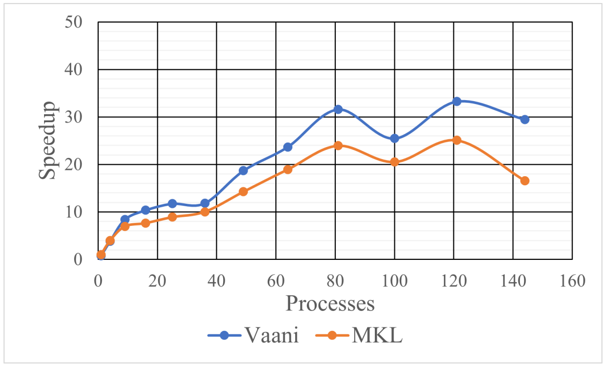
(o) HESSBLK execution time



(p) HESSBLK speedup

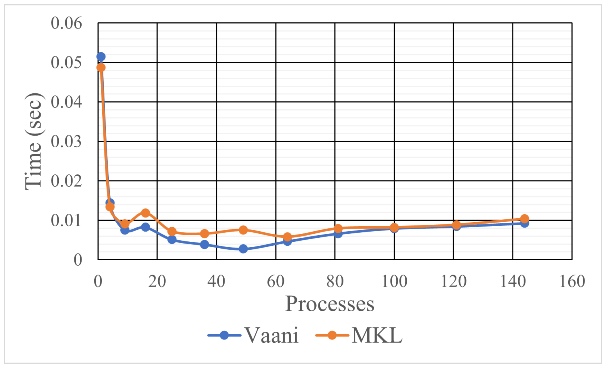


(q) TRILAZY execution time

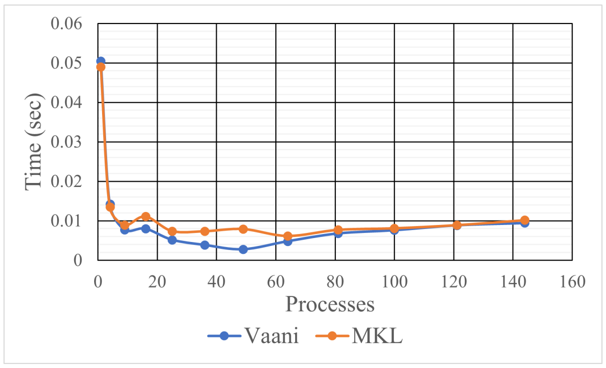


(r) TRILAZY speedup

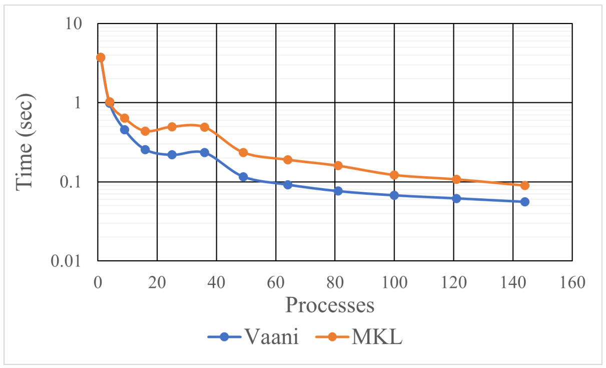
Figure 7.4: Comparison to Intel® MKL (cont.)



