Scheduling and Amortization 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

protocol mixing; linear programming; multiparty computation; program analysis; cryptography

ACM Reference Format:
Benjamin Levy, Benjamin Sherman, Lindsey Kennard, Ana L. Milanova, Muhammad Ishaq, and Vassilis Zikas. 2019. Scheduling and Amortization 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

2 BACKGROUND AND PROBLEM STATEMENT

2.1 Scheduling in MPC

ANA: *Here we need more on MPC before jumping to scheduling. MPC-source, basic assumptions about static loop bounds, etc.*

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. ISHAQ: *This is not an assumption, this is our setting. We work in the single protocol world.*
(2) All MPC instructions have the same unit cost, 1 unit. ISHAQ: *Don't we need to distinguish between local (cheap) vs. interactive (expensive) instructions? I don't see how we could assume this. – Past suggestion from Vassilis: Ideally we should assume cost to be a convex function.*ANA: *After discussion 1/12: replace single costs 1 with two levels of costs: α for non-local operations, e.g., MUL, and β for local ops, e.g., ADD.*
(3) There is unlimited bandwidth—i.e., a single MPC-instruction costs as much as N amortized instructions, namely 1 unit.ISHAQ: *Comment from Vassilis: We assume there are infinite parallel capacities, this is the standard assumption in PRAM (Cryptographic Parallel RAM).*ANA: *PRAM assumption stays. We have to replace the unit cost of 1 with a suitable amortization function $f(n)$, which I don't think changes much in the cost modeling and analysis.*
(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 1, however a MUL and a MUX instructions scheduled in parallel still cost 2. ISHAQ: *We need to reword this assumption to something like "K parallel MUL costs much less than K (because they get amortized), but any mix of K MUL and MUX still cost roughly K. Specifically, we should take away the low constants (2 and 1) because we know for low constants this is not true.* ANA: *Again, core assumption that MUL and MUX don't benefit from amortization stays. We have to replace constant costs with functions, as in (3).*

2.2 Scheduling in HPC

3 PRELIMINARIES

We assume arbitrarily nested loops in the MPC-source IR and read-only arrays. We assume that loops range from 0 to some constant N. Arrays are linearized (row-major order as in MOTION) and accesses are via functions of the induction variables of the enclosing loops. We write $i_1, i_2, ...i_k$ to denote

*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

the loop nest: i_1 is the outermost loop, i_2, is immediately nested in i_1, and so on until i_k. As an example, a statement nested in $i_1, i_2, ...i_k$ can access array $A[f(i_1, i_2, ...i_k)]$. We write $A[i_1, i_2, ...i_k]$ interchangeably. ANA: *Need to state this more precisely.*

3.1 Pseudo ϕ-nodes

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.

3.2 Def-use Edges

The dependence graph has the following def-use edges:

- same-level forward $X \to Y$ where X and Y are in the same loop nest $i_1, i_2, ...i_k$. E.g., $\mathtt{d} = SUM(\mathtt{S}[i,j], \mathtt{C}[j])$ to $\mathtt{p} = MUL(\mathtt{d}, \mathtt{d})$ 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 $X \to Y$ where X is in an outer loop nest, $i_1, i_2, ...i_j$, and Y is in an inner one, $i_1, i_2, ...i_j, ..., i_k$. A ϕ-node can be a source or a target of an outer-to-inner forward edge.
- inner-to-outer forward $X \to Y$ where X is a *phi*-node in an inner loop nest, $i_1, i_2, ...i_k, i_{k+1}$, and Y is in the enclosing loop nest $i_1, i_2, ...i_k$. E.g. $\mathtt{sum}_0 = \phi(\mathtt{sum}_1, 0)$ to $\mathtt{c} = CMP(\mathtt{sum}_0, \mathtt{min}_0)$ is an inner-to-outer forward edge. Note that the source is *always* a ϕ-node in the immediately enclosing loop. The interpretation of this edge is that the use node Y uses the definition made in the last iteration of the inner 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 $X \to Y$. Y is a phi-node in the header of the loop and X is a definition of the variable in the loop body. E.g., $\mathtt{min}_1 = MUX(\mathtt{c}, \mathtt{sum}_1, \mathtt{min}_1)$ to $\mathtt{min}_0 = \phi(\mathtt{min}_1, 10000)$ in Biometric is a same-level back-edge.
- inner-to-outer back-edge $X \to Y$: 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 $X \to Y$. X is a *phi*-node in some loop $i_1, i_2, ...i_k, i_{k+1}$ and Y is a node in a loop nested into $i_1, i_2, ...i_k$. We transform mixed forward edges as follows. Let $\mathtt{x}_j$ be the variable defined at the phi-node X. We add a variable and assignment $\mathtt{x}'_j = \mathtt{x}_j$ immediately after the $i_1, i_2, ...i_k$ loop. Then we replace the use of $\mathtt{x}_j$ at Y with $\mathtt{x}'_j$. 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.

