Scheduling and Vectorization for MPC

Benjamin Levy
levyb3@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

Benjamin Sherman
shermb@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

Lindsey Kennard
kennal@rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

Ana L. Milanova
milanova@cs.rpi.edu
Rensselaer Polytechnic Institute
Troy, New York

Muhammad Ishaq*
m.ishaq@ed.ac.uk
University of Edinburgh
Edinburgh, Scotland

Vassilis Zikas†
vzikas@inf.ed.ac.uk
University of Edinburgh
Edinburgh, Scotland

ABSTRACT

CCS CONCEPTS

• **Theory of computation** → **Program analysis**; **Cryptographic protocols**; • **Security and privacy** → *Cryptography*.

KEYWORDS

multiparty computation; compilers; cryptography

ACM Reference Format:
Benjamin Levy, Benjamin Sherman, Lindsey Kennard, Ana L. Milanova, Muhammad Ishaq, and Vassilis Zikas. 2019. Scheduling and Vectorization for MPC. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, ?? pages. https://doi.org/10.1145/3319535.3339818

1 INTRODUCTION

- We define the scheduling problem for MPC. We present an analytical model to reason about cost of schedules and show that scheduling is NP-hard via a reduction to the shortest common supersequence problem.
- We present a compiler that takes an IMP-like high-level program and produces amortized (i.e., vectorized) low-level cryptographic code in the MOTION framework. Central contributions are 1) a novel compiler framework, 2) a vectorization algorithm that produces optimal schedules for a large number of MPC programs, and 3) reasoning over output arrays without Array SSA; we remove infeasible loop-carried dependences introduced by standard SSA to improve vectorization.
- We present an implementation and evaluation in the MOTION framework. ANA: *Fill in with final results. Mention benchmarks (standard + new ones + HyCC).*

2 OVERVIEW

2.1 Source

As a running example, consider Biometric matching, a standard MPC benchmark. Array `C` is the feature vector of `D` features that we wish to match and array `S` is the database of `N` vectors of size `D` that we match against. An intuitive implementation is as follows:

```
def biometric(C: shared[list[int]], D: int,
              S: shared[list[int]], N: int) ->
                      tuple[shared[int], shared[int]]:
    min_sum = 10000
    min_index = 0
    for i in range(N): #loop over database
        sum = 0
        for j in range(D): #loop over features
            d = S[i * D + j] - C[j] #i.e., d = S[i,j] - C[j]
            p = d * d
            sum = sum + p
        if sum < min_sum:
            min_sum = sum
            min_index = i
    return (min_sum, min_index)
```

Our compiler takes (essentially) standard IMP syntax. The programmer can write intuitive iterative programs as the one above. They annotate certain inputs and outputs as *shared*. Here the code iterates over the entries in the database and computes the sum of squares of the differences of individual features. The program returns the index `i` of the vector that gives the best match plus the corresponding sum of squares.

Our compiler imposes the following restrictions. We note that in some cases, the restrictions can be easily lifted and we plan to do so in future iterations of our compiler.

(1) The program contains arbitrarily nested loops, however, loop bounds are fixed: `0 <= i < N`. A standard restriction in MPC is that the bounds must be known at circuit-generation time.
(2) Arrays are one-dimensional. N-dimensional arrays are linearized and accessed in row-major order and at this

*This work was done in part while the author was at RPI.
†This work was done in part while the author was visiting UCLA and supported in part by DARPA and SPAWAR under contract N66001-15-C-4065 and by a SICSA Cyber Nexus Research Exchanges grant.

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3339818

point the programmer is responsible for linearization and access.

(3) Array subscrpts are plaintext values.

(4) Our compiler allows for output (write) arrays, however it restricts write access to *canonical writes* along the dimensions of the array. I.e., `A[i,j] = ...` where `i` and `j` loop over the two dimensions of `A` is allowed, but `A[i,j+2] = ...` is not allowed. Read access is arbitrary.

2.2 MPC Source and Cost of Schedule

The compiler generates an IR, MPC source:

```
1. min_sum!1 = 10000
2. min_index!1 = 0
3. for i in range(0, N!0):
4.    min_sum!2 = PHI(min_sum!1, min_sum!4)
5.    min_index!2 = PHI(min_index!1, min_index!4)
6.    sum!2 = 0
7.    for j in range(0, D!0):
8.      sum!3 = PHI(sum!2, sum!4)
9.      d!3 = (S!0[((i * D!0) + j)] - C!0[j]) // MPC
10.     p!3 = (d!3 * d!3) // MPC
11.     sum!4 = (sum!3 + p!3) // MPC
12.  !1!2 = (sum!3 < min_sum!2) // MPC
13.  min_sum!3 = sum!3
14.  min_index!3 = i
15.  min_sum!4 = MUX(!1!2, min_sum!3, min_sum!2) // MPC
16.  min_index!4 = MUX(!1!2, min_index!3, min_index!2) // MPC
17.!2!1 = (min_sum!2, min_index!2)
```

The compiler linearizes the source turning conditionals into MUX statements. The PHI nodes are remnants of the SSA IR; the compiler generates code that picks the correct value when producing MOTION output; the MOTION framework in turn linearizes loops when it generates the circuit.

We turn to our analytical model to compute the cost of this program. Assuming fixed cost β for a local MPC operation (essentially just ADD) and cost α for a remote MPC operation (e.g., MUX, CMP, and remaining operations), the cost of the iterative schedule will be $N * D * (2 * \alpha + \beta) + N * 3 * \alpha$.

A key contribution is the vectorizing transformation. We can compute all $N * D$ subtraction operations (line 9) in a single SIMD instruction; similarly we can compute all multiplication operations (line 10) in a single SIMD instruction. And while we cannot vectorize computation of the N individual sums, we can compute the N sums in parallel. Our compiler *automatically detects these opportunities and transforms the program.* It is standard that MPC researchers write vectorized versions of the Biometric program by hand; we are the first (to the best of our knowledge) to automatically transform an intuitive, iterative MPC program into an unintuitive vectorized one.

2.3 Vectorized MPC Source and Cost of Schedule

Our compiler produces the following vectorized program. (Note that this is still higher-level IR, Vectorized MPC Source. Our compiler turns this code into MOTION variables, loops and SIMD primitives, which MOTION then uses to generate the circuit.)

```
min_sum!1 = 10000
min_index!1 = 0
// S!0^ is same as S!0. C!0^ replicates C!0 N-times:
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))

sum!2 = [0,..,0]
// Computes all differences and all products "at once"
d!3[I,J] = SUB_SIMD(S!0^[I,J],C!0^[I,J])
p!3[I,J] = MUL_SIMD(d!3[I,J] * d!3[I,J])

for j in range(0, D!0):
  // sum!2[I], sum!3[I], sum!4[I] are vectors of size N
  // Computes N intermediate sums "at once"
  sum!3[I] = PHI(sum!2[I], sum!4[I])
  sum!4[I] = ADD_SIMD(sum!3[I],p!3[I,j])

min_index!3 = [0,1,...N!0-1]

for i in range(0, N!0):
  min_sum!2 = PHI(min_sum!1, min_sum!4)
  !1!2[i] = CMP(sum!3[i],min_sum!2)
  min_sum!4 = MUX(!1!2[i], sum!3[i], min_sum!2)

for i in range(0, N!0):
  min_index!2 = PHI(min_index!1, min_index!4)
  min_index!4 = MUX(!1!2[i], min_index!3[i], min_index!2)
!2!1 = (min_sum!2, min_index!2)
```

In MPC compilers a vectorized operation that computing M operations "at once" costs essentially the same (α or β) as an individual operation. We elaborate on these in the following section. Thus, the vectorized program costs $2 * \alpha + D * \beta + N * 3 * \alpha$. The first term in the sum corresponds to the vectorized subtraction and multiplication, the second term corresponds to the for loop on j and the third one corresponds to the remaining for loops on i. Clearly, $2 * \alpha + D * \beta + N * 3 * \alpha << N * D * (2 * \alpha + \beta) + N * 3 * \alpha$. Our experimental results illustrate this as well. ANA: *Add numbers.*

3 ANALYTICAL MODEL

3.1 Scheduling in MPC

For this treatment we make the following simplifying assumptions:

(1) All statements in the program execute using the same protocol (sharing). That is, there is no share conversion.