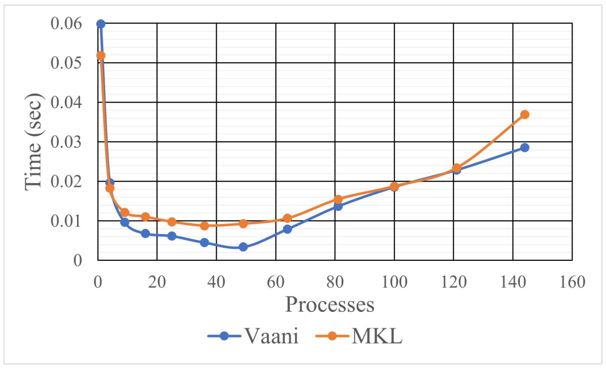
(a) ATAX execution time



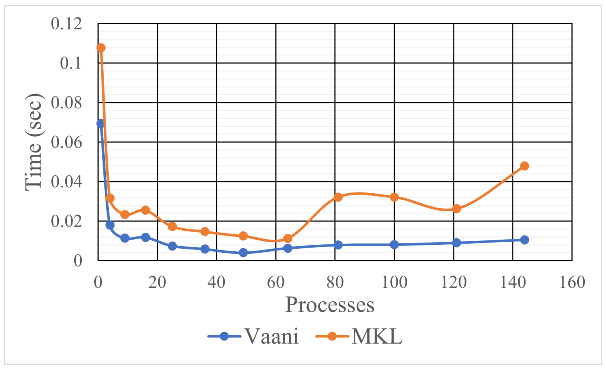
(b) BATAX execution time



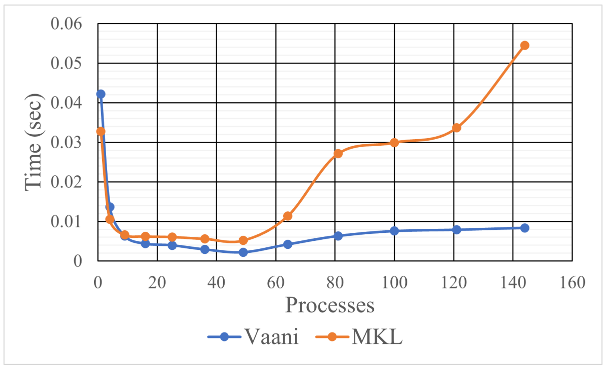
(c) BICGK execution time



(d) GEMV execution time

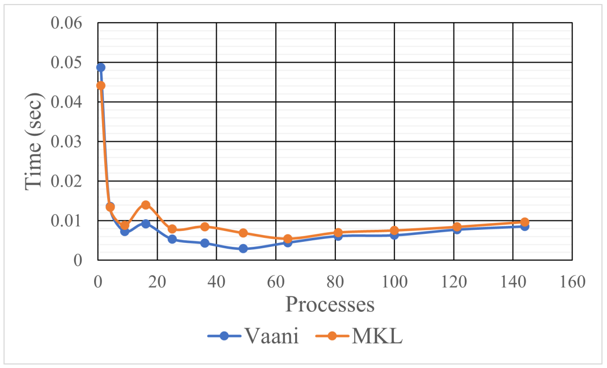


(e) GEMVER execution time

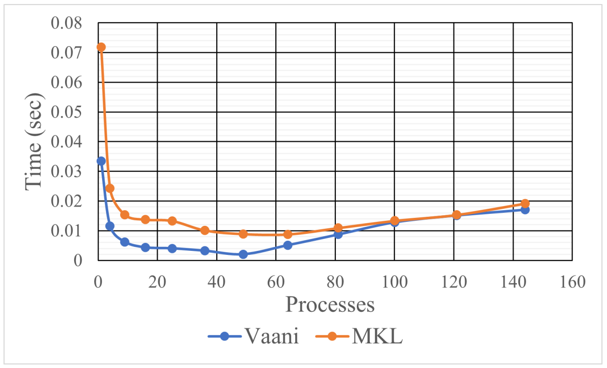


(f) GEMVT execution time

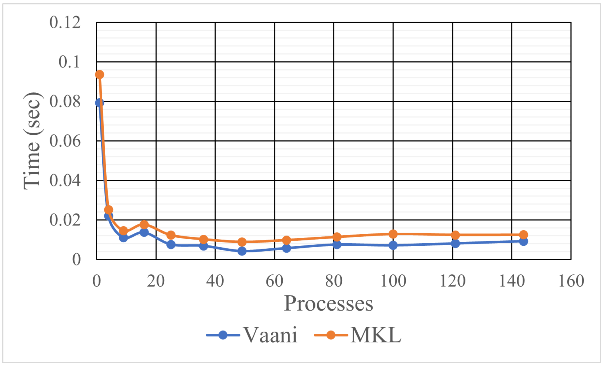
Figure 7.5: Comparison to Intel® MKL for small matrices



(g) GESUMMV execution time



(h) HESSBLK execution time

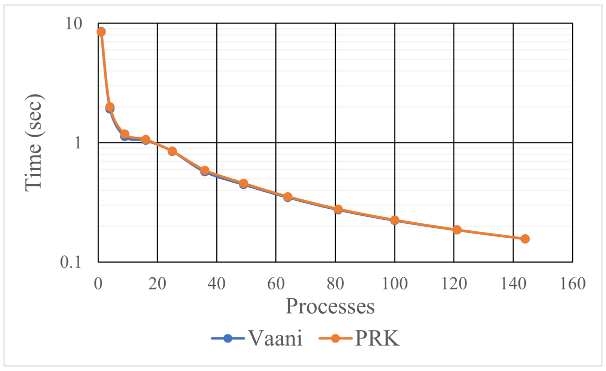


(i) TRILAZY execution time

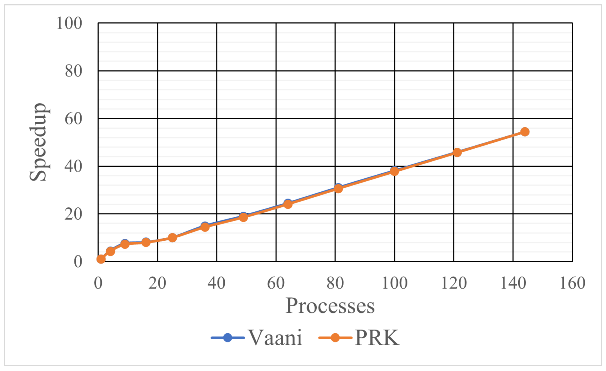
Figure 7.5: Comparison to Intel® MKL for small matrices (cont.)

```
program rad2
in A matrix(n, n, real64)
out A, B

A, B = rec A, B [5] {
  B = -0.125*A{0, -2} + -0.25*A{0, -1}
      + 0.125*A{0, 2} + 0.25*A{0,1}
      + -0.125*A{-2, 0} + -0.25*A{-1, 0}
      + 0.125*A{2, 0} + 0.25*A{1, 0}
      with boundary=none
  A = A + 1
} with {
  A[0] = A
}
```

Figure 7.6: Stencil specification in *Vaani*



(a) execution time



(b) speedup

Figure 7.7: Comparison to PRK Stencil

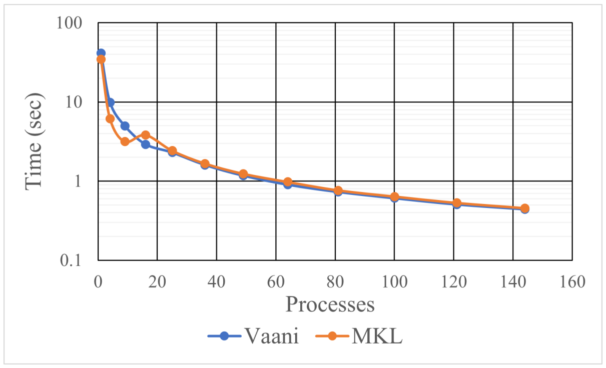
```
program powit
in A matrix(n, real64)
in v cvector(n, real64)
out eig

eig = rec vn [50][diff > 1e-12]
{
    vn = A*vn{-1}
    vn = vn/vn'*vn
    diff = reduce(vn{0} - vn{-1})
} with {
    vn[0] = v
}
```