3.3 Helper Functions

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. 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 $X \to n$ 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).

We define the *raise_dim* (raise dimensions) and *drop_dim* (drop dimension) functions. Raise dimension "lifts" a lower-dimension 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 $d = SUB(S[i,j], C[j])$, 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 $[C[0], C[1], ...C[J], C[0], C[1], ...C[J], ...C[0], C[1], ...C[J]]$, 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.)

raise_dim$(A[i_1, ..., i_k], i_j)$ is defined as follows. It results in a new k+1-dimensional array A' where for every $0 \le i_k < I_{k+1}$, $A'[i_1, \ldots, i_j, \ldots, i_k] = A[i_1, \ldots, i_k]$. 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.

drop_dim$(A[i_1, \ldots, i_j, \ldots, i_k], i_j)$ produces a k-dimensional array A' where $A'[i_1, \ldots, i_k] = A'[i_1, \ldots, I_j - 1, \ldots, i_k]$. 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.*

4 VECTORIZATION

ANA: *Add back-edges into Phase 1. A back-edge from a non-phi-node in loop i to a phi-node in loop i's header is a same-level edge. The only difference with the handling of normal same-level forward def-use is that the operand index will become $i-1$. A back edge from a phi-node in loop j to a phi-node in loop i, where j is nested in i is an inner-to-outer edge and will require dropping dimension. We can show these are the only kinds of back-edges that may occur.*

4.1 Basic Vectorization

{Phase 1: Raise dimension of scalar variables to corresponding loop nest. We may assume linear traversal of the MPC-source.}
for each MPC $stmt : X = Op(Y_1, Y_2)$ in loop $i_1, ..., i_k$ **do**
  for each Y_i **do**
    case def-use edge $stmt'$(def of Y_i) $\rightarrow$ $stmt$(def of X) of
      same-level $\rightarrow$ add $Y_i' = Y_i$
      outer-to-inner $\rightarrow$ add $Y_i'[i_1, ..., i_k] = raise\_dim(Y_i)$
      inner-to-outer $\rightarrow$ add $Y_i'[i_1, ..., i_k] = drop\_dim(Y_i)$
  end for
  {Optimistically vectorize all. $\vec{i}$ means vectorized dimension.}
  change to $X[\vec{i_1}, ... \vec{i_k}] = Op(Y_1'[\vec{i_1}, ..., \vec{i_k}], Y_2'[\vec{i_1}, ..., \vec{i_k}])$
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 $closure(n)$
    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 statements in cl to loop body in some order of dependences
      {Dimension is not vectorizable: }
      change $\vec{i_d}$ to i_d in all statements in loop
      treat FOR loop as monolith node: change def-use edges accordingly. Some edges become same-level.
    end for
  end for
end for
{Phase 3:}
add SIMD for simdified dimensions

ISHAQ: *We need to think of a way to handle 4th case for def-use, when a nested block B def is followed by another, different, nested block B' use. Note: The handwritten notes say we can break it into 2 edges. One idea to this end is to insert a def' node which is a copy of the original def. The def' should be placed in a block that contains both B and B'.* ISHAQ: *THE ABOVE COMMENT IS RESOLVED. After transformation to SSA, there should be an intermediate variable (phi node) that is assigned a value depending on whether control branched or not. Should we put this into the writeup?*

4.2 Example: Biometric

4.2.1 MPC-source

Below is the MPC-source resulting from Benjamin's analysis. It gives rise to a corresponding dependence graph. (Not shown.)

