Compilation and Backend-Independent Vectorization for Multi-Party Computation

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

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

Muhammad Ishaq
ishaqm@purdue.edu
Purdue University
West Lafayette, Indiana

Lindsey Kennard
fire.elemental@gmail.com
STR
Boston, Massachusets

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

Vassilis Zikas
vzikas@purdue.edu
Purdue University
West Lafayette, Indiana

ABSTRACT

ANA: *We need an 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, Muhammad Ishaq, Lindsey Kennard, Ana Milanova, and Vassilis Zikas. 2019. Compilation and Backend-Independent Vectorization for Multi-Party Computation. 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

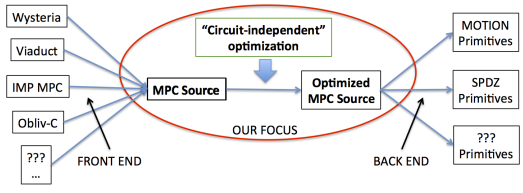
Multi-party computation (MPC) allows N parties $p_1, \ldots, p_N$ to perform a computation on their private inputs securely. Informally, security means that the secure computation protocol computes the correct output (correctness) and it does not leak any information about the individual party inputs, other than what can be deduced from the output (privacy).

MPC theory dates back to the early 1980-ies [**?**, **?**, **?**, **?**]. Long the realm of theoretical cryptography, MPC has seen significant advances in programming technology in recent years. These advances bring MPC closer to practice and wider applicability — MPC technology has been employed in real-world scenarios such as auctions [**?**], biometric identification [**?**], and privacy-preserving machine learning [**?**, **?**]. The goal is to bring the technology to a level where programmers can write *secure* and *efficient* programs without commanding extensive knowledge of cryptographic primitives.

The problem, therefore, is to build a high-level programming language and a compiler, and there has been significant advance in this space, e.g., [**?**, **?**, **?**, **?**, **?**, **?**, **?**] among other work. Current research largely falls at the two ends of the classical compiler: (1) work on *front-end* language design and (2) work on *back-end* protocol implementation. Work on language design focuses on high-level constructs necessary to express multiple parties, computation by different parties, and information flow from one party to another [**?**, **?**]. On the other end, work on protocol implementation focuses on cryptographic foundations and their efficient circuit-level implementation [**?**, **?**, **?**], e.g., implementation of operations (e.g., MUL, ADD) using different sharing protocols (Boolean or Arithmetic GMW [**?**] or Yao's garbled circuits [**?**]), as well as efficient share conversion from one representation to another.

Earlier compilers did both back-end and front-end translation without a specific focus on either, as their aim was to demonstrate applicability of MPC on real-world programming problems. As the field advanced, works have focused more closely on front-end language design (e.g., Wysteria [**?**] and Viaduct [**?**]) or back-end "circuit-level" design and implementation (e.g., MOTION [**?**]).

In this work we focus on an intermediate language and what we call *backend-independent optimizations*, in a close analogy to *machine-independent* optimizations in the classical compiler. The following figure summarizes our key idea:



We emphasize the MPC Source intermediate representation and optimization over MPC Source. As in classical compilers, we envision different front ends (e.g., Wysteria, our front

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

end IMP-MPC) compiling into MPC Source. MPC Source is particularly suitable for optimizations such as protocol mixing [**?**, **?**] and SIMD-vectorization, which takes advantage of amortization at the circuit level. The MPC Source IR exposes the linear structure of MPC programs, which simplifies program analysis; this is in contrast to source, which has if-then-else constructs. In the same time, MPC Source is sufficiently "high-level" to support analyses and optimizations that take into account control and data flow in a specific program. MPC Source is small in size and analysis is tractable, as opposed to analysis over an unrolled circuit [**?**]. Again as in classical compilers, we envision translation of MPC Source (optimized or unoptimized) into MOTION, SPDZ, or other back-end code.

1.1 Our Contribution

In this paper, we develop a compiler framework that takes a Python-like routine and produces MOTION code: we describe (a) the IMP-MPC language, its syntax and semantic restrictions, (b) translation from IMP-MPC into MPC Source, (c) a specific backend-independent optimization: novel SIMD-vectorization on MPC Source, and (d) translation from MPC Source into MOTION code.