Figure 7.8: Power iteration specification in *Vaani*

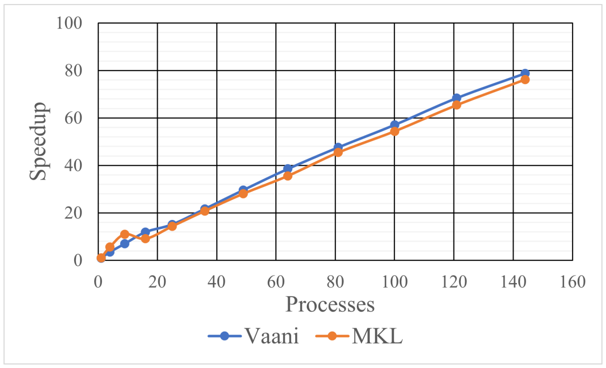
Intel MKL is given in Figure 7.9. The execution time and speedup with respect to single process MKL code are given in Figure 7.10. Similar to BLAS routines, *Vaani* although slower than MKL for small number of processes, performs similar to or better than MKL for larger number of processes.

```
for(rc0 = 1; rc0 < 51, diff > 1e-12; rc0++) {
    pdgemv(&NT, &n, &n, &done, A[0], &one, &one, descA,
                vn[(1+rc0)%2], &one, &one, descn, &one,
                &dzero, vn[rc0%2], &one, &one, descn, &one);
    pdnrm2(&n, &norm, vn[rc0%2], &one, &one, descn, &one);
    norm = 1.0/norm;
    pdscal(&n, &norm, vn[rc0%2], &one, &one, descn, &one);
    pdcopy(&n, vn[rc0%2], &one, &one, descn, &one
               temp0, &one, &one, descn, &one);
    pdaxpy(&n, &dnone, vn[(rc0+1)%2], &one, &one, descn, &one,
               temp0, &one, &one, descn, &one);
    pdasum(&n, &diff, temp0, &one, &one, descn, &one);
    MPI_Bcast(&diff, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Figure 7.9: Power iteration specification in *Vaani*



(a) execution time



(b) speedup

Figure 7.10: Comparison of Power Iteration to Intel MKL

CHAPTER 8: RELATED WORK

Array computations are a major component of high performance computing, and thus, extensive research has been performed to improve programmer productivity and program performance targeting single core, multi-core, accelerator and distributed memory computing. This section describes the relevant related work, and where our work stands in the larger picture.

This body of related work is described in two different ways. First, systems with similar ideas or goals as our system are discussed in comparison to *Vaani*. Then, the literature that shapes each component of *Vaani* are described.

8.1 RELATED SYSTEMS

In this section, systems that have similar goals as *Vaani* are described. The body of work related to *Vaani* can be classified in two ways, first, based on the goal of the system, and second, based on the techniques or underlying design principles.

8.1.1 Goal-Based Classification

The goals can be classified into

1. Automatic parallelization
2. Array notation languages
3. High performance libraries
4. High performance runtime systems
5. Domain specific languages

Automatic Parallelization

Automatic parallelization from sequential code in C/Fortran has been tried and successful to a certain degree in the SUIF [6] [7] and Polaris [8] compilers, and pluto for shared memory [9] and distributed memory [10] systems. These work well for small kernels but do not provide any flexibility to the user. These are good tools to automatically parallelize existing C code, but do not leverage the additional information and representation ease of high-level notations.

Compilation from MATLAB programs to map onto ScaLAPACK [11] and to C on distributed memory systems [12] take a similar high level notation, but do not provide the flexibility and choice as our system.

Array Notation Languages

Array notation languages use arrays as first class objects, and allow users to manipulate arrays directly. Languages such as ZPL [13], Co-array Fortran [14], High Performance Fortran [15], Chapel [16] and X10 [17] are close to our work in terms of the input, target systems, and possible optimizations. However, these languages aim to be general purpose languages, and thus do not deliver as much flexibility and performance as our system can potentially deliver.

High Performance Libraries

Another approach commonly taken for high performance array operations is the use of high performance libraries like PetSc [18] for scientific computing. ScaLAPACK [19] provides a set of linear algebra routines for distributed memory systems and most vendors have their custom implementations. ATLAS [20] autotunes BLAS routines for a system. The main drawback of library implementations is the additional overhead of the library, and the lack of optimizations across library routines.

High Performance Runtime Systems

Bohrium [21] and TensorFlow [22] are runtime systems from python to distributed memory systems and Numba [23] is a just-in-time compiler that compiles python code to C and MPI before executing. [24] describes just-in-time compilation for Julia to use productivity languages for performance too.

Domain Specific Languages

Domain specific languages (DSL) have been used for linear algebraic expressions in [25,26,27]. Lgen [25], [26] generates efficient kernels for small, fixed size inputs and targets single core and vectorization strategies. BTO [27] generates efficient kernels for sequences of level-1 and level-2 BLAS operations and targets shared memory systems, relying on the compiler for vectorization. Hydra [28] explores automatic code generation from linear algebraic equations and targets shared memory systems. Pochoir [29] is a DSL for stencil computations targeting shared memory systems, while Physis [30], Stella [31] use an embedded DSL for stencil computations targeting GPU-accelerated supercomputers. Liszt [32] is a DSL for stencil computations on unstructured grids using graph as a primitive and graph accesses to deduce the stencil. CTF [33] is an embedded DSL for tensor contractions, Elemental [34] is an embedded DSL for linear algebra based on the FLAME [35] approach, implemented as a C++ template library. Eigen [36], Armadillo [37] and MTL4 [38,39] are also C++ template libraries providing linear algebra and other matrix operations for shared memory systems. MTL4 provides a proprietary supercomputing edition that provides distributed data structures and parallel operations on these structures.

Hierarchically tiles arrays (HTA) [40] provide an abstract to view arrays as a hierarchical tiles, and support parallelization onto distributed memory systems, shared memory systems and optimizations for data locality.

Loopy [41] is a code generator for array based codes for accelerators based on polyhedral framework, that allows users to specify desired transformations and optimizations for code generation.

Delite [42,43] is a framework to enable development of domain specific languages and high performance compilers based on lightweight modular staging principles [44], and OptiML [45] is a DSL on top of Delite that supports machine learning operations.