{Begin of outer loop i.}
$\texttt{index}_0 = \phi(-1, \texttt{index}_1)$
$\texttt{min}_0 = \phi(10000, \texttt{min}_1)$
{Begin of inner loop j.}
$\texttt{sum}_0 = \phi(0, \texttt{sum}_1)$
$\texttt{d} = SUB(\texttt{S}[i, j], \texttt{C}[j])$
$\texttt{p} = MUL(\texttt{d}, \texttt{d})$
$\texttt{sum}_1 = \texttt{ADD}(\texttt{sum}_0, \texttt{p})$
{End of inner loop j.}
$\texttt{c} = CMP(\texttt{sum}_0, \texttt{min}_0)$
$\texttt{index}_1 = MUX(\texttt{c}, \texttt{index}_0, i)$
$\texttt{min}_1 = MUX(\texttt{c}, \texttt{min}_0, \texttt{sum}_0)$
{End of outer loop i.}

4.2.2 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.

{Begin of outer loop i.}
$\texttt{index}_0[i] = \phi(-1, \texttt{index}_1[i-1])$
$\texttt{min}_0[i] = \phi(10000, \texttt{min}_1[i-1])$
{Begin of inner loop j.}
$\texttt{sum}_0[i, j] = \phi([0, 0, ...], \texttt{sum}_1[i, j-1])$
$\texttt{C}'[i, j] = raise\_dim(\texttt{C}[j], i)$
$\texttt{d}[\vec{i}, \vec{j}] = SUB(\texttt{S}[\vec{i}, \vec{j}], \texttt{C}'[\vec{i}, \vec{j}])$
$\texttt{p}[\vec{i}, \vec{j}] = MUL(\texttt{d}[\vec{i}, \vec{j}], \texttt{d}[\vec{i}, \vec{j}])$
$\texttt{sum}_1[\vec{i}, \vec{j}] = ADD(\texttt{sum}_0[\vec{i}, \vec{j}], \texttt{p}[\vec{i}, \vec{j}])$
{End of inner loop j.}
$\texttt{sum}_0'[i] = drop\_dim(\texttt{sum}_0[i, j], j)$
$\texttt{c}[\vec{i}] = CMP(\texttt{sum}_0'[\vec{i}], \texttt{min}_0[\vec{i}])$
$\texttt{index}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{index}_0[\vec{i}], [0, 1, ...N])$
$\texttt{min}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{min}_0[\vec{i}], \texttt{sum}_0'[\vec{i}])$
{End of outer loop i.}

ISHAQ: *At least for input variables: when raising dimensions, we should keep some meta data around so we can later write code that could use 1 share if it knows the other N shares are just duplicates of it.*

4.2.3 Phase 2

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 $\texttt{sum}_0[i, j] = \phi([0, 0, ...], \texttt{sum}_1[i, j-1])$, the algorithm first computes its closure. The closure amounts to the phi-node itself

and $\texttt{sum}_1[\vec{i},\vec{j}] = ADD(\texttt{sum}_0[\vec{i},\vec{j}],\texttt{p}[\vec{i},\vec{j}])$, 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 algorithm adds same-level edges, one from $\texttt{p}[\vec{i},\vec{j}] = ...$ to the monolithic FOR-loop node, and one from the FOR-loop node to $\texttt{c}[\vec{i}] = ...$

ISHAQ: *For our radar: Such dependencies will not involve phi nodes only. e.g. $a[i] = a[i] + a[i-1]$*

