

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 a Python-like high-level program and produces amortized (i.e., vectorized) low-level cryptographic code in the MOTION framework. Central to our approach is a new vectorization algorithm that produces optimal schedules for a large number of MPC programs.

^{*}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 SICS Cyber Nexus Research Exchanges grant.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6747-9/19/11...\$15.00

<https://doi.org/10.1145/3319535.3339818>

- 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: $0 \leq i < N$. 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 point the programmer is responsible for linearization and access.

- (3) Array subscripts 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., $A[i,j] = \dots$ where i and j loop over the two dimensions of A is allowed, but $A[i,j+2] = \dots$ 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)

  !2!1 = (min_sum!2, min_index!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, $2 * \alpha + D * \beta + N * 3 * \alpha \ll N * D * (2 * \alpha + \beta) + N * 3 * \alpha$. 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 $\alpha \gg \beta$. 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.

We have def-use dependency graph $G(V, E)$ for the input program. Each vertex/node in this graph represents an operation/instruction in the program and an edge represents dependence i.e. an edge from V_1 to V_2 means the result of operation in V_1 is used in the operation in V_2 . To build a parallel schedule, we translate this graph $G(V, E)$ into another dependency graph $G'(V', E')$ such that:

- (1) We collapse multiple vertices of the same type (i.e. same operation) in V into a single vertex in V' .
- (2) dependencies are not broken i.e. If there is an edge from V_1 to V_2 ($V_1, V_2 \in V$), and we put them in vertices $V'_1, V'_2 \in G'$ respectively, then an edge must exist from V'_1 to V'_2 .

We want to build a graph G' with fewest possible vertices since fewer vertices means more parallelization. *ISHAQ: TODO: make it more concrete, pick some variables e.g. α, β like above and show 1) parallel schedule will have lower cost (this should be easy). and 2) some sort of optimality? this would be hard*

3.3 Scheduling is NP-hard

ANA: Ishaq? ISHAQ: TODO: As a first step, pull the super-sequence proof from Lindsay's thesis?

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 . We write $A[i, j, k]$ to denote canonical access to an array element. *ANA: Need to state this more precisely.*

4.1 Semantic Analysis

ANA: Benjamin?

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

ANA: TOTO Ana: Shorten this section. Lots of this is unnecessary.

4.3.1 Def-use Edges

The dependence graph has the following def-use edges:

- same-level forward $X \rightarrow Y$ where X and Y are in the same loop nest i_1, i_2, \dots, i_k . E.g., $d = SUB(S[i, j], C[j])$ to $p = MUL(d, d)$ in Biometric is a same-level edge. A ϕ node can be a source of a same-level forward edge but not a target.
- outer-to-inner forward $X \rightarrow Y$ where X is in an outer loop nest, i_1, i_2, \dots, i_j , and Y is in an inner one, $i_1, i_2, \dots, i_j, \dots, i_k$. A ϕ -node can be a source or a target of an outer-to-inner forward edge.
- inner-to-outer forward $X \rightarrow Y$ where X is a ϕ -node in an inner loop nest, $i_1, i_2, \dots, i_k, i_{k+1}$, and Y is in the enclosing loop nest i_1, i_2, \dots, i_k . E.g. $sum_0 = \phi(sum_1, 0)$ to $c = CMP(sum_0, min_0)$ is an inner-to-outer forward edge. Note that the source is *always* a ϕ -node in the immediately enclosing loop. *BEN: The current representation of pseudo ϕ -nodes shows them attached to the loop header (i.e. in the loop). We may want to clarify that they are also evaluated at the loop's termination.*
- same-level back-edge $X \rightarrow Y$. Y is a ϕ -node in the header of the loop and X is a definition of the variable in the loop body. E.g., $min_1 = MUX(c, sum_1, min_1)$ to $min_0 = \phi(min_1, 10000)$ in Biometric is a same-level back-edge.
- inner-to-outer back-edge $X \rightarrow Y$: X and Y are both ϕ -nodes for some variable. The source X is in a loop nested into Y 's loop (not necessarily immediately).
- mixed forward edge $X \rightarrow Y$. X is a ϕ -node in some loop $i_1, i_2, \dots, i_k, i_{k+1}$ and Y is a node in a loop nested into i_1, i_2, \dots, i_k . We transform mixed forward edges as follows. Let x_j be the variable defined at the ϕ -node X . We add a variable and assignment $x'_j = x_j$ immediately after the i_1, i_2, \dots, i_k loop. Then we replace the use of x_j at Y with x'_j . 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 ϕ -node. Intuitively, it computes the set of nodes (i.e., statements) that form a dependence cycle with n . *ANA: Cycle(n) is probably a better name.* 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 $X \rightarrow n$ is a same-level back-edge.