(2) There are two tiers of MPC instructions, local and remote. A local instruction (essentially just ADD) has cost β and a remote instruction (e.g., MUX, MUL, SHL, etc.) has cost α, where $\alpha >> \beta$. We assume that all remote instructions have the same cost.

(3) We assume infinite parallel capacity—i.e., a single MPC-instruction costs as much as N amortized instructions, namely α or β. This is a standard assumption in Cryptographic Parallel RAM. ABY presents empirical support for this assumption ANA: *Add citations. PRAM, ABY.*

(4) MPC instructions scheduled in parallel benefit from amortization *only if* they are the same instruction. Given our previous assumption, 2 MUL instructions scheduled in parallel benefit from amortization and cost α, however a MUL and a MUX instructions scheduled in parallel still cost 2α.

3.2 Problem Statement

ANA: *Ishaq? Basically, define sequential schedule, then define an equivalent parallel schedule. A parallel schedule is equivalent if it preserves def-use relations in sequential schedule, or in other words, schedules def ahead of the use. Problem is to minimize cost of Parallel schedule.*

ISHAQ: *TODO: make it consistent with the next section.*

At the lowest level, we have two types of MPC instructions 1) local/non-interactive instruction (i.e. ADD) and 2) remote/interactive instruction (i.e. MUL). Let the cost of a single ADDinstruction be given by a monotonically decreasing $f(n)$, where the argument n is the number of ADDinstructions being executed in parallel. Similarly the cost of a single MULis given by a monotonically decreasing function $g(n)$.

Given a serial schedule (a linear graph) of an MPC program i.e. a sequence of instructions $\mathcal{S} := (S_1, \ldots, S_n)$, where $S_i \in \{\text{ADD}, \text{MUL}\}, 1 \leq i \leq n$, and a def-use dependency graph $G(V, E)$ corresponding to $\mathcal{S}$, our task is to construct a parallel schedule (another linear graph) $\mathcal{P} := (P_1, \ldots, P_n)$ observing the following conditions:

(1) Multiple, not necessarily continuous instructions of the same type (i.e. either ADDor MUL) from $\mathcal{S}$ can be grouped into a single P_i However, all such instructions must be of the same type (either ADDor MUL).
(2) Def-use dependencies of the graph $G(V, E)$ are should be preserved i.e. if instructions $S_i, S_j, i < j$ are a def-use (an edge exists from S_i to S_j in G), then they can only be mapped to $P_{i'}, P_{j'}, i' < j'$.

Our goal is to construct minimize the height of the graph $\mathcal{P}$. Indeed, a graph $\mathcal{P}$ with minimum height will maximize parallelization.

Correctness. Correctness of $\mathcal{P}$ is guaranteed by definition. Since def-use dependencies are preserved, the function (being computed) remains the same.

Cost Comparison. For the sequential schedule $\mathcal{S}$ consisting of L local and R remote instructions, the total cost is $cost(\mathcal{S}) = L \cdot f(1) + R \cdot g(1)$. In the extreme case where all L and all R instructions can be parallelized, the cost of $\mathcal{P}$ is $cost(\mathcal{P}) = L \cdot f(L) + R \cdot g(R)$. Since both f and g are monotonically decreasing, $cost(\mathcal{P}) < cost(\mathcal{S})$. Cost of all other parallel schedules lies between the extremes of $cost(\mathcal{S})$ and $cost(\mathcal{P})$.

Note that we use an MPC-Source control flow graph (CFG) $G'(V', E')$ along with def-use graph $G(V, E)$ to construct $\mathcal{P}$. We consider a linearized MPC schedule $\mathcal{S}$ above for ease of exposition. The argument becomes slightly more involved when dealing with a graph G' that may contain cycles.

3.3 Scheduling is NP-hard

ISHAQ: *TODO: need make amortized cost a function (like in the problem statement above), it is proving to be tricky.*

We consider two operations, call them A and M. A and M are two abstract MPC instruction, but as an example, A stands for the ADD MPC instruction and M stands for the MUL instruction. Each instruction in the program is either an A-instruction or an M-instruction. In order to benefit from parallelization/amortization, we must schedule two or more A-instructions in the same parallel node (or two or more M-instructions in the same parallel node). We also assume that scheduling A-instructions in parallel with M-instruction does not benefit from amortization[1]. It incurs the exact same cost as scheduling the A-instructions in a node P_A, scheduling the M-instructions in a node P_M, and having P_A precede P_M in the parallel schedule.

We use the following cost model:

(1) A costs α units and M costs β unites.
(2) There is unlimited bandwidth i.e. a single A-instruction (or M-instruction) costs as much as N amortized A-instructions (or M-instructions), concretely either α unites or β units.

Consider a loop body that consists of n sequences: S_1, ... S_n of A and M instructions. More precisely, the loop body is such that its instructions can be grouped into such sequences. S_1, ... S_n can execute in parallel, however, all instructions within a sequence must execute sequentially. For example, consider the three sequences (the right arrow indicates a *dependence*, meaning that the source node must execute before the target node):

(1) $A \to M \to A$
(2) $A \to A \to A$
(3) $M \to A \to M$

A *schedule* $P : P_1 \to P_2 \cdots \to P_k$ is such that for each sequence S_i in the set, if $S_i[j]$ precedes $S_i[j']$ in S_i then $S_i[j]$ is scheduled in node P_ℓ, $S_i[j']$ is scheduled in node $P_{\ell'}$, and P_ℓ precedes $P_{l\ell'}$ in P.

The cost of a schedule P is

$$cost(P) = \sum_{i=1}^{k} cost(P_i) \tag{1}$$

where $cost(P_i) = \alpha$ if P_i consists of A-instructions only, β if P_i consists of M-instructions only, and $cost(P_i) = \alpha + \beta$ if P_i mixes A-instructions and M-instructions.

The problem is to find a schedule P with *minimal cost*. For example, a schedule with minimal cost for the sequences above is

$$A(1), A(2) \to M(1), A(2), M(3) \to A(1), A(2), A(3) \to M(3)$$

(The parentheses above indicate the sequence where the instruction comes from: (1), (2), or (3).) The cost of this schedule is $3\alpha + 2\beta$.

[1] this is not strictly true, but assuming it, e.g. as in [**Ishaq2019**, **Demmler2015ABYA**, **Mohassel2018**], helps with the exposition.

The problem of finding a schedule P with a minimal $cost(P)$ for a given loop body has been shown to be an NP-Hard problem, as it can be reduced to the problem of finding a *shortest common supersequence*, a known NP-Hard problem[**Maier1978**],[**Vazirani2010**]. The shortest common supersequence problem is as follows: *given two or more sequences find the the shortest sequence that contains all of the original sequences.* This can be solved in $O(n^k)$ time, where n is the cardinality of the longest sequence and k is the number of sequences. For our problem n is the maximum length of a node and k is the number of total number of nodes.