{Begin of outer loop i.}
$\texttt{index}_0[i] = \phi(-1, \texttt{index}_1[i-1])$
$\texttt{min}_0[i] = \phi(10000, \texttt{min}_1[i-1])$
$\texttt{C}'[i,j] = raise\_dim(\texttt{C}[j], i)$
$\texttt{d}[\vec{i},\vec{j}] = SUB(\texttt{S}[\vec{i},\vec{j}], \texttt{C}'[\vec{i},\vec{j}])$
$\texttt{p}[\vec{i},\vec{j}] = MUL(\texttt{d}[\vec{i},\vec{j}], \texttt{d}[\vec{i},\vec{j}])$
{Begin of inner loop j. Loop is now much "shorter"!}
$\texttt{sum}_0[i,j] = \phi([0,0,...], \texttt{sum}_1[i,j-1])$
FOR j=0; j¡D; j++ { Will turn into a MOTION loop.}
  $\texttt{sum}_1[\vec{i},j] = ADD(\texttt{sum}_0[\vec{i},j], \texttt{p}[\vec{i},j])$
{End of inner loop j.}
$\texttt{sum}'_0[i] = drop\_dim(\texttt{sum}_0[i,j], j)$
$\texttt{c}[\vec{i}] = CMP(\texttt{sum}'_0[\vec{i}], \texttt{min}_0[\vec{i}])$
$\texttt{index}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{index}_0[\vec{i}], [0,1,...N])$
$\texttt{min}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{min}_0[\vec{i}], \texttt{sum}'_0[\vec{i}])$
{End of outer loop i.}

Next, analysis moves to outer dimension i. There are two phi-nodes, $\texttt{min}_0[i] = \phi(10000, \texttt{min}_1[i-1])$ and $\texttt{index}_0[i] = \phi(-1, \texttt{index}_1[i-1])$. The closure of the first is

$$\begin{aligned}&\texttt{min}_0[i] = \phi(10000, \texttt{min}_1[i-1])\\ &\texttt{c}[\vec{i}] = CMP(\texttt{sum}'_0[\vec{i}], \texttt{min}_0[\vec{i}]),\\ &\texttt{min}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{min}_0[\vec{i}], \texttt{sum}'_0[\vec{i}])\end{aligned}$$

and the closure of the second one is

$$\begin{aligned}&\texttt{index}_0[i] = \phi(-1, \texttt{index}_1[i-1]),\\ &\texttt{index}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{index}_0[\vec{i}], [0,1,...N])\end{aligned}$$

Since the two closures *do not* intersect, we have two distinct FOR-loops on i. The first FOR loop (on min) is scheduled first, because of the dependence edge from $\texttt{c}[\vec{i}] = CMP(\texttt{sum}'_0[\vec{i}], \texttt{min}_0[\vec{i}])$ to $\texttt{index}_1[\vec{i}] = MUX(\texttt{c}[\vec{i}], \texttt{index}_0[\vec{i}], [0,1,...N])$. We can now rewrite those statements, getting rid of the vectorization on i in the two FOR loops. Notably, the SUB and MUL computations are fully vectorizable and the ADD computation is vectorizable across the i-dimension.

$\texttt{C}'[i,j] = raise\_dim(\texttt{C}[j], i)$
$\texttt{d}[\vec{i},\vec{j}] = SUB\_SIMD(\texttt{S}[\vec{i},\vec{j}], \texttt{C}'[\vec{i},\vec{j}])$
$\texttt{p}[\vec{i},\vec{j}] = MUL\_SIMD(\texttt{d}[\vec{i},\vec{j}], \texttt{d}[\vec{i},\vec{j}])$

$\texttt{sum}_0[i,j] = \phi([0,0,...], \texttt{sum}_1[i,j-1])$
FOR j=0; j¡D; j++ { Turns into a MOTION loop, was j-loop.}
  $\texttt{sum}_1[\vec{i},j] = ADD\_SIMD(\texttt{sum}_0[\vec{i},j], \texttt{p}[\vec{i},j])$

$\texttt{sum}'_0[i] = drop\_dim(\texttt{sum}_0[i,j], j)$

$\texttt{min}_0[i] = \phi(10000, \texttt{min}_1[i-1])$
FOR i=0; i¡N; i++ { Turns into a MOTION loop, was i-loop.}
  $\texttt{c}[i] = CMP(\texttt{sum}'_0[i], \texttt{min}_0[i])$
  $\texttt{min}_1[i] = MUX(\texttt{c}[i], \texttt{min}_0[i], \texttt{sum}'_1[i])$