Figure 1: Compiler Framework.

- Y is in $\text{closure}(n)$ if there is a same-level path from n to Y and there is a same-level path from Y to some X in $\text{closure}(n)$.

4.4 Taint Analysis

ANA: Ben?

5 VECTORIZATION

ANA: TODO: Still work in progress, all over.

5.1 Raise Dimension and Drop Dimension

We define the *raise_dim* (raise dimensions) and *drop_dim* (drop dimension) functions. Raise dimension "lifts" a lower-dimensional array (ANA: *The right term here is tensor, I am pretty sure!*) into a higher dimension one. This is necessary when a lower-dimensional array is used in a higher dimensional loop and, essentially, is just copying of values. For example, Biometric contains the statement $d = \text{SUB}(S[i, j], C[j])$, in the j -loop which is nested into the i -loop. Here C is a one-dimensional array, however, to vectorize across both loops it is necessary to turn it into a two-dimensional array: $C[i, j]$ becomes $[C[0], C[1], \dots, C[J], C[0], C[1], \dots, C[J], \dots, C[0], C[1], \dots, C[J]]$, which turns the row into a matrix of I identical rows. (We use capital letters to denote the upper bounds of loops, e.g., J is the upper bound of the j -loop and I is the upper bound of the i -loop.)

$\text{raise_dim}(A[i_1, \dots, i_k], i_j)$ is defined as follows. It results in a new $k+1$ -dimensional array A' where for every $0 \leq i_k < I_{k+1}$, $A'[i_1, \dots, i_j, \dots, i_k] = A[i_1, \dots, i_k]$. Adding n dimensions is trivially extended as a composition of n *raise_dim* that each adds a single dimension.

As expected, drop dimension turns a higher-dimensional array into a lower-dimensional one. The key use case is an inner-to-outer def-use edge. The code may define a variable, e.g., x in an inner loop, say j , then use this variable in an enclosing loop, say i . Our algorithm may vectorize the computation of x in the j -loop thus producing a vector $x[j]$ where $x[1]$ is the value of x after the 1st iteration and so on. Drop dimension states that the outer loop will use the value of the variable at the last iteration.

$\text{drop_dim}(A[i_1, \dots, i_j, \dots, i_k], i_j)$ produces a k -dimensional array A' where $A'[i_1, \dots, i_k] = A'[i_1, \dots, i_j - 1, \dots, i_k]$. In our analysis, dimensions are always dropped at the end, and again, one can define dropping n dimensions as a composition of n *drop_dim*.

BEN: *I'm not sure if I wrote up the above paragraphs correctly – the idea is that we're just copying values to extend over dimension j or only retain its last elements. The current writeup implies that $1 < j < k$ though, which isn't always true. Python implementations of *raise_dim()* and *drop_dim()* can be found here: https://github.com/milana2/ParallelizationForMPC/blob/master/compiler/compiler/motion_backend/reference_implementations.py*

ANA: *This section is still informal. Needs work to make more precise.*

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 :  $X = \text{Op}(Y_1, Y_2)$  in loop  $i, j, k$  do
  for each argument  $Y_n$  do
    case def-use edge  $\text{stmt}'(\text{def of } Y_n) \rightarrow \text{stmt}(\text{def of } X)$ 
    of
      same-level:  $Y'_n$  is  $Y_n$ 
      outer-to-inner: add  $Y'_n[i, j, k] = \text{raise\_dim}(Y_n)$  at  $\text{stmt}'$ 
      inner-to-outer: add  $Y'_n[i, j, k] = \text{drop\_dim}(Y_n)$  at  $\text{stmt}$ 
    end for
  { Optimistically vectorize all.  $I$  means vectorized dimension. }
  change to  $X[I, J, K] = \text{Op}(Y'_1[I, J, K], Y'_2[I, J, K])$ 
end for

```

{ Phase 2: Recreating FOR loops for cycles; vectorizable statements hoisted up. }