To see the reduction, suppose P is a schedule with minimal cost (computed by a black-box algorithm). We can derive a schedule P' with the same cost as P, by mapping each mixed node $P_i \in P$ to two consecutive nodes in P': an A-instruction node followed by an M-instruction node. Clearly, P', which now is a sequence of A's and M's, is a supersequence of each sequence S_i, i.e., P' is a common supersequence of $S_1 \ldots S_n$. It is also a shortest common supersequence. To see this, let X and Y denote, respectively, the number of A and M nodes in P'. The cost of P', and P, is $X \cdot \alpha + Y \cdot \beta$. Now suppose, there exists a shorter common supersequence, P'' that consists of X' nodes of type A-instructions Y' nodes of type M-instructions. Since P'' is shorter than P', therefore $X' + Y' < X + Y$, and $X' \cdot \alpha + Y' \cdot \beta < X \cdot \alpha + Y \cdot \beta$ i.e. $cost(P'') < cost(P')$. But $cost(P') = cost(P)$ and $cost(P)$ is the optimal cost. Therefore $cost(P'') < cost(P')$ is contradiction and no such P'' exists.

4 COMPILER FRAMEWORK

Fig. **??** presents an overview of our compiler. In this section, we describe several of the phases of the compiler. Sections §**??** and §**??** describe vectorization and divide-and-conquer. We write i, j, k to denote the loop nest: i is the outermost loop, j, is immediately nested in i, and so on until k and we use I, J, K to denote the corresponding upper bounds. For simplicity, we write $A[i, j, k]$ to denote canonical access to an array element. In the program, canonical access is achieved via the standard row-major order formula: $(J * K) * i + K * j + k$. To simplify the presentation we describe our algorithms in terms of three-element tuples i, j, k. All discussion generalizes to arbitrarily large loop nests.

4.1 Semantic Analysis

Our compiler performs the following semantic analysis steps:

(1) **Parsing**: Use Python's `ast` module to parse the input source code to a Python AST.
(2) **Syntax checking**: Ensure that the AST matches a restricted subset that our compiler supports. This step outputs an instance of the `restricted_ast.Function` class, which represents our restricted subset of the Python AST.
(3) **3-address CFG conversion**: TODO
(4) **SSA conversion**: Convert the 3-address CFG to SSA with Cytron's algorithm.

4.2 MUX Nodes and Pseudo ϕ-nodes

ANA: *Benjamin?*

ANA: *I think we need to add more here. Conversion from SSA to MPC Source is not so trivial.*

A pseudo ϕ-node $X_1 = \phi(X_0, X_2)$ in a loop header is evaluated during circuit generation. If it is the 0-th iteration, then the ϕ-node evaluates to X_0, otherwise, it evaluates to X_2.

4.3 Dependence Analysis

4.3.1 Def-use Edges

The dependence graph has the following def-use edges:

- same-level edge $X \to Y$ where X and Y are in the same loop nest, say i, j, k. E.g., the def-use edge from `d = S[i,j] - C[j]` to `p = d*d` in the Biometric MPC-source is a same-level edge. A same-level edge can be a back-edge in which case a ϕ node is the target of the edge. E.g., $\texttt{min}_1 = MUX(\texttt{c}, \texttt{sum}_1, \texttt{min}_1)$ to $\texttt{min}_0 = \phi(\texttt{min}_1, 10000)$ in Biometric is a same-level back-edge.
- outer-to-inner $X \to Y$ where X is in an outer loop nest, say i, and Y is in an inner one, say i, j, k.
- inner-to-outer $X \to Y$ where X is a *phi*-node in an inner loop nest, i, j, k, and Y is in the enclosing loop nest i, j. E.g. $\texttt{sum}_0 = \phi(\texttt{sum}_1, 0)$ to $\texttt{c} = CMP(\texttt{sum}_0, \texttt{min}_0)$ is an inner-to-outer edge. An inner-to-outer edge can be a back-edge as well in which case both X and Y are phi-nodes with the source X in a loop nested into Y's loop (not necessarily immediately).
- mixed forward edge $X \to Y$. X is in some loop i, j, k and Y is in a loop nested into i, j, k'. We transform mixed forward edges as follows. Let `x` be the variable defined at X. We add a variable and assignment $\texttt{x}' = \texttt{x}$ immediately after the i, j, k loop. Then we replace the use of `x` at Y with $\texttt{x}'$. This transforms a mixed forward edge into an "inner-to-outer" forward edge followed by an outer-to-inner forward edge. Thus, Basic Vectorization handles one of "same-level", "inner-to-outer", or "outer-to-inner" def-use edges.

4.3.2 Closures

We define $closure(n)$ where n is a phi-node. Intuitively, it computes the set of nodes (i.e., statements) that form a dependence cycle with n. The closure of n is defined as follows:

- n is in $closure(n)$
- X is in $closure(n)$ if there is a same-level path from n to X, and $X \to n$ is a same-level back-edge.
- Y is in $closure(n)$ if there is a same-level path from n to Y and there is a same-level path from Y to some X in $closure(n)$.

4.4 Taint Analysis

We require that all inputs are marked as either shared or plaintext. We then determine if intermediate variables are shared through taint analysis with "taintedness" referring to the shared attribute. Specifically, our compiler follows the following rules:

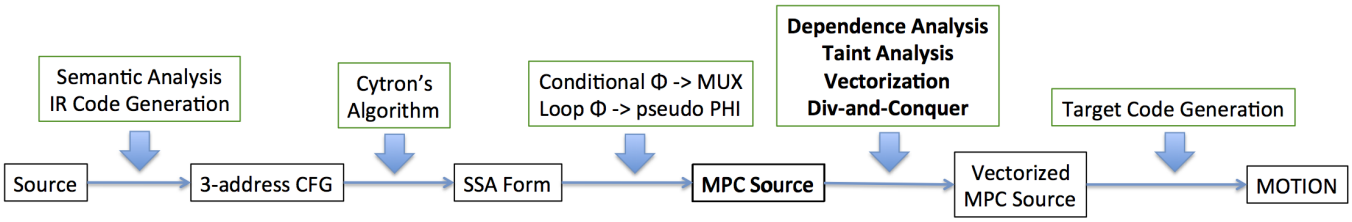

Figure 1: Compiler Framework.

- If any variable on the right-hand side of an assignment is shared, then the assigned variable is shared
- If all variables on the right-hand side of an assignment are plaintext, then the assigned variable is plaintext
- Loop counters are always plaintext
- Any variables which cannot be determined as shared or plaintext via the above rules are plaintext

The first two rules are standard for taint analysis, and the third rule follows from the MPC problem statement. BEN: *Is the explanation for the third rule correct?* The final rule is needed to handle cycles of plaintext values. For example, in the below snippet `sum!2` and `sum!3` form a dependency cycle and cannot be marked as plaintext through simple taint analysis:

```
plaintext_array = [0, 1, 2, ...]
sum!1 = 0
for i in range(0, N):
    sum!2 = PHI(sum!1, sum!3)
    sum!3 = sum!2 + plaintext_array[i]
```

BEN: *I think the above example is unnecessary and could be replaced by an explanation of how "untainted" variables are implicitly plaintext, but I couldn't think of a way to phrase that.*

When converting to MOTION code, any plaintext value used in the right-hand side of a shared assignment is implicitly converted to a shared value for that expression. BEN: *Is this necessary to include?*

5 VECTORIZATION

An important component of our algorithm is the "lifting" of scalars to the corresponding loop dimensionality. For example, `d = S[i * D + j] - C[j]` equiv. to `d = S[i,j] - C[j]`, which gave rise to `N*D` subtraction operations in the sequential schedule, is lifted. The argument arrays `S` and `C` are lifted and the scaler `d` is lifted: `d[i,j] = S[i,j] - C[i,j]`. The algorithm then detects that the statement can be vectorized.