We focus on the MOTION framework as our back-end for several reasons. First, it is the state of the art in terms of performance [**?**]. Second, it provides an API over efficient implementation for a wide variety of cryptographic operations in three different protocols — Arithmetic GMW, Boolean GMW, and BMR — which allows for protocol mixing [**?**, **?**], a known backend-independent optimization. Third, MOTION provides API for SIMD-level operations, which amortize cost and lead to significant improvement in memory footprint and throughput [**?**, **?**, **?**]. It also enables MPC Source-level optimizations such SIMD vectorization.

Our second contribution is an analytical model for cost estimation of amortized schedules. Originally, we hoped that optimal scheduling (under our model) was tractable, as the problem appeared simpler than the classical scheduling problem. Unfortunately, we show that optimal scheduling is NP-hard via a reduction to the Shortest Common Supersequence (SCS) problem. Cost estimation is important as it drives not only vectorization but other optimizations such as protocol mixing optimizations and scheduling as well [**?**, **?**].

Our most important contribution is the implementation and evaluation of the compiler framework. We demonstrate expressivity of the source language by running the compiler on Ana: *X* programs; these include classical MPC benchmarks such as PSI and Biometric matching, as well as kMeans, Histogram, and others. Vectorization leads to Ana: *X* x improvement in number of gates on average, Ana: *Y* x speedup in circuit generation time, Ana: *Z* x speedup in the LAN setting, and Ana: *Z* x speedup in the WAN setting using the Boolean GMW protocol. Results form BMR are similar.

Our results emphasize the importance of backend-independent optimizations — vectorization (described in this work) and protocol mixing (tackled in previous works [**?**, **?**, **?**]) are two optimizations readily available at the level of MPC Source. We believe that our work can lead to future work on backend-independent compilation and optimization, bringing in new MPC optimizations and combinations of optimizations, much in the vein of standard compilers, and bring MPC programming technology closer to practice and wider applicability.

1.2 Outline

The rest of the paper is organized as follows. §**??** presents an overview of the compiler. §**??** describes our model for cost estimation and argues NP-hardness of optimal scheduling and §**??** describes the compiler. §**??** presents implementation and evaluation. §**??** discusses related work and §**??** concludes.

2 OVERVIEW

2.1 Source

As a running example, consider Biometric matching, a standard MPC benchmark. An intuitive (and naive) implementation is as shown in Listing **??**(a). Array `C` is the feature vector of `D` features that we wish to match and `S` is the database of `N` size-`D` vectors that we match against.

Our compiler takes essentially standard IMP Ana: *Need citation here.* syntax and imposes certain semantic restrictions. (We detail the restrictions in the following sections.) The programmer writes an iterative program and annotate certain inputs and outputs as *shared*. In the example arrays `C` and `S` are shared, meaning that they store shares, however, the array sizes `D` and `N` respectively are plaintext. The code iterates over the entries in the database and computes the Euclidean distance (its square actually) of the current entry `S[i]` and `C`. The program returns the index of the vector that gives the best match plus the corresponding sum of squares.

2.2 MPC Source and Cost of Schedule

Our compiler generates an intermediate representation, MPC Source. MPC Source for Biometric Matching is shown in Listing **??**(b). First, MPC Source is an SSA form. Second, it is linear. The conditional in lines 13-14 in IMP Source turns into the linear code in lines 12-16 in MPC Source. The test turns into a CMP operation (`t = CMP(sum!3,min_sum!2)`), followed by the true-branch sequence, followed by the MUX operations. The first MUX operation selects the value of `min_sum`: if `t` is true, then `min_sum` gets the value of the second multiplexer argument, `min_sum!3`, otherwise it takes the value of the third argument, `min_sum!2`. Third, MPC Source is a special form of SSA. The SSA ϕ-nodes at the if-then-else (lines 13-15) turn into MUX operations, while the ϕ-nodes at for loops turn into what we call pseudo PHI nodes with a straightforward semantics.