Halide [46] is an embedded DSL for image processing pipelines which separates the computation specification from the schedule and optimization, so the users can specify the optimizations explicitly providing flexibility and tuning capabilities. They also use auto-tuning to automatically generate tuned code for shared memory and GPU accelerated systems. Distributed Halide [47] is an extension to Halide that targets distributed memory systems.

AlphaZ [48] is a system using polyhedral framework that enables exploration of transformations and optimizations of affine loop nests. Chill [49] is another framework that exposes a scheduling language to allow transformations on affine loop nests.

Tiramisu [50] is a polyhedral compiler with an embedded DSL in C++ for dense and sparse DNN and data parallel algorithms.

Tensor Comprehensions [51] is a C++ library to automatically synthesize high-performance machine learning kernels using Halide [46], ISL [52] and NVRTC [53] or LLVM [54].

Lift [55, 56] is a DSL for generating high performance GPU code using a high-level functional data parallel language with a system of rewrite rules which encode algorithmic and hardware-specific optimisation choices.

8.1.2 Technique-Based Related Systems

Some common methods, techniques and design principles are explored in this section.

Polyhedral Compilation

A large section of the related work is based on polyhedral compilation techniques based on pressburger arithmetic [57]. Automatic compilers like Pluto [9, 10], algorithms in SUIF [7], work by Adve et al. [58], Bondhugala et al. [59], DSLs like LGen [26], Chill [49], AlphaZ [48], Tiramisu [50], TensorComprehensions [51], Loopy [41] all use the polyhedral framework. Integer set library (ISL) [52], PolyLib [60] and Princess [61] are some polyhedral libraries that form the backbone of these techniques. CLooG [62] is a code generator that is used frequently in polyhedral code generation. Polyhedral framework provides a strong and expressive representation for affine loop nests and enables several transformations like loop skewing and strip mining. It gives tools to reason about imperfect (not all statements are in the innermost region) and non-rectangular loop nests. However, the major drawback of these systems is that we cannot split an iteration domain into either parameterized number of pieces or into blocks of a parameterized size. This limitation has restricted the usage to fixed size tiles and even fixed number of processes (in Tiramisu [50]) in these systems. Some works like [7] [58], [63] skirt this restriction by using processes like index variables in the loop nest, and manually dividing the iteration spaces during code generation. *Vaani* does not take this route, as it partitions the data early in the processing (MLIR), and using this framework at this stage is not convenient. However, *Vaani* could use this framework for code generation from LLIR, but does not currently do so.

Pattern Based Compilation

Another common compilation technique is to identify patterns in the program and use optimizations or strategies based on these patterns. These patterns could be used to perform optimizations like in constant folding, strength reduction, expression substitution, etc., or transforming from one level of abstraction to another, like extracting communication. Spiral [64], LGen [25], BTO [27], Lift [55, 56], Delite framework [42] and its derivative DSLs use pattern matching to perform optimizations. *Vaani* also uses pattern based techniques in HLIR optimizations, in the design of the MLIR and the transformation from MLIR to LLIR.

Well Defined Intermediate Representations

Increasingly many systems are exposing a well defined intermediate representation to users, to provide a common platform for optimizations and to allow collective development in an open source framework. Pencil [65] and Lift [55] define intermediate languages for accelerator programming. Halide [46], Tiramisu [50], Delite [42], LGen [25], BTO [27], Bohrium [21], TensorFlow [22] etc. define intermediate representations that expose interfaces that can be modified by a user or a tuner. *Vaani* follows a similar viewpoint in the importance of exposing intermediate representations.

Interfaces for User Selection

POET [66], AlphaZ [48], Halide [46, 47] and Tiramisu [50] provide a language, commands or interface for users or tuners to select transformations. Chapel [16] and HPF [15] provide constructs to define parallelism and mapping in their programming language. Optimizations and transformations for best performance are program and target dependent, and it is difficult to select a set of optimizations that work for all cases. So, these systems provide flexibility to users or autotuners to select the transformations, and explore the design space, without manually writing low level code.

8.1.3 Where Our Work Stands

Our work provides a transparent and flexible interface that enables users to express their program in a high level notation and generate efficient distributed memory code comparable to hand-optimized versions in significantly less time. We do not intend to be a plug-and-play performance boost, and instead we provide a framework to ease the process of development of code by hand.

The closest work, in our opinion, that shares design ideas with *Vaani* is Tiramisu, in that it also uses multiple intermediate layers to define the computations. It believes that providing interfaces for transformations are important for prototyping or autotuning. Major difference is in the way the layers are selected. Tiramisu views all computations as expressions with associated loop nests in the polyhedral framework. Whereas, *Vaani* views computations at an array level, with a focus on the actual operations in the first layer. *Vaani* looks at the patterns in the computation only in MLIR and iteration spaces are not considered till LLIR. Another significant difference is that Tiramisu generates code for distributed memory systems by mapping a dimension of the affine loop nest to the processes and generating send/receive calls for communication. *Vaani*, on the other hand, maps computations to distributed memory systems using multidimensional virtual process grids, and a set of send/recv and collective operations. Tiramisu does not support parametric tiling, multidimensional grids and collective operations. Tiramisu targets a broader set of applications by accepting any affine loop nests, while *Vaani* is restricted to rectangular array operations.