There are three kinds of arrays (for now all kept internally as one-dimensional arrays, but that's under discussion).

- Scalars: These are scalar variables we lift into arrays for the purposes of vectorization. For those, all writes are canonical writes and all reads are canonical reads. We will apply raise dimension when a scalar is used in an inner loop (e.g., `sum0` in line 6 of the MPC source code will be raised to a 1-dimensional array since `sum0` is used in the inner j-loop). Drop dimension applies as well; this happens when a scalar written in an inner loop is used outside of the loop (e.g., `sum0` for which the lifted inner loop computes `D` values, but the outer loop only needs the last one.)
- Read-only input arrays: Read-only inputs. There are NO writes, while we may have non-canonical reads, $f(i,j,k)$. Phase 1 of Basic vectorization will add raise dimension operation at the beginning of the function to lift these arrays, and raise dimension may *reshape* arrays. If there are multiple "views" of the input array, there would be multiple raise dimension statements to create each one of these views. The invariant is that at reads in loops, the reads of "views" of the original input array are canonical. Only raise dimension applies to these arrays, and only in the beginning of the program. For example, 1-dimensional array `C` is lifted into 2-dimensional array `N x D` by copying the row `N` times.
- Read-write output arrays: Writes are canonical (by restriction) but reads can be non-canonical. Dependence analysis limits vectorization when non-canonical read access refers to array writes in previous iterations, thus creating loop-carried dependences. We may apply both raise and drop dimension, however, they respect the fixed dimensionality of the output array. The array cannot be raised to a dimension lower than its canonical (fixed) dimensionality and it cannot be dropped lower. In addition, non-canonical reads may require lifting (i.e., reshaping) of the array after the most recent write, rather than in the beginning of the program, in order to reduce a non-canonical read to a canonical one.

In the sections below we detail the *raise_dim* (raise dimension) and *drop_dim* (drop dimension) operations, followed by our vectorization algorithm.

5.1 Raise Dimension and Drop Dimension

There are two conceptual versions of *raise_dim*. One applies on read-only input arrays and reshapes those arrays when necessary to ensure canonical read access in the corresponding loop. The signature of *raise_dim* is as follows. It takes the original array `C`, the access pattern function $f(i,j,k)$ in loop nest i,j,k and the loop bounds `((i:I),(j:J),(k:K))`:

$$raise\_dim(\texttt{A}, f(i,j,k), \texttt{((i:I),(j:J),(k:K))})$$

It produces a new 3-dimensional array `A'` by iterating over i,j,k and setting each element of `A'` as follows:

```
A'[i,j,k] = A[f(i,j,k)]
```

The end result is that uses of `[A[f(i,jk)]` in loop nest i, j, k are replaced with canonical read-accesses to `A'[i,j,k]` that can be vectorized. In the running Biometric example, `C'` = *raise_dim*`(C, j, (i:N,j:D))` lifts the 1-dimensional array `C` into a 2-dimensional array. The i, j loop now accesses `C'` in the canonical way, `C'[i,j]`. Similarly, `S'` = *raise_dim*`(S, i*D+j, (i:N,j:D))` tries to lift `S`, but the operation turns into a no-op because `S` is already a 2-dimensional array and the read access is canonical.

The other version of *raise_dim* applies on scalars and read-write arrays. It lifts a lower-dimension array into a higher-dimension for access in a nested loop. Here `A` is an i array and raise dimension adds two additional dimensions:

$$raise\_dim(\texttt{A}, \texttt{(j:J,k:K)})$$

This version is reduced to the above version by adding the access pattern function, which is just i:

$$raise\_dim(\texttt{A}, i, \texttt{(j:J,k:K)})$$

The corresponding *drop_dim* is carried out when an array written in an inner loop is used in an enclosing loop. It takes a higher dimensional array, say i, j, k and removes trailing dimensions, say j, k:

$$drop\_dim(\texttt{A}, \texttt{(j:J,k:K)})$$

It iterates over i and takes the result at the maximal index of j and k, i.e., the result at the last iterations of j and k:

```
A'[i] = A[i,J-1,K-1]
```

5.2 Basic Vectorization

{ Phase 1: Raise dimension of scalar variables to corresponding loop nest. We can traverse stmts linearly in MPC-source. }
for each MPC *stmt* : $X = Op(Y_1, Y_2)$ in loop i, j, k **do**
  for each argument Y_n **do**
    case def-use edge *stmt'*(def of Y_n) → *stmt*(def of X) of
      same-level: Y'_n is Y_n
      outer-to-inner: add $Y'_n[i, j, k] = raise\_dim(Y_n)$ at *stmt'*
      inner-to-outer: add $Y'_n[i, j, k] = drop\_dim(Y_n)$ at *stmt*
  end for
  { Optimistically vectorize all. I means vectorized dimension. }
  change to $X[I, J, K] = Op(Y'_1[I, J, K], Y'_2[I, J, K])$
end for
{ Phase 2: Recreating FOR loops for cycles; vectorizable statements hoisted up. }
for each dimension d from highest to 0 **do**
  for each ϕ-node n in loop $i_1, ..., i_d$ **do**
    compute *closure*(n)
  end for
  {cl_1 and cl_2 intersect if they have common statement or update same array; "intersect" definition can be expanded}
  while there are closure cl_1 and cl_2 that intersect **do**
    merge cl_1 and cl_2
  end while
  for each closure cl (after merge) **do**
    create FOR $i_d = 0; ...$ loop
    add ϕ-nodes in cl to header block
    add target-less ϕ-node for A if cl updates array A
    add statements in cl to loop body in some order of dependences
    { Dimension is not vectorizable: }
    change I_d to i_d in all statements in loop
    treat FOR loop as monolith node: some def-use edges become same-level.
  end for
  for each target-less ϕ-node $A_1 = \phi(A_0, A_k)$ **do**
    in vectorizable stmts, replace use of A_1 with A_0
    discard ϕ-node if not used in any cl
  end for
end for
{ Phase 3: Remove unnecessary dimensionality. }
{ A dimension i is dead on exit from stmt $X[...i...] = ...$ if all def-uses with targets outside of the enclosing FOR $i = 0...$ MOTION loop end at target (use) $X' = drop\_dim(X, i)$. }

for each stmt and dimension $X[...i...] = ...$ **do**
  if i is a dead dimension on exit from stmt $X[...i...] = ...$, remove i from X (all defs and uses)
end for
{ Now clean up drop_dim and raise_dim }
for each $X' = drop\_dim(X, i)$ **do**
  replace with $X' = X$ if i is dead in X.
end for
do (1) (extended) constant propagation, (2) copy propagation and (3) dead code elimination to get rid of redundant variables and raise and drop dimension statements
{ Phase 4: }
add SIMD for simdified dimensions

5.3 Example: Biometric

We start from Benjamin's code with linear loops (MPC Source):

```
min_sum!1 = 10000
min_index!1 = 0
for i in range(0, N!0):
   min_sum!2 = PHI(min_sum!1, min_sum!4)
   min_index!2 = PHI(min_index!1, min_index!4)
   sum!2 = 0
   for j in range(0, D!0):
     sum!3 = PHI(sum!2, sum!4)
     d!3 = (S!0[((i * D!0) + j)] - C!0[j])
     p!3 = (d!3 * d!3)
     sum!4 = (sum!3 + p!3)
   !1!2 = (sum!3 < min_sum!2)
   min_sum!3 = sum!3
   min_index!3 = i
   min_sum!4 = MUX(!1!2, min_sum!3, min_sum!2)
   min_index!4 = MUX(!1!2, min_index!3, min_index!2)
!2!1 = (min_sum!2, min_index!2)
```

5.3.1 Phase 1 of Basic Vectorization

The transformation preserves the dependence edges. It raises the dimensions of scalars and optimistically vectorizes all operations. The next phase discovers loop-carried dependences and removes affected vectorization.