We turn to our analytical model to compute the *cost* of the 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 $ND(2\alpha + \beta) + N(3\alpha)$.

```
1  def biometric(C: shared[list[int]], D: int,
2          S: shared[list[int]], N: int) ->
3          shared[tuple[int,int]]:
4      min_sum = MAX_INT
5      min_idx = 0
6      for i in range(N):
7          sum = 0
8          for j in range(D):
9              # d = S[i,j] - C[j]
10             d = S[i * D + j] - C[j]
11             p = d * d
12             sum = sum + p
13         if sum < min_sum:
14             min_sum = sum
15             min_idx = i
16     return (min_sum, min_idx)
```

(a) IMP Source

```
1  min_sum!1 = MAX_INT
2  min_idx!1 = 0
3  for i in range(0, N):
4      min_sum!2 = PHI(min_sum!1, min_sum!4)
5      min_idx!2 = PHI(min_idx!1, min_idx!4)
6      sum!2 = 0
7      for j in range(0, D):
8          sum!3 = PHI(sum!2, sum!4)
9          d = SUB(S[((i * D) + j)],C[j])
10         p = MUL(d,d)
11         sum!4 = ADD(sum!3,p)
12     t = CMP(sum!3,min_sum!2)
13     min_sum!3 = sum!3
14     min_idx!3 = i
15     min_sum!4 = MUX(t, min_sum!3, min_sum!2)
16     min_idx!4 = MUX(t, min_idx!3, min_idx!2)
17 return (min_sum!2, min_idx!2)
```

(b) MPC Source

```
1  min_sum!1 = MAX_INT
2  min_idx!1 = 0
3  # S^ is same as S. C^ replicates C N times:
4  S^ = raise_dim(S, ((i * D) + j), (i:N,j:D)) #S^[i,j] = S[i,j]
5  C^ = raise_dim(C, j, (i:N,j:D)) #C^[i,j] = C[j]
6
7  sum!2[I] = [0,..,0]
8  # computes _all_ "at once"
9  d[I,J] = SUB_SIMD(S^[I,J],C^[I,J])
10 p[I,J] = MUL_SIMD(d[I,J],d[I,J])
11
12 for j in range(0, D):
13     # sum!2[I], sum!3[I], sum!4[I] are size-N vectors
14     # computes N intermediate sums "at once"
15     sum!3[I] = PHI(sum!2[I], sum!4[I])
16     sum!4[I] = ADD_SIMD(sum!3[I],p[I,j])
17
18 min_idx!3[I] = [0,1,...N-1]
19 for i in range(0, N):
20     min_sum!2 = PHI(min_sum!1, min_sum!4)
21     t[i] = CMP(sum!3[i],min_sum!2)
22     min_sum!4 = MUX(t[i], sum!3[i], min_sum!2)
23 for i in range(0, N):
24     min_idx!2 = PHI(min_idx!1, min_idx!4)
25     min_idx!4 = MUX(t[i], min_idx!3[i], min_idx!2)
26 return (min_sum!2, min_idx!2)
```

(c) Optimized MPC Source

Table 1: Biometric Matching: From IMP Source to MPC Source to Optimized MPC Source.

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 a naive, iterative MPC program into an unintuitive vectorized one.

2.3 Vectorized MPC Source and Cost of Schedule

Our compiler produces the vectorized program shown in Listing **??**(c). Note that this is still our intermediate representation, Optimized MPC Source. Subsequently, the compiler turns this code into MOTION variables, loops and SIMD primitives, which MOTION then uses to generate the circuit.

The compiler determines that SUB and MUL in "naive" MPC Source (lines 9 and 10 in (b)) can be fully vectorized into the SIMD SUB and MUL in optimized MPC Source (lines 9 and 10 in (c)). Notation `p[I,J]` denotes a 2-dimensional array with fully vectorized dimensions. The computation of sum (line 11 in (b)) is sequential across the j-dimension, but it is parallel across the i-dimension. The loop in lines 12-16 in (c) illustrates; here `p[I,j]` refers to the j-th column in `p`. Unfortunately, CMP and MUX remain sequential.

In MPC backends, executing n operations "at once" in a single SIMD operation costs a lot less than executing those n operations one by one. This is particularly important when there is communication (i.e., in remote), since many 1-bit values are sent at once rather than sequentially. We elaborate on the cost model in the following section but for now consider that each operation has a *fixed* portion (does benefits from amortization) and a *variable* portion (does not benefit from amortization): $\alpha = \alpha_{fix} + \alpha_{var}$. This gives rise to the following formula for amortized cost: $f(n) = \alpha_{fix} + n\alpha_{var}$, as opposed to unamortized cost $g(n) = n\alpha_{fix} + n\alpha_{var}$. (We extend the same reasoning to β-instructions.)