$\texttt{index}_0[i] = \phi(-1, \texttt{index}_1[i-1])$
FOR i=0; i¡N; i++ { Turns into a MOTION loop, was i-loop.}
  $\texttt{index}_1[i] = MUX(\texttt{c}[i], \texttt{index}_0[i], i)$

BEN: *The above code implicitly stores the final values of min_0 and index_0 in their last indices, but there is no primitive like drop_dim to make it clear. This will have to be added in the SSA code generation (though it might not be necessary to include in the paper?). It also is pretty wishy-washy about the size and layout of each vector (again more of an implementation issue), but maybe we should have a different syntax for array accesses for read/write (e.g. the ADD_SIMD steps) vs the array definitions (e.g. the SUB_SIMD and raise/drop_dim statements)*

4.3 Correctness Argument

4.4 Towards Extension of Basic Vectorization

4.4.1 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. $\texttt{A}_0 = \phi(\texttt{A}, \texttt{A}_1)$
2. $\texttt{B}_0 = \phi(\texttt{B}, \texttt{B}_1)$
3. $\texttt{C}_0 = \phi(\texttt{C}, \texttt{C}_1)$
4. $\texttt{D}_0 = \phi(\texttt{D}, \texttt{D}_1)$
5. $\texttt{A}_1 = \texttt{A}_0$; $\texttt{A}_1$[i] = $\texttt{B}_0$[i] + 10; {equiv. to Update}
6. $\texttt{B}_1 = \texttt{B}_0$; $\texttt{B}_1$[i] = $\texttt{A}_1$[i] * $\texttt{D}_0$[i-1];
7. $\texttt{C}_1 = \texttt{C}_0$; $\texttt{C}_1$[i] = $\texttt{A}_1$[i] * $\texttt{D}_0$[i-1];
8. $\texttt{D}_1 = \texttt{D}_0$; $\texttt{D}_1$[i] = $\texttt{B}_1$[i] * $\texttt{C}_1$[i];

There is a cycle around $\texttt{B}_0 = \phi(\texttt{B}, \texttt{B}_1)$ that includes statement $\texttt{A}_1$[i] = $\texttt{B}_0$[i] + 10; and that statement won't be vectorized even though in fact there is no loop-carried dependency from the write of $\texttt{B}_1$[i] at 8 to the read of ... = $\texttt{B}_0$[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: $\texttt{A}_m[f(i,j,k)] = ...$, 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. $0 \leq \underline{i} < I$, $0 \leq \underline{j}, \underline{j}' < J$, $0 \leq \underline{k}, \underline{k}' < K$, $\underline{j} < \underline{j}'$, 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 ϕ-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) $\texttt{A}_m[f(i,j,k)] = ...$ 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` 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.

ANA: *Commented out previous writeup. Removal/handling of targetless phi-nodes will go into the Extension to Basic vectorization.*

4.4.2 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 $\texttt{A}_j = MUX(..., \texttt{A}_k, \texttt{A}_l)$, 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., $\texttt{A}_j[i] = MUX(c, \texttt{A}_k[i], \texttt{A}_l[i])$.

This is to reduce the dimensionality of simd-ified computation. Technically, $\texttt{A}_j = MUX(..., \texttt{A}_k, \texttt{A}_l)$ 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. ANA: *TODO: Simplify this algorithm, taking into account the restriction to canonical updates. It should be handling _all_ cases.*

for each *stmt*: $\texttt{A}_j = MUX(c, \texttt{A}_k, \texttt{A}_l)$ 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
    $\texttt{A}_j = \texttt{A}_{j-1}$; $\texttt{A}_j[i_1] = MUX(c, \texttt{A}_k[i_1], \texttt{A}_l[i_1])$
  else
    stmt stays as is
  end if
end for

4.5 Extension of Basic Vectorization with Array Writes

ANA: *TODO: Handling of Targetless phi nodes should be done here, as an extension to Basic Vectorization.*

4.5.1 Restricting Array Writes

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

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

The update $\texttt{A}[i, j]$ can be nested into an inner loop and there may be multiple updates, i.e., writes to $\texttt{A}[i, j]$. However, update such as $\texttt{A}[i-1, j] = ...$ or $\texttt{A}[i-1, j-1] = ...$, 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 assign the *canonical dimensionality* to each array in the obvious way. This restriction simplifies reasoning in this early stage of the compiler; we will look to relax the restriction in future work.