We also observe that the design of Tiramisu, and other systems targeting distributed memory systems like Halide [47] concentrate on optimizing first for single core, multi-core, GPUs and finally distributed memory systems, which is how automatic parallelization has evolved. *Vaani*, on the other hand, first optimizes for distributed memory systems and then focuses on local optimizations. We believe that *Vaani* has a more intuitive approach to code generation for distributed systems, and argue that it closely follows the steps a user would take to manually write code in MPI.

8.2 RELATED COMPONENTS

In this section, some individual components that together form the system *Vaani* are considered. The design of the system follows the principle of separation of concerns [67], that is used extensively in compiler design to break complex tasks into simpler manageable tasks.

The design of the high level notation is inspired by array programming languages like MATLAB [1], APL [68], and NumPy [2]. In particular, *Vaani* uses the "'.'$\langle operator \rangle$" notation from MATLAB to indicate element-by-element operations, and the broadcasting definition from NumPy. The recurrence construct is inspired by the mathematical formulations of recurrence relations [69], and its widespread use in formulating scientific computing problems.

The parser for the high level notation is developed using lark parser [4], which uses LALR parsing techniques [70].

The HLIR is a directed acyclic graph (DAG), which has been traditionally used to represent expressions in compilers [70]. The implementation of the expression hierarchy and the internal structures are inspired by the expression handling in symbolic computation libraries like SymPy [71], and Python package Pymbolic.

Merging operations in MLIR is similar to loop fusion and has been presented in [72]. Bohrium [21] implements a fusion algorithm [73] to merge NumPy [2] operations where they formulate the merge as a weighted graph partitioning problem which is NP-hard and define heuristics to approximately solve the problem efficiently. *Vaani* uses a simpler heuristic and only merges parent-child nodes, and provides interface to merge nodes manually. *Vaani* does not aggressively merge all possible nodes to not create false dependencies for the lower levels and to encourage overlap of communication and computation.

Code generation for distributed memory systems from loop nests has been proposed by Amarasinghe et al. [7] and Bondhugula et al. [59] using polyhedral frameworks. These models compute send and receive sets on each process and introduce send/receive communication nodes. This is different from *Vaani*, in that *Vaani* supports a richer communication layer, with point-to-point and collective communications. However, these works support loop nests that are more generic and versatile compared to the ones *Vaani* supports.

Vaani's partitioning of data is a parametric tiling of the data. Parametric tiling in systems like Halide [46] is performed on the intervals domains, which is similar in concept to *Vaani*. Both these systems have perfect loop nests with rectangular iteration spaces. On a related note, parametric tiling of imperfect loop nests is performed in [74], where each statement in the iteration space is embed into a special product space, and tiling is performed on this space. PrimeTile [63] uses polyhedral models to extract polyhedrons based on cloog [62], transform the loop nests to allow rectangualr tiling, and split the loop nests into rectangularly tilable regions, and prologue and epilogue portions.

Vaani generates code for C with MPI [75]. Other target systems that *Vaani* could potentially be extended to support are GasNet [76], a high performance communication interface, Charm++ [77], a parallel programming framework with an adaptive runtime system, and Legion [78], a parallel programming system that separates specification of computation and parallelization.

Development of C programs in MPI is described in [79, 80]. Several algorithms are used for matrix multiplication depending on the distribution and grid dimensions. Modified Cannon's algorithm [81] is used for matrix multiplication on square grids where we allow for non-square matrices, while a variant of SUMMA [82] is used for block-cyclic distribution on 2D rectangular grids. Matrix multiplication on a 3D grid is described in [83], and a 2.5 D algorithm (not currently in *Vaani* is described in [84].

Vaani uses a simple heuristic to determine the order of matrix multiplications, in that if one of them is a vector, it reduces to matrix-vector product, where possible. This is because *Vaani* assumes a symbolic array size, and more complex analyses are not possible. This problem is solved by dynamic programming in [85], and even more optimally using triangulation of polygons in [86, 87].

Transpose of a block-cyclic distributed matrix on a rectangular grid is described in [88]. The algorithm to generate communication for rearrange nodes in translation from MLIR to LLIR is based on a generalization of the algorithms implemented in ScaLAPACK [19] and C++ library Elemental [34].

Program order is computed by topologically sorting the nodes in LLIR. This is obtained by using reverse post order traversal described in [89]. An optimization to overlap communication and computation is not currently implemented in *Vaani*. It promises to be an effective optimization [90], but has mixed results in current MPI implementations as described in [91].

Buffer allocation in *Vaani* is based on register allocation in [70] and [92]. Particularly, [92] talks about allocation on DAGs using dependency analysis, which is what *Vaani* performs. However, *Vaani* does not have a fixed number of registers, like in register allocation, and hence, allocates as many buffers as needed.

Vaani defines recurrence constructs but currently implements them sequentially, similar to Tiramisu [50], OptiML [45]. *Vaani* could potentially benefit from parallelizing certain recurrences, for example, ones in which each iteration is independent, or ones which resolve to be reduction or scan operations. Extracting parallelism from recurrences has been explored in [93, 94].

CHAPTER 9: FUTURE WORK