Thus, the fixed cost of the vectorized program amounts to $2\alpha_{fix} + D\beta_{fix} + N(3\alpha_{fix})$. (The variable cost is the same in both the vectorized and un-vectorized programs.) 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, $2\alpha_{fix} + D\beta_{fix} + N(3\alpha_{fix}) << ND(2\alpha_{fix} + \beta_{fix}) + N3\alpha_{fix}$. Empirically, we observe 10x to 50x improvement over un-vectorized Biometric Matching in

circuit generation time and setup time and 5x to 30x improvement in online time. Additionally, the un-vectorized version runs out of memory for $N = 256$, while the vectorized one runs with the standard maximal input size $N = 4,096$. ANA: *Edit numbers with final experiments.*

3 ANALYTICAL MODEL

ANA: *We need an intro to section here.*

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) In MPC frameworks, executing n operations "at once" in a single SIMD operation costs a lot less than executing those n operations one by one. Following Amdahl's law ANA: *citation!*, we write $\alpha = \frac{1}{s}p\alpha + (1-p)\alpha$, where p is the fraction of execution time that benefits from amortization and $(1-p)$ is the fraction that does not, and s is the available resource. Thus, $n\alpha = \frac{n}{s}p\alpha + n(1-p)\alpha$. For the purpose of the model we assume that s is large enough and the term $\frac{n}{s}p\alpha$ amounts to a *fixed cost* incurred regardless of whether n is $10,000$ or just 1. (This models the cost of sending a packet from party A to party B for example.) The bottom line is that amortized execution of n operations is $f(n) = \alpha_{fix} + n\alpha_{var}$ in contrast to unamortized execution $g(n) = n\alpha_{fix} + n\alpha_{var}$ and, of course, $\alpha_{fix} << n\alpha_{fix}$.
(4) MPC instructions scheduled in parallel benefit from amortization *only if* they are the same instruction. Given our previous assumption, 2 MUL instructions can be amortized in a single SIMD instruction that costs $\alpha_{fix} + 2\alpha_{var}$, however a MUL and a MUX instruction still cost $2\alpha_{fix} + 2\alpha_{var}$ even when scheduled "in parallel".

3.2 Problem Statement

As mentioned earlier, at the lowest level, we have two types of MPC instructions (also called *gates* in similar works) 1) local/non-interactive instruction (i.e. an addition instruction A) and 2) remote/interactive instruction (i.e. a multiplication instruction M).

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

(1) All P's consist of MPC instructions of the same kind, e.g., all MUL, or ADD, etc.
(2) Def-use dependencies of the graph $G(V, E)$ should be preserved i.e. if instructions $S_i, S_j, i < j$ form a def-use i.e. an edge exists from S_i to S_j in G, then they can only be mapped to $P_{i'}, P_{j'}$ such that $i' < j'$.

Correctness. Correctness of P is guaranteed by definition. Preserving def-use *dependencies* means the computed function remains the same in both S and P.

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). Recall that 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:

The cost of schedule S is

$$cost(S) = \sum_{i=1}^{n} cost(S_i) \tag{1}$$

where $cost(S_i) = \alpha$ or β. Similarly, the cost of schedule P is

$$cost(P) = \sum_{i=1}^{m} cost(P_i) \tag{2}$$

Each P_i may contain multiple instructions, and $cost(P_i)$ is amortized. Thus, according to our model $cost(P_i) = \alpha_{fix} + |P_i|\alpha_{var}$ if P_i stores α-instructions, or $cost(P_i) = \beta_{fix} + |P_i|\beta_{var}$ if it stores β-instructions. Our goal is to construct a parallel schedule P that reduces the program cost (when compared to cost of S), possibly an optimal schedule. Originally we hoped that the problem is simper and computation of the optimal schedule is tractable. Unfortunately, the optimal schedule turns out to be NP-hard via a reduction to the Shortest Common Supersequence problem.

Note that we consider a linearized MPC schedule S above for ease of exposition only. In our tool-chain we use an MPC Source control flow graph (CFG) $G'(V', E')$ along with def-use graph $G(V, E)$ to construct P.