In the code below, all initializations (e.g., `min_sum!3 = i`), operations, and PHI nodes are *implicitly vectorized*. *raise_dim* and *drop_dim* statements have slightly different interpretation.

The example illustrates the two different versions of *raise_dim*. `C!0' = raise_dim(C!0, j, (i:N!0,j:D!0))` reshapes the read-only input array, while `sum!3' = drop_dim(sum!3)` removes the j dimension of `sum!3`.

```
min_sum!1 = 10000
min_sum!1^ = raise_dim(min_sum!1, (i:N!0))
min_index!1 = 0
min_index!1^ = raise_dim(min_index!1, (i:N!0))
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))
for i in range(0, N!0):
   min_sum!2 = PHI(min_sum!1^, min_sum!4)
   min_index!2 = PHI(min_index!1^, min_index!4)
   sum!2 = 0 // Will lift, when hoisted
   sum!2^ = raise_dim(sum!2, (j:D!0)) // Special form?
   for j in range(0, D!0):
      sum!3 = PHI(sum!2^, sum!4)
      d!3 = S!0^ - C!0^
      p!3 = (d!3 * d!3)
      sum!4 = (sum!3 + p!3)
   sum!3^ = drop_dim(sum!3)
   !1!2 = (sum!3^ < min_sum!2)
   min_sum!3 = sum!3^
   min_index!3 = i  // Same-level, will lift when hoisted
   min_sum!4 = MUX(!1!2, min_sum!3, min_sum!2)
   min_index!4 = MUX(!1!2, min_index!3, min_index!2)
min_sum!2^ = drop_dim(min_sum!2)
min_index!2^ = drop_dim(min_index!2)
!2!1 = (min_sum!2^, min_index!2^)
```

5.3.2 Phase 2 of Basic Vectorization

This phase analyzes statements from the innermost loop to the outermost. The key point is to discover loop-carried dependencies and re-introduce loops whenever dependencies make this necessary.

Starting at the inner phi-node `sum!3 = PHI(sum!2, sum!4)`, the algorithm first computes its closure. The closure amounts to the phi-node itself and the addition node `sum!4 = (sum!3 + p!3)`, accounting for the loop-carried dependency of the computation of `sum`. The algorithm replaces this closure with a FOR loop on j removing vectoriaztion on j. Note that the SUB and MUL computations remain outside of the loop as they do not depend on phi-nodes that are part of cycles. The dependences are from `p!3[I,J] = (d!3[I,J] * d!3[I,J])` to the monolithic FOR loop and from the FOR loop to `sum!3≙ drop_dim(sum!3)`. (Lower case index, e.g., `i`, indicates non-vectorized dimension, while uppercase index, e.g., `I` indicates vectorized dimension.)

After processing inner loop code becomes:

```
min_sum!1 = 10000
min_sum!1^ = raise_dim(min_sum!1, (i:N!0))
min_index!1 = 0
min_index!1^ = raise_dim(min_index!1, (i:N!0))
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))
for i in range(0, N!0):
  min_sum!2[I] = PHI(min_sum!1^[I], min_sum!4[I])
  min_index!2[I] = PHI(min_index!1^[I], min_index!4[I])
  sum!2 = [0,..,0]
  sum!2^ = raise_dim(sum!2, (j:D!0))
  d!3[I,J] = S!0^[I,J] - C!0^[I,J]
  p!3[I,J] = (d!3[I,J] * d!3[I,J])
  for j in range(0, D!0):
    sum!3[I,j] = PHI(sum!2^[I,j], sum!4[I,j-1])
    sum!4[I,j] = (sum!3[I,j] + p!3[I,j])
  sum!3^ = drop_dim(sum!3)
  !1!2[I] = (sum!3^[I] < min_sum!2[I])
  min_sum!3 = sum!3^
  min_index!3 = i
  min_sum!4[I] = MUX(!1!2[I], min_sum!3[I], min_sum!2[I])
   min_index!4[I] = MUX(!1!2[I], min_index!3[I], min_index!2[I])
min_sum!2^ = drop_dim(min_sum!2)
min_index!2^ = drop_dim(min_index!2)
!2!1 = (min_sum!2^, min_index!2^)
```

When processing the outer loop two closures arise, one for `min_sum!2[I] = PHI(...)` and one for `min_index!2[I] = PHI(...)`. Since the two closures *do not* intersect, we have two distinct FOR-loops on i:

```
min_sum!1 = 10000
min_sum!1^ = raise_dim(min_sum!1, (i:N!0))
min_index!1 = 0
min_index!1^ = raise_dim(min_index!1, (i:N!0))
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))

sum!2 = [0,..,0]
sum!2^ = raise_dim(sum!2, (j:D!0))
d!3[I,J] = S!0^[I,J] - C!0^[I,J]
p!3[I,J] = (d!3[I,J] * d!3[I,J])

for j in range(0, D!0):
  sum!3[I,j] = PHI(sum!2^[I,j], sum!4[I,j-1])
  sum!4[I,j] = (sum!3[I,j] + p!3[I,j])

sum!3^ = drop_dim(sum!3)
min_index!3 = [0,1,2,...N!0-1] // or min_index!3 = [i, (i:N!0)]
min_sum!3 = sum!3^

for i in range(0, N!0):
  min_sum!2[i] = PHI(min_sum!1^[i], min_sum!4[i-1])
  !1!2[i] = (sum!3^[i] < min_sum!2[i])
  min_sum!4[i] = MUX(!1!2[i], min_sum!3[i], min_sum!2[i])

for i in range(0, N!0):
  min_index!2[i] = PHI(min_index!1^[i], min_index!4[i-1])
   min_index!4[i] = MUX(!1!2[i], min_index!3[i], min_index!2[i])

min_sum!2^ = drop_dim(min_sum!2)
min_index!2^ = drop_dim(min_index!2)
!2!1 = (min_sum!2^, min_index!2^)
```

5.3.3 Phase 3 of Basic Vectorization

This phase removes redundant dimensionality. It starts by removing redundant dimensions in MOTION loops followed by removal of redundant drop dimension statements. It then does (extended) constant propagation to "bypass" raise statements, followed by copy propagation and dead code elimination.