Vaani provides a tool to easily generate C code using MPI. We discuss a few directions in which *Vaani* could be expanded upon in the future.

9.1 OPERATION SUPPORT

Vaani has a language that is easy to expand and support more operations. Some possible operations are given below.

- Parameterized offset indexing to specify parameterized stencils, that can be used to express convolutions, or more complex stencils.
- Expansion of recurrence notation to support iterative matrix algorithms.
- Delayed update operator to platform independently specify bulk-synchronous algorithms.

9.2 USER OPTIONS AND OPTIMIZATIONS

A range of options and optimizations are possible to be included in *Vaani*, which is modularly designed to incorporate expansions easily. Some examples are given below.

- Options to select different MPI patterns for the same communication, for example, boundary exchange could use row and column communicators (currently used), create specific neighbor communicators, experiment with different types of send/recv pairs, etc.
- More optimization passes at each of the IR levels.
- Optimize recurrences, if possible, by analyzing the patterns and parallelizing it.

9.3 TUNING

Vaani provides a number of options in various IRs to easily generate different versions of the code for the same program. This provides handles to tune the code by experimenting with various options. A possible future direction is to support autotuning in *Vaani*, either replacing the user interaction or augmenting it.

CHAPTER 10: CONCLUSION

To bridge the gap between productive high-level languages and high-performing C code, this work proposes a series of intermediate representations to provide handles for selection of data and computation partitioning and mapping, optimizations and structure of the generated code. First, it proposes a new notation combining ideas from several existing array notation languages, coupled with a few new constructs, to succinctly represent computations. Then it proposes a set of intermediate representations that can lower a high level specification to low-level C code for distributed memory systems in a natural and intuitive manner. Then it creates an interactive framework based on these representations to generate C code from a high-level notation. We have also demonstrated that the generated code is competent compared to efficient library implementations.We expect our system to be adopted to ease writing of C code by hand.

REFERENCES

[1] MATLAB, *9.7.0.1190202 (R2019b).* Natick, Massachusetts: The MathWorks Inc., 2018.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del R'ıo, M. Wiebe, P. Peterson, P. G'erard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[3] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra.* USA: Society for Industrial and Applied Mathematics, 2011.

[4] E. Shinan, "Lark parser," https://github.com/lark-parser/lark/, 2017.

[5] "Parallel research kernels (prk)," https://github.com/ParRes/Kernels.

[6] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam, "Maximizing multiprocessor performance with the suif compiler," *Computer*, vol. 29, no. 12, pp. 84–89, Dec. 1996. [Online]. Available: http://dx.doi.org/10.1109/2.546613

[7] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, ser. PLDI '93. New York, NY, USA: Association for Computing Machinery, 1993. [Online]. Available: https://doi.org/10.1145/155090.155102 p. 126138.

[8] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, "Polaris: The next generation in parallelizing compilers," in *PROCEEDINGS OF THE WORKSHOP ON LANGUAGES AND COMPILERS FOR PARALLEL COMPUTING.* Springer-Verlag, Berlin/Heidelberg, 1994, pp. 10–1.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375595 pp. 101–113.

[10] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503289 pp. 33:1–33:12.

[11] S. Ramaswamy, E. W. Hodges, IV, and P. Banerjee, "Compiling matlab programs to scalapack: Exploiting task and data parallelism," in *Proceedings of the 10th International Parallel Processing Symposium*, ser. IPPS '96. Washington, DC, USA: IEEE Computer Society, 1996. [Online]. Available: http://dl.acm.org/citation.cfm?id=645606.661031 pp. 613–619.

[12] L. Reis, J. a. Bispo, and J. a. M. P. Cardoso, "Ssa-based matlab-to-c compilation and optimization," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY 2016. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2935323.2935330 pp. 55–62.

[13] B. L. Chamberlain, S. eun Choi, S. J. Deitz, and L. Snyder, "The high-level parallel language zpl improves productivity and performance," in *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[14] R. W. Numrich, *Coarray Fortran*. Boston, MA: Springer US, 2011, pp. 304–310. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_477

[15] C. H. Koelbel and M. E. Zosel, *The High Performance FORTRAN Handbook*. Cambridge, MA, USA: MIT Press, 1993.

[16] B. L. Chamberlain, *Chapel (Cray Inc. HPCS Language)*. Boston, MA: Springer US, 2011, pp. 249–256. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_54

[17] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005. [Online]. Available: https://doi.org/10.1145/1094811.1094852 p. 519538.

[18] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2017. [Online]. Available: http://www.mcs.anl.gov/petsc

[19] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[20] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509096 pp. 1–27.

[21] M. R. B. Kristensen, S. A. F. Lund, T. Blum, K. Skovhede, and B. Vinter, “Bohrium: A virtual machine approach to portable parallelism,” in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, May 2014, pp. 312–321.

[22] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[23] S. K. Lam, A. Pitrou, and S. Seibert, “Numba: A llvm-based python jit compiler,” in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM ’15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[24] T. A. Anderson, H. Liu, L. Kuper, E. Totoni, J. Vitek, and T. Shpeisman, “Parallelizing Julia with a Non-Invasive DSL,” in *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Müller, Ed., vol. 74. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7269 pp. 4:1–4:29.

[25] D. G. Spampinato and M. Püschel, “A Basic Linear Algebra Compiler,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544155 pp. 23:23–23:32.

[26] D. G. Spampinato and M. Püschel, “A Basic Linear Algebra Compiler for Structured Matrices,” in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO ’16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854060 pp. 117–127.