3.3 Scheduling is NP-hard

To prove that optimal scheduling is an NP-Hard problem, we consider the following convenient representation. An MPC program is represented as a set of sequences $S = \{S_1, \ldots, S_n\}$. Each element $S_i \in S$ is a tuple. The items of the tuple S_i are operations, i.e. A or M instructions, that have to be executed in order (operations depend on previous operations i.e. $S_i[j], j > 1$ depends $S_i[j-1]$). However, the sequences $S_i, 1 \leq i \leq n$ themselves, can overlap each other in any way i.e. distinct sequences can be executed in parallel. We argue that an MPC program can be transformed into such collection of sequences by traversing the circuit for each pair of input and output values. ISHAQ: *I am not sure how this will be done, see* **??**

[1] this is not strictly true, but assuming it, e.g. as in [**?**, **?**, **?**], helps simplify the problem.

As an example, consider the MPC program consisting of the following three sequences, all made up of two distinct α-instructions M_1 and M_2. The right arrow indicates a def-use *dependence*, meaning that the source node must execute before the target node:

(1) $M_1 \rightarrow M_2 \rightarrow M_1$
(2) $M_1 \rightarrow M_1 \rightarrow M_1$
(3) $M_2 \rightarrow M_1 \rightarrow M_2$

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

$$M_1(1), M_1(2)\ ;\ M_1(2)\ ;\ M_2(1), M_2(3)\ ;\ M_1(1), M_1(2), M_1(3)\ ;\ M_2(3)$$

The parentheses above indicate the sequence where the instruction comes from: (1), (2), or (3). Cost of schedule P is computed using **??** above and it amounts to $5\alpha_{fix} + 9\alpha_{var}$.

The problem of finding a schedule P with a minimal $cost(P)$ is shown to be NP-Hard problem, as it can be reduced to the problem of finding a *shortest common supersequence*, a known NP-Hard problem[**?**, **?**]. 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 sequence and k is the number of total number of sequences. We can immediately see that the optimal schedule is the shortest schedule, since the shortest schedule minimizes the fixed cost while the variable cost remains the same.

To formalize the reduction, suppose P is a schedule with minimal cost (computed by a black-box algorithm). Clearly, P, which now is a sequence of M_1 and M_2 nodes, 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. The cost of $cost(P) = L\alpha_{fix} + N\alpha_{var}$ where L is the length of P and N is the total number of instructions across all sequences. Now suppose, there exist a shorter common supersequence P' of length L'. $cost(P') < cost(P)$ since $L'\alpha_{var} + N\alpha_{var} < L\alpha_{var} + N\alpha_{var}$, however, this is a contradiction to the assumption that P has the lowest cost. □

4 COMPILER FRAMEWORK

Fig. **??** presents an overview of our compiler. This section describes the front-end phases of the compiler and Section §**??** describes our vectorization algorithm. ANA: *We'll need a section on MOTION code gen too.*

4.1 Syntax and Semantic Restrictions

Source syntax is essentially standard IMP syntax as shown below:

e ::= e *op* e | e | `x` | `const` | `A[`e`]`	*expression*
s ::= s; s |	*sequence*
`x` = e | `A[`e`]` = e |	*assignment stmt*
`for` i `in range(`I`)` : s |	*for stmt*
`if` e `then` s `else` s	*if stmt*

The syntax allows for array accesses, arbitrarily nested loops, and if-then-else control flow. Expressions are typed $q\ \tau$, where qualifier q and base type τ are:

τ ::= `int` | `bool` | `list[int]` | `list[bool]`	*base types*
q ::= `shared` | `plain`	*qualifiers*

The type system is mostly standard, and in our experience, a sweet spot between simplicity and expressivity. The `shared` qualifier denotes shared values, i.e., ones shared among the parties and computed upon under secure computation protocols vs. `plain` which denotes plaintext values. Shared lists denote shared elements, though the bounds of the list are plaintext. ANA: *add subtyping*

Our compiler imposes certain semantic restrictions that it enforces throughout the various phased of compilation. 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) Loops are of the form `0 <= i < I` and bounds are fixed at compile time. It is a standard restriction in MPC 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. (This restriction can be easily lifted.)
(3) Array subscrpts are plaintext values. The typing rule for array access `A[`e`]` ANA: *???* 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 can be an arbitrary function of the indices of the enclosing loops.
(4) ANA: *Add rules for logical ops*
(5) ANA: *Add rules for arithmetic ops*
(6) ANA: *Add rules for MUX*

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.