The code becomes closer to what we started with:

```
min_sum!1 = 10000
min_index!1 = 0
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))

sum!2 = [0,..,0]
d!3[I,J] = S!0^[I,J] - C!0^[I,J]
p!3[I,J] = (d!3[I,J] * d!3[I,J])

// j is redundant for sum!3 and sum!4
for j in range(0, D!0):
  sum!3[I] = PHI(sum!2[I], sum!4[I])
  sum!4[I] = (sum!3[I] + p!3[I,j])

// drop_dim is redundant, removing
// then copy propagation and dead code elimination
min_index!3 = [0,1,2,...N!0-1] // or min_index!3 = [i, (i:N!0)]

// i is redundant for min_sum!2, min_sum!4 but not for !12![i]!
for i in range(0, N!0):
  min_sum!2 = PHI(min_sum!1, min_sum!4)
  !1!2[i] = (sum!3[i] < min_sum!2)
  min_sum!4 = MUX(!1!2[i], sum!3[i], min_sum!2)

// same, i is redundant for min_index!2, min_index!4
for i in range(0, N!0):
  min_index!2 = PHI(min_index!1, min_index!4)
   min_index!4 = MUX(!1!2[i], min_index!3[i], min_index!2)

// drop_dim becomes redundant
!2!1 = (min_sum!2, min_index!2)
```

5.3.4 Phase 4 of Basic Vectorization

And this phase adds SIMD operations:

```
min_sum!1 = 10000
min_index!1 = 0
S!0^ = raise_dim(S!0, ((i * D!0) + j), (i:N!0,j:D!0))
C!0^ = raise_dim(C!0, j, (i:N!0,j:D!0))

sum!2 = [0,..,0]
d!3[I,J] = SUB_SIMD(S!0^[I,J],C!0^[I,J])
p!3[I,J] = MUL_SIMD(d!3[I,J] * d!3[I,J])

for j in range(0, D!0):
  // I dim is a noop. sum is already a one-dimensional vector
  sum!3[I] = PHI(sum!2[I], sum!4[I])
  sum!4[I] = ADD_SIMD(sum!3[I],p!3[I,j])

min_index!3 = [0,1,...N!0-1]

for i in range(0, N!0):
  min_sum!2 = PHI(min_sum!1, min_sum!4)
  !1!2[i] = CMP(sum!3[i],min_sum!2)
  min_sum!4 = MUX(!1!2[i], sum!3[i], min_sum!2)
```

$s ::= s; s$	*sequence*
$\mid \mathsf{x}[i,J,k] = \mathsf{y}[i,J,k]\ op\_SIMD\ \mathsf{z}[i,J,k]$	*operation*
$\mid \mathsf{x}[i,J,k] = \mathtt{PHI}(\mathsf{x}_1[i,J,k], \mathsf{x}_2[i,J,k-1])$	*phi node*
$\mid \mathsf{x}[i,J,k] = raise\_dim(\mathsf{x}'[i], (J:J,k:K))$	*raise dimension*(s)
$\mid \mathsf{x}[i,J] = drop\_dim(\mathsf{x}'[i,J,k], k)$	*drop dimension*(s)
$\mid \mathsf{x} = \mathsf{y}$	*propagation*
$\mid \mathtt{FOR}\ 0 \le i < I : s$	*loop*

Figure 2: MPC Source Syntax

```
for i in range(0, N!0):
  min_index!2 = PHI(min_index!1, min_index!4)
  min_index!4 = MUX(!1!2[i], min_index!3[i], min_index!2)

!2!1 = (min_sum!2, min_index!2)
```

5.4 Correctness Argument

We build a correctness argument that loosely follows the theory of Abstract Interpretation. We define the syntax of MPC Source programs. The domain of MPC Source programs expressible in the syntax (with certain semantic restrictions) is the abstract domain A. We then define the *linearization* of an MPC Source program as an interpretation over the syntax. The linearization, which is a *schedule*, is the concrete domain C. Since we reason over def-use graphs in A we define a partial order relation over elements of A in terms of def-use relations. We define a partial order over elements of C as well, in terms of def-use relations in the concrete domain C. We prove two theorems that state (informally) that the schedule corresponding to the original program computes the same result as the schedule corresponding to the vectorized program.

MPC Source Syntax. Fig. **??** defines the syntax for our intermediate representation, MPC Source. There are semantics restrictions over the syntax as well: a variable $\mathsf{x}[i,j,k]$ is a 3-dimensional array $(i:I, j:J, k:K)$ and also, a statement $\mathsf{x}[i,J,k] = ...$ is enclosed in loops over i and k as shown below. Thus, i and k are in scope.

```
FOR 0 <= i < I:
    ...
    FOR 0 <= k < K:
        x[i,J,k] = ...
```

Statements *operation*, *phi*, *raise dimension*(s), *drop dimension*(s) are *base* statements, and *sequence*, *loop* are *compound statements*.

Linearization. Linearization is the concretization operation, which, as we mentioned earlier computes a schedule. The concretization function $\gamma : A \to C$ is defined as an interpretation of MPC Source syntax, as it is standard. The concretization of each one of the base statements is as follows:

$$\gamma(\mathsf{x}[i,J,k] = op\_SIMD(\mathsf{y}[i,J,k], \mathsf{z}[i,J,k])) =$$
$$\mathsf{x}[i,0,k] = \mathsf{y}[i,0,k]\ op\ \mathsf{z}[i,0,k]\ \ ||$$
$$\mathsf{x}[i,1,k] = \mathsf{y}[i,1,k]\ op\ \mathsf{z}[i,1,k]\ \ ||\ ...\ ||$$
$$\mathsf{x}[i,I-1,k] = \mathsf{y}[i,I-1,k]\ op\ \mathsf{z}[i,I-1,k]$$

meaning that the vectorized dimension(s) are expanded into parallel statements. —— introduces SIMD (parallel) execution.

The concretization of the FOR statement is as follows:

$$\gamma(\texttt{FOR}\ 0 \le i < I : s) = \gamma(s)[0/i]\ ;\ \gamma(s)[1/i]\ ;\ ...\gamma(s)[I-1/i]$$

γ simply unrolls the loop substituting i with 0, 1, etc. Here ; denotes sequential execution.

Partial Orders. For each MPC Source program a we compute the def-use edges in the standard way: if base statement $s1 \in a$ defines variable x, e.g., $x[i,j,k] = ...$, and base statement $s2 \in a$ uses x, e.g., $... = ...x[i,j,k]$ and there is a path in the trivial CFG from $s1$ to $s2$, then there is a def-use edge from $s1$ to $s2$. We extend the dimensionality of a statement into $s1[i,j,k]$ where $s1[i,j,k]$ inherits the dimensionality of the left-hand-side of the assignment.

Let a_0, a_1 be two MPC Source programs in A. Two base statements, $s_0 \in a_0$ and $s_1 \in a_1$ are *same*, written $s_0 \equiv s_1$ if they are of the same operation and they operate on the same variables: same variable name and same dimensionality. Recall that dimensions in MPC Source are either iterative, lower case, or vectorized, upper case. Two statements are same even if one operates on an iterative dimension and the other one operates on a vectorized one, e.g., $s_0[i,j,k] \equiv s_1[I,j,K]$.

Definition 1. *Let $a_0, a_1 \in A$. We say that $a_0 \le a_1$ iff for every def-use edge $s1 \to s2$ in a_0 there is an edge $s1' \to s2'$ where $s1 \equiv s1'$, $s2 \equiv s2'$ and the two edges of either both forward or both backward.*

The def-use edges in the concrete schedule are as expected. There is a def use edge from statement $s1$ that defines $x[\underline{i},\underline{j},\underline{k}]$ to statement $s2$ that uses $x[\underline{i},\underline{j},\underline{k}]$ if $s1$ is scheduled ahead of $s2$ in the linear schedule. We note that the underlined indices, e.g., $\underline{i}$, refer to fixed values, not iterative or vectorized dimensions since in the concrete schedule all induction variables are expanded. E.g., there is a def-use edge from the statement that defines $x[0,1,2]$ and a statement that uses $x[0,1,2]$.

Theorems.

Theorem 1. $a_0 \le a_1 \Rightarrow \gamma(a_0) \subseteq \gamma(a_1)$.