```

for each dimension  $d$  from highest to 0 do
  for each  $\phi$ -node  $n$  in loop  $i_1, \dots, i_d$  do
    compute  $\text{closure}(n)$ 
  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  $i_d = 0; \dots$  loop
    add  $\phi$ -nodes in  $cl$  to header block
    add target-less  $\phi$ -node for  $A$  if  $cl$  updates array  $A$ 
  end for
end for

```

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: Dimensionality reduction: removing unnecessary dimensionality. }

{ A dimension i is dead on exit from stmt $X[\dots i \dots] = \dots$ if all def-uses with targets outside of the enclosing FOR $i = 0 \dots$ MOTION loop end at target (use) $X' = \text{drop_dim}(X, i)$. }

for each stmt and dimension $X[\dots i \dots] = \dots$ **do**
 if i is a dead dimension on exit from stmt $X[\dots i \dots] = \dots$,
 remove i from X (all defs and uses)

end for

{ Now clean up drop_dim and raise_dim }

for each $X' = \text{drop_dim}(X, i)$ **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 simdfied 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., $\text{min_sum!3} = i$), operations, and PHI nodes are *implicitly vectorized*. *raise_dim* and *drop_dim* statements, as well as propagation statements (e.g., $\text{min_sum!3} = \text{sum!3}^{\wedge}$) may have slightly different interpretation. *ANA: This is vague, may need elaboration.*

Note the two different versions of *raise_dim*. *ANA: More here! One just adds a least-significant dimension. The other one may reshape an input array.*

```
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 $\text{sum!3} = \text{PHI}(\text{sum!2}, \text{sum!4})$, the algorithm first computes its closure. The closure amounts to the phi-node itself and the addition node $\text{sum!4} = (\text{sum!3} + \text{p!3})$, accounting for the loop-carried dependency of the computation of sum . The algorithm replaces this closure with a FOR loop on j removing vectorization 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 $\text{p!3}[I, J] = (\text{d!3}[I, J] * \text{d!3}[I, J])$ to the monolithic FOR loop and from the FOR loop to $\text{sum!3} \hat{=} \text{drop_dim}(\text{sum!3})$. (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 !1!2[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)

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)
```

ANA: Begin Work-in-progress notes on implementation. Just some thoughts... 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 may apply both raise dimension and drop dimension on these.
- 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 and raise dimension may reshape arrays. If there are multiple "views" of the input array, there could 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 will be canonical. Only raise dimension applies.
- Read-write output arrays: Writes are canonical (by restriction) but reads can be non-canonical. Conjecture: do nothing. Dependence analysis takes care of limiting vectorization so non-canonical access will work. Still work in progress. We may apply both raise and drop dimension.

ANA: End Work-in-progress notes.

5.4 Correctness Argument

5.5 Towards Extension of Basic Vectorization

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. $A_0 = \phi(A, A_1)$
2. $B_0 = \phi(B, B_1)$
3. $C_0 = \phi(C, C_1)$
4. $D_0 = \phi(D, D_1)$
5. $A_1 = A_0$; $A_1[i] = B_0[i] + 10$; {equiv. to Update}
6. $B_1 = B_0$; $B_1[i] = A_1[i] * D_0[i-1]$;
7. $C_1 = C_0$; $C_1[i] = A_1[i] * D_0[i-1]$;
8. $D_1 = D_0$; $D_1[i] = B_1[i] * C_1[i]$;

There is a cycle around $B_0 = \phi(B, B_1)$ that includes statement $A_1[i] = B_0[i] + 10$; 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 8 to the read of $\dots = B_0[i]$ at 6.

The following algorithm removes certain infeasible loop-carried dependencies that are due to array writes. Consider a

loop with index $0 \leq j < J$ 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:  $A_m[f(i, j, k)] = \dots$ , and use:  $\dots = A_n[f'(i, j, k)]$  in loop j do
    if  $\exists \underline{i}, \underline{j}, \underline{j}', \underline{k}, \underline{k'}$ , s.t.  $0 \leq \underline{i} < I$ ,  $0 \leq \underline{j}, \underline{j}' < J$ ,  $0 \leq \underline{k}, \underline{k}' < K$ ,  $\underline{j} < \underline{j}'$ , 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  $\phi$ -node in loop j.
  end if
end for
```

ISHAQ: Note to self: This algorithm is an instantiation for j loop, the one for k loop will be exactly the same, modulo variable name..

Consider a loop j enclosed in some fixed \underline{i} . Only if an update (definition) $A_m[f(i, j, k)] = \dots$ at some iteration \underline{j} references the *same* array element as a use $\dots = A_n[f'(i, j, k)]$ 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 6-8 and 6-10). Therefore, we remove the back edge from 6 to 1. Analogously, we remove the back edges from 8 to 2 and 10 to 3. However, there are many values $\underline{i} < \underline{i}'$ that make $\underline{i} = \underline{i}' - 1$ and the back edge from 12 to 4 remains (def-use pairs for D). As a result of removing these spurious edges, Basic Vectorization will find that statement 6 is vectorizable. Statements 8, 10 and 12 will correctly appear in the FOR loop.