4.2 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.

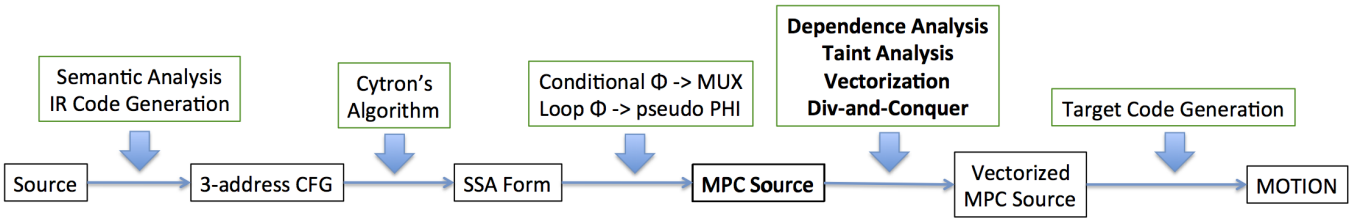


Figure 1: Compiler Framework. ANA: *Change figure since we are not doing div-and-conquer.*

(3) **3-address CFG conversion**: BENJAMIN: *TODO: Is this a good amount of detail?* ANA: *This is good amount of detail for the Tech report. In the actual paper we'll shorten a lot and add pointers to the Tech report.* Convert the restricted-syntax AST to a three-address control-flow graph. To do this, first, add an empty basic block to the CFG and mark it as current. Next, for each statement in the restricted AST's function body, process the statement. Statements can either be for-loops, if-statements, or assignments. Rules for processing each kind of statement are given below:

(a) **For-loops**: Create new basic blocks for the loop condition (the condition block), the loop body (the body block), and the code after the loop (the after-block). Insert a jump from the end of the current block to the condition block. Then, mark the condition block as the current block. Insert a for-instruction at the end of the current block with the loop counter variable and bounds from the AST. Next, add an edge from the current block to the after-block labeled "FALSE" and an edge from the current block to the body block labeled "TRUE". Then, set the body block to be the current block and process all statements in the AST's loop body. Finally, insert a jump to the condition block and set the after-block as current.

(b) **If-statements**: Create new basic blocks for the "then" statements of the if-statement (the then-block), the "else" statements of the if-statement (the else-block), and the code after the if-statement (the after-block). At the end of the current block, insert a conditional jump to the then-block or else-block depending on the if-statement condition in the AST. Next, mark the then-block as current, process all then-statements, and add a jump to the after-block. Similarly, mark the else-block as current, process all else-statements, and add a jump to the after-block. Finally, set the after-block to be the current block, and give it a "merge condition" property equal to the condition of the if-statement.

(c) **Assignments**: In the restricted-syntax AST, the left-hand side of assignments can be a variable or an array subscript. If it is an array subscript such as `A[i] = x`, change the statement to `A = Update(A, i, x)`. If the statement is not already three-address code, for each sub-expression in the right-hand side of the assignment, insert an assignment to a temporary variable.

(4) **SSA conversion**: Convert the 3-address CFG to SSA with Cytron's algorithm.

4.3 MUX Nodes and Pseudo ϕ-nodes

Once the compiler converts the code to SSA, it transforms ϕ-nodes that correspond to if-statements into MUX nodes. From the 3-address CFG conversion step, ϕ-nodes corresponding to if-statements will be in a basic block with the "merge condition" property. For example, if $X_3 = \phi(X_1, X_2)$ is in a block with merge condition C, the compiler transforms it into $X_3 = MUX(C, X_1, X_2)$. Next, the compiler runs the dead code elimination algorithm from Cytron's SSA paper.