Theorem 2. *Let a_0 be the iterative MPC Source and let a_1 be the vectorized MPC Source computed by the vectorization algorithm. We have that $a_0 \le a_1$.*

Ana: *Write the proof sketch and final argument, etc.*

5.5 Extension of Basic Vectorization with Array Writes

5.5.1 Removal of Infeasible Edges

Array writes limit vectorization as they sometimes introduce infeasible loop-carried dependencies. Consider the following example: Ana: *Have to add citation to Aiken's paper*

```
for i in range(N):
  A[i] = B[i] + 10;
  B[i] = A[i] * D[i-1];
  C[i] = A[i] * D[i-1];
  D[i] = B[i] * C[i];
```

In Cytron's SSA this code (roughly) translates into

```
for i in range(N):
1. A_1 = PHI(A_0,A_2)
2. B_1 = PHI(B_0,B_2)
3. C_1 = PHI(C_0,C_2)
4. D_1 = PHI(D_0,D_2)
5. A_2 = update(A_1, i, B_1[i] + 10);
6. B_2 = update(B_1, i, A_2[i] * D_1[i-1]);
7. C_2 = update(C_1, i, A_2[i] * D_1[i-1]);
8. D_2 = update(D_1, i, B_2[i] * C_2[i]);
```

There is a cycle around `B_1 = PHI(B_0,B_2)` that includes statement `A_1 = update(A_0, i, B_1[i] + 10)` and that statement won't be vectorized even though in fact there is no loop-carried dependency from the write of `B_1[i]` at 6 to the read of `... = B_1[i]` at 8.

The following algorithm removes certain infeasible loop-carried dependencies that are due to array writes. Consider a loop with index $0 \le j < J$ nested at i,j,k. Here i represents the enclosing loops of j and k represents the enclosed loops in j.

for each array `A` written in loop j **do**
  { including enclosed loops in j }
  dep = False
  for each pair def: $\texttt{A}_m[f(i,j,k)] = ...$, and use: $... = \texttt{A}_n[f'(i,j,k)]$ in loop j **do**
    if $\exists \underline{i},\underline{j},\underline{j'},\underline{k},\underline{k'}$, s.t. $0 \le \underline{i} < I$, $0 \le \underline{j},\underline{j'} < J$, $0 \le \underline{k},\underline{k'} < K$, $\underline{j} < \underline{j'}$, and $f(\underline{i},\underline{j},\underline{k}) = f'(\underline{i},\underline{j'},\underline{k'})$ **then**
      dep = True
    end if
  end for
  if dep == False **then**
    remove back edge into `A`'s ϕ-node in loop j.
  end if
end for

Consider a loop j enclosed in some fixed $\underline{i}$. Only if an update (definition) $\texttt{A}_m[f(i,j,k)] = ...$ at some iteration $\underline{j}$ references the *same* array element as a use $... = \texttt{A}_n[f'(i,j,k)]$ at some later iteration $\underline{j'}$, we may have a loop-carried dependence for `A` due to this def-use pair. (In contrast, Cytron's algorithm inserts a loop-carried dependency every time there is an array update.) The algorithm above examines all def-use pairs in loop j, including defs and uses in nested loops, searching for values $\underline{i},\underline{j},\underline{j'},\underline{k},\underline{k'}$ that satisfy $f(\underline{i},\underline{j},\underline{k}) = f'(\underline{i},\underline{j'},\underline{k'})$. If such values exist for some def-use pair, then there is a potential loop-carried dependence on `A`; otherwise there is not and we can remove the spurious backward edge thus "freeing up" statements for vectorization.

Consider the earlier example. There is a single loop, i. Clearly, there is no pair $\underline{i}$ and $\underline{i'}$, where $\underline{i} < \underline{i'}$ that make $\underline{i} = \underline{i'}$ due to the def-use pairs of `A` 5-6 and 5-7. Therefore, we remove the back edge from 5 to the phi-node 1. Analogously, we remove the back edges from 6 to 2 and from 7 to 3. However, there are many values $\underline{i} < \underline{i'}$ that make $\underline{i} = \underline{i'} - 1$ and the back edge from 8 to 4 remains (def-use pairs for `D`). As

a result of removing these spurious edges, Basic Vectorization will find that statement 5 is vectorizable. Statements 6, 7 and 8 will correctly appear in the FOR loop.

Note however, that this step renders some array phi-nodes target-less. We handle target-less phi-nodes with a minor extension of Basic Vectorization. First, we merge closures that update the same array. This simplifies handling of array ϕ-nodes: if each closure is turned into a separate loop each loop will need to have its own array phi-node to account for the update and this would complicate the analysis. Second, we add the target-less node of array `A` back to the closure that updates `A` — the intuition is, even if there is no loop-carried dependence from writes to reads on `A`, `A` is written and the write (i.e., update) cannot be vectorized; therefore, the updated array has to carry to the next iteration of the loop. Third, in cases when the phi-node remains target-less, i.e., cases when the array write can be vectorized, we have to properly remove the phi-node replacing uses of the left-hand side of the phi-node with its arguments.

5.5.2 Array MUX refinement

ANA: *TODO: WIP, get to BV without MUX refinement for now? For now, skip MUX refinement.*

Next, the algorithm refines array MUX statements. MPC-source after Cytron's SSA may result in statements $\mathtt{A}_j = MUX(..., \mathtt{A}_k, \mathtt{A}_l)$, which imply that any index of A can be written at this point and therefore there is a loop-carried dependency. In some cases the MUX can be refined to just a single index or a pair of indices, e.g., $\mathtt{A}_j[i] = MUX(c, \mathtt{A}_k[i], \mathtt{A}_l[i])$.

This is to reduce the dimensionality of simd-ified computation. Technically, $\mathtt{A}_j = MUX(..., \mathtt{A}_k, \mathtt{A}_l)$ is a simdified operation that can be carried out in parallel "in one round". However, particularly when `A` is a multi-dimensional array, there is substantial increase in the size of the arrays (vectors) we send to SIMD operations. Refining to an update to a specific index would reduce the size of those vectors. Note that this is a heuristic that handles a common case, but not all cases of array updates.

for each *stmt*: $\mathtt{A}_j = MUX(c, \mathtt{A}_k, \mathtt{A}_l)$ in the MPC-source seq. **do**
  $i_1 = find\_update(A_k)$ { Is null when $A_k = \phi(...)$ }
  $i_2 = find\_update(A_l)$ { Is null when $A_l = \phi(...)$ }
  if $i_1 == i_2$ or i_1 is null or i_2 is null **then**
    { With our restrictions on writes we must have $i_1 = i_2$. }
    replace *stmt* with
    $\mathtt{A}_j = \mathtt{A}_{j-1}$; $\mathtt{A}_j[i_1] = MUX(c, \mathtt{A}_k[i_1], \mathtt{A}_l[i_1])$
  else
    stmt stays as is
  end if
end for

5.5.3 Restricting Array Writes

For now, we restrict array updates to *canonical updates*. Assume (for simplicity) a two-dimensional array `A[I,J]`. A canonical update is the following:

```
for i in range(I):
  ...
  for j in range(J):
    ...
    A[i,j] = ...
    ...
```

The update `A[i,j]` can be nested into an inner loop and there may be multiple updates, i.e., writes to `A[i,j]`. However, update such as `A[i-1,j] = ...` or `A[i-1,j-1] = ...`, etc., is not allowed. Additionally, while there could be several different loops that perform canonical updates, they must be of the same dimensionality, i.e., an update of higher or lower dimension, e.g., `A[i,j,k] = ...` is not allowed. We compute the *canonical dimensionality* of each write array by examining the array writes in the original program and rejecting programs that violate the canonical write restriction. This restriction simplifies reasoning in this early stage of the compiler; we will look to relax the restriction in future work.