[27] G. Belter, E. R. Jessup, I. Karlin, and J. G. Siek, “Automating the Generation of Composed Linear Algebra Kernels,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654119 pp. 59:1–59:12.

[28] D. Padua, D. Barthou, and A. X. Duchateau, “Hydra: Automatic algorithm exploration from linear algebra equations,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO ’13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494999 pp. 1–10.

[29] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The pochoir stencil compiler," in *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1989493.1989508 pp. 117–128.

[30] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: An implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063398 pp. 11:1–11:12.

[31] T. Gysi, C. Osuna, O. Fuhrer, M. Bianco, and T. C. Schulthess, "STELLA: A Domain-specific Tool for Structured Grid Methods in Weather and Climate Models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807627 pp. 41:1–41:12.

[32] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan, "Liszt: A domain specific language for building portable mesh-based pde solvers," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063396 pp. 9:1–9:12.

[33] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2013.112 pp. 813–824.

[34] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Softw.*, vol. 39, no. 2, pp. 13:1–13:24, Feb. 2013. [Online]. Available: http://doi.acm.org/10.1145/2427023.2427030

[35] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn, "Flame: Formal linear algebra methods environment," *ACM Trans. Math. Softw.*, vol. 27, no. 4, pp. 422–455, Dec. 2001. [Online]. Available: http://doi.acm.org/10.1145/504210.504213

[36] G. Guennebaud, B. Jacob et al., "Eigen v3," http://eigen.tuxfamily.org, 2010.

[37] C. Sanderson and R. Curtin, "Armadillo: a template-based c++ library for linear algebra," *Journal of Open Source Software*, vol. 1, no. 2, p. 26, 2016. [Online]. Available: https://doi.org/10.21105/joss.00026

[38] P. Gottschling, D. S. Wise, and M. D. Adams, “Representation-transparent matrix algorithms with scalable performance,” in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS ’07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1274971.1274989 p. 116125.

[39] P. Gottschling, D. S. Wise, and A. Joshi, “Generic support of algorithmic and structural recursion for scientific computing,” *International Journal of Parallel, Emergent and Distributed Systems*, vol. 24, no. 6, pp. 479–503, 2009. [Online]. Available: https://doi.org/10.1080/17445760902758560

[40] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua, “The hierarchically tiled arrays programming approach,” in *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems*, ser. LCR ’04. New York, NY, USA: Association for Computing Machinery, 2004. [Online]. Available: https://doi.org/10.1145/1066650.1066657 p. 112.

[41] A. Klöckner, “Loo.py: Transformation-based code generation for gpus and cpus,” in *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY’14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2627373.2627387 p. 8287.

[42] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, “A domain-specific approach to heterogeneous parallelism,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1941553.1941561 p. 3546.

[43] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, “Delite: A compiler architecture for performance-oriented embedded domain-specific languages,” *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, Apr. 2014. [Online]. Available: https://doi.org/10.1145/2584665

[44] T. Rompf and M. Odersky, “Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls,” in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, ser. GPCE ’10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1868294.1868314 p. 127136.

[45] A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Wu, A. R. Atreya, K. Olukotun, T. Rompf, and M. Odersky, “Optiml: An implicitly parallel domain-specific language for machine learning,” in *Proceedings of the 28th International Conference on International Conference on Machine Learning*, ser. ICML’11. Madison, WI, USA: Omnipress, 2011, p. 609616.

[46] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, July 2012. [Online]. Available: http://doi.acm.org/10.1145/2185520.2185528

[47] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed halide," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851157

[48] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *Languages and Compilers for Parallel Computing*, H. Kasahara and K. Kimura, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 17–31.

[49] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," U.of Southern California, Tech. Rep., 2008.

[50] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. IEEE Press, 2019, p. 193205.

[51] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018. [Online]. Available: http://arxiv.org/abs/1802.04730

[52] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software – ICMS 2010*, K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302.

[53] Nvdia, "Nvrtc," https://docs.nvidia.com/cuda/nvrtc/index.html.

[54] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis &amp; transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, ser. CGO '04. USA: IEEE Computer Society, 2004, p. 75.

[55] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: A functional data-parallel ir for high-performance gpu code generation," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, ser. CGO '17. IEEE Press, 2017, p. 7485.

[56] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3168824 p. 100112.

[57] D. C. Oppen, “Elementary bounds for presburger arithmetic,” in *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC ’73. New York, NY, USA: Association for Computing Machinery, 1973. [Online]. Available: https://doi.org/10.1145/800125.804033 p. 3437.

[58] V. S. Adve, J. Mellor-Crummey, M. Anderson, J.-C. Wang, D. A. Reed, and K. Kennedy, “An integrated compilation and performance analysis environment for data parallel programs,” in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing ’95. New York, NY, USA: Association for Computing Machinery, 1995. [Online]. Available: https://doi.org/10.1145/224170.224340 p. 50es.