Reads through an arbitrary formula, such as $\texttt{A}[i-1]$ for example, are allowed, however, we assume the programmer ensures that the formula is within the bounds of the array.

4.5.2 Changes to 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 $\texttt{A}[i, j] = ...$ that occurs in loop nest i, j, k, $\texttt{A}[i, j]$ will be rewritten into $\texttt{A}[i, j, k]$. Similarly, a read $\texttt{A}[f(i, j)]$ will be rewritten into $\texttt{A}[f(i, j), k]$. ANA: *TODO:*

This notion of $f(i,j)$ needs a precise definition. For now I'm hoping the hand waving works. This can be a preprocessing step or it can be incorporated into the Basic Vectorization algorithm.

The other change concerns def-use edges $X \rightarrow Y$ where X defines and Y uses an array variable (e.g., the definition can be a propagation $A_2 = A_1$ or a pseudo ϕ-node $A_2 = \phi(A_0, A_1)$). These edges are not handled in the same way as in Basic Vectorization, specifically, we do not expand dimension as we do for scalars in Basic Vectorization according to the loop nest. A key invariant is that the dimension of an array A cannot drop below A's canonical dimensionality. Below 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. ANA: *There might be some opportunities to do copy propagation optimization and save some cycles, but let's leave this for later.*

(2) inner-to-outer $X \rightarrow Y$. Let A be the array defined at X. If the dimensionality of A is greater than its canonical dimensionality, then add *drop_dim(...)* at Y, as in Basic Vectorization. Otherwise, do nothing.

(3) outer-to-inner $X \rightarrow Y$. Add *raise_dim(...)* (at X) as in Basic Vectorization.

(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.

4.5.3 Examples with Array Writes

Example 1. Recall that after removal of infeasible edges and redundant phi-nodes, the Aiken's array write example will be (roughly) as follows:

for i in range(N):
1. $\texttt{D}_0 = \phi(\texttt{D}, \texttt{D}_1)$
2. $\texttt{A}_1 = \texttt{A}$; $\texttt{A}_1$[i] = B[i] + 10;
3. $\texttt{B}_1 = \texttt{B}$; $\texttt{B}_1$[i] = $\texttt{A}_1$[i] * $\texttt{D}_0$[i-1];
4. $\texttt{C}_1 = \texttt{C}$; $\texttt{C}_1$[i] = $\texttt{A}_1$[i] * $\texttt{D}_0$[i-1];
5. $\texttt{D}_1 = \texttt{D}_0$; $\texttt{D}_1$[i] = $\texttt{B}_1$[i] * $\texttt{C}_1$[i];

There are no nested loops, thus, array accesses remain as is.

After Phase 1 of Basic vectorization we have the following code:

for i in range(N):
1. $\texttt{D}_0 = \phi(\texttt{D}, \texttt{D}_1)$
2. $\texttt{A}_1 = \texttt{A}$; $\texttt{A}_1[\vec{i}] = \texttt{B}[\vec{i}] + [10, ...]$
3. $\texttt{B}_1 = \texttt{B}$; $\texttt{B}_1[\vec{i}] = \texttt{A}_1[\vec{i}] * \texttt{D}_0[\vec{i} - 1]$
4. $\texttt{C}_1 = \texttt{C}$; $\texttt{C}_1[\vec{i}] = \texttt{A}_1[\vec{i}] * \texttt{D}_0[\vec{i} - 1]$
5. $\texttt{D}_1 = \texttt{D}_0$; $\texttt{D}_1[\vec{i}] = \texttt{B}_1[\vec{i}] * \texttt{C}_1[\vec{i}]$

In Phase 2 we get rid of the arrows on statements 3, 4, and 5 as they are not vectorizable and in Phase 3 we add SIMD appropriately:

1. $\texttt{A}_1 = \texttt{A}$; $\texttt{A}_1[\vec{i}] = ADD\_SIMD(\texttt{B}[\vec{i}], [10, ...])$ {Fully vectorized, size N.}
FOR i=0; i¡N; i++; { MOTION loop }
2. $\texttt{D}_0 = \phi(\texttt{D}, \texttt{D}_1)$
3. $\texttt{B}_1 = \texttt{B}$; $\texttt{B}_1[i] = MUL(\texttt{A}_1[i], \texttt{D}_0[i-1])$
4. $\texttt{C}_1 = \texttt{C}$; $\texttt{C}_1[i] = MUL(\texttt{A}_1[i], \texttt{D}_0[i-1])$
5. $\texttt{D}_1 = \texttt{D}_0$; $\texttt{D}_1[i] = MUL(\texttt{B}_1[i], \texttt{C}_1[i])$

Example 2. Consider MPC-source of Histogram after removal of infeasible edges and redundant phi-nodes, and MUX refinement:

res!1 = []
for i: plaintext in range(0, num_bins):
  res!2 = ϕ(res!1, res!3)
  ...
res!4 = res!2 {Added due to the "mixed" def-use edge.}
for i: plaintext in range(0, num_bins):
  for j: plaintext in range(0, N):
    res!5 = ϕ(res!4, res!7)
    !2!3 = (A[j] == i)
    !3!3 = (res!5[i] + B[j])
    res!6 = res!5; res!6[i] = !3!3
    res!7 = res!6; res!7[i] = MUX(!2!3, res!6[i], res!5[i])

After Phase 1 of Basic Vectorization:

res!1 = []
for i: plaintext in range(0, num_bins):
  res!2 = ϕ(res!1, res!3)
  ...
res!4 = res!2 {Added due to the "mixed" def-use edge. Because the canonical dimensionality of res is 1, we DO NOT drop dimension of res!2.}
A$[i,j]$ = *raise_dim*(A[j],i) {Copying row A num_bin times. Reusing name.}
B$[i,j]$ = *raise_dim*(B[i],j) {Copying column B N times. Reusing name.}
res!4$[i,j]$ = *raise_dim*(res!4[i],j) {Reusing name.}
for i: plaintext in range(0, num_bins):
  for j: plaintext in range(0, N):
    res!5 = ϕ(res!4, res!7) {Do nothing. res!4 has been raised to proper dimension.}
    !2!3$[\vec{i}, \vec{j}]$ = EQ(A$[\vec{i}, \vec{j}]$, [[0,0...], [1,1...],...[num_bins-1,num_bins-1...]])
    !3!3$[\vec{i}, \vec{j}]$ = ADD(res!5$[\vec{i}, \vec{j}]$,B$[\vec{i}, \vec{j}]$)
    res!6 = res!5 {Do nothing, just copy.}
    res!6$[\vec{i}, \vec{j}]$ = !3!3$[\vec{i}, \vec{j}]$
    res!7 = res!6
    res!7$[\vec{i}, \vec{j}]$ = MUX(!2!3$[\vec{i}, \vec{j}]$, res!6$[\vec{i}, \vec{j}]$, res!5$[\vec{i}, \vec{j}]$)

After Phase 2 and Phase 3. The EQ operation is vectorizable across both dimensions, and the rest of the computation is vectorizable across the i-dimension.

...
!2!3$[\vec{i}, \vec{j}]$ = EQ_SIMD(A$[\vec{i}, \vec{j}]$, [[0,0...], [1,1...],...[num_bins-1,num_bins-1...]])
FOR j=0; j¡N; j++; {MOTION loop}
  res!5 = ϕ(res!4, res!7) {Do nothing, just copy.}
  !3!3$[\vec{i}, j]$ = ADD_SIMD(res!5$[\vec{i}, j]$,B$[\vec{i}, j]$)
  res!6 = res!5 {Do nothing, just copy.}
  res!6$[\vec{i}, j]$ = !3!3$[\vec{i}, j]$

res!7 = res!6
res!7[$\vec{i}$, j] = MUX_SIMD(!2!3[$\vec{i}$, j], res!6[$\vec{i}$, j], res!5[$\vec{i}$, j])

5 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.*

6 IMPLEMENTATION AND EVALUATION

7 FUTURE WORK

8 CONCLUSIONS