Another restriction/assumption is that we assume the output array is given as input with initial values, and it is of size consistent with its canonical dimensionality.

Reads through an arbitrary formula, such as `A[i-1]` for example, are allowed; currently, the projection function returns dummy values if the read formula is out of bounds; we assume the programmer ensures that the program still computes correct output in this case.

5.5.4 Changes to Basic Vectorization

In addition to the changes for the handling of target-less phi-nodes, Basic Vectorization has to handle differently def-use edges $X \rightarrow Y$ where X defines and Y uses an array variable. The definition can be an update `A_2 = update(A_1,i,...)` or a pseudo ϕ-node `A_2 = PHI(A_0,A_1)`, etc.. Note that ϕ-nodes for arrays have no subscript operations the way there are subscript operations in analysis-introduced arrays representing scalars. Raising and dropping dimension still apply, however, the dimension cannot be raised or dropped to a dimension lower than the canonical one:

(1) same-level $X \rightarrow Y$. We do nothing, just propagate the array, which happens to be of the right dimension.
(2) inner-to-outer $X \rightarrow Y$. If dimensionality of the loop enclosure of X is greater than the canonical dimensionality of the array, then add *drop_dim(...)* at Y, as in Basic Vectorization. Otherwise, do nothing. ANA: *I think it will be better to add drop dim and raise dim as in the previous case but change the semantics of both as written here.*
(3) outer-to-inner $X \rightarrow Y$. If dimensionality of loop enclosure of Y is greater than the canonical dimensionality of the array, then add *raise_dim(...)* (at X) as in Basic Vectorization. Otherwise, do nothing.
(4) "mixed" $X \rightarrow Y$. We assume that the mixed edge is transformed into an inner-to-outer followed by outer-to-inner edge before we perform vectorization, just as with Basic vectorization.

5.5.5 Examples with Array Writes

Example 1. First, the canonical dimensionality of all `A,B,C` and `D` is 1. After Phase 1 of Basic Vectorization the Aiken's array write example will be (roughly) as follows:

```
for i in range(N):
1. A_1 = PHI(A_0,A_2)
2. B_1 = PHI(B_0,B_2)
3. C_1 = PHI(C_0,C_2)
4. D_1 = PHI(D_0,D_2)
5. A_2 = update(A_1, I, B_1[I] + 10);
6. B_2 = update(B_1, I, A_2[I] * D_1[I-1]);
7. C_2 = update(C_1, I, A_2[I] * D_1[I-1]);
8. D_2 = update(D_1, I, B_2[I] * C_2[I]);
```

Note that since all def-uses are same-level (i.e., reads and writes of the array elements) no raise dimension or drop dimension happens.

Phase 2 computes the closure of 4; $cl = \{4, 6, 7, 8\}$ while 5 is vectorizable. Recall that 1,2, and 3 are target-less phi-nodes. Since the closure cl includes updates to `B` and `C`, the corresponding phi-nodes are added back to the closure and the def-use edges are added back to the target-less nodes. The uses of `A_1` and `B_1` in the vectorized statement turn into uses of `A_0` and `B_0` respectively; this is done for all original target-less phi-node. (But note that `A_0` is irrelevant; the update writes into array `A_2` in parallel.) Finally, the target-less phi-node for `A` is discarded.

```
1. A_2 = update(A_0, I, ADD_SIMD(B_0[I],10));
   equiv. to A_2[I] = ADD_SIMD(B_0[I],10)
FOR i=0; i<N; i++; // MOTION loop
2. B_1 = PHI(B_0,B_2)
3. C_1 = PHI(C_0,C_2)
4. D_1 = PHI(D_0,D_2)
5. B_2 = update(B_1, i, A_2[i] * D_1[i-1]);
   equiv. to B_2 = B_1; B_2[i] = A_2[i] * D_1[i-1];
6. C_2 = update(C_1, i, A_2[i] * D_1[i-1]);
7. D_2 = update(D_1, i, B_2[i] * C_2[i]);
```

Example 2. Now consider the MPC Source of Histogram:

```
for i in range(0, num_bins):
   res1 = PHI(res, res2)
   for j in range(0, N):
      res2 = PHI(res1, res3)
      tmp1 = (A[j] == i)
      tmp2 = (res2[i] + B[j])
      tmp3 = MUX(tmp1, res2[i], tmp2)
      res3 = Update(res2, i, tmp3)
return res1
```

The canonical dimensionality of `res` is 1. Also, the phi-node `res1 = PHI(res, res2)` is a target-less phi-node (the implication being that the inner for loop can be vectorized across i). After Phase 1, Basic vectorization produces the following code (statements are implicitly vectorized along i and j). In a vectorized update statement, we can ignore the incoming array, `res2` in this case. The update writes (in parallel) all locations of the 2-dimensional array, in this case it sets up each `res3[i,j] = tmp3[i,j]`.

```
A1 = raise_dim(A, j, ((i:num_bins),(j:N)))
B1 = raise_dim(B, j, ((i:num_bins),(j:N)))
I = raise_dim(i, ((i:num_bins),(j:N)))
for i in range(0, num_bins):
   res1 = PHI(res, res2) // target-less, no drop_dim
   res1' = raise_dim(res1, (j:N))
   for j in range(0, N):
      res2 = PHI(res1', res3)
      tmp1 = (A1 == I)
      tmp2 = (res2 + B1)
      tmp3 = MUX(tmp1, res2, tmp2)
      res3 = Update(res2, (I,J), tmp3)
return res1
```

Processing the inner loop in Phase 2 vectorizes `tmp1 = (A1 == I)` along the j dimension but leaves the rest of the statements in a MOTION loop. Processing the outer loop is interesting. This is because the PHI node is a target-less node, and therefore, there are no closures! Thus, everything can be vectorized along the i dimension:

```
A1 = raise_dim(A, j, ((i:num_bins),(j:N)))
B1 = raise_dim(B, j, ((i:num_bins),(j:N)))
I1 = raise_dim(i, ((i:num_bins),(j:N)))

tmp1[I,J] = (A1[I,J] == I1[I,J])

res1' = raise_dim(res, (j:N)) // raising res to a 2-dim. array
for j in range(0, N):
   res2 = PHI(res1', res3)
   tmp2[I,j] = (res2[I,j] + B1[I,j])
   tmp3[I,j] = MUX(tmp1[I,j], res2[I,j], tmp2[I,j])
   res3 = Update(res2, (I,j), tmp3)
res1 = drop_dim(res2) // Add drop_dim before removing redundant phi-no
return res1
```

The cleanup phase can then get rid of j in the `res` variables and `tmp` variables, so the loop operates over the `num_bins`-size vectors. This will lead to dropping raise_dim and drop_dim as well:

```
A1 = raise_dim(A, j, ((i:num_bins),(j:N)))
B1 = raise_dim(B, j, ((i:num_bins),(j:N)))
I1 = raise_dim(i, ((i:num_bins),(j:N)))

tmp1[I,J] = (A1[I,J] == I1[I,J])

for j in range(0, N):
   res2 = PHI(res1, res3) // now 1-dim array
   tmp2[I] = ADD_SIMD(res2[I],B1[I,j])
   tmp3[I] = MUX_SIMD(tmp1[I,j], res2[I], tmp2[I])
   res3 = Update(res2, I, tmp3[I])
return res2
```

6 DIVIDE-AND-CONQUER

ANA: *TODO: Now that we have broken FOR loops into smaller chunks, we can add Divide-and-conquer reasoning with Z3 and implement this additional transform.*

7 IMPLEMENTATION AND EVALUATION

8 FUTURE WORK

9 CONCLUSIONS