[59] U. Bondhugula, “Compiling affine loop nests for distributed-memory parallel architectures,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2503210.2503289

[60] D. K. Wild, “A library for doing polyhedral operations,” *Parallel Algorithms and Applications*, vol. 15, no. 3-4, pp. 137–166, 2000. [Online]. Available: https://doi.org/10.1080/01495730008947354

[61] P. Rümmer, “A constraint sequent calculus for first-order logic with linear integer arithmetic,” in *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. LNCS, vol. 5330. Springer, 2008, pp. 274–289.

[62] C. Bastoul, “Code generation in the polyhedral model is easier than you think,” in *PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, Juan-les-Pins, France, September 2004, pp. 7–16.

[63] A. Hartono, M. M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan, “Parametric multi-level tiling of imperfectly nested loops,” in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS ’09. New York, NY, USA: Association for Computing Machinery, 2009. [Online]. Available: https://doi.org/10.1145/1542275.1542301 p. 147157.

[64] M. Puschel, J. M. F. Moura, J. R. Johnson, D. Padua, M. M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, “Spiral: Code generation for dsp transforms,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, Feb 2005.

[65] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, “Pencil: A platform-neutral compute intermediate language for accelerator programming,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 138–149.

[66] Q. Yi, "Poet: A scripting language for applying parameterized source-to-source program transformations," *Softw. Pract. Exper.*, vol. 42, no. 6, p. 675706, June 2012. [Online]. Available: https://doi.org/10.1002/spe.1089

[67] E. W. Dijkstra, *On the Role of Scientific Thought.* Berlin, Heidelberg: Springer-Verlag, 1982.

[68] K. E. Iverson, *A Programming Language.* USA: John Wiley and Sons, Inc., 1962.

[69] G. E. Martin, *Recurrence Relations.* New York, NY: Springer New York, 2001, pp. 113–136. [Online]. Available: https://doi.org/10.1007/978-1-4757-4878-9_6

[70] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition).* USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[71] A. Meurer, C. P. Smith, M. Paprocki, O. Čertík, S. B. Kirpichev, M. Rocklin, A. Kumar, S. Ivanov, J. K. Moore, S. Singh, T. Rathnayake, S. Vig, B. E. Granger, R. P. Muller, F. Bonazzi, H. Gupta, S. Vats, F. Johansson, F. Pedregosa, M. J. Curry, A. R. Terrel, v. Roučka, A. Saboo, I. Fernando, S. Kulal, R. Cimrman, and A. Scopatz, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[72] K. Kennedy and K. S. McKinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 301–320.

[73] M. R. Kristensen, S. A. Lund, T. Blum, and J. Avery, "Fusion of parallel array operations," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2967938.2967945 p. 7185.

[74] N. Ahmed, N. Mateev, and K. Pingali, "Tiling imperfectly-nested loop nests," in *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, ser. SC '00. USA: IEEE Computer Society, 2000, p. 31es.

[75] L. Clarke, I. Glendinning, and R. Hempel, "The mpi message passing interface standard," in *Programming Environments for Massively Parallel Distributed Systems*, K. M. Decker and R. M. Rehmann, Eds. Basel: Birkhäuser Basel, 1994, pp. 213–218.

[76] D. Bonachea, "Gasnet specification, v1.1," University of California at Berkeley, USA, Tech. Rep., 2002.

[77] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York, NY, USA: Association for Computing Machinery, 1993. [Online]. Available: https://doi.org/10.1145/165854.165874 p. 91108.

[78] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC ’12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389086 pp. 66:1–66:11.

[79] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.

[80] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[81] L. E. Cannon, “A cellular computer to implement the kalman filter algorithm,” Ph.D. dissertation, Montana State University, USA, 1969, aAI7010025.

[82] R. A. Van De Geijn and J. Watts, “Summa: scalable universal matrix multiplication algorithm,” *Concurrency: Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291096-9128%28199704%299%3A4%3C255%3A%3AAID-CPE250%3E3.0.CO%3B2-2

[83] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar, “A three-dimensional approach to parallel matrix multiplication,” *IBM J. Res. Dev.*, vol. 39, no. 5, p. 575582, Sep. 1995. [Online]. Available: https://doi.org/10.1147/rd.395.0575

[84] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms,” in *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, ser. Euro-Par’11. Berlin, Heidelberg: Springer-Verlag, 2011, p. 90109.

[85] S. S. Godbole, “On efficient computation of matrix chain products,” *IEEE Transactions on Computers*, vol. C-22, no. 9, pp. 864–866, 1973.

[86] T. C. Hu and M. T. Shing, “Computation of matrix chain products. part i,” *SIAM Journal on Computing*, vol. 11, no. 2, pp. 362–373, 1982. [Online]. Available: https://doi.org/10.1137/0211028

[87] T. C. Hu and M. T. Shing, “Computation of matrix chain products. part ii,” *SIAM Journal on Computing*, vol. 13, no. 2, pp. 228–251, 1984. [Online]. Available: https://doi.org/10.1137/0213017

[88] Jaeyoung Choi, J. J. Dongarra, and D. W. Walker, “Parallel matrix transpose algorithms on distributed memory concurrent computers,” in *Proceedings of Scalable Parallel Libraries Conference*, 1993, pp. 245–252.

[89] M. S. Hecht and J. D. Ullman, “A simple algorithm for global data flow analysis problems,” *SIAM Journal on Computing*, vol. 4, no. 4, pp. 519–532, 1975. [Online]. Available: https://doi.org/10.1137/0204044

[90] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/1810085.1810091 p. 516.

[91] A. Denis and F. Trahay, "Mpi overlap: Benchmark and analysis," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 258–267.

[92] J. R. Goodman and W.-C. Hsu, "Code scheduling and register allocation in large basic blocks," in *ACM International Conference on Supercomputing 25th Anniversary Volume*. New York, NY, USA: Association for Computing Machinery, 1988. [Online]. Available: https://doi.org/10.1145/2591635.2667158 p. 8898.

[93] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Transactions on Computers*, vol. C-22, no. 8, pp. 786–793, 1973.

[94] P. M. Kogge, "Parallel solution of recurrence problems," *IBM J. Res. Dev.*, vol. 18, no. 2, p. 138148, Mar. 1974. [Online]. Available: https://doi.org/10.1147/rd.182.0138