Then, the control-flow graph is *linearized* into MPC Source, which has loops but no if-then-else-statements. This means that both branches of all if-statements are executed, and the MUX nodes determine whether to use results from the then-block or from the else-block. The compiler linearizes the control-flow graph with a variation of breadth-first search. Blocks with the "merge condition" property are only considered the second time they are visited, since that will be after both branches of the if-statement are visited. Each time the compiler visits a block, it adds the block's statements to the MPC source. If the block ends in a for-instruction, the compiler recursively converts the body and code after the loop to MPC source and adds the for-loop and code after the loop to the main MPC source. If the block does not end in a for-instruction, the compiler recursively converts all successor branches to MPC source and appends these to the main MPC source.

Now, the remaining ϕ-nodes in MPC source are *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.

4.4 Dependence Analysis

4.4.1 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.

4.4.2 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).

4.5 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:

- 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 scalar expansion 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) $\rightarrow$ $stmt$(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 $stmt'$
        (more precisely, right after $stmt'$)
      inner-to-outer: add $Y'_n[i,j,k] = drop\_dim(Y_n)$ at $stmt$
        (more precisely, in loop of $stmt$ right after loop of $stmt'$)
  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, replacing A_1 with A_0 or A_k appropriately
  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)
```

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

Figure 2: MPC Source Syntax

```
// 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)
```

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., $\mathsf{x}[i,j,k] = ...$, and base statement $s2 \in a$ uses x, e.g., $... = ...\mathsf{x}[i,j,k]$ 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 $\mathsf{x}[\underline{i},\underline{j},\underline{k}]$ to statement $s2$ that uses $\mathsf{x}[\underline{i},\underline{j},\underline{k}]$ 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 $\mathsf{x}[0,1,2]$ and a statement that uses $\mathsf{x}[0,1,2]$.

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: $\mathtt{A}_m[f(i,j,k)] = ...$, and use: $... = \mathtt{A}_n[f'(i,j,k)]$ 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) $\mathtt{A}_m[f(i,j,k)] = ...$ at some iteration $\underline{j}$ references the *same* array element as a use $... = \mathtt{A}_n[f'(i,j,k)]$ 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 (Phase 2). 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 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.

5.5.3 Changes to Basic Vectorization

In addition to the changes for the handling of target-less phi-nodes, Basic Vectorization has to handle 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,...)`, 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. While there are variations, the most intuitive implementation will perform Basic Vectorization Phase 1 as is, inserting *raise_dim* and *drop_dim* in the same way. However, the implementation of raise dimension and drop dimension will be adapted because the dimension cannot be raised or dropped to a dimension lower than the canonical one. Consider a def-use edge $X \rightarrow Y$ for an array `A`.

(1) same-level $X \rightarrow Y$. Do nothing, propagate the array, which happens to be of the right dimension.
(2) inner-to-outer $X \rightarrow Y$ triggers the addition of *drop_dim*. However, the dimensionality cannot be dropped below the canonical dimensionality of the array. E.g., if the dimensionality of the loop enclosure X is already at the canonical one, then *drop_dim* has no effect.
(3) outer-to-inner $X \rightarrow Y$ riggers *raise_dim*. Again, if the dimensionality of the loop enclosure of Y is smaller or same as the canonical dimensionality of the array, then it has no effect, otherwise, if dimensionality is greater than the canonical dimensionality, *raise_dim*(...) (at X) is the same 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.

If the use of the array is a read $\mathtt{A}[f(i,j,k)]$ different than a canonical read $\mathtt{A}[i,j,k]$, then we need to add a reshape operation as all arrays are $\mathtt{A}[i,j,k]$. It can be added after raise_dim/drop_dim or incorporated in these operations. The bulk of the change is in Phase 2 of Basic Vectorization as outlined earlier.

5.5.4 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 phi-node
   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)
   res2' = drop_dim(res2)
res1'' = drop_dim(res1)
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! Several things happen. (1) Everything can be vectorized along the i dimension. (2) We remove the target-less PHI node, however, we must update uses of `res1` appropriately: the use at raise_dim goes to the first argument of the PHI function and the use at drop_dim goes to the second argument.