5.5.2 Array MUX refinement

ANA: TODO: I think we should implement this.

Next, the algorithm refines array MUX statements. MPC-source after Cytron's SSA may result in statements $A_j = MUX(\dots, A_k, A_l)$, 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., $A_j[i] = MUX(c, A_k[i], A_l[i])$.

This is to reduce the dimensionality of simd-ified computation. Technically, $A_j = MUX(\dots, A_k, A_l)$ 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:  $A_j = MUX(c, A_k, A_l)$  in the MPC-source seq.
do
   $i_1 = \text{find\_update}(A_k)$  { Is null when  $A_k = \phi(\dots)$  }
   $i_2 = \text{find\_update}(A_l)$  { Is null when  $A_l = \phi(\dots)$  }
  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
     $A_j = A_{j-1}; A_j[i_1] = MUX(c, A_k[i_1], A_l[i_1])$ 
  else
    stmt stays as is
  end if
end for

```

5.6 Extension of Basic Vectorization with Array Writes

5.6.1 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):
    ...
     $A[i, j] = \dots$ 
    ...

```

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 $A[i - 1, j] = \dots$ or $A[i - 1, j - 1] = \dots$, 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., $A[i, j, k] = \dots$ is not allowed. We compute the *canonical dimensionality* of each write array in the obvious way *before basic vectorization*. 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, our 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.

5.6.2 Changes to Basic Vectorization

There are two changes to Basic Vectorization to account for write arrays. We can account for them during processing of raise dimension/drop dimension in Phase 1 of Basic Vectorization.

One change to Basic vectorization is the expansion of dimension if the array write or read occurs in a nested loop.

That is, if there is an update $A[i, j] = \dots$ that occurs in loop dimensionality i, j, k , $A[i, j]$ will be rewritten into $A[i, j, k]$. Similarly, a read $A[f(i, j)]$ will be rewritten into $A[f(i, j), k]$. ($A[f(i, j), k]$ will be reshaped as $f(i, j) * I * J + k$ during Phase 1 of Basic Vectorization. **ANA: TODO: Double check.**)

The other change concerns def-use edges $X \rightarrow Y$ where X defines and Y uses an array variable. The definition can be an update $A_2 = \text{update}(A_1, \dots)$ or a pseudo ϕ -node $A_2 = \phi(A_0, A_1)$. Note that ϕ nodes for arrays are no subscript operations on those ϕ -nodes the way there are in scalar arrays. These edges are not handled in the same way as in Basic Vectorization, specifically, we do not raise and drop dimension as we do for scalars in Basic Vectorization. A key invariant is that the dimension of an array A cannot go above or drop below A 's canonical dimensionality. We enumerate the cases of def-use edges.

- (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.
- (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.

5.6.3 Examples with Array Writes

Example 1. First, the canonical dimensionality of all A, B, C and D is 1. Thus, there is no addition of extra dimension for inner loops. 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.  $A_2 = \text{update}(A_1, \vec{i}, B_1[\vec{i}] + 10);$ 
  6.  $B_2 = \text{update}(B_1, \vec{i}, A_2[\vec{i}] * D_1[\vec{i} - 1]);$ 
  7.  $C_2 = \text{update}(C_1, \vec{i}, A_2[\vec{i}] * D_1[\vec{i} - 1]);$ 
  8.  $D_2 = \text{update}(D_1, \vec{i}, B_2[\vec{i}] * C_2[\vec{i}]);$ 

```

Phase 2 computes the closure of 4; $cl = \{4, 6, 7, 8\}$ while 5 is vectorizable. Recall that 1, 2, and 3 are targetless phi-nodes. Since the closure cl includes updates to B and C , the corresponding phi-nodes are added to the closure, i.e., the FOR loop. The uses of A_1 and B_1 in the vectorized statement turn into uses of A_0 and B_0 respectively. (But note that A_0 is irrelevant; the updates writes into array A_2 in parallel.)

```

1.  $A_2 = \text{update}(A_0; \vec{i}, \text{ADD\_SIMD}(B_0[\vec{i}], [10, \dots]))$  {Fully
vectorized, size N.}
FOR i=0; iN; 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. $B_2 = \text{update}(B_1, i, \text{MUL}(A_2[i], D_1[i - 1]))$
6. $C_2 = \text{update}(C_1, i, \text{MUL}(A_2[i], D_1[i - 1]))$
7. $D_2 = \text{update}(D_1, i, \text{MUL}(B_2[i], C_1[i]))$

ANA: TODO: Format either Histogram or Matrix Multiplication. Both are interesting examples that use output arrays.

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