```
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)) // replacing res1 with res, 1st arg
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)
   equiv. to res3 = res2; res3[I,j] = tmp3[I,j]
res2' = drop_dim(res2)
res1 = drop_dim(res2') // replacing with res2', 2nd arg. NOOP
return res1
```

6 IMPLEMENTATION AND EVALUATION

7 RELATED WORK

MPC languages and compilers. Languages and compilers for secure computation have seen significant attention and advances in recent years. The early MPC compilers Fairplay [**?**], and Sharemind [**?**] were followed by PICCO [**?**], Obliv-C [**?**], TinyGarble [**?**], Wystiria [**?**], and others. A new generation of MPC compilers includes SPDZ/SCALE-MAMBA/MP-SPDZ [**?**] and the ABY/HyCC/MOTION [**?**, **?**, **?**] frameworks. These two families are the state-of-the art and are actively developed. Another recent development is Viaduct, a functional language and compiler that supports a range of secure computation frameworks, including MPC and ZKP. Hastings et el. present a review of compiler frameworks [**?**].

While each of these languages and compilers brings in new ideas and advances, none addresses the problem of "circuit independent" intermediate representation and optimization. We envision a classical compiler structure: (1) a Wysteria,

Viaduct, Obliv-C, or IMP Source front end, including rich type systems and AST-level semantic analysis, compile into the MPC Source IR, (2) MPC Source-level optimizations take place, followed by (3) back-end compilers into circuits. Our focus is at the intermediate level.

Many works focus on the implementation of MPC protocols exposing an API to the programmer. For example, the ABY/-MOTION line of compiler frameworks provides a library of MPC primitives; the programmer writes MPC programs in C++ on top of the library. These back ends implement different protocols and allow for mixing, but notably, they leave it to the programer to assign different protocols to different parts of the computation and perform share conversion accordingly. In addition, MOTION provides SIMD primitives, which allows for efficient execution of MPC operations, but again, using SIMD primitives is the responsibility of the programer. There is interest in frameworks for automatic mixing, e.g., [**?**, **?**, **?**].

Other works, e.g., Obliv-C [**?**], Wystiria [**?**] and Viaduct [**?**] focus on higher-level language design, particularly information-flow systems that restrict flow between secure and insecure parts of the program.

Classical HPC compilers. Automatic vectorization is a longstanding problem in high-performance computing (HPC). There are thousands of works in this area reflecting over 40 years of research. We presented a vectorization algorithm for MPC Source, essentially extending classical loop vectorization [**?**]. In HPC vectorization, conditional control flow presents a challenge — one cannot estimate the cost of a schedule or vectorize branches in a straightforward manner — in contrast to MPC Source vectorization. We view Karrenberg's work on Whole function vectorization [**?**] as most closely related to ours — it linearizes the program and vectorizes both branches of a conditional applying masking to avoid execution of the branch-not-taken code, and selection (similar to MUX) to select the correct value based on the result of the condition at runtime. The problem is that masking and selection, or more generally, handling control predicates [**?**, **?**], can lead to *slowdown*.

We argue that vectorization over linear MPC Source is a different problem, one that warrants a new look, while drawing from results in HPC. Since both branches of the conditional and the multiplexer *always* execute, not only can we apply aggressive vectorization on linear code, but (perhaps more importantly) we can also build analytical models that accurately predict execution time. These models in turn would drive optimizations such as vectorization, protocol mixing, and others. Vectorization meshes in with those additional optimizations in non-trivial ways.

Furthermore, extensions of classical loop vectorization with array writes, arbitrary indexing, including non-affine indexing, and interaction with SSA are non-trivial and present novel challenges and opportunities for contribution. Polyhedral parallelization [**?**] considers a higher-level source (typically AST) representation, while our work takes advantage of linear MPC Source and SSA form. The work by Karrenberg [**?**] is rare in that space, in the sense that it considers vectorization over SSA form, which has similarities to MPC Source. We consider different array representation, notion of dependence, and reasoning about dependence, which we conjecture is more suitable for MPC Source. Buscher [**?**] considers SIMD-vectorization at the level of source code, which then combines with circuit-level optimizations in the TinyGarble compiler. He proposes using an off-the-shelf polyhedral compiler, however, application is limited to only two routines, essentially just inner product and euclidian distance; it is unclear how effective the off-the-shelf compiler is. In contrast, we consider vectorization at the level of MPC Source separating "backend-independent" vectorization and circuit-level amortization (done by MOTION). We apply our compiler on a wide range of routines.

8 CONCLUSION AND FUTURE WORK